c++基础与进阶(类与并发)

第一部分：现代 C++ 的语言核心

🔴 一，引用

1️⃣ 引用概述

💬 **背景 / 结论 引用**：C++ 的引用，本质上就是“受严格约束、不能为 null、不能改指向的指针语义”。**这是“语义层面的本质”，不是“语法等价”**

——编译器层面，**它可以实现成指针，也可以完全消掉。**

2️⃣ 常见引用

1. **const T&**

💬 **背景 / 结论** `const T&` 是 C++ 中“只读借用（borrow）”的标准表达方式

作用:

- 不拷贝
- 不可修改
- **可绑定临时对象**

在概念上，可以这么理解：

```
const int& r = 10;
// 等价于
int __temp = 10;
const int& r = __temp;
```

这里补充引用生命周期的常见坑：

一. ❌ 返回局部对象的引用（必死）

```
const int& f() {
    int x = 10;
    return x;   // ❌ 悬空引用
}
```

- 生命周期延长规则 只对“绑定发生的地方”生效。函数返回时，x 已经销毁

具体来说：

- 临时对象的生命周期 只能被延长到“当前完整表达式”

- 函数返回会 切断作用域
- 引用逃逸了

二. ❌ 绑定到函数返回的临时再返回

```
const std::string& g() {
    return std::string("hello"); // ❌一样没用，原理同上
}
```

2. **T&**

🗨️ **背景 / 结论** 判断一个表达式是不是左值，只问一个问题：

👉 “我能不能在程序中再次找到它？”

```
int& g() {
    static int x = 10;
    return x;
}

g() = 20;   // 合法
```

3. **T&&**

🗨️ **背景 / 结论** C++11 把右值分成了两类：**纯右值（prvalue）**,**将亡值（xvalue）**

1️⃣ 纯右值

```
10
a + b
f()
```

- 特点：临时,没身份,用完即弃

2️⃣ 将亡值（为 move 准备的）

```
std::move(a)
```

- 含义：它指向的是：一个仍然存在的对象，只是你明确告诉编译器：“我以后不再需要它的值了”

综合上面

形式	能绑定	语义
T&	左值	可修改、长期存在
const T&	左值 + 右值	只读借用
T&&	右值	可被“掠夺资源”

3️⃣ 引用折叠 & T&& 的“变形”——完美转发的根源

场景：

```cpp
template<typename T>
void f(T&& x) {
    // x 是什么？
}

int a = 10;
f(a);
```

💬 **背景 / 结论** 引出一个事实：**T&& 在模板里，不总是右值引用**

在上面的例子中：

```cpp
void f(int& && x);
```

这里面存在一个规则：**引用折叠（reference collapsing）规则**

原始形式	折叠后
& &	&
& &&	&
&& &	&
&& &&	&&

所以，所以 T&& 的真实含义是：

💬 **背景 / 结论** T&& ≠ 右值引用

💬 **背景 / 结论**

💬 **背景 / 结论** T&& = “我接受任何值，并保留它原本的值类别”

这就是：👉 **完美转发（perfect forwarding）**

于此同时，**auto&&** 也具有同样的含义，如auto&& x1 = a; // int&

因此 **auto&&** 被称作 **通用引用**，上面的 **T&&** 被称为万能引用

然后为了防止引用折叠导致的传参本意错误，引入了**std::forward**

```cpp
template<typename T>
void wrapper(T&& x) {     // x 有名字 → 左值
    foo(std::forward<T>(x));    // 还原回T&&推导出的原本类型
}
```

4️⃣ 移动语义（Move Semantics）——“偷资源，而不是拷贝”

💬 **背景 / 结论** std::move(a) 的作用是：

💬 **背景 / 结论**

💬 **背景 / 结论** 把 a 转成“将亡值（xvalue）”，允许调用移动构造

⚠️ 它不移动任何东西

拷贝 vs 移动：一次完整对比

```
Buffer a(100);
Buffer b = a;              // 拷贝构造
Buffer c = std::move(a);   // 移动构造
```

操作	结果
b = a	深拷贝
c = std::move(a)	偷指针
a	仍然合法，但内容被“掏空”

