C++ template —— 动多态与静多态（六）

前面的几篇博文介绍了模板的基础知识，并且也深入的讲解了模板的特性。接下来的博文中，将会针对模板与设计进行相关的介绍。

与传统的语言构造相比，模板的不同之处在于：它允许我们在代码中对类型和函数进行参数化。把（1）局部特化和（2）递归实例化组合起来，将会产生强大威力。接下来的几篇博文，我们通过下面的一些设计技术来展示这些强大威力：

（1）泛型编程
（2）trait
（3）policy class
（4）metaprogramming
（5）表达式模板

第14章 模板的多态威力

多态是一种能够令单一的泛型标记关联不同特定行为的能力。对面向对象的程序设计范例而言，多态可以说是一块基石。在C++中，这块基石主要是通过继承和虚函数来实现的。由于这两个机制（继承和虚函数）都是（至少一部分）在运行期进行处理的，因此我们把这种多态称为动多态；我们平常所谈论的C++多态指的就是这种动多态。然而，模板也允许我们使用单一的泛型标记，来关联不同的特定行为；但这种（借助于模板的）关联是在编译期进行处理的，因此我们把这种（借助于模板的）多态称为静多态，从而和上面的动多态区分开来。

14.1 动多态

使用继承和虚函数，在这种情况下，多态的设计思想主要在于：对于几个相关对象的类型，确定它们之间的一个共同功能集；然后在基类中，把这些共同的功能声明为多个虚函数接口。每个具体类都派生自基类，生成了具体对象之后，客户端代码就可以通过指向基类的引用或指针来操作这些对象，并且能够通过这些引用或者指针来实现虚函数的调度机制。也就是说，利用一个指向基类（子对象）的指针或者引用来调用虚成员函数，实际上将可以调用（指针或者引用实际上所代表的）具体类对象的相应成员。这种动多态是C++程序设计里面最常见的，这里不过多的阐述。

14.2 静多态 模板也能够被用于实现多态。如下例子：

```
// poly/statichier.hpp
#include "coord.hpp"

// 具体的几何对象类Circle
// - 并没有派生自任何其他的类
class Circle
{
    public:
        void draw() const;
        Coord center_of_gravity() const;
        ...
};

// 具体的几何对象类Line
// - 并没有派生自任何其他的类
class Line
{
    public:
        void draw() const;
        Coord center_of_gravity() const;
        ...
};
....
```

现在，使用这些类的应用程序看起来如下所示：

```
// poly/staticpoly.cpp
#include "statichier.hpp"
#include <vector>

// 画出任意GeoObj
// method2
template <typename GeoObj>
void myDraw(GeoObj const& obj)      // GeoObj是模板参数
{
    obj.draw();       // 根据对象的类型调用相应的draw()
}
......

int main()
{
    Line l;
    Circle c;

    myDraw(l);          // myDraw<Line>(GeoObj&) => Line::draw()
    myDraw(c);          // myDraw<Circle>(GeoObj&) => Circle::draw()
}

// method1:如果使用动多态，myDraw函数会是如下形式:
void myDraw(GeoObj const& obj)       // GeoObj是一个抽象基类
{
    obj.draw();
}
```

通过比较myDraw()的这两个实现，我们可以看出：主要的区别在于method2的GeoObj的规范是模板参数，而不是一个公共基类。然而，在这个现象的背后，还存在更多本质的差别。例如，使用动多态（method1），我们在运行期只具有一个myDraw()函数，而如果使用模板，我们则可能具有多个不同的函数，诸如myDraw<Line>()和myDraw<Circle>()。

14.3 动多态和静多态

我们来对多态进行分类，并对这两种多态进行比较。

14.3.1 术语

动多态和静多态为不同的C++编程idioms提供了支持：
（1）通过继承实现的多态是绑定的和动态的：
1. 绑定的含义是：对于参与多态行为的类型，它们（具有多态行为）的接口是在公共基类的设计中就预先确定的（有时候也把绑定这个概念称为入侵的或者插入的）。
2. 多态的含义是：接口的绑定是在运行期（动态）完成的。
（2）通过模板实现的多态是非绑定的和静态的：
1. 非绑定的含义是：对于参与多态行为的类型，它们的接口是没有预先确定的（有时也称这个概念为非入侵的或者非插入的）。
2. 静态的含义是：接口的绑定是在编译期（静态）完成的。

