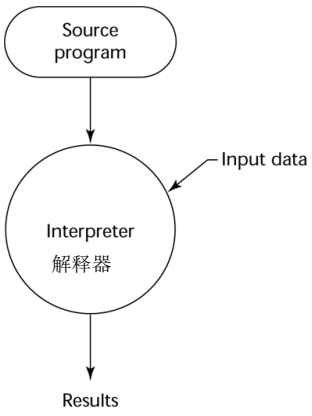
第一章

1. 语言的评价标准（Language Evaluation Criteria）：
   A，可读性(最重要的评价标准):
   因素：1，整体简单性　2，正交性　3，控制语句　4，数据类型和结构　5，语法考虑
   B，可写性
   因素：1，简单性和正交性　2，支持抽象（子程序、二叉树） 3，表现性（Expressivity）
   C，可靠性
   因素：1，类型检查　2，异常处理　3，可读性和可写性
   D，代价
   种类：1，编写程序　2，可靠性　3，维护计划
2. 可读性中的正交性(Orthogonality)的意思：使用该语言中一组相对少量的基本结构，经过相对少的结合步骤，可以构成该语言的控制结构和数据结构。而且他的基本结构的任何组合都是合法的和有意义的！
3. 语言的分类（Language Categories）:命令式语言、函数式语言、逻辑语言、面向对象语言
4. 影响语言设计的因素：计算机体系结构和程序设计方法学
5. 影响冯诺依曼体系的速度瓶颈的因素：内存和 CPU 之间的存取速度
6，纯解释执行语言瓶颈：语句解码（statement decoding）



第二章

6. 第一个高级语言：Fortran
7. 第一个具有结构化和接口化的语言：Algol 60
8. 第一个面向对象的语言：SIMULA 67
9. 美国国防部耗资巨大创建的语言：Ada
10. 第一个纯面向对象的语言：Smalltalk
11. 两个早期的动态语言(Dynamic Language):APL、SNOBOL

第五章

12. (名字部分)关键字和保留字的定义和区别:
    关键字(key word)：是程序设计语言中的一种字，它只在特定的上下文里是特殊的。如 Fortran 中的 real 就是关键字，当在 real apple 中表示一种类型，是个声明语句；而在 real=3.4 这样的赋值语句中代表变量名。
    保留字(reserved word)：是程序设计语言中的特殊字，它不能用作名字。
    保留字比关键字优越，因为重新定义关键字的功能会导致混淆。
13. (名字部分)预定义名字(Predefine names)：介于保留字和用户预定义的名字之间，他们有预定义的含义，但用户可以重新定义。
14. (变量部分)变量(variables)是内存空间的抽象
15. (变量部分)变量的六个部分：名字、地址、值、类型、生命周期、范围！(不是所有的变量都有名字，如匿名类)
16. (变量部分)变量在静态作用域中和动态作用域中的引用顺序：
    静态作用域：自己——>父亲
    动态作用域：自己——>调用者
17. 绑定的概念(The Concept of Binding)：A,左值是变量的地址，B 右值是变量的值

C,绑定时一种关联（属性和实体，操作与符号）

18. 静态绑定：第一次是发生在运行之前，并且执行过程中保持不变
19. 动态绑定：第一次发生在运行时或程序在运行时可以改变
20. 显式声明：是程序中的一条说明语句，列出一批变量名并指明这些变量的特定类型
21. 隐式声明：是一默认的声明机制，来声明特定的类型

    <声明 declaration:仅声明而不会分配内存空间 >

    <定义 definition: 是分配内存的>
22. 生存期（lifetime）：是该变量被绑定与某一特定的存储地址的时间
23. 静态变量（Static Variable）:在程序的运行开始前绑定知道结束时候，仍然保持相同的绑定

    <高校，灵活性差>
24. 动态绑定（Dynamic Binding）:灵活性好，有额外的开销
25. 类型检查：是确保操作符飞操作数为相容类型
26. 相容类型有两条：①要么是合法的操作符，②要么就是在一定条件下进行的隐式转换
27. 强化类型：只要某个程序语言总能够发现其程序中的类型错误，那么这种的语言为强类型的
28. (变量部分)变量的三周等价：