👉 被 move 的对象必须还能安全析构

于是对于**管理资源**（new / malloc / 文件 / mutex），通常需要：

1.析构函数

2.拷贝构造

3.拷贝赋值

4.移动构造

5.移动赋值

```
struct Buffer {
    ~Buffer();
    Buffer(const Buffer&);
    Buffer& operator=(const Buffer&);
    Buffer(Buffer&&) noexcept;
    Buffer& operator=(Buffer&&) noexcept;
};
```

⚠️ **noexcept** 保证性能的关键：启用std::move()而非拷贝构造

🟠 二，类

结合上面的std::move()，不难注意到**构造函数**这一名词：

1️⃣ 构造函数

对于一个类(以下用Buffer，名字随便起的)，一共有 4 类**特殊成员函数（special member functions）**：

函数	类型	什么时候被调用	解决什么问题
~Buffer()	**析构函数**	对象生命周期结束	释放资源
Buffer(const Buffer&)	**拷贝构造函数**	用“已有对象”创建新对象	深拷贝
Buffer(Buffer&&)	**移动构造函数**	用“将亡对象”创建新对象	偷资源
Buffer(Args args,...)	**普通构造函数**	从参数创建对象	初始化资源

1️⃣ ~classf() —— **析构函数（Destructor）**

```
~Buffer() {
    delete[] data;
```

```
}
```

什么时候调用?

- 栈对象离开作用域
- delete 一个堆对象
- 容器销毁元素
- 异常展开（stack unwinding）

它的语义承诺是:

💬 **背景 / 结论 “我负责清理这个对象所拥有的资源”**

2️⃣ classf(const classf& other) —— **拷贝构造函数**

```
Buffer(const Buffer& other)
    : data(new int[other.size]), size(other.size) {
    std::copy(other.data, other.data + size, data);
}
```

它的语义承诺是:

💬 **背景 / 结论 “新对象与旧对象语义等价，但资源彼此独立”**

3️⃣ Buffer(Buffer&& other) noexcept —— **移动构造函数**

```
Buffer(Buffer&& other) noexcept
    : data(other.data), size(other.size) {
    other.data = nullptr;
    other.size = 0;
}
```

它的语义承诺是:

💬 **背景 / 结论 “我会接管你的资源，你以后别再用它的内容”**

即

💬 **背景 / 结论 “用一个即将被销毁的对象，构造一个新对象”**

2️⃣ 运算符重载

✅ 适合重载的（“数学/结构语义清晰”）

- ▪ ◦ ◦ /（数值、向量、复数）
- == != < > <= >=（比较）
- []（索引访问）
- ->（迭代器、智能指针）
- <<（输出）

基本语法

1️⃣ 对于成员函数

💬 **背景 / 结论** [返回类型] operator[要重载的符号](const MyClass& other) const {}

2️⃣ 对于成员函数

💬 **背景 / 结论** [返回类型] operator[要重载的符号](const MyClass& a,const MyClass& b) const {}

两个**const**的含义：

- const MyClass& other → 我不改你
- 函数末尾 const → 我不改自己

常见场景

1️⃣ 比较运算符重载

```
auto operator<=>(const MyClass&) const = default;
bool operator==(const MyClass&) const = default;
```

解释：

- 使用三路运算符，让编译器帮忙生成各种比较运算
- = default 的含义：使用默认的比较方式
- 为什么要把==单独再次声明?: **为了性能！**使用<=>时，它生成的==运算是**不会先比较size**的，而是**比较内容是否相等**，而再次声明的==强调了，如果二者size不相等，就**不用后续比较详细细节**了，提升了性能。
- 为什么使用auto?: 因为<=>返回的类型并非bool,而是一种比较结果对象：

```
std::strong_ordering
std::weak_ordering
std::partial_ordering
std::strong_equality
```

1. **std::strong_ordering**

它表示一种 **严格的全序关系**：

- 任意两个值都可比较
- 满足：
  - 传递性
  - 对称性

典型场景

- int, char, string...

