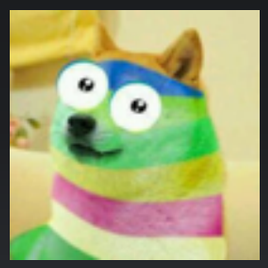
把玩链接器 - Linker 与 Loader 的前世今生

孙源 sunnyxx iOS Developer | 滴滴出行



Weibo · @我就叫Sunny怎么了
Blog · http://blog.sunnyxx.com
GitHub · github.com/forkingdog

从头讲起

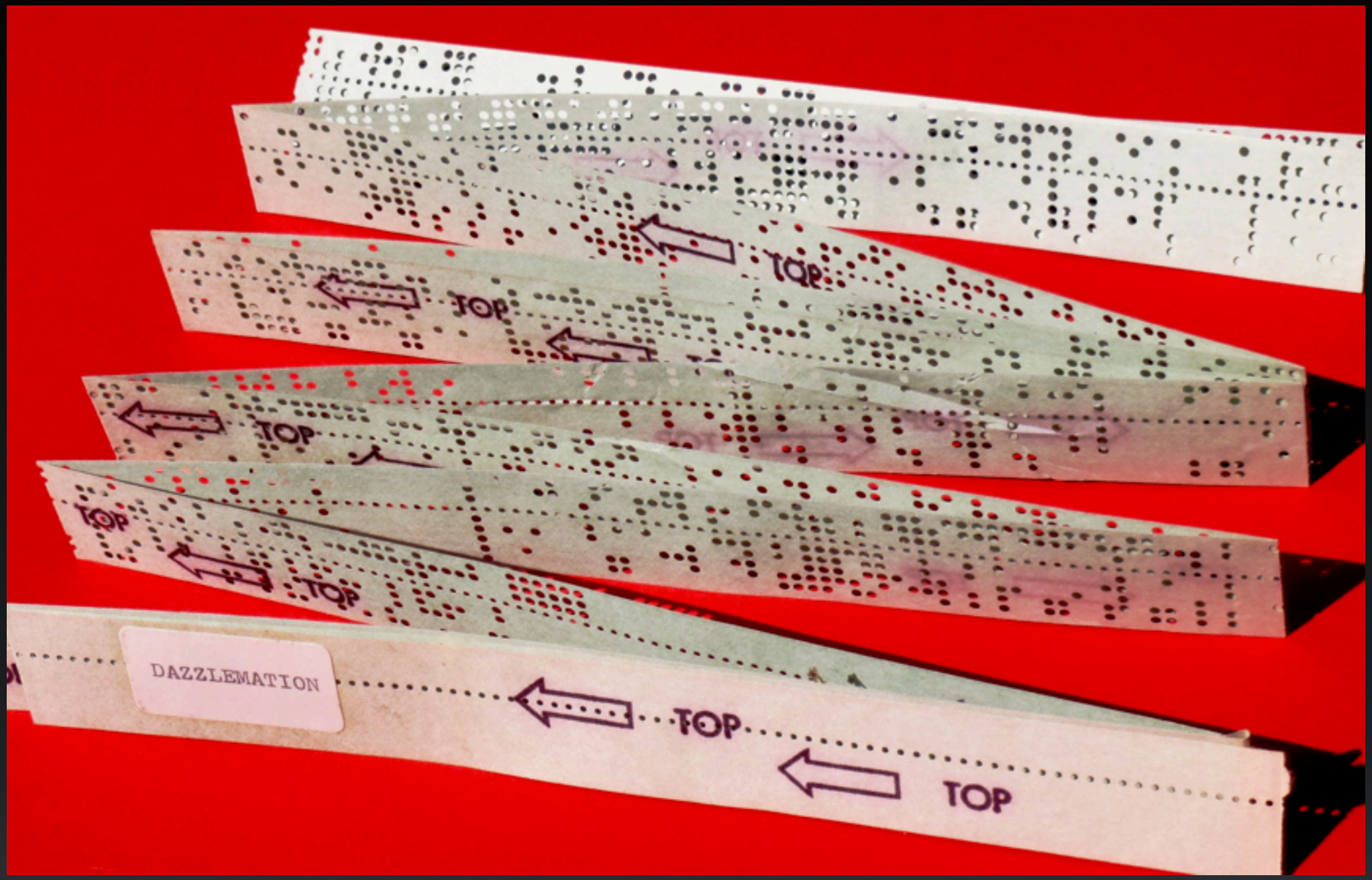
PDP 11 ASCII CHARACTER TEST

MACHINE CODE

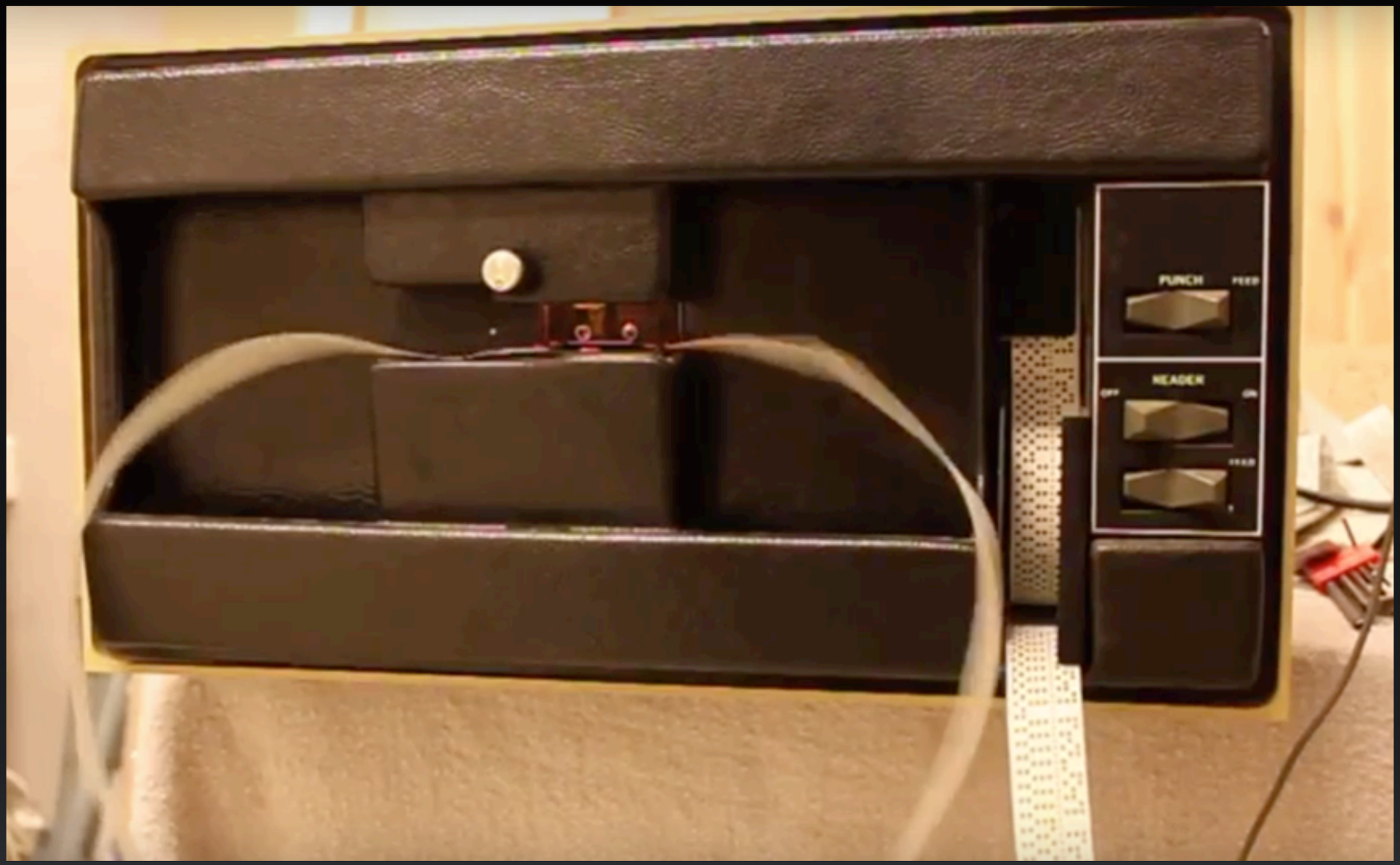
ADD	DATA	COMMENT	
2000	012700	MOV #40, R0	Move Zahl 40 nach R0
2002	40		
2004	105737	TSTB @ # 777564	Kontrolliere Konsole CSR ob OK
2006	777564		
2010	100375	BPL -3	(Branch if plus)
2012	010037	MOV R0, @#777566	Move Inhalt von R0 nach Adresse der Konsole Output
2014	777566		
2016	5200	INC R0	erhöhe R0
2020	020027	CMP R0, #177	(Compare)
2022	177		
2024	001367	BNE → 2004	(Branch not equal)
2026	000137	JMP 2000	
2030	2000		

PDP-11 Machine Code Sample

PDP-11 / 40 1970s



打孔纸带



打孔纸带读取器

将纸带中的数据加载到内存中

最早的 Loader

但程序员们很快就发现了问题

PDP 11 ASCII CHARACTER TEST

MACHINE CODE

ADD	DATA	COMMENT	
2000	012700	MOV #40, R0	Move Zahl 40 nach R0
2002	40		
2004	105737	TSTB @ # 777564	Kontrolliere Konsole CSR ob OK
2006	777564		
2010	100375	BPL -3	(Branch if plus)
2012	010037	MOV R0, @#777566	Move Inhalt von R0 nach Adresse der Konsole Output
2014	777566		
2016	5200	INC R0	erhöhe R0
2020	020027	CMP R0, #177	(Compare)
2022	177		
2024	001367	BNE → 2004	(Branch not equal)
2026	000137	JMP 2000	
2030	2000		

