Fortran 笔记

GasinAn

2024 年 1 月 17 日

Printed in China

目录

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!

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计算也是完全使用 Modern Fortran, 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[1], 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.[2]

- 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，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.

- 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[3]允许快速编写贴近硬件运行的代码.

[1] 太惨了, 都是考点啊!

[2] 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...

[3] 刚学 Fortran 的天文系本科小盆友们将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!...

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, 但还是留下一些恐怕是实在不好填的坑 (比如大小写不分之类的).
- 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[4], 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语言了.[5]

目前 Fortran 有自己的官方网站, 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 Fortran 2018 标准文档及其补充规定[6].

[4] 在 Fortran 官方的不懈努力下, 根据最新的 TIOBE 排行, Fortran 已经比 Matlab 更流行啦!

[5] 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在同数量级, 差不多.

[6] 这两篇文档虽然绝对正确但都超级无敌巨 TM 难读, 同学们还是别碰了, 可直接读高级 Fortran 笔记 (开发中).

第二章　编译器

编译器	Ifort (+VS)	Gfortran (+VS Code)
类型	专有软件	自由软件
总空间占用	约 5.2G	约 780M
语法支持	略多于 Gfortran, 较宽松	略少于 Ifort, 较严格
自动纠错和自动补全	无	有
编译提示信息	次于 Gfortran	优于 Ifort
平均运行速度	快于 Gfortran	慢于 Ifort

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

2.1　Intel® Fortran Compiler (Ifort)

2.1.1　安装

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 使用

使用 CLI

Windows 系统中, 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 (或 Intel oneAPI command prompt for IA32 for Visual Studio 2022). 然后可以使用 cmd 的命令 (比如 `cd` 之类的).

示例 1 新建一个名为 `main.f90` 的文件, 写入下面的内容并保存.

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

然后 `cd` 到 `main.f90` 所在的目录后执行下面这条命令[1].

```
ifort main.f90
```

就把 `main.f90` 编译好啦, 生成了文件 `main.obj` 和 `main.exe`.

执行下面这条命令.

```
main.exe
```

“`Hello, world!`” 被打出来啦!

示例 2 删掉 `main.obj` 和 `main.exe`, 在 `main.f90` 所在目录再新建一个名为
`helloworld.f90` 的文件, 写入下面的内容并保存.

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

[1]这里 `ifort` 对应的其实是 Intel© Fortran Compiler Classic, 想用真 Intel© Fortran Compiler, 请换成 `ifx`. 本书所有示例程序都是用 `ifort` 测试的, 懒得改了, 大家凑合看吧!

把 main.f90 修改成下面这样.

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

执行下面这条命令.

```
ifort main.f90 helloworld.f90
```

生成了 main.obj, helloworld.obj 和 main.exe.

然后再次执行下面的命令.

```
main.exe
```

“Hello, world!” 被打出来啦!

示例 3 以后搞一些大事情时, ifort 后要跟一大堆文件.

一个个打文件名显然要死人. 这时可以使用通配符.

删掉 main.obj, helloworld.obj 和 main.exe. 执行命令.

```
ifort *.f90
```

就相当于 ifort 后跟了当前目录所有最后是.f90 的文件的文件名. 但注意, 这回生成的不是 main.exe, 而是 helloworld.exe, 字母表中 h 可是排在 m 前面呢.

执行命令.

```
helloworld.exe
```

“Hello, world!” 被打出来啦!

示例 4 文件太多时, 使用通配符后可能不清楚最后.exe 文件的文件名是什么. 这时可以干脆直接指定文件名.

删掉 main.obj, helloworld.obj 和 helloworld.exe. 执行命令.

```
ifort *.f90 -o a
```

这回生成的不是 helloworld.exe, 而是 a.exe.

执行命令.

```
a.exe
```

“Hello, world!” 被打出来啦!

示例 5 删掉 main.obj, helloworld.obj 和 a.exe. 执行命令.

```
ifort -c helloworld.f90
```

这回只生成了文件 helloworld.obj.

执行命令.

```
ifort main.f90 helloworld.obj
```

生成了 main.obj 和 main.exe.

执行命令.

```
main.exe
```

“Hello, world!” 被打出来啦!

Ifort 的工作流程是, 先把每个源文件都变成对应的.obj 文件, 再把它们合起来后变成.exe 文件. 遇到一些一万年也不会改, 而且会反复使用的文件 (比如轮子[2]里的文件), 我们就可以先把它们统统变成.obj 文件, 然后反复用, 以节省时间.

使用 GUI

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 HelloWorld 文件夹.

默认使用的是 Ifort. 欲用 Ifx, 右键右边 “解决方案资源管理器” 里的 “Console1 (IFORT)”, 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 “IFORT Intel® Fortran Compiler Classic”, 点最右边的带向下标志的按钮, 改选成 ‘IFX Intel® Fortran

[2]这里玩了个 “重新制造轮子” 的梗.

Compiler", 点 "应用", 点 "确定", 看到右边 "Console1 (IFORT)" 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)".

"`Hello, world!`" 被打出来啦!

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.

2.2 GNU Fortran Compiler (Gfortran)

2.2.1 安装

Windows 系统安装方式如下.[3]

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`. 举个例子, 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.

[3]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...

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[4]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":
    "[cmd]",
    "fortran_fixed-form":
    "[cmd]"
},
"code-runner.runInTerminal": true,
"code-runner.saveAllFilesBeforeRun": true
```

最后一步其实和在 VS Code 的设置里点点点是一样滴, 我其实就是在设置里凭感觉点点点后把 `settings.json` 里的东东弄出来了, 如果玩代码的时候感觉不舒服, 自己解决, 我可不管. 注意

- `[cmd]` 要替换成编译并执行程序的整套命令. 可以参考2.2.2小节, 按自己的需求替换, 也可以戳这里搬运现成的.
- `[dir]` 要替换成上文提到的 `[dir]`.

还有加入键值对的时候要符合 JSON 的规则. 举两个例子. 如果一开始 `settings.json` 里是空的, 加入键值对后可能长这样.

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

[4]有 conda 当然用 conda 啦~

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

如果一开始 `settings.json` 长这样,

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

加入键值对后就可能长这样.

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

注意倒二行最后多了一个逗号. 键值对间要用逗号隔开.

2.2.2 使用

使用 CLI

Gfortran 的用法和2.1.2小节中 Ifort 的用法非常像, 不同点是:

- 使用 `gfortran` 命令代替 `ifort` 命令;

- 中间过程生成的是`.o` 文件, 不是`.obj` 文件;
- 默认生成文件 `a.exe`, 且不会生成`.o` 文件.

直接打开命令行 (cmd 或 powershell), 即可使用 `gfortran` 命令.

在 VS Code 中, 可以直接点上面 “Terminal” 后点 “New Terminal” 来调出命令行. 默认调出的是 powershell. 若想让默认调出的命令行是 cmd, 可在 `settings.json` 里加入下面这个键值对.

```
"terminal.integrated.shell.windows":
"C:\\Windows\\System32\\cmd.exe"
```

使用 GUI

点击右上方的白色三角儿即可编译并执行! 除非之前 `[cmd]` 设错啦~

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. 这些指令是要用一堆字符表示的, 表示指令的字符的整体称为源代码 (source code). 源代码需要被保存在文件中, 保存源代码的文件称为源代码文件 (source code file)[1].

比如, 在一个名为 `helloworld.f90` 的文件内写入下面这些内容并保存.

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

这个 `helloworld.f90` 文件就是一个源代码文件, 里面的四行字符就是一份源代码, 这份源代码代表四条指令, 这四条指令组合起来就是一个程序.

3.2　字符集

Fortran 可以在程序中使用键盘能直接打出的所有字符. 其他字符 (比如汉字, 汉语标点) 等能不能出现在程序中, 要看编译器认不认, 也就是能不能执行 (见3.4节). 注意, 英文标点和中文标点是不同的!

经测试, Ifort 和 Gfortran 都是可以支持汉字和汉语标点的, 但直接把汉字或汉语标点打在屏幕上可能会出现乱码 (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.

规范 1 *不要在程序中使用键盘不能直接打出的字符.*

[1] 显然, 程序, 源代码和源代码文件是高度相关的, 所以今后将不再区分这三者.

Fortran 是大小写不敏感的 (case-insensitive), 也就是说, 很多情况下程序中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). 比如, 下面这个程序和3.1节中的程序是完全等价的.

```
PROGRAM MAIN
    IMPLICIT NONE
    PRINT *, 'Hello, world!'
END PROGRAM MAIN
```

相信我, 在程序中混用大小写字母是很容易出事的.

规范 2 *尽量在程序中使用小写字母.*

3.3 源代码格式

Fortran 的源代码格式 (source form) 分两种: 自由格式 (free form) 和固定格式 (fixed form). 自由格式的源代码需要保存在拓展名为`.f90` 的文件中, 可能可以保存在拓展名为`.F90`, `.f03`, `.F03`, `.f08`, `.F08`, `.f18`, `.F18`, `.fypp`, `.FYPP` 的文件中. 固定格式的源代码需要保存在拓展名为`.for` 或`.f` 的文件中, 可能可以保存在拓展名为`.FOR`, `.F`, `.ftn`, `.FTN`, `.fpp`, `.FPP` 的文件中. 上述 "可能" 要看编译器承认不承认, 也就是能不能执行 (见3.4节).

写固定格式的程序会遇到各种各样的问题. 自由格式就是为了替代固定格式而创立的. 本书 Fortran 部分的全部内容都基于自由格式程序.

规范 3 *永远编写自由格式的程序.*

源代码文件名除拓展名外可以随便取, 但乱取文件名是要粗事情滴.

规范 4 *将源代码文件名取成能表示程序功能的字符串.*

规范 5 *不要在源代码文件名中使用汉字等特殊字符.*

3.3.1 缩进

缩进 (indent) 是指源代码每行最前面的空格 (可能没有). 对 Fortran 而言, 缩进本身通常没有意义, 但适当的缩进能使程序更容易被理解.

规范 6 *在程序中加入适当的缩进, 最好以 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. 合格的文本编辑器 (text editor), 比如 VS Code, 会自动在程序中加上合适的缩进. 有些人可能更喜欢以 2 个空格或 8 个空格为单位的缩进.

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, 会自动把 [Tab] 转换为适当数目的空格, 但还是不保险. 一些经典的优秀编辑器, 比如 Vim, 好像是会严格区分 [Tab] 和空格的[2], 除非另装外挂. 一些编译器可能会接受在程序编辑中使用 [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 7 *不要使用 [Tab] 键.*

3.3.2 空行

为便于理解程序, 还要在程序中加入适当的空行 (就是没有任何字符的一行[3]). 至于什么时候加入一行或加入几行, 完全看心情. 简单的原则就是, 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.

规范 8 *在程序中加入适当的空行.*

正如3.1节所说, 程序是指令的集合. 空行代表的指令是 “啥事也不做”.

3.3.3 注释

程序每一行中, ! 及后面的所有内容, 通常都会被无视 (不会被无视的地方见4.4节), 也就相当于! 后面的内容不存在. 会被无视的! 及后面的内容称为注释 (comment), 只包含注释的行称为注释行 (comment line).

注释写什么无所谓, 但这么强大的东东还是要充分利用的. 普通的小程序, 可以在程序中复杂深奥的部分加上注释, 写上这部分是怎么运作的, 干了什么事, 得到什么结果等. 只要自己看程序的一部分, 觉得头大, 就加上注释吧. 优秀的轮子里都有非常完善的注释, 包括程序每部分的功能, 作者姓名, 机构, 修改时间, 和版权声明等, 注释比程序本身还长是常有的事.

[2] 这并不见得是坏事, 因为有些特殊文件是默认必须使用 [Tab] 的.

[3] 只有空格的行也是空行, 但这样的行除了多占用存储空间外, 和没有任何字符的行没区别.

自己的小程序, 写写汉语注释也无所谓, 不过最好还是写英文的 (顺便提升英语能力). 造轮子请加上纯英文的注释.

规范 9 *在程序中加入适当的注释.*

3.3.4 续行

有时可能遇到一行太长的情况, 这时可以使用 `&` 来把一行拆成两行写. 下面这个程序就用了续行 (continuation).

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

上面这个程序就相当于下面这个程序.

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

如果用一个续行符 `&` 续行后, 一行还是太长, 可以接着用续行符再续. 比如下面这个程序和上面两个程序是一样的.

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

其他使用续行符的规则如下.

- 不能简单地在不能插入空格的地方续行[4].
  比如, 下面这个程序是不成的.

[4] 也就是说用特殊方法可以, 但这样的话这个程序就会让人看得脑壳疼.

