

冒号课堂

编程范式与OOP思想

郑晖 著

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内容简介

本书以课堂对话的形式,借六位师生之口讲述编程要义。上篇对编程范式作了入门性的介绍,并对一些流行的编程语言进行了简评;下篇侧重阐发软件设计思想,其中在范式上以OOP为主,在语言上以C++、Java和C#为主。全书寓庄于谐,深入浅出,既可开阔眼界,又能引发思考,值得编程爱好者品读。

未经许可,不得以任何方式复制或抄袭本书的任何部分。 版权所有,侵权必究。

图书在版编目(CIP)数据

冒号课堂:编程范式与 OOP 思想 / 郑晖著.—北京:电子工业出版社,2009.10 ISBN 978-7-121-09545-0

I. 冒··· II.郑··· III.程序设计 IV.TP311.1

中国版本图书馆 CIP 数据核字(2009)第 166928号

责任编辑:徐定翔

印 刷:北京智力达印刷有限公司

装 订:北京中新伟业印刷有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×1092 1/16 印张: 29.75 字数: 570千字

印 次: 2009年10月第1次印刷

印 数: 4000 册 定价: 65.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010)88254888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

交流・反馈

"冒号"遇知音

徐宥:郑晖先生你好,刚从博文视点编辑那里拿到你联系方式,你的《冒号课堂》是上等作品。

郑晖: 多谢夸奖。你好, 刚看到你的评论, 非常到位, 建议也很中肯。

◆ **书稿最后一章的内容**(对谈时, 郑晖先生完成全书的前 11 章。)

徐宥:很好奇,最后一章你准备放什么内容?

郑晖:其实还有两章,一章讲设计原则,另一章讲设计模式。时间仓促,许多想写的还没写。

徐宥:对话体很难写好。之前要准备很多材料,所以我一拿到书稿看到是对话体,就知道这书是挑战。◎读完第一章,我就觉得这书太牛了。

郑晖:很高兴得到你的肯定。

• 写书的缘由

徐宥: 看得出来你是想倾心教授。

郑晖: 写书是非常偶然的想法。有一天看到当年明月的事迹,便萌生了开博写书的念头,当晚便有了构思: 用标点符号为名写成对话体。

徐宥:大家都一样。我也是看有些作者(例如李笑来)写博客写成系列了,我也就写了一个系列(《编程珠玑番外篇》)。只是我笔头慢,所以到现在还弄不成一本书。这种书肯定都是厚积薄发的,你一旦想写,马上构思,框架就全出来了。况且你的文字文学功底真地好,可以用这些去驾驭内容。

郑晖:一开始书的内容没想好,边写边整理。

徐宥: 同感,常常信马由缰又是一篇。倒不一定是事先周详计划出来的。

从数学转行到计算机的原因

徐宥: 我看过你的简历。在做计算机前,你是大学数学老师? 我也是数学系毕业的,能不能问一下是怎样的机缘使你转到计算机行业的?

交流·反馈——"冒号"遇知音

郑晖: 1996 年我去美国读math phD, 1997 年被同学拉去上计算机课。你知道, 一个纯粹的学数学的人, 会觉得计算机远不如数学的挑战性大。

徐宥: 是的, 我上大学的时候也有同感。

郑晖: 当年一心想当数学家,转行学计算机自己觉得是一种失败的标志。后来是被生活裹挟, 不情愿地转行了。

徐宥: 嗯,就像哈代一样。学数学的都对应用学科有天生的优越感,我大学时的老师老灌输我们这些。

郑晖:这是理论数学系学生的本能吧。数学系的人容易理论化、理想化,这是优点,也是缺点。

徐宥:不过计算机(任何学科)都是有其中精义的,就看读书仔细不仔细了。我到了大三大四, 基本上就把这种心态收起,好好看计算机书了。

郑晖: 一切学问都有其高妙之处。当然数学是科学之王。好心态很重要。学数学不等于高人一等,关键看个人修为。

* 读书心得

徐宥: 我看的第一本外文书,是 *Programming Languages: Concepts and Constructs*,这本书我看了好多好多遍,所以看到你的书真是百感交集。我知道中文世界没这样的书,谭浩强老师的书成吨成吨地卖,而上面那本我说的书一直没中文版。

郑晖:这也是我写书的一大缘由。寻章摘句、东拼西凑,既不能深入也不能浅出的书太多了。

徐宥: 所以看书者,不能"通经脉"。你书中有一句话:好像一个经脉又通了。我就想到脂砚 斋批《红楼梦》里的一句话,叫做"只叫批书人哭死",难寻知己啊。

郑晖: 我也有觅得知音的感觉。国内许多名气很大的人,写出的文章实在不敢恭维。博文有慧眼,寻到你这样的知音。

徐宥: 客气了。

• 书稿的定位

徐宥:这本书,浅读肯定读不出东西,要细读、深读。比如往往一句话冒号说出来,很简单,但其实里面的精妙,需要多次咀嚼,还要很深的阅历。我建议,这本书是一个起点,而不是一个终点,围绕这本书要有一个社区。个人觉得这本书是提升国内从业人员素质的一个里程碑,没多少书能有这个功底。不过我们象牙塔里面出来的,更关注"内功"层面的东西,不知道从业人员会不会关心。

郑晖:有一定技术追求的人应该会关心。

徐宥: 不光要有技术追求, 还要有技术视野。这鱼与熊掌, 目前兼得的人不多。

郑晖: 我们能产生共鸣与相似的背景有关。

徐宥:我们做科班的,视野自然是开阔的,因为这些东西无非就是客观的研究对象而已。可是一些从业人员,不要说 prolog 了,一个 MVC 就折腾不懂。我说的不是虚言。所以这本书是很有价值的,但是前提是读者要知道这书有价值。否则就是 unknown unknown (不知道自己不知道)。

郑晖: 我书中有不少类似数学定理推理的段落,不知一般读者是否适应?

徐宥: 我看行文很流畅,这个也看批书人背景了。

• 书稿的构想

徐宥: 你的文字功底很强, 我要是能写出这么流畅的行文, 至少要改稿几次。

郑晖:前面6课自己加工过,后面还没来得及。数学系的人,应该是追求完美的。我经常会 为几个用词甚至题记而反复修改,虽然知道一般人不会注意它们。

徐宥:的确是,我读得出来。个人觉得,如果想让这个书更加好读,听听其他人的审稿意见或 许有帮助。我是恨不能几年前就有此书,不过就怕其他人觉得云里雾里的,没共鸣。

郑晖:其实开始我是想谈大家都关心的OOP,后来感到说得不透,就从更广义的编程范式着手了。本来还想把其他范式再一一深入复习,但时间不够,只能等有机会写续集了。所以你会觉得有重拳打棉花之感,因为其他范式没有review。

徐宥: 其他范式怕是更加曲高和寡,书里讲OOP,已经是很看内功了。

郑晖: 是的, 所以我后来还是以OOP为主, 能把它说清楚就不错了。

徐宥: 我觉得,最后能不能做到猪肚豹尾。前面草蛇灰线,自由发挥,最后收尾浑然一气 (我知道这个很难写,我是随便想想)。比如,有没有一个好的餐馆 version 2。

郑晖:书的第 12 章是一个设计原则总纲,算得上是编程设计思想的大贯穿。从"数学上"证明了六对关键词的等价性,以及十几个编程原则的等价性。

• 书稿的人物设置

徐宥:基本上你书里的冒号是很能讲问题的,可以考虑让一个标点特别爱插话,这样至少可以帮助读者理解冒号在做什么。

郑晖: 为了让读者有些实在的感觉,还准备写设计模式,证明它们都是设计原则的推论。

徐宥: 有时候不需要每一句都直指人心,这样或许会让文章好读一些。从 Design Principle (设计原则)到 Design Pattern (设计模式),这个 path 高屋建瓴。

郑晖:是的,逗号就是用来插话的。逗者,逗留之逗,逗趣之逗。越到后来这种角色越清晰。

徐宥: 逗号的性格我看出来了。我也是仔细看了后面几章。

郑晖:开始逗号设计为最笨的,因为逗号永远得不出一个结论,后来才赋予逗号逗趣的性格; 引号喜欢引经据典;叹号比较感性;句号喜欢下结论;问号自然问题特别多,但悟性不 如句号。

徐宥:可否在适当的地方介绍这些人物性格?赋予人格化特征、性别,等等(博文有美编团队的)我其实没有读出你这些良苦构思(我还是读了好多遍稿子的)。或者说,问号、引号,基本上我是读出来的。

郑晖:不过,读者会在意这些吗?毕竟这是一本正儿八经的编程书,而不是小说。

徐宥:这个地方我倒是和你看法不大一样。邹欣(《编程之美——微软技术面试心得》、《移山之道——VSTS 软件开发指南》的作者)有一句话说:你在意,读者就在意了。 ◎

郑晖:我是想在适当的地方加入人物性格介绍,像《移山之道——VSTS软件开发指南》那样。

徐宥: 我能读出你的设计,我一直在注意这些标点符号,为什么呢?因为我想知道这些人物,他们怎么认识问题、怎么思考、怎么恍然大悟的。我看《移山之道——VSTS 软件开发指南》这本书,虽然我不懂微软技术,但是我特别记得里面的人物,就知道了这些情景。这么说吧,如果人物不鲜明,则对话无意义,因为你可以让冒号说完就放学。☺

• 小花絮

郑晖: 我的稿子你花多长时间看完的?

徐宥:如果让读者一直不释卷,最好还是能用问号带他求索,逗号引他发笑。我觉得稿子里面已经成功塑造了不少标点,但在叙述繁冗读者容易厌烦的地方,最好更加注意给读者"潜提示"。我大约看了五遍,第三遍是最认真的,花了一早上 5 个小时,后面两遍花了几天陆续看完的,前两遍有猎奇的味道,跳着看的。

郑晖: 竟然读了五遍! 难得这么认真审稿的人。多谢, 比我自己读的都多◎

徐宥:不客气。其实还有一些小的文字细节,我怕冒昧,没写下来。

郑晖:非常欢迎。

徐宥: 我看的时候记得, 待我整理整理发给你。几乎每一页都有, 有的写"有机锋之妙"。

郑晖:对了,你提到的"机锋感很强"是指哪里?

