

概述

原文：[ENG-01-Overview.md](#)

Dragon 是一個基於 C++17/20 的 HTTP 應用程式框架。Dragon 可用於輕鬆建構各類型的 Web 應用伺服器程式，採用 C++ 開發。

Dragon 這個名字來自美國影集《權力遊戲》裡我很喜歡的一隻龍。

Dragon 主要運行於 Linux 平台，同時也支援 Mac OS ·FreeBSD 及 Windows。

其主要特色如下：

- 採用基於 epoll (macOS/FreeBSD 下為 kqueue) 的非阻塞 I/O 網路函式庫，提供高併發、高效能的網路 I/O，詳情請參考 [TFB 測試結果](#)；
- 提供完整的非同步程式設計模式；
- 支援 Http1.0/1.1 (伺服器端與客戶端)；
- 基於模板，實作簡易反射機制，完全解耦主程式框架、控制器與視圖；
- 支援 Cookie 與內建 Session；
- 支援後端渲染，控制器產生資料交由視圖產生 Html 頁面。視圖以 CSP 模板檔描述，可透過 CSP 標籤將 C++ 程式碼嵌入 Html 頁面，並由 drogon 命令列工具自動產生 C++ 程式碼檔案以供編譯；
- 支援視圖頁面動態載入（執行時動態編譯與載入）；
- 提供方便靈活的路由解決方案，從路徑對應至控制器處理函式；
- 支援過濾器鏈，便於在處理 HTTP 請求前執行統一邏輯（如登入驗證、HTTP 方法限制等）；
- 支援 https (基於 OpenSSL)；
- 支援 WebSocket (伺服器端與客戶端)；
- 支援 JSON 格式請求與回應，對 Restful API 應用開發非常友善；
- 支援檔案下載與上傳；
- 支援 gzip ·brotli 壓縮傳輸；
- 支援管線化 (pipelining)；
- 提供輕量級命令列工具 drogon_ctl，簡化 Dragon 各類型類別建立與視圖程式碼產生；
- 支援非阻塞 I/O 非同步讀寫資料庫 (PostgreSQL 及 MySQL/MariaDB 資料庫)；
- 支援基於執行緒池的 sqlite3 資料庫非同步讀寫；
- 支援 ARM 架構；
- 提供方便輕量的 ORM 實作，支援物件與資料庫雙向映射；
- 支援插件，可於載入時透過設定檔安裝；
- 內建 AOP (面向切面程式設計) 支援。

下一步：[安裝 dragon](#)

安裝

原文：[ENG-02-Installation.md](#)

本章以 Ubuntu 24.04 ·CentOS 7.5 ·MacOS 12.2 為例介紹安裝流程，其他系統大致相同。

系統需求

- Linux 核心版本需不低於 2.6.9，且為 64 位元；
- gcc 版本需不低於 5.4.0，建議使用 11 以上版本；
- 建置工具採用 cmake，cmake 版本需不低於 3.5；
- 版本管理工具採用 git；

套件相依性

- 內建
 - trantor：非阻塞 I/O C++ 網路函式庫，由 Drogon 作者開發，已作為 git 子模組，無需事先安裝；
- 必要
 - jsoncpp：C++ 的 JSON 函式庫，版本需 **不低於 1.7**；
 - libuuid：產生 uuid 的 C 函式庫；
 - zlib：支援壓縮傳輸；
- 選用
 - boost：版本需 **不低於 1.61**，僅當 C++ 編譯器不支援 C++17 或 STL 不完整支援 `std::filesystem` 時才需安裝。
 - OpenSSL：安裝後可支援 HTTPS，否則僅支援 HTTP。
 - c-ares：安裝後可提升 DNS 效率；
 - libbrotli：安裝後可支援 brotli 壓縮 HTTP 回應；
 - PostgreSQL ·MariaDB ·sqlite3 的開發函式庫，安裝任一即可支援對應資料庫；
 - hiredis：安裝後可支援 Redis；
 - gtest：安裝後可編譯單元測試；
 - yaml-cpp：安裝後可支援 yaml 格式設定檔。

系統準備範例

Ubuntu 24.04

- 環境

```
sudo apt install git gcc g++ cmake
```

- jsoncpp

```
sudo apt install libjsoncpp-dev
```

- `uuid`

```
sudo apt install uuid-dev
```

- `zlib`

```
sudo apt install zlib1g-dev
```

- OpenSSL (選用, 若需支援 HTTPS)

```
sudo apt install openssl libssl-dev
```

Arch Linux

- 環境

```
sudo pacman -S git gcc make cmake
```

- `jsoncpp`

```
sudo pacman -S jsoncpp
```

- `uuid`

```
sudo pacman -S uuid
```

- `zlib`

```
sudo pacman -S zlib
```

- OpenSSL (選用, 若需支援 HTTPS)

```
sudo pacman -S openssl libssl
```

CentOS 7.5

- 環境

```
yum install git  
yum install gcc  
yum install gcc-c++
```

```
# 預設 cmake 版本過低，請以原始碼安裝  
git clone https://github.com/Kitware/CMake  
cd CMake/  
./bootstrap && make && make install
```

```
# 升級 gcc  
yum install centos-release-scl  
yum install devtoolset-11  
scl enable devtoolset-11 bash
```

注意：scl enable devtoolset-11 bash 只會暫時啟用新版 gcc，若需永久啟用，請執行 echo "scl enable devtoolset-11 bash" >> ~/.bash_profile，系統重啟後自動啟用新版 gcc。

- jsoncpp

```
git clone https://github.com/open-source-parsers/jsoncpp  
cd jsoncpp/  
mkdir build  
cd build  
cmake ..  
make && make install
```

- uuid

```
yum install libuuid-devel
```

- zlib

```
yum install zlib-devel
```

- OpenSSL (選用，若需支援 HTTPS)

```
yum install openssl-devel
```

MacOS 12.2

- 環境

MacOS 內建基本工具，只需升級即可。

```
# 升級 gcc  
brew upgrade
```

- jsoncpp

```
brew install jsoncpp
```

- uuid

```
brew install ossp-uuid
```

- zlib

```
yum install zlib-devel
```

- OpenSSL (選用，若需支援 HTTPS)

```
brew install openssl
```

Windows

- 環境 (Visual Studio 2019) 安裝 Visual Studio 2019 專業版，至少包含：
 - MSVC C++ 編譯工具
 - Windows 10 SDK
 - C++ CMake 工具
 - Google Test 測試介面

conan 套件管理工具可提供 Drogon 所需所有相依套件。若系統已安裝 python，可透過 pip 安裝 conan：

```
pip install conan
```

也可至 conan 官方網站下載安裝檔。

建立 `conanfile.txt` 並加入下列內容：

- jsoncpp

```
[requires]
jsoncpp/1.9.4
```

- uuid

無需安裝，Windows 10 SDK 已內建 uuid 函式庫。

- zlib

```
[requires]
zlib/1.2.11
```

- OpenSSL (選用，若需支援 HTTPS)

```
[requires]
openssl/1.1.1t
```

資料庫環境 (選用)

注意：以下函式庫非必須，可依實際需求選擇安裝一種或多種資料庫。若需開發資料庫相關 Web 應用，請先安裝資料庫開發環境再安裝 drogon，否則會遇到 NO DATABASE FOUND 問題。

- PostgreSQL

需安裝 PostgreSQL 原生 C 函式庫 libpq，安裝方式如下：

- ubuntu 16 : sudo apt-get install postgresql-server-dev-all
- ubuntu 18 : sudo apt-get install postgresql-all
- ubuntu 24 : sudo apt-get install postgresql-all
- arch : sudo pacman -S postgresql
- centos 7 : yum install postgresql-devel
- MacOS : brew install postgresql
- Windows conanfile : libpq/13.4

- MySQL

MySQL 原生函式庫不支援非同步讀寫，建議使用 MariaDB (由原開發社群維護，與 MySQL 相容且支援非同步)，作業系統不應同時安裝 MySQL 與 MariaDB。

MariaDB 安裝方式如下：

- ubuntu 18.04 : sudo apt install libmariadbclient-dev
- ubuntu 24.04 : sudo apt install libmariadb-dev-compat libmariadb-dev
- arch : sudo pacman -S mariadb
- centos 7 : yum install mariadb-devel
- MacOS : brew install mariadb
- Windows conanfile : libmariadb/3.1.13

- **Sqlite3**

- ubuntu : sudo apt-get install libsqlite3-dev
- arch : sudo pacman -S sqlite3
- centos : yum install sqlite-devel
- MacOS : brew install sqlite3
- Windows conanfile : sqlite3/3.36.0

- **Redis**

- ubuntu : sudo apt-get install libhiredis-dev
- arch : sudo pacman -S redis
- centos : yum install hiredis-devel
- MacOS : brew install hiredis
- Windows conanfile : hiredis/1.0.0

注意：部分指令僅安裝開發函式庫，若需安裝伺服器請自行查詢。

Drogon 安裝

假設上述環境與套件皆已準備好，安裝流程如下：

- **Linux 原始碼安裝**

```
cd $WORK_PATH
git clone https://github.com/drogonframework/drogon
cd drogon
git submodule update --init
mkdir build
cd build
cmake ..
make && sudo make install
```

預設編譯為 debug 版本，若需 release 版本，cmake 指令請加下列參數：

```
cmake -DCMAKE_BUILD_TYPE=Release ..
```

安裝完成後，以下檔案會安裝至系統（可用 CMAKE_INSTALL_PREFIX 變更安裝路徑）：

- drogon 標頭檔：/usr/local/include/drogon
- drogon 函式庫檔案 libdrogon.a：/usr/local/lib
- drogon 命令列工具 drogon_ctl：/usr/local/bin
- trantor 標頭檔：/usr/local/include/trantor
- trantor 函式庫檔案 libtrantor.a：/usr/local/lib

• Windows 原始碼安裝

1. 下載 drogon 原始碼

開啟 Windows 搜尋列，搜尋 x64 Native Tools，選擇 x64 Native Tools Command Prompt for VS 2019 作為命令列工具。

```
cd $WORK_PATH
git clone https://github.com/drogonframework/drogon
cd drogon
git submodule update --init
```

2. 安裝相依套件

透過 conan 安裝：

```
mkdir build
cd build
conan profile detect --force
conan install .. -s compiler="msvc" -s compiler.version=193 -s
compiler.cppstd=17 -s build_type=Debug --output-folder . --
build=missing
```

修改 `conanfile.txt` 可變更套件版本。

3. 編譯與安裝

```
cmake .. -DCMAKE_BUILD_TYPE=Debug -
DCMAKE_TOOLCHAIN_FILE="conan_toolchain.cmake" -
DCMAKE_POLICY_DEFAULT_CMP0091=NEW -DCMAKE_INSTALL_PREFIX="D:"
cmake --build . --parallel --target install
```

注意：conan 與 cmake 的 build type 必須一致。

安裝完成後，以下檔案會安裝至系統（可用 CMAKE_INSTALL_PREFIX 變更安裝路徑）：

- drogon 標頭檔：`D:/include/drogon`
- drogon 函式庫檔案 `drogon.dll`：`D:/bin`

- drogon 命令列工具 drogon_ctl.exe : D:/bin
- trantor 標頭檔 : D:/include/trantor
- trantor 函式庫檔 trantor.dll : D:/lib

新增 bin 與 cmake 目錄至 path :

D:\bin

D:\lib\cmake\Dropgon

D:\lib\cmake\Trantor

- **Windows vcpkg 安裝**

[懶人包教學](#)

安裝 vcpkg :

1. 以 git 安裝 vcpkg。

```
git clone https://github.com/microsoft/vcpkg  
cd vcpkg  
./bootstrap-vcpkg.bat
```

更新 vcpkg 只需執行 git pull

2. 將 vcpkg 加入 Windows 環境變數 path。
3. 檢查 vcpkg 是否安裝成功，輸入 vcpkg 或 vcpkg.exe

安裝 Dropgon :

1. 安裝 drogon 框架，指令如下：

- 32 位元 : vcpkg install drogon
- 64 位元 : vcpkg install drogon:x64-windows
- 進階 : vcpkg install jsoncpp:x64-windows zlib:x64-windows openssl:x64-windows sqlite3:x64-windows libpq:x64-windows libpqxx:x64-windows drogon[core,ctl,sqlite3,postgres,orm]:x64-windows

注意：

- 若有套件未安裝導致錯誤，請個別安裝。例如：

zlib：`vcpkg install zlib` 或 `vcpkg install zlib:x64-windows` (64 位元)

- 檢查已安裝套件：

`vcpkg list`

- 使用 `drogon_ctl`：`vcpkg install drogon[ctl]` (32 位元) 或 `vcpkg install drogon[ctl]:x64-windows` (64 位元)。輸入 `vcpkg search drogon` 可查詢更多安裝選項。

2. 新增 `drogon_ctl` 指令與相依套件，請將下列路徑加入環境變數：

```
C:\dev\vcpkg\installed\x64-windows\tools\drogon
C:\dev\vcpkg\installed\x64-windows\bin
C:\dev\vcpkg\installed\x64-windows\lib
C:\dev\vcpkg\installed\x64-windows\include
C:\dev\vcpkg\installed\x64-windows\share
C:\dev\vcpkg\installed\x64-windows\debug\bin
C:\dev\vcpkg\installed\x64-windows\debug\lib
```

重新啟動／開啟 `powershell`。

3. 重新啟動／開啟 `powershell`，輸入：`drogon_ctl` 或 `drogon_ctl.exe`，若顯示：

```
usage: drogon_ctl [-v | --version] [-h | --help] <command>
[<args>]
commands list:
create           create some source files(Use 'drogon_ctl
help create' for more information)
help            display this message
press           Do stress testing(Use 'drogon_ctl help
press' for more information)
version         display version of this tool
```

即安裝成功。

注意：熟悉獨立 `gcc` 或 `g++` (`msys2 mingw-w64 tdm-gcc`) 或 Microsoft Visual Studio 編譯器者，建議使用 `make.exe/nmake.exe/ninja.exe` 作為 `cmake` generator，Linux/Windows 開發與部署切換較不易出錯。

• Docker 映像檔安裝

官方已於 [docker hub](#) 提供預建映像檔，所有 `Drogon` 相依套件與本身皆已安裝，可直接於 `docker` 環境建置 `Drogon` 應用。

• Nix 套件安裝

`Drogon` 於 21.11 版釋出 Nix 套件。

尚未安裝 Nix 者：請參考 [NixOS 官方網站](#) 安裝。

於專案根目錄新增 `shell.nix`，內容如下：

```
{ pkgs ? import <nixpkgs> {} }:  
pkgs.mkShell {  
    nativeBuildInputs = with pkgs; [  
        cmake  
    ];  
  
    buildInputs = with pkgs; [  
        drogon  
    ];  
  
}
```

執行 `nix-shell` 進入 Dargon 開發環境。

Nix 套件可依需求調整選項：

選項	預設值
sqliteSupport	true
postgresSupport	false
redisSupport	false
mysqlSupport	false

範例：

```
buildInputs = with pkgs; [  
    (drogon.override {  
        sqliteSupport = false;  
    })  
];
```

- **CPM.cmake** 引入

可用 [CPM.cmake](#) 引入 drogon 原始碼：

```
include(cmake/CPM.cmake)  
  
CPMAddPackage(  
    NAME drogon  
    VERSION 1.7.5  
    GITHUB_REPOSITORY drogonframework/dragon  
    GIT_TAG v1.7.5
```

```
)  
  
target_link_libraries(${PROJECT_NAME} PRIVATE drogon)
```

- 專案本地引入 drogon 原始碼

亦可將 drogon 原始碼放於專案目錄 third_party (記得更新 submodule) , 於 cmake 檔加入 :

```
add_subdirectory(third_party/dragon)  
target_link_libraries(${PROJECT_NAME} PRIVATE dragon)
```

下一步: 快速開始

快速開始

原文：[ENG-03-Quick-Start.md](#)

靜態網站

先從一個簡單範例介紹 drogon 的使用方式。本例將透過命令列工具 `drogon_ctl` 建立專案：

```
drogon_ctl create project your_project_name
```

專案目錄中已包含多個實用資料夾：

build	建置資料夾
CMakeLists.txt	專案 cmake 設定檔
config.json	Drogon 應用設定檔
controllers	控制器原始碼存放資料夾
filters	過濾器原始碼存放資料夾
main.cc	主程式
models	資料庫模型檔案資料夾
model.json	
views	視圖 csp 檔案資料夾

使用者可將各類檔案（如控制器、過濾器、視圖等）放入對應資料夾。為方便管理且減少錯誤，強烈建議使用 `drogon_ctl` 指令建立自己的 Web 應用專案。詳情請參考 [drogon_ctl](#)。

來看 `main.cc` 檔案內容：

```
#include <drogon/HttpAppFramework.h>
int main() {
    //設定 HTTP 監聽位址與埠號
    drogon::app().addListener("0.0.0.0", 80);
    //載入設定檔
    //drogon::app().loadConfigFile("../config.json");
    //啟動 HTTP 框架，此方法會在事件迴圈中阻塞
    drogon::app().run();
    return 0;
}
```

接著建置專案：

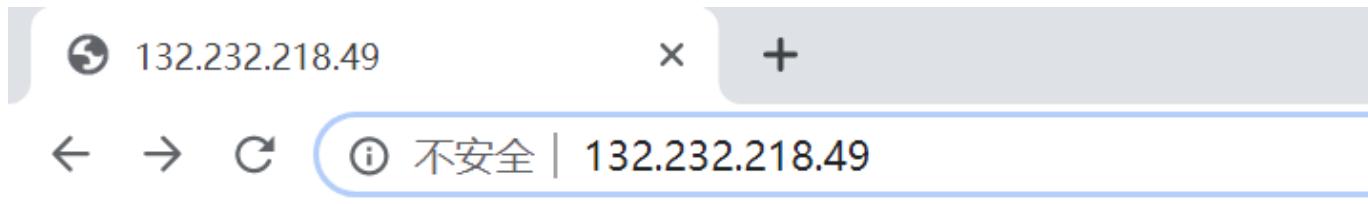
```
cd build
cmake ..
make
```

編譯完成後，執行目標檔 `./your_project_name`。

現在只要在 Http 根目錄新增一個靜態檔案 index.html：

```
echo '<h1>Hello Drogon!</h1>' >>index.html
```

預設根目錄為 `". /"`，可透過 config.json 修改。詳情請參考 [設定檔](#)。然後即可透過網址 `"http://localhost"` 或 `"http://localhost/index.html"` (或伺服器 IP) 瀏覽此頁面。



Hello Drogon!

