热部署可行性报告

目录

项目简介

我们的项目旨在为用户提供一个热部署的解决方案，让用户能在不对源码进行修改的情况下，使用我们的工具对程序进行热部署（即在不重启程序的情况下对程序进行更新）。我们采用了自顶而下的设计模式和分而治之的方法对整个项目的可行性进行了如下讨论。因为热部署是一件并不轻松的事情，因此我们先讨论了我们的工具打算提供什么程度的热部署支持，有哪些限制，怎样保证热部署的安全性。然后我们将热部署的具体实现拆分为几乎没有耦合的如下三个部分

- 工作进程控制权的获取
- 代码的修改
- 代码优化

我们仔细讨论了每个部分可选的解决方案。在最后的结论部分，我们提出了两种可行的方案组合。

热部署功能使用说明

对更新的限制

首先，我们对业务逻辑的具体实现进行分析。为了更加准确、安全、方便地实现业务逻辑的修改，我们顺应目前流行的"模块化开发"思想，将应用分为各个模块。这样，我们就可以将应用逻辑的修改精细化描述，具体到某个模块的修改了。

随后，我们可以将各个模块分类，分为 **无状态模块** 和 **有状态模块**。**无状态模块** 的定义是，该模块的执行结果（输出）仅仅与模块当前输入有关，而与模块的执行历史无关。也就是说，无论模块之前进行了什么操作，都不会影响到现在模块的输出结果。许多数据处理应用的算法都是属于 **无状态模块**。例如：数据清洗操作（将固定格式的字符流数据转化为包含一定数据结构的结构化数据），并行矩阵乘法（在神经网络的高性能计算中属于性能瓶颈之一）

当热部署需求是修改 **无状态模块** 时，热部署的实现将变得相对容易。我们仅仅需要将程序执行的对应模块在合适的时间点进行替换，而不用考虑模块中内部状态的影响。目前解决修改 **无状态模块** 的方案主要是将模块置于动态链接库内加载，如 Windows 操作系统中的`.dll`文件，和 Unix 操作系统的 `.so` 文件。

另一方面，某些模块是拥有内部状态的，这些模块的执行结果除了与当前输入数据有关，还会与模块内部储存的一些状态有关，这些模块我们称它为 **有状态模块**。一个 **有状态模块** 的最简单的例子是一个计数器模块：每次调用该模块，会输出被调用次数。这样，我们尽管每次同样调用同一个模块，它的输出结果却不尽相同（每次输出结果都比上一次的输出结果多1）。

对于这些 **有状态模块** 而言，为了实现热部署的功能，我们并不能像之前提到的 **无状态模块** 一样处理。如果我们简单地像之前那样，通过动态链接库进行替换，那么这些 **有状态模块** 中的状态就会丢失。还是举计数器的例子：当我们需要对计数器进行更新时，如果是简单地替换掉计数器当前的代码，那么计数器记录的次数就会丢失。这对于某些生产环境是不可接受的严重后果，特别是对于一些庞大且重要的系统，如银行，电信运营商，高铁调度系统等。这些系统一旦发生某些状态数据丢失，会造成巨大的经济损失，甚至造成人员伤亡等。

因此，为了避免状态的丢失造成各种数据丢失乃至经济财产损失，我们的一个基本要求就是用户在提供热部署的更新补丁时，这些更新必须是无状态模块。

这些无状态模块在编程语言中主要有四种具体形式：

- 更改函数
- 增加函数
- 更改全局变量
- 增加全局变量

接受更新的方式

用户有两种方式向我们提供更新：

- 用户将更新的源代码发送到我们的应用

用户可以向使用版本管理工具一样将源代码的修正推送到我们的应用中，我们的应用会自动使用文本比较算法比较出需要修改的最小模块，然后我们的工具可以自动提取出这个最小模块的源代码。此时的源代码可能存在一些未定义的外部符号（比如未定义的全局变量，外部函数等等），我们的应用可以通过语法分析等手段搜索到这些外部符号，然后自动加上对应的 **extern** 添加到源代码中。经过这样预处理的源代码再进行编译，就可以获得未经连接的 ELF 格式库文件了。

- 用户将编译完成的更新源代码直接发送给我们

考虑到知识产权保护的需要，一些用户可能不能将源代码对我们开放。因此，我们也支持用户直接发送已编译好，但未链接的 ELF 库文件给我们的应用。

在获取 ELF 文件之后，我们将该 ELF 文件发送到更新器，将文件添加到工作进程的内存空间，完成动态符号链接的工作，并在合适的时间点通过前面的控制权处理程序（“加壳”操作中加入的“壳”程序）接管工作进程的控制流，并修改跳转关系，使得每次调用旧模块时都有相应的跳转指令能重定向到新模块的地址并且执行新模块。这样，一次用户更新就成功部署了。

工作进程控制权的获取

在对一个正在运行的工作进程进行更新时，修改它的代码段是一个不可缺少的操作。很显然，这个操作必须是原子的，因为如果程序执行到了正在被修改的代码，它可能会触发一些不可预知的错误。因此我们需要在适当的时候让工作进程交出控制权，在虚拟机完成对工作进程的更新之后，再将控制权交回给工作进程。

关于获得程序的控制权的问题，我们有以下方案

内核态的解决方案

在内核中，用中断的方式抢占进程的控制权。在内核态对程序执行相关更新操作后，再将控制权交还给程序。这种做法的**优点**在于**安全**和**高效**，因为代码段默认是不可写的，所以我们需要先更改代码段的读写属性，接着修改代码段内容，再把代码段改回不可写的状态。所以，直接在内核态里完成这些工作可以节省一些系统调用的开销，还可以保证修改代码操作的原子性和程序执行的安全性。

用户态的解决方案

- 调试器原理：软中断 (在 x86 上是 INT 3)

  调试器在给一个被调试程序打断点时，先保存断点处的指令，然后用 INT 3 指令去替换原有代码的指令。程序执行到 INT 3 时，操作系统会给调试器发信号。调试器在接收到信号之后会执行以下流程

  - 将 INT 3 改回原来的指令
  - 将代码寻址寄存器回滚一个字节
  - 允许用户和调试器进行交互（读取数据，加新的断点之类）
  - 如果继续执行，调试器会把 INT 3 再填回去，除非用户要求删除断点

