Other languages: 繁體中文

Overview

Drogon is a C++17/20-based HTTP application framework. Drogon can be used to easily build various types of web application server programs using C++.

Drogon is the name of a dragon in the American TV series "Game of Thrones" that I really like.

Drogon's main application platform is Linux. It also supports Mac OS, FreeBSD and Windows.

Its main features are as follows:

- Use a non-blocking I/O network lib based on epoll (kqueue under macOS/FreeBSD) to provide high-concurrency, high-performance network IO, please visit the TFB Tests Results for more details;
- Provide a completely asynchronous programming mode;
- Support Http1.0/1.1 (server side and client side);
- Based on template, a simple reflection mechanism is implemented to completely decouple the main program framework, controllers and views.
- Support cookies and built-in sessions;
- Support back-end rendering, the controller generates the data to the view to generate the Html page. Views are described by CSP template files, C++ codes are embedded into Html pages through CSP tags. And the drogon command-line tool automatically generates the C++ code files for compilation;
- Support view page dynamic loading (dynamic compilation and loading at runtime);
- Provide a convenient and flexible routing solution from the path to the controller handler;
- Support filter chains to facilitate the execution of unified logic (such as login verification, Http Method constraint verification, etc.) before handling HTTP requests;
- Support https (based on OpenSSL);
- Support WebSocket (server side and client side);
- Support JSON format request and response, very friendly to the Restful API application development;
- Support file download and upload;
- Support gzip, brotli compression transmission;
- Support pipelining;
- Provide a lightweight command line tool, drogon_ctl, to simplify the creation of various classes in Drogon and the generation of view code;
- Support non-blocking I/O based asynchronously reading and writing database (PostgreSQL and MySQL(MariaDB) database);
- Support asynchronously reading and writing sqlite3 database based on thread pool;
- Support ARM Architecture;
- Provide a convenient lightweight ORM implementation that supports for regular object-to-database bidirectional mapping;
- Support plugins which can be installed by the configuration file at load time;
- Support AOP with build-in joinpoints.

Next: Install drogon

Other languages: 繁體中文

Installation

This section takes Ubuntu 24.04, CentOS 7.5, MacOS 12.2 as an example to introduce the installation process. Other systems are similar;

System Requirements

- The Linux kernel should be not lower than 2.6.9, 64-bit version;
- The gcc version is not less than 5.4.0, suggest to use version 11 or above;
- Use cmake as the build tool, and the cmake version should be not less than 3.5;
- Use git as the version management tool;

Library Dependencies

- Built-in
  - trantor, a non-blocking I/O C++ network library, also developed by the author of Drogon, has been used as a git repository submodule, no need to install in advance;
- Mandatory
  - jsoncpp, JSON's c++ library, the version should be **no less than 1.7**;
  - libuuid, generating c library of uuid;
  - zlib, used to support compressed transmission;
- Optional
  - boost, the version should be **no less than 1.61**, is required only if the C++ compiler does not support C++ 17 and if the STL doesn't fully support `std::filesystem`.
  - OpenSSL, after installed, drogon will support HTTPS as well, otherwise drogon only supports HTTP.
  - c-ares, after installed, drogon will be more efficient with DNS;
  - libbrotli, after installed, drogon will support brotli compression when sending HTTP responses;
  - the client development libraries of postgreSQL, mariadb and sqlite3, if one or more of them is installed, drogon will support access to the according database.
  - hiredis, after installed, drogon will support access to redis.
  - gtest, after installed, the unit tests can be compiled.
  - yaml-cpp, after installed, drogon will support config file with yaml format.

System Preparation Examples

Ubuntu 24.04

- Environment

```
sudo apt install git gcc g++ cmake
```

- jsoncpp

```
sudo apt install libjsoncpp-dev
```

- uuid

```
sudo apt install uuid-dev
```

- zlib

```
sudo apt install zlib1g-dev
```

- OpenSSL (Optional, if you want to support HTTPS)

```
sudo apt install openssl libssl-dev
```

Arch Linux

- Environment

```
sudo pacman -S git gcc make cmake
```

- jsoncpp

```
sudo pacman -S jsoncpp
```

- uuid

```
sudo pacman -S uuid
```

- zlib

```
sudo pacman -S zlib
```

- OpenSSL (Optional, if you want to support HTTPS)

```
sudo pacman -S openssl libssl
```

CentOS 7.5

- Environment

```
yum install git
yum install gcc
yum install gcc-c++
```

```
# The default installed cmake version is too low, use source installation
git clone https://github.com/Kitware/CMake
cd CMake/
./bootstrap && make && make install
```

```
# Upgrade gcc
yum install centos-release-scl
yum install devtoolset-11
scl enable devtoolset-11 bash
```

> **Note: Command `scl enable devtoolset-11 bash` only activate the new gcc temporarily until the session is end. If you want to always use the new gcc, you could run command `echo "scl enable devtoolset-11 bash" >> ~/.bash_profile`, system will automatically activate the new gcc after restarting.**

- jsoncpp

```
git clone https://github.com/open-source-parsers/jsoncpp
cd jsoncpp/
mkdir build
cd build
cmake ..
make && make install
```

- uuid

```
yum install libuuid-devel
```

- zlib

  ```
  yum install zlib-devel
  ```

- OpenSSL (Optional, if you want to support HTTPS)

  ```
  yum install openssl-devel
  ```

MacOS 12.2

- Environment

  All the essentials are inherent in MacOS, you only need to upgrade it.

  ```
  # upgrade gcc
  brew upgrade
  ```

- jsoncpp

  ```
  brew install jsoncpp
  ```

- uuid

  ```
  brew install ossp-uuid
  ```

- zlib

  ```
  yum install zlib-devel
  ```

- OpenSSL (Optional, if you want to support HTTPS)

  ```
  brew install openssl
  ```

Windows

- Environment (Visual Studio 2019) Install Visual Studio 2019 professional 2019, at least included these options:
    - MSVC C++ building tools
    - Windows 10 SDK
    - C++ CMake tools for windows
    - Test adaptor for Google Test

`conan` package manager could provide all dependencies that Drogon projector needs。If python is installed on system, you could install `conan` package manager via pip.

```
pip install conan
```

> of course you can download the installation file from `conan` official website to install it also.

Create`conanfile.txt`and add the following content to it:

- jsoncpp

    ```
    [requires]
    jsoncpp/1.9.4
    ```

- uuid

    No installation is required, the Windows 10 SDK already includes the uuid library.

- zlib

    ```
    [requires]
    zlib/1.2.11
    ```

- OpenSSL (Optional, if you want to support HTTPS)

    ```
    [requires]
    openssl/1.1.1t
    ```

Database Environment (Optional)

> **Note: These libraries below are not mandatory. You could choose to install one or more database according to your actual needs.**

> **Note: If you want to develop your webapp with database, please install the database develop environment first, then install drogon, otherwise you will encounter a `NO DATABASE FOUND` issue.**

- **PostgreSQL**

  PostgreSQL's native C library libpq needs to be installed. The installation is as follows:

  - `ubuntu 16`: `sudo apt-get install postgresql-server-dev-all`
  - `ubuntu 18`: `sudo apt-get install postgresql-all`
  - `ubuntu 24`: `sudo apt-get install postgresql-all`
  - `arch`: `sudo pacman -S postgresql`
  - `centOS 7`: `yum install postgresql-devel`
  - `MacOS`: `brew install postgresql`
  - `Windows conanfile`: `libpq/13.4`

- **MySQL**

  MySQL's native library does not support asynchronous read and write. Fortunately, MySQL also has a version of MariaDB maintained by the original developer community. This version is compatible with MySQL, and its development library supports asynchronous read and write. Therefore, Drogon uses the MariaDB development library to provide the right MySQL support, as a best practice， your operating system shouldn't install both Mysql and MariaDB at the same time.

  MariaDB installation is as follows：

  - `ubuntu 18.04`: `sudo apt install libmariadbclient-dev`
  - `ubuntu 24.04`: `sudo apt install libmariadb-dev-compat libmariadb-dev`
  - `arch`: `sudo pacman -S mariadb`
  - `centOS 7`: `yum install mariadb-devel`
  - `MacOS`: `brew install mariadb`
  - `Windows conanfile`: `libmariadb/3.1.13`

- **Sqlite3**

  - `ubuntu`: `sudo apt-get install libsqlite3-dev`
  - `arch`: `sudo pacman -S sqlite3`
  - `centOS`: `yum install sqlite-devel`
  - `MacOS`: `brew install sqlite3`
  - `Windows conanfile`: `sqlite3/3.36.0`

- **Redis**

  - `ubuntu`: `sudo apt-get install libhiredis-dev`
  - `ubuntu`: `sudo pacman -S redis`
  - `centOS`: `yum install hiredis-devel`
  - `MacOS`: `brew install hiredis`
  - `Windows conanfile`: `hiredis/1.0.0`

> **Note: Some of the above commands only install the development library. If you want to install a server also, please use Google search yourself.**

Drogon Installation

Assuming that the above environment and library dependencies are all ready, the installation process is very simple;

- **Install by source in Linux**

  ```
  cd $WORK_PATH
  git clone https://github.com/drogonframework/drogon
  cd drogon
  git submodule update --init
  mkdir build
  cd build
  cmake ..
  make && sudo make install
  ```

  > The default is to compile the debug version. If you want to compile the release version, the cmake command should take the following parameters:

  ```
  cmake -DCMAKE_BUILD_TYPE=Release ..
  ```

  After the installation is complete, the following files will be installed in the system（One can change the installation location with the CMAKE_INSTALL_PREFIX option）：

  - The header file of drogon is installed into /usr/local/include/drogon;
  - The drogon library file libdrogon.a is installed into /usr/local/lib;
  - Drogon's command line tool drogon_ctl is installed into /usr/local/bin;
  - The trantor header file is installed into /usr/local/include/trantor;
  - The trantor library file libtrantor.a is installed into /usr/local/lib;

- **Install by source in Windows**

  1. Download drogon source

     Open the Windows taskbar search box, search for x64 Native Tools, and select x64 Native Tools Command Prompt for VS 2019 as your command-line tool.

     ```
     cd $WORK_PATH
     git clone https://github.com/drogonframework/drogon
     cd drogon
     git submodule update --init
     ```

  2. Install dependencies

     install dependencies via conan:

```
mkdir build
cd build
conan profile detect --force
conan install .. -s compiler="msvc" -s compiler.version=193  -s
compiler.cppstd=17 -s build_type=Debug  --output-folder . --
build=missing
```

> Modify `conanfile.txt` to change the version of dependencies.

3. Compile and install

```
cmake ..  -DCMAKE_BUILD_TYPE=Debug -
DCMAKE_TOOLCHAIN_FILE="conan_toolchain.cmake" -
DCMAKE_POLICY_DEFAULT_CMP0091=NEW -DCMAKE_INSTALL_PREFIX="D:"
cmake --build . --parallel --target install
```

> **Note: Must keep build type same in conan and cmake.**

After the installation is complete, the following files will be installed in the system（One can change the installation location with the `CMAKE_INSTALL_PREFIX` option）：

- The header file of drogon is installed into `D:/include/drogon`;
- The drogon library file drogon.dll is installed into `D:/bin`;
- Drogon's command line tool drogon_ctl.exe is installed into `D:/bin`;
- The trantor header file is installed into `D:/include/trantor`;
- The trantor library file trantor.dll is installed into `D:/lib`;

Add `bin` and `cmake` directory to `path`：

```
D:\bin
```

```
D:\lib\cmake\Drogon
```

```
D:\lib\cmake\Trantor
```

- **Install by vcpkg in Windows**

Lazzy to read

Install vcpkg:

1. Install `vcpkg` by `git`.

```
git clone https://github.com/microsoft/vcpkg
cd vcpkg
./bootstrap-vcpkg.bat
```

> note: to update your vcpkg, you just need to type `git pull`

2. add `vpckg` to your windows environment variables ***path***.

3. Now check if vcpkg already installed properly, just type `vcpkg` or `vcpkg.exe`

Now Install Drogon:

1. To install drogon framework. Type:

   - 32-Bit: `vcpkg install drogon`
   - 64-Bit: `vcpkg install drogon:x64-windows`
   - extra : `vcpkg install jsoncpp:x64-windows zlib:x64-windows openssl:x64-windows sqlite3:x64-windows libpq:x64-windows libpqxx:x64-windows drogon[core,ctl,sqlite3,postgres,orm]:x64-windows`

   note:

   - if there's any package is/are uninstalled and you got error, just install that package. e.g.:

     zlib : `vcpkg install zlib` or `vcpkg install zlib:x64-windows` for 64-Bit

   - to check what already installed:

     `vcpkg list`

   - To use `drogon_ctl`, type `vcpkg install drogon[ctl]`(32-bit) or `vcpkg install drogon[ctl]:x64-windows`(64-bit). Type `vcpkg search drogon` for more installation feature options.

2. To add ***drogon_ctl*** command and dependencies, you need to add some variables. By following this guide, you just need to add:

   ```
   C:\dev\vcpkg\installed\x64-windows\tools\drogon
   ```

   ```
   C:\dev\vcpkg\installed\x64-windows\bin
   ```

   ```
   C:\dev\vcpkg\installed\x64-windows\lib
   ```

```
C:\dev\vcpkg\installed\x64-windows\include
```

```
C:\dev\vcpkg\installed\x64-windows\share
```

```
C:\dev\vcpkg\installed\x64-windows\debug\bin
```

```
C:\dev\vcpkg\installed\x64-windows\debug\lib
```

to your windows ***environment variables***. Then restart/re-open your ***powershell***.

3. reload/re-open your ***powershell***, then type: `drogon_ctl` or `drogon_ctl.exe` if:

```
usage: drogon_ctl [-v | --version] [-h | --help] <command> [<args>]
commands list:
create                  create some source files(Use 'drogon_ctl help create' for more information)
help                    display this message
press                   Do stress testing(Use 'drogon_ctl help press' for more information)
version                 display version of this tool
```

showed up, you are good to go.

> Note: you need to be familiar with building cpp libraries by using: independent `gcc` or `g++` (***msys2***, ***mingw-w64***, ***tdm-gcc***) or Microsoft Visual Studio compiler

> consider use ***make.exe/nmake.exe/ninja.exe*** as cmake generator since configuration and build behavior is same as _make* linux, if some devs using Linux/Windows and you are planning to deploy on Linux environment, it's less prone error when switching operating-system.

- **Use Docker Image**

  We also provide a pre-build docker image on the docker hub. All dependencies of Drogon and Drogon itself are already installed in the docker environment, where users can build Drogon-based applications directly.

- **Use Nix Package**

  There is a Nix package for Drogon which was released in version 21.11.

> **if you haven't installed Nix:** You can follow the instructions on the NixOS website.

You can use the package by adding the following `shell.nix` to your project root:

```
{ pkgs ? import <nixpkgs> {} }:
pkgs.mkShell {
  nativeBuildInputs = with pkgs; [
    cmake
  ];

  buildInputs = with pkgs; [
    drogon
  ];

}
```

Enter the shell by running `nix-shell`. This will install Drogon and enter you into an environment with all its dependencies.

The Nix package has a few options which you can configure according to your needs:

option	default value
sqliteSupport	true
postgresSupport	false
redisSupport	false
mysqlSupport	false

Here is an example of how you can change their values:

```
  buildInputs = with pkgs; [
    (drogon.override {
      sqliteSupport = false;
    })
  ];
```

- **Use CPM.cmake**

You can use CPM.cmake to include the drogon source code:

```
include(cmake/CPM.cmake)

CPMAddPackage(
    NAME drogon
    VERSION 1.7.5
    GITHUB_REPOSITORY drogonframework/drogon
    GIT_TAG v1.7.5
)

target_link_libraries(${PROJECT_NAME} PRIVATE drogon)
```

- **Include drogon source code locally**

  Of course, you can also include the drogon source in your project. Suppose you put the drogon under the third_party of your project directory (don't forget to update submodule in the drogon source directory). Then, you only need to add the following two lines to your project's cmake file:

  ```
  add_subdirectory(third_party/drogon)
  target_link_libraries(${PROJECT_NAME} PRIVATE drogon)
  ```

Next: Quick Start

Other languages: 繁體中文

Quick Start

Static Site

Let's start with a simple example that introduces the usage of drogon. In this example we create a project using the command line tool `drogon_ctl`:

```
drogon_ctl create project your_project_name
```

There are several useful folders in the project directory already:

```
├── build                         Build folder
├── CMakeLists.txt                Project cmake configuration file
├── config.json                   Drogon application configuration file
├── controllers                   The folder where the controller source
files are stored
├── filters                       The folder where the filter files are
stored
├── main.cc                       Main program
├── models                        The folder of the database model file
│   └── model.json
└── views                         The folder where view csp files are
stored
```

Users can put various files (such as controllers, filters, views, etc.) into the corresponding folders. For more convenience and less error, we strongly recommend that users create their own web application projects using the `drogon_ctl` command. See drogon_ctl for more details.