若伺服器找不到請求頁面，會回傳 404 頁面：

404 Not Found

drogon/0.9.0.166

注意：請確認伺服器防火牆已開放 80 埠，否則無法瀏覽網頁。（或將埠號由 80 改為 1024 以上，避免出現以下錯誤訊息）：

```
FATAL Permission denied (errno=13) , Bind address failed at 0.0.0.0:80 -  
Socket.cc:67
```

可將靜態網站目錄與檔案複製到 Web 應用啟動目錄，即可用瀏覽器存取。drogon 預設支援的檔案類型有：

- html
- js
- css
- xml
- xsl
- txt
- svg
- ttf

- otf
- woff2
- woff
- eot
- png
- jpg
- jpeg
- gif
- bmp
- ico
- icns

drogon 也提供介面可自訂支援檔案類型，詳情請參考 [HttpAppFramework API](#)。

動態網站

接下來介紹如何在應用程式中加入控制器，讓控制器回應內容。

可用 drogon_ctl 命令列工具產生控制器原始碼，於 `controllers` 資料夾執行：

```
drogon_ctl create controller TestCtrl
```

會新增兩個檔案：`TestCtrl.h` 與 `TestCtrl.cc`：

`TestCtrl.h` 範例如下：

```
#pragma once
#include <drogon/HttpSimpleController.h>
using namespace drogon;
class TestCtrl:public drogon::HttpSimpleController<TestCtrl>
{
public:
    virtual void asyncHandleHttpRequest(const HttpRequestPtr &req,
                                         std::function<void (const
HttpServletResponsePtr &)> &&callback)override;
    PATH_LIST_BEGIN
    //在此定義路徑
    //PATH_ADD("/path","filter1","filter2",HttpMethod1,HttpMethod2...);
    PATH_LIST_END
};
```

`TestCtrl.cc` 範例如下：

```
#include "TestCtrl.h"
void TestCtrl::asyncHandleHttpRequest(const HttpRequestPtr &req,
                                         std::function<void (const
HttpServletResponsePtr &)> &&callback)
```

```
{
    //在此撰寫應用邏輯
}
```

編輯這兩個檔案，讓控制器回應 "Hello World!"：

TestCtrl.h 範例如下：

```
#pragma once
#include <drogon/HttpSimpleController.h>
using namespace drogon;
class TestCtrl:public drogon::HttpSimpleController<TestCtrl>
{
public:
    virtual void asyncHandleHttpRequest(const HttpRequestPtr &req,
                                         std::function<void (const
HttpServletResponsePtr &)> &&callback)override;
    PATH_LIST_BEGIN
    //在此定義路徑

    //範例
    //PATH_ADD("/path","filter1","filter2",HttpMethod1,HttpMethod2...);

    PATH_ADD("/",Get,Post);
    PATH_ADD("/test",Get);
    PATH_LIST_END
};
```

使用 PATH_ADD 將 '/' 與 '/test' 兩路徑對應至處理函式，並可加上 HTTP 方法限制。

TestCtrl.cc 範例如下：

```
#include "TestCtrl.h"
void TestCtrl::asyncHandleHttpRequest(const HttpRequestPtr &req,
                                      std::function<void (const
HttpServletResponsePtr &)> &&callback)
{
    //在此撰寫應用邏輯
    auto resp=HttpResponse::newHttpResponse();
    //注意：下方 enum 常數名稱為 "k200OK" (即 200 OK)，不是 "k2000K"。
    resp->setStatusCode(k200OK);
    resp->setContentTypeCode(CT_TEXT_HTML);
    resp->setBody("Hello World!");
    callback(resp);
}
```

重新以 CMake 編譯專案，執行目標檔 `./your_project_name`：

```
cd ../build  
cmake ..  
make  
./your_project_name
```

於瀏覽器輸入 "<http://localhost/>" 或 "<http://localhost/test>"，即可看到 "Hello World!"。

注意：若伺服器同時有靜態與動態資源，drogon 會優先使用動態資源。本例中 GET <http://localhost/> 回應為 Hello World!（來自 TestCtrl 控制器），而非靜態檔案 index.html 的 Hello Drogon!。

可見在應用程式中加入控制器非常簡單，只需新增對應原始碼檔案，甚至主程式檔案都不需修改。這種鬆耦合設計對 Web 應用開發非常有效。

注意：drogon 對控制器原始碼檔案位置沒有限制，也可放在 "./"（專案根目錄），甚至可於 [CMakeLists.txt](#) 定義新目錄。建議仍以 controllers 資料夾管理，方便維護。

下一步: 控制器 - 簡介

控制器 - 簡介

原文：[ENG-04-0-Controller-Introduction.md](#)

控制器在 Web 應用開發中非常重要。這裡我們會定義 URL、允許的 HTTP 方法、套用哪些過濾器，以及如何處理與回應請求。drogon 框架已協助處理網路傳輸、Http 協定解析等細節，開發者只需專注於控制器邏輯；每個控制器物件可擁有一個或多個處理函式（通常稱為 handler），其介面一般定義如下：

```
Void handlerName(const HttpRequestPtr &req,
                  std::function<void (const HttpResponsePtr &) > &&callback,
                  ...);
```

其中 `req` 是 Http 請求物件（以智慧指標包裝），`callback` 是框架傳給控制器的回呼函式物件，控制器產生回應物件（同樣以智慧指標包裝）後，透過 `callback` 傳給 drogon，框架會自動將回應內容送到瀏覽器。最後的 `...` 是參數列表，drogon 會依照對應規則將 Http 請求中的參數自動對應到函式參數，讓開發更方便。

這是一個非同步介面，可在其他執行緒完成耗時操作後再呼叫 `callback`。

drogon 有三種控制器類型：`HttpSimpleController`、`HttpController` 及 `WebSocketController`。使用時需繼承對應的類別模板。例如，自訂 `HttpSimpleController` 類別 "MyClass" 宣告如下：

```
class MyClass:public drogon::HttpSimpleController<MyClass>
{
public:
    //TestController(){}
    virtual void asyncHandleHttpRequest(const HttpRequestPtr &req,
                                         std::function<void (const
                                         HttpResponsePtr &) > &&callback) override;

    PATH_LIST_BEGIN
    PATH_ADD("/json");
    PATH_LIST_END
};
```

控制器生命週期

註冊到 drogon 框架的控制器，整個應用程式執行期間最多只會有一個實例且不會被銷毀，因此可在控制器類別中宣告並使用成員變數。注意：控制器 `handler` 執行時可能處於多執行緒環境（當框架 IO 執行緒數大於 1），若需存取非暫存變數，請務必做好並行保護。

下一步：[HttpSimpleController](#)

控制器 - HttpSimpleController

原文：[ENG-04-1-Controller-HttpSimpleController.md](#)

你可以使用 `drogon_ctl` 命令列工具，快速產生基於 `HttpSimpleController` 的自訂控制器類別原始碼。指令格式如下：

```
drogon_ctl create controller <[namespace::]class_name>
```

我們建立一個名為 `TestCtrl` 的控制器類別：

```
drogon_ctl create controller TestCtrl
```

執行後會產生兩個新檔案：`TestCtrl.h` 與 `TestCtrl.cc`。以下分別說明：

`TestCtrl.h`：

```
#pragma once
#include <drogon/HttpSimpleController.h>
using namespace drogon;
class TestCtrl:public drogon::HttpSimpleController<TestCtrl>
{
public:
    virtual void asyncHandleHttpRequest(const HttpRequestPtr &req,
                                         std::function<void (const
HttpResponsePtr &)> &&callback)override;
    PATH_LIST_BEGIN
    //在此定義路徑
    //PATH_ADD("/path","filter1","filter2",HttpMethod1,HttpMethod2...);
    PATH_LIST_END
};
```

`TestCtrl.cc`：

```
#include "TestCtrl.h"
void TestCtrl::asyncHandleHttpRequest(const HttpRequestPtr &req,
                                      std::function<void (const
HttpResponsePtr &)> &&callback)
{
    //在此撰寫應用邏輯
}
```

每個 `HttpSimpleController` 類別只能定義一個 Http 請求處理函式，並以虛擬函式覆寫。

URL 路徑到 handler 的路由（或稱映射）是透過巨集完成。可用 **PATH_ADD** 巨集新增多個路徑映射，所有 **PATH_ADD** 需寫在 **PATH_LIST_BEGIN** 與 **PATH_LIST_END** 之間。

第一個參數是要映射的路徑，後續參數則是對該路徑的限制。目前支援兩種限制：一是 **HttpMethod** 列舉型別，代表允許的 Http 方法；另一種是 **HttpFilter** 類別名稱。可任意組合這兩種限制，順序不限。Filter 詳見 [中介層與過濾器](#)。

使用者可將同一個 Simple Controller 註冊到多個路徑，也可在同一路徑註冊多個 Simple Controller（使用不同 HTTP 方法）。

你可以定義一個 **HttpResponse** 物件，然後用 **callback()** 回傳：

```
//在此撰寫應用邏輯
auto resp=HttpResponse::newHttpResponse();
resp->setStatusCode(k200OK);
resp->setContentTypeCode(CT_TEXT_HTML);
resp->setBody("你的頁面內容");
callback(resp);
```

上述路徑到 handler 的映射是在編譯期完成。事實上，drogon 框架也提供執行期完成映射的介面。執行期映射可讓使用者透過設定檔或其他介面動態新增或修改路由，無需重新編譯程式（為效能考量，**app().run()** 執行後禁止再新增任何控制器映射）。

下一步：[HttpController](#)

控制器 - HttpController

原文：[ENG-04-2-Controller-HttpController.md](#)

產生方式

你可以使用 `drogon_ctl` 命令列工具，快速產生基於 `HttpController` 的自訂控制器類別原始碼。指令格式如下：

```
drogon_ctl create controller -h <[namespace::]class_name>
```

我們建立一個命名空間為 `demo v1`、類別名稱為 `User` 的控制器：

```
drogon_ctl create controller -h demo::v1::User
```

執行後會新增兩個檔案：`demo_v1_User.h` 與 `demo_v1_User.cc`。

`demo_v1_User.h` 範例如下：

```
#pragma once

#include <drogon/HttpController.h>

using namespace drogon;

namespace demo
{
namespace v1
{
class User : public drogon::HttpController<User>
{
public:
    METHOD_LIST_BEGIN
        // 可用 METHOD_ADD 新增自訂處理函式 ;
        // METHOD_ADD(User::get, "/{2}/{1}", Get); // 路徑為
        // /demo/v1/User/{arg2}/{arg1}
        // METHOD_ADD(User::your_method_name, "/{1}/{2}/list", Get); // 路徑為
        // /demo/v1/User/{arg1}/{arg2}/list
        // ADD_METHOD_TO(User::your_method_name, "/absolute/path/{1}/{2}/list",
        // Get); // 路徑為 /absolute/path/{arg1}/{arg2}/list

    METHOD_LIST_END
        // 處理函式宣告範例 :
        // void get(const HttpRequestPtr& req, std::function<void (const
        // HttpResponsePtr &)> &&callback, int p1, std::string p2);
        // void your_method_name(const HttpRequestPtr& req, std::function<void
```

```
(const HttpResponsePtr &)> &&callback, double p1, int p2) const;  
};  
}  
}
```

demo_v1_User.cc 範例如下：

```
#include "demo_v1_User.h"  
  
using namespace demo::v1;  
  
// 在此新增處理函式定義
```

使用方式

編輯上述兩個檔案：

demo_v1_User.h 範例如下：

```
#pragma once  
  
#include <drogon/HttpController.h>  
  
using namespace drogon;  
  
namespace demo  
{  
    namespace v1  
    {  
        class User : public drogon::HttpController<User>  
        {  
            public:  
                METHOD_LIST_BEGIN  
                // 可用 METHOD_ADD 新增自訂處理函式；  
                METHOD_ADD(User::login, "/token?userId={1}&passwd={2}", Post);  
                METHOD_ADD(User::getInfo, "/{1}/info?token={2}", Get);  
                METHOD_LIST_END  
                // 處理函式宣告範例：  
                void login(const HttpRequestPtr &req,  
                           std::function<void (const HttpResponsePtr &)> &&callback,  
                           std::string &userId,  
                           const std::string &password);  
                void getInfo(const HttpRequestPtr &req,  
                            std::function<void (const HttpResponsePtr &)> &&callback,  
                            std::string userId,  
                            const std::string &token) const;  
        };  
    }  
}
```

demo_v1_User.cc 範例如下：

```
#include "demo_v1_User.h"

using namespace demo::v1;

// 在此新增處理函式定義

void User::login(const HttpRequestPtr &req,
                  std::function<void (const HttpResponsePtr &)> &&callback,
                  std::string &&userId,
                  const std::string &password)
{
    LOG_DEBUG<<"User "<<userId<<" login";
    //驗證演算法、讀取資料庫、驗證、識別等...
    //...
    Json::Value ret;
    ret["result"]="ok";
    ret["token"]=drogon::utils::getUuid();
    auto resp=HttpResponse::newHttpJsonResponse(ret);
    callback(resp);
}
void User::getInfo(const HttpRequestPtr &req,
                   std::function<void (const HttpResponsePtr &)>
&&callback,
                   std::string userId,
                   const std::string &token) const
{
    LOG_DEBUG<<"User "<<userId<<" get his information";

    //驗證 token 有效性等
    //讀取資料庫或快取取得使用者資訊
    Json::Value ret;
    ret["result"]="ok";
    ret["user_name"]="Jack";
    ret["user_id"]=userId;
    ret["gender"]=1;
    auto resp=HttpResponse::newHttpJsonResponse(ret);
    callback(resp);
}
```

每個 `HttpController` 類別可定義多個 Http 請求處理函式。由於函式數量可任意多，無法用虛擬函式覆寫，因此需在框架中註冊 handler 本身（而非類別）。

- **路徑映射**

URL 路徑到 handler 的映射是透過巨集完成。可用 `METHOD_ADD` 或 `ADD_METHOD_TO` 巨集新增多個路徑映射，所有相關巨集需寫在 `METHOD_LIST_BEGIN` 與 `METHOD_LIST_END` 之間。

METHOD_ADD 巨集會自動在路徑映射加上命名空間與類別名稱前綴。因此本例 login 函式註冊到 `/demo/v1/user/token` 路徑，`getInfo` 註冊到 `/demo/v1/user/xxx/info` 路徑。限制條件與 `HttpSimpleController` 的 **PATH_ADD** 類似。

若使用 **ADD_METHOD** 巨集且類別有命名空間，存取網址需加上命名空間。本例可用 `http://localhost/demo/v1/user/token?userid=xxx&passwd=xxx` 或 `http://localhost/demo/v1/user/xxxxx/info?token=xxxx`。

ADD_METHOD_TO 巨集則註冊絕對路徑，不自動加前綴。

`HttpController` 提供更彈性的路徑映射機制，可將一組函式集中於同一類別。

此外，巨集也提供參數映射方式，可將路徑上的查詢參數對應到函式參數列表。URL 路徑參數數量與函式參數位置對應，常見型別（如 `std::string` `int` `float` `double` 等）皆可自動轉型。建議 `lvalue reference` 使用 `const` 型別。

同一路徑可多次映射，依 HTTP 方法區分，這是 Restful API 常見做法，例如：

```
METHOD_LIST_BEGIN
    METHOD_ADD(Book::getInfo, "/{1}?detail={2}", Get);
    METHOD_ADD(Book::newBook, "/{1}", Post);
    METHOD_ADD(Book::deleteOne, "/{1}", Delete);
METHOD_LIST_END
```

路徑參數佔位符可用多種寫法：

- `{}`：路徑位置即函式參數位置，直接對應。
- `{1}, {2}`：有編號者對應指定參數。
- `{anystring}`：僅提升可讀性，等同 `{}`。
- `{1:anystring}, {2:xxx}`：冒號前為位置，後為說明，等同 `{1}:{2}`。

建議用後兩種，若路徑參數與函式參數順序一致，第三種即可。以下皆等價：

- `"/users/{}/books/{}"`
- `"/users/{}/books/{2}"`
- `"/users/{user_id}/books/{book_id}"`
- `"/users/{1:user_id}/books/{2}"`

注意：路徑比對不分大小寫，但參數名稱區分大小寫。參數值可混用大小寫，並原樣傳給控制器。

• 參數映射

路徑與查詢參數可對應到 `handler` 函式參數。目標參數型別需符合下列條件：

- 必須是值型別、`const` 左值參考或非 `const` 右值參考，不可為非 `const` 左值參考。建議用右值參考，方便處理。
- `int` `long` `long long` `unsigned long` `unsigned long long` `float` `double` `long double` 等基本型別。
- `std::string`

- 可用 `stringstream >>` 指派的型別。

此外，drogon 框架也支援將 `HttpRequestPtr` 物件映射為任意型別參數。若 `handler` 參數數量多於路徑參數，額外參數會由 `HttpRequestPtr` 物件轉換。可自訂型別轉換方式，只需在 `drogon` 命名空間特化 `fromRequest` 模板（定義於 `HttpRequest.h`），例如：

```
namespace myapp{
struct User{
    std::string userName;
    std::string email;
    std::string address;
};

namespace drogon
{
template <>
inline myapp::User fromRequest(const HttpRequest &req)
{
    auto json = req.getJsonObject();
    myapp::User user;
    if(json)
    {
        user.userName = (*json)[ "name" ].asString();
        user.email = (*json)[ "email" ].asString();
        user.address = (*json)[ "address" ].asString();
    }
    return user;
}
}
```

定義並特化模板後，可如下定義路徑映射與 `handler`：

```
class UserController:public drogon::HttpController<UserController>
{
public:
    METHOD_LIST_BEGIN
        //use METHOD_ADD to add your custom processing function here;
        ADD_METHOD_TO(UserController::newUser, "/users", Post);
    METHOD_LIST_END
    //your declaration of processing function maybe like this:
    void newUser(const HttpRequestPtr &req,
                 std::function<void (const HttpResponsePtr &)>
&&callback,
                 myapp::User &&pNewUser) const;
};
```

可見第三個 `myapp::User` 型別參數無對應路徑佔位符，框架會自動以 `req` 物件轉換並取得此參數，十分方便。

進一步說，若使用者只需自訂型別資料，不需存取 `HttpRequestPtr`，可將自訂物件放在第一個參數，框架也能正確完成映射，例如：

```
class UserController:public drogon::HttpController<UserController>
{
public:
    METHOD_LIST_BEGIN
        //use METHOD_ADD to add your custom processing function here;
        ADD_METHOD_TO(UserController::newUser, "/users", Post);
    METHOD_LIST_END
    //your declaration of processing function maybe like this:
    void newUser(myapp::User &pNewUser,
                 std::function<void (const HttpResponsePtr &)>
&&callback) const;
};
```

- **多路徑映射**

`drogon` 支援在路徑映射中使用正則表達式，可寫在 '{}' 之外。例如：

```
ADD_METHOD_TO(UserController::handler1, "/users/.*", Post); /// 匹配所有
/users/ 開頭路徑
ADD_METHOD_TO(UserController::handler2, "/{name}/[0-9]+", Post); /// 匹配
由名稱字串與數字組成的路徑
```

- **正則表達式映射**

上述方法僅有限度支援正則表達式。若需自由使用正則，`drogon` 提供 `ADD_METHOD_VIA_REGEX` 巨集，例如：

```
ADD_METHOD_VIA_REGEX(UserController::handler1, "/users/(.*)", Post); /// 
匹配所有 /users/ 開頭路徑，剩餘部分映射到 handler1 參數
ADD_METHOD_VIA_REGEX(UserController::handler2, ".*([0-9]*)", Post); /// 
匹配所有以數字結尾路徑，該數字映射到 handler2 參數
ADD_METHOD_VIA_REGEX(UserController::handler3, "/(?!data).*", Post); /// 
匹配所有非 /data 開頭路徑
```

可見參數映射也可用正則表達式，所有子表達式比對到的字串會依序映射到 `handler` 參數。

注意：使用正則時請留意比對衝突（多個 `handler` 同時比對成功）。同一控制器衝突時，僅執行第一個 `handler`（先註冊者）；不同控制器則不確定執行哪個 `handler`，請避免衝突。

