C++11 常用新特性介绍

万里红中央研究院

2022 年 3 月 7 日

目录

1 前言

众所周知，C++是由 C 语言发展而来的，最早的 C++是作为带类的 C 而出现的，事实上早期的 C++编译器，是先把 C++的代码转换成 C 语言的代码，然后使用 C 语言的编译器进行编译的。

跟所有软件一样，C++语言本身也是一个软件，用户是遍及全球的程序员，同样程序员们在日常工作中会发现目前正在使用的 C++有各种不便之处，会提出各种需求，所以 C++也在不断的进化，增加各种新的关键字、标准运行库等新特性。另外 C++在发展过程中也会借鉴各种现代语言的一些优点，也会以新特性的形式增加到新版本的 C++中。C++里的各种新特性，都是为了解决某一个或者某一类问题而增加的，如果熟练掌握这些新特性，会提升我们的开发效率，提高代码质量，并且能让写出来的代码更加精简和整洁，同时也能在使用过程中体会到优质设计带来的美感。

一般我们所说的 C++新特性，指的是 C++11 以后续版本中引入的各种新的关键字和标准库。从 C++11 这个名称可以看出，这是 2011 年引入的 C++标准，从时间上看其实已经有 10 多年了，本身已经不是新技术了，C++11 之后又出现了 C++14、C++17、C++20 以及最新的 C++22 了，所以我们更应与时俱进，紧跟时代，持续学习，不断提升自己，与公与私都是非常有好处的。

2 常用类型及关键字

2.1 基本类型

首先介绍一下 C++11 之后引入的基本类型。

早期由于某些原因，在不同平台、不同编译器下，对于 int、long 这样的类型，长度可能不一样，这给我们开发可移植的、跨平台的软件带来极大的麻烦，经常会出现某个软件在 Windows 下运行的很好，在 Linux 下编译之后却出现异常的情况，因此如果一个软件想实现跨平台，很少会直接使用 int、long 这样的类型，一般都会自已定义做一些定义，保证在任何平台、任何编译器下，每个类型的字节数都是固定的。

为此从 C++11 开始，标准库里提供了标准的基本类型定义，从类型的名称中就能明确知道某个标准类型的字节数、是否有符号、以及类型信息，保证在任何平台、任何编译器下，这个类型的字节数都是一样的。

例如 uint32_t，表示这是一个 32 位无符号整数的类型，类似的还有诸如 int8_t、int16_t、uint64_t 等，具体定义可以看 C++标准库中的 stdint.h 文件。

2.2 nullptr

以前我们用来表示空指针，一般用 NULL。在 C++里，NULL 一般被定为 0，例如：

#define NULL 0

这带来一个问题是，由于 NULL 就是 0，因此这个 NULL 无法区分到底是一个指针类型？还是一个整数类型，有时候会带来一些不便，例如在涉及到函数重载的时候，有时候编译器会无法判断最终到底应该调用的哪个函数。

nullptr 是 C++11 中新增件的关键字，表示的是一个真正的空指针，是一个明确的

空指针，是 std::nullptr_t 类型的（constexpr）变量，使用 nullptr 的话，在编译过程中不会引发任何歧义。

2.3 一些常用的 C++标准库

C++11 及后续版本中，标准库中增加了很多新的库，既然这些库是 C++标准中的，意味着任何一个符合标准 C++的实现中都会有同样的库，这对我们开发跨平台的产品带来极大的便利，在很多时候我们没有必要自己再对一些基础功能做封装，直接使用 C++标准库中提供的功能即可。

我们以前为了实现跨平台而做的大量的封装的工作，在新的 C++标准库里都已经提供了更好的封装，直接用就行。在需要跨平台的时候，我们应该养成这样的习惯：**需要用到某类基础功能之前，先看一下公司的基础库有没有提供，如果没有的话再一下 C++标准库里有没有，如果 C++标准库里没有，那再看一下 boost 里有没有，如果 boost 里也没有，在开发大群里喊一声，公共库统一提供。**

下面介绍一些以前经常自己封装，但现在 C++标准库中已经直接提供的常用的库：

- **mutex 及 lock_guard**

  这是一个跨平台的互斥锁，用于做简单的线程同步。

  虽然名字叫 mutex，但是在 Windows 上，std::mutex 实际上是临界区（CRITICAL_SECTION），但要注意的是，Windows 里的 CRITICAL_SECTION 默认是支持递归加锁的，而 std::mutex 默认不是递归加锁的，也就是说即使是在同一个线程中，如果两次去锁 std::mutex，第二次是进不去的，会死锁。如果想达到和临界区一样的效果，可以使用 std::recursive_mutex（递归锁）。

