# C++20模块: 从C++14到C++20的编译效率跃升

陈天骢

2024年11月22日

**摘要**

在传统的头文件机制下，头文件重复编译等问题会导致C++14的项目编译缓慢；而模块(Modules)在C++20的引入显著减少了重复编译。本文通过对比C++14和C++20在编译效率上的差异，详细探讨模块的优势。首先，我们将介绍模块的基本概念和使用方法，然后通过实际案例展示模块在提高编译速度和节省系统资源方面的表现；最后，我们将综合模块的优势，展望模块在未来的C++中的应用前景，并提出一些可能的研究方向。

## 1 引言

C++作为一种广泛使用的编程语言，其编译效率一直是开发者关注的重点。在C++14及之前的版本中，头文件的使用导致了诸多问题，如大量的重复编译、依赖管理复杂等，这些问题在大规模项目中尤为突出。

C++20引入了模块(Modules)这一新特性，尝试解决这些问题，提高编译效率。模块是C++20引入的一种新的代码组织方式，它允许开发者将代码组织成独立的单元，这些单元在编译时只需处理一次。模块的使用可以显著减少重复编译，提高编译效率。

模块的引入标志着C++在编译系统上的重大革新。与传统的头文件机制不同，模块允许开发者将代码组织成独立的单元，这些单元**在编译时只需处理一次**，从而避免了重复编译的问题。此外，模块还提供了更好的依赖管理机制，使得代码的组织和维护更加高效。

先举输出Hello world!的文件为例说明模块的简单运用。使用头文件(hello_header.cpp)的写法如下：

```
#include <iostream>
int main(){
    std::cout<<"Hello world!"<<std::endl;
}
```

使用模块(hello_module.cpp)的写法如下：

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

通过 Complier Explorer(x86-64 clang 19.1.0)进行预编译后，我们发现：使用头文件的代码，待编译代码有足足36929行；而使用模块的代码，待编译代码仅有11行！

使用**头文件**时，需要将整个iostream文件内部的代码重新编译，导致C++编译速度慢；而使用**模块**时，std模块本身可编译，不需要将std内部代码再编译，编译速度大幅提高。

本文的后续部分安排如下：

在第二节中，我们将分析头文件存在的问题，并提出模块对应的解决方案。第三节将详细描述性能测试的方法、结果以及遇到的问题。最后，在第四节中，我们将总结本文的研究成果并得出结论。

## 2 头文件的问题和模块的解决方案

接下来分重复编译、二次编译、并行编译[1]三部分分析头文件的问题和模块对应的解决方案。

**1.重复编译导致编译缓慢：** 假如有 M 份源代码，每份代码都包含了头文件 string（这在项目中是很常见的），在整个项目的构建过程中，我们不得不对string做 M 次预处理、编译与汇编，这非常耗时。

不妨假设一个最坏情况：项目中有 N 个头文件和 M 份源代码，且每份源代码都包含了全部 N 个头文件，那么整个项目的编译时间复杂度就达到了 O(N*M)。这正是大型 C++ 工程构建缓慢的原因之一。

**引入模块以后**，模块文件本身也是一个编译单元，会被单独编译处理，不会被预处理加入到每份源代码中重复编译。假如有 M 份源代码和 N 个模块，编译的时间复杂度仅为 O(N+M)。

**2.缓慢的二次编译：** 在传统头文件机制中，头文件并不被视为一个可以独立编译的编译单元。例如：当源代码中包含头文件mylib.h时，编译器不会独立编译 mylib.h，而是将其与源文件一起预处理、解析和编译。这意味着，无论是首次编译还是二次编译，每次都需要重复整个过程。因此，第二次编译的时间和第一次是差不多的。

而引用模块后，这种问题将不复存在。模块文件在首次编译时会被单独编译为中间文件（如 .bmi 或 .pcm），之后的构建过程中，编译器只需加载这些中间文件，而不需要重新解析模块内容。这样一来，即使多次编译，也能显著缩短编译时间。

**3.无法并行编译、预编译：** 由于头文件机制仅仅简单的将头文件中的代码复制进主文件中，头文件中的代码会被外部代码影响。正由于头文件代码互相影响，则不同头文件不能并行编译，也无法预编译，这些因素都大大限制了编译速度。

模块的使用不仅可以提高编译效率，较头文件机制还有简化依赖管理、实现封装等作用。

# 3 性能测试与对比

为验证模块在提高编译效率方面的优势，我们进行了模拟性能测试。

编译器版本为Clang 19.1.0；测试对象为hello_header.cpp与hello_module.cpp（见引言）；分别使用C++14头文件和C++20模块进行编译；测试工具为Chorme Trace Viewer。

