Fortran 笔记

GasinAn

2024 年 10 月 10 日

Printed in China

前言

天文系的师兄师姐师弟师妹们估计普遍对 Fortran 深恶痛绝. 然而我认为 Fortran 并不是那么可恶. 对 Fortran 深恶痛绝, 可能源于老师上课时对同学们的花式折磨.

学 Fortran 的时候, 老师可能从 FORTRAN 77 开始讲起, 然后就有一千种方法可以折磨人啦. 最常见的就是用 I–N 隐式规则来折磨人了. 比如搞两个变量 `m1` 和 `m2`, 代表两物体质量, 令 `m1 = 3.0`, `m2 = 2.0`, 然后算质量比. 没加类型声明语句, 直接除, 就成功被坑啦. 老师可能很喜欢摆出各种各样有坑的程序问同学们输出是什么, 但实际上现代 Fortran 程序设计是在极力避免这些坑人的特性发挥作用, 比如, 直接在程序一开始加上 `implicit none`, 禁用 I–N 隐式规则, 以避免各种麻烦出现. 竭尽全力地训练同学们分析各种坑人程序, 在实际的程序设计中并没有什么直接的好处.

当然, 老师让同学们分析这些坑人程序也不能说没用. 老师可能觉着, 分析这些坑人程序能让同学们对程序本身的运作过程有更清楚完整的认识, 这我难以直接否定. 然而简单地这么做, 很可能导致同学们觉得 “Fortran 就是这么折磨人”, 不好用, 于是上完课就怒卸 Fortran 编译器, 再也不玩儿了. 其实, 坑人的不是 Fortran, 坑人的是老师. 以我的经验, 只要严格遵守一些规则, Fortran 还是好使的. 当然, 一般情况下肯定不会比 Python 好使.

这份笔记旨在完整讲解天文学研究中会用到的全部 Fortran 相关知识, 让同学们在读完这份笔记后能快乐地玩 Fortran!

目录

前言 i

第一章 简介 1

第二章 编译器 3

2.1 安装 3

2.1.1 Intel® Fortran Compiler . . . 3

2.1.2 GNU Fortran Compiler . . . 4

2.2 使用 6

2.2.1 使用 CLI . . . 6

2.2.2 使用 GUI . . . 10

第三章 基础知识 13

3.1 程序 13

3.2 标准与规范 13

3.3 程序单元 15

3.4 编译与运行 16

3.5 异常 18

3.5.1 错误 18

3.5.2 警告 19

3.6 字符集 21

3.7 源代码格式 22

3.7.1 空格 23

3.7.2 缩进 23

3.7.3 空行 24

3.7.4 注释 26

3.7.5 续行 28

3.7.6 标号 31

3.7.7 标签 32

第四章 固有类型 35

4.1 整型 38

4.2 实型 39

4.3 复型 40

4.4 字符型 42

4.5 逻辑型 45

第五章 赋值与运算 47

5.1 赋值 47

5.2 运算 56

5.2.1 算数运算 . . . 58

5.2.2 字符运算 . . . 62

5.2.3 逻辑运算 . . . 63

5.2.4 关系运算 . . . 63

第六章 执行控制 67

6.1 结构 67

6.1.1 if 结构 67

6.1.2 do 结构 77

6.1.3 do while 结构 . 83

6.2 执行控制语句 84

6.2.1 continue 语句 . 84

6.2.2 exit 语句 . . . 85

6.2.3 cycle 语句 . . . 87

6.2.4 stop 语句 . . . 88
6.2.5 error stop 语句 89
6.2.6 return 语句 . . 90

第七章 数组 91
7.1 数组基础 91
7.2 数组构造 94
7.3 数组声明 95
7.4 数组操作 97
7.5 数组赋值 103
7.5.1 where 结构 . . 107
7.5.2 do concurrent 结构 111
7.6 数组运算 112

第八章 过程 115
8.1 外部过程 118
8.1.1 子例行子程序 . 118
8.1.2 函数子程序 . . 125
8.1.3 固有过程 . . . 128
8.2 过程中的变量 128
8.2.1 save 属性 . . . 128
8.2.2 过程中的字符串 130
8.2.3 过程中的数组 . 131
8.2.4 哑过程 136
8.3 特殊过程 138
8.3.1 纯过程 138
8.3.2 逐元过程 . . . 139
8.3.3 递归过程 . . . 140
8.4 过程接口 144
8.4.1 特定接口 . . . 144
8.4.2 泛型接口 . . . 150
8.4.3 抽象接口 . . . 154

第九章 输入与输出 159
9.1 文件 161
9.2 读取与写入 164
9.3 编辑符 169
9.3.1 数据编辑符 . . 173
9.3.2 控制编辑符 . . 187
9.3.3 字符串编辑符 . 190

附录 A 绘图 193

第一章　简介

Fortran 是一门历史悠久的, 专为科学计算设计的编程语言.
Fortran 的优点[1]有:

- Fortran 是一种相对较小的语言, 令人惊讶地易于学习和使用. 在大型数组上表达大多数数学和算术运算, 就像在白板上写方程一样简单.
  安安锐评: 这点天文系的师兄师姐师弟师妹们估计统统会反对, 然而这件事其实是真的. 觉得 Fortran 超级无敌巨 TM 难学, 恐怕全是老师花式折磨的结果. 安安自己当年上课的时候其实也是啥也没学会, 后来努力自学 Modern Fortran, 发现其实 Modern Fortran 在 Fortran 官方的努力下, 已经基本上没啥坑了, 自己在本科毕设的时候也是完全使用 Modern Fortran 来进行计算[2], 基本上没遇到什么困难.

- Fortran 语法严格, 使得编译器可以在早期捕捉许多编程错误[3], 也使得编译器能够生成高效的二进制代码.

- Fortran 是 "多范式" 的, 允许以最适合问题的编程风格来书写代码: 命令式, 函数式, 面向对象式皆可.[4]

- Fortran 是一种原生的并行编程语言, 具有直观的类似数组的语法[5], 可以在 CPU 之间进行数据通信, 可以在单个 CPU, 共享内存多核系统或分布式内存 HPC 或基于云的系统上运行几乎相同的代码.

[1] 这些是 Fortran 自己的官网上写着的, 安安表示同意.

[2] 安安在 Github 上留下了证据

[3] 太惨了, 都是考点啊!

[4] 本笔记会涉及命令式编程和函数式编程, 由于安安完全是用爱写笔记 (没钱赚), 面向对象式编程在本笔记中出现恐怕遥遥无期...

[5] 有志于伟大天文事业的天文系本科小盆友们必学 Numpy (一个无比基础的 Python 包), 它继承了这样的语法.

- Fortran 专为科学和工程中的计算密集型应用程序而设计, 成熟且经过实战考验的编译器和库[6]允许快速编写贴近硬件运行的代码.

Fortran 的不足有:

- 由于 Fortran 出现的时候, 硬件条件和程序设计观念都不足, 历史上 Fortran 有着许多令人费解的语法特性, 也就是老师们用来花式折磨同学们的坑. 虽然现在 Fortran 官方已经非常努力地填坑了, Modern Fortran 也已经基本上将这些坑填上了, 但还是留下一些没填上的坑 (比如大小写不分之类的).[7]
- 由于 Fortran 专为科学计算设计, 领域有局限, 而且从前某些时候 Fortran 填坑不积极, 填得也有问题, 加之新新编程语言不断出现, Fortran 用的人已经变得太少[8], 可供学习的资料也太少而且很可能内容已经 out 了.
- 因为 Fortran 用的人太少, 所以 Fortran 编译器开发不足, 如今 Fortran 已经不再是运行速度最快的编程语言了.[9]

目前 Fortran 有自己的官网[10], 内有 Fortran 的简单教程和许多 Fortran 的资源链接. 现行 Fortran 标准 Fortran 2023 由标准解释文档[11][12]规定.

[6]刚学 Fortran 的天文系本科小盆友们以后会在天体测量学课上遇到 SOFA 和 NOVAS, 所以不好好学 Fortran 可是要挂俩门的啊!...

[7]这些剩下的坑恐怕是实在不好填, 估计是不会填了, 俺们只得举手投降...

[8]在 Fortran 官方的不懈努力下, 根据 TIOBE 排行, Fortran 流行度已经和 Matlab 差不多啦!

[9]目前 C/C++ 运行速度比 Fortran 快一点, Cython 则比 Fortran 慢一点, 不过他们的运行速度都相差不多. 各主流编程语言运行速度可参阅 Scientific Computing Languages - University of Pennsylvania.

[10]没校园网可能打不开, 同志们自己看着办吧!

[11]真 · 标准文档要花大钱买了才能读, 实在是太可恶啦! 不过这两个文档的内容应该是一致的.

[12]这篇文档虽然绝对正确但超级无敌巨 TM 难读, 同学们还是别碰了...

第二章　编译器

编译器	Ifx (+VS)	Gfortran (+VS Code)
类型	专有软件	自由软件
总空间占用	约 5.2 G	约 780 M
语法支持	略多于 Gfortran, 较宽松	略少于 Ifx, 较严格
自动纠错	无	有
自动补全	无	有
编译提示信息	次于 Gfortran	优于 Ifx
平均运行速度	快于 Gfortran	慢于 Ifx
本笔记使用版本	2024.2.0, on Intel® 64, Build 20240602	13.2.0, 64-posix-seh-rev1, Built by MinGW-Builds

表 2.1: Ifx (+VS) 与 Gfortran (+VS Code) 的对比

2.1　安装

2.1.1　Intel® Fortran Compiler (Ifx)

Windows 系统安装方式如下.

1. 安装 Visual Studio Community 2022. 安装时可选择语言为中文.
2. “工作负载” 选择 “使用 C++ 的桌面开发”.
3. 安装 Intel® Fortran Compiler.
4. 找到开始菜单内 Intel oneAPI 2023 文件夹中的 Intel oneAPI command prompt for Intel 64 for Visual Studio 2022 和 Intel oneAPI command

prompt for IA32 for Visual Studio 2022, 右键选择 “更多” → “打开文件位置”, 可以找这两个东东的快捷方式. 然后右键这两个快捷方式, 选择 “属性”, 把 “起始路径” 改成自己最常访问的路径 (比如桌面的路径), 然后点 “应用”, 点 “继续”, 点 “确定”.

2.1.2 GNU Fortran Compiler (Gfortran)

Windows 系统安装方式如下.[1]

1. 访问 MinGW-Builds-binaries releases, 选择最新的 release 进入.
2. 选带 posix 的; 64 位系统选带 x86_64 的, 32 位系统选带 i686 的; Win 10 及以上选带 ucrt 的, Win 10 以下选带 msvcrt 的. 下载并解压.
3. 把解压出来的名为 `mingw64` 的文件夹剪切到随便哪个目录. 暂称粘贴到的目录为 `[dir]`.
4. 在系统环境变量 `Path` 中加入 `[dir]\mingw64\bin`. 比如, 如果刚才粘贴到 `C:\Program Files`, 就加入 `C:\Program Files\mingw64\bin`.
5. 下载 Visual Studio Code 并安装.
6. 打开 Visual Studio Code, 点击左边四个正方形飞出一个的图标, 搜索 C/C++,Modern Fortran 和 Code Runner 并安装.

   如果已经装了 Python, 装 Modern Fortran 前先用 Pip[2] 装 fortls .
7. 按 Ctrl+Shift+P, 然后选择 “Preferences: Open Settings (JSON)”, 打开名为 `settings.json` 的 JSON 文件.
8. 在 `settings.json` 里加入下面这些键值对.

```
"code-runner.executorMap": {
    "FortranFreeForm":
    "cd $dir; gfortran *.f*; if($?){.\a.exe}",
    "FortranFixedForm":
    "cd $dir; gfortran *.f*; if($?){.\a.exe}"
```

[1] 以下是直接安装 MinGW-w64 来安装 Gfortran 的, 但如果有 Python, 也许能直接通过 Pip (或 Conda) 来安装 Gfortran (或 MinGW-w64, MSYS2, …)? 我也不知…

[2] 有 Conda 当然用 Conda 啦!

```
    },
    "code-runner.runInTerminal": true,
    "code-runner.saveAllFilesBeforeRun": true
```

最后第 8 步还需解释, 因为牵涉到 JSON 文件的语法. JSON 文件里的内容应该满足下面的形式.

```
{
  key_1: value_1,
  ...
  key_n: value_n
}
```

每一个形如 `key_i: value_i` 的东东称作一个键值对, 任意两个键值对之间用逗号隔开. 所以, 如果一开始 `settings.json` 里头空空如也, 则加入键值对后可能长这样.

```
{
  "code-runner.executorMap": {
    "FortranFreeForm":
    "cd $dir; gfortran *.f*; if($?){.\a.exe}",
    "FortranFixedForm":
    "cd $dir; gfortran *.f*; if($?){.\a.exe}"
  },
  "code-runner.runInTerminal": true,
  "code-runner.saveAllFilesBeforeRun": true
}
```

如果一开始 `settings.json` 里长这样,

```
{
  "editor.wordWrap": "wordWrapColumn",
  "editor.wordWrapColumn": 80,
  "workbench.colorTheme": "Red"
}
```

则加入键值对后可能长这样.

```
{
  "editor.wordWrap": "wordWrapColumn",
  "editor.wordWrapColumn": 80,
  "workbench.colorTheme": "Red",
  "code-runner.executorMap": {
    "FortranFreeForm":
    "cd $dir; gfortran *.f*; if($?){.\a.exe}",
    "FortranFixedForm":
    "cd $dir; gfortran *.f*; if($?){.\a.exe}"
  },
  "code-runner.runInTerminal": true,
  "code-runner.saveAllFilesBeforeRun": true
}
```

注意第 4 行最后要多加一个逗号来隔开键值对.

2.2 使用

CLI (Command Line Interface, 命令行界面) 和 GUI (Graphical User Interface, 图形用户界面) 两手都要抓, 两手都要硬. GUI 大家其实老熟了, 平常大家用手机电脑, 只要开机, 见到的就是 GUI. CLI 就是大家看电影里黑客们用的黑框框, 不过现在手机电脑进步了, 框框不一定是黑的, 也可能是红的蓝的之类的.

2.2.1 使用 CLI

我们首先需要请出 CLI 框框. 用 Ifx 的同学请打开开始菜单内 Intel oneAPI 2023 文件夹中的 Intel oneAPI command prompt for Intel 64 for Visual Studio 2022 (32 位系统请打开 Intel oneAPI command prompt for IA32 for Visual Studio 2022). 用 Gfortran 请打开 Powershell, 可以在电脑桌面或文件夹中按鼠标右键然后点 “在终端中打开”, 也可以按 Win+R, 输入 “powershell” 然后点 “确定”, 还可以在 Visual Studio Code 中, 点上面 “Terminal” 后点 “New Terminal”.

我们要熟悉 CLI 的运作流程. 首先我们需要在一些文本文档里写 Fortran 代码, 这些文本文档称为源代码文件 (source code file), 其文件后缀名

一般是 `.f90`/`.for`/`.f`. 然后我们需要用编译器把每个源代码文件依次变成对应的目标文件 (object file), 这个过程称作编译 (compilation), Ifx 编译出的目标文件后缀名是 `.obj`, Gfortran 编译出的目标文件后缀名是 `.o`. 然后我们需要用编译器把所有目标文件合起来变成一个可执行文件 (executable file), 这个过程称作链接 (linking), Windows 系统中链接出的可执行文件后缀名是 `.exe`, Linux 系统中链接出的可执行文件后缀名一般是 `.out`. 广义的编译包括编译和链接. 最后我们需要命令电脑照可执行文件干活儿, 这个过程称之为执行 (execution). 示意图如 2.1 所示.

$$
\begin{array}{lccccc}
 & \text{源代码文件1} & \cdots & \text{源代码文件}n & & \\
\text{编译} & \downarrow & \downarrow & \downarrow & & \\
 & \text{目标文件1} & \cdots & \text{目标文件}n & & \\
\text{链接} & \downarrow & \downarrow & \downarrow & . & (2.1) \\
 & \rightarrow & \text{可执行文件} & \leftarrow & & \\
\text{运行} & & \downarrow & & & \\
 & & \text{电脑工作} & & &
\end{array}
$$

编译执行流程示意图

现在俺们通过一个实例来掌握 CLI 的用法! 我们先在任意一个文件夹里新建两个空白文本文档 (默认的文件名应该是 "**新建 文本文档**`.txt`"), 分别把文件名改为 `main.f90` 和 `helloworld.f90` (注意要把拓展名 `.txt` 也给改掉, 不要改成 `main.f90.txt` 和 `helloworld.f90.txt` 哟).

打开 `main.f90`, 写入下面的内容并保存.

```
program main
    implicit none
    call helloworld()
end program main
```

打开 `helloworld.f90`, 写入下面的内容并保存.

```
subroutine helloworld()
    implicit none
    print *, 'Hello, world!'
end subroutine helloworld
```

请同学们弄出 CLI 框框. CLI 框框中会出现一些文字, 请同学们观察这些文字的最下面一行里包含的文件夹路径, 这个路径对应的文件夹称为当前工作目录 (current working directory). 如果当前工作目录不是 `main.f90` 和 `helloworld.f90` 所在的文件夹, 我们最好 "`cd`" 到 `main.f90` 和 `helloworld.f90` 所在的文件夹去, 用 Ifx 的同学需要在框框里输入 "`cd /D` **目录**" 然后按回车键, 用 Gfortran 的同学需要在框框里输入 "`cd` **目录**" 然后按回车键. 假设 `main.f90` 和 `helloworld.f90` 所在的文件夹是 `C:\Users\GasinAn\Desktop`, 那么我们输入 `cd /D C:\Users\GasinAn\Desktop` 或 `cd C:\Users\GasinAn\Desktop` 然后按回车键. 这样 CLI 框框中的文字就会有变化, 最下面一行里包含的的文件夹路径变成新的当前工作目录 `C:\Users\GasinAn\Desktop` 了.

然后俺们要编译, 用 Ifx 的同学需要在框框里输入 "`ifx /c` **源代码文件**`1 ... 源代码文件n`" 然后按回车键, 用 Gfortran 的同学需要在框框里输入 "`gfortran -c` **源代码文件**`1 ... 源代码文件n`" 然后按回车键. 所以用 Ifx 的同学需要在框框里输入 `ifx /c main.f90 helloworld.f90` 然后按回车键, 用 Gfortran 的同学需要在框框里输入 `gfortran -c main.f90 helloworld.f90` 然后按回车键.

然后俺们要链接, 用 Ifx 的同学需要在框框里输入 "`ifx` **目标文件**`1 ... 目标文件n /o` **可执行文件**" 然后按回车键, 用 Gfortran 的同学需要在框框里输入 "`gfortran` **目标文件**`1 ... 目标文件n -o` **可执行文件**" 然后按回车键. 所以用 Ifx 的同学可以在框框里输入 `ifx main.obj helloworld.obj /o a.exe` 然后按回车键, 用 Gfortran 的同学可以在框框里输入 `gfortran main.o helloworld.o -o a.exe` 然后按回车键, 得到可执行文件 `a.exe`.

最后我们要执行可执行文件, 需要在框框里输入 "**可执行文件**" 然后按回车键. 所以需要输入 `a.exe` 然后按回车键, 看到黑框框里蹦出 `Hello, world!` 就执行成功!

编译和链接的时候我们可以偷懒, 我们可以直接一步进行编译 + 链接两步, 用 Ifx 的同学需要在框框里输入 "`ifx` **源代码文件**`1 ... 源代码文件n /o` **可执行文件**" 然后按回车键, 用 Gfortran 的同学需要在框框里输入 "`gfortran` **源代码文件**`1 ... 源代码文件n -o` **可执行文件**" 然后按回车键. 所以用 Ifx 的同学可以在框框里输入 `ifx main.f90 helloworld.f90 /o a.exe` 然后按回车键, 用 Gfortran 的同学可以在框框里输入 `gfortran main.f90 helloworld.f90 -o a.exe` 然后按回车键, 即可直接得到可执行

文件 a.exe. 同学们肯定要问有偷懒的办法干嘛不早说, 其实是因为同学们如果以后要写超大程序, 里面有成千上万个源代码文件, 这时候分两步编译和链接反而省事. 假设我们已经把成千上万个源代码文件编译 + 链接过一回了, 我们修改了其中一个源代码文件, 希望重新编译和链接, 如果我们再把成千上万个源代码文件一起编译 + 链接一回, 是会非常花时间的 (可能会花几个小时), 这时我们可以只把那个被修改了的源代码文件编译了, 然后再把成千上万个目标文件链接起来 (链接比编译快多了), 这样我们就能省下大把时间了呢.

以上内容总结如下.

“cd” 目录	cd /D 目录
编译	ifx /c 源代码文件1 ... 源代码文件n
链接	ifx 目标文件1 ... 目标文件n /o 可执行文件
编译链接	ifx 源代码文件1 ... 源代码文件n /o 可执行文件
执行	可执行文件

表 2.2: Ifx 编译执行命令

“cd” 目录	cd 目录
编译	gfortran -c 源代码文件1 ... 源代码文件n
链接	gfortran 目标文件1 ... 目标文件n -o 可执行文件
编译链接	gfortran 源代码文件1 ... 源代码文件n -o 可执行文件
执行	可执行文件

表 2.3: Gfortran 编译执行命令

我们还可以偷懒. 首先我们可以用 . 表示 “当前文件夹”, .. 表示 “上一层文件夹”. 所以 main.f90 和 helloworld.f90 等价于 .\main.f90 和 .\helloworld.f90 (虽然没人会自找麻烦这么写), 并且假设我们的当前工作目录是 C:\Users\GasinAn\Desktop, 我们 “cd” 到 ..\.. 就是 “cd” 到 C:\Users.

然后我们可以用通配符 (wildcard character), ? 表示 “任意一个不是 . 或 \ 的字符”, * 表示 “零个或任意多个不是 . 或 \ 的字符”. 例如 ?.f? 表示开头是一个字符, 中间是 .f, 结尾是一个字符的所有文件和文件夹, *.f*

表示开头是零个或任意多个字符, 中间是 `.f`, 结尾是零个或任意多个字符的所有文件和文件夹. 因此之前的 `main.f90 helloworld.f90` 我们都可以直接替换成 `*.f*`, 省大事了.

最后可以只输入文件或文件夹名称的前面几个字符, 然后按 Tab 键来自动补全文件或文件夹名称. 如果补全出的文件或文件夹不是我们想要的, 我们还可以多次按 Tab 键直到补全出的文件或文件夹是我们想要的. 同学们可输入 `he` 然后多次按 Tab 键查看效果.

2.2.2 使用 GUI

Visual Studio (VS)

打开 Visual Studio, 点 "创建新项目".

然后搜索 "Empty Project", 找到下面标有 "Fortran", "Windows", "控制台" 的 "Empty Project", 选择之, 然后点击 "下一步".

然后把 "项目名称" 改成 "HelloWorld", "位置" 选择自己喜欢的文件夹 (比如桌面) 的路径 (下面用 `[dir]` 表示这个路径), 点击 "创建".

然后就出现了编辑界面, 并且 `[dir]` 内多出了一个名为 `HelloWorld` 的文件夹.

默认使用的是 Ifort[3]. 欲用 Ifx, 右键右边 "解决方案资源管理器" 里的 "Console1 (Ifx)", 点最下面的 "属性", 左边选 "配置属性" → "General" (应已自动选上), 点 "Use Compiler" 右边的 "Ifx Intel® Fortran Compiler Classic", 点最右边的带向下标志的按钮, 改选成 "IFX Intel® Fortran Compiler", 点 "应用", 点 "确定", 看到右边 "Console1 (Ifx)" 变成 "Console1 (IFX)" 即成功.

64 位系统, 若上面 Debug 后是 x86, 则可能需要将上面 Debug 后的 x86 改成 x64 (如不需要, 也最好改改). 点 x86 右边的向下箭头可以改, 若没有 x64, 可以点 "配置管理器..." 后尝试把 x64 调出来.

右击右边 "解决方案资源管理器" 中的 "Source Files", 选择 "添加" → "现有项...", 然后把之前的 `main.f90` 和 `helloworld.f90` 添加进来. 然后双击文件名即可打开文件.

点击上面的 "调试", 然后点 "开始执行 (不调试)", 看到下面框框里蹦出 `Hello, world!` 就用 Ifx 编译执行成功!

[3] Intel© Fortran Compiler Classic, 这玩意儿 Intel© 不想玩儿了, 要用 Ifx 来代替.

右击右边 “解决方案资源管理器” 中的 “Source Files”, 选择 “添加” → “新建项…”, 可以新建文件, 默认在 `HelloWorld` 文件夹里的 `HelloWorld` 文件夹里. 右击任意一个文件的文件名后, 可以点击 “删除” 或 “重命名” 进行删除或重命名操作.

关掉 Visual Studio, 重新打开, 左边多出了个 HelloWorld.sln, 点它就能回到编辑界面了.

Visual Studio Code (VS Code)

用 Visual Studio Code 打开 `main.f90`[4], 然后点右上方的白色三角儿就用 Gfortran 编译执行成功!

[4]之后同学们学 3.3 节, 会明白 `main.f90` 里写的是 “主程序”, 所以打开它.

第三章　基础知识

3.1　程序

程序 (program) 是指挥电脑工作的一堆指令. 这堆指令要用一堆字符表示, 表示指令的所有字符合起来称为源代码 (source code). 源代码需要被保存在文件中, 保存源代码的所有文件合起来称为源代码文件 (source code file).

比如, 在一个名为 `main.f90` 的文件内写入下面这些内容并保存.

```
program main
    implicit none
    call helloworld()
end program main
```

在另一个名为 `helloworld.f90` 的文件内写入下面这些内容并保存.

```
subroutine helloworld()
    implicit none
    print *, "Hello, world!"
end subroutine helloworld
```

那么 `main.f90` 和 `helloworld.f90` 就是源代码文件, 这俩文件里头一共有 8 行字符, 这些字符合起来就是源代码, 表示的指令如表 3.1 和表 3.2 所示, 这一大堆指令合起来就是程序.

3.2　标准与规范

任何编程语言都有自己的句法/语法 (syntax), 比如 “让电脑屏幕上出现 ‘A’”, Fortran 要写成 `print *, "A"`, C 要写成 `printf("A");`, Python 2

源代码	指令
`program main`	开始运行主程序 `main`
`implicit none`	禁止隐式声明
`call helloworld()`	调用子程序 `helloworld`
`end program main`	结束运行主程序 `main`

表 3.1: `main.f90` 中源代码与指令的对照

源代码	指令
`subroutine helloworld()`	开始运行子例行 `helloworld`
`implicit none`	禁止隐式声明
`print *, "Hello, world!"`	让电脑屏幕上出现 “Hello, world!”
`end subroutine helloworld`	结束运行子例行 `helloworld`

表 3.2: `helloworld.f90` 中源代码与指令的对照

要写成 `print "A"`, Python 3 要写成 `print("A")`. 编程语言对语法的统一规定称为标准 (standard). Fortran 的现行标准是 Fortran 2023 (第一章中已介绍), 旧标准有 FORTRAN 66, FORTRAN 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008, Fortran 2018.

不太好玩儿的是, 虽然 Fortran 语法已由现行标准规定好了, Fortran 编译器们却偶尔不按 Fortran 标准来干活儿, 自己制定了一些规则, 还有些时候, Fortran 标准直接就把权利赋予编译器, 让他们自定一些规则. 这样, Fortran 编译器们时不时就有和 “官规” 不同的 “器规”.

编程语言的标准是 “硬规则”, 除此之外还有 “软规则”, 称为规范 (convention). 所谓 “规范”, 就是被代码毒打的前人总结的血泪教训. 不遵守规范, 也不一定会出什么问题, 但一旦出了问题, 就不知道问题出在哪儿啦. 请同学们尽量遵守本笔记列出的 Fortran 规范, 不然期末挂了不要怪 Fortran 哟! 不过有时不遵守规范也确实不会出什么问题, 安安会适时说明的.

编译器各有各的器规, 这可能会出事. 想象一下, 同学们期末考写好程序, Ifx 和 Gfortran 说程序没问题, 同学们兴冲冲地交卷了, 然后老师掏出 Visual Fortran (某老款编译器), Visual Fortran 说程序有问题, 同学们就挂科啦. 为防止这种悲惨情形发生, 我们需要避免使用 Fortran 标准没规定而编译器自定的规则.

规范 1 *只使用 Fortran 现行标准自行规定的规则.*

这样, 我们的程序编译器要是不认, 安安唯一能想到的可能就是 Fortran 标准更新了, 而编译器没更新, 还用着旧标准. 不过同学们没必要担心这一情形, 本笔记里的东东绝大多数 Fortran 90 就有了 (我猜老师现在也只教到 Fortran 90), 即使是 Visual Fortran 也会认的, 要是某 Fortran 编译器竟连 Fortran 90 都不认, 我们就把它卸载了…

3.3 程序单元

每个 Fortran 程序都由若干程序单元 (program unit) 组成. 程序单元可能是主程序 (main program), 或外部子程序 (external subprogram), 或模块 (module), 或子模块 (submodule). 一个 Fortran 程序中必须有且只有一个主程序, 其他程序单元数目随便.

主程序长下面这样.

```
program [program-name]
    ...
end program [program-name]
```

其中 `[program-name]` 处是一个名称 (name), 名称指的是由字母, 数字和下划线 "_"[1] 组成的, 开头是字母的字符串[2], "..." 处则是各种指令, 一行一条. 比如, 之前 3.1 节里 `main.f90` 里的程序就是一个名称为 `main` 的主程序[3]

(`helloworld.f90` 里的程序则是一个外部子程序).

外部子程序在第八章里讲, 模块在第**??**章里讲, 子模块在第**??**章里讲.[4]

每个 Fortran 源代码文件里都可以放若干完整的程序单元, 但习惯上一个源代码文件里只放一个程序单元, 并且文件名和文件里的程序单元名相同 (比如主程序 `main` 放在 `main.f90` 里).

规范 2 *一个源代码文件里只放一个程序单元, 并且文件名和文件里的程序单元名相同.*

[1] 下划线是空格的替代字符, 长得还是蛮像滴!

[2] 后文中的 "某某名" 若不另加说明则都是名称, 不再赘述.

[3] 安安习惯把主程序的名称取做 `main` 并放在 `main.f90` 里.

[4] ??的内容安安会在猴年马月写好的!

当然, 安安偶尔也会在一个源代码文件里放多个程序单元, 比如安安在玩 SOFA 的时候, 习惯把除 `t_sofa_f.for` 外的所有文件里的东东通通复制粘贴到一个名为 `sofa.for` 的新文件里, 这个文件里就有一大堆外部子程序了.

今后程序若有多个程序单元, 我将把这多个程序单元写在一起 (虽然按规范 2 它们应该被分别保存). 比如 3.1 节 `main.f90` 和 `helloworld.f90` 中的程序会合写成如下模样.

```
program main
    implicit none
    call helloworld()
end program main

subroutine helloworld()
    implicit none
    print *, "Hello, world!"
end subroutine helloworld
```

3.4 编译与运行

我们用 Fortran 语写好了程序, 命令电脑照此干活儿, 然而电脑会说干不了. 原因在于, 电脑不认得 Fortran 语, 电脑只懂得电脑语 (学名是机器语言), 甚至不同电脑的电脑语还不一样. 而电脑语身为正常人类的我们完全搞不懂…

这时我们就得请编译器 (compiler) 们出场干活儿了, 它们会在操作系统的帮助下, 把 Fortran 语翻译成电脑语, 这个过程称为编译[5] (compile). 2.2.1 小节中输入 `ifx *.f* /o a.exe` 回车就是请 Ifx 编译, 2.2.1 小节中输入 `gfortran *.f*` 回车就是请 Gfortran 编译, 最后电脑语程序保存在称为可执行文件 (executable file) 的 `a.exe` 里.

把程序翻译成电脑语后, 电脑没借口不干活儿了. 我们输入 `a.exe` 回车, 命令电脑照电脑语程序干活儿, 这个过程称为运行/执行 (execute) 程序, 也俗称跑 (run) 程序. Fortran 程序中指令运行的基本顺序是从主程序第一行 `program [program-name]` 开始, 向下逐行依次运行, 到主程序最后第一行 `end program [program-name]` 结束. 本笔记里会介绍许多例外情况, 遇

[5] 这里说的是广义的编译. 狭义的编译是广义的编译中的一个环节.

到例外时会称之为跳转 (jump). 例如 3.1 节 `main.f90` 和 `helloworld.f90` 中的程序, 指令的运行顺序如下, 其中指令 3 跳转到指令 4, 指令 7 跳转到指令 8.

源代码	指令
`program main`	1. 开始运行主程序 `main`
`implicit none`	2. 禁止隐式声明
`call helloworld()`	3. 调用子程序 `helloworld`
`end program main`	8. 结束运行主程序 `main`

表 3.3: `main.f90` 中指令运行顺序

源代码	指令
`subroutine helloworld()`	4. 开始运行子例行 `helloworld`
`implicit none`	5. 禁止隐式声明
`print *, 'Hello, world!'`	6. 让电脑屏幕上出现 “Hello, world!”
`end subroutine helloworld`	7. 结束运行子例行 `helloworld`

表 3.4: `helloworld.f90` 中指令运行顺序

当年编程语言刚刚出世的时候, 人们对程序设计的认识不足, 程序里的指令可以胡乱跳转, 总有人写出让人根本读不懂的鬼魅程序. 后来人们制定了结构化 (structured) 程序设计原则, 其基本特征是只保留尽可能少的, 有明确规则的指令跳转. Fortran 语法如今已基本符合结构化原则了, 但仍有少数不符合的. 本笔记里会介绍一些不符合结构化原则的语法, 但只是因为不用那些语法, 有些活儿干不了. 请同学们尽可能少用那些语法.

规范 3 *尽可能少用不符合结构化程序设计原则的语法.*

最后补充一下, 玩 Fortran 时我们是先用编译器把 Fortran 程序从头到尾全翻译了然后再运行. 但编程语言还有另一种运行方式, 是用解释器 (interpreter) 把程序一点点翻译成机器语言, 翻译一点运行一点, 好像电脑装了解释器后就能直接读懂编程语言一样, 这个过程称为解释 (interpret). 这两种方式还可以混用, 同学们以后玩 Python 的时候就会遇到混合方式.

3.5 异常

程序编译和运行时可能会有不妙的事情发生, 那就是抛出 (raise) 异常 (exception). 异常分为错误 (error) 和警告 (warning), 抛出错误和抛出警告也有些俗称, 比如报错和弹警告之类的.

3.5.1 错误

抛出错误有两种情况. 最常见的是我们写的程序不符合 Fortran 语法, 编译器读不懂, 无法翻译, 就抛出错误然后罢工了. 比如, 编译这样一个保存在 `main.f90` 中的程序.

```
program main
    implicit none
    print 'Hello, world!'
end program main
```

Ifx 给出的结果如下[6].

```
main.f90(3): error #6899: First non-blank character in a cha
racter type format specifier must be a left parenthesis.  [
'Hello, world!']
    print 'Hello, world!'
-----------^
compilation aborted for main.f90 (code 1)
```

Gfortran 给出的结果如下.

```
main.f90:3:10:


    print 'Hello, world!'
          1
Error: Missing leading left parenthesis in format string at
(1)
```

由于程序第三行不符合 Fortran 语法 (`print` 后少了 `*,` 之类的东东), 编译器不知怎么干活儿了, 只好报个错. 智能的 VS Code 会在编译前就请 Gfortran 来检查我们的程序是否有语法错误, 这将节约我们的时间.

[6]假定屏幕宽度是 60 字符宽, 且略去版权声明等无关紧要的内容, 下同.

因为有器规, 所以报错还和编译器有关. 比如运行下面这个程序.

```
program main
    implicit none
    print *, 1.0/0.0
end program main
```

Gfortran 会报错说不能除以 0, 但 Ifx 不会报错并会算出正无穷.

另一种抛出错误的情况是我们写的程序符合 Fortran 语法, 编译器能翻译, 但之后运行的时候程序请操作系统帮忙干活儿, 操作系统发现没法儿干, 程序就抛出错误然后罢工了. 比如在 Windows 系统里运行这样一个程序.

```
program main
    implicit none
    open(10, file='C:\*')
end program main
```

这回编译时 Ifx 和 Gfortran 都不认为有什么问题, 但运行时程序让 Windows 打开文件 `C:\*` (若无此文件则新建一个后打开), 而 Windows 文件名里不可以有 `*`, 程序只好报个错. 这个程序在同学们以后会学的 Linux 系统里就可以正常工作, 因为 Linux 文件名里可以有 `*`.

3.5.2 警告

有时候我们写的程序符合 Fortran 语法, 编译器读得懂能翻译, 但它觉得我们写的程序十分 “危险”, 很有可能程序的实际执行过程并不是我们预想的那样, 这时编译器就会抛出警告. 什么时候应该给个错误, 不同编译器的观点还是比较一致的, 但什么时候给个警告, 那区别就相当大了.

