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

4. 类型

- 类型机制及其作用
- 类型检查
- 类型等价和类型相容
- 类型转换和强制
- 基本类型
- 用户定义类型
- 数据类型与存储管理
- 多态性
- 类型推理

语言的类型机制

类型是程序语言中最重要的特征之一

在目前的常规程序语言里，类型机制的基本方案是：

- 提供一组基本类型（内部类型、预定义类型，是一些基本数据集合），包括类型名，类型值的字面描述方式，为每个类型提供一组基本操作
- 提供一组类型构造机制，可以用它们描述程序所需的新的值形式；或描述（定义）新类型的构造。声明的类型称为用户定义类型

用户定义类型的重要用途：

- 作为描述简单的值或类型的手段，如枚举类型
- 组合已有类型，构造复合的值或者复合类型。复合类型的值可以有复杂的内部结构，以满足各种应用的需要
- 可能为用户定义类型约束相关操作，使之像内部类型一样易于使用

基本类型是一组类型抽象。使用时不必了解各种类型的值的表示，不必了解其操作如何实现，就可以通过类型名和相应操作使用它们

类型机制：历史

程序设计语言里的类型机制随时间和人们的认识不断发展：

- 早期语言如 **Fortran/Algol 60** 只有基本类型
  - 如整数、实数和字符（或字符串），是硬件直接支持的各种数据类型在高级语言层面的反映
  - 允许定义数组变量，但数组不是清晰的类型
- **Lisp** 提供了一种“表”结构作为基本数据构造手段
  - 可以灵活构造各种值
  - 可用于模拟各种复杂的类型机制
- **Pascal** 语言引进类型构造和定义机制（可以给新类型命名）
  - 明确提出了相应的类型等价和相容规则
  - 其中的用户定义类型还不是今天意义下真正的类型

    不支持为把定义的类型关联所需的操作

类型机制：历史

- 大致同年代的 C 语言的类型机制比 Pascal 更弱
  - 其中没有真正的用户定义类型机制
  - typedef 是后来 ANSI C 增加的，只是用于为原本可以描述的类型描述引进一个“别名”，不作为独立的具有区分意义的类型
- 更新的语言，包括新的主流语言如Ada、C++ 和 Java
  - 类型定义机制更完善
  - 目标是使用户类型定义和内部类型具有同等地位（甚至更多）
- 有关类型理论的开拓性研究和实验主要在各种函数式语言里进行
  - 人们先后在 Lisp，ML，Haskell 等语言里深入研究了类型的各种问题
  - 包括实际问题和理论问题
- 下面主要从实践的角度讨论高级语言的类型问题

  也要涉及到类型实现中的一些问题

类型的作用

类型有许多作用：

- 对程序处理的值的概念划分（表示什么？如何操作？...）
- 控制对于相关的值的使用方式
- 为相关操作的执行提供隐含上下文环境。例如 Pascal 的 new(p) 将根据 p 的（指向）类型在堆中分配存储，+ 运算符根据类型进行解析
- 可能更有效实现（不同类型值需要的存储量可能不同，通过类型能更好利用存储；可能静态处理与类型有关的问题，减少动态运行开销）
- 支持静态的类型检查，有助于发现各种各样的编程错误
- 使人可以基于类型来组织程序里的计算过程，确定操作的可用性和操作方式（例如：重载解析）
- 支持各种基于类型的高级程序技术（如抽象数据类型， OO的继承，泛型程序设计等等）
- ...

类型是什么

类型是什么？存在多种不同的观点

类型的指称观点：一个类型是一个值集合。一个值属于某类型的条件是它属于该类型的值集合，一个对象具有某个类型的条件是它的值必定属于这个类型的值集合（有类型的变量）

类型的构造观点：一个类型或者是基本类型，或者是可以基于已有类型通过一组类型构造方式（数组、结构/记录、类等）构造出的复合类型

强调的是类型的实现，类型的值的结构

类型的抽象观点：类型是一种抽象，它有一个良好定义的（使用）界面、有一个名字（类型名）和一组具有相互协调的语义的操作

强调类型的使用，以及实现无关性。具有同样抽象的类型可以互换

程序员对类型的认识，通常是这些观点的某种组合

工作中不同的时候可能关注类型的不同方面的性质

既关心取值，也关心实现和使用

类型系统

程序语言的类型系统包括

- 一套定义类型的机制，通常为它们提供特定的语言结构
- 一套有关类型等价、类型相容和类型推理的规则

- 类型等价规则：判断一些实体是否属于同一类型的准则
- 类型相容性规则：确定一个具有特定类型的值能否合法地使用在某个特定的上下文环境中
- 类型推理规则：是一组规则，利用它们可以基于一个表达式的组成部分的类型，和/或其外围环境的情况，确定这个表达式的类型

