编译与链接

> 参考资料除了给出的链接、以及自己的知识拓展之外，其余是：
>
> - 回忆本科的《编译原理》课程
> - 《精通正则表达式》
> - 《程序员的自我修养——链接、装载与库》

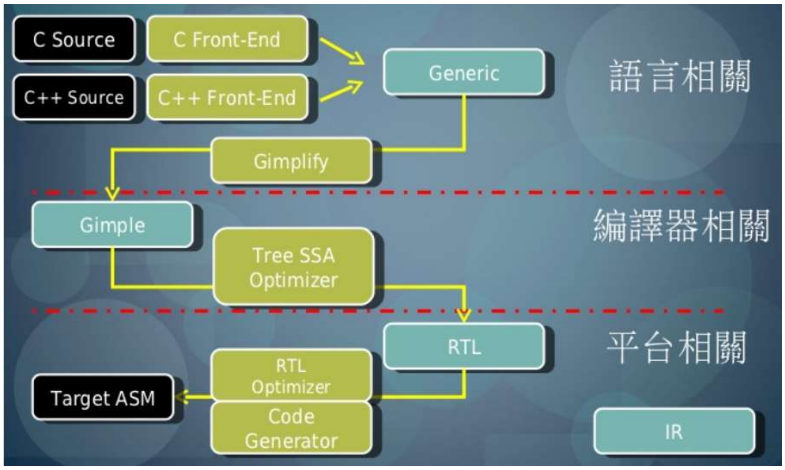
Overview

借用龙书中的两幅图：

【图：一个语言处理系统】

【图：一个编译器的各个步骤】

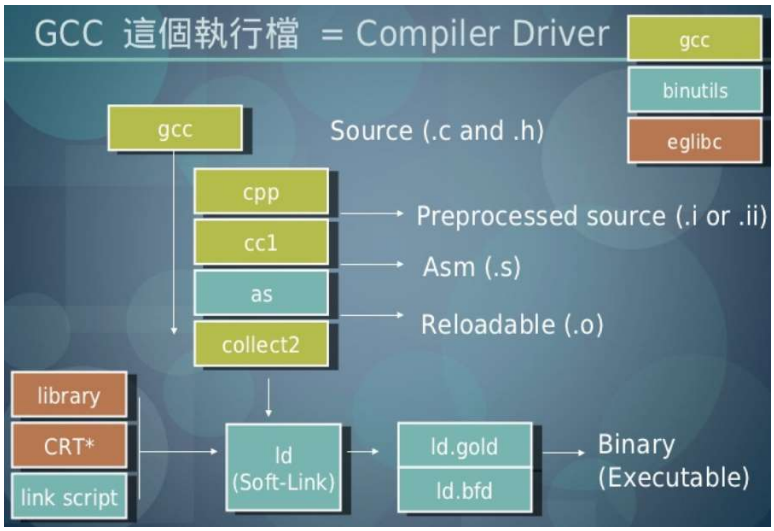
这是教科书上的理论，实际的实现，比如GCC，与理论并不完全一样。GCC支持多种语言（前端，font-end）与多种处理器（后端，back-end），通过在前后端之间引入RTL（Register Transfer Language）中间语言，进行编译器相关优化，这种优化与语言或硬件平台无关。[1]



【图：GCC的最终解决之道（GCC-4以后）】

> Note：IR —— Intermediate Representation

[1]: https://www.slideshare.net/jserv/from-source-to-binary-how-gnu-toolchain-works



【图：GCC编译C源码过程】

编译器前端

> 在大学的编译原理这门课中，涵盖了parser与code generator两部分，要求实现一个SQL解释器（这是我大学中自认为做得最棒的大作业）。时隔一年半，印象比较深的只有词法分析与语法分析两部分了，主要原因还是因为经常用到这两部分。

词法分析

词法分析讲的是**有限状态自动机**（Finite-State Automaton，FSA）[2]，属于三型文法，又叫做正则文法（又译正规文法）[3]，分为两类：非确定有限状态自动机（Nondeterministic Finite Automaton，NFA）[4]，确定有限状态自动机（Deterministic Finite Automaton，DFA）[5]。可以通过Thompson构造法[6]将一个正则表达式（又译正规表达式，Regular Expression）转换成一个与之等价的NFA，再通过幂集构造和最小化过程得到一个对应的最简的DFA。

