

7-函数

子程序 subroutine

概述

主程序在程序开始就自动会执行，而子程序则不会自动执行，它需要被 `call` 命令调用才会执行

子程序中包含可执行的程序代码，就类似于主程序的整体框架。

声明在程序结构中的位置

- 声明的位置应该放在程序代码的可执行命令之前（这点和c++不一样）
- 在程序代码开始出现数值计算和输入输出命令时，就不能再声明变量了

主要结构如下：

```
program main
    implicit none

    声明变量

    可执行命令（赋值、计算、输入输出等）
end program main
```

子程序框架

含有子程序的代码框架为：

```
program main
    ...
    call sub()
    ...
end program main

subroutine sub()
    ...
    return
end subroutine sub
```

或者将子程序嵌套在主程序中

```

program main
  ...
  call sub()
  ...
  contains
  subroutine sub()
    ...
    return
  end subroutine sub
end program main

```

子程序特点

子程序可以在主程序中的任意位置被调用，在fortran90中，支持自己调用自己（递归）

子程序**独立拥有**属于自己的变量声明和行代码

在调用子程序时，可以同时传递一些变量数据过去，让它处理 ---- 传递参数

👉 注意：fortran在传递参数时，使用的是**传地址调用**，也就是说，调用时传递出去的参数和子程序中接受的参数，会使用相同的内存地址来记录数据。

选择语言: Fortran (GFortran 9.2.0)
源代码
☐ 自动运行

```

1  program ex0804
2    implicit none
3    integer :: a = 1
4    integer :: b = 2
5    integer :: ans = 0
6    call add(a, b, ans)
7    print "('ans=',i1)",ans
8    stop
9  end
10
11 subroutine add(first, second, res)
12   implicit none
13   integer :: first, second, res
14   write(*,"(4x,i1)") first + second
15   res = first + second
16   return
17 end

```

命令行参数:

标准输入:

运行结果:

标准输出:
3
ans=3

会进行改变，因为传递的是地址

自定义函数 function

- 自定义函数和子程序很类似
 - 经过调用才能执行
 - 可以独立声明变量
 - 参数传递
- 两者不同:
 - 调用自定义函数前要先声明
 - 自定义函数执行后会返回一个数值
 - 声明时建议使用 **external**，表明其是一个可调用的函数。

示例：

```

program main
  implicit none
  real :: a = 1
  real :: b = 2
  real,external :: add_first !声明外部函数，必须说明类型，可以省略external
  real,external :: add_second

  print *,add_first(a,b) !调用外部函数，不需要使用call命令
  print *,add_second(a,b)
  stop
end program main

```

!第一种实现函数的方式：直接把函数名作为返回结果

```

function add_first(a,b)
  implicit none
  real :: a,b
  real :: add_first !必须在此处再次声明类型，充当返回值
  add_first = a + b
  return
end

```

!第二种实现函数的方式：再定义一个返回值

```

function add_second(a,b) result(c)
  implicit none
  real :: a,b
  real :: c
  c = a + b
  return
end

```

SHELL

```

3.00000000
3.00000000

```

注意：函数的返回值类型的声明可以写在函数的最开头，与function写在一起 比如 **real function add(a,b)**

```

!第一种实现函数的方式：直接把函数名作为返回结果
real function add_first(a,b)
  implicit none
  real :: a,b
  !real :: add_first !必须在此处再次声明类型，充当返回值
  add_first = a + b
  return
end

!第二种实现函数的方式：再定义一个返回值
real function add_second(a,b) result(c)
  implicit none
  real :: a,b
  c = a + b
  return
end

```

☆☆：使用函数时，传递给函数的参数只读取它的值，不要去改变它的数据，因为函数的参数传递也是通过地址进行的，如果改变数据，会使的主程序中的参数数据发生变化。

如果函数很短，并且只会在主程序或者某一函数中使用，可以直接把函数定义写在主程序里

```

program main
  implicit none
  real :: a = 1
  real :: b
  real add
  add(a,b) = a+b
  write(*,*) add(a,3.0)
  stop
end

