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

3. 对象和环境

- 数据和类型
- 对象：创建、销毁和使用（存在期）
- 变量：命名和性质约束
- 作用域
- 名字分类
- 指针和引用
- 别名
- 其他相关问题

约束时间

约束（**binding**，也称绑定）就是建立联系。约束时间指一个联系的建立时间（有时还要考虑持续期间）。广义指做出一种决策的时间

与高级语言和程序有关的重要时间概念：

- 语言设计时。如，语言的各种结构都是在设计时确定的
- 语言实现时。如，各种数据类型的表示，程序实现方式，内部结构的设计
- 编程时。如，程序员选择算法和数据结构、程序结构
- 编译时。从程序中高级结构到机器代码的映射，数据对象的布局
- 连接时。静态数据对象分配，跨模块对象约束，库对象约束
- 装载时。静态对象的内存定位，虚地址到实地址的映射
- 运行时。自动变量的存储位置，变量的值约束，运行时的其他事项

哪些特征能静态约束，或必须动态约束，不同语言之间差别很大

一般而言，静态约束能提高效率、意义清晰，动态约束提供更大灵活性

数据和类型

程序的工作是处理数据

数据：能在程序与外界间传送（输入/输出），在程序里保存/处理的信息

- 常见的数据形式有字符、整数、实数等
- 还有其他许多不同种类的数据

数据是程序运行中存在的，被动性的实体

程序运行中需要在其内部的不同部分之间传递，被程序处理

在运行的程序与其外部环境之间传递（输入/输出，与环境交互，包括与程序的使用者交互）

计算机里的数据最终都以二进制形式表示（统一性）

- 同样二进制编码可能表示不同意义（不同的解释）
- 把数据划分为一些集合，每个集合里的数据具有相同性质，可以满足类似的处理需要，有助于程序处理。为此引入了类型的概念

数据和类型

为什么需要把数据分为类型？特别是在高级语言里？有很多理由，例如：

- 提供了对被处理数据的一种概念划分

  有助于人理解和思考与数据有关的各方面问题

- 有可能更有效地实现

  不同类型的数据大小可能不同（表示它们需要的存储量不同），支持的操作不同，被操作的方式不同

  划分为类型，可能用于确定操作的可用性和具体的操作方式

- 可能对数据的使用加以控制和检查

  确定应该使用的操作和操作方式，发现使用中的错误

- 可能基于类型及类型之间的关系组织计算过程，组织程序

  这件事的价值初看并不明显

  随着程序的实践越来越输入，这一认识的重要性越来越清晰

类型

对类型的一个简单（朴素）的看法，是将其看作数据的集合

同一个类型的数据，通常：

- 具有相同的属性和统一的使用方式
- 采用统一的表示方式，可能具有同样的编码形式或/和编码长度
- 采用统一的解释方式（从二进制编码到数据的“意义”）
- 对它们可以使用同样一组操作