语言的类型系统能确定一个程序是否为类型合法的（**well-typed**）

为了程序执行的效率，人们常常希望类型合法性检查可以静态完成

实际情况：许多语言里的大部分类型检查可以静态完成，但也可能有些检查需要推迟到运行中进行（动态类型检查）

类型检查

在将运算符作用于运算对象，执行赋值，或者把实际参数传递给子程序时，都需要检查相关运算对象的类型是否合适：

- 被操作对象（或者值）有其特定类型，程序中指定的操作（上下文）也有对被操作对象的类型的特定要求
- 如果两方面相互合适，操作就可以正常进行（类型良好）
- 如果两方面在类型上并不相容，就是出现了类型冲突

类型检查（**type checking**）是一个处理过程，其目的是保证每个操作都是对一组数目正确、类型合适的对象进行的，以保证操作的有效性

- 如果语言能贯彻一套规则，保证任何操作都不会作用到与之不相容的对象上，这种语言就称为是强类型语言
- 如果语言设计保证所有类型检查都能静态完成，它就是静态类型语言
- 有些语言特征不可能静态检查，例如数组越界，值越界等
- 有些语言采用了要求动态类型检查的特征，例如变量没有声明类型。要保证操作的合法性，就必须在运行中检查变量值的类型

类型检查

类型检查的作用：

检查类型错误（生成静态或动态类型错报告，停止编译或禁止继续操作）

选择合适的操作（例如由加运算的对象类型选择适当的加法操作）

根据需要确定必要的类型转换（如混合类型运算）

多数常规语言的一个重要设计目标是设法使尽可能多的检查能静态做，静态的类型检查尽可能完全，其余情况需要动态检查（如数组越界检查，子界类型的赋值合法性检查不可能静态完成）

完全的静态检查使执行中不再需要考虑类型，不需要保留类型信息（减少存储开销），一切操作都是计算（不做额外动作），取得高效率

人们也把这样的语言称为静态类型语言。如 **Algol 60**，**Pascal**，**Ada** 等

C 也应看作静态类型的语言，程序运行时不做任何类型检查

有些语言由于设计目标，明确要求广泛的运行时类型检查，以提供更多的灵活性（**Lisp**，**Smalltalk** 等），或使用方便性（如各种脚本语言）

类型检查：静态

为支持静态类型检查，要求：

- 能静态确定程序中使用的每个对象（变量等）的类型
- 能静态确定程序里的每个文字量的类型
- 每个操作对于其各个操作对象都有静态确定的明确的类型要求，操作结果（如果存在）也有明确定义的类型

编译中做静态类型检查的基本技术是维护符号表：

- 遇到变量、常量、类型、子程序等的声明时记入名字及其类型信息
- 处理语句和表达式时，利用符号表信息检查其中的类型是否合适
- 从基本运算对象出发，推断表达式的类型

如果操作名存在重载（对一个操作名，存在针对不同类型和/或参数个数的不同操作定义），编译程序需要根据上下文选出适当的操作

例如 “-”运算符，表示各种算术类型的一元求负值和它们的二元减法运算。类型检查过程需要确定每个“-”对应的操作

类型检查：静态或动态

有些问题无法静态检查，例如数组越界（无论语言如何设计）：

```
...a[i + 5]...
```

不同语言采用的处理方式有两种：

- 运行时检查。为此，数据对象里必须包含支持检查的信息（**dope-vector**，**doper**，内情描述），还有在目标程序里加入检查代码
- 不检查（例如C，这意味着要求程序员负责保证不出现越界）

值越界：把值范围大的类型的值赋给值范围较小的类型的变量时，可能出现值越界（子界类型，**int** 值赋给 **char** 变量）。处理办法：

- 动态检查（需加入检查代码）并报告错误
- 不检查（C 语言不检查，计算结果无明确定义。阿丽亚娜火箭）

OO 语言为支持 **OO** 程序设计，引入了另一些与类、继承关系等等有关的操作，导致了一些新的动态检查问题

类型检查：动态

在完全（或基本）采用动态类型检查的语言里，运行中根据动态类型信息选择实际操作，可以提供更大的灵活性，支持更多的编程技术

- 为支持运行时的动态类型检查，每个对象里都需要包含充分的类型信息，通常采用某种标志域和/或类型编码方式
- 在执行各种操作前需要检查被操作对象的类型，根据具体情况确定实际处理方式（或者在发现无法处理时报告动态类型错误）
- 动态检查可保证实际执行的每个操作都是正确的（类型安全性，不能确定正确操作时不执行。静态检查通常不可能完全，无法提供这种保证）
- 支持运行和排除程序错误交替进行（发现/排除运行错误后转换模式）

