程序设计语言中的类型系统

2022年春季

张 天

软件工程组

计算机科学与技术系

南京大学

课程思路

- 从类型检查的概念开始理解类型系统的意义
- 然后从标量类型延伸到抽象数据类型
- 然后再延伸到面向对象中的类继承体系
  - 可以介绍动态语言的类型系统的优点，如鸭子类型等

本章要点

- 本章主要讨论编程语言中的类型系统的概念，涉及如下一些具体内容：
  - 类型检查、类型错误、强类型化
  - 抽象数据类型
  - 常见的一些编程语言中的类型系统

类型系统概述

- 数据类型
  - 从计算机的角度来看：数据类型定义了一组值，以及这些数据值上预先定义的操作
  - 计算机通过操作数据产生结果，而数据是对现实问题的一种抽象描述
  - 现实世界中的数据是有类型的，例如人的身高、血压、性别、照片、文字简介等，当我们对这些数据进行元建模的时候，就自然而然考试考虑数据所承载的“类型”的问题
  - 类型是抽象把握的基本要求

计算机中的类型

- 从计算机体系结构的角度来看，更加需要类型的概念
  - 计算机内部，任何信息都是通过0和1的组合来表示的，之所以能够获得有意义的解释，是因为“编码”规则
  - 对于数据而言，其“**类型**”就承载了“**编码**”的设定
  - 信息需要编码，而编码需要规则

编程语言的位置

- 编程语言（Programming Language）处于计算机和现实问题的中间位置
  - 程序是写给人看的，翻译后的机器指令才是给计算机看（执行）的
  - 编程语言是给人设计，用来描述对现实问题的求解方案

  - *注：敏捷开发社区普遍共识，程序才是软件最准确的设计说明书*

编程语言中的类型

- 编程语言中的类型既不是完全对应机器的，也不是完全对应现实问题的
  - 程序是写给人看的，翻译后的机器指令才是给计算机看（执行）的
  - 编程语言是给**人**设计，用来描述对**现实问题**的求解方案，但还必须兼顾到**机器**的特点
  - 因此，一个编程语言中所提供的数据类型和结构在很大程度上决定了其解决现实问题的难易程度
  - *注：敏捷开发社区普遍共识，程序才是软件最准确的设计说明书*

类型理论

- 类型理论是数学、逻辑学、哲学和计算机科学广泛研究的领域，包含众多非常复杂的主题
- 在计算机科学中，类型理论有两个分支
  - 实用分支：主要研究编程语言中的数据类型
  - 理论分支：主要研究类型化的lambda计算
- 数据类型定义了一个**值集合**和针对这些值的一个**操作集合**
  - 程序就是通过操作处理数据，以产生结果
  - **因此，“能否对某个数据操作施加某种操作”是类型检查的本质！**

数据类型的演化

- 在计算机中，数据类型化的概念是历经近60年的演化形成的
  - 最早期的语言中，所有问题空间中的数据结构，都不得不用语言支持的少数几种数据结构来建模
  - **COBOL**开始支持对小数精度的指定，**PL/I**引入大量数据类型
  - **ALGOL68**提供少数**基本类型**，同时提供**结构定义操作符**，允许自定义的数据结构（数据类型设计发展的里程碑）
  - **SIMULA**提出了抽象数据类型ADT的概念，分类类型的外部接口和内部数据表示及其上具体操作的实现体（另一个里程碑）
  - ADT再向前发展一步，就是现在最主流的**面向对象编程**

类型系统的用途

- 编程语言的类型系统有很多重要用途
  - 最实用的就是**错误检查**
    - 编译、程序静态分析等
  - 为程序模块化提供帮助
    - 通过模块之间的类型检查，保证接口的一致性
  - 对程序文档化的支持
    - 数据的类型声明（显式或隐式），提供了关于程序行为的线索
  - 另外，类型系统定义了类型如何与表达式建立联系，包括**类型等价**和**类型兼容**的规则

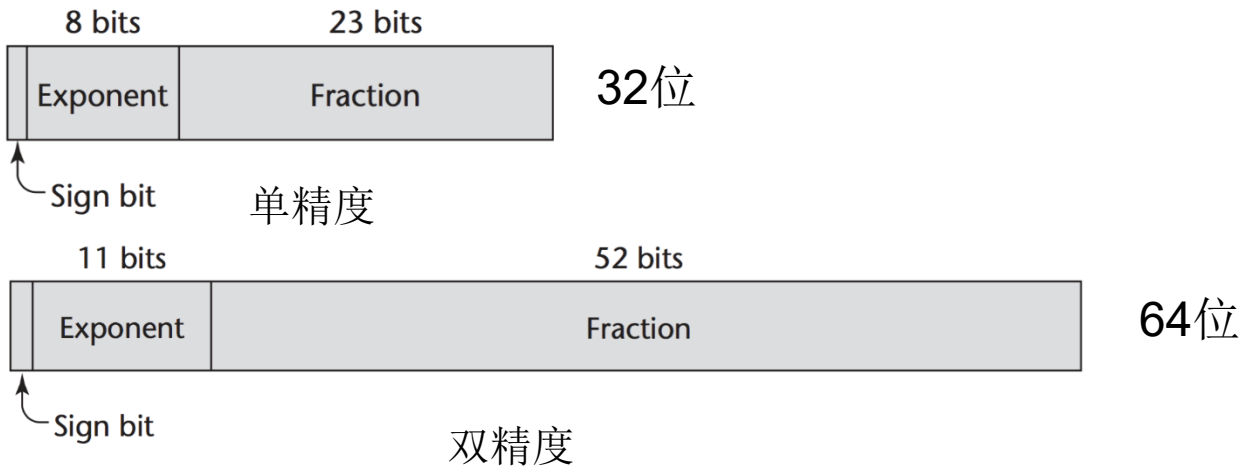
基本数据类型

- 不用其他数据类型来定义的数据类型称为基本数据类型（Primitive Data Types ）
  1. 数值类型
     1) 整数
     2) 浮点数
     3) 小数
     4) 复数
  2. 布尔类型
  3. 字符类型

