29 | C 程序的入口真的是 main 函数吗？

2022-03-02 于航

《深入C语言和程序运行原理》 课程介绍 >

讲述：于航

时长 09:43　大小 8.91M

你好，我是于航。

“main 函数是所有 C 程序的起始入口”，相信对于这句话，每个同学在刚开始学习 C 语言时都很熟悉，因为这是一个被各种教材反复强调的“结论”。但事实真是如此吗？

实际上，这句话对，但也不完全对。在一段 C 代码中定义的 main 函数总是会被优先执行，这是我们在日常 C 应用开发过程中都能够轻易观察到的现象。不过，如果将目光移到那些无法直接通过 C 代码触达的地方，你会发现 C 程序的执行流程并非这样简单。

接下来，我们先通过一个简单的例子，来看看在机器指令层面，程序究竟是如何执行的。

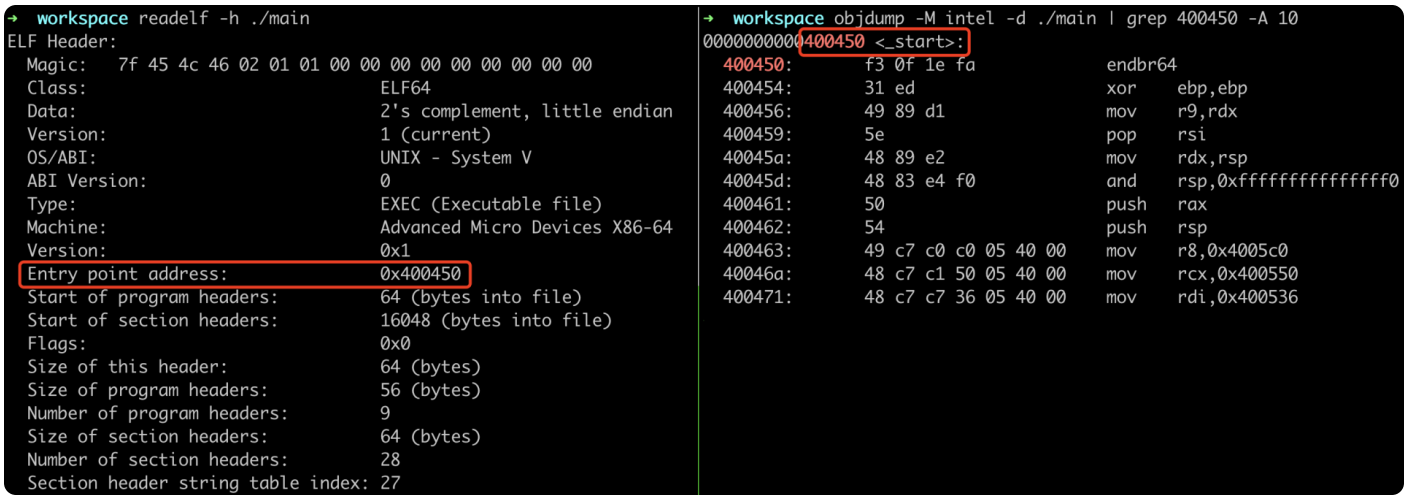
真正的入口函数

这里，我们首先在 Linux 系统中使用命令 “gcc main.c -o main” ，来将如下所示的这段代码，编译成对应的 ELF 二进制可执行文件。

```
// main.c
int main(void) {
  return 0;
}
```

在上述代码中，由于没有使用到任何由其他共享库提供的接口，因此，操作系统内核在将其对应的程序装载到内存后，会直接执行它在 ELF 头中指定的入口地址上的指令。紧接着，使用 readelf 命令，我们可以获得这个地址。然后，通过 objdump 命令，我们可以得到这个地址对应的具体机器指令。

我将这两个命令的详细输出结果放在了一起，以方便你观察，如下图所示：



可以看到，程序并没有直接跳转到 main 函数中执行。相反，它首先执行了符号 _start 中的代码。那么，这个符号从何而来？它有什么作用？相信只要弄清楚这两个问题，你就能够知道 main 函数究竟是如何被调用的。下面让我们详细看看。

