Linux 开发工具

曹东刚

caodg@sei.pku.edu.cn

Linux 程序设计环境

http://c.pku.edu.cn/

Unix 开发模型

- 大量久经考验的高质量专业工具
- cmdline vs IDE
- 让工具自动完成脏活累活
- 编辑器: vi vs emacs

binutils

- size
- strings
- ranlib
- nm
- strip
- as
- ld
- objdump

内容提要

静态检查工具

- lint: 代码分析检查工具
  - 很多功能被现代编译器取代
- cppcheck
- findbugs 等语言特定工具

cppcheck

```
1  void tryit()
2  {
3      int a[4];
4      int z = 4 + 1;
5
6      for (int n = 0; n < z; n++) {
7          a[n] = n;
8      }
9  }
```

cppcheck (cont.)

```
$ cppcheck --enable=all -v z.c
caodg@mars:~/ex$ cppcheck --enable=all -v z.c
Checking z.c...
[z.c:3]: (style) Variable 'a' is assigned a value
  that is never used
[z.c:8]: (error) Buffer access out-of-bounds: a
Checking usage of global functions..
[z.c:1]: (style) The function 'tryit' is never used
```

gcc 历史

- 1980s 中期, RMS 为 FSF 的 GNU 项目开发: Gnu C Compiler
- 1987 年 5 月, gcc 1.0 发布
- 1992 年, gcc 2.0 发布, 支持 C++
- 1997 年, Cygnus EGCS (Experimental/Enhanced GNU Compiler System) 项目
- 1999 年, EGCS 委员会成为 FSF 指定的 gcc 官方维护者 Gnu Compiler Collection
- 2001 年, gcc 3.0 发布, 支持 c++, Java, objective-c 等

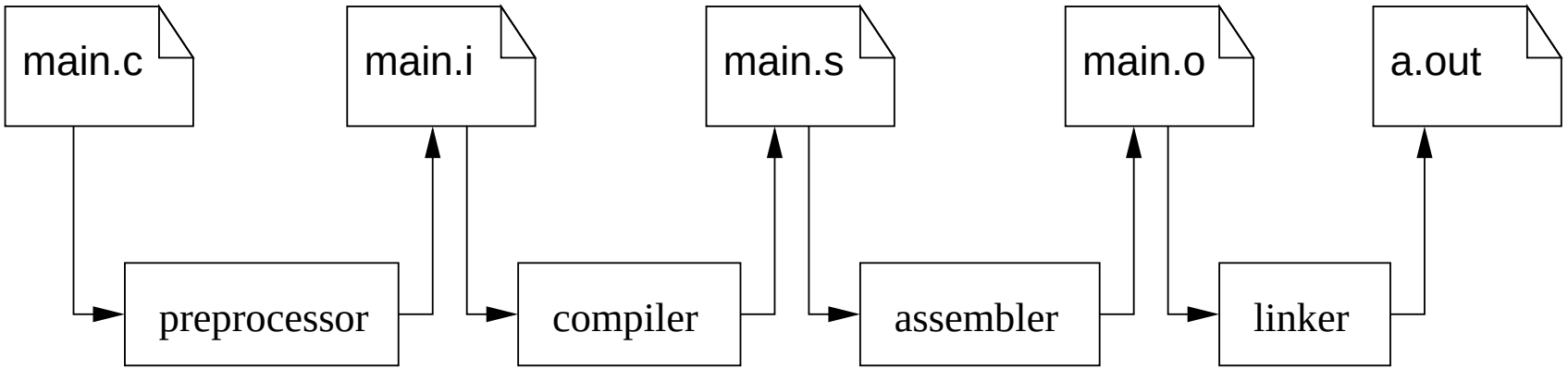
支持语言

- Ada (GCC for Ada aka GNAT)
- C
- C++ (GCC for C++ aka G++)
- Fortran (GCC for Fortran aka GFortran)
- Java (GCC for Java aka GCJ)
- Objective-C
- 以及 Pascal, Modula-2, Modula-3, Mercury, VHDL, PL/I, Objective-C++ 等