浮点数的精确度问题

- 浮点数据类型模拟实数，但浮点数表示的只是大多数实数值的近似值
  - 浮点数借鉴科学计数法的形式，用小数和指数两个主要部分组成，硬件可以直接支持，目前主流标准是IEEE 754
  - 由于很多十进制小数无法用二进制准确表示，所以浮点数表示10进制的实数，是一种不精确的编码方式（如十进制的0.1等于二进制的0.0001100110011）

IEEE 754
浮点数标准



精确的小数

- 为了解决小数数据的不精确问题，在硬件体系层面上专门提供了对小数的BCD编码方式的支持
- BCD编码
  - 二进制编码的十进制数（Binary Coded Decimal，BCD）是硬件直接支持的，即机器指令中有直接对应（这跟浮点数一样，需要专门的指令来支持）
  - BCD编码是使用4位二进制来编码1位十进制数，相当于16个数来表示10个数，因此能够精确表示小数，但存在空间浪费

标量类型

- 标量类型
  - 简单地说，标量就是指char、int、double和枚举型等数值类型，也往往就是前面提到的基本类型

- 结构化类型
  - 像数组、结构体和联合等将多个标量进行组合的类型，通常称为结构化类型或聚合类型（aggregate）

思考一个例子

- C语言中：**if (str == “abc”)**
  - 将“abc”的内容放到str中，但这个样的 代码为什么不能执行预期的动作呢？
  - 从感觉上，“abc”的字面量和str的内容是一直的了，但这个判断却不会为真
- **答案**
  - 通常的回答：这个表达式不是在比较字符串的内容，他只是在比较指针
  - 其实还可以从另一个角度解释：字符串其实就是char类型的数组，也就是说它不是标量，当然在C里面就不能用==进行比较了

常见的结构化数据类型

1. 字符串类型
2. 用户自定义的序数类型
   - *枚举类型和子范围类型*
3. 数组类型
4. 关联数组
5. 记录类型
6. 元组类型
7. 列表类型
8. 联合类型

特殊的类型

- 除了以上列举的标量类型和结构化类型之外，还有一些比较特殊的类型
  - 不完全的类型
    - 在C中，不完全类型指，函数之外、类型的大小不能确定的类型
  - **指针类型**

不完全类型

- 一个常见的情况：如果两个结构需要相互指向的指针
  - struct a { struct b *bp; };
  - struct b { struct a *ap; };
- 这个时候，可以将其中之一先声明为结构体标记：
  - struct b_tag;
  - struct a { struct b_tag *bp; };　// 顺序很重要，要先声明
  - struct b { struct a *ap; };

由于受先声明后使用的规定而产生这样的限制

指针是个特例

- 指针是以上标量类型和结构化类型之外的一种特殊类型
  - 其实一些书上将指针归为标量类型，还有一些书将指针归为结构化类型
  - 个人感觉指针确实是一个比较特殊的类型，因此将之独立出来
- 指针有两种主要的用途
  - 提供间接寻址的功能
  - 提供管理动态存储空间的方式

指针类型

- “指针”是成熟的编程语言所必须具有的概念
  - 虽然对这句话的解释会有些争议，但这是目前比较共识的观点
  - 然后，不同的编程语言对指针的支持却有很多差异
  - 主要差异在于是否直接开放“**地址**”以及“**地址操作**”的概念

C语言中的指针

- 指针的概念在很多高级程序设计语言中都有，但C中的指针是公认最“难用”的
  - C++因为从设计上保持对C的兼容性，因此指针问题非常类似
- C中的指针具有非常高的灵活性
  - 和Pascal不一样，C是“一线程序员”为了解决自己的问题而创造的编程语言，更强调工程性和实用性，追求汇编级的效率
- C的指针是介绍PL指针概念所无法忽略的

问题所在

- C指针难用的主要原因
  - C语言中和指针相关的语法比较奇怪
  - 数组和指针之间微妙的兼容性
  - C**对内存可见性的开放**较其他程序设计语言更多
    - 其他高级语言中的指针因为屏蔽了很多直接对内存的可见性，因而相对容易
    - 其实C规范中并没有将指针和内存地址直接等价，但实现上确实如此

C是极大化指针作用的高级语言

- 在高级程序设计语言中，C是最接近“底层”的语言，其原因就在于C指针的开放性

- ANSI C中的Rationale专门提到“C的精神”：
  - 请信任程序员
  - 不要阻止程序员去做需要做的工作
  - 保持语言的小巧和简单
  - 为每一种操作只提供一种方法
  - 就算不能保证可移植性，也要追求运行效率

关于C指针的语法

- 混乱的声明
  - int x：这是一个典型的“类型 变量名”的形式，非常易于理解
  - int *x_p：这里似乎声明了一个*x_p的变量，但**实际上是x_p变量**，其类型是“指向int的指针”
- 也可以这样写
  - int* x_p：这样看起来更符合“类型 变量名”的形式，但**当同时声明多个变量时，就可能出问题**
  - int* x_p, x2_p：此处的x2_p其实就是一个int型

