程序设计语言原理

Principle of Programming Languages

裘宗燕

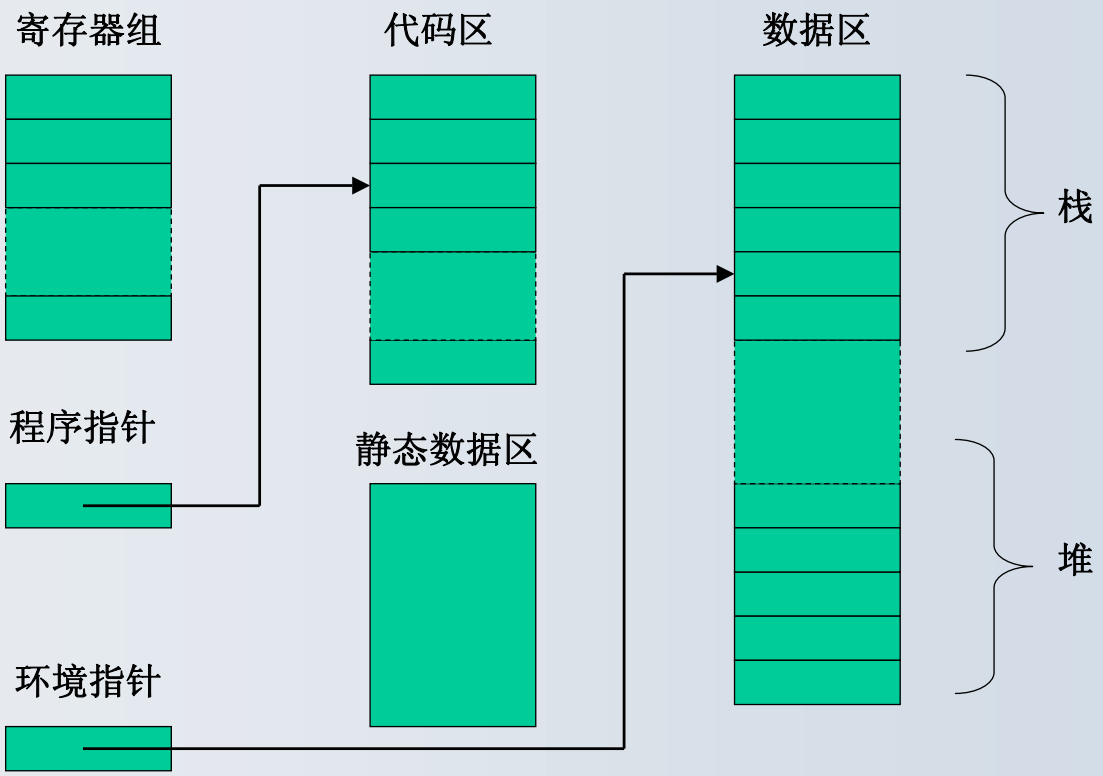
北京大学数学学院

2012.2～2012.6

6. 基本控制抽象

- 子程序抽象
- 子程序活动和局部环境
- 静态实现模型
- 一般实现模型
- 调用序列和在线展开
- 参数机制
- 泛型子程序
- 异常处理
- 其他子程序

机器模型



控制抽象和子程序

抽象是一种手段，用于建立一个名字与一段可能复杂的程序片段的关联，而后可以只考虑名字、功能、使用方式和使用，无须再考虑其实现细节

通常需要区分

- 控制抽象：抽象的对象是实现某种功能的可能复杂的操作
- 数据抽象：抽象的对象是被操作数据概念和相关功能

本章讨论控制抽象，控制抽象的主要目的是定义和实现能良好执行的操作。有关数据抽象的问题是下面两章的内容

子程序是最主要的计算过程抽象机制（子程序抽象）

一个子程序封装一段程序代码并给以命名，

- 可通过子程序名引用这段代码，启动其执行，完成代码所描述的计算
- 建立好的一个子程序可能多次执行，甚至可能同时存在多个不同的执行，甚至可能存在多个同时处于活动状态的执行