    (1)按名字类型等价：定义于同一个声明之中或者定义于使用相同类型名的声明之中。按名字类型等价具有一定的局限性，例如一个整数子范围类型的变量不会与一个整数类型的变量等价，如：

    type Indextype is 1…100;
    count:Integer;
    index:Indextype；

    这里的 count 和 index 就不是按名字等价的！

    (2)按结构类型等价：两个变量具有等价的类型，如果他们的类型具有相同的结构。

    这种等价方式更加灵活，也更难实现。两种特殊情况：

    派生类型不等价：type a is new Float,type b is new Float.中 a 和 b 就不等价！

    子类型等价：subtype Small_type is Integer ranger 0…99;中 Small_type 和 Integer 等价

    (3)按定义等价:两个类型具有完全相同的定义，则两者是按定义等价的！

```
Var x: complex
    z: record
            re: int
            im:int
       end
    u: rational
    v: record
            report:int
            denominator:1-29999
       end
```

    x,y 是按定义等价，但按名不等价

    n,v 安定义不等价

29. Scope (作用域)

    Static Scope 静态作用域：可以静态的决定变量的作用域，即在执行之前决定的变量的作用域

```
program example;
    var a, b : integer;
    ……
    procedure subl;
        var x, y : integer;
        begin   { subl }
          …..   <-------------------------- 1
        end;   { subl }
```

```
procedure sub2;
    var x : integer;
    ……
    procedure sub3;
        var x : integer;
        begin   { sub3 }
          ……                                  2
        end;   { sub3 }          begin   { sub2 }
    ……..                                      3
  end;   { sub2 }
begin   { example }
……                                            4
end.   { example }
```

Point	Referencing Environment
x and y of subl, a and b of example	
x of sub3, (x of sub2 is hidden), a and b of example	
x of sub2, a and b of example	
4	a and b of example

Dynamic Scope 动态作用域：这种作用域只可以在运行时得以确定，基于调用序列的程序单元，而不是他们的文本布局

Function call: Main---> sub2----> sub1

```
void subl ( ) {
      int a, b;
      …….                                     1
    }   /* end of subl */
void sub2() {
   int b, c;
   ……                                         2
   subl();
   }   /* end of sub2 */
void main() {
   int c, d;
   ……                                         3
   sub2 ( );
}   /* end of main */
```

Point	Referencing Environment
1	a and b of sub1, c of sub2, d of main, , (c of main and b of sub2 are hidden)
2	b and c of sub2, d of main, (c of main is hidden)
3	c and d of main

第六章：数据类型

30. 指针类型(Pointer Type):就是变量拥有一系列的包含内存地址和特殊值，NIL 的值
31. 指针的作用:①提供间接地址，②提供动态内存管理
32. 悬挂指针：定义、怎么形成、危害：

    定义：悬挂指针式一个包含了已解除分配的堆动态变量地址的指针。

    产生：在语言中的如下操作会产生悬挂指针：

    1. 设指针 p1 指向一个新的堆动态变量
    2. 给指针 p2 赋予 p1 的值
    3. 将 p1 所指向的堆动态变量显示的回收并将 p1 设为 null，这时不会改变 p2，p2 将成为悬挂指针；若回收操作不改变 p1，则 p1 和 p2 都将成为悬挂指针！

    危害：1，悬挂指针指向的位置可能已经被重新分配给一个新的堆动态变量。2，新堆动态变量的值就会被破坏。3，存储管理的失败！

第七章：表达式与赋值语句

33. 函数副作用：

定义:当函数改变它的一个参数或者一个全局变量时，就会产生函数的副作用，从而就产生了表达式从左向右和从右向左计算结果不一致的现象！**为了避免一般采用从左到右的计算顺序**，例如 Java 语言！

例子：表达式 a+fun(a)

