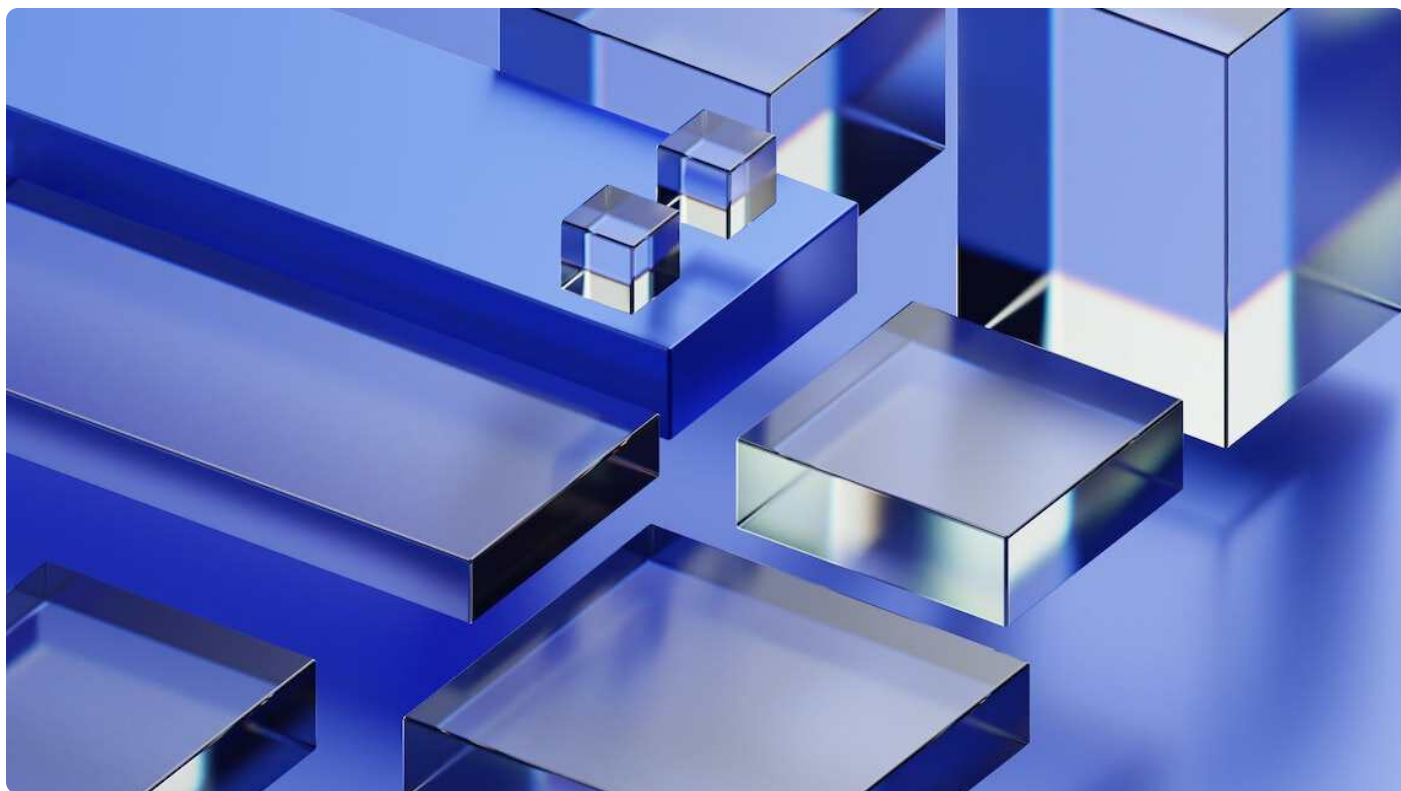


14 | Formatting: 千呼万唤始出来的新format标准

2023-02-22 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 13:09 大小 12.02M



你好，我是卢誉声。

在 C++ 中，我们经常讨论一个看似简单的问题——**如何实现格式化字符串和格式化输出？**

这个问题核心在于字符串格式化，考虑到 C++ 向下兼容的问题，想做出一个能让大家满意的字符串格式化标准方案，其实并不容易。在过去的标准中，C++ 标准委员会一直通过各种修修补补，尝试提供一些格式化的辅助方案，但始终没有一个风格一致的标准化方案。

