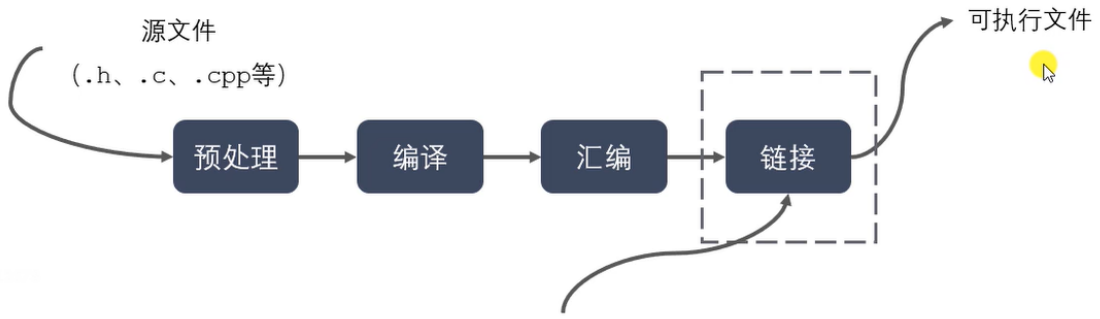
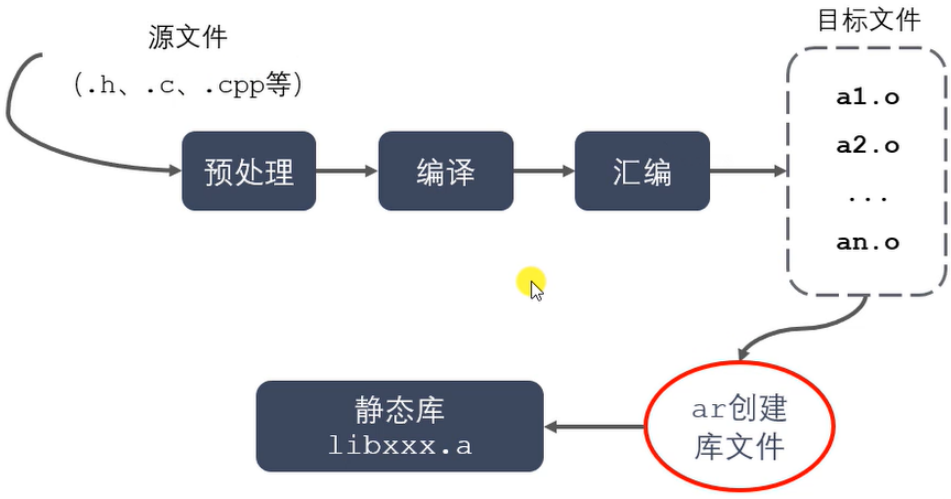
程序编译成可执行程序的过程



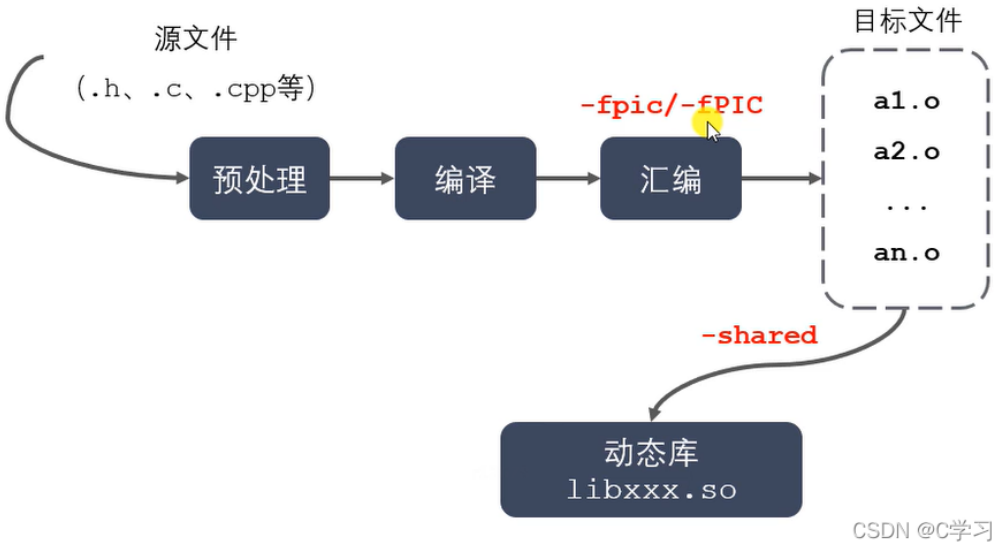
静态库、动态库区别来自链接阶段如何处理，链接成可执行程序。分别称为静态链接方式和动态链接方式。

