26讲Easylogging++和spdlog：两个好用的日志库

你好，我是吴咏炜。

上一讲正文我介绍了两个可以在 C++ 中进行单元测试的库。今天，类似的，我介绍两个实用的日志库，分别是 Easylogging++ [1] 和 spdlog [2]。

Easylogging++

事实上，我本来想只介绍 Easylogging++ 的。但在检查其 GitHub 页面时，我发现了一个问题：它在 2019 年基本没有更新，且目前上报的问题也没有人处理。这是个潜在问题，除非你觉得这个库好到愿意自己动手修问题（话说回来，这个库还是不错的，我在这个项目贡献了 8 个被合并的 pull request）。不管怎样，原先说了要介绍这个库，所以我也还是介绍一下。

概述

Easylogging++ 一共只有两个文件，一个是头文件，一个是普通 C++ 源文件。事实上，它的一个较早版本只有一个文件。正如 Catch2 里一旦定义了 `CATCH_CONFIG_MAIN` 编译速度会大大减慢一样，把什么东西都放一起最终证明对编译速度还是相当不利的，因此，有人提交了一个补丁，把代码拆成了两个文件。使用 Easylogging++ 也只需要这两个文件——除此之外，就只有对标准和系统头文件的依赖了。

要使用 Easylogging++，推荐直接把这两个文件放到你的项目里。Easylogging++ 有很多的配置项会影响编译结果，我们先大致查看一下常用的可配置项：

- `ELPP_UNICODE`：启用 Unicode 支持，为在 Windows 上输出混合语言所必需
- `ELPP_THREAD_SAFE`：启用多线程支持
- `ELPP_DISABLE_LOGS`：全局禁用日志输出
- `ELPP_DEFAULT_LOG_FILE`：定义缺省日志文件名称
- `ELPP_NO_DEFAULT_LOG_FILE`：不使用缺省的日志输出文件
- `ELPP_UTC_DATETIME`：在日志里使用协调世界时而非本地时间

- `ELPP_FEATURE_PERFORMANCE_TRACKING`：开启性能跟踪功能
- `ELPP_FEATURE_CRASH_LOG`：启用 GCC 专有的崩溃日志功能
- `ELPP_SYSLOG`：允许使用系统日志（Unix 世界的 syslog）来记录日志
- `ELPP_STL_LOGGING`：允许在日志里输出常用的标准容器对象（`std::vector` 等）
- `ELPP_QT_LOGGING`：允许在日志里输出 Qt 的核心对象（`QVector` 等）
- `ELPP_BOOST_LOGGING`：允许在日志里输出某些 Boost 的容器（`boost::container::vector` 等）
- `ELPP_WXWIDGETS_LOGGING`：允许在日志里输出某些 wxWidgets 的模板对象（`wxVector` 等）

可以看到，Easylogging++ 的功能还是很丰富很全面的。

开始使用 Easylogging++

虽说 Easylogging++ 的功能非常多，但开始使用它毫不困难。我们从一个简单的例子开始看一下：

```
#include "easylogging++.h"
INITIALIZE_EASYLOGGINGPP

int main()
{
  LOG(INFO) << "My first info log";
}
```

编译链接的时候要把 easylogging++.cc 放进去。比如，使用 GCC 的话，命令行会像：

```
g++ -std=c++17 test.cpp easylogging++.cc
```

运行生成的可执行程序，你就可以看到结果输出到终端和 myeasylog.log 文件里，包含了日期、时间、级别、日志名称和日志信息，形如：

```
2020-01-25 20:47:50,990 INFO [default] My first info log
```

如果你对上面用到的宏感到好奇的话，`INITIALIZE_EASYLOGGINGPP` 展开后（可以用编译器的 `-E` 参数查看宏展开后的结果）是定义了 Easylogging++ 使用到的全局对象，而 `LOG(INFO)` 则是 Info 级别的日志记录器，同时传递了文件名、行号、函数名等日志需要的信息。

