C++ 进阶知识介绍

2023/7/14 DragonAura

编译与链接

在程设课上，我们运行一个 C++ 程序的步骤通常是这样的：打开 Visual Studio 2008，在文件中写好程序，然后点击“开始调试”或者“开始执行（不调试）”，一个黑色的方框就会弹出来。

实际上，从 C++ 源代码文件到可执行文件的过程是十分复杂的，Visual Studio 等现代化的 IDE（Integrated Development Environment，集成开发环境）掩盖了程序构建的复杂流程。本节我们就以 linux 平台上的 C++ 程序为例，简略介绍 C++ 工程中的一些概念。

从 .cpp 到 .exe

C/C++ 程序生成一个可执行文件的过程可以分为 4 个步骤：**预处理（Preprocessing）**、**编译（Compiling）**、**汇编（Assembly）**和**链接（Linking）**。

编译工具

针对不同平台，各大厂家设计了不同的 C++ 编译工具：

- MSVC：是微软开发的 C++ 编译器，主要用于 Windows，Visual Studio 中内置了 MSVC
- GCC：是 GNU 开发的编译器，支持 C、C++、Fortran、Go 等多种语言
- 其他：Clang、NVCC...此处不再赘述

在 Linux 上，可以通过

```
sudo apt-get install gcc g++
```

来安装 GCC 编译器。

预处理

假设我们有如下代码：

```
// hello.cpp
#include <iostream>

void hello()
{
    std::cout << "Hello, World!" << std::endl;
}
```

```
// main.cpp
void hello();

int main()
{
    hello();
    return 0;
}
```

预处理阶段，C++ 执行宏的替换、头文件的插入、删除条件编译中不满足条件的部分

```
g++ -E hello.cpp -o hello.i
```

编译

C++ 程序在编译阶段会将 C++ 文件转换为**汇编文件**。

```
g++ -S hello.i -o hello.s
g++ -S hello.cpp -o hello1.s # 可以直接从 .cpp 生成
```

汇编

汇编文件经过汇编生成目标文件（机器码），每一个源文件对应一个**目标文件**。

```
g++ -c hello.s -o hello.o
g++ -c main.cpp -o main.o # 可以直接从 .cpp 生成
```

> 二进制文件不能直接打开

链接

将每个源文件的目标文件链接起来，形成一个**可执行程序文件**

```
g++ hello.o main.o -o main
g++ hello.cpp main.cpp -o main # 可以直接从 .cpp 生成
```

总结：

- `-E`：预处理
- `-S`：编译
- `-c`：生成
- `-o`：指定输出文件名称

> 一般地，我们约定 `.i` 为预处理后的文件，`.s` 为汇编文件，`.o` 为目标文件

静态库与动态库

我们可以将 C++ 源代码封装为库，以便复用、封装细节、防止源代码泄露等。根据其行为的不同，可以将库分为静态库（static library）和动态库（shared library）。

静态库

静态库的代码在编译的过程中会被载入到可执行文件中，这样的好处为代码执行时不再需要静态库本身，缺点则是生成的可执行文件体积会比较大。

Linux 下，静态库的后缀通常为 `.a`，命名方式通常为 `libxxx.a`；Windows 平台下，则为 `libxxx.lib`。

```
ar crv libhello.a hello.o
g++ -static main.cpp -L . -lhello -o main_static
```

动态库

动态库在编译时并不会被载入到目标代码里，只有运行时才被载入，这样带来了一个好处：不同的程序调用相同的库时，只需要有一份动态库，减小了各个模块之间的耦合度，同时也减小了可执行文件的体积。然而，这也要求用户的电脑上需要同时拥有可执行文件和动态库，也有可能因为版本不匹配等问题发生 DLL Hell 等问题。

Linux 下，动态库的后缀通常为 `.so`，命名方式通常为 `libxxx.so`；Windows 平台下，则为 `libxxx.dll`。

```
g++ hello.cpp -I ./ fPIC -shared -o libhello.so
sudo move libhello.so /usr/local/lib
sudo ldconfig
g++ main.cpp -L . -lhello -o main_shared
```

小结

由于时间所限，还有很多有趣的内容我们没有涉及：

- gcc/g++ 有着丰富的命令行参数设置，比如程序优化、C/C++ 语言标准设置等。
- 在本节中，我们只介绍了如何在 Linux 平台上生成和使用静态库、动态库。实际上，利用 Visual Studio 也可以便捷地在 Windows 平台上生成静态库、动态库。
- ...

略过上述内容不会对我们的教学产生太大影响。感兴趣的同学可以参考以下文档：

- GCC 官网
- learn cpp 一份新手友好的 C++入门文档。
- 演练：使用 Visual Studio 创建并使用静态库
- 演练：使用 Visual Studio 创建并使用动态库

特别鸣谢：

- 吴昊：《复变函数与数理方程讲义》，清华大学

Makefile & CMake

上一节中，我们讲解了 `g++` 的基本操作，并初步用 `g++` 编译了 `hello.cpp` 和 `main.cpp`。但是实际编程中，我们可能面临如下问题：

- 项目中的 `.h` 文件和 `.cpp` 文件十分繁多。
- 各 `.h` 文件、`.cpp` 文件的依赖关系十分复杂。
- 多文件可能会出现重复编译的情况，拖慢编译速度。
- ...

为了解决这些问题，`makefile` 和 `CMake` 应运而生。

Makefile

通过如下命令安装 `make` 工具：

```
sudo apt-get install make
```

`makefile` 文件描述了 `C/C++` 的编译规则，可以用于指明源文件的编译顺序、依赖关系、是否需要重新编译等。考虑：

```
// ./include/hello.h
#pragma once

#ifndef HELLO_H
#define HELLO_H

void hello();

#endif
```

```
// ./src/hello.cpp
#include "hello.h"
#include <iostream>

void hello()
{
    std::cout << "Hello, World!" << std::endl;
}
```

```
// ./src/main.cpp
#include "hello.h"
int main()
{
    hello();
    return 0;
}
```

我们只需要写一个 makefile 文件，然后执行 `make`，即可完成编译工作。

```
# ./makefile

INCLUDES = -I ./include

CXXFLAGS = -std=c++17 -O2

main: main.o hello.o
    $(CXX) $(CXXFLAGS) $(INCLUDES) -o $@ $^

main.o: ./src/main.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDES) -o $@ -c $<

hello.o: ./src/hello.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDES) -o $@ -c $<
```

```
15
16 .PHONY: clean
17 clean:
18     rm main.o hello.o main
```

由此可以看到，makefile 的可读性较差，我们很少直接手动编写 makefile。但是，许多项目依然需要使用 makefile 来构建程序，因此了解 makefile 是很有必要的。一般情况下，有如下几条命令：

- `make`：按照 makefile 的内容编译整个工程。通常，我们可以使用 `make -j` 来加快编译速度（并行启动多个任务），但是直接 `make -j` 会不加限制地并行执行若干任务，这有较大概率导致系统挂掉。因此，使用 `make -jn`（将 n 替换为数字），可以限制任务数量为 n。一般情况下，我们使用 `make -j$(nproc)`，将并行执行的任务限制在处理器核心数量上。
- `make clean`：可以清除编译生成的中间文件
- `make install`（一般需要 root 权限）：可以安装编译好的可执行文件（默认路径为 `/usr/local/bin`，安装好后可以在命令行中直接调用）、库（默认路径为 `/usr/local/lib`，安装好后可以直接链接）、头文件（默认路径为 `/usr/local/include`，安装好后可以直接使用 `#include <xxx.h>` 引用）。

CMake

makefile 存在以下问题：

- 代码可读性极差，难以维护。
- 语法复杂。
- 跨平台性差。比如 linux 平台下的 makefile 在 Windows 下可能无法工作，因为 Linux 的删除指令是 `rm`，windows 下的删除指令是 `del`。
- ...