gcc 特征

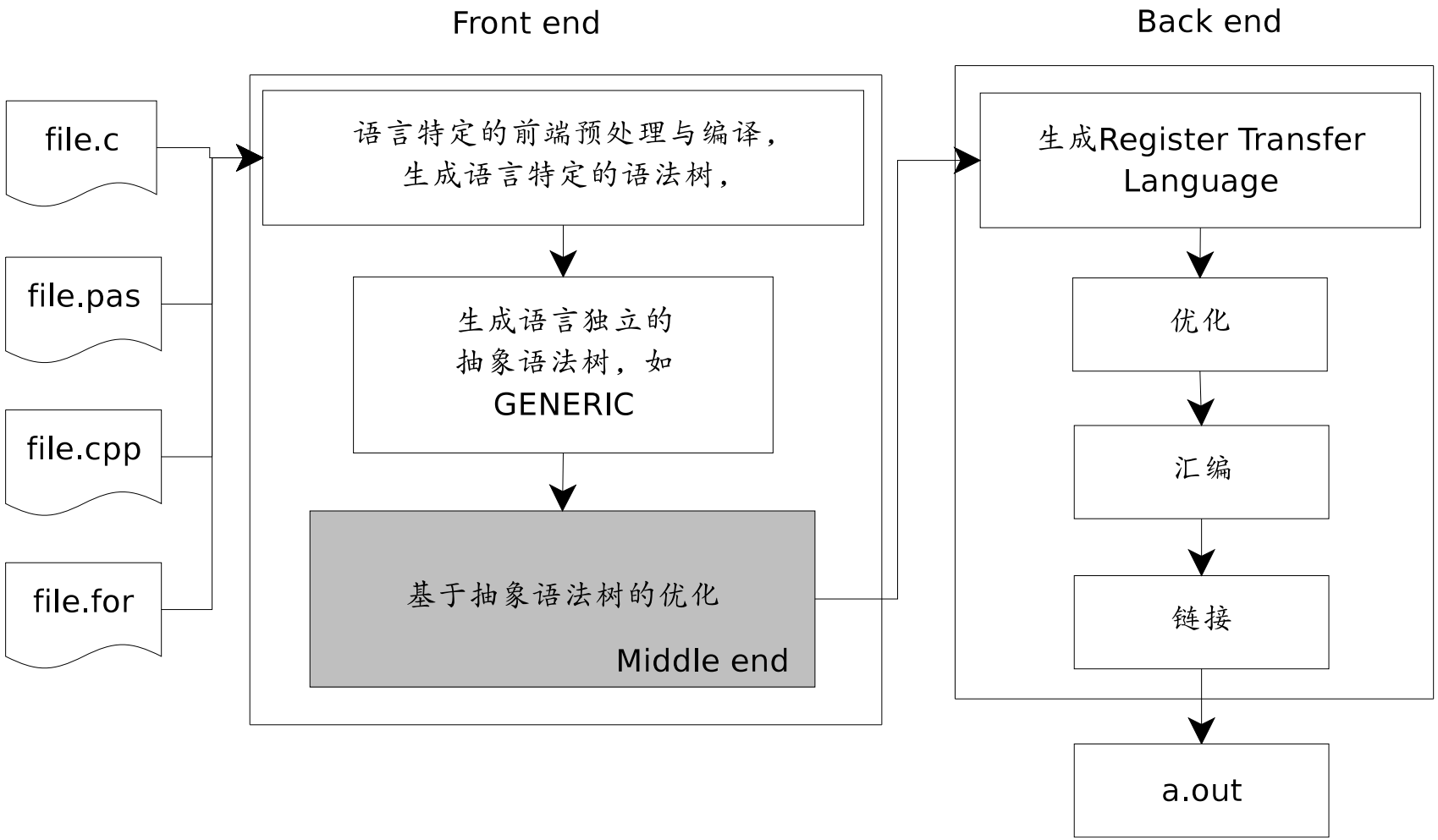
- 可移植
- 支持交叉编译
- 有多种语言前端
- 模块化结构，易扩展
- 免费

gcc 工作过程



- 预处理, e.g., *cpp*
- 编译, e.g., *cc*
- 汇编, e.g., *as*
- 链接, e.g., *ld*

前端与后端


Front end
Back end
file.c
file.pas
file.cpp
file.for
语言特定的前端预处理与编译，
生成语言特定的语法树，
生成语言独立的
抽象语法树，如
GENERIC
基于抽象语法树的优化
Middle end
生成Register Transfer
Language
优化
汇编
链接
a.out

编译一个 C 程序 —hello.c

```
1  #include <stdio.h>
2  int
3  main (void)
4  {
5      printf ("Hello, world!\n");
6      return 0;
7  }
```

