Rd 通讯 2008 – 10 - Vol A

基于泛型技术的工程优化方法

摘要

本文从对系统在工程层面上优化的角度出发，引入泛型程序设计技术，着重讲述了 policy classes 和 type_selector 两种方法对于系统工程设计与性能上的优化。其中在对 policy classes 的介绍中，将其与 C++ virtual function 做了详细分析与对比；在对 type_selector 技术的阐述中，介绍了如何实现源代码层面上的可配置性。本文所采用的优化方法是在程序设计层面上展开的，充分利用了 C++语言的特性与灵巧的设计来达到优化目的的。本文在系统时空复杂度优化与系统设计方法两方面并重，并在最后给出了一个关于可持久化的对象与关系数据库映射（O / R Mapping）的迷你框架的实现，来阐明在实际项目中这些技术与方法的综合应用。

引言

在商业系统中，对搜狗的商业广告所产生的收入需要记录到数据库中，这样就产生了计费系统。计费系统采用 C++平台开发，从 ADD 组接收广告的计费请求，根据广告的状态，从客户的账号减除相应的费用，并记录相关信息，更改广告的状态等一些逻辑过程。

数据库使用的是 oracle，去年我写了一个接口类似于 JDBC 的 C++对于 oracle 的驱动库，名字叫做 soal。它是完全面向对象的，封装了一系列的（恐怖的）oci 函数。我在计费系统中使用 soal 来与 oracle 交互。可是，大量的操作都是使用 soal 来让 oracle 针对不同的数据库表做一些增、删、改、查的动作，而这些操作在一定程序上又是如此的相似，于是，接下来我又写了这样一个很小的框架 bfc::orm（Biz-tech Foundation Classes, Object / Relational Mapping）：只要提供一种对某一类型的对象与数据库表的对应规则，bfc::orm 就会保证将这个类型的对象实现在数据库与应用程序之间存入、取出等操作。

面对商业产品需求的丰富多变，框架应该如何灵活且迅速的应对这些持续变化着的需求？框架应该如何灵活的配置内存管理方案？关于缓存、多线程并发环

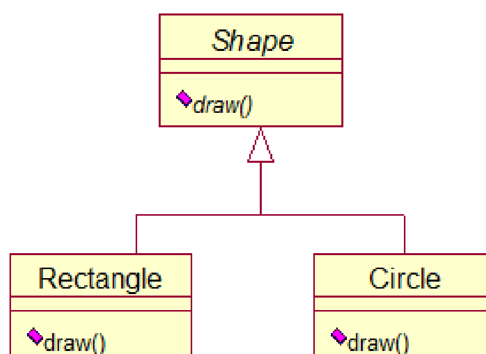
境等与数据库操作紧密相关的特性，框架需要如何演变？本文以下各章节将分别介绍 Policy classes、type_selector 等优化技术，最后以商业系统中的计费项目为例，来阐述使用这些优化技术来解决以上的问题。

另外，在这里需要格外强调的一点就是，本文介绍的方法大量的使用了模板。众所周知，模板的调式工作是非常痛苦的过程。这不仅会提高系统继任者的门槛，同时对于开发人员来讲，也要谨防落入过度设计、使用模板的陷阱。

基于 Policy classes 的优化方法

一个经典的例子

几乎所有面向对象程序设计语言的教程都会使用 shape 的例子，即给定一个 shape 类系 draw()的接口，然后类系中的 rectangle、circle 等各自实现 draw()来画出自己，而使得用户代码不用再去细分自己得到的究竟是一个什么具体的类型，只要持有一个 shape 的指针或引用就可以了。首先，让我们先来回忆一下这个经典的例子。



C++ virtual function

在这里，我们使用 C++ virtual function 机制来实现上述的例子，大家对此都很熟悉，故此处无需多着笔墨。其实现代码如下：

```
class shape
{
public:
    virtual void draw() =0;
    virtual ~shape() { }
};

class rectangle : public shape
{
```

```cpp
public:
    virtual void draw()
    {
        //draw a rectangle.
    }
};
class circle : public shape
{
public:
    virtual void draw()
    {
        //draw a circle.
    }
};

void draw_shape(shape& shape_obj)
{
    shape_obj.draw();
}

int main()
{
    rectangle r;
    circle c;

    draw_shape(r);
    draw_shape(c);

    return 0;
}
```

