測試框架

原文：ENG-19-Testing-Framework.md

DrogonTest 是 drogon 內建的極簡測試框架，支援非同步與同步測試。Drogon 本身的單元測試與整合測試皆採用此框架，亦可用於 drogon 應用程式測試。語法參考 GTest 與 Catch2。

你不必一定用 DrogonTest，可選擇習慣的框架，但 DrogonTest 是一個選項。

基本測試

以下為簡單範例：有一個同步函式計算 1 到 n 的總和，需測試其正確性。

```
// 告訴 DrogonTest 產生 `test::run()`，僅在主檔案定義
#define DROGON_TEST_MAIN
#include <drogon/drogon_test.h>

int sum_all(int n)
{
    int result = 1;
    for(int i=2;i<n;i++) result += i;
    return result;
}

DROGON_TEST(Sum)
{
    CHECK(sum_all(1) == 1);
    CHECK(sum_all(2) == 3);
    CHECK(sum_all(3) == 6);
}

int main(int argc, char** argv)
{
    return drogon::test::run(argc, argv);
}
```

編譯執行...雖通過但有明顯 bug，sum_all(0) 應為 0，可加進測試：

```
DROGON_TEST(Sum)
{
    CHECK(sum_all(0) == 0);
    CHECK(sum_all(1) == 1);
    CHECK(sum_all(2) == 3);
    CHECK(sum_all(3) == 6);
}
```

此時測試失敗：

```
In test case Sum
↳ /path/to/your/test/main.cc:47  FAILED:
  CHECK(sum_all(0) == 0)
With expansion
  1 == 0
```

框架會顯示失敗測試與實際值，方便立即定位問題。解法如下：

```
int sum_all(int n)
{
    int result = 0;
    for(int i=1;i<n;i++) result += i;
    return result;
}
```

斷言類型

DrogonTest 提供多種斷言與動作。基本 `CHECK()` 檢查表達式是否為真，否則輸出至主控台。`CHECK_THROWS()` 檢查表達式是否丟出例外，未丟出則輸出。`REQUIRE()` 檢查表達式是否為真，否則 return，後續測試不執行。

失敗時動作/表達式	為真	丟出例外	未丟出例外	丟出特定型別
無動作	CHECK	CHECK_THROWS	CHECK_NOTHROW	CHECK_THROWS_AS
return	REQUIRE	REQUIRE_THROWS	REQUIRE_NOTHROW	REQUIRE_THROWS_AS
co_return	CO_REQUIRE	CO_REQUIRE_THROWS	CO_REQUIRE_NOTHROW	CO_REQUIRE_THROWS_AS
終止程序	MANDATE	MANDATE_THROWS	MANDATE_NOTHROW	MANDATE_THROWS_AS

實用範例：測試檔案內容是否正確，若開檔失敗則無需後續測試，可用 `REQUIRE` 簡化程式：

```
DROGON_TEST(TestContent)
{
    std::ifstream in("data.txt");
    REQUIRE(in.is_open());
    // 不用
    // CHECK(in.is_open() == true);
    // if(in.is_open() == false)
    //    return;

    ...
}
```

`CO_REQUIRE` 用於協程，功能同 REQUIRE。`MANDATE` 適用於操作失敗且改變不可回復全域狀態時，唯一合理做法是終止測試。

非同步測試

Drogon 為非同步 Web 框架，DrogonTest 亦支援非同步函式測試。DrogonTest 透過 `TEST_CTX` 追蹤測試 context，請以值 capture。例如測試遠端 API 是否成功回傳 JSON：

```
DROGON_TEST(RemoteAPITest)
{
    auto client = HttpClient::newHttpClient("http://localhost:8848");
    auto req = HttpRequest::newHttpRequest();
    req->setPath("/");
    client->sendRequest(req, [TEST_CTX](ReqResult res, const HttpResponsePtr&
resp) {
        // 若請求未達 server 或回傳錯誤
        REQUIRE(res == ReqResult::Ok);
        REQUIRE(resp != nullptr);

        CHECK(resp->getStatusCode() == k200OK);
        CHECK(resp->contentType() == CT_APPLICATION_JSON);
    });
}
```

協程需包在 `AsyncTask` 或用 `sync_wait` 呼叫，因測試框架為 C++14/17 相容，無原生協程支援。

```
DROGON_TEST(RemoteAPITestCoro)
{
    auto api_test = [TEST_CTX]() {
        auto client = HttpClient::newHttpClient("http://localhost:8848");
        auto req = HttpRequest::newHttpRequest();
        req->setPath("/");

        auto resp = co_await client->sendRequestCoro(req);
        CO_REQUIRE(resp != nullptr);
        CHECK(resp->getStatusCode() == k200OK);
        CHECK(resp->contentType() == CT_APPLICATION_JSON);
    };

    sync_wait(api_test());
}
```

啟動 Drogon 事件迴圈

部分測試需啟動 Drogon 事件迴圈，例如 HTTP client 預設於全域事件迴圈執行。以下範例可處理多種情境，保證事件迴圈於測試開始前啟動：

```
int main()
{
    std::promise<void> p1;
    std::future<void> f1 = p1.get_future();

    // 於另一執行緒啟動主迴圈
    std::thread thr([&]() {
        // 事件迴圈啟動後 fulfill promise
```

```
        app().getLoop()->queueInLoop([&p1]() { p1.set_value(); });
        app().run();
    });

    // 事件迴圈啟動後才繼續
    f1.get();
    int status = test::run(argc, argv);

    // 要求事件迴圈結束並等待
    app().getLoop()->queueInLoop([]() { app().quit(); });
    thr.join();
    return status;
}
```

CMake 整合

如多數測試框架，DrogonTest 可整合至 CMake。`ParseAndAddDrogonTests` 會將原始檔中的測試加入 CMake 的 CTest。

```
find_package(Drogon REQUIRED) # 也會載入 ParseAndAddDrogonTests
add_executable(mytest main.cpp)
target_link_libraries(mytest PRIVATE Drogon::Drogon)
ParseAndAddDrogonTests(mytest)
```

即可透過建置系統（如 Makefile）執行測試：

```
❯ make test
Running tests...
Test project path/to/your/test/build/
      Start  1: Sum
 1/1  Test  #1: Sum .................................   Passed    0.00 sec
```