課程 3.1：RAII 與執行緒管理

第三階段：執行緒生命週期管理

課程 3.1：RAII 與執行緒管理

引言

RAII（Resource Acquisition Is Initialization）是 C++ 最重要的資源管理慣用法。將它應用於執行緒管理，可以避免忘記 join 或 detach 導致的程式崩潰。

一、問題：例外導致的資源洩漏

```
#include <iostream>
#include <thread>

void riskyFunction() {
std::thread t([]() {
std::cout << "工作中" << std::endl;
});
// 如果這裡拋出例外...
throw std::runtime_error("發生錯誤！");
t.join(); // 永遠不會執行！
} // t 解構時仍是 joinable → std::terminate()

int main() {
try {
riskyFunction();
} catch (...) {
std::cout << "捕獲例外" << std::endl;
}
return 0;
}
```

這段程式會崩潰，因為例外導致 `join()` 被跳過。

二、RAII 的核心概念

```
┌──────────────────────────────────────────────────────────────┐
│ RAII 原則 │
├──────────────────────────────────────────────────────────────┤
│ │
│ 建構函式：獲取資源 │
│ 解構函式：釋放資源 │
│ │
│ 優點： │
│ • 無論正常返回或例外，解構函式都會被呼叫 │
│ • 資源自動管理，不會忘記釋放 │
│ • 程式碼更簡潔、更安全 │
│ │
└──────────────────────────────────────────────────────────────┘
```

三、簡單的 RAII 執行緒包裝

```
#include <iostream>
#include <thread>

class ThreadGuard {
std::thread& t;
public:
explicit ThreadGuard(std::thread& thread) : t(thread) {}
~ThreadGuard() {
if (t.joinable()) {
t.join();
}
}
// 禁止複製
ThreadGuard(const ThreadGuard&) = delete;
ThreadGuard& operator=(const ThreadGuard&) = delete;
};

int main() {
std::thread t([]() {
std::cout << "工作中" << std::endl;
});
```

```
ThreadGuard guard(t); // 保證 t 會被 join
// 即使這裡拋出例外，guard 的解構函式仍會執行
// throw std::runtime_error("測試");
return 0;
} // guard 解構 → 自動 join
```

四、擁有執行緒的 RAII 類別

上面的 ThreadGuard 只持有引用。更好的設計是擁有執行緒：

```
#include <iostream>
#include <thread>

class ScopedThread {
std::thread t;
public:
explicit ScopedThread(std::thread thread)
: t(std::move(thread))
{
if (!t.joinable()) {
throw std::logic_error("No thread");
}
}
~ScopedThread() {
t.join(); // 一定是 joinable，不用檢查
}
// 禁止複製
ScopedThread(const ScopedThread&) = delete;
ScopedThread& operator=(const ScopedThread&) = delete;
};

int main() {
ScopedThread st(std::thread([]() {
std::cout << "安全的執行緒" << std::endl;
}));
// 自動管理，不需要手動 join
return 0;
}
```

五、選擇 join 或 detach 的 RAII

```
#include <iostream>
#include <thread>

class FlexibleThread {
public:
enum class Action { join, detach };
private:
std::thread t;
Action action;
public:
FlexibleThread(std::thread thread, Action a)
: t(std::move(thread)), action(a) {}
~FlexibleThread() {
if (t.joinable()) {
if (action == Action::join) {
t.join();
} else {
t.detach();
}
}
}
FlexibleThread(const FlexibleThread&) = delete;
FlexibleThread& operator=(const FlexibleThread&) = delete;
};

int main() {
FlexibleThread ft1(
std::thread([]() { std::cout << "Join 我" << std::endl; }),
FlexibleThread::Action::join
);
FlexibleThread ft2(
std::thread([]() { std::cout << "Detach 我" << std::endl; }),
FlexibleThread::Action::detach
);
std::this_thread::sleep_for(std::chrono::milliseconds(100));
```

```
return 0;
}
```

六、RAII vs 手動管理對比

```
// ❌ 手動管理：容易出錯
void manual() {
std::thread t(work);
if (condition1) {
t.join();
return; // 要記得 join
}
if (condition2) {
t.join();
throw error; // 要記得 join
}
t.join(); // 正常路徑也要 join
}

// ✅ RAII：自動管理
void automatic() {
ScopedThread st(std::thread(work));
if (condition1) {
return; // 自動 join
}
if (condition2) {
throw error; // 自動 join
}
// 自動 join
}
```

七、本課重點回顧

1. **RAII** 利用解構函式自動釋放資源
2. 解構函式在正常返回和例外時都會執行
3. `ThreadGuard` 持有引用，保證執行緒被 join

4. `ScopedThread` 擁有執行緒，更安全
5. RAII 讓程式碼更簡潔，避免遺漏 join/detach
6. C++20 的 `std::jthread` 就是標準化的 RAII 執行緒

下一課預告
在 **課程 3.2：執行緒守衛類別設計** 中，我們將：
- 完善 ThreadGuard 的設計
- 處理更多邊界情況
- 加入移動語意支援

準備好繼續嗎？

課程 3.2：執行緒守衛類別設計

第三階段：執行緒生命週期管理
課程 3.2：執行緒守衛類別設計

引言
上一課介紹了基本的 RAII 執行緒包裝。本課將設計一個更完善、更實用的執行緒守衛類別。

一、設計目標

```
┌─────────────────────────────────────────────────────────┐
│ 理想的執行緒守衛應具備 │
├─────────────────────────────────────────────────────────┤
│ │
│ ✓ 自動在解構時 join 或 detach │
│ ✓ 支援移動語意 │
│ ✓ 禁止複製 │
│ ✓ 可以手動提前釋放 │
│ ✓ 可以取得底層執行緒的資訊 │
│ │
└─────────────────────────────────────────────────────────┘
```

二、完整的 JoiningThread 類別

```
#include <iostream>
#include <thread>
#include <utility>

class JoiningThread {
std::thread t;
public:
// 預設建構
JoiningThread() noexcept = default;
// 從 std::thread 移動建構
explicit JoiningThread(std::thread thread) noexcept
: t(std::move(thread)) {}
// 直接建構執行緒（完美轉發）
template<typename Func, typename... Args>
explicit JoiningThread(Func&& f, Args&&... args)
: t(std::forward<Func>(f), std::forward<Args>(args)...) {}
// 移動建構
JoiningThread(JoiningThread&& other) noexcept
: t(std::move(other.t)) {}
// 移動賦值
JoiningThread& operator=(JoiningThread&& other) noexcept {
if (this != &other) {
if (t.joinable()) {
t.join(); // 先處理當前執行緒
}
t = std::move(other.t);
}
return *this;
}
// 禁止複製
JoiningThread(const JoiningThread&) = delete;
JoiningThread& operator=(const JoiningThread&) = delete;
// 解構：自動 join
~JoiningThread() {
if (t.joinable()) {
t.join();
}
}
```

```
// 工具方法
bool joinable() const noexcept { return t.joinable(); }
std::thread::id get_id() const noexcept { return t.get_id(); }
void join() {
t.join();
}
void detach() {
t.detach();
}
std::thread& get() noexcept { return t; }
};
```

三、使用範例

```
#include <iostream>
#include <vector>

int main() {
// 方式一：直接建構
JoiningThread jt1([]() {
std::cout << "執行緒 1" << std::endl;
});
// 方式二：從 std::thread 移動
std::thread t([]() {
std::cout << "執行緒 2" << std::endl;
});
JoiningThread jt2(std::move(t));
// 方式三：帶參數
JoiningThread jt3([](int x) {
std::cout << "執行緒 3: " << x << std::endl;
}, 42);
// 不需要手動 join！
return 0;
}
```

輸出：

```
執行緒 1
執行緒 2
執行緒 3: 42
```

四、在容器中使用

```
#include <iostream>
#include <vector>

int main() {
std::vector<JoiningThread> threads;
for (int i = 0; i < 5; ++i) {
threads.emplace_back([i]() {
std::cout << "Worker " << i << std::endl;
});
}
std::cout << "所有執行緒已建立" << std::endl;
// 離開作用域時，vector 解構
// 每個 JoiningThread 解構時自動 join
return 0;
}
```

五、DetachingThread 變體

如果預設行為是 detach：

```
class DetachingThread {
std::thread t;
public:
template<typename Func, typename... Args>
explicit DetachingThread(Func&& f, Args&&... args)
: t(std::forward<Func>(f), std::forward<Args>(args)...) {}
DetachingThread(DetachingThread&&) = default;
DetachingThread& operator=(DetachingThread&&) = default;
DetachingThread(const DetachingThread&) = delete;
DetachingThread& operator=(const DetachingThread&) = delete;
```

```
~DetachingThread() {
if (t.joinable()) {
t.detach();
}
}
};
```

六、策略模式：可選擇的行為

```
#include <iostream>
#include <thread>

enum class ThreadAction { Join, Detach };

template<ThreadAction action>
class ManagedThread {
std::thread t;
public:
template<typename Func, typename... Args>
explicit ManagedThread(Func&& f, Args&&... args)
: t(std::forward<Func>(f), std::forward<Args>(args)...) {}
ManagedThread(ManagedThread&&) = default;
ManagedThread& operator=(ManagedThread&&) = default;
ManagedThread(const ManagedThread&) = delete;
ManagedThread& operator=(const ManagedThread&) = delete;
~ManagedThread() {
if (t.joinable()) {
if constexpr (action == ThreadAction::Join) {
t.join();
} else {
t.detach();
}
}
}
};

// 型別別名
using AutoJoinThread = ManagedThread<ThreadAction::Join>;
```

```
using AutoDetachThread = ManagedThread<ThreadAction::Detach>;

int main() {
AutoJoinThread jt([]() {
std::cout << "會被 join" << std::endl;
});
return 0;
}
```