因此，现在我们多采用 CMake 来管理项目。CMake 是一种跨平台的编译工具，可以用较为简洁易读的语法描述 C++项目的编译、链接、安装过程等，在现代 C++项目上得到了广泛应用。

在 Linux 上，我们可以通过如下方式安装 `cmake`：

```
1 sudo apt-get install cmake
```

我们使用 CMake 一般分为两步：

- 使用 `CMakeLists.txt` 生成 `makefile`
- 使用 `makefile` 编译项目

简单 CMake 语法

依然考虑上一讲中的文件目录，同时增加：

```
# ./CMakeLists.txt

cmake_minimum_required(VERSION 3.5)
project(hello)

set(INCLUDE_DIR ${PROJECT_SOURCE_DIR}/include)

aux_source_directory(./src CPP_LIST)
add_executable(main ${CPP_LIST})

target_include_directories(main PUBLIC ${INCLUDE_DIR})

```

这等价于

```
# ./CMakeLists.txt

cmake_minimum_required(VERSION 3.5)
project(hello)

add_executable(main ./src/hello.cpp ./src/main.cpp)

target_include_directories(main PUBLIC ${PROJECT_SOURCE_DIR}/include)
```

此时使用 `cmake CMakeLists.txt`，即可生成我们需要的 `makefile` 文件。但是通常，我们并不直接生成，而是采取一些更为优雅的方法，在 `./build` 下生成。

> 语法小结：
>
> `cmake_minimum_required`：版本要求，一般不重要。可以通过 `cmake --version` 查看本机的 cmake 版本。
>
> `project`：设置项目名称
>
> `add_executable`：生成可执行文件
>
> `set`：设置变量
>
> `target_include_directories`：指定包含的头文件

库相关操作

同样的，我们可以使用 `message` 来在终端打印信息。例如：

```
# ./CMakeLists.txt

cmake_minimum_required(VERSION 3.5)
project(hello)

set(INCLUDE_DIR ${PROJECT_SOURCE_DIR}/include)

aux_source_directory(./src CPP_LIST)
add_executable(main ${CPP_LIST})

target_include_directories(main PUBLIC ${INCLUDE_DIR})

```

```
message("Succeed!")
message(${INCLUDE_DIR})
```

此外，我们同样可以用 CMake 生成库、链接库。方法如下：

```
cmake_minimum_required(VERSION 3.5)
project(hello)

add_library(hello_static STATIC ./src/hello.cpp)
target_include_directories(hello_static PUBLIC
${PROJECT_SOURCE_DIR}/include)
add_library(hello_shared SHARED ./src/hello.cpp)
target_include_directories(hello_shared PUBLIC
${PROJECT_SOURCE_DIR}/include)

add_executable(main ./src/main.cpp)
target_link_libraries(hello PRIVATE hello_static)
```

> 语法小结：
>
> `add_library` 生成库
>
> `target_link_libraries` 链接库

第三方库的使用

除此之外，我们还可以链接第三方库，方法如下（以 THUAI6 Cpp 接口代码为例）：

```
cmake_minimum_required(VERSION 3.5)

project(THUAI6_CAPI VERSION 1.0)

set(CMAKE_CXX_STANDARD 17)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O2 -pthread")

aux_source_directory(./API/src CPP_LIST)
aux_source_directory(./proto PROTO_CPP_LIST)

find_package(Protobuf CONFIG REQUIRED)
find_package(gRPC CONFIG REQUIRED)

message(STATUS "Using protobuf ${Protobuf_VERSION}")
message(STATUS "Using gRPC ${gRPC_VERSION}")

add_executable(capi ${CPP_LIST} ${PROTO_CPP_LIST})

target_include_directories(capi PUBLIC ${PROJECT_SOURCE_DIR}/proto
${PROJECT_SOURCE_DIR}/API/include ${PROJECT_SOURCE_DIR}/tclap/include
${PROJECT_SOURCE_DIR}/spdlog/include)

target_link_libraries(capi
protobuf::libprotobuf
gRPC::grpc
gRPC::grpc++_reflection
```

```
gRPC::grpc++
)
```

> 语法小结
>
> `find_package` 查询第三方库，查找到之后，初始化部分变量（例如上面的 `Protobuf_VERSION`）；此外，对于安装好的库，也可以直接链接

小结

由于时间所限，还有很多内容我们没有涉及，例如：

- 设置 C++工程的语言标准、编译优化选项。
- 层级文件之间`CMakeLists.txt`的相互调用，以便应用于目录层级更加复杂的 C++工程。
- 对生成的库、可执行文件等进行安装。
- ...

略过上述内容不会对我们的教学产生太大影响。感兴趣的同学可以参考以下文章：

- CMake 官方文档
- cmake-examples 该 GitHub 仓库中有很多开箱即用的 CMake 实例。

特别鸣谢：

- 吴昊：《复变函数与数理方程》讲义，清华大学

C++ 多文件编程

依然考虑我们上面的例子，首先我们稍加改动一个单文件版本：

```cpp
#include <iostream>

// void hello(); ###removed!###

int main()
{
    hello();
    return 0;
}

void hello()
{
    std::cout << "Hello, World!" << std::endl;
}
```

显然这段程序会引发编译器报错。当编译器看到 `main` 中的 `hello()` 时，并不知道它是什么，我们要么提前声明，要么调整代码顺序。

```
main.cpp: In function ‘int main()’:
main.cpp:7:5: error: ‘hello’ was not declared in this scope; did you mean ‘ftello’?
 7 |     hello();
   |     ^~~~~
   |     ftello
```

接下来，我们回到多文件的版本：

```cpp
// hello.cpp
#include <iostream>

void hello()
{
    std::cout << "Hello, World!" << std::endl;
}
```

```cpp
// main.cpp
// void hello();  // removed!!

int main()
{
    hello();
    return 0;
}
```

这个程序由两个 `.cpp` 文件构成，编译时，编译器要么先编译 `hello.cpp`，要么先编译 `main.cpp`。无论哪一种，在编译 `main.cpp` 时都会报错，因为编译器并不知道 `hello` 是什么。

```
main.cpp: In function ‘int main()’:
main.cpp:5:5: error: ‘hello’ was not declared in this scope
 5 |     hello();
   |     ^~~~~
```

我们的解决方案更少了：由于 `hello` 的定义在另一个文件里，调整顺序是不可能的，我们只能使用前向声明，这就回到了上一节中的例子。现在，编译器就知道 `hello` 是一个函数，不会再报错。接下来就由链接器将主函数对 `hello` 的调用与 `hello` 的定义连接。如果发现 `hello` 并没有被定义，就会引发链接器报错。具体来讲，我们单独编译 `main.cpp` 而不编译 `hello.cpp`，就会发生链接期错误。

```
/usr/bin/ld: /tmp/cc7nUGZF.o: in function `main':
main.cpp:(.text+0x9): undefined reference to `hello()'
collect2: error: ld returned 1 exit status
```

头文件

随着程序越来越大（并使用更多的文件），对想使用的定义在不同文件中的每个函数进行前向声明变得越来越乏味。如果你能把所有的前向声明放在一个地方，然后在需要的时候导入它们，那不是很好吗？由此，我们引入了头文件，其主要目的是将声明传播到代码文件，允许我们把声明放在一个地方而在我们需要的时候导入它们。

例如，`#include <iostream>` 中，`iostream` 就是头文件，`std::cout` 就是在 `iostream` 头文件中声明的。

当涉及到函数和变量时，值得注意的是，**头文件通常只包含函数和变量的声明**，而不是函数和变量的定义（否则可能导致违反单一定义规则，见下）。 `std::cout` 在 `iostream` 头文件中是向前声明的，但定义为 C++标准库的一部分，它在链接器阶段会自动链接到你的程序中。

> **为什么头文件不包含定义**

```cpp
// a.h
int a = 5;
```

```cpp
// a.cpp
#include "a.h"
```

```cpp
// main.cpp
#include "a.h"
#include <iostream>

int main()
{
 std::cout << a;
 return 0;
}
```

此时会引发链接错误。原因在于，`a.cpp` 与 `main.cpp` 都包含了 `a.h`，于是两个源文件都含有 `a` 的定义，导致重定义错误。在多文件编程中，多个源文件包含一个头文件的情况是很常见的，如果在头文件中进行定义，就难免会发生上面的情况。

```
/usr/bin/ld: CMakeFiles/main.dir/src/main.cpp.o:(.data+0x0): multiple definition of `a'; CMakeFiles/main.dir/src/a.cpp.o:(.data+0x0): first defined here
collect2: error: ld returned 1 exit status
```