```
program main
    im&
plicit none
    print *, 'Looooooooooooooooooooooooooooong!'
end program main
```

- 可能不能让一行之中只有空格和一个续行符.
  比如, 下面这个程序可能是不成的

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

- 续行符下一行不能是空行, 也不能是注释行.
  比如, 下面这个程序是不成的.

```
program main
    implicit none
    print *, 'Looooooooooooooooooooooooooooong!' &
    ! This is invalid!
end program main
```

- 没法在注释中使用续行符, 因为注释中的 `&` 会被当成是注释的一部分.
  比如, 下面这个程序是不成的.

```
program main
    implicit none
    print *, 'Looooooooooooooooooooooooooooong!'
    ! This is &
      invalid!
end program main
```

  在 VS code 这样合格的编辑器中, 上面这个程序的第四行和第五行颜色会不一样, 并且第四行中 `&` 的颜色会和第四行其他字符的颜色一样. 这就提示我们 `&` 是注释的一部分, 并且第五行不被当成是注释.

- 续行符后的所有空格会被无视, 并且可以在续行符后加注释.
  比如, 下面这个程序是成的.

```
program main
    implicit none
    print *, & ! This is valid!
    'Looooooooooooooooooooooooooooooong!'
end program main
```

  总的来说就是, 老老实实用续行符是可以的, 搞幺蛾子是不行的.
  如果程序中某些行非常非常长[5], 那这程序读起来真是头疼.

规范 10 *程序每一行应不超过* 80 *个字符.*

规范 11 *在合适的地方使用续行符, 并保证续行符前有一个空格.*

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[6]. 加入这两个键值对只是起到提醒作用, 太长还是要分行写的.

3.3.5 标号

除了空行, 注释行和续行符的后一行, 程序的每一行开头都可以加上标号 (label). 标号是 1–99999 的数字, 后面至少有一个空格. 标号开头可以是 0, 但标号中的数字不能多于五个. 四位数的, 整千或整百的标号比较常见.

下面的程序第三行使用了标号.

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

[5] 按照标准, 一行最多能有 132 个字符.
[6] 可以看到下面一 “行” 是没有行号的.

在程序中显然不能有重复的标号[7], 比如下面这个程序是不成的.

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

3.3.6 标签

在一些特殊的地方 (见6.2.3和6.2.4小节) 可以加标签 (tag).

下面的程序第四行使用了标签.

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

显然在程序中也不能有重复的标签[8].

3.4 运行

Fortran 程序由若干程序单元 (program unit) 组成, 程序单元可能是主程序 (main program), 或外部子程序 (external subprogram, 见第八章), 或模块 (module, 见第**??**章), 或子模块 (submodule, 见第**??**章). 一个 Fortran 程序中必须有且只有一个主程序, 其他程序单元数目任意.

主程序必须满足下面的格式[9].

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

[7] 准确来说是在一个程序单元内不能有重复标号. 程序单元的含义见3.4节.

[8] 同标号一样, 准确来说是在一个程序单元内不能有重复标签.

[9] 其实头尾还可能可以省略一部分, 但还是写全为好.

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 "_"[10]组成的, 开头是字母的字符串. "..." 处则是各种指令, 一行一条.

2.1.2小节和2.2.2小节中已经介绍了程序运行/执行 (run/execute) 的部分方法 (我觉着掌握这些就够了). 所谓运行就是让电脑按程序中的指令干活. 程序运行的规则是:

1. 首先运行 `program [program-name]`.
2. 运行一行指令后, 紧接着运行下一行指令, 除非先前运行的指令中有指明接下来应该运行哪一行指令 (见第六章, 第八章, 第**??**章, 第**??**章).
3. 一旦运行 `end program [program-name]`, 就不再运行任何指令.

3.5 异常

程序的运行过程可未必一帆风顺, 可能出现异常 (exception). 异常分为错误 (error) 和警告 (warning).

3.5.1 错误

电脑运行程序, 运行着运行着, 可能会突然发现自己没法儿干活了, 这时电脑就只好告诉我们, 它莫得干 (这就是抛出错误), 然后直接罢工了.

当然如果老是这样, 俺们和电脑们都会浑身难受, 因为在碰上错误前, 电脑还执行了其他指令, 这可能会白费相当多的时间. 所以在生成`.exe` 文件前, 电脑会先把整份代码读一遍, 瞅一瞅看是不是有地方电脑是不能干活的, 如果有, 电脑会提前抛出错误, 这样就不会白费太多时间了. 当然有时候, 电脑在事先没有办法判断某些个地方是没法干活的, 那它就只能等真执行到那里时再抛出错误了.

比如, 运行这样一个程序 (文件名 `main.f90`).

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

[10]下划线是空格的替代字符, 长得还是蛮像滴!

Ifort 给出的结果如下[11].

```
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规则 (`print` 后少了 `*,`), 电脑不知怎么干活了, 只好报个错儿. 这个错儿是在生成`.exe` 文件时就查出了.

再比如, (Windows 下) 运行这样一个程序.

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

这回生成`.exe` 文件时, Ifort 和 Gfortran 都表现得很正常, 然后就出事. 虽然所有命令都是符合规则的, 然而 Windows 文件名里不能有 `*`, 电脑没法干活, 只好报个错儿.

一个程序会不会报错儿, 其实还和编译器有关. 比如运行下面这个程序.

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

[11]假定屏幕每行仅能容纳 60 字符, 且略去版权声明等无关紧要的内容, 下同.

(默认情形下) Gfortran 会报错儿说不能除以 0, 但 Ifort 就不会报错, 会给个无穷大. 有趣的是, 如果运行下面这个程序, 则 Gfortran 和 Ifort 一样, 都会算出个无穷大来.

```
program main
    implicit none
    real :: r
    r = 0.0
    print *, 1.0/r
end program main
```

3.5.2 警告

有些时候电脑能够执行指令, 但它觉得这些个指令十分 “危险”, 很有可能指令的实际执行过程并不是我们预想的那样, 这时电脑就会抛出警告. 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, 那就全看编译器心情了.

比如, 运行这样一个程序 (文件名 `main.f90`).

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

Ifort 给出这样的结果.

```
main.f90(3): warning #6384: The INTEGER(KIND=4) value is out
-of-range.   [1000000000000]
    integer :: i = 1000000000000
--------------------^
```

Ifort 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. 运行后输出`-727379968`, 果然很奇怪! 至于 Gfortran 嘛, 它也认为 `1000000000000` 过大, 但它不是给出警告, 而是直接报错.

再比如, 运行这样一个程序 (文件名 `main.f90`).

```
program main
```

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

Gfortran 给出这样的结果.

```
main.f90:3:18:

    integer :: 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” 的操作其实是标准规则不允许的, 是自己放水才能这么干的, 于是就给个警告, 但程序仍然可以运行. 运行后输出 `1`. 至于 Ifort 嘛, 它倒是不报错也不给警告, 干脆利落地运行了, 但运行后输出的是`-1`…

第四章　固有类型

变量 (variable) 相当于容器, 可以存放电脑计算时使用的数据, 变量中存储的数据在计算过程中可以改变. 常量 (constant) 相当于存放着固定数据的容器, 分为字面常量 (literal constant) 和具名常量 (named constant, 见8.2.1节), 常量中存储的数据在计算过程中不能被改变. 变量和常量都是数据实体 (data entity).

在 Fortran 中, 每一个数据实体都有一个明确的类型 (type). 类型分为固有类型 (intrinsic type) 和派生类型 (derived type). 所有固有类型的数据实体都有种别 (kind).

Fortran 是静态类型语言 (statically typed language), 也就是说, Fortran 中的变量不能在一个命令执行之后转变类型. Fortran 中要使用一个变量, 必先声明 (specify), 声明后变量的类型就确定了, 以后不能更改.

之前的所有程序都含有个 `implicit none`, 这个命令能阻止很多变量声明时的坑人规则发挥作用, 血泪教训就是一定要把它加上…

规范 12 *永远在程序中加入 `implicit none`.*

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). 使用这五个类型的变量前, 一定要声明变量的具体类型. 种别可以声明也可以不声明, 如果不声明, 会取一个默认的种别.

4.1　整型

整型字面常量没啥好讲, 说白了就和整数一毛一样. 下面这个程序使用了整型字面常量`-1234567890`.

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

用 integer 关键词 (keyword) 可以声明整型变量, 像下面这样.

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

上面这个程序声明了个名称为 i 的默认种别的整型变量.

我们还可以一口气声明多个同类型同种别的变量, 像下面这样.

```
program main
    implicit none
    integer :: i, j, k, l, m, n
end program main
```

上面这个程序一口气声明了六个默认种别的整型变量.

64 位电脑上, Ifort 和 Gfortran 都支持四个整型种别: int8, int16, int32 和 int64. 指明整型字面常量的种别, 在其后面用下划线连上种别名就成. 但种别名不能直接用, 出现在 use, intrinsic :: iso_fortran_env 后就可以用了, 像下面这样. 注意 use ... 要摆在 implicit none 上面.

```
program main
    use, intrinsic :: iso_fortran_env, &
        only: int8, int16, int32, int64
    implicit none
    print *, 127_int8
    print *, 32767_int16
    print *, 2147483647_int32
    print *, 9223372036854775807_int64
end program main
```

Ifort 和 Gfortran 默认的整型种别都是 int32.

指明整型变量的种别, 在 integer 后加括号, 然后填上种别名.

```
program main
    use, intrinsic :: iso_fortran_env, &
        only: int8, int16, int32, int64
    implicit none
    integer(int8) :: i8
    integer(int16) :: i16
    integer(int32) :: i32
    integer(int64) :: i64
end program main
```

我们还可以给种别取个别名, 像这样.