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

检查错误 —bad.c

```
1  #include <stdio.h>
2  int
3  main (void)
4  {
5      printf ("Two plus two is %f\n", 4);
6      return 0;
7  }
```

```
$ gcc -Wall bad.c
bad.c: In function ‘main’:
bad.c:5:5: warning: format ‘%f’ expects argument of type
‘double’, but argument 2 has type ‘int’ [-Wformat]
```

编译多个文件: 将 Hello World 程序分开 –1

```
1 #include "hello.h"
2 int main (void)
3 {
4     hello ("world");
5     return 0;
6 }
```

编译多个文件: 将 Hello World 程序分开 –2

```
1 #include <stdio.h>
2 #include "hello.h"
3 void hello (const char * name)
4 {
5     printf ("Hello, %s!\n", name);
6 }
```

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

单独编译每个文件 —1

- 将一个大文件分成若干小文件
- 只编译改动的源文件
- 两阶段编译过程
  - 编译生成目标文件 file.o
  - 链接生成可执行文件

单独编译每个文件 —2

- 链接顺序: 包含函数定义的目标文件声明在调用者之后

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

重编译与重链接

```
1 // modify main.c
2 #include "hello.h"
3 int main (void)
4 {
5     hello ("everyone"); /* changed from "world" */
6     return 0;
7 }
```

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

链接外部库

库: 库是预编译的目标文件集合, 可以链接库形成可执行文件.

- 通过库提供系统调用, 如数学库, I/O 库等
- 将常用函数实现为库, 可以更好的复用
- 两种库: 静态库与共享库
  - 静态库以.a 结尾, 如 libtest.a. 通过工具ar创建
  - 共享库以.so 结尾
- 库同样存在链接顺序问题

链接数学库

```
1  // cal.c
2  #include <math.h>
3  #include <stdio.h>
4  int
5  main (void)
6  {
7      double x = sqrt (2.0);
8      printf ("The square root of 2.0 is %f\n", x);
9      return 0;
10 }
```

```
$ gcc -Wall calc.c /usr/lib/libm.a -o calc
$ gcc -Wall calc.c -lm -o calc
```

库和头函数的搜索路径

- 包含路径 (include path): 头文件所在路径

  gcc 的缺省头文件搜索路径

  /usr/local/include

  /usr/include

- 库搜索路径 (lib search path), 或链接路径 (link path)

  gcc 的缺省库搜索路径

  /usr/local/lib

  /lib

  /usr/lib

声明头文件搜索路径

- 头文件搜索路径

```
$ gcc -I. -I/net/include
```

或者声明环境变量

```
C_INCLUDE_PATH=.:/net/include
export C_INCLUDE_PATH
```

声明库搜索路径

- 库搜索路径

  ```
  $ gcc -L. -L/net/lib
  ```

  或者声明环境变量

  ```
  export LIBRARY_PATH=.:/net/lib
  ```

  混合使用时的搜索顺序: 命令行 > 环境变量 > 系统标准路径

共享库 —1

- 共享库 (shared library) 是一种特殊的库 (以.so 为扩展名), 要求程序执行前将其从硬盘加载
- 链接静态库时, 静态库的代码被复制到目标程序中
- 链接共享库时, 目标程序只保留共享库的函数表
  - 运行时, 共享库的函数代码被操作系统复制到内存, 此过程称为动态链接 (dynamic linking)

共享库 —2

- 接口不变的情况下，可以更新共享库而不必重新编译程序
- 当使用"-lname" 加载库时，*gcc* 首先尝试加载 *libname.so*，其次才尝试 *libname.a*

共享库搜索路径

gcc 共享库缺省搜索路径

/usr/local/lib

/lib

/usr/lib

可以通过命令行 (-L) 方式声明搜索路径, 也可以通过环境变量声明

```
$ LD_LIBRARY_PATH=.:/net/lib
$ export LD_LIBRARY_PATH
```

也可以通过选项 -static, 强制 gcc 使用静态链接库

建立静态链接库

命令 *ar* 可将多个目标文件合并成一个库文件.

