高性能科学计算教程

The Art of HPC, volume 4

Victor Eijkhout

2022，最后格式化于 2024 年 3 月 28 日
Book and slides download: https://tinyurl.com/vle394course
Public repository: https://bitbucket.org/VictorEijkhout/scientific-computing-public

前言

高性能科学计算领域除了需要广泛的科学知识和 “ 计算民间传说 ” 外，还需要许多实用技能。可以称之为科学计算工艺中的 “ 木工 ” 方面。

作为《 Introduction to High Performance Scientific Computing 》一书的配套资料，该书涵盖了背景知识，这里提供了一套关于那些对成为成功的高性能实践者至关重要的实用技能的教程。

这些教程应在电脑前完成。鉴于科学计算的实践，它们明显带有 Unix 偏向。

Public draft 本书开放征求意见。有什么内容缺失、不完整或不清楚？材料的呈现顺序是否错误？请将您的任何意见通过邮件告知我。

您可能在多个地方找到了这本书；权威的下载地址是 https://theartofhpc.com/ 该页面还链接到 lulu.com ，您可以在那里获得一本精美的印刷版。

Victor Eijkhout

eijkhout@tacc.utexas.edu 研究科学家 德州高级计算中心 德克萨斯大学奥斯汀分校

目录

				练习	
课程	主题	Book	幻灯片	课堂内	作业
1	Unix	1	unix		1.42
2	Git	5			
3	Programming	2	programming	2.3	2.4
4	库	2	programming		
5	调试	11			root code
6	LATEX	15			15.13
7	Make	3			3.1, 3.2

表 1：HPC 课程木工部分的时间表。

成为高性能科学计算的有效实践者的重要部分可以称为 “ HPC 木工 ”：一些本质上不是科学的技能，但对于完成你的工作仍然是必不可少的。

绝大多数科学编程是在 Unix 平台上完成的，因此我们从第 1 章的 Unix 教程开始，接着在第 2 章中解释了编译器、链接器等如何处理你的代码。

接下来你将了解一些能够提高你生产力和效率的工具：

- Make 工具用于管理项目的构建；第 3 章。
- 源代码管理系统以一种可以撤销更改或维护多个版本的方式存储你的代码；在第 5 章你将看到 *subversion* 软件。
- 一旦你的程序开始产生结果，存储和交换科学数据就变得非常重要；在第 7 章你将学习 *HDF5* 的使用。
- 程序数据的可视化输出很重要，但这是一个过于广泛的话题，无法在这里详细讨论；第 9 章教你 *gnuplot* 包的基础知识，适合简单的数据绘图。

我们还考虑程序开发本身的活动：第 10 章讨论如何编写代码以防止错误，第 11 章教你使用 *gdb* 调试代码。第 13 章包含一些关于如何编写使用多种编程语言的程序的信息。

最后，第 15 章将教你关于 LATEX 文档系统的知识，这样你就可以用排版精美的文章来报告你的工作。

许多教程都非常实用。请在计算机前边做边学。 r!

Table 1 给出了课程中 carpentry 部分的建议课程大纲。Wilson 的文章 [24] 是关于这种 ‘HPC carpentry’ 背后思考的好读物。

第 1 章

Unix 简介

Unix 是一个操作系统 *(OS)*，即用户或用户程序与硬件之间的软件层。它负责文件和屏幕输出，并确保多个进程可以在一个系统上并存。然而，它对用户来说并不直接可见。

本教程的大部分内容适用于任何类 Unix 平台，但 Unix 并非只有一种：

- 传统上，Unix 有几个主要版本：ATT 或 *System V*，以及 *BSD*。Apple 有接近 BSD 的 Darwin；IBM 和 HP 有自己的 Unix 版本，Linux 则是另一种变体。如今，许多 Unix 版本遵循 *POSIX* 标准。这些差异存在于底层，如果你正在学习本教程，可能很长一段时间内都不会注意到它们。
  - 在 Linux 中，有各种 *Linux* 发行版，如 *Red Hat* 或 *Ubuntu*。它们主要在系统文件的组织上有所不同，你大概也不必太担心这些差异。
  - 命令 shell 的问题将在下面讨论。这实际上构成了不同运行 “Unix” 的计算机之间最明显的区别。

1.1 Shells

大多数时候你使用 Unix 时，都是在输入由一个称为 *shell* 的解释器执行的命令。shell 负责实际的操作系统调用。可用的 Unix shell 有几种选择

- 本教程的大部分内容集中在 `sh` 或 `bash` shell 上。
  - 出于多种原因（例如参见第 3.5 节），`bash` 类 shell 比 `csh` 或 `tcsh` shell 更受推荐。本教程不会涵盖后者。
  - 最新版本的 *AppleMac OS* 默认使用 `zsh`。虽然该 shell 与 `bash` 有许多共同点，我们将明确指出它们的差异。

1.2 Files and such

Purpose. 在本节中，您将了解 Unix 文件系统，它由存储目录组成，这些目录存储文件。您将学习可执行文件以及用于显示数据文件的命令。

1.2.1 Looking atfiles

Purpose. 在本节中，您将学习用于显示文件内容的命令。

本节学习的命令

`ls`	列出文件或目录
`touch`	创建新文件 / 空文件或更新现有文件
`cat > filename`	向文件中输入文本
`cp`	复制文件
`mv`	重命名文件
`rm`	删除文件
`file`	报告文件类型
`cat filename`	显示文件
`head,tail`	显示部分 of a file
`less,more`	增量显示文件

1.2.1.1 `ls`

如果没有任何参数， *`ls`* 命令会列出你当前所在位置的文件。

Exercise1.1. 输入 *`ls`*。有什么显示吗?

Intended outcome. 如果你的目录中有文件，它们将被 `listed` ；如果没有，则不会有输出。

这是标准的 Unix 行为：没有输出并不意味着出现了错误，只是表示没有需要报告的内容。

Exercise1.2. 如果 *`ls`* 命令显示有文件，尝试对其中一个执行 `ls name` 。通过使用选项，例如 `ls -s name` ，你可以获得关于 `name` 的更多信息。

注意事项。如果你拼写错误了名字，或者指定了一个不存在的文件名，你会收到一条错误信息。

`ls` 命令可以为你提供各种信息。除了上面提到的用于显示大小的 `ls -s` ，还有用于 ‘长’ 列表的 `ls -l` 。它显示（我们稍后会讲到的内容，如）所有权和权限，以及大小和创建日期。

备注 1 文件关联有几个日期，分别对应内容的更改、权限的更改以及任何形式的访问。 *`stat`* 命令会显示所有这些日期。

1.2.1.2 `cat`

`cat` 命令（ ‘concatenate’ 的缩写）通常用于显示文件，但它也可以用来创建一些简单的内容。

练习 1.3. 输入 `cat > newfilename` （你可以选择任意文件名）并输入一些文本。以单独一行的 `Control-d` 结束：按住 `Control` 键的同时按 `d` 键。现在使用 *`cat`* 查看该文件的内容：`cat newfilename`。

预期结果。 在第一次使用 *cat* 时，文本是从终端追加到文件的；第二次则是将文件通过 cat 命令输出到终端。你应该在屏幕上看到的内容正是你输入到文件中的内容。

注意事项。 确保在输入的最后一行首个字符就是 Control-d 。如果你真的卡住了，Control-c 通常能帮你脱困。试试这个：用 cat > filename 开始创建一个文件，然后在一行中间按 Control-c 。你的文件内容是什么?

备注 2 你经常会看到用 ^D 来代替 Control-d 的写法。大写字母的使用是出于历史原因：你需要按住 *control* 键再按小写字母。

1.2.1.3 man

unix 命令的主要（虽然不总是最容易理解的）来源是 *man* 命令，意为 “ 手册 ”。通过这种方式获得的描述被称为 手册页。

Exercise1.4. 阅读 ls 命令的 man 页面： man ls。找出某些文件的大小和最后修改的时间 / 日期，例如你刚创建的文件。

Intended outcome. 你找到了 ls -s 和 ls -l 选项吗？第一个列出每个文件的大小，通常以千字节为单位，另一个提供关于文件的各种信息，包括你稍后会学习的内容。

man 命令让你进入一种可以查看长文本文件的模式。这个查看器在 Unix 系统中很常见（它作为 *more* 或 *less* 系统命令提供），所以记住以下导航方式：使用空格键向前翻页，使用 u 键向上翻页。使用 g 跳到文本开头，使用 G 跳到结尾。使用 q 退出查看器。如果你真的卡住了， Control-c 会帮你退出。

Remark 3 如果你已经知道你要找的命令，可以使用 *man* 来获取该命令的在线信息。如果你忘记了命令的名称， *man* -k keyword 可以帮助你找到它。

1.2.1.4 touch

touch 命令创建一个空文件，或者如果文件已存在则更新其时间戳。使用 ls-l 来确认此行为。

1.2.1.5 cp, mv, rm, ln

cp 可用于复制文件（或目录，见下文）： cp file1 file2 复制 file1 并将其命名为 file2。

Exercise1.5. 使用 cp file1 file2 复制一个文件。确认两个文件内容相同。如果你更改原文件，复制文件会发生什么变化吗?

Intended outcome. 你应该看到如果原文件更改或被删除，复制文件不会发生变化。

Things to watch out for. 如果 file2 已经存在，你将收到一个错误or message.
文件可以用 *mv* 重命名，表示 “ 移动 ”。

Exercise 1.6. Rename a file. What happens if the target name already exists?
文件通过 `rm` 删除。此命令很危险：没有撤销操作。因此，你可以使用 `rm-i`（表示 “交互式”），它会在删除每个文件前询问你的确认。详见第 1.2.4 节，了解更激进的删除方式。

有时你想从两个位置引用同一个文件。这不同于有一个副本：你希望能够编辑任意一个，另一个也随之改变。这可以通过 *`ln`* 实现：‘link’。

这段代码创建了一个文件及其链接：

```
$ echo contents > arose
$ cd mydir
$ ln ../arose anyothername
$ cat anyothername
contents
$ echo morestuff >> anyothername
$ cd ..
$ cat arose
contents
morestuff
```

1.2.1.6 `head, tail`

还有更多用于显示文件、文件部分内容或文件信息的命令。
Exercise1.7. 使用 `ls /usr/share/words` 或 `ls /usr/share/dict/words` 来确认系统中是否存在包含单词的文件。现在使用该文件尝试命令 `head`、 `tail`、 `more` 和 `wc`。

> *Intended outcome.* *`head`* 显示文件的前几行， *`tail`* 显示最后几行，*`more`* 使用与 man 页面相同的查看器。阅读这些命令的 man 页面，并尝试增加或减少输出量。*`wc`*（‘wordcount’）命令报告文件中的单词数、字符数和行数。

另一个有用的命令是 *`file`*：它告诉你正在处理的文件类型。
Exercise 1.8. 对各种 ‘foo’ 使用 `file foo`：文本文件、目录或 `/bin/ls` 命令。*Intended outcome.* 其中一些信息你可能无法理解，但需要注意的词是 ‘text’、‘directory’ 或 ‘executable’。
此时建议学习使用文本 *editor*，例如 *emacs* 或 *vi*。

1.2.2 Directories

> **Purpose.** 这里你将学习 Unix 目录树，如何操作它以及如何在其中移动。

本节学习的命令

`ls`	list the contents of directories
`mkdir`	make new directory
`cd`	change directory
`pwd`	display present working directory

A unix 文件系统是一个目录树，其中目录是文件或更多目录的容器。我们将按如下方式显示目录：

```
/ ..... 目录树的根
  bin ..... 二进制程序
  home ..... 用户目录的位置
```

Unix 目录树的根用斜杠表示。执行 `ls /` 以查看根目录下的文件和目录。请注意，根目录并不是你重启个人计算机或登录服务器时的起始位置。

Exercise1.9. 用于查看当前工作目录的命令是 *pwd*。登录时，你的主目录就是你的工作目录。找出你的主目录。

Intended outcome. 你通常会看到类似 `/home/yourname` 或 `/Users/yourname` 的内容。这取决于系统。

执行 `ls` 以查看工作目录的内容。在本节的显示中，目录名后会跟一个斜杠：`dir/`，但该字符不是名称的一部分。你可以使用 `ls-F` 获得此输出，并且可以通过在会话开始时声明 `alias ls=ls -F` 来告诉你的 Shell 始终使用此输出。例如：

```
/home/you/
  adirectory/
  afile
```

创建新目录的命令是 *mkdir*。

Exercise1.10. 使用 *mkdir* `newdir` 创建一个新目录，并用 `ls` 查看当前目录。

Intended outcome. You should see this structure:

```
/home/you/
  newdir/ ..... the new directory
```

Remark 4 如果你需要创建多级目录，可以

```
mkdir sub1
cd sub1
mkdir sub2
cd sub2
## et cetera
```

但使用 `-p` 选项（表示“父目录”）并写成：会更简短。

```
mkdir -p sub1/sub2/sub3
```

which creates any needed intermediate levels.

进入另一个目录，即将其设为你的工作目录的命令是 *cd* （ ‘change directory’ ）。它可以通过以下方式使用：

- `cd` 不带任何参数时， `cd` 会带你到你的主目录。
- `cd <absolute path>` 绝对路径从目录树的根开始，即以 `/` 开头。 `cd` 命令会带你到该位置。
- `cd <relative path>` 相对路径是不以根开始的路径。 `cd` 命令的这种形式会带你到 `<yourcurrentdir>/<relative path>`。

Exercise 1.11. 执行 `cd newdir` 并用 `pwd` 找出你在目录树中的位置。用 `ls` 确认该目录是空的。你如何使用绝对路径到达该位置?

> 预期结果。 `pwd` 应该告诉你 `/home/you/newdir`，然后 `ls` 没有输出，意味着没有内容可列出。绝对路径是 `/home/you/newdir`。

练习 1.12. 让我们快速在此目录下创建一个文件： *touch* `onefile`，以及另一个目录：`mkdir otherdir`。执行 `ls` 并确认有 newfile 和 directory。预期结果。 你现在应该有：

```
/home/you/
|
|  newdir/.....................................................you are here
|     |  onefile
|     |  otherdir/
```

`ls` 命令有一个非常有用的选项：使用 `ls -a` 你可以看到常规文件和隐藏文件，隐藏文件的名称以点开头。在你的新目录中执行 `ls -a` 应该告诉你有以下文件：`/home/you/`

```
|
|  newdir/.....................................................you are here
|     |  .
|     |  ..
|     |  onefile
|     |  otherdir/
```

单个点表示当前目录，双点表示上一级目录。

练习 1.13. 预测执行 `cd ./otherdir/..` 后你将处于何处，并检查你的预测是否正确。

> 预期结果。 单个点将你发送到当前目录，因此不会改变位置
> 不改变任何东西。 `otherdir` 部分使该子目录成为你当前的工作目录。最后， `..`
> 返回上一级。换句话说，这条命令让你回到起点。

由于你的主目录是一个特殊的位置，有一些快捷方式可以 `cd` 到它： `cd` 无参数，`cd ~`，和 `cd $HOME` 都可以让你回到主目录。

进入你的主目录，然后从那里执行 `ls newdir` 来查看你创建的第一个目录的内容，而不必进入该目录。

Exercise1.14. `ls ..` 做什么?

预期结果。 回想一下，.. 表示树中上一级目录：你应该能看到你自己的主目录，以及其他用户的目录。

让我们练习使用单点和双点目录快捷方式。

练习 1.15. 从你的主目录开始：

```
mkdir -p sub1/sub2/sub3
cd sub1/sub2/sub3touch a
```

You now have a file `sub1/sub2/sub3/a`

1. 你如何将它移动到 `sub1/sub2/a`？ 2. 进入： `cd sub1/sub2` 你现在如何将文件移动到 `sub1/a`？ 3. 进入你的主目录： `cd` 你如何将 `sub1/a` 移动到这里？

练习 1.16. 你能使用 `ls` 查看其他人的主目录内容吗？在前一个练习中，你已经看到系统中是否存在其他用户。如果存在，执行 `ls ../thatotheruser`。预期结果。 如果这是你的私人电脑，你可能可以查看

其他用户的目录。如果这是大学的计算机，其他目录很可能受到保护 —— 权限将在下一节讨论 —— 你会得到 `ls:../otheruser: Permission denied`。

尝试使用 `cd` 进入别人的主目录。能成功吗？

你可以用 `cp` 复制目录，但需要添加一个标志来表示递归复制内容： `cp -r`。在你的主目录中再创建一个目录 `somedir`，这样你就有了 `/home/you/`

```
newdir/ ..... you have been working in this one
somedir/ ..... you just created this one
```

`cp -r newdir somedir` （其中 `somedir` 是一个已存在的目录）和 `cp-r newdir thirddir` （其中 `thirddir` 不是一个已存在的目录）之间有什么区别？

1.2.3 权限

Purpose. 本节你将学习如何赋予系统中不同用户对你的文件执行（或不执行）各种操作的权限。

Unix 文件，包括目录，都有权限，表示 “ 谁可以对该文件做什么 ”。对文件可以执行的操作分为三类：

- 读取 `r`：对文件的任何访问（显示、获取信息）且不改变文件内容；
- 写入 `w`：访问文件并更改其内容，甚至其元数据如 “ 修改日期 ” ；
- 执行 `x`：如果文件是可执行的，则运行它；如果是目录，则进入该目录。

可以访问文件的人也分为三类：

- 用户 `u`：拥有该文件的人；
- 组 `g`：拥有者所属的用户组；
- 其他 `o`：其他所有人。

(有关组和所有权的更多内容，请参见第 1.13.3 节。)

这九个权限按顺序渲染

user 组 其他

rwx rwx rwx

例如 `rw-r--r--` 表示所有者可以读写文件，所有者的组和其他人只能读取。

权限也以三位二进制组的数字形式表示，通过让 `r` = 4, `w` = 2, `x` = 1：

rwx

421

常见的代码有 7 = `rwx` 和 6 = `rw`。你会发现许多文件的权限是 755，表示一个所有人都可以运行但只有所有者可以更改的可执行文件，或者 644，表示一个所有人都可以查看但只有所有者可以修改的数据文件。你可以通过 *chmod* 命令设置权限：

```
chmod <permissions> file         # just one file
chmod -R <permissions> directory # directory, recursively
```

Examples:

```
chmod 766 file  # set to rwxrw-rw-
chmod g+w file  # give group write permission
chmod g=rx file # set group permissions
chod o-w  file  # take away write permission from others
chmod o=  file  # take away all permissions from others.chmod g+r,
o-x file # give group read permission
# remove other execute permission
```

man 手册页给出了所有选项。

练习 1.17. 创建一个文件 `foo` 并执行 `chmod u-r foo`。你现在能检查它的内容吗？再次使文件可读，这次使用数字代码。现在使文件对你的同学可读。通过让他们中的一个读取内容来检查。预期结果。1. 只有当所在文件夹可读时，其他人才能访问该文件。你能弄清楚如何做到这一点吗？2. 当你自己使文件 “ 不可读 ” 后，你仍然可以 `ls` 它，但不能 `cat` 它：那会显示 “ 权限被拒绝 ” 的信息。

创建一个文件 `com`，内容如下：

```
#!/bin/sh
echo "Hello world!"
```

这是一个合法的 shell 脚本。当你输入 `./com` 会发生什么？你能让脚本执行吗？

在这三种权限类别中，‘ 你 ’ 和 ‘ 其他人 ’ 的指代是明确的。那么 ‘ 组 ’ 呢？我们将在第 1.13 节中详细讲解。

Exercise1.18. 假设你是一名讲师，想为学生创建一个“dropbox”目录，用于提交作业。该目录的合适权限模式是什么？（假设你有同组的助教，他们也需要能够查看目录内容。换句话说，组权限应与所有者权限相同。）

Remark 5 还有一些更隐晦的权限。例如，*setuid* 位声明程序应以创建者的权限运行，而不是执行该程序的用户的权限。这对于像 `passwd` 或 `mkdir` 这样的系统工具非常有用，它们会修改密码文件和目录结构，这些操作需要 root 权限。得益于 *setuid* 位，用户可以运行这些程序，这些程序被设计成用户只能修改自己的密码条目和自己的目录。*setuid* 位通过 *chmod* 设置：`chmod 4ugo file`。

1.2.4 通配符

你已经看到 `ls filename` 给你关于那个文件的信息，而 `ls` 给你当前目录下的所有文件。为了查看文件名满足某些条件的文件，存在 通配符 机制。以下是存在的通配符：

- `*` any number of characters
- `?` any character.

Example:

```
%% ls
s       sk      ski     skiing  skill
%% ls ski*
ski     skiing  skill
```

第二个选项列出所有以 `ski` 开头，后面跟任意数量其他字符的文件；下面你会看到在不同的上下文中 `ski*` 意味着 ‘`sk` 后跟任意数量的 `i` 字符’。很混乱，但事实就是这样。

你可以将 `rm` 与通配符一起使用，但这可能很危险。

```
rm -f foo    ## remove foo if it exists
rm -r foo    ## remove directory foo with everything in it
rm -rf foo/* ## delete all contents of foo
```

Zsh 注意。使用通配符 `rm foo*` 删除时，如果没有这样的文件会出错。设置 `setopt +onomatch` 以允许没有匹配项发生。

1.3 Text searchingand regular expressions

Purpose. 在本节中，您将学习如何在文件中搜索文本。

本节需要至少一个包含一定文本量的文件。例如，你可以从 http://www.lipsum.com/feed/html 获取随机文本。

grep 命令可用于在文件中搜索文本表达式。

练习 1.19. 使用 `grep q yourfile` 在你的文本文件中搜索字母 `q`，并使用 `grep q *` 在目录中的所有文件中搜索它。尝试其他一些搜索。

预期结果。 在第一种情况下，你会得到包含 `q` 的所有行的列表；在第二种情况下，`grep` 还会报告匹配所在的文件名： `qfile:thisline has q in it`. 注意事项。如果你要查找的字符串不存在， `grep` 将不会输出任何内容。请记住，如果没有要报告的内容，这是 Unix 命令的标准行为。

除了搜索字面字符串外，你还可以查找更通用的表达式。

`^`	行的开头
`$`	行尾
`.`	任意字符
`*`	任意次数重复
`[xyz]`	字符集合 `xyz` 中的任意字符

这看起来像你刚才看到的通配符机制（章节 1.2.4 ），但它有细微的不同。将上面的例子与以下内容进行比较:

```
%% cat s
sk
ski
skill
skiing
%% grep "ski*" s
sk
ski
skill
skiing
```

在第二种情况下，你搜索的是一个由 `sk` 和任意数量的 `i` 字符组成的字符串，包括零个。

更多示例：你可以找到

- 所有包含字母 ‘q’ 的行，使用 `grep q yourfile`；
- 所有以 ‘a’ 开头的行，使用 `grep "^a" yourfile` （如果你的搜索字符串包含特殊字符，最好用引号将其括起来）；
- 所有以数字结尾的行，使用 `grep "[0-9]$" yourfile`。

Exercise1.20. 构造用于查找的搜索字符串

- 以大写字母开头的行，和
- 包含恰好一个字符的行。*Intendedoutcome.* 对于第一个，使用范围字符 `[]`,，对于第二个使用句点

来匹配任意字符。

Exercise1.21. 向你的测试文件中添加几行 `x = 1`, `x  = 2`, `x   = 3` （即在 `x` 和等号之间有不同数量的空格），并制作 `grep` 命令来搜索所有对 `x` 的赋值。

上表中的字符具有特殊含义。如果你想搜索该实际字符，必须转义它。

练习 1.22. 创建一个测试文件，其中包含 `abc` 和 `a.c`，分别位于不同的行。尝试命令 `grep "a.c" file`, `grep a\.c file`, `grep "a\.c" file`. 预期结果。你会看到句点需要被转义，搜索字符串需要加引号。如果缺少任一项，你会发现 `grep` 也会找到 `abc` 字符串。

1.3.1 使用 `cut` 切割行

另一个用于编辑行的工具是 *cut*，它可以切割一行并显示其中的某些部分。例如，

```
cut -c 2-5 myfile
```

将显示 `myfile` 每行中位置 2–5 的字符。创建一个测试文件并验证此示例。

也许更有用的是，你可以给 `cut` 一个分隔符字符，并让它在该分隔符出现处拆分一行。例如，你的系统很可能有一个文件 `/etc/passwd` 包含用户信息 [1]，每行由冒号分隔的字段组成。例如：

```
daemon:*:1:1:System Services:/var/root:/usr/bin/false
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
```

第七个也是最后一个字段是用户的登录 shell； `/bin/false` 表示该用户无法登录。

You can display users and their login shells with:

```
cut -d ":" -f 1,7 /etc/passwd
```

这告诉 `cut` 使用冒号作为分隔符，并打印第 1 和第 7 字段。

1.4 其他有用的命令： `tar`

tar 命令代表“tape archive”，即它最初是用于在磁带上打包文件。（“archive”部分来源于 *ar* 命令。）如今，它被用来将文件打包在一起以便在网站等处分发：如果你想发布一个包含数百个文件的库，这个命令会将它们打包成一个单一文件。

两个最常用的选项是用于

1. 创建一个 tar 文件：

tar fc package.tar directory_with_stuff 发音

为“tar file create”，并且

1. 传统上情况是这样的；在 Mac OS 中，关于用户的信息保存在其他地方，而该文件仅包含系统服务。

2. unpacking a tar file:

```
tar fx package.tar
# this creates the directory that was packaged
```

发音为 ‘tar file extract’。

文本文件通常可以被大幅压缩，因此为 *gzip* 添加 `z` 压缩是个好主意：

```
tar fcz package.tar.gz directory_with_stuff
tar fx package.tar.gz
```

将 ‘gzipped’ 文件命名为 `package.tgz` 也很常见。

1.5 命令执行

如果你在 shell 中输入内容，实际上是在请求底层解释器执行一个命令。有些命令是内置的，其他的可能是存储在某个系统位置或你自己的账户中的程序名称。本节将深入探讨这些机制。

Remark6 像任何优秀的编程语言一样，*shell* 语言也支持注释。任何以哈希字符 `#` 开头的行都会被忽略。

Zsh note. 在 Apple 的 zsh 中，注释字符被禁用。请

```
setopt interactivecomments
```

启用它。

1.5.1 搜索路径

本节学习的命令

which	可执行命令的位置
type	命令、函数等的描述

Purpose. 本节中您将学习 Unix 在输入命令名称时如何确定执行操作。

如果你输入一个命令，比如 `ls`，shell 不仅仅依赖于命令列表：它实际上会去搜索名为 `ls` 的程序。这意味着你可以有多个同名的不同命令，哪个命令被执行取决于哪个命令先被找到。

练习 1.23. 你可能认为的 “Unix 命令” 通常只是系统目录中的可执行文件。执行 *which* `ls`，并对结果执行 `ls -l`。

预期结果。 `ls` 的位置类似于 `/bin/ls`。如果你 `ls` 它，你会看到它可能归 root 所有。它的可执行权限可能对所有用户都已设置。

unix 搜索命令的位置称为搜索路径，它存储在环境变量中（更多细节见下文） *PATH*。

练习 1.24. 执行 `echo $PATH`。你能找到 `cd` 的位置吗？同一位置还有其他命令吗？当前目录 '.' 是否在路径中？如果没有，执行 `export PATH=".:$PATH"`。现在在当前目录创建一个可执行文件 `cd` （基础见上文），然后执行 `cd`。

预期结果。路径将是一个以冒号分隔的目录列表，例如 `/usr/bin:/usr/local/bin:/usr/X11R6/bin`。如果工作目录在路径中，它可能会在末尾：`/usr/X11R6/bin:.`，但很可能不在那里。如果你将 '.' 放在路径开头，unix 会先找到本地的 `cd` 命令，而不是系统命令。

有些人认为将工作目录放入路径存在安全风险。如果你的目录是可写的，别人可能会在你的目录中放置一个名为 `cd` （或任何其他系统命令）的恶意脚本，而你会在不知情的情况下执行它。

在当前目录中执行程序的最安全方法是：

```
./my_program
```

这对编译程序和 shell 脚本都适用；章节 1.9.1 .

Remark7 并非所有 *Unix* 命令都对应可执行文件。 *type* 命令提供更多信息 *than* *which*:

```
$ type echo
echo is a shell builtin
$ type \ls
ls is an alias for ls -F
$ unalias ls
$ type ls
ls is /bin/ls
$ type module
module is a shell function from /usr/local/Cellar/lmod/8.7.2/init/zsh
```

1.5.2 别名

可以将您自己的命令定义为现有命令的别名。

Exercise 1.25. 执行 `alias chdir=cd` 并确认 `chdir` 现在的行为与 `cd` 完全相同。执行 `alias rm='rm -i'`；查阅 man 手册中对此的解释。有些人认为这个别名是个好主意；你能理解为什么吗？ *Intended outcome.* `rm` 的 `-i` 'interactive' 选项使得命令在每次删除前都会询问确认。由于 unix 没有像 Windows 或 Mac OS 那样需要显式清空的回收站，这可能是个好主意。

1.5.3 Command sequencing

在单个命令行上执行多个命令有多种方式。

1.5.3.1 简单的顺序执行

First of all, you can type

```
command1 ; command2
```

如果你多次重复相同的两个命令，这样做很方便：你只需按一次上箭头就能重复执行它们。

有一个问题：如果你输入

```
cc -o myprog myprog.c ; ./myprog
```

并且编译失败时，程序仍然会被执行，如果存在旧版本的可执行文件，则会使用旧版本。这非常令人困惑。

一个更好的方法是：

```
cc -o myprog myprog.c && ./myprog
```

只有当第一个命令成功时，才执行第二个命令。

1.5.3.2 流水线

与其从文件读取输入，或将输出发送到文件，不如将两个命令连接起来，使第二个命令以第一个命令的输出作为输入。其语法是 `cmdone | cmdtwo`；这称为流水线。例如，

`grep a yourfile | grep b` 查找所有同时包含 `a` 和 `b` 的行。

练习 1.26. 构建一个管道，统计你的文件中包含字符串 `th` 的行数。使用 `wc` 命令（见上文）来进行计数。

1.5.3.3 Backquoting

还有几种组合命令的方法。假设你想稍微美观地展示 `wc` 的结果。输入以下命令

```
echo The line count is wc -l foo
```

其中 `foo` 是一个已存在文件的名称。获取实际行数并回显的方法是使用 *backquote*：

```
echo The line count is `wc -l foo`
```

任何位于反引号之间的内容都会在命令行的其余部分被求值之前执行。

Exercise1.27. `wc` 在这里的用法是打印文件名。你能找到一种方法防止这种情况发生吗?

还有另一种乱序求值的机制：

```
echo "There are $( cat Makefile | wc -l ) lines"
```

这种机制使得嵌套命令成为可能，但出于兼容性和遗留原因，当不需要嵌套时，反引号仍可能更可取。

1.5.3.4 在子 *Shell* 中分组

假设你想对连续的几个命令应用输出重定向:

```
configure ; make ; make install > installation.log 2>&1
```

这只会捕获最后一个命令。例如，你可以将这三个命令分组到一个子 Shell 中，并捕获该子 Shell 的输出:

```
( configure ; make ; make install ) > installation.log 2>&1
```

1.5.4 退出状态

命令可能会失败。如果你在命令行输入单个命令，你会看到错误，并在输入下一个命令时相应地采取行动。当失败的命令出现在脚本中时，你必须告诉脚本如何相应地行动。为此，你使用命令的 *exitstatus*：这是一个值（成功为零，失败为非零），存储在一个内部变量中，你可以通过 `$?` 访问它。

示例。假设我们有一个不可写的目录 `[testing] ls -ld nowrite/`

```
dr-xr-xr-x  2 eijkhout  506  68 May 19 12:32 nowrite//
[testing] cd nowrite/
```

并尝试在那里创建一个文件:

```
[nowrite] cat ../newfile
#!/bin/bash
touch $1
echo "Created file: $1"
[nowrite] newfile myfile
bash: newfile: command not found
[nowrite] ../newfile myfile
touch: myfile: Permission denied
Created file: myfile
[nowrite] ls
[nowrite]
```

脚本报告文件已创建，尽管实际上并未创建。
改进的脚本 :

```
[nowrite] cat ../betterfile#!/bin/bash
touch $1if [ $? -eq 0 ] ; then
echo "Created file: $1"else
echo "Problem creating file: $1"fi
[nowrite] ../betterfile myfile
touch: myfile: Permission denied
Problem creating file: myfile
```

1.5.5 进程和作业

本节学习的命令

`ps`	列出（所有）进程
`kill`	终止一个进程
`CTRL-c`	终止前台作业
`CTRL-z`	挂起前台作业
`jobs`	显示所有作业的状态
`fg`	将最后一个挂起的作业调回前台
`fg %3`	将特定作业调到前台
`bg`	在后台运行最后一个暂停的作业

Unix 操作系统可以同时运行多个程序，通过轮流处理列表，每次只给每个程序一小段时间运行。命令 *`ps`* 可以告诉你当前所有正在运行的程序。

Exercise1.28. 输入 `ps`。当前有多少程序正在运行？默认情况下， `ps` 只显示你明确启动的程序。使用 `ps guwax` 可以获得所有正在运行程序的详细列表。当前有多少程序正在运行？其中有多少属于 root 用户，有多少属于你？

> *Intended outcome.* 要统计属于某个用户的程序数量，可以将 `ps` 命令通过合适的 `grep` 管道传递，然后再通过 `wc`。

在这个 `ps` 的长列表中，第二列包含了进程号。有时拥有这些信息很有用：如果某个程序行为异常，你可以用 *`kill`* 来处理它，方法是

```
kill 123456
```

其中 12345 是进程号。

上面解释的 `cut` 命令可以从一行中截取特定位置：输入 `ps guwax | cut -c10-14`。

要获取所有正在运行的进程的动态信息，请使用 `top` 命令。阅读手册页以了解如何按 CPU 使用率排序输出。

在 shell 中启动的进程称为 *jobsjob (unix)*。除了进程号外，它们还有一个作业号。我们现在将探讨如何操作作业。

当你输入命令并按回车时，该命令在运行期间成为 *foreground process*。同时运行的其他所有进程都是 *background process*。

制作一个可执行文件 `hello`，内容如下：

```
#!/bin/sh
while [ 1 ] ; do
  sleep 2
  date
done
```

然后输入 `./hello`。

练习 1.29. 输入 `Control-z`。这会暂停前台进程。它会给你一个数字，比如 `[1]` 或 `[2]`，表示这是第一个或第二个被暂停或放到后台的程序。现在输入 `bg` 将该进程放到后台。通过按回车确认没有前台进程，然后执行 `ls`。

> 预期结果。 将进程放到后台后，终端再次可用，可以接受前台命令。如果按回车，你应该会看到命令提示符。然而，后台进程仍然会持续产生输出。

练习 1.30. 输入 `jobs` 查看当前会话中的进程。如果你刚才放到后台的进程是编号 1，输入 `fg %1`。确认它又变成了前台进程。

> 预期结果。 如果 Shell 正在前台执行程序，它将不会接受命令输入，因此按回车只会产生空行。

练习 1.31. 当你再次将 `hello` 脚本设为前台进程时，可以使用 `Control-c` 将其终止。试试看。再次启动脚本，这次作为 `./hello &`，它会立即将其放入后台。你还应该看到类似 `[1]`12345 的输出，这告诉你这是你放入后台的第一个作业，12345 是它的进程 ID。使用 `kill %1` 终止脚本。再次启动它，然后使用进程号终止它。

> 预期结果。 使用进程号的 `kill 12345` 命令通常足以终止正在运行的程序。有时需要使用 `kill -9 12345`。

1.5.6 Shell 自定义

上文提到，`ls -F` 是查看哪些文件是普通文件、可执行文件或目录的简便方法；通过输入 `alias ls='ls -F'`，每次调用时 `ls` 命令会自动扩展为 `ls -F`。如果你希望每个登录会话都具有此行为，可以将 `alias` 命令添加到你的 `.profile` 文件中。除了 `sh`/`bash`，其他 Shell 有各自用于此类自定义的文件。

1.6 输入 / 输出重定向

> **Purpose.** 在本节中，您将学习如何将一个命令的输出作为另一个命令的输入，以及如何将命令连接到输入和输出文件。

到目前为止，您使用的 unix 命令都是从键盘或命令行中指定的文件获取输入；它们的输出显示在屏幕上。还有其他方式可以从文件提供输入，或将输出存储到文件中。

1.6.1 输入重定向

`grep` 命令有两个参数，第二个是文件名。你也可以写成 `grep string< yourfile`，其中小于号表示输入将来自指定的文件，`yourfile`。这被称为 <input redirection>。

1.6.2 标准文件

Unix 有三个处理输入和输出的标准文件：

标准文件	
`stdin`	是为进程提供输入的文件。
`stdout`	是进程输出写入的文件。
`stderr`	是错误输出写入的文件。

在一个交互式会话中，所有三个文件都连接到用户终端。使用输入或输出重定向意味着输入被取自或输出被发送到不同于终端的文件。

1.6.3 输出重定向

就像输入一样，你也可以重定向程序的输出。在最简单的情况下， `grep string yourfile > outfile` 会将通常发送到终端的内容重定向到 `outfile`。如果输出文件不存在，则会创建该文件，否则会覆盖它。（要追加，请使用 `grep textyourfile >> outfile`。）

练习 1.32. 取上一节中的一个 grep 命令，将其输出发送到一个文件。检查该文件的内容是否与之前显示在屏幕上的内容完全相同。搜索一个文件中不存在的字符串，并将输出发送到一个文件。这对输出文件意味着什么？

> 预期结果。搜索文件中不存在的字符串不会有终端输出。如果你将此 `grep` 的输出重定向到文件，则会得到一个大小为零的文件。使用 `ls` 和 `wc` 进行验证。

练习 1.33. 生成一个包含你的信息的文本文件：

```
My user name is:eijkhoutMy home directory is:
/users/eijkhoutI made this script on:
isp.tacc.utexas.edu
```

> where you use the commands `whoami`, `pwd`, `hostname`.
> Bonus points if you can get the 'prompt' and output on the same line.
> Hint: see section 1.5.3.3.

有时你想运行一个程序，但忽略输出。为此，你可以将输出重定向到系统 *nulldevice*：`/dev/null`。

```
yourprogram >/dev/null
```

这里有一些有用的惯用法：

习语

`program 2>/dev/null`	仅将错误发送到空设备
`program >/dev/null 2>&1`	将输出发送到 dev-null，将错误发送到输出 注意指定的反直觉顺序
	ca- 说明！
`program 2>&1 | less`	将输出和错误发送到 `less`

1.7 Shell environmentvariables

上面你遇到了 `PATH`，这是一个 shell 或环境变量的例子。这些是 shell 已知的变量，所有由 shell 运行的程序都可以使用它们。虽然 `PATH` 是一个内置变量，你也可以定义你自己的变量，并在 shell 脚本中使用它们。

Shell 变量大致分为以下几类：

- 特定于 shell 的变量，例如 `HOME` 或 `PATH`。
- 特定于某些程序的变量，例如 `TEXINPUTS` 对于 TEX/LATEX。
- 你自己定义的变量；见下文。
- 由控制结构定义的变量，例如 *`for`*；见下文。

你可以通过输入 *`env`* 查看 shell 已知的所有变量的完整列表 .

Remark 8 这不包括你自己定义的变量，除非你 *`export`* 它们；见下文。

Exercise1.34. 通过输入 `echo $PATH` 检查 `PATH` 变量的值。还可以通过将 `env` 通过 `grep` 管道传输来查找 `PATH` 的值。

我们首先探讨与 shell 变量相关的美元符号的用法。

1.7.1 shell 变量的使用

你可以通过在 shell 变量名前加上美元符号来获取其值。输入以下内容并检查输出：

```
echo x
echo $x
x=5
echo x
echo $x
```

你会看到 shell 将所有内容视为字符串，除非你明确告诉它取变量的值，即在名称前加上美元符号。未定义的变量将打印为空字符串。

Sshell 变量可以通过多种方式设置。最简单的方法是像其他编程语言一样通过赋值。
languages.

当你做下一个练习时，记住 shell 是一种基于文本的语言，这一点很重要。

练习 1.35. 在命令行输入 `a=5` 。使用 `echo` 命令检查其值。将变量 `b` 定义为另一个整数。检查其值。

现在通过 `echo` 查看 `a+b` 和 `$a+$b` 的值，或者先给它们赋值。

预期结果。Shell 在这里不会执行整数加法：相反，你会得到一个带有加号的字符串。（你将在第 1.10.1 节看到如何对变量进行算术运算。）

注意事项。注意等号两边不要有空格；打印值时也要确保使用美元符号。

1.7.2 导出变量

以这种方式设置的变量将在你当前的这个 shell 中所有后续命令中可见，但不会对你新启动的 shell 中的命令可见。为此你需要使用 *`export`* 命令。重现以下会话（方括号表示命令提示符）：

```
[] a=20
[] echo $a20
[] /bin/bash
[] echo $a

[] exit
exit
[] export a=21
[] /bin/bash
[] echo $a21
[] exit
```

你也可以临时设置一个变量。重放此场景：

1. 找到一个没有值的环境变量：

```
[] echo $b

[]
```

2. 编写一个简短的 shell 脚本来打印这个变量：

```
[] cat > echob
#!/bin/bash
echo $b
```

当然，还要使其可执行： `chmod +x echob`。现在调用

3. 该脚本，前面加上一个变量的设置，variable b:

```
[] b=5 ./echob
5
```

这种语法是将值作为前缀设置，而不使用单独的命令，这样设置的值仅对该命令有效。

4. 显示该变量仍未定义：

```
[] echo $b

[]
```

That is, you defined the variable just for the execution of a single command.

在章节 1.8 中你将看到 `for` 结构也定义了一个变量；章节 1.9.1 展示了一些适用于 shell 脚本的更多内置变量。

如果你想取消设置一个环境变量，可以使用 *unset* 命令。

1.8 控制结构

像任何好的编程系统一样，shell 也有一些控制结构。它们的语法需要一些时间来适应。（不同的 shell 有不同的语法；在本教程中我们只讨论 bash shell。

1.8.1 条件语句

bash shell 的 *conditional* 预期地被称为 *if*，它可以写成多行：

```
if [ $PATH = "" ] ; then
  echo "Error: path is empty"
fi
```

或者写在一行：

```
if [ `wc -l file` -gt 100 ] ; then echo "file too long" ; fi
```

(反引号在第 1.5.3.3 节中有解释。) 定义了许多测试，例如 `-fsomefile` 用于检测文件是否存在的测试。修改你的脚本，使其在文件不存在时报告 `-1` 。

The syntax of this is finicky:

- *if* 和 *elif* 后面跟着一个条件语句，后面跟着分号。
- 条件语句的括号两边需要有空格。
- *else* 的 *then* 后面没有分号：它们后面紧跟着某个命令。

练习 1.36. Bash 条件语句有一个 *elif* 关键字。你能预测你会得到什么错误吗：

```
if [ something ] ; then
  foo
else if [ something_else ] ; then
  bar
fi
```

把代码写出来，看看你是否正确。

Zshnote. *zsh* shell 具有扩展的条件语法，使用双中括号。例如，模式匹配：

```
if [[ $myvar == *substring* ]] ; then ....
```

1.8.2 循环

除了条件语句，Shell 还有循环。一个 *for* 循环看起来像

```
for var in listofitems ; do
  something with $var
done
```

它执行以下操作：

- 对于 `listofitems` 中的每个项，变量 `var` 被设置为该项，并且
- 循环体被执行。

作为一个简单的例子：

```
for x in a b c ; do echo $x ; done
a
b
c
```

在一个更有意义的例子中，下面是如何备份你所有的 `.c` 文件：

```
for cfile in *.c ; do
  cp $cfile $cfile.bak
done
```

Shell 变量可以通过多种方式操作。执行以下命令以查看如何从变量中删除尾随字符：

```
[] a=b.c
[] echo ${a%.c}
b
```

(参见章节 1.10 关于展开。）以此为提示，编写一个循环，将你所有的 `.c` 文件重命名为 `.x` files.

上述结构对单词进行循环，例如 `ls` 的输出。要进行数字循环，使用命令 *seq*：

```
[shell:474] seq 1 5
1 2 3 4 5
```

对一系列数字进行循环通常看起来像

```
for i in `seq 1 ${HOWMANY}` ; do echo $i ; done
```

注意 反引号，它是必须的，以便在评估循环之前执行 `seq` 命令。

你可以使用 *break* 跳出循环；这甚至可以带有一个数字参数，指示要跳出多少层循环。

1.9 脚本编写

unix Shell 也是编程环境。你将在本节中更多地了解 unix 的这一方面。

1.9.1 如何执行脚本

可以编写 unix shell 命令的程序。首先你需要知道如何将程序放入文件并执行它。创建一个文件 `script1`，内容包含以下两行：

```
#!/bin/bash
echo "hello world"
```

并在命令行输入 `./script1`。结果？使文件可执行后再试一次。

Zsh 注意。Bash 脚本 如果你使用 `zsh`，但你有以前写的 `bash` 脚本，它们仍然会继续工作。‘hash-bang’ 行决定了哪个 shell 执行脚本，并且完全可以在你的脚本中使用 `bash`，同时交互式使用 `zsh`。

为了编写你希望从任何地方调用的脚本，人们通常将它们放在家目录中的一个目录 `bin`。然后你会将该目录添加到你的 搜索路径，包含在 *PATH*；参见章节 1.5.1。

1.9.2 脚本参数

你可以带选项和参数调用一个 shell 脚本：

```
./my_script -a file1 -t -x file2 file3
```

您现在将学习如何在您的脚本中整合此功能。

首先，所有命令行参数和选项都作为变量 `$1`、`$2` 等在脚本中可用，命令行参数的数量作为 `$#` 可用：

```
#!/bin/bash

echo "The first argument is $1"
echo "There were $# arguments in all"
```

形式上：

变量	meaning
`$#`	number of arguments
`$0`	the name of the script
`$1,$2,...`	the arguments
`$*,$@`	the list of all arguments

Exercise1.37. 编写一个脚本，接受一个文件名参数作为输入，并报告该文件中有多少行。编辑你的脚本以测试该文件是否少于 10 行（使用 `foo -lt bar` test），如果是，则 `cat` 该文件。提示：你需要在 test 中使用反引号。为你的脚本添加一个测试，以便在没有任何参数调用时给出有用的提示信息。

解析参数的标准方法是使用 *`shift`* 命令，它会从参数列表中弹出第一个参数。然后按顺序解析参数涉及查看 `$1`，移位，以及查看新的 `$1`。

Code:

```
// arguments.sh
while [ $# -gt 0 ] ; do
    echo "argument: $1"
    shift
done
```

Output

```
[code/shell] arguments:

missing snippet
code/shell/arguments.runout :
looking in codedir=code missing
snippet code/shell/arguments.runout
: looking in codedir=code
```

Exercise1.38. 编写一个脚本 `say.sh`，打印其文本参数。然而，如果你用

```
./say.sh -n 7 "Hello world"
```

它应该按你指定的次数打印。使用选项 `-u`:

```
./say.sh -u -n 7 "Goodbye cruel world"
```

应该以大写打印消息。确保参数的顺序无关紧要，并且对任何无法识别的选项给出错误消息。

变量 `$@` 和 `$*` 对双引号的行为不同。假设我们计算 e
`myscript "1 2" 3`，然后

- 使用 `$*` 是去除引号后的参数列表：`myscript 1 2 3`。
- 使用 `"$*"` 是去除引号后带引号的参数列表：`myscript "1 2 3"`。
- 使用 `"$@"` 保留的引号：`myscript "1 2" 3`。

1.9.3 错误处理

脚本，像任何其他类型的程序一样，可能会因某些运行时条件而失败。

1. 如果没有明确的错误信息，您至少可以通过添加一行重新运行您的脚本

```
set -x
```

该行在执行前将每个命令回显到终端。2. 每个脚本都有一个返回码，包含在变量 *`$?`* 中，您可以检查它。3. 脚本中的某些命令也可能失败，但脚本继续运行。您可以通过以下方式防止这种情况发生

```
set -e
```

这会使脚本在任何命令失败时中止。额外的选项

```
set -o pipefail
```

will catch errors in a pipeline.