数组的写法

- C中的数组和指针关系非常暧昧，连声明都受到影响
  - int arr[10]：这个写法显然也不符合“**类型 变量名**”的形式
  - Java在声明“int的数组”时，进行了改进：int[] arr，这样更符合“类型 变量名”形式
- 因为以上原因，有人评论C的语法是“奇怪而又变态的”

C对指针的定义

- C语言标准中最初介绍指针的文字：
  - 指针类型（pointer type）可由函数类型、对象类型或不完全的类型**派生**，派生指针类型的类型称为**引用类型**。
  - 指针类型描述一个对象，该类对象的值提供对该引用类型实体的引用。由引用类型T派生的指针类型有时称为“（指向）T的指针”。从应用类型构造指针类型的过程称为“指针类型的派生”。这些构造派生类型的方法可以递归地应用。

派生类型

- 指针类型不是单独存在的，它是由其他类型派生出来的
  - 例如，实际上存在的类型是，“**指向int的指针类型**”、“**指向char的指针类型**”等等

- 数组、记录等结构化的类型也是通过其它类型派生出来的

指针的主要问题

- 指针的主要问题有：
  - 悬挂指针（dangling pointer）：指向内存空间已经释放掉的堆动态变量的指针
  - 丢失的堆动态变量：已经分配空间的堆动态变量，但用户程序不能再访问它了，也叫内存泄露。这些变量也称为垃圾变量。

- *关于指针的问题将在后续部分结构变量、绑定等概念再进行更深入的讨论*

PL的内置类型

- 不同的编程语言会选择以上罗列的数据类型中的一个子集，作为语言自身的内置数据类型

- 目前，所有高级语言所提供的内置数据类型都是抽象数据类型ADT
  - ADT的概念将在后面具体介绍

类型检查

- 问题A：
  - 什么是类型检查？什么地方需要检查类型？检查什么？不检查会出现什么问题？
- 问题B：
  - 面向对象中的“类”是不是类型？要不要检查？什么时候检查？检查的时候和整型、实型、布尔型等有没有区别？
- *如果你能轻松回答出问题A，那么基本上可以自己研究B的答案了*

操作数和运算符

- 为了讨论类型检查的概念，需要先一般化**操作数**和**运算符**的概念
  - 赋值运算可以看成是二元运算符，目标变量和表达式是操作数
  - **子程序**可以看成是运算符，它们的参数和返回结果是操作数

类型检查的本质

- 类型检查就是保证**运算符**的**操作数**具有**兼容的类型**的过程
- 兼容的类型是指
  1. 对运算符合法，或者
  2. 在语言规则中允许由编译器生成的代码（或解释器）**隐式**转换为合法的类型（注：这个自动转换称为**强制类型转换**）

强制类型转换

- 强制类型转换是自动发生的，对于程序而言，并没有直接在字面上的反映
  - 第一，这个过程是自动发生，而手工编写代码
    - 例如，在Java程序中用**括号**指明类型转换的结果
  - 第二，这个过程是隐式的，从代码中没有明确的说明
    - 例如，Java中int变量和float变量相加时，int变量会自动强制转换为float类型

- 注意，强制类型转换是一种容易引起错误的机制，但它提供了编写程序的灵活性，所以很多语言仍然支持（如C/C++、 Java等）

类型错误

- **类型检查**就是为了找出会引发类型错误的地方，因此有必要深刻理解什么是类型错误

- 类型错误是运算符用来操作不合适的操作数的情况
  - 注意，运算符发生作用的含义是机器要针对操作数执行具体的操作，如果被操作的数据并不是该运算符能够适用的，那么必然会引起运行时的错误！

发现类型错误

- 类型错误的**发生**一定是在动态运行时
- 但发现类型错误主要在两个阶段
  - 静态编译期间的类型检查
  - 动态运行时的类型检查
- 直观地讲，类型检查的工作就是针对**变量**（即操作数），检查其**类型**，然后判断在当前**运算**的上下文中，该类型是否符合类型系统的规定
- 思考：变量的类型是什么时候可以确定下来的？

变量及其属性

- 变量是语言中一个或一组**存储单元**的抽象
  - 对于某些类型而言，这种抽象和存储单元的特征非常相像，例如整型变量，完全可以由单个或多个存储字节来代表
  - 但多数情况下，这种抽象和硬件存储单元的差异很大
- 对于程序而言，变量可以用一个**属性六元组**来刻画
  - （名字，地址，数值，类型，生存期，作用域）
  - **各个属性值的确定通常称为“绑定”**，而其中变量与其**类型属性值**的绑定是现代编程语言设计上非常重要的一个主题

绑定的概念

- 在程序设计语言的语义中，**绑定**和**绑定时间**是非常重要的概念
  - 简单地说，绑定是**符号**与表示其含义的一些列属性值进行确定的过程（这里抽象到“符号”层面）
  - *它们的值什么时候可以确定下来？有什么时间阶段？*
- 绑定可以发生在**语言设计阶段、语言实现阶段、编译阶段、链接阶段或运行阶段**
  - 如，星号（*）在**语言设计阶段**就常常与乘法操作相绑定；
  - int类型在**语言实现阶段**与可能的取值范围相绑定（与具体平台的OS以及体系结构相关）；
  - 编译期，i变量和声明的类型int相绑定
  - 在程序载入内存时，变量与存储单元相绑定