  我们可以借鉴调试器的方式，通过软中断获取工作进程的控制权。这种方法的优点是可以**精确**地在某些地址停下来，这对热部署有重要意义。为了更新的安全性，我们通常不希望程序在任意时刻被中止，然后更新。我们希望程序在某些安全的时间节点停下来接收更新，所以软中断的方法可以很好的解决这个问题。软中断的缺点也很明显，它对**效率**的影响非常大。

- 虚拟机接管控制流

  和动态语言类似，我们也可以造一个虚拟机，让每一次函数调用都由虚拟机来处理。这种动态寻址方案优点是**灵活性好**，缺点是**效率不高**，而且**实现比较困难**，尤其是对已经编译好的二进制代码。

  虚拟机和工作进程之间的连接并不是一件轻松的事情，因为工作进程有自己的完整控制流，而且我们假设工作进程并未预先为虚拟机设计通信接口。如果像真正的虚拟机那样对工作进程进行解释，那么它的效率将是我们完全无法忍受的。一个可行的解决方案是，在每次函数调用时，将调用的目标函数地址压栈，并跳转到虚拟机里的控制函数，由虚拟机来处理这次函数调用。

  类似于下面将下面这个程序

```c
#include <stdio.h>

int fun()
{
    return 1;
}

int main()
{
    printf("%d\n",fun());
    return 0;
}
```

变换为

```c
#include <stdio.h>

int vm(int (*pf)())
{
    return pf();
}

int fun()
{
    return 1;
}

int main()
{
    printf("%d\n",vm(fun));
    return 0;
}
```

- 信号处理函数获得控制流

  我们可以给工作进程加壳（即给可执行文件增加一些代码），并且这些代码可以在main函数之前执行。在执行"壳"代码时，可以给工作进程注册一些信号处理函数。当需要获得控制流时，外部程序就给工作进程发信号，这时，进程执行中断，进入信号处理函数，这样就达到了获取控制流的目的。通过信号处理函数来获得控制流的方法优点是**执行效率**比较高，缺点是收到信号时停下来的**断点分布**很随机，不一定是我们想要的点。

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void signal_handle(int sig_num)
{
    if(sig_num == SIGUSR1)
    {
        printf("\nCapture SIGUSR1\n");
    }
```

```
    printf("signal_handle running ...\n");
}

int main(int argc,char **argv)
{
    signal(SIGUSR1, signal_handle);
    while(1)
    {
        sleep(1); putchar('1'); fflush(stdout);
        sleep(1); putchar('2'); fflush(stdout);
        sleep(1); putchar('3'); fflush(stdout);
        sleep(1); putchar('4'); fflush(stdout);
    }
    return 0 ;
}
```

例如上面这段代码，main函数在入口处注册了信号处理函数，然后在主循环部分每隔1s就输出一个数字。在接收到SIGUSR1信号时，主线程的控制流就会交给signal_handle函数，因而我们就获得了工作进程的控制流。但是工作进程刚才是在什么地方停下来的我们并不知情，它很可能停在了我们不希望它停下的位置 。

代码的修改

工作进程代码的修改方式

要实现热更新必须要修改代码，我们已经实现了将可执行文件的代码段的属性修改为可写，那么工作进程代码可能的修改方式有以下几种。

- 整段替换

  一个比较简便的方法是将工作进程的代码段整段替换为新的代码，或者将整段函数替换为新的代码，这样做是**不可行的。**

  首先，工作进程正在执行的指令不确定，如果将代码段整段替换，将引起无法预知的错误。例如，如果进程正在执行函数递归，整段替换代码后进程的执行进入未知状态，无法继续控制，这是绝对不可行的。

  其次，考虑将函数整段替换的方法。即使可以保证替换时机合适，即该函数未在栈中时进行替换，如果新的代码长度超过了原有代码，那么这也是不可行的。其一，原有的可执行文件的代码段可能将无法容纳新的代码，无法做到整段替换。其二，整段替换代码可能需要调整后续代码的内存地址，进而需要调整所有用到直接寻址、寄存器寻址等的指令，还可能需要更新符号表等更多内容。效率低、操作复杂。

- 新增代码

  另一种方法是将新函数作为新增的代码。在原代码调用旧函数时跳转到新的函数入口，函数执行完成后返回到被替换函数返回的地址。这种方法**可行。**

  在这里暂时先不考虑如何实现新增代码能被执行，而更详细的考虑怎么样通过新增代码的方式实现热更新以及这种方法的可行性。

  以下将被替换的函数称为旧函数，新增的函数代码称为新函数。

- 旧函数入口处跳转

  将旧函数函数体的第一条指令改为跳转指令，跳转到新函数的入口地址。这样做会使每次调用函数时，先jump到旧函数的入口地址，再jump到新函数的入口地址，最后返回到相应的位置。这是一个**简单、通用**的做法，但是**效率低**，而且**对cache不友好。**考虑一种情况，如果我们三次更新同一个函数，不妨称之为函数1、2、3，那么每次调用这个函数会从先跳转到旧函数，之后跳转到函数1，之后函数2，最后函数3，执行并返

回，而旧函数和函数1、2、3很可能位于内存不同的页上，这就违反了空间局部性的原理，必定会造成低效、对cache不友好。

- 在函数调用位置跳转

  将跳转到旧函数的指令直接修改为跳转到新函数。要分为以下几种情况。

- 如果跳转指令采用直接寻址的方式，这样只需要将地址改为新函数的地址即可。
- 如果跳转指令采用寄存器寻址、间接寻址等方式，会有些麻烦。出现这种情况可能的原因比如，使用了函数指针进行函数调用，如果汇编指令是 jump register，这时只有在执行到了这条指令时才能确定要跳转的位置，即register的值，是不是旧函数的地址，而在执行之前很难确定，所以很难实现在函数调用位置跳转。

这样做总体来说效率更高，为了避免寄存器寻址引发的麻烦可以与第一种方法"旧函数入口处跳转"一起使用，可以**提高效率。**

- 由虚拟机控制跳转

  对于第一种方式"旧函数入口处跳转"，跳转需要两次，而第二种方法能优化第一种方法，那么就自然想到可以用虚拟机控制跳转，即调用旧函数时先跳转到虚拟机的控制部分，由虚拟机决定是否跳转到新函数还是其他函数，这样也是跳转两次，但是相比第一种方式，虚拟机可以从中**获得控制权**，从而更易控制代码的修改、断点的设置等等；**灵活性更高**，可以方便灵活的切换函数。但是与第一种方式同样，**效率低。**

- 代码全面重排

  为了解决效率问题，还有一种策略就是将代码全面重排。使用前三种方法进行热更新，都是新增代码的方法，新增代码就必然会打破程序的空间局部性。为了使前三种方式能获得更高的效率，考虑可以使用 mmap 系统调用的方法，将代码放入 mmap 的空间中全面重排。按照时间局部性和空间局部性原理，将修改后的代码，对于经常被调用的函数，调用者和被调用函数尽可能放在临近的内存空间中，结合前三种方式实现动态更新。这样做对前三种方式都在一定程度上**提高了效率**，但是非常**繁琐**，程序中所有含有寻址的指令都要进行修改，代价较大，**不易实现。**

代码更新的实现方法

明确了工作进程代码的修改方式，现在研究代码更新的实现方法。根据前面的讨论，整段替换是不可行的，所以这里主要研究新增代码这个方式的实现方法。

新增代码，无论采取哪种新增代码的方式，都有一个问题：新增的代码放在哪里，怎么执行。只是新增代码也不够，必须使旧函数的调用可以跳转到新函数入口，这就涉及另一个问题：旧的代码怎么修改。为了解决这两个问题，需要两个函数：

- mmap() 系统调用

使用mmap()系统调用解决第一个问题 —— 新增代码放在哪里，怎么执行。

