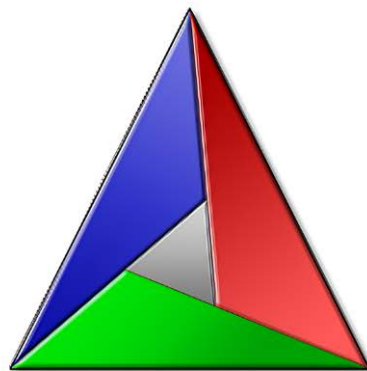


学C++从CMake学起

by 彭于斌 (@archibate)



CMake
Cross-platform Make

高性能并行编程与优化 - 课程大纲

- 分为前半段和后半段，前半段主要介绍现代 C++，后半段主要介绍并行编程与优化。

1. 课程安排与开发环境搭建：cmake与git入门
2. 现代C++入门：常用STL容器，RAII内存管理
3. 现代C++进阶：模板元编程与函数式编程
4. 编译器如何自动优化：从汇编角度看C++
5. C++11起的多线程编程：从mutex到无锁并行
6. 并行编程常用框架：OpenMP与Intel TBB
7. 被忽视的访存优化：内存带宽与cpu缓存机制
8. GPU专题：wrap调度，共享内存，barrier
9. 并行算法实战：reduce，scan，矩阵乘法等
10. 存储大规模三维数据的关键：稀疏数据结构
11. 物理仿真实战：邻居搜索表实现pbf流体求解
12. C++在ZENO中的工程实践：从primitive说起
13. 结业典礼：总结所学知识 with 优秀作业点评

硬件要求：

64位（32位时代过去了）

至少2核4线程（并行课...）

英伟达家显卡（GPU 专题）

软件要求：

Visual Studio 2019（Windows用户）

GCC 9 及以上（Linux用户）

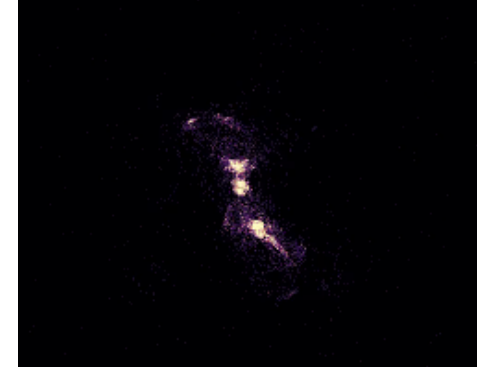
CMake 3.12 及以上（跨平台作业）

Git 2.x（作业上传到 GitHub）

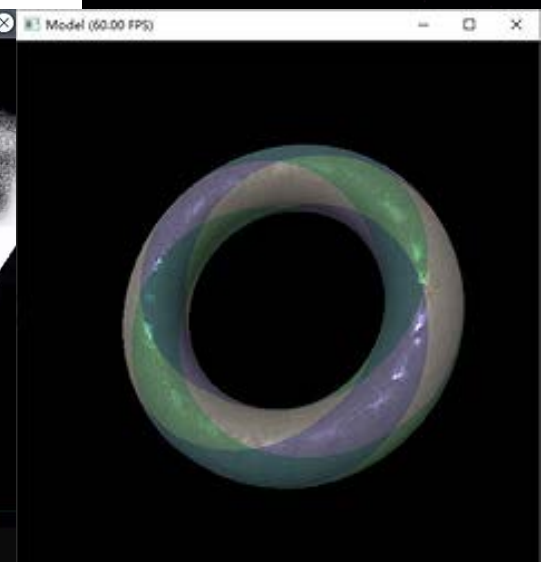
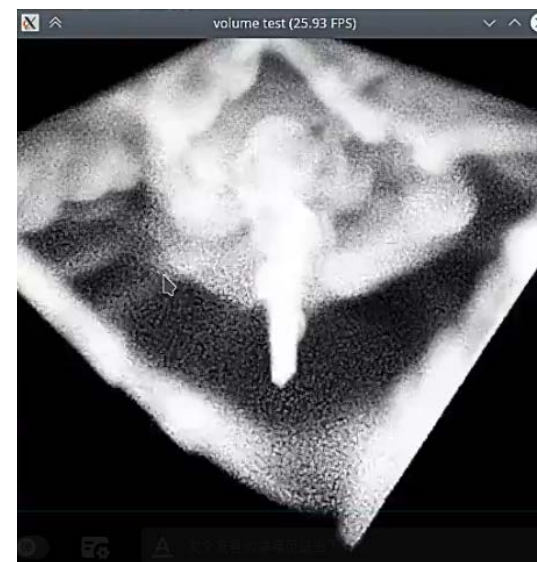
CUDA Toolkit 10.0 以上（GPU 专题）



