

01 | Modules（上）：C++模块化问题的前世今生

2023-01-16 卢誉声 来自北京



《现代C++20实战高手课》

[课程介绍 >](#)



讲述：卢誉声

时长 16:30 大小 15.07M



你好，我是卢誉声。

今天是第一讲，我们会从 C++20 中的核心特性变更——Modules 模块开始，了解现代 C++ 的编程范式、编程思想及现实问题解决方法上的革新。

我们都知道，无论是编写 C 还是 C++ 程序，总少不了使用 `include` 头文件机制，这是 C/C++ 编程中组织代码的最基本方式，用法简单、直接，而且符合直觉。但很多人不知道的是，其实 `include` 头文件机制不仅有坑，在编程过程中还会出现一些难以解决的两难问题。

接下来，我们就从这个“简单”机制开始探讨代码组织方式的门道，看看里面究竟存在哪些问题，语言设计者和广大语言使用者又是如何应对的，对这些问题的思考，将串联起我们关于 C++ 代码组织方案的所有知识点，也终将引出我们的主角——Modules（课程配套代码点击[这里](#)获取）。

首先来看看整个故事的背景，include 头文件机制，它的出现是为了什么？

万物始于 include



作为 C 语言的超集，C++ 语言从设计之初就沿袭了 C 语言的 include 头文件机制，即通过包含头文件的方式来引用其他符号，包括类型定义、接口声明这样的代码，起到分离符号定义与具体实现的作用。

早期，能放在头文件中的符号比较有限，头文件设计是足以支撑系统设计的。但是为了提高运行时性能，开发者也会考虑将实现直接放在头文件中。

一开始这看起来似乎没什么，但随着软件技术的发展，C++ 从 C++98 过渡到现代 C++ 之后，越来越多的特性可以被定义或声明在头文件中，头文件对现代软件开发的支持就显得捉襟见肘了。

首先，由于模板元编程的特性，模板类、模板函数及其实现，我们往往需要全部定义在头文件中。

我们还可以在头文件中定义编译时（compile-time）常量表达（constexpr）的变量和函数（constexpr 函数也可以在运行时调用）。

• 常量表达

常量表达，指的是现代C++倾向于将运行时（runtime）重复出现的计算从运行时迁移至编译时，这些重复出现的计算都有一个特点，运算结果总是相同的。举个例子，我们可以在编译时计算好一个正整数的阶乘（比如 $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$ ），无论何时执行程序，阶乘的计算结果就在那。

这个计算会发生在C++代码的编译期，也就是说，编译进程序的是计算的结果。

另外，类型推导 auto、inline 函数与变量、宏也可以被定义在头文件中。

这样看来，头文件包罗万象实至名归，但伴随而来的是一系列问题。



第一个问题是**模糊的模块划分**。

传统的 `include` 头文件机制并没有提供清晰的模块划分能力，如果要在一份代码中使用来自不同目录的相同符号定义，编译器，在链接阶段没有足够的信息来区分这些相同符号定义的强弱关系，会出现符号覆盖或者链接错误的问题。

第二个问题是**依赖符号导入顺序**。

一个头文件可能会因为导入符号顺序的不同，影响到代码中包含的另一个语义，如宏定义就有可能影响到导入符号的顺序。因此，在大规模 `C++` 项目中，有时候导入头文件的顺序都是有讲究的，我们往往不能控制所有的代码编写工作，而来自不同开发团队的头文件可能会互相影响，导致头文件难以组合。

第三个问题是**编译效率低下**。

`C++` 语言由核心语言特性与库构成，那些可以定义在头文件中的特性（如类型推导 `auto`、模板函数和模板类等）会对编译速度造成影响，而导入一个看似简单的 `STL` 库头文件，如 `sstream`，在编译期展开后会达到数万行代码。`include` 头文件会让这些计算、解析在编译阶段反复发生，进一步拖慢甚至拖垮编译效率。

