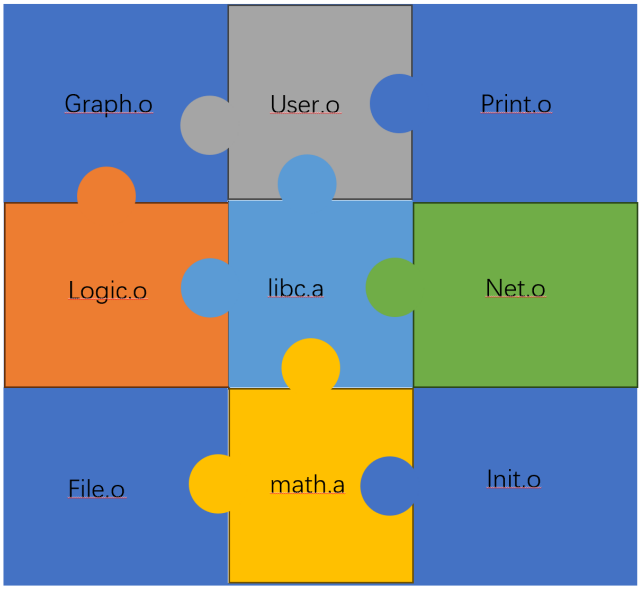
第二章：链接与库

在现代软件开发过程中，软件的规模往往都很大，动辄数百万行代码，如果都放在一个模块肯定无法想象。所以现代的大型软件往往拥有成千上万个模块，这些模块之间相互依赖又相对独立。这种按照层次化及模块化存储和组织源代码有很多好处,比如代码更容易阅读.理解、重用，每个模块可以单独开发、编译、测试，改变部分代码不需要编译整个程序等。

在一个程序被分割成多个模块以后,这些模块之间最后如何组合形成一个单一的程序是须解决的问题。模块之间如何组合的问题可以归结为模块之间如何通信的问题，最常见的属于静态语言的 C/C++模块之间通信有两种方式，一种是模块间的函数调用，另外一种是模块间的变量访问。函数访问须知道目标函数的地址，变量访问也须知道目标变量的地址，所以这两种方式都可以归结为一种方式，那就是模块间符号的引用。模块间依靠符号来通信类似于拼图版，定义符号的模块多出一块区域，引用该符号的模块刚好少了那一块区域，两者拼接刚好完美组合。这个模块的拼接过程就是本书的一个主题：链接。

2.1 为什么需要链接?



程序设计的模块化是人们一直在追求的目标，因为当一个系统十分复杂的时候，

我们不得不将一个复杂的系统逐步分割成小的系统以达到各个突破的目的。一个复杂的软件也如此，人们把每个源代码模块独立地编译，然后按照须要将它们“组装”起来，这个组装模块的过程就是链接(Linking)。

现在的链接器空间分配的策略基本上采用叫做两步链接的方法，也就是说整个链接过程分为两步：

第一步 空间与地址分配 扫描所有的输入目标文件，并且获得它们的各个段的长度、属性和位置，并且将输入目标文件中的符号表中所有的符号定义和符号引用收集起来，统一放到一个全局符号表。这一步中，链接器将能够获得所有输入目标文件的段长度，并且将它们合并，计算出输出文件中各个段合并后的长度与位置，并建立映射关系。

第二步 符号解析与重定位 使用上面第一步中收集到的所有信息，读取输入文件中段的数据、重定位信息，并且进行符号解析与重定位、调整代码中的地址等。事实上第二步是链接过程的核心，特别是重定位过程。

执行链接有三个时机，依据这三个实际，可以分为三种链接方式：

静态链接：在程序运行之前先将目标模块以及他们所需要的库函数链接成一个完成的可执行文件。

动态链接：将各个目标模块装入到内存，装入时再进行链接。

运行时链接：在程序执行需要该模块的时候，才进行链接。

2.1.1 系统调用与库

系统调用是内核提供给应用程序使用的功能函数，由于应用程序一般运行在用户态，处于用户态的进程有诸多限制（如不能进行 I/O 操作），所以有些功能必须由内核代劳完成。而内核就是通过向应用层提供系统调用，来完成一些在用户态不能完成的工作。