比如, 编译这样一个保存在 `main.f90` 中的程序.

```
program main
    implicit none
    integer :: i
    i = 1000000000000
    print *, i
end program main
```

Ifx 给出这样的结果.

```
main.f90(4): warning #8221: This integer constant is outside
 the default integer range - using INTEGER(8) instead.   [10
00000000000]
    i = 1000000000000
--------^
main.f90(4): warning #6384: This value is out-of-range for a
n INTEGER(KIND=4) type.   [1000000000000]
    i = 1000000000000
--------^
```

Ifx 认为 1000000000000 过大, 无法正确存储, 于是就给个警告, 但程序仍然可以运行. 正常想来, 程序运行后电脑屏幕上应该出现 1000000000000, 结果出现的却是 -727379968, 果然很奇怪! 至于 Gfortran 嘛, 它也认为 1000000000000 过大, 但它不是给个警告, 而是直接报错.

再比如, 编译这样一个保存在 main.f90 中的程序.

```
program main
    implicit none
    integer :: i
    i = .true.
    print *, i
end program main
```

Gfortran 给出这样的结果.

```
main.f90:4:8:


    4 |     i = .true.
      |        1
Warning: Extension: Conversion from LOGICAL(4) to INTEGER(4)
 at (1)
```

Gfortran 觉得令整数 i 等于逻辑 “真” 的操作其实是标准不允许的, 是自己放水才能这么干的, 于是就给个警告 (VS Code 会提前请 Gfortran 来看是否会弹警告), 但程序仍然可以运行. 运行后电脑屏幕上出现 1 . 至于 Ifx 嘛, 它倒是不报错也不给警告, 干脆利落地编译完了, 但因为器规不同, 运行后电脑屏幕上出现的是 -1…

3.6 字符集

Fortran 可以在程序中使用所有能直接用英语输入法打出的字符, 其他字符 (比如汉字, 汉语标点) 能否在程序中用则由器规决定. 注意, 英文标点和中文标点是完全不同的[7], 英文标点是 “半角标点”, 中文标点是 “全角标点” 哟![8]

经俺测试, Ifx 和 Gfortran 都是支持汉字和汉语标点的, 但直接让电脑屏幕上出现汉字或汉语标点可能会出现 “稽堵筵豫畋[9]” 之类的乱码 (这可以解决).

为避免各种各样的问题, 还是只用正常字符为好. 另外源代码文件名最好也如此, 不然编译器可能会无法识别源代码文件呢.

规范 4 *在程序和源代码文件名中只使用能直接用英语输入法打出的字符.*

Fortran 是大小写不敏感的 (case-insensitive), 也就是说, 很多情况下程序里的字母大写小写效果是完全相同的 (效果不同的地方见 4.4 节和 9.3.3 小节). 比如, 下面这两个程序和 3.1 节的 `main.f90` 内的程序是完全等价的.

```
PROGRAM MAIN
    IMPLICIT NONE
    CALL HELLOWORLD()
END PROGRAM MAIN
```

Fortran 大小写不分是有历史原因的. 当年电脑太烂了, 为节省存储空间, 一开始 Fortran 只许用大写字母, 后来电脑不那么烂了, 能用小写字母了, Fortran 大概是因为想尽量保证老程序能用, 所以直接规定大小写不分, 结果挖了个如今没法儿填的史诗级查坑, 学 Fortran 玩 Fortran 的历来都深受其害. 举个例子, 当年俺搞毕设, 需要解 Kepler 方程 $E = M + e\cos E$, 在程序里对应地写成 `E = M + e*cos(E)`, 然而 Fortran 是大小写不分的, 所以 `E = M + e*cos(E)` 其实对应于 $e = M + e\cos e$ 或 $E = M + E\cos E$, 然后俺忘了这茬, 结果死活找不出自己哪里写错了…

被 Fortran 反复毒打后, 人们决定做些什么来避免被毒打. 既然 Fortran 大小写不分, 那避免混淆的最好办法就是在程序里统一只用小写或大写字母

[7] 用 VS Code 的同学会发现英文标点和中文标点显示出来完全不一样.

[8] 本笔记里使用的全部是英文标点呢.

[9] 不认识这东东的同学请自行搜索.

(俺搞毕设时知道这事, 但俺偷懒了, 结果惨遭报应…). 因为单词写成小写字母比写成大写字母好认, 所以现在人们都统一只用小写字母.

规范 5 *只在程序中使用小写字母.*

如上所述, 有时 Fortran 大小写是分的, 这时规范 5 没法儿遵守, 当然就不遵守了. 除此之外, 规范 5 还有其他一些可以不遵守的情形, 俺留着后面说. 这样, `E = M + e*cos(E)` 就得改写了. 因为 Kepler 方程里 E 是偏近点角, 是个角, 所以 `E` 可以写成 `e_angle` (也可以写成 `angle_e`, 但绝不能写成 `E_angle`!!!), e 则是离心率, 虽然写成 `e` 也可以和 `e_angle` 区别开, 但为了让区别更明显, 可以写成 `ecc` (eccentricity 的缩写).

3.7 源代码格式

Fortran 有两种源代码格式 (source form): 自由格式 (free form) 和固定格式 (fixed form). 固定格式是将被废弃的 (obsolescent) 老格式. 安安的笔记里只讲自由格式.

规范 6 *永远编写自由格式的程序.*

自由格式源代码文件拓展名一般是 `.f90`, 固定格式源代码文件拓展名一般是 `.for`/`.f`, 这是 Fortran 编译器的惯例. 查 Ifx 文档 和 Gfortran 文档 可知, 它俩都遵从这个惯例. 请同学们把自由格式的 Fortran 程序保存在拓展名为 `.f90` 的文件里, 以保证它俩能认得. 要是某 Fortran 编译器竟不遵从惯例, 我们就把它卸载了…

规范 7 *保证自由格式源代码文件拓展名为 `.f90`.*

这里需澄清, 有习惯说法是把自由格式程序说成 “Fortran 90 程序”, 把固定格式程序说成 “FORTRAN 77 程序”. 这种说法的根源是 Fortran 历史上有一场巨变: FORTRAN 77 只有固定格式, 而 Fortran 90 新引入了自由格式. 然而, 这并不意味着固定格式程序中不能使用 Fortran 90 及以后的标准新出的语法.

在 VS Code 中, 把鼠标移到右下角铃铛图标按键左边第一个, 鼠标移到后会显示 “Select Language Mode” 的按键上, 点击之, 然后弹出的菜单下拉选择 “Fortran (FortranFreeForm)”, VS Code 便会自动检查程序是否符合 Fortran 自由格式.

3.7.1 空格

Fortran 代码里空格 (blank/space) 通常没有意义 (4.4 节和 9.3.3 小节中有例外), 可多可少可没有, 但有些时候空格是必要的, 否则会出现混淆, 比如 `program main` 就不能写成 `programmain` (没有编译器能智能到把它拆成两个词).

虽然空格没有意义, 但能让代码更清楚. 如果把 `print *, 'A'` 写成 `print*,'A'`, 挤成一团, 也不能说是错误, 但即便像 VS Code 里那样代码有颜色区分, 也让人看得晕乎, 更何况笔记里这样代码没颜色呢.

规范 8 *在程序中适当地加入空格.*

至于什么时候加入空格, 其实没什么具体原则, 不同的人意见不一. 同学们可以参考本笔记, Fortran 官网上的代码示例和 Python 的 PEP 8 规范来决定是否加空格.

3.7.2 缩进

缩进 (indent) 是指源代码每行最前面的空格. 对 Fortran 而言, 缩进本身没有意义, 但适当的缩进能表示程序的某个结构嵌套在另一个结构里, 这能使程序更容易被理解. 来看下面这个程序.

```
program main
    implicit none
    integer :: i
    do i = 1, 1
        if (.true.) then
            print *, 'I'
        end if
    end do
end program main
```

在上面的程序中: 第 1 行到第 9 行是个主程序, 之间的代码缩进一层, 有 4 个空格的缩进; 第 4 行到第 8 行是个 6.1.2 小节会介绍的 `do` 结构, 嵌套在主程序里, 之间的代码又缩进一层, 有 8 个空格的缩进; 第 3 行到第 7 行是个 6.1.1 小节会介绍的 `if` 结构, 嵌套在 `do` 结构里, 之间的代码又又缩进一层, 有 12 个空格的缩进.

规范 9 *在程序中加入适当的缩进, 以 4 个空格为单位.*

“以 4 个空格为单位”, 是指保持缩进为 4 个空格, 或 8 个空格, 或 12 个空格, 以此类推. 有些人可能更喜欢以 8 个空格为单位的缩进, 如果他们的电脑屏幕比较宽, 不怕代码跑到屏幕右侧以外的话. 比如, 上面的程序如果缩进改以 8 个空格为单位的话, 就会变成下面的程序.

```
program main
        implicit none
        integer :: i
        do i = 1, 1
                if (.true.) then
                        print *, 'I'
                end if
        end do
end program main
```

请注意, 在文本编辑器中按 [Tab] 键, 也会出现一长串 “空格”, 但实际上对电脑而言, 按 [Tab] 键和按一堆空格, 效果是不同的. 智能的文本编辑器, 比如 VS Code, 可能会自动把 [Tab] 转换为适当数目的空格, 但这不保险. 一些经典的优秀编辑器, 比如 Vim, 是会严格区分 [Tab] 和空格的[10], 除非另装外挂. 编译器可能会允许在程序中使用 [Tab] 键, 但这也不保险. 总而言之, 使用 [Tab] 键, 实在不清楚会不会出问题.

规范 10 *不要使用* [Tab] *键.*

在 VS Code 中, 把鼠标移到右下角铃铛图标按键左边第四个, 鼠标移到后会显示 “Select Indentation” 的按键上, 点击之, 然后弹出的菜单下拉选择 “Indent Using Spaces”, 再选择 “4”, VS Code 便会智能添加以 4 个空格为单位的缩进了.

3.7.3 空行

为使得程序更容易被理解, 还要在程序中加入适当的空行, 以把程序划分成多个部分. 我们可以给上面 3.7.2 小节的程序加一个空行, 变成下面的程序.

[10] 这并不见得是坏事, 因为有些特殊文件是必须使用 [Tab] 的.

```
program main
    implicit none
    integer :: i

    do i = 1, 1
        if (.true.) then
            print *, 'I'
        end if
    end do
end program main
```

学第四章后同学们可知, 在上面的程序中: 空行以上的部分可以称为 “说明部分”; 空行以下的部分可以称为 “执行部分”. 还常用双空行和单空行来进一步划分. 比如下面这个程序.

```
program main
    implicit none
    integer :: m, n


    do m = 1, 1
        if (.true.) then
            print *, 'M'
        end if
    end do

    do n = 1, 1
        if (.true.) then
            print *, 'N'
        end if
    end do
end program main
```

在上面的程序中: 双空行划分 “说明部分” 和 “执行部分”; 单空行划分 “执行部分” 中的各部分.

规范 11 *在程序中加入适当的空行.*

至于什么时候加入一个或几个空行, 其实没什么具体原则, 完全看个人心情. 简单原则就是, 如果读程序长长的一部分, 感觉头昏脑涨, 或觉得以后肯定会头昏脑涨, 插入空行来把这部分划分成各自能完成一件事的几部分就是不错的选择.

3.7.4 注释

程序每一行中, ! 及后面的所有内容都会被无视 (例外见 4.4 节和 9.3.3 小节), 也就相当于 ! 及后面的内容不存在. 会被无视的 ! 及后面的内容称为注释 (comment). 例如下面两个程序是完全一样的.

```
program main
    !                     _ooOoo_
    !                    o8888888o
    !                    88" . "88
    !                    (| -_- |)
    !                    O\  =  /O
    !                 ___/`---'\___
    !               .' \\|     |// '.
    !              / \\|||  :  |||// \
    !             / _||||| -:- |||||- \
    !            |   | \\\  - /// |   |
    !            | \_|  ''\---/''  |_/ |
    !            \  .-\__  '-'  ___/-. /
    !          ___'. .'  /--.--\  `. .'___
    !       ."" '<  `.___\_<|>_/___.' >' "".
    !      | | :  `- \`.;`\ _ /`;.`/ - ` : | |
    !      \  \ `_.   \_ __\ /__ _/   .-` /  /
    ! =====`-.____`.___ \_____/___.-`___.-'=====
    !                   `=---='
    implicit none
    print *, 'No bugs forever!' ! The Buddha bless us!
end program main
```

```
program main
    implicit none
    print *, 'No bugs forever!'
end program main
```

注释看起来没什么用, 其实用处大极了, 主要功能就是在代码中插入“笔记”, 方便读代码的人理解代码在干什么. 比如下面这个程序, 其实是算 10 的阶乘, 但是代码本身只能表示 “从 1 乘到 10” 的意思, 要理解是算 10 的阶乘, 还要费点脑子, 而加上个注释, 就能一眼看出代码是在算 10 的阶乘了 (除非英语看不懂...).

```
program main
    implicit none
    integer :: i, f
    ! Calculate f == factorial(10).
    f = 1
    do i = 1, 10
        f = f * i
    end do
    print *, f
end program main
```

因为经常有偷懒不加注释, 结果写代码的人自己过几天都看不懂自己之前写的代码的事情发生, 人们总是强调要写注释. 但这种说法有偏颇之处, 因为有些代码自己足以表达意思, 根本没必要加注释. 比如下面这个程序, 注释就是废话, 因为看代码本身就能一眼看出代码是在算 $\tan 0$.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    ! Calculate tan(0).
    print *, tan(0.0_dp)
end program main
```

当然, 如果代码自己不足以表达意思, 还是尽可能加上注释吧, 偷懒可是要遭报应的呀!

规范 12 *在代码本身不足以表明代码含义的时候尽可能加入注释以说明.*

因为注释会被忽略, 所以规范 5 我们可以无视. 同学们乃华夏儿女, 看来都很希望能在注释里用中文, 然而用中文可能导致编译器不能干活, 还是用英文吧! (尤其是代码要写给外国人看的时候)

3.7.5 续行

写代码的时候, 我们不希望一行里有过多的字符, 因为代码会跑到屏幕右侧以外, 不方便阅读.

规范 13 *程序每一行不应超过 80 个字符.*

有些人可能会放宽到不超过 120 个字符, 配合以 8 个空格为单位的缩进和宽屏电脑.

用 VS Code 的话, 可以在 `settings.json` 里加入下面这两个键值对.

```
"editor.wordWrap": "wordWrapColumn",
"editor.wordWrapColumn": 80
```

这样如果一行超过 80 个字符, 就会分行显示. 注意, 只是分行显示, 这一行还是超过 80 个字符的 (可以看到 “分行” 后下面一 “行” 是没有行号的). 加入这两个键值对只是起提醒作用, 太长还是要分行写的 (照顾那些没有像 VS Code 这样智能的编辑器的可怜人).

但有时代码一行确实写不完, 这时我们可以使用续行 (continuation). 首先我们需要介绍注释行 (comment line). 注释行就是相当于空行的行, 包括空行, 只有空格的行和除空格外只有注释的行. 而续行就是在一行的末尾加上 `&`, 表示把下面第一个不是注释行的行剪切下来粘贴到这行后 (并去掉用于表示续行的 `&`). 比如下面两个程序是一样的.

```
program main
    implicit none
    print *, 'All Phenomena, ' &
             //'are illusions.'
end program main
```

```
program main
    implicit none
    print *, 'All Phenomena, '//'are illusions.'
end program main
```

如果一个续行不够, 我们还可以连着用, 像下面的程序这样.

```
program main
    implicit none
    print *, 'All the Relative, are like ' &
             //'dreams, illusions, ' &
             //'bubbles, shadows, ' &
             //'dew drops and lightning flashes: ' &
             //'This is what we must believe in.'
end program main
```

我猜同学们会觉得续行的定义非常繁复难懂, 非得先定义一个 “注释行”, 然后再巴拉巴拉, 难道不能直接定义成 “把下面第一行剪切下来粘贴到这行后”? 还真不行. 来看下面这个程序.

```
program main
    implicit none
    print *, 'All Phenomena, ' &
             ! Important!
             //'are illusions.'
end program main
```

如果把续行定义成 “把下面第一行剪切下来粘贴到这行后”, 上面那个程序就等同于下面这个程序.

```
program main
    implicit none
    print *, 'All Phenomena, ' ! Important!
             //'are illusions.'
end program main
```

这个程序无法工作. 实际上按正确定义, 上面上面那个程序等同于下面这个程序 (注释被忽略), 没有问题.

```
program main
    implicit none
    print *, 'All Phenomena, '//'are illusions.'
end program main
```

续行有许多需要补充说明的. 首先注释相比续行是优先的, 所以下面这个程序不行, 因为 `&` 在 `!` 后, 被当成注释了 (笔记里显示成斜体 *&*).

```
program main
    implicit none
    print *, 'All Phenomena, ' ! --- The Buddha &
             //'are illusions.'
end program main
```

写成下面这样才行, 注释被删去后可以成功续行.

```
program main
    implicit none
    print *, 'All Phenomena, ' & ! --- The Buddha
             //'are illusions.'
end program main
```

4.4 节和 9.3.3 小节中还有其他续行失效的例子.

乱用续行搞幺蛾子可不行. 来看下面这个程序. 这个程序没有错, 但让人看得晕乎.

```
program &
main
    implicit &
    none
    print &
    * &
    , &
    'All Phenomena, '//'are illusions.'
end &
program &
main
```

为了不晕乎, 我们首先要尽量少用续行 (把 `program main` 拆成两行简直是犯罪), 其次我们不应该在不应加空格的地方用续行 (比如在 `*,` 之间用续行拆成两半).

规范 14 *只在不用续行就无法遵守规范 13 的时候使用续行.*

规范 15 *不在不应加空格的地方使用续行.*

有同学可能会突然发现不对, 本章的许多程序不符合规范 9, 例如本章的第一个程序. 其实用续行的时候, 规范 9 可以不遵守. 本章的第一个程序采用的那种缩进方式, 是让下一行和上一行 “内容对齐”, 这也是续行时常用的方式. 具体到程序, 就是让 `*,` 所在的行的下一行的第一个非空格字符, 和 `*,` 之后的第一个非空格字符对齐. 不过续行时的缩进还有其他方式. 同学们可以参考本笔记, Fortran 官网上的代码示例和 Python 的 PEP 8 规范来决定续行时如何缩进.

续行还有一个冷门规则, 在程序的每一行中, 都不能除空格和注释外, 单单只有一个用于续行的 `&`. 比如下面这个程序是不行的. 然而, Ifx 和 Gfortran 却认为这个程序没问题, 同学们遇到器规啦!

```
program main
    implicit none
    print *, 'All Phenomena, '//'are illusions.'
    & ! --- The Buddha
end program main
```

至于为什么有这么个怪怪规则, 我猜是由于其他我不想讲的续行规则, 如果不定这个规则, 代码就可能会出现歧义. 而为什么那些续行规则我不想讲, 是因为那些规则会让同学们不遵守规范搞幺蛾子, 写出读不懂的程序来.

3.7.6 标号

除后面会说的例外情况外, 在程序中的每一行的开头都可以加上标号 (label). 标号是用不超过五个数字表示的 1–99999 的整数, 开头可以是 0, 后面需要跟至少一个空格. 习惯上使用四位数的, 整千或整百的标号. 下面的程序第三行使用了标号.

```
program main
    implicit none
    1000 print *, 'label'
end program main
```

显然在一个程序单元内不能有重复的标号, 比如下面这个程序不成 (两个标号都是 9999).

```
program main
    implicit none
    09999 print *, '09999'
    9999  print *, '9999'
end program main
```

注释行不被允许加标号, 原因可能和续行一样, 也是会有歧义 (但这回 Ifx 和 Gfortran 的器规也是不允许). 另外续行符 `&` 所在的行的下面第一个不是注释行的行, 也就是 “用于续行的行”, 也不被允许加标号, 因为 “用于续行的行” 其实只是续行符 `&` 所在的行的后面部分, 它的开头其实不是一行的开头. 以上规则的反例如下.

```
program main
    implicit none
    print *, 'All Phenomena, '//'are illusions.'
    1000 ! --- The Buddha
end program main

program main
    implicit none
    print *, 'All Phenomena, ' &
             ! Important!
             1000 //'are illusions.'
end program main
```

标号是 Fortran 老得要死的语法, 不是个好东西. 首先, 标号只是个数字, 没法儿给所在的那行是干什么用的提供任何有效信息. 更要命的是, 使用标号写出来的程序极可能不符合结构化原则. 请同学们根据规范 3 少用标号.

3.7.7 标签

在程序中某些特殊的地方 (见 6.2.2 小节和 6.2.3 小节) 可以加标签[11] (tag). 标签用其标签名表示. 下面的程序第四行和第六行使用了标签名为 `tag` 的标签.

[11] 标准解释文档里没这个词, 但官网教程里有.

```
program main
    implicit none
    integer :: i
    tag: do i = 1, 1
        print *, 'tag'
    end do tag
end program main
```

显然在一个程序单元内也不能有重复的标签, 比如下面这个程序不成.

```
program main
    implicit none
    integer :: i, j
    tag: do i = 1, 1
        tag: do j = 1, 1
            print *, 'tag'
        end do tag
    end do tag
end program main
```

标签不像标号, 是好东西. 同学们可以尽情使用.

第四章　固有类型

程序需要获取计算所需的原始数据, 并保存计算的结果. 程序可以从文件中获取数据, 并保存结果到文件中, 但只有文件能干这个事的话, 程序设计就会非常繁复, 并且因为程序读写文件比较费时, 程序运行效率会很差. 程序需要另外在电脑内存[1]中创立 “临时文件” 一样的东东来从中获取数据并保存结果到其中. 这样的东东就是变量 (variable) 和常量 (constant), 而其中常量又分为字面常量 (literal constant) 和具名常量 (named constant). 变量和常量的区别是变量中的数据可以被修改而常量中的数据不可以被修改. 字面常量没有名称, 变量和具名常量则有变量名和具名常量名.[2]

Fortran 规定变量和常量都属于数据实体 (data entity). 每个数据实体都有若干 “决定其用途的特性”, 称为属性[3] (attribute), 而其中又有一个无比特殊的属性, 称为类型[4] (type). 类型分为固有类型 (intrinsic type) 和派生类型 (derived type), 固有类型在本章中讲, 派生类型在第**??**章中讲.

Fortran 的固有类型分五种: 整型 (integer type), 实型 (real type), 复型 (complex type), 字符型 (character type) 和逻辑型 (logical type). 整型, 实型和复型都是数字型 (numeric type). 每种固有类型都有与其对应的类型参量[5] (type parameter). 类型参量分为种别类型参量 (kind type parameter) 和长度类型参量 (length type parameter), 简称种别和长度. 所有 Fortran 固有类型都有相应的种别, 字符型另有长度.

Fortran 是静态类型语言 (statically typed language), 因为 Fortran 程

[1] 连内存是什么都不懂的同学赶紧自己恶补电脑知识.

[2] 这话儿其实不太对. 安安仔细查阅标准解释文档, 发现里头 “变量” 的定义和一般人对变量的理解不完全一致, 结果是有些变量不可能有变量名. 但如果那样定义 “变量”, 安安就要费老鼻子劲儿才能把后面的内容讲清楚了, 所以同学们暂且就认为变量都有变量名吧!

[3] 特别提醒, Fortran 中的属性和 Python 中的属性不是一个东东.

[4] 类型过于特殊, 以至于习惯上不认为其是属性之一, 但 Fortran 标准明确规定其为属性.

[5] 类型参量过于特殊, 以至于习惯上不认为其是属性之一, 但 Fortran 标准明确规定其为属性.

序执行时变量和常量的属性会被定死不变 (偶有例外). 变量和具名常量需要由我们用声明/说明[6] (declaration/specification) 来创建并规定其名称和属性, 字面常量则由编译器创建. 声明的语法变化多端, 但同学们莫要被它吓破了胆, 我们先学其基本形式, 以后再学基本形式的变形. 声明的基本形式如下.

```
[type-with-param], [attr_1], ..., [attr_m] &
    :: [name_1], ..., [name_n]
```

基本形式的解释如下.

- `[name_1], ..., [name_n]` 都是名称, 表示创建 `n` 个数据实体, 名称分别为 `[name_1], ..., [name_n]`.
- `[type-with-param]` 规定创建的 `n` 个数据实体共同的类型, 另可规定创建的 `n` 个数据实体共同的类型参量. 若 `[type-with-param]` 没规定创建的 `n` 个数据实体共同的类型参量, 则创建的 `n` 个数据实体共同的类型参量为默认类型参量. 默认种别由器规决定. 默认长度为 1.
- `[attr_1], ..., [attr_m]` 规定创建的 `n` 个数据实体共同的 `m` 个属性. `m` 可以是 0. `[attr_1], ..., [attr_m]` 中如果有 `parameter`, 则创建的 `n` 个数据实体都是具名常量并且都有 parameter 属性, 否则创建的 `n` 个数据实体都是变量并且都没有 parameter 属性.

同学们读了上面一大堆抽象的描述后估计已经想卸载 Fortran 编译器了, 我赶紧来个示例让同学们缓一缓.

```
program main
    implicit none
    real :: m, q
    real, dimension(:), allocatable :: v
end program main
```

示例第三行的解释如下.

- `[name_1], ..., [name_n]` 是 `m, q`, 表示创建 2 个数据实体, 名称分别为 `m` 和 `q`.

[6] 这俩概念有点区别但区别不大, 我将混用.

- `[type-with-param]` 是 `real`, 表示数据实体 `m` 和 `q` 的类型为实型, 类型参量为默认类型参量.

- `[attr_1], ..., [attr_m]` 没有 (`m` 是 0), 表示数据实体 `m` 和 `q` 没有其他属性, 也就没有 parameter 属性, 因此数据实体 `m` 和 `q` 是变量.

示例第四行的解释如下.

- `[name_1], ..., [name_n]` 是 `v`, 表示创建 1 个数据实体, 名称为 `v`.

- `[type-with-param]` 是 `real`, 表示数据实体 `v` 的类型为实型, 类型参量为默认类型参量.

- `[attr_1], ..., [attr_m]` 是 `dimension(:), allocatable`, 其中第 1 个 `dimension(:)` 表示数据实体 `v` 有 dimension 属性 (7.3 节介绍), 第 2 个 `allocatable` 表示数据实体 `v` 有 allocatable 属性 (4.4 节和 7.3 节介绍), 所以没有 parameter 属性, 因此数据实体 `v` 是变量.

这个示例中我没写具名常量的声明, 是因为具名常量的声明得用变种形式, 需放在 5.1 节中讲.

如果我们需要声明种别, 我们就得做点准备工作, 那就是把种别使用 (use) 到程序里. 使用种别的形式如下.

```
use, intrinsic :: iso_fortran_env, &
    only: [only_1], ..., [only_n]
```

其中 `[only_1], ..., [only_n]` 中的每一个都是 `[kind]` 或 `[alias] => [kind]`, `[kind]` 表示使用名为 `[kind]` 的种别, `[alias] => [kind]` 表示使用名为 `[kind]` 的种别, 但用别名 `[alias]` 表示. 示例如下.

```
program main
    use, intrinsic :: iso_fortran_env, &
        only: real32, dp => real64
    implicit none
    real(real32) :: m, q
    real(dp) :: v
end program main
```

示例第二行到第三行使用了种别 `real32` 和 `real64`, 并给 `real64` 取别名 `dp`. 示例第五行声明了种别为 `real32` 的 `m` 和 `q`. 示例第六行声明了种别为 `real64` 的 `v`.

Fortran 程序的每一行, 除了注释和某个本笔记里不想讲的东东外, 都是语句 (statement). Fortran 语句的位置不能乱放, 同学们现在需要记的规则是, 在每一个程序单元内, 以 `use` 做开头的使用语句放最前面, `implicit none` 放第二个, 声明语句放第三个, 其他称为执行语句的语句放最后面.

本笔记中的所有示例里都有个 `implicit none`, 这是个 Fortran 避坑大法咒, 有了它, 老师用来折磨同学们的 I–N 隐式规则就被禁掉了. 请同学们务必在每个程序单元里都加上它.

规范 16 *在程序的每个程序单元里都加上* `implicit none`.

4.1 整型

整型数据实体代表整数. 默认种别的整型字面常量长得和整数一样, 但不能加小数点及之后的 `0`. 下面这个程序使用了默认种别的整型字面常量 `-1234567890`, 它表示 -1234567890.

```
program main
    implicit none
    print *, -1234567890
end program main
```

整型用 `integer` 声明. 下面这个程序声明了默认种别的整型变量 `i`.

```
program main
    implicit none
    integer :: i
end program main
```

整型种别有 `int8`, `int16`, `int32` 和 `int64`. Ifx 和 Gfortran 规定的默认种别都是 `int32`. 原则上种别 `intn` 的 `n` 越大, 可存入数据实体中的整数的范围越广. 说明整型种别的方式和 4.2 节介绍的说明实型种别的方式一样, 但通常情况下没必要说明整型种别, 直接使用默认种别即可.

4.2 实型

实型数据实体代表实数, $+\infty$, $-\infty$ 和 NaN. NaN 称为非数 (Not a Number), 表示数据或计算结果不正常而且还不是 $\pm\infty$, $0/0$ 的结果就是 NaN ($\pm 1/0$ 的结果则是 $\pm\infty$). 默认种别的实型字面常量长得和实数一样, 最后可带 `e[n]`, 其中 `[n]` 是默认种别的整型字面常量, 来表示 $\times 10^n$. 下面这个程序使用了默认种别的实型字面常量 `6.62607015e-34`, 它照道理应该表示 $6.62607015 \times 10^{-34}$, 然而实际上表示的是个非常近似于 $6.62607015 \times 10^{-34}$ 的数, 这是因为计算机用的是二进制, 不一定能精确存储十进制的实数.

```
program main
    implicit none
    print *, 6.62607015e-34
end program main
```

如果实型字面常量中没有小数点, 则必须附加 `e[n]`, 否则会变成整型字面常量. 如果实型字面常量中有小数点, 则小数点前和小数点后的数字可以省略一边, 若省略则为 `0`, 但不能都省略. 安安用实型字面常量时不会省略小数点前和小数点后的数字, 例如 `1.0` 不会写成 `1.`, `0.1` 不会写成 `.1`, 因为日常写小数时不太有这样省略的写法, 程序里写成这样, 可读性会下降一丢丢. 但是有很多人在程序里用这省略的写法, 并不觉得有什么问题, 所以安安不好把不省略小数点前后的数字定成规范.

实型用 `real` 声明. 下面这个程序声明了默认种别的实型变量 `h`.

```
program main
    implicit none
    real :: h
end program main
```

实型种别有 `real16`, `real32`, `real64` 和 `real128`, 分别称为半精度, 单精度, 双精度和四精度, 不过俺手里的 Ifx 和 Gfortran 都 out 了, 不认得 `real16`. Ifx 和 Gfortran 规定的默认种别都是 `real32`. 原则上种别 `realn` 的 `n` 越大, 可存入数据实体中的实数的范围越广, 精度越高, 所以照道理用 `real128` 最好, 不过 `real128` 出现得比较晚, 天文人历来用的都是 `real64`, 我们沿用即可.

声明实型数据实体的种别时在 `real` 后加 `([kind])` 或 `(kind=[kind])`, 和 `real` 间可以有空格但一般不加, 一般都用第一种方式, 虽然这种方式和

4.4 节介绍的声明字符型数据实体的长度的方式长得一样, 总让人觉得会混淆. 说明实型字面常量的种别时在它屁股后面加尾巴 _[kind]. 以上的 [kind] 都是种别名 (如果取了别名则必须用别名). 下面这个程序声明了 real32 种别的实型变量 g 和 real64 种别的实型变量 c, 使用了 real32 种别的实型字面常量 6.6743e-11_real32 (程序里必须写成 6.6743e-11_sp) 和 real64 种别的实型字面常量 299792458.0_real64.

```
program main
    use, intrinsic :: iso_fortran_env, &
        only: sp => real32, real64
    implicit none
    real(sp) :: g
    real(real64) :: c
    print *, 6.6743e-11_sp
    print *, 299792458.0_real64
end program main
```

说明实型种别有两种不规范的写法. 第一种是用 4 和 8 代替 real32 和 real64, 第二种是声明双精度实型数据实体时用 double precision 代替 real(real64), 使用双精度实型字面常量时用 d 代替 e. 这两种不规范的写法同学们不许用.

4.3 复型

复型数据实体有实部和虚部, 两者都是实型数据实体, 如果实部和虚部都代表实数, 那么复型数据实体自然代表复数. 复型用 complex 声明. 复型种别和实型种别一样, 声明的方式也相同. 下面这个程序声明了 real128 种别的复型变量 tilde_s.

```
program main
    use, intrinsic :: iso_fortran_env, only: real128
    implicit none
    complex(real128) :: tilde_s
end program main
```

复型字面常量长成 `([real], [imag])` 的样子, 其中 `[real]` 和 `[imag]` 是整型或实型常量, 分别代表实部和虚部. 下面这个程序使用了复型字面常量 `(0.0, 1.0)`, 它表示 $0+1i$.

```
program main
    implicit none
    print *, (0.0, 1.0)
end program main
```

不能在复型字面常量的屁股后面加尾巴来说明复型字面常量的种别, 复型字面常量的种别由构造它时使用的代表实部和虚部的常量来确定.

- 如果构造时实部和虚部都用实型常量, 则复型字面常量的种别是两个实型常量的种别中能提供更高数值精度的那个 (若精度一样则由编译器自己随便挑一个).
- 如果构造时实部和虚部一个用整型常量, 另一个用实型常量, 则复型字面常量的种别是那个实型常量的种别.
- 如果构造时实部和虚部都用整型常量, 则复型字面常量的种别是默认实型种别.

下面这个程序中, 第一个复型字面常量的种别是默认实型种别 (对 Ifx 和 Gfortran 而言是 `real32`), 第二个复型字面常量的种别是 `real16`, 第三个复型字面常量的种别原则上是 `real128`, 最后一个复型字面常量的种别的表示方法不对, 程序将报错.

```
program main
    use, intrinsic :: iso_fortran_env, &
        only: hp => real16, qp => real128
    implicit none
    print *, (0, 1)
    print *, (0.0_hp, 1)
    print *, (0.0_hp, 1.0_qp)
    print *, (0.0, 1.0)_qp ! Wrong!
end program main
```

复型数据实体的实部和虚部的种别总是和复型数据实体本身的种别相同, 例如 `(0.0_real64, 1.0_real32)` 的实部的种别是 `real64`, 虚部的种

别也是 `real64` (而不是 `real32`). 任意复型数据实体 `[z]`, 其实部和虚部可分别用 `real([z])` 和 `aimag([z])` 获取, 例如 `real((0, 1))` 是 `0.0`, `aimag((0, 1))` 是 `1.0`.

4.4 字符型

字符型数据实体代表字符串 (character string). 字符串就是排成一串的字符. 字符串的长度可以是 0. 字符型字面常量用夹在两个 `'` 或 `"` 之间的字符串表示. 下面这个程序使用了两个字符型字面常量, 第一个代表长度为 0 的字符串 “”, 第二个代表字符串 “character” (两端的引号不属于字符串).

```
program main
    implicit none
    print *, ''
    print *, "character"
end program main
```

字符型字面常量中夹在引号之间的字符都属于字符串的规则优先于其他许多语法规则, 所以字符型字面常量中大小写效果不同, 空格不可有可无, `!` 和 `&` 也不表示注释和续行. 示例如下.

```
program main
    implicit none
    print *, 'QWERTY'
end program main

program main
    implicit none
    print *, 'qwerty'
end program main

program main
    implicit none
    print *, '!&'
end program main
```

```
program main
    implicit none
    print *, '!     &'
end program main
```

如果在夹于 ' 间的字符串中又出现 ', 在夹于 " 间的字符串中又出现 ", 编译器就看不懂代码了. Fortran 特别规定在夹于 ' 间的字符串中需要用 '' 表示 ', 在夹于 " 间的字符串中需要用 "" 表示 ". 示例如下. 另外请注意老古董 Fortran 的这一规定和 C 的规定不同, 并且其他绝大多数语言的规定都是照着 C 来的.