第四个问题是**命名空间污染**。

初学 `C++` 的时候，我们常常会为了方便大量使用 `using namespace` 来简化后续代码的编写工作，如果在这些库中含有与程序中定义的全局实体同名的实体，或者不同库之间有同名的实体，在编译时会出现名字冲突。如果在头文件中使用了 `using namespace`，甚至会导致所有直接或间接包含了该头文件的代码都受到影响，产生不可预计的后果。

所以，为了解决 `C++` 头文件中的符号隔离问题，简单的理解 `include` 头文件是远不够的，问题实在不少。接下来，让我们看看在传统 `C++` 编程中是如何解决模块化问题的，进而引出 `C++ Modules`。

传统 `C++` 模块化解决方案

事实上，在 C++20 以前，即便是现代 C++（C++11~C++17）也没有在模块化方面有什么实质性的突破，一直没有统一的抽象模块概念。



但开发者在这么多年的实践中确总结出了一套较为行之有效的经验，可以在一定程度上表示“模块”，以达到两大目的。

一个目的是划分业务逻辑代码，将大规模的代码划分为小规模代码，通过层层划分和模块组织，让每个模块代码足够内聚，专注自身业务，最终提升代码的可维护性。

第二个目的是提升代码的复用性，我们可能会抽离出很多功能模块，供其他模块调用，C++ 的标准库正是这种“模块”的代表，最终可以减少系统中的重复代码，提升系统的稳定性和可维护性。

既然 C++ 没有提供标准的模块特性，那么传统项目中，我们会使用哪些基础设施来模拟模块呢？

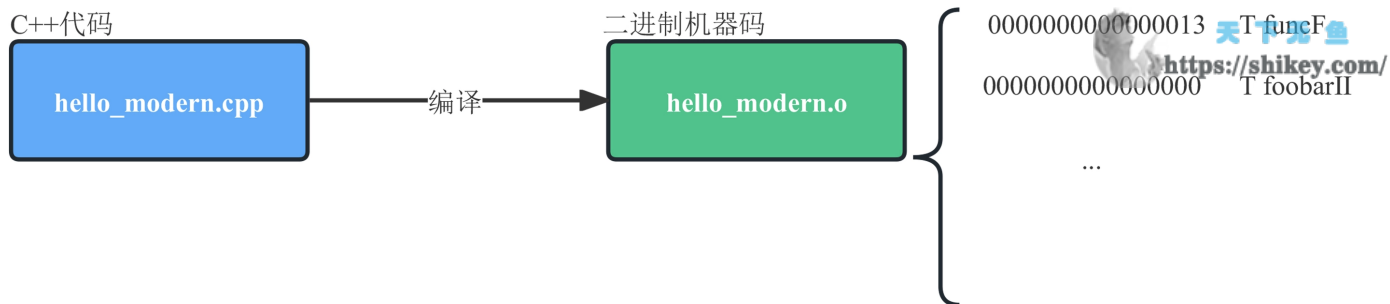
结合实践经验，我们来看看几个常用特性，包括应对模块划分与符号复用的编译链接、头文件，应对符号隔离的命名空间。

分别编译 / 链接 / 头文件

刚才提到模块化的两个目的，业务逻辑划分、代码复用。为了实现这两个目的，我们首先会利用从 C 开始就支持的编译 - 链接两阶段的构建过程。

C++ 编译过程中最基础的概念就是“编译单元”，所以每个实现文件（通常以.cpp 结尾）都是完全独立的编译单元。我们会先在“编译”这个阶段对每个编译单元进行独立编译，生成独立的目标文件，也就是将 C++ 代码转换成二进制机器码，在这个过程中，每个目标文件中的函数在编译过程中会生成一个“符号”，而每个符号包含一个名称和函数代码的首地址。

假设我们想在编译单元 A（简称 A）中引用编译单元 B（简称 B）中的函数，只需要确保引用的函数同名即可。



在 A 的编译过程中，如果引用的函数不存在，就会把函数的调用位置空出来，等到链接的时候，链接器会从其他的编译单元，比如 B 中，搜索编译时空出位置的函数符号。如果能找到，就将符号地址填入空出来的部分，如果找不到就会报出链接错误。

