

整合应用

C++模板编程与面向对象编程范型 ——混搭编程风格

荣耀

www.royaloo.com

2009年12月 上海

中国第二届C++技术大会

动机

将模板编程的编译期评估优势与面向对象的继承与复用机制相结合，获得更优雅、灵活、高效的代码。

议题

- 模板编程与面向对象编程的基本特征
- 整合应用的语言基础
- 整合应用的典型场景
- 相关话题讨论

术语约定

- 模板编程
 - 模板编程，泛型编程，模板元编程
- 面向对象编程
 - 基于对象编程，面向对象编程
- 模板
 - 类模板 vs. 模板类，函数模板 vs. 模板函数
 - 模板的特化/特化体 (specialization) = 模板的实例 (即普通类或普通函数)
 - 模板的特化 (行为) (specialization) = 类模板的完全特化、局部特化，函数模板的完全特化
- 模式
 - C++编程惯用法，设计模式
- 聚合/组合/复合/包含

议题 I 基本特征

- 模板编程
 - ▣ 能力与特点
 - ▣ 优势与弱项
- 面向对象编程
 - ▣ 能力与特点
 - ▣ 优势与弱项
- 共同点

模板编程——能力与特点

- 类型参数化

- 函数模板——函数生成器

- 比函数重载节省工作量，且可以对付未知的类型
 - 可以重载或完全特化函数模板

- 类模板——类生成器

- 通过局部特化或完全特化实现针对特定类型的逻辑
 - 或通过继承机制实现所需的逻辑

- 模板实例化发生于编译期/链接期

- 模板是编译器提供的一种代码复用方式

- 参数类型离散化，程序代码扁平化

模板编程——优势与弱项

- 优势

- 效率
- 通用性
- 松耦合
- 适配性
- 静态多态

模板编程——优势与弱项

● 弱项

- 语法复杂
- 调试困难
- 编译耗时
- 代码膨胀
- 编译器支持的兼容性问题

面向对象编程——能力与特点

- 程序操纵的基本单元
 - 类
 - 类的实例——对象
- 三大特点：封装、继承和多态
 - 因为封装，所以隐藏
 - 因为继承（和虚函数），所以多态
 - 因为多态，所以可扩展、自适应
- 程序代码层次化

面向对象编程——优势与弱项

● 优势

- 符合程序员对客观世界的直觉认识
- 实现与使用有效分离
- 可复用性
- 动态多态

面向对象编程——优势与弱项

● 弱项

■ 效率

- 动态绑定导致运行时开销

- 动态绑定导致编译器无法执行某些优化

■ 可能导致紧密耦合的、笨重的类层次结构

- 修改基类的影响

- 修改派生类的影响

共同之处

- 抽象工具
 - 函数模板、类模板、类、类层次结构
- 接口与实现相分离
 - 模板编程
 - 模板定义接口，类型参数定义实现
 - 面向对象编程
 - 基类定义接口，派生类定义实现
- 多态性
 - 模板编程
 - 静态多态/参数式多态/编译期多态
 - 面向对象编程
 - 动态多态/运行时多态

议题 II 整合应用的语言基础

● 熟悉的

- 类模板的特化对象作为普通类的成员
- 类及对象作为函数模板的处理对象
- 函数模板作为普通类的成员
 - 支持函数模板重载
 - 可能支持函数模板完全特化
 - 但不可虚拟

● 熟悉的/可能不熟悉的

- 类模板作为普通类的嵌套成员
 - 支持成员类模板的局部特化
 - 可能支持成员类模板的完全特化
- 类作为类模板的嵌套成员

(接上)

● 熟悉的/可能不熟悉的

■ 类模板支持继承

- 类模板继承自类模板
- 类模板继承自普通类（包括类模板的特化）
- 类模板的成员函数可以是虚拟的
 - 类模板的成员函数并非函数模板
 - 类模板可以有函数模板成员（不可为虚拟）
- 普通类可以继承自类模板的特化
- 面向对象的编程规范在此同样有效
 - 例如，如果一个类模板被用作基类模板，则其析构函数应为虚拟的

■ 类作为类模板的参数

- 例如策略（policy）类，虽然策略类模板更常用

(接上)