使用 Unicode

如果你在 Windows 上，那有一个复杂性就是是否使用“Unicode”的问题（[第 11 讲] 中讨论了）。就我们日志输出而言，启用 Unicode 支持的好处是：

- 可以使用宽字符来输出
- 日志文件的格式是 UTF-8，而不是传统的字符集，只能支持一种文字

要启用 Unicode 支持，你需要定义宏 `ELPP_UNICODE`，并确保程序中有对标准输出进行区域或格式设置（如 [第 11 讲] 中所述，需要进行设置才能输出含非 ASCII 字符的宽字符串）。下面的程序给出了一个简单的示例：

```
#ifdef _WIN32
#include <fcntl.h>
#include <io.h>
#else
#include <locale>
#endif
#include "easylogging++.h"
INITIALIZE_EASYLOGGINGPP

int main()
{
#ifdef _WIN32
  _setmode(_fileno(stdout),
           _O_WTEXT);
#else
  using namespace std;
  locale::global(locale(""));
  wcout.imbue(locale());
#endif

  LOG(INFO) << L"测试 test";
  LOG(INFO)
    << "Narrow ASCII always OK";
}
```

编译使用的命令行是：

```
cl /EHsc /DELPP_UNICODE test.cpp easylogging++.cc
```

改变输出文件名

Easylogging++ 的缺省输出日志名为 myeasylog.log，这在大部分情况下都是不适用的。我们可以直接在命令行上使用宏定义来修改（当然，稍大点的项目就应该放在项目的编译配置文件里了，如 Makefile）。比如，要把输出文件名改成 test.log，我们只需要在命令行上加入下面的选项就可以：

```
-DELPP_DEFAULT_LOG_FILE=\"test.log\"
```

使用配置文件设置日志选项

不过，对于日志文件名称这样的设置，使用配置文件是一个更好的办法。Easylogging++ 库自己支持配置文件，我也推荐使用一个专门的配置文件，并让 Easylogging++ 自己来加载配置文件。我自己使用的配置文件是这个样子的：

```
* GLOBAL:
   FORMAT               =  "%datetime{%Y-%M-%d %H:%m:%s.%g} %levshort %msg"
   FILENAME             =  "test.log"
   ENABLED              =  true
   TO_FILE              =  true     ## 输出到文件
   TO_STANDARD_OUTPUT   =  true     ## 输出到标准输出
   SUBSECOND_PRECISION  =  6        ## 秒后面保留 6 位
   MAX_LOG_FILE_SIZE    =  2097152  ## 最大日志文件大小设为 2MB
   LOG_FLUSH_THRESHOLD  =  10       ## 写 10 条日志刷新一次缓存
* DEBUG:
   FORMAT               = "%datetime{%Y-%M-%d %H:%m:%s.%g} %levshort [%fbase:%line] %msg"
   TO_FILE              =  true
   TO_STANDARD_OUTPUT   =  false    ## 调试日志不输出到标准输出
```

这个配置文件里有两节：第一节是全局（global）配置，配置了适用于所有级别的日志选项；第二节是专门用于调试（debug）级别的配置（你当然也可以自己配置 fatal、error、warning 等其他级别）。

假设这个配置文件的名字是 log.conf，我们在代码中可以这样使用：

```
#include "easylogging++.h"
INITIALIZE_EASYLOGGINGPP

int main()
{
  el::Configurations conf{
    "log.conf"};
  el::Loggers::
    reconfigureAllLoggers(conf);
  LOG(DEBUG) << "A debug message";
  LOG(INFO) << "An info message";
}
```

注意编译命令行上应当加上 `-DELPP_NO_DEFAULT_LOG_FILE`，否则 Easylogging++ 仍然会生成缺省的日志文件。

运行生成的可执行程序，我们会在终端上看到一条信息，但在日志文件里则可以看到两条信息。如下所示：

```
2020-01-26 12:54:58.986739 D [test.cpp:11] A debug message
2020-01-26 12:54:58.987444 I An info message
```