JMP 2000

PDP-11 Machine Code Sample

手写机器码时的问题

- 程序员要为每行指令，在写代码时就要设好**实际内存地址**
- 若代码有改动，需要把所有代码的地址检查一遍，比如 JMP 2000

问题本质

代码与实际内存地址绑定过早

程序中没有什么是加一个中间层不能解决的。如果有，那就两层。

- 尼古拉斯 · 赵四

汇编出现

将指令 符号化

将地址 符号化

最早 Assembler 的作用

- 使程序员摆脱**机器码的人肉翻译工作**
- 使程序员摆脱**地址的人肉绑定工作（ Linker 的雏形 ）**

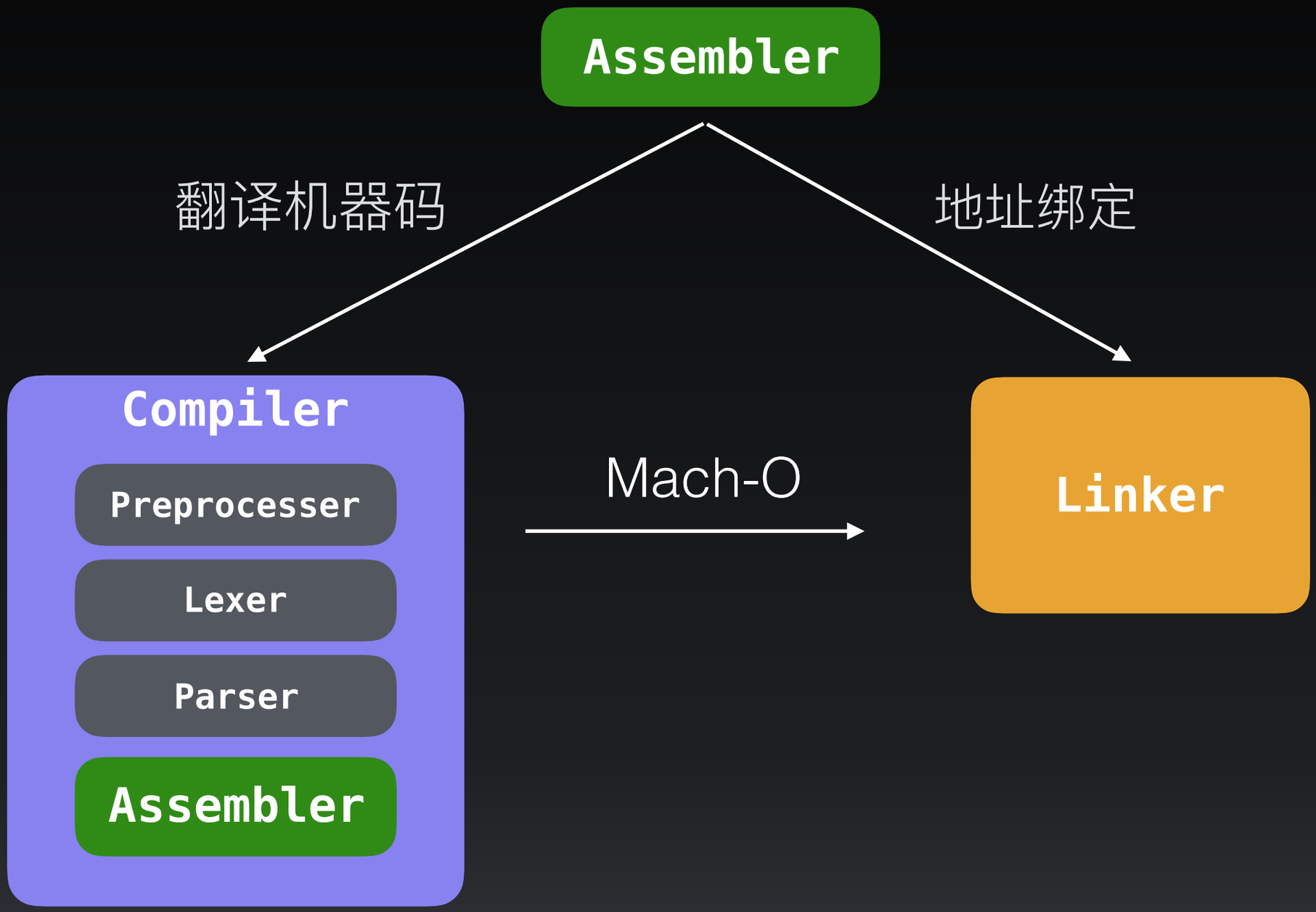

後來……
劉若英

后来

- **更复杂的源码结构**：谁说一个程序只能一个文件了？
- **Library**：我们不甘心写重复代码！
- **高级语言**：还要写汇编？你是原始人？

Assembler?

Assembler 拆分职责



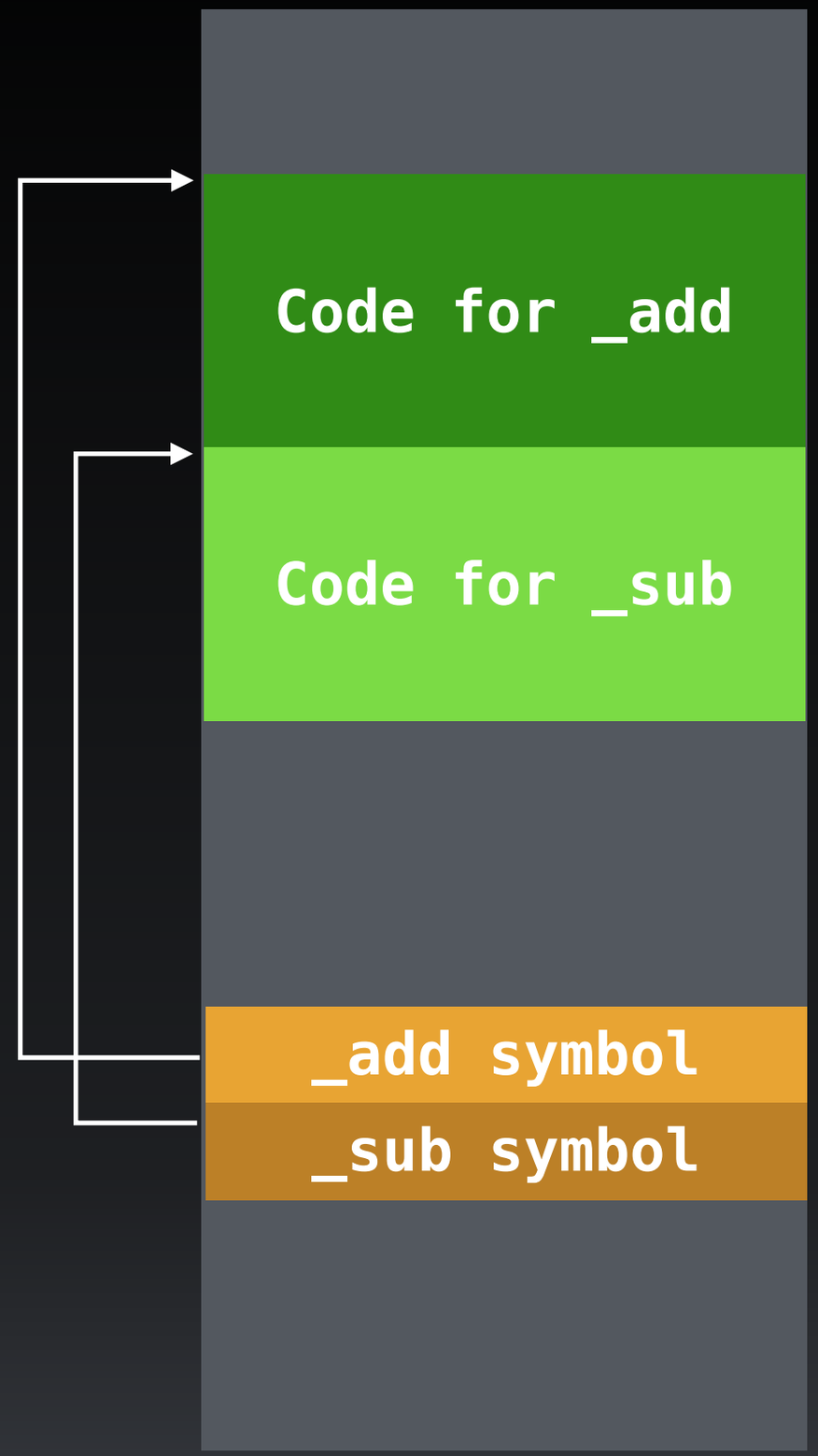
Compiler + Linker 是如何配合工作的?

```c
// my_math.c

int add(int a, int b) {
    return a + b;
}

int sub(int a, int b) {
    return a - b;
}
```

```c
// main.c

int main() {
    int ret = add(1, 2);
    return 0;
}
```

