Other languages: 繁體中文

Database - DbClient

DbClient Object Construction

There are two ways to construct a DbClient object. One is through the static method of the DbClient class. You can see the definition in the DbClient.h header file, as follows:

```
#if USE_POSTGRESQL
    static std::shared_ptr<DbClient> newPgClient(const std::string
&connInfo, const size_t connNum);
#endif
#if USE_MYSQL
    static std::shared_ptr<DbClient> newMysqlClient(const std::string
&connInfo, const size_t connNum);
#endif
```

Use the above interface to get the smart pointer of the DbClient implementation object. The parameter connInfo is a connection string. Set a series of connection parameters in the form of key=value. For details, please refer to the comments in the header file. The parameter connNum is the number of database connections of DbClient, which has a key impact on concurrency. Please set it according to the actual situation.

The object obtained by the above method, the user has to find a way to persist it, such as putting it in some global container. **Creating a temporary object and then releasing it after use is a very unrecommended solution** for the following reasons:

- This will waste time creating connections and disconnections, increasing system latency;
- The interface is also a non-blocking interface. That is to say, when the user gets the DbClient object, the connection managed by the it has not been established yet. The framework does not (intentionally) provide a callback interface for successful connection establishment. Do you still have to sleep before starting the query?? This is contrary to the original intention of the asynchronous framework.

Therefore, DbClient objects should be built at the beginning of the program and held and used throughout the life time. Obviously, this work can be done entirely by the framework. So the drogon framework provides the second build method, which is built by configuration file or the `createDbClient()` method. For the configuration method of the configuration file, see db_clients.

When needed, the DbClient smart pointer is obtained through the interface of the framework. The interface is as follows:

```
orm::DbClientPtr getDbClient(const std::string &name = "default");
```

The parameter name is the value of the name configuration option in the configuration file to distinguish multiple different DbClient objects of the same application. The connections managed by DbClient are always reconnected, so users don't need to care about the connection status. They are almost always connected. **Note**: This method cannot be called before running app.run(), otherwise the user will get an empty shared_ptr.

Execution Interface

DbClient provides several different interfaces to users, as listed below:

```
/// Asynchronous mothod
template <
        typename FUNCTION1,
        typename FUNCTION2,
        typename... Arguments>
void execSqlAsync(const std::string &sql,
                  FUNCTION1 &&rCallback,
                  FUNCTION2 &&exceptCallback,
                  Arguments &&... args) noexcept;

/// Asynchronous mothod by 'future'
template <typename... Arguments>
std::future<const Result> execSqlAsyncFuture(const std::string &sql,
                                             Arguments &&... args)
noexcept;

/// Synchronous method
template <typename... Arguments>
const Result execSqlSync(const std::string &sql,
                         Arguments &&... args) noexcept(false);

/// Streaming-type method
internal::SqlBinder operator<<(const std::string &sql);
```

Since the number and type of binding parameters cannot be predetermined, these methods are function templates.

The properties of these methods are shown in the following table:

Methods	Synchronous/Asynchronous	Blocking/Non-blocking	Exception
void execSqlAsync	Asynchronous	Non-blocking	Will not throw an exception
std::future<const Result> execSqlAsyncFuture	Asynchronous	Block when calling the get method of the future	May throw an exception when calling the get method of the future

Methods	Synchronous/Asynchronous	Blocking/Non-blocking	Exception
const Result execSqlSync	Synchronous	Blocking	May throw an exception
internal::SqlBinder operator<<	Asynchronous	Default non-blocking	Will not throw an exception

You may be confused about the combination of asynchronous and blocking. In general, the synchronization method involving network IO is blocking, and the asynchronous method is non-blocking. However, the asynchronous method can also work in blocking mode, meaning that this method will Block until the callback function has finished executing. When the asynchronous method of DbClient works in blocking mode, the callback function will be executed in the thread of the caller, and then the method will return.

