文章编号:1001 - 9081(2009)01 - 0261 - 04

多类型混合运算的面向对象设计与实现

师鸣若[1], 姜中华[2]
(1. 北京物资学院 信息学院, 北京 101149; 2. 中国科学院 软件研究所, 北京 100080)
(shimingruo@163.com)

摘 要:针对多种数据类型的混合运算问题进行了研究。讨论了面向过程的解决方案和两种可能的面向对象解决方案及不足。依据敏捷设计原则,提出了一种双层函数指针数组(虚表)的面向对象新方案。分析表明,该设计具备开放封闭性,解决了灵活性、可重用性、可维护性和效率问题。

关键词:开放封闭原则(OCP);虚函数表;敏捷设计原则;面向对象;运行时类型信息
中图分类号: TP311.11 **文献标志码**:A

Object-oriented design and implementation for multi-type mixed operation

SHI Ming-ruo[1], JIANG Zhong-hua[2]
(1. *School of Information, Beijing Wuzi University, Beijing* 101149, *China*;
2. *Institute of Software, Chinese Academy of Sciences, Beijing* 100080, *China*)

Abstract: Through studying multi-type arithmetic in many systems, it is found that the existing solutions are not flexible. Process-oriented solution and two potential object-oriented solutions were given and their shortages were analyzed. To achieve flexibility, reusability, maintainability and efficiency, a two-layer virtual table object-oriented solution was proposed based on agile design policy and the Open-Closed Principle (OCP). Analysis shows that the solution is relatively good.

Key words: Open-Closed Principle (OCP); virtual table; agile design policy; object oriented; run-time type identification (RTTI)

0 引言

在动态隐式类型脚本语言解释器开发过程中,不可避免的要遇到多种数据类型的混合运算问题。从功能上讲,解释器就是一种环境,向解释器输入一条命令,解释器就会执行这条命令,立刻看到命令的执行结果。从实现上讲,解释器主要负责词法分析、语法分析和语义分析,一般没有生成、优化中间代码和生成机器码的过程。多种数据类型的混合运算需要在语义分析过程中有效解决。

针对多类型混合运算的问题,在分析面向过程和两种可能的面向对象[2]的解决方案的基础上,依据敏捷设计原则[1],提出了一种较好的面向对象的设计方案,并对其优缺点进行了分析。

1 问题提出

项目需要实现一个解释执行的动态隐式类型语言,其中有多种内置类型:整数(多精度),分数(多精度),字符串等。这里所说的动态语言是指在运行期间才能去做数据类型检查的语言,隐式类型语言对变量没有显式的声明,即一个变量没有固定类型,它的类型会随着当前存储的内容而变化,如python 语言。假定动态语言希望支持如图 1 所示的多种类型的运算,具体的运算规则根据数学性质进行定义。

项目选用C++语言来实现该解释器,解释器需要做三项工作:词法分析、语法分析和语义分析。对于语法分析和词法分析部分,选择 Lex 和 Yacc 作为词法和语法分析工具;语义分析的工作包括如何支持多种类型之间的混合运算,例如加、减、乘、除等二元运算。

下面是多类型混合运算的实例:

```
a = 5;
b = 1.0;
s = "ab";
c = a * b;                                  // c = 5.0;
d = a * s;                                  // d = "abababababab";
e = b * s;                                  // 不支持的运算类型
```

为了方便描述,将问题进行抽象:该语言中有整数(Int),分数(Fraction),字符串(String)内置类型,如何描述它们之间的四则运算? 问题是,解释器并不知道其真实类型,它所知道的是某种抽象类型,而具体四则运算的结果取决于它们的真实类型,如 Int 和 Fraction 做运算得到 Fraction。

2 面向过程的解决方案

现在开始处理多种类型的四则运算。一个比较通用的面向过程的解决方案是定义一些全局函数,代码片段如下:

```
SymbolObj * multiply(SymbolObj * lhs, SymbolObj * rhs)
{ if (lhs ->type == INT)
  { if (rhs ->type == INT)
    { ...                    /* Do something to process Int * Int */
    }
    else if (rhs ->type == FRACTION)
    { ...           /* Do something to process Int * FRACTION */
    }
  }
```

收稿日期:2008 - 07 - 28;**修回日期**:2008 - 10 - 06。 **基金项目**:国家 863 计划项目(2007AA012447);北京市教育委员会科技计划项目(KM200810037001);北京市属高等学校人才强教计划项目(PHR-IHLB);北京市教育委员会科研基地建设项目。

