泛型和继承是现代编程语言中两种比较重要的特性，对提高语言的表达能力，增强软件的质量、健壮性、可维护性有重要作用。前者常见于函数式编程语言，如 *Haskell*；后者则是面向对象（*OO*）语言的基础。泛型对类型的描述更细化，表达能力更强，然而，泛型是编译期的信息，无法提供像继承中的动态绑定功能，这也许是过去二十年中 *OO* 语言得到广泛使用的原因。

之所以说泛型不能实现动态的效果，主要原因在于：

```
1） 泛型信息仅仅在编译期起作用

2） 泛型不支持类型的向下转换（Down Casting），即子类对象转换成作为父类型使用
```

这使得泛型在实现类似虚函数的多态时，无法实现或者极为麻烦。考虑如下的情况

```
class Animal:
    method eat ...


class Cat inherite Animal:
    method eat ...


class Dog inherite Animal:
    method eat ...
```

这在 *OO* 中是很常见、很基本的。然而如果用泛型实现，则很麻烦。比如用 *Pattern Matching*

```
func eat animal:
    Cat cat = ...
    Dog dog = ...
```

调用时

```
eat(cat, ...)

eat(dog, ...)
```

似乎也可以。但是，如果 cat 或 dog 是由某个 Factory 根据配置文件动态产生的，也就是

```
cat = Factory.create_animal ...
```

现在 create_animal 的返回值类型如何写？显然，由于泛型中不能将 Cat 和 Dog 都转换成父类 Animal 来处理，就无法在运行时依据 animal 的实际类型，调用对应的函数；而必须在编译期确定所有的类型和应当调用的函数。

无法进行动态绑定，也就无法进行软件的动态扩展。比如，在 Java 中，上述代码已经打包成 Jar 包，现在要增加一个类型 Pig，我们只需让 Pig 继承 Animal，将新代码打成 Jar 包即可使用，无需对原有的代码进行改动和编译。而对于泛型的版本而言，不但无法实现动态扩展，而且还要修改原始的 eat 函数，加入 Pig 对应的 Pattern 代码。

这一特性使得 OO 语言能够很好的支持“开闭”原则，即代码对扩展开放，对修改封闭。通常人们认为，OO 的优势在于对现实世界中“对象”的模拟，但是我更认同松本行弘的观点，即：如同结构化编程一样，OO 是一种代码组织的方式，使得软件开发中的复杂度能够得到更好的控制，至于它是否模拟了世界，并不重要。

在我开来，OO 的作用是把接口和实现分离，并且将实现的函数体拆分到了多处。在上述例子中，增加了 Pig 类型，实际上是在 eat 函数中增加了功能，能够处理 Pig 类型的变量。

从另一个角度说，多态的 eat 函数等价于

```
eat animal, ...:


    if animal instance of Cat:
```

```
    ...
elif animal instance of Dog:
    ...
```

加入了 *Pig* 类型，等于增加了一个 *if* 分支：

```
elif animal instance of Pig:
    ...
```

而这种增加，既没有改变接口，也不需要修改原来的代码，而是通过类的继承。所以，**通过类型继承实现的多态，在不改变接口的前提下，把实现函数的函数体根据具体类型拆分到了多处，并且可以增加新的部分，而不影响原有部分。**这显然就是对“开闭”原则的实践。

“开闭”原则对提高软件的可扩展性，控制复杂性有重要作用，在大型软件开发中尤为明显。

过去 *20* 年间，*C++*、*Java* 等 *OO* 语言获得了广泛使用。而 *Haskell*、*Lisp*、*Erlang* 等语言尽管更清晰、简洁，有更好的数学基础，或者并发的效率更高，但是却没有获得广泛的普及。仅仅把原因归咎于曲高和寡是不够的。在作者看来，这些函数式语言因为无法提供类型继承和动态绑定的功能，导致不能很好的支持开闭原则，在大型软件开发中不能很好的控制复杂度，是它们没有获得广泛应用的主要原因，尤其是在应对需求复杂多变的场合方面。

C++用模板来实现泛型编程，模板分为函数模板和类模板。

基本概念：泛型编程范式 GP：模板也叫参数类型多态化。泛型在编译时期确定，相比面向对象的虚函数多态，能够有更高的效率。

泛型编程是从一个抽象层面描述一种类型的算法，不管容器类型是什么，是一种不同于 OOP 的角度来抽象具体算法。

C++0X 目前对 GP 的支持的趋势来看，确实如此，auto/varadic templates 这些特性的加入象征着 C++ GP 的形式正越来越转向一种更纯粹的泛性语法描述。

GP 的一个主要的弱点就是它不是二进制可复用的，源码基本是公开的，因为编译时候才决定具体类型，决定了类型后就不能更改了不像 OOP 一样拥有关闭开放的原则。

本文主要对函数模板和类模板，在定义语法，模板内部使用泛型类型和非泛型类型，模板的声明和定义分离实现方法，外部客户调用泛型和非泛型参数类型的模板方法，这四个方面比较系统的总结了 C++中泛型的使用经验，博客前面的文字有些凌乱建议想了解本文讲述的泛型使用经验的读者把代码拷贝下来，运行一遍分析代码的用意。