```
program main
    implicit none
    print *, 'GasinAn said, "I love using ''wheels'','
    print *, "especially the 'liber' ones."""
end program main
```

Fortran 字符串中可以有换行字符. 电脑里的字符本质上是小段数据, 而换行字符则是电脑操作系统规定的专门用来表示前面的字符和后面的字符分别在上下两行的数据. Windows 和 Linux 的换行字符还不一样, Windows 连用回车符和换行符表示换行, 而 Linux 单用换行符表示换行. 不好玩的是, Fortran 字符型字面常量中没法儿加换行字符 (C 没这问题). 在 Fortran 字符串中加入换行字符的方法放在 5.2.2 小节介绍.

字符型用 `character` 声明. 通常情况下没必要说明字符型种别, 直接使用默认种别即可. 字符型字面常量可以直接表示自己的长度. 声明字符型数据实体的长度时在 `character` 后加 `([len])` 或 `(len=[len])`, 和 `character` 间可以有空格但一般不加, 一般都用第一种方式, 虽然这种方式和 4.2 节介绍的声明实型数据实体的种别的方式长得一样, 总让人觉得会混淆. `[len]` 可以是整型常量, `*` 或 `:`.

如果 `[len]` 是整型常量, 则直接表示字符型数据实体的长度, 也就是字符型数据实体表示的字符串的长度. 下面这个程序声明了一个长度为 1×10^{12} 的字符型变量 `very_long_char`.

```
program main
    use iso_fortran_env, only: l => int64
    implicit none
```

```
    character(1000000000000_1) :: very_long_char
end program main
```

但下面这个程序不成, 因为 `1e12` 是实型的.

```
program main
    implicit none
    character(1e12) :: very_long_char
end program main
```

任意字符型数据实体 `[c]`, 其长度可用 `len([c])` 获取, 例如 `len('')` 是 `0`.

如果 `[len]` 是 `*`, 则表示声明的是假定长度 (assumed-length) 字符型数据实体. 假定长度字符型数据实体涉及些比较高深的知识, 需放在 5.1 节和 8.2.3 小节中讲.

如果 `[len]` 是 `:`, 则表示声明的是延迟长度 (deferred-length) 字符型数据实体, 此时必须附带地用 `allocatable` 加上 allocatable 属性. 延迟长度字符型数据实体的长度在声明后未定. 任意延迟长度字符型数据实体 `[c_ds]`, 我们可以用形如 `allocate(character([len]) :: [c_ds])` 的 allocate 语句让 `[c_ds]` 的长度由未定变成 `[len]`, 用形如 `deallocate([c_ds])` 的 deallocate 语句让 `[c_ds]` 的长度由确定长度变成未定. 这也意味着延迟长度字符型数据实体只能是变量. 示例如下.

```
program main
    implicit none
    character(:), allocatable :: deferred_len_char
    ! Now len(deferred_len_char) is undefined.
    allocate(character(1000000) :: deferred_len_char)
    ! Now len(deferred_len_char) is 1000000.
    deallocate(deferred_len_char)
    ! Now len(deferred_len_char) is undefined.
    allocate(character(0) :: deferred_len_char)
    ! Now len(deferred_len_char) is 0.
end program main
```

使用 allocate 语句和 deallocate 语句需遵守的规范放在 7.3 节讲.

可以对字符串进行切片 (slicing) 操作. 切片就是在字符串后加上个形如 `(nl:nu)` 的东东, 表示用字符串的第 `nl` 到第 `nu` 个字符组成一个新字符

串. nl 和 nu 必须是整型数据实体. nl 和 nu 都可以不写. 如果 nl 不写, nl 就等于 1. 如果 nu 不写, nu 就等于旧字符串的长度. 示例如下.

```
program main
    implicit none
    print *, '123456789'(3:7) ! 34567
    print *, '123456789'(:7)  ! 1234567
    print *, '123456789'(3:)  ! 3456789
    print *, '123456789'(:)   ! 123456789
end program main
```

4.5 逻辑型

逻辑型数据实体代表逻辑值, 逻辑型字面常量只有 .true. 和 .false., 分别代表 “真” 和 “假”. 逻辑型用 logical 声明. 通常情况下没必要说明逻辑型种别, 直接使用默认种别即可. 示例俺懒得写了, 留给同学们作练习.

第五章　赋值与运算

5.1　赋值

Fortran 中的赋值 (assignment), 其一般含义可总结为把数据拷贝到一个数据实体中, 分为四种: (普通的) 固有赋值 (intrinsic assignment), 掩码数组赋值 (masked array assignment), 指针赋值 (pointer assignment), 和超载赋值 (defined assignment). 本节只讲固有赋值的最简单情形, 固有赋值的其他情形一和掩码数组赋值一起放在 7.5 节讲, 固有赋值的其他情形二和指针赋值一起放在 **??** 节讲, 超载赋值放在 **??** 节讲.

固有赋值的形式是 `[var] = [expr]`. `[var]` 可以是很多东东, 最常见的情况是 `[var]` 是一个变量, 其他情况安安将一一列举. `[expr]` 是一个表达式 (expression), 放在 5.2 节详细介绍. 运行 `[var] = [expr]` 时, 程序会先对 `[expr]` 进行求值 (evaluation) 并得到一个结果 (result), 此结果是一个数据实体, 然后将得到的结果中的数据拷贝到 `[var]` 里去. 下面的例子中, 第四行对表达式 `10.0` 进行求值, 得到的结果中的数据 10.0 拷贝到 `a` 里去, 第五行对表达式 `a + a*a - a/a` 进行求值, 得到的结果中的数据 109.0 还拷贝到 `a` 里去, 所以最后 `a` 里的数据是 109.0.

```
program main
    implicit none
    real :: a
    a = 10.0
    a = a + a*a - a/a
    print *, a
end program main
```

为避免后面讲解时啰里巴嗦的, 安安需要先啰嗦几句. 如果数据实体中

有 Fortran 程序拷贝到其中存储的数据, 安安就称数据实体中存储的数据为数据实体的值 (value), 例如上面的程序第四行运行后 `a` 的值是 10.0. 如果数据实体的值和一个字面常量的值相同, 安安也称数据实体的值为那个字面常量, 例如上面的程序第四行运行后 `a` 的值是 `10.0`. 安安还把对表达式进行求值得到的结果的值简称为表达式的值, 例如上面的程序第五行运行时表达式 `a + a*a - a/a` 的值是 `109.0`.

因为赋值会修改数据实体的值, 所以字面常量是不能被赋值的, 例如下面这个程序是不成的.

```
program main
    implicit none
    10.0 = 10.0
end program main
```

而具名常量也是不能被赋值的, 这就出现了具名常量的值如何确定的问题. 具名常量的值需要且必须由初始化 (initialization) 确定, 所谓初始化实际上就是在声明的同时赋值[1], 方法是在声明时在具名常量名 `[name]` 后加 `= [value]`, 其中 `[value]` 是常量. 下面这个程序声明了具名常量 `ten`, 并令其值为 `10.0`

```
program main
    implicit none
    real, parameter :: ten = 10.0
end program main
```

如果是声明字符型具名常量, 那么初始化时还有个长度匹配的问题. 我们可以在声明时把长度写成 `*`, 这样字符型具名常量的长度直接由字面常量 `[value]` 确定为 `[value]` 的长度, 非常方便. 下面这个程序声明了具名常量 `author`, 它的长度直接由字面常量 `'GasinAn'` 确定为 7.

```
program main
    implicit none
    character(*), parameter :: author = 'GasinAn'
    print *, author, len(author)
end program main
```

[1]这话儿其实不太对, 因为按标准解释文档的定义, 初始化和赋值是不同的东东, 但通常人们都会觉得初始化就是在声明的同时赋值, 同学们就这么认为拉倒吧!

另外, 如果一个变量在程序运行的时候第一次被赋值, 我们也称这个变量被初始化, 而如果一个变量在程序运行的时候没被赋值, 我们则称这个变量是未定义的. 在下面这个程序中, a 有被初始化, 而 b 没被初始化, 从始至终一直是未定义的.

```
program main
    implicit none
    real :: a, b
    a = 0.0
    print *, a, b
end program main
```

但同学们会发现上面这个程序运行到最后, a 和 b 里头都有数据. 其中, a 里头的数据是我们用程序清清楚楚地指挥电脑赋给 a 的有意义的数据, 而 b 里头的数据是我们用程序指挥电脑创建 b 的时候 b 会随机拥有的无意义的数据. 如果一个变量是未定义的, 存在那个变量中的数据是无意义的, 那么取用那个变量中的数据也是无意义的, 程序必然有毛病. 然而 Fortran 和 C 这样古早的语言, 它们的编译器不一定会检查变量在取用时是否未定义, 程序因此容易出现检查不出的毛病, 同学们只好自己小心. Python 吸取教训, 不存在这个问题, 例如在 Python 3 中, 如果 b 未定义, 那么 `print(b)` 会直接报错.

当我们要用延迟长度字符型变量时, 我们本可能需要经常使用 allocate 语句和 deallocate 语句, 像下面的程序那样, 非常麻烦.

```
program main
    implicit none
    character(:), allocatable :: char
    allocate(character(len('allocated')) :: char)
    char = 'allocated'
    print *, char, len(char), len('allocated')
    deallocate(char)
    allocate(character(len('reallocated')) :: char)
    char = 'reallocated'
    print *, char, len(char), len('reallocated')
end program main
```

Fortran 为我们简化了语法. 我们可以直接对延迟长度字符型变量进行赋值. 如果延迟长度字符型变量 `[var]` 的长度未定, 那么 `[var] = [expr]` 时电脑会自动用 allocate 语句把 `[var]` 的长度定成 `[expr]` 的长度. 如果延迟长度字符型变量 `[var]` 的长度已定但不等于 `[expr]` 的长度, 那么 `[var] = [expr]` 时电脑会自动用 deallocate 语句让 `[var]` 的长度未定, 然后再用 allocate 语句把 `[var]` 的长度定成 `[expr]` 的长度. 所以上面的程序可简化成下面的程序, 非常方便.

```
program main
    implicit none
    character(:), allocatable :: char
    char = 'allocated'
    print *, char, len(char), len('allocated')
    char = 'reallocated'
    print *, char, len(char), len('reallocated')
end program main
```

Fortran 是强类型语言 (strongly typed language), 因为 Fortran 程序赋值时用于赋值的和被赋值的数据实体的类型要一致, 数字型数据实体只能赋值给数字型数据实体, 字符型数据实体只能赋值给字符型数据实体, 逻辑型数据实体只能赋值给逻辑型数据实体. 不过官规虽是这么说, Ifx 和 Gfortran 却会在某些情况下允许数字型数据实体和逻辑型数据实体相互赋值, 根据规范 1, 同学们不许这么做.

数字型数据实体赋值给数字型数据实体的时候, "=" 两边的类型或种别可能不一样, 这时自然有个类型和种别转化的问题. 因为 Fortran 是静态类型语言, 所以赋值时 "=" 左边数据实体的类型和种别不会变, 而右边数据实体的类型和种别会临时转化成左边数据实体的类型和种别. 安安努力查阅标准解释文档后, 把 Fortran 数字型数据实体类型和种别转化的规则总结成表 5.1, 其中的术语解释如下:

- 值相等: 转化后的值和转化前的值相等.
- 值近似: 转化后的值和转化前的值近似相等, 如何近似则由编译器自己决定.
- 实部近似虚部 0: 转化后的值的实部和转化前的值的实部近似相等, 转化后的值的虚部为 0, 如何近似则由编译器自己决定.

- 向 0 取整: 转化后的值由转化前的值向 0 取整而成. 即转化后的值的绝对值是不大于转化前的值的绝对值的最大的整数, 转化后的值的符号和转化前的值的符号相同.

- 二次转化: 先转化成实型, 再转化成整型.

转化前 \ 转化后	整型	实型	复型
整型	值相等	值近似	实部近似虚部 0
实型	向 0 取整	值近似	实部近似虚部 0
复型	二次转化	实部近似虚部 0	值近似

表 5.1: Fortran 数字型数据实体转化表

同学们莫要被这表吓破了胆, 我们可以简化记忆. 首先, 编译器们在自己决定 “如何近似” 的时候, 总是尽可能地近似 (敢不如此我们就卸载), 以至于引入的计算误差没人考虑, 因此我们可以把所有的 “近似相等” 直接当成 “相等”. 这样, 表 5.1 就简化成表 5.2.

转化前 \ 转化后	整型	实型	复型
整型	值相等	值相等	值相等
实型	向 0 取整	值相等	值相等
复型	二次转化	实部近似虚部 0	值相等

表 5.2: 简化 Fortran 数字型数据实体转化表

最后同学们会发现只需记下面的内容即可:

- 实型转整型: 向 0 取整.

- 复型转实型: 取实部.

- 复型转整型: 先转实型, 再转整型.

- 其他: 直接令值相等.

我们用一个示例来强化一下认识. 在下面的程序中, 第七行应使 `b` 的值为 -5.5, 然后第八行需要类型转化. 看 "`=`" 右边 `b` 是实型, 左边 `a` 是整型, 实型转整型向 0 取整, -5.5 向 0 取整是 -5, 所以 `a` 的值为 -5. 但这里同学们常犯一个错误, 就是认为第八行后 `b` 的值也变为 -5 了, 而实际上右边的类型和种别是临时转化成左边的类型和种别, 所以第八行后 `b` 的类型和种别不会变, 值也不会变, 还是 -5.5.

```
program main
    implicit none
    integer :: a
    real :: b
    b = -5.5
    a = b
    print *, a, b
end program main
```

默认情形下 Fortran 和 C 的的类型和种别转化是 "隐式" 的转化, 意思是, 在类型和种别转化的时候 Fortran 程序和 C 程序里没有一个明确的标记来说明, 需要仔细分析各个数据实体的类型和种别然后判断出有转化. 于是乎老师就可以借机整出一堆怪题来疯狂折磨同学们了, 并且实际人们用 Fortran 和 C 干活的时候, 也很容易没看出程序某某地方有类型和种别转化. Python 没有这个问题, 因为 Python 不是静态类型语言.[2] 这样就有必要在 Fortran 程序和 C 程序里使用 "显式" 的转化. 对 Fortran 而言, 在实型转整型, 复型转实型, 复型转整型的时候, 转化前后的值可能会不一样, 我们应当用 `int([entity])/real([entity])` 来说明数据实体 `[entity]` 被转化成整型/实型. 其他情形下, 转化前后的值一样, 就没必要啰嗦了. 例如上面的程序应该改写成下面的程序, 这样看到明确的标记 `int` 就可以直接判断出 `b` 被转化成整型了. 仍要注意类型和种别转化只是临时的, 第八行后 `b` 的值还是 -5.5.

[2] 详细说来, Fortran 和 C 因为是静态类型语言, "`=`" 左边的类型和种别不能变, 所以赋值时是 "右配合左", 把 "`=`" 右边的类型和种别 (临时) 变成左边的类型和种别, 这样在实型转整型, 复型转实型, 复型转整型的时候, "`=`" 左右两边的值可能是不一样的. 而 Python 因为不是静态类型语言, "`=`" 左边的类型和种别能变, 所以赋值时是 "左配合右", 把 "`=`" 左边的类型和种别变成右边的类型和种别, 这样 "`=`" 左右两边的值永远是一样的, 不会折磨人, 所以 Python 大家都爱用. 然而这并不意味着 Python 无敌了, 正因为 Python "`=`" 左边的类型和种别能变, 所以电脑在运行 Python 程序的时候老是要查看变量的类型和种别到底是什么, 这样运行 Python 程序的时候就快不起来了…

```
program main
    implicit none
    integer :: a
    real :: b
    b = -5.5
    a = int(b)
    print *, a, b
end program main
```

规范 17 *在实型转整型, 复型转实型, 复型转整型的时候, 用* `int([entity])` */*`real([entity])` *来说明数据实体* `[entity]` *被转化成整型/实型.*

字符型数据实体赋值给字符型数据实体的时候, "=" 两边的长度可能不一样, 这时自然有个长度转化的问题. 如果 "=" 左边字符串的长度小于右边字符串的长度, 则临时把右边字符串尾巴多出的部分砍掉, 然后赋值. 如果 "=" 左边字符串的长度大于右边字符串的长度, 则临时在右边字符串尾巴处补上空格让长度一样, 然后赋值. 示例如下. 仍要注意长度转化只是临时的.

```
program main
    implicit none
    character(4) :: sc
    character(5) :: nc
    character(6) :: lc
    sc = 'hello'
    print *, '"', sc, '"'
    nc = 'hello'
    print *, '"', nc, '"'
    lc = 'hello'
    print *, '"', lc, '"'
end program main
```

最后介绍 "对变量的一部分赋值". 对任意复型变量 `[z]`, 我们可以直接对 `[z]%re`/`[z]%im` 赋值以改变 `[z]` 的实部/虚部而不改变 `[z]` 的虚部/实部. 示例如下.

```
program main
```

```
    implicit none
    complex :: i
    i%re = 0.0
    i%im = 1.0
    print *, i
end program main
```

在 4.3 节我们介绍过可以用 `real([z])/aimag([z])` 获取 `[z]` 的实部/虚部, 但 `real([z])/aimag([z])` 不能用在 "=" 左边. `[z]%re/[z]%im` 虽然可能可以用在 "=" 右边, 但同学们去用的时候, 程序就经常会报错, 这是由于 Fortran 的语法有缺陷, 而在 "=" 右边用 `real([z])/aimag([z])` 就永远不会出问题. Python 的语法简洁优美, 类似 `[z]%re/[z]%im` 的语法用在 "=" 左右两边都没问题, 而类似 `real([z])/aimag([z])` 的语法只能用在 "=" 右边. C 就麻烦了, 复型都不是基本类型, 所以用复型的时候老费劲儿了…

对任意字符型变量 `[c]`, 我们可以直接对 `[c](nl:nu)` 赋值以改变 `[c]` 的第 `nl` 到第 `nu` 个字符而不改变其他字符. 这个语法和字符串切片的语法在形式上是一样的. `nl` 和 `nu` 必须是整型数据实体. `nl` 和 `nu` 都可以不写. 如果 `nl` 不写, `nl` 就等于 `1`. 如果 `nu` 不写, `nu` 就等于 `[c]` 的长度. 示例如下. 这个示例需要解释一下, 第五行我们把 `'HELLO, WORLD!'` 赋给 `hello_world` 的第 `1` 到第 `1` 个字符, 但 `hello_world` 的第 `1` 到第 `1` 个字符长度只为 1, 所以根据字符串赋值规则, `'HELLO, WORLD!'` 的尾巴被砍掉, 只剩最前面的 `'H'` 赋给 `hello_world` 的第 `1` 到第 `1` 个字符.

```
program main
    implicit none
    character(13) :: hello_world
    hello_world = 'hello, world!'
    hello_world(1:1) = 'HELLO, WORLD!'
    print *, hello_world
end program main
```

Python 的语法简洁优美, 但 Python 不允许直接修改字符串的一部分, 同学们需另寻他法. C 就可怕了, 详细说来, C 的字符串有两种定义法, 方法一定义的字符串能直接修改整个字符串但不能直接修改字符串中的单个字符, 方法二定义的字符串能直接修改字符串中的单个字符但不能直接修改整个字符串…

但是, 这里有个小坑. 来看下面这个程序. 在这个程序中, 我们对延迟长度字符型变量 `char` 的一部分 `char(:)` 赋值. 我们会期待电脑将自动用 allocate 语句和 deallocate 语句来帮我们定好 `char` 的长度, 然而, 因为语法上 "=" 左边不是 `char`, 而是 `char` 的一部分 `char(:)`, 所以电脑不会自动用 allocate 语句和 deallocate 语句来帮我们定好 `char` 的长度 (即使 `char(:)` 代表 `char` "从头到尾的一部分", 和 `char` 本身是相当的). 这个程序 Gfortran 运行时会报错, Ifx 不会报错, 但运行后屏幕上出现的结果怪怪的, 原因应该是 Ifx 在 `char` 长度未被确定过的情况下, 会自己认定 `char` 的长度是 0, 然后在赋值时又根据赋值规则, 把 `'allocated'` 和 `'deallocated'` 的尾巴砍掉, 但 `char` 的长度是 0, 所以砍完就变成 `''` 了...

```
program main
    implicit none
    character(:), allocatable :: char
    char(:) = 'allocated'
    print *, char, len(char), len('allocated')
    char(:) = 'reallocated'
    print *, char, len(char), len('reallocated')
end program main
```

再来两个例子让同学们把这个问题弄清楚. 在下面这个例子中, 第四行 `char` 的长度被定了, 然后第六行和第八行 "=" 左边是 `char`, 这时遵循简化赋值语法, "=" 左边字符串的长度被重定, 来和 "=" 右边字符串的长度一致.

```
program main
    implicit none
    character(:), allocatable :: char
    char = '******'
    print *, char, len(char), len('******')
    char = 'allocated'
    print *, char, len(char), len('allocated')
    char = 'reallocated'
    print *, char, len(char), len('reallocated')
end program main
```

而在下面这个例子中, 第四行 `char` 的长度被定了, 然后第六行和第八行 "="

左边是 `char(:)`, 这时不遵循简化赋值语法, "`=`" 左边字符串的长度不被重定, 而 "`=`" 右边字符串被砍尾巴/补空格来和 "`=`" 左边字符串的长度一致.

```
program main
    implicit none
    character(:), allocatable :: char
    char = '******'
    print *, char, len(char), len('******')
    char(:) = 'allocated'
    print *, char, len(char), len('allocated')
    char(:) = 'reallocated'
    print *, char, len(char), len('reallocated')
end program main
```

5.2 运算

表达式由操作数 (operand), 运算符 (operator), 和左小括号 "`(`" 与右小括号 "`)`" 组成. 同学们其实平常就经常见到表达式, 最常见的就是代数式了, 例如代数式 $(a+b)\times(a-b)$, 其中 a 和 b 就是操作数, 而 $+$ 和 $\times$ 则是运算符. Fortran 中所有操作数都是数据实体[3], 所有数据实体都可以是操作数, 而表达式本身也是数据实体, 可以作为操作数出现在更大的表达式中. 同学们其实平常也经常见到这种情况, 例如在 $(a+b)\times(a-b)$, $a+b$ 和 $a-b$ 就是作为表达式的操作数.

运算符总是代表运算 (operation), 例如代数中运算符 $+$ 代表加法运算, 而运算符 $\times$ 代表乘法运算. Fortran 中的运算分为固有运算 (intrinsic operation) 和超载运算 (defined operation), 本节只讲固有运算, 超载运算放在 **??** 节讲. 固有运算又分为四种: 算数运算 (numeric operation), 字符运算 (character operation), 逻辑运算 (logical operation), 关系运算 (relational operation).

如果运算符 `[op]` 把一个数据实体 `[q]` 变成另一个数据实体 `[op][q]`, 我们就称运算符 `[op]` 为一个一元运算符 (unary operator). 如果运算符 `[op]` 把两个数据实体 `[q1]` 和 `[q2]` 变成另一个数据实体 `[q1][op][q2]`,

[3]这话儿其实不太对, 因为按标准解释文档的定义, 有些操作数不被定义成数据实体, 但这些操作数和数据实体也没什么区别, 同学们就直接认为是数据实体拉倒吧!

我们就称运算符 `[op]` 为一个二元运算符 (binary operator). 同学们平常也经常见到一元运算符和二元运算符, 例如在 $a-(-b)$ 中, 括号里的 $-$ 把 b 变成 $-b$, 所以是一元运算符, 而括号外的 $-$ 把 a 和 $(-b)$ 变成 $a-(-b)$, 所以是二元运算符.

在代数式中运算符有优先级, 例如在 $a+b\times c$ 中, $\times$ 比 $+$ 优先, 所以结果是 $a+(b\times c)$ 而不是 $(a+b)\times c$. Fortran 中的运算符也是如此, 在形如 `[q1][op12][q2][op23][q3]` 的表达式中, 如果 `[op12]` 和 `[op23]` 有优先级高低之分, 则优先级高的先算优先级低的后算, 如果 `[op12]` 和 `[op23]` 没有优先级高低之分, 则一般是和平常一样, 左边的 `[op12]` 先算右边的 `[op23]` 后算, 但有一个反例后面会讲.

在代数式中可以加括号, 来强行让括号里的表达式先算, 例如在 $(a+b)\times c$ 中是 $a+b$ 先算. Fortran 中也是如此, 但要注意, Fortran 中 (和几乎所有编程语言中) 都只能加小括号来强行让括号里的表达式先算, 代数式里的中括号和大括号在 Fortran 中都得用小括号替代.

大致说来, 如果一个表达式里只有常量, 这个表达式就是一个常量表达式 (constant expression). 之所以说 "大致说来", 是因为标准解释文档里常量表达式的定义复杂至极, 但同学们不用管它. 在需要使用常量的地方改用常量表达式看来都是可以的 (安安想不出反例), 例如下面两个程序都是没有问题的.

```
program main
    use iso_fortran_env, only: l => int64
    implicit none
    character(1000000000000_l) :: very_long_char
end program main
```

```
program main
    use iso_fortran_env, only: l => int64
    implicit none
    character(10_l**12_l) :: very_long_char
end program main
```

5.2.1 算数运算

一元算数运算符有两个: `+`, `-`, 二元算数运算符有五个: `+`, `-`, `*`, `/`, `**`, 它们的左边右边必须是数字型数据实体. 如果数据实体 `[a]` 的值为 a, 数据实体 `[b]` 的值为 b, 那么 `+[a]`, `-[a]`, `[a]+[b]`, `[a]-[b]`, `[a]*[b]`, `[a]/[b]`, `[a]**[b]` 的值 (在不考虑计算误差的情况下) 一般是 $+a$, $-a$, $a+b$, $a-b$, $a\times b$, $a\div b$, a^b, 但有重要的例外后面会讲. 因为 $\sqrt[b]{a}=a^{1/b}$, 所以就没必要给开方准备一个运算符了.

使用 `**` 的时候请注意, 如果 `[a]**[b]` 中的 `[a]` 和 `[b]` 都是实型的, 且 a 是负的, 那么高中数学课告诉我们 a^b ill defined, 所以程序会报错. 如果 `[a]**[b]` 中的 `[a]` 和 `[b]` 都是复型的, 那么复变函数课[4]告诉我们 a^b 是多值的, 所以程序会算 a^b 的主值.

另外 `**` 的运算顺序和其他运算符相反, 在形如 `[a]**[b]**[c]` 的表达式中, 是先算 `[b]**[c]` 而不是 `[a]**[b]`. 可以这么记: `([a]**[b])**[c]` 其实等于 `[a]**([b]*[c])`, 如果 `[a]**[b]**[c]` 是 `([a]**[b])**[c]`, 因为乘法比较简单, 电脑算乘法应该比算乘方快, 所以写 `[a]**[b]**[c]` 显得太傻了, 还不如直接写 `[a]**([b]*[c])`, 所以 `[a]**[b]**[c]` 应该是 `[a]**([b]**[c])`. 但 `**` 的运算顺序和其他运算符相反这件事很容易忘了, 所以请同学们不写 `[a]**[b]**[c]` 而写 `[a]**([b]**[c])`.

规范 18 *在表达式中运算符的运算顺序容易被误解的情况下, 用括号指明运算顺序.*

代数式中应该只出现类似 $a-(-b)$ 之类的东东, 不出现类似 $a--b$ 之类的东东. 类似地, Fortran 表达式里不允许连着出现两个算数运算符, 因此只允许写 `[a] - (-[b])` 之类的东东, 不允许写 `[a] - -[b]` 之类的东东 (虽然 Ifx 和 Gfortran 其实不完全禁止这么写).

算数运算符的运算顺序和代数一致, 首先是 `**`, 然后是 `*` 和 `/`, 然后是一元 `+` 和一元 `-`, 最后是二元 `+` 和二元 `-`, 好记得很. 但同学们先别高兴得太早, 请回答 `-2.0 ** 2.0` 是多少? 如果同学们觉得是 `4.0` 就上当啦, 答案是 `-4.0`, 因为根据算数运算符的运算顺序, 是 `**` 先算 `-` 后算, 所以即使 `-2.0 ** 2.0` 里 `-` 和 `2.0` 粘在一起, 好像 `-2.0 ** 2.0` 是 `(-2.0) ** 2.0` 一样, `-2.0 ** 2.0` 也还是 `-(2.0 ** 2.0)`. 特别说明, Fortran 规定 `**` 先算 `-` 后算, 这不是在坑大家, 因为形如 $-a^b$ 的代数式本来就是 $-(a^b)$ 而不是 $(-a)^b$.

[4]哎呀, 好像同学们还没上到复变函数课!?...

根据规范 8, 我们需要 “在程序中适当地加入空格”, 在表达式中有些地方加空格好, 有些地方不加空格好, 上面的 `-2.0 ** 2.0` 就是一个典型反例 (应该写成 `- 2.0**2.0` 或 `-(2.0**2.0)`, 最不济也应该写成 `-2.0**2.0`). 但是到底应不应该加空格, 判断的原则很复杂, 而且不同的人意见不一. 同学们可以参考本笔记, Fortran 官网上的代码示例和 Python 的 PEP 8 规范来决定是否加空格.

现在我们又要讨论烦人的类型和种别转化的问题了. 我们需要先定义类型和种别的高低. 我们定义类型从低到高依次为整型, 实型, 复型, 并定义整型种别是低种别, 实型种别是高种别. 对整型种别, 我们定义能表示的数的十进制表示的位数更少/多的种别为低/高种别, 这意味着 `intn` 的 `n` 更小/大的种别为低/高种别. 对实型种别, 我们定义能表示的数的十进制表示的精度更低/高的种别为低/高种别, 这意味着 `realn` 的 `n` 更小/大的种别为低/高种别. 以下 `[op]` 代表算符, `[q]`, `[q1]`, `[q2]` 代表数据实体.

- `[op][q]` 的类型/种别是 `[q]` 的类型/种别.
- `[q1][op][q2]` 的类型/种别是 `[q1]` 和 `[q2]` 的类型/种别中更高的类型/种别; 如果 `[q1]` 和 `[q2]` 的类型/种别一样高, 那么编译器自己决定 `[q1][op][q2]` 的类型/种别是 `[q1]` 和 `[q2]` 的类型/种别中的哪一个.
- 在计算 `[q1][op][q2]` 前, `[q1]` 和 `[q2]` 的类型/种别总是被临时转化成 `[q1][op][q2]` 的类型/种别.

一言以蔽之, 运算时类型和种别转化遵循着 “就高不就低” 的规则.

我们已经把运算时类型和种别转化的规则简洁凝练地总结出来了, 但同学们莫要高兴, 马上同学们就会遇到一个非常传统, 非常典型, 非常折磨人的大坑, 巨坑, 沓坑, 这个坑就是两个整型数据实体相除. 来看下面这个例子.

```
program main
    implicit none
    print *, 2 ** (1/4)
end program main
```

我猜同学们看到这个程序的第一眼肯定会觉得结果是 $2^{1/4}$, 然而同学们跑一跑就知道结果不是, 这是咋肥事哩? 同学们请看 `1/4`, 它的值看起来应该是 1/4, 然而根据类型与种别转化的规则, `1` 和 `4` 都是整型的, `1/4` 只能是整型的, 所以 `1/4` 的值不可能是 1/4, 好奇怪呢.

这是因为 Fortran 特别规定, 如果 [a] 和 [b] 都是整型数据实体, 那么 [a]/[b] 不是代表 [a] 除以 [b], 而是代表 [a] 整除以 [b]!!! 具体说来, 如果 [a] 和 [b] 都是整型数据实体, 值分别是 a 和 b, 那么 [a]/[b] 的值不是 $a \div b$ 而是 $a \div b$ 的向 0 取整. 所以 1/4 的值不是 $1/4$, 而是 $1/4$ 的向 0 取整, 是 0, 那 2 ** (1/4) 当然是 1 了!

再来看下面这个例子.

```
program main
    implicit none
    integer :: p, q
    real :: r
    p = 1.0
    q = 4.0
    r = p / q
    print *, 2.0 ** r
end program main
```

首先给 p 和 q 分别赋上 1.0 和 4.0. 然而, p 和 q 都是整型的. 根据赋值时的类型转化规则, p 和 q 分别是 1 和 4.

然后计算 p / q 并赋给 r. p 是 1, q 是 4, $1 \div 4 = 1/4$. 然而, p 和 q 都是整型的, 因此不是要除而是要整除, $1/4$ 要向 0 取整, 所以 p / q 是 0. 虽然 r 是实型的, 但是并没有什么卵用. 因为是先计算, 再赋值. 先计算 p / q 得到 0, 然后赋给 r, 再类型转化, 所以 r 是 0.0!!!

最后计算 2.0 ** r, 结果自然是 1.0!

最后来看下面这个例子.

```
program main
    implicit none
    print *, 2 ** (-4)
end program main
```

我猜同学们看到这个程序肯定不敢觉得结果是 2^{-4} 了吧. 2 和 4 都是整型的, 那么结果肯定不是 2^{-4} 喽. Fortran 还特别规定, 如果 [a] 和 [b] 都是整型数据实体, 值分别是 a 和 b, 且 b 是负的, 那么 [a]**[b] 的值不是 a^b 而是 a^b 的向 0 取整!!! 所以 2 ** (-4) 当然是 0 了!

我们把这么个坑死人不偿命的规则总结如下: 如果数据实体 [a] 是整

型的且值为 a, 数据实体 `[b]` 是整型的且值为 b, `[op]` 是二元算数运算符且代表运算 OP, 那么 $a\,\mathrm{OP}\,b$ 不一定是整数, 但运算时类型和种别转化的规则又需遵守, 因此 `[a][op][b]` 的值不规定为 $a\,\mathrm{OP}\,b$ 而规定为 $a\,\mathrm{OP}\,b$ 的向 0 取整. 这么规定也不是为了折磨大家, 凭安安拥有的知识大致可以推断, 这么规定后电脑处理数据和人们造编译器都会比较方便, 因此不仅 Fortran 是这么规定的, C 和 Python 2 也是这么规定的. 但实践表明, 这么规定对写程序的人来说实在是太不友好了, 很容易会有诸如运算符 `/` 的两边一不小心都是整型数据实体, 然后又忘了不是除而是整除, 结果计算出错的情况发生, 所以到 Python 3 那儿规则终于改了, 规定如果一种二元运算能用整型数据实体算出不是整数的值, 那么结果会自动变成实型数据实体而不会向 0 取整, 大家终于不会被折磨了!

Fortran 语法我们没法儿改, 我们只好自己想办法避坑. 我们发现, 坑会出现在用整型数据实体的时候, 那我们尽量不用整型数据实体就完事儿喽, 但有些地方 (比如字符串的长度) 只能用整型数据实体, 那就没办法了, 不过这些地方非常少, 以安安的经验, 我们是不会因这些整型数据实体而掉坑里去的.

规范 19 *尽可能避免使用整型数据实体; 如果一定要使用, 在表达式中先将其转化成实型数据实体.*

如果有一个实型变量 `a`, 要算 `a` 的 `2` 次方, 安安会写 `a**2.0` 而不会写 `a**2`. 如果有一个实型变量 `b`, 一个整型变量 `n`, 要算 `b` 的 `n` 次方, 安安会写 `b**real(n)` 而不会写 `b**n`. 同学们可能会觉得这么写实在是繁琐, 因为即使 `a**2.0` 写成 `a**2`, `b**real(n)` 写成 `b**n`, 由于 `a` 和 `b` 不是整型的, 这式子没可能算错的. 而且用整型数据实体还有些好处, 例如因为电脑算乘法应该比算乘方快, 所以聪明的编译器在看到 `a**2` 的时候, 很可能会先把它变成等价的式子 `a*a`, 然后再编译运行 (这也是 Fortran 官规明确允许的), 但如果写成 `a**2.0`, 编译器就要先判断出实型的 `2.0` 的值是整数, 然后再把它变成 `a*a`, 这对编译器和造编译器的人来说太难了, 老师也会因这个原因而倡导大家尽量用整型数据实体. 但不管同学们有没有被 Fortran 整怕 (希望不会), 反正俺是被整怕了, 所以宁愿写表达式的时候麻烦得要死也要全力避免踩坑…

5.2.2 字符运算

字符运算符只有一个: `//`, 其作用是把左右两边的字符串连起来得到一个新字符串. 示例如下.

```
program main
    implicit none
    print *, 'Hello'//','//' '//'world'//'!'
end program main
```

如果整型数据实体 `[n]` 的值为 n, 那么 `achar([n])` 是 ASCII 码为 n 的字符. 英文缩写 "CR" 的回车符 ASCII 码是 13, 英文缩写 "LF" 的换行符 ASCII 码是 10, 所以 Windows 的换行字符是 `achar(13)//achar(10)`, Linux 的换行字符是 `achar(10)`, 我们再用 `//` 就可以在字符串中加入换行字符了. 例如在 Windows 中给 `'Hello, world!'` 后面加 Windows 换行字符可写成下面这样.

```
program main
    implicit none
    print *, 'Hello, world!'//achar(13)//achar(10)
end program main
```

但这么写让人晕乎, 因为不熟悉 ASCII 的人 (比如俺) 得查表才知道 ASCII 码 13 和 10 的字符是什么. 我们可以 "用字面常量当注释" 来提示 ASCII 码 13 和 10 的字符是什么, 写成下面这样子.

```
program main
    implicit none
    character(*), parameter :: &
        cr = achar(13), lf = achar(10)
    print *, 'Hello, world!'//cr//lf
end program main
```

或者更简单地, 写成下面这样子.