子程序抽象

广义的子程序就是指一段代码，可以独立地启动和运行，是最一般的概念，常见的具体子程序概念包括：

- 过程：完成一系列动作的顺序子程序，具有典型的后进先出性质
- 函数：一类过程，其执行的基本目的是算出一个值（可用在表达式里）
- OO 语言里类的方法，是附着在类的实例对象上的子程序
- 协作子程序：允许同时存在多个正在执行中的协程，协程之间采用显式的挂起/唤醒方式转移控制权，每个时刻只有一个协程处于活动状态
- 并行子程序：可同时存在多个执行，由基础支持系统（调度器等）确定控制转移，一个时刻可能有多个并行的活动（进程）

“子程序”通常专指过程和函数。其他子程序在很多基本问题上与之类似

子程序可以看成一种操作扩充机制，用于扩充语言里描述计算的基本词汇表，从而建立起新的描述层次。这是最基本的扩充需要

定义好的子程序就像基本操作，通过这种方式可以分层建立复杂的计算抽象

子程序抽象

子程序的其他重要作用：

- 子程序实现一个局部的计算环境，是一种控制的封装和屏蔽机制
  - 局部环境具有私密性，与外围环境之间有清晰的划分
  - 与外围环境之间有清晰的接口，可通过接口单向或双向传递信息
- 子程序定义一种作用域，通常是“单向透明”的（开作用域）

  外部定义在其内部可见，内部定义在子程序外不可见
- 子程序是一种决定对象生存期的程序单元，子程序里定义的对象在子程序的运行开始时创建，其生存期在子程序的执行结束时都结束
- 过程和函数的后进先出性质，可以利用栈结构方便地管理子程序调用所需的内存，管理方便，效率高
- 其他子程序需要更复杂的运行支持，子程序间控制转移的代价可能更高，资源消耗更大。所以：只要可能，最好是用常规的函数/过程描述计算，不使用更复杂的“子程序”（如协作子程序，并发子程序等）

子程序：参数化

子程序的最直接发展是参数化

- 想法来源于汇编语言的宏
- 参数化使子程序更加通用。实现参数化的方式就是把子程序性质中的某个部分抽取出来，建立反映了这一部分的变化的参数
- 参数化使同一段代码可以作用于不同的实际参数集合
- 实际程序设计对参数有许多不同的需要
  - 人们针对不同需要提出了许多不同的参数机制
  - 参数机制的设计集考虑编程需要，也考虑实现的方便性和效率

子程序的定义、调用、参数化，局部环境的建立和使用等等，都引起复杂的语义问题和实现技术问题

理解子程序带来的这些问题，是理解程序语言的最关键问题之一

语言的子程序结构

不同程序设计语言可能采用不同的子程序结构。常见的：

- 单层结构：一个主程序和位于同一个层次的一组子程序
  - 不允许在子程序里嵌套定义子程序。这种结构最简单
  - 如 **Fortran**。**C** 语言的情况类似，但它没有真正的主程序，只有一个预先定义的执行入口子程序（通常以 **main** 作为它的名字）
- 嵌套结构：一个主程序和在它内部嵌套定义的一些子程序。允许任意深层的嵌套 。（**Algol 60/Pascal** 等等）
- 更复杂的嵌套结构
  - 如 **Ada**，允一组最外层程序单元，可以是子程序，也可以是另外几种结构，包括：包、作业等。可指定其中任何一个作为执行入口
  - 面向对象语言允许在数据描述里嵌套定义子程序
- 许多语言允许子程序内嵌套分程序（复合语句）局部环境
  - 分程序建立起子程序内部的局部定义环境

子程序的定义与执行

理解子程序行为的第一步是分清子程序的定义和执行

- 子程序定义是一种语言结构，其最基本的部分是一个代码体，其中描述该子程序需要执行的动作
- 子程序有一个接口描述，说明它的使用形式，对这个子程序的所有使用都必须符合这个描述的要求。接口描述通常包括：
  - 子程序名和参数/返回值描述
  - 可能包括有关数据传递的约定（类型，采用什么参数模式等）
- 要求子程序执行的语法形式称为子程序调用
  - 执行子程序的调用，将导致子程序代码体的实际执行