类型是计算机科学技术里的一个重要概念

- 有许多实践性研究和理论研究
- 下一章将专门讨论程序语言里的类型问题（从实践的角度讨论）
- 前面说过，有关类型的理论研究成果丰富（国内做得很少）

这里只简单提出类型的概念及其在程序设计语言里的意义

对象：创建、销毁和使用

对象指程序运行中“存在”、在运行的环境中真有体现的实体，如变量等

注意：源程序中描述的东西，未必在程序运行时存在。例如

- 类型：在多数语言里，类型在程序运行时并不存在，不能使用
- 常量：可能有不同的情况，整数（大/小）、字符串的情况不同

我们把程序运行中实际存在的实体统称为程序对象（简称对象）

一个对象可能只在程序运行中的某一段时间存在（生存期问题）

为理解程序的意义，需要弄清楚程序中的对象：

什么时候存在，何时创建，如何创建，何时销毁，如何销毁

可以怎样使用

这里的术语对象比“面向对象”中的对象的意义更广泛

OO 里的“对象” 是这里讨论的对象的一个子集（“类的实例”）

对象：生存期

每个程序对象都有其创建和销毁，有生命周期（生存期，**extent**）

一个对象的生存期是程序执行中的一段时间（具体生存期由语言规定）

- 一个对象只在从其创建到销毁的期间中存在
- 创建和销毁可能伴有特定的动作（例：常量创建时需要给定值）

对象的创建（**creation**）可能是

- 静态创建，包括编译时完成，连接时完成，或者装载时完成
- 动态创建，运行中创建

对象的销毁（**destroy**）可能在

- 运行中完成，可能有资源的处理问题
- 程序终止时完成

生存期跨越程序运行期的对象称为持续性（**persistent**，持久）对象

对象：生存期

对象根据其生存期，首先可以分为两大类：

- 静态对象：静态（程序开始执行前）创建，直至程序终止时销毁
- 非静态对象：程序运行中创建和销毁

一些情况：

- **Fortran** 中的所有变量都可以实现为静态对象

  语言设计时有这种考虑，但具体实现是否采用可以选择

- **C** 中的外部变量和局部 **static** 变量是静态对象，自动变量是非静态对象
- 创建的对象可能需要给定一个值（初始化）。但创建和初始化动作有可能同时做或者不同时做，存在不同的想法

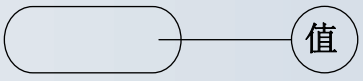
  例如 **C++** 的局部 **static** 变量是静态创建，但在执行首次进入变量所在函数时进行初始化

对象：生存期

存在的（生存的）对象通常有

- 一个固定的标识，通常（可以）用对象的存储位置表示
- 对象总占据着一块存储空间，在其中保存自己的“值”

后面有时用图示表示：



存储位置

对象创建也常常被称为分配（**allocation**）

创建对象可能需要分配资源

最基本的资源是存储，一个对象要在程序执行中存在，程序代码执行中可以找到它，就必须有一个存储位置（最基本的资源占用）

具体对象还可能占用其他资源

例如，一个文件指针对象，可能需要安排一个文件缓冲区

对象：生存期

- 静态创建的对象在程序执行中始终存在，生存期是程序的整个执行过程
  - 例：C语言的全局变量
- 动态创建的对象具有较短的生存期
  - 例：C的局部自动变量，生存期是它的定义所在的函数（或者定义所在的复合语句）的执行期间

程序对象创建时可能需要执行一些动作，销毁时也可能需要执行一些动作

- 例：C 标准函数 **fopen** 创建一个 **FILE** 对象，建立文件链接（执行一些操作系统动作），分配数据缓冲区和附属结构。对相应文件指针执行 **fclose** 操作将销毁这个对象。销毁时做的操作包括：清理缓冲区，释放数据缓冲区的存储，释放文件链接等
- C++ 等一些语言允许程序员为一些对象的创建和销毁定义特定的动作（在定义类时，为该类的对象定义构造函数和析构函数）

生存期：存储分配

创建对象时首先需要为其存储分配。对象分配有三种情况：

- 静态分配：静态确定对象的地址和范围，执行中保持不变
- 栈分配：具有后创建先销毁性质的对象，可以采用栈分配技术
- 堆分配：不能采用上述方式分配的对象，只能在堆里分配

一个程序运行时将拥有一块存储，其存储区域通常分为三部分：静态区，栈区（程序运行栈）和堆区（动态管理区），分别对应上面的三种分配

静态区可能还分为只读区和读写区（依赖于操作系统支持）：

- 代码放在只读区，常量对象也可能被放在只读区
- 静态分配的变量放在静态区里的可读写区

例：C语言不允许修改字符串常量（下面程序是错误的）：

```
char *p = "a string";
*p = 'A'; /*可能导致程序崩溃，字符串常量可能在只读区*/
```

生存期：静态分配

静态分配的对象，其所有性质（包括大小）必须都能静态确定。常见：

- 全局变量（有些语言没有全局变量，如 Java）
- 需要建立对象的常量，包括：
  - 一些全局常变量，字符串和浮点数字面量（可能还有些整数字面量）
- 子程序代码
- 局部静态变量。如C 函数的 static 变量，C++/Java 类的 static 数据成员

子程序代码、常变量、字面量对象有可能被分配在只读存储区，修改它们将导致动态运行错误（相关检查通常需要操作系统支持）

编译系统可能在静态区建立一些为支持程序运行所需的静态数据结构，例如为支持动态检查和执行的表格。如数组维数和上下界，支持越界检查；OO 语言里类的虚表，支持存储管理和异常处理的数据结构等

静态分配实例：Fortran

Fortran 语言的基本设计考虑（对 Fortran 90 前的 Fortran 都适用）：所有对象都可以静态分配，程序运行中可以只有静态区

这一设计是为了保证 Fortran 程序执行的高效率，带来的一些情况：

- 不允许子程序的递归调用，子程序里的局部变量只需要一套
- 静态分配使程序里所有变量访问都可以编译为直接的地址访问
- 程序运行前完成所有对象创建，运行中不做任何对象分配和释放工作