```
program main
    implicit none
    character(*), parameter :: &
```

```
        crlf = achar(13)//achar(10)
    print *, 'Hello, world!'//crlf
end program main
```

5.2.3 逻辑运算

逻辑运算符一共有五个, 左右需跟逻辑型数据实体. 逻辑运算符中间不能有空格, 例如 `.not.` 不能写成 `. not ..` 常用的有三个: `.not.`, `.and.`, `.or.`, 分别代表同学们高中学过的 “非”, “与”, “或”. 不常用的有两个: `.eqv.` 和 `.neqv.`, 分别代表同学们高中没学过的 “同或” 和 “异或”. 逻辑运算符详细解释如下:

- `.not. [b]`: `[b]` 为假则为真, 否则为假.
- `[a] .and. [b]`: `[a]` 和 `[b]` 都为真则为真, 否则为假.
- `[a] .or. [b]`: `[a]` 和 `[b]` 中有一个为真则为真, 否则为假.
- `[a] .eqv. [b]`: `[a]` 和 `[b]` 都为真或都为假则为真, 否则为假.
- `[a] .neqv. [b]`: `[a]` 和 `[b]` 中有且只有一个为真则为真, 否则为假.

示例俺懒得写了, 请同学们自己写例子来练习.

用逻辑运算符时要注意逻辑运算符是分优先级的, 但优先级俺不讲. 请同学们根据规范 18, 加括号来指明逻辑运算符的先后顺序, 例如同学们可以写 `(a .and. b) .or. c` 或 `a .and. (b .or. c)`, 但不可以写 `a .and. b .or. c`.

5.2.4 关系运算

关系运算符有六个: `<`, `<=`, `>`, `>=`, `==`, `/=`, 它们分别有等价的写法 `.lt.`, `.le.`, `.gt.`, `.ge.`, `.eq.`, `.ne.`, 但这些写法安安不喜欢用, 因为显得老气而且要多打几个字符. 关系运算符中间不能有空格, 例如 `==` 不能写成 `= =`, `.ge.` 不能写成 `. ge ..`

如果运算符左右两边是数, 则六个运算符分别代表 $<$, $\leqslant$, $>$, $\geqslant$, $=$, $\neq$. 这种情况下关系运算符经常和其他运算符混用, 请同学们根据规范 18, 加上括号来指明运算符的先后顺序, 例如同学们可以写 `((2 - 1) < 0) .and.`

`((3 - 2) < 0)`, 但不可以写 `2 - 1 < 0 .and. 3 - 2 < 0`. 示例俺懒得写了, 请同学们自己写例子来练习.

严格意义上说, 关系运算符是不能连用的. 比如下面这个程序, Gfortran 是运行不了的. 然而 Ifx 是可以的…

```
program main
    implicit none
    print *, 1 < 2 < 3
end program main
```

要做 “连续的关系运算”, 标准做法是把 “连续的关系运算” 拆分成 “单独的关系运算”, 然后用 `.and.` 连起来, 像下面这样.

```
program main
    implicit none
    print *, (1 < 2) .and. (2 < 3)
end program main
```

这里有一个经典的小坑, 像下面这样的程序可能得出 “错误” 的结果 `.false.`.

```
program main
    implicit none
    print *, (0.1 + 0.2) == 0.3
end program main
```

因为电脑是二进制的, 十进制小数不一定能准确存储, 结果严格上说, `0.1`, `0.2`, `0.3` 的值只是非常接近于 0.1, 0.2, 0.3, 那 `(0.1 + 0.2) == 0.3` 如果是 `.false.` 其实也是不奇怪的. 这个 “问题” 广泛存在于各种编程语言中, 连 Python 3 默认情形下都是如此, 然而上面的程序 Ifx 和 Gfortran 竟都能得出正确的结果, 似乎非常高级…

因为电脑存储数据和计算的时候不免有误差, 所以如果两个表达式的结果是实型或复型的, 那么即使这两个表达式数学上是等价的, 电脑计算后的结果也不一定相等, 因此比较这两个表达式的结果是否相等其实是没有意义的. 不过有些时候, 比较两个表达式的结果是否相等还是有意义的, 这些情况下, 表达式中只有 “整数到整数的运算”, 例如整数的加, 减, 乘, 整除之类的, 这类表达式不会引入存储数据和计算的误差.

规范 20 如果两个表达式中所包含的所有表达式的结果不都是整型的, 那么不比较这两个表达式是否相等.

还有一个小坑, 像下面这样的程序 `==` 左右两边一样, 结果算出来竟是 `.false.`

```
program main
    implicit none
    real :: zero
    zero = 0.0
    print *, (zero / zero) == (zero / zero)
end program main
```

这是因为 `zero / zero` 算出来是 NaN, 而 NaN 被规定成和任何数, 甚至它自己, 都不相等. 所以任意一个数据实体 `[r]`, 我们不能直接把它和 NaN 放在 `==` 两边来判断 `[r]` 是不是 NaN (不论是不是, 结果都是 `.false.`). 但因为只有 NaN 被规定成和自己不相等, 所以当且仅当 `[r]` 是 NaN 时, `[r] /= [r]` 为真, 我们可以利用这点巧妙地判断 `[r]` 是不是 NaN.

如果运算符左右两边是字符串, 则比较运算是比较字符串的先后顺序. 首先, 编译器自己定义了一套字符的先后顺序. 然后比较字符串的先后顺序的时候, 编译器先把两个字符串暂时补成一样长 (短的那个后面补空格).

- 如果两边的字符串前 n 个字符都相同, 且左边字符串第 $(n+1)$ 个字符先于右边字符串第 $(n+1)$ 个字符, 则左边字符串 "`<`" 右边字符串.
- 如果两边的字符串前 n 个字符都相同, 且左边字符串第 $(n+1)$ 个字符后于右边字符串第 $(n+1)$ 个字符, 则左边字符串 "`>`" 右边字符串.
- 如果两边的字符串完全相同, 则左边字符串 "`==`" 右边字符串.

自然, "`>=`" 就是不 "`<`", "`<=`" 就是不 "`>`", "`/=`" 就是不 "`==`".

至于编译器要怎么定义字符的先后顺序, Fortran 又挖了一坑, 在标准里给了一些规定, 使得 ASCII 的顺序是符合规定的顺序之一, 但又没规定死, 也就是说不同编译器进行字符串比较可能得出不同的结果, 所以根据规范 1, 我们不应该比较字符串的先后顺序. 不过 Fortran 填坑还是积极的, 如果 `[a]` 和 `[b]` 是字符串, 那么我们可以分别用 `lge([a], [b])`, `lgt([a], [b])`, `lle([a], [b])`, `llt([a], [b])` 代替 `[a] >= [b]`, `[a] > [b]`, `[a] <= [b]`, `[a] < [b]`, 这样编译器就不会用自己定的顺序, 而会用 ASCII 的

顺序来比较字符串的先后顺序了. 至于 `[a] == [b]` 和 `[a] /= [b]`, 它们是比较字符串是否相同的, 结果和字符的先后顺序没关系, 就不用另搞一套来填坑了. 示例俺懒得写了, 请同学们自己写例子来练习.

第六章　执行控制

6.1　结构

有两种结构 (construct) 是经常使用的: 条件结构 (conditional construct) 和循环结构 (loop construct). 接下来讲的 if 结构是条件结构, do 结构和 do while 结构[1]是循环结构. 在使用结构的时候需要缩进, 同学们需要遵守规范 9.

6.1.1　if 结构

在讲 if 结构之前同学们需要知道 `read *,` 语句, 这种语句将大量出现在示例中. `read *,` 语句干的事和 `print *,` 语句相反, 它会把我们输入在屏幕上的内容赋值给变量. 请看下面这个例子, 这个例子在运行到第五行时会停住.

```
program main
    use,intrinsic :: iso_fortran_env, only: real128
    implicit none
    real(real128) :: number
    read *, number
    print *, number
end program main
```

不论是用 VS 2019 + Ifx 还是用 VS Code + Gfortran 来运行这个例子, 都会弹出个框框. 如果是用 VS 2019 + Ifx, 那么框框里应该有一个闪烁白竖线光标. 如果是用 VS Code + Gfortran, 那么框框里应该有一个白块光标. 假如框框里没有光标, 我们需要在框框的中央用鼠标左键点击一下让光标出现.

[1] 标准解释文档把 do while 结构划成 do 结构中的一种, 但一般还是把 do while 结构单看成一种结构.

光标出现后, 我们需要用键盘输入数字, 假设我们输入 123.456, 我们会发现 123.456 随着我们的输入依次出现在框框里. 然后我们按回车键, 123.456 就被赋值给 number 了. 之后程序会继续运行, number 的值 123.456 因运行第六行而重新出现在屏幕上 (虽然数字的显示格式可能有区别). 特别注意, 我们输入的数字不能带种别, 例如我们不能输入 123.456_real128.

现在开讲 if 结构的基本型. 请看下面这个 "判断是否为 0" 示例. 注意 number 变成整型的了, 所以如果同学们输入的不是整数的话, Gfortran 会报错, 但 Ifx 不会.

```
program main
    implicit none
    integer :: number
    read *, number
    print *, number
    if (number /= 0) then
        ! (number /= 0) == .true.
        print *, 'The number is not 0!'
    else
        ! (number /= 0) == .false.
        print *, 'The number is 0!'
    end if
end program main
```

这个例子中第六行到第十二行是个 if 结构. 程序运行这个 if 结构的规则是: 先计算括号里的表达式, 看它是 .true. 还是 .false..

- 如果是 .true., 那么从 then 下面第一行开始依次运行, 直到运行到 else , 然后跳转从 end if 下面第一行开始依次运行. 这样, else 和 end if 之间的行不会运行.
- 如果是 .false., 那么跳转从 else 下面第一行开始依次运行, 直到运到行 end if , 然后从 end if 下面第一行开始依次运行. 这样, then 和 else 之间的行不会运行.

让我们按 if 结构的规则分析两个具体情况:

- number 是 1: 此时括号里的 number /= 0 是 .true., 那么从 then 下面第一行开始依次运行, 所以第七第八行会运行, 屏幕上出现 "The

number is not 0!", 然后第九行是 `else`, 所以跳转运行 `end if` 下面一行, 也就是 `end program main`. 最终结论: 屏幕上出现 "The number is not 0!".

- `number` 是 `0`: 此时括号里的 `number /= 0` 是 `.false.`, 那么跳转从 `else` 下面第一行开始依次运行, 所以第十第十一行会运行, 屏幕上出现 "The number is 0!", 然后第十二行是 `else`, 所以运行 `end if` 下面一行, 也就是 `end program main`. 最终结论: 屏幕上出现 "The number is 0!".

特别说明, 上面的例子有个小问题, 就是运行后弹出的框框里没有提示语, 如果人们没仔细读过源代码, 他们就不知道程序想让他们干啥了. 所以我们本应加上提示语, 比如像下面这样.

```
program main
    implicit none
    integer :: number
    print *, 'Enter a integer number:'
    read *, number
    print *, number
    if (number /= 0) then
        ! (number /= 0) == .true.
        print *, 'The number is not 0!'
    else
        ! (number /= 0) == .false.
        print *, 'The number is 0!'
    end if
end program main
```

但本笔记里的例子中安安都是偷懒不加提示语的呢.

上面的例子还是太简单了, 如果 if 结构里还有 if 结构, 同学们分析程序估计是要头大的, 我们需要练习. 请看下面这个 "判断是否及格" 示例.

```
program main
    implicit none
    integer :: score
    read *, score
```

```
    print *, score
    if ((score < 0) .or. (score > 100)) then
        ! ((score < 0) .or. (score > 100)) == .true.
        print *, 'Invalid Score!'
    else
        ! ((score < 0) .or. (score > 100)) == .false.
        if (score >= 60) then
            ! (score >= 60) == .true.
            print *, 'Succeed.'
        else
            ! (score >= 60) == .false.
            print *, 'Fail.'
        end if
    end if
end program main
```

让我们按 if 结构的规则分析三个具体情况:

- `number` 是 `101`: 第六行判断为 `.true.`, 程序会运行第六行的 `then` 后的代码, 屏幕上会出现 “Invalid Score!”; 然后第九行是 `else`, 所以会跳转运行第十八行的 `end if` 后的代码. 最终结论: 屏幕上出现 “Invalid Score!”.
- `number` 是 `61`: 第六行判断为 `.false.`, 程序会跳转运行第九行的 `else` 后的代码; 然后第十一行判断为 `.true.`, 程序会运行第十一行的 `then` 后的代码, 屏幕上会出现 “Succeed.”; 然后第十四行是 `else`, 所以会跳转运行第十七行的 `end if` 后的代码; 然后第十八行是 `end if`, 所以会运行第十八行的 `end if` 后的代码. 最终结论: 屏幕上出现 “Succeed.”.
- `number` 是 `1`: 第六行判断为 `.false.`, 程序会跳转运行第九行的 `else` 后的代码; 然后第十一行判断为 `.false.`, 程序会跳转运行第十四行的 `else` 后的代码, 屏幕上会出现 “Fail.”; 然后第十七行是 `end if`, 所以会运行第十七行的 `end if` 后的代码; 然后第十八行是 `end if`, 所以会运行第十八行的 `end if` 后的代码. 最终结论: 屏幕上出现 “Fail.”.

同学们可能还是头大, 安安表示没法子, 这是任何人学第一门编程语言都要闯过的关, 同学们只能刻苦修炼, 争取早日闯关成功. 像安安这样身经百战

(被毒打过不知多少回) 的, 看这个示例是能一秒分析清楚它的运行方式的. 这个程序还体现出缩进的用处, 缩进后可以很清楚地看出, 第六行的 `then`, 第九行的 `else`, 第十八行的 `end if` 是一个 if 结构的, 第十一行的 `then`, 第十四行的 `else`, 第十七行的 `end if` 是一个 if 结构的. 如果没有缩进, 安安分析程序也是要头大的...

现在开讲 if 结构的简化型和复杂型, 没有熟练掌握 if 结构的基本型的同学请不要学习. 请看下面这个示例, 这个示例由 68 页的 "判断是否为 0" 示例删去注释和 `else` 到 `end if` 之间的语句得来.

```
program main
    implicit none
    integer :: number
    read *, number
    print *, number
    if (number /= 0) then
        print *, 'The number is not 0!'
    else
    end if
end program main
```

在这个示例中, `else` 和 `end if` 之间什么也没有. 此时我们可以连 `else` 也删掉, 让 if 结构变成简化型.

```
program main
    implicit none
    integer :: number
    read *, number
    print *, number
    if (number /= 0) then
        print *, 'The number is not 0!'
    end if
end program main
```

再请看下面这个示例, 这个示例由 69 页的 "判断是否及格" 示例删去注释得来.

```
program main
    implicit none
    integer :: score
    read *, score
    print *, score
    if ((score < 0) .or. (score > 100)) then
        print *, 'Invalid Score!'
    else
        if (score >= 60) then
            print *, 'Succeed.'
        else
            print *, 'Fail.'
        end if
    end if
end program main
```

这个示例中, 内层的 if 结构在外层的 if 结构的 `else` 和 `end if` 之间, 这种情况下我们可以用复杂型 if 结构来改写这个示例. 第一步, 我们在内层的 if 结构的前后加注释, 把内层的 if 结构区分出来.

```
program main
    implicit none
    integer :: score
    read *, score
    print *, score
    if ((score < 0) .or. (score > 100)) then
        print *, 'Invalid Score!'
    else
    ! -------------------------
        if (score >= 60) then
            print *, 'Succeed.'
        else
            print *, 'Fail.'
        end if
    ! -------------------------
```

```
    end if
end program main
```

第二步, 我们让内层的 if 结构的缩进少 4 个空格.

```
program main
    implicit none
    integer :: score
    read *, score
    print *, score
    if ((score < 0) .or. (score > 100)) then
        print *, 'Invalid Score!'
    else
    ! -------------------------
    if (score >= 60) then
        print *, 'Succeed.'
    else
        print *, 'Fail.'
    end if
    ! -------------------------
    end if
end program main
```

第三步, 我们删掉上面一行注释, 并把被删掉的注释的下面一行剪切下来, 粘贴到被删掉的注释的上面一行 `else` 后, 删去因剪切粘贴而新出现的空行, 再让 `else` 和 `if` 间至少有一个空格.

```
program main
    implicit none
    integer :: score
    read *, score
    print *, score
    if ((score < 0) .or. (score > 100)) then
        print *, 'Invalid Score!'
    else if (score >= 60) then
        print *, 'Succeed.'
```

```
    else
        print *, 'Fail.'
    end if
    ! -------------------------
    end if
end program main
```

第四步, 我们删掉下面一行注释和注释上面一行的 `end if`.

```
program main
    implicit none
    integer :: score
    read *, score
    print *, score
    if ((score < 0) .or. (score > 100)) then
        print *, 'Invalid Score!'
    else if (score >= 60) then
        print *, 'Succeed.'
    else
        print *, 'Fail.'
    end if
end program main
```

大功告成! 上面的程序的 `if` 和 `end if` 之间就是复杂型 if 结构. 复杂型 if 结构同学们也需要努力练习才能熟练掌握. 请看下面这个 "判断优良中差" 示例, 这个示例 if 结构一层套一层, 看着就可怕. 请同学们练习用复杂型 if 结构来改写这个示例, 改写过程中同学们需要反复多次走上面讲的四步骤, 直到程序中只用一个复杂型 if 结构.

```
program main
    implicit none
    integer :: score
    read *, score
    print *, score
    if (score > 100) then
        print *, 'Invalid Score!'
```

```
    else
        if (score >= 90) then
            print *, 'A'
        else
            if (score >= 80) then
                print *, 'B'
            else
                if (score >= 60) then
                    print *, 'C'
                else
                    if (score >= 0) then
                        print *, 'D'
                    else
                        print *, 'Invalid Score!'
                    end if
                end if
            end if
        end if
    end if
end program main
```

请同学们对答案.

```
program main
    implicit none
    integer :: score
    read *, score
    print *, score
    if (score > 100) then
        print *, 'Invalid Score!'
    else if (score >= 90) then
        print *, 'A'
    else if (score >= 80) then
        print *, 'B'
    else if (score >= 60) then
```

```
        print *, 'C'
    else if (score >= 0) then
        print *, 'D'
    else
        print *, 'Invalid Score!'
    end if
end program main
```

可以看到用复杂型 if 结构改写后, 相比之下看着不那么可怕了. 请同学们熟练掌握复杂型 if 结构并多多使用.

最后讲如何给 if 结构加标签. 给 if 结构加标签很简单, 在 if 结构开头的 `if` 前面加 `[tag]:`, 结尾的 `end if` 后面加 `[tag]`, 即可给 if 结构加标签 `[tag]`. `if` 的左右须有空格. 下面的示例中, 外层的 if 结构加了标签 `outer`, 内层的 if 结构加了标签 `inner`.

```
program main
    implicit none
    integer :: score
    read *, score
    print *, score
    outer: if ((score < 0) .or. (score > 100)) then
        print *, 'Invalid Score!'
    else
        inner: if (score >= 60) then
            print *, 'Succeed.'
        else
            print *, 'Fail.'
        end if inner
    end if outer
end program main
```

但请注意, 因为整个复杂型 if 结构是同一个 if 结构, 所以在用复杂型 if 结构的时候, 只能在整个复杂型 if 结构的头尾加标签. 下面的示例是错误的, 不加标签 `inner` 则是正确的.

```
program main
    implicit none
    integer :: score
    read *, score
    print *, score
    outer: if ((score < 0) .or. (score > 100)) then
        print *, 'Invalid Score!'
    else inner: if (score >= 60) then
            print *, 'Succeed.'
    else
        print *, 'Fail.'
    inner
    end if outer
end program main
```

6.1.2 do 结构

欧拉告诉我们, $\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$. 我们可以将左边的级数截断至一个 $n_{\max}$ 来算出 π 的近似值 $(6 \cdot \sum_{n=1}^{n_{\max}} \frac{1}{n^2})^{1/2}$. 假如我们取 $n_{\max} = 10$, 我们尚且可以像下面这样写长长的式子硬算.

```
program main
    implicit none
    print *, (6.0*(1.0/1.0**2.0 &
                   + 1.0/2.0**2.0 &
                   + 1.0/3.0**2.0 &
                   + 1.0/4.0**2.0 &
                   + 1.0/5.0**2.0 &
                   + 1.0/6.0**2.0 &
                   + 1.0/7.0**2.0 &
                   + 1.0/8.0**2.0 &
                   + 1.0/9.0**2.0 &
                   + 1.0/10.0**2.0))**(1.0/2.0)
end program main
```

但如果我们取 $n_{\max} = 1000$, 写一千行代码就要死人了. 这时我们就有必要使出 do 结构, 像下面这样.

```
program main
    implicit none
    integer :: n
    real :: s
    s = 0.0
    do n = 1, 1000
        s = s + 1.0/real(n)**2.0
    end do
    print *, (6.0*s) ** (1.0/2.0)
end program main
```

这个程序运行到第六行后发生什么事了? 首先电脑把 `n` 赋成 `1`, 然后运行 `do` 到 `end do` 之间的命令, 也就是第七行. 运行完第七行后, 电脑把 `n` 加上 `1`, 然后再一次运行第七行, 然后再把 `n` 加上 `1`, 然后再一次运行第七行… 如此循环进行, 直到 `n` 等于 `1000` 后, 电脑最后一次运行第七行, 然后就运行 `end do` 之后的命令了. 对上面这个例子, 总的过程就是: 令 `n` 等于 `1`, 运行第七行, 令 `n` 等于 `2`, 运行第七行… 令 `n` 等于 `999`, 运行第七行, 令 `n` 等于 `1000`, 运行第七行, 最后运行第九行就得到 π 的近似值了.

上面的程序每次 `n` 都增加 `1`, 可以不如此. 因为 $(6 \cdot \sum_{n=1}^{n_{\max}} \frac{1}{n^2})^{1/2} = (6 \cdot \sum_{n=-1}^{-n_{\max}} \frac{1}{n^2})^{1/2}$, 所以我们可以改用后式, 然后取 $n_{\max} = 1000$ 来算出 π 的近似值, 像下面这样.

```
program main
    implicit none
    integer :: n
    real :: s
    s = 0.0
    do n = -1, -1000, -1
        s = s + 1.0/real(n)**2.0
    end do
    print *, (6.0*s) ** (1.0/2.0)
end program main
```

注意第六行现在多出了个 `, -1` 在最后, 这就表示现在每次 `n` 都增加 `-1`, 也就是减 `1`. 对上面这个例子, 这意味着 `n` 的值会依次是 `-1`, `-2`, ..., `-999`, `-1000`.

我们快速总结一下 do 结构的运行方式, 一个 do 结构前头是 `do [m] = [m1], [m2], [m3]`, 后头是 `end do`. `[m]` 必须是变量, 称作计数变量 (counter variable). 最后的 `, [m3]` 可以省略, 如果省略, 那么 `[m3]` 为 `1`. do 结构的运行方式如下:

1. 计算 `[m1]`, `[m2]`, `[m3]`, 结果分别记作 m_1, m_2, m_3, 分别称作初始参数 (initial parameter), 终端参数 (terminal parameter), 增量参数 (incrementation parameter).

2. 若 m_3 等于 0, 则报错.

3. 让 `[m]` 等于 m_1.

4. 若 m_3 大于 0 且 `[m]` 大于 m_2, 或 m_3 小于 0 且 `[m]` 小于 m_2, 则运行 `end do` 下一行, 否则进行第 5 步.

5. 运行 `do [m] = [m1], [m2], [m3]` 和 `end do` 之间的部分.

6. 运行到 `end do` 时, 让 `[m]` 增加 m_3, 然后回到第 4 步.

我们用下面这个示例来练习分析 do 结构.

```
program main
    implicit none
    integer :: n
    n = 100
    print *, 'Before loop: ', n
    do n = 1, 10, 2
        print *, 'In loop: ', n
    end do
    print *, 'After loop: ', n
end program main
```

在 do 结构前 `n` 当然是 100, 在 do 结构中, m_1, m_2, m_3 分别是 1, 10, 2, 其中 m_3 不是 0, 程序可以继续. 一开始 `n` 是 1, 然后 m_3 是 2, 所以 `n` 依次是

3, 5, 7, 9. 因为 m_3 大于 0, 所以我们要查看 `n` 是否大于 m_2. 之后 `n` 是 11, `n` 大于 m_2 了, 所以运行 `end do` 下一行, 在 do 结构后 `n` 是 11.

运行 `do [m] = [m1], [m2], [m3]` 和 `end do` 之间的部分时, 计数变量 `[m]` 不能被赋值, 否则电脑会昏头, 所以下面这个程序不成.

```
program main
    implicit none
    integer :: n
    n = 100
    do n = 1, 10, 2
        n = 5
    end do
end program main
```

`[m1]`, `[m2]`, `[m3]` 的值则可以变化, 但 m_1, m_2, m_3 在 do 结构运行开始的第 1 步就确定了, 不会改变. 下面这个程序, 因为 do 结构运行开始时 `[m1]`, `[m2]`, `[m3]` 的值是 1, 10, 2, 所以 m_1, m_2, m_3 始终是 1, 10, 2 不变. 但 Fortran 没学好的那些家伙看这样的程序会昏头, 总觉得 m_1, m_2, m_3 会一直跟着 `[m1]`, `[m2]`, `[m3]` 的值变, 因此请同学们不要写这样的程序.

```
program main
    implicit none
    integer :: n
    n = 100
    print *, 'Before loop: ', n
    do n = n-99, n-90, n-98
        print *, 'In loop: ', n
    end do
    print *, 'After loop: ', n
end program main
```

规范 21 *在使用 do 结构时, 让 `[m1]`, `[m2]`, `[m3]` 的值保持不变.*

上面程序的计数变量 `[m]` 都是整型的. 按 Modern Fortran 标准, 计数变量 “shall be” 整型的. 这是无比正确的. 请看下面这个例子.

```
program main
    implicit none
    integer :: n
    real :: r, s
    n = 0
    s = 0.0
    do r = 1.0, 2.0, 0.001
        n = n + 1
        s = s + r**0.5
    end do
    print *, n, r, s
end program main
```

用 Gfortran 跑这个程序的同学们会发现出现一堆警告, 而且 `n` 的输出结果怎么是 `1000` 而不是 `1001`. 仔细看看, `r` 的输出结果可不是 `2.001`. 要知道, 实型运算是有误差的, 这里 `r` 被加了个 1000 次, 疯狂积累误差, 1000 次后刚好使 `r` 大于 `2.0` 一丢丢, 结果电脑就跳出循环了没算第 1001 次, 这结果不是我们期待的!

用 Ifx 跑这个程序的同学们会发现没出警告, 而且结果还正是我们期待的. 可别高兴得太早, 请把增量参数 `0.001` 改成 `0.000001`, 再跑一回, 就会发现 `n` 的输出结果是 `1000000` 而不是 `1000001`, 结果不是我们期待的了, 而且连警告也没有…

规范 22 *在使用 do 结构时, 让 [m], [m1], [m2], [m3] 是整型的.*

我们可以用一些偷鸡摸狗的正确办法来避免用实型的计数变量, 例如我们可以把上面的程序改成下面的程序.

```
program main
    implicit none
    integer :: n, r_
    real :: s
    n = 0
    s = 0.0
    do r_ = 1000, 2000, 1
```

```
        n = n + 1
        s = s + (real(r_)/1000.0)**0.5
    end do
    print *, n, r_, s
end program main
```

这个程序中, 第七行本来要写 `do r = 1.0, 2.0, 0.001`, 第九行本来要写 `s = s + r**0.5`, 但这样 `r` 得是实型的不成. 现在另造一个整型的 `r_`, 并计划让 `r == real(r_)/1000.0`, 于是第七行可以写成 `do r_ = 1000, 2000, 1` (最后的 `, 1` 也可省略), 第九行用 `real(r_)/1000.0` 间接地算 `r`, 所以写成 `s = s + (real(r_)/1000.0)**0.5`, 这样就可以巧妙地避免用实型的计数变量了!

给 do 结构加标签和给 if 结构加标签方式类似, 在 do 结构开头的 `do` 前面加 `[tag]:`, 结尾的 `end do` 后面加 `[tag]`, 即可给 do 结构加标签 `[tag]`. `do` 的左右须有空格. 示例如下.

```
program main
    implicit none
    integer :: n
    real :: s
    s = 0.0
    loop: do n = 1, 1000
        s = s + 1.0/real(n)**2.0
    end do loop
    print *, (6.0*s) ** (1.0/2.0)
end program main
```

最后给大家留两个练习题.

第一个是斐波那契兔子问题. 第 0 月有一对小兔, 每过一个月一对小兔都会长成一对大兔, 每过一个月一对大兔都会生出一对小兔, 问第 12 月大兔生小兔后有多少对兔? 我敢保证结果是 233, 但我觉着相当多的同学一开始肯定会写错程序, 算不出来 233.

第二个是求下面这个矩阵中所有元素的和. 我觉得做这题同学们一定需要让一个 do 结构里再有另一个 do 结构, 正好给同学们练习.

$$\begin{bmatrix} \sqrt{1} & \sqrt{2} & \sqrt{3} & \ddots & \ddots \\ \sqrt{2} & \sqrt{3} & \ddots & \ddots & \ddots \\ \sqrt{3} & \ddots & \ddots & \ddots & \sqrt{99} \\ \ddots & \ddots & \ddots & \sqrt{99} & \sqrt{100} \\ \ddots & \ddots & \sqrt{99} & \sqrt{100} & \sqrt{101} \end{bmatrix}$$

6.1.3 do while 结构

之前我们用一个级数来计算 π, 现在要考虑这个级数收敛速度如何. 假如要计算 n 取多少时级数和能大于 3.14, 我们可以这么做.

```
program main
    implicit none
    integer :: n
    real :: s
    n = 0
    s = 0.0
    do while (s < (3.14**2.0 / 6.0))
        n = n + 1
        s = s + 1.0/real(n)**2.0
    end do
    print *, n
end program main
```

这个程序用了 do while 结构, 其运作过程不难理解. `do while` 后的括号内的表达式必须是逻辑型的, 我们把这表达式记作 `[do-while-expr]`.

1. 若 `[do-while-expr]` 的值为 `.true.`, 则运行 `end do` 下一行, 否则进行第 2 步.
2. 运行 `do while ([do-while-expr])` 和 `end do` 之间的部分.
3. 运行到 `end do` 时, 回到第 1 步.

这里要注意的是 do while 结构的最后一行是 `end do` 而不是 `end do while`, 因为 do while 结构其实是 do 结构中的一种. 给 do while 结构加标签和给 do 结构加标签方式相同.

6.2 执行控制语句

有时我们需要强行打破一个结构, 一个程序单元, 乃至整个程序的执行方式, 这时我们需要一些统称为执行控制语句 (execution control statement) 的东东. 这些执行控制语句只能摆在程序的 “执行部分”, 例如下面这个程序把执行控制语句 `continue` 摆在变量说明上面不成.

```
program main
    implicit none
    continue
    integer :: number
    read *, number
    print *, number
end program main
```

6.2.1 continue 语句

continue 语句写作 `continue`, 表示 “啥也不干”, 和注释行没有太大区别. 这个语句看起来没什么用, 其实还是有点用的. 请看下面这个示例, 这个示例由 68 页的 “判断是否为 0” 示例删去注释和 `then` 到 `else` 之间的语句得来.

```
program main
    implicit none
    integer :: number
    read *, number
    print *, number
    if (number /= 0) then
    else
        print *, 'The number is 0!'
    end if
end program main
```

在这个示例中, `then` 和 `else` 之间什么也没有, Fortran 没学好的那些家伙看这样的程序可能脑子转不过来. 此时我们可以在 `then` 和 `else` 之间插入一个 continue 语句来占个位置, 清楚表明 `then` 到 `else` 之间啥也不干.

```
program main
    implicit none
    integer :: number
    read *, number
    print *, number
    if (number /= 0) then
        continue
    else
        print *, 'The number is 0!'
    end if
end program main
```

特别提醒, Fortran 的 continue 语句相当于 C 和 Python 的 pass 语句, 而 C 和 Python 的 continue 语句相当于 Fortran 的 cycle 语句 (见 6.2.3 小节), 同学们莫要搞混.

6.2.2 exit 语句

已知任意正整数 n , n^2 和 $(n+1)^2$ 之间肯定有个质数. 现打算造个程序来判断 n^2 和 $(n+1)^2$ 之间的每个数是否是质数. 先造出个能跑的程序, 摆在下面.

```
program main
    implicit none
    integer :: i, j, n
    logical :: is_prime
    read *, n
    do i = n**2, (n+1)**2
        is_prime = .true.
        do j = 2, i-1
            if (i == i/j*j) then
                is_prime = .false.
            end if
        end do
        print *, i, is_prime
```

```
    end do
end program main
```

这个程序在效率上让人不大满意, 因为在第九行判断 `j` 是否整除 `i` 时, 其实只需有一回 `i == i/j*j` 就可以知道 `i` 不是质数了, 所以希望在 `i == i/j*j` 时, 将 `is_prime` 赋成 `.false.`, 然后直接运行里层的 `end do` 之后的第十三行输出结果. 这时只需加个 exit 语句就好, 像下面这样.

```
program main
    implicit none
    integer :: i, j, n
    logical :: is_prime
    read *, n
    do i = n**2, (n+1)**2
        is_prime = .true.
        inner_loop: do j = 2, i-1
            if (i == i/j*j) then
                is_prime = .false.
                exit inner_loop       !------+
            end if                            !
        end do inner_loop                     !
        print *, i, is_prime          !<-----+
    end do
end program main
```

在上面这个程序中, 电脑只要一见到 `exit inner_loop` 就浑身一激灵, 然后就直接跳转运行 `end do inner_loop` 之后的语句了. `exit` 后的标签指明了跳转的位置, 如果我们想跳转到外层的 `end do` 后 (上面的程序第十五行后), 我们可以这么写.

```
program main
    implicit none
    integer :: i, j, n
    logical :: is_prime
    read *, n
    outer_loop: do i = n**2, (n+1)**2
```

```
        is_prime = .true.
        do j = 2, i-1
            if (i == i/j*j) then
                is_prime = .false.
                exit outer_loop       !----------+
            end if                                !
        end do                                    !
        print *, i, is_prime                      !
    end do outer_loop                             !
end program main                      !<---------+
```

如果结构只有一层, 不是内外嵌套的, 那么 `exit` 后可以不写标签. 如果结构不只有一层, 是内外嵌套的, 那么 `exit` 后不写标签会引发混乱, 例如上面的程序, 会不知道跳转到第十三行 `end do` 后还是第十五行 `end do` 后.

在 do while 结构和 if 结构中也可使用 exit 语句, 请同学们自己尝试.

6.2.3 cycle 语句

一个循环结构总是代表一个循环, 这个循环会一轮一轮进行, exit 语句是终止整个循环, 而 cycle 语句是终止本轮循环, 进行下一轮循环. 如果我们想终止一轮内层循环, 我们可以这么写.

```
program main
    implicit none
    integer :: i, j, n
    logical :: is_prime
    read *, n
    do i = n**2, (n+1)**2
        is_prime = .true.
        inner_loop: do j = 2, i-1     !<-----+
            if (i == i/j*j) then              !
                is_prime = .false.            !
                cycle inner_loop      !------+
            end if
        end do inner_loop
```

```
        print *, i, is_prime
    end do
end program main
```

在上面这个程序中, 电脑只要一见到 `cycle inner_loop` 就浑身一哆嗦, 然后就不再进行循环第 5 步, 直接跳转到内层循环的开头 (上面的程序第八行), 并进行循环第 4 步. `cycle` 后的标签指明了跳转的位置, 如果我们想终止一轮外层循环, 我们可以这么写.

```
program main
    implicit none
    integer :: i, j, n
    logical :: is_prime
    read *, n
    outer_loop: do i = n**2, (n+1)**2 !<----------+
        is_prime = .true.                         !
        do j = 2, i-1                             !
            if (i == i/j*j) then                  !
                is_prime = .false.                !
                cycle outer_loop      !-----------+
            end if
        end do
        print *, i, is_prime
    end do outer_loop
end program main
```

如果结构只有一层, 不是内外嵌套的, 那么 `cycle` 后可以不写标签. 如果结构不只有一层, 是内外嵌套的, 那么 `cycle` 后不写标签会引发混乱, 例如上面的程序, 会不知道跳转到第八行还是第六行.

在 do while 结构中也可使用 cycle 语句, 请同学们自己尝试. 在 if 结构中没法儿使用 cycle 语句.

6.2.4 stop 语句

stop 语句是 `stop` 后跟字符串, 作用是强行跳转到主程序最后一行. 运行 stop 语句时, 电脑会参考 `stop` 后的字符串来在屏幕上显示编译器认为电

脑应该显示的信息. 示例如下.