  但要注意的是，递归锁这种用法，其实是不太规范的，本身我们在编写代码过程中就

应该尽量避免出现递归锁的情况。

如果我们平时喜欢自己去调用 Lock/Unlock，或者 EnterCriticalSection、LeaveCriticalSection 这些函数去加锁和解锁的话，一定要改变习惯，使用 C++的方式来自动的完成加锁和解锁，例如我一般会这么用：

```
using auto_locker = std::lock_guard<std::mutex>;
std::mutex m_mutex;
void Test(void)
{
    auto_locker    locker(m_mutex);
    // .....
}
```

上面的 Test 函数中就是一个使用 lock_guard 完成 mutex 自动加锁和解锁的例子。但要注意的是，这种直接在函数头部用自动加锁的方式，其实是有很大隐患的，原因是加锁的范围太大了，可能这个函数执行时间很长，那由于这把锁的存在，很可能会导致整个程序性能的大幅下降，甚至导致死锁。正确的做法应该是只在必不可少的地方加锁，用完之后尽快把锁释放掉，例如：

```
Base::common_result _Policy::CPolicyCenterBase::InternalRegister(const char_type*
{
    if (Base::String::IsEmpty(name) || handler == nullptr) return e_invalidarg;

    do
    {
        _auto_locker __locker(m_mutex);
        m_container.insert(std::make_pair(name, handler));
    } while (0);

    return s_ok;
}
```

在这个函数里，只有在使用到类成员变量的地方才加上锁保护了一下，别的地方因为

变量都在堆栈中，因此不存在多线程安全问题，不需要保护。

总而言之，写多线程相关的函数时，应该非常明确的知道哪些东西必须要加锁保护，哪些东西不需要保护，既要避免保护不充分，也要避免过度保护。

- **atomic**

众所周知，如果某个变量可能会被多个线程使用，我们应该设法保护这个变量不能被同时访问，避免出现多线程安全问题，最长用的方式就是加锁。

但如果我们只是要保护一个 int、bool 之类的基本类型，为了能安全的使用这些数据，每次都 lock 和 unlock 一下，肯定会觉得很麻烦，甚至会有杀鸡用牛刀的感觉，但如果不保护，又会觉得心里不安，良心上接受不了。

这种感觉非常正常，全世界所有优秀的程序员都会有这种感觉，好在 C++里可以通过模板的方式来提供通用的解决方案，让我们安全的使用各种类型。

std::atomic 就是用来做这个的，各种类型以模板参数的形式传入 atomic 总，例如：

std::atomic<uint32_t>、std::atomic<bool>等。

也可以直接用 std::atomic_uint32_t、std::atomic_bool 这样的用法，但要注意的是，个别编译器可能并没有很好的遵循 C++标准，可能会出现类型找不到的问题，遇到这种问题的时候，我们改用 std::atomic<uitn32_t>即可。

- **filesystem**

filesystem 是一个功能非常强大，并且使用非常方便的类库，凡是涉及到文件相关操作的，在 filesystem 库中都可以找到想要的。例如判断文件是否存在、遍历文件夹、一次性创建多级目录等。

需要注意的是，filesystem 中的函数，如果出现失败，默认是会抛出异常的，所以使用的时候需要加上捕获异常的代码。如果不想用异常，filesystem 里大多数函数同时也

提供了不抛异常的版本，以创建目录为例，有两个不同的实现，分别为：

这种用法会抛异常

```
fs::create_directories("c:\\test\\test2\\test3\\");
```

这种用法不会抛出异常

```
std::error_code ec;
fs::create_directories("c:\\test\\test2\\test3\\", ec);
```

- **thread**

跨平台的多线程封装类，结合 lambda 和 shared_ptr 或者 unique_ptr，能实现性能非常高并且非常安全的多线程。

具体用法后续专门讲多线程的时候再详细讲。

- **智能指针（shared_ptr、unique_ptr 等）**

智能指针的概念，大部分人多少都有些了解，但很多人其实并没有习惯使用智能指针。熟练使用智能指针，能带来可靠性的极大提高，如果用的好的话，还能带来程序运行性能的提升。我们应该习惯使用智能指针、并且熟练使用智能指针，同时应该做到一点：**在代码中不要直接出现任何动态分配内存的代码，new 和 malloc 之类的都最好不要用，改用智能指针，这样会避免很多的麻烦，同时不会增加额外的开销。**

