



Linux Programming Prerequisite

Enyi Tang
Software Institute, Nanjing
University



Programming Principle

- Abstract vs. Concrete
- 库(API)的调用与选择



Programming Principle

- Abstract vs. Concrete
- 库(API)的调用与选择



编程工具

- 编辑工具
 - vi, emacs
- 编译、链接
 - gcc
- 调试
 - gdb
- make命令
- 版本控制工具
 - CVS等



Programming Language

- High-level Language

- C/C++, Java, Fortran...
- ELF binary format
 - Executable and Linkable Format
 - 工具接口标准委员会(TIS)选择了正在发展中的**ELF**体系上不同操作系统之间可移植的二进制文件格式

- Script

- Shell: sh/bash, csh, ksh
- Perl, Python, tcl/tk, sed, awk...



Development Tools

- GCC

- GNU C Compiler -> GNU Compiler Collection
- The gcc command: Front end

- GDB

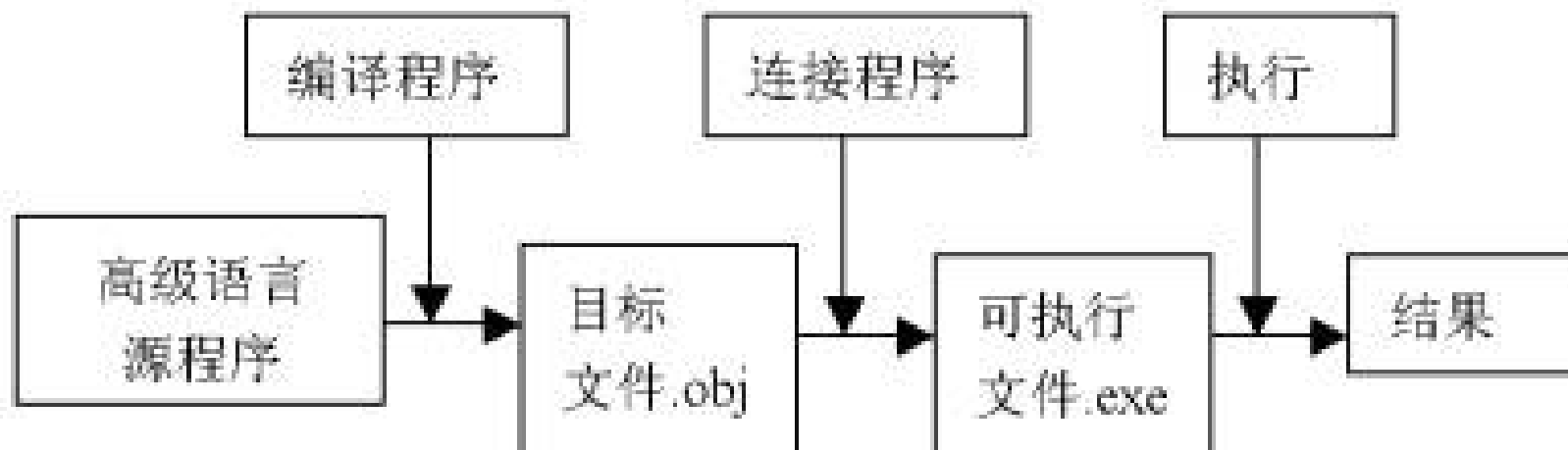
- GNU Debugger
- The gdb command
- xxdgb, ddd...

