

针对 VASP 的材料计算教程 为材料计算而生

Jiaqi Z.¹

2025 年 1 月 15 日

¹Copyright © 2024 Jiaqi Z. All rights reserved.

目录

前言	iii
为创作者而作的教程	vii
I Linux 基础	1
1 Linux 命令行操作	3
2 文本编辑工具 vi 和 vim	31
3 高级 Linux 命令	45
4 Shell 脚本基础	61
II VASP 计算	115
5 能带计算	121
6 声子谱计算	141
III Python 与机器学习	155

前言

“为材料计算而生”，是抱着多大的觉悟说出这种话啊。这只是一本书，有办法背负其他人的人生吗？

真是满脑子只想着自己呢……¹

俗话说得好，好记性不如烂笔头，这句话在任何时候都显得格外贴切。尤其是在科研领域（特别是材料科学这样规范和流程化的学科），记录的重要性更是不言而喻。随着计算任务的不断增加，我们掌握的计算方法和参数也日益繁多。将这些知识、操作和方法记录下来，不仅能够帮助我们避免遗忘，还能在遇到类似问题时快速查找，而不必在海量的网络搜索结果中苦苦寻觅。

正是基于这样的考虑，我们结合自己科研团队在材料计算方面遇到的一些实际问题，整理编写了这本书。我们编写这本书的目的有两个：一是为了方便自己，在面对类似问题时能够迅速回忆起解决方案；二是为了通过集体的智慧，汇聚大家的方法和思路，以便在遇到新问题时能够迅速找到答案。

这本书的诞生，既是为了服务于材料计算，也是因材料计算而生。既然如此，我们为何不称它为“为材料计算而生”呢？

我们衷心希望这本书能够惠及更多的人，无论是我们团队的新成员，还是其他团队的老师或同学，都能从中获得帮助。

同时，我们也清楚地认识到自己的能力和知识是有限的，书中的内容难免会有疏漏或错误。我们诚挚地希望读者在使用过程中能够提出宝贵的意见和建议，或者分享你们的经验，共同促进我们的成长和进步。

最后，再次感谢您阅读并使用这本书。

Jiaqi Z.

2024 年 8 月青岛

¹ 以上内容改编自动漫《BanG Dream! It's MyGO!!!!!》中丰川祥子的台词

如何联系作者

可通过以下任意一种方式联系:

- GitHub 的 Issue, 这是最直接的方式²;
- email, 请发送邮件至 zhangjq00@sdust.edu.cn 或 zhangjq_sd@163.com

如何使用这本书

在使用时, 请按照如下方法:

1. 根据研究问题, 寻找合适的章节; 如果没有, 可以在 GitHub 上提交 Issue 或者贡献 Pull Request;
2. 在每一节开始, 会介绍本节的内容和知识点, 查看是否与你的研究问题符合; 如果不符合, 返回第 1 步重新查找新的章节;
3. 阅读这一节内容, 并试着针对自己的问题进行操作 (或简单检查自己操作是否正确)。如果报错或出现异常结果, 进行第 4 步; 如果成功, 进行第 5 步;
4. 在该节后面的“错误处理”部分, 会介绍如何处理报错或异常结果, 并给出解决方案。请查找是否有你需要的解决方案, 并尝试解决。如果已经解决, 进入第 5 步; 否则重新查找新的解决方案; 若所有解决方案都无法解决, 请提交 Issue 或者贡献 Pull Request;
5. 放下教程, 继续你的研究; 或者阅读这一章其他内容, 了解其他相关内容。

上述步骤可能 (也一定) 会重复许多次

关于本笔记的版权使用说明

- 本书可免费用于学习, 科研等非商业活动;

²GitHub 仓库地址: <https://github.com/JackyZhang00/Computational-Materials-Tutorial>

- 可以以非商业目的进行传播,但在传播过程中必须保证内容的完整性(截止到最新发布时,包括但不限于仓库内 Latex 源码, pdf 文件等.下同),需保证作者信息完整,不得进行修改;
- 本书不可用于任何商业用途(如确有需要,需联系作者);
- 除在 GitHub 仓库以 pull request 形式进行编辑修改外,不允许进行修改并公开传播私自修改版本(以 GitHub 仓库版本为标准版本);
- 本书著作权归作者(Jiaqi Z.)所有,其他进行创作的人员也可获得著作权,其他著作权所有者不得违反上述版权说明;
- 如因违反上述说明传播而造成不良影响,与作者和其他创作者无关,特此声明;
- 以上说明解释权归 Jiaqi Z. 所有,且如有后续更新,以 GitHub 仓库最新版说明为准.

创作者名单

感谢以下人员参与贡献了内容:

Jiaqi Z.

Isay K.

为创作者而作的教程

0.1 关于如何使用 L^AT_EX 编写模板

本节作者：Jiaqi Z.

在本节，你将要学到：

- 一些基本的 L^AT_EX 语法
- 如何输入公式
- 如何插入代码
- 如何插入图片
- 如何插入创建索引

对于创作者而言，这一部分可以帮助你快速了解 L^AT_EX 的基本语法，帮助你按照规范编写正确的文件。同时对于普通读者而言，这一部分也是你了解本书内容样式的一部分。

因此，任何人都应当阅读这一部分³。

0.1.1 文章结构

请在编写正文内容时，以“节”（section）为单位创建 tex 文档，同时为方便引用，请在每个小节的后面按照 `\label{sec:节标题}` 的格式创建标签。

若需要添加小节，使用 `\subsection{小节标题}` 命令，同时类似于上方关于节标题标签的创建规范，以 `\label{subsec:节标题-小节标题}` 的格式创建标签，方便他人引用。

³对于创作者而言，还应当试着从 GitHub 上寻找源码阅读

对于更小一级的小节 (`\subsubsection{}`), 对标签不作规范。事实上, 我们不建议在引用时涉及到这一层级。通常涉及到小节即可。

注意: 若你需要修改某一节 (或小节) 的标题, 编译后需要确认是否与他人的标签产生冲突 (这通常出现在他人提前按照原有格式引用后发生了修改, 从而导致无法指向正确标签)。因此, 你需要检查编译后文件是正确的, **至少要求通过编译**, 在必要时需要修改他人代码当中的引用标签与新标签一致。

对于正文内容, 请使用正常的 \LaTeX 语法。例如, 当你希望对某一段文字进行强调时, 请使用 `\emph{}` 语句。例如, **这是一句强调的话** 在代码中体现为 `\emph{这是一句强调的话}`。你并不需要关注具体的强调格式— \LaTeX 会按照统一的格式进行编排。

当你希望分段时, 使用空行即可。

对于条目, 请在必要的时候使用 `itemize` 环境 (没有顺序列表) 或 `enumerate` 环境 (有顺序列表)。在环境内使用 `\item` 进行编号。环境之间可以嵌套。例如:

- 列表 1
- 列表 2
 - 1. 列表 2-1
 - 2. 列表 2-2
- 列表 3

在代码中体现为:

```
\begin{itemize}
  \item 列表1
  \item 列表2
  \begin{enumerate}
    \item 列表2-1
    \item 列表2-2
  \end{enumerate}
  \item 列表3
\end{itemize}
```

大多数用法与基本 L^AT_EX 一样, 少有的需要特别注意的是波浪线符号 `~`, 如果使用习惯的打法 `\~` 则会显示较小, 因此, 在输入时请使用修改后的波浪线符号 `\texttt{tilde}`, 或者使用更简洁的形式 `\tttilde`。例如, 在使用 `\~/bin` 显示的结果为 `7bin` 而使用 `\tttilde/bin` 显示为 `~/bin`

0.1.2 一些特殊的格式

在有些时候, 会希望在书后添加一些辅助性的文字说明, 可以使用脚注。脚注应当使用命令 `\footnote{}` 创建, 例如,

这里有一段文字 `\footnote{这里是说明性的脚注}`。

编译后效果为

这里有一段文字⁴。

此外, 为了激发读者思考, 在编写时可以添加一些简单的思考题贯穿于正文中。这些思考题不应当很难 (对于较难的题目, 可以放置在一节后), 应当做到读者经过简单的思考 (约 1 分钟以内) 即可得到正确答案。此时在编写时应当将答案放置在题目后面, 考虑到避免读者直接看到答案, 所有答案都按照特定的格式排版。在编写时应当使用 `\answer{}` 命令, 例如:

这里是思考题。

`\answer{倒着看便是答案}`

排版的效果是

这里是思考题。

这里是思考题。 【这里是答案】

0.1.3 一些特殊的环境

为统一教程格式, 当你希望添加一段让读者注意的文字时, 请使用环境 `attention` 例如, 下面的语句

`\begin{attention}`

当你写注意语句时, 不需要在前面加任何符号。

`\end{attention}`

⁴这里是说明性的脚注

在编译后的结果为：

注意：当你写注意语句时，不需要在前面加任何符号。

类似地，对于一些补充性质的内容，可以使用`extend`环境，例如：

```
\begin{extend}
    这是一段补充的内容，同样不需要在前面加任何符号。
\end{extend}
```

编译后的结果为：

补充：这是一段补充的内容，同样不需要在前面加任何符号。

0.1.4 数学公式

当你希望添加数学公式时，请使用`equation`环境。同时，在使用`\label`语句进行标签注明时，请如同代码所示那样，添加“节标题”避免冲突且方便引用。

```
\begin{equation}
    \label{eqn:关于如何使用LaTeX编写模板-1}
    a^2+b^2=c^2
\end{equation}
```

$$a^2 + b^2 = c^2 \tag{1}$$

在引用公式时，请使用如`\ref{eqn:数学公式-1}`方式进行交叉引用。请勿直接在正文内写编号以免出现引用错误。

注意：在引用公式时，所引用的公式尽量保持在本节内。同时，为避免他人引用，请在编写完成后尽量不要修改相关标签。

为避免公式删除导致的错误，如确实需要引用其他章节的公式，一个较合理的做法是将其他章节的公式在使用时拷贝至当前章节，同时另起标签名。之后的引用限制在当前章节内。

如需要添加多行公式，请使用`gather`环境或`align`环境。例如，

```
\begin{gather}
    a^2+b^2=c^2\label{eqn:关于如何使用LaTeX编写模板-2}\\
    \int_a^b f(x)\mathrm{d}x=F(b)-F(a)\label{eqn:关于如何使用LaTeX编写模板-3}
```

```
\end{gather}
```

$$a^2 + b^2 = c^2 \quad (2)$$

$$\int_a^b f(x) \mathrm{d}x = F(b) - F(a) \quad (3)$$

0.1.5 图片

当添加图片前，请首先在相关文件夹内创建名为`fig`的文件夹，在插入图片时如正常 $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ 代码插入即可。同样，在添加标签时，使用节标题作为开头方便他人引用。即使用下面的代码方式

```
\begin{figure}
  \centering
  \includegraphics[width=1\linewidth]{图片路径}
  \caption{图片标题}
  \label{fig:关于如何使用LaTeX编写模板-标签}
\end{figure}
```

注意：在引用图片时，务必使用`\ref{fig:节标题-标签}`进行引用。例如，图 1。不要使用“如上图”、“如下图”的表述形式，以免图片位置发生移动造成指代不明。

0.1.6 代码

代码使用`lstlisting`环境。

```
SYSTEM = 0
ISMEAR = 0
SIGMA = 0.05
NSW = 1
```

如果希望在代码左侧添加行号以示说明，请在引用环境的右侧添加`numbers=left`设置，即采用下面的代码：



图 1: 图片标题

```
\begin{lstlisting}[numbers=left]
..
```

得到效果如下所示

```
1  SYSTEM = 0
2  ISMEAR = 0
3  SIGMA = 0.05
4  NSW = 1
```

注意：若你希望在代码块中添加代码名称（例如文件名），可以使用 `caption` 选项进行说明，例如，下面的代码：

```
\begin{lstlisting}[caption=简单的 INCAR 文件]
..
```

实际编译后结果为

Listing 1: 简单的 INCAR 文件

```
SYSTEM = 0
ISMEAR = 0
SIGMA = 0.05
NSW = 1
```

0.1.7 添加索引

建议在编写过程中，为文中的程序关键字创建索引。为方便使用，可以使用命令 `\keyword{}`。例如，在 VASP 讲解 SIGMA 函数时，可以使用下面的代码

```
\keyword{SIGMA}
```

从而在表述关键字的同时创建索引。同时，也可以使用 `\keywordin{}{}` 的形式创建带有所属关系的索引，例如

```
\keywordin{INCAR}{SIGMA}
```

便是创建了属于 INCAR 条目下的 SIGMA 索引。

注意：在编写时，鼓励使用索引方便他人根据关键字直接检索。在创建索引时，请提前确认现有的索引表是否有现成的索引。例如，
`\keyword{SIGMA}`和`\keywordin{INCAR}{SIGMA}`会得到不同的结果。

同时，在编写过程中已经将输出和索引集成在一个命令中，你不需要特地再编写一个输出的命令如 `SIGMA\keyword{SIGMA}`，只需要使用
`\keyword{SIGMA}` 便可在输出 SIGMA 的同时创建索引。

对于一些特殊的内容，可能不希望给出索引（通常是命令或者文件名等），它们既不属于关键字，也不属于简单的英文单词。为了将它们区分，使用`\code{}`命令进行编写。例如，`\code{cd \ $OLDPWD}`的输出结果为 `cd $OLDPWD`

注意：在一些特殊的情况下（例如上面的例子），可能会包含 *L^AT_EX* 本身的特殊符号（如 *\$* 本身作为公式符号）。在输入时，应当使用反斜杠 `\` 作为转义字符。

Part I

Linux 基础

Chapter 1

Linux 命令行操作

Contents

1.1 认识 Linux 目录	4
1.1.1 命令格式	4
1.1.2 目录表示方法	5
1.1.3 绝对路径和相对路径	5
1.2 目录操作	6
1.2.1 显示目录文件 <code>ls</code>	7
1.2.2 关于隐藏文件	8
1.2.3 创建目录 <code>mkdir</code>	9
1.2.4 切换目录 <code>cd</code>	9
1.2.5 错误处理	10
1.3 文件操作	12
1.3.1 移动文件 <code>mv</code>	12
1.3.2 如何删除文件 <code>rm</code>	13
1.3.3 如何复制文件 <code>cp</code>	13
1.3.4 一次性处理多个文件	14
1.3.5 错误处理	14
1.4 查看文件	15
1.4.1 Linux 文件类型	16
1.4.2 查看文件内容 <code>cat</code> , <code>tac</code>	17
1.4.3 关于文件后缀名	18
1.4.4 按页查看文件 <code>more</code> , <code>less</code>	18

1.4.5	取头部 <code>head</code> 和取尾部 <code>tail</code>	19
1.4.6	错误处理	20
1.5	压缩与解压缩	21
1.5.1	备份和压缩	21
1.5.2	使用 <code>tar</code> 命令压缩文件	22
1.5.3	解压缩	22
1.5.4	查看压缩文件	23
1.5.5	压缩文件的追加与合并	23
1.5.6	错误处理	24
1.6	文件权限管理	25
1.6.1	用户和用户组	25
1.6.2	文件权限	26
1.6.3	修改文件权限 <code>chmod</code>	27
1.6.4	错误处理	29

1.1 认识 Linux 目录

本节作者: Jiaqi Z.

在本节, 你将要学到:

- Linux 命令格式
- 如何在 Linux 当中表示目录
- 绝对路径和相对路径
- 如何快速表示当前目录和上一级目录

1.1.1 命令格式

与 Windows 使用可视化界面不同, Linux 大多时候使用命令行 (shell) 进行操作。因此, 在 Linux 的学习过程中, 一个最重要的任务, 就是掌握一些常见的 Linux 命令。对于大多数科研课题组而言, Linux 系统都是在远程云端服务器上, 因此在本地往往只需要一个终端程序即可连接到服务器。一些常见的终端软件包括 Xshell、MobaXterm、甚至 VS Code¹等。

¹对于 VS Code 而言, 可能需要扩展插件 (例如 Remote-SSH) 的支持

注意：如果你熟悉其他操作系统，可能听闻过类似于 *Windows Server*，或者 *Linux* 的 *Ubuntu* 这样的操作系统。明明也可以使用可视化界面，为什么在科研过程中从来不会用到它们呢？（更严谨地说，在远程服务器上）。实际上，当使用可视化界面进行远程连接时，所产生的网络资源消耗是巨大的，通常需要更大的带宽，而使用命令行就可以提高数据传输效率。此外，更重要的一点是，使用命令行可以很容易实现批量处理，这在后续的章节会介绍到。

在 *Linux* 当中，输入命令通常采用的格式是命令 [-选项] [参数]，其中中括号表示这个部分是可选的，即可以没有的。例如，当我们希望列出当前目录下所有文件时，可以使用 `ls` 直接输出，也可以使用 `ls -l` 以列表格式输出。

注意：在后面可能会看到选项有多个的情况，此时为了简化，可以将选项合并在一起。例如，`ls -l -a` 可以简化为 `ls -la`。

命令与选项、参数之间是以空格进行分割，且这个空格不能省略。

1.1.2 目录表示方法

在 *Linux* 当中，所有目录都是以根目录/为起点，任何目录都是根目录的子目录。根目录下存在一些固定的目录（这些目录通常有特定的含义），例如，在根目录下有一个叫做 `bin` 的目录（通常写作 `/bin`），它存放的都是二进制文件，也就是系统可以执行的程序文件。

注意：在 *Linux* 当中，任何命令实际上都是可执行程序。你可以在 `/bin` 目录下看到后面所学的所有 *Linux* 终端命令。

另一个比较重要的位置是家目录 `/home`，它存放的是用户个人文件。在这一目录下，你可以看到系统所注册的所有用户名。但是，这些文件夹大多数是无法查看的²。对于用户自己的家目录，通常也可以表示为 `~`。通常来说，当你使用终端等连接登录时，默认的所在目录就是家目录 `~`。

1.1.3 绝对路径和相对路径

任何目录在操作时都具有两种表示方式，绝对路径和相对路径。正如 1.1.2 所介绍的那样，任何目录都是从根目录开始的。因此在描述一个目录时，

²这涉及到 *Linux* 操作权限的问题，通常来说，权限分为三组，即所有者权限、所属组权限和其他用户权限。对于 `/home` 目录下而言，所有目录都是对所有者（即这个用户本身）提供全部权限，而其他人无法访问、修改。

可以从根目录（即/）开始。例如，若你在你的家目录下有一个叫做 `vasp` 的目录，那么它的绝对路径就是 `/home/< 你的用户名 >/vasp`。

随着层级逐渐增多，这种表示方法也会越来越复杂，因此，在表示一个目录时，默认也可以从当前所在目录开始算起（即**相对路径**）。例如，若你刚刚进入终端，此时所在目录就是 `~` 目录，即 `/home/< 你的用户名 >` 下，此时若想表示 `vasp`，则只需要使用 `vasp` 即可。

注意：在这种情况下，你可以将目录 `vasp` 理解为 `< 当前所在目录 >/vasp`，即等价于 `/home/< 你的用户名 >/vasp`。

千万不要写成 `/vasp`，它表示根目录下的 `vasp` 目录。如果你希望特别强调当前目录，可以使用符号 `.`（一个点）表示“当前目录”，即可以写成 `./vasp`

然而，在这种情况下，回到**当前目录的上一级目录**是麻烦的，即在目前所学范围内，只能使用绝对路径。好在 Linux 提供了一个命令：`..`（两个点）表示**上一级目录**。因此，如果你当前处在目录 `/home/< 你的用户名 >/vasp` 当中，则 `..` 表示 `/home/< 你的用户名 >`

同理，`../..` 表示父目录的父目录，在上面的例子中即为 `/home` 目录。

注意：在终端当中，`..`（两个点）表示父目录（即上一级目录），而一个点 `.` 表示当前目录。

这些符号（指令）在后续关于目录操作中都是可以使用的。

看到这里，可以思考下面的问题：如果在你的家目录下有两个目录 `python` 和 `vasp`，此时你在 `/home/< 你的用户名 >/vasp` 目录下，如何可以快速表示 `python` 目录呢（不能使用绝对路径）？

【答案】：`../python` 即可表示 `/home/< 你的用户名 >/python`

1.2 目录操作

本节作者：Jiaqi Z.

在本节，你将要学到：

- 如何显示当前目录下所有文件
- 如何创建目录

- 如何切换至其他目录

1.2.1 显示目录文件 `ls`

在这一节以及下一节，我们将讨论如何对目录和文件做基本的操作。无论是哪一种，一个最基本的前提是知道当前目录有哪些文件和目录，从而才能进行后续操作（例如编辑、删除、移动、进入目录等）

在 Linux 当中，列出一个目录下所有文件使用的是 `ls` 命令。在没有任何参数与选项的前提下，它输出的结果就是当前所在目录下的所有文件和目录。以 1.1.3 一节最后的例子为例，家目录下有 `vasp` 和 `python` 两个目录，当在家目录下执行 `ls` 命令时，结果如下：

```
$ ls
vasp python
```

同时，`ls` 支持在后面添加一个参数表示要输出的目录。例如，在这一例子下，若在家目录当中执行命令 `ls vasp`，将会输出 `vasp` 目录下的所有文件和目录。利用 `..` 表示上一级目录的用法，若当前处在 `~/vasp` 目录下，使用 `ls ..` 便可得到上一级目录（即家目录）下的所有文件和目录。

```
ls -l
```

下面介绍两个常见的 `ls` 选项，首先是 `-l` 选项，它表示以列表形式输出结果。例如，还是上面的例子，使用这一命令的结果为：

```
$ ls -l
total 0
drwxrwxr-x 2 zjq zjq 6 Aug 12 16:35 python
drwxrwxr-x 2 zjq zjq 6 Aug 12 16:35 vasp
```

补充：每一个文件的输出结果可以分为 9 个部分，分别是：权限、文件硬链接数或目录子目录数、拥有者用户名、拥有者所在组、文件大小、文件修改月份、日期、时间、文件名。

关于权限，可以将其分成四部分：第一部分（一个字符）表示文件类型（这里的 *d* 表示目录），第二部分（三个字符）表示拥有者权限（*rw**x* 表示可读可写可执行），第三部分（三个字符）表示组用户权限，第四部分（三个字符）表示其他用户权限（*r-x*）表示可读，可执行但不可编辑。

对于文件硬链接数和目标子目录数，对于初始创建的文件而言，通常为 1，而对于目录而言，默认为 2（因为有两个子目录 *.* 和 *..*）

有时，也可以使用 *ll* 代替指令 *ls -l*，其二者是完全等价的。

ls -a

-a 选项表示列出所有文件，包括隐藏文件。例如，在 *~/vasp* 目录下，使用 *ls -a* 命令，结果为：

```
$ ls -a
.  ..
```

补充：正如前面所介绍的那样，任何一个空目录都会默认有两个隐藏目录—自身和它的上一级目录。而这也解释了 1.1.3 一节所介绍的 *.* 和 *..* 的本质，它们实际上就是任何当前目录下的两个子目录。

注意：前面所介绍的 *-l* 选项和 *-a* 选项是可以合并使用的，此时可以将两个选项之间以空格分割，如 *ls -l -a*，或者将两个选项写在一起 *ls -la*。当选项写在一起时，选项的排列顺序不重要。

与最开始介绍 *ls* 后面加参数表示目录一样，带有选项的 *ls* 同样可以在后面添加参数，例如，*ls -a vasp* 表示列出当前目录下的 *vasp* 子目录下的所有文件和目录（包括隐藏文件）

1.2.2 关于隐藏文件

补充：隐藏文件是指在文件名前面加上 *.* 的，例如 *.bashrc*。

隐藏文件在 *Linux* 当中的常见用途有：

- 配置文件
- 临时文件
- 缓存文件

- 等

总而言之，隐藏文件是为了防止误操作而存在的。（这可能与一些人认为的“隐藏文件是避免别人看到”不同）事实上，哪怕在 Windows 操作系统中，隐藏文件也是存在且方便查看的³

1.2.3 创建目录 `mkdir`

如果所有操作都在家目录下进行，那文件管理就太复杂了。试想一下，在科研里面算了好几年的结果，全部“一股脑”堆在一起，既难找，也容易忘记当时是做了什么。因此，一个好的目录管理至关重要。而前提，就是知道如何创建目录。

在 Linux 当中，创建目录的方法是使用 `mkdir` 命令。与前面介绍的 `ls`，以及后面要介绍的 `cd` 不同的是，`mkdir` 必须带有一个参数，表示创建的目录路径。对于刚开始接触的初学者，一个最简单的命令格式是：`mkdir < 目录名 >`，其中表示在当前目录下创建一个名为 `< 目录名 >` 的目录。例如，希望在当前目录下创建一个名为 `ML` 的目录，则可以使用命令 `mkdir ML`。

正如前面所介绍的路径，`mkdir` 后面的路径也可以是绝对路径或相对路径。无论是哪种形式，其含义是一样的，即在你所描述的路径下创建目录。利用这种方法，你可以在更远的层级关系下创建目录。例如，在 `~/vasp/lattice/Fe` 目录下创建 `~/python/ML/plot` 目录。

注意：你所写的路径名，应当是你所要创建的目录。这句话似乎有点绕，举个例子，如果你希望在 `/home/zjq/vasp` 下创建一个名为 `lattice` 的目录，则你需要运行的命令是 `mkdir /home/zjq/vasp/lattice`。注意到，后面的路径实际上就是你要创建的目录。

1.2.4 切换目录 `cd`

在 Linux 当中，切换目录使用的命令是 `cd`，通常来说，后面需要配合一个参数，表示要切换到哪里。例如，使用命令 `cd /home` 则是将当前目录切换到 `/home` 目录下。配合以 `..`，可以使用 `cd ..` 切换到上一级目录。

思考：如果使用 `cd .`，会得到什么结果？

³在 Windows 操作系统中，可以通过右键-属性-隐藏的方式将文件或文件夹设置为隐藏；相对地，对 Windows10 操作系统而言，可以通过文件夹菜单栏的“查看”-“隐藏的项目”找到那些隐藏文件。只不过在 Linux 当中，隐藏文件使用前面加点的方式设置，但无论如何，隐藏文件永远不是不让别人看见的方法，如果想达成这一目的，正确的方法是设置权限。

【答案】这个命令的含义是切换到当前目录，来自当前目录，最终效果就是什么也不发生。

特殊的，对于家目录而言，除了可以使用 `cd ~` 外，Linux 也支持直接使用 `cd`，不添加任何参数实现这一功能，即二者是等价的。

1.2.5 错误处理

-bash: cd: < 目录名 >: Not a directory

`cd` 后面的参数必须是目录，不能是文件。如果参数是文件，则会报该错误。

如果不知道哪个是目录，哪个是文件，可以使用 `ls -l` 查看第一个字符（文件类型），如果第一个字符是 `d`，则表示目录，如果是 `-`，则表示文件⁴。例如，

```
$ ls -l
total 4
-rw-rw-r-- 1 zjq zjq 4 Aug 12 17:12 INCAR
drwxrwxr-x 2 zjq zjq 6 Aug 12 16:35 python
drwxrwxr-x 2 zjq zjq 6 Aug 12 16:35 vasp
```

表示 `INCAR` 是文件，而 `vasp` 和 `python` 是目录。如果执行了 `cd INCAR`，则会报错。

-bash: cd: < 目录名 >: No such file or directory

这是因为你所要进入的目录不存在。请再次检查你所输入的目录是否正确。

-bash: cd: < 目录名 >: Permission denied

这表明你尝试进入一个你没有权限的目录。例如，在 `/home` 目录下，有 `ljk` 和 `zjq` 两个目录，分别表示两个用户。如果执行 `ls -l` 则会发现：

⁴ 在一些比较新的终端程序中，可能会将文件和目录以不同颜色区分。例如，在 MobaXterm 当中，默认情况下文件是白色，目录是蓝色。当然，这些颜色设置都是可以通过 `Settings-Terminal-Default color settings` 设置颜色主题，这里所说的这一例子为“Dark background / Light text”主题

```
$ ls -l
total 32
drwx----- 13 ljk  ljk   4096 Aug 5 17:34 ljk
drwx----- 75 zjq  zjq   4096 Aug 12 17:12 zjq
```

很显然，每个目录只有目录拥有者自己可以访问。例如，作为用户 `zjq`，当尝试执行 `cd ljk` 时，则会报错。

补充：这种情况有一个特例：`root` 用户。对于 `root` 用户而言，可以进入任何目录。但通常来说，`root` 用户是由服务器管理者所持有的，作为一般用户而言，不需要也不应该进入没有权限的目录，或者执行没有权限的操作⁵。

mkdir: cannot create directory < 目录名 >: No such file or directory

虽然我们说可以用绝对路径或相对路径在更远的层级关系下创建目录。但这一操作的前提是，这个目录的上一级目录需要存在。例如，当你执行 `mkdir vasp/lattice/Fe` 时，首先需要确保目录 `vasp` 和 `vasp/lattice` 存在，才会创建 `vasp/lattice/Fe`。如果你要创建的目录其上一级目录不存在，则会报错。

一个很自然的解决方法是：一层一层创建。这种方法虽然麻烦，但可以确保目录是清晰的⁶。

mkdir: missing operand

很显然，你在使用 `mkdir` 时没有给任何参数。正如 1.2.3所说的那样，在调用 `mkdir` 时必须提供一个参数表示要创建的目录路径。

⁵ “删库跑路”这件事对于一般用户来说是不可能的

⁶ 如果你确实想要一个快捷的方法，可以使用选项 `-p`。这一选项可以在遇到没有的目录时自动为你创建。例如，上面的例子也可以直接使用 `mkdir -p vasp/lattice/Fe`，但这一操作需要保证输入内容是正确的。一旦有内容输错，则极有可能造成目录结构混乱。

1.3 文件操作

本节作者: Jiaqi Z.

在本节, 你将要学到:

- 如何移动文件 (目录), 如何给文件 (目录) 重命名
- 如何删除文件 (目录)
- 如何复制文件 (目录)

这一节, 我们专注于文件的相关操作。类似于 Windows 的基本操作, Linux 的文件操作也无外乎就是**移动**、**删除**、**复制**。同时, 这一节的许多命令对于文件和目录都是适用的, 但可能会有一个注意事项, 这往往会出错。

1.3.1 移动文件 mv

在 Linux 当中, 移动文件使用的命令是 `mv`。其基本用法是 `mv < 源文件路径 > < 目标文件路径 >`。例如, 我们在 `vasp` 目录下, 希望将里面的 `OUTCAR` 移动至上一级目录, 可以使用 `mv OUTCAR ..`。类似地, 对于更复杂的文件移动, 只不过在描述路径时稍微复杂一点, 其他的步骤没有什么不同。

如果你足够敏感, 也许会发现一点问题: 为什么前面的命令, `OUTCAR` 是文件, 而 `..` 是目录? 两个难道不应该统一吗?

