1.1 Linux 下的 C 语言开发环境

第 1 章 Linux 环境下 C 语言的开发

本章介绍 Linux 操作系统环境下 C 语言开发的基本概念和程序运行的原理。

在本章的学习中，读者应重点关注以下内容：

Linux 中 C 语言开发的流程和工具

Linux 中 C 语言程序的运行机制

1.1 Linux 下的 C 语言开发环境

Linux 和 C 语言有很深的渊源，因为 Linux 本身就是用 C 语言编写的。同时，在 Linux 操作系统中也提供了 C 语言的开发环境。这些开发环境一般包括程序生成工具、程序调试工具、工程管理工具等。

1．程序生成工具

在 Linux 中，一般使用 GCC（GNU Compiler Collection）作为程序生成工具。GCC 提供了 C 语言的编译器、汇编器、连接器以及一系列辅助工具。GCC 可以用于生成 Linux 中的应用程序， 也可以用于编译 Linux 内核和内核模块，是 Linux 中 C 语言开发的核心工具。

2．程序调试工具

GDB 是 Linux 中一个强大的命令行调试工具，使用 GDB 调试 C 语言的时候，可以使用设置断点、单步运行、查看变量等功能。

3．工程管理工具

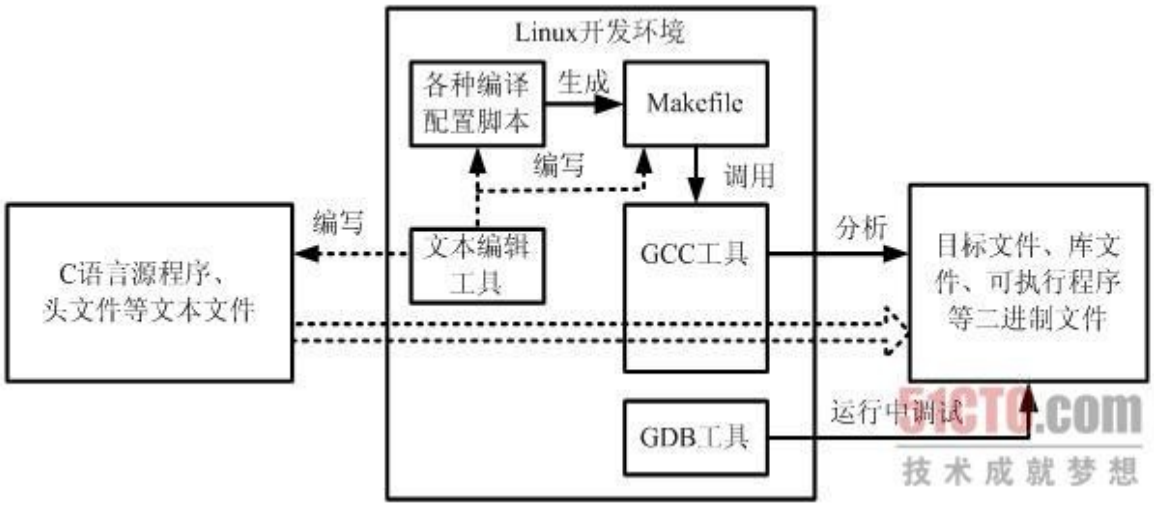
在 Linux 操作系统下的程序开发中，一般使用 make 和 Makefile 作为工程管理工具。在工程管理方面，有效地使用它们可以统筹工程中的各个文件，并在编译过程中根据时间戳，有选择地进行编译，减少程序生成时间。

1.2 在 Linux 中使用 C 语言开发

在 Linux 操作系统中，C 语言程序的开发和其他环境类似，程序生成主要分成编译、汇编、连接等几个步骤。在 Linux 中使用文本编辑工具编辑程序源代码也是程序开发的重要步骤。

1.2.1 开发流程和开发工具

C 语言程序的开发过程是：使用编辑工具编写文本形式的 C 语言源文件，然后编译生成以机器代码为主的二进制可执行程序的过程。由源文件生成可执行程序的开发过程如图 1-1 所示。



（点击查看大图）图1-1　Linux中C语言程序的开发流程

编译是指把用高级语言编写的程序转换成相应处理器的汇编语言程序的过程。从本质上讲，编译是一个文本转换的过程。对嵌入式系统而言，一般要把用C语 言编写的程序转换成处理器的汇编代码。编译过程包含了C语言的语法解析和汇编语言的生成两个步骤。汇编一般是逐个文件进行的，对于每一个C语言编写的文 件，可能还需要进行预处理。

汇编是从汇编语言程序生成目标系统的二进制代码（机器代码）的过程。机器代码的生成和处理器有密切的联系。相对于编译过程的语法解析，汇编的过程相 对简单。这是因为对于一款特定的处理器，其汇编语言和二进制的机器代码是一一对应的。汇编过程的输入是汇编代码，这个汇编代码可能来源于编译过程的输出， 也可以是直接用汇编语言书写的程序。

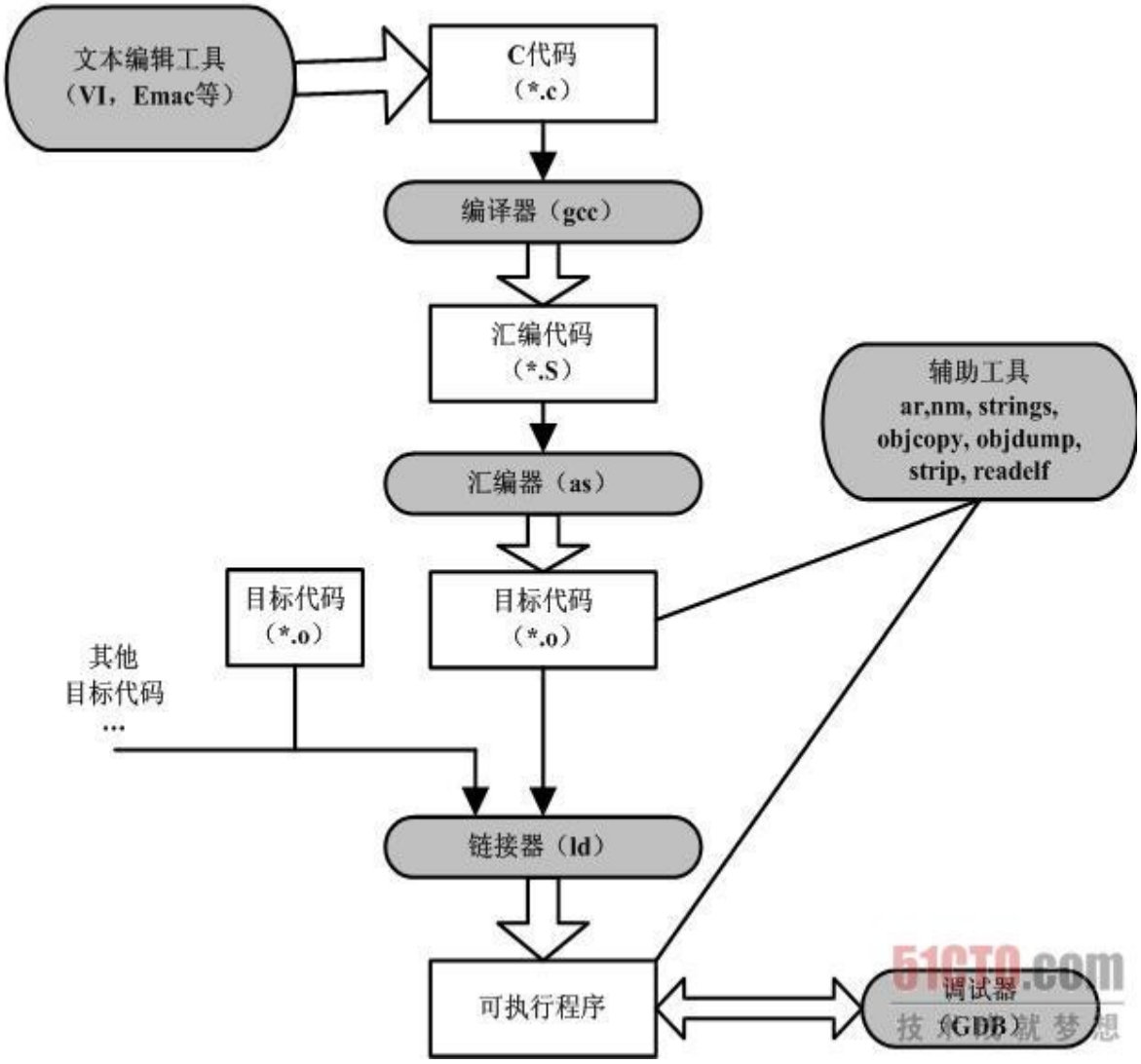
连接过程是指将汇编生成的多段机器代码组合成一个可执行程序。一般来说，通过编译和汇编过程，每一个源文件将生成一个目标文件。连接器的作用就是将这些目标文件组合起来，组合的过程包括了代码段、数据段等部分的合并，以及添加相应的文件头。

在Linux的C语言程序生成过程中，源代码经过编译–汇编–连接生成可执行程序。GCC是Linux下主要的程序生成工具，它除了编译器、汇编器、连接器外，还包括一些辅助工具。

调试是程序开发一个很重要的环节。在Linux的程序开发中，最主要的调试工具是GDB。GDB是一个命令行调试工具，可以实现在程序中设置断点、单步执行、查看对应源代码等功能。

虽然Linux中基本的开发工具GCC和GDB都是命令行工具，但是它们也可以和IDE（集成开发环境）结合使用。

Linux下程序的开发过程及相关工具的使用如图1-2所示。



1.2.2 Linux 中程序的运行原理

在 Linux 的开发环境中，C 语言程序的运行环境如图 1-3 所示。

作为 UNIX 操作系统的一种，Linux 的操作系统提供了一系列的接口，这些接口被称为系统调用（System Call）。在 UNIX 的理念中，系统调用"提供的是机制，而不是策略"。C 语言的库函数通过调用系统调用来实现，库函数对上层提供了 C 语言库文件的接 口。在应用程序层，通过调用 C 语言库函数和系统调用来实现功能。一般来说，应用程序大多使用 C 语言库函数实现其功能，较少使用系统调用。

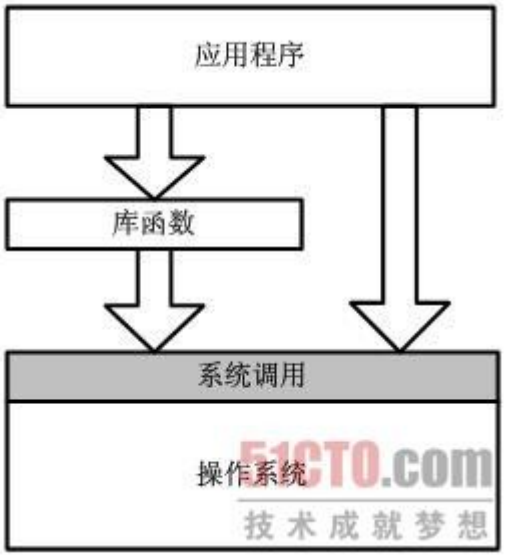


图1-3　Linux下C语言程序的结构

在Linux等系统的环境中，C语言库及其头文件都是系统的一部分，只要安装了编译工具即可以完成C语言程序的开发。这点与Windows中程序的开发有所不同，Windows中一般需要安装开发包才能进行程序开发。

C语言程序经过编译-汇编-连接，最终生成可执行程序格式。可执行程序中包含两个部分的内容:

程序头

程序主体（二进制机器代码）

在程序头中包含了供操作系统加载的信息，操作系统根据这些信息加载可执行程序。而可执行程序的主体依然是二进制的机器代码。程序在运行的时候，正是靠逐条地执行这些机器代码，形成程序运行的序列。

在Linux操作系统中，普遍使用ELF格式来作为可执行程序或者程序生成过程中的中间格式。ELF（Executable and Linking Format，可执行连接格式）是UNIX系统实验室（USL）作为应用程序二进制接口（Application Binary Interface，ABI）而开发和发布的。工具接口标准委员会（TIS）选择了正在发展中的ELF标准作为工作在32位Intel体系上不同操作系统 之间可移植的二进制文件格式。

如果开发者定义了一个二进制接口集合，ELF标准允许用它来支持流线型的软件运行。通过它可以减少不同执行接口的数量，也可以减少重新编程和重新编译的代码。

ELF文件格式包括三种主要的类型：可执行文件、可重定向文件、共享库。

1．可执行文件（应用程序）

可执行文件包含了代码和数据，是可以直接运行的程序。

2．可重定向文件（*.o）

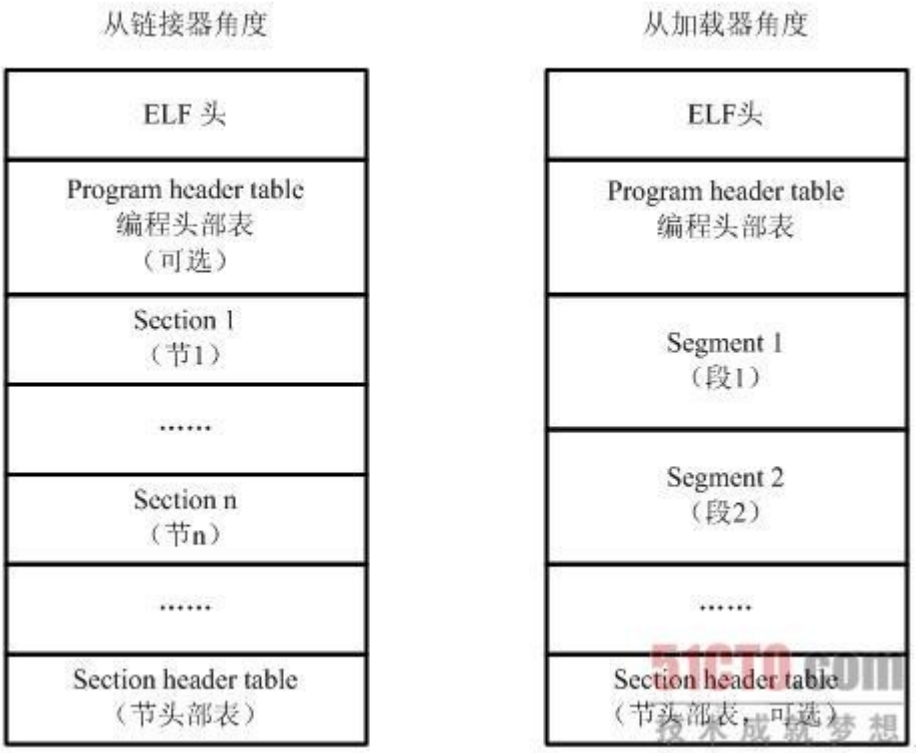
可重定向文件又称为目标文件，它包含了代码和数据（这些数据是和其他重定位文件和共享的object文件一起连接时使用的）。

3．共享文件（*.so）

也称为动态库文件，它包含了代码和数据（这些数据是在连接时候被连接器 ld 和运行时动态连接器使用的）。动态连接器可能称为 ld.so.1，libc.so.1 或者 ld-linux.so.1。

object 文件参与程序的连接（创建一个程序）和程序的执行（运行一个程序）。object 文件格式提供了一个方便有效的方法来用并行的视角看待 文件的内容，这些 object 文件的活动可以反映出不同的需要。一个 ELF 头存在于文件的开始处，在这里保存了路线图（road map），描述了该文件的组织情况。节（section）保存着 object 文件的信息，从连接角度看它包括指令、数据、符号表和重定位信息等。