_start 从何而来？

实际上，_start 这个标记本身并没有任何特殊含义，它只是一个人们约定好的，长久以来一直被用于指代程序入口的名字。

通常来说，_start 被更多地用在类 Unix 系统中，它是链接器在生成目标可执行文件时，会默认使用的一个符号名称。链接器在链接过程中，会在全局符号表中找到该符号，并将其虚拟地址直接存放到所生成的可执行文件里。具体来说，它会将这个值拷贝至 ELF 头的 e_entry 字段中。

而这一点，也能够在各个链接器的默认配置中得到验证。比如，通过命令 “ld --verbose”，我们便能够打印出 GNU 链接器所使用的链接控制脚本的默认配置。在下面的图片中，命令语句 “ENTRY(_start)” 便用于指定**其输出的可执行文件在运行后，第一条待执行指令的位置**，这里也就是符号 _start 对应的地址。

```
➜  workspace ld --verbose | grep "_start" -A 10
ENTRY(_start)
SEARCH_DIR("=/usr/x86_64-redhat-linux/lib64"); SEARCH_DIR("=/usr/lib64"); SEAR
/lib"); SEARCH_DIR("=/usr/local/lib"); SEARCH_DIR("=/lib"); SEARCH_DIR("=/usr/
SECTIONS
{
```

既然链接器控制着程序执行入口的具体选择，我们便同样可以对此进行修改。比如，对于 GCC 来说，参数 “-e” 可用于为链接器指定其他符号，以作为其输出程序的执行入口。

至此，我们已经知道了 _start 这个标记的具体由来。但是在程序对应的 C 代码，以及编译命令中，我们都没有引入同名的函数实现。那么，它所对应的实际机器代码从何而来呢？

通过在编译时为编译器添加额外的 “-v” 参数，你可能会有新的发现。该参数可以让 GCC 在编译时，将更多与编译过程紧密相关的信息（如环境变量配置、执行的具体指令等）打印出来。这里，我截取了其中的关键一段，如下图所示：

```
 /usr/libexec/gcc/x86_64-redhat-linux/8/collect2 -plugin /usr/libexec/gcc/x86_64-redhat-linux/8/
liblto_plugin.so -plugin-opt=/usr/libexec/gcc/x86_64-redhat-linux/8/lto-wrapper -plugin-opt=-fre
solution=/tmp/ccb2YAFL.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -pl
ugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s --b
uild-id --no-add-needed --eh-frame-hdr --hash-style=gnu -m elf_x86_64 -dynamic-linker /lib64/ld-
linux-x86-64.so.2 -o main /usr/lib/gcc/x86_64-redhat-linux/8/../../../../lib64/crt1.o /usr/lib/g
cc/x86_64-redhat-linux/8/../../../../lib64/crti.o /usr/lib/gcc/x86_64-redhat-linux/8/crtbegin.o
-L/usr/lib/gcc/x86_64-redhat-linux/8 -L/usr/lib/gcc/x86_64-redhat-linux/8/../../../../lib64 -L/l
ib/../lib64 -L/usr/lib/../lib64 -L/usr/lib/gcc/x86_64-redhat-linux/8/../../.. /tmp/ccQVH8lM.o -l
gcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc
/x86_64-redhat-linux/8/crtend.o /usr/lib/gcc/x86_64-redhat-linux/8/../../../../lib64/crtn.o
```


实际上，GCC 在内部会使用名为 “collect2” 的工具来完成与链接相关的任务。该工具基于 ld 封装，只是它在真正调用 ld 之前，还会执行一些其他的必要步骤。可以看到，在实际生成二

进制可执行文件的过程中，collect2 还会为应用程序链接多个其他的对象文件。而 _start 符号的具体定义，便来自于其中的 crt1.o 文件。

_start 有何作用？

crt1.o 是由 C 运行时库（C Runtime Library，CRT）提供的一个用于辅助应用程序正常运行的特殊对象文件，该文件在其内部定义了符号 _start 对应的具体实现。