七、與 std::jthread 的比較

```
┌──────────────────────────────────────────────────────────────┐
│ 自訂 JoiningThread vs std::jthread │
├──────────────────────────────────────────────────────────────┤
│ │
│ JoiningThread (自訂) std::jthread (C++20) │
│ ──────────────────── ──────────────────── │
│ • 可用於 C++11/14/17 • 需要 C++20 │
│ • 可自訂行為 • 標準化，跨平台一致 │
│ • 需要自己維護 • 內建支援取消機制 │
│ • 經過充分測試 │
│ │
│ 建議：C++20 可用時，優先使用 std::jthread │
│ │
└──────────────────────────────────────────────────────────────┘
```

八、本課重點回顧

1. 完善的執行緒守衛需要支援**移動語意**
2. 移動賦值前要先處理當前的 joinable 執行緒
3. 使用**完美轉發**支援直接建構執行緒
4. 可用模板參數在編譯期選擇 join 或 detach 行為
5. 執行緒守衛可以安全地放入容器
6. C++20 有標準的 `std::jthread` ，功能更完整

下一課預告

在 **課程 3.3：std::jthread (C++20)** 中，我們將學習：

- 標準化的自動 join 執行緒
- 內建的取消機制（stop_token）
- 為什麼它比自訂類別更好

準備好繼續嗎？

C++ explicit 關鍵字複習

一、explicit 的作用

`explicit` **用於禁止隱式轉換，強制要求明確呼叫建構函式。**

二、沒有 explicit 的問題

```
#include <iostream>

class MyInt {
int value;
public:
MyInt(int v) : value(v) {} // 沒有 explicit
int get() const { return value; }
};

void printMyInt(MyInt m) {
std::cout << m.get() << std::endl;
}

int main() {
MyInt a = 42; // 隱式轉換：int → MyInt
MyInt b = 3.14; // 隱式轉換：double → int → MyInt（可能非預期！）
printMyInt(100); // 隱式轉換：int → MyInt
return 0;
}
```

這些隱式轉換可能導致意外的行為。

三、加上 explicit

```
#include <iostream>

class MyInt {
int value;
public:
explicit MyInt(int v) : value(v) {} // 禁止隱式轉換
int get() const { return value; }
};

void printMyInt(MyInt m) {
std::cout << m.get() << std::endl;
}

int main() {
// MyInt a = 42; // 編譯錯誤！
// MyInt b = 3.14; // 編譯錯誤！
// printMyInt(100); // 編譯錯誤！
MyInt a(42); // OK：明確呼叫建構函式
MyInt b{42}; // OK：直接初始化
MyInt c = MyInt(42); // OK：明確轉換
printMyInt(MyInt(100)); // OK：明確轉換
return 0;
}
```

四、圖解對比

```
┌──────────────────────────────────────────────────────────┐
│ 沒有 explicit vs 有 explicit │
├──────────────────────────────────────────────────────────┤
│ │
│ 沒有 explicit： │
│ MyInt x = 42; ✓ 允許（隱式轉換） │
│ func(42); ✓ 允許（隱式轉換） │
│ │
│ 有 explicit： │
│ MyInt x = 42; ✗ 編譯錯誤 │
```

```
| MyInt x(42); ✓ 允許（明確建構） |
| MyInt x{42}; ✓ 允許（直接初始化） |
| func(MyInt(42)); ✓ 允許（明確轉換） |
| |
└──────────────────────────────────────────────────────────────┘
```

五、實際案例：為什麼 std::thread 用 explicit

```
class thread {
public:
explicit thread(Function&& f, Args&&... args);
};

int main() {
void work();
// std::thread t = work; // 編譯錯誤！防止意外建立執行緒
std::thread t(work); // OK：明確表示要建立執行緒
t.join();
return 0;
}
```

建立執行緒是**重大操作**，不應該意外發生。

六、何時使用 explicit

情況	是否用 explicit
單參數建構函式	通常要用
轉換建構函式	通常要用
多參數建構函式	可選（C++11 前不需要）
複製/移動建構函式	不用

經驗法則：除非你明確想要隱式轉換，否則單參數建構函式都加 `explicit` 。

七、C++11 擴展：多參數也可用

```
class Point {
public:
explicit Point(int x, int y) : x(x), y(y) {}
int x, y;
};

int main() {
// Point p = {1, 2}; // 編譯錯誤！
Point p{1, 2}; // OK
Point q(1, 2); // OK
return 0;
}
```

八、explicit 用於轉換運算子（C++11）

```
class Boolean {
bool value;
public:
Boolean(bool v) : value(v) {}
explicit operator bool() const { // 明確轉換
return value;
}
};

int main() {
Boolean b(true);
// int x = b; // 編譯錯誤！
if (b) {} // OK：條件判斷允許明確轉換
bool y = static_cast<bool>(b); // OK
return 0;
}
```

總結

重點	說明
目的	禁止隱式型別轉換
適用	建構函式、轉換運算子
效果	必須明確呼叫，不能偷偷轉換
原則	**單參數建構函式預設加上 explicit**

還有其他問題嗎？或者繼續下一課？

C++ noexcept 關鍵字複習

一、noexcept 的作用

`noexcept` **表示函式保證不會拋出例外。這是對編譯器和使用者的承諾。**

二、基本語法

```
void func1() noexcept; // 保證不拋出例外
void func2() noexcept(true); // 同上
void func3() noexcept(false); // 可能拋出例外（等同不寫）
void func4(); // 可能拋出例外
```

三、為什麼需要 noexcept？

```
┌──────────────────────────────────────────────────────────┐
│ noexcept 的三大價值 │
├──────────────────────────────────────────────────────────┤
│ │
│ 1. 效能優化 │
│ 編譯器知道不會有例外，可以生成更高效的程式碼 │
│ │
```

```
│ 2. 啟用移動語意 │
│ 標準容器只在移動操作是 noexcept 時才會使用移動 │
│ │
│ 3. 程式碼文件化 │
│ 明確告訴使用者這個函式不會拋出例外 │
│ │
└──────────────────────────────────────────────────────────────┘
```

四、效能優化範例

```
#include <iostream>
#include <vector>

class Widget {
public:
// 沒有 noexcept：vector 擴容時會用複製
Widget(Widget&& other) {
std::cout << "Move (可能拋例外)" << std::endl;
}
};

class BetterWidget {
public:
// 有 noexcept：vector 擴容時會用移動
BetterWidget(BetterWidget&& other) noexcept {
std::cout << "Move (保證安全)" << std::endl;
}
};
```

`std::vector` 擴容時，只有當移動建構是 `noexcept` 時，才會使用移動而非複製。

五、違反 noexcept 的後果

如果 `noexcept` 函式拋出例外，程式會呼叫 `std::terminate()`：

```
#include <iostream>

void dangerous() noexcept {
```

```
throw std::runtime_error("Oops!"); // 違反承諾！
}

int main() {
try {
dangerous();
} catch (...) {
// 不會執行到這裡！
std::cout << "Caught" << std::endl;
}
return 0;
}
```

輸出：

```
terminate called after throwing an instance of 'std::runtime_error'
Aborted
```

六、條件式 noexcept

根據條件決定是否 noexcept：

```
#include <type_traits>

template<typename T>
void wrapper(T& obj) noexcept(noexcept(obj.doSomething())) {
obj.doSomething();
}

// 如果 T::doSomething() 是 noexcept，則 wrapper 也是 noexcept
```

七、noexcept 運算子

`noexcept(expr)` 可以檢查表達式是否為 noexcept：

```
#include <iostream>
```

```
void mayThrow() {}
void noThrow() noexcept {}

int main() {
std::cout << std::boolalpha;
std::cout << "mayThrow: " << noexcept(mayThrow()) << std::endl;
std::cout << "noThrow: " << noexcept(noThrow()) << std::endl;
return 0;
}
```

輸出：

```
mayThrow: false
noThrow: true
```

八、何時使用 noexcept

函式類型	是否用 noexcept
移動建構函式	**強烈建議**
移動賦值運算子	**強烈建議**
解構函式	預設就是 noexcept
swap 函式	強烈建議
簡單的 getter	可以加
可能失敗的操作	不要加

九、標準函式庫的期望

```
class MyClass {
public:
// 標準容器期望這些是 noexcept
```

```
MyClass(MyClass&& other) noexcept;
MyClass& operator=(MyClass&& other) noexcept;
~MyClass() noexcept; // 解構函式預設就是 noexcept
friend void swap(MyClass& a, MyClass& b) noexcept;
};
```

十、實際案例：JoiningThread

回到我們的執行緒守衛類別：

```
class JoiningThread {
std::thread t;
public:
// 預設建構：不會拋例外
JoiningThread() noexcept = default;
// 移動建構：std::thread 的移動是 noexcept
JoiningThread(JoiningThread&& other) noexcept
: t(std::move(other.t)) {}
// 移動賦值：noexcept
JoiningThread& operator=(JoiningThread&& other) noexcept {
if (t.joinable()) t.join();
t = std::move(other.t);
return *this;
}
// 這些簡單操作也是 noexcept
bool joinable() const noexcept { return t.joinable(); }
std::thread::id get_id() const noexcept { return t.get_id(); }
};
```