```
program main
    use, intrinsic :: iso_fortran_env, &
        only: s=>int16, i=>int32, l=>int64
    implicit none
    integer(s) :: short
    integer(i) :: int
    integer(l) :: long
    print *, 32767_s
    print *, 2147483647_i
    print *, 9223372036854775807_l
end program main
```

整型种别名最后的数字实际上表示数字在电脑中用几个单位空间来存储. 设存储的单位空间数为 n, Ifort 能存储 -2^{n-1} 到 $2^{n-1}-1$ 的整数, Gfortran 能存储 $-2^{n-1}+1$ 到 $2^{n-1}-1$ 的整数. 如果选择的种别 "太小", 空间不足却强令电脑存储数字, Ifort 会给警告然后算出个奇奇怪怪的结果, Gfortran 会直接报错. 后面实型和复型也是如此.

选择什么整型种别, 看需要, 一般选默认的就成.

4.2 实型

实型字面常量也没啥好讲, 说白了也就和实数一毛一样. 不过这里的实数可以有一个以 10 为底的指数部分, 用字母 `e` 后加整数来表示. 下面这个程序使用了实型字面常量 `6.62607015e-34`, 其相当于 6.62607015×10^{-34}.

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

运行程序后 Terminal 出现的其实只是个近似 6.62607015e-34 的值, 这是由于计算机实际上不总能绝对精确地存储实型数据实体. 多数情况下这样的不精确不妨事, 但有时会大事!

用 `real` 关键词可以声明实型变量, 像下面这样.

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

上面这个程序声明了个变量名为 `h` 的默认种别的实型变量.

我们还可以一口气声明多个同类型同种别的变量, 方法同整型变量.

64 位电脑上, Ifort 和 Gfortran 都支持三种实型种别: `real32`, `real64` 和
`real128`, 分别称为单精度, 双精度和四精度. 指明实型字面常量或变量的种别的方式, 还有给实型种别取别名的方式, 都和整型一样.

同样, 实型种别名最后的数字实际上表示数字在电脑中用几个单位空间来存储. 原则上存储的单位空间数越多, 实型量有效位数就越多, 并且可取得的范围也越广, 也就是说既能取超级大的数, 也能取超级小的数.

只要不是在用古董电脑, 统统用 `real128` 就好了. 统统用 `real128`, 很多坑都能不踩到了, 这是好事, 然而程序的运行时间必然蹭蹭上涨, 而且一般计算并不需要那么高的精度[1]. 天文中很多经典轮子都是运行在 `real64` 这样的精度的, 所以统统用 `real64` 一般也是可的.

4.3 复型

复型真的好不常用... 如果是搞物理的, 还可能用用.

复型字面常量长成 `(..., ...)` 的样子, 其中`...` 是整型或实型常量, 逗号前后分别是实部和虚部.

[1] 众所周知, 俺们天文观测数据的不确定度是相当大滴.

下面这个程序使用了复型字面常量 `(3.0, 4.0)`, 其相当于 $3+4i$.

```
program main
    implicit none
    print *, (3.0, 4.0)
end program main
```

构造复型字面常量时, 是可以在实部和虚部用整型字面常量, 但构造完的复型字面常量的实部和虚部都会自动变成相应的实型字面常量, 比如把上面的 `(3.0, 4.0)` 改成 `(3, 4)`, 结果是一样的.

- 如果构造时实部和虚部都用实型字面常量, 则实部和虚部的种别是两个实型字面常量的种别中较精确的一个.
- 如果构造时实部和虚部一个用实型字面常量, 另一个用整型字面常量, 则实部和虚部的种别是实型字面常量的种别.
- 如果构造时实部和虚部都用整型字面常量, 则实部和虚部的种别是默认的实型种别, 也就是 `0.0` 的种别.

用 `complex` 关键词可以声明复型变量. 复型的种别与实型相同, 复型数据实体的种别与实部和虚部的种别相同. 声明复型变量方法, 标明复型变量种别的方法同实型和整型. 但要注意, 给复型常量加尾巴是不成的. 要标明复型常量的种别, 给实部和虚部分别加上尾巴就成.

4.4 字符型

字符串用`'...'` 或`"..."` 表示, 其中`...` 是字符串的内容, 可以没有.

下面这个程序使用了字符型字面常量`'!@#$%^&*'`, 代表'!@#$%^&*', 还有字符型常量`''`, 代表一个字符也无的 "空字符串".

```
program main
    implicit none
    print *, '!@#$%^&*', '', '!@#$%^&*'
end program main
```

注意引号内的所有内容都是字符串的一部分, 包括空格, 并且字符串中的字母是区分大小写的. 下举一例.

```
program main
    implicit none
    print *, '.', 'HELLO', 'HELLO', 'HELLO', '.'
    print *, '.', ' hello', '  hello   ', 'hello    ', '.'
end program main
```

自然而然出现的问题就是, 如何在字符串中加' 和". 自然可以在用' 括起来的字符串里用", 在用" 括起来的字符串里用'. 如果非得在用'/" 的字符串里用'/", 则可以用两个连续的'/" 来表示'/", 像下面这样.

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

```
program main2
    implicit none
    print *, 'GasinAn said, "I love using ''wheels'',' 
    print *, "especially the 'liber' ones."""
end program main2
```

用 character 关键词可以声明字符型变量. 注意, 需要告诉电脑字符型变量所装的字符串的长度, 在 character 后紧跟括号, 内加整型常量来表示.

下面这个程序声明了个非常非常长的字符型变量.

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

要注意这么写是不成的, 因为 1e12 实际上是实型的.

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

但这么写是可的. 若不知为何, 可暂且不管这个例子.

```
program main
    implicit none
    character(10**12) :: very_long_char
end program main
```

有些时候非常尴尬, 程序运行前我们并不知道字符型变量的长度应该取多少, 这时我们可以在声明中加上`,allocatable`, 并将长度设成一个冒号":", 造一个延迟长度字符型变量 (deferred-length character variable). 随后我们可以用 allocate 语句 (statement) 将字符型变量的长度变成一个确定值. 接着我们还可以用 deallocate 语句将字符型变量的长度重新变成不确定的, 然后再对字符型变量使用 allocate 语句...

使用延迟长度字符型变量的示例如下.

```
program main
    implicit none
    character(:),allocatable :: deferred_len_char
    ! Now length of deferred_len_char is undefined.
    allocate(character(100000000) :: deferred_len_char)
    ! Now length of deferred_len_char is 100000000.
    deallocate(deferred_len_char)
    ! Now length of deferred_len_char is undefined again.
    allocate(character(0) :: deferred_len_char)
    ! Now length of deferred_len_char is 0.
end program main
```

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, 将字符串的第 `nl` 到第 `nu` 个字符截下来, 得到一个新字符串. `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` 就等于字符串的长度.

看下面这个例子就懂了.

```
program main
    implicit none
    character(9) :: numbers
    numbers = '123456789'
    print *, numbers(3:7)
    print *, numbers(:7)
    print *, numbers(3:)
    print *, numbers(:)
end program main
```

4.5 逻辑型

逻辑型字面常量只有两个: `.true.` 和`.false.`, 分别对应于逻辑中的“真” 和 “假”. 声明逻辑型变量用 `logical` 关键词.

第五章　赋值与运算

5.1　赋值

赋值[1](assignment) 是将一个数据实体的值拷贝给一个变量的操作.

用形如 `a = b` 的语句即可将 `b` 中的内容拷贝给 `a`. 请看下面的例子.

```
program main

    implicit none
    integer :: a, b

    ! Assign 1 to a.
    a = 1
    ! Now "print *, a" will return 1.

    ! Assign a to b.
    b = a
    ! Now "print *, b" will return 1.

end program main
```

Fortran 是强[2]类型语言 (strongly typed language), 这意味着 Fortran 在赋值时 “不允许” 类型转化, 数字型数据实体只能赋给数字型变量, 字符型数据实体只能赋给字符型变量, 逻辑型数据实体只能赋给逻辑型变量.

[1]实际上按标准来说, 赋值有两种: 固有赋值 (intrinsic assignment) 和超载赋值 (defined assignment). 本章 “赋值” 仅指固有赋值.

[2]事实上 “强” 和 “弱” 的定义是模糊的, 不过一般都将 Fortran 归为强类型语言.

官方规则是这么说, 实际却不完全是这样. Ifort 和 Gfortran 在不同程度上是允许数字型数据实体和逻辑型数据实体相互赋值的, 赋值时遵循的规则还不太一样. 我懒得总结 Ifort 和 Gfortran 在这方面的 “器规” 了, 同志们自己试试, 自己总结.

规范 13 *即使编译器允许, 也不要跨类型赋值.*

赋值的时候, “=” 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. Fortran 是静态类型语言, 所以转化的总则必然是: 把 “=” 右边的数据实体转化成类型和种别与 “=” 左边变量相同的数据实体, 然后赋值.

任何一个 Fortran 数字型数据实体都对应于一个数. 要细说来, 数字型数据实体赋值时的转化规则是非常复杂的, 我也说不清楚. 但基本原则是确定的, 就是 “尽量” 让转化前后的数据实体对应的数接近. 如果是 “精度低” 的数据实体转化成 “精度高” 的数据实体, 只要能转化, 那么转化前后的数据实体对应的数就是相等的. 比如整型转实型[3], 实型转复型, 低精度种别转高精度种别, 都是这样, 这时我们就可以放心大胆地赋值, 反正赋值后数据实体的实际意义不变.

如果是 “精度高” 的数据实体转化成 “精度低” 的数据实体, 事情就比较复杂.

如果是实型转整型, 我也不清楚标准规则怎么说, 但 Ifort 和 Gfortran 都是 “向 0 取整”. 可以运行下面这个程序来感受一下.

```
program main
    implicit none
    integer :: i
    i = +9.87654321
    print *, i
    i = -9.87654321
    print *, i
end program main
```

如果是复型转实型, 自然是取实部.

如果是复型转整型, 则是先转实型, 再转整型.

[3] 一般情形下整型转实型都是 “无损” 的, 但有些极端情形不是这样. 幸好这些极端情形是很难碰上的.

还有很多具体情况 (比如同类型 “高精度” 种别向 “低精度” 种别转化) 的规则我也不清楚, 但实型和复型数据实体都是比较精确的, “=” 左右的数据实体, 对应的数非常接近, 一般其中的区别可以不用管.

总而言之, 数字型数据实体赋值时的转化规则是非常复杂, 但一般情况下可以不管这些规则. 原因有以下三点:

- 除了复型转实型, 复型转整型, 和实型转整型, 转换前后有较大的区别外, 其他情况下转换前后的区别通常可以忽略;
- 复型转实型, 复型转整型, 和实型转整型的规则是比较明确的;
- (最重要的一点) 通常在计算时都是使用同种别的实型 (和复型), 基本没有类型转化的时候 (这时候就根本不需要去记这些个复杂的规则啦).

统一使用同种别的实型 (和复型), 能防止很多意料之外的情况出现.

规范 14 *尽量使用同种别的实型 (和复型) 常量和变量.*

字符串赋值时还有个长度匹配的问题. 如果 “=” 左边字符串的长度小于右边字符串的长度, 则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, 则在右边字符串尾巴处补上空格让长度一样, 然后赋值.