14.3.2 优点和缺点

（1）C++的动多态具有下列优点：
1. 能够优雅地处理异类集合；
2. 可执行代码的大小通常比较小（因为只需要一个多态函数，但对于静多态而言，为了处理不同的类型，必须生成多个不同的模板实例）；
3. 可以对代码进行完全编译；因此并不需要发布实现源码（但是，分布模板库通常都需要同时发布模板实现的源代码）；
（2）另一方面，C++静多态则具有下列优点：

1. 可以很容易地实现内建类型的集合。更广义地说，并不需要通过公共基类来表达接口的共同性；

2. 所生成的代码效率通常都比较高（因为并不存在通过指针的间接调用，而且，可以进行演绎的非虚拟函数具有更多的内联机会）；

3. 对于只提供部分接口的具体类型，如果在应用程序中只是使用到这一部分接口，那么也可以使用该具体类型，而不必在乎该类型是否提供其他部分的接口。

通常而言，与动多态相比，静多态被认为具有更好的类型安全性：因为静多态在编译期会对所有的绑定操作进行检查。例如，假设我们尝试把一个错误类型的对象插入到一个容器中，如果这个容器是根据模板实例化而生产的话，那么几乎不会有危险，因为在编译期就可以检查出这个错误；但如果该容器所期望的元素是指向公共基类的指针，那么这些指针最后很有可能会指向不同类型的完整对象，而这就有可能会插入错误类型的对象。

在实际应用中，对于看起来相同的接口，如果在它们背后隐藏着一些语义假设的话，那么模板实例化体(静多态)有时也会导致一些问题。例如，对于一个假设具有关联运算符 + 的模板，如果基于一个没有关联该运算符的类型来实例化这个模板，那么就会出现一些问题。然而，基于继承体系的多态则很少会出现这种语义非匹配的问题，因为公共接口规范已经在基类中（更加）显式地指定了。

14.3.3 组合这两种多态

显然，你可以组合这两种形式的多态。例如，你可以从一个公共基类派生出不同种类的几何对象类，从而能够处理属于异类集合的不同几何对象。另一方面，你仍然可以使用模板来编写针对某种几何对象的代码。

在后面的博文xxxx中将进一步阐述继承和模板的组合。在第16章中，我们将看到：如何对成员函数的虚拟性进行参数化；当使用基于继承的奇异递归模板模式的时候，静多态要牺牲哪些额外的灵活性。

14.4 新形式的设计模板

这种新形式的静多态带来了实现设计模式的新方法。例如，以在C++程序设计中扮演重要角色的桥模式为例。我们使用桥模式的目的是为了能够在同一接口的多个不同实例中进行切换。我们通常可以使用一个指针来引用具体的实现，然后把所有的调用都委托给这个（包含具体实现的）类，从而达到我们的目的（见图14.3）