静态库：在原程序链接时，会把静态库的代码合并到源程序的可执行文件中。
动态库：在原程序链接时，不会把动态库的代码合并源程序的可执行文件中，而只是添加了动态库的一些信息到其中。

静态库的制作过程



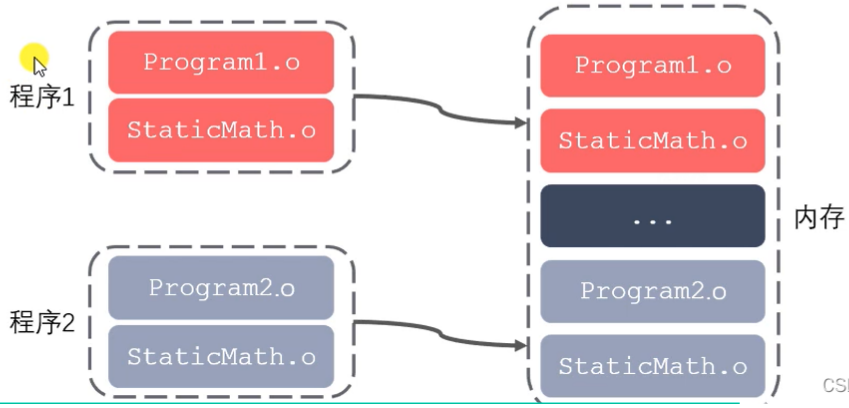
动态库的制作过程



-fpic / -fPIC是指生成位置无关的代码，因为动态库是动态的可以被多个程序共同加载的，所以无法和静态库一样将代码嵌入到可执行程序中。

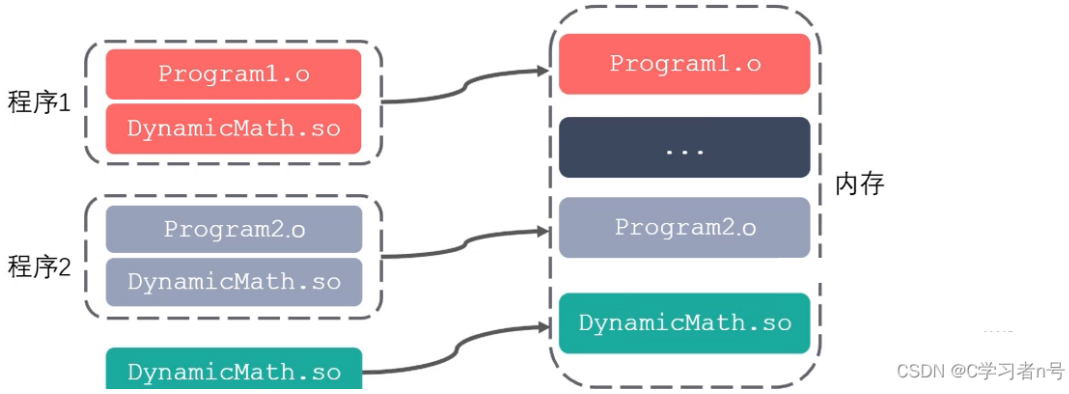
静态库的优缺点

- 优点：
  - 静态库被打包到应用程序中加载速度快
  - 发布程序无需提供静态库，移植方便
- 缺点：
  - 消耗系统资源，浪费内存
  - 更新、部署、发布麻烦



动态库的优缺点

- 优点：
  - 可以实现进程间资源共享（共享库）
  - 更新、部署、发布简单
  - 可以控制何时加载动态库
- 缺点：
  - 加载速度比静态库慢
  - 发布程序时需要提供依赖的动态库