合理使用头文件

头文件与源文件

一般而言，头文件会与对应的源文件对应。例如，我们创立了 `hello.h`，就应当有对应的 `hello.cpp`，在头文件中声明、在源文件中定义。于是，在其他文件中（例如 `main.cpp`），我们就可以使用 `hello.h` 中的 `hello()` 函数。

使用双引号导入自定义头文件

使用尖括号时，它告诉预处理器这不是我们自定义的头文件。编译器将只在包含目录（include directories）所指定的目录中搜索头文件。包含目录是作为项目/IDE 设置/编译器设置的一部分来配置的，通常默认为编译器和/或操作系统所带头文件的目录。编译器不会在你项目的源代码目录中搜索头文件。

使用双引号时，我们告诉预处理器这是我们自定义的头文件。编译器将首先在当前目录下搜索头文件。如果在那里找不到匹配的头文件，它就会在包含目录中搜索。

因此，使用双引号来包含你编写的头文件，或者预期在当前目录中可以找到的头文件。使用尖括号来包含编译器、操作系统或安装的第三方库所附带的头文件。

显式包含所需的所有头文件

每个文件都应该明确地 `#include` 它所需要的所有头文件来进行编译。不要依赖从其他头文件中转来的头文件。

如果你包含了 `a.h`，而 `a.h` 中包含了 `b.h`，则 `b.h` 为隐式包含，`a.h` 为显式包含。你不应该依赖通过这种方式包含的 b.h 的内容，因为头文件的实现可能会随着时间的推移而改变，或者在不同的系统中是不同的。而这一旦发生，你的代码就可能不能在别的系统上编译，或者将来不能编译。直接同时显式包含 `a.h` 和 `b.h` 就能解决这个问题。

加入 header guard

如果遇到了不可避免的在头文件中定义变量/函数的情况（这通常是代码设计得有问题），那么 header guard 可以在一定程度上缓解问题。

考虑如下代码：

```
// a.h
int a = 5;
```

```
// b.h
#include "a.h"
```

```
// main.cpp
#include "a.h"
#include "b.h"
#include <iostream>

int main()
{
    std::cout << a;
    return 0;
}
```

这显然会报错：

```
In file included from b.h:1,
                 from main.cpp:2:
a.h:1:5: error: redefinition of ‘int a’
 1 | int a = 5;
   |     ^
In file included from main.cpp:1:
a.h:1:5: note: ‘int a’ previously defined here
 1 | int a = 5;
   |     ^
```

但是在 `main.cpp` 的角度看，自己不过是 `include` 了两个不同的头文件而已。因此，解决这个问题应该从头文件本身下手，即 header guard。header guard 可以保证相同的部分只被包含一次。

```
// a.h

#ifndef A_H
#define A_H

int a = 5;

#endif
```

```
// b.h

#ifndef B_H
#define B_H

#include "a.h"

#endif
```

根据我们的 C/C++ 基础可知，这样可以保证相同的部分只被包含一次。

当然，对于现代编译器，我们还可以使用另一种更简洁的 header guard：`#pragma once`

这和上述代码效果一致。但是，我们建议两种都写，或只写传统的 header guard，因为 `#pragma once` 在某些编译器上不被支持。

严禁循环包含

```
// b.h
#include "a.h"

// a.h
#include "b.h"
```

未使用 header guard 时，这样的写法会引发连锁反应：b.h 包含 a.h，后者包含 b.h，再包含 a.h.....

```
            from a.h:1,
            from b.h:1,
            from a.h:1,
            from main.cpp:1:
a.h:1:15: error: #include nested too deeply
1 | #include "b.h"
```

然而，即使使用了 header guard，这样也会出现问题：如果 b 包含 a，那么 a 包含 b 的代码则会因为 header guard 在预处理阶段就失去作用，于是 a 没能成功包含 b，则 a 无法看到其需要的 b 中的声明。

考虑

```
// a.h
#pragma once
#include "b.h"

void aa()
{
}

void a()
{
    bb();
}
```

```cpp
// b.h
#pragma once
#include "a.h"

void bb()
{
}

void b()
{
    aa();
}
```

```cpp
#include "a.h"

int main()
{
    return 0;
}
```

报错如下：

```
In file included from a.h:2,
                 from main.cpp:1:
b.h: In function ‘void b()’:
b.h:10:5: error: ‘aa’ was not declared in this scope
10 |     aa();
   |     ^~
```

尽量避免使用 using namespace std

namespace std 是标准命名空间，其中包括了 cout、cin、max、min 等一系列很有用处的函数。根据命名空间的使用规则（::），我们可以写出这样的代码：

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello!" << std::endl;
    return 0;
}
```

using 的意义是，在整个文件中对应的函数名都默认在其后的命名空间中，using namespace std 就是在整个文件范围内指定使用标准命名空间（如果不存在歧义）。

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello!" << endl;
    return 0;
}
```

当然，这能帮我们少做很多琐碎活，如果你可以确保不会使用来自其他命名空间的同名函数，就可以大胆使用 using（比如 OJ 题，它们通常只要求在一个文件完成所有内容）。但除此之外的情况，尽可能不要使用 `using namespace std`！即使一定要用，也**严禁**在头文件中使用！！

一个经典的知乎笑话：https://www.zhihu.com/question/543813788

下面我们自己来一个例子：

```cpp
#include <iostream>

using namespace std;

int cout()
{
    return 5;
}

int main()
{
    cout << "Hello!" << endl;
    return 0;
}
```

这会引发报错：

```
main.cpp: In function ‘int main()’:
main.cpp:12:5: error: reference to ‘cout’ is ambiguous
12 |     cout << "Hello!" << endl;
   |     ^~~~
In file included from main.cpp:1:
/usr/include/c++/9/iostream:61:18: note: candidates are: ‘std::ostream std::cout’
61 |   extern ostream cout;  /// Linked to standard output
   |                  ^~~~
main.cpp:5:5: note:                 ‘int cout()’
 5 | int cout()
   |     ^~~~
```

Reference

https://docs.eesast.com/docs/languages/C&C++/multi-file_programming

面向对象程序设计基础

在大一的程设课上，我们系统学习了 C++的语法，掌握了一些编写小型程序的技能。实际上，要想写出一个可读性好、可复用、鲁棒性强的程序，掌握一些基本的设计原则是十分必要的。

本讲的内容并不针对具体的某一语言，而且相比之前的一些内容，本讲的知识更需要在长期的实践中“内化”；与此同时，与软件工程相关的理论博大精深，本讲仅仅挑选一些代表性的原则，只能带领大家入门，想要了解更多还需要仔细阅读文末提供的书单~

KISS

KISS 代表着“Keep It Simple and Stupid”。KISS 原则指出，简单性应该是软件开发的主要目标，应该避免不必要的复杂性。

不过，如何界定“简单”？KISS 原则指出，为了保证代码的**灵活性**和**可扩展性**，我们可能不得不增加代码的复杂度。但除此之外，在这种问题固有复杂性的基础之上增加自制的复杂性，是十分不明智的做法——程序并非程序员炫技的场所，而应该是一件简约的艺术品。

一言以概之：如无必要，勿增实体。

Loose Coupling

Loose Coupling，即松耦合原则。这一原则指出：模块与模块之间的耦合（即相互关联的程度）应该越小越好，或者说，它们应该尽可能少地感知到对方的存在。

举一个例子吧（本例选自 *Clean C++* 一书）：

考虑你有一台电灯，和一个用于控制电灯的开关：

```
class Lamp
{
public:
    void on()
    {

    }

    void off()
    {

    }
}

class Switch
{
public:
    Switch(Lamp& lamp): lamp(lamp) {}

    void toggle()
    {
        if (state)
        {
            state = false;
            lamp.off();
        }

```

```
        else
        {
            state = true;
            lamp.on();
        }
    }
}
```

在这样的设计方法下，开关可以工作，但可能会带来一个问题：`Switch`类中包含了`Lamp`类的引用，`Switch`类与`Lamp`类之间存在着强耦合关系——`Switch`类可以感知到`Lamp`类的存在。

这种写法不仅不符合常理，而且不便于维护和扩展：试想，如果我们想要用开关控制电扇、充电器等其它电器该怎么办？难道我们需要分别设计`SwitchForLamp`、`SwitchForFan`、`SwitchForCharger`类吗？