我们也可以明确看到我们在配置文件中定义的日志格式生效了，包括：

- 日期时间的格式使用“.”分隔秒的整数和小数部分，并且小数部分使用 6 位
- 日志级别使用单个大写字母
- 对于普通的日志，后面直接跟日志的信息；对于调试日志，则会输出文件名和行号

我们现在只需要修改配置文件，就能调整日志格式、决定输出和不输出哪些日志了。此外，我也推荐在编译时定义宏 ELPP_DEBUG_ASSERT_FAILURE，这样能在找不到配置文件时直接终止程序，而不是继续往下执行、在终端上以缺省的方式输出日志了。

性能跟踪

Easylogging++ 可以用来在日志中记录程序执行的性能数据。这个功能还是很方便的。下面的代码展示了用于性能跟踪的三个宏的用法：

```
#include <chrono>
#include <thread>
#include "easylogging++.h"
INITIALIZE_EASYLOGGINGPP

void foo()
{
  TIMED_FUNC(timer);
  LOG(WARNING) << "A warning message";
}

void bar()
{
  using namespace std::literals;
  TIMED_SCOPE(timer1, "void bar()");
  foo();
  foo();
  TIMED_BLOCK(timer2, "a block") {
    foo();
    std::this_thread::sleep_for(100us);
  }
}

int main()
{
  el::Configurations conf{
    "log.conf"};
  el::Loggers::
    reconfigureAllLoggers(conf);
  bar();
}
```

简单说明一下：

- TIMED_FUNC 接受一个参数，是用于性能跟踪的对象的名字。它能自动产生函数的名称。示例中的 TIMED_FUNC 和 TIMED_SCOPE 的作用是完全相同的。

- `TIMED_SCOPE` 接受两个参数，分别是用于性能跟踪的对象的名字，以及用于记录的名字。如果你不喜欢 `TIMED_FUNC` 生成的函数名字，可以用 `TIMED_SCOPE` 来代替。
- `TIMED_BLOCK` 用于对下面的代码块进行性能跟踪，参数形式和 `TIMED_SCOPE` 相同。

在编译含有上面三个宏的代码时，需要定义宏 `ELPP_FEATURE_PERFORMANCE_TRACKING`。你一般也应该定义 `ELPP_PERFORMANCE_MICROSECONDS`，来获取微秒级的精度。下面是定义了上面两个宏编译的程序的某次执行的结果：

```
2020-01-26 15:00:11.99736 W A warning message
2020-01-26 15:00:11.99748 I Executed [void foo()] in [110 us]
2020-01-26 15:00:11.99749 W A warning message
2020-01-26 15:00:11.99750 I Executed [void foo()] in [5 us]
2020-01-26 15:00:11.99750 W A warning message
2020-01-26 15:00:11.99751 I Executed [void foo()] in [4 us]
2020-01-26 15:00:11.99774 I Executed [a block] in [232 us]
2020-01-26 15:00:11.99776 I Executed [void bar()] in [398 us]
```

不过需要注意，由于 Easylogging++ 本身有一定开销，且开销有一定的不确定性，这种方式只适合颗粒度要求比较粗的性能跟踪。

性能跟踪产生的日志级别固定为 Info。性能跟踪本身可以在配置文件里的 GLOBAL 节下用 `PERFORMANCE_TRACKING = false` 来关闭。当然，关闭所有 Info 级别的输出也能达到关闭性能跟踪的效果。

记录崩溃日志

在 GCC 和 Clang 下，通过定义宏 `ELPP_FEATURE_CRASH_LOG` 我们可以启用崩溃日志。此时，当程序崩溃时，Easylogging++ 会自动在日志中记录程序的调用栈信息。通过记录下的信息，再利用 `addr2line` 这样的工具，我们就能知道是程序的哪一行引发了崩溃。下面的代码可以演示这一行为：

```
#include "easylogging++.h"
INITIALIZE_EASYLOGGINGPP

void boom()
{
  char* ptr = nullptr;
  *ptr = '\0';
}

int main()
{
  el::Configurations conf{
    "log.conf"};
  el::Loggers::
    reconfigureAllLoggers(conf);
  boom();
}
```