对于这个问题, 可以分两个部分讨论: 如果前面是文件, 后面也是文件, 例如 `mv OUTCAR ../OUTCAR`, 这个命令与前面的命令效果是完全等价的。但是, 有趣的地方在于, 如果你试着执行 `mv OUTCAR ../INCAR` 的话, 你会发现, Linux 将 `OUTCAR` 移动到 `..` 的同时, 还将其改名为 `INCAR`。

进一步想一下, 如果我们现在直接写成 `mv OUTCAR INCAR` 的话, 可以将其看作把当前目录下的 `OUTCAR` 移动至当前目录, 同时改名为 `INCAR`, 总的效果就是, 文件被重命名为 `INCAR`。

注意: 正如你所见到的那样, *Linux* 没有单独的重命名文件命令, 而是通过 *mv* 命令来完成。

进一步, 如果前后两个参数都是目录会发生什么呢? 很简单, **就是将前面的目录移动至后面的目录**, 从效果上看, 近似于将第一个参数的目录看作文件。

注意：与文件移动类似的操作，如重命名，对目录的移动同样成立。

1.3.2 如何删除文件 `rm`

相比于移动文件需要两个参数，删除文件的命令 `rm` 只需要一个参数即可，也许你也能猜到这个参数的含义，即 `rm < 删除的文件路径 >`。例如，要删除当前目录下的 `INCAR` 文件，只需要执行 `rm INCAR` 即可。同样的，你也可以使用更复杂的绝对路径或相对路径，例如，删除上一级目录下的 `OUTCAR` 文件，可以使用 `rm ../OUTCAR`。

补充：与 *Windows* 不同，*Linux* 删除文件通常是直接删除，而不是放在所谓的回收站内。因此，在删除文件时务必小心。

在有些版本的 *Linux* (例如 *Ubuntu*) 当中，删除的文件被移动至 `/home/<用户名>/.local/share/Trash/files` 当中，这个目录起到的临时的回收站功能，但你不应该寄希望于这个功能，而是仔细检查删除文件的正确性，并做好合适的备份。

对于删除目录而言，情况有点特殊，需要使用 `rm -r` 命令删除一个目录，此时后面所接参数为目录的路径，例如，删除当前目录下的 `vasp` 目录，则可以使用 `rm -r vasp`。

注意：`-r` 选项通常表示递归，例如，在 `rm -r` 当中，表示递归删除，从而达到删除一个目录的效果。在删除目录时会连同里面的所有内容都删除掉，因此需要特别小心。

如果担心删除错误的文件，可以在选项中使用 `-i`。`rm -i` 表示在删除时询问是否删除。

对于空目录而言，*Linux* 还提供了一个命令 `rmdir`，其用法为 `rmdir < 目录路径名 >`，可以删除一个空目录。

1.3.3 如何复制文件 `cp`

复制文件的命令为 `cp`，其用法与移动文件 `mv` 几乎完全一样，无非就是将移动改为复制。简单来说，语法就是 `cp < 源文件路径 > < 目标文件路径 >`，类似于 1.3.1 当中所介绍的重命名方法，使用 `cp` 命令同样可以做到复制的同时重命名。例如，`cp vasp/OUTCAR ../INCAR` 表示将 `vasp` 目录下的 `OUTCAR` 文件复制到上一层目录，并重命名为 `INCAR`。

如果想要复制一个目录，也需要使用选项 `cp -r`。例如，`cp -r vasp/python/` 表示将 `vasp` 目录复制到当前目录并重命名为 `python`。

注意: 我们在上面的命令当中使用 `vasp/`和 `python/`表示两个目录。其中使用了符号`/`作为结尾, 这个符号通常强调该路径是个目录。对于 *Linux* 本身而言, 有没有这个符号并没有区别。例如, `cp -r vasp python`也可以表示上面的操作。我们这么写只是为了强调这两个路径是目录而不是文件。

1.3.4 一次性处理多个文件

前面介绍的 `rm`, `cp`, `mv`, 以及在 1.2一节所介绍的 `mkdir`, 都是可以针对多个文件同时操作的。以 `rm` 为例, 如果想同时删除多个文件, 只需要在后面添加多个参数即可, 其中参数之间以空格分割。例如, `rm INCAR KPOINTS` 表示删除当前目录下的 `INCAR` 文件和 `KPOINTS` 文件。对于 `mkdir` 创建多个目录而言, 也是一样的用法, 例如, 使用 `mkdir vasp ML` 表示在当前目录下创建 `vasp` 目录和 `ML` 目录。

对于 `cp` 和 `mv` 而言, 情况稍有不同。它们自身就需要两个参数, 第一个是源路径, 第二个是目标路径。如果有多个文件需要处理, *Linux* 默认最后一个路径为目标路径, 前面的所有参数都是源路径。例如, `cp INCAR KPOINTS POSCAR POTCAR ..` 表示将 `INCAR`, `KPOINTS`, `POSCAR` 和 `POTCAR` 复制到上一级目录中。

注意: 对于 `cp` 和 `mv` 而言, 若一次性移动多个文件, 则最后一个参数必须是目录。这就意味着不能进行重命名操作。

1.3.5 错误处理

`rmdir: failed to remove < 路径名 >: Directory not empty`

使用 `rmdir` 命令时, 只能用来删除空目录。当要删除的目录不是空目录时, 执行该命令则会报错。使用 `rm -r < 路径名 >` 往往是删除非空目录的常见方法。

`cp: -r not specified; omitting directory < 路径名 >`

当使用 `cp` 复制目录时, 需要添加`-r` 选项。如果没有添加这一选项则会报错。

cp: target < 路径名 > is not a directory

这通常出现在尝试使用 `cp` 复制多个文件时，最后的参数必须是目录。如果此时不是目录，则会报错。

rm: cannot remove < 路径名 >: No such file or directory

表明你正在使用 `rm` 命令删除一个不存在的文件。请仔细检查你的文件路径名是否正确。

rm: cannot remove < 路径名 >: Is a directory

类似于 `cp` 复制目录，使用 `rm` 删除目录时，也需要添加 `-r` 选项。特别地，对于一次性删除多个文件，如果在删除文件的同时也存在把目录删除的情况，也需要添加这一选项。

rm: remove write-protected regular file < 文件名 >?

当你尝试对没有权限（不可写）的文件进行删除时，会提示该错误。关于权限的内容，将在 1.6 一节详细讨论。在 Linux 当中，是有方法对文件权限进行修改的，但这并不是一个明智的方法。仔细检查文件操作，遵守这些权限，不要“越界”，可以保证你“安全”地使用操作系统（不会引起系统崩溃等严重问题）。

如果你确实需要删除，则只需要输入 `y`（表示“yes”）并回车即可；反之则输入 `n`（表示“no”）。

1.4 查看文件

本节作者：Jiaqi Z.

在本节，你将要学到：

- Linux 文件类型
- 如何查看 Linux 文件

这一节看似知识点不多，但命令还是挺多的。因此，一节只讲这一部分内容完全足够了。

1.4.1 Linux 文件类型

补充：在 1.2.1 当中，我们介绍了 `ls -l` 命令可以以列表形式查看文件。当时仅仅提到，第一个字符如果是 `d` 则表示目录，如果是 `-` 则表示普通文件。在这一部分，我们稍微详细介绍一下更多的文件类型。

- 普通文件 (`-`)：就是普通的文件，通常可以分为文本文件、可执行文件和压缩文件等；
- 目录 (`d`)：在 *Linux* 当中，目录也是一种文件，该文件下存放的是这一目录下的 *inode* 号（又名索引节点）和文件名等信息。当执行打开文件时，*Linux* 实际上是通过 *inode* 号找到当前文件所在 *block*（8 个磁盘扇区组成一个 *block*），从而执行文件；
- 设备文件，又分为块设备文件 (`b`) 和字符设备文件 (`c`)。其中前者可以以“块”为单位进行访问（例如硬盘、软盘等），而后者则是以“字节流”的方式访问（例如字符终端、键盘等）。一般来说，设备文件存放在 `/dev/` 目录下；
- 链接文件 (`l`)：一般情况下指的是符号链接（软链接），类似于 *Windows* 操作系统下的“快捷方式”。创建符号链接的方法是使用 `ln` 的 `ln -s` 选项⁷，例如，`ln -s INCAR INCAR_link` 表示创建了一个指向 *INCAR* 文件的链接文件 *INCAR_link*。当源文件删除时，符号链接文件也会删除；
- 管道文件 (`p`)：通常用于进程间的通信，创建方法是 `mkfifo` 命令⁸，即 `mkfifo fifo_file`。
- 套接字文件 (`s`)：用于通信（尤其是网络上的通信）。简单来说，这是为了避免多个进程或多个 *TCP* 连接同时在一个 *TCP* 协议端口传输数据造成混淆。一般来说，套接字文件包含目的 *IP* 地址，传输层使用

⁷相对地还有“硬链接”，直接使用 `ln` 即可。对于硬链接而言，二者本质上是一个文件（类似于做了备份），当其中一个删除时，另一个不会删除；当其中一个文件修改时，另一个也会同时修改

⁸也许你会疑惑什么是 `fifo` 而不是管道的单词 `pipe`。事实上，*FIFO* 是一种数据缓存器执行方法，即“先进先出”（*First In First Out*）。作为数据缓存器，其与普通存储器的区别是没有外部读写地址线，这样使用起来非常简单，但缺点就是只能顺序写入数据，顺序的读出数据。数据地址在内部由指针自动加 1 实现，而不能通过地址线寻找地址。而 *Linux* 进程间的通信大多就是采用这种通信方式，这种方式也是管道的特性。相对的，还有一种 *LIFO*，即“后进先出”（*Last In First Out*），通常“堆栈”（*Stack*）就是采用这种方法。

协议（TCP 或 UDP）和使用的端口号，利用套接字文件将三个参数组合起来，从而在传输过程中实现并发服务。

1.4.2 查看文件内容 `cat`, `tac`

在整个 Linux 使用过程中，最关键的仍然是普通文件和目录。虽然其他文件对于操作系统本身而言也很重要，但对于非计算机相关专业的科研用户而言，其意义不大。上面的介绍仅仅是作为一个补充。下面地内容将专注在文件本身。首先一个关键的问题是：如何查看文件内容。

注意：当然，从文件操作本身来说，第一件事应当是创建文件。但是，创建文件需要的内容较多（例如，需要一些 `vi` 编辑器的使用，可能还需要重定向命令，在后面的章节再详细介绍。

如果是初学者，希望可以尽快上手的话，你可以试着在 *Windows* 本地用记事本创建一个文本文件，并在里面随意输入一些你喜欢的文字（建议使用英文，对于中文等非 *ASCII* 字符而言，可能会出现乱码。），然后利用远程终端将文件发送至服务器（对于 *MobaXterm* 而言，在终端左侧有一个目录列表，你可以直接将文件拖拽至相应的目录中；对于其他终端软件，请参考其软件具体的操作方法）。后面对文件的查看操作，都可以对这个文本文件进行。

首先需要了解的是，如何查看完整的文件。在 Linux 当中，查看文件内容的命令是 `cat`，其基本用法是 `cat < 文件路径名 >` 例如，对于位于当前目录下的 `INCAR` 文件，可以使用 `cat INCAR` 查看其内容。

`cat` 命令有一些常用选项，例如，可以使用 `cat -n` 或 `cat -b` 显示行号，二者的区别在于前者会显示所有行号，而后者只对有内容的行显示行号。如果文本中空行内容太多，可以使用 `cat -s` 对空行进行压缩，使其缩减为一个空行。

相对地，命令 `tac` 也是查看所有内容，只不过它是从最后一行倒着输出。可以看出，`tac` 本身就是命令 `cat` 倒着写。例如，`tac INCAR` 表示从最后一行开始输出 `INCAR` 文件。

注意：命令 `cat` 并不是单词“猫”的意思，而是连接 *concatenate* 的缩写。正如单词所表示的那样，`cat` 最原始的功能，是连接多个文件。例如，有一个文件叫 `a`，另一个文件叫 `b`，执行命令 `cat a b`，则会将两个文件内容按照顺序连接起来并输出。

1.4.3 关于文件后缀名

对于熟悉 Windows 的用户而言，看到上面（包括之前的所有示例）也许都会有一个疑惑：在 Linux 当中，文件名为什么没有后缀？事实上，后缀名的重要性仅仅是 Windows 操作系统给你的一个“错觉”，让你误以为**后缀名很重要**。事实上，**Windows 操作系统的文件名后缀并不会影响这个文件本身**。

例如，在 Windows 操作系统下创建一个文本文件，后缀名为 `.txt`，此时试着将其后缀名改为 `*.mp3` 或者 `*.jpg` 等，并再次在打开方式中用记事本打开（如果使用默认的打开方式一定会出现错误，例如音乐播放器或者图片查看器等）。可以发现，用记事本打开的结果与之前的文本文件内容是完全一样的。

注意：虽然表示后缀名的，可以任意放置，但有一个地方比较特殊-文件名开头。对于以 `.` 开头的文件名而言，它表示的含义是隐藏文件（这在 1.2.2 一节介绍过了）

对于 Windows 操作系统而言，使用后缀名往往是为了决定文件的打开方式（取决于 Windows 特有的**注册表**）；而 Linux 文件大多都是文本文件（甚至系统配置也是文本文件），因此在 Linux 当中，文件后缀名就变得不重要了。也正因如此，在 Linux 当中你可以类似于 Windows 后缀名的方式创建任何的后缀（`*.jpg`, `*.xyz` 甚至 `*.zjq`, `*.ykn` 都是可以的），在 Linux 看来，它们仅仅是文件名的一部分。

甚至，在 Linux 当中，大多时候文件都是没有后缀名的。这也就是之前的 `INCAR` 和 `OUTCAR` 为什么没有后缀。对于从 Windows 创建的文本文件上传至服务器而言，可能还留有所谓的后缀名 `*.txt`，你完全可以使用 `mv` 命令将后缀名删去，丝毫不影响文件本身和其他命令的运行。

1.4.4 按页查看文件 `more`, `less`

使用 `cat` 和 `tac` 查看文件，都是“一股脑”输出到终端里，对于比较短的文件而言，这种方法是可行的；如果这个文件很长，则要上下翻页就会比较繁杂。

对于多页的文件而言，Linux 可以使用 `more` 命令查看。基本用法是 `more < 文件路径名 >`。例如，使用 `more ../band/OUTCAR` 就可以查看上一级目录下的 `band` 目录下的 `OUTCAR` 文件。在查看过程中，**可以使用空格进行翻页，使用回车进行下一行查看**。

在查看过程中，可以随时使用 `q` 键退出。

对于一些需要来回翻页查看的文件，可以使用 `less` 命令。基本调用格式与 `more` 类似，即 `less < 文件路径名 >`。与 `more` 不同的是，`less` 命令可以向上翻页（使用 `Page Up` 键或者 `b` 键）⁹

注意：对于 `more` 而言，实际上也可以通过 `b` 键实现向前翻一页的效果。但相比于 `less` 而言，`more` 的自由性并不是太高。而且，使用 `b` 向前翻页的效果对于管道文件无法实现。

此外，`less` 还有更复杂的“搜索功能”，例如，可以使用符号 `/< 字符串 >` 的方法实现向下搜索，使用符号 `?< 字符串 >` 的方法实现向上搜索。同时，`less` 的其他命令都是在显示文件后的操作，并不是类似于之前的“选项”（即使用 `-` 的形式），这种方法与 `vi` 的使用类似。

无论是 `more` 还是 `less`，都可以使用 `q` 键退出显示文件。

1.4.5 取头部 `head`和取尾部 `tail`

有时，可能会希望仅仅查看一个文件的开头或者结尾。此时可以使用 Linux 操作系统下的 `head` 和 `tail` 命令。这两个命令的基本调用方法都是一样的，即 `head < 文件路径名 >` 和 `tail < 文件路径名 >`。例如，使用 `head POSCAR` 就可以查看当前目录下 `POSCAR` 文件开头几行，同理，使用 `tail relax/OSZICAR` 就可以查看 `relax` 目录下的 `OSZICAR` 文件结尾几行。

注意：通常情况下，直接调用 `head` 和 `tail` 得到的都是开头（或结尾）10 行的内容。在有些时候，可能会希望输出更多行，或者少输出几行避免混乱。此时可以使用参数 `head -n` 和 `tail -n` 实现，其基本格式为 `head -n < 行数 > < 文件路径名 >` 和 `tail -n < 行数 > < 文件路径名 >`，这一选项表示输出指定的行数。例如，`head -n 5 POSCAR` 可以查看 `POSCAR` 文件开头 5 行。对于 `tail` 同理。

除此之外，`head` 和 `tail` 还提供了选项 `head -c` 和 `tail -c`，表示输出开头（或结尾）多少个字符的内容，格式与上面 `-n` 选项类似，即 `head -c < 字符数 > < 文件路径名 >` 和 `tail -c < 字符数 > < 文件路径名 >`。

⁹除此之外，也可以使用 `d` 向后翻半页，使用 `u` 向前翻半页。

1.4.6 错误处理

cat: < 文件名 >: Is a directory

`cat` 命令仅限于查看文件内容，若后面所接内容为一个目录，例如，`cat vasp/`则会报错

输入 `cat` 命令后忘记输入文件名直接回车，输入文件名后结果只输出了文件名，并没有输出内容

当直接调用 `cat` 而没有接任何参数时，表示将终端标准输入所读取到的内容输出到终端。对于普通调用 `cat < 文件路径名 >` 而言，是将读取到的文件输出到终端。若没有任何参数，则会读取后面输入的内容。

退出的方法则是使用 `ctrl+d` 键结束当前输入，或者使用 `ctrl+c` 键强制终止当前命令。

cat: < 文件名 >: No such file or directory

文件路径不存在，检查一下路径（尤其是当前工作路径）是否正确。

head (或 tail) : invalid number of lines: < 文件名 >

当你使用 `head -n` 或 `tail -n` 时，后面的行数是必须提供的一个参数。若没有提供行数，则会报错。

head (或 tail) : cannot open < 文件名 > for reading: No such file or directory

文件路径不存在，检查一下路径（尤其是当前工作路径）是否正确。

head (或 tail) : error reading < 文件名 >: Is a directory

类似于使用 `cat` 打开目录，使用 `head` 或 `tail` 打开目录就会报这种错误。

使用 `more` 查看文件，输出 ***** < 文件名 >: directory *****

这是因为试着用 `more` 查看目录而不是文件。

使用 `less` 查看文件，输出许多奇怪的路径，不是想要的内容

如果你仔细看一下里面的内容就会发现，当你用 `less` 查看目录时，输出的是这个目录下所有的文件和目录（包括隐藏文件）。事实上，使用 `less < 目录路径 >` 得到的结果和使用 `ls -l < 目录路径 >` 是一样的。只不过前者是单独输出的，而后者是直接输出在终端里。

Missing filename ("less -help" for help)

在调用 `less` 时忘记提供文件路径了。

more: bad usage Try 'more -help' for more information.

与上面的错误类似，在调用 `more` 时忘记提供文件路径了。

1.5 压缩与解压缩

本节作者：Jiaqi Z.

在本节，你将要学到：

- 如何压缩文件
- 如何解压缩文件

1.5.1 备份和压缩

补充：虽然在许多场合，会将 *Linux* 的一些使用 *tar* 的操作说成是压缩文件和解压缩文件，但这个表述实际上是不贴切的。事实上，*tar* 的本意是 *tape archive*，指的是“磁带存档”，是为将若干个文件归档到磁带上，从而方便备份而设计的。而压缩文件实际上在 *tar* 当中经历了另外的步骤，即 *gzip* 压缩，或者是 *bzip2* 压缩等。这些在命令上都是通过额外的选项实现的。

但是，由于现在大多数时候都是习惯于将两个步骤合二为一，包括使用 *gzip* 压缩后得到的 *.gz* 文件也可以在 *Windows* 操作系统下解压缩，从而极大方便了文件之间的跨系统传输。因此，在通常情况下，我们使用到的都是“压缩”。这里之所以给出两者的不同仅作为补充扩展用，在后续表述中，往往不做区分，一律表述为“压缩”和“解压缩”。

1.5.2 使用 tar命令压缩文件

`tar` 命令在使用时通常会配合许多选项，在官方文档中，选项就有 50 个左右甚至更多，因此，我们不可能在这里完全介绍完所有的选项。对于一般的科研工作而言，只需要掌握几个最基本的选项即可。

首先一个最基本的选项是 `tar -c`，表示**创建备份文件**。通常仅有这一个参数是不够的，还需要配合以如 `tar -f` 参数，这一参数表示**指定备份文件**。结合这两个选项，可以得到一个备份文件的基本模式为：`tar -cf < 备份文件路径 > < 要备份的文件 1 路径 > < 要备份的文件 2 路径 > ...`。例如，`tar -cf vasp.tar INCAR KPOINTS POSCAR POTCAR` 表示将当前目录下的 `INCAR`、`KPOINTS`、`POTCAR` 和 `POSCAR` 备份至当前目录的 `vasp.tar` 当中。

注意：`-cf` 后面的参数，除第一个是备份文件路径外，后面所有参数都是要备份的文件路径。

正如 1.5.1 所说的关于压缩和备份的区别一样，我们这里所做的仅仅是备份。对于真正的压缩，我们还需要添加一个压缩格式¹⁰。对于常见的 `gzip` 压缩格式而言，使用的选项是 `tar -z`。因此，一个完整的压缩命令可以表示为 `tar -czf < 压缩文件路径 > < 要压缩的文件 1 路径 > < 要压缩的文件 2 路径 > ...`。

注意：一般情况下，使用 `gzip` 压缩的文件后缀名都是 `.gz`。

对于上面所提到的备份例子，你能想到它的压缩命令是什么吗？

```
tar -czf vasp.tar.gz INCAR KPOINTS POSCAR POTCAR
```

【答案】

1.5.3 解压缩

相对地，有了压缩过程，就一定会有**解压缩**过程。首先，先忽略掉压缩格式（即 `gzip` 等相关内容），仅仅从备份的角度，考虑它的逆过程，也就是**还原文件**。

在 `tar` 当中，可以使用选项 `tar -x` 实现备份文件的还原。例如，在开始的备份操作中，可以使用 `tar -xf vasp.tar` 实现对备份文件 `vasp.tar` 的还原。对于解压缩过程，选项完全类似，只需要使用 `tar -xzf` 即可。例如，

¹⁰所谓的**压缩格式**，在 Windows 系统下常见的如 `zip`、`rar` 等，而在 Linux 操作系统下，最常见的是 `gzip`，当然也有如 `bzip`、`xz` 等。

对上面的 `vasp.tar.gz` 进行解压缩，可以使用 `tar -xzf vasp.tar.gz`。

1.5.4 查看压缩文件

这里所说的查看压缩文件，主要指的是查看压缩包内的文件，从更广义的角度看，就是查看所谓的“备份”文件。

首先，在 `tar` 里面有一个选项 `tar -v`，可以在压缩（解压）过程中查看压缩（解压）的文件。例如，上面的压缩和解压命令，分别可以写成 `tar -cvf vasp.tar INCAR KPOINTS POSCAR POTCAR` 和 `tar -xvf vasp.tar`。对于 `gzip` 格式的压缩和解压缩，只需要在参数里额外添加 `-z` 即可。

另外一种可能的场景是已经存在一个压缩文件（例如 `vasp.tar`），需要查看里面的内容。虽然直接解压查看是一种办法，但如果发现不是想要的文件再删除，就稍显麻烦（尤其是有多个文件时）。因此，Linux 提供了一种类似于 Windows 直接查看压缩包文件的方法，即使用 `tar -t` 表示列出压缩包内文件。

注意：在使用 `tar -t` 时，往往需要配合以 `-f` 参数指定压缩文件名。其完整用法为 `tar -tf < 压缩文件路径 >`。例如，使用 `tar -tf vasp.tar` 可以查看 `vasp.tar` 压缩文件中的文件列表。

无论是普通的备份文件，还是使用 `gzip` 压缩的文件，都是使用 `tar -tf` 查看（没有选项 `-z`）。

使用 `tar -tvf` 同样可以得到文件列表，只是输出的内容更详细（类似于 `ls -l` 的输出结果）

除此之外，查看压缩文件还有一种方法，使用 `less` 命令。通过 `less < 压缩文件路径 >` 可以直接查看压缩文件内容，其形式上类似于 `tar -tvf` 和 `ls -l`。

1.5.5 压缩文件的追加与合并

虽然已经非常小心地创建了压缩文件，但有时还是会有遗漏。例如，当你将 `INCAR`, `POSCAR`, `KPOINTS` 和 `POTCAR` 已经添加到 `vasp.tar` 之后，突然发现还应当把 `CONTCAR` 添加进去。如果此时文件还保留着，当然，重新使用 `tar -cf vasp.tar ...` 也是可以的（其中... 表示五个文件路径）。但是，如果之前的文件已经删除了呢？解压后再重新压缩也不是不可行，但总是麻烦一步。

在 `tar` 的选项中, 提供了一个选项 `tar -r` 表示将文件追加到压缩文件内。例如, 上面的例子, 可以直接使用 `tar -rf vasp.tar CONTCAR` 即可将 `CONTCAR` 添加到 `vasp.tar` 中 (哪怕原先的四个原始文件删除了也没关系)。

上面的例子是将文件追加到压缩文件内, 如果是将压缩文件内的所有文件全部追加到另一个压缩文件里呢? 可以使用 `tar -A` 选项。其格式为 `tar -Af < 追加的目标压缩文件路径 > < 追加的压缩文件路径 >`。例如, 我们已经有包含 `INCAR,KPOINTS,POSCAR,POTCAR` 的压缩文件 `vasp.tar`, 此时又有一个压缩文件 `result.tar`, 里面包含有 `OUTCAR,CONTCAR`, 如何将其合并到共同的 `vasp.tar` 当中呢? 可以使用 `tar -Af vasp.tar result.tar`。

注意: 这里的选项 `-A` 是大写字母, 千万不要写成小写字母。二者的含义不同, 对于小写字母 `tar -a`, 表示根据后缀来决定压缩格式。例如, 使用 `tar -caf vasp.tar.gz INCAR` 将会以 `gzip` 格式创建压缩文件。

同时, 使用 `-A` 合并压缩文件时, 只能对两个文件进行合并。

1.5.6 错误处理

```
tar: < 压缩文件路径 >: Cannot stat: No such file or directory
tar: Exiting with failure status due to previous errors
```

通常这是因为在调用 `tar` 时错误放置了压缩文件路径和被压缩的文件路径的位置。在使用 `tar` 进行压缩时, 第一个参数是压缩文件路径, 第二个参数是被压缩的文件路径。

例如, 对前面的例子, 如果使用的是 `tar -czvf INCAR KPOINTS POSCAR POTCAR vasp.tar.gz`, 就会报错。

```
tar: Refusing to write archive contents to terminal (missing -f
option?)
tar: Error is not recoverable: exiting now
```

在使用 `tar` 进行压缩 (或解压) 时, 需要给定选项 `-f` 并指定压缩文件名, 例如 `tar -cf vasp.tar INCAR`。如果没有选项 `-f` 则会报错。

tar: Cowardly refusing to create an empty archive

这意味着你在压缩文件时试图压缩空的文件。这通常是因为你没有指定压缩文件（例如，直接调用 `tar -cf vasp.tar` 就会报错）。

还有一种可能是你错用了压缩选项 `-c` 和解压缩选项 `-x`。例如，也许上面的命令你是想解压 `vasp.tar`，那么你需要的命令是 `tar -xf vasp.tar`。

tar: < 压缩文件路径 >: file is the archive; not dumped

这可能是因为你在压缩文件时对压缩文件本身进行压缩，这可能会造成递归压缩。例如，`tar -cf vasp.tar vasp.tar` 时就会报错。

但是，压缩文件本身是可以被压缩的。例如，`tar -cf vasp.tar result.tar` 是允许的，这执行的操作是将 `result.tar` 文件压缩至压缩包 `vasp.tar` 当中。

1.6 文件权限管理

本节作者：Jiaqi Z.

在本节，你将要学到：

- 用户、用户组和其他用户
- 如何查看文件权限
- 如何修改文件权限

1.6.1 用户和用户组

Linux 是一个多用户操作系统，因此，如何管理不同用户就成为一个至关重要的话题。例如，在科研过程中，同一课题组的多个成员可能会使用同一个服务器，此时每一个成员就是 Linux 当中的用户。每一个用户通常都有一个主目录，通常为 `/home/` 下的目录¹¹。

在 1.2.1 当中介绍过如何使用 `ls -l` 查看一个文件的完整信息，其中提到了用户组的概念。顾名思义，用户组就是用户的组合。举一个例子：如果

¹¹ 虽尽管我们经常说 `/home/< 用户名 >` 是用户的家目录，但实际上用户在创建时可以通过 `useradd` 命令的 `useradd -d` 选项指定主目录。但除非你是服务器管理人员（具有 `root` 账号），否则用不到该功能。

我们把你家庭的房子看作一个 Linux 操作系统的话，那么你的家人和你就组成一个用户组。而每一个人就是一个用户。对于家庭的共有物品而言（例如空调、冰箱等）是所有家人可以共同使用的，即对整个用户组可用；而相对地，你的房间，你的柜子可能只是你自己可以打开，此时我们说只对某一特定用户可用。

相对地，对于不是你家庭成员的其他人（比如你的邻居等），他们是属于其他用户组的用户，对于你家的所有东西都不可用。

注意：上面的例子也许你还看得“一头雾水”，什么可用、不可用，到底有什么用。事实上，用户组的应用场景大多集中在关于权限的操作上。而这件事则是下一部分的内容。

同时，前面提到的所有用户中有一个特殊用户 `root` 用户。对于他而言，拥有最高的权限和能力，即可以进入任何地方。也正如在前面多次提到的那样，对于一般科研工作而言，不需要了解 `root` 的相关内容。因此在这里，我们就将其略过去了。

1.6.2 文件权限

前面介绍 `ls -l` 命令时已经说明了一些关于文件权限的内容，现在来进一步介绍如何查看文件权限，以及如何理解文件权限的含义。

与前面所介绍的一样，文件权限可以利用 `ls -l` 或者 `ll` 查看。通常来说，每一个文件（包括目录）的第一组字符串表示了文件类型和参考文献。其中文件类型的相关内容已经在 1.4.1 当中介绍过了，现在我们重点关注后面九个字符，即文件权限。

以 `/bin` 目录下的 `cd` 文件为例¹²，使用命令 `ls -l /bin/cd` 可以得到如下结果

```
$ ls -l cd
-rwxr-xr-x. 1 root root 26 Oct 9 2021 cd
```

输出结果的第一个部分就是文件权限。其中第一个字符 `-` 表示这个文件是一个普通文件，后面的 9 个字符，每三个一组，分别表示拥有者，所属用户组和其他用户的权限。例如，对于 `cd` 文件为例，拥有者（即输出结果的第

¹² 正如你所见的那样，`/bin/` 目录下存放着所有系统命令。因此，Linux 的命令行实际上就是调用了这些程序。

三部分 `root`）的权限是 `rw`；而所属用户组（输出结果的第四部分 `root`）具有 `r` 和 `x` 的权限；同样，对于其他用户来说，也是具有 `r` 和 `x` 权限。

补充：在这里你见到了 `ls -l` 的新用法，即在后面添加一个文件的路径。即 `ls -l < 文件路径 >`，可以查看该文件的属性。

同时，也许你注意到上面的第一部分输出结果还有一个 `.`，这表明该文件是在 *SELinux* 模式下创建的。其中 *SELinux* 叫做安全增强型 *Linux* (*Security-Enhanced Linux*)，其目的在于最大限度地减小系统中服务进程可访问的资源¹³。对于使用 *SELinux* 模式创建的文件，在权限后面会有 `.` 作为标志（如同上面的 `cd` 命令一样）

在文件权限中，每种用户又包含有可读权限 (`r`)，可写权限 (`w`) 和可执行权限 (`x`)。可读权限表明用户可以读取文件内容，对于目录而言表示用户可以查看目录内文件；可写权限表明用户可以修改文件内容，对于目录表示用户可以移动、删除目录内文件；可执行文件表明用户可以执行文件（一般为脚本文件或其他程序），对于目录表明用户可以进入目录。

注意：每种用户的权限一共就这三种，且数量和顺序都是固定的（即 `rw`）；对于没有的权限，使用短横线表示无该权限。例如，`r-x` 表示可读可执行，但不可写（如同上面的 `/bin/cd` 一样）。

考虑一个很简单的例子：如果一个文件的权限表示为 `-rw-rw-r--`，对其他用户而言，含义是什么呢？

°梨台呈里只由册其秘书文袋 【孝嘉】

1.6.3 修改文件权限 `chmod`

在 *Linux* 当中，可以使用 `chmod` 修改文件的权限。修改方式有 2 种，第一种是直接设定三种用户的所有权限（共 9 位）。但是，如果直接写类似于 `rw-rw-r--` 这样子的形式的话，就稍显繁琐。如果考虑到特定的位数，可以把这三个权限的开关看作二进制的“0”和“1”，其中拥有权限为“1”。这样子就可以通过 1 个十进制数字表示一种用户的权限。例如，对于 `r--`，可以将其写作 100 也就是十进制的 4；类似地，对于 `r-x`，可以写作 101 也就是 5。

¹³详细的内容可以参考网址 <https://blog.csdn.net/yanjun821126/article/details/80828908> 和 https://docs.redhat.com/zh_hans/documentation/red_hat_enterprise_linux/8/html/using_selinux/index 了解更多。

补充：关于进制转换，对于任意一个 p 进制数， $\cdots a_2 a_1 a_0 . a_{-1} \cdots$ ，其中 $a_n, n \in \mathbb{Z}$ 表示在 $0 \sim p-1$ 范围内的数，将其转化为十进制的方法是

$$\sum_i a_i p^i$$

例如，对于二进制数 101 ，转化为十进制可以算作 $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$ 。对于十进制转二进制，可以使用短除法进行，具体可以参考网站 https://blog.csdn.net/weixin_51472673/article/details/122482602。

在有些专业教材中，可能会使用 $0b$ 表示二进制数，例如 $0b101$ ，其中 b 为二进制单词 *binary* 的缩写。

使用这种数字表示方法修改权限的格式为 `chmod < 权限 > < 文件路径 >`。例如，对 `supercell` 文件执行 `chmod 755 supercell` 则表示最终的权限为 `rwxr-xr-x`。

思考一下，如果想让一个目录只对所有者提供全部权限，而其他人无权限，用数字表示应当怎么写？

-----XMI 兰泽 00L :【瀚景】

虽然这种方法可以修改所有权限，但有时我们仅仅希望添加或删除某一特定的权限。例如，当我们编写了一个程序后，可能只希望给它添加一个对所有者的可执行权限，甚至不关心它对其他用户的权限如何，如果再一点点算，就有点麻烦。例如，原本的权限是 `rw-r--r--`，诚然使用 `chmod 744 < 文件路径名 >` 可以实现这一功能，但有没有更简单的方法呢？

在 `chmod` 命令中，除了使用数字表示权限外，还可以利用如 `+`、`-` 和 `=` 这样的符号进行增添、删除或修改权限。其基本用法为 `chmod [用户]< 操作符 >< 权限 > < 文件路径 >`。其中 `[用户]` 表示想给所有者 (`u`)，所属用户组 (`g`) 还是其他用户 (`o`) 修改权限。对于操作符而言，`+`、`-` 和 `=` 分别表示增加权限、删除权限和更改权限。后面的 `< 权限 >` 使用 `r`、`w`、`x` 这种表示方法。

举个例子，如果我们希望给文件所有者添加一个可执行权限，则可以直接执行 `chmod u+x < 文件路径 >` 即可；如果希望给其他人删除可读权限，则使用 `chmod o-r < 文件路径 >`。如果希望给所有者和所属组设置为可读写权限的话，可以用 `chmod ug=rw < 文件路径 >`。

注意：正如后面的例子所展示的那样，`[用户]` 和 `< 权限 >` 都可以一次性写多个。当然，有时可能会希望一次性给所有用户设置权限，例如给所

有人可读写权限，在 `chmod` 当中，对于 [用户] 还提供了一个选项 `a`，表示所有人。例如，`chmod a=rw < 文件路径 >` 表示设置为所有人可读写。

同时，利用这种方法也可以设置不同的权限，即一次性设置多个用户，其中用逗号分隔。例如，如果希望将权限设置为 `rwrxw-r--`，则可以直接写作 `chmod u=rwx,g=rw-,o=r-- < 文件路径 >`

1.6.4 错误处理

希望给文件所有者自己设置权限，使用数字之后为什么给其他用户设置权限了，而所有者没有权限

在使用数字表示的时候，三位分别表示所有者、所属组和其他用户。对于不足三位的数字，其前面默认补零，例如，`chmod 7 < 文件路径 >` 实际上等价于 `chmod 007 < 文件路径 >`

补充：对于稍微了解计算机的读者来说，也许你已经有所察觉了。我们之前说这个数字是十进制数字，但严格来说，它是八进制数字。对于八进制而言，它的每一个数位上的数字，都与 3 位二进制对应。例如，八进制的 7 表示二进制的 111，八进制的 3 表示二进制的 011 等。具体转换如表 1.1 所示。

十进制	二进制	八进制	十进制	二进制	八进制
0	000	0	4	100	4
1	001	1	5	101	5
2	010	2	6	110	6
3	011	3	7	111	7

表 1.1: 十进制，二进制和八进制对应表

在一些专业的计算机书籍或其他地方，八进制会用 `0o` 作为前缀（其中第一个是数字 0，第二个是小写字母 o）。例如，`0o5=0b101=5`。当然，因为八进制每一个数位的范围 0 到 7 小于十进制 0 到 10 的区间，因此对于 7 以内的数字而言，八进制和十进制是一样的。但随着数字增加，二者会出现差别，但八进制和二进制之间仍存在一一对应关系。例如，`23=0b010111=0o27`。

事实上，每三位二进制的对应关系产生了八进制，而目前更常用的是四位二进制对应关系所产生的十六进制（前缀为 `0x`）。

Chapter 2

文本编辑工具 vi 和 vim

Contents

2.1 使用 nano 简单创建文件	32
2.1.1 使用 nano 创建第一个文件	33
2.1.2 使用 nano 进行查找和替换	34
2.1.3 错误处理	34
2.2 使用 vi,vim 创建文件	35
2.2.1 通过 vi 创建文件	35
2.2.2 vi 编辑器的三种模式	35
2.2.3 通过 vi 打开已有文件	37
2.2.4 错误处理	37
2.3 查找与替换	38
2.3.1 查找	39
2.3.2 替换	39
2.3.3 错误处理	40
2.4 初窥正则表达式	40
2.4.1 关于正则表达式	40
2.4.2 元字符	41
2.4.3 总结	42
2.4.4 错误处理	42

2.1 使用 nano 简单创建文件

本节作者: Jiaqi Z.