```

子程序 vs. 函数

相同点

- 经过调用才能执行
- 可以独立声明变量
- 需要传递参数

不同点

子程序

- 没有返回值
- 通过变量将结果传回。
- 输入输出更灵活，可以输出多个变量，方便实现数组的输出。

函数

- 有且只有一个返回值，通常为一个数值，也可以实现数组的输出，但不能输出多变量。
- 写法上格式更灵活。可以直接参与计算。

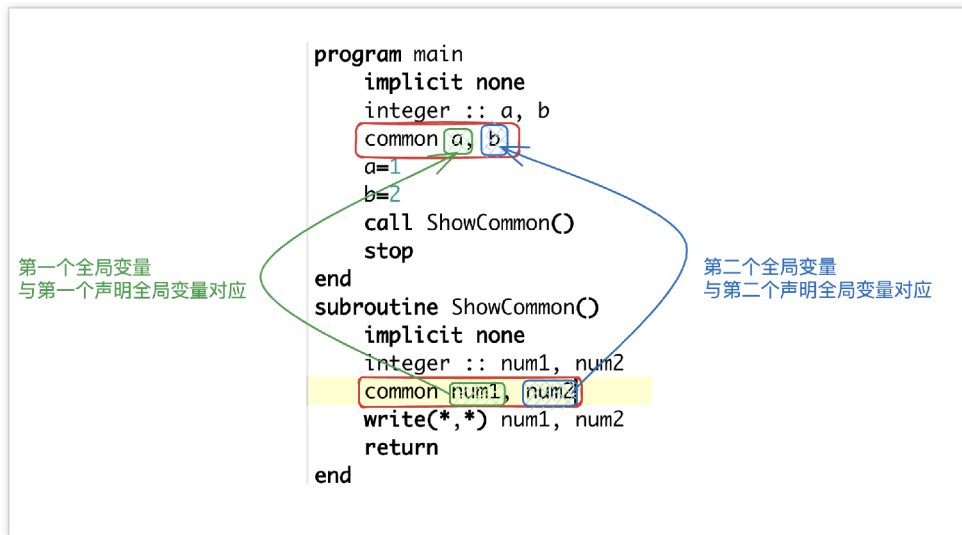
全局变量 common

common的使用

common：定义一块共享的内存空间，实现不同程序和函数之间传递参数

全局变量可以在程序中的任何一个部分被使用，采用地址对应的方法来实现数据的共享。

示例：



类似的，当全局变量较多时，可以把变量进行分类管理，将其放在彼此独立的common区间中

```
program ex0811
  implicit none
  integer :: a, b
  common /group1/ a
  common /group2/ b
  a=1
  b=2
  call ShowGroup1()
  call ShowGroup2()
  stop
end
subroutine ShowGroup1()
  implicit none
  integer :: num1
  common /group1/ num1
  write(*,*) num1
  return
end
subroutine ShowGroup2()
  implicit none
  integer :: num1
  common /group2/ num1
  write(*,*) num1
  return
end
```

block data

由于common变量不能直接在子程序或主程序中使用data命令来设置其初始值，为了解决这一问题，规定可以将common变量放到block data模块中进行data赋值。

block data：

- 类似于子程序，是一段独立的程序模块，拥有自己的变量声明（不能声明常量）
- 只能设置全局变量的值，不能有其它执行命令的出现。
- 不需要调用就可以自己执行，在主程序执行前就会生效

例如：

```
program main
  implicit none
  integer :: a,b
  common a,b
  integer :: c,d
  common /group1/ c,d
  integer :: e,f
  common /group2/ e,f
  write(*,"(6I4)") a,b,c,d,e,f
  stop
end
block data
  implicit none
  integer a,b
  common a,b
  data a,b /1,2/
  integer c,d
  common /group1/ c,d
  data c,d /3,4/
  integer e,f
  common /group2/ e,f
  data e,f /5,6/
end block data
```

执行结果如下：

```
program main
  implicit none
  integer :: a,b
  common a,b
  integer :: c,d
  common /group1/ c,d
  integer :: e,f
  common /group2/ e,f
  write(*,"(6I4)") a,b,c,d,e,f
  stop
end
block data
  implicit none
  integer a,b
  common a,b
  data a,b /1,2/
  integer c,d
  common /group1/ c,d
  data c,d /3,4/
  integer e,f
  common /group2/ e,f
  data e,f /5,6/
end block data
```

命令行参数：

标准输入：

运行结果：

标准输出：

1 2 3 4 5 6

函数中的变量

传递参数的注意事项

🔊 注意：fortran在传递参数时，使用的是**传地址调用**，也就是说，调用时传递出去的参数和子程序中接受的参数，会使用相同的内存地址来记录数据。

☆☆：使用函数时，传递给函数的参数只读取它的值，不要去改变它的数据，因为函数的参数传递也是通过地址进行的，如果改变数据，会使的主程序中的参数数据发生变化。

数组参数

数组参数 == 字符串参数

在传递数组参数时，实际上是传递数组元素当中的**某个内存地址**

类似于c++中的&引用

变量的生存周期

在声明中使用 **save** 可以增加变量的生存周期，保留住所保存的数据

```

program main
  implicit none
  call sub()
  call sub()
  call sub()
  stop
end program
subroutine sub()
  implicit none
  integer :: count = 1
  save count ! 在子程序执行完后, count的值不会消失, 而是一直存在
  write(*,*) count
  count = count+1
  return
end

