Scaffold – C++ 网站开发框架
设计概述、使用指导与应用程序接口参考

Boyin Chen

2017 年 8 月 29 日

目录

第一章　概述

1.1　项目简介

Scaffold（脚手架）是一个 C++ 编写的网站开发框架（库），面向用户暴露的部分启发自著名 Node.js 网站应用框架 Express.js[1]，集成了 Web 开发所需的常用功能。该框架内置 HTTP(S) 支持（`HTTP/1.1`），底层使用 Cesanta 公司的纯 C 编写的 Mongoose[2] 嵌入式服务器，具有很强的可移植性。出于性能与 API 美观性考虑，使用 Scaffold 需要 C++11 的支持。

1.2　你好，世界！

让我们来从一个最简单直接的例子看看如何使用 Scaffold 进行网站开发。

```
#include <scaffold/Request.hpp>
#include <scaffold/Response.hpp>
#include <scaffold/Scaffold.hpp>
int main(void)
{
    auto app = Scaffold();
    app.get("/", [](Request &req, Response &res)
    {
        res.send("Hello World!");
    });
    app.listen(3000);
}
```

编译这段程序(记得链接 `scaffold`),并打开浏览器,访问 `http://localhost:3000/`。看看有什么结果输出?

[1]请参看 `https://expressjs.com`

[2]请参看 `https://cesanta.com`

请注意，此时在您的标准错误输出上将打印出服务器的访问日志。这不要紧，在之后的内容里将会介绍如何将日志写入到文件、写入到特定的文件指针（FILE *）或关闭日志器。

1.3　从源代码构建

1.3.1　解决依赖

Scaffold 有一些依赖关系需要手动解决。下列依赖是必须的：

- RapidJSON。从软件源中安装或访问它的 GitHub 仓库[3]。CMake 将会检查它。

下列依赖不是必须的，它们被一些插件或额外功能所需要：

- OpenSSL。从软件源中安装或访问它的官方网站[4]。CMake 将会检查它。请注意，目前 OpenSSL 1.1 不被支持，您可能需要安装一份较低版本。
- Boost C++ Libraries。从软件源中安装或访问它的官方网站[5]。
- MySQL Connector/C++。访问它的官方网站[6]，并下载源码包在本地构建。官网提供的二进制包可能有兼容性问题。

1.3.2　配置 CMake

在默认的情况下所有的插件和额外功能均会进行构建，若需要关闭，可定义一些额外的符号。如：

```
cmake .. -DCMAKE_BUILD_TYPE=RELEASE -DDISABLE_SSL=1
```

则禁用了 SSL 支持，也不必解决 OpenSSL 的依赖了。所有支持的选项及意义见下表：

表 1.1: CMake 选项及意义

选项	值为 0（或未定义）	值为 1
DYNAMIC_ONLY	构建动态库、静态库	仅构建动态库
DISABLE_SSL	启用 SSL 支持	禁用 SSL 支持
DISABLE_MYSQL	启用 MySQL 插件	禁用 MySQL 插件
DISABLE_SESSION	启用 Session 插件	禁用 Session 插件

[3]miloyip/rapidjson：https://github.com/miloyip/rapidjson

[4]OpenSSL 官方网站：https://www.openssl.org

[5]Boost 官方网站：http://www.boost.org

[6]MySQL 官方网站：https://dev.mysql.com/downloads/connector/cpp

1.3.3　构建后事项

一旦由 CMake 生成了 `Makefile` 便可使用 make 来构建。构建完成后的默认安装位置在`/usr/local`，因此安装时可能需要更高的权限，当然也可在配置 CMake 时指定其他的安装目录。

在 Linux 平台上，要使得动态链接库生效，还需运行 `ldconfig`。请先确认动态库所安装到的目录已存在于`/etc/ld.so.conf` 文件中。

1.4　使用方法

Scaffold 就是一个普通的库，没有特别的使用说明。考虑之前的章节 1.2 的示例代码，将它保存为 `app.cpp`，然后执行：

```
c++ app.cpp -lscaffold
```

便得到了动态链接到 Scaffold（`libscaffold.so`）的 `a.out` 可执行文件。

1.5　代码量统计

以下统计数据以 Git 仓库 `0b2f1069e07619160cd8d433969264bf6757a96f` 提交为准，使用 Tokei [7] 得出。

表 1.2: 代码统计数据

类型	文件数	总行数	代码行数	纯注释行数	空白行数
C++ 源文件	18	1696	1637	59	0
C++ 头文件	15	465	450	5	10
CMake	1	83	68	7	8
总计	34	2244	2155	71	18