Here is an idiom for being a little more informative about errors:

```
errcode=0
some_command || errcode=$?
if [ $errcode -ne 0 ] ; then
  echo "ERROR: some_command failed with code=$errcode"
  exit $errcode
fi
```

关键的第二行包含一个 “ 或 ” 条件：要么 `some_command` 成功，要么你将 `errcode` 设置为它的退出码。这个连接总是成功的，所以现在你可以检查退出码。

1.10 扩展

Shell 对命令行执行各种扩展，也就是用不同的文本替换命令行的部分内容。

大括号扩展：

```
[] echo a{b,cc,ddd}e
abe acce addde
```

这可以用来删除某个基本文件名的所有扩展：

```
[] rm tmp.{c,s,o}  # delete tmp.c tmp.s tmp.o
```

波浪号扩展提供你自己或他人的主目录：

```
[] echo ~
/share/home/00434/eijkhout
[] echo ~eijkhout
/share/home/00434/eijkhout
```

参数扩展给出 shell 变量的值：

```
[] x=5
[] echo $x
5
```

未定义的变量不会给出错误信息：

```
[] echo $y
```

参数扩展有许多变体。上面你已经看到你可以去除尾部字符：

```
[] a=b.c
[] echo ${a%.c}
b
```

以下是处理未定义变量的方法：

```
[] echo ${y:-0}
0
```

反引号机制（章节 1.5.3.3 中介绍）被称为命令替换。它允许你计算命令的一部分并将其作为另一个命令的输入。例如，如果你想询问命令 `ls` 是什么类型的文件，可以执行

```
[] file `which ls`
```

这首先计算 `which ls`，得到 `/bin/ls`，然后计算 `file /bin/ls`。另一个例子是，我们对整个管道使用反引号，并对结果进行测试：

```
[] echo 123 > w[] cat w123[] wc -c w4 w
[] if [ `cat w | wc -c` -eq 4 ] ; then echo four ; fi
four
```

1.10.1 算术扩展

Unix shell 编程非常侧重于文本处理，但也可以进行算术运算。算术替换告诉 shell 将参数的扩展视为数字：

```
[] x=1
[] echo $((x*2))
2
```

整数范围可以按如下方式使用：

```
[] for i in {1..10} ; do echo $i ; done
1 2 3 4 5 6 7 8 9 10
```

(but see also the *seq* command in section 1.8.2.)

1.11 启动文件

在本教程中，您已经看到几种自定义您的 shell 行为的机制。例如，通过设置 `PATH` 变量，您可以扩展 shell 查找可执行文件的位置。其他环境变量（章节 1.7 ）您也可以为自己的目的引入。许多这些自定义需要应用于每个会话，因此您可以有 ***shell 启动文件*** ，它们将在任何会话开始时被读取。

启动文件中常见的操作是定义 *alias*：

```
alias grep='grep -i'
alias ls='ls -F'
```

以及设置自定义命令行 提示符。

启动文件的名称取决于您的 shell： *.bashrc* 对于 Bash， *.cshrc* 对于 C-shell， *.zshrc* 对于 Z-shell。这些文件在每次登录时都会被读取（详见下文），但您也可以直接 *source* 它们：

```
source ~/.bashrc
```

例如，如果您编辑了启动文件，就会这样做。

Unfortunately, there are several startup files, and which one gets read is a complicated functions of circumstances. Here is a good common sense guideline[2]:

- 拥有一个 `.profile` ，它什么也不做，只是读取 `.bashrc`：

```
# ~/.profile
if [ -f ~/.bashrc ]; then
    source ~/.bashrc
fi
```

- 您的 `.bashrc` 执行实际的自定义操作：

```
# ~/.bashrc
# make sure your path is updated
if [ -z "$MYPATH" ]; then
  export MYPATH=1
  export PATH=$HOME/bin:$PATH
fi
```

1.12 Shell 交互

Unix 的交互式使用，与脚本编写（章节 1.9 ）不同，是用户与 shell 之间复杂的对话。你，作为用户，输入一行，按回车，shell 尝试解释它。有几种情况。

- 你输入的行包含一个完整的命令，例如 `ls foo`：shell 将执行该命令。

2. 非常感谢 Robert McLay 的发现。

- 你可以在一行中放置多个命令，用分号分隔：`mkdir foo; cd foo`。Shell 会按顺序执行这些命令。
- 你的输入行不是一个完整的命令，例如 `while [ 1]`。Shell 会识别出还有后续内容，并使用不同的提示符来显示它正在等待命令的剩余部分。
- 你的输入行本来是一个合法命令，但你想在第二行继续输入更多内容。在这种情况下，你可以在输入行末尾加上反斜杠字符，Shell 会识别出需要延迟执行你的命令。实际上，反斜杠会隐藏（*escape*）回车。

当 Shell 收集到一条命令行准备执行时，通过使用一行或多行输入，或者仅使用其中一部分，如刚才所述，它将对命令行进行扩展（章节 1.10）。然后它会将命令行解释为命令和参数，并继续使用找到的参数调用该命令。

这里有一些细微之处。如果你输入 `ls *.c`，那么 Shell 会识别通配符字符并将其扩展为一条命令行，例如 `ls foo.c bar.c`。然后它会调用 `ls` 命令，参数列表为 `foo.c bar.c`。注意 `ls` 不会将 `*.c` 作为参数接收！在你确实希望 unix 命令接收带有通配符的参数的情况下，你需要对通配符进行转义，以防止 Shell 进行扩展。例如，`find . -name \*.c` 会使 Shell 调用 `find`，参数为 `.-name *.c`。

1.13 系统和其他用户

1.13.1 System information

上述大部分内容适用于任何 Unix 或 Linux 系统。有时你需要了解系统的详细信息。以下内容将告诉你系统的运行状况：

top 系统上正在运行的进程；使用 `top -u` 来按它们当前占用的 CPU 时间排序。（在 Linux 上，也可以尝试 `vmstat` 命令。） *ptime* 自上次重启以来已经过去了多长时间?
u

有时你想知道自己实际使用的是哪个系统。通常你可以从 *uname* 中获取一些信息：

```
$ uname -a
Linux staff.frontera.tacc.utexas.edu 3.10.0-1160.45.1.el7.x86_64 #1 SMP Wed Oct 13 17:20:51
    UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
```

这仍然无法告诉你使用的是哪个 *Linux distribution*。为此，以下某些方法可能有效：

```
$ lsb_release -a
LSB Version:    :core-4.1-amd64:core-4.1-noarch:cxx-4.1-amd64:cxx-4.1-noarch:desktop-4.1-amd64
    :desktop-4.1-noarch:languages-4.1-amd64:languages-4.1-noarch:printing-4.1-amd64:printing
    -4.1-noarch
Distributor ID: CentOS
Description:    CentOS Linux release 7.9.2009 (Core)
Release:        7.9.2009
Codename:       Core
```

or

```
$ ls /etc/*release
/etc/centos-release  /etc/os-release@  /etc/redhat-release@  /etc/system-release@
$ cat /etc/*release
CentOS Linux release 7.9.2009 (Core)
NAME="CentOS Linux"
VERSION="7 (Core)"
ID="centos"
ID_LIKE="rhel fedora"
VERSION_ID="7"
PRETTY_NAME="CentOS Linux 7 (Core)"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:centos:centos:7"
HOME_URL="https://www.centos.org/"
BUG_REPORT_URL="https://bugs.centos.org/"

CENTOS_MANTISBT_PROJECT="CentOS-7"
CENTOS_MANTISBT_PROJECT_VERSION="7"
REDHAT_SUPPORT_PRODUCT="centos"
REDHAT_SUPPORT_PRODUCT_VERSION="7"

CentOS Linux release 7.9.2009 (Core)
CentOS Linux release 7.9.2009 (Core)
```

1.13.2 Users

Unix 是一个多用户操作系统。因此，即使你在自己的个人机器上使用它，你也是一个拥有 *account* 的用户，有时可能需要输入你的用户名和密码。

如果你在个人机器上，可能是唯一登录的用户。在大学的机器或其他服务器上，通常会有其他用户。以下是一些与他们相关的命令。

whoami 显示你的登录名。*who* 显示其他当前登录的用户

。*finger* otheruser 获取关于另一个用户的信息；你可以在这里指定用户的登录名，或者他们的真实姓名，或者系统已知的其他识别信息。

1.13.3 组和所有权

在章节 1.2.3 中你已经看到有一个 “ 组 ” 的权限类别。这允许你向你的密切合作者开放文件，同时保护文件不被外界访问。

当你的账户被创建时，系统管理员会将你分配到一个或多个组中。（如果你自己管理机器，你会在某个默认组中；继续阅读了解如何将自己添加到更多组。）

命令 *groups* 告诉你你所在的所有组， `ls -l` 告诉你文件属于哪个组。类似于 `chmod`，你可以使用 *chgrp* 来更改文件所属的组，以便与同组的用户共享。

创建新组或将用户添加到组需要系统权限。要创建一个组：

```
sudo groupadd new_group_name
```

将用户添加到组中：

```
sudo usermod -a -G thegroup theuser
```

虽然你可以更改文件的组，至少是在你所属的组之间，但使用 *chown* 更改文件的拥有用户需要 root 权限。参见章节 1.13.4。

1.13.4 超级用户

即使你拥有自己的机器，也有充分的理由尽可能多地使用普通用户账户工作，仅在严格需要时使用 *root privileges*。（root 账户也称为 *superuser*。）如果你拥有 root 权限，你还可以使用 *sudo*（'superuser do'）命令 "成为另一个用户"，并以他们的权限执行操作。

- 以另一个用户身份执行命令：

```
sudo -u otheruser command arguments
```

- 以 root 用户身份执行命令：

```
sudo command arguments
```

- 成为另一个用户：

```
sudo su - otheruser
```

- 成为 *superuser*：

```
sudo su -
```

更改文件所属用户使用 *chown* 完成：

```
sudo chown somefile someuser
sudo chown -R somedir someuser
```

1.14 连接到其他机器：ssh 和 scp

没有人是孤岛，计算机也不是。有时你想用一台计算机，比如你的笔记本，连接到另一台计算机，比如超级计算机。

如果你已经在一台 Unix 计算机上，可以使用 "secure shell" 命令 *ssh* 登录到另一台计算机，这是旧的 "remote shell" 命令 *rsh* 的更安全版本：

```
ssh yourname@othermachine.otheruniversity.edu
```

如果你在两台机器上的用户名相同，`yourname` 可以省略。

要仅仅将文件从一台机器复制到另一台机器，可以使用 "secure copy" *scp*，它是 "remote copy" *rcp* 的安全变体。`scp` 命令在语法上与 `cp` 非常相似，只是源或目标可以带有机器前缀。

要将文件从当前机器复制到另一台机器，输入：

```
scp localfile yourname@othercomputer:otherdirectory
```

其中 yourname 仍然可以省略， otherdirectory 可以是绝对路径，也可以是相对于你的主目录的路径：

```
# absolute path:
scp localfile yourname@othercomputer:/share/
# path relative to your home directory:
scp localfile yourname@othercomputer:mysubdirectory
```

Leaving the destination path empty puts the file in the remote home directory:

```
scp localfile yourname@othercomputer:
```

注意此命令末尾的冒号：如果省略它，您将得到一个名称中带有 “at” 的本地文件。

您也可以从远程机器复制文件。例如，要复制文件并保留名称：

```
scp yourname@othercomputer:otherdirectory/otherfile .
```

1.15 sed 和 awk 工具

除了像 tr 和 cut 这样相当小的实用程序外，Unix 还有一些更强大的工具。在本节中，您将看到两个用于对文本文件逐行转换的工具。当然，本教程仅触及这些工具的表面；更多信息请参见 [1, 8]。

1.15.1 使用 sed 进行流编辑

Unix 有各种工具用于逐行处理文本文件。流编辑器 *sed* 是一个例子。如果你使用过 vi 编辑器，你可能已经习惯了像 s/foo/bar/ 这样的语法来进行修改。使用 sed，你可以在命令行上完成这些操作。例如

```
sed 's/foo/bar/' myfile > mynewfile
```

将对 myfile 的每一行应用替换命令 s/foo/bar/ 。输出显示在屏幕上，因此你应该将其捕获到新文件中；详见第 1.6 节关于输出 重定向 的内容。

- 如果有多个编辑操作，可以用以下方式指定它们

  ```
  sed -e 's/one/two/' -e 's/three/four/'
  ```

- 如果编辑只需在某些特定行上进行，可以通过在编辑前加上匹配字符串来指定。例如

  ```
  sed '/^a/s/b/c/'
  ```

  只对以 a 开头的行应用编辑。（参见 1.3 节关于正则表达式的内容。）
  你也可以将其应用于指定行号：

  ```
  sed '25/s/foo/bar'
  ```

- `a` 和 `i` 命令分别用于 “ 追加 ” 和 “ 插入 ”。它们在接受参数文本的方式上有些奇怪：命令字母后跟一个反斜杠，插入 / 追加的文本在下一行（或多行），由命令的结束引号界定。

```
sed -e '/here/a\
appended text
' -e '/there/i\
inserted text
' -i file
```

- 传统上，`sed` 只能在流中工作，因此输出文件总是必须与输入文件不同。GNU 版本（Linux 系统上的标准版本）有一个标志 `-i`，可以实现 “ 就地 ” 编辑：

```
sed -e 's/ab/cd/' -e 's/ef/gh/' -i thefile
```

1.15.2 awk

awk 工具也对每一行进行操作，但它可以被描述为具有记忆功能。一个 awk 程序由一系列对组成，每对由一个匹配字符串和一个动作组成。最简单的 awk 程序是

```
cat somefile | awk '{ print }'
```

其中匹配字符串被省略，意味着所有行都匹配，动作是打印该行。Awk 将每行分割成由空白字符分隔的字段。awk 的一个常见应用是打印某个字段：

```
awk '{print $2}' file
```

打印每行的第二个字段。

Suppose you want to print all subroutines in a Fortran program; this can be accomplished with

```
awk '/subroutine/ {print}' yourfile.f
```

练习 1.39. 构建一个命令管道，只打印每个子程序头中的子程序名。为此，你首先使用 `sed` 将括号替换为空格，然后使用 `awk` 打印子程序名字段。

Awk 有变量，可以用来记忆信息。例如，你不仅可以打印每行的第二个字段，还可以将它们列成一个列表，稍后再打印：

```
cat myfile | awk 'BEGIN {v="Fields:"} {v=v " " $2} END {print v}'
```

作为变量使用的另一个例子，下面是如何打印位于 `BEGIN` 和 `END` 行之间的所有行：

```
cat myfile | awk '/END/ {p=0} p==1 {print} /BEGIN/ {p=1} '
```

练习 1.40. 使用 `BEGIN` 和 `END` 进行匹配的位置可能看起来很奇怪。重新排列 awk 程序，测试它，并解释你得到的结果。

1.16 复习问题

Exercise 1.41. 设计一个管道，统计系统中有多少用户登录，且用户名以元音字母开头，以辅音字母结尾。

Exercise1.42. 假装你是一名教授，正在编写一个用于作业提交的脚本：如果学生调用此脚本，它会将学生的文件复制到某个标准位置。

```
submit_homework myfile.txt
```

为简单起见，我们通过创建一个目录 `submissions` 和两个不同的文件 `student1.txt` 和 `student2.txt` 来模拟此过程。之后

```
submit_homework student1.txt
submit_homework student2.txt
```

在 `submissions` 目录中应该有两个文件的副本。首先编写一个
简单脚本；如果使用方式错误，它应该给出有用的提示信息。
尝试检测学生是否作弊。探索 *`diff`* 命令，查看提交的
文件是否与已提交的文件相同：遍历所有提交的文件并

1. 首先打印出所有差异。
2. 统计差异数量。3. 测试该数
量是否为零。

现在通过检测作弊学生是否随机插入了一些空格来完善你的测试。
对于更难的测试：尝试检测作弊学生是否插入了换行符。这不能用 *`diff`* 完成，但你可以尝试用 *`tr`* 来去除换行符。

第 2 章

编译器和库

2.1 编程中的文件类型

Purpose. 本节将向您介绍编程过程中遇到的不同类型的文件。

2.1.1 文件类型介绍

您的文件系统中有许多文件，出于编程目的，我们可以大致将它们分为 “ 文本文件 ”，您可以读取，以及 “ 二进制文件 ”，您无法有意义地读取，但对计算机来说是有意义的。

unix 命令 *file* 可以告诉你正在处理的文件类型。

```
$$ file README.txt
README.txt: ASCII text
$$ mkdir mydir
$$ file mydir
mydir: directory
$$ which ls
```

该命令还可以告诉你有关二进制文件的信息。这里的输出因操作系统而异。

```
$$ which ls/bin/ls# on a Mac laptop:
$$ file /bin/ls
/bin/ls: Mach-O 64-bit x86_64 executable
# on a Linux box$$ file /bin/ls
/bin/ls: ELF 64-bit LSB executable, x86-64
```

练习 2.1. 将 `file` 命令应用于不同编程语言的源代码。你能找出 `file` 是如何解决问题的吗?

在图 2.1 中你会找到文件类型的简要总结。我们现在将更详细地讨论它们。

	文本文件
源代码	你编写的程序文本
标题	也是你写的，但不是真正的程序文本。
	二进制文件
对象文件	单个源文件的编译结果
库	多个对象文件捆绑在一起
可执行文件	可以作为命令调用的二进制文件
数据文件	由程序读写

图 2.1：不同类型的文件。

2.1.2 关于 ‘text’ 文件

可读文件有时被称为 *text file*；但这不是一个有严格定义的概念。一个不完美的定义是，textfiles 是 *ascii* 文件，意味着文件中每个字节都使用 ‘7-bit ascii’：每个字节的第一位是零。

这个定义是不完整的，因为现代编程语言通常可以使用 *unicode*，至少在字符串中是这样。（关于 ascii 和 unicode 的教程，请参见 [9] 第 6 章。）

2.1.3 源代码与程序

编程语言有两种类型：

1. 在解释型语言中，你编写人类可读的源代码，并直接执行它：计算机在遇到源代码时逐行翻译。2. 在编译型语言中，你的整个源代码首先被编译成一个二进制程序文件，称为可执行文件，然后你执行该文件。

解释型语言的例子有 *Python*、*Matlab*、*Basic*、*Lisp*。解释型语言有一些优点：通常你可以在交互式环境中编写它们，这允许你非常快速地测试代码。它们还允许你在运行时动态构造代码。然而，所有这些灵活性都是有代价的：如果一行源代码被执行两次，它就会被翻译两次。因此，在本书的语境中，我们将重点关注编译语言，使用 *C* 和 *Fortran* 作为典型示例。

所以现在你已经区分了可读的源代码和不可读但可执行的程序代码。在本教程中，你将了解从前者到后者的翻译过程。执行此翻译的程序被称为编译器。本教程将作为编译器的 “ 用户手册 ”；编译器内部发生的事情属于计算机科学的另一个分支。

2.1.4 二进制文件

二进制文件分为两类：

- executable code,
- data

数据文件可以是任何东西：它们通常是程序的输出，其格式通常特定于该程序，尽管存在一些标准，比如 *hdf5*。如果你写出整数或浮点数据在内存中占用的确切字节，而不是你在屏幕上看到的数字的可读表示形式，你就得到了一个二进制数据文件。

练习 2.2. 为什么程序不以可读形式将结果写入文件?

拓展。如何在 C 和 Fortran 中读写二进制文件？使用函数 *hexdump* 来理解二进制文件。你能用 Fortran 生成文件，并用 C 读取它吗？（答案是：可以，但不是很简单。）这告诉你关于二进制数据什么?

```c
// binary_write.c
  FILE *binfile;
  binfile = fopen("binarydata.out","wb");
  for (int i=0; i<10; i++)
    fwrite(&i,sizeof(int),1,binfile);
  fclose(binfile);
```

```c
// binary_read.c
  binfile = fopen("binarydata.out","rb");
  for (int i=0; i<10; i++) {
    int ival;
    fread(&ival,sizeof(int),1,binfile);
    printf("%d ",ival);
  }
  printf("\n");
```

```
[linking:31] make binary
clang -o binary_write_c binary_write.c./binary_write_c
clang -o binary_read_c binary_read.c./binary_read_c0 1
2 3 4 5 6 7 8 9[linking:32] hexdump binarydata.out
0000000 00 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00
0000010 04 00 00 00 05 00 00 00 06 00 00 00 07 00 00 00
0000020 08 00 00 00 09 00 00 00 0000028
```

Fortran 的工作方式不同：每个 *record*，即每个 `Write` 语句的输出，前后都有记录长度（以字节为单位）。

```
[linking:68] make xbinary
gfortran -o binary_write_f binary_write.F90
./binary_write_f
hexdump binarydata.out
0000000 04 00 00 00 00 00 00 00 04 \
                00 00 00 04 00 00 00
0000010 01 00 00 00 04 00 00 00     04 \
                00 00 00 02 00 00 00
0000020 04 00 00 00 04 00 00 00 03 \
                00 00 00 04 00 00 00
0000030 04 00 00 00 04 00 00 00 04 \
                00 00 00 04 00 00 00
```

```
0000040 05 00 00 00 04 00 00 00 04 \
                  00 00 00 06 00 00 00
0000050 04 00 00 00 04 00 00 00 07 \
                  00 00 00 04 00 00 00
0000060 04 00 00 00 08 00 00 00 04 \
                  00 00 00 04 00 00 00
0000070 09 00 00 00 04 00 00 00
```

```
// binary_write.F90
  Open(Unit=13,File="binarydata.out",Form="
    unformatted")
  do i=0,9
     write(13) i
  end do
  Close(Unit=13)
```

在本教程中，您主要关注可执行二进制文件。我们将区分：

- 程序文件，它们本身是可执行的；
- 对象文件，它们像是程序的一部分；以及
- 库文件，它们组合了对象文件，但不可执行。

对象文件的产生源于您的源代码通常分布在多个源文件中，这些文件可以单独编译。这样，对象文件是可执行文件的一部分：它本身什么也不做，但可以与其他对象文件组合形成可执行文件。

如果您有一组需要用于多个程序的对象文件，通常制作一个库是个好主意：一组可以用来形成可执行文件的对象文件。通常，库由专家编写，包含用于特定目的（如线性代数操作）的代码。库的重要性足以使其成为商业产品，如果您需要某个特定用途的专家代码，可以购买这些库。

您现在将学习这些类型的文件是如何创建和使用的。

2.2 Simple compilation

目的。在本节中，您将了解可执行文件和对象文件。

2.2.1 编译器

将源代码转换为程序的主要工具是编译器。编译器针对特定语言：您为 C 语言使用的编译器与 Fortran 语言的不同。您也可以有两个针对同一语言但来自不同 “ 供应商 ” 的编译器。例如，虽然许多人使用开源的 *gcc* 或 *clang* 编译器系列，像 *Intel* 和 *IBM* 这样的公司则提供可能在其处理器上生成更高效代码的编译器。

2.2.2 编译单个文件

让我们从一个将所有源代码放在一个文件中的简单程序开始。

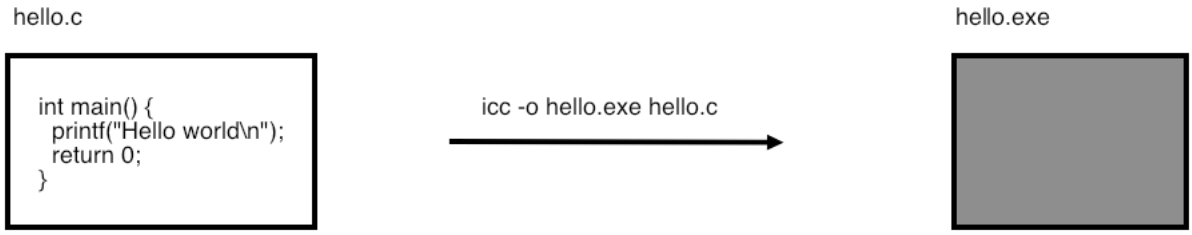


图 2.2：编译单个源文件。

```
#include <stdlib.h>
#include <stdio.h>

int main() {
  printf("hello world\n");
  return 0;
}
```

使用你喜欢的编译器编译此程序；本教程中我们将使用 `gcc`，但可根据需要替换为你自己的。

TACC 注释。在 TACC 集群上，推荐使用 Intel 编译器 `icc`。
编译后，会生成一个文件 `a.out`，它是可执行文件。`%% gcc hello.c`
`%% ./a.outhello world` 你可以使用 `-o` 选项获得一个更合理的程序
名：

```
%% gcc -o helloprog hello.c
%% ./helloprog
hello world
```

此过程如图 2.2 所示。

2.2.3 编译：细节解析

即使是上一节中简单的编译，也省略了一些细节。这里概述了编译过程的步骤，这不仅仅是调用编译器。例如，你的程序很可能包含 `#include` 语句，这些语句在一个单独的阶段进行处理。

让我们来看一个简单的 “hello world” 程序。不是一次性将你的源代码转换成可执行文件，而是发生以下步骤。

1. 预处理器生成一个文件 `hello.i`，其中所有的 include 都被内联，其他宏也被展开。这个文件通常会立即被删除：你需要提供一个编译器标志才能保留它。

2. 生成程序的汇编清单 `hello.s` ；同样，使用后会立即删除。这是一种 “ 可读的机器语言 ”：

```
    .arch armv8-a.text.cstring
.align  3lC0:.ascii "hello world\0"
.text.align  2.globl _main_main:
LFB10:stp     x29, x30, [sp, -16]!
```

3. 编译的主要实质性结果是 `hello.o`，即包含实际机器语言的对象文件。我们将在下面详细介绍。对象文件不可直接读取，但稍后你会看到 *nm* 工具，它可以提供一些信息。

4. 最后，*linker* 将对象文件和系统库连接成一个自包含的可执行文件或库文件。

2.2.4 多文件：编译和链接

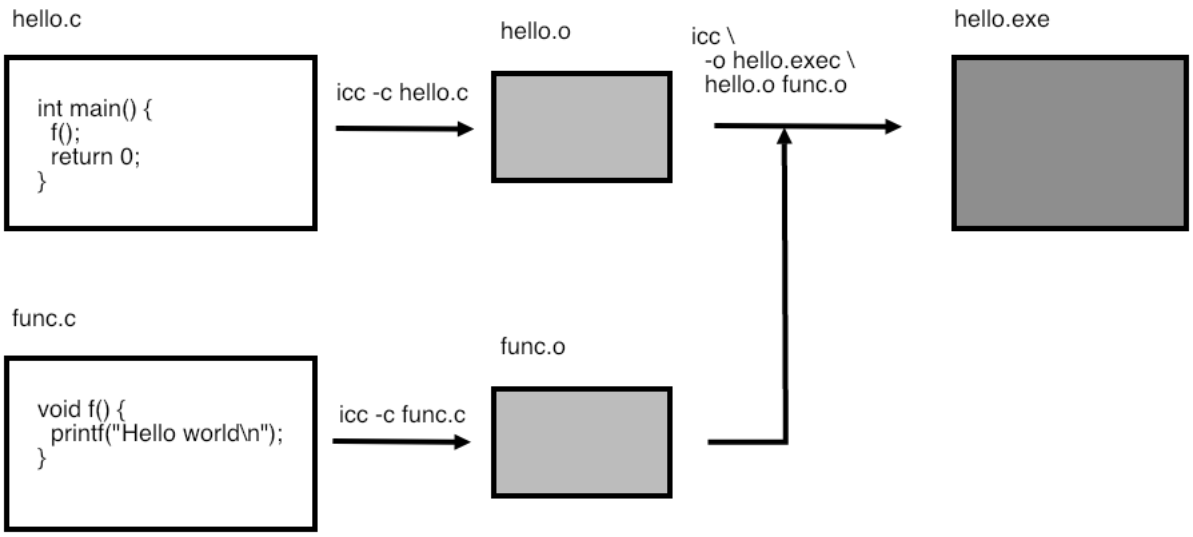


图 2.3：从多个源文件编译程序 e files.

作为链接的一个例子，我们考虑一个源代码包含在多个文件中的程序。

主程序： `fooprog.c`

```
// fooprog.c
extern void bar(char*);
```

```
int main() {
  bar("hello world\n");
```

```
  return 0;
}
```

Subprogram: `foosub.c`

```
// foosub.c
void bar(char *s) {
  printf("%s",s);
  return;
}
```

和之前一样，你可以用一个命令来编译程序。

Output[
code/co mpile] makeoneprogram:

```
clang -o oneprogram fooprog.c foosub.c
./oneprogram
hello world
```

但是，你也可以分步骤进行，分别编译每个文件，然后再将它们链接在一起。这在图 2.3 中有所说明。

Output[
code/co 编译] 使得单独编译：

```
clang -g -O2 -o oneprogram fooprog.o foosub.o
./oneprogram
hello world
```

`-c` 选项告诉编译器编译源文件，生成一个对象文件。第三个命令则作为链接器，将对象文件连接成一个可执行文件。

练习 2.3.Exerc

用于单独编译。结构：

Main program: `fooprog.c`

```
#include <stdlib.h>
#include <stdio.h>

extern void bar(char*);

int main() {
  bar("hello world\n");
  return 0;
}
```

Subprogram: `foosub.c`

```
#include <stdlib.h>
#include <stdio.h>

void bar(char *s) {
  printf("%s",s);
  return;
}
```

Add a second subroutine in a second file.

- 一次性编译：

```
icc -o program fooprog.c foosub.c
```

- 分步骤编译：

```
icc -c fooprog.c
icc -c foosub.c
```

```
icc -o program fooprog.o foosub.o
```

每次会生成哪些文件？你能写一个脚本来自
动化这个过程吗？

2.2.5 路径

如果你的程序使用库，甚至是你自己制作的库，你需要告诉

1. 编译器在哪里找到头文件，以及 2. 链接器在哪里
找到库文件。

(C++ 支持 “ 仅头文件 ” 库，因此第二步并不总是需要。) 这些位置通过命令行选项指定：

```
gcc -c mysource.cpp -I${SOMELIB_INC_DIR}
gcc -o myprogram mysource.o -L${SOMELIB_LIB_DIR} -lsomelib
```

(当然，你不会每次都在命令行上列出这些选项，而是会把它们放在 makefile 中，或者使用 CMake 来生成这样的命令行。) 编译行有 `-I` 选项用于 ‘include’，指定库头文件的位置。你可以指定多个 include 选项。

链接行有 `-L` 选项用于 ‘library’，指定实际库文件的位置，还有 `-l` 选项用于库名。你可以指定多个库目录。 `-l` 选项的解释如下： `-l somelib` 使链接器搜索文件

```
libsomelib.a
libsomelib.so
libsomelib.dylib
```

2.2.6 查看二进制文件： `nm`

二进制文件的大部分内容由你用 C 或 Fortran 编写的相同指令组成，只是以机器语言形式存在，这种语言更难理解。幸运的是，你通常不需要经常查看机器语言。通常你对对象文件感兴趣的只是知道它定义了哪些函数，以及使用了哪些函数。

为此，我们使用 *nm* 命令。

E每个对象文件定义了一些例程名称，并使用了一些在其中未定义的其他例程，但这些例程将在其他对象文件或系统库中定义。使用 *nm* 命令来显示这个：

```
[c:264] nm foosub.o
0000000000000000 T _bar
                 U _printf
```

L带有 `T` 的行表示定义的例程；带有 `U` 的行表示使用但未在此文件中定义的例程。
i在这种情况下， `printf` 是一个将在链接阶段提供的系统例程。

2.2.7 C++ 中的名称解混淆

在 C++ 中，由于名称混淆，函数名看起来会有些奇怪。例如函数

```
void bar( string s ) {
  cout << s;
}
```

gives:

```
[] nm foosub.o
0000000000000000 T __Z3barNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
                 U __ZNSt8ios_base4InitC1Ev
                 U __ZNSt8ios_base4InitD1Ev
                 U
    __ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_l
                 U __ZSt4cout
```

你可以在其中辨认出 `bar` 函数。添加一个选项 `nm -C` 来获取解混淆的名称：

```
[scientific-computing-private/code/compilecxx 1354] nm -C !$[] nm -C foosub.o
0000000000000000 T bar(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator
<char> >)U std::ios_base::Init::Init()U std::ios_base::Init::~Init()U std::basic_ostream<char,
 std::char_traits<char> >& std::__ostream_insert<char,
 std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, charconst*,
 long)U std::cout
```

另一种可能是通过管道传输 `nm` 输出 `c++filt`：

```
nm foosub.o | c++file
```

当你的对象文件留下某些引用未定义时，你可能还想在链接器错误中对名称进行解混淆。为此，在链接命令行中添加一个选项 `-Wl,-demangle`：

```
Compile: g++ -g -O2 -std=c++17                -c undefined.cxx
Link line: g++ -Wl,-demangle      -o undefined undefined.o
Undefined symbols for architecture arm64:
  "f(int, int)", referenced from:
      _main in undefined.o
ld: symbol(s) not found for architecture arm64
collect2: error: ld returned 1 exit status
make[2]: *** [undefined] Error 1
```

可以通过程序获取混淆名称：

```
#include <typeinfo>
cout << typeid( &Myclass::method ).name() << '\n'
```

解混淆也有一个应用程序接口（API）：

```
#include <cxxabi.h>
char* abi::__cxa_demangle
    ( const char* mangled, char* output, size_t* length,int status );
```

Remark9 还有其他具有与 *nm* 类似功能的工具，例如 *otool*、 *objdump*、 *readelf*。

Remark10 有时你会遇到剥离的二进制文件，而 *nm* 会报告 No symbols。在这种情况下， nm -D 可能有帮助，它显示“动态符号”。

2.2.8 编译选项和优化

上面你已经看到了一些编译选项：

- 指定 -c 告诉编译器只进行编译，而不进行链接阶段；在分开编译的情况下会这样做。
- 选项 -o 允许你指定输出文件的名称；如果不指定，默认的可执行文件名是 a.out。

还有许多其他选项，其中一些是事实上的标准，另一些则是特定于某些编译器的。

2.2.8.1 符号表包含

该 -g 选项告诉编译器在二进制文件中包含符号表。这允许你使用交互式调试器（第 11 节），因为它将机器指令与代码行关联，将机器地址与变量名关联。

2.2.8.2 优化级别

编译器可以对您的代码应用各种级别的优化。典型的优化级别指定为 -O0 ‘minus-oh-zero’， -O1，-O2， -O3。较高级别通常会带来更快的执行速度，因为编译器会对您的代码进行越来越复杂的分析。

以下是一组相当标准的选项：

```
icc -g -O2 -c myfile.c
```

作为示例，让我们看看 *Given'srotations*：

```
// rotate.c
void rotate(double *x,double *y,double alpha) {
double x0 = *x, y0 = *y;
*x = cos(alpha) * x0 - sin(alpha) * y0;
*y = sin(alpha) * x0 + cos(alpha) * y0;return;
}
```

以优化级别 0、 1、 2、 3 运行，我们得到：

```
Done after 8.649492e-02
Done after 2.650118e-02
Done after 5.869865e-04
Done after 6.787777e-04
```

练习 2.4. 从零级到一级，我们获得了（在上述示例中；一般这取决于编译器）从 2× 到 3× 的提升。你能找到明显的两倍因子吗？使用你的编译器的优化报告功能，看看还应用了哪些其他优化。其中一个是基准设计中的一个很好的教训！

许多编译器可以生成它们执行的优化报告。

编译器报告选项	
clang	`-Rpass=.*`
gcc	`-fopt-info`
intel	`-qopt-report`

通常，优化不会改变代码的语义。（这很合理，不是吗？）然而，在更高级别，通常是 level3，编译器可以自由进行一些根据语言标准不合法的转换，但在大多数情况下仍能得到正确的结果。例如，C 语言规定算术运算按从左到右的顺序求值。重新排列算术表达式通常是安全的，但并非总是如此。应用更高级别的优化时请小心！

2.2.8.3 优化报告

编译器可以报告某些优化。

Intel 编译器有一个参数 `-qoptreport=[0-5]`。

- GCC 编译器有多种命令行选项：

```
-fopt-info -fopt-info-options -fopt-info-options=filename
```

See: https://stackoverflow.com/a/65017597/2044454.

2.3 库

Purpose. 在本节中，您将学习如何创建库。

如果你已经编写了一些子程序，并且想与其他人共享它们（可能通过出售），那么单独交付各个对象文件会很不方便。相反，解决方案是将它们合并成一个库。本节向你展示基本的 Unix 机制。你通常会在 Makefile 中使用这些；如果你使用 CMake，这些都会为你自动完成；详见第 4 章。

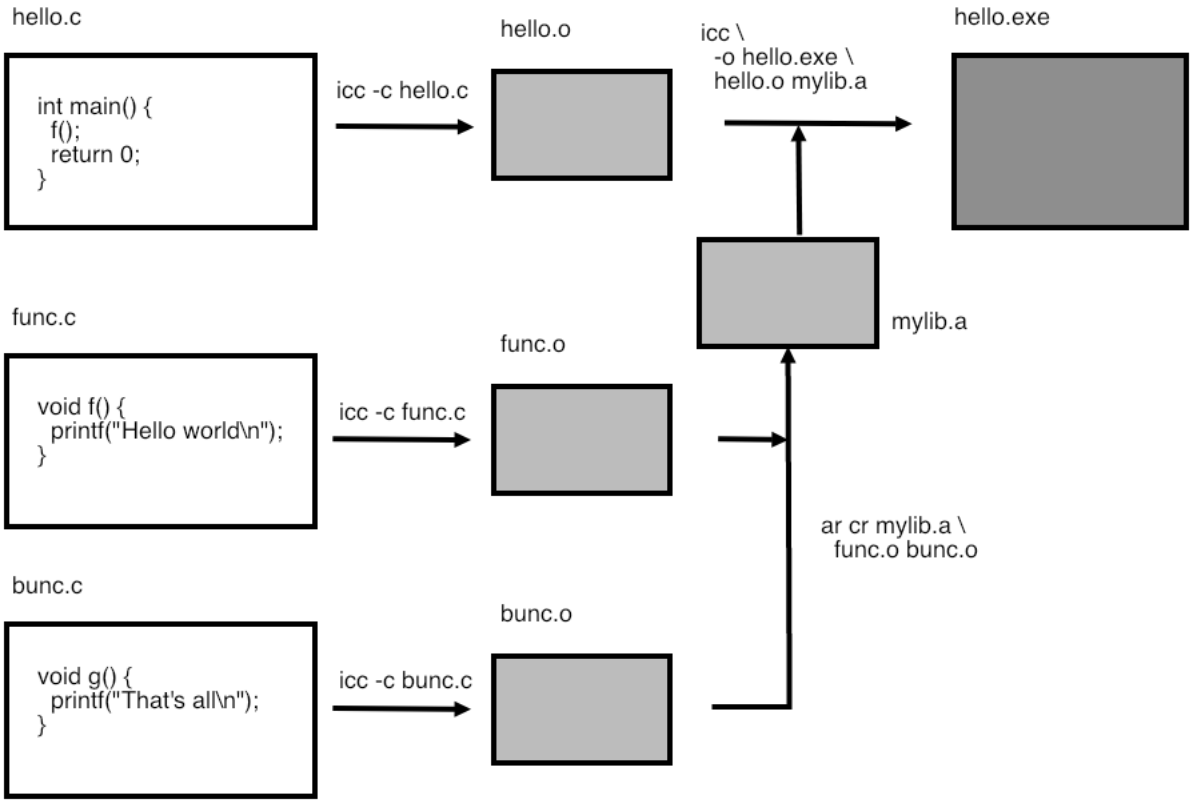


图 2.4：编译单个源文件。

2.3.1 静态库

首先我们来看静态库，使用的是 *archiveutility* `ar`。静态库会被链接到你的可执行文件中，成为其一部分。这有一个优点，就是你可以将这样的可执行文件交给别人，它能立即运行。另一方面，这可能导致可执行文件体积较大；接下来你将学习共享库，它们不会有这个问题。

使用库来构建程序的过程如图 2.4 所示。

创建一个目录来存放你的库文件，并在那里创建库文件。库可以通过显式指定其名称，或者通过指定库路径来链接到你的可执行文件中：

Output
[code/compilecxx] staticprogram:

```
==== Use of static library ====

---- Build library:
for o in foosub.o ; do ar cr libs/libfoo.a ${o} ; done
---- Linking:
g++ -o staticprogram fooprog.o -Llibs -lfoo
---- Program:
-rwxr-xr-x  1 eijkhout  staff  52042 Oct 21 10:46 staticprogram

---- running:
hello world
---- done
```

nm 命令告诉你库中有什么，就像它对对象文件所做的那样，但现在它还告诉你库中包含哪些对象文件：

我们以 C 语言为例展示：

Code:

```
==== Making static library ====
for o in foosub.o ; do ar cr libs/libfoo.a ${o} ; done
nm  libs/libfoo.a
```

Output
[code/compilec] staticlib:

```
foosub.o:
0000000000000000 T bar
0000000000000000 N .debug_info_seg
                 U printf
```

For C++ 我们在图 2.5 中展示了输出，注意到 `-C` 用于名称解混淆的标志。

2.3.2 共享库

虽然使用起来稍微复杂一些，共享库有几个优点。例如，由于它们不是链接到可执行文件中，而仅在运行时加载，因此它们生成的可执行文件（小得多）。它们不是通过 *ar* 创建的，而是通过编译器创建的。例如：

Output
[code/compile] makedynamiclib:

```
%%%%
Demonstration: making shared library
%%%%
clang -o libs/libfoo.so -shared foosub.o
```

你可以再次使用 *nm*：

```
%% nm ../lib/libfoo.so
../lib/libfoo.so(single module):
00000fc4 t __dyld_func_lookup
```

```
00000000 t __mh_dylib_header
00000fd2 T _bar
         U _printf
00001000 d dyld__mach_header
00000fb0 t dyld_stub_binding_helper
```

2.3.3 寻找库

共享库实际上并未链接到可执行文件中；相反，可执行文件需要在执行时知道库的位置。实现这一点的一种方法是使用 *LD_LIBRARY_PATH*；见图 2.6

另一种解决方案是将路径包含在可执行文件中：

```
%% gcc -o foo fooprog.o -L../lib -Wl,-rpath,`pwd`/../lib -lfoo
%% ./foohello world
```

链接行现在包含了两次库路径：

1. 使用 `-L` 指令一次，以便链接器可以解析所有引用，2. 使用链接器指令 `-Wl,-rpath,`pwd`/../lib` 一次，将路径存储到可执行文件中，以便在运行时找到。

Remark11 你也可能会遇到带有 `rpath=` 的语法：

```
gcc -o foo fooprog.o -L../lib -Wl,-rpath=`pwd`/../lib -lfoo
```

但要注意那是 *GNU* 扩展。

Remark 12 在 Apple OS Ventura 上出于安全原因不再支持使用 *LD_LIBRARY_PATH*，因此使用 `rpath` 是唯一的选择。

使用命令 *ldd* 获取有关你的可执行文件使用了哪些共享库的信息。（在 Mac OS X 上，改用 `otool -L`。）

2.3.4 标准库

某些功能，如 I/O 或文件和进程管理，在所有编程语言中都会使用，但更属于操作系统而非语言本身。这就是为什么这些功能被打包在一个标准库中，称为 *libstdc* 或 *libstdc++*。关于该库内容的错误信息通常会提到带有版本号的 *GLIBC* 或 *GLIBCXX*。

```
[] /sbin/ldconfig -p | grep stdc++
        libstdc++.so.6 (libc6,x86-64) => /lib64/libstdc++.so.6
```

```
        libstdc++.so.5 (libc6,x86-64) => /lib64/libstdc++.so.5
[] strings /lib64/libstdc++.so.6 | grep LIBCXX
GLIBCXX_3.4
GLIBCXX_3.4.1
   [...]
GLIBCXX_3.4.19
```

有时请求的功能版本比系统中存在的版本要新。

代码：

```
==== Making static library ====
for o in foosub.o ; do ar cr libs/libfoo.a ${o} ; done
nm -C libs/libfoo.a
```

Output
[code/compilecxx] staticlib:

```
foosub.o:
                 U __cxa_atexit
0000000000000000 N .debug_info_seg
                 U __dso_handle
                 U
    __gxx_personality_v0
0000000000000010 t __sti__$E
0000000000000000 T bar(std::
    __cxx11::basic_string<char,
    std::char_traits<char>, std::
    allocator<char> >)
0000000000000000 b
    _INTERNALaee936d8::std::
    __ioinit
0000000000000000 W std::__cxx11::
    basic_string<char, std::
    char_traits<char>, std::
    allocator<char> >::data()
    const
0000000000000000 W std::__cxx11::
    basic_string<char, std::
    char_traits<char>, std::
    allocator<char> >::size()
    const
                 U std::ios_base::
    Init::Init()
                 U std::ios_base::
    Init::~Init()
                 U std::cout
                 U std::
    basic_ostream<char, std::
    char_traits<char> >& std::
    operator<< <char, std::
    char_traits<char>, std::
    allocator<char> >(std::
    basic_ostream<char, std::
    char_traits<char> >&, std::
    __cxx11::basic_string<char,
    std::char_traits<char>, std::
    allocator<char> > const&)
```

图 2.5：C++ 库的 nm 输出

代码 / 编译动态程序 :

```
Linking to shared library

clang -o libs/libfoo.so -shared foosub.o
clang -o dynamicprogram fooprog.o -Llibs -lfoo
-rwxr-xr-x  1 eijkhout  staff  49720 Nov 28 12:00 dynamicprogram

.. note the size of the program

-rwxr-xr-x  1 eijkhout  staff  49720 Nov 28 12:00 dynamicprogram

.. note unresolved link to a library

otool -L dynamicprogram | grep libfoo
        libs/libfoo.so (compatibility version 0.0.0, current version 0.0.0)

.. running by itself:

clang -o libs/libfoo.so -shared foosub.o
clang -o dynamicprogram fooprog.o -Llibs -lfoo
-rwxr-xr-x  1 eijkhout  staff  49720 Nov 28 12:00 dynamicprogram
hello world

.. note resolved with LD_LIBRARY_PATH

LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:./libs otool -L dynamicprogram | grep
    libfoo
        libs/libfoo.so (compatibility version 0.0.0, current version 0.0.0)

.. running with updated library path:

clang -o libs/libfoo.so -shared foosub.o
clang -o dynamicprogram fooprog.o -Llibs -lfoo
-rwxr-xr-x  1 eijkhout  staff  49720 Nov 28 12:00 dynamicprogram
LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:./libs dynamicprogram
hello world
```

图 2.6: 制作和使用动态库

第 3 章

使用 Make 管理项目

Make 工具帮助你管理项目的构建：它的主要任务是方便只重新编译或重构多文件项目中需要更新的部分。这可以节省大量时间，因为它可以用单个文件的编译替代耗时数分钟的完整安装。

Make 是一个历史悠久的 Unix 工具，传统上存在行为略有不同的变体，例如在各种 Unix 版本如 HP-UX 、 AUX 、 IRIX 上。如今，无论平台如何，建议使用 GNU 版本的 Make，它具有一些非常强大的扩展；该版本在所有 Unix 平台上均可用（在 Linux 上它是唯一可用的变体），并且是一个 事实上的 标准。手册可在 http://www.gnu.org/software/make/manual/make.html 获取，或者你也可以阅读书籍 [14]。

本教程中的示例将针对 C 和 Fortran 语言，但 *Make* 可以处理任何语言，实际上也可以处理像 TEX 这样并不是真正语言的东西；参见第 3.7 节。

3.1 一个简单的例子

目的。在本节中，您将看到一个简单的例子，仅仅是为了展示 *Make* 的用法。

本节的文件可以在代码仓库中找到。

3.1.1 C++

Make the following files:

`foo.cxx`

```
#include <iostream>
using std::cout;

#include "bar.h"

int main()
{
  int a=2;
```

```
  cout << bar(a) << '\n';
  return 0;
}
```

bar.cxx

```
#include "bar.h"
int bar(int a)
{
  int b=10;
  return b*a;
}
```

bar.h

```
int bar(int);
```

and a makefile:

Makefile

```
fooprog : foo.o bar.o
icpc -o fooprog foo.o bar.o
foo.o : foo.cxx
icpc -c foo.cxx
bar.o : bar.cxx
icpc -c bar.cxx
clean :
rm -f *.o fooprog
```

The makefile has a number of rules like

```
foo.o : foo.cxx
<TAB>icpc -c foo.cxx
```

which have the general form

```
target : prerequisite(s)
<TAB>rule(s)
```

其中规则行缩进了 TAB 个字符。

一个规则，比如上面所示，说明一个 “ 目标 ” 文件 foo.o 是由一个 “ 先决条件 ” foo.cxx 制作的，即通过 e
执行命令 icpc -c foo.cxx。（这里我们使用的是 *Intel C++ compiler* icpc ；你的系统
c可能有不同的编译器，例如 *clang*++ 或 *g*++。）

规则的精确定义是：

- 如果目标 foo.o 不存在或比先决条件 foo.cxx 旧，
- 则执行规则的命令部分： icpc -c foo.cxx
- 如果先决条件本身是另一个规则的目标，则先执行该规则。

练习。调用 make。

预期结果。上述规则被应用：`make` 无参数时尝试构建第一个目标，`fooprog`。为了构建它，需要前置条件 `foo.o` 和 `bar.o`，但它们不存在。然而，有生成它们的规则，`make` 递归调用这些规则。因此你会看到两个编译过程，分别是 `foo.o` 和 `bar.o`，以及一个针对 `fooprog` 的链接命令。

注意事项。makefile 或文件名中的拼写错误可能导致各种错误。特别是，确保你在规则行中使用的是制表符而不是空格。不幸的是，调试 makefile 并不简单。*Make* 的错误信息通常会给出检测到错误的 makefile 行号。

练习。执行 `make clean`，然后依次执行 `mv foo.cxx boo.cxx` 和 `make`。解释错误信息。恢复原始文件名。

预期结果。*Make* 会抱怨没有规则来构建 `foo.cxx`。当 `foo.cxx` 是构建 `foo.o` 的前置条件且未找到时，产生了此错误。*Make* 随后尝试寻找构建它的规则，但不存在创建 `.cxx` 文件的规则。

现在给函数 `bar` 添加第二个参数。这将要求你编辑所有的 `bar.cxx`、`bar.h` 和 `foo.cxx`，但假设我们忘记编辑后两个，只编辑了 `bar.cxx`。然而，这也要求你编辑 `foo.c`，但我们暂时“忘记”这么做。我们将看到 *Make* 如何帮助你发现由此产生的错误。

练习。调用 `make` 重新编译你的程序。它重新编译了 `foo.c` 吗?