关于作者



- 我是 Taichi 编译器的贡献者之一 (<https://github.com/taichi-dev/taichi>)



```
import taichi as ti

ti.init(arch=ti.gpu) # Run on GPU by default

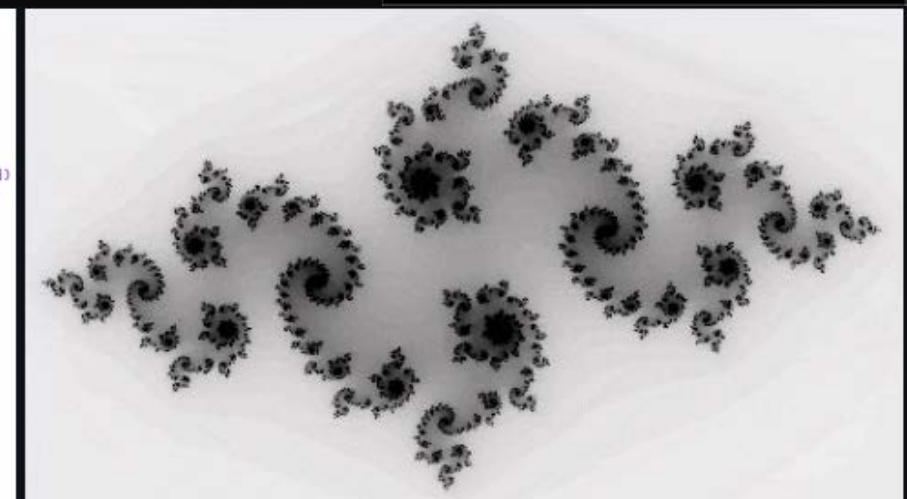
n = 320
pixels = ti.field(dtype=float, shape=(n * 2, n))

@ti.func
def complex_sqr(z):
    return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def paint():
    for i, j in pixels: # Parallelized over all pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
        iterations = 0
        while z.norm() < 20 and iterations < 50:
            z = complex_sqr(z) + c
            iterations += 1
        pixels[i, j] = 1 - iterations * 0.02

gui = ti.GUI("Julia Set", res=(n * 2, n))

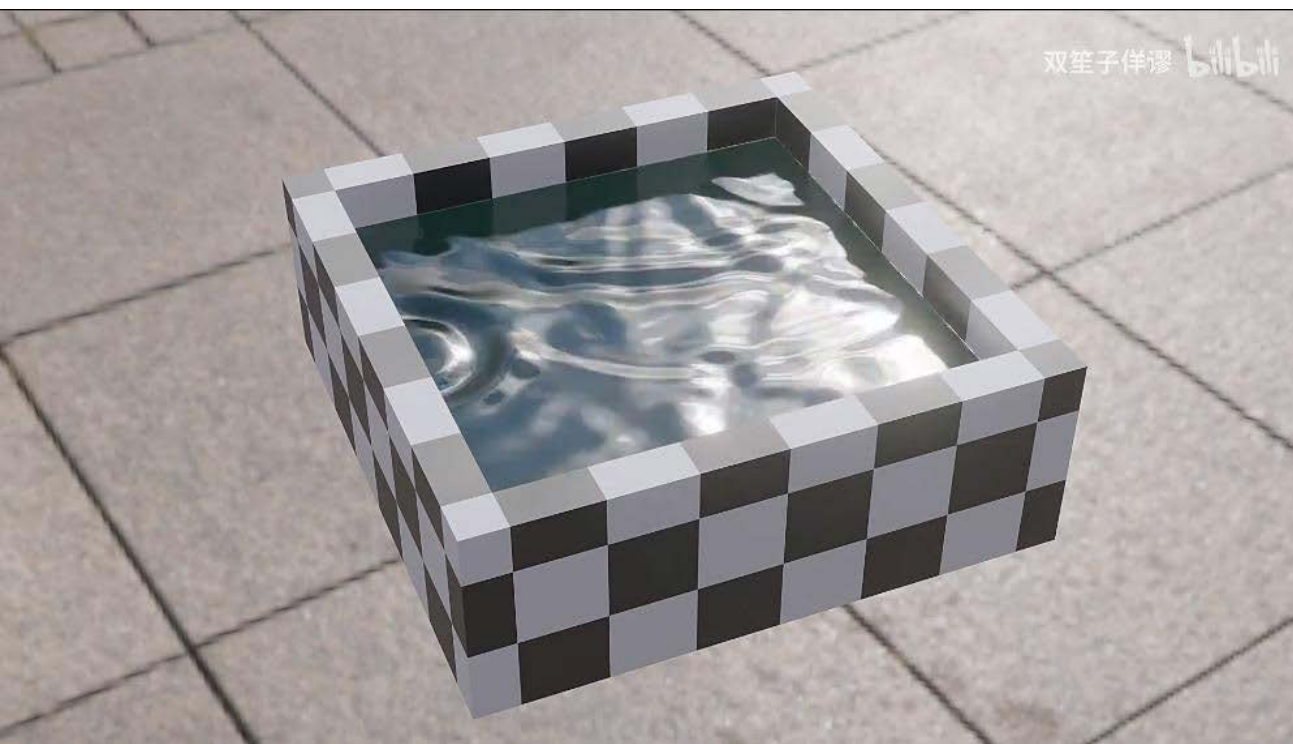
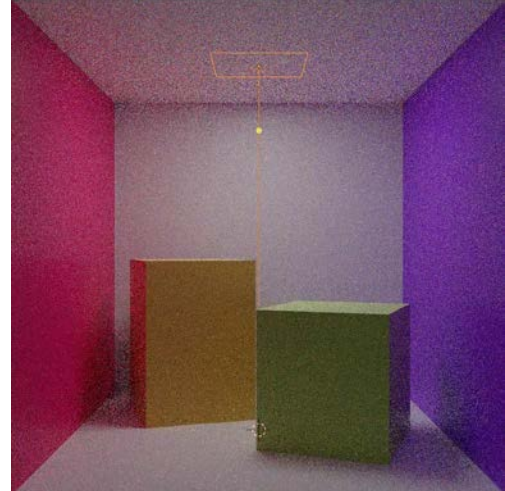
for i in range(1000000):
    paint(i * 0.03)
    gui.set_image(pixels)
    gui.show()
```



