C/C++杂谈：CRTP

月踏
关注我
第大海哥 137 人赞同了该文章

一、简介

CRTP是*Curiously Recurring Template Pattern*的缩写，中文可以翻成奇异递归模板，它是通过将子类类型作为模板参数传给基类的一种模板的使用技巧，类似下面代码形式：

```
template<typename T>
class Base {};

class Derived : public Base<Derived> {};
```

CRTP的应用很广泛，特别多的开源项目都会用到这种技术，经常被用在下面三种场景中：

1. 静态多态
2. 代码复用
3. 实例化多套基类静态变量和方法

本文来详细分析总结下这几种应用场景

二、静态多态

通过CRTP这种编程技巧可以在C++中实现静态多态，也可以叫编译期多态。这是相对运行时多态发明的名字。下面通过一个具体的例子来理解静态多态。

定义一个基类Base，两个子类Derived1、Derived2，每个子类各自都有foo、bar这两个方法的自身实现。如果用运行时多态来实现这个需求，代码如下：

```
class Base {
public:
  virtual void foo() = 0;
  virtual void bar() = 0;
};

class Derived1 : public Base {
public:
  virtual void foo() override final { cout << "Derived1 foo" << endl; }
  virtual void bar() override final { cout << "Derived1 bar" << endl; }
};

class Derived2 : public Base {
public:
  virtual void foo() override final { cout << "Derived2 foo" << endl; }
  virtual void bar() override final { cout << "Derived2 bar" << endl; }
};
```

如果用静态多态来实现类似的功能，需要在基类中把this指针static_cast成子类类型指针，然后调用子类的相关函数，代码如下：

```
template<typename T>
class Base {
public:
  void foo() { static_cast<T *>(this)->internal_foo(); }
  void bar() { static_cast<T *>(this)->internal_bar(); }
};

class Derived1 : public Base<Derived1> {
public:
  void internal_foo() { cout << "Derived1 foo" << endl; }
  void internal_bar() { cout << "Derived1 bar" << endl; }
};

class Derived2 : public Base<Derived2> {
public:
  void internal_foo() { cout << "Derived2 foo" << endl; }
  void internal_bar() { cout << "Derived2 bar" << endl; }
};
```

这样的话每个子类对象都可以通过调用基类的foo、bar函数来redirect到自己的特定实现，还可以再增加下面两个helper function，这样在外面直接调foo、bar即可：

```
template <typename T> void foo(Base<T> &obj) { obj.foo(); }
template <typename T> void bar(Base<T> &obj) { obj.bar(); }
```

接下来可以用下面的代码来测试：

```
int main(int argc, char** argv) {
  Derived1 d1;
  Derived2 d2;

  foo(d1);
  foo(d2);
  bar(d1);
  bar(d2);

  return 0;
}
```

输出如下：

```
Derived1 foo
Derived2 foo
Derived1 bar
Derived2 bar
```

从上面示例可以看出，使用CRTP可以使得类具有类似virtual function的效果，同时还没有virtual function的调用开销。因为virtual function调用需要通过vptr来找到vtbl进而找到真正的函数指针进行调用，同时对象size相比使用virtual function也会减小。但是CRTP也有明显的缺点，最直接的就是代码可读性降低（模板代码的通病），还有模板实例化之后的code size有可能更大，这有可能会对指令cache有影响（纯属推测，不大好验证，不一定准确），还有就是无法动态绑定。

静态多态的具体应用非常多，比如TVM中就通过静态多态来调用子类定义的__VisitAttrs__方法，对外提供的是VisitAttrs接口：

```
// https://github.com/apache/tvm/blob/main/include/tvm/ir/attrs.h
template <typename DerivedType>
class AttrsNode : public BaseAttrsNode {
 public:
  void VisitAttrs(AttrVisitor* v) {
    ::tvm::detail::AttrNormalVisitor vis(v);
    self()->__VisitAttrs__(vis);
  }
 private:
  DerivedType* self() const {
    return const_cast<DerivedType*>(static_cast<const DerivedType*>(this));
  }
};
```

还有前文讲到的《深入理解TVM：内存分配器》也是静态多态的一种应用，只是基类dispatch的是子类的static function，但是原理是相似的，这里不再细讲了

三、代码复用

如果一些不同的类有一些相同的操作或者相同功能的变量，可以使用CRTP来复用代码，不用每个类都去定义一次这些相同的操作和变量了，还以前面的Base、Derived1、Derived2三个类来举例，假如Derived1、Derived2都有同样的foo、bar操作，区别只是里面操作的数据类型不同，那么可以把foo、bar提到基类中，这三个类略加修改之后如下：

```
template<typename T>
class Base {
public:
  void foo(const T& t) { ... }
  void bar(const T& t) { ... }

protected:
  T data_;
};

class Derived1 : public Base<Derived1> {
public:
  void self_func1() { ... }
  void self_func2() { ... }
};

class Derived2 : public Base<Derived2> {
public:
  void self_func1() { ... }
  void self_func2() { ... }
};
```

这种应用场景相对比较简单，但也应用最多，上面代码只列出了大概思路，在实际的应用场景中需要针对实际情况来修改代码，在之前介绍的《深入理解TVM：RELAY_REGISTER_OP》中的AttrRegistry就是这种应用的一个例子，还有之前介绍的《内存管理：具有共享所有权的智能指针（二）》中的enable_shared_from_this也是这种应用的一个例子，每个继承自enable_shared_from_this的子类都可以使用其中的shared_from_this方法

四、实例化多套基类静态变量和方法

如果一些不同的类有一些相同性质的静态变量或者方法，可以使用CRTP的方法来定义这些静态变量和方法，不用每个类都去定义一次了，这种应用场景和上一节代码复用的场景很相似，只是因为这里是静态变量和方法，所以单独拆开来说

还以前面的Base、Derived1、Derived2三个类来举例，假如Derived1、Derived2需要记录实例化对象的个数，也就是实现一个对象计数器，这时只要在基类中定义一个static类型的计数器和对应static函数即可，这三个类略加修改之后如下：

```
template<typename T>
class Base {
public:
  static int getObjCnt() { return cnt; }

protected:
  static int cnt;
};
template <typename T> int Base<T>::cnt = 0;

class Derived1 : public Base<Derived1> {
public:
  Derived1() { cnt++; }
};

class Derived2 : public Base<Derived2> {
public:
  Derived2() { cnt++; }
};
```

使用下面代码做下测试：

```
int main(int argc, char** argv) {
  Derived1 d11, d12, d13;
  Derived2 d21, d22;

  cout << "Derived1::getObjCnt() = " << Derived1::getObjCnt() << endl;
  cout << "Derived2::getObjCnt() = " << Derived2::getObjCnt() << endl;

  return 0;
}
```

输出如下：

```
Derived1::getObjCnt() = 3
Derived2::getObjCnt() = 2
```

虽然举的是对象计数器的例子，但是凡是和某个类而不是具体某个对象相关的属性操作，都可以用这种方法来实现

五、最后

本文只介绍了CRTP最常见的几种应用场景，后面想先抽时间写点代码测一下CRTP的性能，是不是真的如自己分析的这样有优势，敬请期待

编辑于 2021-11-13 16:11

C / C++ 高性能计算