编译 my_math.c

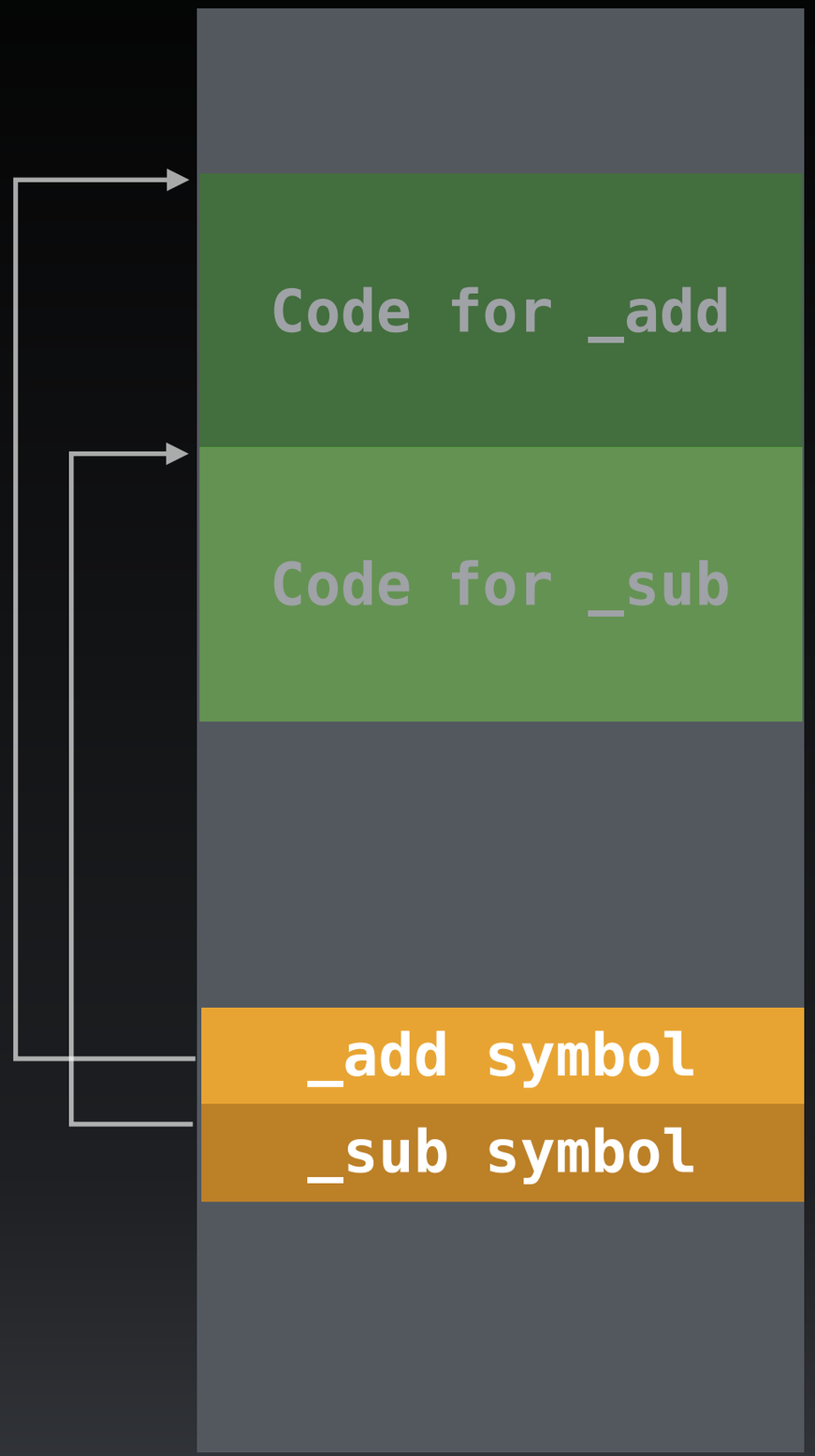
```c
// clang -c my_math.c -o
// my_math.o

int add(int a, int b) {
    return a + b;
}

int sub(int a, int b) {
    return a - b;
}
```