[7] Aaronepower/tokei: https://github.com/Aaronepower/tokei

第二章　项目架构

2.1　基本组件

基本组件部分是 Scaffold 的主体，由 `scaffold`、`Request`、`Response`、`Router` 和 `Logger` 共 5 个类组成。它们之间存在一定的耦合关系，但不对用户造成影响。目前 `Router` 类不直接向用户暴露，它被 `scaffold` 所使用。

2.1.1　`scaffold` 类

`scaffold` 是单例[1]的，Scaffold 中只允许存在一个 HTTP 服务。`scaffold` 作为框架的主体，负责的任务是运行 HTTP 服务，针对每一个 HTTP 请求构建它的 `Request` 和 `Response` 对象，并通过路由找到对应的回调函数调用之，完成请求的处理。此外，它还负责 Session 的管理与访问日志记录。

在 HTTP 服务开始运行前，用户可以设置静态文件根目录、SSL 证书路径，并通过 `scaffold` 来添加路由规则、注册回调函数。

2.1.2　`Request` 类

`scaffold` 收到 HTTP 请求时会对请求进行处理，构造一个 `Request` 对象。`Request` 是便于编写业务代码的，在构造时，请求被分析，将其中的来源 IP、请求体、请求路径、主机名称、HTTP 方法提取出来，放入对应的属性中，便于用户使用。

`Request` 含有下列映射表：

- `post`：当 HTTP 请求的头中 `Content-Type` 为 `x-www-form-urlencoded` 时，表单内容将被解析到此处。
- `query`：当 HTTP 请求的 URL 含有查询字符串（query string）时，它的键–值对将被解析到此处。
- `params`：当该请求对应匹配到的路由规则含有参数时，这些参数将被解析到此处。

[1]严格地说不是，但 `scaffold` 有合适的拷贝构造函数和移动构造函数，使得程序运行的 HTTP 服务是有效的。调用 `Scaffold()` 可以获取单例的引用。

- `cookies`：请求附带的 Cookie 被解析到此处。
- `headers`：该映射表完整保留请求中的所有 HTTP 头信息。

此外，相关 Session 信息会自动匹配，并构造一个 `rapidjson::Document` 对象，在 `Request` 中对应的成员是 `Request::session`。这个 `rapidjson::Document` 可以直接使用，在请求结束时系统将处理好 Session 信息的保存。

2.1.3 Response 类

`scaffold` 收到 HTTP 请求时会处理好这个 TCP 连接，构造一个 `Request` 对象。`Request` 有许多便捷的方法供用户使用，来针对请求做出响应。

`Response` 参考了 `Express.js` 的实现，也针对性地按照需要实现、新增、删去了相应的 API。`Response` 可以便捷地操作响应头、Cookie、状态码、`Content-Type` 和跨域[2]设置；可以简单地完成页面跳转和重定向。对于响应的正文部分，除了最基本地动态正文发送（`Response::send`）外，还可直接发送与状态码对应的描述字符串、伺服静态文件或进行文件下载等操作。

2.1.4 Router 类

`Router` 是 `scaffold` 的一个组成部分，它是一个非常简单的路由器，每个请求通过它匹配出对应的回调函数。

路由规则可以带参数。例如存在路由规则`/hello/:name`，设请求为`/hello/Sharon`，则这个请求的 `Request` 的 `params` 将含有 `name => Sharon` 这一键值对。

若路由规则匹配失败：当 `scaffold` 的静态文件根目录被设置时，若对应静态文件存在或对应目录含有缺省文件[3]，对应的静态文件将被发送；其他情况下，响应的状态码将设为 `404`，并发出对应的说明字符串。

2.1.5 Logger 类

`Logger` 是一个日志器类，提供简单的日志功能。Scaffold 自带一个访问日志记录器[4]，它是位于 `scaf` 命名空间的 `accesslog`。用户可以关闭它，或者设置它写入到文件中。

用户可以创建 `Logger` 的实例用于其他日志。日志内容默认输出到标准错误输出（`stderr`），但用户可以设置写入文件或写入给定的 `FILE *`（不推荐）。用户也可在代码中随时停用、启用某个日志器。