```
program main
    implicit none
    call helloworld()
    print *, "main"
end program main              !<-------+
                                       !
subroutine helloworld()                !
    implicit none                      !
    stop "Oh, yes!"           !--------+
    print *, "Hello, world!"
end subroutine helloworld
```

6.2.5 error stop 语句

error stop 语句是 `error stop` 后跟字符串, 作用是强行让电脑报错. 运行 error stop 语句时, 电脑会参考 `error stop` 后的字符串来在屏幕上显示编译器认为电脑应该显示的信息. 示例如下.

```
program main
    implicit none
    call helloworld()
    print *, "main"
end program main

subroutine helloworld()
    implicit none
    error stop "Oh, no!"      ! Raise an error!
    print *, "Hello, world!"
end subroutine helloworld
```

error stop 语句和 stop 语句的不同是 stop 语句造成程序的正常终止 (normal termination) 而 error stop 语句造成程序的错误终止 (error termination).

6.2.6 return 语句

return 语句是 `return`, 后面不能跟字符串, 只能用在子程序中, 作用是强行跳转到子程序最后一行. 示例如下.

```
program main
    implicit none
    call helloworld()
    print *, "main"
end program main

subroutine helloworld()
    implicit none
    return                    !----+
    print *, "Hello, world!"      !
end subroutine helloworld     !<---+
```

没有 “error return 语句”.

第七章　数组

迄今为止我们折腾的东东都是标量 (scalar), 那都是小 case, 大 case 是数组 (array). 数组的名字虽然是 "数" 组, 但其实数组可以是任意类型的, 只不过平常俺们用的数组基本上都是数字型的, 俺就只讲数字型的了. 数组这个东东, 俺觉得俺讲起来和初学编程的同学们理解起来都会是十分困难滴, 对俺来说, 彻底讲清楚数组最好的办法就是来一大堆数学定义, 但偏偏绝大部分同学都一看数学定义就头大, 俺只能不管了. 另外, 之前俺们讲代码的时候, 实际上经常使用左右中括号 `[]`, 表示左右中括号和里头的内容需要按需求被替换, 但笔记写到这里俺突然发现左右中括号从本章开始有大用, 所以从本章开始俺们会改使用左右大括号 `{}`, 表示左右大括号和里头的内容需要按需求被替换.

7.1　数组基础

Fortran 中的数组分为全数组 (whole array) 和非全数组, 先讲全数组. 全数组是一个映射 $a\colon S_1\times\cdots\times S_n\to\mathbb{C}$, $(s_1,\ldots,s_n)\mapsto a_{s_1\ldots s_n}$, 其中任意 $i\in\{1,\ldots,n\}$, $S_i=\{j_i,\ldots,k_i\}\subset\mathbb{Z}$. 换言之, 全数组是一堆带着 n 个整数下标的复数 $a_{s_1\ldots s_n}$, $s_1,\ldots,s_n$ 的取值范围分别是 $\{j_1,\ldots,k_1\},\ldots,\{j_n,\ldots,k_n\}$. 数组的应用非常多, 数学中的向量, 矩阵[1], 物理中的矢量, 张量[2]都可以表示成数组, 所以这一章特别是这一节同学们铁定得啃下来呢.

按上一段中数组的映射定义, 定义一个数组 a, 则定义中的正整数 n 称为数组 a 的维数/秩 (rank), 标量相当于 0 维数组. 任意 $i\in\{1,\ldots,n\}$, 称数组 a 有第 i 个维度 (dimension), 整数 j_i 和整数 k_i 分别称为数组 a 的第 i 个维度的下界 (lower-bound) 和上界 (upper-bound), $\{j_i,\ldots,k_i\}$ 的元素个数

[1]同学们会学到的.

[2]同学们会学到的.

$l_i = (k_i - j_i) + 1$ 称为数组 a 的第 i 个维度的长度 (extent). 向量 $(l_1, \ldots, l_n)$ 称为数组 a 的形状 (shape), 其本身被认定成 1 维数组, 正整数 $l_1 \times \cdots \times l_n$ 称为数组 a 的大小 (size). 数组 a 的值域中的任意元素 $a_{s_1 \ldots s_n}$ 称为数组 a 的一个元素 (element), $s_1, \ldots, s_n$ 中的第 i 个整数 s_i 称为 $a_{s_1 \ldots s_n}$ 的第 i 个下标/索引 (subscript/index). 以上这一大堆定义是同学们需要牢记的.

若程序中变量 `a` 代表全数组 a, 则 `rank(a)`, `lbound(a)`, `ubound(a)`, `shape(a)`, `size(a)` 分别是全数组 a 的维数 n, 由所有维度的下界组成的向量 $(j_1, \ldots, j_n)$, 由所有维度的上界组成的向量 $(k_1, \ldots, k_n)$, 形状 $(l_1, \ldots, l_n)$, 大小 $l_1 \times \cdots \times l_n$.

同学们玩数组的时候的难点通常是不能领会数据结构 (data structure), 而数据结构正是程序设计的灵魂. 最简单帮助同学们领会数据结构的方式就是让数据结构直观化. 例如一个 1 维数组 $a\colon \{1,2,3\} \to \mathbb{C}$, 我们可以把它想象成一横条 $[a_1\, a_2\, a_3]$, 但数学中通常喜欢把这横条竖起来摆. 又例如一个 2 维数组 $a\colon \{1,2,3\} \times \{1,2,3\} \to \mathbb{C}$, 我们可以把它想象成一大方表

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

3 维数组就麻烦了, 我们需要想象有一大堆正方体小箱子堆成一个大长方体, 每个小箱子中都装着一个复数. 更高维的数组就更麻烦了, 例如 4 维数组, 只能要么想象成有一大堆正方体小箱子堆成一个 “ 4 维大长方体”, 每个小箱子中都装着一个复数, 要么想象成一横条, 横条由大长方体排列成, 每个大长方体都由一大堆正方体小箱子堆成, 每个小箱子中都装着一个复数.

Fortran 数组中的元素有一个被规定死的排列顺序: 任意 $a_{s_1 \ldots s_n}$ 和 $a_{t_1 \ldots t_n}$, 当 $s_{i+1} = t_{i+1}, \ldots, s_n = t_n$ 时, 若 $s_i < t_i$, 则 $a_{s_1 \ldots s_n}$ 在 $a_{t_1 \ldots t_n}$ 前面, 若 $s_i > t_i$, 则 $a_{s_1 \ldots s_n}$ 在 $a_{t_1 \ldots t_n}$ 后. 如果用 `print *,` 输出数组, 那么 Fortran 会按元素顺序挨个输出数组中的每个元素. Fortran 规定的排列顺序称为列优先顺序 (column-major order). Matlab 学 Fortran, 也是列优先顺序. C 规定的排列顺序不一样, 是行优先顺序 (row-major order), Python 学 C, 也是行优先顺序. 例如一个 2 维数组 $a\colon \{1,2\} \times \{1,2\} \to \mathbb{C}$, 在 Fortran 规定的列优先顺序中, 位置最前的下标变动最快, 排序是 $a_{11}, a_{21}, a_{12}, a_{22}$, 示意图如 7.1 所示, 在 C 规定的行优先顺序中, 位置最后的下标变动最快, 排

序是 $a_{11}, a_{12}, a_{21}, a_{22}$, 示意图如 7.2 所示.

$$\begin{array}{ccccc} a_{11} & & \to & \to & a_{12} \\ \downarrow & & \uparrow & & \downarrow \\ \downarrow & & \uparrow & & \downarrow . \\ \downarrow & & \uparrow & & \downarrow \\ a_{21} & \to & \uparrow & & a_{22} \end{array} \tag{7.1}$$

Fortran 规定的列优先顺序示意图

$$\begin{array}{ccccc} a_{11} & \to & \to & \to & a_{12} \\ & & & & \downarrow \\ \downarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow . \\ \downarrow & & & & \\ a_{21} & \to & \to & \to & a_{22} \end{array} \tag{7.2}$$

C 规定的行优先顺序示意图

如果两个数组 a 和 b, 它们的形状一样, 那么这两个数组的元素有自然的一一对应关系, 即按 Fortran 规定的元素排列顺序, a 的元素中排在第 i 个的元素和 b 的元素中排在第 i 个的元素是对应的. 换言之, 如果用相同的直观化方法来展示 a 和 b 的元素, 那么 a 和 b 的元素中摆在相同位置的元素是对应的. 例如一个 2 维数组 $a\colon \{0,1,2\} \times \{0,1\} \to \mathbb{C}$ 和另一个 2 维数组 $b\colon \{1,2,3\} \times \{1,2\} \to \mathbb{C}$, 它们的元素对应关系示意图如 7.3 所示 (数组元素的左边 $n\colon$ 一样的元素是对应的).

$$\begin{array}{cccc} 1\colon a_{00} & 4\colon a_{01} & 1\colon b_{11} & 4\colon b_{12} \\ 2\colon a_{10} & 5\colon a_{11} & 2\colon b_{21} & 5\colon b_{22}. \\ 3\colon a_{20} & 6\colon a_{21} & 3\colon b_{31} & 6\colon b_{32} \end{array} \tag{7.3}$$

元素对应关系示意图

如果两个数组可以有元素对应关系, 那么它们需要有相同的形状, 但它们的每个维度都不需要有相同的上下界. 非全数组是和全数组一样具有确定的形状, 但每个维度都没有确定的上下界的数组, 也就是说, 非全数组和全数组一样可以用横条, 方表等进行直观化表示, 可以和全数组有元素对应关系, 但非全数组中的元素都无法写出其下标. 若程序中变量 `a` 代表非全数组 a, 则 `rank(a)`, `shape(a)`, `size(a)` 仍然分别是非全数组 a 的维数 n, 形状

$(l_1, \ldots, l_n)$, 大小 $l_1 \times \cdots \times l_n$, 而 `lbound(a)`, `ubound(a)` 则分别是由 n 个 1 组成的向量 $(1, \ldots, 1)$, 由所有维度的长度组成的向量 $(l_1, \ldots, l_n)$, 仿佛每个维度的下界都确定是 1 一样 (但其实不是). 为表述方便, 以下会用形状和非全数组相同的, 每个维度的下界都确定是 1 的全数组来表示非全数组.

7.2 数组构造

我们可以用数组构造器 (array constructor) 来构造一维数组常量. 用数组构造器构造出来的一维数组常量都是非全数组. 数组构造器的用法复杂至极, 我们只学两种用法. 所有数组构造器都形如 `[...]`, 且两边的 `[` 和 `]` 都可以用 `(/` 和 `/)` 替代而写成 `(/.../)`.

第一种用法是直接写 `[a1, ..., an]`, `a1, ..., an` 中的任意 `ai` 是值为 a_i 的标量, 来代表 $[a_1, ..., a_n]$. 例如 `[1, 2, 3]` 代表 $[1, 2, 3]$, 非常简单.

第二种用法是用隐式 do 循环 (implied DO loop), 这玩意儿比较难. 同学们需要掌握常用的一重隐式 do 循环, 如果老师敢考难死人的多重隐式 do 循环, 俺们就赶紧揭竿而起… 假如我们要造个数组 $[1, \ldots, 1000]$, 写 `[1, ..., 1000]` 肯定不成, 这时我们可以写 `[(i, i = 1, 1000)]`, 其中 `i` 必须是声明过的整型变量. 一般地, 隐式 do 循环 `[({expr}, {m} = {m1}, {m2}, {m3})]` 的第 i 个元素是下面这样的显式 do 循环输出的第 i 个数, 其中 `, {m3}` 若省略, `{m3}` 则为 `1`.

```
do {m} = {m1}, {m2}, {m3}
    print *, {expr}
end do
```

例如 `[(n**2, n = 1, 10, 2)]` 代表 $[1^2, 3^2, 5^2, 7^2, 9^2]$. 再次提醒, `{m}` 必须是声明过的整型变量.

Ifx 有器规, 可以用简写 `[{m1}:{m2}:{m3}]` 代替 `[({m}, {m} = {m1}, {m2}, {m3})]`. 这种不规范的写法同学们不许用.

一般地, 多重隐式 do 循环 `[(...({expr}, m_1 = {m1_1}, {m2_1}, {m3_1}), ..., m_n = {m1_n}, {m2_n}, {m3_n})]` 的第 i 个元素是下面这样的多重显式 do 循环输出的第 i 个数, 其中 `, {m3_i}` 若省略, `{m3_i}` 则为 `1`.

```
do {m_n} = {m1_n}, {m2_n}, {m3_n}
```

```
    ......
        do {m_1} = {m1_1}, {m2_1}, {m3_1}
            print *, expr
        end do
    ......
end do
```

例如 `[((m+n**2, m = 0, 1), n = 1, 2)]` 代表 $[0+1^2, 1+1^2, 0+2^2, 1+2^2]$. Python 有一个和多重隐式 do 循环很相似的, 叫列表推导式的东东, 能给出多维数组. 但请注意, Fortran 的多重隐式 do 循环只能给出一维数组常量, 如果需要多维数组常量, 那么需要用 7.4 节讲的数组操作把一维数组常量变成多维数组常量.

7.3 数组声明

用数组变量和数组具名常量之前当然要声明. 在数组声明中需要声明数组的类型, 还可以另外声明数组的种别, 数组的类型/种别是数组中的每个元素的类型/种别.

在数组声明中还需要给数组加上 dimension 属性, 来确定数组的维数和每个维度的上下界[3]. 有两种方法可行, 一种是直接在声明的 `::` 后面的数组变量/数组具名常量的名称的后面加 `(j1:k1, ..., jn:kn)`, 一种是在声明的 `::` 前面加 `, dimension(j1:k1, ..., jn:kn)`. `j1`, ..., `jn` 中的所有 `ji` 和 `k1`, ..., `kn` 中的所有 `ki` 都必须是整型常量 (或整型常量表达式). `j1:k1`, ..., `jn:kn` 中的所有 `ji:` 都可省略, 若省略, 则 `ji` 为 `1`, 换言之, Fortran 数组的任意维度的下界默认是 `1`. Matlab 学 Fortran, 也默认是 `1`. C 不一样, 默认是 `0`, Python 学 C, 也默认是 `0`. 下面是数组声明示例.

```
program main
    implicit none
    real :: array1(10, 10)
    real, dimension(10, 10) :: array2
    real :: array3(1:10, 1:10)
    real, dimension(1:10, 1:10) :: array4
end program main
```

[3]这表明数组变量和数组具名常量是全数组.

选择哪种声明方式呢? 一般大家喜欢用不写 dimension 的第一种方式, 毕竟少打几个字, 而且看着比较简洁. 但如果是要一口气声明一堆上下界都一样的数组, 这时第一种方式反而不好使了, 大家就喜欢用写 dimension 的第二种方式. 下面也是数组声明示例.

```
program main
    implicit none
    real :: a(1, 1), b(2, 2), c(3, 3)
    real, dimension(4, 4) :: d, e, f
end program main
```

如果是声明数组具名常量, 那么我们可以在声明时把任意维度的上界写成 *, 这样数组具名常量的这个维度的上界直接由初始化时使用的数组字面常量确定, 非常方便. 不过这种数组的名字和字符串不一样, 不叫假定形状数组 (假定形状数组在 8.2.3 小节讲), 而叫隐式形状数组 (implied-shape array). 下面是隐式形状数组示例, 其中数组具名常量 v 唯一的维度的下界为 1 (声明中 * 前的 1: 可省略), 上界为 3.

```
program main
    implicit none
    real, parameter :: v(1:*) = [0.1, 1.1, 2.1]
end program main
```

和延迟长度字符串类似, 我们可以在声明中除去所有 ji:ki 中的 ji 和 ki, 剩下 :, 并附带地用 allocatable 加上 allocatable 属性, 来表示声明的是延迟形状数组 (deferred-shape array). 延迟形状数组各维度的上下界在声明后未定, 但维数确定. 任意延迟形状数组 a, 我们可以用形如 allocate(a(j1:k1, ..., jn:kn)) 的 allocate 语句让 a 各维度的下界确定成 j1, ..., jn (任意 j1: 仍可省略), 上界确定成 k1, ..., kn, 用形如 deallocate(a) 的 deallocate 语句让 a 各维度的上下界未定. 这也意味着延迟形状数组只能是变量. 示例如下.

```
program main
    implicit none
    real, allocatable :: array1(:)
    real, dimension(:, :), allocatable :: array2
```

```
    allocate(array1(10))
    allocate(array2(10, 10))
    deallocate(array1)
    deallocate(array2)
end program main
```

请注意使用 allocate 语句的时候数组的维数要匹配. 例如下面这个例子, a 是二维的, 但 allocate 语句却想让 a 是一维的, 这将失败.

```
program main
    implicit none
    real, allocatable :: a(:, :)
    allocate(a(10)) ! Error: ranks are different!
end program main
```

和延迟长度字符串类似, 7.5 节将介绍, 我们给延迟形状数组赋值的时候, 电脑会自动使用 allocate 语句和 deallocate 语句, 我们不用操心了. 但是, 如果程序运行到某时已经不需要使用某个延迟形状数组, 请对那个延迟形状数组使用 deallocate 语句. 需要这么做的原因是, 通常情况下如果程序运行到某时已经不需要使用某个变量, 那个变量很可能会被电脑自动删掉, 这个操作的学名是垃圾回收 (garbage collection), 但因为延迟形状数组需要我们或电脑手动用 allocate 语句和 deallocate 语句操作一番, 所以延迟形状数组很可能不会被电脑自动删掉, 偏偏实践中的延迟形状数组又通常巨大无比, 1 个 G 都完全不算什么, 最后的结果是大大的延迟形状数组一直把内存占着, 电脑不堪重负, 程序跑得慢甚至崩溃出错. 所以我们得老老实实不厌其烦地用 deallocate 语句, 它能让延迟形状数组把内存让出来. 延迟长度字符串本也有这个问题, 但实践中的延迟长度字符串通常很小 (天文人经常做大规模数据处理, 但不经常做大规模文本处理), 这个问题姑且可以无视.

规范 23 *如果程序运行到某时已经不需要使用某个有 allocatable 属性的变量, 那么对这个变量使用 deallocate 语句.*

7.4 数组操作

我们可以对数组进行变形 (reshape), 这很简单. 变形就是用一个旧数组来造一个形状未必一样的新数组, 使得新数组中排在第 i 个的元素和旧数组

中排在第 i 个的元素相同, 换言之, 把旧数组中的元素按元素顺序复制出来, 再按元素顺序重新排列成某种形状来生成新数组. 如果 `{a}` 是旧数组, 那么 `reshape({a}, {shape})` (其中 `{shape}` 是一维数组) 是对旧数组 `{a}` 进行变形得到的形状为 `{shape}` 的新数组. 例如, 假设我们想造式 (7.4) 中的数组,

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \tag{7.4}$$

我们可以把 `[(i, i = 1, 9)]` 变形成形状为 `[3, 3]` 的数组, 因此我们写 `reshape([(i, i = 1, 9)], [3, 3])` 即可.

我们可以对数组进行转置 (transpose), 这有点儿难. 用数学来说, 设旧数组可表示为 $a_{s_1 \dots s_n}$, 新数组可表示为 $b_{s_1 \dots s_n}$, 另给表示 $a_{s_1 \dots s_n}$ 和 $b_{s_1 \dots s_n}$ 的维度的关系的一维整型向量, 用 `{order}` 表示, 此向量中的元素按顺序依次为 i_1, …, i_n, 则 $b_{s_1 \dots s_n} = a_{s_{i_1} \dots s_{i_n}}$. 直观化地理解, 请同学们想象有一个 n 维空间, 坐标轴为 s_1 轴, … s_n 轴, 数组 a 的元素 $a_{s_1 \dots s_n}$ 挂在坐标点 $(s_1, \dots, s_n)$ 上, 然后我们变换空间的坐标轴, 让旧 s_1 轴变成新 s_{i_1} , … 旧 s_n 轴变成新 s_{i_n}, 变换后挂在坐标点 $(s_1, \dots, s_n)$ 上的元素就是 b 的元素 $b_{s_1 \dots s_n} = a_{s_{i_1} \dots s_{i_n}}$. 我们需要更具体的例子. 假设我们想把式 (7.4) 中的数组转置成式 (7.5) 中的数组.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{7.5}$$

我们先给式 (7.4) 中的数组补上坐标轴如 (7.6) 所示,

$$\begin{array}{cccccc} \cdot & - & - & - & \to & s_2 \\ | & 1 & 4 & 7 & & \\ | & 2 & 5 & 8 & & \\ | & 3 & 6 & 9 & & \\ \downarrow & & & & & \\ s_1 & & & & & \end{array} \tag{7.6}$$

再给式 (7.5) 中的数组补上坐标轴如 (7.7) 所示,

$$\begin{array}{cccccc} \cdot & - & - & - & \to & s_2 \\ | & 1 & 2 & 3 & & \\ | & 4 & 5 & 6 & & \\ | & 7 & 8 & 9 & & \\ \downarrow & & & & & \\ s_1 & & & & & \end{array} \tag{7.7}$$

对比 (7.6) 和 (7.7), 我们可发现旧 s_1 轴是新 s_2 轴, 旧 s_2 轴是新 s_1 轴, 所以表示维度的关系的向量 `{order}` 应是 `[2, 1]`. 如果 `{a}` 是旧数组, 那么 `reshape({a}, shape({a}), order={order})` 是转置后的新数组. 因此我们写 `reshape(reshape([(i, i = 1, 9)], [3, 3]), shape(reshape([(i, i = 1, 9)], [3, 3])), order=[2, 1])` 即可.

请同学们练熟数组的变形和转置, 然后我们学简化的语法. 在上一段的示例中, 我们干了两步, 先变形后转置, 代码写得老长. 我们可以直接把 `reshape(reshape({a}, {shape}), shape({a}), order={order})` 简写成 `reshape({a}, {shape}, order={order})`, 所以我们可以写 `reshape([(i, i = 1, 9)], [3, 3], order=[2, 1])`.

我们可以对数组进行取元, 得到数组中的元素, 这超简单. 设 `a` 为数组 a, 而 `s1, ..., sn` 为整数 $s_1, \ldots, s_n$, 则 `a(s1, ..., sn)` 为 $a_{s_1 \ldots s_n}$, 例如 `a(1, 3)` 是 a_{13}.

我们可以获取数组片段 (section), 这难上天了. 我先来讲用向量下标 (vector subscript) 来获取数组片段, 但即使是 Python 那里, 向量下标也是超级难点, 如果老师敢考难死人的向量下标, 俺们就赶紧揭竿而起…. 首先我们需要数组 `a`, 表示 $a_{s_1 \ldots s_n}$. 然后我们需要 n 个一维整型向量 `i1, ..., in`, 分别表示 $[i_{11}, \ldots, i_{1l_1}], \ldots, [i_{n1}, \ldots, i_{nl_n}]$, 那么 `a(i1, ..., in)` 的元素可一一对应于形状为 $[l_1, \ldots, l_n]$ 的数组 $b_{s_1 \ldots s_n}$ 的元素, 并且 $b_{s_1 \ldots s_n} = a_{i_{1s_1} \ldots i_{ns_n}}$. 请看下面这个例子.

```
program main
    implicit none
    integer :: i, a(3, 3)
    a = reshape([(i, i = 1, 9)], [3, 3])
    print *, a([1, 3, 2], [2, 1, 1, 3])
end program main
```

这个例子中 a 的元素可以排成表, 如 (7.8) 所示.

$$\begin{array}{ccc} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{array} \tag{7.8}$$

先可确定例子中的数组片段形状为 $[3,4]$, 我们先在一个 3×4 的表中填满 $a_{??}$, 并在周围补上坐标轴, 如 (7.9) 所示.

$$\begin{array}{cccccccc} \cdot & - & - & - & - & \to & s_2 \\ | & a_{??} & a_{??} & a_{??} & a_{??} \\ | & a_{??} & a_{??} & a_{??} & a_{??} \\ | & a_{??} & a_{??} & a_{??} & a_{??} \\ \downarrow \\ s_1 \end{array} \tag{7.9}$$

向量 i1 是 $[1,3,2]$, 向量 i2 是 $[2,1,1,3]$, 所以我们我们需要这么做: 沿着坐标轴 s_1 方向, 每一列从上到下, 让表中 $a_{??}$ 的第 1 个坐标依次为 $1,3,2$; 沿着坐标轴 s_2 方向, 每一行从左到右, 让表中 $a_{??}$ 的第 2 个坐标依次为 $2,1,1,3$. 这样得到的表如 (7.10) 所示.

$$\begin{array}{cccccccc} \cdot & - & - & - & - & \to & s_2 \\ | & a_{12} & a_{12} & a_{13} & a_{13} \\ | & a_{32} & a_{31} & a_{31} & a_{33} \\ | & a_{22} & a_{21} & a_{21} & a_{23} \\ \downarrow \\ s_1 \end{array} \tag{7.10}$$

这样我们就成功获得数组片段啦. 如果 a 是 n 维数组, 亦可如法炮制, 请同学们自己尝试.

如果向量 i1, ..., in 中的任意 im 是 [im_1], 其中 [im_1] 是标量, 那么我们可以用 im_1 代替 im 或 [im_1]. 但这么做之后得到的数组片段和原来的不太一样, 数组片段的第 m 个维度会被灭掉, 而后面的维度会依次递补上来. 请看下面这个例子.

```
program main
    implicit none
    integer :: i, a(3, 3)
```

```
    a = reshape([(i, i = 1, 9)], [3, 3])
    print *, a([3], [2, 1, 1, 3])
end program main
```

用之前的方法可知这个例子中数组片段维数为 2, 形状为 $[1,4]$, 元素可以排成表, 如 (7.11) 所示.

$$\begin{array}{cccccccc} \cdot & - & - & - & - & \to & s_2 \\ \downarrow & a_{32} & a_{31} & a_{31} & a_{33} & & \\ s_1 & & & & & & \end{array} \tag{7.11}$$

又请看下面这个例子.

```
program main
    implicit none
    integer :: i, a(3, 3)
    a = reshape([(i, i = 1, 9)], [3, 3])
    print *, a(3, [2, 1, 1, 3])
end program main
```

原本向量 `i1` 是 $[3]$, 这个例子中用 3 代替, 结果是 (7.11) 所示的表的 s_1 维被灭掉, 而 s_2 维依次递补上来变成 s_1 维, 所以这个例子中数组片段的维数为 1, 形状为 $[4]$, 元素可以排成表, 如 (7.12) 所示.

$$\begin{array}{ccccccc} \cdot & - & - & - & - & \to & s_1 \\ & a_{32} & a_{31} & a_{31} & a_{33} & & \end{array} \tag{7.12}$$

我接着讲用三元下标 (vector subscript[4]) 来获取数组片段. 三元下标是 `m1:m2:m3`, 等价于向量下标 `[(i, i = m1, m2, m3)]`, 请注意这是官规而不是 Ifx 的器规, 并且三元下标两边没有左右中括号 `[]`. 若省略 `m1`/`m2`, 则 `m1`/`m2` 是用于获取片段的数组的, 三元下标对应的维度的上界/下界. 若省略 `:m3`, 则 `m3` 是 `1`. 请看下面这个例子.

```
program main
    implicit none
    integer :: i, a(3, 3)
```

[4] 标准解释文档里用的词是 “subscript triplet”, 可翻译成 “下标三元组”.

```
    a = reshape([(i, i = 1, 9)], [3, 3])
    print *, a(::2, 2)
end program main
```

这个例子中, 三元下标 `::2` 对应 `a` 的第 1 个维度, `a` 的第 1 个维度的上界和下界分别是 1 和 3, 所以 `::2` 等价于 `1:3:2`, 亦即等价于 `[(i, i = 1, 3, 2)]`, 亦即等价于 `[1, 3]`. 并且数组片段的第 2 个维度被灭掉. 所以这个例子中数组片段的维数为 1, 形状为 [2], 元素可以排成表, 如 (7.13) 所示.

$$\begin{array}{l} \cdot \\ \vert \quad a_{12} \\ \vert \quad a_{32} \\ \downarrow \\ s_1 \end{array} \tag{7.13}$$

学 Fortran 的 Matlab 的三元下标用法和 Fortran 相同, 而 C 和学 C 的 Python 的三元下标用法和 Fortran 不相同, 请同学们注意.

同学们如果学数组片段学得晕乎, 请至少掌握使用三元下标 `m1:m2` 且 `m1 <= m2` 的情形, 此情形很常用. 若 `a` 是数组, 且任意 `m1_i:m2_i`, `m1_i <= m2_i`, 则 `a(m1_1:m2_1, ..., m1_n:m2_n)` 代表把 `a` 的元素中第 `i` 个下标不在左右闭区间 [`m1_i`, `m2_i`] 中的元素灭掉得到的数组片段. 请看下面这个例子.

```
program main
    implicit none
    integer :: i, a(3, 3)
    a = reshape([(i, i = 1, 9)], [3, 3])
    print *, a(1:2, 2:3)
end program main
```

先将 `a` 的元素排成表, 如 (7.14) 所示.

$$\begin{array}{lll} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{array} \tag{7.14}$$

第 1 个三元下标是 `1:2`, 而 3 不在 $[1,2]$ 中, 所以灭掉 `a` 的元素中第 1 个下标是 3 的元素. 第 2 个三元下标是 `2:3`, 而 1 不在 $[2,3]$ 中, 所以灭掉 `a` 的

元素中第 2 个下标是 1 的元素. 结果如 (7.15) 所示.

$$\begin{array}{ccc} \times & a_{12} & a_{13} \\ \times & a_{22} & a_{23} \\ \times & \times & \times \end{array} \tag{7.15}$$

所以得到的数组片段如 (7.16) 所示.

$$\begin{array}{cc} a_{12} & a_{13} \\ a_{22} & a_{23} \end{array} \tag{7.16}$$

本节讲的数组操作, 得到的都是非全数组. 非全数组没有确定的下标, 所以对非全数组取元素/片段将会失败. 例如下面这个程序对数组片段 `a(:, :)` 再取片段不成.

```
program main
    implicit none
    integer :: i, a(3, 3)
    a = reshape([(i, i = 1, 9)], [3, 3])
    print *, a(:, :)(:, :) ! Invalid!
end program main
```

7.5 数组赋值

给数组 `{var}` 用 `{var} = {expr}` 赋值, 情况可分两种, 一种 `{expr}` 是数组, 另一种 `{expr}` 是标量.

如果 `{expr}` 是数组, 那么 `{var}` 和 `{expr}` 形状必须一样. 此时让 `{var}` 和 `{expr}` 一一对应的元素相等. 示例如下, 其中 `{expr}` 是非全数组, 上下界不定, 但元素的一一对应关系和上下界无关, 所以赋值可行.

```
program main
    implicit none
    integer :: i
    real :: a(3, 3)
    a = reshape([(real(i), i = 1, 9)], [3, 3])
    a = -a
    do i = 1, 3
```

```
        print *, a(i, :)
    end do
end program main
```

如果 {var} 是延迟形状数组, 那么 {var} 和 {expr} 维数必须一样, 此时 lbound({var}) 和 ubound({var}) 分别是 lbound({expr}) 和 ubound({expr}). 示例如下, 其中 {expr} 是非全数组, 则 lbound({var}) 和 ubound({var}) 分别是 $[1, 1]$ 和 $[3, 3]$.

```
program main
    implicit none
    integer :: i
    real, allocatable :: a(:, :)
    a = reshape([(real(i), i = 1, 9)], [3, 3])
    do i = 1, 3
        print *, a(i, :)
    end do
    print *, lbound(a)
    print *, ubound(a)
    deallocate(a)
end program main
```

如果 {expr} 是标量, 那么先临时造一个形状与 {var} 一样, 而其中元素皆为 {expr} 的数组 {expr_}, 然后执行 {var} = {expr_}. 示例如下.

```
program main
    implicit none
    integer :: i
    real :: a(3, 3)
    a = 0.0
    do i = 1, 3
        print *, a(i, :)
    end do
end program main
```

如果 {var} 是延迟形状数组且形状未定, 那么执行 {var} = {expr} 将失败. 错误示例如下, 此程序 Ifx 能跑起, 原因应该是 Ifx 在 {var} 形状未定的

情况下, 会自己认定 {var} 的各维度的长度是 0.

```
program main
    implicit none
    integer :: i
    real, allocatable :: a(:, :)
    a = 0.0 ! Invalid!
    do i = 1, 3
        print *, a(i, :)
    end do
    print *, lbound(a)
    print *, ubound(a)
    deallocate(a)
end program main
```

如果 {var} 是延迟形状数组且形状已定, 那么 {var} 的各维度的上界和下界虽被重定, 但重定后的各维度的上界和下界和重定前的各维度的上界和下界相同, 相当于上界和下界没被重定, 并且随后执行 {var} = {expr}. 示例如下.

```
program main
    implicit none
    integer :: i
    real, allocatable :: a(:, :)
    allocate(a(0:2, 0:2))
    a = 0.0
    do i = 1, 3
        print *, a(i, :)
    end do
    print *, lbound(a)
    print *, ubound(a)
    deallocate(a)
end program main
```

仿照对整个数组赋值, 我们可以轻松地对数组元素/片段赋值, 示例如下, 示例中数组元素 a(3, 1) 被赋值成 0, 数组片段 a(1:2, 2:3) 对应的元

素 a(2, 1), a(2, 2), a(3, 1), a(3, 2) 分别被赋值成 1, 2, 3, 4, 其他元素不变.

```
program main
    implicit none
    integer :: i, a(3, 3)
    a = reshape([(i, i = 1, 9)], [3, 3])
    a(3, 1) = 0
    a(1:2, 2:3) = reshape([(i, i = 1, 4)], [2, 2])
    do i = 1, 3
        print *, a(i, :)
    end do
end program main
```

Fortran 数组赋值是并行 (parallel) 计算, 请看下面的例子.

```
program main
    implicit none
    integer :: i, a(9)
    a = [(i, i = 1, 9)]
    a(2:9) = a(1:8)
    print *, a
end program main
```

在上面的例子中, a(1) 赋值给 a(2), a(2) 赋值给 a(3), ..., a(8) 赋值给 a(9), 这些赋值是需要同时进行的, 最后 a 是 $[1, 1, 2, 3, 4, 5, 6, 7, 8]$. 同时进行多个计算就是并行计算, 当然这需要电脑有同时进行多个计算的能力. 如果电脑不行, 比如安安用家里的老破电脑跑上面的例子, 电脑每时每刻总是只能进行一个计算, 此时编译器和电脑会自己想办法操作一波, 模拟同时进行多个计算, 以保证最后干出来的结果和那些能同时进行多个计算的好电脑干出来的结果一样, 这样的计算就是并发 (concurrent) 计算. 又请看下面的例子.

```
program main
    implicit none
    integer :: i, a(9)
    a = [(i, i = 1, 9)]
```

```
    do i = 1, 8
        a(i+1) = a(i)
    end do
    print *, a
end program main
```

这个例子不一样, `a(1)` 赋值给 `a(2)`, 所以 `a(2)` 变成 `1`, `a(2)` 赋值给 `a(3)`, 所以 `a(3)` 变成 `1`, ..., `a(8)` 赋值给 `a(9)`, 所以 `a(9)` 变成 `1`, 这些赋值是需要按顺序依次进行的, 最后 `a` 是 $[1,1,1,1,1,1,1,1,1]$. 按顺序依次多个计算就是串行 (serial) 计算.

并行计算我们是非常需要的, 因为能省大量时间, 比如我们有一个大小为 1000×1000 的数组 (这样大的数组实践中并不罕见). 如果我们的电脑, 比如一台超级计算机, 能同时进行 1×10^6 个计算, 程序运行时我们给数组赋值很多次, 总共用时 1 秒. 如果我们的电脑, 比如安安家里的老破电脑, 只能同时进行 1 个计算, 程序运行时我们给数组赋值很多次, 总共用时 1×10^6 秒 > 10 天, 用时太多了. Fortran 的语法可以直接区分串行计算和并行计算, 这是 Fortran 的巨大优势 (Python 的 Numpy 已借鉴).

数组赋值还可以用 forall 结构, where 结构和 do concurrent 结构, 其中 forall 结构已被弃用, 加之同学们学完 where 结构和 do concurrent 结构估计已经晕菜了, 安安拒讲 forall 结构. 同学们如果弄不懂 where 结构和 do concurrent 结构可以先不掌握, 因为不用这俩玩意儿同学们也能干活儿, 就是程序跑得可能比较慢, 但其他数组赋值方法同学们必须掌握, 不然期末一定挂科呢.

7.5.1 where 结构

实践中我们经常会碰到需要 "根据数组元素的性质决定对元素的操作" 的情况. 假如有个数组 `a`, 我们想让 `a` 中 `<= 3` 的元素都加上 `2`, 让 `a` 中不 `<= 3` 且 `<= 6` 的元素都加上 `2`, 让 `a` 中其他元素都加上 `3`, 我们可以傻乎乎地用双重 do 循环加 if 判断这么写.