好在 C++20 及其后续演进中，终于出现了满足我们要求的格式化方案。因此，在这一讲中，我们就聚焦于讲解这个新的字符串格式化方案。

好，话不多说，就让我们开始今天的内容吧 (课程配套代码可以从 [这里](#) 获取)。

复杂的文本格式化方案

首先，我们要弄明白什么是“文本格式化”。

下面一个常见的 HTTP 服务的日志输出，我们结合这个典型例子来讲解。

复制代码

```
1 www | [2023-01-16T19:04:19] [INFO] 127.0.0.1 - "GET /api/v1/info HTTP/1.0"
```

可以看到，日志输出中包含了一些固定字符、需要根据实际情况替换的值。输出类似内容的这种需求就被称为“文本格式化”。

事实上，许多现代编程语言都提供了便利、安全的格式化方案。

但遗憾的是，在 C++20 以前虽然也有文本化格式方案，但都存在着这样那样的缺陷，而且不够现代化。

甚至就此出现了一些临时拼凑的方案，接下来，我们通过一个表格来回顾一下，在 C++20 以前的文本格式化方案。

序号	方案	说明	优缺点	具体优缺点解释
1	C 风格	在 C 语言中提供的接口 <code>sprintf</code> 、 <code>printf</code> 和 <code>fprintf</code> 函数进行格式化。	优点	简单、清晰，只要熟悉了基本格式化字符串方法即可。
			缺点	类型不安全、缓冲区溢出问题、线程安全问题。
2	C++ 字符串拼接	在 C++ 中为了解决 C 风格字符串的各种问题，提供了标准的 <code>string</code> 类型。标准库也为 <code>string</code> 提供了 <code>+</code> 操作符的重载，支持通过 <code>+</code> 进行字符串拼接。	优点	相比于传统的 C 风格文本格式化方案，这种依赖于 <code>string</code> 类型的方案在类型上当然更加安全，至少不会发生缓冲区溢出，如果通过拷贝传值也不会发生数据竞争。
			缺点	C++ 没有提供类似于其他语言中（在拼接其他数据类型与字符串时）将其他数据类型隐式转换成 <code>string</code> 类型的能力。 在 C++11 以前，除了输入流方案以外，C++ 甚至没有提供将基础数值类型转换成 <code>string</code> 类型的标准方案，所以这种方案虽然简单利于理解，但在 C++ 中使用实在不太方便。
3	C++ 流	在设计之初，C++ 想鼓励大家使用的就是基于 C++ 流的格式化方案。	优点	C++ 标准库在很长一段时间内希望用流解决所有类似于输入输出的问题，因此设计了复杂的输入输出流继承方案与统一的接口。类型安全、可以避免缓冲区溢出、代码风格一致（与 C++ 的标准输出流）。
			缺点	相对于格式化字符串方案，基于流的代码会显得更加冗长，随之导致码可读性受到影响。同时，在修改输出格式或内容时不方便。

看完表格，你应该也发现了。在 C++20 出现以前，各种文本格式化方案都存在一些较为明显的缺点，无论是本身的安全性问题，还是编码层面的易用性方面。这导致了，C++ 开发者在选择文本格式化方案的时候难以抉择。

幸运的是，C++20 终于提出了标准化的文本格式化方案——这就是 **Formatting** 库。

Formatting

Formatting 库提供了类似于其他现代化编程语言的文本格式化接口，而且这些接口设计足够完美、便于使用。同时，它还提供了足够灵活的框架。因此，我们可以轻松地对其进行扩展，支持更多的数据类型与格式。

想要了解 **Formatting** 库，我们循序渐进。先从最基础的格式化函数 **format** 开始，其定义是后面这样。

 复制代码

```
1 template<class... Args>
2 std::string format(std::format_string<Args...> fmt, Args&&... args);
```

该函数的第一个参数是格式化字符串，描述文本格式，后续参数就是需要被格式化的其他参数。

关于 `std::format_string<Args...>` 这个类型，我们在后面深入理解 **Formatting** 中再具体讨论，现在你只需要知道，这是用格式描述的字符串即可。

