Bonus 1：生成共享库

完成了主线任务后，你已经实现了一个功能完整的静态链接器。现在让我们探索另一种重要的代码组织方式——共享库（shared library）。

静态链接的代价

想象你的系统中运行着十个不同的程序。每个程序都调用了标准C库的函数——`printf`、`malloc`、`strlen`等等。如果这些程序都是静态链接的，那么C库的代码会在内存中存在十份完全相同的副本。这不仅浪费了内存，也浪费了磁盘空间。

更重要的是维护成本。假设C库发现了一个安全漏洞需要修复。对于静态链接的程序，你必须重新编译每一个使用了该库的程序，重新分发所有的可执行文件，用户也必须更新每一个程序。这个过程既耗时又容易出错——很可能有些程序会被遗漏，导致系统中仍然存在漏洞。

共享库提供了另一种选择。库的代码只在系统中存在一份，所有程序在运行时动态地加载和使用它。当库需要更新时，只需要替换那一个库文件，所有使用它的程序自动获得更新。这就是"共享"的含义——代码被共享，维护工作也被简化了。

共享库的本质挑战

共享库引入了一个新的技术问题。在确定基址的链接模式中，链接器知道程序会被加载到`0x400000`，所以可以计算出每个符号的绝对地址。但共享库不知道自己会被加载到哪里——不同的程序可能把它加载到不同的地址，甚至同一个程序的不同运行也可能加载到不同的地址（地址空间随机化的一部分）。

这意味着共享库不能依赖绝对地址。它的代码必须能够在任何地址正常工作——这就是位置无关代码（Position Independent Code, PIC）的要求。你在前面的任务中已经见过这个概念的一个例子：函数调用使用PC相对寻址，不管代码加载到哪里，只要函数之间的相对位置保持不变，跳转就能正确工作。

在这个任务中，你将实现生成共享库的能力。但为了让任务更聚焦于共享库的核心概念，我们采用一个在历史上广泛使用过的方案——运行时代码修正（Load-time Relocation）。在这个方案中，共享库在链接时不需要完全做到位置无关。当共享库引用外部符号时，链接器只需要在输出文件中标记"这里有一个外部引用"。加载器在加载共享库时，会查找这些外部符号的地址，然后直接修改代码段，填入正确的地址或偏移。

这种方案有一个重要的缺点：代码段在加载时被修改了，变成了"脏页"，无法在多个进程之间共享物理内存。每个使用这个库的进程都需要自己的一份代码段副本。虽然库文件本身只有一份，但在内存中实际上存在多份副本，失去了"共享"的主要优势。

但从教学角度，这个简化让我们可以专注于共享库的基本概念——符号的导出、外部引用的保留、动态重定位的意义——而不需要立即处理GOT和PLT这些更复杂的机制。

> **Note**
> 真实的共享库使用完全的位置无关代码（PIC）。库内对外部符号的访问也通过全局偏移表（GOT）和过程链接表（PLT）间接完成，代码段在运行时保持完全只读，可以在多个进程之间真正共享内存。
>
> 我们在这个任务中采用简化方案，是为了让你先建立对共享库基本机制的理解。当你完成Bonus 2并掌握了GOT/PLT机制后，可以尝试将这个任务改造为完整的PIC版本。我们会在后面说明如何改造。