ELF 文件格式如图 1-4 所示。



（点击查看大图）图 1-4　ELF 可执行程序结构

一个 ELF 文件从连接器（Linker）的角度看，是一些节的集合；从程序加载器（Loader）的角度看，它是一些段（Segments）的集合。ELF 格式的程序和共享库具有相同的结构，只是段的集合和节的集合上有些不同。

ELF 格式的共享库可以加载到任何地址。事实上，共享库使用 PIC（Place Independence Code，位置无关代码），使得文件的代码段（Text Page）不需要重定位，并且可以被多个进程共享。ELF 格式的连接器通过 GOT（Global Offset Table）来支持 PIC 代码。

5.1.1 make 机制概述

第 5 章 make 工程管理工具

本章介绍 Linux 下面的常用工程管理机制：make 和 Makefile，也介绍使用 autoconf 和 automake 生成 Makefile 的机制和方法。

在本章的学习中，读者应重点关注以下内容：

make 的工作机制

Makefile 的基本语法

使用 Makefile 典型工程

自动生成 Makefile 的流程

5.1 make 和 Makefile

make 和 Makefile 提供了一种非常简单有效的工程管理方式。使用这种方式管理工程的原理很简单：Makefile 是一个决定怎样编译工程的文本文件，有一定的书写规则。在工程更新的时候，使用 GNU 的 make 工具根据当前的 Makefile 对工程进行编译。

5.1.1 make 机制概述

在 Linux 的程序开发环境下，一般不具有集成开发环境（IDE）。因此，当需要大量编译工程文件的时候，就需要使用自己的方法来管理。如果仅仅手 动使用 gcc 的编译命令，将变得烦琐而单调，而且不利于工程管理。而如果使用 Makefile 进行工程管理，就可以较好地处理这个问题。

make 程序最初设计的目的是为了维护 C 程序文件，防止不必要的重新编译。例如，在使用命令行进行编译的时候，修改了一个工程中的头文件，如何确保包含这个头文件 的所有文件都得到编译呢？这些工作可以让 make 程序来自动完成。make 工具对于维护一些具有相互依赖关系的文件特别有用，它对文件和命令的联系（在文 件改变时调用来更新其他文件的程序）提供一套编码方法。make 工具的基本概念类似于 Proglog 语言，在使用的过程中只告诉 make 需要做什么，即提 供一些规则，其他的工作由 make 自动完成。

make 工具的工作是自动确定工程的哪部分需要重新编译，然后执行命令去编译它们。虽然这种方式多用于 C 程序，然而只要提供命令行的编译器，就可以 将其用于任何语言。实际上，make 工具不仅应用于编程，也可以用于描述一些文件改变时，需要自动更新另一些文件的任务。在程序开发的过程中， Makefile 带来的好处就是自动化编译。当编译规则制定完成后，只需要一个 make 命令，整个工程就会根据 Makefile 判断是否需要更新来完成自 动编译，极大地提高了软件开发的效率，降低了开发的复杂度。

make 机制的运行环境需要一个命令行程序 make 和一个文本文件 Makefile。

make 是一个命令工具，具体来说是一个解释 Makefile 中的指令的命令工具。Makefile 的工作原理是调用系统中的 make 命令解释当前 的 Makefile，完成其中指定的功能。在很多的 IDE 中都有这个命令，如：Delphi 的 make，Visual C++的 nmake，Linux 下 GNU 的 make。可见，Makefile 已经成为一种在工程方面的编译方法。

Makefile 里主要包含了 5 个方面的内容：显式规则、隐式规则、变量定义、文件指示和注释。

1．显式规则。显式规则说明了如何生成一个或多个目标。这需要由 Makefile 的书写者显式指出要生成的文件、文件的依赖文件及生成的命令。

2．隐式规则。由于 make 有自动推导的功能，会选择一套默认的方法进行 make，所以隐式的规则可以让开发者比较、简略地书写 Makefile，这是由 make 所支持的。

3．变量定义。在 Makefile 中需要定义一系列的变量，一般都是字符串，它类似 C 语言中的宏，当 Makefile 被执行时，其中的变量都会被扩展到相应的引用位置上。

4．文件指示。包括三个部分，第一部分是在一个 Makefile 中引用另一个 Makefile，就像 C 语言中的 include 一样包含进来；第二部 分是指根据某些情况指定 Makefile 中的有效部分，就像 C 语言中的预编译宏#ifdef 一样；第三部分就是定义一个多行的命令。

5．注释。Makefile 中只有行注释，和 UNIX 的 Shell 脚本一样，其注释符使用井号"#"字符，这个就像 C/C++中的双斜杠"//"一样。如果需要在 Makefile 中使用井号"#"字符，可以用反斜杠进行转义，如："\\#"。

5.1.2　make 和 Makefile 的使用

make 是一个 Linux 下的二进制程序，用来处理 Makefile 这种文本文件。在 Linux 的 Shell 命令行键入 make 的时候，将自动寻找 名称为"Makefile"的文件作为编译文件，如果没有名称为"Makefile"的文件，将继续查找名称为"makefile"的文件。找到编译文件 后，make 工具将根据 Makefile 中的第一个目标自动寻找依赖关系，找出这个目标所需要的其他目标。如果所需要的目标也需要依赖其他的目标， make 工具将一层层寻找直到找到最后一个目标为止。

make 工具的使用格式为：

```
make [options] [target] …
```

options 为 make 工具的选项，target 为 Makefile 中指定的目标。表 5-1 给出了 make 工具的参数选项。

表 5-1　make 工具的参数选项

选 项	含　义
-f filename	显式地指定文件作为 Makefile
-C dirname	制定 make 在开始运行后的工作目录为 dirname
-e	不允许在 Makefile 中替换环境变量的赋值
-k	执行命令出错时，放弃当前目标，继续维护其他目标
-n	按实际运行时的执行顺序模拟执行命令 (包括用@开头的命令)，没有实际执行 效果，仅仅用于显示执行过程
-p	显示 Makefile 中所有的变量和内部规则

-r	忽略内部规则
-s	执行但不显示命令，常用来检查 Makefile 的正确性
-S	如果执行命令出错就退出
-t	修改每个目标文件的创建日期
-I	忽略运行 make 中执行命令的错误
-V	显示 make 的版本号

在 Makefile 中，经常使用的变量如表 5-2 所示。

表 5-2　Makefile 中常用变量

变量	描　　述
$@	目标文件名
$<	规则中的第一个文件名
$^	规则中所有相关文件的名称
$?	规则中日期比目标新的文件列表，用空格分开
$@(D)	目标文件的目录部分
$@(F)	目标文件的文件名部分

在 Makefile 中，目标名称的指定常常有以下惯例：

all：表示编译所有的内容，是执行 make 时默认的目标。

clean：表示清除目标。

distclean：表示清除所有的内容。

install：表示进行安装的内容。

5.2　Makefile 使用示例

5.2.1　简单的 Makefile

本节的 Makefile 是一个简单的示例，它只用于显示信息，并不生成具体的目标。文件如下所示：

```
all:
@echo "+++++++ make all +++++++"

rule0:
@echo "Input = $(INPUT)"
@echo 'This Target is $@'

.PHONY : clean
clean:
```

```
@echo "------- clean ------"
```

在该文件中，每一条语句的@echo 'This Target is $@'的前面需要使用 Tab 键作为开头，这是 Makefile 书写的要求。

在命令行键入"make":

```
$ make
+++++++ make all +++++++
```

执行的结果是 Makefile 文件中的 all 规则。注意，在文件命令行中使用@，表示不在命令行显示该程序的运行输出状态，对于没有使用@标注的命令，在执行 make 的时候，命令行的内容将显示在屏幕上。

使用 clean:

```
$ make clean
------- clean ------
```

此时，make 工具根据 Makefile 中的 clean 目标执行相关的内容。

以上执行 make 和 make clean 是使用 make 工具的时候最常见的方式，make 默认执行 all 目标，make clean 表示清除目标。

对于 Makefile 中的其他目标，可以通过在命令行指定让其得到执行，如下所示:

```
$ make rule0
Input =
This Target is rule0
```

此时，明确指定执行 rule0 目标。由于使用的变量$(INPUT)没有初始值，因此打印出的内容为空。

在 Makefile 中，可以使用"变量=值"的方式，在命令行指定执行过程中变量的值。如下所示:

```
$ make rule0 INPUT=abcde
Input = abcde
This Target is rule0
```

此时由于命令行指定了 INPUT=abcde，因此在执行的过程中$(INPUT)变量的值为 abcde，在执行 echo 时输出了这个值。由此，对于同一个 Makefile 文件，其执行的结果，可以根据命令行的参数进行选择，由此实现其可配置的特性。

5.2.2 Makefile 中的依赖关系

依赖关系是 Makefile 在执行过程中的核心内容。在应用中 Makefile 不仅可以用于编译，也可用于处理其他的逻辑，本节以一个 Makefile 为例，说明在执行 make 工作中处理依赖关系的过程。

Makefile 文件如下所示：

```
all:rule0 file.o

rule0:rule1
@echo "+++++++ rule0 +++++++"
@echo 'The deps:$^'
@echo 'The target:$@'

rule1:rule2
@echo "+++++++ rule1 +++++++"
rule2:rule3
@echo "+++++++ rule2 +++++++"
rule3:
@echo "+++++++ rule3 +++++++"
file.o:
@echo "1234567890" > file.o
@echo "File path: $(@D) File name : $(@F)"
.PHONY : clean rule0 rule1 rule2 rule3

clean:
@echo "------- clean ------"
rm -f file.o
```

在这个 Makefile 的路径下，执行 make 命令：

```
$ make
```

执行后显示的结果为：

```
+++++++ rule3 +++++++
+++++++ rule2 +++++++
+++++++ rule1 +++++++
+++++++ rule0 +++++++
The deps:rule1
The target:rule0
File path: . File name : file.o
```

由于 make 没有指定选项和目标，将默认使用 Makefile 文件，并执行其中的 all 目标。在执行的过程中，首先发现 all 目标依赖于 rule0 和 file.o 两个目标，因此需要完成这两个目标的处理。对于 rule0 目标，依次寻找它的依赖关系，直到找到 rule3 目标，然后再从 rule3 目标执 行，依次执行 rule3，rule2，rule1，rule0。对于 file.o 目标，将生成 file.o 文件，它由 file.o 目标生成，内容为 "1234567890"，变量@D 表示目标的所在目录的路径，@F 表示目标的文件名。

在规则 rule0:rule1 中，使用了变量^和@，前者表示依赖的所有文件，后者表示目标的

名称。事实上，Makefile 的执行顺序不是按照每条规则书写的先后顺序，而是由规则之间的依赖关系确定的。

在 Makefile 中，将目标 clean rule0 rule1 rule2 rule3 定义为伪目标（.PHONY），这是由于它们不是需要生成的内容的名称；file.o 是实际生成的结果，因此它是真实的目标，而不是伪目标。

在执行过一次 make 之后，再次执行 make 命令，得到的结果如下所示：

```
+++++++ rule3 +++++++
+++++++ rule2 +++++++
+++++++ rule1 +++++++
+++++++ rule0 +++++++
The deps:rule1
The target:rule0
```

从执行结果中可见，这次只执行了 rule0 及其依赖的目标，没有执行目标 file.o。这是由于目标 file.o 依赖的内容没有变化，所以这条目标不需要被执行，这说明了 Makefile 条件编译的特性。

执行 make clean 的结果如下所示：

```
$ make clean
------- clean ------
rm -f file.o
```

这次执行删除了 file.o 文件，状态已经退回到 make 执行之前。因此再次执行 make 的时候，将和首次执行是一致的。

在 make 命令的使用中，可以使用-n 选项显示执行的序列：

```
$ make -n
```

本次执行的结果为：

```
echo "+++++++ rule3 +++++++"
echo "+++++++ rule2 +++++++"
echo "+++++++ rule1 +++++++"
echo "+++++++ rule0 +++++++"
echo 'The deps:rule1'
echo 'The target:rule0'
echo "1234567890" > file.o
echo "File path: . File name : file.o"
```

由此可见，在本次的执行中，只显示了需要执行的命令，而不是真正地执行这些命令。在这个过程中，寻找依赖关系的过程和直接的 make 过程是一致的，但是只显示要执行命令而不执行命令。

在使用 make 的过程中，也可以指定一条单独的目标来执行，例如：

```
$ make rule2
+++++++ rule3 +++++++
+++++++ rule2 +++++++
```

这时，将指定目标 rule2 来执行，执行的过程发现它依赖于目标 rule3，因此先执行 rule3 的内容，再执行目标 rule2 的内容。对于其他的目标则不需要执行。

5.2.3 Makefile 中使用隐含规则来编译程序

本示例演示一个程序的生成过程，使用的程序文件为第 4 章中的文件。Makefile 文件和程序文件在一个文件夹中。Makefile 文件如下所示:

```
CC      := gcc
HEAD    := getarg.h
SRC     := getarg.c writeinfo.o main.c
OBJS    := getarg.o writeinfo.o main.o
TT      := test

Files := $(wildcard ./*)

INC = .
CFLAGS  = -pipe -g -Wall -I$(INC)
LDFLAGS = -Wall -g

all:$(TT)
$(TT):$(OBJS)
@echo "+++++++ Build Standalone Programe : $@ +++++++"
$(CC) $(LDFLAGS) $(OBJS)  -o $@
libtest_d.so:getarg.o writeinfo.o
@echo "+++++++ Build Dynamic lib : $@ +++++++"
$(CC) -shared $(LDFLAGS) getarg.o writeinfo.o -o $@

test_dlib:libtest_d.so main.o
@echo "+++++++ Build Exe by Dynamic lib : $@ +++++++"
$(CC) $(LDFLAGS) main.o -L. -ltest_d -o $@

filelist:
@echo "<<<<<<< Files in this folder >>>>>>"
@file $(Files)

.PHONY : clean filelist

%.o:%c
$(CC) $(CFLAGS) -c $<  -o $@

clean:
```

```
@echo "------- clean ------"
rm -f *.o
rm -f $(TT)
rm -f libtest_d.so
rm -f test_dlib
```

在本例中，定义了CC等变量，在变量引用和使用这些变量的时候，需要用$(CC)的形式。

在Makefile和源文件所在目录中，在命令行执行make命令：

```
$ make
```

执行的结果如下所示：

```
gcc -pipe -g -Wall -I.   -c -o getarg.o getarg.c
gcc -pipe -g -Wall -I.   -c -o writeinfo.o writeinfo.c
gcc -pipe -g -Wall -I.   -c -o main.o main.c
+++++++ Build Standalone Programe : test +++++++
gcc -Wall -g getarg.o writeinfo.o main.o  -o test
```

在执行的过程中，默认执行all目标，由于all目标依赖于变量$(TT)，$(TT)实际上是test。$(TT)依赖于$(OBJS)，$(OBJS)就是getarg.o writeinfo.o main.o。因此，需要产生这三个目标文件。

上述make工作的处理过程是这样的：首先寻找三个目标文件（getarg.o，writeinfo.o和main.o）的生成规则。在所有的规则 中，并没有这三个目标文件的生成规则，因此使用默认的目标%.o:%c中的规则生成这三个目标文件。这个时候会使用gcc编译生成这三个目标文件。生成完 三个目标文件之后，将执行test目标，进行目标文件的连接。

事实上，上述执行过程中只是直接执行了all目标，在Makefile中还有libtest_d.so、test_dlib和filelist几个目标没有执行，而这些目标可以单独执行。