```

SHELL

```

1
2
3

```

传递参数

- 传递参数可以是数字, 字符等数据, 可以是函数名称或者子程序

传入函数

```

program main
  implicit none
  real, external :: func ! 声明func是一个用户自定义函数
  real, intrinsic :: sin ! 声明sin是库函数
  call ExecFunc(func)
  call ExecFunc(sin)
  stop
end program

subroutine ExecFunc(f)
  implicit none
  real, external :: f
  write(*,*) f(3.1415926/2)
  return
end

real function func(num)
  implicit none
  real :: num
  func = num*2
  return
end function

```

传入函数的名称

通过函数名调用函数

命令行参数:

标准输入:

运行结果:

标准输出:

```

3.14159250
1.00000000

```


传入子程序

<pre>program main implicit none external sub1, sub2 ! 声明sub1和sub2是子程序的名称 call sub(sub1) ! 把子程序sub1当参数传出去 call sub(sub2) ! 把子程序sub2当参数传出去 stop end program subroutine sub(sub_name) implicit none external sub_name ! 声明传入的是子程序 call sub_name() ! 调用子程序 return end subroutine subroutine sub1() implicit none write(*,*) "sub1" end subroutine subroutine sub2() implicit none write(*,*) "sub2" end subroutine</pre>	<p>命令行参数:</p> <input type="text"/>
	<p>标准输入:</p> <input type="text"/>
	<p>运行结果:</p> <p>标准输出:</p> <pre>sub1 sub2</pre>

调用子程序

根据传入的不同子程序名称，调用不同的子程序

注意：上面的声明external，不可以省略，因为这是要声明子程序的名称，从而当做参数去传递的。

特殊参数的使用方法

设置参数属性 (intent)

- **intent(in)** : 输入变量，不可以改变其值
- **intent(out)** : 输出变量
- **intent(inout)** : 既可以输入也可以输出

不指定参数属性不会影响程序执行的结果，但是可以避免编写程序的错误

函数的使用接口 (interface)

interface 是一段程序模块，用来描述调用函数的参数类型及返回值类型等的 **使用接口**

类似于使用手册，这样就不用直接定义了

```

interface
    function func_name
        ! 里面只能说明参数或返回值类型
        implicit none
        real.....
        integer.....
    end [function [func_name]]

    subroutine sub_name
        implicit none
        integer.....
    end [subroutine [sub_name]]
end interface

```

例如：

```

program main
    implicit none
    interface
        function random10(lbnd, ubnd)
            implicit none
            real :: lbnd, ubnd
            real :: random10(10)
        end function
    end interface
    real :: a(10)
    call random_seed() ! 使用随机函数前调用
    a = random10(1.0, 10.0)
    write(*,"(10F6.2)") a
end

function random10(lbnd, ubnd)
    implicit none
    real :: lbnd, ubnd
    real :: db
    real :: random10(10)
    real t
    integer i
    db = ubnd - lbnd
    do i=1,10
        call random_number(t)
        random10(i) = lbnd + db * t
    end do
    return
end

```

命令行参数：

标准输入：

运行结果：

标准输出：

6.64 7.05 4.51 1.82 9.52 2.92 6.88 9.18 3.52 1.57

不定个数的参数传递

可以用 **optional** 命令来表示某些参数是可以省略的

要调用这类不定数目参数的函数时，一定要先声明出函数的 **interface**

函数 **present** 可以检查一个参数是否传递过来，返回值是逻辑型变量

```

program main
  implicit none
  interface
    subroutine sub(a,b) ! 定义子程序的接口，从而使用optional
      implicit none
      integer :: a
      integer, optional :: b
    end subroutine sub
  end interface
  call sub(1)
  call sub(2,3)
stop
end program main

subroutine sub(a,b)
  implicit none
  integer :: a
  integer, optional :: b
  write(*, "( '是否传入b参数: ',L1)" ) present(b)
  if ( present(b) ) then
    write(*, "( 'a=',I1, ' b=',I1)" ) a, b
  else
    write(*, "( 'a=',I1, ' b=unknown' )" ) a
  end if
  return
end subroutine sub

```

命令行参数:

标准输入:

运行结果:

标准输出:

是否传入b参数: F
a=1 b=unknown
是否传入b参数: T
a=2 b=3

改变参数传递位置的方法

前提：一定要声明 **interface**

可以根据变量名称来传递参数

例如：

```

call sub(b=2,c=3,a=1)

subroutine sub(a,b,c)
  ...
end

```

封装函数和参数

contians

contains所包含的函数只能被使用contains的主程序或者函数调用，之外的函数无法调用

```
program main
  implicit none
  call sub1()
  call sub2()
contains
  subroutine sub2()
    implicit none
    print*, 'This is sub2'
    call sub1()
  end subroutine sub2
end program

subroutine sub1()
  implicit none
  print*, 'This is sub1'
end subroutine sub1
```

