

Code Generators for MySQL Plugins and User Defined Functions

Hartmut Holzgraefe
<hartmut@mysql.com>

Why UDFs / Plugins?

- to add functionality not available elsewhere
- to build upon functionality already available in a C/C++ library
- to get better performance by using native code

Why a code generator?

- less steep learning curve
- faster development, focus on actual functionality
- API changes can be dealt with in one central place
- automate boring tasks

Aspects of building a UDF or Plugin

- Requirements & Dependencies
- Build Infrastructure
- API Infrastructure
- **Implementation of actual functionality**
- Testing
- Documentation
- Packaging & Distribution

Generator Infrastructure

Generator Requirements

- PHP 5.x
- PEAR
(PHP Extension & Application Repository)
- PEAR Packages:
 - CodeGen & CodeGen_MySQL
 - CodeGen_MySQL_UDF
 - CodeGen_MySQL_Plugin
 - CodeGen_Drizzle

Build Infrastructure

Requirements for building:

- MySQL Development Packages
- sometimes full MySQL Server Source
- C/C++ compiler and “make”
- autoconf, automake, libtool
- or on Windows: CMake (w.i.p.)

Getting started on Debian/Ubuntu

- `apt-get build-dep mysql-server`
- `apt-get install php5-cli pear`
- `pear install CodeGen_MySQL_UDF`
- `pear install CodeGen_MySQL_Plugin`
- `pear install CodeGen_Drizzle`

Build Infrastructure #2

Things taken care of:

- `configure` script
- `Makefile`
- `make` targets for:
 - `make all`
 - `make test`
 - `make clean`
 - `make dist / distcheck`
 - `make html / pdf`

Getting Started

A minimal UDF specification file:

```
<?xml version="1.0"?>  
<udf name="dummy">  
</udf>
```

and code is generated from it using:

```
udf-gen dummy.xml
```

Generated Files

```
twice
|-- AUTHORS
|-- CMakeLists.txt
|-- ChangeLog
|-- INSTALL
|-- Makefile.am
|-- NEWS
|-- README
|-- configure.ac
|-- twice.c
|-- m4/
|   |-- ax_compare_version.m4
|   |-- mysql.m4
|-- manual.xml
|-- mysql.m4
|-- tests/
|   |-- create_functions.inc
|   |-- drop_functions.inc
|   |-- r/
|   |-- t/
|   |-- test.sh
|-- udf_twice.h
```

A 'real' Example

```
<?xml version="1.0"?>
<udf name="twice">
  <function name="twice"
            returns="int">
    <param name="val" type="int"/>
    <code>
      return val * 2;
    </code>
  </function>
</udf>
```

Compiling the generated Code

```
$ udf-gen twice.xml
```

```
$ cd example
```

```
$ ./configure --with-mysql=...  
                --libdir=...
```

```
$ make
```

Instaling the generated Function

- Install the UDF library
- `sudo make install`

- Register the function
- `CREATE FUNCTION twice
RETURNS INTEGER
SONAME "twice.so";`

- Test the function
- `SELECT twice(3); -- result: 6`

Aggregate functions

```
<function name="my_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>
```

Automated Tests

Automated tests that use the `mysqltest` tool can be generated using a simplified syntax:

```
<test name="twice">
  <code>
    SELECT twice(3);
  </code>
  <result>
twice(3);
6
  </result>
</test>
```

Invoking the Tests

- Both the `mysqltest` and `mysqld` binaries have to be in `$PATH`
- invoke tests with `make test`

```
$ PATH=$PATH/usr/local/mysql/bin  
$ PATH=$PATH/usr/local/mysql/libexec  
$ export PATH
```

```
$ make test  
*** testing ***
```

```
[twice] ok
```

Packing a Source Release

- To pack the generated source for distribution use either `make dist` or `make distcheck`
- `make dist` creates a source tarball
- `make distcheck` creates a source tarball, unpacks it in a temporary directory and checks that building it works (e.g. no files left out when packing)
- Documentation can be generated using `make html` and `make pdf` if you have `docbook-utils` installed

Things not shown here

- Author, Copyright and License Information
- Library and Header File Dependencies
- Adding C/C++ code files
- Lex/Flex and Yacc/Bison support
- Windows Builds using CMake
- DEBUG functions