```
void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
int munmap(void *addr, size_t length);
```

mmap是一种内存映射文件的方法，即将一个文件或者其它对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对映关系。成功执行时，mmap()返回被映射区的指针.

munmap在进程地址空间中解除一个映射关系，addr是调用mmap()时返回的地址，len是映射区的大小。成功执行时，munmap()返回0。当映射关系解除后，对原来映射地址的访问将导致段错误发生。

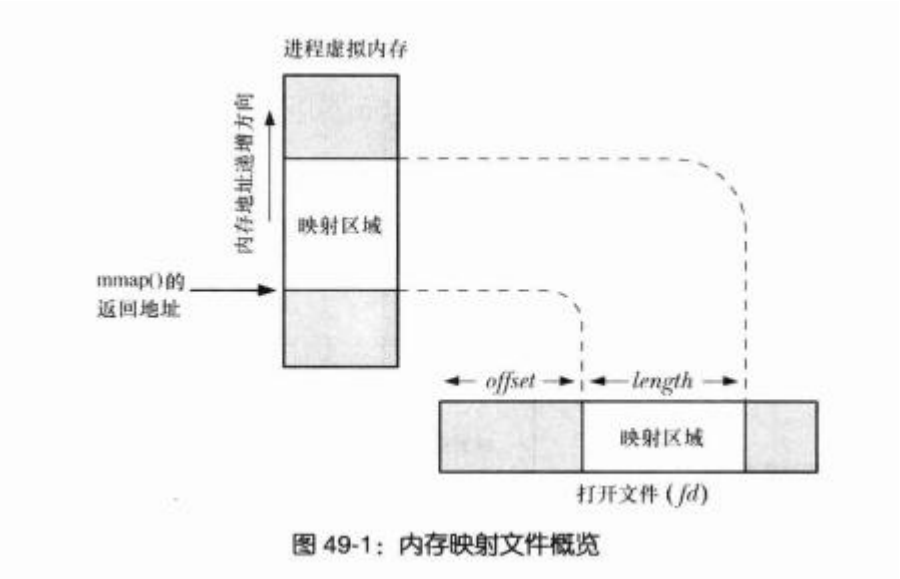
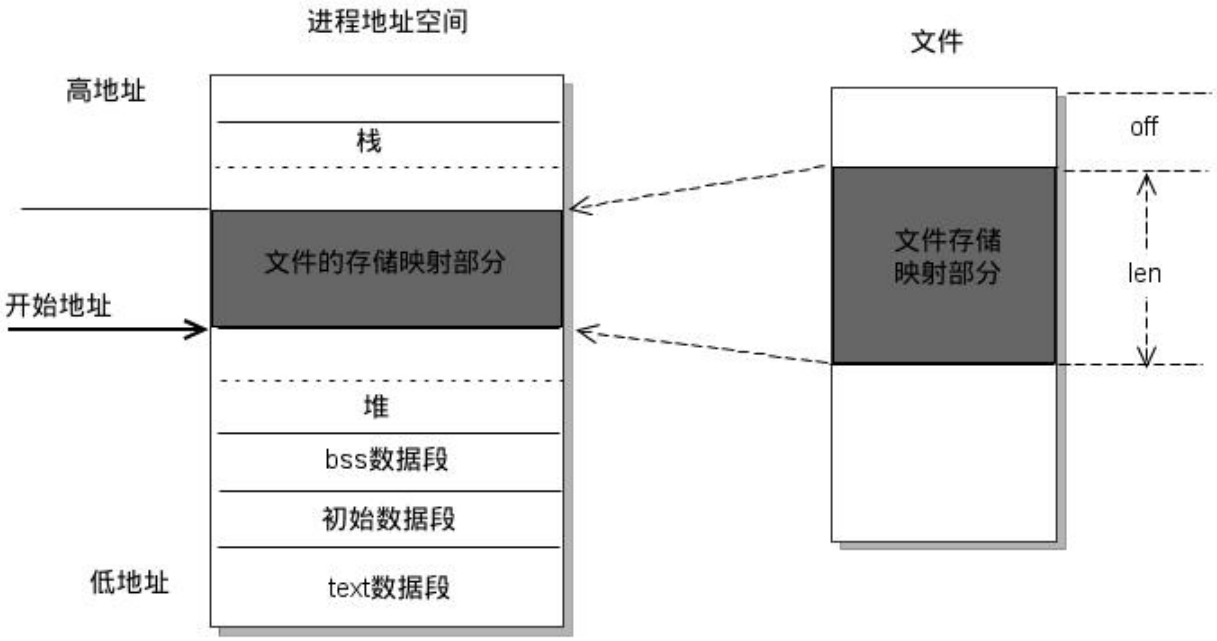


图 49-1：内存映射文件概览



malloc()这些函数也能分配内存空间容纳新增函数代码，使用mmap的优点在于：对于int prot参数，可以指定

PROT_READ：表示内存段内的内容可写； PROT_WRITE：表示内存段内的内容可读； PROT_EXEC：表示内存段中的内容可执行； PROT_NONE：表示内存段中的内容根本没法访问。

使用malloc直接得到的内存无法执行，而使用mmap可以直接获得**可执行的内存区域**，可以指定内存地址（尽管有时不必要这么做）。将代码存入mmap的空间中可以执行，一个demo如下：（在/demo/jit.c中）

