文章编号:1000-5862(2018)03-0304-07

泛型编程在面向对象语言中的对比研究

周卫星[1],左正康[1*],王昌晶[1],石海鹤[1],游 珍[2],谢武平[2],陶小明[1]
(1. 江西师范大学计算机信息工程学院,江西 南昌 330022;2. 江西师范大学江西省高性能计算重点实验室,江西 南昌 330022)

摘要:泛型程序设计通过从更高的层次对具体的算法和数据结构进行抽象,提高了软件的可扩展性、复用性和类型安全性,它主要是基于一系列自定义的类型约束而不是仅局限于预定义类型. 该文对 C + +、Concepts C + +、Java、C#以及 Python 等 5 种不同的、支持泛型程序设计的面向对象程序设计语言进行综合比较,并执行一个典型的且能够较好支持泛型编程机制的例子来分析这些面向对象语言的特征及对泛型编程机制的支撑程度,从而探寻泛型编程的本质特性,为减少程序开发的冗余、增强程序的可维护性提供参考.

关键词:泛型程序设计;类型约束;面向对象程序语言;泛型机制
中图分类号:TP 311 **文献标志码**:A **DOI**:10.16357/j.cnki.issn1000-5862.2018.03.14

0 引言

随着面向对象程序设计(Object-Oriented Programming,OOP)的发展,人们在软件开发过程中不断总结发现,通过将程序中的类型或操作作为自定义的参数类型,然后在程序执行过程中对这些类型参数进行实例化,会极大地减少因类型不匹配而导致的程序崩溃等问题,从而使得这些由自定义类型参数构成的组件更具通用性,该发现成为了泛型程序设计(Generic Programming,GP)思想的开端,并使得 GP 变得越来越至关重要.

泛型程序设计是一种把算法程序从独立的源程序中抽离出来的抽象编程机制,具有简化程序、提高代码可读性与复用性的优势,是 OOP 思想的更进一步发展[1-2]. 泛型程序设计的高抽象性使得程序设计中的类型错误更易发现和纠正,以至于更清晰、更深刻地了解程序语言中出现的一些问题,成为了软件设计中的一种更为通用和高效的抽象范式[3-4],现今这种思想被广泛应用于多种支持泛型程序机制的程序设计语言中[5]. 这种抽象编程机制运用最为成熟而且广泛的是 C + + 中的 STL(Standard Template Library)[6-7] 容器库. C + + 在泛型编程思想的指导下,克服了 OOP 思想抽象程度不高、代码重用性差的劣势,因此 STL 成为 C + + 标准库中的重要组成部分,是泛型编程思想的一个里程碑[8]. 了解并深入学习 GP 的广泛特征也变得越来越重要.

针对以上这些思想,本文针对 C + +、Concepts C + +、Java、C#、Python 等几种 OOP 语言进行泛型设计,并利用运行的结果来分析其对泛型程序设计机制的支撑程度. 本文不是试图证明哪一类 OOP 语言最好,而是展示它们在支持泛型编程方面,哪种语言表现的效果具有优势,支撑程度更高. 为了研究效果更为直观,设计了一个支持泛型编程机制的案例,并使用上述 5 种编程语言进行实现,通过观察程序执行过程以及最终结果,来对比分析以上几类语言对泛型程序设计的不同实现机制,同时也能够深刻了解泛型程序设计在上述几种编程语言的程序通用性与安全性方面的支持程度.

1 泛型编程

面向对象程序思想较好地适应了软件规模的膨胀速度,在一定程度上提高了编程软件的稳定性、可

收稿日期:2017-12-29
基金项目:国家自然科学基金(61462039,61762049,61662035,61462041),江西省自然科学基金(20171BAB202013,20171BAB202008)和江西省教育厅科技课题(GJJ150349,GJJ160329)资助项目.
通信作者:左正康(1980-),男,江西抚州人,副教授,博士,主要从事泛型程序设计和可信软件的研究. E-mail:kerrykaren@126.com

修改性和重用性,受到了编程人员的广泛认可和推崇. 但是随着其在软件工程等方面的广泛应用,也暴露出了越来越多的问题,主要包括抽象程度不高,通用性较差等. 因此针对 OOP 的不足,泛型编程思想也随之出现,最早是由 Alexander Stepanov 和 David Musser 提出,其目的是实现 C + + 中的 STL,并且将模板作为实现类型参数化的主要支撑[9],也就是将本属于某一个特定类型的数据或者算法的类型抽取出来,用一个类型模板参数来表示,从而达到泛化的目的.