十一、noexcept vs throw()（舊式）

```
// C++11 之前的舊語法（已棄用）
void oldStyle() throw();

// C++11 之後的新語法
void newStyle() noexcept;
```

永遠使用 `noexcept`，不要用 `throw()`。

總結

重點	說明
意義	承諾函式不會拋出例外
違反後果	程式呼叫 `std::terminate()` 終止
效能影響	讓編譯器生成更高效的程式碼
關鍵用途	移動建構/賦值必須加，否則容器不會使用移動
檢查語法	`noexcept(expr)` 回傳 bool

還有其他問題嗎？或者繼續下一課？

課程 3.3：std::jthread (C++20)

第三階段：執行緒生命週期管理

課程 3.3：std::jthread (C++20)

引言

C++20 引入了 `std::jthread`（joining thread），它是 `std::thread` 的改良版本，**內建 RAII 自動 join 和協作式取消機制**。

一、std::jthread vs std::thread

```
┌──────────────────────────────────────────────────────────────┐
│ std::thread vs std::jthread │
├──────────────────────────────────────────────────────────────┤
│ │
```

```
│ std::thread std::jthread │
│ ──────────── ──────────── │
│ • 必須手動 join/detach • 解構時自動 join │
│ • 忘記會導致 terminate • 不會忘記 │
│ • 無取消機制 • 內建 stop_token 取消機制 │
│ • C++11 起可用 • C++20 起可用 │
│ │
└──────────────────────────────────────────────────────────────┘
```

二、基本用法

```
#include <iostream>
#include <thread>

int main() {
std::jthread jt([]() {
std::cout << "Hello from jthread!" << std::endl;
});
// 不需要呼叫 join()！
// 離開作用域時自動 join
return 0;
}
```

編譯（需要 C++20）：

```
g++ -std=c++20 -pthread -o jthread_demo jthread_demo.cpp
```

三、自動 join 的安全性

即使發生例外，也能正確處理：

```
#include <iostream>
#include <thread>
#include <stdexcept>

void riskyFunction() {
```

```
std::jthread jt([]() {
std::this_thread::sleep_for(std::chrono::milliseconds(100));
std::cout << "執行緒完成" << std::endl;
});
throw std::runtime_error("發生錯誤！");
// 不需要擔心！jt 解構時會自動 join
}

int main() {
try {
riskyFunction();
} catch (const std::exception& e) {
std::cout << "捕獲: " << e.what() << std::endl;
}
return 0;
}
```

輸出：

```
執行緒完成
捕獲: 發生錯誤！
```

四、stop_token 取消機制

`std::jthread` **最強大的功能是內建的協作式取消**：

```
#include <iostream>
#include <thread>

int main() {
std::jthread jt([](std::stop_token stoken) {
int count = 0;
while (!stoken.stop_requested()) {
std::cout << "工作中... " << ++count << std::endl;
std::this_thread::sleep_for(std::chrono::milliseconds(200));
}
std::cout << "收到停止請求，結束" << std::endl;
});
```

```
// 讓執行緒跑一下
std::this_thread::sleep_for(std::chrono::seconds(1));
// 請求停止
jt.request_stop();
// jt 解構時會等待執行緒結束
return 0;
}
```

輸出：

```
工作中... 1
工作中... 2
工作中... 3
工作中... 4
工作中... 5
收到停止請求，結束
```

五、stop_token 詳解

```
┌──────────────────────────────────────────────────────────┐
│ stop_token 機制 │
├──────────────────────────────────────────────────────────┤
│ │
│ std::stop_source → 發出停止請求的來源 │
│ std::stop_token → 檢查是否有停止請求 │
│ std::stop_callback → 停止時自動執行的回調 │
│ │
│ jthread 內部自動管理 stop_source │
│ 執行緒函式可接收 stop_token 作為第一個參數 │
│ │
└──────────────────────────────────────────────────────────┘
```

六、手動取得 stop_source 和 stop_token

```
#include <iostream>
```

```
#include <thread>

int main() {
std::jthread jt([](std::stop_token stoken) {
while (!stoken.stop_requested()) {
std::cout << "Running..." << std::endl;
std::this_thread::sleep_for(std::chrono::milliseconds(300));
}
});
// 取得 stop_source
std::stop_source& source = jt.get_stop_source();
// 取得 stop_token
std::stop_token token = jt.get_stop_token();
std::this_thread::sleep_for(std::chrono::seconds(1));
// 兩種方式都可以請求停止
// source.request_stop();
jt.request_stop();
return 0;
}
```

七、stop_callback 自動回調

當停止被請求時，自動執行回調：

```
#include <iostream>
#include <thread>

int main() {
std::jthread jt([](std::stop_token stoken) {
// 註冊停止時的回調
std::stop_callback callback(stoken, []() {
std::cout << "停止回調被觸發！" << std::endl;
});
while (!stoken.stop_requested()) {
std::cout << "工作中..." << std::endl;
std::this_thread::sleep_for(std::chrono::milliseconds(300));
}
std::cout << "執行緒結束" << std::endl;
});
```

```
std::this_thread::sleep_for(std::chrono::seconds(1));
jt.request_stop();
return 0;
}
```

輸出：

```
工作中...
工作中...
工作中...
停止回調被觸發！
執行緒結束
```

八、jthread 的完整介面

```
class jthread {
public:
// 建構
jthread() noexcept;
template<typename F, typename... Args>
explicit jthread(F&& f, Args&&... args);
// 解構：自動 request_stop() + join()
~jthread();
// 移動（不可複製）
jthread(jthread&&) noexcept;
jthread& operator=(jthread&&) noexcept;
// 停止機制
stop_source get_stop_source() noexcept;
stop_token get_stop_token() const noexcept;
bool request_stop() noexcept;
// 與 std::thread 相同的介面
bool joinable() const noexcept;
void join();
void detach();
id get_id() const noexcept;
native_handle_type native_handle();
static unsigned int hardware_concurrency() noexcept;
```

```
};
```

九、何時用 jthread vs thread

情況	選擇
C++20 可用	優先使用 `jthread`
需要取消機制	使用 `jthread`
需要 detach	使用 `thread`
舊專案相容性	使用 `thread`

十、完整範例：可取消的工作執行緒

```
#include <iostream>
#include <thread>
#include <vector>

void worker(std::stop_token stoken, int id) {
std::cout << "Worker " << id << " 啟動" << std::endl;
int count = 0;
while (!stoken.stop_requested()) {
++count;
std::this_thread::sleep_for(std::chrono::milliseconds(100));
}
std::cout << "Worker " << id << " 結束，執行了 " << count << " 次" << std::endl;
}

int main() {
std::vector<std::jthread> workers;
// 建立 3 個工作執行緒
for (int i = 0; i < 3; ++i) {
workers.emplace_back(worker, i);
}
```

```
std::cout << "主執行緒等待 1 秒..." << std::endl;
std::this_thread::sleep_for(std::chrono::seconds(1));
std::cout << "請求所有執行緒停止" << std::endl;
for (auto& w : workers) {
w.request_stop();
}
// vector 解構時，每個 jthread 自動 join
return 0;
}
```

十一、本課重點回顧

1. `std::jthread` 是 C++20 新增的執行緒類別
2. **自動 join**：解構時自動等待執行緒結束
3. **stop_token**：內建協作式取消機制
4. 執行緒函式可接收 `std::stop_token` 作為第一個參數
5. `request_stop()` 請求執行緒停止
6. `stop_callback` 可在停止時自動執行回調
7. 優先使用 `jthread` ，除非需要 detach 或舊版相容

下一課預告

在 **課程 3.4：執行緒例外處理** 中，我們將學習：

- 執行緒中的例外如何處理
- 例外如何在執行緒間傳遞
- 安全的錯誤處理模式

準備好繼續嗎？

課程 3.4：執行緒例外處理

第三階段：執行緒生命週期管理

課程 3.4：執行緒例外處理

引言

執行緒中拋出的例外**不會自動**傳遞到建立它的執行緒。如果例外未被捕獲，程式會直接終止。
本課學習如何正確處理執行緒中的例外。

一、問題：例外不會跨執行緒傳遞

```
#include <iostream>
#include <thread>
#include <stdexcept>

void worker() {
throw std::runtime_error("執行緒內的錯誤！");
}

int main() {
std::thread t(worker);
try {
t.join();
} catch (const std::exception& e) {
// 這裡捕獲不到！
std::cout << "捕獲: " << e.what() << std::endl;
}
return 0;
}
```

輸出：

```
terminate called after throwing an instance of 'std::runtime_error'
Aborted
```

例外在執行緒內未被捕獲，程式直接終止。

二、解決方案一：執行緒內部捕獲

最簡單的方法是在執行緒函式內部處理例外：

```
#include <iostream>
#include <thread>
#include <stdexcept>

void worker() {
```

```
try {
throw std::runtime_error("執行緒內的錯誤！");
} catch (const std::exception& e) {
std::cout << "執行緒內捕獲: " << e.what() << std::endl;
}
}

int main() {
std::thread t(worker);
t.join();
std::cout << "程式正常結束" << std::endl;
return 0;
}
```