在本节, 你将要学到:

- 如何使用 nano 创建并编辑文本文件

在第 1 章当中, 已经了解了如何对 Linux 进行基本的操作, 例如查看目录、移动或删除文件等, 同时在 1.6 一节讨论了如何给文件添加权限, 例如, 给脚本程序添加可执行权限。

然而, 我们在 Linux 的所有对文件的操作, 目前只限于读取, 对于编辑, 目前所采用的方法是将其保存至 Windows 下, 利用记事本等软件进行编辑, 完成后再上传回 Linux 系统。然而, 无论是使用 Linux 本地操作系统, 还是在服务器上使用, 如果可以在系统中直接编辑文件, 显然更方便¹。在本章, 我们将详细介绍 Linux 下如何编辑文件。

目前, Linux 最常用的文本编辑器是 vi 和 vim, 而在这之前, 我们先介绍一个更简单的文本编辑器 nano。相比于 vi 和 vim, nano 功能可能会更少, 但是作为开始 Linux 文件编辑的第一步, 也许是合适的。

补充: 在很多时候, 我们会把 vi 和 vim 放在一起讨论。它们具有类似的界面, 类似的工作模式, 因此很多人容易将其混为一谈。实际上, vi 是由 Bill Joy 在 1976 开发的一款 Unix 操作系统下的可视化编辑器 (Linux 是 1991 年诞生的); 而 vim 是 Bram Moolenaar 在 1991 年开发的 vi 改进版 (Vi improved), 其功能包含语法高亮、插件支持等。

虽然我们经常在 Linux 当中使用 vim, 但它本身是可以跨平台运行的, 如 Windows 本身也是可以安装支持 vim。只不过由于 Windows 本身的文本编辑软件足够丰富, 同时大多数 Windows 用户并不熟悉命令行操作本身。因此, 很多人也是在接触 Linux 的时候第一次接触 vim 编辑器。

关于二者之间的更多区别, 可以查看网址:

https://blog.csdn.net/weixin_53269650/article/details/138137434

¹虽然在 VS Code 当中, 也许可以如同本地文件一般编辑服务器上的文件, 但作为 Linux 教程, 我们还是会尽可能介绍普遍适用的方法。



图 2.1: nano 界面

2.1.1 使用 nano 创建第一个文件

使用 `nano` 命令非常简单，通常只需要使用 `nano < 要打开的文件路径名 >`，对于不存在的文件，它会自动创建一个；而已经存在的文件则会将其打开。

例如，在家目录下，我们直接创建一个名为 `hello` 的文件。使用命令 `nano hello`，则会进入 nano 编辑器的模式。如果你用过老式操作系统，则会发现这个界面十分“复古”——上面是编辑区，下面是一些选项（类似于 Windows 软件的“菜单”）如图 2.1 所示。

当打开时，软件默认就是**编辑模式**，你可以在里面随意输入一些内容，例如，输入“hello world”，屏幕上就是直接显示你的内容。对于删除和换行，其操作就如图在 Windows 下的记事本一样（使用键盘上下左右、删除键等）。重点是下面的菜单选项。难度本身也不大，只需要记住两个符号所表示的含义即可：表 示键盘上的 `Ctrl` 键，而 `M-`表示键盘上的 `Alt` 键。因此，正如你所看到的那样，在 nano 当中，使用 `Ctrl+X`退出；使用 `Ctrl+O`保存。

注意：在 `nano` 当中，一个很特殊的地方在于它的复制、剪切和粘贴与我们所熟悉的快捷键不一样。根据下面的说明，可以看到，复制是 `Alt+6`，剪切是 `Ctrl+K`，而粘贴是 `Ctrl+U`。

同时，无论是复制还是剪切，默认都是**对行进行操作**。也可以使用

Alt+A², 并用方向键选中文本, 进行操作。

此外, nano 也支持撤销 (*Alt+U*) 和恢复 (*Alt+E*)。

2.1.2 使用 nano 进行查找和替换

几乎所有的文本编辑器, 都需要有一些如查找和替换的功能方便我们进行编辑。在 nano 当中, 查找的命令是 *Ctrl+W*, 此时下方会弹出一个输入框, 输入要查找的内容, 回车后光标便会定位在光标下方第一个匹配的开头位置。使用 *Alt+↓* 和 *Alt+↑* 可以切换到下一个匹配位置或上一个匹配位置。

注意: nano 在匹配查找时不区分大小写, 例如, 想查找 SIGMA 时, 在输入查找内容时输入 sigma 同样可以。

对于替换功能, 其命令为 *Ctrl+*, 此时首先弹出对话框, 输入要查找的内容的, 之后弹出的对话框输入要替换的内容。之后光标会从当前位置开始向后搜索, 当查找到一个后会定位到此处并询问是否替换。输入 *y* 表示确认, 输入 *n* 表示不替换此处。如果确认要全部替换的话, 可以直接输入 *a*; 相对地, 如果发现有错 (例如要查找的词语或要替换的词语拼写错了), 可以输入 *c* 取消替换命令。

除此之外, 还有更多的命令 (例如查看字数是 *Alt+D*), 可以直接使用 *Ctrl+G* 查看帮助文档。在帮助文档中还包含有一些命令的快捷方式, 例如查看文档除了可以使用 *Ctrl+G* 外, 也可以直接使用 *F1* 键。

注意: nano 的使用方法看似讲了很多, 实际上只需要记住: 表[^]示键盘上的 Ctrl 键, 而 M-表示键盘上的 Alt 键, 其他的, 都可以通过下方的说明, 或者帮助文档找到。

2.1.3 错误处理

[File < 文件名 > is unwritable]

这是因为你没有这个文件的可编辑权限。借助于 1.6.3 一节所介绍的 *chmod* 命令可以添加可编辑权限。

注意: 大多数时候, 之所以这个文件不可编辑, 是因为这个文件含有重要内容 (可能是你误打了一个系统文件的路径, 虽说这个可能性很小)。因此, 遵守这个权限, 不要修改是最好的。如果确实需要修改, 仔细检查。

²这个命令可能和部分软件 (如微信) 的截图快捷键冲突。

[Error reading < 文件名 >: Permission denied]

这是因为你没有这个文件的可读权限，解决方法与上一个错误一样（使用 `chmod` 命令）

与前面的注意内容一样，遵守这个权限往往是最正确的选择。

2.2 使用 vi,vim创建文件

本节作者：Jiaqi Z.

在本节，你将要学到：

- 如何通过 `vi,vim` 创建并保存文件
- 如何通过 `vi,vim` 打开已有文件

2.2.1 通过 vi 创建文件

从本节开始，这一章就要开始讨论 `vi` 和 `vim` 的操作方法。类似于使用 `nano` 编辑文件，在 Linux 当中通过 `vi(vim)` 创建文件的方法是 `vi < 文件名 >` 或 `vim < 文件名 >`。通常来说，使用 `vi` 创建文件后的界面如图 2.2所示。

注意：仔细看图 2.2标题的话，可能会发现，明明说的是 *vi* 的创建文件，为什么显示的界面是 *vim* 呢？正如 2.1开头所说的那样，相比于 *vi*，*vim* 的功能更加强大。目前在很多操作系统当中，都是使用 *vim* 代替 *vi*。因此，在本节标题中，我们使用 *vi* 和 *vim* 作为区分，在后面的讨论中，可能为了方便，我们使用 *vi* 代替 *vim*（二者操作方法基本一致）。

如果你确实想知道使用 *vi* 命令打开的是 *vim* 编辑器还是 *vi* 编辑器，可以使用 `alias`命令，在输出中如果看到有 `alias vi='vim'`，那么说明实际上你所打开的是 *vim* 编辑器；如果没有，则意味着打开的是 *vi*。此时如果希望打开 *vim* 编辑器，则需要使用命令 `vim` 代替 *vi*。

2.2.2 vi 编辑器的三种模式

与 `nano` 界面相比，`vi` 界面显得更加“简洁”（没有了下方的菜单栏）。但是，如果你尝试着往里面输入内容的话，会发现往往不会是你想要的结果



图 2.2: vim 界面

(也有可能“误打误撞”可以输入进去)。这是因为，在 vi 当中存在三种工作模式：

普通模式

当你使用 `vi` 命令打开编辑器后，则进入了编辑器的**普通模式**。在这一模式下，你可以使用方向键移动光标，也可以进行删除、剪切、粘贴等简单操作。

一些简单的操作是使用 `x` 键删除当前光标所在字符，使用 `dd` 删除当前行（实际上是“剪切”），`yy` 复制当前行；使用 `p`（小写）将剪贴板内容粘贴到光标下方，`P` 大写表示粘贴到光标上方。`u` 表示撤销，`Ctrl+r` 表示恢复撤销。

上面这些操作都是比较基础简单的，通常是用于对文件进行**修改**的。而对于新创建的文件，则可以使用 `i` 进入到“编辑模式”。同时，使用 `a` 可以在光标下一个位置开始“编辑模式”，`o`（小写字母）和 `O`（大写字母）分别表示在当前行下方和上方插入新的一行，并进入“编辑模式”

编辑模式

这是最熟悉的模式。可以在这一模式下如同正常文本编辑器一般进行编辑（例如，方向键移动光标，编辑字符，删除键等都是可用的）。除此之

外，还有一些快捷键需要介绍一下³。

使用键盘上的 `Home` 键和 `End` 键可以将光标定位到行首和行尾；使用 `Page Up` 和 `Page Down` 可以上下翻页；使用 `Insert` 可以在“插入模式”和“替换模式”下切换。

在“编辑模式”下使用键盘上的 `Esc` 键可以返回到“普通模式”。

命令行模式

这一模式将会是最复杂的，许多 `vi` 的高级操作都是基于一系列的命令完成的。进入命令行模式的方法是在“普通模式”下输入键盘上的`:`。

虽然大多数命令要在后面的章节提到它们，但一些必要的命令还是需要现在知道的——它们涉及到文件的保存和编辑器的关闭。例如，`:w`表示保存文件，`:q`表示关闭编辑器，`:q!`表示强制退出（不保存），而`:wq`表示保存后退出⁴。

2.2.3 通过 `vi` 打开已有文件

类似于使用 `nano < 文件路径 >` 的方法，使用 `vi` 打开已有文件的方法是 `vi < 文件路径 >`。与前面所介绍的内容一样，打开后的 `vi` 界面默认是“普通模式”，此时可以使用一些简单的方式（如 `dd` 删除整行等）对文件进行简单的编辑，或者可以使用“编辑模式”进行修改操作。

注意：在修改文件时，请确保是否有修改文件的权限。对于没有权限的文件进行修改，在退出时将会返回“*'readonly' option is set (add ! to override)*”的错误。

正如错误中所说的那样，你可以使用 `w!` 的方式强行覆盖文件，但这始终是一种“下策”。

2.2.4 错误处理

E37: No write since last change (add ! to override)

这表明你在`:q`退出时文件发生了修改。类似于 Windows 操作系统下退出时询问是否保存一样，你需要选择是否保存你的编辑。如果保存，则

³这些快捷键很多在 Windows 当中也有，但可能大多数人并不熟悉。

⁴它还有一个形式：`:x`。

需要先执行:w 再:q, 或者直接执行:wq; 相对地, 如果你不需要保存, 则执行:q! 强制退出。

W10: Warning: Changing a readonly file

这是一个警告信息, 说明你正在编辑一个对你而言有权限限制的文件 (大多数时候是“只读”文件, 但对于某些“不可读”文件, 如果强行编辑, 可能也会引起该错误)。如果无视编辑并保存的话, 通常会引发下面的错误:

E45: 'readonly' option is set (add ! to override)

这是正文最后所提到的错误, 说明你编辑了一个有权限限制的文件。使用:w! 可以强行覆盖保存文件, 但这并不是一个正确的方法 (至少是不推荐的方法)。

[Permission Denied]

这是因为你在查看一个不可读的文件。当你尝试编辑时, 则会引发上面的警告或错误。

2.3 查找与替换

本节作者: Jiaqi Z.

在本节, 你将要学到:

- 使用 vi 的/和? 进行字符串查找
- 使用 vi 进行字符串的替换

对于一个现代文本编辑器, 一个最基本的功能就是对某一特定的字符串进行查找, 以及将其替换为另一字符串。相比于其他在 Windows 操作系统中常见的文本编辑器 (无论是记事本、word、还是 VS Code 等), Linux 的 vi 编辑器下的查找和替换都显得更加复杂。这确实可能带来了一些学习上的困难, 但随着使用场景逐渐复杂, 你会发现这种代码式的操作的便利性。

2.3.1 查找

首先先来了解如何对一个字符串进行查找。在 vi 当中，查找的方法是使用 / 或者 ?，其基本格式为 /[要查找的字符串] 或者 ?[要查找的字符串]。例如，在当前文件中查找 Hello，可以输入 /Hello，然后回车。

注意：在输入字符串时，vi 会同时在文本内将所有匹配的字符串进行高亮（即便没有按回车）。

/ 和 ? 的作用都是查找字符串，二者的区别在于，/ 是从当前光标开始向后查找，而 ? 是向前查找。当输入完成后，点击回车，光标会自动定位到最近的相应位置。若要切换，则可以使用 n 查找下一个或者使用 N 查找上一个。

2.3.2 替换

相比于查找命令，vi 中的替换命令就显得更加复杂了。最基本的命令是 s，但通常会配以更多的命令（类似于参数）。一般来说，替换命令可以用下面的方式表示：:< 开始行号 >,< 结束行号 >s[分隔符][要替换的字符串][分隔符][替换为的字符串][分隔符]<g>。其中 < 开始行号 > 和 < 结束行号 > 都是可选的，若省略则表示只对当前行进行替换。命令结尾的 <g> 也是可选的，表示对所有进行替换，若省略则只替换第一个（每一行或当前行，取决于是否有行号）。

同时，在替换时需要使用 [分隔符] 对字符串进行分割，通常情况下习惯于使用 / 表示，但在一些特殊的情况下（例如要替换的字符串内带有这一字符），则可能会将其改为其他分隔符。命令当中所有出现分隔符的地方都需要统一。

下面是一些例子，例如，若希望将当前行的第一个“hello”替换为“bye”，则需要命令:s/hello/bye/，若希望对所有字符串进行替换，则使用:s/hello/bye/g。

若希望对第一行到第三行的所有“hello”进行替换，则使用:1,3s/hello/bye/g，若没有最后的 g，则表示仅对第一行到第三行每一行里面的第一个字符串进行替换。

如果希望对第一行到最后一行的所有“hello”进行替换，则使用:1,\$s/hello/bye/g。其中，\$表示最后一行。

注意：在 vi 当中，数字可以具有重复若干次的含义。例如，在前面所介绍的 x 表示删除当前光标所在字符，若前面加上一个数字，则表示重复这一操作多少次（即删除多少字符），例如，10x 表示删除 10 个字符。

同时，在 *vi* 当中，往往使用 `$` 表示最后的意思。例如，在普通模式下直接输入 `$` 则直接跳转到这一行最后一个字符，类似的，输入 `0` 则跳转到这一行第一个字符。输入 `:$` 可以直接跳转到文件最后一行。

2.3.3 错误处理

查找（替换）完之后字符串总是高亮显示，怎么将其关闭

使用 `:noh` 命令。

E488: Trailing characters

这可能是在输入命令时使用了错误的格式。请仔细检查使用的命令（尤其是替换命令）的格式

想要替换，却发现把光标上的字符删除了

这是因为在使用替换命令时，前面需要有冒号`:`。若没有添加这一符号，直接使用 `s` 则意味着删除当前字符并插入

2.4 初窥正则表达式

本节作者：Jiaqi Z.

在本节，你将要学到：

- 什么是正则表达式
- 如何使用简单的正则表达式进行查找和替换

2.4.1 关于正则表达式

在 2.3 一节当中，我们提到过，*vi* 的查找和替换相比于其他文本编辑器都稍显复杂。而这一节所介绍的正则表达式，则是其十分强大的功能之一。

简单来说，正则表达式是一种用于匹配和操作文本的强大工具，它是由一系列字符和特殊字符组成的模式，用于描述要匹配的文本模式。借助于正则表达式，我们可以很方便对许多具有相同模式的字符串进行匹配与处理。例如，对于 `ENCUT=200` 和 `ENCUT=400`，从字符串本身来看是不同的，但二

者具有相同的模式（ENCUT= 加上一系列整数字符）。因此，可以使用正则表达式进行批量处理。

在 Linux 当中，正则表达式是相对比较复杂的内容。在这一节只是简单介绍一下基本用法，对于更完整的内容，将在后面章节进行介绍。

2.4.2 元字符

正则表达式最有特色的部分，就是可以使用元字符来匹配一系列特定的字符。在介绍一些复杂的元字符之前，先熟悉一个最简单的符号，`[]`，在中括号里面，可以放入一些字符。正则表达式将会匹配这些字符中的一个。例如，对于字符串“hello”，使用正则表达式 `[aeiou]` 就可以匹配到字符串里面的所有元音字母。

注意：在 *vi* 当中，可以使用正常的查找方式和替换方式，只不过需要在输入查找的内容时使用正则表达式。简单说，你可以将正则表达式看作是一个表达多个字符串集合的方式，而可以使用这种方式一次性对这个集合内的每一个元素进行查找和替换。这样的话，其使用方法就与普通的查找和替换基本无异了。

同时，特别需要注意的一点是，在 *vi* 当中，有一些符号（后面会提到）与 Linux 本身的正则表达式不同（Linux 的命令行本身也是支持正则表达式的），通常区别在于是否添加一个反斜杠（`\`）。后面遇到时会特别指出。

在上面的例子中，我们可以直接在 *vi* 当中直接使用 `/[aeiou]` 实现对所有元音字母的查找。

在使用 `[]` 时，可以使用-对特定范围内的字符进行查找。例如，使用 `[a-h]` 表示对 a 到 h 之间的所有字母（小写字母）进行查找。常用的还有，使用 `[A-Z]` 表示对所有大写字母进行匹配，`[a-z]` 表示对小写字母进行匹配，`[0-9]` 表示对所有阿拉伯数字进行匹配。

补充：也许你会有疑问：这个范围是按照什么排序的？在计算机当中，这些字符都是根据 ASCII 码将其转化为二进制存储在计算机内。因此，这里的排序也是根据每一个字符所对应的 ASCII 码排序的。

ASCII (American Standard Code for Information Interchange, 美国信息交换标准代码) 是基于拉丁字母的一套电脑编码系统。它主要用于显示现代英语，而其扩展版本延伸美国标准信息交换码则可以部分支持其他西欧语言，并等同于国际标准 ISO/IEC 646。

ASCII 由电报码发展而来。第一版标准发布于 1963 年，1967 年经历

了一次主要修订，最后一次更新则是在 1986 年，至今为止共定义了 128 个字符；其中 33 个字符无法显示（一些终端提供了扩展，使得这些字符可显示为诸如笑脸、扑克牌花式等 8-bit 符号），且这 33 个字符多数都已是陈废的控制字符。控制字符的用途主要是用来操控已经处理过的文字。在 33 个字符之外的是 95 个可显示的字符。

例如，0 的 ASCII 码为 48，A 的 ASCII 码为 65，而 a 的 ASCII 码为 97。因此，可以使用 `[0-a]` 匹配到大写字母 A。

同时，中括号里面的字符是可以组合使用的，例如，可以使用 `[A-Za-z]` 表示所有的字母。那如果希望表达所有的字母和数字呢？

`[6-0Z-9Z-V]` **【错误】**

除此之外，对于一些常见的字符，为其设置了特殊的符号，例如，`\d` 就表示所有的数字字符，`\w` 表示所有的字母、数字和下划线，也就等价于 `[A-Za-z0-9_]`。

而在使用中括号时，也可以使用符号 `^` 进行反选。例如，使用 `[A-Z]` 表示排除所有大写字母的字符。

2.4.3 总结

本节简单介绍了一些常见的元字符，并可以将其用于查找和替换。例如，在本节开头所介绍的 `ENCUT=200` 和 `ENCUT=400`，使用正则表达式可以直接表示为：`ENCUT=\d\d\d5`。

正如最开始所说的那样，正则表达式的功能远不止此，对于更复杂的部分（例如，目前使用 `[]` 只能匹配一个字符，如何匹配多个字符？），将在后面的章节进行更加详细的介绍。

2.4.4 错误处理

如何查找如 `[hello]` 这样的字符串？

在正则表达式当中，已经将中括号作为特殊符号使用。因此，如果想查找带有中括号的字符串，则需要将中括号前面添加一个反斜杠 `\` 表示中括号这一字符本身。例如，对于上面的例子，如果直接使用 `[hello]` 表示匹

⁵事实上，它还有更简洁的表示方法 `ENCUT=\d\+`，但碍于本节的内容，详细的含义将放在后面章节介绍。

配这 5 个字母（实际为 4 个）当中的任意一个字符；而使用 `\[hello\]` 或者 `\[hello]` 都可以表示字符串 “[hello]”

Chapter 3

高级 Linux 命令

Contents

3.1 通配符	46
3.1.1 关于通配符 <code>*</code> , <code>?</code>	46
3.1.2 使用通配符进行文件（目录）操作	47
3.1.3 错误处理	48
3.2 <code>grep</code> 匹配字符串	49
3.2.1 使用 <code>grep</code> 查找字符串	49
3.2.2 在 <code>grep</code> 当中使用正则表达式	50
3.2.3 错误处理	50
3.3 <code>sed</code> 文本替换	50
3.3.1 使用 <code>sed</code> 显示	51
3.3.2 使用 <code>sed</code> 添加和删除	52
3.3.3 使用 <code>sed</code> 替换	52
3.3.4 错误处理	53
3.4 管道与重定向	53
3.4.1 管道	53
3.4.2 输出重定向与输入重定向	54
3.4.3 错误处理	55
3.5 简单 <code>for</code> 循环	56
3.5.1 <code>for</code> 语句基本结构	56
3.5.2 关于变量	57
3.5.3 <code>seq</code> 序列	58

3.5.4 一些 <code>for</code> 循环使用例	59
3.5.5 错误处理	60

3.1 通配符

本节作者: Jiaqi Z.

在本节, 你将要学到:

- 什么是通配符
- 如何使用通配符批量处理文件

在第 1 章当中, 我们介绍了如何使用 Linux 的命令行进行简单的操作, 如查看文件、对文件和目录进行操作等。同时, 在第 2 章, 我们详细介绍了如何使用 vi 对文本文件进行编辑。对于绝大多数情况, 以上两个章节的内容足够后续的计算任务了。

但“人的本性终究是懒惰的”, 在大多数时候, 我们可能不希望打开 vi 后再使用/这样的命令来查找, 而是希望直接在命令行查找我们需要的内容。进一步的, 对于更多的文件, 有时我们希望同时对这些文件的相同内容进行查找, 这时在 vi 中操作就显得麻烦了。因此, 在本章, 我们会进一步讨论一些在命令行当中的“进阶操作”, 主要是为了能够更方便的处理文件和数据。

3.1.1 关于通配符 `*`, `?`

通配符是一种特殊语句, 主要有星号 (`*`) 和问号 (`?`), 用来模糊搜索。例如, 当查找文件时, 如果不知道真正的字符, 或者希望匹配一系列具有相似模式的文件时, 可以使用它来代替一个或多个真正字符。

其中, `*` 表示零个或多个任意字符。例如, 使用 `*.txt` 表示文件名最后为 `.txt` 的所有文件, 可以是 `hello.txt`, `bye.txt`, `roselia.txt` 等。

补充: 上面的例子实际上就是后缀名的表示方法。它并不是一个很新奇的事情, 事实上, 在 Windows 操作系统中, 当你在打开或者保存文件时, 都可以在下面的“文件类型”中看到这种使用通配符的表示方法 (如图 3.1 所示)

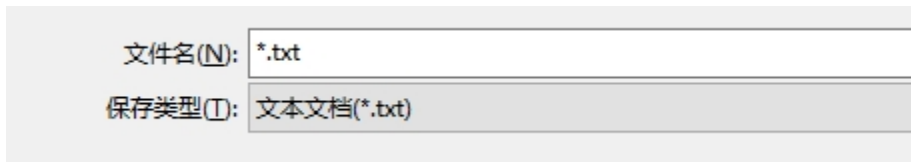


图 3.1: 通配符在 Windows 操作系统下的应用

同时，如果你“思维敏捷”的话，可能会联想到 2.4 一节所介绍的正则表达式。事实上，通配符在某种程度上也是正则表达式其中的一个特例。

相对地，`?` 表示零个或一个任意字符，例如，`he*o` 可以匹配到 `hello`，而 `he?o` 却不能（可以匹配到 `he!o`）

3.1.2 使用通配符进行文件（目录）操作

下面将会简单展示一下，如何使用通配符对文件进行“批量操作”。首先，一个最常见的例子是：批量删除文件。例如，假设当前目录下有 `data1.dat` 和 `data2.dat` 这样两个后缀名为 `.dat` 的文件，如果希望同时删除它们，按照原来的做法，则可能需要按照顺序执行两遍命令：`rm data1.dat` 和 `rm data2.dat`。而借助于通配符，可以只使用 `rm *.dat` 表示删除所有后缀名为 `.dat` 的文件。

注意：在这一行命令里，通配符所表示的文件可以将其“展开”，等价于 `rm data1.dat data2.dat`。

另外，在命令行当中，通配符的优先级比较高，因此，如果一个文件本身含有符号 `*`，则需要使用反斜线（`\`）进行转义，例如，`rm *` 表示删除当前目录下文件名为 `*` 的文件。相反，使用 `rm *` 表示删除当前目录下所有文件（务必要小心这一命令）

同样，使用这一方法也可以批量复制文件，与删除类似，可以使用如 `cp *.dat data` 这样的命令将所有后缀名为 `.dat` 的文件复制到 `data` 目录下。

正如前面所说的那样，这一命令可以将其展开成 `cp data1.dat data2.dat data`

同样，上面对文件名的通配符使用也可以用于目录当中，例如，`rm */*.dat` 表示删除当前目录下的子目录当中所有后缀名为 `.dat` 的文件。

注意：`*` 虽然可以表示零个或多个任意字符，但却不能表示目录下的目录。例如，`rm */data` 可以匹配到 `1/data`，`2/data` 但无法匹配到 `1/1/data`

这样更深一级的目录。

关于使用? 通配符的使用方法, 其基本原理和使用 `*` 类似, 这里不再举例。同时, 你也应该意识到的是, 上面的例子中, 我们都是把通配符放在了开头或者结尾。不一定总是这样的, 例如, 完全可以使用如 `hello*.txt` 这样的方式表示如 `hello.txt`, `hello1.txt` 这样子的文件。

补充: 虽然这一套教程是关于 *Linux* 的, 但在 *Windows* 当中, 通配符同样十分强大。与我们通常使用 *Windows* 的方式不同, 它通常是在命令提示符 (*cmd*) 当中使用的。如果你希望在 *Windows* 当中体验这一功能, 可以在开始菜单搜索 *cmd* (对于新版 *Windows* 操作系统也可以是更高级的 *PowerShell*)。

一些基本的操作语法与 *Linux* 类似, 但有些操作可能有些许区别。例如, 在 *Windows* 的命令行当中, 使用 *dir* 查看当前目录下的文件, 使用 *del* 删除文件, 使用 *copy* 复制文件, 使用 *move* 移动文件等¹。因此, 在 *Windows* 当中, 可以使用如 `del *.txt` 这样的命令删除当前目录下所有后缀名为 `.txt` 的文件, 使用 `move *.jpg jpg` 将当前目录下所有后缀名为 `.jpg` 的文件移动到 *jpg* 目录下。

这种方式可以有效帮助你批量处理电脑中的文件。

3.1.3 错误处理

rm: cannot remove < 路径名 >: No such file or directory

在 1.3.5 一节当中已经介绍了这一错误, 但对于通配符的使用而言, 这种错误更加常见。例如, 上面的例子, 当你试着使用通配符 `*/*.dat` 删除 `1/1/data.dat` 文件时, 由于无法匹配到对应的文件, 因此则会报出这一错误。

cp: cannot stat < 路径名 >: No such file or directory

这一错误与上面的错误类似, 都是由于文件不存在所导致的。对于使用通配符的情况, 请仔细检查文件名是否正确。

¹对于 *cmd* 命令的完整操作, 可以使用 *cmd* 下的 *help* 命令查看。

3.2 grep 匹配字符串

本节作者: Jiaqi Z.

在本节, 你将要学到:

- 如何使用 `grep` 查找文件中的字符串

在 2.3 一节当中, 已经详细介绍了如何在 `vi` 当中进行查找和替换。正如本章一开始所说的那样, 很多时候我们希望在不开 `vi` 的前提下直接找到我们所需要的信息。或者我们希望能够在一系列类似的文件中查找同样的内容², 在命令行下借助于 3.1 一节所介绍的内容, 可以很容易实现这一点。

因此, 本节我们需要了解如何在命令行当中直接查找特定字符串。同时, 我们还将再一次“复习”关于正则表达式的内容 (暂时只会用到 2.4 一节所介绍过的内容)。

3.2.1 使用 `grep` 查找字符串

在 Linux 当中, 查找文本内容的常见命令是 `grep`, 其基本语法为: `grep [匹配的内容] [文件名]`, 其中 `[匹配的内容]` 是要查找的正则表达式, 而 `[文件名]` 则是希望查找的文件。

注意: 虽然说查找的是正则表达式, 但实际上直接输入一个普通的字符串也是可行的。此时查找的内容就是普通的字符串查找。

同时, 查找的文件可以有多个, 甚至可以使用通配符。若没有提供文件名, `grep` 将会从标准输入³中读取内容。

例如, 使用 `grep ENCUT INCAR` 就可以查找在 `INCAR` 文件下所有 `ENCUT`。

同时, 还可以使用 `-r` 参数递归搜索目录下的所有文件。例如, `grep -r ENCUT .` 表示在当前目录下递归搜索 `ENCUT` 字符串。

除此之外还有其他选项。例如, 使用 `-v` 表示查找不匹配的行。例如, `grep -v ENCUT INCAR` 表示打印 `INCAR` 文件中所有没有 `ENCUT` 的行; `-i` 表示忽略大小写匹配; `-n` 表示显示行号输出; `-l` 表示只打印匹配的文件名。

其中多数选项都是可以混用的, 例如, `grep -l -r ENCUT .` 表示什么含义?

²这在后续 VASP 批量处理中可能十分有用, 例如, 我们希望查找所有 `INCAR` 文件当中的 `ENCUT` 设置情况, 就需要这个方法。

³从终端、键盘输入

【答案】递归查找当前目录下所有含有 ENCUT 的文件，并只输出文件名。

3.2.2 在 grep 当中使用正则表达式

在 grep 当中，使用正则表达式有两个地方：查找内容和文件名。关于文件名，大多数内容都如同 2.4 一节和 3.1 一节所介绍的那样，例如，使用 `grep ENCUT */INCAR` 表示查找当前目录所有子目录下名为 INCAR 的文件当中含有 ENCUT 的行。

除此之外，在查找的内容当中，也可以使用正则表达式，此时的用法就完全类似于在 vi 当中使用正则表达式进行查找（见 2.4）。

3.2.3 错误处理

grep: < 路径名 >: Is a directory

这表示你尝试在目录当中查找字符串，这显然是行不通的。如果你希望查找这一路径下所有文件，可以使用 `grep -r < 字符串 > < 目录名 >` 或者 `grep < 字符串 > < 目录名 >/*`。

3.3 sed 文本替换

本节作者：Jiaqi Z.