2. **std::weak_ordering（“看起来相等，但不是真的相等”）**

典型例子（概念）

- "abc" 与 "ABC"：排序上等价，但内容不相等

可取值

```
std::weak_ordering::less
std::weak_ordering::equivalent
std::weak_ordering::greater
```

3. **std::partial_ordering（可能“不可比较”）**

典型例子

```
double x = NAN;
double y = 1.0;

auto r = x <=> y;   // unordered
```

可取值

```
std::partial_ordering::less
std::partial_ordering::equivalent
std::partial_ordering::greater
std::partial_ordering::unordered
```

4. **std::strong_equality（只关心等不等）**

使用场景

- 你只想支持 == / !=

可取值

```
std::strong_equality::equal
std::strong_equality::not_equal
```

2️⃣ 迭代器/指针运算符重载

```
T& operator*() const;
T* operator->() const;
```

3️⃣ 输出运算符重载

```
std::ostream& operator<<(std::ostream& os, const Vec& v) {
    return os << "(" << v.x << "," << v.y << ")";
}
```

或者

```
friend std::ostream& operator<<(std::ostream& os, const Vec& v) {
    return os << "(" << v.x << "," << v.y << ")";
}
```

解释：

- ostream对于你要重载的信息输出来说是一个非成员函数。
- 输出类型要使用std::ostream&，这是保证输出流唯一的方式，且是为了保证后面链式调用的关键。

```
std::cout << a << b << c;
```
等价于

```
(((std::cout << a) << b) << c);
```
4️⃣ 结合tuple的多元素比较简洁写法

```
bool operator<=>(const MyClass& other) const {
    return std::tie(x, y) <=> std::tie(other.x, other.y);
}
```
解释：

- tuple已经帮我们重载好了相关运算符

```
std::tie(x, y) <=> std::tie(other.x, other.y);
```
等价于

```
(x, y) <=> (other.x,other.y)
```
🟡 三，指针

💬 **背景 / 结论** C语言中的指针能使程序员能很好的接触底层，但也常常导致某些问题（如忘记delete，导致内存泄漏），于是:

1️⃣ 智能指针

类型	含义
std::unique_ptr	一个只允许唯一拥有者的指针
std::shared_ptr	带引用计数的智能指针,用于共享所有权
std::weak_ptr	不参与所有权的观察者

1️⃣ std::unique_ptr

unique_ptr的**灵魂实现**（极简）

```
template<typename T>
class unique_ptr {
    T* ptr;
public:
    ~unique_ptr() {
        delete ptr;
    }

    unique_ptr(const unique_ptr&) = delete;       // ❌ 禁拷贝
    unique_ptr(unique_ptr&& other) noexcept {     // ✅ 可移动
        ptr = other.ptr;
        other.ptr = nullptr;
    }
};
```
使用上:

- 创建:

```
std::unique_ptr<Buffer> p = std::make_unique<Buffer>(100);
//或者
auto p = std::make_unique<Buffer>(100);
```

- 传参：

```
//只查看
void display(const std::unique_ptr<MyClass>& ptr) {}
useObject(obj.get());
void display(const std::unique_ptr<MyClass>* ptr) {}
useObject(*obj);

//要转移所有权时
std::unique_ptr<MyClass> display(std::unique_ptr<MyClass> ptr) {return
ptr; // 所有权转移回调用者}
auto obj = std::make_unique<MyClass>();
obj = processAndTransfer(std::move(obj));

//可能转移所有权
void maybeTakeOwnership(std::unique_ptr<MyClass>&& ptr) {
 if (/* 某些条件 */) {auto owned = std::move(ptr);  // 取得所有权}
}
auto obj = std::make_unique<MyClass>();
maybeTakeOwnership(std::move(obj));
```

2️⃣ **std::shared_ptr**

💬 **背景 / 结论** 场景：多个对象“共同使用”一个资源，如图形材质的共享

💬 **背景 / 结论** shared_ptr = “带引用计数的智能指针”

他保证了：

- 有多少个 shared_ptr 指向同一资源，资源就活着
- 最后一个销毁时，资源才释放

使用上：

- 创建：

```
auto p = std::make_shared<T>(args...);
shared_ptr<T> p(new T(args...));
```

- 前者更好，因为在内存分配上

前者：[control block + T 对象]

后者：[control block] + [T]