Symbol Table

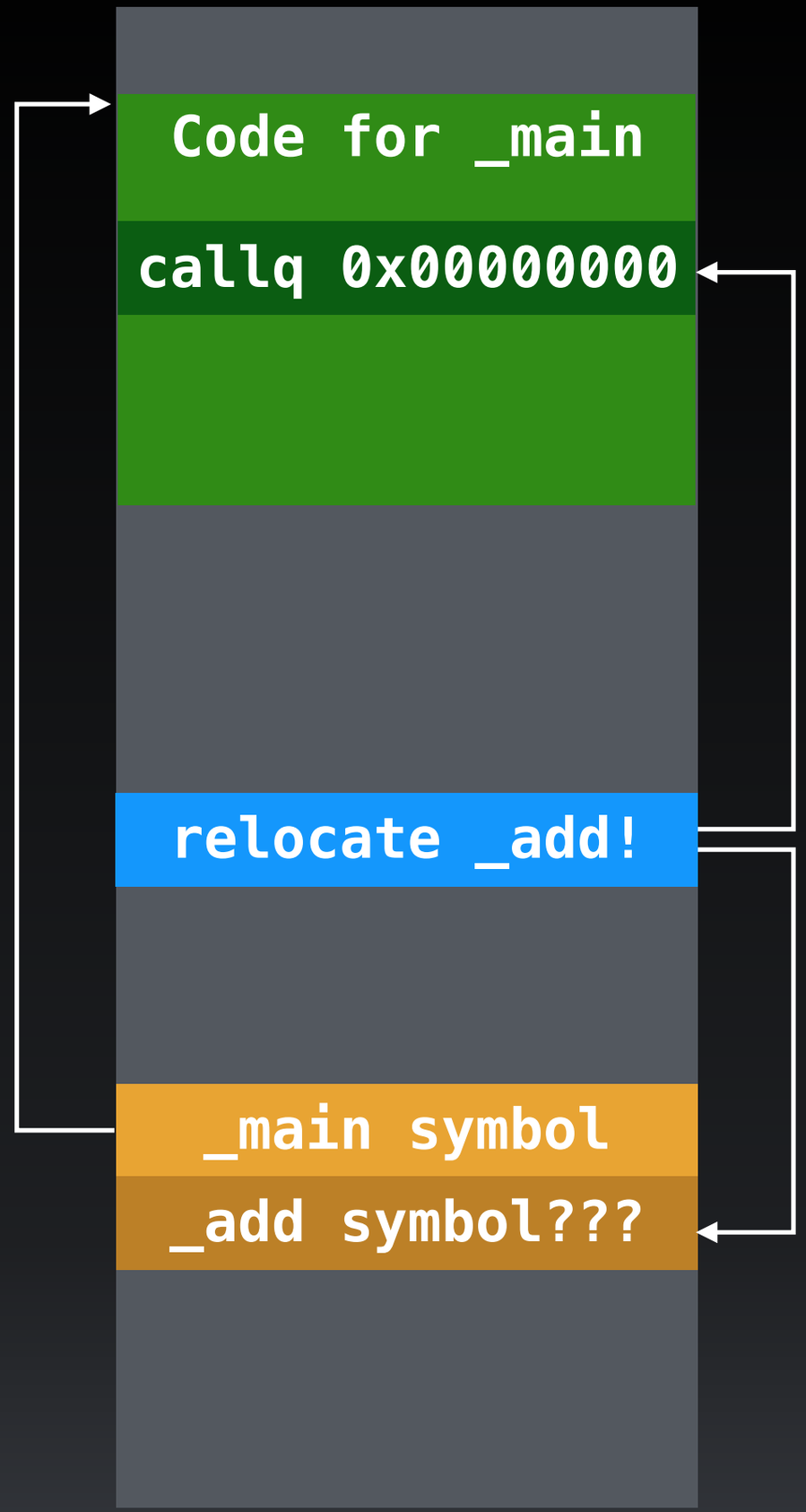
记录当前文件中：

- 所有**已定义**的符号
- 所有**未定义**的符号

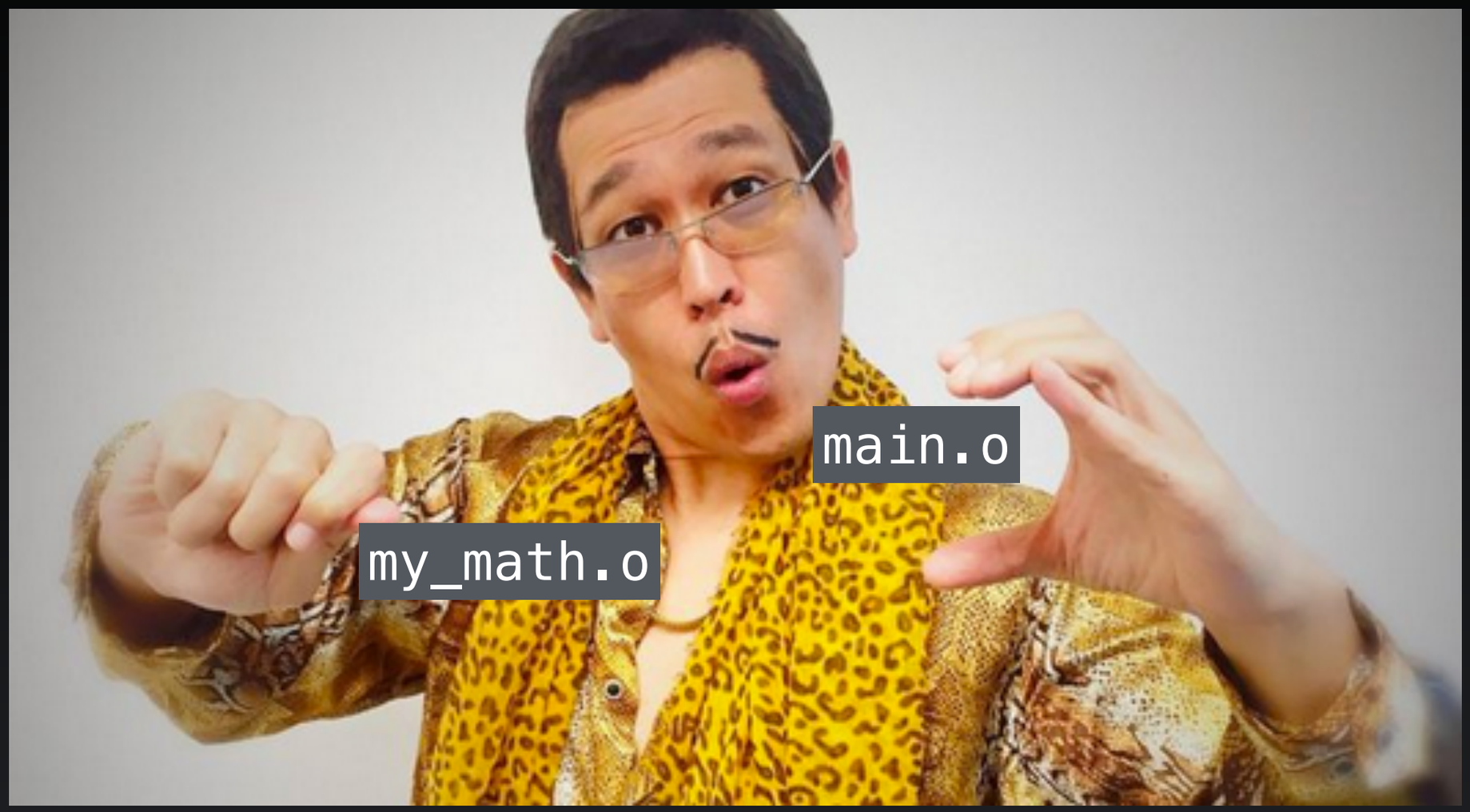


编译 main.c

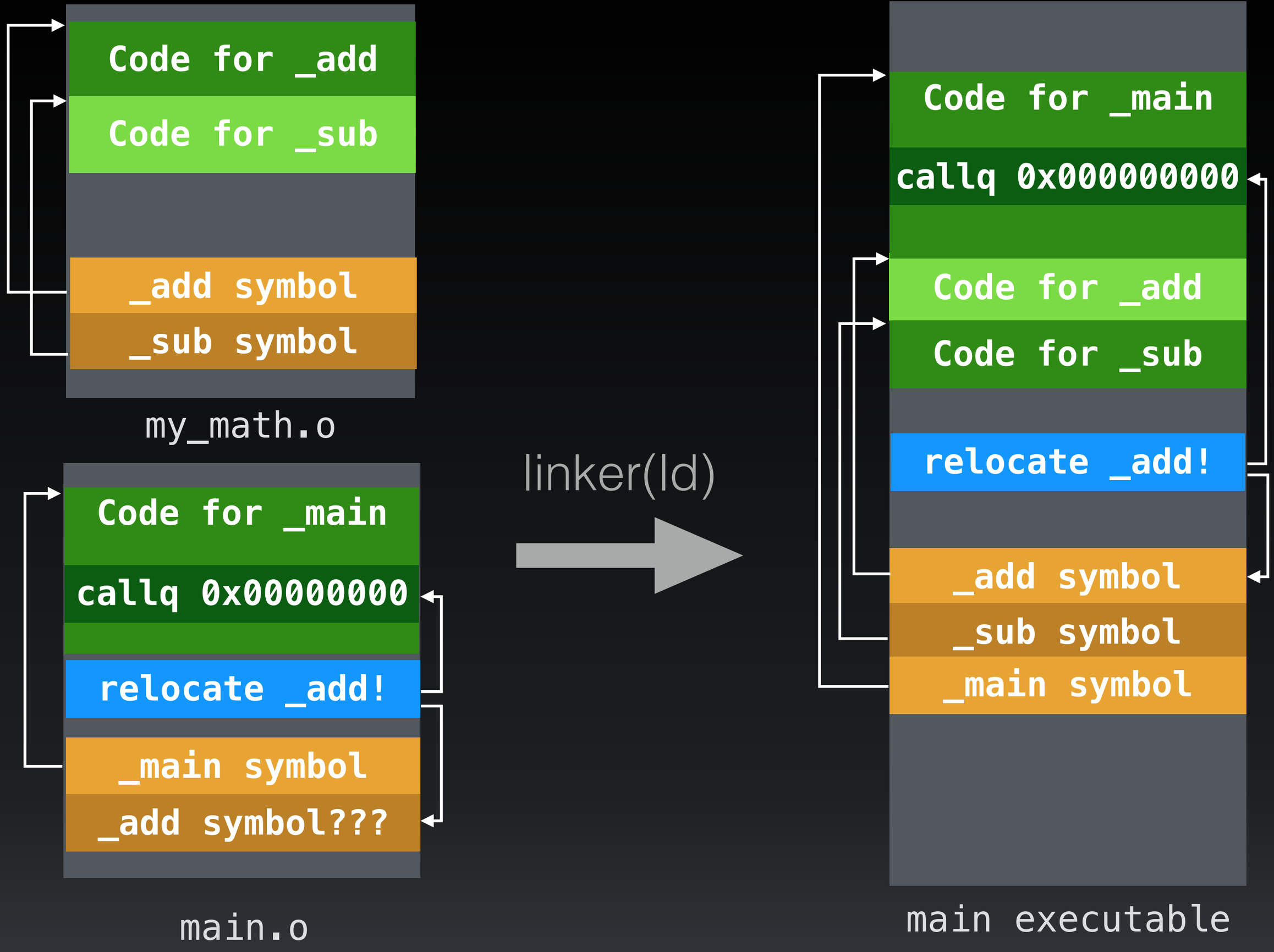
```
// clang -c main.c -o
// main.o

int main() {
    int ret = add(1, 2);
    return 0;
}
```