```
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
```

我们还可以给字符串的切片赋值, 这样只会修改切片的那部分.

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

5.2 运算

运算[4](operation) 是由一个或两个数据实体得到另一个数据实体的操作, 其中很多是与数学中的运算是对应的. 运算是要使用运算符 (operator) 的, 运算符分四类: 算数运算符 (numeric operator), 字符运算符 (character operator), 逻辑运算符 (logical operator), 关系运算符 (relational operator).

如果一个运算符将一个数据实体变为另一个数据实体, 则其一定要按 `[op][q]` 的方式使用, 其中 `[op]` 是运算符, `[q]` 是一个数据实体. 如果一个运算符将两个数据实体变为另一个数据实体, 则其一定要按 `[q1][op][q2]` 的方式使用, 其中 `[op]` 是运算符, `[q1]` 和 `[q2]` 是两个数据实体.

形如 `[op][q]` 或 `[q1][op][q2]` 的东东称为表达式 (expression), 表达式本身也是数据实体, 这就形成一个套娃定义.

形如 `[q1][op12][q2][op23][q3]` 的东东, 按套娃定义, 这也是一个表达式. 问题是这个表达式有两种解释.

1. 先把 `[q1][op12][q2]` 算出来, 得到一个数据实体 `[q12]`,
   则 `[q1][op12][q2][op23][q3]` 等同于 `[q12][op23][q3]`.

2. 先把 `[q2][op23][q3]` 算出来, 得到一个数据实体 `[q23]`,
   则 `[q1][op12][q2][op23][q3]` 等同于 `[q1][op12][q23]`.

按两种解释方式分别计算, 最后得到的结果可能不同, 怎么办? 数学中的运算符可能有优先级, 比如乘除优先于加减. Fortran 中的任何一个运算符都有优先级, 对应地, Fortran 中的运算有两个规则:

[4]实际上按标准来说, 运算有两种: 固有运算 (intrinsic operation) 和超载运算 (defined operation). 本章 "运算" 仅指固有运算.

1. 若 [op12] 的优先级不低于 [op23],
   则对于表达式 [q1][op12][q2][op23][q3], 先计算 [q1][op12][q2].[5]

2. 若 [op12] 的优先级低于 [op23],
   则对于表达式 [q1][op12][q2][op23][q3], 先计算 [q2][op23][q3].

这和数学中的运算是完全对应的.

可以加 (), 来强制让 () 里的部分先算, 这和数学也是对应的. 需要注意的是, 不能用 [] 和, 统统用 () 代替.

5.2.1 算数运算

算数运算符一共五个: +, -, *, /, **, 分别对应于加, 减, 乘, 除, 乘方.

```
program main
    implicit none
    real :: a, b
    a = 3.0
    b = 4.0
    print *, a+b ! Add 3.0 and 4.0.
    print *, a-b ! Subtract 3.0 from 4.0.
    print *, a*b ! Multiply 3.0 by 4.0.
    print *, a/b ! Divide 3.0 by 4.0.
    print *, a**b ! Raise 3.0 to the power 4.0.
end program main
```

开方怎么算? 自己想!

五个算数运算符的优先级和数学中是一样的.

```
program main
    implicit none
    print *, 1.0+2.0*3.0 ! Result is 7.0.
    print *, (1.0+2.0)*3.0 ! Result is 9.0.
    print *, 4.0*5.0**6.0 ! Result is 6.25e4.
    print *, (4.0*5.0)**6.0 ! Result is 6.4e7.
end program main
```

[5] 有特例, 见5.2.1.

严格上来说, Fortran 有七个算数运算符, 另两个也是 “+” 和 “-”. 众所周知, 数学中 “+” 和 “−” 前面可以没有数. Fortran 中也一样. 对于任意一个数据实体 `[q]`, `+[q1]/-[q1]` 就相当于 `[zero]+[q1]/[zero]-[q1]`, 其中 `[zero]` 是精度最低的种别的 `0`.

看下面这个例子就懂了.

```
program main
    implicit none
    real :: a
    a = 1.0
    print *, -a**2.0 ! Result is -1.0.
    print *, (-a)**2.0 ! Result is 1.0.
end program main
```

数学中, $-a^2$ 也是表示 $-(a^2)$ 哟!

运算符两边的类型相同时, 结果就是那个类型. 运算符两边的类型和种别都相同时, 结果就是那个类型那个种别. 运算符两边的类型或种别不同时, 又遇到类型转化或种别转化的问题了. 转化的基本规则就是两个数据实体中哪个更精确, 结果就转化成那个数据实体的类型和种别. 比如上面的程序, 把 `2.0` 改成 `2`, 结果不变. 因为无论是 `a` 还是`-a`, 都是实型, 肯定比整型的 `2` 精确, 所以结果就是实型的`-1.0` 和 `1.0`.

表达式既然是数据实体, 自然可以赋值给变量. 注意赋值时还要遵循赋值的类型和种别转化规则.

一般情况下都不需理会运算时的类型转化, 毕竟转化前后数学上差别不大. 但这里有一个非常传统, 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
```

乍一看, 这个程序非常聪明地将 $\sqrt[4]{2}$ 算了出来, 然而一运行, 就会发现不对. 为什么? 让我们来一点一点分析电脑是怎么干活的.

首先, 电脑先算 `1/4`. $1/4 = 0.25$, 这没什么问题. 然而, 根据刚才说的

类型转化规则, `1/4` 的结果必须转化成整型. 转化后什么结果? 和赋值时一样的规则, 向 0 取整, 结果是 `0`. 所以 `1/4` 就是 `0`!

然后电脑再算 `2**0`, 结果自然是 `1`.

也就是说, 将两个整型数据实体相 "除", 实际上是将它们整除!

再来看下面这个程序.

```
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
```

这个程序的结果也不大对. 让我们再来分析分析这个程序.

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`, $1/4 = 0.25$. 然而 `p` 和 `q` 都是整型, 所以结果是 `0`. 虽然 `r` 是实型, 然而并没有什么卵用. 因为是先计算, 再赋值, 所以计算时会先进行一次类型转化, 转化完了赋值时再进行一次类型转化. 先计算 `p/q`, 类型转化后得到 `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`.

这个坑非常经典, 好多程序语言都有, 就连 Python2 默认都会如此...

有一个非常简单的方法能避开上面所说的坑, 就是遵守规范14, 也就是: 统统用实型. 这可真是血泪教训啊!

最后还有一个小坑, 就是乘方从右边先算 (非常特殊). 比如 `a**b**c`, 到底是 a^{b^c}, 还是 a^{b^c}? 答案是 a^{b^c}, 也就是先算 b^c. 可以这么记: `(a**b)**c` 其实等于 `a**(b*c)`, 如果 `a**b**c` 是 `(a**b)**c`, 就显得太傻了, 还不如直接写 `a**(b*c)`[6], 所以 `a**b**c` 应该是 `a**(b**c)`.

[6] 乘法应该比乘方快...

```
program main
    implicit none
    real :: a, b, c
    a = 4.0
    b = 3.0
    c = 2.0
    print *, a**b**c ! This is a**(b**c).
end program main
```

只要记得这里有个坑就好了. 真遇到连着两个 **, 不清楚哪个先算, 加个括号呀. 即使自己清楚, 也最好加个括号, 一是让自己更清楚, 二是照顾照顾别人 (造轮子的时候), 因为别人很可能忘了到底哪个 ** 先算.

规范 15 *在表达式中适当地多用括号以保证表达式能被轻松理解.*

还有一个小细节. 严格意义上说, + 和-前面直接跟运算符是非法的. 然而器有器规呀, Gfortran 只是送个警告, Ifort 直接放行. 不过还是多加一个括号吧. 可以运行运行下面这个程序来看看结果.

```
program main
    implicit none
    real :: a, b
    a = 3.0
    b = 4.0
    print *, a+-b ! It is proper to write a+-b as a+(-b).
    print *, a--b ! It is proper to write a--b as a-(-b).
    print *, a*-b ! It is proper to write a*-b as a*(-b).
    print *, a/-b ! It is proper to write a/-b as a/(-b).
    print *, a**-b ! It is proper to write a**-b as a**(-b).
end program main
```

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. 来看下面这个例子.

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

如果把连接后得到的字符串赋给变量, 也要遵循字符串赋值的长度转换规则. 比如, 下面这个程序运行后 “什么也没输出”[7].

```
program main
    implicit none
    character(0) :: null_char
    null_char = 'Hello''','''' ''''world''''!'
    print *, null_char
end program main
```

5.2.3 逻辑运算

逻辑运算符一共有五个. 常用的有三个: .and., .or., .not., 分别代表与, 或, 非. 不常用的有三个: .eqv. 和.neqv., 分别代表同或和异或. 一般用真值表来展现逻辑运算的结果. 下面的程序会输出完整的真值表.

```
program main
    implicit none
    print *, '.and.'
    print *, ' ', .true., .false.
    print *, .true., .true..and..true., .true..and..false.
    print *, .false., .false..and..true., .false..and..false.
    print *, '.or.'
    print *, ' ', .true., .false.
    print *, .true., .true..or..true., .true..or..false.
    print *, .false., .false..or..true., .false..or..false.
    print *, '.not.'
    print *, .true., .not..true.
```

[7]请自己思考为什么这里加了引号. 可以另造个真什么也没干的程序, 运行看看有什么区别.

```
    print *, .false., .not..false.

    print *, '.eqv.'
    print *, ' ', .true., .false.
    print *, .true., .true..eqv..true., .true..eqv..false.
    print *, .false., .false..eqv..true., .false..eqv..false.
    print *, '.neqv.'
    print *, ' ', .true., .false.
    print *, .true., .true..neqv..true., .true..neqv..false.
    print *, .false., .false..neqv..true., .false..neqv..false.
end program main
```

5.2.4 关系运算

关系运算符有六个: `<`, `<=`, `>`, `>=`, `==`, `/=`. 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逻辑运算符混用. 此时请务必遵循规范15, 加上括号来明确究竟哪部分先算.

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 Ifort 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
```

这里有一个经典的坑. 像下面这样的程序可能得出 “错误” 的结果.

```
program main
    implicit none
```

```
    print *, (0.1+0.2)==0.3
end program main
```

因为实型其实是不能 “准确存储” 的, 比如对电脑而言, `0.1` 可能不是真的 0.1, 而是非常接近 0.1 的某个值, `0.2` 和 `0.3` 也是这样, 所以 `0.1` 加上 `0.2` 还真可能不等于 `0.3`. 这个 “问题” 广泛存在于各种编程语言中, 连 Python3 默认情形下都是如此. 然而上面的程序 Ifort 和 Gfortran 都能得出正确的结果, 似乎非常高级…

尽管得出正确结果了, 还是不好说其他情形下会不会出问题. 所以还是要慎重判断实型 (和复型) 变量是否相等 (或不等). 不过整型总是可以随便判断的, 因为整型量的存储是绝对精确的.

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个字符的先后顺序. 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 “`==`”.

注意, 上面 “补空格” 的一步只是暂时的, 比较前后变量本身没有改变. 比如下面这个程序, 比较前后 `c` 都是 “`I`”, 没有变成 “`I `”[8].