这个过程具体是怎么运作的呢？我们举个具体例子来看看，在 B 中定义了一个函数 `add`，A 中引用这个函数 `add`。

a.cpp 代码是这样的。

复制代码

```
1 #include <stdint>
2 #include <iostream>
3 extern int32_t add(int32_t a, int32_t b);
4 int main() {
5     int32_t sum = add(1, 2);
6     std::cout << "sum: " << sum << std::endl;
7     return 0;
8 }
```

b.cpp 代码是这样的。

复制代码

```
1 #include <stdint>
2 int32_t add(int32_t a, int32_t b) {
3     return a + b;
4 }
```

可以看到，A 中其实也“声明”了一个函数 `add`，只不过声明的时候加了一个 `extern` 修饰符，并没有函数定义。

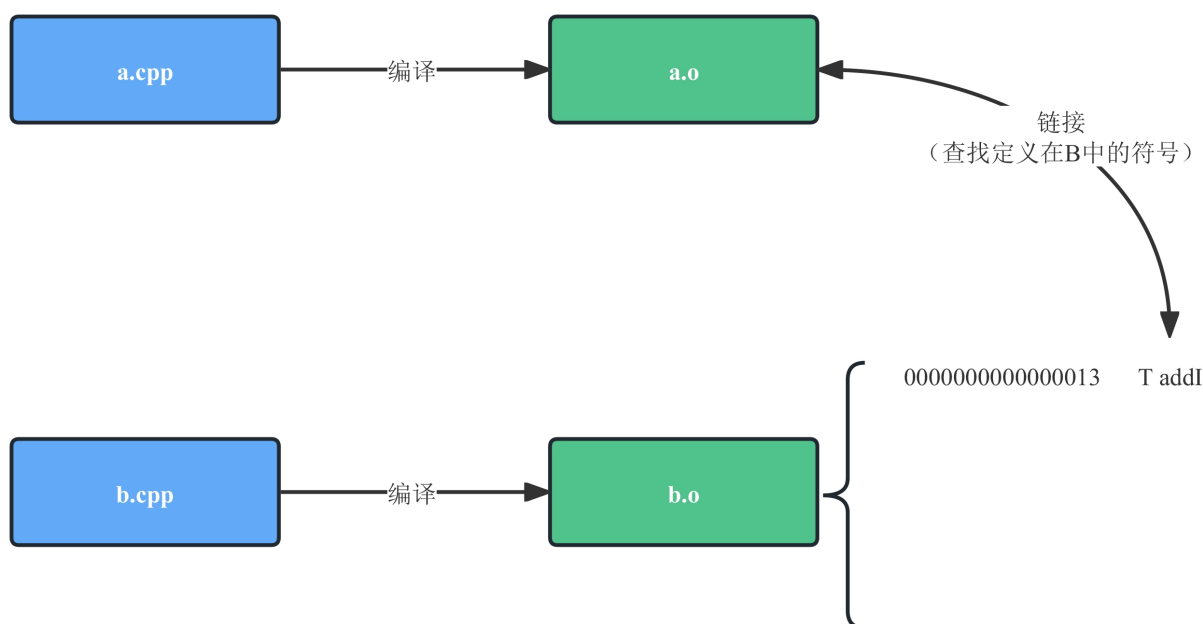


这是因为在编译 A 的时候，编译器并不知道 B 的存在，我们需要通过这种方式“告知”编译器其实有这个函数 `add`，只是在其他的编译单元中，准备等到链接时再使用。这样一来，虽然编译器没有找到这个函数的定义，但是会暂时“放过”它，在生成的函数调用机器码中会将这个符号的地址空出来，等到链接的时候再来填充。