---

[2] `Access-Control-Allow-Origin` 的便捷操作

[3] 目前的缺省文件定义为 `index.html`、`index.htm`、`default.html` 或 `default.htm`

[4] Nginx 风格，目前不可自定义

Logger 的 API 是 C 风格的，提供一次性格式化输出一行带时间的日志的方法，也提供普通的字符串输出（类似 puts）、字符输出（类似 putc）和格式化输出（类似 printf）等方法。

2.1.6 回调函数的形态

在 C++11 中，std::function 被引入，实现了函数、伪函数和 Lambda 表达式的大一统。Scaffold 中回调函数作为 std::function 声明，以便用户使用。

回调函数的形态是：std::function<void(Request&, Response&)>，该类型的别名为 callback_t。以下是对应的三种回调函数的实现。

（一）函数：

```
void callback(Request &req, Response &res)
{
    res.send("It works!");
}
```

（二）伪函数：

```
class callback
{
public:
    void operator () (Request &req, Response &res)
    {
        res.send("It works!");
    }
};
```

（三）Lambda 表达式：

```
auto callback = [](Request &req, Response &res)
{
    res.send("It works!");
};
```

其中 Lambda 表达式形式可用于路由规则注册时直接实现简单的响应回调。

2.2 插件

除了基础组件以外，其它的部分均放在 scaf 命名空间中，以免造成符号冲突。

部分插件与基础组件存在耦合关系，但仍可以通过进行 CMake 配置时开启相应的选项禁用指定的插件。

2.2.1 Session 会话操作

Session 的原理是，在服务器端保存数据，并向客户端发送一个含有 Session ID 信息的 Cookie，通常有效期是会话期。一个 Session 成功创建之后，服务端就可以保存有对应会话的一些状态，例如实现验证码。

Scaffold 的会话管理基于文件系统进行存储，使用 JSON 格式。解析采用 RapidJSON，在 Session 启用时 Scaffold 将自动管理会话数据，并且将它作为一个 `Document` 类型的成员置于 `Request` 对象中。会话的管理是完全自动的，用户只需简单地使用 `Request::session` 即可。

由于时间原因，目前 Session 数据的存放位置硬编码为 `/tmp/scsess` 目录。因此，Session 功能目前只能用于 UN*X 平台，或是在 `Cygwin`、`MSYS` 等 Windows 模拟环境下运行。该目录可以随时进行清理[5]，清理后会丢失所有已存在的会话信息。

2.2.2 MySQL 数据库操作

`MySQL` 插件是 MySQL Connector/C++ （`mysql-connector-c++`）的封装。以下组件均位于 `scaf` 命名空间。

MySQL 类

`scaf::MySQL` 封装了非常简单的、常用的数据库操作：连接、运行查询、进行 Prepared Statement 查询等。

mstream 流式封装

由于 MySQL Connector/C++ 的使用较为繁琐，Scaffold 实现了 `imysqlstream` 和 `omysqlstream` 两个类，帮助用户简单地操作数据库。其中尤以 `omysqlstream` 较为常用，以下是一个简单的例子。

```
auto db = new scaf::MySQL("127.0.0.1", 3306, "user", "pass");
auto oms = db->prepare("INSERT INTO employee (name, age) VALUES (?, ?);");
oms << "Cheehwa Tung" << 80;
try {
    db->execute(oms);
} catch(sql::SQLException e) {
    fputs(e.what(), stderr);
    delete db;
}
delete db;
```

[5]可以直接删除

2.3　杂项

以下内容均为项目内部使用的实用工具，均位于 `scaf` 命名空间。这些实用工具不对用户暴露。

2.3.1　基本组件相关

`RequestHelper` 和 `ResponseHelper` 用来帮助 `scaffold` 构造 `Request`、`Response` 对象。

2.3.2　字符串相关

- HTTP 日期串、时间戳互转。
- HTTP 状态码到描述字符串的映射。
- 若干 RFC 文件[6]中规定的文件后缀到 MIME 类型的映射。
- URL 编码、URL 解码。

