

19 | 其他重要标准库特性实战：利用日历应用熟悉新特性

2023-03-06 卢誉声 来自北京

《现代C++20实战高手课》

课程介绍 >



讲述：卢誉声

时长 13:24 大小 12.24M



你好，我是卢誉声。

我们想要提升 C++ 的编程效率，就需要对重要标准库的变更保持关注。在第 18 讲已经涵盖了绝大多数 C++20 带来的重要库变更。不过，我当时有意忽略了其中一个，就是我们今天的主角——C++20 Calendar、Timzone。

它们是对现有 chrono 库的重要补充。Calendar 提供了日历的表示与计算工具。而 Timezone 提供了标准的时区定义，可以构建包含时区信息的 zoned_time。

今天，我会围绕 C++20 Calendar、Timzone 带你进行编程实战，并结合上一讲涵盖的特性：jthread、source location、sync stream 和 u8string，实现一个使用新标准实现的日程序。

哦，对了，我们还会在这一讲中使用 C++ 20 Formatting 库，帮你进一步加深对这个特性的理解。好，话不多说，就让我们从模块设计开始今天的内容（课程配套代码可以从 [这里](#)获

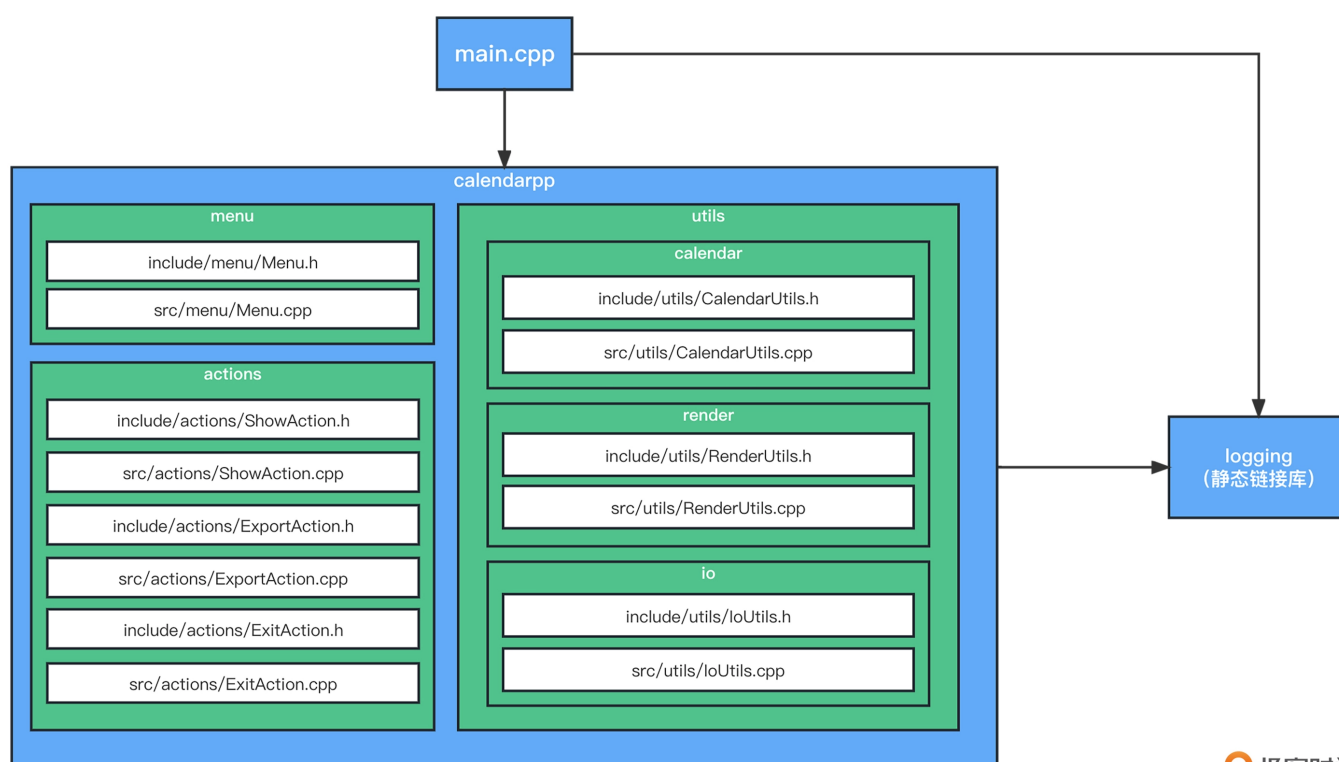
取）。

模块设计

我们准备构建的命令行日历应用，具备以下特性。

- 使用 C++20 chrono：支持显示本月日历，显示日期和星期信息。
- 使用 u8string：支持导出本年的全年日历到文本文件，编码为 UTF-8。

我们依然采用传统 C++ 模块结构设计，整体模块设计如下图所示。



该工程包含两个子项目，一个是可执行文件 `calendarpp`，另一个是静态链接库 `logging`。从图中可以看到，`calendarpp` 的入口是 `main.cpp`，其他实现都在 `calendarpp` 模块下，包括后面这几个模块。

- `menu`：主菜单实现，包括菜单定义与菜单交互。
- `actions`：各个菜单项的具体实现，完成具体的功能。
- `utils`：具体的底层实现模块，包括日历计算模块 `calendar`、文本渲染模块 `render` 和输入输出模块 `io`。

我们沿用了 [第 15 讲](#) 中的实现，即日志框架来构建并包装成了 **logging** 库，并利用第 18 讲中讲过的部分特性进行了改造。沿着这条路线，我们先从改造日志框架开始看。

改造日志框架

日志框架的所有代码文件都在 `projects/logging` 下，你可以结合代码来理解如何集成日志框架，并用它来记录系统运行日志。我们来看看，使用 C++20 的特性，可以在原有的框架基础上做出哪些改造。

之前的 **Handler** 在多线程场景下使用时，因为没有采用线程同步方案，导致输出时可能会产生错乱的问题，因此我们改造了 **Handler**。

比如 **DefaultHandler** 的实现改造是后面这样。

 复制代码

```
1  #pragma once
2
3  #include "logging/Handler.h"
4  #include <syncstream>
5
6  namespace logging::handlers {
7      // 默认日志处理器
8      template <Level HandlerLevel = Level::Warning>
9          // 继承BaseHandler
10         class DefaultHandler : public BaseHandler<HandlerLevel> {