在本节，你将要学到：

- 如何使用 sed 命令对文本文件进行替换
- 如何使用 sed 查看文件内容
- 如何使用 sed 添加（删除）文本文件的内容

在 2.3 当中，已经介绍了如何使用 vim 进行文本文件的替换。与 vi 里面的 s 命令类似，在命令行也有类似的方式可以对文本文件进行替换（甚至更复杂的操作）。那就是使用 sed 命令。

3.3.1 使用 sed 显示

sed 的语法格式是：sed [选项] "< 动作 >" [文件路径]。其中，常见的选项包括

- -n 表示只有处理的文本显示在屏幕上（默认是全部文本）
- -i 表示直接修改文件内容。

注意：在本节，我们所处理的文件都会使用 -i 选项直接对文件进行处理。但这一行为是危险的，尤其是对于不确定的修改操作，由于它是对原文件进行修改，因此需要提前备份。此外，千万不要将 -n 和 -i 结合起来使用，它表示仅将处理的文本修改为文件内容。

在下一节将介绍管道运算符和重定向运算符，将会帮助你避免这一问题。

补充：如果你仔细观察上面的语法格式，需要特别注意的是里面的 [文件路径] 是“可选”的，也就是说，对于 sed 命令而言，可以不提供文件路径。这件事可能会比较“难以置信”，毕竟如果没有文件，怎样处理文件呢？

事实上，sed 命令是一个管道命令，简单来说，它是可以读取并处理终端输出的内容。除此之外，像前面的 grep 也是类似的（虽然我们前面并没有特别强调这一点）。

在下一节，我们将详细讨论管道命令的内容。

让我们先来看一个最简单的操作——打印，即试着将特定几行的内容打印在屏幕上。其在“动作”部分的写法是 < 开始行号 >, < 结束行号 >p。例如，我们希望将 POSCAR 的第 3-5 行输出，可以使用命令 sed -n "3,5p" POSCAR 表示输出第 3-5 行内容。

补充：如果你此时忘记了 -n 选项的话，命令将会输出所有的 POSCAR 文件，但对应行号的内容会重复输出。（可以试试）

另外，如果希望对最后一行进行操作，需要使用 \$ 符号表示最后一行，但此时需要将外面的双引号变为单引号。通常情况下，双引号和单引号是等价的，但在涉及到 \$ 符号时，使用双引号则表示将 \$ 和后面的符号（例如 \$p）解析成变量 p，而使用单引号则表明这一符号就是符号本身。变量的作用，将在后面循环和批量处理中体现，在此不做讨论。

3.3.2 使用 sed 添加和删除

使用 sed 命令删除某一行的方法是使用 d, 例如, 如果希望删除 POSCAR 文件的第 3-5 行, 类似于上面的显示, 使用方法是 `sed -n "3,5d" POSCAR`

注意: 这一操作是对文件内容进行删除, 因此你应该谨慎使用 `-i` 选项 (会直接将原文件对应内容删除), 在下一节的重定向运算符将会提供灵活的解决方法。目前的一个方法是 创建一个备份文件。

对于添加某行内容, 可以使用 a 或 i, 前者表示 新增 (在下一行), 后者表示 插入 (在上一行)。具体的, 其命令为: `sed [选项] " 行号 a(i) 添加字符串" [文件名]` 例如, 希望在 POSCAR 的第一行后面添加一个字符串 "1.0", 可以使用 `sed -n "1a 1.0" POSCAR`。

那么, 如果希望在第一行添加内容, 应当怎么办呢?

`sed -n "1i 1.0" POSCAR` 【参考】

3.3.3 使用 sed 替换

使用 sed 替换有两种模式, 分别是 整行替换 和 字符串替换。对于整行替换, 命令为 `sed [选项] " 行号 c 替换目标字符串" [文件名]`。例如, 若希望将 POSCAR 的第 2 行替换为 "0.8", 则可以使用命令: `sed -n "2c 0.8" POSCAR`。

相比于整行替换, 字符串替换可能更为强大。与 vim 的替换类似, 在 sed 当中替换字符串的命令是 `sed [选项] "s/要被替换的字符串/新字符串/g"`。例如, 希望在 INCAR 当中的 `ENCUT=400` 替换为 `ENCUT=600`, 可以使用命令 `sed "s/ENCUT=400/ENCUT=600/g" INCAR`。

对于字符串替换而言, 更重要的是 支持正则表达式 (关于正则表达式的内容详见 2.4 一节), 例如, 在上面的例子中, 我们希望将 `ENCUT=400` 或 `ENCUT=600` 全部替换为 `ENCUT=800`, 则可以使用 `sed "s/ENCUT=\ d\ d\ d/ENCUT=800/g" INCAR`

注意: 同样, 由于这是对内容的删除 (更改), 因此需要提前备份。除非有足够的把握, 否则不要使用 `-i` 选项。

3.3.4 错误处理

`sed: -e expression #1, char 2: invalid usage of line address 0`

在使用行号时，第一行是 1 而不是 0。通常来说，当你试图在第 1 行插入数据时，应当使用 `sed -n "1i 字符串" [文件名]`，其中 `i` 表示在第 1 行前面插入数据（即在第一行写入字符串）

3.4 管道与重定向

本节作者：Jiaqi Z.

在本节，你将要学到：

- 管道运算符、如何使用管道运算符连接多个命令
- 使用重定向写入（读取）文本文件

3.4.1 管道

在 3.3 一节当中，我们讨论了如何使用 `sed` 命令对文本文件进行编辑。当时我们提到，`sed` 命令是一个管道命令，可以读取终端输出的内容。除此之外，在很多时候我们希望对某一文件进行多次操作，例如，提取某一文件的前 10 行，并将其中的“C”改成“B”，然后再写入新的文件当中。

像上面这种输入和输出层层传递的（一个命令的输出作为另一个命令的输入），就可以使用这一节的“管道”命令。在 Linux 当中，管道运算符是 `|`（通常是 Shift+Backspace 下面的那个键），它的作用是把前面命令的输出作为下一个命令的参数输入。例如，我们希望将 `POSCAR` 输出，可以使用 `cat POSCAR`，若此时又想取出前 5 行，则可以使用 `cat POSCAR | head -n 5`

注意：也许使用 `head -n 5 POSCAR` 一样可以解决上述问题，但使用管道更具有“可扩展性”。例如，`cat` 命令本意是将多个文件连接起来，若希望将连接之后的文件读取前 5 行，则使用管道运算符是最简单的方法之一。

上面的过程，可以看作是将 `cat` 命令输出结果作为 `head` 命令的参数输入，之后运行 `head` 命令并输出（至标准输出），进一步，如果我们希望

将其中的“C”全部替换成“B”，则需要借助于 `sed` 命令，表示为：`cat POSCAR | head -n 5 | sed "s/C/B/g"`

注意：此时使用 `sed` 命令由于没有添加 `-i` 选项，因此结果也仅仅在标准输出当中进行输出，源文件并没有修改。

类似地，如果我们希望得到 OUTCAR 最后一个包含“without”字符串的行，在使用管道之前是“几乎不可能”的，而在利用管道时，便可以使用命令：`grep without OUTCAR | tail -n 1` 得到结果⁴。

3.4.2 输出重定向与输入重定向

在之前，我们得到的输出结果仅仅是在屏幕上输出（也被称为“标准输出”），但必要的时候，我们也希望将结果保存至本地，以便后续处理（无论是进一步使用程序语言读取并处理，还是过一段时间再看）。正因如此，寻找一种方法将输出结果保存就十分重要。

在 Linux 当中，保存终端输出的本质就是“将输出**重定向**至文件”，其运算符是 `>` 或 `>>`，后面需要有一个文件名路径表示**希望写入的文件**。其中，前者（`>`）表示**创建**，当文件存在时则会覆盖；后者（`>>`）表示**追加**，当文件不存在时新建，存在时则会在后面追加新的内容。

有了这一方法，我们终于可以解决 3.3 一节所遗留的关键问题：如何将编辑后的文件保存至新的文件？答案就是**使用重定向运算符**。例如，我们希望将 INCAR 文件里面的 `ENCUT=400` 改为 `ENCUT=600` 并保存至 INCAR2，则可以使用命令：`sed "s/ENCUT=400/ENCUT=600/g" INCAR > INCAR2`

再一个例子，如果希望将 Si/POTCAR 和 O/POTCAR 合并至一个新的 POTCAR 当中，应当怎样写呢？

【答案】`cat Si/POTCAR O/POTCAR > POTCAR`

补充：请注意运算符是 `>` 而不是 `<`，前者表示“输出重定向”，而后者表示“输入重定向”，即将文件的内容作为命令的输入。例如，`cat < POSCAR` 与 `cat POSCAR` 等价。

二者的方向虽然容易混淆，但似乎可以从箭头的方向看出一点规律——`>` 表示将命令的内容“输出至”文件中，而 `<` 表示将文件“输入至”命令中。

⁴这一过程实际上是在 VASP 当中得到最后收敛能量的过程。

在必要的时候，我们当然也可以将输入和输出同时重定向，例如，`cat < POSCAR > POSCAR2` 也是可行的（将 `POSCAR` 重定向输入至 `cat` 命令，并将命令输出结果重定向至 `POSCAR2` 输出）

补充：除了“标准输入”和“标准输出”之外，*Linux* 还有一个“标准错误输出”，用来输出命令运行报错的结果。例如，当我们希望删除一个名为 `POSCUT` 的文件时（该文件并不存在），使用 `rm POSCUT` 会报错（这一点在 1.3 已经详细讨论过了）。如果试图将这一输出重定向，例如，`rm POSCUT > output`，效果是一样的。但如果使用 `rm POSCUT 2> output` 呢？你会发现，终端没有报错了，而将报错输出至文件 `output` 当中了。这其中，`2>` 就表示将“标准错误输出”重定向至后面的文件。

在这一基础上，稍微扩展一下。在后面的 *VASP* 教程中，我们将看到提交脚本中有 `mpirun vasp_std > vasp.out 2>vasp.err` 这一行⁵。暂且忽略掉前面的 `mpirun`（表示分布式计算系统下并行运行任务），可以发现，这一行的作用就是运行 `vasp_std`，并将输出结果输出至 `vasp.out`，而将错误信息输出至 `vasp.err`。

另外，我们这里重定向的文件并不一定是文本文件。如果你回顾一下 1.4.1 一节，可能会发现里面有一个“设备文件”，它也是可以作为重定向输入输出的一部分。例如，`rm POSCUT 2> /dev/null` 则表示将输出报错信息重定向至 `/dev/null` 设备（这是一个“空设备”，用于消除所输出的内容）。运行这一命令，你将不会得到任何输出结果（输出被消除了）

3.4.3 错误处理

使用 `cat << POSCAR` 没有反应

如果你确实希望使用输入重定向，应当注意是 `<`（一个）而不是 `<<`（两个）。后者在 *Linux* 中通常用于终端交互中，例如，上面的命令表示将 `POSCAR` 之间的内容作为输入。一个演示例子为：

```
$ cat << POSCAR
> hello
> world
> POSCAR
hello
```

⁵不同课题组可能有所不同，这里仅仅作为例子。

```
world
```

可以看到，它把“POSCAR”之间的内容传入了 `cat` 命令（输出）

3.5 简单 for 循环

本节作者：Jiaqi Z.

在本节，你将要学到：

- 命令行 `for` 循环的基本结构
- 如何使用 `for` 循环批量处理任务
- `$OLDPWD` 变量的使用

在这一章的最后，让我们看一下关于命令行的最后一个话题——**批量处理**。其中，批量处理的一个基本方法是使用 `for` 循环语句。

注意：更复杂的批量处理可能需要配合下一章介绍的 `bash` 批处理脚本，那里面会有进一步复杂的如条件判断等。一般来说，更复杂的批量处理会放到脚本中执行。在本节，我们仅仅讨论一些能使用 `for` 循环简单处理的任务。

补充：虽然说是“最后一个话题”，但实际上关于命令行的使用远不止此。只不过目前教程（或者说科研过程中）可能涉猎到的也就这些。一些更复杂的，或者更细节的使用，例如系统本身操作等，我们并没有对此进行讲解。对于需要的人来说（例如服务器维护相关人员），请查阅更专业的书籍或相关手册了解更多如系统目录架构，以及服务器维护，`root` 权限等相关内容。⁶

对于一些后续可能会用到的命令，会有新的补充，请继续关注仓库内版本更新，或者网页端动态更新。

3.5.1 for 语句基本结构

在 Linux 当中，`for` 语句的基本结构为：`for [变量名] in [列表范围]; do [要执行的语句]; done`。其中，`[列表范围]` 可以使用大括号将其

⁶我们不讲解还有一个原因：担心一般使用时出现错误导致服务器崩溃等异常情况出现。

依次列出，例如 {a,b,c,d,e} 表示遍历这五个字母。同时，在 do 后面需要是按顺序执行的语句，其中每一个语句后面以分号 (;) 结尾。

注意：在 Linux 当中，分号表示一个命令的结束，对于最后一个语句可以不加分号（此时回车表示结束）。例如，希望一次性输出 POSCAR 和 CONTCAR 文件的最后几行，可以一次性运行两句命令 `tail POSCAR; tail CONTCAR`，其中分号表示这是两个语句。

在 for 循环中，你也可以这样理解分号，其中 done 前面只需要有一个分号（表示语句与 done 的分割）。换言之，不可能有两个分号相邻。

3.5.2 关于变量

如果只是重复执行几个没有丝毫变化的语句，似乎有点“无趣”。但我们如何在命令中加入一些可以变化的东西呢？那就是变量。在 Linux 命令行当中，变量是以 \$ 符号表示的。例如，\$i 表示变量 i。因此，如果我们希望一次性创建 a 到 e 五个目录，则可以使用下面的语句：`for i in a,b,c,d,e; do mkdir $i; done`。其中，`for i in a,b,c,d,e` 表示遍历后面的列表，并分别将变量 i 赋值为列表中的元素；后面的语句表示执行 mkdir 命令，但其中的参数为变量 i，例如，第一次时执行的为 `mkdir a`，第二次就为 `mkdir b`，以此类推。

补充：在所有变量中，有一些变量是比较特殊的，它们具有特定的含义。比较常见的如 \$PWD 表示当前所在的目录路径；\$OLDPWD 表示上一个所在的目录路径。

在一般的命令行中，你也可以在必要的时候使用这些命令（哪怕不是在循环中）。例如，`echo $PWD` 表示打印当前路径⁷。（事实上，你也可以直接使用 `pwd` 命令快速实现这一功能）

特别注意的是，\$OLDPWD 表示上一个所在的目录路径，而不是上一级目录。例如，原先在 1/a/ 目录下，当你使用 `cd 1/b/` 切换到 1/b/ 目录时，变量 \$OLDPWD 表示的是 1/a/ 而不是它的上一级目录 1/。使用 `cd $OLDPWD` 可以帮助你快速切换到上一个操作的目录，但还有一个更简单的方法是使用 `cd -`，这二者是等价的。

注意：就目前写作的过程中，我还没有具体了解到如何在 for 循环中使用两个变量。类似于 `for i,j in a,b,c,1,2,3` 这种写法是不可行的。

⁷echo 表示输出显示某一个内容，其后面的参数表示输出的内容。

如果你了解到了具体实现这一效果的方法（在命令行下），请通过前面的联系方式联系我。

3.5.3 seq 序列

在前面使用 `for` 循环时，我们需要把遍历元素全部列举出来。通常这种方法适合于一些没有特定模式的序列，且数量较少。对于数量较多，或者我们明确知道其规律（通常是一些数字序列），一般会选择使用序列的方式让命令行自动生成。在 Linux 当中，最普遍的**数字序列**生成的方法是使用 `seq` 命令，其形式为 `seq < 起始数值 > < 步长 > 终止数值`。当只有一个参数时，起始数值和步长默认为 1；两个参数则分别表示起始数值和终止数值（此时步长默认为 1）。

当希望在 `for` 循环中使用序列时，需要将 `seq` 命令（包括后面的参数）使用反引号（```）括起来。例如，`for i in `seq 1 2 10`` 表示对序列 1,3,5,7,9 进行遍历。

注意：正如你所见，`seq 1 2 10` 表示序列 1,3,5,7,9 而不包括 11，这是因为 11 超过了终止数值 10。而对于 `seq 1 2 11` 则包括 11。这一点可能与其他编程语言如 *Python* 不同，在 *Python* 当中，`range(1,11,2)` 不包括最后的 11。

`seq` 命令可以使用负步长表示递减序列。例如，`seq 5 -1 1` 表示序列 5,4,3,2,1。

同时，`seq` 也可以使用浮点数，例如，`seq 0.1 0.1 0.5` 表示序列 0.1 到 0.5，间隔 0.1。

除此之外，对于一些整数的序列，也可以使用更简单的方式，其形式为 起始数值..`终止数值`..`< 步长 >`。例如，`for i in 1..10` 表示对 1 到 10 进行遍历；而 `for i in 1..5..2` 则对 1,3,5 遍历。

与 `seq` 不同的是，使用..`的表示方法还可以对字母序列进行表示，例如，a..z 表示所有小写字母，同理 A..Z 表示所有大写字母。`

注意：同样的，你也可以使用 `A..z` 对大写字母和小写字母进行表示，但此时还包括一些特殊字符如 `[,]` 等，它们是在 *ASCII* 码介于字母中间的部分（91 到 96）。

特别注意的一点是，使用..`的方法不能对浮点数进行操作。当然，对于负步长仍然可用，例如，5..1..-1 表示序列 5,4,3,2,1。但对于这一方法，更特别的是你可以省略后面的步长（加上也不错），Linux 会自动做降`

序。更甚者，你可以使用 `10..2..2` 的方法来生成 `10,8,6,4,2` 这一序列（此时步长完全是错误的）。

最后再来总结一下，对于整数序列，无论使用哪种方式都能得到；而对于小数（浮点数）序列，只能使用 `seq` 方法；对于字母序列则只能使用 `..` 方法生成。

3.5.4 一些 for 循环使用例

下面，讨论一些 `for` 循环的使用方式，仅供启发用（实际使用时可能需要根据实际情况进行个性化调整）。

首先，如果我们希望生成序列为 200,400,600,800 的目录，则可以使用命令 `for i in 200..800..200; do mkdir $i; done`。正如你所见的那样，这一命令表示对于这样一个序列进行遍历，执行 `mkdir` 命令，其中参数为变量 `i`，依次为 200,400,600,800（前面的序列）。

下面，假设我们在当前目录下有一个文件 `INCAR`，希望将其拷贝至这里面每一个目录下，可以使用命令 `for i in 200..800..200; do cp INCAR $i; done`。其中表示对序列的每一个变量 `i`，执行 `cp` 命令，其中参数（拷贝目标路径）分别设置为序列里的元素。

最后，当我们希望将每个目录里面的 `INCAR` 里面的 `ENCUT = 200` 分别改为 `ENCUT = [目录名]`，可以使用下面的方法：`for i in `seq 200 200 800`; do cd $i; sed -i "s/200/$i/g" INCAR; cd $OLDPWD; done`。这表示对前面的序列⁸，依次执行下面的操作：

1. 进入其目录；
2. 使用 `sed` 命令修改（使用方法详见 3.3 一节），其目标文本为变量 `$i`，也就是目录名；
3. 返回到之前的目录（以便于能够继续遍历其他元素，进入对应目录）

补充：你可以试一试，如果没有后面的 `cd $OLDPWD`，会发生什么？稍加分析可以看出，当进入 200 目录时，修改完毕之后，会进入下一次循环遍历，此时尝试执行 `cd 400`。但当前目录下并没有该目录（200 和 400 是并列关系，不是父子目录关系），因此程序会报错。具体的报错可以看下面的 3.5.5 一部分。

⁸这里我们使用 `seq` 方式生成，仅是为了多一种展示，也可以使用 `200..800..200`，它们是一样的。

3.5.5 错误处理

-bash: cd: < 目录名 >: No such file or directory

这就是前面所说的忘记 `cd $OLDPWD` 的错误信息，此时会发现，没有目录，怎么办？只能报错了（如果你足够“机敏”，也许你会联想到 1.2 一节。没错，在那里的 1.2.5 当中，也有这一错误。实际上，二者的本质是相同的。

seq: invalid floating point argument: < 字符 >

这是因为你在使用 `seq` 时尝试生成非数值（整数或浮点数）序列。例如，你也许本意是生成字母 a 到 z 的序列，但使用 `seq a z` 则会产生上面的错误。正确方法是使用 `a..z`

使用..的方法生成浮点数，结果错误

使用 `..` 不能生成浮点数。因此，如果你希望生成浮点数，一种方法是使用 `seq`，当然，还有一种“投机取巧”的方法，即尝试将浮点数用整数表示。例如，如果希望生成 0.1,0.2,0.3,0.4,0.5 的目录，除了使用 `for i in `seq 0.1 0.1 0.5`; do mkdir $i; done` 外，还可以使用 `for i in 1..5; do mkdir 0.$i; done`。其中 `0.$i` 的 `$i` 需要替换为遍历的序列（整数），也就是 0.1,0.2,0.3,0.4,0.5。

Chapter 4

Shell 脚本基础

Contents

4.1	第一个脚本	62
4.1.1	编写第一个脚本	63
4.1.2	添加至环境变量	65
4.1.3	错误处理	66
4.2	变量	66
4.2.1	定义变量与初始化	66
4.2.2	调用变量	67
4.2.3	字符串中的 \$ 符号	69
4.2.4	变量简单运算	70
4.2.5	错误处理	71
4.3	输入	72
4.3.1	用户输入	72
4.3.2	参数输入	73
4.3.3	读取命令作为输入	75
4.4	判断语句	76
4.4.1	if 语句	76
4.4.2	关系运算符	77
4.4.3	带有 else 的判断语句	82
4.4.4	嵌套 if 语句	86
4.4.5	逻辑运算	87
4.4.6	错误处理	88

4.5 case 分支语句	88
4.5.1 case 语句基本结构	89
4.5.2 case 语句应用	90
4.5.3 错误处理	94
4.6 循环	94
4.6.1 for 循环	95
4.6.2 while 循环	97
4.6.3 until 循环	98
4.7 循环控制	100
4.7.1 使用 break 停止循环	100
4.7.2 使用 continue 跳过循环	102
4.8 * 函数	104
4.8.1 定义函数	104
4.8.2 函数调用	106
4.9 * 数组	108
4.9.1 定义数组	108
4.9.2 数组调用	109
4.9.3 * 关联数组	112

4.1 第一个脚本

本节作者: Jiaqi Z.

在本节, 你将要学到:

- 什么是 Shell 脚本
- 如何编写第一个 Shell 脚本——Hello World
- 如何运行脚本

在前面的学习中, 我们已经了解如何使用 Linux 命令在 Shell 进行操作, 我们了解了如何对文件和目录进行简单的操作 (如删除、复制等), 同时我们也了解了一些更复杂的操作, 例如使用 `grep` 和 `sed` 进行文本的查找和替换等。最后, 我们还了解了如何使用 `for` 循环来进行批量操作。

而在一些特殊的场景下，我们可能希望做更复杂的操作，或者说，我们希望更简单地执行一些操作（这两句话本质上是一样的）。例如，如果我们每次都需要使用 `sed` 命令修改特定的内容，如 `ENCUT = 400`，刚开始还好，时间长了可能会“嫌麻烦”。此时可能会希望有一个单独的命令（比如叫做 `changeENCUT`）来实现这一功能。而实现这一功能的方法，就是使用脚本。

在本章，我们将会讨论如何编写自己的脚本。类似于编程语言，脚本里面将会包含大量的编程思想——如输入、输出、条件、判断、函数等。在学习这一部分之前，希望你已经有了部分编程语言基础（没有也没有关系）。

补充：关于 *Shell* 和 *Bash* 的区别：一般来说，*Shell* 指的是系统和用户交互的那层“外壳”，之前我们所学习的内容，其操作都是在 *Shell* 当中进行的。*Shell* 具有多种版本，如“*Bourne Shell*”、“*Bourne Again Shell*”、“*C Shell*”等。其中，“*Bourne Again Shell*”就是我们所谓的“*bash*”。在你的操作系统下，可以使用 `top` 命令查看其 *Shell* 类型。

Shell 脚本，全称叫做“*Shell Script*”，是一种在 *Shell* 当中批量运行多条语句的程序。

由于目前主流的 *Shell* 是基于 *Bash* 解释的，而我们所写的 *Shell* 脚本，实际上也大多都是 *Bash* 脚本，因此在后文当中，我们可能不会精确区分 *Bash* 脚本和 *Shell* 脚本的区别。

4.1.1 编写第一个脚本

正如任何程序的开始都是“Hello World”，在本章我们也不例外。在 Linux 当中编写 *Shell* 脚本不需要额外的程序，只需要使用 `vi` 编写一段文本文件，并赋予它运行权限，就可以作为脚本运行了。首先通过 `vi` 创建一个名为 `hello` 的文件，并输入如下内容：

Listing 4.1: hello

```
1  #!/bin/bash
2  # 输出Hello World!
3  echo "Hello_World!"
```

编写完成后保存，并添加运行权限（`chmod +x hello`，详见第 1.6 一节），然后执行 `./hello`，即可在屏幕上看到输出结果。

注意：在运行时需要加上 `./` 表示在当前目录寻找命令。在 *Linux* 当中，不添加 `./` 表示在环境变量 *PATH* 下查找文件运行，你可以在家目录下找到 `.bashrc` 的文件，里面包含有一系列配置 *Bash* 的命令，其中就有对环境变量的设置。

在运行前，你需要保证程序已经具有运行权限，或者可以使用 `source ./hello` 或 `./hello` 的方式（二者等价）“临时赋予运行权限”¹。

其中，代码第一行 `#!/bin/bash` 表示使用 *bash* 运行。正如前面所说的那样，Shell 具有多种版本，因此，在编写时应当特别指定你所使用的版本。由于目前大多数 Shell 都是使用 *Bash*，因此这一行在有些时候“可以省略”。但我们不建议将其省略，因为你永远不能保证你的这个脚本今后会在哪个版本的 Shell 下运行。

补充：你可以想见，`/bin/bash` 就是 *bash* 命令所在位置，你可以去看一下是不是真的存在。在查看的时候，注意是从“根目录”开始而不是“家目录”

如果你真的这么做了，一种简单的方法是在 `/bin/` 目录下使用 `ls | grep bash` 只输出具有“*bash*”的文件，从而简化输出结果。当然，你也可以直接使用 `ls bash` 查看。

当然，不建议你尝试使用 `vi bash` 查看里面的内容，它不是文本文件。

代码第二行以 `#` 开头表示注释。如同编写其他代码一样，使用注释是一个好习惯，它可以帮助你划分代码段落，以及记住对应的功能。随着学习的深入，我们会编写越来越长的脚本。因此，记得加注释是个好习惯。

第三行是这一脚本的关键，它使用 `echo` 实现字符串的输出。事实上，Shell 脚本的每一个命令都可以在 Shell 本身下运行。因此，你也可以直接在 Shell 运行这一命令，会实现同样的效果。而编写脚本之后，就可以直接通过 `./hello` 实现这一功能，这便是脚本的作用。

注意：`echo` 会将它后面的所有内容输出（在其他一些编程教材中，会将这一功能叫做“应声虫”，实际上，`echo` 也就是“回音”的意思）。我们在这里添加双引号是为了强调它们是整体的，事实上，当你去掉这两个双引号，对程序运行结果没有任何影响。

如果你在脚本中编写了代码，并同样使用了双引号，请注意：使用

¹这是表面上的用法，事实上，`source` 本意是在当前 Shell 下运行文件。相对的，其他的用法（不加 `source`）则是在当前 shell 下新建了一个“子 Shell”运行代码，其运行结果（例如一些变量）并不会带回外面。

英文符号而不是中文符号，这一点在后续所有脚本编写过程中都应当注意。一般来说，我们不建议在脚本当中添加中文，虽然你写 `echo "你好，世界！"` 可能也会得到正确的结果，但不会永远如此。

在本章的教程中，为了考虑到读者水平，我们的注释部分都会采用中文，如果这样也会引起脚本运行的失败（在测试时正常，但不敢保证在你的电脑也会正常），请删除中文注释后运行。

4.1.2 添加至环境变量

正如前面所说，使用 `./hello` 表示在当前目录下查找名为 `hello` 的脚本并运行。这可以帮助我们快速调试代码，但在真正应用时，我们可能会希望在任何目录下运行脚本。此时就会希望将代码添加至环境变量，也就是前面所说的 `PATH`。添加方法有两种——将脚本放置到已有的环境变量中，或者将脚本所在的目录设置为环境变量。

你可以通过 `$PATH` 命令输出当前环境变量，通常来说，你可以将你所编写的脚本命令放置在 `~/bin/` 目录下（这一般都是用户的环境变量）完成后，你可以在任何目录下运行你的脚本了（不需要 `./` 了）。例如，在完成上述配置后，在任何目录下运行只需要使用 `hello` 即可。

如果你编写了一系列脚本，一个简单的方法是直接将它们所在的目录设置成“环境变量”。此时需要通过 `.bashrc` 文件。假设你的脚本所在目录为 `~/bash/`，使用 `vi` 打开 `.bashrc` 文件，在最后一行添加 `export PATH=$HOME/bash:$PATH` 即可。

其中，`export` 是用来设置“环境变量”的命令，后面的 `PATH=...` 则是“变量赋值”的过程（后面就会学到）。`$HOME` 是系统内置的变量，表示用户的家目录，你可以在 Shell 下使用 `echo $HOME` 查看变量的值²，后面的 `$PATH` 则表示原先的环境变量。

简单说，这一语句的意思就是在原有的 `PATH` 变量前面添加一个新的 `$HOME/bash`。添加完成后你需要使用 `source ~/.bashrc` 命令“激活”这一环境变量（或者重启也可以实现这一功能），然后即可在任何地方如一般运行命令一样运行你在 `~/bash/` 目录下所有的脚本了。

注意：在后面的教学演示中，我们都不会添加 `./` 运行脚本（或者说，只有在这一节我们会详细提到如何运行脚本，后面都简单说作“运行脚本”）。

²在这里我们又不知不觉接触到 `echo` 的新用法：输出变量的值。这本是后面的内容，在这里你可以提前先了解一下。

如无特殊说明，无论是哪种方法（在当前目录、添加环境变量），运行最终效果都是一样的，后面不再赘述。

为了你的方便，建议新建一个目录作为你后续练习脚本的目录，并使用上面的方法将其添加到环境变量中。

4.1.3 错误处理

-bash: < 脚本名 >: Permission denied

这可能是因为你在运行脚本时没有赋予其运行权限而直接运行脚本，如果你没有赋予权限，请使用 `source` 命令或 `.` 运行脚本。这同样适用于环境变量中的命令。对于环境变量中没有运行权限的脚本，需要使用 `source < 脚本名 >` 或 `. < 脚本名 >` 执行。

4.2 变量

本节作者：Jiaqi Z.