进阶：动态加载库

动态加载库和动态链接库不同，动态链接库在程序启动的时候就要寻找动态库，找到库函数；而动态加载库可以用程序的方法来控制什么时候加载。动态加载库主要有函数dlopen、dlerror()、dlsym()和dlclose()。

```
#include <dlfcn.h>

1、打开动态库dlopen函数
该函数按照用户指定的方式打开动态链接库，其中参数filename为动态链接库的文件名，flag为打开方式，一般为RTLD_LAZY，函数的返回值为库的指针。其函数原型如下：
void * dlopen(const char * filename, int flag);

2、获取函数指针dlsym()
使用动态链接库的目的时调用其中的函数，完成特定的功能。函数dlsym可以获取动态连接库中特定的函数的指针，然后可以使用这个函数指针进行操作。
void * dlsym(void * handle, char * symbol);
其中参数handle为dlopen函数打开动态库后返回的句柄，参数symbol为函数的名称，返回值为函数指针。

3、获取错误信息
dlerror（）函数返回一个可读的、以null结尾的字符串，该字符串描述自上次调用dlerr（）以来，调用dlopen API中的一个函数时发生的最新错误。返回的字符串不包括尾随的新行。如果没有错误，返回NULL
char *dlerror(void);

4、关闭动态加载库
int dlclose(void *handle);
```

1、用纯C代码使用动态加载库

```
//1、创建动态库，这里写一个简单的

//computer.c
#include <stdio.h>
#include <stdlib.h>

int sum(int i, int j) {
    return (i + j) * 100;
```

```
}

//2、生成libcomputer.so动态库
gcc -c -fpic computer.c   //生成与位置无关的.o文件
gcc -shared computer.o -o libcomputer.so    //生成动态库文件so


//3、使用动态加载库（lnk.c）
#include <stdio.h>
#include <dlfcn.h>

int main(void)
{
    void *pHandle = dlopen("./libcomputer.so", RTLD_LAZY);  //open so

    if(!pHandle)
    {
        printf("Failed Load libary!\n");
    }
    char *perr = dlerror(); //error value
    if(perr != NULL) {
        printf("%s\n", perr);
        return 0;
    }

    int (*psum)(int , int) = dlsym(pHandle, "sum");

    int r = psum(1, 2);
    printf("r = %d\n", r);

    dlclose(pHandle);
    return 0;
}

//4、生成可执行程序
gcc -o app lnk.c libcomputer.so -ldl    //生成app可执行文件
```

2、用C++封装使用动态加载库

```
//1、创建动态加载（cplus.cpp）

#include <iostream>

class Test {
public:
  Test() {
    std::cout << "Test object created" << std::endl;
  }
  ~Test() {
    std::cout << "Test object destroyed" << std::endl;
  }
  void hello() {
    std::cout << "Hello, World!" << std::endl;
  }
};

//通过这两个函数实现类对象的获取和释放
```

```
extern "C" {
  Test* create_test_object() {
    return new Test();
  }

  void destroy_test_object(Test* obj) {
    delete obj;
  }
}


//2、生成动态库（这里一步生成）
g++ -shared -fPIC -o libtest.so cplus.cpp

//3、使用动态加载库cplus_lnk.cpp
// main.cpp
#include <cstdlib>
#include <cstdio>
#include <dlfcn.h>
#include "cplus.cpp"

int main() {
  void *handle = dlopen("./libcplus.so", RTLD_LAZY);
  if (!handle) {
    fprintf(stderr, "%s\n", dlerror());
    return 1;
  }

  // 使用dlsym函数获取共享库中的符号地址
  void* (*create_test_object)() = (void*(*)())dlsym(handle,
"create_test_object");
  if (!create_test_object) {
    fprintf(stderr, "%s\n", dlerror());
    return 1;
  }

  // 调用共享库中的函数创建类对象
  void* obj_ptr = create_test_object();
  if (!obj_ptr) {
    fprintf(stderr, "Failed to create test object\n");
    return 1;
  }

  // 将void*类型的指针转换为Test*类型的指针
  typedef void (*deleter)(void*);
  Test* obj = reinterpret_cast<Test*>(obj_ptr);

  // 调用类对象的方法
  obj->hello();

  // 使用dlsym函数获取共享库中的符号地址
  deleter destroy_test_object = (deleter)dlsym(handle, "destroy_test_object");
  if (!destroy_test_object) {
    fprintf(stderr, "%s\n", dlerror());
    return 1;
  }

  // 调用共享库中的函数销毁类对象
```

```
  destroy_test_object(obj_ptr);

  // 关闭共享库
  dlclose(handle);

  return 0;
}

//4、生成可执行程序app
 g++ -o app cplus_lnk.cpp libcplus.so -ldl
```