作者简介:师鸣若(1975 -),女,河南洛阳人,讲师,主要研究方向:软件设计、网络安全、数据挖掘; 姜中华(1972 -),男,湖北襄樊人,博士,主要研究方向:密码理论、网格计算、软件设计。

```
    else if
    ...
}
```

该解决方案的代码不是很长,其优点是浅显直接,容易理解。不过也存在重大的缺陷:任何新的类型(如:多项式(Polynomial))加入系统都需要更改这些 if…else…语句,即使只是忘记了一处,程序都将有 bug,而且它不那么显眼。这样的程序在 C 语言中已经有很久的历史了,本质上其实是没有可维护性的。

3 面向对象的解决方案

上面的面向过程的方案违背了开放封闭原则(Open-Closed Principle, OCP)[1],即软件实体(类,模块,函数等)应该对于扩展是开放的,对于修改是封闭的。依据 OCP 原则,以后再进行同样的修改时,只需要添加新的代码,而不必改动已经正常运行的代码。在很多方面,OCP 都是面向对象设计的核心所在,遵循这个原则可以带来很多好处(如:灵活性、可重用性以及可维护性)。假定数据类型的类图如图1所示。下面将依据此原则来寻找面向对象[2-5]的解决方案。

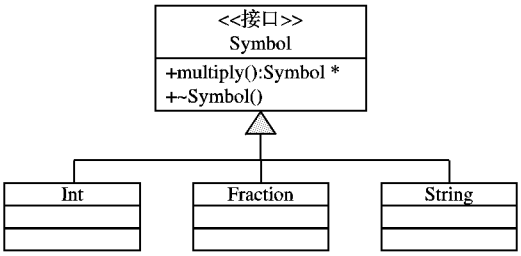

图1 面向对象多数据类型的继承关系

现在所面临的问题是处理过程取决于两个对象的类型,即作用在多个对象上的虚函数,但C++,Java 语言并没有提供这样的虚函数。换一种支持这种特性的语言通常是行不通的,所以只能针对已有的面向对象语言寻找解决方案。

3.1 解决方案一

C++语言本身提供了一层虚函数调度,这只为该问题的解决完成了一半工作,一个比较直接的想法是,把虚函数和 if…else 语句混用。代码片段如下:

```
class Symbol
{ public:
    virtual Symbol * multiply(Symbol * other, bool reversed =
      false) =0;
    virtual ~Symbol(){}
};
Symbol * Int::multiply(Symbol * other, bool reversed)
{ if (typeid(*other) == typeid(Int))
    {.../* Do something to process Int * Int*/}
  else if (reversed == true)
    throw UnSupportedOp("*");
  else
    return other->multiply(this, true);
}
Symbol * Fraction::multiply(Symbol * other, bool reversed)
{ if (typeid(*other) == typeid(Int))
    {.../* Do something to process Fraction * Int*/}
  else if (typeid(*other) == typeid(Fraction))
    {.../* Do something to process Fraction*Fraction */}
  else if (reversed == true)
    throw UnSupportedOp("*");
  else
    return other->multiply(this, true);
}
```

该解决方案的实现也比较简单,并很容易让它工作。代码里的标志位 reversed 有两个作用:一是为了支持某些类型(如矩阵乘)不满足乘法交换律的情况;二是为了作为调用的出口。需要说明的是它也遵循 OCP 原则,即后期再增加新的类型,无需更改上面的代码。注意到最后一个 else 语句调用了反向乘,这其实是为后期加入的新类型提供了“接口”。加入 String 和 Polynomial 类型的代码片段如下:

```
Symbol * String::multpily(Symbol * other, bool reversed)
{ const type_info & objectType = typeid(*other);
  if (objectType == typeid(Int))
    {.../* Do something to process String * Int*/}
  else if (reversed == true)
    throw UnSupportedOp();
  else
    return other->multpily(this, true);
}
Symbol * Polynomial::multpily(Symbol * other, bool reversed)
{ const type_info & objectType = typeid(*other);
  if (objectType == typeid(Int))
    {.../* Do something to process Polynomial * Int*/}
  else if (objectType == typeid(Fraction))
    {.../* Process Polynomial * Fraction*/}
  else if (reversed == true)
    throw UnSupportedOp();
  else
    return other->multpily(this, true);
}
```