Let's look at the main.cc file:

```
#include <drogon/HttpAppFramework.h>
int main() {
    //Set HTTP listener address and port
    drogon::app().addListener("0.0.0.0",80);
    //Load config file
    //drogon::app().loadConfigFile("../config.json");
    //Run HTTP framework,the method will block in the internal event loop
    drogon::app().run();
    return 0;
}
```

Then build your project as below:

```
cd build
cmake ..
make
```

After the compilation is complete, run the target `./your_project_name`.

Now, we simply add one static file index.html to the Http root path:

```
echo '<h1>Hello Drogon!</h1>' >>index.html
```

The default root path is `"./"`, but could also be modified by config.json. See Configuration File for more details. Then you can visit this page by URL `"http://localhost"` or`"http://localhost/index.html"` (or the IP of the server where your wepapp is running).

Hello Drogon!

If the server cannot find the the page you have requested, it returns a 404 page: 404 page

> **Note: Make sure your server firewall has allowed the 80 port. Otherwise, you won't see these pages.(Another way is to change your port from 80 to 1024(or above) in case you get the error message below):**

```
FATAL Permission denied (errno=13) , Bind address failed at 0.0.0.0:80 -
Socket.cc:67
```

We could copy the directory and files of a static website to the startup directory of this running webapp, then we can access them from the browser. The file types supported by drogon by default are

- html
- js
- css
- xml
- xsl
- txt
- svg
- ttf
- otf
- woff2
- woff
- eot
- png
- jpg
- jpeg
- gif

- bmp
- ico
- icns

Drogon also provides interfaces to change these file types. For details, please refer to the HttpAppFramework API.

Dynamic Site

Let's see how to add controllers to this application， and let the controller respond with content.

One can use the drogon_ctl command line tool to generate controller source files. Let's run it in the `controllers` directory:

```
drogon_ctl create controller TestCtrl
```

As you can see, there are two new files, TestCtrl.h and TestCtrl.cc：

TestCtrl.h is as follows:

```
#pragma once
#include <drogon/HttpSimpleController.h>
using namespace drogon;
class TestCtrl:public drogon::HttpSimpleController<TestCtrl>
{
public:
    virtual void asyncHandleHttpRequest(const HttpRequestPtr &req,
                                        std::function<void (const
HttpResponsePtr &)> &&callback)override;
    PATH_LIST_BEGIN
    //list path definitions here;
    //PATH_ADD("/path","filter1","filter2",HttpMethod1,HttpMethod2...);
    PATH_LIST_END
};
```

TestCtrl.cc is as follows:

```
#include "TestCtrl.h"
void TestCtrl::asyncHandleHttpRequest(const HttpRequestPtr &req,
                                      std::function<void (const
HttpResponsePtr &)> &&callback)
{
    //write your application logic here
}
```

Let's edit the two files and let the controller handle the function response to a simple "Hello World!"

TestCtrl.h is as follows:

```
#pragma once
#include <drogon/HttpSimpleController.h>
using namespace drogon;
class TestCtrl:public drogon::HttpSimpleController<TestCtrl>
{
public:
    virtual void asyncHandleHttpRequest(const HttpRequestPtr &req,
                                        std::function<void (const
HttpResponsePtr &)> &&callback)override;
    PATH_LIST_BEGIN
    //list path definitions here

    //example
    //PATH_ADD("/path","filter1","filter2",HttpMethod1,HttpMethod2...);

    PATH_ADD("/",Get,Post);
    PATH_ADD("/test",Get);
    PATH_LIST_END
};
```

Use PATH_ADD to map the two paths '/' and '/test' to handler functions and add optional path constraints (here, the allowed HTTP methods).

TestCtrl.cc is as follows:

```
#include "TestCtrl.h"
void TestCtrl::asyncHandleHttpRequest(const HttpRequestPtr &req,
                                      std::function<void (const
HttpResponsePtr &)> &&callback)
{
    //write your application logic here
    auto resp=HttpResponse::newHttpResponse();
    //NOTE: The enum constant below is named "k200OK" (as in 200 OK), not
"k2000K".
    resp->setStatusCode(k200OK);
    resp->setContentTypeCode(CT_TEXT_HTML);
    resp->setBody("Hello World!");
    callback(resp);
}
```

Recompile this project with CMake, then run the target `./your_project_name`:

```
cd ../build
cmake ..
make
./your_project_name
```

Type `"http://localhost/"` or `"http://localhost/test"` in the browser address bar, and you will see "Hello World!" in the browser.

> **Note: If your server has both static and dynamic resources, Drogon uses dynamic resources first. In this example，the response to `GET http://localhost/` is `Hello World!` (from the `TestCtrl` controller file) instead of `Hello Drogon!` (from the static file index.html).**

We see that adding a controller to an application is very simple. You only need to add the corresponding source file. Even the main file does not need to be modified. This loosely coupled design is very effective for web application development.

> **Note: Drogon has no restrictions on the location of the controller source files. You could also save them in "./" (the project root directory), or you could even define a new directory in `CMakeLists.txt`. It is recommended to use the controllers directory for the convenience of management.**

Next: Controller - Introduction

Other languages: 繁體中文

Controller - Introduction

The controller is very important in web application development. This is where we will define our URLs, which HTTP methods are allowed, which filters will be applied and how requests will be processed and responded to. The drogon framework has helped us to handle the network transmission, Http protocol analysis and so on. We only need to pay attention to the logic of the controller; each controller object can have one or more processing functions (generally called handlers), and the interface of the function is generally defined as follows:

```
Void handlerName(const HttpRequestPtr &req,
                  std::function<void (const HttpResponsePtr &)> &&callback,
                 ...);
```

Where `req` is the object of the Http request (wrapped by the smart pointer), the `callback` is the callback function object that the framework passes to the controller, and the controller generates the response object (also wrapped by the smart pointer) and then passes the object to the drogon through the callback. Then the framework will send the response content to the browser for you. The last part `...` is a list of parameters. The drogon maps the parameters in the Http request to the corresponding parameter parameters according to the mapping rules. This is very convenient for application development.

Obviously, this is an asynchronous interface, one can call the callback after completing the time-consuming operation at other threads;

Drogon have three types controllers, HttpSimpleController, HttpController, and WebSocketController. When you use them, the corresponding class template needs to be inherited. For example, a custom class "MyClass" declaration of HttpSimpleController is as follows:

```
class MyClass:public drogon::HttpSimpleController<MyClass>
{
public:
    //TestController(){}
    virtual void asyncHandleHttpRequest(const HttpRequestPtr &req,
                                         std::function<void (const
HttpResponsePtr &)> &&callback) override;

    PATH_LIST_BEGIN
    PATH_ADD("/json");
    PATH_LIST_END
};
```

Controller life cycle

A controller registered to a drogon framework will have at most only one instance and will not be destroyed during the entire application run, so users can declare and use member variables in the controller class. Note that when the handler of the controller is called, it is in a multi-threaded environment (when the number of IO threads of the framework is configured to be greater than 1), if you need to access non-temporary variables, please do the concurrent protection work.

Next: HttpSimpleController

Other languages: 繁體中文

Controller - HttpSimpleController

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

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

We create one controller class named `TestCtrl`:

```
drogon_ctl create controller TestCtrl
```

As you can see, there are two new files, TestCtrl.h and TestCtrl.cc. Now, let's have a look at them:

TestCtrl.h：

```
#pragma once
#include <drogon/HttpSimpleController.h>
using namespace drogon;
class TestCtrl:public drogon::HttpSimpleController<TestCtrl>
{
public:
    virtual void asyncHandleHttpRequest(const HttpRequestPtr &req,
                                        std::function<void (const
HttpResponsePtr &)> &&callback)override;
    PATH_LIST_BEGIN
    //list path definitions here;
    //PATH_ADD("/path","filter1","filter2",HttpMethod1,HttpMethod2...);
    PATH_LIST_END
};
```

TestCtrl.cc:

```
#include "TestCtrl.h"
void TestCtrl::asyncHandleHttpRequest(const HttpRequestPtr &req,
                                      std::function<void (const
HttpResponsePtr &)> &&callback)
{
    //write your application logic here
}
```

Each HttpSimpleController class can only define one Http request handler, and it is defined by a virtual function override.

The route (or called mapping) from the URL path to the handler is done by a macro. You could add multipath mappings with the `PATH_ADD` macro. All `PATH_ADD` statements should be set between the `PATH_LIST_BEGIN` and `PATH_LIST_END` macro statements.

The first parameter is the path to be mapped, and parameters beyond the path are constraints on this path. Currently, two types of constraints are supported. One is the `HttpMethod` enum Type, which means the Http method allowed. The other type is the name of the `HttpFilter` class. One can configure any number of these two types of constraints, and there are no order requirements for them. For Filter, please refer to Middleware and Filter.

Users can register the same Simple Controller to multiple paths, or register multiple Simple Controllers on the same path (using different HTTP methods).

You could define an HttpResponse class variable, and then use the callback() to return it:

```
    //write your application logic here
    auto resp=HttpResponse::newHttpResponse();
    resp->setStatusCode(k200OK);
    resp->setContentTypeCode(CT_TEXT_HTML);
    resp->setBody("Your Page Contents");
    callback(resp);
```

> **The mapping from the above path to the handler is done at compile time. In fact, the drogon framework also provides an interface for runtime completion mapping. The runtime mapping allows the user to map or modify the mapping through configuration files or other user interfaces without recompiling this program (For performance reasons, it is forbidden to add any controller mapping after running the app().run() method).**

Next: HttpController

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

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

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

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 :

view 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

Other languages: **繁體中文**

Session

`Session` is an important concept of the web application. It is used to save the state of the client on the server. Generally, it cooperates with the browser's `cookie`, and drogon provides support for the session. Drogon **close** the session selection by default, you can also close or open it through the following interface:

```
        void disableSession();
        void enableSession(const size_t timeout=0, Cookie::SameSite sameSite=Cooie::SameSite::kNull);
```

The above methods are all called through the `HttpAppFramework` singleton. The timeout parameter represents the time when the session is invalid. The unit is second. The default value is 1200. That is, if the user does not access the web application for more than 20 minutes, the corresponding session will be invalid. Setting timeout to 0 means that drogon will retain the user's session for the entire lifetime; The sameSite parameter changes the SameSite attribute of the Set-Cookie HTTP response header.

Make sure your client supports cookies before opening the session feature. Otherwise, drogon will create a new session for each request without `SessionID` cookie, which will waste memory and computing resources.

Session object

The session object type of drogon is `drogon::Session`, which is very similar to `HttpViewData`. It can access any type of object through keywords; support concurrent reading and writing; please refer to the description of Session class for specific usage;

The drogon framework will pass the session object to the `HttpRequest` object and pass it to the user. The user can get the Session object through the following interface of the `HttpRequest` class.

```
SessionPtr session() const;
```

The interface returns a smart pointer of the `Session` object, through which various objects can be accessed;

Examples of sessions

We add a feature that requires session support. For example, we want to limit the user's access frequency. After a visit, if it is accessed again within 10 seconds, it will return an error, otherwise it will return ok. We need to record the last access time in the session, and then compare it with the time of this visit, you can achieve this function.

We create a Filter to implement this function, assuming the class name is TimeFilter, the implementation is as follows:

```
#include "TimeFilter.h"
#include <trantor/utils/Date.h>
#include <trantor/utils/Logger.h>
#define VDate "visitDate"
void TimeFilter::doFilter(const HttpRequestPtr &req,
                          FilterCallback &&cb,
                          FilterChainCallback &&ccb)
{
    trantor::Date now=trantor::Date::date();
    LOG_TRACE<<"";
    if(req->session()->find(VDate))
    {
        auto lastDate=req->session()->get<trantor::Date>(VDate);
        LOG_TRACE<<"last:"<<lastDate.toFormattedString(false);
        req->session()->modify<trantor::Date>(VDate,
                                        [now](trantor::Date &vdate) {
                                            vdate = now;
                                        });
        LOG_TRACE<<"update visitDate";
        if(now>lastDate.after(10))
        {
            //10 sec later can visit again;
            ccb();
            return;
        }
        else
        {
            Json::Value json;
            json["result"]="error";
            json["message"]="Access interval should be at least 10
seconds";
            auto res=HttpResponse::newHttpJsonResponse(json);
            cb(res);
            return;
        }
    }
    LOG_TRACE<<"first access,insert visitDate";
    req->session()->insert(VDate,now);
    ccb();
}
```

We then register a lambda expression to the `/slow` path and attach the TimeFilter with the following code:

```
drogon::HttpAppFramework::instance()
            .registerHandler("/slow",
                            [=](const HttpRequestPtr &req,
                                std::function<void (const HttpResponsePtr
&)> &&callback)
                            {
                                Json::Value json;
                                json["result"]="ok";
                                auto
resp=HttpResponse::newHttpJsonResponse(json);
                                callback(resp);
                            },
                            {Get,"TimeFilter"});
```

Call the framework interface to open the session:

```
drogon::HttpAppFramework::instance().enableSession(1200);
```

Recompile the entire project with CMake, run the target program webapp, and you can see the effect through the browser.

Next: Database

Other languages: 繁體中文

Database - General

General

Drogon has built-in database read/write engine. The operation of database connection is based on non-blocking I/O technology. Therefore, the application works in an efficient non-blocking asynchronous mode from the bottom to the upper layer, which ensures Drogon's high concurrency performance. Currently, Drogon supports PostgreSQL and MySQL databases. If you want to use a database, the development environment of the corresponding database must be installed first. Drogon will automatically detect the header files and library files of these libraries and compile the corresponding parts. For the preparation of the database development environment, see Development Environment.

Drogon supports the sqlite3 database in order to support lightweight applications. The asynchronous interface is implemented through the thread pool, which is the same as the interfaces of the aforementioned databases.

DbClient

The basic class of Drogon's database is `DbClient` (this is an abstract class, the specific type depends on the interface that constructs it). Unlike a generic database interface, a `DbClient` object does not represent a single database connection. It can contain one or more database connections, so you can think of it as a **connection pool object**.

`DbClient` provides both synchronous and asynchronous interfaces. The asynchronous interface also supports both blocking and non-blocking modes. Of course, for the cooperation with the Drogon asynchronous framework, it is recommended that you use the the asynchronous interface with non-blocking mode.

Usually, when an asynchronous interface is called, `DbClient` will randomly select one of the idle connections it manages to perform related query operations. When the result returns, `DbClient` will process the data and return it to the caller through the callback function object; Without an idle connection, the execution content will be cached. Once a connection has executed its own sql request, the pending command will be fetched from the cache to execute.

For details on `DbClient`, see DbClient.

Transaction

The transaction object can be generated by `DbClient` to support transaction operations. In addition to the extra `rollback()` interface, the transaction object is basically the same as `DbClient`. The transaction class is `Transaction`. For details of the `Transaction` class, see Transaction.

ORM

Drogon also provides support for **ORM**. Users can use the drogon_ctl command to read the tables in the database and generate the corresponding model source code. Then, execute the database operations of these models through the `Mapper<MODEL>` class template. Mapper provides simple and convenient interfaces for standard database operations, allowing users to make the additions, deletions, and changes to the table without writing sql statements. For **ORM**, please refer to ORM

Next: DbClient

Other languages: **繁體中文**

Database - DbClient

DbClient Object Construction

There are two ways to construct a DbClient object. One is through the static method of the DbClient class. You can see the definition in the DbClient.h header file, as follows:

```
#if USE_POSTGRESQL
    static std::shared_ptr<DbClient> newPgClient(const std::string &connInfo, const size_t connNum);
#endif
#if USE_MYSQL
    static std::shared_ptr<DbClient> newMysqlClient(const std::string &connInfo, const size_t connNum);
#endif
```

Use the above interface to get the smart pointer of the DbClient implementation object. The parameter connInfo is a connection string. Set a series of connection parameters in the form of key=value. For details, please refer to the comments in the header file. The parameter connNum is the number of database connections of DbClient, which has a key impact on concurrency. Please set it according to the actual situation.

The object obtained by the above method, the user has to find a way to persist it, such as putting it in some global container. **Creating a temporary object and then releasing it after use is a very unrecommended solution** for the following reasons:

- This will waste time creating connections and disconnections, increasing system latency;
- The interface is also a non-blocking interface. That is to say, when the user gets the DbClient object, the connection managed by the it has not been established yet. The framework does not (intentionally) provide a callback interface for successful connection establishment. Do you still have to sleep before starting the query?? This is contrary to the original intention of the asynchronous framework.

Therefore, DbClient objects should be built at the beginning of the program and held and used throughout the life time. Obviously, this work can be done entirely by the framework. So the drogon framework provides the second build method, which is built by configuration file or the `createDbClient()` method. For the configuration method of the configuration file, see db_clients.