在本节，你将要学到：

- 如何在脚本中定义变量
- 如何输出变量
- 如何对变量进行简单运算

如果所有脚本都只能按照固定的内容运行，显然功能太弱了。与其他编程语言类似，脚本语言应当也具有类似于“变量”的功能实现“可拓展性”。

所谓“变量”，指的就是在运行过程中会发生变化的量，这些值可能是由用户输入给定的，或者在运行过程中生成的，或者是通过文件读取得到的。

4.2.1 定义变量与初始化

与 C 语言等强类型语言不同，Shell 脚本的变量在使用之前不需要对其进行“声明”，相对地则是需要对其进行初始化。与其他编程语言类似，在 Shell 脚本中，第一次使用变量时需要对变量进行赋值（也可以叫做“初始

化”)。例如，我们希望将字符串“Hello World!”赋值给一个变量，则可以使用 `STRING="Hello World!"`

注意：与其他编程语言类似，在 *Shell* 脚本中，赋值也是使用 `=` 运算符。但不同的一点是，在运算符两侧不能有空格。

在变量命名时，需要遵守如下原则：变量名只能包括数字、字母和下划线 (`_`)，第一个字符不能是数字，不能是已有的关键字³。

在一个程序中，可以同时存在多个变量，对于已经赋值的变量，也可以对其再次进行赋值（原有值会发生变化）。例如，下面的代码：

Listing 4.2: variable

```
1  #!/bin/bash
2  # 变量初始化
3  STRING1="Hello World!"
4  STRING2="I Like Bash"
5  STRING2="I Like Shell Script"
```

上述代码第 3 行定义了一个变量 `STRING1`，其赋值为“Hello World!”；在第 4 行，首先定义了一个变量叫做 `STRING2`，首先赋值为“I Like Bash”，在第 5 行又一次对其进行赋值，此时 `STRING2` 的值变为“I Like Shell Script”。

4.2.2 调用变量

如果你熟悉其他编程语言如 C、Java、Python 等，也许在编写上面的语句时，你会很自然写出如 `STRING2=STRING1` 这样的语句。在你看来，这好像是把 `STRING1` 的值赋值给 `STRING2`，但当你运行时，发现事实并非如此。这是因为在 *Shell* 脚本中，变量的调用需要用到其他的方法。

注意：在上面我们提到 `STRING2=STRING1` 的含义是将 `STRING1` 的值赋值给 `STRING2`，这对于了解过其他编程语言的读者而言是很自然的。但如果你没有学习过其他编程语言，需要特别注意的一点是：在程序设计语言中（几乎大多数的程序语言），`=` 所表示的含义与你所熟悉的数学上的含义

³ “关键字”指的是在 *Shell* 脚本中已经具有特定含义的词语，如 `echo` 就不能作为变量名。

不同。在数学上， $=$ 表示一种状态，表达两个值相等；而在程序设计中， $=$ 表示将右边的值赋值给左边的值（一种动作）。

尽管在最后的结果上，二者是相同的，但数学上的 $=$ 表达一种“状态”，而程序设计中表达一种“动作”，是不同的。一个很简单的例子就是上面的 $STRING2=STRING1$ ，从数学的角度看，这显然不成立，因为“I Like Shell Script”显然不可能等于“Hello World!”，但程序设计上可行的，因为它表达了“赋值”的动作。

也正因如此，在数学上， a 和 b 相等写成 $a=b$ 或者 $b=a$ 都是可行的（这也就是等式的“对称性”）；而对于程序设计而言， $a=b$ 和 $b=a$ 显然是不同的，因为它们所表达的动作“方向”是不同的。

在 Shell 脚本中，调用变量需要使用到 $\$$ 符号。事实上，这不是你第一次见到它（如果你忘记了，请回到 3.5 一节，或者更准确的，3.5.2 一节）。与定义变量不同，在 Shell 脚本中，但凡是需要调用变量的地方，都需要使用 $\$$ 符号。例如，在上面的例子中，如果你确实希望表达 $STRING2=STRING1$ ，需要写作 $STRING2=\$STRING1$ 。

这里有一点“小绕”的地方在于，为什么在 $STRING2$ 前面不需要添加 $\$$ 符号。这是因为，我们实际上只是调用了 $STRING1$ 变量的值，并不关心 $STRING2$ 里面是“I Like Shell Script”还是“I Like Roselia”。因此，我们只需要通过 $\$STRING1$ 来获得 $STRING1$ 的值。

补充：你还可以做一个“不准确”的理解： $\$$ 总是视图将右边的内容“展开”为完整的内容。例如，假设 $STRING2$ 的值为“Hello World!”，那么在调用 $STRING2=\$STRING1$ 时，可以写作 $STRING2="Hello World!"$ （将变量 $STRING1$ 展开）

对于前面所介绍的 for 循环，其本质是类似的。例如， $for i in \{1..5\}; do echo \$i; done$ ，实际上也是将变量 i 展开为具体的 1 到 5。

当你了解这个时，对于后面的一些操作，会很有帮助。

在了解了如何调用变量后，我们就可以做一些完整的事情了。例如，下面的一段完整代码实现了变量的初始化，修改赋值和调用，并在最后使用 $echo$ 语句进行输出。

Listing 4.3: variable

```
1  #! /bin/bash
2  # 变量初始化
```

```
3 STRING1="Hello_World!"
4 STRING2="I_Like_Bash"
5 # 修改变量的值
6 STRING2="I_Like_Shell_Script"
7 # echo输出变量的值（调用变量）
8 echo $STRING1
9 echo $STRING2
```

运行上述代码，就可以看到输出了“Hello World!”和“I Like Shell Script”。

4.2.3 字符串中的 \$ 符号

像上面这样在字符串中使用 \$ 符号，最简单的情况就是上面这种单纯输出一个变量。但大多数时候，我们可能希望在输出时提供更复杂的内容。例如，我们有下面的变量

Listing 4.4: dollar_in_string

```
#!/bin/bash
# 初始化变量
name="Jiaqi_Z."
band="Roselia"
```

如果我希望输出“My name is Jiaqi Z., and I like Roselia”。如果考虑到 echo 可以连续输出多个参数，也许你会想写出 echo "My name is" \$name", and I like" \$band 这样的语句。确实，从运行的角度，这个句子没有任何问题。但显然从可读性的角度，稍显复杂。那么，有没有更简单，更清晰的方式呢？

在使用 \$ 表示变量时，我们可以将变量名使用大括号将其括起来。例如，上面的例子，我们就可以写作 echo "My name is \${name}, and I like \${band}"

补充：如果你尝试将大括号去掉，在这一例子中，同样可以运行出正确的结果。这是因为，每一个变量名后面都跟着一个“标点符号”。如果对于

再一般的情况，我们的变量名后面跟着另外一些字母。例如，如果我们在 `$name` 后面再加个 `s`，对于使用大括号的情况，则会输出 `Jiaqi Z.s`，而对于没有大括号的情况，则会输出错误的结果（由于没有变量叫做 `names`）。

因此，为了更一般的情况，我们建议在使用变量名调用变量时加上大括号。

4.2.4 变量简单运算

在前面的例子中，我们都是针对于字符串变量进行讨论。事实上，变量还可以保存一些数值信息（例如整数、小数等）。例如，我们可以定义变量 `a=3` 和 `b=2.5`。

同时，在脚本中，我们也可以进行简单的四则运算。简单的方式则是利用 `$((表达式))` 的格式写出运算内容。例如，下面的代码则是简单计算 `1+1` 的结果：

Listing 4.5: calculate

```
1  #!/bin/bash
2  # 定义变量
3  a=1
4  b=1
5  # 计算
6  c=$(( {a} + {b} ))
7  # 输出
8  echo {c}
```

在 Shell 脚本中，四则运算（加减乘除）的符号分别为 `+`, `-`, `*`, `/`，同时需要特别注意的两个符号是 `%` 表示取余，即求得两个整数相除后的余数，例如，计算 `echo $((9%4))` 可以得到 1；`**`（两个乘号）表示幂运算，例如，`echo $((2**10))` 表示 2^{10} ，即 1024。

补充：在 Shell 脚本中，你可以使用这种方法进行整数的四则运算。对于小数而言，则需要使用更复杂的方法。例如，对于变量 `a=1.5` 和 `b=2`，如果希望做小数的四则运算，可以使用 `bc` 命令，写作 `echo "${a} + {b}" | bc`。

但对于脚本来说，通常你不应寄希望于它的运算功能（这主要由于运算效率的限制）。通常来说，对于需要使用复杂运算的任务，应当考虑其他效率更高的编程语言如 *C* 语言⁴和 *Python* 语言。

例如，上述问题如果希望使用 *C* 语言编写，则可以写作下面的代码：

Listing 4.6: calculate.c

```
1  #include "stdio.h"
2  int main()
3  {
4      double a = 1.5;
5      int b = 2;
6      printf("%f\n", a+b);
7      return 0;
8  }
```

并通过 `gcc calculate.c` 的方式编译代码，得到 `a.out` 可执行文件。通过 `./a.out` 即可运行得到正确结果。

正因如此，在本教程中，我们不会详细讨论脚本的计算（如果你确实需要使用脚本进行复杂运算，请查阅其他相关资料（例如 *bc* 和 *awk* 的相关使用方法）

4.2.5 错误处理

-bash: < 表达式 > : syntax error: invalid arithmetic operator (error token is "< 表达式 >")

这可能是由于你错误使用了四则运算符，例如，在使用 `$(())` 的方式进行计算时，要求只能进行整数四则运算。如果你的运算符两边出现了小数，则会出现错误。

bash: < 变量名 >: command not found...

这是因为你在对变量进行赋值时，在 `=` 两边加了空格。在 *Shell* 脚本中，赋值符号 (`=`) 两边不能有空格，这一点与其他编程语言不同。

⁴在大多数 *Linux* 当中都有 *C* 语言的编译器 `gcc`，你可以使用 `gcc --version` 查看对应版本。

4.3 输入

本节作者：Jiaqi Z.

在本节，你将要学到：

- 如何读取用户输入
- 如何读取命令参数
- 如何将命令执行结果赋值给变量

在前面，我们仅仅讨论了脚本内定义的变量，这对于脚本而言远远不够。在实际使用脚本的时候，有一些信息只有在调用时才能知道。例如，如果我们希望编写一个可以删除文件的脚本⁵，在编写时不可能知道需要删除哪些文件。因此，有必要在写脚本时考虑实现“交互”。

通常来说，交互的方式有三种：程序运行时输入、程序调用参数、以及外部文件。在本节，我们将讨论这三种交互方式如何在脚本中实现。

注意：严格来说，还有一种：管道输入。在本节我们不详细讨论管道的输入方式。

4.3.1 用户输入

在 Shell 脚本中，实现用户输入的方法是使用 `read` 语句。一般格式是 `read < 变量名 >`，例如，`read a` 表示在运行时读取用户输入，并将输入结果赋值给变量 `a`。

在调用 `read` 命令时，可以提供一個选项，`-p` 表示在屏幕上显示提示信息，其格式为 `read -p < 提示信息 > < 变量名 >`。

利用这一命令，我们可以实现简单的交互。例如，我们可以写一个简单的“应声虫”小程序，即当用户输入一个内容后，程序原封不动将其输出。

Listing 4.7: my_echo

```
1  #!/bin/bash
2  # 读取输入
```

⁵尽管我们已经有了 `rm` 命令，但我们可能还会有其他想法。例如，我们希望在删除完成后输出删除了哪些文件，或者我们希望将“删除”改为移动至某一个目录实现“回收站”的功能。


```
3 read -p "Please input a string:" STRING
4 # 输出
5 echo "You said: ${STRING}."
6 echo "Good Luck!"
```

其中,第3行我们使用 `read` 命令读取用户输入,并将其赋值给 `STRING`。之后在第5行使用 `echo` 语句输出了用户的内容(在前面加了一些其他内容)。

在运行时,程序会首先输出 `Please input a string:` 并等待用户输入。当用户输入完毕后,按下回车表示完成,此时程序执行后面的内容(输出)。

注意: 在读取输入时,不要在变量名前面“画蛇添足”加上一个 `$` 符号。如果你试着这样做,会得到错误的结果——它有可能会输出空白信息,或者输出一些其他的内容。

补充: 输出空白或者其他信息取决于你的运行方式是使用 `source` 还是添加执行权限。对于前者, `source` 本质上相当于在当前 `Shell` 终端下执行了脚本里的命令,其变量会延续到脚本外。因此,如果你在刚开始正确时输入了一个内容,如 `"Hello World!"`,脚本会将其赋值给 `STRING` 变量并延续到 `Shell` 外部(用更专业的说法,这种“延续”实际上是“作用域”的体现)。此时如果你尝试在外面直接运行 `echo $STRING`,也会得到对应的结果。

如果你是添加了执行权限并运行的话,脚本实际上是在当前 `Shell` 下新建了一个“子 `Shell`”并运行,运行过程中产生的变量仅会在这一 `Shell` 内有效(表现为程序内),当退出脚本时,变量也就因此失效了。

在使用 `read` 输入变量时,如果后面加了 `$` 符号,则不会输入任何内容。此时在 `echo $STRING` 时,则会根据目前环境下已有的变量,输出对应的结果(已有的内容或空白)

4.3.2 参数输入

除了使用前面所介绍的 `read` 方法在程序运行时读取用户输入,在有些时候可能也希望通过类似于参数调用的方式输入。可以类比一下最开始我们所接触到的如 `cd` 和 `rm`,在切换目录或者删除文件时,都是在调用时直接给出对应的文件名,而不是在运行过程当中输入。那我们有没有类似的方

法实现这一功能？

答案肯定是有，而且这一部分你不需要任何特殊的命令。因为在脚本中，如果调用时提供了一个参数，默认在程序中就是 `$1`，以此类推，如果有两个或多个参数，则分别是 `$2`，`$3`，`$4` 等等。例如，下面的代码则实现了位置参数的调用：

Listing 4.8: loc_parameters

```
1  #!/bin/bash
2  # 输出第一个参数
3  echo "First_parameter_is_$1"
4  # 输出第二个参数
5  echo "Second_parameter_is_$2"
```

当调用时，类似于之前使用其他内置命令那样，可以往其中传递参数（使用空格分割），如 `./loc_parameters hello world`，则第一个参数为“hello”，第二个参数为“world”。

注意：通常来说，我们将这种形式上如 `$n` 的参数叫做**位置参数**。也请务必注意的一点是：位置参数默认是从 1 开始而不是从 0 开始的。当参数数量在 9 个及以内时，可以直接使用 `$1` 到 `$9` 这种形式，但如果到了 10 个及以上参数，需要在数字外面加大括号如 `$10`。

你也可以尝试在程序中使用 `$0`，它表示**正在运行的脚本名称**。

补充：如果你真的尝试输出 `$0` 的值，可能会意外地输出一个叫做 `-bash` 的内容而不是脚本名称。这是因为如果你使用的是 `source` 方式运行，在这种情况下，你的脚本实际上是在当前命令行环境下运行，此时程序中的 `$0` 与你直接在命令行中输入 `$0` 运行结果应当是一致的。

另外，如果你的程序是在其他目录下运行（假设你已经将这一目录添加进环境变量），此时 `$0` 会输出这一脚本所在的完整目录。

除了直接使用位置编号表示参数本身外，Bash 脚本还提供其他内置的参数表示其他信息。常见的例如 `$#` 表示传递的参数个数（不包括 `$0`），`$@` 表示整个参数列表。下面的程序使用 `for` 循环遍历了所有参数（完整的 `for` 循环教程在后面介绍）

Listing 4.9: special_parameters

```
1  #!/bin/bash
2  # 输出一些特殊参数
3  echo "Current_file_is_${0}"
4  echo "We_have_${#_parameters}"
5  echo "They_are:"
6  for i in $@
7  do
8      echo $i
9  done
```

其中，第 6 行使用 `$` 符号表示传入的参数列表，并对其中的所有元素进行遍历（输出）

4.3.3 读取命令作为输入

除了在运行时输入，在很多时候我们需要借助于一些命令读取文件的内容，并将其作为变量进行处理。例如，我们希望读取 `INCAR` 文件中的 `ENCUT` 所在的一行，根据前面所学习的方法，我们可以使用 `grep` 命令，如 `grep ENCUT INCAR` 命令来输出这一行。如果我们希望将这一命令作为变量输入到脚本中，只需要使用 `$(grep ENCUT INCAR)` 这种形式即可，其命令使用小括号，且前面加上变量的 `$` 符号。同样，在命令里面也可以使用变量以实现更复杂的交互功能，下面的代码实现了用户输入一个字符串和文件名，查找文件中包含这一字符串所在一行的内容：

Listing 4.10: `print_string`

```
1  #!/bin/bash
2  # 读取字符串
3  read -p "Please_input_a_string:_" STRING
4  # 读取文件名
5  read -p "Please_input_a_file_name:_" FILE
6  # 读取命令并赋值
7  result=$(grep ${STRING} ${FILE})
8  echo ${result}
```

注意：上述代码并不是一个“完美”的代码，因为在读取文件时，并没有对文件是否存在进行检查。因此，如果输入了一个不存在的文件名，则会输出错误的结果（如同你正常使用 `grep` 时输入了错误的文件名那样报错）。当你测试这一段代码时，请提前创建好一个对应的文件。

在后面的学习中，我们将进一步完善这一代码（设置一段代码实现文件是否存在的检查）

4.4 判断语句

本节作者：Jiaqi Z.

在本节，你将要学到：

- 如何在脚本中使用 `if` 及相关语句
- 如何对数值、字符串和文件进行判断
- 如何进行逻辑判断（与、或、非）

在之前的脚本中，我们只能按照顺序进行执行，从某种程度上来看，这并不“智能”。通常来说，一个好的脚本应该会根据实际情况来决定执行的内容，比如，当用户输入了一个文件名后，如果这个文件并不存在，程序应当做出相应的反馈。这一节，我们就会稍微了解以下如何进行判断，并且让程序根据情况做事。

4.4.1 `if` 语句

在开始这一切的学习之前，让我们先来了解一个最基本的判断语句框架，以便在后面更好地讨论深入的话题。类似于 C 语言和 Python 语言等，在 Shell 脚本中，最简单的判断语句（`if` 语句）框架如下：

```
commands1
if [ condition ]; then
    commands2
```

```
fi
commands3
```

这一段代码当执行完 `commands1` 之后，进行 `condition` 判断（条件是否成立），如果（`if`）成立，然后（`then`）执行 `commands2` 语句，**否则不执行任何语句**，最后执行 `commands3`。

注意：无论条件是否满足，`commands3` 都会执行（它是 `if` 语句之外的内容，类似于 `commands1`）。在这里的 `commands1, commands2, commands3` 都代表语句（们），每一部分可以是一条或多条语句。`condition` 是**判断条件**，在后面的部分我们将详细介绍如何描述这一部分，简单来说，它描述的内容就是“是否……?”。

需要特别注意 `condition` 前后与中括号之间的空格，这个空格不能省略！千万不要写成 `[condition]` 这种形式。

补充：与 *C* 语言使用大括号，*Python* 语言使用缩进不同，*Shell* 脚本采用类似于 *Basic* 语言和 *MATLAB* 那样使用语句表示一整个语句块的方式。一个判断语句一定是以 `if` 开始，以 `fi` 结束。这个 `fi` 不代表“*finish*”或者“*final if*”等类似含义，而是 `if` 的倒序（在后面其他内容的学习中，会逐渐印证这一点）。

换言之，在上述代码中，使用缩进与否并不会影响脚本执行效果，但为了增强可读性，我们仍然建议使用缩进表示每一级之间的层级关系。事实上，如果你使用 `vi` 创建 `*.sh` 文件时，在输入 `if` 后默认会进行缩进，这也是大多数现代程序语言文本编辑器应当具有的功能。

4.4.2 关系运算符

任何一个分支判断语句，都应当首先给定一个关系运算，并根据这个结果来判断应该执行哪些命令。在 *Shell* 脚本中，我们大致可以将关系运算符分成三类：

整数比较

类似于数学上的大于、小于和等于，在 *Linux* 当中也有对数值的比较，包括等于（`-eq, ==`）、不等于（`-ne, !=`）、大于（`-gt, >`）、小于（`-lt, <`）、大于等于（`-ge`）和小于等于（`-le`）六种。

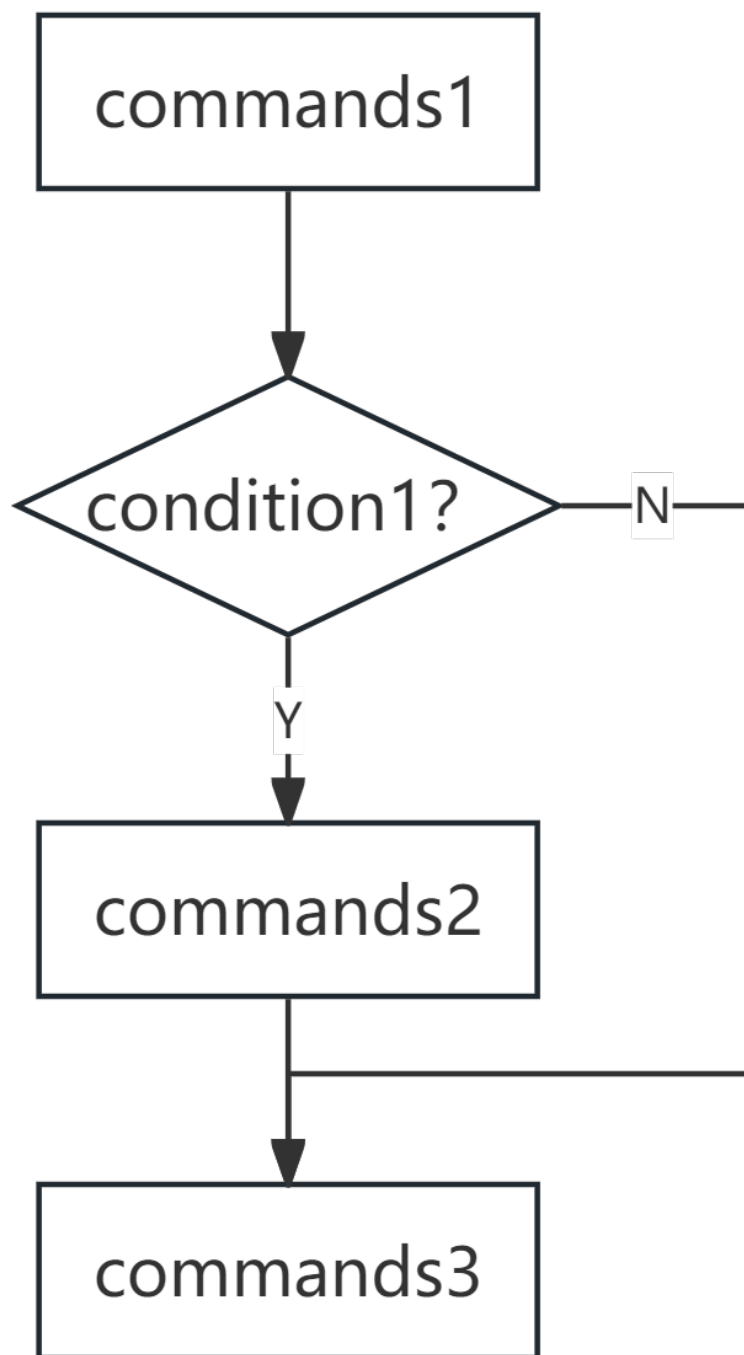


图 4.1: if 语句流程图

注意：在 *Shell* 脚本中，许多运算符都是通过上面这种“选项”的形式给出。与一般输入命令类似，选项前后也需要有空格进行分割。

除了选项格式外，前四种（等于、不等于、大于和小于）我们也给出了类似于 *C* 和 *Python* 等其他编程语言所使用的运算符格式。这些在 *Shell* 脚本中同样可用。但是，对于大于等于和小于等于，没有 `>=` 和 `<=`。如果你使用了这两个符号，大概率会报错。关于这一问题，目前还没有找到相关的解决方法，如果你有了对应解决方法（当然，不是后面要讲的内容），请联系我。

补充：这些选项实际上是英文单词的缩写，例如，`eq=equal`; `ne=not equal`; `gt=greater than`; `lt=less than`; `ge=Greater or Equal`; `le=Less or equal`

在后面你还会见到其他类似的语句，了解它们的实际含义可以帮助你记忆这些选项。

下面的程序可以用来判断两个数字是否相等（使用前面的 `if` 语句）

Listing 4.11: `if_equal`

```
1  #!/bin/bash
2  # 判断两个数字是否相等
3  a=10
4  b=10
5  # if语句判断是否相等
6  if [ $a -eq $b ]; then
7      echo "$a_is_equal_to_$b"
8  fi
9  # 判断结束后输出
10 echo "Bye!"
```

你可以试着修改变量的值，查看输出结果是否有不同。在上述代码中，当变量相等时，判断结果为“1”，从而执行里面的语句；如果不相等，则判断结果为“0”，跳过里面的语句。无论结果如何，你都会看见第 10 行所输出的“Bye!”（它在 `if` 语句外面）。

补充：在这里我们提到了“1”和“0”，它们实际上叫做“布尔值”或者“逻辑值”，也是关系运算（与后面要提到的逻辑运算）的返回结果，其

值只包含两种：“真”（也可以用“*True*”、“*1*”等代替）和“假”（也可以用“*False*”、“*0*”等代替）。

注意：在中括号里面表示条件判断时，请务必记得中括号内前后要加空格，同时`-eq`前后也要加空格。

在完成 `if` 语句后，不要忘记后面的 `fi`。

你也可以试着修改判断条件，例如改成`-gt`，并修改变量的值，查看结果。

* 浮点数比较

补充：由于前面所介绍的浮点数在脚本中的局限性，这一部分内容关于浮点数的比较并非必须了解。但如果你确实有此方面需求，在确定不能使用其他如 *C* 和 *Python* 等编程语言实现的前提下，可以参考这一部分所介绍的方法。

相比于前面的整数比较，浮点数比较不能使用前面的“选项”格式。例如，你写出 `if [3.5 -ne 2.5]` 是错误的。但是，前面所使用的运算符形式如 `==`、`!=` 等还是可用的。基于此，对于判断等于和不等（包括大于和小于），最简单的方法就是使用如 `if [3.5 != 2.5]` 的格式。

但是，对于大于等于和小于等于这两种情况，整数部分尚且还有选项可用，浮点数则完全没有对应的简单方法。参考前面 4.2.4 一节所介绍的 `bc` 命令，我们可以退而求其次，借助于逻辑运算符的输出结果 *1* 和 *0*，来进行判断。例如，我们希望判断 *3.5* 是否大于等于 *2.5*，则可以使用 `if [$(echo "3.5 >= 2.5" | bc) -eq 1]` 这种方式，其中小括号部分借助管道运算符和 `bc` 命令计算 `3.5 >= 2.5` 的结果，根据前面所介绍的逻辑值，输出结果应当是 *0* 或 *1*（在这里为 *1*）。然后利用整数的判断方法，判断它与 *0* 或 *1* 的关系，从而实现浮点数对大于等于和小于等于的判断。

字符串判断

与数值判断类似，字符串也可以进行相应的判断，一般常见的包括判断两个字符串是否相等（`==`），是否不相等（`!=`），以及判断一个字符串是否为空字符串（`-z` 和 `-n`）

注意：在判断是否为空字符串时，可以使用`-z`和`-n`，二者在本质上判断的内容是一样的，但返回结果相反，前者当内容为空时返回“真”，后者当

内容不为空时返回“真”。借助于后面的“求非”运算，这二者只需要有一个即可，但为了简洁易读，还是建议在对应的时候使用正确的关系运算符。

此外，与前面数值判断不同，判断字符串是否为空是“一元运算符”，即只需要一个变量。后面的代码则给出了这种一元运算符的一般格式。

下面的代码实现了判断一个字符串是否为空字符串：

Listing 4.12: string_empty

```
1  #!/bin/bash
2  # 判断字符串是否为空
3  read -p "Please_input_a_string:" string
4  # -n当字符串不为空时为真
5  if [ -n "$string" ]; then
6      echo "Right!I_got_something..."
7      echo "You_input:$string"
8  fi
9  echo "Bye!"
```

上述代码第 5 行通过 `-n` 判断输入字符串是否不为空，如果有内容（结果为真），则执行 `if` 语句里面的内容。

注意：在对字符串进行判断时（包括使用二元关系运算符），请务必将字符串变量前后加上双引号。如果不加双引号可能会造成奇怪的错误。

文件判断

Shell 脚本也提供了大量的运算符选项，用来判断文件的相关信息。例如，使用 `-e` 判断文件是否存在，使用 `-f` (`-d`) 判断是否为普通文件（目录），使用 `-r`，`-w`，`-x` 依次判断文件是否可读、可写、可执行。

与前面字符串的代码类似，这里的所有运算符都是一元运算符（即采用 `< 选项 > 变量` 的格式。例如，下面的代码简单实现了判断文件 `INCAR` 是否存在的功能：

Listing 4.13: file_exist

```
1  #!/bin/bash
```

```
2 # 判断文件 INCAR 是否存在
3 if [ -e "INCAR" ]; then
4     echo "This file exists."
5 fi
6 echo "Bye!"
```

其中，`-e` 后面的 `"INCAR"` 是指当前运行目录下的 `INCAR` 文件，你也可以使用绝对路径来描述文件。

4.4.3 带有 `else` 的判断语句

下面，我们将进一步介绍 `if` 语句，在之前我们仅仅用来判断某一个条件是否满足，且当满足时执行某一（些）语句。但有时，我们可能有更复杂的需求。例如，判断某一个文件是否存在，如果存在则对文件进行处理，如果不存在则输出文件不存在，并提示用户重新输入。我们姑且忽略到继续输入这一动作（需要用到后面的循环），当条件不满足时如何执行另外的语句？类似于其他编程语言，在 Shell 当中也可以使用 `if-else` 语句。其基本格式如下：

```
commands1
if [ condition ]; then
    commands2
else
    commands3
fi
commands4
```

首先程序会执行 `commands1`，然后进行判断，如果（`if`）条件成立，则（`then`）执行 `commands2`，否则（`else`）执行 `commands3`。无论结果如何，最后执行 `commands4`。

下面的代码则是利用上面的语法结构，对前面的判断文件是否存在的脚本（`file_exist`）进行了修改：

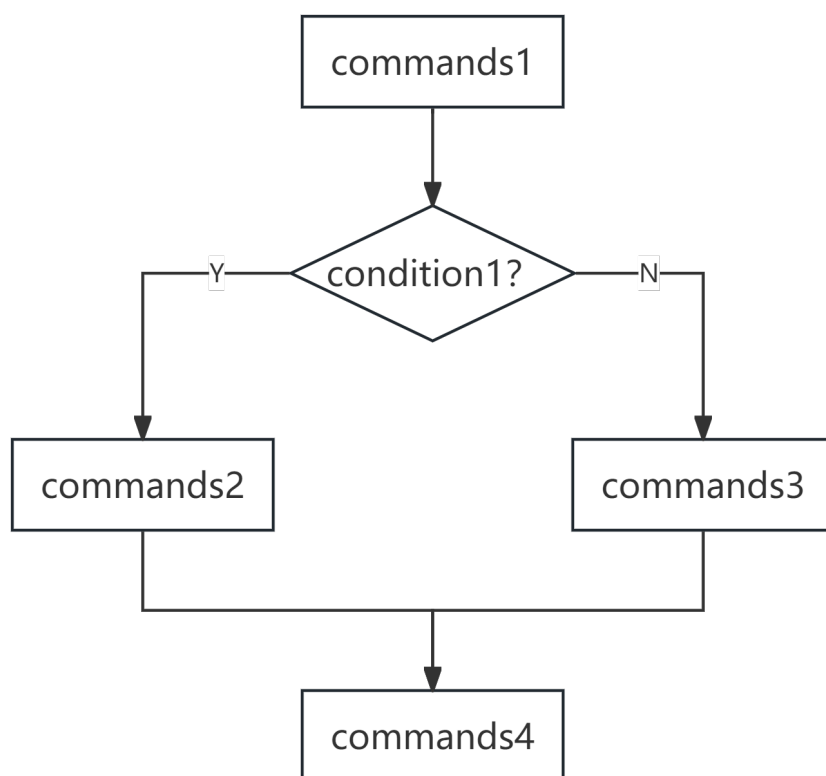


图 4.2: if-else 语句

Listing 4.14: file_exist(2)

```
1  #!/bin/bash
2  # 判断文件INCAR是否存在
3  if [ -e "INCAR" ]; then
4      echo "This_file_exists."
5  else
6      echo "This_file_NOT_exists."
7  fi
8  echo "Bye!"
```

除此之外，与 Python 语言类似，Shell 脚本也有 `if-elif-else` 语句，用来对多条件进行判断，语法如下：

```
commands1
if [ condition1 ]; then
    commands2
elif [ condition2 ]; then
    commands3
elif [ condition3 ]; then
    commands4
.....
else
    commands5
fi
commands6
```

程序首先判断 `condition1` 是否满足，如果（if）满足，则（then）执行 `commands2` 并结束判断语句，反之如果（else if, elif）满足 `condition2`，则（then）执行 `commands3` 并结束判断语句；反之如果……；否则都不满足（else），执行 `commands5`。在结束判断语句后，执行 `commands6`。

下面的程序可以用来判断两个整数之间的关系：

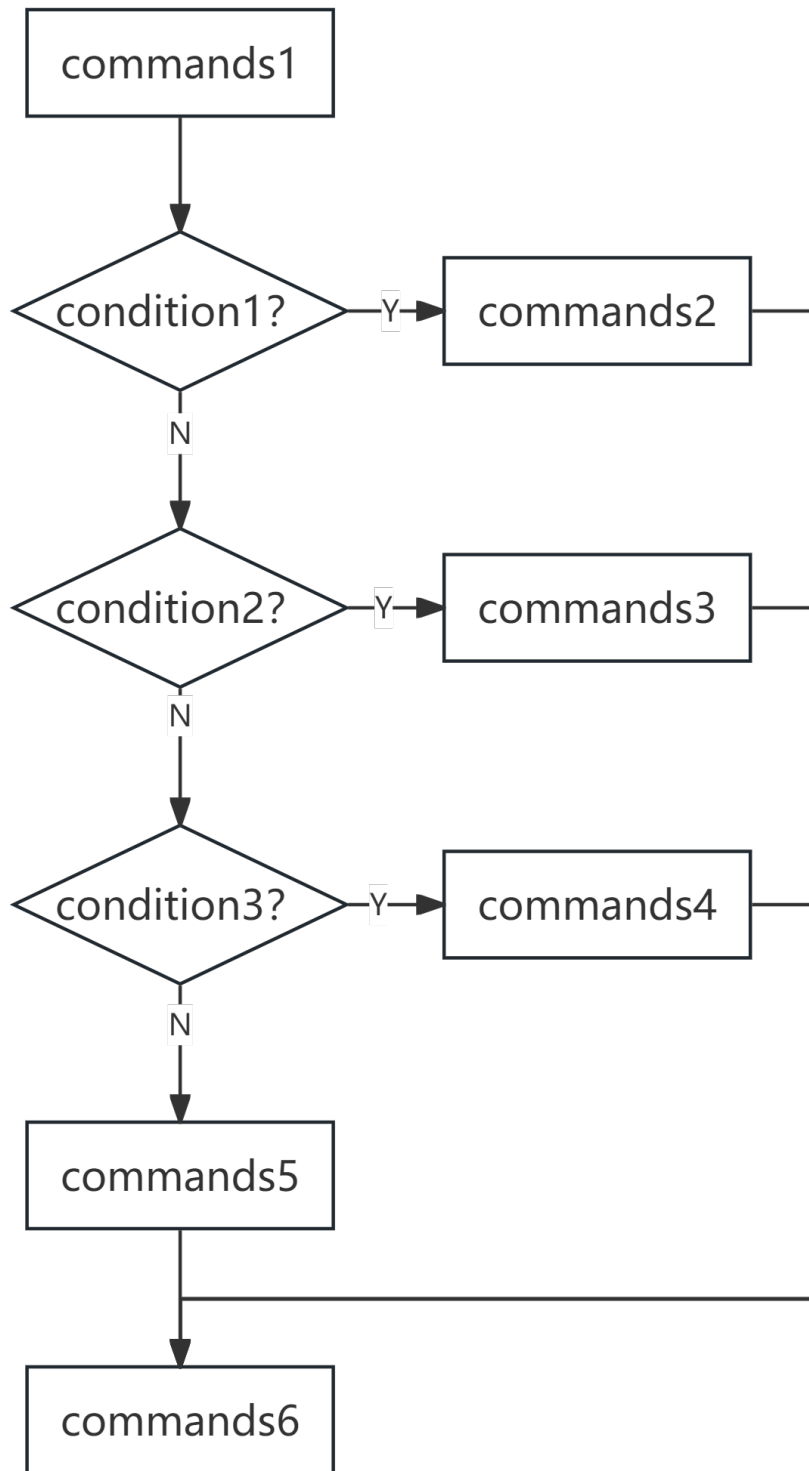


图 4.3: if-elif-else 语句

Listing 4.15: compare_number

```
1  #!/bin/bash
2  # 判断两个整数之间的关系
3  # 输入两个整数
4  read -p "Please input an integer(a):" a
5  read -p "Please input an integer(b):" b
6  # 判断两个整数之间的关系
7  if [ $a -eq $b ]; then
8      echo "$a=$b"
9  elif [ $a -gt $b ]; then
10     echo "$a>$b"
11 else
12     echo "$a<$b"
13 fi
14 echo "Bye!"
```

4.4.4 嵌套 if 语句

与其他编程语言类似，脚本也可以使用嵌套的 `if` 语句，甚至可以更复杂的 `if-elif-else` 嵌套，基于此可以实现复杂的功能。在这里我们不举例子，但有一个需要特别注意的地方：

注意：任何一个 `if` 语句，其后面都需要配合一个 `fi` 作为语句结束。尤其是对于嵌套时的 `if-else` 匹配问题，`else` 总是与最近的未完成的 `if` 语句匹配（与缩进无关）。例如，下面的代码就违反了第一个原则（`if` 必须配合有一个 `fi`）；同时，看似 `else` 是属于 `condition1` 所对应的判断，但事实上是属于里面的 `if`。

```
if [ condition1 ]; then
    commands1
    if [ condition2 ]; then
        commands2
    else
```

```
commands3
fi
```

这段代码是无法正常运行的。所缺少的 `fi` 可以加在 `commands2` 后面，也可以加在 `commands3` 后面，二者所实现的效果是不一样的。你可以尝试一下不同位置所对应的程序运行结果。

4.4.5 逻辑运算

在这一节的最后，让我们再来讨论一下逻辑运算。逻辑运算一共分为三种：与（`-a`）、或（`-o`）和非（`-n`）。其中与运算和或运算都是二元运算符，而非运算是一元运算符。

对于与运算而言，只有当两个值都为真时结果为真。而对于或运算，只要有一个为真，结果为真⁶。对于非运算，是一元运算符，其运算结果就是真变假，假变真。

下面一个例子实现了判断三个数字是否从小大大排序：

Listing 4.16: sort

```
1  #!/bin/bash
2  # 判断三个数字是否从小到大排序
3  a=5
4  b=8
5  c=10
6  # 使用与运算符连接
7  if [ $a -le $b -a $b -le $c ]; then
8      echo "$a, $b, $c"
9  fi
```

补充：在这里我们使用多个运算符进行连接，与其他编程语言类似，这里也具有“优先级”的问题。可以简单的理解：关系运算符的优先级高于逻辑运算符的优先级（这在大多数编程语言中都是如此）。虽然上面的写法比较简单，但对于条件复杂的时候可能不具有较好的可读性。因此，可以使用

⁶或者还有一种表述是，只有当两个值都为假时，结果为假。

类似于 C 语言的运算符 (`&&` 表示 “与”，`||` 表示 “或”)，而中间每一个条件使用中括号分割，写作 `if [$a -le $b] && [$b -le $c]`

4.4.6 错误处理

这一部分有可能出现的错误太多了，以至于难以在这里全部列出。这里仅列举一些常见的错误，且这些错误的解决方法仅是一部分可能的原因。当你出现错误时，请首先查看对应的格式是否正确，并且可以尝试联系我们。

-bash: < 文件名 >: line < 行号 >: syntax error: unexpected end of file

这是因为在使用 `if` 后没有对应的 `fi` 作为结束。通常出现在嵌套语句中或者一个 `if` 语句太长，到最后忘记了匹配。对此，我们建议，在开始就按照 `if-fi` 对应关系输入。

bash: [< 变量 >: command not found...

