PERCONA LIVE ONLINE

MAY 12 - 13th

2021

Creating MySQL User-Defined Functions in C++ Has Never Been Easier

Yura Sorokin
Principal Software Engineer
Percona

User-Defined Functions (UDFs)

It is a way to extend MySQL with a new function that works like a native (built-in) MySQL function; i.e., by using a UDF you can create native code to be executed on the server from inside MySQL.

User-Defined Functions (UDFs)

1. Build a shared object (**.so**) with the function logic
2. Register this function in MySQL Server via **CREATE FUNCTION ... SONAME ...**
3. Use it, say, in a **SELECT** statement
4. Unregister this function in MySQL Server via **DROP FUNCTION**

Oversimplified Example

Implement **`native_reverse()`** function that would accept exactly one string argument and return a copy of it with characters in reverse order.

```
SELECT native_reverse('abcd');
-> 'dcba'
SELECT native_reverse(NULL);
-> NULL
```

Spoiler alert: there is already **REVERSE()** in MySQL
https://dev.mysql.com/doc/refman/8.0/en/string-functions.html#function_reverse

UDF Reference

Official MySQL documentation
"6.2 Adding a User-Defined Function"
https://dev.mysql.com/doc/extending-mysql/8.0/en/adding-udf.html

Build "**libstring_udfs.so**" with these symbols:

```
extern "C" … native_reverse(…);
extern "C" … native_reverse_init(…);
extern "C" … native_reverse_deinit(…);
```

"**_init**()" and "**_deinit()**" functions are optional

The Main Function Declaration

Functions having **STRING** return type must have the following signature

```
extern "C" char *native_reverse( // 1
  UDF_INIT *initid,       // 2
  UDF_ARGS *args,         // 3
  char *result,           // 4
  unsigned long *length, // 5
  char *is_null,          // 6
  char *error             // 7
);
```

The Main Function Definition

```
extern "C" char *native_reverse(...) {
  if (args->arg_count != 1) { // 1
    *error = 1;      // 2
    return nullptr; // 2
  }
  ...
```

The Main Function Definition

```
...
auto arg_ptr = args->args[0]; // 1
if (arg_ptr == nullptr) {
  *error = 0;      // 2
  *is_null = 1;    // 2
  return nullptr; // 2
}
std::size_t arg_length =
  args->lengths[0]; // 3
...
```

The Main Function Definition

```
  using buffer_type = std::vector<char>;
  buffer_type buffer(arg_length + 1,
                     '\0'); // 1
  std::reverse_copy(
    arg_ptr,
    arg_ptr + arg_length,
    std::begin(buffer)); // 2
  *error = 0;            // 3
  *is_null = 0;          // 3
  *length = arg_length; // 3
  return buffer.data(); // 4
}
```

UDF Calling Sequences for Simple Functions

- **native_reverse_init(UDF_INIT *initid, ...)** - performs any required setup, such as argument checking or memory allocation
- **native_reverse(UDF_INIT *initid, ...)** - called once for each row
- **native_reverse_deinit(UDF_INIT *initid, ...)** - performs any required cleanup

UDF_INIT to the Rescue

```
typedef struct UDF_INIT {
  ...
  char *ptr; // free pointer for
             // function data
  ...
} UDF_INIT;
```

Custom UDF Context

Defining a context class that would be used to hold intermediate data between **xxx_init() / xxx() / xxx_deinit()** calls

```
struct native_reverse_context {
  using buffer_type = std::vector<char>;
  buffer_type buffer;
};
```

UDF init Function

Creating an instance of the context inside **xxx_init()**

```
extern "C" bool native_reverse_init(
  UDF_INIT *initid,
  UDF_ARGS *args,
  char *message) {
  auto context = // 1
    new native_reverse_context;
  initid->ptr = // 2
    reinterpret_cast<char *>(context);
  return false; // 3
}
```