缺点：

- 运行中，对象携带的类型信息要占用存储空间
- 运行中检查类型、选择正确操作的开销很大，效率可能受到很大影响
- 程序里的类型错误在开发与调试中未必都能发现，可能遗留到运行中

类型等价

基本类型通常相互不等价。在支持用户定义类型的语言里，必须有判断两个类型是否等价的规则，以便判断两个对象是否类型相同

例：

```
type T1 = array[1..10] of integer;
type T2 = array[1..10] of integer;
type T3 = T1;
var x, y : T1;
var z : T2;
var s, t : array[1..10] of integer;
var u : T2;
var v : array[1..10] of integer;
var w : T3;
```

问题： T1, T2, T3之间是否等价？

x, y, z, s, t, u, v, w 中哪些是同类型的变量？

要回答这些问题，必须给类型等价一个清晰的定义。

类型等价

判断类型等价（type equivalence）的方式主要有两种：

- 名字等价：任何两个名字不同的类型都不等价。如果两个变量在一起声明，或是通过同一个类型名声明，则它们的类型相同
- 结构等价：基于类型的成分。两个类型等价，仅当它们的结构构成相同

```
type T1 = array[1..10] of integer;
type T2 = array[1..10] of integer;
type T3 = T1;
var x, y : T1;
var z : T2;
var s, t : array[1..10] of integer;
var u : T2;
var v : array[1..10] of integer;
var w : T3;
```

按不同类型等价规则，这里的各个类型是否等价，变量在否具有相同类型的结论会不同

类型等价：名字等价

名字等价：等价规则严格，判断简单，但要求程序员为每个类型命名

名字等价的基本假设：用户定义了两个类型名，就是想用它们代表不同的类型（表示不同的类型），即使其成分（结构）相同

```
type T1 = array[1..10] of integer;
type T2 = array[1..10] of integer;
type T3 = T1;
var x, y : T1;
var z : T2;
var s, t : array[1..10] of integer;
var u : T2;
var v : array[1..10] of integer;
var w : T3;
```

名字等价的一个问题是类型别名（如 T1 和 T3），由此又分为宽松名字等价规则（认为别名类型是同一类型）和严格名字等价规则

注意：C 语言的 typedef 就是引进类型别名，认为是同一个类型

类型等价：结构等价

结构等价规则比较宽松，可能认为更多类型是等价的。但其判定比较困难。基本方法是展开类型定义，直到只剩下类型构造符、域名和内部类型名的嵌套表示。如果得到的嵌套描述形式相同，那么两个类型等价

- 递归类型和基于指针的类型使事情复杂化，此时这种展开不终止（可以通过一些技术来解决）
- 如果语言的类型系统很丰富，类型等价判定算法的效率就可能很低，甚至可能出现类型等价性在理论上不可判定的情况

在采用结构等价的不同语言里，有关结构等价的定义也有很多变化。人们也有许多不同考虑。例如下面几个描述是否等价：

```
struct A {          struct B {          struct C {
    int a, b;           int m, n;           int n, m;
};                  };                  };
```

说它们等价或不等价都有道理。有些语言规定成员顺序不重要（ML），有些语言允许结构相同（域名可以不同）的记录相互赋值等

类型等价：结构等价

重要缺陷：无法区分程序员认为不同，但恰好内部结构相同的类型

例如，两个类型 **LENGTH** 和 **AREA** 都用实数类型表示。如果有

```
a : LENGTH; b : AREA;
```

采用结构等价规则，编译器就不能判定下面表达式有问题：

```
x := a + b;
```

结构等价的意思简单，比较低级，有不清晰的地方（要求语言给出细节的定义），检查的实现可能耗时，区分能力不足

C 语言的类型等价问题：

- **struct** 和 **union** 按名字等价（把 **struct A** 整个看作类型名）
- 数组不是独立的类型，**enum** 常量取 **int** 值，每个枚举类型等同于某个整数类型（由实现确定），指针的情况比较特殊
- 对通过 **typedef** 引入类型别名采用宽松规则，认为类型别名是相同类型