When needed, the DbClient smart pointer is obtained through the interface of the framework. The interface is as follows:

```
orm::DbClientPtr getDbClient(const std::string &name = "default");
```

The parameter name is the value of the name configuration option in the configuration file to distinguish multiple different DbClient objects of the same application. The connections managed by DbClient are always reconnected, so users don't need to care about the connection status. They are almost always connected. **Note**: This method cannot be called before running app.run(), otherwise the user will get an empty shared_ptr.

Execution Interface

DbClient provides several different interfaces to users, as listed below:

```
/// Asynchronous mothod
template <
        typename FUNCTION1,
        typename FUNCTION2,
        typename... Arguments>
void execSqlAsync(const std::string &sql,
                  FUNCTION1 &&rCallback,
                  FUNCTION2 &&exceptCallback,
                  Arguments &&... args) noexcept;

/// Asynchronous mothod by 'future'
template <typename... Arguments>
std::future<const Result> execSqlAsyncFuture(const std::string &sql,
                                             Arguments &&... args)
noexcept;

/// Synchronous method
template <typename... Arguments>
const Result execSqlSync(const std::string &sql,
                         Arguments &&... args) noexcept(false);

/// Streaming-type method
internal::SqlBinder operator<<(const std::string &sql);
```

Since the number and type of binding parameters cannot be predetermined, these methods are function templates.

The properties of these methods are shown in the following table:

Methods	Synchronous/Asynchronous	Blocking/Non-blocking	Exception
void execSqlAsync	Asynchronous	Non-blocking	Will not throw an exception
std::future<const Result> execSqlAsyncFuture	Asynchronous	Block when calling the get method of the future	May throw an exception when calling the get method of the future

Methods	Synchronous/Asynchronous	Blocking/Non-blocking	Exception
const Result execSqlSync	Synchronous	Blocking	May throw an exception
internal::SqlBinder operator<<	Asynchronous	Default non-blocking	Will not throw an exception

You may be confused about the combination of asynchronous and blocking. In general, the synchronization method involving network IO is blocking, and the asynchronous method is non-blocking. However, the asynchronous method can also work in blocking mode, meaning that this method will Block until the callback function has finished executing. When the asynchronous method of DbClient works in blocking mode, the callback function will be executed in the thread of the caller, and then the method will return.

If your application involves high-concurrency scenarios, please use asynchronous non-blocking methods. If it is in a low concurrent scene (such as a network device management page), you can choose synchronization methods for convenience and intuitiveness.

- **execSqlAsync**

  ```
  template <typename FUNCTION1,
          typename FUNCTION2,
          typename... Arguments>
  void execSqlAsync(const std::string &sql,
                    FUNCTION1 &&rCallback,
                    FUNCTION2 &&exceptCallback,
                    Arguments &&... args) noexcept;
  ```

  This is the most commonly used asynchronous interface, working in non-blocking mode;

  The parameter `sql` is a string of sql statements. If there are placeholders for binding parameters, use the placeholder rules of the corresponding database. For example, PostgreSQL placeholders are $1, $2 ..., while MySQL placeholders are `?` without any numbers.

  The indefinite parameter `args` represents the bound parameter, which can be zero or more. The number of parameters is the same as the number of placeholders in the sql statement. The types can be the following:

  - Integer type: can be an integer of various word lengths, and should match the database field type;
  - Floating point type: can be `float` or `double`, should match the database field type;
  - String type: can be `std::string` or `const char[]`, corresponding to the string type of the database or other types that can be represented by strings;
  - Date type: `trantor::Date` type, corresponding to the database date, datetime, timestamp types.
  - Binary type: `std::vector<char>` type, corresponding to PostgreSQL's byte type or Mysql's blob type;

These parameters can be left or right, can be variables or literal constants, and users are free to use them.

The parameters rCallback and exceptCallback represent the result callback function and the exception callback function, respectively, which have a fixed definition, as follows:

- The result callback function: the call type is void (const Result &), various callable objects conforming to this call type, std::function, lambda, etc. can be passed as parameters;
- Exception callback function: the call type is void (const DrogonDbException &), which can pass various callable objects that are consistent with this call type;

After the execution of sql is successful, the execution result is wrapped by the Result class and passed to the user through the result callback function; if there is any exception in the sql execution, the exception callback function is executed, and the user can obtain the exception information from the DrogonDbException object.

Let us give an example:

```
auto clientPtr = drogon::app().getDbClient();
clientPtr->execSqlAsync("select * from users where org_name=$1",
                             [](const drogon::orm::Result &result) {
                                 std::cout << result.size() << " rows
selected!" << std::endl;
                                 int i = 0;
                                 for (auto row : result)
                                 {
                                     std::cout << i++ << ": user name
is " << row["user_name"].as<std::string>() << std::endl;
                                 }
                             },
                             [](const DrogonDbException &e) {
                                 std::cerr << "error:" <<
e.base().what() << std::endl;
                             },
                             "default");
```

From the example we can see that the Result object is a std standard compatible container, support iterators, you can get the object of each row through the range loop. the various interfaces of Result, Row and Field objects, please refer to the source code.

The DrogonDbException class is the base class for all database exceptions. Please refer to the comments in the source code.

- **execSqlAsyncFuture**

```
template <typename... Arguments>
std::future<const Result> execSqlAsyncFuture(const std::string &sql,
                                             Arguments &&... args)
noexcept;
```

The asynchronous future interface omits the two callback parameters of the previous interface. Calling this interface will immediately return a future object. The user must call the get() method of the future object to get the returned result. The exception is obtained through the try/catch mechanism. if the get() method isn't in the `try/catch`, and there is no `try/catch` in the entire call stack, the program will exit when the sql execution exception occurs.

For example:

```
auto f = clientPtr->execSqlAsyncFuture("select * from users where
org_name=$1",
                                        "default");
try
{
    auto result = f.get(); // Block until we get the result or catch
the exception;
    std::cout << result.size() << " rows selected!" << std::endl;
    int i = 0;
    for (auto row : result)
    {
        std::cout << i++ << ": user name is " <<
row["user_name"].as<std::string>() << std::endl;
    }
}
catch (const DrogonDbException &e)
{
    std::cerr << "error:" << e.base().what() << std::endl;
}
```

- **execSqlSync**

```
template <typename... Arguments>
const Result execSqlSync(const std::string &sql,
                         Arguments &&... args) noexcept(false);
```

Synchronous interface is the most simple and intuitive, the input parameters are sql string and bound parameters, return a Result object, the call will block the current thread, and throw an exception when an error occurs, so also pay attention to catch exception with `try/catch`.

E.g:

```
try
{
    auto result = clientPtr->execSqlSync("update users set
user_name=$1 where user_id=$2",
                                         "test",
                                        1); // Block until we get the
result or catch the exception;
    std::cout << result.affectedRows() << " rows updated!" <<
std::endl;
}
catch (const DrogonDbException &e)
{
    std::cerr << "error:" << e.base().what() << std::endl;
}
```

- **operator<<**

```
internal::SqlBinder operator<<(const std::string &sql);
```

The streaming interface is special. It inputs the sql statement and parameters in turn through the `<<` operator, and specifies the result callback function and the exception callback function through the `>>` operator. For example, the previous example of selecting, using the streaming interface is Look like this:

```
*clientPtr  << "select * from users where org_name=$1"
            << "default"
            >> [](const drogon::orm::Result &result)
                {
                    std::cout << result.size() << " rows selected!" <<
std::endl;
                    int i = 0;
                    for (auto row : result)
                    {
                        std::cout << i++ << ": user name is " <<
row["user_name"].as<std::string>() << std::endl;
                    }
                }
            >> [](const DrogonDbException &e)
                {
                    std::cerr << "error:" << e.base().what() <<
std::endl;
                };
```

This usage is completely equivalent to the first asynchronous non-blocking interface, and which interface is used depends on the user's usage habits. If you want it to work in blocking mode, you can use `<<` to enter a `Mode::Blocking` parameter, which is not described here.

In addition, the streaming interface has a special usage. Using a special result callback, the framework can pass the result to the user row by row. The call type of this callback is as follows:

```
void (bool,Arguments...);
```

When the first bool parameter is true, it means that the result is a empty row, that is, all the results have been returned, this is the last callback; Behind is a series of parameters, corresponding to the value of each column of a row of records, the framework will do type conversion, of course, the user should also pay attention to the type of matching. These types can be const-type lvalue references, or rvalue references, and of course value types.

Let's rewrite the previous example with this callback:

```
int i = 0;
*clientPtr  << "select user_name, user_id from users where 
org_name=$1"
            << "default"
            >> [&i](bool isNull, const std::string &name, int64_t id)
                    {
                    if (!isNull)
                        std::cout << i++ << ": user name is " << name 
<< ", user id is " << id << std::endl;
                    else
                        std::cout << i << " rows selected!" << 
std::endl;
                    }
            >> [](const DrogonDbException &e)
                {
                    std::cerr << "error:" << e.base().what() << 
std::endl;
                };
```

It can be seen that the values of the user_name and user_id fields in the select statement are respectively assigned to the name and id variables in the callback function, and the user does not need to handle these conversions by themselves, which obviously provides a certain convenience, and the user can use it flexibly.

> **Note: It is important to emphasize that in asynchronous programming the user must pay attention to the variable i in the above example. The user must ensure that the variable i is valid when the callback occurs because it is caught by the reference. The callback will be called in another thread, and the current context may have failed when the callback occurred. Programmers typically use smart pointers to hold temporarily created variables and then capture them through callbacks to ensure the validity of the variables.**

Summary

Each DbClient object has one or multiple its own EventLoop threads controlling the database connection IO, accepting the request via an asynchronous or synchronous interface, and returning the result via a callback function.

Blocking interfaces of DbClient only block the caller thread, as long as the caller thread is not the EventLoop thread, it will not affect the normal operation of the EventLoop thread. When the callback function is called, the program inside the callback is run on the EventLoop thread. Therefore, do not perform any blocking operations within the callback, otherwise it will affect the concurrency performance of database read and write. Anyone familiar with non-blocking I/O programming should understand this constraint.

Next: Transaction

Other languages: 繁體中文

Database - Transactions

> **Transactions** are an important feature of relational databases, and Drogon provides transaction support with the `Transaction` class.

Objects of the `Transaction` class are created by `DbClient`, and many transaction-related operations are performed automatically:

- At the beginning of the `Transaction` object creation, the begin statement is automatically executed to `start` the transaction;
- When the `Transaction` object is destructed, the `commit` statement is automatically executed to end the transaction;
- If there is an exception that causes the transaction to fail, the `rollback` statement is automatically executed to roll back the transaction;
- If the transaction has been rolled back, then the sql statement will return an exception (throw an exception or perform an exception callback);

Transaction Creation

The method of transaction creation is provided by `DbClient` as follows:

```
std::shared_ptr<Transaction> newTransaction(const std::function<void(bool)>
&commitCallback = std::function<void(bool)>())
```

This interface is very simple, it returns a smart pointer to a `Transaction` object. Obviously, when the smart pointer loses all the holders and destructs the transaction object, the transaction ends. The parameter `commitCallback` is used to return whether the transaction commit is successful. It should be noted that this callback is only used to indicate whether the `commit` command is successful. If the transaction is automatically or manually rolled back during execution, the `callback` will not be executed. Generally, the `commit` command will succeed， the bool type parameter of this callback is true. Only some special cases, such as the connection disconnection during the commit process, will cause the `commitCallback` to notify the user that the commit fails, at this time, the state of the transaction on the server is not certain, the user needs to deal with this situation specially. Of course, considering that this situation rarely occurs, with non-critical services the user can choose to ignore this event by ignoring the `commitCallback` parameter when creating the transaction (The default empty callback will be passed to the newTransaction method).

The transaction must monopolize the database connection. Therefore, during transaction creation, `DbClient` needs to select an idle connection from its own connection pool and hand it over to transaction object management. This has a problem. If all connections in the `DbClient` are executing sql or other transactions, the interface will block until there is an idle connection.

The framework also provides an asynchronous interface for creating transactions, as follows:

```
void newTransactionAsync(const std::function<void(const
std::shared_ptr<Transaction> &)> &callback);
```

This interface returns the transaction object through the callback function, does not block the current thread, and ensures high concurrency of the application. Users can use it or the synchronous version according to the actual situation.

Transaction Interface

The `Transaction` interface is almost identical to `DbClient`, except for the following two differences:

- `Transaction` provides a `rollback()` interface that allows the user to roll back the transaction under any circumstances. Sometimes, the transaction has been automatically rolled back, and then calling the `rollback()` interface has no negative impact, so explicitly using the rollback() interface is a good strategy to at least ensure that it is not committed incorrectly.
- The user cannot call the transaction's `newTransaction()` interface, which is easy to understand. Although the database has the concept of a sub-transaction, the framework does not currently support it.

In fact, `Transaction` is designed as a subclass of `DbClient`, in order to maintain the consistency of these interfaces, and at the same time, it also creates convenient conditions for the use of ORM.

The framework currently does not provide an interface to control transaction isolation levels, that is, the isolation level is the default level of the current database service.

Transaction Life Cycle

The smart pointer of the transaction object is held by the user. When it has unexecuted sql, the framework will hold it, so don't worry about the transaction object being destructed when there is still unexecuted sql. In addition, the transaction object smart pointer is often caught and used in the result callback of one of its interfaces. This is the normal way to use, don't worry that the circular reference will cause the transaction object to never be destroyed, because the framework will help the user break the circular reference automatically.

One Example

For the simplest example, suppose there is a task table from which the user selects an unprocessed task and changes it to the state being processed. To prevent concurrent race conditions, we use the `Transaction` class, the program is as follows:

```
{
    auto transPtr = clientPtr->newTransaction();
    transPtr->execSqlAsync( "select * from tasks where status=$1 for update
order by time",
                            "none",
                            [=](const Result &r) {
                                if (r.size() > 0)
                                {
                                    std::cout << "Got a task!" <<
std::endl;
                                    *transPtr << "update tasks set
status=$1 where task_id=$2"
                                              << "handling"
                                              << r[0]
["task_id"].as<int64_t>()
                                              >> [](const Result &r)
                                                 {
                                                    std::cout <<
"Updated!";
                                                    ... do something about
the task;
                                                 }
                                              >> [](const DrogonDbException
&e)
                                                 {
                                                    std::cerr << "err:" <<
e.base().what() << std::end;
                                                 };
                                }
                                else
                                {
                                    std::cout << "No new tasks found!" <<
std::endl;
                                }
                            },
                            [](const DrogonDbException &e) {
                                std::cerr << "err:" << e.base().what() <<
std::end;
                            });
}
```

In this case, select for update is used to avoid concurrent modifications. The update statement is completed in the result callback of the select statement. The outermost braces are used to limit the scope of the transPtr so that it can be destroyed in time after the execution of sql to end the transaction.

Next: ORM

Other languages: 繁體中文

Database - ORM

Model

Using Drogon's ORM, you first need to create model classes. Drogon's command-line program drogon_ctl provides the ability to generate model classes. The program reads tables information from a user-specified database and automatically generates multiple source files for the model classes based on this information. When the user uses the model, please include the corresponding header file.

Obviously, each Model class corresponds to a specific database table, and an instance of a model class, corresponds a row of records in the table.

The command to create model classes is as follows:

```
drogon_ctl create model <model_path>
```

The last parameter is the path to store model classes. There must be a configuration file model.json in the path to configure the connection parameters of drogon_ctl to the database. It is a file in JSON format and supports comments. The examples are as follows:

```
{
  "rdbms": "postgresql",
  "host": "127.0.0.1",
  "port": 5432,
  "dbname": "test",
  "user": "test",
  "passwd": "",
  "tables": [],
  "relationships": {
      "enabled": false,
      "items": []
  }
}
```

The configured parameters are the same as the application's configuration file. Please refer to Configuration File.

The `tables` configuration option is unique to the model configuration. It is an array of strings. Each string represents the name of the table to be converted into a model class. If this option is empty, all tables will be used to generate model classes.

The models directory and the corresponding model.json file have been created in advance in the project directory created with the `drogon_ctl create project` command. The user can edit the configuration file and create model classes with the drogon_ctl command.

Model Class Interface

There are mainly two types of interfaces that the user directly uses, getter interfaces and setter interfaces.

There are two types of getter interfaces:

- An interface of the form like `getColumnName` gets the smart pointer of the field. The return value is a pointer instead of a value is primarily used for the NULL field. The user can determine whether the field is a NULL field by determining whether the pointer is empty.
- An interface of the form like `getValueOfColumnName`, hence the name, is the value obtained. For efficiency reasons, the interface returns a constant reference. If the corresponding field is NULL, the interface returns the default value given by the function parameter.

In addition, the binary block type (blob, bytea) has a special interface, in the form of `getValueOfColumnNameAsString`, which loads the binary data into the std::string object and returns it to the user.

The setter interface is used to set the value of the corresponding field, in the form of `setColumnName`, and the parameter type and field type correspond. Automatically generated fields (such as self-incrementing primary keys) do not have a setter interface.

The toJson() interface is used to convert the model object into a JSON object. The binary block type is base64 encoded. Please experiment with it yourself.

