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 `drogon::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