11         public:
12             // 构造函数，需要指定格式化器，默认格式化器为defaultFormatter
13             DefaultHandler(Formatter formatter = defaultFormatter) : BaseHandler<Ha
14             // 禁止拷贝构造函数
15             DefaultHandler(const DefaultHandler&) = delete;
16             // 定义移动构造函数
17             DefaultHandler(const DefaultHandler&& rhs) noexcept : BaseHandler<Handl
18
19             // emit用于提交日志记录
20             // emitLevel > HandlerLevel的日志会被丢弃
21             template <Level emitLevel>
22                 requires (emitLevel > HandlerLevel)
23             void emit(const Record& record) {
24             }
25
26             // emitLevel <= HandlerLevel的日志会被输出到标准输出流中
27             template <Level emitLevel>
28                 requires (emitLevel <= HandlerLevel)
29             void emit(const Record& record) {
30                 // 调用format将日志记录对象格式化成文本字符串
```

```

31     std::osyncstream(std::cout) << this->format(record) << std::endl;
32     }
33     };
34 }

```

代码中的改动在第 31 行，通过 **std::osyncstream** 包装了 **std::cout**，然后通过包装后的输出流进行输出，这样就能完成输出的线程同步控制（详细讲解，你可以回顾上一讲“**sync stream**”这部分）。

日志框架的第二个改动在于，它通过 **source location** 记录了输出日志的源代码位置信息。实现在 **Record.h** 下，后面展示的是重点代码。

 复制代码

```

1  class Record {
2  public:
3      // Logger名称
4      std::string name;
5      // 日志等级
6      Level level;
7      // 日志时间
8      TimePoint time;
9      // 日志消息
10     std::string message;
11     std::source_location sourceLocation;
12
13     // getLevelName: 获取日志等级文本
14     const std::string& getLevelName() const {
15         // 调用toLevelName获取日志等级文本
16         return toLevelName(level);
17     }
18 };