下面是使用 **format** 函数编写的日志输出代码。

 复制代码

```
1 #include <iostream>
2 #include <format>
3 #include <string>
4 #include <cstdint>
5 #include <chrono>
6
7 // 使用 std::chrono 来打印日志的时间
8 using TimePoint = std::chrono::time_point<std::chrono::system_clock>;
9
10 struct HttpLogParams {
11     std::string user;
```

```
12     TimePoint requestTime; // C++20 提供了chrono对format的支持
13     std::string level;
14     std::string ip;
15     std::string method;
16     std::string path;
17     std::string httpVersion;
18     int32_t statusCode;
19     int32_t bodySize;
20     std::string refer;
21     std::string agent;
22 };
23
24 void formatOutputParams(const HttpLogParams& params);
25
26 int main() {
27     HttpLogParams logParams = {
28         .user = "www",
29         .requestTime = std::chrono::system_clock::now(),
30         .level = "INFO",
31         .ip = "127.0.0.1",
32         .method = "GET",
33         .path = "/api/v1/info",
34         .httpVersion = "HTTP/1.0",
35         .statusCode = 200,
36         .bodySize = 6934,
37         .refer = "http://127.0.0.1/index.html",
38         .agent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:108.0) Gecko/201
39     };
40
41     formatOutputParams(logParams);
42
43     return 0;
44 }
45
46 void formatOutputParams(const HttpLogParams& params) {
47     std::string logLine = std::format("{0:<16}|{1:%Y-%m-%d}T{1:%H:%M:%OS}Z {2}
48         params.user,
49         params.requestTime,
50         params.level,
51         params.ip,
52         params.method,
53         params.path,
54         params.httpVersion,
55         params.statusCode,
56         params.bodySize,
57         params.refer,
58         params.agent
59     );
60
61     std::cout << logLine << std::endl;
62 }
```

C++20 在 C++11 的基础上，为 chrono 库提供了完善的 format 支持，我们再也不需要使用旧的 C 风格时间格式化函数了（见代码第 12 行）。

这里简单说明一下 format 的格式化字符串格式。格式化字符串由以下三类元素组成。

- 普通字符（除了 { 和 } 以外），这些字符会被直接拷贝到输出中，不会做任何更改。
- 转义序列，包括 {{ 和 }}，在输出中分别会被替换成{和}。
- 替换字段，由 { ... } 构成，这些替换字段会替换成 format 后续参数中对应的参数，并根据格式控制描述生成输出。

对于替换字段的两种形式，你可以参考后面这张表格。

序号	替换字段形式	说明
1	{[arg-id]}	不指定输出格式的替换字段，arg-id 为 format 后续参数中对应参数的序号，从 0 开始，如果 arg-id 被省略，那么就按照次序使用 format 的后续参数。
2	{[arg-id] : format-spec}	在第一种形式的基础上指定输出格式说明。如果输出参数为基础类型或者 string 类型对象，那么就遵从标准格式化规范，将在后文说明。如果输出参数为 chrono类型，那么就遵从 chrono 的格式化规范。否则会遵从用户自定义的格式化规范（如果用户定义了相应的 formatter）。



如果你了解过 Python，就会发现 format 函数的格式化字符串格式，其实类似于 Python 的格式化规范。不得不承认的是，C++20 标准借鉴了相应的规范。