```
/*
 * jit
 * http://blog.reverberate.org/2012/12/hello-jit-world-joy-of-simple-jits.html
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>
```

```c
#include <sys/mman.h>

typedef int (*pfun)();

int main(int argc, char *argv[]) {
    // Machine code for:
    //   mov eax, 0
    //   ret
    unsigned char code[] = {0xb8, 0x00, 0x00, 0x00, 0x00, 0xc3};

    if (argc < 2) {
        fprintf(stderr, "Usage: ./a.out <integer>\n");
        return 1;
    }

    // Overwrite immediate value "0" in the instruction
    // with the user's value.  This will make our code:
    //   mov eax, <user's value>
    //   ret
    int num = atoi(argv[1]);
    memcpy(&code[1], &num, 4);

    // Allocate writable/executable memory.
    // Note: real programs should not map memory both writable
    // and executable because it is a security risk.
    void *mem = mmap(NULL, 4096, PROT_WRITE | PROT_EXEC,
                     MAP_ANON | MAP_PRIVATE, -1, 0);
    memcpy(mem, code, sizeof(code));
    mprotect(mem,4096,PROT_EXEC);

    // The function will return the user's value.
    int (*func)() = (pfun)mem;
    printf("%d\n",func());
    return 0;
}
```

在这个demo中，成功将可执行代码写到了mmap的空间中并执行，最后成功返回值并打印在屏幕上。

- mprotect() 系统调用

使用mprotect()系统调用解决第二个问题 —— 旧的代码怎么修改。

```c
int mprotect(const void *start, size_t len, int prot);
```

mprotect()函数把自start开始的、长度为len的内存区的保护属性修改为prot指定的值。prot同样可以取以下几个值，并且可以用“|”将几个属性合起来使用：

PROT_READ：表示内存段内的内容可写； PROT_WRITE：表示内存段内的内容可读； PROT_EXEC：表示内存段中的内容可执行； PROT_NONE：表示内存段中的内容根本没法访问。

其中，指定的内存区间必须包含整个内存页（4K）。区间开始的地址start必须是一个内存页的起始地址，并且区间长度len必须是页大小的整数倍。如果执行成功，则返回0；如果执行失败，则返回-1，并且设置errno变量。

可以看到，使用mprotect即可将工作进程的**代码段设置为可写**，即可实现旧代码的修改。一个小demo如下：（在/demo/change_code.c中）

```
 /*
 * change code segment
 * https://stackoverflow.com/a/12952341
 */
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>
#include <string.h>

int f1()
{
    return 1;
}

int f2()
{
    return 2;
}

int main()
{
    int rc;
    int pagesize;
    char *p;
    int i;

    printf("f1=0x%08X.\n", f1);
    printf("f2=0x%08X.\n", f2);

    printf("f%d\n",f1());
    printf("f%d\n",f2());

    pagesize = sysconf(_SC_PAGE_SIZE);
    printf( "pagesize=%d (0x%08X).\n", pagesize, pagesize );
    if(pagesize == -1) return(2);

    p = (char*) f1;
    printf( "p=0x%08X.\n", p );
    p = (char*) ((size_t) p & ~(pagesize - 1));
    printf( "p=0x%08X.\n", p );

    rc = mprotect( p, pagesize, PROT_READ | PROT_WRITE | PROT_EXEC );
    printf( "rc=%d.\n", rc );
    if(rc!=0) return(2);

    printf("'mprotect()' succeeded.\n");
    memcpy(f2,f1,(size_t)f2-(size_t)f1);

    printf("Write succeeded.\n");
```

```
    printf("f%d\n",f1());
    printf("f%d\n",f2());

    printf("Call succeeded.\n");