如何解决这类耦合问题？一个方法是：将两个类之间相关联的部分抽象成一个接口（interface），第二个类此时不需要包含第一个类的实例或引用，而只需要对接口负责，从而降低耦合度，提高程序的可扩展性。

以上程序可以改写如下（在 C++中，接口可以使用虚基类实现）：

```
#include <iostream>

class Switchable
{
public:
    virtual void on() = 0;
    virtual void off() = 0;
};

class Switch
{
public:
    Switch(Switchable& switchable) : Switchable(switchable) {}
    void toggle()
    {
        if (state)
        {
            state = false;
            switchable.off();
        }
        else
        {
            state = true;
            switchable.on();
        }
    }

private:
    Switchable& switchable;
    bool state {false};
};

class Lamp: public Switchable
{
```

```
public:
    void on() override
    {
        std::cout << "Lamp is on!" << std::endl;
    }

    void off() override
    {
        std::cout << "Lamp is off!" << std::endl;
    }
};

class Fan: public Switchable
{
public:
    void on() override
    {
        std::cout << "Fan is on!" << std::endl;
    }

    void off() override
    {
        std::cout << "Fan is off!" << std::endl;
    }
};

int main()
{
    Lamp lamp;
    Switch switch1(lamp);
    switch1.toggle();
    switch1.toggle();

    Fan fan;
    Switch switch2(fan);
    switch2.toggle();
    switch2.toggle();
}
```

在以上更改中，开关与其它电器耦合的部分被抽象为一个接口`Switchable`，开关只需要对这一接口进行操作，避免了开关与具体电器类的耦合。

SOLID

SOLID 是以下五大面向对象设计原则的缩写：

- 单一功能原则（**S**ingle Responsibility Principle，SRP）
- 开闭原则（**O**pen Closed Principle，OCP）
- 里氏替换原则（**L**iskov Substitution Principle，LSP）
- 接口隔离原则（**I**nterface Segregation Principle，ISP）
- 依赖反转原则（**D**ependency Inversion Principle，DIP）。

单一功能原则

单一功能原则指出，每个软件单元（类、函数等），应该只有一个单一的、定义明确的责任。

如何界定单一责任？一个比较普适的定义是，改变该软件单元只能有一个原因。如果有多个原因，那么该单元就应该拆分。

开闭原则

开闭原则指出，软件单元（类、函数等）应该对于扩展是开放的，但是对于修改是封闭的。

具体来讲，如果我们需要给一个软件添加新的功能，我们通常不建议修改源码，而更加建议通过**继承**的方式。

里氏替换原则

里氏原则指出，派生类（子类）对象可以在程序中代替其基类（超类）对象。

换句话说，一个软件实体如果使用的是一个父类，那么也一定适用于其子类——把一个软件里面的父类都替换为它的子类，程序的行为是不会发生变化的。

利用这一原则，我们可以判断类与类之间的继承关系是否合适。

举个例子，假设我们拥有一个矩形类：

```
class Rectangle
{
public:
    Rectangle(int width, int height) : width(width), height(height) {}

    void setWidth(int width)
    {
        this->width = width;
    }

    void setHeight(int height)
    {
        this->height = height;
    }

    void setEdges(int width, int height)
    {
        this->width = width;
        this->height = height;
    }

private:
    int width;
    int height;
};
```

我们想要再新建立一个正方形类。根据初中几何知识：正方形是一种特殊的矩形——因此一种直观的想法是：让正方形类去继承矩形类：

```
class Square: public Rectangle
{
    // ...
};
```

但如果站在里氏替换原则的角度来看，这一设计是不科学的！比如我们考虑以下操作：

```
Rectangle rectangle;
rectangle.setHeight(20);
rectangle.setEdges(10, 5);
```

根据里氏替换原则，派生类对象（Square）一定可以替换基类对象（Rectangle），假如我们进行这一替换：

```
Square square;
square.setHeight(20);
square.setEdges(10, 5);
```

这时就出现了问题：

- 第一个操作会产生歧义：该操作是只改变正方形的宽（这样会违背正方形的定义），还是同时改变正方形的长和宽（这样违背函数的字面意思）。
- 第二个操作则会直接违背正方形的定义。

可以看到，派生类对象在此处替换基类对象会产生很多问题，这一继承是不科学的！

接口隔离原则

接口隔离原则指出，程序员在设计接口时应当将臃肿庞大的接口拆分成更小的和更具体的接口，让接口中只包含客户感兴趣的方法——使用多个专门的接口比使用单一的总接口要好。

换句话讲，接口约束了类的行为，是一种减轻代码耦合程度的好方法。但如果一个接口太过宽泛，可能会带来一些不必要的麻烦。举例说明：

我们想要定义一个“鸟”接口：

```
class Bird
{
public:
    virtual void eat() = 0;
    virtual void breathe() = 0;
    virtual void fly() = 0;
};
```

在此基础上实现一个鸽子类，现在一切看上去都正常：

```
class Pigeon: public Bird
{
public:
```

```
    virtual void eat() override
    {
        // ...
    }

    virtual void breathe() override
    {
        // ...
    }

    virtual void fly() override
    {
        // ...
    }
};
```

我们再实现一个企鹅类：

```
class Penguin: public Bird
{
public:
    virtual void eat() override
    {
        // ...
    }

    virtual void breathe() override
    {
        // ...
    }

    virtual void fly() override
    {
        // ???
    }
};
```

问题发生了。我们在一开始设计“鸟”这一接口时，想当然地以为所有地鸟类都会飞，却忽略了企鹅不会飞这一特例。

为了避免这样的情况发生，我们需要小心地将接口拆分：

```
class Lifeform
{
public:
    virtual void eat() = 0;
    virtual void breathe() = 0;
};

class Flyable
{
public:
    virtual void fly() = 0;
};

```

```
class Pigeon: public Lifeform, public Flyable
{
public:
    void eat() override
    {
        // ...
    }

    void breathe() override
    {
        // ...
    }

    void fly() override
    {
        // ...
    }
};

class Penguin: public Lifeform
{
public:
    void eat() override
    {
        // ...
    }

    void breathe() override
    {
        // ...
    }
};
```

如上文所示，所有的鸟类都需要呼吸和进食，我们可以大胆地将其封装为 `Lifeform` 接口，而并非所有鸟类都会飞，所以需要将其单独提取出来作为 `Flyable` 接口。在实现不同的鸟类时，我们将这些接口进行筛选组合即可。

依赖倒转原则

依赖倒转原则指出，在实际的开发场景中，类与类之间的依赖关系是十分复杂，在设计依赖关系时，高层模块不应该依赖低层模块，二者都应该依赖其抽象。

什么意思呢？考虑以下实例，一个用户在某在线网络平台上拥有一个账户，而这个账户又存储着该用户的信息。由此，两者不可避免地产生了下列的循环依赖关系——你中有我，我中有你：

```
class Account;

class Customer
{
public:
    // ...
    void setAccount(Account *account)
    {
        customerAccount = account;
    }
```

```
    // ...
private:
    Account *customerAccount;
};

class Account
{
public:
    void setOwner(Customer *customer)
    {
        owner = customer;
    }

private:
    Customer *owner;
};

int main()
{
    Account* account = new Account { };
    Customer* customer = new Customer { };
    account->setOwner(customer);
    customer->setAccount(account);
}
```

这会导致很严重的问题：首先代码的可读性由于循环依赖下降，而且两者的生命周期不相互独立——如果`Account`对象的生命周期先于`Customer`对象结束，`Customer`对象中将会产生一个空指针，调用`Customer`对象中的成员函数可能会导致程序崩溃。

而依赖倒转原则为解决此类问题提供了一套流程：

1. 不允许两个类中的其中一个直接访问另一个类，要想进行这种访问操作，需要通过接口。
2. 实现这个接口。

在本例中，我们不再使得`Account`类中包含有`Customer`类的指针，所有`Account`类需要访问`Customer`类的行为，都被定义进一个叫做`Owner`的接口中，而后，`Customer`类需要实现这个接口：

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

class Owner
{
public:
    virtual std::string getName() = 0;
};

class Account;

class Customer : public Owner
{
public:
    void setAccount(Account* account)
    {
        customerAccount = account;
    }
```