```

通过代码可以看到，**Record** 定义增加了 **sourceLocation** 用于记录源代码位置。

接着，我们还要修改 **Logger** 定义，增加记录源代码位置信息的功能。具体实现在 **Logger.h** 下，我们将 **log** 的定义改为以下形式。

 复制代码

```

1  template <Level level>
2      requires (level > loggerLevel)
3  Logger& log(const std::string& message, std::source_location sourceLocation = s
4      return *this;

```

```

5  }
6
7  // 通过requires约束提交等级为日志记录器设定等级及以上的日志
8  template <Level level>
9      requires (level <= loggerLevel)
10  Logger & log(const std::string& message, std::source_location sourceLocation =
11      // 构造Record对象
12      Record record{
13          .name = _name,
14          .level = level,
15          .time = std::chrono::system_clock::now(),
16          .message = message,
17          // 记录源代码位置
18          .sourceLocation = sourceLocation
19      });
20
21  // 调用handleLog实际处理日志输出
22  handleLog<level, HandlerCount - 1>(record);
23
24  return *this;
25  }

```

可以看到，我增加了 `source_location` 定义，并通过默认参数自动记录调用者的所在位置，调用者也可以自己指定需要记录的 `source_location`。

我们同时修改了几个级别的相关接口，比如 `debug` 成员函数改造。

 复制代码

```

1  // 提交调试信息（log的包装）
2  Logger& debug(const std::string& message, std::source_location sourceLocation =
3      return log<Level::Debug>(message, sourceLocation);
4  }

```

其他的几个成员函数都和 `debug` 一样，这里就不展示出来了。

最后，我还修改了 `format` 函数输出 `sourceLocation` 中的信息。具体实现，我们以 `ModernFormatter.cpp` 的修改为例来看看。

 复制代码

```

1  #include "logging/formatters/ModernFormatter.h"
2  #include "logging/Record.h"
3
4  namespace logging::formatters::modern {

```

```

5 // formatRecord: 将Record对象格式化为字符串
6 std::string formatRecord(const Record& record) {
7     const auto& sourceLocation = record.sourceLocation;
8
9     try {
10         return std::format(
11             "{0:<16}| [{1}] {2:%Y-%m-%d}T{2:%H:%M:%OS}Z - <{3}:{4} [{5}]> -
12             record.name,
13             record.getLevelName(),
14             record.time,
15             sourceLocation.file_name(),
16             sourceLocation.line(),
17             sourceLocation.function_name(),
18             record.message
19         );
20     }
21     catch (std::exception& e) {
22         std::cerr << "Error in format: " << e.what() << std::endl;
23
24         return "";
25     }
26 }
27 }

```

代码中 `format` 的部分加入了 `sourceLocation` 信息，其他部分相较于之前则没有变化。

主程序与菜单

在了解了日志框架的改造后，我们继续看如何通过 `C++20 Formatting` 库实现主程序和菜单。

主程序定义非常简单，在 `main.cpp` 中，代码是后面这样。

 复制代码

```

1 #include "menu/Menu.h"
2 #include <iostream>
3 #include <format>
4
5 int main() {
6     while (true) {
7         // 展示菜单
8         calendarpp::menu::showMenu();
9         // 读取并执行菜单项
10        calendarpp::menu::readAction();
11    }
12
13    return 0;
14 }