    return 0;
}
```

在这个demo中，将一个程序的代码段设置为可写，写入了另一个程序的代码，并重新修改代码段只读，并成功运行了新的程序。

通过这两个系统调用，解决了代码更新的实现方法问题。我们已经有了充分的进程代码的修改方式和方法。

代码优化（可选）

为了实现工作进程的热部署，我们不得不对工作进程的控制流进行一些干扰。这些干扰会导致程序性能的下降。因此，我们应该想办法弥补这些性能损失。

性能分析数据指导的优化(PGO)

PGO(profile-guided optimization) 是一种广泛应用于即时编译（JIT, just-in-time compilation）中的优化技术。它的基本思想是使用运行时数据来指导程序优化。程序在运行时会产生一些数据，如各个函数的调用频率，热点路径等。这些数据在提前编译（ahead-of-time compilation）时是无法获取和利用的。

- 热点路径

  我们可以运行时反复调用的函数放在同一个或相邻的几个页里，这样可以使得CPU缓存命中率大大提高。

- 运行时内联

  通过分析性能数据，决定是否将某些函数在运行时内联。

常量折叠

对某些需要计算的常量，我们可以用计算出来的值去替代计算过程。至于哪些常量是可以被安全地折叠，可能需要人为提供一些配置才能确定。

结论

对于程序热部署这个问题，我们采用了自顶而下的设计思想，将原问题分解成了几个阶段的问题。在上文中，我们对这几个阶段的解决方案进行了详尽地讨论。最后，我们得出了以下几种可行的解决方案组合，作为我们备选的实现方法。

Plan A

- 给工作进程注册信号处理函数
  - 加壳法
  - ptrace 法

- 工作进程/守护进程在接收到更新信号后，为工作进程添加软中断，让进程停在安全的更新点
- 读取 elf 文件提供的更新，将更改的函数、新增的函数和新增的全局变量装入工作进程的地址空间中
- 将程序中对旧函数的调用变为对新函数的调用
    - 函数是位置无关码
        - 没有任何信息时：在旧函数入口处加跳转，跳转到新函数入口
        - 没有通过函数指针调用的情况：在程序中搜索对旧地址的调用指令，替换为对新地址的调用
    - 函数不是位置无关码
        - 新代码比旧代码短且没有正在栈中的递归调用：原地替换代码
- 更新镜像的制作方法
    - 单独用一个文件编写函数，函数中引用到的外部符号用 extern 引入。用编译器对该文件进行编译但不链接。
    - 编写编译器插件
- 性能优化

  几乎不能进行性能优化

Plan B

- 用加壳的方法把虚拟机加到程序镜像中
- 修改工作进程中的函数调用，将寻址的权力交给虚拟机
- 虚拟机可以灵活地进行寻址，也可以灵活地加载函数，重排函数
- 更新镜像的制作方法

  同 Plan A
- 性能优化
    - 热点路径
    - 运行时内联
    - 常量折叠

综合起来看，Plan A 的可行性更高，稳定性更强。在运行效率方面，Plan A 的开销更小，但不能做运行时优化；Plan B 的开销较大，但可以根据运行时信息进行特定的优化。具体哪种方案的性能更好，对于我们来说仍是未知数。不过整体来看，我们更倾向于使用 Plan A。

附录

elf文件格式

- elf文件的分类
    - 可重定位的对象文件(Relocatable file)

      这是由汇编器汇编生成的 .o 文件。后面的链接器拿它作为输入，经链接处理后，生成一个可执行的对象文

件（Executable file）或者一个可被共享的对象文件(Shared object file)

- 可执行的对象文件(Executable file)

  很常见，如文本编辑器vi等等。可执行的脚本(如shell脚本)不是 Executable object file，它们只是文本文件，一般无扩展名。

- 可被共享的对象文件(Shared object file)

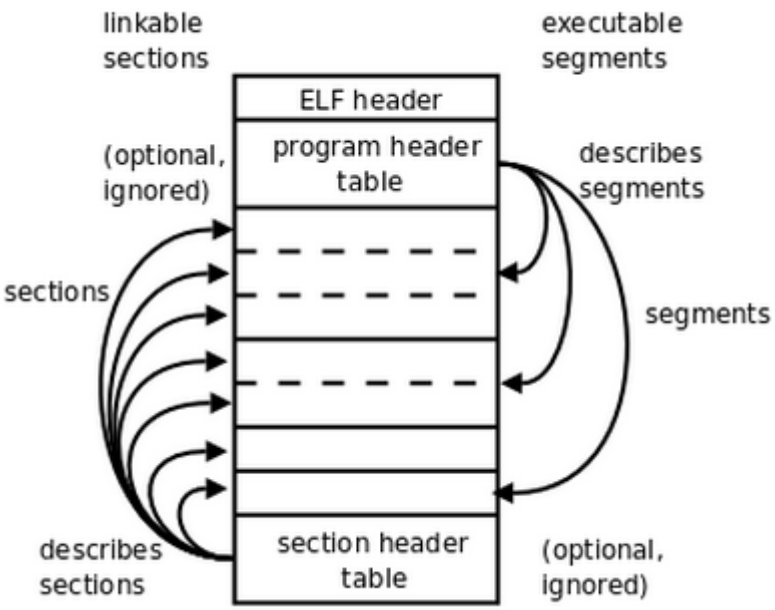
  这就是所谓的动态库文件，也即 .so 文件。动态库在发挥作用的过程中，必须经过两个步骤：
  （1） 链接编辑器拿它和其他Relocatable object file以及shared object file作为输入，经链接处理后，生存另外的 shared object file 或者 executable file。
  （2） 在运行时，动态链接器(dynamic linker)拿它和一个Executable file以及另外一些 Shared object file 来一起处理，在Linux系统里面创建一个进程映像。

- 核心转储文件(Core dump file)

  当进程意外终止时，系统可以将，该进程的地址空间内容及终止时的一些其他信息转储到核心转储文件，比如linux下的core dump

- elf文件的组成

  elf文件大都包含ELF头部、程序头部表、节区或段、节区头部表。 ELF 头部用来描述整个文件的组织。节区部分包含链接视图的指令、数据、符号表、重定位信息等。程序头部表告诉系统如何创建进程映像。节区头部表包含了描述文件节区的信息，如名称、大小。



- 组成不同的可重定位文件（.o 文件）参与可执行文件或者可被共享的对象文件的链接构建。不需要程序头部表。必须包含节区头部表。以节为单位。
- 组成可执行文件或者可被共享的对象文件（.so文件）在运行时内存中进程映像的构建。 必须具有程序头部表。可以无节区头部表。以段为单位。

可重定位文件	可执行或可被共享的对象文件
ELF头部	ELF头部
程序头部表（可无）	程序头部表（必须）
很多节区	很多段
节区头部表（必须）	节区头部表（可无）

- 文件头

  首先，写一段C程序代码

```
#include<stdio.h>

int addTwoIntegers(int firstInFunction, int secondInFunction){
    return firstInFunction + secondInFunction;
}

int main(void){
    int firstInteger, secondInteger, sumInteger;
    firstInteger = 2;
    secondInteger = 3;
    sumInteger = addTwoIntegers(firstInteger, secondInteger);
    printf("sum is %d", sumInteger);
    return 0;
}
```

  使用

```
readelf -h ./a.out
```

  命令可以读取elf文件（编译生成的 a.out）的文件头，内容如下

```
ELF 头:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别:                              ELF64
  数据:                              2 补码，小端序 (little endian)
  版本:                              1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本:                          0
  类型:                              EXEC (可执行文件)
  系统架构:                          Advanced Micro Devices X86-64
  版本:                              0x1
  入口点地址:               0x400430
  程序头起点:          64 (bytes into file)
  Start of section headers:          6656 (bytes into file)

  标志:             0x0
```

```
本头的大小:        64 (字节)
程序头大小:        56 (字节)
Number of program headers:         9
节头大小:          64 (字节)
节头数量:          31
字符串表索引节头:  28
```

- 入口点地址
  是指程序第一条要运行的指令的运行时地址，如这个a.out的进入点是 0x400430。可重定位文件只是供再链接，不会有程序进入点（这里会是0），而可执行文件和动态库都存在进入点。可执行文件的进入点指向C库中的_start，动态库中的进入点指向 call_gmon_start。
- Number of program headers、Number of section headers
  指程序头部表和段表的数量。可重定位文件必须有节区头部表（有section），可执行文件或可共享文件必须有程序头部表（有segments），本头的大小(size of this header)指这个ELF文件头本身的大小，程序头大小指程序头部表的大小，特别地其中字符串索引节头表示字符串表中ELF文件头的字符串所在表中的索引。
- 魔数
  最前面Magic的十六个字节被规定用来标识ELF文件的，操作系统通过这16个字节来达成识别ELF文件的目的，7f对应ASCII码中DEL控制符，随后三个字节分别表示'E'，'L'，'F'对应的ASCII码，之后第五个字节02表示是64位的，第六个字节表示字节序01表示小端序，第七个字节表示ELF文件的主版本号，大多数均为01，这是因为ELF标准自1.2后已经不更新了，后9位ELF标准没有特殊规定，但一些平台上被用作扩展标记，在加载前会先确认魔数是否正确，如果不正确会拒绝加载。
- 其中start of section header和start of program header表示段和程序起始地址

- 节区头部表 使用