- 不熟悉的

- 在函数模板内可以定义局部类
 - 在普通函数内也可以
 - 支持局部类的继承
 - 可能不支持局部类模板

- 其他特性

- 友元

- 函数或函数模板
- 类或类模板

- 操作符重载的形式

- 自由函数或自由函数模板
- 普通类或类模板的成员函数
- 普通类或类模板的成员函数模板

议题 III 整合应用的典型场景

- 模板为面向对象编程提供服务
- 面向对象为模板编程提供服务
- 参数化继承模式
- 奇特的递归模板模式（Curiously Recurring Template Pattern, CRTP）
- 组合使用静态多态和动态多态

模板为面向对象编程服务

- 软件开发的基本原则之一：尽早暴露问题
- 编译期约束和断言
 - ▣ 将问题暴露于编译期
 - ▣ 采用模板实现（当然也可以不……）
 - ▣ 为面向对象编程服务（当然不限于……）
- 两个简单的例子
 - ▣ `is_same_or_have_base`
 - ▣ `StaticAssert`（摘自C++2005技术大会讲义）

```

·template<typename D, typename B>
·class must_same_or_have_base
·{
·public:
·    must_same_or_have_base()
·    { void(*p)(D*, B*) = constraints; }
·private:
·    static void constraints(D* pd, B* pb)
·    { pb = pd; }
·};

```

```

template<typename D>
class must_same_or_have_base<D, void>
{
private:
    must_same_or_have_base();
};

```

```

·class A { /* ... */ };
·class B: public A { /* ... */ };
·class C { /* ... */ };
·template <class T1, class T2>
·class D: public must_same_or_have_base<T1, T2> { /* ... */ };
·int main()
·{
·    D<B, A> dba;           // ok
·    D<A, A> daa;           // ok
·    D<A, B> dab;           // error
·    D<C, A> dca;           // error
·    D<int, double> did;    // error
·}

```

编译期约束

//主模板

```
template<bool> struct StaticAssert;
```

// 完全特化

```
template<> struct StaticAssert<true> {};
```

// 辅助宏

```
#define STATIC_ASSERT(exp)\  
{ StaticAssert<((exp) != 0)>(); }
```

```
class EmptyBase { };
```

```
class Child: public EmptyBase
```

```
{
```

```
    typedef int my_int;
```

```
    void f () { /* ... */ }
```

```
};
```

```
int main()
```

```
{
```

```
    STATIC_ASSERT(sizeof(EmptyBase) != 0);
```

```
    STATIC_ASSERT(sizeof(Child) == sizeof(EmptyBase));
```

```
}
```

编译期断言

面向对象为模板编程服务

- 面向对象继承机制的能力

- 继承接口

- 虚函数、纯虚函数

- 类型定义

- 继承实现

- 来自STL的经典例子

- 面向对象带来扩展性

- 遵守模板框架约定

- 使用轻量级的面向对象

- STL中少见的专用基类

- unary_function, binary_function

- 大量的派生实现

- negate, greater, ... 函数对象适配器

```
template <typename Arg1, typename Arg2, typename Result>
struct binary_function
{
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

继承让函数对象自动获得基类中的成员类型

- 自动获得适配能力
- 且无运行时开销

```
template <typename T>
struct greater: public binary_function<T, T, bool>
{
    bool operator() (const T& x, const T& y) const
    { return x > y; }
};
```

参数化继承模式

- 类模板继承自其参数类型（之一）
 - 又称基类参数化
 - 继承方式
 - public: 接口继承/实现继承
 - private: 实现继承
 - 参数也可以是类模板或类模板局部特化或完全特化
- 典型案例
 - 饰面（Veneer）
 - 螺栓（Bolt）
 - 静态封装器（Wrapper）
 - Template Method设计模式
 - 参数化绑定类型

```
class AbstractClass
{
public:
    virtual void F1() = 0;
    virtual void F2() = 0;
public:
    void TemplateMethod()
    {
        // ...
        F1();
        // ...
        F2();
        // ...
    }
};
```

```
class ConcreteClass1: public AbstractClass
{
public:
    void F1()
    { std::cout << "ConcreteClass1::F1()" << std::endl; }
    void F2()
    { std::cout << "ConcreteClass1::F2()" << std::endl; }
};

class ConcreteClass2: public AbstractClass
{
public:
    void F1()
    { std::cout << "ConcreteClass2::F1()" << std::endl; }
    void F2()
    { std::cout << "ConcreteClass2::F2()" << std::endl; }
};
```

面向对象的 TM设计模式

