

## 18 | 其他重要标准库特性：还有哪些库变更值得关注？

2023-03-03 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 15:24 大小 14.06M



你好，我是卢誉声。

在第二章的开头，我们曾提到过，通常意义上所讲的 C++，其实是由核心语言特性和标准库（C++ Standard Library）共同构成的。

在学习了 Ranges 库和 Formatting 库之后，还有一些比较重要的标准库变更值得我们关注，包括 jthread、source location、Sync stream 和 u8string。今天，我会带你了解它们的用法和注意事项。

好，话不多说，就让我们从 jthread 开始今天的学习之旅吧（课程配套代码可以从 [这里](#) 获取）。

### jthread

长久以来，在 C++ 中实现多线程都需要借助于操作系统 API 或者第三方库。好在这一情况在 C++11 中得以扭转，C++11 为标准库带来了并发库，即标准的 `thread` 类。

但是，我们在工程中使用 C++11 的 `thread` 类，仍然存在一些问题。

首先是线程运行时默认行为不够灵活。`thread` 的内部线程是进程的子线程，当 `thread` 还关联着一个活动线程时，C++ 运行时会调用 `terminate()` 中断整个程序的执行，这种行为对于很多没有认真管理线程资源的程序，不但非常危险，而且难以追踪。

另外，`thread` 类还缺乏强制取消或通知取消线程的功能，在很多使用线程的场景中，这都是经常需要使用到的功能。还记得么？在第七讲至第十讲中讨论 C++ coroutines 的时候，我们就不得不自己实现了请求取消线程特性。

那时候我们的实现非常粗糙。比如说，没有考虑确保请求线程的线程安全，也无法告知请求方是否成功发送请求。如果要实现这些特性需要考虑很多边界条件，还真不容易。

由于这些问题的存在，我们在实际开发过程中使用 C++11 标准 `thread` 类时，就需要非常小心谨慎，说白了就是不但难用，而且容易出错。为此，C++20 终于增加了 `jthread` 类来解决这些问题。

我们先结合后面这段示例代码，对 `jthread` 建立初步的认识。

 复制代码

```
1 #include <iostream>
2 #include <thread>
3 #include <chrono>
4 #include <cstdint>
5 #include <string>
6
7 void simpleSleep() {
8     using namespace std::literals::chrono_literals;
9
10    std::cout << "[SIMPLE] Before simple sleep" << std::endl;
11    std::this_thread::sleep_for(2000ms);
12    std::cout << "[SIMPLE] After simple sleep" << std::endl;
13 }
14
15 // jthread的工作函数可以通过第一个类型为stop_token的参数获取线程中断请求
16 void stopTokenSleep(std::stop_token token, std::string workerName) {
17     using namespace std::literals::chrono_literals;
18
```

```

19     std::cout << "Worker name: " << workerName << std::endl;
20
21     while (true) {
22         std::cout << "[STOP_TOKEN] Before sleep" << std::endl;
23         std::this_thread::sleep_for(100ms);
24         std::cout << "[STOP_TOKEN] After sleep" << std::endl;
25
26         // 调用stop_requested可以得知是否有其他线程请求中断本线程
27         if (stoken.stop_requested()) {
28             std::cout << "[STOP_TOKEN] Received stop request" << std::endl;
29
30             return;
31         }
32     }
33 }
34
35 int main() {
36     // ms等自定义文字量定义在std::literals::chrono_literals名称空间中
37     using namespace std::literals::chrono_literals;
38
39     std::cout << "[MAIN] Before create simple thread" << std::endl;
40     // 创建线程
41     std::jthread simpleWorker(simpleSleep);
42     std::cout << "[MAIN] After create simple thread" << std::endl;
43
44     std::cout << "[MAIN] Before create stop token thread" << std::endl;
45     // 创建线程
46     std::jthread stopTokenWorker(stopTokenSleep, "Worker1");
47     // 注册request_stop成功后的回调
48     std::stop_callback callback(stopTokenWorker.get_stop_token(), [] {
49         std::cout << "[MAIN] Called after thread stop requested" << std::endl;
50     });
51     std::cout << "[MAIN] After create stop token thread" << std::endl;
52
53     std::this_thread::sleep_for(500ms);
54
55     std::cout << "[MAIN] Request stop" << std::endl;
56     stopTokenWorker.request_stop();
57
58     std::cout << "[MAIN] Main function exited" << std::endl;
59
60     return 0;
61 }