如果 fun 在 a 变更时没有副作用，那么操作数 a 和 fun(a)的求值顺序不会影响结果，而如果 fun 在 a 变更时有副作用，那么 a 和 fun(a)的求值顺序将会影响结果!像下面的程序中就有副作用！

```
int a=5;
inf fun(){
      a=17;
      return 3;
}
void  main(){
      a=a+fun();
}
```

Answer：
From left→right
8=5+3=a a=5,fun()=3,result=8
From right→left
20=17+3=a a=5,fun()=3,result=20

34. 在在表达式中出现的函数调用

可能出现“副作用”，即可能存在同一个赋值语句中的同名量的值不相同，所以在串行流程中必须规定计算次序。

Example
i, j : integer;
A, B : array[1..100] of integer;
Function f (x:integer):integer;
  begin i:=i+1;
         j:=j+2;
         f:=x+1
  end

From left→right
例： i:=3; j:=0;
A[i]:=B[f(j)+i]:=i+f(j)+i*j
(1)求 A[i]的位置，i=3；
(2)求 B[f(j)+i]的位置, j=0, f(0)=0+1=1,
同时有，i=3+1=4, **j=0+2=2(副作用)**,
即 f(j)+i =5， B[f(j)+i] =B[5]，
(3)表达式的值为：4+f(2)+(4+1)*(2+2)=27,
在求 f(2)时 i=4+1=5, **j=2+2=4(副作用)**
(4)将 27 赋给 B[5]
(5)将 27 赋给 A[3]

35. 操作符重载：考试方式不明了!!!

① 操作符重载的危害与好处：可以有多重用途，但可能影响可读性和可靠性！

② 一些支持抽象数据类型的语言，例如 Ada、c++、fortran 95 及 C#允许程序员使用操作符重载！

第九章：子程序

36. 通用子程序的特点：

① 每个子程序都有一个入口点

② 在被调用子程序的执行期间，调用程序单位被停止执行，意味着在任何定的时刻只有一个子程序在运行

③ 子程序在执行结束时候，总是将控制返回到调用程序

37. 过程子程序和函数子程序：

① 过程：定义参数化语句的序列，通过单个调用语句来启动这些计算，过程实际上定义了新的语句，例如 Ada 中不存在排序语句，用户就可以建立一个过程对数据数组进行排序，这个排序语句就是

过程。

② 函数：在结构上模仿了过程，在语义上模仿了数学中的函数。函数大多有副作用。

38. 函数结构：结构类似于过程，包括返回值、函数名、参数值、函数体!
39. 子程序定义：子程序定义描述的是子程序的接口以及子程序的抽象行为。
40. 子程序调用：是显式的要求执行子程序。如果一个子程序之后他被开始执行，但是还没有完成这种执行，我们称这个子程序是活跃的
41. 子程序的首部有几个目的：①它说明下面的语法单位是关于某个特殊子程序的定义，②首部给子程序提供一个名称，③首部可以通过可选的方式来说明一列参数

首部例子：Fortran：subroutine Adder(parameters)

Ada: procedure Adder(parameters)

Python: def Adder(parameters)

Ada 和 Fortran 子程序的首部不会出现特殊字

C ： void Adder(parameters)

42. 参数：子程序获取参数有两种方式：①直接通过非局部变量②通过参数传递（参数传递是十分灵活的，比第一种方法，大量使用非局部变量，降低可靠性）

子程序的首部被称为形式参数，只有当子程序被调用时，它们才能与存储空间相绑定

子程序的调用语句必须包含子程序的名称，以及一组将与子程序中的形参相绑定的参数（实参）

43. 子程序中的几种参数及其定义：
    ① 形式参数：函数定义中使用的形参
    ② 实质参数：函数调用中传递的参数
    ③ 位置参数：实参和形参的绑定是简单的按位置进行的，第一个实参和第一个形参对应，这样的一些参数被称为位置参数!
    ④ 关键字参数:将一个与实参相绑定的形参的名称与这个实参在一起说明的参数就是关键字参数，主要是为了应对当参数表很长时，程序人员容易在表中实参的次序上犯错误！例如：

sumer(length=my_length,list=my_list,sum=my_sum)