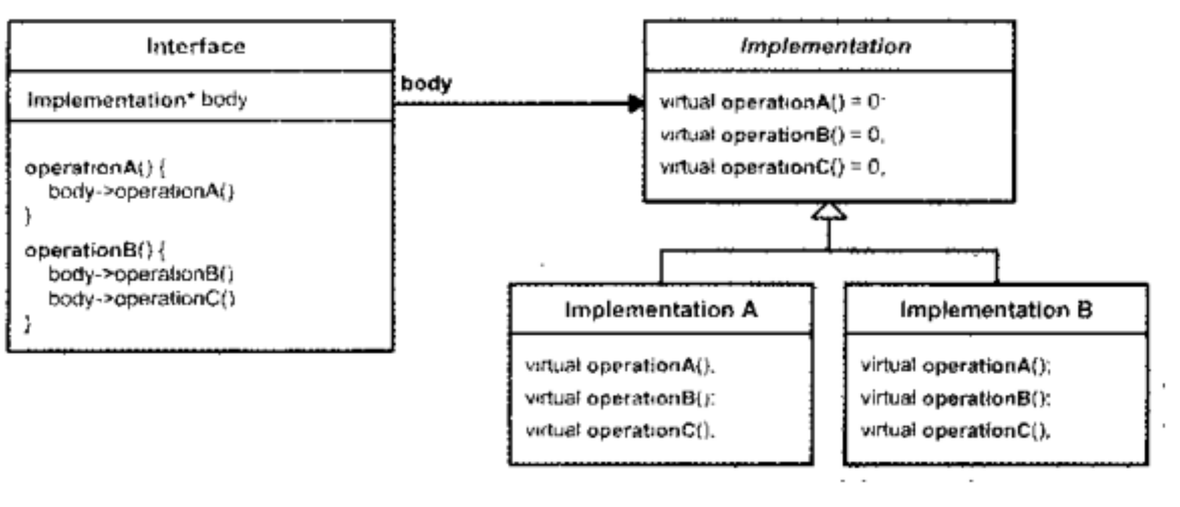

图 14.3 使用继承实现的桥模式

然而，如果实现的类型在编译期就已经是确定的，那么我们就可以借助于模板的方法来实现桥模式（见图14.4）。这将可以带来更好的类型安全性，并且也能避免使用指针，而且还能带来更高的效率。

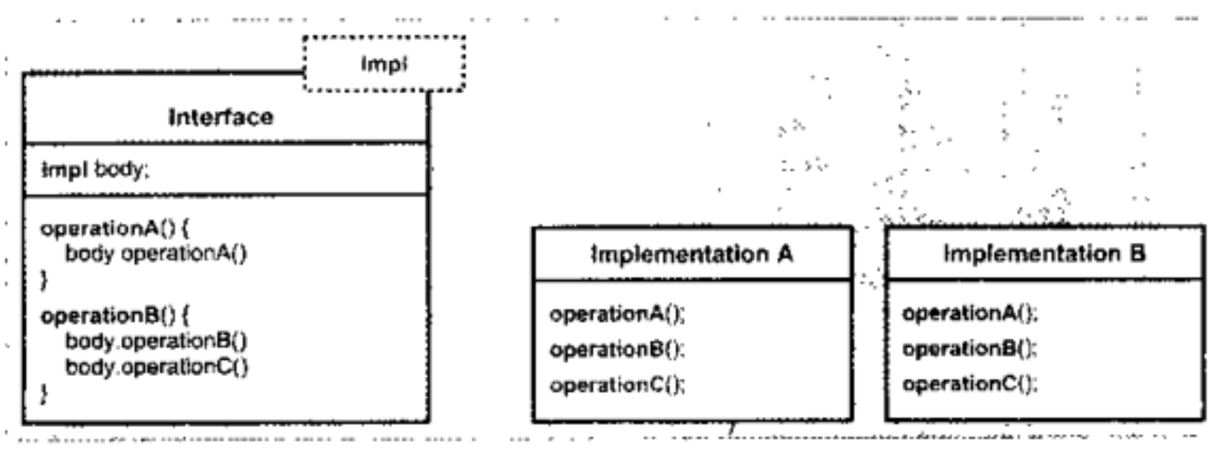

图 14.4 使用模板实现的桥模式

14.5 泛型程序设计

到目前为止，在C++泛型程序设计领域中，最显著的贡献就是STL（Standard Template Library），它后来被采纳并引入到C++标准库中。STL借助于迭代器对操作进行了参数化，从而避免了操作定义在数量上的过度膨胀。在此，你并不需要为每个容器都实现每一个操作，只需要实现某个算法一次，就可以把该算法应用到每个容器中。换句话说，泛型程序设计的"粘合剂"就是：由容器提供的并且能被算法所使用的迭代器。迭代器之所以能够肩负这样的任务，是由于容器为迭代器提供了一些特定的接口，而算法所使用的正是这些接口。我们通常也把每个这样的接口称为一个concept（即约束），它说明一个模板（即容器）如果要并入这个框架（即STL），就必须履行或者实现这些约束（也即，符合STL框架标准）。
从原则上讲，也可以使用动多态来实现这些类似于STL的功能。然而，用动多态实现的功能使用起来肯定会很受限制，因为与迭代器的概念相比，动多态的虚函数调用机制将会是一种重量级的实现机制，这就会对效率产生很大的影响。譬如增加一层基于虚函数的接口层，通常就会影响操作的效率，而且这种影响的程度可能是几个数量级的（甚至更加严重）。事实上，泛型程序设计是相当实用的，因为它所依赖的是静多态，而静多态会要求在编译期对接口进行解析。另一方面，这种要求（即对接口在编译期进行解析）还会带来一些与面向对象程序设计原则截然不同的新设计原则。

分类: C++ Template

« 上一篇： C++ template —— 模板特化（五）
» 下一篇： C++ template —— trait与policy类（七）

posted @ 2016-01-25 15:10 小天_y 阅读(4530) 评论(0) 编辑 收藏 举报