下一步：[WebSocketController](#)

控制器 - WebSocketController

原文：[ENG-04-3-Controller-WebSocketController.md](#)

如其名，`WebSocketController` 用於處理 websocket 邏輯。Websocket 是一種基於 HTTP 的長連線協議，初始階段會有 HTTP 格式的請求與回應交換，連線建立後所有訊息都透過 websocket 傳送，訊息有固定格式包裝，內容與順序皆不受限制。

產生方式

`WebSocketController` 的原始檔可透過 `drogon_ctl` 工具產生，指令格式如下：

```
drogon_ctl create controller -w <[namespace::]class_name>
```

假設要實作一個簡單 echo 功能（伺服器回傳用戶端送來的訊息），可用 `drogon_ctl` 建立 `EchoWebsock` 實作類別：

```
drogon_ctl create controller -w EchoWebsock
```

指令會產生 `EchoWebsock.h` 與 `EchoWebsock.cc` 兩個檔案，內容如下：

```
//EchoWebsock.h
#pragma once
#include <drogon/WebSocketController.h>
using namespace drogon;
class EchoWebsock:public drogon::WebSocketController<EchoWebsock>
{
public:
    void handleNewMessage(const WebSocketConnectionPtr&,
                          std::string &&,
                          const WebSocketMessageType &) override;
    void handleNewConnection(const HttpRequestPtr &,
                            const WebSocketConnectionPtr&) override;
    void handleConnectionClosed(const WebSocketConnectionPtr&) override;
    WS_PATH_LIST_BEGIN
    //在此定義路徑
    WS_PATH_LIST_END
};
```

```
//EchoWebsock.cc
#include "EchoWebsock.h"
void EchoWebsock::handleNewMessage(const WebSocketConnectionPtr
&wsConnPtr, std::string &&message)
```

```

{
    //在此撰寫應用邏輯
}
void EchoWebsock::handleNewConnection(const HttpRequestPtr &req, const
WebSocketConnectionPtr &wsConnPtr)
{
    //在此撰寫應用邏輯
}
void EchoWebsock::handleConnectionClosed(const WebSocketConnectionPtr
&wsConnPtr)
{
    //在此撰寫應用邏輯
}

```

使用方式

- 路徑映射

編輯後：

```

//EchoWebsock.h
#pragma once
#include <drogon/WebSocketController.h>
using namespace drogon;
class EchoWebsock:public drogon::WebSocketController<EchoWebsock>
{
public:
    virtual void handleNewMessage(const WebSocketConnectionPtr&,
                                  std::string &&,
                                  const WebSocketMessageType &)override;
    virtual void handleNewConnection(const HttpRequestPtr &,
                                    const
WebSocketConnectionPtr&)override;
    virtual void handleConnectionClosed(const
WebSocketConnectionPtr&)override;
    WS_PATH_LIST_BEGIN
    //在此定義路徑
    WS_PATH_ADD("/echo");
    WS_PATH_LIST_END
};

```

```

//EchoWebsock.cc
#include "EchoWebsock.h"
void EchoWebsock::handleNewMessage(const WebSocketConnectionPtr
&wsConnPtr, std::string &&message)
{
    //在此撰寫應用邏輯
    wsConnPtr->send(message);
}

```

```

void EchoWebsock::handleNewConnection(const HttpRequestPtr &req, const
WebSocketConnectionPtr &wsConnPtr)
{
    //在此撰寫應用邏輯
}
void EchoWebsock::handleConnectionClosed(const WebSocketConnectionPtr
&wsConnPtr)
{
    //在此撰寫應用邏輯
}

```

本例中，控制器透過 `WS_PATH_ADD` 巨集註冊至 `/echo` 路徑。`WS_PATH_ADD` 用法與前述控制器巨集類似，也可搭配多個過濾器。由於 websocket 在框架中獨立處理，可與前兩種控制器 (`HttpSimpleController` `HttpApiController`) 路徑重複，互不影響。

實作三個虛擬函式時，僅 `handleNewMessage` 有實質內容，直接用 `send` 介面回傳收到的訊息。編譯後可測試效果。

注意：如同一般 HTTP 協定，websocket 也可被嗅探，若需安全性建議用 HTTPS 加密。當然也可自行在伺服器與用戶端加解密，但 HTTPS 更方便，底層由 drogon 處理，開發者只需專注業務邏輯。

自訂 websocket 控制器類別需繼承 `drgon::WebSocketController` 類別模板，模板參數為子類型。需實作下列三個虛擬函式，分別處理 websocket 建立、關閉與訊息：

```

virtual void handleNewConnection(const HttpRequestPtr &req, const
WebSocketConnectionPtr &wsConn);
virtual void handleNewMessage(const WebSocketConnectionPtr
&wsConn, std::string &&message,
const WebSocketMessageType &);
virtual void handleConnectionClosed(const WebSocketConnectionPtr
&wsConn);

```

說明如下：

- `handleNewConnection`：websocket 建立後呼叫，`req` 為用戶端送來的建立請求，框架已回應。可用 `req` 取得額外資訊（如 token）。`wsConn` 為 websocket 物件智慧指標，常用介面後述。
- `handleNewMessage`：收到新訊息時呼叫，`message` 為訊息內容，框架已解包與解碼，可直接處理。
- `handleConnectionClosed`：連線關閉後呼叫，可做收尾處理。

介面

WebSocketConnection 物件常用介面如下：

```

//傳送 websocket 訊息，編碼與包裝由框架處理
void send(const char *msg, uint64_t len);
void send(const std::string &msg);

```

```
//取得 websocket 本地與遠端位址  
const trantor::InetAddress &localAddr() const;  
const trantor::InetAddress &peerAddr() const;  
  
//取得 websocket 連線狀態  
bool connected() const;  
bool disconnected() const;  
  
//關閉 websocket  
void shutdown(); //關閉寫入  
void forceClose(); //強制關閉  
  
//設定與取得 websocket context，可儲存業務資料  
//any 型別可存任意物件  
void setContext(const any &context);  
const any &getContext() const;  
any *getMutableContext();
```

下一步: 中介層與過濾器

中介層與過濾器

原文：[ENG-05-Middleware-and-Filter.md](#)

在 `HttpController` 的範例中，`getInfo` 方法應在回傳使用者資訊前先檢查是否已登入。雖然可直接在 `getInfo` 方法中撰寫此邏輯，但檢查登入屬於多數介面都會用到的通用邏輯，應獨立抽出並於 `handler` 執行前配置，這正是 filter 的用途。

`drogon` 的中介層（middleware）採用洋蔥模型。框架完成 URL 路徑比對後，會依序呼叫該路徑註冊的中介層。每個中介層可選擇攔截或放行請求，並加入前置與後置處理邏輯。

若中介層攔截請求，則不會進入洋蔥內層，對應的 `handler` 也不會被呼叫，但仍會執行外層中介層的後置邏輯。

filter 實際上是省略後置操作的 middleware。註冊路徑時可同時使用 middleware 與 filter。

內建中介層／過濾器

`drogon` 內建常用 filter 如下：

- `drogon::IntranetIpFilter`：僅允許內網 IP 的 HTTP 請求，否則回傳 404。
- `drogon::LocalHostFilter`：僅允許 127.0.0.1 或 ::1 的 HTTP 請求，否則回傳 404。

自訂中介層／過濾器

中介層定義

使用者可自訂 middleware，需繼承 `HttpMiddleware` 類別模板，模板型別為子類型。例如，若要讓部分路由支援跨域，可如下定義：

```
class MyMiddleware : public HttpMiddleware<MyMiddleware>
{
public:
    MyMiddleware(){}; // 不可省略建構子

    void invoke(const HttpRequestPtr &req,
                MiddlewareNextCallback &&nextCb,
                MiddlewareCallback &&mcb) override
    {
        const std::string &origin = req->getHeader("origin");
        if (origin.find("www.some-evil-place.com") != std::string::npos)
        {
            // 直接攔截
            mcb(HttpResponse::newNotFoundResponse(req));
            return;
        }
        // 執行進入下一層前的處理
        nextCb([mcb = std::move(mcb)](const HttpResponsePtr &resp) {
            // 執行回傳後的處理
            resp->addHeader("Access-Control-Allow-Origin", origin);
        });
    }
}
```

```

        resp->addHeader("Access-Control-Allow-Credentials","true");
        mcb(resp);
    });
}
};

```

需覆寫父類別的 `invoke` 虛擬函式以實作 filter 邏輯。

此虛擬函式有三個參數：

- `req`：HTTP 請求物件；
- `nextCb`：進入洋蔥內層的 callback，呼叫即執行下一個 middleware 或最終 handler。呼叫時可傳入另一函式，該函式於回傳時執行，並接收內層回傳的 `HttpResponsePtr`。
- `mcb`：回到洋蔥外層的 callback，呼叫即回到外層。若略過 `nextCb` 僅呼叫 `mcb`，表示攔截請求直接回外層。

過濾器定義

使用者亦可自訂 filter，需繼承 `HttpFilter` 類別模板，模板型別為子類型。例如，若要建立 `LoginFilter`，可如下定義：

```

class LoginFilter:public drogon::HttpFilter<LoginFilter>
{
public:
    void doFilter(const HttpRequestPtr &req,
                  FilterCallback &&fcb,
                  FilterChainCallback &&fccb) override ;
};

```

可用 `drogon_ctl` 指令建立 filter，詳見 [drogon_ctl](#)。

需覆寫父類別的 `doFilter` 虛擬函式以實作 filter 邏輯。

此虛擬函式有三個參數：

- `req`：HTTP 請求物件；
- `fcb`：filter callback，型別為 `void (HttpResponsePtr)`，當判斷請求不合法時，透過此 callback 回傳特定回應給瀏覽器；
- `fccb`：filter chain callback，型別為 `void ()`，當判斷請求合法時，通知 `drogon` 執行下一個 filter 或最終 handler。

具體實作可參考 `drogon` 內建 filter。

中介層／過濾器註冊

註冊 middleware/filter 時，通常會搭配 controller 註冊。前述巨集（`PATH_ADD`・`METHOD_ADD` 等）可於最後加上 middleware/filter 名稱。例如將前述 `getInfo` 方法註冊行改為：

```
METHOD_ADD(User:: getInfo, "/{1}/info?token={2}", Get, "LoginFilter", "MyMiddleware");
```

路徑比對成功後，僅當下列條件皆成立才會呼叫 getInfo 方法：

1. 請求必須為 HTTP Get；
2. 請求方必須已登入；

可見 middleware/filter 註冊與配置非常簡單。controller 原始檔註冊 middleware 時無需 include 其標頭檔，middleware 與 controller 完全解耦。

注意：若 middleware/filter 定義於命名空間，註冊時必須完整寫出命名空間。

[下一步: 視圖](#)

視圖

[原文：ENG-06-View.md](#)

視圖簡介

雖然前端渲染技術盛行，後端應用服務通常只需回傳資料給前端，但好的 Web 框架仍應提供後端渲染技術，讓伺服器程式能動態產生 HTML 頁面。視圖（View）可協助使用者產生這些頁面，僅負責呈現相關工作，複雜業務邏輯則交由控制器處理。

早期 Web 應用是將 HTML 直接嵌入程式碼以動態產生頁面，但效率低且不直觀。後來如 JSP 等語言則反其道而行，將程式碼嵌入 HTML 頁面。drogon 採用後者方案，但因 C++ 需編譯執行，必須將嵌入 C++ 程式碼的頁面轉為 C++ 原始檔再編譯。因此 drogon 定義專屬 CSP（C++ Server Pages）描述語言，並用 drogon_ctl 命令列工具將 CSP 檔轉為 C++ 原始檔以供編譯。

Drogon 的 CSP

drogon 的 CSP 解決方案很簡單，使用特殊標記將 C++ 程式碼嵌入 HTML 頁面：

- `<%inc` 與 `%>` 之間內容視為需引用的標頭檔，只能寫 `#include`，如 `<%inc#include "xx.h" %>`，但多數常用標頭檔已自動引用，通常不需用此標籤；
- `<%c++` 與 `%>` 之間內容視為 C++ 程式碼，如 `<c++ std::string name="drogon"; %>`；
- C++ 程式碼一般會原樣轉至目標原始檔，僅下列兩個特殊標籤例外：
 - `@@` 代表控制器傳入的資料變數，可取得要顯示的內容；
 - `$$` 代表頁面內容的串流物件，可用 `<<` 將內容顯示於頁面；
- `[[` 與 `]]` 之間內容視為變數名稱，視圖會以該名稱為 key 從控制器傳入資料中尋找並輸出至頁面。變數名稱前後空白會被忽略，且成對標籤須同一行。效能考量下僅支援三種字串型別（`const char*`, `std::string` `const std::string`），其他型別請用上述 `$$` 方式輸出；
- `{%` 與 `%}` 之間內容視為變數名稱或 C++ 表達式（非控制器資料關鍵字），視圖會輸出變數或表達式值至頁面。`{%val.xx%}` 等同 `<%c++$$<<val.xx;%>`，但前者更簡潔直觀。成對標籤勿分行；
- `<%view` 與 `%>` 之間內容視為子視圖名稱，框架會尋找對應子視圖並填入標籤位置，名稱前後空白會忽略，勿分行。可多層巢狀但不可循環巢狀；
- `<%layout` 與 `%>` 之間內容視為版型名稱，框架會尋找對應版型並將本視圖內容填入版型指定位置（版型以 `[[[]]]` 標記此位置），名稱前後空白會忽略，勿分行。可多層巢狀但不可循環巢狀，一個模板檔僅能繼承一個版型，不支援多重繼承。

視圖使用方式

drogon 應用的 http 回應由控制器 handler 產生，視圖渲染的回應也是由 handler 產生，透過下列介面呼叫：

```
static HttpResponsePtr newHttpViewResponse(const std::string &viewName,
                                         const HttpViewData &data);
```

此介面為 `HttpResponse` 類別的靜態方法，含兩個參數：

- `viewName`：視圖名稱，即 csp 檔名（副檔名可省略）；

- **data**：控制器 handler 傳給視圖的資料，型別為 **Http ViewData**，這是一種特殊 map，可儲存與取得任意型別物件，詳見 [Http ViewData API] (API-Http ViewData) 說明。

控制器無需引用視圖標頭檔，兩者高度解耦，唯一連結即資料變數。

簡單範例

以下製作一個顯示瀏覽器送來 HTTP 請求參數的 html 頁面。

本例直接用 HttpAppFramework 介面定義 handler，在主程式 run() 前加入：

```
drogon::HttpAppFramework::instance()
    .registerHandler("/list_para",
                      [=](const HttpRequestPtr &req,
                           std::function<void (const HttpResponsePtr &)>
&&callback)
    {
        auto para=req->getParameters();
        Http ViewData data;
        data.insert("title","ListParameters");
        data.insert("parameters",para);
        auto
        resp=HttpResponse::newHttpViewResponse("ListParameters.csp",data);
        callback(resp);
    });
}
```

上述程式碼於 **/list_para** 路徑註冊 lambda handler，將請求參數傳給視圖顯示。

接著至 views 資料夾建立 ListParameters.csp，內容如下：

```
<!DOCTYPE html>
<html>
<%c++
    auto
para=@@.get<std::unordered_map<std::string, std::string, utils::internal::SafeStringHash>>("parameters");
%>
<head>
    <meta charset="UTF-8">
    <title>[[ title ]]</title>
</head>
<body>
    <%c++ if(para.size())%>
    <H1>Parameters</H1>
    <table border="1">
        <tr>
            <th>name</th>
            <th>value</th>
        </tr>
        <%c++ for(auto iter:para){%>
            <tr>
                <td>[[ iter.first ]]</td>
                <td>[[ iter.second ]]</td>
            </tr>
        <%c++ }%>
    </table>
</body>

```

```
<tr>
    <td>{%iter.first%}</td>
    <td><%c++ $$<<iter.second;%></td>
</tr>
<%c++%>
</table>
<%c++ %else{%
<H1>no parameter</H1>
<%c++%>
</body>
</html>
```

可用 **dragon_ctl** 工具將 ListParameters.csp 轉為 C++ 原始檔：

```
dragon_ctl create view ListParameters.csp
```

執行後會產生 ListParameters.h 與 ListParameters.cc，可編譯進 Web 應用。

以 CMake 重新編譯專案，執行目標程式 webapp，於瀏覽器輸入 http://localhost/list_para?p1=a&p2=b&p3=c，即可看到下列頁面：

Parameters

name	value
p1	a
p2	b
p3	c

後端渲染的 html 頁面就這樣產生了。

csp 檔自動化處理

注意：若專案是用 **dragon_ctl** 建立，以下步驟會自動由 **dragon_ctl** 處理。

顯然每次修改 csp 檔都手動執行 drogon_ctl 太不方便，可將 drogon_ctl 處理流程寫入 CMakeLists.txt。仍以前例為例，假設所有 csp 檔都放在 views 資料夾，CMakeLists.txt 可加上：

```
FILE(GLOB SCP_LIST ${CMAKE_CURRENT_SOURCE_DIR}/views/*.csp)
foreach(cspFile ${SCP_LIST})
    message(STATUS "cspFile: " ${cspFile})
    execute_process(COMMAND basename ARGS "-s .csp ${cspFile}"
OUTPUT_VARIABLE classname)
    message(STATUS "view classname: " ${classname})
    add_custom_command(
        OUTPUT ${classname}.h ${classname}.cc
        COMMAND drogon_ctl ARGS create view ${cspFile}
        DEPENDS ${cspFile}
        VERBATIM)
    set(VIEWSRC ${VIEWSRC} ${classname}.cc)
endforeach()
```

然後於 add_executable 加入新檔案集 \${VIEWSRC}：

```
Add_executable(webapp ${SRC_DIR} ${VIEWSRC})
```

視圖動態編譯與載入

drogon 提供於應用執行期間動態編譯與載入 csp 檔的方式，介面如下：

```
void enableDynamicViewsLoading(const std::vector<std::string> &libPaths);
```

此介面為 `HttpAppFramework` 成員方法，參數為 csp 檔所在目錄字串陣列。呼叫後 drogon 會自動搜尋這些目錄的 csp 檔，發現新檔或有修改時會自動產生原始檔、編譯成動態庫並載入應用，無需重啟。可自行實驗並觀察 csp 檔修改後頁面變化。

此功能依賴開發環境，若 drogon 與 webapp 都在本機編譯，動態載入 csp 頁面應無問題。

注意：動態視圖不可靜態編譯進應用，若已靜態編譯則無法用動態載入更新。可於開發時將視圖移至編譯目錄外的資料夾。

注意：此功能建議僅於開發階段調整 HTML 頁面，正式環境建議直接編譯 csp 檔進目標檔，主要考量安全與穩定性。

注意：若載入動態視圖時出現 `symbol not found` 錯誤，請用 `cmake .. -DCMAKE_ENABLE_EXPORTS=on` 設定專案，或取消註解 CMakeLists.txt 最後一行 `(set_property(TARGET ${PROJECT_NAME} PROPERTY ENABLE_EXPORTS ON))`，再重新編譯。

下一步: 會話

會話 (Session)

原文：[ENG-07-Session.md](#)

