Other languages: 繁體中文

Controller - HttpController

Generation

You can use the `drogon_ctl` command line tool to quickly generate custom controller class source files based on `HttpController`. The command format is as bellow:

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

We create one controller class named `User`, under namespace `demo v1`:

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

As you can see, two files have been added to the current directory, demo_v1_User.h and demo_v1_User.cc.

demo_v1_User.h is as follows:

```
#pragma once

#include <drogon/HttpController.h>

using namespace drogon;

namespace demo
{
namespace v1
{
class User : public drogon::HttpController<User>
{
  public:
    METHOD_LIST_BEGIN
    // use METHOD_ADD to add your custom processing function here;
    // METHOD_ADD(User::get, "/{2}/{1}", Get); // path is
/demo/v1/User/{arg2}/{arg1}
    // METHOD_ADD(User::your_method_name, "/{1}/{2}/list", Get); // path is
/demo/v1/User/{arg1}/{arg2}/list
    // ADD_METHOD_TO(User::your_method_name, "/absolute/path/{1}/{2}/list",
Get); // path is /absolute/path/{arg1}/{arg2}/list

    METHOD_LIST_END
    // your declaration of processing function maybe like this:
    // 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 is as follows:

```
#include "demo_v1_User.h"

using namespace demo::v1;

// Add definition of your processing function here
```

Usage

Let's edit the two files:

demo_v1_User.h is as follows:

```
#pragma once

#include <drogon/HttpController.h>

using namespace drogon;

namespace demo
{
namespace v1
{
class User : public drogon::HttpController<User>
{
  public:
    METHOD_LIST_BEGIN
    // use METHOD_ADD to add your custom processing function here;
    METHOD_ADD(User::login,"/token?userId={1}&passwd={2}",Post);
    METHOD_ADD(User::getInfo,"/{1}/info?token={2}",Get);
    METHOD_LIST_END
    // your declaration of processing function maybe like this:
    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 is as follows:

```
#include "demo_v1_User.h"

using namespace demo::v1;

// Add definition of your processing function here

void User::login(const HttpRequestPtr &req,
                 std::function<void (const HttpResponsePtr &)> &&callback,
                 std::string &&userId,
                 const std::string &password)
{
    LOG_DEBUG<<"User "<<userId<<" login";
    //Authentication algorithm, read database, verify, identify, etc...
    //...
    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";

    //Verify the validity of the token, etc.
    //Read the database or cache to get user information
    Json::Value ret;
    ret["result"]="ok";
    ret["user_name"]="Jack";
    ret["user_id"]=userId;
    ret["gender"]=1;
    auto resp=HttpResponse::newHttpJsonResponse(ret);
    callback(resp);
}
```

Each `HttpController` class can define many Http request handlers. Since the number of functions can be arbitrarily large, it is unrealistic to overload them with virtual functions. We need to register the handler itself (not the class) in the framework.

- **Path Mapping**

  The mapping from the URL path to the handler is done by macros. You can add a multipath map with the `METHOD_ADD`macro or the `ADD_METHOD_TO` macro. All `METHOD_ADD` and `ADD_METHOD_TO`

statements should be sandwiched between the `METHOD_LIST_BEGIN` and `METHOD_LIST_END` macro statements.

The `METHOD_ADD` macro automatically prefixes the namespace and class name in the path map. Therefore, in this example, the login function is registered to the `/demo/v1/user/token` path, and the getInfo function is registered to the `/demo/v1/user/xxx/info` path. Constraints are similar to the `PATH_ADD` macro of HttpSimpleController and are not described here.

When you use the `ADD_METHOD` macro and the class belongs to some namespace, you should add that namespace to the access url. In this example, use `http://localhost/demo/v1/user/token?userid=xxx&passwd=xxx` or `http://localhost/demo/v1/user/xxxxx/info?token=xxxx`.

The `ADD_METHOD_TO` macro does almost as much as the former, except that it does not automatically add any prefixes, i.e. the path registered by the macro is an absolute path.

We see that `HttpController` provides a more flexible path mapping mechanism - we can put a class of functions in a class.

In addition, you can see that the macros provide a method for parameter mapping. We can map the query parameters on the path to the parameter list of the function. The number of URL path parameters corresponds to the function parameter's position, this is very convenient. The common types which can be converted by string type all can be used as function parameters (such as std::string, int, float, double, etc.), and the drogon framework will automatically help you convert the type. This is very convenient for developing. Note that lvalue references must be of a const type.