除了最简单的 format 参数，C++20 还提供了三个有用的工具函数，作为扩展功能。

1. format_to
2. format_to_n
3. formatted_size

你可以参考下面的示例代码，来看看它们的具体用法。



```

1 #include <iostream>
2 #include <format>
3 #include <string>
4
5 int main() {
6     // format_to
7     // 将生成的文本输出到一个输出迭代器中，
8     // 其他与format一致，这样可以兼容标准STL算法函数的风格，
9     // 也便于将文本输出到其他的流中或者自建的字符串类中。
10    std::string resultLine1;
11    std::format_to(std::back_inserter(resultLine1), "{} + {} = {}", 1, 2, 1 + 2);
12    std::cout << resultLine1 << std::endl;
13
14    // format_to_n
15    // 将生成的文本输出到一个输出迭代器中，同时指定输出的最大字符数量。
16    // 其他与format一致，相当于format_to的扩展版本，
17    // 在输出目标有字符限制的时候非常有效。
18    std::string resultLine2(5, ' ');
19    std::format_to_n(resultLine2.begin(), 5, "{} + {} = {}", 1, 2, 1 + 2);
20    std::cout << resultLine2 << std::endl;
21
22    // formatted_size
23    // 获取生成文本的长度，参数与format完全一致。
24    //
25    auto resultSize = std::formatted_size("{} + {} = {}", 1, 2, 1 + 2);
26    std::cout << resultSize << std::endl;
27
28    std::string resultLine3(resultSize, ' ');
29    std::format_to(resultLine3.begin(), "{} + {} = {}", 1, 2, 1 + 2);
30    std::cout << resultLine3 << std::endl;
31 }

```

可以看出，这三个函数使用方法基本和 `format` 没有太大区别。

这里我们重点留意一下 `formatted_size`。如果部分场景需要生成特定长度的输出缓冲区，那么我们就可以先通过 `formatted_size` 获取输出长度，然后分配特定长度缓冲区，最后再输出。除此以外，在只需要获取字符数量的场景中，也可以使用这个函数。

从上面案例可以看到，`format` 函数的基本用法简单易懂。接下来，我们进一步讨论有关 `format` 的具体细节，先从格式化参数包开始。

格式化参数包

`format` 函数，可以直接以函数参数形式进行传递。此外，C++20 还提供了 `format_args` 相关接口，可以把“待格式化的参数”合并成一个集合，通过 `vformat` 函数进行文本格式化。

你可以结合后面的代码来理解。

 复制代码

```
1 #include <iostream>
2 #include <format>
3 #include <string>
4 #include <cstdint>
5
6 int main() {
7     std::string resultLine1 = std::vformat("{} * {} = {}", std::make_format_arg
8         3, 4, 3 * 4
9     );
10    std::cout << resultLine1 << std::endl;
11
12    std::format_args args = std::make_format_args(
13        3, 4, 3 * 4
14    );
15
16    std::string resultLine2;
17    std::vformat_to(std::back_inserter(resultLine2), "{} * {} = {}", args);
18    std::cout << resultLine2 << std::endl;
19 }
```

针对上述代码中用到的类型和函数，我依次为你解释一下。

第一，format_args 类型，表示一个待格式化的参数集合，可以包装任意类型的待格式化参数。这里需要注意的是 format_args 中包装的参数是**引用语义**，也就是并不会拷贝或者扩展包装参数的生命周期，所以开发者需要确保被包装参数的生命周期。所以一般来说，format_args 也就用于格式化函数的参数，不建议用于其他用途。

第二，make_format_args 函数，用于通过一系列参数构建一个 format_args 对象。类似地，需要注意返回的 format_args 的引用语义。

第三，vformat 函数。包含两个参数，分别是格式化字符串（具体规范与 format 函数完全一致）和 format_args 对象。该函数会根据格式化字符串定义去 format_args 对象中获取相关参数并进行格式化输出，其他与 format 函数没有差异。

第四，vformat_to 函数。该函数与 format_to 类似，都是通过一个输出迭代器进行输出的。差异在于，该函数接收的“待格式化参数”，需要通过 format_args 对象进行包装。因此，

`vformat` 可以在某些场景下替代 `format`。至于具体使用哪个，你可以根据自己的喜好进行选择。

深入理解 Formatting

在了解了 `Formatting` 的基本用法后，我们有必要深入 `Formatting` 的细节，了解如何基于 `Formatting` 库进行扩展，来满足我们的复杂业务需求。

首先，`Formatting` 库的核心是 `formatter` 类，对于所有希望使用 `format` 进行格式化的参数类型来说，都需要按照约定实现 `formatter` 类的特化版本。

`formatter` 类主要完成的工作就是：格式化字符串的解析、数据的实际格式化输出。`C++20` 为基础类型与 `string` 类型定义了标准的 `formatter`。此外，我们还可以通过特化的 `formatter` 来实现其他类型、自定义类型的格式化输出。