```
program main
    implicit none
    character(1) :: c
    c = 'I'
    print *, c=='You'
    print *, '"', c, '"'
end program main
```

至于编译器要怎么定义字符的先后顺序, Fortran 标准给了一些规定, 但没规定死, 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

[8] 否则长度就变成 3 了, 这不符合 “字符串变量长度不变” 的规则.

比较字符串的前后顺序不是件好事. 不过一般只会比较字符串是相同还是不同[9], 这可以放心比, 合格的编译器得出的结果都相同. 经测试, Ifort 和 Gfortran 在这方面都是合格的. 同志们可以自己研究研究, Ifort 和 Gfortran 的字符先后顺序到底是不是 ASCII 的顺序.

[9]恭喜同志们, 不必记上面那一大段规则了.

第六章　结构体

一般来说程序都是一行一行按顺序执行命令的, 但有一些特殊命令能让电脑瞬间跳到其他地方, 从那里开始向后按顺序执行命令, 比如在遇到一些结构体 (construct) 的时候. 有两种结构体是经常使用的: 条件结构体 (conditional construct) 和循环结构体 (loop construct).

6.1　条件结构体

6.1.1　if 结构体

假如我们要搞个轮子, 这个轮子能接受一个数, 然后判断这个数是不是零, 我们可以这么搞.

```
program main
    use, intrinsic :: iso_fortran_env, only: real128
    implicit none
    real(real128) :: number
    read(*,*) number
    if (number/=0) then
        print *, 'The number is not 0!'
    end if
end program main
```

这玩楞第五行是准备读取输入的.

如果是用 VS 2019 + Ifort 来运行这个轮子, 运行后啥子也没有, 连 “请按任意键继续. . .” 也没有, 还不报错. 这时只需要在那个黑框框里输入一个数儿, 然后回车, 就会有变化了.

如果是用 VS Code + Gfortran 来运行这个轮子, 运行后 Terminal 里会蹦出一堆编译命令, 然后这串命令下面会多出一行. 在这行里输入个数儿, 回车, 就会出新变化了.

无论是 VS 2019 + Ifort 还是 VS Code + Gfortran, 程序运行后都会先执行前四行, 然后执行第五行时停下了, 等输入数字按回车后, 电脑就会把变量 `number` 赋成输入的数儿, 然后继续执行下面的代码.

运行到第六行时, 电脑干了啥事? 看代码也许就能猜出来. 电脑是这么干活的: 首先计算 `number/=0`, 看其是`.True.` 还是`.False.`[1].

- 如果 `number/=0` 是`.True.`, 则运行 `then` 后面的所有代码. 对上面这个例子, 就是第七行, 然后运行 `end if` 及之后的代码.
- 如果 `number/=0` 是`.False.`, 则直接运行 `end if` 及之后的代码. 对上面这个例子, 第七行就被跳过去, 不执行了.

上面这个轮子里的 `if (...) then` 到 `end if` 的部分就是个最简单的 if 结构体, 其功能就是让电脑先判断刮弧里的是`.True.` 还是`.False.`.

- 如果是`.True.`, 则正常执行之后的每行命令
- 如果是`.False.`, 则直接从 `end if` 开始执行之后的命令.

`then` 之后直到 `end if` 前的部分我们可以称之为 then 块 (block). 这部分是从属于 if 结构体的一串代码儿, 我们应当按照规范6, 给这段缩个进, 来提醒我们这段是很特殊的, 是在 if 结构体里头的.

上面的这个轮子有个小问题, 就是运行后首先什么也没说. 如果人们用这个轮子, 他们就不知道要干啥了. 所以我们本应在第五行之前加句话, 输出串提示 (其他轮子里也要这么做). 但我突然懒了, 就不干了.

另外这个轮子在 `number==0` 时啥子也没输出, 这可能让同志们头晕. 俺在设计这个例子的时候, 是把 `number==0` 视作 “正常情形”, 把 `number/=0` 视作 “异常情形”, 这其实是很通行的一个约定. 而 “正常情形” 下, 按 UNIX 哲学, 就应该啥子也不输出, 表示 “一切正常”, “异常情形” 下才说话.

当然让轮子在 `number==0` 时也说话, 也是好的. 我们可以这么干.

[1]这个玩楞似乎违背了之前所说的一些原则, 但相信我这个轮子是不会判断出错的, 不过之前的原则还是要在其他场合遵守的!

```
program main
    use, intrinsic :: iso_fortran_env, only: real128
    implicit none
    real(real128) :: number
    read(*,*) number
    if (number/=0) then
        print *, 'The number is not 0!'
    else
        print *, 'The number is 0!'
    end if
end program main
```

现在轮子里多了 `else` 和后面到 `end if` 中间称之为 else 块的东东, `else` 前到 `then` 后的内容则是 then 块. 现在整个 if 结构体的执行方式就是, 先判断刮弧里的是`.True.` 还是`.False.`.

- 如果是`.True.`, 则一行一行执行 then 块, 然后跳到 `end if`.
- 如果是`.False.`, 则一行一行执行 else 块, 然后跳到 `end if`.

知道上面的内容, 干活就已经不成问题了. 但人嘛总是想着偷懒的. 下面介绍一些个偷懒的办法.

首先第一个轮子, 我们可以少打一些字儿, 像这样.

```
program main
    use, intrinsic :: iso_fortran_env, only: real128
    implicit none
    real(real128) :: number
    read(*,*) number
    if (number/=0) print *, 'The number is not 0!'
end program main
```

其中第六行的这种东东称为 if 语句, 就是删掉 `then`, 然后把 then 块里的话搬到刮弧后面. 如果我们需要在 then 块里加很多很多行, 那 if 语句就不好使了. 这时我们就老老实实用 if 结构体.

还有, 如果我们要让上面的轮子还能判断是正是负, 我们可以把轮子改成下面这样.

```
program main
    use, intrinsic :: iso_fortran_env, only: real128
    implicit none
    real(real128) :: number
    read(*,*) number
    if (number==0) then
        print *, 'The number is zero!'
    else
        if (number>0) then
            print *, 'The number is positive!'
        else
            print *, 'The number is negative!'
        end if
    end if
end program main
```

也就是说我们在 if 结构体里再搞个 if 结构体 (当然要按规范6再缩进一回). 只套两层还是能活的, 如果要搞一些其他活儿, 比如输入一个考试成绩, 判断是优良中差, 缩进都要缩死人了. 啥子? 不缩进了? 那您可放心了, 保证您要绞尽脑汁地去判断计几的一句话到底是在哪层 if 结构体中.

像下面这样, 就可以偷懒了.

```
program main
    use, intrinsic :: iso_fortran_env, only: real128
    implicit none
    real(real128) :: number
    read(*,*) number
    if (number==0) then
        print *, 'The number is zero!'
    else if (number>0) then
        print *, 'The number is positive!'
    else
        print *, 'The number is negative!'
    end if
end program main
```

这样不仅可以少打个 `end if`, 还能少缩进一回, 让我不由得想起王司徒的经典语录呀.

同志们可以尝试自己造一个判断分数优良中差的轮子, 练练手. 这个轮子可能有些同志造好之后, 觉得看着代码就不舒服, 因为 if 结构体只能实现双分支, 堆叠 `else if` 是在强行用双分支来堆出多分支来, 有些同志可能就不爽, 想直接在轮子里弄出多分支来. 这些个同志可以自己去搜 `select case` 的用法. 还有上面的轮子都有个问题, 就是如果用户突然发神经, 输个什么字母之类的, 这个轮子就崩了. 也许用 `select type` 能解决这个问题.

最后再讲个小技巧. 假如现在要搞个轮子, 在 `number==0` 时输出条警告, 在 `number/=0` 时啥也不干, 下面这样当然是可的.

```
program main
    use, intrinsic :: iso_fortran_env, only: real128
    implicit none
    real(real128) :: number
    read(*,*) number
    if (number==0) print *, 'Warning: the number is zero.'
end program main
```

但我大胆猜测, 机器判断 `number/=0` 显然比判断 `number==0` 要容易[2], 所以为提高效率, 最好把判断条件变成 `number/=0`. 我们可以这么做.

```
program main
    use, intrinsic :: iso_fortran_env, only: real128
    implicit none
    real(real128) :: number
    read(*,*) number
    if (number/=0) then
        continue
    else
        print *, 'Warning: the number is zero.'
    end if
end program main
```

[2]别信, 这我真是猜的.

其中 `continue` 就是让电脑运行这行时啥都不干. 事实上把 `continue` 这行删了, 电脑还是可以正确工作, 在 `number/=0` 时直接跳到 `end if` 的, 但我觉得不写 `continue` 这行, 同志们和很多被 Fortran 折磨的人 (相信不包括同志们) 见了都要吓一跳, 不知道这玩楞是否能跑起来. 为了可读性, 还是写上好. 也不用担心加上这行后电脑会因为执行这个啥都不干的命令而变慢. 事实上电脑接到一个空转命令后, 可能反而跑得更快[3]. 我们要坚信 Ifort 和 Gfortran 这样优秀的编译器会竭尽所能地帮我们提高效率.

6.2 循环结构体

6.2.1 do 结构体

这是一个漂亮的定理.

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

我们可以将左边的级数截断至一个 $n_{\max}$ 来计算出 π 的近似值. 假如我们取 $n_{\max} = 10$, 我们还可以这么做.

```
program main
    use iso_fortran_env, only: qp=>real128
    implicit none
    print *, (6*(1/1_qp**2+1/2_qp**2 &
                +1/3_qp**2+1/4_qp**2 &
                +1/5_qp**2+1/6_qp**2 &
                +1/7_qp**2+1/8_qp**2 &
                +1/9_qp**2+1/10_qp**2))**0.5_qp
end program main
```

但如果我们取 $n_{\max} = 100$, 这样做就要死人了. 这时我们就有必要使出 do 结构体, 像下面这样.

[3]这是真的.

```
program main
    use iso_fortran_env, only: qp=>real128
    implicit none
    integer :: n
    real(qp) :: s
    s = 0.0_qp
    do n = 1, 100
        s = s+1.0_qp/n**2.0_qp
    end do
    print *, (6.0_qp*s)**0.5_qp
end program main
```

这个轮子运行到第七行后发生什么事了? 首先电脑把 n 赋成 1, 然后执行 do 到 end do 之间的命令, 也就是第八行. 执行完第八行后, 电脑把 n 加上 1, 然后再一次执行第八行, 然后再把 n 加上 1… 直到 n 等于 100 后, 电脑最后一次执行第八行, 然后就执行 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, 执行第八行, 令 n 等于 2, 执行第八行… 令 n 等于 99, 执行第八行, 令 n 等于 100, 执行第八行, 然后执行第十行. 总而言之是完成目的了!

第七行逗号前后的东东也可以是变量或表达式, 只要是个数据实体就成. 假如我们要让用户自己选择 $n_{\max}$ 多大, 我们就可以这么做.