这些绑定有没有可能发生变化？

示例

- 对于Java赋值语句：**count = count + 5**
  - count的类型在编译期绑定
  - count的可能取值集合在设计编译器的时候绑定
  - 操作符 + 的含义在编译时绑定，在确定操作数的类型之后进行
  - 字面量 5 的内部表示在设计编译器时绑定
  - count的值在这条语句执行时绑定

静态绑定与动态绑定

- 如果绑定是在**运行之前**的一次出现，而且**在整个程序的执行过程中保持不变**，这种绑定就是静态的

- 如果绑定是在**运行期间第一次出现**，或者**能够在程序执行期间发生改变**，这种绑定就是动态的

变量与类型的绑定

- 变量能够在程序中**引用**之前，必须被绑定到一个数据类型上
  - 事实上，对变量的引用多数发生在**运算**操作上（包括子程序调用和赋值等）
- 类型绑定的两个重要方面
  - 一是**指定类型的方式**，如显式或隐式声明
  - 二是绑定发生的**时间**（编译期和运行时）

显式与隐式声明

- 显式声明是程序中的一条**声明语句**，它直接列出变量名并指明其为特定类型
- 隐式声明则是通过默认约定（编译器理解）而不是声明语句将变量与类型相关联
  - 在隐式声明的情况下，变量名在程序中第一次出现就构成了它的隐式声明
  - 注意：并不是说“变量名在程序中第一次出现”的情况都属于隐式声明，动静态语言是在设计时规定好的，在这种情况下，才分为显式和隐式
- *注：通过声明的绑定属于静态绑定*

隐式绑定

- 隐式的变量类型绑定是由语言处理器（**编译器**或者**解释器**）完成的，最简单的是**命名约定**
  - 命名约定使用变量名的语法形式将变量与类型相绑定，例如 Fortran中，I / i 开头的为Integer
  - Go语言中也有类似的命名约定规则

命名约定的改进

- 命名约定的主要问题
  - 由于命名约定容易带来程序错误（如拼写等疏漏造成的错误），因此很多语言是不支持命名约定的
  - 很多现代语言都是将命名约定放在“**编程风格**”中讨论
- 改进方式
  - 通过特殊字符，如@、%和$等进行类型规定（也用于作用域/可见性的规定）

回顾类型错误

- 类型错误就是指运算符作用于了操作类型不合适的操作数之上的情况

- 关于“类型错误发生”的理解
  - 注：这里有个模糊地带，即类型不合适的操作数到底有没有真正被用来运算！
  - 在运算之前的某一时刻，发现了这个问题，并报错了
  - **直到运算都没有发现，结果运算出错了，进而可以引发一系列问题**
  - **注意：以上两种情况都是理解为出错，但我们不希望后者发生！**

动静态检查

- 静态类型检查
  - 在代码的编译期间进行类型检查，称为“静态类型检查”
- 动态类型检查
  - 在程序的运行期间进行类型的检查，称为“动态类型检查”
- 这两种检查并不是互斥的，一种语言可以既有静态类型检查，也有动态类型检查
  - 但动态类型语言往往只运行进行动态类型检查，因为这些语言是动态类型绑定的！
  - 类型检查仍是语言方面具有很强研究性的问题，而且在语言自身的演化上也不断调整

类型绑定

- 对于静态类型的语言，在编译过程中，变量和参数等都会被解析并放到**符号表**中，同时附带其一些列属性，其中类型就是最重要的属性之一

- 但是，对于动态类型的语言而言，直到程序执行的时候，才会通过**运行时环境**（运行时支撑系统）来确定变量和参数的类型，以及在运算前检查操作数的类型
  - 注意，很多动态语言都是解释执行的，或编译加解释执行的（如Java基于虚拟机）

动静态检查的区别

- 在编译期间进行类型检查比在运行时再检查要对提高程序的性能更有帮助
  - 因为这样可以尽早发现错误，而错误改正得越早，所花费的代价也就越少
  - 运行时的性能一直都是各种语言追求的目标，即使是像Python、Ruby这样的动态语言

- 静态类型检查的代价是减少了程序的灵活性，开放给程序员的技巧会少一些
  - 例如，同一个对象在运行时根据需要动态“裁剪”其类型

强类型化

- 强类型化作为一种重要的编程语言设计思想，在20世纪70年代的“**结构化程序设计**”改革中，被广泛认可
- 目前没有对“强类型化”的非常统一的定义，但其思想如下：
  - 无论在静态编译期，还是在动态运行期，如果程序中“操作数的类型和运算不兼容”的问题总能够被检查出来，那么就这种语言为强类型化的语言！