```

代码整体是一个 **while** 循环，首先展示菜单，然后读取用户输入并执行菜单项。所以只有用户选择退出程序时，整个程序才会退出。

菜单实现在 `menu/Menu.cpp` 中，代码是后面这样。

 复制代码

```
1 #include "menu/Menu.h"
2 #include "actions/ShowAction.h"
3 #include "actions/ExportAction.h"
4 #include "actions/ExitAction.h"
5 #include "utils/RenderUtils.h"
6
7 #include <iostream>
8 #include <string>
9 #include <vector>
10 #include <cstdlib>
11 #include <format>
12 #include <algorithm>
13
14 namespace calendarpp::menu {
15     // 菜单项类型
16     struct MenuItem {
17         std::string title;
18         Action action;
19     };
20
21     // 所有菜单
22     static const std::vector<MenuItem> MenuItems = {
23         std::vector<MenuItem> {
24             {
25                 .title = "展示本月日历",
26                 .action = actions::showCurrentMonth
27             },
28             {
29                 .title = "导出本年日历",
30                 .action = actions::exportCurrentYear
31             },
32             {
33                 .title = "退出程序",
34                 .action = actions::exitApp
35             }
36         }
37     };
38
39     // 菜单选项范围
40     static const int32_t MinActionNumber = 1;
41     static const int32_t MaxActionNumber = MenuItems.size();
```

```

42
43 // 展示菜单
44 void showMenu() {
45     // 输出系统标题
46     std::cout << std::format("\n{:=^80}\n", " Calendar++ v1.0 ") << std::endl;
47     // 输出当前时间与本地时区信息
48     std::cout << utils::renderNow() << std::endl;
49
50     // 输出所有菜单
51     std::cout << std::format("{:=^80}\n", " MENU ");
52     std::cout << std::format("*{: ^78}*\n", "");
53     std::int32_t menuIndex = 0;
54     for (const auto& menuItem : MenuItems) {
55         menuIndex += 1;
56
57         // 输出菜单序号与菜单名称
58         std::string menuLine = std::format("{} {} ", menuIndex, menuItem.title);
59         std::cout << std::format("* {: <76} *", menuLine) << std::endl;
60     }
61     std::cout << std::format("*{: ^78}*\n", "");
62     std::cout << std::format("{:=^80}\n", "");
63
64     // 提示用户输入菜单编号
65     std::cout << std::format("\n请输入菜单编号({}-{}):", 1, menuIndex);
66 }
67
68 // 读取用户输入并执行动作
69 void readAction() {
70     std::string actionNumberString;
71     // 读取用户输入
72     std::getline(std::cin, actionNumberString);
73
74     try {
75         // 解析用户输入
76         int32_t actionNumber = std::stoi(actionNumberString);
77         if (actionNumber < MinActionNumber || actionNumber > MaxActionNumber)
78             std::cerr << std::format("菜单编号超出范围({}-{})\n", MinActionNumber, MaxActionNumber);
79     }
80
81     // 执行相应菜单项的action
82     int32_t actionIndex = std::max(actionNumber - 1, 0);
83     const auto& action = MenuItems[actionIndex].action;
84     action();
85
86     return;
87 }
88 catch (const std::exception& e) {
89     std::cerr << e.what() << std::endl;
90 }
91
92 std::cerr << "无法识别用户输入" << std::endl;
93 }

```


这段代码中值得注意的部分是 **showMenu 函数**，它用于展示菜单。该函数输出标题后，会遍历所有的 **MenuItems** 并输出菜单内容。发现了么？这段代码通过 **format** 完成了大量的文本格式化工作。

最后我们还定义了 **readAction** 函数，用于读取用户输入的菜单编号并执行对应的菜单动作。这里使用了 **C++11** 引入的 **std::stoi** 函数将字符串转换成对应的整型数字。

菜单的展示效果是后面这样。

```
===== calendar++ v1.0 =====
2023-02-13 11:34:22                                     Asia/Shanghai GMT+8
***** MENU *****
*
* (1) 展示本月日历
* (2) 导出本年日历
* (3) 退出程序
*
*****
```

需要注意的是，本项目代码采用了 **UTF-8** 编码，因此如果你在 **Windows** 下执行，需要在支持 **UTF-8** 编码输出的控制台下使用（比如 **MinGW** 的 **bash**）。

展示日历

接下来，我们看一下该如何展示本月日历。在 **actions** 中定义了菜单项的函数实现，代码实现在 **src/actions/ShowAction.cpp** 中。