```
    virtual std::string getName()  override
    {
        // return the Customer's name here...
    }
    // ...
private:
    Account* customerAccount;
    // ...
};

class Account
{
public:
    void setOwner(Owner* owner)
    {
        this->owner = owner;
    }
    //...
private:
    Owner* owner;
};
```

经过修改之后，`Account`类将不依赖于`Customer`类。

Reference

https://docs.eesast.com/docs/languages/C&C++/OOP

小结

由于篇幅所限，我们此处略过了设计模式相关的内容，而设计模式是面向对象程序设计基础中十分重要的内容。感兴趣的同学，可以通过 https://refactoringguru.cn/ 学习。

Modern C++ 选讲

本节中，我们介绍一些常用的 Modern C++ 内容。由于时间限制，我们只精选部分最常用的内容在此处介绍。

常量

nullptr

替代 NULL。C++11 引入了 `nullptr` 关键字，专门用来区分空指针、`0`。而 `nullptr` 的类型为 `nullptr_t`，能够隐式的转换为任何指针或成员指针的类型。

```
void foo(int);
void foo(char*);

foo(0);          // 调用 foo(int)
// foo(NULL);    // 该行不能通过编译
foo(nullptr);    // 调用 foo(char*)
```

constexpr

显式声明常量表达式。const 修饰的变量**只在一定情况下**是常量表达式，这有时可能带来困扰。C++11 提供了 `constexpr` 让用户显式的声明函数或对象构造函数**在编译期**会成为常量表达式。 从 C++14 开始，`constexpr` 函数可以在内部使用局部变量、循环和分支等简单语句。

```
const int len_1 = 5; // 常量表达式
const int len_2 = len_1 + 1; // 常量表达式
constexpr int len_2_constexpr = 1 + 2 + 3; // 显式声明的常量表达式

int len = 5;
const int len_3 = len + 1; // 非常量表达式

// 使用了static_assert，当其第一个参数为常量表达式时才不会报错
static_assert(len_1, "");
static_assert(len_2, "");
static_assert(len_2_constexpr, "");
static_assert(len_3, ""); // 报错，说明len_3不是常量表达式
```

类型推导

auto

从 C++11 起, 使用 auto 关键字进行类型推导。

```
class MagicFoo {
public:
    std::vector<int> vec;
    MagicFoo(std::initializer_list<int> list) {
        // 不用再写冗长的迭代器类型名了
        for (auto it = list.begin(); it != list.end(); ++it) {
            vec.push_back(*it);
        }
    }
};

auto i = 5;              // i 被推导为 int
auto arr = new auto(10); // arr 被推导为 int *
```

从 C++ 20 起，`auto` 甚至能用于函数传参。

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

auto i = 5; // 被推导为 int
auto j = 6; // 被推导为 int
```

注意：`auto` 还不能用于推导数组类型：

```
auto auto_arr2[10] = {arr}; // 错误，无法推导数组元素类型
```

decltype

`decltype` 关键字是为了解决 `auto` 关键字只能对变量进行类型推导的缺陷而出现的，可推导表达式的类型。

```
auto x = 1;
auto y = 2;
decltype(x+y) z; // z的类型是int
```

`std::is_same<T, U>` 用于判断 `T` 和 `U` 这两个类型是否相等。

```
if (std::is_same<decltype(x), int>::value) // 为真
    std::cout << "type x == int" << std::endl;
if (std::is_same<decltype(x), float>::value) // 为假
    std::cout << "type x == float" << std::endl;
```

尾返回类型推导

C++11 引入了一个尾返回类型（trailing return type），利用 `auto` 关键字将返回类型后置。 C++14 开始可以直接让普通函数具备返回值推导。

```
// after c++11
template<typename T, typename U>
auto add2(T x, U y) -> decltype(x+y){
    return x + y;
}
// after c++14
template<typename T, typename U>
auto add3(T x, U y){
    return x + y;
}
```

decltype(auto)

C++14 引入的 `decltype(auto)` 主要用于对转发函数或封装的返回类型进行推导，它使我们无需显式指定 `decltype` 的参数表达式。

```
std::string  lookup1();
std::string& lookup2();

// C++11
std::string look_up_a_string_1() {
    return lookup1();
}
std::string& look_up_a_string_2() {
    return lookup2();
}
// after C++14
decltype(auto) look_up_a_string_1() {
    return lookup1();
}
decltype(auto) look_up_a_string_2() {
    return lookup2();
```

```
}
```

控制流

基于范围的 for 循环

C++11 引入了基于范围的循环写法。

```
std::vector<int> vec = {1, 2, 3, 4};
for (auto element : vec)
    std::cout << element << std::endl; // 不会改变vec的元素，用于读
for (auto &element : vec)
    element += 1;                      // 可以改变vec的元素，用于写
```

面向对象

显式虚函数重载

C++11 引入了 `override` 和 `final` 这两个关键字来防止**意外重载虚函数**和基类的虚函数被删除后**子类的对应函数变为普通方法**的情况发生。

override

当重载虚函数时，引入 `override` 关键字将显式的告知编译器进行重载，编译器将检查基函数是否存在这样的虚函数，否则将无法通过编译：

```
struct Base {
    virtual void foo(int);
};
struct SubClass: Base {
    virtual void foo(int) override; // 合法
    virtual void foo(float) override; // 非法，父类没有此虚函数
};
```

final

`final` 则是为了防止类被继续继承以及终止虚函数继续重载引入的。

```
struct Base {
    virtual void foo() final;
};
struct SubClass1 final: Base {
}; // 合法

struct SubClass2 : SubClass1 {
}; // 非法，SubClass1 已 final

struct SubClass3: Base {
    void foo(); // 非法，foo 已 final
};
```

显式禁用默认函数

C++11 允许显式的声明采用或拒绝编译器默认生成的函数。

```
class Magic {
    public:
    Magic() = default; // 显式声明使用编译器生成的构造
    Magic& operator=(const Magic&) = delete; // 显式声明拒绝编译器生成默认赋值函数
    Magic(int magic_number);
}
```

强类型枚举

C++11 引入了枚举类（enumeration class），并使用 `enum class` 的语法进行声明。枚举类实现了类型安全，首先他不能够被隐式的转换为整数，同时也不能够将其与整数数字进行比较， 更不可能对不同的枚举类型的枚举值进行比较。希望获得枚举值的值时，必须**显式**的进行类型转换。

```
#include <iostream>
enum class new_enum : unsigned int {
    value1,
    value2,
    value3 = 100,
    value4 = 100
};
int main(){
    if (new_enum::value3 == new_enum::value4) {
        std::cout << "new_enum::value3 == new_enum::value4" << std::endl;
    }
    std::cout<<(int)new_enum::value4;
    return 0;
}
```

Lambda 表达式

Lambda 表达式是现代 C++ 中最重要的特性之一，而 Lambda 表达式，实际上就是提供了一个类似匿名函数的特性， 而匿名函数则是在需要一个函数，但是又不想费力去命名一个函数的情况下去使用的。

基础

Lambda 表达式的基本语法如下：

```
[捕获列表](参数列表) mutable(可选) 异常属性 -> 返回类型 {
// 函数体
}
```

所谓捕获列表，其实可以理解为参数的一种类型，Lambda 表达式内部函数体在默认情况下是不能够使用函数体外部的变量的， 这时候捕获列表可以起到**传递外部数据**的作用。根据传递的行为，捕获列表也分为以下几种。

值捕获

与参数传值类似，值捕获的前提是变量可以拷贝，不同之处则在于，被捕获的变量在 Lambda 表达式被创建时拷贝， 而非调用时才拷贝。

```
int value = 1;
auto copy_value = [value] {
    return value;
};
value = 100;
auto stored_value = copy_value();
// stored_value == 1, 而 value == 100.
// 因为 copy_value 在创建时就保存了一份 value 的拷贝
```

引用捕获

与引用传参类似，引用捕获保存的是引用，值会发生变化。

```
int value = 1;
auto copy_value = [&value] {
    return value;
};
value = 100;
auto stored_value = copy_value();
// 这时, stored_value == 100, value == 100.
// 因为 copy_value 保存的是引用
```

隐式捕获

可以在捕获列表中写一个 `&` 或 `=` 向编译器声明采用引用捕获或者值捕获。

```
int value = 1;
auto copy_value = [&] {
    return value;
};
```