强类型与弱类型

- 类型系统的强度描述了该系统在实施类型约束时的严格程度。
- 弱类型系统会隐式地尝试将值从其实际类型转换为使用该值时期望的类型。

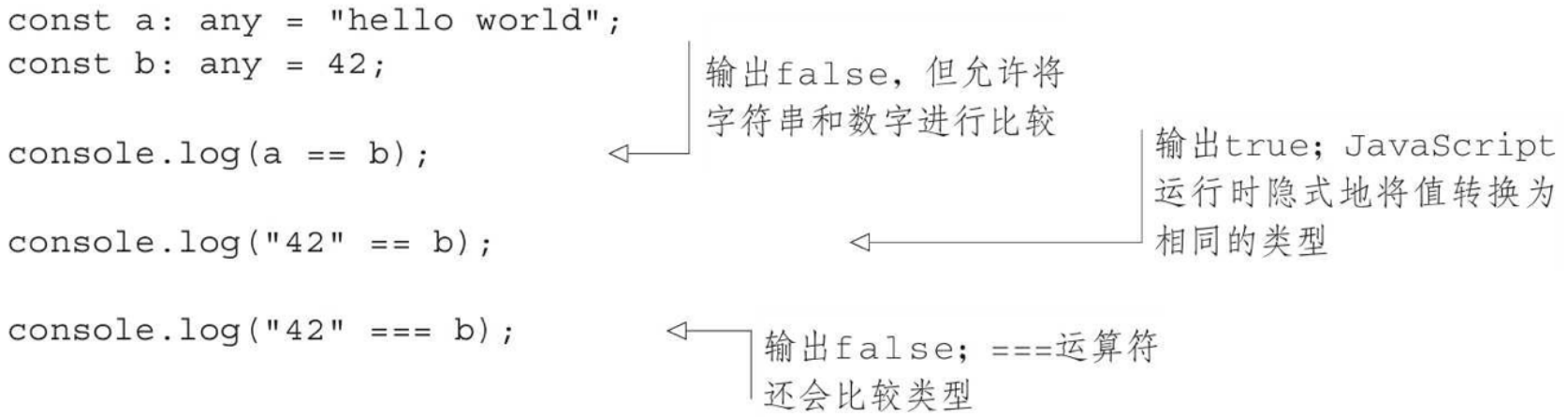
例如，C语言中的隐式类型转换

```
int x = 1.23;  // 1.23是double类型，先隐式转换为int
float y = 66;  // 66是int类型，先隐式转换为float
```

在对变量赋值时，若等号两边的数据类型不同，需要把右边表达式的类型转换为左边变量的类型，这可能会导致数据失真（精度降低），所以隐式类型转换不一定是安全的。

JavaScript的弱类型示例

- 通过在TypeScript中使用any类型并让JavaScript在运行时处理类型，可以看出这一点。
- JavaScript提供了两种相等运算符：==检查两个值是否相等，===检查值是否相等，以及值的类型是否相同

- JavaScript是弱类型的，所以"42" == 42这样的表达式的结果为true
  - 

注意思考：这个结果会让一些人感到意外，因为“42”是文本，而42是数字，而这并不是他们所认为的！

TypeScript的强类型示例

- 在前面的例子中，如果我们将a声明为string，将b声明为number，那么TypeScript将不会编译上面的比较语句

```
const a: string =c"hello world";
const b: number = 42;

console.log(a == b);

console.log("42" == b);

console.log("42" === b);
```

a和b不再声明为any，所以其类型会被检查

这三个比较都将无法编译，因为TypeScript不允许比较不同的类型

整体而言：

- 弱类型系统**更容易使用**，因为它不要求程序员显式转换不同类型的值，但是弱类型系统不能提供强类型系统那样的保证
- 执行的隐式转换越多，该类型系统就越弱

抽象数据类型

- “在最近50年里关于编程方法学和编程语言设计的诸多新思想中，数据抽象是**最深远**的思想之一”

- “**几乎所有**当代语言都支持面向对象编程，而且**几乎所以**不支持面向对象编程和抽象数据类型的语言都逐渐被淘汰了”
  - 注意：这里只用到“几乎”，毕竟像C这样的语言虽然没有严格支持抽象数据类型，但还是依靠UNIX文化的强大力量生存的很好

- *以上摘自《编程语言原理》10E*

数据抽象的出现

- 数据抽象的演变可以追溯到20世纪60年代的COBOL，它提供了**记录**数据结构
  - 记录数据结构存储了字段，字段有名称，其类型可以互不相同
  - 这时候**用户自定义的复杂类型**可以方便地从一些简单类型上构造出来
  - 可以C语言的**结构类型**来理解记录记录，其实**结构**的概念就是在记录的基础上形成的

ADT的语法特点

- 从语法上来看，抽象数据类型（Abstract Data Type，ADT ）是一个包，它包含一种特定数据类型的数据描述，以及在该类型上进行相关操作的子程序
  - 通过访问控制，在此包的外部单元使用该数据时，可以隐藏不必要的类型细节
  - 使用者可以声明该类型的变量，但看不到其实际表示
  - ADT的实例称为对象

- *结合现在常见的过程式风格的语言来体会什么是ADT，会更加明确以上所提到的数据和操作的封装的意义*

ADT实例：浮点型

- 高级语言中的浮点类型具有非常明显的ADT特点
  - 信息隐藏：浮点型数据在存储单元中的实际格式对于用户而言是隐藏的（即看不见）
  - 抽象好的操作：用户所使用的对该类型变量的操作都直接由语言提供，
- 其实，所有的内置数据都是ADT！

用户自定义ADT

- 用户自定义的抽象数据类型应该提供和语言定义的类型（如浮点类型）相同的特性
  - 类型定义：创建一个新的类型，允许声明此类型的变量，同时隐藏该类型对象的表示形式
  - 在该类型对象上的操作集合

ADT的定义

- 抽象数据类型应该满足以下两个条件：
  - 此类型对象的表示方式对于使用它的程序单元来说是隐藏的，因此只有在此类型定义中提供的操作才能直接操作这些对象
  - 把类型声明和在该类型对象上的操作协议（它提供了类型的接口）包含在一个语法单元中。类型接口不依赖对象的表述方式或操作的实现方式，另外，其他程序单元可以创建所定义类型的变量

