维普资讯 http://www.cqvip.com

计算机科学 1999Vol. 26№. 3

面向对象语言中的子类型关系

Subtyping in Object-Oriented Languages

李 勇 陈家骏 郑国梁

（南京大学计算机科学与技术系 南京 210093）

Abstract Developing software with object-oriented method can be considered as progressive refinement based on data abstraction, implemented by inheritance mechanism. In order to guarantee software design, new data abstraction and the old should have the subtype-supertype relationship, satisfying the principle of substitutability. However, most of popular object-oriented programming languages only provide the implementation inheritance mechanism for code reuse. They do not support directly the behavioral inheritance which embodies the design reuse directly. In this paper, we discuss subtype-supertype relationship and how to make restrictions on the use of implementation inheritace so that class and its subclass may have the relation of behavioral inheritance.

Keywords Behavioral inheritance, Implementation inheritance, Substitutability, Subtype-supertype relationship, Covariance rule, Contravariance rule

一、前言

面向对象程序设计中的继承，作为一种模块扩充机制和一种类型精化机制，一方面能通过增加或修改已有类的特征去定义新类，为实现软件的重用提供了一种途径；另一方面能支持通过例化已有的类型去定义新类型，提供了由分析设计向实现的平滑转换。因此，继承相应地也应分为实现继承和行为继承两种方式。实现继承主要是为了代码的重用和共享；行为继承考虑的是由指引的多态导致了用子类对象替换父类对象的可能性，为了保证这种替换的正确性而要求子父类型间应满足的关系。在构造一个系统的过程中，行为继承更使得我们可以将子类行为当作父类行为进行推导。然而，大多数面向对象程序设计语言仅提供了用于实现代码重用的继承机制，即实现继承，而没有对另一种更深层次的体现设计重用的行为继承（子类型关系）给予足够的支持。不加限制地使用实现继承将使得子类与父类有完全不同的外部行为，这不仅使得程序难以理解，而且使得程序难以维护。本文中将以 Eiffel[5] 和 C++[10] 作为具体的面向对象程序设计语言进行讨论。

二、数据类型

2.1 数据抽象

抽象是构造和理解大型复杂系统强有力的手段，它把行为和行为的实现分开考虑。最早的程序抽象机制是过程抽象，数据抽象是另外一种程序抽象机制，它是围绕数据进行的。在数据抽象中，数据及其操作被封装在一起，数据使用者只需要知道对数据能进行哪一些操作，而不必知道数据的具体表示及数据操作的具体实现，它体现了一种信息隐蔽原则。在面向对象程序设计语言中，数据抽象是由类来实现的，特别在 C++ 中由于类（class）的成员变量允许是公有的（public），即可以被外界直接操作修改。因此，严格地讲，在面向对象程序设计语言中数据抽象是由成员变量为私有的类实现的。

2.2 类型

类型为程序设计语言中重要设施，刻画了一组值及其上可施行的一组操作，是对其成分行为的描述。类型机制不仅可在编译时提供信息，还能对程序的设计和理解提供很大的帮助。

在面向对象程序设计中，类型描述了一组具有相同性质的对象，这里性质是指从对象外部可观察到的与实现无关的那些属性，即对象的行为，可使用

维普资讯 http://www.cqvip.com

前后置条件将行为表示为：

{P}Action{Q}({P}S{Q}表示若S在条件P下执行，则S一定终止且终止状态满足条件Q[4])

前置条件P说明了对象行为的作用范围，公理语义表示则说明了数据与操作的关系。对象的外部行为可以通过向它们发送消息来观察，因此，类型描述了具有相同外部行为的一组对象，但各个对象的具体实现可以不同，以同样的方式使用同类型的对象将得到相同的结果。

在面向对象程序设计中，类是类型的基础，类型要靠类来定义和实现。一个类型对应一个类，类既用于描述行为又给出了实现，对象由类来创建。这样做的好处在于简化程度设计。

三、子类型关系

3.1 重定义

在大多数面向对象程序设计语言中提供的继承会直接复制父类的代码(特征的实现)，然而考虑到正确性和功效性，通常必须根据特定的条件利用重定义设施修改已有类型的特征，进行适当的调整。