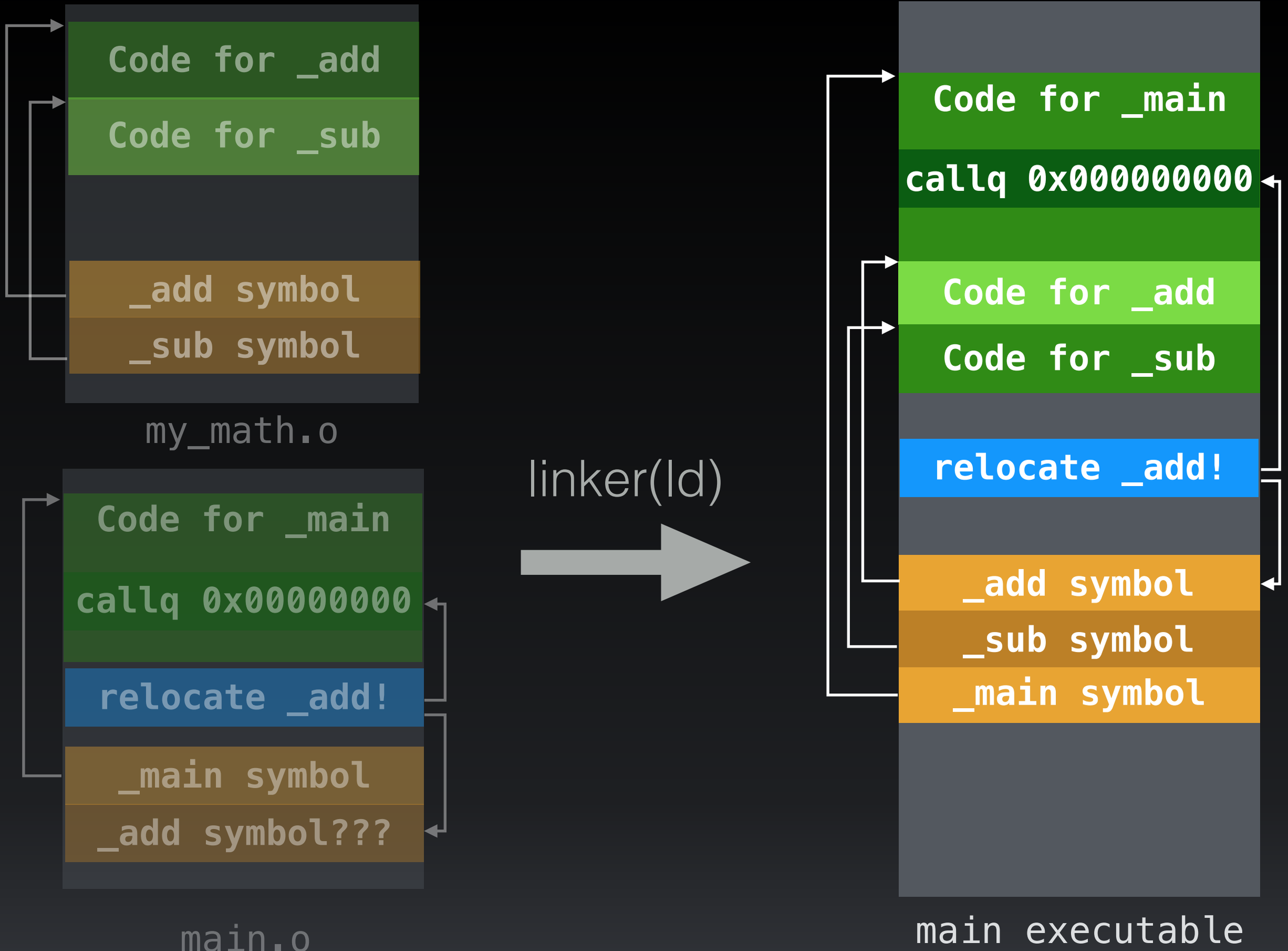


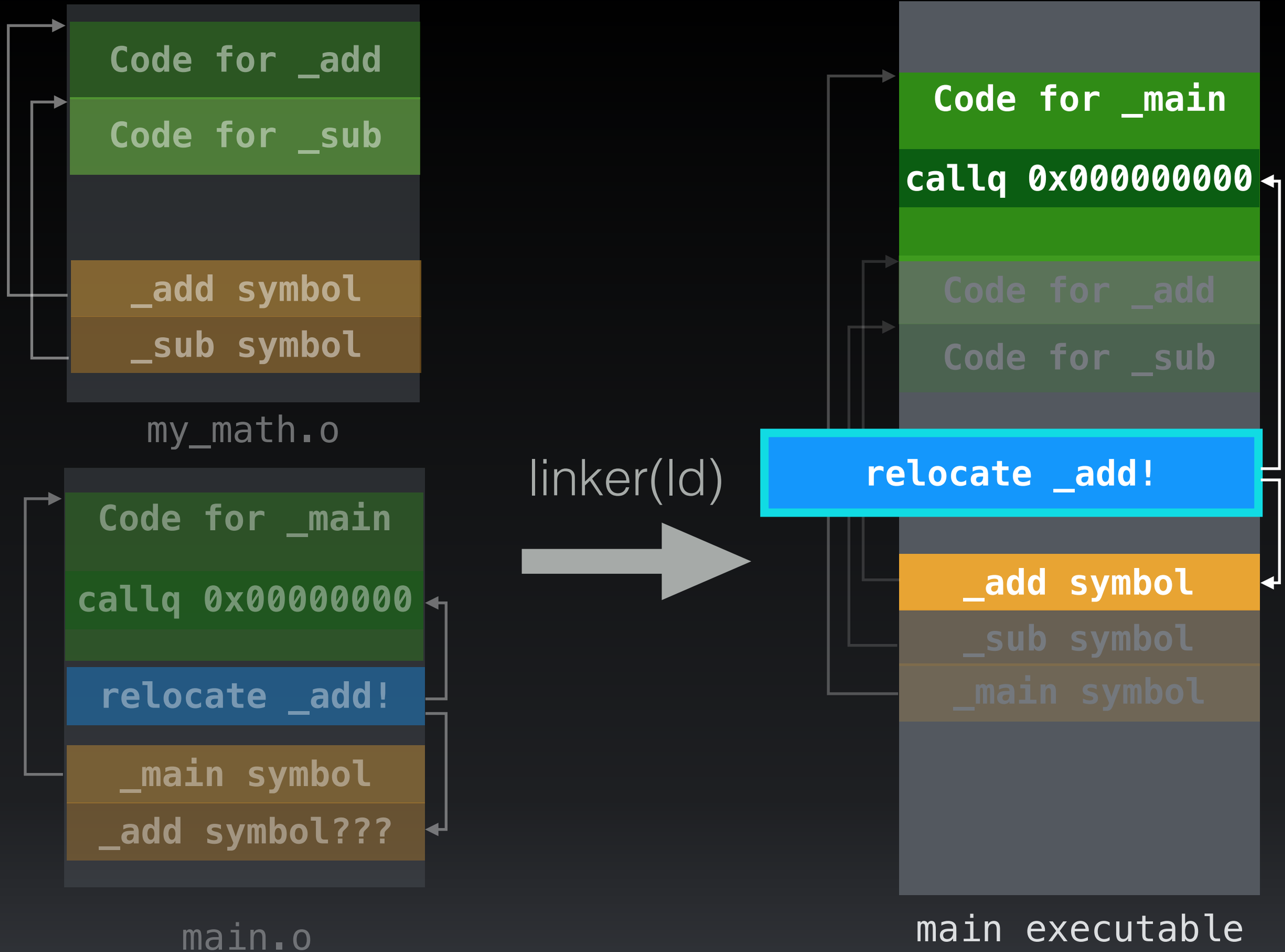
编译完成，进入链接

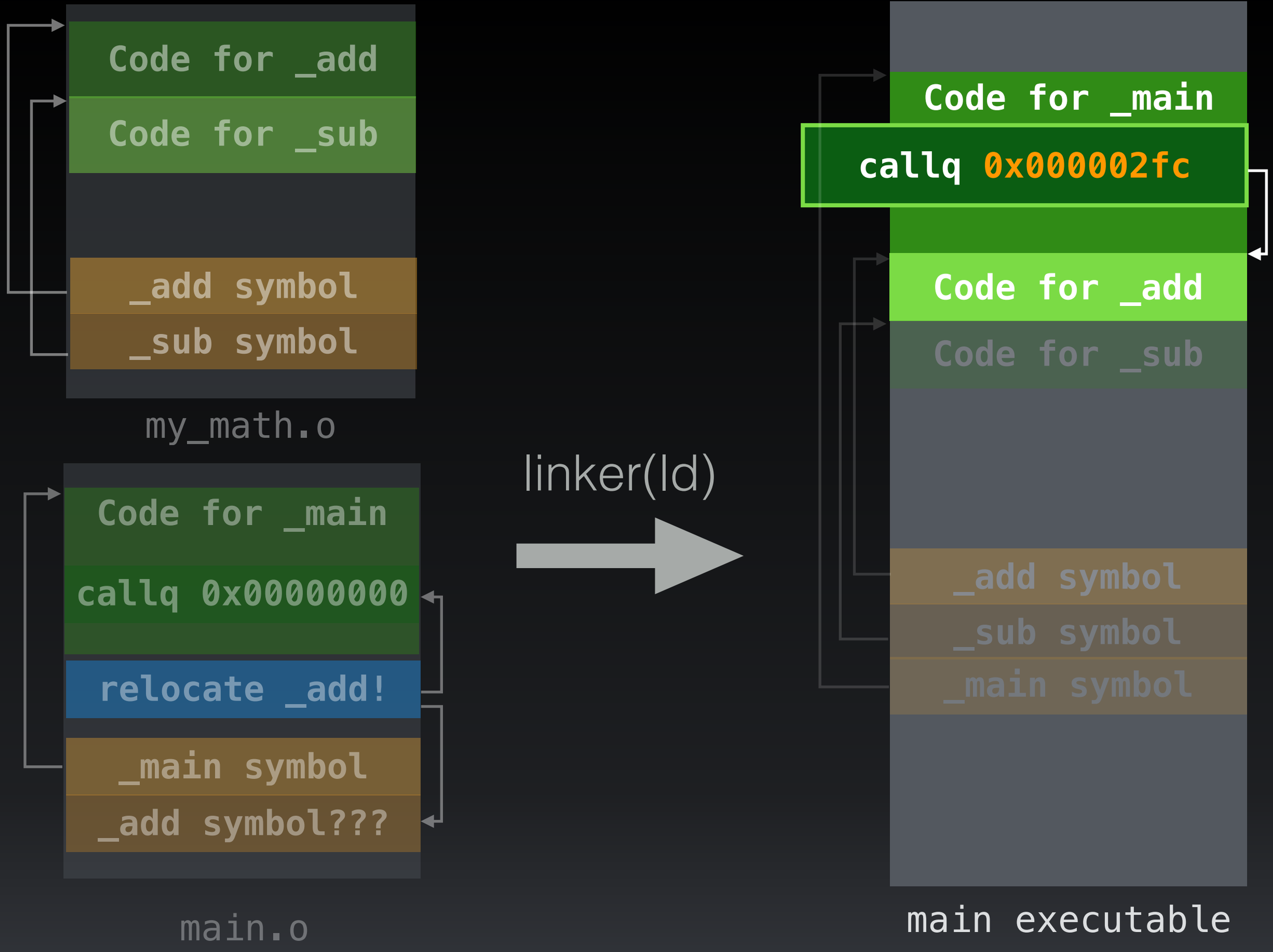


clang main.o my_math.o -o main


Code for _add
Code for _sub
_add symbol
_sub symbol
my_math.o
Code for _main
callq 0x00000000
relocate _add!
_main symbol
_add symbol???
main.o
linker(ld)
Code for _main
callq 0x000000000
Code for _add
Code for _sub
relocate _add!
_add symbol
_sub symbol
_main symbol
main executable

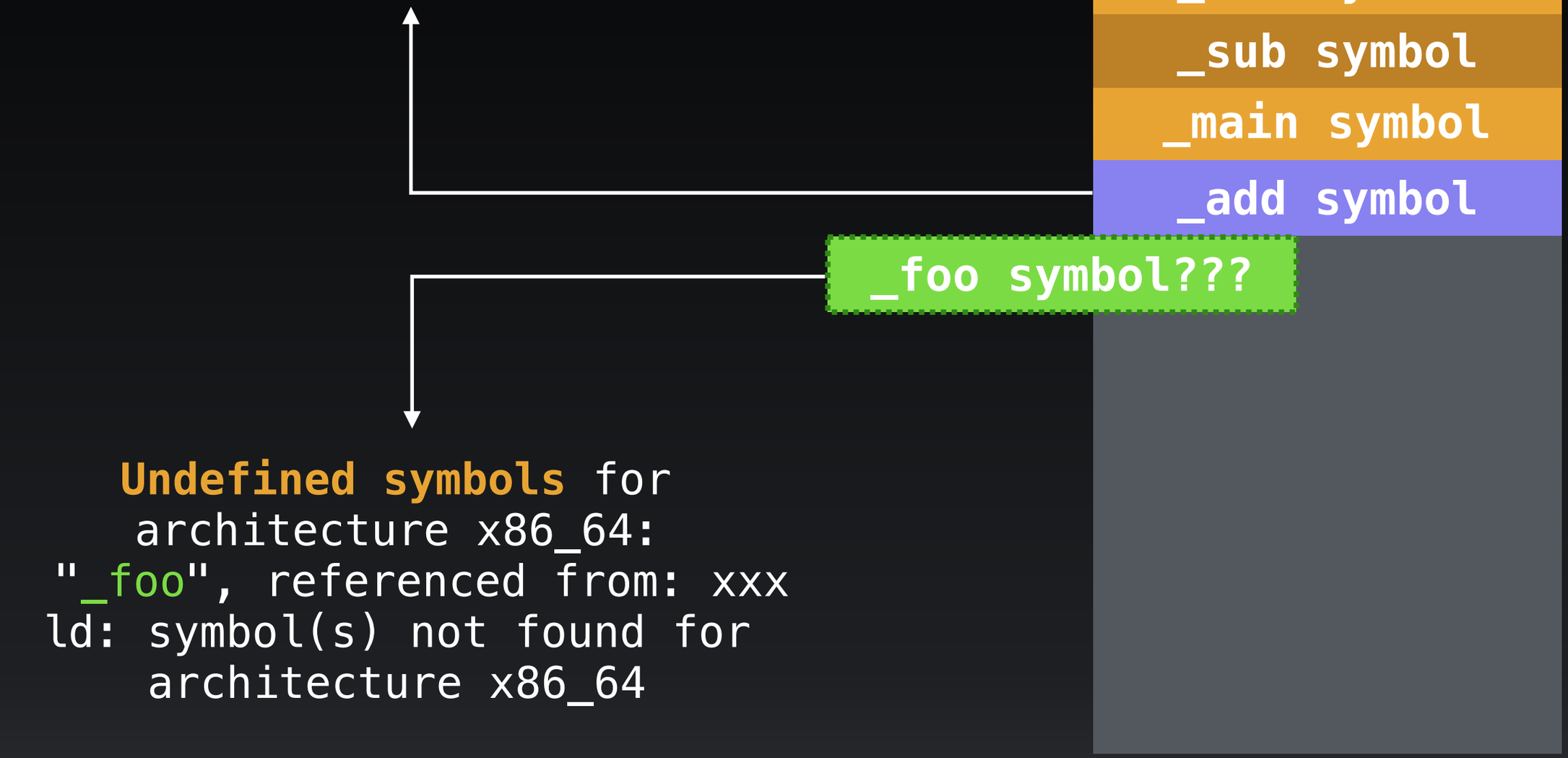

Code for _add
Code for _sub
_add symbol
_sub symbol
my_math.o
Code for _main
callq 0x00000000
relocate _add!
_main symbol
_add symbol???
main.o
linker(ld)
Code for _main
callq 0x000000000
Code for _add
Code for _sub
relocate _add!
_add symbol
_sub symbol
_main symbol
main executable


Code for _add
Code for _sub
_add symbol
_sub symbol
my_math.o
Code for _main
callq 0x00000000
relocate _add!
_main symbol
_add symbol???
main.o
linker(ld)
Code for _main
callq 0x000000000
relocate _add!
Code for _add
Code for _sub
_add symbol
_sub symbol
_main symbol
main executable