如果你使用 `gcc` 编译这两个源代码文件，可以看到 `b.o` 中会生成一个名为 `add` 的符号，这个符号就是准备在链接过程中使用的。

好，编译完成，我们继续执行链接动作。

此时，链接器首先将编译生成的 A 和 B 的目标文件（`a.o` 和 `b.o`）组装在一起，然后开始“填坑”——填补在编译过程中空出来的符号调用的符号地址。编译器会从 `b.o` 的代码中寻找符号 `add`，找到使用这个符号的地址，去填补 `a.o` 在编译过程中空出来的调用符号的地址。



链接完成后，所有二进制代码中预留的地址全部都要被修补，如果所有编译单元中都找不到这个符号，就会在链接阶段报错。最后，链接生成的二进制代码不允许出现空缺的调用地址。

Hmm，这种模式似乎可以解决业务逻辑划分问题，但有一个问题——现在，如果有一个新的编译单元 **C**，**C** 也希望使用 **B** 中定义的函数，那么我们也需要在 **C** 中重写一遍 **add** 函数的声明吗？



的确需要如此，因为编译器并不知道其他编译单元的定义。

不过，这会让我们引用其他编译单元的函数变得非常麻烦，而且更大的问题是，如果在引用符号的编译单元中写错了声明，只要符号一样，链接的时候也不会报错。怎么办呢？这个时候就要用头文件来解决这个问题了。

我们可以先定义一个头文件 **b.h**。

复制代码

```
1 #ifndef _MODULE_B_H
2 #define _MODULE_B_H
3 #include <cstdint>
4 extern int32_t add(int32_t a, int32_t b);
5 #endif //_MODULE_B_H
```

接着修改 **a.cpp** 和 **b.cpp**（其中 **b.cpp** 暂无变化），修改后代码是这样。

复制代码


```
1 #include <cstdint>
2 #include <iostream>
3 #include "b.h"
4 int main() {
5     int32_t sum = add(1, 2);
6     std::cout << "sum: " << sum << std::endl;
7     return 0;
8 }
```

接着编译并链接写好的代码，命令是这样。

复制代码

```
1 g++ -o add a.cpp b.cpp -std=c++11
```

以后 **B** 中增加了新的函数，只需在 **b.h** 中补充相应的声明即可。

这样在每个引用 **B** 的编译单元，都不用重复声明这个函数了，不过我们也需要知道这其实是通过“包含”头文件代码这种非常“低级”的技术方式实现的。而且这种方式可能还会产生一个问题，两个编译单元的符号可能会重复！也就是我们前面提到的“命名空间污染”。 <https://shikey.com/>

比如我们在 **a.cpp** 中也定义一个 **add** 函数，然后进行编译、链接。

 复制代码

```
1 #include <cstdint>
2 #include <iostream>
3 #include "b.h"
4
5 int32_t add(int32_t a, int32_t b) {
6     return a + b;
7 }
8
9 int main() {
10     int32_t sum = add(1, 2);
11     std::cout << "sum: " << sum << std::endl;
12
13     return 0;
14 }
```