Session 是 Web 應用的重要概念，用於在伺服器端保存用戶端狀態，通常搭配瀏覽器的 cookie 使用，drogon 也有內建 session 支援。drogon 預設關閉 session，可透過下列介面開啟或關閉：

```
void disableSession();
void enableSession(const size_t timeout=0, Cookie::SameSite
sameSite=Cookie::SameSite::kNull);
```

上述方法皆透過 `HttpAppFramework` 單例呼叫。timeout 參數為 session 失效時間（秒），預設 1200 秒（20 分鐘未存取即失效），設為 0 則 session 保留至應用結束；sameSite 參數可設定 Set-Cookie HTTP 回應標頭的 SameSite 屬性。

開啟 session 前請確認用戶端支援 cookie，否則 drogon 會為每個無 `SessionID` cookie 的請求建立新 session，造成記憶體與運算資源浪費。

Session 物件

drogon 的 session 物件型別為 `drogon::Session`，用法類似 `Http ViewData`，可用關鍵字存取任意型別物件，支援並發讀寫，詳見 Session 類別說明。

drogon 框架會將 session 物件傳給 `HttpRequest` 物件，使用者可透過下列 `HttpRequest` 介面取得 Session 物件：

```
SessionPtr session() const;
```

此介面回傳 Session 物件智慧指標，可用來存取各種物件。

Session 範例

以下新增一個需 session 支援的功能：限制用戶存取頻率，若 10 秒內重複存取則回傳錯誤，否則回傳 ok。需在 session 記錄上次存取時間，並與本次存取時間比對。

建立一個 Filter 實作此功能，假設類別名為 TimeFilter，實作如下：

```
#include "TimeFilter.h"
#include <trantor/utils/Date.h>
#include <trantor/utils/Logger.h>
#define VDate "visitDate"
void TimeFilter::doFilter(const HttpRequestPtr &req,
                        FilterCallback &&cb,
                        FilterChainCallback &&ccb)
{
```

```

trantor::Date now=trantor::Date::date();
LOG_TRACE<<"";
if(req->session()->find(VDate))
{
    auto lastDate=req->session()->get<trantor::Date>(VDate);
    LOG_TRACE<<"last:"<<lastDate.toFormattedString(false);
    req->session()->modify<trantor::Date>(VDate,
                                                [now](trantor::Date &vdate) {
                                                    vdate = now;
                                                });
    LOG_TRACE<<"update visitDate";
    if(now>lastDate.after(10))
    {
        //10 秒後可再次存取
        ccb();
        return;
    }
    else
    {
        Json::Value json;
        json["result"]="error";
        json["message"]="存取間隔需至少 10 秒";
        auto res=HttpResponse::newHttpJsonResponse(json);
        cb(res);
        return;
    }
}
LOG_TRACE<<"首次存取，插入 visitDate";
req->session()->insert(VDate,now);
ccb();
}

```

接著於 `/slow` 路徑註冊 lambda 並掛上 TimeFilter，程式如下：

```

drogon::HttpAppFramework::instance()
    .registerHandler("/slow",
                     [=](const HttpRequestPtr &req,
                         std::function<void (const HttpResponsePtr
&) > &&callback)
    {
        Json::Value json;
        json["result"]="ok";
        auto
resp=HttpResponse::newHttpJsonResponse(json);
        callback(resp);
    },
    {Get, "TimeFilter"});

```

呼叫框架介面開啟 session：

```
drogon::HttpAppFramework::instance().enableSession(1200);
```

以 CMake 重新編譯專案，執行目標程式 webapp，即可於瀏覽器測試效果。

[下一步: 資料庫](#)

資料庫 - 概述

原文：[ENG-08-0-Database-General.md](#)

概述

Drogon 內建資料庫讀寫引擎，連線操作採用非阻塞 I/O 技術，應用程式自底層至上層皆以高效非阻塞非同步模式運作，確保高併發效能。目前支援 PostgreSQL 與 MySQL 資料庫。若需使用資料庫，必須先安裝對應資料庫的開發環境，Drogon 會自動偵測相關標頭與函式庫並編譯對應部分。資料庫開發環境準備請參考[開發環境](#)。

Drogon 亦支援 sqlite3 資料庫，適合輕量級應用。非同步介面以執行緒池實作，與上述資料庫介面一致。

DbClient

Drogon 資料庫操作的基本類別為 **DbClient**（抽象類別，具體型別依建構介面而定）。與一般資料庫介面不同，**DbClient** 物件不代表單一連線，而是可包含多個連線，可視為**連線池物件**。

DbClient 提供同步與非同步介面，非同步介面同時支援阻塞與非阻塞模式。建議配合 Drogon 非同步框架時使用非阻塞非同步介面。

通常呼叫非同步介面時，**DbClient** 會隨機選擇管理的閒置連線執行查詢，結果回傳後由 **DbClient** 處理資料並透過 callback 回傳給呼叫者；若無閒置連線，執行內容會暫存，待有連線完成自身 SQL 請求後，會從暫存取出待執行命令。

DbClient 詳細用法請參考[DbClient](#)。

Transaction

可由 **DbClient** 產生 **transaction** 物件以支援交易操作。除多出 **rollback()** 介面外，**transaction** 物件基本用法與 **DbClient** 相同。**transaction** 類別為 **Transaction**，詳情請參考[Transaction](#)。

ORM

Drogon 亦支援 **ORM**。使用者可用 drogon_ctl 指令讀取資料庫表格並產生對應 model 原始碼，再透過 **Mapper<MODEL>** 類別模板執行 model 的資料庫操作。**Mapper** 提供標準資料庫操作的簡易介面，讓使用者無需撰寫 SQL 即能對表格進行增刪改查。**ORM** 詳細用法請參考[ORM](#)。

下一步: [DbClient](#)

資料庫 - DbClient

[原文：ENG-08-1-Database-DbClient.md](#)

DbClient 物件建構

DbClient 物件有兩種建構方式：一是透過 DbClient 類別的靜態方法（見 DbClient.h 標頭檔），如下：

```
#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
```

上述介面可取得 DbClient 實作物件的智慧指標。connInfo 為連線字串，採 key=value 參數格式，詳情請見標頭檔註解。connNum 為 DbClient 的連線數，對併發有關鍵影響，請依實際需求設定。

以此方式取得的物件，需自行持有（如存入全域容器）。**不建議建立暫時物件用完即釋放**，原因如下：

- 會浪費連線建立與斷開時間，增加系統延遲；
- 介面本身為非阻塞，取得 DbClient 時其管理的連線尚未建立，且框架不提供連線成功 callback，若要查詢還得 sleep？這違背非同步框架初衷。

因此 DbClient 物件應於程式啟動時建立並全程持有。此工作可完全交由框架處理，drogon 提供第二種建構方式：透過設定檔或 `createDbClient()` 方法建構。設定檔方式詳見[db_clients](#)。

需要時可透過框架介面取得 DbClient 智慧指標，介面如下：

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

name 參數為設定檔中的 name 選項值，用於區分同一應用的多個 DbClient 物件。DbClient 管理的連線會自動重連，使用者無需關心連線狀態，幾乎隨時可用。**注意**：此方法不可於 `app.run()` 執行前呼叫，否則僅會取得空的 shared_ptr。

執行介面

DbClient 提供多種執行介面，列舉如下：

```
/// 非同步方法
template <typename FUNCTION1, typename FUNCTION2, typename... Arguments>
void execSqlAsync(const std::string &sql,
                  FUNCTION1 &&rCallback,
```

```

FUNCTION2 &&exceptCallback,
Arguments &&... args) noexcept;

/// 非同步方法 (future)
template <typename... Arguments>
std::future<const Result> execSqlAsyncFuture(const std::string &sql,
                                              Arguments &&... args)
noexcept;

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

/// 串流型方法
internal::SqlBinder operator<<(const std::string &sql);

```

因綁定參數數量與型別不定，這些方法皆為函式模板。

各方法特性如下表：

方法	同步/非同步	阻塞/非阻塞	例外處理
void execSqlAsync	非同步	非阻塞	不會丟出例外
std::future<const Result> execSqlAsyncFuture	非同步	get() 時阻塞	get() 時可能丟出例外
const Result execSqlSync	同步	阻塞	可能丟出例外
internal::SqlBinder operator<<	非同步	預設非阻塞	不會丟出例外

同步方法通常涉及網路 IO，故為阻塞；非同步方法則為非阻塞。但非同步方法也可阻塞執行，即方法會阻塞至 callback 執行完畢。此時 callback 會在呼叫者執行緒執行，然後方法才回傳。

高併發場景請用非同步非阻塞方法，低併發（如設備管理頁）可用同步方法以求直觀。

execSqlAsync

```

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

```

最常用的非同步介面，採非阻塞模式。

sql 為 SQL 字串，若有綁定參數請用對應資料庫的 placeholder 規則，如 PostgreSQL 用 \$1, \$2..., MySQL 用 ?。

args 為綁定參數，可為零或多個，數量需與 SQL placeholder 相同，型別可為：

- 整數型：各種長度整數，應與資料庫欄位型別相符；
- 浮點型：`float` 或 `double`，應與欄位型別相符；
- 字串型：`std::string` 或 `const char[]`，對應資料庫字串型或可用字串表示之型別；
- 日期型：`trantor::Date`，對應資料庫 date/datetime/timestamp；
- 二進位型：`std::vector<char>`，對應 PostgreSQL byte 或 MySQL blob；

參數可為左值或右值、變數或常數，皆可自由使用。

Callback 與 exceptCallback 分別為結果 callback 與例外 callback，定義如下：

- 結果 callback：型別 `void (const Result &)`，可傳入 `std::function<lambda>` 等；
- 例外 callback：型別 `void (const DrogonDbException &)`，可傳入相符型別 callable。

SQL 執行成功後，結果以 `Result` 類別包裝並透過結果 callback 回傳；若執行例外則執行例外 callback，可由 `DrogonDbException` 物件取得例外資訊。

範例：

```
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");
```

`Result` 物件為標準容器，支援 iterator，可用 range loop 取得每列物件。`Result::Row::Field` 物件介面詳見原始碼。

`DrogonDbException` 為所有資料庫例外基底類別，詳見原始碼註解。

execSqlAsyncFuture

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

非同步 future 介面省略前述兩個 callback，呼叫後立即回傳 future 物件，需呼叫 get() 取得結果。例外以 try/catch 機制取得，若 get() 未包在 try/catch 內且呼叫堆疊皆無 try/catch，SQL 執行例外時程式會直接結束。

範例：

```
auto f = clientPtr->execSqlAsyncFuture("select * from users where
org_name=$1",
                                         "default");
try
{
    auto result = f.get(); // 阻塞至取得結果或例外
    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 DragonDbException &e)
{
    std::cerr << "error:" << e.base().what() << std::endl;
}
```

execSqlSync

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

同步介面最簡單直觀，輸入 SQL 字串與綁定參數，回傳 Result 物件，呼叫時會阻塞執行緒，錯誤時丟出例外，需用 try/catch 處理。

範例：

```
try
{
    auto result = clientPtr->execSqlSync("update users set user_name=$1
where user_id=$2",
                                         "test",
                                         1); // 阻塞至取得結果或例外
    std::cout << result.affectedRows() << " rows updated!" << std::endl;
}
catch (const DragonDbException &e)
{
    std::cerr << "error:" << e.base().what() << std::endl;
}
```

operator<<

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

串流介面較特殊，透過 << 依序輸入 SQL 與參數，並用 >> 指定結果 callback 與例外 callback。例如前述 select 範例，串流介面寫法如下：

```
*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 DragonDbException &e)
    {
        std::cerr << "error:" << e.base().what() << std::endl;
    };
}
```

此用法等同第一個非同步非阻塞介面，選用何種介面可依習慣。若需阻塞模式，可用 << 輸入 Mode::Blocking 參數，本文不詳述。

此外串流介面有特殊用法，若用特殊結果 callback，框架可逐列傳回結果，callback 型別如下：

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

第一個 bool 參數為 true 時表示已無資料（所有結果已回傳），即最後一次 callback；後面參數依序對應每列欄位值，框架會自動型別轉換，使用者也需注意型別相符。可用 const 左值參考、右值參考或值型別。

以下用此 callback 改寫前述範例：

```
int i = 0;
*clientPtr << "select user_name, user_id from users where org_name=$1"
    << "default"
    >> [&i](boolisNull, 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 DragonDbException &e)
{
    std::cerr << "error:" << e.base().what() << std::endl;
};
```

可見 select 語句的 user_name 與 user_id 欄位值分別賦予 callback 的 name 與 id 變數，無需自行處理型別轉換，十分方便，可彈性運用。

注意：非同步程式設計時，需留意上述範例中的 i 變數。必須確保 callback 執行時 i 仍有效，因為是以參考捕獲。callback 會在其他執行緒呼叫，當下 context 可能已失效。通常用智慧指標持有暫時變數並於 callback 捕獲，以確保變數有效。

小結

每個 DbClient 物件都有一或多個 EventLoop 執行緒控制資料庫連線 IO，透過非同步或同步介面接收請求，並以 callback 回傳結果。

DbClient 的阻塞介面僅阻塞呼叫者執行緒，只要呼叫者非 EventLoop 執行緒，不會影響 EventLoop 正常運作。callback 執行時，程式會在 EventLoop 執行緒運作，因此勿於 callback 內執行阻塞操作，否則會影響資料庫讀寫併發效能。熟悉非阻塞 I/O 程式設計者應能理解此限制。

下一步: [Transaction](#)

資料庫 - 交易 (Transaction)

原文：[ENG-08-2-Database-Transaction.md](#)

交易 (Transaction) **是關聯式資料庫的重要功能，Drogon 以 **Transaction 類別提供交易支援。

Transaction 物件由 **DbClient** 建立，許多交易相關操作會自動執行：

- 在建立 **Transaction** 物件時，自動執行 begin 指令以「啟動」交易；
- 當 **Transaction** 物件被解構時，自動執行 commit 指令以結束交易；
- 若發生例外導致交易失敗，則自動執行 rollback 指令以回滾交易；
- 若交易已回滾，則 SQL 指令會回傳例外（丟出例外或執行例外 callback）。

交易建立

由 **DbClient** 提供交易建立方法如下：

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

此介面非常簡單，回傳 **Transaction** 物件的智慧指標。顯然，當智慧指標失去所有持有者並解構交易物件時，交易即結束。**commitCallback** 參數用於回傳交易提交是否成功。需注意，此 callback 僅用於指示 **commit** 指令是否成功。若交易在執行過程中自動或手動回滾，則不會執行此 callback。一般來說，**commit** 指令會成功，callback 的 bool 參數為 true。僅在特殊情況（如提交過程中連線中斷）才會通知提交失敗，此時伺服器端交易狀態不確定，使用者需特別處理。當然，考量此情況極少發生，非關鍵服務可選擇忽略此事件，建立交易時不傳入 **commitCallback**（預設為空 callback）。

交易必須獨佔資料庫連線。因此，建立交易時，**DbClient** 需從自身連線池選擇一個閒置連線交由交易物件管理。這會有一個問題：若 **DbClient** 所有連線都在執行 SQL 或其他交易，介面會阻塞直到有閒置連線。

框架也提供非同步交易建立介面如下：

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

此介面透過 callback 回傳交易物件，不會阻塞目前執行緒，確保應用高併發。使用者可依實際情況選用同步或非同步版本。

交易介面

Transaction 介面幾乎與 **DbClient** 相同，僅有以下兩點差異：

- **Transaction** 提供 **rollback()** 介面，允許使用者在任何情況下回滾交易。有時交易已自動回滾，再呼叫 **rollback()** 也不會有負面影響，故明確呼叫 **rollback()** 是良好策略，至少可確保不會誤提交。
- 使用者不可呼叫交易物件的 **newTransaction()** 介面，這很容易理解。雖然資料庫有子交易概念，框架目前尚未支援。

事實上，`Transaction` 設計為 `DbClient` 的子類別，目的是維持介面一致性，同時也方便 `ORM` 使用。

目前框架未提供交易隔離層級控制介面，即隔離層級為資料庫服務預設值。

交易生命週期

交易物件的智慧指標由使用者持有。當有未執行的 SQL 時，框架會持有該物件，因此不必擔心交易物件在尚有未執行 SQL 時被解構。此外，交易物件的智慧指標常在其介面的結果 callback 中捕獲並使用。這是正常用法，不必擔心循環引用導致物件無法銷毀，框架會自動協助打破循環引用。

範例

以最簡單的例子說明，假設有一個任務表，使用者選取未處理任務並將其狀態改為處理中。為避免併發競爭，使用 `Transaction` 類別，程式如下：

```
{  
    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!"  
                    // ... 處理任務 ...  
                }  
                >> [](const DragonDbException  
&e)  
                {  
                    std::cerr << "err:" <<  
                    e.base().what() << std::endl;  
                };  
            }  
            else  
            {  
                std::cout << "No new tasks found!" <<  
                std::endl;  
            }  
        },  
        [](const DragonDbException &e) {  
            std::cerr << "err:" << e.base().what() <<  
            std::endl;  
        }  
    );  
}
```

```
    } );
```

此例中，select for update 用於避免併發修改。update 語句在 select 結果 callback 中完成。最外層大括號用於限制 transPtr 生命週期，確保 SQL 執行完畢後能及時解構並結束交易。

下一步: [ORM](#)

資料庫 - ORM

原文：[ENG-08-3-Database-ORM.md](#)

Model (模型)

使用 Drogon ORM 時，首先需建立模型類別。Drogon 的指令工具 `drogon_ctl` 可自動產生模型類別，會根據使用者指定的資料庫讀取資料表資訊，並自動產生多個模型類別的原始碼檔案。使用模型時，請 `include` 對應的標頭檔。

每個 Model 類別對應一個資料表，Model 類別的實例則對應資料表的一筆紀錄。

建立模型類別的指令如下：

```
drogon_ctl create model <model_path>
```

最後一個參數是模型類別儲存路徑。該路徑下必須有 `model.json` 設定檔，內容為 `drogon_ctl` 連線資料庫的參數，格式為 JSON 並支援註解。範例如下：

```
{
    "rdbms": "postgresql",
    "host": "127.0.0.1",
    "port": 5432,
    "dbname": "test",
    "user": "test",
    "passwd": "",
    "tables": [],
    "relationships": {
        "enabled": false,
        "items": []
    }
}
```

設定參數與應用程式設定檔相同，詳見[設定檔](#)。

`tables` 為模型設定獨有選項，為字串陣列，每個字串代表要轉換為模型類別的資料表名稱。若此選項為空，則所有資料表都會產生模型類別。

專案目錄（由 `drogon_ctl create project` 指令建立）已預先建立 `models` 目錄及對應 `model.json`，使用者可編輯設定檔並用 `drogon_ctl` 指令建立模型類別。