可以被使用contains的主程序进行调用

命令行参数:

标准输入:

运行结果:
标准输出:
This is sub1
This is sub2
This is sub1

```
program main
  implicit none
  call sub1()
  call sub2()
contains
  subroutine sub2()
    implicit none
    print*, 'This is sub2'
  end subroutine sub2
end program

subroutine sub1()
  implicit none
  print*, 'This is sub1'
  call sub2()
end subroutine sub1
```

不可以被之外的函数进行调用

命令行参数:

标准输入:

运行结果:
编译错误:
/usr/bin/ld: /tmp/ccQFr7Fp.o: in function `sub1':
main.f90:(.text+0x71): undefined reference to `sub2_'
collect2: error: ld returned 1 exit status

module

概述

module 用来封装变量和函数，要配合contains来封装函数

module 可以被任何主程序或函数进行调用，命令为：**use 模块名**

在use声明后，其中的函数就可以被使用。

```
module test
  contains
  subroutine sub1()
    implicit none
    print*, 'This is sub1'
  end subroutine sub1
end module test

program main
  use test
  implicit none
  call sub1()
  call sub2()
  contains
    subroutine sub2()
      implicit none
      print*, 'This is sub2'
    end subroutine sub2
end program
```

注意👉：如果是在同一个.f90文件中，定义主程序和module，一定要把**module**定义在**main**之前，程序才能够正常去运行。

public和private

module里面的数据和函数，可以通过public或private命令，设置成公开或私密

没有特殊说明时，默认函数或数据都是public的

```
module bank
  implicit none
  private money ! 变量声明为私有
  public LoadMoney, SaveMoney, Report ! 函数声明为公有
  integer :: money = 1000000
  contains
    subroutine LoadMoney(num)
      implicit none
      integer :: num
      money=money-num
      return
    end subroutine
    subroutine SaveMoney(num)
      implicit none
      integer :: num
      money=money+num
      return
    end subroutine
    subroutine Report()
      implicit none
      write (*, "( '银行目前库存为: ', I9, '元' )") money
    end subroutine
end module

program main
  use bank
  implicit none
  call LoadMoney(100)
  call SaveMoney(1000)
  call Report()
  stop
end
```

命令行参数:

标准输入:

运行结果:

标准输出:

银行目前库存为: 1000900元

注意事项

- Module不是必须写在最前面，但必须写在use这个module的程序单元的前面
- 如果所有代码写在同一个源代码文件中，则module应该写在前面
- 如果代码写在多个源代码文件中，确保module要先于use这个module的程序单元进行编译
- Module中不能直接书写执行语句（比如read、write），所有执行语句都要写在contains 下面的子程序或自定义函数中

递归函数

递归函数每次被调用执行时，函数中所声明的局部变量（不是传递的参数，没有save的变量）都会使用不同的内存地址，也就是说，每次调用时都是独立存在的。

```
recursive integer function fact(n) result(ans)
```

- 开头以recursive来表示可以递归，被自己调用
- 要使用result来返回参数，传递结果
- 递归调用时，要明确递归结束的条件

- 不使用recursive还可以使用间接递归，来实现递归
 - 即在函数中先去调用别的函数，再经过那个函数来调用自己

比如：通过递归来计算n的阶乘

```
program main
  implicit none
  integer :: n
  integer, external :: fact
  read(*,*) n
  write(*, "(I2,'! = ',I2)" ) n, fact(n)
  stop
end
recursive integer function fact(n) result(ans) ! recursive 表示可以递归
  implicit none
  integer, intent(in) :: n
  if ( n < 0 ) then
    ans = -1
    return
  else if ( n <= 1 ) then
    ans = 1
    return
  end if
  ans = n * fact(n-1)
  return
end
```

使用多个文件

将程序代码分散到不同文件中

优点如下：

1. 独立文件中的函数，方便多人协同工作，方便移植，可以把部分文件给其它程序使用
2. 可以加快编译速度，修改其中一个文件时，编译器只需要重新编译这个文件后再链接即可，不需要编译全部文件

include

include 用来在程序代码中插入另一个文件的内容

include 命令可以写在任何地方，通常应用在声明变量处

```
! file : main.f90
program main
    implicit none
    include 'test.inc' !插入test.inc的内容
    a=1
    b=2
    call sub()
    stop
end
subroutine sub()
    implicit none
    include 'test.inc' !插入test.inc的内容
    write(*,*) a,b
    return
end
```

```
! file : test.inc
integer a,b
common a,b
```