我们都知道，C++ virtual function 的调用机制，是 C++编译器在 class object 中插入了一个指向 virtual table 的指针 vptr，通过这个 vptr 在运行期找到需要绑定的函数指针的地址，即 virtual table 中的一个 slot，然后再执行该函数的调用的。例如上面的 draw_shape 函数中调用 shape_obj.draw();的这句话可能会是这样:

```cpp
(*shap_obj.vptr[1])(&shape_obj);
```

这样，在运行期不但每一个 class object 内存要多分配一个指针大小的空间，更是在时间上，virtual function 的调用都需要间接的内存寻址动作，从而使得效率下降。

Policy classes

所谓 Policy，是用来定义一个 class 或 class template 接口的方法。Policy 很容

易让人联想到设计模式中的 Strategy，只不过 policy 注重的是编译期。任何一个 policy 都可以生成任意多份实体，而这些实体便称为 policy classes。通常 policy class 并不被单独使用，它们主要用于继承或被内含于其他 class 中。为了能够更直接的来阐述 policy classes，先来用以上的例子给出一个 policy-based 版本的实现。

```
template <typename draw_policy>
class shape : public draw_policy
{
public:
    void draw_me()
    {
        this->draw();
    }
};

class rectangle_policy
{
public:
    void draw()
    {
        //draw a rectangle.
    }
};
class circle_policy
{
public:
    void draw()
    {
        //draw a circle.
    }
};

typedef shape<rectangle_policy> rectangle;
typedef shape<circle_policy> circle;

template <typename T>
void draw_shape(T& shape_obj)
{
    shape_obj.draw_me();
}

int main()
```

```
{
    rectangle r;
    circle c;

    draw_shape(r);
    draw_shape(c);

    return 0;
}
```

现在来看这个版本的实现，与上小节 virtual function 版本相比，它比较松散，没有像 pure virtual function 那样明确的要求按照特定的声明来实现一个函数。因为 policy 是语法导向的，而并非是标记导向的。也即是说，draw_policy 明确定义的是怎样的语法构造符合其要求，而不是必须实现出哪些函数。比如，draw_policy 并没有要求函数 draw()是 static 还是 virtual 的，或是返回什么类型的值，它只是简单的要求 class 定义了 draw()，能够提供给 draw_me()调用即可。

现在，就可以基于 shape class 来使用各种各样的 draw_policy 进行配置，来画出不同类型的图形了，这样 policy 带给了 shape class 非常大的弹性。并且，draw_policy 的实现本来就十分简短，这样也会使得代码更容易被 inline。

动多态 vs. 静多态

多态是一种能够令单一的泛型标记关联不同的特定行为的能力。在 C++中，多态主要是通过继承和 virtual function 来实现的。由于这两个机制都是在运行期进行处理的，因此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，但这种关联是在编译期进行处理的，因此将这种多态（相对于动多态）称为静多态。在这里，静多态只是一个相对的概念，用以区分与动多态的不同机制。

在前面的 virtual function 版本的例子中，通过函数 draw_shape 来展开动多态的，其参数是 reference-to-shape，这样就在运行期动态的关联相应的实参类型的行为了，这种关联是通过前面所述的 vptr 和 virtual table 来实现的。

再来看 policy-based 版本的实现，其依赖于静多态，通过 draw_shape 函数来展现这种特性的。在这里，draw_shape 是一个 template function，它的参数的类型是模板参数 T 的类型，在编译期，C++编译器会对这个模板参数 T 进行评估，得到它的实际类型。这样，就与前面的动多态版有了本质上的区别：使用动多态，在运行期只有一个 draw_shape 函数；而使用静多态，则有多个不同的函数。在我们的这个例子中，则是：

```
void draw_shape <shape<rectangle_policy> >(shape<rectangle_policy>&);
```

```
void draw_shape <shape<circle_policy> >(shape<circle_policy>&);
```