例: 将第 15页的 hello.c 文件和下面的 bye.c 文件编译为库文件

```
1 #include <stdio.h>
2 #include "hello.h"
3 void bye (void)
4 {
5     printf ("Goodbye!\n");
6 }
```

```
$ gcc -Wall -g -c hello.c bye.c
$ ar cr libhello.a hello.o bye.o
$ gcc -Wall main.c -L. -lhello -o hello
```

检查文件

- *file* 命令查看可执行文件的基本信息，例：

```
$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), for GNU/Linux 2.2.0, dynamically
linked (uses shared libs), not stripped
```

- *nm* 命令查看可执行文件或目标文件的符号表，例：

```
$ nm a.out
```

- *ldd* 命令查看可执行文件需要的共享库，例：

```
$ ldd a.out
```

生成共享库

- 为共享库编译目标文件

```
$ gcc -Wall -g -fPIC -c hello.c bye.c
```

- 生成共享库，链接生成可执行文件

```
gcc -shared -fPIC -o libhello.so hello.o bye.o
gcc -o app -L. -lhello main.c
```

- 运行

```
$ export LD_LIBRARY_PATH=.
$ ./app boys
```

llvm 与 clang

LLVM: 一种开源编译设施, 拥有系列工具

- 技术现代先进, 始于 2003 年
- 模块化, 易扩展
- 支持多种语言
- BSD License

clang: llvm 的一个前端工具, 目标是代替 gcc

- 获得 google/apple/bsd 社区的支持
- 包括 clang 前端和 clang 静态检查工具 scan-build

clang in FreeBSD

FreeBSD 从 v10 开始采用 clang, 放弃 gcc

- GPL v3 不兼容 BSD License
  - Tivoisation
- Apple 公司投资影响
- 吸引和留住 FreeBSD 的企业用户
- GCC 自身存在问题, 对标准的兼容不够, 难以定制
  - 超过 3 百万行代码, “最复杂的开源项目之一”
- clang 比 GCC 拥有技术优势

clang vs gcc

gcc 的优点

- 支持 java, ada, fortran 等
- 支持的目标机器比 llvm 多

clang 的优点

- 容易学习，容易扩展，容易复用，容易集成
- 错误诊断更友好
- 运行速度更快，占用资源更少
- 对静态分析和代码生成支持的更好

scan-build: clang 静态分析工具

```
1  int tryit()
2  {
3      int a[4];
4      int z = 4 + 1;
5
6      for (int n = 0; n < z; n++) {
7          a[n] = n;
8      }
9
10     if ( z = 400 )
11         z ++ ;
12     return z ;
13 }
```

```
caodg@mars:~/ex$ scan-build clang -c z1.c
z1.c:11:12: warning: using the result of an assignment as
  a condition without parentheses [-Wparentheses]
    if ( z = 400 )
         ~~^~~~
z1.c:11:12: note: place parentheses around the assignment
  to silence this warning
    if ( z = 400 )
           ^
         (     )
z1.c:11:12: note: use '==' to turn this assignment into
  an equality comparison
    if ( z = 400 )
           ^
           ==
```

运行 cppcheck

```
caodg@mars:~/ex$ cppcheck --enable=all -v z1.c
Checking z1.c...
[z1.c:3]: (style) Variable 'a' is assigned a value that
  is never used
[z1.c:8]: (error) Buffer access out-of-bounds: a
Checking usage of global functions..
[z1.c:1]: (style) The function 'tryit' is never used
```

运行 gcc

```
caodg@mars:~/ex$ gcc -std=c99 -Wall -c z1.c
z1.c: In function ‘tryit’:
z1.c:11:5: warning: suggest parentheses around assignment
  used as truth value [-Wparentheses]
z1.c:3:9: warning: variable ‘a’ set but not used
  [-Wunused-but-set-variable]
```

内容提要

调试开关

通常可执行文件中不包含对源程序的引用信息，如变量名，函数名，行号等.

gcc 提供了 '-g' 开关，将源程序的信息存放在目标文件和可执行文件的符号表中，允许

- 调试器 (debugger) 跟踪程序的执行
- 当程序崩溃的时候，根据生成的 "core" 文件，检查程序崩溃前的状态，例如非法内存访问，被零除等
- 通过 *ulimit* 命令，设置是否允许生成 "core" 文件

```
$ ulimit -c unlimited
```

gdb

- 运行并调试

  $ gdb program

- 调试崩溃程序

  $ gdb program core

- 调试已运行程序

  $ gdb program processId

gdb 常用命令

break [file:]function	设置断点，或者 break [file:]linenumber
run [arglist]	启动待调试程序
bt	backtrace，显示程序栈
where	显示当前位置
print expr	打印表达式的值
c	continue，继续运行
next	执行下一行，跳过函数入口
step	执行下一行，跳进函数入口
list [file:]function	显示程序停止位置的源程序
help [cmd]	显示 cmd 命令的使用帮助
quit	退出