> **Tip**
> 虽然我们称之为"简化方案"，但加载时修改代码段（Text Relocation）在计算机系统发展史上是一个真实存在过、被广泛应用过的技术。
>
> 在一些Unix系统的早期动态链接实现中，位置相关的共享对象确实需要运行时重定位。那时的系统运行的程序数量相对较少，每个程序都有自己的地址空间。而随着服务器和工作站需要同时运行越来越多的进程，代码段失去共享性带来的内存成本变得更加显著。这种变化推动了完全位置无关代码的普及——通过GOT和PLT机制，把需要修改的内容放到数据段，让代码段保持只读和可共享。
>
> Windows的动态链接库（DLL）采用了一个有趣的混合策略。每个DLL有一个"首选基址"，如果能加载到这个地址，就不需要任何修改，可以直接映射共享的代码页。但如果地址被占用或者启用了地址空间随机化（ASLR），加载器就需要应用重定位表（Rebasing）来修补地址引用。到了64位Windows时代，虽然地址空间变大大大降低了地址冲突的概率，但重定位机制仍然存在，特别是在ASLR启用的情况下，它依然在发挥作用。
>
> 在32位x86架构上，位置无关代码还有一个实际的代价。x86的通用寄存器本来就很少，传统的PIC实现需要保留一个寄存器（通常是`%ebx`）来维护GOT指针，这在寄存器压力大的代码中会影响性能。因此在那个时代，一些场景下会权衡是否值得为了共享性而承担这个开销。到了x86-64时代，通用寄存器增加到16个，并且引入了RIP相对寻址，这让PIC的实现变得更加自然，性能代价也大幅降低。
>
> 即使在现代系统中，text relocation也不是完全消失的历史遗迹。比如Android在其早期版本中需要处理大量遗留的text relocation，这带来了内存浪费和安全隐患，最终在较新的API级别中被限制甚至禁止。在某些特定场景下——比如只运行固定几个程序的嵌入式系统，不需要在多个进程间共享代码——允许加载时修改代码段可以简化链接器实现，避免间接访问的开销。
>
> 理解这个机制的演进历史，不仅对理解现代动态链接有帮助，也能让你在阅读不同平台的系统编程文档时，明白为什么它们的加载器设计会有所不同，以及这些设计背后的工程权衡。

实验约束与假设

在开始实现之前，需要明确本任务的一些约束和假设。这些约束简化了实现的复杂度，让你可以专注于核心概念。

第一个假设是关于库内符号的引用方式。我们假设编译器为共享库生成的代码中，库内符号之间的引用都使用PC相对寻址（RIP-relative）。比如，库内的函数A调用同一个库中的函数B，编译器会生成`call`指令的相对偏移形式。这样，即使库被加载到不同的地址，只要函数A和B的相对位置不变，调用就能正确工作。

如果代码中存在对库内符号的绝对地址引用——比如数据段中有一个函数指针表，表项存储的是库内函数的地址——那么这些地址也需要在加载时根据库的实际基址调整。在完整的实现中，这需要生成额外的重定位项（通常称为RELATIVE类型的重定位）。但在本实验的测试用例中，我们会避免这种情况，让你可以专注于处理外部引用。

第二个约束是关于动态符号表和重定位的组织方式。在标准ELF格式中，动态重定位项通过符号表索引来引用目标符号，这要求动态符号表既包含导出的符号（供其他模块使用），也包含未定义的外部符号（供重定位项引用）。在我们的FLE格式中，我们采用了一个简化的设计：动态重定位项直接存储符号的名字字符串，而不是索引。这意味着动态符号表只需要包含导出的符号，加载器可以通过名字直接查找外部引用的目标。

部分链接的概念

生成共享库本质上是一个"部分链接"的过程。链接器需要合并多个目标文件，但不是完全封闭地解析所有符号。

假设你要把两个目标文件链接成一个共享库：

```
// foo.c
extern int helper(int x);  // 在bar.c中定义

int public_func(int x) {    // 导出给其他程序使用
    return helper(x) + 1;
}

// bar.c
extern void printf(const char*, ...);  // 在libc.so中定义

int helper(int x) {
    printf("helper called\n");
    return x * 2;
}
```

链接器需要识别两类符号。`public_func`和`helper`是库内部的符号——它们在当前要链接的目标文件中被定义。`printf`是外部符号——它来自其他共享库，不在当前链接范围内。

对于内部符号之间的引用（`public_func`调用`helper`），链接器应该像静态链接那样处理：合并节内容，确定符号的最终位置，处理重定位。`helper`在合并后的`.text`节中有确定的偏移，链接器可以计算并填充正确的相对偏移。

但对于外部符号的引用（`helper`调用`printf`），链接器不应该尝试解析。它应该在输出文件中保留这个引用，标记为"需要运行时解析"。这就是动态重定位信息——告诉加载器"在某个位置有一个对某个符号的引用"。