根据第1章所示的类型定义规则,String 和 Int 可以做乘法运算,其结果是相当于 String 的连接,String 和 Fraction 不可运算;Polynomial 和 Int 以及 Fraction 都可做运算,但不可与 String 做运算。

这里使用 RTTI 的问题之一就是效率,首先一个 typeid 调用要做几次整型比较和一次取址操作,可能还有2次整型加法,这依赖于不同的编译器实现;再者就是类型重复判断,即如果没有找到 $A * B$ 相匹配的处理,反转时 A 的类型信息丢失,需要在处理 B 时再次判断。第二个问题就是每个类都必须知道它的同胞类,这在多层的继承体系中是很困难的。

这种 if…else 技巧在 C 语言中导致了错误,在C++中也是一样。为了支持不能处理的未知类型,该方案抛出了一个异常,但调用者往往不能从该异常中得到详细原因。

3.2 解决方案二

能否只用虚函数来解决这个问题,这个方案和方案一有同样的基本架构,只是基类中的虚函数个数增加了。其基本原理就是用两层虚函数,第一层决定第一个对象的动态类型,第二层决定第二个对象的动态类型,代码片段如下:

```
class Fraction;
class String;
class Int;
class Symbol
{ public:
    virtual Symbol * multiply(Symbol * other, bool reversed =
      false) = 0;
```

```
    virtual Symbol * multiply(Int * other, bool reversed =
      false) = 0;
    virtual Symbol * multiply(Fraction * other, bool reversed =
      false) =0;
    virtual Symbol * multiply(String * other, bool reversed =
      false) = 0;
    virtual ~Symbol(){}
};
class Int : public Symbol
{ public:
    Symbol * multiply(Symbol * other, bool reversed)
    { other -> multiply ( *this, !reversed); }
    Symbol * multiply(Int * other, bool reversed = false);
    Symbol * multiply(Fraction * other, bool reversed = false);
    Symbol * multiply(String * other, bool reversed = false);
};
class Fraction : public Symbol
{ public:
    Symbol * multiply(Symbol * other, bool reversed);
    Symbol * multiply(Int * other, bool reversed = false);
    Symbol * multiply(Fraction * other, bool reversed = false);
    Symbol * multiply(String * other, bool reversed = false);
};
class String : public Symbol
{ public:
    Symbol * multiply(Symbol * other, bool reversed);
    Symbol * multiply(Int * other, bool reversed = false);
    Symbol * multiply(Fraction * other, bool reversed = false);
    Symbol * multiply(String * other, bool reversed = false);
};
```

这种解决方案,代码非常简单,也没有 RTTI,并且不需要为意料之外的对象抛出异常。但是它却有个致命的缺陷,即违反了 OCP 原则,当增加新类时所有代码都需要更新,虽然这里没有 if…else 需要更改,但是更糟糕的是每个类都需要增加一个新的虚函数。像 python 这样的开源的可扩展的语言,在无需更改原来框架及接口的情况下,可以在此基础上增加新类型,完成扩展库。如果采用了方案二,则维护和重编译的代价将是巨大的。

如果需要这种行为依赖于两种或者多种类型的设计,虚函数的方法比 RTTI 的方法安全,但同时也限制了程序的可控性,而 RTTI 不需要重编译,但这导致了代码的不可维护性。