```

在这段代码中，我们没有使用 `thread` 类，而是通过 `jthread` 类来创建线程。

代码中创建了两个子线程，从第 39 行到 42 行创建了第 1 个子线程对象 `simpleWorker`，从第 44 到 56 行创建了第 2 个子线程对象 `stopTokenWorker`。

这里我们并没有像 `thread` 类一样，调用 `join` 主动等待线程结束。但是，程序会自动等待所有线程停止后才会退出，这是如何实现的呢？

这一切的玄机，都在 **`jthread` 类的析构函数**中！

在 `jthread` 对象析构时，如果 `jthread` 依然关联了活动线程（线程为 `joinable`），会自动调用关联线程的 `request_stop`，并调用 `join` 等待线程结束。线程结束后才会继续执行析构。

现在回到我们的代码，`main` 函数在 `return` 前，会自动依次调用两个 `jthread` 对象的析构函数，销毁线程对象。这时主线程就会自动 `join` 这两个线程，确保线程对象销毁之前，子线程已经结束。这种默认行为可以确保主线程结束前，子线程必定都已经结束，不会引发不可预期的错误。

如果不希望 `jthread` 自动 `join`，可以不在栈上直接创建 `jthread` 对象、或者直接调用 `detach` 解除线程与 `jthread` 对象的关联。从 C++20 开始支持的 `jthread`，既保留了灵活性，又确保了默认行为的安全性，更符合一般使用线程的场景。

另外，`stopTokenWorker` 演示了 `jthread` 的第二个重要特性—— **`stop_token`**。

每个 `jthread` 的工作函数的第一个参数，都可以定义为 `std::stop_token` 类型。这时，其他线程可以调用该 `jthread` 对象的 `request_stop` 成员函数，向 `jthread` 绑定的线程发送中断请求。

实际上 `request_stop` 并不会真的中断线程，而是将 `stop_token` 对象的 `stop_requested` 设置为 `true`。`jthread` 绑定的线程可以通过 `stop_requested` 获知是否有线程通知其中断，并自行决定是否结束线程。

按照标准规定，调用 `request_stop` 的过程是线程安全的——只有一个线程可以成功发送请求，一个线程发送请求成功后，其他线程调用 `request_token` 会失败，但不会引发异常。还有一个作用相同的类型是 `stop_source`，如果你感兴趣，可以自己阅读相关文档了解如何使用。

代码的第 48 到 50 行还演示了 **`stop_callback`** 的用法，该类用于在一个 `jthread` 上注册一个成功调用 `request_stop` 后的回调函数——如果其他线程已经成功 `request_stop` 了一个 `jthread` 线程，那么，这个线程调用 `request_stop` 是不会触发本线程注册的回调函数的。

另外我们需要注意的是，一个线程可以注册多个 `stop_callback`，标准只能保证所有的 `stop_callback` 会被同步依次调用，不能保证 `stop_callback` 的调用顺序（也就是并不一定按照注册顺序调用）。

综上所述，我们可以看到 `jthread` 提供了安全的默认行为，具备线程中断机制，可以根据实际情况调整具体行为，在确保安全的前提下支持灵活调用，在大多数场景中是更符合实际需求的设计。

## source location


了解了 `jthread` 后，我们继续了解下一个相当重要的标准库变更——`source location`。

在 C++ 中，如果我们希望获取当前行的源代码位置，一直都需要使用 C 预处理指令中预设的 `__FILE__` 和 `__LINE__` 两个宏。

但在 C++ 中，这两个宏并不足以支持我们对程序跟踪调试的需求。

- `__LINE__` 无法自动包含函数名等关键信息，需要采用 `#line` 指令手动控制输出的标记。
- `__FILE__` 和 `__LINE__` 都会在预处理阶段被替换为特定的字符串。但是，对于 C++ 中使用的模板函数来说，只有在编译阶段，才能获知当前行所在函数的参数实例化信息。因此，使用 `__LINE__` 也无法获取所在行的模板实例化情况。

