C 与 C++的相互调用方法

https://blog.csdn.net/qq_43899283/article/details/132343699

C 与 C++为什么相互调用的方式不同

C 和 C++ 之间的相互调用方式存在区别，主要是由于 C 和 C++ 语言本身的设计和特性不同。

函数调用和参数传递方式不同：C 和 C++ 在函数调用和参数传递方面有一些不同之处。C 使用标准的函数调用约定，而 C++ 在函数调用中可能包含额外的信息，如函数重载和默认参数。为了正确匹配函数签名，C++ 编译器可能会在函数名上进行名称修饰（name mangling）。

函数重载和名称修饰：C++ 支持函数重载，即可以有相同的函数名但不同的参数列表。为了在可执行文件中区分这些重载函数，C++ 编译器会在函数名中添加一些信息，以便于重载解析。这与 C 的函数名约定不同，C 中函数名是平铺的。

链接库的差异：C 和 C++ 编译器链接不同的标准库。C 编译器链接 C 标准库，而 C++ 编译器链接 C++ 标准库。由于标准库可能涉及不同的函数和数据结构，因此在链接阶段可能会有不同的处理。

编译器特性：C 和 C++ 编译器对代码的解析、优化、链接等可能会有不同的处理方式，这可能会导致在 C 和 C++ 相互调用时需要进行适当的处理。

解决手段：为了在 C 和 C++ 之间实现相互调用，C++ 引入了 extern “C” 语法，它可以用来告诉 C++ 编译器在函数声明上使用 C 的调用约定，以便在链接阶段能够正确解析函数名。这种设计是为了在 C 和 C++ 之间实现互操作性，但由于两者的语法和特性存在差异，因此在调用方式、编译器行为和链接方式上会存在一些差异。

C++中调用 C

话不多说，直接上案例，下面是一个简单的示例，演示了如何在 C++ 代码中调用 C 函数：

首先分别创建三个文件：mylib.c、mylib.h 和 main.cpp

mylib.c 如下：

```
// mylib.c
```

```c
#include <stdio.h>

void my_c_function() {
    printf("This is a C function.\n");
}
```

mylib.h 如下：

```c
// mylib.h
#ifndef MYLIB_H
#define MYLIB_H

void my_c_function();

#endif // MYLIB_H
```

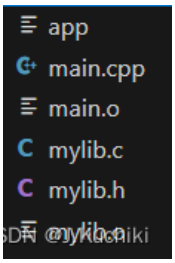
main.cpp 如下：

```c
// main.cpp
#include <iostream>

extern "C" {
    // 声明 C 函数的原型
    void my_c_function();
}

int main() {
    std::cout << "Calling a C function from C++:" << std::endl;

    // 调用 C 函数
    my_c_function();

    return 0;
}
```

在这个示例中，我们使用了 #include "mylib.h" 来引入头文件，并在 C++ 中调用了 my_c_function()。这样就能正确地在 C++ 中调用 C 函数。编译步骤如下：

```
gcc -c mylib.c -o mylib.o       # 编译 C 文件为目标文件
g++ -c main.cpp -o main.o       # 编译 C++ 文件为目标文件
g++ main.o mylib.o -o app       # 链接目标文件生成可执行文件
```

编译后的文件列表如下：



然后运行可执行文件：

./app 得到输出结果：

这里可以使用 objdump 命令查看编译之后的中间文件 mylib.o 和 main.o 的符号表：

```
[kuchiki@localhost test]$ objdump -t mylib.o

mylib.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 mylib.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .rodata        0000000000000000 .rodata
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     F .text  0000000000000011 my_c_function
0000000000000000         *UND*  0000000000000000 puts
```


可以发现，my_c_function()函数编译出的名称在 mylib.o 和 main.o 是相同。这是由于 C++ 文件中使用 extern “C” 来声明 C 调用约定，以便 C 能够正确解析函数名。

我们来看看如果没有使用 extern “C” 后的编译情况吧：

```
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 main.cpp
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .rodata        0000000000000000 .rodata
0000000000000000 l     0 .rodata        0000000000000001 _ZStL19piecewise_construct
0000000000000000 l     0 .bss   0000000000000001 _ZStL8__ioinit
000000000000002c l     F .text  000000000000003e _Z41__static_initialization_and_destruction_0ii
000000000000006a l     F .text  0000000000000015 _GLOBAL__sub_I_main
0000000000000000 l    d  .init_array    0000000000000000 .init_array
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     F .text  000000000000002c main
0000000000000000         *UND*  0000000000000000 _ZSt4cout
0000000000000000         *UND*  0000000000000000 _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
0000000000000000         *UND*  0000000000000000 _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
0000000000000000         *UND*  0000000000000000 _ZNSolsEPFRSoS_E
0000000000000000         *UND*  0000000000000000 _Z13my_c_functionv
0000000000000000         *UND*  0000000000000000 _ZNSt8ios_base4InitC1Ev
0000000000000000         *UND*  0000000000000000 .hidden __dso_handle
0000000000000000         *UND*  0000000000000000 _ZNSt8ios_base4InitD1Ev
0000000000000000         *UND*  0000000000000000 __cxa_atexit
```