輸出：

```
執行緒內捕獲: 執行緒內的錯誤！
程式正常結束
```

三、解決方案二：使用 std::exception_ptr 傳遞例外

```
#include <iostream>
#include <thread>
#include <stdexcept>
#include <exception>

std::exception_ptr threadException = nullptr;

void worker() {
try {
throw std::runtime_error("執行緒內的錯誤！");
} catch (...) {
// 捕獲並儲存例外
threadException = std::current_exception();
}
}
```

```
int main() {
std::thread t(worker);
t.join();
// 檢查是否有例外
if (threadException) {
try {
std::rethrow_exception(threadException);
} catch (const std::exception& e) {
std::cout << "主執行緒捕獲: " << e.what() << std::endl;
}
}
return 0;
}
```

輸出：

```
主執行緒捕獲: 執行緒內的錯誤！
```

四、exception_ptr 關鍵函式

```
┌──────────────────────────────────────────────────────────┐
│ 例外傳遞相關函式 │
├──────────────────────────────────────────────────────────┤
│ │
│ std::current_exception() │
│ → 捕獲當前例外，回傳 exception_ptr │
│ │
│ std::rethrow_exception(ptr) │
│ → 重新拋出 exception_ptr 指向的例外 │
│ │
│ std::make_exception_ptr(e) │
│ → 從例外物件建立 exception_ptr │
│ │
└──────────────────────────────────────────────────────────┘
```

五、封裝成類別

```
#include <iostream>
#include <thread>
#include <stdexcept>
#include <exception>
#include <functional>

class SafeThread {
std::thread t;
std::exception_ptr exPtr = nullptr;
public:
template<typename Func, typename... Args>
explicit SafeThread(Func&& f, Args&&... args) {
t = std::thread([this, f = std::forward<Func>(f),
...args = std::forward<Args>(args)]() mutable {
try {
f(args...);
} catch (...) {
exPtr = std::current_exception();
}
});
}
void join() {
t.join();
if (exPtr) {
std::rethrow_exception(exPtr);
}
}
~SafeThread() {
if (t.joinable()) {
t.join();
}
}
};

int main() {
try {
SafeThread st([]() {
throw std::runtime_error("錯誤！");
});
```

```
st.join();
} catch (const std::exception& e) {
std::cout << "捕獲: " << e.what() << std::endl;
}
return 0;
}
```

六、解決方案三：使用 std::future（推薦）

`std::async` 和 `std::future` 自動處理例外傳遞：

```
#include <iostream>
#include <future>
#include <stdexcept>

int worker() {
throw std::runtime_error("非同步任務的錯誤！");
return 42;
}

int main() {
std::future<int> result = std::async(std::launch::async, worker);
try {
int value = result.get(); // 例外在這裡被重新拋出
std::cout << "結果: " << value << std::endl;
} catch (const std::exception& e) {
std::cout << "捕獲: " << e.what() << std::endl;
}
return 0;
}
```

輸出：

```
捕獲: 非同步任務的錯誤！
```

七、使用 std::promise 傳遞例外

```
#include <iostream>
#include <thread>
#include <future>
#include <stdexcept>

void worker(std::promise<int> prom) {
try {
throw std::runtime_error("Worker 錯誤！");
prom.set_value(42);
} catch (...) {
prom.set_exception(std::current_exception());
}
}

int main() {
std::promise<int> prom;
std::future<int> fut = prom.get_future();
std::thread t(worker, std::move(prom));
try {
int value = fut.get();
std::cout << "結果: " << value << std::endl;
} catch (const std::exception& e) {
std::cout << "捕獲: " << e.what() << std::endl;
}
t.join();
return 0;
}
```

八、各種方法比較

方法	優點	缺點
執行緒內捕獲	簡單	無法傳遞給呼叫者
exception_ptr	可傳遞任何例外	需要手動管理
std::future	自動傳遞	需要用 async 或 promise
promise	靈活控制	較複雜

九、最佳實踐

```
#include <iostream>
#include <future>
#include <vector>
#include <stdexcept>

int task(int id) {
if (id == 2) {
throw std::runtime_error("Task 2 失敗");
}
return id * 10;
}

int main() {
std::vector<std::future<int>> futures;
// 啟動多個任務
for (int i = 0; i < 5; ++i) {
futures.push_back(std::async(std::launch::async, task, i));
}
// 收集結果，處理例外
for (int i = 0; i < 5; ++i) {
try {
int result = futures[i].get();
std::cout << "Task " << i << " 結果: " << result << std::endl;
} catch (const std::exception& e) {
std::cout << "Task " << i << " 例外: " << e.what() << std::endl;
}
```

```
}
return 0;
}
```

輸出：

```
Task 0 結果: 0
Task 1 結果: 10
Task 2 例外: Task 2 失敗
Task 3 結果: 30
Task 4 結果: 40
```

十、本課重點回顧

1. 執行緒中的例外**不會自動傳遞**到其他執行緒
2. 未捕獲的例外會導致 `std::terminate()`
3. `std::exception_ptr` 可以儲存和傳遞例外
4. `std::current_exception()` 捕獲當前例外
5. `std::rethrow_exception()` 重新拋出例外
6. **推薦使用 `std::future`**，它自動處理例外傳遞

下一課預告

在 **課程 3.5：執行緒本地儲存** 中，我們將學習：

- `thread_local` 關鍵字
- 每個執行緒獨立的變數
- 使用場景與注意事項

準備好繼續嗎？

好的！讓我幫你快速掌握 `std::exception_ptr`。

一句話解釋

`std::exception_ptr` 讓你可以「捕捉、儲存、傳遞」例外，稍後在其他地方重新拋出。

為什麼需要它？

問題：例外無法跨執行緒傳遞

```
std::thread t([]() {
throw std::runtime_error("子執行緒出錯！");
});

t.join();
// 主執行緒完全不知道子執行緒拋了例外
// 程式直接 terminate()！
```

```
子執行緒 主執行緒
──────────────────────────────────────────
throw exception
  │
  ▼
沒人接住！
  │
  ▼
std::terminate() 💥
```

解決：用 `exception_ptr` 儲存並傳遞

```
子執行緒 主執行緒
──────────────────────────────────────────
throw exception
  │
  ▼
catch 並儲存到 exception_ptr
  │
  └──────────────────────► 讀取 exception_ptr
                             │
                             ▼
                           rethrow_exception()
                             │
                             ▼
                           正常處理例外
```

三個核心函式

函式	作用
`std::current_exception()`	捕捉當前例外，回傳 `exception_ptr`
`std::rethrow_exception(ptr)`	重新拋出儲存的例外
`std::make_exception_ptr(e)`	直接把例外包成 `exception_ptr`

基礎範例

```
#include <iostream>
#include <exception>
#include <stdexcept>

int main() {
std::exception_ptr eptr; // 初始為 nullptr
// ========================================
// 步驟 1：捕捉並儲存例外
// ========================================
try {
throw std::runtime_error("發生錯誤！");
}
catch (...) {
eptr = std::current_exception(); // 儲存例外
std::cout << "例外已儲存\n";
}
// ========================================
// 步驟 2：稍後重新拋出
// ========================================
if (eptr) {
try {
std::rethrow_exception(eptr); // 重新拋出
```

```
}
catch (const std::exception& e) {
std::cout << "捕捉到: " << e.what() << "\n";
}
}
return 0;
}
```

輸出：

```
例外已儲存
捕捉到: 發生錯誤！
```

最重要的應用：跨執行緒傳遞例外

```
#include <iostream>
#include <thread>
#include <exception>
#include <stdexcept>

int main() {
std::exception_ptr eptr = nullptr;
std::thread t([&eptr]() {
try {
// 模擬工作中發生錯誤
throw std::runtime_error("子執行緒發生錯誤！");
}
catch (...) {
// 捕捉並儲存，不讓程式崩潰
eptr = std::current_exception();
}
});
t.join();
// 主執行緒檢查是否有例外
if (eptr) {
try {
std::rethrow_exception(eptr);
```

```
}
catch (const std::exception& e) {
std::cout << "主執行緒處理: " << e.what() << "\n";
}
}
else {
std::cout << "子執行緒正常完成\n";
}
return 0;
}
```

輸出：

```
主執行緒處理: 子執行緒發生錯誤！
```

完整的多執行緒範例

```
#include <iostream>
#include <thread>
#include <vector>
#include <exception>
#include <stdexcept>
#include <mutex>

int main() {
std::vector<std::exception_ptr> exceptions;
std::mutex mtx;
auto worker = [&](int id) {
try {
if (id == 2) {
throw std::runtime_error("Worker " + std::to_string(id) + " 失敗");
}
if (id == 4) {
throw std::logic_error("Worker " + std::to_string(id) + " 邏輯錯誤");
}
std::cout << "Worker " << id << " 完成\n";
}
```

```
catch (...) {
std::lock_guard<std::mutex> lock(mtx);
exceptions.push_back(std::current_exception());
}
};
// 啟動多個執行緒
std::vector<std::thread> threads;
for (int i = 1; i <= 5; i++) {
threads.emplace_back(worker, i);
}
for (auto& t : threads) {
t.join();
}
// 處理所有收集到的例外
std::cout << "\n=== 處理例外 ===\n";
for (auto& eptr : exceptions) {
try {
std::rethrow_exception(eptr);
}
catch (const std::logic_error& e) {
std::cout << "[邏輯錯誤] " << e.what() << "\n";
}
catch (const std::runtime_error& e) {
std::cout << "[執行錯誤] " << e.what() << "\n";
}
catch (const std::exception& e) {
std::cout << "[其他錯誤] " << e.what() << "\n";
}
}
return 0;
}
```