一、函数模板

1. 函数模板的定义

1.1 模板的定义，用 template 和 typename 声明后，函数内直接使用；函数不支持 template 参数列表是空的，类重复具体定义时才支持。

1.2. 非泛型类型参数，直接使用就可以了，可以传入该类型的常量或者变量

关于形参：如果是类型形参，我们就知道该形参表示未知类型，如果是非类型形参，我们就知道它是一个未知值。

2. 函数模板内部使用泛型和非泛型类型

泛型类型定义后，函数内部直接使用即可,类型变量符合作用域规则，包括不可重定义或覆盖，可见作用域。非泛型类型直接使用。

3. 函数模板声明和实现分离原因和做法

将 Template 函数的声明和实现分离，声明在.h 中，实现在.cpp 中，其中.h 需要包含.cpp；类模板中不能这样做，需要特殊处理。

需要分离的原因有：

真正原因：

1.简化条理化代码，实现声明和实现分离，高内聚低耦合，对外提供简单的接口，对内高内聚 。

几乎不会出现，STL 都没有分离：

2.将模板类的声明与实现都放在.h 中(在多个 cpp 中使用不同模板参数时可能会引起重复定义的编译错误)。

4. 客户对函数模板的使用

4.1 实参推演和类型匹配

实参推演是：模板函数不支持类型显式实例化声明的，直接用实参变量实例化调用就好。

类型匹配是：一种类型的只能是一种类型，不能两种类型传入一种类型的模板函数中，是类型引用别名的不能用实例值传入。

4.2 非泛型类型参数，直接使用就可以了，可以传入该类型的常量或者变量

二、类模板

1.类模板的定义

1.1.定义类模板语法

例如：

template<class 形参名，class 形参名，...> class 类名

1.2.非泛型类型参数，直接使用就可以了，可以传入该类型的常量或者变量

关于形参：如果是类型形参，我们就知道该形参表示未知类型，如果是非类型形参，我们就知道它是一个未知值。

1.3 针对类模板声明附加特性，模板子类继承半具体化的模板父类，以及子类调用父类函数：

1）.模板类的继承，继承还是和正常类一样的

例如：

```
template< typename T, int N >
class TinyVector
{
public:
    TinyVector();
    TinyVector( const T& value );

    template< typename T2 >
    TinyVector( const TinyVector<T2,N>& v );
    ...
}

template< typename T >
class Point3D : public TinyVector<T,3>
{
public:
    Point3D() {}

    Point3D( T x, T y, T z )
    {
        (*this)(0) = x;
        (*this)(1) = y;
        (*this)(2) = z;
    }

    Point3D( const TinyVector<T,3>& v )
        : TinyVector<T,3>(v)
```

```
    {
    }

    const T& x() const { return this->at(0); }
    const T& y() const { return this->at(1); }
    const T& z() const { return this->at(2); }
    T& x() { return (*this)(0); }
    T& y() { return (*this)(1); }
    T& z() { return (*this)(2); }
};
```

2）模板子类继承半具体化的模板父类

模板不能是空模板参数类型，但是声明了正常的声明了一个模板类后，如果要对这个模板类进行更加具体的限定那么可以重新定义这个模板的细节，这个时候模板参数类型可以为空，但是这个时候重定义的名称一定要相同，使用的模板参数也是要定义了的。

例如 template<> class BinaryNumericTraits1 会报错，template<> class BinaryNumericTraits<T1, double>也会报错，正确定义如下：

```
template< typename T1, typename T2 >
class BinaryNumericTraits
{
public:
 typedef T1 OpResult;
};

template<>
class BinaryNumericTraits<int, double>
{
public:
 typedef double OpResult;
};

template<>
class BinaryNumericTraits<double, int>
{
public:
 typedef double OpResult;
```

```
};
```

3）子类调用父类函数方式

```
// 继承中调用父类的方法  Array<T,2>::operator= ( Array<T,2>(value_) );

template <typename T, int N>
class Array
{
public:
 Array()
 {
   for(int i = 0; i < N; i++)
   {
    data[i] = 0;
   }
 }
 Array(T t1)
 {
   for(int i = 0; i < N; i++)
   {
    data[i] = t1;
   }
 }
 int GetSize() {return N;}
 T GetValue(int nIndex ){ return data[nIndex];}
private:
 T data[N];
};

#include <iostream>
using namespace std;

template <typename T>
class Array2D: public Array<T,2>
{
public:
```

```
  Array2D( int m, int n );

  void Display()
  {
   cout<<"Array2D Value:";
   for(int i = 0; i < GetSize(); i++)
   {
     cout<<GetValue(i)<<",";
   }
   cout<<endl;
  }
};

template <typename T>
Array2D<T>::Array2D(int m, int n): Array<T,2>()
{
 int value_ = m *   n;

 // 这个语法其实和 Array<T,2>(value_) 是一样的

  Array<T,2>::operator= ( Array<T,2>(value_) );
}
```

调用端：

```
 Array2D<int> arrayObj(1,2);
 arrayObj.Display();
```

2. 类模板内部使用泛型和非泛型类型，类外定义函数成员

2.1 使用类模板的类型声明,包括数据成员，和函数参数、函数返回值都可以直接使用泛型类型。类型变量符合作用域规则，包括不可重定义或覆盖，可见作用域。
非泛型类型直接使用。

2.2 在类声明外定义类函数（在 **CPP** 中也是可以的，但是包含情况就会发生变化）,格式如下：

```
template<模板形参列表> 函数返回类型 类名<模板形参名>::函数名(参数列表){函数体}
// 例如
```

```
template< typename T, int N >
class TinyVector
{
public:
    // 模板类里面的正常函数
    TinyVector( const T& value );

    // 模板类里面含有模板函数
   template< typename T2 >
    TinyVector<T,N>& operator= ( const TinyVector<T2,N>& v );
}
```