Code for _add
Code for _sub
_add symbol
_sub symbol
my_math.o
Code for _main
callq 0x00000000
relocate _add!
_main symbol
_add symbol???
main.o
linker(ld)
Code for _main
callq 0x000002fc
Code for _add
Code for _sub
_add symbol
_sub symbol
_main symbol
main executable

链接完成

```
ld: duplicate symbols for
   architecture x86_64
 clang: error: linker command
failed with exit code 1 (use -v
      to see invocation)
```



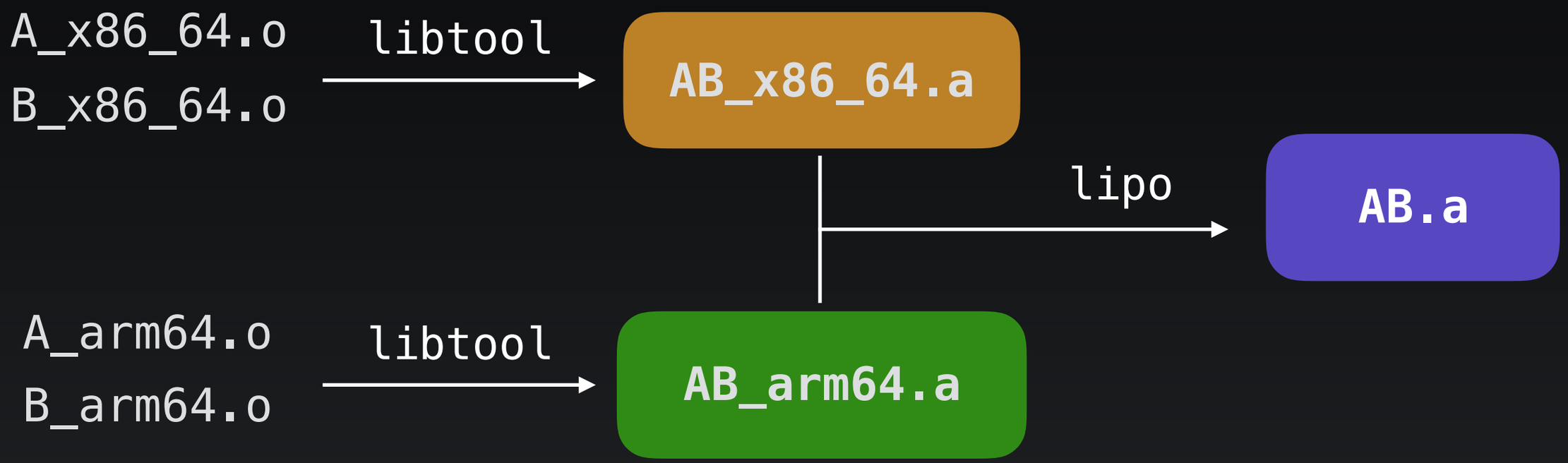

```
   Undefined symbols for
    architecture x86_64:
"_foo", referenced from: xxx
ld: symbol(s) not found for
    architecture x86_64
```

Static Library?

静态库 ≈ .o 的 zip 包

```
libtool -static my_math.o my_util.o -o my_math.a
```

Mach-O universal binary (fat binary)



Linker 将符号到地址的绑定

延迟到到**链接时**


後來……
劉若英

静态链接的问题

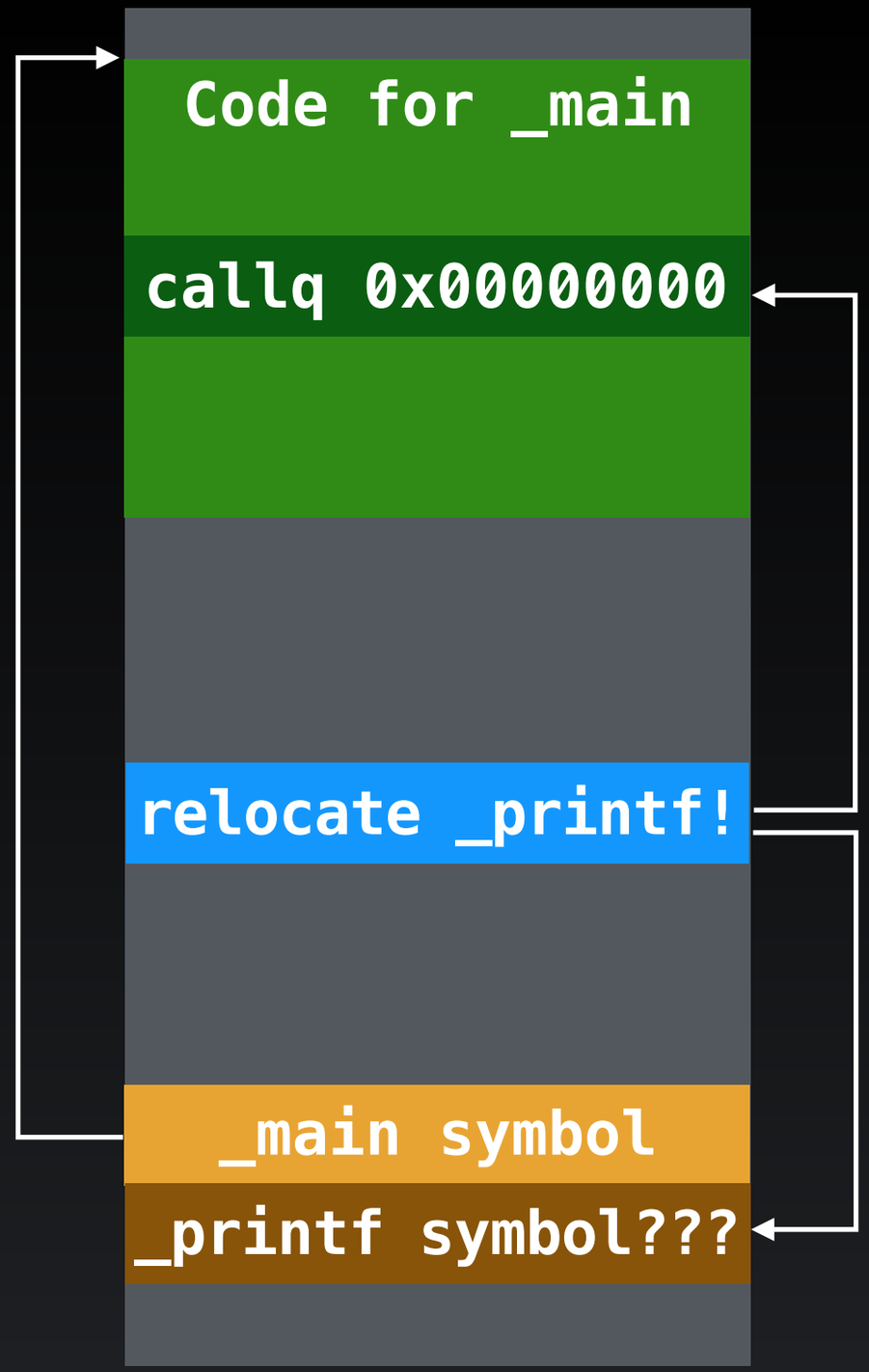
- **多个程序使用同一个库**（比如 stdio）但每个程序都要静态链接进自己的可执行文件，非常冗余
- **无法动态更新和加载**：依赖库升级后只能重新编译二进制文件