抽象数据类型的设计

- 语言中用于定义抽象数据类型的机制必须提供语法级的单元，来封装类型的声明和子程序的原型
  - 类型声明和子程序原型必须是可见的，这样才能够声明变量，操作他们的值
  - 类型的表示方式必须是隐藏的
  - 单类型的表示方式和实现操作的子程序定义都可以放在这个语法单元的内部或者外部，只有符合上面的要求即可

- Class Student{
  - ID：int
  - Height：int
  - Public getID(){return this.ID}
  - Public getHeight()
- }
- Class StudentExt extend Student{
  - Score: Float
  - getScore()
  - SetID()
  - Public getID() {return this.ID+100}
- }

继承：复用代码，又可以扩展它

- 多态
- ADT：
  - Student
  - StudentExt
  - 运行时：
    - S：Student
    - Sx: StudentExt
    - S = new Student //地址
    - Sx = new StudentExt
    - S.getID
    - S = Sx
    - S.getID
    - S.getScore

C++的抽象数据类型

- C++于85年首次发布，是向C上加入了面向对象特性创建的，在ADT方面，提供了两个很类似的结构：结构体和类
  - 结构体和类都属于C++类型体系中的一种类型
  - 结构体的设计主要用来针对只包含数据的情况
  - 这里主要讨论C++中类对ADT的支持

封 装

- C++类中定义的数据成为**数据成员**，类中定义的函数（方法）称为**成员函数**
  - 类的成员与类相连：其使用往往和元编程相关联
  - 实例的成员与类实例相连：类的所有实例都共享一套成员函数，但各自拥有自己的数据成员（很多时候我们也称之为**成员变量**）
  - 在讨论ADT的时候，主要是考察类实例的成员

成员函数的定义

- 主要有两种不同的方式来定义类的成员函数
  - 把函数的完整定义都放在类中
  - 函数的声明部分放在类定义的头部，而完整定义放在类定义外部
- 两种方式的差异
  - 完全放在类中可以在编译和链接时，采用**内联函数**的方式来处理
    - 在编译期处理调用某对象的方法时，已经知道了其完整实现体，因此可以优化为内联方式，以提高效率
  - 将成员函数的定义放在类定义的外部，可以将声明与实现相分离，这是现代编程思想的一个常见目标

信息隐藏

- 类封装的隐藏：对于类的客户代码而言是隐藏或可见
  - 隐藏的实体放在private子句下，可见的实体放在public子句下
  - 因此，public子句实际上就描述了该类的接口
- C++在类封装的基础上又进一步划分了隐藏层次（这个跟继承相关）
  - 在继承的上下文中，还有一种可见性protected

关于包封装

- 关于信息封装而言，Ada和Go都提供了包（package）的概念，并可以在这个概念上支持可见性的控制
- 有种观点认为，包也可以支持ADT，因此Ada和Go都是支持ADT的
  - 但如果所定义类型的数据表达形式不是隐藏的，则所定义的类型就不是抽象数据类型
- 特别注意：这里要深刻理解“**隐藏**”的相对性问题，即“**隐藏**”不是绝对的，而是相对于**客户代码**而言的
  - 如果做到了相对客户代码是不可见的，那么不就是ADT的隐藏吗

关于包可见性的讨论

- 就信息隐藏而言，C语言是没有提供这方面的支持，而Go只提供了包（package）封装结构
  - 如果单从定义上来看，包级的封装性显然粒度太大，不能很好支持单个抽象数据类型的定义
  - 但如果考虑到ADT的定义往往单独放在一个包（如标准库），而客户代码是放在其它包里，这样自然就形成了相对的隐藏性
  - 不过，再从另一个方面考虑，如果用户希望自己定义，然后直接使用，是否也必须被迫使用不同的包来分开保存呢？这就涉及到代码风格的问题了
  - 也就是说，**包封装结构对ADT的支持是需要代码风格的协同的**！但话又说回来，类也可以将数据部分设置为public的呀！

对比

- C++等语言通过类结构支持抽象数据类型，而Ada和Go是通过包来支持，两者的表达能力是相似的
  - 它们都可以做到对抽象数据类型的信息隐藏
- 主要区别：
  - 类是一种类型，包则是更加一般化的封装
  - 包结构不仅仅使用在抽象数据类型方面，其实更常用的方式就是命名空间的概念

Java中的ADT

- Java中对ADT的支持与C++相似，区别在于：
  - 在Java中，所有的对象都是通过new在堆中分配空间，并使用引用变量来访问
  - 方法的定义完全在类中，除了专门的“接口”类型
  - Java编译器可以内联未重写的方法
  - 包的概念从逻辑上的命名空间上升到物理文件系统层面
    - 这种更加严格的模块组织管理方式加强了部署上的
    - 但在不同OS上会出现一些影响

关于引用对封装的破坏

- 在Java中，标量类型的变量使用**值拷贝**的方式传递，而对象则使用**引用**的方式传递
  - 此时，会出现由于“指针”别名的问题而产生的封装性破坏

考虑如下Java代码：

```
Data data = new Data(100);
DataRepo dr = new DataRepo(data);
Data d_Ref = dr.getData(); // 用一个引用取出了data
d_Ref.setIntData(200);
System.out.println(d_Ref.getIntData()); //显然是200
System.out.println(data.getIntData()); //100 还是200
```