外部实现：

```
// 模板类的正常函数
template< typename T, int N >
TinyVector<T,N>::TinyVector( const T& value )
{
    for ( int i = 0; i < N; ++i ) {
        data_[i] = value;
    }
}

// 模板类里面的模板函数
template< typename T, int N >
template< typename T2 >
TinyVector<T,N>& TinyVector<T,N>::operator= ( const TinyVector<T2,N>& v )
{
    for ( int i = 0; i < N; ++i ) {
        data_[i] = v(i);
    }
    return *this;
}
```

3. 关于模板声明和实现分离后的调用原理

模板不能分离的原因：C++标准明确表示，当一个模板不被用到的时侯它就不该被实例化出来，cpp 中没有将模板类实例化，所以实际上.cpp 编译出来的 t.obj 文件中关于模板实例类型的一行二进制代码，于是连接器就傻眼了，只好给出一个连接错误。所以这里将.cpp 包含进来，其实还是保持模板的完整抽象，main 中可以对完整抽象的模板实例化。

进行分离的方法：

类的声明必须在实现的前面，所以使用的时候包含 cpp 就可以把.h 和.cpp 中定义的内容一并包含了，其实是放在同一个文件中一个意思。

这种模板分离的方式缺点：如果这个模板类的 cpp 有非模板的定义，能够有效实例化，但是会导致重复包含定义而出错所以有些模板的实现和分离放到了.tcc 格式或者.inl 格式的文件中(这些格式的文件都是来自于 txt 文本的不能从 cpp 直接修改得到)，在.h 中直接包含进去,代码中就可以直接包含 ClassTemplate.h 了，避免包含.cpp 奇怪的行为。

例如：

#include "Test.tcc"

4. 客户对类模板的使用

4.1 模板类的使用，类对象显式类型声明不支持实参推演，类的成员函数要求实参推演和类型匹配。

4.2 非泛型类型形参，一般是简单类型，非泛型类型参数，直接使用就可以了，可以传入该类型的常量或者变量。

三、关于函数模板和类模板的定义、定义使用、声明和定义分离，客户使用的规则的测试代码

上面的总结代码中都用，测试工程中包含 6 个文件，分别为：

FunctionTempplate.h

FunctionTempplate.cpp

ClassTemplate.h

ClassTemplate.tcc

mian.cpp

main2.cpp（测试特性用的）

FunctionTempplate.h

[cpp] view plain copy print?

```
/*0.基本概念：泛型编程范式 GP：模板也叫参数类型多态化。
在编译时期确定，相比面向对象的虚函数多态，能够有更高的效率。
泛型编程是从一个抽象层面描述一种类型的算法，不管容器类型是什么，是一种不同于 OOP 的角度来抽象具体算法。
C++0X 目前对 GP 的支持的趋势来看，确实如此，auto/varadic templates 这些特性的加入象征着 C++ GP 的形式正越来越转
向一种更纯粹的泛性语法描述。
GP 的一个主要的弱点就是它不是二进制可复用的，源码基本是公开的，因为编译时候才决定具体类型，决定了类型后就不能更改了不像 OOP 一样拥有关闭开放的原则。
*/
#ifndef FUNCTIONTEMPPLATE_H
#define FUNCTIONTEMPPLATE_H
// 1.1 模板的定义和实现，template 和 typename 声明后，函数内直接使用；函数不支持 template 参数列表是空的，类才支持
#include "FunctionTempplate.cpp"
template<typename T1,typename T2>
void SwapTwoType(T1 &t1, T2 &t2)
{
    T1 temp = t1;
    t1 = (T1)t2;
    t2  = (T2)temp;
}


// 1.2. 非泛型类型参数，直接使用就可以了，可以传入该类型的常量或者变量
// 关于形参：如果是类型形参，我们就知道该形参表示未知类型，如果是非类型形参，我们就知道它是一个未知值。
template<typename T>
bool TestOver(T t1, int t2)
{
    return int(t1) > t2;
}

// 2.泛型类型定义后，函数内部直接使用即可,类型变量符合作用域规则，包括不可重定义或覆盖，可见作用域
template<typename T>
void SwapOneType(T &t1, T &t2)
{
    T temp = t1;
    t1 = t2;
    t2  = temp;
}

```

```
// 函数不支持 template 是空的，类才支持，也不支持 SwapImp<double &t1, int &t2>这样的写法
//template<>
//void SwapImp<double &t1, int &t2>
//{
//  double temp = t1;
//  t1 = t2;
//  t2 = (int)temp;
//};

// 3.函数模板声明和实现分离原因和做法

// 将 Template 函数的声明和实现分离，声明在.h 中，实现在.cpp 中，其中.h 需要包含.cpp
// 需要分离的原因有：
//真正原因：1.简化条理化代码，实现声明和实现分离，高内聚低耦合，对外提供简单的接口，对内高内聚
//几乎不会出现，STL 都没有分离：2.将模板类的声明与实现都放在.h 中(在多个 cpp 中使用不同模板参数时可能会引起重复定义的编译错误)

template<typename T>
void SwapOneTypeImp(T &t1, T &t2);

#endif
```

FunctionTempplate.cpp

[cpp] view plain copy print?

```
template<typename T>
void SwapOneTypeImp(T &t1, T &t2)
{
    T temp = t1;
    t1 = t2;
    t2  = temp;
}
```