有限状态自动机的一个很重要用处是**正则表达式**。大多数系统都提供了少量的附加特殊字符，比如`*.txt`，此类文件名（称为"文件群组" file globs，或者"通配符" wildcards）的表达能力有限，正则表达式具有更强的描述能力。

正则表达式有许多流派（flavor），不同的流派所支持的元字符，以及这些元字符的意义并不一样。此外，在特定的宿主语言或工具软件中，正则表达式与正则表达式的交互方式、正则表达式引擎如何将表达式应用到文本也有区别。正则表达式的实现可以分为基本不同的两大类：NFA与NFA，加上POSIX标准，又可以分为三类：DFA（符合或不符合POSIX标准的都属于此类）、传统型NFA、POSIX NFA。

具体的内容太多且杂，详细了解看《精通正则表达式》吧。

> 理论上，NFA和DFA引擎应该匹配完全一样的文本，提供完全一样的功能。但是在实际中，因为人们需要更强的功能，更具表达能力的正则表达式，它的语义发生了变化。
> ——《精通正则表达式》，第180页

[2]: https://en.wikipedia.org/wiki/Finite-state_machine
[3]: https://en.wikipedia.org/wiki/Regular_grammar
[4]: https://en.wikipedia.org/wiki/Nondeterministic_finite_automaton
[5]: https://en.wikipedia.org/wiki/Deterministic_finite_automaton
[6]: https://en.wikipedia.org/wiki/Thompson%27s_construction
[7]: https://en.wikipedia.org/wiki/Glob_(programming)
[8]: 《精通正则表达式》

上下文无关文法

语法分析大都是在讲述**上下文无关文法**（Context-Free Grammar，CFG）[9]，属于二型文法，可通过下推自动机（Pushdown Automaton）[10]实现，它是使用了包含栈的有限自动机。CFG可使用巴科斯范式（Backus Normal Form，BNF）[11]表示，一些语法的描述[12]即采用此方法。

> C语言的语法并非CFG，比如`T *p`，不通过上下文无法确定是声明一个指针，还是两数相乘。[13]

至于CFG中的LL文法[14]、LR文法[15]，以及衍生出来的各类分析器：LL(k)、SLR(k)、LALR(k)、LR(k)。除了LL(k)是自顶向下的分析器外，其余皆是自下而上的分析器；LL、LR中第一个L代表从左到右处理，第二个L（或R）代表对句型执行最左推导（或最右推导），SLR中的S代表Simple，而LALR的LA代表Look Ahead；k代表向前看k个字符。至于具体的算法与区别，忘光了……

[9]: https://en.wikipedia.org/wiki/Context-free_grammar
[10]: https://en.wikipedia.org/wiki/Pushdown_automaton
[11]: https://de.wikipedia.org/wiki/Backus-Naur-Form
[12]: http://faculty.salina.k-state.edu/tim/unix_sg/nonprogrammers/man.html#understanding-a-man-page
[13]: https://eli.thegreenplace.net/2007/11/24/the-context-sensitivity-of-cs-grammar
[14]: https://en.wikipedia.org/wiki/LL_parser
[15]: https://en.wikipedia.org/wiki/LR_parser

乔姆斯基谱系

乔姆斯基谱系是计算机科学中刻画形式文法表达能力的一个分类谱系。对于n型文法的定义，参考[17]：

文法	语言	自动机	产生式规则
0-型	递归可枚举语言	图灵机	α -> β（无限制）
1-型	上下文相关语言	线性有界非确定图灵机	αAβ -> αγβ
2-型	上下文无关语言	非确定下推自动机	A -> γ
3-型	正规语言	有限状态自动机	A -> aB A -> a

[17]: https://en.wikipedia.org/wiki/Chomsky_hierarchy

可执行文件

分类

编译之后生成目标文件（object file），当前存在多种可执行文件格式（Executable），比如Windows下的PE（Portable Executable）和Linux下的ELF（Executable Linkable Format）。
PE与ELF的类型可分为：