C++20 终于提出了 `source_location` 这个标准类，可以获取当前行更完整的源代码信息。我们结合这段示例代码来看看。

 复制代码

```
1 #include <iostream>
2 #include <source_location>
3 #include <string>
4
5 void logInnerLocation(const std::string& message);
6 void logLocation(
7     const std::string& message,
8     std::source_location location = std::source_location::current()
9 );
10
11 int main() {
12     logInnerLocation("Inner location message");
13     // 通过默认参数通过current获取source_location对象
```

```

14 // 这时source_location包含的信息就是在main内
15 logLocation("Location message");
16
17 return 0;
18 }
19
20 void logInnerLocation(const std::string& message) {
21     // 在logInnerLocation内部通过current获取source_location对象
22     // 这时source_location包含的信息就是在logInnerLocation内
23     std::source_location location = std::source_location::current();
24
25     std::cerr << message << std::endl <<
26         " [" <<
27         location.file_name() << "(" <<
28         location.line() << ":" <<
29         location.column() << ")@" <<
30         location.function_name() << "]" << std::endl;
31 }
32
33 void logLocation(
34     const std::string& message,
35     std::source_location location
36 ) {
37     std::cerr << message << std::endl <<
38         " [" <<
39         location.file_name() << "(" <<
40         location.line() << ":" <<
41         location.column() << ")@" <<
42         location.function_name() << "]" << std::endl;
43 }

```

代码中通过 `source_location` 的静态成员函数 `current`，获取了当前位置的源代码信息。其中，获取到的信息包含 `file_name`、`line`、`column` 和 `function_name` 这几个字段，每个字段的含义你可以参考下表。

序号	字段	含义
1	file_name	源代码文件名，返回的文件名和路径的具体格式由各编译器实现自行决定。
2	line	源代码所在行数，有些实现如果无法获取到具体行号会返回 0。
3	column	源代码所在列数。
4	function_name	代码所在函数名、返回的格式以及包含的信息完全由编译器实现自行决定，有些实现甚至可以包含模板参数实例化情况以及函数类型（比如是否为一个 Lambda 函数）。



可以看到，`source_location` 这一标准类，能为我们提供精确的编译时源代码信息，涵盖了普通函数调用和模板函数所有使用场景。这对于发布用于调试的程序极为有用。

## sync stream

在多线程场景中，使用 C++ 传统输出流接口会存在一个问题：多个线程直接向同一个输出流对象输出内容时，会得到无法预估的错乱输出。

因此，我们一般需要自己通过互斥锁等方式，实现输出的线程同步。这种方式虽然能够解决问题，但是编程效率低下，运行时也会有潜在的性能问题。

C++20 提出的 `sync stream` 解决了这个问题。先来看一段代码。

复制代码

```
1 #include <iostream>
2 #include <string>
3 #include <syncstream>
4 #include <thread>
5 #include <vector>
6 #include <cstdlib>
7
8 namespace chrono = std::chrono;
9
10 // 普通stream版本
11 void coutPrinter(const std::string message1, const std::string message2);
12 // syncstream版本
13 void syncStreamPrinter(const std::string message1, const std::string message2);
14
15 int main() {
```

```

16     std::cout << "Cout workers:" << std::endl;
17     // 创建多个thread
18     std::vector<std::thread> coutWorkers;
19     for (int32_t workerIndex = 0; workerIndex < 10; ++workerIndex) {
20         std::thread coutWorker(coutPrinter,
21                                "ABCDEFGHJKLMNOPQRSTUVWXYZ",
22                                "abcdefghijklmnopqrstuvwxyz"
23                                );
24         coutWorkers.push_back(std::move(coutWorker));
25     }
26
27     // 普通thread需要手动join
28     for (auto& worker : coutWorkers) {
29         if (worker.joinable()) {
30             worker.join();
31         }
32     }
33
34     std::cout << "SyncStream workers:" << std::endl;
35     // 创建多个jthread, 会在析构时自动join
36     std::vector<std::jthread> syncStreamWorkers;
37     for (int32_t workerIndex = 0; workerIndex < 10; ++workerIndex) {
38         std::jthread syncStreamWorker(
39             syncStreamPrinter,
40             "ABCDEFGHJKLMNOPQRSTUVWXYZ",
41             "abcdefghijklmnopqrstuvwxyz"
42             );
43         syncStreamWorkers.push_back(std::move(syncStreamWorker));
44     }
45
46     return 0;
47 }
48
49 void coutPrinter(const std::string message1, const std::string message2) {
50     std::cout << message1 << " " << message2 << std::endl;
51 }
52
53 void syncStreamPrinter(const std::string message1, const std::string message2)
54     // 使用std::osyncstream包装输出流对象即可
55     std::osyncstream(std::cout) << message1 << " " << message2 << std::endl;
56 }