我们以最常见的 write 和 exit 为例：write 系统调用用于向文件描述符（通常是标准输出、文件或套接字）写入数据。它的原型如下：

```
ssize_t write(int fd, const void *buf, size_t count);
```

fd 是文件描述符，指定要写入数据的目标。buf 是要写入的数据的缓冲区的指针。count 是要写入的字节数。

exit 系统调用用于终止当前进程的执行，并可选地返回一个状态码。它的原型如下：

```
void exit(int status);
```

status 是要返回的状态码。通常，0 表示成功，非零值表示错误。

通过上述原型描述，我们可以用 write 系统调用实现 printf()函数：

```
#include <stdio.h>
#include <stdarg.h>
#include <unistd.h>

void my_printf(const char *format, ...) {
    va_list args;
    va_start(args, format);
    char buffer[1024]; // 假设格式化后的字符串不超过 1024 字节
    int length = vsnprintf(buffer, sizeof(buffer), format, args);
    write(1, buffer, length);
    va_end(args);
}

int main() {
    my_printf("Hello, %s!\n", "this is myprint");
    return 0;
}
```

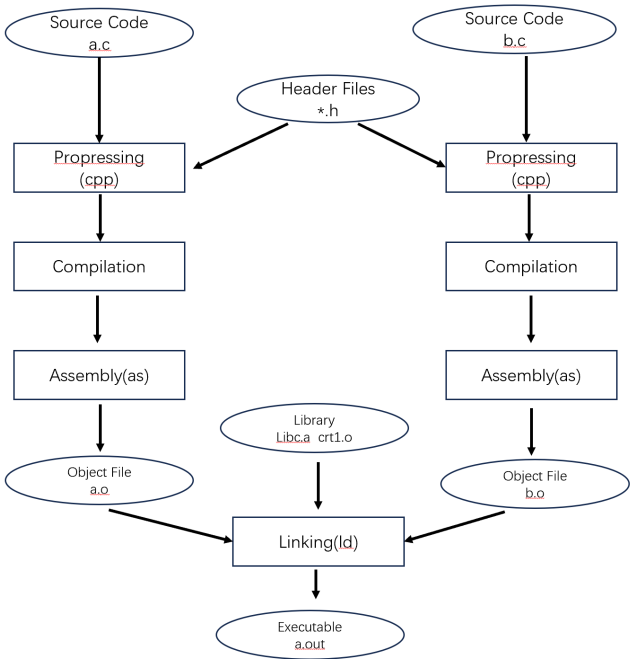
同样的，也可以使用 exit 系统调用实现简单的 exit()函数：

```c
#include <stdlib.h>
int my_exit(int status) {
    exit(status);
}
int main() {
    my_exit(0); // 正常退出
    return 0;
}
```

2.2 静态链接

2.2.1 什么是静态链接

当我们有两个目标文件时，如何将他们链接起来行程一个可执行文件？这个过程中发生了什么？这基本上就是链接的核心内容：静态链接。



最基本的静态链接如上图所示，每个模块的源代码文件（如.c）文件经过编译器编

译成目标文件（Object File，一般扩展名为.o 或.obj），目标文件和库（Library）一起链接形成最终可执行文件。

在这一节里，我们将使用下面这两个源代码文件 hello.c 和 main.c,以及 hello.c 的头文件 hello.h 作为例子展开分析：

```
// hello.c
#include <stdio.h>
int hello(){
    printf("hello, world\n");
    return 0;
}
```

```
// hello.h
#ifndef HELLO_H
#define HELLO_H

int hello();

#endif
```

```
// main.c
#include "hello.h"
int main(){
    hello();
    return 0;
}
```

假设我们的程序只有这两个模块，首先我们使用 gcc 将 hello.c 和 main.c 分别编译成目标文件 hello.o 和 main.o。

```
$ gcc -c main.c -o main.o
$ gcc -c hello.c -o hello.o
```