- 可重定位文件（Relocatable File）：用于静态链接或生成动态链接库
  - Linux的`.o`/`.a`
  - Window的`.lib`
- 可执行文件（Executable File）：包含了可直接执行的程序
  - 比如Linux的`/bin/bash`（无后缀）
  - Windows的`.exe`
- 共享目标文件（Shared Object File）：动态链接库
  - Linux的`.so`
  - Windows的`.dll`
- 核心转储文件（Core Dump File）[18]：操作系统在进程收到某些信号终止时，将此时进程地址空间的内容以及有关进程状态的其他信息写出一个磁盘文件，这种信息往往用于调试
  - Linux下的core dump
  - Windows下的user dump

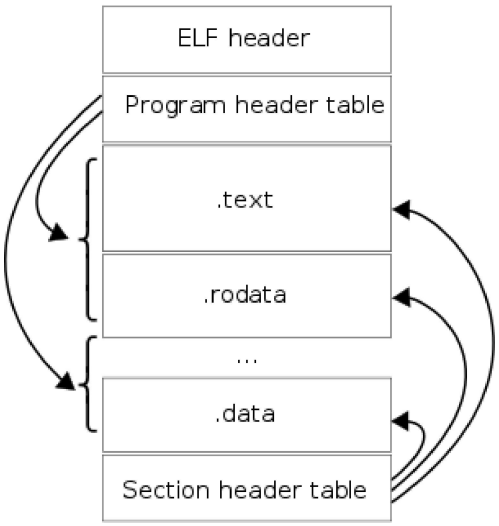
> Note：
> Linux下可执行文件并不只有ELF一种，还有一种常见的Shebang[19]。Shebang为字符序列`#!`，出现在文本文件的第一行的前两个字符。在文件中存在Shebang的情况下，类Unix系统会分析Shebang后的内容，并调用该指令。
> 更确切地说，在Linux kernel中，有项功能叫做`binfmt_misc`，它允许在shell中执行几乎所有的程序[20]，本质上是通过匹配magic number来解析这类可执行文件类型达到调用的目

> 的。2018年，有RFC（LWN: ns: introduce binfmt_misc namespace）提议增加 `binfmt_misc` namespac。

ELF

ELF由ELF header及文件数据组成，这些数据包括[21]：

- Program header tables: **execution view**, describes runtime memory segments.
- Section header tables: **linking view**, describes the ELF file offset of its corresponding "content".
- Data referred to by entries in the program header table or section header table



【图：ELF文件格式】

> Note：
> 可使用readelf查看ELF文件信息，比如`readelf -a /bin/bash`，亦可用objdump。
>
> Program header table中：
> - `INTERP` segment holds the full path name of runtime linker.
> - `LOAD` means loadable program segment, only this type segment are loaded into memory during execution.
>
> 在Section header tables中，`.`前缀表示这些section name是系统保留的，常见的比如`.text / .data / .bss`。此外：
> - `.dyn*`表明这是动态链接（dynamic link）时用到的section。
> - 有`.rel / .rela`字段的代表这是一个runtime / dynamic relocation table。
> - `.init / .fini`构成了进程的初始化/终止代码，位于此section中的函数会在`main()`之前/之后执行。利用这个特性，C++的全局构造和析构函数由此实现。
> - `.got / .plt`，Global Offset Table / Procedure Linkage Table，这是动态链接的跳转表和全局入口表。
> - `.tbss / .tdata`，这与线程局部存储（Thread-Local storage）有关。[23]

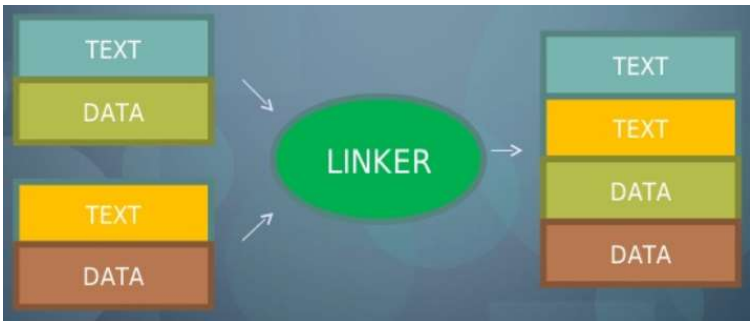
> GCC提供扩展机制，在全局变量或函数之前加上`__attribute__((section(<section name>))`属性，可以把相应的变量或函数放到以<section name>作为section name的section中。