輸出：

```
Worker 1 完成
Worker 3 完成
Worker 5 完成

=== 處理例外 ===
[執行錯誤] Worker 2 失敗
```

[邏輯錯誤] Worker 4 邏輯錯誤

make_exception_ptr 的用法

不需要真的 throw，直接創建 exception_ptr ：

```
#include <iostream>
#include <exception>
#include <stdexcept>

std::exception_ptr create_error() {
// 方法 1：傳統方式（需要 throw + catch）
/*
try {
throw std::runtime_error("錯誤");
}
catch (...) {
return std::current_exception();
}
*/
// 方法 2：直接創建（更簡潔）
return std::make_exception_ptr(std::runtime_error("錯誤"));
}

int main() {
auto eptr = create_error();
try {
std::rethrow_exception(eptr);
}
catch (const std::exception& e) {
std::cout << "捕捉: " << e.what() << "\n";
}
return 0;
}
```

封裝成可重用的 SafeThread

```
#include <iostream>
#include <thread>
#include <exception>
#include <functional>

class SafeThread {
std::thread t;
std::exception_ptr eptr = nullptr;
public:
template<typename Func, typename... Args>
void start(Func&& f, Args&&... args) {
if (t.joinable()) {
t.join();
}
eptr = nullptr; // 重置
// 包裝函式，捕捉任何例外
auto wrapped = [this, f = std::forward<Func>(f),
...args = std::forward<Args>(args)]() mutable {
try {
f(args...);
}
catch (...) {
eptr = std::current_exception();
}
};
t = std::thread(wrapped);
}
void join() {
if (t.joinable()) {
t.join();
}
// join 後檢查是否有例外，重新拋出
if (eptr) {
std::rethrow_exception(eptr);
}
}
~SafeThread() {
if (t.joinable()) {
```

```
t.join();
}
// 解構時不拋例外，避免 terminate
}
};

int main() {
SafeThread st;
st.start([]() {
throw std::runtime_error("任務失敗！");
});
try {
st.join(); // 這裡會拋出子執行緒的例外
}
catch (const std::exception& e) {
std::cout << "主執行緒捕捉: " << e.what() << "\n";
}
return 0;
}
```

重點整理

函式	用途	範例
`std::exception_ptr`	儲存例外的智慧指標	`std::exception_ptr eptr;`
`std::current_exception()`	在 catch 內捕捉當前例外	`eptr = std::current_exception();`
`std::rethrow_exception(eptr)`	重新拋出儲存的例外	`std::rethrow_exception(eptr);`
`std::make_exception_ptr(e)`	直接創建（不需 throw）	`eptr = std::make_exception_ptr(err);`

注意事項

事項	說明
初始值	`exception_ptr` 預設為 `nullptr`
判斷是否有例外	`if (eptr) { ... }`
可以複製	多處可以持有同一個例外
執行緒安全	需要 mutex 保護共享的 `exception_ptr`
生命週期	只要有 `exception_ptr` 指向，例外物件就不會被銷毀

快速對照表

傳統例外處理 使用 exception_ptr

```
try { try {
throw Error(); throw Error();
} }
catch (Error& e) { catch (...) {
handle(e); eptr = std::current_exception();
} }

// 稍後...
if (eptr) {
try {
std::rethrow_exception(eptr);
}
catch (Error& e) {
handle(e);
}
}
```

有其他問題嗎？

好問題！讓我直接比較兩種方式。

兩種方式對比

方式 1：傳統 throw + catch

```
std::exception_ptr create_error_v1() {
try {
throw std::runtime_error("錯誤"); // 拋出
}
catch (...) {
return std::current_exception(); // 捕捉
}
}
```

方式 2：make_exception_ptr

```
std::exception_ptr create_error_v2() {
return std::make_exception_ptr(std::runtime_error("錯誤"));
}
```

優點比較

比較項目	throw + catch	make_exception_ptr
程式碼量	5-6 行	1 行
效能	較慢（涉及 stack unwinding）	較快（直接創建）
可讀性	看起來像真的在處理錯誤	意圖明確：只是創建
語意	「發生錯誤並捕捉」	「準備一個例外物件」

效能差異

```
#include <iostream>
#include <exception>
#include <stdexcept>
#include <chrono>

std::exception_ptr via_throw() {
try {
throw std::runtime_error("error");
}
catch (...) {
return std::current_exception();
}
}

std::exception_ptr via_make() {
return std::make_exception_ptr(std::runtime_error("error"));
}

int main() {
const int iterations = 100000;
// 測試 throw + catch
auto start1 = std::chrono::high_resolution_clock::now();
for (int i = 0; i < iterations; i++) {
auto eptr = via_throw();
}
auto end1 = std::chrono::high_resolution_clock::now();
// 測試 make_exception_ptr
auto start2 = std::chrono::high_resolution_clock::now();
for (int i = 0; i < iterations; i++) {
auto eptr = via_make();
}
auto end2 = std::chrono::high_resolution_clock::now();
auto time1 = std::chrono::duration_cast<std::chrono::milliseconds>(end1 -
start1);
auto time2 = std::chrono::duration_cast<std::chrono::milliseconds>(end2 -
start2);
std::cout << "throw + catch: " << time1.count() << " ms\n";
std::cout << "make_exception_ptr: " << time2.count() << " ms\n";
```

```
return 0;
}
```

典型輸出：

```
throw + catch: 450 ms
make_exception_ptr: 85 ms
```

`make_exception_ptr` **通常快 3-10 倍。**

為什麼 throw 較慢？

```
throw + catch 的過程：
──────────────────────────────────────
1. 創建例外物件
2. 開始 stack unwinding（回溯堆疊）
3. 搜尋匹配的 catch
4. 執行 catch
5. 複製到 exception_ptr

make_exception_ptr 的過程：
──────────────────────────────────────
1. 創建例外物件
2. 直接包裝成 exception_ptr
（沒有 unwinding，沒有搜尋 catch）
```

什麼時候用哪個？

用 `make_exception_ptr`

```
// 場景：主動創建一個例外來傳遞
std::exception_ptr report_error(const std::string& msg) {
return std::make_exception_ptr(std::runtime_error(msg));
}

// 場景：設定預設錯誤
class Task {
```

```
std::exception_ptr error =
std::make_exception_ptr(std::logic_error("未初始化"));
};
```

用 `current_exception`

```
// 場景：真的發生了例外，需要捕捉並儲存
void worker() {
try {
some_function_that_may_throw(); // 真的可能拋例外
}
catch (...) {
eptr = std::current_exception(); // 捕捉真實發生的例外
}
}
```

實際應用範例

```
#include <iostream>
#include <exception>
#include <stdexcept>
#include <future>

// 用 make_exception_ptr 創建失敗結果
std::exception_ptr validate_input(int value) {
if (value < 0) {
return std::make_exception_ptr(
std::invalid_argument("值不能為負數")
);
}
if (value > 100) {
return std::make_exception_ptr(
std::out_of_range("值超出範圍")
);
}
return nullptr; // 無錯誤
}
```

```
int main() {
for (int test : {50, -5, 150}) {
std::cout << "測試 " << test << ": ";
auto error = validate_input(test);
if (error) {
try {
std::rethrow_exception(error);
}
catch (const std::exception& e) {
std::cout << "錯誤 - " << e.what() << "\n";
}
}
else {
std::cout << "通過\n";
}
}
return 0;
}
```

輸出：

```
測試 50: 通過
測試 -5: 錯誤 - 值不能為負數
測試 150: 錯誤 - 值超出範圍
```

總結

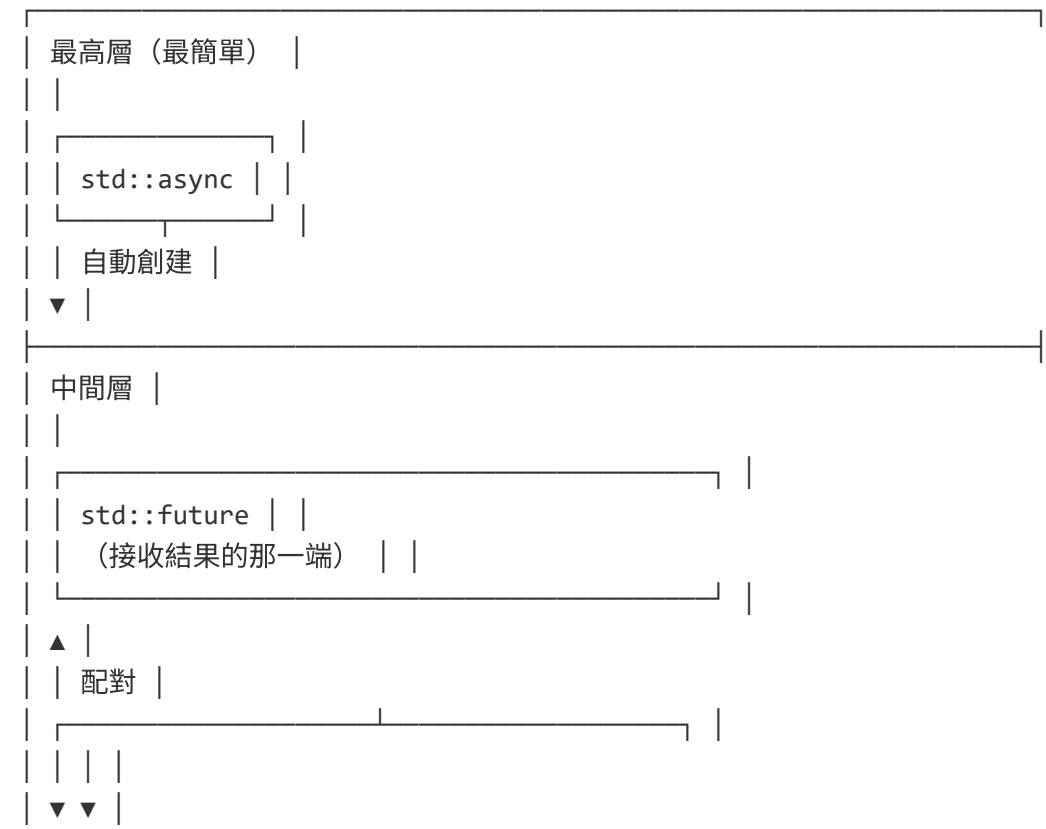
情況	建議用法
主動創建例外物件來傳遞	`make_exception_ptr` ✓
捕捉真實發生的例外	`current_exception` ✓
效能敏感的程式碼	`make_exception_ptr` ✓
需要保留原始例外的完整資訊	`current_exception` ✓