The static members of the Model class represent the information of the table. For example, the name of each field can be obtained through the `Cols` static member, which is convenient to use in an editor that supports automatic prompting.

Mapper Class Template

The mapping between the model object and the database table is performed by the Mapper class template. The Mapper class template encapsulates common operations such as adding, deleting, and changing, so that the user can perform the above operations without writing a SQL statement.

The construction of the Mapper object is very simple. The template parameter is the type of the model you want to access. The constructor has only one parameter, which is the DbClient smart pointer mentioned earlier. As mentioned earlier, the Transaction class is a subclass of DbClient, so you can also construct a Mapper object with a smart pointer to a transaction, which means that the Mapper mapping also supports transactions.

Like DbClient, Mapper also provides asynchronous and synchronous interfaces. The synchronous interface is blocked and may throw an exception. The returned future object is blocked in get() and may throw an exception. The normal asynchronous interface does not throw an exception, but returns the result through two callbacks (result callback and exception callback). The type of the exception callback is the same as that in the DbClient interface. The result callback is also divided into several categories according to the interface function. The list is as follows (T is the template parameter, which is the type of the model):

Mapper method 1 interface Mapper method 2 interface Mapper method 3 interface

> **Note: When using a transaction, the exception does not necessarily cause a rollback. Transactions will not be rolled back in the following cases: When the findByPrimaryKey interface does not find a qualified row, when the findOne interface finds fewer or more than one record, the mapper will throw an exception or enter an exception callback, the exception type is UnexpectedRows. If the business logic needs to be rolled back in this condition, please explicitly call the rollback() interface.**

Criteria

In the previous section, many interfaces required input criteria object parameters. The criteria object is an instance of the Criteria class, indicating a certain condition, such as a field greater than, equal to, less than a given value, or a condition such as `is Null`.

```
template <typename T>
Criteria(const std::string &colName, const CompareOperator &opera, T &&arg)
```

The constructor of a criteria object is very simple. Generally, the first argument is the name of the field, the second argument is the enumeration value representing the comparison type, and the third argument is the value being compared. If the comparison type is IsNull or IsNotNull, the third parameter is not required.

E.g:

```
Criteria("user_id",CompareOperator::EQ,1);
```

The above example shows that the field user_id is equal to 1 as a condition. In practice, we prefer to write the following:

```
Criteria(Users::Cols::_user_id,CompareOperator::EQ,1);
```

This is equivalent to the previous one, but this can use the editor's automatic prompts, which is more efficient and less prone to errors;

The Criteria class also supports custom where conditions along with a custom constructor.

```
template <typename... Arguments>
explicit Criteria(const CustomSql &sql, Arguments &&...args)
```

The first argument is a `CustomSql` object of sql statements with `$?` placeholders, while the `CustomSql` class is just a wrapper of a std::string. The second indefinite argument is a parameter pack represents the bound parameter, which behaves just like the ones in execSqlAsync.

E.g:

```
Criteria(CustomSql("tags @> $?"), "cloud");
```

The `CustomSql` class also has a related user-defined string literal, so we recommend to write the following instead:

```
Criteria("tags @> $?"_sql, "cloud");
```

This is equivalent to the previous one.

Criteria objects support AND and OR operations. The sum of two criteria objects constructs a new criteria object, which makes it easy to construct nested conditions. For example:

```
Mapper<Users> mp(dbClientPtr);
auto users = mp.findBy(
(Criteria(Users::Cols::_user_name,CompareOperator::Like,"%Smith")&&Criteria
(Users::Cols::_gender,CompareOperator::EQ,0))
||
(Criteria(Users::Cols::_user_name,CompareOperator::Like,"%Johnson")&&Criter
ia(Users::Cols::_gender,CompareOperator::EQ,1))
));
```

The above program is to query all the men named Smith or the women named Johnson from the users table.

Mapper's Chain Interface

Some common sql constraints, such as limit, offset, etc., Mapper class templates also provide support, provided in the form of a chained interface, meaning that users can string multiple constraints to write. After executing any of the interfaces in Section 10.5.3, these constraints are cleared, that is, they are valid in one operation:

```
Mapper<Users> mp(dbClientPtr);
auto users =
mp.orderBy(Users::Cols::_join_time).limit(25).offset(0).findAll();
```

This program is to select the user list from the `users` table, return the first page of 25 rows per page.

Basically, the name of the chain interface expresses its function, so I won't go into details here. Please refer to the Mapper.h header file.

Convert

The `convert` configuration option is unique to the model configuration. It adds a convert layer before or after a value is read from or written to database. The object consists of a boolean key `enabled` to use this function or not. The array of `items` objects consists of following keys:

- `table`: name of the table, which holds the column
- `column`: name of the column
- `method`: object
  - `after_db_read`: string, name of the method which is called after reading from database, signature: void([const] std::shared_ptr [&])
  - `before_db_write`: string, name of the method which is called before writing to database, signature: void([const] std::shared_ptr [&])
- `includes`: array of strings, name of the include files surrounded by the " or <,>

Relationships

The relationship between the database tables can be configured through the `relationships` option in the model.json configuration file. We use manual configuration instead of automatically detecting the foreign key of the table because the actual project does not use foreign keys. It is also very common.

If the `enable` option is true, the generated model classes will add corresponding interfaces according to the `relationships` configuration.

There are three types of relationships,'has one','has many' and 'many to many'.

- **has one**

  `has one` represents a one-to-one relationship. A record in the original table can be associated with a record in the target table, and vice versa. For example, the `products` table and `skus` table have a one-to-one relationship, we can define as follows:

  ```
  {
    "type": "has one",
    "original_table_name": "products",
    "original_table_alias": "product",
    "original_key": "id",
    "target_table_name": "skus",
    "target_table_alias": "SKU",
    "target_key": "product_id",
    "enable_reverse": true
  }
  ```

  among them:

  - "type": Indicates that this relationship is one-to-one;
  - "original_table_name": the name of the original table (the corresponding method will be added to the model corresponding to this table);
  - "original_table_alias": alias (the name in the method, because the one-to-one relationship is singular, so set it to `product`), if this option is empty, the table name is used to generate the method name;
  - "original_key": the associated key of the original table;
  - "target_table_name": the name of the target table;

  - "target_table_alias": the alias of the target table, if this option is empty, the table name is used to generate the method name;
  - "target_key": the associated key of the target table;
  - "enable_reverse": Indicate whether to automatically generate a reverse relationship, that is, add a method to obtain records of the original table in the model class corresponding to the target table.

  According to this setting, in the model class corresponding to the products table, the following method will be added:

  ```
      /// Relationship interfaces
      void getSKU(const DbClientPtr &clientPtr,
                  const std::function<void(Skus)> &rcb,
                  const ExceptionCallback &ecb) const;
  ```

  This is an asynchronous interface that returns the SKU object associated with the current product in the callback.

  At the same time, since the enable_reverse option is set to true, the following method will be added to the model class corresponding to the skus table:

  ```
      /// Relationship interfaces
      void getProduct(const DbClientPtr &clientPtr,
                      const std::function<void(Products)> &rcb,
                      const ExceptionCallback &ecb) const;
  ```

- **has many**

  `has many` represents a one-to-many relationship. In such a relationship, the table representing `many` generally has a field associated with the primary key of another table. For example, products and reviews usually have a one-to-many relationship, we can define as follows:

  ```
  {
    "type": "has many",
    "original_table_name": "products",
    "original_table_alias": "product",
    "original_key": "id",
    "target_table_name": "reviews",
    "target_table_alias": "",
    "target_key": "product_id",
    "enable_reverse": true
  }
  ```

  The meaning of each configuration above is the same as the previous example, so I won't repeat it here, because there are multiple reviews for a single product, so there is no need to create an alias of reviews. According to this setting, after running `drogon_ctl create model`, the following interface will be added to the model corresponding to the products table:

```
    void getReviews(const DbClientPtr &clientPtr,
                    const std::function<void(std::vector<Reviews>)>
&rcb,
                    const ExceptionCallback &ecb) const;
```

In the model corresponding to the reviews table, the following interface will be added:

```
    void getProduct(const DbClientPtr &clientPtr,
                    const std::function<void(Products)> &rcb,
                    const ExceptionCallback &ecb) const;
```

- **many to many**

As the name implies, `many to many` represents a many-to-many relationship. Usually, a many-to-many relationship requires a pivot table. Each record in the pivot table corresponds to a record in the original table and another record in the target table. For example, the `products` table and `carts` table have a many-to-many relationship, which can be defined as follows:

```
{
  "type": "many to many",
  "original_table_name": "products",
  "original_table_alias": "",
  "original_key": "id",
  "pivot_table": {
    "table_name": "carts_products",
    "original_key": "product_id",
    "target_key": "cart_id"
  },
  "target_table_name": "carts",
  "target_table_alias": "",
  "target_key": "id",
  "enable_reverse": true
}
```

For the pivot table, there is an additional `pivot_table` configuration. The options inside easy to understand and are omitted here.

The model of `products` generated according to this configuration will add the following method:

```
    void getCarts(const DbClientPtr &clientPtr,
                  const
```

```
std::function<void(std::vector<std::pair<Carts,CartsProducts>>)> &rcb,
                                  const ExceptionCallback &ecb) const;
```

The model class of the carts table will add the following method:

```
    void getProducts(const DbClientPtr &clientPtr,
                      const
std::function<void(std::vector<std::pair<Products,CartsProducts>>)>
&rcb,
                      const ExceptionCallback &ecb) const;
```

Restful API controllers

drogon_ctl can also generate restful-style controllers for each model (or table) while creating models, so that users can generate APIs that can add, delete, modify, and search tables with zero coding. These APIs support many functions such as querying by primary key, querying by conditions, sorting by specific fields, returning specified fields, and assigning alias for each field to hide the table structure. It is controlled by the `restful_api_controllers` option in model.json. these options have corresponding comments in the json file.

It should be noted that the controller of each table is designed to be composed of a base class and a subclass. Among them, the base class and the table are closely related, and the subclass is used to implement special business logic or modify the interface format. The advantage of this design is that when the table structure changes, users can update only the base class without overwriting the subclass(by setting the `generate_base_only` option to `true`).

Next: FastDbClient

Other languages: 繁體中文

Database - FastDbClient

As the name implies, FastDbClient will provide higher performance than the normal DbClient. Unlike DbClient has own event loop, it shares the event loop with network IO threads and the main thread of the web application, which makes the internal implementation of FastDbClient available in a lock-free mode and more efficient.

Tests show that FastDbClient has a 10% to 20% performance improvement over DbClient under extremely high load conditions.

Create and Get

FastDbClient must be created automatically by the framework with the configuration file, or by calling the app.createDbClient() interface:

The sub-option `is_fast` of the db_client option in the configuration file indicates if the client is a FastDbClient.Or user can create a FastDbClient by calling the app.createDbClient() method with the last parameter set to true.

The framework creates a separate FastDbClient for each IO's event loop and the main event loop, and each FastDbClient manages several database connections internally. The number of event loop of IO is controlled by the framework's "threads_num" option, which is generally set to the number of CPU cores of the host. The number of the DB connections per event loop is the value of the DB client "connection_number" option. Please refer to Configuration File. Therefore, the total number of DB connections held by FastDbClient is `(threads_num+1) * connection_number`.

The interface to get a FastDbClient is similar to the normal DbClient, as follows:

```
orm::DbClientPtr getFastDbClient(const std::string &name = "default");
/// Use drogon::app().getFastDbClient("clientName") to get a FastDbClient
object.
```

It should be pointed out that due to the special nature of FastDbClient, the user must call the above interface in the IO event loop thread or the main thread to get the correct smart pointer. In other threads, only the null pointer can be obtained and cannot be used.

Usage

The use of FastDbClient is almost identical to that of the normal DbClient, except for the following limitations:

- Both the get and the use of it must be in the framework's IO event loop thread or the main thread. If it is used in other threads, there will be unpredictable errors (because the lock-free condition is destroyed). Fortunately, most of the application programming is in the IO thread, such as within the processing functions of various controllers, within the filter function of filters. It is easy to know that the various callback functions of the FastDbClient interface are also in the current IO thread, and can be safely nestedly used.

- Never use the blocking interface of FastDbClient, because this interface will block the current thread, and the current thread is also the thread that handles the database IO of this object, which will cause permanent blocking, and the user has no chance to get the result.
- Synchronous transaction creation interfaces are likely to block (when all connections are busy), so FastDbClient's synchronous transaction creation interface returns null pointers directly. If you want to use transactions on FastDbClient, please use the asynchronous transaction creation interface.
- After using the FastDbClient to create an Orm Mapper object, you should also use only asynchronous non-blocking interfaces of the mapper object.

Next: Automatic batch mode

Other languages: **繁體中文**

Database - Automatic Batch Mode

The automatic batch mode is only valid for the client library of postgresql 14+ version, and will be ignored in other cases. Before talking about automatic batch processing, let's understand the pipeline mode first.

pipeline mode

Since postgresql 14, its client library provides a pipelining mode interface. In pipelining mode, new sql requests can be sent directly to the server without waiting for the result of the previous request to return (this is consistent with the concept of HTTP pipelining), For details, please refer to Pipeline mode. This mode is very helpful for performance, allowing fewer database connections to support larger concurrent requests. drogon began to support this mode after version 1.7.6, drogon will automatically check whether libpq supports pipeline mode, if so, all requests sent through drogon's DbClient are in pipeline mode.

automatic batch mode

By default, for non-transactional clients, drogon creates synchronization points for each sql request, that is, each individual sql statement is an implicit transaction, which ensures that sql requests are independent of each other, which makes pipeline mode and non-pipeline mode Modes appear to the user to be completely equivalent.

However, creating a synchronization point for each sql statement also has a certain performance overhead, so drogon provides an automatic batch mode in which instead of creating a synchronization point after each sql statement, a synchronization point is created after several sql statements. the rules for creating a synchronization point in the same connection are as follows:

- A synchronization point must be created after the last sql in an EventLoop loop;
- Create a synchronization point after a sql that writes to the database;
- Create a synchronization point after a big sql;
- Create a synchronization point when the number of consecutive SQL statements after the last synchronization point reaches the upper limit;

Note that SQL statements between two synchronization points in the same link belong to the same implicit transaction. Drogon does not provide an explicit interface for opening and closing synchronization points, so these SQLs may be logically unrelated to each other, but due to they are in the same transaction, they will affect each other, therefore, this mode is not completely safe mode, it has the following problems:

- A failed sql will cause its previous sql statement to be rolled back after the last synchronization point, but the user will not receive any notification, because no explicit transaction is used;
- A failed sql will cause all subsequent sql statements before the next synchronization point to return failure;
- The judgment of writing database is based on simple keyword matching (insert, update, etc.), which does not cover all cases, such as the case of calling stored procedures through select statements, so although drogon strives to reduce the negative effects of automatic batch mode, it is not completely safe;

Therefore, automatic batch mode is helpful to improve performance, but it is not safe. It is up to the user to decide under what circumstances to use it. For example, to execute pure read-only sql statements via the DbClient in automatic batch mode.

> **Note** Even read-only sql can sometimes cause transaction failure (such as select timeout), so its subsequent sql will also be affected by it and fail (this may not be acceptable to users, because they may not be related to each other in application logic), so, strictly speaking, the application scenarios of automatic batch mode should be limited to read-only and non-critical data queries. It is recommended that users create a separate automatic batch DbClient for such sql statements. Of course, it is easy to realize that transaction objects generated by DbClient in automatic batch mode are safe to use.

Enable automatic batch mode

When using the newPgClient interface to create a client, set the third parameter to true to enable automatic batch mode; When using a configuration file to create a client, set the auto_batch option to true to enable automatic batch mode for the client;

Next: Request References

Other languages: 繁體中文

References Request

The HttpRequest type pointer commonly named `req` in the examples in this documentation represents the data contained in a request received or sent by drogon, below are the some methods by which you can interact with this object:

- ### `isOnSecureConnection()`

  #### Summary

  Function that returns if the request was made on https.

  #### Inputs

  None.

  #### Returns

  bool type.

- ### `getMethod()`

  #### Summary

  Function that returns the request method. Useful to differentiate the request method if a single handle allows more than one type.

  #### Inputs

  None.

  #### Returns

  `HttpMethod` request method object.

  #### Examples

  ```
  #include "mycontroller.h"

  using namespace drogon;

  void mycontroller::anyhandle(const HttpRequestPtr &req,
  std::function<void (const HttpResponsePtr &)> &&callback) {
      if (req->getMethod() == HttpMethod::Get) {
        // do something
  ```

```
    } else if (req->getMethod() == HttpMethod::Post) {
      // do other something
    }
}
```

- `getParameter(const std::string &key)`

  Summary

  Function that returns the value of a parameter based on an identifier. The behavior changes based on the Get or Post request type.

  Inputs

  string type identifier of param.

  Returns

  param content on `string` format.