- Binary utilities

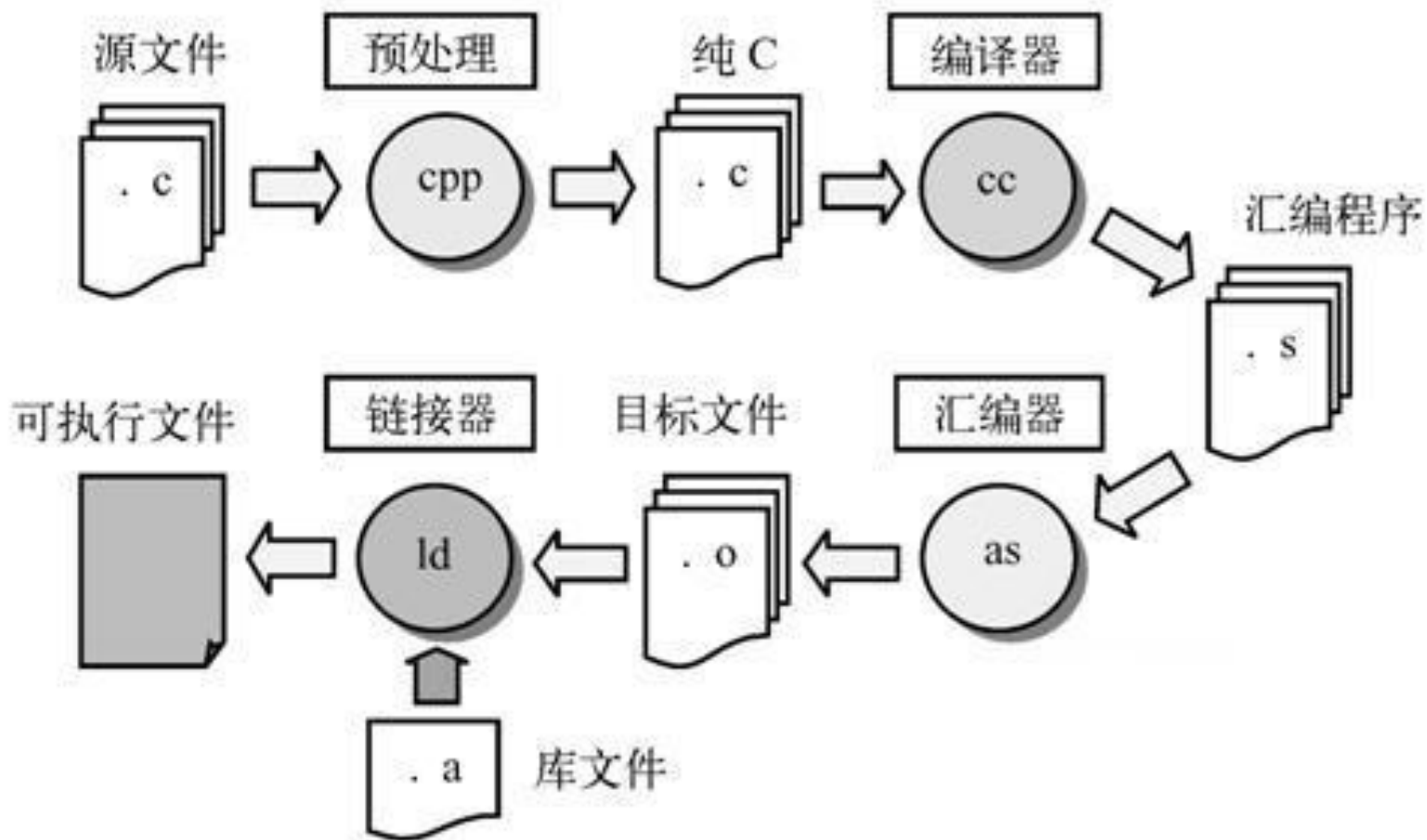
- as, ld, ar, ldd...

- Make

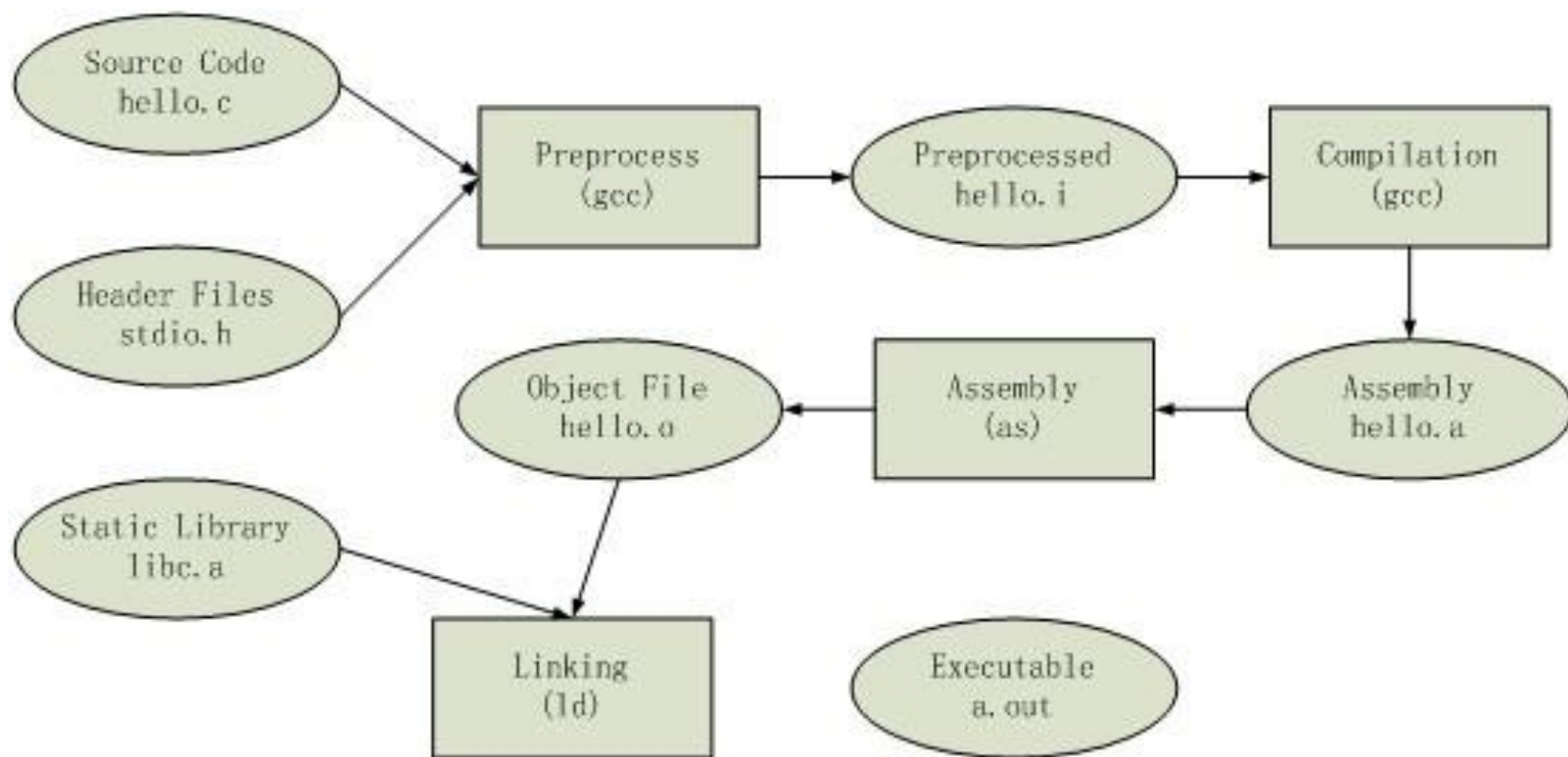
最简单的编译链接图(Win)