👉 **缓存友好 + 更快 + 更安全**

使用代价

- 原子操作（线程安全）
- 每次拷贝 / 析构都要 ++ / --

- 容易产生“循环引用”灾难 → 引用计数永远 >= 1 → 内存泄漏

3️⃣ std::weak_ptr

💬 **背景 / 结论** 为了**防止循环引用灾难**，诞生了**std::weak_ptr**

它的**核心特性时：**

- 不增加引用计数
- 不能直接访问资源
- 必须先 lock()

破解循环引用的例子

```
struct B;

struct A {
    std::shared_ptr<B> b;
};

struct B {
    std::shared_ptr<A> a;
};
auto a = std::make_shared<A>();
auto b = std::make_weak<B>();     //不使用make_shared

a->b = b;
b->a = a;
```

weak_ptr的正确使用

```
if (auto sp = wp.lock()) {
    // 安全使用 sp
} else {
    // 对象已经被销毁
}
```

综上所述

指针	所有权	可拷贝	成本	使用场景
`unique_ptr`	唯一	❌	最低	默认选择
`shared_ptr`	共享	✅	较高	多方共享
`weak_ptr`	无	✅	低	打破环

🟢 四，迭代器

💬 **背景 / 结论** 访问stl容器，常用于遍历的修改或读取

1️⃣ 迭代器概述

迭代器的本质 = 「被严格封装的、带语义的指针抽象」

或者说

迭代器就是**我不直接给你地址，但我允许你按我规定的方式访问内存**

在迭代器的**实现**上，我们常常可以将其近似理解为**指针的封装**，且基于c++独有的运算符重载，于是我们可以像指针一样使用部分运算符

2️⃣ 迭代器类型

👉 迭代器 = 行为接口，而不是数据结构，于是

STL 把迭代器按**能力**分级：

迭代器类别	能力
InputIterator	读、前进
OutputIterator	写、前进
ForwardIterator	多次遍历
BidirectionalIterator	++ / --
RandomAccessIterator	+ n、- n

在使用上，主要针对竞赛，由于常常用到**遍历**而非具体前进后退，所以用**auto**来直接让编译器选择迭代器即可

🔵 五, 函数与模版元编程

1️⃣ std::tuple 与 std::apply

💬 **背景 / 结论** tuple往往内部可以放置多个参数，如果他们能成为一个函数的参数呢?

```
std::apply(function, t);
```

在语义上等价于:

```
f(std::get<0>(t), std::get<1>(t), std::get<2>(t));
```

如果使用了多参数+折叠表达式，那么就可以实现:

```
f(std::get<0>(t)),f(std::get<1>(t)),f(std::get<2>(t))
```

例子:

```
auto t = std::make_tuple(1, 2, 3);
auto f = [](auto... xs) {
    ((std::cout << xs << " "), ...);
};
std::apply(f, t);
```

在语义上等价于:

```
((std::cout << xs0 << " "), ((std::cout << xs1 << " "), (std::cout <<
xs2 << " ")));
```

2️⃣ 折叠表达式(c++17)

1️⃣ 一元右折叠

`(expr op ...)`
例子:

`(xs + ...)`
展开（逻辑）：

`(x1 + (x2 + (x3 + x4)))`
2️⃣ **一元左折叠**

`(... op expr)`
例子:

`(... + xs)`
展开（逻辑）：

`((x1 + x2) + x3) + x4`
3️⃣ **二元左折叠（有初值）**

`(expr op ... op init)`
例:

`(xs + ... + 0)`
展开:

`x1 + (x2 + (x3 + (x4 + 0)))`
4️⃣ **二元左折叠（有初值）**

`(init op ... op expr)`
例:

`(0 + ... + xs)`
展开:

`(((0 + x1) + x2) + x3) + x4`
注意!!!: op指的是**任意运算符**，要结合常见自己选择

3️⃣ std::function<Type0(Type...)>

语法:

- Type0: 函数返回值类型
- Type...:函数接受的参数

💬 **背景 / 结论** 功能概括：一个**可以装下任意可调用对象的盒子**

💬 **背景 / 结论** 如lambda,普通函数，函数对象

当然，使用它存在代价:

❌ 1️⃣ 不能内联

❌ 2️⃣ 可能发生堆分配

❌ 3️⃣ 体积大、调用慢

使用场景:

- 回调接口
- 跨模块 / 插件系统
- 运行时替换行为
- GUI / 事件系统

👉 接口稳定性 > 性能

❌ 不合适场景

- STL 算法内部
- 高频率调用
- 模版本来可以解决的地方

🟣 六，类型转换

💬 **背景 / 结论** 杜绝C语言中的强行转换，让错误在编译时发生而非运行时发生

转换	功能
static_cast	**正常、可预期的编译期转换**
const_cast	**加 / 去 const（极少用）**
reinterpret_cast	**位级重解释（危险），常用于系统底层**
dynamic_cast	**多态体系下的安全向下转型，在运行时检查继承关系是否真实存在**

第二部分：并发与线程

🔴 一：线程std::thread

💬 **背景 / 结论** std::thread = **C++ 标准库里“表示一条线程”的对象**

1️⃣ 线程的创建：

```
#include <thread> //注意加入头文件

std::thread Thread_name([一个“可调用对象”])
```

这行代码做了两件事（顺序非常重要）：

1. **创建一个 std::thread 对象**
2. **立刻启动一个新线程，执行 work()**

👉 线程不是 start() 才跑，是构造时就跑

2️⃣ 线程的结束

```
Thread_name.join();
```

1. join() = **“等这个线程执行完”**
2. 主线程（main）在这里**阻塞**,直到 work() 执行完毕

如果忘记join()

👉 程序直接 std::terminate()

3️⃣ 线程的运行检查

```
Thread_name.joinable()
```

```
~thread() {
    if (joinable()) terminate();
}
```

1. 需要线程已启动，但是还没join()或detach()
2. 返回值为bool

4️⃣ join() vs detach()

join()

- 等线程结束
- 生命周期受控
- 99% 场景首选

detach()

- 线程“放飞自我”
- 主线程不再管它
- 程序结束时可能直接被 OS 杀掉

5️⃣ 给线程传参数 例子：

```
void work(int x, std::string s) {
    std::cout << x << " " << s << "\n";
}

int main() {
    std::thread t(work, 42, "hello");
    t.join();
}
```

规则（非常重要）

💬 **背景 / 结论 参数默认是“按值拷贝”到新线程,即使传的是引用变量名**

正确传引用的方式:

```
std::thread t(f, std::ref(a));
```

二，数据的使用

背景 / 结论 线程之间需要通过**lock来防止数据竞争导致的UB行为**

1 std::mutex

背景 / 结论 mutex（互斥量）= 一把锁，一次**只允许一个线程**进入临界区

使用

```
#include <mutex>
std::mutex m;

void function() {
    m.lock();
    //操作
    m.unlock();
}
```

---

2 std::lock_guard

背景 / 结论 当然，为了防止程序员忘记unlock(),于是RAII再次登场：

```
#include <mutex>
std::mutex m;

void function() {
    {//要lock()的代码区段
        std::lock_guard<std::mutex> guard(m);
        //操作
    }
}
```

这样做可以极大的增强程序的安全性

同时，不难看出**lock_guard的本质**:

```
template<typename Mutex>
class lock_guard {
    Mutex& m;
public:
    explicit lock_guard(Mutex& m) : m(m) {
        m.lock();
    }
    ~lock_guard() {
        m.unlock();
    }
};
```

三，原子操作

背景 / 结论 原子操作：不可再被分割的操作

1 std::atomic

背景 / 结论 能把一些简单的变量操作转换为原子操作

当操作被原子化时，某些简单操作就可以不用加锁了

例子:

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

std::atomic<int> x{0};

void work() {
    x++;
}

int main() {
    std::thread t1(work);
    std::thread t2(work);

    t1.join();
    t2.join();

    std::cout << x << "\n"; // 一定是 2
}
```

需要注意的一点

💬 **背景 / 结论** atomic无法保证两两变量之间的关系

如

```
std::atomic<int> x{0};
std::atomic<int> y{0};

void f() {
    x++;
    y++;
}

if (x == y) { ... }  //👉 atomic 完全无能为力
```

维度	`std::atomic`	`std::mutex`
保护对象	**单个变量**	**一段代码**
是否阻塞	❌	✅
是否睡眠	❌	✅
性能	高	较低
能否组合多个变量	❌	✅
能否表达复杂不变量	❌	✅

🟢 四，线程的生命周期 & 参数传递陷阱

💬 **背景 / 结论** 线程最大的危险不是并发，而是：线程"活得比对象久"，即**悬空引用**

❌ **错误示例 1：引用参数 + 线程**