```
int main()
{
    ConcreteClass1().TemplateMethod();
    ConcreteClass2().TemplateMethod();
}
```

```
template <typename  
ConcreteClass>  
class AbstractClass :  
public ConcreteClass  
{  
public:  
    void TemplateMethod()  
    {  
        // ...
```

```
AbstractClass  
<ConcreteClass>::F1();  
    // ...  
AbstractClass  
<ConcreteClass>::F2();  
    // ...  
}  
};
```

```
class ConcreteClass1  
{  
public:  
    void F1()  
    { std::cout << "ConcreteClass1::F1()" << std::endl; }  
    void F2()  
    { std::cout << "ConcreteClass1::F2()" << std::endl; }  
};  
  
class ConcreteClass2  
{  
public:  
    void F1()  
    { std::cout << "ConcreteClass2::F1()" << std::endl; }  
    void F2()  
    { std::cout << "ConcreteClass2::F2()" << std::endl; }  
};
```

问题：还有别的写法吗？

模板化的 TM设计模式

```
int main()  
{  
    AbstractClass<ConcreteClass1>().TemplateMethod();  
    AbstractClass<ConcreteClass2>().TemplateMethod();  
}
```

```

·template <typename BindMode>
·class Demo : private BindMode
·{
·public:
·void F1() { std::cout << "Demo::F1()" << std::endl; }
·void F2() { std::cout << "Demo::F2()" << std::endl; }
·};

```

```

·class Child : public Demo<Mixed>
·{
·public:
· void F1() { std::cout << "Child::F1()" << std::endl; }
· void F2() { std::cout << "Child::F2()" << std::endl; }
·};
·int main()
·{
· Demo<Mixed>* d = new Child();
· d->F1(); // Child::F1()
· d->F2(); // Demo::F2()
· delete d;
·}

```

```
class Static {};
```

```

class Mixed
{
protected:
    virtual ~Mixed() {}
public:
    virtual void F1() = 0;
};

```

```

class Dynamic
{
protected:
    virtual ~Dynamic() {}
public:
    virtual void F1() = 0;
    virtual void F2() = 0;
};

```

参数化绑定类型

CRTP

- 奇特的递归模板模式（Curiously Recurring Template Pattern, CRTP）

- 一个类派生于一个类模板的实例，后者使用前者作为模板实参
- 推广：一个类模板派生于一个类模板，后者使用前者作为模板实参

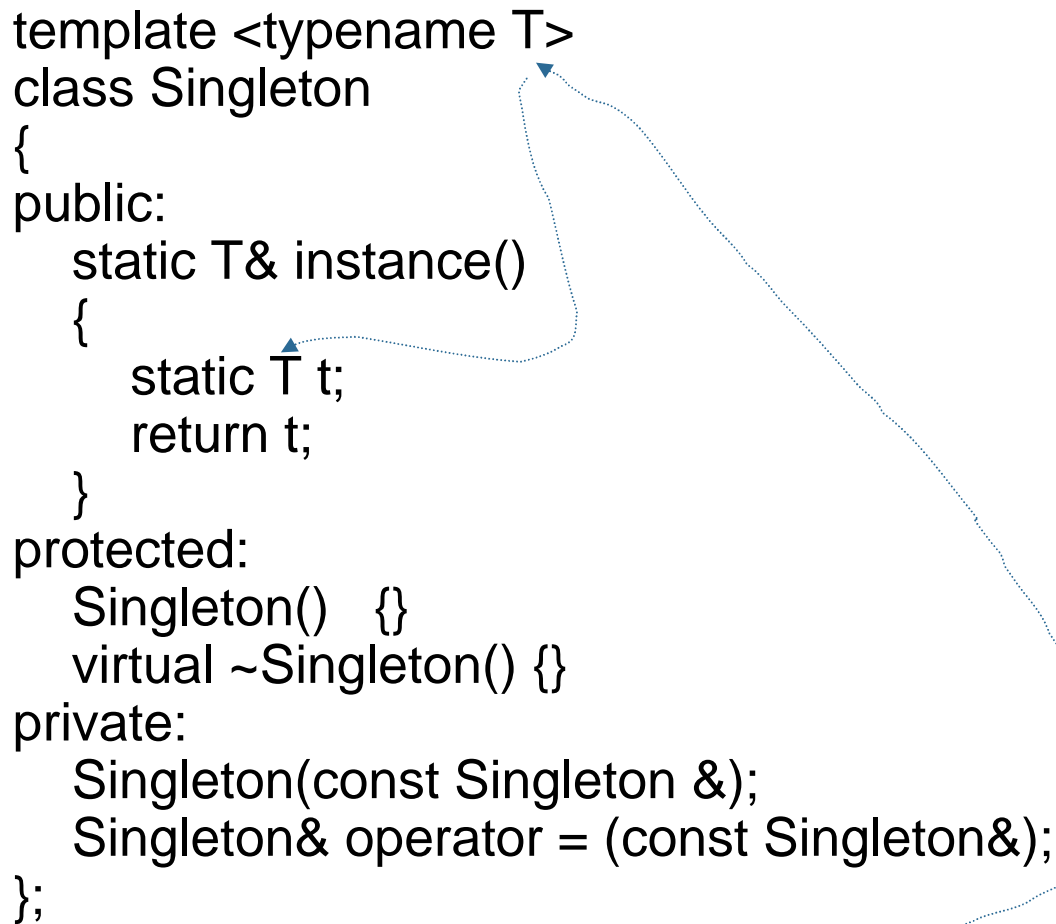
- 例子

- 泛型单件模式
- 通用引用计数器