所有智能指针的原理都类似，都是利用 C++析构函数的机制自动的完成资源的清理，但是在具体实现细节上有不少差别，这也使得在不同场合中使用不同智能指针，能起到最大的效果。

本章节简单介绍一下智能指针，后续会专门详细讲各种智能指针。

3 auto 和 decltype

这两个 C++11 里新增的关键字，是 C++新特性中最受欢迎的两个关键字，合理使用这两个关键字，能让我们的代码更加简洁。

和动态语言不同的是，C++的 auto 和 decltype 是在编译过程中起作用的，可以认为是编译器给程序员提供的遍历，避免程序员去写复杂的类型。

对于一些简单的类型，例如 int 之类的，没有太大必要使用 auto 来代替，使用 auto 最方便是那些使用了模板的场合，尤其是有些比较复杂的模板，要想一次性正确的把对应的某个类型写出来，还是相当麻烦的，例如我们要写 std::vector< std::string> 这个容器的迭代器类型，其实是相当麻烦的，就算写出来的代码，感官上也非常丑陋，而如果使用 auto，代码就整洁多了。在这种场合，其实即使用了 auto 我们也明确知道这是个迭代器，只是不想写这么多代码而已。

多说一句，在 C++20 里，支持 auto 作为参数传入函数，很多时候可以代替模板，例如我们要实现一个支持各种类型的加法函数，使用模板的话我们会这么写：

```
template < typename T1, typename T2 >
__inline T1 add(T1 v1, T2 v2)
{
    return v1 + v2;
}
```

如果在 C++20 里，我们可以直接这么写：

```
__inline auto add(auto x, auto y)
{
    return x + y;
}
```

很显然第二种写法会让程序更加简洁和优雅，也容易理解。

虽然是在 C++20 里才支持，但之所以要在这里提一句，是希望大家能与时俱进的了

解各种新技术，通常新技术的出现都是为了解决以往的一些问题，因此新技术的出现会带来效率的提高，成本的降低，或者让生活更加美好，掌握新技术也能让我们的技术生命永远年轻，避免被时代抛弃。

auto 一般用来方便的定义变量并赋值，而 decltype 则是用来根据某个输入的变量来推导出变量的类型，以便把这个类型用在别的需要用到这个类型的场合。有时候我们回想，我都知道这个变量了，我难道还不知道它的类型吗？这话固然没错，但有些变量我们虽然在用，但要写出这个变量对应的类型，也是比较麻烦的，典型的例子就是函数指针类型，如果直接写的话还是有一定难度的，并不是人人都能熟练的写出来。如果要求写一个类成员函数的指针类型，那更加难写，而有了 decltype 的话，这一切就非常简单了，如图：

```
// 连接回调函数模板
extern "C" common_result STDCALLTYPE ConnectCallbackTemplate(
    int32_t nConnectResult, void* pContext, IConnection * pConnect);

using CONNECT_CALLBACK_TYPE = decltype(&ConnectCallbackTemplate);

// 本结构体中的所有字符串，一律为UTF8编码
typedef struct _ConnectParam
{
    // 服务器地址
    const char* m_strServerAddress;

    // 回调函数
    CONNECT_CALLBACK_TYPE m_connect_callback;

    // 回调函数里会把这东西传回去
    void* m_pContext;

    // 现在我也不知道会有啥参数，先保留一些字节
    const void* m_pReserved[16];
}CONNECT_PARAM;
```

在上图中，要写一个回调函数指针的类型，直接定义一个模板函数，然后用 decltype，就可以了。万一要修改回调函数指针类型，只需要改一下模板函数就可以了。

使用的人用起来也很简单，只需要把模板函数拷贝过去，修改一下名称就可以了。

4 std::move

不知大家以前有没有这样的感觉，有时候为了把一个类作为参数传递到别的地方，会不得不使用拷贝构造函数进行一次数据的拷贝，此时会觉得很郁闷，例如这样一个过程：

```
CTest src = xxxx;        // 原始的类，里面有一个指针，指向一个缓冲区
std::vector<CTest> vec;
vec.push_back(src);    // 此处会发生一次数据的拷贝，通过 CTest 的拷贝构造函数
                       // 构造一个新的对象，并把新对象放到 vector 中。
```

很显然，由于 CTest 中存在一个指针指向一个缓冲区，这个拷贝构造函数执行的性能不可能高，我们可以想象的到，在这个过程中会发生以下行为：

1）在新实例中的拷贝函数内部重新分配内存；

2）把老实例中的数据拷贝到新实例中；

3）由于 src 这个实例不再使用了，因此后面还会发生 src 这个实例的析构，在析构函数中释放内存；

众所周知，分配内存、拷贝和释放内存这几个动作，都是耗时比较多的，网上很多关于 C++性能低的言论，并非完全空穴来风。我们都知道，C++性能实际上是非常高的，之所以有时候会给人性能低的印象，很多时候来自于这类毫无价值的拷贝。