你可以自己尝试编译运行一下，就会在终端和日志文件中看到崩溃的信息了。

使用 macOS 的需要特别注意一下：由于缺省方式产生的可执行文件是位置独立的，系统每次加载程序会在不同的地址，导致无法通过地址定位到程序行。在编译命令行尾部加上 `-Wl,-no_pie` 可以解决这个问题。

其他

Easylogging++ 还有很多其他功能，我就不再一一讲解了。有些你简单试一下就可以用起来的。对于 `ELPP_STL_LOGGING`，你也可以在包含 easylogging++.h 之前包含我的 output_container.h，可以达到类似的效果。

此外，Easylogging++ 的 samples 目录下有不少例子，可以用作参考。比如常见的日志文件切换功能，在 Easylogging++ 里实现是需要稍微写一点代码的：Easylogging++ 会在文件满的时候调用你之前注册的回调函数，而你需要在回调函数里对老的日志文件进行重命名、备份之类的工作，samples/STL/roll-out.cpp 则提供了最简单的实现参考。

注意我使用的都是全局的日志记录器，但 Easylogging++ 允许你使用多个不同的日志记录器，用于（比如）不同的模块或功能。你如果需要这样的功能的话，也请你自行查阅文档了。

spdlog

跟 Easylogging++ 比起来，spdlog 要新得多了：前者是 2012 年开始的项目，而后者是 2014 年开始的。我在 2016 年末开始在项目中使用 Easylogging++ 时，Easylogging++ 的版本是 9.85 左右，而 spdlog 大概是 0.11，成熟度和热度都不那么高。

整体上，spdlog 也确实感觉要新很多。项目自己提到的功能点是：

- 非常快（性能是其主要目标）
- 只需要头文件即可使用
- 没有其他依赖
- 跨平台
- 有单线程和多线程的日志记录器
- 日志文件旋转切换
- 每日日志文件
- 终端日志输出
- 可选异步日志
- 多个日志级别
- 通过用户自定义式样来定制输出格式

开始使用 spdlog

跟 Easylogging++ 的例子相对应，我们以最简单的日志输出开头：

```
#include "spdlog/spdlog.h"


int main()
{
  spdlog::info("My first info log");
}
```

代码里看不到的是，输出结果中的“info”字样是彩色的，方便快速识别日志的级别。这个功能在 Windows、Linux 和 macOS 上都能正常工作，对用户还是相当友好的。不过，和 Easylogging++ 缺省就会输出到文件中不同，spdlog 缺省只是输出到终

端而已。

你也许从代码中已经注意到，spdlog 不是使用 IO 流风格的输出了。它采用跟 Python 里的`str.format`一样的方式，使用大括号——可选使用序号和格式化要求——来对参数进行格式化。下面是一个很简单的例子：

```
  spdlog::warn(
    "Message with arg {}", 42);
  spdlog::error(
    "{0:d}, {0:x}, {0:o}, {0:b}",
    42);
```

输出会像下面这样：

```
[2020-01-26 17:20:08.355] [warning] Message with arg 42
[2020-01-26 17:20:08.355] [error] 42, 2a, 52, 101010
```

事实上，这就是 C++20 的`format`的风格了——spdlog 就是使用了一个`format`的库实现 fmt [3]。

设置输出文件

在 spdlog 里，要输出文件得打开专门的文件日志记录器，下面的例子展示了最简单的用法：

```
#include "spdlog/spdlog.h"
#include "spdlog/sinks/basic_file_sink.h"

int main()
{
  auto file_logger =
    spdlog::basic_logger_mt(
      "basic_logger",
      "test.log");
  spdlog::set_default_logger(
    file_logger);
  spdlog::info("Into file: {1} {0}",
               "world", "hello");
}
```

执行之后，终端上没有任何输出，但 test.log 文件里就会增加如下的内容：

```
[2020-01-26 17:47:37.864] [basic_logger] [info] Into file: hello world
```

估计你立即会想问，那我想同时输出到终端和文件，该怎么办呢?