另外，只有当我们引用到了 draw_shape 对相关类型的调用的时候，编译器才会实例化出来对应这个类型的函数。这样看来，静多态会产生更多代码，使得代码膨胀，而动多态则不会存在这个问题。但是，静多态所生成的代码效率却要比动多态高，因为其调用方式是静态连接的，而且有着更多的 inline 机会。并且，更进一步，静多态都是在编译期被确定下来的，那么就意味着其有着更好的类型安全性。

设计话题

设计就是一种选择，通常我们的困难并不在于找不到解决方案，而是有太多的解决方案，我们必须要找到一组方案可以圆满解决所面临的问题。大至架构平台，小至代码片段，都需要进行选择。另外，选择还是可以组合的，这样就又雪上加霜，带给了设计可怕的多样性。

通用型接口的绞链

通用型接口，或者叫它为一个无所不包的接口更为贴切一些。接口是用来声明一些约束与限制的，即实现了一个接口则意味着要遵循接口定义的所有规范。一个过于丰富的接口则很难于厉行大量的约束，也违反了我们经常讨论的单一职责原则。这样，当外界的需求变化的时候，一个大的接口则会成倍的膨胀，使得其越来越难以维护。这更对使用这些接口的客户代码造成了更大的担负。而且也给开发人员造成了过重的智力上的负荷，他们很难想起这个肥大的接口究竟都具体定义了哪些事情。所以，被这种通用型接口的绞链牵绊住了会极其的痛苦，程序的开发工作也变成了无法承受之重。

多重继承的重量

多重继承似乎是可以承受住这种复杂性，它可能会通过少量的、明确选择后的 base classes，来组合出一组新的设计，这样一段程序则可以继承 Engineer 和 PartialTimeEmployee 类来组合生成 PartailTimeEngineer 类，从而复用 Engineer 和 PartialTimeEmployee 类的特性。但是在效率上，除了 virtual function 之外，多重继承难免要在多个 base classes 之间将 this 指针转来转去，来应付类成员相对于 this 的偏移量的不同，使得效率下降。

Policy classes 的优与劣

相对于以上两个小节中所提到的设计方面的问题，Policy classes 最灵活的应用是可以将多个短小的 policy 组合成为一个功能强大的 class。比如上面的例子，

假设我们需要得到一个填充了颜色的 rectangle，我们的 shape 可能会变成这样：

```
template <typename draw_policy, typename fill_policy>
class shape : public draw_policy, fill_policy
{
public:
    void draw_me()
    {
        this->draw();
        this->fill();
    }
};
```

由此，我们可以组合出各种各样类型的 shape 来，比如用虚线勾勒的、红色填充的、30%透明度的圆型。而其中每一个 policy class 都是功能单一、职责明确、短小精悍的，它们给程序带来了很大的弹性。通过组合它们，可以得出具有各种行为的类型的对象来。

当然，如果不想引入多重继承而承受将多个 base classes 的 this 指针进行转换的话，那么可以使用聚合，但这样会使 shape 类的运行时内存布局空间增大：

```
template <typename draw_policy,
    typename fill_policy, typename line_policy>
class shape
{
public:
    void draw_me()
    {
        _lp.set_line_style(LINE_STYLE_DOT);
        _dp.draw();
        _fp.fill();
    }

private:
    draw_policy _dp;
    fill_policy _fp;
    line_policy _lp;
};
```

更或者，可以使用模板的模板参数：

```
template < template <typename, typename> class graph_policy >
class shape : public graph_policy
{
public:
    void draw_me()
    {
```

```
        this->set_fill_color(COLOR_RED);
        this->draw();
    }
};

typedef shape< graph_policy<rect_policy, dot_line_style> >
    rectangle;
```

