Other languages: 繁體中文

Controller - WebSockerController

As the name implies, `WebSocketController` is used to process websocket logic. Websocket is a persistent HTTP-based connection scheme. At the beginning of the websocket, there is an HTTP format request and response exchange. After the websocket connection is established, all messages are transmitted on the websocket. The message is wrapped in a fixed format. There is no limit to the message content and the order in which messages are transmitted.

Generation

The source file of the `WebSocketController` can be generated by the `drogon_ctl` tool. The command format is as follows:

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

Suppose we want to implement a simple echo function through websocket, that is, the server simply sends back the message sent by the client. We can create the implementation class EchoWebsock of `WebSocketController` through `drogon_ctl`, as follows:

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

The command will generate two files of EchoWebsock.h and EchoWebsock.cc,as follows:

```
//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
    //list path definitions here;
    WS_PATH_LIST_END
};
```

```
//EchoWebsock.cc
#include "EchoWebsock.h"
void EchoWebsock::handleNewMessage(const WebSocketConnectionPtr
&wsConnPtr,std::string &&message)
{
    //write your application logic here
}
void EchoWebsock::handleNewConnection(const HttpRequestPtr &req,const
WebSocketConnectionPtr &wsConnPtr)
{
    //write your application logic here
}
void EchoWebsock::handleConnectionClosed(const WebSocketConnectionPtr
&wsConnPtr)
{
    //write your application logic here
}
```

Usage

- **Path Mapping**

  After edited:

```
//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
    //list path definitions here;
    WS_PATH_ADD("/echo");
    WS_PATH_LIST_END
};
```

```
//EchoWebsock.cc
#include "EchoWebsock.h"
void EchoWebsock::handleNewMessage(const WebSocketConnectionPtr
     &wsConnPtr,std::string &&message)
     {
         //write your application logic here
         wsConnPtr->send(message);
     }
     void EchoWebsock::handleNewConnection(const HttpRequestPtr &req,const
     WebSocketConnectionPtr &wsConnPtr)
     {
         //write your application logic here
     }
     void EchoWebsock::handleConnectionClosed(const WebSocketConnectionPtr
     &wsConnPtr)
     {
         //write your application logic here
     }
```

First, in this example, the controller is registered to the `/echo` path via the `WS_PATH_ADD` macro. The usage of the `WS_PATH_ADD` macro is similar to the macros of other controllers introduced earlier. One can also register the path with several Filters. Since websocket is handled separately in the framework, it can be repeated with the paths of the first two controllers （`HttpSimpleController` and `HttpApiController`） without affecting each other.

Secondly, in the implementation of the three virtual functions in this example, only the handleNewMessage has the substance, but simply sends the received message back to the client through the send interface.Compile this controller into the framework, you can see the effect, please test it yourself.

Note: Like the usual HTTP protocol, http websocket can be sniffed. If security is required, encryption should be provided by HTTPS. Of course, it is also possible for users to complete encryption and decryption on the server and client side, but HTTPS is more convenient. The underlying layer is handled by `drogon`, and users only need to care about business logic.

The user-defined websocket controller class inherits from the `drogon::WebSocketController` class template. The template parameter is a subclass type. The user needs to implement the following three virtual functions to process the establishment, shutdown, and messages of the websocket:

```
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);
```

Easy to know:

- `handleNewConnection` is called after the websocket is established. req is the setup request sent by the client. At this time, the framework has returned the response. What users can do is

to get some additional information through req, such as `token`. wsConn is a smart pointer to this websocket object, and the commonly used interface will be discussed later.
  - `handleNewMessage` is called after the websocket receives the new message. The message is stored in the message variable. Note that the message is the message payload. The framework has finished the decapsulation and decoding of the message. The user can directly process the message itself.
  - `handleConnectionClosed` is called after the websocket connection is closed, and the user can do some finishing work.

Interface

The common interfaces of the WebSocketConnection object are as follows:

```
//Send a websocket message, the encoding and encapsulation
//of the message are the responsibility of the framework
void send(const char *msg,uint64_t len);
void send(const std::string &msg);

//Local and remote addresses of the websocket
const trantor::InetAddress &localAddr() const;
const trantor::InetAddress &peerAddr() const;

//The connection state of the weosocket
bool connected() const;
bool disconnected() const;

//close websocket
void shutdown();//close write
void forceClose();//close

//set up and get the context of the websocket, and store some business data
from users.
//the any type means that you can store any type of object.
void setContext(const any &context);
const any &getContext() const;
any *getMutableContext();
```

Next: Middleware and Filter