捕获列表常用的四种形式：

- [] 空捕获列表
- [name1, name2, ...] 捕获一系列变量
- [&] 引用捕获, 让编译器自行推导引用列表
- [=] 值捕获, 让编译器自行推导值捕获列表

表达式捕获

上面提到的值捕获、引用捕获都是已经在外层作用域声明的变量，因此这些捕获方式捕获的均为左值，而不能捕获右值。C++14 允许捕获的成员用任意的表达式进行初始化，这就允许了右值的捕获， 被声明的捕获变量类型会根据表达式进行判断，判断方式与使用 `auto` 本质上是相同的。

```cpp
#include <memory>  // std::make_unique
#include <utility> // std::move, 将important转换为右值

void lambda_expression_capture() {
    auto important = std::make_unique<int>(1);
    auto add = [v1 = 1, v2 = std::move(important)](int x, int y) -> int {
        return x+y+v1+(*v2);
    };
    std::cout << add(3,4) << std::endl;
}
```

泛型 Lambda

从 C++14 开始， Lambda 函数的形式参数可以使用 `auto` 关键字来自动推导参数类型。

```cpp
auto add = [](auto x, auto y) {
    return x+y;
};

add(1, 2);
add(1.1, 2.2);
```

函数对象包装器

这部分内容虽然属于标准库的一部分，但是从本质上来看，它却增强了 C++ 语言运行时的能力。

`std::function`

Lambda 表达式的本质是一个和函数对象类型相似的类类型（称为闭包类型）的对象（称为闭包对象）。当 Lambda 表达式的捕获列表为空时，闭包对象还能够转换为函数指针值进行传递：

```cpp
using foo = void(int); // 定义函数类型
void functional(foo f) { // 定义在参数列表中的函数类型 foo 被视为退化后的函数指针类型 foo*
    f(1); // 通过函数指针调用函数
}

auto f = [](int value) {
    std::cout << value << std::endl;
};// f是闭包对象
functional(f); // 传递闭包对象，隐式转换为 foo* 类型的函数指针值
f(1); // lambda 表达式调用
```

上面的代码给出了两种不同的调用形式，一种是将 Lambda 作为函数类型传递进行调用， 而另一种则是直接调用 Lambda 表达式，在 C++11 中，统一了这些概念，将能够被调用的对象的类型， 统一称之为可调用类型。

C++11 `std::function` 是一种通用、多态的函数封装， 它的实例可以对任何可以调用的目标实体进行存储、复制和调用操作，可以理解为**函数的容器**。

```cpp
#include <functional>

int foo(int para) {
```

```
    return para;
}

// std::function 包装了一个返回值为 int, 参数为 int 的函数
std::function<int(int)> func = foo;

int important = 10;
std::function<int(int)> func2 = [&](int value) -> int {
    return 1+value+important;
};
func(10);
func2(10);
```

右值引用

右值引用是 C++11 引入的与 Lambda 表达式齐名的重要特性之一。它的引入解决了 C++ 中大量的历史遗留问题， 消除了诸如 `std::vector`、`std::string` 之类的额外开销， 也才使得函数对象容器 `std::function` 成为了可能。

左值、右值的纯右值、亡值、右值

左值(lvalue, left value)，顾名思义就是赋值符号左边的值。准确来说， 左值是表达式（不一定是赋值表达式）后依然存在的持久对象。（**注意**：字符串字面量为左值，如"hello"）

右值(rvalue, right value)，右边的值，是指表达式结束后就不再存在的临时对象（或引用）。

而 C++11 中为了引入强大的右值引用，将右值的概念进行了进一步的划分，分为：纯右值、将亡值。

纯右值(prvalue, pure rvalue)，纯粹的右值，要么是纯粹的字面量，例如 `10`，`true`； 要么是临时对象，例如 `-1`，`1+2`,a++的结果。非引用返回的临时变量、运算表达式产生的临时变量、Lambda 表达式都属于纯右值。

亡值(xvalue, expiring value)，是 C++11 为了引入右值引用而提出的概念（因此在传统 C++ 中，只有右值这一个概念），也就是即将被销毁、却能够被移动（move）的值。

右值引用和左值引用

顾名思义，右值引用可以引用一切右值（包括纯右值和亡值）：`T &&`，其中`T`为非左值引用类型（若 T 为左值引用类型，则结果仍是左值引用）。 **右值引用的声明让这个临时对象的生命周期得以延长，只要右值引用生命期还未结束，那么这个临时对象的生命期也不会结束。**

C++11 提供了 `std::move` 这个方法将左值参数无条件的转换为右值， 有了它我们就能够方便的获得一个右值引用（匿名右值引用）。

```
#include <iostream>
#include <utility> // std::move
void reference(std::string& str) {
    std::cout << "左值" << std::endl;
}
void reference(std::string&& str) {
    std::cout << "右值" << std::endl;
}
int main(){
    std::string lv0 = "string,"; // lv0 是一个左值

    std::string& lv1 = lv0; // lv1 是左值引用
```

```cpp

    // std::string&& r1 = lv0; // 非法，右值引用不能引用左值
    std::string&& rv1 = std::move(lv0); // 合法，std::move可以将左值转换为右值

    const std::string& lv2 = lv0 + lv0; // 合法，常量左值引用能够延长临时变量的生命周期
    // lv2 += "Test"; // 非法，常量引用无法被修改

    std::string&& rv2 = lv0 + lv2; // 合法，右值引用延长临时对象生命周期
    rv2 += "Test"; // 合法，非常量引用能够修改临时变量

    reference(lv1); // lv1是左值引用，为左值
    reference(rv2); // rv2是一个具名右值引用，为左值，见完美转发
    reference(std::move(lv0)); // 匿名右值引用，为右值
    return 0;
}
```

移动语义

右值引用的出现恰好解决了传统 C++没有区分『移动』和『拷贝』的概念的问题。

```cpp
#include <iostream>
class A {
public:
    int *pointer;
    A():pointer(new int(1)) {
        std::cout << "构造" << pointer << std::endl;
    }
    A(A& a):pointer(new int(*a.pointer)) {
        std::cout << "拷贝" << pointer << std::endl;
    } // 无意义的对象拷贝
    A(A&& a):pointer(a.pointer) {
        a.pointer = nullptr;
        std::cout << "移动" << pointer << std::endl;
    }
    ~A(){
        std::cout << "析构" << pointer << std::endl;
        delete pointer;
    }
};
A return_rvalue(bool test) {
    A a,b;
    if(test) return a; // 等价于 static_cast<A&&>(a);
    else return b;     // 等价于 static_cast<A&&>(b);
}
int main(){
    A obj = return_rvalue(false);
    // 输出：
    // 构造0xa9eeb0
    // 构造0xa9fee0
    // 移动0xa9fee0
    // 析构0
    // 析构0xa9eeb0
    // 析构0xa9fee0
    return 0;
```

```
}
```

在上面的代码中：

1. 首先会在 `return_rvalue` 内部构造两个 `A` 对象，于是获得两个构造函数的输出；
2. 函数返回后，产生一个亡值，被 `A` 的移动构造（`A(A&&)`）引用，从而延长生命周期，并将这个右值中的指针拿到，保存到了 `obj` 中，而亡值的指针被设置为 `nullptr`。

下面是涉及标准库的例子，使用右值引用避免无意义拷贝以提升性能。

```
std::string str = "Hello world.";
std::vector<std::string> v;

// 将使用 push_back(const T&), 即产生拷贝行为
v.push_back(str);
// 将输出 "str: Hello world."
std::cout << "str: " << str << std::endl;

// 将使用 push_back(const T&&), 不会出现拷贝行为
// 而整个字符串会被移动到 vector 中，所以有时候 std::move 会用来减少拷贝出现的开销
// 这步操作后, str 中的值会变为空
v.push_back(std::move(str));
// 将输出 "str: "
std::cout << "str: " << str << std::endl;
```

完美转发

一个具名右值引用其实是一个左值。这就为我们进行参数转发（传递）造成了问题：

```
void reference(int& v) {
    std::cout << "左值" << std::endl;
}
void reference(int&& v) {
    std::cout << "右值" << std::endl;
}
template <typename T>
void pass(T&& v) {
    reference(v); // v是具名右值引用，为左值，故始终调用 reference(int&)
}

std::cout << "传递右值:" << std::endl;
pass(1); // 1是右值，但输出是左值

std::cout << "传递左值:" << std::endl;
int l = 1;
pass(l); // l 是左值，输出左值
```

