function/bind 的救赎（上）

标签： smalltalkbutton 语言.netwindowscomponents

2010-10-09 00:04 72481 人阅读 评论(118) 收藏 举报

这是那篇 C++0X 的正文。太长，先写上半部分发了。

Function/bind 可以是一个很简单的话题，因为它其实不过就是一个泛型的函数指针。但是如果那么来谈，就没意思了，也犯不上写这篇东西。在我看来，这个事情要讲的话，就应该讲透，讲到回调（callback）、代理（delegate）、信号（signal）和消息传递（messaging）的层面，因为它确实是太重要了。这个话题不但与面向对象的核心思想密切相关，而且是面向对象两大流派之间交锋的中心。围绕这个问题的思考和争论，几乎把 20 年来所有主流的编程平台和编程语言都搅进来了。所以，如果详尽铺陈，这个话题直接可以写一本书。

写书我当然没那个水平，但这个题目确实一直想动一动。然而这个主题实在太大，我实在没有精力把它完整的写下来；这个主题也很深，特别是涉及到并发环境有关的话题，我的理解还非常肤浅，总觉得我认识的很多高手都比我更有资格写这个话题。所以犹豫了很久，要不要现在写，该怎么写。最后我觉得，确实不能把一篇博客文章写成一本 20 年面向对象技术史记，所以决定保留大的架构，但是对其中具体的技术细节点到为止。我不会去详细地列举代码，分析对象的内存布局，画示意图，但是会把最重要的结论和观点写下来，说得好听一点是提纲挈领，说的不好听就是语焉不详。但无论如何，我想这样一篇东西，一是谈谈我对这个事情的看法，二是“抛砖引玉”，引来高手的关注，引出更深刻和完整的叙述。

下面开始。

0. 程序设计有一个范式（paradigm）问题。所谓范式，就是组织程序的基本思想，而这个基本思想，反映了程序设计者对程序的一个基本的哲学观，也就是说，他认为程序的本质是什么，他认为一个大的程序是由什么组成的。而这，又跟他对于现实世界的看法有关。显然，这样的看法不可能有很多种。编程作为一门行业，独立存在快 60 年了，但是所出

现的范式不过三种——过程范式、函数范式、对象范式。其中函数范式与现实世界差距比较大，在这里不讨论。而过程范式和对象范式可以视为对程序本质的两种根本不同的看法，而且能够分别在现实世界中找到相应的映射。

- 过程范式认为，程序是由一个又一个过程经过顺序、选择和循环的结构组合而成。反映在现实世界，过程范式体现了劳动分工之前“全能人”的工作特点——所有的事情都能干，所有的资源都是我的，只不过得具体的事情得一步步地来做。
- 对象范式则反映了劳动分工之后的团队协作的工作特点——每个人各有所长，各司其职，有各自的私有资源，工件和信息在人们之间彼此传递，最后完成工作。因此，对象范式也就形成了自己对程序的看法——程序是由一组对象组成，这些对象各有所能，通过消息传递实现协作。

对象范式与过程范式相比，有三个突出的优势，第一，由于实现了逻辑上的分工，降低了大规模程序的开发难度。第二，灵活性更好——若干对象在一起，可以灵活组合，可以以不同的方式协作，完成不同的任务，也可以灵活的替换和升级。第三，对象范式更加适应图形化、网络化、消息驱动的现代计算环境。

所以，较之于过程范式，对象范式，或者说“面向对象”，确实是更具优势的编程范式。最近看到一些文章抨击面向对象，说面向对象是胡扯，我认为要具体分析。对面向对象的一部分批评，是冲着主流的“面向对象”语言去的，这确实是有道理的，我在下面也会谈到，而且会骂得更狠。而另一个批评的声音，主要而来自 STL 之父 Alex Stepanov，他说的当然有他的道理，不过要知道该牛人是前苏联莫斯科国立罗蒙诺索夫大学数学系博士，你只要翻翻前苏联的大学数学教材就知道了，能够在莫大拿到数学博士的，根本就是披着人皮的外星高等智慧。而我们编写地球上的程序，可能还是应该以地球人的观点为主。

1. 重复一遍对象范式的两个基本观念：

- 程序是由**对象**组成的；
- 对象之间互相**发送消息**，协作完成任务；

请注意，这两个观念与后来我们熟知的面向对象三要素“封装、继承、多态”根本不在一个层面上，倒是与再后来的“组件、接口”神合。

2. 世界上第一个面向对象语言是 Simula-67，第二个面向对象语言是 Smalltalk-71。Smalltalk 受到了 Simula-67 的启发，基本出发点相同，但也有重大的不同。先说相同之