2.3.3　文件系统相关

- 测试文件或目录是否存在[7]。
- 测试目录是否存在[8]。

2.3.4　其他

项目中实现了部分散列算法，目前有：`SHA-256`。

[6]RFC-2045 至 RFC-2049

[7]目前方法是利用 `access` 这个 API

[8]利用 `opendir` 测试

第三章　应用程序接口参考

3.1　scaffold 类

该类的头文件为 <scaffold/Scaffold.hpp>。

3.1.1　数据成员解释

表 3.1: scaffold 数据成员

类型	标识符	意义
mg_mgr *	mgr	HTTP 服务器的 mg_mgr 结构指针
mg_connection *	conn	用于运行服务器的 mg_connection 结构指针
string	sslKey	SSL 密钥的路径
string	sslCert	SSL 证书的路径
bool	isListening	是否已经启动服务器、监听端口

其中，string 类型指的是 std::string，下同。

3.1.2　构造函数与析构函数

由于 scaffold 是单例的，拷贝构造函数、移动构造函数的用途是正确处理用户使用 auto 声明时造成的拷贝或移动，保证仅有一个 HTTP 服务运行。

表 3.2: scaffold 的构造函数与析构函数

类型	参数列表	意义
默认构造函数	void	初始化 HTTP 服务的相关设置
拷贝构造函数	scaffold&	保证拷贝构造后单例运行
移动构造函数	scaffold&&	保证移动构造后单例运行
析构函数	void	关闭服务、释放内存

3.1.3　公共成员函数

- `static scaffold* getPointer(void);`
  返回 `scaffold` 单例的指针。
- `static scaffold& getReference(void);`
  返回 `scaffold` 单例的引用。
- `void setSSL(const string &cert, const string &key);`
  设置 SSL 证书与密钥的路径。
- `void setRoot(const string &root);`
  设置静态文件服务的根目录。
- `void setListing(bool enable);`
  设置是否允许在静态文件服务时列目录。
- `void listen(int port, bool ssl = false);`
  开始监听端口、启动 HTTP 服务。当第二个参数为 `true` 时将以 HTTPS 形式启动。

还有一部分路由相关的公共成员函数将在下一小节解释。

3.1.4　路由规则注册

Scaffold 支持 8 种 HTTP 方法[1]：

表 3.3: 8 种受支持的 HTTP 方法列表

GET	POST	PUT	DELETE
HEAD	TRACE	OPTIONS	CONNECT

相同的路由规则，针对不同的 HTTP 方法可以注册不同的回调函数。在之前的章节 2.1.4 中简单地介绍了 `scaf::Router` 的功能；章节 2.1.6 中介绍了回调函数的形态[2]。这里进行展开。

路由规则是一个字符串，现在的路由功能非常简单，仅分为不带参数和带参数两类：

1. 不带参数的路由规则，如`/hello`，是最简单的路由规则。当请求与规则完全一致时匹配成功。
2. 带参数的路由规则，如`/:name`，可以进行简单的参数提取。参数使用冒号（`:`）开头，后接参数名。斜杠（`/`）之间若有参数，则有且只能有一个参数。

[1]枚举 `HttpMethod` 定义了它们

[2]回调函数类型现在均使用 `callback_t` 表示

在上述例子中，可以发现，若请求为/hello，则它也是符合/:name 这个规则的。但是，在 Scaffold 的路由逻辑中会优先匹配到不带参数的规则上，请注意。

表 3.4: 9 个路由注册用途的成员函数

scaffold::get	scaffold::post	scaffold::put
scaffold::Delete	scaffold::head	scaffold::trace
scaffold::options	scaffold::connect	scaffold::all

scaffold 类中提供了一系列方法用于注册路由规则。请注意，用于 DELETE 方法的 scaffold::Delete 首字母为大写，因为 delete 是 C++ 的一个关键字。scaffold::all 用于将某规则注册到所有被支持的 HTTP 方法上。

3.2 Request 类

该类的头文件为 <scaffold/Request.hpp>。

3.2.1 数据成员解释

- mg_connection * conn
  私有成员，这个 HTTP 请求的 Mongoose 连接的指针。
- http_message * hm
  私有成员，MG_EV_HTTP_REQUEST 事件触发的回调，传入的 HTTP 请求信息指针。
- bool xhr
  判断是否为 AJAX 请求。若 HTTP 头中 X-Requested-With 为 XMLHttpRequest，则 xhr 的值为 true。Scaffold 的实现对这个头信息大小写敏感。
- string ip
  字符串表示的该连接的 IP 地址，有 IPv6 支持。
- string body
  该 HTTP 请求的正文。
- string path
  该 HTTP 请求的路径。
- string hostname
  该 HTTP 请求所请求的主机名。头中若有 X-Forwarded-Host 字段则采用它，否则采用 Host 字段的值。遵循 Express.js 的实现，端口号在这里被去除。

