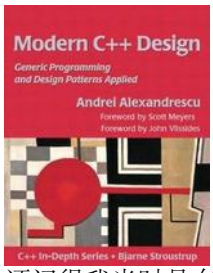
Policy-based Programming

http://freemind.pluskid.org/programming/policy-based-programming

posted on Free Mind on June 11, 2012
generated with pandoc on December 3, 2015
category: Programming

tags: Programming Paradigm, C++, Google Go



以前很喜欢一本 C++ 的书叫做《Modern C++ Design: Generic Programming and Design Patterns Applied》，这是一本讲 C++ 模板编程的书，不过讲的东西比较偏实际一点，并不像其他 C++ 模板的书那么“重口味”.....好吧，我承认，只要涉及到 C++ 模板编程的，重口味是在所难免的，不过总的来说我特别喜欢里面叫做 Policy-Based Design 的一章内容，我还记得我当时是在东四那个眼前一片青葱的大木桌那里看的。

Policy-based Programming 基本上算是用模板 + OO 的方式来进行代码共享。例如，我现在正在实现一种叫做 Conditional Probability Tree [Beygelzimer et al., 2009] 的树状 multiclass 分类器。在一棵树中每一类对应一个叶子节点，该分类器动态地构建树结构，如果碰到一种在树中没有的新类别，则会创建一个新的叶子节点来对应该类。新的叶子节点的插入位置是从树根开始往下找，在每一个节点处根据一个策略决定是放在左边子树还是右边子树。这里暂时有两种策略：一是随机，二是根据当前得到的目标函数来做判断。在通常的 C++ 或者 OO 中（或者至少是在 shogun 的 dev-team 的习惯里），一般会用继承和重载来对这里进行处理。大概会像这个样子：

```
class CPT
{
// ...
protected:
    virtual bool which_subtree()=0;
};

class RandomCPT: public CPT
{
protected:
    virtual bool which_subtree()
    {
        if (rand() % 2 == 0)
            return true;
        return false;
    }
};
```

```
18
19 class BalancedCPT: public CPT
20 {
21 protected:
22     virtual bool which_subtree()
23     {
24         // do some sophisticated calculating
25     }
26 };
```

在每个子类中重载 `which_subtree` 方法来实现不同的策略。这样的做法需要用到虚函数，我们都知道 C++ 中虚函数是通过 virtual table 来实现的，调用的时候需要在运行时解析函数的地址，因此会比较慢。比较臭名昭著的例子就是 C 的标准库里的 `qsort` ，它接受一个函数指针来对数组的元素进行比较，由于函数指针的调用很慢，所以 `qsort` 也就出奇的慢，这个也许经常用 C 做 ACM 题的同学可能会比较有感触。最近我还发现 `qsort` 引发了一宗“冤假错案”：有一个比较抵制 STL 的朋友（似乎主要原因是他开始用 C++ 的时候 C++ 标准和 STL 本身都还处于相当不完善的状态，给人留下不好的印象吧），我问他为什么讨厌 STL 的时候，他说了一些理由，然后提到 STL 的实现各种有问题，我自己写了个排序的函数比标准库里的函数快了好几个数量级。我觉得很难以置信，后来仔细一问才发现原来他说的是 C 的 `qsort` 。`:D`

实际上，在 C++ 中 `qsort` 这个问题已经得到解决了。C 之所以会出现这种尴尬的情况是因为它除了函数指针之外没有办法传递一个东西可以用来自定义排序时候数组元素大小比较的行为。而 C++ 里多了一个叫做 functor 的东西。看一下 C++ 的 `std::sort` 的签名：

```
1 template <class RandomAccessIterator, class Compare>
2 void sort(RandomAccessIterator first, RandomAccessIterator last,
3           Compare comp);
```

这里的 `comp` 参数就是一个 functor （还有另外一个重载版本可以省略该参数，直接使用 `operator <` 进行比较）。一个 functor 其实就是一个普通的 C++ 对象，不过我们要求它定义了 `operator ()` ，并接受合适的参数以及有合适的返回值：比较可惜的是 C++ 11 之前都没有合适的语法来描述这些要求，甚至都没法事先指定 `comp` 是一个什么类型的对象——事实上它可以是任意类型，只要实现了合适的 `operator ()` 的语法，甚至可以就是一个普通的裸函数指针。所以这里只好把 `comp` 的类型纳入模板参数里。