Fortran 不支持许多非常重要的程序设计技术：

- 不能采用递归方式描述和实现算法
- 不支持变长度的数组或字符串
- 没有动态存储分配，不支持动态数据结构（如链表），等等

Fortran 90 加入了递归和动态存储分配，语言的实现模型必须改变

生存期：栈分配

在允许递归的语言里，子程序里定义的局部对象不能静态分配

- 一个局部对象可能同时存在多个活动的副本（因为可以有递归）
- 由于子程序调用有后进先出性质，因此其局部对象可以采用栈分配

栈对象的分配和使用方式：

- 编译时为每个子程序确定栈帧的结构：计算各局部对象所需存储量，在帧里为它们安排位置，确定各对象相对于帧起始位置的偏移量
- 运行进入一个子程序时，为这个子程序的帧在栈里分配空间
- 设置帧指针（通常用一个寄存器）指向当前子程序的帧
- 子程序里使用局部对象都通过帧指针按特定偏移量进行（编译生成的代码里都用这种方式）。利用 CPU 的位移寻址模式可实现有效访问

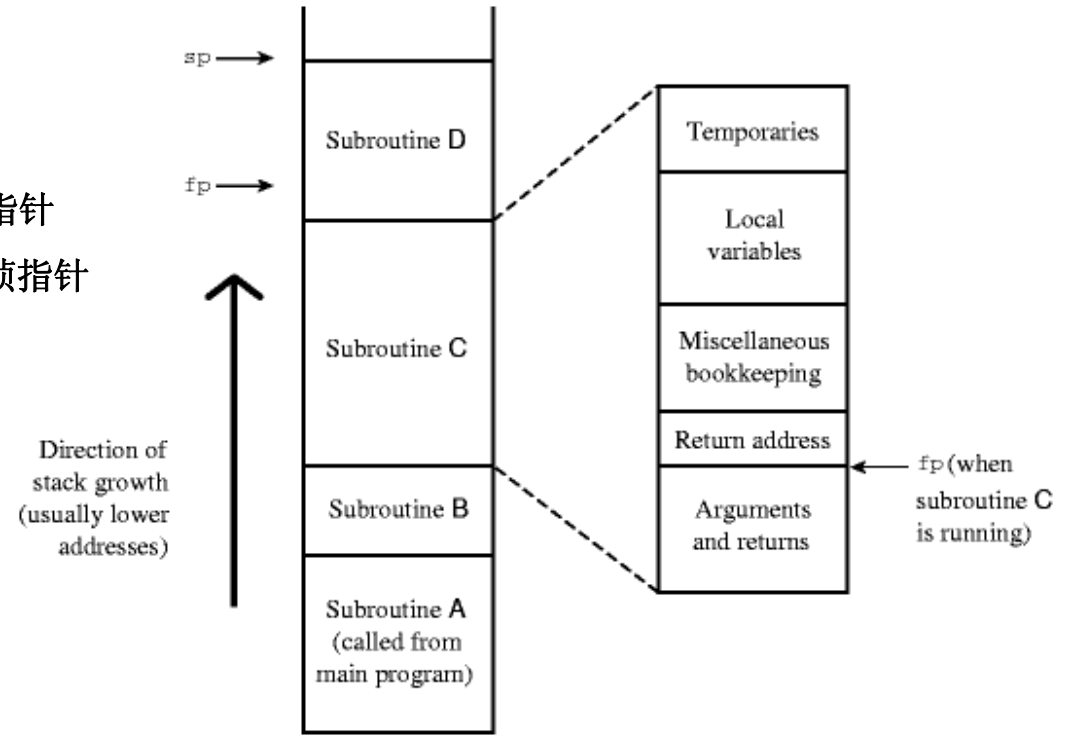
子程序结束时释放对应的栈帧，所有局部对象都销毁（有些语言在销毁前执行一些特定动作。如 C++ 可以定义销毁动作，析构函数）

生存期：栈分配

栈和栈帧的可能布局情况

sp：栈顶指针

fp：当前帧指针



生存期：堆分配

如果对象的生存期有后创建先销毁的性质，就可以在运行栈里分配

没有这种性质的对象只能在堆里分配

堆支持任意顺序的分配和释放，最灵活也最麻烦最容易弄错

堆管理就是平常说的“动态存储管理”