ClassTemplate.h

[cpp] view plain copy print?

```
#ifndef CLASSTEMPLATE_H
#define CLASSTEMPLATE_H

// 1.1.声明类模板，例如：
```

```
// template<class  形参名，class 形参名，…>   class 类名
template<typename T>
class CMathOperation
{
// 2.1 使用类模板的类型声明,包括数据成员，和函数参数、函数返回值都可以直接使用泛型类型
// 类型变量符合作用域规则，包括不可重定义或覆盖，可见作用域
public:
    T Add(T &t1, T &t2)
    {
        return t1 + t2;
    }
    T Multi(T &t1, T &t2);

    T Min(T &t1, T &t2);

    // 1.2.非泛型类型参数，直接使用就可以了，可以传入该类型的常量或者变量
    // 关于形参：如果是类型形参，我们就知道该形参表示未知类型，如果是非类型形参，我们就知道它是一个未知值。
    bool  bOverMax(T &t1, int maxNum);
private:
    T a;
    T b;
};

// 2.2 在类声明外定义类函数（在 CPP 中也是可以的，但是包含情况就会发生变化）,格式如下：
// template<模板形参列表> 函数返回类型 类名<模板形参名>::函数名(参数列表){函数体}
template<typename T> T CMathOperation<T>::Multi(T &t1, T &t2)
{
    return t1*t2;
}

// 3.这种模板分离的方式缺点：如果这个模板类的 cpp 有非模板的定义，能够有效实例化，那么会导致重复包含定义而出错
// 所以有些模板的实现和分离放到了.tcc 格式或者.inl 格式的文件中(这些格式的文件都是来自于 txt 文本的不能从 cpp 直接修改得到)，
// 在.h 中直接包含进去,代码中就可以直接包含 ClassTemplate.h 了，避免包含.cpp 奇怪的行为;这种方式编辑时候需要修改下后缀。
#include "ClassTemplate.tcc"

// 1.3 针对类模板声明附加特性：
```

```
41.         // 模板不能是空模板参数类型，但是声明了正常的声明了一个模板类后，如果要对这个模板类进行更加具体的限定那么可以
42.         // 重新定义这个模板的细节，这个时候模板参数类型可以为空，但是这个时候重定义的名称一定要相同，使用的模板参数也是要定义了的。
43.         // 例如 template<> class BinaryNumericTraits1 会报错，template<> class BinaryNumericTraits<T1, double>也会报错，正确定义如下：
44.         template< typename T1, typename T2 >
45.         class BinaryNumericTraits
46.         {
47.         public:
48.             typedef T1 OpResult;
49.         };
50.
51.         template<>
52.         class BinaryNumericTraits<int, double>
53.         {
54.         public:
55.             typedef double OpResult;
56.         };
57.
58.         template<>
59.         class BinaryNumericTraits<double, int>
60.         {
61.         public:
62.             typedef double OpResult;
63.         };
64.
65.         #endif
```

ClassTemplate.tcc

[plain] view plain copy print?

```
1.         // 类模板可以直接在.h 中声明，在文本文件中定义
2.         template<typename T>
3.         T CMathOperation<T>::Min(T &t1, T &t2)
4.         {
5.             return t1 > t2 ? t2 : t1;
6.         }
7.
8.         template<typename T>
9.         bool CMathOperation<T>::bOverMax( T &t1, int maxNum )
10.        {
11.            if( t1 > maxNum )
12.            {
```

```
        return true;
    }
    return false;
}

// 如果这个模板类的 cpp 有非模板的定义，能够有效实例化，那么会导致重复包含定义而出错
// 所以有些模板的实现和分离放到了.tcc 格式或者.inl 格式的文件中,在.h 中直接包含进去
//int g_TestValue = 100;
//bool IsNegtiveValue()
//{
//  if( g_TestValue < 0 )
//  {
//      return true;
//  }
//  return false;
//}
```