```
program main
    implicit none
    integer :: i, j, a(3, 3)
    a = reshape([(i, i = 1, 9)], [3, 3])
```

```
    do i = 1, 3
        do j = 1, 3
            if (a(i, j) <= 3) then
                a(i, j) = a(i, j) + 1
            else if (a(i, j) <= 6) then
                a(i, j) = a(i, j) + 2
            else
                a(i, j) = a(i, j) + 3
            end if
        end do
    end do
    do i = 1, 3
        print *, a(i, :)
    end do
end program main
```

Fortran 党看这个程序要跳脚, 因为第一这个程序太冗长, 第二这个程序的双重 do 循环里是完全串行计算的. 我们可以使用 where 结构进行掩码数组赋值, 改写上面的程序成下面的程序, 其中 `tag` 是 where 结构的标签, 可省去, 并且注释标注的行分别都是并行计算的. 相信同学们能总结出 where 结构的用法. 另外 where 结构和 if 结构的语法很像但又有点不一样, 请同学们自己对比, 不要混淆.

```
program main
    implicit none
    integer :: i, a(3, 3)
    a = reshape([(i, i = 1, 9)], [3, 3])
    tag: where (a <= 3)
        a = a + 1 ! Parallel!
    elsewhere (a <= 6)
        a = a + 2 ! Parallel!
    elsewhere
        a = a + 3 ! Parallel!
    end where tag
    do i = 1, 3
```

```
        print *, a(i, :)
    end do
end program main
```

又假如有个数组 a, 我们想让 a(1:2, 2:3) 中 <= 3 的元素都加上 2, 让 a(1:2, 2:3) 中不 <= 3 且 <= 6 的元素都加上 2(1:2, 2:3), 让 a(1:2, 2:3) 中其他元素都加上 3, 我们可以这么写.

```
program main
    implicit none
    integer :: i, a(3, 3)
    a = reshape([(i, i = 1, 9)], [3, 3])
    tag: where (a(1:2, 2:3) <= 3)
        a(1:2, 2:3) = a(1:2, 2:3) + 1 ! Parallel!
    elsewhere (a(1:2, 2:3) <= 6)
        a(1:2, 2:3) = a(1:2, 2:3) + 2 ! Parallel!
    elsewhere
        a(1:2, 2:3) = a(1:2, 2:3) + 3 ! Parallel!
    end where tag
    do i = 1, 3
        print *, a(i, :)
    end do
end program main
```

又假如有个数组 a 且有个数组 b, 它们形状一样, 我们想当 a(i, j) <= 3 时 b(i, j) 加上 1, 当不 a(i, j) <= 3 且 a(i, j) <= 6 时 b(i, j) 加上 2, 其他情况 b(i, j) 加上 3, 我们可以这么写.

```
program main
    implicit none
    integer :: i, a(3, 3), b(3, 3)
    a = reshape([(i, i = 1, 9)], [3, 3])
    b = reshape([(i, i = 1, 9)], [3, 3])
    tag: where (a <= 3)
        b = b + 1 ! Parallel!
    elsewhere (a <= 6)
```

```
        b = b + 2 ! Parallel!
    elsewhere
        b = b + 3 ! Parallel!
    end where tag
    do i = 1, 3
        print *, b(i, :)
    end do
end program main
```

看懂上面几个示例的同学会觉得 where 结构真好用, 但请这些同学不要高兴得太早, where 结构乍看着方便, 实则正因如此而有一个 Fortran 没学好的人不可能知道的坑. 要懂得这个坑, 同学们需要先学习 8.3.2 小节然后倒回头来看这里. 来看下面这个标准解释文档里给的经典程序.

```
program main
    implicit none
    integer :: i
    real :: a(9)
    a = [(real(i), i = -4, +4)]
    where(a > 0.0)
        ! log is invoked only for positive elements,
        ! because log is elemental.
        a = log(a)
    end where
    print *, a
    print *, a / sum(a)
    a = [(real(i), i = -4, +4)]
    where(a > 0.0)
        ! log is invoked for ALL elements,
        ! because sum is NOT elemental.
        a = a / sum(log(a))
    end where
    print *, a
end program main
```

在这个程序中, `a = log(a)` 里的 `log` 是逐元函数, 所以 `log(a)` 里的 `a` 代表的是 `a` 的元素 `> 0.0` 的部分, 所以这个程序中的 `a = log(a)` 相当于 `a(6:9) = log(a(6:9))`. 而 `a = a / sum(log(a))` 里的 `sum` 不是逐元函数, `sum(log(a))` 里的 `a` 出现在非逐元函数的 `sum` 后跟的括号 `()` 里, 这时请注意, `sum(log(a))` 里的 `a` 代表的不是 `a` 的元素 `> 0.0` 的部分, 而是 `a` 整个数组, 即便 `log` 是逐元函数也不影响这点, 所以这个程序中的 `a = a / sum(log(a))` 相当于 `a(6:9) = a(6:9) / sum(log(a(:)))`.

7.5.2 do concurrent 结构

有些程序不好用 where 结构改写, 比如下面的程序.

```
program main
    implicit none
    integer :: i, j, a(9), b(9)
    a = [(i, i = 1, 9)]
    b = [(j, j = 1, 9)]
    do i = 1, 3, 1
        do j = 7, 9, 1
            if ((a(i) > 1) .and. (b(j) < 9)) then
                b(i) = -j
                a(j) = -i
            end if
        end do
    end do
    print *, a
    print *, b
end program main
```

我们可以使用同样是并行计算的 do concurrent 结构[5]来改写上面的程序成下面的程序, 其中 `tag` 是 do concurrent 结构的标签, 可省去, 两个 `:1` 因为是 `:{m3}` 所以也可省去, 但 `{m1}` 和 `{m2}` 不可省去. 相信同学们能总结出 do concurrent 结构的用法. 另外 do concurrent 结构和 do 结构, where 结构和数组三元下标的语法很像但又有点不一样, 请同学们自己对比, 不要混淆.

[5]标准解释文档把 do concurrent 结构划成 do 结构中的一种, 但一般还是把 do concurrent 结构单看成一种结构.

```
program main
    implicit none
    integer :: i, j, a(9), b(9)
    a = [(i, i = 1, 9)]
    b = [(j, j = 1, 9)]
    tag: do concurrent (i = 1:3:1, j = 7:9:1, &
                        (a(i) > 1) .and. (b(j) < 9))
        b(i) = -j
        a(j) = -i
    end do tag
    print *, a
    print *, b
end program main
```

7.6 数组运算

如果一个代表运算 OP 的一元运算符 {op} 右跟 {q}, 且 {q} 是数组, 那么结果的形状与 {q} 一样, 且任意 {q} 的元素 $q_{s_1 \dots s_n}$, {op}{q} 中与 $q_{s_1 \dots s_n}$ 对应的元素为 OP $q_{s_1 \dots s_n}$. 示例如下.

```
program main
    implicit none
    integer :: i
    real :: a(3, 3)
    a = reshape([(real(i), i = 1, 9)], [3, 3])
    a = -a
    do i = 1, 3
        print *, a(i, :)
    end do
end program main
```

如果一个代表运算 OP 的二元运算符 {op} 左右分别跟 {q1} 和 {q2}, 且 {q1} 和 {q2} 是形状相同的数组, 那么结果的形状也相同, 且任意 {q1} 和 {q2} 的元素 $q_{1s_1 \dots s_n}$ 和 $q_{2s_1 \dots s_n}$, {q1}{op}{q2} 中与 $q_{1s_1 \dots s_n}$ 和 $q_{2s_1 \dots s_n}$ 对应的元素为 $q_{1s_1 \dots s_n}$ OP $q_{2s_1 \dots s_n}$. 示例如下.

```
program main
    implicit none
    integer :: i
    real :: a(3, 3)
    a = reshape([(real(i), i = 1, 9)], [3, 3])
    a = a * a
    do i = 1, 3
        print *, a(i, :)
    end do
end program main
```

如果一个代表运算 OP 的二元运算符 {op} 左右分别跟 {q1} 和 {q2}, 且 {q1} 和 {q2} 是形状不同的数组, 那么运算将失败. 错误示例如下.

```
program main
    implicit none
    integer :: i
    real :: a(3, 3)
    a = reshape([(real(i), i = 1, 9)], [3, 3])
    a = a(3, :) * a(3:3, :) ! Invalid!
    do i = 1, 3
        print *, a(i, :)
    end do
end program main
```

如果一个代表运算 OP 的二元运算符 {op} 左右分别跟 {q1} 和 {q2}, 且 {q1} 和 {q2} 一个是标量一个是数组, 那么先临时把 {q1} 和 {q2} 中是标量的那个替换成一数组, 此数组形状与 {q1} 和 {q2} 中是数组的那个一样, 且此数组中元素皆为 {q1} 和 {q2} 中是标量的那个, 然后按 {q1} 和 {q2} 是形状相同的数组时的方法进行运算. 示例如下.

```
program main
    implicit none
    integer :: i
    real :: a(3, 3)
    a = reshape([(real(i), i = 1, 9)], [3, 3])
```

```
    a = a(3, 3) * a
    do i = 1, 3
        print *, a(i, :)
    end do
end program main
```

本节讲的数组运算, 得到的都是非全数组. 所以下面这个程序不成.

```
program main
    implicit none
    integer :: i
    real :: a(3, 3)
    a = reshape([(real(i), i = 1, 9)], [3, 3])
    print *, (-a)(3, 3) ! Invalid!
end program main
```

但下面这个程序成, 因为数组取元素/片段优先于数组运算, 所以 `-a(3, 3)` 实际上代表 `-(a(3, 3))`, 没有问题.

```
program main
    implicit none
    integer :: i
    real :: a(3, 3)
    a = reshape([(real(i), i = 1, 9)], [3, 3])
    print *, -a(3, 3)
end program main
```

第八章　过程

假如我们需要用 Fortran 算阶乘 10!, 那还是很容易滴.

```
program main
    implicit none
    integer :: i, p
    p = 1
    do i = 1, 10
        p = p * i
    end do
    print *, p
end program main
```

假如我们需要用 Fortran 算组合数 $\mathrm{C}_7^3 = \frac{7!}{3!(7-3)!}$, 那就有点麻烦.

```
program main
    implicit none
    integer :: i, c1, c2, c3, c
    c1 = 1
    do i = 1, 7
        c1 = c1 * i
    end do
    c2 = 1
    do i = 1, 3
        c2 = c2 * i
    end do
    c3 = 1
    do i = 1, 7-3
```

```
        c3 = c3 * i
    end do
    c = c1 / (c2*c3)
    print *, c
end program main
```

麻烦的地方在于那个阶乘老是要 `do` 来 `do` 去, 不过就 `do` 三回, 还能活.

假如我们需要用 Fortran 算 CG 系数 $\langle 3,2;5,4|7,6\rangle$,

$$\begin{aligned}\langle j_1,m_1;j_2,m_2|j_3,m_3\rangle = \delta_{m_3,m_1+m_2}\Big[&(2j_3+1)\\ &\cdot\frac{(j_1+j_2-j_3)!(j_2+j_3-j_1)!(j_3+j_1-j_2)!}{(j_1+j_2+j_3+1)!}\\ &\cdot\prod_{i=1,2,3}(j_i+m_i)!(j_i-m_i)!\Big]^{1/2}\sum_{\nu\in F}[(-1)^\nu\nu!\\ &\cdot(j_1+j_2-j_3-\nu)!\\ &\cdot(j_1-m_1-\nu)!(j_2+m_2-\nu)!\\ &\cdot(j_3-j_1-m_2+\nu)!(j_3-j_2+m_1+\nu)!],\end{aligned}$$

那不知要 `do` 多少回, 算个大头鬼哟! 不算了, 准备卸 Fortran 了!

桥豆麻袋 (ちょっと待って), Fortran 是有法子能偷懒滴 (如果没有我第一个卸 Fortran), 比如算 C_7^3 可以这样.

```
program main
    implicit none
    integer :: c, factorial
    c = factorial(7) &
        / (factorial(3)*factorial(7-3))
    print *, c
end program main

function factorial(n) result(p)
    integer, intent(in) :: n
    integer :: p
    integer :: i
    p = 1
```

```
    do i = 1, n
        p = p * i
    end do
end function factorial
```

写成这样, 确实能少打几个字儿. 不知道我写的是什么也可以先猜猜看. 把 `function ...` 到 `end function ...` 单看成一个程序, `result(p)` 里的 `p` 就是 `factorial(n)` 里的 `n` 的阶乘, 如是这般, `factorial(7)` 就是 `7` 的阶乘, `factorial(3)` 就是 `3` 的阶乘, `factorial(7-3)` 就是 `7-3` 的阶乘, 非常完美! 那算 CG 系数也简单多了 (虽然还是很复杂), 只要算 `X` 的阶乘的时候无脑写上 `factorial(X)` 就成了, 不用 `do` 来 `do` 去了!

于是乎我们便发现, 想要玩 Fortran 而不是被 Fortran 玩, 就必须懂过程 (procedure). 因为一个 "重新创造轮子" 的梗 (请同志们自行搜索了解), 过程又俗称轮子 (wheel). 过程的定义是 "封装可以在程序执行期间直接调用的任意操作序列的实体", 玄玄乎乎的, 我们不理它. 我们可以直接把过程理解成程序运行时的一个操作, 比如上面的例子中 `function ...` 到 `end function ...` 就是 "计算 `n` 的阶乘" 这一操作. 使用过程后, 我们就进入面向过程程序设计 (procedure-oriented programming, POP) 阶段了.

啥叫面向过程呢? 众所周知, 置象于冰箱中, 步骤有三: 一开冰箱, 二塞大象, 三关冰箱. 用 Fortran 来写便这般.

```
program main
    ...
    implicit none
    ...
    call open_door()
    call put_in(elephant)
    call close_door()
end program main

subroutine open_door()
    ...
end subroutine open_door

subroutine put_in(what_put_in)
```

```
    ...
end subroutine put_in

subroutine close_door()
    ...
end subroutine close_door
```

这是个典型的面向过程的程序, 程序把自己要干的事分成几步骤, 每个步骤都单造一个过程表示. 这样一看主程序就知道这程序干仨事儿: 第一步 `call open_door()` 开冰箱, 第二步 `call put_in(elephant)` 塞大象, 第三步 `call close_door()` 关冰箱, 非常清楚. 至于具体咋么开的冰箱, 咋么塞的大象, 咋么关的冰箱, 看对应的 `subroutine` 到 `end subroutine` 里咋写的便能知晓. 把整个程序 (冰箱里塞大象) 拆成一个个步骤 (开冰箱, 塞大象, 关冰箱), 然后每个步骤造个过程, 这就是面向过程. 那为什么塞大象要写成 `put_in(elephant)` 而不是 `put_elephant_in()`? 这是因为说不准以后要把别的东西放进冰箱, 真有那天, 简单地把 `elephant` 换成别的东西就行了, 哦, 还要把大象拿出来, 再多造个名称为 `put_out` 的过程, 然后写 `call put_out(elephant)` 就行了.

8.1 外部过程

我们先细掰一些基本概念. 任何过程都是要用一堆字符表示的, 这堆字符便称为子程序 (subprogram). 过程和子程序的关系就像程序和源代码的关系一样. 子程序按摆的位置, 分为外部子程序 (external subprogram), 内部子程序 (internal subprogram) 和模块子程序 (module subprogram). 内部子程序瞅着没啥用还容易让同志们脑壳疼, 俺不打算讲, 模块子程序见第**??**章, 本章只讲外部子程序. 子程序按长的样子, 又分为子例行子程序 (subroutine subprogram) 和函数子程序 (function subprogram).

8.1.1 子例行子程序

我们来详细分析下面这个程序.

```
program main
    use iso_fortran_env, only: dp => real64
```

```
    implicit none
    real(dp) :: a, b, c
    real(dp) :: x, y, z
    a = 1.0_dp
    b = 2.0_dp
    c = 3.0_dp
    x = 4.0_dp
    y = 5.0_dp
    z = 6.0_dp
    call ab2bc_then_sumabc(x, y, z)
    print *, a, b, c
    print *, x, y, z
end program main

subroutine ab2bc_then_sumabc(a, b, c)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(in) :: a
    real(dp), intent(inout) :: b
    real(dp), intent(out) :: c
    real(dp) :: s
    c = b
    b = a
    s = a + b + c
    print *, s
end subroutine ab2bc_then_sumabc
```

1. 从 `subroutine ...` 到 `end subroutine ...` 便是子例行子程序了. 这部分可以和主程序放在一个文件里, 顺序也随便, 但通常是单独放在另一个文件里.

2. `subroutine` 和 `end subroutine` 后的 `XX` 称为子例行子程序名, 例如示例中 `XX` 为 `ab2bc_then_sumabc`. 一般存这个子程序的文件就会取成 `XX.f90`, 例如示例中的子程序可以存入名为 `ab2bc_then_sumabc.f90`

的文件中.

3. 子程序和主程序一样都是程序单元 (见 3.3 节), 一样得变量声明一波, 所以 `implicit none` 得加, 要用种别的话 `use iso_fortran_env` 也得加.

4. 子程序里有三个变量 `a`, `b`, `c`, 我在声明时加了 `, intent(...)`, 使这三个变量有了 intent 属性. 这三个变量在 `ab2bc_then_sumabc` 后的 `()` 里出现, 在 `()` 里出现的称为哑参量 (dummy argument), 只有哑参量能在声明时加 `, intent(...)`. 我称加 `, intent(in)` 的为只读 (read-only) 参量, 加 `, intent(inout)` 的为读写 (read-write) 参量, 加 `, intent(out)` 的为只写 (write-only) 参量.

5. 主程序里也有三个变量 `a`, `b`, `c`, 但只是变量名和子程序里的 `a`, `b`, `c` 一样, 实际上是不同的三个变量. 看我这示例, 子程序里 `a`, `b`, `c` 变来变去, 花里胡哨, 主程序里 `a`, `b`, `c` 岿然不动.

6. 主程序里出现 `call ab2bc_then_sumabc(x, y, z)`, `()` 里的 `x`, `y`, `z` 称为实参量 (actual argument). 实参量可以是任意数据实体, 也就是说实参量可以是变量, 也可以是常量或其他东东. 子程序里三个哑参量排排坐, 主程序里三个实参量排排坐, 位置一样的哑参量和实参量 (`a` 和 `x`, `b` 和 `y`, `c` 和 `z`) 称为对应的. 对应的哑参量和实参量在程序运行时会相互赋值来赋值去, 这称为参量结合 (argument association), 我们一般称为哑实结合.

7. 现在分析示例程序的运行过程. 程序当然从 `program main` 开始运行了. 按顺序一行一行运行, 前面不需讲解. 到 `call ...`, 就要说道说道了. 我们可以把主程序和子程序当成两个小人儿.

   (a) `call ab2bc_then_sumabc(x, y, z)`: 跳到子程序的开头, 也就是 `subroutine ab2bc_then_sumabc(a, b, c)`. `call ...` 这里程序运行的操作称为 “主程序调用 (invoke/call) 子程序” .

   (b) `subroutine ab2bc_then_sumabc(a, b, c)`: 子程序先按后面的变量声明语句声明好变量, 然后把所有只读的和读写的实参量赋值给对应的哑参量 (主程序里的 `x` 赋给子程序里的 `a`, 主程序里的 `y` 赋给子程序里的 `b`, 这当然就是传说中的哑实结合啦).

(c) 向下一行一行运行, 直到 `end subroutine ab2bc_then_sumabc`. 啰嗦一下具体过程. 首先主程序里的 `x` 赋给子程序里的 `a`, 主程序里的 `y` 赋给子程序里的 `b`, 所以子程序里的 `a` 为 `4.0_dp`, 子程序里的 `b` 为 `5.0_dp`. 然后 `c = b`, 子程序里的 `c` 为 `5.0_dp`, 然后 `b = a`, 子程序里的 `b` 为 `4.0_dp`, 然后 `s = a + b + c`, `s` 为 `13.0_dp`, 最后输出 `s` 的值 `13.0_dp`.

(d) `end subroutine ab2bc_then_sumabc`: 把所有读写的和只写的哑参量赋值给对应的实参量 (子程序里的 `b` 赋给主程序里的 `y`, 子程序里的 `c` 赋给主程序里的 `z`, 这当然也是传说中的哑实结合啦), 然后跳到 `call ab2bc_then_sumabc(x, y, z)` 的下一行.

然后主程序继续按顺序一行一行运行至 `end program main` 结束. 再啰嗦啰嗦, 子程序里的 `b` 赋给主程序里的 `y`, 子程序里的 `c` 赋给主程序里的 `z`, 所以 `x` 还是 `4.0_dp`, `y` 则变为 `4.0_dp`, `z` 则变为 `5.0_dp`.

啊! 上面那个程序终于分析完毕! 不仅是主程序能调用子程序, 任何程序单元都能调用子程序, 所以我们还可以玩点更花的. 假如我们现在要算组合数 C_7^3, 我们可以造一个主程序, 一个算组合数的子程序, 一个算阶乘的子程序, 然后让主程序调用算组合数的子程序, 算组合数的子程序调用算阶乘的子程序, 就像下面这样. 请同志们自己分析其运行过程.

```
program main
    implicit none
    integer :: result
    call combinatorial(7, 3, result)
    print *, result
end program main

subroutine combinatorial(n, m, comb)
    implicit none
    integer, intent(in) :: n
    integer, intent(in) :: m
    integer, intent(out) :: comb
    integer :: a, b, c
    call factorial(n, a)
```

```
    call factorial(m, b)
    call factorial(n-m, c)
    comb = a / (b*c)
end subroutine combinatorial

subroutine factorial(n, fact)
    implicit none
    integer, intent(in) :: n
    integer, intent(out) :: fact
    integer :: i
    fact = 1
    do i = 1, n
        fact = fact * i
    end do
end subroutine factorial
```

子程序灰常管用, 但也是要遵守一些禁令的. 首先哑实结合时, 哑参量和实参量的类型和种别都要相等, 也就是说莫得类型种别转化了. 比如下面这个程序, Gfortran 日常严格, 会直接报错, Ifx 日常宽松, 不出警告, 但输出的竟然是 10.00000…

```
program main
    use iso_fortran_env, only: sp => real32
    implicit none
    real(sp) :: a
    a = 10.0_sp
    call add_one(a)
    print *, a
end program main

subroutine add_one(a)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(inout) :: a
    a = a + 1.0_dp
```

```
end subroutine add_one
```

然后只读的 (加 `, intent(in)` 的) 哑参量不能在子程序运行的时候被赋值 (哑实结合时当然还是可以的), 比如下面这个程序跑不得, Ifx 和 Gfortran 都是如此. 这个规则是非常适当的, 因为如果我们可以确定一个哑参量不应当在子程序运行的时候被赋值, 我们就可以让这个哑参量成为只读的, 这样如果我们一不小心写错了, 在子程序运行的时候给这个哑参量赋值了, 编译器就能在编译时马上查出来, 免得我们乱跑程序跑了很久结果还不对.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp) :: a
    a = 10.0_dp
    call add_one(a)
    print *, a
end program main

subroutine add_one(a)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(in) :: a
    a = a + 1.0_dp
end subroutine add_one
```

注意间接的赋值也是不行的, 比如下面这个程序, `call add_one_(a)` 实际上给 `add_one` 里的 `a` 赋值了. 但这样 "隐晦的" 赋值, 编译器就不一定会查了, Gfortran 会报错, 而 Ifx 会直接放行. 但无论如何这么写都是不对的!

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp) :: a
    a = 10.0_dp
    call add_one(a)
    print *, a
```

```
end program main

subroutine add_one(a)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(in) :: a
    call add_one_(a)
end subroutine add_one

subroutine add_one_(a)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(inout) :: a
    a = a + 1.0_dp
end subroutine add_one_
```

还有只写的 (加 `, intent(out)` 的) 哑参量, 在子程序运行的一开始都是未定义的, 不论与哑参量结合的实参量是否未定义. 所以下面这程序中, 即使主程序里的 `a` 不是未定义的, 子程序 `add_one` 的第六行赋值时 "`=`" 右边的 `a` 也是未定义的, 这个程序老天也不知会出什么结果. 但 Ifx 和 Gfortran 有器规, 会把只写参量当成读写参量 (我猜是这样了啦), 一开始也实参量赋值给哑参量了, 所以结果没事. 但无论如何这么写都是不对的!

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp) :: a
    a = 10.0_dp
    call add_one(a)
    print *, a
end program main

subroutine add_one(a)
    ! Dummy argument may be undefined here!
    use iso_fortran_env, only: dp => real64
```

```
    implicit none
    real(dp), intent(out) :: a
    a = a + 1.0_dp
end subroutine add_one
```

正确标注哑参量为只读参量, 只写参量或读写参量 (加 `intent(...)`), 是灰常灰常必要的, 血泪教训告诉我们这么做能避免踩很多很多坑. 同志们一定不能怕麻烦, 老老实实一个个标注. 如果哑参量除哑实结合时外不被赋值, 就标只读, 如果哑参量在哑实结合时不需要被赋值, 就标只写, 剩下的标读写.

规范 24 *正确标注每个哑参量为只读参量, 只写参量或读写参量.*

子程序还有很多禁令, 请同志们自己写程序测试, 编译器会告诉大家答案的. 比如, 子程序名能和主程序名一样吗? 子程序名能和子程序里的变量的变量名一样吗? 子程序名能和调用子程序的程序单元里的变量的变量名一样吗? 子程序能直接或间接地调用自己吗?...

8.1.2 函数子程序

如果同志们没被子例行子程序弄晕, 那理解函数子程序便轻而易举了. 我们还是写一个计算组合数的子例行子程序, 不过我用 ... 省略一部分, 想来同志们自己补上没问题.

```
program main
    implicit none
    integer :: result
    call combinatorial(7, 3, result)
    print *, result
end program main

subroutine combinatorial(n, m, comb)
    implicit none
    integer, intent(in) :: n
    integer, intent(in) :: m
    integer, intent(out) :: comb
```

```
    integer :: a, b, c
    integer :: i
    ...
    comb = a / (b*c)
end subroutine combinatorial
```

上面这个程序完全可以用函数子程序改写成下面这样, 请同志们对比改写前后的样子.

```
program main
    implicit none
    integer :: combinatorial
    integer :: result
    result = combinatorial(7, 3)
    print *, result
end program main


function combinatorial(n, m) result(comb)
    implicit none
    integer, intent(in) :: n
    integer, intent(in) :: m
    integer :: comb
    integer :: a, b, c
    integer :: i
    ...
    comb = a / (b*c)
end function combinatorial
```

1. 从 `function ...` 到 `end function ...` 便是函数子程序了. 和子例行子程序一样, 可以和主程序放在一个文件里, 顺序也随便, 但通常是单独放在另一个文件里. `function` 和 `end function` 后的 `XX` (示例中为 `combinatorial`) 一样称为函数子程序名. 一般存这个子程序的文件也一样会取成 `XX.f90`.

2. 函数子程序名后的 `()` 里的当然是哑参量, `()` 后的 `result(...)` 里的变量 `...` (示例中为 `comb`) 相当于一个只写哑参量, 称为结果 (result),

但结果本身不是哑参量. 声明结果时不需也不能加 , intent(out).

3. 调用函数子程序后, 结果赋值给函数名和其之后的括号及括号内的内容组成的整体. 示例中, 一开始主程序里 7 和 3 赋值给子程序里 n 和 m, 最后子程序里结果 comb 赋值给主程序里 combinatorial(7, 3) 这一整个长串, combinatorial(7, 3) 就成为一个数据实体, 因此可以再赋值给主程序里的 result 变量.

4. 调用函数子程序前必须对函数子程序本身进行声明 (声明的类型和种别是函数子程序的结果的类型和种别), 子例行子程序是不需要的. 比如示例程序的主程序加了 integer :: combinatorial 一句, 因为整型是函数 combinatorial 的结果 comb 的类型.

按照当今的规范, 我们必须保证函数子程序的所有哑参量都是只读的 (结果不是哑参量). 如果不遵守此规范, 我保证同志们之后会无比头痛.

规范 25 *标注函数子程序的所有哑参量为只读参量.*

使用函数子程序的好处是调用函数子程序后会生成一个数据实体, 经验表明多数情况下这样能让我们偷懒少打几个字, 即便使用函数子程序前必须多加一行函数子程序的声明. 我把之前第 121 页用计算阶乘的子程序计算组合数的程序用函数子程序改写如下, 同志们会不会觉得看着简单一点?

```
program main
    implicit none
    integer :: combinatorial
    print *, combinatorial(7, 3)
end program main

function combinatorial(n, m) result(comb)
    implicit none
    integer, intent(in) :: n
    integer, intent(in) :: m
    integer :: comb
    integer :: factorial
    comb = factorial(n) &
           / (factorial(m)*factorial(n-m))
```

```
end function combinatorial

function factorial(n) result(fact)
    implicit none
    integer, intent(in) :: n
    integer :: fact
    integer :: i
    fact = 1
    do i = 1, n
        fact = fact * i
    end do
end function factorial
```

8.1.3 固有过程

为了让我们能快乐地玩轮子, 合格的 Fortran 编译器都已经自己造好了一大堆轮子, 称为固有过程 (intrinsic procedure), 我们直接调用就可以了. 同志们可猛看标准解释文档第 355 页表 16.1 或猛戳这个链接查询固有轮子有哪些怎么用, 没必要全背. 同志们造轮子前都应该先查查有没有已经造好的固有轮子可以用. 比如我们如果想算 π, 如果我们很熟悉固有轮子的话, 我们就会想到有个轮子 `acos`, 是算反余弦的, 我们用它算 $\arccos(-1)$ 即可. 另外固有过程都不需要声明, 即使固有过程是函数.

```
program main
    use iso_fortran_env, only: qp => real128
    implicit none
    print *, acos(-1.0_qp)
end program main
```

请同学们自己想办法用固有过程算 e.

8.2 过程中的变量

8.2.1 save 属性

下面这个程序, 连续输出三个 `1`, 这当然没有问题.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    call count()
    call count()
    call count()
end program main

subroutine count()
    implicit none
    integer :: n
    n = 0
    n = n + 1
    print *, n
end subroutine count
```

但下面这个程序输出的却是 1, 2, 3. 在这个程序里, 子程序 count 里的 n 声明的时候加了 , save, 并赋值 0, 这使 n 有了 save 属性, 成为已保存变量 (saved variable), 相当于 C 的静态局域变量 (static local variable). 第一次调用 count 的时候, n 一开始是 0, 然后 n = n + 1, n 就是 1. 而第二次调用 count 的时候, n 一开始并没有重新被赋值成 0, 而是保存着上一次调用到最后的值 1, 所以再次 n = n + 1 后 n 变成 2. 第三次调用 count 的时候, n 一开始是 2, 所以最后是 3.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    call count() ! 0 -> 1
    call count() ! 1 -> 2
    call count() ! 2 -> 3
end program main

subroutine count()
    implicit none
    integer, save :: n = 0
```

```
    n = n + 1
    print *, n
end subroutine count
```

有的同志可能会尝试在变量声明的时候直接给变量初始化, 因为这样可以偷一点懒, 但这么做是非常危险的, 因为这么做的时候, 即使没加 , save, 变量也悄咪咪地带上 save 属性了, 这是 Fortran 又一个经典的坑. 比如下面这个程序和上面那个程序是一样的, 但因为没写 , save, 同志们可能就会忘记变量 n 有 save 属性!

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    call count()
    call count()
    call count()
end program main


subroutine count()
    implicit none
    ! n has SAVE attribute!!!
    integer :: n = 0
    n = n + 1
    print *, n
end subroutine count
```

因此, 不要这么写, 除非声明时标有 , parameter 或 , save.

规范 26 除非已经明确标明变量有 *parameter* 属性或 *save* 属性, 否则不得在声明变量的时候直接给变量初始化.

8.2.2 过程中的字符串

在用子程序的时候碰上字符串, 就会比较麻烦, 因为哑实结合还和字符串的长度有关. 我们可以直接把字符串哑参量的长度写成 *, 这样字符串哑参量的长度会自动确定成与它结合的字符串实参量的长度, 非常方便. 下面的程序中, 子程序的 char_in 的长度会自动确定成主程序的 char_in 的长

度 3, 子程序的 char_out 的长度会自动确定成主程序的 char_out 的长度 1.

```
program main
    implicit none
    character(3), parameter :: char_in = 'Hi!'
    character(1) :: char_out
    print *, char_in
    call to_screen(char_in, char_out)
    print *, char_out
end program main

subroutine to_screen(char_in, char_out)
    implicit none
    character(*), intent(in) :: char_in
    character(*), intent(out) :: char_out
    print *, char_in, len(char_in)
    char_out = char_in
    print *, char_out, len(char_out)
end subroutine to_screen
```

8.2.3 过程中的数组

在用子程序的时候碰上数组, 就会比较麻烦, 因为哑实结合还和数组的形状有关. 我们必须细掰细掰.

显式形状数组

一个 n 维向量 $r = (r_1, \ldots, r_n)$ 的 1 范数为 $\|r\|_1 := \sum_{i=1}^{n} |r_i|$. 假如我们要算 $(-1, 2)$ 的 1 范数, 写个轮子还是很容易滴.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp) :: r(2)
    real(dp) :: norm_1
```

```
    r = [-1.0_dp, 2.0_dp]
    print *, norm_1(r)
end program main

function norm_1(r) result(norm)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(in) :: r(2)
    real(dp) :: norm
    norm = sum(abs(r))
end function
```

其中主程序里的 `r` 和函数里的 `r`, 形状都大大咧咧的写明在那里, 这就叫显式形状数组 (explicit-shape array). 但我们上面这个程序大有问题, 如果我们要算 $(-1, 2, -3)$ 的 1 范数, 因为函数里的 `r` 形状被定死为 `[2]`, 所以要出事儿. 这时我们可以这么写.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp) :: r_2(2), r_3(3)
    real(dp) :: norm_1
    r_2 = [-1.0_dp, 2.0_dp]
    r_3 = [-1.0_dp, 2.0_dp, -3.0_dp]
    print *, norm_1(r_2, size(r_2))
    print *, norm_1(r_3, size(r_3))
end program main

function norm_1(r, size_r) result(norm)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(in) :: r(size_r)
    integer, intent(in) :: size_r
    real(dp) :: norm
    norm = sum(abs(r))
```

```
end function
```

这个程序, 函数里哑参量 `r` 的形状是由哑参量 `size_r` 决定的, 这样的哑参量数组就叫可调数组 (adjustable array), 定义为显式形状数组中的一种. 虽然 `size_r` 的声明在 `r` 下面, 但放心, 声明 `r` 的时候会先查看 `size_r` 的值的.

第 82 页的小作业二是计算一个奇怪矩阵的所有元素的和, 我们可以把问题扩大点, 算任意 $n \times n$ 的那种矩阵的所有元素的和, 我们则可以这么写.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp) :: strange_mat_element_sum
    print *, strange_mat_element_sum(50)
end program main

function strange_mat_element_sum(n) result(s)
    use iso_fortran_env, only: dp => real64
    implicit none
    integer, intent(in) :: n
    real(dp) :: s
    real(dp) :: mat(n, n)
    integer :: i, j
    do i = 1, n
        do j = 1, n
            mat(i, j) = sqrt(real(i+j-1))
        end do
    end do
    s = sum(mat)
end function
```

这个程序, 函数里 `mat` 的形状是由哑参量 `n` 决定的, 这样的数组就叫自动数组 (automatic array), 也定义为显式形状数组中的一种. 自动数组和可调数组的区别是可调数组一定是哑参量, 自动数组一定不是哑参量.

