C++20 jthread

C++20 jthread

C ++ 20 中新引入的的新线程（jthread）功能修复了 std::thread 不是 RAII 类型的设计缺陷，并且增加了能够主动取消或停止线程执行的新特性。它基本上是一个包装器，它为线程带来了两个新功能：**默认情况下，**它们可以**协同中断**并**加入**。std::jthread 对象包含 std::thread 一个成员，提供完全相同的公共函数，这些函数只是向下传递调用。

它拥有同 std::thread 的行为外，主要增加了以下两个功能：

- std::jthread 对象被 destruct 时，会自动调用 join，等待其所表示的执行流结束
- 支持外部请求中止（通过 get_stop_source、get_stop_token 和 request_stop ）

jthread 声明

成员类型

成员类型	定义
id	std::thread::id
native_handle_type (可选)	std::thread::native_handle_type

成员函数

(构造函数)	创建新的 jthread 对象(公开成员函数)
(析构函数)	若 joinable() 为 true ，则调用 request_stop() 然后 join() ；不论如何都会销毁 jthread 对象。(公开成员函数)
operator=	移动 jthread 对象(公开成员函数)
观察器	
joinable	检查线程是否可合并，即潜在地运行于平行环境中(公开成员函数)
get_id	返回线程的 id (公开成员函数)
native_handle	返回底层实现定义的线程句柄(公开成员函数)
hardware_concurrency [静态]	返回实现支持的并发线程数(公开静态成员函数)
操作	
join	等待线程完成其执行(公开成员函数)
detach	容许线程从线程句柄独立开来执行(公开成员函数)
swap	交换二个 `jthread` 对象(公开成员函数)
停止记号处理	
get_stop_source	返回与线程的停止状态关联的 stop_source 对象(公开成员函数)
get_stop_token	返回与线程的共享停止状态关联的 stop_token(公开成员函数)
request_stop	请求执行经由线程的共享停止状态停止(公开成员函数)

非成员函数

swap(std::jthread)(C++20)	特化 std::swap 算法(函数)

std::jthread 修复了 std::thread 不是 RAII 类型的设计缺陷

当使用 std::thread 时，需要在线程对象的生命周期结束时调用 join() 或 detach() 函数。如果这两个函数都没有被调用，析构函数将立即导致程序异常终止并产生 core dump（程序异常终止或崩溃时，操作系统将程序的内存状态保存到一个特殊文件中，该文件称为核心转储文件）。下面是示例代码：

C/C++

```
void FuncWithoutJoinOrDetach() {
  std::thread t{task, task_args};
  // 没有调用 t.join() 或 t.detach()
}  // t 的生命周期结束时将调用 std::terminate()，异常终止程序
```

即使我们调用了 join() 来等待正在运行的线程结束，仍然可能出现异常安全的问题：

C/C++

```
void FuncWithExceptionSafety() {
  // 启动线程执行 task
  std::thread t{task, task_args};
  ... // 中间可能会发生异常，在异常时调用 std::terminate()
  // 等待 task 执行结束
  t.join();
}
```

因此，需要使用 try-catch 来进行异常处理，确保异常发生后 join() 函数也能被正常调用：

C/C++

```
void FuncWithoutExceptionSafety() {
  // 启动线程执行 task
  std::thread t{task, task_args};
  try {
    ...
  }
  catch (...) {
    // 阻塞等待 t 运行结束
    t.join();
    // 重新抛出异常
    throw;
  }
  t.join();
}
```

jthread 包装了 std::thread，在析构函数中调用 join()函数(jthread 的 j 是 joining 的缩写)，修复了 std::thread 不是 RAII 类型的缺陷：

C/C++

```cpp
#include <iostream>
 #include <thread>
 using namespace std;

 int main(int argc, char* argv[])
 {
     std::cout << std::boolalpha << std::endl;

     std::thread thr{
         [] { cout << "joinable std::thread<<" << std::endl; }
     };

     std::cout << thr.joinable() << std::endl;
 }
```

上述程序会报错，因为上述程序在 main 函数退出资源就要被释放，但是线程对象的析构函数被调用时，线程并没有被 join() 或 detach()。由于没有调用 join() 或 detach()，程序会抛出 std::terminate() 异常并终止。

在 c++11 中，为了解决这个问题，可以在主函数结束之前，调用 join() 或 detach() 来管理线程的生命周期。添加 thr.join() 或 thr.detach()，将线程加入主线程或分离线程，确保线程的正确结束。

C/C++

```cpp
#include <iostream>
#include <thread>
using namespace std;

int main(int argc, char* argv[])
{
    std::cout << std::boolalpha << std::endl;

    std::thread thr{
        [] { cout << "joinable std::thread<<" << std::endl; }
    };

    std::cout << thr.joinable() << std::endl;

    thr.join(); // 或者使用 thr.detach()

    return 0;
}
```

上述程序可以正常运行，但是和 new delete 一样，每次都需要手动析构很可能造成资源泄露或者其他异常问题，c++20 提供的 jthread 利用 RAII 特性，可以实现线程的自动分离。

C/C++

```cpp
#include <iostream>
#include <thread>
using namespace std;

int main(int argc, char* argv[])
{
    std::cout << std::boolalpha << std::endl;

    std::jthread thr{
        [] { cout << "joinable std::thread" << std::endl; }
    };

    std::cout << thr.joinable() << std::endl;

    return 0;
}
```

std::jthread 增加了能够主动取消或停止线程执行的新特性

调用线程的 join()函数后可能需要等待很长时间，甚至是永远等待。由于线程不像进程允许我们主动发送 kill 信号终止它，已经启动的线程只能自己结束运行或结束整个程序来结束该线程。

因此，std::jthread 除了提供 std::stop_token 能够主动取消或停止正在执行的线程，还增加了 std::stop_callback 允许在停止线程操作时调用一组回调函数。

例如如下代码将允许当前线程调用 request_stop()之后，主动终止线程：

C/C++

```cpp
#include <iostream>
#include <thread>
#include <chrono>

using namespace std;

int main(int argc, char *argv[]) {
    // jthread负责传入stop_token
    auto f = [](const stop_token &st) {
        // jthread并不会强制停止线程，需要我们依据stop_token的状态来进行取消/停止操作
        while (!st.stop_requested()) {
            cout << "other: " << this_thread::get_id() << "\n";
            this_thread::sleep_for(1s);
        }
        cout << "other thread stopped!\n";
    };
    jthread jth(f);

    cout << "main: " << this_thread::get_id() << "\n";
    this_thread::sleep_for(5s);

    // 请求停止线程，对应的stop_token的stop_requested()函数返回true（除了手动调用外，jthread销毁时也会自动调用该函数）
    jth.request_stop();
    // 我们无需在jthread上调用join()，它在销毁时自动join
}
```

它不会立即停止执行该线程，也不会停止执行。注意，它说请求停止，而不是坚持或强制。所以我们（从线程外部）只能请求停止，并且该线程本身具有最终发言权。这就是它可以**协作**中断的原因。

如果不没有手动调用 `jth.request_stop();` 也是可以的，因为 jthread 是 RAII 类型的。