```

我们在代码 18 到 32 行，创建了多个 `thread` 对象，并以 `coutPrinter` 为入口函数，创建线程后逐个调用 `join` 等待线程完成。`coutPrinter` 中直接调用 `cout` 输出字符串，这样可以方便我们看到，同时调用相同的输出流对象输出时会发生什么。

在代码 36 到 44 行，创建了多个 `jthread` 对象，并以 `syncStreamPrinter` 为入口函数。由于 `jthread` 会在析构时自动 `join`，因此，这里就不需要手动等待线程完成了。`syncStreamPrinter`



中调用 `std::ostream` 将 `cout` 包装成一个 `syncstream`，然后将字符串输出到 `osyncstream` 对象中。

运行这段程序，你可能会看到下图这样的输出。

```
Cout workers:
ABCDEFGHIJKLMN...ABCDEFGHIJKLMN...ABCDEFGHIJKLMN...
opqrstuvwxyz

ABCDEFGHIJKLMN...ABCDEFGHIJKLMN...ABCDEFGHIJKLMN...
pqrstuvwxyz  abcdefghijklmnopqrstuvwxyzabcdefghij
ABCDEFGHIJKLMN...abcdefghijklmnopqrstu
  abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMN...abcdefghijklmnopqrstuABCDEFGHIJKLMN...

abcdefghijklmnopqrstu
  abcdefghijklmnopqrstuvwxyz
SyncStream workers:
ABCDEFGHIJKLMN...abcdefghijklmnopqrstu
ABCDEFGHIJKLMN...abcdefghijklmnopqrstu
ABCDEFGHIJKLMN...abcdefghijklmnopqrstu
ABCDEFGHIJKLMN...abcdefghijklmnopqrstu
ABCDEFGHIJKLMN...abcdefghijklmnopqrstu
ABCDEFGHIJKLMN...abcdefghijklmnopqrstu
ABCDEFGHIJKLMN...abcdefghijklmnopqrstu
ABCDEFGHIJKLMN...abcdefghijklmnopqrstu
ABCDEFGHIJKLMN...abcdefghijklmnopqrstu
ABCDEFGHIJKLMN...abcdefghijklmnopqrstu
ABCDEFGHIJKLMN...abcdefghijklmnopqrstu
```

很明显，并发输出到 `cout` 时出现了无法预期的混乱输出。然而，输出到 `osyncstream` 时没有出现直接使用 `cout` 输出时的混乱。为什么会出现这样的情况呢？

这是因为 `osyncstream` 会包装输出流对象的内部缓冲区，确保通过 `osyncstream` 输出时每次输出都具备原子性，因此也就不会出现错乱的输出了。

## u8string

在 C++11 中，引入了 `std::u16string` 和 `std::u32string`，用于描述 utf-16 和 utf-32 的字符串。它们分别使用 `char16_t` 和 `char32_t` 两个新的字符类型，描述 UTF-16 与 UTF-32 代码点。但奇怪的是，标准始终没有提供对 utf-8 字符描述方式。

好在 C++20 中终于引入了 `u8string`，用于描述 UTF-8 字符串。在引入 `u8string` 的同时，C++20 还定义了一个新的字符类型 `char8_t`，用于描述 UTF-8 的代码点。`u8string` 就是类型为 `char8_t` 的序列。

但是这里有一个问题，为什么 C++20 要引入新的字符类型，而不是用 `char` 呢？

这是因为，C++ 标准定义的 `char` 存在两个比较大的坑。

首先，与其他整数类型不同，标准没有定义 `char` 是 `signed char` 还是 `unsigned char`，有无符号具体由实现决定。

其次，标准只为 `char` 定义了最小长度为 8 位，实际长度也由实现决定（虽然事实标准的确是 8 位），这就导致我们无法严格采用 `char` 来描述 UTF-8 的代码点（UTF-8 代码点固定为 8 位）。

因此，标准必须引入 `char8_t` 这个新类型。不过需要注意的是，`char8_t` 和 `char` 是无法直接隐式转换的，而标准库的很多标准函数都是基于 `char` 这个类型定义的，如果需要转换，必须要强制类型转换。