C中的ADT—背景

- C语言是在72年开发出来的（71年UNIX的汇编版完成）
  - **数据抽象**的演变始于**COBOL60**提出的记录数据结构
  - ADT的思想最早是在**SIMULA67**中提出来的，但没有完整的实现
  - 此时ADT的概念已经在业内得到重视了，Ritchie和Thompson在开发C的时候必然不会忽视它
  - Dijkstra于68年提出“**goto有害**”，72年与Hoare发表**《结构化程序设计》**
  - 68 69年NATO会议上正式提出**软件工程**概念
  - 这个时候**模块化**的概念也非常普遍了
  - C没有概念上的创新，但C是个务实的语言，它支持了这些概念

C对ADT的支持

- 抽象数据类型应该满足以下两个条件：
  - 此类型对象的**表示方式**对于使用它的程序单元来说是**隐藏**的，因此只有在此类型定义中提供的操作才能直接操作这些对象
  - 把类型声明和在该类型对象上的操作协议（它提供了类型的接口）包含在一个语法单元中。**类型接口**不依赖对象的表述方式或操作的实现方式，另外，其他程序单元可以创建所定义类型的变量
- 从**信息隐藏**、**操作与数据的绑定**、**操作接口与实现分离**等角度来深刻理解C语言对ADT的支持

意义

- 理解C对于ADT支持的意义所在：
  - 从面向对象语言来理解ADT，其实很难激发我们对于ADT封装和抽象的深刻理解
  - 而如果尝试从没有面向对象机制的语言来尝试体会如何支持或模拟支持ADT，则可以更加清晰地体会其中的奥妙所在
  - 这是一种从“使用者”换位到“设计者”的思维方式，往往可以激发对本质问题的理解！

模块化

- 模块化设计是“分而治之”思想在软件设计中最早的应用
  - 分而治之的思想其实不是计算机领域创造和特有的，它可以追溯到古代人类处理复杂问题的方式
  - 但计算机领域确实一直将这个思想视为处理问题的一个基本出发点，模块化分析与设计是最早处理软件复杂性的有效手段
  - 随着软件工程的提出和发展，人们对模块化的理解和运用也越来越规范化了
- 对“模块”的共识
  - 模块是可以**分别开发**、**单独编译**并通过**链接**技术形成系统的
  - 为了支持这个目标，模块往往分为两个部分，即**接口**与**实现**

接口与实现

- 接口（interface）
  - 接口规定了模块做什么，是其外部功能契约
  - 接口会声明标识符、类型等，并以源代码的方式提供给客户程序
  - 客户程序就是使用模块的代码
- 实现（implementation）
  - 实现具体定义了如何完成其接口规定的目标
- 接口与实现
  - 对于给定的模块通常只有一个接口，但可以有许多不同的实现
  - 每个实现可以使用不同的数据结构以及操作算法，只要合乎接口的规定即可

耦合性

- 耦合（coupling）
  - 客户程序如果需要依赖实现的特定细节，则形成了两者之间的耦合，即客户程序对实现的依赖性
  - 当实现发生改变时，耦合会导致bug
  - 接口仅规定客户程序可能用到的标识符，而尽可能隐藏不相关的表示细节和算法，就可以有效避免耦合的发生

C中的接口与实现

- 对于信息隐藏和接口与实现相分离，C语言只提供了最低限度的支持
  - 在C语言封装性的基础上，还需要设定一些编程风格上的约定，才能获得接口/实现方法学的大多数好处
  - C语言的接口通过一个头文件指定
  - 客户程序用C预处理器指令#include导入接口
- 这里需要明确C中**作用域**和**链接属性**的概念
  - 文件作用域、代码作用域
  - external和internal

作用域

- 和模块化相关的最重要的两个作用域是
  - **代码块作用域** 和 **文件作用域**
- 代码块作用域（block scope）
  - 位于一对花括号之间的所有语句成为一个代码块
  - 代码块作用域指在该代码块的范围内是可见的
  - 代码块可以嵌套，内层可以用同名标识符来屏蔽外层名字
- 文件作用域（file scope）
  - 任何在所有代码块之外声明的标识符都具有文件作用域，表示这些标识符从声明之处到其所在的源文件结尾处都可以访问

链接属性

- 标识符的**链接属性（linkage）**决定如何处理在不同文件中出现的标识符
  - 标识符的**作用域**与其**链接属性**有关，但这两个属性并不相同
- 内部（internal）链接
  - internal用于描述定义在函数内部的函数参数及变量
- 外部（external）链接
  - 外部变量定义在函数之外，因此可以在多个函数中使用
  - 通过同一个名字对外部变量的所有引用（包括来自独立编译的不同模块）实际上都是引用同一个对象

不透明指针

- 不透明指针主要就是用来在C中实现数据封装的
  - 在头文件中声明不包含任何实现的结构体
  - 在实现文件中定义与数据结构的特定实现配合使用的函数
  - 数据的用户可以看到声明和函数原型的源文件，但实现会被隐藏（如在.c / .obj文件中）

- 注：***不透明指针并不是C语言提供的一种语言机制，而是一种使用语言的编程风格***

- C语言中常使用typedef定义，比如：
  - `typedef struct stack *stack_t;`
- 该定义表示一个指向栈结构的指针，但并没有给出结构的任何信息
  - 这里的stack_t是一个不透明指针类型，客户程序可以自由的操纵这种指针
  - 客户无法反引用，即无法查看指针所指向结构的内部信息，只有接口的实现才有这种特权。

链表示例

- 这里使用一个链表来说明不透明指针的用法
  - 用户使用一个函数来获取链表指针，然后用这个指针来向链表添加信息以及删除信息
  - 链表的内部结构细节和支持函数的实现对于用户不可见
  - 这个结构的唯一可见部分通过头文件提供