一个子程序应该有且只有一个定义，可以有0个或多个调用。需要分清：

- 定义时环境：定义所在的上下文环境
- 执行时环境：调用所在的上下文环境

子程序的定义和执行

- 子程序描述的主要部分是其代码体（子程序体），子程序实现的主要部分就是由其代码体生成的目标代码
- 子程序的接口描述通常只是为编译器、阅读者提供信息，支持其工作
  - 子程序接口本身通常并不生成代码
- 为实现子程序功能，主要就是实现子程序的调用和正确执行，还需要生成一些辅助性代码，主要是两段代码：
  - 前序代码：在子程序本身的代码执行前完成一些准备工作
  - 后序代码：结束前的清扫，为正确返回调用方而做一些的恢复工作

理解子程序执行的意义，需要：

- 为“执行中”的子程序建立一个清晰模型
- 理解子程序执行时的引用环境（包括局部环境）的建立、使用和撤销
- 理解参数的传递机制和返回值机制

子程序定义：界面与实现

子程序的定义分为两个部分：

- 描述子程序与外界关系的界面定义部分
  - 子程序名字
  - 子程序的参数
    - 各个参数的类型
    - 信息传递方式（参数模式，下面有详细讨论）
    - 参数名（以便在子程序体里使用外界传来的信息）
  - 返回值（是否有，如何返回等）
- 描述子程序执行时的局部环境和动作的实现部分
  - 局部定义，定义局部环境里的变量等成分
  - 代码体，定义子程序调用时的动作

子程序活动

当子程序 **P** 被调用时，需要为它建立一个活动，其中包括

- **P** 的代码体（或当时代码执行情况的记录，即当前的代码执行位置）
- 一个表征 **P** 的局部状态的数据结构，称为活动记录（**activity record**）
- 活动记录是根据 **P** 的局部数据做出的安排（布局），其中包括：
  - **P** 的所有参数和局部变量
  - 为支持子程序启动、执行和结束所需的各种辅助性数据结构

- **P** 的每个调用启动时将建立一个新活动记录，该活动结束时销毁这个记录
- 这种活动记录也是数据对象（内部数据对象），因此有
  - 创建和销毁
  - 在子程序运行过程中被使用和修改

子程序活动

- 执行中可能同时存在 **P** 的多个不同活动（如有递归调用的过程）：
  - 每个活动需要有一个独立的活动记录
  - 多个活动可以共享 **P** 的同一代码段（在常规语言里，代码是静态的，不必建立多个副本）
- 不应该说“在子程序 **P** 里执行语句 **S**”，严格说法应该是
  - “在子程序 **P** 的某个具体活动 **R** 里执行语句 **S**”
  - 应区分子程序 **P** 和子程序 **P** 的一个具体活动

子程序 **P** 的一个具体活动的执行状态由以下几部分组成，其中最主要的部分就是 **P** 的这个活动的局部引用环境：

- 代码体的当前执行位置（确定下一条要执行的指令）
- 当时活动记录的状态，主要是 **P** 的所有局部变量的取值情况
- 可能存在的尚未记录到内存（活动记录）里的寄存器状态

子程序执行

为了支持程序的执行，需要两个指针：

- **IP**（指令指针）。它总指向当前活动子程序的代码体里的一个位置（指向将要执行的那条指令），并随着指令的执行而更新
- **EP**（环境指针，**fp**，帧指针）。它指向当前子程序的当前活动记录
  - 由它出发可以确定当前子程序里的所有数据引用
  - 子程序代码体里的大部分数据引用都是基于这个指针得到
    - 局部数据（变量/参数）通过这个指针加上静态确定的偏移量
    - 对静态环境中的变量，代码中可以直接引用
    - 其他非局部引用需要专门支持（后面讨论）
- 随着指令的执行，**IP** 自动更新到新确定的指令位置（下一条/其他位置）
- 代码中的指令基于环境指针更新当前活动记录中的某些数据项（还可能修改静态环境和外围环境），从而实现程序执行中的状态改变