从需求及方法思想层次来看,泛型编程已成为面向对象编程思想的一种正向扩展,从这 2 种思想的发展历程中可以发现,它们都是为了解决同时期编程思想无法满足快速发展的软件系统稳定性及通用性需求的问题,目前研究人员已经开始去探索一种新的语法泛型编程机制[10],甚至于利用更高层次的抽象表达方法建立一种泛型之上的泛型模式[11]. 与 OOP 思想中根据对象的行为及特征的不同而使用相异方法不同,泛型编程是将一个特定于一些类型的算法中的与上述类型无关的共性抽象出来,从而提高程序的通用性和高效性.

面向对象编程主要是用对象类来表示所描述事物的概念并作为其基本抽象单位,而泛型编程的基本抽象单位则被称为概念,它表示为满足相同类型需求的对象的类型集合. 下面给出了概念最一般的定义[12]:设 S 是一个抽象集合,一个泛型编程的概念 C 是 A 中所有满足特定类型需求集合 R 的子集,即 $C(R/S)=\{a\in A\mid \forall r\in R, r(a)=\text{true}\}$.

在某些支持泛型编程的程序编程语言中,S 为基本的抽象类型集合;R 为一些特定类型需求的集合(为操作,如! =,* 和 + +);概念 C 则为 S 中满足需求集合 R 的类型集合. 因此泛型概念为一组由需求确定的集合,即它为需求的抽象.

2　泛型编程之 C + +

在此需要介绍 C + + 泛型的几个基本特征.

(i)概念(Concept). Concept 是对一系列满足确定需求的类型的一种形式化抽象,这些需求有可能是语法的、语义的或者是表现相关的[13],该特征成为了 ISO C + + 标准化组织修订 C + + 0x 中的一部分[14]. 对于一种属于 Concept 中的类型,当这种类型满足 Concept 的类型需求,就可以称之为 Concept 的 Model. 如定义了一个模板参数 T,且 int 满足 T 的类型需求,则可称 int 为抽象类型 T 的 Model.

(ii)容器(Container). Container 表示一些对象区间的集合. 对于链表或者是数组,他们都可看成属于一段存储对象的空间. 因此,可以将其抽象出来,变成一个通用性的容器,其中可以存储满足概念需求的对象,并利用迭代器来连接各对象,从而方便管理这些对象.

(iii)迭代器(Iterators). Iterators 表示一种指向对象的对象的指针(即双重指针),主要是用来遍历整个 Container 空间,能将容器与作用在其上的算法分离开来,以保证容器的通用性和类型安全性.

现在大部分的研究是关注抽象数据类型方面[15-17],基于此,本文研究建立一种泛型算法应用于数据存储空间的抽象模型. 在 C + + 语言中,主要使用泛型模板来实现该抽象模型:

```
template < typename T,typename U >
T Find(T first,T last,const U & value) {
while((first! = last)&&! (* first = = value))
 + + first;
return first;
}
```

该程序段为一个 C + + 函数模板,其中 Find 函数是一个判断需要查找的值是否在容器中的模板函数,程序中循环条件是判断查询类型为 T 的对象中是否存在待查询的值,若条件表达式为真,则通过容器中的迭代器将指针指向容器中的下一个对象,继续执行循环;若没超过容器结束位置,就返回所找到的对象,否则返回一个默认构造,即表示该容器中不存在值 value. 从这里可以看出,每一个适合类型 T 的对象都可以使用这个模板算法,当且仅当这个对象能够执行! =、* 和 + + 操作. 但是在项目程序中,并没有办法完全确定最后到底有哪些对象会调用 Find 模板函数,因此也无法确定对象是否可以执行! =、* 和 + + 操作. 这就需要编程人员把这种需求写入说明文档,也就是说必须满足这个说明文档才能调用且执行这个算法. 现在假设有 2 个满足类型需求为 Dereference 概念的类的对象,分别为 first 和 last,还有一个属于 Dereference 概念关联类型的对象 value,其中说明文档见表 1.