假定形状数组

之前第 132 页用可调数组的轮子, 每次都要算数组的大小, 然后和子程序哑实结合, 还是麻烦. 用假定形状数组 (assumed-shape array) 就可以这么解决这个问题.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp) :: r_2(2), r_3(3)
    real(dp) :: norm_1
    r_2 = [-1.0_dp, 2.0_dp]
    r_3 = [-1.0_dp, 2.0_dp, -3.0_dp]
    print *, norm_1(r_2)
    print *, norm_1(r_3)
end program main


function norm_1(r) result(norm)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(in) :: r(:)
    real(dp) :: norm
    norm = sum(abs(r))
end function
```

在这个程序里, 函数中 `r` 就是假定形状数组, 其声明的时候写 `r(:)`, 1 个 `:` 表示 `r` 必须是 1 维数组, 形状是其对应的实参量的形状, 这样就不用每次都计算实参量的大小然后哑实结合了. 不过上面这个轮子是跑不了的, 即使编译器允许跑, 结果也很可能是错的, 因为按 Fortran 的语法, 有假定形状数组哑参量的过程, 必须带过程接口 (见 8.4 节), 所以要写成下面这个样子才行.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp) :: r_2(2), r_3(3)
    interface
```

```
        function norm_1(r) result(norm)
            use iso_fortran_env, only: dp => real64
            implicit none
            real(dp), intent(in) :: r(:)
            real(dp) :: norm
        end function
    end interface
    r_2 = [-1.0_dp, 2.0_dp]
    r_3 = [-1.0_dp, 2.0_dp, -3.0_dp]
    print *, norm_1(r_2)
    print *, norm_1(r_3)
end program main

function norm_1(r) result(norm)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(in) :: r(:)
    real(dp) :: norm
    norm = sum(abs(r))
end function
```

使用假定形状数组的时候, 我们还可以指定假定形状数组每一维的下界. 比如下面这个程序, 函数里的 `r` 是 2 维数组, 第 1 维下界是 0, 第 2 维下界没写, 默认是 1.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp) :: r_2(2, 1), r_3(3, 1)
    interface
        function norm_1(r) result(norm)
            use iso_fortran_env, only: dp => real64
            implicit none
            real(dp), intent(in) :: r(0:,:)
            real(dp) :: norm
```

```
        end function
    end interface
    r_2 = reshape([0.0_dp, -1.0_dp], [2, 1])
    r_3 = reshape([0.0_dp, -1.0_dp, 2.0_dp], [3, 1])
    print *, norm_1(r_2)
    print *, norm_1(r_3)
end program main

function norm_1(r) result(norm)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(in) :: r(0:,:)
    real(dp) :: norm
    norm = sum(abs(r))
end function
```

8.2.4 哑过程

有的时候我们需要把子程序本身当成参量, 比如我们如果要造个定积分的轮子, 我们就要被积函数, 下界, 上界三个参量, 被积函数参量当然得是函数啦. 是哑参量的过程称为哑过程 (dummy procedure). 下面就是个定积分轮子, 虽然看着非常复杂, 但确实能跑. 这个轮子将在 8.4 节中讲解.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    interface
        function identity(x) result(y)
            use iso_fortran_env, only: dp => real64
            implicit none
            real(dp), intent(in) :: x
            real(dp) :: y
        end function
    end interface
```

```
    real(dp) :: integrate
    print *, integrate(identity, 0.0_dp, 1.0_dp)
end program main

function integrate(f, a, b) result(s)
    ! Use trapezoidal rule.
    use iso_fortran_env, only: dp => real64
    implicit none
    interface
        function f(x) result(y)
            use iso_fortran_env, only: dp => real64
            implicit none
            real(dp), intent(in) :: x
            real(dp) :: y
        end function
    end interface
    real(dp), intent(in) :: a
    real(dp), intent(in) :: b
    real(dp) :: s
    real(dp) :: h
    integer :: i
    h = (b-a) / 10000
    s = (f(a)+f(b)) / 2
    do i = 1, 9999
        s = s + f(a+i*h)
    end do
    s = s * h
end function integrate

function identity(x) result(y)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(in) :: x
```

```
    real(dp) :: y
    y = x
end function identity
```

8.3 特殊过程

8.3.1 纯过程

按规范 25 写出的函数都是纯 (pure) 过程. 纯过程的定义有点复杂, 但同学们只需知道最典型的纯过程, 就是过程的哑参量 (函数结果不算) 都是只读参量的过程. 写纯过程子程序时, 在开头的 `subroutine` 或 `function` 前加 `pure`, 示例如下.

```
program main
    implicit none
    integer :: factorial
    print *, factorial(3)
end program main

pure function factorial(n) result(p)
    integer, intent(in) :: n
    integer :: p
    integer :: i
    p = 1
    do i = 1, n
        p = p * i
    end do
end function factorial
```

给哑参量加 intent 属性能帮我们避坑, 造纯过程能进一步帮我们避坑, 因为如果我们可以确定一个过程的哑参量都是只读参量, 我们就可以让这个过程成为纯过程, 这样如果我们一不小心写错了, 让哑参量不是只读参量了, 编译器就能在编译时马上查出来, 免得我们乱跑程序跑了很久结果还不对.

8.3.2 逐元过程

如果我们使用固有函数 log, 我们会发现它非常神奇, 它可以接收任意形状的数组, 并把数组中的所有元素分别都求自然对数, 形成形状相同的新数组. 其实很多常用的固有函数都如此.

```
program main
    implicit none
    integer :: i
    print *, log(real(1))
    print *, log([(real(i), i = 1, 9)])
end program main
```

这是因为 log 是逐元 (elemental) 过程. 逐元过程就是可以接收任意形状的数组, 并把数组中的所有元素分别都进行操作, 形成形状相同的新数组的过程. 逐元过程必须是纯过程. 我们可以把 8.3.1 小节的示例中的纯过程子程序中 pure 改成 elemental, 将纯过程子程序改写成逐元过程子程序, 注意 elemental 已表明过程是纯过程, 我们没必要再写 pure.

```
program main
    implicit none
    integer :: i, factorial
    print *, factorial(3)
    print *, factorial([(i, i = 1, 9)])
end program main


elemental function factorial(n) result(p)
    integer, intent(in) :: n
    integer :: p
    integer :: i
    p = 1
    do i = 1, n
        p = p * i
    end do
end function factorial
```

不过上面这个轮子是跑不了的, 即使编译器允许跑, 结果也很可能是错的, 因为按 Fortran 的语法, 被使用的逐元过程, 必须带过程接口 (见 8.4 节), 所以要写成下面这个样子才行.

```
program main
    implicit none
    integer :: i
    interface
        elemental function factorial(n) result(p)
            integer, intent(in) :: n
            integer :: p
        end function factorial
    end interface
    print *, factorial(3)
    print *, factorial([(i, i = 1, 9)])
end program main

elemental function factorial(n) result(p)
    integer, intent(in) :: n
    integer :: p
    integer :: i
    p = 1
    do i = 1, n
        p = p * i
    end do
end function factorial
```

8.3.3 递归过程

因为 $n! = n \cdot (n-1)!$, 所以我们可以玩点花活儿. 我们可以尝试把 8.3.1 小节的示例改写成下面这样, 其中函数 `factorial` 调用自己, 表示 $n! = n \cdot (n-1)!$. 但下面这个轮子是跑不了的, 因为 Fortran 不允许一个普通的过程调用自己.

```
program main
```

```
    implicit none
    integer :: factorial
    print *, factorial(3)
end program main

function factorial(n) result(p)
    integer, intent(in) :: n
    integer :: p
    if (n == 1) then
        p = 1
    else
        ! factorial(n) == n * factorial(n-1)
        p = n * factorial(n-1)
    end if
end function factorial
```

我们可以在子程序开头的 `subroutine` 或 `function` 前加 `recursive`, 将过程改造成递归 (recursive) 过程. 递归过程就是可以调用自己的过程. 于是花活儿就像下面这样玩成了.

```
program main
    implicit none
    integer :: factorial
    print *, factorial(3)
end program main

recursive function factorial(n) result(p)
    integer, intent(in) :: n
    integer :: p
    if (n == 1) then
        p = 1
    else
        ! factorial(n) == n * factorial(n-1)
        p = n * factorial(n-1)
    end if
```

```
end function factorial
```

不过上面这个例子我们还要细掰细掰. 同学们可能会这么分析上面这个例子.

1. 主程序第四行要算 `factorial(3)`, 所以函数 `factorial` 的哑元 `n` 变成 3.
2. `n /= 1`, 所以 `p = n * factorial(n-1)`, 要算 `factorial(n-1)`, 所以函数 `factorial` 的哑元 `n` 变成 `n-1`, 所以 `n` 变成 2.
3. `n /= 1`, 所以 `p = n * factorial(n-1)`, 要算 `factorial(n-1)`, 所以函数 `factorial` 的哑元 `n` 变成 `n-1`, 所以 `n` 变成 1.
4. `n == 1`, 所以 `p = 1`, `p` 变成 1, 然后到 `end function factorial`, `p` 是结果, 所以 `factorial(3)` 是 1.

可这明显不对呀, 程序跑出来结果应该是 6 才对嘛. 要想弄懂上面这个例子, 我们需要更准确地理解过程的调用. 过程调用时, 严格意义上说, 不是过程本身被调用了, 而是过程连同过程中的所有常量变量都被复制出分身, 然后分身被调用了. 所以上面这个例子应分析如下.

1. 主程序第四行要算 `factorial(3)`, 所以函数 `factorial` 被复制出分身 $\texttt{factorial}_1$, 哑元 $\texttt{n}_1$ 变成 3.
2. $\texttt{n}_1$ `/= 1`, 所以 $\texttt{p}_1$ `= n` $_1$ `* factorial(n` $_1$`-1)`, 要算 `factorial(n` $_1$`-1)`, 所以函数 `factorial` 被复制出分身 $\texttt{factorial}_2$, $\texttt{n}_2$ `= n` $_1$`-1`, 哑元 $\texttt{n}_2$ 变成 2.
3. $\texttt{n}_2$ `/= 1`, 所以 $\texttt{p}_2$ `= n` $_2$ `* factorial(n` $_2$`-1)`, 要算 `factorial(n` $_2$`-1)`, 所以函数 `factorial` 被复制出分身 $\texttt{factorial}_3$, $\texttt{n}_3$ `= n` $_2$`-1`, 哑元 $\texttt{n}_3$ 变成 1.
4. $\texttt{n}_3$ `== 1`, 所以 $\texttt{p}_3$ `= 1`, $\texttt{p}_3$ 变成 1, 然后到 `end function factorial`, $\texttt{p}_3$ 是结果, 而 $\texttt{p}_2$ `= n` $_2$ `* factorial(n` $_2$`-1)` 时调用 $\texttt{factorial}_3$, 所以 `factorial(n` $_2$`-1)` 是 1.
5. $\texttt{n}_2$ 是 2, 所以 $\texttt{p}_2$ 变成 2, 然后到 `end function factorial`, $\texttt{p}_2$ 是结果, 而 $\texttt{p}_1$ `= n` $_1$ `* factorial(n` $_1$`-1)` 时调用 $\texttt{factorial}_2$, 所以 `factorial` `(n` $_1$`-1)` 是 2.

6. n_1 是 3, 所以 p_1 变成 6, 然后到 `end function factorial`, p_1 是结果, 而主程序算 `factorial(3)` 时调用 $factorial_1$, 所以 `factorial(3)` 是 6.

啊! 这么分析就正确了!

如果想让过程又是纯过程又是递归过程, 我们把 `pure` 和 `recursive` 都加在子程序开头的 `subroutine` 或 `function` 前即可 (`pure` 和 `recursive` 顺序任意), 像下面这样.

```
program main
    implicit none
    integer :: factorial
    print *, factorial(3)
end program main

recursive pure function factorial(n) result(p)
    integer, intent(in) :: n
    integer :: p
    if (n == 1) then
        p = 1
    else
        ! factorial(n) == n * factorial(n-1)
        p = n * factorial(n-1)
    end if
end function factorial
```

想让过程又是逐元过程又是递归过程亦是同理, 请同学们自己尝试.

如果能理解透递归过程的运作原理, 在可以使用递归过程的时候用递归过程, 程序是会很简洁很秀的. 但是如果想让程序跑得快, 递归过程最好不用. 因为程序每调用一个过程, 都会在内存中专门为这个过程分配一块存储空间, 术语称作进栈 (push stack), 而用递归过程时, 过程可能会被调用灰常多次 (例如 `factorial(1000)` 会调用 `factorial` 1000 次), 最后的结果是递归过程被多多地调用, 内存被多多地占着, 电脑不堪重负, 程序跑得慢甚至崩溃出错.

8.4 过程接口

之前第 134 页, 第 136 页, 第 140 页的轮子用了过程接口 (procedure interface). 过程接口可分为三类: 特定接口 (specific interface), 泛型接口 (generic interface) 和抽象接口 (abstract interface).

8.4.1 特定接口

特定接口相当于过程的声明. 通常情况下, 子例行不需要声明, 函数也只需要声明结果就可以了. 但如果碰上了以下情形之一, 那么就一定要加特定接口[1]:

- 过程有一个哑参量, 此哑参量满足下列条件之一:
  - 是延迟长度字符型变量 (见 4.4 节) 或延迟形状数组 (见7.3 节);
  - 是假定形状数组;
- 过程是函数且结果是数组;
- 过程是哑过程;
- 过程是逐元过程.

这不一定需要背, 编译器应该是要告诉我们的. 我们可以先不加接口, 然后如果编译器告诉我们 “Explicit interface required for …” 或 “Expected a procedure for argument …” 或其他七七八八的话, 那就是要加接口了.

特定接口都放在从 `interface` 到 `end interface` 的一整块里, 这一整块称为接口块 (interface block), 一个接口块里可以放一堆接口. 接口是过程的一部分, 我们只要把过程的变量声明部分下面的执行部分都删掉, 然后变量声明部分, 除了哑参量和结果的声明, 其他声明都删掉, 然后复制粘贴到接口块里就可以了.

我们通过实战来熟悉这个过程. 假设我们现在想搞个算单位矩阵的轮子, 我们正常地这么一写, 结果跑不得.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
```

[1] 一定要加特定接口的情形还有很多, 其他同志们暂时不需要掌握.

```
    real(dp) :: i(3, 3)
    i = eye(3)
end program main

function eye(n) result(mat)
    use iso_fortran_env, only: dp => real64
    implicit none
    integer, intent(in) :: n
    real(dp) :: mat(n, n)
    !----------------------------------------
    integer :: i
    mat = 0.0_dp
    do i = 1, n
        mat(i, i) = 1.0_dp
    end do
    !----------------------------------------
end function eye
```

我们需要在主程序中加个接口. 我们先在主程序的声明部分写上 `interface` 和 `end interface`, 然后把整个 `eye` 子程序复制粘贴进去. (我们还可以用列选择[2]来调整格式)

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    interface
        function eye(n) result(mat)
            use iso_fortran_env, only: dp => real64
            implicit none
            integer, intent(in) :: n
            real(dp) :: mat(n, n)
            !----------------------------------------
            integer :: i
            mat = 0.0_dp
```

[2]不知道什么是列选择的同志请自行搜索了解.

```
            do i = 1, n
                mat(i, i) = 1.0_dp
            end do
            !----------------------------------------
        end function eye
    end interface
    real(dp) :: i(3, 3)
    i = eye(3)
end program main

function eye(n) result(mat)
    use iso_fortran_env, only: dp => real64
    implicit none
    integer, intent(in) :: n
    real(dp) :: mat(n, n)
    !----------------------------------------
    integer :: i
    mat = 0.0_dp
    do i = 1, n
        mat(i, i) = 1.0_dp
    end do
    !----------------------------------------
end function eye
```

然后我们定睛一看, 两个注释行中间的部分, 第一行是 `i` 的声明, `i` 既不是哑参量也不是结果, 所以删去, 后面几行是执行部分也删去. 删完以后就成下面这个样子.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    interface
        function eye(n) result(mat)
            use iso_fortran_env, only: dp => real64
            implicit none
```

```
            integer, intent(in) :: n
            real(dp) :: mat(n, n)
        end function eye
    end interface
    real(dp) :: i(3, 3)
    i = eye(3)
end program main

function eye(n) result(mat)
    use iso_fortran_env, only: dp => real64
    implicit none
    integer, intent(in) :: n
    real(dp) :: mat(n, n)
    !----------------------------------------
    integer :: i
    mat = 0.0_dp
    do i = 1, n
        mat(i, i) = 1.0_dp
    end do
    !----------------------------------------
end function eye
```

然后这个轮子就能跑了, 欧耶!

同志们会不会觉得这么做挺麻烦的? 俺也觉得, 可是这也是没办法的. Fortran 在设置这个接口规则的时候可是经过深思熟虑的, 因为造编译器的那些人, 凭他们的经验告诉我们, 有些情况 (比如第 144 页列出来的), 如果没有接口, 编译器很容易编译错程序, 然后出大事情. 不过还是有办法能让我们少费点脑子, 那就是使用模块 (见第**??**章), 但如果不需要加接口, 那么造一个模块反而比较费事...

这里还有个问题, 如果我们碰到个固有过程需要接口怎么办, 比如我们如果要算 sin 的积分, 我们用固有过程 `sin` 直接干算不得.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
```

```
    real(dp) :: integrate
    print *, integrate(sin, 0.0_dp, 1.0_dp)
end program main

function integrate(f, a, b) result(s)
    ! Use trapezoidal rule.
    use iso_fortran_env, only: dp => real64
    implicit none
    interface
        function f(x) result(y)
            use iso_fortran_env, only: dp => real64
            implicit none
            real(dp), intent(in) :: x
            real(dp) :: y
        end function
    end interface
    real(dp), intent(in) :: a
    real(dp), intent(in) :: b
    real(dp) :: s
    real(dp) :: h
    integer :: i
    h = (b-a) / 10000
    s = (f(a)+f(b)) / 2
    do i = 1, 9999
        s = s + f(a+i*h)
    end do
    s = s * h
end function integrate
```

最无脑的办法就是我们再造个过程, 把固有过程变成外部过程, 像下面这样, 然后再加上接口即可. 请同志们自己给下面这个程序补上接口.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
```

```
    real(dp) :: integrate
    print *, integrate(sin_, 0.0_dp, 1.0_dp)
end program main

function integrate(f, a, b) result(s)
    ! Use trapezoidal rule.
    use iso_fortran_env, only: dp => real64
    implicit none
    interface
        function f(x) result(y)
            use iso_fortran_env, only: dp => real64
            implicit none
            real(dp), intent(in) :: x
            real(dp) :: y
        end function
    end interface
    real(dp), intent(in) :: a
    real(dp), intent(in) :: b
    real(dp) :: s
    real(dp) :: h
    integer :: i
    h = (b-a) / 10000
    s = (f(a)+f(b)) / 2
    do i = 1, 9999
        s = s + f(a+i*h)
    end do
    s = s * h
end function integrate

function sin_(x) result(y)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(in) :: x
```

```
    real(dp) :: y
    y = sin(x)
end function sin_
```

8.4.2 泛型接口

泛型接口是一个比较妙的东东. 假如我们现在要造个算符号函数 sgn 的轮子, 那是轻而易举的.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp) :: sgn
    print *, sgn(10.0_dp)
end program main

function sgn(x) result(y)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(in) :: x
    real(dp) :: y
    if (x > 0.0_dp) then
        y = 1.0_dp
    else if (x < 0.0_dp) then
        y = -1.0_dp
    else
        y = 0.0_dp
    end if
end function sgn
```

但有个问题, 就是假如如果我们要算四精度, 只能另造个轮子.

```
program main
    use iso_fortran_env, only: dp => real64, &
                               qp => real128
    implicit none
```

```
    real(dp) :: sgn_real64
    real(qp) :: sgn_real128
    print *, sgn_real64(10.0_dp)
    print *, sgn_real128(10.0_qp)
end program main

function sgn_real64(x) result(y)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(in) :: x
    real(dp) :: y
    if (x > 0.0_dp) then
        y = 1.0_dp
    else if (x < 0.0_dp) then
        y = -1.0_dp
    else
        y = 0.0_dp
    end if
end function sgn_real64

function sgn_real128(x) result(y)
    use iso_fortran_env, only: qp => real128
    implicit none
    real(qp), intent(in) :: x
    real(qp) :: y
    if (x > 0.0_qp) then
        y = 1.0_qp
    else if (x < 0.0_qp) then
        y = -1.0_qp
    else
        y = 0.0_qp
    end if
end function sgn_real128
```

这就让我们有点小小的不开心, Fortran 的语法也太严格了! 我们希望能造个 sgn, 又能算双精度又能算四精度. 用泛型接口, 我们就可以偷鸡摸狗地“做成” 这件事, 把上面的 sgn_real64 和 sgn_real128 “粘起来” .

```
program main
    use iso_fortran_env, only: dp => real64, &
                               qp => real128
    implicit none
    interface sgn
        function sgn_real64(x) result(y)
            use iso_fortran_env, only: dp => real64
            implicit none
            real(dp), intent(in) :: x
            real(dp) :: y
        end function sgn_real64
        function sgn_real128(x) result(y)
            use iso_fortran_env, only: qp => real128
            implicit none
            real(qp), intent(in) :: x
            real(qp) :: y
        end function sgn_real128
    end interface
    print *, sgn(10.0_dp)
    print *, sgn(10.0_qp)
end program main

function sgn_real64(x) result(y)
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp), intent(in) :: x
    real(dp) :: y
    if (x > 0.0_dp) then
        y = 1.0_dp
    else if (x < 0.0_dp) then
```

```
        y = -1.0_dp
    else
        y = 0.0_dp
    end if
end function sgn_real64

function sgn_real128(x) result(y)
    use iso_fortran_env, only: qp => real128
    implicit none
    real(qp), intent(in) :: x
    real(qp) :: y
    if (x > 0.0_qp) then
        y = 1.0_qp
    else if (x < 0.0_qp) then
        y = -1.0_qp
    else
        y = 0.0_qp
    end if
end function sgn_real128
```

我们会发现泛型接口和特定接口很像, 只不过 `interface` 后多了一串 (示例中为 `sgn`), 然后 `sgn_real64` 和 `sgn_real128` 的接口都写在接口块里. 这么写后, 电脑看到 `sgn(10.0_dp)`, 就会发现 `10.0_dp` 是双精度的, 然后电脑就会在标 `sgn` 的泛型接口里找, 看 `sgn_real64` 的接口, 发现 `sgn_real64` 这个函数参量是双精度的, 匹配, 又看 `sgn_real64` 的接口, 发现 `sgn_real128` 这个函数参量是四精度的, 不匹配, 于是乎电脑就会自动把 `sgn(10.0_dp)` 里的 `sgn` 当成 `sgn_real64` 了. 然后电脑看到 `sgn(10.0_qp)`, 也是一样, 只不过最后是把 `sgn` 当成 `sgn_real128`. 这样就仿佛有一个又能算双精度又能算四精度的 `sgn` 了!

Fortran 的类型–种别–维数匹配 (type-kind-rank compatibility, TKR compatibility) 超严格的, 但我们用泛型接口便能有所突破, 这当然是大好事. 但同志们还是可能不开心, 因为同志们会发现上面的 `sgn_real64` 和 `sgn_real128` 其实长得基本上是一模一样, 就是变量种别不同, 如果我们还要 `sgn` 能接收单精度参量, 能接收整型参量, 岂不是要复制粘贴写一大堆一

模一样的过程, 读起来还脑壳疼? 非常遗憾, 据我的了解, Fortran 自己确实就只能这么玩儿了. 不过用第**??**章介绍的预处理器的话, 就可以省事儿不少还易读 (我用过, 超好用), 但这已经是超 Fortran 的内容了.

8.4.3 抽象接口

特定接口只对应一个过程, 而抽象接口则对应所有特征 (characteristic) 相同的过程, 我们来看下面这个程序.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    interface
        function eye(n) result(mat)
            use iso_fortran_env, only: dp => real64
            implicit none
            integer, intent(in) :: n
            real(dp) :: mat(0:n-1, 0:n-1)
        end function eye
        function minkowski(n) result(eta)
            use iso_fortran_env, only: dp => real64
            implicit none
            integer, intent(in) :: n
            real(dp) :: eta(0:n-1, 0:n-1)
        end function minkowski
    end interface
    real(dp), dimension(0:3, 0:3) :: mat_i, mat_m
    mat_i = eye(4)
    mat_m = minkowski(4)
end program main

function eye(n) result(mat)
    use iso_fortran_env, only: dp => real64
    implicit none
    integer, intent(in) :: n
```

```
    real(dp) :: mat(0:n-1, 0:n-1)
    integer :: i
    mat = 0.0_dp
    do i = 0, n-1
        mat(i, i) = 1.0_dp
    end do
end function eye

function minkowski(n) result(eta)
    use iso_fortran_env, only: dp => real64
    implicit none
    integer, intent(in) :: n
    real(dp) :: eta(0:n-1, 0:n-1)
    integer :: i
    eta = 0.0_dp
    eta(0, 0) = -1.0_dp
    do i = 1, n-1
        eta(i, i) = 1.0_dp
    end do
end function minkowski
```

这个程序, 接口块里写了两个过程的接口, 这当然没有问题. 但同志们会发现这两个过程其实接口长得 “一模一样”, 都是函数, 都只有一个整型只读参量, 结果的类型, 结果的种别和结果的形状也一样, 只不过各种名称 (过程名, 参量名, 结果名) 不一样而已. 接口长得 “一模一样” 的过程, 我们称为特征相同的. 能不能趁机偷一小点懒? 能, 像下面这样.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    abstract interface
        function dim2mat(dim) result(mat)
            use iso_fortran_env, only: dp => real64
            implicit none
            integer, intent(in) :: dim
```

```
            real(dp) :: mat(0:dim-1, 0:dim-1)
        end function dim2mat
    end interface
    procedure(dim2mat) :: eye, minkowski
    real(dp), dimension(0:3, 0:3) :: mat_i, mat_m
    mat_i = eye(4)
    mat_m = minkowski(4)
end program main

function eye(n) result(mat)
    use iso_fortran_env, only: dp => real64
    implicit none
    integer, intent(in) :: n
    real(dp) :: mat(0:n-1, 0:n-1)
    integer :: i
    mat = 0.0_dp
    do i = 0, n-1
        mat(i, i) = 1.0_dp
    end do
end function eye

function minkowski(n) result(eta)
    use iso_fortran_env, only: dp => real64
    implicit none
    integer, intent(in) :: n
    real(dp) :: eta(0:n-1, 0:n-1)
    integer :: i
    eta = 0.0_dp
    eta(0, 0) = -1.0_dp
    do i = 1, n-1
        eta(i, i) = 1.0_dp
    end do
end function minkowski
```

和特定接口相比, 抽象接口的 `interface` 前多了个 `abstract`. 接口块里过程名 `dim2mat` 的接口和过程 `eye`, `minkowski` 的特定接口长得 "一模一样": 都是函数, 都只有一个整型只读参量等等. 至于抽象接口里函数名是 `dim2mat`, 参量名是 `dim`, 结果名是 `mat`, 无关紧要, 这些名称都是可以随便取的[3], 只需保证抽象接口的特征和本需加特定接口的过程的特征相同即可. 然后我们只需要再写 `procedure(dim2mat) :: eye, minkowski` 就相当于给和 `dim2mat` 特征相同的过程 `...` (示例中是 `eye` 和 `minkowski`) 加特定接口了.

[3]如果碰上了什么规矩, 编译器会告诉我们.

第九章　输入与输出

之前我们都只是小打小闹而已, 现在我们来干一些大活儿. 我们来玩玩 Gaia DR3. 我们想知道 Gaia DR3 里的哪个源距银道面最远且最远有多远, 比较不动脑筋地做这件事呢, 我们想知道哪个源的 $|\sin b|/p$ 最大, 其中 p 是源的视差, b 是源的银纬. 但这里有一个问题, 就是 Gaia DR3 显然太大了, 同志们得交个几十杯奶茶钱的网费, 所以我们先用 Gaia DR3 的一个小样本试试水.

首先我们要在这个页面里找到 ReadMe 点进去, 找到 "File Summary".

```
File Summary:
--------------------------------------------------------
 FileName    Lrecl Records Explanations
--------------------------------------------------------
ReadMe          80        . This file
gaiadr3.sam   1788     1000 Gaia DR3 source catalog
                              (1811709771 sources)
                            ......
```

从这个 "File Summary" 可知, `gaiadr3.sam` 是星表[1], 一共 1000 行, 每行 1788 个字符. 不对呀, 后面明明写着一共 1811709771 个源嘛, 怎么只有 1000 行? 那当然是因为这只是个 1000 个源的样本啦, 文件名后缀可是 `.sam` 呢. 于是乎我们要在这个页面里找到 FTP, 里头找到 `gaiadr3.sam.gz` 下载下来然后解压得到 `gaiadr3.sam`.

回到 ReadMe, 下面有 `gaiadr3.sam` 的 "Byte-by-byte Description".

```
Byte-by-byte Description of file: gaiadr3.sam
--------------------------------------------------------
```

[1] 认不得英文词儿的请上天文学名词词典查询.

```
  Bytes   Format Units Label   Explanations
--------------------------------------------------------
   1-  28  A28   ---   DR3Name Unique source
                                designation (unique
                                across all Data
                                Releases)
                                (designation)
 129- 137  F9.4  mas   Plx     ? Parallax (parallax)
 980- 994 F15.11 deg   GLAT    Galactic latitude (b)
```

从这个 “Byte-by-byte Description” 可知, `gaiadr3.sam` 的每一行, 第 1–28 个字符都是源的编号, 第 129–137 个字符都是源的视差 (单位为毫角秒), 第 980–994 个字符都是源的银纬 (单位为度). 好, 现在就造轮子!

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    integer, parameter :: sam_len = 1000
    real(dp), parameter :: &
        mas2rad = acos(-1.0_dp)/(1.8e2_dp*3.6e6_dp)
    real(dp), parameter :: &
        deg2rad = acos(-1.0_dp)/(1.8e2_dp)
    real(dp) :: p(sam_len), b(sam_len)
    real(dp) :: d(sam_len)
    p = p * mas2rad
    b = b * deg2rad
    d = abs(sin(b)) / p ! Unit: AU
    print *, maxloc(d), maxval(d)
end program main
```

这轮子现在有个大问题, 我们需要把星表里的数据转移到数组 `p` 和 `b` 中, 难道用列选择将星表里的数据复制粘贴到轮子里? 好家伙, 一个轮子两千多行, 其中两千行是数据, 要是处理原始星表, 就有三十六亿行是数据, 这不是要崩溃[2]? 这时候我们就需要让 Fortran 自己干转移数据的事情.

[2]不仅我们要崩溃, 文本编辑器也要崩溃…

9.1 文件

我们平常所说的文件, Fortran 称其为外部文件 (external file), 不过一般来说外部文件得是纯文本文件[3]. 既然是纯文本文件, 就相当于一个字符串, 说到字符串, 就让我们想到字符串变量. Fortran 称字符串变量为内部文件 (internal file), 将外部文件与内部文件合称为文件 (file).

我们需要让 Fortran 知道我们要用什么文件. 内部文件 Fortran 是认得的, 因为是自家的字符串变量嘛, 但外部文件 Fortran 不认得. 为了让 Fortran 认得外部文件, 我们要将外部文件打开 (open). 下面这个轮子, Fortran 会在第三行后认得 `test.txt`, 这个 `test.txt` 应该和编译完最后生成的 `.exe` 文件[4]在一个目录里.

```
program main
    implicit none
    open(10, file='test.txt')
end program main
```

其中 `10` 这个位置填的是文件单位 (file unit), 说白了就是一个编号, 这编号必须是非负整数, 而且最好不小于 `10`, 因为小于 `10` 的编号可能有些奇奇怪怪的用途. `file=` 后加的是文件路径, 不知什么是文件路径的同志们请自行补习. 上个轮子的第三行就是让 Fortran 认得文件路径为 `test.txt` 的文件, 并且称其为 10 号文件.

使用完外部文件后我们应当将其关闭 (close), 也就是让 Fortran 忘记外部文件. 我们在上面那个轮子中加上一行, Fortran 就会在第四行后忘记 10 号文件 (即 `test.txt`).

```
program main
    implicit none
    open(10, file='test.txt')
    close(10)
end program main
```

每当我们暂时不需要使用外部文件时, 我们都应该将其关闭, 这是一个好习惯. 原因一是打开文件的时候, 文件单位可能会重. 例如下面的程序, 我

[3]某些编译器可能有器规来应对二进制文件. 不知什么是纯文本文件和二进制文件的同志们赶紧恶补.

[4]注意不是源代码文件.

们让文件单位 10 和 test1.txt 连接, 又让文件单位 10 和 test2.txt 连接, 这个程序能跑, 但因为文件单位 10 重了, 所以电脑其实会悄咪咪地把 test1.txt 关掉, 然后打开 test2.txt, 但悄咪咪关掉文件这事我们很可能会没注意!

```
program main
    implicit none
    open(10, file='test1.txt')
    open(10, file='test2.txt') ! test1.txt closed!
end program main
```

原因二是打开文件的时候, 被打开的文件可能被占住, 其他程序用不了. 想象一下, 同学们跑一个程序, 需要用一个文件里的数据, 同学们用程序打开了文件但没关闭, 然后老师或同学们的同学也要跑一个程序, 也需要用这个文件里的数据, 结果同学们居然把文件占住了, 打不开这个文件, 同学们得被暴打一顿. 所以请同学们养成关掉没用文件的好习惯.

规范 27 *在不需要使用外部文件的时候将其关闭.*

注意, 内部文件是不需要也没法打开或关闭的.

上面的轮子和下面的轮子等价.

```
program main
    implicit none
    open(10, file='test.txt', status='UNKNOWN')
    close(10)
end program main
```

其中 status='UNKNOWN' 的意思是这个轮子在打开文件的时候进行了一波操作, 但这波操作是编译器自己决定的, 也就是说不同的编译器可能会有不同的操作, 所以不写 status=... 有点危险[5]. 我们可以把 'UNKNOWN' 换成 'OLD', 'NEW', 或 'REPLACE'. 换成 'OLD' 的话, test.txt 必须在轮子运行前存在, 运行时直接打开. 换成 'NEW' 的话, test.txt 必须在轮子运行前不存在, 运行时, Fortran 会自己造一个空白 test.txt 文件并打开. 换成 'REPLACE' 的话, 如果 test.txt 在轮子运行前不存在, 则还是造一个空白 test.txt 文件并打开, 如果 test.txt 在轮子运行前存在, 则会把原来的

[5] 如果摸透了编译器会怎么决定的话, 偷个懒不写也罢了.

test.txt 直接删掉, 再造一个新的空白 test.txt 文件并打开. 请同志们自行造轮子实验上述操作.

上面的程序其实一直有个问题, 就是因为 Fortran 太上古了, 所以用整数的文件单位来表示文件, 但看整数的文件单位没法儿直接看出文件单位到底对应什么文件. 我们可以巧用字面常量给文件单位 “加注释”, 例如写成下面这样.

```
program main
    implicit none
    integer, parameter :: test1_txt = 10
    integer, parameter :: test2_txt = 11
    open(test1_txt, file='test1.txt')
    open(test2_txt, file='test2.txt')
    close(test1_txt)
    close(test2_txt)
end program main
```

这样看 test1_txt 就知对应文件 test1.txt, 看 test2_txt 就知对应文件 test2.txt. 但本笔记里的例子都很简单, 安安就偷懒不用字面常量给文件单位 “加注释” 了.

最后指明, 文件单位在各个程序单元中是通用的, 也就是说我们完全可以在主程序中打开一文件然后在子程序中关闭此文件, 或在子程序中打开一文件然后在主程序中关闭此文件. 示例如下.

```
program main
    implicit none
    open(10, file='test1.txt')
    call open_and_close()
    close(11)
end program main

subroutine open_and_close()
    implicit none
    open(11, file='test2.txt')
    close(10)
end subroutine open_and_close
```

9.2 读取与写入

读取 (read) 是把文件里的内容变成 Fortran 数据实体, 写入 (write) 是把 Fortran 数据实体变成文件里的内容. 让我们来看下面这个轮子.

```
program main
    implicit none
    real :: e, pi
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, *) exp(1.0), acos(-1.0)
    close(10)
    open(10, file='test.txt', status='OLD', &
         action='READ', position='REWIND')
    read(10, *) e, pi
    print *, e, pi
    close(10)
end program main
```

其中 `write(10, *) ...` 那行是把 `exp(1.0)` 和 `acos(-1.0)` 写入到 10 号文件里去, 所以打开 `test.txt` 就会看到 e 和 π. `read(10, *) ...` 那行则是把 10 号文件里的内容读取出来赋值给 `e` 和 `pi`, 所以最后会输出 e 和 π.

上面这个轮子还有些细节. `action=` 后的字符串如果是 `'READ'`, 表示打开的文件是只读的, 不能写入. `action=` 后的字符串如果是 `'WRITE'`, 表示打开的文件是只写的, 不能读取. `action=` 后的字符串如果是 `'READWRITE'`, 表示打开的文件是读写的, 读取写入皆可. 这和之前 8.1.1 小节的只读参量只写参量读写参量是很像的. 如果不加 `action=...`, 则是编译器自己决定是只读的只写的还是读写的, 这又有点危险了, 所以 `action=...` 还是要加的[6].

`position=...` 则指明打开文件后的 "文件定位", 加 `position=...` 的原因还是不加的话文件定位会由编译器自己决定[7]. 同志们不需知道文件定位是什么, 只需记得只写文件加 `position='APPEND'`, 写入时会写入在文件的最后, 只读文件加 `position='REWIND'`, 读取时会从文件的开头读取. 读

[6] 同脚注5.

[7] 同脚注5.

写文件嘛, 我们选择不玩儿. 下面用一个长长的轮子来详细讲解.