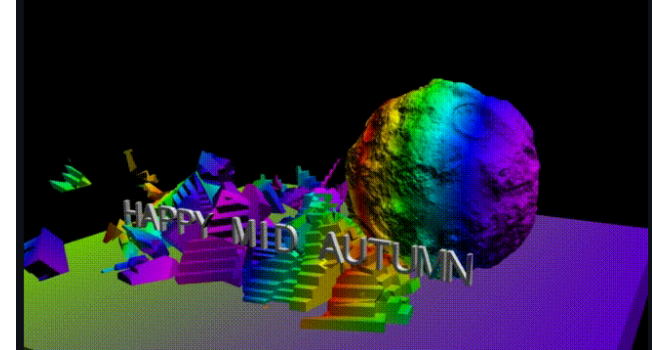
关于作者（续）



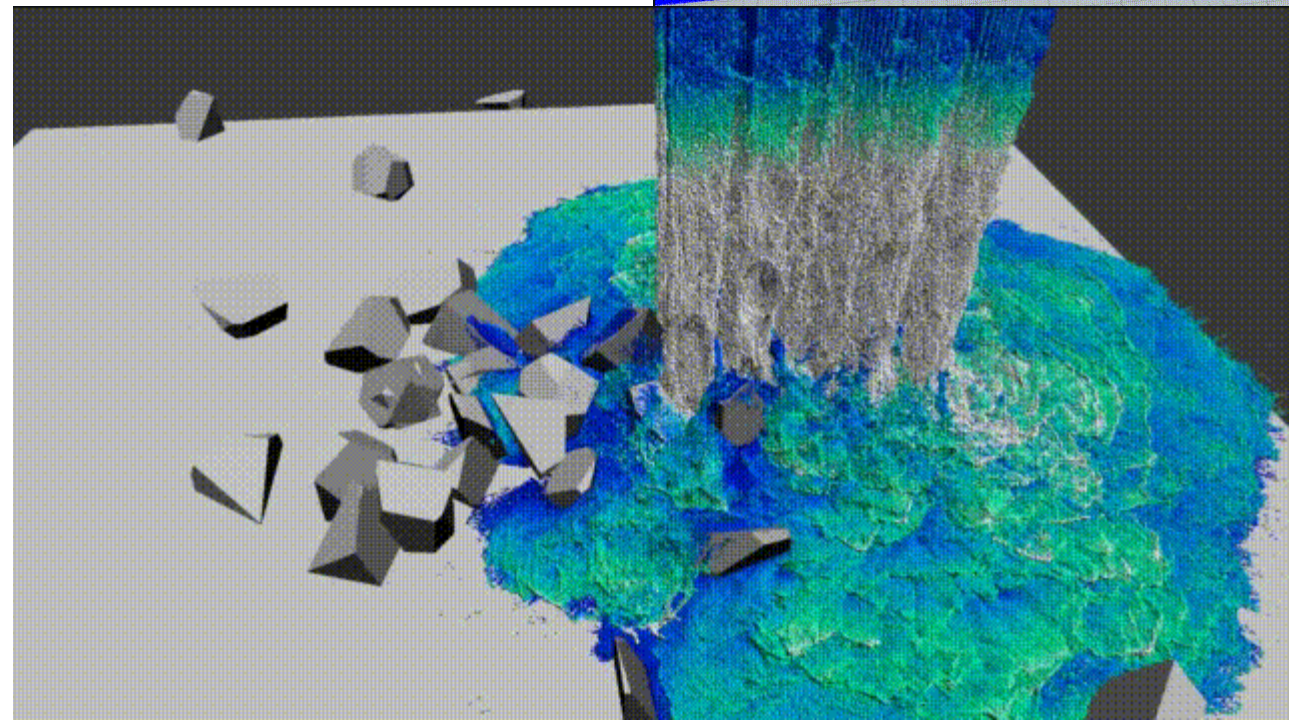
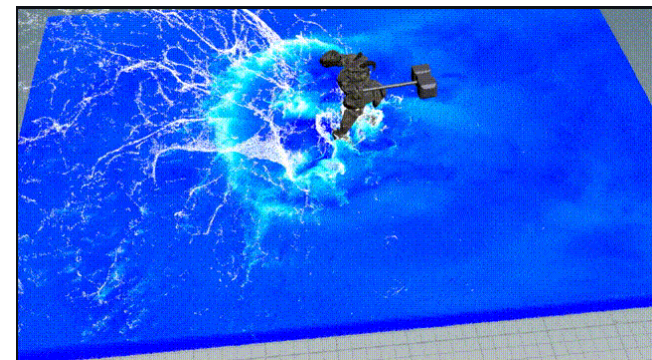
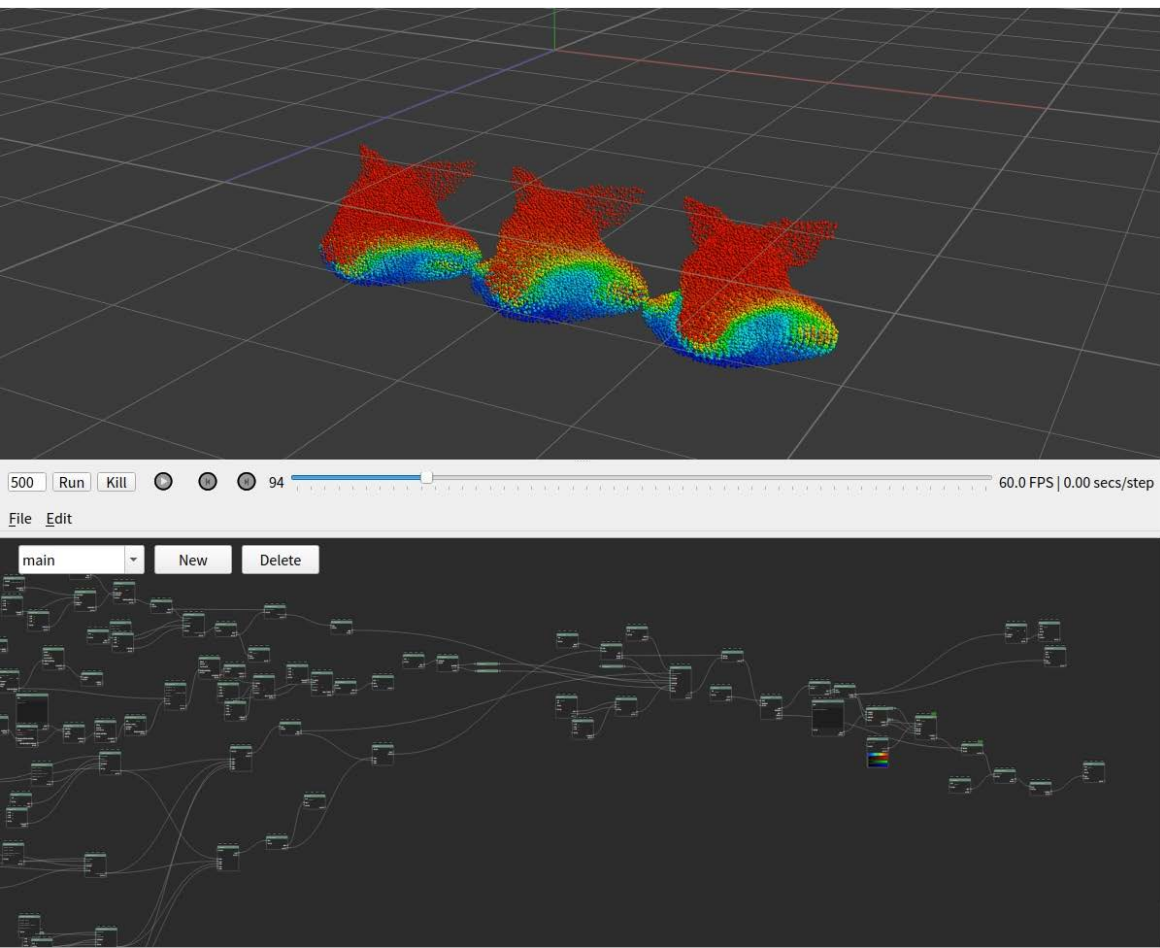
- 我是 Taichi Blend 的作者（https://github.com/taichi-dev/taichi_blend）



关于作者（再续）



- 主导 Zeno 节点仿真框架的开发 (<https://github.com/zenustech/zeno>)





厂商	C	C++	Fortran
GNU	gcc	g++	gfortran
LLVM	clang	clang++	flang

什么是编译器

- **编译器**，是一个根据**源代码**生成**机器码**的程序。
- `> g++ main.cpp -o a.out`
- 该命令会调用编译器程序g++，让他读取main.cpp中的字符串（称为源码），并根据C++标准生成相应的机器指令码，输出到a.out这个文件中，（称为可执行文件）。
- `> ./a.out`
- 之后执行该命令，操作系统会读取刚刚生成的可执行文件，从而执行其中编译成机器码，调用系统提供的printf函数，并在终端显示出Hello, world。

main.cpp

```
1 #include <stdio>
2
3 int main() {
4     printf("Hello, world!\n");
5     return 0;
6 }
```



```
bate@archer ~/Codes/course/01/01 (master) $ ./a.out
Hello, world!
bate@archer ~/Codes/course/01/01 (master) $
```


多文件编译与链接

```
hello.cpp
1 #include <stdio>
2
3 void hello() {
4     printf("Hello, world\n");
5 }
```

```
main.cpp
1 #include <stdio>
2
3 void hello();
4
5 int main() {
6     hello();
7     return 0;
8 }
```

- 单文件编译虽然方便，但也有如下缺点：
 1. 所有的代码都堆在一起，不利于模块化和理解。
 2. 工程变大时，编译时间变得很长，改动一个地方就得全部重新编译。
- 因此，我们提出多文件编译的概念，文件之间通过**符号声明**相互引用。
- > g++ -c hello.cpp -o hello.o
- > g++ -c main.cpp -o main.o
- 其中使用 -c 选项指定生成临时的**对象文件** main.o，之后再根据一系列对象文件进行链接，得到最终的a.out:
- > g++ hello.o main.o -o a.out