表 1　Dereference 概念

Dereference 概念	
first! = last&&(* first = = value)	类型为 bool
+ + first;	类型为 T

将 STL 中的 array 作为 Dereference 概念的一个

Model,且其迭代器也符合这个概念说明文档,即 ! =、* 和 + +操作,因此可以将他们作为模板参数进行相关操作. 下面为主函数简单代码示例:

```
int main(){
array < int,5 > arr = {{2,3,8,6,4}};int value = 2;
array < int > ::iterator it1 = arr. begin(), it2 = arr.
    end();
array < int > ::iterator nRet = Find(it1,it2,value);
return 0;
}
```

从上面的 C + +泛型程序中可以看出,虽然传入的参数类型是一个数组,但是由于该数组能够满足预定义的类型约束,因此该数组能够调用该 Find 泛型算法,这就是 C + +泛型的最重要特征,而对于常规的查找过程是不能满足 Find 泛型算法类型约束的. 由此可见,泛型编程机制能够将类型参数抽象为一种存储数据的容器,并通过迭代器使之满足类型约束,达到了算法与数据结构分离的目的.

3 泛型编程之 Concepts C + +

虽然 C + +通过建立 STL 实现了泛型编程,但随着程序的变革和软件需求的发展,一个非常严重的问题也随之暴露出来,当程序运行出错时,将无法确定地捕捉程序出错信息,导致程序的调试和运行效率极大地降低. 如算法中的泛型 Model 无法满足需求操作(int 类型不能执行取值* 操作). 因此为了快速地定位到错误发生的位置,必须在模板参数上声明一种用户自定义的类型系统用来进行语义检查. 一般来说,定义一个概念用来作为一系列的操作说明,但是实验表明这种方法只适用于小例子并且需要耗费大量的时间[18-21]. 为了更好地解决此问题,于是 Concepts C + +[22] 产生(至今还没有一个开放的程序平台支持运行,目前仅能够在 Concept GCC 执行). Concepts 由 3 个主要的部分组成[23]: Concept 定义、Where 限制和 Concept 匹配. 其中 Concept 定义指定了类操作的类型需求,Where 限制则指定了对 concepts 模板参数的限制,Concept 匹配说明类型怎样与 Concept 需求相匹配.

因此,对于上面 C + +泛型中提到的模板,可以 Concepts C + +实现如下:

Concept 定义:

```
auto concept EqualityComparable < typename T,
typename U = T > {
bool operator = = (T,U);
}
concept InputIterator < typename T > {
typename value_type;//value_type 为关联类型
T& operator + + (T&); //前向 + +
bool operator = = (T,T);
bool operator! = (T,T);
value_type operator* (T);
}; // 将泛型 C + +说明文档程序显示
```

Where 限制:

```
template < InputIterator T,typename U >
where
EqualityComparable < InputIterator < T > ::
    value_type,U >
T Find(T first,T last,const U& value){
while((first! = last)&&! (* first = = value))
    + + first;
return first;
}
```

最后,若想找到一个整形数组中的元素,则 Concept 匹配可用代码表示如下:

```
bool Maps(int* array,int n,int value){
return Find(array,array + n,value)! = array + n;
}
```

以上就是 Concepts C + +的实现原理,它在现有的 C + +泛型上增加了需求抽象及类型抽象等限制,使得在执行程序时能保证在编译阶段就能准确地找到程序错误位置,因此可极大地提高编程的效率和代码安全性.

4 泛型编程之 Java

Java 泛型是通过接口(interface)来表示概念的,同时类型声明也通过接口被用来作为一个概念的 Model,因此类型参数约束可以被接口直接表达并被编译器执行. 而且泛型算法中的参数和这个算法主体部分可以通过类型需求分开检查,从而有利于各类型应用在泛型算法的编译和执行过程中. 执行代码如下:

```
public class Array {
public static void main(String I args) throws
    IOException {
Array1 it1 = new Array1(0);
Array1 it2 = new Array1(5);
```

```
final int num = 2;
Array1 resultArray = Concept.Find(it1,it2,
    num);
}
}
interface EqualityComparable<T>{
boolean Comparable(T last);//比较值的大小
int value();//取值
void next();//查看下一个
}
class Concept {
public static <T extends EqualityComparable<T>>
    T Find(T first,T last, final int value){
while((! first.Comparable(last)) && !
    (first.value() == value))
first.next();
return first;
}
}
class Array1 implements EqualityComparable
    <Array1>{
static int[] beanArray = {2,3,8,6,4};
int currentIndex;
Array1(int index)
{ currentIndex = index; }
public boolean Comparable(Array1 last)
{ return this.currentIndex == last.currentIndex;
}
public int value()
{ return beanArray[currentIndex]; }
public void next()
{ currentIndex++; }
}
```

以上是一个抽象泛型查找算法的 Java 实现. Java 中的泛型是 Sun 公司在 JDK5.0 中发布的一个重要特性[24-26],主要基于满足程序设计中类型参数化的需求. Java 泛型编程机制通过在编译阶段利用类型擦除原理将所有的泛型参数信息都转换成传入的限定类型. 由于加入了类型擦除机制,使得程序在编译阶段就能够发现隐秘的类型需求问题,增强了代码的安全性和可靠性.