编译过程非常顺利。但是，在链接时，哦？出错了。

```
> g++ a.cpp b.cpp -o v3
/usr/bin/ld: /tmp/ccRtHrtY.o: in function `add(int, int)':
b.cpp:(.text+0x0): multiple definition of `add(int, int)'; /tmp/ccnZU3xV.o:a.cpp:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

这是因为 **a.o** 中引用了 **add** 这个符号，但是 **b.o** 和 **a.o** 中都包含这个符号，就导致了冲突，因为链接器不知道 **a.o** 中想要使用的 **add** 到底是 **b.o** 中的还是 **a.o** 中的。

这就是所谓的符号隔离问题。那么一般应该如何解决呢？

在 **C** 和早期的 **C++** 中我们的解决方案非常简单粗暴，那就是添加前缀。比如我们在 **b.cpp** 中在所有定义的函数之前都添加前缀 **module_b_**，在 **a.cpp** 中所有定义的函数之前都添加前缀 **module_a_**。如果我们在 **a.cpp** 中希望引用的是 **b.cpp** 中的 **add**，那么就调用 **module_b_add**，否则调用 **module_a_add**，这就非常简单地解决了问题。

但这种方式继续带来了两个问题。

第一个问题是代码冗长，尤其是在 `b.cpp` 中定义的时候，所有函数都需要添加前缀 `module_b_`，会让所有的函数定义非常复杂。



第二个问题是，如果前缀也重复怎么办？毕竟两个编译单元的编写者是不知道对方使用什么前缀的，在技术上无法避免，只能通过不同编译单元的编写者提前约定好双方的前缀来解决。真是一个问题接着一个问题.....

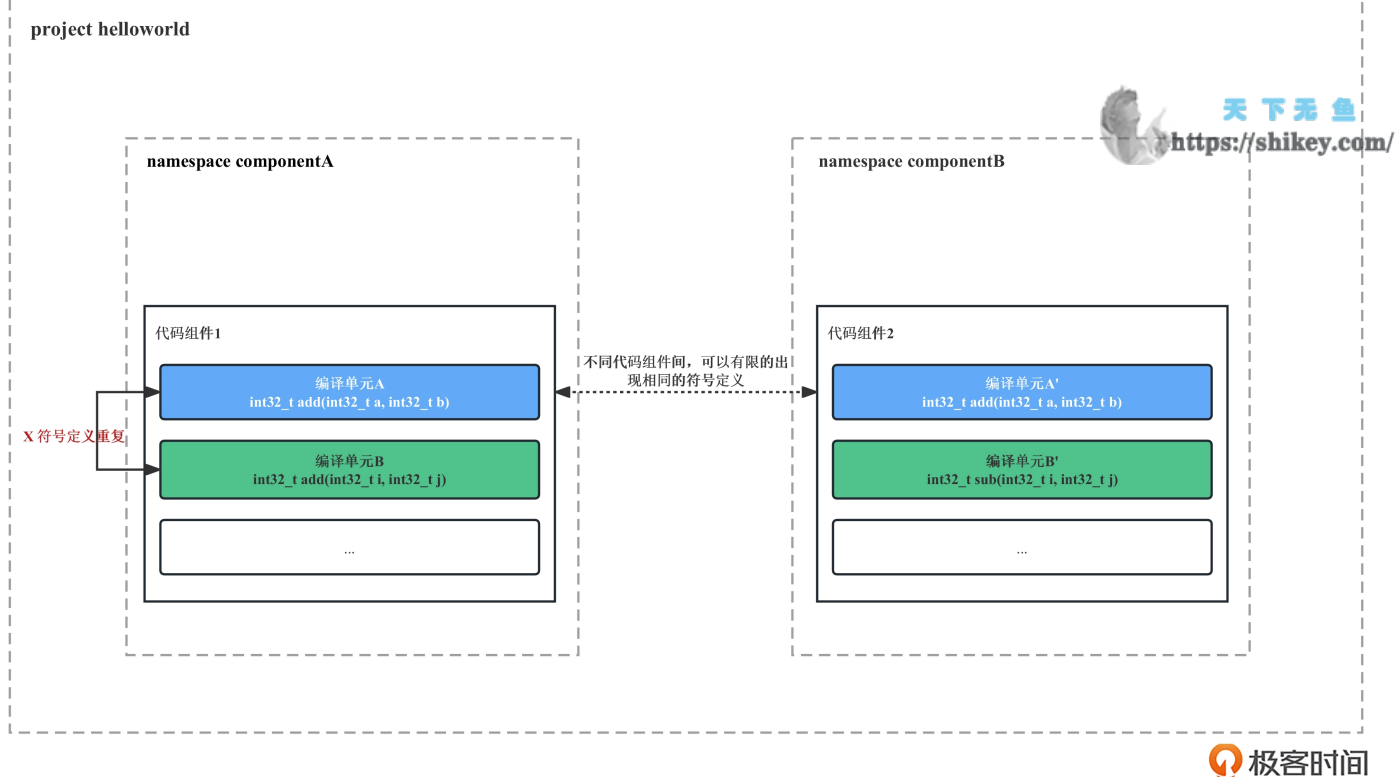
不过，在 `C++` 中符号隔离问题，也可以通过隐藏一个编译单元中的私有符号来解决，也就是不让其他编译单元“看到”这些符号。

这也能大量减少编译单元之间的符号冲突问题，毕竟可能出现，两个编译单元定义了同名，但只想在编译单元内部使用函数的情况，我们并不想给这些函数加上冗长的前缀。那这个时候，只需要使用 `static` 修饰符。比如我们可以在 `A` 和 `B` 中都定义 `static` 函数 `to_int`，然后再编译链接，这样就不会出现符号冲突的问题。

命名空间

虽然在不同编译单元中，相同的符号会引发链接错误，不过在不同的代码组件中，我们是完全可以定义相同符号的，这其中可以包含符号常量、变量、函数、结构体、类、模板和对象等等。

但是，相同的符号并不意味着它们有相同的功能，而且随着 `C++` 工程越来越大，导入的库变多，这种命名冲突的可能性就越大。下图展示了不同编译单元、不同代码组件、`namespace` 之间的关系和层次。



为了避免在 C++ 编程过程中避免命名冲突，C++ 标准提供了关键字命名空间（namespace）的支持，可以更好地控制符号作用域。但通过 namespace 进行符号隔离，仍然存在局限性。

namespace 可以通过命名空间避免符号名称冲突，但本身并不管理符号的可见性问题，不同 namespace 之间的符号可见性取决于编译单元的符号可见性，这让符号的管理变得非常复杂。另外不同编译器产生的 namespace 的符号修饰方式不同（毕竟这不是 C++ 标准定义的内容），也就是 ABI 层面不同，会导致跨编译器的符号引用出现很大的问题。

总结

不知道你有没有发现，今天的思考其实是围绕一个核心问题展开的：为了隐藏代码实现细节，我们往往要使用哪些编程范式或技巧？

方法一：通过 include 头文件来统一声明符号

这种方法的优点是简单粗暴，简洁明了，一定程度上解决了同一组件内相同符号定义冲突的问题。但这种方法也有硬伤，它利用了头文件代码这种非常“低级”的技术方式实现，而且仍无法避免两个编译单元的重复符号的问题。

方法二：添加前缀来避免符号冲突