执行单独的目标filelist：

```
$ make filelist
```

显示的结果如下：

```
<<<<<<< Files in this folder >>>>>>
./getarg.c:        ASCII C program text
./getarg.h:        ASCII text
./getarg.o:        ELF 32-bit LSB relocatable, Intel
80386, version 1 (SYSV), not stripped
./main.c:         ASCII C program text
./main.o:         ELF 32-bit LSB relocatable, Intel
80386, version 1 (SYSV), not stripped
./Makefile:       ASCII make commands text
./test:           ELF 32-bit LSB executable, Intel
```

```
80386, version 1 (SYSV), for GNU/Linux 2.6.4,
dynamically linked (uses shared libs),
for GNU/Linux 2.6.4, not stripped
./writeinfo.c:         ASCII C program text
./writeinfo.h:         ASCII text
./writeinfo.o:         ELF 32-bit LSB relocatable,
Intel 80386, version 1 (SYSV), not stripped
```

这条目标执行的命令是使用 file 命令查看本文件夹下所有的文件。其中，Files := $(wildcard ./*)表示使用通配符寻找目录下的所有文件。

执行生成可执行程序 test_dlib 的命令：

```
$ make test_dlib
```

执行的结果如下所示：

```
+++++++ Build Dynamic lib : libtest_d.so +++++++
gcc -shared -Wall -g getarg.o writeinfo.o -o libtest_d.so
+++++++ Build Exe by Dynamic lib : test_dlib +++++++
gcc -Wall -g main.o -L. -ltest_d -o test_dlib
```

test_dlib 目标是一个可执行程序，它本身需要连接一个动态库 libtest_d.so，因此它依赖于目标 libtest_d.so 和 main.o 目标，由于 main.o 已经生成，这样还需要生成 libtest_d.so 目标。在 libtest_d.so 目标中，依赖的文件 getarg.o 和 writeinfo.o 都已经生成了，因此直接生成这个动态库即可。libtest_d.so 生成后，再生成 test_dlib 可执行程序。

在以上的示例中的 test 和 test_dlib 都是可执行的程序，它们的区别在于前者包含了三个目标文件，可以直接执行，后者只包括了 main.o 一个目标文件，它的执行必须依赖动态库。

继续使用 clean 清除目标：

```
$ make clean
```

执行的结果如下所示：

```
------- clean ------
rm -f *.o
rm -f test
rm -f libtest_d.so
rm -f test_dlib
```

在清除目标之后，生成 test_dlib 可执行程序：

```
$ make test_dlib
gcc -pipe -g -Wall -I.   -c -o getarg.o getarg.c
```

```
gcc -pipe -g -Wall -I.   -c -o writeinfo.o writeinfo.c
+++++++ Build Dynamic lib : libtest_d.so +++++++
gcc -shared -Wall -g getarg.o writeinfo.o -o libtest_d.so
gcc -pipe -g -Wall -I.   -c -o main.o main.c
+++++++ Build Exe by Dynamic lib : test_dlib +++++++
gcc -Wall -g main.o -L. -ltest_d -o test_dlib
```

在这次执行的过程中，由于 getarg.o,writeinfo.o 和 main.o 三个目标文件还没有生成，因此在生成库 libtest_d.so 之前，需要先编译生成 getarg.o 和 writeinfo.o 两个目标，它们使用的是默认的规则。在 libtest_d.so 生成后，还需要生成 main.o 的过程，它也需要使用默认的规则。

知识点：通常情况下，为了加速开发，程序员自己编写的程序在无特殊要求下都可以使用隐含规则，这样还可以防止因为 Makefile 编写错误而导致程序运行错误。

5.2.4 Makefile 中指定依赖关系的编译

仍以第 4 章的程序文件为例，本节介绍另外的一种形式的 Makefile，在这个 Makefile 中指定了各个目标的依赖关系。该文件如下所示：

```
CC      := gcc

HEAD    := getarg.h writeinfo.h
SRC     := getarg.c writeinfo.o main.c
OBJS    := getarg.o writeinfo.o main.o
TT      := test

INC = .
CFLAGS  = -pipe -g -Wall -I$(INC)
LDFLAGS = -Wall -g

all:$(TT)
$(TT):$(OBJS)
@echo "+++++++ Build Standalone Programe : $@ +++++++"
$(CC) $(LDFLAGS) $(OBJS)  -o $@
main.o:main.c getarg.h writeinfo.h
$(CC) $(CFLAGS) -c $<  -o $@

getarg.o:getarg.c getarg.h
$(CC) $(CFLAGS) -c $<  -o $@

writeinfo.o:writeinfo.c writeinfo.h
$(CC) $(CFLAGS) -c $<  -o $@
.PHONY : clean

clean:
```

```
@echo "------- clean ------"
rm -f *.o
rm -f $(TT)
```

在这个 Makefile 文件中没有使用默认的规则，而是对每一个目标文件实现单独的规则和依赖的文件。

在 Makefile 文件所在目录下执行 make 命令：

```
$ make
```

执行的结果如下所示：

```
gcc -pipe -g -Wall -I. -c getarg.c  -o getarg.o
gcc -pipe -g -Wall -I. -c writeinfo.c  -o writeinfo.o
gcc -pipe -g -Wall -I. -c main.c  -o main.o
+++++++ Build Standalone Programe : test +++++++
gcc -Wall -g getarg.o writeinfo.o main.o  -o test
```

上面执行的结果依然是先编译生成目标文件，然后连接生成可执行程序。实际上，这个执行过程依次执行了 getarg.o，writeinfo.o，main.o 和 test 的规则。

在目标生成之后，如果再次使用 make 命令，将显示以下的内容：

```
$ make
make: Nothing to be done for 'all'.
```

此时由于目标依赖的文件均没有更改，因此没有什么需要做的。更新 main.c 文件，继续执行 make 命令：

```
$ touch main.c
$ make
```

结果如下所示：

```
gcc -pipe -g -Wall -I. -c main.c  -o main.o
+++++++ Build Standalone Programe : test +++++++
gcc -Wall -g getarg.o writeinfo.o main.o  -o test
```

在执行的过程中，可以发现先后执行了 main.o 和 test 目标中的规则。这是由于 main.o 目标依赖于 main.c 文件，因此 main.c 更新 后，这个目标就需要重新生成。 getarg.o 和 writeinfo.o 目标的规则是不需要执行的，因为它们依赖的文件没有更新。

下面对 getarg.h 文件进行更新，然后重新生成，命令如下：

```
$ touch getarg.h
$ make
```

执行结果如下所示：

```
gcc -pipe -g -Wall -I. -c getarg.c  -o getarg.o
gcc -pipe -g -Wall -I. -c main.c  -o main.o
+++++++ Build Standalone Programe : test +++++++
gcc -Wall -g getarg.o writeinfo.o main.o  -o test
```

在这次执行的过程中，由于 getarg.o 和 main.o 目标都依赖 getarg.h 文件，因此，两个目标的规则都需要重新执行。

执行清除命令 clean:

```
$ make clean
```

执行的结果如下所示:

```
------- clean ------
rm -f *.o
rm -f test
```

这里 make 工具执行的是 clean 伪目标，删除了目标文件和可执行程序。

知识点：make 和 Makefile 工程管理工具，可以使用隐含规则进行编译，也可以由 Makefile 编写者制定自己特定的编译规则。

5.3 自动生成 Makefile

5.3.1 自动生成 Makefile 的意义和相关工具

在实际的项目中，由于 make 规则的复杂性和不确定性，自己编写 Makefile 是一件费时费力的事情。Makefile 本身具有一定的相似性，因 此利用 GNU autoconf 及 automake 这两套工具可以协助我们自动产生 Makefile 文件，并且让开发出来的软件可以像大多数源代码包那样，只需运行命令 "./configure"、"make"、"make install"就可以把程序安装到系统中，对于各种源代码包的分发和兼容性具有很好的效果。

1．autoconf 工具介绍

autoconf 是一个用于产生可以自动配置源代码包，生成 Shell 脚本的工具，它可以适应各种类 UNIX 系统的需要。autoconf 产生的配置脚本在运行时独立于 autoconf，也就是说使用这些脚本的用户不需要安装 autoconf。

autoconf 生成的配置脚本通常名称是 configure，得到这个文件，通常需要以下的依赖文件:

configure.in 文件：生成 configure 的必需文件，需要手动编写。

aclocal.m4 和 acsite.m4 文件：在编写了除 autoconf 提供的测试外的其他测试补充的时候，才会用到这两个文件，也需要手动编写。

acconfig.h 文件：如果使用了含有#define 指令的头文件，则需要自行编写该文件，一般都需要使用，这个时候会生成另外一个 config.h.in 文件，这个文件需要和软件包一同发布。

总之，在 autoconf 运行完毕后，得到两个需要和软件包同时发布的文件： configure 和

config.h.in，当然 config.h.in 可以不存在。

2．automake 工具介绍

automake 是一个从文件 Makefile.am 自动生成 Makefile.in 的工具。每个 Makefile.am 基本上是一系列 make 的宏定义（make 规则也会偶尔出现）。生成的 Makefile.in 也服从 GNU Makefile 标准。

典型的 automake 输入文件是一系列简单的宏定义。处理所有相关的文件并创建 Makefile.in 文件。在一个项目的每个目录中通常仅包含一个 Makefile.am。

目前 automake 支持三种目录层次：平坦模式（flat）、混合模式（shallow）和深层模式（deep）。

（1）平坦模式指的是所有文件都位于同一个目录中。就是所有源文件、头文件及其他库文件都位于当前目录中，且没有子目录。

（2）混合模式指的是主要的源代码都存储在顶层目录，其他各个部分则存储在子目录中。也就是主要源文件在当前目录中，而其他一些实现各部分功能的源文件位于各自不同的目录。

（3）深层模式指的是所有源代码都被存储在子目录中；顶层目录主要包含配置信息。也就是所有源文件及程序员自己写的头文件都位于当前目录的一个子目录中，而当前目录里没有任何源文件。

在这三种支持的目录层次中，平坦模式类型是最简单的，深层模式类型是最复杂的。但是这些模式使用 autoconf 和 automake 所遵循的基本原则和流程是一样的。

3．其他工具介绍

（1）autoheader：能够产生供 configure 脚本使用的 C #define 语句模板文件。

（2）autom4te：对文件执行 GNU M4。

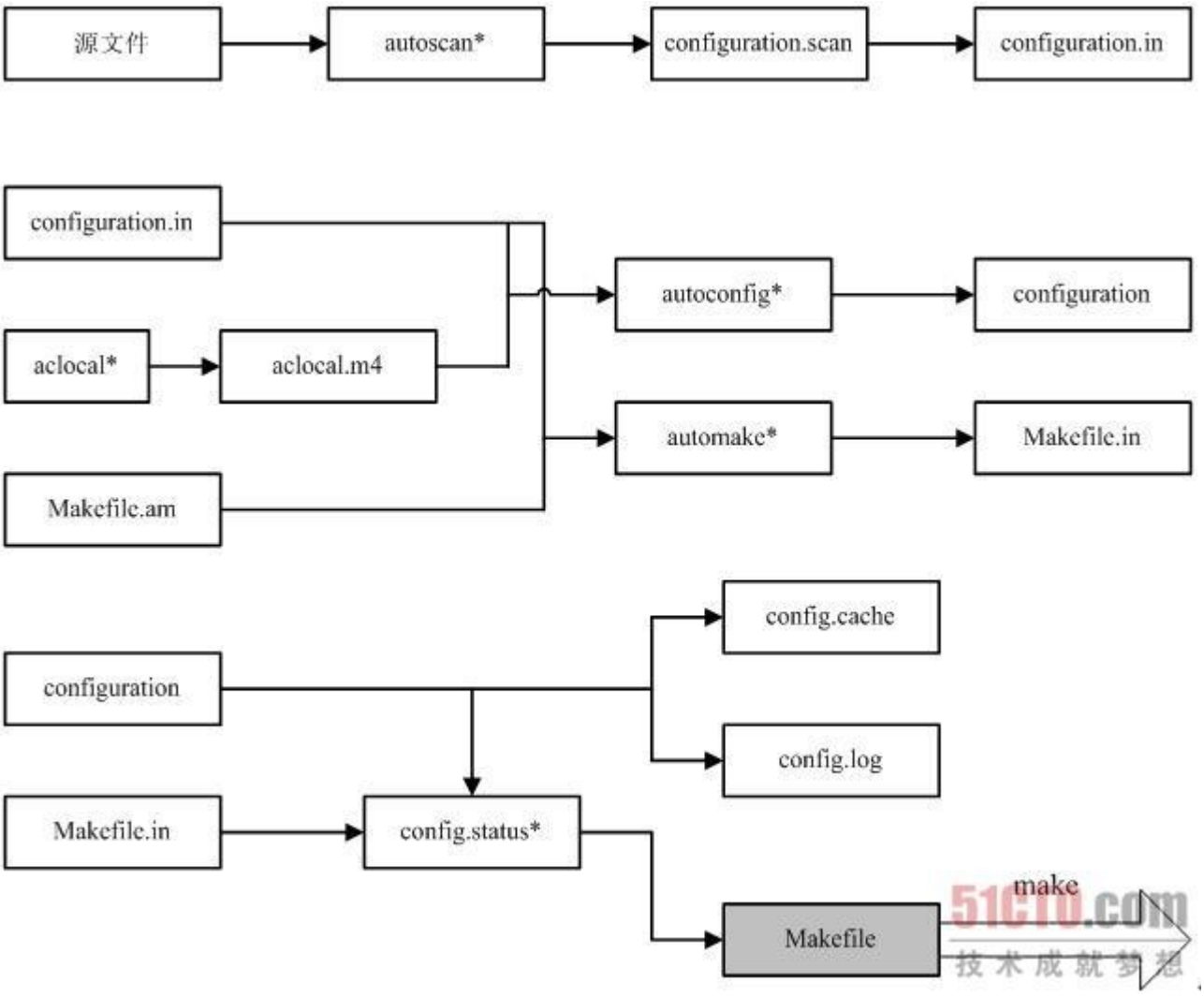
（3）autoreconf：如果有多个 autoconf 产生的配置文件，autoreconf 可以保存一些相似的工作，它通过重复运行 autoconf（以及在合适的地方运行 autoheader）以重新产生 autoconf 配置脚本和配置头模板，这些文件保存在以当前目录为根的目录树 中。

（4）autoscan：该程序可以用来为软件包创建 configure.in 文件。autoscan 在以命令行参数中指定的目录为根（如果未给定 参数，则以当前目录为根）的目录树中检查源文件。为软件包创建一个 configure.scan 文件，该文件就是 configure.in 的前身。

（5）autoupdate：该程序的作用是转换 configure.in，从而使用新的宏名。

5.3.2 自动生成 Makefile 的流程

在进行自动化生成 Makefile 之前，务必要设定好工作的根目录，在当前环境下，至少要保证 autoscan、autoconf、 aclocal、automake 这些命令能够正常运行。在这一节中，我们就以一个最简单的示例来说明 automake 和 autoconf 的基本使用方 法，这个例子是一个平坦模式的模型。自动生成 Makefile 的流程如图 5-1 所示。



（点击查看大图）图 5-1　自动生成 Makefile 的流程

在这里使用标准的 sntp 客户端源代码进行示例，该源代码只有一个源文件和一个头文件，分别是 ntp.h 和 sntp.c。

（1）首先把所有的源代码文件复制到当前的根目录下，然后运行 autoscan 扫描所有的代码，并得到 autoscan.log 和 configure.scan 两个文件。