5　泛型编程之 C#

C#泛型扩展了传统 C#编程语言的类、接口和方法,并且除了语法上有些许不同外,它的泛型类和方法定义和 Java 泛型是相似的. 以下为上述例子的 C#泛型代码实现:

```
public static class Array{
static void Main(string[] args){
Array1 it1 = new Array1(0);
Array1 it2 = new Array1(5);
const int num = 2;
Array1 resultArray =
    Concept<Array1>.Find(it1,it2,num);
}
}
public interface EqualityComparable<T>{
bool Comparable(T last);//比较值的大小
int value();//取值
void next();//查看下一个
}
class Concept<T> where T:
    EqualityComparable<T> {
public static T Find(T first,T last,int value){
while((! first.Comparable(last)) && !
    (first.value() == value))
first.next();
return first;
}
}
public class Array1 : EqualityComparable
    <Array1>{
static int[] beanArray = {2,3,8,6,4};
int currentIndex;
public Array1(int index)
{currentIndex = index;}
public bool Comparable(Array1 last)
{return this.currentIndex == last.curren
    tIndex;}
public int value()
{return beanArray[currentIndex];}
public void next()
{currentIndex++;}
}
```

C#中的泛型主要是通过 2 步编译机制,首先为需求类型 Array 产生 IL(Intermediate Language)代码和元数据,而并不将泛型实例化,然后再通过 JIT(just-in-time)编译时才进行限定类型的实例化. 这样可以保证不同封闭泛型类有不同的本地代码,从而使得程序更为灵活,提高程序性能.

6 泛型编程之 Python

Python 是一种动态的强类型语言,该语言的泛型是通过使用动态类型机制来实现的,即在运行时才确定算法参数类型,从而提高了代码编写效率并减少代码编译时间,但是会在一定程度上降低代码的排错能力、可读性和运行时效率. 以下是 Python 的对于例子的代码实现:

```
#! /usr/bin/env python
# coding:utf-8
from collections import Iterator
def main():
x = [2,3,8,6,4]
first = iter(x) #获取 x 的前向迭代器
last = reversed(x) #获取 x 的反向迭代器
value = 2
rtn = Find(first,last,value)
def Find(first,last,value):
try:
x_value = next(first)
while first ! = last and x_value ! = value:
x_value = next(first)
except StopIteration:
return
return x_value
if __name__ = = '__main__':
main()
```

该语言忽略了类型的显示表示,可使程序员能够更好地进行代码编写,学习者更好地了解泛型编程机制的内涵,但在其内部还是需要通过类型推导机制来进行隐式转换,从而达到动态类型的效果,降低代码的出错率. 相较于 C、C++、Perl 等编程语言,能够在一定程度上保证程序的安全性与可靠性,提高代码的编写和优化过程,更容易达到一个更为高级的抽象层次,从而降低代码的复杂度;缺点则是由于程序是解释型语言,导致执行过程中容易出现变量类型、属性以及方法错误,需要程序员凭经验确定,存在一定的安全隐患,并且需要逐个确定变量类型,增加了时间开销.

7 对比与展望

针对以上介绍的支持泛型编程机制的程序设计语言,可以较清晰地了解上述几种编程语言对泛型编程机制的支撑程度和实现方式,使得开发人员能够更好地运用,甚至于改进编程语言的错误定位方式和寻找出有效算法的最抽象表示. 以下对上述编程语言泛型编程机制中的泛型算法、Concept、泛型约束等特性进行对比(如表 2 所示).

表 2 几种编程语言泛型特性对比

	泛型算法	Concept	泛型约束	泛型建模
C++	template 和函数重载	说明文档	说明文档	说明文档
Concepts C++	Concept 匹配	Concept 定义	where 限制	Concept 匹配
Java	参数化静态方法	interface	extends	implements 相关接口
C#	参数化静态方法	interface	基于接口的算法参数约束	inherit 相关接口
Python	动态类型机制	说明文档	运行时隐式类型以及属性、方法匹配	duck typing