mian.cpp

[cpp] view plain copy print?

```
#include "FunctionTempplate.h"
// 3.关于模板声明和实现分离后的调用原理
// 类的声明必须在实现的前面，所以使用的时候包含 cpp 就可以把.h 和.cpp 中定义的内容一并包含了，其实是放在同一个文件中一个意思
// 原因是：C++标准明确表示，当一个模板不被用到的时侯它就不该被实例化出来，
// cpp 中没有将模板类实例化，所以实际上.cpp 编译出来的 t.obj 文件中关于模板实例类型的一行二进制代码，
// 于是连接器就傻眼了，只好给出一个连接错误。
// 所以这里将.cpp 包含进来，其实还是保持模板的完整抽象，main 中可以对完整抽象的模板实例化。

// 这种模板分离的方式缺点：如果这个模板类的 cpp 有非模板的定义，能够有效实例化，那么会导致重复包含定义而出错
// 所以有些模板的实现和分离放到了.tcc 格式或者.inl 格式的文件中(这些格式的文件都是来自于 txt 文本的不能从 cpp 直接修改得到)，
// 在.h 中直接包含进去,代码中就可以直接包含 ClassTemplate.h 了，避免包含.cpp 奇怪的行为。
#include "ClassTemplate.h"
#include <iostream>
using namespace std;

int n1,n2;
double dValue1, dValue2;
void ResetTestData()
```

```
{
    n1 = 10;
    n2 = 100;
    dValue1 = 1.5f;
    dValue2 = 16.5f;
}

void DisplayTestData(int nCount)
{
    cout<<"---------------Result of count: "<<nCount<<"-------------"<<endl;

    cout<<"n1= "<<n1<<",n2= "<<n2<<",dValue1= "<<dValue1<<",dValue2= "<<dVal
ue2<<endl;
}

extern void TestSwapFunc();
void main()
{
    // 一、函数模板使用
    cout<<"-------------函数模板使用------------"<<endl;
    TestSwapFunc();
    ResetTestData();
    // 4.1 函数模板的使用，实参推演和类型匹配
    // 实参推演是：模板函数不支持类型显式实例化声明的，直接用实参变量实例化调用就
好。
    // 类型匹配是：一种类型的只能是一种类型，不能两种类型传入一种类型的模板函数中，
是类型引用别名的不能用实例值传入。
    //SwapOneType(<int>(n1), <int>(n2));语法错误 : 缺少“)”(在“<”的前
面),Swap(T1 &,T2 &)”: 应输入 2 个参数，却提供了 0 个
    //SwapTwoType(<int>(n1), <int>(n2)); 语法错误 : 缺少“)”(在“<”的前
面),Swap(T1 &,T2 &)”: 应输入 2 个参数，却提供了 0 个
    //SwapOneType(2, 3);不能将参数 1 从“int”转换为“int &”
    //SwapTwoType(2, 3);不能将参数 1 从“int”转换为“int &”

    SwapTwoType(n1, n2); // OK
    DisplayTestData(1);

    //SwapTwoType(2, 3); // 错误，SwapTwoType”: 不能将参数 1 从“int”转换为
“int &”

    ResetTestData();
    SwapTwoType(dValue1, dValue2); // OK，直接实例化
    DisplayTestData(2);
```

```
56.         ResetTestData();
57.         SwapTwoType(n1, dValue1);// 也 OK,直接实例调用因为是支持两个类型的，只是
    dValue1 的 1.5f 转换为 n1 时候被截取了整型变为了 1
58.         DisplayTestData(3);
59.
60.         ResetTestData();
61.         //SwapOneType(n1, dValue1);// 编译错误，因为只有一个类型，
    void SwapOneType(T &,T &)”: 模板 参数“T”不明确
62.         SwapOneType(dValue1, dValue2); // OK，同样的一个类型实例化
63.         DisplayTestData(4);
64.
65.         ResetTestData();
66.         SwapOneTypeImp(dValue1, dValue2);
67.         DisplayTestData(5);
68.
69.         // 4.2 非泛型类型参数，直接使用就可以了，可以传入该类型的常量或者变量
70.         int nCurNum = 10;
71.         int nMaxValue = 100;
72.         bool bTestOver = TestOver(nCurNum, nMaxValue);
73.
74.         // 二、类模板使用
75.         // 4.1 模板类的使用，类对象显式类型声明不支持实参推演，类的成员函数要求实参推演
    和类型匹配
76.         cout<<"-------------类模板使用------------"<<endl;
77.         CMathOperation<int> oprObj; // OK,必须要显式类型指定，不能用函数模板的实参推
    演
78.         int a = 10;
79.         int b = 15;
80.         int nAddResult = oprObj.Add(a,b); // OK,类成员函数参数不能使用显式类型指定
    了
81.         int nMultiResult = oprObj.Multi(a,b);// OK,类外部定义也是可以的
82.
83.         //int nAddResult2 = add.Add(<int>(a),<int>(b));// NO 显式类型指定是错误
    的
84.         //float fTestValue = 15.0f;
85.         //int nAddResult = add.Add(a, fTestValue);// NO 和函数模板一样也是要求类型
    匹配的
86.
87.         /*float fValue1 = 1.5f;
88.         float fValue2 = 3.5f;
89.         float fRes3 = oprObj.Add(fValue1, fValue2);*/ // NO 对象是 int 类型的，因为
    类型匹配，故其它类型的参数传入报错
90.
91.         int minNum = oprObj.Min(a,b);
```

```
92.
93.     // 4.2 非泛型类型形参，只能是简单类型
94.     int nValue = 10;
95.     bool bOver = oprObj.bOverMax(nValue, 100); // OK
96.     int nMaxValue1 = 100;
97.     bool bOver1 = oprObj.bOverMax(nValue, nMaxValue1); // 也是 OK 的，只要类型匹配就可以了
98.
99.     // 非泛型类型参数，直接使用就可以了，可以传入该类型的常量或者变量
100.    const int nMaxValue2 = 100;
101.    bool bOver2 = oprObj.bOverMax(nValue, nMaxValue2); // 也是 OK 的，只要类型匹配就可以了
102.
103.    while(1);
104. }
```

main2.cpp（测试特性用的）

[cpp] view plain copy print?

```
1.  //本文件主要针对于不同的模板传入不同的参数会导致编译重定义问题
2.  // 几乎不会出现，STL 都没有分离：
3.  // 将模板类的声明与实现都放在.h 中(在多个 cpp 中使用不同模板参数时可能会引起重复定义的编译错误)
4.  #include "FunctionTempplate.h"
5.  // 将模板 cpp 多次包含是不会参数多重包含的，因为编译时候这个模板的 cpp 并不能有效的实例化。
6.  //#include "ClassTemplate.tcc"
7.  void TestSwapFunc()
8.  {
9.      float fTestValue1 = 1.0f;
10.     float fTestValue2 = 1000.0f;
11.     double dValue3 = 10.001f;
12.     SwapOneType(fTestValue1, fTestValue2);
13.     SwapTwoType(fTestValue1, dValue3);
14. }
```

我们知道，程序员的朝三暮四，虽说不一定都是本性，但也几乎成了一种习惯。每当出现一种新的语言，很多程序员也不顾一切地投怀送抱。不管这种语言是好是坏，进步还是退步。尽管我们无法指责这种行为，就像无法把阿尔马维瓦伯爵告上法庭。但是，事实的真相还是应当昭示与天下的。

