Other languages: 繁體中文

Database - Transactions

> **Transactions** are an important feature of relational databases, and Drogon provides transaction support with the `Transaction` class.

Objects of the `Transaction` class are created by `DbClient`, and many transaction-related operations are performed automatically:

- At the beginning of the `Transaction` object creation, the begin statement is automatically executed to `start` the transaction;
- When the `Transaction` object is destructed, the `commit` statement is automatically executed to end the transaction;
- If there is an exception that causes the transaction to fail, the `rollback` statement is automatically executed to roll back the transaction;
- If the transaction has been rolled back, then the sql statement will return an exception (throw an exception or perform an exception callback);

Transaction Creation

The method of transaction creation is provided by `DbClient` as follows:

```
std::shared_ptr<Transaction> newTransaction(const std::function<void(bool)>
&commitCallback = std::function<void(bool)>())
```

This interface is very simple, it returns a smart pointer to a `Transaction` object. Obviously, when the smart pointer loses all the holders and destructs the transaction object, the transaction ends. The parameter `commitCallback` is used to return whether the transaction commit is successful. It should be noted that this callback is only used to indicate whether the `commit` command is successful. If the transaction is automatically or manually rolled back during execution, the `callback` will not be executed. Generally, the `commit` command will succeed， the bool type parameter of this callback is true. Only some special cases, such as the connection disconnection during the commit process, will cause the `commitCallback` to notify the user that the commit fails, at this time, the state of the transaction on the server is not certain, the user needs to deal with this situation specially. Of course, considering that this situation rarely occurs, with non-critical services the user can choose to ignore this event by ignoring the `commitCallback` parameter when creating the transaction (The default empty callback will be passed to the newTransaction method).

The transaction must monopolize the database connection. Therefore, during transaction creation, `DbClient` needs to select an idle connection from its own connection pool and hand it over to transaction object management. This has a problem. If all connections in the `DbClient` are executing sql or other transactions, the interface will block until there is an idle connection.

The framework also provides an asynchronous interface for creating transactions, as follows:

```
void newTransactionAsync(const std::function<void(const
std::shared_ptr<Transaction> &)> &callback);
```

This interface returns the transaction object through the callback function, does not block the current thread, and ensures high concurrency of the application. Users can use it or the synchronous version according to the actual situation.

Transaction Interface

The `Transaction` interface is almost identical to `DbClient`, except for the following two differences:

- `Transaction` provides a `rollback()` interface that allows the user to roll back the transaction under any circumstances. Sometimes, the transaction has been automatically rolled back, and then calling the `rollback()` interface has no negative impact, so explicitly using the rollback() interface is a good strategy to at least ensure that it is not committed incorrectly.
- The user cannot call the transaction's `newTransaction()` interface, which is easy to understand. Although the database has the concept of a sub-transaction, the framework does not currently support it.

In fact, `Transaction` is designed as a subclass of `DbClient`, in order to maintain the consistency of these interfaces, and at the same time, it also creates convenient conditions for the use of ORM.

The framework currently does not provide an interface to control transaction isolation levels, that is, the isolation level is the default level of the current database service.

Transaction Life Cycle

The smart pointer of the transaction object is held by the user. When it has unexecuted sql, the framework will hold it, so don't worry about the transaction object being destructed when there is still unexecuted sql. In addition, the transaction object smart pointer is often caught and used in the result callback of one of its interfaces. This is the normal way to use, don't worry that the circular reference will cause the transaction object to never be destroyed, because the framework will help the user break the circular reference automatically.

One Example

For the simplest example, suppose there is a task table from which the user selects an unprocessed task and changes it to the state being processed. To prevent concurrent race conditions, we use the `Transaction` class, the program is as follows:

```
{
    auto transPtr = clientPtr->newTransaction();
    transPtr->execSqlAsync( "select * from tasks where status=$1 for update
order by time",
                            "none",
                            [=](const Result &r) {
                                if (r.size() > 0)
                                {
```

```
                                        std::cout << "Got a task!" <<
std::endl;
                                        *transPtr << "update tasks set
status=$1 where task_id=$2"
                                                  << "handling"
                                                  << r[0]
["task_id"].as<int64_t>()
                                                  >> [](const Result &r)
                                                     {
                                                        std::cout <<
"Updated!";
                                                        ... do something about
the task;
                                                     }
                                                  >> [](const DrogonDbException
&e)
                                                     {
                                                        std::cerr << "err:" <<
e.base().what() << std::end;
                                                     };
                                    }
                                    else
                                    {
                                        std::cout << "No new tasks found!" <<
std::endl;
                                    }
                                },
                                [](const DrogonDbException &e) {
                                    std::cerr << "err:" << e.base().what() <<
std::end;
                                });
}
```

In this case, select for update is used to avoid concurrent modifications. The update statement is completed in the result callback of the select statement. The outermost braces are used to limit the scope of the transPtr so that it can be destroyed in time after the execution of sql to end the transaction.

Next: ORM