处，Simula 和 Smalltalk 都秉承上述对象范式的两个基本观念，为了方便对象的构造，也都引入了类、继承等概念。也就是说，类、继承这些机制是为了实现对象范式原则而构造出来的第二位的、工具性的机制，那么为什么后来这些第二位的东西篡了主位，后面我会再来分析。而 Simula 和 Smalltalk 最重大的不同，就是 Simula 用方法调用的方式向对象发送消息，而 Smalltalk 构造了更灵活和更纯粹的消息发送机制。

具体的说，向一个 Simula 对象中发送消息，就是调用这个对象的一个方法，或者称成员函数。那么你怎么知道能够在这个对象上调用这个成员函数呢？或者说，你怎么知道能够向这个对象发送某个消息呢？这就要求你必须确保这个对象具有合适的类型，也就是说，你得先知道哦这个对象是什么，才能向它发消息。而消息的实现方式被直接处理为成员函数调用，或虚函数调用。

而 Smalltalk 在这一点上做了一个历史性的跨越，它实现了一个与目标对象无关的消息发送机制，不管那个对象是谁，也不管它是不是能正确的处理一个消息，作为发送消息的对象来说，可以毫无顾忌地抓住一个对象就发消息过去。接到消息的对象，要尝试理解这个消息，并最后调用自己的过程来处理消息。如果这个消息能被处理，那个对象自然会处理好，如果不能被处理，Smalltalk 系统会向消息的发送者回传一个 doesNotUnderstand 消息，予以通知。对象不用关心消息是如何传递给另一个对象的，传递过程被分离出来（而不是像 Simula 那样明确地被以成员函数调用的方式实现），可以是在内存中复制，也可以是进程间通讯。到了 Smalltalk-80 时，消息传递甚至可以跨越网络。

为了方便后面的讨论，不妨把源自 Simula 的消息机制称为“**静态消息机制**”，把源自 Smalltalk 的消息机制称为**“动态消息机制”**。

Simula 与 Smalltalk 之间对于消息机制的不同选择，主要是因为两者于用途。前者是用于仿真程序开发，而后者用于图形界面环境构建，看上去各自合情合理。然而，就是这么一点简单的区别，却造成了巨大的历史后果。

3. 到了 1980 年代，C++出现了。Bjarne Stroustrup 在博士期间深入研究过 Simula，非常欣赏其思想，于是就在 **C 语言**语法的基础之上，几乎把 Simula 的思想照搬过来，形成了最初的 C++。C++问世以之初，主要用于解决规模稍大的传统类型的编程问题，迅速取得了巨大的成功，也证明了对象范式本身所具有的威力。

大约在同期，Brad Cox 根据 Smalltalk 的思想设计了 **Objective-C**，可是由于其语法怪异，没有流行起来。只有 Steve Jobs 这种具有禅宗美学鉴赏力的世外高人，把它奉为瑰宝，与 1988 年连锅把 **objective-c** 的团队和产品一口气买了下来。

4. 就在同一时期，GUI 成为热门。虽然 GUI 的本质是对象范型的，但是当时（1980 年代中期）的面向对象语言，包括 C++语言，还远不成熟，因此最初的 GUI 系统无一例外是使用 C 和汇编语言开发的。或者说，最初的 GUI 开发者硬是用抽象级别更低的语言构造了一个面向对象系统。熟悉 Win32 SDK 开发的人，应该知道我在说什么。

5. 当时很多人以为，如果 C++更成熟些，直接用 C++来构造 Windows 系统会大大地容易。也有人觉得，尽管 Windows 系统本身使用 C 写的，但是其面向对象的本质与 C++更契合，所以在其基础上包装一个 C++的 GUI framework 一定是轻而易举。可是一动手人们就发现，完全不是那么回事。用 C++开发 Windows 框架难得要死。为什么呢？主要就是 Windows 系统中的消息机制实际上是动态的，与 C++的静态消息机制根本配合不到一起去。在 Windows 里，你可以向任何一个窗口发送消息，这个窗口自己会在自己的 wndproc 里来处理这个消息，如果它处理不了，就交给 default window/dialog proc 去处理。而在 C++里，你要向一个窗口发消息，就得确保这个窗口能处理这个消息，或者说，具有合适的类型。这样一来的话，就会导致一个错综复杂的窗口类层次结构，无法实现。而如果你要让所有的窗口类都能处理所有可能的消息，且不论这样在逻辑上就行不通（用户定义的消息怎么处理？），单在实现上就不可接受——为一个小小的不同就得创造一个新的窗口类，每一个小小的窗口类都要背上一个多达数百项的 v-table，而其中可能 99%的项都是浪费，不要说在当时，就是在今天，内存数量非常丰富的时候，如果每一个 GUI 程序都这么搞，用户也吃不消。