答案是你可以设立一个日志记录器，让它有两个（或更多个）日志槽（sink）即可。示例代码如下：

```
#include <memory>
#include "spdlog/spdlog.h"
```

```
#include "spdlog/sinks/basic_file_sink.h"
#include "spdlog/sinks/stdout_color_sinks.h"

using namespace std;
using namespace spdlog::sinks;

void set_multi_sink()
{
  auto console_sink = make_shared<
    stdout_color_sink_mt>();
  console_sink->set_level(
    spdlog::level::warn);
  console_sink->set_pattern(
    "%H:%M:%S.%e %^%L%$ %v");

  auto file_sink =
    make_shared<basic_file_sink_mt>(
      "test.log");
  file_sink->set_level(
    spdlog::level::trace);
  file_sink->set_pattern(
    "%Y-%m-%d %H:%M:%S.%f %L %v");

  auto logger =
    shared_ptr<spdlog::logger>(
      new spdlog::logger(
        "multi_sink",
        {console_sink, file_sink}));
  logger->set_level(
    spdlog::level::debug);
  spdlog::set_default_logger(
    logger);
}

int main()
{
  set_multi_sink();
  spdlog::warn(
    "this should appear in both "
    "console and file");
  spdlog::info(
    "this message should not "
    "appear in the console, only "
    "in the file");
```

```
}
```

大致说明一下：

- `console_sink` 是一个指向 `stdout_color_sink_mt` 的智能指针，我们设定让它只显示警告级别及以上的日志信息，并把输出式样调整成带毫秒的时间、有颜色的短级别以及信息本身。
- `file_sink` 是一个指向 `basic_file_sink_mt` 的智能指针，我们设定让它显示跟踪级别及以上（也就是所有级别了）的日志信息，并把输出式样调整成带微秒的日期时间、短级别以及信息本身。
- 然后我们创建了日志记录器，让它具有上面的两个日志槽。注意这儿的两个细节：1. 这儿的接口普遍使用 `shared_ptr`；2. 由于 `make_shared` 在处理 `initializer_list` 上的缺陷，对 `spdlog::logger` 的构造只能直接调用 `shared_ptr` 的构造函数，而不能使用 `make_shared`，否则编译会出错。
- 最后我们调用了 `spdlog::set_default_logger` 把缺省的日志记录器设置成刚创建的对象。这样，之后的日志缺省就会记录到这个新的日志记录器了（我们当然也可以手工调用这个日志记录器的 `critical`、`error`、`warn` 等日志记录方法）。

在某次运行之后，我的终端上出现了：

```
20:44:45.086 W this should appear in both console and file
```

而 test.log 文件中则增加了：

```
2020-01-26 20:44:45.086524 W this should appear in both console and file
2020-01-26 20:44:45.087174 I this message should not appear in the console, only in the file
```

跟 Easylogging++ 相比，我们现在看到了 spdlog 也有复杂的一面。两者在输出式样的灵活性上也有不同的选择：Easylogging++ 对不同级别的日志可采用不同的式样，而 spdlog 对不同的日志槽可采用不同的式样。

日志文件切换

在 Easylogging++ 里实现日志文件切换是需要写代码的，而且完善的多文件切换代码需要写上几十行代码才能实现。这项工作在 spdlog 则是超级简单的，因为 spdlog 直接提供了一个实现该功能的日志槽。把上面的例子改造成带日志文件切换我们只需要修改两处：

```
#include "spdlog/sinks/rotating_file_sink.h"
// 替换 basic_file_sink.h
…
  auto file_sink = make_shared<
    rotating_file_sink_mt>(
    "test.log", 1048576 * 5, 3);
  // 替换 basic_file_sink_mt，文件大
  // 小为 5MB，一共保留 3 个日志文件
```

这就非常简单好用了。

适配用户定义的流输出

虽然 spdlog 缺省不支持容器的输出，但是，它是可以和用户提供的流 << 运算符协同工作的。如果我们要输出普通容器的

话，我们只需要在代码开头加入：

```
#include "output_container.h"
#include "spdlog/fmt/ostr.h"
```

前一行包含了我们用于容器输出的代码，后一行包含了 spdlog 使用 ostream 来输出对象的能力。注意此处包含的顺序是重要的：spdlog 必须能看到用户的 << 的定义。在有了这两行之后，我们就可以像下面这样写代码了：

```
vector<int> v;
// …
spdlog::info(
  "Content of vector: {}", v);
```

只用头文件吗?