虽然 policy classes 与 virtual function 提供类似的效果，但是二者有着很大的不同。在使用 virtual function 客户代码中，以基本的 virtual function 来建立高端的功能，并允许客户代码改写这些基本的 virtual function 的行为。然而，policy classes 因为有丰富的类型信息及静态连接等特性，这正是指定了执行期前类型如何互相作用、能够做什么、不能够做什么的完整规则的设计，使得在类型安全的前提下，可以组合各个简单的需求来产生你的设计。另外，由于在编译期才将 class（如上述的 shape class template）和其 policies 结合在一起，所以和手工打造的程序比较起来更加牢固且更有效率。

当然，由于 policy classes 的特质，它不适用于动态链接和二进制接口。并且，policy classes 不会隐藏 class 的定义实现（如果你想那么做的话），因为模板的声明与定义目前还需要在一个编译单元内（尽管 C++标准并不要求一定要这样做，并引入了 export 关键字，但是目前只有 EDG 一个 C++编译器支持这一特性）。

另外，由于 policy classes 使用了继承，那么就意味着 shape<circle_policy>的指针可以 up-casting 到 circle_policy*，我们又不希望引入 virtual destructor 而担负 virtual function 的开销。在这里，最直接的办法就是将 circle_policy 的 destructor 声明为 protected 的。这样，就不允许使用 base class 的指针进行 delete 操作了。

Policy vs. Strategy

在此处将该话题单独作为一个小节，是因为我觉得很有必要在此讨论一下 Policy 与 Strategy 模式的异同。尽管两者看上去基本是一致的，而且所要完成的功能在目标上也是完全相同的，所以此小节将会更注重一些两者之间区别的讨论。

Strategy 模式是试图对程序进行动态配置，将决定程序行为的决策推迟到运行期进行。而在运行到这一片代码之前，程序的具体行为是无法预测的。这样，应用程序就可以拥有很大的灵活性，在设计上表达的思想也更加的清晰，代码可复用增强。

Policy 同样也是为了解决 Strategy 模式面临的问题。但是，乍看之下，policy 似乎只是比 Strategy 在性能上更快之外，没有再多的优势而言。而我认为，除了比 strategy 性能更好之外，policy 在设计的灵活性上更具有优势。不用多说，policy

具有静态特性——全面的语法检查、编译期配置、代码更容易被编译器优化与内联等诸多好处。更重要的一点是，policy 的代码可复用性与配置灵活性更高。

举一个例子来说明 policy 在设计上灵活性的体现。设计这样的一个场景，一个应用程序类 Widget 需要使用一个 lock 来做线程的同步，根据需求的不同，Widget 可能会根据情境被配置成使用 ACE_Recursive_Thread_Mutex 、My_Thread_Mutex 两个类来完成线程同步。可是，如果要使用 Strategy 模式，那么其前提是要为 Widget 开放一个抽象接口来描述这种约束规范，以使得 Widget 可以忘记 lock_stragegy 的具体类型。而在 C++中这种约束是通过类继承来实现的。但是，一个 lock_stragegy 是基于现有的 ACE 库实现的，另一个则是自己的一份实现。两者在前世并没有什么太多联系。当然，可以为 ACE_Recursive_Thread_Mutex 写一个与 My_Thread_Mutex 类具有共同基类的 adpter 来转发这些请求，这个方法可以付出一点儿小小的代价暂时发挥作用。可是，这时另外一个类 Another_Widget 却希望能够通过配置 My_Thread_Mutex 和 boost.Thread::Mutex 来使用。这时，strategy 会面临一种很尴尬的局面：是要平行地为 boost.Thread::Mutex 写一个 adapter 并入先前的类系呢？还是另 My_Thread_Mutex 多重继承自两个 lock_stragegy 类系体系呢？显然，两种做法都不是理想方案。何况，在需求的驱使下，应用程序所面临的情况越来越险恶。

而面对这种情况，policy 却可以轻松应对，只需下面这样：

```
template <typename lock_t> class Widget;

template <typename lock_t> class Another_Widget;
```

这里只需直接用一个具体的 Mutex 类实例化 Widget 或 Another_Widget 即可，不再需要继承，也不再需要 adapter。一切都同编译器在程序运行之前准备好了。致此，有了性能，亦有了灵活性。不过，policy 也为此付出了代价——没有 strategy 详细的接口规范，这有时会让人感到迷惑，由此也招致了一部分 C++开发者的痛骂。