为什么需要构建系统（Makefile）



- 文件越来越多时，一个个调用g++编译链接会变得很麻烦。
- 于是，发明了 **make** 这个程序，你只需写出不同文件之间的**依赖关系**，和生成各文件的规则。
- `> make a.out`
- 敲下这个命令，就可以构建出 **a.out** 这个可执行文件了。
- 和直接用一个脚本写出完整的构建过程相比，**make** 指明依赖关系的好处：
 1. 当更新了**hello.cpp**时只会重新编译**hello.o**，而不需要把**main.o**也重新编译一遍。
 2. 能够自动**并行**地发起对**hello.cpp**和**main.cpp**的编译，加快编译速度（`make -j`）。
 3. 用通配符批量生成构建规则，避免针对每个**.cpp**和**.o**重复写 `g++` 命令（`%.o: %.cpp`）。
- 但坏处也很明显：
 1. **make** 在 **Unix** 类系统上是通用的，但在 **Windows** 则不然。
 2. 需要准确地指明每个项目之间的依赖关系，有头文件时特别头疼。
 3. **make** 的语法非常简单，不像 **shell** 或 **python** 可以做很多判断等。
 4. 不同的编译器有不同的 **flag** 规则，为 **g++** 准备的参数可能对 **MSVC** 不适用。

Makefile+

```
1 a.out: hello.o main.o
2     g++ hello.o main.o -o a.out
3
4 hello.o: hello.cpp
5     g++ -c hello.cpp -o hello.o
6
7 main.o: main.cpp
8     g++ -c main.cpp -o main.o
```

构建系统的构建系统（CMake）

```
CMakeLists.txt  
1 add_executable(a.out main.cpp hello.cpp)
```

↑
输出的可执行文件 ↖ ↗
 输入的多个源文件

- 为了解决 make 的以上问题，跨平台的 CMake 应运而生！
- ~~make 在 Unix 类系统上是通用的，但在 Windows 则不然。~~
- 只需要写一份 CMakeLists.txt，他就能在调用时生成当前系统所支持的构建系统。
- ~~需要准确地指明每个项目之间的依赖关系，有头文件时特别头疼。~~
- CMake 可以自动检测源文件和头文件之间的依赖关系，导出到 Makefile 里。
- ~~make 的语法非常简单，不像 shell 或 python 可以做很多判断等。~~
- CMake 具有相对高级的语法，内置的函数能够处理 configure, install 等常见需求。
- ~~不同的编译器有不同的 flag 规则，为 g++ 准备的参数可能对 MSVC 不适用。~~
- CMake 可以自动检测当前的编译器，需要添加哪些 flag。比如 OpenMP，只需要在 CMakeLists.txt 中指明 target_link_libraries(a.out OpenMP::OpenMP_CXX) 即可。




CMake 的命令行调用



CMake

- 读取当前目录的 CMakeLists.txt，并在 build 文件夹下生成 build/Makefile:
- > cmake -B build
- 让 make 读取 build/Makefile，并开始构建 a.out:
- > make -C build
- 以下命令和上一个等价，但更跨平台:
- > cmake --build build
- 执行生成的 a.out:
- > build/a.out

```
1 cmake_minimum_required(VERSION 3.12)
2 project(hellocmake LANGUAGES CXX)
3
4 add_executable(a.out main.cpp hello.cpp)
```



```
-- The CXX compiler identification is GNU 11.1.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/bate/Codes/course/01/05/build
[ 33%] Building CXX object CMakeFiles/a.out.dir/main.cpp.o
[ 66%] Building CXX object CMakeFiles/a.out.dir/hello.cpp.o
[100%] Linking CXX executable a.out
[100%] Built target a.out
Hello, world
```

为什么需要库 (library)



- 有时候我们会有多个可执行文件，他们之间用到的某些功能是相同的，我们想把这些共用的功能做成一个库，方便大家一起共享。
- 库中的函数可以被可执行文件调用，也可以被其他库文件调用。
- 库文件又分为静态库文件和动态库文件。

为了更好地理解静态-动态库，最好对可执行文件进行汇编下的查看

- ~~• 其中静态库相当于直接把代码插入到生成的可执行文件中，会导致体积变大，但是只需要一个文件即可运行。~~
- ~~• 而动态库则只在生成的可执行文件中生成“插桩”函数，当可执行文件被加载时会读取指定目录中的.dll文件，加载到内存中空闲的位置，并且替换相应的“插桩”指向的地址为加载后的地址，这个过程称为重定向。这样以后函数被调用就会跳转到动态加载的地址去。~~
- Windows: 可执行文件同目录，其次是环境变量%PATH%
- Linux: ELF格式可执行文件的RPATH，其次是/usr/lib等

插装

```
bate@archer ~/Codes/course/01/06 (master) $ s
build CMakeLists.txt hello.cpp main.cpp run.sh
bate@archer ~/Codes/course/01/06 (master) $ cd build
bate@archer ~/Codes/course/01/06/build (master) $ s
a.out
CMakeFiles
libhellolib.so
CMakeCache.txt cmake_install.cmake Makefile
bate@archer ~/Codes/course/01/06/build (master) $ ldd a.out
linux-vdso.so.1 (0x00007ffe34f40000)
libhellolib.so => /home/bate/Codes/course/01/06/build/libhellolib.so (0x00007f28a9338000)
libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x00007f28a90f5000)
libm.so.6 => /usr/lib/libm.so.6 (0x00007f28a8fb1000)
libgcc_s.so.1 => /usr/lib/libgcc_s.so.1 (0x00007f28a8f96000)
libc.so.6 => /usr/lib/libc.so.6 (0x00007f28a8dca000)
/lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007f28a9344000)
bate@archer ~/Codes/course/01/06/build (master) $
```

```
bate@archer ~/Codes/course/01/06/build (master) $ objdump -D a.out | less
bate@archer ~/Codes/course/01/06/build (master) $ rm libhellolib.so
bate@archer ~/Codes/course/01/06/build (master) $ ./a.out
./a.out: error while loading shared libraries: libhellolib.so: cannot open shared object file: No such file or directory
bate@archer ~/Codes/course/01/06/build (master) $
```

```
bate@archer ~/Codes/course/01/06/build
[ 25%] Building CXX object CMakeFiles/hellolib.dir/hello.cpp.o
[ 50%] Linking CXX shared library libhellolib.so
[ 50%] Built target hellolib
[ 75%] Building CXX object CMakeFiles/a.out.dir/main.cpp.o
[100%] Linking CXX executable a.out
[100%] Built target a.out
Hello, world
bate@archer ~/Codes/course/01/06 (master) $ s
build CMakeLists.txt hello.cpp main.cpp run.sh
bate@archer ~/Codes/course/01/06 (master) $ cd build
bate@archer ~/Codes/course/01/06/build (master) $ s
a.out
CMakeFiles
libhellolib.so
CMakeCache.txt cmake_install.cmake Makefile
bate@archer ~/Codes/course/01/06/build (master) $ ldd a.out
linux-vdso.so.1 (0x00007ffe34f40000)
libhellolib.so => /home/bate/Codes/course/01/06/build/libhellolib.so (0x00007f28a9338000)
libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x00007f28a90f5000)
libm.so.6 => /usr/lib/libm.so.6 (0x00007f28a8fb1000)
libgcc_s.so.1 => /usr/lib/libgcc_s.so.1 (0x00007f28a8f96000)
libc.so.6 => /usr/lib/libc.so.6 (0x00007f28a8dca000)
/lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007f28a9344000)
bate@archer ~/Codes/course/01/06/build (master) $ objdump -D a.out | less
```

```
bate@archer ~/Codes/course/01/06/build
0000000000001020 <_Z5hellov@plt-0x10>:
1020: ff 35 e2 2f 00 00 push 0x2fe2(%rip) # 4008 <_GLOBAL_OFFSET_TABLE_+0x8>
1026: ff 25 e4 2f 00 00 jmp *0x2fe4(%rip) # 4010 <_GLOBAL_OFFSET_TABLE_+0x10>
102c: 0f 1f 40 00 nopl 0x0(%rax)

0000000000001030 <_Z5hellov@plt>:
1030: ff 25 e2 2f 00 00 jmp *0x2fe2(%rip) # 4018 <_Z5hellov>
1036: 68 00 00 00 00 push $0x0
103b: e9 e0 ff ff jmp 1020 <_init+0x20>

Disassembly of section .text:

0000000000001040 <_start>:
1040: f3 0f 1e fa endbr64
1044: 31 ed xor %ebp,%ebp
1046: 49 89 d1 mov %rdx,%r9
1049: 5e pop %rsi
104a: 48 89 e2 mov %rsp,%rdx
104d: 48 83 e4 f0 and $0xffffffffffffff0,%rsp
1051: 50 push %rax
```