If your application involves high-concurrency scenarios, please use asynchronous non-blocking methods. If it is in a low concurrent scene (such as a network device management page), you can choose synchronization methods for convenience and intuitiveness.

- **execSqlAsync**

```
template <typename FUNCTION1,
          typename FUNCTION2,
          typename... Arguments>
void execSqlAsync(const std::string &sql,
                  FUNCTION1 &&rCallback,
                  FUNCTION2 &&exceptCallback,
                  Arguments &&... args) noexcept;
```

This is the most commonly used asynchronous interface, working in non-blocking mode;

The parameter `sql` is a string of sql statements. If there are placeholders for binding parameters, use the placeholder rules of the corresponding database. For example, PostgreSQL placeholders are $1, $2 ..., while MySQL placeholders are `?` without any numbers.

The indefinite parameter `args` represents the bound parameter, which can be zero or more. The number of parameters is the same as the number of placeholders in the sql statement. The types can be the following:

- Integer type: can be an integer of various word lengths, and should match the database field type;
- Floating point type: can be `float` or `double`, should match the database field type;
- String type: can be `std::string` or `const char[]`, corresponding to the string type of the database or other types that can be represented by strings;
- Date type: `trantor::Date` type, corresponding to the database date, datetime, timestamp types.
- Binary type: `std::vector<char>` type, corresponding to PostgreSQL's byte type or Mysql's blob type;

These parameters can be left or right, can be variables or literal constants, and users are free to use them.

The parameters rCallback and exceptCallback represent the result callback function and the exception callback function, respectively, which have a fixed definition, as follows:

- The result callback function: the call type is void (const Result &), various callable objects conforming to this call type, std::function, lambda, etc. can be passed as parameters;
- Exception callback function: the call type is void (const DrogonDbException &), which can pass various callable objects that are consistent with this call type;

After the execution of sql is successful, the execution result is wrapped by the Result class and passed to the user through the result callback function; if there is any exception in the sql execution, the exception callback function is executed, and the user can obtain the exception information from the DrogonDbException object.

Let us give an example:

```
auto clientPtr = drogon::app().getDbClient();
clientPtr->execSqlAsync("select * from users where org_name=$1",
                            [](const drogon::orm::Result &result) {
                                std::cout << result.size() << " rows
selected!" << std::endl;
                                int i = 0;
                                for (auto row : result)
                                {
                                    std::cout << i++ << ": user name
is " << row["user_name"].as<std::string>() << std::endl;
                                }
                            },
                            [](const DrogonDbException &e) {
                                std::cerr << "error:" <<
e.base().what() << std::endl;
                            },
                            "default");
```

From the example we can see that the Result object is a std standard compatible container, support iterators, you can get the object of each row through the range loop. the various interfaces of Result, Row and Field objects, please refer to the source code.

The DrogonDbException class is the base class for all database exceptions. Please refer to the comments in the source code.

- **execSqlAsyncFuture**

```
template <typename... Arguments>
std::future<const Result> execSqlAsyncFuture(const std::string &sql,
                                             Arguments &&... args)
noexcept;
```

The asynchronous future interface omits the two callback parameters of the previous interface. Calling this interface will immediately return a future object. The user must call the get() method of the future object to get the returned result. The exception is obtained through the try/catch mechanism. if the get() method isn't in the `try/catch`, and there is no `try/catch` in the entire call stack, the program will exit when the sql execution exception occurs.

For example:

```
auto f = clientPtr->execSqlAsyncFuture("select * from users where
org_name=$1",
                                       "default");
try
{
    auto result = f.get(); // Block until we get the result or catch
the exception;
    std::cout << result.size() << " rows selected!" << std::endl;
    int i = 0;
    for (auto row : result)
    {
        std::cout << i++ << ": user name is " <<
row["user_name"].as<std::string>() << std::endl;
    }
}
catch (const DrogonDbException &e)
{
    std::cerr << "error:" << e.base().what() << std::endl;
}
```

- **execSqlSync**

```
template <typename... Arguments>
const Result execSqlSync(const std::string &sql,
                         Arguments &&... args) noexcept(false);
```

Synchronous interface is the most simple and intuitive, the input parameters are sql string and bound parameters, return a Result object, the call will block the current thread, and throw an exception when an error occurs, so also pay attention to catch exception with `try/catch`.