重定义设施可以对特征的型构、实现或规格说明进行调整改动，但重定义特征的型构必须和父类中原特征的型构相适应，型构相适应在不同的语言中有着不同的含义。在Eiffel中，型构相适应要求子类型重定义行为的参数(输入和输出参数)的类型必须是父类原行为相应参数的子类型，这就是协变规则，以Eiffel为代表的这类语言也被称为协变语言；而C++中重定义行为的型构必须和原行为完全相同，即型构的参数个数相同且对应参数类型和结果类型相同。

3.2 继承多态和动态定联

多态的一般含义是，某一论域中的元素可以有多种解释。从这个意义上来说，传统语言中的一名多用和类属都是支持多态的设施。继承机制所支持的多态的基本含义是：子类的实例亦是父类的实例，这是因为父类所表述的概念比子类所表述的概念更一般。但在具有静态类型检查的面向对象程序设计语言系统(如：Eiffel、C++等)中，并不能简单地将子类实例作为父类的实例进行处理，因为它们是不同的对象。多态并不涉及在运行时刻改变对象，或将其从一种形式变为另一种形式，而只表示一给定的指引类型变量在运行时刻可指向不同类(被指引类型的子类)的实例，且只有对象的指引变量才能成为多态变量，其动态类型由运行时刻所具体指向的对象类型所决定。这也正是继承多态在面向对象程序设计语言中的实现。

对于具有继承关系的子类型与父类型，对同一特征通过重定义可以有不同的实现版本。多态变量在运行时刻可以有多种动态类型，对其特征的调用无法在编译时刻确定，如只按照其定义点声明类型来静态定联就无法体现其类型的多态性，因此必须使特征具有自动适用于对象的能力，动态定联设施提供了这种支持。

动态定联规则是指变量的动态类型决定了所用特征的版本。对于多态的对象指引变量其所用的特征的不同版本是指父类中特征和子类中对其进行了重定义的特征。在大多数面向对象程序设计语言(如Eiffel)中，对象指引变量的特征调用就会自动通过动态定联机制根据其动态类型选择合适的特征版本，由于动态定联需要在运行时刻才能确定，不利于生成高质量的代码且降低了运行效率，故在C++中，子类只有对在父类中声明为虚(virtural)方法的特征进行重定义才可动态定联，将能静态定联的成分尽量静态定联，以提高运行效率。

3.3 子类型关系

最初引入继承机制的目的，是因为定义新的类时，如果能利用一个已存在的类，直接使用其所有属性和方法，然后再加入新的属性和方法去构造将十分方便。但此继承关系又产生了另一种关系：如果类B是从类A继承而来，则类B的对象就至少具有类A对象所具有的一切属性和方法。那么在任何需要类A对象的地方使用类B对象应导致同样的效果。依据此可替换原则可给出子类型的定义：对所有用类型T来定义的程序P，如果用类型S的对象O_1来替换程序中类型T的对象O_2，P的行为不变，则称S是T的子类型。这种可替换原则可表示为：

$\{Pre\}P\{Q\}\Rightarrow\{Pre\}P^{O_2}_{O_1}\{Q\}$($P^{O_2}_{O_1}$表示将在P中将所有自由出现的$O_2$同时合法代以$O_1$的结果[5])。这种类型之间在语义上的可替换关系，通常又称为行为继承。

3.4 行为继承的优点

程序Q是程序P的精化可定义为：$P\sqsubseteq Q$ iff $\forall R: P\langle R\rangle\Rightarrow Q\langle R\rangle$[6](其中$P\langle R\rangle$表示程序P满足后置条件R的最弱前置条件)。可以进一步推出：$P\sqsubseteq Q$ iff $\forall Pre\forall R:\{Pre\}P\{R\}\Rightarrow\{Pre\}Q\{R\}$。

根据可替换原则的表示可以发现实际上程序$P^{O_2}_{O_1}$正是程序P的精化，由此可见采用行为继承来进行软件开发，就是一个逐步精化的过程。因此，行为

维普资讯 http://www.cqvip.com

继承的一个好处是可以实现软件的增量开发，在软件开发的早期阶段，我们可能只知道某个数据抽象的一部分操作和一部分行为，随着软件开发过程的进展，对原有的数据抽象又有了新的要求，必须增加新的操作，从而产生新的数据抽象。新的数据抽象可看成是由老的数据抽象精化而来，它们分别由两种类型来描述，这两种类型之间存在子类型（行为继承）关系。上述的精化可重复多次，从而形成一个层次的子类型结构，它反映了程序各部分的不同要求，以层次方式来设计类型可以控制设计错误的影响范围，把握设计过程以及对实现给予指导。