其中 autoscan.log 是一个空文件，而 configure.scan 文件则需要用户手动进行编辑，在该例中，修改之前 configure.scan 的内容应该如下：

```
#-*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.57)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([sntp.c])
AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CC
```

```
# Checks for libraries.
# Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS([netdb.h netinet/in.h stdlib.h
string.h sys/socket.h sys/time.h unistd.h])
# Checks for typedefs, structures, and compiler characteristics.
AC_HEADER_TIME
AC_STRUCT_TM
# Checks for library functions.
AC_FUNC_ERROR_AT_LINE
AC_FUNC_MALLOC
AC_FUNC_SELECT_ARGTYPES
AC_CHECK_FUNCS([bzero gethostbyname gettimeofday select socket
strchr])
AC_OUTPUT
```

这是一个标准的 config 模板，后面针对 configure 文件的修改都是基于这个文件进行的，configure.scan 的基本格式如下：

```
AC_INIT
测试程序 AC_PROG_CC
测试函数库 （在源文件中没有用到函数库）
测试头文件 AC_HEADER_STDC, AC_CHECK_HEADERS
测试类型定义 AC_HEADER_TIME
测试结构 AC_STRUCT_TM
测试编译器特性
测试库函数 AC_FUNC_ERROR_AT_LINE, AC_FUNC_MALLOC
AC_FUNC_SELECT_ARGTYPES
测试系统调用 AC_CHECK_FUNCS
AC_OUTPUT
```

这个文件中有几个非常重要的语句，一般来说所有的 configure.scan 文件都是以 AC_INIT 开头和以 AC_OUTPUT 结束的；而且中 间的顺序一般不要进行随意的改变，因为通常在本列表中靠后的项目依赖于表中靠前的项目。例如，库函数可能受到 typedefs 和库的影响。

（2）configure.scan 文件准备好之后，就开始准备把它改造成自己定制的 configure.in 文件，先把它的名字修改为 configure.in，然后打开该文件进行编辑。在 configure.in 文件中，必须修改的内容有：

将 AC_CONFIG_HEADER([config.h])修改为 AM_CONFIG_HEADER([config.h])，也就是说把 AC 改成 AM。

在 AM_CONFIG_HEADER 下面添加一行 AM_INIT_AUTOMAKE(sntp，1.0)，这一行命令说明了使用 automake 最终要得到的结果和版本号。

如果工程中使用了外部的库，比如 pthread 线程库，在# Checks for libraries 这一行

的下面还会生成像下面这样的行：

```
AC_CHECK_LIB([pthread],[pthread_rwlock_init])
```

最后，还要在 AC_OUTPUT 后面加上要创建的文件名称：AC_OUTPUT([Makefile])。

如果是混合模式或者是深层模式的结构，还需要添加子目录的目标 Makefile 文件路径到 AC_OUTPUT 后面的输入中。

此时，configure.in 文件的最终内容应该如下：

```
#-*- Autoconf -*-
# Process this file with autoconf to produce a configure
script.
AC_PREREQ(2.57)
AC_INIT(sntp, 1.0, author@gmail.com)
AC_CONFIG_SRCDIR([sntp.c])
AM_CONFIG_HEADER([config.h])
AM_INIT_AUTOMAKE(sntp, 1.0)
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS([netdb.h netinet/in.h stdlib.h
string.h sys/socket.h sys/time.h unistd.h])
# Checks for typedefs, structures, and compiler
characteristics.
AC_HEADER_TIME
AC_STRUCT_TM
# Checks for library functions.
AC_FUNC_ERROR_AT_LINE
AC_FUNC_MALLOC
AC_FUNC_SELECT_ARGTYPES
AC_CHECK_FUNCS([bzero gethostbyname
gettimeofday select socket strchr])
AC_OUTPUT
```

完成了 configure.in 文件的编写后，还有一个文件需要用户手动编写，就是 Makefile.am 文件，针对本工程的一个标准的简易 Makefile.am 模板如下：

```
AUTOMAKE_OPTIONS = foreign

# Note:target part
bin_PROGRAMS = sntp
```

```
# noinst_PROGRAMS = sntp

# Note:source part
sntp_SOURCES = sntp.c
# sntp_LDADD =
# sntp_LDFLAGS =
# sntp_DEPENDENCIES =

# Note:lib part
# lib_LIBRARIES =
# sntp_a_SOURCES =
# sntp_a_LDADD =
# sntp_a_LIBADD =
# sntp_a_LDFLAGS =

# Note:header part
include_HEADERS = ntp.h

# Note:data part
# data_DATA =
```

可以看到 Makefile.am 文件主要由五个部分组成，分别是目标描述部分、源代码描述部分、库描述部分、头描述部分和数据描述部分。在本例中，由于只有源代码文件和头文件，没有数据和依赖的库，所以对不用的段用井号"#"进行注释。

在目标描述部分，一般有两种选择，如果描述为 bin_PROGRAMS，则表明该程序是需要进行安装的，而如果描述为 noinst_PROGRAMS，则表明不需要安装。

一般来说，如果不显式地进行声明，默认的几个全局路径如下：

安装路径前缀：$(prefix) = /usr/local

目标文件安装路径：bindir = $(prefix)/bin

库文件安装路径：libdir = $(prefix)/lib

数据文件安装路径：datadir = $(prefix)/share

系统配置安装路径：sysconfdir = $(prefix)/etc

写完了 Makefile.am 文件之后，生成 Makefile 的前期准备工作就做完了。接下来就是使用 auto 系列工具生成的过程，按照如下顺序输入命令：

```
$./aclocal  # 得到 aclocal.m4 文件
$./autoconf  # 得到 configure 文件
$./automake -a # 得到 Makefile.in 文件
```

到此为止，就完成了所有自动化的配置任务，把生成的一系列相关文件压缩打包，就可以进行版本的发布了。

用户拿到这个安装包解压后，就可以按照一般软件包的方式进行 Makefile 的最后生成、编译和安装，也就是输入以下命令：

```
$ ./configure     （得到 makefile 文件）
$ make
$ make install
```

知识点：使用 autoconf 和 automake 来进行自动化配置和生成 Makefile 的流程可以概括如下：

（1）运行 autoscan 命令。

（2）将 configure.scan 文件重命名为 configure.in，并修改 configure.in 文件。

（3）运行 aclocal 命令得到 aclocal.m4 文件。

（4）运行 autoconf 命令得到 configure 文件。

（5）在工程目录下新建 Makefile.am 文件，如果存在子目录，子目录中也要创建此文件。

（6）将/usr/share/automake-1.X/目录下的 depcomp 和 compile 文件复制到需要处理目录下。

（7）运行 automake -a 命令得到 Makefile.in 文件。

（8）运行./configure 脚本（这一步已经属于使用自动化管理的范畴了）。

第 13 章 C 语言程序的内存布局

本章介绍 C 语言程序的内存布局结构，包括连接过程中目标程序各个段的组成和运行过程中各个段加载的情况。

在本章的学习中，读者应重点关注以下内容：

C 语言程序在内存中各个段的组成

C 语言程序连接过程中的特性和常见错误

C 语言程序的运行方式

13.1 C 语言程序的存储区域

由 C 语言代码（文本文件）形成可执行程序（二进制文件），需要经过编译-汇编-连接三个阶段。编译过程把 C 语言文本文件生成汇编程序，汇编过程把汇编程序形成二进制机器代码，连接过程则将各个源文件生成的二进制机器代码文件组合成一个文件。

C 语言编写的程序经过编译-连接后，将形成一个统一文件，它由几个部分组成。在程序运行时又会产生其他几个部分，各个部分代表了不同的存储区域：

1．代码段（Code 或 Text）

代码段由程序中执行的机器代码组成。在 C 语言中，程序语句进行编译后，形成机器代码。在执行程序的过程中，CPU 的程序计数器指向代码段的每一条机器代码，并由处理器依

次运行。

2．只读数据段（RO data）

只读数据段是程序使用的一些不会被更改的数据，使用这些数据的方式类似查表式的操作，由于这些变量不需要更改，因此只需要放置在只读存储器中即可。

3．已初始化读写数据段（RW data）

已初始化数据是在程序中声明，并且具有初值的变量，这些变量需要占用存储器的空间，在程序执行时它们需要位于可读写的内存区域内，并具有初值，以供程序运行时读写。

4．未初始化数据段（BSS）

未初始化数据是在程序中声明，但是没有初始化的变量，这些变量在程序运行之前不需要占用存储器的空间。

5．堆（heap）

堆内存只在程序运行时出现，一般由程序员分配和释放。在具有操作系统的情况下，如果程序没有释放，操作系统可能在程序（例如一个进程）结束后回收内存。

6．栈（stack）

栈内存只在程序运行时出现，在函数内部使用的变量、函数的参数以及返回值将使用栈空间，栈空间由编译器自动分配和释放。

C 语言目标文件的内存布局如图 13-1 所示。

（点击查看大图）图 13-1　C 语言目标文件的内存布局

代码段、只读数据段、读写数据段、未初始化数据段属于静态区域，而堆和栈属于动态区域。代码段、只读数据段和读写数据段将在连接之后产生，未初始化数据段将在程序初始化的时候开辟，而堆和栈将在程序的运行中分配和释放。

C 语言程序分为映像和运行时两种状态。在编译-连接后形成的映像中，将只包含代码段（Text）、只读数据段（RO Data）和读写数据段（RW Data）。在程序运行之前，将动态生成未初始化数据段（BSS），在程序的运行时还将动态形成堆（Heap）区域和栈（Stack）区域。

一般来说，在静态的映像文件中，各个部分称之为节（Section），而在运行时的各个部分称之为段（Segment）。如果不详细区分，可以统称为段。

知识点：C 语言在编译和连接后，将生成代码段（Text）、只读数据段（RO Data）和读写数据段（RW Data）。在运行时，除了以上三个区域外，还包括未初始化数据段（BSS）区域和堆（Heap）区域和栈（Stack）区域。

13.2 C 语言程序的段

13.2.1 段的分类

根据 C 语言的特点，每一个源程序生成的目标代码将包含源程序所需要表达的所有信息和功能。目标代码中各段生成情况如下：

1．代码段（Code）

代码段由程序中的各个函数产生，函数的每一个语句将最终经过编译和汇编生成二进制机器代码（具体生成哪种体系结构的机器代码由编译器决定）。

顺序代码

基本数学运算（+，-），逻辑运算（&&，‖），位运算（&，|，^）等都属于顺序代码。

选择代码

if，if…else 语句等将由编译器生成选择代码。

循环代码

while()，do…while()语句等将由编译器生成循环代码。

对于一些较为复杂的数学运算如除法（\），取余（%）等，虽然它们是 C 语言的基本运算，但在各种编译系统中的处理方式却不一定相同。根据编译器和体 系结构的特点，对它们的处理方式有可能与加减等运算相同，即直接生成处理器的机器代码，也有可能转换成一个库函数的调用。例如，在没有除法指令的体系结构 中，编译器在编译 a/b 这类除法运算的时候，由于处理器没有与其对应的指令，因此会使用调用库函数来模拟除法运算。浮点数的处理与之类似：对于支持浮点运 算的体系结构，将直接生成浮点代码；对于不支持浮点数的处理器，编译器将会把每一个浮点运算用库函数调用的方式模拟。

2．只读数据段（RO Data）

只读数据段由程序中所使用的数据产生，该部分数据的特点是在运行中不需要改变，因此编译器会将该数据放入只读的部分中。C 语言的一些语法将生成只读数据段。

只读全局量

例如：定义全局变量 const char a[100]={"ABCDEFG"}将生成大小为 100 个字节的只读数据区，并使用字串"ABCDEFG"初始化。如果定义为 const char a[]={"ABCDEFG"}，没有指定大小，将根据"ABCDEFG"字串的长度，生成 8 个字节的只读数据段。

只读局部量

例如：在函数内部定义的变量 const char b[100] ={"9876543210"};其初始化的过程和全局量一样。

程序中使用的常量

例如：在程序中使用 printf("information \n")，其中包含了字串常量，编译器会自动把常量"information \n"放入只读数据区。

在 const char a[100]={"ABCDEFG"}中，定义了 100 个字节的数据区，但是只初始化了前面的 8 个字节（7 个字符和表示结束的'\0'）。在这种用法中， 实际后面的字节没有初始化，但是在程序中也不能写，实际上没有任何用处。因此，在只读数据段中，一般都需要做完全的初始化。

3．读写数据段（RW Data）

读写数据段表示了在目标文件中一部分可以读也可以写的数据区，在某些场合它们又被称为已初始化数据段。这部分数据段和代码段，与只读数据段一样都属于程序中的静态区域，但是具有可写的特点。

已初始化全局静态变量

例如：在函数外部，定义全局的变量 char a[100]={"ABCDEFG"}

已初始化局部静态变量

例如：在函数中定义 static char b[100] ={"9876543210"}。函数中由 static 定义并且已经初始化的数据和数组将被编译为读写数据段。

读写数据区的特点是必须在程序中经过初始化，如果只有定义，没有初始值，则不会生成读写数据区，而会定位为未初始化数据区（BSS）。如果全局变量 （函数外部定义的变量）加入 static 修饰符，写为类似 static char a[100]的形式，这表示只能在文件内使用，而不能被其他文件使用。

4．未初始化数据段（BSS）

未初始化数据段常被称之为 BSS（英文 Block Start by Symbol 的缩写）。与读写数据段类似，它也属于静态数据区，但是该段中的数据没有经过初始化。因此它只会在目标文件中被标识，而不会真正称为目标文件 中的一个段，该段将会在运行时产生。未初始化数据段只有在运行的初始化阶段才会产生，因此它的大小不会影响目标文件的大小。

在 C 语言的程序中，对变量的使用还有以下几点需注意：

1．在函数体中定义的变量通常是在栈上，不需要在程序中进行管理，由编译器处理。

2．用 malloc，calloc，realloc 等分配内存的函数所分配的内存空间在堆上，程序必须保证在使用后用 free 释放，否则会发生内存泄漏。

3．所有函数体外定义的是全局变量，加了 static 修饰符后的变量不管在函数内部或者外部都存放在全局区（静态区）。

4．使用 const 定义的变量将放于程序的只读数据区。

在 C 语言中，可以定义 static 变量：在函数体内定义的 static 变量只能在该函数体内有效；在所有函数体外定义的 static 变量，也只能在 该文件中有效，不能在其他的源文件中使用；对于没有使用 static 修饰的全局变量，可以在其他的源文件中使用。这些区别是编译的概念，即如果不按要求使 用变量，编译器会报错。使用 static 和没有使用 static 修饰的全局变量最终都将放置在程序的全局区（静态区）。

3.2.2 程序中段的使用

本小节使用简单的例子，说明C语言中变量和段的对应关系。C语言程序中的全局区（静态区），实际对应着下述几个段：

只读数据段：RO Data

读写数据段：RW Data

未初始化数据段：BSS Data

一般来说，直接定义的全局变量在未初始化数据区，如果该变量有初始化则是在已初始化数据区（RW Data），加上const修饰符将放置在只读区域（RO Data）。

示例1：

```
const char ro[]={"this is readonly data"}; /* 只读数据段 */
static char rw1[]={"this is global readwrite data"};
 /* 已初始化读写数据段 */
char bss_1[100];         /* 未初始化数据段 */
const char* ptrconst = "constant data"; /* "constant data"放在只读数据段 */
int main()
{
short b;        /* b放置在栈上，占用2个字节 */
char a[100];     /* 需要在栈上开辟100个字节,a的值是其首地址 */
char s[] = "abcde";    /* s在栈上，占用4个字节 */
/* "abcde "本身放置在只读数据存储区，占6字节 */
char *p1;        /* p1在栈上，占用4个字节   */
char *p2 = "123456";    /* "123456"放置在只读数据存储区，占7字节 */
  /* p2在栈上，p2指向的内容不能更改。 */
static char rw2[]={"this is local readwrite data"};
 /* 局部已初始化读写数据段 */
static char bss_2[100];  /* 局部未初始化数据段 */
static int c = 0;     /* 全局（静态）初始化区 */
p1= (char *)malloc(10*sizeof(char));
/* 分配的内存区域在堆区。 */
strcpy(p1, "xxxx");    /* "xxxx"放置在只读数据存储区，占5字节 */
free(p1);       /* 使用free释放p1所指向的内存 */
return 0;
}
```