徐宥:如冒号听罢不语,直接问 6 个问题那一段,是标准的禅师带弟子顿悟。禅师不语,弟 子问,禅师答,弟子不服,禅师再解,众弟子悟。

郑晖:哈哈,我倒没有刻意去追求这个,只是顺其自然。

徐宥: 嗯, 我是读到了, 欣欣然。

郑晖: 不过写对话体真的很累,明明能一口气写完的,不得不分成几段,但这样也留给读者思考的空间,避免信息的狂轰滥炸。

交流・反馈

群英会"冒号"

• 结构篇

序号

提出人:邹 飞

该书上篇对一些编程语言和范式做了介绍和比较,第 二部分主要是集中在 OOP 的介绍。第一部分因为牵 涉很多内容, 显得有点散。

阅读样章后的看法和建议

■ 炬:同意。

提出人:徐 宥

面向对象讲得很多, 其他可能需要扩充。

这本书第 5 章往后的内容主要集中在面向对象的 paradigm 上。如果本书的定位是 OOP 的话,这样是 无可厚非的,只是前文的那些铺垫有重拳打棉花之 只是铺垫越写越长,以致独立成篇了。后来 感。

■ 炬:同意。

提出人:霍 炬

概括来看,本书只讲了两部分,语言范式和面向对象。 语言范式部分相当精彩。对各语言的描述, 概括也恰 当。

GUI 开发,面向对象技术未必那么重要。用半本书来 不是本书关心的问题。但可以肯定的是它不 讲一种特定的技术, 有点可惜。

简言之, 我认为两部分的跳跃太大。这两部分的立意 高度有差距。

徐 宥:同意。

郑晖的回复

第一部分是对各种编程范式的介绍, 自然得 牵扯到很多内容。但毕竟针对同一主题,形 散而神不散。狭而深固然是一种学习方式 (如书中的后七课),广而博同样也是一种 (如书中的前六课)。

本书的最初定位的确是 OOP, 那是最广大的 程序员使用的编程范式。但又不想让读者的 眼光过于局限, 故而在前面增加了各种编程 范式和语言的介绍。

准备对其他范式再作进一步深入, 但时间有 限,只能等以后有机会再说了。

本书重点正是 OOP, 尤其针对人们对 OOP 中常见误区和模糊区。介绍 OOP 的书汗牛 充栋,用半本书来介绍不是太多,而是太少。 至于是否拖沓,应以内容而非篇幅为标准。 面向对象部分个人认为过于拖沓。另外,如果不进行 OOP 究竟有多重要,这是一个开放的话题,

仅仅专长于 GUI, 否则那么多的中间件 (middleware)、Web Service 也不会用 OOP 来开发了。

本书之所以以 OOP 为主题, 是出于实用的 角度, 而不是理论的角度。毕竟现在的程序 员大多数只知道 OOP。

两部分的跳跃的确有些大,这与本书的写作 过程有关。立意高度有差距也不假,原因前 面已经解释过。

• 读者定位篇

阅读样章后的看法和建议 序号

1 提出人:邹 飞

该书提到了很多范式, 而且尝试对这些范式做出比较 并给出自己的理解。个人认为,对一个语言或范式的 理解更多依赖于对其特性的深入学习和把握, 他人的 说教较难让人有醍醐灌顶的感觉。个人对于这种思想 其或接近哲学理念的东西(比如语言之争)不很感兴 趣,或者说语言的优缺很难三言两语表述清楚,而且 这些观念很难说放之四海而皆准,每个人根据他学习 和把握的知识可能都会有些自己不一样的看法, 比如 我个人就会觉得该书中有些观点有待商榷。

如果该书定位于之前没有了解这些语言或范式的读 者,对他们进行知识普及,我觉得还是很可取的,毕 竟该书有着较广的知识面, 而且对一些错误观念的批 判也很有意义。但对于在某方面有所专长的工程人 员,可能不是太适合,做技术书太泛了。

■ 炬:同意。

徐 宥:邹飞先生关于"观念很难说放之四海皆准" 的意见我是同意的,但作者也说了,冒号也只是和大 至于说到做技术书太泛了,这话应该只适用 家探讨,而不是自己强迫大家接受,因此从这个角度 前半部分——那本就是一个泛泛的介绍,后 看,不同的人对文本有不同的理解是正常的事情。事 半部分对 OOP 的最重要的特征进行了较为 实上,有些文本需要看很多书做很多实践才能弄明 细节的讨论,恐怕算不得泛泛吧?而后半部 白,这也是邹飞先生说的"更多是依赖对特性的深入 分才是全书的重点,我相信对某方面有所专 学习"。这本书的内容就天生拒绝了那些不喜欢在思 想层面反思的读者, 也会让一部分持有不同思考的读 者觉得读着不对劲。所以, 我觉得邹先生给出的是正 常的,客观的评价,也无需因此担心书的内容:)

提出人:霍炬

定位存在一定问题。本书涉及的方面过于宽泛。对缺 乏足够实际经验的读者很难领会其内涵,恐怕也不易 获得认同。

郑晖的回复

本书反复说明编程范式是一个培养的过程, 不是一朝一夕能领会的。但凡事总得有个开 始的过程吧, 本书的前半部分就是为了给不 了解它们的读者一个编程范式的入门(但不 是编程入门)。

我认为思想甚或接近哲学理念的东西非常 重要,关系到程序员的境界,对知识的融会 贯通的把握。至于语言的优缺点当然很难三 言两语表述清楚, 本书特别说明是一家之 言,并无定论。

其实本书开宗明义就提出本课堂是开放式 的,从来没有说哪一些观点是放之四海而皆 准的。关键是给大家一些启发或触动,无论 是引起共鸣还是争论、都是不错的结果。

(当然这不是为可能出现的错误找借口。) 这本书本就刻意与大多数书籍有所不同,开 班发言中便已声明,请不要拿老眼光来看待。 长的工程人员同样会有所帮助的。

缺乏足够实际经验的读者的确很难领会书 中的内涵, 但正如书中所说, 学习是一个迭 代提高的过程,需要多次刨磨。另外,容忍 无知也是学习的一个必经阶段, 就怕许多人 根本不知道自己无知。如果本书不能让一些 人完全理解, 让他们知道还有一些自己未知 的领域, 也是一种贡献。本书不强求认同, 否则就不会如此观点鲜明地谈一些可能有 争论的问题了。如果只是照本宣科, 或只是 机械地拼凑总结,那么本书的意义便完全丧 失, 本书所提倡的精神也完全被阉割。

• 体例篇

阅读样章后的看法和建议 郑晖的回复 1 提出人:邹飞 这是双刃剑,有人认为过场多余,有人认为过 该书采用对话式来组织,形式很新颖,但缺点在于 场有助于缓冲节奏,避免信息轰炸。如果句句 "废话"太多,很多时候需要用一些过场语言,可 无闲话,那就不是对话体了。 能会分散读者注意力。 提出人: 蒋波涛 这也是见仁见智的问题。只要书在排版上做些 在每一节的安排上紧凑些,不要搞预览、提问、讲 合理的安排, 增强层次感, 可以让读者很清楚 哪是正文, 哪是附文。 解、插语、总结和参考这么复杂, 因为它们会不断 预览是为了提起读者的兴趣; 提问是为了让读 打断读者的思路。 者带着问题看书:插语是补充一些必要的知 徐 宥:我是非常同意的,但是作者也有放置这些 识, 但又不妨碍正文的连贯性: 的理由, 我也很难评价哪种方法好。 总结是不少读者的要求, 也是对较为分散的全 文的再次整理;参考是对引用文献的尊重,更 是必不可少的。 最后,这六个部分正好暗合文中的六位主角的 特征,也是一个创意。 提出人:霍 炬 是一个好的建议。不过暂不在书中回答的主要 课后思考建议配参考答案,或者以后在网上补全。 原因是: 有些问题是开放的, 没有固定的答案, 主要的目的是促使读者进一步思考。

• 其他

徐 宥:同意。

建立社区, 注意习题、思考的编号。

序号 阅读样章后的看法和建议 郑晖的回复 提出人:徐 宥 其实, 本书曾多处(至少三四处)提到肤 浅的比方反而会误人, 必须进一步严谨地 metaphor 不要太多。 思考。比如书中的一句话: 浅显的比方只 P100 讲到大轮船开到 duck type 的池塘里面, 具体是 是门槛前的台阶, 借之或可拾级入门, 却 什么含义呢? 是用 duck type 做了一个重量级的怪兽? 无法登堂入室。 第6页说小兵器一寸短一寸险,这个 metaphor 就变成 但比方的确会减少距离感。对干专注的主 了用低级语言难免风险, 似乎思维上也很跳跃。 题, 肯定不能只停留于表面的 metaphor。 过多的不落到实处的隐喻会让读者摸不着头脑。 你提到的鸭子类型, 因为是简介部分, 故 **恒**: metaphor 这个问题我很同意,很多书都是, 而没有多加解释, 其实涉及 specification 的 用一个人们不熟悉的概念来讲解另外一个不熟悉的。 问题。在后面的章节, 已经一再强调规范 的重要性。 提出人:徐 宥 很好的建议。

如果代答,似乎有违本书宗旨。

序号 阅读样章后的看法和建议

3 提出人:徐 宥

人物个性的塑造不够完整饱满。

作者试图让冒号在禅宗棒喝的导师形象和循循善诱的 老夫子形象之间求得平衡。比如 p39, 机锋感很强, 换成古文就是禅宗导师在教育弟子,第 13~14 页也 引经据典;问号勤奋好学,刨根问底;逗 是,可算是得道妙论,可是换到其他场合,冒号又成 号插科打诨,调节气氛; 句号颖悟灵性, 了一个"道貌岸然"的老师了(p11)。有时候冒号又 举一反三; 叹号: 浅尝辄止, 多愁善感。 想成为大家的"同学"。

提出人:邹飞

一些不一样的观念,如关干栈和堆的比较:

- a) 栈也可以再运行期决定空间大小(alloca)。
- b) 关于堆是否 thread-safe, 从 OS 角度看, heap api 基本都已经提供了 thread-safe 的版本, 而语言层面, C/C++标准是 undefined, 但并不是 no, 事实上众多实 现都是 thread-safe 的。
- c) "如果所需内存空间较小且较固定,则尽量采用栈 分配",采用堆还是栈,不是取决于对象大小而是生 命周期吧?

恒:其实 C 众多实现不是 thread-safe 的。而且我 们现在都提倡尽量用多进程,分开数据共享区域。云 风也有类似 blog。但是邹飞的说法也不能说不对,只 是这种争论已经超过这个书的范围了。所以邹飞说的 是"一些不一样的观念:"是观念,但不算错误。

提出人:邹 飞

关于实现继承和接口继承的定论上升到境界的层面, 这个其实也是要看适用场景的。

提出人:邹 飞

把 VB 和 Delphi 称为前台语言不太合适吧?

- a) Delphi 也是可以脱离 IDE 而存在的。
- b) VB 和 Delphi 并不单纯用于 UI 开发,如果强调 IDE 程。这似乎并不矛盾吧? 的便利性, C#也不差啊。

提出人:邹 飞 7

Python 效率为什么会成为最主要的问题?

徐 宥:技术层面的几个细节, 邹飞先生说的都是对 的, 但是和原作者出发点不同, 经历不同, 所以认识 不一样。我已经认真比较过文本了, 基本我同意原作 者的文本。作者这么写, 其实背后是有很多经历的,

郑晖的回复

这个有待改进, 也是时间上过于仓促, 没 有进一步完善人物, 更注重技术层面。但 六个人物还是有他们的特点的:

冒号寓庄于谐,深入浅出;引号博闻强记,

- a) 本书在 § 11.1 的插语 3 中已经对此进行 了补充,特意还提到了 alloca。
- b) 栈一定是线程安全的, 堆则不一定。既 然 C/C++标准是 undefined, 也可以说是不 安全的,因为不在程序员的掌控之中便可 理解为不安全的。哪怕对于某种实现是安 全的, 但谁也不能保证同样的源码一定在 同样的 OS 或编译器下进行编译。如果确定 了编译器和平台,有些 heap 可以是线程安 全的,但那毕竟过于深入了,超出了本书 的范围。
- c) 采用堆还是栈, 当然与生命周期紧密相 关, 文中对此已作了说明。但与对象大小 同样有关, 栈空间是有限的, 分配过大的 对象将导致栈溢出。在嵌入式开发中,多 用堆分配,就是出于这个考虑。

当然要看适用场景。但提倡接口继承而不 是实现继承, 这是一个一般的原则, 书中 已对此作了详细的说明。

在提到的 12 种语言中, VB 和 Delphi 显然 是更侧重前台的,而且文中也专门提到它 们(尤其是 Delphi)同样可胜任于后台编

Python 的效率不仅比不上 C/C++/Java/C#等 编译语言,也不如 Perl 等解释型语言。在 大多数场合下, 这是公论, 当然不排斥 Python 解释器以后改进的可能。