牢骚就不再多发了。今天的主题是 GP。不，不是电池，也不是摩托比赛。全称是 Generic Programming，泛型编程。一种非常强大，却有为人们所忽略的关键性技术。这种技术代表的是未来。

不，不要提 **Java**、C#。他们的那种也叫 Generic（泛型）的东西，和真正的 GP 沾不上什么边。只能算作大号的 OOP，一会儿就会知道为什么我这么说。

本文起源于 TopLanguage（http://groups.google.com/group/pongba）上的一次讨论（http://groups.google.com/group/pongba/browse_thread/thread/e553a21476ba2ebd）。我在讨论中做了一个案例，以说明 GP 的作用。现在我把这个案例整理出来，一同探讨。这个讨论还涉及了更深层次的理论和技术，其余内容看那个帖子。

案例提出这样一个需求：

搞过帐务系统或者学过财务的都应该知道，帐务系统里最核心的是科目。在中国，科目是分级的，（外国人好像没有法定的分级体系），一般有 4、5 级，多的有 7、8 级，甚至 10 多级。不管怎么分，科目的结构是树。

在科目上有一个操作称为"汇总"，就是把子科目的金额累加起来，作为本科目的金额。这实际上是对指定科目的所有下级科目的遍历汇总。这是一个非常简单，但却非常重要的帐务操作。

首先，我通过两种方法：OOP 和传统的 SP 来实现这种操作。先来看 SP：

```
//科目类，具备树形结构

class Account

{

public:

   typedef vector<Account> child_vect;

pubilc:

   child_vect children();   //子级科目

   int account_type();       //本科目类型
```

```
    float ammount();          //本科目的凭证金额累计，非最明细科目返回 0
};
//汇总算法
double collect(Account& item) {
 double result(0);
    Account::child_vect& children=item.children();
    if(children.size==0)          //最明细科目，没有子科目
        return   ammount(item.ammount);
    Account::child_vect::iterator ib(children.begin()), ie(children.end());
    for(; ib!=ie; ++ib)
    {
        result+=collect(*ib, filter, ammount);
    }
  return result;
}
```

当我需要对一个科目对象 acc_x 执行汇总算法，那么就是这样：

```
double res=collect(acc_x);
```

这非常简单。不过请注意，我这里还是利用了 OOP 的封装机制，为了使 Account 的实现和接口分离。但所使用的算法/数据分离的模式，则是 SP 风格的。

OOP 风格的更加简单：

```
class Account
```

```
{
public:
    child_vect children();   //子级科目
    int account_type();            //本科目类型
    double ammount() {              //此成员直接执行子级科目的汇总任务
        double result(0);
        if(children.size==0)     //最明细科目，没有子科目
            return   m_ammount;   //或从其他途径获得，如数据库访问
        T::child_vect& children=item.children();
        T::child_vect::iterator ib(children.begin()), ie(children.end());
        for(; ib!=ie; ++ib)
        {
            result+=ib->ammount();
                }
                return result;
    }
};
```

使用起来是这样：

```
double res=acc_x.ammount();
```

都很好。不过，现在项目增加了需求，我们面临挑战：

假设现实世界的古怪客户，使我们面临一个挑战：他们的业务模型中，有部分科目不参与汇总计算，是一群特殊的科目。（这种科目我还真见过）。

那么，SP 方式，可以另写一个 collect 函数：

```
double collect(Account& item) {
  //假设 g_SpecialAccounts 是个 Singleton，负责管理特殊科目
  if(g_SpecialAccounts.IsSpecial(item->account_type()))
      return 0
  … //其余代码与原来的 collect 相同
}
```

而 OOP 方式，则需要修改 Account 类（也可以利用重载和多态）：

```
class Account
{
public:
  double ammount() {
  //假设 g_SpecialAccounts 是个 Singleton，负责管理特殊科目
  if(g_SpecialAccounts.IsSpecial(account_type())
    return 0;
  }
  … //其余代码与原来的 Account 相同
};
```

相比之下，SP 更灵活些。如果我没有 Account 的源码，或者我无法修改 Account，那么我可以直接重写一个 collect（比如，collect_x）也能解决问题。而在这种情况下，OOP

方式，只能重载或重写这个类。（重载不仅仅需要重写相关成员，而且还需要编写诸如构造函数等辅助代码）。更深层次的因素，是代码耦合的问题。关于这个问题请看前面给出的那个讨论。

接下来的一个需求，则提出了更大的挑战：

我们如果注意的话，MIS 系统中有很多地方同科目有着相同的逻辑结构。比如，销售部门的分销组织机构，一个企业的部门组织机构。在这些结构上，通常也会发生汇总操作，比如某个省的分销商业绩汇总，或者某个部门的人数汇总。

于是，充满优化意识的程序员，会想到复用在帐务系统上已有的成果。假设我们定义了部门类：

```
class Department

{

public:

   typedef vector<Account> child_vect;

public:

   child_vect Children();

   int dept_type();

   int employee_num();

    ...

};
```

对于 SP 方案而言，意味着需要写一个 collect 算法，可以同时用于这两个（甚至更多）的类型。我们努力地尝试着：

```
double collect_g(void* item, bool (*pred)(void*), void (*mem)(void*, void *)) {

  if(pred(item))
```

```
        return 0;
    double result(0);
     vector<void*>& children=item.children();
     if(children.size==0)          //最明细科目，没有子科目
    {
        mem(item, &result);
          return   result;
    }
     vector<void*>::iterator ib(children.begin()), ie(children.end());
     for(; ib!=ie; ++ib)
     {
          result+=collect(*ib, pred, mem);
     }
    return result;
}
```

