



## c++ 中CRTP的用途

 **creekee**  
与苟且安然相处，方能寻得诗和远方。

+ 关注他

99 人赞同了该文章

今天我们来聊聊CRTP比较常见的用法，不知道何为CRPT的请看上一篇文章。

creekee: c++ CRTP（奇异的递归模板模式）介绍

38 赞同 · 4 评论 · 文章

### 1. 静态多态（Static polymorphism）。

下面通过一个例子来进行说明。

```
template <class T>
struct Base
{
    void interface()
    {
        // ...
        static_cast<T*>(this)->implementation();
        // ...
    }

    static void static_func()
    {
        // ...
        T::static_sub_func();
        // ...
    }
};

struct Derived : Base<Derived>
{
    void implementation();

    static void static_sub_func();
};
```

以 Base<Derived>::interface() 为例，在 Derived : Base<Derived> 中，Base<Derived>是先于 Derived 而存在的，所以当 Base<Derived>::interface() 被申明时，编译器并不知道 Derived 的存在，但由于此时 Base<Derived>::interface() 并不会被实例化。只有当 Base<Derived>::interface() 被调用时，才会被实例化，而此时编译器也已经知道了 Derived::implementation() 的声明了。

这里等同于通过查询虚函数动态绑定以达到多态的效果，但省略了动态绑定虚函数查询的时间。

### 2. 轻松地实现各个子类实例创建和析构独立的计数。

```
template <typename T>
struct counter
{
    static int objects_created;
    static int objects_alive;

    counter()
    {
        ++objects_created;
        ++objects_alive;
    }

    counter(const counter&)
    {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~counter() // objects should never be removed through pointers of this type
    {
        --objects_alive;
    }
};

template <typename T> int counter<T>::objects_created( 0 );
template <typename T> int counter<T>::objects_alive( 0 );

class X : counter<X>
{
    // ...
};

class Y : counter<Y>
{
    // ...
};
```

每次X或者Y实例化时， counter<X> 或者 counter<Y> 就会被调用，对应的就会增加对创建对象的计数。同样，每次X或者Y结构后，对应的减少 objects\_alive 。这里最重要的是实现了对不同子类单独的计数。

### 3. 多态链（Polymorphic chaining）。

还是通过一个具体的例子来对此进行说明。

```
class Printer
{
public:
    Printer(ostream& pstream) : m_stream(pstream) {}

    template <typename T>
    Printer& print(T&& t) { m_stream << t; return *this; }

    template <typename T>
    Printer& println(T&& t) { m_stream << t << endl; return *this; }
private:
    ostream& m_stream;
};

class CoutPrinter : public Printer
{
public:
    CoutPrinter() : Printer(cout) {}

    CoutPrinter& SetConsoleColor(Color c)
    {
        // ...
        return *this;
    }
};
```

上面Printer定义打印的方法， CoutPrinter 是 Printer 的子类，并且添加了一个设置打印颜色的方法。接下来我们看看下面这行代码：

```
CoutPrinter().print("Hello ").SetConsoleColor(Color.red).println("Printer!");
```

前半段 CoutPrinter().print("Hello ") 调用的是 Printer 实例，后面接着 SetConsoleColor(Color.red) 实际上又需要调用 CoutPrinter 实例，这样编译器就会报错。

而CRTP就可以很好的解决这个问题，代码如下：

```
// Base class
template <typename ConcretePrinter>
class Printer
{
public:
    Printer(ostream& pstream) : m_stream(pstream) {}

    template <typename T>
    ConcretePrinter& print(T&& t)
    {
        m_stream << t;
        return static_cast<ConcretePrinter*>(*this);
    }

    template <typename T>
    ConcretePrinter& println(T&& t)
    {
        m_stream << t << endl;
        return static_cast<ConcretePrinter*>(*this);
    }
private:
    ostream& m_stream;
};

// Derived class
class CoutPrinter : public Printer<CoutPrinter>
{
public:
    CoutPrinter() : Printer(cout) {}

    CoutPrinter& SetConsoleColor(Color c)
    {
        // ...
        return *this;
    }
};

// usage
CoutPrinter().print("Hello ").SetConsoleColor(Color.red).println("Printer!");
```