- `HttpMethod`[3] `method`
  该 HTTP 请求的方法。

- `Document`[4] `session`
  该客户端连接的会话信息的 RapidJSON 文档。可以直接读写，修改在请求结束时将由 Scaffold 负责保存。

- `map<string, string>`[5] `post`
  POST 请求的表单信息映射。当 `Content-Type` 为 `application/x-www-form-urlencoded` 时，将尝试把 POST 正文的请求串解析成键值对映射并进行 URL 解码。

- `map<string, string> query`
  将 HTTP 请求的查询串解析成键值对映射，并进行 URL 解码。

- `map<string, string> params`
  若匹配到的路由规则中含有参数，将把参数名称与值建立映射。注意，没有对值进行 URL 解码。

- `map<string, string> cookies`
  若 HTTP 请求头中含有 Cookie 信息，将把这些信息的键值建立映射，并进行 URL 解码。

- `map<string, string> headers`
  HTTP 请求中各个头字段的映射表。

对于 `path` 与 `query` 的解释，请看下面的某 HTTP 请求报文片段：

```
GET /foo/bar?param1=val1&param2=val2
    | path  |     query string     |
```

其中，问号既不属于 path 也不属于 query string，`query` 由 query string 解析而来。

3.2.2 公共成员函数

- `bool is(const string &type);`
  用于便捷判断请求的 `Content-Type`。参数 `type` 可以不用传入完整的 MIME 类型，例如 `application/json` 简写为 `json` 是允许的。

[3] `HttpMethod` 是 `Declaration.hpp` 中定义的枚举类型

[4] RapidJSON 库的文档类型，位于 `rapidjson` 命名空间

[5] `map` 指 `std::map`，下同

- `string get(const string &field);`
  快速地获得 HTTP 请求头中某个字段的值，若不存在该字段则返回空串。参数是大小写不敏感的，但建议传入大小写规范的参数以提高性能——该方法先在红黑树上以大小写敏感形式进行查找，未命中后才进行遍历式的大小写不敏感的查找。

3.3 Response 类

该类的头文件为 `<scaffold/Response.hpp>`。

3.3.1 数据成员解释

由于该类的特殊性质，向用户暴露的主要是方法，而非属性。下列成员均是私有的。

- `Request * req`
  `Request` 和 `Response` 总是成对出现的。`req` 是这个 `Response` 对应的 `Request` 对象的指针。

- `http_message * hm`
  `MG_EV_HTTP_REQUEST` 事件触发的回调，传入的 HTTP 请求信息指针。

- `mg_connection * conn`
  该请求对应的 Mongoose 连接的指针。

- `int statusCode`
  将发送回客户端的响应的状态码，默认为 200。

- `bool _typeSet`
  响应头的 `Content-Type` 是否已由用户调用 `Response::type` 自行重设。默认的 `Content-Type` 为 `text/html`[6]。

- `bool _headersSent`
  响应头是否已经发送出去。用户调用所有需要发送正文的方法时，`Response` 将自行构造响应头并发送。

- `bool _contentEnded`
  正文是否已经结束。用户调用 `Response::end` 后则视为正文已经发送结束。

- `map<string, Cookie> cookies`
  将要发送到客户端的 Cookie 设置信息。

- `map<string, string> headers`
  除 Cookie 外，将要发送到客户端的响应头映射。

[6]编码为 `utf-8`

3.3.2　公共成员函数

- `bool headersSent(void);`
  响应头已经发送出去则返回 `true`。该方法直接返回 `_headersSent` 私有成员。

- `void type(const string &type);`
  设置响应头的 `Content-Type`。若传入的是完整的 MIME 类型，则直接设为传入串；否则在根据标准制作的映射表中搜索缩写或后缀名对应的 MIME 类型。若查表失败，类型将设为 `application/octet-stream`。

- `string get(const string &key);`
  取得响应头中对应字段的值，若不存在，返回空串。参数大小写不敏感。

- `void set(const string &key, const string &value);`
  设置响应头的某字段。若有特殊字符需要自行对值进行 URL 编码。

- `void header(const string &key, const string &value);`
  与 `Response::set` 完全相同。