然后，我们使用链接器将 hello.o 和 main.o 链接起来：

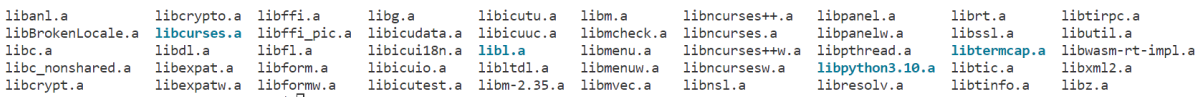
```
$ gcc main.o hello.o -o a.out
```

2.2.2 使用

事实上，编译器一共了许多标准静态链接库，在 Linux 下，标准静态链接库位于许多目录下，例如：

```
$ cd /usr/lib/x86_64-linux-gnu
$ ls *.a
```

我们就可以查看在/usr/lib/x86_64-linux-gnu 目录下的链接库：



libanl.a	libcrypto.a	libffi.a	libg.a	libicutu.a	libm.a	libncurses++.a	libpanel.a	librt.a	libtirpc.a
libBrokenLocale.a	libcurses.a	libffi_pic.a	libicudata.a	libicuuc.a	libmcheck.a	libncurses.a	libpanelw.a	libssl.a	libutil.a
libc.a	libdl.a	libfl.a	libicui18n.a	libl.a	libmenu.a	libncurses++w.a	libpthread.a	libtermcap.a	libwasm-rt-impl.a
libc_nonshared.a	libexpat.a	libform.a	libicuio.a	libltdl.a	libmenuw.a	libncursesw.a	libpython3.10.a	libtic.a	libxml2.a
libcrypt.a	libexpatw.a	libformw.a	libicutest.a	libm-2.35.a	libmvec.a	libnsl.a	libresolv.a	libtinfo.a	libz.a

其中，libc.a 是 C 标准库的静态版本。这个库包含了许多标准 C 函数的实现，例如输入输出函数、字符串处理函数、内存管理函数等；libutil.a 通常包含与系统实用工具和功能有关的函数。这个库中的函数通常与系统编程、文件操作、进程管理等有关。每个库文件中都包含了大量的功能模块，这些模块可以在应用程序中复用。

输入

```
$ ar t libc.a
```

我们就可以查看 libc.a 下的内容：

```
wmemchr-evex-rtm.o
offtime.o
asctime.o
clock.o
ctime.o
ctime_r.o
difftime.o
gmtime.o
localtime.o
mktime.o
time.o
gettimeofday.o
settimeofday.o
settimezone.o
```

事实上，每个.a 文件其实封装了多个.o 目标文件，这有利于更好地组织、管理和重用编译后的代码，在上面的例子中 main.o 使用到了 hello.o 目标文件，那么自然而然地，我们也可以把我们自己的 hello.o 封装进.a 文件，用下述指令更新该静态链接库：

```
$ ar r libc.a /.../hello.o
```

再次查看 libc.a 里面所包含的所有目标文件，可见 hello.o 已经成功添加：

```
dl-profstub.o
dl-reloc-static-pie.o
dl-support.o
dl-sym.o
dl-sysdep.o
enbl-secure.o
libc_early_init.o
rtld_static_init.o
get-cpuid-feature-leaf.o
hello.o
```

2.2.3 创建

经过之前的操作，我们已经成功把 hello.o 添加进了静态链接库中，接下来，我们就可以使用静态链接直接链接库中的 hello 模块了，我们使用-static 命令，进行静态链接，然后运行 a.out，成功输出 hello world。

```
$ gcc main.o -static -o a.out
$ ./a.out

hello, world
```

我们也可以把 hello.o 打包成 libhello.a 文件，制作自己的静态库：

```
ar rcs libhello.a hello.o
```

使用静态链接方式生成可执行文件，注意：此处我们需要指定链接库的名称和所在位置，-L.表示当前目录。-lhello 是 libhello.a 库的名称。

```
gcc main.o -static -o a.out -lhello -L.
```

2.3 动态链接

2.3.1 什么是动态链接

静态链接使得开发者能够相对独立地开发和测试自己的程序模块，大大促进了程序开发的效率，但是，静态链接的方式对于计算机内存和磁盘的空间浪费非常严重，特别是在多进程操作系统情况下，因为每个程序内部都保留着类似于 printf()函数、scanf()函数，strlen()等这样的公用库函数，极大地浪费了内存的空间。静态链接的另一个问题是对程序的更新、部署和发布带来很多麻烦，每当有一个模块修改后，都需要对整个程序进行重新链接，而后将新的版本整个发布给用户，更新程序将会非常不便。