  Examples

  On Get type:

```
#include "mycontroller.h"
#include <string>

using namespace drogon;

void mycontroller::anyhandle(const HttpRequestPtr &req,
std::function<void (const HttpResponsePtr &)> &&callback) {
    // https://mysite.com/an-path/?id=5
    std::string id = req->getParameter("id");
    // or
    long id = std::strtol(req->getParameter("id"));
}
```

  Or On Post type:

```
#include "mycontroller.h"
#include <string>

using namespace drogon;

void mycontroller::loginHandle(const HttpRequestPtr &req,
std::function<void (const HttpResponsePtr &)> &&callback) {
    // request contain a Form Login
    std::string email = req->getParameter("email");
    std::string password = req->getParameter("password");
}
```

- `getPath()`

  Similar

  path()

  Summary

  Function that returns the request path. Useful if you use an ADD_METHOD_VIA_REGEX or other type of dynamic URL in the controller.

  Inputs

  None.

  Returns

  string representing the request path.

  Examples

  ```
  #include "mycontroller.h"
  #include <string>

  using namespace drogon;

  void mycontroller::anyhandle(const HttpRequestPtr &req,
  std::function<void (const HttpResponsePtr &)> &&callback) {
      // https://mysite.com/an-path/?id=5
      std::string url = req->getPath();

      // url = /an-path/
  }
  ```

- `getBody()`

  Similar

  body()

  Summary

  Function that returns the request body content (if any).

Inputs

None.

Returns

String representing the request body (if any).

- getHeader(std::string key)

  Summary

  Function that returns a request header based on an identifier.

  Inputs

  String header identifier.

  Returns

  The content of the header in string format.

  Examples

  ```
  #include "mycontroller.h"
  #include <string>

  using namespace drogon;

  void mycontroller::anyhandle(const HttpRequestPtr &req,
  std::function<void (const HttpResponsePtr &)> &&callback) {
      if (req->getHeader("Host") != "mysite.com") {
        // return http 403
      }
  }
  ```

- `headers()`

  Summary

  Function that returns all headers of a request.

  Inputs

  None.

  Returns

  An unordered_map containing the headers.

Examples

```
#include "mycontroller.h"
#include <unordered_map>
#include <string>

using namespace drogon;

void mycontroller::anyhandle(const HttpRequestPtr &req,
std::function<void (const HttpResponsePtr &)> &&callback) {
    for (const std::pair<const std::string, const std::string> &header
: req->headers()) {
      auto header_key = header.first;
      auto header_value = header.second;
    }
}
```

- `getCookie()`

  Summary

  Function that returns request cookie based on an identifier.

  Inputs

  None.

  Returns

  Value of cookie on string format.

- `cookies()`

  Summary

  Function that returns all cookies of a request.

  Inputs

  None.

  Returns

  An unordered_map containing the cookies.

  Examples

```
#include "mycontroller.h"
#include <unordered_map>
#include <string>

using namespace drogon;

void mycontroller::anyhandle(const HttpRequestPtr &req,
std::function<void (const HttpResponsePtr &)> &&callback) {
    for (const std::pair<const std::string, const std::string> &header
: req->cookies()) {
        auto cookie_key = header.first;
        auto cookie_value = header.second;
    }
}
```

- `getJsonObject()`

  Summary

  Function that converts the body value of a request into a Json object (normally POST requests).

  Inputs

  None.

  Returns

  A Json object.

  Examples

  ```
  #include "mycontroller.h"

  using namespace drogon;

  void mycontroller::anyhandle(const HttpRequestPtr &req,
  std::function<void (const HttpResponsePtr &)> &&callback) {
      // body = {"email": "test@gmail.com"}
      auto jsonData = *req->getJsonObject();

      std::string email = jsonData["email"].asString();
  }
  ```

Useful Things

From here are not methods of the Http Request object, but some useful things you can do to process the requests you will receive

Parsing File Request

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req, std::function<void
(const HttpResponsePtr &)> &&callback) {
    // Only Post Requests (File Form)

    MultiPartParser file;
    file.parse(req);

    if (file.getFiles().empty()) {
      // Not Files Found
    }

    // Get First file and save then
    const HttpFile archive = file.getFiles()[0];
    archive.saveAs("/tmp/" + archive.getFileName());
  }
```

For more information about parsing file: File Handler

Next: File Handler

Other languages: 繁體中文

File Handler

File parsing is extracting the file (or files) from a multipart-data POST request to an `HttpFile` object through `MultiPartParser`, here is some information about:

`MultiPartParser` Object

Summary

```
It is the object that you will use to extract and temporarily store the
request files.
```

- `parse(const std::shared_ptr<HttpRequest> &req)`

  Summary

  Receives the request object as a parameter, reads and identifies the files (if heard) and transfers it to the MultiPartParser variable.

  Example

  ```
  #include "mycontroller.h"

  using namespace drogon;

  void mycontroller::postfile(const HttpRequestPtr &req, std::function<void (const HttpResponsePtr &)> &&callback) {
      // Only Post Requests (File Form)

      MultiPartParser fileParser;
      fileParser.parse(req);
  }
  ```

- `getFiles()`

  Summary

  Must be called after `parse()`, returns files of request in the format std::vector&ltHttpFile&gt.

  Example

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req, std::function<void (const HttpResponsePtr &)> &&callback) {
    // Only Post Requests (File Form)

    MultiPartParser fileParser;
    fileParser.parse(req);

    // Check if have files
    if (fileParser.getFiles().empty()) {
        // No files found
    }

    size_t num_of_files = fileParser.getFiles().size();
}
```

- `getParameters()`

  Summary

  Must be called after `parse()`, returns the list of other parts from the MultiPartData form.

  Returns

  std::unordered_map&ltstd::basic_string&ltchar&gt&gt (key, value)

  Example

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req, std::function<void (const HttpResponsePtr &)> &&callback) {
    // Only Post Requests (File Form)

    MultiPartParser fileParser;
    fileParser.parse(req);

    if (!fileParser.getFiles().empty()) {
        for (const auto &header : fileParser.getParameters()){
            header.first // Key form
            header.second // Value from key form
        }
    }
}
```

- `getParameter<Typename T>(const std::string &key)`

  Summary

  Must be called after `parse()`, individual version of getParameters().

  Inputs

  The type of the expected object (will be converted automatically), the key value of the parameter.

  Returns

  The content of the parameter corresponding to the key in the informed format, if it does not exist, will return the default value of the T Object.

  Example

```
  #include "mycontroller.h"

  using namespace drogon;

  void mycontroller::postfile(const HttpRequestPtr &req, std::function<void (const HttpResponsePtr &)> &&callback) {
      // Only Post Requests (File Form)

      MultiPartParser fileParser;
      fileParser.parse(req);

      std::string email = fileParser.getParameter<std::string>("email_form");

      // Default type of string is ""
      if (email.empty()) {
          // email_form not found
      }
  }
```

`HttpFile` Object

Summary

It is the object that represents a file in memory, used by `MultiPartParser`.

- `getFileName()`

  Summary

  Self-explanatory name, gets the original name of the file that was received.

Returns

std::string.

Example

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req, std::function<void (const HttpResponsePtr &)> &&callback) {
    // Only Post Requests (File Form)

    MultiPartParser fileParser;
    fileParser.parse(req);

    std::string filename = fileParser.getFiles()[0].getFileName();
}
```

- `fileLength()`

Summary

Gets the file size.

Returns

`size_t`

Example

```
#include "mycontroller.h"

using namespace drogon;

void mycontroller::postfile(const HttpRequestPtr &req, std::function<void (const HttpResponsePtr &)> &&callback) {
    // Only Post Requests (File Form)

    MultiPartParser fileParser;
    fileParser.parse(req);

    size_t filesize = fileParser.getFiles()[0].fileLength();
}
```

- `getFileExtension()`

Summary

Gets the file extension.

Returns

std::string

Example

```
  #include "mycontroller.h"

  using namespace drogon;

  void mycontroller::postfile(const HttpRequestPtr &req,
std::function<void (const HttpResponsePtr &)> &&callback) {
      // Only Post Requests (File Form)

      MultiPartParser fileParser;
      fileParser.parse(req);

      std::string file_extension = fileParser.getFiles()
[0].getFileExtension();
  }
```

- `getMd5()`

  Summary

  Get MD5 hash of file to check integrity.

  Returns

  std::string

- `save()`

  Summary

  Save the file to the file system. The folder saving the file is `UploadPath` configured in config.json (or equivalent). The full path is

  ```
  drogon::app().getUploadPath()+"/"+this->getFileName()
  ```

  Or to simplify, it is saved as: UploadPath/filename

- `save(const std::string &path)`

Summary

Version if parameter is not omitted, uses the &path parameter instead of UploadPath.

Example

```
  #include "mycontroller.h"

  using namespace drogon;

  void mycontroller::postfile(const HttpRequestPtr &req, std::function<void (const HttpResponsePtr &)> &&callback) {
      // Only Post Requests (File Form)

      MultiPartParser fileParser;
      fileParser.parse(req);

      // Relative path
      fileParser.getFiles()[0].save("./"); // Writes the file to the same directory on the server, with the original name

      // Absolute path
      fileParser.getFiles()[0].save("/home/user/downloads/"); // Writes the file in the indicated directory, with the original name
  }
```

- `saveAs(const std::string &path)`

Summary

Writes the file to the path parameter with a new name (ignores the original name).

Example

```
  #include "mycontroller.h"

  using namespace drogon;

  void mycontroller::postfile(const HttpRequestPtr &req, std::function<void (const HttpResponsePtr &)> &&callback) {
      // Only Post Requests (File Form)

      MultiPartParser fileParser;
      fileParser.parse(req);

      // Relative path
      fileParser.getFiles()[0].saveAs("./image.png"); // Same path of server
      /* Just example, Don't do this, you would overwrite the file
```

```
format without checking if it really is png */

      // Absolute path
      fileParser.getFiles()[0].save("/home/user/downloads/anyname." +
fileParser.getFiles()[0].getFileExtension());
  }
```

Next: Plugins

Other languages: 繁體中文

Plugins

Plugins are use to help users build complex applications. In Drogon, all plugins are built and installed into the application based on the configuration file. Plugins in Drogon are single-instance, and users can implement any functionality they want with plugins.

When Drogon runs the run() interface, it instantiates each plugin one by one according to the configuration file and calls the `initAndStart()` interface of them.

Configuration

Plugin configuration is done through the configuration file, for example:

```
    "plugins": [
        {
        //name: The class name of the plugin
        "name": "DataDictionary",
        //dependencies: Plugins that the plugin depends on. It can be
commented out
        "dependencies": [],
        //config: The configuration of the plugin. This json object is the
parameter to initialize the plugin.
        //It can be commented out
        "config": {
        }
        }],
```

It can be seen that there are three configurations for each plugin:

- name: is the class name of the plugin (including the namespace). The framework will create a plugin instance based on the class name. If the item is commented out, the plugin becomes disabled.
- dependencies: Is a list of names of other plugins that the plugin depends on. The framework creates and initializes all plugins in a specific order. Prioritize the creation and initialization of plugins that are dependent by others. At the end of the program, plugins are closed and destroyed in reverse order. Please note that circular dependencies in plugins are forbidden. Drogon will report an error and exit the program if it detects a circular dependency. If the item is commented out, the list of dependencies is empty.
- config: is the json object used to initialize the plugin, the object is passed as an input parameter to the plugin's `initAndStart()` interface. If the item is commented out, the json object passed to the `initAndStart` interface is an empty object;

Definition

User-defined plugins must inherit from the drogon::Plugin class template, and the template parameter is the plugin type, such as the following definition:

```
class DataDictionary : public drogon::Plugin<DataDictionary>
{
public:
    virtual void initAndStart(const Json::Value &config) override;
    virtual void shutdown() override;
    ...
};
```

One can create source files of plugin by drogon_ctl command:

```
drogon_ctl create plugin <[namespace::]class_name>
```

Getting Instance

The plugin instance is created by drogon, and the user can get the plugin instance through the following interface of drogon:

```
template <typename T> T *getPlugin();
```

Or

```
PluginBase *getPlugin(const std::string &name);
```

Obviously, the first method is more convenient. For example, the DataDictionary plugin mentioned above can be obtained like this:

```
auto *pluginPtr=app().getPlugin<DataDictionary>();
```

Note that it is best to get the plugin after calling the framework's run() interface, otherwise one will get an uninitialized plugin instance (this doesn't necessarily lead to an error, it is ok to just make sure to use the plugin after initialization). Of course, since the plugin is initialized in a dependency order, it is no problem to get the instance of another plugin in the `initAndStart()` interface.

Life Cycle

All plugins are initialized in the run() interface of the framework and are destroyed when the application exits. Therefore, the plugin's lifecycle is almost identical to the application, which is why the getPlugin() interface does not need to return a smart pointer.

Next: Configuration File

Other languages: 繁體中文

Configuration File

You can control various behaviors of the Http server by configuring various parameters through multiple interfaces of the DrogonAppFramework instance. However, using a configuration file is a better way for the following reasons:

- Using a configuration file instead of source code can determine the behavior of the application at runtime rather than at compile time, which is undoubtedly a more convenient and flexible way;
- Using a configuration file can make the main file more concise;

Based on these additional benefits, it is recommended that application developers use configuration files to configure various parameters of the application.

The configuration file can be loaded very simply by calling the loadConfigFile() interface before calling the run() interface. The parameter of the loadConfigFile() method is the file name of the configuration file, for example:

```
int main()
{
    drogon::app().loadConfigFile("config.json");
    drogon::app().run();
}
```

The above program loads the configuration file `config.json` and then runs the application. The specific listening port, log output, database configuration and so on can be configured by the configuration file. In fact, this program can basically be the entire code of the main file of the web application.

Configuration File Details

An example of a configuration file is at the top level of the source directory, config.example.json. If you use the drogon_ctl create project command to create a project, you can also find the file config.json with the same content in the project directory. So, you basically don't need to create a new configuration file but make some changes to this file to complete the configuration of the web application.

The file is in `JSON` format and supports comments. You can comment out the unneeded configuration items with the c++ comment symbols /**/ and //.

After commenting out a configuration option, the framework initializes it with default values. The default value for each option can be found in the comments in the configuration file.

Supportted Format

- json

- yaml, should install yaml-cpp library to provide the yaml file parser.

- SSL

  The ssl option is to configure SSL files of the https service as follows:

  ```
    "ssl": {
      "cert": "../../trantor/trantor/tests/server.pem",
      "key": "../../trantor/trantor/tests/server.pem",
      "conf":[
        ["Options", "Compression"],
        ["min_protocol", "TLSv1.2"]
      ]
    }
  ```

  Where `cert` is the path of the certificate file and `key` is the path of the private key file. If a file contains both a certificate and a private key, the two paths can be made the same. The file is in PEM encoding format.

  `conf` are optional SSL options that are directly passed into SSL_CONF_cmd to allow low level configuration of the encryption. The options must be one or two element arrays.

- listeners

  As the name implies, the `listeners` option is to configure listeners for the web application. It is a JSON array type. Each JSON object represents a listener. The specific configuration is as follows:

  ```
  "listeners": [
      {
        "address": "0.0.0.0",
        "port": 80,
        "https": false
      },
      {
        "address": "0.0.0.0",
        "port": 443,
        "https": true,
        "cert": "",
        "key": ""
      }
    ]
  ```

  Among them:

  - `address`: With the string type indicates the IP address to be listened to. If this option is not available, the default value "0.0.0.0" is used.
  - `port`: `port`: An integer type option indicating the port to be listened to. It must be a valid port number. There is no default value.
  - `https`: Boolean type, indicating whether to use https, the default value is false, which means using http.

  - `cert` and `key`: The string type, is valid when `https` is true, indicating the certificate and private key of https. The default value is an empty string, indicating the certificate and private key file configured by the global option `ssl`;

- ### db_clients

  This option is used to configure the database client. It is a JSON array type. Each JSON object represents a separate database client. The specific configuration is as follows:

  ```
  "db_clients":[
    {
      "name":"",
      "rdbms": "postgresql",
      "host": "127.0.0.1",
      "port": 5432,
      "dbname": "test",
      "user": "",
      "passwd": "",
      "is_fast": false,
      "connection_number": 1,
      "filename": ""
    }
  ]
  ```

  Among them:

  - `name`: string type, client name, the default value is "default", name is the application developer to get the database client's markup from the framework, if there are multiple clients, the name field must be different;
  - `rdbms`：A string indicating the type of database server. Currently supports "postgresql","mysql" and "sqlite3", which is not case sensitive;
  - `host`：String, database server address, `localhost` is the default value;
  - `port`：An integer representing the port number of the database server;
  - `dbname`：String, database name;
  - `user`：String, user name;
  - `passwd`：String, password;
  - `is_fast`：bool， false by default, indicate if the client is a FastDbClient
  - `connection_number`：A integer indicating the number of connections to the database server, at least 1, the default value is also 1, affecting the concurrent performance of data read and write; If the 'is_fast' is true, the number is the number of connections per event loop, otherwise it is the total number of all connections.
  - `filename`: The filename of sqlite3 database;

- ### threads_num

  Suboption belonging to the `app` option, an integer, the default value is 1, indicating the number of IO threads, which has a clear impact on network concurrency. This number is not as big as possible. Users who understand the non-blocking I/O principle should know that this value should be the same as the number of processors that he expects network IO to occupy. If the value is set to 0, the number of IO threads will be the number of all hardware cores.

```
"threads_num": 16,
```

The above example shows that network IO uses 16 threads and can run up to 16 CPU cores under high load conditions.