缺乏语法来对 `comp` 的行为进行描述的后果就是：当代码符合要求的时候，一切都可以正常工作；但是如果代码有不符合的情况（例如 functor 的括号运算符对应的参数列表和要求的不匹配），就完蛋了：可

能会出现各种稀奇古怪的编译错误提示，这可能算是 C++ 的模板的最大的缺点了吧（另一个巨大缺点是编译异常缓慢），特别是像 boost 这种 heavily 使用模板的库，用对了固然很爽，但是一旦出错了通常会非常崩溃，有时候看起来明明似乎是对的代码可是就是编译不过，而长达几千页的错误输出也只会帮倒忙而已。

好在 C++ 11 出来了许多新的语言特性用于解决这些问题，例如 static_assert 通常可以让许多诡异的模板错误变得更加 human-friendly，而特地针对 functor，也有许多改进。比如引入了一个可以定义 functor 的“类型”的东西，例如

```
1 std::function<int (int, int)> func;
```

指定 `func` 是一个接受两个 `int` 参数，返回一个 `int` 参数的函数指针或者 functor 或者 whatever。这样语法也就明确了许多，签名不匹配的时候也能得到更友好的错误信息。

说了这么多最重要的一点查点忘了：functor 之所以比函数指针好，是因为 functor 的 `operator()` 的地址是可以在编译期确定的（除非你很邪恶地用虚操作符然后还用 functor 引用来调用……），所以不需要再在运行的时候一遍一遍地计算地址，编译器和 CPU 也都能做更多的优化了：比如内联。这些东西如果用函数指针的话，是完全没法做的（除非再搞个强大的 JIT）。所以虽然 `qsort` 很慢，STL 里的 `std::sort` 应该还是很快的。

不过，如果仅仅因为调用效率问题就直接完全拒绝掉虚函数的话，又太武断了。因为除非是像 `qsort` 这种需要反复调用那个传进来的函数的情况，否则一般函数指针调用和普通函数调用之间的性能差异是感觉不出来的，这么做就完全犯了 premature optimization 的大忌了。

> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*
>
> —Donald Knuth. Structured Programming with go to Statements, ACM Journal Computing Surveys, Vol 6, No. 4, Dec. 1974. p.268.

但是话说回来，像 `which_subtree` 这种函数应该也是会在训练模型的时候被调用不少次的，至少可以用一个 profiler 来分析一下。不过这里还有另一个重要的理由让我对于这种纯 OO 的设计有所疑虑。Conditional Probability Tree 作为一个树状的分类器，它其实是由许多子分类器组成的。具体来说，每一个节点都对应一个回归模型，用于估计走左边和走右边的概率。节点上使用的回归模型不妨称为 base regressor，可选的 base regressor 可以有 LibLinear、Vowpal Wabbit 等等，但是由于历史原因以及各个算法本身的差异性等各种原因，这些各种类型的 base regressor 没有一个抽象得很好的基类可以供使用，所以我得针对不同的算法来定制一些操作，于是麻烦来了：如果继续使用继承和重载的方式的话，我就

需要实现 `RandomLibLinearCPT`，`RandomVwCPT`，`BalancedLibLinearCPT` 和 `BalancedVwCPT` 等各种子类。其中有一些代码会不得不被复制两份，bad smell 出现了……

类似的情况经常在实际中发生：你需要从不同的角度来定制一个类的行为，这些不同的角度之间互相是**正交**的，所以如果简单地使用继承的方式来处理的话，需要实现的类的数量将会按照笛卡尔乘积的方式增长。这个时候 policy-based programming 就可以帮忙了。啥也不说，先上代码：

```
template <class SubtreePolicy, class RegressorPolicy>
class CPT: public SubtreePolicy, public RegressorPolicy
{
public:
    void train()
    {
        // ...
        if (SubtreePolicy::which_subtree() == true) {
            // go to left subtree
        } else {
            // go to right subtree
        }

        // ...
        m_regressors.push_back(RegressorPolicy::clone(...));
    }

private:
    std::vector<typename RegressorPolicy::regressor_t> m_regressors;
};
```

啊，其实是伪代码。这里我调用函数的时候故意把它属于哪个父类的 qualifier 给写出来了，只是为了方便展示。这里的所谓 policy 可以看作是 functor 的一种推广，它不再局限于 `operator()`，而是可以有各种成员函数（例如 `which_subtree`）以及可以定义类型（例如 `regressor_t`）等等所以东西。这里我们采用继承的方式来使用 policy，这样可以很方便地使用 policy 里的成员（函数、类型等），当然也可以不使用继承，而将 policy object 作为成员变量保存起来使用也是没有任何问题的。