Model 類別介面

主要有兩種使用者直接操作的介面：getter（取得）與 setter（設定）。

getter 介面分為兩種：

- `getColumnName`：取得欄位的智慧指標，回傳指標而非值，主要用於判斷欄位是否為 NULL。

- `getValueOfColumnName`：取得欄位值，為效率回傳常數參考。若欄位為 NULL，則回傳函式參數指定的預設值。

此外，二進位區塊型別（blob, bytea）有特殊介面 `getValueOfColumnNameAsString`，會將二進位資料載入 `std::string` 並回傳。

setter 介面用於設定欄位值，型式為 `setColumnName`，參數型別與欄位型別相符。自動產生的欄位（如自增主鍵）不會有 setter 介面。

`toJson()` 介面可將模型物件轉換為 JSON 物件，二進位型別會以 base64 編碼。

Model 類別的靜態成員代表資料表資訊，例如可透過 `cols` 靜態成員取得各欄位名稱，方便在支援自動提示的編輯器使用。

Mapper 類別模板

模型物件與資料表的映射由 Mapper 類別模板負責。Mapper 類別模板封裝了新增、刪除、修改等常用操作，讓使用者無需撰寫 SQL 即可操作資料。

Mapper 物件建構非常簡單，模板參數為要存取的模型型別，建構子僅有一個參數，即前述 DbClient 的智慧指標。Transaction 類別為 DbClient 子類別，因此也可用交易的智慧指標建構 Mapper，代表 Mapper 也支援交易。

Mapper 與 DbClient 一樣，提供非同步與同步介面。同步介面會阻塞且可能丟出例外，回傳的 future 物件在 `get()` 時阻塞且可能丟出例外。一般非同步介面不會丟出例外，而是透過結果 callback 與例外 callback 回傳結果。例外 callback 型別與 DbClient 介面相同。結果 callback 依介面功能分為多種，列表如下（T 為模板參數，即模型型別）：

Method	Return value	Parameter	Result callback	Blocking/ Non-blocking	exception	Description
T findByPrimaryKey	T	Value of the primary key	None	Blocking	Throw	Find the model object based on the primary key value, throw an exception if there is no result
void findByPrimaryKey	void	Value of the primary key and two callbacks	void(T)	Non-blocking	None	Find the model object based on the primary key value, call exception callback if there is no result
findFutureByPrimaryKey	future< T >	Value of the primary key	None	Block when calling the get() method	Throw when calling the get() method	Find the model object based on the primary key value, throw an exception if there is no result
vector<T> findAll	vector< T >	Node	None	Blocking	Throw	Return all rows in the table
void findAll	void	Two callbacks	void(vector< T >)	Non-blocking	None	Ibid
findFutureAll	future< vector< T >>	Node	None	Block when calling the get() method	Throw when calling the get() method	Ibid
size_t count	size_t	Criteria object	None	Blocking	Throw	Returns the number of rows that meet the criteria
void count	void	Criteria object and two callbacks	void(const size_t)	Non-blocking	None	Ibid
countFuture	future< size_t >	Criteria object	None	Block when calling the get() method	Throw when calling the get() method	Ibid
T findOne	T	Criteria object	None	Blocking	Throw	Return a row that meets the condition, if more or less than one row, throw an exception
		Criteria	/			Returns a qualifying row, if more than

void findOne	void	Criteria object and two callbacks	void(T)	Non-blocking	None	If more than one row is found, then an exception callback is executed
findFutureOne	future	Criteria object	None	Block when calling the get() method	Throw when calling the get() method	Return a row that meets the criteria, if more than one row is found, throw an exception
vector<T> findBy	vector<T>	Criteria object	None	Blocking	Throw	Return 0 or more rows that meet the criteria
void findBy	void	Criteria object and two callbacks	void(vector<T>)	Non-blocking	None	Ibid
findFutureBy	future<vector<T>>	Criteria object	None	Block when calling the get() method	Throw when calling the get() method	Ibid
insert	void	T&	None	Blocking	Throw	Insert a row of data, automatic field update in the parameter
insert	void	T&, two callbacks	void(T)	Non-blocking	None	Insert a row of data, automatic field update in the callback parameters
insertFuture	future	T&	None	Block when calling the get() method	Throw when calling the get() method	Insert a row of data, automatic field update in the object of future.get()
size_t update	size_t	const T&	None	Blocking	Throw	Update a row of data, return the number of updated rows which is 1 or 0, the table must have a primary key
void update	void	const T&, two callbacks	void(const size_t)	Non-blocking	None	Ibid

updateFuture	<code>future<size_t></code>	<code>const T&</code>	None	Block when calling the <code>get()</code> method	Throw when calling the <code>get()</code> method	Ibid
<code>size_t deleteOne</code>	<code>size_t</code>	<code>const T&</code>	None	Blocking	Throw	Delete a row of data, return the number of deleted rows which is 1 or 0, the table must have a primary key
<code>void deleteOne</code>	<code>void</code>	<code>const T&, two callbacks</code>	<code>void(const size_t)</code>	Non-blocking	None	Ibid
deleteFuture	<code>future<size_t></code>	<code>const T&</code>	None	Block when calling the <code>get()</code> method	Throw when calling the <code>get()</code> method	Ibid
<code>size_t deleteBy</code>	<code>size_t</code>	Criteria object	None	Blocking	Throw	Delete the eligible rows and return the deleted rows
<code>void deleteBy</code>	<code>void</code>	Criteria object and two callbacks	<code>void(const size_t)</code>	Non-blocking	None	Ibid
deleteFutureBy	<code>future<size_t></code>	Criteria object	None	Block when calling the <code>get()</code> method	Throw when calling the <code>get()</code> method	Ibid

注意：使用交易時，例外不一定會導致回滾。以下情況交易不會自動回滾：`findByPrimaryKey` 未找到合格列、`findOne` 找到少於或多於一筆紀錄時，Mapper 會丟出 `UnexpectedRows` 例外或進入例外 callback。若業務邏輯需回滾，請明確呼叫 `rollback()` 介面。

Criteria (條件物件)

許多介面需輸入 `criteria` 物件參數。`criteria` 物件為 `Criteria` 類別實例，表示某種條件，如欄位大於、等於、小於某值，或 `isNull` 等。

```
template <typename T>
Criteria(const std::string &colName, const CompareOperator &opera, T &&arg)
```

`criteria` 物件建構非常簡單，第一個參數為欄位名稱，第二個為比較型態的列舉值，第三個為比較值。若型態為 `IsNull` 或 `IsNotNull`，則不需第三個參數。

範例：

```
Criteria("user_id", CompareOperator::EQ, 1);
```

上述例子表示 user_id 欄位等於 1。實務上更常寫成：

```
Criteria(Users::Cols::_user_id, CompareOperator::EQ, 1);
```

此寫法可用編輯器自動提示，較有效率且不易出錯。

Criteria 類別也支援自訂 where 條件與自訂建構子：

```
template <typename... Arguments>
explicit Criteria(const CustomSql &sql, Arguments &&...args)
```

第一個參數為帶有 \$? placeholder 的 CustomSql 物件，CustomSql 類別僅為 std::string 包裝。第二個不定參數為綁定參數，與 execSqlAsync 相同。

範例：

```
Criteria(CustomSql("tags @> $?"), "cloud");
```

CustomSql 類別也有字串常值語法，建議寫成：

```
Criteria("tags @> $"_sql, "cloud");
```

Criteria 物件支援 AND 與 OR 運算，兩個 criteria 相加可建構新條件，方便組合巢狀條件。例如：

```
Mapper<Users> mp(dbClientPtr);
auto users = mp.findBy(
    (Criteria(Users::Cols::_user_name, CompareOperator::Like, "%Smith"))&&Criteria
    (Users::Cols::_gender, CompareOperator::EQ, 0))
|| 
    (Criteria(Users::Cols::_user_name, CompareOperator::Like, "%Johnson"))&&Criteria
    (Users::Cols::_gender, CompareOperator::EQ, 1))
);
```

上述程式會查詢 users 資料表中所有名為 Smith 的男性或名為 Johnson 的女性。

Mapper 鏈式介面

Mapper 類別模板也支援常見 SQL 約束（如 limit offset），以鏈式介面提供，使用者可串接多個約束。執行任一 10.5.3 節介面後，這些約束即被清除，僅於單次操作有效：

```
Mapper<Users> mp(dbClientPtr);
auto users =
mp.orderBy(Users::Cols::_join_time).limit(25).offset(0).findAll();
```

此程式會從 users 資料表選取使用者列表，回傳第一頁 25 筆。

鏈式介面名稱即其功能，詳情請參考 Mapper.h 標頭檔。

Convert (轉換層)

convert 為模型設定獨有選項，於資料庫讀寫前後加上轉換層。物件包含 enabled 布林值（是否啟用），items 陣列包含：

- table：欄位所屬資料表
- column：欄位名稱
- method：物件
 - after_db_read：讀取後呼叫的方法名稱，簽名：void([const] std::shared_ptr<type> [&])
 - before_db_write：寫入前呼叫的方法名稱，簽名：void([const] std::shared_ptr<type> [&])
- includes：字串陣列，include 檔名（加 " 或 <>）

Relationships (關聯)

資料表間關聯可透過 model.json 的 relationships 設定。採手動設定而非自動偵測外鍵，因實務專案常不使用外鍵。

若 enable 為 true，產生的模型類別會依設定加入對應介面。

關聯分三種：has one (一對一)、has many (一對多)、many to many (多對多)。

• has one (一對一)

一對一關聯，原始表一筆紀錄可關聯目標表一筆紀錄，反之亦然。例如 products 與 skus 表一對一，可設定如下：

```
{
  "type": "has one",
  "original_table_name": "products",
  "original_table_alias": "product",
  "original_key": "id",
  "target_table_name": "skus",
  "target_table_alias": "SKU",
  "target_key": "product_id",
  "enable_reverse": true
}
```

各欄位意義同上，若 enable_reverse 為 true，則目標表模型類別也會加入反向關聯介面。

依此設定，products 表模型類別會加入：

```
// Relationship interfaces
void getSKU(const DbClientPtr &clientPtr,
            const std::function<void(Skus)> &rcb,
            const ExceptionCallback &ecb) const;
```

這是非同步介面，會在 callback 回傳與目前 product 關聯的 SKU 物件。

skus 表模型類別則會加入：

```
// Relationship interfaces
void getProduct(const DbClientPtr &clientPtr,
                const std::function<void(Products)> &rcb,
                const ExceptionCallback &ecb) const;
```

- **has many (一對多)**

一對多關聯，目標表通常有欄位與原始表主鍵關聯。例如 products 與 reviews 一對多，可設定如下：

```
{
  "type": "has many",
  "original_table_name": "products",
  "original_table_alias": "product",
  "original_key": "id",
  "target_table_name": "reviews",
  "target_table_alias": "",
  "target_key": "product_id",
  "enable_reverse": true
}
```

依此設定，products 表模型類別會加入：

```
void getReviews(const DbClientPtr &clientPtr,
                const std::function<void(std::vector<Reviews>)>
&rcb,
                const ExceptionCallback &ecb) const;
```

reviews 表模型類別則會加入：

```
void getProduct(const DbClientPtr &clientPtr,
                const std::function<void(Products)> &rcb,
                const ExceptionCallback &ecb) const;
```

- **many to many (多對多)**

多對多關聯通常需中介表，每筆中介表紀錄對應原始表與目標表各一筆。例如 products 與 carts 多對多，可設定如下：

```
{
  "type": "many to many",
  "original_table_name": "products",
  "original_table_alias": "",
  "original_key": "id",
  "pivot_table": {
    "table_name": "carts_products",
    "original_key": "product_id",
    "target_key": "cart_id"
  },
  "target_table_name": "carts",
  "target_table_alias": "",
  "target_key": "id",
  "enable_reverse": true
}
```

依此設定，products 表模型類別會加入：

```
void getCart(const DbClientPtr &clientPtr,
             const
             std::function<void(std::vector<std::pair<Cart, CartProducts>>)> &rcb,
             const ExceptionCallback &ecb) const;
```

carts 表模型類別則會加入：

```
void getProducts(const DbClientPtr &clientPtr,
                  const
                  std::function<void(std::vector<std::pair<Product, CartProducts>>)>
                  &rcb,
                  const ExceptionCallback &ecb) const;
```

Restful API 控制器

drogon_ctl 也可在建立模型時自動產生每個模型（資料表）對應的 restful 風格控制器，讓使用者零程式碼即可產生增刪改查 API。這些 API 支援主鍵查詢、條件查詢、欄位排序、指定欄位回傳、欄位別名等功能，並可隱藏資料表結構。由 model.json 的 `restful_api_controllers` 選項控制，詳細請參考 json 檔註解。

每個資料表的控制器設計為基底類別加子類別。基底類別與資料表緊密關聯，子類別則用於實作特殊業務邏輯或修改介面格式。此設計可在資料表結構變更時只更新基底類別而不覆蓋子類別（設定 `generate_base_only` 為 true）。

下一步: [FastDbClient](#)

資料庫 - FastDbClient

原文：[ENG-08-4-Database-FastDbClient.md](#)

如其名，FastDbClient 相較於一般 DbClient 具備更高效能。不同於 DbClient 擁有獨立事件迴圈，FastDbClient 與網路 IO 執行緒及主執行緒共用事件迴圈，使其內部實作可採無鎖（lock-free）模式，效能更佳。

測試顯示，在極高負載下，FastDbClient 效能比 DbClient 高出 10% 至 20%。

建立與取得

FastDbClient 必須由框架自動依設定檔建立，或透過 app.createDbClient() 介面建立：

設定檔中 db_client 選項的子選項 `is_fast` 用來指定是否為 FastDbClient。或可呼叫 app.createDbClient() 方法，最後一個參數設為 true 即建立 FastDbClient。

框架會為每個 IO 事件迴圈及主事件迴圈分別建立 FastDbClient，每個 FastDbClient 內部管理多個資料庫連線。IO 事件迴圈數由框架的 "threads_num" 選項控制，通常設為主機 CPU 核心數。每個事件迴圈的 DB 連線數由 DB client 的 "connection_number" 選項決定。詳情請參考[設定檔](#)。因此 FastDbClient 持有的總連線數為 `(threads_num+1) * connection_number`。

取得 FastDbClient 的介面與一般 DbClient 類似，如下：

```
orm::DbClientPtr getFastDbClient(const std::string &name = "default");
/// 使用 drogon::app().getFastDbClient("clientName") 取得 FastDbClient 物件。
```

需特別注意，因 FastDbClient 的特殊性，必須在 IO 事件迴圈執行緒或主執行緒呼叫上述介面，才能取得正確的智慧指標。在其他執行緒僅會取得空指標，無法使用。

使用方式

FastDbClient 的使用方式幾乎與一般 DbClient 相同，但有以下限制：

- 取得與使用 FastDbClient 必須在框架的 IO 事件迴圈執行緒或主執行緒，否則會有不可預期錯誤（因破壞 lock-free 條件）。幸好大多數應用程式邏輯都在 IO 執行緒，如各控制器處理函式、過濾器 filter 函式等。FastDbClient 介面的各 callback 也都在目前 IO 執行緒，可安全巢狀使用。
- 絶不可使用 FastDbClient 的阻塞介面，因該介面會阻塞目前執行緒，而該執行緒同時負責此物件的資料庫 IO，將導致永久阻塞，使用者無法取得結果。
- FastDbClient 的同步交易建立介面可能會阻塞（當所有連線皆忙碌），因此同步交易建立介面會直接回傳空指標。若需在 FastDbClient 上使用交易，請改用非同步交易建立介面。
- 使用 FastDbClient 建立 Orm Mapper 物件後，也僅能使用 Mapper 的非同步非阻塞介面。

下一步：自動批次模式

資料庫 - 自動批次模式

原文：[ENG-08-5-Database-auto_batch.md](#)

自動批次模式僅適用於 postgresql 14 以上版本的 client library，其他情況會被忽略。說明自動批次處理前，先介紹 pipeline 模式。

pipeline 模式

自 postgresql 14 起，其 client library 提供 pipeline 模式介面。在 pipeline 模式下，可直接將新的 SQL 請求送至伺服器，無需等待前一請求結果回傳（與 HTTP pipeline 概念一致）。詳情請參考 [Pipeline mode](#)。此模式有助於效能提升，讓較少的資料庫連線可支撐更大量併發請求。

drogon 自 1.7.6 版起支援此模式，會自動檢查 libpq 是否支援 pipeline，若支援，透過 drogon DbClient 發送的所有請求皆採 pipeline 模式。

自動批次模式

預設下，非交易型 client，drogon 會為每個 SQL 請求建立同步點（synchronization point），即每個 SQL 都是隱式交易，確保 SQL 請求彼此獨立，讓 pipeline 與非 pipeline 模式對使用者而言完全等價。

但每個 SQL 都建立同步點會有效能負擔，因此 drogon 提供自動批次模式，改為每隔數個 SQL 才建立同步點。建立同步點的規則如下：

- EventLoop 迴圈中最後一個 SQL 必須建立同步點；
- 寫入資料庫的 SQL 後建立同步點；
- 大型 SQL 後建立同步點；
- 同一連線自上次同步點後連續 SQL 數達上限時建立同步點；

注意，同一連線兩個同步點間的 SQL 屬於同一隱式交易。drogon 不提供顯式開關同步點介面，因此這些 SQL 可能邏輯上無關，但因同屬一交易，會互相影響，故此模式並非完全安全，可能有以下問題：

- 失敗的 SQL 會導致前一同步點後的所有 SQL 回滾，但使用者不會收到通知（因未用顯式交易）；
- 失敗的 SQL 會導致下個同步點前的所有 SQL 都回傳失敗；
- 判斷寫入資料庫僅靠關鍵字比對（如 insert, update），無法涵蓋所有情境，例如 select 語句呼叫儲存程序，因此 drogon 雖盡力降低自動批次模式負面影響，但仍非完全安全；

因此，自動批次模式有助效能提升，但安全性不足，使用時請自行斟酌。例如僅用於純讀取 SQL。

注意 即使純讀取 SQL 也可能導致交易失敗（如 select timeout），其後續 SQL 也會受影響而失敗（可能不符使用者預期，因邏輯上彼此無關），故嚴格來說，自動批次模式僅適用於讀取且非關鍵資料查詢。

建議使用者為此類 SQL 另建自動批次 DbClient。當然，透過自動批次模式 DbClient 產生的交易物件則可安全使用。

啟用自動批次模式

使用 newPgClient 介面建立 client 時，第三個參數設為 true 即啟用自動批次模式；使用設定檔建立 client 時，auto_batch 選項設為 true 即啟用自動批次模式。

下一步：請求參考

參考 - Request

原文：[ENG-08-5-Database-auto_batch.md](#)

範例中的 `HttpRequest` 型別指標（通常命名為 `req`）代表 `drogon` 所接收或送出的請求資料，下列為可操作此物件的常用方法：

- `isOnSecureConnection()`

`isOnSecureConnection()` 說明

判斷請求是否為 `https`。

`isOnSecureConnection()` 輸入

無。

`isOnSecureConnection()` 回傳

`bool` 型別。

- `getMethod()`

`getMethod()` 說明

取得請求方法。適用於同一處理函式支援多種方法時判斷。

`getMethod()` 輸入

無。

`getMethod()` 回傳

`HttpMethod` 請求方法物件。

`getMethod()` 範例

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::anyhandle(const HttpRequestPtr &req,
    std::function<void (const HttpResponsePtr &)> &&callback) {
    if (req->getMethod() == HttpMethod::Get) {
        // do something
    } else if (req->getMethod() == HttpMethod::Post) {
        // do other something
    }
}
```

- `getParameter(const std::string &key)`

getParameter() 說明

依識別字取得參數值。GET 與 POST 請求行為略有不同。

getParameter() 輸入

參數識別字 (string 型別) 。

getParameter() 回傳

參數內容 (string 型別) 。

getParameter() 範例

GET 範例：