子程序执行

- 程序开始执行前建立主程序的活动记录（静态创建，如有主程序），令 EP 指向该记录，IP 指向主程序代码段的起始位置，程序开始执行
- 在启动（调用）一个子程序时，需要
  - 保存当前子程序活动记录和 IP/EP，还需保存一些有用的寄存器值（子程序恢复执行时还需要恢复和使用的值）
  - 为被调用的子程序创建一个新活动记录，设置 IP/EP 指针（转换环境，转换执行点），而后令控制转入被调用的子程序（按设置好的新 IP 执行），实现执行现场的转移
- 当需要转到一个以前暂停的子程序时，需要恢复原来保存的现场
  - 保存当前子程序的状态（活动记录，有用的寄存器和 IP/EP 指针）
  - 恢复以前保存的寄存器值和 IP/EP 值
  - 按恢复后的 IP 值继续执行

这是最一般的子程序间控制转移和执行模型。过程/函数调用的实现等都是它的特例。下面将讨论

函数/过程的抽象执行模型

子程序之间控制转移的最简单情况是过程/函数的调用/返回（下面讨论中的“子程序”专指它们）。这种情况下子程序之间的控制转移有层次性：

- 调用时，控制转到被调子程序
- 被调子程序结束时，控制转回调用（主调）子程序（后进先出）

处于活动状态（执行中，尚未结束）的子程序可能有许多，在调用一个新子程序时，当时的 IP/EP 值保存在哪里？

对函数和过程，由于后进先出性质，合适的方式是将它们保存在被调子程序活动记录里的某个固定位置，子程序返回时可以方便地取出使用

为了恢复子程序执行的现场，可能需要同时保存另外一些信息

采用活动记录和 IP/EP 指针对，可很好解决子程序调用中的基本问题

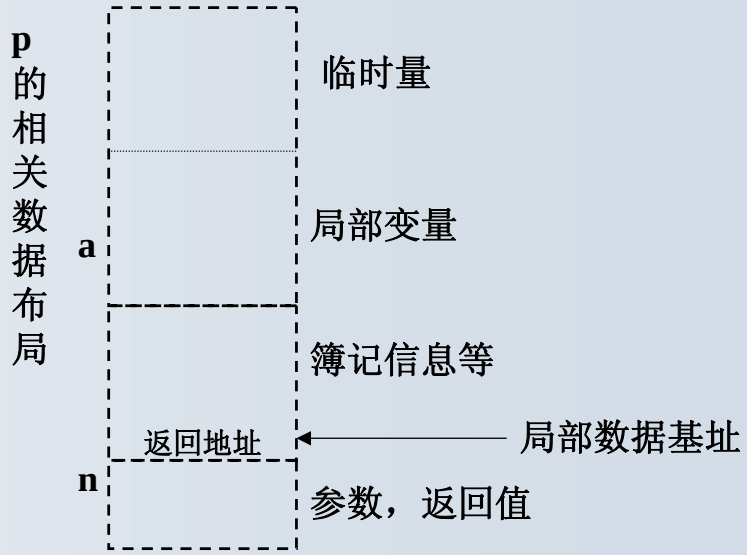
这是抽象的过程/函数调用模型，可以（根据情况）派生出多种具体实现

函数过程的局部数据布局

函数/过程的定义中描述了它执行时的局部环境：

- 函数/过程内部各变量的类型、数组的形状等
- 编译时为子程序 p 里的局部定义做出某种安排，确定局部环境的布局

```
int p (int n) {
  int a[10];
  ...
  if (...) {
    int m; ...
  }
  else {
    int b[2]; ...
  }
  ...
  return a[0];
}
```