其中 length ,list,sum 是形参！

44. 子程序中的几种变量及其定义：
    ① 局部变量：定义于子程序内部的变量，作用域限制在子程序中！局部变量可以是静态的也可以是栈动态的！静态的是直接存取的，具有较高的效率！而栈动态的是运行时自动绑定的，有一定的灵活性！

    栈动态局部变量的缺点：这种在存储空间分配，初始化以及变量解除分配都有时间上的代价；对这种栈动态变量的存取必须是间接地；栈动态局部变量的子程序不是历史敏感的
    ② 非局部变量：
    ③ 全局变量
45. 参数传递方法：
    ① 按值传递：实参的值用来初始化对应参数的值，这种实现通常为数值的复制。

缺点是：给形参额外的存储空间；参数很大时，存储空间和复制操作具有很高的代价

    ② 按结果传递：用于输出参数的一种实现模式，并没有值传给子程序，相应的形参的行为就如同一个局部变量；但是在将控制返回到调用程序之前，形参值被传递给调用程序的实参；也需要存储空间和复制操作
    ③ 按值-结果传递：实参的值在子程序的入口处复制给形参，在结束时形参值传回到实参。

缺点是：需要额外的存储空间和数字复制的时间；实参赋值次序的问题

    ④ 按引用传递：传递的是地址（path），仅一个地址到达了子程序

有点：高效

缺点：比按值传递慢，因为需要额外的间接寻址；如果只要求对被调用子程序的单项传递，可能会在实参口产生一些不易察觉但是错误的改变；可能产生别名错误

⑤ 按名传递：对于子程序中的所有情形，实参实际上都是以文本形式代替了与他对应的形参

46. 重载子程序

定义：重载子程序是与另一个相同引用环境中的子程序具有相同的名称的子程序。每个重载子程序的版本必须只具有一个协议；也就是说，它必须与子程序的其他版本在参数的数目、参数的顺序。或者参数的类型上不相同；如果他是一个函数的话，则也可以是返回的类型不相同！