静态库就是把相应的实现放进可执行程序里面

```
bate@archer ~/Codes/course/01/06/build
0000000000001139 <main>:
1139: 55 push %rbp
113a: 48 89 e5 mov %rsp,%rbp
113d: e8 07 00 00 00 call 1149 <_Z5hellov>
1142: b8 00 00 00 00 mov $0x0,%eax
1147: 5d pop %rbp
1148: c3 ret

0000000000001149 <_Z5hellov>:
1149: 55 push %rbp
114a: 48 89 e5 mov %rsp,%rbp
114d: 48 8d 05 b0 0e 00 00 lea 0xeb0(%rip),%rax # 2004 <_IO_stdin_used+0x4>
1154: 48 89 c7 mov %rax,%rdi
1157: e8 d4 fe ff ff call 1030 <puts@plt>
115c: 90 nop
115d: 5d pop %cbp
115e: c3 ret
115f: 90 nop

0000000000001160 <__libc_csu_init>:
1160: f3 0f 1e fa endbr64
:
```

CMake 中的静态库与动态库

- CMake 除了 `add_executable` 可以生成可执行文件外，还可以通过 `add_library` 生成库文件。
- `add_library` 的语法与 `add_executable` 大致相同，除了他需要指定是动态库还是静态库：
- `add_library(test STATIC source1.cpp source2.cpp)` # 生成静态库 `libtest.a`
- `add_library(test SHARED source1.cpp source2.cpp)` # 生成动态库 `libtest.so`
- 动态库有很多坑，特别是 Windows 环境下，初学者自己创建库时，建议使用静态库。
- 但是他人提供的库，大多是作为动态库的，我们之后会讨论如何使用他人的库。
- 创建库以后，要在某个可执行文件中使用该库，只需要：
- `target_link_libraries(myexec PUBLIC test)` # 为 `myexec` 链接刚刚制作的库 `libtest.a`
- 其中 `PUBLIC` 的含义稍后会说明（CMake 中有很多这样的大写修饰符）

CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.12)
2 project(hellocmake LANGUAGES CXX)
3
4 add_library(hellolib STATIC hello.cpp)
5 add_executable(a.out main.cpp)
6 target_link_libraries(a.out PUBLIC hellolib)
```

```
-- The CXX compiler identification is GNU 11.1.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/bate/Codes/course/01/06/build
[ 25%] Building CXX object CMakeFiles/hellolib.dir/hello.cpp.o
[ 50%] Linking CXX static library libhellolib.a
[ 50%] Built target hellolib
[ 75%] Building CXX object CMakeFiles/a.out.dir/main.cpp.o
[100%] Linking CXX executable a.out
[100%] Built target a.out
Hello, world
```


为什么 C++ 需要声明

```
hello.cpp
1 #include <stdio>
2
3 void hello() {
4     printf("Hello, world\n");
5 }

main.cpp
1 #include <stdio>
2
3 void hello();
4
5 int main() {
6     hello();
7     return 0;
8 }
```

- 在多文件编译章中，说到了需要在 main.cpp 声明 hello() 才能引用。为什么？
 1. 因为需要知道函数的参数和返回值类型：这样才能支持**重载**，**隐式类型转换**等特性。例如 show(3)，如果声明了 void show(float x)，那么编译器知道把 3 转换成 3.0f 才能调用。
 2. 让编译器知道 hello 这个名字是一个**函数**，不是一个**变量**或者**类**的名字：这样当我写下 hello() 的时候，他知道我是想调用 hello 这个函数，而不是创建一个叫 hello 的类的对象。
- 其实，**C++** 是一种强烈依赖上下文信息的编程语言，举个例子：
- `vector < MyClass > a;` // 声明一个由 MyClass 组成的数组
- 如果编译器不知道 `vector` 是个模板类，那他完全可以把 `vector` 看做一个变量名，把 `<` 解释为小于号，从而理解成判断 ‘vector’ 这个变量的值是否小于 ‘MyClass’ 这个变量的值。
- 正因如此，我们常常可以在 C++ 代码中看见这样的写法：`typename decay<T>::type`
- 因为 T 是不确定的，导致编译器无法确定 `decay<T>` 的 `type` 是一个类型，还是一个值。因此用 `typename` 修饰来让编译器确信这是一个类型名.....

为什么需要头文件

- 为了使用 `hello` 这个函数，我们刚才在 `main.cpp` 里声明了 `void hello()`。
- 但是如果另一个文件 `other.cpp` 也需要用 `hello` 这个函数呢？也在里面声明一遍？
- 如果能够只写一遍，然后自动插入到需要用 `hello` 的那些 `.cpp` 里就好了.....

```
other.cpp
1 #include <stdio>
2
3 void hello();
4
5 void otherfunc() {
6     hello();
7 }
```

```
hello.cpp
1 #include <stdio>
2
3 void hello() {
4     printf("Hello, world\n");
5 }
```

```
main.cpp
1 #include <stdio>
2
3 void hello();
4
5 int main() {
6     hello();
7     return 0;
8 }
```


头文件 - 批量插入几行代码的硬核方式

- 没错，C 语言的前辈们也想到了，他们说，既然每个 .cpp 文件的这个部分是一模一样的，不如我把 `hello()` 的声明放到单独一个文件 `hello.h` 里，然后在需要用到 `hello()` 这个声明的地方，打上一个记号，`#include "hello.h"`。然后用一个小程序，自动在编译前把引号内的文件名 `hello.h` 的内容插入到记号所在的位置，这样不就只用编辑 `hello.h` 一次了嘛~
- 后来，这个编译前替换的步骤逐渐变成编译器的了一部分，称为预处理阶段，`#define` 定义的宏也是这个阶段处理的。
- 此外，在实现的文件 `hello.cpp` 中导入声明的文件 `hello.h` 是个好习惯，可以保证当 `hello.cpp` 被修改时，比如改成 `hello(int)`，编译器能够发现 `hello.h` 声明的 `hello()` 和定义的 `hello(int)` 不一样，避免“沉默的错误”。