方案三给出了一种可控性和可维护性都比较理想的方案。

3.3　解决方案三

注意到编译器通常创建一个函数指针数组(vtbl)来实现虚函数,并在虚函数被调用时进行下标索引,这样就避免使用 if…else 链。如果用户自己实现一套这样的虚表,则比基于 RTTI 的代码效率更高,也限制了 RTTI 的使用范围。第一层使用编译器的虚函数、第二层使用自己模拟虚表的方法实现的程序段如下:

```
class Symbol
{ public:
    virtual Symbol * multiply(Symbol * other, bool reversed =
      false) = 0;
    virtual ~Symbol(){}
};
class Fraction : public Symbol{
  typedef Symbol * (Fraction::*MulFuncPtr)(Symbol * other);
  typedef map<const type_info*, MulFuncPtr> MulMap;
  static MulMap initMap();
  static MulFuncPtr lookMulFuncPtr(Symbol * other);
  public:
    Symbol * multiply(Symbol * other, bool reversed = false);
    Symbol * mulInt(Symbol * other);
    Symbol * mulFraction(Symbol * other);
};
Symbol * Fraction::multiply(Symbol * other, bool reversed){
  MulFuncPtr mptr = lookMulFuncPtr(other);
  if(mptr)
    return (this->*mptr)(other);
  else if(reversed == false)
        return other->multiply(this, true);
  else
    throw UnSupportedOp();
}
Fraction::MulMap Fraction::initMap(){
  MulMap map;
  map[&typeid(Int)] = &mulInt;
  map[&typeid(Fraction)] = &mulFraction;
  return map;
}
Fraction::MulFuncPtr Fraction::lookMulFuncPtr(Symbol * other){
  static MulMap mulMap(initMap());
  MulMap::iterator funcEntry = mulMap.find(&typeid(*other));
  if(funcEntry == mulMap.end()) return NULL;
  return funcEntry->second;
}
class Int : public Symbol
{   typedef Symbol * (Int::*MulFuncPtr)(Symbol * other);
    typedef map<const type_info*, MulFuncPtr> MulMap;
    static MulMap initMap();
    static MulFuncPtr lookMulFuncPtr(Symbol * other);
  public:
    Symbol * multiply(Symbol * other, bool reversed = false);
    Symbol * mulInt(Symbol * other);
};
```

与方案一基于 RTTI 的方法类似,Symbol 只有一个处理乘法运算的函数,它实现了第一层虚函数,而每种乘法都有一个独立的函数处理,这里使用了不同的函数名字,是因为乘法运算有相同的参数类型,必须使用名字来区分。为了实现第二层模拟虚函数表,每个 Symbol 实现类为 multiply()运算增加模拟虚表、模拟虚表初始化函数以及模拟虚表查询函数,方案利用了模拟虚表的局部性。上述程序段是该方案的一种实现。

Fraction 是方案三的完整实现。multiply()函数需要将参数 other 的动态类型映射到一个成员指针,通过查询模拟虚表 mulMap 完成。模拟虚表 mulMap 为 std::map,是 type_info 指针到成员函数指针的映射。initMap()用于初始化模拟虚表,仅需要在第一次查询模拟虚表时初始化。multiply()调用 lookMulFuncPtr()查找运算的实际处理函数。lookMulFuncPtr()查询模拟虚表,根据 other 类型(type_info)决定与之对应的独立处理函数指针,进而可以调用实际函数完成运算。篇幅所限,省略了 String 类型的定义以及 Int 类的 initMap()、multiply()和 lookMulFuncPtr()的实现,它们

与 Fraction 中相应方法的实现类似。

假定需要计算 $r = i \times f$,其中 i 和 f 分别为给定的多精度整数 Int 和多精度分数 Fraction。下面结合实例 r = i. multiply(f);说明第二层模拟虚函数表的工作过程:1)首先 Int :: multiply()被调用,如果 Int :: multiply()首次被调用,则 Int :: initMap()初始化模拟虚表 Int :: mulMap。进而查询 Int :: mulMap,但没有找到实际处理函数,又由于 reversed = false,调用被反转;2) Fraction :: multiply()被调用,由于 Fraction :: multiply()首次被调用,则 Fraction :: initMap()初始化模拟虚表 Fraction :: mulMap, Fraction :: lookMulFuncPtr()进而查询模拟虚表,返回 Fraction :: mulFraction, Fraction :: multiply()调用 Fraction :: mulFraction 函数,完成运算 $i \times f$,结果保存至 r 中。

方案三兼具方案一和方案二的优点,程序的可控性和程序的可维护性都得到了保证,满足 OCP 设计原则,是一种比较理想的方案。对于 Java(或 C#)语言,可以用接口代替函数指针来实现方案三。

4 结语

在许多研究和应用领域中都会面临多类型混合运算问题,如动态隐式类型脚本语言解释器以及支持多种复杂数据类型的计算环境。首先分析了面向过程解决方案以及面向对象的两种可能的方案,然后提出了一种比较理想的面向对象的解决方案。在该方案中,采用了两层虚函数机制,第一层使用编译器的虚函数,第二层使用自己模拟虚表实现。该方案满足 OCP 设计原则,而且计算效率高,灵活性强。仅以乘法运算为例,若将其他运算加入到方案三中,只需要另外增加一个虚函数,为每一个数据类型增加一个与运算对应的模拟虚表即可。

参考文献:

[1] MARTIN R C. Agile software development principles, patterns, and practices[M]. 影印版. 北京: 中国电力出版社, 2003.