所以从作者的角度看看,这些对技术的问题的叙述都 是正确的,可以理解的。其实,对于技术问题的理解, 没有一定对或者一定错的结论。

9 提出人: 霍 炬

本书质量比较高,作者知识结构完整、广泛,实例和 的要求。流程图严谨,确实是不可多得的参考书。但相应对读者的要求也会有所提高,缺乏足够实际经验的读者很难领会其内涵,恐怕也不易认同它。作为学习或提高用书有一定难度,但用来作高级的知识普及、开拓服界是相当不错的。其实书中还是有大量观点和我一致,或者说,有经验的工程师,对此书大部分是认同的。阅读过程中,我也的确补充了一些以往不太清楚,或是不太确知的领域知识,也是有收获的。

这本书对读者的理论和实践水平确有一定的要求。

徐 宥 Washington University 就读(博士)

邹 飞 趋势科技(中国)研发中心 资深软件工程师

霍 炬 银杏搜索 创始人 技术 blogger

蒋波涛 宁波市规划与地理信息中心

序

去年3月的一个下午,过于明媚的春光唤醒了一份久违的情怀,书摊上的一本《青年文摘》便成了合宜的载体。与其说是为了阅读,不如说是为了回忆——对20年前读书心境的回忆。孰料读罢开篇,怀旧之窗随即悄然关掩,一扇求新之门却戛然开启。那是一篇人物介绍,讲述一位籍籍无名的年轻人是如何因撰写博客而声名鹊起的。抚卷思之,网络平台已成大众舞台,人人皆可登台献技,自己何不前去一试?心念甫动,顿感技痒难耐,当晚寝不安席,于辗转反侧之中磨出了一本书的轮廓。

尽管钟书先生认为鸡与蛋应为松耦合关系,但一只来历不明的鸡确会招致人们对其产品可靠性的怀疑。故而在介绍《冒号课堂》的创作思路之前,先自我介绍一番。1986 年我怀揣着成为数学家的梦想,考入武汉大学数学系。7 年的大学生涯在浑浑噩噩中度过,毕业后在广州一所高校教了3 年的高等数学。在混沌与迷茫中挣扎了10 年,终于不堪蹉跎,1996 年赴美攻读数学博士。始料不及的是,在大洋彼岸不仅没能一圆数学之梦,反倒从一个未曾碰过鼠标的电脑排斥者变成了一名 IT工作者。"罪魁祸首"正是电脑和 Internet,它们潜移默化地改变了人们的生活和思维方式,我亦未能幸免。1998 年开始选修计算机课程,两年后拿到硕士学位,并在华尔街的一家软件公司找到了工作。2004 年年底,选择回国发展,再度主导了人生的一次急转弯。回到广州后,顺利地进入了一家著名的外企。平淡而安逸的生活似乎注定与我无缘,不久又转去一家小公司作技术总监。如果用一句话来描述自己的职业生涯,那便是:数学是我的初恋情人,计算机是我的终生伴侣。无论成败,都是命运与人生双向选择的结果。

本书的创作虽出偶然,却也有其必然性。一方面,市面上的计算机书籍多为拼凑之作,且不少带有应试教育的痕迹。另一方面,论坛上充斥着各种谬言妄论,人们或目空一切,或人云亦云;每当争论一起,常常硝烟弥漫,出言无状者甚众。如此诸般,不忍卒睹。深感激浊扬清之必要,此念一直郁积于心,终至一朝爆发。自知虽无澄清玉字之力,唯奢念带来一缕清风。

《冒号课堂》采用对话体,是为了借不同背景、不同水平、不同性格的人物之

口,多层次、多维度、多角度地展现知识的内涵与活性。人物皆以标点符号命名,是为了塑造让人过目不忘的形象:冒号善解释,引号善引用,问号善提问,逗号善缓冲,叹号善感叹,句号善总结。此外,6个标点符号还对应着每小节的6个部分:冒号是正文讲解,引号是文献参考,问号是问题列表,逗号是补充插语,叹号是精华预览,句号是本节总结。与一般纯技术类图书不同,本书非常强调学习方法和学习精神的重要性。在内容组织上也一反常规,以思想为主、以知识为辅,以抽象为主、以具体为辅,以范式为主、以语言为辅。人们常把书籍比作一种食物,其实书籍也是一种药物。一本书应当同时提供两种价值:一种是让人获取正确知识的食用价值,一种是让人抛弃错误观点的药用价值。《冒号课堂》更侧重后者,这多少给读者带来一定的阅读障碍,因为抛弃往往比获取更加困难。此外,全书涉及的知识点较多,覆盖的知识面较广,一些流行的语言或技术反被刻意地淡化。假如读者没有足够的计算机理论和实践基础,难免会感到一些困难和不适。古语有云:"学然后知不足",认识到不足何尝不是学习的一种收获呢?从另一面说,假如读者发现书中疏谬,还请不吝赐教,本人将不胜感激。坦而言之,随着写作的深入,自得之心日敛,惴惴之心日甚,正应了上面古语的后半句:"教然后知困"。

本书的完成首先需要感谢 3 位母亲: 我的母亲、我太太的母亲、我女儿的母亲。没有她们默默无闻的支持和帮助,冒号课堂只能在梦中开班。还要感谢博文视点的周筠老师对本书的大力支持和关怀; 白爱萍编辑负责而又耐心,不厌其烦地和我讨论一个个文字和版式的细节; 博文的陈宜、杨小勤、陈琼、徐定翔、许莹、胡文佳等编辑也以同样的热情带给我很好的出版体验。我从其他亲友和网友那里也得到了许多热情的鼓励和有益的启示,是他们让虚拟的课堂变得真实和生动。

郑晖

2009年8月24日干广州

目 录

上篇:	编桯	『范式与编桯语言	1
第	1课	开班导言	3
	1.1	开班发言——程序员的 4 层境界	4
	1.2	首轮提问——什么语言好?	7
	1.3	语言选择——合适的就是好的	10
	1.4	初识范式——程序王国中的世界观与方法论	15
	1.5	开发技术——实用还是时髦?	18
第	2课	重要范式	25
	2.1	命令范式——一切行动听指挥	26
	2.2	声明范式——目标决定行动	31
	2.3	对象范式——民主制社会的编程法则	37
	2.4	并发范式——合作与竞争	43
第	3课	常用范式	49
	3.1	泛型范式——抽象你的算法	50
	3.2	超级范式——提升语言的级别	55
	3.3	切面范式——多角度看问题	63
	3.4	事件驱动——有事我叫你,没事别烦我	69
第	4课	重温范式	85
	4.1	函数范式——精巧的数学思维	86
	4.2	逻辑范式——当算法失去了控制	95
	4.3	汇总范式——一张五味俱全的大烙饼	103
	4.4	情景范式——餐馆里的编程范式	109
第	5 课	语言小谈	119
	5.1	教学计划——接下来的故事	120
	5.2	数据类型——规则与变通	125
	5.3	动态语言——披着彩衣飞舞的脚本语言	131
	5.4	语言误区——语言的宗教情结	137
第	6课	语言简评	145
	6.1	系统语言——权力的双刃剑	146
	6.2	平台语言——先搭台后唱戏	154

6.3	前台语言一	—视觉与交互的艺术	161
6.4	后台脚本—	敏捷开发的利器	167
第:抽象标	机制与对象范	ō 式	175
第7课	抽象封装		177
7.1	抽象思维一	减法和除法的学问	178
7.2	数据抽象—	— "做什么"重于"怎么做"	189
7.3	封装隐藏—	—包装的讲究	200
第8课	抽象接口		211
8.1	软件应变一	—随需而变,适者生存	212
8.2	访问控制—	—代码的多级管理	220
8.3	接口服务一	—讲诚信与守规矩	228
第9课组	迷承机制		237
9.1	继承关系一	-继承财富,更要继承责任	238
9.2	慎用继承—	以谨慎之心对待权力	254
第 10 课	多态机制		277
10.1	多态类型一	静中之动	278
10.2	抽象类型-	实中之虚	291
第 11 课	值与引用		311
11.1	语法类型-	——体用之分	312
11.2	语义类型一	—阴阳之道	322
第 12 课	设计原则		341
12.1	间接原则一	柔胜于刚,曲胜于直	342
12.2	依赖原则一	——有求皆苦,无欲则刚	352
12.3	内聚原则一	不是一家人,不进一家门	363
12.4	保变原则一	与魔鬼打交道的艺术	374
第 13 课	设计模式		385
13.1	创建模式一	——不要问我从哪里来	386
13.2	结构模式一	—建筑的技巧	403
13.3	行为模式-	—君子之交淡如水	418
13.4	闭班小结一	软件无形,编程有道	440
索引			447
设计手记			451
编辑手记			453

第1课 开班导言

课前导读

第1课为整个课堂学习的内容和风格定调,主要围绕3个问题展开: 要成为一个优秀的程序员,最须要学习什么知识?领会什么思想?具备什么精神?



本课共分5节——

- 1. 开班发言——程序员的 4 层境界
- 2. 首轮提问——什么语言好?
- 3. 语言选择——合适的就是好的
- 4. 初识范式——程序王国中的世界观与方法论
- 5. 软件技术——实用还是时髦?

1.1 开班发言——程序员的 4 层境界

授人以鱼, 不如授人以渔。

——古语

关键词:程序员;学习;知识;思想;精神

摘 要:对程序员的一些忠告和建议



预 览

学会不如会学, 会学不如会用, 会用不如被用。

如果知识是水,我们要挖掘最先涌动的泉眼。

如果知识是火, 我们要捕捉起初点燃的火花。

如果知识是树,其树大根深,不究立固之本则无以知过去;其枝繁叶茂,不握 支撑之干则无以知当下,其蓬勃旺盛,不察生长之点则无以知将来。

越是喧嚣的世界,越需要宁静的思考,让躁动的心灵得以平息,让蕴藏的灵性得以释放。

知识之上是思想,思想之上是精神。



提问

软件开发者的成长须要经历哪些阶段?

要想在 IT 业中生存与发展, 传统的学习方式是否够用?

优秀程序员应该具备哪些素质?

讲解

冒号开了个程序员提高班,今天迎来了首期学员,他们是问号、句号、逗号、引号和叹号,皆为 IT 业的新兵。望着台下洋溢着青春与渴望的脸庞,冒号开始了他的开班发言——

大家好! 先自我介绍一下,本人姓冒名号字解之。诸位不必叫我老师,就叫老冒好了。比在座各位痴长几岁,"老"是担得的,"师"却不敢妄言。在下编程多年,自觉小有所成,不敢专藏,特开此班与众共享。虽系一家之言、一孔之见,若能抛砖引玉,又何惧方家之哂?疏谬之处,还望海涵斧正,不致自误误人。

客套已毕,言归正传。本班主要采取讨论的形式,只要是软件开发中值得讨论的,但凡本人力之所及,均可共同探讨。

本班的宗旨是: **学会不如会学,会学不如会用,会用不如被用**。对于一个软件 开发者来说,这意味着 4 个阶段:

学会(知其所然)——掌握一些具体编程知识的初级程序员。

会学(知所以然)——能快速而深刻地理解技术并举一反三的程序员。

会用(人为我用)——能将所学灵活运用到实际编程设计之中的高级程序员。

被用(我为人用)——能设计出广为人用的应用程序(application)、库(libr -ary)、工具包(toolkit)、框架(framework)等的系统分析师和架构师。

至于被用的更高层次,如发明出主流的设计模式、算法、语言,乃至理论等,则可称得上计算机专家了。本班的目的,正是为各位向更高阶段的提升助一臂之力。

大家可能都习惯了在小学、中学和大学里的课堂,那里的知识大多是系统而完备且貌似终极的,那里的学习大多是单向而被动的。但习惯并不意味着享受,更多的是因为别无选择。你们曾被引入一座座知识殿堂,被告知它们如何美轮美奂、巧夺天工,尽管很多时候你们或不以为然、或不解其妙,但还是不得不记下每一处被指点的细微结构。很少有人带你们看看当初为建造这些殿堂而打下的地基、搭设的脚手架,哪怕只是上漆前的模样也好,更遑论一瞻数易其稿的设计图纸了。那些与殿堂相比显得有些原始、甚至丑陋的东西,被有意无意地挡在视线之外。可没有那些,你们将来如何为这些宫殿添砖加瓦,又如何另起楼阁呢?