使用 type_selector 的优化方法

模板偏特化机制

模板偏特化（Partial template specialization）是 C++泛型程序设计的基石之一，使得程序可以在 template 的所有可能实体中特化出一组子集。举个例子，有以下这样一个类模板：

```
template <typename T, typename U>
class widget
```

```
{
public:
    T foo(U& arg)
    {
        //do something generic with type U&...
    }
};
```

类模板 widget 带有两个模板参数 T 和 U，分别被其成员函数 foo 用来传入一个 U&类型的参数，并且返回一个类型为 T 的值。这是对 widget 的泛型定义。这时，假设我们有这样的需求：当模板参数为 U = int 的时候，我们希望成员函数 foo 能够做一些不同于泛型定义的特别动作。这时，以上的泛型定义明显就力不从心了。不过，C++提供了模板偏特化机制，可以使得针对这种情况来实现一个特殊的模板。这样的特殊模板如下：

```
template <typename T>
class widget<T, int>
{
    T foo(int& arg)
    {
        //do something special with int&...
    }
};
```

C++编译器在对模板实例化的过程中，总是优先尝试使用最特殊的模板来进行当前匹配，如不能很好的匹配，则再尝试次特殊的模板，依次类推，直到最后使用最基本的那个泛型模板来匹配。这样的机制赋予了我们很大的弹性，不过这种机制目前还不能施用在函数身上。

type_selector 实现

有了上述的模板偏特化机制，就可以定义 type_selector 如下：

```
//primary template to select type T by default.
template <bool expr, typename T, typename U>
class type_selector
{
public:
    typedef T result;
};

//partial specialization when expr=false and select type U.
template <typename T, typename U>
class type_selector<false, T, U>
{
```

```
public:
    typedef U result;
};
```

type_selector很简单，当第一个模板参数为true的时候，type_selector将result置为第二个模板参数的类型 T；否则，result 被置为第三个模板参数 U。可见，type_selector 可以实现编译期的条件分支语句。在实例化处，type_selector 会根据表达式的值（expr）对两个备选的模板进行筛选，使得编译器只对那个被选中的模板进行实例化。

设计话题

在泛型程序设计中，type_selector 提供给了我们可以将 if-else 分支逻辑施加于类型的操作上的能力。这样，我们就可以根据一个 boolean 表达式，来决定究竟要使用什么类型了，而且是在编译期就已经决定了的，类型安全首先得到保证。这给我们的设计带来了很大的灵活性，使得程序可以经过简单的配置，就能够相应的选择合适的类型来完成任务。比如，需要决定是要使用 new 运算符构造一个新的对象，还是使用 prototype 来 clone 一个对象。又或者，要知道目前使用的对象是通过原始指针（raw pointer）还带引用计数的智能指针（smart pointer）来操控的，以决定要不要在最后 delete 掉它。而且，type_selector 是可以嵌套使用的，这样就可以表达出 if-elseif-else 等复杂的编译期条件分支语句。

```
typedef type_selector<expr1, typename type_selector<expr2, T1,
T2>::result, T3 >::result type_selected;
```

当然，type_selector 的赖以生存的基础就是，一切都是需要在编译期就能够确定下来的。也就是说 expr 只能是编译期常量。这多少也会给我们造成一些限制，但是，我们所得到的灵活性已经足以弥补这一点小小的不足了。