下面，我们先看一下标准 `formatter` 的格式化标准，然后在此基础上实现自定义 `formatter`。

标准格式化规范

`C++ Formatting` 的标准格式化规范，是以 `Python` 的格式化规范为基础的。基本语法是后面这样。

1 填充与对齐 符号 # 0 宽度 精度L 类型

复制代码

这里的每个参数**都是可选参数**，我们解释一下这些参数。

第一，填充与对齐，用于设置填充字符与对齐规则。

该参数包含两部分，第一部分为填充字符，如果没有设定，默认使用空格作为填充。第二部分为填充数量与对齐方式，填充数量就是指定输出的填充字符数量，对齐方式指的是待格式化参数输出时相对于填充字符的位置。

目前 `C++` 支持三种对齐方式，你可以参考后面的表格。

序号	对齐语法	说明
1	<	格式化参数输出位置在填充字符的开始位置，这是非数字类型的默认对齐方式。
2	>	格式化参数输出位置在填充字符的结束位置，这是数字类型的默认对齐方式。
3	^	格式化参数输出位置在填充字符的中间，换言之就是在格式化参数的左右两侧各插入一般数量的填充字符。



第二，“符号”“#”和“0”，用于设定数值类型的前缀显示方式。我们分别来看看。

“符号”可以设置数字前缀的正负号显示规则。需要注意的是，“符号”也会影响 inf 和 nan 的显示方式。后面的表格包含了这三种情况。

序号	符号	说明
1	+	正数前会插入 +，负数前插入 -，也就是输出 1 时会变成 +1，输出 -1 时为 -1。
2	-	只有负数前会插入符号，这也是默认显示行为。
3	空格	正数前会插入空格，负数前插入 -1，也就是输出 1 时会变成 1，输出 -1 时为 -1。



“#”会对整数和浮点数有不同显示行为。

如果被格式化参数为整数，并且将整数输出设定为二进制、八进制或十六进制时会在数字前添加进制前缀，也就是 0b、0 和 0x。如果被格式化参数为浮点数，那么即使浮点数没有小数位数，也会强制在数字后面追加一个小数点。

“0”用于为数值输出填充 0，并支持设置填充位数。比如 04 就会填充 4 个 0。

第三，宽度与精度。

宽度用于设置字段输出的最小宽度，可以使用一个十进制数，也可以通过 {} 引用一个参数。

精度是一个以 . 符号开头的非负十进制数，也可以通过 {} 引用一个参数。对于浮点数，该字段可以设置小数点的显示位数。对于字符串，可以限制字符串的字符输出数量。

宽度与精度都支持通过 {} 引用参数，此时如果参数不是一个非负整数，在执行 **format** 时就会抛出异常。

第四，L 与类型。

L 用于指定参数以特定语言环境（**locale**）方式输出参数。如果感兴趣的话，你可以参考标准文档来查询有关语言环境的具体说明。参考标准文档足以涵盖语言环境的问题，因此不是我们讨论的重点。

类型选项用于设置参数的显示方式，我同样准备了表格，为你梳理了 **C++20** 支持的所有参数类型选项。

参数类型	类型选项	解释
字符串	无选项或者 s	将字符串直接拷贝至输出（默认行为）
整型 (不包括字符类型与 bool)	b	以二进制格式显示，实际调用了 to_chars，前缀为 0b
	B	与 b 相同，前缀为 0B
	d	将整数直接转换成编码对应的字符输出，实质是调用了 static_cast<char>(value) 。如果整数超出了字符编码范围那么会抛出异常。
	d	以十进制格式显示，实际调用了 to_chars
	o	以八进制格式显示，实际调用了 to_chars，前缀为 0
	x	以十六进制格式显示，实际调用了 to_chars，前缀为 0x
	X	与 x 相同，实际调用了 to_chars，会将所有大于 9 的字符都转换为大写，前缀为 0X
字符类型	无选项或者 c	将字符直接拷贝至输出（默认行为）
	b, B, d, o, x, X	使用对应进制的数字形式显示字符编码
布尔类型	无选项或者 s	显示为 true 或 false
	b, B, d, o, x, X	使用对应进制的数字形式显示字符编码
浮点类型	a	以十六进制显示浮点数，如果指定了精度选项也会生效，实际调用了 to_chars
	A	与 a 相同，会将所有大于 9 的数字字符转换成大写形式
	e	以科学计数法显示，如果指定了精度选项也会生效，实际调用了 to_chars
	E	与 e 相同，使用 E 表示指数
	f, F	使用固定位数显示数字，如果没有指定精度选项，默认为 6 位小数，实际调用了 to_chars
	g	自动选用固定位数或科学计数法中最短表示形式输出，默认为 6 位小数，实际调用了 to_chars
	G	与 g 相同，使用 E 表示指数
	无选项	如果指定了精度选项，同 g 选项，如果没有指定精度则直接调用 to_chars 输出（默认行为）