后面这段代码演示了如何定义 `u8string`，以及如何处理输出与转码。

 复制代码

```
1 #include <fstream>
2 #include <string>
3 #include <iostream>
4 #include <locale>
5 #include <cuchar>
6 #include <cstdint>
7
8 int main() {
9     std::setlocale(LC_ALL, "en_US.utf8");
10
11     // 使用u8创建u8string字面量
12     std::u8string utf8String = u8"你好，这是UTF-8";
13     // 调用size()获取UTF-8代码点数量
14     std::cout << "Processing " << utf8String.size() << " bytes: [ " << std::sho
15     for (char8_t c : utf8String) {
16         std::cout << std::hex << +c << ' ';
17     }
18     std::cout << "]\n";
19
20     // 获取UTF-8代码点序列的起始位置与结束位置
21     const char* utf8Current = reinterpret_cast<char*>(&utf8String[0]);
22     const char* utf8End = reinterpret_cast<char*>(&utf8String[0] + utf8String.s
23     char16_t c16;
24     std::mbstate_t convertState{};
25
26     // 定义UTF-16字符串
27     std::u16string utf16String;
28     // 调用mbrtoc16执行转码
29     while (std::size_t rc = std::mbrtoc16(&c16, utf8Current, utf8End - utf8Curr
30         std::cout << "Next UTF-16 char: " << std::hex
31         << static_cast<int32_t>(c16) << " obtained from ";
```

```

32
33     if (rc == (std::size_t)-3)
34         std::cout << "earlier surrogate pair\n";
35     else if (rc == (std::size_t)-2)
36         break;
37     else if (rc == (std::size_t)-1)
38         break;
39     else {
40         std::cout << std::dec << rc << " bytes [ ";
41         for (std::size_t n = 0; n < rc; ++n)
42             std::cout << std::hex << +static_cast<unsigned char>(utf8Current
43             std::cout << "]\n";
44             utf8Current += rc;
45             utf16String.push_back(c16);
46     }
47 }
48
49 // 输出UTF-8编码字符串
50 std::ofstream u8OutputFile("out.utf8.txt", std::ios::binary);
51 u8OutputFile.write(reinterpret_cast<char*>(utf8String.data()), utf8String.s
52
53 // 输出UTF-16编码字符串
54 std::cout << std::dec << utf16String.size() << std::endl;
55 std::ofstream u16OutputFile("out.utf16.txt", std::ios::binary);
56 u16OutputFile.write(reinterpret_cast<char*>(utf16String.data()), utf16Strin
57
58 return 0;
59 }

```

在代码第 12 行使用了 `u8` 创建 `u8string` 的字面量字符串。

在代码 14 到 18 行，首先输出了 `u8string` 的代码点数量，然后使用十六进制输出了 `u8string` 内的代码点数量。我们处理 UTF-8 字符串的时候，必须要知道 UTF-8 是变长编码，所以一个真正的字符可能包含不同数量的代码点，就像代码中字符串包含 10 个字符，但是包含 20 个代码点。

在代码第 29 到 47 行，我们演示如何调用 `mbrtoc16` 函数，完成 UTF-8 编码向 UTF-16 编码的转换。

mbrtoc16 是 <uchar> 内的编码转换函数，在 C11 标准中添加到头文件 uchar.h（C++11 自然也继承了）。该函数用于将当前语言环境（locale）的多字节编码集转换成 UTF-16 编码，因此如果当前语言环境是 UTF-8 并且当前 C++ 运行时支持，就能将 UTF-8 编码字符串转换成 UTF-16 字符串，这就是为什么代码第 9 行调用了 setlocale 将语言环境设置为 en\_US.utf8。



由于 UTF-8 是变长编码集，因此我们需要多次调用 mbrtoc16 将 UTF-8 的字符逐个转换成 UTF-16 的代码点。

每次调用时，会将第二个参数（类型为 const char\*，这里相当于 UTF-8 字符串的开始位置）开始的特定数量代码点，转换成一个 UTF-16 中对应的代码点，写入到该函数的第一个参数中（类型为 char16\_t\*）。

这里需要注意 mbrtoc16 的返回值 rc，我画了一张表，帮你梳理了 rc 可能的返回值。

返回值	备注
>0	表示当次转换时，成功转换的本地字符编码的字节数（这里可以理解为成功转换的 UTF-8 代码点数量）。可以根据 rc 计算下次调用 mbrtoc16 时待转换字符串指针的新起始位置。
0	表示遇到了 C 风格字符串的结尾 \0，本示例中相当于编码转换结束。
-1	发生编码错误，不会输出任何数据到第一个参数中，并将 errno 设置为 EILSEQ。
-2	后续字符串中的指定的字节数量不足以完整转换成一个 UTF-16 的代码点，不会输出任何数据到第一个参数中。
-3	只会出现在处理一个 UTF-16 字符包含多个 char16_t 代码点的场景中。表示下一个代码点已经被输出到第一个参数中，本次转换不会处理任何输入字符。比如 UTF-16 扩展字符集的代理对（Surrogate Pair）就包含 2 个代码点。