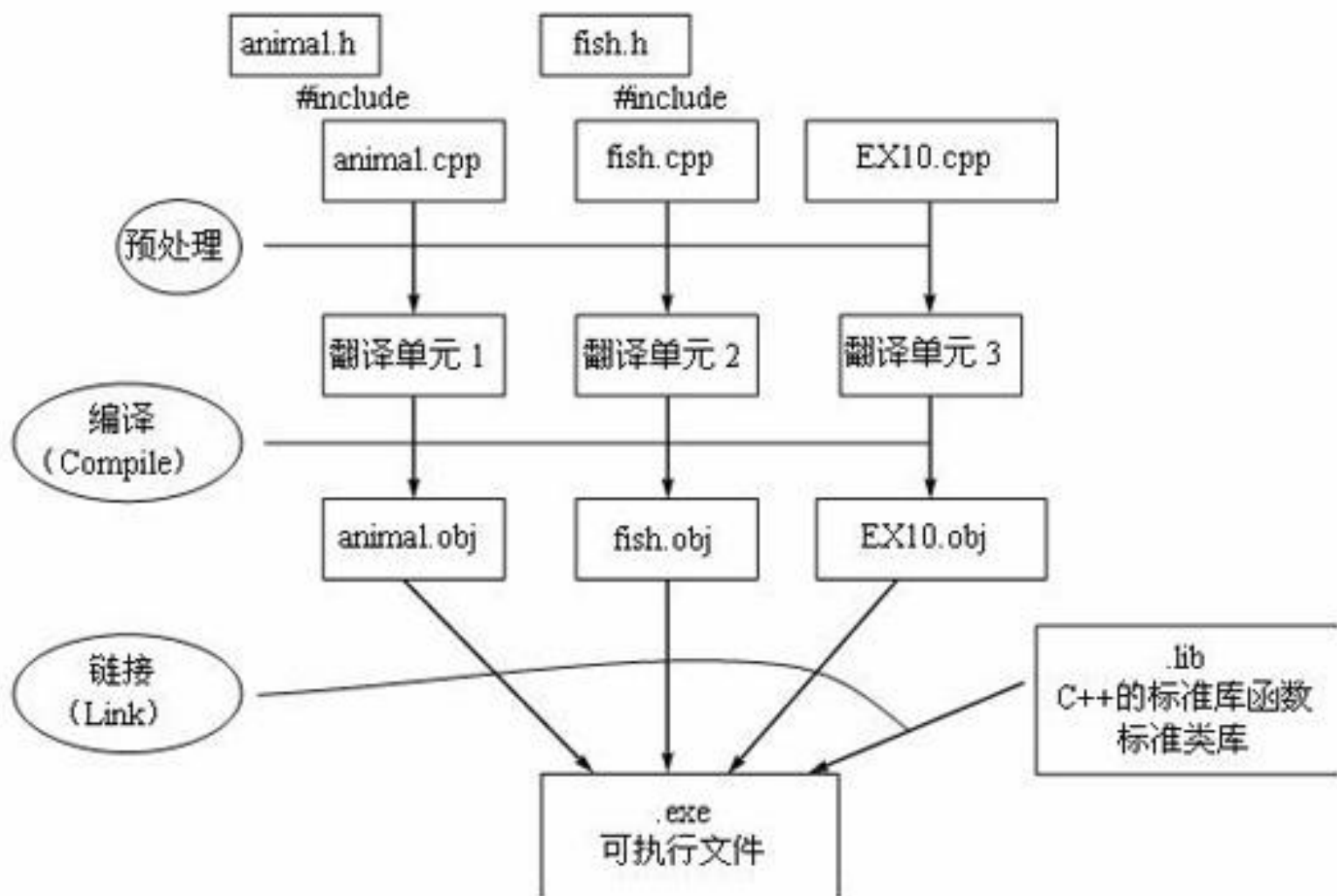


编译链接图(展开)



编译链接图(头文件展开)







编译链接

- 头文件和`#include` (预处理 – 编译时处理)
- 为什么要做链接? (link)