这可能是由于你在输入关系表达式时忘记了中括号里前后要加空格。

-bash: [: missing ‘]’

与前面的错误类似，这个通常指右中括号前面没有空格。

4.5 case分支语句

本节作者：Jiaqi Z.

在本节，你将要学到：

- 如何使用 `case` 语句实现分支判断

在前面的 `if` 语句中，已经了解了如何进行判断。理论上了解了 `if` 语句后可以解决所有分支情况，但是有一些情况可能比较特殊。例如，我们希望实现一个菜单界面，此时可能需要用户输入一些指令表示对应的功能。可

以想见，如果使用 `if` 语句，将会有许多的判断条件，甚至随着条件的增多，也会影响程序运行效率⁷。

因此，我们希望可以寻找一种方法，直接对变量进行判断，并根据它的值选择合适的语句执行。可以猜到，这就是本节 `case` 语句所要解决的问题。

4.5.1 case 语句基本结构

类似于 `if` 判断语句，`case` 判断语句的基本结构为：

```
case <变量> in
pattern1)
    statements1
    ;;
pattern2)
    statements2
    ;;
.....
*)
    statements3
    ;;
esac
```

程序会根据变量所在 (`in`) 的模式 (可以是一个单独的值，或者正则表达式)，选择合适的语句进行执行。例如，如果变量满足 `pattern1`，则会执行 `statements1` 语句；如果满足 `pattern2`，则会执行 `statements2` 语句；以此类推，如果所有的都不满足，则执行 `*` 所对应的 `statements3` 语句。

注意：有几个细节需要特别关注：所有匹配模式都是以小括号作为结束分割（没有左括号）；当每一个分支里面的语句结束时，需要有`;;`作为结束的标志；在 `case` 语句结束时，需要有 `esac` 作为结束标志。

补充：正如在 4.4 中所讲的那样，`esac` 也是 `case` 的倒序。另外，模式当中的 `*` 实际上可以想见，表示的就是通配符里面的可以表示任意多个字符的 `*` 符号。

⁷这是因为在进行判断时，脚本需要对所有条件进行遍历。

4.5.2 case 语句应用

下面让我们来看几个常见的应用例子。

单一字符匹配

在 case 语句中，最简单的就是对单独一个字符进行匹配。下面这个例子则实现了一个对菜单的模拟，用户通过输入 1 或者 2 来实现对应不同的功能。

Listing 4.17: simple_menu

```
1  #!/bin/bash
2  # 模拟菜单选项
3  echo "-----"
4  echo "1)_option_1"
5  echo "2)_option_2"
6  echo "-----"
7  read -p "Please_input_a_number:" number
8  case $number in
9  1)
10     echo "You_input_1"
11     echo "I_will_do_something_for_option_1..."
12     ;;
13  2)
14     echo "You_input_2"
15     echo "I_will_do_something_for_option_2..."
16     ;;
17  *)
18     echo "You_did_NOT_input_1_or_2"
19     ;;
20 esac
21 echo "Bye!"
```

运行上面的代码，可以看见：当用户输入 1 时，程序可以直接跳转到 1) 所对应的语句；对应的，当用户输入 2 时，可以跳转到 2) 所对应的语句；如果输入其他内容（例如输入 3），则会跳转到最后提示输入错误。

注意：与 *if* 语句类似，*case* 语句的前后，即开始的提示和后面的“Bye!”无论哪种情况都会输出，因为它们不属于 *case* 语句内。

字符串匹配

字符串匹配与单一字符匹配完全一样（你可以将单个字符理解成一种特殊的字符串）。但是，在这里我们也有一点新东西要讲。如果我们希望多个选项匹配同一个分支，可以使用 **|** 符号表示“或”进行分隔。例如，下面的程序则是根据用户输入的字符串选择特定的分支（有时多个输入对应同一个分支）：

Listing 4.18: favourite_band

```
1  #!/bin/bash
2  # 字符串匹配
3  read -p "Please input your favourite band:" band
4  case $band in
5  "PPP"|"ppp"|"Poppin'Party")
6      echo "Thanks! I have known that you like Poppin'Party!"
7      ;;
8  "Roselia"|"R")
9      echo "Thanks! I have known that you like Roselia!"
10     ;;
11  "RAS"|"RAISE_A_SUILEN"|"Raise a suilen")
12     echo "Thanks! I have known that you like RAISE_A_SUILEN!"
13     ;;
14  "MyGO!!!!"|"mygo"|"mygo!!!!")
15     echo "Thanks! I have known that you like MyGO!!!!"
16     ;;
17  *)
18     echo "Sorry... I don't know this band."
19     echo "But now I have known it -- $band"
```

```
20     echo "Thank_you!"
21     ;;
22 esac
23 echo "Bye!"
```

与前面的代码分析完全类似，但特别的是，这里面分支的判断使用 `|` 表示“或”，例如，“PPP”、“ppp”和“Poppin’Party”对应的都是同一个分支；类似地，“R”和“Roselia”也是对应同一个分支，当用户输入“R”或者输入“Roselia”时，得到的结果是一样的。

正则表达式匹配

正如一开始所说的那样，在匹配时可以使用**正则表达式**。例如，下面的代码试图读取用户输入的参数为字母还是数字⁸：

Listing 4.19: number_or_letter

```
1  #!/bin/bash
2  # 读取调用时的选项
3  export LC_ALL=C
4  case $1 in
5    [a-z])
6      echo "You_input_a_lowercase"
7      ;;
8    [A-Z])
9      echo "You_input_an_uppercase"
10     ;;
11    [1-9])
12      echo "You_input_a_number"
13      ;;
14    *)
15      echo "You_input_other_things"
16      ;;
```

⁸为了复习之前的变量种类，我们在这里尝试使用参数给定变量而不是使用 `read` 命令。

```
17  esac
18  echo "Bye!"
```

补充：也许你足够灵敏，注意到了第 3 行的 `export LC_ALL=C`，这句命令表示将程序运行语言环境设置为默认值（可以通过 `locale` 查看环境语言设置），其中 `C` 表示系统默认值。

在这里我们需要添加这一行命令以确保程序运行正确，但这行命令并不影响我们对 `case` 的理解。

同时，当你运行这一行代码之后，可能程序中的中文注释发生乱码，这并不影响程序运行结果，你可以重新启动系统以恢复开始的状态。

在调用时，当用户传递给一个小写字母时，则会匹配到 `[a-z]`，相对地，若给定一个大写字母作为参数，则会匹配到 `[A-Z]`。但是，我们这里只能对用户输入的一个参数进行判断，如果用户输入多个字符的参数（例如 `abc`），程序则会认为输入了其他内容。如何修改程序呢？请自己思考并尝试写一段代码⁹，同时自己测试它。（为简单起见，我们只需要匹配第一个字符即可）

同时，正则表达式也可以使用 `|` 作为分隔以进行多个情况的判断，例如，我们希望将上面的代码修改为无论大小写字母都判断为字母，则可以写成下面的样子：

Listing 4.20: `number_or_letter2`

```
1  #!/bin/bash
2  # 读取调用时的选项
3  export LC_ALL=C
4  case $1 in
5  [a-z]|[A-Z])
6      echo "You_input_a_letter"
7      ;;
8  [1-9])
9      echo "You_input_a_number"
10     ;;
11  *)
12     echo "You_input_other_things"
```

⁹请真的这样做，这会极大提高你的编程思维。

```
13     ;;  
14 esac  
15 echo "Bye!"
```

4.5.3 错误处理

-bash: < 文件名 >: line < 行号 >: syntax error near unexpected token ‘)’

这可能是因为你在结束分支时少了`;;`表示当前分支内结束。如果你了解 C 语言，你可能会很自然将这个符号作为 C 语言中 `switch-case` 的 `break` 语句。但显然，C 语言的 `break` 更加灵活（它可以没有从而越过其他分支），但 Shell 脚本不能没有`;;`结束。

-bash: < 文件名 >: line < 行号 >: syntax error near unexpected token ‘< 字符串 >’

这可能是由于你在使用 `case` 语句后忘记结束 `esac` 而后面还有其他语句从而报错。如果后面没有语句，则可能会出现下面的错误：

-bash: < 文件名 >: line < 行号 >: syntax error: unexpected end of file

这也表明你可能没有使用 `esac` 作为 `case` 语句的结束。

4.6 循环

本节作者：Jiaqi Z.

在本节，你将要学到：

- 如何使用 `for` 循环
- 如何使用 `while` 循环
- 如何使用 `until` 循环

- while 循环和 until 循环的区别

在前面介绍高级 Linux 命令时，我们曾说过脚本处理的任务大多是批量处理任务，而批量处理的一个基本方法就是使用循环语句（毕竟没有人会想写上上百行相同的代码）。在 3.5 一节中，我们介绍过 for 循环的使用，当时是在命令行中编写的。本节我们将首先复习一下 for 循环的基本使用方法，以及它在脚本程序中的格式（和命令行类似），然后再介绍两种不同的循环语句——while 循环和 until 循环——它们虽然在一定程度上是等价的，但在不同的情况下，选择合适的循环语句会让程序更加简洁易读。

4.6.1 for 循环

下面的代码完整演示了常见的三种 for 循环格式：

Listing 4.21: for_example

```
1  #!/bin/bash
2  # 使用for循环输出
3  # 简单列表
4  for i in 1 2
5  do
6      echo $i
7  done
8  echo ""
9  # 生成列表
10 for i in $(seq 1 2 10)
11 do
12     echo $i
13 done
14 echo ""
15 # 使用C语言格式
16 for ((i=0;i<10;i++))
17 do
18     echo $i
19 done
```

其中第 8 行、第 14 行使用 `echo ""` 输出一个空行用来区分不同的输出结果。

简单列表

使用基本列表的格式调用 `for` 循环的基本格式为：

```
for <变量名> in <列表>
do
    循环体
done
```

补充：在循环语句内部的语句，我们常常称其为循环体（实际上和前面所说的 *commands*，程序块，语句块等一样）

在这里的 `< 列表 >` 可以如同上面的代码一样直接以空格分隔表示，也可以与 3.5 一节所说的那样使用大括号将其括起来，并用逗号分割。

注意：一个常见的错误是将两者混淆，即使用大括号表示列表的同时，其内部元素用空格分隔。可以验证的是，这并不会如你所愿得到正确的信息。事实上，空格的“优先级”会高于大括号的“优先级”（这里加引号表示这不是传统意义上运算符的优先级），当括号和空格同时存在时，程序会将列表解析为以空格分隔的列表，从而在输出的前后（第一个元素和最后一个元素）带有大括号。

与前面的介绍类似，在循环体内使用变量需要加 `$` 符号。

生成列表

与 3.5 里所介绍的方法类似，可以使用 `seq` 命令生成序列。与前面所介绍的方法不同的是，前面所使用的是 ``` 符号括起来的命令，而在脚本中，由于将 `seq` 看作是一个命令，因此与前面 4.3.3 一节所介绍的输入方式类似，使用 `$()` 的方式得到 `seq` 命令的结果，并将其作为一个变量传递给 `for` 循环。

当然，也可以使用前面所说的 `..` 的方式生成序列。但需要复习的是：对于 `seq` 语句而言，三个参数分别是开始、步长、结束；而对 `..` 方式而言，三

个参数分别是**开始**、**结束**、**步长**。因此，上面代码的 `seq 1 2 10` 也可以写作 `{1..10..2}`

运算格式生成

如果你熟悉 C 语言的话，这一种格式会显得很“亲切”。因为它的基本结构与 C 语言几乎完全类似，类似地写法，如果让我们用 C 语言实现上面代码的第 3 部分，则可以写作这样：

```
1  #include <stdio.h>
2  void main()
3  {
4      for (int i=0;i<10;i++)
5          printf("%d\n",i);
6  }
```

可以看到，相比于 C 语言，脚本只是一个括号和两个括号的区别¹⁰。

注意：在脚本的 `for` 循环当中，括号前面没有 `$` 符号，也千万不要“画蛇添足”加上这个符号。

4.6.2 while 循环

与 C 语言的 `while` 循环类似，在 `bash` 脚本中也存在根据条件判断循环与否的 `while` 循环。其基本格式如下：

```
while [ condition ]
do
    循环体
done
```

其中，`condition` 表示**循环进行的条件**，其格式与 4.4 当中所介绍的条件表达式类似。例如，下面的代码实现了从 0 到 9 的输出

¹⁰由于 C 语言是强类型语言，因此在 `for` 循环条件中定义了变量类型 `int`，但这不是必需的，因为完全可以将变量定义放在循环外面作为单独的语句，此时 C 语言的括号内与脚本的括号内就完全相同了。在比较二者不同时，我们有意忽略了这一点，只是为了让大家关注到本质

Listing 4.22: while_example

```
1  #!/bin/bash
2  # 使用while循环输出数字
3  i=0
4  while [ $i -lt 10 ]
5  do
6      echo $i
7      ((i++))
8  done
```

可以复习一下，条件`-lt`表示小于，因此，循环体进行的条件是变量 `i` 小于 `10`。当条件满足时，程序进行循环体（`do` 和 `done` 中间的部分）。因此，程序会从 `0` 开始，逐渐输出并加 `1`，直到条件不满足时（`i` 为 `10`）则停止循环。

注意：与 `for` 循环相比，`while` 循环更有可能写出“死循环”语句。所谓“死循环”，指的是程序在循环体内一直循环，永无停止。在上面的代码中，如果少了那句 `((i++))`，则变量始终为 `0`，条件始终满足，从而无法停止。

具体的解决方法，可以见下一节所介绍的 `break` 和 `continue` 语句。

在实际操作中，如果出现死循环导致程序无法停止，则可以使用 `Ctrl+C` 快捷键终止当前命令。

此外，在上面程序的第 7 行，使用 `((i++))` 表示进行计算，这是比较简洁的 C 风格递增运算符，类似的还有递减运算符`--`。这种方式的使用需要以两个括号括起来。你也可以使用 4.2 一节所介绍的方法，使用 `i=$(($i + 1))` 的方式。这二者在这一功能上是等价的。

4.6.3 until 循环

与 `while` 循环几乎完全类似，`bash` 脚本中 `until` 循环的格式为：

```
until [ condition ]
do
    循环体
```

done

与上面的 `while` 循环格式比较可以发现，除了关键字从 `while` 变成了 `until` 外，其他语句在格式上没有任何不同。但 `until` 循环的最大特点是：循环体执行的条件是 `condition` 为假，即你可以简单的将其理解为：循环体会一直执行，“直到”（`until`）条件为真。

注意：这一点稍微有点绕，但上面的两种表述是“等价”的。对 `while` 循环而言，当条件为真时执行循环体，而对 `until` 循环而言，当条件为真时退出循环（当条件为假时进入循环）

补充：在 `C` 语言中，确实没有类似的循环与其对应，但我们可以从其他一些编程语言中找到这个例子，一个最简单的例子就是——`Visual Basic` 语言（简称 `VB` 语言）。在 `VB` 语言中，也存在类似的 `until` 循环，其格式为：

```
Do
    循环体
Loop Until 条件
```

如果你熟悉 `VB` 语言的话，`bash` 脚本的 `until` 循环与 `VB` 语言的 `until` 循环最大的不同在于 `VB` 语言的条件是放在循环后面（类似于 `C` 语言的 `do-while` 循环）

当然，如果你不熟悉 `VB` 语言的话，也不必担心。这一部分仅仅是对那些熟悉 `VB` 语言的读者所准备的，以防他们混淆条件的位置。如果你本来不了解 `VB` 语言的话，这一部分完全可以跳过。

可以简单想见的是，`while` 循环和 `until` 循环之间的转换仅仅是“条件的取反”，例如，上面关于 `while` 循环的例子，我们只需要将里面的条件 `$i -lt 10` 改为 `$i -ge 10`，同时将 `while` 改为 `until` 就可以实现相同的功能。这一部分代码我们不做演示，请自己尝试修改上面的代码并测试其输出结果是否与之前的结果一致。

4.7 循环控制

本节作者：Jiaqi Z.

在本节，你将要学到：

- 如何使用 `break` 语句
- 如何使用 `continue` 语句

在 4.6 一节中，我们曾简单提到了“死循环”问题。在当时看来，死循环是一个应当避免的“错误”，但事实可能不总是如此——在有些情况下，我们可能不得不希望使用死循环。例如，我们希望一直读取用户的输入，直到用户输入 0 时退出。此时当然可以使用 `while` 循环或者 `until` 循环来解决这一问题。但还有一种思路——设计一个死循环，当用户输入 0 时跳出循环。

补充：在其他领域，死循环可能是更常见的。例如，在一些单片机系统中，会使用死循环来执行对应的任务，例如读取传感器数据、显示处理数据等；而在 *Windows* 操作系统中，任何一个窗口在创建时都会启动叫做“消息循环”的死循环来保持窗口——对于 *Windows* 来说，如果没有这个死循环，窗口会立即关闭。

因此，在程序中，死循环并不是“绝对的错误”，而要根据需要选择。一些可控的、必需的死循环也是程序所必要的一部分。

回到最开始的例子，在 `bash` 语言中，我们有两种方式对循环语句进行控制——`break` 语句和 `continue` 语句。其中前者表示“终止循环”，而后者表示“继续循环”

4.7.1 使用 `break` 停止循环

补充：本节所要介绍的 `break` 与 `continue`，在使用方式与效果上与 *C* 语言的 `break` 和 `continue` 完全相同。因此，如果你足够熟悉 *C* 语言，本节完全可以跳过。但我们还是建议你通过阅读这一节复习一下相关的内容。

`break` 语句的作用是用来跳出循环，例如，下面的程序，用户输入一系列数字，当用户输入 0 时，循环结束，输出“end”

Listing 4.23: `break_example`

```
1  #!/bin/bash
2  # break跳出循环
3  read a
4  while [ true ]
5  do
6      if [ $a -eq 0 ]; then
7          break
8      fi
9      echo $a
10     read a
11 done
12 echo "end"
```

在第 4 行，我们设置了一个条件 `true`，使循环条件始终成立，从而令其成为一个“死循环”。在第 6 行利用 `if` 语句判断输入的变量是否为 0，如果是 0，则通过 `break` 跳出循环，执行最后一行的“end”输出。如果输入的数字不为 0，则输出对应的变量值。

注意：请务必仔细思考一下，程序的第 10 行又写了一句 `read a` 有什么作用？如果没有这句命令会发生什么？在程序中有两句 `read` 命令稍显繁琐，有没有简单的方法使用一句即可实现相同的功能？

除了在 `while` 循环外，`for` 循环和 `until` 循环也可以使用 `break` 语句。例如，下面的程序遍历了 10-20 之间所有的数字，且当数字为 7 的倍数时停止。

Listing 4.24: break_example2

```
1  #!/bin/bash
2  # 在for循环中使用break语句
3  # 当遍历到7的倍数时停止
4  for i in {10..20}
5  do
6      if [ $(( $i%7 )) -eq 0 ]; then
7          break
8      fi
```

```
9      echo $i
10  done
11  echo "end"
```

其中，变量 `i` 会依次从 10 递增到 20，当 `i` 取值为 14 时，满足 7 的倍数¹¹，从而进入条件判断内，执行 `break` 命令，跳出循环输出 `end`

注意：上面的两个例子还说明了，`break` 跳出循环是“立刻跳出”，即如果循环体内有剩下的其他语句也不会执行了。例如，上面的判断倍数的例子，当 `i` 取值为 14 时，也没有输出这个值（因为输出命令在 `break` 后面。

可以尝试一下：如果将 `echo $i` 放在循环体内第一行，输出结果会有什么不同？

4.7.2 使用 `continue` 跳过循环

与 C 语言类似，在 `bash` 脚本中，也可以使用 `continue` 跳过循环。这里所说的“跳过”，指的是跳过当前轮次的循环。让我们直接来看一个例子——假设将上面判断 7 的倍数的例子中的 `break` 改成 `continue`，看看会发生什么。

Listing 4.25: `continue__example2`

```
1  #!/bin/bash
2  # 使用 continue 语句跳过循环
3  for i in {10..20}
4  do
5      if [ $(( $i%7 )) -eq 0 ]; then
6          continue
7      fi
8      echo $i
9  done
10 echo "end"
```

¹¹这里使用常见的方法——“取余”判断倍数，具体的计算方法可以查看 4.2.4 一节。

运行后可以看见，程序输出了 10-20 的所有数字，除了 14。这是因为当变量 `i` 为 14 时，条件满足，执行 `continue` 语句，跳过了当前轮次的循环，直接进入下一个循环轮。与 `break` 类似，这里也不再执行后面的语句（即不会输出 14 这个数字）

虽然 `continue` 在语法上和 `break` 类似，难度也不大，但其使用时存在一个非常潜在的“隐患”。让我们再来看一下本节最开始的程序，如果将其中的 `break` 改成 `continue`，会发生什么？此时代码长成下面这样：

Listing 4.26: `continue_example`

```
1  #!/bin/bash
2  # 使用continue跳过0的判断，继续输入？
3  read a
4  while [ true ]
5  do
6      if [ $a -eq 0 ]; then
7          continue
8      fi
9      echo $a
10     read a
11 done
12 echo "end"
```

一些“粗心大意”的人可能会认为：这段代码一直读取用户的输入并输出对应的内容，当用户输入 0 时跳过输出。让我们执行一下这段代码，当用户输入非 0 的数字时，程序很正常地输出对应的数字；当用户输入 0 时，程序也确实跳过了 0 的输出；但是，从这开始程序再也读取不了输入了。从程序代码的执行过程可以很容易理解这一点，当用户输入 0 时，此时变量 `a` 的值为 0，进入条件判断内，跳过当前循环。正如前面所说的那样，程序跳过循环不再执行后面的语句。因此，在后面的循环中变量 `a` 始终保持 0 的值，即始终在 4-7 行语句间循环。

注意：对于 `for` 循环而言，由于它是对序列进行遍历，一般情况下总是“有限”的，因此循环总是能停止的。而对于 `while` 和 `until` 循环而言，在循环开始和 `continue` 中间，要有改变循环条件或者终止循环的语句，否

则循环将陷入“死循环”。

4.8 * 函数

本节作者：Jiaqi Z.

在本节，你将要学到：

- 如何定义与调用函数
- 函数参数
- 函数返回值

前面我们已经学习了程序的基本结构——顺序语句、条件语句和循环语句。此时，应当已经足够完成大多数脚本程序的编写。在本节和下一节，我们将介绍一些扩展性的内容——函数和数组。这两部分可以帮助你更快、更方便地编写程序。

4.8.1 定义函数

在 bash 脚本中，函数的定义必须在调用之前。这是因为脚本语言是按照顺序执行的，如果在程序不知道的情况下直接调用，那必然会报错。在 bash 脚本中，对一个函数定义的基本格式为：

```
[function] function_name()
{
    函数体;
    [return int. ;]
}
```

在定义时，可以在前面加上 `function` 以示区分，也可以不加直接以函数名作为开始。在函数内部的语句块使用大括号括起来（函数体），在函数的结束可以使用 `return` 语句返回一个范围在 0-255 之间的整数。

注意：与大多数程序语言不一样，*bash* 脚本对函数返回类型有明确的数值要求，且这个要求是不随程序员所改变的。如果要返回函数运行结果是

否成功，可以使用 `return` 语句，但如果返回的是其他内容（例如返回两个数字相乘的结果），则可以使用字符串的形式返回。

在 `bash` 的函数定义中，我们不会在函数开始的地方强调参数的个数和类型（这点与 `C` 语言等类似语言不同），具体的参数将在函数体内部使用 `$num` 的形式体现（与 4.3.2 所介绍的参数类似）

例如，下面的程序简单定义了一个名字叫做 `my_echo` 的函数：

Listing 4.27: `my_echo_function`

```
1  #!/bin/bash
2  # 定义函数
3  my_echo()
4  {
5      echo "Hello"
6  }
```

这个函数内部只有一个输出语句，且没有返回值。

补充：我们这里所说的“没有返回值”，特别指的是不使用 `return` 返回的返回值。你也完全可以将这个函数的 `echo` 输出看作是一个字符串返回。

有参数函数的调用

与 4.3.2 一节所介绍的方法类似，在函数内部我们也可以使用 `$1` 表示第一个参数，当参数个数大于等于 10 时，则需要使用大括号将数字括起来，例如，`${10}` 表示第 10 个参数。

在函数体内部，我们也可以使用 `$#` 表示参数的个数（与前面所介绍的方法一样），可以使用 `$` 将所有参数以字符串的方式输出。例如，下面的程序就实现了两个整数相加的计算函数：

Listing 4.28: `add_function`

```
1  #!/bin/bash
2  # 两个整数相加
3  function add_function() {
```

```
4     c=$(( $1 + $2 ))
5     echo $c
6 }
```

这里的第 4 行我们使用 `$1` 和 `$2` 分别表示第 1 个参数和第 2 个参数，并将计算结果赋值给变量 `c`，并在最后输出变量 `c` 的值。

在这里请思考一下：为什么我们的程序使用 `echo` 语句输出返回值，而不是使用 `return` 语句返回结果？

【答案】：因为两个数字相加的结果可能超过 255。

4.8.2 函数调用

在定义函数的基础上，调用函数就可以将函数看作是一个新的命令使用了。例如，对于我们定义的 `my_echo` 函数，在程序内就可以直接使用 `my_echo` 命令调用这个函数，从而输出“Hello”。

对于下面的 `add_function` 函数，类似于前面在介绍命令行传递参数一样，我们也可以使用类似的方式调用这个函数。下面的代码则完整展示了函数定义与调用的全过程：

Listing 4.29: `add_function`

```
1  #!/bin/bash
2  # 两个整数相加
3  function add_function() {
4      c=$(( $1 + $2 ))
5      echo $c
6  }
7  result=$(add_function 200 300)
8  echo ${result}
```

我们在第 7 行使用了一个赋值语句特别说明了如何将函数 `echo` 的返回值作为一般的返回值处理，也可以看到，对于这个例子，当计算 `200+300` 时，程序可以给出正确的结果（500）。

在一般使用时,我们也可以直接省略掉后面两行,而直接使用 `add_function` 200 300 也能输出类似的结果,当然,本例所给的方式更具有可扩展性。例如,你可以对这个结果进行进一步的计算,例如判断它是否大于 255。(请试着完成这段代码)

最后,让我们再稍微扩展一下,如果用户在最开始给定一个数字个数,使用上面的函数,实现多个整数相加的结果。例如,用户一开始输入 5 表示有 5 个整数进行相加,然后会依次输入 5 个整数,程序输出 5 个数字相加的结果。我们在这里给出最终答案,请试着理解这段代码,并在理解后尝试自己写出来实现相同的效果:

Listing 4.30: add_function2

```
1  #!/bin/bash
2  # 多个整数相加
3  function add_function() {
4      c=$(( $1 + $2 ))
5      echo $c
6  }
7
8  read -p "Please input a number" count
9  index=1
10 a=0
11 while [ $index -le $count ]
12 do
13     read b
14     result=$(add_function $a $b)
15     a=$result
16     ((index++))
17 done
18 echo $result
```

在分析这段代码时,请务必考虑的内容是: `while` 内部的循环条件为什么是这样(如果写成 `-lt` 会发生什么);为什么第 15 行要有一句 `a=$result`,它的作用是什么?在整个程序中,`index` 变量的作用是什么,为什么它的初

始值设置为 1；同时，为什么在开始时 `a` 有一个初始为 0 的值？

当你想明白这些答案时，相信这段代码就没有难度了。当然，我们这里是逐一计算的，那有没有办法在存下多个数据后一次性计算呢？这里的难点在于如何存储多个数据，这就需要下一节**数组**相关的内容了。

4.9 * 数组

本节作者：Jiaqi Z.