47. 访问非局部变量

非局部变量：在子程序中是可见的，但不是在局部的声明的

局部变量：是所有程序单元中都是可见的

48. 静态作用域：一个好的程序结构是被子程序的访问权限和其他子程序变量指定的，而不是在设计精良的解决方案

动态作用域：静态类型检查不能引用非局部变量，因为静态判断一个变量的引用为非局部的声明是不可能的

49. 分别编译：在不同的时间，都能对编译单元编译，如果是访问或使用其他实体，他们的编译是不独立的

50. 静态编译和独立编译

静态编译是将所用的子程序编入到 exe 文件中，体积小，容易移植；

独立编译的程序比较大，不易打包移植！

独立编译最主要的特征是：在独立编译单元之间的接口不被进行类型的一致检查

第十章

51. 调用子程序的步骤：（简单调用语义要求）***（必考）***

① 保留程序单位当前的执行状态

② 传递参数

③ 将返回地址传递给被调用的程序

④ 将控制转移给被调用的程序

52. 调用子程序返回的步骤：

① 如果是按值或者按结果或按输出型的参数，将这些参数的形参值转移到对应的实参上

② 如果子程序是一个函数，将函数值转移到调用程序可以存取的位置

③ 恢复调用程序的执行状态

④ 将控制转回调用程序

53. Fortran77 包含两个独立的部分

① 实际的子程序代码（是一个常量）

② 局部变量上面列出的数据，执行子程序时可以改变的

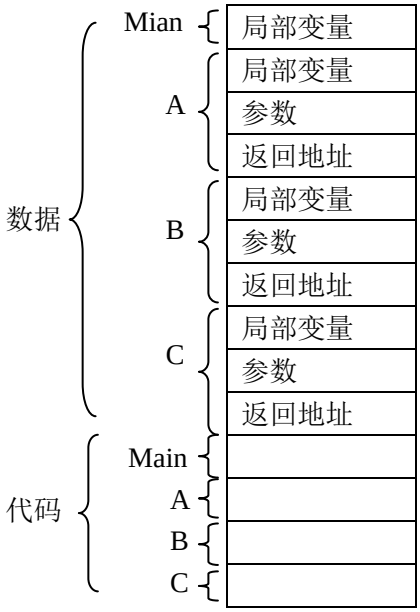
54. 活动记录实例（Actration record instance）

是活动记录的一个具体示例，它是活动记录形式的一组数据

局部变量
参数
返回地址

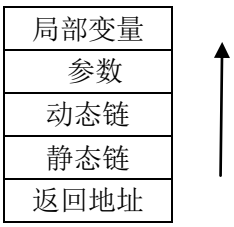
简单子程序的活动记录

下图为一个具有简单子程序的程序代码及活动记录



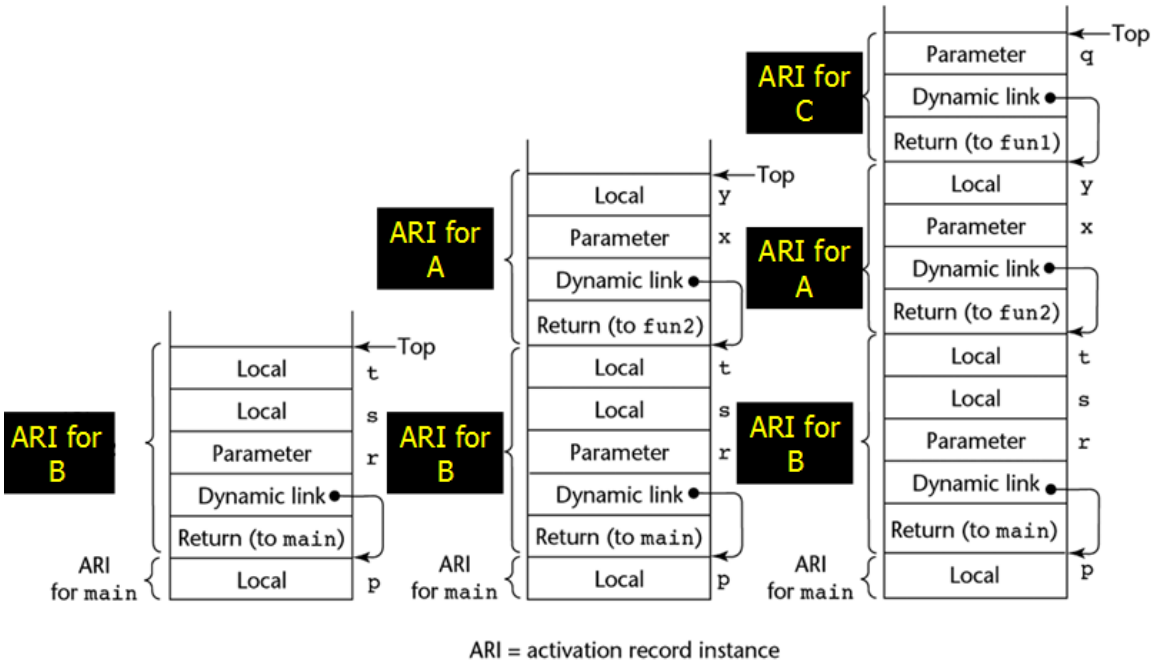
活动记录：子程序中的非代码部分的格式（或布局）
形式是 static 的

55. 调用与返回要求空间存储的信息：
    ① 关于调用程序的状态
    ② 参数
    ③ 返回的地址
56. 动态链接
    ① 动态链接是一个指向调用程序活动记录实例顶端的指针，栈顶被设置为前一个动态链接的值，之所以是动态链接，是应为一个子程序从栈里分配到的空间会超出活动记录的范围。具体的看书吧、、
    ② 动态链：在给定的时刻出现在栈中的一组动态链接被称为动态链！它代表执行时怎么达到当前位置的一个动态经历！
    ③ 局部偏移：活动记录实例的顶部总是表示当前位置，可以在代码中将对局部变量的引用表示为从局部作用于中的活动记录开始的偏移。这种偏移被称为局部偏移
    ④ 实例：factorial 语言中阶乘递归!
57. 静态链接
    ① 静态链：在允许嵌套子程序的语言中，为了实现静态作用域的最常用的方式是静态链，使用这种方式时，将一个称为静态连接的指针添加到活动记录中，它指向静态父辈活动的活动记录的低层。能够使用这种链在静态作用域中实现非局部变量的存取。
    ② 静态链接：有时也被称为静态作用域指针，用来存取非局部变量！添加了静态链接之后活动记录中在参数之前的位置就有了三个元素：返回地址、静态链接、动态链接[从低到高排列]！
    ③ 静态深度：一个与静态作用域相关的整数，它表示一个作用于从最外层作用域开始所嵌套的深度。
    ④ 链偏移：在对变量 x 的非局部引用中，达到正确的活动记录实例所需要的静态链长度，正好是包含 x 的引用过程的静态深度和包含 x 的声明过程的静态深度！