```
template <typename T>
class Base
{ /*...*/ };

class Derived : public Base<Derived>
{ /*...*/ };
```

```
template <typename T>
class Singleton
{
public:
    static T& instance()
    {
        static T t;
        return t;
    }
protected:
    Singleton() {}
    virtual ~Singleton() {}
private:
    Singleton(const Singleton &);
    Singleton& operator = (const Singleton&);
};
```



```
class Demo: public Singleton<Demo>
{
protected:
    friend class Singleton<Demo>;
    Demo() { /* ... */ }
};
```

CRTP
泛型单件模式

```
template <typename T>
class ObjectCounter
{
public:
    ObjectCounter() { ++count; }
    ObjectCounter (const ObjectCounter<T> &) { ++count; }
    ~ObjectCounter() { --count; }
public:
    static int GetLiveCount() { return count; }
private:
    static int count;
};
template <typename T> int ObjectCounter<T>::count = 0;
```

● 未对参数类型提任何要求!

通用
引用计数器

```
class Demo1: public ObjectCounter<Demo1> { /* ... */ };
class Demo2: public ObjectCounter<Demo2> { /* ... */ };
```

```
int main()
{
    Demo1 d11, d12;
    Demo2 d21, d22, d23;
    std::cout << Demo1::GetLiveCount() << std::endl;
    std::cout << Demo2::GetLiveCount() << std::endl;
}
```

● 问题

- 如果ObjectCounter不是模板会怎样?
- Demo1与Demo2有何关系?

组合使用静态多态和动态多态

- 结合使用动态多态和静态多态两种机制
 - 优雅
 - 安全
 - 高效
- 一个简单而亲切的例子
 - 泛型容器容纳多态对象指针
 - 泛型容器容纳具体对象

·// 公共抽象基类Vehicle

```
·class Vehicle  
·{  
·public:  
·    virtual void run() const = 0;  
·    virtual ~Vehicle() {}  
·};
```

·// 派生于Vehicle的具体类Car

```
·class Car: public Vehicle  
·{  
·public:  
·    virtual void run() const  
·    {  
·        std::cout << "run a car\n";  
·    }  
·};
```

·// 派生于Vehicle的具体类Airplane

```
·class Airplane: public Vehicle  
·{  
·public:  
·    virtual void run() const  
·    { std::cout << "run a airplane\n"; }  
·    void add_oil() const  
·    { std::cout << "add oil to airplane\n"; }  
·};
```

·// 派生于Vehicle的具体类Airship