```
other.cpp
1 #include <stdio>
2
3 #include "hello.h"
4
5 void otherfunc() {
6     hello();
7 }
```

编译前替换

```
hello.h
1 void hello();
```

编译前替换

```
hello.cpp
1 #include <stdio>
2
3 void hello() {
4     printf("Hello, world\n");
5 }
```

```
main.cpp
1 #include <stdio>
2
3 #include "hello.h"
4
5 int main() {
6     hello();
7     return 0;
8 }
```

头文件 - 批量插入几行代码的硬核方式

- 实际上 `cstdio` 也无非是提供了 `printf` 等一系列函数声明的头文件而已，实际的实现是在 `libc.so` 这个动态库里。其中 `<cstdio>` 这种形式表示**不要在当前目录下搜索**，只在系统目录里搜索，`"hello.h"` 这种形式则**优先搜索当前目录下有没有这个文件**，找不到再搜索系统目录。

用 `<>` 括号优先在系统的 `usr/include` 目录下查找，`" "` 就是在当前的目录下查找或者给定路径下查找

- 此外，在实现的文件 `hello.cpp` 中也导入声明的文件 `hello.h` 是个好习惯：
 - 可以保证当 `hello.cpp` 被修改时，比如改成 `hello(int)`，编译器能够发现 `hello.h` 声明的 `hello()` 和定义的 `hello(int)` 不一样，避免“沉默的错误”（虽然对支持重载的 `C++` 不奏效）
 - 可以让 `hello.cpp` 中的函数需要相互引用时，不需要关心定义的顺序。

```
other.cpp
1 #include <cstdio>
2
3 #include "hello.h"
4
5 void otherfunc() {
6     hello();
7 }
```

编译前替换

```
hello.h
1 void hello();
```

编译前替换

```
hello.cpp
1 #include <cstdio>
2
3 void hello() {
4     printf("Hello, world\n");
5 }
```

```
main.cpp
1 #include <cstdio>
2
3 #include "hello.h"
4
5 int main() {
6     hello();
7     return 0;
8 }
```

头文件进阶 - 递归地使用头文件

- 在 C++ 中常常用到很多的类，和函数一样，类的声明也会被放到头文件中。
- 有时候我们的函数声明需要使用到某些类，就需要用到声明了该类的头文件，像这样递归地 `#include` 即可：

```
main.cpp
1 #include <stdio>
2
3 #include "hello.h"
4
5 int main() {
6     MyClass mc;
7     mc.m_number = 42;
8     hello(mc);
9     return 0;
10 }
```

```
hello.h
1 #include "MyClass.h"
2
3 void hello(MyClass mc);
```

```
MyClass.h
1 struct MyClass {
2     int m_number;
3 };
```

预处理后变成:

```
main.cpp | hello.h | MyClass.h |
1 // .....
2 extern "C" int printf(const char *fmt, ...);
3 // .....
4
5 struct MyClass {
6     int m_number;
7 };
8
9 void hello(MyClass mc);
10
11 int main() {
12     MyClass mc;
13     mc.m_number = 42;
14     hello(mc);
15     return 0;
16 }
```


头文件进阶 - 递归地使用头文件（续）

- 但是这样造成一个问题，就是如果多个头文件都引用了 MyClass.h，那么 MyClass 会被重复定义两遍：

```
main.cpp
1 #include <stdio>
2
3 #include "hello.h"
4 #include "goodbye.h"
5
6 int main() {
7     MyClass mc;
8     mc.m_number = 42;
9     hello(mc);
10    goodbye(mc);
11    return 0;
12 }
```

```
hello.h
1 #include "MyClass.h"
2
3 void hello(MyClass mc);
```

```
goodbye.h
1 #include "MyClass.h"
2
3 void goodbye(MyClass mc);
```

```
MyClass.h
1 #pragma once
2
3 struct MyClass {
4     int m_number;
5 };
~
```

```
MyClass.h
1 #pragma once
2
3 struct MyClass {
4     int m_number;
5 };
~
```

```
In file included from /home/bate/Codes/course/01/09/goodbye.h:1,
                 from /home/bate/Codes/course/01/09/main.cpp:4:
/home/bate/Codes/course/01/09/MyClass.h:1:8: error: redefinition of 'struct MyClass'
1 | struct MyClass {
  | ~~~~~
In file included from /home/bate/Codes/course/01/09/hello.h:1,
                 from /home/bate/Codes/course/01/09/main.cpp:3:
/home/bate/Codes/course/01/09/MyClass.h:1:8: note: previous definition of 'struct MyClass'
1 | struct MyClass {
  | ~~~~~
```

头文件进阶 - 递归地使用头文件（再续）

- 解决方案：在头文件前面加上一行：`#pragma once`
- 这样当预处理器第二次读到同一个文件时，就会自动跳过
- 通常头文件都不想被重复导入，因此建议在每个头文件前加上这句话

```
main.cpp
1 #include <stdio>
2
3 #include "hello.h"
4 #include "goodbye.h"
5
6 int main() {
7     MyClass mc;
8     mc.m_number = 42;
9     hello(mc);
10    goodbye(mc);
11    return 0;
12 }
```

```
hello.h
1 #include "MyClass.h"
2
3 void hello(MyClass mc);
```

```
goodbye.h
1 #include "MyClass.h"
2
3 void goodbye(MyClass mc);
```

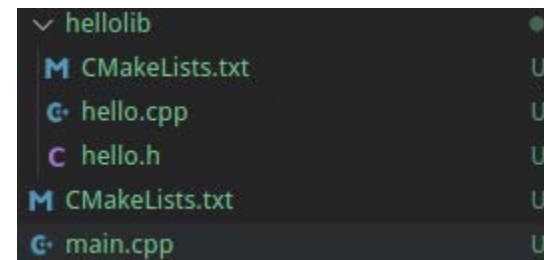
```
MyClass.h
1 #pragma once
2
3 struct MyClass {
4     int m_number;
5 };
~
```