    ⑤ 局部偏移：将局部变量的应用表示为从局部作用域中的活动记录的开始的偏移！
    ⑥ 会画静态链，理解静态链和动态链的区别，会找到链偏移值和局部偏移值！
58. ALGOL 语言中的 activitation record instances 必须被动态创建



返回值：包含指向代码段调用者的指针和调用语句之后那条指令的偏移地址

```
Program MAIN_1;
    Var P ： real;
     procedure A(x:integer);
         var Y : boolean;
         procedure C(Q: boolean);
              begin { C }
                …        ←——— 3
                end;   { C }
           begin {A}
            ….          ←——— 2
            C(Y);
              …
            end;   { A }
procedure B(R:real);
    var S, T : integer;
     begin   { B }
        …      ←——— 1
              A(S);
               …
      end;     { B }
begin   { MAIN_1}
  …
 B(P);
  …
 end.   { MAIN_1}
          MAIN_1 calls   B
B calls A
A calls C
```



59. 非局部变量的两种不同的引用方式：深访问和浅访问

① 深访问：在动态作用域语言中，对非局部变量的而引用可以通过搜索其他当前的子程序的活动记录实例来解决！在这里只是跟踪动态链而不是跟踪静态链，这种在动态程序语言中通过动态链实现引用非局部变量所需要的技术就是深访问！静态链往往出现在具有子程序嵌套的程序中，在没有嵌套的程序中的活动记录实例中往往不需要静态链的！深访问中先是在本程序中找变量，当没有时在沿着动态链挨个查找和变量比对，当找到后就不再往更高层寻找了！

② 浅访问是一种替代的实现方法而不是一种替代语义。深访问的语义和浅访问的语义是相同的。只是在浅访问中并不将子程序中声明的变量存储在这个子程序的活动记录中。因为在动态作用域中，任何一个特定名称的变量在任意给定的时刻最多只能有一个可见的版本，因而是一种非常不同的方式。

③ 实现浅访问的两种方式：栈和中央表格方式！

④ 深访问和浅访问之间的选择：取决于子程序调用以及非局部引用的相对频率。深访问提供了快速的子程序链接，但对非局部变量的引用代价昂贵；浅访问刚好相反!

第十一章：抽象数据类型和封装结构

60. 抽象：是对于实体的一种观念或者实质体的一种表示，它仅仅包括这个实体的最重要的属性
    抽象类型：过程抽象：所有子程序数据抽象
61. 什么是封装：将程序组织成为以系列逻辑上相关联的代码与数据的组合，可以单独的编译其中的每一个组合而不需要编译程序的其余部分，而封装就是这样的一个组合！
62. 数据抽象的动机：对抗复杂性的一种武器，使用管理大型以及复杂程序比较容易的一种方法
63. 什么是抽象数据类型：从语法角度来讲，抽象数据类型是一个封装，它仅仅包括一种特定数据类型的数据表示，以及一些给这种类型提供操作的子程序，通过访问控制，可以将类型的一些不必要的细节对这个封装以外的。使用这种类型的单位隐藏起来!