可以发现，不使用 extern “C”， 函数 my_c_function 编译后名称变为了(_Z13my_c_functionv） 。

是由于在 C++中，函数名在编译后会根据函数的参数类型和返回类型进行名称重整（Name Mangling），以支持函数重载等特性。这是因为 C++支持函数的参数类型和个数可以不同，所以需要在编译后为每个函数生成一个唯一的名称。

当你在 C++中调用一个 C 函数时，如果不使用 extern “C” 声明，C++ 编译器会默认对函数名进行名称重整。而在 C 语言中，函数名不会被重整。

如果你在 C++中调用了一个 C 函数，并且没有使用 extern “C” 声明，C++ 编译器会对函数名进行名称重整，生成一个新的名字，类似 _Z13my_c_functionv 这样的名称。这个过程被称为名称重整（Name Mangling），是为了确保函数在 C++中能够正确处理函数重载等特性。

C 中调用 C++

下面还是来看一个简单的示例，演示了如何在 C 代码中调用 C++ 函数：

首先分别创建三个文件：mylib.cpp、mylib.h 和 main.c

mylib.cpp 如下：

```
// mylib.cpp
#include <iostream>

#include "mylib.h"

void my_cpp_function(int num) {
    std::cout << "C++ function called with number: " << num << std::endl;
}
```

mylib.h 如下：

```
// mylib.h
#ifndef MYLIB_H
#define MYLIB_H

#ifdef __cplusplus
extern "C" {
#endif

void my_cpp_function(int num);

#ifdef __cplusplus
}
#endif
```

```
#endif // MYLIB_H

#endif // MYLIB_H
```

main.c 如下：

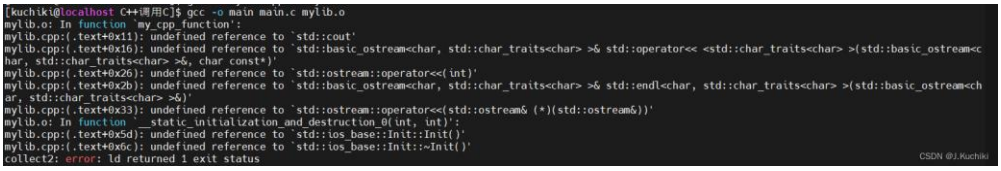
```
// c_main.c
#include <stdio.h>

#include "mylib.h"

int main() {
    printf("Calling C++ function from C\n");

    // Call the C++ function
    my_cpp_function(42);

    return 0;
}
```

在这个示例中，我们使用了 #include "mylib.h" 来引入头文件，并在 main.c 中调用了 my_cpp_function()。这样就能正确地在 C 中调用 C++ 函数。编译步骤如下：

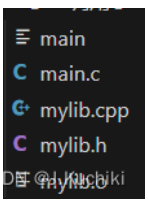
```
g++ -c mylib.cpp -o mylib.o     # 编译 C 文件为目标文件
gcc -o main main.c mylib.o -lstdc++    # 链接目标文件生成可执行文件
```

注释：-lstdc++ 是用于链接 C++ 标准库的编译选项。在 Linux 系统中，C++ 标准库通常被命名为 libstdc++.so，使用 -lstdc++ 编译选项可以将这个库链接到可执行文件中，以便在运行时使用 C++的标准库函数和功能。

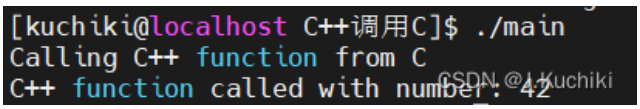
如果缺少 -lstdc++ 则会报错：



编译后的文件列表如下：



然后运行可执行文件：./main 得到输出结果：



这里解释一下 mylib.h 头文件中的 #ifdef __cplusplus：在 main.c 文件夹中调用 mylib.h 头文件，但是 C 语言中并没有 extern 这个关键字，因此，使用 #ifdef __cplusplus 来充当一个译时候的阀门。

总结

对于 C 调用 C++的情况，没有 extern “C” 这样的关键字。您需要在 C++代码中使用 extern “C” 来确保 C++函数按照 C 的方式进行链接，同时在 C 代码中包含相应的头文件并调用这些函数。