The same path can be mapped multiple times, distinguished from each other by Http Method, which is legal and is a common practice of the Restful API, such as:

```
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
```

The placeholders of path parameters can be written in several ways:

- `{}`: The position on the path is the position of the function parameter, which indicates that the path parameter maps to the corresponding position of the handler parameters.
- `{1},{2}`: The path parameters with a number in them are mapped to the handler parameters specified by the number.
- `{anystring}`: Strings here have no practical effect, but can improve the readability of the program. Equivalent to `{}`.
- `{1:anystring},{2:xxx}`: The number before the colon represents the position. The string after the colon has no effect but can improve the readability of the program. Equivalent to the `{1}` and `{2}`.

The latter two are recommended, and if the path parameters and function parameters are in the same order, the third is enough. It is easy to see that the following are equivalent:

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

> **Note: Path matching is not case sensitive, but parameter names are case sensitive. Parameter values can be mixed in uppercase and lowercase and passed unchanged to the controller.**

- **Parameter Mapping**

Through the previous description, we know that the parameters on the path and the query parameters after the question mark can be mapped to the parameter list of the handler function. The type of the target parameter needs to meet the following conditions:

- Must be one of a value type, a constant left value reference, or a non-const right value reference. It cannot be a non-const lvalue reference. It is recommended to use an rvalue reference so that the user can dispose of it at will.
- Basic types such as int, long, long long, unsigned long, unsigned long long, float, double, long double, etc can be used as parameter types.
- std::string
- Any type that can be assigned using the `stringstream >>` operator.

> **In addition, the drogon framework also provides a mapping mechanism from the HttpRequestPtr object to any type of parameter.**. When the number of mapping parameters in your handler parameter list is more than the number of parameters on the path, the extra parameters will be converted by the HttpRequestPtr object. The user can define any type of conversion. The way to define this conversion is to specialize the `fromRequest` template (which is defined in the HttpRequest.h header file) in the drogon namespace, for example, say we need to make a RESTful interface to create a new user, we define the user's structure as follows:

```
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;
}

}
```

With the above definition and template specialization, we can define the path map and handler as follows:

```
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;
};
```

It can be seen that the third parameter of `myapp::User` type has no corresponding placeholder on the mapping path, and the framework regards it as a parameter converted from the `req` object and obtains this parameter through the user-specialized function template. This is very convenient for users.

Further, some users do not need to access the HttpRequestPtr object except for their custom type data. They can put the custom object in the position of the first parameter, and the framework will correctly complete the mapping such as the above example. It can also be written as follows:

```
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;
};
```

- **Multiple Path Mapping**

  Drogon supports the use of regular expressions in path mapping, which can be used outside the '{}' curly brackets. For example:

```
ADD_METHOD_TO(UserController::handler1,"/users/.*",Post); /// Match
any path prefixed with `/users/`
ADD_METHOD_TO(UserController::handler2,"/{name}/[0-9]+",Post);
///Match any path composed with a name string and a number.
```

- **Regular Expressions Mapping**

  The above method has limited support for regular expressions. If users want to use regular expressions freely, drogon provides the `ADD_METHOD_VIA_REGEX` macro to achieve this, such as:

```
ADD_METHOD_VIA_REGEX(UserController::handler1,"/users/(.*)",Post); ///
Match any path prefixed with `/users/` and map the rest of the path to
a parameter of the handler1.
ADD_METHOD_VIA_REGEX(UserController::handler2,"/.*([0-9]*)",Post); ///
Match any path that ends in a number and map that number to a
parameter of the handler2.
ADD_METHOD_VIA_REGEX(UserController::handler3,"/(?!data).*",Post); ///
Match any path that does not start with '/data'
```

  As can be seen, parameter mapping can also be done using regular expressions, and all strings matched by subexpressions will be mapped to the parameters of the handler in order.

> **It should be noted that when using regular expressions, you should pay attention to matching conflicts (multiple different handlers are matched). When conflicts happen in the same controller, drogon will only execute the first handler (the one registered in the framework first). When conflicts happen between different controllers, it is uncertain which handler will be executed. Therefore, users need to avoid these conflicts.**

Next: WebSocketController