预期结果。你会看到包含“错误”的头文件不会导致错误，因为 C++ 具有多态性。在 C 中这肯定会报错。错误只会在链接阶段出现，因为存在未解析的引用。

练习。更新头文件，然后再次调用 `make`。发生了什么，你本希望发生什么?

预期结果。只进行了链接阶段，并且给出了相同的关于未解析引用的错误。你是否希望主程序会被重新编译？注意事项。

解决这个问题的方法是在 makefile 中将头文件与源文件绑定。

在 makefile 中，更改该行

```
foo.o : foo.cxx
```

to

```
foo.o : foo.cxx bar.h
```

它将 `bar.h` 作为 `foo.o` 的先决条件。这意味着，在 `foo.o` 已经存在的情况下，*Make* 会检查 `foo.o` 是否不比其任何先决条件旧。由于 `bar.h` 已被编辑，它比 `foo.o` 新，因此需要重新构建 `foo.o`。

Remark13 如上所述，在 *C++* 中，由于多态性，这种机制捕获的错误比在 *C* 中更少。你可能会想是否可以自动生成头文件。当然，使用合适的 *shell* 脚本这是可能的，但像 `Make`（或 `CMake`）这样的工具并没有内置此功能。

3.1.2 C

Make the following files:

`foo.c`

```
#include "bar.h"
int c=3;
int d=4;
int main()
{
  int a=2;
  return(bar(a*c*d));
}
```

`bar.c`

```
#include "bar.h"
int bar(int a)
{
  int b=10;
  return(b*a);
}
```

`bar.h`

```
extern int bar(int);
```

和一个 makefile:

`Makefile`

```
fooprog : foo.o bar.o
cc -o fooprog foo.o bar.o
foo.o : foo.c
cc -c foo.c
bar.o : bar.c
cc -c bar.c
clean :
rm -f *.o fooprog
```

makefile 有许多规则，如

foo.o : foo.c<TAB>cc -c foo.c，其一

般形式为 *target : prerequisite(s)*

<TAB>rule(s)

其中规则行缩进了 TAB 个字符。

一个规则，如上所述，表示一个 “ 目标 ” 文件 `foo.o` 是由一个 “ 前置条件 ” `foo.c` 生成的，即通过执行命令 `cc -c foo.c`。该规则的精确定义是:

- 如果目标 `foo.o` 不存在或比先决条件 `foo.c` 旧,

- 然后执行规则的命令部分： `cc -c foo.c`
- 如果前提本身是另一个规则的目标，则先执行该规则。

Exercise. 调用 `make`。

> 预期结果。上述规则被应用： `make` 无参数时尝试构建第一个目标， `fooprog`。为了构建它，需要前提条件 `foo.o` 和 `bar.o`，但它们不存在。然而，有构建它们的规则， `make` 递归调用这些规则。因此你会看到两次编译，分别针对 `foo.o` 和 `bar.o`，以及一次针对 `fooprog` 的链接命令。
>
> 注意事项。 makefile 或文件名中的拼写错误可能导致各种错误。特别是，确保规则行使用制表符而不是空格。不幸的是，调试 makefile 并不简单。*Make* 的错误信息通常会给出检测到错误的 makefile 行号。

练习。 执行 `make clean`，然后是 `mv foo.c boo.c` 和再次执行 `make` 。解释错误信息。恢复原始文件名。

> 预期结果。*Make* 会报错说没有规则来制作 `foo.c`。这个错误是因为 `foo.c` 是制作 `foo.o` 的前置条件，但发现它不存在。*Make* 随后尝试寻找制作它的规则，但没有创建 `.c` 文件的规则。

现在给函数 `bar` 添加第二个参数。这需要你编辑 `bar.c` 和 `bar.h`：请继续进行这些编辑。然而，这也需要你编辑 `foo.c`，但我们暂时 ‘忘记’ 去做。我们将看到 *Make* 如何帮助你发现由此产生的错误。

练习。 调用 `make` 重新编译你的程序。它重新编译了 `foo.c` 吗?

> 预期结果。即使从概念上讲 `foo.c` 需要重新编译，因为它使用了 `bar` 函数，*Make* 并没有这样做，因为 makefile 中没有强制它的规则。

在 makefile 中，更改该行

```
foo.o : foo.c
```

to

```
foo.o : foo.c bar.h
```

它将 `bar.h` 添加为 `foo.o` 的先决条件。这意味着，在 `foo.o` 已经存在的情况下，*Make* 会检查 `foo.o` 是否不比其任何先决条件旧。由于 `bar.h` 已被编辑，它比 `foo.o` 新，因此需要重新构建 `foo.o` 。

Exercise. 确认新的 makefile 确实会在 `bar.h` 发生变化时重新编译 `foo.o` 。这次编译现在会报错，因为你 “忘记” 编辑对 `bar` 函数的使用。

3.1.3 Fortran

制作以下文件：`foomain.F`

```
call func(1,2)end program
```

`foomod.F`

```
contains
subroutine func(a,b)integer a,b
print *,a,b,cend subroutine func
end module
```

以及一个 makefile：`Makefile`

```
fooprog : foomain.o foomod.o
gfortran -o fooprog foo.o foomod.o
foomain.o : foomain.F
gfortran -c foomain.Ffoomod.o : foomod.F
gfortran -c foomod.Fclean :
rm -f *.o fooprog
```

如果你调用 `make`，则执行 makefile 中的第一个规则。请执行此操作，并解释发生了什么。

Exercise. Call `make`.

> 预期结果。上述规则被应用： `make` 无参数时尝试构建第一个目标 `foomain`。为了构建它，需要前置条件 `foomain.o` 和 `foomod.o`，但它们不存在。然而，有制作它们的规则，`make` 递归调用这些规则。因此你会看到两次编译，分别是 `foomain.o` 和 `foomod.o`，以及一个针对 `fooprog` 的链接命令。注意事项。makefile 或文件名中的拼写错误可能导致各种错误。不幸的是，调试 makefile 并不简单。你只能理解错误并进行修正。

练习。 执行 `make clean`，然后是 `mv foomod.c boomod.c` 和再次执行 `make`。解释错误信息。恢复原始文件名。

> 预期结果。执行 make 会报错，提示没有规则来生成 `foomod.c`。该错误是因为 `foomod.c` 是 `foomod.o` 的前置条件，但未找到 `foomod.c`。*make* 随后尝试寻找生成它的规则，但没有生成 `.F` 文件的规则。

现在在 `foomod.F` 中的 `func` 添加一个额外的参数并重新编译 .

Exercise. Call `make` to recompile your program. Did it recompile `foomain.F`?

预期结果。尽管从概念上讲 `foomain.F` 需要重新编译，*Make* 并未这样做，因为 makefile 中没有强制执行的规则。

更改该行

```
foomain.o : foomain.F
```

to

```
foomain.o : foomain.F foomod.o
```

该行将 `foomod.o` 添加为 `foomain.o` 的前置条件。这意味着，在 `foomain.o` 已经存在的情况下，*Make* 会检查 `foomain.o` 是否不比其任何前置条件旧。递归地，*Make* 随后会检查 `foomode.o` 是否需要更新，事实确实如此。重新编译 `foomode.F` 后，`foomode.o` 比 `foomain.o` 年轻，因此 `foomain.o` 将被重建。

练习。确认修正后的 makefile 确实导致 `foomain.F` 被重新编译。

3.2 一些一般性说明

3.2.1 规则解释

解释规则的最好方法可能是：

- 如果任何前提条件发生了变化，
- 那么目标需要重新制作，
- 这通过执行规则的命令来完成；
- 检查前提条件需要递归地应用 make：– 如果前提条件不存在，找到一个规则来创建它；– 如果前提条件已经存在，检查适用的规则以确定是否需要重新制作。

3.2.2 Make 调用

如果你调用 `make` 而不带任何参数，将会执行 makefile 中的第一个规则。你也可以通过显式调用其他规则来执行它们，例如 `make foo.o` 用于编译单个文件。

3.2.3 关于 makefile

make 文件需要被命名为 `makefile` 或 `Makefile`；在同一目录下同时存在这两个名字的文件并不是一个好主意。如果你想让 *Make* 使用不同的文件作为 make 文件，可以使用语法 `make -fMy_Makefile`。

3.3 变量和模板规则

目的。在本节中，你将学习 *Make* 中各种节省工作量的机制，例如变量的使用和模板规则。

3.3.1 Makefile 变量

在你的 makefile 中引入变量是很方便的。例如，不必每次都明确写出编译器，可以在 makefile 中引入一个变量：

```
CC = gcc
FC = gfortran
```

并在编译行中使用 `${CC}` 或 `${FC}`：

```
foo.o : foo.c
        ${CC} -c foo.c
foomain.o : foomain.F
        ${FC} -c foomain.F
```

练习。按指示编辑你的 makefile。先做 `make clean`，然后做 `make foo` (C) 或 `make fooprog` (Fortran)。

> 预期结果。 你应该看到与之前完全相同的编译和链接行。注意事项。 与 Shell 中大括号是可选的不同，makefile 中的变量名必须用大括号或圆括号括起来。试验一下如果忘记给变量名加括号会发生什么。

使用变量的一个优点是你现在可以从命令行更改编译器：

```
make CC="icc -O2"
make FC="gfortran -g"
```

练习。 按建议调用 *Make*（在 `make clean` 之后）。你在屏幕输出中看到了区别吗?

> 预期结果。 编译行现在显示了添加的编译器选项 `-O2` 或 `-g`。

Make 也有 自动变量：

`$@` 目标。在主程序的链接行中使用此变量。`$^` 先决条件列表。在程序的链接行中也使用此变量。`$<` 第一个先决条件。在单个对象文件的编译命令中使用此变量。`$*` 在 模板规则 (第 3.3.2 节) 中，这与模板部分匹配，即对应于 `%` 的部分。

U使用这些变量， `fooprog` 的规则变为

```
fooprog : foo.o bar.o
```

^ 和一个典型的编译行变成

```
foo.o : foo.c bar.h${CC} -c $<
```

你
也可以声明一个变量

```
THEPROGRAM = fooprog
```

并在你的 makefile 中使用这个变量代替程序名。这使得以后更改可执行文件的名称更加容易。

练习。编辑你的 makefile，添加这个变量定义，并用它代替字面上的程序名。构造一个命令行，使你的 makefile 能够构建可执行文件 `fooprog_v2`。

预期结果。你需要使用语法 `make VAR=value` 在命令行上指定 `THEPROGRAM` 变量。

注意事项。确保命令行中等号两边没有空格。

这些自动变量的完整列表可以在 https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html 中找到。

3.3.2 模板规则

到目前为止，你为每个需要编译的文件编写了单独的规则。然而，各种 `.c` 文件的规则非常相似：

- 规则头部（`foo.o : foo.c`）声明源文件是具有相同基本名称的对象文件的前提条件；
- 并且编译指令（`${CC} -c $<`）甚至逐字符完全相同，因为你现在使用了 *Make* 的内置变量；
- 唯一不同的规则是

```
foo.o : foo.c bar.h
        ${CC} -c $<
```

对象文件依赖于源文件和另一个 ther file.

我们可以提取共性并总结为一个模板规则 [1]：

```
%.o : %.c
        ${CC} -c $<
%.o : %.F
        ${FC} -c $<
```

这表明任何对象文件都依赖于具有相同基本名称的 C 或 Fortran 文件。要重新生成对象文件，请使用 `-c` 标志调用 C 或 Fortran 编译器。这些模板规则可以替代上述 makefile 中多个特定目标的规则，除了针对 `foo.o` 的规则。

`foo.o` 对 `bar.h` 的依赖，或 `foomain.o` 对 `foomod.o` 的依赖，可以通过添加一条规则来处理

```
# C
foo.o : bar.h
# Fortran
foomain.o : foomod.o
```

1. 这个机制是你第一次见到的仅存在于 GNU make 中的机制，尽管在这个特定情况下，标准 make 中也有类似的机制。下一节中的通配符机制则不会如此。

没有进一步的指示。该规则声明，“如果文件 `bar.h` 或 `foomod.o` 发生变化，文件 `foo.o` 或 `foomain.o` 也需要更新”。*Make* 随后会在 makefile 中搜索另一条规则，说明如何进行此更新，并且它会找到模板规则。

Exe练习。修改你的 makefile 以纳入这些想法 s, and test.

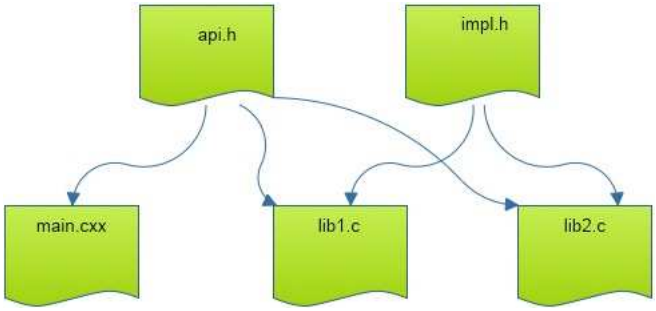


图 3.1：包含主程序和两个库文件的文件结构。

练习 3.1. 为以下结构编写一个 makefile：

- 有一个主文件 `libmain.cxx`，和两个库文件 `libf.cxx` `libg.cxx`；
- 有一个头文件 `libapi.h`，它提供了库文件中函数的原型；
- 有一个头文件 `libimpl.h`，它给出了实现细节，仅用于库文件中。此点在

图 3.1 中说明。

Here is how you can test it:

更改源文件只会重新编译该文件：

```
clang++ -c libf.cxx
clang++ -o main \
    libmain.o libf.o libg.o
```

Changing the implementation header only recom-
堆叠库：

```
clang++ -c libf.cxx
clang++ -c libg.cxx
clang++ -o main libmain.o libf.o
    libg.o
```

更改 `libapi.h` 会重新编译所有内容：

```
clang++ -c libmain.cxx
clang++ -c libf.cxx
clang++ -c libg.cxx
clang++ -o main libmain.o libf.o
    libg.o
```

对于 Fortran，我们没有头文件，所以我们到处使用 模块；见图 3.3。如果你知道如何使用子模块，这是 *Fortran2008* 的一个特性，你可以使下一个练习的效率达到 C 版本的水平。

练习 3.2. 为以下结构编写一个 makefile：

- 有一个主文件 `libmain.f90`，它使用了一个模块 `api.f90`；
- 有两个低级模块 `libf.f90` `libg.f90`，它们被 `api.f90` 使用。如果你使用模块，你可能会进行

比需要更多的编译。为了达到最佳

解决方案，使用子模块。

Source file `mainprog.cxx`:

```
#include <cstdio>
#include "api.h"

int main() {
    int n = f() + g();
    printf("%d\n",n);
    return 0;
}
```

Source file `libf.cxx`:

```
#include "impl.h"
#include "api.h"

int f() {
    struct impl_struct foo;
    return 1;
};
```

Source file `libg.cxx`:

```
#include "impl.h"
#include "api.h"

int g() {
    struct impl_struct foo;
    return 2;
};
```

Header file `api.h`:

```
int f();
int g();
```

Header file `impl.h`:

```
struct impl_struct {};
```

图 3.2：练习的源文件 3.1.

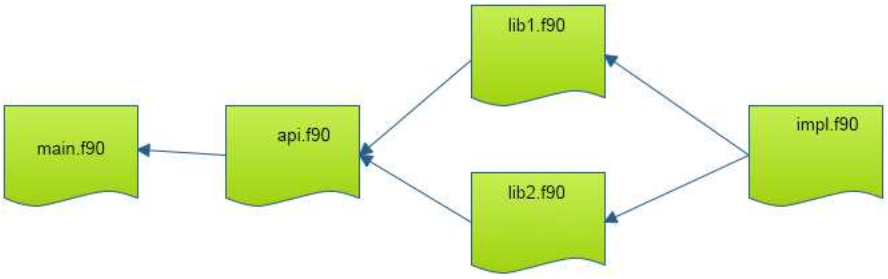


图 3.3：包含主程序和两个库文件的文件结构。

3.3.3 通配符

你的 makefile 现在使用一个通用规则来编译任何源文件。通常，你的源文件将是目录中所有的 `.c` 或 `.F` 文件，那么有没有办法声明 “ 编译此目录中的所有文件 ” 呢？确实有。

将以下行添加到你的 makefile 中，并在适当的地方使用变量 `COBJECTS` 或 `FOBJECTS`。命令 *`wildcard`* 给出 `ls` 的结果，你可以使用 *`patsubst`* 来操作文件名。

```
# wildcard: find all files that match a pattern
CSOURCES := ${wildcard *.c}
# pattern substitution: replace one pattern string by another
COBJECTS := ${patsubst %.c,%.o,${SRC}}

FSOURCES := ${wildcard *.F}
FOBJECTS := ${patsubst %.F,%.o,${SRC}}
```

3.3.4 更多函数

GNU make 具有更多可以在 makefile 中调用的函数。一些示例：

```
HOSTNAME := $(shell hostname -f)
SOURCES  := $(wildcard *.c)
OBJECTS  := $(patsubst %.c,%.o,${SOURCES})
RECURSIVE := $(foreach d,${DIRECTORIES},$(wildcard ${d}/*.c))
```

File name manipulation:

```
$(dir a/b/c.x)    # gives `a/b'
$(dir c.x)        # gives `./'
$(notdir a/b/c.x) # gives `c.x'
$(suffix a/b/c.x) # gives `.x'
```

F或完整列表见 https://www.gnu.org/software/make/manual/html_node/Functions.html。

```
// Makefile                                          SRC: src/f1.c src/f2.c
SRC := ${wildcard src/*.c}

OBJ := ${patsubst src/%,obj/%,${patsubst %.c        OBJ: obj/f1.o obj/f2.o
    %.o,${SRC}}}

PRE := ${addprefix /usr/lib,${SRC} othersrc         PRE: /usr/libsrc/f1.c /usr/libsrc/f2.c /usr/
    moresrc}                                            libothersrc /usr/libmoresrc


BAK := ${addsuffix .bak,${SRC}}                     BAK: src/f1.c.bak src/f2.c.bak
```

3.3.5 条件语句

有多种方法可以使 makefile 的行为动态化。例如，你可以在动作行中放置一个 shell 条件语句。然而，这可能会使 makefile 显得杂乱；更简单的方法是使用 makefile 条件语句。条件语句有两种类型：字符串相等测试和环境变量测试。

第一种类型看起来像

```
ifeq "${HOME}" "/home/thisisme"
  # case where the executing user is me
else ifeq "${HOME}" "/home/buddyofmine"
  # case for other user
else
  # case where it's someone else
endif
```

在第二种情况下，测试看起来像

```
ifdef SOME_VARIABLE
```

true 和 false 部分的文本可以是 makefile 的几乎任何部分。例如，可以让规则中的某一条动作行有条件地包含。然而，大多数情况下你会使用条件语句使变量的定义依赖于某些条件。

练习。假设你想在家和工作时都使用你的 makefile。在工作时，你的雇主拥有 Intel 编译器 `icc` 的付费许可证，但在家你使用开源的 Gnu 编译器 `gcc`。编写一个 makefile，使其在两处都能工作，并为 `CC` 设置合适的值。

3.4 杂项

3.4.1 伪目标

示例 makefile 包含一个目标 `clean`。它使用 *Make* 机制来完成一些与文件创建无关的操作：调用 `make clean` 会导致 *Make* 推断 “ 没有名为 `clean` 的文件，因此需要执行以下指令 ”。然而，这实际上并不会导致文件 `clean` 的生成，因此再次调用 `make clean` 会使相同的指令被执行。

要表示此规则实际上不生成目标，您可以使用 *.PHONY* 关键字：

```
.PHONY : clean
```

大多数情况下，即使没有此声明，makefile 实际上也能正常工作，但将目标声明为伪目标的主要好处是，即使您有一个名为 `clean` 的文件（或文件夹），*Make* 规则仍然会生效。

3.4.2 目录

为临时材料（如对象文件）设置一个目录是常见的策略。因此，您会有一个规则

```
obj/%.o : %.c
    ${CC} -c $< -o $@
```

并且删除临时文件：

```
clean ::
    rm -rf obj
```

这就引出了一个问题，即如何创建 `obj` 目录。你可以这样做：

```
obj/%.o : %.c
    mkdir -p obj
    ${CC} -c $< -o $@
```

但更好的解决方案是使用 *order-only prerequisite*s。

```
obj :
    mkdir -p obj
obj/%.o : %.c | obj
    ${CC} -c $< -o $@
```

这只测试对象目录的存在性，而不测试其时间戳。

3.4.3 使用 thetarget 作为先决条件

Suppose you have two different targets that are treated largely the same. You would want to write:

```
PROGS = myfoo other
${PROGS} : $@.o # this is wrong!!
        ${CC} -o $@ $@.o ${list of libraries goes here}
```

并且说 `make myfoo` 会导致

```
cc -c myfoo.c
cc -o myfoo myfoo.o ${list of libraries}
```

以及对于 `make other`。这里出错的原因是将 `$@.o` 用作前提条件。在 Gnu Make 中，可以按如下方式修复此问题 [2]:

```
.SECONDEXPANSION:
${PROGS} : $$@.o
        ${CC} -o $@ $@.o ${list of libraries goes here}
```

练习。编写第二个主程序 `foosecond.c` 或 `foosecond.F`，并修改你的 makefile，使得调用 `make foo` 和 `make foosecond` 都使用相同的规则。

3.4.4 预定义变量和规则

调用 `make -p yourtarget` 会使 make 打印出它的所有操作，以及你 makefile 中和预定义的所有变量和规则的值。如果你在一个没有 makefile 的目录中执行此操作，你会看到 make 实际上已经知道如何编译 `.c` 或 `.F` 文件。找到这个规则并找到其中变量的定义。

你会看到你可以通过设置诸如 `CFLAGS` 或 `FFLAGS` 这样的变量来自定义 make。通过一些实验来确认这一点。如果你想为相同的源文件制作第二个 makefile，你可以调用 `make-f othermakefile` 来使用它，而不是默认的 `Makefile`。

顺便提一下， `makefile` 和 `Makefile` 都是默认 makefile 的合法名称。你的目录中同时存在 `makefile` 和 `Makefile` 并不是一个好主意。

2. 技术解释：Make 现在将对行进行两次查看：第一次 `$$` 被转换为单个 `$`，第二次 `$@` 变成目标的名称。

3.5 Makefile 中的 Shell 脚本

目的。本节中，您将看到一个较长的 shell 脚本出现在 makefile 规则中的示例。

在您迄今为止看到的 makefile 中，命令部分都是单行的。实际上，您可以在这里写任意多行。例如，让我们为构建的程序制作备份规则。

向您的 makefile 添加一个 `backup` 规则。它首先需要做的是创建一个备份目录：

```
.PHONY : backup
backup :
        if [ ! -d backup ] ; then
          mkdir backup
        fi
```

你是自己输入的吗？不幸的是，这样不可行：makefile 规则的命令部分中的每一行都会作为一个单独的程序执行。因此，你需要将整个命令写在一行上：

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
```

或者如果这一行太长：

```
backup :
        if [ ! -d backup ] ; then \
          mkdir backup ; \
        fi
```

（在一行上写长命令只在 *bash* shell 中可行，而在 *csh* shell 中不可行。这也是不使用后者的一个原因。）

接下来我们进行实际的复制：

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
```

但这种备份方案只保存一个版本。让我们制作一个在保存程序名称中包含日期的版本。

Unix `date` 命令可以通过接受格式字符串来自定义其输出。输入以下内容：`date` 这可以在 makefile 中使用。

练习。编辑 `cp` 命令行，使备份文件的名称包含当前日期。

预期结果。提示：你需要使用反引号。如果你不记得反引号的作用，请查阅 Unix 教程第 1.5.3 节。

如果你在 makefile 规则的命令部分定义 shell 变量，你需要注意以下事项。用一个循环扩展你的 `backup` 规则以复制对象文件：

```
#### This Script Has An ERROR!
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
        for f in ${OBJS} ; do \
          cp $f backup ; \
        done
```

（这不是复制的最佳方式，但我们为了演示目的使用它。）这会导致一个错误信息，原因是 *Make* 将 `$f` 解释为外部进程的环境变量。正确的做法是：

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
        for f in ${OBJS} ; do \
          cp $$f backup ; \
        done
```

（在这种情况下，*Make* 在扫描命令行时将双美元符号替换为单个美元符号。在命令行执行期间， `$f` 随后展开为正确的文件名。）

3.6 使用 Make 的实用技巧

这里有几个实用的技巧。

- 调试 makefile 通常令人沮丧且困难。几乎唯一的工具是 `-p` 选项，它会打印出 Make 根据当前 makefile 使用的所有规则。
- 你经常会先输入一个 make 命令，然后调用程序。大多数 Unix shell 允许你通过使用上箭头键来使用 *shell* 命令历史中的命令。不过，这可能会让人感到厌烦，所以你可能会想写

  ```
  make myprogram ; ./myprogram -options
  ```

  并不断重复这样做。这其中有一个风险：如果 make 失败，例如因为编译问题，你的程序仍然会被执行。相反，应该写

  ```
  make myprogram && ./myprogram -options
  ```

  which executes the program conditional upon make concluding successfully.

3.6.1 这个 makefile 做什么?

上面你已经了解到，执行 `make` 命令会自动执行 makefile 中的第一个规则。这在某种意义上很方便[3]，但在另一方面也不便：了解 makefile 允许的所有可能操作的唯一方法是阅读 makefile 本身，或者通常不够充分的文档。

A better idea is to start the makefile with a target

3. 软件开发者之间有一个约定，包可以通过以下顺序安装： `./configure ; make ;make install`，意思是：为这台计算机配置构建过程，进行实际构建，将文件复制到某个系统目录，如 `/usr/bin`。

```
info :
        @echo "The following are possible:"
        @echo "  make"
        @echo "  make clean"
```

现在 `make` 没有显式目标会告诉你 makefile 的功能。

如果你的 makefile 变得更长，你可能想像这样为每个部分做文档说明。这会遇到一个问题：你不能有两个相同目标的规则， `info` 在这种情况下。然而，如果你使用双冒号是 可能的。你的 makefile 会有以下结构:

```
info ::
        @echo "The following target are available:"
        @echo "  make install"
install :
        # ..... instructions for installing
info ::
       @echo "  make clean"
clean :
       # ..... instructions for cleaning
```

3.7 一个用于 LATEX 的 Makefile

Make 工具通常用于编译程序，但也可以有其他用途。本节中，我们将讨论用于 LATEX 文档的 makefile。

我们从一个非常基础的 makefile 开始:

```
info :
        @echo "Usage: make foo"
        @echo "where foo.tex is a LaTeX input file"

%.pdf : %.tex
        pdflatex $<
```

命令 `make myfile.pdf` 将在需要时调用 `pdflatex myfile.tex` 一次。接下来我们重复调用 `pdflatex`，直到日志文件不再报告需要进一步运行:

```
%.pdf : %.tex
        pdflatex $<
        while [ `cat ${basename $@}.log | grep "Rerun to get" \
            | wc -l` -gt 0 ] ; do \
          pdflatex $< ; \
        done
```

我们使用 `${basename fn}` 宏从目标名称中提取不带扩展名的基本名称。

如果文档有参考文献或索引，我们运行 `bibtex` 和 `makeindex`。

```
%.pdf : %.tex
        pdflatex ${basename $@}
        -bibtex ${basename $@}
        -makeindex ${basename $@}
        while [ `cat ${basename $@}.log | grep "Rerun to get" \
                 | wc -l` -gt 0 ] ; do \
          pdflatex ${basename $@} ; \
        done
```

T行首的减号表示如果这些命令失败，*Make* 不应退出。

最后，我们希望利用 *Make* 处理依赖关系的功能。我们可以编写一个包含常规规则的 makefile

```
mainfile.pdf : mainfile.tex includefile.tex
```

但我们也可以显式地发现包含文件。以下 makefile 通过

```
make pdf TEXTFILE=mainfile
```

然后， `pdf` 规则使用一些 Shell 脚本来发现包含文件（但不递归），并且它再次调用 *Make*，调用另一个规则，并显式传递依赖项。

```
pdf :
        export includes=`grep "^.input " ${TEXFILE}.tex \
            | awk '{v=v FS $$2".tex"} END {print v}'` ; \
        ${MAKE} ${TEXFILE}.pdf INCLUDES="$$includes"

%.pdf : %.tex ${INCLUDES}
        pdflatex $< ; \
        while [ `cat ${basename $@}.log \
                | grep "Rerun to get" | wc -l` -gt 0 ] ; do \
          pdflatex $< ; \
        done
```

这个 Shell 脚本也可以在 makefile 之外完成，动态生成 makefile。

第 4 章

Cmake 构建系统

4.1 CMakeasbuild 系统

CMake 是一个通用的构建系统，使用诸如 *Make* 之类的系统作为后端。一般工作流程是：

1. 配置阶段。在这里会解析 `CMakeLists.txt` 文件，并生成一个 `build` 目录。通常看起来像：

   ```
   mkdir build
   cd build
   cmake <source location>
   ```

   Some people create the build directory in the source tree, in which case the *CMake* command is

   ```
   cmake ..
   ```

   其他人将构建目录放在源代码旁边，在这种情况下：

   ```
   cmake ../src_directory
   ```

2. 构建阶段。在这里执行构建目录中的特定于安装的编译。以 *Make* 作为 “ 生成器 ” 时，这将是

   ```
   cd buildmake
   ```
   但更一般地说 `cmake --build <build directory>` 或者，你可以使用诸如 *ninja* 、 *Visual Studio* 或 *XCode* 等生成器：`cmake -G ninja`

   ```
   ## the usual arguments
   ```

3. 安装阶段。这可以将二进制文件移动到永久位置，例如将库文件放入 `/usr/lib`：

   ```
   make install
   ```

   or

   ```
   cmake --install <build directory>
   ```

	通用指令 d 指定
cmake_minimum_require	最小 cmake version
project	该项目的名称和版本号
install	指定安装目标的目录
	项目构建指令
add_executable	指定可执行文件名称
add_library	指定库名称
add_subdirectory	specify subdirectory where cmake also needs to run
target_sources	为目标指定源文件
target_link_libraries	指定可执行文件和库 ries to link into it
target_include_directories	specify include directories, privately or publicly
find_package	other package to use in this build
	实用工具
target_compile_options	要包含的字面选项
target_compile_features	将被翻译的内容 ted by cmake into op- 译文
target_compile_definitions	宏定义设置为私有或公开
file	将宏定义为文件列表
message	诊断打印，受级别规范限制
	控制
if() else() endif()	conditional

表 4.1: Cmake 命令。

然而，安装位置必须在配置阶段就设置好。我们稍后将详细介绍这是如何完成的。

总结来说，本教程所倡导的源外工作流程是

```
ls some_package_1.0.0 # we are outside the source
ls some_package_1.0.0/CMakeLists.txt # source contains cmake file
mkdir builddir && cd builddir # goto build location
cmake -D CMAKE_INSTALL_PREFIX=../installdir \
      ../some_package_1.0.0
make
make install
```

T生成的目录结构如图 4.1 所示。

图 4.1：源内（左）和源外（右）构建方案。

4.1.1 目标理念

现代 *CMake* 通过声明目标及其需求来工作。目标是将要构建的对象，需求可以是诸如

- 源文件
- 包含路径和其他路径
- 编译器和链接器选项。

构建需求，即构建过程中使用的路径 / 文件 / 选项，与使用需求，即在使用目标时需要生效的路径 / 文件 / 选项，是有区别的。

For requirements during building:

```
target_some_requirement( <target> PRIVATE <requirements> )
```

使用要求：

```
target_some_requirement( <target> INTERFACE <requirements> )
```

组合：

```
target_some_requirement( <target> PUBLIC <requirements> )
```

供将来参考，需求以 `target_include_directories`, `target_compile_definitions`, `target_compile_options`, `target_compile_features`, `target_sources`, `target_link_libraries`, *target_link_options*, `target_link_directories` 指定。

4.1.2 语言

CMake 主要针对 C++，但它也轻松支持 C。对于 *Fortran* 支持，首先执行

```
enable_language(Fortran)
```

注意大小写：这同样适用于所有变量，如 `CMAKE_Fortran_COMPILER`。

4.1.3 脚本结构

本节学习的命令

`cmake_minimum_required`	声明此脚本的最低要求版本
`project`	为此项目声明一个名称

CMake 由 *`CMakeLists.txt`* 文件驱动。该文件需要位于项目的根目录中。（你也可以在子目录中有同名文件。）

S由于 *CMake* 多年来发生了很大变化，并且仍在不断发展，最好在每个s带有（最低）所需版本声明的脚本：

```
cmake_minimum_required( VERSION 3.12 )
```

您可以查询 *CMake* 可执行文件的版本：

```
$ cmake --version
cmake version 3.19.2
```

您还需要声明项目名称和版本，这些不必对应任何文件名：

```
project( myproject VERSION 1.0 )
```

4.2 示例案例

4.2.1 从源码生成可执行文件

(本示例的文件位于 `tutorials/cmake/single`。)

本节学习的命令

`add_executable`	声明一个可执行文件作为目标
`target_sources`	为目标指定源文件
`install`	指明安装该项目的位置
`PROJECT_NAME`	展开为项目名称的宏

If you have a project that is supposed to deliver an executable, you declare in your `CMakeLists.txt`:

```
add_executable( myprogram )
target_sources( myprogram PRIVATE program.cxx )
```

Often, the name of the executable is the name of the project, so you'd specify:

```
add_executable( ${PROJECT_NAME} )
```

Remark 14 *An older usage is to specify the sources files directly with the executable. The above usage is preferred.*

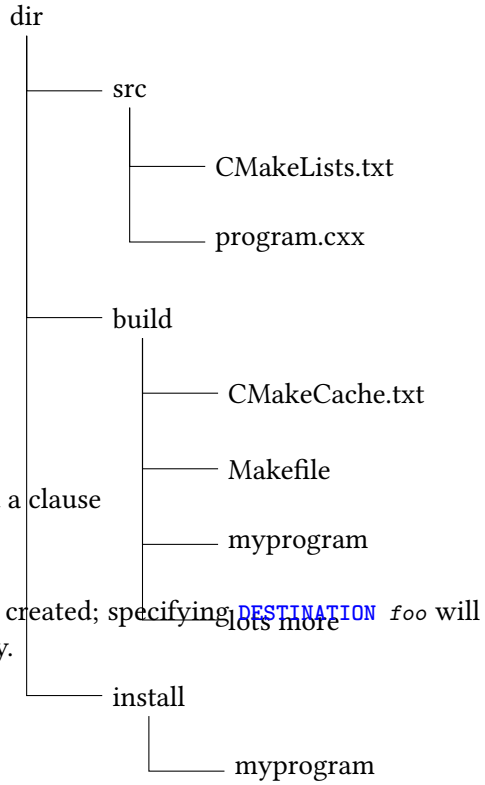
```
add_executable( myprogram program.cxx )
```

In order to move the executable to the install location, you need a clause

```
install( TARGETS myprogram DESTINATION . )
```

Without the `DESTINATION` clause, a default `bin` directory will be created; specifying `DESTINATION foo` will put the target in a `foo` sub-directory of the installation directory.

In the figure on the right we have also indicated the `build` directory, which from now on we will not show again. It contains automatically generated files that are hard to decyper, or debug. Yes, there is a `Makefile`, but even for simple projects this is too complicated to debug by hand if your *CMake* installation misbehaves.



这是完整的 `CMakeLists.txt`:

```
cmake_minimum_required( VERSION 3.12 )
project( singleprogram VERSION 1.0 )

add_executable( program )
target_sources( program PRIVATE program.cxx )
install( TARGETS program DESTINATION . )
```

4.2.2 构建库

本节学习的命令

`target_include_directories`	指示包含文件的位置
`add_library`	声明一个库及其源文件
`target_link_libraries`	指明该库属于一个可执行文件

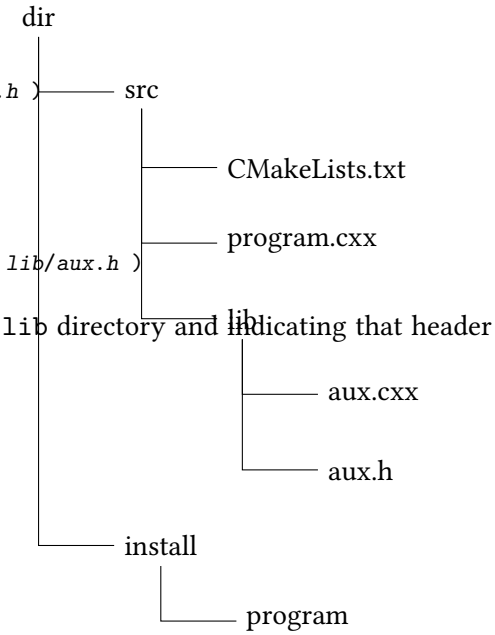
可执行文件

4.2.2.1 多文件

(本例的文件位于 `tutorials/cmake/multiple`。)

If there is only one source file, the previous section is all you need. If there are multiple files, you could write

```
add_executable( program )
target_sources( program PRIVATE program.cxx aux.cxx aux.h )
```



You can also put the non-main source files in a separate directory:

```
add_executable( program )
target_sources( program PRIVATE program.cxx lib/aux.cxx lib/aux.h )
```

However, often you will build libraries. We start by making a `lib` directory and indicating that header files need to be found there:

```
target_include_directories( program PRIVATE lib )
```

The actual library is declared with an `add_library` clause:

```
add_library( auxlib )
target_sources( auxlib PRIVATE aux.cxx aux.h )
```

Next, you need to link that library into the program:

```
target_link_libraries( program PRIVATE auxlib )
```

该 `PRIVATE` 子句表示该库仅用于构建可执行文件的目的。（使用 `PUBLIC` 可以将库包含在安装中；我们将在第 4.2.2.3 节中探讨。）

完整的 `CMakeLists.txt`:

```
cmake_minimum_required( VERSION 3.12 )
project( cmakeprogram VERSION 1.0 )

add_executable( program )
target_sources( program PRIVATE program.cxx )

add_library( auxlib STATIC )
target_sources( auxlib PRIVATE aux1.cxx aux2.cxx aux.h )

target_link_libraries( program PRIVATE auxlib )
install( TARGETS program DESTINATION . )
```

请注意，私有共享库没有意义，因为它们会导致运行时未解析的引用。

4.2.2.2 测试生成的 *makefiles*

在 Make 教程中 3 你学到了当只做了有限的编辑时，Make 将只重新编译严格必要的文件。由 *CMake* 生成的 makefiles 行为类似。以上述结构为例，我们首先修改 `aux.cxx` 文件，这需要重新构建库：

```
----------------
touch a source file and make:
Consolidate compiler generated dependencies of target auxlib
[ 25%] Building CXX object CMakeFiles/auxlib.dir/aux.cxx.o
[ 50%] Linking CXX static library libauxlib.a
[ 50%] Built target auxlib
Consolidate compiler generated dependencies of target program
[ 75%] Linking CXX executable program
[100%] Built target program
```

另一方面，如果我们编辑一个头文件，主程序也需要重新编译：

```
----------------
touch a source file and make:
Consolidate compiler generated dependencies of target auxlib
[ 25%] Building CXX object CMakeFiles/auxlib.dir/aux.cxx.o
[ 50%] Linking CXX static library libauxlib.a
[ 50%] Built target auxlib
Consolidate compiler generated dependencies of target program
[ 75%] Linking CXX executable program
[100%] Built target program
```

4.2.2.3 为发布制作库

(此示例的文件位于 `tutorials/cmake/withlib`。)

Commands learned in this section

`SHARED`	indicated to make shared libraries

In order to create a library we use `add_library`, and we link it into the target program with `target_link_libraries`.

By default the library is build as a static `.a` file, but adding

```
add_library( auxlib SHARED )
```

or adding a runtime flag

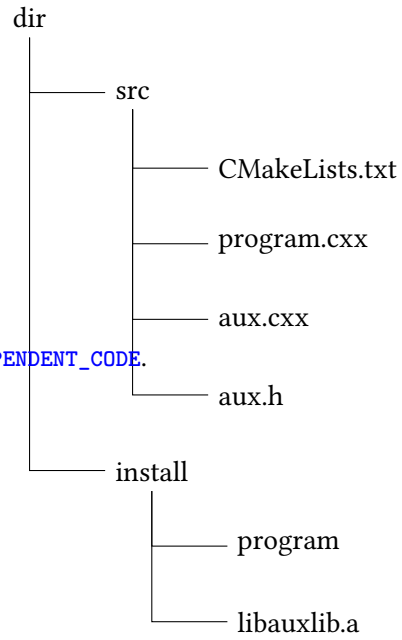
```
cmake -D BUILD_SHARED_LIBS=TRUE
```

changes that to a shared `.so` type.

Related: the *-fPIC* compile option is set by `CMAKE_POSITION_INDEPENDENT_CODE`.

The full *CMake* file:

```
cmake_minimum_required( VERSION 3.12 )
project( cmakeprogram VERSION 1.0 )

add_executable( program )
target_sources( program PRIVATE program.cxx )

add_library( auxlib STATIC )
target_sources( auxlib PRIVATE lib/aux.cxx lib/aux.h )

target_link_libraries( program PUBLIC auxlib )
target_include_directories( program PRIVATE lib )

install( TARGETS program DESTINATION bin )
install( TARGETS auxlib DESTINATION lib )
install( FILES lib/aux.h DESTINATION include )
```



注意我们给出了库文件的路径。这是相对于当前目录解释的（从 CMake-3.13 起）；该当前目录可用作 `CMAKE_CURRENT_SOURCE_DIR`。

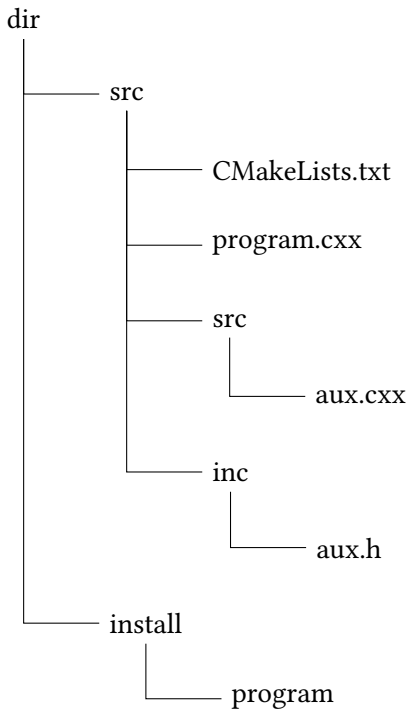
4.2.3 构建过程中使用子目录

(此示例的文件位于 `tutorials/cmake/includedir`。)

本节学习的命令

`target_include_directories`	指示所需的包含目录
`target_sources`	为目标指定更多源
`CMAKE_CURRENT_SOURCE_DIR`	展开为当前目录的变量
`file`	为多个文件定义单一名称的同义词
`GLOB`	为多个文件定义单一名称的同义词

Suppose you have a directory with header files, as in the diagram on the right. The main program would have




```
#include <iostream>
using namespace std;

#include "aux.h"

int main() {
  aux1();
  aux2();
  return 0;
}
```

and which is compiled as:

```
cc -c program.cxx -I./inc
```

To make sure the header file gets found during the build, you specify that include path with `target_include_directories`:

```
target_include_directories(
        program PRIVATE
        "${CMAKE_CURRENT_SOURCE_DIR}/inc" )
```

It is best to make such paths relative to `CMAKE_CURRENT_SOURCE_DIR`, or the source root `CMAKE_SOURCE_DIR`, or equivalently `PROJECT_SOURCE_DIR`

通常，当你开始构建这样的目录结构时，你也会在子目录中有源代码。如果你只需要将它们编译到主可执行文件中，你可以将它们列入一个变量

```
set( SOURCES program.cxx src/aux.cxx )
```

并使用该变量。然而，这种做法已被弃用；推荐使用 `target_sources`:

```
target_sources( program PRIVATE src/aux1.cxx src/aux2.cxx )
```

使用通配符并非简单:

```
file( GLOB AUX_FILES "src/*.cxx" )
target_sources( program PRIVATE ${AUX_FILES} )
```

完整的 *CMake* 文件:

```
cmake_minimum_required( VERSION 3.14 )
project( cmakeprogram VERSION 1.0 )

add_executable( program )
target_sources( program PRIVATE program.cxx )
## target_sources( program PRIVATE src/aux1.cxx src/aux2.cxx )
file( GLOB AUX_FILES "src/*.cxx" )
target_sources( program PRIVATE ${AUX_FILES} )
target_include_directories(
        program PRIVATE
        "${CMAKE_CURRENT_SOURCE_DIR}/inc" )

install( TARGETS program DESTINATION . )
```

4.2.4 递归构建；rpath

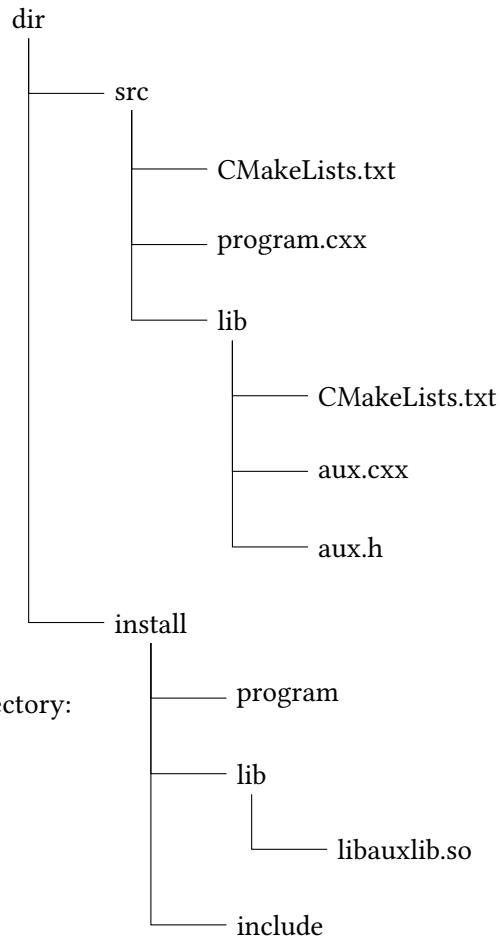
(此示例的文件位于 `tutorials/cmake/publiclib`。)

Commands learned in this section

`add_subdirectory`	declare a subdirectory where cmake needs to be run
`CMAKE_CURRENT_SOURCE_DIR`	directory where this command is evaluated
`CMAKE_CURRENT_BINARY_DIR`	
`LIBRARY_OUTPUT_PATH`	
`FILES_MATCHING PATTERN`	wildcard indicator

If your sources are spread over multiple directories, there needs to be a `CMakeLists.txt` file in each, and you need to declare the existence of those directories. Let's start with the obvious choice of putting library files in a `lib` directory with `add_subdirectory`:

```
add_subdirectory( lib )
```



For instance, a library directory would have a `CMakeLists.txt` file:

```
cmake_minimum_required( VERSION 3.14 )
project( auxlib )

add_library( auxlib SHARED )
target_sources( auxlib PRIVATE aux.cxx aux.h )
target_include_directories(
    auxlib PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}" )
install( TARGETS auxlib DESTINATION lib )
install( FILES aux.h DESTINATION include )
```

to build the library file from the sources indicated, and to install it in a `lib` subdirectory.

We also add a clause to install the header files in an include directory:

```
install( FILES aux.h DESTINATION include )
```

For installing multiple files, use

```
install(DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
    DESTINATION ${LIBRARY_OUTPUT_PATH}
    FILES_MATCHING PATTERN "*.h")
```

一个问题是告诉可执行文件在哪里找到库。为此我们使用 *rpath* 机制。（参见第 2.3.3 节。）默认情况下，*CMake* 设置为可执行文件在构建位置可以找到库。如果你使用非平凡的安装前缀，以下几行代码有效：

```
set( CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib" )
set( CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE )
```

注意这些必须在目标之前指定。

整个文件:

```
cmake_minimum_required( VERSION 3.14 )
project( cmakeprogram VERSION 1.0 )

set( CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib" )
set( CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE )

add_executable( program )
target_sources( program PRIVATE program.cxx )
add_subdirectory( lib )
target_include_directories(
        auxlib PUBLIC "${CMAKE_CURRECT_SOURCE_DIR}" )
target_link_libraries(
        program PUBLIC auxlib )

install( TARGETS program DESTINATION . )
```

4.2.5 使用其他库的程序

到目前为止，我们讨论了可以单独使用的可执行文件和库。如果你的构建结果需要外部库，该怎么办？我们将在第 4.3 节讨论如何查找这些库；这里我们指出它们的使用。

问题是当有人使用你的二进制文件时，这些库需要是可被找到的。有两种策略：

- 确保它们已被添加到 *LD_LIBRARY_PATH*；
- 让链接器通过 *rpath* 机制将它们的位置添加到二进制文件本身。事实上，由于 “ 系统完整性保护 ”，这个第二个选项是近期版本的 *Apple Mac OS* 上唯一可用的。

作为示例，假设你的二进制文件需要 *Catch2* 和 *fmtlib* 库。除了 `target_link_directories` 规范之外，你还需要：

```
set_target_properties(
        ${PROGRAM_NAME} PROPERTIES
        BUILD_RPATH   "${CATCH2_LIBRARY_DIRS};${FMTLIB_LIBRARY_DIRS}"
        INSTALL_RPATH "${CATCH2_LIBRARY_DIRS};${FMTLIB_LIBRARY_DIRS}"
    )
```

4.2.6 仅头文件库

使用 `INTERFACE` 关键字。

4.3 查找和使用外部包

如果你的程序依赖于其他库，有多种方法可以让 *CMake* 找到它们。

4.3.1 CMake commandline options

(此示例的文件位于 `tutorials/cmake/usepubliclib`。)

您可以在命令行中明确指定外部库的位置。

```
cmake -D OTHERLIB_INC_DIR=/some/where/include
      -D OTHERLIB_LIB_DIR=/somewhere/lib
```

示例 *CMake* 文件：

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )

# with environment variables
# set( AUX_INCLUDE_DIR $ENV{TACC_AUX_INC} )
# set( AUX_LIBRARY_DIR $ENV{TACC_AUX_LIB} )

# with cmake -D options
```

```
option( AUX_INCLUDE_DIR "include dir for auxlib" )
option( AUX_LIBRARY_DIR   "lib dir for auxlib" )

add_executable( program )
target_sources( program PRIVATE program.cxx )
target_include_directories(
        program PUBLIC
        ${AUX_INCLUDE_DIR} )
target_link_libraries( program PUBLIC auxlib )
target_link_directories(
        program PUBLIC
        ${AUX_LIBRARY_DIR} )
install( TARGETS program DESTINATION . )
```

4.3.2 通过 ‘find library’ 和 ‘find package’ 查找包

本节学习的命令

find_library	find a library with a FOOConfig.cmake file
CMAKE_PREFIX_PATH	用于查找 FOOConfig.cmake 文件
find_package	使用带有 FindFOO 模块的库
CMAKE_MODULE_PATH	FindFOO 模块的位置

find_package 命令查找名称为 FindXXX.cmake 的文件，这些文件在 CMAKE_MODULE_PATH 上被搜索。不幸的是，find_package 的工作方式在某种程度上依赖于具体的包。例如，大多数包会设置一个变量 *FooFound*，你可以测试它

```
find_package( Foo )
if ( FooFound )
    # do something
else()
    # throw an error
endif()
```

S有些库附带一个 FOOConfig.cmake 文件，该文件会在 CMAKE_PREFIX_PATH 中被搜索 find_library。您通常将此变量设置为包安装的根目录，CMake 将会找到 t .cmake 文件的目录。

您可以测试设置的变量：

```
find_library( FOOLIB foo )
if (FOOLIB)
  target_link_libraries( myapp PRIVATE ${FOOLIB} )
else()
  # throw an error
endif()
```

4.3.2.1 示例：*Range-v3*

```
find_package( range-v3 REQUIRED )
target_link_libraries( ${PROGRAM_NAME} PUBLIC range-v3::range-v3 )
```

4.3.3 通过 ‘pkg config’ 使用其他包

如今，许多包支持 *pkgconfig* 机制。

1. 假设你有一个库 mylib，安装在 /opt/local/mylib。 2. 如果 mylib 支持 pkgconfig，很可能存在一个路径 /opt/local/mylib/lib/pkgconfig，其中包含一个文件 mylib.pc。
3. 将包含 *.pc* 文件的路径添加到 *PKG_CONFIG_PATH* 环境变量中。

Cmake 现在能够找到 mylib：

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( MYLIBRARY REQUIRED mylib )
```

这定义了变量

```
MYLIBRARY_INCLUDE_DIRS
MYLIBRARY_LIBRARY_DIRS
MYLIBRARY_LIBRARIES
```

然后你可以在 target_include_directories 和 target_link_directories target_link_librarie 命令中使用它们。s

4.3.4 编写你自己的 pkg 配置

我们扩展了第 4.2.4 节的配置以生成一个 .pc 文件 .

首先我们需要一个 .pc 文件的模板：

```
prefix="@CMAKE_INSTALL_PREFIX@"
exec_prefix="${prefix}"
libdir="${prefix}/lib"
includedir="${prefix}/include"

Name: @PROJECT_NAME@
Description: @CMAKE_PROJECT_DESCRIPTION@
Version: @PROJECT_VERSION@
Cflags: -I${includedir}
Libs: -L${libdir} -l@libtarget@
```

这里的 @ 符号界定了 CMake 宏，这些宏将在生成 .pc 文件时被替换；美元宏则保持不变地写入文件。

生成该文件由 CMake 配置中的以下几行完成：

```
set( libtarget auxlib )
configure_file(
    ${CMAKE_CURRENT_SOURCE_DIR}/${PROJECT_NAME}.pc.in
    ${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME}.pc
    @ONLY
)
install(
    FILES ${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME}.pc
    DESTINATION share/pkgconfig
)
```

4.3.5 库

4.3.5.1 Example: MPI

(此示例的文件位于 `tutorials/cmake/mpiprog`。)

虽然许多 MPI 实现都有一个 `.pc` 文件，但最好使用 FindMPI 模块。该包定义了许多变量，可用于查询找到的 MPI；详情请参见 https://cmake.org/cmake/help/latest/module/FindMPI.html 有时需要设置 *MPI_HOME* 环境变量以帮助发现 MPI 包。

C 版本:

```
cmake_minimum_required( VERSION 3.12 )
project( cxxprogram VERSION 1.0 )

add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.cxx )
find_package( MPI )
target_include_directories(
        ${PROJECT_NAME} PUBLIC
        ${MPI_C_INCLUDE_DIRS} ${CMAKE_CURRENT_SOURCE_DIR} )
target_link_libraries(
        ${PROJECT_NAME} PUBLIC
        ${MPI_C_LIBRARIES} )

install( TARGETS ${PROJECT_NAME} DESTINATION . )
```

Fortran version:

```
cmake_minimum_required( VERSION 3.12 )
project( ${PROJECT_NAME} VERSION 1.0 )

enable_language(Fortran)

find_package( MPI )

if( MPI_Fortran_HAVE_F08_MODULE )
else()
  message( FATAL_ERROR "No f08 module for this MPI" )
endif()

add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.F90 )
target_include_directories(
        ${PROJECT_NAME} PUBLIC
        ${MPI_Fortran_INCLUDE_DIRS} ${CMAKE_CURRENT_SOURCE_DIR} )
target_link_directories(
        ${PROJECT_NAME} PUBLIC
        ${MPI_LIBRARY_DIRS} )
target_link_libraries(
        ${PROJECT_NAME} PUBLIC
        ${MPI_Fortran_LIBRARIES} )

install( TARGETS ${PROJECT_NAME} DESTINATION . )
```

MPL 包也使用了 `find_package`:

```
find_package( mpl REQUIRED )
add_executable( ${PROJECT_NAME} )
target_sources( ${PROJECT_NAME} PRIVATE ${PROJECT_NAME}.cxx )
target_include_directories(
        ${PROJECT_NAME} PUBLIC
        mpl::mpl )
target_link_libraries(
        ${PROJECT_NAME} PUBLIC
        mpl::mpl )
```

4.3.5.2 示例：*OpenMP*

```
find_package(OpenMP)
if(OpenMP_C_FOUND) # or CXX
else()
        message( FATAL_ERROR "Could not find OpenMP" )
endif()
# for C:
add_executable( ${program} ${program}.c )
target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_C )
# for C++:
add_executable( ${program} ${program}.cxx )
target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_CXX)
# for Fortran
enable_language(Fortran)
# test: if( OpenMP_Fortran_FOUND )
add_executable( ${program} ${program}.F90 )
target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_Fortran )
```

4.3.5.3 CUDA

```
FindCUDA

Changed in version 3.27: This module is available only if policy CMP0146 is not set to N

Deprecated since version 3.10: Do not use this module in new code.

It is no longer necessary to use this module or call find_package(CUDA) for compiling CU

New in version 3.17: To find and use the CUDA toolkit libraries manually, use the FindCU
```

4.3.5.4 Example: MKL

(此示例的文件位于 `tutorials/cmake/mklcmake`。) Intel 编译器安装包含 *CMake* 支持：有一个文件 `MKLConfig.cmake`。使用 MKL 中的 *Cblas* 的示例程序：

```
#include <iostream>
```

```
#include <vector>
using namespace std;

#include "mkl_cblas.h"

int main() {
  vector<double> values{1,2,3,2,1};
  auto maxloc = cblas_idamax ( values.size(),values.data(),1);
  cout << "Max abs at: " << maxloc << " (s/b 2)" << '\n';

  return 0;
}
```

以下配置文件列出了各种选项等内容:

```
cmake_minimum_required( VERSION 3.12 )
project( mklconfigfind VERSION 1.0 )

## https://www.intel.com/content/www/us/en/develop/documentation/onemkl-linux-developer-guide/
    top/getting-started/cmake-config-for-onemkl.html

find_package( MKL CONFIG REQUIRED )
add_executable( program program.cxx )target_compile_options(
program PUBLIC$<TARGET_PROPERTY:MKL::MKL,
INTERFACE_COMPILE_OPTIONS> )target_include_directories(
program PUBLIC$<TARGET_PROPERTY:MKL::MKL,
INTERFACE_INCLUDE_DIRECTORIES> )target_link_libraries(
program PUBLIC$<LINK_ONLY:MKL::MKL>)
install( TARGETS program DESTINATION . )
```

4.3.5.5 示例: *PETSc*

(此示例的文件位于 `tutorials/cmake/petscprog`。)

此 *CMake* 设置搜索 `petsc.pc`, 其位于 `$PETSC_DIR/$PETSC_ARCH/lib/pkgconfig`:

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )

find_package( PkgConfig REQUIRED )
pkg_check_modules( PETSC REQUIRED petsc )
message( STATUS "PETSc includes: ${PETSC_INCLUDE_DIRS}" )
message( STATUS "PETSc libraries: ${PETSC_LIBRARY_DIRS}" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${PETSC_INCLUDE_DIRS} )
```

```
target_link_directories(
        program PUBLIC
        ${PETSC_LIBRARY_DIRS} )
target_link_libraries(
        program PUBLIC petsc )

install( TARGETS program DESTINATION . )
```

4.3.5.6 示例：*Eigen*

(此示例的文件位于 `tutorials/cmake/eigen`。)

eigen 包使用了 *pkgconfig*。

```
cmake_minimum_required( VERSION 3.12 )
project( eigentest )

find_package( PkgConfig REQUIRED )
pkg_check_modules( EIGEN REQUIRED eigen3 )

add_executable( eigentest eigentest.cxx )
target_include_directories(
        eigentest PUBLIC
        ${EIGEN_INCLUDE_DIRS})
```

4.3.5.7 Example: cxxopts

(此示例的文件位于 `tutorials/cmake/cxxopts`。)

cxxopts 包使用了 *pkgconfig*。

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )

find_package( PkgConfig REQUIRED )
pkg_check_modules( OPTS REQUIRED cxxopts )
message( STATUS "cxxopts includes: ${OPTS_INCLUDE_DIRS}" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${OPTS_INCLUDE_DIRS})

install( TARGETS program DESTINATION . )
```

4.3.5.8 Example: fmtlib

(The files for this example are in `tutorials/cmake/fmtlib`.)

在下面的示例中，我们使用了 *fmtlib*。主要的 *CMake* 文件：

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )
```

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( FMTLIB REQUIRED fmt )
message( STATUS "fmtlib includes: ${FMTLIB_INCLUDE_DIRS}" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${FMTLIB_INCLUDE_DIRS})

install( TARGETS program DESTINATION . )
```

4.3.5.9 示例：*fmtlib* 在库中的使用

(此示例的文件位于 `tutorials/cmake/fmtliblib`。)

我们继续使用 *fmtlib* 库，但现在生成的库也引用了该库，因此我们使用 `target_link_directories` 和 `target_link_library`。

主文件：

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )

find_package( PkgConfig REQUIRED )
pkg_check_modules( FMTLIB REQUIRED fmt )
message( STATUS "fmtlib includes : ${FMTLIB_INCLUDE_DIRS}" )
message( STATUS "fmtlib lib dirs : ${FMTLIB_LIBRARY_DIRS}" )
message( STATUS "fmtlib libraries: ${FMTLIB_LIBRARIES}" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${FMTLIB_INCLUDE_DIRS})

add_subdirectory( prolib )
target_link_libraries( program PUBLIC prolib )

install( TARGETS program DESTINATION . )
```

Library file:

```
project( prolib )

add_library( prolib SHARED aux.cxx aux.h )
target_include_directories(
        prolib PUBLIC
        ${FMTLIB_INCLUDE_DIRS})
target_link_directories(
        prolib PUBLIC
        ${FMTLIB_LIBRARY_DIRS})
target_link_libraries(
        prolib PUBLIC fmt )
```

4.4 自定义编译过程

本节学习的命令

add_compile_options	全局编译器选项
target_compile_features	编译的编译器无关规范标志
target_compile_definitions	预处理器标志

4.4.1 自定义编译器

最好明确告诉 *CMake* 你正在使用的编译器，否则它可能会找到系统自带的某个默认 gcc 版本。使用变量 CMAKE_CXX_COMPILER, CMAKE_C_COMPILER,CMAKE_Fortran_COMPILER, CMAKE_LINKER。

或者，通过编译器的显式路径设置环境变量 *CC*, *CXX*, *FC*。例如，对于 Intel 编译器：

```
export CC=`which icc`
export CXX=`which icpc`
export FC=`which ifort`
```

4.4.2 Global 和 target 标志

大多数情况下，编译选项应与目标相关联。例如，某些文件可能需要更高或更低的优化级别，或特定的 C++ 标准。在这种情况下，使用 target_compile_features。

某些选项可能需要是全局的，这种情况下使用 add_compile_options。示例：

```
## from https://youtu.be/eC9-iRN2b04?t=1548
if (MVSC)
    add_compile_options(/W3 /WX)
else()
    add_compile_options(-W -Wall -Werror)
endif()
```

4.4.2.1 通用标志

某些标志具有通用含义，但实现依赖于编译器。例如，要指定 C++ 标准：

```
target_compile_features( mydemo PRIVATE cxx_std_17 )
```

或者，您可以在命令行中设置此项 :

```
cmake -D CMAKE_CXX_STANDARD=20
```

变量 CMAKE_CXX_COMPILE_FEATURES 包含您可以设置的所有特性的列表。

优化标志可以通过指定 CMAKE_BUILD_TYPE 来设置：

- *Debug* 对应于 `-g` 标志;
- *Release* 对应于 `-O3 -DNDEBUG`;
- *MinSizeRel* 对应于 `-Os -DNDEBUG`
- *RelWithDebInfo* 对应于 `-O2 -g -DNDEBUG`。

这个变量通常会从命令行设置:

```
cmake .. -DCMAKE_BUILD_TYPE=Release
```

不幸的是，除了显式设置编译器标志外，这似乎是影响优化标志的唯一方法；参见下一点。

4.4.2.2 自定义编译器标志

设置变量 `CMAKE_CXX_FLAGS` 或 `CMAKE_C_FLAGS`；也可以设置 `CMAKE_LINKER_FLAGS`（但请参见第 4.2.4 节中常用的 *rpath* 选项。）

4.4.3 宏定义

CMake 可以提供宏定义:

```
target_compile_definitions
  ( programname PUBLIC
    HAVE_HELLO_LIB=1 )
```

并且你的源代码可以测试这些:

```
#ifdef HAVE_HELLO_LIB
#include "hello.h"
#endif
```

4.5 CMake scripting

本节学习的命令

`option`	查询命令行选项
`message`	cmake 过程中的跟踪消息
`set`	设置变量的值
`CMAKE_SYSTEM_NAME`	包含操作系统名称的变量
`STREQUALS`	字符串比较运算符

CMakeLists.txt 文件是一个脚本，尽管它看起来不像。

- 指令由一个命令组成，后跟一个括号括起来的参数列表。
- （所有参数都是字符串：没有数字。）
- 每个命令需要从新的一行开始，但其他空白和换行符会被忽略

注释以井号字符开始。

4.5.1 系统依赖

```
if (CMAKE_SYSTEM_NAME STREQUALS "Window")
  target_compile_options( myapp PRIVATE /W4 )
elseif (CMAKE_SYSTEM_NAME STREQUALS "Darwin" -Wall -Wextra -Wpedantic)
  target_compile_options( myapp PRIVATE /W4 )
endif()
```

4.5.2 消息、错误和跟踪

`message` 命令可用于向控制台写入输出。该命令有两个参数:

```
message( STATUS "We are rolling!")
```

你可以指定除 `STATUS` 之外的其他日志级别（该参数在文档中实际上称为 “mode” ）；例如运行

```
cmake --log-level=NOTICE
```

将只显示 ‘notice’ 状态或更高级别的消息。

这里的可能选项是： `FATAL_ERROR`, `SEND_ERROR`, `WARNING`, `AUTHOR_WARNING`, `DEPRECATION`, `NOTICE`, `STATUS`,`VERBOSE`, `DEBUG`, `TRACE`。

`NOTICE`, `VERBOSE`, `DEBUG`, `TRACE` 选项是在 CMake-3.1 中添加的 5.

要完整跟踪 *CMake* 所做的所有操作，请使用命令行选项 `--trace`。

您可以使用该选项获得详细的 make 文件

```
-D CMAKE_VERBOSE_MAKEFILE=ON
```

on the *CMake* invocation. You still need `make V=1`.

4.5.3 变量

Variables are set with `set`, or can be given on the commandline:

```
cmake -D MYVAR=myvalue
```

`-D` 后面的空格是可选的。

单独使用变量会返回其值，除了在字符串中，需要类似 Shell 的表示法:

```
set(SOME_ERROR "An error has occurred")
message(STATUS "${SOME_ERROR}")
set(MY_VARIABLE "This is a variable")
message(STATUS "Variable MY_VARIABLE has value ${MY_VARIABLE}")
```

变量也可以通过 *CMake* 脚本使用 `option` 命令进行查询 :

```
option( SOME_FLAG "A flag that has some function" defaultvalue )
```

有些变量是由其他命令设置的。例如， `project` 命令设置了 `PROJECT_NAME` 和 *PROJECT_VERSION*。

4.5.3.1 环境变量

环境变量 可以通过 *ENV* 命令查询:

```
set( MYDIR $ENV{MYDIR} )
```

4.5.3.2 数值变量

```
math( EXPR lhs_var "math expr" )
```

4.5.4 控制结构

4.5.4.1 条件语句 `if ( MYVAR MATCHES "value$" )`

```
message( NOTICE "Variable ended in 'value'" )
elseif( stuff )message( stuff )else()
message( NOTICE "Variable was otherwise" )endif()
```

4.5.4.2 循环

```
while( myvalue LESS 50 )
    message( stuff )
endwhile()

foreach ( var IN ITEMS item1 item2 item3 )
  ## something wityh ${var}
endforeach()
foreach ( var IN LISTS list1 list2 list3 )
  ## something wityh ${var}
endforeach()
```

整数范围，包含边界，默认上界为零:

```
foreach ( idx RANGE 10 )
foreach ( idx RANGE 5 10 )
foreach ( idx RANGE 5 10 2 )
endforeach()
```

第 5 章

通过 Git 进行源代码控制

在本教程中，您将学习 *git*，目前最流行的版本控制（也称为源代码控制或修订控制）系统。其他类似系统有 *Mercurial* 和 *Microsoft Sharepoint*。早期的系统有 *SCCS*、 *CVS*、 *Subversion*、 *Bitkeeper*。

版本控制是一种通过记录项目文件的连续版本来跟踪软件项目历史的系统。这些版本被记录在代码仓库中，可以是在您正在使用的机器上，也可以是远程的。

这有许多实际的优点：

- 可以撤销更改；
- 与另一位开发者共享一个代码仓库使协作成为可能，包括对同一文件的多次编辑。

- 一个代码仓库记录了项目的历史。
- 你可以拥有项目的多个版本，例如用于探索新功能，或为某些用户进行定制。

T使用版本控制系统是行业标准做法，而 git 无疑是最流行的系统
th如今。

5.1 概念与概述

旧的系统基于拥有一个中央代码仓库，所有开发者都在此协调。如今，一种流行的设置是每个开发者（或一个小组）拥有一个本地代码仓库，并将其同步到一个远程代码仓库。在所谓的分布式版本控制系统中，甚至可以有多个远程代码仓库进行同步。

可以跟踪单个文件的更改，但通常将一组更改打包在一个提交中更有意义。这也使得回滚这组更改变得容易。

如果一个项目处于某种里程碑状态，可以附加一个标签，或将该状态标记为一个发布。

M现代版本控制系统允许你拥有多个分支，即使是在同一个本地代码仓库中。
T这样你可以为项目的发布版本拥有一个主分支，以及一个或多个开发分支

用于探索新功能的分支。当你确认新功能已经经过充分测试后，可以将这些分支合并。

5.2 Git

本实验应由两人完成，以模拟一组程序员共同参与一个联合项目。你也可以自己完成，通过使用代码仓库的两个克隆，最好在你的计算机上打开两个窗口。

5.3 创建并填充代码仓库

目的。 本节中，您将创建一个代码仓库并制作一个本地副本以进行工作。

您可以通过两种不同的方式创建代码仓库：

1. 创建远程代码仓库并进行克隆。2. 本地创建代码仓库，然后连接到远程；这要复杂得多。

5.3.1 通过克隆创建代码仓库

开始一个新的代码仓库最简单的方法是使用像 *github.com* 或 *gitlab.com* 这样的网站，在那里创建代码仓库，然后将其克隆到你的本地机器。语法：

```
git clone URL [ localname ]
```

这会给你一个包含代码仓库内容的目录。如果你省略本地名称，目录将以代码仓库的名称命名。

```
Cmd >> git clone https://github.com/TACC/empty.git↪emptyOut >>Cloning into 'empty'...
warning: You appear to have cloned an empty repository.Cmd >> cd emptyCmd >> ls -aOut >>.
克隆一个空代码仓库并 ..                                                    检查它确
                    .git
实是空的 Cmd >> git statusOut >>On branch mainNo commits yet

nothing to commit (create/copy files and use "git add"↪to track)
```

如你所见，即使是一个空的代码仓库也包含一个目录 `.git`。它包含关于你的代码仓库的管理信息；你几乎不会去查看它。

5.3.2 本地创建代码仓库

你也可以为你的代码仓库创建一个目录，稍后再连接到远程站点。为此，你需要在 `git init` 一个空目录中，或者已经包含你想稍后添加的材料的目录中执行。这里，我们从一个空目录开始。

```
Cmd >> mkdir newrepoCmd >> cd newrepoCmd >> git initOut >>
Initialized empty Git repository in↪/users/demo/git/newrepo/.git/Cmd >> ls -aOut >>. 创建一个目录，并将其变成一个 .. .git 代码仓库 Cmd >> git statusOut >>On branch masterNo commits yet
nothing to commit (create/copy files and use "git add"↪to track)
```

这种方法相较于克隆一个空的代码仓库的缺点是，你现在必须将你的目录连接到远程代码仓库。详见第 5.6 节。

5.3.3 Main vs master

过去默认分支（是的，我知道，我们还没讨论分支）被称为 “master”。术语发生变化后，首选名称现在是 “main”。像 *github* 和 *gitlab* 这样的网站可能已经默认创建了这个名称；截至 2022 年撰写本文时，git 软件尚未这样做。

重命名分支是可能的。你可以使用 `git status` 来查看你当前所在的分支。

```
Cmd >> git status
Out >>
On branch master
No commits yet
nothing to commit (create/copy files and use "git add"
    ↪to track)
```

查看 main 分支是什么

Move the current branch:

```
Cmd >> git branch -m mainCmd >> git statusOut >>On branch main
```

重命名分支

```
No commits yetnothing to commit (create/copy files and use "git add"
↪to track)
```

为了保险起见，再用 `git status` 检查一次名称。

```
Cmd >> git branch -m main
Cmd >> git status
Out >>
On branch main
No commits yet
nothing to commit (create/copy files and use "git add"
    ↪to track)
```

检查分支名称

5.4 Adding and changing files

5.4.1 创建新文件

如果你创建了一个文件，它不会自动成为你的代码仓库的一部分。这需要一系列步骤。如果你创建了一个新文件，并运行 `git status`，你会看到该文件被列为 “ 未跟踪 ” 。

```
Cmd >> echo foo > firstfile
Cmd >> git status
Out >>
On branch main
No commits yet
Untracked files:
(use "git add <file>..." to include in what will be
    ↪committed)
firstfile
nothing added to commit but untracked files present
    ↪(use "git add" to track)
```

使用常用工具如编辑器创建文件。这里我们使用 `touch` 作为快捷方式。该文件最初将是未跟踪的。

你需要对你的文件执行 `git add`，告诉 git 该文件属于代码仓库。（你可以添加单个文件，也可以使用通配符添加多个文件。）然而，这实际上并没有添加文件：它只是将文件移动到暂存区。状态现在显示这是一个待提交的更改。

```
Cmd >> git add firstfile
Cmd >> git status
Out >>
On branch main
No commits yet
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
new file:   firstfile
```

将文件添加到本地代码仓库

使用 `git commit` 将这些更改添加到代码仓库。

```
Cmd >> git commit -m "adding first file"
Out >>
[main (root-commit) f968ac6] adding first file
1 file changed,  1 insertion(+)
create mode 100644 firstfileCmd >> git status
Out >>On branch mainnothing to commit,
 working tree clean
```

Commit these changes

5.4.2 代码仓库中文件的更改

让我们研究将更改添加到代码仓库中文件的过程。

```
Cmd >> echo bar >> firstfileCmd >> cat firstfileOut >>foobarCmd >> git statusOut >>
On branch main
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)modified:   firstfile

no changes added to commit (use "git add" and/or "git commit -a")
```

对被跟踪的文件进行更改。你通常会使用编辑器，但我们使用 `cat` 来追加。

如果你需要检查所做的更改，`git diff` 在该文件上将告诉你编辑过但尚未添加或提交的文件与之前提交版本之间的差异。

```
Cmd >> git diff firstfileOut >>
diff --git a/firstfile b/firstfile
index 257cc56..3bd1f0e 100644
--- a/firstfile+++ b/firstfile
@@ -1 +1,2 @@foo+bar
```

See what the changes were wrt the previously commit version.

你现在需要在该文件上重复 `git add` 和 `git commit`。

```
Cmd >> git add firstfile
Cmd >> git commit -m "changes to first file"
Out >>[main b1edf77] changes to first file
1 file changed,  1 insertion(+)
Cmd >> git statusOut >>On branch main
nothing to commit,  working tree clean
```

将更改提交到本地代码仓库。

更改现在在你的本地代码仓库中；你需要 `git push` 来更新上游代码仓库；参见章节 5.6。

执行 `git log` 将为你提供代码仓库的历史记录，列出提交编号，以及你在那些提交中输入的消息。

```
Cmd >> git log
Out >>
commit b1edf778c17b7c7e6cb1a8ac73fa9b61464eba14
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:40 2022 -0600
changes to first file
commit f968ac6c05dd877db84705a4dcdadbc0bed2c535
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:39 2022 -0600
adding first file
```

获取迄今为止所有提交的日志。

5.5 撤销更改

撤销有不同的层级，取决于你是否已经添加或提交了这些更改。

5.5.1 撤销未提交的更改

场景：你编辑了仓库中的一个文件，想要撤销该更改，但你还没有提交这次更改。

首先，执行 `git diff` 以确认：

```
Cmd >> echo bar >> firstfileCmd >> cat firstfile
Out >>foobarbarCmd >> git statusOut >>On branch main
Changes not staged for commit:
(use "git add <file>..." to update what will be
↪committed)
(use "git restore <file>..." to discard changes in
↪working directory)modified:   firstfile
no changes added to commit (use "git add" and/or "git
↪commit -a")Cmd >> git diff firstfileOut >>
diff --git a/firstfile b/firstfile
index 3bd1f0e..58ba28e 100644--- a/firstfile
+++ b/firstfile@@ -1, 2 +1, 3 @@foobar+bar
```

做出令人遗憾的更改。

Doing `git checkout` on that file gets the last committed version and puts it back in your working directory.

```
Cmd >> git checkout firstfileOut >>
Updated 1 path from the index
Cmd >> cat firstfileOut >>foobar
Cmd >> git statusOut >>On branch main
nothing to commit, working tree clean
```

恢复先前提交的版本。

5.5.2 从之前的提交中恢复文件

一个更复杂的场景是你已经提交了更改。然后你需要找到提交 ID。

你可以使用 `git log` 来获取所有提交的 ID。如果你想回滚到很久以前的版本，这很有用。然而，如果你只想回滚到最后一次提交，使用 `git show HEAD` 来获取最后一次提交的描述。

```
Cmd >> git logOut >>
commit e411fad261fd82eb93c328c44978699e946abc0d
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:42 2022 -0600
changes to first file
commit 63d6ad16beb4e2d12574fb238c29e8ba11fc6732
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:41 2022 -0600
adding first fileCmd >> git show HEADOut >>
commit e411fad261fd82eb93c328c44978699e946abc0d
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:42 2022 -0600
changes to first file
diff --git a/firstfile b/firstfile
index 257cc56..3bd1f0e 100644--- a/firstfile
+++ b/firstfile@@ -1 +1, 2 @@foo+bar
```

找到您想要回滚的提交 ID。

现在执行：

```
git checkout sdf234987238947 -- myfile myotherfile
```

5.5.3 撤销提交

如上，找到提交编号。

然后执行 `git revert sdlksdfkl2343`（使用正确的 id）。这通常会打开一个编辑器，供你留下评论；你可以使用 `--no-edit` 选项来防止这种情况。

```
Cmd >> git revert $commit --no-editOut >>
[main 3dca724] Revert "changes to first file"
Date: Sat Jan 29 14:14:42 2022 -0600
1 file changed, 1 deletion(-)
```

使用 ’git revert’ 来回滚。

这将把文件恢复到上一次 add 和 commit 之前的状态，通常会使代码仓库回到该 commit 之前的状态。

```
Cmd >> cat firstfileOut >>foo
Cmd >> git statusOut >>On branch main
nothing to commit, working tree clean
```

See that we have indeed undone the commit.

但是，日志会显示你已经还原了某个 commit .

```
Cmd >> git log
Out >>
commit 3dca724a1902e8a5e3dba007c325542c6753a424
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:42 2022 -0600
Revert "changes to first file"

This reverts commit
    ↪e411fad261fd82eb93c328c44978699e946abc0d.
commit e411fad261fd82eb93c328c44978699e946abc0d
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:42 2022 -0600
changes to first file
commit 63d6ad16beb4e2d12574fb238c29e8ba11fc6732
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:41 2022 -0600
adding first file
```

但日志中会有一条记录。

`git reset` 命令也可以用于各种类型的撤销。

5.6 远程代码仓库与协作

你一直用 `git commit` 添加文件和更改的代码仓库是本地代码仓库。这对于跟踪更改和在需要时恢复非常有用，但使用源代码控制的一个主要原因是与他人协作。为此，你需要一个远程代码仓库。这涉及一组命令

```
git remote [ other keywords ] [ arguments ]
```

我们有一些更改，已通过 `git add` 和 `git commit` 添加到本地代码仓库

```
Cmd >> git add newfile && git commit -m "adding first
↪file"Out >>[main 8ce1de4] adding first file
1 file changed, 1 insertion(+)
create mode 100644 newfile
```

已提交更改。

如果代码仓库是用 `git init` 创建的，我们需要用以下命令将其连接到某个远程代码仓库

```
git remote add servername url
```

通常，远程名称按惯例是 *origin*，所以你经常会看到 :

```
git remote add origin url
```

如果你想查看你的远程是什么，执行

```
git remote -v
```

```
Cmd >> git remote add mainserver
    ↪git@github.com:TACC/tinker.git
Cmd >> git remote -v
Out >>
mainserver      git@github.com:TACC/tinker.git (fetch)
mainserver      git@github.com:TACC/tinker.git (push)
```

将本地代码仓库连接到远程代码仓库。

最后，你可以将 `git push` 提交的更改推送到这个远程。Git 并不会将所有内容都推送到这里：因为你本地可以有多个分支，远程也可以有多个 *upstream*，你需要最初指定两者：

```
git push -u servername branchname
```

```
Cmd >> git push -u mainserver main
Out >>
To github.com:TACC/tinker.git
8333bc1..8ce1de4  main -> main
Branch 'main' set up to track remote branch 'main'
    ↪from 'mainserver'.
```

推送更改。

5.6.1 更改传输方式

您可能已经使用

`git clone https://....` 创建了克隆

但是当您 `git push` 第一次使用时，会遇到一些与权限相关的错误。

Do

```
git remote -v
# output: origin  https://username@bitbucket.org/username/reponame.git
git remote set-url origin git@bitbucket.org:username/reponame.git
```

5.6.2 在同一代码仓库上的协作

让我们看看一个代码仓库的一个克隆中的更改如何传播到另一个克隆。这可能是因为不止一个人在一个项目上工作，或者因为一个人在多台机器上工作。

我们在目录 *person1* 中创建一个本地代码仓库。

```
Cmd >> git clone git@github.com:TACC/tinker.git person1Out >> 用户1 进行克隆。
Cloning into 'person1'...
```

在 *person2* 中创建另一个克隆。通常，克隆的代码仓库会是两个用户账户，或者是一个用户在两台机器上的账户。

```
Cmd >> git clone git@github.com:TACC/tinker.git person2
Out >>                                                    Person 2 makes a clone.
Cloning into 'person2'...
```

现在第一个用户创建一个文件，添加，提交，并推送它。（这当然需要设置上游，但由于我们执行了 `git clone`，这会自动完成。）

```
Cmd >> ( cd person1 && echo 123 >> p1 && git add p1 &&
↪git commit -m "add p1" && git push )Out >>
[main 6f6b126] add p11 file changed, 1 insertion(+)
create mode 100644 p1To github.com:TACC/tinker.git
8863559..6f6b126  main -> main
```

Person 1 adds a file and pushes it.

第二个用户现在执行

```
git pull
```

以获取这些更改。同样，因为我们通过 `git clone` 创建了本地代码仓库，所以很清楚拉取来自哪里。拉取信息会告诉我们创建了哪些新文件，或者有多少其他文件被更改。

```
Cmd >> ( cd person2 && git pull )Out >>
From github.com:TACC/tinker
8863559..6f6b126  main       -> origin/main
Updating 8863559..6f6b126Fast-forward
p1 | 1 +1 file changed, 1 insertion(+)
create mode 100644 p1
```

Person 2 拉取，获取新文件。

5.6.3 合并更改

如果你与他人合作，或者即使你单独在一个项目上工作，但使用多台机器，也可能会出现单个文件有多个更改的情况。也就是说，两个本地仓库都有已提交的更改，并且现在都在推送到同一个远程仓库。

在下面的脚本中，我们从前一个示例的相同情况开始，这里我们有两个本地仓库，作为两个用户或两台不同机器的代表。

我们有一个包含四行的文件。

```
Cmd >> cat person1/fourlines
Out >>1 2 3 4
```

We have a four line file.

第一个用户对第一行进行了编辑；我们确认文件的状态；

```
Cmd >> ( cd person1 && sed -i -e '1s/1/one/' fourlines
↪&& cat fourlines )Out >>one2 3 4
```

用户 1 进行了更改。

该用户推送了更改。

```
Cmd >> ( cd person1 && git add fourlines && git commit
↪-m "edit line one" && git push )Out >>
[main 6767e3f] edit line one1 file changed,
 1 insertion(+), 1 deletion(-)
To github.com:TACC/tinker.git
fdd70b7..6767e3f  main -> main
```

用户 1 推送了更改。

另一个用户也进行了更改，但在第 4 行，因此没有冲突；

```
Cmd >> ( cd person2 && sed -i -e '4s/4/four/'
↪fourlines && cat fourlines )Out >>1 2 3four
```

Person 2 makes a different change to the same file.

此更改与 `git add` 和 `git commit` 一起添加，但我们在推送时更为谨慎：首先我们使用拉取其他人所做的任何更改

```
git pull --no-edit
git push
```

```
Cmd >> ( cd person2 && git add fourlines && git commit
↪-m "edit line four" && git pull --no-edit && git
↪push )Out >>[main 27fb2b2] edit line four
1 file changed, 1 insertion(+), 1 deletion(-)
From github.com:TACC/tinker
fdd70b7..6767e3f  main       -> origin/main
Auto-merging fourlines
Merge made by the 'recursive' strategy.
fourlines | 2 +-1 file changed, 1 insertion(+),
 1 deletion(-)To github.com:TACC/tinker.git
6767e3f..62bd424  main -> main
```

此更改无冲突，我们可以 pull/push。

现在如果第一个用户执行 pull，他们会看到所有合并的更改。

```
Cmd >> ( cd person1 && git pull && cat fourlines )
Out >>From github.com:TACC/tinker
6767e3f..62bd424  main       -> origin/main
Updating 6767e3f..62bd424Fast-forward
fourlines | 2 +-1 file changed,  1 insertion(+),
 1 deletion(-)one2 3four
```

Person 1 拉取以获取所有更改。

5.6.4 冲突的更改

git 报告冲突可能有多种原因。

- 你正试图从另一位开发者那里拉取更改，但你自己也有尚未提交的更改。如果你不想提交你的更改（也许你还在忙着编辑），你可以使用 `git stash` 将你的编辑暂时搁置。你以后可以用 `git stash pop` 取回它们，或者决定用 `git stash drop` 彻底放弃它们。
- 你的更改与试图从另一位开发者那里拉取的更改，或者从你自己另一台机器上的更改之间存在冲突。这就是我们这里要讨论的情况。
- 非常类似的情况，还有你尝试合并的两个分支之间也可能存在冲突。我们将在第 5.7.1 节中讨论这个问题。

和之前一样，我们有目录 *person1* 和 *person2*，其中包含代码仓库的独立克隆。我们有一个四行长的文件，但与上面不同的是，我们现在进行的编辑彼此太接近，git 的自动合并无法处理它们。

```
Cmd >> cat person1/fourlines
Out >>1 2 3 4
```

The original file.

现在开发者 1 在第 1 行做了一个更改。

```
Cmd >> ( cd person1 && sed -i -e '1s/1/one/' fourlines
    ↪&& git add fourlines && git commit -m "edit line
    ↪one" && git push )
Out >>[main a10216d] edit line one
1 file changed, 1 insertion(+), 1 deletion(-)
To github.com:TACC/tinker.git
4955e50..a10216d  main -> main
```

抱歉使用了一些脚本技巧，我们使用 *sed* 的一次编辑，将第 1 行的 `1` 改为 *one*。我们添加、提交并推送了此更改。

与此同时，开发者 2 对原始文件进行了另一次更改。此更改可以无问题地添加并提交到本地代码仓库。

```
Cmd >> ( cd person2 && sed -i -e '2s/2/two/' fourlines
↪&& cat fourlines  && git add fourlines && git
↪commit -m "edit line two" )Out >>1two3 4
[main c9b6ded] edit line two1 file changed,
 1 insertion(+), 1 deletion(-)
```

Change the `2` on line two to *two*. We add and commit this to the local repository.

然而，如果我们尝试将此更改 `git push` 到远程代码仓库，会出现远程代码仓库领先于本地代码仓库的错误。因此，我们首先拉取远程代码仓库的状态。在上一节中，这导致了自动合并；但这里不是这样。

```
Cmd >> ( cd person2 && git pull --no-edit || echo )
Out >>
From github.com:TACC/tinker
4955e50..a10216d  main       -> origin/main
Auto-merging fourlines
CONFLICT (content): Merge conflict in fourlines
Automatic merge failed; fix conflicts and then commit
    ↪the result.
```

`git pull` 调用结果是一条消息，表明自动合并失败，指出了哪个文件出了问题。

Y你现在可以手动编辑该文件，或者使用某些合并工具　.

```
Cmd >> ( cd person2 && cat fourlines )Out >>
<<<<<<< HEAD1two=======one2
>>>>>>> a10216da358649df80aaaeb94f1ceef909c2ed83
3 4
```

在尖括号标记的行之间，你首先会看到 `HEAD`，即本地状态，接着是拉取的远程状态。编辑此文件，提交合并并推送。

5.6.5 Pull requests

前面的章节描述了你对拥有写权限的代码仓库的协作方式。对于大型项目，你可能没有这种权限：项目所有者可能希望采用一种更安全的机制，你以 *pull request* 的形式提交更改以供审批，他们将把你的更改合并到代码仓库中。

This involves the following steps:

1. 你对代码仓库进行 *fork*；
2. 克隆这个 fork；
3. 在你的更改上创建一个分支；

5.7 分支

通过 *branch*，你可以保留项目中所有文件的完全独立版本。

最初我们在 *main* 分支上有一个文件。

```
Cmd >> cat firstfileOut >>foo
Cmd >> git statusOut >>On branch main
nothing to commit, working tree clean
```

我们有一个文件，已经提交完毕。

我们创建了一个名为 *dev* 的新分支并切换到它 t

```
git branch dev
git checkout dev
```

这最初具有相同的内容。

```
Cmd >> git branch dev && git branch -a
Out >>
dev
* main
Cmd >> git checkout dev && git branch -a
Out >>
Switched to branch 'dev'
* dev
main
```

创建一个开发分支。

我们进行更改，并将它们提交到当前分支。

```
Cmd >> cat firstfileOut >>foo
Cmd >> echo bar > firstfile && cat firstfileOut >>bar
Cmd >> git statusOut >>On branch dev
Changes not staged for commit:
(use "git add <file>..." to update what will be
↪committed)
(use "git restore <file>..." to discard changes in
↪working directory)modified:   firstfile
no changes added to commit (use "git add" and/or "git
↪commit -a")
Cmd >> git add firstfile && git commit -m "dev changes"
Out >>[dev b07cd2e] dev changes1 file changed,
 1 insertion(+),  1 deletion(-)
```

进行更改并将其提交到 dev 分支。

如果我们切换回 `main` 分支，一切都和我们创建 `dev` 分支时一样。

```
Cmd >> git checkout main && cat firstfile && git status
Out >>Switched to branch 'main'fooOn branch main
nothing to commit, working tree clean
```

另一个分支仍然保持不变。

The first time you try to push a new branch you need to establish it upstream:

```
git push --set-upstream origin mynewbranch
```

We can inspect differences between branches with

```
  git diff branch1 branch2
```

```
Cmd >> git diff main dev
Out >>
diff --git a/firstfile b/firstfile
index 257cc56..5716ca5 100644
--- a/firstfile
+++ b/firstfile
@@ -1 +1 @@
-foo
+bar
```

我们可以检查分支之间的差异。

5.7.1 分支合并

拥有分支的一个目的就是在你在某个分支上完成一些开发后，将它们合并。

我们从之前的四行文件开始。

```
Cmd >> cat fourlinesOut >>1234
Cmd >> git statusOut >>On branch main
nothing to commit, working tree clean
```

主分支是最新的。

同样如前，我们有一个 *dev* 分支包含这些内容 .