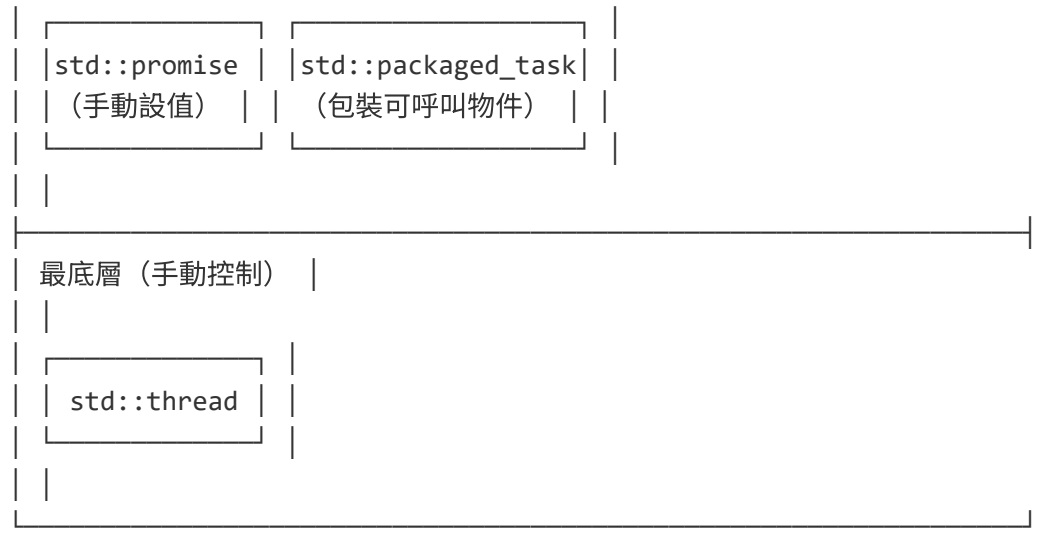
簡單原則：

- 「真的拋了」→ `current_exception()`
- 「只是想創建」→ `make_exception_ptr()`

好問題！讓我幫你建立一個清晰的全局觀。

層次架構圖

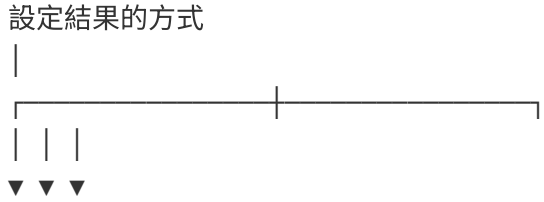


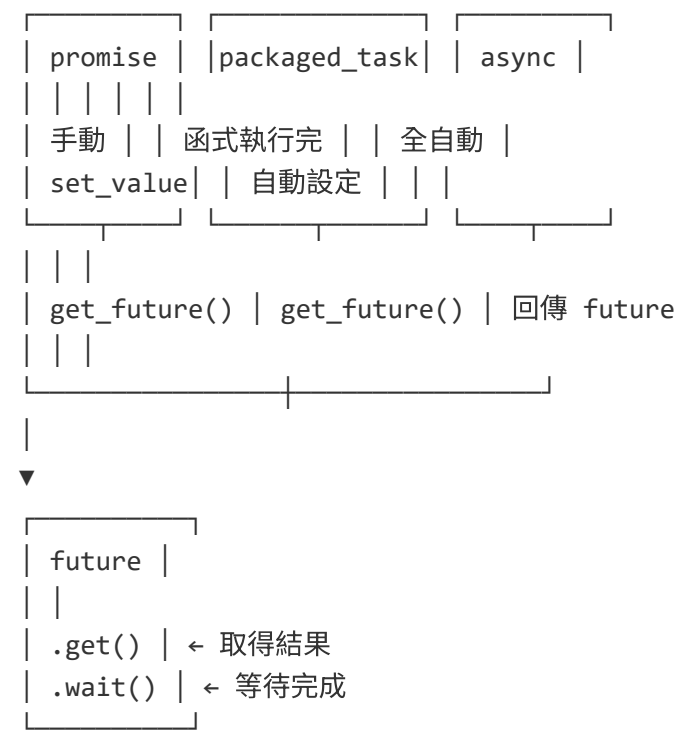


一句話總結每個元件

元件	角色	一句話說明
`std::thread`	最底層	原始執行緒，不管回傳值
`std::promise`	生產者	手動設定結果，配對一個 future
`std::packaged_task`	包裝器	把函式包起來，自動設定 future
`std::future`	消費者	等待並取得結果
`std::async`	最高層	全自動：創建執行緒 + 回傳 future

關係圖





程式碼對比：同一件事，四種寫法

目標：在另一個執行緒計算 10 + 20，主執行緒取得結果

層級 1：std::thread（最底層，最麻煩）

```
#include <iostream>
#include <thread>

int main() {
int result = 0;
bool done = false;
std::thread t([&]() {
result = 10 + 20; // 透過共享變數傳遞
done = true;
});
t.join();
std::cout << "結果: " << result << "\n";
return 0;
}
```

缺點： 需要自己管理共享變數、同步、沒有例外處理

層級 2a：std::promise + std::future

```
#include <iostream>
#include <thread>
#include <future>

int main() {
std::promise<int> prom;
std::future<int> fut = prom.get_future(); // 配對
std::thread t([&prom]() {
int result = 10 + 20;
prom.set_value(result); // 手動設定結果
});
std::cout << "結果: " << fut.get() << "\n"; // 等待並取得
t.join();
return 0;
}
```

特點： 手動控制何時設定結果

層級 2b：std::packaged_task + std::future

```
#include <iostream>
#include <thread>
#include <future>

int main() {
// 包裝函式
std::packaged_task<int()> task([]() {
return 10 + 20;
});
std::future<int> fut = task.get_future(); // 取得配對的 future
std::thread t(std::move(task)); // 必須 move
std::cout << "結果: " << fut.get() << "\n";
t.join();
return 0;
```

```
}
```

特點： 函式執行完自動設定結果

層級 3：std::async（最高層，最簡單）

```
#include <iostream>
#include <future>

int main() {
std::future<int> fut = std::async([]() {
return 10 + 20;
});
std::cout << "結果: " << fut.get() << "\n";
return 0;
}
```

特點： 一行搞定，不用管 thread、promise、packaged_task

內部關係

```
std::async 內部實作（概念上）：
═══════════════════════════════════════════════════

std::future<T> async(Func f) {
// 1. 創建 packaged_task
std::packaged_task<T()> task(f);
// 2. 取得 future
std::future<T> fut = task.get_future();
// 3. 創建 thread 執行 task
std::thread t(std::move(task));
t.detach(); // 或其他管理方式
// 4. 回傳 future
return fut;
}
```

packaged_task 內部實作（概念上）：
═══

```
template<typename T>
class packaged_task {
std::function<T()> func;
std::promise<T> prom; // 內含 promise！
public:
void operator()() {
try {
T result = func();
prom.set_value(result); // 自動設定結果
}
catch (...) {
prom.set_exception(std::current_exception());
}
}
std::future<T> get_future() {
return prom.get_future();
}
};
```

何時用哪個？

```
┌──────────────────────────────────────────────────────────────┐
│ 選擇指南 │
├──────────────────────────────────────────────────────────────┤
│ │
│ 「我只想非同步執行函式，拿到結果」 │
│ │ │
│ └──→ std::async │
│ │
│ 「我想把任務排進執行緒池或佇列」 │
│ │ │
│ └──→ std::packaged_task │
│ │
│ 「我想完全控制何時、如何設定結果」 │
```

```
│ │ │
│ └──► std::promise + std::future │
│ │
│ 「我不需要回傳值，只是平行執行」 │
│ │ │
│ └──► std::thread │
│ │
└──────────────────────────────────────────────────────────────────┘
```