E.g:

```
try
{
    auto result = clientPtr->execSqlSync("update users set
user_name=$1 where user_id=$2",
                                         "test",
```

```
                                            1); // Block until we get the
result or catch the exception;
    std::cout << result.affectedRows() << " rows updated!" <<
std::endl;
}
catch (const DrogonDbException &e)
{
    std::cerr << "error:" << e.base().what() << std::endl;
}
```

- **operator<<**

```
internal::SqlBinder operator<<(const std::string &sql);
```

The streaming interface is special. It inputs the sql statement and parameters in turn through the `<<` operator, and specifies the result callback function and the exception callback function through the `>>` operator. For example, the previous example of selecting, using the streaming interface is Look like this:

```
*clientPtr  << "select * from users where org_name=$1"
            << "default"
            >> [](const drogon::orm::Result &result)
                {
                    std::cout << result.size() << " rows selected!" <<
std::endl;
                    int i = 0;
                    for (auto row : result)
                    {
                        std::cout << i++ << ": user name is " <<
row["user_name"].as<std::string>() << std::endl;
                    }
                }
            >> [](const DrogonDbException &e)
                {
                    std::cerr << "error:" << e.base().what() <<
std::endl;
                };
```

This usage is completely equivalent to the first asynchronous non-blocking interface, and which interface is used depends on the user's usage habits. If you want it to work in blocking mode, you can use `<<` to enter a `Mode::Blocking` parameter, which is not described here.

In addition, the streaming interface has a special usage. Using a special result callback, the framework can pass the result to the user row by row. The call type of this callback is as follows:

```
void (bool,Arguments...);
```

When the first bool parameter is true, it means that the result is a empty row, that is, all the results have been returned, this is the last callback; Behind is a series of parameters, corresponding to the value of each column of a row of records, the framework will do type conversion, of course, the user should also pay attention to the type of matching. These types can be const-type lvalue references, or rvalue references, and of course value types.

Let's rewrite the previous example with this callback:

```
int i = 0;
*clientPtr  << "select user_name, user_id from users where
org_name=$1"
            << "default"
            >> [&i](bool isNull, const std::string &name, int64_t id)
                  {
                  if (!isNull)
                      std::cout << i++ << ": user name is " << name
<< ", user id is " << id << std::endl;
                  else
                      std::cout << i << " rows selected!" <<
std::endl;
                  }
            >> [](const DrogonDbException &e)
                {
                    std::cerr << "error:" << e.base().what() <<
std::endl;
                };
```

It can be seen that the values of the user_name and user_id fields in the select statement are respectively assigned to the name and id variables in the callback function, and the user does not need to handle these conversions by themselves, which obviously provides a certain convenience, and the user can use it flexibly.

> **Note: It is important to emphasize that in asynchronous programming the user must pay attention to the variable i in the above example. The user must ensure that the variable i is valid when the callback occurs because it is caught by the reference. The callback will be called in another thread, and the current context may have failed when the callback occurred. Programmers typically use smart pointers to hold temporarily created variables and then capture them through callbacks to ensure the validity of the variables.**

Summary

Each DbClient object has one or multiple its own EventLoop threads controlling the database connection IO, accepting the request via an asynchronous or synchronous interface, and returning the result via a callback function.

Blocking interfaces of DbClient only block the caller thread, as long as the caller thread is not the EventLoop thread, it will not affect the normal operation of the EventLoop thread. When the callback function is called,

the program inside the callback is run on the EventLoop thread. Therefore, do not perform any blocking operations within the callback, otherwise it will affect the concurrency performance of database read and write. Anyone familiar with non-blocking I/O programming should understand this constraint.

Next: Transaction