

道语言类型系统

傅利民(phoolimin@gmail.com)

September 19, 2009

Contents

1	介绍	1
2	隐式类型	2
3	显式类型	4

1 介绍

道语言的类型系统是一个介于动态与静态之间的类型系统，既支持显式的类型声明，也支持隐式的类型推断。对于类型可在编译期确定的变量，它们的操作的有效性也将会在编译期被检查，否则将此检查推迟到运行期进行。这种类型系统既提供了动态语言的灵活性，也在一定程度上提供的静态语言的安全性。

在大多数情况下，当一个变量被一程序语句定义时，道的类型系统可以根据该程序语句里的操作数操作符等推断出该变量的类型。因此在很多情况下，显式声明变量的类型是不必要的，只是偶尔有必要明确指定函数参数或返回的类型。当类型推测系统不能确定某个变量的类型时，它将假定此变量可以取任何值，也就是此变量的类型为**any**。在这种情况下，用户可显式地指定该变量的类型。这样的好处是，这使得道编译器可以采取一定的优化。

实际上，在道语言里准确的类型信息将被用来生成更优化的虚拟机指令。而且允许用户写通用函数，以接受不同类型的变量为参数，而当此函数以某个特定类型的变量为参数调用时，道编译将可以以此通用函数为模板，专门生成一个支持该特定类型的函数。这一点有点象C++里的模板函数。

2 隐式类型

2.1 基本类型

当一个变量在申明时同时被赋值，其类型最容易被推测，该变量将和被赋的值有一样的类型。

```
a = "a string"
b = 123
c = 456$
d = a + a
e = b + b
f = c + c

stdlib.debug();
```

当程序以调试模式运行时（`dao -d ...`），标准方法`stdlib.debug()`将会给出调试命令行提示（调试终端），在此命令行提示下，用户可以检查程序当前的运行情况，如当前运行的虚拟指令（`list` 或 `l` 命令），变量的类型（`about` 或 `a`）和值（`print` 或 `p`），函数调用的栈（`trace` 或 `t`）。

在调试终端下可以看出，*a* 被推断为 *string* 类型，*b* 为 *int* 及 *c* 为 *complex*；并且，这三个加法被分别编译为：*ADD_SS*，*ADD_II* 和 *ADD_CC*。

2.2 列表/映射表/元组/数值数组

当使用枚举或值区间等方式定义列表/映射表/元组/数值数组时，列表等的类型由枚举或值区间表达式里的元素的类型确定。在进行下标操作时，结果变量的类型由列表等的类型以及下标类型确定。

2.2.1 列表

```
a = { 1, 2, 3 } # 枚举,list<int>
b = a[1]      # 单元素,int
e = a[1:1]    # 子列表,list<int>

c = { 1, "A" } # list<any>
d = c[1]      # any

e = { 0.0 : 2.5 : 10 }; # 值区间,list<int>
```

2.2.2 映射表

```
a = { "B"=>1, "A"=>2 } # map<string, int>
b = a["A"]           # int
b = a["A":"B"]       # map<string, int>, slicing, sub-map

c = { "B"=>1, "A"=>{} } # map<string, any>
d = c["A"]           # any

e = { { "B"=>1, "A"=>2 }, { "BB"=>1, "AA"=>2 } } # list<map<string, int>>
```

2.2.3 元组

```
a = ( "abc", 123, 10.55 ) # tuple<string, int, float>
a = ( name=>"abc", index=>123, value=>10.55 ) # tuple<name:string, index:int, value:float>
```

2.2.4 数值数组

```
a = [ 1, 2, 3; 4, 5, 6 ] # array<int>
b = a[0,2]              # int
c = a[1]                # int, treat "a" as a vector
d = a[1,]               # array<int>, slicing, second row
d = a[:,1:]             # array<int>, slicing, sub-matrix
```

道类型系统能对空列表等作特殊处理，使得它们可以在需要时用作某特定类型的列表等使用。如

```
a = { { 1, 2, 3 } }
a.append( {} );

stdio.println( a[1], stdlib.about( a, a[1] ) );
```