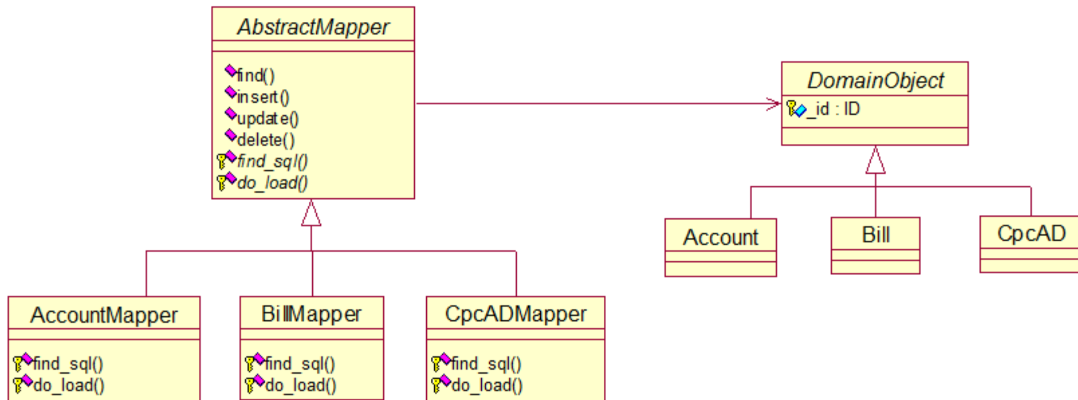
在商业系统中的应用

bfc::orm 框架

由于计费系统是按照面向对象方法设计的，所以，我们拥有了大量的用来描述客观世界中的系统的 classes，基于这些 classes 生成的对象，打造出了用以支撑计费系统的实体。然而，计费系统中需要将这些内存中的对象持久化到关系数据库中，或者从关系数据库中读取这些持久化的数据，来还原生成一个内存中的对象。但是，关系数据库是扁平的，它由一条一条的记录所排成的表格组成。这样，就需要提供一种方案，来搞定内存中的对象与数据库表的一一对应规则，从而实现二者的互相转化。

实现版本一：virtual function

理所当然的，任何一个面向对象的拥护者都会立刻意识到提供一个对这些映射方法的抽象，然后针对各个具体的可持久化实体类型，来实现其与数据库表的对应规则。当然，为了做到这些，我们还要进一步的对这些可持久化实体进行抽象，以供映射方法类的接口来使用。我当然也不例外，首选了这样的设计。



DomainObject 定义这些可持久化实体对象的一个总体上的抽象，用以提供给 AbstractMapper 类族系的接口统一操作。Account、Bill、CpcAD 分别定义了在业务领域中需要进行持久化的对象类。AbstractMapper 针对 DomainObject 类型定义了一个映射方案的接口，而将 find_sql()、do_load()等延迟到其派生类来实现。例如，Account 类实现了查询时所用的 sql：find_sql()，将持久化数据读出来还原成内存中的对象的方法： do_load()。相应地，还有 insert_sql()、do_insert()等等。这样，前面设计的机制可以运作了。

实现版本二：policy classes

在版本一中，AbstractMapper 类族系的接口都是施加于 DomainObject*类型上的，这样会引起很多的麻烦事。比如应该如何确定内存管理方案？应该在什么时候将一个 DomainObject*所指对象 delete 掉？当捕获一个数据库异常后，是否应该将该对象摧毁？首先，我决定使用 policy classes 的方案来替换上述的基于 virtual function 的方案，来发挥 policy classes 的性能优势，以达到性能上的优化。

```
template<typename obj_type>
class mapping_policy;

template <typename obj_type,
       template<typename> class mapping_policy_t =mapping_policy>
class mapper : public mapping_policy_t<obj_type>
```

我使用上述的类模板 mapper 来实现 policy classes 机制，模板参数 obj_type 指定了将要存取的对象的类型，第二个模板参数 mapping_policy_t 是一个模板，

定义了将 obj_type 类型的对象映射到关系数据库的策略。这里将其默认的参数指定为 mapping_policy，注意，这是一个只有声明，而没有定义的 class template。这即意味着客户代码须要提供一个 mapping_policy 的明确定义，或者是另外指定一个相应形式的模板。这里客户代码最简单、直接的使用方法就是提供一个 mapping_policy 的特化版本来定义如何将特定的类型与关系数据库之间关联起来，就可以使用 mapper 了。而相应的通用操作均都一次在 mapper 类中实现了。

```
template<>
class mapping_policy<test_obj>
{/*{{{*/
public:
    string find_statement()
    {
        return "SELECT id, name FROM test_table WHERE id = :id";
    }

    test_obj::ptr do_load(Soal::ResultSet& rs)
    {
        ID id = rs.getLong(1);
        string name = rs.getString(2);

        return test_obj::ptr(new test_obj(id, name));
    }

    //other implenmentation...
};

typedef mapper<test_obj> test_obj_mapper;
```