- Data声明为void指针，这样运行实现处理任何类型的数据
- **LinkedList*** 这个指针就称为不透明指针
- 客户代码通过这个指针并不能直接操作结构体的数据，而只能使用它来调用其他接口

```
//link.h
typedef void  *Data;
typedef struct  _linkedList LinkedList;

LinkedList*  getLinkedListInstance();
void  removeLinkedListInstance(LinkedList* list);
void addNode(LinkedList*, Data);
Data  removeNode(LinkedList*);
```

- 这是实现的第一部分，定义持有用户数据和下一个链表节点的结构体，接着是 _linkedList 结构体的定义

```
// link.c
#include <stdlib.h>
 #include "link.h"
typedef struct  _node {
        Data* data;
        struct _node* next;
} Node;

struct _linkedList {
        Node* head;
};
```

```
LinkedList* getLinkedListInstance() {
        LinkedList* list = (LinkedList*)malloc(sizeof(LinkedList));
        list->head = NULL;
        return list;
}

void addNode(LinkedList* list, Data data) {
        Node *node = (Node*)malloc(sizeof(Node));
        node->data = data;
        if(list->head == NULL) {
                list->head = node;
                node->next = NULL;
        } else {
                node->next = list->head;
                list->head = node;
        }
}
```

```
void removeLinkedListInstance(LinkedList* list) {
        Node *tmp = list->head;
        while(tmp != NULL) {
                free(tmp->data); // Potential memory leak!
                Node *current = tmp;
                tmp = tmp->next;
                free(current);
        }
        free(list);
}
```

使用该ADT的示例

```
#include "link.h";
    ...
    LinkedList* list = getLinkedListInstance();

    Person *person = (Person*) malloc(sizeof(Person));
    initializePerson(person, "Peter", "Underwood", "Manager", 36);
    addNode(list, person);
    person = (Person*) malloc(sizeof(Person));
    initializePerson(person, "Sue", "Stevenson", "Developer", 28);
    addNode(list, person);

    person = removeNode(list);
    displayPerson(*person);
    person = removeNode(list);
    displayPerson(*person);

    removeLinkedListInstance(list);
```

- **几个值得关注的地方：**
  - 我们只能在list.c文件中创建_linkedList结构体的实例，因为没有完整的结构体声明就无法使用sizeof操作符
  - 在main函数中尝试为这个结构体分配内存
    - LinkedList* list = (LinkedList*)malloc(sizeof(LinkedList));
  - 会得到如下错误
    - error: invalid application of ‘sizeof’ to incomplete type ‘LinkedList’
  - 类型不完整是因为编译器看不到list.c文件中的实际定义

面向对象中的类型

- 面向对象产生的背景
  - 面向对象编程概念的根源可以追溯到SIMULA67，但是直到Smalltalk80才得到全面发展
  - 80年代，软件复用的概念被普遍认为是提高软件开发效率的重要途径，而封装和访问控制的ADT成为主要的复用技术
  - 人们在应用ADT的过程中发现一些ADT对复用支持的问题

ADT的问题

- 抽象数据类型ADT对于复用有这样一些问题：
  - 已有类型的功能和特性往往不能很好地适用于新的应用，而修改需要了解太多的实现细节
  - 所有的数据类型都是独立的，而且都在同一个层次上，而现实问题中的很多实体彼此之间具有类型上的复杂关系，如彼此相似性、概念上的抽象和具体等

继承对ADT的扩展

- 针对ADT存在的问题，继承提供了一种解决方案
  - 在**封装性**方面，继承提供了对被复用ADT**访问控制**的新方式，使得修改可以不用在被复用ADT的原始代码中进行
  - 在类型的关系上，继承提供了一种途径，可以刻画复用与被复用的ADT之间的特殊关系

继承的便利

- 增加了继承机制的ADT，给程序员带来了如下便利：
  - 用继承来直接复用ADT，在不需要了解其源代码的情况下，就可以扩展这个ADT的特性
  - 通过继承，可以建立一个类型体系，用来定义相关类型的层次关系，并以此来反映问题空间中这些类型的上下代关系

继承机制的反面

- 继承机制是在扩展ADT复用能力方面的一种改进，但它也并非只有好处，其不足之处在于：
  - 继承机制强化了类型层次结构中的各个类型之间的相互依赖性，这正是ADT有点的反面
- ADT的最大优点之一
  - 单纯通过ADT构造出来的类型，它们彼此之间具有很好的独立性

权衡

- 问题：能否增加抽象数据类型的复用性，而又不在他们之间产生相互依赖性？
  - **即使不是完全不可能的，至少也是十分困难的！**

- 这一直是一个在高级程序语言的设计和实现方面的权衡难点
  - 这也是为什么很多人非常喜欢动态类型语言的原因
  - *鸭子类型？*

讨论

- 你怎么看待这个问题：
  - 如果你要设计和开发一个新的PL，你会如何看待这个问题？你的取舍是什么？

阅读资料

1. 编程语言原理 10E，Robert Sebesta，清华大学出版社，2013
2. 面向对象分析与设计（第3版），Grady Booch等，电子工业出版社，2012
3. Think in Java 4th, Bruce Eckel, 2006
4. C和C指针，Kenneth Reek，人民邮电出版社，2008
5. C语言接口与实现，David Hanson，人民邮电出版社，2011
6. C程序设计语言（第2版）K&R，机械工业出版社，2004