6. 实际上 C++的静态消息机制还引起了更深严重的问题——扭曲了人们对面向对象的理解。既然必须要先知道对象的类型，才能向对象发消息，那么“类”这个概念就特别重要了，而对象只不过是类这个模子里造出来的东西，反而不重要。渐渐的，“面向对象编程”变成了“面向类编程”，“面向类编程”变成了“构造类继承树”。放在眼前的鲜活的对象活动不重要了，反而是其背后的静态类型系统成为关键。“封装、继承”这些第二等的特性，喧宾夺主，俨然成了面向对象的要素。每个程序员似乎都要先成为领域专家，然后成为领域分类学专家，然后构造一个完整的继承树，然后才能 new 出对象，让程序跑起来。正是因为这个过程太漫长，太困难，再加上 C++本身的复杂度就很大，所以 C++出现这么多年，真正堪称经典的面向对象类库和框架，几乎屈指可数。很多流行的库，比如 MFC、iostream，都暴露出不少问题。一般程序员总觉得是自己的水平不够，于是下更大功夫去

练剑。殊不知根本上是方向错了，脱离了对象范式的本质，企图用静态分类法来对现实世界建模，去刻画变化万千的动态世界。这么难的事，你水平再高也很难做好。

可以从一个具体的例子来理解这个道理，比如在一个 GUI 系统里，一个 Push Button 的设计问题。事实上在一个实际的程序里，一个 push button 到底“是不是”一个 button，进而是不是一个 window/widget，并不重要，本质上我根本不关心它是什么，它从属于哪一个类，在继承树里处于什么位置，只要那里有这么一个东西，我可以点它，点完了可以发生相应的效果，就可以了。可是 Simula –> C++ 所鼓励的面向对象设计风格，非要上来就想清楚，a Push Button **is-a** Button, a Button **is-a** Command-Target Control, a Command-Target Control **is-a** Control, a Control **is-a** Window. 把这一圈都想透彻之后，才能 new 一个 Push Button，然后才能让它工作。这就形而上学了，这就脱离实际了。所以很难做好。你看到 MFC 的类继承树，觉得设计者太牛了，能把这些层次概念都想清楚，自己的水平还不够，还得修炼。实际上呢，这个设计是经过数不清的失败和钱磨出来、砸出来的，MFC 的前身 Afx 不是就失败了吗？1995 年还有一个叫做 Taligent 的大项目，召集了包括 Eric Gamma 在内的一大堆牛人，要用 C++做一个一统天下的 application framework，最后也以惨败告终，连公司都倒闭了，CEO 车祸身亡，牛人们悉数遣散。附带说一下，这个 Taligent 项目是为了跟 NextSTEP 和 Microsoft Cairo 竞争，前者用 Objective-C 编写，后来发展为 Cocoa，后者用传统的 Win32 + COM 作为基础架构，后来发展为 Windows NT。而 Objective-C 和 COM，恰恰就在动态消息分派方面，与 C++迥然不同。后面还会谈到。

客观地说，“面向类的设计”并不是没有意义。来源于实践又高于实践的抽象和概念，往往能更有力地把握住现实世界的本质，比如 MVC 架构，就是这样的有力的抽象。但是这种抽象，应该是来源于长期最佳实践的总结和提高，而不是面对问题时主要的解决思路。过于强调这种抽象，无异于假定程序员各个都是哲学家，具有对现实世界准确而深刻的抽象能力，当然是不符合实际情况的。结果呢，刚学习面向对象没几天的程序员，对眼前鲜活的对象世界视而不见，一个个都煞有介事地去搞哲学冥想，企图越过现实世界，去抽象出其背后本质，当然败得很惨。

其实 C++问世之后不久，这个问题就暴露出来了。第一个 C++编译器 Cfront 1.0 是单继承，而到了 Cfront 2.0，加入了多继承。为什么？就是因为使用中人们发现逻辑上似乎完美的静态单继承关系，碰到复杂灵活的现实世界，就破绽百出——蝙蝠是鸟也是兽，水上飞机能飞也能游，它们该如何归类呢？本来这应该促使大家反思继承这个机制本身，但是那个时候全世界陷入继承狂热，于是就开始给继承打补丁，加入多继承，进而加入虚继

承，。到了虚继承，明眼人一看便知，这只是一个语法补丁，是为了逃避职责而制造的一块无用的遮羞布，它已经完全已经脱离实践了——有谁在事前能够判断是否应该对基类进行虚继承呢？

到了 1990 年代中期，问题已经十分明显。UML 中有一个对象活动图，其描述的就是运行时对象之间相互传递消息的模型。1994 年 Robert C. Martin 在《Object-Oriented C++ Design Using Booch Method》中，曾建议面向对象设计从对象活动图入手，而不是从类图入手。而 1995 年出版的经典作品《Design Patterns》中，建议优先考虑组合而不是继承，这也是尽人皆知的事情。这些迹象表明，在那个时候，面向对象社区里的思想领袖们，已经意识到“面向类的设计”并不好用。只可惜他们的革命精神还不够。