中国学生恐怕是世界上最擅长考试、最习惯考试、也最厌倦考试的群体了。你

们告别了学生生涯,踏上了职业之旅。首先我要恭喜你们,脱离苦海了!同时也要 悲告你们,掉进火坑了!危言耸听吗?如果你选择了做程序员,你时时都得学习, 没有手把手教你的老师,没有指定的教科书和参考书,有的是层出不穷令人眼花缭 乱的新概念、新技术、新问题,好不容易学到一些皮毛,有的已成明日黄花。你时 时都得考试,每提交一段代码就是上交一份答卷,你不知道什么时候、什么人会批 改,直到——开发组同事发现你的代码难以看懂,系统分析员指出你的程序不符合 规范,测试工程师检验到你的软件有缺陷,客户抱怨你的产品太慢太难用,最后老 板倒可能告诉你一个好消息:明天起放长假!

其实,又有哪行哪业的人不需要学习和考试呢? IT 业只是相对更激烈、更富挑战性而已。在这个瞬息万变、适者生存的时代,如果还沿用封闭、被动的学习方式,恐有淘汰之虞。有鉴于此,本班的风格与你们习惯的课堂氛围有所不同: 这里的知识不一定是系统或完备的,但一定是生动鲜活的。如果知识是水,我们要挖掘最先涌动的泉眼; 如果知识是火,我们要捕捉起初点燃的火花。如果知识是树,其树大根深,不究立固之本则无以知过去; 其枝繁叶茂,不握支撑之干则无以知当下; 其蓬勃旺盛,不察生长之点则无以知将来。这里的问题不一定是预设的,结论不一定是终极的,甚至不一定是正确的,但一定是有的放矢、发人深思的。由此决定了这里的学习方式将是开放多元、双向互动的。

越是喧嚣的世界,越需要宁静的思考,让躁动的心灵得以平息,让蕴藏的灵性得以释放。学习编程没有速成大法。没有必杀之技、没有锦囊秘笈、没有终南捷径,只有思考、实践、再思考、再实践。中国的 IT 界乃至整个学术界都过于浮躁和急功近利,既盲从又自大,缺乏务实精神与研究精神、独立精神与合作精神、批判精神与自省精神。如果一个程序员沾染这种风气,哪怕有再好的学习方法和学习能力,他都注定与"优秀"绝缘。这就是本班极力倡导并将贯穿始终的理念——知识之上是思想,思想之上是精神。

我的开场白到此为止,现在把话语权交给你们,大家自由发问吧。

总结

▶ 软件开发者的成长阶段: 学会→会学→会用→被用。

这是一个从"知其所然"到"知所以然"、从"人为我用"到"我为人用"的历程。

▶ 传统的学习方式大多有如下特征:

封闭——系统完备的终极式知识。

单向——师教生学的单向式传输知识。

被动——师命生从的被动式接受知识。

静态——只注重知识的现状,忽略知识的起源、历程和未来趋势。

继续沿袭这种学习方式,是很难在竞争日趋激烈、技术日新月异的IT业中 求生存、谋发展的。开放多元、双向互动的现代课堂乃大势所趋。

"知识之上是思想,思想之上是精神。"

一个优秀的程序员,除了要迅速掌握知识、善于领悟思想外,还必须具备 务实与研究精神、独立与合作精神、批判与自省精神。

1.2 首轮提问——什么语言好?

敬畏老师莫如敬畏真理。

题记

关键词: 计算机语言: 编程语言 摘 要: 讨论流行的计算机语言



预览

真正的老师是你自己。

没有激情作氧气, 灵感的火花注定转瞬即灭。



提问

谁是你真正的老师?

程序员是吃青春饭的吗?

计算机语言这么多, 到底学哪个好?

讲解

众人面面相觑,一阵沉默后开始窃窃私语,显然有些不太习惯这种教学方式—— 笔记本上还没写两个字呢,老师就把球给踢回来了。

冒号也不说话, 只是微笑地望着大家。

还是问号打破僵局,开始发问:"老师——"

冒号扬手打断他:"请不要管我叫老师,**真正的老师是你自己**。本班的一个特色是:师生角色模糊,大家自主学习,相互启发,教学相长。"

"老冒——"问号顿了顿,全班哄堂大笑,"学软件开发,当然得先学语言, 计算机语言这么多,到底哪个好,或者说学哪个好?"

冒号笑道: "这个问题很典型,很实在,也很初级。"

问号被"初级"这个字眼刺得面上一红。

"如果信奉流行的就是好的,那么也许可以给你一个参考答案。"冒号转身在 黑板上写下一串清单——

Java(19.40%) C(15.84%) C++(9.63%) VB(8.84%) PHP(8.78%) C#(5.06%) Python(4.57%) Perl(4.12%) Delphi(3.62%) JavaScript(3.54%) Ruby(3.28%) D(1.26%)

"根据TIOBE^[1]截至 2009 年 2 月份的统计结果,选出以上流行度超过 1%的 12 种编程语言。从中可以看出,它们的总占有率接近 90%,应该算得上是当今主流语言的代表。尽管有人置疑TIOBE排名的权威性和合理性,但这份名单应该还是八九不离十的。"冒号道。

引号很疑惑: "怎么可能那么流行的 ASP 和 JSP 都不在其中呢?"

"对啊,"逗号附和着,"还有 HTML 和 XML 怎么不算呢?"

冒号解释道: "ASP、JSP和PHP是动态网页最流行的 3 种解决方案。动态网页的实现方式很多,但它们采取的几乎是同样的方式——在静态网页中植入一些能在服务器端运行的代码。在ASP和JSP中,这些代码并不涉及新的语言,故称之为模板、框架或脚本环境更合适些。PHP则不同,本身是一种新的编程语言,并且除了应用于服务端外,还能编写命令行脚本和桌面应用程序。至于HTML和XML,还有XHTML、WML等,均为SGML(Standard Generalized Markup Language)的子集,属于标记语言(Markup Language)。与通常意义上的编程语言有所不同,它们是处理的对象,而不是处理的主体。可以说它们更接近数据格式标准,正如CSV和



[1]TIOBE (http://www.tiobe.com)是一家评估编程语言流行度的权威机构,每月公布一次编程语言排行榜。

JSON一样。当然也不绝对,XSLT是一种特殊的XML,包含变量定义和处理逻辑,更学术地说,它是图灵完备的(Turing-complete)^[2],应当属于编程语言。"

问号杀了个回马枪: "那 CSS、RSS 算是编程语言吗?"

冒号从容作答:"与 XSLT 类似, CSS 是一种样式语言(Stylesheet Language),但不是以 XML 的形式出现的。它将传统的 HTML 中的样式逻辑提炼出来,大大丰富和简化了 HTML。不过它没有执行指令或运算,更谈不上图灵完备,因此不属编程语言。至于 RSS,只是一种用 XML 来描述的数据交换规范,甚至连计算机语言都算不上。"

叹号也插了进来:"近来网络开发语言 AJAX 特别火,难道不算编程语言吗?"

冒号摇头道: "的确有不少人以为 AJAX 是一门语言,但如果知道 AJAX 是Asynchronous JavaScript And XML 的简称,便知其谬矣。事实上,它是综合了JavaScript、XML、HTML、CSS 等多种语言的一种网络应用技术。"

"就算这些不是编程语言,那也是计算机语言或与语言密切相关的技术,该学的还是得学。"句号想起问号开始问的是计算机语言,老冒有偷换概念之嫌。

"不错,"冒号点点头,"不仅要学语言,还要熟悉相应的开发环境和开发工具等,当然最重要的是学习其中的思想。"

"唉,学完这些头发都白了,程序员可是吃青春饭的。"叹号叹息道。

冒号扫视了一下,说道:"现在班上每个人都尊口已开,这是一个很好的开始。 开放言论才能解放思想,思想解放了才能产生灵感和激情。缺乏灵感和激情的程序 员,学习起来吃力,工作起来辛苦,最后就会感慨这是吃青春饭的职业。"

叹号不好意思地挠了挠头。

逗号接言: "灵感嘛,偶尔也许能闪一下,激情可就难喽!

冒号注视着他,一字一顿地说:"没有激情作氧气,灵感的火花注定转瞬即灭。"

总结

- ▶ 本班倡导自主学习、相互启发,真正的老师不是别人,正是自己。
- 当今主流语言的代表: Java、C、C++、VB、PHP、C#、Python、Perl、
 Delphi、JavaScript、Ruby 和 D。



插语

[2]一个能计算出每个图 灵可计算函数(Turingcomputable function)的 计算系统被称为图灵完 备的。一个语言是图灵完 备的,意味着该语言的计 算能力与一个通用图灵 机 (Universal Turing Machine)相当,这也是 现代计算机语言所能拥 有的最高能力。 "程序员是吃青春饭的职业"出自那些缺乏灵感和激情的人之口。

1.3 语言选择——合适的就是好的

尺有所短, 寸有所长。

-《楚辞》

关键词: 计算机语言: 低级语言: 高级语言: 中级语言

摘 要: 简要回顾计算机语言



预览

评判语言优劣,如同争论兵器高下,倘若撇开使用的主体和对象,皆为空泛之 谈。

高级语言好比长兵器,威力强大却难免滞重,长于大型应用,可谓"一寸长, 一寸强":低级语言好比短兵器,轻便灵活却难免有风险,长于底层应用,可谓"一 寸短,一寸险"。

西门吹雪的西来一剑, 那是 C 语言: 李寻欢的小李飞刀, 那是汇编语言: 陆 小凤的灵犀一指, 那是机器语言。



提问

语言好坏的标准是什么?

计算机语言的发展经历了哪几个阶段?

第4代语言和第5代语言与前3代语言相比,有什么不同?

什么是低级语言和高级语言?它们各自的特点与应用范围是什么?

为什么称 C 语言为中级语言?

计解

问号觉得自己的问题并未解决,追问:"这么多种语言,仅凭流行度就能分出 主次优劣吗?"

"流行度当然不是唯一的指标。"冒号答道,"语言的主次优劣因人而异,答案在你们自己身上。还是刚才那句话,真正的老师就是你自己。"

期待的目光如风中之烛般开始黯淡。

冒号又道:"评书里名师授艺时,常常要徒弟自己挑选称手的兵器。威武的刀,灵活的枪,飘逸的剑,浑厚的棍,粗犷的斧,霸道的锤,诡异的鞭,无不谙合武者的个性。评判语言优劣,如同争论兵器高下,倘若撇开使用的主体和对象,皆为空泛之谈。"

句号若有所悟: "所以好的语言就是适合编程者和解决对象的语言。"

"非常正确!"冒号赞许道,"这就是问号同学要的答案。"

引号并不满足:"可我记得评书里经常描述高手的一句话:十八般兵器样样精通。"

冒号一笑: "兵器虽多,其理相通,高手精通多种兵器何足为奇?但如果让赵云使锤,李元霸使枪,武力恐怕还是要大打折扣吧?"

逗号依然困惑: "我们如何判断一种语言是否适合自己,是否适合解决对象呢?"