- `void crossOrigin(const string &dest);`
  跨域设置。将响应头的 `Access-Control-Allow-Origin` 字段设为 `dest`。

- `void cookie(const string &name, const string &value, bool forever);`
  快速进行 Cookie 的键–值设置。`forever` 参数默认值为 `false`，Cookie 有效期为会话期；当 `forever` 为 `true` 时，`Max-Age` 属性将被设为 `315360000`（10 年）。

- `void cookie(const string &name, Cookie &_cookie);`
  完整的可自定义所有属性的 Cookie 设置。

- `void clearCookie(const string &name);`
  清除将要发送在响应头中的已设置的所有 Cookie 信息。

- `void download(const string &file, const string &name);`
  向客户端发送路径为 `file` 的文件，并将头字段 `Content-Disposition` 中的 `filename` 设为 `name`（若有特殊字符需要自行进行 URL 编码）。

- `void link(const string &rel, const string &link);`
  向头字段 `Link` 中添加一条信息。`rel` 为关系，`link` 为链接。参数均不需要自行进行 URL 编码。

- `void location(string path);`
  将头字段 `Location` 设为 URL 编码后的 `link`。若 `link` 的值为`"back"`，则设为请求头中的 `Referer`；若请求头中不存在 `Referer` 字段，则设为`"/"`。

- `void redirect(string location, int status);`
  附带状态码设置的重定向。状态码的默认值为 `302`。

- `Response& status(int code);`
  遵循 Express.js 的设计。设置响应的状态码并返回自身的引用，若标准未定义该状态码则会设为 `500`。

- `Response* status(int code, bool);`
  同上，但返回的是自身的指针（即 `this` 指针）。`bool` 类型的参数用来区分重载。

- `Response& vary(const string &value);`
  遵循 Express.js 的设计。将 `value` 添加到响应头的 `Vary` 字段，并返回自身的引用。

- `Response* vary(const string &value, bool);`
  同上，但返回的是自身的指针（即 `this` 指针）。`bool` 类型的参数用来区分重载。

- `void sendStatus(int code);`
  发送对应状态码的描述字符串，如 `code` 为 `200` 时发送 `OK`。若标准未定义该状态码则不发送任何内容。

- `void send(const string &content);`
  向客户端动态发送内容。

- `void send(const char *content, int len);`
  向客户端动态发送内容。若未传入 `len`，则其默认值为 `-1`，这会调用 `strlen` 来计算 `content` 的长度。

- `void end(void);`
  标记已结束响应正文的发送。之后调用 `Response::send` 等方法也不会发送内容。

3.3.3　关于 Cookie 设置的解释

在 `Declaration.hpp` 中，结构 `Cookie` 定义如下：

```
struct Cookie
{
    string  value;
    time_t  expires;
    int     maxAge;
    string  path;
    string  domain;
```

```
        bool    secure;
        bool    httpOnly;
        Cookie(void):
            expires(0), maxAge(0),
            secure(false), httpOnly(false) { }
    };
```

从中我们可以看到各个属性的默认值。当调用第二个参数是字符串的 Response::cookie 重载时，仅有 Cookie 的值被设置，其他属性均为默认；若调用传入 Cookie 对象版的重载，则生成的 Set-Cookie 头字段以用户传入信息为准。

3.4 Logger 类

该类位于 scaf 命名空间，头文件为 <scaffold/Logger.hpp>。

3.4.1 公共成员函数

- void setFP(FILE *ptr);
  传入的 ptr 应当是一个有效的 FILE *，之后的日志内容将写入到此处。
- void setFile(const string &path);
  传入一个有效的文件路径 path，日志内容以追加的形式写入到该文件。
- void enable(void);
  启用该日志器的记录。在调用 Logger::disable 后欲重新启用日志器，应调用此方法。
- void disable(void);
  停用该日志器的记录。调用该方法后，该日志器的所有写入操作均不会执行。
- void log(const char *fmt, ...);
  以 printf 风格输出一行日志记录。记录前自动加入当前时间，行末自动换行。
- void printTime(void);
  将当前的时间以 [YYYY/mm/dd HH:ii:ss] 的形式输出到日志。
- void print(const char *fmt, ...);
  以 printf 风格输出日志记录。没有任何其他元素会被自动添加。
- void puts(const char *s);
  以 puts 风格输出一行日志记录。参数 s 末尾将自动加上换行。