```cpp
#include <thread>

void work(int& x) {
    x++;
}

int main() {
    int a = 10;
    std::thread t(work, a);
    t.join();
}
```

问题：

- std::thread **默认是"按值拷贝参数"**

所以：

- a 不会被修改

✅ **正确写法：std::ref**

```cpp
std::thread t(work, std::ref(a));
```

💬 **背景 / 结论** `std::ref` 的作用是：

"告诉 thread：别拷贝，用引用"

❌ 错误示例 2：对象先死，线程还在跑

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

void work(const int& x) {
    std::cout << x << "\n";
}

int main() {
    std::thread t;

    {
        int a = 10;
        t = std::thread(work, std::ref(a));
    } // 👈 a 已经被销毁

    t.join(); // ❌ UB
}
```

❌ 错误示例 3：引用捕获 + 线程

```cpp
std::thread t;
{
    int x = 10;
    t = std::thread([&]() {
        std::cout << x << "\n";
    });
} // x 已销毁

t.join(); // ❌ UB
```

✅ **正确写法 1：值捕获**

```
std::thread t;
{
    int x = 10;
    t = std::thread([x]() {
        std::cout << x << "\n";
    });
}

t.join(); // ✓ 安全
```

✅ **正确写法 2：延长生命周期（shared_ptr）**

```
auto p = std::make_shared<int>(10);

std::thread t([p]() {
    std::cout << *p << "\n";
});

t.join();
```

💬 **背景 / 结论** 只要有线程还在工作，shared_ptr就能保证计数器不为0，延长p的存活时间

🔵 五，如何让线程“等条件”，而不是“傻等”

💬 **背景 / 结论 目标：写出不浪费 CPU、不丢信号、没有竞态的等待/通知代码。**

1️⃣ 引入condition_variable

💬 **背景 / 结论** 本质：条件变量 = **让线程在“条件不满足时休眠”，并在“条件满足时被唤醒”的同步工具**

❌错误例子：

```
bool ready = false;

void worker() {
    while (!ready) {
        // 空转等待
    }
    do_work();
}
```

导致的问题：

- ❌ 占满 CPU
- ❌ 没有同步（ready 有数据竞争）
- ❌ 多核下行为不可预测

✅解决方案

```
#include <bits/stdc++.h>

std::mutex m;
```

```
std::condition_variable cv;
std::queue<int> q;

void producer() {
    {
        std::lock_guard<std::mutex> lock(m);
        q.push(42);
    }               // 👉 解锁
    cv.notify_one(); // 👉 通知
}

void consumer() {
    std::unique_lock<std::mutex> lock(m);

    cv.wait(lock, [] {
        return !q.empty();
    });

    int value = q.front();
    q.pop();

    std::cout << value << "\n";
}

int main() {
    std::thread t1(consumer);
    std::thread t2(producer);

    t1.join();
    t2.join();
}
```

解释

- **cv.wait(lock);**
  - 原子地释放 mutex
  - 让线程休眠
  - 被唤醒后：**重新加锁 mutex**，然后返回

👉 “释放锁 + 睡觉”是一个原子动作

- **cv.wait(lock, [] {return !q.empty();})的第二个参数含义：**
  - 由于操作系统可能会毫无征兆的唤醒线程
  - 为了防止这种情况，需要加一个条件，保证异常苏醒后能重新入睡

👉 “释放锁 + 睡觉”是一个原子动作

- **为什么用 unique_lock，而不是 lock_guard?**
  - wait() 需要：解锁,再重新上锁
  - lock_guard ❌ 不能中途解锁

    - unique_lock ✅ 可以中途解锁
- **notify_one vs notify_all** | 方法 | 含义 | | ---------- | ---------- | | `notify_one()` | 唤醒 **一个** 等待线程 | | `notify_all()` | 唤醒 **所有** 等待线程 |