冒号看出大家共同的疑惑,不紧不慢地说:"要想从中选择,自然必须先了解,不然怎知兵器称不称手、合不合用?现在进入正题,我们先对计算机语言作个简要的回顾。"

大伙均想,总算要挠着痒处了。

"计算机语言按其发展历程通常分为5代。"冒号说完,在黑板上写下——

第1代语言(1GL): 机器语言;

第2代语言(2GL): 汇编语言——IA-32 Assembly、SPARC Assembly等;

第3代语言(3GL): 高级语言——Fortran、Pascal、C、Java、VB等;

第4代语言(4GL):面向问题语言——SQL、SAS、SPSS等;

第5代语言(5GL):人工智能语言——Prolog、Mercury、OPS5等。

[&]quot;谁能简要地谈谈这段历史?"冒号又开始踢回传球了。

"最新的两代语言我不是特别熟悉,就说一下前几代吧。" 一阵沉默后,引 号终于毛遂自荐,"计算机语言是人用来指挥计算机的语言,而计算机只懂一种语 言——由0和1组成的机器语言(machine language)。最初人们直接用这种语言下 达指令,可它们实在太难记忆和阅读了,开发和维护起来既费时又易错,严重桎梏 了程序员的生产力。后来人们发明了汇编语言(assembly language),用接近英语 单词的助记码(mnemonic code)来代替 0、1 串,由助手——汇编器(assembler) 将其转化为机器语言。这些助手很称职,但有两个缺点:一是毫无主见,基本上只 会一一对应地翻译,程序员必须不厌其烦地交代每一个细节;二是不知变通,换种 机器就傻眼了。于是人们陆续引进了各种高级语言(high-level programming language),同时启用更得力的助手——编译器(compiler)和解释器(interpreter)。 这些助手除了能理解更简洁更抽象的高级语言外,还能因地制宜地对一些指令进行 优化处理。程序员的劳动力得以极大的解放,生产效率得以大幅的提升。直到现在, 高级语言还是最主要的开发语言,包括前面提到的12种最流行的语言。"

引号发言甫毕,冒号立即献上溢美之词: "精彩!精当!一气呵成!看看,你 还怀疑自己不够格作老师吗?"

一种晕眩感向引号袭来。

冒号继续引号的讲述: "从机器语言到汇编语言,再到高级语言的演变,堪比 从徒步行走到乘自行车,再到乘汽车的变革,越来越省时、省力、省心。循此方向, 第4代语言更专注业务逻辑和问题领域。程序员主要负责分析和描述问题,不再花 大量时间去考虑具体的算法和逻辑。事实上,最初提出第4代语言的概念,就是希 望非专业程序员都能做应用开发。"

逗号心下一惊: "那我等岂不是要失业了?"

冒号宽慰道: "倒不用太担心。正如引号所说,语言越来越高级,背后靠的是 越来越能干的助手。这些助手本身就是软件,还是需要专业程序员开发的。更何况, 这种理想的全面实现依然任重而道远。"

问号百思莫解: "第4代语言到第5代语言的发展路线似乎不够清晰,在逻辑 上如何解释呢?"

冒号作出解答: "第4代语言虽然足够强大,但过于局限某些特定领域,基本 上属于领域特定语言[1] (Domain Specific Language, 简称DSL), 而不是我们所熟 悉的通用编程语言(General-Purpose Programming Language,简称GPPL)。专门用 于数据库操作的SQL、用于统计分析的SAS和SPSS、用于科学计算的Mathematica



[1]领域特定语言,简称 DSL。它区别于通用语 言,一般用于特定的问题 领域, 多属于第 4 代语 言。比如, SQL 是专门 针对数据库的语言, LaTeX 是专门用于排版 的语言, 正则表达式 (regular expression) 是 专门处理字符匹配的语

都是典型的第4代语言。然而一个系统往往横跨多个领域,如果每个领域使用不同的语言,并且不同领域的语言在概念和方法上也不统一,必然会给集成和整合带来困难。第5代语言在保持第3代语言的通用性的前提下,继承了第4代语言的优点,即重在目标而非过程、重在描述而非实现。如果把这种优点用在汽车上,那么下一代的交通工具也许是无人驾驶的智能汽车。只要输入目的地,它会自动通过GPS寻找最佳路径,自动根据路况变速转向,一直驶到终点。"

叹号身形微颤: "坐这种车我可不放心。"

冒号一撇嘴:"这当然只是一种假想。同样地,第5代语言号称人工智能语言,虽然雄心勃勃,试图让机器理解人类的自然语言,并且具备人类的思维能力,但目前看来这一目标还显得遥不可及。"

句号很赞同: "是啊,超级计算机虽然可以战胜国际象棋的世界冠军,但在围棋上弱智得很。"

冒号提纲挈领: "也有人简单地将前两代语言统称为低级语言,其他的统称为高级语言。语言从低级到高级,离机器语言更远,离人类语言更近,因而更易读写、调试和维护,安全性、通用性和可移植性更强,开发效率更高,更加抽象和宏观;但同时运行速度和效率下降,用法和功能上局限性更大。如果拿兵器作比,高级语言好比长兵器,威力强大却难免滞重,长于大型应用,可谓'一寸长,一寸强';低级语言好比短兵器,轻便灵活却难免风险,长于底层应用,可谓'一寸短,一寸险'。"

大伙心里话, 敢情来这儿不是学编程, 是学武术的。

叹号说道: "我看还是高级语言好,现在谁还学低级语言啊?"

冒号纠正道: "低级语言并不低级,只是随着高级语言的出现,计算机硬件性能的提高,渐渐有些边缘化了。虽然几乎没有人再用机器语言编程了,汇编语言仍有其用武之地。常见的有:包括嵌入式系统在内的系统开发,如操作系统、编译器、驱动程序、无线通信、DSP、PDA、GPS等;其他对资源、性能、速度和效率极为敏感的软件开发;以信息安全、软件维护与破解等为目的的逆向工程等。即使你不打算从事系统开发,也不想作红客、黑客或骇客,掌握汇编语言对你深入了解计算机内部运行机制、调试软件和改进程序中某些关键代码的算法也是有帮助的。"

引号提出: "好像有些书上把 C 语言称为中级语言。"

冒号答道: "这是因为 C 兼具高级语言和低级语言的特征。一方面它提供了

高层抽象和可移植性,使程序员更多地专注问题逻辑而不是机器逻辑:另一方面它 也提供诸如指针、位字段(bitfield)等工具进行底层操作,甚至可直接内嵌汇编代 码。C语言既简洁灵活又高效强大,是迄今为止最具影响力的语言。几乎所有的操 作系统和大多数高级语言都用它来实现, C 家族的语言 C、C++、Java、C#、D、 Objective C 等占据主流语言的半壁江山。如果再拿兵器作比, C 语言就是一把剑, 轻灵飘逸、锐利快捷。一名武将无论擅用什么兵器,往往都会腰悬宝剑。不会 C 的程序员正如不会使剑的武将,无论如何都是一种缺憾。相比之下,汇编语言就像 小刀匕首,而机器语言则近乎赤手空拳了。"

句号灵光一闪: "我明白了——西门吹雪的西来一剑,那是 C 语言: 李寻欢的 小李飞刀, 那是汇编语言, 陆小凤的灵犀一指, 那是机器语言。"

大家会心地笑了。

逗号冷不防冒出一句: "我会跆拳道哦!"

句号一乐: "哈哈,等你打赢了陆小凤,就封你为机器语言。"

冒号也笑言:"这位是古龙的粉丝吧?武侠小说里的侠客多轻功高绝且喜单打 独斗,故使用轻、短兵器居多;而历史小说里的战将多骑马进行大规模作战,故除 了佩剑外,使用重、长兵器居多。这就是前面提到的,中低级语言更适合中小型或 底层应用, 高级语言更适合大型应用。"

众人活跃起来,开始议论纷纷。冒号放耳听去,净是些古龙金庸、三国水浒里 的人物情节,暗想:通俗小说到底比计算机编程更吸引人啊。

总结

- ▶ 评判语言优劣,不能离开使用语言的主体和对象。好的语言就是适合编 程者和解决对象的语言。
- 计算机语言按其发展历程分为 5 代,依次为:机器语言、汇编语言、高 级语言、面向问题语言和人工智能语言。通常,前两代统称为低级语言, 后面的统称为高级语言。
- ▶ 第4代语言和第5代语言与前3代语言最大的不同在于: 重目标轻过程、 重描述轻实现。
- ▶ C兼具高级语言和低级语言的特征,因此也被称为中级语言。

▶ 计算机语言从低级发展到高级,渐渐远离机器,靠近人类,以牺牲部分性能和效率为代价,换来更高的开发效率和可维护性。中低级语言更适合中小型或底层应用,高级语言更适合大型应用。

1.4 **初识范式**——程序王国中的世界观与方法论

言者所以在意,得意而忘言。

——《庄子·外物》



得形而忘意, 无异舍本逐末: 得意而忘形, 方能游刃有余。

编程之时,你便进入自己创造的世界之中。这是你的世界,只有注入你的想象力、创造力和激情,它才有勃勃生机。你编写的岂止是代码,分明还有乐曲;你敲击的岂止是键盘,分明还有琴键;你运行的岂止是程序,分明还有世界。当优美的旋律奏起,整个世界都随之翩然起舞,一种莫可名状的满足是否会充溢你的全身?

找对象是"对象导向"的,去约会是"面向对象"的。



提问

什么是编程范式?

编程范式与编程语言的关系是什么?

🚼 讲解

问号第一个从小说里走出来,问道: "刚才谈到了低级语言和中级语言,现在

该谈高级语言了吧?"

冒号微叹: "高级语言大概有近千种,流行的也不下几十种,有时候选择过多 反而无所适从啊。"

逗号不以为然: "最流行的不就那么几个: Java、C++、C#还有 VB 吗?"

不意此言遭到冒号连珠炮似的反问:"可你知道它们为什么会流行吗?是不是 学会这几样就是一个合格的程序员了?它们会不会变得不那么流行,甚至被其他语 言取代?如果不会,为什么?如果会,又怎么办?"

逗号赧然语塞。

冒号口气放缓: "掌握一门语言的语法、工具和技巧固然重要,但那只相当于 学会一门兵器的招法, 更重要的当然是心法。招法重形, 心法重意。得形而忘意, 无异舍本逐末:得意而忘形,方能游刃有余。下面要谈的就是一种心法:编程范式。"

问号不解: "编程范式? 听上去很学究, 那是什么东东?"

冒号续道: "范式译自英文的 paradigm,也有译作典范、范型、范例的。所谓 编程范式(programming paradigm),指的是计算机编程的基本风格或典范模式。 借用哲学的术语,如果说每个编程者都在创造虚拟世界,那么编程范式就是他们置 身其中自觉不自觉采用的世界观和方法论。"

叹号吸口气: "好抽象哦!"

句号心中一动:"您是说我们都是虚拟世界的创造者,都在创造自己的黑客帝国?" 大家不禁莞尔。

冒号动情地说: "难道不是吗?只不过帝国有大小之分、优劣之别罢了。编程 之时,你便进入自己创造的世界之中。这是你的世界,只有注入你的想象力、创造 力和激情,它才有勃勃生机。你编写的岂止是代码,分明还有乐曲;你敲击的岂止 是键盘,分明还有琴键:你运行的岂止是程序,分明还有世界。当优美的旋律奏起, 整个世界都随之翩然起舞,一种莫可名状的满足是否会充溢你的全身?"

大家都被冒号诗化的语言感染了,没想到编程也可以如此感性。

良久,引号试探地问:"面向对象编程就是一种编程范式吧?"