这种方法简单明了，可操作性强，基本可以解决符号冲突问题。但是会大幅降低代码可读性，还会让符号声明变得更长，而且当前缀也重复了，我们真的很难解决问题，难道全局替换前缀？因此这种方式仍然不能解决符号重复和可见性的问题。



方法三：通过 namespace 避免符号冲突

*namespace 可以通过命名空间避免符号名称冲突。但是，它本身并不管理符号的可见性问题，不同 namespace 之间的符号可见性取决于编译单元的符号可见性，这让符号的管理变得非常复杂。而且 namespace 在不同编译器上的表现不同，在 ABI 层面无法实现兼容。

我们可以看出，include 头文件机制没能跟上现代 C++ 标准的演进，提供一套行之有效的进行代码组合、符号和功能复用的方案。那么，有没有什么办法来保证库的独立性、易用性，同时提高代码编译速度呢？


下一讲，Modules 在现代 C++ 中就要粉墨登场了，敬请期待。

课后思考

今天，我们研究了 include 头文件机制以及 C++20 前解决符号可见性问题的一些最佳实践。那么在 C++ Modules 出现以前，你是怎么解决符号隔离问题的？

欢迎把你碰到的情况与解决方法，与大家一起分享。我们一同交流。下一讲见！

分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享

 赞 2  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 开篇词 | 为什么掌握现代C++新特性如此重要？

下一篇 02 | Modules（中）：解决编译性能和符号隔离的银弹

精选留言 (7)



tanatang

2023-01-18 来自四川

写面向对象的C++，成员和函数都在设计在类中，禁止使用这种不属于任何类的全局函数，全局变量。

作者回复: OOP原教旨主义



wilby

2023-01-18 来自瑞典

为什么namespace让符号管理变得很复杂？老师好像就给了个结论，能展开讲讲么？

作者回复: 文中的意思并非说namespace导致符号管理变复杂。这里的意思是，诸如namespace这类语言特性可以解决符号冲突问题，并且可以在编译时给予友好的提示，但由于C++的编译链接分离机制，C++只为链接定义了一些指导性原则，链接中很多具体技术规范大多由ABI来定义，这就导致在不同编译器和不同体系结构下可能在链接时会遇到不同的问题，这才是C++符号管理复杂性的核心问题所在。而C++ Modules则是更完备的符号可见性管理语言特性，可以与namespace实现互补，而且在链接上也给出了更多的原则（虽然还是无法完全解决ABI问题），让符号管理更加易于理解，更接近现代编程语言，可以一定程度降低整体符号管理的复杂性。



小样

2023-01-24 来自江西

太可怕了，以前没有想过这种问题。

实际用到的代码规模并不是很大，一个人就能完全掌握各个模块，也就不会有冲突。现在来看传统上符号隔离和污染问题根本就没有在设计上解决过。软件工程规模大后必然有冲突问题。

作者回复: 是的，所以软件工程需要确定种种规范，就是为了尽量规避这些问题，但在实际的大项目中还是很难完全避免，尤其是要集成大量第三方库的时候会更加头痛。
经验积累很重要，而且很值钱。记得总结出自己的一套实践方法。



不二

2023-01-19 来自浙江

历史挺好的，但是里面概念太多，解释不清楚感觉消化不了。

作者回复: 掌握C++的过去，才能更好的理解C++的未来。虽然这是C++编程语言复杂性的一个缩影，但是有助于大家理解现代C++。如果对这些旧有概念感兴趣，可以回顾一下这些概念。有任何问题也欢迎在评论区提出，我会跟你一起探讨。



Geek_7c0961

2023-01-18 来自美国

"这也能大量减少编译单元之间的符号冲突问题，毕竟可能出现，两个编译单元定义了同名，但只想在编译单元内部使用函数的情况，我们并不想给这些函数加上冗长的前缀。那这个时候，只需要使用 **static** 修饰符。比如我们可以在 **A** 和 **B** 中都定义 **static** 函数 **to_int**，然后再编译链接，这样就不会出现符号冲突的问题。" 这块儿能否给个具体的代码示例？

作者回复: 比如以下代码

a.cpp:

```
#include <iostream>
```

```
static void print() {  
    std::cout << "Print in A" << std::endl;  
}
```

```
void fa() {  
    print();  
}
```

b.cpp:

```
#include <iostream>
```

```
static void print() {  
    std::cout << "Print in B" << std::endl;  
}
```

```
void fb() {  
    print();  
}
```

main.cpp

```
extern void fa();
```

```
extern void fa();
```

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



天下无鱼

<https://shikey.com/>

这里a.cpp和b.cpp两个编译单元都有print函数，但是函数使用了static修饰符，因此print函数仅对各自编译单元内部可见，所以链接时不会导致符号冲突问题。会正确输出：

Print in A

Print in B

共 2 条评论 >



tang_ming_wu

2023-01-17 来自广东

链接过程，函数地址填充，是不是都是相对地址？

作者回复: 不一定，C++并没有定义二进制函数地址填充的方式，这完全取决于操作系统与特定体系结构的ABI，有可能是相对地址，也有可能是绝对地址。



peter

2023-01-17 来自北京

请教两个问题：

Q1: 第一个专栏是什么？

Q2: 实现放在头文件中为什么可以提高运行性能？

文中有这样一句话“但是为了提高运行时性能，开发者也会考虑将实现直接放在头文件中。”为什么？

作者回复: A1: 《动态规划面试宝典》

A2: 因为C++编译器支持针对函数调用的内联优化。

将函数实现放到头文件中，编译期可以根据实际情况对函数调用进行优化，比如将函数调用替换成函数的实现代码，这样针对部分较短的函数可以抵消掉函数调用的性能损耗，这也就是为何明确标为inline的函数定义都一定要放在头文件中。

共 2 条评论 >