```
#include "mycontroller.h"
#include <string>

using namespace drogon;

void mycontroller::anyhandle(const HttpRequestPtr &req,
std::function<void (const HttpResponsePtr &)> &&callback) {
    // https://mysite.com/an-path/?id=5
    std::string id = req->getParameter("id");
    // 或
    long id = std::stol(req->getParameter("id"));
}
```

POST 範例：

```
#include "mycontroller.h"
#include <string>

using namespace drogon;

void mycontroller::loginHandle(const HttpRequestPtr &req,
std::function<void (const HttpResponsePtr &)> &&callback) {
    // request 為表單登入
    std::string email = req->getParameter("email");
    std::string password = req->getParameter("password");
}
```

- `getPath()`

getPath() 相關

path()

getPath() 說明

取得請求路徑。適用於使用 ADD_METHOD_VIA_REGEX 或其他動態 URL 的控制器。

getPath() 輸入

無。

getPath() 回傳

請求路徑 (string 型別)。

getPath() 範例

```
#include "mycontroller.h"
#include <string>

using namespace drogon;

void mycontroller::anyhandle(const HttpRequestPtr &req,
    std::function<void (const HttpResponsePtr &)> &&callback) {
    // https://mysite.com/an-path/?id=5
    std::string url = req->getPath();
    // url = /an-path/
}
```

- [getBody\(\)](#)

getBody() 相關

body()

getBody() 說明

取得請求 body 內容 (如有)。

getBody() 輸入

無。

getBody() 回傳

請求 body (string 型別，若有)。

- [getHeader\(std::string key\)](#)

getHeader() 說明

依識別字取得請求 header。

getHeader() 輸入

header 識別字 (string 型別)。

getHeader() 回傳

header 內容 (string 型別)。

getHeader() 範例

```
#include "mycontroller.h"
#include <string>

using namespace drogon;

void mycontroller::anyhandle(const HttpRequestPtr &req,
    std::function<void (const HttpResponsePtr &)> &&callback) {
    if (req->getHeader("Host") != "mysite.com") {
        // return http 403
    }
}
```

- **headers()**

headers() 說明

取得所有請求 header。

headers() 輸入

無。

headers() 回傳

unordered_map，包含所有 header。

headers() 範例

```
#include "mycontroller.h"
#include <unordered_map>
#include <string>

using namespace drogon;
```

```
void mycontroller::anyhandle(const HttpRequestPtr &req,
    std::function<void (const HttpResponsePtr &) > &&callback) {
    for (const std::pair<const std::string, const std::string> &header
        : req->headers()) {
        auto header_key = header.first;
        auto header_value = header.second;
    }
}
```

- **getCookie()**

getCookie() 說明

依識別字取得請求 cookie。

getCookie() 輸入

無。

getCookie() 回傳

cookie 值 (string 型別)。

- **cookies()**

cookies() 說明

取得所有請求 cookie。

cookies() 輸入

無。

cookies() 回傳

unordered_map，包含所有 cookie。

cookies() 範例

```
#include "mycontroller.h"
#include <unordered_map>
#include <string>

using namespace drogon;

void mycontroller::anyhandle(const HttpRequestPtr &req,
    std::function<void (const HttpResponsePtr &) > &&callback) {
    for (const std::pair<const std::string, const std::string> &header
        : req->cookies()) {
```

```
        auto cookie_key = header.first;
        auto cookie_value = header.second;
    }
}
```

- **getJsonObject()**

getJsonObject() 說明

將請求 body 轉為 Json 物件（通常用於 POST 請求）。

getJsonObject() 輸入

無。

getJsonObject() 回傳

Json 物件。

getJsonObject() 範例

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::anyhandle(const HttpRequestPtr &req,
std::function<void (const HttpResponsePtr &)> &&callback) {
    // body = {"email": "test@gmail.com"}
    auto jsonData = *req->getJsonObject();
    std::string email = jsonData["email"].asString();
}
```

實用技巧

以下非 `HttpRequest` 物件方法，但有助於處理收到的請求

檔案請求解析

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req, std::function<void (const HttpResponsePtr &)> &&callback) {
    // 僅限 Post 請求（檔案表單）

    MultiPartParser file;
```

```
file.parse(req);

if (file.getFiles().empty()) {
    // 未找到檔案
}

// 取得第一個檔案並儲存
const HttpFile archive = file.getFiles()[0];
archive.saveAs("/tmp/" + archive.getFileName());
}
```

更多檔案解析資訊請見：[檔案處理器](#)

下一步：[檔案處理器](#)

檔案處理器

原文：[ENG-09-1-File-Handler.md](#)

檔案解析是指將 multipart-data POST 請求中的檔案（或多個檔案）透過 **MultiPartParser** 物件解析為 **HttpFile** 物件，以下為相關說明：

MultiPartParser 物件

MultiPartParser 物件說明

用於解析並暫存請求中的檔案。

- **parse(const std::shared_ptr<HttpRequest> &req)**

parse() 說明

接收請求物件參數，讀取並識別檔案（如有），並轉存至 MultiPartParser 物件。

parse() 範例

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req, std::function<void
(const HttpResponsePtr &)> &&callback) {
    // 僅限 Post 請求(檔案表單)

    MultiPartParser fileParser;
    fileParser.parse(req);
}
```

- **getFiles()**

說明

必須在 **parse()** 之後呼叫，回傳請求中的檔案，型態為 **std::vector<HttpFile>**。

範例

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req,
std::function<void (const HttpResponsePtr &)> &&callback) {
```

```
// 僅限 Post 請求 (檔案表單)

MultiPartParser fileParser;
fileParser.parse(req);

// 檢查是否有檔案
if (fileParser.getFiles().empty()) {
    // 未找到檔案
}

size_t num_of_files = fileParser.getFiles().size();
}
```

- **getParameters()**

getParameters() 說明

必須在 **parse()** 之後呼叫，回傳 MultiPartData 表單中的其他欄位。

getParameters() 回傳

`std::unordered_map<std::string, std::string> (key, value)`

getParameters() 範例

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req, std::function<void
(const HttpResponsePtr &)> &&callback) {
    // 僅限 Post 請求 (檔案表單)

    MultiPartParser fileParser;
    fileParser.parse(req);

    if (!fileParser.getFiles().empty()) {
        for (const auto &header : fileParser.getParameters()){
            header.first // 表單欄位名稱
            header.second // 欄位值
        }
    }
}
```

- **getParameter<T>(const std::string &key)**

getParameter() 說明

必須在 **parse()** 之後呼叫，取得指定 key 的單一欄位值。

getParameter() 輸入

欲取得的型別（自動轉型）、參數 key。

getParameter() 回傳

指定 key 的參數內容（型別自動轉換），若不存在則回傳 T 型別預設值。

getParameter() 範例

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req, std::function<void
(const HttpResponsePtr &)> &&callback) {
    // 僅限 Post 請求(檔案表單)

    MultiPartParser fileParser;
    fileParser.parse(req);

    std::string email = fileParser.getParameter<std::string>
("email_form");

    // string 預設值為 ""
    if (email.empty()) {
        // email_form 未找到
    }
}
```

HttpFile 物件

HttpFile 物件說明

代表記憶體中的檔案，由 MultiPartParser 使用。

- **getFileName()**

getFileName() 說明

取得收到檔案的原始檔名。

getFileName() 回傳

std::string。

getFileName() 範例

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req, std::function<void
(const HttpResponsePtr &)> &&callback) {
    // 僅限 Post 請求(檔案表單)

    MultiPartParser fileParser;
    fileParser.parse(req);

    std::string filename = fileParser.getFiles()[0].getFileName();
}
```

- **fileLength()**

fileLength() 說明

取得檔案大小。

fileLength() 回傳

`size_t`

fileLength() 範例

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req, std::function<void
(const HttpResponsePtr &)> &&callback) {
    // 僅限 Post 請求(檔案表單)

    MultiPartParser fileParser;
    fileParser.parse(req);

    size_t filesize = fileParser.getFiles()[0].fileLength();
}
```

- **getFileExtension()**

getFileExtension() 說明

取得檔案副檔名。

getFileExtension() 回傳

std::string

getFileExtension() 範例

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req, std::function<void
(const HttpResponsePtr &)> &&callback) {
    // 僅限 Post 請求(檔案表單)

    MultiPartParser fileParser;
    fileParser.parse(req);

    std::string file_extension = fileParser.getFiles()
[0].getFileExtension();
}
```

- **getMd5()**

getMd5() 說明

取得檔案 MD5 雜湊值，可用於檢查完整性。

getMd5() 回傳

std::string

- **save()**

save() 說明

儲存檔案至檔案系統。儲存目錄為 config.json (或等效設定) 中的 **UploadPath**。完整路徑如下：

```
drogon::app().getUploadPath() + "/" + this->getFileName()
```

或簡化為：UploadPath/檔名

- **save(const std::string &path)**

save(path) 說明

傳入 path 參數時，使用指定路徑儲存檔案而非 UploadPath。

save(path) 範例

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req, std::function<void
(const HttpResponsePtr &) > &&callback) {
    // 僅限 Post 請求(檔案表單)

    MultiPartParser fileParser;
    fileParser.parse(req);

    // 相對路徑
    fileParser.getFiles()[0].save("./"); // 儲存至伺服器同目錄，檔名不變

    // 絕對路徑
    fileParser.getFiles()[0].save("/home/user/downloads/"); // 儲存至指定目
錄，檔名不變
}
```

- `saveAs(const std::string &path)`

saveAs(path) 說明

以新檔名儲存檔案 (忽略原始檔名)。

saveAs(path) 範例

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req, std::function<void
(const HttpResponsePtr &) > &&callback) {
    // 僅限 Post 請求(檔案表單)

    MultiPartParser fileParser;
    fileParser.parse(req);

    // 相對路徑
    fileParser.getFiles()[0].saveAs("./image.png"); // 儲存至伺服器同目錄，
    檔名自訂
    /* 僅為範例，實務勿直接覆蓋檔案格式，應先檢查是否真為 png */

    // 絶對路徑
    fileParser.getFiles()[0].save("/home/user/downloads/anyname." +
    fileParser.getFiles()[0].getFileExtension());
}
```

下一步: 插件

插件 (Plugins)

原文：[ENG-10-Plugins.md](#)

插件可協助使用者建構複雜應用程式。在 Drogon 中，所有插件皆依設定檔建置並安裝至應用程式。Drogon 的插件為單一實例，使用者可自訂任意功能。

當 Drogon 執行 run() 介面時，會依設定檔逐一實體化每個插件並呼叫其 `initAndStart()` 介面。

設定方式

插件設定於設定檔，例如：

```
"plugins": [
  {
    //name: 插件類別名稱
    "name": "DataDictionary",
    //dependencies: 依賴的其他插件，可省略
    "dependencies": [],
    //config: 插件設定，為初始化參數 json 物件，可省略
    "config": {}
  },
]
```

每個插件有三個設定：

- name：插件類別名稱（含命名空間），框架會依此建立插件實例。若註解此項，插件即停用。
- dependencies：依賴其他插件的名稱列表，框架會依依賴順序建立與初始化所有插件，並於程式結束時反向關閉與銷毀。禁止循環依賴，若偵測到循環依賴，Drogon 會報錯並退出。若註解此項，依賴列表為空。
- config：初始化插件的 json 物件，會傳入插件的 `initAndStart()` 介面。若註解此項，則傳入空物件。

定義方式

自訂插件需繼承 `drogon::Plugin` 類別模板，模板參數為插件型別，定義如下：

```
class DataDictionary : public drogon::Plugin<DataDictionary>
{
public:
    virtual void initAndStart(const Json::Value &config) override;
    virtual void shutdown() override;
    ...
};
```

可用 `drogon_ctl` 指令建立插件原始碼檔案：

```
dragon_ctl create plugin <[namespace::]class_name>
```

取得實例

插件實例由 dragon 建立，使用者可透過以下介面取得插件實例：

```
template <typename T> T *getPlugin();
```

或

```
PluginBase *getPlugin(const std::string &name);
```

一般建議用第一種方式。例如上述 DataDictionary 插件可如下取得：

```
auto *pluginPtr=app().getPlugin<DataDictionary>();
```

建議於框架 run() 介面呼叫後再取得插件，否則僅會取得未初始化的插件實例（雖不一定出錯，但請確保初始化後再使用）。由於插件依賴順序初始化，在 `initAndStart()` 介面中取得其他插件實例亦無問題。

生命週期

所有插件於框架 run() 介面初始化，應用程式結束時銷毀。因此插件生命週期幾乎與應用程式一致，`getPlugin()` 介面無需回傳智慧指標。

下一步: [設定檔](#)

設定檔 (Configuration File)

原文：[ENG-11-Configuration-File.md](#)

可透過多種介面設定 DrogonAppFramework 實例的 Http 伺服器行為，但建議使用設定檔，原因如下：

- 設定檔可在執行時決定應用行為，較原始碼設定更方便彈性；
- 設定檔可讓主程式更簡潔；

因此建議開發者以設定檔配置各項參數。

只需在呼叫 run() 前執行 loadConfigFile() 介面即可載入設定檔，例如：

```
int main()
{
    drogon::app().loadConfigFile("config.json");
    drogon::app().run();
}
```

上述程式載入 config.json 設定檔並執行應用。監聽埠、日誌、資料庫等皆可由設定檔配置。事實上，這段程式碼可作為 Web 應用主程式。

設定檔細節

範例設定檔可見於原始碼目錄頂層的 config.example.json。使用 drogon_ctl create project 指令建立專案時，專案目錄也會有 config.json。通常只需修改此檔即可完成 Web 應用設定。

檔案格式為 JSON，支援註解。可用 /**/ 或 // 註解不需的項目。

註解掉的選項會採用預設值，預設值可見於設定檔註解。

支援格式

-
- json
 - yaml (需安裝 yaml-cpp 套件)

SSL

ssl 選項用於設定 https 服務的 SSL 檔案：

```
"ssl": {
    "cert": "../trantor/trantor/tests/server.pem",
    "key": "../trantor/trantor/tests/server.pem",
    "conf": [
        ["Options", "Compression"],
        ["min_protocol", "TLSv1.2"]
    ]
}
```

`cert` 為憑證檔路徑，`key` 為私鑰檔路徑。若同檔案含憑證與私鑰，路徑可相同，格式為 PEM。

`conf` 為可選 SSL 參數，會直接傳給 `SSL_CONF_cmd` 以低階設定加密，選項需為一或兩元素陣列。

listeners

`listeners` 用於設定 Web 應用監聽器，為 JSON 陣列，每個物件代表一個監聽器：

```
"listeners": [
  {
    "address": "0.0.0.0",
    "port": 80,
    "https": false
  },
  {
    "address": "0.0.0.0",
    "port": 443,
    "https": true,
    "cert": "",
    "key": ""
  }
]
```

- `address`：字串型，監聽 IP，預設 "0.0.0.0"
- `port`：整數型，監聽埠，必填
- `https`：布林型，是否啟用 https，預設 false
- `cert`、`key`：https 時有效，憑證與私鑰路徑，預設空字串，表示用全域 ssl 設定

db_clients

`db_clients` 用於設定資料庫 client，為 JSON 陣列，每個物件代表一個 client：

```
"db_clients": [
  {
    "name": "",
    "rdbms": "postgresql",
    "host": "127.0.0.1",
    "port": 5432,
    "dbname": "test",
    "user": "",
    "passwd": "",
    "is_fast": false,
    "connection_number": 1,
    "filename": ""
  }
]
```

- `name` : client 名稱，預設 "default"，多 client 時需不同
 - `rdbms` : 資料庫型別，支援 "postgresql" ~"mysql" ~"sqlite3"
 - `host` : 資料庫主機，預設 "localhost"
 - `port` : 資料庫埠號
 - `dbname` : 資料庫名稱
 - `user` : 使用者名稱
 - `passwd` : 密碼
 - `is_fast` : 是否為 `FastDbClient`，預設 false
 - `connection_number` : 連線數，至少 1，預設 1，影響併發效能。`is_fast` 為 true 時為每個事件迴圈的連線數，否則為總連線數
 - `filename` : sqlite3 資料庫檔名
-

threads_num

app 子選項，整數型，預設 1，表示 IO 執行緒數，影響網路併發。此值不宜過大，建議設為預期網路 IO 佔用的處理器數。設為 0 時，執行緒數等於硬體核心數。

```
"threads_num": 16,
```

Session

Session 相關選項亦為 app 子選項，控制是否啟用 session 及逾時：

```
"enable_session": true,  
"session_timeout": 1200,
```

- `enable_session` : 是否啟用 session，預設 false。若 client 不支援 cookie，請設為 false，否則每次請求都會建立新 session，浪費資源
 - `session_timeout` : session 逾時秒數，預設 0 (永久有效)，僅在啟用 session 時有效
-

document_root

app 子選項，字串型，表示 Http 根目錄對應的文件路徑，也是靜態檔案下載根目錄，預設 "./" (程式執行目錄)。

```
"document_root": "./",
```

upload_path

app 子選項，字串型，表示檔案上傳預設路徑，預設 "uploads"。若非以 `./` 開頭，且不為 `.` 或 `..`，則為相對於 `document_root` 的路徑，否則為絕對路徑或相對於目前目錄。

```
"upload_path": "uploads",
```

client_max_body_size

app 子選項，字串型，表示請求 body 最大總大小。可用 k|m|g|t（大小寫皆可）指定單位。

```
"client_max_body_size": "10M",
```

client_max_memory_body_size

app 子選項，字串型，表示緩衝區最大大小，超過則快取至檔案。可用 k|m|g|t 指定單位。

```
"client_max_memory_body_size": "50K"
```

file_types

app 子選項，字串陣列，預設如下，表示支援下載的靜態檔案類型。若請求副檔名不在此列表，框架回傳 404。

```
"file_types": [  
    "gif",  
    "png",  
    "jpg",  
    "js",  
    "css",  
    "html",  
    "ico",  
    "swf",  
    "xap",  
    "apk",  
    "cur",  
    "xml"  
,
```

mime

app 子選項，字典型或字串陣列，宣告副檔名對應 MIME 類型（未被預設識別者）。此選項僅註冊 MIME，若副檔名不在 file_types，仍回傳 404。

```
"mime" : {  
    "text/markdown": "md",  
    "text/gemini": ["gmi", "gemini"]  
}
```

連線數量控制

app 子選項有兩個：

```
"max_connections": 100000,  
"max_connections_per_ip": 0,
```

- `max_connections`：最大同時連線數，預設 100000，達上限時新 TCP 連線會被拒絕
- `max_connections_per_ip`：單一 IP 最大連線數，預設 0（不限制）

日誌選項

app 子選項，JSON 物件，控制日誌輸出行為：

```
"log": {  
    "log_path": "./",  
    "logfile_base_name": "",  
    "log_size_limit": 100000000,  
    "log_level": "TRACE"  
},
```

- `log_path`：日誌檔儲存路徑，預設空字串（輸出至標準輸出）
- `logfile_base_name`：日誌檔名，預設空字串（即 drogon）
- `log_size_limit`：日誌檔大小上限 (bytes)，預設 100000000 (100M)，達上限時切換檔案
- `log_level`：最低日誌等級，預設 "DEBUG"，可選 "TRACE" "DEBUG" "INFO" "WARN"，TRACE 僅在 DEBUG 模式有效