```
·class Airship: public Vehicle  
·{  
·public:  
·    virtual void run() const  
·    { std::cout << "run a airship\n"; }  
·    void add_oil() const  
·    { std::cout << "add oil to airship\n"; }  
·};
```

·// run异质vehicles集合

·void *run_vehicles(const std::vector<Vehicle*>& vehicles)*

```
·{  
·  for (size_t i = 0; i != vehicles.size(); ++i)  
·  {  
·    vehicles[i]->run();  
·  }  
·}
```

·// 为某种特定的aircrafts同质对象集合进行“空中加油”

·template <typename Aircraft>

·void *add_oil_to_aircrafts_in_the_sky(const std::vector<Aircraft>& aircrafts)*

```
·{  
·  for (size_t i = 0; i != aircrafts.size(); ++i)  
·  {  
·    aircrafts[i].add_oil();  
·  }  
·}
```

```
·int main()
·{
·    Car car1, car2;
·    Airplane airplane1, airplane2;
·    Airship airship1, airship2;
·    std::vector<Vehicle*> v;           // 异质vehicles集合
·    v.push_back(&car1);
·    v.push_back(&airplane1);
·    v.push_back(&airship1);
·    run_vehicles(v);                 // run不同种类的vehicles

·    std::vector<Airplane> vp;        // 同质airplanes集合
·    vp.push_back(airplane1);
·    vp.push_back(airplane2);
·    add_oil_to_aircrafts_in_the_sky(vp); // 为airplanes进行“空中加油”

·    std::vector<Airship> vs;        // 同质airships集合
·    vs.push_back(airship1);
·    vs.push_back(airship2);
·    add_oil_to_aircrafts_in_the_sky(vs); // 为airships进行“空中加油”
·}
```

议题IV 相关话题讨论

- 转变思维
- 复杂性
- 效率
- 混搭编程风格
- 下一代C++标准与混搭编程风格

转变思维

- 整合应用需要转变思维

- “面向对象程序员”要接受模板



- 朴素的模板化

- 如果连Hello world都可以被模板化，还有什么不可以

- 简单的类型泛化

- 深入模板化

- 需要转变思维方式

- Template Method重现

- CRTP重现

- 继承 vs. 聚合

```
·#include <iostream>
·template <typename T>
·void Output(const T & value)
·{ std::cout << value << std::endl; }
·int main()
·{
·    Output("Hello, world!");
·}
```

```

template <typename
ConcreteClass>
class AppFramework :
private ConcreteClass
{
public:
    void TemplateMethod()
    {
        // ...

AppFramework
<ConcreteClass>::F1();
        // ...

AppFramework
<ConcreteClass>::F2();
        // ...
    }
};

```

```

class ConcreteClass1
{
public:
    void F1()
    { std::cout << "ConcreteClass1::F1()" << std::endl; }
    void F2()
    { std::cout << "ConcreteClass1::F2()" << std::endl; }
};

class ConcreteClass2
{
public:
    void F1()
    { std::cout << "ConcreteClass2::F1()" << std::endl; }
    void F2()
    { std::cout << "ConcreteClass2::F2()" << std::endl; }
};

```

模板化的TM设计模式

```
template <typename T>
class CountableObject
{
public:
    static int GetLiveCount() { return count; }
protected:
    CountableObject() { ++count; }
    CountableObject (CountableObject<T> const&) { ++count; }
    ~CountableObject() { --count; }
private:
    static int count;
};
template <typename T> int CountableObject<T>::count = 0;
```

```
class Demo1 : public CountableObject<Demo1> { /* ... */ };
class Demo2 : public CountableObject<Demo2> { /* ... */ };
```

```
int main()
{
    Demo1 d11, d12;
    Demo2 d21, d22, d23;
    std::cout << Demo1::GetLiveCount() << std::endl;
    std::cout << Demo2::GetLiveCount() << std::endl;
}
```

通用
引用计数器

● private?
● public?