實際應用場景

async：簡單的非同步計算

```
// 最適合：fire and forget，之後要結果
auto fut1 = std::async(download_file, "file1.txt");
auto fut2 = std::async(download_file, "file2.txt");

// 做其他事...

auto data1 = fut1.get();
auto data2 = fut2.get();
```

packaged_task：任務佇列 / 執行緒池

```
// 最適合：需要把任務存起來，稍後執行
std::queue<std::packaged_task<int()>> task_queue;

// 生產者：加入任務
std::packaged_task<int()> task(compute_something);
auto fut = task.get_future();
task_queue.push(std::move(task));

// 消費者（執行緒池的 worker）：取出並執行
auto task = std::move(task_queue.front());
task_queue.pop();
task(); // 執行
```

promise：跨多個步驟設定結果

```
// 最適合：結果不是由單一函式產生
std::promise<int> prom;
auto fut = prom.get_future();

std::thread t([&prom]() {
int partial = step1();
if (some_condition) {
prom.set_value(partial); // 提前完成
return;
}
int final_result = step2(partial);
prom.set_value(final_result); // 完整結果
});
```

總結表

層級	元件	自動程度	使用場景
高	`std::async`	全自動	簡單非同步呼叫
中	`std::packaged_task`	半自動	任務佇列、執行緒池
中	`std::promise`	手動	完全控制結果設定
低	`std::thread`	手動	不需要回傳值

記憶口訣：

- `async` = 懶人專用，全包
- `packaged_task` = 函式包裝紙，可以存起來
- `promise` = 我說了算，手動設值
- `future` = 等快遞，取結果

有其他問題嗎？

課程 3.5：執行緒本地儲存

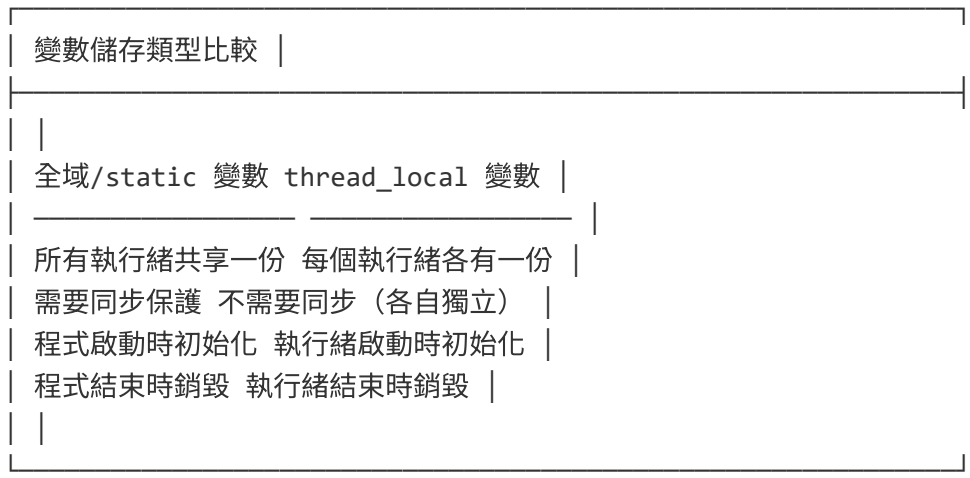
第三階段：執行緒生命週期管理

課程 3.5：執行緒本地儲存

引言

有時我們希望**每個執行緒都有自己獨立的變數副本，而不是共享同一個變數。C++11 引入的 `thread_local` 關鍵字正是為此設計。**

一、thread_local 基本概念



二、基本用法

```
#include <iostream>
#include <thread>

thread_local int counter = 0; // 每個執行緒有獨立的 counter

void increment(const std::string& name) {
for (int i = 0; i < 3; ++i) {
++counter;
```

```
std::cout << name << ": counter = " << counter << std::endl;
}
}

int main() {
std::thread t1(increment, "Thread A");
std::thread t2(increment, "Thread B");
t1.join();
t2.join();
return 0;
}
```

輸出（順序可能不同）：

```
Thread A: counter = 1
Thread A: counter = 2
Thread A: counter = 3
Thread B: counter = 1
Thread B: counter = 2
Thread B: counter = 3
```

注意：兩個執行緒的 counter 各自獨立從 0 開始。

三、對比：有無 thread_local

```
#include <iostream>
#include <thread>

int globalCounter = 0; // 共享
thread_local int localCounter = 0; // 各自獨立

void work(const std::string& name) {
++globalCounter;
++localCounter;
std::cout << name
<< " global=" << globalCounter
<< " local=" << localCounter << std::endl;
}
```

```
int main() {
std::thread t1(work, "A");
std::thread t2(work, "B");
std::thread t3(work, "C");
t1.join();
t2.join();
t3.join();
return 0;
}
```

可能的輸出：

```
A global=1 local=1
B global=2 local=1
C global=3 local=1
```

`globalCounter` 累加，`localCounter` 每個執行緒都是 1。

四、thread_local 的位置

可以用於三種地方：

```
#include <iostream>
#include <thread>

// 1. 全域變數
thread_local int global_tl = 0;

void func() {
// 2. 函式內的 static 變數
thread_local static int local_tl = 0;
++global_tl;
++local_tl;
std::cout << "global_tl=" << global_tl
<< " local_tl=" << local_tl << std::endl;
}

class MyClass {
public:
// 3. 類別的 static 成員
```

```
thread_local static int member_tl;
};

thread_local int MyClass::member_tl = 0;

int main() {
std::thread t1([]() { func(); func(); });
std::thread t2([]() { func(); func(); });
t1.join();
t2.join();
return 0;
}
```

五、常見用途

用途一：錯誤碼（如 errno）

```
#include <iostream>
#include <thread>

thread_local int lastError = 0;

void setError(int code) {
lastError = code;
}

int getError() {
return lastError;
}

void worker(int id) {
setError(id * 100);
std::cout << "Thread " << id << " error: " << getError() << std::endl;
}

int main() {
std::thread t1(worker, 1);
std::thread t2(worker, 2);
t1.join();
```

```
t2.join();
return 0;
}
```

用途二：快取

```
#include <iostream>
#include <thread>
#include <string>

thread_local std::string cache;

std::string expensiveCompute(int id) {
if (cache.empty()) {
// 模擬耗時計算
cache = "Result for thread " + std::to_string(id);
std::cout << "Computing..." << std::endl;
}
return cache;
}

void worker(int id) {
std::cout << expensiveCompute(id) << std::endl;
std::cout << expensiveCompute(id) << std::endl; // 使用快取
}

int main() {
std::thread t1(worker, 1);
std::thread t2(worker, 2);
t1.join();
t2.join();
return 0;
}
```

輸出：

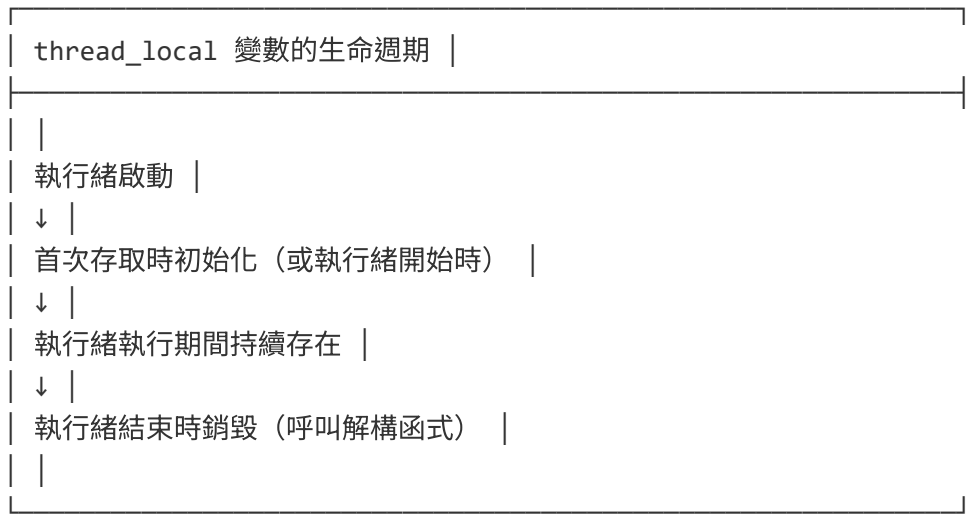
```
Computing...
```

```
Result for thread 1
Result for thread 1
Computing...
Result for thread 2
Result for thread 2
```

用途三：隨機數產生器

```
#include <iostream>
#include <thread>
#include <random>

thread_local std::mt19937 rng{std::random_device{}()};

int randomInt(int min, int max) {
std::uniform_int_distribution<int> dist(min, max);
return dist(rng);
}

void worker(int id) {
std::cout << "Thread " << id << ": "
<< randomInt(1, 100) << ", "
<< randomInt(1, 100) << ", "
<< randomInt(1, 100) << std::endl;
}

int main() {
std::thread t1(worker, 1);
std::thread t2(worker, 2);
t1.join();
t2.join();
return 0;
}
```

六、生命週期



七、注意事項

```
#include <iostream>
#include <thread>

class Resource {
public:
Resource() { std::cout << "Resource 建構" << std::endl; }
~Resource() { std::cout << "Resource 銷毀" << std::endl; }
};

thread_local Resource res; // 每個執行緒有自己的 Resource

void worker(int id) {
std::cout << "Thread " << id << " 使用資源" << std::endl;
}

int main() {
std::cout << "=== 主執行緒 ===" << std::endl;
std::thread t1(worker, 1);
std::thread t2(worker, 2);
t1.join();
t2.join();
```

```
std::cout << "=== 結束 ===" << std::endl;
return 0;
}
```