接下来，我们以 GNU 的 C 运行时库 glibc 为例（版本对应于 Commit ID 581c785），来看看它是如何为 X86-64 平台实现 _start 的。在下面的代码中，我为一些关键步骤添加了对应的注释信息，你可以先快速浏览一遍，以对它的整体功能有一个简单了解。当然，你也可以点击这个链接来获取它的原始版本。

复制代码

```
#include <sysdep.h>

ENTRY (_start)
  cfi_undefined (rip)
  xorl %ebp, %ebp  /* 复位 ebp */
  mov %RDX_LP, %R9_LP   /* 保存 FINI 函数的地址到 r9 */
#ifdef __ILP32__
  /* 模拟 ILP32 模型下的栈操作，将位于栈顶的 argc 放入 rsi */
  mov (%rsp), %esi
  add $4, %esp  /* 同时让栈顶向高地址移动 4 字节 */
#else
  popq %rsi  /* 将位于栈顶的 argc 放入 rsi */
#endif
  mov %RSP_LP, %RDX_LP  /* 将 argv 放入 rdx */
  and $~15, %RSP_LP  /* 对齐栈到 16 字节 */
  pushq %rax  /* 将 rax 的值存入栈中，以用于在函数调用前保持对齐状态 */
  pushq %rsp  /* 将当前栈顶地址存入栈中 */

  xorl %r8d, %r8d  /* 复位 r8 */
  xorl %ecx, %ecx  /* 复位 ecx */
#ifdef PIC
  /* 将 GOT 表项中的 main 函数地址存放到 rdi */
  mov main@GOTPCREL(%rip), %RDI_LP
#else
  mov $main, %RDI_LP  /* 将 main 函数的绝对地址存放到 rdi */
#endif
  /* 调用 __libc_start_main 函数 */
  call *__libc_start_main@GOTPCREL(%rip)
  hlt
END (_start)
  .data
  .globl __data_start
```

```
__data_start:
  .long 0
  .weak data_start
  data_start = __data_start
```

总的来看，这部分汇编代码主要完成了相应的参数准备工作，以及对函数 __libc_start_main 的调用过程。这个函数的原型如下所示：

复制代码

```
int __libc_start_main(int (*main) (int, char**, char**),
                      int argc,
                      char **argv,
                      void (*init) (void),
                      void (*fini) (void),
                      void (*rtld_fini) (void),
                      void *stack_end);
```

该函数一共接收 7 个参数。接下来，让我们分别看看其中每个参数的具体准备过程。

- 第一个参数为用户代码中定义的 main 函数的地址。在汇编代码的第 21~26 行，根据宏 PIC 是否定义，程序将选择性地使用 GOT 表项中存放的 main 函数地址，或是 main 符号的绝对地址，并将它放入寄存器 rdi。
- 第二个参数为 argc。在汇编代码的第 7~13 行，根据宏 **ILP32** 是否定义，程序将选择性地按照不同的数据模型方式，操纵位于栈顶的 argc 参数的值。
- 第三个参数为 argv。在汇编代码的第 14 行，程序直接通过 mov 指令，将它的值（即此刻栈顶地址）放入了寄存器 rdx。
- 第四、五个参数为当前程序的“构造函数”与“析构函数”。从 ELF 标准中可以得知，在动态链接器处理完符号重定位后，每一个程序都有机会在 main 函数被调用前，去执行一些必要的初始化代码。类似地，它们也可以在 main 函数返回后，进程完全结束之前，执行相应的终止代码。而新版本的 glibc 为了修复 “ROP 攻击” 漏洞，优化了这部分实现。因此，这里对应的两个参数只需传递 0 即可。更多信息你可以参考这个链接。
- 第六个参数为用于共享库的终止函数的地址，该地址会在 _start 的代码执行前，被默认存放在 rdx 寄存器中。因此，这里在汇编代码的第 6 行，rdx 寄存器的值被直接拷贝到了 r9 中。