冒号点头: "不错,它是时下最流行的一种编程范式。顺便说一句,'面向对 象'译自 Object-Oriented,但'面向'二字令人费解。据说有本书叫'面向对象方 法',比别的计算机书都畅销,知道为什么吗?不少同学把它当成恋爱指南买走了。" 全班笑倒。

冒号认真地说: "将Object-Oriented译成'对象导向'^[1],虽然稍嫌拗口,但更贴切。并非刻意要咬文嚼字,这关系到对编程范式的理解。我们知道,编程是为了解决问题,而解决问题可以有多种视角和思路,其中普适且行之有效的模式被归结为范式。由于着眼点和思维方式的不同,相应的范式自然各有侧重和倾向,因此一些范式常用'oriented'来描述。换言之,每种范式都引导人们带着某种的倾向去分析问题、解决问题,这不就是'导向'吗?而'面向'的宾语往往是预先确定的目标,如面向世界、面向未来、面向用户、面向问题等。此外,'面向'强调静态结果,而'导向'强调动态趋势,显然后者更生动,也更符合编程的特质^[2]。"

句号一语惊人: "找对象是'对象导向'的,去约会是'面向对象'的。" 全班再倒。

句号得意地解释:"按梦中情人的标准去找对象,具体目标未定但选择倾向已定,这就是一种导向,而且是对象导向。找到之后再约会,不就面向对象了吗?" 众人称绝。

"我们是来谈编程范式的,不是来谈对象的。"冒号一脸的道貌岸然,"编程范式是抽象的,必须通过具体的编程语言来体现。它代表的世界观往往体现在语言的核心概念中,代表的方法论往往体现在语言的表达机制中。一种范式可以在不同的语言中实现,一种语言也可以同时支持多种范式。任何语言在设计时都会倾向某些范式,同时回避某些范式,由此形成了不同的语法特征和语言风格。"

● 总结

- ▶ 编程范式是计算机编程中的基本风格和典范模式,是编程者在其所创造的虚拟世界中自觉不自觉采用的世界观和方法论。每种范式都引导人们带着其特有的倾向和思路去分析和解决问题。OOP就是一种编程范式。
- Dobject-Oriented 多译作"面向对象",但不如"对象导向"贴切。
- 如果把一门编程语言比作兵器,它的语法、工具和技巧等是招法,它采用的编程范式则是心法。
- 抽象的编程范式须要通过具体的编程语言来体现。范式的世界观体现在语言的核心概念之中,范式的方法论体现在语言的表达机制中。一种语言的语法和风格与其所支持的编程范式密切相关。



插语

[1]港澳台地区将其译为 "物件导向"。即使单从 字面上翻译,oriented 是 "以……为方向的; 以……为目的的;导向 的;定向的"的意思,也 比译为"面向"更合适。

[2]作为类比, 经济学中的"market-oriented"译为"市场导向(或取向)的"的远多于译为"面向市场的"。

6 参考

- [1] Wikipedia. Programming paradigm. http://en.wikipedia.org/wiki/Programming_paradigm
- [2] Stephen H. Kaisler. SOFTWARE PARADIGMS. New Jersey: Wiley, 2005. 21-22

1.5 开发技术——实用还是时髦?

借我借我一双慧眼吧,让我把这纷扰看得清清楚楚明明白白真真切切。

——《雾里看花》

关键词: 开发技术;编程范式;框架;设计模式;架构;库;工具包;惯用法

摘 要: 关于框架; 设计模式; 架构和编程范式等开发技术的讨论



预 览

任何概念和技术都不是孤立的,如果不能在纵向的时间和横向的联系中找准坐标,便似那群摸象的盲人,各执一端却又自以为是。

库和工具包是为程序员带来自由的,框架是为程序员带来约束的。

设计模式是软件的战术思想,架构是软件的战略决策。

知识的学习有几种方式:一种靠记忆,一种靠练习,一种靠培养。

学习编程范式能增强编程语言的语感。



提 问

库和工具包与框架有何不同?

什么是设计模式、惯用法和架构?

为什么要谈编程范式,而不是框架、设计模式或架构?

讲解

"现在我们具体介绍一下编程范式。"冒号忽然顿住,隐觉一抹失望从众人脸上掠过,问号更是欲言又止,便鼓励他开口。

问号略显迟疑: "您说编程范式是一种心法,那框架、设计模式还有架构呢?" "原来如此!"冒号心下了然,"让我说说你们最想听些什么吧。"

众现不信之色。

冒号说道:"一种是具体而实用的,最好能立马解决学习和工作中的问题;一种是时髦而花哨的,管它有用没用,不学点心里就是不踏实。"

众人虽觉此话有些尖刻,细想起来也有几分道理,但老冒明知而不为,不走群众路线,偏去扯什么劳什子的范式——当然,直接谈 OOP 倒是不错的。

"自以为懂的未必真的懂,自以为不懂的未必真的不懂。" 冒号玩起了玄学, "有些概念和技术即使背得烂熟,甚至用得烂熟,那也不代表真正掌握;有些概念 和技术看起来很新奇,却不过是新瓶装旧酒。"

叹号颇不服气: "用得烂熟都不算掌握,难不成只有发明概念和技术才算掌握?"

"哈哈,那倒不必。"冒号笑道,"用得烂熟不等于用得恰到好处,能解决问题不等于没有后顾之忧。"

逗号问道: "那掌握的标准是什么?"

"许多应聘者喜欢在简历中言必称精通某某语言、某某技术云云,大多不必面试即知其大言炎炎——倘若真的精通,他自当应聘更高的职位。"冒号有感而发却又似不着边际,"任何概念和技术都不是孤立的,如果不能在纵向的时间和横向的联系中找准坐标,便似那群摸象的盲人,各执一端却又自以为是。"

众人心想,老冒虽言辞旦旦却有凿空之嫌,一节课下来,天马行空的扯了不少, 真刀真枪的一个也无,该不是只会纸上谈兵吧?

句号紧扣主题:"您为何选择谈编程范式,而不是框架、设计模式还有架构呢? 难道它们真如您所说只是时髦而花哨的东西吗?" "我可没这么说。"冒号矢口否认,"但在弄清一样东西存在的意义之前就随众跟风,早晚会跟丢的。我先问问你们:什么是框架(framework)?它与一般的库(library)和工具包(toolkit)有何不同?"

引号应答: "框架就是一组协同工作的类,它们为特定类型的软件构筑了一个可重用的设计。与库和工具包不同之处在于前者侧重设计重用而后两者侧重代码重用。"

"嗯,有点标准答案的味道。"冒号夸道,"如果吹毛求疵的话,框架并不限于 OOP,可以是协同工作的类,也可以是协同工作的函数。一个足够复杂的应用软件开发,为确保快速有效,通常采取的方式是:在宏观管理上选取一些框架以控制整体的结构和流程;在微观实现上利用库和工具包来解决具体的细节问题。框架的意义在于使设计者在特定领域的整体设计上不必重新发明轮子;库和工具包的意义在于使开发者摆脱底层编码,专注特定问题和业务逻辑。"

问号提出问题: "框架与库和工具包看起来很相似——都是一些代码集合,都提供一些 API(应用编程接口),是什么使得它们不同呢?"

"问得好!"冒号赞言,"框架与工具包最大的差别在截然相反的设计理念上: **库和工具包是为程序员带来自由的,框架是为程序员带来约束的**。具体地说,库和工具包是为程序员提供武器装备的,框架则利用控制反转(IoC)^[1]机制实现对各模块的统一调度,从而剥夺了程序员对全局的掌控权,使他们成为手执编程武器、随时听候调遣的士兵。"

叹号苦着脸: "程序员原来就是一小卒子啊!"

"哪个将军不是从小卒做起的?"冒号反问道,"不错,框架是在语言的语法规则之外施加于程序员的又一层枷锁,但没有规矩不成方圆。正如行军打仗,讲究排兵布阵,程序员就是那兵,框架就是那阵。"

句号说:"可不可以这么理解,框架就是一些人——也就是框架设计者,把一个软件开发中最甜的部分啃掉了,剩下部分留给下面的人?"

"从某种意义上说,是这样。"冒号点点头。

逗号很不甘心: "我就想啃最甜的部分。"

"当心别把牙给崩掉。"冒号笑道, "不是打击你,首先你还没那本事; 其次即使你有本事也未必有机会; 最后即使有本事也有机会,重新设计框架也未必是好的选择。就说大名鼎鼎的 Struts 吧,哪怕你设计出比它更高明的框架也难以被采用,



[1]控制反转(Inversion of Control)是一种软件设计原则。与通常的用户代码调用可重用的库(library)代码不同, IoC 倒转了控制流方向:由库代码调用用户代码。有人将此比作好菜坞法则: "不要打电话给我们,我们会打给你的"。

因为前者早已成为 Java 平台上网络开发的事实(de facto)标准,公司很容易从市场上招到懂 Struts 的程序员,不必培训即可上手,成本低见效快。过去许多公司都有自己的网络框架,但最后大多都放弃了,并不是因为 Struts 更优秀,而是因为它更普及。毕竟大多数软件开发是以金钱而不是技术为中心的。"

问号提议: "您能谈谈设计模式和架构吗?"

冒号侃侃而谈:"与前面说的框架与库和工具包不同,设计模式(design pattern)和架构(architecture)不是软件产品,而是软件思想。设计模式是软件的战术思想,架构是软件的战略决策。设计模式是针对某些经常出现的问题而提出的行之有效的设计解决方案,它侧重思想重用,因此比框架更抽象、更普适,但多限于局部解决方案,没有框架的整体性。与之相似的还有惯用法(idiom),也是针对常发问题的解决方案,但偏重实现而非设计,与实现语言密切相关,是一种更底层更具体的编程技巧。至于架构,一般指一个软件系统的最高层次的整体结构和规划,一个架构可能包含多个框架,而一个框架可能包含多个设计模式。"

引号愈发疑惑: "这些不是都很重要吗?"

"当然都很重要。不过——"冒号话锋一转,"在没有打好基础前,架构只是空中楼阁,因此不可能现在谈它。至于框架,不同的应用领域有不同的框架,如表现层的Struts、业务层的Spring、持久层的Hibernate,等等,即使相同领域的框架也有多个选择,更不用说不同的语言框架还不一样,从何谈起?再说框架其实一点也不高深,完全可以无师自通,关键是领会思想,多学习多实践。说到设计模式,一共就那么几十个,一本'四人帮'(GoF)[2]的书足矣,自己慢慢去啃,又何须多谈?简言之,一个谈之过早,一个无从谈起,一个不必多谈。"

下面开始交头接耳窃窃私语起来。

"知识的学习有几种方式:一种靠记忆,一种靠练习,一种靠培养。就拿英语学习来说吧,学单词,单靠记忆即可;学句型、语法,光记忆是不够的,须要勤加练习方可熟能生巧;而要讲出地道的英语,光记忆和练习是远远不够的。从小学到大学,甚至博士毕业,除了英语类专业的学生外,大多数人英语练了一二十年,水平如何?不客气但很客观地说:一个字,烂;两个字,很烂;三个字,相当烂!口语甚至连一个英语国家的三岁小孩都不如。"冒号越说越激动,"原因只有一个,那就是国内的英语教学方式严重失策。教学总是围绕单词、词组、句型、语法转,缺乏对语感的重视和培养,导致学生只会'中式英语'。同样道理,一个惯用 C语言编程的人也许很快就能写一些 C++程序,但如果他只注重 C++的语法而不注重



插语

[2]设计模式最经典书籍 《Design Patterns: Elements of Reusable Object-Oriented Software》的四位作者常 被称为 GoF 或 Gang of 培养 OOP 的语感,那么写出的程序一定是'C式 C++'。与其如此,倒不如直接 用 C 呢。"

句号悟道: "您是想告诉我们,学习编程范式能增强编程语言的语感?"