```
program main
    implicit none
    real :: e, pi
    real :: val
    e = exp(1.0)
    pi = acos(-1.0)
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    close(10)
    open(10, file='test.txt', status='OLD', &
         action='WRITE', position='APPEND')
    write(10, *) e + pi
    write(10, *) e - pi
    close(10)
    open(10, file='test.txt', status='OLD', &
         action='WRITE', position='APPEND')
    write(10, *) e * pi
    write(10, *) e / pi
    close(10)
    open(10, file='test.txt', status='OLD', &
         action='READ', position='REWIND')
    read(10, *) val
    print *, val
    read(10, *) val
    print *, val
    close(10)
    open(10, file='test.txt', status='OLD', &
         action='READ', position='REWIND')
    read(10, *) val
    print *, val
    read(10, *) val
    print *, val
```

```
    close(10)
end program main
```

第一次打开关闭文件, 因为 `status='REPLACE'`, 后面也没有写入, 所以最后得到一个空白文件. 第二次打开关闭文件, 先写入 $e+\pi$, $e+\pi$ 后面再写入 $e-\pi$. 第三次打开关闭文件, 在文件最后先写入 $e\cdot\pi$, 再写入 e/π, 所以最后文件里依次是 $e+\pi$, $e-\pi$, $e\cdot\pi$, e/π. 第四次打开关闭文件, 先读取文件开头的 $e+\pi$ 赋值给 `val`, 再读取下面的 $e-\pi$ 赋值给 `val`. 第五次打开关闭文件, 还是先读取文件开头的 $e+\pi$ 赋值给 `val`, 再读取下面的 $e-\pi$ 赋值给 `val`, 所以最后输出的依次是 $e+\pi$, $e-\pi$, $e+\pi$, $e-\pi$.

注意每次 `read` 或 `write` 后, 下次 `read` 或 `write` 都会从下一行开始. 下面这个轮子, 第一次 `write` 后 `1` 和 `2` 写入在第一行, 第二次 `write` 后 `3` 和 `4` 写入在第二行, 第一次 `read` 从第一行开始, 因为后面只跟着一个 `val1`, 所以 `val1` 为 `1`, 第二次 `read` 从第二行开始, 因为后面只跟着一个 `val2`, 所以 `val2` 为 `3`, `2` 和 `4` 则被无视!

```
program main
    implicit none
    integer :: val1, val2
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, *) 1, 2
    write(10, *) 3, 4
    close(10)
    open(10, file='test.txt', status='OLD', &
         action='READ', position='REWIND')
    read(10, *) val1 ! val1=1, while 2 is ignored!
    read(10, *) val2 ! val2=3, while 4 is ignored!
    print *, val1, val2
    close(10)
end program main
```

如果读取或写入时碰上数组, 则会把数组里的所有元素按元素顺序挨个读取或写入, 比如下面这个轮子, 先挨个写入 1 到 9, 再写入 10, 读取则是反向操作. 特别提醒, `one2nine(1, 3)` 是 7, `one2nine(3, 1)` 是 3, 同志们要是迷惑了请自行复习第七章数组元素顺序的内容.

```
program main
    implicit none
    integer :: i
    integer :: one2nine(3, 3), ten
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, *) reshape([(i, i = 1, 9)], [3, 3]), 10
    close(10)
    open(10, file='test.txt', status='OLD', &
         action='READ', position='REWIND')
    read(10, *) one2nine, ten
    print *, one2nine, ten
    print *, one2nine(1, 3), one2nine(3, 1)
    close(10)
end program main
```

把文件单位换成字符串变量即可读取和写入内部文件了. 注意内部文件是不需要也没法打开或关闭的, 所以没法加 `position=...`, 每次读取或写入都是从内部文件的开头第一个字符开始的, 不论之前读取写入几次. 下面这个轮子, Ifx 可以把 e^{π} 写入 `f` 并读取至 `val`[8], 但 Gfortran 不行, 要让 Gfortran 行, `f` 的长度必须大于 `16`. 原因见 9.3 节.

```
program main
    implicit none
    character(16) :: f
    real :: val
    write(f, *) exp(1.0) ** acos(-1.0)
    print *, f
    read(f, *) val
    print *, val
end program main
```

我们还可以再把字符串变量换成 `*`, 输出的话 `*` 通常代表电脑屏幕. 比如下面这个轮子就会把 `Hello, world!` “写入电脑屏幕”, 电脑屏幕上就出

[8]这就实现了数字型和字符型的相互转换.

现 `Hello, world!` 了. `print {fmt}`, 等价于 `write(*, {fmt})`, 也就是说 `print *`, 等价于 `write(*, *)`.

```
program main
    implicit none
    write(*, *) 'Hello, world!'
    print *, 'Bye bye, world!'
end program main
```

输入的话 `*` 则通常代表键盘. 比如下面这个轮子, 运行到第五行时程序会停下来, 我们在命令行 (cmd 呀 powershell 呀之类的东东) 中可以打字儿, 假设打入 `98 54e1` 然后回车, `98` 和 `54e1` 就赋值给 `mat(1, 1)` 和 `mat(1, 2)` 了, 第七行会干和第五行一样的事儿, 但语法更简单, 假设打入 `7.6 3.2e0` 然后回车, `7.6` 和 `3.2e0` 就赋值给 `mat(2, 1)` 和 `mat(2, 2)` 了, 最后输出矩阵行列式. 注意 "`*`" 也是不需要且没法打开或关闭的. `read {fmt}`, 等价于 `read(*, {fmt})`, 也就是说 `read *`, 等价于 `read(*, *)`.

```
program main
    implicit none
    real :: mat(2, 2)
    print *, 'Calculate determinant (det) '// &
             'of 2x2 matrix'
    print *, 'row 1 of matrix:'
    read(*, *) mat(1, 1), mat(1, 2)
    print *, 'row 2 of matrix:'
    read *, mat(2, :)
    print *, 'det:', &
             mat(1, 1)*mat(2, 2) - mat(1, 2)*mat(2, 1)
end program main
```

相信同志们现在懂得如何读取和写入外部文件, 内部文件和 "`*`" 了, 不过同志们还需要注意一些细节问题. 在上个轮子中, 我一开始就打出了 "算 2×2 矩阵的行列式" 的提示, 告诉用这个轮子的人这个轮子会干什么, 后面输入和输出是什么也有提示. 如果不这么干, 用轮子的人估计会一脸懵, 轮子在干什么, 自己要干什么, 最后得到的又是什么统统弄不懂, 哪怕是造轮子

的人自己可能也会懵, 忘了自己干了什么在干什么该干什么[9].

规范 28 输出充足的与程序功能及输入输出有关的提示信息.

9.3 编辑符

如果我们分别用 Ifx 和 Gfortran 跑下面这个处心积虑造出来的简单轮子, 我们会发现结果的格式有点区别, Ifx 的结果是用科学计数法表示的而 Gfortran 的不是.

```
program main
    implicit none
    print *, 7.0e7
end program main
```

这是由于我们让编译器自己决定输入输出的格式, 有时这会出事情, 比如之前第 167 页有个轮子, Ifx 跑得了但 Gfortran 跑不了, 原因在于 Gfortran 在把 `exp(1.0) ** acos(-1.0)` 写入内部文件 `f` 时会在数字前后补上些空格之类的, 按 Gfortran 自己的规定, 最少要写入 17 个字符 (包含 1 个换行符), 而 `f` 长度只有 16 不够长, Ifx 规定不同, 就没这个问题.

我们可以自己规定输入输出的格式, 比如下面这个轮子[10], 第一个输出的结果一定不是科学计数法表示的, 第二个一定是.

```
program main
    implicit none
    print "(F10.1)", 7.0e7
    print "(ES6.1E1)", 7.0e7
end program main
```

再比如修改第 167 页那个轮子可得下面这个轮子, Gfortran 也是可以跑的.

```
program main
    implicit none
    character(16) :: f
    real :: val
```

[9] 如果轮子只是自己用, 还敢赌自己能晓得轮子在干什么, 那想偷懒就偷懒呗.

[10] Ifx 编译时会出现一个 “remark”, 是些建议, 我们选择无视, 不过听从 Ifx 的建议也是好事.

```
    write(f, "(F15.12)") exp(1.0) ** acos(-1.0)
    print *, f
    read(f, "(F15.12)") val
    print *, val
end program main
```

可见自己规定格式就是把原来的 * 换成一个字符串, 这个字符串里是 (...), 称为格式声明 (format specification), * 则是代表编译器自己决定格式.

我们回归一开始玩 Gaia DR3 遇到的问题, 现在我们可以把第 160 页的轮子补成下面的样子, 其中倒数第二行我们写 format 后跟格式声明 (注意格式声明只是 (...), 不带引号), 前面加标号, 这样我们就能在倒数第三行用标号代替字符串了[11]. 不过呢, 同志们会发现怎么输出的最大距离是无穷大, 那是因为星表里有些源根本没有视差的数据, 硬读出来是 0, 看来研究还是没那么容易做的... 不过我们可以用 7.5.1 小节介绍的 where 结构, 把视差测量值不为 0 的源的数据提出来然后再算, 但安安懒得写了, 请同学们自己练习.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    integer, parameter :: sam_len = 1000
    real(dp), parameter :: &
        mas2rad = acos(-1.0_dp)/(1.8e2_dp*3.6e6_dp)
    real(dp), parameter :: &
        deg2rad = acos(-1.0_dp)/(1.8e2_dp)
    real(dp) :: p(sam_len), b(sam_len)
    real(dp) :: d(sam_len)
    integer :: i
    open(10, file='gaiadr3.sam ', status='OLD', &
         action='READ', position='REWIND')
    do i = 1, sam_len
        read(10, "(128X,F9.4,842X,F15.11)") p(i), b(i)
    end do
    close(10)
```

[11] 同志们这么写的时候, 带标号的行最好就在用标号的行的下面, 不然查格式时真的是会找不到的...

```
    p = p * mas2rad
    b = b * deg2rad
    d = abs(sin(b)) / p
    print 1000, maxloc(d), maxval(d)
    1000 format ('Object: ', I3, ', ', ES11.4, ' AU')
end program main
```

每个格式声明都由 () 里的一堆编辑符 (edit descriptor) 组成, 编辑符间用 , 隔开, 每个编辑符都代指一个操作, 比如上面的轮子读取时编辑符依次是 `128X`, `F9.4`, `842X`, `F15.11`, 依次代表跳过 128 个字符不读, 读占 9 个字符的 4 位小数的实数, 跳过 842 个字符不读, 读占 15 个字符的 11 位小数的实数. 话说我是怎么知道这么读就行的? 我们再次打开 ReadMe 里的 `gaiadr3.sam` 的 "Byte-by-byte Description", 里头写着.

```
Byte-by-byte Description of file: gaiadr3.sam
--------------------------------------------------------------------------------
  Bytes   Format Units Label   Explanations
--------------------------------------------------------------------------------
   1-  28  A28    ---   DR3Name Unique source
                                designation (unique
                                across all Data
                                Releases)
                                (designation)
 129- 137  F9.4   mas   Plx     ? Parallax (parallax)
 980- 994 F15.11 deg   GLAT    Galactic latitude (b)
```

同志们看表里分明写着视差的格式是 "F9.4", 银纬的格式是 "F15.11", 这不就是编辑符么. 首先视差从第 129 个字符开始, 所以我们要先跳过前 128 个字符, 写上 `128X`, 然后把 `F9.4` 无脑抄过去, 然后本来是读到第 138 个字符, 但银纬从第 980 个字符开始, 所以要再跳过 $980 - 138 = 842$ 个字符, 写上 `842X`, 然后无脑抄上 `F15.11`, 因为每次 `read` 完再 `read` 都会从下一行开始, 所以后面的字符就不用管了. 这里我们的活儿干得不是那么优雅, 因为到底要跳过几个字符我们还要手算一波, 说不定算错了呢. 最好有个轮子, 能接收 ReadMe 文件里的内容, 然后自动算出要跳过几个字符并给出编辑符来, 不过安安还没空开发, 同志们可以自己造个轮子来当练习. 另外给同志们留一个学完本章后的作业: 再次修改上面的轮子, 输出第 `maxloc(d)` 个源的

“Unique source designation”.

如果我们把一些编辑符用 () 起来, 前面加个数, 数是几就表示把 () 里的编辑符重复几遍, 比如下面这个轮子写入和读取时格式声明是一样的 (要不然就不知道读出什么东西了).

```
program main
    implicit none
    real :: e, pi
    real :: a, s, m, d
    e = exp(1.0)
    pi = acos(-1.0)
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, 1006) e+pi, e-pi, e*pi, e/pi
    1006 format (F6.4,ES11.4,F6.4,ES11.4)
    close(10)
    open(10, file='test.txt', status='OLD', &
         action='READ', position='REWIND')
    read(10, 1005) a, s, m, d
    1005 format (2(F6.4,ES11.4))
    print *, a, s, m, d
    close(10)
end program main
```

() 还可以嵌套, 比如下面这个轮子两次写入时格式声明是一样的.

```
program main
    implicit none
    ! A staff.
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, 1001) 'OXOX', 'OXXO', 'OX-X', 'OXXX'
    1001 format ('X','O','X','O','}',A4,'}', &
                 'X','O','X','O','}',A4,'}', &
                 'X','O','X','O','}',A4,'}', &
                 'X','O','X','O','}',A4,'}')
```

```
    close(10)
    open(10, file='test.txt', status='OLD', &
         action='WRITE', position='APPEND')
    write(10, 1002) 'OXOX', 'OXXO', 'OX-X', 'OXXX'
    1002 format (4(2('X','O'),'}',A4,'}'))
    close(10)
end program main
```

编辑符又分三大类: 数据编辑符 (data edit descriptor), 控制编辑符 (control edit descriptor), 字符串编辑符 (character string edit descriptor). 接下来我们一个个扒.

9.3.1 数据编辑符

数据编辑符和读取与写入时的数据实体是一一对应的, 比如下面这个轮子, `A4` 对应 `'ello'`, `A5` 对应 `'world'`, 其他编辑符不是数据编辑符, 没有对应的数据实体.

```
program main
    implicit none
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "('H',A4,',',1X,A5,'!')") &
        'ello', 'world'
    close(10)
end program main
```

每个数据编辑符还都对应于一种数据类型, 比如下面这个轮子照道理是跑不得的, 因为 `I1` 是整型编辑符而 `0.0` 是实型的, 但 Ifx 居然能跑, 可恶的器规又出现了…

```
program main
    implicit none
    print "(I1)", 0.0
end program main
```

数据编辑符都可以直接在前面加数来表示重复, 比如下面这个轮子三个格式声明全都是一样的.

```
program main
    implicit none
    ! Violin Strings.
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(A,A,A,A)") 'G', 'D', 'A', 'E'
    write(10, "(4(A))") 'G', 'D', 'A', 'E'
    write(10, "(4A)") 'G', 'D', 'A', 'E'
    close(10)
end program main
```

整型编辑符

Iw 编辑符表示一共输出 w 个字符. 我们需要保证 $w > 0$. 下面这个轮子, 前面几次写入是正常的, 但最后一次写入的是一堆 `*`, 因为 `1000000` 分明是个 7 位数, 却只能输出 6 个字符, 编译器只能摆烂了.

```
program main
    implicit none
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(I6)") 1000
    write(10, "(I6)") 10000
    write(10, "(I6)") 100000
    write(10, "(I6)") 1000000
    close(10)
end program main
```

下面这个轮子, 后两次写入编译器都摆烂了, 因为 "`-`" 也占 1 个字符, 千万千万要注意!

```
program main
    implicit none
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(I6)") -1000
```

```
    write(10, "(I6)") -10000
    write(10, "(I6)") -100000
    write(10, "(I6)") -1000000
    close(10)
end program main
```

I$w.m$ 编辑符则表示一共输出 w 个字符, 其中数字字符 (0–9) 至少 m 个, 当然必须 $m \leqslant w$. 下面这个轮子, 1000 只占 4 个字符, 所以前面要补上一个 0, 10000 占 5 个字符, 正常输出, 100000 占 6 个字符, 也正常输出, 因为是至少输出 5 个字符, 1000000 还是一堆 *.

```
program main
    implicit none
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(I6.5)") 1000
    write(10, "(I6.5)") 10000
    write(10, "(I6.5)") 100000
    write(10, "(I6.5)") 1000000
    close(10)
end program main
```

特别注意输入时 I$w.m$ 的 $.m$ 会被无视, 也就是说 I$w.m$ 等价于 Iw. 下面这个轮子是能正常运作的, 因为读取的时候 I6.5 等价于 I6.

```
program main
    implicit none
    integer :: k, dak, hk
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(I6)") 1000
    write(10, "(I6)") 10000
    write(10, "(I6)") 100000
    close(10)
    open(10, file='test.txt', status='OLD', &
        action='READ', position='REWIND')
```

```
    read(10, "(I6.5)") k
    read(10, "(I6.5)") dak
    read(10, "(I6.5)") hk
    print *, k, dak, hk
    close(10)
end program main
```

反过来, 下面这个轮子也是能正常运作的, 虽然写入文件的时候 1000 前加了 0, 但读取的时候开头的 0 都会被无视.

```
program main
    implicit none
    integer :: k, dak, hk
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(I6.5)") 1000
    write(10, "(I6.5)") 10000
    write(10, "(I6.5)") 100000
    close(10)
    open(10, file='test.txt', status='OLD', &
        action='READ', position='REWIND')
    read(10, "(I6)") k
    read(10, "(I6)") dak
    read(10, "(I6)") hk
    print *, k, dak, hk
    close(10)
end program main
```

但下面这个轮子就不对了, 因为写入 6 个字符但只读取 5 个字符, 这意味着最后的 0 没被读取, 结果 1000, 10000, 100000 被读成 100, 1000, 10000.

```
program main
    implicit none
    integer :: k, dak, hk
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
```

```
    write(10, "(I6.5)") 1000
    write(10, "(I6.5)") 10000
    write(10, "(I6.5)") 100000
    close(10)
    open(10, file='test.txt', status='OLD', &
        action='READ', position='REWIND')
    read(10, "(I5)") k
    read(10, "(I5)") dak
    read(10, "(I5)") hk
    print *, k, dak, hk ! Wrong!
    close(10)
end program main
```

实型编辑符

F$w.d$ 编辑符表示一共输出 w 个字符, 其中小数部分 d 个字符, 我们需要保证 $w > d \geqslant 0$. 下面这个轮子, 第一个输出是正常的, 第二个要输出 "-12." 再加 2 个字符, 一共 6 个字符, 但却只能输出 5 个, 编译器又摆烂了.

```
program main
    implicit none
    print "(F5.2)", 12.3456789
    print "(F5.2)", -12.3456789
end program main
```

输入的时候 F$w.d$ 的 $.d$ 则会被无视, 但 $.d$ 不能被省略. 下面这个轮子, F11.99 看着很鬼, 一共 11 个字符, 小数部分 99 个? 但 .99 会被无视, 反正就是读 11 个字符然后赋值给 `val` 完事, 所以轮子是跑得的. 在下面的轮子中我们还写入双精度后读取成单精度, 这也没问题, 读取和写入可以跨种别.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    real :: val
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
```

```
    write(10, "(F11.9)") 0.123456789_dp
    close(10)
    open(10, file='test.txt', status='OLD', &
        action='READ', position='REWIND')
    read(10, "(F11.99)") val
    print *, val
    close(10)
end program main
```

但下面这个轮子的结果是不对的, 因为写入和读取的 1000 没有小数点, 在没有小数点的时候, $.d$ 复活了, 编译器会认为读取到的字符的最后 d 个是小数部分, 读到 1000, 最右边 0 是小数部分, 前面 100 是小数部分, 当然错啦!

```
program main
    implicit none
    real :: val
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(I4)") 1000
    close(10)
    open(10, file='test.txt', status='OLD', &
        action='READ', position='REWIND')
    read(10, "(F4.1)") val
    print *, val ! Wrong!
    close(10)
end program main
```

有时我们会碰到 $+\infty$, $-\infty$, 和 NaN. Fortran 规定输入输出时用前面加正负号的字符串 INF 或 INFINITY 表示 $\pm\infty$, 用字符串 NAN 表示 NaN, 这些字符串不分大小写. 如果遇到 $\pm\infty$ 或 NaN, 则不论输入输出 F$w.d$ 的 $.d$ 都会被无视[12], 比如下面的轮子里 d 可以随便乱写, 只需保证 $d \geqslant 0$.

```
program main
    implicit none
    real :: one, zero
```

[12] 虽然 Fortran 官方规则中没写明, 但想来是这样的.

```
    real :: val1, val2, val3, val4
    one = 1.0
    zero = 0.0
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write (10, "(F9.12)") +(one/zero)
    write (10, "(F9.34)") -(one/zero)
    write (10, "(F9.56)") +(zero/zero)
    write (10, "(F9.78)") -(zero/zero)
    close(10)
    open(10, file='test.txt', status='OLD', &
        action='READ', position='REWIND')
    read(10, "(F9.87)") val1
    read(10, "(F9.65)") val2
    read(10, "(F9.43)") val3
    read(10, "(F9.21)") val4
    print *, val1, val2, val3, val4
    close(10)
end program main
```

F 编辑符经常会不大好用, 举个例子, 假如我们要输出 1.234×10^{-3}, 1.234, 1.234×10^{3} 量级不同的三个数, 编辑符都用 F5.3, 那就只有 1.234 的输出是比较正常的, 1.234×10^{-3} 有效数字丢了, 1.234×10^{3} 干脆输出不了, 但麻烦的是实际的观测数据的量级有差别是经常出现的事情.

```
program main
    implicit none
    print "(F5.3)", 1.234e-3 ! output: 0.001
    print "(F5.3)", 1.234    ! output: 1.234
    print "(F5.3)", 1.234e+3 ! output: *****
end program main
```

这时我们可以用 E 编辑符, E$w.d$ 表示一共输出 w 个字符, 输出时先将实数表示成 $a \times 10^{n}$, $0.1 \leqslant a < 1$, n 为整数, 把 a 转换成小数部分占 d 个字符的字符串 `a`, n 转换成开头带正负号的字符串 `n`, 然后输出字符串 `a//'E'//n`. 我们需要保证 $w > d \geqslant 0$. 最后输出的字符串 `a//'E'//n` 中, 开头的 `0` 和中

间的 E 可省略, 但输出的字符串去掉 a 后剩下的部分必占 4 个字符, 所以保证 $w \geqslant 3+d+4$ 一般就没什么事情了. 假如我们想保留 4 位有效数字, 编辑符设成 E11.4 就好啦.

```
program main
    implicit none
    print "(E11.4)", 1.234e-3 ! output:  0.1234E-02
    print "(E11.4)", 1.234    ! output:  0.1234E+01
    print "(E11.4)", 1.234e+3 ! output:  0.1234E+04
end program main
```

用 E$w.d$ 编辑符的时候, n 是三位数则 E 必须省略, n 是四位数就只能罢工了, 虽然平常我们基本上不会用上这么极端的数...

```
program main
    use iso_fortran_env, only: qp => real128
    implicit none
    print "(E11.4)", 1e10_qp   ! output:  0.1000E+11
    print "(E11.4)", 1e100_qp  ! output:  0.1000+101
    print "(E11.4)", 1e1000_qp ! output: ***********
end program main
```

这时我们可以用 E$w.d$Ee 编辑符, Ee 表示 n 为正负号后接 e 个数字的字符串, 其他和 E$w.d$ 相同, 除了 E 不可省略外. 这时我们需要额外保证 $e>0$, 再保证 $w \geqslant 3+d+2+e$ 一般就没什么事情了.

```
program main
    use iso_fortran_env, only: qp => real128
    implicit none
    ! output: 0.1000E+0011
    print "(E13.4E4)", 1e10_qp
    ! output: 0.1000E+0101
    print "(E13.4E4)", 1e100_qp
    ! output: 0.1000E+1001
    print "(E13.4E4)", 1e1000_qp
end program main
```

不过 E$w.d$E0 也是合法的编辑符, 其中 E0 表示 e 等于 n 的位数.

```
program main
    use iso_fortran_env, only: qp => real128
    implicit none
    ! output: 0.1000E+11
    print "(E13.4E0)", 1e10_qp
    ! output: 0.1000E+101
    print "(E13.4E0)", 1e100_qp
    ! output: 0.1000E+1001
    print "(E13.4E0)", 1e1000_qp
end program main
```

和 F 编辑符类似, 输入的时候 $.d$, Ee, E0 都会被无视, $.d$ 不能被省略. 不仅如此, 用 F 编辑符写入后还能用 E 编辑符读取, 用 E 编辑符写入后也能用 F 编辑符读取, 读取的时候字符 `E` 还不分大小写.

```
program main
    implicit none
    character(13+8+1) :: f
    real :: val_fe, val_ef
    integer :: i
    write(f, "(F13.1, E8.1)") 1e10, 1e10
    do i = 1, len(f)
        if (f(i:i)=='E') then
            f(i:i) = 'e'
        end if
    end do
    print *, f
    read(f, "(E13.13, F8.8)") val_fe, val_ef
    print *, val_fe, val_ef
end program main
```

E 编辑符也可以应付 $\pm\infty$ 和 NaN, 遇到 $\pm\infty$ 或 NaN 的时候 d 和 e 被无视, 其他和 F 编辑符相同[13], 也就是说我们又可以乱写了.

```
program main
```

[13] 同脚注12

```
    implicit none
    real :: one, zero
    real :: val1, val2, val3, val4
    one = 1.0
    zero = 0.0
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write (10, "(F9.12)") +(one/zero)
    write (10, "(E9.34)") -(one/zero)
    write (10, "(E9.5E6)") +(zero/zero)
    write (10, "(E9.78E0)") -(zero/zero)
    close(10)
    open(10, file='test.txt', status='OLD', &
        action='READ', position='REWIND')
    read(10, "(E9.87E0)") val1
    read(10, "(E9.6E5)") val2
    read(10, "(E9.43)") val3
    read(10, "(F9.21)") val4
    print *, val1, val2, val3, val4
    close(10)
end program main
```

E 编辑符蛮好用, 就是最后输出的结果不太符合俺们的习惯, 因为不是用科学计数法表示的. 我们只要把 E 换成 ES, 结果就是科学计数法表示的了, 也就是说 $1 \leqslant a < 10$. 我们还可以把 E 换成 EN, 这样的结果是用工程计数法表示的, $1 \leqslant a < 1000$ 且 n 能被 3 整除, 这样单位换算就会比较方便. 除了 a 和 n 不同外 E 编辑符, ES 编辑符, EN 编辑符没有区别.

```
program main
    implicit none
    print "(E7.1E1)", 10000.0
    print "(ES7.1E1)", 10000.0
    print "(EN7.1E1)", 10000.0
end program main
```

复型编辑符

复型编辑符是没有的, 输出复数的时候, 永远是把实部和虚部分别输出, 我们可以分别给实部和虚部加实型编辑符.

```
program main
    implicit none
    print "(F4.1,E8.1)", (0.1, 1.0)
end program main
```

输入也是一样的道理, 注意输入的时候实型编辑符是可以乱来的.

```
program main
    implicit none
    complex :: z
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(F4.1,E8.1)") (0.1, 1.0)
    close(10)
    open(10, file='test.txt', status='OLD', &
        action='READ', position='REWIND')
    read(10, "(E4.1,F8.1)") z
    print *, z
    close(10)
end program main
```

字符型编辑符

Aw 编辑符表示一共输出 w 个字符. 设字符串长度为 l, 如果 $l > w$, 则只输出字符串最左边 w 个字符, 如果 $l < w$, 则先输出 $w - l$ 个空格再输出 w 个字符[14]. A 编辑符则表示 $w = l$.

```
program main
    implicit none
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
```

[14]这里是左补空格, 字符串赋值 (5.1节) 是右补空格.

```
    write(10, "(A4)") 'hello'
    write(10, "(A5)") 'hello'
    write(10, "(A6)") 'hello'
    close(10)
    open(10, file='test.txt', status='OLD', &
         action='WRITE', position='APPEND')
    write(10, "(A)") 'hello'
    write(10, "(A)") 'hellohello'
    write(10, "(A)") 'hellohellohello'
    close(10)
end program main
```

输入的话情况比较复杂. 首先同志们要认定每一行最后都有无数个空格, 下面这个轮子, 写入完第一行是 1234567890, 读取完 c 当然等于 '12345', 第二行是 1, 后面没了, 同学们要认定 1 后面跟着无数个空格, 所以读取完 c 等于 '1 '

```
program main
    implicit none
    character(5) :: c
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(A)") '1234567890'
    write(10, "(A)") '1'
    close(10)
    open(10, file='test.txt', status='OLD', &
        action='READ', position='REWIND')
    read(10, "(A5)") c
    print "(2A)", c, '}'
    read(10, "(A5)") c
    print "(2A)", c, '}'
    close(10)
end program main
```

然后 A 编辑符表示 $w = l$ 这点不变.

```
program main
    implicit none
    character(4) :: sc
    character(6) :: lc
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(A)") '1234567890'
    write(10, "(A)") '1234567890'
    close(10)
    open(10, file='test.txt', status='OLD', &
        action='READ', position='REWIND')
    read(10, "(A)") sc ! (A4)
    print "(2A)", sc, '}'
    read(10, "(A)") lc ! (A6)
    print "(2A)", lc, '}'
    close(10)
end program main
```

Aw 编辑符则表示读取 w 个字符. 若 $l < w$, 则赋值最右边 l 个字符[15], 若 $l > w$, 则赋值 w 个字符后跟 $l - w$ 个空格.

```
program main
    implicit none
    character(4) :: sc
    character(5) :: nc
    character(6) :: lc
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(A)") '1234567890'
    write(10, "(A)") '1234567890'
    write(10, "(A)") '1234567890'
    close(10)
    open(10, file='test.txt', status='OLD', &
        action='READ', position='REWIND')
```

[15]这里是赋值最右边的字符, 字符串赋值 (5.1节) 是赋值最左边的字符.

```
    read(10, "(A5)") sc
    print "(2A)", sc, '}'
    read(10, "(A5)") nc
    print "(2A)", nc, '}'
    read(10, "(A5)") lc
    print "(2A)", lc, '}'
    close(10)
end program main
```

可见字符型编辑符十分让人头大, 如果老师敢考我们就揭竿而起...
注意字符型编辑符和 9.3.3 节的字符串编辑符是八竿子打不着的.

逻辑型编辑符

Lw 表示一共输出 w 个字符, 前面 $w-1$ 个是空格, 最后 1 个是 `T` 或 `F`, `T` 代表 `.true.`, `F` 代表 `.false.`.

```
program main
    implicit none
    print "(L7)", .true.
    print "(L7)", .false.
end program main
```

输入的时候, 字符 `T` 和 `F`, 字符串 `TRUE` 和 `FALSE`, 字符串 `.TRUE.` 和 `.FALSE.` 都代表 `.true.` 和 `.false.`, 而且不分大小写.

```
program main
    implicit none
    logical :: true, false
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(A)") 't f '
    write(10, "(A)") 'true false'
    write(10, "(A)") '.true. .false. '
    close(10)
    open(10, file='test.txt', status='OLD', &
        action='READ', position='REWIND')
```

```
    read(10, "(2L2)") true, false
    print "(2L2)", true, false
    read(10, "(2L5)") true, false
    print "(2L5)", true, false
    read(10, "(2L7)") true, false
    print "(2L7)", true, false
    close(10)
end program main
```

9.3.2 控制编辑符

nX 编辑符表示把 “文件定位” 右移 n 位, 这通常等价于输出 n 个空格, 但如果 nX 编辑符后没有数据编辑符或字符串编辑符, 则 nX 编辑符相当于没有.

```
program main
    implicit none
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(2(A,1X))") 'X', 'descriptor'
    close(10)
end program main
```

输入的时候 nX 编辑符则表示跳过 n 个字符不读.

```
program main
    implicit none
    character(5) :: world
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(A)") 'Hello, world!'
    close(10)
    open(10, file='test.txt', status='OLD', &
        action='READ', position='REWIND')
    read(10, "(7X,A5)") world
    print "(A)", world
```

```
    close(10)
end program main
```

/ 编辑符表示接下来从下一行第一个字符开始读取或写入.

```
program main
    implicit none
    character(5) :: hello, world
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "(A,/,A)") 'hello', 'world'
    close(10)
    open(10, file='test.txt', status='OLD', &
        action='READ', position='REWIND')
    read(10, "(A,/,A)") hello, world
    print "(A,/,A)", hello, world
    close(10)
end program main
```

SS 编辑符表示之后输出正数时最开头都不带正号, SP 编辑符则表示都带正号. [16]

```
program main
    implicit none
    print 1001, 0.1, 2.3, 4.5, 6.7, 8.9
    1001 format (SS,F5.1,F5.1,SP,F5.1,SS,F5.1,F5.1)
    print 1002, 0.1, 2.3, 4.5, 6.7, 8.9
    1002 format (SP,F5.1,F5.1,SS,F5.1,SP,F5.1,F5.1)
end program main
```

注意正号会占一个字符, 编译器有可能因此罢工.

```
program main
    implicit none
    print "(SS,F3.1)", 1.0 ! 1.0
    print "(SP,F3.1)", 1.0 ! ***
```

[16] “S” 代表 “sign”. “S” 代表 “suppress”, “P” 代表 “plus”.

```
end program main
```

输入的时候 SS 编辑符和 SP 编辑符不起作用, 我们又可以乱来了.

```
program main
    implicit none
    real :: val0, val1, val2, val3, val4
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, 1061) 0.1, 2.3, 4.5, 6.7, 8.9
    1061 format (SS,2F5.1,SP,F5.1,SS,2F5.1)
    write(10, 1062) 0.1, 2.3, 4.5, 6.7, 8.9
    1062 format (SP,2F5.1,SS,F5.1,SP,2F5.1)
    close(10)
    open(10, file='test.txt', status='OLD', &
         action='READ', position='REWIND')
    read(10, 1051) val0, val1, val2, val3, val4
    1051 format (SP,F5.1,SS,3F5.1,SP,F5.1)
    print *, val0, val1, val2, val3, val4
    read(10, 1052) val0, val1, val2, val3, val4
    1052 format (SS,F5.1,SP,3F5.1,SS,F5.1)
    print *, val0, val1, val2, val3, val4
    close(10)
end program main
```

在输出实数的时候, 我们只能输出若干位有效数字, 所以这里有个舍入的问题. 比如测量值我们希望四舍五入但不确定度我们希望向上舍入. 我们可以用 RU, RD, RZ, RC 编辑符, 这四个编辑符分别表示之后输出实数时都向上舍入, 都向下舍入, 都向零舍入, 都四舍五入. [17]

```
program main
    implicit none
    print "(RU,4F5.1,/,"// &
          "RD,4F5.1,/,"// &
          "RZ,4F5.1,/,"// &
```

[17]"R" 代表 "round". "U" 代表 "up", "D" 代表 "down", "Z" 代表 "zero", "C" 代表 "compatible".

```
          "RC,4F5.1)", &
        -5.6789, -0.1234, +0.1234, +5.6789, &
        -5.6789, -0.1234, +0.1234, +5.6789, &
        -5.6789, -0.1234, +0.1234, +5.6789, &
        -5.6789, -0.1234, +0.1234, +5.6789
end program main
```

输入时 RU, RD, RZ, RC 编辑符也起作用, 因为用字符串表示的实数都是形如 $\sum_{i=m_{\min}}^{m_{\max}} 10^i$ 的实数, 而电脑是二进制的, 读取后实数都必须是形如 $\sum_{i=n_{\min}}^{n_{\max}} 2^i$ 的实数, 所以也得舍入.

9.3.3 字符串编辑符

字符串编辑符就是一个字符串, 作用就是输出这个字符串.

```
program main
    implicit none
    print "(A,', world!')", 'Hello'
    print "('Hello, ',A,'!')", 'world'
end program main
```

读取和写入的时候, 我们其实可以不加任何数据实体, 所以可以秀波操作.

```
program main
    implicit none
    print "('Hello, world!')",
end program main
```

输入的时候不能用字符串编辑符, 我们可以用 nX 编辑符代替.

```
program main
    implicit none
    character :: the_end
    open(10, file='test.txt', status='REPLACE', &
         action='WRITE', position='APPEND')
    write(10, "('Hello, world',A1)") '!'
    close(10)
    open(10, file='test.txt', status='OLD', &
```

```
          action='READ', position='REWIND')
    read(10, "(12X,A1)") the_end
    print *, the_end
    close(10)
end program main
```

附录 A　绘图

Fortran 一直没有标准的绘图轮子, 毕竟 Fortran 刚出世的那个年代, 电脑破烂到图都不知道哪里存. Fortran 官网上挂着一些第三方轮子, 但这些轮子安安都不太满意.

安安认为, Fortran 它的强项本来就是计算而不是绘图, 绘图完全可以交给其他语言的好轮子来干. 首选的轮子当然是天文人必用的 Python 的 Matplotlib, 那么最好有一个能简单地把 Fortran 算出的数据存起来, 再用 Python 读出来的轮子. 于是安安造了一个轮子 fdata2pyplot 挂在 Github 上. 有了它 Fortran 绘图就很方便啦.