为了解决这个问题，我们应该使用 `std::forward` 来进行参数的转发（传递）：

```
#include <iostream>
#include <utility>
void reference(int& v) {
    std::cout << "左值引用" << std::endl;
}
```

```cpp
void reference(int&& v) {
    std::cout << "右值引用" << std::endl;
}
template <typename T>
void pass(T&& v) {// v是左值
    reference(v);// 普通传参，永远输出左值引用
    reference(std::move(v));// std::move 传参，永远输出右值引用
    reference(std::forward<T>(v));// std::forward 传参
    reference(static_cast<T&&>(v));// static_cast<T&&> 传参
}
int main(){
    int a = 1;
    pass(1);
    pass(a);
    return 0;
}
```

`std::forward` 不会造成任何多余的拷贝，同时**完美转发**函数的实参给内部调用的其他函数：

当实参是右值（int、或者 int 的引用），T 被推导为 int，T&& 是 int&&； 当实参是左值（int、或者 int 的引用），T 被推导为 int&，T&& 是 int&。

`std::forward` 和 `std::move` 一样，只是类型转换，`std::move` 单纯的将左值转化为右值，`std::forward` 也只是单纯的将参数做了一个类型的转换，从现象上来看，`std::forward<T>(v)` 和 `static_cast<T&&>(v)` 是完全一样的。

智能指针与内存管理

RAII 与引用计数

引用计数这种计数是为了防止内存泄露而产生的。 基本想法是对于动态分配的对象，进行引用计数，每当增加一次对同一个对象的引用，那么引用对象的引用计数就会增加一次， 每删除一次引用，引用计数就会减一，当一个对象的引用计数减为零时，就自动删除指向的堆内存。

在传统 C++ 中，『记得』手动释放资源，总不是最佳实践。因为我们很有可能就忘记了去释放资源而导致泄露。 所以通常的做法是对于一个对象而言，我们在构造函数的时候申请空间，而在析构函数（在离开作用域时调用）的时候释放空间， 也就是我们常说的 RAII 资源获取即初始化技术。

C++11 引入智能指针的概念，让程序员不再需要关心手动释放内存。使用它们需要包含头文件 `<memory>`。

`std::unique_ptr`

`std::unique_ptr` 是一种独占的智能指针，它禁止其他智能指针与其共享同一个对象，从而保证代码的安全。

```cpp
std::unique_ptr<int> pointer = std::make_unique<int>(10); // make_unique 从 C++14 引入
std::unique_ptr<int> pointer2 = pointer; // 非法
```

> `make_unique` 并不复杂，C++11 没有提供 `std::make_unique`，可以自行实现：

```cpp
template<typename T, typename ...Args>
std::unique_ptr<T> make_unique( Args&& ...args ) {
return std::unique_ptr<T>( new T( std::forward<Args>(args)... ) );
}
```

既然是独占，换句话说就是不可复制。但是，我们可以利用 `std::move` 将其转移给其他的 `unique_ptr`。

```cpp
#include <memory>

struct Foo {
    Foo() { std::cout << "Foo::Foo" << std::endl; }
    ~Foo() { std::cout << "Foo::~Foo" << std::endl; }
    void foo() { std::cout << "Foo::foo" << std::endl; }
};

void f(const Foo &) {
    std::cout << "f(const Foo&)" << std::endl;
}

int main() {
    std::unique_ptr<Foo> p1(std::make_unique<Foo>());
    // p1 不空，输出
    if (p1) p1->foo();
    {
        std::unique_ptr<Foo> p2(std::move(p1));
        // p2 不空，输出
        f(*p2);
        // p2 不空，输出
        if(p2) p2->foo();
        // p1 为空，无输出
        if(p1) p1->foo();
        p1 = std::move(p2);
        // p2 为空，无输出
        if(p2) p2->foo();
        std::cout << "p2 被销毁" << std::endl;
    }
    // p1 不空，输出
    if (p1) p1->foo();
    // Foo 的实例会在离开作用域时被销毁
}
```

此外，由于独占，`std::unique_ptr`不会有引用计数的开销，因此常常是首选。

`std::shared_ptr`

`std::shared_ptr` 是一种智能指针，它能够记录多少个 `shared_ptr` 共同指向一个对象，从而消除显式的调用 `delete`，当引用计数变为零的时候就会将对象自动删除。

但还不够，因为使用 `std::shared_ptr` 仍然需要使用 `new` 来调用，这使得代码出现了某种程度上的不对称。

`std::make_shared` 就能够用来消除显式的使用 `new`，会分配创建传入参数中的对象， 并返回这个对象类型的 `std::shared_ptr` 指针。

```cpp
#include <iostream>
#include <memory>
void foo(std::shared_ptr<int> i) {
    (*i)++;
}
int main() {
    // Constructed a std::shared_ptr
    auto pointer = std::make_shared<int>(10);
    foo(pointer);
    std::cout << *pointer << std::endl; // 11
    // The shared_ptr will be destructed before leaving the scope
    return 0;
}
```

`std::shared_ptr` 可以通过 `get()` 方法来获取原始指针，通过 `reset()` 来减少一个引用计数， 并通过`use_count()`来查看一个对象的引用计数。

```cpp
auto pointer = std::make_shared<int>(10);
auto pointer2 = pointer; // 引用计数+1
auto pointer3 = pointer; // 引用计数+1
int *p = pointer.get(); // 这样不会增加引用计数
std::cout << "pointer.use_count() = " << pointer.use_count() << std::endl;
// 3
std::cout << "pointer2.use_count() = " << pointer2.use_count() << std::endl;
// 3
std::cout << "pointer3.use_count() = " << pointer3.use_count() << std::endl;
// 3

pointer2.reset();
std::cout << "reset pointer2:" << std::endl;
std::cout << "pointer.use_count() = " << pointer.use_count() << std::endl;
// 2
std::cout << "pointer2.use_count() = " << pointer2.use_count() << std::endl;
// 0, pointer2 已 reset
std::cout << "pointer3.use_count() = " << pointer3.use_count() << std::endl;
// 2
pointer3.reset();
std::cout << "reset pointer3:" << std::endl;
std::cout << "pointer.use_count() = " << pointer.use_count() << std::endl;
// 1
std::cout << "pointer2.use_count() = " << pointer2.use_count() << std::endl;
// 0
std::cout << "pointer3.use_count() = " << pointer3.use_count() << std::endl;
// 0, pointer3 已 reset
```

`std::weak_ptr`

`std::shared_ptr`引入了引用成环的问题。

```cpp
struct A;
struct B;

struct A {
    std::shared_ptr<B> pointer;
```

```
    ~A() {
        std::cout << "A 被销毁" << std::endl;
    }
};
struct B {
    std::shared_ptr<A> pointer;
    ~B() {
        std::cout << "B 被销毁" << std::endl;
    }
};
int main() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();
    a->pointer = b;
    b->pointer = a;
    // 结果是A和B依然不能被销毁，因为二者间形成了一个环，导致两个shared_ptr引用计数都为1
}
```

解决这个问题的办法就是使用弱引用指针 `std::weak_ptr`，`std::weak_ptr`是一种弱引用（相比较而言 `std::shared_ptr` 就是一种强引用）。弱引用不会引起引用计数增加。

对于上面的代码，将 A 或 B 中的任意一个`std::shared_ptr` 改为 `std::weak_ptr`即可解决问题。

```
struct A {
    std::shared_ptr<B> pointer;
    ~A() {
        std::cout << "A 被销毁" << std::endl;
    }
};
struct B {
    std::weak_ptr<A> pointer;
    ~B() {
        std::cout << "B 被销毁" << std::endl;
    }
};
// 将B的智能指针改为weak_ptr，这样程序结束时A的引用计数就会变成0而销毁，B的引用计数随之变为0而销毁
```

