Other languages: 繁體中文

View

Views Introduction

Although the front-end rendering technology is popular, the back-end application service only needs to return the corresponding data to the front-end. However, a good web framework should provide back-end rendering technology, so that the server program can dynamically generate HTML pages. Views can help users generate these pages. As the name implies, it is only responsible for doing the work related to the presentation, and the complex business logic should be handed over to the controller.

The earliest web applications embed HTML into the program code to achieve the purpose of dynamically generating HTML pages, but this is inefficient, not intuitive, and so on. So there are languages such as JSP, which are the opposite. , embed the program code into the HTML page. The drogon is of course the latter solution. However, it is obvious that since C++ is compiled and executed, we need to convert the page embedded in C++ code into a C++ source program to compile into the application. Therefore, drogon defines its own specialized CSP (C++ Server Pages) description language, using the command line tool drogon_ctl to convert CSP files into C++ source files for compilation.

Drogon's CSP

Drogon's CSP solution is very simple, we use special markup symbols to embed C++ code into the HTML page. among them:

- The content between the tags `<%inc` and `%>` is considered to be the part of the header file that needs to be referenced. Only the `#include` statement can be written here, such as `<%inc#include "xx.h" %>`, but many common header files are automatically included by drogon. The user basically does not use this tag;
- Everything between the tags `<%c++` and `%>` is treated as C++ code, such as `<c++ std:string name="drogon"; %>`;
- C++ code is generally transferred to the target source file intact, except for the following two special tags:
    - `@@` represents the data variable passed by the controller, from which you can get the content you want to display;
    - `$$` represents a stream object representing the content of the page, and the content to be displayed can be displayed on the page by the `<<` operator;
- The content sandwiched between the tags `[[` and `]]` is considered to be the variable name. The view will use the name as the key to find the corresponding variable from the data passed from the controller and output it to the page. Spaces before and after the variable name will be omitted. Paired `[[` and `]]` should be on the same line. And for performance reasons, only three string data types are supported(const char *, std::string and const std::string), other data types should be output in the above-mentioned way(by `$$`);
- The content sandwiched between the tags `{%` and `%}` is considered to be the name of a variable or an expression of the C++ program (not the keyword of the data passed by the controller), and the view will output the contents of the variable or the value of the expression to the page. It's easy to know

that `{%val.xx%}` is equivalent to `<%c++$$<<val.xx;%>`, but the former is simpler and more intuitive. Similarly, do not write two tags in separate lines;
- The content sandwiched between the tags `<%view` and `%>` is considered to be the name of the sub-view. The framework will find the corresponding sub-view and fill its contents to the location of the tag; the spaces before and after the view name will be ignored. Do not write `<%view` and `%>` in separate lines. Can use multiple levels of nesting, but not loop nesting;
- The content between the tags `<%layout` and `%>` is considered as the name of the layout. The framework will find the corresponding layout and fill the content of this view to a position in the layout (in the layout the placeholder `[[]]` marks this position); spaces before and after the layout name will be ignored, and `<%layout` and `%>` should not be written in separate lines. You can use multiple levels of nesting, but not loop nesting. One template file can only inherit from one base layout, multiple inheritance from different layouts is not supported.

The use of views

The http response of the drogon application is generated by the controller handler, so the response rendered by the view is also generated by the handler, generated by calling the following interface:

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

This interface is a static method of the HttpResponse class, which has two parameters:

- **viewName**: the name of the view, the name of the incoming csp file (**the extension can be omitted**);
- **data**: The controller's handler passes the data to the view. The type is `HttpViewData`. This is a special map. You can save and retrieve any type of object. For details, please refer to [HttpViewData API] (API-HttpViewData) Description

As you can see, the controller does not need to reference the header file of the view. The controller and the view are well decoupled; their only connection is the data variable.

A simple example

Now let's make a view that displays the parameters of the HTTP request sent by the browser in the returned html page.

This time we directly define the handler with the HttpAppFramework interface. In the main file, add the following code before calling the run() method:

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

```
                            data.insert("parameters",para);
                            auto
resp=HttpResponse::newHttpViewResponse("ListParameters.csp",data);
                            callback(resp);
                        });
```

The above code registers a lambda expression handler on the `/list_para` path, passing the requested parameters to the view display. Then, Go to the views folder and create a view file ListParameters.csp with the following contents:

```
<!DOCTYPE html>
<html>
<%c++
    auto
para=@@.get<std::unordered_map<std::string,std::string,utils::internal::Saf
eStringHash>>("parameters");
%>
<head>
    <meta charset="UTF-8">
    <title>[[ title ]]</title>
</head>
<body>
    <%c++ if(para.size()>0){%>
    <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><%c++ $$<<iter.second;%></td>
      </tr>
      <%c++}%>
    </table>
    <%c++ }else{%>
    <H1>no parameter</H1>
    <%c++}%>
</body>
</html>
```

We can use `drogon_ctl` command tool to convert ListParameters.csp into C++ source files as bellow:

```
drogon_ctl create view ListParameters.csp
```

After the operation is finished, two source files, ListParameters.h and ListParameters.cc, will appear in the current directory, which can be used to compile into the web application;

Recompile the entire project with CMake, run the target program webapp, you can test the effect in the browser, enter `http://localhost/list_para?p1=a&p2=b&p3=c` in the address bar, you can see the following page :

The html page rendered by the backend is simply added.

Automated processing of csp files

Note: If your project is create by the `drogon_ctl` command, the work described in this section is done automatically by `drogon_ctl`.

Obviously, it is too inconvenient to manually run the drogon_ctl command every time you modify the csp file. We can put the processing of drogon_ctl into the CMakeLists.txt file. Still use the previous example as an example. Let's assume that we put all the csp files In the views folder, CMakeLists.txt can be added as follows:

```
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()
```

Then add a new source file collection ${VIEWSRC} to the add_executable statement as follows:

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

Dynamic compilation and loading of views

Drogon provides a way to dynamically compile and load csp files during the application runtime, using the following interface:

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

The interface is a member method of `HttpAppFramework`, and the parameter is an array of strings representing a list of directories in which the view csp file is located. After calling this interface, drogon will automatically search for csp files in these directories. After discovering new or modified csp files, the source

files will be automatically generated, compiled into dynamic library files and loaded into the application. The application process does not need to be restarted. Users can experiment on their own and observe the page changes caused by the modification of csp file.

Obviously, this function depends on the development environment. If both drogon and webapp are compiled on this server, there should be no problem in dynamically loading the csp page.

> **Note: Dynamic views should not be compiled into the application statically. This means that if the view is statically compiled, it cannot be updated via dynamic view loading. You can create a directory outside the compilation folder and move views into it during development.**

> **Note: This feature is best used to adjust the HTML page during the development phase. In the production environment, it is recommended to compile the csp file directly into the target file. This is mainly for security and stability.**

> **Note: If a `symbol not found` error occurs while loading a dynamic view, please use the `cmake .. -DCMAKE_ENABLE_EXPORTS=on` to configure your project, or uncomment the last line (`set_property(TARGET ${PROJECT_NAME} PROPERTY ENABLE_EXPORTS ON)`) in your project's CMakeLists.txt, and then rebuild the project**

Next: Session