局部数据所需的存储量应能静态计算，位置可任意安排

静态模型

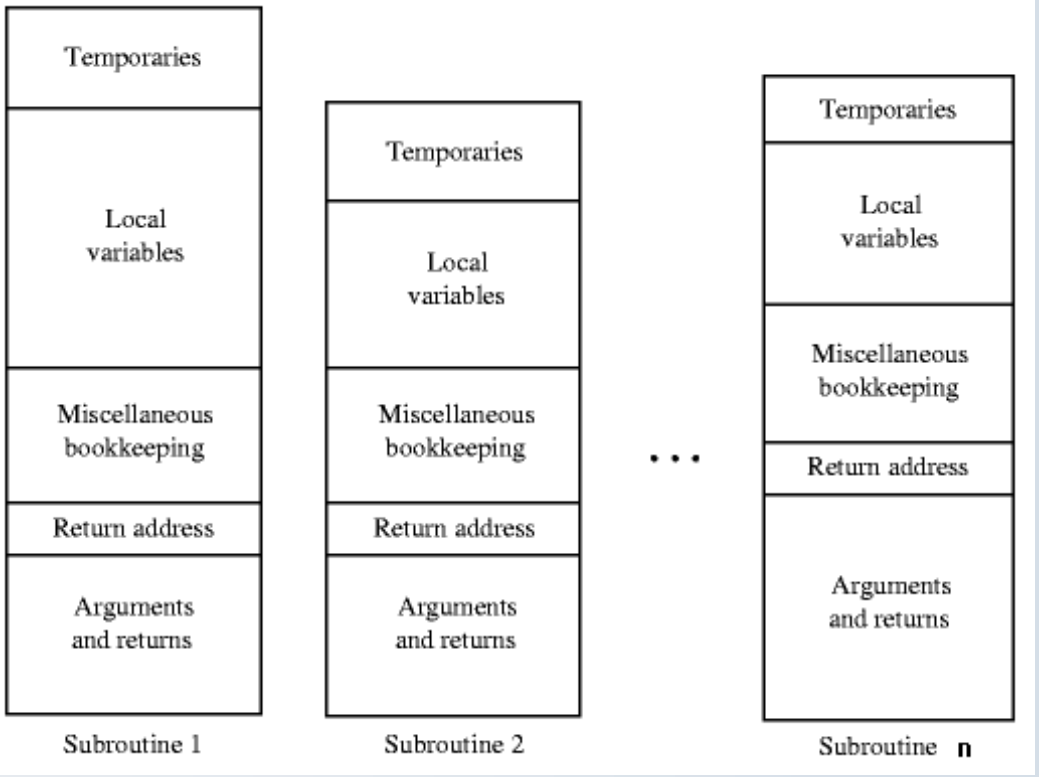
早期的 **Fortran**（**90** 之前）**/Cobol** 等可以采用静态模型实现子程序：

- 运行中任何时刻每个子程序（至多）只有一个活动记录，因此可以在执行前静态建立所有子程序的活动记录（都在静态区分配）
- 子程序代码中都能直接访问活动记录里的对象（不需要**EP**），效率高
- 程序执行时装入代码，将 **IP** 设置到主程序代码的开始后启动执行
- 调用子程序时，将当时的 **IP** 存入被调子程序的活动记录里的固定位置，将 **IP** 设置到被调子程序代码第一条指令，使控制转入子程序代码
- 返回时，由当前子程序活动记录取出保存的 **IP** 值，转入调用程序执行

由于 **Fortran/Cobol** 语言不允许子程序的递归调用，因此每个子程序至多只存在一个活动，因此才能采用简单的静态实现模型

静态模型下的局部和全局数据都可以直接寻址，调用的辅助工作较少（可能需要保存一些寄存器，也可存入被调子程序的活动记录），实现效率高

静态模型



对这类语言，也可以考虑采用后面介绍的技术，动态创建活动记录

如果程序实际执行的子程序比程序里定义的子程序少得多，采用动态建立活动记录的技术可以节约存储

调用返回的一般模型

递归是很有用的编程技术。如果语言支持递归，程序运行中，就可能同时存在一个子程序的多个活动，需要用多个活动记录表示它们。这时就必须采用 **IP/EP** 对的技术，在调用子程序时动态地为其创建新的活动记录