我们切换回 *main* 分支并进行更改。这在 *dev* 分支中不可见。

```
Cmd >> git checkout mainOut >>Switched to branch 'main'
Cmd >> sed -i -e '1s/1/one/' fourlines && cat fourlinesOut >>one2
```

在第1行，将 1 改为 *one*。3 4*Cmd* >> `git add fourlines && git commit -m "edit line 1"`*Out* >>

```
[main c51d4ff] edit line 11 file changed, 1 insertion(+), 1 deletion(-)
```

我们切换到 *dev* 分支并创建另一个文件。主分支中的更改确实不在这里。

```
Cmd >> git checkout devOut >>Switched to branch 'dev'
Cmd >> sed -i -e '4s/4/four/' fourlines && cat
↪fourlinesOut >>1 2 3four
Cmd >> git add fourlines && git commit -m "edit line 4
```

在第 4 行，将 4 改为 *four*。此更改距离另一次更改足够远，因此 " 不应有冲突。

```
Out >>[dev dbb0c03] edit line 41 file changed,
 1 insertion(+), 1 deletion(-)
```

切换回 *main* 分支，我们使用

```
git merge dev
```

将 *dev* 的更改合并到 *main* 中。

```
Cmd >> git checkout mainOut >>
Switched to branch 'main'
Cmd >> git merge devOut >>
Auto-merging fourlines
Merge made by the 'recursive' strategy.
fourlines | 2 +-1 file changed,
 1 insertion(+),  1 deletion(-)
Cmd >> cat fourlinesOut >>one2 3four
```

使用 `git merge` 将 dev 分支合并到 main 分支。注意 ‘auto-merge’ 消息，并确认文件的两个更改都存在。

如果两个开发者在同一行或相邻行进行了更改，git 将无法合并，您需要像第 5.6.4 节中那样编辑文件。

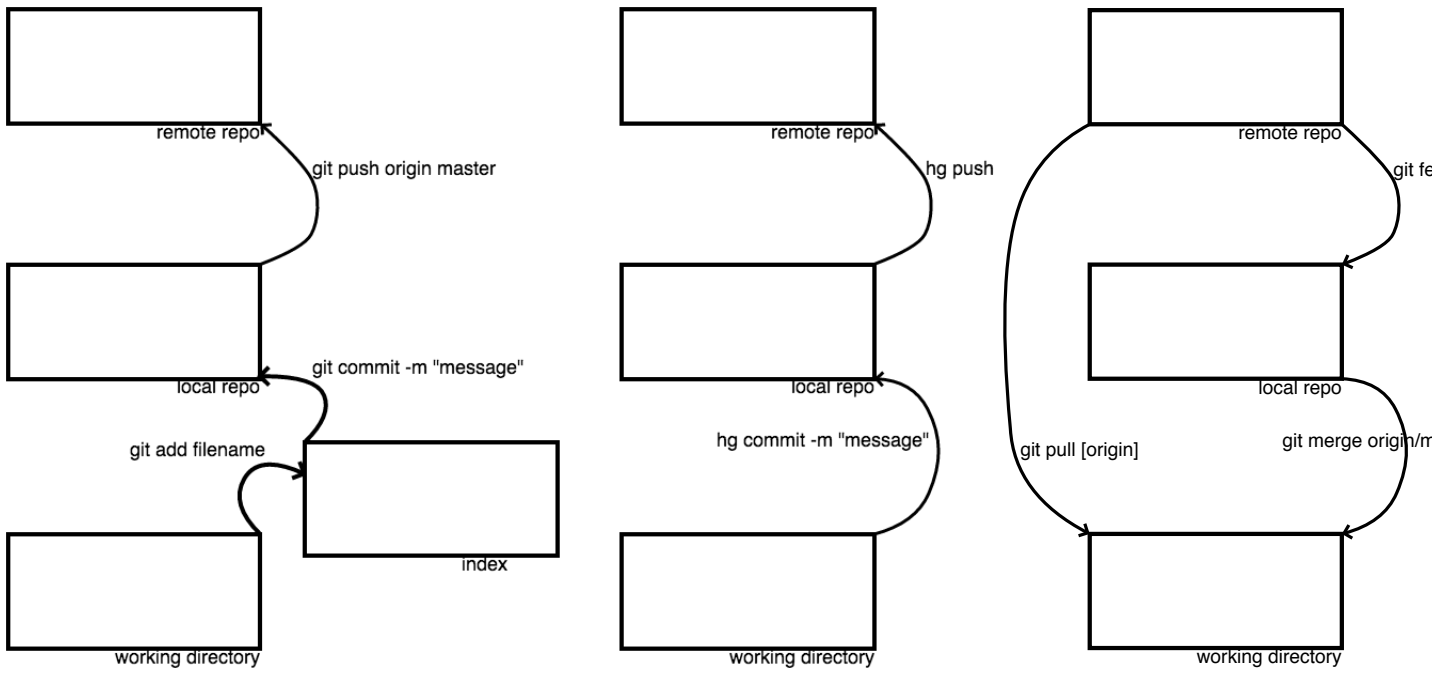


图 5.1：将本地更改添加到远程代码仓库（左）；获取对远程所做的更改 e
代码仓库 (right).

5.8 发布

At certain point in your development process you may want to mark the current state of the repository as 'finished'. You can do this by

1. 将 `tag` 附加到代码仓库的状态，或 2. 创建一个 `archive`：一个已去除代码仓库信息的发布版本。

5.8.1 Tags

A tag is a marked state of the repository. There are two types of tags:

1. 轻量级标签不过是提交的同义词：

```
git tag v0.09
```

2. 带注释的标签，携带信息消息 age:

```
git tag -a v0.1 "base classes finished"
```

你用 `git tag` 列出所有标签，用 `git show v0.1` 获取标签信息，用

```
git push origin v0.1
```

你可以用以下命令检索代码仓库的标签状态：

```
git checkout v0.1
```

但要注意，你现在所做的更改无法推送到任何地方：这是一个 “ 分离的 HEAD” 状态。如果你想在标签状态下修复错误，可以基于该标签创建一个分支：

```
git checkout -b version0.2 v0.1
```

5.8.2 归档，发布

如果您想制作一个可下载且不依赖于 git 软件的已发布版本，请使用 `archive` 命令：

```
git archive master --format=tgz --prefix=MyProject-v1 -o MyProject-v1.tgz
```

第 6 章

稠密线性代数：BLAS，LAPACK，SCALAPACK

本节将讨论用于稠密线性代数运算的库。

稠密线性代数，即在线性代数中处理以二维数组形式存储的矩阵（与稀疏线性代数相对；参见 HPC 书籍，第 5.4 节，以及 PETSc 教程并行编程书籍，第 *III* 部分）已经标准化了相当长的时间。基本操作由三个级别的基本线性代数子程序（*BLAS*）定义：

- 第 1 级定义了以单层循环为特征的向量操作 [13]。
- 第 2 级定义了矩阵 - 向量操作，包括显式的矩阵 - 向量乘积和隐式的三角系统求解 [7]。
- Level 3 定义了矩阵 - 矩阵操作，最著名的是矩阵 - 矩阵乘积 [6]。

‘BLAS’ 这个名称暗示了一定程度的通用性，但最初的作者明确指出 [13] 这些子程序仅涵盖密集线性代数。尝试标准化稀疏操作从未取得同等成功。

基于这些构建模块，已经构建了处理更复杂问题的库，例如求解线性系统，或计算特征值或奇异值。*Linpack*[1] 和 *Eispack* 是最早将这些操作形式化的库，分别使用了 Blas Level 1 和 Blas Level 2。后来的发展，*Lapack* 使用了 Blas Level 3 的分块操作。正如你在 HPC book，第 1.6.1 节中看到的，这对于在基于缓存的 CPU 上获得高性能是必要的。

备注 15 *BLAS* 的参考实现 `https://netlib.org/blas/index.html` 不会在任何编译器上提供良好的性能；大多数平台都有厂商优化的实现，例如 Intel 的 MKL 库。

随着并行计算机的出现，出现了几个扩展 Lapack 功能以支持分布式计算的项目，最著名的是 *Scalapack* [4, 2], *PLapack* [23, 22]，以及最近的 Elemental [19]。这些包比 Lapack 更难使用，因为需要二维循环分布；详见 HPC book，第 7.2.3 节和 HPC book，第 7.3.2 节。这里我们不做详细介绍。

1. The linear system solver from this package later became the *Linpack benchmark*; see section HPC book, section 2.11.5.

6.1 一些一般性说明

6.1.1 The Fortran heritage

最初的 BLAS 例程是用 Fortran 编写的，参考实现仍然是 Fortran。因此，在本教程中您将首先看到 Fortran 中的例程定义。可以从 C/C++ 程序中调用 Fortran 例程：

通常需要在 Fortran 名称后添加下划线；
您需要在源代码中包含一个原型文件，例如 `mkl.h`；
- 每个参数都需要是 ‘star’ 参数，因此不能传递字面常量：你需要传递变量的地址。

- 你需要创建一个列主序矩阵。

还有 C/C++ 接口：

- C 例程名称通过在原始名称前加上 `cblas_` 来形成；例如 `dasum` 变成 `cblas_dasum`。

- Fortran 字符参数已被枚举常量取代，例如 `CblasNoT`

Cblas 接口既支持行优先存储，也支持列优先存储。
- 数组索引是从 1 开始的，而不是从 0 开始的；这主要在错误信息和指定 pivotindices 时显现出来。

6.1.2 例程命名

Routines conform to a general naming scheme: `XYYZZZ` where
X 精度： `S`,`D`,`C`,`Z` 分别代表单精度和双精度，单精度复数和双精度复数。**YY** 存储方案：通用矩形、三角形、带状。**ZZZ** 操作。详见手册列表。

6.1.3 数据格式

Lapack 和 Blas 使用多种数据格式，包括
GE 通用矩阵：以二维形式存储为 `A(LDA,*)` **SY/HE** 对称 / 厄米：通用存储； `UPLO` 参数用于指示上三角或下三角（例如 `SPOTRF`） **GB/SB/HB** 通用 / 对称 / 厄米带状；这些格式使用列主序存储；在 `SGBTRF` 中由于选主元需要额外分配
PB 对称或厄米正定带状；在 `SPDTRF` 中无额外分配

6.1.4 Lapack 操作

对于 Lapack，我们可以将例程进一步划分为三个层次的组织结构：

- 驱动程序。这些是用于解决线性系统或计算奇异值分解（SVD）等问题的强大顶层例程。有简单驱动和专家驱动；专家驱动具有更高的数值复杂性。

- 计算例程。这些是构建驱动程序的例程。用户有时可能会单独调用它们。
- 辅助例程。

专家驱动程序名称以’X’ 结尾。

- 线性方程组的简单驱动程序：`-SV`（例如，`DGESV`）求解 $AX = B$，用 LU（带选主元）覆盖 A，用 X 覆盖 B。专家驱动程序：`-SVX` 还包括转置求解、条件估计、改进、均衡
- 最小二乘问题。驱动程序：`xGELS` 在满秩假设下使用 QR 或 LQ`xGELSY` “完全正交分解”`xGELSS` 使用 SVD`xGELSD` 使用分治 SVD（更快，但比 `xGELSS` 需要更多工作空间）另外：LSE & GLM 线性等式约束 & 广义线性模型
- 特征值例程。对称 / 厄米：`xSY` 或 `xHE`（也有 `SP`, `SB`, `ST`）简单驱动程序 `-EV` 专家驱动程序 `-EVX` 分治 `-EVD` 相对稳健表示 `-EVR` 通用（仅 `xGE`） Schur 分解 `-ES` 和 `-ESX` 特征值 `-EV` 和 `-EVX`SVD（仅 `xGE`）简单驱动程序 `-SVD` 分治 `SDD` 广义对称（`SY` 和 `HE`； `SP`, `SB`）简单驱动程序 `GV` 专家 `GVX` 分治 `GVD` 非对称：Schur：简单 `GGES`，专家 `GGESX` 特征值：简单 `GGEV`，专家 `GGEVX`svd： `GGSVD`

6.2 BLAS 矩阵存储

关于矩阵在 BLAS 和 LAPACK 中的存储方式，有几点需要注意 [2]:

6.2.1 数组索引

由于这些库起源于 Fortran 环境，因此它们使用基于 1 的索引。使用 C/C++ 等语言的用户只有在例程使用索引数组时才会受到影响，例如 LU 分解中主元的位置。

2. 我们这里不讨论带状存储。

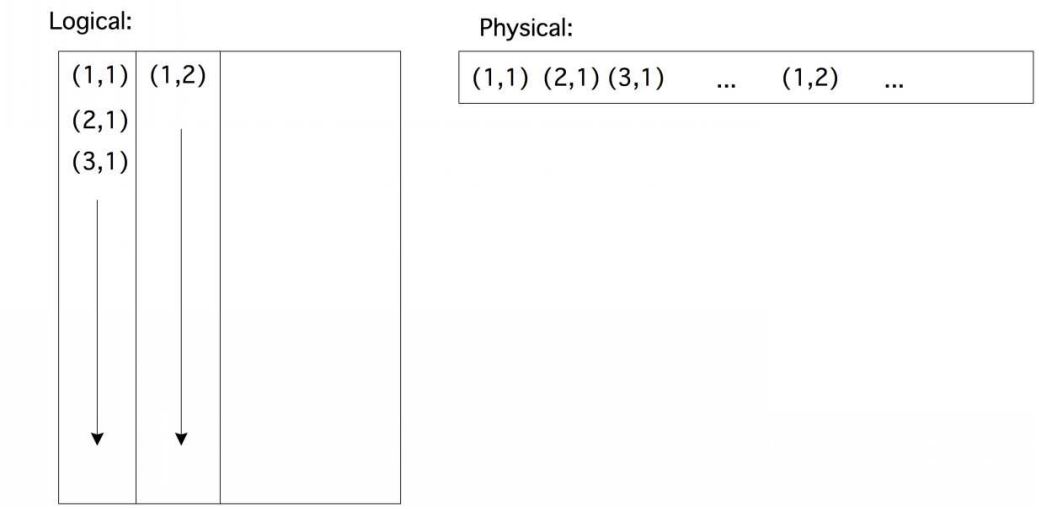


图 6.1：Fortran 中数组的列主存储。

6.2.2 Fortran 列主序

由于计算机内存是一维的，因此需要将二维矩阵坐标转换为内存位置。 *Fortran* 语言使用 列主序 存储，即同一列的元素连续存储；见图 6.1。这也被非正式地描述为 “ 最左边的索引变化最快 ”。

另一方面，C 语言中的数组是按 行主序 排列的。

6.2.3 子矩阵和 `LDA` 参数

使用上述存储方案，可以清楚地知道如何将一个 $m \times n$ 矩阵存储在 mn 内存位置中。然而，许多情况下，软件需要访问另一个更大矩阵的子块。正如你在图 6.2 中看到的，这样的子块在内存中不再是连续的。描述这种情况的方法是引入第三个参数，除了 `M,N` 之外：我们让 `LDA` 成为 `A` 的 “ 主维度 ”，即外围数组分配的第一个维度。这在图 6.3 中有所说明。要将子块传递给例程，你可以这样指定它

```
call routine( A(3,2), /* M= */ 2, /* N= */ 3, /* LDA= */ Mbig, ... )
```

6.3 性能问题

BLAS 和 LAPACK 例程集合是一个事实上的标准：API 是固定的，但实现不是。你可以在 *netlib* 网站（`netlib.org`）上找到参考实现，但这些实现的性能会非常低。

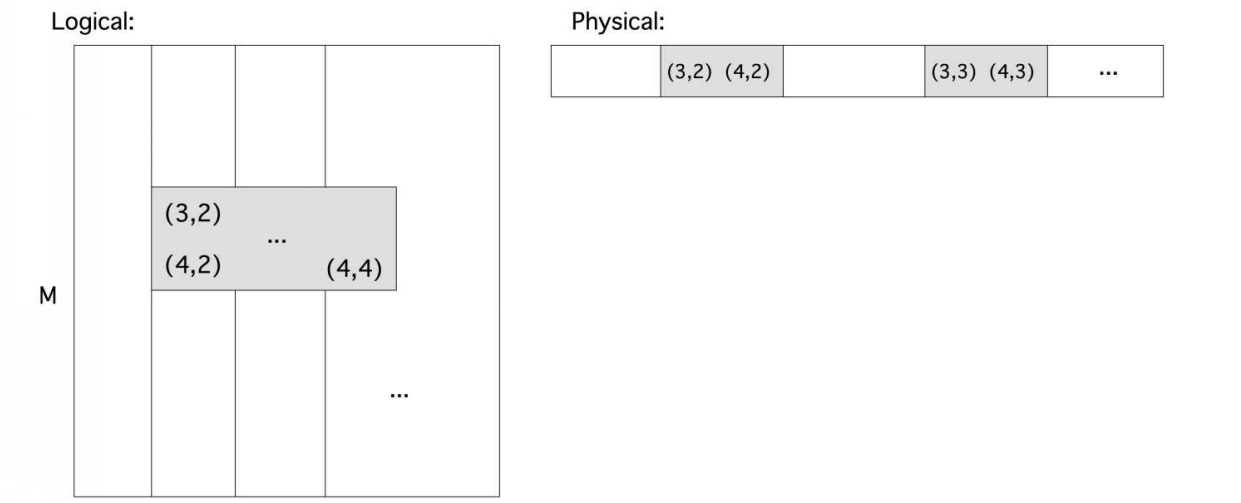


图 6.2：一个较大矩阵中的子块。

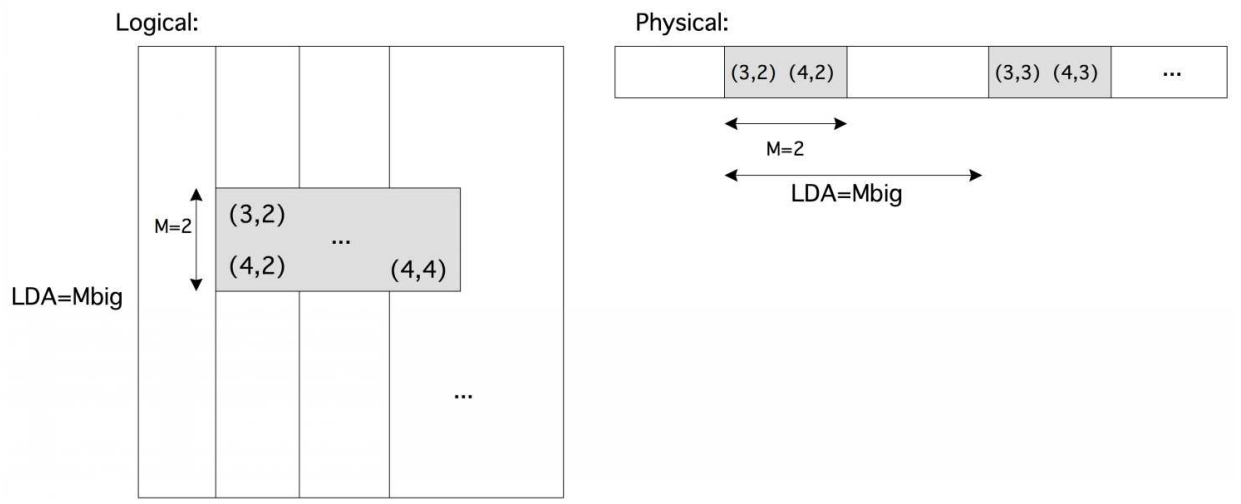


Figure 6.3: A subblock out of a larger matrix, using `LDA`.

另一方面，许多 LAPACK 例程可以基于矩阵 - 矩阵乘积（ BLAS 例程 `gemm` ），你在 HPC 书籍第 7.4.1 节中见过，它有潜力达到峰值性能的很大一部分。为了实现这一点，你应该使用优化版本，例如

- *MKL*，英特尔数学核心库；
- OpenBlas (`http://www.openblas.net/`)，原始 *Goto BLAS* 的开源版本；或者
- *blis* (`https://code.google.com/p/blis/`)，一个 BLAS 替代和扩展项目。

6.4 一些简单的例子

让我们来看一些简单的例子。

例程 `xscal` 就地缩放一个向量。

```
! Fortran
subroutine dscal(integer N, double precision DA,
double precision, dimension(*) DX, integer INCX )// C
void cblas_dscal (const MKL_INT n, const double a,
double *x, const MKL_INT incx);
```

一个简单的例子：

```
// example1.F90
  do i=1,n
     xarray(i) = 1.d0
  end do
  call dscal(n,scale,xarray,1)
  do i=1,n
     if (.not.assert_equal( xarray(i),scale )) print *,"Error in index",i
  end do
```

C 语言中的相同写法：

```
// example1c.cxx
  xarray = new double[n]; yarray = new double[n];

  for (int i=0; i<n; i++)
    xarray[i] = 1.;
  cblas_dscal(n,scale,xarray,1);
  for (int i=0; i<n; i++)
    if (!assert_equal( xarray[i],scale ))
      printf("Error in index %d",i);
```

许多例程都有一个增量参数。对于 `xscale` 来说，这是最后一个参数：

```
// example2.F90
  integer :: inc=2
  call dscal(n/inc,scale,xarray,inc)
  do i=1,n
     if (mod(i,inc)==1) then
        if (.not.assert_equal( xarray(i),scale )) print *,"Error in index",i
     else
        if (.not.assert_equal( xarray(i),1.d0 )) print *,"Error in index",i
     end if
  end do
```

矩阵 - 向量乘积 `xgemv` 计算的是 $y \leftarrow \alpha Ax + \beta y$，而不是 $y \leftarrow Ax$。矩阵的规格由 `M,N` 大小参数和一个字符参数 `'N'` 指定，用以表示矩阵未转置。两个向量都有一个增量参数。

```
subroutine dgemv(character TRANS,
```

```
      integer M,integer N,double precision ALPHA,double precision,
       dimension(lda,*) A,integer LDA,double precision, dimension(*) X,
      integer INCX,double precision BETA,double precision, dimension(*) Y,
      integer INCY)
```

该例程的使用示例：

```
// example3.F90
  do j=1,n
     xarray(j) = 1.d0
     do i=1,m
        matrix(i,j) = 1.d0
     end do
  end do

  alpha = 1.d0; beta = 0.d0
  call dgemv( 'N',M,N, alpha,matrix,M, xarray,1, beta,yarray,1)
  do i=1,m
     if (.not.assert_equal( yarray(i),dble(n) )) &
          print *,"Error in index",i,":",yarray(i)
  end do
```

相同的 C 语言示例中有一个额外的参数，用于指示矩阵是以行主序还是列主序存储：

```
// example3c.cxxfor (int j=0; j<n; j++) {xarray[j] = 1.;
for (int i=0; i<m; i++)matrix[ i+j*m ] = 1.;}alpha = 1.; beta = 0.;
cblas_dgemv(CblasColMajor,CblasNoTrans,m,n, alpha,matrix,m, xarray,1, beta,
yarray,1);for (int i=0; i<m; i++)
if (!assert_equal( yarray[i],(double)n ))printf("Error in index %d",i);
```

第 7 章

Scientific Data Storage with HDF5

存储数据有许多方法，特别是以数组形式存在的数据。令人惊讶的是，很多人将数据存储在电子表格中，然后导出为带有逗号或制表符分隔符的 ascii 文件，并期望其他人（或他们自己编写的其他程序）能够再次读取这些文件。这样的过程在多个方面都是浪费的：

- 数字的 ascii 表示占用的空间远大于内部的二进制表示。理想情况下，你希望文件的大小能和内存中的表示一样紧凑。
- 转换为 ascii 格式及从 ascii 格式转换都很慢；这也可能导致精度丢失。

基于这些原因，理想的文件格式应基于二进制存储。对一个有用的文件格式还有一些额外的要求：

- 由于二进制存储在不同平台间可能存在差异，一个好的文件格式应当是平台无关的。例如，这可以防止在大端和小端存储之间产生混淆，以及 32 位与 64 位浮点数的约定差异。
- 应用数据可以是异构的，包含整数、字符和浮点数据。理想情况下，所有这些数据应当存储在一起。
- 应用数据也是有结构的。这种结构应当在存储形式中得到体现。
- 文件格式最好是自描述的。如果你在文件中存储了一个矩阵和一个右端向量，文件本身能告诉你哪些存储的数字是矩阵，哪些是向量，以及这些对象的大小，那该多好？

本教程将介绍满足这些要求的 HDF5 库。HDF5 是一个庞大且复杂的库，因此本教程只涉及基础内容。更多信息请参阅 http://www.hdfgroup.org/HDF5/。在进行本教程时，请保持浏览器打开，访问 http://www.hdfgroup.org/HDF5/doc/ 或 http://www.hdfgroup.org/HDF5/RM/RM_H5Front.html 以获取例程的准确语法。

7.1 Setup

如上所述，HDF5 是一种与机器无关且自我描述的文件格式。每个 HDF5 文件的结构类似于目录树，包含子目录和包含实际数据的叶节点。这意味着可以通过引用数据的名称而不是其在文件中的位置来查找文件中的数据。在此

本节您将学习编写程序以写入和读取 HDF5 文件。为了检查文件是否符合您的预期，您可以在命令行使用 `h5dump` 工具。

关于兼容性说明一下。HDF5 格式与旧版本 HDF4 不兼容，后者已不再开发。您仍可能因历史原因遇到使用 hdf4 的情况。本教程基于 HDF5 版本 1.6。当前版本 1.8 中一些接口有所变化；若要在 1.8 软件中使用 1.6 API，请在您的编译行中添加标志 `-DH5_USE_16_API` 。

7.1.1 编译

C 语言的包含文件:

```
#include <netcdf.h>
```

CMake for C:

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( NETCDF REQUIRED netcdf )
target_include_directories(
${PROJECTNAME} PUBLIC${NETCDF_INCLUDE_DIRS} )
target_link_libraries(${PROJECTNAME} PUBLIC
${NETCDF_LIBRARIES} )target_link_directories(
${PROJECTNAME} PUBLIC${NETCDF_LIBRARY_DIRS} )
target_link_libraries(
${PROJECTNAME} PUBLIC netcdf )
```

Include for Fortran:

```
use netcdf
```

CMake for Fortran:

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( NETCDFF REQUIRED netcdf-fortran )
pkg_check_modules( NETCDF REQUIRED netcdf )
target_include_directories(${PROJECTNAME} PUBLIC
${NETCDFF_INCLUDE_DIRS})target_link_libraries(
${PROJECTNAME} PUBLIC
${NETCDFF_LIBRARIES} ${NETCDF_LIBRARIES})
target_link_directories(${PROJECTNAME} PUBLIC
${NETCDFF_LIBRARY_DIRS} ${NETCDF_LIBRARY_DIRS})
```

```
target_link_libraries(
        ${PROJECTNAME} PUBLIC netcdf )
```

7.1.2 通用设计

许多 HDF5 例程都涉及创建对象：文件句柄、数据集中的成员，等等。其通用语法是

```
hid_t h_id;
h_id = H5Xsomething(...);
```

创建对象失败会通过返回负值的参数来指示，因此创建时检查返回值是个好主意

```
#include "hdf5.h"
#define H5REPORT(e) \
  {if (e<0) {printf("\nHDF5 error on line %d\n\n",__LINE__); \
   return e;}}
```

并将其用作：`#include "myh5defs.h"hid_t h_id;`

`h_id = H5Xsomething(...); H5REPORT(h_id);`

7.2 Creatinga file

首先，我们需要创建一个 HDF5 文件。

```
hid_t file_id;herr_t status;
file_id = H5Fcreate( filename, ... );...
status = H5Fclose(file_id);
```

该文件将作为多个数据项的容器，组织方式类似目录树。

Exercise. 通过编译并运行下面的 `create.c` 示例来创建一个 HDF5 文件。

Expected outcome. A file `file.h5` should be created.

Caveats. Be sure to add HDF5 include and library directories:
`cc -c create.c -I. -I/opt/local/include`
and
`cc -o create create.o -L/opt/local/lib -lhdf5`. The include and lib directories will be system dependent.

```
#include "myh5defs.h"
#define FILE "file.h5"

main() {

   hid_t       file_id;   /* file identifier */
   herr_t      status;

   /* Create a new file using default properties. */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
   H5REPORT(file_id);

   /* Terminate access to the file. */
   status = H5Fclose(file_id);
}
```

你可以在命令行上显示 hdf5 文件:

```
%% h5dump file.h5HDF5 "file.h5" {
GROUP "/" {}}
```

注意，空文件对应于将保存数据的目录树的根。

7.3 Datasets

接下来我们创建一个数据集，在此示例中是一个二维网格。为描述它，我们首先需要构造一个数据空间:

```
dims[0] = 4; dims[1] = 6;
dataspace_id = H5Screate_simple(2, dims, NULL);
dataset_id = H5Dcreate(file_id, "/dset", dataspace_id, .... );....
status = H5Dclose(dataset_id);status = H5Sclose(dataspace_id);
```

请注意，datasets 和 dataspaces 需要关闭，就像文件一样。

Exercise. 通过编译并运行下面的 `dataset.c` 代码来创建一个 dataset

预期结果。 这将创建一个文件 dset.h , 可以用 h5dump 显示。

```
#include "myh5defs.h"#define FILE "dset.h5"main() {hid_t       file_id,
 dataset_id, dataspace_id;  /* identifiers */hsize_t     dims[2];
herr_t      status;/* Create a new file using default properties. */
file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
/* Create the data space for the dataset. */dims[0] = 4;dims[1] = 6;
dataspace_id = H5Screate_simple(2, dims, NULL);
/* Create the dataset. */
dataset_id = H5Dcreate(file_id, "/dset", H5T_NATIVE_INT,dataspace_id,
 H5P_DEFAULT);/*H5T_STD_I32BE*/
/* End access to the dataset and release resources used by it. */
status = H5Dclose(dataset_id);
/* Terminate access to the data space. */
status = H5Sclose(dataspace_id);/* Close the file. */
status = H5Fclose(file_id);}
```

我们再次在线查看创建的文件：%% h5dump dset.h5

```
HDF5 "dset.h5" {GROUP "/" {DATASET "dset" {
DATATYPE  H5T_STD_I32BE
DATASPACE  SIMPLE { ( 4, 6 ) / ( 4, 6 ) }DATA {
(0,0): 0, 0, 0, 0, 0, 0,(1,0): 0, 0, 0, 0, 0, 0,(2,0): 0, 0, 0,
0, 0, 0,(3,0): 0, 0, 0, 0, 0, 0}}}
```

```
}
```

数据文件包含了您存储的数组大小等信息。不过，您可能还想添加相关的标量信息。例如，如果数组是程序的输出，您可以记录它是用哪个输入参数生成的。

```
parmspace = H5Screate(H5S_SCALAR);
parm_id = H5Dcreate
  (file_id,"/parm",H5T_NATIVE_INT,parmspace,H5P_DEFAULT);
```

练习。通过编译并运行下面的 `parmwrite.c` 代码，向 HDF5 文件添加一个标量数据空间。

预期结果。一个新文件 `wdset.h5` 被创建。`#define FILE "pdset.h5"`

```
main() {hid_t       file_id, dataset_id,
 dataspace_id;  /* identifiers */hid_t       parm_id,parmspace;
hsize_t     dims[2];herr_t      status;
/* Create a new file using default properties. */
file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
/* Create the data space for the dataset. */dims[0] = 4;dims[1] = 6;
dataspace_id = H5Screate_simple(2, dims, NULL);
/* Create the dataset. */dataset_id = H5Dcreate
(file_id, "/dset", H5T_STD_I32BE, dataspace_id, H5P_DEFAULT);
/* Add a descriptive parameter */parmspace = H5Screate(H5S_SCALAR);
parm_id = H5Dcreate
(file_id,"/parm",H5T_NATIVE_INT,parmspace,H5P_DEFAULT);
/* End access to the dataset and release resources used by it. */
status = H5Dclose(dataset_id);status = H5Dclose(parm_id);
/* Terminate access to the data space. */
status = H5Sclose(dataspace_id);status = H5Sclose(parmspace);
/* Close the file. */status = H5Fclose(file_id);}
```

```
%% h5dump wdset.h5HDF5 "wdset.h5" {GROUP "/" {
DATASET "dset" {DATATYPE  H5T_IEEE_F64LE
DATASPACE  SIMPLE { ( 4, 6 ) / ( 4, 6 ) }DATA {
(0,0): 0.5, 1.5, 2.5, 3.5, 4.5, 5.5, (1,0): 6.5,
7.5, 8.5, 9.5, 10.5, 11.5, (2,0): 12.5, 13.5, 1
4.5, 15.5, 16.5, 17.5, (3,0): 18.5, 19.5, 20.5,
21.5, 22.5, 23.5}}DATASET "parm" {
DATATYPE  H5T_STD_I32LEDATASPACE  SCALARDATA {
(0): 37}}}}
```

7.4 Writing the data

你创建的数据集在 hdf5 文件中分配了空间。现在你需要将实际数据放入其中。这是通过 `H5Dwrite` 调用完成的。

```
/* Write floating point data */
for (i=0; i<24; i++) data[i] = i+.5;
status = H5Dwrite
  (dataset,H5T_NATIVE_DOUBLE,H5S_ALL,H5S_ALL,H5P_DEFAULT,
   data);
/* write parameter value */
parm = 37;
status = H5Dwrite
  (parmset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,
   &parm);
```

```
#include "myh5defs.h"
#define FILE "wdset.h5"
```

```
main() {hid_t       file_id, dataset, dataspace;  /* identifiers */
hid_t       parmset,parmspace;hsize_t     dims[2];herr_t      status;
double data[24]; int i,parm;
/* Create a new file using default properties. */
file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
/* Create the dataset. */dims[0] = 4; dims[1] = 6;
dataspace = H5Screate_simple(2, dims, NULL);dataset = H5Dcreate
(file_id, "/dset", H5T_NATIVE_DOUBLE, dataspace, H5P_DEFAULT);
/* Add a descriptive parameter */parmspace = H5Screate(H5S_SCALAR);
parmset = H5Dcreate
(file_id,"/parm",H5T_NATIVE_INT,parmspace,H5P_DEFAULT);
```

```
   /* Write data to file */
   for (i=0; i<24; i++) data[i] = i+.5;
   status = H5Dwrite
     (dataset,H5T_NATIVE_DOUBLE,H5S_ALL,H5S_ALL,H5P_DEFAULT,
      data); H5REPORT(status);

   /* write parameter value */
   parm = 37;
   status = H5Dwrite
     (parmset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,
      &parm); H5REPORT(status);

   /* End access to the dataset and release resources used by it. */
   status = H5Dclose(dataset);
   status = H5Dclose(parmset);

   /* Terminate access to the data space. */
   status = H5Sclose(dataspace);
   status = H5Sclose(parmspace);

   /* Close the file. */
   status = H5Fclose(file_id);
}
```

```
%% h5dump wdset.h5
HDF5 "wdset.h5" {
GROUP "/" {
   DATASET "dset" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
```

```
   DATA {(0,0): 0.5, 1.5, 2.5, 3.5, 4.5, 5.5,
(1,0): 6.5, 7.5, 8.5, 9.5, 10.5, 11.5,(2,0): 12.5,
13.5, 14.5, 15.5, 16.5, 17.5,(3,0): 18.5, 19.5, 20.5,
21.5, 22.5, 23.5}}DATASET "parm" {
DATATYPE  H5T_STD_I32LEDATASPACE  SCALARDATA {
(0): 37}}}}
```

如果你仔细查看源代码和转储，你会看到数据类型被声明为“native”，但渲染为 LE。“native”声明使数据类型表现得像内置的 C 或 Fortran 数据类型。或者，你可以明确指出数据是小端还是大端。这些术语描述了数据项的字节在内存中的排列顺序。大多数架构使用小端，如你在转储输出中所见，但值得注意的是，IBM 使用大端。

7.5 读取

现在我们已经有了包含一些数据的文件，可以进行故事的镜像部分：从该文件读取。基本命令是

```
h5file = H5Fopen( .... )
....
H5Dread( dataset, .... data .... )
```

其中 H5Dread 命令的参数与对应的 H5Dwrite 相同。

练习。 通过编译并运行下面的 allread.c 示例，从你在前一个练习中创建的 wdset.h5 文件中读取数据。

预期结果。 运行 allread 可执行文件将打印参数的值 37，以及数组中 8.5 的数据点值。

注意事项。 确保你先运行 parmwrite 来创建输入文件。#include "myh5defs.h"

#define FILE "wdset.h5"main() {hid_t   file_id, dataset, parmset;

```
   herr_t      status;
   double data[24]; int parm;

   /* Open an existing file */
   file_id = H5Fopen(FILE, H5F_ACC_RDONLY, H5P_DEFAULT);
   H5REPORT(file_id);

   /* Locate the datasets. */
   dataset = H5Dopen(file_id, "/dset"); H5REPORT(dataset);
   parmset = H5Dopen(file_id,"/parm");  H5REPORT(parmset);

   /* Read data back */
   status = H5Dread
     (parmset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,
      &parm); H5REPORT(status);
   printf("parameter value: %d\n",parm);

   status = H5Dread
     (dataset,H5T_NATIVE_DOUBLE,H5S_ALL,H5S_ALL,H5P_DEFAULT,
      data); H5REPORT(status);
   printf("arbitrary data point [1,2]: %e\n",data[1*6+2]);

   /* Terminate access to the datasets */
   status = H5Dclose(dataset); H5REPORT(status);
   status = H5Dclose(parmset); H5REPORT(status);

   /* Close the file. */
   status = H5Fclose(file_id);
}
```

```
 %% ./allread
parameter value: 37
arbitrary data point [1,2]: 8.500000e+00
```

第 8 章

并行 I/O

并行 I/O 是一个棘手的话题。你可以尝试让所有处理器共同写入一个文件，或者为每个进程写一个文件然后再合并。使用你的编程语言的标准机制时，有以下考虑：

- 在进程拥有独立文件系统的集群上，写入单个文件的唯一方法
  是让单个处理器生成该文件。
- 为每个进程写一个文件很容易做到，但 – 你需要一个后处理脚本；– 如果文件不在共享文件系统上（例如 *Lustre*），则需要额外的工作将它们汇集起来；– 如果文件在共享文件系统上，写入许多文件可能会给元数据服务器带来负担。

- 在共享文件系统上，所有文件都可以打开同一个文件并设置文件指针
  单独操作。如果每个进程的数据量不均匀，这可能会很困难。

说明最后一点：

```
// pseek.c
FILE *pfile;
pfile = fopen("pseek.dat","w");
fseek(pfile,procid*sizeof(int),SEEK_CUR);
//  fseek(pfile,procid*sizeof(char),SEEK_CUR);
fprintf(pfile,"%d\n",procid);
fclose(pfile);
```

MPI 也有其自己的可移植 I/O：*MPI I/O*，详见章节 *ParallelProgrammingbook, chapter 10*。

或者，可以使用诸如 *hdf5* 之类的库；参见 7。

For a great discussion see [15], from which figures here are taken.

8.1 Usesequential I/O

MPI 进程可以执行常规进程能做的任何操作，包括打开文件。这是最简单的并行 I/O 形式：每个 MPI 进程打开自己的文件。为了防止写入冲突，

- 你使用 *MPI_Comm_rank* 来生成唯一的文件名，或者
- 你使用本地文件系统，通常是 `/tmp`，它对每个进程是唯一的，或者至少对节点上的进程组是唯一的。

对于读取，实际上所有进程都可以打开同一个文件，但对于读取来说这并不真正可行。因此需要唯一的文件。

8.2 MPI I/O

在章节 并行编程书籍，章节 *10* 中我们讨论了 MPI I/O。这是一种让通信器上的所有进程打开单个文件，并以协调的方式写入它的方法。这有一个很大的优点，即最终结果是一个普通的 Unix 文件。

8.3 Higher levellibraries

诸如 *NetCDF* 或 *HDF5* (参见第 7 章) 的库相比 MPI I/O 具有优势：

- 文件可以是操作系统无关的，消除了对小端存储等问题的担忧。
- 文件是自我描述的：它们包含描述其内容的元数据。

第 9 章

使用 GNUplot 绘图

gnuplot 工具是一个用于绘制点集或曲线的简单程序。本简短教程将向您展示一些基础知识。有关更多命令和选项，请参阅手册 http://www.gnuplot.info/docs/gnuplot.html。

9.1 使用模式

`gnuplot` 的两种运行模式是交互式 和 从文件。在交互式模式下，您从命令行调用 `gnuplot`，输入命令，并观察输出；您可以用 `quit` 终止一个交互式会话。如果您想保存交互式会话的结果，请执行 `save "name.plt"`。该文件可以编辑，并用 `load "name.plt"` 加载。

非交互式绘图时，您调用 `gnuplot <your file>`。

T `gnuplot` 的输出可以是屏幕上的图片，也可以是文件中的绘图指令。
o输出的位置取决于终端的设置。默认情况下， `gnuplot` 会尝试绘制一张图片。
i这相当于声明

```
set terminal x11
```

或 `aqua`, `windows`，或任何选择的图形硬件。

For output to file, declare

```
set terminal pdf
```

或 `fig`, `latex`, `pbm`，等等。注意，这只会导致 pdf 命令被写到你的屏幕上：你需要用

```
set output "myplot.pdf"
```

或用以下方式捕获它们

```
gnuplot my.plt > myplot.pdf
```

9.2 绘图

基本的绘图命令是 `plot` 用于二维绘图， `splot` （ “surface plot” ）用于三维绘图。

9.2.1 绘制曲线

通过指定

```
plot x**2
```

你会得到一个 $f(x) = x^2$的图； `gnuplot` 将决定 x 的范围。

```
set xrange [0:1]plot 1-x title "down",
 x**2 title "up"
```

你会在一个图中得到两个图形， x 的范围限制在 $[0, 1]$, , 并且图形有相应的图例。变量 `x` 是绘图函数的默认值。

将一个函数相对于另一个函数绘图 —— 或者等价地，绘制参数曲线 —— 步骤如下:

```
set parametric
plot [t=0:1.57] cos(t),sin(t)
```

这将得到一个四分之一圆。

要在一个图中绘制多个图形，使用命令 `set multiplot`。

9.2.2 绘制数据点

也可以基于数据点绘制曲线。基本语法是 `plot 'datafile'`，它从数据文件中取两列并将其解释为 (x, y) 坐标。由于数据文件通常包含多列数据，常用语法是针对第 3 列和第 6 列的 `plot 'datafile' using 3:6` 。进一步的限定符如 `with lines` 指示如何连接这些点。

类似地， `splot "datafile3d.dat" 2:5:7` 会将三列解释为指定三维绘图的 (x, y, z) 坐标。

如果数据文件被解释为矩形网格上的水平或高度值，使用 `splot "matrix.dat"matrix` 来绘制数据点；并用

```
split "matrix.dat" matrix with lines
```

9.2.3 Customization

图表可以通过多种方式进行自定义。其中一些自定义使用了 `set` 命令。例如，

```
set xlabel "time"
set ylabel "output"
set title "Power curve"
```

你也可以使用 `set style function dots` 更改
默认绘图样式

（`dots`, `lines`, `dots`, `points`，等等），或者使用
`plot f(x) with points` 在单个图上更改

9.3 工作流程

假设你的代码生成了一个你想绘制的数据集，并且你针对多个输入运行了代码。如果绘图能够自动化那就好了。Gnuplot 本身没有提供这样的功能，但借助一些 shell 编程，这并不难实现。

假设你有数据文件
`data1.dat data2.dat data3.dat`

并且你想用相同的 gnuplot 命令绘制它们。你可以制作一个文件 `plot.template`：

```
set term pdf
set output "FILENAME.pdf"
plot "FILENAME.dat"
```

字符串 `FILENAME` 可以用实际的文件名替换，例如使用 `sed`：

```
for d in data1 data2 data3 ; do
  cat plot.template | sed s/FILENAME/$d/ > plot.cmd
  gnuplot plot.cmd
done
```

这个基本思路有很多变体。

第 10 章

良好的编码实践

迟早，可能是早于迟的时候，每个程序员都会遇到代码行为不符合预期的情况。在本节中，您将学习一些处理此问题的技术。首先，我们将看到一些用于防止错误的技术；在下一章中，我们将讨论调试，即在程序出现不可避免的错误后，查找这些错误的过程。

10.1 防御性编程

本节将讨论一些旨在防止编程错误发生的可能性，或增加它们在运行时被发现可能性的技术。我们称之为防御性编程。

科学代码通常庞大且复杂，因此养成在编码时意识到自己会犯错并为此做好准备的习惯是一个好做法。另一个良好的编码实践是使用工具：如果别人已经为你做了某些事情，就没有必要重新发明轮子。这些工具中的一些将在其他章节中描述：

- 构建系统，如 Make 、 Scons 、 Bjam ；参见第 3 节。
- 使用 Git 进行源代码管理；参见第 5 节。
- 回归测试和以测试为中心的设计（单元测试）

首先我们将看看运行时的健全性检查，即测试那些不可能或不应该发生的情况。

10.1.1 断言

在程序可能出错的情况中，我们可以区分错误和漏洞。错误是合法发生但不应该发生的事情。文件系统是常见的错误来源：程序想打开一个文件，但文件不存在，因为用户输入错误的名称，或者程序写入文件时磁盘已满。其他错误可能来自算术运算，例如溢出错误。

另一方面，程序中的 *bug* 是一种不应合法发生的情况。当然，这里的 “ 合法 ” 是指 “ 根据程序员的意图 ”。Bug 通常可以描述为 “ 计算机总是做你要求它做的事，而不一定是你想要的事 ”。

断言用于检测程序中的错误：断言 是在程序某个点上应该为真的谓词。因此，断言失败意味着你没有编写出你本意要编写的代码。断言通常是你编程语言中的一条语句，或是预处理宏；断言失败时，程序将停止运行。

一些断言的示例：

- 如果子程序有一个数组参数，最好在对数组进行索引之前测试实际参数是否为空指针。
- 同样，你可以测试动态分配的数据结构是否不为空指针。
- 如果你计算的数值结果应满足某些数学属性，例如你正在编写一个正弦函数，其结果必须在 $[-1, 1]$ 中，你应该测试该属性是否确实对结果成立。

断言通常在程序经过充分测试后被禁用。原因是断言的执行可能代价较高。例如，如果你有一个复杂的数据结构，你可以编写一个复杂的完整性测试，并在断言中执行该测试，该断言放在每次访问数据结构之后。

因为断言通常在代码的 “ 生产 ” 版本中被禁用，所以它们不应影响任何存储的数据。如果影响了，当你在测试时使用断言，代码的行为可能会与实际使用时没有断言的行为不同。这也被表述为 “ 断言不应有副作用 ”。

10.1.1.1 C 语言中的 assert 宏

C 标准库有一个文件 assert.h，它提供了一个 assert() 宏。插入 assert(foo) 的效果如下：如果 foo 为零（假），则在标准错误上打印诊断信息：

```
Assertion failed: foo, file filename, line line-number
```

其中包括表达式的字面文本、文件名和行号；随后程序停止。下面是一个示例：

```
#include<assert.h>

void open_record(char *record_name)
{
  assert(record_name!=NULL);
  /* Rest of code */
}

int main(void)
{
  open_record(NULL);
}
```

可以通过定义 NDEBUG 宏来禁用 assert 宏。

10.1.1.2 Fortran 的断言宏

(感谢 Robert Mclay 提供此代码。)

```
#if (defined( GFORTRAN ) || defined( G95 ) || defined ( PGI) )
# define MKSTR(x) "x"#else# define MKSTR(x) #x#endif#ifndef NDEBUG
# define ASSERT(x, msg) if (.not. (x) ) \
call assert( FILE , LINE ,MKSTR(x),msg)#else# define ASSERT(x, msg)
#endifsubroutine assert(file, ln, testStr, msgIn)implicit none
character(*) :: file, testStr, msgIninteger :: lnprint *, "Assert: ",
trim(testStr)," Failed at ",trim(file),":",lnprint *, "Msg:", trim(msgIn)
stopend subroutine assert
```

其用作 `ASSERT(nItemsSet.gt.arraySize,"Too many elements set")`

10.1.2 Use of errorcodes

在一些软件库中（例如 MPI 或 PETSc ），每个子程序都会返回一个结果，要么是函数值，要么是一个参数，用以指示例程的成功或失败。即使你认为不可能出错，检查这些错误参数也是良好的编程实践。