[18]: https://en.wikipedia.org/wiki/Core_dump
[19]: https://en.wikipedia.org/wiki/Shebang_(Unix)
[20]: https://access.redhat.com/solutions/1985633
[21]: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
[22]: https://www.cs.stevens.edu/~jschauma/631/elf.html
[23]: https://en.wikipedia.org/wiki/Thread-local_storage

链接

Overview

多个源码文件编译后会生成多个目标文件（一般扩展名是`.o`或`.obj`），链接（Linking）的主要内容就是**把各个模块之间互相引用的部分处理好**，使得各个模块之间能够正确地链接。



【图：linker的工作】

链接是基于符号完成的，它需要使用与每一个目标文件都对应着的符号表（Symbol Table），这个表里面记录了目标文件中所用到的所有符号——变量名或函数名。为了防止符号冲突，现代编程语言采用了命名空间（namespace）的策略。对于符号重复定义问题，有了强引用（Strong Reference）与弱引用（Weak Reference）的概念。[25]

> Note:
> C++具有面向对象的概念，为了区分诸如函数重载带来的函数名相同、方法实现不同的情况，发明了名字修饰（Name Description）的机制。
> 可C并不具有这项机制。在C++代码中使用`extern "C"`关键字方法，可以使C++编译器区分C代码，从而实现兼容C。
> 为了应对C语言不支持`extern "C"`语法，可以使用宏`__cplusplus`来判断。
>
> GCC中默认函数和初始化了的全局变量为强符号。可通过`__attribute__((weak))`定义一个强符号为弱符号，通过`__attribute((weakref))`声明对一个外部函数的引用为弱符号。

[24]: https://en.wikipedia.org/wiki/Name_mangling

[25]: https://en.wikipedia.org/wiki/Weak_reference

静态链接

静态链接的一种策略相似端合并，一种方法是两步链接：1. 空间与地址分配；2. 符号解析与重定位。后者是通过每一个ELF文件种都具有的重定位表来完成的。如何链接，是通过linker scripts[26]来决定的。在脚本中规定了入口函数、section的组织方式、内存布局等信息。

> Note：
> `$ld --verbose`可查看默认的linker script。若要查看linker scripts文件的位置：
> - SUSE distribution，`$rpm -qf binutils`
> - Debian distribution, `$dpkg -L binutils-x86-64-linux-gnu`

[26]: http://wen00072.github.io/blog/2014/03/14/study-on-the-linker-script/

动态链接

动态链接将链接过程推迟到了运行时，它有很多好处，比如通过共享内存减少内存的占用、分割程序模块使之易于升级、动态选择加载程序等等，尽管带来了可以忽略不记的性能损失。通过延迟绑定（Lazy Binding）的做法，即函数第一次被用到才进行绑定，可减少性能损失，ELF通过PLT（Procedure Linkage Table）的方法来实现。
与静态链接一样，动态链接也需要解决重定位问题，不过共享对象的最终装载地址在编译时是不确定的，不能假设自己在虚拟内存中的位置。区别于静态链接时候的链接时重定位（Link Time Relocation），它叫做装载时重定位（Load Time Relocation），通过全局偏移表（GOT）间接跳转和调用，以及访问数据。

> Note:
> - Linux下区别于静态链接器ld，动态链接器往往名为ld-<version>.so。Windows的动态链接器内置在内核中。
> - GCC对装载重定位的支持是使用两个参数`-shared / -fPIC`来实现的。PIC即地址无关代码（Position-independent Code）。
> - 此外，参数`-fPIE`代表地址无关可执行文件（Position-independent Executable），是一项编译器相关的安全机制。
> - Linux下`$top`会看到*VIRT / RES / SHR*三部分信息[27]
>   - VIRT代表进程所占用的虚拟内存大小。
>   - RES是进程消耗的实际物理内存大小。因为动态链接库的存在，它总是小于VIRT。
>   - SHR代表VIRT中共享内存的大小。

另一种动态链接的方式是显式运行时链接（Explicit Run-time Linking）。Linux下提供了`dlopen() / dlsym() / dlerror() / dlclose()`系列函数解析动态链接库。

[27]: https://lilyfeng.wordpress.com/2013/07/17/the-difference-among-virt-res-and-shr-in-top-output/