如果调用后进先出，活动记录可用栈分配，称为栈帧，用帧指针 **fp** 作为 **EP**

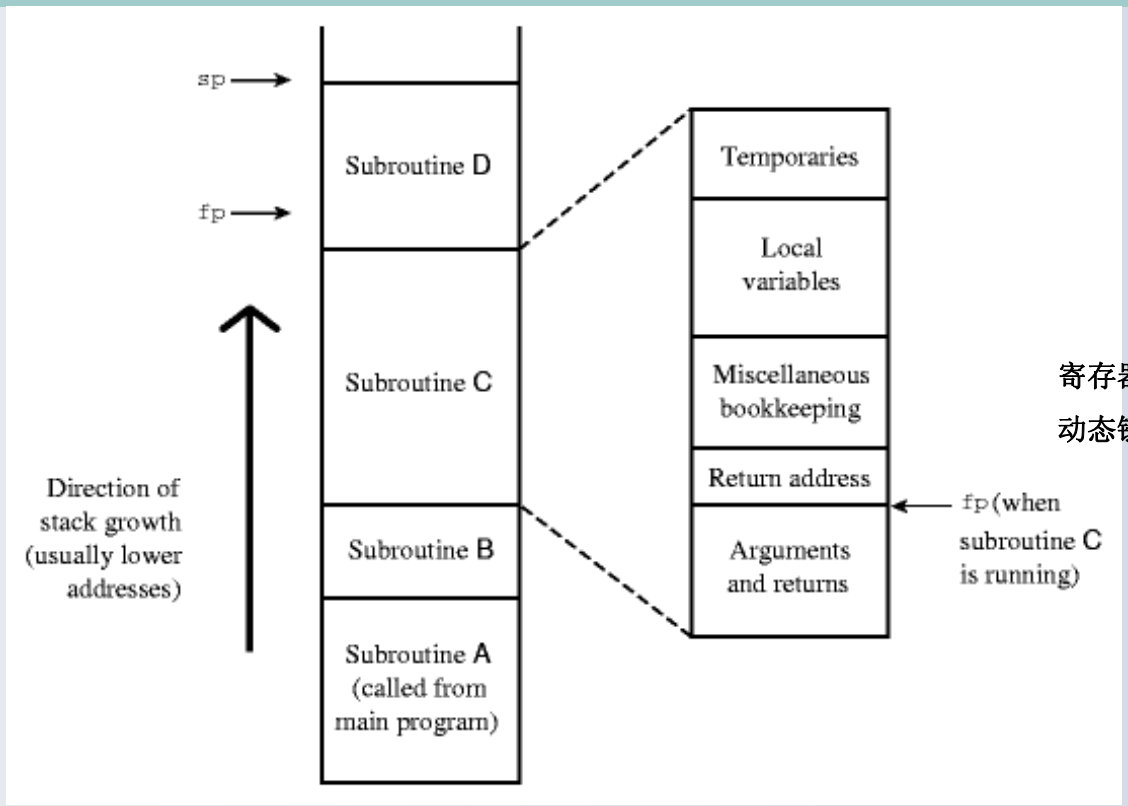
- 程序运行中栈里保存着一些帧（子程序活动记录），每个帧里保存的 **ep** 是栈里下一个子程序帧的地址，形成一条链（称为动态调用链）
- 帧的分配释放表现为栈的压入和弹出

计算机硬件为这种模型提供了支持。许多机器提供了专用的栈寄存器（或者允许用任何寄存器作为栈指针）和专门的栈指令

Fortran 90 里定义可能递归的子程序需加关键字**RECURSIVE**（借自**PL/I**），要求这些子程序的活动记录必须实现为动态创建的栈帧，其余子程序的活动记录可以不入栈。（常见语言里，如 **Pascal/C** 等，所有活动记录都入栈）