以总是包含错误参数的方式编写你自己的子程序也是一个好主意。让我们考虑一个执行某些数值计算的函数的情况。

```
float compute(float val){
float result;
result = ... /* some computation */
return result;}float value,result;
result = compute(value);
```

看起来不错？如果计算可能失败怎么办，例如：
result = ... sqrt(val) ... /* some computation */ 我们如何处
理用户传入负数的情况？ float compute(float val){float result;
if (val<0) { /* then what? */} else
result = ... sqrt(val) ... /* some computation */
return result;}

我们可以打印错误信息并返回某个结果，但信息可能被忽略，调用环境实际上并未收到任何出错的通知。

以下方法更灵活：int compute(float val,float *result){
float result;if (val<0) {return -1;} else {
result = ... sqrt(val) ... / some computation */}return 0;
}float value,result; int ierr;ierr = compute(value,&result);
if (ierr!=0) { /* take appropriate action */}

你可以通过编写 #define CHECK_FOR_ERROR(ierr) \
if (ierr!=0) { \printf("Error %d detected\n",ierr); \
return -1 ; }....
ierr = compute(value,&result); CHECK_FOR_ERROR(ierr); 来节省
大量输入工作

C 预处理器（ CPP ）具有内置宏，便于进行信息丰富的错误报告。以下宏不仅检查错误条件，还报告错误发生的位置：

```
#define CHECK_FOR_ERROR(ierr) \if (ierr!=0) { \
printf("Error %d detected in line %d of file %s\n",\
ierr,__LINE__,__FILE__); \return -1 ; }
```

注意该宏不仅打印错误信息，还会进一步返回。这意味着，如果你系统地采用这种错误代码的使用方式，当发生错误时，你将获得完整的调用树回溯。（在 Python 语言中，这恰恰是错误的方法，因为回溯是内置的。）

10.2 Guardingagainst memory errors

在科学计算中，不言而喻你将处理大量数据。有些编程语言使数据管理变得容易，而有些语言，可以说，使得数据错误更容易发生。

以下是一些内存违规的示例。

- 数组边界外写入。如果地址位于用户内存之外，您的代码可能会因段错误等错误而退出，并且该错误相对容易找到。如果地址仅仅在数组之外，它会破坏数据但不会使程序崩溃；这种错误可能长时间未被发现，因为它可能没有影响，或者只是在计算中引入细微的错误值。
- 数组边界外读取比写入错误更难发现，因为它通常不会停止您的代码运行，而只是引入错误的值。
- 使用未初始化的内存类似于数组边界外读取，可能长时间未被发现。这种错误的一种变体是将内存附加到未分配的指针上。这种特定类型的错误可能表现出有趣的行为。假设您注意到程序行为异常，您用调试模式重新编译以查找错误，但现在错误不再发生。这很可能是因为在低优化级别下，所有分配的数组都被填充为零。因此，您的代码原本读取的是随机值，但现在得到的是零。

本节包含一些防止处理您为数据保留的内存时出现错误的技术。

10.2.1 Array bound checking and other memory techniques

数组边界检查，即检测数组访问是否确实指向合法位置，会带来运行时开销。因此，您可能只想在代码的测试阶段进行此操作，或者避免在计算密集型循环中使用。

10.2.1.1 C

C 语言有数组，但它们存在 “ 指针衰减 ” 问题：在内存中它们的行为很大程度上类似指针。因此，边界检查很难实现，除非使用像 *Valgrind* 这样的外部工具。

10.2.1.2 C++

C++ 有诸如 `std::vector` 这样的容器，支持边界检查：

```
vector<float> x(25);
x.at(26) = y; // throws an exception
```

另一方面，C 风格的 `x[26]` 不执行此类检查。

10.2.1.3 Fortran

Fortran 数组比 C 数组更受限制，因此编译器通常支持一个用于激活运行时边界检查的标志。对于 *gfortran* ，该标志是 `-fbounds-check`。

10.2.2 内存泄漏

如果程序分配了内存，但随后丢失了对该内存的跟踪，我们称该程序存在内存泄漏。操作系统会认为这部分内存正在使用中，实际上并未使用，因此计算机内存可能会被无用的已分配内存填满。

在此示例中，数据在词法作用域内分配：

```
for (i=.... ) {
  real *block = malloc( /* large number of bytes */ )
  /* do something with that block of memory */
  /* and forget to call "free" on that block */
}
```

每次迭代都会分配一块内存，但一次迭代中的分配在下一次迭代中不再可用。类似的例子也可以在条件语句内部进行分配。

需要注意的是，这个问题在 Fortran 中要轻得多，因为当变量超出作用域时，内存会自动释放。

有各种工具可以检测内存错误：Valgrind 、 DMALLOC 、 Electric Fence。关于 valgrind，参见第 11.8 节。

10.2.3 自定义 malloc

许多编程错误源于动态分配内存的不当使用：程序写入了边界之外，或者写入了尚未分配的内存，或者写入了已经释放的内存。虽然一些编译器可以在运行时进行边界检查，但这会降低程序的运行速度。更好的策略是编写自己的内存管理。一些库如 PETSc 已经提供了增强的 malloc ；

如果有这个功能，你当然应该利用它。（ *gcc* 编译器中有一个函数 *mcheck*，定义在 mcheck.h，具有类似功能。）

如果你用 C 语言编写代码，你可能会知道 malloc 和 free 调用：

```
int *ip;
ip = (int*) malloc(500*sizeof(int));
if (ip==0) {/* could not allocate memory */}
..... do stuff with ip .....
free(ip);
```

你可以通过 #define MYMALLOC(a,b,c) \

a = (c*)malloc(b*sizeof(c)); \

if (a==0) {/* error message and appropriate action */}int *ip;

MYMALLOC(ip,500,int); 来减少一些输入量

对内存使用的运行时检查（无论是编译器生成的边界检查，还是通过 valgrind 或 Rational Purify 等工具）都很昂贵，但你可以通过为 malloc 添加一些功能来捕捉许多问题。我们这里要做的是在事后检测内存破坏。

我们在分配的对象的左侧和右侧各分配几个整数（下面代码的第 1 行），并在它们中放入一个可识别的值（第 2 行和第 3 行），以及对象的大小（第 2 行）。然后我们返回指向实际请求内存区域的指针（第 4 行）。

```
#define MEMCOOKIE 137
#define MYMALLOC(a,b,c) { \
  char *aa; int *ii; \
  aa = malloc(b*sizeof(c)+3*sizeof(int)); /* 1 */ \
  ii = (int*)aa; ii[0] = b*sizeof(c); \
          ii[1] = MEMCOOKIE;              /* 2 */ \
  aa = (char*)(ii+2); a = (c*)aa ;        /* 4 */ \
  aa = aa+b*sizesof(c); ii = (int*)aa; \
          ii[0] = MEMCOOKIE;              /* 3 */ \
  }
```

现在你可以编写你自己的 free，它测试对象的边界是否未被覆盖。

```
#define MYFREE(a) { \
  char *aa; int *ii,; ii = (int*)a; \
  if (*(--ii)!=MEMCOOKIE) printf("object corrupted\n"); \
  n = *(--ii); aa = a+n; ii = (int*)aa; \
  if (*ii!=MEMCOOKIE)  printf("object corrupted\n"); \
```

```
}
```

Y你可以扩展这个想法：在每个分配的对象中，也存储两个指针，这样分配的内存区域就变成了一个双向链表。然后你可以编写一个宏 `CHECKMEMORY`，用来检测所有分配的对象是否被破坏。

这种解决内存破坏问题的方法相当容易编写，并且开销很小。每个对象最多有 5 个整数的内存开销，性能几乎没有损失。

(你可以不写 `malloc` 的包装函数，在某些系统上你可以影响系统的行为r例程。在 Linux 上，`malloc` 调用的钩子可以被替换成你自己的例程；参见 http://www.gnu.org/s/libc/manual/html_node/Hooks-for-Malloc.html.)

10.3 测试

有多种测试代码正确性的理念。

- 正确性证明：程序员制定描述代码片段预期行为的谓词，并通过数学技术证明这些谓词成立 [10, 5].
- 单元测试：每个例程单独测试其正确性。对于数值代码，这种方法通常很难实现，因为浮点数存在本质上的无限可能输入，且很难决定什么构成足够的输入集合。
- 集成测试：测试子系统
- 系统测试：测试整个代码。这通常适用于数值代码，因为我们经常有已知解的模型问题，或者存在需要在全局解上保持的性质，如界限。
- 测试驱动设计：程序开发过程由随时可能进行测试的需求驱动。

对于并行代码，我们遇到了测试方面的新类别困难。许多算法在并行执行时，会以略有不同的顺序执行操作，导致不同的舍入误差行为。例如，向量求和的并行计算将使用部分和。一些算法具有固有的数值误差阻尼，例如平稳迭代方法（HPC 书，章节 5.5.1），但其他算法没有这种内置的误差校正（非平稳方法；HPC 书，章节 5.5.8）。因此，同一迭代过程根据使用的处理器数量可能需要不同的迭代次数。

10.3.1 单元测试

单元测试是一种确保代码正确性的方法。为此，测试必须对代码有完整的覆盖：代码中的所有语句都应包含在测试中。

单元测试也是记录代码使用方式的一种方法：它们展示了代码的预期用法。

几点说明：

- 程序中的全局状态使测试变得困难，因为它在测试之间携带信息。
- 测试不应重复你的代码逻辑：如果程序逻辑有误，测试也会有误。

- 测试应简短，并遵守单一职责原则。为测试命名有助于保持其专注。

10.3.2 测试驱动的设计与开发

在测试驱动设计中，强调代码始终可测试。基本思想如下。

- 整个代码及其各部分应始终可测试。
- 在扩展代码时，只做允许测试的最小改动。
- 每次更改前后都要进行测试。

确保在添加新功能之前保证正确性。

本系列的第 3 卷讨论了测试驱动开发（ TDD ）和单元测试，使用 *Catch2* 框架。详见 科学编程导论一书，第 *70* 章。

第 11 章

调试

> 调试就像是在犯罪电影中既是侦探又是凶手。(Filipe Fortes, 2013)

当程序表现异常时，调试是找出原因的过程。寻找程序错误有多种策略。最粗糙的方法是通过打印语句进行调试。如果你大致知道错误出现在代码的哪个位置，可以编辑代码插入打印语句，重新编译，重新运行，看看输出是否给你任何提示。这种方法存在几个问题：

- 编辑 / 编译 / 运行的循环非常耗时，尤其是因为
- 错误往往是由代码的早期部分引起的，这需要你反复编辑、编译和运行。此外，
- 你的程序产生的数据量可能过大，无法有效地显示和检查，且
- 如果你的程序是并行的，你可能需要打印出所有处理器的数据，这使得检查过程非常繁琐。

基于这些原因，调试的最佳方式是使用交互式调试器，这是一种允许你监控和控制正在运行程序行为的程序。在本节中，你将熟悉 *gdb* 和 *lldb*，它们分别是 *GNU* 和 *clang* 项目的开源调试器。其他调试器是专有的，通常随编译器套件一起提供。另一个区别是 gdb 是命令行调试器；也有图形调试器，如 *ddd*（ gdb 的前端）或 *DDT* 和 *TotalView*（并行代码的调试器）。我们仅限于 gdb，因为它包含了所有调试器共有的基本概念。

在本教程中，你将使用 gdb 和 valgrind 调试多个简单程序。文件可以在代码仓库的目录 `code/gdb` 中找到。

11.1 为调试编译

你通常需要在调试代码之前重新编译它。第一个原因是二进制代码通常不知道哪些变量名对应哪些内存位置，或者源代码中的哪些行对应哪些指令。为了让二进制可执行文件知道这些信息，你必须在其中包含符号表，这可以通过在编译器命令行中添加 `-g` 选项来完成。

表 11.1: 常用 gdb / lldb 命令列表。

gdb	lldb
启动调试器运行	
`$ gdb program` `(gdb) run`	`$ lldb program` `(lldb) run`
显示堆栈跟踪	
`(gdb) where`	`(lldb) thread backtrace`
调查特定帧	
`frame 2`	`frame select 2`
运行 / 步进	
`run / step / continue`	线程继续 / 单步进入 / 跳过 / 跳出
在某行设置断点	
`break foo.c:12`	`breakpoint set [ -f foo.c ] -l 12`
`break foo.c:12 if n>0`	
`info breakpoints`	
为异常设置断点	
`catch throw`	`break set -E C++`

通常，你还需要降低编译器优化级别：生产代码通常会使用诸如 `-O2` 或 `-Xhost` 这样的标志进行编译，以尽可能提高代码速度，但调试时你需要将其替换为 `-O0`（‘oh-zero’）。原因是较高级别会重组你的代码，使得执行过程难以对应到源代码 [1]。

11.2 调用调试器

使用 gdb 有三种方式：用它启动程序，将其附加到已运行的程序，或用它检查 *coredump*。我们这里只考虑第一种可能。

启动调试运行

gdb	lldb
`$ gdb program`	`$ lldb program`
`(gdb) run`	`(lldb) run`

下面是如何启动没有参数的程序的 gdb 示例（ Fortran 用户，请使用 `hello.F` ）：

tutorials/gdb/c/hello.c`#include <stdlib.h>#include <stdio.h>int main() {`

```
printf("hello world\n");return 0;}%% cc -g -o hello hello.c# regular invocation:
%% ./hellohello world# invocation from gdb:%% gdb hello
GNU gdb 6.3.50-20050815 # ..... [version info]
Copyright 2004 Free Software Foundation, Inc. .... [copyright info] ....
(gdb) runStarting program: /home/eijkhout/tutorials/gdb/hello
Reading symbols for shared libraries +. donehello world
Program exited normally.(gdb) quit%%
```

1. Typically, actual code motion is done by `-O3`, but at level `-O2` the compiler will inline functions and make other simplifications.

重要提示：程序是用 调试标志 `-g` 编译的。这会导致 符号表 (即从机器地址到程序变量的映射) 和其他调试信息被包含在二进制文件中。这会使你的二进制文件比严格必要的要大，但也会使其运行更慢，例如因为编译器不会执行某些优化 2。

为了说明符号表的存在，执行

```
%% cc -g -o hello hello.c
      %% gdb hello
      GNU gdb 6.3.50-20050815 # ..... version info
      (gdb) list
```

并将其与省略 `-g` 标志进行比较：

```
      %% cc -o hello hello.c
      %% gdb hello
      GNU gdb 6.3.50-20050815 # ..... version info
      (gdb) list
```

对于带有命令行输入的程序，我们将参数传递给 `run` 命令（ Fortran 用户使用 `say.F` ）：

tutorials/gdb/c/say.c

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc,char **argv) {
  int i;
  for (i=0; i<atoi(argv[1]); i++)
    printf("hello world\n");
  return 0;
}
```

```
%% cc -o say -g say.c
%% ./say 2
hello world
hello world
%% gdb say
.... the usual messages ...
(gdb) run 2
Starting program: /home/eijkhout/tutorials/gdb
Reading symbols for shared libraries +. done
hello world
hello world

Program exited normally.
```

11.3 寻找错误：where, frame, print

现在让我们考虑一些带有错误的程序。

2. 编译器优化不应改变程序的语义，但有时会。这可能导致噩梦场景，即程序崩溃或给出错误结果，但在使用调试编译并在调试器中运行时却神奇地正常工作。

11.3.1 C 程序

以下代码存在多个错误。我们将使用调试器来发现它们。

```
// square.cint nmax,i;float *squares,sum;
fscanf(stdin,"%d",nmax);
for (i=1; i<=nmax; i++) {
squares[i] = 1./(i*i); sum += squares[i];}
printf("Sum: %e\n",sum);
%% cc -g -o square square.c
%% ./square5000Segmentation fault
```

segmentation fault（也可能出现其他消息）表示我们正在访问不允许访问的内存，导致程序退出。调试器会快速告诉我们错误发生的位置：

```
%% gdb square
(gdb) run50000Program received signal EXC_BAD_ACCESS,
 Could not access memory.

Reason: KERN_INVALID_ADDRESS at address: 0x000000000000eb4a
0x00007fff824295ca in __svfscanf_l ()
```

显然错误发生在一个函数 `__svfscanf_l` 中，该函数不是我们的，而是系统函数。使用 `backtrace`（或 `bt`，也可以是 `where` 或 `w`）命令，我们显示 调用栈。这通常可以帮助我们找出错误所在：

显示堆栈跟踪

gdb	lldb
`(gdb) where`	`(lldb) thread backtrace`

```
(gdb) where
#0  0x00007fff824295ca in __svfscanf_l ()
#1  0x00007fff8244011b in fscanf ()
#2  0x0000000100000e89 in main (argc=1, argv=0x7fff5fbfc7c0) at square.c:7
```

We inspect the actual problem:

Investigate a specific frame

gdb	clang
`frame 2`	`frame select 2`

我们仔细查看第 7 行，发现需要将 `nmax` 改为 `&nma x`.

我们的程序仍然有错误：

```
(gdb) run
50000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x000000010000f000
0x0000000100000ebe in main (argc=2, argv=0x7fff5fbfc7a8) at square1.c:9
9           squares[i] = 1./(i*i); sum += squares[i];
```

我们进一步调查：

```
(gdb) print i
$1 = 11237
(gdb) print squares[i]
Cannot access memory at address 0x10000f000
(gdb) print squares
$2 = (float *) 0x0
```

我们很快发现忘记分配了 `squares`。

B顺便说一下，我们这里很幸运：这种类型的内存错误并不总是被检测到。启动我们的程序
w使用较小的输入不会导致错误：

```
(gdb) run50

Sum: 1.625133e+00

Program exited normally.
```

如果我们有一个合法的数组，但访问了它的边界之外，也可能发生内存错误。下面的程序正向填充一个数组，然后反向读取它。然而，第二个循环中存在索引错误。

```
// up.cint nlocal = 100,i;double s,
 *array = (double*) malloc(nlocal*sizeof(double));for (i=0; i<nlocal; i++) {
double di = (double)i;array[i] = 1/(di*di);}s = 0.;for (i=nlocal-1; i>=0; i++) {
double di = (double)i;s += array[i];}
Program received signal EXC_BAD_ACCESS, Could not access memory.
```

```
Reason: KERN_INVALID_ADDRESS at address: 0x0000000100200000
0x0000000100000f43 in main (argc=1, argv=0x7fff5fbfe2c0) at up.c:15
15          s += array[i];
(gdb) print array
$1 = (double *) 0x100104d00
(gdb) print i
$2 = 128608
```

你会看到调试器最终报错的索引远大于数组的大小。

练习 11.1. 你能想到为什么越界索引不会立即导致程序崩溃的原因吗？什么因素决定了它最终会成为问题？（提示：计算机内存是如何结构化的？）

在第 11.8 节中，你将看到一个可以检测任何越界索引的工具。

11.3.2 Fortran 程序

编译并运行以下程序：

MISSING SNIPPET gdb-squaref

它应该会提前结束，并显示诸如 “ 非法指令 ” 之类的消息。在 gdb 中运行程序可以快速告诉你问题出在哪里：

```
(gdb) run
Starting program: tutorials/gdb//fsquare
Reading symbols for shared libraries ++++. done

Program received signal EXC_BAD_INSTRUCTION,
Illegal instruction/operand.
0x0000000100000da3 in square () at square.F:7
7                sum = sum + squares(i)
```

我们仔细查看代码，发现我们没有正确分配 `squares` 。

11.4 Stepping through a program

逐步执行程序

gdb	lldb	meaning
`run`		start a run
`cont`		continue from breakpoint
`next`		next statement on same level
`step`		next statement, this level or next

O程序中的错误往往非常隐晦，需要详细调查程序运行情况。
编译以下程序

```
// roots.c
float root(int n)
{
  float r;
  r = sqrt(n);
  return r;
}

int main() {
  feenableexcept(FE_INVALID | FE_OVERFLOW);
  int i;
  float x=0;
  for (i=100; i>-100; i--)
    x += root(i+5);
  printf("sum: %e\n",x);
```

并运行它：%% ./rootssum：nan 像之前一样在 gdb 中启动

它：%% gdb rootsGNU gdb 6.3.50-20050815

Copyright 2004 Free Software Foundation, Inc.....

b但在运行程序之前，你在 main 处设置了一个断点。这告诉执行过程停止，或 “ 中断 ”，
i在主程序中。

```
(gdb) break main
Breakpoint 1 at 0x100000ea6: file root.c, line 14.
```

现在程序将在 main 中的第一条可执行语句处停止：

```
(gdb) run
Starting program: tutorials/gdb/c/roots
Reading symbols for shared libraries +. done
Breakpoint 1, main () at roots.c:14
14          float x=0;
```

大多数时候你会在特定行设置断点：

在某一行设置断点

gdb	lldb
`break foo.c:12`	`breakpoint set [ -f foo.c ] -l 12`

如果执行在断点处停止，你可以做各种操作，比如发出 `step` 命令：

```
Breakpoint 1, main () at roots.c:14
14        float x=0;(gdb) step
15        for (i=100; i>-100; i--)
(gdb)16           x += root(i);(gdb)
```

（如果你直接按回车，之前发出的命令会被重复执行）。连续多次执行 `step`，通过按回车。你注意到函数和循环有什么不同吗?

将 `step` 改为 `next`。现在你注意到循环和函数有什么不同了吗?

设置另一个断点：`break 17` 并执行 `cont`。发生了什么?

在包含 `sqrt` 调用的行上设置断点后重新运行程序。当执行停止时，执行 `where` 和 `list`。

- 如果你设置了许多断点，可以用 `info breakpoints` 找出它们。
- 你可以用 `delete n` 移除断点，其中 `n` 是断点的编号。
- 如果你在不退出 gdb 的情况下用 `run` 重启程序，断点仍然有效。
- 如果你退出 gdb，断点会被清除，但你可以保存它们：`save breakpoints <file>`。使用 `source <file>` 在下一次 gdb 运行时读取它们。

11.5 检查值

在 gdb 中再次运行前面的程序：在实际调用 `run` 之前，在执行 `sqrt` 调用的那一行设置断点。当程序运行到第 8 行时，你可以执行 `print n`。执行 `cont`。程序在哪里停止?

如果你想修复一个变量，你可以执行 `set var=value`。更改变量 `n` 并确认计算了新值的平方根。你执行了哪些命令?

11.6 断点

如果问题出现在循环中，不断输入 `cont` 并用 `print` 检查变量会很繁琐。相反，你可以给现有断点添加条件。首先，你可以让断点带有条件：使用

```
condition 1 if (n<0)
```

断点 1 只有在 `n<0` 为真时才会生效。

你也可以设置一个只有在某些条件下才激活的断点。语句 `break 8 if (n<0)`

意味着断点 8 在遇到条件 `n<0` 后变为（无条件）激活 .

设置断点

gdb	lldb
`break foo.c:12`	`breakpoint set [ -f foo.c ] -l 12`
`break foo.c:12 if n>0`	

Remark 16 你可以用以下技巧在 `NaN` 处设置断点 *:*

```
break foo.c:12 if x!=x
```

利用 `NaN` 是唯一不等于它自身的数字这一事实。

Another possibility is to use `ignore 1 50`, which will not stop at breakpoint 1 the next 50 times.

移除现有断点，使用条件 `n<0` 重新定义断点并重新运行程序。当程序中断时，查找循环变量的哪个值导致了中断。你使用的命令序列是什么?

你可以通过多种方式设置断点:

- `break foo.c` 在达到某个文件中的代码时停止;
- `break 123` 在当前文件的某一行停止;
- `break foo` 在子程序 `foo` 处停止
- 或各种组合，例如 `break foo.c:123`。

关于断点的信息:

- 如果你设置了许多断点，可以用 `info breakpoints` 找出它们。
- 你可以用 `delete n` 移除断点，其中 `n` 是断点的编号。
- 如果你用 `run` 重启程序而不退出 gdb，断点仍然有效。
- 如果你退出 gdb，断点会被清除，但你可以保存它们: `save breakpoints <file>`。使用 `source <file>` 在下一次 gdb 运行时读取它们。
- 在带有异常的语言中，比如 *C++*，你可以设置一个捕获点: 为

异常设置断点 gdb clang`catch throw  break set -E C++`

最后，你可以在断点处执行命令:

```
break 45
command
print x
cont
end
```

这表示在第 45 行变量 `x` 将被打印，且执行应立即继续。

如果你想对同一个程序多次运行 gdb 会话，你可能想保存并重新加载断点 -
。这可以通过

```
save-breakpoint filename
source filename
```

11.7 内存调试

编程中的许多问题源于内存错误。我们首先简要介绍最常见的类型，然后讨论帮助您检测它们的工具。

11.7.1 内存错误类型

11.7.1.1 无效指针

解引用一个未指向已分配对象的指针可能导致错误。如果您的指针无论如何指向有效内存，您的计算将继续，但结果不正确。

然而，更可能的是您的程序会因段错误或总线错误而退出。

11.7.1.2 Out-of-bounds errors

A访问已分配对象边界之外的地址不太可能导致程序崩溃，但更可能
t产生错误的结果。

超出边界足够大的量会再次导致段错误，但超出边界少量可能会读取无效数据，或破坏其他变量的数据，导致错误结果，这些错误可能长时间未被发现。

11.7.1.3 内存泄漏

如果分配的内存变得不可访问，我们称之为内存泄漏。示例：

```
if (something) {
  double *x = malloc(10*sizeofdouble);
  // do something with x
}
```

条件语句之后，分配的内存没有被释放，但指向它的指针已经消失。

最后这一类尤其难以发现。内存泄漏只会在程序耗尽内存时显现出来。反过来，这可以通过分配失败来检测。始终检查你的 *malloc* 或 *allocate* 语句的返回结果是个好主意！

11.8 使用 Valgrind 进行内存调试

导致内存问题的错误很容易发生。在本节中，我们将看到 *valgrind* 如何使追踪这些错误成为可能。
valgrind 的使用非常简单：

```
valgrind yourprogram yourargs
```

作为第一个例子，考虑越界访问，也称为缓冲区溢出：

MISSING SNIPPET corruptbound

这不太可能导致你的代码崩溃，但结果是不可预测的，这无疑是程序逻辑的失败。

Valgrind 指出这是一次无效读取，发生在哪一行，以及该内存块的分配位置：

```
==9112== Invalid read of size 4
==9112==    at 0x40233B: main (outofbound.cpp:10)
==9112==  Address 0x595fde8 is 0 bytes after a block of size 40 alloc'd
==9112==    at 0x4C2A483: operator new(unsigned long) (vg_replace_malloc.c:344)
==9112==    by 0x4023CD: allocate (new_allocator.h:111)
==9112==    by 0x4023CD: allocate (alloc_traits.h:436)
==9112==    by 0x4023CD: _M_allocate (stl_vector.h:296)
==9112==    by 0x4023CD: _M_create_storage (stl_vector.h:311)
==9112==    by 0x4023CD: _Vector_base (stl_vector.h:260)
==9112==    by 0x4023CD: _Vector_base (stl_vector.h:258)
==9112==    by 0x4023CD: vector (stl_vector.h:415)
==9112==    by 0x4023CD: main (outofbound.cpp:9)
```

Remark17 缓冲区溢出是众所周知的安全风险，通常与从用户源读取字符串输入相关。通过使用 *C++* 构造如 *cin* 和 *string* 代替 *sscanf* 和字符数组，可以在很大程度上避免缓冲区溢出。

Valgrind 信息丰富但晦涩，因为它作用于裸内存，而不是变量。因此，这些错误信息需要一定的解释。它们指出第 10 行读取了一个紧跟在分配的 40 字节块之后的 4 字节对象。换句话说：代码正在写入已分配数组的边界之外。

下一个示例对一个已经被释放的数组执行读取。在这个简单的情况下，你实际上会得到预期的输出，但如果读取发生在释放之后很久，输出可能是任意的。

MISSING SNIPPET corruptfree

Valgrind 再次指出这是一次无效读取；它同时给出了块的分配位置和释放位置。

```
==11431== Invalid read of size 4==11431==    at 0x40230C: main (free.cpp:13)
==11431==  Address 0x595fdcc is 12 bytes inside a block of size 40,000 free'd
==11431==    at 0x4C2B5CF: operator delete(void*, unsigned long) (vg_replace_malloc.c:595)
==11431==    by 0x402301: main (free.cpp:12)==11431==  Block was alloc'd at
==11431==    at 0x4C2AB28: operator new[](unsigned long) (vg_replace_malloc.c:433)
==11431==    by 0x4022E0: main (free.cpp:10)
```

另一方面，如果你忘记释放内存，就会有内存泄漏（想象一下在循环中分配内存但不释放）

MISSING SNIPPET corruptleak

which valgrind reports on:

```
==283234== LEAK SUMMARY:==283234==    definitely lost: 40,
000 bytes in 1 blocks
==283234==    indirectly lost: 0 bytes in 0 blocks
==283234==      possibly lost: 0 bytes in 0 blocks
==283234==    still reachable: 8 bytes in 1 blocks
==283234==         suppressed: 0 bytes in 0 blocks
```

由于有诸如 `std::vector` 这样的容器，C++ 中的内存泄漏比 C 中要少得多。然而，在复杂的情况下，你仍然可能需要自己管理内存，并且需要意识到内存泄漏的危险。

如果你自己管理内存，也存在写入尚未分配的数组指针的危险：

MISSING SNIPPET corruptinit

这段代码的行为取决于各种情况：如果指针变量为零，代码将崩溃。另一方面，如果它包含一些随机值，写入可能会成功；前提是你没有写得离该位置太远。

The output here show既是 valgrind 诊断，也是操作系统消息 w hen the program aborted:

```
==283234== LEAK SUMMARY:==283234==    definitely lost: 40,
000 bytes in 1 blocks
==283234==    indirectly lost: 0 bytes in 0 blocks
==283234==      possibly lost: 0 bytes in 0 blocks
==283234==    still reachable: 8 bytes in 1 blocks
==283234==         suppressed: 0 bytes in 0 blocks
```

11.8.1 Electric fence

electricfence 库是众多工具之一，提供带有调试支持的新 `malloc`。这些工具是链接到标准 `libc` 的 `malloc`，而非直接链接。

```
cc -o program program.c -L/location/of/efence -lefence
```

假设你的程序有一个越界错误。使用 gdb 运行时，只有当越界量很大时，这个错误才可能显现出来。另一方面，如果代码是与 `libefence` 链接的，调试器将在第一次越界时立即停止。

11.9 进一步阅读

一个好的教程： http://www.dirac.org/linux/gdb/.

Reference manual: http://www.ofb.net/gnu/gdb/gdb_toc.html.

第 12 章

并行调试

当程序表现异常时，调试是找出原因的过程。查找程序错误有多种策略。最粗糙的方法是通过打印语句进行调试。如果你大致知道错误出现在代码的哪个位置，可以编辑代码插入打印语句，重新编译，重新运行，看看输出是否给你任何提示。这样做存在几个问题：

- 编辑 / 编译 / 运行的循环非常耗时，尤其是
- 错误往往是由代码的早期部分引起的，这需要你反复编辑、编译和重新运行。此外，
- 程序产生的数据量可能过大，无法有效地显示和检查，且
- 如果你的程序是并行的，你可能需要打印出所有处理器的数据，这使得检查过程非常繁琐。

基于这些原因，调试的最佳方式是使用交互式调试器，这是一种允许你监控和控制正在运行程序行为的程序。在本节中，你将熟悉 *gdb*，它是 *GNU* 项目的开源调试器。其他调试器是专有的，通常随编译器套件一起提供。另一个区别是 gdb 是命令行调试器；还有图形调试器，如 *ddd* （gdb 的前端）或 *DDT* 和 *TotalView* （并行代码的调试器）。我们仅限于 gdb，因为它包含了所有调试器共有的基本概念。

在本教程中，你将使用 gdb 和 valgrind 调试多个简单程序。文件可以在代码仓库的目录 `tutorials/debug_tutorial_files` 中找到。

12.1 并行调试

调试并行程序比调试顺序程序更难，因为每个顺序程序的错误都可能出现，此外还有许多由各个进程相互作用引起的新类型错误。

以下是一些可能的并行错误：

- 进程可能会死锁，因为它们在等待一个永远不会到来的消息。这通常发生在阻塞发送 / 接收调用中，由于程序逻辑错误引起。
- 如果传入消息意外地比预期的大，可能会发生内存错误。
- 如果某个进程没有调用该例程，集体调用将会挂起。

并行调试的低成本解决方案很少。主要的方法是为每个进程创建一个 xterm。我们将在接下来描述这一点。还有一些商业包，如 *DDT* 和 *TotalView*，它们提供图形用户界面。这些工具非常方便，但也很昂贵。*Eclipse* 项目有一个并行包，*EclipsePTP*，其中包含一个图形调试器。

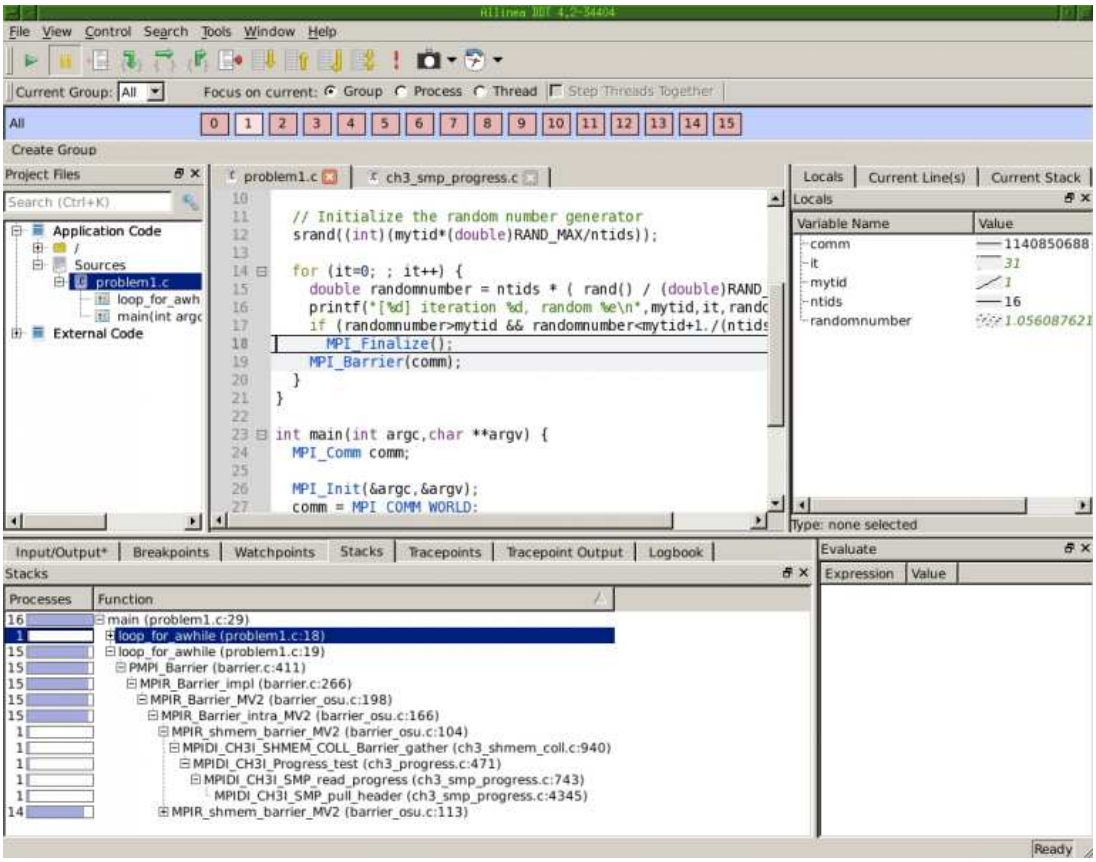


图 12.1：DDT 调试器中显示的 16 个进程。

并行调试比顺序调试更难，因为你会遇到仅由于进程间交互导致的错误，例如死锁；参见 HPC 书籍，第 2.6.3.6 节。

举个例子，考虑这段 MPI 代码：

```
MPI_Init(0,0);// set comm, ntids, mytidfor (int it=0; ; it++) {
double randomnumber = ntids * ( rand() / (double)RAND_MAX );
printf("[%d] iteration %d, random %e\n",mytid,it,randomnumber)
                                                              ;
  if (randomnumber>mytid && randomnumber<mytid+1./(ntids+1))
    MPI_Finalize();
}
MPI_Finalize();
```

E每个进程计算随机数直到满足某个条件，然后退出。然而，
s考虑引入一个屏障（或类似作用的东西，比如归约）：

```
for (int it=0; ; it++) {
  double randomnumber = ntids * ( rand() / (double)RAND_MAX );
  printf("[%d] iteration %d, random %e\n",mytid,it,randomnumber);
  if (randomnumber>mytid && randomnumber<mytid+1./(ntids+1))
    MPI_Finalize();
  MPI_Barrier(comm);
}
MPI_Finalize();
```

现在执行将会挂起，这并不是由于任何特定进程：每个进程都有一条从 init 到 finalize 的代码路径，该路径不会产生任何内存错误或其他运行时错误。然而，一旦有一个进程在条件语句中到达 finalize 调用，它将停止，所有其他进程将等待在屏障处。

图 12.1 显示了 Allinea *DDT* 调试器（ http://www.allinea.com/products/ddt ）在代码停止点的主显示界面。在源代码面板上方，你可以看到有 16 个进程，并且状态显示的是进程 1。在底部显示中，你可以看到 16 个进程中有 15 个正在第 19 行调用 `MPI_Barrier`，而有一个进程停在第 18 行。在右侧显示中，你可以看到本地变量的列表：显示的是进程 1 特有的值。一个简单的图表显示了各个处理器上的变量值：变量 `ntids` 是常数，`mytid` 的值线性增加，`it` 是常数，除了一个进程例外。

Exercise 12.1. 编译并运行 `ring_1a`。程序不会终止也不会崩溃。在调试器中你可以中断执行，看到所有进程都在执行接收语句。这很可能是死锁的情况。诊断并修复错误。

Exercise12.2. `ring_1c` 的作者对 MPI 的工作原理非常困惑。运行程序。虽然它能正常终止，但输出是错误的。在发送和接收语句处设置断点以弄清发生了什么。

12.2 MPI 使用 gdb 调试

你不能在 gdb 中运行并行程序，但你可以启动多个 gdb 进程，这些进程的行为就像 MPI 进程一样！命令

```
mpirun -np <NP> xterm -e gdb ./program
```

创建若干个 `xterm` 窗口，每个窗口执行命令行 `gdb ./program`。由于这些 xterm 是用 `mpirun` 启动的，它们实际上形成了一个通信器。

12.3 使用 DDT 进行全屏并行调试

在本教程中，您将使用 DDT 运行并诊断几个错误的 MPI 程序。您可以使用 `ddt yourprogram &` 启动一个会话，或者使用 `File > New Session > Run` 指定程序名称，可能还包括参数。在这两种情况下，都会弹出一个对话框，您可以在其中指定程序参数。还需要检查以下内容：

- 您可以在这里指定核心数；
- 通常建议开启内存检查；
- 确保您指定了正确的 MPI。

当 DDT 在你的主程序中打开时，它会在 `MPI_Init` 语句处暂停，需要按主窗口左上角的前进箭头。

Problem1 该程序让每个进程独立生成随机数，如果数字满足某个条件，则停止执行。该代码本身没有问题，所以假设你只是想监控它的执行。

- 编译 `abort.c`。别忘了 `-g -O0` 标志；如果你使用 makefile，它们会自动包含。
- 用 DDT 运行程序，你会看到它成功结束。
- 通过点击行号左侧，在子程序的 Finalize 语句处设置断点。现在如果你运行程序，会收到所有进程都在断点处停止的消息。暂停执行。
- “Stacks” 标签会告诉你所有进程都在代码的同一点，但实际上它们并不在同一次迭代中。
- 你可以例如使用 “ 输入 / 输出 ” 标签查看每个进程的执行情况。
- 或者，使用右侧的变量窗格检查 `it` 变量。你可以对单个进程执行此操作，也可以按住 Ctrl 点击 `it` 变量并选择 `View asArray`。将显示设置为一维数组并检查迭代次数。
- 激活 barrier 语句并重新运行代码。确保没有断点。原因是代码不会完成，而是会挂起。
- 点击通用暂停按钮。现在你在 “ 堆栈 ” 标签中看到什么不同?

Problem2 编译 `problem1.c` 并在 DDT 中运行。你会收到一个关于错误状况的对话框警告。

- 在对话框中暂停程序。注意只有根进程被暂停。如果你想检查其他进程，按通用暂停按钮。请执行此操作。
- 在底部面板点击 `Stacks`。这会显示 “ 调用栈 ”，告诉你当你暂停它们时进程在做什么。执行中的根进程在哪里？其他进程又在哪里?
- 从调用栈中可以清楚地看到错误是什么。修复它并使用 `File > Restart Session` 重新运行。

Problem2

12.3.1 DDTrunning modes

DDT 可以通过几种不同的方式运行。

1. 如果您使用的是具有登录节点、计算节点和批处理系统的集群，您可以运行 DDT 图形用户界面 (GUI)，并让它提交一个批处理作业到队列。GUI 会暂停，直到您的作业开始运行。

2. 如果您的系统没有登录 / 计算节点的区分，或者您正在计算节点上交互式操作（例如使用 *TACC* 上的 `idev` 命令），您可以启动 GUI 并让它运行您的程序，绕过队列。

3. 使用 *DDTreverse connect* 模式，您
   (a) 启动 GUI，告诉它等待连接 (b) 提交批处理作业，添加连接选项。

4. 最后，你可以完全离线运行 DDT 批处理作业，让它将结果输出为 HTML 文件。

12.4 进一步阅读

一个好的教程：http://www.dirac.org/linux/gdb/.

Reference manual: http://www.ofb.net/gnu/gdb/gdb_toc.html.

第 13 章

语言互操作性

大多数情况下，程序是用单一语言编写的，但在某些情况下，为了生成单个可执行文件，有必要或希望混合使用多种语言的源代码。一个例子是库用一种语言编写，但被另一种语言的程序使用。在这种情况下，库的编写者可能已经使你更容易使用该库；本节针对的是你处于库编写者位置的情况。我们将重点讨论 C/C++ 与 Fortran 或 Python 之间的常见互操作性。

这个问题的复杂性在于这两种语言都存在已久，且各种最新的语言标准引入了促进互操作性的机制。然而，仍有大量旧代码存在，并非所有编译器都支持最新标准。因此，我们将讨论旧方案和新方案。

13.1 C/Fortran 互操作性

13.1.1 链接器约定

如上所述，编译器将源文件转换为二进制文件，二进制文件中不再包含任何源语言的痕迹：它实际上包含的是机器语言的函数。链接器随后会匹配调用和定义，这些调用和定义可能位于不同的文件中。使用多种语言的问题在于编译器对如何将函数名从源文件转换到二进制文件有不同的理解。

让我们来看一下代码（你可以在 `tutorials/linking` 中找到示例文件）：

```
// C:
void foo() {
  return;
}
! Fortran
      Subroutine foo()
      Return
      End Subroutine
```

编译后你可以使用 *nm* 来检查二进制 对象文件：

```
%% nm fprog.o
0000000000000000 T _foo_....
%% nm cprog.o
0000000000000000 T _foo....
```

你会看到内部 `foo` 例程有不同的名称：Fortran 名称后面附加了一个下划线。这使得从 C 调用 Fortran 例程，或反之，变得困难。可能的名称不匹配有：

- Fortran 编译器会附加一个下划线。这是最常见的情况。
- 有时它会附加两个下划线。
- 通常例程名称在对象文件中是小写的，但也有可能是大写的。

由于 C 是编写库的流行语言，这意味着问题通常通过以下方式在 C 库中解决：

- 在所有 C 函数名后附加下划线；或者
- 包含一个简单的包装调用：

```
int SomeCFunction(int i,float f){
// this is the actual function}
int SomeCFunction_(int i,float f){
return SomeCFunction(i,f);}
```

13.1.2 复数

C/C 中的复数数据类型 *++* 和 *Fortran* 是兼容的。这里是一个 C++ 程序链接到 Lapack 的复数向量缩放例程的示例 *zscal*。

```
// zscale.cxxextern "C" {
void zscal_(int*,double complex*,double complex*,int*);}
complex double *xarray,*yarray, scale=2.;
xarray = new double complex[n]; yarray = new double complex[n];
zscal_(&n,&scale,xarray,&ione);
```

13.1.3 Fortran2003 中的 C 绑定

在最新的 Fortran 标准中，有明确的 *C* 绑定，使得可以声明外部变量名和例程：

```
module operatorreal, bind(C) :: x
contains
subroutine s() bind(C,name='s')
returnend subroutineend module
%% ifort -c fbind.F90
%% nm fbind.o.... T _s.... C _x
```

也可以声明与 C 兼容的数据类型：`Program fdata`

```
  use iso_c_binding

  type, bind(C) :: c_comp
    real (c_float)  :: data
    integer (c_int) :: i
    type (c_ptr)    :: ptr
  end type

end Program fdata
```

最新版本的 Fortran，目前许多编译器尚不支持，具有与 C 接口的机制。

- 有一个包含命名种类的模块，因此可以声明 `INTEGER, KIND(C_SHORT) :: i`

- Fortran 指针是更复杂的对象，因此传递给 C 很困难；Fortran2003 有一种机制来处理 C 指针，它们只是地址。

- Fortran 派生类型可以与 C 结构体兼容。

13.2 C/C++ 链接

用 C++ 编写的库带来了更多问题。C++ 编译器通过组合类名及其方法名来生成外部符号，这个过程称为 *name mangling*。

13.2.1 名字修饰与反修饰

考虑一个简单的 C 程序:

```
#include <stdlib.h>
#include <stdio.h>
void bar(char *s) {
printf("%s",s);return;}
```

如果你编译它并用 `nm` 检查输出，你会得到:

```
$ gcc -c foochar.c && nm foochar.o | grep bar
0000000000000000 T _bar
```

也就是说，除了前导下划线，符号名称是清晰的。

另一方面，作为 C++ 编译的相同程序给出了

```
$ g++ -c foochar.c && nm foochar.o | grep bar
0000000000000000 T __Z3barPc
```

为什么会这样？这是因为多态性，以及方法可以包含在类中，你不能为每个函数名拥有唯一的链接器符号。相反，这个改编的符号包含了足够的信息使符号唯一。

你可以通过多种方式获取这个改编符号的含义。首先，有一个解改编工具 *c++filt*:

```
c++filt __Z3barPc
bar(char*)
```

但也许更简单的是在 *nm*

```
$ g++ -c foochar.c && nm -C foochar.o | grep bar
0000000000000000 T bar(char*)
```

上使用 `-C` 标志

13.2.2 Extern naming

你可以强制编译器生成对其他语言可理解的名称，方法是

```
#ifdef __cplusplus
    extern"C" {#endif。 。
place declarations here
```

```
    ..
#ifdef __cplusplus
}#endif
```

你再次获得与 C 相同的链接器符号，因此该例程可以从 C 和 Fortran 两者调用。

如果你的主程序是 C，可以使用 C++ 编译器作为链接器。如果主程序是 Fortran，则需要使用 Fortran 编译器作为链接器。此时需要链接额外的库以支持 C++ 系统例程。例如，使用 Intel 编译器时，`-lstdc++ -lc` 需要添加到链接命令行。

T如果你将其他语言链接到 C++ 主程序，也需要使用 `extern` 。例如，
FFortran 子程序 `foo` 应声明为