示例1程序中描述了C语言源文件中语句如何转换成各个段。

只读数据段需要包括程序中定义的const型的数据（如：const char ro[]），还包括程序中需要使用的数据如"123456"。对于const char ro[]和const char* ptrconst的定义，它们指向的内存都位于只读数据区，其指向的内容都不允许修改。区别在于前者不允许在程序中修改ro的值，后者允许在程序中修改 ptrconst本身的值。对于后者，改写成以下的形式，将不允许在程序中修改ptrconst本身的值：

```
const char* const ptrconst = "constant data";
```

读 写数据段包含了已经初始化的全局变量 static char rw1[]以及局部静态变量 static char rw2[]。rw1 和 rw2 的差别只是在于编译时，是在函数内部使用的还是可以在整个文件中使用。对于前者，static 修饰在于控制程序的其他文件是否 可以访问 rw1 变量，如果有 static 修饰，将不能在其他的 C 语言原文件中使用 rw1，这种影响针对编译-连接的特性，但无论有无 static，变量 rw1 都将被放置在读写数据段。对于后者 rw2，它是局部的静态变量，放置在读写数据区;如果不使用 static 修饰，其意义将完全改变，它将会是开辟在 栈空间局部变量，而不是静态变量。在这里，rw1 和 rw2 后面的方括号（[]）内没有数值，表示静态区的大小由后面字符串的长度决定，如果将它们的定义改 写为以下形式:

```
static char rw1[100]={"that is global readwrite data"};
static char rw2[200]={"that is local readwrite data"};
```

则读写数据区域的大小直接由后面的数值（100 或者 200）决定，超出字符串长度的空间的内容为 0。

未初始化数据段，示例 1 中的 bss_1[100]和 bss_2[200]在程序中代表未初始化的数据段。其区别在于前者是全局的变量，在所有文件中 都可以使用；后者是局部的变量，只在函数内部使用。未初始化数据段不设置后面的初始化数值，因此必须使用数值指定区域的大小，编译器将根据大小设置 BBS 中需要增加的长度。

栈空间包括函数中内部使用的变量如 short b 和 char a[100]，以及 char *p1 中 p1 这个变量的值。

变量 p1 指向的内存建立在堆空间上，栈空间只能在程序内部使用，但是堆空间（例如 p1 指向的内存）可以作为返回值传递给其他函数处理。示例 1 中各段的使用如表 13-1 所示。

表 13-1　示例 1 中各段的使用

段	使用情况	列　　表
只读数据（RO data）	程序中不需要更改的全局变量	const char ro[]={"it is readonly data"}; char s[] = "abcde";中的"abcde"，占用 6 字节 char *p2 = "123456";中的"123456"，占用 7 字节 strcpy(p1, "xxxx");中的"xxxx"，占用 5 字节
读写数据（RW data）	程序中需要更改，但是具有初始化值的全局变量	static char rw1[]={"that is global readwrite data"}; static char rw2[]={"that is local readwrite data"};
未初始化数据（BSS）	没有初始化值的全局变量	char bss_1[100]; char bss_2[100];

堆（heap）	需要用户分配的空间	p1=（char *)malloc(10*sizeof(char)); p1 指向的内存空间
栈（stack）	函数中临时变量 函数入口和返回值	函数中的变量： short b; char a[100]; char *p1; 变量 p1 本身占用 4 字节

示例 2：

```
long fun(long c)
{
char a[100];                                    /* 栈上的 100 个字节 *
/
long b;                                          /* 栈上的 4 个字节
*/
/* ..... */
return b;
}
```

栈空间主要用于以下 3 数据的存储：

函数内部的动态变量

函数的参数

函数的返回值

栈空间主要的用处是供函数内部的动态变量使用，变量的空间在函数开始之前开辟，在函数退出后由编译器自动回收。

栈空间的另外一个作用是在程序进行函数调用的时候进行函数的参数传递以及保存函数返回值。

在示例 2 程序的调用过程中，需要使用栈保存临时变量。在调用之前，需要将变量 c 保存在堆栈上，占用 4 字节，在函数返回之前，需要将返回值 b 保存在栈上，也占用 4 个字节的空间。同时，为数组 a 开辟 100 字节的栈空间。

在函数的调用过程中，如果函数调用的层次比较多，所需要的栈空间也逐渐增大。对于参数的传递和返回值，如果使用较大的结构体，在使用的栈空间也会比较大。

13.3 可执行程序的连接

13.3.1 可执行程序的组成

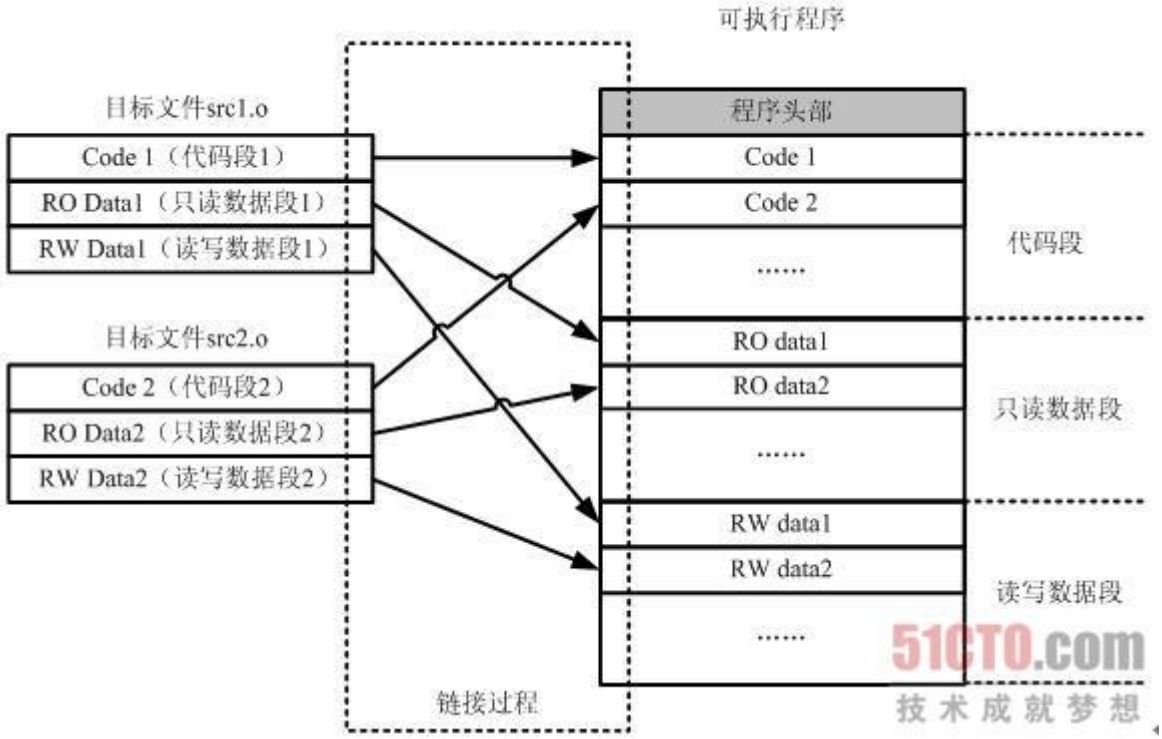
上一节分析了 C 语言应用程序中各段的情况，实际的 C 语言可执行程序，将由各个文件经过连接生成。目标文件是由每一个 C 语言源程序（*.c）经过编译 器生成，目标文件（.o）的主要组成部分即代码段、只读数据段和读写数据段三个段。未初始化数据段、堆和栈不会占用目标文件的空间。

可执行程序是由各个目标文件经过连接而成。其主体部分依然是代码段、只读数据段和读写数据段，这三个段由各个目标文件（.o）经过"组合"而成。C 语言目标文件到可执行程序的连接如图 13-2 所示。

连接器将根据连接顺序将各个文件中的代码段取出，组成可执行程序的代码段，只读数据段和读写数据段也是如此。在连接过程中，如果出现符号重名、符号未定义等问题，将会产生连接错误。如果连接成功，将会生成一个统一的文件，这就是可执行程序。

由连接器生成的可执行程序包含了各个目标文件的各个段的内容，并且会附加可执行程序的头信息。在可执行程序中，各个目标文件的代码段、只读数据段、读写数据段经过了重新的排列组合。因此，在最终的可执行程序中，已经没有了各个目标文件的概念。

值得注意的是，在连接的过程中，连接器可以得到未初始化数据段的大小，它也是各个目标文件的各个未初始化数据段数据段之和，但是这个段是不会影响可 执行程序大小的。从 C 语言使用的角度，读写数据段和未初始化数据段都是可读写的。实质上，在目标文件（*.o）中未初始化数据段和读写数据段的区别也在于 此：读写数据段占用目标文件的容量，而未初始化数据段只是一个标识，不需要占用实际的空间。



（点击查看大图）图 13-2　C 语言目标文件到可执行程序的连接

例如，在某一个 C 语言的源程序文件中，具有以下的内容：

```
static char bss_data[2048];
static char rw_data[1024] = {""};
```

以上定义了两个静态数组，由于 bss_data 没有初始化，是一个未初始化数据段的数组，编译器只需要标识它的大小即可，而 rw_data 已经有了 一定的初始化数据（即使这个初始化数据没有实际的内容），它建立在已初始化数据段之上，编译器需要在读写数据段内为其开辟空间并赋初值。因此，在生成目标 文件的时候，由于 rw_data[1024]的存在，

目标文件的大小将增加1024字节，而bss_data [2048]虽然定义了2048字节的数组，目标文件的大小并不会因此而增加。

连接器在处理的过程中，会将各个目标文件读写数据段组合成可执行程序的读写数据段，类似rw_data等内容都会被组合，因此可执行程序中读写数据 段的大小会等于各个目标文件读写数据段之和。对于bss_data等未初始化数据段上的变量，连接器也将各个目标文件中的信息相加得到可执行程序的未初始 化段的大小，但是这个段同样不会占用可执行文件的空间。

在C语言中，读写数据段和未初始化数据段都包含了以下几种情况：整个程序的全局变量、单个文件内使用的全局变量（函数外部用static修饰的）、局部静态变量（函数内部用static修饰的）。对于这几种变量，连接器都会按照相同的方式进行组合。

知识点：在连接过程中，C语言的各个目标文件的代码段、只读数据段和读写数据段将组合成可执行程序的这三个段，未初始化数据段只有在运行时才会产生。

13.3.2 各个目标文件的关系

程序中通常会有大量的函数调用，这些被调用的函数只要有声明（而不需要定义实现），编译器就可以成功处理。在生成可执行文件的过程中，连接器将各个 可执行程序的代码段组合到一起，而有函数调用的地方还需要找到真正的函数定义才可以完成连接。因此，函数的定义和调用者可以在一个代码段内，也可以在不同 的代码段内。连接器会根据需要根据实际的情况修改编译器生成的机器代码，完成正确的跳转。

函数跳转的连接过程如图13-3所示。

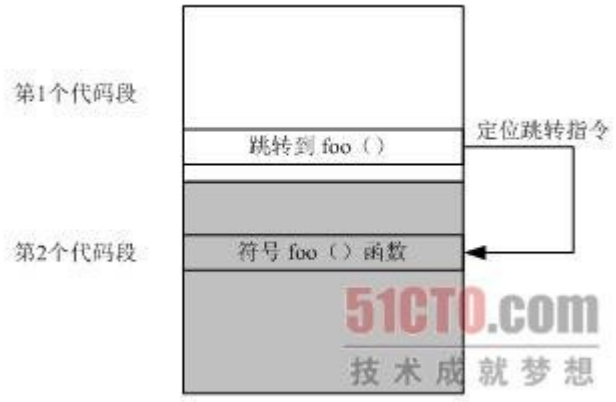
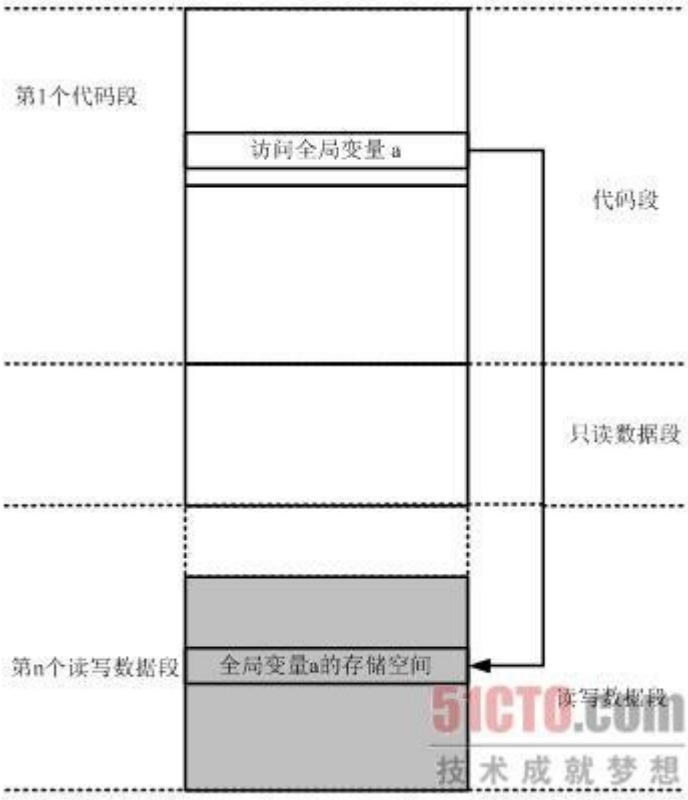


图13-3 函数跳转的连接

与函数跳转类似的是全局变量的访问，在C语言编译的过程中，程序可以访问用extern声明的外部全局变量，在连接的过程中，连接器需要找到实际变量在数据段中的位置，完成正确的变量访问。

程序中全局变量的连接如图13-4所示。



（点击查看大图）图13-4　全局变量的连接

对 于可执行文件的生成，其主要的工作是组合各个目标文件中的三个段。还将包含一些其他的过程。首先，所有的可执行程序都需要指定一个入口，在C语言中入口即 main函数，在一个C语言的应用程序各个源文件中，只能包含一个 main函数。其次，不同系统所使用的可执行程序可能包含不同的头信息，头信息是在主要 段之外附加的信息，可以供操作系统加载可执行程序的时候使用。

知识点：在连接过程之前，各个源文件生成目标文件相互没有关系。在连接之后，各目标文件函数和变量可以相互调用和访问，从而被联系在一起。

13.2　C语言程序的段

13.2.1　段的分类

根据C语言的特点，每一个源程序生成的目标代码将包含源程序所需要表达的所有信息和功能。目标代码中各段生成情况如下：

1．代码段（Code）

代码段由程序中的各个函数产生，函数的每一个语句将最终经过编译和汇编生成二进制机器代码（具体生成哪种体系结构的机器代码由编译器决定）。

顺序代码

基本数学运算（+，-），逻辑运算（&&，‖），位运算（&，|，^）等都属于顺序代码。

选择代码

if，if…else 语句等将由编译器生成选择代码。

循环代码

while()，do…while()语句等将由编译器生成循环代码。

对于一些较为复杂的数学运算如除法（\），取余（%）等，虽然它们是 C 语言的基本运算，但在各种编译系统中的处理方式却不一定相同。根据编译器和体 系结构的特点，对它们的处理方式有可能与加减等运算相同，即直接生成处理器的机器代码，也有可能转换成一个库函数的调用。例如，在没有除法指令的体系结构 中，编译器在编译 a/b 这类除法运算的时候，由于处理器没有与其对应的指令，因此会使用调用库函数来模拟除法运算。浮点数的处理与之类似：对于支持浮点运 算的体系结构，将直接生成浮点代码；对于不支持浮点数的处理器，编译器将会把每一个浮点运算用库函数调用的方式模拟。