[2] LIPPMAN S B. Inside the C++ object model[M]. 影印版. 北京: 中国电力出版社, 2003.

[3] MEYERS S. Effective C++[M]. 影印版. 北京: 电子工业出版社, 2006.

[4] MEYERS S. More effective C++[M]. 影印版. 北京: 电子工业出版社, 2006.

[5] GAMMA E, HELM R, JOHNSON R, *et al*. 设计模式: 可复用面向对象软件的基础[M]. 影印版. 北京: 机械工业出版社, 2004.

(上接第257页)

OPC 客户程序首先要实现浏览 OPC 服务器的功能。客户程序需要创建 OPC 基金会提供的 OPC 服务器浏览器对象(OPCServerList),再通过该对象的 IOPC ServerList 接口获得 OPC 服务器名称的列表。

OPC 客户程序需要实现与 OPC 服务器进行交换的功能,包括创建服务器对象、组对象,读写数据等。开发时需要注意 COM 对象的引用计数问题、内存管理问题和处理错误返回代码问题[4]。

OPC 客户端由 3 大模块组成:

1)OPC 接口模块。即可进行和控制层 OPC 服务器之间的实时数据的双向读写操作,包括建立和断开 OPC 连接,读写实时数据,并将数据存在内存表中。

2)数据切换控制模块。包括监测服务器状态字,内存拷贝,数据切换逻辑。

3)OPC 配置模块。定义和维护 OPC 服务器、OPC 组和 OPC 项清单,以及 OPC 通信参数。值得注意的是,不同厂商的 OPC 服务器名字,通信方式(轮询、事件)各不相同,在进行配置时需注意。

3.2 OPC 连接实现步骤

OPC 数据集成模块通过读取其配置文件或 OPC Browser 接口,按次序建立 OPC 服务器、OPC 组及 OPC 项,并进行连接。一旦建立 OPC 连接后,OPC 客户端即可进行和 OPC 服务器之间的实时数据的双向读写操作。主要实现步骤如下:

1)实例化期望的 OPC 数据服务器并建立连接;

2)获取该服务器的组列表和项集合;

3)访问项目名;

4)断开连接,释放资源。

以 Rockwell Automation 的 RSLinx OPC 服务器为例,以上步骤采用自动化接口的 VB 实现如下:

```
'Object reference to RSLinx OPC interface
Private MyOPCServer As RSLinxOPCAutomation.OPCServer
'Object reference to an OPCGroup
Private WithEvents MyOPCGroup As RSLinxOPCAutomation.
  OPCGroup
…
MyOPCServer.Connect "RSLinx Remote OPC Server",
  sRemoteMachine
Set MyOPCGroup = MyOPCServer.OPCGroups.Add("OPCDemo")
…
Dim oOPCItem As RSLinxOPCAutomation.OPCItem
MyOPCGroup.OPCItems.AddItems lIndex, arItemIDs,
  arClientHandles, arServerHandles, arErrors
…
MyOPCServer.OPCGroups.RemoveAll
Set MyOPCGroup = Nothing
MyOPCServer.Disconnect
…
```

4 结语

在冗余系统中,并联的子系统间的切换及切换时间是检验冗余实现的重要指标。本文提出的双 OPC 服务器同时运行的“热备”冗余系统,较之于国内外其他通过 OPC 服务监视软件进行的 OPC 服务器主、备切换方式,具有快速、无扰动的特点,而且客户端内部的实现逻辑较简单,开发容易,具有较高的实用价值。

参考文献:

[1] 王达宗, 马增良. SCADA 系统冗余模式下数据同步的实现模型[J]. 微计算机信息, 2005, 21(4): 76-77.

[2] 陈丹丹, 钱美, 夏立, 等. OPC 服务器开发的几种方法[J]. 微计算机信息, 2006, 22(16): 28-29.

[3] 陈迪泉. OPC 技术与服务器开发[J]. 广东通信技术, 2005, 25(5): 6-11.

[4] 马欣, 李京, 程峥嵘, 等. “工控软件互操作规范 OPC 技术”讲座: 第4讲 OPC 服务器与客户程序的设计[J]. 自动化仪表, 2002, 23(7): 68-70.