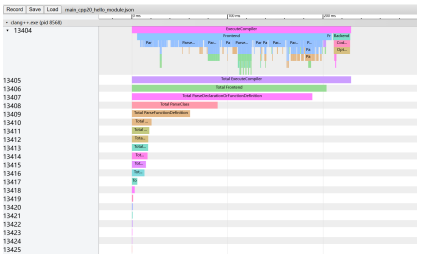
在编译代码时，使用 Clang 的 -ftime-trace 参数生成追踪文件。并在Chrome浏览器中的Trace Viewer加载json追踪文件，生成时间线（包括Frontend,Parse,InstantiateFunction）并进行对比。

对于C++20 modules，Clang无法直接处理，尝试先通过clang++ -std=c++20 -E hello_header.cpp -o preprocessed.cpp将其预编译（此过程也应生成一个json追踪文件），再使用-ftime-trace命令生成json追踪文件。

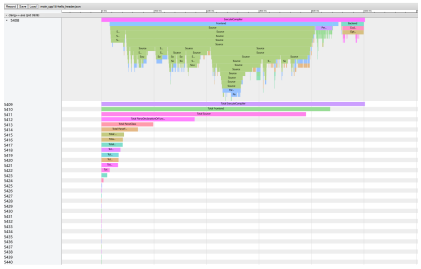
测试结果显示，使用C++20模块的项目**编译时间显著减少**。具体数据、运行图如下：

| 编译方式 | 总编译时间 | 单独预编译 | Frontend | Backend | Compiler.selftime |
|---|---|---|---|---|---|
| 头文件 | 502.152 | 0 | 435.911 | 43.129 | 23.112 |
| 模块 | 238.697 | 9.289 | 203.716 | 18.209 | 7.483 |

表 1: hello_module.cpp与hello_header.cpp编译时间对比(单位:ms)



(a) hello_module.cpp编译运行图



(b) hello_header.cpp编译运行图

图 1: hello_module.cpp与hello_header.cpp编译运行图

如表1与图1(a)(b)所示，两程序的运行过程包括ExecuteCompiler.selftime（在Frontend前、Frontend与Backend中、Backend后），Frontend（分为无缝连接的两段），Backend。

*Frontend*:代码分析后生成中间表示(IR),耗时最长,包括ParseClass等.

*Backend*:优化中间表示(IR)并将其转化为目标机器代码,再进行优化,包括 CodeGenPasses、OptModule、RunPass等.

*ExecuteCompiler.selftime*:编译器本身代码的执行时间.

模块实现在这三阶段均显著快于头文件实现。表明模块在代码逻辑的优化、转化为目标机器代码的速度均强于头文件实现。

在机理上，模块代码消耗时间以ParseDeclarationOrFunctionDefinition、ParseClass 为主，即处理声明或函数定义；头文件代码消耗时间以Source为主，即源代码的处理。

综合引言中Complier Explorer显示的预编译代码行数差距与Trace Viewer得到的运行时间差距，我们得到模块在**提高编译速度和节省系统资源方面具有显著优势**。

在性能测试的过程中，还出现了以下三个问题：

**1.生成json文件中出现错误：** 在生成json时间追踪文件中出现了多次错误，显示如下：

`error LNK2019(2001)：无法解析的外部符号`

此类Error，在hello_header.cpp中出现38次，在hello_module.cpp中出现33次。但并不影响两程序的编译与运行，相关研究可作未来方向。

**2.两种文件的大小差异：** hello_header.cpp生成的exe文件大小为67KB，而hello_module.cpp生成的exe文件大小为194KB，近乎前者3倍。

这可能与std modules和iostream头文件大小相关，可能与编译器默认的处理方式，也可能与exe的执行方式相关，可作未来研究。

**3.缺乏大量级比较途径：** 碍于目前模块生态的不成熟，难以找到完全或大多使用模块的C++项目，无法在大量级上比较模块法与头文件方法的内存占用、性能。

# 4 结论

本文通过对比C++14和C++20在编译效率上的差异，详细探讨了模块的优势。

实验结果表明，由于模块的封装性与可执行性，模块在提高编译速度和节省系统资源方面具有显著优势，在头文件与模块的编译时间对比上尤为显著。

目前，C++项目中模块生态尚未成熟，大部分代码仍依赖头文件，仅少部分使用模块。其原因在于C++模块类型的大型项目往往需要从头开始编写，难以摆脱头文件形成完整生态，而与模块配套的工具也尚未完善（例如使用 Clang++ 时暂无法直接编译 import std; 代码）。这种现象导致模块的并行编译、预编译、二次编译和封装等优势难以发挥，同时在模块文件调用过程中仍会多次使用头文件。

未来，随着模块生态的日渐完善，模块有望成为C++及其他编程语言中代码组织和编译的主流方式；我们也期待在大型项目中通过对模块与头文件方案在编译时间、内存占用和性能上的比较，进一步验证其优势。

# 参考文献

[1] 告别头文件！c++ modules 实战解析. https://openanolis.cn/video/1210525868042378600, 2024. Accessed: 2024-10-20.