- 静态库与动态库



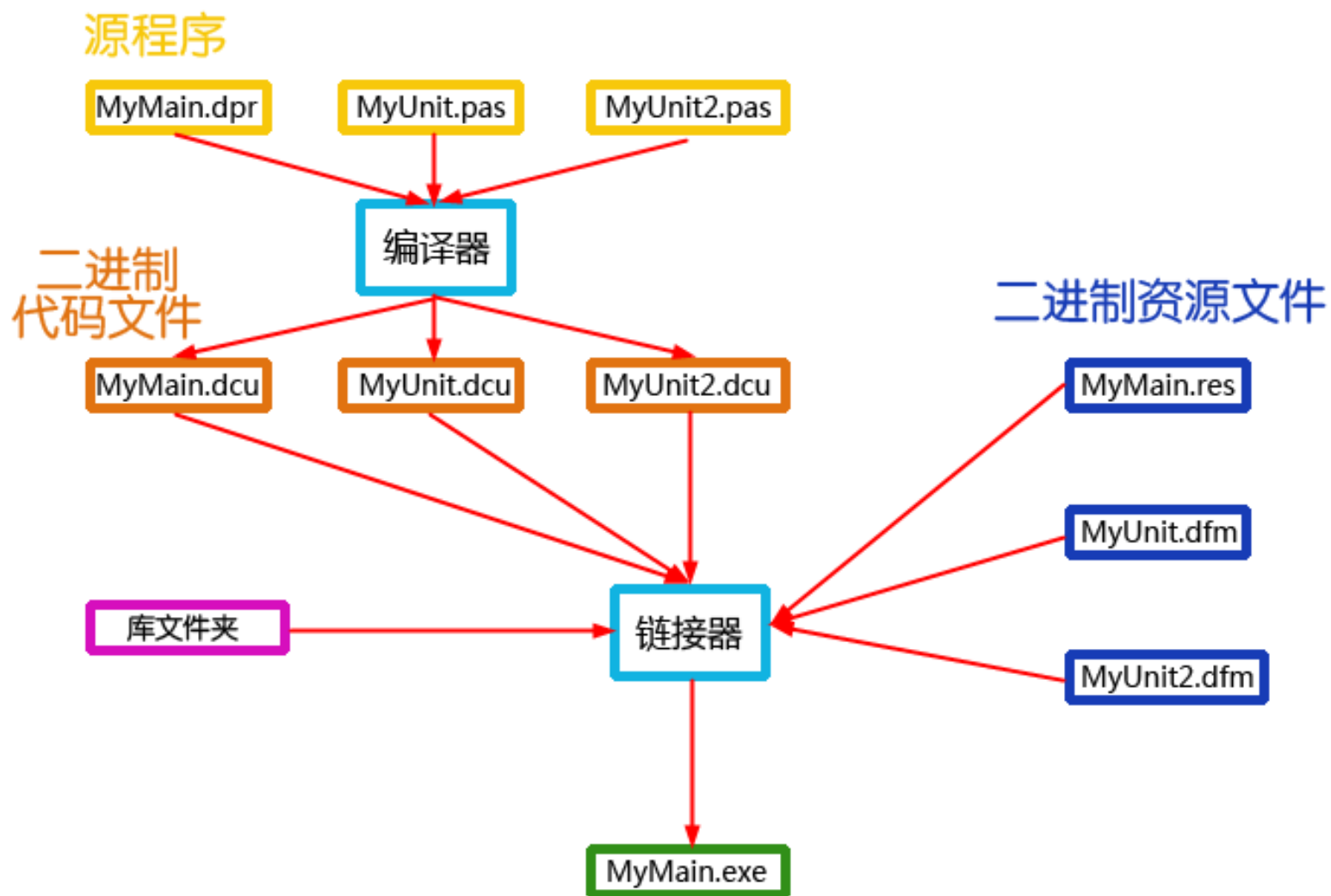
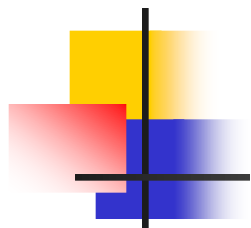
Libraries and Head Files