```
readelf -S ./a.out
```

命令可以读取elf文件（编译生成的 a.out）的节区头部表 section head table，内容如下 共有 31 个节头，从偏移量 0x1a00 开始：

```
 (摘取了部分)
节头:
  [号] 名称              类型             地址              偏移量
       大小              全体大小         旗标    链接    信息    对齐
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 5] .dynsym           DYNSYM           00000000004002b8  000002b8
       0000000000000060  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000400318  00000318
       000000000000003f  0000000000000000   A       0     0     1
  [23] .got              PROGBITS         0000000000600ff8  00000ff8
       0000000000000008  0000000000000008  WA       0     0     8
  [24] .got.plt          PROGBITS         0000000000601000  00001000
       0000000000000028  0000000000000008  WA       0     0     8
  [25] .data             PROGBITS         0000000000601028  00001028
       0000000000000010  0000000000000000  WA       0     0     8
  [26] .bss              NOBITS           0000000000601038  00001038
       0000000000000008  0000000000000000  WA       0     0     1

  [27] .comment          PROGBITS         0000000000000000  00001038
```

```
       0000000000000034  0000000000000001  MS       0     0     1
  [28] .shstrtab         STRTAB           0000000000000000  000018f4
       000000000000010c  0000000000000000           0     0     1
  [29] .symtab           SYMTAB           0000000000000000  00001070
       0000000000000660  0000000000000018          30    47     8
  [30] .strtab           STRTAB           0000000000000000  000016d0
       0000000000000224  0000000000000000           0     0     1
```

- 偏移量 是该section离开文件头部位置的距离
- 大小 表示section的字节大小
- 全体大小： 只对某些形式的sections 有意义（如符号表 .symtab section），其内部包含了一个表格，这个字段表示表格的每一个条目的特定长度
- 旗标 Key to Flags: W (write), A (alloc), X (execute), M (merge), S (strings), l (large) I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown) O (extra OS processing required) o (OS specific), p (processor specific)
- 一些section
  - .text section
    里面存储的是代码，是可执行的 (旗标为 X)
  - .data section
    里面存放的都是可写的(W)数据(非在堆栈中定义的数据),储存初始化过的数据如定义的赋过初值的全局变量等;
  - .bss section
    存放的数据可写(W)，是未经过初始化的数据。当可执行程序被执行的时候，动态链接器会在内存中开辟一定大小的空间来存放这些未初始化的数据，里面的内存单元都被初始化成0。可执行程序文件中记录有在程序运行时，需要开辟多大的空间来容纳这些未初始化的数据。
  - .symtab和.strtab section
    没有标志A，即 non-Allocable 的 sections，它们只是被链接器、调试器或者其他类似工具所使用的，而并非参与进程的运行。当运行最后的可执行程序时，加载器会加载那些 Allocable 的部分，而 non-Allocable 的部分则会被继续留在可执行文件内。因此这些 non-Allocable 的 section 都可以从最后的可执行文件中删除掉，而照样能够运行，但不可以做调试之类的事情。这两个 section 后面会说明

- .text section 使用

```
objdump -d -j .text ./a.out
```

命令可以读取elf文件（编译生成的 a.out）的.text section，内容如下

```
0000000000400430 <_start>:
      400430:   31 ed                   xor    %ebp,%ebp
      400432:   49 89 d1                mov    %rdx,%r9
      400435:   5e                      pop    %rsi
      400436:   48 89 e2                mov    %rsp,%rdx
      400439:   48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
      40043d:   50                      push   %rax
      40043e:   54                      push   %rsp
      40043f:   49 c7 c0 f0 05 40 00    mov    $0x4005f0,%r8
      400446:   48 c7 c1 80 05 40 00    mov    $0x400580,%rcx
```

```
  40044d:   48 c7 c7 3a 05 40 00    mov    $0x40053a,%rdi
  400454:   e8 b7 ff ff ff          callq  400410 <__libc_start_main@plt>
  400459:   f4                      hlt
  40045a:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

0000000000400460 <deregister_tm_clones>:
  400460:   b8 3f 10 60 00          mov    $0x60103f,%eax
  400465:   55                      push   %rbp
  400466:   48 2d 38 10 60 00       sub    $0x601038,%rax
  40046c:   48 83 f8 0e             cmp    $0xe,%rax
  400470:   48 89 e5                mov    %rsp,%rbp
  400473:   76 1b                   jbe    400490 <deregister_tm_clones+0x30>
  400475:   b8 00 00 00 00          mov    $0x0,%eax
  40047a:   48 85 c0                test   %rax,%rax
  40047d:   74 11                   je     400490 <deregister_tm_clones+0x30>
  40047f:   5d                      pop    %rbp
  400480:   bf 38 10 60 00          mov    $0x601038,%edi
  400485:   ff e0                   jmpq   *%rax
  400487:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
  40048e:   00 00
  400490:   5d                      pop    %rbp
  400491:   c3                      retq
  400492:   0f 1f 40 00             nopl   0x0(%rax)
  400496:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  40049d:   00 00 00

00000000004004a0 <register_tm_clones>:
  4004a0:   be 38 10 60 00          mov    $0x601038,%esi
  4004a5:   55                      push   %rbp
  4004a6:   48 81 ee 38 10 60 00    sub    $0x601038,%rsi
  4004ad:   48 c1 fe 03             sar    $0x3,%rsi
  4004b1:   48 89 e5                mov    %rsp,%rbp
  4004b4:   48 89 f0                mov    %rsi,%rax
  4004b7:   48 c1 e8 3f             shr    $0x3f,%rax
  4004bb:   48 01 c6                add    %rax,%rsi
  4004be:   48 d1 fe                sar    %rsi
  4004c1:   74 15                   je     4004d8 <register_tm_clones+0x38>
  4004c3:   b8 00 00 00 00          mov    $0x0,%eax
  4004c8:   48 85 c0                test   %rax,%rax
  4004cb:   74 0b                   je     4004d8 <register_tm_clones+0x38>
  4004cd:   5d                      pop    %rbp
  4004ce:   bf 38 10 60 00          mov    $0x601038,%edi
  4004d3:   ff e0                   jmpq   *%rax
  4004d5:   0f 1f 00                nopl   (%rax)
  4004d8:   5d                      pop    %rbp
  4004d9:   c3                      retq
  4004da:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