遗憾的是，这种标准库支持的编码转换方式依赖于 C 的 locale。但是，C 的 locale 实现支持完全取决于具体的 C/C++ 运行时环境（也会进一步依赖于操作系统）。

因此，我们的代码虽然可以运行在主流系统与运行时环境中，但不能保证兼容性。

另外，C 的 `locale` 还有 C 标准库经常遇到的多线程环境问题，因为 `setlocale` 是全局的，所以在一个线程中 `setlocale` 对其他线程中行为的影响是未知的。也就是说，`setlocale` 并非线程隔离，也不是线程安全的，所以在多线程程序中使用 C 的 `locale`，我们需要慎之又慎。

因此，在大部分场景下，我还是建议你使用 `iconv` 之类的第三方编码转换库执行编码转换。希望 C++ 能在日后标准中，进一步脱离 C 的 `locale`，然后彻底解决编码问题吧。

此外，与 C++11 加入的 `u16string` 一样，C++20 的 `u8string` 也缺乏输入输出流的直接支持（根本原因是 C++17 废弃了 `codecvt` 的具体实现）。因此，在代码 50 到 56 行中，我们不得不使用二进制的方式将其输出到文件中。

不得不说，C++ 针对语言编码的支持，依然任重而道远！我们期待着在后续演进标准中逐步解决这些问题。

## 总结

我们在这一讲中，进一步补充了 C++20 标准中重要的库变更，包括它们的用法和注意事项。

`jthread` 是自 C++11 之后对标准的并发编程的一次重要补充，它支持了安全的默认行为，具备线程中断机制，可以根据实际情况调整具体行为，在确保安全的前提下支持灵活的调整，在大多数场景中是更符合实际需求的设计。

`source location` 对输出代码的准确位置提供了有力支撑，同时解决了模板函数中长久以来存在的问题——无法输出准确代码执行位置。这为调试复杂程序和输出信息提供了新工具。

另外，我们还讨论了 `u8string`，以及 C++ 针对语言编码支持的问题。标准库支持的编码转换方式仍然依赖于 C 的 `locale`，我们期待着在后续演进标准中逐步解决这些问题。

## 思考题

请你结合 `char` 的宽度问题，思考一下 `u8string` 转换示例代码中可能还会出现什么兼容性问题。

欢迎说出你的看法，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 17 | Bit library（二）：如何利用新bit操作库释放编程生产力？

下一篇 19 | 其他重要标准库特性实战：利用日历应用熟悉新特性

## 精选留言 (1)

写留言



peter

2023-03-04 来自北京

请教老师几个问题：

Q1: pthread\_create属于thread类吗？

我从网上搜到了一个安卓代码，有C++代码。线程部分，用的是pthread\_create、pthread\_join、pthread\_exit等函数。能正常工作，但不太了解这套东西。这套东西属于C++类库中的thread类吗？还是类似于系统调用一类的API？

Q2: stop\_requested设置为true，但线程本身不退出，会发生什么？

Q3: 一个线程发送request\_stop成功后，什么时候其他线程才能发送？文中提到“一个线程发送请求成功后，其他线程调用 request\_token 会失败，但不会引发异常”，发送成功后，按道理其他线程就可以发送了啊；不能发送的话，何时才能发送？

作者回复: 1. pthread\_create不属于thread类。Pthread是POSIX的Thread标准，是支持POSIX标准的系统提供的线程库，并不属于C++标准的一部分，而是属于实现部分（也就是C++标准库可能在某些支持POSIX标准的系统中使用pthread实现thread类）。

2. stop\_requested设置为true，线程本身不退出不会发生什么，stop\_requested只是一种线程安全的中止线程的通知手段而已。

3. 一个线程向一个jthread对象发送request\_stop成功后其他的线程就不会发送成功了，因为标准中规定的就是有且只有一个线程可以发送request\_stop成功，因为发送成功后stop\_token就设置成true了，其他的线程再发送也就没啥意义了，你可以把request\_token发送成功理解为“成功将jthread的stop\_token从false修改成true”，如果stop\_token本来就是true了，那肯定就不用修改了，就是发送失败。