7. 你可能要问，Java 和.NET 也是用继承关系组织类库，并进行设计的啊，怎么那么成功呢？这里有三点应该注意。第一，C++的难不仅仅在于其静态结构体系，还有很多源于语言设计上的包袱，比如对 C 的兼容，比如没有垃圾收集机制，比如对效率的强调，等等。一旦把这些包袱丢掉，设计的难度确实可以大大下降。第二，Java 和.NET 的核心类库是在 C++十几年成功和失败的经验教训基础之上，结合 COM 体系优点设计实现的，自然要好上一大块。事实上，在 Java 和.NET 核心类库的设计中很多地方，体现的是基于接口的设计，和真正的基于对象的设计。有了这两个主角站台，“面向类的设计”不能喧宾夺主，也能发挥一些好的作用。第三，如后文指出，Java 和.NET 中分别对 C++最大的问题——缺少对象级别的 delegate 机制做出了自己的回应，这就大大弥补了原来的问题。

尽管如此，Java 还是沾染上了“面向类设计”的癌症，基础类库里就有很多架床叠屋的设计，而 J2EE/Java EE 当中，这种形而上学的设计也很普遍，所以也引发了好几次轻量化的运动。这方面我并不是太懂，可能需要真正的 Java 高手出来现身说法。我对 Java 的看法以前就讲过——平台和语言核心非常好，但风气不好，崇尚华丽繁复的设计，装牛逼的人太多。

至于.NET，我听陈榕介绍过，在设计.NET 的时候，微软内部对于是否允许继承爆发了非常激烈的争论。很多资深高人都强烈反对继承。至于最后引入继承，很大程度上是营销需要压倒了技术理性。尽管如此，由于有 COM 的基础，又实现了非常彻底的 delegate，所以 .NET 的设计水平还是很高的。它的主要问题不在这，在于太急于求胜，更新速度太快，基础不牢。当然，根本问题还是微软没有能够在 Web 和 Mobile 领域里占到多大的优势，也就使得.NET 没有用武之地。

8. COM。COM 的要义是，软件是由 COM Components 组成，components 之间彼此通过接口相互通讯。这是否让你回想起本文开篇所提出的对象范型的两个基本原则？有趣的是，在 COM 的术语里，“COM Component ”与“object ”通假，这就使 COM 的心思昭然若揭了。Don Box 在 Essential COM 里开篇就说，COM 是更好的 C++，事实上就是告诉大家，形而上学的“面向类设计”不好使，还是回到对象吧。

用 COM 开发的时候，一个组件“是什么”不重要，它具有什么接口，也就是说，能够对它发什么消息，才是重要的。你可以用 IUnknown::QueryInterface 问组件能对哪一组消息作出反应。向组件分派消息也不一定要被绑定在方法调用上，如果实现了 IDispatch，还可以实现“自动化”调用，也就是 COM 术语里的 Automation，而通过 列集（mashal），可以跨进程、跨网络向另一组件发送消息，通过 moniker，可以在分布式系统里定位和发现组件。如果你抱着“对象——消息”的观念去看 COM 的设计，就会意识到，整个 COM 体系就是用规范如何做对象，如何发消息的。或者更直白一点，COM 就是用 C/C++硬是模拟出一个 Smalltalk。而且 COM 的概念世界里没有继承，就其纯洁性而言，比 Smalltalk 还 Smalltalk。在对象泛型上，COM 达到了一个高峰，领先于那个时代，甚至于比它的继任.NET 还要纯洁。

COM 的主要问题是它的学习难度和安全问题，而且，它过于追求纯洁性，完全放弃了“面向类设计” 的机制，显得有点过。

9. 好像有点扯远了，其实还是在说正事。上面说到由于 C++的静态消息机制，导致了形而上学的“面向类的设计”，祸害无穷。但实际上，C++是有一个补救机会的，那就是实现对象级别的 delegate 机制。学过.NET 的人，一听 delegate 这个词就知道是什么意思，但 Java 里没有对应机制。在 C++的术语体系里，所谓对象级别 delegate，就是一个对象回调机制。通过 delegate，一个对象 A 可以把一个特定工作，比如处理用户的鼠标事件，委托给另一个对象 B 的一个方法来完成。A 不必知道 B 的名字，也不用知道它的类型，甚至都不需要知道 B 的存在，只要求 B 对象具有一个签名正确的方法，就可以通过 delegate 把工作交给 B 的这个方法来执行。在 **c 语言**里，这个机制是通过函数指针实现的，所以很自然的，在 C++里，我们希望通过指向成员函数的指针来解决类似问题。

然而就在这个问题上，C++让人扼腕痛惜。