另外，行为继承还可实现软件的设计重用，往往重用设计比重用代码更有利于软件开发。

四、行为继承

4.1 语法条件

为满足可替换原则，首先要满足替换的合法性，即用一个类型 S 的对象替换任何一处类型 T 的对象的出现时不会导致运行时刻的类型错误。这也被称为子类型关系的弱条件定义[11]。为了保证这种满足语法上可替换原则的子类型关系，需满足以下几个（语法）条件：

（1）子类型中必须至少具有父类型中的所有行为；

（2）子类型中可以增加新的行为；

（3）若对父类型中行为进行重定义，则重新定义的行为必须和父类型中的原行为相适应。

这里的两个行为相适应是指：

（a）它们具有相同的参数数目；

（b）重定义行为的输入参数的类型必须是原行为相应参数的类型的父类型（或为相同类型）；

（c）重定义行为的输出参数的类型必须是原行为相应参数的类型的子类型（或为相同类型）。

其中条件（b）被称为反变规则，看上去是有违常理和直觉的。因为面向对象的程序设计是一个逐步精化的过程，精化后的程序应处理更特殊的情况，故精化类型的行为的输入参数也应进一步精化，所以子类型重新定义的行为的输入参数的类型应该是原行为相应参数类型的子类型（或为相同类型），即满足协变规则。但反变规则恰恰是保证动态类型正确所必需的，下面以协变语言 Eiffel 为例说明反变规则的必要性。

定义如下两个类：食物（Food）和动物（Animal），其中类 Animal 中有方法吃（Eat），其参数类型是 Food，

```
class Food
    ……
end-class Food
class Animal
    Feature
        ……
        Eat(X: Food)is
            ……
            end
end-class Animal
```

然后再定义类 Food 的两个子类：肉（Meat）和草（Grass）及类 Animal 的两个子类：老虎（Tiger）和羊（Sheep），其中类 Tiger 和 Sheep 对父类 Animal 的方法 Eat 进行了重定义，使得它们的参数类型分别精化为 Meat 和 Grass。

```
class Meat
    inherit
        Food
    end
    ……
end--class Meat
class Tiger
    inherit
        Animal
            redefine Eat
            end
    feature
        ……
        Eat(X:Meat)is ……
            end
        ……
end--class Tiger
class Grass
    inherit
        Food
    end
    ……
end--class Grass
class Sheep
    inherit
        Animal
            redefine Eat
            end
    feature
        ……
        Eat(X:Grass)is ……
            end
        ……
end--class Sheep
```

在下面两个的程序片断中：

```
a:Animal;          a:Animal;
t:Tiger;           t:Tiger;
f:Food;            f:Food;
m:Meat;            g:Grass;
……                 ……
a:=t;              a:=t;
f:=m;              f:=g;
a.Eat(f);          a.Eat(f);
片段1              片段2
```

都声明了类 Animal 和类 Food 的指引变量 a 和 f，a 指向子类 Tiger 的实例 t，f 可以分别动态地指向子类 Meat 和 Grass 的实例 m 和 g，调用 a 的方法 Eat 时，由于其类型需动态确定，故输入参数只要是类 Food 或其子类的实例即可通过编译时刻的静态

维普资讯 http://www.cqvip.com

语法检查。在程序片断1中，输入是类 Meat 的实例 m，故动态语法也正确；但在程序片断2中，输入是类 Grass 的实例 g，故将产生动态类型错误。由此可见协变语言中仅使用动态定联，将不可避免地产生动态类型错误。

C++中重定义方法的型构必须和原方法完全相同，既是协变特例，也是反变特例，因为满足反变规则，故原则上 C++中仅使用方法重定义和动态定联不会产生动态类型错误，但由于其又反映一定的协变要求和 C++提供的强制类型转换机制，在实际应用中也可能产生动态类型错误，就上例考察如下的 C++实现：

```
class Food{……};
class Meat:public Food{……};
class Grass:public Food{……};
class Animal {public:virtual void Eat(Food* f){……};};
class Tiger: public Animal {public: void Eat(Food* f)
      {……};};
Tiger* tiger=new Tiger();
Grass* grass=new Grass();
Animal* animal=tiger;
animal→Eat(grass);
```