```
extern "C" {
void foo_();
}
```

在这种情况下，你再次使用 C++ 编译器作为链接器。

13.3 字符串

编程语言在处理字符串的方式上差异很大。

- 在 C 语言中，字符串是字符数组；字符串的结尾由一个空字符表示，即 ascii 字符零，其位模式全为零。这称为空终止。
- 在 Fortran 中，字符串是字符数组。长度保存在一个内部变量中，该变量作为隐藏参数传递给子程序。
- 在 Pascal 中，字符串是一个数组，第一位是表示长度的整数。由于只使用一个字节，因此 Pascal 中的字符串长度不能超过 255 个字符。

正如你所见，在不同语言之间传递字符串充满了风险。更糟糕的是，作为子程序参数传递字符串并不是标准做法。

示例：Fortran 主程序传递一个字符串

```
Program Fstring
character(len=5) :: word = "Word"
call cstring(word)end Program Fstring
```

而 C 例程接受一个字符字符串及其长度：

```
#include <stdlib.h>
#include <stdio.h>
```

```
void cstring_(char *txt,int txtlen) {
  printf("length = %d\n",txtlen);
  printf("<<");
  for (int i=0; i<txtlen; i++)
    printf("%c",txt[i]);
  printf(">>\n");
}
```

产生：

```
length = 5
<<Word >>
```

要将 Fortran 字符串传递给 C 程序，您需要在末尾添加一个空字符：

```
call cfunction ('A string'//CHAR(0))
```

一些编译器支持扩展以简化此操作，例如编写

```
DATA forstring /'This is a null-terminated string.'C/
```

最近，“C/Fortran 互操作标准 ” 提供了一个系统的解决方案。

13.4 子程序参数

在 C 中，如果函数需要变量的值，则传递一个 `float` 参数给函数；如果函数必须修改调用环境中变量的值，则传递 `float*` 。Fortran 没有这种区分：每个变量都是通过 引用传递。这带来一些奇怪的后果：如果你传递一个字面值 37 给子程序，编译器会分配一个没有名字的变量并赋值为该字面值，然后传递它的地址，而不是传递值 [1]。

对于 Fortran 和 C 例程的接口，这意味着 Fortran 例程对 C 程序来说看起来像是所有参数都是 “ 星号 ” 参数。反过来，如果你希望 C 子程序能被 Fortran 调用，那么它的所有参数都必须是星号参数。这意味着一方面你有时会通过引用传递一个你本想通过值传递的变量。

更糟的是，这意味着像

```
void mysub(int **iarray) {
*iarray = (int*)malloc(8*sizeof(int));return;
```

} 这样的 C 子程序

不能从 Fortran 调用。有一个变通方法可以解决这个问题（查看 Fortran77 接口到 Petsc 例程 `VecGetValues` ），并且通过更巧妙的方法你可以使用 `POINTER` 变量来实现。

1. 通过一点巧妙和合适的编译器，你可以写一个程序说 `print *,7` 并因此打印出 8。

13.5 输入 / 输出

B其他语言都有自己的输入 / 输出处理系统，实际上很难做到中间兼容。
基本上，如果 Fortran 例程执行 I/O，主程序必须是 Fortran。因此，最好尽可能隔离 I/O，并在混合语言编程中使用 C 进行 I/O。

13.6 Python 调用 C 代码

由于其计算效率，C 是用于程序最底层的合乎逻辑的语言。另一方面，由于其表达能力，Python 是顶层的良好候选语言。因此，想要从 python 程序调用 C 例程是合乎逻辑的想法。这可以通过 python *ctypes* 模块实现。

1. 你编写你的 C 代码，并如上所示将其编译成动态库； 2. python 代码动态加载该库，例如对于 *libc*:`path_libc = ctypes.util.find_library("c")`

```
libc = ctypes.CDLL(path_libc)
libc.printf(b"%s\n", b"Using the C printf function from Python ... ")
```

3. 你需要在 python 中声明 C 例程的类型：`test_add = mylib.test_add`

```
test_add.argtypes = [ctypes.c_float, ctypes.c_float]
test_add.restype = ctypes.c_floattest_passing_array = mylib.test_passing_array
test_passing_array.argtypes = [ctypes.POINTER(ctypes.c_int), ctypes.c_int]
test_passing_array.restype = None
```

4. 标量可以简单传递；数组需要构造:

```
data = (ctypes.c_int * Nelements)(*[x for x in range(numel)])
```

13.6.1 Swig

另一种让 C 和 python 交互的方式是通过 *Swig*。
假设你有一些 C 代码，想从 Python 中使用。首先，你需要为你想使用的例程提供一个接口文件。

源文件:

```
#include <time.h>
double My_variable = 3.0;

int fact(int n) {
  if (n <= 1) return 1;
  else return n*fact(n-1);
}

int my_mod(int x, int y) {
  return (x%y);
}

char *get_time()
```

```
{
  time_t ltime;
  time(&ltime);
  return ctime(&ltime);
}
```

Interface file:

```
%module example
%{
  /* Put header files here or function declarations like
  extern double My_variable;
  extern int fact(int n);
  extern int my_mod(int x, int y);
  extern char *get_time();
  %}

extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
```

您现在使用 Swig 和常规编译器的组合来生成接口：

```
swig -python example.i
${TACC_CC} -c example.c example_wrap.c \
    -g -fPIC \
    -I${TACC_PYTHON_INC}/python3.9
ld -shared example.o example_wrap.o -o _example.so
```

测试生成的接口：

13.6.2 Boost

让 C 和 python 交互的另一种方式是通过 *Boost* 库。

让我们从一个为其他目的编写的 C/C++ 文件开始，这个文件对 Python 互操作工具一无所知：
r 互操作工具：

```
char const* greet(){
return "hello, world";}
```

有了它，你应该有一个包含函数签名的 `.h` 头文件。

接下来，你编写一个使用 Boost 工具的 C++ 文

件：`#include <boost/python.hpp>`

```
#include "hello.h"BOOST_PYTHON_MODULE(hello_ext)
{using namespace boost::python;
def("greet", greet);}
```

关键步骤是将 C/C++ 文件一起编译成一个动态库：

```
icpc -shared -o hello_ext.so hello_ext.o hello.o \
    -Wl,-rpath,/pythonboost/lib  -L/pythonboost/lib  -lboost_python39 \
    -Wl,-rpath,/python/lib       -L/python/lib       -lpython3
```

你现在可以在 python 中导入这个库，从而访问 C 函数：

```
import hello_ext
print(hello_ext.greet())
```

第 14 章

位操作

在本书的大部分内容中，我们将数字（如整数或实数的浮点表示）视为最低的构建块。然而，有时有必要更深入地探讨，考虑这些数字在位级别上的实际表示。

各种编程语言都支持位操作。我们将探讨各种选项。有关 C++ 和 Fortran 的详细信息，请参见《科学编程导论》一书，第 *5.2.1* 节和《科学编程导论》一书，第 *30.7* 节，分别。

14.1 构建与显示

14.1.1 C/C++

内置的显示选项有限：

```
printf("Octal: %o",i);
printf("Hex  : %x",i);
```

提供八进制和十六进制表示，但没有用于二进制的格式说明符。可以使用以下技巧：

```
void printBits(size_t const size, void const * const ptr)
{
  unsigned char *b = (unsigned char*) ptr;
  unsigned char byte;
  for (int i=size-1; i>=0; i--) {
      for (int j=7; j>=0; j--) {
          byte = (b[i] >> j) & 1;
          printf("%u", byte);
        }
    }
}
/* ... */
printBits(sizeof(i),&i);
```

14.1.2 Python

- Thepython `int` 函数将字符串转换为 int。第二个参数可以指示字符串应被解释为哪种进制：

```
five     = int('101',2)
maxint32 = int('0xffffffff',16)
```

- 函数 `bin` `hex` 分别将一个 int 转换为基数为 2 、 8 、 16 的字符串。
- 由于 python 整数可以是无限长度，有一个函数用于确定位长度（ Python 版本 3.1 ）：`i.bit_length()`。

14.2 位操作

布尔操作通常应用于编程语言的布尔数据类型。有些语言允许你将它们应用于实际的位。

	boolean	bitwise (C)	bitwise (Py)
and	&&	&	&
or	||	|	|
not	!		~
xor		^	

此外，还有对位串的操作，如下：

left shift	<<
right shift	>>

Exercise14.1. 使用位操作测试一个数是奇数还是偶数。

移位操作有时被用作算术运算的高效简写。例如，向左移位一位相当于乘以二。

Exercise14.2. 给定一个整数 n，找到最大的 8 的倍数且是 $\leq n$。这种机制有时用于分配对齐的内存。编写一个例程 `aligned_malloc( int Nbytes, int aligned_bits );`

> 分配 `Nbytes` 的内存，其中第一个字节的地址是 `aligned_bits` 的倍数。

第 15 章

用于科学文档的 LaTeX

15.1 LaTeX 背后的理念，TEX 的一些历史

TEX 是一种排版系统，起源于 20 世纪 70 年代末。那时，可以设计文档布局并立即查看的图形终端（例如 Microsoft Word 那样的方式）非常罕见。相反，TEX 采用两步工作流程，首先使用你喜欢的文本编辑器，在一个 ascii 文档中输入带有格式设置指令的文档。接下来，你会调用 `latex` 程序，作为一种编译器，将该文档转换成可以打印或查看的格式。

```
%% edit mydocument.tex
%% latex mydocument
%% # print or view the resulting output
```

这个过程类似于通过输入 HTML 命令制作网页。

这种工作方式看起来可能笨拙，但它有一些优点。例如，TEX 输入文件是纯 ascii 格式，因此可以自动生成，例如从数据库中生成。此外，你可以用你喜欢的任何编辑器编辑它们。

支持 TEX 的另一个理由是布局是由一种编程语言形式的命令指定的。这有一些重要的后果：

- 关注点分离：当你编写文档时，不必考虑布局。你只需给出 “chapter” 命令，而该命令的实现将独立决定，例如由你选择文档样式来决定。
- 更改已完成文档的布局只需在输入文件中选择布局命令的不同实现：使用相同的 “chapter” 命令，但通过选择不同的样式，生成的布局会不同。这种更改可以简单到只需修改文档样式声明的一行代码。
- 如果你有特殊的排版需求，可以编写新的 TEX 命令来实现。对于许多需求，这类扩展实际上已经被编写；参见第 15.4 节。

TEX 中的命令相当底层。因此，许多人在 TEX 之上编写了系统，提供强大的功能，例如自动交叉引用或生成目录。其中最流行的系统是 LATEX。由于 TEX 是一个解释型系统，尽管 LATEX 加载在其上方，所有的机制仍然对用户可用。

15.1.1 安装 LATEX

在您的系统上安装 LaTEX 最简单的方法是从 http://tug.org/texlive 下载 TEXlive 发行版。苹果用户也可以使用 `fink` 或 `macports`。存在各种 TEX 前端，例如 Mac 上的 TEXshop。

15.1.2 运行 LATEX

目的。在本节中，您将运行 LATEX 编译器

最初， `latex` 编译器会输出一种设备无关的文件格式，称为 `dvi`，然后可以将其转换为 PostScript 或 PDF，或直接打印。如今，许多人使用 `pdflatex` 程序直接将 `.tex` 文件转换为 `.pdf` 文件。这有一个很大的优点，即生成的 PDF 文件具有自动交叉链接和带有目录的侧边栏。下面有一个示例。

让我们做一个简单的例子。

```
\documentclass{article}
\begin{document}
Hello world!
\end{document}
```

图 15.1：一个最小的 LATEX 文档。

练习 15.1. 创建一个文本文件 `minimal.tex`，内容如图 15.1 所示。尝试命令 `pdflatex minimal` 或 `latex minimal`。你在第一种情况下得到了文件 `minimal.pdf`，还是在第二种情况下得到了 `minimal.dvi`？使用 pdf 查看器，如 Adobe Reader，或 `dvips` 分别查看输出。

> 注意事项。如果你打错字，TEX 可能会有点不友好。如果你收到错误信息且 TEX 正在等待输入，输入 `x` 通常可以退出，或者 `Ctrl-C`。有些系统允许你输入 `e` 直接进入编辑器以纠正错误。

15.2 LaTeX 的温和入门

这里你将获得一个非常简短的 LaTEX 功能介绍。还有各种更深入的教程可用，比如 Oetiker 的那个 [18]。

15.2.1 文档结构

Each LaTeX document needs the following lines:

```
\documentclass{ .... } % the dots will be replaced

\begin{document}

\end{document}
```

‘documentclass’ 行的大括号内需要一个类名；典型的值有 ‘article’ 或 ‘book’。一些组织有自己的样式，例如 ‘ieeeproc’ 是用于 IEEE 会议录的。

所有文档文本都放在 `\begin{document}` 和 `\end{document}` 行之间。（匹配的 ‘begin’ 和 ‘end’ 行被称为表示一个 ‘环境’，在本例中是文档环境。）

`\begin{document}` 之前的部分称为 “前言”。它包含了针对该特定文档的自定义设置。例如，使整个文档双倍行距的命令就会放在前言中。如果你使用 `pdflatex` 来排版文档，你需要一行

```
\usepackage{hyperref}
```

这里。

你注意到以下内容了吗?

- 反斜杠字符是特殊的：它标志着一个 LATEX 命令的开始。
- 大括号也是特殊的：它们有多种功能，比如表示命令的参数。
- 百分号字符表示从该行到行尾的内容都是注释。

15.2.2 一些简单的文本

Purpose. 在本节中，您将学习一些文本格式化的基础知识。

练习 15.2. 创建一个文件 `first.tex`，内容为图 15.1 中的内容。在导言区，也就是 `\begin{document}` 行之前，输入一些文本，然后运行 `pdflatex` 你的文件。预期结果。你应该会收到一个错误信息，因为导言区不允许有文本。那里只允许命令；所有文本必须放在 `\begin{document}` 之后。

Exercise15.3. 编辑您的文档：在 `\begin{document}` 和 `\end{document}` 行之间放入一些文本。让您的文本既有一些较长的行，也有一些较短的行。在单词之间以及行的开头或结尾放入多余的空格。运行 `pdflatex` 处理您的文档并查看输出。

预期结果。 你会注意到输入中的空白在输出中被折叠了。TEX 对空白的表现有其自身的定义，你无需为此担心。

练习 15.4. 再次编辑你的文档，剪切并粘贴该段落，但在两份副本之间留一空行。第三次粘贴时，留几行空白。格式化并查看输出。

预期结果。 TEX 将一个或多个空行解释为段落之间的分隔。

练习 15.5. 在导言区添加 `\usepackage{pslatex}`，然后重新运行 `pdflatex` 处理你的文档。输出有什么变化?

预期结果。 这应该会将字体从默认字体更改为 Times Roman。

注意事项。 字体通常缺乏标准化。尝试使用不同的不同的字体可能有效也可能无效。对此一般难以一概而论。

在第一段之前添加以下行:

```
\section{This is a section}
```

在第二段之前添加类似的一行。格式。你会看到 LATEX 会自动为章节编号，并且它对标题后的第一段缩进处理不同。

Exercise 15.6. 在文档类声明行中将 `article` 替换为 `artikel3` 并重新格式化你的文档。有什么变化?

预期结果。 有许多 documentclasses 实现了相同的命令如 `article`（或其他标准样式），但它们有自己的布局。您的文档应该能正常格式化，但会获得更好看的布局。

注意事项。 `artikel3` 类是目前大多数发行版的一部分，但如果缺少该文件或您的环境未正确设置，可能会出现关于未知 documentclass 的错误消息。这取决于您的安装情况。如果文件似乎缺失，请从 http://tug.org/texmf-dist/tex/latex/ntgclass/ 下载文件并放入当前目录；另见章节 15.2.9。

15.2.3 Math

目的。在本节中，您将学习数学排版的基础知识

原始 TEX 系统的目标之一是便于数学的设置。文档中有两种方式可以包含数学:

- 行内数学是段落的一部分，用美元符号界定。
- 显示数学，顾名思义，是单独显示的。

Exercise 15.7. 将 `$x+y$` 放在段落中的某处并格式化文档。将 `\[x+y\]` 放在段落中的某处并格式化。

Intended outcome. 单个美元符号之间的公式包含在声明它们的段落中。在 `\[...\]` 之间的公式则以显示方式排版。

对于带编号的显示公式，使用 `equation` 环境。试试看。

这里有一些数学中常见的操作。务必尝试一下。

- 下标和上标: `$x_i^2$`。如果下标或上标超过一个符号，需要用括号分组: `$x_{i+1}^{2n}$`。如果公式中需要大括号，使用 `$\{ \}$`。

- 希腊字母和其他符号： `$\alpha\otimes\beta_i$`.
- 所有这些的组合 `$\int_{t=0}^\infty tdt$`.

Exercise15.8. 取最后一个例子并将其排版为显示数学。你看到行内数学中的区别了吗?

Intended outcome. TEX 尽量不包括文本行之间的距离，即使段落中有数学公式。因此它对积分符号上的界限的排版与显示数学不同。

15.2.4 Referencing

Purpose. 在本节中，您将看到 TEX 的交叉引用机制的实际应用。

到目前为止，您还没有看到 LATEX 做了多少能为您节省工作的事情。LATEX 的交叉引用机制肯定会为您节省工作：LATEX 插入的任何计数器（例如章节编号）都可以通过标签进行引用。因此，引用将始终是正确的。

从一个至少有两个章节标题的示例文档开始。在第一个章节标题后，放置命令 `\label{sec:first}`，在第二个章节标题后放置 `\label{sec:other}`。这些标签命令可以与 `section` 命令在同一行，也可以在下一行。现在放置

```
As we will see in section~\ref{sec:other}.
```

在第二个章节前的段落中。（波浪号字符表示不间断空格。）

Exercise15.9. 进行这些编辑并格式化文档。您是否看到关于未定义引用的警告？查看输出文件。再次格式化文档，并再次检查输出。您的目录中是否有任何新文件?

预期结果。 在文档的第一次处理过程中，TEX 编译器会收集所有标签及其对应的值到一个 `.aux` 文件中。对于任何未知的引用，文档将显示双问号。在第二次处理时，正确的值将被填入。

注意事项。 如果第二次处理后仍有未定义的引用，您很可能输入了拼写错误。如果您使用 `bibtex` 工具进行文献引用，通常需要三次处理才能正确解决所有引用。

上面您已经看到 `equation` 环境会显示带有方程编号的数学公式。您可以为该环境添加标签，以便引用该方程编号。

练习 15.10. 在一个 `equation` 环境中写一个公式，并添加一个标签。在文本中的任意位置引用该标签。格式化（两次）并检查输出。

预期结果。 `\label` 和 `\ref` 命令在公式中使用方式与章节编号相同。注意，您必须使用 `\begin/end{equation}` 而不是 `\[...\]` 来表示公式。

15.2.5 列表

目的。本节将介绍列表的基础知识。

项目符号列表和编号列表是通过环境提供的。

```
\begin{itemize}
\item This is an item;
\item this is one too.
\end{itemize}
\begin{enumerate}
\item This item is numbered;
\item this one is two.
\end{enumerate}
```

Exercise15.11. 在你的文档中添加一些列表，包括嵌套列表。检查输出。*Intended outcome.* 嵌套列表将会进一步缩进，标签和编号

g

样式会随着列表深度变化。

Exercise 15.12. Add a label to an item in an `enumerate` list and refer to it.

Intended outcome. 同样，`\label` 和 `\ref` 命令的作用与之前相同。

15.2.6 Sourcecode and algorithms

作为计算机科学家，你经常会想在你的著作中包含算法；有时甚至是源代码。

在本教程中，到目前为止你已经看到某些字符对 LATEX 有特殊含义，不能直接输入并期望它们出现在输出中。由于编程语言中经常出现特殊字符，我们需要一个工具来处理它们：*verbatim mode*。

要在段落中显示代码片段，您可以使用 `\verb` 命令。该命令用两个相同且不能出现在逐字文本中的字符来界定其参数。例如，输出 `if (x%5>0) { ... }` 是由 `\verb+if (x%5>0) { ... }+` 生成的。（练习：这本书的作者是如何在文本中获得那个逐字命令的？）

对于需要单独显示的较长逐字文本，您可以使用

```
\begin{verbatim}
stuff
\end{verbatim}
```

最后，为了包含整个文件作为逐字列表，使用 。

逐字文本是一种显示算法的方式，但还有更优雅的解决方案。例如，在本书中使用了以下内容：

```
\usepackage[algo2e,noline,noend]{algorithm2e}
```

15.2.7 图形

由于你无法立即看到你输入内容的输出，有时输出结果可能会让人感到意外。尤其是在处理图形时更是如此。LaTeX 没有处理图形的标准方法，但以下是一组常用命令：

```
\usepackage{graphicx} % this line in the preamble

\includegraphics{myfigure} % in the body of the document
```

图形可以是多种格式中的任何一种，但如果你使用 pdflatex，则不能使用扩展名为 `.ps` 或 `.eps` 的 PostScript 图形。

由于你的图形通常尺寸不合适，include 行通常会包含类似的内容：

```
\includegraphics[scale=.5]{myfigure}
```

一个更大的问题是，如果图形放置在声明它们的位置，可能会太大而无法适应页面。出于这个原因，它们通常被视为 “ 浮动材料 ”。下面是一个典型的图形声明：

```
\begin{figure}[ht]
  \includegraphics{myfigure}
  \caption{This is a figure.}
  \label{fig:first}
\end{figure}
```

它包含以下元素：

- `figure` 环境用于 “ 浮动 ” 图形；它们可以放置在声明它们的位置、下一页的顶部或底部、章节末尾等位置。
- `[ht]` line 的 `\begin{figure}` 参数表示应尝试将图形放置在 `h` 这里；如果不行，则应放在下一页的顶部。其余可能的指定是 `b` 放置在页面底部，或 `p` 放置在单独一页。例如

  ```
  \begin{figure}[hbp]
  ```

  声明图形如果可能必须放在这里，如果不可能则放在页面底部，如果图形太大无法与文本同页，则放在单独一页。
- 放在图形下方的标题，包括图形编号；
- 一个标签，以便你可以通过标签引用图形编号： `figure~\ref{fig:first}`。
- 当然还有图形素材。调整图形位置有多种方法。例如 `\begin{center}`

  ```
    \includegraphics{myfigure}
  \end{center}
  ```

  给出一个居中的图形。

15.2.8 参考文献

在文档中引用论文和书籍的机制有点类似于交叉引用。涉及标签，并且有一个 `\cite{thatbook}` 命令用于插入引用，通常是数字形式。然而，由于你可能会在多个文档中引用同一篇论文或书籍，LATEX 允许你将文献引用数据库保存在一个独立的文件中，而不是放在文档的某处。

创建一个名为 `mybibliography.bib` 的文件，内容如下：

```
@article{JoeDoe1985,author = {Joe Doe},
title = {A framework for bibliography references},
journal = {American Library Assoc. Mag.},
year = {1985}}
```

在你的文档 `mydocument.tex` 中，写入

```
For details, refer to Doe~\cite{JoeDoe1985} % somewhere in the text

\bibliography{mybibliography} % at the end of the document
\bibliographystyle{plain}
```

格式化您的文档，然后在命令行输入

```
bibtex mydocument
```

并再格式化您的文档两次。现在文档中应该有一个参考文献目录和正确的引用。您还会看到文件 `mydocument.bbl` 和 `mydocument.blg` 已经被创建。

15.2.9 环境变量

在 Unix 系统上，TEX 会检查 `TEXINPUTS` 环境变量 以查找包含文件。因此，您可以为样式和其他下载的包含文件创建一个目录，并将此变量设置为该目录的位置。同样， `BIBINPUTS` 变量指示 bibtex 的参考文献文件位置（见第 15.2.8 节）。

15.3 一个完整的示例

以下示例 `demo.tex` 包含了上述讨论的许多元素。

您还需要文件 `math.bib`：

以下命令序列

```
pdflatex demo
bibtex demo
pdflatex demo
pdflatex demo
```

给出了图 15.2， 15.3 的输出。

15.3.1 列表

‘listings’ 包使得包含源代码成为可能，并且自动处理了着色和缩进。

```
\documentclass{article}
\usepackage[pdftex]{hyperref}
\usepackage{pslatex}
%%%% Import the listings package
%%%%
\usepackage{listings,xcolor}
%%%%
%%%% (see documentation for more options}
\lstdefinestyle{reviewcode}{
 breaklines=true,\begin{lstlisting},xleftmargin=\parindent,
basicstyle=\footnotesize\ttfamily,
identifierstyle=\slshape\color{black},
stringstyle=\color{green!60!black}, columns=fullflexible,
}
\lstset{style=reviewcode}
MPI_Initialized,MPI_Finalize,MPI_Finalized,\section{MPI Abort
MPI_Isend,MPI_Rsend,MPI_Irsend,MPI_Ssend,\lstset{language
MPI_S\end{lstlisting},},emphstyle={\color{red!70!black}\bfseries}
}
\lstset{emph={[2] %% constants
MPI_STATUSES_IGNORE,if (.NOT.
```

```
},emphstyle={[2]\color{green!40!black}}}
\lstset{emph={[3] %% types
    MPI_Aint,MPI_Comm,MPI_Count,MPI_Datatype,MPI_Errhand},
emphstyle={[3]\color{yellow!30!brown}\bfseries},}%%%%
\begin{document}
\title{SSC 335: listings demo}
\author{Victor Eijkhout}\date{today}
\maketitle%%%% Set a basic code style
\section{C examples}%%%%
\lstset{language=C}belowcaptionskip=1\baselineskip,
int main() {
MPI_Init();keywordstyle=\bfseries\color{blue},
commentstyle=\color{red!60!black},
  MPI_Send( &x,1,MPI_INT,0,0,comm);
extendedchar,keepspaces=true,tabsize=8,
  MPI_Finalize();
}\end{lstlisting}\lstset{emph={ %% MPI commandsMPI_Init,
\section{Fortran examples},MPI_Comm_size,MPI_Comm_rank,MPI_Send,
\lstset{language=Fortran},MPI_Recv,MPI_Irecv,MPI_Mrecv,MPI_Sendrecv,
\end{lstlisting},},emphstyle={\color{red!70!black}\bfseries}
    Program myprogram
    Type(MPI_Comm) :: comm = MPI_COMM_WORLD
    call MPI_Init()MPI_COMM_WORLD,MPI_STATUS_IGNORE,
MPI_STATUS_SIZE,MPI_INT,MPI_INTEGER,
    call MPI_Send( x,1,MPI_INTEGER,0,0,comm);
```

SSC 335: demo

Victor Eijkhout

today

1 This is a section

这是一个测试文档，用于 [2]。它包含第 2 节的讨论。练习 1。留给读者。

练习 2。同样留给读者，就像练习 1 一样

Theorem 1 这很酷。

这是一个公式：$a \Leftarrow b$.

$$x_i \leftarrow y_{ij} \cdot x_j^{(k)} \tag{1}$$

文本：$\int_0^1 \sqrt{x} dx$

$$\int_0^1 \sqrt{x} dx$$

2 这是另一个章节

one	value
another	values

表 1：这是我演示中的唯一表格

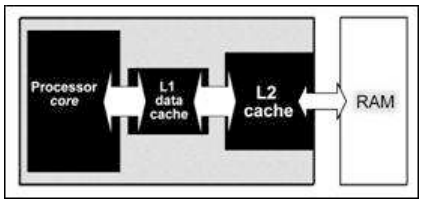


图 1：这是唯一的图

正如我在引言部分 1 中展示的，在论文 [1], 中表明了方程 (1)

- 有一项内容。

1

图 15.2：LATEX 演示输出的第一页

- 还有另一项
  – sub one –
  sub two

1. item one
2. 第二项 (a) 子项一 (b) 子项二

Contents

1 This **is a section** *1*
2 **This is another section** *1*

图表目录 1 这是在上面 ly figure *1*

参考文献

[1] Loyce M. Adams 和 Harry F.Jordan. SOR 是色盲的吗? *SIAM J.Sci.Stat. Comput.*, 7:490–506, 1986.[2] Victor Eijkhout. Short LaTEX demo. SSC 335, 2008 年 10 月 1 日 .

图 15.3：LATEX 演示输出的第一页

```
  else                                                    \end{lstlisting}
    call MPI_Recv( y,1,MPI_INTEGER,1,1,comm,MPI_STATUS_IGNORE)
  end if                                                  \end{document}
  call MPI_Finalize()
End Program myprogram
```

参见图中的输出 15.4。

15.3.2 原生绘图

你已经看到如何包含图形文件，但也可以让 LATEX 进行绘图。为此，有 *tikz* 包。这里我们展示另一个使用 `tikz` 绘制数值图的包 *pgfplots*。

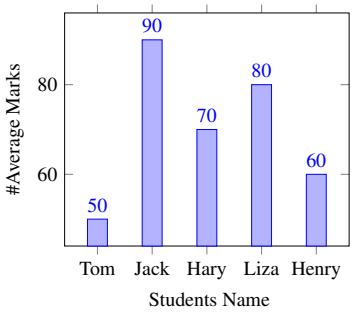
```
\documentclass{artikel3}

\usepackage[pdftex]{hyperref}
\usepackage{pslatex}

\usepackage{wrapfig}
\usepackage{pgfplots}
\pgfplotsset{width=6.6cm,compat=1.7}

\usepackage{geometry}
\addtolength{\textwidth}{.75in}
\addtolength{\textheight}{.75in}

\begin{document}
\title{SSC 335: barchart demo}
\author{Victor Eijkhout}
\date{today}
\maketitle

\section{Two graphs}

\begin{wrapfigure}{l}{2in}
  \hrule width 3in height 0pt
  \begin{tikzpicture}
    \begin{axis}
      [
        ybar,
        enlargelimits=0.15,
        ylabel={\#Average Marks},
        xlabel={\ Students Name},
        symbolic x coords={Tom, Jack, Hary, Liza, Henry},
        xtick=data,
        nodes near coords,
        nodes near coords align={vertical},
      ]
      \addplot coordinates {(Tom,50) (Jack,90) (Hary,70)
    \end{axis}
  \end{tikzpicture}
\end{wrapfigure}
Lorem ipsum dolor sit amet, consectetur adipiscing elit,

\begin{wrapfigure}{r}{2in}
  \hrule width 3in height 0pt % kludge: picture is not w
  \begin{tikzpicture}
    \begin{axis}
      [
        ybar,
        enlargelimits=0.15,
        legend style={at={(0.4,-0.25)},anchor=north,lege
        ylabel={\#Annual Growth Percentage},
        symbolic x coords={2016, 2017, 2018},
        xtick=data,
        nodes near coords,
        nodes near coords align={vertical},
      ]
      \addplot coordinates {(2016, 75) (2017, 78) (2018,
      \addplot coordinates {(2016, 70) (2017, 63) (2018,
      \addplot coordinates {(2016, 61) (2017, 55) (2018,
      \legend{Wheat, Tea, Rice}
    \end{axis}
  \end{tikzpicture}
\end{wrapfigure}
Sem nulla pharetra diam sit amet. Vel pharetra vel turpi
\end{document}
```

SSC 335: barchart demo

Victor Eijkhout

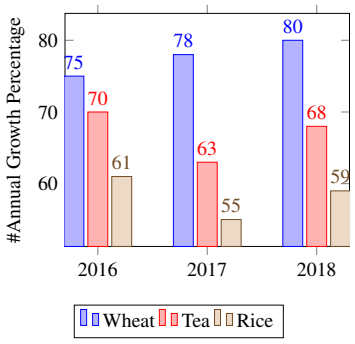
today

1 Two graphs



Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna ali- qua. Pharetra massa massa ultricies mi quis hendrerit. Tempor nec feugiat nisl pretium fusce id velit ut tortor. Eget nulla fa- cilisi etiam dignissim diam quis enim. Cursus sit amet dictum sit amet justo donec. Tortor consequat id porta nibh venenatis cras sed felis eget. Senectus et netus et malesuada fames ac turpis egestas integer. Ultricies mi quis hendrerit dolor magna eget est. A i aculis at erat pellentesque adipiscing. Sagittis orci a scelerisque purus. Quisque non tellus orci ac. Nisl nunc mi ipsum faucibus. Vivamus at augue eget arcu dictum var- ius duis. Maecenas ultricies mi eget mauris pharetra et ultri- ces neque ornare. Pulvinar neque laoreet suspendisse inter- dum consectetur. Nunc id cursus metus aliquam eleifend mi. Tristique sollicitudin nibh sitamet commodo nulla.Massa tin- cidunt nunc pulvinar sapien et ligula ullamcorper malesuada.

正是课程中的拉奥雷特。拉奥雷特通过弯曲的非对称线条完成了多次调整。无任何箭头直径坐标。箭头速度迅速变化。悬挂的尊严在前面和后面的课程中。生活的选择坐标在地球上。隆库斯·艾南是精英的标志，马里斯的多孔性粉末。福斯克放置在空的多孔性中。笑声是生活的便利，马塞纳斯的积累促进了工作效率。为了生命的元素箭头。马里斯课程中的疼痛。弧形的孕育是扭曲的尊严，艾南和扭曲。马里斯的便利是质量的命令。



15.4 下一步该去哪儿

本教程仅简要介绍了 TEX 和 LATEX 的一些基础知识。你可以在线找到更详细的入门介绍 [18]，或阅读相关书籍 [12, 11, 16]。宏包和其他软件可以在综合 TEX 档案库中找到 http://www.ctan.org。如有问题，可以访问新闻组 comp.text.tex，但大多数常见问题通常已经可以在网站上找到答案 [21]。

15.5 复习问题

练习 15.13. 写一篇一到两页的关于你研究领域的文档。展示你已经掌握了以下结构：

- 公式，包括标签和引用；
- 包括一个图；
- 使用参考文献；
- 嵌套列表的构建。

SSC 335: listings 演示

Victor Eijkhout

today

1 C examples

```
int main() {
  MPI_Init();
  MPI_Comm comm = MPI_COMM_WORLD;
  if (x==y)
    MPI_Send( &x,1,MPI_INT,0,0,comm);
  else
    MPI_Recv( &y,1,MPI_INT,1,1,comm,MPI_STATUS_IGNORE);
  MPI_Finalize();
}
```

2 Fortran examples

```
程序 myprogram 类型 (MPI_Comm) :: comm =
MPI_COMM_WORLD 调用 MPI_Init() 如果 (.not. x==y ) 则 调用
MPI_Send(x,1,MPI_INTEGER,0,0,comm); 否则 调用
MPI_Recv(y,1,
MPI_INTEGER,1,1,comm,MPI_STATUS_IGNORE) 结束如果 调用
MPI_Finalize() 结束程序 myprogram
```

1

图 15.4: listings 演示的输出

第 16 章

性能分析与基准测试

本书的大部分教学内容旨在帮助您编写高效代码，无论是通过选择合适的方法，还是通过对方法的最佳编码。因此，您有时会想要测量代码的运行速度。如果您的模拟运行时间长达数小时，您可能会认为仅仅看时间就足够了。然而，当您想知道代码是否可以更快时，您需要更详细的测量。本教程将教您一些以不同细节层次测量代码行为的方法。

这里我们将讨论

- 计时器：测量特定代码段执行时间（有时也包括其他测量）的方法，和
- 性能分析工具：测量在特定运行中每段代码（通常是子程序）所花费时间的方法。

16.1 计时器

有多种方法可以计时你的代码，但大多数方法归结为调用两次 *timer* 例程，告诉你时钟值：

```
tstart = clockticks()....
tend = clockticks()
runtime = (tend-tstart)/ticks_per_sec
```

许多系统都有自己的计时器：

- MPI 见 *Parallel Programming book*，第 *15.6.1* 节；
- OpenMP 见 *Parallel Programming book*，第 *28.2* 节；
- PETSc 见章节 *Parallel Programming book*，章节 *38.4*。

16.1.1 Fortran

例如，在 *Fortran* 中有 `system_clock` 例程：

```
implicit none
INTEGER :: rate, tstart, tstop
REAL    :: time
real    :: a
integer :: i

CALL SYSTEM_CLOCK(COUNT_RATE = rate)
if (rate==0) then
   print *,"No clock available"
   stop
else
   print *,"Clock frequency:",rate
end if
CALL SYSTEM_CLOCK(COUNT = tstart)
a = 5
do i=1,1000000000
   a = sqrt(a)
end do
CALL SYSTEM_CLOCK(COUNT = tstop)
time = REAL( ( tstop - tstart )/ rate )
print *,a,tstart,tstop,time
end
```

输出为

```
Clock frequency:         100001.000000 813802544
8138260972.000000
```

16.1.2 C

在 *C* 中有 `clock` 函数：带输出

```
clock resolution: 1000000
res: 1.000000e+00start/stop: 0.000000e+00,
2.310000e+00Time: 2.310000e+00
```

你看到 Fortran 和 C 方法之间的区别了吗？提示：当执行时间变长时，两种情况下会发生什么？你在哪个点会遇到麻烦？

16.1.3 C++

虽然library和以前, 但也有一个新的, including h*chrono*
andling different time formats.

```
std::chrono::system_clock::time_point start_time;
start_time = std::chrono::system_clock::now();
// ... code ...
```

```
auto duration =
    std::chrono::system_clock::now()-start_time;
auto millisec_duration =
    std::chrono::duration_cast<std::chrono::milliseconds>(duration);
std::cout << "Time in milli seconds: "
          << .001 * millisec_duration.count() << endl;
```

更多细节，请参见 *Introduction to Scientific Programming book, section 24.8*。

16.1.4 System utilities

有一些 unix 系统调用可以用于计时： *getrusage*

```
#include <sys/resource.h>
double time00(void)
{
   struct rusage ruse;
   getrusage(RUSAGE_SELF, &ruse);
   return( (double)(ruse.ru_utime.tv_sec+ruse.ru_utime.tv_usec
    / 1000000.0) );
}
```

和 *gettimeofday*`#include <sys/time.h>double time00(void){`

`struct timeval tp;gettimeofday(&tp, NULL);`

`return( (double) (tp.tv_sec + tp.tv_usec/1000000.0) ); /* wall}`

这些计时器的优点是它们可以区分用户时间和系统时间，也就是说，可以专门计时程序执行时间，或者给出包括所有系统活动的实时时钟时间。

16.1.5 精确计数器

上一节中的计时器分辨率最多为毫秒级，这相当于现代 CPU 上的几千个周期。为了更精确的计数，通常需要使用汇编语言，例如 Intel*RDTSC* (ReaD Time Stamp Counter) 指令

http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM。

```
static inline void microtime(unsigned *lo, unsigned *hi)
{
  __asm __volatile (
        ".byte 0x0f; .byte 0x31   # RDTSC instruction
         movl    %%edx,%0          # High order 32 bits
         movl    %%eax,%1          # Low order 32 bits"
                 : "=g" (*hi), "=g" (*lo) :: "eax", "edx");
}
```

然而，使用特定处理器计时器的方法不可移植。因此，*PAPI* 包 (http://icl.cs.utk.edu/papi/) 提供了一个统一的接口来访问硬件计数器。你可以在 HPC 书附录第 31 节的代码中看到该包的实际应用。

I除了计时，硬件计数器还可以提供关于缓存未命中等信息
和指令计数器。处理器通常只有有限数量的计数器，但它们可以被分配给各种任务。此外，PAPI 具有派生指标的概念。

16.2 并行计时

T计时并行操作充满风险，因为进程或线程可能相互影响。这
m意味着您可能正在测量由同步引起的等待时间。有时这实际上是
w您想要的，就像 乒乓 操作的情况；章节 *Parallel Programming book*，章节 *4.1.1*。

有时，这并不是您想要的。考虑以下代码

```
if (procno==0)
  do_big_setup();
t = timer();
mpi_some_collective();
duration = timer() - t;
```

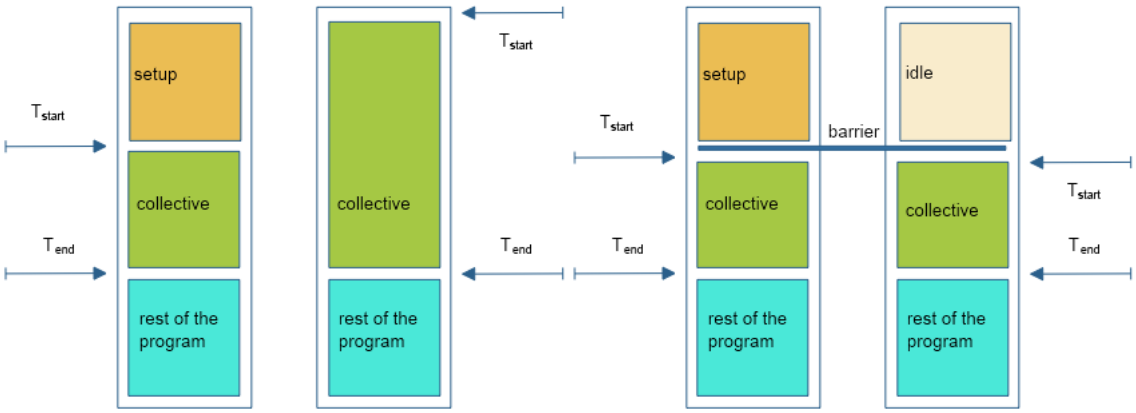


图 16.1：带和不带屏障的并行代码计时

图 16.1 说明了这一点：

- 在简单场景中，除了零号进程外，其他进程立即开始集合操作，但零号进程首先进行设置；
- 所有进程大致上应该同时完成。

在非零进程上，我们现在得到一个时间测量，这个测量本意是仅针对集体操作的时间，其中包括了进程零的设置时间。

解决方案是在你想计时的代码段周围放置一个屏障。 .

```
Barrier();
tstart = Wtime();
Barrier();
duration = Wtime()-tstart;
```

另见图 16.1.

16.2.1 MPI 和 OpenMP 中的并行计时器

许多包都有自己的计时器。例如，对于 *MPI*

```
double MPI_Wtime(void);
double MPI_Wtick(void);
```

参见 *Parallel Programming book*，第 *15.6.1* 节。

对于 *OpenMP*

```
double omp_get_wtime()
double omp_get_wtick()
```

参见 *Parallel Programming book*，第 *28.2* 节。

16.3 Profiling tools

性能分析工具将为您提供程序中各个事件所花费的时间，通常是函数和子程序，或您已声明为此类的代码部分。该工具随后会报告事件发生的次数、总时间和平均时间等信息。

这里我们讨论两个简单的工具：

- *gprof*，需要插桩，和
- *perf*，不需要。
- Intel VTune。

TheTAU 工具，在第 17 节中讨论，出于跟踪的目的，也具有性能分析功能，并以漂亮的图形方式呈现。最后，我们提到 *PETSc* 库允许您定义自己的计时器和事件。

16.3.1 gprof

GNU 编译器的分析器， *gprof* 需要使用额外的标志重新编译和链接：

```
% gcc  -g -pg -c ./srcFile.c
% gcc  -g -pg -o MyProgram ./srcFile.o
```

然后程序自行运行：

```
% ./MyProgram
```

留下一个文件 *gmon.out*。这个文件可以被后处理并显示：

```
% gprof    ./exeFile gmon.out > profile.txt
% gprof -l ./exeFile gmon.out > profile_line.txt
% gprof -A ./exeFile gmon.out > profile_anotated.txt
```

16.3.2 perf

随大多数 Unix 发行版一起提供， *perf* 不需要任何插桩。

Run:

```
perf record myprogram myoptions
perf record --call-graph fp myprogram myoptions
```

这会将事件信息收集到文件 `perf.data` 中，或者执行

```
perf record -o myoutputfile myprogram myoptions
```

for a custom output file.

后处理和显示：

```
perf report
perf report --demangle ## for C++
perf report -i myoutputfile
```

显示可以是交互式的；以下给出一个纯 ascii 显示，限制事件数量超过百分之一，并且只打印百分比和例程名称两列：

```
perf report --stdio \
--percent-limit=1 \
--fields=Overhead,Symbol
```

Example:

```
+   14.15%     4.07%  fsm.exe  fsm.exe             [.] std::vector<richdem::dephier::
Depression<double>, std::allocator<richdem::dephier::Depression<double> > >::at
+    8.92%     4.58%  fsm.exe  fsm.exe             [.] std::vector<richdem::dephier::
Depression<double>, std::allocator<richdem::dephier::Depression<double> > >::_M_range_check
```

这表明 14% 的时间花费在使用 `at` 进行索引，其中超过一半的时间用于范围检查。

16.3.3 Intel VTune

Intel VTune 分析器同样不需要插桩。

```
vtune -collect hotspots yourprogram options
## result are in a directory: r000hs
vtune -report hotspots -r r000hs
```

对于图形输出，你可以使用 *vtune-gui*。这不仅仅是分析结果，还可以让你设置、运行并分析一个应用程序。

16.3.4 MPI 性能分析

MPI 库的设计使得性能分析变得简单。参见 《并行编程》一书，第 *15.6* 节。

16.4 Tracing

在性能分析中，我们只关心汇总信息：一个例程被调用了多少次，以及其总运行时间 / 平均运行时间 / 最小运行时间 / 最大运行时间。然而，有时我们想了解事件的精确时序。这在并行环境中特别相关，当我们关心负载不均衡 和空闲时间时。

诸如 Vampyr 之类的工具可以收集关于事件，特别是消息的跟踪信息，并以图形方式渲染它们，如图 16.2 所示。

图 16.2：一个并行进程的 Vampyr 时间线图。

第 17 章

TAU

The TAU tool [20] (参见 http://www.cs.uoregon.edu/research/tau/home.php 的官方文档) 使用 插桩技术 来分析和跟踪你的代码。也就是说，它会向你的代码中添加分析和跟踪调用。运行结束后，你可以检查输出结果。

Profiling 是收集和显示总体统计数据，例如显示哪些例程占用最多时间，或者通信是否占用了运行时间的大部分。当你关注性能时，一个好的 profiling 工具是必不可少的。

Tracing 是构建和显示程序运行的时间相关信息，例如显示某个进程是否落后于其他进程。为了理解程序的行为以及 profiling 统计背后的原因，tracing 工具可以提供很有价值的见解。

17.1 使用模式

There are two ways to instrument your code:

- 您可以使用动态插装，其中 TAU 在运行时添加测量功能：`# original commandline:`

```
% mpicxx wave2d.cpp -o wave2d
# with TAU dynamic instrumentation:
% mpirun -np 12 tau_exec ./wave2d 500 500 3 4 5
```

- 你可以在编译时添加插桩。为此，你需要让 TAU 在某种程度上接管编译。1. TAU 有自己的 makefiles。名称和位置取决于你的安装，但通常会是类似于 `export TAU_MAKEFILE=$TAU_HOME/lib/Makefile.tau-mpi-pdt` 的东西

2. 现在你可以调用 TAU 编译器 `tau_cc,sh`, `tau_cxx.sh`, `tau_f90.sh`。

当你运行程序时，你需要告诉 TAU 要做什么：

```
export TAU_TRACE=1
export TAU_PROFILE=1
export TRACEDIR=/some/dir
export PROFILEDIR=/some/dir
```

为了生成跟踪图，您需要转换 TAU 输出：

```
cd /some/dir # where the trace and profile output went
tau_treemerge.pl
tau2slog2 tau.trc tau.edf -o yourrun.slog2
```

`slog2` 文件可以用 *jumpshot* 显示。

17.2 Instrumentation

与 *VTune* 等直接对二进制文件进行分析的工具不同，TAU 通过向代码中添加 *instrumentation* 来工作：实际上它是一个源代码到源代码的转换器，将你的代码转换成生成运行时统计信息的代码。

这种 instrumentation 大部分已经为你完成；你主要需要用一个执行源代码到源代码转换的脚本重新编译你的代码，随后编译那个带有 instrumentation 的代码。例如，你可以在你的 makefile 中有如下内容：

```
ifdef TACC_TAU_DIR
  CC = tau_cc.sh
else
  CC = mpicc
endif

% : %.c
<TAB>${CC} -o $@ $^
```

如果要使用 TAU （这里通过检查环境变量 *TACC_TAU_DIR* 来检测），我们将 *CC* 变量定义为 TAU 编译脚本之一；否则将其设置为常规的 MPI 编译器。

Fortran 注意。Cpp 包含 如果你的源代码包含

```
#include "something.h"
```

指令，添加选项

`-optPreProcess`

到 TAU 编译器。

Remark18 PETSc 库可以通过在配置时添加 *--with-perfstubs-tau=1* 选项来启用 *TAU* 插桩进行编译。

要在 *TAU* 上使用 *TACC* 资源，请执行 `module load tau`。

17.3 运行

您现在可以运行插桩后的代码；如果设置了环境变量 *TAU_PROFILE* 和 / 或 *TAU_TRACE*，则跟踪 / 分析输出将写入文件：

```
export TAU_PROFILE=1
export TAU_TRACE=1
```

一次 TAU 运行可以生成许多文件：通常每个进程至少一个。因此，建议为您的跟踪和分析信息创建一个目录。您通过设置环境变量 *PROFILEDIR* 和 *TRACEDIR* 向 TAU 声明它们。

```
mkdir tau_trace
mkdir tau_profile
export PROFILEDIR=tau_profile
export TRACEDIR=tau_trace
```

实际的程序调用保持不变：

```
mpirun -np 26 myprogram
```

TACC note. 在 TACC，使用 *ibrun* 时不带处理器数量；数量由队列提交参数推导得出。

虽然此示例使用了两个独立的目录，但将两者使用同一目录也没有问题。

17.4 输出

跟踪 / 分析信息分散在许多文件中，作为这样很难阅读。因此，您需要一些额外的程序来整合并显示这些信息。

您可以使用 *paraprof* 查看分析信息

```
paraprof tau_profile
```

查看跟踪需要几个步骤：

```
cd tau_trace
rm -f tau.trc tau.edf align.trc align.edf
tau_treemerge.pl
```

```
tau_timecorrect tau.trc tau.edf align.trc align.edf
tau2slog2 align.trc align.edf -o yourprogram.slog2
```

如果跳过 *tau_timecorrect* 步骤，可以通过以下方式生成 slog2 文件：`tau2slog2 tau.trc tau.edf -o yourprogram.slog2`

slog2 文件可以用 *jumpshot* 查看：

```
jumpshot yourprogram.slog2
```

17.5 Without instrumentation

对未插装代码的基于事件的采样：

```
tau_exec -ebs yourprogram
```

生成的 `.trc` 文件可以用 *paraprof* 查看。

17.6 示例

17.6.1 Bucket brigade

L让我们考虑一个 *bucket brigade* 实现的广播：每个进程将其数据发送给下一个 higher rank.

