Other languages: **繁體中文**

Middleware and Filter

In HttpController's example, the getInfo method should check whether the user is logged in before returning the user's information. We can write this logic in the getInfo method, but obviously, checking the user's login membership is general logic which will be used by many interfaces, it should be extracted separately and configured before calling handler, which is what filters do.

Drogon's middleware uses the onion model. After the framework completes URL path matching, it sequentially invokes the middleware registered for that path. Within each middleware, users can choose to intercept or pass through the request and add pre-processing and post-processing logic.

If a middleware intercepts a request, it will not continue to the inner layers of the onion, and the corresponding handler will not be invoked. However, it will still go through the post-processing logic of the outer layer middleware.

Filters, in fact, are middleware that omit the post-processing operation. Middleware and filters can be used in combination when registering paths.

Built-in Middleware/Filter

Drogon contains the following common filters:

- `drogon::IntranetIpFilter`: allow HTTP requests from intranet IP only, or return the 404 page.
- `drogon::LocalHostFilter`: allow HTTP requests from 127.0.0.1 or ::1 only, or return the 404 page.

Custom Middleware/Filter

- **Middleware Definition**

  Users can customize the middleware, you need to inherit the `HttpMiddleware` class template, the template type is the subclass type, for example, if you want to enable cross-region support for come routes, you could define it as follows:

```
class MyMiddleware : public HttpMiddleware<MyMiddleware>
{
public:
    MyMiddleware(){};  // do not omit constructor

    void invoke(const HttpRequestPtr &req,
                MiddlewareNextCallback &&nextCb,
                MiddlewareCallback &&mcb) override
    {
        const std::string &origin = req->getHeader("origin");
        if (origin.find("www.some-evil-place.com") !=
std::string::npos)
```

```
        {
            // intercept directly
            mcb(HttpResponse::newNotFoundResponse(req));
            return;
        }
        // Do something before calling the next middleware
        nextCb([mcb = std::move(mcb)](const HttpResponsePtr &resp) {
            // Do something after the next middleware returns
            resp->addHeader("Access-Control-Allow-Origin", origin);
            resp->addHeader("Access-Control-Allow-
Credentials","true");
            mcb(resp);
        });
    }
};
```

You need to override the `invoke` virtual function of the parent class to implement the filter logic;

This virtual function has three parameters, which are:

- **req**: http request;
- **nextCb**：The callback function to enter the inner layer of the onion. Calling this function means invoking the next middleware or the final handler. When calling nextCb, it accepts another function as a parameter. This function will be called when returning from the inner layers, and the HttpResponsePtr returned from the inner layers will be passed as an argument to this function.
- **mcb**：The callback function to return to the upper layer of the onion. Calling this function means returning to the outer layer of the onion. If nextCb is skipped and only mcb is called, it means intercepting the request and directly returning to the upper layer.

- **Filter Definition**

Of course, users can customize the filter, you need to inherit the HttpFilter class template, the template type is the subclass type, for example, if you want to create a LoginFilter, you could define it as follows:

```
class LoginFilter:public drogon::HttpFilter<LoginFilter>
{
public:
    void doFilter(const HttpRequestPtr &req,
                  FilterCallback &&fcb,
                  FilterChainCallback &&fccb) override ;
};
```

You could create filter by the `drogon_ctl` command, see drogon_ctl.

You need to override the doFilter virtual function of the parent class to implement the filter logic;

This virtual function has three parameters, which are:

  - **req**: http request;
  - **fcb**: filter callback function, the function type is void (HttpResponsePtr), when the filter determines that the request is not valid, the specific response is returned to the browser through this callback;
  - **fccb**: filter chain callback function, the function type is void (), when the filter determines that the request is legal, tells drogon to call the next filter or the final handler through this callback;

  The specific implementation can refer to the implementation of the drogon built-in filter.

- **Middleware/Filter Registration**

  The registration of middlewares/filters is always accompanied by the registration of controllers.the macros (PATH_ADD, METHOD_ADD, etc.) mentioned earlier can add the name of one or more middlewares/filters at the end; for example, we change the registration line of the previous `getInfo` method to the following form:

  ```
  METHOD_ADD(User::getInfo,"/{1}/info?token=
  {2}",Get,"LoginFilter","MyMiddleware");
  ```

  After the path is successfully matched, the `getInfo` method will be called only when the following conditions were met:

  1. The request must be an HTTP Get request;
  2. The requesting party must have logged in;

  As you can see, the configuration and registration of middlewares/filters are very simple. The controller source file that registers middlewares does not need to include the middlewares's header file. The middleware and controller are fully decoupled.

  > **Note: If the middleware/filter is defined in the namespace, you must write the namespace completely when you register it.**

Next: View