使用 spdlog 可以使用只用头文件的方式，也可以使用预编译的方式。只用头文件的编译速度较慢：我的机器上使用预编译方式构建第一个例子需要一秒多，而只用头文件的方式需要五秒多（Clang 的情况；GCC 耗时要更长）。因此正式使用的话，我还是推荐你使用预编译、安装的方式。

在安装了库后，编译时需额外定义一个宏，在命令行上要添加库名。以 GCC 为例，命令行会像下面这个样子：

```
g++ -std=c++17 -DSPDLOG_COMPILED_LIB test.cpp -lspdlog
```

其他

刚才介绍的还只是 spdlog 的部分功能。你如果对使用这个库感兴趣的话，应该查阅文档来获得进一步的信息。我这儿觉得下面这些功能点值得提一下：

- 可以使用多个不同的日志记录器，用于不同的模块或功能。
- 可以使用异步日志，减少记日志时阻塞的可能性。
- 通过 `spdlog::to_hex` 可以方便地在日志里输出二进制信息。
- 可用的日志槽还有 syslog、systemd、Android、Windows 调试输出等；扩展新的日志槽较为容易。

内容小结

今天我们介绍了两个不同的日志库，Easylogging++ 和 spdlog。它们在功能和实现方式上有很大的不同，建议你根据自己的实际需要来进行选择。

我目前对新项目的推荐是优先选择 spdlog：仅在你需要某个 Easylogging++ 提供、而 spdlog 不提供的功能时才选择 Easylogging++。

当然，C++ 的日志库远远不止这两个：我挑选的是我觉得比较好的和有实际使用经验的。其他可选择的日志库至少还有 Boost.Log [4]、g3log [5]、NanoLog [6] 等（Log for C++ 接口有着 Java 式的啰嗦，且感觉有点“年久失修”，我明确不推荐）。在严肃的项目里，选择哪个日志库是值得认真比较和评估一下的。

课后思考

请对比一下 Easylogging++ 和 spdlog，考虑以下两个问题：

1. Easylogging++ 更多地使用了编译时的行为定制，而 spdlog 主要通过面向对象的方式在运行时修改日志的行为。你觉得

哪种更好？为什么？

2. Easylogging++ 使用了 IO 流的方式，而 spdlog 使用了 `std::format` 的方式。你更喜欢哪种？为什么？

参考资料

[1] Amrayn Web Services, easyloggingpp. https://github.com/amrayn/easyloggingpp

[2] Gabi Melman, spdlog. https://github.com/gabime/spdlog

[3] Victor Zverovich, fmt. https://github.com/fmtlib/fmt

[4] Andrey Semashev, Boost.Log v2. https://www.boost.org/doc/libs/release/libs/log/doc/html/index.html

[5] Kjell Hedström, g3log. https://github.com/KjellKod/g3log

[6] Stanford University, NanoLog. https://github.com/PlatformLab/NanoLog

精选留言

hello world

之前都在用glog，觉得有必要换成spdlog

2020-01-31 09:35

作者回复

嗯，spdlog看起来是一个很现代、很好用的库。如果不要求Windows的Unicode支持，是挺好（不过，查了下发现glog也不支持Unicode，所以没有变差）。

另外一个选择是 g3log，接口差不多，性能提高，支持异步。

2020-01-31 13:47

吴先生

glog, gflags, gtest 我们的工程里都用这些，可不可以比较一下

2020-01-31 05:11

作者回复

你可以跟我的描述比较啊……

后面提到的 g3log 就是基于 glog 开发的，但功能、性能都有改进。如果你目前用的觉得够用，是没必要换的。主要是看其他家有没有提供你想要、目前又没有的功能或性能。

2020-01-31 09:26