补编 —— 模块（modules） C++20

一直以来，在我的个人世界中，C++编译器就是最强大的编译器，它们灵活、稳定并且十分高效。但不幸的是，源代码引入方式的落后导致C++编译器在面对巨型工程的时候总是力不从心，如果读者编译过Chromium、QT、LLVM这种规模的项目应该知道我的意思。

代码引入的问题

在解释C++代码引入的问题之前，我们先看一段这样一段代码：

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

上面是一段最简单的“Hello world”的代码，算上空格和换行字符一共仅仅70个字符。但是编译器在编译这份代码的时候却不能只解析这70个字符，它还需要处理`<iostream>`这个头文件。处理头文件的方式和宏一样，预处理器会把这个头文件完整的替换到源代码中，并且重新扫描源代码再替换源代码中新增的`#include`的头文件，一直递归处理下去，直到所有的替换都完成。

我们可以使用GCC生成替换后的文件：

```
gcc -E -P test.cpp -o expand.cpp
```

然后会发现整个expand.cpp文件有717192个字符！而我们的代码在这份代码中占据不到万分之一，但现实是这正是C++编译器需要处理的内容。事实上，大多数情况下编译器要处理的头文件内容要比原本的代码要多的多，真正我们自己编写的代码不到头文件的十分之一也是很常见的事情。一个更大的问题是，几乎所有的源代码都会包含一些头文件，比如标准库头文件，编译器不得不做巨量重复的工作。

像chromium这样的巨型项目会采用一些办法减少这种情况的出现，例如Jumbo/Unity builds，原理上就是讲多个源文件包含在一个文件中，这样多次的递归替换就能够合并为一次，事实证明这种方式非常有效，chromium的编译时间缩减了50%。但是这种方式对于代码组织非常苛刻，稍有不慎就会造成编译错误。

原始的头文件替换出了造成了编译低效之外，还有一些问题是一个现代语言需要解决的，例如：难以做到组件化、无法做代码隔离、难以支持现代语义感知开发工具等。C++急需一种高效的新方式代替原始的代码替换。

模块介绍

事实上，C++委员会早就发现了这些问题，并且在2007年就逐步开展了研究工作，只不过进度非常缓慢，直到2018年C++委员会才确定了最终草案（微软提供的方案），并在C++20标准中引入了模块。

我们通常可以认为，一个程序是由一组翻译单元（Translated units）组合而成的。这些翻译单元在没有额外信息的情况下是互相独立的，要将他们联系到一起需要这些翻译单元声明外部名称，编译器和链接器就是通过这些外部名称把独立的翻译单元组合起来的，而模块就可以认为是一个或者一组独立的翻译单元以及一组外部名称的组合体。那么模块名（Module name）就是引用组合体符号，模块单元（Module unit）其实就是组合体的翻译单元，模块接口单元（Module interface unit）很显然就是组合体里的那一组外部名称。

正规来说，一个模块由模块单元组成，模块单元分为模块接口单元和模块实现单元（Module implementation unit）。另外一个模块可以有多个模块分区，模块分区也是模块单元，模块分区的目的是方便模块代码的组织。对于每个模块，必须有一个没有分区的模块接口单元，该模块单元称为主模块接口单元。 导入一个模块，实际上导入的就是主模块的接口。

模块的语法

模块的语法应该算是非常简单的了，关键字包括`export`、`import`和`module`，其中`module`可以用来定义模块名、模块分区和模块片段，先来看看定义模块名：

```
module MyModule;
```

上面的代码定义了一个名为`MyModule`的模块单元，但是请注意这个模块不能作为主模块接口单元，因为定义主模块接口单元必须加上`export`：

```
export module MyModule;
```

注意，我们以后可能会在一些库中看到如下命名：

```
import std.core;
```

这里的`std.core`是一个模块名，看上去表达的是标准库中的核心模块，`.`在其中表示层次关系，但是其注意，这里的`.`并没有任何的语法规定，它在这纯粹是为了一种层次。

在定义了模块名之后，就可以导出指定名称了：

```
// mymodule.ixx
export module MyModule;

export int x = 1;

export int foo() { return 2; }

export class bar {
public:
    int run() { return 3; }
};

export namespace baz {
    int foo() { return 4; }
}
```