"一语中的!"冒号庆幸总算没有白费口舌,"语感是一个人对语言的敏锐感知 力,反映了他在语言方面的整体上的直觉把握能力。语感强者,能听弦外之音,能说 双关之语,能读隽永之作,能写晓畅之文。这是一种综合的素质和修养,其重要性是 不言而喻的。那么如何培养语感呢?普通的学习和训练固不可少,但如果忽视语言背 后的文化背景和思维方式,终究只是缘木求鱼。编程范式正体现了编程的思维方式, 因而是培养编程语言的语感的关键。现在如果我开始介绍范式,你们还有意见吗?"

众人受了鼓动,个个把头摇得跟拨浪鼓似的。

冒号语重心长地说: "既然范式关乎语感,就须要慢慢地培养和渗透,不可能 一蹴而就,因此有些地方不太明白也没关系。现在只是撒下一些种子,慢慢地会生 根发芽,直至长成大树。那些设计模式、框架,甚至架构,等看似神秘高深的东西, 都会自然而然地在这棵大树上结出果实。到那时,你们个顶个的都是内外兼修的武 林高手了。怎么样?大家准备好了吗?"

"准备好了!"众人齐声道, 求知的目光再度点燃。

"准备好了就下课吧。"冒号狡笑着, "下节课,下节课我们再谈。"

总结

- 库和工具包侧重代码重用,框架侧重设计重用。库和工具包从微观上解 决具体问题, 是为程序员带来自由的: 框架从宏观上控制软件整体的结 构和流程,是为程序员带来约束的。框架是通过控制反转(IoC)机制反 客为主的。
- ▶ 设计模式是软件的战术思想,架构是软件的战略决策。与框架、库和工 具包不同,它们不是软件产品,而是软件思想。
- 设计模式与惯用法都是针对常发问题的解决方案, 但前者偏重设计, 后 者偏重实现。
- 架构太高, 谈之过早; 框架太多, 无从谈起; 设计模式太少, 不必多谈。 至于编程范式,对培养编程语言的语感至关重要,需要充分的重视和长 期的积累,方能悟其精髓。

6 参考

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns:
 Elements of Reusable Object-Oriented Software. Boston, MA: Addison-Wesley, 1994. 26-28

课后思考

- 01-01 作为一个软件开发者, 你现在处于哪个阶段? 你未来的目标是什么?
- 01-02 传统的学习方式的弊端在哪里? 你是否有切肤之痛?
- 01-03 你认为一个优秀的程序员须要具备什么素质和精神?
- 01-04 你了解哪些计算机语言? 你对一门语言的取舍与喜恶的根据是什么?
- 01-05 你认为计算机语言未来的发展方向是什么?
- 01-06 你能否在编程中感受到自己的激情和灵性?
- 01-07 你了解哪些框架?它们主要解决了哪些问题?应用范围是什么?实现的机理是什么?
- 01-08 你了解哪些设计模式?它们为什么能成其为模式?

第3课 常用范式

课前导读

这一课介绍了 4 个常用的编程范式: 泛型式、元编程、切面式和事件驱动式。与上节课类似,这些介绍也都是入门性的,目的是让读者认识到范式的多样性与差异性。



本课共分4节——

- 1. 泛型范式——抽象你的算法
- 2. 超级范式——提升语言的级别
- 3. 切面范式——多角度看问题
- 4. 事件驱动——有事我叫你,没事别烦我

3.1 泛型范式——抽象你的算法

以类行杂, 以一行万。

---《荀子·王制篇》

关键词: 编程范式: 泛型编程; STL; 算法; 容器; 迭代器

摘 要: 泛型式编程简谈



预 览

算法串联数据,如脊贯肉;数据实化算法,如肉附脊。

泛型编程是算法导向的,即以算法为起点和中心点,逐渐将其所涉及的概念内涵模糊化、外延扩大化,将其所涉及的运算抽象化、一般化,从而扩展算法的适用范围。

思想是鸡,结论是蛋。

•••

提问

泛型编程有哪些优点?

STL 有哪些要素?各自有什么作用?

泛型编程的泛化对象是什么?

泛型编程的核心思想是什么?

讲解

冒号重新开讲: "你们会不会经常遇到这样的情景:一遍又一遍地写着相似的 代码,有心将其归并,却因种种原因无法践行。"

逗号心有戚戚焉道: "是啊,有时明明两个函数的实现几乎是一模一样的,就

因为某些参数不匹配,无法合而为一。"

"有一种编程范式可以解决这个问题,它打破了不同数据类型之间的壁垒,让你的代码不再臃肿,这——就是泛型编程。"冒号的语调和说辞不免令人联想到电视上的减肥广告,"Generic Programming,简称GP,其基本思想是:将算法与其作用的数据结构分离,并将后者尽可能泛化,最大限度地实现算法重用。这种泛化是基于模板(template)的参数多态(parametric polymorphism),相比OOP基于继承(inheritance)的子类型多态(subtyping polymorphism),不仅普适性更强,而且效率也更高。这不能不说是一种异数——我们知道,普适性往往是以效率为代价的。如果一定要找出代价的话,那就是其用法稍微复杂一些,可读性稍微差一些。GP最著名的代表是C++中的STL(Standard Template Library),其后亦为Java、C#、D等语言所吸纳[1]。此外,一些函数式语言如Ocaml、Standard ML、Haskell等也支持GP。"

冒号写下如下两段代码——

"求两个数中的较大值是经常遇到的问题。"冒号解说着,"对于静态类型语言^[2]来说,若参数类型不同,即使函数体相同也不能合为一体。如果语言不支持重载(overload),还可能出现maxInt、maxLong、maxFloat、maxDouble之类的函数名,冗赘而丑陋。尽管在C中可用宏定义(macro definition)来实现,但无法保证类型安全,而C++模板则兼顾类型安全和代码重用,并且由于是在编译期间展开的,效率上也不损失。不止于此,C++支持运算符重载,除数值类型外,一切定义了'>'运算的数据类型均可调用max函数,真是一举N得,N趋向无穷大啊!"

冒号边说边比划,夸张的语气和手势逗得大家都笑了。

引号提出疑问: "Java 的一切对象都是 Object, 只要将所有参数都换成 Object 类型, 岂不也是一种泛化?"

冒号答道: "首先,基本类型如 int、float 等不是 Object 的子类,虽然 Java 新



插语

[1]但它们的实现机制却 大不相同: C++和 D 采用 类型模板 (template), Java 采用类型擦除 (type erasure), C#采用类型具 化 (reification), 相应的 表现也有显著差异。

[2]静态类型语言在编译 期间或运行之前施行类 型检查(type checking)。 后有详论。 增了自动装拆箱(autoboxing/unboxing)的功能,但要付出性能的代价。更重要的是,这将不可避免地需要类型的显式转换(explicit conversion 或 cast),无法在编译期间施行严格的类型检查,由此丧失了静态类型语言的优势,为 bug 大开方便之门。这也是 Java 最终引入泛型的原因,虽然有些姗姗来迟。类似地,C/C++中的通用指针 void *也有类型安全问题。"

句号发表他的看法: "泛型虽好,似乎只是某些局部才用到的技术,不具有前面几种范式的渗透性。"

冒号听罢不语, 返身在黑板上写下几道题——

- 1. 从一个整数数组中随机抽取十个数,对其中的素数求和。
- 2. 将一个无序整数集中所有的完全平方数换成其平方根。
- 3. 从学生成绩表中,列出门门都及格且平均分在70分以上的学生名单。
- 4. 在一个着色二元树中,将所有的红色结点涂成蓝色。
- 5. 将一个字符串从倒数第3个字符开始反向拷贝到另一个字符串中。
- 6. 每从标准输入读取一个非数字的字符X,于标准输出打印"X不是数字字符"。

句号暗忖,这有何难?不过是些常规题罢了。不料冒号的问题却出人意表:"请问它们之间有何共同之处?能否共享同一段代码?"

见众人缄默已久,冒号接着投影出一段代码——

"STL有3要素:算法(algorithm)、容器(container)和迭代器(iterator)。 算法是一系列切实有效的步骤;容器是数据的集合,可理解为抽象的数组;迭代器 是算法与容器之间的接口,可理解为抽象的指针或游标。"冒号讲述道,"算法串 联数据,如脊贯肉;数据实化算法,如肉附脊。只有抽象出表面的数据,算法的脊 梁才能显现。以上几题看似风马牛不相及,若运用泛型思维,便可发现它们的共性: 对指定集合中满足指定条件的元素进行指定处理。用模板语言,寥寥数行即勾勒完 毕。"

问号诧异道: "相比前面的 max 模板,这儿连元素的数据类型 T 都不见了?"

冒号回答: "元素被容器封装了。"

问号追问: "可连容器也看不到啊?"

冒号料有此问: "容器通过它的迭代器参与算法。"

句号豁然开朗:"通过模板,泛化了容器——可以是数组、列表、集合、映射、队列、栈、字符串,等等;泛化了元素——可以是任何数据类型;泛化了处理方法和限定条件——可以是任何函数。"

冒号提醒道:"补上两点:这里的处理方法和限定条件不限于函数,还可以是函子(functor)^[3]——自带状态的函数对象;另外,迭代器也被泛化了——可以从前往后移动,可以从后往前移动,可以来回移动,可以随机移动,可以按任意预先定义的规律移动。"

叹号由衷感叹: "果然强悍无比啊!"

逗号倒也心细: "最后一题中标准输入也算容器吗?"

"为什么不呢?只要一个对象配备了迭代器,它就可以作为容器来对待。I/O 流上就有现成的迭代器,当然你也可以自行定制。索性我们来看看这道题的解法吧。"冒号给出了第6题的实现代码——

```
#include <iostream>
#include "process.h" // 前述process所在的头文件

using namespace std;

// 判断字符是否为非数字字符
bool notDigit(char c)
{
    return (c < '0') || (c > '9');
}

// 打印非数字字符

void printNondigit(char c)
{
    cout << c << "不是数字字符" << endl;
}

int main()
{
    process(istream_iterator<char>(cin),istream_iterator<char>(),
        printNondigit, notDigit);

    return 0;
}
```



[3]又称 function object, 在 C++中指重载了函数 调用算符 (operator())的 类,在 Java 和 C#中可通 过 interface 来实现。 逗号打量了半天: "这里完全看不到 I/O 读取的过程,也看不到通常的迭代循环,简洁得难以置信。"

冒号补充道: "不光是代码简洁,它还让人摆脱了底层编码的细节,在更高、 更抽象的层次上进行编程设计。"

引号发觉: "开始谈起泛型编程时,您特别强调它对数据类型的抽象。现在看起来,它也能对函数进行抽象呢。"

"说得没错,条件是被抽象的函数或方法具有相同的签名(signature)或接口(interface)。不过别忘了,在 C 和 C++中的函数——准确地说是函数指针——也能作为数据类型。但不管怎样,这都表明泛型编程不仅能泛化概念,还能泛化行为。"冒号目光转向句号,"现在还有人认为泛型编程的渗透性不够强吗?"