```
program main
    use iso_fortran_env, only: qp=>real128
    implicit none
    integer :: n_max
    integer :: n
    real(qp) :: s
    s = 0.0_qp
    read(*,*) n_max
    do n = 1, n_max
        s = s+1.0_qp/n**2.0_qp
    end do
    print *, (6.0_qp*s)**0.5_qp
end program main
```

上面的程序每次 n 都增加 1. 我们可以改变这一点. 看这个极限.

$$\lim_{n_{\min}\to-\infty}\sum_{n=n_{\min}}^{-1}\frac{1}{n^2}$$

我们可以这么计算这个极限的近似值.

```
program main
    use iso_fortran_env, only: qp=>real128
    implicit none
    integer :: n_min
    integer :: n
    real(qp) :: s
    s = 0.0_qp
    read(*,*) n_min
    do n = -1, n_min, -1
        s = s+1.0_qp/n**2.0_qp
    end do
    print *, s
end program main
```

注意第九行现在后面多出了个-1, 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 会取-1, -2, 一直到 n_min. 同样, 这个-1 也可以被替换成变量或表达式之类的.

让我们进一步探讨到底 do 结构体工作时会干什么事. 我们可以使用 print 大法来看程序的工作过程. 下面是一个示例程序, 可以执行它看看.

```
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
```

一个 do 结构体前头肯定有一个类似于 `do m = m1, m2, m3` 的东东, 这里的 `m` 称作计数变量 (counter variable), `m1` 称作初始参数 (initial parameter), `m2` 称作终端参数 (terminal parameter), `m3` 称作增量参数 (incrementation parameter), 当然 `m3` 可以没有, 没有时 `m3` 就是 `1`. 我们暂且把 `do` 所在那行和 `end do` 所在那行之间的部分称作 do 块.

总的来说, do 结构体的工作流程就是:

1. 将计数变量赋成初始参数.
2. 执行 do 块.
3. 将计数变量加上增量参数.
4. 若计数变量在以初始参数和终端参数为端点的闭区间中, 则执行 do 结构体后面的内容, 否则回到第 2 步.

上面程序的计数变量都是整型的. 按 Modern Fortran 规范, 计数变量 "shall be" 整型的. 这是无比正确的. 来看下面这个例子.

```
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 跑这个程序的同志们会发现出现一堆警告, 而且 `n` 的输出结果怎么是 `1000`. 仔细看看, `r` 的输出结果可不是 `2.001`. 要知道, 实型量运算是有误差的, 这里 `r` 被加了个一千次, 疯狂积累误差, 刚好使 `r` 大于 `2.0` 一丢丢, 电脑就退出循环了, 算出了个我们不想要的结果!

用 Ifort 的同志们会发现没出警告, 而且结果还不错. 可别高兴得太早, 把 `0.001` 改成 `0.000001`, 再跑一回, 就会发现事情不对了, 还连警告也没

有…

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, 放在下面, 大家自己品品.

```
program main
    implicit none
    integer :: n, r
    real :: s
    n = 0
    s = 0.0
    do r = 1000, 2000, 1
        n = n+1
        s = s+(r/1000.0)**0.5
    end do
    print *, n, r, s
end program main
```

最后给大家留两个小作业.

第一个是斐波那契兔子问题. 第 0 月有一对小兔, 每过一个月小兔都会长成大兔, 每过一个月大兔都会生一对小兔, 问第 12 月有多少对兔? 我敢保证结果是 233, 但我觉着很多同志一开始都算不出来 233[4].

第二个是求下面这个矩阵所有元素的和.

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

我觉得这一定需要在一个 do 结构体里再来另一个 do 结构体.

6.2.2 do while 结构体

严格上说 do while 结构体是 do 结构体中的一种, 不过一般还是把 do while 结构体单划成一类.

[4]我在双关, 发现了么?(^v^)

之前我们用了一个级数计算 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
    use iso_fortran_env, only: qp=>real128
    implicit none
    integer :: n
    real(qp) :: s
    n = 0
    s = 0.0_qp
    do while (s<3.14_qp**2/6)
        n = n+1
        s = s+1.0_qp/n**2.0_qp
    end do
    print *, n
end program main
```

这个程序的运作过程不难理解. 运行到第八行时, 电脑会判断刮弧里的量是否为真. 若为真, 则执行后边直到 `end do` 的内容, 然后回到第八行, 再判断刮弧里的量是否为真… 若刮弧里的量不为真, 则执行 `end do` 之后的内容. `do while ...` 到 `end do` 这样的部分就是 do while 结构体

这里要注意最后一句话是 `end do` 而不是 `end do while`, 因为 do while 结构体其实是 do 结构体中的一种.

6.2.3 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. 现打算造个轮子来判断 n^2 和 $(n+1)^2$ 之间的每个数是否是质数. 先造出个能跑的轮子如下.

```
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
```

这个轮子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` 就可以得出 `i` 不是质数了, 所以希望在 `i==i/j*j` 时, 将 `is_prime` 赋成`.false.`, 然后直接执行里层的 `end do` 之后的语句 (也就是第十三行). 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
```

在上面这个轮子中, 电脑只要一见到 `exit inner_loop` 就浑身一激灵, 然后就直接执行 `end do inner_loop` 之后的语句了. 注意上面这个轮子, 使用了鲜少出现的标签, 来明确指明执行 `exit` 后究竟执行哪个 `end do` 之后的语句. 其实直接写个 `exit`, 电脑还是干一样的活儿, 但不加标签的话, 总会令人脑子有点晕: 到底 `exit` 后执行的是倒数第三行还是倒数第一行?

所以还是得用用标签. 当然如果 do 结构体只有一层就不用了.

如果要让电脑在 `i==i/j*j` 时, 将 `is_prime` 赋成`.false.`, 然后直接执行外层的 `end do` 之后的语句 (也就是最后一行), 可以这么做.

```
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
```

同志们可以自己尝试在 do while 结构中使用 exit 语句.

6.2.4 cycle 语句

cycle 语句和 exit 语句只是功能不同, 语法完全一致.

对内层使用 cycle 语句就像下面这样.

```
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.
                cycle inner_loop
            end if
        end do inner_loop
        print *, i, is_prime
    end do
end program main
```

跑这轮子, 电脑只要一见到 `cycle inner_loop` 就浑身一哆嗦, 然后就直接执行 `end do inner_loop`, 也就是把 `j` 加上 `1`, 然后再执行 `inner_loop: do` 之后的语句, 之后可能又遇上 `cycle inner_loop`, 那就再来一回…

对外层使用 cycle 语句就像下面这样.

```
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
```

同志们可以自己先想想这个轮子会跑出什么结果, 然后跑跑看.

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). 用非常严谨的方式来讨论数组, 以我这语文水平肯定是不行滴, 讲着讲着同志们就头疼, 所以我接下来讲的内容会不太严谨, 但对于实际应用来说肯定是不成问题的.

数组是形如 $a:\{j_1,\ldots,k_1\}\times\ldots\times\{j_n,\ldots,k_n\}\to\mathbb{C},(s_1,\ldots,s_n)\mapsto a_{s_1\ldots s_n}$ 的东东. 正整数 n 称为数组 a 的维数/秩 (rank). 任意 $i\in\{1,\ldots,n\}$, j_i 和 k_i 称为数组 a 的第 i 个维度 (dimension) 的下界 (lower-bound) 和上界 (upper-bound), k_i-j_i+1 称为数组 a 的第 i 个维度的长度 (extent). 矢量 $(k_1-j_1+1,\ldots,k_n-j_n+1)$ 称为数组 a 的形状 (shape), 其本身是一维数组. $\prod_{i=1}^{n}(k_i-j_i+1)$ 称为数组 a 的大小 (size). 标量可以当成维数为 0 的数组.

形如 $a_{s_1\ldots s_n}$ 的东东称为数组 a 中的元素 (element). 任意 $i\in\{1,\ldots,n\}$, 称 s_i 为 $a_{s_1\ldots s_n}$ 的第 i 个下标 (subscript) 或第 i 个索引 (index). 同一个数组中的所有元素都有相同的类型和种别.

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 $a_{t_1\ldots t_n}$ 后面[1].

7.1　数组声明

用数组之前当然要声明它. 数组的声明和标量的声明还是很像的, 也是需要给一个数据类型 (整型/实型/复型), 还可以给一个种别. 数组的类型和种别就是数组中每一个元素的类型和种别.

[1]这和 Matlab 的默认情形一样, 但和 C/Python 的默认情形不一样!

为表明声明的变量是个数组, 还需给出 $j_1,\ldots,j_n,k_1,\ldots,k_n$. 有两种方法可行, 一种是在变量名后加 `(j1:k1,...,jn:kn)`, 另一种是在类型和种别后加`,dimension(j1:k1,...,jn:kn)`. 所有 `ji`, `ki` 都必须是整型常量或整型常量表达式[2]. 可以省略任意 `ji:`, 如果省略 `ji:`, 则 `ji` 为 1[3].

请大家猛戳这个链接获取数组声明示例.

选择哪种声明方式呢? 一般大家喜欢用第一种方式, 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一样的数组[4], 这时第一种方式反而不好使了, 大家就喜欢用第二种方式.

和字符串一样, 有些时候我们并不清楚到底要用一个什么样的数组, 这时我们就可以和字符串一样, 先在类型和种别后加`,allocatable`, 然后再在变量名后加 `(:,...,:)` 或在类型和种别后加`,dimension(:,...,:)`, 其中`:`的个数等于数组的维数, 这样我们就搞出一个延迟形状数组 (deferred-shape array). 假如我们声明的数组叫 `a`, 接下来我们就可以像延迟长度字符串那样, 用 `allocate(a(j1:k1,...,jn:kn))` 把 $j_1,\ldots,j_n,k_1,\ldots,k_n$ 确定下来 (仍可省略 `ji:`), 用 `deallocate(a)` 将 $j_1,\ldots,j_n,k_1,\ldots,k_n$ 重新变成未定义的, 然后再来个 `allocate(a(j1:k1,...,jn:kn))`…

请大家猛戳这个链接获取延迟形状数组声明示例.

7.2 数组构造

数组构造器 (array constructor) 是一维数组常量, 形如 `[e1,...,em]`, 其中 `ei` 可以是标量也可以是数组. 若将 `[e1,...,em]` 记为 $(a_1,\ldots,a_n)$, `ei` 的大小记为 S_i[5], 则 $a_{(S_1+\cdots+S_{i-1}+l)}$ 为 `ei` 的第 l 个元素. 如果 `ei` 维数大于 1, 则按本章开头 "数组元素顺序" 找到第 l 个元素.

直接了当的说法是: 把所有 `ei` 重新排成一维数组并保证元素顺序不变, 然后首尾相接拼起来.

在数组构造器中还可以使用一种神奇的隐式 do 循环 (implied `do` loop), 这玩意儿形如 `((...(e(i1,...,in),i1=p1,q1,r1)...),in=pn,qn,rn)`, 里头 `e(i1,...,in)` 是个数据实体, 通常是个含有 `i1`,…,`in` 的表达式, `i1`,…,`in` 是一堆事先声明过的整型变量, 整个隐式 do 循环相当于一维数组. 下面这

[2]就是只含整型常量的表达式.