```
MyClass.h
1 #pragma once
2
3 struct (自动跳过) {
4     int m_number;
5 };
~
```

```
-- Configuring done
-- Generating done
-- Build files have been written to: /home/bate/Codes/course/01/09
Consolidate compiler generated dependencies of target hellolib
[ 20%] Building CXX object CMakeFiles/hellolib.dir/hello.cpp.o
[ 40%] Linking CXX static library libhellolib.a
[ 60%] Built target hellolib
Consolidate compiler generated dependencies of target a.out
[ 80%] Building CXX object CMakeFiles/a.out.dir/main.cpp.o
[100%] Linking CXX executable a.out
[100%] Built target a.out
Hello, my number is 42!
Good bye, number 42!
```

CMake 中的子模块

- 复杂的工程中，我们需要划分子模块，通常一个库一个目录，比如：
- 这里我们把 `hellolib` 库的东西移到 `hellolib` 文件夹下了，里面的 `CMakeLists.txt` 定义了 `hellolib` 的生成规则。
- 要在根目录使用他，可以用 CMake 的 `add_subdirectory` 添加子目录，子目录也包含一个 `CMakeLists.txt`，其中定义的库在 `add_subdirectory` 之后就可以在外面使用。
- 子目录的 `CMakeLists.txt` 里路径名（比如 `hello.cpp`）都是相对路径，这也是很方便的一点。



CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.12)
2 project(hellocmake LANGUAGES CXX)
3
4 add_subdirectory(hellolib)
5
6 add_executable(a.out main.cpp)
7 target_link_libraries(a.out PUBLIC hellolib)
```

h/CMakeLists.txt

```
1 add_library(hellolib STATIC hello.cpp)
```


子模块的头文件如何处理

- 因为 hello.h 被移到了 hellolib 子文件夹里，因此 main.cpp 里也要改成：
- 如果要避免修改代码，我们可以通过 `target_include_directories` 指定
- a.out 的头文件搜索目录：(其中第一个 hellolib 是库名，第二个是目录)

```
6 add_executable(a.out main.cpp)
7 target_link_libraries(a.out PUBLIC hellolib)
8 target_include_directories(a.out PUBLIC hellolib)
```

- 这样甚至可以用 `<hello.h>` 来引用这个头文件了，因为通过 `target_include_directories` 指定的路径会被视为与系统路径等价：

```
main.cpp
1 #include <stdio>
2 #include <hello.h>
3
4 int main() {
5     hello();
6     return 0;
7 }
```

```
main.cpp
1 #include <stdio>
2
3 #include "hellolib/hello.h"
4
5 int main() {
6     hello();
7     return 0;
8 }
```

子模块的头文件如何处理（续）

- 但是这样如果另一个 b.out 也需要用 hellolib 这个库，难道也得再指定一遍搜索路径吗？
- 不需要，其实我们只需要定义 hellolib 的头文件搜索路径，引用他的可执行文件 CMake 会自动添加这个路径：

```
h/CMakeLists.txt
1 add_library(hellolib STATIC hello.cpp)
2 target_include_directories(hellolib PUBLIC .)
```

- 这里用了 . 表示当前路径，因为子目录里的路径是相对路径，类似还有 .. 表示上一层目录。
- 此外，如果不希望让引用 hellolib 的可执行文件自动添加这个路径，把 **PUBLIC** 改成 **PRIVATE** 即可。这就是他们的用途：决定一个属性要不要在被 link 的时候传播。

目标的一些其他选项

- 除了头文件搜索目录以外，还有这些选项，PUBLIC 和 PRIVATE 对他们同理：
- `target_include_directories(myapp PUBLIC /usr/include/eigen3)` # 添加头文件搜索目录
- `target_link_libraries(myapp PUBLIC hellolib)` # 添加要链接的库
- `target_add_definitions(myapp PUBLIC MY_MACRO=1)` # 添加一个宏定义
- `target_add_definitions(myapp PUBLIC -DMY_MACRO=1)` # 与 `MY_MACRO=1` 等价
- `target_compile_options(myapp PUBLIC -fopenmp)` # 添加编译器命令行选项
- `target_sources(myapp PUBLIC hello.cpp other.cpp)` # 添加要编译的源文件
- 以及可以通过下列指令（不推荐使用），把选项加到所有接下来的目标去：
- `include_directories(/opt/cuda/include)` # 添加头文件搜索目录
- `link_directories(/opt/cuda)` # 添加库文件的搜索路径
- `add_definitions(MY_MACRO=1)` # 添加一个宏定义
- `add_compile_options(-fopenmp)` # 添加编译器命令行选项

第三方库 - 作为纯头文件引入

- 有时候我们不满足于 C++ 标准库的功能，难免会用到一些第三方库。
- 最友好的一类库莫过于纯头文件库了，这里是一些好用的 **header-only** 库：
 1. **nothings/stb** - 大名鼎鼎的 **stb_image** 系列，涵盖图像，声音，字体等，只需单头文件！
 2. **Neargye/magic_enum** - 枚举类型的反射，如枚举转字符串等（实现方式很巧妙）
 3. **g-truc/glm** - 模仿 **GLSL** 语法的数学矢量/矩阵库（附带一些常用函数，随机数生成等）
 4. **Tencent/rapidjson** - 单纯的 **JSON** 库，甚至没依赖 **STL**（可定制性高，工程美学经典）
 5. **ericniebler/range-v3** - **C++20 ranges** 库就是受到他启发（完全是头文件组成）
 6. **fmtlib/fmt** - 格式化库，提供 **std::format** 的替代品（需要 **-DFMT_HEADER_ONLY**）
 7. **gabime/spdlog** - 能适配控制台，安卓等多后端的日志库（和 **fmt** 冲突！）
- 只需要把他们的 **include** 目录或头文件下载下来，然后 **include_directories(spdlog/include)** 即可。
- 缺点：函数直接实现在头文件里，没有提前编译，从而需要重复编译同样内容，编译时间长。

glm - 使用这个神奇的数学库

CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.12)
2 project(hellocmake LANGUAGES CXX)
3
4 add_executable(a.out main.cpp)
5 target_include_directories(a.out PUBLIC glm/include)
```

```
bate@archer ~/Codes/course/01/12 (master) $ ls
build CMakeLists.txt glm main.cpp run.sh
bate@archer ~/Codes/course/01/12 (master) $
```

main.cpp

```
1 #include <glm/vec3.hpp>
2 #include <iostream>
3
4 inline std::ostream &operator<<(std::ostream &os, glm::vec3 const &v) {
5     return os << v.x << ' ' << v.y << ' ' << v.z;
6 }
7
8 int main() {
9     glm::vec3 v(1, 2, 3);
10    v += 1;
11    std::cout << v << std::endl;
12    return 0;
13 }
```

```
bate@archer ~/Codes/course (master) $ git clone https://github.com/g-truc/glm.git
t --depth=1
Cloning into 'glm'...
remote: Enumerating objects: 1522, done.
remote: Counting objects: 100% (1522/1522), done.
remote: Compressing objects: 100% (773/773), done.
remote: Total 1522 (delta 943), reused 904 (delta 741), pack-reused 0
Receiving objects: 100% (1522/1522), 4.16 MiB | 1.96 MiB/s, done.
Resolving deltas: 100% (943/943), done.
bate@archer ~/Codes/course (master) $ rm -rf glm/.git
```

main.cpp

```
1 #include <glm/vec3.hpp>
2 #include <iostream>
3
4 int main() {
5     glm::vec3 v(1, 2, 3);
6     v += 1;
7     std::cout << v.x << ' ' << v.y << ' ' << v.z << std::endl;
8     return 0;
9 }
```

```
bate@archer ~/Codes/course/01/12 (master) $ s glm/include/glm/
CMakeLists.txt fwd.hpp mat2x2.hpp mat4x2.hpp trigonometric.hpp
common.hpp geometric.hpp mat2x3.hpp mat4x3.hpp vec2.hpp
detail glm.hpp mat2x4.hpp mat4x4.hpp vec3.hpp
exponential.hpp gtc mat3x2.hpp matrix.hpp vec4.hpp
ext gtx mat3x3.hpp packing.hpp vector_relational.hpp
ext.hpp integer.hpp mat3x4.hpp simd
```