Plugins

- allow to extend the server using shared libraries and well defined registration APIs
- currently supported plugin types are
 - “daemons”
 - storage engines
 - fulltext parsers
 - INFORMATION_SCHEMA tables
 - Audit
 - Replication
 - Authentication

Building a Plugin

Generating a plugin works like generating a UDF:

```
mysql-plugin-gen my-plugin.xml
```

To configure the plugin use

```
configure --with-mysql=...  
          --libdir=...
```

or if the plugin requires server internals

```
configure --with-mysql-src=...  
          --libdir=...
```

Installing a Plugin

- first install the plugin library
`sudo make install`
- then for each plugin defined in the library do
`INSTALL PLUGIN "plugin_name"
SONAME "plugin_library.so";`
- to uninstall use
`UNINSTALL PLUGIN;`

A Daemon Plugin

```
<plugin name="daemon_example">
  <maintainer>
    <name>Hartmut Holzgraefe</name>
    <email>hartmut@mysql.com</email>
  </maintainer>

  <license>GPL</license>

  <release>
    <version>0.0.1</version>
    <date>2008-04-16</date>
    <state>alpha</state>
    <notes>just an experiment</notes>
  </release>

  <daemon name="dummy">
    <summary>minimal example plugin</summary>
    <init>fprintf(stderr, "daemon started\n");</init>
    <deinit>fprintf(stderr, "daemon stopped\n");</deinit>
  </daemon>
</plugin>
```

Adding server variables

```
<statusvar  
    type="int"  
    name="THE_ANSWER"  
    init="42"  
>
```

```
SHOW STATUS LIKE 'THE_ANSWER';
```

Variable name	Value
THE_ANSWER	42

An INFORMATION_SCHEMA Plugin

```
<deps language="c++" > <src/> <header="sql/mysql_priv.h"/> </deps>
<infoschema name="is_dummy">
  <summary>minimal information_schema plugin</summary>
  <statusvar type="int"      name="dummy_count"      init="0"/>
  <field name="FOO_INT"      type="LONG"/>
  <field name="FOO_STRING"   type="STRING"/>
</code>
<![CDATA
  TABLE* table= tables->table;

  table->field[FIELD_FOO_INT]->store(23);
  table->field[FIELD_FOO_STRING]->store("foobar", 7, system_charset_info);
  schema_table_store_record(thd, table);

  table->field[FIELD_FOO_INT]->store(42);
  table->field[FIELD_FOO_STRING]->store("barfoo", 7, system_charset_info);
  schema_table_store_record(thd, table);

  dummy_count++;
  return 0;
]]>
</code>
</infoschema>
```

A Fulltext Parser Plugin

```
<fulltext name="dummy">
  <summary>minimal example plugin</summary>

  <parser>
    <code>
<![CDATA[
  char *doc;
  int result;

  doc = strdup(param->doc, param->length);
  result = param->mysql_parse(param, param->doc, param->length);
  free(doc);

  return result;
]]>
    </code>
  </parser>
</fulltext>
```

Things not covered (here) ...

- Storage engine plugins
- Fulltext plugin details
- Status variable datatypes and callbacks
- Command line options
- header file and library dependencies
- ...

Conclusion

The generator tools save time:

- less code to write
- less things to learn
- no debugging of API infrastructure code
- on API changes just re-generate with current generator

Conclusion #2

Things already implemented using these tools:

- UDF_REGEX
- UDF_ORA
- I_S.MASTER_STATUS and I_S.SLAVE_STATUS
- a Fulltext Parser for gzip compressed data

Online Resources

- Trac/Wiki
 - <http://codegenerators.php-baustelle.de/>
- PEAR Packages
 - <http://pear.php.net/package/CodeGen>
 - http://pear.php.net/package/CodeGen_MySQL
 - http://pear.php.net/package/CodeGen_MySQL_UDF
 - http://pear.php.net/package/CodeGen_MySQL_Plugin
- UDF projects
 - <http://udf-ora.php-baustelle.de/>
 - <http://udf-regexp.php-baustelle.de/>
 - ...