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：

```cpp
#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：

```cpp
#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