注意：Drogon 的檔案日誌採非阻塞結構，可達百萬行/秒，效能可靠。

應用程式控制

app 子選項有兩個：

```
"run_as_daemon": false,  
"relaunch_on_error": false,
```

- `run_as_daemon`：是否以 daemon 方式在背景執行，預設 false
 - `relaunch_on_error`：是否錯誤時自動重啟，預設 false
-

use_sendfile

app 子選項，布林型，表示傳送檔案時是否用 linux sendfile 系統呼叫，預設 true。sendfile 可提升效率並降低大檔案記憶體用量。

```
"use_sendfile": true,
```

注意：即使設為 true，實際是否用 sendfile 由框架最佳化策略決定。

use_gzip

app 子選項，布林型，預設 true，表示 Http 回應 body 是否壓縮傳輸。啟用時：

- client 支援 gzip
- body 為文字型
- body 長度大於一定值

```
"use_gzip": true,
```

static_files_cache_time

app 子選項，整數型（秒），表示靜態檔案快取時間。重複請求於此時間內直接回傳快取內容，預設 5 秒，0 為永久快取（僅讀一次檔案，請慎用），負值為不快取。

```
"static_files_cache_time": 5,
```

simple_controllers_map

app 子選項，JSON 陣列，每個物件代表 Http 路徑對 HttpSimpleController 的映射。此設定為選用，詳見 [HttpSimpleController](#)。

```
"simple_controllers_map": [
  {
    "path": "/path/name",
    "controller": "controllerClassName",
    "http_methods": ["get", "post"],
    "filters": ["FilterClassName"]
```

```
    },  
],
```

- **path** : Http 路徑
- **controller** : HttpSimpleController 名稱
- **http_methods** : 支援的 Http 方法陣列，未在列表者回傳 405
- **filters** : 路徑上的過濾器列表，詳見 [中介層與過濾器](#)

閒置連線逾時控制

app 子選項，整數型（秒），預設 60。連線超過此時間未讀寫即強制斷線。

```
"idle_connection_timeout":60
```

動態視圖載入

app 子選項，控制動態視圖啟用與路徑，有兩個選項：

```
"load_dynamic_views":true,  
"dynamic_views_path":["./views"],
```

- **load_dynamic_views** : 布林型，預設 false。啟用時，框架會在視圖路徑動態編譯 .so 檔並載入，視圖檔變更時自動編譯與重載
- **dynamic_views_path** : 字串陣列，動態視圖搜尋路徑。若非以 `./` 開頭，且不為 `.` 或 `..`，則為相對於 `document_root` 的路徑，否則為絕對路徑或相對於目前目錄

詳見 [視圖](#)

Server header field

app 子選項，設定所有回應的 server header 欄位，預設空字串。若空字串，框架自動產生 **Server: drogon/版本字串**。

```
"server_header_field": ""
```

Keepalive requests

`keepalive_requests` 設定 keep-alive 連線可處理的最大請求數，達上限即關閉連線，預設 0（不限制）。

```
"keepalive_requests": 0
```

Pipelining requests

pipelining_requests 設定 pipelining 緩衝區可快取的最大未處理請求數，達上限即關閉連線，預設 0（不限制）。詳見 rfc2616-8.1.1.2。

```
"pipelining_requests": 0
```

下一步: [AOP 面向切面程式設計](#)

dragon_ctl - 指令說明

原文 : ENG-12-drogon_ctl-Command.md

當 **Dragon** 框架編譯安裝完成後，建議使用隨框架安裝的命令列工具 **drogon_ctl**（簡化指令為 **dg_ctl**）來建立第一個專案。使用者可依喜好選擇。

此工具主要功能是協助使用者快速建立各種 drogon 專案檔案。可用 `dg_ctl help` 指令查詢支援功能如下：

```
$ dg_ctl help
usage: drogon_ctl <command> [<args>]
commands list:
create          建立原始檔 (詳見 'drogon_ctl help create')
help            顯示說明訊息
version         顯示工具版本
press           壓力測試 (詳見 'drogon_ctl help press')
```

version 子指令

`version` 子指令用於顯示目前系統已安裝的 drogon 版本，例如：

create 子指令

`create` 子指令用於建立各種物件，目前是 `drogon_ctl` 的主要功能。可用 `dg_ctl help create` 查詢詳細說明：

```
$ dg_ctl help create  
使用 create 指令建立 drogon webapp 原始檔
```

用法: drogon_ctl create <view|controller|filter|project|model> [-options] <物件名稱>

```
dragon_ctl create view <csp 檔名> [-o <輸出路徑>] [-n <命名空間>] | [--path-to-namespace] //由 csp 檔產生 HttpView 原始檔

dragon_ctl create controller [-s] <[namespace:::]class_name> //建立 HttpSimpleController 原始檔

dragon_ctl create controller -h <[namespace:::]class_name> //建立 HttpController 原始檔

dragon_ctl create controller -w <[namespace:::]class_name> //建立 WebSocketController 原始檔

dragon_ctl create filter <[namespace:::]class_name> //建立 filter 原始檔

dragon_ctl create project <project_name> //建立專案

dragon_ctl create model <model_path> //建立 model 類別
```

• 視圖建立

`dg_ctl create view` 用於由 csp 檔產生視圖原始檔，詳見 [視圖](#)。一般不需直接使用此指令，建議於 cmake 設定自動執行。範例：

```
dg_ctl create view UsersList.csp
```

• 控制器建立

`dg_ctl create controller` 用於建立控制器原始檔，支援三種控制器：

- 建立 HttpSimpleController：

```
dg_ctl create controller SimpleControllerTest
dg_ctl create controller webapp::v1::SimpleControllerTest
```

最後參數為類別名稱，可加命名空間。

- 建立 HttpController：

```
dg_ctl create controller -h ControllerTest
dg_ctl create controller -h api::v1::ControllerTest
```

- 建立 WebSocketController：

```
dg_ctl create controller -w WsControllerTest
dg_ctl create controller -w api::v1::WsControllerTest
```

- **過濾器建立**

`dg_ctl create filter` 用於建立 filter 原始檔，詳見 [中介層與過濾器](#)。

```
dg_ctl create filter LoginFilter
dg_ctl create filter webapp::v1::LoginFilter
```

- **專案建立**

建立新 Drogon 應用專案最佳方式為 `drogon_ctl` 指令：

```
dg_ctl create project ProjectName
```

執行後會於目前目錄建立完整專案目錄，名稱為 `ProjectName`，可直接於 build 目錄編譯 (`cmake .. && make`)，預設無業務邏輯。

專案目錄結構如下：

└── build	編譯目錄
└── CMakeLists.txt	cmake 設定檔
└── cmake_modules	第三方函式庫查找腳本
└── FindJsoncpp.cmake	
└── FindMySQL.cmake	
└── FindSQLite3.cmake	
└── FindUUID.cmake	
└── config.json	應用設定檔，詳見設定檔章節
└── controllers	控制器原始檔目錄
└── filters	過濾器原始檔目錄
└── main.cc	主程式
└── models	資料庫 model 目錄，model 原始檔建立詳見 11.2.5
└── model.json	
└── tests	單元/整合測試目錄
└── test_main.cc	測試入口
└── views	視圖 csp 檔目錄，原始檔不需手動建立，編譯 時自動預處理產生

- **model 建立**

使用 `dg_ctl create model` 建立資料庫 model 原始檔，最後參數為 model 目錄，該目錄需有 `model.json` 設定檔，指定資料庫連線與映射資料表。

例如於上述專案目錄建立 models :

```
dg_ctl create model models
```

執行後會提示檔案將直接覆寫，輸入 **y** 後產生所有 model 檔案。

其他原始檔如需引用 model 類別，請 include model 標頭檔，例如：

```
#include "models/User.h"
```

注意 models 目錄名稱需包含，以區分同專案多資料來源。詳見 [ORM](#)。

壓力測試

可用 **dg_ctl press** 指令進行壓力測試，選項如下：

- **-n num** 設定請求數（預設 1）
- **-t num** 設定執行緒數（預設 1），設為 CPU 數可達最大效能
- **-c num** 設定併發連線數（預設 1）
- **-q** 不顯示進度（預設顯示）

例如測試 HTTP 伺服器：

```
dg_ctl press -n1000000 -t4 -c1000 -q http://localhost:8080/
dg_ctl press -n 1000000 -t 4 -c 1000
https://www.domain.com/path/to/be/tested
```

下一步: [AOP 面向切面程式設計](#)

面向切面程式設計 (AOP)

原文：[ENG-13-AOP-Aspect-Oriented-Programming.md](#)

AOP (面向切面程式設計, Aspect Oriented Programming) 是一種程式設計範式，目的是將橫切關注點 (cross-cutting concerns) 模組化（引自維基百科）。

受限於 C++ 語言特性，Drogon 並未如 Spring 提供彈性 AOP 解決方案，而是內建一組預定義的 joinpoint，使用者可透過框架的 AOP 介面將處理器（在 Drogon 稱為 advice）註冊到特定 joinpoint。

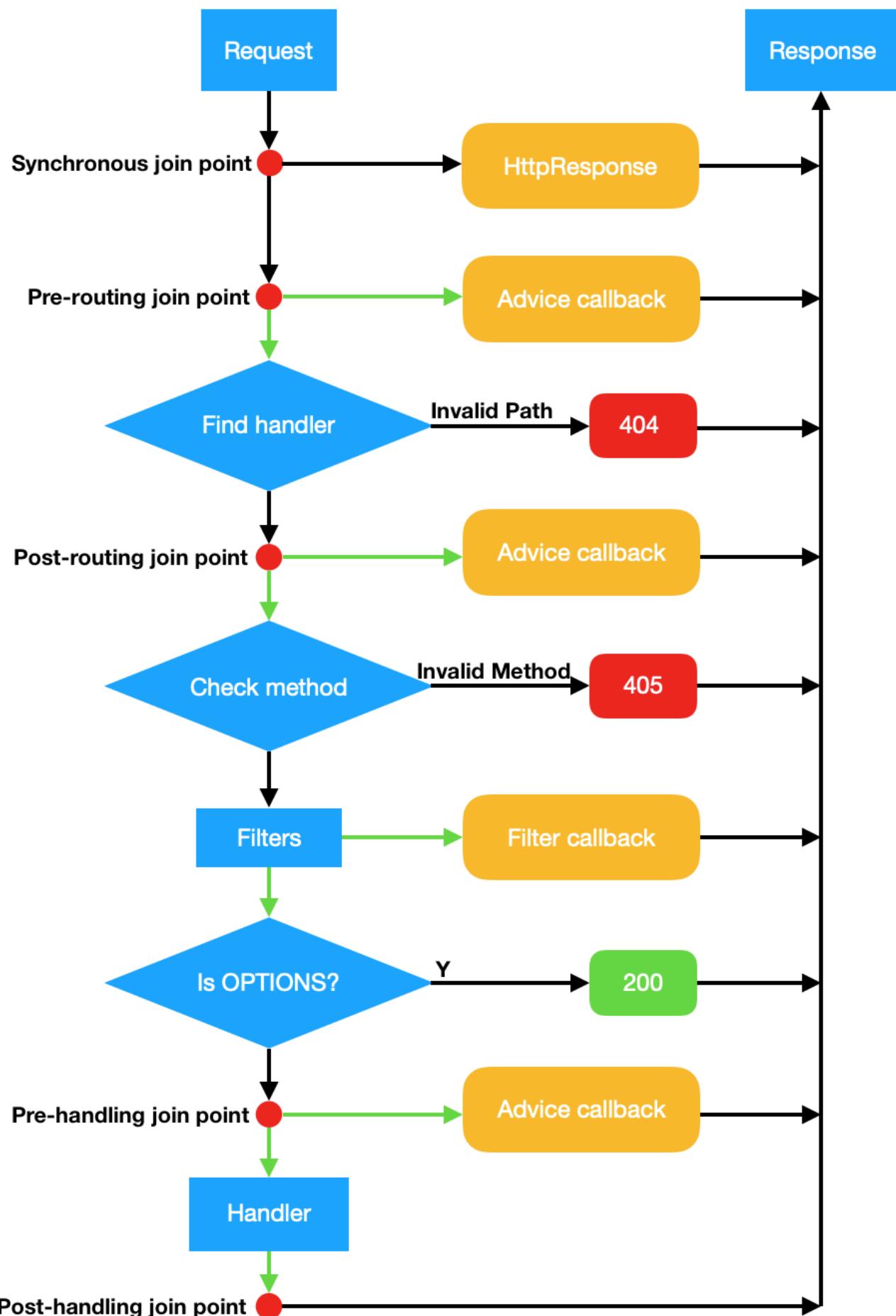
預定義 joinpoint

Drogon 提供七個 joinpoint。當應用執行至 joinpoint 時，會依序呼叫使用者註冊的 advice。各 joinpoint 說明如下：

- Beginning：於程式啟動時執行。即 app().run() 初始化完成後，所有 controller/filter/plugin、資料庫 client 均已建立，使用者可在此取得物件參照或執行初始化。此 advice 僅執行一次，簽名為 `void()`，註冊介面為 `registerBeginningAdvice`。
- NewConnection：每建立一個新的 TCP 連線時呼叫。簽名為 `bool(const trantor::InetAddress &, const trantor::InetAddress &)`，第一參數為遠端位址，第二為本地位址。回傳 false 則斷線。註冊介面為 `registerNewConnectionAdvice`。
- HttpResponseCreation：每建立一個 HTTP Response 物件時呼叫。簽名為 `void(const HttpResponsePtr &)`，參數為新建立的 Response，可統一處理所有 Response (如加 header)。此 joinpoint 影響所有 Response，包括 404 與內部錯誤，以及使用者產生的 Response。註冊介面為 `registerHttpResponseCreationAdvice`。
- Sync：位於 Http 請求處理前端。可回傳非空 Response 物件攔截請求。簽名為 `HttpRequestPtr(const HttpRequestPtr &)`。註冊介面為 `registerSyncAdvice`。
- Pre-Routing：於請求建立後、路徑比對前呼叫。advice 有兩種簽名：`void(const HttpRequestPtr &, AdviceCallback &&, AdviceChainCallback &&)` 及 `void(const HttpRequestPtr &)`，前者與 filter 的 `doFilter` 相同，可攔截或放行請求；後者無攔截功能但效能較佳，僅需統一處理時可用。註冊介面為 `registerPreRoutingAdvice`。
- Post-Routing：於請求路徑比對後立即呼叫，advice 簽名同上。註冊介面為 `registerPostRoutingAdvice`。
- Pre-Handling：於所有 filter 通過後、handler 執行前呼叫，advice 簽名同上。註冊介面為 `registerPostRoutingAdvice`。
- Post-Handling：handler 執行完畢、產生 Response 後呼叫。簽名為 `void(const HttpRequestPtr &, const HttpResponsePtr &)`，註冊介面為 `registerPostHandlingAdvice`。

AOP 示意圖

下圖顯示上述四個 joinpoint 在 HTTP 請求處理流程中的位置，紅點為 joinpoint，綠箭頭為非同步呼叫。



下一步: 效能測試

效能測試

[原文：ENG-14-Benchmarks.md](#)

作為 C++ Http 應用框架，效能理應是重點之一。本章介紹 Drogon 的簡易測試與成果。

測試環境

- 系統：Linux CentOS 7.4
- 設備：Dell 伺服器，CPU 為兩顆 Intel(R) Xeon(R) E5-2670 @ 2.60GHz，16 核心 32 執行緒
- 記憶體：64GB
- gcc 版本：7.3.0

測試方案與結果

僅測 Drogon 框架效能，故盡量簡化 controller 處理。僅建立一個 `HttpSimpleController`，註冊於 `/benchmark` 路徑，回傳 `<p>Hello, world!</p>`。Drogon 執行緒數設為 16。處理函式如下，原始碼可見於 `drogon/examples/benchmark`：

```
void BenchmarkCtrl::asyncHandleHttpRequest(const HttpRequestPtr &req,
    std::function<void (const HttpResponsePtr &)> &&callback)
{
    // 實作應用邏輯
    auto resp = HttpResponse::newHttpResponse();
    resp->setBody("<p>Hello, world!</p>");
    resp->setExpiredTime(0);
    callback(resp);
}
```

為比較，選用 nginx 進行測試，撰寫 `hello_world_module` 並以 nginx 原始碼編譯，`worker_processes` 設為 16。

測試工具為 `httpress`，一款高效能 HTTP 壓力測試工具。

調整 httpress 參數，每組參數測五次，記錄每秒處理請求數最大值與最小值。結果如下：

指令行	說明	Drogon(kQPS)	nginx(kQPS)
<code>httpress -c 100 -n 1000000 -t 16 -k -q URL</code>	100 連線，100 萬請求，16 執行緒，Keep-Alive	561/552	330/329
<code>httpress -c 100 -n 1000000 -t 12 -q URL</code>	100 連線，100 萬請求，12 執行緒，無 Keep-Alive	140/135	31/49
<code>httpress -c 1000 -n 1000000 -t 16 -k -q URL</code>	1000 連線，100 萬請求，16 執行緒，Keep-Alive	573/565	333/327
<code>httpress -c 1000 -n 1000000 -t 16 -q URL</code>	1000 連線，100 萬請求，16 執行緒，無 Keep-Alive	155/143	52/50

指令行	說明	Drogon(kQPS)	nginx(kQPS)
httpress -c 10000 -n 4000000 -t 16 -k -q URL	10000 連線，400 萬請求，16 執行緒，Keep-Alive	512/508	316/314
httpress -c 10000 -n 1000000 -t 16 -q URL	10000 連線，100 萬請求，16 執行緒，無 Keep-Alive	143/141	43/40

可見於 client 端啟用 Keep-Alive 時，drogon 可於單一連線多次請求下處理超過 50 萬次/秒，表現相當優異。若每次請求都建立新連線，CPU 時間多花於 TCP 連線建立與斷線，吞吐量降至 14 萬次/秒，屬合理範圍。

以上測試 Drogon 明顯優於 nginx。如有更精確測試，歡迎指正。

下圖為測試截圖：

```
[root@antao ~]# httpress -c 1000 -n 1000000 -t 16 -k -q http://localhost:7770/benchmark
TOTALS: 1000 connect, 1000000 requests, 1000000 success, 0 fail, 1000 (1000) real concurrency
TRAFFIC: 20 avg bytes, 142 avg overhead, 2000000 bytes, 142000000 overhead
TIMING: 1.768 seconds, 565456 rps, 89456 kbps, 1.8 ms avg req time
```

下一步: Coz 因果分析

使用 Coz 進行因果分析 (Causal profiling)

原文：[ENG-15-Coz.md](#)

Coz 可分析兩項指標：

- 吞吐量 (throughput)
- 延遲 (latency)

若要分析應用程式吞吐量，請於 cmake 啟用 `COZ_PROFILING` 選項，並以 `Debug` 或 `RelWithDebInfo` 模式編譯，確保可執行檔含除錯資訊。如此會於處理請求時插入 coz 進度點。全域延遲分析目前尚未支援，但可於使用者程式碼中進行。

編譯完成後，需以 coz profiler 執行可執行檔，例如：

```
coz run --- [執行檔路徑]
```

最後，應對應用程式進行壓力測試，建議涵蓋所有程式路徑並執行足夠時間（15 分鐘以上）。

分析結果會產生 `profile.coz` 檔於目前目錄。可用官方 `viewer` 開啟，或自官方 `git repo` 下載本地版。

Coz 亦支援以 `--source-scope <pattern>` 或 `-s <pattern>` 限定分析檔案範圍，十分實用。

更多資訊請參考：

- `coz run --help`
- [Git repo](#)
- [Coz 白皮書](#)