[3]这和 Matlab 的默认情形一样, 但和 C/Python 的默认情形不一样!

[4]方法同标量声明.

[5]标量的大小当然是 1.

样的两个程序总是等价的[6]. 自然任意,ri 都可以省略 (默认为 1).

```
program normal_do_loop
    ...
    integer :: i1
    ...
    integer :: in
    integer :: i
    i = 0
    do in = pn, qn, rn
        ...
            do i1 = p1, q1, r1
                i = i+1
                a(i) = e(i1,...,in) ! a is [a(1),...,a(S)].
            end do
        ...
    end do
    ! Now i == S.
end program normal_do_loop

program implied_do_loop
    ...
    integer :: i1
    ...
    integer :: in
    a = [((...(e(i1,...,in),i1=p1,q1,r1)...),in=pn,qn,rn)]
end program implied_do_loop
```

隐式 do 循环属于比较高级的语法, 还是稍稍让人不好理解滴. 幸好不用隐式 do 循环也总是可以完成任务, 所以可以干脆不用. Python 中也有一个类似的叫列表推导式的东东, 不过当年老师好像根本没讲过, 我也只是在耍帅时会用这玩楞.

用 Ifort 的话, 可以用 `p:q:r` 代替 `(i1,i1=p,q,r)`, 当然`:r` 可以省略. 这是 Ifort 的器规, Gfortran 是不认的[7]. `(i1,i1=p,q,r)` 这样的一重简单隐

[6]看懂这两个程序, 同志们可能需要先读7.5节.

[7]所以造轮子时最好不用这东东.

式 do 循环比较常用, 是要掌握的. 我再把上一对示例程序的简单情形重新列一下, 其中 `a(i) = i1` 的意思就是令 a_i 为那时的 i_1.

```
program normal_do_loop
    ...
    integer :: i1
    integer :: i
    i = 0
    do i1 = p, q, r
        i = i+1
        a(i) = i1
    end do
end program normal_do_loop
```

```
program implied_do_loop
    ...
    integer :: i1
    a = [(i1,i1=p,q,r)]
end program implied_do_loop
```

举个更具体的例子: 输出 1 到 9 中奇数的平方.

```
program main
    integer :: i, odd_squares(5)
!   integer :: s
!   s = 1
!   do i = 1, 9, 2
!       odd_squares(s) = i**2
!       s = s+1
!   end do
    odd_squares = [(i**2, i=1,9,2)]
    print *, odd_squares
end program main
```

我们可以对先前构造出来的一维数组进行变形 (reshape) 操作来获取多维数组, 只需来个 `a_new = reshape(a_old,s)` 就成. `a_new` 和 `a_old` 是两个数组, `s` 是 `a_new` 的形状 (当然得是整型一维数组), `a_new` 的第 l 个

元素和 a_old 的第 l 个元素总是相同的[8]. 整个变形操作说白了就是: 将数组 a_old 中的元素复制到形状为 s 的数组 a_new 中, 并保证元素顺序不变, 比如下面这样.

```
program main
    implicit none
    integer :: one2four(2,2)             ! a11=1 a12=3
    one2four = reshape([1,2,3,4],[2,2]) ! a21=2 a22=4
end program main
```

7.3 数组切片

数组切片 (slicing) 是用一个数组得到另一个数组的操作. 现假设有一个维数为 n 的数组 a, 则 a(e1,...,en) 是另一个数组, 其中 e1,…,en 乃整型一维数组或整型标量. 如何确定 a(e1,...,en)?

引入 v1,…,vn, 保证若 ei 为数组则 vi 为 ei, 若 ei 为标量则 vi 为 [ei]. 这样 a(v1,...,vn) 等于 b. 记 vi 为 $(v_{i;1},\ldots,v_{i;l_i})$, 则 $b_{\iota_1,\ldots,\iota_n} = a_{v_{1;\iota_1},\ldots,v_{n;\iota_n}}$.

假设 e1,…,en 中 ei1,…,eim $(i_1<\ldots<i_m)$ 是长度为 $l_{i_1},\ldots,l_{i_m}$ 的数组, 其他是标量, 则可将 b 变形成形状为 $(l_{i_1},\ldots,l_{i_m})$ 的 c, c 就是 a(e1,...,en).

我本来处心积虑地想再来几段话来把这切片讲得更明白些, 然后我就放弃了, 只好先来个示例给同志们做练习. 我敢保证自己写的东东肯定是真实不虚的, 但看来是很难理解记忆了. 幸好非常复杂的数组切片一般是用不上的. 如果老师敢考那些难死人的切片, 我们就当即暴动~

```
program main       ! a000=1 a001=5   a100=2 a101=6
    implicit none ! a010=3 a011=7   a110=4 a111=8
    integer :: i, one2eight(0:1,0:1,0:1)
    integer :: result(1,4)
    one2eight = reshape([(i,i=1,8)],[2,2,2])
    result = one2eight(0,[0],[0,1,1,0]) ! The shape is [1,4].
    ! result X one2eight(0,0,[0,1,1,0]) ! The shape is [4]!
    print *, result
```

[8]不考虑赋值时的类型和种别转化的情况下.

```
end program main
```

先前用向量下标 (vector subscript) 来切片, 我们还可以用三元下标 (triplet subscript)[9], 以 `p:q:r` 代替 `(i,i=p,q,r)`, 当然`:r` 可以省略. 这不是器规, 是通用的. 而且三元下标中 `p` 和 `q` 也可以省略 (但是注意 `p` 和 `q` 之间的`:` 不能省), `p` 省略就等于那一维的下界, `q` 省略就等于那一维的上界. 这样的切片简单且比较常用 (尤其是省略`:r` 的时候), 是要掌握的. 比如我们可以方便地摘出 1 到 9 中的奇数.

```
program main
    implicit none
    integer :: i
    integer :: singles(9), odds(5)
    singles = [(i, i=1,9)]
    odds = singles(::2) ! singles(1:9:2)
end program main
```

还有, 如果 `i1,...,in` 都是整型标量, 则 `a(i1,...,in)` 就是 $a_{i_1\dots i_n}$, 这更要掌握. 问: `a(i1,...,in)` 和 `a(i1:i1,...,in:in)` 有什么区别?

对数组切片, 可以得到一个新数组, 看起来可以对这个新数组再切片. 然而这是不成的, 原因在于对数组切片得到的新数组, 其每一维的上下界其实都是不确定的[10], 所以新数组中每个元素的下标都是不确定的, 因此没法切片. 同样地, 由数组构造器得到的数组也是不能切片的.

7.4 数组运算

有了数组, 总是要用数组来算些什么东西. 现在我们可以在表达式中混用数组, 标量和5.2节中的所有运算符. 运算符的优先级, 和之前是一样的, `+` 和`-`之前如果什么也没有, 也还是默认有个 `0`. 只需要知道, 当运算符两边出现数组时会有什么结果, 我们就能推理出任意表达式的结果了. 这又分两种情况.

- 运算符两边都是数组.
- 运算符一边是标量, 另一边是数组.

[9] 官方文档里用的是 “subscript triplet”.

[10] 虽然在介绍切片规则时看起来有确定的上下界, 那只是为了说话方便.

如果运算符两边都是数组, 我们首先必须保证这两个数组形状完全一致, 绝对一致, 这样这两个数组中的元素按所处的位置, 自然就能一一对应. 我们假设两个数组分别是 a 和 b, 并用符号 $\star$ 表示一个运算符, 现在我们要算 $a \star b$. 假设 a 和 b 的下界分别为 $j_{a;1}, \ldots, j_{a;n}$ 和 $j_{b;1}, \ldots, j_{b;n}$, 则首先可以搞到另外两个数组 α 和 β, 使得 $\alpha_{i_1 \ldots i_n} = a_{(i_1+j_{a;1}) \ldots (i_n+j_{a;n})}$, $\beta_{i_1 \ldots i_n} = b_{(i_1+j_{b;1}) \ldots (i_n+j_{b;n})}$, 然后可以弄出一个数组 c, 使得 $c_{i_1 \ldots i_n} = \alpha_{i_1 \ldots i_n} \star \beta_{i_1 \ldots i_n}$, 则 c 就是 $a \star b$. 简单来说就是对 a 和 b 中两两对应的元素进行 $\star$ 运算, 得到新数组. 示例如下.

```
program main      ! 1+5=6 3+7=10
    implicit none ! 2+6=8 4+8=12
    integer :: one2four(2,2), five2eight(2,2,1)
    one2four = reshape([1,2,3,4],[2,2])
    five2eight = reshape([5,6,7,8],[2,2,1])
    print *, one2four+reshape(five2eight,[2,2])
    ! print *, one2four+five2eight (X)
end program main
```

如果运算符一边是标量, 另一边是数组, 不妨设数组为 a, 标量为 b, 仍设运算符为 $\star$. 此时若 a 的形状为 $\vec{s}$, 则可以另搞一个形状为 $\vec{s}$ 的数组 $\tilde{b}$, 使得 $\tilde{b}$ 中任意元素都是 b, 然后就有 $a \star b = a \star \tilde{b}$, $b \star a = \tilde{b} \star a$. 示例如下.

```
program main      ! 1+9=10 3+9=12
    implicit none ! 2+9=11 4+9=13
    integer :: one2four(2,2)
    one2four = reshape([1,2,3,4],[2,2])
    print *, one2four+9
end program main
```

即使表达式的结果是一个数组, 也不能对其切片, 因为此时数组的上下界依然是不确定的. 比如, 有两个一维数组, 一个下界是 1, 一个下界是 0, 把这俩加起来, 得到的数组上下界应该是多少? 不好规定. 有些情况下, 比如让两个下界都是 1 的一维数组相加, 看起来数组上下界是好规定的, 然而若真来个规定, 造编译器的人就得处心积虑地要让编译器能够区分这两种情形, 他们会很不开心, 很不快乐, 所以统统规定上下界不确定是好的.

7.5 数组赋值

数组赋值可分两种, 一种 "=" 右侧是数组, 另一种 "=" 右侧是标量.

如果 "=" 右侧是数组, 则俩玩楞形状必须一样滴. 假设我们要令 `a=b`, 并设 n 维数组 a 和 b 的下界分别为 $j_{a;1},\ldots,j_{a;n}$ 和 $j_{b;1},\ldots,j_{b;n}$, 则 a 和 b 的元素自然能一一对应. 首先我们要搞一个 $\tilde{b}$, 使得 $\tilde{b}_{(i_1-j_{b;1}+j_{a;1})\ldots(i_n-j_{b;n}+j_{a;n})}=b_{i_1\ldots i_n}$, 然后令 a 等于 $\tilde{b}$ 即可. 简单来说就是令 a 和 b 中两两对应的元素相等. 这里 b 的上下界可能是不确定的, 但赋值给 a 后, a 先前声明过, 所以其上下界一定是确定的.