Shared Library / 动态链接过程出现


举个栗子

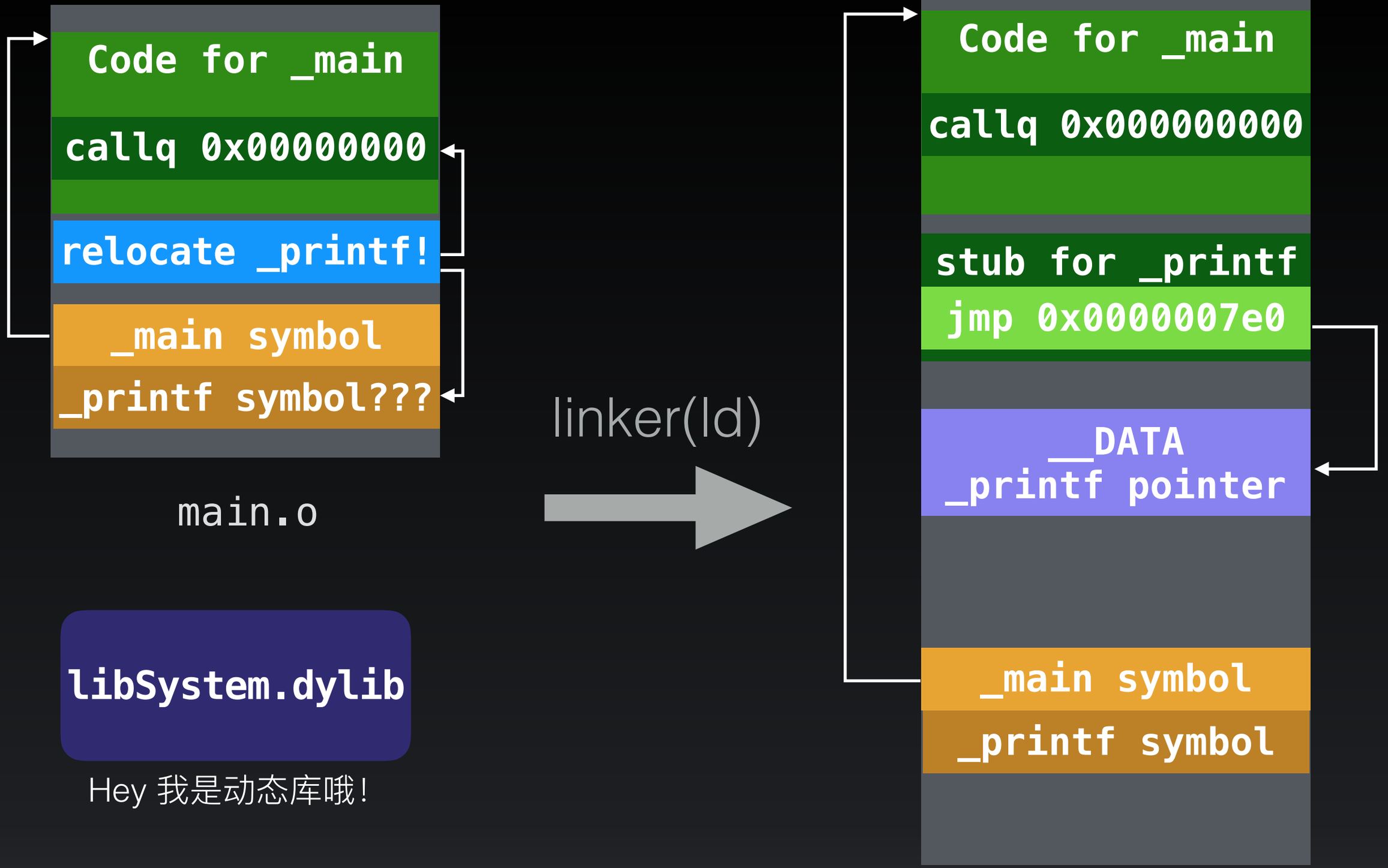
编译包含 printf 的 main

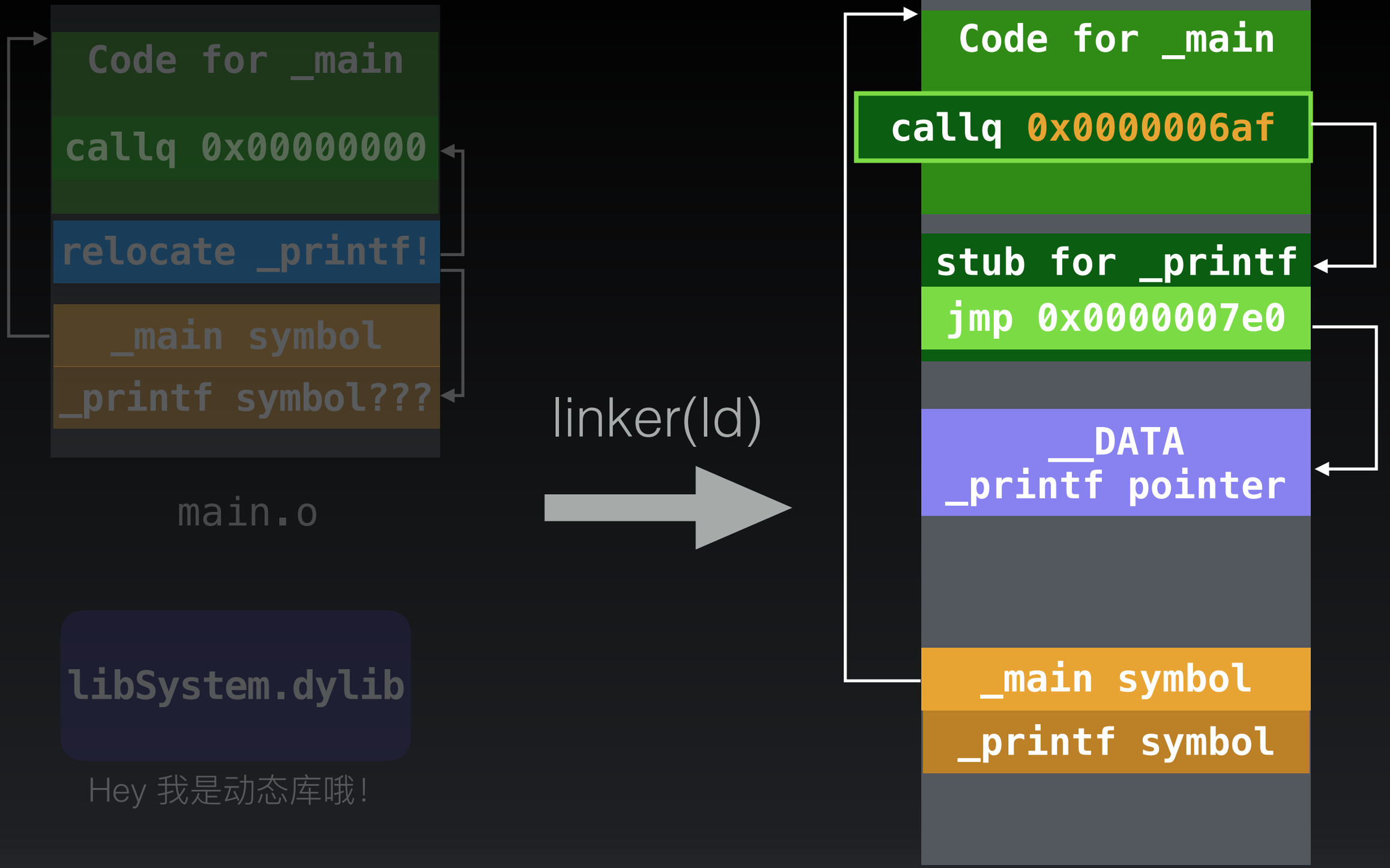
```
// clang -c main.c -o
// main.o

int main() {
    printf("hello world");
    return 0;
}
```



main.o

编译完成，进入链接


Code for _main
callq 0x00000000
relocate _printf!
_main symbol
_printf symbol???
main.o
linker(ld)
Code for _main
callq 0x000000000
stub for _printf
jmp 0x0000007e0
__DATA
_printf pointer
_main symbol
_printf symbol
main executable
libSystem.dylib
Hey 我是动态库哦!


Code for _main
callq 0x00000000
relocate _printf!
_main symbol
_printf symbol???
main.o
linker(ld)
Code for _main
callq 0x0000006af
stub for _printf
jmp 0x0000007e0
__DATA
_printf pointer
_main symbol
_printf symbol
main executable
libSystem.dylib
Hey 我是动态库哦!

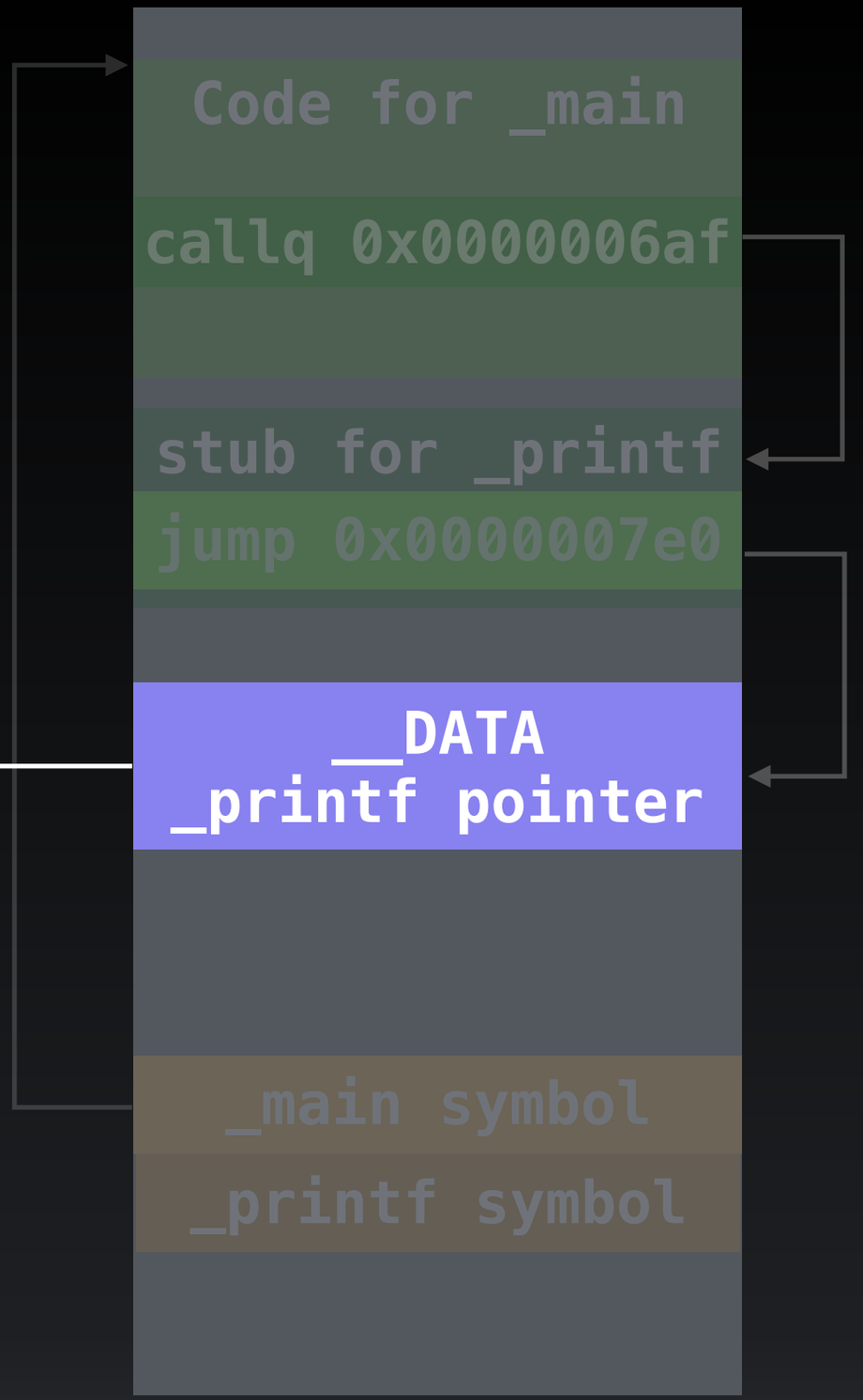
静态链接完成

dyld

(dynamic loader)

dyld 程序加载时 binding

- **Non-Lazy** Pointer Binding
  程序启动，加载时绑定
- **Lazy** Pointer Binding
  符号第一次被用到时绑定