在该研究中,通过对同一个泛型例子应用在不同编程语言中,展示泛型强大的通用性和良好的代码复用性,因此将泛型编程思想融入到各大主流编程语言成为了开发人员越来越热衷的一个方向,但是对于泛型编程在程序中出现的错误信息不明确和可靠性难以保证以及函数范式不能自适应数据类型[28]等问题,也给项目调试和维护带来了较大的问题. 对此本文较为清晰地讨论并分析了 C++、Concepts C++、Java、C#和 Python 的泛型编程实现机制和对泛型的支持程度,为其他编程人员提供了较充分的框架参考. 可以看到,相比上述几种编程语言,Concepts C++不仅综合利用了 C++语言速度快的优势,同时使用 Concept 匹配来自动化保证对参数类型需求检查的正确性,减少了人为因素的影响;

同时 Concepts C + + 具有更高层次的抽象和简洁的代码,并能快速定位到错误代码的位置,把隐式的类型实例化转变为显式的,提高了代码的安全性和通用性,因此 Concepts C + + 在完整实现泛型编程机制上具有明显的优势.

软件工程应用极大发展推动了泛型编程思想的广泛应用,了解并更深入地使用成为了教学以及课题研究的关注热点,因此,在目前研究工作的基础上,可以对以下的问题进行展望:

1)泛型编程在函数式编程语言中使用非常广泛,但是由于函数式语言中函数功能的定义比较具体化,程序扩展性和复用性较差. 因此针对此类函数式编程语言中的未知类型的函数参数,可以建立一种泛型算法分析参数的结果,从而进行相关转换,自动生成函数定义,提高代码的复用性.

2)大量编程语言可以在一定程度上支持泛型编程机制,对于这些编程语言,可以利用生成式程序设计[29]思想,以形式化 PAR 方法[30-31]为指导,利用 Apla[32]语言对各种语言机制进行泛型机制表达建立,形成一种具有更高抽象层次的编程语言构件库,从而能够综合各类语言优势,拓展 PAR 平台应用范围.

3)开发一种具有灵活性的元编程专用模式机制,通过将泛型编程机制的范围扩展到时间维度上,从而能够利用元编程专用模式机制来解决高度抽象化的程序和性能开销之间的问题,同时利用 LMS[33](Lightweight Modular Staging)来设计更为抽象化的程序生成器,可以较大地降低泛型编程机制应用于编程语言的时空开销并保证了类型安全.

8　参考文献

[1] Chen Yewang, Jiang Zhixiong, Zhao Wenyun, et al. Generic component: a generic programming approach [EB/OL]. [2017-10-12]. https://www.computer.org/csdl/proceedings/cit/2007/2983/00/29830087-abs.html.

[2] Yallop J. Staging generic programming [M]. New York: ACM, 2016: 85-96.

[3] Järvi J, Lumsdaine A, Gregor D P, et al. Generic programming and high-performance libraries [J]. International Journal of Parallel Programming, 2005, 33(2): 145-164.

[4] Wirth N. Algorithms + data structures = programs [M]. Englewood: Prentice Hall Press, 1976.

[5] Garcia R, Jarvi J, Lumsdaine A, et al. An extended comparative study of language support for generic programming [J]. Journal of Functional Programming, 2007, 17(2): 145-205.

[6] Stepanov A, Meng L. The standard template library [J]. C/C + + Users Journal, 1999, 13(12): 10-20.

[7] Matthew H. Austern. Generic programming and the STL [M]. Englewood: Addison-Wesley, 2003.

[8] 孙斌. 面向对象、泛型程序设计与类型约束检查 [J]. 计算机学报, 2004, 27(11): 1492-1504.

[9] 陈林,徐宝文. 基于源代码静态分析的 C + + 0x 泛型概念抽取 [J]. 计算机学报, 2009: 1973-1974.

[10] Érdi G. Generic description of well-scoped, well-typed syntaxes [EB/OL]. [2017-10-12]. arXiv preprint arXiv: 1804.00119.

[11] Magalhães J P, Löh A. Generic generic programming [M]. Berlin: Springer International Publishing, 2014: 216-231.

[12] Musser D R. The tecton concept description language [EB/OL]. [2017-10-12]. http://www.cs.rpi.edu/~musser/gp.

[13] Bachelet B, Mahul A, Yon L. Generic programming: controlling static specialization with Concepts in C + + [EB/OL]. [2017-10-12]. http://pdfs.semanticscholar.org/8486/ce9e2bb6d1061f55b6057b0d98aa8c654ada.pdf.

[14] Reis G D, Stroustrup B. Specifying C + + concepts [C] // Conference Record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, 2006: 295-308.

[15] Dijkstra E W. Notes on structured programming [EB/OL]. [2017-10-12]. http://www.cs.utexas.edu/~EWD/ewd02xx/EWD249.PDF.