动态存储管理子系统是程序运行系统的一部分，它管理一片存储区

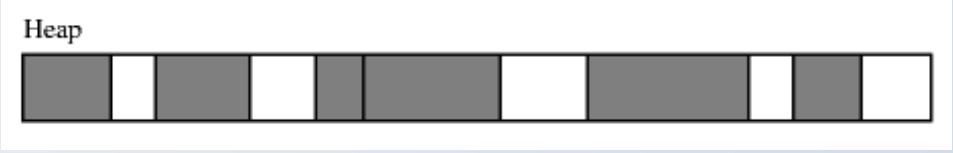
- 如果有存储申请，管理系统就从当前可用的空闲区里分配一块
- 如果无法满足要求，动态存储管理系统可能向操作系统申请新存储块，或者直接返回分配失败的信息
- C 标准库的 **malloc**，**Pascal** 的 **new** 是动态存储管理系统的接口操作

堆的一个场景：

生存期：堆分配

一些语言提供了动态存储块的释放机制，程序员可以根据需要释放以前分配的块。存管系统回收存储块，将其恢复为空闲空间

回收时通常需要合并释放块和相邻空闲块，以便满足以后的大块请求



人工释放的麻烦在于确定正确的释放时刻。在复杂程序里弄清正确时刻对程序员是巨大负担，有时可能很难确定。堆对象的使用错误是复杂程序最常见的一类错误，易出现且很难排除（下面有讨论）

自动释放技术方面，目前主要有两种想法：

- 把堆对象约束于适当的栈对象，借助栈对象的销毁自动释放堆对象
- 设计某种自动机制，自动回收不再有用堆存储块（**garbage collection**）

对象：分类

程序对象可以分为很多类，主要有：

- 变量：有名字，用于保存值的对象。其值在生存期间可以变化
  - 常变量：有名字且有值。在创建时给定值，生存期间不变
- 常量，如建立的字符串等
- 匿名对象：无名对象，通过语言提供的其他手段使用，或隐含使用
  - 通过动态存储分配创建的匿名对象，通过指针或者引用使用
  - 保存中间结果的临时对象，如函数的返回值对象，隐含使用
- 子程序（函数、过程）对象：使用（通常）就是执行其代码
- 程序的内部对象（由具体的语言确定）
  - 为支持程序运行而创建的各种内部数据结构（运行栈，栈帧等）
  - 类型的运行时表示，等等

对象：分类

程序对象可分为简单对象和复合对象

- 简单对象是原子，不能再分割
- 复合对象里包含一些成分对象，这些成分对象有自己的标识，可以独立引用。有些情况下，某些成分对象甚至可能有自己的独立生存期

实例：

- 基本类型的对象通常是原子对象，代码也应看成是原子对象
- 最常见的复合对象是数组和结构（记录），它们的成分有自己的标识，可以独立地访问和使用
- 程序语言通常提供原子对象的字面量描述形式
- 有些语言提供了描述复合对象的值的功能，以便在程序运行中动态构造复合对象（脚本语言、函数式语言通常都提供这方面的功能，**C99** 和 **Ada**等语言也提供了这方面功能）

对象：类型

程序里的对象通常具有固定的类型（有例外）。有类型的变量只能保存特定类型的值。其类型决定了该对象在程序里的使用方式。

- 一个 **integer** 类型的变量占据若干个字节的存储，其中保存它的值，可以在表达式里访问，其值可以通过赋值操作修改（访问和修改都是使用）
- 取两个整型参数返回实型值的函数也是程序对象，其类型可记为

  integer × integer → real

  其值就是它的代码，使用就是执行它的代码。它的类型决定了使用它时需要提供两个整型参数，它返回的是实型值
- 有些语言里存在无类型的对象。如许多脚本语言里的变量没有类型，可以赋给它任何类型的值（**Basic** 语言里的情况类似）
- **Lisp** 等函数式语言里的符号原子（类似于变量）能以任何东西为值，同一符号在不同时刻可以以整数、字符串、函数等为值

声明与定义

名字（标识符）是程序语言里最基本的抽象机制

需要用名字指称程序中出现的各种“东西”：变量、常量、过程/函数、结构/记录的成分、类型等等

名字与事物（包括程序对象）之间的约束关系由定义或声明建立

一个定义或声明引进一个名字，被定义/声明的事物及相关属性就是该名字的意义。该声明/定义建立了名字与相关事物和属性之间的约束关系

例：C 声明 `const double x = 3.876;`

将名字 **x** 约束于一个**double** 类型的变量，**const** 和值 **3.876** 为其属性

例：C 声明 `int fun (int, double);`

使名字 `fun` 约束于函数类型 `int×double →int`。该函数的定义应该具有这种类型，它的使用必须符合这一类型的要求