内容提要

性能度量含义

通过改变用户负载测试度量系统的行为

- Benchmark Testing
- Durability Testing
- Load Testing
- Scalability Testing
- Stress Testing
- Volume Testing

gprof

gprof 用于度量程序的性能，统计函数的调用次数和执行时间

例：统计一个 Collatz conjecture 猜想计算步长的程序

$$x_{n+1} \longleftarrow \begin{cases} x_n/2 & x_n \quad \text{is even} \\ 3x_n + 1 & x_n \quad \text{is odd} \end{cases}$$

```
$ gcc -Wall -pg collatz.c
$ ./a.out
```

性能统计数据存放在文件 gmon.out 中

```
$ gprof a.out
```

Valgrind

Valgrind 是一款强大的开源工具集，它包含有包括内存检测、线程监测等多种工具，其中最常用的是内存检测功能，它能监测出以下的各种内存错误：

- 访问非法内存区域
- 使用未被初始化的内存区域
- 非法释放内存，比如多次 free 一个内存
- 内存泄露

开源 Java 度量工具

- NetBean Profiler
- VisualVM
- Grinder

大家可以和商业工具 JProfiler 和 JProbe 比较.

JMeter

JMeter 是 Apache 的 Java 桌面应用程序，用于度量被测试软件的性能。

- 初衷是测试 Web 应用，后来又扩充了其它的功能。
- 可以完成针对静态资源和动态资源（HTTP, Servlets, Perl 脚本, Java 对象, 数据查询 s, FTP 服务等）的性能测试。
- 可以模拟大量的服务器负载、网络负载、软件对象负载，通过不同的加载类型全面测试软件的性能。
- 提供图形化的性能分析。

内容提要

gcov: 统计哪些语句被执行，以及执行频率

```
1  #include <stdio.h>
2  int main (void)
3  {
4      int i;
5      for (i = 1; i < 10; i++) {
6          if (i % 3 == 0)
7              printf ("%d is divisible by 3\n", i);
8          if (i % 11 == 0)
9              printf ("%d is divisible by 11\n", i);
10     }
11     return 0;
12 }
```

编译并测试

```
$ gcc -Wall -fprofile-arcs -ftest-coverage cov.c
$ ./a.out
$ gcov cov.c
```

在当前目录生成 cov.c.gcov, 部分内容

```
   10:    5:  for (i = 1; i < 10; i++) {
    9:    6:    if (i % 3 == 0)
    3:    7:      printf ("%d is divisible by 3\n", i);
    9:    8:    if (i % 11 == 0)
#####:    9:      printf ("%d is divisible by 11\n", i);
    -:   10:  }
```

gcov 衍生工具

ggcov

ggcov is a GTK+ GUI for exploring test coverage data produced by C and C++ programs compiled with gcc -fprofile-arcs -ftest-coverage.

lcov

LCOV is a graphical front-end for GCC's coverage testing tool gcov. It collects gcov data for multiple source files and creates HTML pages containing the source code annotated with coverage information.

Java 覆盖测试工具

- Cobertura
- EclEmma
- Clover (商业 License，但对开源项目免费)

内容提要

gettext

主要文件:

- .pot: 由 xgettext 生成的 po 文件模板
- .po: 语言特定翻译文件，可由 msginit + .pot 生成
- .mo: 编译后的 po，可由 msgfmt + .po 生成

主要工具:

- xgettext: 从源代码中提取需要翻译的字串，生成 pot 文件
- msginit: 替换 pot 中的 Entry 信息，如译者，文件编码等
- msgmerge: 合并现有的.po 文件
- msgfmt: 把.po 文件生成.mo 文件

示例：源程序 hello.c

```
1  #include <locale.h>
2  #include <libintl.h>
3  #include <stdio.h>
4  #define _(string) gettext(string)
5  const char *DOMAIN  = "hello";
6  const char *DIRNAME = "locales";
7  int main(int argc, char **argv) {
8          setlocale(LC_ALL, "");
9          bindtextdomain(DOMAIN, DIRNAME);
10         textdomain(DOMAIN);
11         printf(_("Hello World"));
12         return 0;
13 }
```

示例：制作 mo

```
$ xgettext -k_  hello.c  -o hello.pot
$ msginit -l zh_CN
```

此时目录下有 hello.pot 和 zh_CN.po，将 zh_CN.po 内容更改为

```
msgid "Hello World"
msgstr " 你好世界"
```

```
$ msgfmt zh_CN.po -o zh_CN.mo
$ cp zh_CN.mo locales/zh_CN/LC_MESSAGES/hello.mo
$ gcc hello.c
$ LC_ALL=zh_CN ./a.out
```