下一步：[Brotli 壓縮](#)

Brotli 壓縮說明

原文：[ENG-16-Brotli.md](#)

Dragon 原生支援 Brotli 靜態壓縮檔案，只要資源旁有對應的 Brotli 壓縮檔即可。

例如，若請求 `/path/to/asset.js`，Dragon 會自動尋找 `/path/to/asset.js.br`。

此功能預設於 `config.json` 中將 `br_static` 設為 `true`。

若需動態以 Brotli 壓縮，請於 `config.json` 設定 `use_brotli` 為 `true`。

若不需 Brotli 靜態壓縮，可將 `br_static` 設為 `false`，避免多餘的檔案檢查。

下一步：[協程](#)

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<void>
    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

Redis

[原文：ENG-18-Redis.md](#)

Dragon 支援 Redis，一款高速記憶體資料庫，可用於資料庫快取或訊息代理。Dragon 的 Redis 連線皆為非同步，確保高併發效能。

Redis 支援需安裝 [hiredis](#) 套件。若建置時未安裝 hiredis，則無法使用 Redis 功能。

建立 client

可透過 `app()` 動態建立與取得 Redis client：

```
app().createRedisClient("127.0.0.1", 6379);
...
// app.run() 之後
RedisClientPtr redisClient = app().getRedisClient();
```

亦可於設定檔建立 Redis client：

```
"redis_clients": [
  {
    //name: client 名稱，預設 'default'
    //"name": "",
    //host: 伺服器 IP，預設 127.0.0.1
    "host": "127.0.0.1",
    //port: 伺服器埠號，預設 6379
    "port": 6379,
    //passwd: 預設空字串
    "passwd": "",
    //db index: 預設 0
    "db": 0,
    //is_fast: 預設 false，true 時效能更高但不可呼叫同步介面，且僅限 IO 執
行緒與主執行緒使用
    "is_fast": false,
    //number_of_connections: 預設 1，is_fast 為 true 時為每 IO 執行緒
連線數，否則為總連線數
    "number_of_connections": 1,
    //timeout: 預設 -1.0 (秒)，命令執行逾時，0 或負值表示無逾時
    "timeout": -1.0
  }
]
```

使用 Redis

`execCommandAsync` 以非同步方式執行 Redis 指令，至少需三個參數：成功與失敗 callback，以及指令本身。指令可為 C 風格格式字串，後續為格式字串參數。例如設定 key name 為 drogon：

```
redisClient->execCommandAsync(
    []([](const drogon::nosql::RedisResult &r) {},  

    []([](const std::exception &err) {  

        LOG_ERROR << "something failed!!! " << err.what();  

    },  

    "set name drogon");
```

或設定 `myid` 為 `587d-4709-86e4`

```
redisClient->execCommandAsync(
    []([](const drogon::nosql::RedisResult &r) {},  

    []([](const std::exception &err) {  

        LOG_ERROR << "something failed!!! " << err.what();  

    },  

    "set myid %s", "587d-4709-86e4");
```

同樣 `execCommandAsync` 也可讀取 Redis 資料：

```
redisClient->execCommandAsync(
    []([](const drogon::nosql::RedisResult &r) {
        if (r.type() == RedisResultType::kNil)
            LOG_INFO << "找不到 key 'name' 對應的值";
        else
            LOG_INFO << "Name is " << rasString();
    },
    []([](const std::exception &err) {
        LOG_ERROR << "something failed!!! " << err.what();  

    },  

    "get name");
```

交易 (Transaction)

Redis 交易可一次執行多個指令，所有指令依序執行，期間不會插入其他 client 指令。注意交易非原子性，收到 `exec` 指令後即執行，若有指令失敗，剩餘指令仍會執行，且不支援 `rollback`。

`newTransactionAsync` 建立新交易，交易物件用法同 `RedisClient`，最後以 `RedisTransaction::execute` 執行交易。

```
redisClient->newTransactionAsync([](const RedisTransactionPtr &transPtr) {  

    transPtr->execCommandAsync(  

        []([](const drogon::nosql::RedisResult &r) { /* 指令成功 */ },  

        []([](const std::exception &err) { /* 指令失敗 */ },  

        "set name drogon");  
  

    transPtr->execute(  

        []([](const drogon::nosql::RedisResult &r) { /* 交易成功 */ },
```

```
 [] (const std::exception &err) { /* 交易失敗 */ });
});
```

協程

Redis client 支援協程。建議使用 GCC 11 或更新版本，並以 `cmake -DCMAKE_CXX_FLAGS="-std=c++20"` 啟用。詳見 [協程](#)。

```
try
{
    auto transaction = co_await redisClient->newTransactionCoro();
    co_await transaction->execCommandCoro("set zzz 123");
    co_await transaction->execCommandCoro("set mening 42");
    co_await transaction->executeCoro();
}
catch(const std::exception& e)
{
    LOG_ERROR << "Redis failed: " << e.what();
}
```

下一步: [測試框架](#)

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

DragonTest 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	<code>CHECK</code>	<code>CHECK_THROWS</code>	<code>CHECK_NO_THROW</code>	<code>CHECK_THROWS_AS</code>
<code>return</code>	<code>REQUIRE</code>	<code>REQUIRE_THROWS</code>	<code>REQUIRE_NO_THROW</code>	<code>REQUIRE_THROWS_AS</code>
<code>co_return</code>	<code>CO_REQUIRE</code>	<code>CO_REQUIRE_THROWS</code>	<code>CO_REQUIRE_NO_THROW</code>	<code>CO_REQUIRE_THROWS_AS</code>
終止程序	<code>MANDATE</code>	<code>MANDATE_THROWS</code>	<code>MANDATE_NO_THROW</code>	<code>MANDATE_THROWS_AS</code>

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，DragonTest 可整合至 CMake。[ParseAndAddDragonTests](#) 會將原始檔中的測試加入 CMake 的 CTest。

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

常見問題 FAQ

原文：[ENG-FAQ.md](#)

這裡列出一些常見問題及解答，並附有延伸說明。

Drogon 的執行緒模型與最佳實踐是什麼？

Drogon 會在執行 `app().run()` 時建立 HTTP 伺服器執行緒與資料庫執行緒，這些執行緒組成一個執行緒池。Drogon 採用序列化任務處理系統。因此，建議盡量使用非同步 API 或協程來開發。詳細說明請參考：[深入理解 Drogon 執行緒模型](#)。

認識 Dragon 的執行緒模型

原文：[ENG-FAQ-1-Understanding-dragon-threading-model.md](#)

Dragon 是一個高速的 C++ 網頁應用框架。它的高效部分原因在於不對底層執行緒模型做過度抽象，但這也常常造成一些困惑。常見的問題包括：為什麼某些阻塞呼叫後才送出回應、為什麼在同一個事件迴圈呼叫阻塞式網路函式會造成死結等。這篇文章旨在說明造成這些現象的條件，以及如何避免。

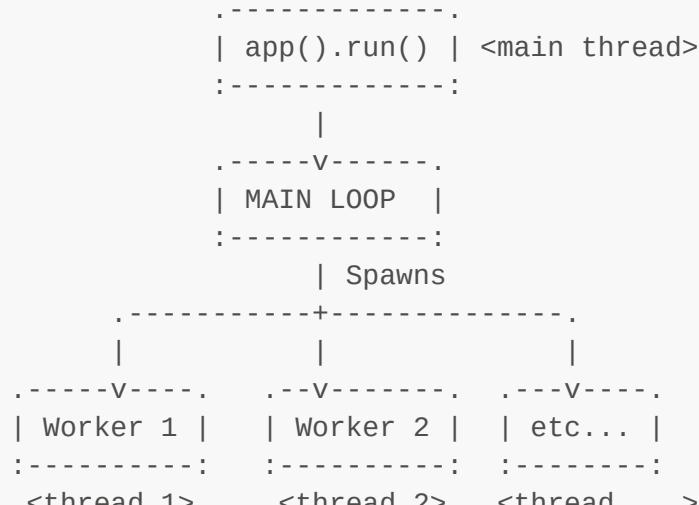
事件迴圈與執行緒

Dragon 以執行緒池運作，每個執行緒都有自己的事件迴圈（Event Loop）。事件迴圈是 Dragon 的核心。每個 Dragon 應用至少有兩個事件迴圈：主迴圈和工作迴圈。只要你沒做奇怪的事，主迴圈永遠在主執行緒（啟動 `main` 的執行緒）上執行，負責啟動所有工作迴圈。以 Hello World 範例來說，`app().run()` 會在主執行緒啟動主迴圈，然後產生三個工作執行緒/迴圈。

```
#include <dragon/dragon.h>
using namespace drogon;

int main()
{
    app().registerHandler("/", [](const HttpRequest& req
        , std::function<void (const HttpResponsePtr &) > &&callback) {
        auto resp = HttpResponse::newHttpResponse();
        resp->setBody("Hello wrold");
        callback(resp);
    });
    app().addListener("0.0.0.0", 8080);
    app().setNumThreads(3);
    app().run();
}
```

執行緒結構如下：



工作迴圈數量取決於多個因素，像是 HTTP 伺服器指定的執行緒數、非 fast DB 及 NoSQL 連線數等（fast 與非 fast 連線後述）。總之，Drogon 不只 HTTP 伺服器執行緒。每個事件迴圈本質上就是一個任務佇列，具備以下功能：

- 從任務佇列讀取並執行任務。你可以從任何執行緒提交任務到事件迴圈。提交任務完全無鎖（lock-free），在所有情況下不會造成資料競爭。事件迴圈一次只處理一個任務，因此任務有明確的執行順序。但如果排在長時間任務後，其他任務就會延遲。
- 監聽並分派它管理的網路事件
- 執行定時器（通常由使用者建立）

當上述都沒發生時，事件迴圈/執行緒就會阻塞等待。

```
// 在主迴圈排入兩個任務
trantor::EventLoop* loop = app().getLoop();
loop->queueInLoop([]{
    std::cout << "task1: 我要等 5 秒\n";
    std::this_thread::sleep_for(5s);
    std::cout << "task1: 哈囉！\n";
});
loop->queueInLoop([]{
    std::cout << "task2: 世界！\n";
});
```

所以執行上會先看到 task1：我要等 5 秒，暫停 5 秒後才會同時出現 task1：哈囉 和 task2：世界。

技巧 1：不要在事件迴圈裡呼叫阻塞式 IO，其他任務都得等它完成。

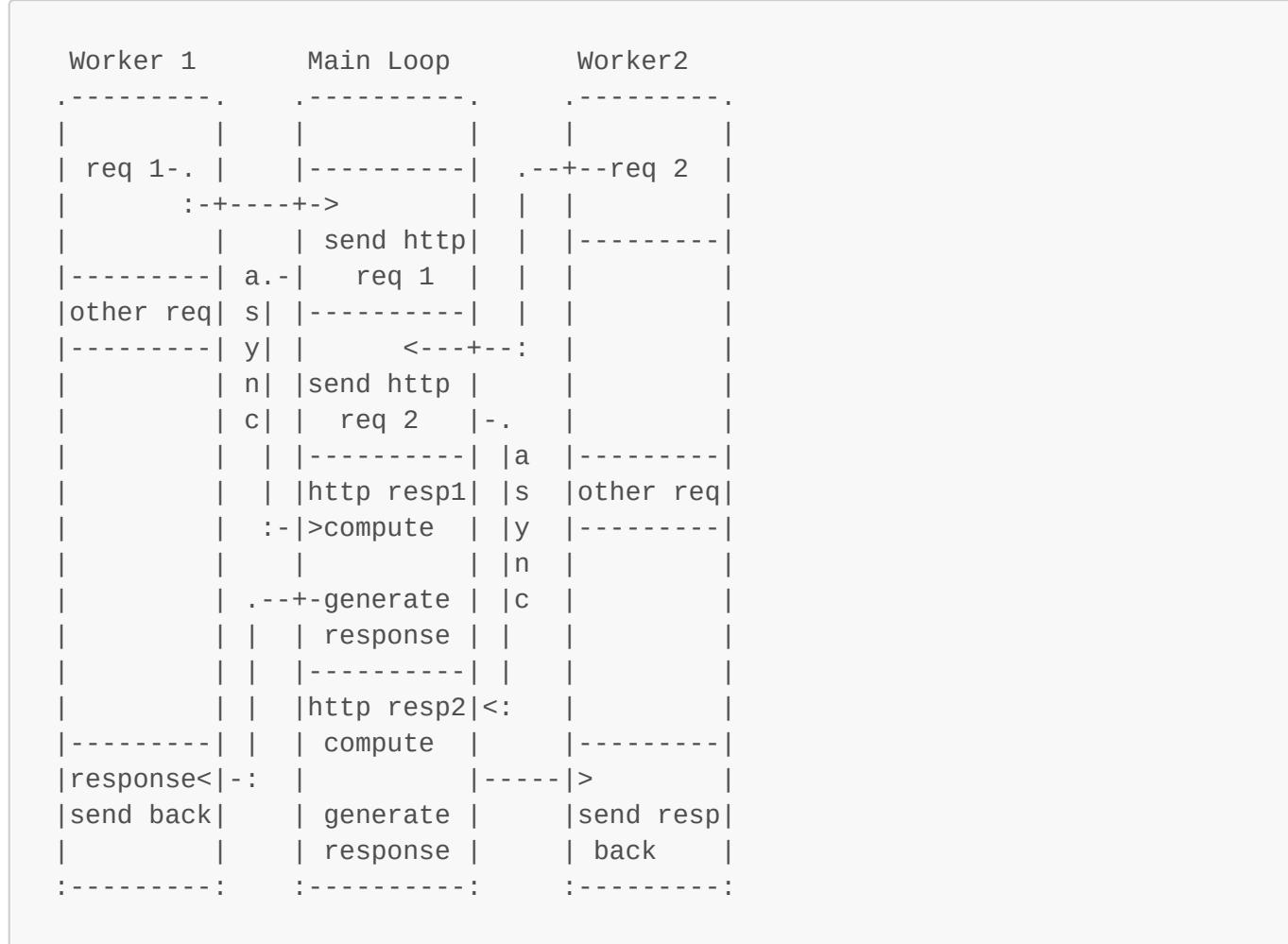
實務上的網路 IO

Drogon 幾乎所有東西都跟事件迴圈綁定，包括 TCP 連線、HTTP client ·DB client、資料快取等。為了避免競爭，所有 IO 都在所屬事件迴圈執行。如果從其他執行緒呼叫 IO，參數會被存起來並以任務方式提交到正確的事件迴圈。這有些影響，例如在 HTTP handler 裡呼叫遠端端點或 DB 查詢時，client 的 callback 不一定（通常不會）在 handler 的同一執行緒執行。

```
app().registerHandler("/send_req", [](const HttpRequest& req
, std::function<void (const HttpResponsePtr &) > &&callback) {
// 這個 handler 會在某個 HTTP 伺服器執行緒執行

// 建立在主迴圈運作的 HTTP client
auto client = HttpClient::newHttpClient("https://drogon.org",
app().getLoop());
auto request = HttpRequest::newHttpRequest();
client->sendRequest(request, [](ReqResult result, const HttpResponse&
resp) {
    // 這個 callback 會在主執行緒執行
});
});
```

所以如果你不注意，可能會塞爆事件迴圈，例如在主迴圈建立大量 HTTP client 並送出所有請求，或是在 DB callback 裡執行大量運算，導致其他 DB 查詢被延遲。



HTTP 伺服器也是一樣。如果回應是在其他執行緒產生（例如在 DB callback 裡）, 回應會排入所屬執行緒等候送出，而不是立即送出。

技巧 2：注意你的運算放在哪裡，不小心會拖慢整體效能。

事件迴圈死結

了解 Dargon 的設計後，應該不難理解如何讓事件迴圈死結。只要在同一迴圈提交遠端 IO 請求並等待結果（同步 API 就是這樣），它會提交 IO 請求然後等 callback。

```
app().registerHandler("/dead_lock", [](const HttpRequest& req
    , std::function<void (const HttpResponsePtr &)> &&callback) {
    auto currentLoop = app().getIOLoops()[app().getCurrentThreadIndex()];
    auto client = HttpClient::newHttpClient("https://dragon.org",
    currentLoop);
    auto request = HttpRequest::newHttpRequest();
    auto resp = client->sendRequest(resp); // 死結！呼叫同步介面
});
```

可以這樣想：

Some loop

```
graph TD; Start(( )) -->|new client| Client[ ]; Client -->|new request| Request[ ]; Request -->|send request| Send[ ]; Send -->|WAIT resp| Wait[ ]; Wait -->|read resp| Read[ ]; Read -->|oops| Oops[ ]; Oops -->|deadlock| Deadlock[ ]; Deadlock -. deadlock .-> Start;
```

其他像 DB 的 NoSQL callback 也一樣。幸好非 fast DB client 會在自己的執行緒運作，每個 client 都有自己的執行緒，所以在 HTTP handler 裡用同步查詢是安全的。但千萬不要在同一 client 的 callback 裡用同步查詢，否則還是會死結。

技巧 3：同步 API 既慢又危險，能不用就不用。如果真的要用，請確保 client 在不同執行緒。

Fast DB client

Drogon 以效能為優先，易用性其次。Fast DB client 會共用 HTTP 伺服器執行緒，這樣可以省去提交任務到其他執行緒的開銷，也避免 OS 的 context switch。但因此不能用同步查詢，否則會死結事件迴圈。

協程救援

Drogon 開發時遇到的兩難是：非同步 API 雖然高效但難用，同步 API 雖然好寫但容易出問題且慢。Lambda 宣告又冗長，語法也不美觀，程式碼不是自上而下而是充滿 callback。同步 API 雖然乾淨，但效能差。

```
// drogon 的非同步 DB API
auto db = app().getDbClient();
db->execSqlAsync("INSERT.....", [db, callback](auto result){
    db->execSqlAsync("UPDATE .....", [callback](auto result){
        // 成功處理
    },
    [callback](const DbException& e) {
        // 失敗處理
    }
});
```

```
    })
},
[callback](const DbException& e){
    // 失敗處理
})
```

對比：

```
// dragon 的同步 API，例外可自動由框架處理
db->execSqlSync("INSERT.....");
db->execSqlSync("UPDATE.....");
```

有沒有辦法兩者兼得？C++20 協程（coroutine）就是答案。它本質上是編譯器支援的 callback 包裝器，讓你的程式碼看起來像同步，但其實全程非同步。以下是協程寫法：

```
co_await db->execSqlCoro("INSERT.....");
co_await db->execSqlCoro("UPDATE.....");
```

看起來就像同步 API，但其實更好。你能享受非同步效能又能用同步語法。協程的原理超出本文範圍，維護者強烈建議能用協程就用（GCC >= 11, MSVC >= 16.25）。但協程不是萬靈丹，還是要注意事件迴圈塞爆和競爭問題。不過用協程寫非同步程式碼更好 debug、也更容易理解。

技巧 4：能用協程就用

小結

- 優先用 C++20 協程和 Fast DB 連線
- 同步 API 會拖慢甚至死結事件迴圈
- 如果一定要用同步 API，請確保 client 在不同執行緒