Fortran 90 方式可能提高效率，但有危险（递归子程序忘记加关键字）

调用返回的一般模型



子程序：引用环境

子程序运行时的完整引用环境包括：

- 静态环境（可能没有）
  - 全局和静态变量在静态区分配，子程序代码里的全局数据访问都编译为绝对地址寻址，或者相对于静态区基址的偏移量寻址
- 局部环境
  - 按编译时确定的局部环境布局创建栈帧，其中的每个量都在编译时确定在帧里的位置，是基于帧指针（**fp**）值的相对位置
  - 子程序代码里的局部数据访问，都编译为相对于 **fp** 的偏移量寻址
- 非全局外围环境（可能没有，如 **Fortran**、**C** 等）
  - 外围子程序形成的引用环境，需要用更复杂的寻址方式（后面讨论）
  - **OO** 语言方法的调用对象形成的外围环境，通过自动产生的特殊指针参数 **this** 和偏移量访问
  - **… …**

全局环境和静态区

一些语言提供了全局环境，作为所有子程序的环境的一部分

- 全局环境在程序运行前静态建立，在程序执行期间始终存在
- 用一块静态存储区实现全局环境。分配所有全局变量，确定位置和大小
- 其他静态对象也在静态区分配：C 局部静态变量，C++/Java 类静态成员
- 全局对象的初始化在程序运行前完成。一般不允许在这种初始化描述里包含源程序代码，只允许写能静态计算的表达式
- 有些语言允许在初始化中执行代码（例如C++），模糊了静态/动态的界线
- 其他静态变量的初始化与此类似，有些语言有特殊规定
- 静态环境建立后，程序就可开始执行
- 各种语言都规定了程序的执行入口（主程序，或默认/指定的子程序）

Java 没有全局环境，对象都是局部的，但有静态数据对象（类的静态成员）

局部环境的栈实现

子程序 P 的运行环境主要通过 P 的栈帧实现，多数局部对象在 P 的帧里有一个存储位置（相对于帧指针的特定偏移量）；也可能有些变量不分配内存，始终放在寄存器，特别是使用频繁且定义域局部的变量

当 P 的某个活动是当前活动时，指针 fp 指向 P 当时活动的帧，在 P 的代码执行中，所有（分配了存储的）局部变量访问都相对于指针 fp 进行

子程序里的局部分程序（复合语句）环境也在活动记录里分配位置

- 如果有可能只使用寄存器，就可以不在栈帧里为它们分配存储位置
- 如果分配的位置，运行中同样通过 EP 间接访问
- 互不相交的分程序可以共享活动记录里的位置，以节约存储

这种局部分程序没有独立的活动，因此比较容易处理

对于像 C 语言一样采用“平坦”子程序结构的语言，引用环境只有两层，所有全局和局部的访问问题都已经解决了

调用序列

调用序列是编译器生成的代码段，完成子程序调用和退出时的运行栈维护

- 不同机器上的不同语言系统（对不同语言）可能采用不同调用序列
- 子程序调用时，从原程序暂停执行到子程序开始执行之间执行的代码段称为子程序的前序代码
- 子程序完成到原程序恢复执行间执行的代码段称为子程序的后序代码

前序代码和后序代码都可能分为两部分，由调用方和被调方分段完成

- 调用方执行子程序前序代码段1（保存调用方状态的主要部分，进入）
- 被调方执行子程序前序代码段2（保存调用方的必要信息）
- 执行被调子程序的代码
- 被调方执行后序代码段1（恢复调用方的必要信息，转回）
- 调用方执行后续代码段2（恢复调用方状态后继续执行）

具体实现需要精心设计（栈帧的设计、前后序代码设计和功能分配）。相关代码分别放在调用放代码中，被调用方代码的前后

调用序列：平坦结构（例如 C 语言）

- 为返回值分配空间（移sp）
- 计算并压入各个实参值
- 压入返回地址
- 分配帧，保存原 fp 设新 fp
- 保存必要的寄存器
- 执行子程序基本代码
- 计算并存入返回值
- 恢复保存的寄存器
- 恢复 sp（根据fp）
- 弹出子程序帧，恢复 fp
- 调用方继续

前序代码

后序代码