- Static Libraries (.a files)
 - Lab (gcc + ar)
- Dynamic Libraries/Shared Objects (.so files)
 - Lab (gcc)



其它语言

- Java (只有编译+解释执行)
- .Net平台 (VB.net, C# C++.net等)
- VC++ 及 Delphi 等 (一般称为Win编程)



Delphi编译/链接过程

编译的具体过程

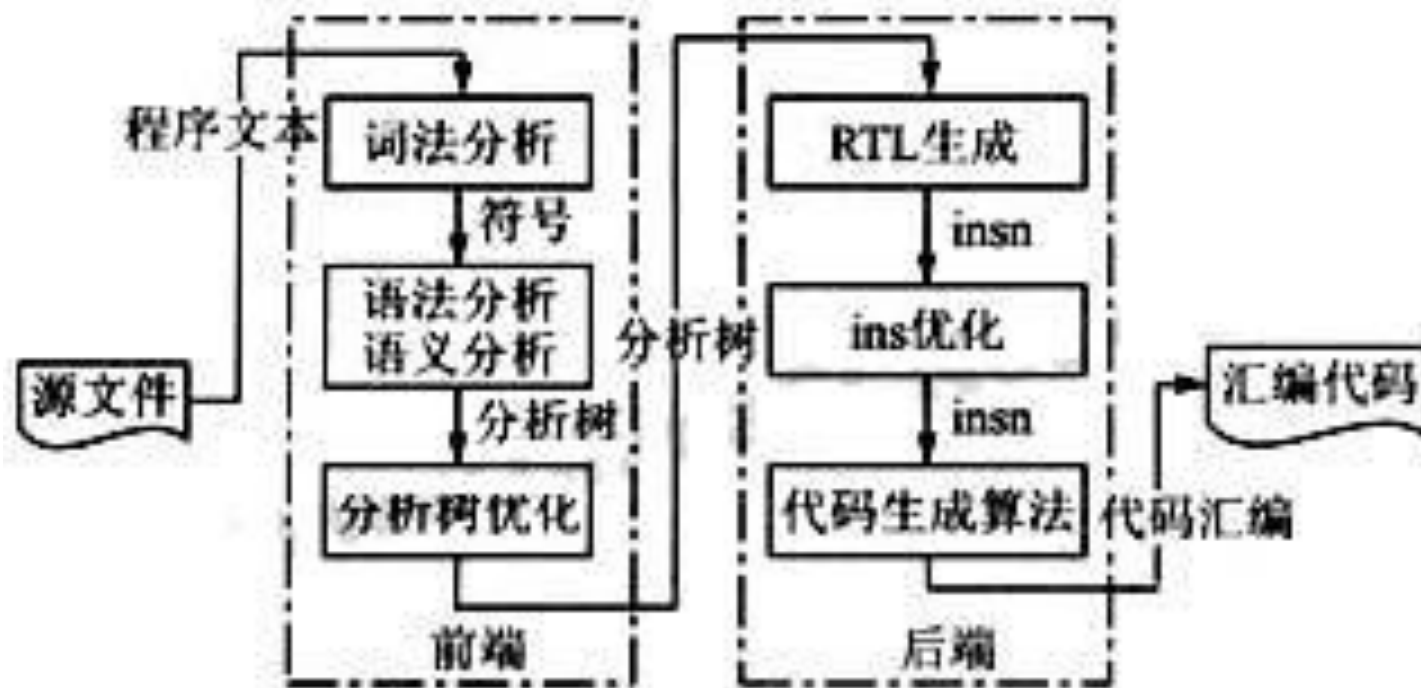
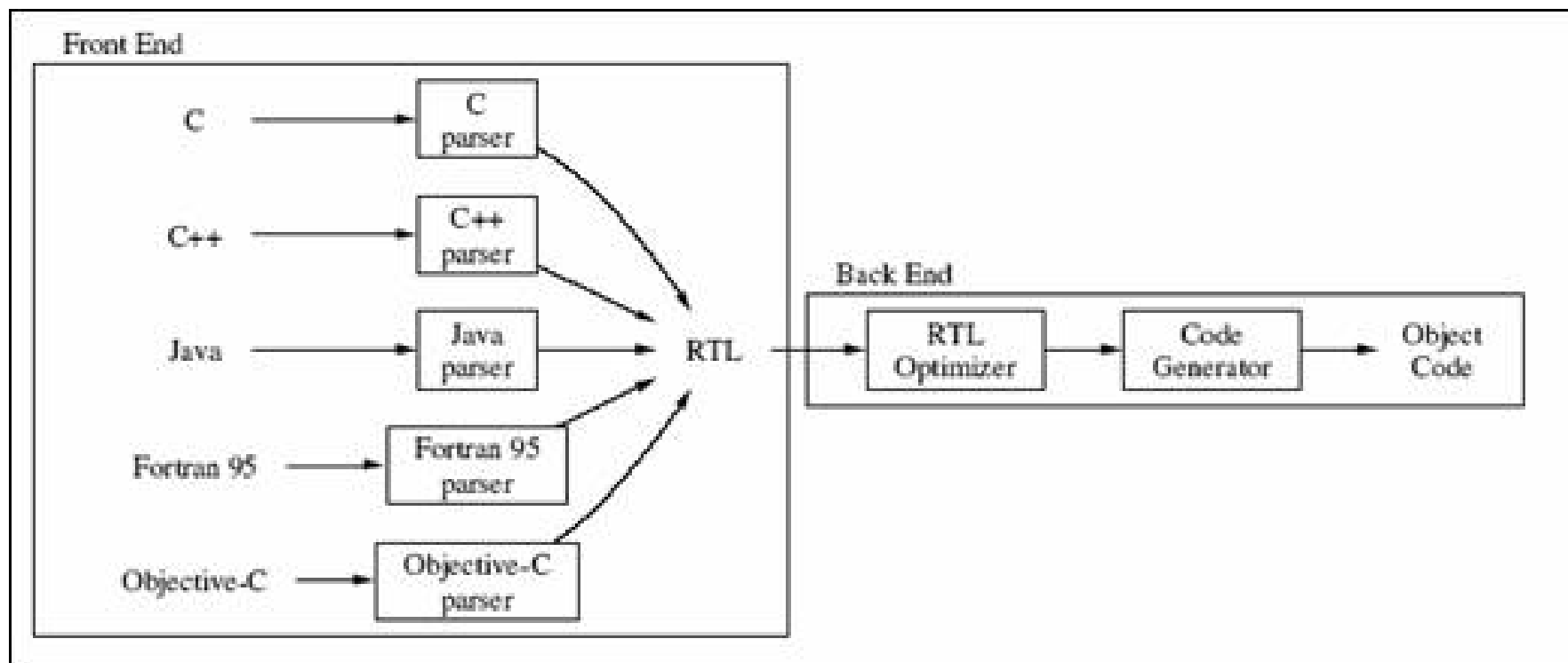