```
-- Build files have been written to: /home/bate/Codes/cours
[ 50%] Building CXX object CMakeFiles/a.out.dir/main.cpp.o
[100%] Linking CXX executable a.out
[100%] Built target a.out
2 3 4
```

第三方库 - 作为子模块引入

- 第二友好的方式则是作为 CMake 子模块引入，也就是通过 `add_subdirectory`。
- 方法就是把那个项目（以fmt为例）的源码放到你工程的根目录：
- 这些库能够很好地支持作为子模块引入：

1. fmtlib/fmt - 格式化库，提供 `std::format` 的替代品
2. gabime/spdlog - 能适配控制台，安卓等多后端的日志库
3. ericniebler/range-v3 - C++20 ranges 库就是受到他启发
4. g-truc/glm - 模仿 GLSL 语法的数学矢量/矩阵库
5. abseil/abseil-cpp - 旨在补充标准库没有的常用功能
6. bombela/backward-cpp - 实现了 C++ 的堆栈回溯便于调试
7. google/googletest - 谷歌单元测试框架
8. google/benchmark - 谷歌性能评估框架
9. glfw/glfw - OpenGL 窗口和上下文管理
10. libigl/libigl - 各种图形学算法大合集

CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.12)
2 project(hellocmake LANGUAGES CXX)
3
4 add_subdirectory(fmt)
5
6 add_executable(a.out main.cpp)
7 target_link_libraries(a.out PUBLIC fmt)
```

```
bate@archer ~/Codes/course/01/12 (master) $ git clone https://github.com/fmtlib/
fmt.git --depth=1
Cloning into 'fmt'...
remote: Enumerating objects: 238, done.
remote: Counting objects: 100% (238/238), done.
remote: Compressing objects: 100% (229/229), done.
remote: Total 238 (delta 4), reused 139 (delta 1), pack-reused 0
Receiving objects: 100% (238/238), 846.96 KiB | 909.00 KiB/s, done.
Resolving deltas: 100% (4/4), done.
bate@archer ~/Codes/course/01/12 (master) $ ls
CMakeLists.txt  fmt  main.cpp  run.sh
bate@archer ~/Codes/course/01/12 (master) $ ls fmt
ChangeLog.rst  CONTRIBUTING.md  include  README.rst  support
CMakeLists.txt  doc  LICENSE.rst  src  test
bate@archer ~/Codes/course/01/12 (master) $
```


fmt - 使用这个神奇的格式化库

- `fmt::format` 的用法和 Python 的 `str.format` 大致相似:

main.cpp

```
1 #include <fmt/core.h>
2
3 int main() {
4     fmt::print("The answer is {}.\\n", 42);
5     return 0;
6 }
```

main.cpp+

```
1 #include <fmt/core.h>
2 #include <iostream>
3
4 int main() {
5     std::string msg = fmt::format("The answer is {}.\\n", 42);
6     std::cout << msg << std::endl;
7     return 0;
8 }
```

```
[ 20%] Building CXX object fmt/CMakeFiles/fmt.dir/src/format.cc.o
[ 40%] Building CXX object fmt/CMakeFiles/fmt.dir/src/os.cc.o
[ 60%] Linking CXX static library libfmt.a
[ 60%] Built target fmt
[ 80%] Building CXX object CMakeFiles/a.out.dir/main.cpp.o
[100%] Linking CXX executable a.out
[100%] Built target a.out
The answer is 42.
```

CMake - 引用系统中预安装的第三方库

- 可以通过 `find_package` 命令寻找系统中的包/库：
- `find_package(fmt REQUIRED)`
- `target_link_libraries(myexec PUBLIC fmt::fmt)`
- 为什么是 `fmt::fmt` 而不是简单的 `fmt`？
- 现代 CMake 认为一个包 (package) 可以提供多个库，又称组件 (components)，比如 TBB 这个包，就包含了 `tbb`, `tbbmalloc`, `tbbmalloc_proxy` 这三个组件。
- 因此为避免冲突，每个包都享有一个独立的名字空间，以 `::` 的分割（和 C++ 还挺像的）。
- 你可以指定要用哪几个组件：
- `find_package(TBB REQUIRED COMPONENTS tbb tbbmalloc REQUIRED)`
- `target_link_libraries(myexec PUBLIC TBB::tbb TBB::tbbmalloc)`

CMakeLists.txt

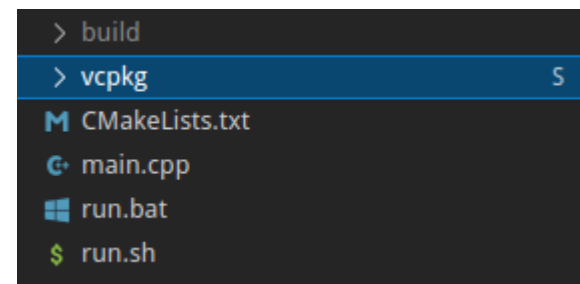
```
1 cmake_minimum_required(VERSION 3.12)
2 project(hellocmake LANGUAGES CXX)
3
4 add_executable(a.out main.cpp)
5
6 find_package(fmt REQUIRED)
7 target_link_libraries(a.out PUBLIC fmt::fmt)
```

第三方库 - 常用 package 列表

1. `fmt::fmt`
 2. `spdlog::spdlog`
 3. `range-v3::range-v3`
 4. `TBB::tbb`
 5. `OpenVDB::openvdb`
 6. `Boost::iostreams`
 7. `Eigen3::Eigen`
 8. `OpenMP::OpenMP_CXX`
- 不同的包之间常常有着依赖关系，而包管理器的作者为 `find package` 编写的脚本（例如 `/usr/lib/cmake/TBB/TBBConfig.cmake`）能够自动查找所有依赖，并利用刚刚提到的 `PUBLIC PRIVATE` 正确处理依赖项，比如如果你引用了 `OpenVDB::openvdb` 那么 `TBB::tbb` 也会被自动引用。
 - 其他包的引用格式和文档参考：
<https://cmake.org/cmake/help/latest/module/FindBLAS.html>

安装第三方库 - 包管理器

- Linux 可以用系统自带的包管理器（如 apt）安装 C++ 包。
- > pacman -S fmt
- Windows 则没有自带的包管理器。因此可以用跨平台的 vcpkg:
<https://github.com/microsoft/vcpkg>
- 使用方法：下载 vcpkg 的源码，放到你的项目根目录，像这样：
- > cd vcpkg
- > .\bootstrap-vcpkg.bat
- > .\vcpkg integrate install
- > .\vcpkg install fmt:x64-windows
- > cd ..
- > cmake -B build -DCMAKE_TOOLCHAIN_FILE="%CD%/vcpkg/scripts/buildsystems/vcpkg.cmake"



个人GitHub: <https://github.com/archibate>
Bilibili主页: <https://space.bilibili.com/263032155>
PPT和代码: <https://github.com/parallel101/course>

感谢观看!

presents by 彭于斌