其中类 Animal 的指引变量 animal 指向子类 Tiger 的实例，故在调用 animal 的方法 Eat 时，将自动地动态定联到子类 Tiger 重定义的方法 Eat，由于 C++中对方法重定义型构一致的要求，类 Tiger 中所重定义的方法 Eat 的输入参数也必须是类 Food 的指引变量，故虽然它可指向类 Food 子类的任一实例，但在重定义的方法 Eat 中只会将其做为类 Food 的指引变量来使用，即仅根据类 Food 具有的特征对该实例进行特征调用。由于子类对父类的类型适应性，这种直接使用虽然在语义上可能是错误的（如上例中老虎吃草的错误），但是不会导致动态类型出错。C++中又提供了强制类型转换机制，使得在子类重定义的方法中可以将输入参数强制转换成所应进一步精化的类型，从而支持类型精化过程中对方法参数类型的精化。在上例中，由于类 Tiger 的方法 Eat 中的参数类型应精化成类 Food 的子类 Meat，故在实现重定义类 Tiger 中重定义方法 Eat 时，程序员会很自然地将输入的指引变量参数所指向的实例转换成类 Meat 的实例来使用，这就将可能导致动态的类型错误，实际上任何提供强制类型转换机制语言的动态类型都是不安全的。

由此可见反变规则是保障语法上类型正确所必需的，但从类型精化的角度来说，面向对象程序设计语言中又必须支持协变。David L. Shang[6]指出，利用面向对象程序设计语言中的类属机制采用参数化类来准确恰当地描述程序中存在的各种类型依赖关系，协变所产生的动态类型错误问题可以被完全解决。对此解决方法在这里不再赘述。

4.2 语义条件

上面给出了子类型关系的语法条件，这样虽然使得运行时刻的动态类型错误不再发生，但仍无法保证替换后的语义正确性（如上例中老虎吃草的错误），即满足行为继承关系。语义性质直接关系到软件设计的正确性，只从语法检查的角度考虑软件设计的正确性是不够的，子类对象不仅应能合法地成为父类对象的替换物，更应拥有父类对象的所有性质，能无差别地替换父类型对象，满足子类型关系。不难证明其充要条件为：

$\forall i, \{Pre\}O_2.action\{Q\} \Rightarrow \exists j, (\{Pre\}O_1.action_j\{Q\})$

为满足这种具有行为继承的可替换原则的子类型关系，需再增加以下两个（语义）条件[3]：

（1）如果对父类中的行为进行了重定义，则重定义行为被调用后的返回值必须和调用父类中原行为的返回值相同。

（2）如果对父类中的行为进行了重定义，则重定义行为被调用后修改子类中继承父类的属性值必须和在相同初始属性值条件下调用父类中原行为后的修改值相同。

4.3 行为继承的实现

在 Eiffel 中提供了包括前置条件和后置条件以及类的不变式的断言设施描述软件系统的语义性质，对象的行为也可由前置条件和后置条件以及类的不变式来描述，故可以直接利用 Eiffel 语言中的断言设施来实现行为继承。

在大多数没有提供断言设施的面向对象程序语言中，必须遵从一定的规则来保障行为继承，下面以 C++为例，详细讨论如何对其类衍生（实现继承）机制加以限制，使得父类与子类之间满足行为继承关系。例如，对于一家商场的顾客，可以用下述的一个 C++类 Customer 来描述。

```
class Customer
{private:
    int payment;
    public:
    Customer(int n=0){payment=n;};
    virtual void set_payment(int n ){payment=n;};
    int get_payment(){return payment;};
};
```

为简明起见，此例中认为行为的作用范围为一切状态，则可把行为表示成类型的该行为后应满足的一些恒等式，该类每个对象 cust 的行为 set_payment

维普资讯 http://www.cqvip.com

可表示成下述的恒等式：

[cust set_payment(n)]. payment==n

([obj. func(para)]表示对象 obj 在 func 操作之后的新状态)持商场优惠卡的顾客除具有一般顾客的所有行为以外，还具有一些特殊行为，如：他们可凭借持卡类型对部分或全部购买商品打折，用类 Customer_with_card 描述如下：

```
class Customer_with_card public Customer
{private
    CardType card;
    public:
    Customer_with_card(int n,CardType c):Customer
        (n){card=c;},
    CardType get_card(int n ){return card;};
    void set_card(CardType c){card=c;},
};
```