这样一来就很好地解决了我们刚才碰到的正交性的问题，现在我们可以分别定义好各个 policy，然后可以通过模板实例化得到像不同的组合类了：

```
CPT<RandomSubtreePolicy, VwRegressorPolicy> cpt;
```

```
CPT<BalancedSubtreePolicy, VwRegressorPolicy> cpt2;
```

注意类的数量是没有减少的，只是现在我们使用了模板，让编译器来帮我们生成代码，从而避免了不必要的重复。当然，顺带地，使用这种类似 functor 的方式的效率会比使用虚函数要高一些。当然啦，由于使用了模板，模板的所有缺点也被引入进来了：代码必须放在头文件里（除非你很想折腾一下 `export` ）、编译速度减慢、编译错误不知所云等等。另外，虽然 C++ 11 对于 functor 有了很多支持和改进，但是对于这种 general 的 policy object 来说，支持却仍然比较原始，比如我没有办法在代码里要求我的 policy object 里必须定义一个 `which_subtree` 函数，或者必须定义一个 `regressor_t` 类型，我只能假设它已经定义好了，直接去使用，如果它确实定义好了，那么就 OK 没问题，但是如果没有定义好或者和预期的不一样的话，就会出错，麻烦在于出错提示完全是随不同的编译器实现不同，并且有可能是风马牛不相及的。这种行为倒是很像动态语言里的 Duck Typing ，只是动态语言里一般不太会给几千页的 random error message 了。

其实原本有一个饱受期待的 C++0x 特性提议是用来解决这个问题的，那就是 C++ Concept，可惜的是在 09 年的时候这个东西居然被腰斩了，最终没有出现在 C++ 11 标准中。:(

如果是在动态语言中的话，就更加方便了，例如，在 Ruby 和 Python 中，我们可以通过 Mixin 的方式来实现 policy ，更美妙的是，在 Mixin 中还可以直接访问到被 mixin 的那个类的成员变量之类的。当然这只是一个语法包装，在 C++ 里我们也可以显示地把 `this` 指针传到 policy object 的函数里去，并把 policy 类声明为 friend 来避免访问权限上的问题。不过即使只是一个语法包装也确实使得代码写起来方便了许多。

动态语言不需要 Concept 因为他们使用 Duck Typing，方法可以动态地定义，并且还有 `method_missing` 这种可以拦截一切方法调用的大杀器。不过在静态语言里如果能够做这样的类型检查的话，会帮助避免很多潜在错误。虽然 C++ 里暂时没有了 Concept ，不过在 Google Go 里倒是有类似的一个东西。在 Go 里有一个叫做 interface 的东西，类似这样：

```
type SubtreePolicy interface {
    WhichSubtree() bool
}
```

看起来和 Java 之类的 interface 好像很类似，一个接口里指定了一些未实现的方法。不过其实却很不一样，因为 Go 里的接口和 OO 的类型 hierarchical 没有任何关系，当然 Go 里面本身也没有这种 OO hierarchical 。这里的一个 trick 是：在 Go 里你并不需要显式地说明一个东西实现了某个接口，只要你的类型实现了接口里指定的那些方法，就算自动实现了该接口，所有接受该 interface 类型的参数的函数都可以传递一个你自己的对象进去。所以说它更像 Duck Typing 一些，然而却又有严格的编

译期类型检查，这应该也正是 C++ 的 Concept 想要做的事情吧。

Go 本身确实也是很有意思的语言（除了它的 Logo 有点丑之外-.-bbbb），并且也已经发布了 1.0 版，从而使得 Language Specification 稳定下来了。等各种周边的库丰富起来之后，也许真的该好好看一下呢。我现在对 Go 的了解非常少，特别是不知道它的函数调用是怎么弄的，因为它的 closure 把函数的局部变量给 capture 出来似乎也是没事的。由于 Go 里很重要的一个 feature 是它的那个 Channels，所以也许它的函数调用并不是像传统的 C 语言系的那样在 stack 上来弄的也不一定呢？

References

[Beygelzimer et al., 2009] Beygelzimer, A., Langford, J., Lifshits, Y., Sorkin, G. B., and Strehl, A. L. (2009). Conditional probability tree estimation analysis and algorithms. In *UAI*, pages 51–58.