类型相容

语言需要提出一套类型相容性（**type compatibility**）规则，用于确定特定类型的对象能否用在特定上下文里（程序员关心的问题）

极端情况是要求对象类型与上下文期望的类型等价

但在大部分语言里（的大部分情况下），两者的类型不等价时也可以认为其相容（这说明，“类型相容”的条件可能比类型等价更宽松）

考虑类型相容的一些情况（例）：

- 赋值语句：右部表达式的类型必须与左部（被赋值对象）的类型相容
- + 的两个运算对象或者都与整数类型相容，或者都与浮点数类型相容
- 传入子程序的各实参的类型都必须和对应的形参类型相容，从子程序里传回调用处的各形参的类型都必须与对应实参的类型相容
- 函数里的返回值表达式必须与函数头部声明的类型相容

类型相容

不同的语言里，有关类型相容性的定义差异很大：

Ada 是一个比较严格的典型，**C** 是比较宽松的典型

- **Ada** 最严格：类型 **S** 与期望类型 **T** 相容，当且仅当 **(1) S** 和 **T** 等价，或 **(2)** 一个是另一个的子类型（或同为另一类型的子类型），或 **(3)** 两者都是数组，且各维的元素个数和类型相同。**Ada** 不支持自动类型转换
- **Pascal** 比 **Ada** 宽松，允许同一基类型的子界类型之间的混合运算，并允许将整数用在期望实数的上下文里（自动转换）
- **C** 的规则更宽松。数值类型都相互“相容”，例如可以把 **char** 类型是值用在要求 **double** 的上下文里；把 **double** 送给要求 **int** 的函数；用 **double** 给**char** 变量赋值，把一切数值类型和指针类型当作逻辑值，等等

OO 语言里还有派生类型（子类型）与基类型（父类型）相容的问题

语言设计中对类型相容性的考虑反映了设计者的一些想法，反应了他们对于使用方便性与防止（和检查）程序错误的不同考虑

类型转换

如果 S 类型的对象不能用在期望 T 类型的上下文中（类型不相容），实际中又需要用，就必须显式要求做类型转换（type conversion/cast）。转换的源类型 S 与目标类型 T 的关系有多种情况，这里讨论4种主要情况：

1）源类型 S 与目标类型 T 的值集合不同，但有交集，共有值的表示方式相同。例如 S 是 T 的子类型，或者 S 是带符号整数 T 是无符号整数

这种转换不需要做任何动作，把源类型的值直接作为目标类型的值使用

如果源类型的一些值在目标类型中没有，该怎么办？

例如，将有符号整数转换到无符号整数

- 生成在运行时执行值的检查代码，以保证被转换是目标类型的合法值。检查成功时直接使用该值；失败时产生动态语义错误。有些语言实现允许关闭这种检查，以得到更快速但可能不安全的代码
- 把检查这种问题的责任交给用户，要求用户保证正确性。语言本身不做任何检查（C 语言就是这样）

类型转换

2）源类型 S 与目标类型 T 的值集合的底层表示形式不同，但在它们的值之间存在某种由实际情况确定的对应关系（牵涉到值的变换）。例如：

整数可以变换到 IEEE 标准双精度浮点数，不损失信息（但可能损失精度）

许多处理器提供了完成这种变换的机器指令

双精度数变换到整数有意义清晰的定义，可通过舍入或截断完成变换

大部分处理器也提供了相应的机器指令（有信息损失）

不同长度整数之间的变换可以通过丢掉高位或符号扩充完成。不同长度的浮点数之间可以做尽可能保证精度的转换（可能有表示范围问题）

被变换的值可能在目标类型里不存在等价表示，怎么办？

例如双精度数变换到整数，可能遇到无法转换的值

这种情况应该看着是一种“错误”。在这里也有是否检查错误和是否报告的问题。有速度与安全性之间的权衡问题

类型转换

3）两个类型可认为是结构等价的。所涉及类型采用同样底层表示，具有同样值集合。这时的变换纯粹是概念操作，运行时不需要执行任何代码

这种类型转换的例子（C/C++）：

- 需要把指针值存入整数，或者将保存在整数里的指针值赋回
- 通用指针类型和其他指针类型之间的转换（不同的指针类型之间的转换）

```
int *p = (int*)malloc(....);
```

C++ 用static_cast描述，C 的类型转换都用 (*type_name*) 形式的类型转换描述

一些语言允许下面的 struct A、struct B 和 struct C 之间转换和赋值。

C 禁止非标量类型之间的转换，C++ 里可以自己定义转换

```
struct A {
    int a, b;
};
```

```
struct B {
    int m, n;
};
```

```
struct C {
    int n, m;
};
```