- `void putchar(const char c);`
  以 `putchar` 风格输出一个字符。

以上所有输出相关的操作均会清空缓冲区，将数据保存。若一个日志器没有显式指定输出的文件，内容将输出到标准错误输出（`stderr`）上。

3.4.2　访问日志记录器

Scaffold 的 HTTP 服务内置一个访问日志记录器，符号为 `scaf::accesslog`。头文件 `Logger.hpp` 中以 `extern` 形式声明了它，包含后即可设置输出的位置或将其停用等。默认输出到标准错误输出。

下面是一个禁用访问日志的例子：

```
scaf::accesslog.disable();
```

下面是一个将访问日志保存到磁盘文件上的例子：

```
scaf::accesslog.setFile("/var/log/scaffold.log");
```

目前访问日志的风格[7]不能由用户自定义。访问日志模板如下：

```
[时间] "请求行" 客户端IP地址 响应状态码 "Referer字段" "User-Agent字段"
```

当某些字段不存在时，将不会被记录。下面是一个真实的访问日志片段：

```
[2017/08/29 00:08:22] "POST /api/export/json HTTP/1.1" 221.227.2.8 200
"https://recruit.mogician.cn:8080/admin" "Mozilla/5.0 (Macintosh;
Intel Mac OS X 10.12; rv:55.0) Gecko/20100101 Firefox/55.0"
```

[7]使用 Nginx 风格

第四章　插件

4.1　MySQL 数据库操作

该类[1]位于 scaf 命名空间，头文件为 <scaffold/Plugin/MySQL/MySQL.hpp>。

4.1.1　公共成员函数

- `MySQL(const string &host, int port, const string &user, const string &pass);`
  构造函数[2]。传入主机、端口、用户名及密码，在构造时将建立一个 MySQL 连接。

- `sql::Connection* getConnection(void);`
  返回这个 MySQL 连接的 `sql::Connection` 指针，以便用户深度使用 MySQL Connector/C++ 的功能。

- `MySQL* use(const string &t);`
  使用指定的数据库。如 `t = test` 则实际执行的 SQL 查询为 `USE test`。返回值为对象的 this 指针。

- `bool execute(const string &sql);`
  执行一条 SQL 查询，并返回是否成功。

- `imysqlstream executeQuery(const string &sql);`
  执行一条 SQL 查询，并返回封装为 `scaf::imysqlstream` 的结果。

- `omysqlstream prepare(const string &psql);`
  传入一条 Prepared Statement，由 `scaf::MySQL` 构造一个流式封装对象并返回。

- `imysqlstream execute(omysqlstream &os);`
  将已填充的 `scaf::omysqlstream` 流式封装对象传入，执行 Prepared Statement。结果封装为 `scaf::imysqlstream` 并返回。

[1]目前的实现仍然非常简单，没有 ORM 功能

[2]默认的构造函数、拷贝构造函数、移动构造函数被标记为 `delete`

4.1.2　MySQL 数据类型

由于 MySQL 中的类型种类繁多，且与程序设计语言的类型系统关联性不强，不便于完整地一一对应。头文件 `Plugin/MySQL/MySQL.hpp` 中定义了枚举 `scaf::MysqlType`，共 11 种类型。在处理过程中，它们与 C++ 的类型的对应情况如表。

表 4.1: MySQL 类型与 C++ 类型的对应关系

MySQL 类型	枚举标识符	C++ 类型
BIGINT	BigInt	std::string
BLOB	Blob	std::istream *
BOOLEAN	Bool	bool
DATETIME	DateTime	std::string
浮点数	Double	double
整数	Int	int32_t
整数	UInt	uint32_t
整数	Int64	int64_t
整数	UInt64	uint64_t
NULL	Null	sql::DataType::SQLNULL
字符串	String	std::string

在 Scaffold 中，对用户暴露的部分其实是使用 `std::nullptr_t` 代表 `NULL` 类型。

4.2　imysqlstream 流式封装

该类位于 `scaf` 命名空间，头文件为 `<scaffold/Plugin/MySQL/imstream.hpp>`。

4.2.1　数据成员解释

表 4.2: `scaf::imysqlstream` 数据成员

类型	标识符	意义
bool	tied	标记是否绑定到结果集
bool	copied	标记该对象是否发生拷贝或移动
unsigned int	index	当前的列索引
sql::ResultSet *	rs	MySQL Connector/C++ 提供的结果集指针