UDF Main Function

Extracting context from '**initid**' inside **xxx()**

```
char *native_reverse(...) {
  ...
  auto context = reinterpret_cast< // 1
    native_reverse_context*>(initid->ptr);
  context->buffer.resize( // 2
    arg_length + 1,
    '\0');
  std::reverse_copy( // 3
    arg_ptr,
    arg_ptr + arg_length,
    std::begin(context->buffer));
```

UDF Main Function

```
  ...
  *error = 0;             // 4
  *is_null = 0;           // 4
  *length = arg_length; // 4
  return context->buffer.data(); // 5
}
```

UDF deinit Function

Deleting the instance of the context from '**initid**' inside **xxx_deinit()**

```
extern "C" void native_reverse_deinit(
  UDF_INIT *initid) {
  delete reinterpret_cast< // 1
    native_reverse_context*>(
      initid->ptr);
}
```

Summary for Standard Approach

A lot of boilerplate code for such a trivial task.

- Defining a context class with shared data and storage buffer for the result
- Defining an **xxx_init()** function just to create an instance of the context
- Ugly casts in the main **xxx()** function to access context and its members
- Ugly casts for non-string argument extractions
- Defining **xxx_deinit()** function just to delete the context

Something Better From Percona

In **Percona Server for MySQL 8.0.22-13** we introduced "**C++ UDF wrappers**"

Blueprint:
https://jira.percona.com/browse/PS-7348

Github Pull Request:
https://github.com/percona/percona-server/pull/4059

Header-Only C++ Library

"**C++ UDF wrappers**" is just a set of C++ headers

```
#include <mysqlpp/common_types.hpp>
#include <mysqlpp/udf_context.hpp>
#include <mysqlpp/udf_exception.hpp>
#include <mysqlpp/udf_traits.hpp>
#include <mysqlpp/udf_wrappers.hpp>
```

No More Boilerplate

The same '**native_reverse()**' UDF done right

```
class native_reverse_impl { // 1
 public:
  native_reverse_impl( // 2
    mysqlpp::udf_context &ctx);
  mysqlpp::udf_result_t<STRING_RESULT> // 3
  calculate(
    const mysqlpp::udf_context &ctx);
};
DECLARE_STRING_UDF(
  native_reverse_impl,
  native_reverse)
```

Implementing Constructor

```
native_reverse_impl::native_reverse_impl(
  mysqlpp::udf_context &ctx) {
  if (ctx.get_number_of_args() != 1) // 1
    throw mysqlpp::udf_exception( // 2
      "function requires exactly one "
      "argument");
}
```

Implementing Main Function

```
mysqlpp::udf_result_t<STRING_RESULT>
native_reverse_impl::calculate(
  const mysqlpp::udf_context &ctx) {
  auto arg_sv = // 1
    ctx.get_arg<STRING_RESULT>(0);

  if (arg_sv.data() == nullptr) // 2
    return {};

  return std::string( // 3
    std::crbegin(arg_sv),
    std::crend(arg_sv));
}
```

Summary for C++ UDF Wrappers Approach

Spend your time on implementing logic only - let the framework do all the dirty work behind the scenes for you.

- Define an implementation class for your function
- Define a constructor for this class that accepts single argument of type '**mysqlpp::udf_context**' by non-const reference
- Define a '**calculate()**' method in this class that accepts single argument of type '**mysqlpp::udf_context**' by const reference and returns '**mysqlpp::udf_result_t<XXX_RESULT>**'
- Instantiate all the boilerplate code with a macro '**DECLARE_XXX_UDF()**'

Summary for C++ UDF Wrappers Approach

- '**mysqlpp::udf_context**' is a much more convenient and safer way to access function arguments('**args**') and shared data ('**initid**')
- No need to worry about adding custom buffers for values being returned - the framework will automatically allocate them for you depending on the function return type
- Use exceptions ('**mysqlpp::udf_exception**') for error handling - they will be automatically translated into MySQL errors with provided messages

C++ UDF Wrappers TODOs

- Add support for aggregates
- Consider switching to 8.0 component model which would allow function auto-registration
- Add support for character sets for both string arguments and string return values

Q&A

It’s time for your questions!

THANK YOU!

PERCONA
LIVEONLINE
MAY 12 - 13th
2021