qicosmos

一点梦想：尽自己一份力，让c++的世界变得更美好！顶级c++社区欢迎你：http://purecpp.org/

C++11实现一个轻量级的AOP框架

AOP介绍

AOP（Aspect-Oriented Programming，面向方面编程），可以解决面向对象编程中的一些问题，是OOP的一种有益补充。面向对象编程中的继承是一种从上而下的关系，不适合定义从左到右的横向关系，如果继承体系中的很多无关联的对象都有一些公共行为，这些公共行为可能分散在不同的组件、不同的对象之中，通过继承方式提取这些公共行为就不太合适了。使用AOP还有一种情况是为了提高程序的可维护性，AOP将程序的非核心逻辑都“横切”出来，将非核心逻辑和核心逻辑分离，使我们能集中精力在核心逻辑上，例如图1所示的这种情况。

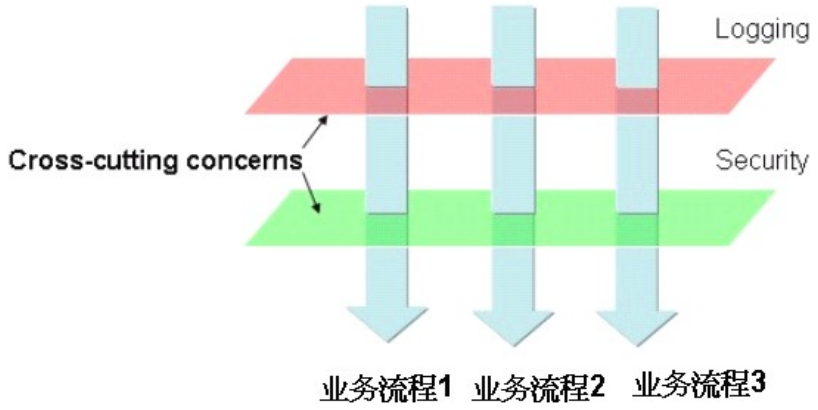

图1 AOP通过“横切”分离关注点

在图1中，每个业务流程都有日志和权限验证的功能，还有可能增加新的功能，实际上我们只关心核心逻辑，其他的一些附加逻辑，如日志和权限，我们不需要关注，这时，就可以将日志和权限等非核心逻辑“横切”出来，使核心逻辑尽可能保持简洁和清晰，方便维护。这样“横切”的另外一个好处是，这些公共的非核心逻辑被提取到多个切面中了，使它们可以被其他组件或对象复用，消除了重复代码。

AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，它们经常发生在核心关注点的多处，而各处都基本相似，比如权限认证、日志、事务处理。AOP 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

实现**AOP**的一些方法

实现AOP的技术分为：静态织入和动态织入。静态织入一般采用抓们的语法创建“方面”，从而使编译器可以在编译期间织入有关“方面”的代码，AspectC++就是采用的这种方式。这种方式还需要专门的编译工具和语法，使用起来比较复杂。我将要介绍的AOP框架正是基于动态织入的轻量级AOP框架。动态织入一般采用动态代理的方式，在运行期对方法进行拦截，将切面动态织入到方法中，可以通过代理模式来实现。下面看看一个简单的例子，使用代理模式实现方法的拦截，如代码清单1所示。

代码清单1代理模式拦截方法的实现

```
#include<memory>
#include<string>
#include<iostream>
using namespace std;
class IHello
{
public:

    IHello()
    {
    }

    virtual ~IHello()
    {
    }

    virtualvoid Output(const string& str)
    {

    }
};
```

```
class Hello : public IHello
{
public:
    void Output(const string& str) override
    {
        cout <<str<< endl;
    }
};

class HelloProxy : public IHello
{
public:
    HelloProxy(IHello* p) : m_ptr(p)
    {

    }

    ~HelloProxy()
    {
        delete m_ptr;
        m_ptr = nullptr;
    }

    void Output(const string& str) final
    {
        cout <<"Before real Output"<< endl;
        m_ptr->Output(str);
        cout <<"After real Output"<< endl;
    }

private:
    IHello* m_ptr;
};


void TestProxy()
{
    std::shared_ptr<IHello> hello =
std::make_shared<HelloProxy>(newHello());
    hello->Output("It is a test");
}
```