在本节，你将要学到：

- 如何初始化定义数组
- 如何调用数组的元素
- 如何定义“关联数组”

在上一节的最后，我们提到：需要有一种手段保存所有的元素。这在当时是一种可选的手段，除此之外，还有一个更常见的例子：假设我们现在需要计算一个班级所有同学考试的平均分，并在最后输出高于平均分的分数。此时，寻找一个方法用来存放所有数据就是必要的了。而与其他编程语言类似，在 `bash` 脚本中，也有类似于“数组”这样的数据结构。

在本节，我们将介绍如何在程序中显式定义¹²一个数组，以及如何调用数组的元素；在最后，我们再简单讨论一下**关联数组**的内容，这是一个类似于 Python 中“字典”的哈希表（Hash Map）结构。

4.9.1 定义数组

定义数组的方法并不困难，其基本格式为 `< 变量名 >=(value1 value2 value3 ...)`，其中数组变量元素整体用小括号括起来，且各个元素之间用空格间隔。例如，下面的程序我们定义了一个数组：

Listing 4.31: `array_test`

```
1  #!/bin/bash
2  # 定义数组
```

¹²这里所说的“显式定义”，指的是在程序代码中直接写明数组变量的值

```
3 array1=(1 2 3 4 5)
4 array2=("Hello" "World")
5 array3=("Hello" 3 true)
```

在这段代码中，我们定义了三个数组，第一个数组 `array1` 和第二个数组 `array2` 分别是由数字和字符串组成的数组。在 `bash` 脚本中，数组不仅可以是数字，也可以是字符串或者其他数据类型。与 C 语言固定数组类型不同，在 `bash` 脚本中，一个数组内可以有多个数据类型，正如 `array3` 数组所演示的那样，其中可以包括字符串、数字、甚至逻辑值，都是可以放在一个数组变量下。

注意：在定义数组时，与 C 语言不同，`bash` 脚本不需要提供数组长度。换句话说，`bash` 脚本的数组是可变长度的数组。

而且，与其他编程语言不同，`bash` 脚本仅支持一维数组。

4.9.2 数组调用

与 C 语言类似的是，数组调用也是采用 `< 数组名 >[下标]` 的格式，其中下标是使用中括号括起来。一般数组的调用分为“读取”和“写入”两种情况，我们将依次讨论这两种情况的区别。

注意：一个很关键的事情是：数组下标始终是从 0 开始计数的，即 `array1[0]` 表示数组 `array1` 的第一个元素。这件事情几乎是当今所有编程语言的基础¹³

读取调用

下面的代码演示了如何读取数组的元素：

Listing 4.32: `array_test_cont1`

```
1 #!/bin/bash
2 # 定义与调用数组
3 array1=(1 2 3 4 5)
4 array2=("Hello" "World")
5 array3=("Hello" 3 true)
```

¹³这一切是从 C 语言开始的。

```
6 # 调用数组
7 result=$(( ${array1[0]}+${array1[1]} ))
8 echo $result
9 echo ${array2[0]}
10 if [ ${array3[2]} ]; then
11     echo ${array3[0]}
12 fi
```

上面代码的第 7、9、10 行分别演示了三种常见的调用方式，分别是在表达式中调用；在语句中调用；在逻辑表达式中调用。可以发现，无论哪一种调用方式，其格式基本是一样的，因此，读者完全不必将其分类到如此精确的程度，只需要记住一个原则：在读取时需要加 \$ 符号和大括号即可。

写入调用

在必要的时候，我们也可以对已有的数组进行修改。例如，下面的代码是对上面代码的进一步补充，并在其中演示了如何修改数组的元素值：

Listing 4.33: array_test_cont2

```
1 #!/bin/bash
2 # 定义与调用数组
3 array1=(1 2 3 4 5)
4 array2=("Hello" "World")
5 array3=("Hello" 3 true)
6 # 调用数组
7 array1[0]=5
8 result=$(( ${array1[0]}+${array1[1]} ))
9 array1[5]=$result
10 result[1]=${array2[0]}
11 if [ ${array3[2]} ]; then
12     array3[2]=false
13 fi
14 result[2]=${array3[2]}
15 for i in {0..2}
```

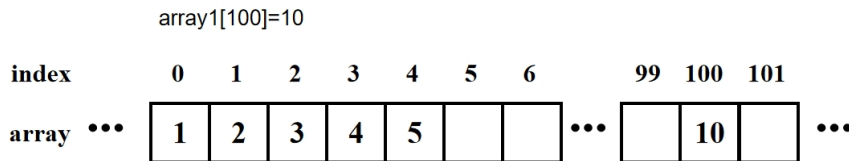


图 4.4: 添加数组元素后数组内元素的值

```

16 do
17     echo ${result[$i]}
18 done

```

在程序的第 8 行，我们将数组元素当作一般的变量，并对其进行赋值。但稍有难度的是第 9 行和第 10 行，其中第 9 行明明数组的最大下标为 4（想想为什么 5 个元素的最大下标为 4），但我们却可以对下标为 5 的元素赋值。在 bash 脚本中，这可以视作**添加元素**。与 Python 等语言可能还需要类似于 `append()` 函数不同，bash 脚本只需要如同正常的数组进行操作即可，程序会自动追加新的元素在对应的位置。

更奇怪的是第 10 行代码：为什么 `result` 明明是一个普通变量，却可以带有下标追加新的元素？事实上，一个普通的变量也可以看作是一个**长度为 1 的数组**。正因如此，我们可以对普通变量如同数组般进行操作（追加新的元素），甚至对于普通的变量，例如定义了变量 `a=5`，我们也可以使用 `a[0]` 来代表这个变量。

补充：我们可以将整个数组看作一个一维的表格，其编号从 0 开始，每创建一个元素，就在对应位置加上内容。因此，你可以将下标设置成不连续的（例如，`array1[10]` 是允许的，此时的数组如图 4.4 所示）

正如图中所示的那样，数组前面也可以有元素（负下标），但这些除非特殊情况，否则不建议使用。同样，也正如你所见，对于没有的元素，其变量值为“空”。

除此之外，还有一些特别的方法对数组进行调用。例如，借助于 `*` 符号，我们可以获取数组内所有元素的值（其方法为 `< 变量名 >[*]`；使用 `#` 符号可以获取数组的长度。例如，对于数组 `array1`，可以通过 `${#array1[*]}` 的方式获取其数组长度。

补充：关于获取数组长度的方法，实际上近似等价于在介绍参数时(4.3.2)所提到的判断参数个数的方法。也如同当时所介绍的方法一样，这里的 `*` 也可以换成符号。

4.9.3 * 关联数组

补充：在创建数组时，默认都是以数字作为下标。类似于 *Python* 中的“字典”结构，在 *bash* 脚本中我们也可以创建带有“键值对”的哈希表结构，称为关联数组。在定义关联数组前，需要使用 `declare -A` 的方式声明这个数组是关联数组。之后，就可以在下标中使用其他类型的数据（如字符串）对数组内的元素进行赋值。例如，下面的代码演示了如何定义与调用关联数组：

Listing 4.34: associative_array

```
1  #!/bin/bash
2  # 定义与调用关联数组
3  declare -A data=([name]="Jiaqi Z.")
4  data["band"]="Roselia"
5  # 调用关联数组
6  echo "My_name_is_${data["name"]},_and_I_love_${data["band"]}."
```

在第 3 行，我们使用 `declare -A` 定义了一个“关联数组”，并在其中初始化了一个下标¹⁴为“name”的元素，其值为“Jiaqi Z.”。

在第 4 行，我们又如同调用一般的数组一样，对其进行赋值（追加了一个新的键）。之后在第 6 行调用了这个数组，并如同一般调用下标那样调用这个数组的“键”从而获得对应的“值”。

需要注意的是：在初始化定义时，需要特别指明元素的键（其基本格式如上面代码第 3 行那样），而在后续的使用中，则完全可将其视为特殊的数组使用（调用）

与前面所介绍的获取数组所有元素类似，对于“关联数组”而言，我们也可以使用 `< 变量名 >[*]` 的方式获得其所有元素的值。只不过在这里仅输出所有“值”（不包括“键”）。如果希望输出所有“键”的元素，可以在前

¹⁴在哈希结构中，我们通常称这个下标为“键”（key），而称这个数组的元素值为“值”（value）。

面加感叹号 (如 `!
变量名
[*]`)。例如, 对于上面的代码, 可以使用 `echo
${data[*]}` 的方式输出所有值, 使用 `echo ${!data[*]}` 的方式输出所有“键”。

Part II

VASP 计算

写在前面的一些说明

从本部分开始，就进入到了这一教程的主体部分—**VASP 计算**。在开始计算之前，有一些注意事项需要说明：

关于输入文件

随着计算任务的不同，VASP 所需要的输入文件也是不同的。对于 INCAR 和 KPOINTS 文件¹⁵，通常是与具体的计算任务有关，且可以借助于如 vaspkit 的脚本生成。而对于 POSCAR 文件而言，其表示所要计算的结构信息，对于不同的课题组、不同的研究课题，所研究的结构也会千差万别。在教程中为了演示方便，有时会设定某一特定的结构作为 POSCAR 文件，仅作为演示用，在实际使用时需要根据具体问题设置不同的文件。

对于 POTCAR 文件而言，通常对于已经购买版权的课题组而言，都会有一个配套的 POTCAR 目录，里面会包含有所有元素的赝势文件。对于这种情况，通常使用如 vaspkit 的脚本生成并不是难事（同样也可以使用 Linux 命令手动生成，具体内容在后续章节会进行介绍）。对于没有购买版权的课题组来说，可以“暂时借用”别人已有的文件**作为练习**，但不能将其用于课题组的论文当中。

关于提交脚本

之前所介绍的 Linux 命令，在大多数课题组的系统中都是可以使用的。但 VASP 却不是如此。首先，不同课题组的 VASP 版本可能不同，有时不同版本的命令或参数含义可能会有些许变化，但这种变化通常是影响较小

¹⁵这些文件的具体含义在后续教程中都会详细介绍。

的。最重要的是，由于所使用的系统环境不同，例如，对于本地运行和运算集群运行，其提交任务的方法可能会有些许差异。目前，大多数课题组在计算 VASP 任务时都是采用服务器集群进行计算，此时就会需要一个叫做**排队系统**的东西。

对于不同课题组的不同集群，所使用的排队系统可能不同。本教程在编写时，通常是使用 slurm 作业管理系统，目前如中国科学技术大学、上海交通大学等学校的计算中心都是采用这一管理系统。对于使用其他管理系统的课题组而言，需要参考自己课题组的使用方法。

使用 slurm 的命令与方法

考虑到教程的完整性，这一节简单介绍关于 slurm 的命令。

注意：这一部分仅仅适用于那些使用 *slurm* 作业管理系统的课题组，对于其他课题组，则需要参考自己课题组的使用方法。

在使用 slurm 时，需要配合以一个提交任务脚本。一个典型的提交任务脚本 `sub.vasp` 如下所示：

Listing 4.35: sub.vasp

```
#!/bin/bash
#SBATCH -n 56
#SBATCH -N 1

# 打印任务信息
echo "Starting job $SLURM_JOB_ID at " `date`
echo "SLURM_SUBMIT_DIR is $SLURM_SUBMIT_DIR"
echo "Running on nodes: $SLURM_NODELIST"

# 执行任务
## 载入 vasp
module load VASP
ulimit -s unlimited
mpirun vasp_std > vasp.out 2>vasp.err
```

```
# 任务结束
echo "Job $SLURM_JOB_ID done at " `date`
```

其中, `#SBATCH -n` 表示任务所使用的核数, 在本例中设定为 56 核; `#SBATCH -N` 表示使用的节点数。在上述代码中, 表示使用 56 核, 1 个节点进行计算。

而对于中间的命令, 特别的如 `ulimit -s unlimited` 表示不设置内存限制; `mpirun vasp_std > vasp.out 2>vasp.err` 表示通过 `mpirun` (即并行计算程序) 运行 `vasp_std` 命令并将输出结果保存至 `vasp.out`, 而将错误信息输出到 `vasp.err` 当中。

补充: 对于有显卡加速的课题组而言, 可能需要在前面指定 `#SBATCH --partition=GPU` 指定使用的 GPU (如 `a100` 等), 同时使用 `#SBATCH --gres=gpu:n` 表示调用 n 张显卡进行计算。

提交任务时, 需要将提交脚本和计算目录放在一起, 在计算目录下使用 `sbatch sub.vasp`¹⁶ 即可。除此之外, `slurm` 还有如下命令:

- `squeue` 表示查看当前任务队列
- `scancel [jobID]` 表示取消 `[jobID]` 编号的任务, 例如, `scancel 6066` 表示取消编号为 6066 的任务;

除此之外, `slurm` 也可以使用如 `srun` 或 `salloc` 提交交互式作业, 或者申请特定的资源并登录至节点。这一部分内容在 VASP 计算时不会使用到, 因此不在此处介绍。

¹⁶ 提交脚本名

Chapter 5

能带计算

Contents

5.1 能带基础理论	121
5.1.1 什么是能带	122
5.1.2 能带理论三个近似	123
5.2 VASP 计算能带过程	123
5.2.1 结构优化	123
5.2.2 自洽计算	126
5.2.3 能带计算	127
5.3 能带绘图与后处理	129
5.3.1 使用 Origin 绘制能带图	130
5.3.2 使用 vaspkit 自动生成能带图	133
5.3.3 如何计算带隙	135
5.4 HSE 能带计算	137
5.4.1 结构优化与自洽计算	137
5.4.2 HSE 能带计算	138
5.4.3 HSE 能带计算后处理	140

5.1 能带基础理论

本节作者: Jiaqi Z.

在本节，你将要学到：

- 什么是能带
- 能带的三个重要近似

在本章，我们将要了解材料计算的一个重要内容—关于能带的计算。毫不夸张地说，能带论是目前研究固体中的电子状态，说明固体性质最重要的理论基础。一个最简单的例子是，利用能带的相关计算，我们可以从严格的角度判断材料的导电性。

补充：导体和绝缘体的概念，贯穿了我们的学习生涯。而这一概念也在随着认知水平的增长发生变化。

最开始接触的时候，我们认为，导体是那些带有电荷的物质，而绝缘体内部没有电荷。这一结论是我们对“电”和“电荷”的初步认识，显然是不准确的。进一步学习了物理后，我们了解到：任何原子都是由原子核和电子组成的，所谓的导体，就是存在“自由移动的电子”，相反，绝缘体就是没有自由电子的物质。这一概念结合了原子和电子的认识，相比于最开始的“电荷”，显然更准确了。

到了现在，我们将了解到能带。基于能带理论，在导体中，价带（价电子所在的能带）和导带（电子可以自由移动的能带）是重叠的，或者价带顶部和导带底部之间的能隙（带隙）非常小，甚至为零。这意味着电子可以轻易地从价带跃迁到导带，从而在电场作用下自由移动，形成电流。而对于半导体而言，价带和导带之间存在一个较大的带隙，电子要跃迁到导带需要吸收足够的能量（如热能或光能）。在常温下，电子通常没有足够的能量来跃迁，因此电子不能自由移动，导致绝缘体不导电。

5.1.1 什么是能带

从原子物理的知识来看，一个孤立原子的电子只能处在特定的能级当中。而我们所计算的材料，往往是周期性的多原子材料¹。对于多原子而言，电子和电子之间、电子和原子核之间会产生相互作用，此时电子的能级就会发生“展宽”，从而变成一系列的带状结构。称为“能带”。

注意：通常情况下，能带仅在周期性材料中讨论。对于单分子（如气体分子等），计算能带往往没有意义。

¹对于 VASP 而言，所有材料都是“周期性”的。我们所说的单原子或单分子计算，通常是通过调整晶格的大小，从而减弱相互之间的作用，近似成“单分子”计算。

5.1.2 能带理论三个近似

我们不会在这一教程中详细介绍能带的推导过程，但是我们还是有必要在这里提到三个重要的近似。这些近似可以说是能带理论的基础，甚至可以说是整个第一性原理计算的基础。

- Born-Oppenheimer 近似，又名绝热近似：因为原子核比电子重的多，所以原子核比电子具有更大的惯性，更难运动。因此，我们只考虑电子的运动，原子核是被固定住的。
- 单电子近似（独立电子近似、平均场近似）：将电子与电子相互作用等效成一个平均值，电子是在一个平均场中运动。
- 周期场近似：平均场是周期性的。

具体内容在任何一本固体物理教材中都会详细提到，这里不再赘述。

5.2 VASP 计算能带过程

本节作者：Jiaqi Z.

在本节，你将要学到：

- 如何使用 VASP 计算 PBE 能带

在本节，我们将详细讨论如何使用 VASP 计算能带。我们先讨论最简单的 PBE 能带计算过程，旨在通过这一流程，掌握计算能带的完整步骤。在这一基础上，后面将会详细讨论精度更高的计算方法（如 HSE 能带计算等）。

注意：使用 *PBE* 计算能带往往会得到较小的带隙，如果你使用数据库或文献中的能带图进行复现，可能会得到与文献不同的带隙。这一点是 *PBE* 泛函计算能带所固有的缺陷，

5.2.1 结构优化

在这一部分计算能带时我们使用 SiO_2 为例进行分析。所使用数据库来源自 Materials Project²。

²<https://legacy.materialsproject.org/materials/mp-546794/>

SiO₂的结构文件 POSCAR 如下所示：

Listing 5.1: POSCAR

```

Si2 O4
1.0
      5.1358423233      0.0000000000      0.0000000000
      0.1578526541      5.1334159104      0.0000000000
      -2.6468476750     -2.5667081359      3.5753437737
Si    O
2     4
Direct
      0.750000000      0.250000000      0.500000000
      0.000000000      0.000000000      0.000000000
      0.787033975      0.625000000      0.662033975
      0.875000000      0.212965995      0.837966025
      0.962966025      0.125000000      0.337965995
      0.375000000      0.037034001      0.162034005

```

注意：在计算能带时，大多数时候我们只讨论原胞的计算。因此，在使用 *Materials Project* 等数据库导出结构时，应当优先导出原胞结构（*Primitive Cell*）。

对于无法导出原胞的情况，可以借助于其他程序或脚本文件。以 *vaspkit* 为例，借助于 *vaspkit-602*，可以得到 *PRIMCELL.vasp* 文件，将其重命名为 *POSCAR* 文件即可。

补充：对于一些特殊情形（例如需要掺杂等情况），不得不使用超胞进行计算。如果确实需要计算能带结构，往往需要对能带进行反折叠以得到更清楚的图像。我们将在后面的部分对这一技术进行讨论。

目前，我们所讨论的结构都是原胞。

在计算能带之前，首先需要对材料进行结构优化。为得到结构优化所用 INCAR 文件，使用 *vaspkit-101-LR* 生成。同时调整其中的部分参数，修改后的 INCAR 文件如下：

Listing 5.2: INCAR

```
Global Parameters
ISTART = 1
LREAL = .FALSE.
ENCUT = 600
PREC = Accurate
LWAVE = .TRUE.
LCHARG = .TRUE.
ADDGRID= .TRUE.

Lattice Relaxation
NSW = 300
ISMEAR = 0
SIGMA = 0.05
IBRION = 2
ISIF = 3
EDIFFG = -1.5E-02
```

其中，需要特别注意并调整的是：

- **ENCUT**: 截断能。通常设置为 600 或更高，但更高的截断能往往意味着更长的机时。同时，在后续所有计算中，截断能应当保持不变。
- **ISIF**: 表示优化方式。对于一般的结构优化，通常设置为 **ISIF=3** 表示**优化原子坐标和晶格参数**。对于一些特殊的材料（如二维材料），一些晶格参数可能不希望发生变化，此时可以设置 **OPTCELL** 文件，其内容为 3×3 的矩阵，分别对应 **POSCAR** 当中的晶格参数坐标。其元素可以是 0（表示不优化该坐标）或 1（表示优化该坐标）。对于晶格参数不变的情况，可以设置为 **ISIF=2** 表示**只优化原子坐标**。
- **EDIFFG**: 表示优化收敛标准。其中正数表示**能量收敛标准**（即能量变化小于这一数值时停止计算），而负数表示**力收敛标准**（原子作用力小于这一数值的绝对值时停止计算）。
- **NSW**: 表示**最大离子步**。当优化离子步达到设定数值时停止计算（此时往往未达到收敛标准，需要重新计算）。或者，当 **EDIFFG=0** 时，计算

达到设定 NSW 时停止计算。

对于 KPOINTS 文件，在结构优化时可以使用“自洽计算”的 K 点，使用 `vaspkit-102` 生成，通常设定 Gamma 点 (2)，选择密度时通常设定为 0.02-0.04 即可。

本例使用 `vaspkit-102-2-0.02` 生成 KPOINTS 文件如下所示：

Listing 5.3: KPOINTS

```
K-Spacing Value to Generate K-Mesh: 0.020
0
Gamma
  12 12 14
0.0 0.0 0.0
```

注意:使用 `vaspkit` 生成 K 点的同时,脚本会同步生成赝势文件 `POTCAR`。但为了确保生成文件的正确性,建议使用 `grep TITEL POTCAR` 查看赝势文件是否正确(与 `POSCAR` 文件相比较)³。

将以上文件放置在一个目录下,提交任务计算后得到 `CONTCAR` 文件,即为优化后得到的结构文件。

5.2.2 自洽计算

计算完成后,新建一个目录(例如命名为 `scf`),将结构优化得到的 `CONTCAR` 文件复制(或移动)到 `scf` 目录内,并重命名为 `POSCAR`。

将结构优化的 KPOINTS 和 POTCAR 复制(或移动)到 `scf` 目录内。

使用 `vaspkit-101-ST` 命令生成自洽计算所需要的 `INCAR` 文件。其中需要将截断能 `ENCUT` 设置为结构优化所使用的标准。修改后得到的 `INCAR` 文件如下所示：

Listing 5.4: INCAR

```
Global Parameters
ISTART = 1
```

³我也不知道为什么它是“TITEL”而不是“TITLE”，如果实在记不住的话用 `TIT` 也能搜索到对应内容。

```
LREAL = .FALSE.
ENCUT = 600
PREC = Accurate
LWAVE = .FALSE.
LCHARG = .TRUE.
ADDGRID= .TRUE.

Static Calculation
ISMear = 0
SIGMA = 0.05
LORBIT = 11
NEDOS = 2001
NELM = 60
EDIFF = 1E-08
```

其中需要特别注意的设置是：

- **LWAVE**: 表示写入波函数文件 **WAVECAR**，通常用于继续计算时的初始化设定。由于文件较大，因此如无必要，通常可以将其设定为 **.FALSE.** 表示不写入文件。
- **LCHARG**: 表示写入电荷密度文件 **CHGCAR**。在计算能带时，由于需要使用到这一文件，因此需要将其设定为 **.TRUE.**。
- **EDIFF**: 表示电子收敛标准。当能量变化达到这一标准时结束迭代计算。

将所有文件准备好后提交任务。

5.2.3 能带计算

相比于自洽计算，能带计算所需要的 K 点是特殊的高对称点路径，因此关键在于 KPOINTS 的生成。

将自洽计算得到的 **CHGCAR**, **POSCAR**, **INCAR**, **POTCAR** 全部复制（或移动）到一个新的目录下（假设为 **band**）。

使用 `vaspkit-3` 生成计算能带所用 `KPOINTS` 文件。其中需要根据结构特点选择是二维材料还是三维材料，在本例中由于 SiO_2 是二维材料，因此使用 `KPOINTS-303` 生成 `KPATH.in` 文件，将其命名为 `KPOINTS` 文件。生成后得到的文件如下所示。

Listing 5.5: KPOINTS

```
K-Path Generated by VASPKIT.
20
Line-Mode
Reciprocal
0.0000000000 0.0000000000 0.0000000000 GAMMA
0.0000000000 0.0000000000 0.5000000000 X

0.0000000000 0.0000000000 0.5000000000 X
0.2500000000 0.2500000000 0.2500000000 P

0.2500000000 0.2500000000 0.2500000000 P
0.0000000000 0.5000000000 0.0000000000 N

0.0000000000 0.5000000000 0.0000000000 N
0.0000000000 0.0000000000 0.0000000000 GAMMA

0.0000000000 0.0000000000 0.0000000000 GAMMA
0.5000000000 0.5000000000 -0.5000000000 M

0.5000000000 0.5000000000 -0.5000000000 M
0.3674577537 0.6325422463 -0.3674577537 S

-0.3674577537 0.3674577537 0.3674577537 S_0
0.0000000000 0.0000000000 0.0000000000 GAMMA

0.0000000000 0.0000000000 0.5000000000 X
-0.2349155075 0.2349155075 0.5000000000 R
```



```
0.5000000000 0.5000000000 -0.2349155075 G
0.5000000000 0.5000000000 -0.5000000000 M
```

其中每相邻两个点都是对应于一条高对称点路径。

对于 INCAR 文件，需要在自洽所使用文件的基础上，添加 ICHARG=11 表示读取当前目录下的 CHGCAR 文件，从而用于非自洽计算。

补充：在这里我们提到了自洽计算和非自洽计算，简单来说，自洽计算就是用来计算电子结构最稳定状态，而非自洽计算则是利用这一结构计算电子的其他性质（如能带、态密度等）。

将以上文件整理后提交任务计算。至此我们就已经完成了 VASP 计算能带的所有过程。

5.3 能带绘图与后处理

本节作者：Jiaqi Z.

在本节，你将要学到：

- 如何使用 Origin 绘制能带图
- 如何使用 vaspkit 自动生成能带图

在上一节的计算中，我们已经得到了能带所需要的全部信息。但是，使用计算软件得到的仅仅是数据，与我们所需要的“能带图”还差一个步骤—数据后处理。

在这一节，我们将讨论如何借助于 vaspkit 软件绘制能带图。其中，最基本的方法是使用 Origin 绘图，但在 vaspkit 的更新过程中，也添加了借助于 Python 脚本自动绘图的功能。我们将在本节首先介绍如何配置可自动绘图的 vaspkit，然后介绍如何使用 vaspkit 自动绘图。

注意：vaspkit 自动绘图并不一定在所有版本都可用，同时这依赖于服务器的配置是否有必要的包（如 Python, matplotlib 等）。因此，你不应当将 vaspkit 自动绘图作为依赖，而是作为一个“备选项”。

使用 Origin 绘图应当是绘制能带图首先需要掌握的内容。

5.3.1 使用 Origin 绘制能带图

使用 Origin 绘制能带图的第一步是得到能带图上每一点的坐标，借助于 vaspkit 我们可以很容易实现。利用 vaspkit-211 可以得到绘制能带图所需要的 BAND.dat 文件，其中包含了能带图的数据点坐标，下面则是文件开始的一部分：

Listing 5.6: BAND.dat

```
#K-Path(1/A) Energy-Level(eV)
# NKPTS & NBANDS: 180 64
# Band-Index 1
  0.00000 -19.370800
  0.04630 -19.368108
  0.09260 -19.360010
  0.13890 -19.346551
  0.18520 -19.327775
  0.23150 -19.303736
  0.27780 -19.274533
```

除此之外，我们还会得到 KLINE.dat 文件，用于生成高对称点坐标。将这两个文件下载到本地后，导入 Origin 软件当中。在 BAND.dat 文件后新添一列，并将 KLINE.dat 文件复制到新添的列中，得到的文件如图 5.1 所示。

将新添加的 C 列 (KLINE.dat 文件中的 x 坐标的属性设置为 “X” (方法：右键点击上方列名-“设置为”-X))

将这 4 列数据全选后点击菜单栏 “绘图” - “基础 2D 图” - “折线图” 即可得到所绘制的能带如图 5.2 所示。

调整线型与坐标范围，并在下方添加高对称点标签 (可以借助于 vaspkit 生成的 KLABELS 文件，其中包含所有高对称点坐标所对应的 x 轴坐标)

注意：一般来说，默认生成的能带图都会如图 5.2 所示包含较多的能带。但在实际研究中，我们往往仅关心费米能级附近 (即 0 点附近) 的情况。此时除了可以使用 Origin 调整坐标轴范围的方法，也可以在使用 VASP 计算自洽和能带的时候使用 NBANDS 参数，设置需要显示的能带数量。

在使用 vaspkit 自动生成能带图时，则需要对这一参数进行设置。



BAND - BAND.dat *				
	A(X)	B(Y)	C(Y)	D(Y)
长名称	#K-Path(1/A)	Energy-Level (eV)		
单位	#	NKPTS		
注释	#	Band-Index		
F(x)=				
迷你图				
1	0	-19.3708	0	-22.57932
2	0.0463	-19.36811	0.87971	-22.57932
3	0.0926	-19.36001	0.87971	38.6143
4	0.1389	-19.34655	0.87971	-22.57932
5	0.1852	-19.32778	1.30609	-22.57932
6	0.2315	-19.30374	1.30609	38.6143
7	0.2778	-19.27453	1.30609	-22.57932
8	0.3241	-19.24027	1.92814	-22.57932
9	0.3704	-19.20107	1.92814	38.6143
10	0.4167	-19.15707	1.92814	-22.57932
11	0.46301	-19.10844	2.68229	-22.57932
12	0.50931	-19.05537	2.68229	38.6143
13	0.55561	-18.99807	2.68229	-22.57932
14	0.60191	-18.93675	3.53505	-22.57932
15	0.64821	-18.87169	3.53505	38.6143
16	0.69451	-18.80316	3.53505	-22.57932
17	0.74081	-18.73148	3.86484	-22.57932
18	0.78711	-18.657	3.86484	38.6143
19	0.83341	-18.58005	3.86484	-22.57932
20	0.87971	-18.50105	4.77915	-22.57932
21	0.87971	-18.50105	4.77915	38.6143
22	0.90215	-18.50105	4.77915	-22.57932
23	0.92459	-18.50104	5.19246	-22.57932
24	0.94703	-18.50102	5.19246	38.6143
25	0.96947	-18.501	5.19246	-22.57932
26	0.99192	-18.501	5.65886	-22.57932
27	1.01436	-18.50096	5.65886	38.6143
28	1.0368	-18.50094	0	38.6143
29	1.05924	-18.50092	0	-22.57932
30	1.08168	-18.50089	0	0
31	1.10412	-18.50086	5.65886	0
32	1.12656	-18.50083		
33	1.149	-18.5008		
34	1.17144	-18.50078		

图 5.1: BAND.dat 修改后

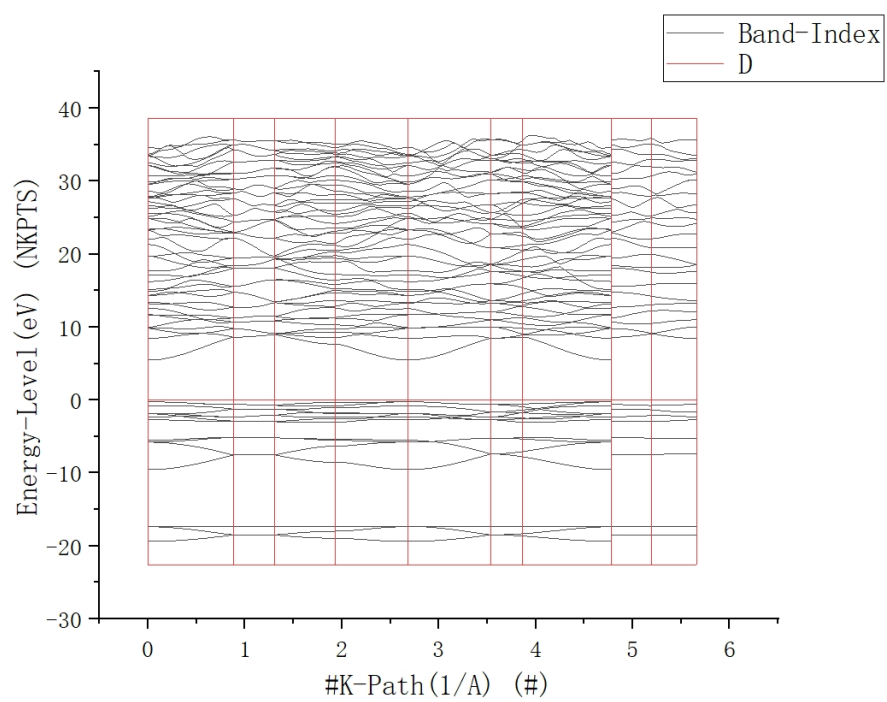
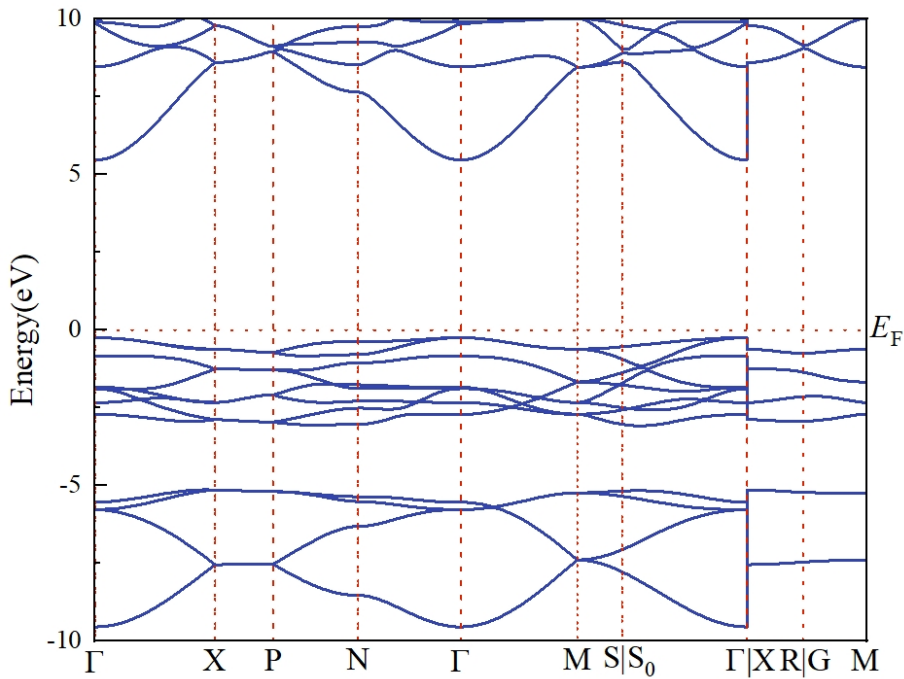


图 5.2: Origin 生成的能带图

图 5.3: SiO₂能带图

处理后得到的能带图如图 5.3所示⁴

5.3.2 使用 vaspkit 自动生成能带图

自 1.2.5 版本之后, vaspkit 更新了自动绘图的功能。利用这一功能, 可以不将数据下载至本地后使用 Origin 绘图, 而是直接在 Linux 操作系统中得到能带图像。