Output

```
{ } list<list<int>>[0x1c6060] list<int>[0x1c6160]
```

a 将被推断为 `list<list<int>>` 类型， 当一个空列表被插入中时， 此空列表将自动成为特定的 `list<int>` 类型。

2.3 函数

在道语言里，每个函数将会自动与一个函数类型关联，该函数类型将包含函数的参数名，参数类型和返回值类型等信息。如`routine<a: string=>int>`表示的函数均以`a`为参数名，`string`为参数类型，`int`为返回值类型。而`routine<a= string=>int>`则表示，`a`的参数类型为`string`且有缺省值。

```
routine foo( a ) # routine<a:?=>?>
{
    b = a + a;
    stdlib.debug();
    return a;
}
stdio.println( stdlib.about(foo) );

foo( 1 ) # routine<a:int=>int>
foo( "ABC" ) # routine<a:string=>string>
```

此例中，三个名为`foo()`将被生成。其中一个是为原本类型`routine<a:?=>?>`，用于参数类型不确定的情况。另外两个是为函数调用`foo(1)`和`foo("ABC")`自动生成的，它们的类型分别为`routine<a:int=>int>`和`routine<a:string=>string>`。这两生成的函数将分别针对参数类型`int`和`string`作优化，即在它们的函数体里使用特别用于`int`类型和`string`类型的虚拟指令。

3 显式类型

变量的类型可以按以下方式被显示的申明：

```
变量名 : 类型名
变量名 : 类型名 = 值
```

```
a : int
b : string = "ABC"
```

在函数的参数列表里可以使用同样的方式申明参数的类型。

类型名 可以是以下内置的类型关键词：`int`，`float`，`double`，`long`，`string`，`complex`，`list`，`map`，`tuple`，`pair`，`array`，`buffer`，`routine`，以及它们的组合。类型名也可以是道类名或载入模块里用户定义的C/C++类型名。类型名也可以是特殊的符号，如：`any`表示任意类型，`?`表示未定义类型，以及`@X`表示待定类型。`@X`主

要是被用于函数参数中，如

```
routine test( a : list<@T>, b : @T )
{
}
```

当某种特定类型的列表作为参数 a 被传递给该函数时，那么道类型系统将要求第二个参数 b 的类型跟 a 的元素的类型一致。

3.1 函数类型

函数类型由关键词**routine**和其他信息组成。用户通常不需要显示地定义它们。函数类型也可以按其他类型一样的方式使用，例如：

```
myfunc : routine<p1:string,p2=int=>int>
```

这将申明`myfunc`为一个函数，此函数将以字符串和整形为参数，并返回一个整数。在这个函数类型里，参数名也被给定，这将使得此函数被调用时，参数可按参数名传入。并且第二参数被申明为有缺省值，可在调用时省略。另外，`...`也可出现在函数类型的参数列表里，表示此函数可接受可变个数目的参数(类似于C里的`valist`)。

例子：

```
routine foo( a, b : string, c = 1 )
{
    return c;
}
stdio.println( stdlib.about( foo ) );
```

Output

```
routine<a:?,b:string,c=int=>?[0x25e510]
```

值得注意的是，函数类型之间的匹配有些不同于普通数据类型的匹配。对于普通的类型，更具体的（专门的窄的）类型可以匹配到一样的或更宽泛的类型，反之则不行。而对于函数类型，情况则刚好相反，也就是只可以宽泛的函数类型匹配到一样的或更窄的函数类型。

下面例子将展示为什么函数类型的匹配应当这样。考虑下面的情形：

```
routine foo( a : int ){ stdio.println( a ); }
```

```
routine bar( callback : routine<any=>any> ){ callback( "ABC" ) }  
  
bar( foo );  
  
myfunc : routine<any=>any> = foo;  
myfunc( "ABC" );
```

如果是允许更窄的函数类型`foo:routine<int=>?>` 匹配到更宽泛的函数类型`routine<any=>any>`，那么，`bar()` 就可以以`foo` 为参数`callback` 而被调用，这时就会出现一个问题。因为从`bar()` 的参数`callback` 的类型看，`bar()` 可能会以任意参数如字符串等来调用`callback`。如果`foo` 被作为参数传递给了`bar()`，显然`callback("ABC")` 将导致一个异常。因此，只允许宽泛的函数类型匹配到一样的或更窄的函数类型才可以保证这种回调函数的安全。