- 第七个参数为当前栈顶的指针，即 rsp 的值。这里在汇编代码的第 17 行，程序将这个值通过栈进行了传递。

这样，__libc_start_main 的调用参数便准备完毕了。在汇编代码的 28 行，我们对它进行了调用。

__libc_start_main 在其内部，会为用户代码的执行，进行一系列前期准备工作，其中包括但不限于以下这些内容：

- 执行针对用户 ID 的必要安全性检查；
- 初始化线程子系统；
- 注册 rtld_fini 函数，以便在动态共享对象退出（或卸载）时释放资源；
- 注册 fini 处理程序，以便在程序退出时执行；
- 调用初始化函数 init；
- 使用适当参数调用 main 函数；
- 使用 main 函数的返回值调用 exit 函数。

可以看到，一个二进制可执行文件的实际运行过程十分复杂，应用程序代码在被执行前，操作系统需要为其准备 main 函数调用依赖的相关参数，并同时完成全局资源的初始化工作。而在程序退出前，这些全局资源也需要被正确清理。

什么是 CRT？

到这里，我们已经把 _start 的由来和作用这两个关键问题弄清楚了，我想你已经知道了 main 函数究竟是如何被调用的。最后我们再来看一个问题：在上面我提到了 C 运行时库，即 CRT，那么它究竟是什么呢？

实际上，CRT 为应用程序提供了对启动与退出、C 标准库函数、IO、堆、C 语言特殊实现、调试等多方面功能的实现和支持。CRT 的实现是平台相关的，它与具体操作系统结合得非常紧密。

当然，真正参与到 CRT 功能实现的并不只有 crt1.o 这一个对象文件。通过观察我之前介绍 collect2 程序调用时给出的参数截图，你会发现与程序代码一同编译的还有其他几个对象文

件。这里我将它们的名称与主要作用整理如下：

- crt1.o，提供了 _start 符号的具体实现，它仅参与可执行文件的编译过程；
- crti.o 和 crtn.o，两者通过共同协作，为共享对象提供了可以使用“构造函数”与“析构函数”的能力；
- crtbegin.o 和 crtend.o，分别提供了上述“构造函数”与“析构函数”中的具体代码实现。

到这里，对于“C 程序的入口真的是 main 函数吗”这个问题，相信你已经有了答案。虽然在这一讲中，我主要以 Linux 下的程序执行过程为例进行了简单介绍，但我想让你了解的并不是这其中的许多技术细节，而是“操作系统在真正执行 main 函数前，实际上会帮助我们提前进行很多准备工作”这个事实。这些工作都为应用程序的正常运行提供了保障。

总结

这一讲，我从“C 程序的入口真的是 main 函数吗”这个问题入手，围绕它带你进行了一系列的实践与研究。

通过观察 Linux 系统下程序的运行步骤，我们可以发现，程序在执行时的第一行指令并非位于 main 函数中。相对地，通过首先执行 _start 符号下的代码，操作系统可以完成执行应用程序代码前的准备工作，这些工作包括堆的初始化、全局变量的构造、IO 初始化等一系列重要步骤。随着这些重要工作的推进，用户定义的 main 函数将会在 __libc_start_main 函数的内部被实际调用。

而上述提到的所有这些重要工作，都是由名为 CRT 的系统环境为我们完成的。它在支持应用程序正常运行的过程中，扮演着不可或缺的角色。

思考题

你知道当我们在 Linux 的 Shell 中运行程序时，操作系统是怎样对程序进行处理的吗？请试着查阅资料，并在评论区告诉我你的理解。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

上一篇　28 | 程序可以在运行时进行链接吗？

下一篇　30 | ABI 与 API 究竟有什么区别？

精选留言 (1)

写留言

一个工匠
2022-03-30

操作系统对 Shell 的执行，是靠 Shell 解释器完成的。在操作系统运行后，Shell 解释器本身就

加载并运行了。其中如 pwd,cd 这些是内部命令，本质是函数调用，可以直接使用。ls 这些是外部命令，需要 fork 一个新进程执行当前命令。一个shell脚本，有很多个这些内外部命令组成，通过 shell 解释器逐行解释完毕后执行。shell 解释器也是一个应用程序，本质是一个 C 程序，不过在该程序中，手动模拟了函数调用栈，和 JVM 有相似之处。所以 shell 解释器，也有静态库/动态库/静态链接/动态链接这些，为 shell 命令的执行保障护航。

作者回复: 回答的很赞！