4.2.2 公共成员函数

- `void tie(sql::ResultSet* _rs);`
  把给定的数据集绑定到对象。
- `void untie(void);`
  将已绑定的数据集解绑。若未绑定，则无效果。
- `sql::ResultSet& ref(void);`
  返回结果集对象的引用，即 `*rs`。
- `sql::ResultSet* ptr(void);`
  返回结果集对象的指针，即 `rs`。
- `size_t rowsCount(void);`
  获取结果集中的记录数量。
- `bool startRow(void);`
  标记开始提取一行数据。若无法开始（无数据等情况）则返回 `false`。
- `void skip(void);`
  跳过一列。
- `void preventDeleting(void);`
  当对象析构时会 `delete` 绑定的结果集。调用这个方法将阻止这个动作。

4.2.3 运算符重载

共实现了 8 个形如 `scaf::imysqlstream& operator >> (scaf::imysqlstream &is, T &val)` 的重载。一般来说获取结果时较少会用到此类流式操作，原生的 SQL 连接器操作可能更加方便。建议直接使用 `rs` 或调用 `ref()`、`ptr()` 来获得结果集对象，使用原生的 `get` 系列操作。

表 4.3: 运算符 `>>` 重载的右操作数类型

`bool &`	`double &`	`std::string &`	`std::istream *&`
`int32_t &`	`uint32_t &`	`int64_t &`	`uint64_t &`

4.3 omysqlstream 流式封装

该类位于 `scaf` 命名空间，头文件为 `<scaffold/Plugin/MySQL/omstream.hpp>`。

4.3.1 数据成员解释

表 4.4: scaf::omysqlstream 数据成员

类型	标识符	意义
bool	tied	标记是否绑定到参数化语句
bool	copied	标记该对象是否发生拷贝或移动
unsigned int	index	当前的列索引
sql::PreparedStatement *	pstmt	MySQL Connector/C++ 提供的参数化语句指针

4.3.2 公共成员函数

- `void tie(sql::PreparedStatement* _rs);`
  把给定的参数化语句绑定到对象。
- `void untie(void);`
  将已绑定的参数化语句解绑。若未绑定，则无效果。
- `sql::ResultSet& ref(void);`
  返回参数化语句对象的引用，即 `*rs`。
- `sql::ResultSet* ptr(void);`
  返回参数化语句对象的指针，即 `rs`。
- `unsigned int getIndex(void);`
  获取当前填充到的列索引。
- `unsigned int increaseIndex(void);`
  将当前列索引增加 1。
- `void clear(void);`
  清空已填入的查询参数，重新使用该语句。
- `void preventDeleting(void);`
  当对象析构时会 `delete` 绑定的参数化语句。调用这个方法将阻止这个动作。
- `void set(const sql::SQLString &s, MysqlType t = MysqlType::String);`
  将一个 `sql::SQLString` 或 `std::string` 类型的对象填充到当前参数并预备填写下一个。参数 `t` 的可选值有 `MysqlType::String`、`MysqlType::BigInt` 或 `MysqlType::DateTime`。

- `void set(const char *s, MysqlType t = MysqlType::String);`
  同上，但针对 `const char *` 类型的值。
- `void set(std::istream &blob);`
  填充一个 BLOB 并预备填写下一个。
- `void set(std::istream *blob);`
  同上。
- `void set(bool boolean);`
  填充一个 BOOLEAN 并预备填写下一个。
- `void set(double dbl);`
  填充一个浮点数并预备填写下一个。
- `void set(int32_t i32);`
  填充一个整数并预备填写下一个。
- `void set(uint32_t ui32);`
  填充一个整数并预备填写下一个。
- `void set(int64_t i64);`
  填充一个整数并预备填写下一个。
- `void set(uint64_t ui64);`
  填充一个整数并预备填写下一个。
- `void setNull(void);`
  填充一个 NULL 并预备填写下一个。

4.3.3 运算符重载

每个 `omysqlstream::set` 都可以匹配到形如 `scaf::omysqlstream& operator << (scaf::omysqlstream &os, T value)` 的重载。

表 4.5: 运算符 << 重载的右操作数类型

bool	double	const char *	std::nullptr_t
int32_t	uint32_t	int64_t	uint64_t
std::string &	sql::SQLString &	std::istream *	std::istream &