2．只读数据段（RO Data）

只读数据段由程序中所使用的数据产生，该部分数据的特点是在运行中不需要改变，因此编译器会将该数据放入只读的部分中。C 语言的一些语法将生成只读数据段。

只读全局量

例如：定义全局变量 const char a[100]={"ABCDEFG"}将生成大小为 100 个字节的只读数据区，并使用字串"ABCDEFG"初始化。如果定义为 const char a[]={"ABCDEFG"}，没有指定大小，将根据"ABCDEFG"字串的长度，生成 8 个字节的只读数据段。

只读局部量

例如：在函数内部定义的变量 const char b[100] ={"9876543210"};其初始化的过程和全局量一样。

程序中使用的常量

例如：在程序中使用 printf("information \n")，其中包含了字串常量，编译器会自动把常量"information \n"放入只读数据区。

在 const char a[100]={"ABCDEFG"}中，定义了 100 个字节的数据区，但是只初始化了前面的 8 个字节（7 个字符和表示结束的'\0'）。在这种用法中， 实际后面的字节没有初始化，但是在程序中也不能写，实际上没有任何用处。因此，在只读数据段中，一般都需要做完全的初始化。

3．读写数据段（RW Data）

读写数据段表示了在目标文件中一部分可以读也可以写的数据区，在某些场合它们又被称为已初始化数据段。这部分数据段和代码段，与只读数据段一样都属于程序中的静态区域，但是具有可写的特点。

已初始化全局静态变量

例如：在函数外部，定义全局的变量 char a[100]={"ABCDEFG"}

已初始化局部静态变量

例如：在函数中定义 static char b[100] ={"9876543210"}。函数中由 static 定义并且已经初始化的数据和数组将被编译为读写数据段。

读写数据区的特点是必须在程序中经过初始化，如果只有定义，没有初始值，则不会生成读写数据区，而会定位为未初始化数据区（BSS）。如果全局变量 （函数外部定义的变量）加入 static 修饰符，写为类似 static char a[100]的形式，这表示只能在文件内使用，而不能被其他文件使用。

4．未初始化数据段（BSS）

未初始化数据段常被称之为 BSS（英文 Block Start by Symbol 的缩写）。与读写数据段类似，它也属于静态数据区，但是该段中的数据没有经过初始化。因此它只会在目标文件中被标识，而不会真正称为目标文件 中的一个段，该段将会在运行时产生。未初始化数据段只有在运行的初始化阶段才会产生，因此它的大小不会影响目标文件的大小。

在 C 语言的程序中，对变量的使用还有以下几点需注意：

1．在函数体中定义的变量通常是在栈上，不需要在程序中进行管理，由编译器处理。

2．用 malloc，calloc，realloc 等分配内存的函数所分配的内存空间在堆上，程序必须保证在使用后用 free 释放，否则会发生内存泄漏。

3．所有函数体外定义的是全局变量，加了 static 修饰符后的变量不管在函数内部或者外部都存放在全局区（静态区）。

4．使用 const 定义的变量将放于程序的只读数据区。

在 C 语言中，可以定义 static 变量：在函数体内定义的 static 变量只能在该函数体内有效；在所有函数体外定义的 static 变量，也只能在 该文件中有效，不能在其他的源文件中使用；对于没有使用 static 修饰的全局变量，可以在其他的源文件中使用。这些区别是编译的概念，即如果不按要求使 用变量，编译器会报错。使用 static 和没有使用 static 修饰的全局变量最终都将放置在程序的全局区（静态区）。

3.2.2 程序中段的使用

本小节使用简单的例子，说明 C 语言中变量和段的对应关系。C 语言程序中的全局区（静态区），实际对应着下述几个段：

只读数据段：RO Data

读写数据段：RW Data

未初始化数据段：BSS Data

一般来说，直接定义的全局变量在未初始化数据区，如果该变量有初始化则是在已初始化数据区（RW Data），加上 const 修饰符将放置在只读区域（RO Data）。

示例 1：

```
const char ro[]={"this is readonly data"}; /* 只读数据段 */
static char rw1[]={"this is global readwrite data"};
 /* 已初始化读写数据段 */
char bss_1[100];        /* 未初始化数据段 */
const char* ptrconst = "constant data"; /* "constant data"放在只读数据段 */
int main()
```

```
{
short b;        /* b 放置在栈上，占用 2 个字节 */
char a[100];     /* 需要在栈上开辟 100 个字节,a 的值是其首地址 */
char s[] = "abcde";    /* s 在栈上，占用 4 个字节 */
/* "abcde "本身放置在只读数据存储区，占 6 字节 */
char *p1;        /* p1 在栈上，占用 4 个字节   */
char *p2 = "123456";    /* "123456"放置在只读数据存储区，占 7 字节 */
  /* p2 在栈上，p2 指向的内容不能更改。 */
static char rw2[]={"this is local readwrite data"};
 /* 局部已初始化读写数据段 */
static char bss_2[100];  /* 局部未初始化数据段 */
static int c = 0;     /* 全局（静态）初始化区 */
p1= (char *)malloc(10*sizeof(char));
/* 分配的内存区域在堆区。 */
strcpy(p1, "xxxx");    /* "xxxx"放置在只读数据存储区，占 5 字节 */
free(p1);       /* 使用 free 释放 p1 所指向的内存 */
return 0;
}
```

示例 1 程序中描述了 C 语言源文件中语句如何转换成各个段。

只读数据段需要包括程序中定义的 const 型的数据（如：const char ro[]），还包括程序中需要使用的数据如"123456"。对于 const char ro[]和 const char* ptrconst 的定义，它们指向的内存都位于只读数据区，其指向的内容都不允许修改。区别在于前者不允许在程序中修改 ro 的值，后者允许在程序中修改 ptrconst 本身的值。对于后者，改写成以下的形式，将不允许在程序中修改 ptrconst 本身的值：

```
const char* const ptrconst = "constant data";
```

读 写数据段包含了已经初始化的全局变量 static char rw1[]以及局部静态变量 static char rw2[]。rw1 和 rw2 的差别只是在于编译时，是在函数内部使用的还是可以在整个文件中使用。对于前者，static 修饰在于控制程序的其他文件是否 可以访问 rw1 变量，如果有 static 修饰，将不能在其他的 C 语言原文件中使用 rw1，这种影响针对编译-连接的特性，但无论有无 static，变量 rw1 都将被放置在读写数据段。对于后者 rw2，它是局部的静态变量，放置在读写数据区;如果不使用 static 修饰，其意义将完全改变，它将会是开辟在 栈空间局部变量，而不是静态变量。在这里，rw1 和 rw2 后面的方括号（[]）内没有数值，表示静态区的大小由后面字符串的长度决定，如果将它们的定义改 写为以下形式：

```
static char rw1[100]={"that is global readwrite data"};
static char rw2[200]={"that is local readwrite data"};
```

则读写数据区域的大小直接由后面的数值（100 或者 200）决定，超出字符串长度的空间的内容为 0。

未初始化数据段，示例 1 中的 bss_1[100]和 bss_2[200]在程序中代表未初始化的数据段。其区别在于前者是全局的变量，在所有文件中 都可以使用；后者是局部的变量，只在函数

内部使用。未初始化数据段不设置后面的初始化数值，因此必须使用数值指定区域的大小，编译器将根据大小设置 BBS 中需要增加的长度。

栈空间包括函数中内部使用的变量如 short b 和 char a[100]，以及 char *p1 中 p1 这个变量的值。

变量 p1 指向的内存建立在堆空间上，栈空间只能在程序内部使用，但是堆空间（例如 p1 指向的内存）可以作为返回值传递给其他函数处理。示例 1 中各段的使用如表 13-1 所示。

表 13-1　示例 1 中各段的使用

段	使用情况	列　　表
只读数据（ROdata）	程序中不需要更改的全局变量	const char ro[]={"it is readonly data"}; char s[] = "abcde";中的"abcde"，占用 6 字节 char *p2 = "123456";中的"123456"，占用 7 字节 strcpy(p1, "xxxx");中的"xxxx"，占用 5 字节
读写数据（RWdata）	程序中需要更改，但是具有初始化值的全局变量	static char rw1[]={"that is global readwrite data"}; static char rw2[]={"that is local readwrite data"};
未初始化数据（BSS）	没有初始化值的全局变量	char bss_1[100]; char bss_2[100];
堆（heap）	需要用户分配的空间	p1= (char *)malloc(10*sizeof(char)); p1 指向的内存空间
栈（stack）	函数中临时变量 函数入口和返回值	函数中的变量: short b; char a[100]; char *p1; 变量 p1 本身占用 4 字节

示例 2:

```
long fun(long c)
{
char a[100];                                    /* 栈上的 100 个字节 *
/
long b;                                           /* 栈上的 4 个字节
*/
/* ..... */
return b;
}
```

栈空间主要用于以下3数据的存储：

函数内部的动态变量

函数的参数

函数的返回值

栈空间主要的用处是供函数内部的动态变量使用，变量的空间在函数开始之前开辟，在函数退出后由编译器自动回收。

栈空间的另外一个作用是在程序进行函数调用的时候进行函数的参数传递以及保存函数返回值。

在示例2程序的调用过程中，需要使用栈保存临时变量。在调用之前，需要将变量c保存在堆栈上，占用4字节，在函数返回之前，需要将返回值b保存在栈上，也占用4个字节的空间。同时，为数组a开辟100字节的栈空间。

在函数的调用过程中，如果函数调用的层次比较多，所需要的栈空间也逐渐增大。对于参数的传递和返回值，如果使用较大的结构体，在使用的栈空间也会比较大。

13.3.3 连接错误示例（1）

连接过程中常见的错误是符号未找到（undefined reference）和符号重定义（redefinition）。由于在编译器在处理各个符号的时候，已经没有了各个C语言源文件的概念，只有目标文件。 因此对于这种错误，连接器在报错的时候，只会给出错误的符号的名称，而不会像编译器报错一样给出错误程序的行号。

符号未定义的错误经常发生在符号已经声明，但是并没有具体的定义的情况下。在C语言中，符号可以是一个函数，也可以是一个全局变量。在程序的编译过程中，只要符号被声明，编译就可以通过，但是在连接的过程中符号必须具有具体的实现才可以成功连接。

例如：某一个源程序的文件的某一个地方调用了一个函数，如果这个函数具有声明，这时编译就可以通过。在连接的过程中，连接器将在各个代码段中寻找函 数，如果函数没有在程序的任何一个位置中定义，那么就不会有函数符号，这时连接器将发生符号未定义的连接错误。请阅读如下程序：

```
extern void function(void);
int main(void)
{
/* …… */
function ();
/* …… */
return 0;
}
```

在以上的例子中函数function可以和其调用者main在同一个源文件中，也可以在其他的源文件被调用中，但是它必须被定义。

符号重定义错误与符号未定义错误类似，如果连接器在连接的时候，发现一个符号在不同的地方有多于一个定义，这时就会产生符号重定义错误。对于同一个符号，如果在多个

源文件中出现多次，将会产生符号重定义错误。

知识点：在连接过程中，符号未定义和符号重定义是两种最基本的错误。

下面以一个包含 3 个文件的程序为例，说明在连接过程中的错误。这个程序的三个文件为 hello.h、hello.c 和 main.c。

main.c 文件如下所示：

```
#include "hello.h"

int main(void)
{
hello();
string[0] = 'a';
return 0;
}
```

hello.h 文件如下所示：

```
#ifndef HELLO_H
#define HELLO_H

void hello(void);
extern char string [];
#endif
```

hello.c 文件如下所示：

```
#include <stdio.h>
#include "hello.h"

char string [1024] = "";
void hello(void)
{
printf("=== hello ===\n");
}
```

以上是一个可以正常运行的程序，编译器将对 main.c 和 hello.c 两个源文件分别进行编译。在 main 函数中，调用了函数 hello，并访问 了数据 string，由于包含了 hello.h 头文件，hello 和 string 具有声明，因此 main.c 可以被成功编译。hello.c 中定义了一个 读写数据段上的变量 string 和函数 hello。在连接的过程中，目标文件 main.o 对 string 和 hello 符号进行访问，hello.o 提供了 这两个符号，因此可以连接成功。

1．由于无数据定义导致符号未定义错误

将 hello.c 中对 data 的定义去掉，这时编译器依然可以成功编译 main.c 和 hello.c。但是，由于 string[0] ='a'编译后产生的代码将对 string 访问，在连接的过程中找不到 string 这个数据，因此会产生符号未定义连接错误（找不到数据）。

2．由于无函数实现导致符号未定义错误

将 hello.c 中对 hello 函数的定义去掉，这时编译器还是可以成功编译 main.c，而且有这个符号，因此也会产生目标文件 hello.o。但是，在连接器处理函数调用的时候，需要跳转到 hello 符号，由于实际上并没生符号而报告未定义连接错误（找不到函数）。

知识点：在程序中使用只有声明而未定义的函数或数据，可以成功编译，但是连接时将发生符号未定义错误。

3．由于数据仅能在文件内部使用，导致符号未定义错误

将 hello.c 的数组 string []更改为静态变量：

```
static char string [1024] = "";
```

此时编译依然是可以通过的，这时候由于 string 已经是一个静态数据，因此它不会出现在 hello.c 的目标文件的符号中。也就是说，增加 static 修饰后，目标文件和以前将略有不同。

连接器在处理的时候，虽然有 string 这个数据，但是它只有数据区而没有符号，因此依然会出现未定义符号错误。

4．函数具有声明可编译通过，但有连接错误

取消 hello.c 中对函数的声明，在 main.c 增加对该函数的声明：

```
void hello(void);

在 hello.c 中，将 hello 函数的定义改为静态的：

static void hello(void)
{
printf("=== hello ===\n");
}
```

这时，编译还是能通过，由于 hello 成为静态函数，只能在文件内部使用，不会产生外部的符号。因此，虽然在 main 中对该函数进行了声明，连接也会产生未定义符号错误。

函数在各个源文件中的声明不会对连接产生影响，因此，在本例中，只要 hello 函数不是静态的，连接就可以通过。

知识点：使用 static 的函数和变量不能给其他文件调用，否则会发生连接的符号未定义错误。

5．定义同名变量导致符号重定义错误

在 hello.h 文件中，取消对 string 的外部声明。在 main 函数中增加一个定义在读写数据段的字符数组 string[]，

```
char string[1024] = "main string";
```

编译通过后，在连接时将会产生符号重定义的连接错误。main.c 和 hello.c 中各自有一个

读写数据段的字符数组 string[]，虽然它们看似没有直接的关系，但连接器无法处理这种情况，依然会产生连接错误。

在这种情况下，即使没有对 string 的引用（即没有 string[0] = 'a'），连接器依然无法处理两个同名的读写数据段全局变量，这时还是会报告符号重定义的连接错误。

13.3.3 连接错误示例（2）

6．实现同名函数导致符号重定义错误

在 main.c 文件中，增加对 hello 函数的定义。

```
void hello(void)
{
printf("+++ main hello +++\n");
}
```