此外，需要为 Account 和 Department 分别编写两个辅助函数：

```
//Account
bool Account_Pred(void* item) {
    Acount* acc_=(Account*)item;
    return  g_SpecialAccount.IsSpecial(acc_);
}
```

```
void Account_Ammount(void* item, void* val) {
   Acount* acc_=(Account*)item;
   *((double*)val)=acc_->ammount();
}
//Department
bool Dept_Pred(void* item) {
   Department* dpt_=(Department*)item;
   return  g_SpecialDepartment.IsSpecial(dpt_);
}
void Dept_EmpNum(void* item, void* val) {
   Department* dpt_=(Department*)item;
   *((int*)val)=dpt_->Employee_Num();
}
```

用起来，则是这样：

```
double res1=collect(&acc_x, &Account_Pred, &Account_Ammount);
int res2=collect(&acc_x, &Dept_Pred, &Dept_EmpNum);
```

对于 OOP 方案而言，则没有那么简单了。我们尝试着一步步来：

首先，应设法将算法分离。按照标准的 OOP 思路，用一个接口封装算法：

```
class ICollect
{
public:
```

```
  virtual ??? cacul(??? item)=0;
};
class AccountCollect : publie ICollect
{
public:
  AccountCollect
  virtual ??? cacul(??? item){…};
};
class DepartmentCollect : publie ICollect
{
public:
  virtual ??? cacul(??? item){…};
};
```

但是，我们立刻发现了问题，这些???的地方填什么？由于 Account 上需要汇总 double 类型的 ammount，而 Department 上需要汇总人数，是 int 类型。两种实现的 cacul()的类型不同。为解决问题，使用 variant。参数 item 对应 Account 和 Department，是不同的类型。解决的办法，就是使 Account 和 Department（以及其他想共享算法的类型）都实现同一个接口：

```
class Account : public IData{…};
class Department : public IData {…};
```

于是，ICollect 的接口变成：

```
class ICollect
```

```
{
public:
  virtual variant cacul(IData* item)=0;
};
```

当然，还有一些细节上的问题还没解决，比如，IData 中定义那些成员，如何处理类型问题等等。这里限于篇幅，不再深究了。

到目前为止，两种方案中，SP 明显比 OOP 来的简洁灵活。原因还是因为耦合。OOP 为了解决不同类型的共享算法问题，不得不通过接口一层一层地将类型归一化。其结果就是形成肥厚的“粘合层”（详见那个讨论）。SP 天然地将算法和数据分离，并且通过强制类型转换将类型归一化。（尽管 OOP 也可以通过强制类型转换归一化类型，但受制于类的集成结构，也无法象 SP 那样灵活）。但是，强制类型转换所带来的负面作用，是人所共知的。

两种方案，都有共同的一个特点，就是类型归一化。SP 方案的 collect 算法必须将类型归一化为 void*，并在计算时转换回来。而 OOP 方案的 ICollect 和 IData 接口也都是为了归一化不同的类型而“强行”添加的。

针对这个问题，我们可以做一个想象（不是白日梦）：如果算法不是只接受一种类型，而是可以接受一族类型的实例，而这些类型具备某些共有的特征。那么我们就无需象 SP 那样强制类型转换，或象 OOP 那样凭空增加一堆接口。

这就是 GP 的工作。

下面，我利用 GP 来改进 SP 和 OOP 的实现。首先是 SP：

在改进版的 SP 实现中，Account 和 Department 都没有什么变化，变动的是 collect 算法：

```
template<typename T, typename Pred, typename Sub, typename M>
typename M::return_type collect(T& item, Pred p, Sub s, M m) {
  if(p(item)==false)
      return   0;

  typename M::return_type result(0);
```

```
    T::child_vect& children=s(item);
    if(children.size()==0)
        return m(item);

    T::child_vect::iterator ib(children.begin()), ie(children.end());
    for(; ib!=ie; ++ib)
    {
        result+=collect(*ib, p, s, m);
    }
     return result;
}
```

（需要感谢一下 pongba，他指出了原来算法的一个不足，我按照他的思路改进了算法，进一步减小了算法的耦合度，提高了灵活性）。

先解释一下模板参数：T，被计算的对象的类型，Account 或 Deparment 等等；Pred，判别项目是否参与计算的“可调用物”的类型，可以认为是 delegate；Sub，是用于从被计算对象上提取子级对象的“可调用物”类型；M，用于从被计算对象上获取计算数据的“可调用物”类型。

现在看一下如何使用这个算法：

```
double res1=collect(acc_x, mem_fun(&SpecialAccount::IsSpecial),

    mem_fun(&Account::children), mem_fun(&Account::ammount));

int res2=collect(dpt_x, mem_fun(&SpecialDept::IsSpecial),

    mem_fun(&Department::children), mem_fun(&Department::empl_num));
```

调用很长，比较晦涩，简单地解释一下：前一个调用把一个科目节点作为第一参数；第二个参数用 mem_fun 把特殊科目管理器绑定成一个“可调用物”（关于 mem_fun 参考 stl）；第三个参数包装了获取子级科目的成员函数；第四个参数包装了取得科目金额的成员函数。后一个调用也一样，只是把 Account，及其相关内容，换成 Department。

引入 GP 后，算法 collect 变成了一个计算框架。具体的对象判别，子级获取，数据获取，都参数化。使得这个算法是“可装配的”，根据不同的业务类型，和被计算的成员函数和数据类型，由算法的使用者在使用时“拼装”起来。