测试代码将输出：

```
Before real Output
It is a test
Before real Output
```

可以看到我们通过HelloProxy代理对象实现了对Output方法的拦截，这里Hello::Output就是核心逻辑，HelloProxy实际上就是一个切面，我们可以把一些非核心逻辑放到里面，比如在核心逻辑之前的一些校验，在核心逻辑执行之后的一些日志等。

虽然通过代理模式可以实现AOP，但是这种实现还存在一些不足之处：

1.不够灵活，不能自由组合多个切面。代理对象是一个切面，这个切面依赖真实的对象，如果有多个切面，要灵活地组合多个切面就变得很困难。这一点可以通过装饰模式来改进，虽然可以解决问题但还是显得“笨重”。

2.耦合性较强，每个切面必须从基类继承，并实现基类的接口。

我们希望能有一个耦合性低，又能灵活组合各种切面的动态织入的AOP框架。

1. 需要的技术

要实现灵活组合各种切面，一个比较好的方法是将切面作为模板的参数，这个参数是可变的，支持1到N（N>0）切面，先执行核心逻辑之前的切面逻辑，执行完之后再执行核心逻辑，然后再执行核心逻辑之后的切面逻辑。这里，我们可以通过可变参数模板来支持切面的组合。AOP实现的关键是动态织入，实现技术就是拦截目标方法，只要拦截了目标方法，我们就可以在目标方法执行前后做一些非核心逻辑，通过继承方式来实现拦截，需要派生基类并实现基类接口，这使程序的耦合性增加了。为了降低耦合性，这里通过模板来做解耦，即每个切面对象需要提供Before(Args...)或After(Args...)方法，用来处理核心逻辑执行前后的非核心逻辑。

支持任意类型和数量的切面组合，我们通过可变模版参数实现即可，关于可变模板参数的用法和使用技巧，读者可以参

考我在《程序员》2015年2月刊上的文章《泛化之美--C++11可变模版参数的妙用》。

为了实现切面的充分解耦合，我们的切面不必通过继承方式实现，而且也不必要求切面必须具备Before和After方法，只要具备任意一个方法即可，给使用者提供最大的便利性和灵活性。实现这个功能稍微有点复杂，复杂的地方在于切面可能具有某个方法也可能不具有某个方法，具有就调用，不具有也不会出错。问题的本质上是需要检查类型是否具有某个方法，在C++中是无法在运行期做到这个事情的，因为C++像不托管语言c#或java那样具备反射功能，然而，我们可以在编译期检查类型是否具有某个方法。下面来看看是如何实现在编译期检查某个类型的方法是否存在的。我们先定义一个检查方法是否存在的元函数（关于元函的概念数读者可以参考我在《程序员》2015年3月刊上的文章《C++11模版元编程》）。

```
template<typename T>
struct has_member_foo
{
private:
    template<typename U> static auto Check(int)
-> decltype(std::declval<U>().foo(),
std::true_type());

    template<typename U> static std::false_type
Check(...);
public:
    enum{ value =
std::is_same<decltype(Check<T>(0)),
std::true_type>::value };
};
```

这个has_member_foo的作用就是检查类型是否存在非静态成员foo函数，具体的实现思路是这样的：定义一个两个重载函数Check，利用C++的SIFNA（Substitution failure is not an erro--替换失败不算错）特性，由于模板实例化的过程中会优先选择匹配程度最高的重载函数，在模板实例化的过程中检查类型是否存在foo函数，如果存在则返回std::true_type，否则返回std::false_type，最后检查Check函数的返回类型即可知道类型是否存在foo函数。需要注意的是这里用到了C++11中

auto+decltype实现的返回类型后置特性，用这个特性的主要目的是我们无法直接得到Check的返回类型，我们需要借助模板参数T才能得到Check的返回类型（关于返回类型后置这个特性的详细用法和应用场景介绍，读者可以参考《深入应用C++11：代码优化与工程级应用》一书第一章的内容）。我们来看一下实现检查很关键的一行代码：

```
template<typename U> static auto Check(int) ->
decltype(std::declval<U>().foo(),
std::true_type());
```