00000000004004e0 <__do_global_dtors_aux>:
  4004e0:   80 3d 51 0b 20 00 00    cmpb   $0x0,0x200b51(%rip)        # 601038
<__TMC_END__>
  4004e7:   75 11                   jne    4004fa <__do_global_dtors_aux+0x1a>

  4004e9:   55                      push   %rbp
```

```
      4004ea:   48 89 e5                mov    %rsp,%rbp
      4004ed:   e8 6e ff ff ff          callq  400460 <deregister_tm_clones>
      4004f2:   5d                      pop    %rbp
      4004f3:   c6 05 3e 0b 20 00 01    movb   $0x1,0x200b3e(%rip)        # 601038
<__TMC_END__>
      4004fa:   f3 c3                   repz retq
      4004fc:   0f 1f 40 00             nopl   0x0(%rax)

    0000000000400500 <frame_dummy>:
      400500:   bf 20 0e 60 00          mov    $0x600e20,%edi
      400505:   48 83 3f 00             cmpq   $0x0,(%rdi)
      400509:   75 05                   jne    400510 <frame_dummy+0x10>
      40050b:   eb 93                   jmp    4004a0 <register_tm_clones>
      40050d:   0f 1f 00                nopl   (%rax)
      400510:   b8 00 00 00 00          mov    $0x0,%eax
      400515:   48 85 c0                test   %rax,%rax
      400518:   74 f1                   je     40050b <frame_dummy+0xb>
      40051a:   55                      push   %rbp
      40051b:   48 89 e5                mov    %rsp,%rbp
      40051e:   ff d0                   callq  *%rax
      400520:   5d                      pop    %rbp
      400521:   e9 7a ff ff ff          jmpq   4004a0 <register_tm_clones>

    0000000000400526 <addTwoIntegers>:
      400526:   55                      push   %rbp
      400527:   48 89 e5                mov    %rsp,%rbp
      40052a:   89 7d fc                mov    %edi,-0x4(%rbp)
      40052d:   89 75 f8                mov    %esi,-0x8(%rbp)
      400530:   8b 55 fc                mov    -0x4(%rbp),%edx
      400533:   8b 45 f8                mov    -0x8(%rbp),%eax
      400536:   01 d0                   add    %edx,%eax
      400538:   5d                      pop    %rbp
      400539:   c3                      retq

    000000000040053a <main>:
      40053a:   55                      push   %rbp
      40053b:   48 89 e5                mov    %rsp,%rbp
      40053e:   48 83 ec 10             sub    $0x10,%rsp
      400542:   c7 45 f4 02 00 00 00    movl   $0x2,-0xc(%rbp)
      400549:   c7 45 f8 03 00 00 00    movl   $0x3,-0x8(%rbp)
      400550:   8b 55 f8                mov    -0x8(%rbp),%edx
      400553:   8b 45 f4                mov    -0xc(%rbp),%eax
      400556:   89 d6                   mov    %edx,%esi
      400558:   89 c7                   mov    %eax,%edi
      40055a:   e8 c7 ff ff ff          callq  400526 <addTwoIntegers>
      40055f:   89 45 fc                mov    %eax,-0x4(%rbp)
      400562:   8b 45 fc                mov    -0x4(%rbp),%eax
      400565:   89 c6                   mov    %eax,%esi
      400567:   bf 04 06 40 00          mov    $0x400604,%edi
      40056c:   b8 00 00 00 00          mov    $0x0,%eax
      400571:   e8 8a fe ff ff          callq  400400 <printf@plt>
      400576:   b8 00 00 00 00          mov    $0x0,%eax

      40057b:   c9                      leaveq
```

```
      40057c:   c3                      retq
      40057d:   0f 1f 00                nopl   (%rax)

    0000000000400580 <__libc_csu_init>:
      400580:   41 57                   push   %r15
      400582:   41 56                   push   %r14
      400584:   41 89 ff                mov    %edi,%r15d
      400587:   41 55                   push   %r13
      400589:   41 54                   push   %r12
      40058b:   4c 8d 25 7e 08 20 00    lea    0x20087e(%rip),%r12        # 600e10
<__frame_dummy_init_array_entry>
      400592:   55                      push   %rbp
      400593:   48 8d 2d 7e 08 20 00    lea    0x20087e(%rip),%rbp        # 600e18
<__init_array_end>
      40059a:   53                      push   %rbx
      40059b:   49 89 f6                mov    %rsi,%r14
      40059e:   49 89 d5                mov    %rdx,%r13
      4005a1:   4c 29 e5                sub    %r12,%rbp
      4005a4:   48 83 ec 08             sub    $0x8,%rsp
      4005a8:   48 c1 fd 03             sar    $0x3,%rbp
      4005ac:   e8 17 fe ff ff          callq  4003c8 <_init>
      4005b1:   48 85 ed                test   %rbp,%rbp
      4005b4:   74 20                   je     4005d6 <__libc_csu_init+0x56>
      4005b6:   31 db                   xor    %ebx,%ebx
      4005b8:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
      4005bf:   00
      4005c0:   4c 89 ea                mov    %r13,%rdx
      4005c3:   4c 89 f6                mov    %r14,%rsi
      4005c6:   44 89 ff                mov    %r15d,%edi
      4005c9:   41 ff 14 dc             callq  *(%r12,%rbx,8)
      4005cd:   48 83 c3 01             add    $0x1,%rbx
      4005d1:   48 39 eb                cmp    %rbp,%rbx
      4005d4:   75 ea                   jne    4005c0 <__libc_csu_init+0x40>
      4005d6:   48 83 c4 08             add    $0x8,%rsp
      4005da:   5b                      pop    %rbx
      4005db:   5d                      pop    %rbp
      4005dc:   41 5c                   pop    %r12
      4005de:   41 5d                   pop    %r13
      4005e0:   41 5e                   pop    %r14
      4005e2:   41 5f                   pop    %r15
      4005e4:   c3                      retq
      4005e5:   90                      nop
      4005e6:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
      4005ed:   00 00 00

    00000000004005f0 <__libc_csu_fini>:
      4005f0:   f3 c3                   repz retq
```

指令对.text section 内容进行反汇编，也就是由机器码出发以x86汇编的形式显示具体内容，可以看到函数名addTwoIntegers，该位置的汇编指令为 callq 400526

- .data section 使用

```
objdump -d -j .data  ./a.out
```

命令可以读取elf文件（编译生成的 a.out）的.data section，内容如下

```
Disassembly of section .data:

0000000000601028 <__data_start>:
    ...