上面的代码使用`export`说明符导出了变量、函数、类以及命名空间，这些名称都是可以导出的。其他源文件可以使用`import`说明符导入这些名称：

```
// test.cpp
import MyModule;
int main()
{
    int y = x + foo() + bar().run() + baz::foo();
}
```

编译运行上面的代码，y的最终结果为10。当然，每个名称都依次使用`export`导出并不方便，标准还提供了更加简洁的写法：

```
// mymodule.ixx
export module MyModule;

export {
    int x = 1;

    int foo() { return 2; }

    class bar {
    public:
        int run() { return 3; }
    };

    namespace baz {
        int foo() { return 4; }
    }
}
```

注意，没有`export`的名称是不能被`import`到其他源代码中的：

```
// mymodule.ixx
export module MyModule;

export {
    ...
}

int z = 5;

// test.cpp
import MyModule;
int main()
{
    int y = x + foo() + bar().run() + baz::foo() + z; // 编译错误
}
```

这里test.cpp会编译报错，编译器会提示找不到标识符`z`。

`import`说明符不仅能引入模块，也能引入头文件，例如：

```
// mymodule.ixx
export module MyModule;
import <iostream>;
export {
    int x = 1;

    int foo() { return 2; }

    class bar {
    public:
        int run() { return 3; }
    };
```

```
    namespace baz {
        int foo() { return 4; }
    }

    void print(int n) { std::cout << n; }
}

// test.cpp
import MyModule;
int main()
{
    int y = x + foo() + bar().run() + baz::foo();
    print(y);
}
```

请注意，这里使用了`import`来引入`<iostream>`而不是使用`#include`。在模块单元中不要使用`#include`来引入头文件因为这样会导致这些内容成为模块单元的一部分。

另外还有一个地方需要特别注意，`import`进来的头文件是不会被源文件中的宏修改的，例如：

```
// mymodule.ixx
export module MyModule;
import <iostream>;
export {
#ifdef OUTPUT_HELLO
    void print() { std::cout << "hello"; }
#else
    void print() { std::cout << "world"; }
#endif
}

// test.cpp
#define OUTPUT_HELLO
import MyModule;
int main()
{
    print();
}
```

上面这段代码在`test.cpp`中定义了宏`OUTPUT_HELLO`，然后`import`了`MyModule`模块，如果`OUTPUT_HELLO`能够影响引入的模块，那么运行结果输出`hello`，否则输出`world`。编译运行这段代码会发现最终结果为`world`，`import`的内容不受宏的影响。但是，如果确实有这样的需求该怎么做呢？标准提供了一种叫做模块片段机制，模块片段通常用来做一些配置相关的工作，它通过`module;`开始，注意这里的`module`后直接跟着分号而没有模块名：

```
module;
// module fragment begin
#define SOME_CONFIG 20211102
#include <some_header>
// module fragment end
export module MyModule;
export {
...
}
```

模块片段还可以分为全局和私有，上面的代码编写的是全局的模块片段，要设置私有代码片段需要叫上`private`：

```
module : private;
```

标准规定，私有模块片段只能出现在主模块接口单元中，并且具有私有模块片段的模块单元应是其模块的唯一模块单元。

最后，让我们来看一看什么是模块分区。如果要导出的模块内容很多，我们不能将所有的代码放到一个文件中，需要将其按照逻辑做合理的物理分割，这个时候就需要用到模块分区了，请看下面的例子：

```
// part1.ixx
export module MyModule:part1;

void foo_impl() {}
export void foo() { foo_impl(); }

// part2.ixx
export module MyModule:part2;
void bar() {}

// mymodule.ixx
export module MyModule;
export import :part1;
import :part2;

export void print() {
    foo();
    bar();
}

// test.cpp
import MyModule;
int main()
{
    print();
    foo();
}
```

在上面的代码中，`part1.ixx`和`part1.ixx`的模块名分别为`MyModule:part1`和`MyModule:part2`，其中`MyModule`当然就是模块名，而紧跟在`:`后的名称则是它们的分区名。主模块接口单元可以通过`import`将模块分区合并到主模块接口单元中，并且无论模块分区是否导出了它的内容，它的内容都是对主模块接口单元可见的，所以`print`函数可以调用`bar`函数。

另外，主模块接口单元还可以决定直接导出分区定义的接口，比如代码中的：

```
export import :part1;
```

这样模块分区`part1`的函数`foo`也成为了导出接口。