我们用到了std::declval<U>().foo()，std::declval不会受到类型U是否存在共有构造函数的影响，也不会实例化类型，它仅仅是为了配合decltype来获取函数foo的返回类型，如果获取成功则表明存在foo函数，否则就会替换失败，而选择默认的Check函数。decltype在这里还有另外一个妙用，它可以通过逗号表达式连续推断多个函数的返回类型，假如我们不仅仅是要推断类型是否存在foo函数，我们还希望推断类型是否存在foo1函数，这时我们仅仅需要在加一个逗号表达式就行了：

```
template<typename U> static auto Check(int) ->
decltype(std::declval<U>().foo(),std::declval<U>
().foo1(), std::true_type());
```

这样我们就可以同时判断某个类型是否具有foo和foo1函数了，decltype在这里体现了良好的扩展性。

虽然我们通过has_member_foo可以检查出来类型是否存在foo函数，但是还存在一个问题，即重载函数的问题，假设我们有多个重载的foo函数，这个has_member_foo就不一定能检查出来，因为decltype(std::declval<U>().foo())仅仅检查的是不含参数的foo函数，因此这个has_member_foo还不够完美，我们还需要改进它，让它能对所有的重载函数有效。借助可变模板参数就可以轻松解决这个问题了：

```
template<typename T, typename... Args>
struct has_member_foo
{
private:
    template<typename U> static auto Check(int)
-> decltype(std::declval<U>
```

```
    ().foo(std::declval<Args>()...),
    std::true_type());

        template<typename U> static std::false_type
    Check(...);
    public:
        enum{ value =
    std::is_same<decltype(Check<T>(0)),
    std::true_type>::value };
    };
```

改进之后的has_member_foo就可以对所有版本的重载函数进行检查了，具体用法是这样的：

```
    cout << has_member_foo<MyStruct>::value << endl;
    //判断是否存在无参的foo函数
    cout << has_member_foo<MyStruct, int>::value <<
    endl;//判断是否存在含int参数的foo函数
```

再借助std::enable_if在编译期根据has_member_foo::value可以做一个分支选择就可以解决前面提出的问题：如果类型具有某个方法就调用，不具有也不会出错。这个问题解决之后我们就可以实现一个AOP框架了

1. 完整的实现

下面AOP完整的实现，我们对has_member_foo通过宏做了一个扩展，让它方便对所有的非成员函数进行检查，然后再借助std::enable_if和变参实现对方法的拦截。

代码清单2 AOP的实现

```
    #define HAS_MEMBER(member)\
    template<typename T, typename... Args>struct
    has_member_##member\
    {\
    private:\
            template<typename U> static auto
    Check(int) -> decltype(std::declval<U>
    ().member(std::declval<Args>()...),
    std::true_type()); \
        template<typename U> static std::false_type
    Check(...);\
    public:\
```

```
    enum{value =
std::is_same<decltype(Check<T>(0)),
std::true_type>::value};\
};\

HAS_MEMBER(Foo)
HAS_MEMBER(Before)
HAS_MEMBER(After)

#include <NonCopyable.hpp>
template<typename Func, typename... Args>
struct Aspect : NonCopyable
{
    Aspect(Func&& f) :
m_func(std::forward<Func>(f))
    {
    }

    template<typename T>
    typename std::enable_if<has_member_Before<T,
Args...>::value&&has_member_After<T,
Args...>::value>::type Invoke(Args&&... args, T&&
aspect)
    {
        aspect.Before(std::forward<Args>
(args)...);//核心逻辑之前的切面逻辑
        m_func(std::forward<Args>(args)...);//核
心逻辑
        aspect.After(std::forward<Args>
(args)...);//核心逻辑之后的切面逻辑
    }

    template<typename T>
    typename std::enable_if<has_member_Before<T,
Args...>::value&&!has_member_After<T,
Args...>::value>::type Invoke(Args&&... args, T&&
aspect)
    {
        aspect.Before(std::forward<Args>
(args)...);//核心逻辑之前的切面逻辑
        m_func(std::forward<Args>(args)...);//核
心逻辑
    }
```

```
    template<typename T>
    typename std::enable_if<!has_member_Before<T,
Args...>::value&&has_member_After<T,
Args...>::value>::type Invoke(Args&&... args, T&&
aspect)
    {
        m_func(std::forward<Args>(args)...);//核
心逻辑
        aspect.After(std::forward<Args>
(args)...);//核心逻辑之后的切面逻辑
    }

    template<typename Head, typename... Tail>
    void Invoke(Args&&... args, Head&&headAspect,
Tail&&... tailAspect)
    {
        headAspect.Before(std::forward<Args>
(args)...);
        Invoke(std::forward<Args>(args)...,
std::forward<Tail>(tailAspect)...);
        headAspect.After(std::forward<Args>
(args)...);
    }

private:
    Func m_func; //被织入的函数
};
template<typenameT> using identity_t = T;

//AOP的辅助函数，简化调用
template<typename... AP, typename... Args,
typename Func>
void Invoke(Func&&f, Args&&... args)
{
    Aspect<Func, Args...> asp(std::forward<Func>
(f));
    asp.Invoke(std::forward<Args>(args)...,
identity_t<AP>()...);
}
```