要解决这两个问题，最简单的方式就是把程序的模块相互分割开来，形成独立的文件，而不再将他们静态的链接在一起。简单的讲，就是不对那些组成程序的目标文件进行链接，等到需要运行时才进行链接。也就是说，把链接这个过程推迟到运行时再进行，这就是动态链接的基本思想。

动态链接可以让程序在运行时可以动态地选择加载各种程序模块，还可以减少可

执行文件的大小，因为库的代码不会重复出现在每个可执行文件中。这可以节省磁盘空间，尤其是当多个程序使用相同的库时。当然，动态链接也有诸多的问题， 一个很常见的问题是，当程序所依赖的某个模块更新后，由于新的模块和旧的模块之间接口不加弄，导致原有的程序无法运行，这个问题在早期的 Windows 版本中尤为严重。

仍然使用上一节提到的例子，我们首先编译 main.c 文件:

```
$ gcc -c -fno-builtin main.c -o main.o
```

其中，-fno-builtin 表示禁用 gcc 内置的函数优化，以防止 gcc 优化后导致目标文件中的 hello 函数与链接模块不匹配。

而后编译 hello.c 文件：

```
$ gcc -c -fPIC hello.c -o hello.o
```

-fPIC 用于生成位置无关代码，它可以在内存中的任何位置运行而不受其实际加载地址的限制。具体来说，-fPIC 选项会告诉编译器生成位置无关代码，以便在共享库中使用。这是因为共享库可以被加载到不同的进程的内存地址中，每个进程的内存布局可能不同。使用位置无关代码可以确保代码中的地址引用是相对的，而不是绝对的，从而使共享库能够在不同的进程中正确运行。

```
$ gcc -shared hello.o -o libhello.so
```

-shared 告诉编译器生成一个共享库。共享库是一种包含可执行代码的文件，它可以在程序运行时被动态加载和链接。在 linux 系统中，共享库的文件扩展名通常是 .so，通过上述命令，就可以将 hello.o 目标文件编译成一个名为 libhello.so 的共享库文件。

然后，将目标文件 main.o 和共享库进行链接:

```
$ gcc main.o -L./ -lhello -o a.out
```

2.3.2 动态链接库的使用

由于动态链接的诸多优点，大量的程序开始使用动态链接机制，系统中就会存在着大量共享的程序模块，为了维护这些共享的程序模块，我们将其放入共享库中。改

变共享库查找路径最简单的方法是使用 LD_LIBRARY_PATH 环境变量，这个方法可以临时改变某个应用程序的共享库查找路径，而不会影响系统中的其他程序。

在 linux 系统中，LD_LIBRARY_PATH 是一个由若干个路径组成的环境变量，动态链接器在查找共享库时，会首先查找由 LD_LIBRARY_PATH 指定的目录。

现在，我们把 libhello.so 的位置添加到 LD_LIBRARY_PATH 中，使 main.o 在后续运行时能够找到 hello()函数。

```
$ export LD_LIBRARY_PATH=./
```

我们尝试运行 a.out，可以得到正确结果：

```
$ ./a.out

hello, world
```

至此，动态链接完成。

2.3.3 安全问题

虽然我们已经得到了正确结果，但是这个过程中其实存在着非常严重的安全问题，如果攻击者能够控制 LD_LIBRARY_PATH 的地址，那么，攻击者就可以生成一个与我们同名的共享库，在共享库声明同名函数，而在函数体中实现入侵系统的功能，这样我们在执行可执行文件时，会将黑客的共享库链接进来，导致系统出现问题。

此处有一个 danger.c，其中同样实现了 hello()方法：

```
extern int puts(char *);

int hello(){
    puts("you are going to output: ");
    // do something;
    // ...;
    puts("\nyou are hacked...\n");
    return 0;
}
```