自定义 formatter

Formatting 库中的 formatter 类型对各种类型的格式化输出毕竟是有限的——它不可能覆盖所有的场景，特别是我们的自定义类型。

因此，它也支持开发者对 formatter 进行特化，实现自定义的格式化输出。现在，让我们来看看如何自定义 formatter。

我们先看一个最简单的自定义 formatter 案例。

 复制代码

```
1 #include <format>
2 #include <iostream>
3 #include <vector>
4 #include <cstdlib>
5
6 template<class CharT>
7 struct std::formatter<std::vector<int32_t>, CharT> : std::formatter<int32_t, Ch
8     template<class FormatContext>
9     auto format(std::vector<int32_t> t, FormatContext& fc) const {
10         auto it = std::formatter<int32_t, CharT>::format(t.size(), fc);
11
12         for (int32_t v : t) {
13             *it = ' ';
14             it++;
15
16             it = std::formatter<int32_t, CharT>::format(v, fc);
17         }
18
19         return it;
20     }
21 };
22
23 int main() {
24     std::vector<int32_t> v = { 1, 2, 3, 4 };
25
26     // 首先，调用format输出vector的长度，
27     // 然后遍历vector，每次输出一个空格后再调用format输出数字。
28     std::cout << std::format("{: #x}", v);
29 }
```

在这段代码中，实现了格式化显示 `vector<int32_t>` 类型的对象的功能。我们重点关注的是第 7 行实现的 `formatter` 特化——`std::formatter<std::vector<int32_t>, CharT>`。

其中，`CharT` 表示字符类型，它可以根据用户的实际情况替换成 `char` 或者 `wchar_t` 等。

通过代码你会发现，我们重载了 `format` 成员函数，该函数用于控制格式化显示。该函数包含两个参数。

1. `t: std::vector<int32_t>`: 被传入的待格式化参数。
2. `fc: FormatContext&`: 描述格式化的上下文。

作为延伸阅读，你可以参考 `std::basic_format_context` 这个类型的定义，了解格式化的上下文中具体包含的信息。当然了，在编码过程中，IDE 也会在使用它时给出提示。

`format` 函数返回一个迭代器，表示下一个用于输出的位置，我们通过控制这个迭代器，就可以输出自己想要的格式化字符了。

示例没有实现 `parse` 来解析格式化字符串，如果你有兴趣的话，课后可以自行了解相关细节。

总结

传统的文本格式化方案包括基于 C 接口的格式化输出、C++ 字符串拼接或 C++ 流这几种方式。它们各有优劣，但往往难以解决类型安全、缓冲区溢出、线程安全等问题。

C++20 的推出改变了这一局面，我们可以利用 `Formatting` 库和 `formatter` 类型高度灵活地实现格式化文本输出。其中 `formatter` 支持特化，因此我们可以通过这个全新的方式，解决长久以来缺乏标准化的文本格式化的问题。

对于 `formatter` 的特化实现，我们记住两个重点即可。


1. 重载 `format` 函数，实现输出自己想要的格式化文本。
2. 重载 `parse` 函数，实现自定义格式化文本解析。

课后思考

我们在这一讲中展示了如何通过重载 `formatter` 中的 `format` 函数，实现了自定义输出格式化文本。那么，你能否进一步拓展这一案例，通过重载 `parse` 来实现解析格式化字符串？

欢迎给出你的代码方案。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

- 上一篇
- 13 | Ranges实战：数据序列函数式编程
- 下一篇
- 15 | Formatting实战：如何构建一个数据流处理实例？

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。