图 2 GCC 编译器结构

前端和后端





常用命令

- `gcc -c` (编译)
- `gcc` (链接 或者 编译 + 链接)
- `g++` (C++对应的命令，其实就是换了前端)



GCC options (1)

- Usage:
 - gcc [options] [filename]
- Basic options:
 - -E: 只对源程序进行预处理(调用cpp预处理器)
 - -S: 只对源程序进行预处理、编译
 - -c: 执行预处理、编译、汇编而不链接
 - -o output_file: 指定输出文件名
 - -g: 产生调试工具必需的符号信息
 - -O/On: 在程序编译、链接过程中进行优化处理
 - -Wall: 显示所有的警告信息



GCC options (2)

- Basic options:

- -Idir: 指定额外的头文件搜索路径
- -Ldir: 指定额外的库文件搜索路径
- -lname: 链接时搜索指定的库文件
- -DMACRO[=DEFN]: 定义MACRO宏



File Name Suffix (1)

.c	C source code which must be preprocessed
.i	C source code which should not be preprocessed
.cc .cp .cpp .CPP. c++ .C .cxx	C++ source code which must be preprocessed
.ii	C++ source code which should not be preprocessed
.h	C or C++ header file to be turned into a precompiled header
.H .hh	C++ header file to be turned into a precompiled header
.s	Assembler code
.S	Assembler code which must be preprocessed



File Name Suffix (2)

.o	Object file
.a	Static library file (archive file)
.so	Dynamic library file (shared object)