- Session

  Session-related options are also children of the `app` option, controlling whether session is used and the session timeout. Such as:

  ```
  "enable_session": true,
  "session_timeout": 1200,
  ```

  Among them:

  - `enable_session`: A Boolean value indicating whether to use a session. The default is false. If the client does not support cookies, set it to false because the framework will create a new session for each request without a session cookie, which will result in completely unnecessary resource and performance loss;
  - `session_timeout`: An integer value indicating the timeout period of the session, in seconds. The default value is 0, indicating permanent validity. Only works if enable_session is true.

- document_root

  The suboption of the `app` option, a string, indicates the document path corresponding to the Http root directory, and is the root path of the static file download. The default value is "./", which indicates the current path of the program running. such as:

  ```
  "document_root": "./",
  ```

- upload_path

  The child of the `app` option, a string, represents the default path for uploading files. The default value is "uploads". If the value is not starting with `/`, `./` or `../`, and this value is not .`or`..`, then this path is the relative path of the previous document_root entry, otherwise it is an absolute path or a relative path to the current directory. Such as:

  ```
  "upload_path":"uploads",
  ```

- client_max_body_size

  A child of the `app` object, a string, represents the overall maximum size of the body of a request. You can use the suffixes `k`, `m`, `g`, to specify kilobyte, megabyte or gigabye (byte * 1024), on 64bit build you can also use `t` for terabytes. The suffixes can be both lower case or upper case.

  ```
  "client_max_body_size": "10M",
  ```

- client_max_memory_body_size

  A child of the `app` object, a string, represents the maximum buffer size to store a request body before caching to a file. You can use the suffixes `k`, `m`, `g`, to specify kilobyte, megabyte or gigabye (byte * 1024), on 64bit build you can also use `t` for terabytes. The suffixes can be both lower case or upper case.

  ```
  "client_max_memory_body_size": "50K"
  ```

- file_types

  The suboption of the app option, an array of strings, with default values as follows, indicates the static file download type supported by the framework. If the requested static file extension is outside of these types, the framework will return a 404 error.

  ```
  "file_types": [
        "gif",
        "png",
        "jpg",
        "js",
        "css",
        "html",
        "ico",
        "swf",
        "xap",
        "apk",
        "cur",
        "xml"
      ],
  ```

- mime

  The suboption of the app option, a dictionary of strings to strings or an array of strings. Declares how file extensions are mapped to new MIME types (i.e. for those not recognized by default) when sending static files. Note that this options merely registers the MIME. The framework still sends a 404 if the extension is not in `file_types` described above.

```
"mime" : {
  "text/markdown": "md",
  "text/gemini": ["gmi", "gemini"]
}
```

- Connection number control

  The children of the `app` option have two options, as follows:

  ```
  "max_connections": 100000,
  "max_connections_per_ip": 0,
  ```

  Among them:

  - `max_connections`: Integer, the default value is 100000, which means the maximum number of simultaneous concurrent connections; when the number of connections maintained by the server reaches this number, the new TCP connection request will be rejected directly.
  - `max_connections_per_ip`: Integer, the default value is 0, which means the maximum number of connections for a single client IP, and 0 means no limit.

- Log option

  The child of the `app` item, a JSON object, controls the behavior of the log output as follows:

  ```
  "log": {
        "log_path": "./",
        "logfile_base_name": "",
        "log_size_limit": 100000000,
        "log_level": "TRACE"
      },
  ```

  Among them:

  - `log_path`: String, the default value is an empty string, indicating the path where the log file is stored. If it is an empty string, all logs are output to the standard output.
  - `logfile_base_name`: A string indicating the `basename` of the log file. The default value is an empty string which means the basename will be `drogon`.
  - `log_size_limit`: A integer, in bytes. The default value is 100000000 (100M). When the size of the log file reaches this value, the log file will be switched.
  - `log_level`: A string, the default value is "DEBUG", which indicates the lowest level of log output. The optional values are from low to high: "TRACE", "DEBUG", "INFO", "WARN", where the TRACE level is only valid when compiling in DEBUG mode.

> **Note: Drogon's file log uses a non-blocking output structure that can achieve a log output of millions of lines per second and can be used with confidence.**

- Application control

  They are also children of the app option and have two options, as follows:

  ```
       "run_as_daemon": false,
       "relaunch_on_error": false,
  ```

  Among them：

  - run_as_daemon：Boolean value, the default value is false. When it is true, the application will be a child process of the No.1 process in the form of a daemon running in the background of the system.
  - relaunch_on_error：Boolean value, the default value is false. When it is true, the application will relaunches itself on error.

- use_sendfile

  The suboption of app option, boolean, indicates whether the linux system call sendfile is used when sending the file. The default value is true. Using sendfile can improve the sending efficiency and reduce the memory usage of large files. as follows:

  ```
  "use_sendfile": true,
  ```

  > **Note: Even if this option is true, the sendfile system call will not be used, because the use of sendfile for small files is not necessarily cost-effective, and the framework will decide whether to adopt it according to its own optimization strategy.**

- use_gzip

  The app suboption, boolean, default value is true, indicating whether the body of the Http response uses compressed transmission. When it is true, compression is used in the following cases:

  - The client supports gzip compression;
  - The Http body is the text type;
  - The length of the body is greater than a certain value;

  The configuration example is as follows:

  ```
  "use_gzip": true,
  ```

- static_files_cache_time

  The app suboption, integer value, in seconds, indicates the cache time of the static file, that is, for the repeated request for the file during this time, the framework will return the response directly from the memory without reading the file system. The default value is 5 seconds, 0 means always cache (only read the file system once, use with caution), negative value means no cache. as follows:

```
"static_files_cache_time": 5,
```

- simple_controllers_map

  The `app` suboption, an array of JSON objects, each representing a mapping from the Http path to the HttpSimpleController, this configuration is just an alternative, not necessarily configured here, see HttpSimpleController. The specific configuration is as follows:

  ```
  "simple_controllers_map": [
        {
          "path": "/path/name",
          "controller": "controllerClassName",
          "http_methods": ["get","post"],
          "filters": ["FilterClassName"]
        }
      ],
  ```

  Among them：

  - `path`：String, Http path;
  - `controller`：String, the name of the HttpSimpleController;
  - `http_methods`：An array of strings representing the supported Http methods. Requests outside this list will be filtered out, returning a 405 error.
  - `filters`：String array, list of filters on the path, see Middleware and Filter;

- Idle connection timeout control

  The `app` suboption, integer value, in seconds, the default value is 60. When a connection exceeds this time without any reading and writing, the connection will be forcibly disconnected. as follows:

  ```
  "idle_connection_timeout":60
  ```

- Dynamic view loading

  The sub-options of the app, which control the enabling and the path of the dynamic view, have two options, as follows:

  ```
  "load_dynamic_views":true,
  "dynamic_views_path":["./views"],
  ```

Among them：

- `dynamic_views_path`：Boolean value, the default value is false. When it is true, the framework searches view files in the view path and dynamically compiles them into .so files, then loads them into the application. When any view file changes, it will also cause automatic compilation and re-loading;
- `dynamic_views_path`：An array of strings, each of which represents the search path of the dynamic view. If the path value is not starting with `/`, `./` or `../`, and the value is not `.` or `..`, then This path is the relative path of the previous document_root entry, otherwise it is an absolute path or a relative path to the current directory.

See View

- Server header field

  The sub-option of the app configures the server header field of all responses sent by the framework. The default value is an empty string. When this option is empty, the framework automatically generates a header field of the form `Server: drogon/version string`. It's as follows:

  ```
  "server_header_field": ""
  ```

- Keepalive requests

  The `keepalive_requests` option sets the maximum number of requests that can be served through one keep-alive connection. After the maximum number of requests are made, the connection is closed. The default value of 0 means no limit. It's as follows:

  ```
  "keepalive_requests": 0
  ```

- Pipelining requests

  The `pipelining_requests` sets the maximum number of unhandled requests that can be cached in pipelining buffer. After the maximum number of requests are made, the connection is closed. The default value of 0 means no limit. For details about pipelining, please see the `rfc2616-8.1.1.2`. It's as follows:

  ```
  "pipelining_requests": 0
  ```

Next: Aspect Oriented Programming (AOP)

Other languages: 繁體中文

drogon_ctl - Command

After the **Drogon** framework is compiled and installed, it is recommended to create your first project using the command line program `drogon_ctl` which is installed alongside the framework, for convenience there is the shortened command `dg_ctl`. Users can choose according to their preferences.

The main function of the program is to make it easy for users to create various drogon project files. Use the `dg_ctl help` command to see the functions it supports, as follows:

```
$ dg_ctl help
usage: drogon_ctl <command> [<args>]
commands list:
create                  create some source files(Use 'drogon_ctl help
create' for more information)
help                    display this message
version                 display version of this tool
press                   Do stress testing(Use 'drogon_ctl help press' for
more information)
```

Version subcommand

The `version` subcommand is used to print the drogon version currently installed on the system, as follows:

```
$ dg_ctl version
     _
  __| |_ __ ___   __ _  ___  _ __
 / _` | '__/ _ \ / _` |/ _ \| '_ \
| (_| | | | (_) | (_| | (_) | | | |
 \__,_|_|  \___/ \__, |\___/|_| |_|
                 |___/

drogon ctl tools
version:0.9.30.771
git commit:d4710d3da7ca9e73b881cbae3149c3a570da8de4
compile config:-O3 -DNDEBUG -Wall -std=c++17 -I/root/drogon/trantor -
I/root/drogon/lib/inc -I/root/drogon/orm_lib/inc -I/usr/local/include -
I/usr/include/uuid -I/usr/include -I/usr/include/mysql
```

Create sub command

The `create` subcommand is used to create various objects. It is currently the main function of drogon_ctl. Use the `dg_ctl help create` command to print detailed help for this command, as follows:

```
$ dg_ctl help create
Use create command to create some source files of drogon webapp

Usage:drogon_ctl create <view|controller|filter|project|model> [-options]
<object name>

drogon_ctl create view <csp file name> [-o <output path>] [-n <namespace>]|
[--path-to-namespace] //create HttpView source files from csp file

drogon_ctl create controller [-s] <[namespace::]class_name> //create
HttpSimpleController source files

drogon_ctl create controller -h <[namespace::]class_name> //create
HttpController source files

drogon_ctl create controller -w <[namespace::]class_name> //create
WebSocketController source files

drogon_ctl create filter <[namespace::]class_name> //create a filter named
class_name

drogon_ctl create project <project_name> //create a project named
project_name

drogon_ctl create model <model_path> //create model classes in model_path
```

- **View creation**

  The `dg_ctl create view` command is used to generate source files from csp files, see the View section. In general, this command does not need to be used directly. It is better practice to configure the cmake file to executed this command automatically. The command example is as follows, assuming the csp file is `UsersList.csp`.

  ```
  dg_ctl create view UsersList.csp
  ```

- **Controller creation**

  The `dg_ctl create controller` command is used to help the user create the controller's source files. The three controllers currently supported by drogon can be created by this command.

  - The command to create an HttpSimpleController is as follows:

  ```
  dg_ctl create controller SimpleControllerTest
  dg_ctl create controller webapp::v1::SimpleControllerTest
  ```

  The last parameter is the controller's class name, which can be prefixed by a namespace.

- The command to create an HttpController is as follows:

```
dg_ctl create controller -h ControllerTest
dg_ctl create controller -h api::v1::ControllerTest
```

- The command to create a WebSocketController is as follows:

```
dg_ctl create controller -w WsControllerTest
dg_ctl create controller -w api::v1::WsControllerTest
```

- **Filter creation**

The `dg_ctl create filter` command is used to help the user create the source files for filters, see the Middleware and Filter section.

```
dg_ctl create filter LoginFilter
dg_ctl create filter webapp::v1::LoginFilter
```

- **Create project**

The best way the user creates a new Drogon application project is via the drogon_ctl command, as follows:

```
dg_ctl create project ProjectName
```

After the command is executed, a complete project directory will be created in the current directory. The directory name is `ProjectName`, and the user can directly compile the project in the build directory (cmake .. && make). Of course, it does not have any business logic.

The directory structure of the project is as follows:

```
├── build                          Build folder
├── CMakeLists.txt                 Project cmake configuration file
├── cmake_modules                  Cmake scripts for third-party
libraries lookup
│   ├── FindJsoncpp.cmake
│   ├── FindMySQL.cmake
│   ├── FindSQLite3.cmake
│   └── FindUUID.cmake
├── config.json                    The configuration file of the drogon
application, please refer to the introduction section of the
configuration file.
├── controllers                    The directory where the controller
```

```
source files are stored
├── filters                     The directory where the filter files
are stored
├── main.cc                     Main program
├── models                      The directory of the database model
file, model source file creation see 11.2.5
│   └── model.json
├── tests                       The direftory for unit/integration
tests
│   └── test_main.cc            Entry point for tests
└── views                       The directory where view csp files
are stored, the source file does not need to be manually created by
the user, and csp files are automatically preprocessed to obtain view
source files when the project is compiled.
```

- **Create models**

  Use the `dg_ctl create model` command to create database model source files. The last parameter is the directory where models is stored. This directory must contain a model configuration file named `model.json` to tell dg_ctl how to connect to the database and which tables to be mapped.

  For example, if you want to create models in the project directory mentioned above, execute the following command in the project directory:

  ```
  dg_ctl create model models
  ```

  This command will prompt the user that the file will be overwritten directly. After the user enters `y`, it will generate all the model files.

  Other source files need to reference model classes should include model header files, such as:

  ```
  #include "models/User.h"
  ```

  Note that the models directory name is included to distinguish between multiple data sources in the same project. See ORM.

Stress Testing

One can use the `dg_ctl press` command to do a stress testing, there are several options for this command.

- `-n num` Set the number of requests(default : 1)
- `-t num` Set the number of threads(default : 1), Set the number to the number of CPUs to achieve maximum performance
- `-c num` Set the number of concurrent connections(default : 1)
- `-q` No progress indication(default: no)

For example, users can test an HTTP server as follows:

```
dg_ctl press -n1000000 -t4 -c1000 -q http://localhost:8080/
dg_ctl press -n 1000000 -t 4 -c 1000 https://www.domain.com/path/to/be/tested
```

Next: AOP Aspect-Oriented-Programming

Other languages: 繁體中文

Aspect Oriented Programming

AOP(Aspect Oriented Programming) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns(Quoted from Wikipedia).

Limited to the features of the C++ language, Drogon does not provide a flexible AOP solution like Spring, but a simple AOP in which all join points are predefined in the framework, and by using the framework's AOP series interfaces, one can register handlers(called 'advices' in Drogon) onto specific join points.

Predefined joinpoints

Drogon provides seven joinpoints for users. When the application runs to the joinpoints, the user-registered handlers(Advices) are called one by one. The description of these joinpoints is as follows:

- Beginning: As the name implies, the joinpoint is at the beginning of the program. Specifically, all handlers registered at this joinpoint are executed immediately after the app().run() method has finished initialization. At this joinpoint, all the controllers, filters, plugins, and database clients have been built completely, users can get the desired object reference or perform some other initialization work here. The advice on the joinpoint is only run once, the call signature of the advice is `void()`, the registration interface is `registerBeginningAdvice`;

- NewConnection: Advices registered to this joinpoint are called when each new TCP connection is established. The call signature of the advice is `bool(const trantor::InetAddress &, const trantor::InetAddress &)`, where the first argument is the remote address of the TCP connection and the second one is the local address. Note that the return type is bool, if the user returns false, the corresponding connection will be disconnected. The registration interface is `registerNewConnectionAdvice`;

- HttpResponseCreation: Advices registered to this joinpoint are called when each HTTP Response object is created. The call signature of the Advice is `void(const HttpResponsePtr &)`, where the parameter is the newly created object, and the user can perform some unified operations on all Responses with this joinpoint, such as adding a special header, etc. This joinpoint affects all Responses, including 404 or any drogon internal error response, and also including all responses generated by user's application. The registration interface is `registerHttpResponseCreationAdvice`;

- Sync: This joinpoint is located at the front end of Http request processing. Users can intercept this request by returning a non-empty Response object. The call signature of Advices is `HttpRequestPtr(const HttpRequestPtr &)`. The registration interface is `registerSyncAdvice'.