需要注意的是，这些保留的重定位项在加载时会被处理成不同的形式。对于PC相对重定位（如函数调用），加载器会根据库的实际加载位置计算相对偏移并填入；对于绝对重定位（如数据段中的函数指针），加载器会填入符号的绝对地址。我们的测试用例中主要涉及前者——通过相对寻址调用外部函数。

动态符号表的作用

除了标记未解析的引用，共享库还需要告诉外界"我提供了哪些符号"。这就是动态符号表的作用。

其他程序想要使用这个库时，加载器需要能够查询"这个库是否提供了符号X？如果提供了，它在哪里？"动态符号表回答了这些问题。它包含库导出的所有符号，以及每个符号相对于库加载基址的偏移。

在这个任务中，我们可以采用一个简单的约定：所有全局符号（标记为📩的符号）都应该被导出。这包括强符号和弱符号，只要它们是全局定义的。局部符号（标记为🤏的）不应该被导出——它们只在库内部可见，不是库的公开接口。

你的链接器需要遍历所有已定义的全局符号，计算它们相对于库起始位置的偏移，构建动态符号表。注意只导出已定义的全局符号——那些未定义的外部符号不应该出现在导出表中，它们是库的依赖，而不是库提供的接口。

实现策略

生成共享库的链接过程与静态链接类似，但有几个关键的差异。

第一步仍然是合并节内容。遍历所有输入的目标文件，将它们的节按类别合并。这一步和静态链接完全一样。

第二步是符号解析，但这次需要区分内部和外部符号。构建全局符号表时，标记每个符号是在当前链接单元中定义的，还是外部的（未定义的）。一个实用的做法是先收集所有定义的符号，然后在处理重定位时，遇到的符号如果不在这个集合中，就标记为外部符号。

第三步是处理重定位，这里需要分支处理。当遇到一个重定位项时，检查目标符号是内部的还是外部的。如果是内部符号，像静态链接那样计算地址并填充——对于PC相对重定位，计算相对偏移；对于绝对重定位，计算绝对地址。如果是外部符号，不要填充，而是将这个重定位项保留到输出文件中，添加到动态重定位表。

第四步是生成动态符号表。遍历全局符号表，筛选出已定义的全局符号（未定义的不导出），记录每个符号的名称、大小、所在节、以及相对于库起始位置的偏移。

最后，设置输出文件的类型为`.so`，确保动态重定位表和动态符号表被正确写入。共享库不需要入口点（因为它不是独立运行的程序），但需要程序头来描述各个段的加载信息，就像可执行文件一样。

> **Tip**
> 在调试时，你可以用任务一实现的`nm`工具查看生成的共享库，确认导出的符号是否正确。你也可以检查动态重定位表，确认外部引用是否被正确保留。如果某个外部符号的引用被错误地解析了，或者某个内部符号的引用被错误地保留了，程序在运行时会出现意外的行为。

与静态链接的对比

完成这个任务后，你可以对比静态链接和共享库链接的差异。静态链接器追求"完全解析"——递归地查找每个符号的定义，直到所有引用都被解析或报告错误。最终的输出是一个封闭的、自包含的程序，不依赖外部的任何东西（除了操作系统内核）。

共享库链接器追求"有选择的解析"——只解析当前链接单元内部的符号，对外部符号保持开放状态。最终的输出是一个开放的模块，需要在运行时与其他模块组合。这个模块声明了它提供什么（导出符号表），也声明了它需要什么（动态重定位表中引用的外部符号）。

这种开放性正是共享库的核心价值。它允许代码在程序之间共享（至少在文件系统层面），允许库独立更新而不需要重新编译使用它的程序，允许延迟加载——程序可以在运行时决定是否加载某个库。但这也意味着程序的完整性要到运行时才能确定——如果加载时找不到需要的外部符号，或者找到的符号版本不兼容，程序才会报错。这是一种工程上的权衡。

完成后，运行测试来验证：

```
make test_bonus1
```