GDB

- GDB: GNU Debug

- 设置断点
- 监视变量值
- 单步执行
- 修改变量值



`gdb` commands

<code>file</code>	打开要调试的文件
<code>break/tbreak</code>	设置断点，可以是行号、函数名及地址(以*开头) tbreak : 设置临时断点
<code>run</code>	执行当前调试的程序
<code>list</code>	列出源代码的一部分
<code>next</code>	执行一条语句但不进入函数内部
<code>step</code>	执行一条语句，是函数则进入函数内部
<code>display</code>	显示表达式的值
<code>print</code>	临时显示表达式的值
<code>kill</code>	中止正在调试的程序
<code>quit</code>	推出 gdb
<code>shell</code>	不退出 gdb 就执行 shell 命令
<code>make</code>	不退出 gdb 就执行 make



make & makefile

- Multi-file project
 - IDE
 - make
- make & makefile
 - makefile描述模块间的依赖关系；
 - make命令根据makefile对程序进行管理和维护； make判断被维护文件的时序关系



Hello的makefile

- **TOPDIR = ../**
- **include \$(TOPDIR)Rules.mak**
- **EXTRA_LIBS +=**
- **EXEC = \$(INSTALL_DIR)/hello**
- **OBJS = hello.o**

- **all: \$(EXEC)**
- **\$(EXEC): \$(OBJS)**
- **\$(CC) \$(LDFLAGS) -o \$@ \$(OBJS) \$(EXTRA_LIBS)**
- **install:**
- **\$(EXP_INSTALL) \$(EXEC) \$(INSTALL_DIR)**

- **clean:**
- **-rm -f \$(EXEC) *.elf *.gdb *.o**



makefile

- **定义整个工程的编译规则**

一个工程中的源文件不计数，其按类型、功能、模块分别放在若干个目录中，makefile定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作。

- **自动化编译**

只需要一个make命令，整个工程完全自动编译；
make是一个命令工具，是一个解释makefile中指令的命令工具；



makefile

- **GNU make**是一个命令工具，是一个用来控制软件构建过程的自动化管理工具。**Make**工具通过称为**Makefile**的文件来完成并自动维护编译工作,由**Richard Stallman**与**Roland McGrath**设计开发。
- **Makefile**是用于自动编译和链接的，一个工程有很多文件组成，每一个文件的改变都会导致工程的重新链接，但是不是所有的文件都需要重新编译，**Makefile**中记录有文件的信息，在**make**时会决定在链接的时候需要重新编译哪些文件。
- **make**命令格式：**make [-f Makefile] [option] [target]**
- **#make target** **#make** **#make clean**



make

- `make [-f filename] [targetname]`
- Targets
 - A target is usually the name of a file that is generated by a program; examples of targets are executable or object files.
 - A target can also be the name of an action to carry out, such as 'clean' (phony target).



Makefile 规则结构

- **target ... : prerequisites ...
command**
...
...
 - target是一个目标文件，可以是Object File，也可以是执行文件
 - prerequisites是要生成target所需要的文件或是目标
 - command是make需要执行的命令。（可以是任意的Shell命令）
- 举例

```
hello : main.o kbd.o
    gcc -o hello main.o kbd.o
main.o : main.c defs.h
    cc -c main.c
kbd.o : kbd.c defs.h command.h
    cc -c kbd.c
clean :
    rm edit main.o kbd.o
```



Makefile 执行次序

- 1、**make**会在当前目录下找名字叫“**Makefile**”或“**makefile**”的文件。
- 2、查找文件中的第一个目标文件（**target**），举例中的**hello**
- 3、如果**hello**文件不存在，或是**hello**所依赖的文件修改时间要比**hello**新，就会执行后面所定义的命令来生成**hello**文件。
- 4、如果**hello**所依赖的**.o**文件不存在，那么**make**会在当前文件中找目标为**.o**文件的依赖性，如果找到则再根据那一个规则生成**.o**文件。（类似一个堆栈的过程）
- 5、**make**根据**.o**文件的规则生成 **.o** 文件，然后再用 **.o** 文件生成**hello**文件。



伪目标

clean:

```
rm *.o hello
```

- “伪目标”并不是一个文件，只是一个标签，所以 **make** 无法生成它的依赖关系和决定它是否要执行，只能通过显示地指明这个“目标”才能让其生效
- “伪目标”的取名不能和文件名重名
- 为了避免和文件重名的这种情况，可以使用一个特殊的标记“**.PHONY**”来显示地指明一个目标是“伪目标”，向 **make** 说明，不管是否有这个文件，这个目标就是“伪目标”
- 伪目标一般没有依赖的文件，但也可以为伪目标指定所依赖的文件。
- 伪目标同样可以作为“默认目标”，只要将其放在第一个。

多目标

- 用处

- 当多个目标同时依赖于一个文件，并且其生成的命令大体类似，可以使用一个自动化变量“\$@"表示着目前规则中所有的目标的集合

- 举例

```
bigoutput littleoutput : text.g
generate text.g -$(subst output,, $@) > $@
```