🟣 六，future

💬 背景 / 结论 有点类似快餐店点菜，你点了菜后，等待厨师做完饭，你领着凭证去取餐

1️⃣ 核心工具：std::packaged_task + std::future

💬 背景 / 结论 packaged_task<R()> = “一个能把结果/异常放进 future 的任务包装器”

💬 背景 / 结论

💬 背景 / 结论 future = “将来某一刻会有一个 R类型 的盒子”

示例：

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

int main() {
    std::packaged_task<int()> task([] {
        return 42;
    });

    std::future<int> fut = task.get_future();

    std::thread t(std::move(task));
    t.join();

    std::cout << fut.get() << "\n"; // 42
}
```

解释

- 使用了std::packaged_task<int()> task([] {...}),创建了一个任务，任务会将返回值放入task
- 使用了std::future fut = task.get_future(),创建了一个int变量
- = task.get_future()让fut获得了一个“凭证”，让它可以取得task中的值

🟤 七，线程池

💬 背景 / 结论 你当老板，招募多个牛马为你干活

展示一个完成了基本功能的线程池：

```
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>
```

```cpp
#include <functional>
#include <vector>
#include <future>
#include <type_traits>

class ThreadPool {
public:
    explicit ThreadPool(size_t n) : stop(false) { //explict关键字，防止隐式转换导致错误的使用，没有该关键字可以这样使用：ThreadPool a = 5
        workers.reserve(n);      //创建若干个工人位
        for (size_t i = 0; i < n; ++i) {
            workers.emplace_back([this] {  //调用this->worker_loop()创建工人，并放入工位
                worker_loop();             //让工人开始待命
            });
        }
    }

    ThreadPool(const ThreadPool&) = delete;
    ThreadPool& operator=(const ThreadPool&) = delete;

    template<typename F, typename... Args>   //难点！！
    auto submit(F&& f, Args&&... args)       //老板下达任务的函数，submit()的第一个参数传任务，后面传任务需要的参数
        -> std::future<std::invoke_result_t<F, Args...>>  //使用invoke_result_t推导任务返回值类型，返回任务结果
    {
        using R = std::invoke_result_t<F, Args...>; //起别名，方便后面用

        // 1. 把任务包装成 packaged_task
        auto task = std::make_shared<std::packaged_task<R()>>(
//用一个shared_ptr指向任务，防止任务消失
            std::bind(std::forward<F>(f), std::forward<Args>(args)...)
//同时，packaged_task放回R类型的任务结果
        );                                                          //使用std::bind()，将函数和参数转化为void()

        std::future<R> fut = task->get_future();     //生成一个“令牌”，用于后面领取结果

        // 2. 把“有返回值任务”变成 void() 丢进队列
        {
            std::lock_guard<std::mutex> lk(m);
            jobs.emplace([task]() { //将任务捕获
                (*task)();          //这行的第二个括号是task指向的函数的括号
            });
        }

        cv.notify_one();                                //任务包装结束，叫醒一个工人干活
        return fut;                                     //返回令牌，用于外部变量获取任务结果
    }

    ~ThreadPool() {
```

```
        {
            std::lock_guard<std::mutex> lk(m);
            stop = true;
        }
        cv.notify_all();              //叫醒所有工人准备下班
        for (auto& t : workers) {
            if (t.joinable()) t.join();
        }
    }

private:
    void worker_loop() {
        while (true) {                    //让工人一直待命或工作
            std::function<void()> job;  //工人的任务，用函数存放

            {
                std::unique_lock<std::mutex> lk(m);  //要使用job_queue
了，上锁！
                cv.wait(lk, [this] {                  //检查是否领取到了工
作，否则睡觉摸鱼
                    return stop || !jobs.empty();
                });

                if (stop && jobs.empty()) return;    //没任务或工厂休息，
返回

                job = std::move(jobs.front());       //领取到了任务，将任
务领取到工位
                jobs.pop();
            }

            job();                            //干活
        }
    }

    std::vector<std::thread> workers;
    std::mutex m;  //为jobs的操作上锁
    std::condition_variable cv;
    std::queue<std::function<void()>> jobs;
    bool stop;     //工头：操作工厂的工作状态
};
```

使用示例：