相比没有使用 GP 的 SP 方案，这个方案是类型安全的，无需使用强制类型转换。同时，原有的灵活性非但没有降低，反而提高。因为算法的三个 delegate 不再针对一个函数签名的，而是针对一簇函数签名，无需再为成员函数的返回类型变化，而放弃强类型的保护。更进一步，GP 提高了整个算法的复用性，不再需要为每一个 delegate 包装器编写代码，只需复用已存在的 mem_fun 即可。

对于 OOP 而言，GP 可以为其带来更好的效果：

```
//Biz_Alg 类包含了若干常用的算法。做成 traits 形式，便于未来针对特别情况特化。
template<typename T>
struct Biz_Alg
{
   template<typename R, typename C, typename Pred, typename Sub, typename M>
   R collect(C& item, M mem) {
       //执行遍历汇总，调用 item 子节点的 mem 成员
   }

};

 //Account 类和 Department 类

template<typename Alg=Biz_Alg<Account> >

class Account

{

public:

   float ammount() {

      Alg::collect(*this, mem_fun(&SpecialAccount::IsSpecial),

            mem_fun(&Account::children), mem_fun(&Account::ammount));

   }

   …
```

```
};

template<typename Alg= Biz_Alg<Department> >

class Department

{

public:

   float elec_fee() {

        return Alg::collect(*this, &Department::elec_fee);

   }

   int empl_num() {

      Alg::collect(*this, mem_fun(&SpecialDepartment::IsSpecial),

          mem_fun(&Department::children), mem_fun(&Department::ammount));

   }

   …

};
```

使用起来倒是比 GP 化的 SP 方案更简单：

```
float res1=acc_x.ammount();

int res2=dept_x.empl_num();
```

这种方案则是以 OOP 为出发点，利用 GP 技术使其解耦，并组件化。而前一个方案则是以 SP 为出发点，利用 GP 使其泛化和组件化。也就是说，GP 被分成了两种风格：OO 风格的 GP 和 SP 风格的 GP。从本质上来说，两者是等价的，都是将算法和数据对象分离。只不过前者是以类为主导，而后者则是以自由函数为主导。两者各有优势：前者集成度高，使用起来简便、清晰，更符合业务逻辑；后者更灵活，耦合度更小，扩展性更强。

根本上而言，OO 风格的 GP 是以业务逻辑为核心，用类封装具体实现，程序员的注意力焦点都在业务逻辑上。比较适合高层的业务级别开发。而 SP 风格的 GP ，以数据操作为核心，由自由函数将算法施加在数据实体上，程序员的注意力都集中在算法和数据本身上。这种风格比较适合底层的系统级开发。

从原先纯 OO 的方案来看，由于受制于强类型系统的约束，无法很优雅地将算法与数据分离。由此造成越来越严重的耦合。而 GP 的引入，则利用其泛型特性，消除了强类型的负面作用，实现了解耦。

同样，在纯 SP 方案中，算法与数据分离，但却没有完全独立于数据。一个自由函数要么只能针对一种类型（除非放弃类型安全的保证，和代码复用），要么促使数据对象的类型归一化。后者将依旧会引发数据对象间的耦合。但 GP 引入后，算法不再针对单一的类型，而是可以面向一族类型。在这种情况下，类型安全得到保证，代码复用得到实现，也无需强制地将不相关的类型归一化。

至于 Java 和 C#的泛型存在的问题就很明显了。我们利用 GP 费尽心机，使得 Account 和 Department 不再需要实现同一个接口，大大弱化了类间的耦合。但 Java 和 C#的泛型都要求类型参数共同继承自一个基类或接口。这反而迫使我们让类型耦合，做原本力图消除的事。这就是为什么 Java 和 C#的泛型没什么用处的原因。

最后，我这里还有一个方案：

```
template<typename C, typename R, typename Pred，R (C::*mem)() >

class collect

{

public:

   collect(C const& item): m_Item(item), m_Mem(mem){}

   R operator() {

       //执行遍历汇总

   }

private:
```

```
    C&        m_Item;

    Pred      m_Pred;

    R (C::*m_Mem);

}

//用起来像这样

typedef collect<Account, float, AllTrue<Account>,

                &Account::ammount() > AccountCollect;

typedef collect<Account, int, AllTrue<Department>,

                &Department::employee_num > DeptElecFeeCollet;

//使用 SP 风格的 GP 方案中的 Account 和 Department 的对象，即简单类

float res=AccountCollect(acc1)();

float res=AccountCollect(dept1)();
```

这种方式看上去有些古怪。实际上使用一个函数对象封装了算法，构成了一个"算法适配器"。collect 是个算法框架，通过 typedef 将相应的判别式、成员函数，同框架一起组装起来，构成一个类型。之后，直接使用这个类型，适配一个对象，以执行相应的计算。这可以看作是介于 OOP 和 SP 风格之间的一种 GP 方案。它比 OOP 风格方案稍微灵活一点点，比 SP 风格方案使用上稍微简便一点点。但是，在这个方案能为我们带来多少好处，我不敢说。放在这里，只是为了展示 GP 的几种可能的设计方案而已。

总结一下。在这里。我们首先试图用 OOP 和 SP 两种设计风格实现一个比较复杂的任务：让两种完全不同的业务类，共享相同的算法，而且希望获得最大的扩展性。但是，两种方案都不尽如人意。随后，我们引入 GP，非常好的解决了两种方案的问题，却没有牺牲各自的优点。从中，我们可以看出，无论是 OO 还是 SP，都可以通过 GP 来加以扩展，消除各自的缺陷。也就是“用 OO 或 SP 设计，用 GP 来优化（代码）”。