上述规则等价于

```
bigoutput : text.g
generate text.g -big > bigoutput

littleoutput : text.g
generate text.g -little > littleoutput
```




Hello的makefile

- **TOPDIR = ../**
- **include \$(TOPDIR)Rules.mak**
- **EXTRA_LIBS +=**
- **EXEC = \$(INSTALL_DIR)/hello**
- **OBJS = hello.o**


- **all: \$(EXEC)**
- **\$(EXEC): \$(OBJS)**
- **\$(CC) \$(LDFLAGS) -o \$@ \$(OBJS) \$(EXTRA_LIBS)**
- **install:**
- **\$(EXP_INSTALL) \$(EXEC) \$(INSTALL_DIR)**

- **clean:**
- **-rm -f \$(EXEC) *.elf *.gdb *.o**



预定义变量

- `$<` 第一个依赖文件的名称
- `$?` 所有的依赖文件，以空格分开，这些依赖文件的修改日期比目标的创建日期晚
- `$+` 所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件
- `$^` 所有的依赖文件，以空格分开，不包含重复的依赖文件
- `$*` 不包括扩展名的目标文件名称
- `$@` 目标的完整名称
- `$%` 如果目标是归档成员，则该变量表示目标的归档成员名称



- edit : main.o kbd.o command.o display.o \
insert.o search.o files.o utils.o
gcc -o edit main.o kbd.o command.o display.o \
insert.o search.o files.o utils.o
- main.o : main.c defs.h
gcc -c main.c
- kbd.o : kbd.c defs.h command.h
gcc -c kbd.c
- command.o : command.c defs.h command.h
gcc -c command.c
- display.o : display.c defs.h buffer.h
gcc -c display.c
- insert.o : insert.c defs.h buffer.h
gcc -c insert.c
- search.o : search.c defs.h buffer.h
gcc -c search.c
- files.o : files.c defs.h buffer.h command.h
gcc -c files.c
- utils.o : utils.c defs.h
gcc -c utils.c
- clean :
rm edit main.o kbd.o command.o display.o \
insert.o search.o files.o utils.o

- OBJECTS = main.o kbd.o command.o display.o \
insert.o search.o files.o utils.o



```
edit : $(OBJECTS)
```

```
gcc -o edit $(OBJECTS)
```

```
main.o : main.c defs.h
```

```
gcc -c main.c
```

```
kbd.o : kbd.c defs.h command.h
```

```
gcc -c kbd.c
```

```
command.o : command.c defs.h command.h
```

```
gcc -c command.c
```

```
display.o : display.c defs.h buffer.h
```

```
gcc -c display.c
```

```
insert.o : insert.c defs.h buffer.h
```

```
gcc -c insert.c
```

```
search.o : search.c defs.h buffer.h
```

```
gcc -c search.c
```

```
files.o : files.c defs.h buffer.h command.h
```

```
gcc -c files.c
```

```
utils.o : utils.c defs.h
```

```
gcc -c utils.c
```

```
clean :
```

```
rm edit $(OBJECTS)
```



多目标扩展

- 语法

```
<targets ...>: <target-pattern>: <prereq-patterns ...>  
    <commands>  
    ...
```

- 举例

```
objects = foo.o bar.o  
all: $(objects)  
$(objects): %.o: %.c  
    $(CC) -c $(CFLAGS) $< -o $@
```

- 目标从`$object`中获取
- “%.o”表明要所有以“.o”结尾的目标，即“foo.o bar.o”，就是变量`$object`集合的模式
- 依赖模式“%.c”则取模式“%.o”的“%”，也就是“foo bar”，并为其加下“.c”的后缀，于是依赖的目标就是“foo.c bar.c”



上述规则等价于

foo.o : foo.c

`$(CC) -c $(CFLAGS) foo.c -o foo.o`

bar.o : bar.c

`$(CC) -c $(CFLAGS) bar.c -o bar.o`

使用函数

- 调用语法
 - `$(<function> <arguments>)`
 - `${<function> <arguments>}`
- 字符串处理函数
 - `$(subst <from>,<to>,<text>)`
 - `$(strip <string>)`
 -
- 文件名操作函数
 - `$(dir <names...>)`
 - `$(basename <names...>)`
 -
- **foreach** 函数
 - `$(foreach <var>,<list>,<text>)`
- **if** 函数
 - `$(if <condition>,<then-part>)`
 - `$(if <condition>,<then-part>,<else-part>)`
- **call**函数
 - `$(call <expression>,<parm1>,<parm2>,<parm3>...)`
-



软件设计原则

- 清晰原则
- 吝啬原则
- 扩展原则