编译通过后，连接时会产生符号重定义的错误。实际上，由于在 hello.c 和 main.c 的目标文件的代码段中分别定义 hello 函数。连接器认为 出现重定义。与上例的程序类似，即使没有对函数 hello 的调用，编译器也不允许在代码段中出现 2 个 hello 函数的符号，所以还是会产生符号重定义连接 错误。

知识点：在多个文件中定义全局的同名函数和变量，连接时将发生符号重定义错误。

7．静态函数与其他文件中的函数重名，可以正常使用

将 main.c 文件更改成如下：

```
#include "hello.h"

int main(void)
{
hello();
string[0] = 'a';
return 0;
}

static void hello(void)                          /* 静态的函数，内部使用
*/
{
printf("+++ main hello +++\n");
}
```

程序主要的变化是增加了静态的 hello 函数，在这种情况下，是可以成功地进行编译连接和运行的，运行结果如下所示：

```
+++ main hello +++
```

从运行结果中可以看到，main 中调用的 hello 函数是 main.c 文件中定义的 static 的 hello 函数。

值得注意的是，在这种情况下，编译器在进行编译的时候， main 函数写在静态的函数 hello 前面，因此可以通过编译。这是由于 main.c 文件中 包含了 hello.h 文件，其中具有对 hello()函数的声明。但是，当编译器编译到 main.c 之中的 hello 函数的时候，由于 static 头文件 中声明函数原型不同，可能出现一个编译报警（warning）。

8．静态变量与其他文件中的变量重名，可以正常使用

在 main.c 中，增加静态（static）的读写数据段的字符数组 string[]的定义。

```
char string[1024] = "main string";
```

在这种情况下，编译连接可以成功。当连接工作完成后，可执行程序的读写数据段将出现两个 string[1024]数组，均占用的空间。一个是 hello.c 中定义的全局的数组，一个是 main.c 定义的文件内部使用的数组。在数据访问的过程中，语句 string[0] = 'a'访问的将是 main.c 中定义的数组 string[]。

知识点：如果全局变量和函数已定义，而在某个文件中另外定义静态的同名变量和函数，可以在文件内部使用同名的静态变量和函数。在使用的过程中，将优先使用文件内的变量和函数。

9．在头文件中定义已经初始化数据，可能产生问题

在程序中，将 string[]的定义放入 hello.h 的头文件中：

```
#ifndef HELLO_H
#define HELLO_H

void hello(void);
char string[1024] = "";
#endif
```

同时，取消在 hello.c 中对 string[]数组的定义。此时，由于 hello.c 和 main.c 同时包含了 hello.h 头文件，因此 string 在内存中有两份，连接的时候将产生符号重定义错误。

如果将头文件中 string 的定义改为静态的，这时不会产生连接错误，但是会在 hello.c 和 main.c 的目标文件中各产生一个 string[1024]。最终可执行程序的读写数据段中也会有两个 string。

如果在头文件中使用如下方式定义：

```
const char string_const[1024] = {"constant data"};
```

由于具有 const 属性，string_const 是一个在只读数据段上的常量。这样有多个文件包含该头文件的时候，在连接过程中也会出现符号重定义的错误。连接器在这个问题上，对读写数据区和只读数据区的处理方式是类似的。

知识点：具有初始值的变量将被连接到读写数据区。在头文件中不应该定义有初始值的全局变量。同样，也不应该定义只读数据段的常量。否则，在头文件被多个文件包含的时候，将发生连接错误。

10．在头文件中定义未初始化数据段，可以正常使用

在程序中，hello.h 的头文件中定义 string[]，但是没有初值：

```
#ifndef HELLO_H
#define HELLO_H

void hello(void);
char string[1024];
#endif
```

同时，取消在 hello.c 中对 string[]数组的定义。在这种情况下，编译连接都是可以通过的，程序也可以正常运行。

知识点：无初始化的变量将被连接到未初始化数据段，在头文件中可以定义。当头文件被多个文件包含时，该未初始化段在运行时也将只有一份。

事实上，由于没有初值，string[]将不再是读写数据段上的变量，而是未初始化数据段上的变量。未初始化段上的变量并不会占用目标文件或者可执 行文件中的空间，它们只是一些标识。由于不需要分配空间，编译器允许这种做法。未初始化数据段的变量在运行的时候才会产生，而且只会有一个。

同理，可以将 string 修改为 static 的未初始化变量：

```
#ifndef HELLO_H
#define HELLO_H

void hello(void);
static char string[1024];
#endif
```

在这种情况下，在编译的时候将会在两个目标文件中各自记录一个未初始化数据段，在运行时程序将在内存上开辟两个独立的 1024 字节的数据区。

比较以上的两个示例（9 和 10），总结出以下的结论：

首先，不应该在头文件中使用全局的读写数据变量，这样当两个文件同时引用这个头文件的时候，将会产生符号重定义连接错误。

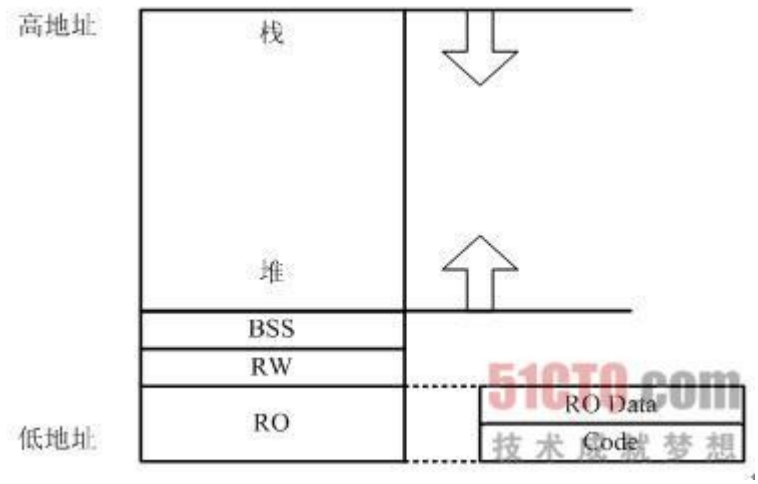
其次，在头文件中也不应该使用静态的变量，无论它有没有初值（即在读写数据段或者未初始化数据段），这样虽然不会引起连接错误，但是在各个源文件中各自产生变量，不但占用更多的空间，而且在逻辑上是不对的，也违背头文件的使用原则。

最后，在头文件中使用全局的没有初始化的变量是可以的，它在程序运行的过程中，在内存中只会有一份，可以被包含该头文件的程序访问。

从 C 语言程序设计的角度，不应该在头文件中定义变量或者函数。对于函数，在头文件中只是声明，需要在源文件中定义；对于变量，无论何种性质（只读数据段、可读写数据段、未初始化数据段），最好的方式是在 C 语言的源文件中定义，在头文件中使用 extern 声明。

13.4　C 语言程序的运行

在嵌入式系统中，程序最终是要放置在内存中运行的，程序的几个段，最终会转化为内存中的几个区域。C 语言可执行程序的内存布局如图 13-5 所示。



（图 13-5　C 语言可执行程序的内存布局

在内存中，从低地址到高地址，依次是只读段、读写段、未初始化数据段、堆段、栈段。

映像文件中将包含代码段（Code）、只读数据段（RO Data）以及读写数据段（RW Data），未初始化数据段（BSS）将在程序的初始化阶段中开辟，堆栈在程序运行时动态开辟。

只读区（RO）包括了代码和只读数据，在内存区域中，代码段（Code）和只读数据段（Ro Data）的存放形式上基本没有区别。

对于程序运行时的内存使用，堆和栈一般是相向扩展的。堆的分配由程序决定，栈由编译器管理。

在以上概念中，只是一种内存分布，并没有考虑实际系统的情况。在实际的系统中，程序有载入和运行两个概念。嵌入式系统由两种内存，一种是可以固化只 读的内存（如：ROM，Nor Flash），另一种是易失的可读写的内存（如：SRAM 和 SDRAM）。程序中的各个段也有需要固化和需要读写的。程序中的各段必须载入到内存的恰当位 置，程序才可以运行。C 语言各部分的需要固化和可写的情况如表 13-2 所示。

表 13-2　C 语言各部分的需要支持固化和可写的情况

段	需要固化	需要可写
代码（Code）	是	否
只读数据（RO data）	是	否
读写数据（RW data）	是	是

未初始化数据（BSS）	否	是
堆（heap）	否	是
栈（stack）	否	是

在嵌入式系统中，经过编译的C语言程序可以通过操作系统运行，也可以在没有操作系统的情况下运行。程序存放的位置和运行的位置通常是不一样的。

一般情况下，经过编译后的程序存储在Flash或者硬盘中，在运行时需要将程序加载到RAM中。嵌入式系统的Nor Flash和硬盘还有一定的差别，在硬盘的程序必须加载到RAM中才可以运行，但是在Nor Flash中的程序可以通过XIP（eXcutive In Place）的方式运行。

在嵌入式系统中，C语言程序的运行包括3种类型：第一种是调试阶段的程序运行，这个阶段程序存放的位置和运行的位置是相同的；第二种是程序直接在Flash中运行（XIP）；第三种是将Flash或者硬盘中的程序完全加载到RAM中运行。

在C语言程序的运行中，存在着两个基本的内存空间，一个是程序的存储空间，另一个是程序的运行空间。程序的存储空间必须包括代码段、只读数据段和读写数据段，程序的加载区域必须包括读写数据段和未初始化数据段如表13-3所示。

表13-3　C语言各部分使用的存储空间

段	代码	只读数据	读写数据	未初始化数据
程序的存储空间（ROM）	需要			不需要
程序的加载空间（RAM）	不需要		需要	

由于程序放入系统后，必须包括所有需要的信息，代码表示要运行的机器代码，只读数据和读写数据包含程序中预先设置好的数据值，这些都是需要固化存储的，但是未初始化数据没有初值，因此只需要标示它的大小，而不需要存储区域。

在程序运行的初始化阶段，将进行加载动作，其中读写数据和未初始化数据都是要在程序中进行"写"操作，因此不可能放在只读的区域内，必须放入RAM中。当然，程序也可以将代码和只读数据放入RAM。

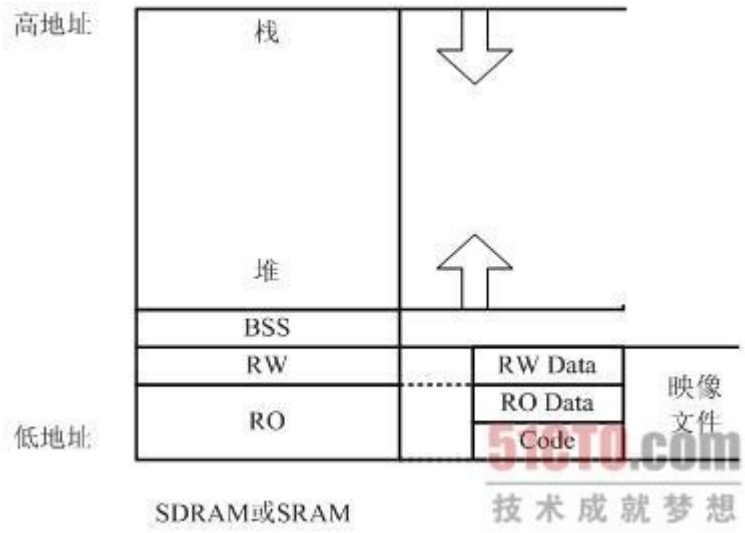
在程序运行后，堆和栈将在程序运行过程中动态地分配和释放。

13.4.1　RAM调试运行

本节介绍程序的一种特殊的运行方式，即在程序的调试阶段将主机的映像文件直接放置到目标系统的RAM中。在这种应用中，RAM既是程序的存储空间，也是程序的运行空间。

在嵌入式系统中，这是一种常用的调试方式，而不是通常的运行方式。在通常的运行方式下，程序运行的起始地址一般不可能是RAM。RAM在掉电之后内 容会丢失，因此系统上电的时候，RAM中一般不会有有效的程序。但是在程序的调试阶段，可以将程序直接载入RAM，然后在RAM的程序载入地址处运行程 序。

嵌入式系统 RAM 中的调试程序的内存布局如图 13-6 所示。



（点击查看大图）图 13-6　RAM 中的调试程序的内存布局

这是一种相对简单的形式，因为代码段的存储地址和运行地址是相同的，都是 RAM（SDRAM 或者 SRAM）中的地址。在这种情况下，程序没有运行初始化阶段加载的问题。

从主机向目标机载入程序的时候，程序映像文件中代码段（code 或 text）、只读数据段、读写数据段依次载入目标系统 RAM（SDRAM 或者 SRAM）的空间中。

程序载入到目标机之后，将从代码区的地址开始运行，在运行的初始化阶段，将开辟未初始化数据区，并将其初始化为 0，在运行时将动态开辟堆区和栈区。

在没有操作系统的情况下，开辟内存的工作都是由编译器生成的代码完成的，实现的原理是在映像文件中加入这些代码。主要工作包括：在程序运行时根据实 际大小开辟未初始化的数据段；初始化栈区的指针，这个指针和物理内存的实际大小有关；在调用相关函数（malloc、free）时使用堆区，这些函数一般 由调用库函数实现。表 13-4 列出了 C 语言程序在 RAM 中的调试过程。

表 13-4　C 语言程序在 RAM 中的调试过程

阶段	涉及的部分	主要工作
程序的映像	代码段（Code） 只读数据段（RO Data） 读写数据段（RW Data）	将程序放置在 RAM 中
初始化阶段	未初始化数据段（BSS）	开辟 BSS 段并且清零
运行阶段	代码段（Code） 只读数据段（RO Data）	运行 RAM 代码段中的程序 ，动态地在 RAM 中开辟

	读写数据段（RW Data） 未初始化数据段（BSS） 堆（heap） 栈（stack）	堆和栈

知识点：程序直接载入 RAM 运行时，程序的加载位置和运行位置是一致的，因此不存在段复制的问题，需要在初始化阶段开辟未初始化区域，在运行时使用堆栈。

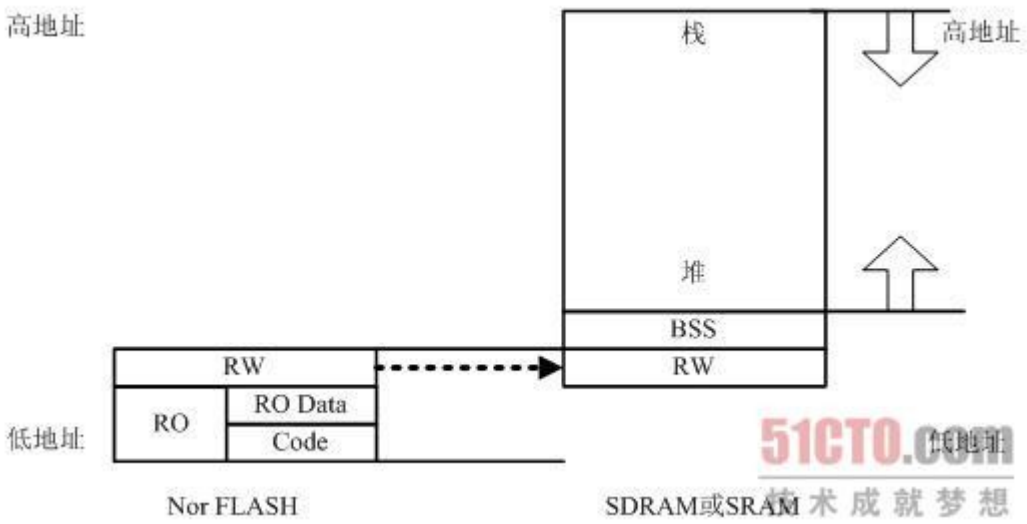
13.4.2 固化程序的 XIP 运行

固化应用是一种嵌入式系统常用的运行方式，其前提是目标代码位于目标系统 ROM（Flash）中。ROM 中的区域包括映像文件的代码段（code 或 text）、只读数据段（RO Data）、读写数据段（RW Data）。