在很多时候，真是场景下发生的事情都是，老的实例的数据需要转移到一个新的实例中去，例如把类的实例作为返回值返回，或者需要把类的实例放到一个 STL 的容器中，诸如此类。

在 C++11 之前，STL 里的很多类都会有一个 swap 函数，swap 的作用是在两个类之间进行交换，一般用在一个老类和一个新类之间进行交换，如果我们去看内部实现的

话，内部对指针的处理，是直接交换两个指针的值，并没有发生数据的拷贝。交换之后新类的内容和老类，而老类的内容变成新类。由于一般使用场景中，最初的那个新类是个空类，因此交换之后老类的析构函数中，也就不需要做太多事情了，因此此时内部的指针类型的成员变量已经是空指针了。

swap 的机制在 C++里是一个挺巧妙的设计，在前面里子中提到的场景中，如果使用了 swap，性能会大幅提升，如果数据量比较大的话，性能提高上百倍乃至上千倍也不是不可能，所以以后如果再听到有人说 C++性能不如 C 语言高，我们可以直接怼回去：那是因为你不会用 C++。

虽然 swap 能解决这个问题，但毕竟不是太方便，因此在 C++11 中增加了一个叫“**右值引用**”的机制。右值引用这个词从字面上看有点晦涩，不太好理解，这里从实际使用场景出发，用通俗的方式来描述右值引用这东西在 C++11 中起了什么作用。

前面提到，C++的拷贝构造函数是导致 C++给人性能不好的元凶之一，虽然我们可以用 swap 来解决问题，但这不符合习惯。那有什么办法，能让我们还是使用类似于拷贝构造函数的机制，同时又能实现 swap 这样的性能呢?

我们肯定不能直接在拷贝构造函数上动手脚，原因如下：

- 标准的拷贝构造函数的参数是 const 的，无法修改；

  标准构造函数的定义形如： CTest(const CTest & __s)，是一个 const 的引用参数

- 从道理上讲，在拷贝构造函数里，应该只干拷贝的事情；

既然我们在拷贝构造函数里做不到这一点,那我们何不增加一个特殊的构造函数呢?

事实上所谓的右值引用，在类上面就是这么做的，在 C++11 里，除了传统的构造函数和拷贝构造函数之外，还增加了一个新的构造函数，称之为：**移动构造函数，**这个构造

函数的定义为：

CTest(CTest && __s);

注意，和拷贝构造函数不同的是，这个移动构造函数里，参数不是 const 的，并且有两个 && 符号，在这个构造函数里，我们可以把 __s 参数里的值，直接复制给新的类中的成员那函数，然后 __s 里原有的值恢复成默认即可，这样 __s 析构的时候，看到里面都是默认值，就不用真的去析构了（正常来说我们析构的时候，如果要释放指针，应该先判断指针是否为空，不为空再释放，因此 __s 里如果有指针，实际上并不会发生释放内存的行为）。

那如果调用类的移动构造函数呢？在 C++11 里提供了一个 std::move 的函数，用来实现这个 **“移动”** 的操作。注意，这里的移动我特别加了双引号，之所以加了双引号，是因为 std::move 其实本质上是一个移动的语义，本身并没有真的去做移动的事情，而仅仅只把一个左值强制转换成了一个右值。我们可以看一下 C++标准库中 std::move 的实现：

```
// FUNCTION TEMPLATE move
template <class _Ty>
_NODISCARD constexpr remove_reference_t<_Ty>&& move(_Ty&& _Arg) noexcept {
    return static_cast<remove_reference_t<_Ty>&&>(_Arg);
}
```

其实里面就是一个简单的类型转换，本身并没有真的去做移动。这个转换的目的是为了能让类通过移动构造函数来构造。

在多线程或者异步的场景中，移动机制和智能指针里的 unique_ptr 搭配使用，能大幅降低代码复杂度，并且能大幅提升并发的性能，这个后面单独讲。

5 Lambda 表达式

6 智能指针

6.1 unique_ptr

6.2 shared_ptr

7 多线程及异步

7.1 C++标准库中的多线程

7.2 智能指针在多线程中的应用

7.3 我们应该如何使用锁

lock_guard

7.4 安全的单实例

初始化问题