```
#include <iostream>

int main() {
    ThreadPool pool(4);

    auto f1 = pool.submit([] {
        return 1 + 2;
    });

    auto f2 = pool.submit([] {
        return std::string("hello");
    });
```

```
    auto f3 = pool.submit([] {
        throw std::runtime_error("error!");
        return 0;
    });

    std::cout << f1.get() << "\n";
    std::cout << f2.get() << "\n";

    try {
        f3.get();
    } catch (const std::exception& e) {
        std::cout << "caught: " << e.what() << "\n";
    }
}
```

八，CPU的乱序执行 & 编译器优化 & cache的延迟可见

背景 / 结论 效率提升，但是可能导致结果错误

c++的**memory_order**:

顺序	直觉理解
`relaxed`	只保证原子性，不保证顺序
`acquire`	“我之后的读，必须在这个 load 之后”
`release`	“我之前的写，必须在这个 store 之前”
`seq_cst`	“所有线程看到一个全局统一顺序”

规则

1 seq_cst

这是程序的**默认**规则，它的性质：

- 最安全
- 最符合直觉
- 性能损失在绝大多数程序里可以忽略

---

2 std::memory_order_release & std::memory_order_acquire

- **release**:

背景 / 结论 “在我之前写的东西（data），

对之后 acquire 的线程都可见”

- **acquire**:

背景 / 结论 “在我之后读的东西（data），

一定是最新的”

3️⃣ relaxed

💬 **背景 / 结论** 存在乱序优化

使用:

```
#include <bits/stdc++.h>

std::atomic<int> a{3};

int main() {
    a.store(5,std::memory_order_xxx);
}
```

第三部分：工程化

🔴 错误处理

决策表:

场景	推荐方式	原因
构造函数失败	**异常**	构造函数不能返回错误码
资源获取失败	**异常**	RAII 自动清理
极少发生的错误	**异常**	不污染主逻辑
高频分支（if/else）	**返回值**	性能 & 可读性
底层系统 / 图形 API	**返回值**	C 接口风格
并发任务	**future + 异常**	自动跨线程传播

维度	Java	C++
异常是否是语言核心	是	是
是否有“受检异常”	✅ 有	❌ 没有
是否强制声明 throws	✅ 必须	❌ 不存在
是否参与函数签名	是	否
是否影响性能模型	较少	**非常明显**
是否用于“流程控制”	偶尔有人这么干	**绝对不该**

使用**抛异常**的语法： Java:

```
try {
    risky();
} catch (Exception e) {
```

```
    handle(e);
}
```

c++:

```
try {
    risky();
} catch (const std::exception& e) {
    handle(e);
}
```

1️⃣ 普通异常（最常见）

```
void f() {
    throw std::runtime_error("error");
}

int main() {
    try {
        f();
    } catch (const std::exception& e) {
        // 程序继续
    }
}
```

2️⃣ 自定义异常

```
struct MyError {};

try {
    throw MyError{};
} catch (const MyError&) {
}
```

3️⃣ 构造函数抛异常

```
try {
    File f("no_such_file");
} catch (...) {
}
```

⚠️ **C++ 中"catch 不了、一定终止"的情况**

❌ 情况 1：noexcept 函数里抛异常

❌ 情况 2：析构函数在异常传播中再抛异常

❌ 情况 3：线程函数里抛异常，没人接

❌ 情况 4：抛了异常，但没被任何地方 catch

🟠 一些修饰关键字的工程用法

noexcept:

告诉编译器我不会出现抛异常，否则程序直接终止

[[nodiscard]]:

你要是调用了却不看返回值，我就警告你

🟡 c++中一点点“现代风格补丁”

1️⃣ std::optional

💬 **背景 / 结论** std::optional 表示：**我可能给你一个 T，也可能什么都没有(返回 nullptr)。**

2️⃣ std::span

💬 **背景 / 结论** std::span 表示：**给我一段 连续的 Vertex，我不管它从哪来，也不拥有它**

3️⃣ enum class

4️⃣ using

1️⃣ 2️⃣ 3️⃣ 4️⃣ 5️⃣ 6️⃣ 7️⃣ 8️⃣ 9️⃣ 🔟 🔴 🟠 🟡 🟢 🔵 🟣 🟤 ⚫ ⚪ ✅ ❌ ⚠️ ❗ ‼️

emoji_website