以 XIP（在位置执行）方式运行程序时内存布局如图 13-7 所示。

代码的运行也是在 ROM（Flash）中，因此，在编译过程中代码的存储地址和运行地址是相同的，由于上电时需要启动，因此该代码的位置一般是（0x0）。

在这种应用中，一件重要的事情就是将已初始化读写段的数据从 Flash 中复制到 SDRAM 中，由于已初始化读写段既需要固化，也需要在运行时修改，因此这一步是必须有的，在程序的初始化阶段需要完成这一步。



（点击查看大图）图 13-7 XIP 运行程序时的内存布局

一般来说，在编译过程中需要定义读写段和未初始化段的地址。在程序中可获取这些地址，然后就可以在程序的中加入复制的代码，实现读写段的转移。表 13-5 列出了 C 语言程序的 XIP 运行过程。

表 13-5 C 语言程序的 XIP 运行过程

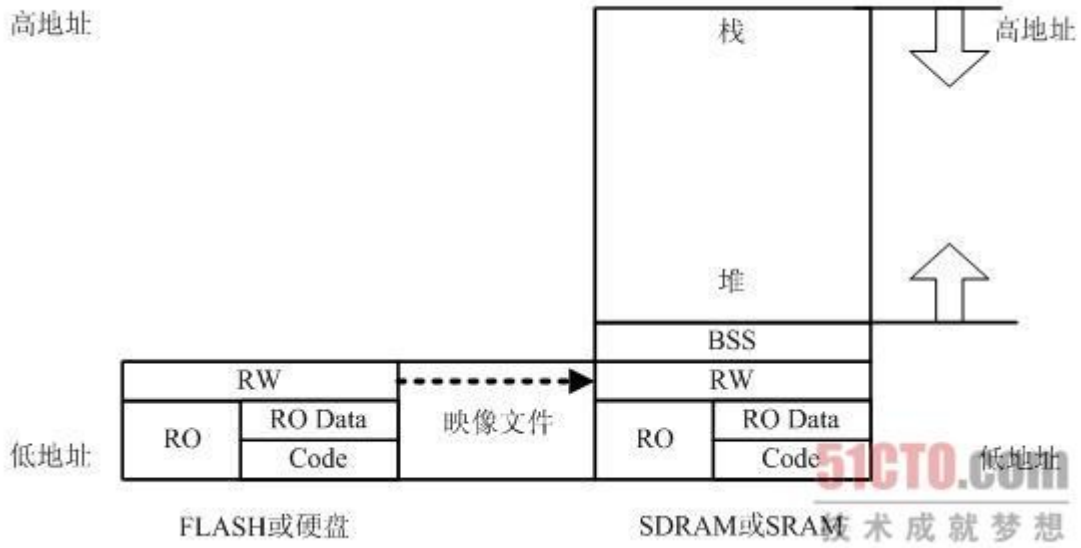
阶段	涉及的部分	主要工作

程序的映像	代码段（Code） 只读数据段（RO Data） 读写数据段（RW Data）	程序放置在 Flash 中
初始化阶段	读写数据段（RW Data） 未初始化数据段（BSS）	复制读写数据段到 RAM 中 开辟未初始化段并且清零
运行阶段	代码段（Code） 只读数据段（RO Data） 读写数据段（RW Data） 未初始化数据段（BSS） 堆（heap） 栈（stack）	运行 Flash 代码段中的程序， 动态地在 RAM 中开辟堆和栈

知识点：程序在 ROM 或者 Flash 中以 XIP 形式运行的时候，不需要复制代码段和只读数据段，但是需要在 RAM 中复制读写数据段，并另辟未初始化数据段。

13.4.3 固化程序的加载运行

在某些时候，在存放程序的位置是不能运行程序的，例如程序存储在不能以 XIP 方式运行的 Nand-Flash 或者硬盘中，在这种情况下，必须将程序完全加载到 RAM 中才可以运行固化程序加载运行的内存布局如图 13-8 所示：



（点击查看大图）图 13-8 固化程序加载运行的内存布局

依照这种方式运行程序，需要将 Flash 中所有的内容全部复制到 SDRAM 或者 SRAM 中。在

一般情况下，SDRAM 或者 SRAM 的速度要快于 Flash。这样做的另外一个好处是可以加快程序的运行速度。也就是说，即使 Flash 可以运行程序，将程序加载到 RAM 中运行也还有一定的优势。

这样做也产生了另外一个问题：代码段的载入地址和运行地址是不相同的，载入地址是在 ROM（Flash）中，但是运行的地址是在 RAM（SDRAM 或者 SRAM）中。对于这个问题，不同的系统在加载程序的时候有不同的解决方式。

C 语言固化程序的加载运行过程如表 13-6 所示。

表 13-6　C 语言固化程序的加载运行时各段的情况

阶段	涉及的部分	主要工作
代码的映像	代码段（Code） 只读数据段（RO Data） 读写数据段（RW Data）	将程序放置在 Flash 中
初始化阶段	代码段（Code） 只读数据段（RO Data） 读写数据段（RW Data） 未初始化数据段（BSS）	加载代码段和只读数据段到 RAM 中 复制读写数据段到 RAM 中 开辟未初始化段并且清零
运行阶段	代码段（Code） 只读数据段（RO Data） 读写数据段（RW Data） 未初始化数据段（BSS） 堆（heap） 栈（stack）	运行 RAM 代码段中的程序， 动态地在 RAM 中开辟堆和栈

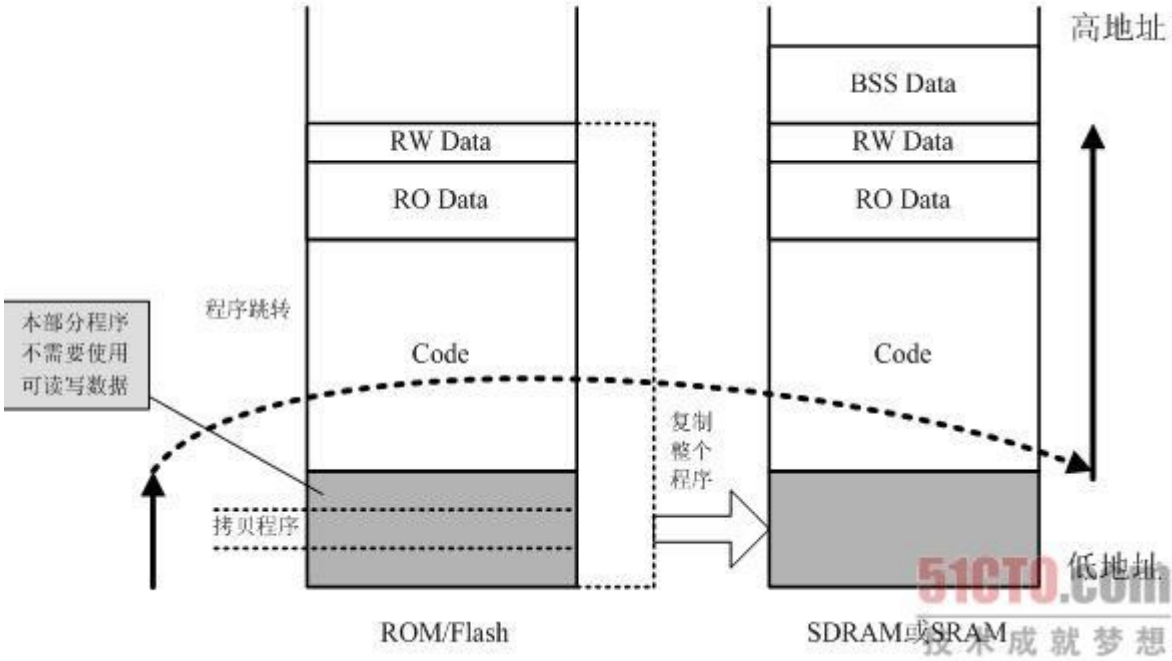
知识点：固化程序在加载运行时，需要复制代码段、只读数据段和读写数据段到 RAM 中，并另辟未初始化数据段，然后在 RAM 中运行程序（执行代码段）。

以这种加载方式的运行程序，另外一个重要的问题是：如何把代码移到 RAM 中。在有操作系统的情况下，代码的复制工作是由操作系统完成的，在没有操作 系统的情况下，处理方式相对复杂，程序需要自我复制。显然，这种方式实现的前提是代码最初放置在可以以 XIP 方式执行的内存中。

程序本身复制的过程也是需要通过程序代码完成的，这时需要程序中的代码根据将包含自己的程序从 ROM 或者 Flash 中复制到 RAM 中。这是一个比较 复杂的过程，程序的最前

面部分是具有复制功能的代码。系统上电后，从 ROM 或者 Flash 起始地址运行，具有复制功能的代码将全部代码段和其他需要复制的 部分复制到 RAM 中，然后跳转到 RAM 中重新运行程序。

固化程序加载复制和跳转过程如图 13-9 所示。



（点击查看大图）图 13-9　固化程序加载复制和跳转过程

在代码的前面一小部分是初始化的内容，这部分内容中有一部分是复制程序，这段复制程序将代码段复制至 RAM 中，当这段初始化程序运行完成后，将跳转到 RAM 中的某个地址运行。

13.4.4　C 语言程序的运行总结

在上面几节，主要介绍了 C 语言运行时内存的使用情况。其关注点是程序中主要的段，事实上，程序可能不仅包括了上述主要段，还可能包括一些头信息。程 序实际的运行也分为在操作系统下运行和直接运行等情况。在具有操作系统的情况下，程序由操作系统加载运行，加载的时候可执行程序可以是一个文件，这个文件 将包含程序的主要段以及头信息。

对于 Linux 操作系统，目标程序是可执行的 ELF（Executable and linking Format）格式；对于 uCLinux 操作系统，目标程序是 Flat 格式；对于需要在系统直接运行的程序，目标程序应该是纯粹的二进制代码，载入系统 后，直接转到代码区地址执行。

事实上，无论运行环境如何，C 语言程序在运行时所进行的动作都是类似的。程序在准备开始运行的时候，以下几个条件都是必不可少的：

1．代码段必须位于可运行的存储区。

2．读写数据段必须在可以读写的内存中，而且必须经过初始化。

3．未初始化数据段必须在可以读写的内存中开辟，并被清空。

对于第1点，代码段如果位于可以运行的存储区域中（例如Nor Flash或者RAM），它就不需要加载，可以直接运行；如果代码段位于不能运行的存储区域中（例如：Nand Flash或者硬盘）中，它就必须被加载到RAM运行。

《嵌入式1inux上的c语言编程实践》（亚嵌教材）学习笔记---------by Andy
第九章内存的堆和栈

c使用的内存分为：静态区和动态区，静态区
静态区：只读数据区，初始化数据区，未初始化数据区
动态区：堆区，栈区

栈:
1.使用依赖硬件机制，有两种增长方向。有空栈和满栈。空栈：栈指针指向未使用的数据区。满栈：栈指针指向没使用的数据区。
2.由编译器管理。函数退出内容释放，栈上的内存不能被别的函数使用（不能返回站内变量的地址）。
3.使用线性存储方式
4.主要特性：FILO，先进后出
5.有一个指针，通过指针+偏移量来访问
6.

堆:
1.堆的增长方向和栈相反。（大体上如此）
2.堆由程序员管理，调用和释放由程序的库函数完成。每次分配返回一个指针，可多次分配，得到多个指针，可用每个指针来访问。
3.用链表实现
4.四个分配和释放的函数：malloc，free，calloc，realooc
5.可以再一个函数中开辟，另一个函数中释放。
堆内存管理容易出现的问题:
1. 内存泄露
2. 野指针的使用和释放
3. 非法释放指针。
正确的使用方法：在内存释放后，将内存指针置为NULL，在使用时判断指针释放为NULL
正确使用堆

```
/* how to use malloc correctly */

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main(void)
```

```
{
        char *pa;
        pa = (char *)malloc(sizeof(char)*20);
        if(NULL != pa)
        {
                strcpy(pa, "memory leak");
                printf("pa = %x \n", (unsigned int)pa);
                printf("pa = %s \n", pa);
        }
        free(pa);
        pa = NULL;
        if(NULL != pa)
        {
                printf("pa = %s \n", (unsigned int)pa);
        }
        return;
}
```

堆，栈的使用比较：

1.　　利用返回值传递信息

（1）返回值可以是任何内存的地址，但不能返回内部栈区的地址。

当希望返回栈上较多的内容时，不能用指针，可以用结构体（结构体在函数外定义），但开销比较大。

（2）返回结构体是，要从被掉函数的栈空间中将结构体复制的调用函数的栈空间上，push 和 pop 开销大。

2．利用参数传递信息

（1）参数是变量的情况，swap 函数，不能交换

（2）参数是指针的情况，swap 可以交换

（3）参数是结构体的情况和变量类似。开销大，可以定义结构体指针，来传递参数。

（4）参数是数组的情况，当做指针来处理。实际入栈的是数组地址的指针。

第十章 函数指针的使用

　　函数指针是指向函数代码地址的指针。C 语言中，函数名表示函数代码在内存中的地址。

例如：函数 foo()，&foo 和 foo 的含义相同。和数组名类似。

1.函数指针的声明：int (*pf)()　因为（）的结合优先级高于*，所以用括号，

否则为*g(), 意为返回值为指针的函数。
简单函数指针的定义:
Void (*pf) (void);
然后复制: pf=foo;
2. 函数指针的类型转换
类型转换只需要将声明中的变量名和声明末尾的分号去掉, 然后用括号括起来即可。
如 int (*fp) (); 表示 fp 是个指向返回值为整形的函数的指针。因此
(int (*) ()) 表示一个 "指向返回值为整形的的函数的指针" 的类型转换符。
注: 函数的声明和调用不同。
第十四章 嵌入式 c 语言常用语法
1.使用指针操作内存
向地址 0x0040 处写一个字节的数据 0xf0

```
Unsigned char *p = (unsigned char *)0x0040;
*p = 0x40;
```

或:

```
*(unsigned char *)0x40;                    //注意 unsigned 的使用, 思考读? 16 位的读? 32 位的读?
```

3. 结构体成员的对齐问题

```
typedef struct _s1
{
    char m1;
    int          m2;
    char m3;
    shaort     m4;
}S1;
```

sizeof(S1)不等于 8 个字节, 等于 12 个字节, 故在定义结构体时, 注意结构体成员的排列顺序, 使结构体最小。
4. 变量的初始化技巧

```
//数组的初始化
char a[10] = "abcde"
//编译器做了很多事情才完成, 先在栈上开辟 10 个字节空间, 然后从内存中复制。
static const rodata[6] = "abcde";
char b[10];
strcpy(b,rodata);
//使用函数完成赋值, 在函数运行时完成
//两者依赖的库不通, 效率和规模类似。
```

结构体的初始化
使用每个赋值的方式比使用列表赋值的方式开销小。

```
struct S
{
```

```
        int a;
        int b;
        int c;
}s1;
//列表方式，开销大
fun()
{
        s1 = {1,2,3};
}
//成员分别赋值开销小
fun()
{
        s1.a=1;
        s1.b=2;
        s1.c=3;
}
```