 复制代码

```
1 #include "actions/ShowAction.h"
2 #include "utils/RenderUtils.h"
3
4 #include <chrono>
5 #include <iostream>
6 #include <format>
7
8 namespace chrono = std::chrono;
9
10 namespace calendarpp::actions {
11     void showCurrentMonth() {
12         // 获取当前时间
13         chrono::time_point now{ chrono::system_clock::now() };
```

```

14 // 将时间转换为year_month_day
15 chrono::year_month_day ymd{ chrono::floor<chrono::days>(now) };
16 // 获取当前年月（类型为year_month）
17 chrono::year_month currentYearMonth = ymd.year() / ymd.month();
18
19 // 调用渲染模块渲染当月日历
20 std::cout << std::endl;
21 std::cout << utils::renderMonth(currentYearMonth);
22 }
23 }

```

代码中定义了 `showCurrentMonth` 函数，该函数首先调用了 `chrono` 的 `Calendar` 获取当日所在的日期和月份。

`Calendar` 是 C++20 引入到 `chrono` 中新的库特性，提供了标准的格里高利历（Gregorian calendar）实现。

这里简单介绍一下 `year_month_day`，用于描述一个完整日期（年月日），该类型支持多种构造方法，这里使用 `chrono::system_clock::now()` 获取系统本地时间，然后采用 `chrono::floor<chrono::days>` 将时间转换为以日为单位的时间，最后调用 `year_month_day` 得到当日的日期。

`year_month_day` 支持通过 `year` 和 `month` 成员函数获取当日的年份与月份，代码 17 行就调用了 `year / month` 这种形式构造了一个 `year_month` 对象，表示某年的某个月，这里最后得到的就是本日所在的当月。`/` 是 `Calendar` 的一个操作符重载，是创建日期类型对象的一个语法糖。

代码最后调用 `renderMonth` 渲染了当月日历并将其输出到控制台，该函数定义在 `src/Utils/RenderUtils.cpp` 中，属于渲染模块，我们分析一下相关代码。

首先定义了 `Weekdays` 常量，包含了周一到周日的所有星期几的定义。

 复制代码

```

1 // 定义一周七天的weekday常量
2 static std::vector<chrono::weekday> Weekdays = {
3     chrono::Monday,
4     chrono::Tuesday,
5     chrono::Wednesday,
6     chrono::Thursday,
7     chrono::Friday,

```

```

8     chrono::Saturday,
9     chrono::Sunday,
10 };

```

该数组的元素类型为 `chrono::weekday`，这是 `chrono` 的标准类型，用来表示周几，其中 **Monday** 到 **Sunday** 都是 `chrono` 定义的常量，表示周一到周日。

 复制代码

```

1 // 渲染某个月份的日历，返回string
2 std::string renderMonth(chrono::year_month yearMonth) {
3     std::ostringstream os;
4
5     // 获取当月的所有周
6     auto monthWeeks = utils::buildMonthWeeks(yearMonth);
7
8     // 获取当月第一天
9     const auto firstDay = yearMonth / 1d;
10    // 获取当月最后一天
11    const auto lastDay = chrono::year_month_day(yearMonth / chrono::last);
12
13    // 输出格式化的标题（年份与月份）
14    std::string titleLine = std::format("** {%Y-%m} **", yearMonth);
15    os << std::format("{:^35}", titleLine) << std::endl;
16    os << std::format("{:->35}", "") << std::endl;
17
18    // 输出日历表头（从周一到周日）
19    std::vector<std::string> headerLineParts;
20    for (const auto& weekday : Weekdays) {
21        headerLineParts.push_back(std::format(ZhCNLocale, "{:L%a}", weekday));
22    }
23    // 利用renderWeekLine生成格式化的表头（控制7个元素的位置与宽度）
24    std::string headerLine = renderWeekLine(headerLineParts);
25    os << headerLine << std::endl;
26    os << std::format("{:->35}", "") << std::endl;
27
28    // 遍历monthWeeks，调用renderWeek生成日历中的每一行
29    for (const auto& currentWeek : monthWeeks) {
30        std::string weekLine = renderWeek(currentWeek, yearMonth);
31        os << weekLine << std::endl;
32    }
33
34    // 返回渲染的字符串
35    return os.str();
36 }
37
38 // 渲染日历中的某一周
39 std::string renderWeek(const std::vector<chrono::year_month_day> week, chrono::
40    // 获取当月第一天
41    const auto firstDay = yearMonth / 1d;