第十三章：并发

64. 物理并发和逻辑并发：
    ① 物理并发：假设存在多个处理器可供使用，并且来自相同程序的几个程序单位同时的运行！
    ② 逻辑并发：允许程序员和应用软件假设存在着多个处理器提供真实的并发，然而事实上，程序实在单处理器上被分时的执行。
65. 控制线和准并发：
    ① 控制线：就是控制经过程序时到达程序中之点的序列，一条控制线就代表一个处理器，物理并发中有多条控制线，而在逻辑并发中只有一条控制线!
    ② 准并发：具有协同的程序，这种程序有一条控制线。
66. 任务(进程)和子程序之间的区别：
    ① 任务可以隐式的启动，而子程序必须显示的调用！
    ② 当一个程序单位调用一个任务时，他在继续自己的工作之前并不需要等待任务执行完成，而子程序需要等待！
    ③ 当完成一个任务的执行时，控制可能会也可能不会返回到启动任务执行的程序单位，而子程序一定会将控制返回！
67. 任务之间通信所依靠的是什么？
    ① 信号量
    ② 管程
    ③ 消息传递
68. 合作同步和竞争同步
    ① 合作同步：当任务 A 继续他的执行之前，如果必须等待任务 B 完后某种特定活动，这时在任务 A 和任务 B 之间就需要合作同步！在合作同步问题中，因为要想正确的使用缓冲区，共享数据的使用者就必须合作！
    ② 竞争同步：当两个任务都需要不可能同时使用的某种资源时，这两个任务之间就需要竞争同步！竞争同步防止两个任务在同一时间同时存取一个共享的数据结构，这种情况可能会破坏共享数据的正确性，要提供竞争同步，就必须要保证共享数据的互斥访问！
69. 任务的五个状态：
    ① 新生状态
    ② 可运行状态
    ③ 运行状态
    ④ 阻塞状态
    ⑤ 死亡状态

70. Ada 里的会合
    ① 会合：如果任务 A 需要发送一个消息给任务 B，并且任务 B 愿意接收，消息就能被传递过来，这种实际的传输被称为会合，会合只能够发生在发送者和接受者都希望它发生时候！
    ② Ada 里的会合发生方式：
        A. 接收者任务，task_example 可能正在等待另外一个任务发送消息给入口 entry，当消息被发送时会合就发生了！
        B. 当另外一个任务试图发送一条消息给相同的入口时，接受姐任务可能正忙碌于一个会合，或忙碌于与会合无关的其他的一些处理，这种情况下，发送者就被悬挂起来，知道接受者可以在一个会合中接收那一条消息为止。在没有接收前发送者被放入队中进行等待！

第十四章：异常处理

71. 异常的分类
    ① 继续模型
    ② 终止模型
    ③ 重试模型
72. 比较 Ada、C++、Java 三者异常处理的异同点
    ① 结构上：
        A． Ada 采用 EBNF 的形式：when 异常选择{|异常选择}=>语句序列
        B．C++采用 try catch 块
        C．Java 采用 try catch 块和 finally 语句对 C++结构进行补充
    ② 引发语句上：
        A． Ada 采用 raise[异常名]语句引发
        B．C++和 Java 中采用 throw[表达式]语句引发
    ③ 传播方式上：
        A． Ada 中逐步向调用者上层去寻找，如果没有找到就自动结束
        B．C++和 Java 中也是逐步向调用者上层去寻找，如果没有找到的话就调用默认的处理程序进行处理！
    ④ 匹配方式上：
        A． Ada 中采用字符串匹配形式，属于类型匹配
        B．C++和 Java 中除了类型匹配外，还进行最先匹配和最佳匹配方式
    ⑤ java 语言相对于 C++语言的异常处理机制更接近于面向对像语言，而且提供了 java 虚拟机所隐式提出的预定义异常！另外 java 更加接近于 Ada 语言！