### 4. 多态的复制构造体（Polymorphic copy construction）。

当我们用到多态时，经常会需要通过基类的指针来复制子对象。通常我们可以通过在基类里面构造一个 clone() 虚函数，然后在每个子类里面定义这个 clone() 函数。使用CRTP可以让我们避免反复地在子类中去定义 clone() 函数。

```
// Base class has a pure virtual function for cloning
class AbstractShape {
public:
    virtual ~AbstractShape () = default;
    virtual std::unique_ptr<AbstractShape> clone() const = 0;
};

// This CRTP class implements clone() for Derived
template <typename Derived>
class Shape : public AbstractShape {
public:
    std::unique_ptr<AbstractShape> clone() const override {
        return std::make_unique<Derived>(static_cast<Derived*>(*this));
    }
protected:
    // We make clear Shape class needs to be inherited
    Shape() = default;
    Shape(const Shape&) = default;
};

// Every derived class inherits from CRTP class instead of abstract class

class Square : public Shape<Square>{};

class Circle : public Shape<Circle>{};
```

### 参考文献

Curiously recurring template pattern

编辑于 2020-12-31 07:48

C++C++ 编程C++ 应用

赞同 9913 条评论分享喜欢收藏申请转载...

写下你的评论...

13 条评论

默认时间

陈晨祺  
1和4是一回事，做静态分发。2和3其实是实现了mixin的语义。  
2020-05-04 · 作者回复了  
回复 1

今晚的月光那么美 · creekee  
说反了吧，mixi只是一种定义模式，并不是具体的语法规则，装饰器模式也是mixi的一种实现而已  
2021-08-25  
回复 赞

山楂山楂片  
偶遇会长误  
2021-02-28  
回复 赞

展开其他 1 条回复 >

咏冲星  
那CRTP怎么做到像动态多态那样把不同的子类指针给放到一个vector内?  
01-01  
回复 赞

cycoe  
例4说的就是这个事情呀，但是还是要运行期的多态，因为实例化的子类类型在编译期是无法确定的  
03-25  
回复 1

清风徐来 · Raiscies  
所以CRTP实现不了动态多态，也就是用配置文件配置来选择子类，赋值给父类是行不通的。只能在代码中写死配置文件，重新编译才行。所以感觉这种在编译阶段的多态不适合大项目，只适合小项目，比如高频交易这种简单但是需要低延时的项目，必须多策略，当切换策略时，需要修改代码，重新编写，上传到机器上。当然这些都可以通过脚本来处理。不知道这么理解对不  
02-11  
回复 1

展开其他 1 条回复 >

i4oolish  
假如我有个需求是这样的，CRTP或者C++17之前可以做到吗。  
有一个函数fun需实现如下功能，函数参数接口自己定义：  
1. 传给fun函数的第一个参数是const std::string &name;  
2. 之后的参数是不固定长度和类型的参数列表。  
fun函数需要根据第一个参数name来决定如何对之后的参数列表进行解析，然后进行操作。  
这样的需求，可以做到吗？谢谢楼主！  
2021-11-05  
回复 赞

thegoodtgg  
这个听上去就像fmt。template<typename...Ts> void print(const char \*fmt, Ts... vs);  
2021-12-07  
回复 1

Micro  
没太看懂，我再努力看看

王峰峰  
2021-04-22  
回复 赞

王峰峰的精进日常  
牛逼  
2020-05-04 · 作者回复了  
回复 赞

creekee 作者  
你咋跑这来了  
2020-05-04  
回复 赞

### 文章被以下专栏收录

知言  
致知言，以养浩然之气。

c++: 从入门到放弃