如上所示，test_obj_mapper 定义了如何从数据库 test_table 表格中读取一条记录，并还原成为内存中的一个 test_obj 类型的对象的。相应的 insert、update 等操作也大致类似。

性能实测

现在，来对比一下优化后框架的性能，在此以操作的时间为检验标准。由于框架在实际运行中，需要连接数据库，而数据库部系统的响应时间直接影响到了实验结果的可靠性。所以，此处使用了 soal 提供的一组 stub，用来模拟连接数据库并进行增、删、改、查等操作。这就保证两个版本的框架程序在外部环境上的公平了。

两个版本的框架性能在没有打开优化指令的时候，运行时间上相差无几，policy classes 尽管要好一些，但是优势不很明显，在此也就无需给出数据了。但

是在打开了-O2 的优化指令后（atuomake 的默认优化选项），policy classes 的优势开始明显的展现出来了。经过-O2 优化后的两个版本性能上的对比数据如下表：

版本	Insert (us)	Update (us)	Delete (us)	Find (us)
Virtual Function	0.4293	0.3083	0.4006	0.5273
Policy Classes	0.3694	0.2494	0.3605	0.4508
性能提升	13.95%	19.10%	10.01%	14.51%

从上表格可以看出，在性能上 policy classes 是占据优势的，其优化幅度均在 10%以上，其中以 update 操作最为明显，接近了 20%的提升。这些时间都是从避免了 virtual function 的多次间接内存寻址中省下来的。并且，我们又一次看到了，policy classes 版本的程序会更多的受到编译器的优化指令所青睐。

在这里需要指出的是，这个结果只一个实验环境下数据，因为我使用 soal 的 stub 来保证两个版本的程序在外部环境上的公平。在真实的环境中，由于应用程序与数据库的交互占据了在部分的运行时间，所以对于实际的性能的提升不会如此的明显。

可配置的内存管理方案

现在，我们不但拥有了比 virtual function 版本更大的弹性，还得到了更出色的性能。接下来，我们开始考虑内存管理上的问题。由于 AbstractMapper 类族系的接口都是施加于 DomainObject*类型上的，所以无法引入智能指针来把我们从内存管理的泥潭中解救出来，因为智能指针是另外的类型（不是 DomainObject*）。但是，基于目前的 mapper class template 的设计，我很容易就可以加上内存管理方案的 policy。

```
static const bool SMART_POINTER = true;
static const bool RAW_POINTER = false;

template <typename obj_type,
        template<typename> class mapping_policy_t =mapping_policy,
        bool use_smart_ptr =SMART_POINTER>
class mapper : public mapping_policy_t<obj_type>
{
public:
    typedef typename type_selector<use_smart_ptr,
shared_ptr<obj_type>, obj_type*>::result obj_ptr;

    //rest of definition...
};
```

在这里，使用了一个 bool 类型的模板参数 use_smart_ptr 来指明所使用的内

存管理策略。我们已经意识到了，发现 bool 后第一反应就是，可能 type_selector 将要派上用场了。确实，这里使用了 type_selector 来选择究竟是使用了智能指针（shared_ptr<obj_type>)还是原始指针(obj_type*)。并将选择出来的类型 typedef 到 obj_ptr 上，这样，后面的程序就不用理会究竟是使用了什么类型的指针了，只需使用 obj_ptr 来表示这个类型。在这里，不但避免了性能的下降，而且最主要的是我们兼得了灵活性。