```

```

42 // 获取当月最后一天
43 const auto lastDay = chrono::year_month_day(yearMonth / chrono::last);
44
45 // 生成本周的所有日期
46 std::vector<std::string> weekLine;
47 for (const auto& currentDay : week) {
48     std::string inCurrentMonthFlag = currentDay >= firstDay && currentDay <
49
50     weekLine.push_back(std::format("{}{:>2}", inCurrentMonthFlag, currentDa
51 }
52
53 // 利用renderWeekLine生成格式化的本周日期
54 return renderWeekLine(weekLine);
55 }
56
57 // 生成某一周的格式化输出
58 std::string renderWeekLine(const std::vector<std::string>& weekLine) {
59     std::string renderResult;
60     for (const auto& weekLineItem : weekLine) {
61         // 所有内容按照宽度为4右对齐
62         renderResult.append(std::format("{: >4} ", weekLineItem));
63     }
64
65     return renderResult;
66 }

```

代码中的注释已经比较详细了，所以后面我们只讨论一些重点实现。

代码第 5 行，作用是获取当月的所有周的数组，`utils::buildMonthWeeks` 属于日历计算模块，我们后面详细解释其实现。

在代码第 8 行，用 `yearMonth / 1d` 生成当月的第一天，返回的类型就是 `year_month_day`。其中 `1d` 是一个自定义文字量，定义在名称空间 `std::literals::chrono_literals` 中，相当于 `chrono::days(1)` 的语法糖，表示某个月 1 号。`yearMonth / 1d` 中的 `/` 是 `yearMonth` 的操作符重载，表示当月第一天。

在代码第 10 行，用 `yearMonth / chrono::last` 表示当月最后一天。其中，`chrono::last` 是 C++20 中引入的，用来表示一个时间序列末尾的标记。最后，我们调用 `chrono::year_month_day` 转换成 `year_month_day`。

代码第 18 到 25 行，调用了 `renderWeekLine` 帮助我们生成格式化的标题，因为输出的日历需要将一周七天按照一定的布局输出到控制台上，该函数可以帮助我们完成渲染布局。代码第

20 行利用了 `locale` 输出中文，有兴趣可以自己看 `RenderUtils.cpp` 中的定义以及 `locale` 相关内容。

在代码第 28 到 31 行，生成日历的内容。由于日历里每周输出到一行中，因此这里遍历当月所有周，调用 `renderWeek` 完成一周的布局输出。

`renderWeekLine` 函数实现也会比较简单，输入参数是一个包含 `N` 个字符串的数组，数组元素就是在日历中的每一个格子中需要输出的内容（比如表头的周几或者日期），这里主要通过 `format` 将每一格内容的输出宽度限制在 4，确保布局工整。

渲染出来的效果如下图所示。

** 2023-02 **						
周一	周二	周三	周四	周五	周六	周日
30	31	*01	*02	*03	*04	*05
*06	*07	*08	*09	*10	*11	*12
*13	*14	*15	*16	*17	*18	*19
*20	*21	*22	*23	*24	*25	*26
*27	*28	01	02	03	04	05