在上面的代码中，“template<typename T> using identity_t = T;”是为了让vs2013能正确识别出模板参数，因为各个编译器

对变参的实现是有差异的。在GCC下，“msp.Invoke(std::forward<Args>(args)..., AP()...);”是可以编译通过的，但是在vs2013下就不能编译通过，通过identity_t就能让vs2013正确识别出模板参数类型。这里将Aspect从NonCopyable派生，使Aspect不可复制。关于NonCopyable的实现请读者参考8.2节的内容。上面的代码用到完美转发和可变参数模板，关于它们的用法，读者可以参考第2章和第3章内容。

实现思路很简单，将需要动态织入的函数保存起来，然后根据参数化的切面来执行Before(Args…)处理核心逻辑之前的一些非核心逻辑，在核心逻辑执行完之后，再执行After(Args…)来处理核心逻辑之后的一些非核心逻辑。上面的代码中的has_member_Before和has_member_After这两个traits是为了让使用者用起来更灵活，使用者可以自由的选择Before和After，可以仅仅有Before或After，也可以二者都有。

需要注意的是切面中的约束，因为通过模板参数化切面，要求切面必须有Before或After函数，这两个函数的入参必须和核心逻辑的函数入参保持一致，如果切面函数和核心逻辑函数入参不一致，则会报编译错误。从另外一个角度来说，也可以通过这个约束在编译期就检查到某个切面是否正确。

下面看一个简单的测试AOP的例子，这个例子中我们将记录目标函数的执行时间并输出日志，其中计时和日志都放到切面中。在执行函数之前输出日志，在执行完成之后也输出日志，并对执行的函数进行计时，如代码清单3所示。

代码清单3带日志和计时切面的AOP

```
struct TimeElapsedAspect
{
    void Before(int i)
    {
        m_lastTime = m_t.elapsed();
    }

    void After(int i)
    {
        cout <<"time elapsed: "<< m_t.elapsed() -
m_lastTime << endl;
    }
```

```
private:
    double m_lastTime;
    Timer m_t;
};

struct LoggingAspect
{
    void Before(int i)
    {
        std::cout <<"entering"<< std::endl;
    }

    void After(int i)
    {
        std::cout <<"leaving"<< std::endl;
    }
};

void foo(int a)
{
    cout <<"real HT function: "<<a<< endl;
}

int main()
{
    Invoke<LoggingAspect, TimeElapsedAspect>
(&foo, 1); //织入方法
cout <<"-----------------------"<< endl;
    Invoke<TimeElapsedAspect, LoggingAspect>
(&foo, 1);

    return 0;
}
```

测试结果如图3所示。

```
entering
start timer
real foo function: 1
time elapsed: 0.001
leaving
----------------------
start timer
entering
real foo function: 1
leaving
time elapsed: 0.0020001
```

图3 AOP切面组合的测试结果

从测试结果中看到，我们可以任意组合切面，非常灵活，也不要求切面必须从某个基类派生，只要求切面具有Before或After函数即可（这两个函数的入参要和拦截的目标函数的入参相同）。

总结

轻量级的AOP可以方便的实现对核心函数的织入，还支持切面的组合，但也存在不足之处，比如不能像Java的AOP框架一样支持通过配置文件去配置切面，仍然需要手动对核心函数配置切面，如果需要通过配置文件去配置切面可以考虑使用AspectC++。

备注：本文节选自《深入应用C++11：代码优化与工程级应用》，这本书是为了和广大读者分享学习和应用C++11的经验和乐趣，告诉读者C++11的新特性是如何综合运用于实际开发中的，具有实践的指导作用。特此感谢机械工业出版社授权。

本文是我发表于《程序员》7月刊，转载请注明出处。

posted on 2015-08-31 10:13 qicosmos(江南) 阅读(...) 评论(...) 编辑 收藏