使用 vaspkit 自动绘图的过程是比较简单的, 但我们需要提前对 vaspkit 做一些配置上的设置。

vaspkit 配置过程

注意: 请首先检查你所使用的 *vaspkit* 是否为 1.2.5 或更新版本。你可以直接使用 *vaspkit* 命令, 在菜单栏上方, 可以查看所使用的软件版本。

⁴绘图配色可以根据自己喜好选择。

同时, 你还应当确认你的系统上已经配置了 *Python* 相关环境, 以及绘图所必须的 *matplotlib* 包。通常, 使用 *Anaconda* 可以“一次性”完成 *Python* 所需要的所有配置。确认是否安装 *Anaconda* 的一个方法是使用命令 `conda --version`, 若输出版本号, 则表明已经配置了 *Anaconda*⁵。

如果存在没有的命令或模块, 可能需要重新安装。详细安装过程请查阅对应软件官网⁶⁷的说明。

假设你的系统已经确认可以配置, 在开始之前还需要做如下操作:

1. 使用 `which python3` 查看 **python3 所在目录**。你需要记住这一路径, 将其首先复制到本地的记事本或其他地方是一个好方法;
2. 使用 `which vaspkit` 查看 **vaspkit 所在目录**, 并使用 `cd` 命令进入这一目录 (通常只需要进入到版本号所在目录即可, 例如, 在我所在课题组当中, 目录为 `/opt/pub/softwares/VASPKIT/1.5.1`);
3. 在这一目录下, 找到 `how_to_set_environment_variables` 文件, 并将其中从 `#BEGIN_CUSTOMIZE_PLOT` 到 `#END_CUSTOMIZE_PLOT` 之间的所有内容 (包括这两行) 复制到某个地方, 以便稍后使用;
4. 回到所在家目录, 新建一个 `.vaspkit` 文件, 并在其中创建如下两行:

Listing 5.7: `.vaspkit`

```
PYTHON_BIN  /opt/pub/toolkits/anaconda3/bin/python3
AUTO_PLOT   .TRUE.
```

其中, `PYTHON_BIN` 后面对应的是之前所复制的 `python3` 所在目录。然后, 在文件下方将所复制的 `#BEGIN_CUSTOMIZE_PLOT` 到 `#END_CUSTOMIZE_PLOT` 之间的所有内容粘贴至后面。

使用 `vaspkit` 绘图

在使用 `vaspkit` 绘图前, 应当首先保证你已经导入了相应模块 (如 *Anaconda* 等)。与正常使用 `vaspkit` 导出能带图类似, 在使用 `vaspkit-211` 导

⁵在有些服务器上, 可能需要其他设置引入 *Anaconda* 模块。例如, 在我所在课题组的服务器上, 使用之前需要调用 `module load anaconda3` 导入相关模块。

⁶VASPKIT 官网: <https://vaspkit.com/>

⁷Anaconda 官网: <https://www.anaconda.com/>

出时，会询问导出图片是仅导出能带图，还是能带图加态密度。我们在这里选择仅绘制能带图（1），得到如图所示的图像。

补充：正如你所见那样，使用 *vaspkit* 默认生成的能带图是很大的能量范围，且不能对其进行调整。一个简单的方法是在计算时使用 *EMIN*和 *EMAX*参数设置能量区间。

5.3.3 如何计算带隙

有时，我们可能不是强制要求需要得到能带图，而仅仅关注结构的带隙性质（如带隙大小，类型等）。此时，可以直接使用在绘图过程中生成的 *BAND_GAP* 文件（使用 *vaspkit-211* 生成）。以 SiO_2 为例，文件内容为：

Listing 5.8: *BAND_GAP*

```
Band Character: Direct
Band Gap (eV): 5.7105
Eigenvalue of VBM (eV): -0.6699
Eigenvalue of CBM (eV): 5.0407
Fermi Energy (eV): -0.4270
HOMO & LUMO Bands:      16      17
Location of VBM: -0.000000 0.000000 0.000000
Location of CBM: -0.000000 0.000000 0.000000
```

其中，*Band Character* 表明带隙类型是直接带隙（Direct）或间接带隙（Indirect）；*Band Gap* 表明所计算结构的带隙大小，在本例中为 5.71 eV。除此之外，这一文件还提供了如 HOMO（最高占据分子轨道）和 LUMO（最低未占据分子轨道）所对应的能带条数，以及 VBM（价带顶）和 CBM（导带底）所对应的 K 点坐标。

注意：如果你关注过最开始 *Materials Project* 所记录的数据，可能会发现，数据库内所给的带隙约为 5.8 eV。而我们所计算得到的比实际带隙小了约 0.1 eV。正如 5.2 开始所说的那样，这一误差是由于 *PBE* 泛函所导致的，而这是 *PBE* 泛函的固有缺陷。在一些需要精确计算的情况下，我们可能希望计算 *HSE* 能带而不是 *PBE* 能带。

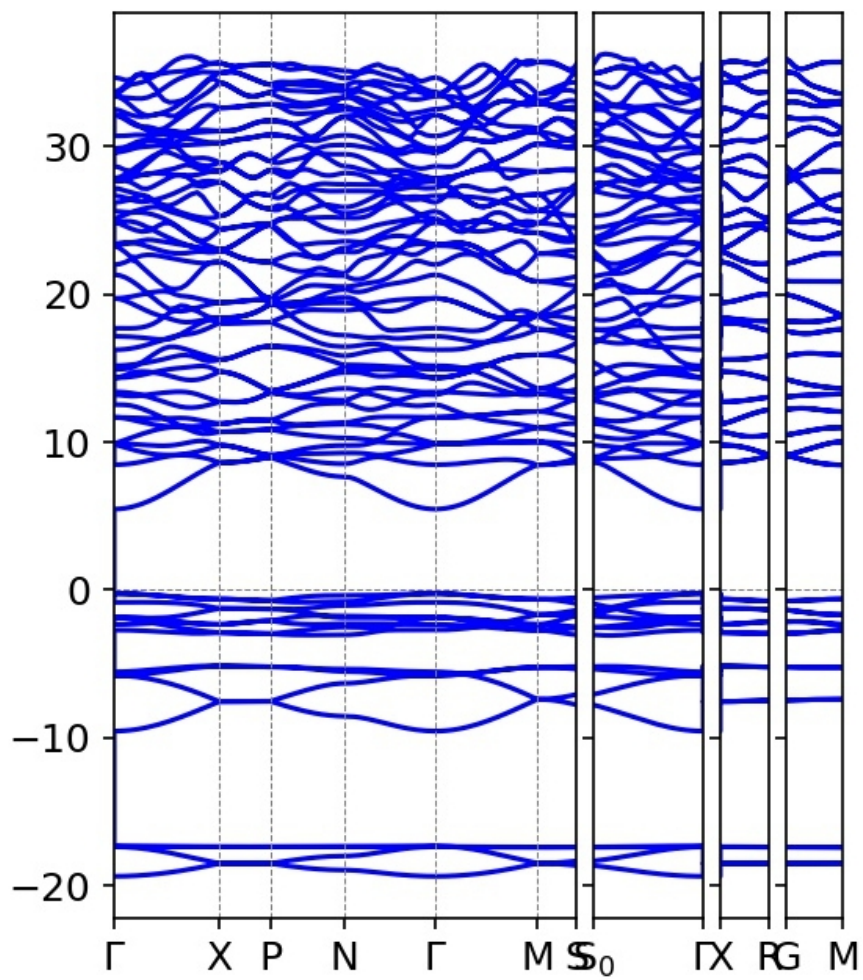


图 5.4: 使用 vaspkit 生成能带图

5.4 HSE 能带计算

本节作者: Jiaqi Z.

在本节, 你将要学到:

- 如何计算 HSE 能带

在之前介绍 PBE 能带计算时提到, 使用 **PBE** 计算会得到带隙偏小的情况, 而这对于高精度计算显然是不合适的。因此, 在大多数计算能带的论文中都需要其他泛函如 HSE 计算得到的能带。因此, 在本节我们将讨论如何使用 HSE 计算能带并给出更加准确的带隙。

5.4.1 结构优化与自洽计算

在本节我们将讨论的结构是金刚石, 其 POSCAR 文件如下所示:

Listing 5.9: POSCAR

```
C2
1.0
      2.5269944668      0.0000000000      0.0000000000
      1.2634972334      2.1884414035      0.0000000000
      1.2634972334      0.7294804678      2.0632823422
C
2
Direct
      0.7500000000      0.7500000000      0.7500000000
      0.5000000000      0.5000000000      0.5000000000
```

对于 HSE 泛函计算能带, 结构优化过程与前面 5.2.1 所介绍的步骤完全相同, 因此这里不再赘述。

对于自洽计算, KPOINTS 则需要使用 hse 的 K 点。生成方法可以使用 `vaspkit-3-302/303` 生成 `KPATH.in` 后使用 `vaspkit-25-251` 生成 KPOINTS, 其中可以根据需要选择 Monkhorst-Pack Scheme 或 Gamma Scheme 方法

生成 K 点⁸。然后需要依次输入 scf 所使用 k 点密度（通常设定为 0.04）以及计算能带所需要的 k 点密度（通常也设定为 0.04）

同时，在计算 scf 时还需要将 INCAR 的 LWAVE = .TRUE. 以生成波函数文件，以便计算能带时使用。下面是一个参考使用的 INCAR 文件：

Listing 5.10: INCAR

```
Global Parameters
ISTART = 1
ISPIN = 1
LREAL = .FALSE.
ENCUT = 600
PREC = Accurate
LWAVE = .TRUE.
LCHARG = .FALSE.
ADDGRID= .TRUE.

Static Calculation
ISMEAR = 0
SIGMA = 0.05
LORBIT = 11
NEDOS = 2001
NELM = 60
EDIFF = 1E-08
```

5.4.2 HSE 能带计算

上一步计算完成后，将 KPOINTS, POTCAR, POSCAR 和 WAVECAR 文件拷贝（或移动）至新的目录（暂且命名为 hse-band，使用 vaspkit-101-STH6 生成 HSE 计算的 INCAR 文件，调整必要的参数后如下所示：

Listing 5.11: INCAR

⁸一般来说，使用 Monkhorst-Pack 生成 K 点精度较高，但对于六方晶系，则需要选择 Gamma 方法。

```
Global Parameters
ISTART = 1
ISPIN = 1
LREAL = .FALSE.
ENCUT = 600
PREC = Accurate
LWAVE = .FALSE.
LCHARG = .FALSE.
ADDGRID= .TRUE.

Static Calculation
ISMEAR = 0
SIGMA = 0.05
LORBIT = 11
NEDOS = 2001
NELM = 60
EDIFF = 1E-08

HSE06 Calculation
LHFCALC= .TRUE.
AEXX = 0.25
HFSCREEN= 0.2
ALGO = ALL
TIME = 0.4
PRECFOCK= N
```

这里面的关键参数是 **AEXX**, 表示杂化泛函所占的比重, 通常设定为 0.25 是比较合适的。但**这一比值会影响到带隙的大小**, 因此在必要的时候需要进行调整以适应实验结果。

将上述文件提交计算, 由于 HSE 能带计算精度较高, 因此需要时间较长⁹。

⁹在测试计算时, 64 核计算时间为 7.7 个小时。

5.4.3 HSE 能带计算后处理

与 5.3 一节所介绍的类似，在 HSE 计算后也需要对数据进行处理，绘图。但在一些细节上有些许不同：

- 在绘制能带图之前，需要确保当前目录下有 `KPATH.in` 文件（可以直接使用 `vaspkit-3-302/303` 的方式生成，也可以直接从之前 `scf` 目录下复制过来；
- 使用 `vaspkit-25-252` 生成能带数据，所得到的 `BAND.dat` 和 `KLINE.dat` 文件可以将其放入 Origin 等软件中绘图（方法参考 5.3 一节，在此略）

生成能带数据后，可以通过 `BAND_GAP` 文件查看计算得到的带隙。在本地测试时，得到结果为 5.29 eV，与文献¹⁰结果 5.38 eV 接近。

同时，我们也可以利用 5.2 所介绍的 PBE 能带计算方法，对金刚石的 PBE 带隙进行计算。结果发现其带隙为 4.12 eV，显然小于 HSE 带隙。

¹⁰Stoliaroff, A. & Latouche, C. Accurate ab initio calculations on various PV-based materials: Which functional to be used? The Journal of Physical Chemistry C 124, 8467–8478 (2020).

Chapter 6

声子谱计算

Contents

6.1	什么是声子、声子谱	142
6.1.1	声子	142
6.1.2	声子谱	143
6.2	计算方法简介	143
6.2.1	密度泛函微扰理论 (DFPT)	143
6.2.2	有限位移法 (Finite Displacement Method)	144
6.2.3	适用情境比较	144
6.3	计算软件 PHONOPY	145
6.3.1	开始安装之前	145
6.3.2	PHONOPY 快速安装	146
6.3.3	PHONOPY 使用方法	146
6.4	具体计算步骤	147
6.4.1	计算声子谱前的结构优化	147
6.4.2	DFPT 方法计算声子谱	149
6.4.3	有限位移法计算声子谱	152
6.5	声子谱分析	154
6.6	错误处理	154

6.1 什么是声子、声子谱

本节作者: Isay K.

在本节, 你将要学到:

- 声子
- 声子谱

6.1.1 声子

声子 (Phonon), 即 “晶格振动的简正模能量量子”, 是晶体中原子振动的量子化描述。

在固体物理学中, 声子是晶格振动的准粒子, 其携带能量和动量, 并且可以像粒子一样进行相互作用。

声子是简谐近似下的产物, 如果振动太剧烈, 超过小振动的范围, 那么晶格振动就要用非简谐振动理论描述。

声子并不是一个真正的粒子, 声子可以产生和湮灭, 有相互作用的声子数不守恒, 声子动量的守恒律也不同于一般的粒子, 并且声子不能脱离固体存在。声子只是格波激发的量子, 在多体理论中称为集体振荡的元激发或准粒子。

声子的化学势为零, 属于玻色子, 服从玻色-爱因斯坦统计。声子本身并不具有物理动量, 但是携带有准动量, 并具有能量, 它的能量等于 $\hbar\omega_q$ 。

声子可以分为以下两类:

- 声学支: 与晶格的纵向和横向振动相关, 类似于声波, 表示原胞的整体振动。
- 光学支: 与晶格的非均匀振动相关, 通常与电荷的重新分布有关, 表示原胞内原子间的相互振动。

如果一个材料的原胞中有 N 个原子, 那么声子谱就会有 $3N$ 支, 其中 3 条声学支, $3N - 3$ 条光学支。

6.1.2 声子谱

声子谱，也称为声子色散关系，是描述声子能量与动量之间关系的图表。

声子谱通常在第一布里渊区内绘制，因为其包含了所有可能的声子模式。

通常，使用声子谱研究体系的动力学稳定性，使用分子动力学研究体系的热力学稳定性。

声子谱的其他物理意义：

- 电子-声子耦合：在半导体和超导体中，电子-声子耦合相互作用对材料的电子性质至关重要；
- 声子散射：在金属和半导体中，声子散射是影响电子迁移率的关键因素；
- 热容：声子谱可以解释材料在不同温度下的热容行为
- ...

6.2 计算方法简介

本节作者：Isay K.

在本节，你将要学到：

- 密度泛函微扰理论（DFPT）
- 有限位移法（Finite Displacement Method）
- 适用情境比较

6.2.1 密度泛函微扰理论（DFPT）

DFPT 是一种基于第一性原理的方法，它直接从周期性边界条件的 Kohn-Sham 波函数计算出声子谱。在 DFPT 中，通过计算原子间相互作用的微扰来得到力常数矩阵，这是描述晶格动力学性质的关键量。

补充：1987 年，Baroni、Giannozzi 和 Testa 提出了一种新的晶格动力学性质计算方法—微扰密度泛函方法 (*Density Function Perturbation Theory*)。

DFPT 通过计算系统能量对外场微扰的响应来求出晶格动力学性质。该方法最大的优势在于它不限定微扰的波矢与原胞边界 (*super size*) 正交, 不需要超原胞也可以对任意波矢求解。因此可以应用到复杂材料性质的计算上。此外, 能量对外场微扰的响应不仅可以推导出声子的晶体性质, 还能求出弹性系数、声子展宽、拉曼散射截面等性质, 这种方法本身就能算出 *Born effective charge dielectric constant*, 可以很好的预言 *LO-TO splitting* 甚至 *Kohn anomalies*。这些优势使得 DFPT 一经提出就被广泛应用到了半导体、金属和合金、超导体等材料的计算上。比较常用的程序是 *pwscf* 和 *abinit*, *castep* 等采用的是一种 *linear response theory* 的方法 (或者称为 *density perturbation functional theory*, DFPT), 直接计算出原子的移动而导致的势场变化, 再进一步构造出动力学矩阵。

6.2.2 有限位移法 (Finite Displacement Method)

有限位移法通过在超原胞中引入原子的有限位移来模拟晶格振动。这种方法基于位移-响应理论, 通过计算原子位移后系统的受力来构造动力学矩阵

补充: 直接法, 或称 *frozen-phonon* 方法, 是通过在优化后的平衡结构中引入原子位移, 计算作用在原子上的 *Hellmann-Feynman* 力, 进而由动力学矩阵算出声子色散曲线。用该方法计算声子色散曲线最早开始于 80 年代初, 由于计算简便, 不需要特别编写的计算程序, 很多小组都采用直接法计算材料性质。直接法的缺陷在于它要求声子波矢与原胞边界 (*super size*) 正交, 或者原胞足够大使得 *Hellmann-Feynman* 力在原胞外可以忽略不计。这使得对于复杂系统, 如对称性高的晶体、合金、超晶格等材料需要采用超原胞。超原胞的采用使计算量急剧增加, 极大的限制了该方法的使用。这种方法不能很好的预言 *LO-TO splitting*, 只有在计算了 *Born effective charge* 和 *dielectric constant* 之后, 进一步考虑了 *non-analyticity term*, 才能计算出; 但 *Direct Method* 本身并不能给出 *Born effective charge* 和 *dielectric constant*, 所以这也是它的一个缺陷。

6.2.3 适用情境比较

DFPT 适用情境:

- 需要高精度声子谱的系统, 尤其是小到中等大小的晶胞;

- 研究者希望避免有限位移法可能引入的系统误差时。

注意：DFPT 方法计算成本较高，尤其对于大晶胞或高对称点附近的计算。

有限声子法适用情境：

- 当计算资源有限或需要对多种材料进行筛选时；
- 对于大晶胞材料的初步声子谱分析。

总得来说，对于较重的任务，DFPT 方法可能会造成内存溢出，且 DFPT 方法由于其特性而无法进行并行计算，而有限声子法可以并行。对于较小的体系，可以根据需要和组内资源选择方法。

补充：建议优先使用有限位移法。

一些教程中有时会将有限位移法又称为冷冻声子法或直接法。但笔者并没有找到更官方的资料说明有限位移法和冷冻声子法是同一种方法，谨奉上 PHONOPY 官网供读者自行分辨：<https://phonopy.github.io/phonopy/index.html>

参考：<http://muchong.com/html/200802/723527.html>

6.3 计算软件 PHONOPY

本节作者：Isay K.

在本节，你将要学到：在本节，你将要学到：

- 开始安装之前
- PHONOPY 快速安装
- PHONOPY 使用方法

6.3.1 开始安装之前

由于本教程面向的群体是计算小白（包括笔者也是通过本教程记录一下自己掉的坑），所以在开始安装软件之前，我们强烈建议先咨询组内老师或师兄师姐：服务器上是否已经配置了相应的软件？

补充：当然也可以使用 `module avail` 命令自己检查系统内已安装的软件，如果没有找到的话再咨询更有自主性哦。

```
(phonopy) [lj@master relax]$ module avail
-----
VASP/5.4.4-gbuid  VASP/6.3.2-gbuid-intel_8380 (D)  VASPKIT/1.4.1  Wannier90/2.1.0  Wannier90/3.1.0 (D)  bader/1.04  conda (L)  gnu  intel  lobster/4.1.0  vtstscripts/2033
-----
Enyghid/4.6.2  charliecloud/0.15  gnu2/2.2.0  intel-omp/2023.1.0  os  pandas/4.2.1  setx/1.11.2
autotools  cmake/3.24.2  huioc/2.7.0  libfabric/1.13.0  popl/6.0.0  prun/2.2  valgrind/3.19.0
-----
Where:
L: Module is loaded
D: Default Module
If the avail list is too long consider trying:
"module --default avail" or "ml -d av" to just list the default modules.
"module overview" or "ml ov" to display the number of modules for each name.
Use "module spider" to find all possible modules and extensions.
Use "module keyword key1 key2 ..." to search for all possible modules matching any of the "keys".
```

图 6.1: 检查是否含有所需软件

通常情况下，组内服务器的根目录下已经配置了相应的软件，这个时候再在自己的用户目录下进行配置的话，一方面在使用过程中可能会出现命令的冲突，另一方面也是一种时间、精力和资源的浪费。

如果组内确实并没有安装，或者你是传说中的开山大弟子，又或者是自学，请放心进入 6.3.2 小节。

6.3.2 PHONOPY 快速安装

参考:https://blog.csdn.net/qq_41866202/article/details/124407208?spm=1001.2101.3001.666task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ECtr-1-124407208-blog-139391379.235%5Ev43%5Epc_blog_bottom_relevance_base9&depth_1-utm_source=distribute.task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ECtr-1-124407208-blog-139391379.235%5Ev43%5Epc_blog_bottom_relevance_base9&utm_relevant_index=1

6.3.3 PHONOPY 使用方法

加载 PHONOPY 环境

- 1. 加载 Anaconda 应用: `module load conda`
- 2. 激活 PHONOPY 环境: `conda activate PHONOPY`

命令详解

进入下面的官网之后点击 Command options 即可看到所有功能。
<https://phonopy.github.io/phonopy/index.html>
注意：在进行扩胞时的标准是：扩胞后的原子达到 80~100 个，每个晶轴方向大于 10，不然得到的声子谱中很容易出现虚频。

6.4 具体计算步骤

本节作者：Isay K.

在本节，你将要学到：

- 计算声子谱前的结构优化
- DFPT 方法计算声子谱
- 有限位移法计算声子谱

6.4.1 计算声子谱前的结构优化

注意：在计算声子时需要先对原胞结构做高精度的结构优化，不然得到的声子谱中很容易出现虚频。

我们以 TiTe_2 为例，以下是高精度优化的具体参数。

Listing 6.1: INCAR

```
Global Parameters
ISTART = 1
ISPIN = 1
LREAL = .FALSE.
ENCUT = 380

LWAVE = .FALSE.
LCHARG = .FALSE.
ADDGRID= .TRUE.
LASPH = .TRUE.
PREC = Accurate
NCORE = 8
ISYM = 0

Lattice Relaxation
NSW = 300
ISMear = 0
```

```
SIGMA = 0.03  
IBRION = 2  
ISIF = 3  
IOPTCELL = 1 0 0 1 1 0 0 0 0  
EDIFF = 1E-08  
EDIFFG = -1E-03
```

为了保证优化精度足够高，其中需要注意的是：

1. EDIFF表示电子收敛标准，至少要取 $1\text{E-}06$ ，体系小的话尽量取 $1\text{E-}08$ ；
2. EDIFFG取负值时表示力收敛标准，取 $1\text{E-}03$ ；
3. ADDGRID表示是否添加额外网格提高精度，设定为`.TRUE.`；
4. PREC表示“精度”模式，设定为 `Accurate`（准确）；
5. NSW表示电子优化步数，取 300 防止计算中断；
6. ENCUT可以自行做测试，详见 VASP 计算-结构优化章节（先别去找，我没写）；

另外，其中 `ISIF=3` 表示既优化晶格又优化原子坐标，配合 `IOPTCELL` 可以实现晶轴的单独固定，以达到计算二维材料的目的，详见 VASP 计算-结构优化章节（也还没写）。

下面的其他输入文件没有需要特别说明的，如有疑问请参考 VASP 计算-结构优化章节（哈哈，又是这）或参考 VASP 官网：https://www.vasp.at/wiki/index.php/The_VASP_input_files

Listing 6.2: KPOINTS

```
A  
0  
Gamma  
24 24 1  
0.0 0.0 0.0
```

Listing 6.3: POSCAR

```

TiTe2-1m1
1.0000000000000000
  3.7458432095936396 -0.0000184453725456 0.0000000000000000
 -1.8729216047968198 3.2439861579338527 0.0000000000000000
  0.0000000000000000 0.0000000000000000 18.0000000000000000
Ti   Te
  1     2
Direct
0.0000000000000000 0.0000000000000000 0.5127400160000022
0.6666666132020026 0.3333334158033583 0.6098496477195335
0.3333334157979960 0.6666666131966403 0.4156403382804701

0.00000000E+00 0.00000000E+00 0.00000000E+00
0.00000000E+00 0.00000000E+00 0.00000000E+00
0.00000000E+00 0.00000000E+00 0.00000000E+00

```

提交任务进行计算，得到 `CONTCAR` 为优化后的更合理的结构，作为后续声子计算的初始晶胞。（后续小节中提到“初始晶胞”均指优化后得到的晶胞，为避免歧义在此说明。）

6.4.2 DFPT 方法计算声子谱

```
1.mkdir method_DFPT
```

新建文件夹。

```
2.cp relax/CONTCAR method_DFPT/POSCAR
```

将上一步高精度结构优化得到的 `CONTCAR` 复制进文件夹内，并重命名为 `POSCAR`。

```
3.cd method_DFPT
```

进入新文件夹。

```
4.module load conda conda activate phonopy
```

加载 `conda` 模块，并激活 `phonopy` 环境，详情可参考 6.3.3

```
5.phonopy -d --dim="6 6 1"
```

使用 PHONOPY 进行 6×6 的扩胞。

此时会产生数个名为 POSCAR-0? 的位移文件，以及名为 SPOSCAR 的扩胞后的结构。

DFPT 方法使用的是 SPOSCAR，而有限位移法使用的是这些位移文件。

注意：笔者研究的是 2D 结构，仅对两个方向进行扩胞，读者可根据需要自行调整。扩胞的标准是扩胞后达到 80~100 个原子，且晶轴长度大于 10 埃，不然得到的声子谱中很容易出现虚频。

6.mkdir vasp-calculations

新建文件夹用于后续计算。

补充：此处根据个人习惯不同，也可以不新建文件夹。将 POSCAR 重命名为 POSCAR-unit，将第 5 步新产生的 SPOSCAR 重命名为 POSCAR，直接在当前文件夹中进行计算。

7.cp SPOSCAR vasp-calculations/POSCAR

将第 5 步新产生的 SPOSCAR 复制进文件夹，并重命名为 POSCAR。

8. 准备其它基本文件：

Listing 6.4: INCAR

```
SYSTEM = TiTe2
#ISIF = 3
NSW = 1
IBRION = 8

LWAVE = F
LCHARG = F

ENCUT = 380
EDIFF = 1E-8
EDIFFG = -1E-3
ISMEAR = 0

LREAL = F
SIGMA = 0.03
```

```
PREC = A
ADDGRID = .TRUE.
```

Listing 6.5: KPOINTS

```
A
0
Gamma
3 3 1
0.0 0.0 0.0
```

注意：因为该计算使用的是扩胞之后的结构，所以 K 点没有必要取太大。

9.sbatch sub.vasp

提交任务进行计算。

10.cd ..

返回 method_DFPT 文件夹。

11.cp vasp-calculations/vasprun.xml .

将 vasprun.xml 复制到当前文件夹。

12.phonopy --fc vasprun.xml

使用 phonopy 读取 vasprun.xml 生成力常数文件 FORCE_CONSTANTS。

13.vi band.conf

编辑 band.conf 文件：

Listing 6.6: band.conf

```
ATOM_NAME =Ti Te
DIM = 6 6 1
BAND =0 0 0 0.5 0 0 0.33333 0.33333 0 0 0 0
BAND_POINTS = 101
FORCE_CONSTANTS = READ
```

1. DIM 根据体系的扩胞大小设置，如扩胞扩到 332，就设置成 332。
2. BAND 和能带的取点是一样的，也可以用 vaspkit 生成。
3. FORCE_CONSTANTS 一定设置成 READ。
4. 更多设置可以看 PHONOPY 官网。

```
14.phonopy -p -s band.conf
```

使用 phonopy 读取 band.conf 文件，作图并保存。

```
15.phonopy-bandplot --gnuplot > phonon.out
```

将数据导出方便后续用 Origin 等软件重新绘图。

补充:旧版本的 *phonopy* 的导出命令为:*bandplot --gnuplot> phonon.out*

6.4.3 有限位移法计算声子谱

```
1.mkdir method_yxwy
```

```
2.cp relax/CONTCAR method_yxwy/POSCAR
```

```
3.cd method_yxwy
```

```
4.phonopy -d --dim="6 6 1"
```

前四步和 DFPT 法完全相同。

```
5.for i in 01..12; do mkdir $i; cp POSCAR-0$i $i/POSCAR;done
```

假如第四步产生了 12 个位移文件，使用 for 循环生成 12 个文件夹，并将对应的位移 POSCAR 移入文件夹重命名为 POSCAR。

6. 准备其它基本文件：

Listing 6.7: INCAR

```
ADDGRID = .TRUE.
PREC = Accurate
IBRION = -1
ENCUT = 380
EDIFF = 1E-8
EDIFFG = -1E-3

ISMEAR = 0
```



```

SIGMA = 0.03

IALGO = 38

LREAL = .FALSE.
LWAVE = .FALSE.
LCHARG = .FALSE.

NCORE = 4

```

注意：有限位移法的单个计算实际上就是高精度的静态自洽。

Listing 6.8: KPOINTS

```

Automatic mesh
0
Gamma
3 3 1
0 0 0

```

注意：同 *DFPT* 法一样，有限位移法使用的也是扩胞之后的结构，所以 *K* 点没有必要取太大。

```

7.for i in 01..12; do cp INCAR KPOINTS POTCAR sub.vasp $i; cd
$i; sbatch sub.vasp; cd $OLDPWD;done

```

将基本文件复制进各个小文件夹中并进行计算。

```

8.phonopy -f 01..12/vasprun.xml

```

计算全部结束后,使用 phonopy 读取全部的计算文件夹中的 `vasprun.xml`, 生成 `FORCE_SETS` 文件。

```

9.vi band.conf

```

Listing 6.9: band.conf

```

ATOM_NAME =Ti Te
DIM = 6 6 1

```

```
BAND =0 0 0 0.5 0 0 0.33333 0.33333 0 0 0 0  
BAND_POINTS = 101  
FORCE_SETS = READ
```

与 DFPT 方法唯一不同的部分在于将 `FORCE_CONSTANTS=READ` 改成 `FORCE_SETS=READ`。

```
10.phonopy -p -s band.conf
```

使用 phonopy 读取 `band.conf` 文件，作图并保存。

```
11.phonopy-bandplot --gnuplot > phonon.out
```

将数据导出方便后续用 Origin 等软件重新绘图。

6.5 声子谱分析

本节作者：Isay K.

我还不太会，等我学学。

<http://pubs.acs.org/doi/abs/10.1021/acs.jpcc.5b04669>

6.6 错误处理

`vasprun.xml` 没有必要信息

Part III

Python 与机器学习

索引

*, 46

?, 46

\$, 68

AEXX, 139

alias, 35

bc, 70

case, 88

cat, 17

cat -b, 17

cat -n, 17

cat -s, 17

cd, 9

chmod, 27

cp, 13

cp -r, 13

echo, 57, 64

EDIFF, 127

EDIFFG, 125

else, 82

EMAX, 135

EMIN, 135

ENCUT, 125

export, 65

fi, 77

for, 56

grep, 49

-i, 49

-l, 49

-n, 49

-r, 49

-v, 49

head, 19

head -c, 19

head -n, 19

if, 76

INCAR

ADDGRID, 148

EDIFF, 148

EDIFFG, 148

ENCUT, 148

IOPTCELL, 148

ISIF, 148

NSW, 148

PREC, 148

ISIF, 125

LCHARG, 127

less, 18, 23

ll, 8

ln, 16

ln -s, 16

ls, 5, 7

ls -a, 8

ls -l, 7, 26, 27

LWAVE, 127

mkdir, 9

mkfifo, 16

more, 18

mv, 12

nano, 32

Alt+6, 33

Alt+A, 34

Alt+D, 34

Alt+E, 34

Alt+U, 34

Alt+↑, 34

Alt+↓, 34

Ctrl+\\, 34

Ctrl+G, 34

Ctrl+K, 33

Ctrl+O, 33

Ctrl+U, 33

Ctrl+W, 34

Ctrl+X, 33

NSW, 125

OPTCELL, 125

pwd, 57

read, 72

-p, 72

rm, 13

rm -i, 13

rm -r, 13

rmdir, 13

sbatch, 119

scancel [jobID], 119

sed, 50

-i, 51

-n, 51

seq, 58

source, 64

squeue, 119

tac, 17

tail, 19

tail -c, 19

tail -n, 19

tar, 21, 22

tar -A, 24

tar -a, 24

tar -c, 22

tar -f, 22

tar -r, 24

tar -t, 23

tar -v, 23

tar -x, 22

tar -z, 22

useradd, 25

useradd -d, 25

- vi, 35
 - /, 39
 - :q, 37
 - :w, 37
 - :wq, 37
 - :x, 37
 - ?, 39
 - \$, 39
 - a, 36
 - dd, 36
 - i, 36
 - N, 39
 - n, 39
 - O, 36
 - o, 36
 - P, 36
 - p, 36
 - s, 39, 40
 - u, 36
 - x, 36
 - yy, 36
- vim, 35

参考文献