如果 "=" 右侧是标量, 比如数组是 a, 标量是 b, 则先把 b 变成和 a 形状相同的数组 c, 使得 c 中任意元素都是 b, 然后令 `a=c` 即可.

对于数组, 我们还可以用一些特殊东东来赋值, 比如 forall, where 和 do concurrent, 不过这些东东我自己貌似会用, 却研究不出它们的明确规则. 我计划在第**??**章介绍它们.

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, 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, 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). 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
```

这样一看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. 至于到底咋么开的冰箱, 咋么塞的大象, 咋么关的冰箱, 看对应的 `subroutine` 到 `end subroutine` 里咋写的就知道了. 把整个程序 (冰箱塞大象) 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, 把 `elephant` 换掉就行了, 哦, 还要把大象拿出来, 再多造个过程…

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 (见3.1节), 法律和法律条文的关系一样, 后者是表述前者的一堆字符[1]. 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.

[1] 今后不再区分过程和子程序.

```
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` (示例中为 `ab2bc_then_sumabc`)

称为子例行子程序名. 一般存这个子程序的文件就会取成 `XX.f90` (比如示例中的子程序就可以存入名为 `ab2bc_then_sumabc.f90` 的文件中). 当然, 子程序名得尽量表明子程序的功能.

3. 子程序和主程序一样都是程序单元 (见3.4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(...)`, 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, 会出警告, 输出 `0.00000000`, Ifort 日常宽松, 不出警告, 但也输出 `0.0000000E+00`…

```
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 (哑实结合当然还是可以的), 比如下面这个程序跑不得, Ifort 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在程序中是不应当被赋值的, 我们就可以让其成为只读的, 这样如果我们一不小心写错了, 在程序中给这个哑参量赋值了, 就能马上查出来.

```
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 会给警告, Ifort 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是没有被赋值的, 所以下面这程序是天也不知会出什么结果的. 但 Ifort 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 not be defined 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 16 *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, 我安排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嘛? 子程序名能和子程序里的变量的变量名一样嘛? 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嘛? 子程序能直接或间接地调用自己嘛?...

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 17 *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71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. 固有过程有哪些怎么用请猛戳这个链接查询. 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. 注意, `acos` 是个函数, 要声明的.

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

8.2 过程中的变量

8.2.1 parameter 属性

我们在玩轮子的时候经常会遇到这么个问题, 比如我们需要反复进行一些和 π 有关的计算, 假设我们的程序以双精度运行, 我们会造一个实型

双精度变量 pi, 然后赋值 acos(-1.0_dp), 其中 dp 为 real64, 这本来没什么问题, 但如果我们一不小心写错了程序, 悄咪咪地又给这个 pi 赋了个别的值, 那就出了大问题, 而且编译器还不会告诉我们, 我们就会死都找不出哪里写错了. 如果我们把 pi 改成 3.141592653589793_dp 或 acos(-1.0_dp), 那倒是不大会出错了, 但每次看轮子, 我们脑子都要反复判断出 3.141592653589793_dp
或 acos(-1.0_dp) 其实就是 π, 我们的眼睛和脑子就会觉得好伤心好难过呜呜呜呜.

这时候我们就可以给 pi 加上 parameter 属性[2], 造一个具名常量 (named constant). 比如下面这个程序, 我们在声明 pi 时加了,parameter, 并在声明的同时给 pi 赋值 acos(-1.0_dp), 这个 pi 就永远是 acos(-1.0_dp) 改不了了, 比如我们下面又把 pi 赋成 exp(1.0_dp), 合格的编译器就会立即跳出来把我们打回去重新改程序.

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    real(dp),parameter :: pi=acos(-1.0_dp)
    pi = exp(1.0_dp) ! It is impossible to change pi!
end program main
```

同志们可以自己尝试用子程序给具名常量赋值, 看看会出什么事, 可以把第73页的 “隐晦赋值” 程序改一改来考验考验编译器.

8.2.2 save 属性

和 parameter 属性有一丢丢像的是 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()
```

[2]属性的含义见第??章.

```
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. 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赋值, 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. 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
    integer :: n = 0 ! n has SAVE attribute, though no ,save is here!
    n = n+1
    print *, n
end subroutine count
```

因此, 不要这么写, 除非前面已经标好,parameter 或,save.

规范 18 *除非已经明确标明变量有* parameter *属性或* save *属性, 否则不要在声明变量的时候直接给变量赋值.*

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的.

```
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52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.

假定形状数组

之前第82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 `:` 表示其必须是 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 Gfortran 是跑不了的, 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8.3节), 所以要写成下面这个样子才行. Ifort 能跑, 但结果是错的...

```
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8.3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) ! trapezoidal rule
    use iso_fortran_env, only: dp => real64
    implicit none
    abstract interface
        function func(x) result(y)
            use iso_fortran_env, only: dp => real64
            implicit none
            real(dp),intent(in) :: x
            real(dp) :: y
        end function
    end interface
    procedure(func) :: f
    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 过程接口

之前第84页和第86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).

8.3.1 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:

- 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7.1节),
  - 是假定形状数组,
- 过程是函数且结果是数组,
- 过程是另一个过程的实参量,

那么就一定要加特定接口[3]. 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

[3] 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.

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
    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[4]来调整格式)

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

[4]不知道什么是列选择的同志请自行了解.

```
        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
    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

凭他们的经验告诉我们, 有些情况 (比如上面列出来的), 如果没有接口, 会出大事情. 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) ! trapezoidal rule
    use iso_fortran_env, only: dp => real64
    implicit none
    abstract interface
        function func(x) result(y)
            use iso_fortran_env, only: dp => real64
            implicit none
            real(dp),intent(in) :: x
            real(dp) :: y
        end function
    end interface
    procedure(func) :: f
    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) ! trapezoidal rule
    use iso_fortran_env, only: dp => real64
    implicit none
    abstract interface
        function func(x) result(y)
            use iso_fortran_env, only: dp => real64
            implicit none
            real(dp),intent(in) :: x
            real(dp) :: y
        end function
    end interface
    procedure(func) :: f
    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_
```

8.3.2 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`). 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, 发现 `sgn_real64` 这个函数参量是双精度的, 匹配, `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!

哑实结合时的类型–种别–维数匹配 (type-kind-rank compatibility, TKR compatibility), 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` 其实长得一模一样, 就是变量种别不同, 如果我们还要造单精度的轮子, 还要造整型的轮子, 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. 不过用预处理器 (见第**??**章) 的话, 应该可以省事儿不少还易读, 但这已经是超 Fortran 的内容了.

8.3.3 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 “一样”, 都是函数, 都只有一个整型只读参量, 结果的类型, 种别和形状也一样, 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[5], 只要在接口块里一致即可. 然后我们只需要再写 `procedure(dim2mat) :: ...` 就相当于给`...` 加特定接口了, 但前提是`...` 必须是和 `dim2mat` 特征相同的过程 (如 `eye` 和 `minkowski`).

在第86页的轮子里, 我在声明哑过程的时候用了抽象接口, 这个地方, 根据我自己的实验, 非用抽象接口不可. 按我的理解, 这是因为哑过程根本不是一个 "真" 过程: 既不是我们用代码定义的过程, 也不是编译器定义好的固有过程. 只有 "真" 过程能有特定接口, 因为电脑碰到特定接口的时候, 会去查特定接口对应的过程的定义, 而哑过程只有声明没有定义, 所以声明哑过程的时候就只能用抽象接口了.

[5] 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    ... Units ... Explanations
-----------...-------...--------------------------------
     ...          ...          ......
     1-  28 ... ---    ... Unique source designation ...
     ...          ...          ......
   129- 137 ... mas    ... ? Parallax (parallax)
     ...          ...          ......
   987-1001 ... deg    ... 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 987–1001 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)/(180*3600000)
    real(dp),parameter :: deg2rad=acos(-1.0_dp)/(180)
    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, 事实证明这是一个好习惯.

[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.

规范 19 *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 `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.

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)
```

[5] 如果摸透了编译器会怎么决定的话, 偷个懒不写也罢了.

```
    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)
```

[6] 同脚注5.

[7] 同脚注5.

```
    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, Ifort 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”, 电脑屏幕上就出现 `Hello, world!` 了. 要是问我 `write(*,*)` 和 `print *,` 有什么区别, 我的回答是没有区别…

```
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

[8]这就实现了数字型和字符型的相互转换.

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.

```
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 20 *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 Ifort 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, Ifort 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
```

[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.

```
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110页有个轮子, Ifort 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, Ifort 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110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
    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104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

[10] Ifort 编译时会出现一个 “remark”, 是些建议, 我们选择无视, 不过听从 Ifort 的建议也是好事.

[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…

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...

```
program main
    use iso_fortran_env, only: dp => real64
    implicit none
    integer,parameter :: sam_len=1000
    real(dp),parameter :: mas2rad=acos(-1.0_dp)/(180*3600000)
    real(dp),parameter :: deg2rad=acos(-1.0_dp)/(180)
    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,F9.4,849X,F15.11)") p(i), b(i)
    end do
    close(10)
    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`, `849X`, `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 849 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 ... Explanations
  ------------------------...------------------------------
```

```
  ...     ....   ...          ......
  1-  28  A28    ---    ... Unique source designation ...
  ...     ....   ...          ......
129- 137  F9.4   mas    ... ? Parallax (parallax)
  ...     ....   ...          ......
987-1001 F15.11 deg    ... 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 987 个字符开始, 所以要再跳过 $987 - 138 = 849$ 个字符, 写上 `849X`, 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. 给同志们留一个看完本章后的作业: 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 Ifort 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
```

有时我们会碰到一些奇奇怪怪的数: $+\infty$, $-\infty$, 和 NaN. $+\infty$ 和 $-\infty$ 好理解, NaN 名为 “非数”, 表示 “Not a Number”, 遇到什么不定式呀多值函数呀结果连 $\pm\infty$ 都没法表示的时候结果就是 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
    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)
```

[12] 虽然 Fortran 官方规则中没写明, 但想来是这样的.

```
    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
    print "(E13.4E4)", 1e10_qp   ! output:  0.1000E+0011
    print "(E13.4E4)", 1e100_qp  ! output:  0.1000E+0101
    print "(E13.4E4)", 1e1000_qp ! output:  0.1000E+1001
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, 但俺手里的 Gfortran 不认这个编辑符, 它 out 了!

```
program main
    use iso_fortran_env, only: qp => real128
    implicit none
    print "(E13.4E0)", 1e10_qp   ! output:    0.1000E+11
    print "(E13.4E0)", 1e100_qp  ! output:   0.1000E+101
    print "(E13.4E0)", 1e1000_qp ! output:  0.1000E+1001
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
    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')
```

[13] 同脚注12

```
    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')
    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
```

[14]这里是左补空格, 字符串赋值 (5.1节) 是右补空格.

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')
    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当即暴动!

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.

[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.

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 ! ***
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
```

[16]"S" 代表 "sign". "S" 代表 "suppress", "P" 代表 "plus".

```
    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,/,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.

[17] "R" 代表 "round". "U" 代表 "up", "D" 代表 "down", "Z" 代表 "zero", "C" 代表 "compatible".

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
```