- Pre-Routing: Advices registered to this joinpoint are called immediately after the request is created and before it matches any handler paths. Advices for the joinpoint have two call signatures, `void(const HttpRequestPtr &,AdviceCallback &&,AdviceChainCallback &&)` and `void(const HttpRequestPtr &)`, the previous one is exactly the same as the call signature of the filter's `doFilter` method. In fact, they all run in the same way (please refer to [05-Filter]), users can intercept the client request or let it pass through this joinpoint. The advice with second call signature has no interception capability, but the overhead of it is lower, if the user does not intend to intercept requests, please select this kind of advices. The registration interface is `registerPreRoutingAdvice`;

- Post-Routing: Advices registered to this joinpoint are called immediately after the request matches a handler path, The call signatures of Advices are the same as the above joinpoint's. The registration interface is `registerPostRoutingAdvice`;
- Pre-Handling: Advices registered to this joinpoint are called immediately after the request is approved by all filters and before it is handled, The call signatures of Advices are the same as the above joinpoint's. The registration interface is `registerPostRoutingAdvice`;
- Post-Handling: Advices registered to this joinpoint are called immediately after the request is handled and a response object is created by the handler, The call signature of Advices is `void(const HttpRequestPtr &, const HttpResponsePtr &)`, The registration interface is `registerPostHandlingAdvice`;

AOP schematic

The following figure shows the location of the above four joinpoints in the HTTP Requests processing flow, where the red dots represent the joinpoints and the green arrows represent the asynchronous calls.

AOP

Next: Benchmarks

Other languages: 繁體中文

Benchmarks

As a C++ Http application framework, performance should be one of the focus of attention. This section introduces Drogon's simple tests and achievements;

Test environment

- The system is Linux CentOS 7.4;
- The device is a Dell server, the CPU is two Intel(R) Xeon(R) CPUs E5-2670 @ 2.60GHz, 16 cores and 32 threads;
- Memory 64GB;
- gcc version 7.3.0;

Test plan and results

We just want to test the performance of the drogon framework, so we want to simplify the controller's processing as much as possible. We only do an HttpSimpleController and register it on the `/benchmark` path. The controller returns `<p>Hello, world!</p>` for any request. Set the number of drogon threads to 16. The processing function is as follows and you can find the source code at the `drogon/examples/benchmark` path:

```
void BenchmarkCtrl::asyncHandleHttpRequest(const HttpRequestPtr &req,
std::function<void (const HttpResponsePtr &)> &&callback)
{
    //write your application logic here
    auto resp = HttpResponse::newHttpResponse();
    resp->setBody("<p>Hello, world!</p>");
    resp->setExpiredTime(0);
    callback(resp);
}
```

For comparison, I chose nginx for comparison testing, wrote a `hello_world_module`, and compiled it with the nginx source. The nginx worker_processes parameter is set to 16.

The test tool is `httpress`, a good performance HTTP stress test tool.

We adjust the parameters of httpress, test each set of parameters five times, and record the maximum and minimum values of the number of requests processed per second. The test results are as follows:

Command line	Description	Drogon(kQPS)	nginx(kQPS)
httpress -c 100 -n 1000000 -t 16 -k -q URL	100 connections, 1 million requests, 16 threads,Keep-Alive	561/552	330/329
httpress -c 100 -n 1000000 -t 12 -q URL	100 connections, 1 million requests, 12 threads, no Keep-Alive	140/135	31/49

Command line	Description	Drogon(kQPS)	nginx(kQPS)
httpress -c 1000 -n 1000000 -t 16 -k -q URL	1000 connections, 1 million requests, 16 threads,Keep-Alive	573/565	333/327
httpress -c 1000 -n 1000000 -t 16 -q URL	1000 connections, 1 million requests, 16 threads,no Keep-Alive	155/143	52/50
httpress -c 10000 -n 4000000 -t 16 -k -q URL	10000 connections, 4 million requests, 16 threads,Keep-Alive	512/508	316/314
httpress -c 10000 -n 1000000 -t 16 -q URL	10000 connections, 1 million requests, 16 threads,no Keep-Alive	143/141	43/40

As you can see, using the Keep-Alive option on the client side, drogon can process more than 500,000 requests per second in the case where a connection can send multiple requests. This score is quite good. In the case that each request initiates a connection, CPU time will be spent on TCP connection establishment and disconnection, and the throughput will drop to 140,000 requests per second, which is reasonable.

It's easy to see that drogon has a clear advantage over nginx in the above test. If someone does a more accurate test, please correct me.

The image below is a screenshot of a test:

Test Result

Next: Causal profiling with coz

Other languages: 繁體中文

Causal profiling with Coz

With Coz you can profile two things:

- throughput
- latency

If you want to profile throughput of your application, you should switch on the `COZ_PROFILING` cmake option and include debug information in your executable with `Debug` or `RelWithDebInfo` release modes in cmake. Doing so will include coz progress points when serving a request. Profiling latency is currently not supported in whole application scope, but can still be done in user code.

When you're done compiling you application with progress points included. You need to run the executable with the coz profiler, for example `coz run --- [path to your executable]`.

Lastly, the application needs to be stressed, for best results you need to stress all code paths and run the profile for a good amount of time, 15+ min.

The final profile will be a `profile.coz` file created in the current working directory. To view results, open the profile in the official viewer, or you could run a local copy from the official git repo.

Coz also supports scoping source files included for the profile with `--source-scope <pattern>` or `-s <pattern>` among other things, that should prove useful.

For more information checkout:

- `coz run --help`
- Git repo
- Coz whitepaper

Next: Brotli compression

Other languages: 繁體中文

Brotli Info

Drogon supports out of the box brotli statically compressed files if it finds next to asset the corresponding brotli compressed asset.

So for instance, Drogon will search `/path/to/asset.js.br` for the request `/path/to/asset.js`.

It does so by setting `br_static` to `true` by default in `config.json` file.

if you want to dynamically compress with brotli you'll have to set `use_brotli` to `true` in `config.json`.

Users who don't intend to use brotli static, might want to get rid of brotli extra 'sibling check' by setting `br_static` to `false` in `config.json`.

Next: Coroutines

Other languages: 繁體中文

Coroutines

Drogon supports C++ coroutines starting from version 1.4. They provide a way to flatten the control flow of asynchronous calls, i.e. escaping the callback hell. With it, asynchronous programming becomes as easy as synchronous programming.

Terminology

This page isn't intended to explain what is a coroutine nor how it works. But to show how to use coroutines in drogon. The usual vocabulary tends get messy as subroutines (functions) uses the same terminology as coroutine does, yet they have slightly different meanings. The fact that C++ coroutines can act as if they are functions doesn't help either. To reduce confusion, we'll use the terminology that follows - it is by no means perfect, but it is good enough.

Coroutine is a function that can suspend execution then resume.
Return means a function finishing execution and fiving a return value to its caller. Or a coroutine generating a *resumable* object; which can be used to resume the coroutine.
Yielding is when a coroutine generates a result for the caller.
co-return means a coroutine yields and then exits.
(co-)awaiting means the thread is waiting for a coroutine to yield. The framework is free to use the thread for other purpose while awaiting.

Enabling coroutines

The coroutine feature in Drogon is header-only. This means the application can use coroutines even if Drogon is built without coroutine support. How to enable coroutines depends on the compiler used. In GCC >= 10 it can be enabled by setting `-std=c++20 -fcoroutines` while with MSVC (tested on MSVC 19.25) it can be enabled with `/std:c++latest` and `/await` must not be set.

Note that Drogon's implementation of coroutines won't work on clang (as of clang 12.0). GCC 11 enables coroutines by default when C++20 is enabled. And though GCC 10 does compile coroutines, it contains a compiler bug causing nested coroutine frames not being released; leading to potential memory leak.

Using coroutines

Each and every coroutine in drogon is suffixed with `Coro`. E,g. `db->execSqlSync()` becomes `db->execSqlCoro()`. `client->sendRequest()` becomes `client->sendRequestCoro()`, so on and so forth. All coroutines return an *awaitable* object. Then `co_await` on the object results in a value. The framework is free to use the thread to process IO and tasks when it's awaiting results to arrive - that's the beauty of coroutines. The code looks synchronous; but it's in fact asynchronously.

For example, querying the number of users that exist in the database:

```
app.registerHandler("/num_users",
    [](HttpRequestPtr req, std::function<void(const HttpResponsePtr&)>
callback) -> Task<>
    //                                      Must mark the return type as a
_resumable_ ^^^
{
    auto sql = app().getDbClient();
    try
    {
        auto result = co_await sql->execSqlCoro("SELECT COUNT(*) FROM
users;");
        size_t num_users = result[0][0].as<size_t>();
        auto resp = HttpResponse::newHttpResponse();
        resp->setBody(std::to_string(num_users));
        callback(resp);
    }
    catch(const DrogonDbException &err)
    {
        // Exception works as sync interfaces.
        auto resp = HttpResponse::newHttpResponse();
        resp->setBody(err.base().what());
        callback(resp);
    }
    // No need to return anything! This is a coroutine that yields `void`.
Which is
    // indicated by the return type of Task<void>
    co_return; // If want to (not required), use co_return
}
```

Notice a few important points:

1. Any handler that calls a coroutine MUST return a *resumable*
   - Turning the handler itself into a coroutine
2. `co_return` replaces `return` in a coroutine
3. Most parameters are passed by value

A *resumable* is an object following the coroutine standard. Don't worry too much about the details. Just know that if you want the coroutine to yield something typed `T`, then the return type will be `Task<T>`.

Passing most parameters by value is a direct consequence of coroutines being asynchronous. It's impossible to track when a reference goes out of scope as the object may destruct while the coroutine is waiting. Or the reference may live on another thread. Thus, it may destruct while the coroutine is executing.

It makes sense to not have the callback but to use the straightforward `co_return`. Which is supported, but may cause up to 8% of throughput under certain conditions. Please consider the performance drop and whether it's too great for the use case. Again, the same example:

```
app.registerHandler("/num_users",
    [](HttpRequestPtr req) -> Task<HttpResponsePtr>)
    //          Now returning a response ^^^
{
    auto sql = app().getDbClient();
    try
    {
        auto result = co_await sql->execSqlCoro("SELECT COUNT(*) FROM users;");
        size_t num_users = result[0][0].as<size_t>();
        auto resp = HttpResponse::newHttpResponse();
        resp->setBody(std::to_string(num_users));
        co_return resp;
    }
    catch(const DrogonDbException &err)
    {
        // Exception works as sync interfaces.
        auto resp = HttpResponse::newHttpResponse();
        resp->setBody(err.base().what());
        co_return resp;
    }
}
```

Calling coroutines from websocket controllers isn't supported yet. Feel free to open an issue if you need this feature.

Common pitfalls

There are some common pitfalls you may encounter when using coroutines.

- **Launching coroutines with lambda capture from a function**

  Lambda captures and coroutines have separate lifetimes. A coroutine lives until the coroutine frame is destructed. While lambdas commonly destruct right after being called. Thus, due to the asynchronous nature of coroutines, the coroutines's lifetime can be much longer than the lambda, for example in SQL execution. The lambda destructs right after awaiting for SQL to complete (and returns to the event loop to process other events), while the coroutine frame is awaiting SQL. Thus the lambda will have been destructed when SQL has finished.

  Instead of

  ```
  app().getLoop()->queueInLoop([num] -> AsyncTask {
      auto db = app().getDbClient();
      co_await db->execSqlCoro("DELETE FROM customers WHERE last_login < CURRENT_TIMESTAMP - INTERVAL $1 DAY". std::to_string(num));
      // The lambda object, thus captures destruct right at awaiting. They are destructed at this point
      LOG_INFO << "Remove old customers that have no activity for more than " << num << "days"; // use-after-free
  });
  // BAD, This will crash
  ```

  Drogon provides `async_func` that wraps around the lambda to ensure its lifetime

```
app().getLoop()->queueInLoop(async_func([num] -> Task<void> {
//                               ^^^^^^^^^^^^^^^^^^^^^^^^^ wrap with
async_func and return a Task<>
    auto db = app().getDbClient();
    co_await db->execSqlCoro("DELETE FROM customers WHERE last_login <
CURRENT_TIMESTAMP - INTERVAL $1 DAY", std::to_string(num));
    LOG_INFO << "Remove old customers that have no activity for more
than " << num << "days";
}));
// Good
```

- **Passing/capturing references into coroutines from function**

  It's a good practice in C++ to pass objects by reference to reduce unnecessary copy. However passing by reference into a coroutine from a function commonly causes issues. This is caused by the the coroutine is in fact asynchronous and can have a much longer lifetime compared to a regular function. For example, the following code crashes

```
void removeCustomers(const std::string& customer_id)
{
    async_run([&customer_id] {
        //      ^^^^ DO NOT pass/capture objects by reference into a
coroutine
        // Unless you are sure the object has a longer lifetime than
the coroutine

        auto db = app().getDbClient();
        co_await db->execSqlCoro("DELETE FROM customers WHERE
customer_id = $1", customer_id);
        // `customer_id` goes out of scope right at awaiting SQL.
Crashes here
        co_await db->execSqlCoro("DELETE FROM orders WHERE customer_id
= $1", customer_id);
    }
}
```

  However passing objects as reference from a coroutine is considered a good practice

```
Task<> removeCustomers(const std::string& customer_id)
{
    auto db = app().getDbClient();
    co_await db->execSqlCoro("DELETE FROM customers WHERE customer_id
= $1", customer_id);
    co_await db->execSqlCoro("DELETE FROM orders WHERE customer_id =
$1", customer_id);
}

Task<> findUnwantedCustomers()
```

```
{
    auto db = app().getDbClient();
    auto list = co_await db->execSqlCoro("SELECT customer_id from 
customers "
        "WHERE customer_score < 5;");
    for(const auto& customer : list)
        co_await 
removeCustomers(customer["customer_id"].as<std::string>());
        //                               ^^^^^^^^^^^^^^^^^
        // This is perfectly fine and preferred although it's a const 
reference
        // since we are calling it from a coroutine
}
```

Next: Redis

Other languages: 繁體中文

Redis

Drogon supports Redis, a very fast, in-memory data store. Which could be used as a database cache or a message broker. Like everything in Drogon, Redis connections are asynchronous. Which ensures Drogon running with very high concurrency even under heavy load.

Redis support depends on the `hiredis` library. Redis support won't be available if hiredis is not available when building Drogon.

Creating a client

Redis clients can be created and retrieve pragmatically through `app()`.

```
app().createRedisClient("127.0.0.1", 6379);
...
// After app.run()
RedisClientPtr redisClient = app().getRedisClient();
```

Redis clients can also be created via the configuration file.

```
    "redis_clients": [
        {
            //name: Name of the client,'default' by default
            //"name":"",
            //host: Server IP, 127.0.0.1 by default
            "host": "127.0.0.1",
            //port: Server port, 6379 by default
            "port": 6379,
            //passwd: '' by default
            "passwd": "",
            //db index: 0 by default
            "db": 0,
            //is_fast: false by default, if it is true, the client is faster but user can't call any synchronous interface of it and can't use it outside of the IO threads and the main thread.
            "is_fast": false,
            //number_of_connections: 1 by default, if the 'is_fast' is true, the number is the number of connections per IO thread, otherwise it is the total number of all connections.
            "number_of_connections": 1,
            //timeout: -1.0 by default, in seconds, the timeout for executing a command.
            //zero or negative value means no timeout.
            "timeout": -1.0
        }
    ]
```

Using Redis

execCommandAsync executes Redis commands in an asynchronous manner. It takes at least 3 parameters, the first and second are callback whom will be called when the Redis command succeed or failed. The third being the command it self. The command could be a C-style format string. And the rests are arguments for the format string. For example, to set the key `name` to `drogon`:

```
redisClient->execCommandAsync(
    [](const drogon::nosql::RedisResult &r) {},
    [](const std::exception &err) {
        LOG_ERROR << "something failed!!! " << err.what();
    },
    "set name drogon");
```

Or set `myid` to `587d-4709-86e4`

```
redisClient->execCommandAsync(
    [](const drogon::nosql::RedisResult &r) {},
    [](const std::exception &err) {
        LOG_ERROR << "something failed!!! " << err.what();
    },
    "set myid %s", "587d-4709-86e4");
```

The same execCommandAsync can also retrieve data from Redis.

```
redisClient->execCommandAsync(
    [](const drogon::nosql::RedisResult &r) {
        if (r.type() == RedisResultType::kNil)
            LOG_INFO << "Cannot find variable associated with the key 'name'";
        else
            LOG_INFO << "Name is " << r.asString();
    },
    [](const std::exception &err) {
        LOG_ERROR << "something failed!!! " << err.what();
    },
    "get name");
```

Transaction

Redis transaction allows multiple commands to be executed in a single step. All commands within a transaction are executed in order, no commands by other clients can be executed **in middle** of a transaction. Note that a transaction is not atomic. This means that after receiving the exec command, the transaction will be executed, If any command in the transaction fails to execute, the rest of the commands will still be executed. redis transactions do not support rollback operations.