最后，我们看一下 `utils::buildMonthWeeks` 的实现，代码实现在 `src/Utils/CalendarUtils.cpp` 中。

 复制代码

```
1 #include "utils/CalendarUtils.h"
2 #include <format>
3
4 namespace chrono = std::chrono;
5 using namespace std::literals::chrono_literals;
6
7 static uint32_t MaxWeekdayIndex = 6;
8
```

```

9 namespace calendarpp::utils {
10     std::vector<std::vector<std::chrono::year_month_day>> buildMonthWeeks(std::
11         // 获取当月第一天
12         const auto firstDay = yearMonth / 1d;
13         // 获取当月最后一天
14         const auto lastDay = chrono::year_month_day(yearMonth / chrono::last);
15
16         std::vector<std::vector<chrono::year_month_day>> monthWeeks;
17
18         // 将当月第一天设定为当日
19         auto currentDay = firstDay;
20         // 当每周当日出出当月最后一天时中止循环
21         while (currentDay <= lastDay) {
22             // 每次循环都计算出当日所在周的7天（周一到周日）
23             std::vector<chrono::year_month_day> currentMonthWeek;
24
25             // 通过weekday获取某一天是周几
26             auto currentWeekday = chrono::weekday{ std::chrono::sys_days{ curre
27             // 通过iso_encoding获取周几的编码（1-7）
28             auto currentWeekdayIndex = currentWeekday.iso_encoding() - 1;
29
30             // 计算本周第一天
31             auto firstDayOfWeek = chrono::year_month_day{
32                 std::chrono::sys_days{ currentDay } - chrono::days(currentWeekd
33             };
34
35             currentDay = firstDayOfWeek;
36             // 计算出本周的所有日期并添加到currentMonthWeek中
37             for (uint32_t weekdayIndex = 0; weekdayIndex <= MaxWeekdayIndex; ++
38                 currentMonthWeek.push_back(currentDay);
39
40                 currentDay = chrono::year_month_day{
41                     std::chrono::sys_days{ currentDay } + chrono::days(1)
42                 };
43             }
44             // 将计算好的当前周添加到monthWeeks中
45             monthWeeks.push_back(currentMonthWeek);
46         }
47
48         return monthWeeks;
49     }
50 }

```

在这段代码中，我们会生成当月的所有周，通过 **Calendar** 完成了大量的日期计算，相比自己实现格里高利历要方便不少。

导出日历

除了展示日历，我们继续讨论序列化日历的实现。在 `src/actions/ExportAction.cpp` 中定义了导出菜单项的代码实现。

 复制代码

```
1 #include "actions/ExportAction.h"
2 #include "utils/RenderUtils.h"
3 #include "utils/IOUtils.h"
4
5 #include <chrono>
6 #include <iostream>
7 #include <string>
8
9 namespace chrono = std::chrono;
10
11 namespace calendarpp::actions {
12     void exportCurrentYear() {
13         // 获取当前时间
14         chrono::time_point now{ chrono::system_clock::now() };
15         // 将时间转换为year_month_day
16         chrono::year_month_day ymd{ chrono::floor<chrono::days>(now) };
17         // 获取当前年份
18         chrono::year currentYear = ymd.year();
19
20         // 提示并读取用户输入
21         std::cout << "请输入导出文件路径: ";
22         std::string filePath;
23         std::getline(std::cin, filePath);
24
25         // 调用IO模块将renderYear的渲染结果输出到文件中
26         utils::writeFile(filePath, utils::renderYear(currentYear));
27
28         std::cout << std::format("已将日历导出到文件中: {}", filePath) << std::endl;
29         std::cout << std::endl;
30     }
31 }
```

这段代码非常简单，核心是通过 `renderYear` 渲染当年的日历，然后通过 `writeFile` 以 UTF-8 编码写入到文件中。

函数 `renderYear` 实现在 `src/utils/RenderUtils.cpp` 中，相关代码如下所示。

 复制代码

```
1 // 定义一年12个月的month常量
2 static std::vector<chrono::month> Months = {
3     chrono::January,
```

```

4     chrono::February,
5     chrono::March,
6     chrono::April,
7     chrono::May,
8     chrono::June,
9     chrono::July,
10    chrono::August,
11    chrono::September,
12    chrono::October,
13    chrono::November,
14    chrono::December,
15 };
16
17 std::string renderYear(std::chrono::year year) {
18     std::ostream os;
19
20     // 输出格式化的标题（年份）
21     std::string titleLine = std::format("**** {:%Y}年 ****", year);
22     os << std::format("{:^35}", titleLine) << std::endl;
23     os << std::format(":{:=^35}\n", "") << std::endl;
24
25     // 调用renderYearMonths并行生成12个月的日历
26     std::vector<std::string> renderedMonths(Months.size(), "");
27     renderYearMonths(renderedMonths, year);
28
29     // 将12个月的日历输入到输出流中
30     for (const std::string& renderedMonth : renderedMonths) {
31         os << renderedMonth;
32         os << std::endl;
33     }
34
35     // 返回渲染好的字符串
36     return os.str();
37 }
38
39 static void renderYearMonths(std::vector<std::string>& renderedMonths, chrono::
40     std::vector<std::jthread> renderThreads;
41
42     int32_t monthIndex = 0;
43     for (const auto& currentMonth : Months) {
44         auto currentYearMonth = year / currentMonth;
45         auto& renderedMonth = renderedMonths[monthIndex];
46
47         // 创建jthread对象，开启计算线程，每个线程负责生成一个月的日历
48         renderThreads.push_back(std::jthread([currentYearMonth, &renderedMonth]
49             renderedMonth = renderMonth(currentYearMonth);
50             }));
51
52         ++monthIndex;
53     }
54
55     // 退出函数时，所有jthread对象会自动等待计算完成然后退出