`std::weak_ptr` 没有 `*` 运算符和 `->` 运算符，所以不能够对资源进行操作，它可以用于检查 `std::shared_ptr` 是否存在，其 `expired()` 方法能在资源未被释放时，会返回 `false`，否则返回 `true`；除此之外，它也可以用于获取指向原始对象的 `std::shared_ptr` 指针，其 `lock()` 方法在原始对象未被释放时，返回一个指向原始对象的 `std::shared_ptr` 指针，进而访问原始对象的资源，否则返回默认构造的`std::shared_ptr`（即未托管任何指针）。

```cpp
std::weak_ptr<int> b;
{
    auto a = std::make_shared<int>(1);
    b = a;
    if(auto c = b.lock()) { // 输出
        std::cout << *c << std::endl;
    }
}
if(auto c = b.expired()) { // 输出
    std::cout << "b is expired" << std::endl;
}
```

并行与并发

并发基础

`std::thread` 用于创建一个执行的线程实例，所以它是一切并发编程的基础，使用时需要包含 `<thread>` 头文件， 它提供了很多基本的线程操作，例如 `get_id()` 来获取所创建线程的线程 ID，使用 `join()` 来加入一个线程等等。

```cpp
#include <thread>

int main() {
    std::thread t([](){
        std::cout << "hello world." << std::endl;
    });
    t.join();
    return 0;
}
```

互斥量与临界区

互斥量 `mutex` 是并发技术中的核心之一。 C++11 引入了 `mutex` 相关的类，其所有相关的函数都放在 `<mutex>` 头文件中。

`std::mutex` 是 C++11 中最基本的 `mutex` 类，通过实例化 `std::mutex` 可以创建互斥量， 而通过其成员函数 `lock()` 可以进行上锁， `unlock()` 可以进行解锁。 但是在实际编写代码的过程中，最好不去直接调用成员函数， 因为调用成员函数就需要在每个临界区的出口处调用 `unlock()`，当然，还包括异常。 这时候 C++11 还为互斥量提供了一个 RAII 语法的模板类 `std::lock_guard`。 RAII 在不失代码简洁性的同时，很好的保证了代码的异常安全性。

在 RAII 用法下，对于临界区的互斥量的创建只需要在作用域的开始部分。

```cpp
#include <mutex>
#include <thread>

int v = 1;

void critical_section(int change_v) {
    static std::mutex mtx;
    std::lock_guard<std::mutex> lock(mtx);
    // 执行竞争操作
    v = change_v;
    // 离开此作用域后 mtx 会被释放
}
```

这样的代码也是异常安全的。 无论临界区正常返回、还是在中途抛出异常，都会自动调用 `unlock()`。

而 `std::unique_lock` 则相对于 `std::lock_guard` 出现的，`std::unique_lock` 更加灵活，`std::unique_lock` 的对象会以独占所有权（没有其他的 `unique_lock` 对象同时拥有某个 `mutex` 对象的所有权） 的方式管理 `mutex` 对象上的上锁和解锁的操作。所以在并发编程中，推荐使用 `std::unique_lock`。

`std::lock_guard` 不能显式的调用 `lock` 和 `unlock`， 而 `std::unique_lock` 可以在声明后的任意位置调用， 可以缩小锁的作用范围，提供更高的并发度。

如果你用到了条件变量 `std::condition_variable::wait` 则必须使用 `std::unique_lock` 作为参数。

```cpp
#include <mutex>
#include <thread>

int v = 1;

void critical_section(int change_v) {
    static std::mutex mtx;
    std::unique_lock<std::mutex> lock(mtx);
    // 执行竞争操作
    v = change_v;
    // 将锁进行释放
    lock.unlock();

    // 在此期间，任何人都可以抢夺 v 的持有权

    // 开始另一组竞争操作，再次加锁
    lock.lock();
    v += 1;
    // 离开此作用域后 mtx 会被释放
}
```

期值 future

如果我们的主线程 A 希望新开辟一个线程 B 去执行某个我们预期的任务，并返回一个结果。 而这时候，线程 A 可能正在忙其他的事情，无暇顾及 B 的结果， 所以我们会很自然的希望能够在某个特定的时间获得线程 B 的结果。

在 C++11 的 `std::future` 被引入之前，通常的做法是： 创建一个线程 A，在线程 A 里启动任务 B，当准备完毕后发送一个事件，并将结果保存在全局变量中。 而主函数线程 A 里正在做其他的事情，当需要结果的时候，调用一个线程等待函数来获得执行的结果。

而 C++11 提供的 `std::future` 简化了这个流程，可以用来获取异步任务的结果。 自然地，我们很容易能够想象到把它作为一种简单的线程同步手段，即屏障（barrier）。

标准库中的`std::async`可在其中调用的函数执行完成前就返回其`std::future`对象，对该对象使用get()即可阻塞程序到函数执行完成时取得返回值：

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

int main(){
    auto result = std::async(std::launch::async, [](){return 7;});
    std::cout << "waiting..." << std::endl;
    std::cout << "future result is " << result.get() << std::endl;
    return 0;
}
```

条件变量

条件变量 `std::condition_variable` 是为解决死锁而生。 比如，线程可能需要等待某个条件为真才能继续执行， 而一个忙等待循环中可能会导致所有其他线程都无法进入临界区使得条件为真时，就会发生死锁。 所以，`condition_variable` 实例被创建出现主要就是用于唤醒等待线程从而避免死锁。 `std::condition_variable` 的 `notify_one()` 用于唤醒一个线程； `notify_all()` 则是通知所有线程。下面是一个生产者和消费者模型的例子。

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


int main() {
    std::queue<int> produced_nums;
    std::mutex mtx;
    std::condition_variable cv;

    // 生产者
    auto producer = [&]() {
        for (int i = 0; ; i++) {
            std::this_thread::sleep_for(std::chrono::milliseconds(900)); //
生产时间
            {
                std::unique_lock<std::mutex> lock(mtx);
                // 生产1个
                std::cout << "producing " << i << std::endl;
                produced_nums.push(i);

                cv.notify_one();
            }
```

```
        }
    };
    // 消费者
    auto consumer = [&]() {
        while (true) {
            std::this_thread::sleep_for(std::chrono::milliseconds(1000)); // 消费慢于生产
            {
                std::unique_lock<std::mutex> lock(mtx);
                while (produced_nums.empty()) {  // 避免虚假唤醒
                    cv.wait(lock);
                }
                // 等价写法 cv.wait(lock, [&]{ return !produced_nums.empty(); });
                // 消费1个
                std::cout << "consuming " << produced_nums.front() << std::endl;
                produced_nums.pop();
            }
        }
    };

    // 分别在不同的线程中运行
    std::thread p(producer);
    std::thread cs[2];
    for (int i = 0; i < 2; ++i) {
        cs[i] = std::thread(consumer);
    }
    p.join();
    for (int i = 0; i < 2; ++i) {
        cs[i].join();
    }
    return 0;
}
```

杂项

noexcept

C++11 将异常的声明简化为以下两种情况：

1. 函数可能抛出任何异常
2. 函数不能抛出任何异常

并使用 `noexcept` 对这两种行为进行限制，例如：

```
void may_throw(); // 可能抛出异常
void no_throw() noexcept; // 不可能抛出异常
```

使用 `noexcept` 修饰过的函数如果抛出异常，编译器会使用 `std::terminate()` 来立即终止程序运行。

`noexcept` 还能够做操作符，用于操作一个表达式，当表达式无异常时，返回 `true`，否则返回 `false`。

```cpp
#include <iostream>
void may_throw() {
    throw true;
}
auto non_block_throw = []{
    may_throw();
};
void no_throw() noexcept {
    return;
}

auto block_throw = []() noexcept {
    no_throw();
};
int main()
{
    std::cout << std::boolalpha
        << "may_throw() noexcept? " << noexcept(may_throw()) << std::endl
        << "no_throw() noexcept? " << noexcept(no_throw()) << std::endl
        << "lmay_throw() noexcept? " << noexcept(non_block_throw()) <<
std::endl
        << "lno_throw() noexcept? " << noexcept(block_throw()) << std::endl;
    return 0;
}
```

```cpp
try {
    may_throw();
} catch (...) {
    std::cout << "捕获异常，来自 may_throw()" << std::endl;
}
try {
    non_block_throw();
} catch (...) {
    std::cout << "捕获异常，来自 non_block_throw()" << std::endl;
}
try {
    block_throw();
} catch (...) {
    std::cout << "捕获异常，来自 block_throw()" << std::endl;
}
// output
// 捕获异常，来自 may_throw()
// 捕获异常，来自 non_block_throw()
```

Reference

https://docs.eesast.com/docs/languages/C&C++/modern_cpp