```
class Usable;
```

```
class Usable_lock
```

```
{
```

```
    friend class Usable;
```

```
private:
```

```
    Usable_lock() { }
```

```
    Usable_lock(const Usable_lock&) { }
```

```
};
```

```
class Usable : public virtual Usable_lock
```

```
{
```

```
    // ...
```

```
public:
```

```
    Usable();
```

```
    Usable(char*);
```

```
    // ...
```

```
};
```

```
Usable a;
```

```
class DD : public Usable { };
```

```
DD dd; // error: DD::DD() cannot access
```

```
    // Usable_lock::Usable_lock(): private member
```

通用引用计数
器衍生问题
——阻止派生

```
template <typename T>
class Final
{
    friend class T;
private:
    Final() { }
};

class NoDerived : public virtual Final<NoDerived>
{ /* ... */ };

class IllegalChild : public NoDerived{};

int main()
{
    NoDerived a;
    //IllegalChild c;
}
```

遗憾的是，标准不支持

~~CRTMP
通用阻止派生机制~~

转变思维——继承 vs. 聚合

- 能使用聚合就不使用继承？
 - ▣ 该聚合的时候聚合，该继承的时候继承
- 考虑的因素
 - ▣ 耦合性
 - ▣ 稳定接口，隐藏实现
- 一个例子
 - ▣ 继承比聚合更合适

```
template <typename T>
class Demo
{
public:
    // 精化函数
    void F1()
    {
        /* 一些处理代码 */
        t.F1();
        /* 一些处理代码 */
    }
    // 转发函数
    void F2()
    {
        t.F2();
    }
    // ...
    // 转发函数n
private:
    T t;
};
```

```
template <typename T>
class Demo : public/private/protected T
{
public:
    // 精化函数
    void F1()
    {
        /* 一些处理代码 */
        T::F1();
        /* 一些处理代码 */
    }

    // override
    virtual void G1()
    {
        /* ... */
    }
};
```

继承 vs. 聚合

复杂性

- C++本来就是一门复杂的语言
 - C++程序员接受并能对付C++语言的复杂性
 - 因为复杂，所以强大；因为强大，所以复杂
- 如何降低复杂性
 - 借鉴专家的实践经验
 - 使用库
 - 使用熟知的编程惯用法、模式
 - 规避陷阱
 - 裁剪？
 - 使用何种特性，你有自由
 - 引入模板不是为了让方案更复杂，而是更简单、更直接、更高效



效率

- 善用模板，提高程序整体运行效率
 - 编译效率 vs. 运行效率
- 使用标准库（以及其他经过验证的库）
 - 选择高效的容器和算法
 - 避免运行效率随数据的增长而急剧下降
- 避免创建不必要的临时对象
 - 传递const&而不是传值
 - 在传参数和返回值时尽量避免类型转换
- 避免创建不必要的局部对象
- 跟C++无关的影响C++程序效率的因素
 - 数据库访问.....
- 效率需要实际地评估
- 效率有时是一种用户体验
 - 多线程.....



混搭编程风格

- 编程风格

- 代码风格
- 编码习惯
- 编程范型

- 混搭编程风格

- 混搭的代码风格
- 混搭使用编程范型
- C++程序员习惯于混搭的编程风格
 - 混搭不是为了复杂化，而是为了简单、直接
 - 如果不这样，替代的做法往往是迂回的、次优的



下一代C++标准与混搭编程风格

- 与模板编程有关的两个重要任务
 - ▣ 提供简化模板编程的设施
 - ▣ 提供更多、更强的库设施
- 更好地支持多范型编程是重要目标之一
 - ▣ 改变面向对象编程与模板编程的割裂局面
- 混搭编程风格 \approx 多范型编程
 - ▣ 混搭编程风格是讲师杜撰的术语
 - ▣ 混搭编程风格强调对模板积极主动的使用

结语——用C++的思维解决问题

- C++程序员要有C++的思维
 - ▣ 语言绝非表象，语言决定思维
 - ▣ 领会C++的自由精神与实用主义
- C++程序员的思维不应该是纯面向对象的
 - ▣ C++编程将越来越混搭化

参考文献

大部分代码示例来源于以下参考资料，但做了不同程度的改编或改善。

- 《Imperfect C++中文版》
- 《C++ Templates全览》
- 《产生式编程》
- 《C++编程思想》
- 《C++语言的设计和演化》
- 《C++模板元编程》
- 《C++设计新思维》
- 《设计模式》
- 《STL源码剖析》
- “C++多态技术”
- 《C++必知必会》
- 《C++程序设计语言》
- 《C++编程规范》

自由交流时间 欢迎畅所欲言

THANK YOU!