# C++ module学习笔记

d41d8c



说是学习，结果只是把规范抄了一遍……。不保证正确，保证不准确。

划重点：

模块接口： `export module M;`
导出声明： `export foo bar;`
导入模块： `import M;`
如果有东西希望被模块外部当成不存在（最常见的大概是某些类的定义），可以放到模块实现文件： `module M;` ，或者偷懒用 `module :private;`

## 模块声明

可以按有没有 `export` 、有没有冒号分成四种：

```
export module M;   // (1)
module M;          // (2)
export module M:B; // (3)
module M:B;        // (4)
```

声明当前翻译单元（translation unit）是一个模块单元（module unit）。

模块名可以带点号： `module M.foo;` 。不能带关键字或 `module` 或 `import` 。 不要用 std 或 std接数字 开头的模块名，这些是给标准用的。 不要用保留字（带双下划线的名字 或 以下划线+大写字母开头的名字），这些是给编译器用的。

带export的叫做“module interface unit”，不带export的叫做“module implementation unit”。只有前者能包含 `export int f();` 这样的导出声明（export-declaration）。

带冒号的叫做“module partition”，在模块M内部可以通过 `import :B;` 这样来包含。M的module partition只能在模块M之内使用，模块M之外看不到M的module partition。

`export module M;` 这样（带export且不带冒号）的模块单元叫做“primary module interface unit”，一个模块必须有且只有一个primary module interface unit。

primary module interface unit必须直接或间接包含模块M的所有导出声明。

直接包含： `export int f();` 。
间接包含： `export import :B;` 。另一个文件中： `export module M:B; export int f();` 。

模块声明一般写在文件开头。不过出于兼容性考虑，如果一个模块要 `#include` 其他文件的话，可以在文件开头用 `module ;` 的特殊语法：

```
module;
#include <cstdio>
#include "foo/bar.h"
module M; // 模块声明
// 属于模块M的各种东西
```

除此之外模块声明不可以出现在其他地方。

模块声明之前 `#include` 的文件不属于这个模块。这些文件里，没有被模块用到的东西会被当成不存在。（术语叫decl-reachable，具体有一套复杂的规则。）

不属于任何模块的东西组成了global module。 模块内的 `extern "C"` / `extern "C++"` 声明和replaceable allocation/deallocation function也属于global module。 namespace声明大概因为是可以合并的，也算作属于global module。

## 导出声明

```
export int f();
export { int f(); int g(); }
```

使得声明在模块外可见。

虽然这里拿 `int f();` 和 `int g();` 做例子，但实际上各种声明都可以。不过嵌套export声明是不允许的。

```
export { export int f(); } // 这样是**不允许**的
```

export可以出现在普通的namespace之内。这样导出的名字可以在模块外用 `ns::f` 来使用（当然也可以 `using ns::f;` 或者 `using namespace ns;` ）。

```
namespace ns { export int f(); }
export namespace ns { int f(); } // 效果和上一句一样
```

模块M的所有导出声明叫做模块M的“interface”。

用 `export` 导出的名字不可以有internal linkage。

`export using namespace N;` 因为没有声明任何名字，所以不允许。 同理 `export static_assert(true);` 也是不可以的（注意这导致 `export namespace foo { int x; static_assert(true); }` 也不被允许）。

`export using ns::foo;` 中的 `ns::foo` 必须有external linkage，不能是模块内未导出的名字。

export类型别名（typedef或者using）是可以的，即使类型本身没有导出。

export必须在第一次声明的时候就指定，之后声明同一个东西的时候不需要写export（这些声明会自动导出）。

```
export module M;
struct S; // 没有导出 S
export using T = S; // OK

struct S { int n; };
export typedef S S; // OK，导出了类型别名S
export struct S; // 错误，struct S在第一次声明的时候没有用export
```

## 导入声明

三种：

```
import M;        // (1)
import :B;       // (2)
import <vector>; // (3)
```

(1): 导入模块M。
(2): 导入当前模块的module partition `:B` （见“模块声明”对module partition的介绍）。
(3): 导入头文件。

模块单元中import声明必须位于其他声明之前。其他翻译单元可以把import放在#include之类的后面，但不放在开头可能导致不直观的行为。

三种都可以加 `export` ， `export import foo;` 表示当前模块导出了翻译单元 `foo` 。比如说 `export import <vector>;` 表示当前模块导出了 `<vector>` 。

如果模块T之外的某个翻译单元导入了模块T，那么它也同时导入了模块T中用 `export import` 导出的所有单元。 如果模块T之中的某个模块单元导入了同模块的另一个单元，那么它也同时导入了那个单元import的所有单元。（因为是同模块所以export没有影响。） 此规则具有传递性。

(3)这样导入的头文件叫做header unit。

header unit默认导出里面的所有声明。不同于 `export` ，有internal linkage的名字也会被导出，但是不能odr-use。

由于预处理器的特殊规则，header unit之中的宏也会被“导出”，不过此规则大概没有传递性。

不是所有的头文件都可以当成header unit导入，标准只要求标准库头文件（不含C库）是可导入的。 对于可导入的头文件 `<foo>` ，编译器可以把 `#include <foo>` 改写成 `import <foo>;` 。

`module M; import M;` 是不允许的。但是 `module M;` 会自动import自己的primary module interface unit。

## global module fragment

就是模块声明之前 `#include` 的东西，见“模块声明”一节。

```
module;
#include <cstdio> // 归入 global module fragment
#include "foo/bar.h" // 归入 global module fragment
module M;
// ...
```

## private module fragment

以 `module : private ;` 开头的东西，必须放在其他声明之后。 private module fragment之中的声明在模块外会被当作不存在（不是不可见而是不存在，见“reachable”一节）。

```
export module foo;
export struct X;
export X& f();
module :private;
```

```
struct X {};
X& f() { static X a; return a; }
```

上例中，模块foo外的单元会把X的定义当成不存在，所以X会被视作incomplete type（不能定义X类型的对象）。而模块内则可以定义X类型的对象。

如果用了private module fragment，整个模块必须定义在一个单元里。

## Reachable

对于模块单元：在某一个位置P之前直接或间接导入（import）的所有模块单元都是可达（reachable）的。

对于声明：前面说了

global module fragment里，没有被模块用到（不是decl-reachable）的声明会被当成不存在。
private module fragment里的声明，在模块外会被当成不存在。

在某一个位置P，同一个翻译单元中所有位于P之前的声明，加上所有可达的模块单元中不符合上面两条的声明，都是可达（reachable）的。

注意可达（reachable）与可见（visible）不同，如果一个声明没有被导出（export），那么它在模块外不可见，但它也许是可达的。

比如：

```
export module foo;
struct X {};
export X& f();
```

因为没有 `export struct X;` 所以外部不能使用 `X` 这个类名，但是 `auto x = f();` 是可以的。

对比上面private module fragment的例子： 在private module fragment的例子中， `struct X {};` 不可达，所以不允许 `auto x = f();` ， 但是因为X的声明可见（ `export struct X;` ），所以 `X& x = f();` 是可以的。