輸出：

```
=== 主執行緒 ===
Resource 建構
Thread 1 使用資源
Resource 建構
Thread 2 使用資源
Resource 銷毀
Resource 銷毀
=== 結束 ===
```

八、thread_local vs 其他方案

方案	優點	缺點
thread_local	語法簡潔、自動管理	無法跨執行緒存取
傳遞參數	明確、可控	需要到處傳遞
執行緒 ID map	可跨執行緒存取	需要同步、較複雜

九、本課重點回顧

1. `thread_local` 讓每個執行緒有獨立的變數副本
2. 不需要同步，因為各執行緒存取自己的副本
3. 變數在執行緒首次存取時初始化，結束時銷毀
4. 常見用途：錯誤碼、快取、隨機數產生器
5. 可用於全域變數、函式內 static、類別 static 成員

下一課預告

在 **課程 3.6：執行緒安全的初始化** 中，我們將學習：

- `std::call_once` 確保只初始化一次
- `std::once_flag` 的使用
- 單例模式的執行緒安全實作

準備好繼續嗎？

課程 3.6：執行緒安全的初始化

第三階段：執行緒生命週期管理

課程 3.6：執行緒安全的初始化

引言

在多執行緒環境中，某些資源只應該被初始化一次（例如單例物件、全域配置）。本課學習如何確保初始化只發生一次，即使多個執行緒同時嘗試初始化。

一、問題：不安全的初始化

```
#include <iostream>
#include <thread>

class Database {
public:
Database() { std::cout << "Database 初始化" << std::endl; }
void query() { std::cout << "查詢中..." << std::endl; }
};

Database* db = nullptr;

void initAndUse() {
if (db == nullptr) { // 執行緒 A 檢查
db = new Database(); // 執行緒 A 和 B 都可能執行這行！
}
db->query();
}

int main() {
```

```
std::thread t1(initAndUse);
std::thread t2(initAndUse);
t1.join();
t2.join();
delete db;
return 0;
}
```

可能輸出：

```
Database 初始化
Database 初始化
查詢中...
查詢中...
```

Database 被初始化了兩次！這是競爭條件。

二、解決方案：std::call_once

```
#include <iostream>
#include <thread>
#include <mutex>

class Database {
public:
Database() { std::cout << "Database 初始化" << std::endl; }
void query() { std::cout << "查詢中..." << std::endl; }
};

Database* db = nullptr;
std::once_flag initFlag;

void initDatabase() {
db = new Database();
}

void initAndUse() {
std::call_once(initFlag, initDatabase);
db->query();
```

```
}

int main() {
std::thread t1(initAndUse);
std::thread t2(initAndUse);
std::thread t3(initAndUse);
t1.join();
t2.join();
t3.join();
delete db;
return 0;
}
```

輸出：

```
Database 初始化
查詢中...
查詢中...
查詢中...
```

不管多少執行緒，Database 只初始化一次。

三、call_once 工作原理

```
┌──────────────────────────────────────────────────────────────┐
│ std::call_once 機制 │
├──────────────────────────────────────────────────────────────┤
│ │
│ std::once_flag flag; │
│ → 記錄是否已經執行過 │
│ │
│ std::call_once(flag, func); │
│ → 第一個到達的執行緒執行 func │
│ → 其他執行緒等待直到 func 完成 │
│ → 之後的呼叫直接跳過（flag 已設定） │
│ │
│ 如果 func 拋出例外： │
│ → flag 不會被設定 │
│ → 下一個執行緒會再次嘗試執行 │
```

四、使用 Lambda

更簡潔的寫法：

```
#include <iostream>
#include <thread>
#include <mutex>

class Database {
public:
Database() { std::cout << "Database 初始化" << std::endl; }
void query() { std::cout << "查詢中..." << std::endl; }
};

Database* db = nullptr;
std::once_flag initFlag;

void initAndUse() {
std::call_once(initFlag, []() {
db = new Database();
});
db->query();
}

int main() {
std::thread t1(initAndUse);
std::thread t2(initAndUse);
t1.join();
t2.join();
delete db;
return 0;
}
```

五、執行緒安全的單例模式
方法一：使用 call_once

```
#include <iostream>
#include <thread>
#include <mutex>

class Singleton {
static Singleton* instance;
static std::once_flag initFlag;
Singleton() { std::cout << "Singleton 建立" << std::endl; }
public:
Singleton(const Singleton&) = delete;
Singleton& operator=(const Singleton&) = delete;
static Singleton& getInstance() {
std::call_once(initFlag, []() {
instance = new Singleton();
});
return *instance;
}
void doSomething() { std::cout << "工作中" << std::endl; }
};

Singleton* Singleton::instance = nullptr;
std::once_flag Singleton::initFlag;

int main() {
std::thread t1([]() { Singleton::getInstance().doSomething(); });
std::thread t2([]() { Singleton::getInstance().doSomething(); });
t1.join();
t2.join();
return 0;
}
```

方法二：使用 static 區域變數（更簡潔，C++11 保證安全）

```
#include <iostream>
#include <thread>
```

```
class Singleton {
Singleton() { std::cout << "Singleton 建立" << std::endl; }
public:
Singleton(const Singleton&) = delete;
Singleton& operator=(const Singleton&) = delete;
static Singleton& getInstance() {
static Singleton instance; // C++11 保證執行緒安全
return instance;
}
void doSomething() { std::cout << "工作中" << std::endl; }
};

int main() {
std::thread t1([]() { Singleton::getInstance().doSomething(); });
std::thread t2([]() { Singleton::getInstance().doSomething(); });
t1.join();
t2.join();
return 0;
}
```

C++11 標準保證：區域 static 變數的初始化是執行緒安全的。

六、兩種方法比較

```
┌──────────────────────────────────────────────────────────────┐
│ call_once vs static 區域變數 │
├──────────────────────────────────────────────────────────────┤
│ │
│ std::call_once static 區域變數 │
│ ──────────────── ──────────────────── │
│ • 明確控制初始化時機 • 更簡潔 │
│ • 可用於非建構函式的初始化 • 只能用於建構函式 │
│ • 需要 once_flag • 不需要額外變數 │
│ • 較靈活 • C++11 自動保證安全 │
│ │
│ 建議：單例優先用 static 區域變數 │
│ 複雜初始化用 call_once │
│ │
└──────────────────────────────────────────────────────────────┘
```

七、call_once 與例外

如果初始化函式拋出例外，下一個執行緒會重新嘗試：

```
#include <iostream>
#include <thread>
#include <mutex>
#include <atomic>

std::once_flag flag;
std::atomic<int> attempt{0};

void mayFail() {
int current = ++attempt;
std::cout << "嘗試 #" << current << std::endl;
if (current < 3) {
throw std::runtime_error("初始化失敗");
}
std::cout << "初始化成功！" << std::endl;
}

void worker() {
try {
std::call_once(flag, mayFail);
std::cout << "繼續執行" << std::endl;
} catch (const std::exception& e) {
std::cout << "捕獲例外: " << e.what() << std::endl;
}
}

int main() {
std::thread t1(worker);
std::thread t2(worker);
std::thread t3(worker);
std::thread t4(worker);
t1.join();
t2.join();
t3.join();
```

```
t4.join();
return 0;
}
```

可能輸出：

```
嘗試 #1
捕獲例外: 初始化失敗
嘗試 #2
捕獲例外: 初始化失敗
嘗試 #3
初始化成功！
繼續執行
繼續執行
```

八、類別成員的延遲初始化

```
#include <iostream>
#include <thread>
#include <mutex>
#include <memory>

class Service {
mutable std::once_flag cacheFlag;
mutable std::unique_ptr<std::string> cache;
void initCache() const {
std::cout << "初始化快取..." << std::endl;
cache = std::make_unique<std::string>("快取資料");
}
public:
const std::string& getCache() const {
std::call_once(cacheFlag, &Service::initCache, this);
return *cache;
}
};

int main() {
```

```
Service service;
std::thread t1([&]() { std::cout << service.getCache() << std::endl; });
std::thread t2([&]() { std::cout << service.getCache() << std::endl; });
t1.join();
t2.join();
return 0;
}
```

九、本課重點回顧

1. 多執行緒環境下的初始化可能發生競爭條件
2. `std::call_once` 確保函式只執行一次
3. `std::once_flag` 記錄是否已執行
4. 如果初始化拋出例外，下一個執行緒會重新嘗試
5. C++11 保證區域 static 變數初始化是執行緒安全的
6. 單例模式優先使用區域 static 變數實作

第三階段完成！

恭喜你完成了執行緒生命週期管理階段！你已經學會：

- ✅ RAII 執行緒管理
- ✅ 執行緒守衛類別設計
- ✅ std::jthread 與 stop_token
- ✅ 執行緒例外處理
- ✅ thread_local 儲存
- ✅ std::call_once 安全初始化

下一階段預告

第四階段：共享資料與競爭條件 將深入探討：

- 課程 4.1：共享資料的問題
- 課程 4.2：不變量與競爭條件
- 課程 4.3：臨界區段概念
- ...

準備好進入第四階段嗎？