类型转换

4）在复杂系统程序设计中，有时需要改变一个值的类型，但却不改变其表示形式，也就是说，希望按照另一类型的方式解释所用的值。例如：

- 实现存储分配子系统时，常用一个很大的整数数组表示堆，而后把数组的一些部分重解释为指针或整数（用于簿记），或各种用户数据结构
- 在有些高性能数值软件里，需要把浮点数重解释为整数或者记录，以提取其中的指数、尾数和正负号部分，用于实现某些特殊算法

改变类型但不修改基础二进制位表示的转换称为非变换类型转换。Ada 用内部子程序 unchecked_conversion 实现这种转换；C++ 提供 reinterpret_cast 运算符。C里可通过指针实现这种转换，例如：

```
n = *((int *)p); /* 假设 p 是 double 指针 */
```

这种转换直接改变对象的类型（换一种解释），因此是很危险的手段

使用时应特别慎重！（考虑上面的转换）

类型强制

如果某特定对象可以用在某环境中（类型相容），但其类型与环境的类型要求不等价，就需要执行自动隐式类型转换，称为类型强制（**coercion**）

与前面讨论的显式类型转换一样，类型强制也可能需要运行时动态类型检查，可能动态执行变换代码完成底层表示的转换，或出现其他情况

有些语言的相容性规则较宽松，能自动做许多强制转换。例如 **C** 会做：

- 各种数值之间的自动转换（可能执行或不执行代码）
- 整数与指针之间的自动转换（不执行代码）
- 普通指针到通用指针（**void ***）的自动转换（不执行代码）

有些语言采用较严格相容性规则，提供很少的自动转换，甚至不提供任何自动转换。从 **Pascal** 到 **Ada** 的传统是只提供尽可能少的类型强制，要求程序员明确说明所需转换，以支持对程序的严格类型检查和静态类型

这种做法的缺点是常迫使程序员自己写很多显式类型转换

类型强制

如果语言提供范围广泛的自动类型转换，那么：

- 写程序比较方便，可以比较自由地混合使用多种数据类型

对这种设计选择的批评包括：

- 这样做鼓励懒散的程序员，使他们不认真地明确表达自己的意图，把许多工作的决定权交给系统自动进行
- 如果对语言的自动转换规则认识不清晰准确，就可能出现程序实际行为不符合实际需求的情况，可能掩盖实际错误，威胁程序的安全性

总而言之，理解一个具体程序设计语言允许哪些强制，采用什么样的转换规则，对于正确完成程序设计是非常重要的

注意：应该看到自动类型转换的正面和负面作用

在程序设计时，不应该过分依赖程序设计语言本身的自动类型转换，要考虑清楚其中的转换，尽可能多地明确表达所希望的转换

类型强制

有些语言设计者认为自动类型转换能自然地支持抽象和程序扩充

目标是使用户定义的新类型更容易与现有类型一起工作

为此就需要提供用户类型转换的语言描述形式

例如：C++ 继承了 C 语言丰富的强制转换规则，而且

- 程序员可以扩充自动强制转换规则
- 在定义新类型（C++类）时可以定义强制操作，描述如何在新类型和现有类型的值之间来回转换
- 这些规则和重载解析规则相互作用，可以大大增强语言的灵活性，使许多特殊的编程技术很方便使用
- 但这些也是C++里最难理解，难以正确使用的特征之一

  特别是在大系统里，可能有很多用户定义类型，不同开发者可能定义了许多类型之间的相互转换（注意 C++ 在这方面的考虑）

类型推理（类型推断）

某些结构有类型但没有明确给出，此时就需要通过相关结构的信息去推断

最常见的类型推断出现在表达式处理中：

- 推断的基础是变量和文字量的类型
- 需要基于成分的类型和运算符去确定复合表达式的类型
- 确定子表达式与运算符（或子程序）的类型相容性
- 确定是否需要做类型强制

面向对象、子类型（派生类型）、泛型类型、泛型操作等，与自动类型转换、重载、多态性、作用域规则等等相互作用，使一些常规语言里的“类型推断”问题变得越来越复杂

例如 C++ 里的类型推断就有许多规定，相当复杂。Ada 语言的类型推断也很复杂。这方面走得最远的语言之一是 ML，它允许用户不定义类型，语言系统通过一套规则去推导程序里变量和表达式的类型