为缩短程序启动时间

绝大多数 iOS SDK 中的符号

都采用 **Lazy Binding**

```
0x100000b95 <+213>: movq   %rdx, %rdi
0x100000b98 <+216>: movb   %al, -0x49(%rbp)
0x100000b9b <+219>: callq  0x100000c82               ; symbol stub for: objc_release
0x100000ba0 <+224>: leaq   0x4f9(%rip), %rdx         ; @"Hello, World!"
0x100000ba7 <+231>: movq   %rdx, %rdi
0x100000baa <+234>: movb   $0x0, %al
0x100000bac <+236>: callq  0x100000c64               ; symbol stub for: NSLog
0x100000bb1 <+241>: xorl   %ecx, %ecx
0x100000bb3 <+243>: movl   %ecx, %esi
0x100000bb5 <+245>: leaq   -0x28(%rbp), %rdx
0x100000bb9 <+249>: movq   %rdx, %rdi
0x100000bbc <+252>: callq  0x100000c94               ; symbol stub for: objc_storeStrong
0x100000bc1 <+257>: xorl   %ecx, %ecx
0x100000bc3 <+259>: movl   %ecx, %esi
```

调试中常见的 symbol stub

dyld 在 main 函数前做了什么

- 加载 dylibs (LC_LOAD_DYLIB)
- Fix-ups: Rebase 修正随机 base 地址偏移
- Fix-ups: Binding 确定 Non-Lazy Pointer 地址
- Load Objective-C Runtime 加载所有类
- Call Initializers: +load __attribute__((constructor))
- main()

WWDC 2016 《Optimizing App Startup Time》

手写机器码

code time

静态链接器

link time

动态加载器

run time

assemble time

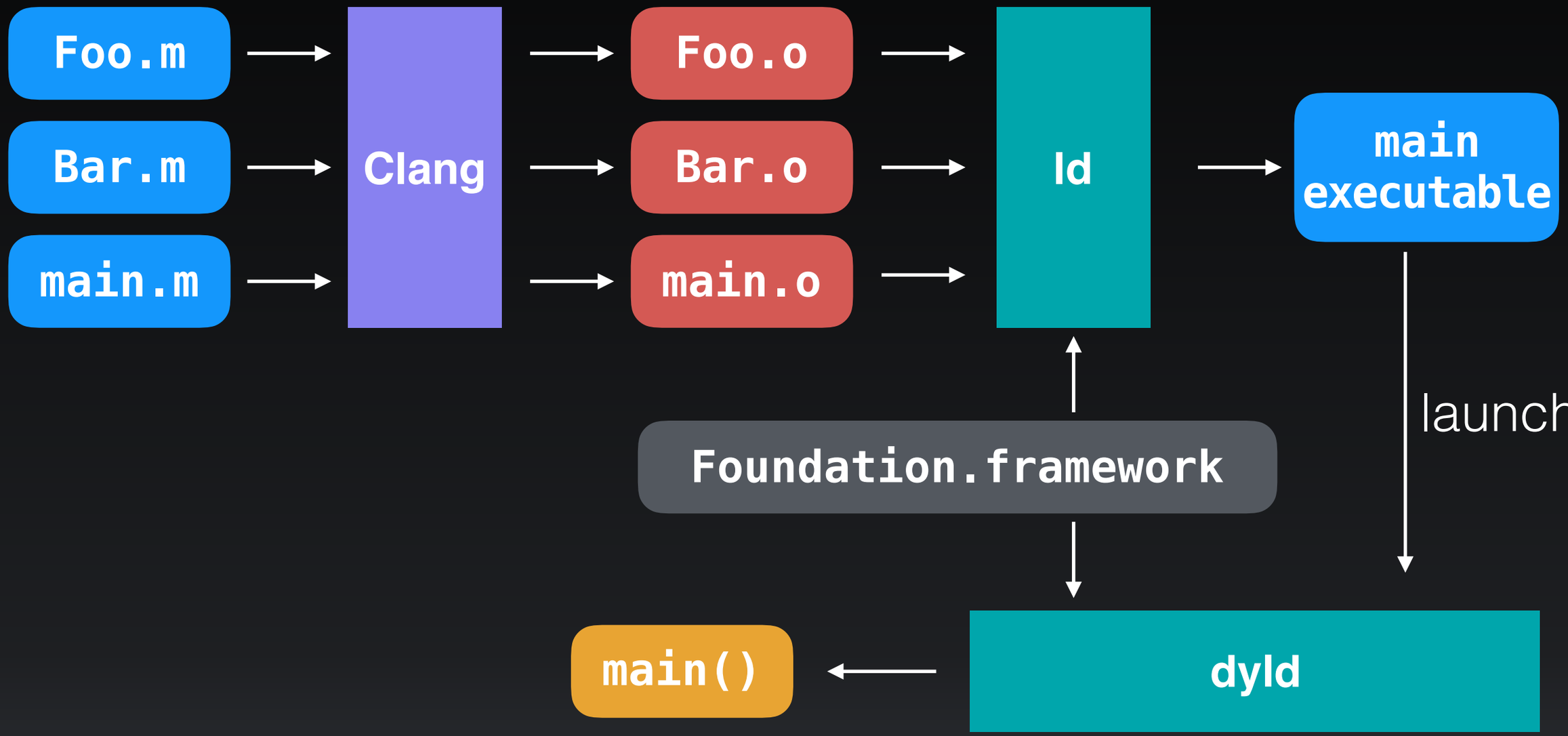
汇编器

load time

动态加载器

真实地址绑定时间

目前 Objective-C 的编译-链接-执行过程



Linker 和 Loader 相关技术实践

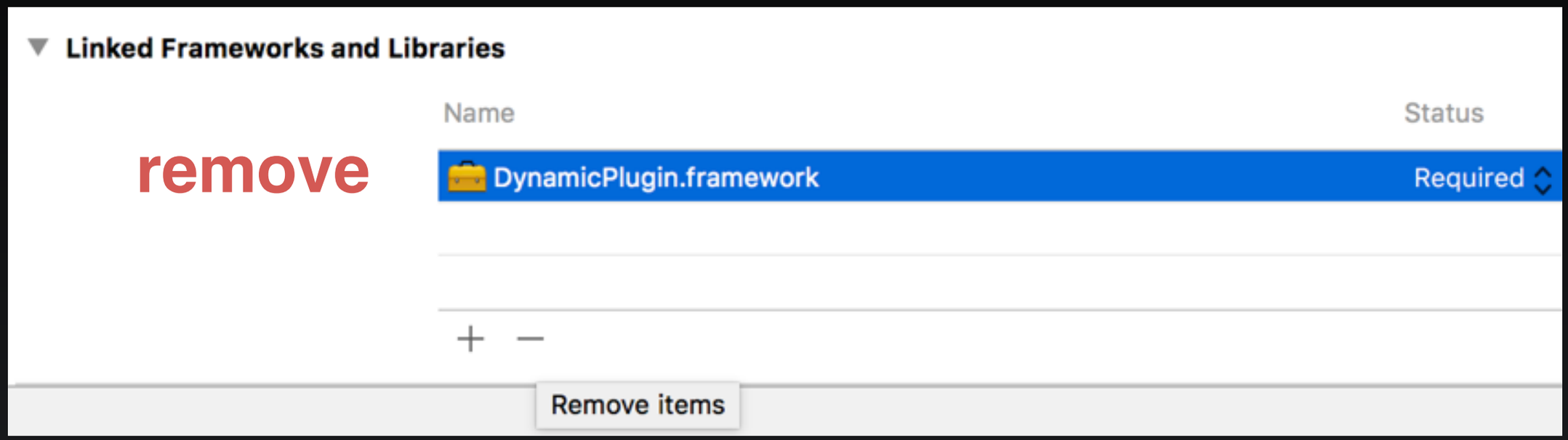
解决第三方库 Category 方法消失问题

- linker 默认**选择性链接**，只链接用到的 .o
- Objective-C 动态特性，到运行时才确定符号使用
- 在 Other Linker Flags 添加 **-ObjC** 强制链接

猜想 Apple 如何审核私有 API 调用?

- 查看 Load Commands 中的 LC_LOAD_DYLIB 是否动态链接私有 dylib， **otool -L**
- 查看 Symbol Table，有没有私有符号， **nm -u**
- 在 __TEXT,__objc_methname 段查找 **objc selector** 的使用
- 在 __TEXT,__cstring 中查找字符串的使用

按需加载 Embed Framework



不直接链接 Embed Framework

按需加载 Embed Framework

```
    NSString *bundlePath = [[NSBundle mainBundle]
pathForResource:@"DynamicPlugin" ofType:@"framework"];
    NSBundle *bundle = [NSBundle bundleWithPath:bundlePath];
     [bundle load];
```

```
Class cls = NSClassFromString(@"Sark");
```

或者使用 dlopen / dlsym

在非越狱手机上进行App Hook（蒸米）

- 修改可执行文件 Mach-O，向 Load Commands 插入 **LC_LOAD_DYLIB** 来加载自己的 dylib (**yololib**)
- 使用 **__attribute__((constructor))** 获得程序 load 回调
- 使用 **class-dump** 找到感兴趣的方法
- runtime swizzle

黑科技：将第三方可执行文件转成 dylib（杨君）

- 可执行文件和动态库都是经过链接过程的 Mach-O，**结构非常类似**（静态库只是 .o 的集合，没有经过链接）
- 修改可执行文件的 **Mach Header**、**Load Commands**、**Symbol Table**、**Rebase Info** **等**，使之变成 dylib
- 像使用 dylib 一样调用里面的方法（如支付宝里的加密解密算法）

fishhook - 替换系统 C 函数

- 利用动态链接 **Lazy / Non-Lazy Pointer** 机制
- 通过函数名从 **Symbol Table / String Table** 中寻找要 Hook 的符号地址
- 由于 Pointer 在 __DATA 段，可修改，将其值改成要替换的函数地址，完成 Hook
- 无法修改应用内的 C 函数（因为 release 时会 strip 掉内部符号表）
- fishhook + libffi 可以用于 hot patch，但没太大意义

天书终于讲完了

References

- 《linkers and loaders》
- http://www.codeproject.com/Articles/187181/Dynamic-Linking-of-Imported-Functions-in-Mach-O
- http://blog.imjun.net/2016/10/08/黑科技：把第三方-iOS-应用转成动态库/
- 《iOS 冰与火之歌》by 蒸米
- WWDC 2016 《Optimizing App Startup Time》

Q&A

我就叫Sunny怎么了

扫一扫二维码图案，关注我吧