[16] 陈叶旺,余金山. 泛型编程与设计模式 [J]. 计算机科学, 2006, 33(4): 254-255.

[17] Belyakova J. Language support for generic programming in object-oriented languages: peculiarities, drawbacks, ways of improvement [EB/OL]. [2017-10-12]. http://www.springer.com/cda/content/document/cda_downloaddocument/9783319452784-t1.pdf? SGWID = 0-0-45-1588893-p180207745.

[18] Bjarne Stroustrup, Gabriel Dos Reis. Concepts: design choices for template argument checking [EB/OL]. [2017-10-12]. http://www2.open-std.org/JTC1/SC22/WG21/docs/papers/2003/n1522.pdf.

[19] Bjarne Stroustrup, Gabriel Dos Reis. Concepts: syntax and composition [EB/OL]. [2017-10-12]. http://www2.open-std.org/JTC1/SC22/WG21/docs/papers/2003/n1536.pdf.

[20] Bjarne Stroustrup, Gabriel Dos Reis. A Concept Design (rev. 1) [EB/OL]. [2017-10-12]. http://www2.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1782.pdf.

[21] Stroustrup B, Reis G D. Supporting SELL for high-performance computing [C]. Berlin: Heidelberg, 2005: 458-465.

[22] Gregor D, Siek J G, Willcock J, et al. Concepts for C++0x(Revision 1) [EB/OL]. [2017-10-19]. http://www.www.stroustrup.com/SELL-HPC.pdf.

[23] Gregor D, Stroustrup B. Concepts(Revision 1) [EB/OL]. [2017-10-19]. http://www2.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n2081.pdf.

[24] 田方,石海鹤,左正康,等. 一种抽象泛型机制的新型 Java 实现 [J]. 江西师范大学学报:自然科学版,2016,40(1):77-82.

[25] 吴国凤,方珏. 基于 Java 语言中的泛型研究 [C]//全国第 19 届计算机技术与应用(CACIS)学术会议论文集(下册),2008:1043-1047.

[26] Y Daniel Liang. Introduction to java programming [D]. California: Armstrong Atlantic State University, 2012.

[27] Belyakova J, Mikhalkovich S. Pitfalls of C# Generics and their solution using concepts [EB/OL]. [2017-10-19]. http://www.ispras.ru/proceedings/docs/2015/27/3/isp_27_2015_3_29.pdf.

[28] Lämmel R, Jones S P. Scrap your boilerplate: a practical design pattern for generic programming [J]. ACM SIGPLAN Notices, 2003, 38(3): 26-37.

[29] 韩何,梁海华. 产生式编程:方法、工具与应用 [M]. 北京:中国电力出版社,2004.

[30] 石海鹤,薛锦云. 基于 PAR 的算法形式化开发 [J]. 计算机学报,2009,32(5):982-991.

[31] Xue Jinyun. Genericity in PAR platform [C]//International Workshop on Structured Object-Oriented Formal Language and Method. Springer, Cham, 2015: 3-14.

[32] 左正康,薛锦云. Apla 中泛型约束机制研究 [J]. 软件学报,2015,26(6):1340-1355.

[33] Ofenbeck G, Rompf T, Püschel M. Staging for generic programming in space and time [C]//The ACM SIGPLAN International Conference. ACM, 2017: 15-28.

The Contrastive Study of Generic Programming in Object-Oriented Languages

ZHOU Weixing[1], ZUO Zhengkang[1*], WANG Changjing[1], SHI Haihe[1], YOU Zhen[2], XIE Wuping[2], TAO Xiaoming[1]

(1. College of Computer Information Engineering, Jiangxi Normal University, Nanchang Jiangxi 330022, China;
2. Provincial Key Laboratory High Performance Computing, Jiangxi Normal University, Nanchang Jiangxi 330022, China)

Abstract: Generic programming improves the scalability, reusability, and type safety of software by abstracting the concrete algorithms and data structures from a higher level. It is mainly based on a series of custom type constraints rather than just predefined type. Five different object-oriented programming languages are compared comprehensively in this article that support generic programming such as C++, Concepts C++, Java, C#, and Python, and perform a typical example that can better support generic programming mechanisms to analysis the characteristics of these object-oriented languages and the degree of support, so that can explore the nature features of generic programming, and provide references for reducing the development of redundant programs and enhancing the maintainability of the program.

Key words: generic programming; type constraints; object-oriented language; generic mechanism

(责任编辑:冉小晓)