```


我们重点看一下 `renderYearMonths` 函数的实现。

该函数定义了一个 `jthread` 数组，然后循环创建了 12 个线程，用于分别调用 `renderMonth` 渲染各自月份的日历。由于 `jthread` 会在析构时自动等待本线程完成。因此，我们并不需要去 `join` 这些线程，在退出 `renderYearMonths` 之前，所有的工作线程肯定能结束任务。

最后看一下 `writeFile` 的实现，代码在 `src/utls/IOUtils.cpp` 中。

 复制代码

```

1  #include "utls/IOUtils.h"
2  #include "Logger.h"
3
4  #include <filesystem>
5  #include <format>
6  #include <iostream>
7
8  namespace fs = std::filesystem;
9
10 namespace calendarpp::utls {
11     void writeFile(const std::string& filePath, const std::string& fileContent)
12         auto& logger = getLogger();
13
14         // 检测文件是否存在
15         if (fs::exists(filePath)) {
16             logger.warning(std::format("Override existed file: {}", filePath));
17         }
18
19         // 由于源代码使用UTF-8，生成的字符串就是UTF-8，因此可以直接强制类型转换构建u8string
20         std::u8string utf8FileContent(reinterpret_cast<const char8_t*>(fileCont
21
22         // 将u8string以二进制形式写入到文件中
23         std::ofstream outputFile(filePath, std::ios::binary);
24         outputFile.write(reinterpret_cast<char*>(utf8FileContent.data()), utf8F
25     }
26 }
```

在这段代码中，首先通过文件系统库中的 `std::filesystem::exists` 检测文件是否存在，如果存在则提示用户会覆盖原有文件。

接着，将字符串转换成 `u8string`，由于源代码使用 `UTF-8`，生成的字符串就是 `UTF-8`。因此，我们可以直接强制类型转换构建 `u8string`。

最后，将 `u8string` 的 `data` 强制类型转换为 `char*`，并通过二进制形式写入到文件中。

总结

在这一讲中，我们通过一个工程，串讲了一系列 C++20 带来的重要库变更。我们在原有的日志框架基础上，追加了 `source_location`，用于记录打印日志时代码的所在位置。除此之外，将传统的输出流替换成 `sync stream`，就可以轻松实现输出的线程同步控制。

在实现的过程中，我们还使用了 `TimeZone` 获取带时区的本地时间，并通过使用 `Calendar` 完成日历计算。这些针对 `chrono` 的标准库补充，大大降低了时间处理的复杂度。

最后，我们利用 `u8string`，轻松实现了 UTF-8 编码的文本导出。


通过这些案例，我们可以感受到，**现代 C++ 的库变更建立在新的核心语言特性基础上，为我们日常编程工作提供了极大的便利。避免造轮子，提升编程效率**——这些对于库的核心变更来说是核心议题，所以我们应该保持对标准库演进的持续关注。

课后思考

现在日历中显示的时间是包含时区信息的本地时间，如果我们希望日历中显示的当前时间是 UTC 时间（日历显示依然按照本地时间计算，不需要变化），我们要对代码做什么改动？

欢迎给出你的方案，与大家一起分享。我们一同交流。下一讲见！

分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 18 | 其他重要标准库特性：还有哪些库变更值得关注？

下一篇 20 | 漫游C++23：更好的C++20

精选留言 (1)

写留言



peter

2023-03-07 来自北京

C++20之前没有Calendar吗？

文中“Calendar 是 C++20 引入到 chrono 中新的库特性”，之前没有Calendar吗？有点小怀疑。如果没有，以前是怎么处理日历问题的？

作者回复: 需要基于非常基础的C函数进行封装，没有标准化方案。