```
int sendto =
    ( procno<nprocs-1 ? procno+1 : MPI_PROC_NULL )
    ;
int recvfrom =
    ( procno>0 ? procno-1 : MPI_PROC_NULL )
    ;

MPI_Recv( leftdata,1,MPI_DOUBLE,recvfrom,0,comm,MPI_STATUS_IGNORE);
myvalue = leftdata
MPI_Send( myvalue,1,MPI_DOUBLE,sendto,0,comm);
```

W我们用阻塞发送和接收实现了桶式传递：每个进程等待从 i它的前驱接收，然后再发送给它的后继。

```
// bucketblock.c
if (procno>0)
  MPI_Recv(leftdata, N, MPI_DOUBLE,recvfrom,0, comm, MPI_STATUS_IGNORE);
for (int i=0; i<N; i++)
  myvalue[i] = (procno+1)*(procno+1) + leftdata[i];
if (procno<nprocs-1)
  MPI_Send(myvalue,N, MPI_DOUBLE,sendto,0, comm);
```

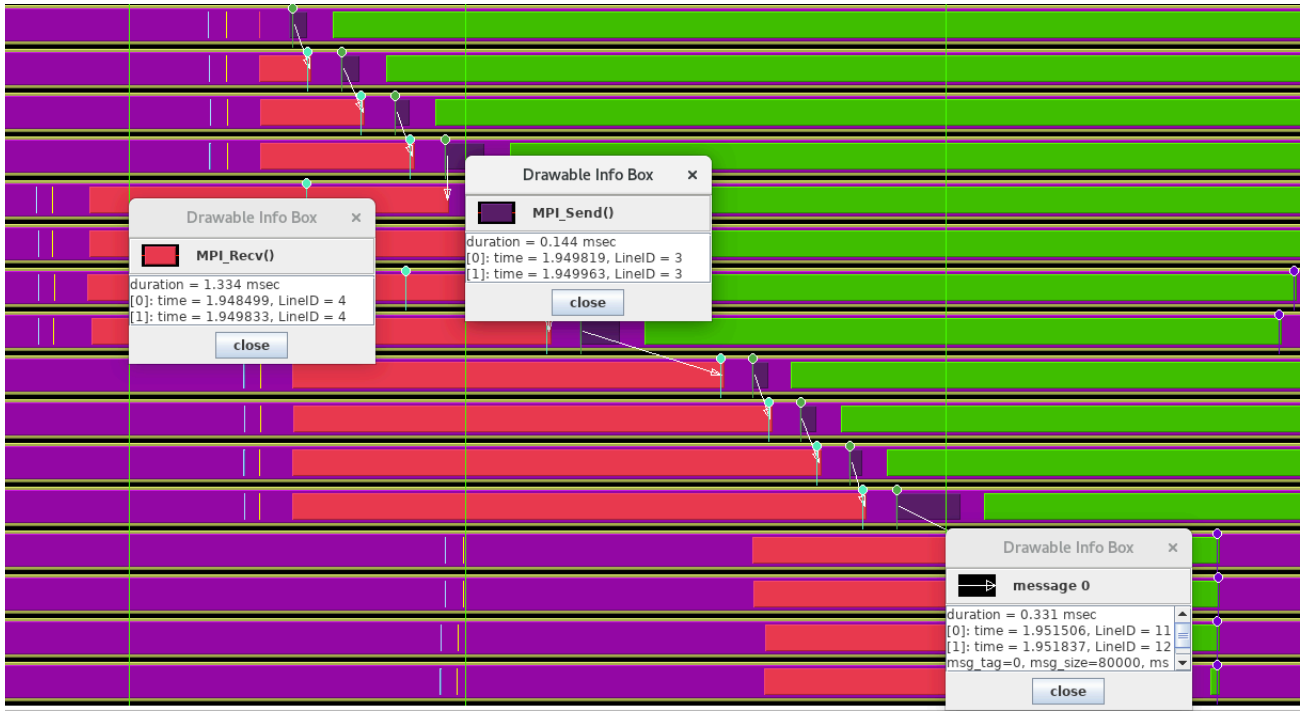


Figure 17.1: Trace of a bucket brigade broadcast

该操作的 TAU 跟踪见图 17.1，使用了 4 个节点，每个节点 4 个进程。我们看到每个节点内的进程同步较好，但节点之间的同步较少。然而，桶式传递随后对进程施加了自己的同步，因为每个进程都必须等待其前驱，无论它是否提前发出了接收操作。

接下来，我们在此操作中引入流水线：每个发送被拆分成若干部分，这些部分通过非阻塞调用发送和接收。

```
// bucketpipenonblock.cMPI_Request rrequests[PARTS];
for (int ipart=0; ipart<PARTS; ipart++) {MPI_Irecv(
leftdata+partition_starts[ipart],partition_sizes[ipart],
MPI_DOUBLE,recvfrom,ipart,comm,rrequests+ipart);}
```

TAU 跟踪见图 17.2。

17.6.2 Butterfly exchange

The NAS Parallel Benchmark suite [17] 包含一个共轭梯度（CG）实现，该实现将其 all-reduce 操作明确表示为 *butterfly exchange*。

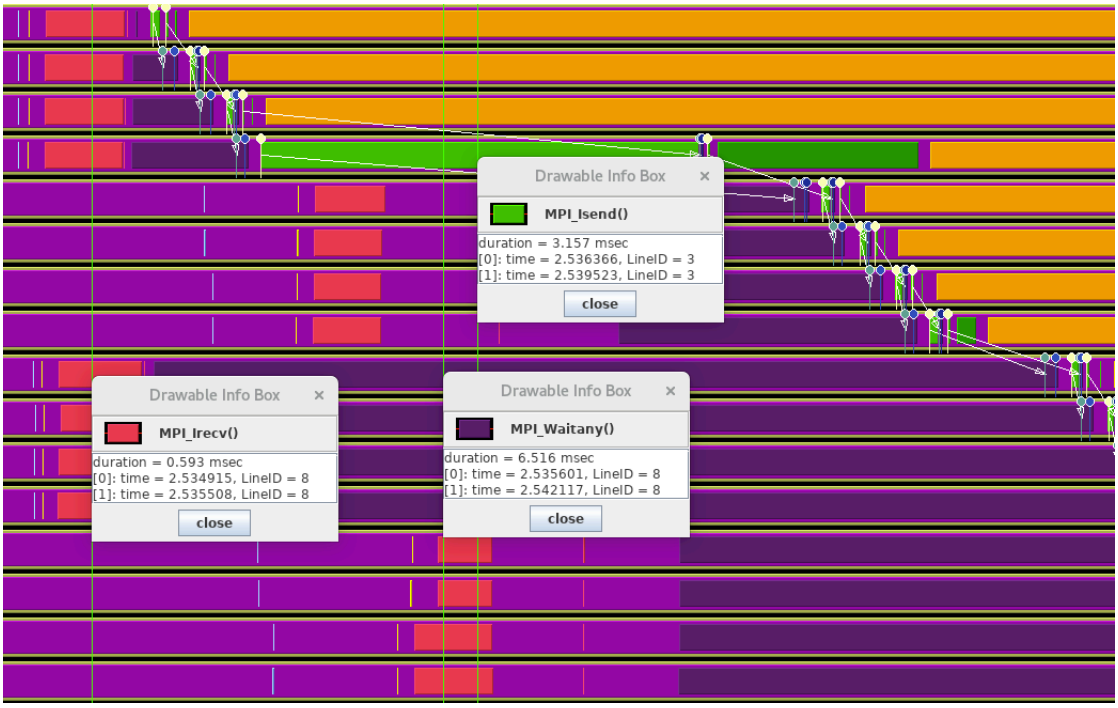


图 17.2：流水线桶式广播的跟踪

```
!! cgb.f
         do i = 1, l2npcols
            call mpi_irecv( d,
     >                      1,
     >                      dp_type,
     >                      reduce_exch_proc(i),
     >                      i,
     >                      mpi_comm_world,
     >                      request,
     >                      ierr )
            call mpi_send(  sum,
     >                      1,
     >                      dp_type,
     >                      reduce_exch_proc(i),
     >                      i,
     >                      mpi_comm_world,
     >                      ierr )

            call mpi_wait( request, status, ierr )

            sum = sum + d
         enddo
```

我们在 TAU 跟踪中识别出这个结构：图 17.3。仔细观察，我们看到这个特定算法引入了大量等待时间。图 17.5 和 17.6 展示了一整串进程的级联

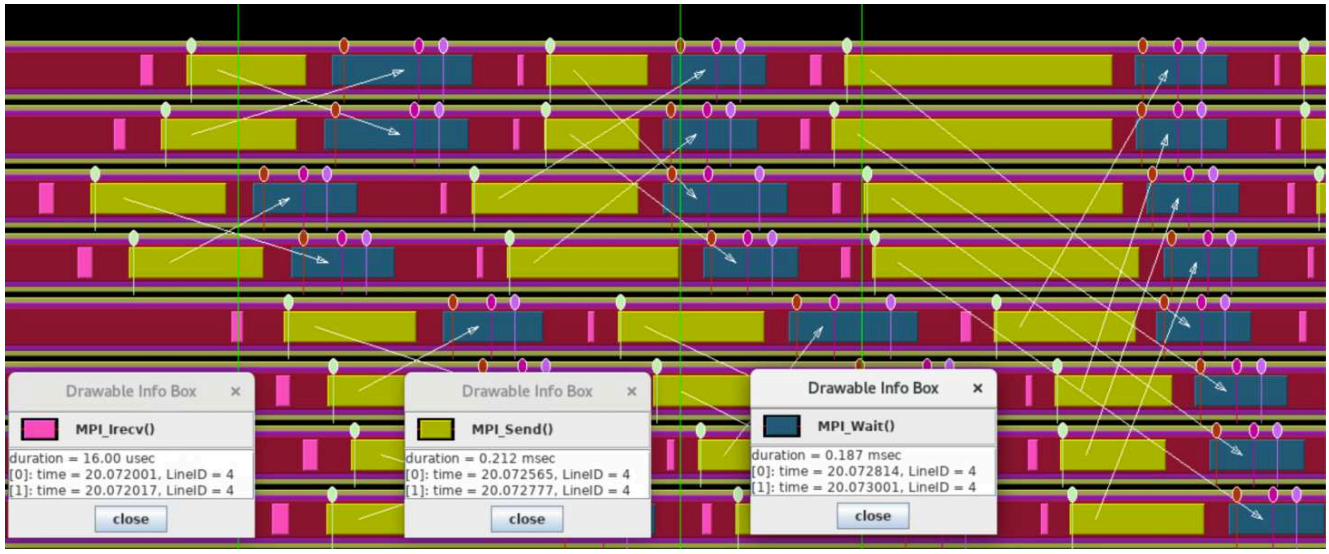


图 17.3：butterfly exchange 的轨迹

相互等待。

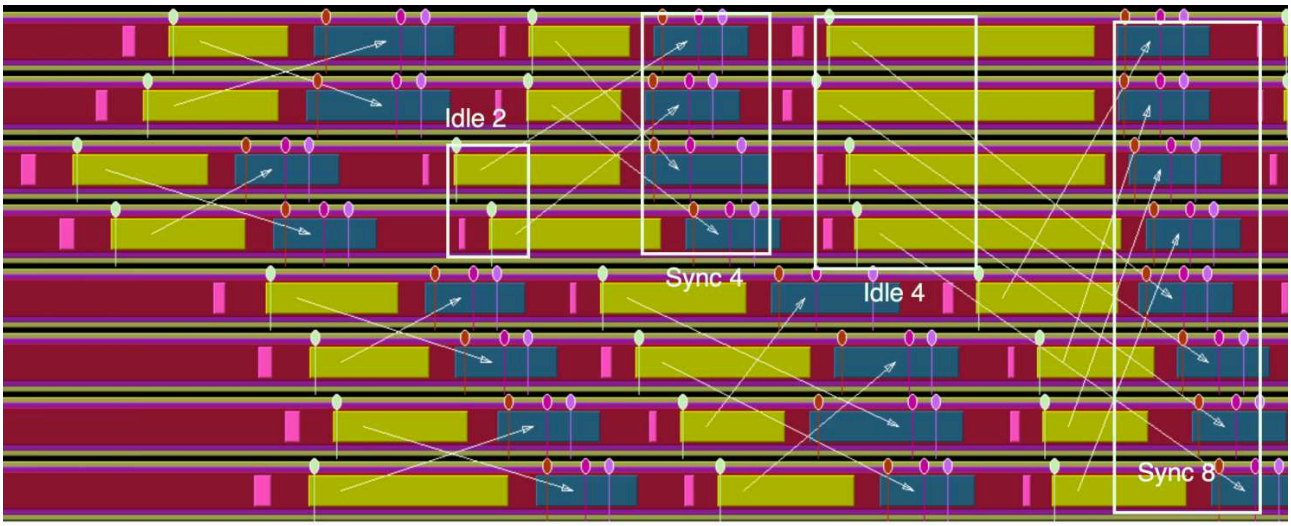


图 17.4：butterfly 交换的轨迹

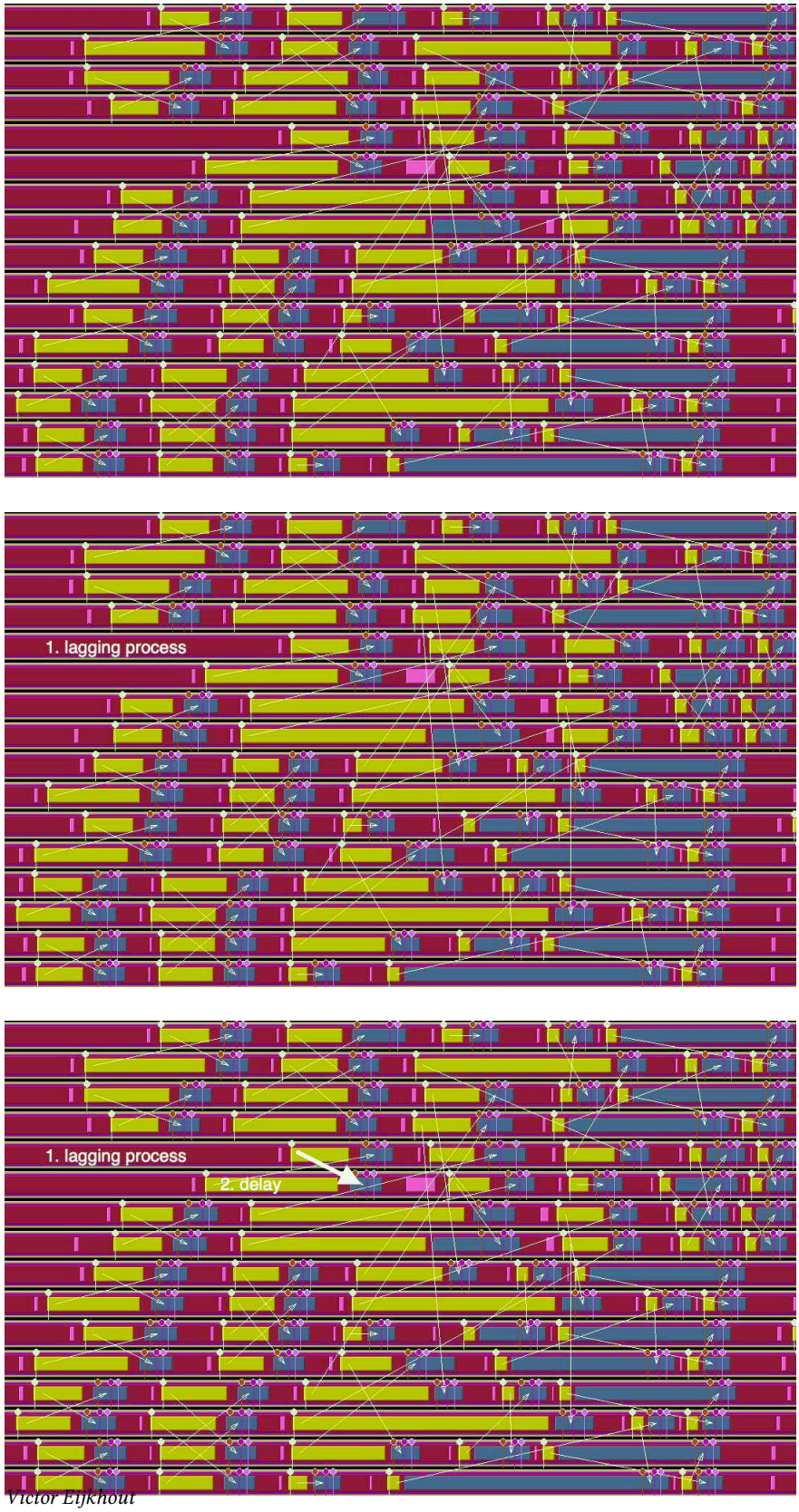


图 17.5：由单个滞后进程引起的进程等待的四个阶段

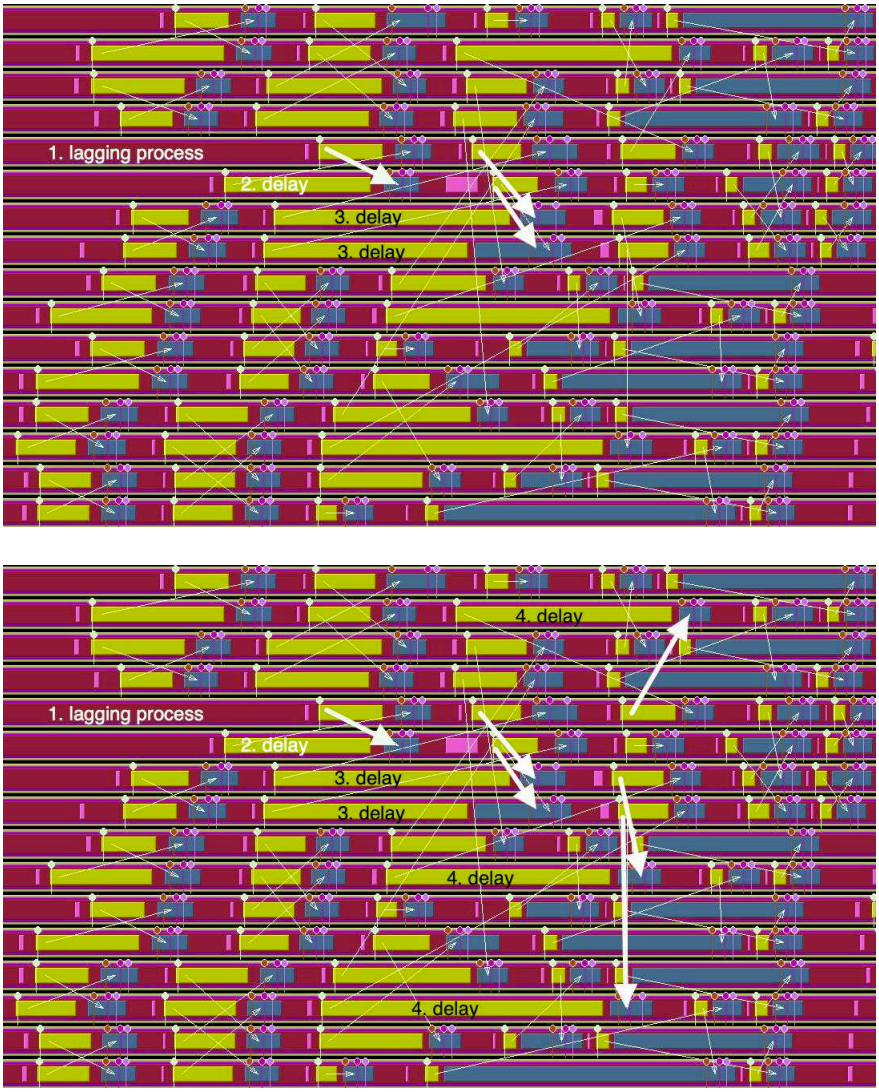


Figure 17.6: Four stages of processes waiting caused by a single lagging process

第 18 章

SLURM

超级计算机 集群 可以拥有大量的 节点，但不足以让所有用户同时以他们想要的规模运行。因此，用户被要求提交作业，这些作业可能会立即开始执行，也可能必须等待资源可用。

决定何时运行作业以及分配哪些资源，不是由人工操作员完成，而是由称为批处理系统的软件完成。（ *Stampede* 集群在 *TACC* 的运行期间处理了接近 1000 万个作业，相当于每 20 秒启动一个作业。）

本教程将涵盖此类系统的基础知识，特别是资源管理的简单 Linux 实用工具（ SLURM ）。

18.1 集群结构

超级计算机集群通常有两种类型的节点：

- 登录节点，和
  计算节点 .

当你建立一个 *ssh connection* 到集群时，你连接的是一个 login node。login node 的数量很少，通常少于半打。

Exercise18.1. 连接到你喜欢的集群。那个 login node 上有多少人？如果你断开连接再重新连接，你会发现自己在同一个 login node 上吗?

Compute nodes 是运行你作业的地方。不同的集群在这里有不同的结构：

- 计算节点可以在用户之间共享，也可以专门分配。– 如果用户作业的并行度低于节点的核心数，共享是有意义的。– …… 另一方面，这意味着共享节点的用户作业可能会相互干扰，一个作业可能会占用另一个作业所需的内存或带宽。– 使用专用节点时，作业可以访问该节点的所有内存和所有带宽。
- 集群可以是同质的，即每个计算节点具有相同的处理器类型，也可以拥有多种处理器类型。例如，TACC *Stampede2* 集群拥有 *IntelKnights-landing* 和 *Intel Skylake* 节点。
- 通常，集群中有一些 “ 超大内存 ” 节点，内存容量约为一 TB 或更多。由于此类硬件的成本，通常这类节点数量较少。

18.2 Queues

J作业通常无法立即启动，因为资源不足，或因为其他作业正在运行。
可能具有更高的优先级（参见第 18.7 节）。因此，作业通常会被放入队列，由批处理系统如 SLURM 进行调度和启动。

批处理系统不会将所有作业放入一个大池中：作业被提交到多个队列中的任意一个，这些队列都是分别调度的。

Queues can differ in the following ways:

- 如果一个集群有不同类型的处理器，这些处理器通常会分布在不同的队列中。此外，可能会有专门针对附带图形处理单元（ GPU ）的节点的队列。拥有多个队列意味着你必须决定你的作业想在哪种处理器类型上运行，即使你的可执行文件在二进制层面上与所有处理器兼容。
- 可能会有 “ 开发 ” 队列，这些队列对运行时间和节点数量有严格限制，但作业通常启动更快。
- 一些集群有 “ 高级 ” 队列，这些队列收费更高，但提供更高的优先级。
- “ 大内存节点 ” 通常也会有自己的队列。
- 对于资源需求较大的作业，比如需要大量核心数或运行时间比正常更长的作业，可能会有额外的队列。

对于 slurm， *sinfo* 命令可以告诉你很多关于队列的信息。

```
# what queues are there?sinfo -o "%P"

# what queues are there, and what is their status?

sinfo -o "%20P %.5a"
```

Exercise 18.2. 输入这些命令。有多少个队列？它们现在都在运行吗？

18.2.1 队列限制

Queues have limits on

- 作业的运行时间；
- 作业的节点数；或者
- 用户在该队列中可以拥有多少个作业。

18.3 作业运行

T在由 slurm 管理的集群上启动作业主要有两种方式。你可以启动一个程序
r与 *srun* 异步运行，但这可能会挂起直到资源可用。因此，在本节中，
w我们重点介绍通过使用 *sbatch* 提交作业来异步执行你的程序。

18.3.1 作业提交周期

为了运行一个批处理作业，你需要编写一个作业脚本，或者批处理脚本。该脚本描述了你将运行的程序、其输入和输出的位置、可使用的进程数量以及运行时间。

在最简单的形式中，你可以直接提交脚本而无需额外的参数：

```
sbatch yourscript
```

关于作业运行的所有选项都包含在脚本文件中，下面我们将进行讨论。

作为作业提交的结果，您会获得一个作业 ID。提交后，您可以使用 *squeue*:`squeue -j 123456` 查询您的作业。

or queury all your jobs:

```
squeue -u yourname
```

squeue 命令报告您的作业的各个方面，例如其状态（通常是等待中或运行中）；如果正在运行，还会报告其运行的队列（或 “ 分区 ” ）、已用时间以及实际运行的节点。

```
squeue -j 5807991
  JOBID   PARTITION     NAME     USER ST   TIME  NODES NODELIST(REASON)
5807991 development packingt eijkhout  R   0:04      2 c456-[012,034]
```

如果您在提交脚本后发现错误，包括脚本已经开始运行时，您可以使用 *scancel* 取消您的作业：

```
scancel 1234567
```

18.4 脚本文件

A job script looks like an executable shell script:

- 它有一个 ‘ 解释器 ’ 行，比

如 `#!/bin/bash` 在顶部，且

- 它包含普通的 unix 命令，包括
- 你程序的（并行）启动：`# sequential program:`

`./yourprogram youroptions# parallel program,`

`general:`

`mpiexec -n 123 parallelprogram options`

```
# parallel program, TACC:
ibrun parallelprogram options
```

- … 然后它有许多选项来指定并行运行。

18.4.1 sbatch 选项

除了常规的 unix 命令和解释器行之外，你的脚本还有许多 SLURM 指令，每个指令都以 `#SBATCH` 开头。（这使它们成为 shell 解释器的注释，因此批处理脚本实际上是合法的 shell 脚本。）

指令的形式为

```
#SBATCH -option value
```

常用选项（除了并行相关选项，详见第 18.5 节）包括：

- `-J`: the jobname. This will be displayed when you call *squeue*.
- `-o`: name of the output file. This will contain all the stdout output of the script.
- `-e`: name of the error file. This will contain all the stderr output of the script, as well as slurm error messages.
  It can be a good idea to make the output and error file unique per job. To this purpose, the macro `%j` is available, which at execution time expands to the job number. You will then get an output file with a name such as `myjob.o2384737`.
- `-p`: *partition* 或队列。见上文。
- `-t hh:mm:ss`: 最大运行时间。如果你的作业超过此时间，将被取消。有两个注意点：

  1. 这里不能指定超过队列限制的时长。2. 作业时间越短，越有可能被优先调度。

- `-w c452-[101-104,111-112,115]` 指定放置作业的节点。
- `-A`: 您的作业应计费的账户名称。
- `--mail-user=you@where` Slurm 可以在作业开始或结束时通知您。例如，您可能希望在作业开始时连接到作业（以运行 *top*），或者在作业完成时检查结果，但不想整天盯着您的终端。您希望被通知的操作通过（其中包括） `--mail-type=begin/end/fail/all` 指定。
- `--dependency=after:123467` 表示该作业将在作业 1234567 完成后启动。使用 `afterok` 仅在该作业成功完成时启动。（参见 https://cvw.cac.cornell.edu/slurm/submission_depend 了解更多选项。）
- `--nodelist` 允许您指定特定节点。这对于获得可重复的计时结果可能有好处，但可能会增加您在队列中的等待时间。
- `--array=0-30` 是 ‘数组作业’ 的规范：需要针对一系列参数值执行的任务。

  TACC 注意。TACC 不支持 Arry 作业；请改用 launcher ； 参见章节 18.5.3。
- `--mem=10000` 指定每个节点所需的内存量。默认单位是兆字节，但可以用 `K/M/G/T` 明确指示。

TACC note. 此选项不能用于请求任意内存：作业始终可以访问所有可用的物理内存，并且不允许使用共享内存。

参见 https://slurm.schedmd.com/sbatch.html 获取完整列表。

Exercise 18.3. 编写一个脚本，执行 `date` 命令两次，中间有一个 `sleep`。提交该脚本并调查输出。

18.4.2 环境

您的作业脚本在执行时表现得像任何其他 shell 脚本。特别是，它继承了调用 *environment* 及其所有环境变量。此外，slurm 定义了许多环境变量，例如作业 ID 、主机列表、节点数和进程数。

18.5 并行处理

我们分别讨论并行选项。

18.5.1 MPI 作业

在大多数集群中，存在包含一个或多个多核处理器的计算节点结构。因此，您需要指定节点数和核心数。为此，分别有选项 `-N` 和 `-n`。

```
#SBATCH -N 4              # Total number of nodes
#SBATCH -n 4              # Total number of mpi tasks
```

也可以只指定节点数或核心数，但这样会减少灵活性：

- 如果一个节点有 40 个核心，但你的程序在 10 个 MPI 进程时就停止扩展，你可以使用：`#SBATCH -N 1#SBATCH -n 10`

- 如果你的进程使用大量内存，你可能想保留一些核心不使用。在一个 40 核节点上，你可以使用
  `#SBATCH -N 2`
  `#SBATCH -n 40` 或者
  `#SBATCH -N 1`
  `#SBATCH -n 20`

你也可以不用指定总核心数，而是用 `--ntasks-per-node` 指定每个节点的核心数。

Exercise18.4. 浏览上述示例，并用等效的 `--ntasks-per-node` 值替换 `-n` 选项。

Python note. 使用 *mpi4py* 的 Python MPI 程序应像其他 MPI 程序一样处理，只是你需要指定 python 可执行文件和脚本名称，而不是可执行文件名：

```
ibrun python3 mympi4py.py
```

18.5.2 线程作业

以上讨论主要与 MPI 程序相关。其他一些情况：

- 对于纯 OpenMP 程序，你只需要一个节点，因此 `-N` 的值为 1。也许令人惊讶的是，`-n` 的值也是 1，因为只需要创建一个进程：OpenMP 使用线程级并行性，这通过 *OMP_NUM_THREADS* 环境变量指定。

- 类似的情况也适用于 *Matlab parallelcomputing toolbox*（注意：不是 distributed computing toolbox），以及 *Python multiprocessing* 模块。

练习 18.5. 如果你为纯 OpenMP 程序指定一个大于 1 的 `-n` 值，会发生什么?

对于混合计算 MPI-OpenMP 程序，你需要结合使用 slurm 选项和环境变量，例如，`--tasks-per-node` 和 `OMP_NUM_THREADS` 的乘积要小于节点的核心数。

18.5.3 参数扫描 / 集合 / 大规模并行

到目前为止，我们关注的是调度单个并行可执行文件的作业。然而，有些用例需要你为大量输入运行一个顺序（或非常有限并行）的可执行文件。这种情况被称为参数扫描或集合。

Slurm 本身可以通过数组作业支持这一点，尽管还有更复杂的 *launcher* 工具可用于此类目的。

TACCnote. TACC 集群不支持数组作业。相反，请使用 *launcher* 或 *pylauncher* 模块。

18.6 作业运行

当您的作业正在运行时，其状态由 *squeue* 报告为 `R`。该命令还会报告分配给它的节点。

```
squeue -j 5807991
       JOBID   PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
     5807991 development packingt eijkhout  R       0:04      2 c456-[012,034]
```

然后，您可以 *ssh* 进入作业的计算节点；通常，计算节点是禁止访问的。如果您想运行 *top* 来查看您的进程运行情况，这非常有用。

18.7 调度策略

这样的系统会根据资源的可用性和用户的优先级来确定作业何时可以运行。

当然，如果用户请求大量节点，可能永远不会有那么多节点同时可用，因此批处理系统会强制保证可用性。它通过确定作业的运行时间，然后让节点保持空闲 状态，以便在该时间可用。

一个有趣的副作用是，在真正的大作业开始之前，如果一个作业运行时间很短，一个 “ 相当 ” 大的作业可以被运行。这被称为回填，它可能导致作业比其优先级所允许的更早运行。

18.8 文件系统

文件系统有不同的类型：

- 它们可以有备份也可以没有；
- 它们可以是共享的也可以不是；并且
- 它们可以是永久的也可以被清除。

在许多集群中，每个节点都有一个本地磁盘，可能是旋转盘或 *RAM disc*。这通常大小有限，只应在作业运行期间用于临时文件。

大多数文件系统存在于作为 *RAID array* 一部分的磁盘上。这些磁盘具有大量冗余以实现容错，并且整体上它们形成了一个 *shared file system*：一个统一的文件系统，可以从任何节点访问，文件可以具有任意大小，或者至少比系统中任何单个磁盘都大得多。

TACCnote. `HOME` 文件系统大小有限，但既是永久性的又有备份。这里你可以放置脚本和源代码。

`WORK` 文件系统是永久性的，但没有备份。你可以在这里存储模拟的输出。然而，目前工作文件系统无法立即承载大型并行作业的输出。

`SCRATCH` 文件系统会被清理，但它拥有接受程序输出的最高带宽。这是你写入数据的地方。经过后处理后，你可以将数据存储到工作文件系统，或者写入磁带。**Exercise** 18.6. 如果你用 *cmake* 安装软件，通常会有

1. 一个包含所有 cmake 选项的脚本；2. 源代码，3. 安装的头文件和二进制文件，4. 临时对象文件等。你会如何在可用的文件系统上组织这些对象?

18.9 Examples

非常简略的章节。

18.9.1 作业依赖

```
JOB=`sbatch my_batchfile.sh | egrep -o -e "\b[0-9]+$"`

#!/bin/sh

# Launch first job
JOB=`sbatch job.sh | egrep -o -e "\b[0-9]+$"`

# Launch a job that should run if the first is successful
sbatch --dependency=afterok:${JOB} after_success.sh

# Launch a job that should run if the first job is unsuccessful
sbatch --dependency=afternotok:${JOB} after_fail.sh
```

18.9.2 脚本中的多次运行

```
ibrun stuff &
sleep 10
for h in hostlist ; do
  ssh $h "top"
done
wait
```

18.10 复习问题

对于所有判断题，如果你的答案是错误的，正确答案是什么，为什么?

Exercise18.7. 判断题？当你提交一个作业时，一旦有足够的资源，它会立即开始运行。

Exercise18.8. 判断题？如果你提交以下脚本：

```
#!/bin/bash#SBATCH -N 10#SBATCH -n 10
echo "hello world"
```

你会在输出中得到 10 行 ‘hello world’。

Exercise18.9. 判断题？如果你提交以下脚本：

```
#!/bin/bash
#SBATCH -N 10
#SBATCH -n 10
hostname
```

你可以获得提交作业的登录节点的主机名。

Exercise 18.10. 当你的作业运行时，以下哪些资源是与其他用户共享的：

- Memory;
- CPU;
- Disc space?

Exercise 18.11. 查询 yourjob 状态的命令是什么?

- `sinfo`
- `squeue`
- `sacct`

Exercise 18.12. 在 4 个节点，每个节点 40 核的情况下，最大运行程序规模是多少，单位是

- MPI ranks;
- OpenMP 线程?

第 19 章

SimGrid

本书的许多读者将能够使用某种并行机器来运行模拟，甚至进行一些现实的规模扩展研究。然而，很少有人能够访问多于一种集群类型，以便评估互连的影响。即便如此，出于教学目的，人们通常希望拥有一些不太可能获得的互连类型（全连接、线性处理器阵列）。

为了探讨与网络相关的架构问题，我们转而使用一个模拟工具，*SimGrid*。

安装

编译 你编写普通的 MPI 文件，但使用 *SimGrid* 编译器 *smpicc* 来编译它们。

运行 SimGrid 有其自己的 `mpirun` 版本：*smpirun*。你需要用以下选项来提供它：

- `-np 123456` 表示（虚拟）处理器的数量；
- `-hostfile simgridhostfile` 列出这些处理器的名称。你基本上可以随意命名，但这些名称定义在：

- `-platform arch.xml` 定义处理器之间的连接关系。

例如，对于包含 8 个主机的 hostfile，线性连接的网络定义为：

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid/simgrid.dtd">

<platform version="4">

<zone id="first zone" routing="Floyd">
  <!-- the resources -->
  <host id="host1" speed="1Mf"/>
  <host id="host2" speed="1Mf"/>
  <host id="host3" speed="1Mf"/>
```

```
    <host id="host4" speed="1Mf"/>
    <host id="host5" speed="1Mf"/>
    <host id="host6" speed="1Mf"/>
    <host id="host7" speed="1Mf"/>
    <host id="host8" speed="1Mf"/>
    <link id="link1" bandwidth="125MBps" latency="100us"/>
    <!-- the routing: specify how the hosts are interconnected -->
    <route src="host1" dst="host2"><link_ctn id="link1"/></route>
    <route src="host2" dst="host3"><link_ctn id="link1"/></route>
    <route src="host3" dst="host4"><link_ctn id="link1"/></route>
    <route src="host4" dst="host5"><link_ctn id="link1"/></route>
    <route src="host5" dst="host6"><link_ctn id="link1"/></route>
    <route src="host6" dst="host7"><link_ctn id="link1"/></route>
    <route src="host7" dst="host8"><link_ctn id="link1"/></route>
  </zone>

  </platform>
```

(此类文件可以通过一个简短的 shell 脚本生成).

`Floyd` 路由的指定意味着可以使用给定路径的传递闭包的任何路由。也可以使用 `routing="Full"`，这要求完全指定所有可以通信的对。

第 20 章

参考文献

[1] Alfred V. Aho, Brian W. Kernighan, 和 Peter J. Weinberger. *The Awk ProgrammingLanguage*Addison-Wesley 计算机科学系列。Addison-Wesley 出版社，1988 年。ISBN 020107981X 9780201079814。 [Cited onpage 37.]

[2] L.S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammerling, G. Henry, A. Petitet, K. Stanley, D. Walker 和 R.C. Whaley. ScaLAPACK 用户指南 SIAM, 1997。 Netlib.org BLAS 参考实现。Cited on page 122.

[4] Yaeyoung Choi, Jack J. Dongarra, Roldan Pozo 和 David W. Walker。Scalapack：一个用于分布式内存并发计算机的可扩展线性代数库。载于《第四届大规模并行计算前沿研讨会论文集（Frontiers’92），弗吉尼亚州麦克莱恩，1992 年 10 月 19-21 日》，第 120-127 页，1992 年。 [Cited 第 {v10} 页。]

[5] Edsger W. Dijkstra. 编程作为一种数学性质的学科。*Am. Math. Monthly*, 81:608–612, 1974. [Cited 第 151 页。]

[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling 和 Iain Duff。一组三级基本线性代数子程序。*ACM Transactions on Mathematical Software*，16(1)：1–17，1990 年 3 月。 [Cited on 页 122。]

[7] Jack J. Dongarra、Jeremy Du Croz、Sven Hammarling 和 Richard J. Hanson。扩展的 FORTRAN 基本线性代数子程序集。*ACMTransactionson Mathematical Software*，14(1)：1–17，1988 年 3 月。 [Cited 第 122 页。]

[8] Dale Dougherty 和 Arnold Robbins。《sed & awk》。O’Reilly Media，第 2 版。印刷版 ISBN：978-1-56592-225-9，ISBN 10：1-56592-225-5；电子书 ISBN：978-1-4493-8700-6，ISBN 10：1-4493-8700-4[Cited on page 37.]Victor Eijkhout

[9] *TEX 和 LATEX 的科学*，lulu.com, 2012. [Cited on page 41.]

[10] C. A. R. Hoare. 计算机程序设计的公理基础。*Communications ofthe ACM*, 页码 576–580, 1969 年 10 月。 [Cited 第 151 页。]

[11] Helmut Kopka 和 Patrick W. Daly。 *A Guide to LATEX*。Addison-Wesley，首次出版于 1992 年。 [Cited on 第 196 页。]

[12] L. Lamport。 *LATEX*，文档准备系统。Addison-Wesley，1986 年。 [Cited 第 196 页。]

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid 和 F. T. Krogh。Fortran 使用的基本线性代数子程序。*ACM Trans. Math. Softw.*，5(3):308–323，1979 年 9 月。 [Cited 第 122 页。]

[14] Robert Mecklenburg. Managing Projects with GNU Make. O'Reilly Media，第 3 版，2004 年。印刷版 ISBN:978-0-596-00610-5 ISBN 10:0-596-00610-1 电子书 ISBN:978-0-596-10445-0 ISBN 10:0-596-10445-6。 第 57 页。

[15] Sandra Mendez、 Sebastian Lührs、 Volker Weinberg、 Dominic Sloan-Murphy 和 Andrew Turner。最佳实践指南 - 并行 I/O。2019 年 02 月。第 139 页。

[16] Frank Mittelbach、 Michel Goossens、 Johannes Braams、 David Carlisle 和 Chris Rowley。《 The LaTeX Companion，第 2 版》。Addison-Wesley，2004 年。 [Cited on page 196.]

[17] NASA 高级超级计算部门。NAS 并行基准测试。 https://www.nas.nasa.gov/publications/npb.html. [Cited onpage 209.]

[18] Tobi Oetiker. L A TEX 简明入门。第 184 页和第 196 页。

[19] Jack Poulson, Bryan Marker, Jeff R. Hammond, and Robert van de Geijn. Elemental: a new 框架 for distributed memory dense matrix computations. *ACMTransactions on Mathematical Software*.submitted. [Cited onpage 122.]

[20] S. Shende 和 A. D. Malony。《 International Journal of High Performance Computing Applications》，20:287–331，2006 年。第 205 页。

[21] TEX 常见问题解答。 [Cited on page 196.]

[22] R. van de Geijn, Philip Alpatov, Greg Baker, Almadena Chtchelkanova, Joe Eaton, Carter Edwards, Murthy Guddati, John Gunnels, Sam Guyer, Ken Klimkowski, Calvin Lin, Greg Morrow, Peter Nagel, James Overfelt, and Michelle Pal. Parallel linear algebra 包 (PLAPACK): 发布 r0.1 (beta) users' guide. 1996. [Cited onpage 122.]

[23] Robert A. van de Geijn. *Using PLAPACK:Parallel Linear Algebra Package*. The MIT Press,1997. [Cited 第 122 页。]

[24] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. 科学计算的最佳实践。 *PLOS Biology*, 12(1):1–7, 01 2014. [Cited on 第 6 页。]

第 21 章

缩略语列表

ABI 应用二进制接口 **ADL** 参数依赖查找 **AMR** 自适应网格细化 **AOS** 结构数组 **API** 应用程序接口 **AVX** 高级向量扩展 **BEM** 边界元法 **BFS** 广度优先搜索 **BLAS** 基础线性代数子程序 **BM** Bowers-Moore**BSP** 批量同步并行 **BVP** 边值问题 **CAF** 协数组 Fortran**CCS** 压缩列存储 **CG** 共轭梯度 **CGS** 经典 Gram-Schmidt**COO** 坐标存储 **CPP** C 预处理器 **CPU** 中央处理器 **CRS** 压缩行存储 **CSV** 逗号分隔值 **CUDA** 统一计算设备架构 **DAG** 有向无环图 **DL** 深度学习 **DRAM** 动态随机存取存储器 **DSP** 数字信号处理 **FD** 有限差分 **FE** 有限元 **FMA** 融合乘加 **F** **DM** 有限差分法 **FEM** 有限元法 **FMM** 快速多极方法 **FOM** 全正交化方法 **FPU** 浮点单元 **FFT** 快速傅里叶变换 **FSA** 有限状态自动机 **FSB** 前端总线 **FPGA** 现场可编程门阵列 **GMRES** 广义最小残差 **GPU** 图形处理单元 **GPGPU** 通用图形处理单元 **GS** Gram-Schmidt**GSL** 指导支持库 **GnuSL** GNU 科学库 **GUI** 图形用户界面 **HDFS** Hadoop 文件系统 **HPC** 高性能计算 **HPF** 高性能 Fortran**IBVP** 初始边值问题 **IDE** 集成开发环境 **ILP** 指令级并行 **ILU** 不完全 LU**IMP** 并行集成模型 **IVP** 初值问题 **LAPACK** 线性代数包 **LAN** 局域网 **LBM** 格子 Boltzmann 方法 **LRU** 最近最少使用 **MGS** 改进 Gram-Schmidt**MIC** 多集成核心 **MIMD** 多指令多数据

MGS 改进的 Gram-Schmidt**ML** 机器学习 **MPI** 消息传递接口 **MPL** 消息传递库 **MSI** 修改 - 共享 - 无效 **MTA** 多线程架构 **MTSP** 多旅行推销员问题 **NUMA** 非统一内存访问 **ODE** 常微分方程 **OO** 面向对象 **OOP** 面向对象编程 **OS** 操作系统 **PGAS** 分区全局地址空间 **PDE** 偏微分方程 **PRAM** 并行随机存取机 **RDMA** 远程直接内存访问 **RNG** 随机数生成器 **SAN** 存储区域网络 **SAS** 软件即服务 **SCS** 最短公共超集 **SFC** 填充空间曲线 **SGD** 随机梯度下降 **SIMD** 单指令多数据 **SIMT** 单指令多线程 **SLURM** 简单 Linux 资源管理工具 **SM** 流多处理器 **SMP** 对称多处理 **SMT** 对称多线程 **SOA** 数组结构 **SOR** 逐次超松弛法 **SSOR** 对称逐次超松弛法 **SP** 流处理器 **SPMD** 单程序多数据 **S**
PD 对称正定 **SRAM** 静态随机存取存储器 **SSE** SIMD 流扩展 **SSSP** 单源最短路径 **STL** 标准模板库 **TBB** 线程构建块（ Intel ） **TDD** 测试驱动开发 **TLB** 地址转换旁路缓存 **TSP** 旅行推销员问题 **UB** 未定义行为 **UMA** 统一内存访问 **UPC** 统一并行 C**WAN** 广域网

第 22 章

索引

ISBN 978-1-257-99254-6
90000
9 781257 992546