将其进行编译：

```
$ gcc -c -fPIC hello.c -o hello.o
```

然后生成共享库，也叫做 libhello.so:

```
$ gcc -shared hello.o -o libhello.so
```

然后在进行动态链接时，就会把有安全问题的 danger.c 中的 hello()函数链接进来了：

```
$ gcc main.o -L./ -lhello -o a.out
```

执行结果：

```
$ ./a.out
you are going to output:
you are hacked...
```

从 danger.c 中我们可以看到，攻击者定义了外部的指针，在真实情景下，这将会造成严重的问题。

2.4 运行时链接

2.4.1 什么是运行时链接

运行时链接（Runtime Linking）是一种在程序执行过程中动态加载和链接模块或库的过程。它允许程序在运行时（而不是编译时或加载时）获取和使用外部模块、库或者库函数。运行时链接使得程序的模块组织更加灵活，由于外部模块不会在启动时全部加载到内存中，运行时链接可以有效减少程序的启动时间和内存占用。此外，使用动态链接时，程序可以不必重启就完成外部模块的增加、删除和更新等操作。

2.4.2 运行时链接

运行时链接的完成主要依靠动态链接器提供的 4 个 API：dlerror()、dlopen()、dlsym()和 dlclose()，使用它们时需要包含#include <dlfcn.h>。

它们的函数原型为：

```
void *dlopen(const char *filename, int flag);
void *dlsym(void *handle, const char *symbol);
int dlclose(void *handle);
const char *dlerror(void);
```

dleror()可以检测其他动态链接库函数执行状况，当动态链接库操作函数执行失败时，dlerror 可以返回出错信息，返回值为 NULL 时表示操作函数执行成功。

dlopen()的作用是打开一个动态库，并将其加载到进程的地址空间，完成初始化过程。

filename 是被加载的.so 动态库的路径。如果是绝对路径，函数会直接尝试打开此动态库；如果是相对路径，函数会按以下顺序查找该动态库文件：

1. 查找环境变量 LD_LIBRARY_PATH 指定的目录
2. 查找由 /etc/ld.so.cache 里面指定的共享库路径
3. /lib, /usr/lib

注意，如果把该参数设置为 0，将会返回全局符号表的句柄。也就是说，我们可以

在运行时找到全局符号表里的任何一个符号并执行它们。

flag 是函数符号的解析方式。取值可以是 RTLD_LAZY / RTLD_NOW | RTLD_GLOBAL，其中，RTLD_LAZY 表示延迟绑定，即当函数第一次被调用时才进行绑定，而 RTLD_NOW 则是模块被加载时就进行绑定。这两种绑定方式只能选择一个，而它们都可以与 RTLD_GLOBAL 一起使用，表示将被加载的模块的全局符号合并到进程的全局符号表中，使得以后加载的模块可以使用这些符号。

dlopen()的返回值是被加载模块的句柄，供后面调用其他函数时使用。如果加载失败，返回 NULL。如果模块已经被加载过了，则返回同一个句柄。

dlsym()的功能是从动态库中获取符号（全局变量与函数）的地址。其中 handle 可以是 dlopen()函数返回的句柄，也可以是 RTLD_DEFAULT 或 RTLD_NEXT。

RTLD_DEFAULT 代表按默认顺序搜索动态库中符号 symbol 第一次出现的地址，RTLD_NEXT 代表在当前库以后按默认顺序搜索动态库中符号 symbol 第一次出现的地址。

dlclose()和 dlopen()的功能恰好相反，其作用为卸载掉已经加载的某个动态库。handle 为待卸载动态库的句柄。事实上，系统会为动态库维持一个加载引用计数器，每当调用 dlopen()函数加载一个动态库时，相应的计数器会加 1；调用 dlclose()卸载一个动态库时，计数器便减 1。而只有当计数器减至 0 时，动态库才会真正被卸载。

利用运行时链接，我们甚至不必重新启动应用程序就能改变程序的运行状态。例如改写 main.c:

```
//main.c
#include <dlfcn.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]){
    void *h;
    void (*func)(void);
    while(1){
        h = dlopen(argv[1], RTLD_NOW|RTLD_GLOBAL);
        func = dlsym(h, argv[2]);
        func();
        dlclose(h);
        sleep(5);
    }

    return 0;
}
```

