Other languages: 繁體中文

Coroutines

Drogon supports C++ coroutines starting from version 1.4. They provide a way to flatten the control flow of asynchronous calls, i.e. escaping the callback hell. With it, asynchronous programming becomes as easy as synchronous programming.

Terminology

This page isn't intended to explain what is a coroutine nor how it works. But to show how to use coroutines in drogon. The usual vocabulary tends get messy as subroutines (functions) uses the same terminology as coroutine does, yet they have slightly different meanings. The fact that C++ coroutines can act as if they are functions doesn't help either. To reduce confusion, we'll use the terminology that follows - it is by no means perfect, but it is good enough.

Coroutine is a function that can suspend execution then resume.
Return means a function finishing execution and fiving a return value to its caller. Or a coroutine generating a *resumable* object; which can be used to resume the coroutine.
Yielding is when a coroutine generates a result for the caller.
co-return means a coroutine yields and then exits.
(co-)awaiting means the thread is waiting for a coroutine to yield. The framework is free to use the thread for other purpose while awaiting.

Enabling coroutines

The coroutine feature in Drogon is header-only. This means the application can use coroutines even if Drogon is built without coroutine support. How to enable coroutines depends on the compiler used. In GCC >= 10 it can be enabled by setting `-std=c++20 -fcoroutines` while with MSVC (tested on MSVC 19.25) it can be enabled with `/std:c++latest` and `/await` must not be set.

Note that Drogon's implementation of coroutines won't work on clang (as of clang 12.0). GCC 11 enables coroutines by default when C++20 is enabled. And though GCC 10 does compile coroutines, it contains a compiler bug causing nested coroutine frames not being released; leading to potential memory leak.

Using coroutines

Each and every coroutine in drogon is suffixed with `Coro`. E,g. `db->execSqlSync()` becomes `db->execSqlCoro()`. `client->sendRequest()` becomes `client->sendRequestCoro()`, so on and so forth. All coroutines return an *awaitable* object. Then `co_await` on the object results in a value. The framework is free to use the thread to process IO and tasks when it's awaiting results to arrive - that's the beauty of coroutines. The code looks synchronous; but it's in fact asynchronously.

For example, querying the number of users that exist in the database:

```
app.registerHandler("/num_users",
    [](HttpRequestPtr req, std::function<void(const HttpResponsePtr&)>
```

```
callback) -> Task<>
    //                                  Must mark the return type as a
_resumable_ ^^^
{
    auto sql = app().getDbClient();
    try
    {
        auto result = co_await sql->execSqlCoro("SELECT COUNT(*) FROM
users;");
        size_t num_users = result[0][0].as<size_t>();
        auto resp = HttpResponse::newHttpResponse();
        resp->setBody(std::to_string(num_users));
        callback(resp);
    }
    catch(const DrogonDbException &err)
    {
        // Exception works as sync interfaces.
        auto resp = HttpResponse::newHttpResponse();
        resp->setBody(err.base().what());
        callback(resp);
    }
    // No need to return anything! This is a coroutine that yields `void`.
Which is
    // indicated by the return type of Task<void>
    co_return; // If want to (not required), use co_return
}
```

Notice a few important points:

1. Any handler that calls a coroutine MUST return a *resumable*
    - Turning the handler itself into a coroutine
2. `co_return` replaces `return` in a coroutine
3. Most parameters are passed by value

A *resumable* is an object following the coroutine standard. Don't worry too much about the details. Just know that if you want the coroutine to yield something typed `T`, then the return type will be `Task<T>`.

Passing most parameters by value is a direct consequence of coroutines being asynchronous. It's impossible to track when a reference goes out of scope as the object may destruct while the coroutine is waiting. Or the reference may live on another thread. Thus, it may destruct while the coroutine is executing.