MISC

共享库查找过程

Linux distributions中，在`/etc/ld.so.conf`文件中配置动态链接库的查找路径。为了加快查找速度，动态链接器使用的是通过`ldconfig`建立动态链接库绝对地址的缓存文件`/etc/ld.so.cache`。
此外，存在以下环境变量：

- `LD_LIBRARY_PATH`：临时改变动态链接库的查找路径，优先级高于`/etc/ld.so*`。
- `LD_PRELOAD`：指定预先装载的动态链接库或目标文件，优先级最高。
- `LD_DEBUG`：打开动态链接器的调试功能。

编译器安全机制

有如下几种[28][29]：

- Stack Protector
- Built as PIE
- NX
- Fortity
- Built with RELRO

GCC中的堆栈保护技术称为canaries，编译参数为`-fstack-protector / -fstack-protector-all`。原理是通过在缓冲区和控制信息间插入一个canary word，当缓冲区被溢出时，在返回地址被覆盖之前canary word会首先被覆盖。通过检查canary word的值是否被修改，就可以判断是否发生了攻击。[30][31]

地址无关可执行文件（Position Independent Executable，PIE），GCC编译参数为`-fPIE / -fpie`。在kernel开启ASLR（Address Space Layout Randomization）选项（即`kernel.randomize_va_space`）后完全生效，通过随机放置数据区域的地址空间来防止攻击者跳转到内存的特定位置。

NX（No execute），GCC编译参数为`-z execstack`。将数据所在的内存页标记为不可执行。[32]

Fortity，GCC编译参数`-D_FORTIFY_SOURCE=1 / -D_FORTIFY_SOURCE=2`。用于检测是否存在缓冲区溢出的错误。

只读重定位（Relocation Read Only，RELRO），GCC编译参数`-z relro / -z relro -z now`。将dynamic linker用到的relocation sections设置为只读。[33]

[28]: https://bbs.pediy.com/thread-226696.html
[29]: https://introspelliam.github.io/2017/09/30/linux%E7%A8%8B%E5%BA%8F%E7%9A%84%E5%B8%B8%E7%94%A8%E4%BF%9D%E6%8A%A4%E6%9C%BA%E5%88%B6/
[30]: https://en.wikipedia.org/wiki/Buffer_overflow_protection#GNU_Compiler_Collection_(GCC)
[31]: https://www.ibm.com/developerworks/cn/linux/l-cn-gccstack/index.html
[32]: https://hardenedlinux.github.io/system-security/2016/06/01/NX-analysis.html
[33]: https://medium.com/@HockeyInJune/relro-relocation-read-only-c8d0933faef3

BFD

现代硬件平台与软件平台的种类繁多，为了便于处理编译器和链接器处理不同平台之间的目标文件，发展出了BFD[34]。BFD（Binary File Descriptor library）的目标是希望通过一种统一的接口来处理不同的目标格式文件。
[34]: https://en.wikipedia.org/wiki/Binary_File_Descriptor_library

LWN Articles

- Randomizing structure layout
  - https://lwn.net/Articles/722293/
- Better kernels with GCC plugins
  - https://lwn.net/Articles/461696/
- Kernel building with GCC plugins
  - https://lwn.net/Articles/691102/
- An introduction to creating GCC plugins
  - https://lwn.net/Articles/457543/

GNU tool chain

GNU工具链是一个包含了由GNU计划所产生的各种编程工具的集合，由自由软件基金会负责维护制作。这些工具形成了一条工具链，用于开发应用程序和操作系统。
它包括：

- GNU make: an automation tool for compilation and build
- GNU Compiler Collection (GCC): a suite of compilers for several programming languages
- GNU C Library (glibc): core C library including headers, libraries, and dynamic loader
- GNU Binutils: a suite of tools including linker, assembler and other tools
- GNU Bison: a parser generator, often used with the Flex lexical analyser
- GNU m4: an m4 macro processor
- GNU Debugger (GDB): a code debugging tool

- GNU build system (autotools)
  - Autoconf: a tool for producing configure scripts for building, installing and packaging software
  - Automake: a programming tool to automate parts of the compilation process
  - Libtool: a tool for creating portable compiled libraries

https://en.wikipedia.org/wiki/GNU_toolchain

---

by river[river@vvl.me]

2019.0303: initialization