此时编译时无需再指定动态库：

```
$ gcc main.c
```

我们可以在运行时指定我们需要的动态库和执行的函数，例如：

```
$ ./a.out ./libhello.so hello
hello world
hello world
hello world
```

程序每隔 5 秒便会输出一次 hello world。而利用运行时链接，我们可以做到不重新启动程序，通过改变动态库的方式控制程序的运行，例如改写 hello.c：

```
//hello.c
#include <stdio.h>
#include <stdlib.h>

int hello(){
    printf("Bye bye!\n");
    exit(0);
}
```

再将其重新编译为动态库：

```
$ gcc -shared -fPIC hello.c -o libhello.so
```

程序的下一次输出便会开始发生变化，并且运行终止：

```
hello world
hello world
hello world
Bye bye!
$
```

可见在使用运行时链接后，我们能在程序执行过程中动态加载、更新和删除外部模块，能更进一步提高程序的灵活性、可维护性、可扩展性和资源利用率。

2.5 即时编译

2.5.1 什么是即时编译

即时编译（Just-In-Time Compilation，JIT）是一种在程序运行时将源代码或中间代码（通常是字节码）即时编译成机器码的技术。与传统的静态编译不同，即时编译延迟了编译的时机，将编译过程推迟到程序运行时的特定阶段。以根据当前的环境和上下文生成优化的机器码。即时编译器通常作为运行时环境的一部分，例如虚拟机、解释器或语言运行时。它相对于静态编译的主要优势在于，它能够利用运行时信息来

优化代码生成和执行。它经常用于动态语言（如 Java、JavaScript 等）的实现，也在许多虚拟机和解释器中使用，以加速程序的执行。此外，通过即时编译，我们也可以更自由地对程序进行更新和拓展，更进一步地提高程序的灵活度。

2.5.2 基于运行时链接的实现

通过上一节提到的运行时链接，我们可以实现简单的即时编译，改写 main.c:

```
//main.c
#include <dlfcn.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

char cmd[1024];

int main(int argc, char *argv[]){
    void *h;
    void (*func)(void);
    while(1){
        sprintf(cmd,"gcc -shared -fPIC %s -o libtmp.so",argv[1]);
        system(cmd);
        h = dlopen("./libtmp.so", RTLD_NOW|RTLD_GLOBAL);
        func = dlsym(h, argv[2]);
        func();
        dlclose(h);
        sleep(5);
    }
    return 0;
}
```

编译完成后可以在执行时指定想添加的.c 文件和需要运行的函数，程序便可以灵活地完成想要的功能：

```
$ gcc main.c
$ ./a.out ./hello.c hello
hello world
hello world
```

与前面类似，这时我们要想改变程序的运行状态，只需要更新指定的 C 文件或函数即可，也无需再手动编译动态库。这样的方法进一步增加了程序的灵活性，消除动态库带来的架构有关的影响，能够更自由地下发更新的文件或组件，提高程序的可移植性、可维护性和可拓展性。

2.6 本章小结

通过本章的学习，我们一起了解了链接的原理和几种链接方式。

静态链接在编译时进行链接，所有程序模块（函数和库）都被包含在可执行文件中，它不需要外部库文件，因为所有代码都已包含在可执行文件中，但是生成的可执行文件较大，因为它包含了所有所需的代码和数据。

为了解决可执行文件较大的问题，我们接下来介绍了动态链接的方式，动态链接在运行时进行链接，只有程序的引用部分被包含在可执行文件中，实际代码和数据存储在共享库文件中，这些库文件在运行时加载到内存中。动态链接更节省磁盘空间，因为多个程序可以共享相同的库。

如果在链接时我们还不知道要链接哪些库，那么此时就需要用到运行时链接了，通过参数的方式，在执行时传入共享库的名称和地址，运行时链接允许更大的灵活性和可扩展性，因为程序可以根据需要加载不同版本的库。

运行时链接所需要的库如果还没有编译成.o 文件，就可以即时编译技术，将链接库先进行编译再链接进来，完成程序的正常运行。

2.7 深入阅读

《Compilers: Principles, Techniques, and Tools》

《程序员的自我修养：链接、装载与库》