It makes sense to not have the callback but to use the straightforward `co_return`. Which is supported, but may cause up to 8% of throughput under certain conditions. Please consider the performance drop and whether it's too great for the use case. Again, the same example:

```
app.registerHandler("/num_users",
    [](HttpRequestPtr req) -> Task<HttpResponsePtr>)
    //          Now returning a response ^^^
{
    auto sql = app().getDbClient();
    try
```

```
    {
        auto result = co_await sql->execSqlCoro("SELECT COUNT(*) FROM
users;");
        size_t num_users = result[0][0].as<size_t>();
        auto resp = HttpResponse::newHttpResponse();
        resp->setBody(std::to_string(num_users));
        co_return resp;
    }
    catch(const DrogonDbException &err)
    {
        // Exception works as sync interfaces.
        auto resp = HttpResponse::newHttpResponse();
        resp->setBody(err.base().what());
        co_return resp;
    }
}
```

Calling coroutines from websocket controllers isn't supported yet. Feel free to open an issue if you need this feature.

Common pitfalls

There are some common pitfalls you may encounter when using coroutines.

- **Launching coroutines with lambda capture from a function**

  Lambda captures and coroutines have separate lifetimes. A coroutine lives until the coroutine frame is destructed. While lambdas commonly destruct right after being called. Thus, due to the asynchronous nature of coroutines, the coroutines's lifetime can be much longer than the lambda, for example in SQL execution. The lambda destructs right after awaiting for SQL to complete (and returns to the event loop to process other events), while the coroutine frame is awaiting SQL. Thus the lambda will have been destructed when SQL has finished.

  Instead of

  ```
  app().getLoop()->queueInLoop([num] -> AsyncTask {
      auto db = app().getDbClient();
      co_await db->execSqlCoro("DELETE FROM customers WHERE last_login <
  CURRENT_TIMESTAMP - INTERVAL $1 DAY". std::to_string(num));
      // The lambda object, thus captures destruct right at awaiting.
  They are destructed at this point
      LOG_INFO << "Remove old customers that have no activity for more
  than " << num << "days"; // use-after-free
  });
  // BAD, This will crash
  ```

  Drogon provides `async_func` that wraps around the lambda to ensure its lifetime

  ```
  ```

```
app().getLoop()->queueInLoop(async_func([num] -> Task<void> {
//                              ^^^^^^^^^^^^^^^^^^^^^^^^^^ wrap with
async_func and return a Task<>
    auto db = app().getDbClient();
    co_await db->execSqlCoro("DELETE FROM customers WHERE last_login <
CURRENT_TIMESTAMP - INTERVAL $1 DAY". std::to_string(num));
    LOG_INFO << "Remove old customers that have no activity for more
than " << num << "days";
}));
// Good
```

- **Passing/capturing references into coroutines from function**

  It's a good practice in C++ to pass objects by reference to reduce unnecessary copy. However passing by reference into a coroutine from a function commonly causes issues. This is caused by the the coroutine is in fact asynchronous and can have a much longer lifetime compared to a regular function. For example, the following code crashes

```
void removeCustomers(const std::string& customer_id)
{
    async_run([&customer_id] {
        //      ^^^^ DO NOT pass/capture objects by reference into a
coroutine
        // Unless you are sure the object has a longer lifetime than
the coroutine

        auto db = app().getDbClient();
        co_await db->execSqlCoro("DELETE FROM customers WHERE
customer_id = $1", customer_id);
        // `customer_id` goes out of scope right at awaiting SQL.
Crashes here
        co_await db->execSqlCoro("DELETE FROM orders WHERE customer_id
= $1", customer_id);
    }
}
```

  However passing objects as reference from a coroutine is considered a good practice

```
Task<> removeCustomers(const std::string& customer_id)
{
    auto db = app().getDbClient();
    co_await db->execSqlCoro("DELETE FROM customers WHERE customer_id
= $1", customer_id);
    co_await db->execSqlCoro("DELETE FROM orders WHERE customer_id =
$1", customer_id);
}

Task<> findUnwantedCustomers()
```

```
{
    auto db = app().getDbClient();
    auto list = co_await db->execSqlCoro("SELECT customer_id from
customers "
        "WHERE customer_score < 5;");
    for(const auto& customer : list)
        co_await
removeCustomers(customer["customer_id"].as<std::string>());
        //                                ^^^^^^^^^^^^^^^^^
        // This is perfectly fine and preferred although it's a const
reference
        // since we are calling it from a coroutine
}
```

Next: Redis