类 Customer_with_card 是类 Customer 的子类，若要满足子类型关系，类 Customer_with_card 的每个对象 cust_wc 也满足上述的恒等式：

[cust_wc. set_payment(n)]. payment==n

但持卡顾客可以打折，折扣金额由某算法 discount()计算得到，则在类 Customer_wiht_card 的定义中可把类 Customer 的成员函数 set_payment 重定义为：

```
Customer_with_card::set_payment(int n)
{Customer. set_payment(n-discount());}
```

这样得到的类 Customer_with_card 将不再满足上述的恒等式，这时，

```
[cust_wc. set_payment(n)] payment==
    n-discount()
```

如果在类 Customer_with_card 中不对 set_payment 重定义，而是给出一个新的成员函数 set_discount 其定义如下：

```
Customer_with_card::set-discount()
{set_payment(get_payment()-discount());}
```

或者，在类 Customer_with_card 中定义一个新的成员变量 Discount，把 set_payment 重定义成：

```
Customer_with_card::set_payment(int n)
{Customer. set_payment(n);Discount=discount();}
```

然后定义新的成员函数 set_discount 为：

```
Customer_with_card::set_discount()
{set_payment(get_payment()-Discount);}
```

则类 Customer_with_card 的对象仍然满足上述的恒等式。

综上所述，在 C++ 中，首先为了保证数据的封装性，子类成员函数最好不要直接操作父类的成员变量，而应通过父类的成员函数来进行，这样，当父类的数据表示发生变化不至于影响到子类的实现。其次，要使得类与子类之间符合行为继承关系，在定义子类时应采取下述原则：(1)不对父类的成员函数重定义，(2)如果对父类的成员函数重定义，则重定义时应以相同的方式改变父类的成员变量，并且要保证重定义的函数作用于子类对象时所得到的返回值，应与父类相应函数作用于父类对象所得到的值相同。条件(2)也就是为上面所列为满足行为继承所增加的语义条件。

结束语 数据的封装与继承用一般的程序设计语言也能实现，但用面向对象语言实现起来显得更方便、更直接。对于目前大多数仅提供实现继承机制的面向对象程序设计语言来讲，其目的是为了使用上的简单，但使用者必须遵循一些规则，才能设计出符合面向对象思想的程序。也有一些语言直接提供了行为继承机制[5]，分别对行为和实现进行继承，强调于类型关系是为了类型安全(语法和语义上)，但安全是个相对的概念，不管一种语言有多安全，也无法保证用此语言编写的程序是不会出错或崩溃的。我们希望的是通过一定的措施避免错误，而不是让错误产生然后再去发现它们，纠正它们，这也正是子类型关系的重要性所在。

参 考 文 献

1 徐家福，王志坚，翟成祥. 对象式程序设计语言. 南京：南京大学出版社，1992

2 America P Designing an Object-Oriented Programmung Language with Behavior Subtyping. LNCS. 489, Springer Verlag, 1991

3 Clark R G. Type safety and behavioural inheritance. Information and Software Technology, 1995

4 Liskov B. Data Abstraction and Hierarchy. OOPSLA '87 Addendum to the Proceedings, 1987

5 Gries D. The Science of Programming, Springer-Verlag New York Inc., 1981

6 Morgan C. The Specification Statement. ACM Transactions on Programming Languages and Systems, 1988, 3 (7)

7 Booch G. Object-Oriented Analysis and Design. The Benjamin/Cummings Publishing Company, Inc., 1994

8 Shang D L. Covariant Deep Subtyping Reconsidered. ACM Sigplan Notices, 1995(5)

9 Meyer B. Eiffel: The Language, Prentice Hall, 1991

10 Stroustrup B. The C++ Programming Language, Addison-Wesley, 1986

11 Porter H H. Separation the subtype hierarchy from the inheritance of implementation. Object-oriented Programming, 1992, 4(9)

12 Cook W R. Object-Oriented Programmung versus Abstract Data Types, LNCS, 489, Springer Verlag, 1991