協程（Coroutines）

原文：ENG-17-Coroutines.md

Drogon 自 1.4 版起支援 C++ 協程，可將非同步呼叫流程扁平化，擺脫 callback hell，讓非同步程式設計如同步般簡單。

名詞定義

本章不解釋協程原理，只說明如何在 drogon 使用。常見術語容易混淆，因為副程式（函式）與協程用詞相近但意義略異，C++ 協程又可當函式用。為減少混亂，以下採用簡化定義：

協程（Coroutine）：可暫停並恢復執行的函式。

Return：函式執行結束並回傳值，或協程產生 *resumable* 物件，可用於恢復協程。

Yield：協程產生結果給呼叫者。

co-return：協程 yield 並結束。

(co-)await：執行緒等待協程 yield，框架可於等待時重用執行緒處理其他任務。

啟用協程

Drogon 協程功能為 header-only，無論是否支援協程皆可用。啟用方式依編譯器而異：GCC >= 10 設定 `-std=c++20 -fcoroutines`，MSVC（測試於 19.25）設 `/std:c++latest` 且不可設 `/await`。

注意：Drogon 協程於 clang（12.0）尚不支援。GCC 11 啟用 C++20 時預設支援協程。GCC 10 雖可編譯協程，但有巢狀協程 frame 未釋放的 bug，可能導致記憶體洩漏。

協程用法

drogon 所有協程皆以 `Coro` 結尾，如 `db->execSqlSync()` 變成 `db->execSqlCoro()`，`client->sendRequest()` 變成 `client->sendRequestCoro()`。所有協程回傳 *awaitable* 物件，`co_await` 該物件可取得結果。框架可於等待結果時重用執行緒處理 IO 與任務，程式看似同步，實則非同步。

例如查詢資料庫使用者數：

```
app.registerHandler("/num_users",
    [](HttpRequestPtr req, std::function<void(const HttpResponsePtr&)>
callback) -> Task<>
    // 必須標註回傳型別為 _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)
    {
        // 例外處理同同步介面
        auto resp = HttpResponse::newHttpResponse();
        resp->setBody(err.base().what());
        callback(resp);
    }
    // 不需回傳任何值！此協程 yield `void`，由 Task<void> 型別表示
    co_return; // 可用 co_return（非必要）
}
```

重點：

1. 呼叫協程的 handler 必須回傳 *resumable*，即 handler 本身也是協程
2. 協程用 co_return 取代 return
3. 參數多以值傳遞

resumable 為符合協程標準的物件。若 yield 型別為 `T`，則回傳型別為 `Task<T>`。

多數參數值傳遞是因協程為非同步，無法追蹤 reference 何時離開 scope，物件可能於協程等待時析構，或 reference 存於其他執行緒，導致協程執行時物件已析構。

可不使用 callback，直接 co_return，雖支援但某些情況下 throughput 可能降 8%。請評估效能是否可接受。範例：

```
app.registerHandler("/num_users",
    [](HttpRequestPtr req) -> Task<HttpResponsePtr>)
    // 現在回傳 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)
    {
        auto resp = HttpResponse::newHttpResponse();
        resp->setBody(err.base().what());
        co_return resp;
    }
}
```

WebSocket controller 尚不支援協程，如有需求請提出 issue。

常見陷阱

使用協程時常見陷阱如下：

- **以 lambda capture 啟動協程**

  Lambda capture 與協程生命週期不同。協程存活至 frame 析構，lambda 常於呼叫後即析構。因協程非同步，協程生命週期可能遠長於 lambda，例如 SQL 執行時，lambda 於 await SQL 完成後即析構（回 event loop 處理其他事件），而協程 frame 仍在等待 SQL，故 SQL 完成時 lambda 已析構。

  錯誤範例：

```
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));
    // lambda 物件於 await 時即析構
    LOG_INFO << "Remove old customers that have no activity for more
than " << num << "days"; // use-after-free
});
// 錯誤，會 crash
```

  Drogon 提供 `async_func` 包裝 lambda，確保生命週期：

```
app().getLoop()->queueInLoop(async_func([num] -> Task<void> {
// 用 async_func 包裝並回傳 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";
}));
// 正確
```

- **由函式傳遞/捕獲 reference 進協程**

  C++ 常以 reference 傳遞物件以減少複製，但由函式傳 reference 進協程常出問題。因協程非同步，生命週期遠長於一般函式。錯誤範例：

```
void removeCustomers(const std::string& customer_id)
{
    async_run([&customer_id] {
        // 不要由函式傳/capture reference 進協程，除非確定物件生命週期比協程長

        auto db = app().getDbClient();
        co_await db->execSqlCoro("DELETE FROM customers WHERE
customer_id = $1", customer_id);
```

```
        // customer_id 於 await SQL 時即離開 scope，crash
        co_await db->execSqlCoro("DELETE FROM orders WHERE customer_id
= $1", customer_id);
    });
}
```

但由協程傳 reference 則屬良好做法：

```
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>());
        // 這樣傳 const reference 沒問題，因為是由協程呼叫
}
```

下一步: Redis