The newTransactionAsync method creates a new transaction. Then the transaction could be used just like a normal RedisClient. Finally, the RedisTransaction::execute method executes said transaction.

```
redisClient->newTransactionAsync([](const RedisTransactionPtr &transPtr) {
    transPtr->execCommandAsync(
        [](const drogon::nosql::RedisResult &r) { /* this command works */
},
        [](const std::exception &err) { /* this command failed */ },
    "set name drogon");

    transPtr->execute(
        [](const drogon::nosql::RedisResult &r) { /* transaction worked */
},
        [](const std::exception &err) { /* transaction failed */ });
});
```

Coroutines

Redis clients support coroutines. One should use the GCC 11 or a newer compiler and use `cmake -DCMAKE_CXX_FLAGS="-std=c++20"` to enable it. See the coroutine section for more information.

```
try
{
    auto transaction = co_await redisClient->newTransactionCoro();
    co_await transaction->execCommandCoro("set zzz 123");
    co_await transaction->execCommandCoro("set mening 42");
    co_await transaction->executeCoro();
}
catch(const std::exception& e)
{
    LOG_ERROR << "Redis failed: " << e.what();
}
```

Next: Testing Framework

Other languages: 繁體中文

Testing Framework

DrogonTest is a minimal testing framework built into drogon to enable easy asynchronous testing as well as synchronous ones. It is used for Drogon's own unittests and integration tests. But could also be used for testing applications built with Drogon. The syntax of DrogonTest is inspired by both GTest and Catch2.

You don't have to use DrogonTest for your application. Use whatever you are comfortable with. But it is an option.

Basic testing

Let's start with a simple example. You have a synchronous function that computes the sum of natural numbers up to a given value. And you want to test it for correctness.

```
// Tell DrogonTest to generate `test::run()`. Only defined this in the main file
#define DROGON_TEST_MAIN
#include <drogon/drogon_test.h>

int sum_all(int n)
{
    int result = 1;
    for(int i=2;i<n;i++) result += i;
    return result;
}

DROGON_TEST(Sum)
{
    CHECK(sum_all(1) == 1);
    CHECK(sum_all(2) == 3);
    CHECK(sum_all(3) == 6);
}

int main(int argc, char** argv)
{
    return drogon::test::run(argc, argv);
}
```

Compile and run... Well, it passed but there's an obvious bug isn't it. `sum_all(0)` should have been 0. We can add that to our test

```
DROGON_TEST(Sum)
{
    CHECK(sum_all(0) == 0);
    CHECK(sum_all(1) == 1);
    CHECK(sum_all(2) == 3);
    CHECK(sum_all(3) == 6);
}
```

Now the test fails with:

```
In test case Sum
↳ /path/to/your/test/main.cc:47  FAILED:
  CHECK(sum_all(0) == 0)
With expansion
  1 == 0
```

Notice the framework printed the test that failed and the actual value at both ends of the expression. Allows us to see what's going on immediately. And the solution is simple:

```
int sum_all(int n)
{
    int result = 0;
    for(int i=1;i<n;i++) result += i;
    return result;
}
```

Types of assertions

DrogonTest comes with a variety of assertions and actions. The basic `CHECK()` simply checks if the expression evaluates to true. If not, it prints to console. `CHECK_THROWS()` checks if the expression throws an exception. If it didn't, print to console. etc.. On the other hand `REQUIRE()` checks if a expression is true. Then return if not, preventing expressions after the test being executed.

action if fail/expression	is true	throws	does not throw	throws certain type
nothing	CHECK	CHECK_THROWS	CHECK_NOTHROW	CHECK_THROWS_AS
return	REQUIRE	REQUIRE_THROWS	REQUIRE_NOTHROW	REQUIRE_THROWS_AS
co_return	CO_REQUIRE	CO_REQUIRE_THROWS	CO_REQUIRE_NOTHROW	CO_REQUIRE_THROWS_AS
kill process	MANDATE	MANDATE_THROWS	MANDATE_NOTHROW	MANDATE_THROWS_AS

Let's try a slightly practical example. Let's say you're testing if the content of a file is what you're expecting. There's no point to further test if the program failed to open the file. So, we can use `REQUIRE` to shorten and reduce duplicated code.

```
DROGON_TEST(TestContent)
{
    std::ifstream in("data.txt");
    REQUIRE(in.is_open());
    // Instead of
    // CHECK(in.is_open() == true);
    // if(in.is_open() == false)
    //    return;

    ...
}
```

Likewise, CO_REQUIRE is like REQUIRE. But for coroutines. And MANDATE can be used when an operation failed and it modifies an unrecoverable global state. Which the only logical thing to do is to stop testing completely.

Asynchronous testing

Drogon is a asynchronous web framework. It only follows DrogonTest supports testing asynchronous functions. DrogonTest tracks the testing context through the TEST_CTX variable. Simply capture the variable **by value**. For example, testing if a remote API is successful and returns a JSON.

```
DROGON_TEST(RemoteAPITest)
{
    auto client = HttpClient::newHttpClient("http://localhost:8848");
    auto req = HttpRequest::newHttpRequest();
    req->setPath("/");
    client->sendRequest(req, [TEST_CTX](ReqResult res, const HttpResponsePtr& resp) {
        // There's nothing we can do if the request didn't reach the server
        // or the server generated garbage.
        REQUIRE(res == ReqResult::Ok);
        REQUIRE(resp != nullptr);

        CHECK(resp->getStatusCode() == k200OK);
        CHECK(resp->contentType() == CT_APPLICATION_JSON);
    });
}
```

Coroutines have to be wrapped inside AsyncTask or called through sync_wait due to no native support of coroutines and C++14/17 compatibility in the testing framework.

```
DROGON_TEST(RemoteAPITestCoro)
{
    auto api_test = [TEST_CTX]() {
        auto client = HttpClient::newHttpClient("http://localhost:8848");
        auto req = HttpRequest::newHttpRequest();
        req->setPath("/");

        auto resp = co_await client->sendRequestCoro(req);
        CO_REQUIRE(resp != nullptr);
        CHECK(resp->getStatusCode() == k200OK);
        CHECK(resp->contentType() == CT_APPLICATION_JSON);
    };

    sync_wait(api_test());
}
```

Starting Drogon's event loop

Some tests need Drogon's event loop running. For example, unless specified, HTTP clients runs on Drogon's global event loop. The following boilerplate handles many edge cases and guarantees the event loop is running before any test starts.

```
int main()
{
    std::promise<void> p1;
    std::future<void> f1 = p1.get_future();

    // Start the main loop on another thread
    std::thread thr([&]() {
        // Queues the promise to be fulfilled after starting the loop
        app().getLoop()->queueInLoop([&p1]() { p1.set_value(); });
        app().run();
    });

    // The future is only satisfied after the event loop started
    f1.get();
    int status = test::run(argc, argv);

    // Ask the event loop to shutdown and wait
    app().getLoop()->queueInLoop([]() { app().quit(); });
    thr.join();
    return status;
}
```

CMake integration

Like most testing frameworks, DrogonTest can integrate itself into CMake. The `ParseAndAddDrogonTests` function adds tests it sees in the source file to CMake's CTest framework.

```
find_package(Drogon REQUIRED) # also loads ParseAndAddDrogonTests
add_executable(mytest main.cpp)
target_link_libraries(mytest PRIVATE Drogon::Drogon)
ParseAndAddDrogonTests(mytest)
```

Now the test could be ran through build system (Makefile in this case).

```
❯ make test
Running tests...
Test project path/to/your/test/build/
      Start  1: Sum
 1/1  Test  #1: Sum ....................................   Passed    0.00 sec
```

Other languages: **繁體中文**

FAQ

This is a list of commonly asked questions and the answers, with some extended explanation.

What's drogon's threading model and best practices?

Drogon runs on a thread pool where the HTTP server threads as well as DB threads are created when `app().run()` is called. It is a sequential task based system. Thus it is recommended to always use the asynchronous APIs or coroutines when possible. See Understanding drogon's threading model for detail.

Other languages: 繁體中文

Understanding Drogon's threading model

Drogon is a fast C++ web app framework. It's fast in part by not abstracting the underlying threading model away. However, it also causes some confusions. It's not uncommon to see issues and discussions about why responses are only sent after some blocking call, why calling a blocking networking function on the same event loop blocks causes a deadlock, etc.. This page aims to explain the exact condition that causes them and how to avoid them.

Event loops and threads

Drogon runs on a thread pool where each thread has it's own event loop. And event loops are at the core of Drogon. Every drogon application has at least 2 event loops. A main loop and a worker loop. As long as you are not doing anything funny. The main loop always runs on the main thread (the thread that started `main`). And it is responsible for starting all worker loops. Take the hello world app for example. The line `app().run()` starts the main loop on the main thread. Which in turn spawns 3 worker thread/loop.

```
#include <drogon/drogon.h>
using namespace drogon;

int main()
{
    app().registerHandler("/", [](const HttpRequest& req
        , std::function<void (const HttpResponsePtr &)> &&callback) {
        auto resp = HttpResponse::newHttpResponse();
        resp->setBody("Hello wrold");
        callback(resp);
    });
    app().addListener("0.0.0.0", 80800);
    app().setNumThreads(3);
    app().run();
}
```

The thread hierarchy looks like this

```
            .-------------.
            | app().run() | <main thread>
            :-------------:
                  |
            .-----v------.
            | MAIN LOOP  |
            :------------:
                  | Spawns
      .-----------+--------------.
      |           |              |
.-----v----.   .--v-------.  .---v----.
```

```
| Worker 1 |   | Worker 2 |  | etc... |
:----------:   :----------:  :--------:
 <thread 1>     <thread 2>   <thread ...>
```

The number of worker loops depends on numerous variables. Namely, how many threads are specified for the HTTP server, how many non-fast DB and NoSQL connections are created - we'll get to fast vs non-fast connections later. Just know that drogon has more threads than just the HTTP server threads. Each event loop is essentially a task queue with the following functionality.

- Reads tasks from a task queue and execute them. You can submit task to run on the a loop from any other threads. Task submitting is totally lock free (thanks to lock free data structure!) and won't cause data race in all circumstances. Event loops process tasks one-by-one. Thus tasks have a well-defined order of execution. But also, tasks that's queued after a huge, long running task gets delayed.
- Listen to and dispatch network events that it manages
- Execute timers when they timeout (usually created by the user)

When non of the above is happening. The event loop/thread blocks and waits for them.

```
// queuing two tasks on the main loop
trantor::EventLoop* loop = app().getLoop();
loop->queueInLoop([]{
    std::cout << "task1: I'm gonna wait for 5s\n";
    std::this_thread::sleep_for(5s);
    std::cout << "task1: hello!\n";
});
loop->queueInLoop([]{
    std::cout << "task2: world!\n";
});
```

Hopefully is's clear why running the above snippet will result in `task1: I'm gonna wait for 5s` appear immediately. Pauses for 5 seconds and then both `task1: hello` and `task2: world!` showing up.

> So tip 1: Don't call blocking IO in the event loop. Other tasks has to wait for that IO.

Network IO in practice

Almost everything in drogon is associated with an event loop. This includes the TCP stream/connection, HTTP client, DB clients and data cache. To avoid race conditions, all IO are done in the associated event loop. If an IO call is made from another thread, then parameters are stored and submitted as a task to the appropriate event loop. This has some implications. For example when calling a remote endpoint from within a HTTP handler or making a DB call. The callback from the client may not necessarily (in fact, commonly does not) runs on the same thread as the handler.

```
app().registerHandler("/send_req", [](const HttpRequest& req
    , std::function<void (const HttpResponsePtr &)> &&callback) {
    // This handler will run on one of the HTTP server threads
    // Create a HTTP client that runs on the main loop
    auto client = HttpClient::newHttpClient("https://drogon.org",
app().getLoop());
    auto request = HttpRequest::newHttpRequest();
    client->sendRequest(request, [](ReqResult result, const HttpResponse&
resp) {
        // This callback runs on the main thread
    });
});
```

Therefor, it is possible to clog up an event loop if you are not aware of what your code is actually doing. Like creating a ton of HTTP clients on the main loop and sending all out-going requests. Or running compute-heavy functions in a DB callback. Stopping other DB queries being processed.

```
 Worker 1       Main Loop        Worker2
.---------.    .----------.     .---------.
|         |    |          |     |         |
| req 1-. |    |----------|  .--+--req 2  |
|       :-+----+->        |  |  |         |
|         |    | send http|  |  |---------|
|---------| a.-|   req 1  |  |  |         |
|other req| s| |----------|  |  |         |
|---------| y| |      <---+--:  |         |
|         | n| |send http |     |         |
|         | c| |  req 2   |-.   |         |
|         |  | |----------| |a  |---------|
|         |  | |http resp1| |s  |other req|
|         |  :-|>compute  | |y  |---------|
|         |    |          | |n  |         |
|         | .--+-generate | |c  |         |
|         | |  | response | |   |         |
|         | |  |----------| |   |         |
|         | |  |http resp2|<:   |         |
|---------| |  | compute  |     |---------|
|response<|-:  |          |-----|>        |
|send back|    | generate |     |send resp|
|         |    | response |     | back    |
:---------:    :----------:     :---------:
```

The same principle is also true for the HTTP server. If a response is generated from a separate thread (ex: in a DB callback). Then the response is queue on associated thread for sending instead of sending immediately.

> Tip 2: Be aware where you place your computations. They can also harm throughput if not careful.

Deadlocking the event loop

With the understanding on how Drogon is designed. It shouldn't be hard to see how to deadlock an event loop. You simply submit a remote IO request and wait on it on the same loop - The sync API is exactly that. It submits an IO request and waits for the callback.

```
app().registerHandler("/dead_lock", [](const HttpRequest& req
    , std::function<void (const HttpResponsePtr &)> &&callback) {
    auto currentLoop = app().getIOLoops()[app().getCurrentThreadIndex()];
    auto client = HttpClient::newHttpClient("https://drogon.org",
currentLoop);
    auto request = HttpRequest::newHttpRequest();
    auto resp = client->sendRequest(resp); // DEADLOCK! calling the sync
interface
});
```

Which could be visualized as

```
     Some loop
   .------------.
   | new client |
   | new request|
   |send request|
 .->WAIT resp---+-.
 | |  ....      | |
?| |            | |
?| |------------| |
?| |            | |
?| |            | |
?| |            | |
?| |            | |
 | |            | |
 | |            | |
 | |            | |
 | |            | |
 | |            | |
 | |            | |
 | |------------| |
 | | read resp  | |
 :-+-         <-+-:
   | oops       |
   | deadlock   |
   :------------:
```

The same is true for everything else too. DB, NoSQL callbacks, you name it. Fortunately non-fast DB clients runs on their own threads; each client gets their own thread. Thus it's safe to make synchronous DB queries from a HTTP handler. However you should not run sync DB queries inside the callback of the same client. Otherwise the same thing happens.

> Tip 3: Sync APIs are evil for both performance and safety. Avoid them like the plague. If you have to, make sure you run the client on a separate thread.

Fast DB clients

Drogon is designed for performance first, ease of use kinda second. Fast DB clients are DB clients that shares the HTTP server threads. Theses improves performance by eliminating the need to submit requests to another thread and avoids content switch by the OS. However due to that fact. You cannot make sync queries on them. It'll deadlock the event loop.

Coroutines to the rescue

A dilemma in Drogon's development and use has being that, async APIs are more efficient but annoying to use. While sync APIs can be problematic and slow. But they are easy to program for. Lambda declaration can be long and tedious. And the syntax is just not fun to look at. Besides the code doesn't run top-to-bottom; it's full of callbacks. The sync API is much cleaner than the async once. At the cost of performance.

```
// drogon's async DB API
auto db = app().getDbClient();
db->execSqlAsync("INSERT......", [db, callback](auto result){
    db->execSqlAsync("UPDATE .......", [callback](auto result){
        // Handle success
    },
    [callback](const DbException& e) {
        // handle failure
    })
},
[callback](const DbException& e){
    // handle failure
})
```

versus

```
// drogon's sync API. Exception can be handled automatically by the framework
db->execSqlSync("INSERT.....");
db->execSqlSync("UPDATE.....");
```

There must be a way to have the cake an eat it too right? Enter C++20's coroutines. Essentially they are a first-class, compiler supported wrapper around callbacks to make your code look like it's synchronous. But in fact async all the way. Here's the same code in coroutines.

```
co_await db->execSqlCoro("INSERT.....");
co_await db->execSqlCoro("UPDATE.....");
```

It's exactly like the sync API! But better in almost every way. You get all the benefit of async but keep using the sync interface. How it actually works if out of the scope of this post. The maintainers urge to use coroutines when you can (GCC >= 11. MSVC >= 16.25). It is however, not quite magic. It does not solve clogging event loops and race conditions for you. However it's easier to debug and understand async code with coroutines.

Tip 4: Use coroutines when you can

Summary

- Use C++20 coroutines and `Fast DB` connections when you can
- Synchronous API can slow down or event deadlock the event loop
- If you have to use a synchronous API. Make sure they are associated with a different thread