需求在变化

至此，我们十分轻松的为这个小框架添加上了可配置的内存管理策略。下面，将会（同样是如此轻松地）添加另一个 policy 来实现一个新需求——有的类型需求要为从数据库中读取出来的数据缓存，而有的则不允许缓存。

```
static const bool IDENTITY_MAP = true;
static const bool NULL_IDENTITY_MAP = false;

template <typename obj_type,
        template<typename> class mapping_policy_t =mapping_policy,
        bool use_smart_ptr =SMART_POINTER,
        bool use_cache =NULL_IDENTITY_MAP>
class mapper : public mapping_policy_t<obj_type>
{
public:
    // as before...

    typedef typename type_selector<use_cache, identity_map<obj_ptr,
use_smart_ptr>,
            null_identity_map<obj_ptr, use_smart_ptr> >::result
cache_type;

    //rest of definition...
protected:
    cache_type _cache;
};
```

这一次又增加了一个模板参数 use_cache，注意，后增加的模板参数我都赋给了一个默认值，这个默认值都是兼容现在程序的使用状态的，这样就不会影响到现在程序的使用而不得不到每一个调用点修改代码了。

在这里，我并没有使用多重继承的方案，因为不想承受多个 base classes 之间 this 指针转换的负担而使得性能有所下降。所以使用了 use_cache 来指明是否使用了缓存方案。这次又是 type_selector，从一个缓存池类型和一个什么也不做的假的缓存池类型之间做出选择，将选择出来的类型 typedef 到 cache_type，在

后面声明了一个 cache_type 类型的成员变量，以用来保存缓存的结果。

需求变化竟是如此迅速

这正是商业系统的特性——就在这篇通讯还在审稿阶段的时候，业务上就又出现了新的需求：框架需要在并发任务环境下使用。这个需求对于 virtual function 的版本来说，似乎会令人很恼火！DomainObject 类族系（见上面的 UML 图）中的每一个具现类的缓存都需要分别的保证线程安全。这简直就是一件没有太多价值的体力劳动。而且大量重复性工作即会招致发生错误的风险。

然而，现在我们基于 policy classes 的解决方案中，其根基是针对类型的程序设计——类型被我们视为变量。我只需要针对泛化过后的类型 T 来保证其线程安全性，那么从该模板实例化出来的类型将都是线程安全的。这使得我的工作轻松了许多，而且很大程度上减少了发生错误的可能性。

现在，mapper 的定义已经是下面这样了：

```
template <typename obj_type,
        template<typename> class mapping_policy_t =mapping_policy,
        bool use_smart_ptr =SMART_POINTER,
        bool use_cache =NULL_IDENTITY_MAP,
        typename lock_t =ACE_Null_Mutex>

class mapper : public mapping_policy_t<obj_type>
```

模板参数 lock_t 来描述实际应用中将要使用的线程同步/互斥工具。同样，赋予其默认类型值 ACE_Null_Mutex 来保证当前无需并发任务的应用的正常运转。现在，框架就可以轻松地使用 lock_t 类开的对象来进行多线程的控制了——当然，是对符合框架约定规则的任意类型都生效的。

框架的未来

bfc::orm 框架现在已经应用在全部的计费程序中，支持三个项目的运作。在未来，这个框架希望能够做到根据外部文本的配置，来描述数据库与内存中可持久化对象的元信息，动态的生成相应的映射方案，而无需手工编码来实现这些映射策略了。但是实现中所采用的回调机制，可能会让程序在性能上付出一些代价。

总结

尽如你所见，我们使用 policy classes 带来的弹性是多么的显而易见，要为系统添加一个新的特性是如此的简单而轻松，更使得程序容易被优化器优化，在性能上得到显著的提高。再加以 type_selector 的辅助，使得可配置性更加灵活与简

单，并且在性能上避免了 policy 的多重继承造成的一些负担。通过将二者的结合，实现了应用程序在源代码层面上的可配置性。

基于使用以上技术，在设计上的灵活性使得应用程序面对需求变化的响应是如此的轻松而迅捷，在不影响现有应用的基础上，新的特性已经悄悄的添加到了应用程序中了。应用程序的储多功能，都被分解到了各个短小的、更加专注的、功能单一的、职责明确的 policy class 中去了。而应用程序所要做的只是一件事情：整合——将各个简单的 policy classes 整合成功能强大的类。