0000000000601030 <__dso_handle>:
    ...
```

可以看到没有什么数据

- .strtab section 使用

```
readelf -x 30 ./a.out
```

命令可以读取elf文件的.strtab section，其中30是 .strtab section 在SHT表格中的索引值（在前面能看到）。内容如下

```
  0x00000000 00637274 73747566 662e6300 5f5f4a43 .crtstuff.c.__JC
  0x00000010 525f4c49 53545f5f 00646572 65676973 R_LIST__.deregis
  0x00000020 7465725f 746d5f63 6c6f6e65 73005f5f ter_tm_clones.__
  0x00000030 646f5f67 6c6f6261 6c5f6474 6f72735f do_global_dtors_
  0x00000040 61757800 636f6d70 6c657465 642e3735 aux.completed.75
  0x00000050 3835005f 5f646f5f 676c6f62 616c5f64 85.__do_global_d
  0x00000060 746f7273 5f617578 5f66696e 695f6172 tors_aux_fini_ar
  0x00000070 7261795f 656e7472 79006672 616d655f ray_entry.frame_
  0x00000080 64756d6d 79005f5f 6672616d 655f6475 dummy.__frame_du
  0x00000090 6d6d795f 696e6974 5f617272 61795f65 mmy_init_array_e
  0x000000a0 6e747279 00436164 642e6300 5f5f4652 ntry.Cadd.c.__FR
  0x000000b0 414d455f 454e445f 5f005f5f 4a43525f AME_END__.__JCR_
  0x000000c0 454e445f 5f005f5f 696e6974 5f617272 END__.__init_arr
  0x000000d0 61795f65 6e64005f 44594e41 4d494300 ay_end._DYNAMIC.
  0x000000e0 5f5f696e 69745f61 72726179 5f737461 __init_array_sta
  0x000000f0 7274005f 5f474e55 5f45485f 4652414d rt.__GNU_EH_FRAM
  0x00000100 455f4844 52005f47 4c4f4241 4c5f4f46 E_HDR._GLOBAL_OF
  0x00000110 46534554 5f544142 4c455f00 5f5f6c69 FSET_TABLE_.__li
  0x00000120 62635f63 73755f66 696e6900 5f49544d bc_csu_fini._ITM
  0x00000130 5f646572 65676973 74657254 4d436c6f _deregisterTMClo
  0x00000140 6e655461 626c6500 5f656461 74610070 neTable._edata.p
  0x00000150 72696e74 66404047 4c494243 5f322e32 rintf@@GLIBC_2.2
  0x00000160 2e350061 64645477 6f496e74 65676572 .5.addTwoInteger
  0x00000170 73005f5f 6c696263 5f737461 72745f6d s.__libc_start_m
  0x00000180 61696e40 40474c49 42435f32 2e322e35 ain@@GLIBC_2.2.5
  0x00000190 005f5f64 6174615f 73746172 74005f5f .__data_start.__
  0x000001a0 676d6f6e 5f737461 72745f5f 005f5f64 gmon_start__.__d
  0x000001b0 736f5f68 616e646c 65005f49 4f5f7374 so_handle._IO_st
  0x000001c0 64696e5f 75736564 005f5f6c 6962635f din_used.__libc_
```

```
0x000001d0 6373755f 696e6974 005f5f62 73735f73 csu_init.__bss_s
0x000001e0 74617274 006d6169 6e005f4a 765f5265 tart.main._Jv_Re
0x000001f0 67697374 6572436c 61737365 73005f5f gisterClasses.__
0x00000200 544d435f 454e445f 5f005f49 544d5f72 TMC_END__._ITM_r
0x00000210 65676973 74657254 4d436c6f 6e655461 egisterTMCloneTa
0x00000220 626c6500                           ble.
```

- 上面结果中的十六进制数据部分从右到左看是地址递增的方向，而字符内容部分从左到右看是地址递增的方向。
- 可以在字符串表里找到函数名 addTwoIntegers
- 符号表 .symtab section 中，有一个条目是用来描述符号 addTwoIntegers的，那么在该条目中就会有一个字段(st_name)记录着字符串 addTwoIntegers 在 .strtab section中的索引。
- .shstrtab 也是字符串表，只不过其中存储的是 section 的名字，而非所函数或者变量的名称。
- 字符串表在真正链接和生成进程映像过程中是不需要使用的，但是对调试程序就特别有帮助，前面使用objdump来反汇编.text section 的时候，之所以能看到定义了函数 addTwoIntegers，是因为存在这个字符串表的原因。符号表 .symtab section 在其中作为中介，也起到关键作用。

- .symtab section

  使用

```
readelf -s ./a.out
```

  命令可以读取elf文件的.symtab section。内容如下

```
 （节选了部分）
Symbol table '.symtab' contains 68 entries:
 Num:    Value          Size Type    Bind   Vis      Ndx Name
   0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
   1: 0000000000400238     0 SECTION LOCAL  DEFAULT    1
  49: 0000000000601028     0 NOTYPE  WEAK   DEFAULT   25 data_start
  50: 0000000000601038     0 NOTYPE  GLOBAL DEFAULT   25 _edata
  51: 00000000004005f4     0 FUNC    GLOBAL DEFAULT   15 _fini
  52: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
  53: 0000000000400526    20 FUNC    GLOBAL DEFAULT   14 addTwoIntegers
  54: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
  55: 0000000000601028     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
  56: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
  57: 0000000000601030     0 OBJECT  GLOBAL HIDDEN    25 __dso_handle
  58: 0000000000400600     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
  59: 0000000000400580   101 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
  60: 0000000000601040     0 NOTYPE  GLOBAL DEFAULT   26 _end
  61: 0000000000400430    42 FUNC    GLOBAL DEFAULT   14 _start
  62: 0000000000601038     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
  63: 000000000040053a    67 FUNC    GLOBAL DEFAULT   14 main
  64: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
  65: 0000000000601038     0 OBJECT  GLOBAL HIDDEN    25 __TMC_END__
  66: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
  67: 00000000004003c8     0 FUNC    GLOBAL DEFAULT   11 _init
```

  - FUNC类型表示函数，可以找到自己的函数addTwoIntegers，还可以看到它在字符串表中的偏移

- OBJECT 表示和该符号对应的是一个数据对象，比方程序中定义过的变量、数组等，
- SECTION 表示该符号和一个 section 相关，这种符号用于重定位。
- 在可执行文件或者动态库中，符号的值表示的是运行时的内存地址