句号腆然一笑。

"这些只是泛型编程的冰山一角。重要的是,我们不是在玩弄花哨的技巧,而是在用一种新的视角去审视问题。"冒号总结道,"泛型编程是算法导向(Algorithm-Oriented)的,即以算法为起点和中心点,逐渐将其所涉及的概念(如数据结构、类)内涵模糊化、外延扩大化,将其所涉及的运算(函数、方法、接口)抽象化、一般化,从而扩展算法的适用范围。这非常类似数学思维——当数学家证明完一个定理后,总会试图在保持核心思想的前提下,尽可能地放宽题设,增强结论,从而推广定理。外行人常以为数学定理最重要,其实数学思想才是数学的精髓。比如,举世皆知的哥德巴赫猜想和费尔马大定理,人们在攻克它们的过程中产生的新思想、新理论、新方法,已远远超过了定理本身的意义。数学家们甚至不愿这些猜想被过早地解决,怕扼杀了会下金蛋的鸡。在他们眼里,思想是鸡,结论是蛋。这也无怪乎STL会出自一位学数学的人之手了。[4]"

"我怎么觉得更像是出自一位菜场大妈之口呢?" 逗号打趣道, "不信你们听嘛,算法好比脊骨、数据好比猪肉、思想好比母鸡、结论好比鸡蛋。我没说错吧?"。

众人哑然失笑。

9 插语

[4]STL 的发明者 Alexander Stepanor 曾是莫斯科大学 数 学 系 的 学 生 (1967~1972)

•

总结

- 泛型编程能打破静态类型语言的数据类型之间的壁垒,在不牺牲效率并确保类型安全的情况下,最大限度地提高算法的普适性。
- ▶ STL 有 3 要素, 算法、容器和和迭代器。算法是一系列可行的步骤: 容

器是数据的集合,是抽象化的数组; 迭代器是算法与容器之间的接口, 是抽象化的指针。算法串联数据,数据实化算法。

- 泛型编程不仅能泛化算法中涉及的概念(数据类型),还能泛化行为(函数、方法、运算)。
- 》 泛型编程是算法导向的,以算法为中心,逐渐将其所涉及的概念内涵模糊化、外延扩大化,并将其所涉及的运算抽象化、一般化,从而提高算法的可重用性。

6 参考

[1] Bjarne Stroustrup. The C++ Programming Language, Special ed.. Reading, MA: Addison-Wesley, 2000. 507-516, 549-560

3.2 超级范式——提升语言的级别

智能繁衍: 机器人生产机器人。

---题记

关键词:编程范式;模板元编程;元编程;语言导向式编程;产生式编程

摘 要: 元编程简谈



预 览

元编程作为超级范式的一个体现是,它能提升语言的级别。

如果说 OOP 的关键在于构造对象的概念,那么 LOP 的关键在于构造语言的语法。

离开 IDE 就无法编写、编译或调试的程序员,如同卸盔下马后便失去战斗力的武士,是残缺和孱弱的。

既然有重复的代码,不能从语法上提炼,不妨退一步从文字上提炼。

元程序将程序作为数据来对待,能自我发现、自我赋权和自我升级,有着其他 程序所不具备的自觉性、自适应性和智能性,可以说是一种最高级的程序。



什么是元编程? 它与通常的编程有何不同?

元编程有何用处?它有哪些应用?

相比自编的元程序,用 IDE 自动生成的代码有什么缺陷?

语言导向式编程有何优点? 它与元编程有何关系?

元编程与产生式编程有何异同?

为什么说元程序是一种最高级的程序?

讲解

问号忽然想起一事,问道: "有一本名为《C++模版元编程》的书,既然提到了模板,想来也属于泛型编程吧?"

冒号答道: "模板元编程即 Template Metaprogramming,与泛型编程密切相 关但自成一派,隶属于另一种编程范式——元编程(Metaprogramming),简称 MP。此处的前缀'meta-'常译作'元',其实就是'超级'、'行而上'的意思。比如,元数据(Metadata)是关于数据的数据,元对象(Metaobject)是关于对象的对象,依此类推,元编程自然是关于程序的程序,或者说是编写、操纵程序的程序。"

叹号皱着眉: "听起来有点绕。"

冒号投影出如下代码——

```
C++ (元编程):

template <int N>
struct factorial
{
   enum { value = N * factorial<N - 1>::value };
};

template <> // 特化(specialization)
struct factorial<0> // 递归中止
```

```
{
    enum { value = 1 };
};

void main()
{
    // 以下等价于 cout << 120 << endl;
    cout << factorial<5>::value << endl;
}
```

"以上用模板元编程实现了阶乘运算。"冒号讲解道,"与前面3种核心范式的阶乘实现有着根本的不同:这里阶乘的值是在编译时而非运行时计算出来的。换句话说,这段代码以模板形式通过编译器生成了新的代码,并在编译期间获得执行。"

叹号大惑不解: "这又说明什么呢?"

冒号并不直接回答: "假设你需要批量处理用户文档, 其格式结构预先给定, 但既不像 CSV (逗号分隔) 那么简单, 也不像 XML 那么标准, 并且用户随时可能 改变格式标准, 请问如何设计这段程序?"

叹号略一思索,便回答:"3 大模块:阅读器读出输入文档,解析器按照格式标准去解析,处理器对解析结果进行处理。"

"显然关键在解析器,如果从头做起,那么问题至少有 4 个。"冒号扳着指头数,"第一,费时写解析器代码;第二,费时调试解析器代码;第三,如果用户更改格式标准,你得重复做上两件事;第四,如果这段程序是大型程序的一部分,任何改动都可能意味着软件的重新编译、连接、测试、打包、部署,等等。如果因为你的缘故公司不得不频频发布补丁包的话,你的饭碗恐怕是朝不保夕了。"

还是句号机灵:"既然谈到了元编程,一定是利用元编程,根据不同的格式标准自动生成相应的解析器代码。不过——此法虽一劳永逸,但难度似乎不小啊。"

"思路对头!"冒号赞许道,"大家听说过 Lex 和 Yacc 吗?它们能根据格式标准生成相应的解析器代码。更妙的是,格式标准不限于静态数据,甚至可以含有动态指令!这意味着用户不仅能定义业务数据格式,还能定义业务流程。"

"这敢情好!"叹号兴奋地说。

"如果知道 Lex 和 Yacc 本来就是编写编译器和解释器的工具,你就不会惊讶于它们的强大了。顺带说一句,编译器本身就是元编程的典型范例——把高级语言转化为汇编语言或机器语言的程序,不就是能写程序的程序吗?"冒号引申开来,

"更进一步地,我们可以定义自己的领域特定语言 DSL,更加灵活方便地处理客户逻辑。"

逗号有点糊涂了:"领域特定语言?就是前两堂课提到的非通用编程语言吧? 怎么和元编程也扯上关系了?"

"不是扯上关系,而是它们之间本来就有着千丝万缕的联系。"冒号纠正着,"相比第 3 代的通用编程语言,领域特定语言由于其在应用范围上和语法上的限制而显得简单、针对性强,有时被成为'小语言'(little language),也是一种特高级语言(very high-level programming language,简称 VHLL),属于第 4 代编程语言。"

冒号说到此处,逗号猛地一拍脑门: "哦,我明白了。第 4 代语言最终须要编译为机器语言,而编译器就是元编程的应用。"

"你只说对了一半。"冒号不疾不缓地说,"DSL 一般不会一步到位地编译为第 1 代的机器语言或第 2 代的汇编语言,而是通过现成的编译器生成器(compiler-compiler 或 compiler generator)首先转化为第 3 代的高级语言。这样不仅大大降低了难度,也方便了程序的调试。刚才提到的 Yacc(Yet Another Compiler Compiler)便是这样的工具,能为解析器(parser)产生 C 程序,多用于 Unix 下的编程。更现代的工具如 ANTLR (ANother Tool for Language Recognition),能生成 C、C++、Java、C#、Python 等多种语言的源程序。"

引号立刻联想到:"我记得框架 Hiberate 的必备库中就含有 antlr.jar 文件,与这个 ANTLR 有关吗?"

"正是!"冒号很满意学员完美的配合,"Hiberate 中的 HQL (Hibernate Query Language)是典型的 DSL,须要通过 ANTLR 来解析。你们可以验证一下,在 Hibernate 的 API 中有 org.hibernate.hql.antlr 的 package,但在其发布的源代码中相应的目录下却看不到一个 Java 源文件。却是为何?盖因此 package 中所有的源代码都是在 ant build 中自动生成的,这些非人工编辑的文件是不会放在版本控制中的。"

众人茅塞顿开。

句号想通了一个逻辑: "元编程作为超级范式的一个体现是,它能**提升语言的级别**。比如,有了编译器的存在,汇编语言升级为第 3 代高级语言;同样借助 Yacc、ANTLR 之类的元编程工具,第 3 代语言可以升级为第 4 代的 DSL 语言。"

冒号并未就此止步: "将这一模式发挥到极致,便是更加激进的语言导向式编

程^[1](Language-Oriented Programming,简称LOP)。这种编程范式的思路是:在建立一套DSL体系之后,直接用它们来编写软件,尽量不用通用语言。"



叹号莫明其妙:"想法近乎疯狂啊!放着好端端的通用语言不用,先造一套专用语言,这么做划算吗?"

[1]Martin Ward 最早提出 此范式,见参考文献[1]。

"如果一个大型系统涉及的领域十分专业,包含的业务逻辑十分复杂,为其定制 DSL 或许会磨刀不误砍柴工。我们通过图 3-1 和图 3-2 比较一下这种范式与主流编程范式的不同之处。"冒号映出新的投影——



图 3-1 诵用语言编程

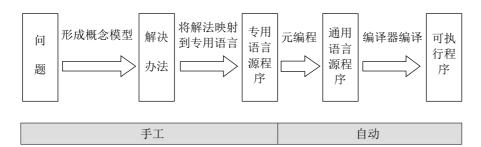


图 3-2 专用语言编程

"由于 DSL 比通用语言更简单、更抽象、更专业、更接近自然语言和声明式语言,开发效率显著提高,因此图中手工部分的时间相应减少。此外尤为关键的是,这种方式填补了专业程序员与业务分析员之间的鸿沟。要求一个非专业编程的业务分析员用 DSL 来开发固是勉为其难,但要做到读懂代码并审查其中的业务逻辑则已非难事。"冒号细解个中要点,"如果说 OOP 的关键在于构造对象的概念,那么 LOP 的关键在于构造语言的语法。有人认为 LOP 是继 OOP 之后的下一个重要的编程范式,我们不妨拭目以待。"

句号整理了一下头绪: "能不能这么说:如果处理一些复杂、非标准格式的文档,可以考虑用元编程;如果整个业务逻辑复杂多变,可以考虑利用现有的 DSL

或创造新的 DSL 来处理业务,即所谓的语言导向式编程。"

"总结得不错,不过特定格式的文档有了专门的解析器后,这种文档格式标准 就可视为一种语言了,不是吗?这本质上就是 DSL 啊。"冒号出语点化。

句号顿时醒悟: "是啊,就像 XML、HTML 一样,能被程序认识的格式可不 就是一种计算机语言嘛。"

冒号将话题延伸: "我们的想象力可以再狂野些,在文本 DSL 的基础上裹以 图形界面,从而引进图形语言。如果再将部分业务逻辑开放给用户定制,那么你的 客户会欣喜地发现,他们的经理只要点点鼠标就可以改变整个业务流程了,而这一 切不仅不需要软件开发方或第3方的参与,连本公司的技术人员也免了。这时候倒 是你的老板发愁了:你的设计太过完美,客户的后续开发费怕是赚不到啰。"

众人一乐。

问号继续发问: "还有其他元编程的应用吗?"

冒号随口举了几例:"元编程的例子比比皆是:许多IDE如Visual Studio、Delphi、 Eclipse 等均能通过向导、拖放控件等方式自动生成源码; UML 建模工具将类图转 换为代码; Servlet 引擎将 JSP 转换为 Java 代码; 包括 Spring、Hibernate、XDoclet 在内的许多框架和工具都能从配置文件、annotation/attribute 等中产生代码。"

引号仍不知足:"这些应用虽然典型,但都是些开发工具、框架引擎之类的基 础软件,有没有平时编程就能用到的例子?"

"当然有!"冒号坚定地答复,"有时程序中会出现大量的重复代码,却囿于 语法上的限制无法进一步抽象化和模块化。如果采用手工编写或单纯拷贝的方法, 既费时又易错,显为下策。有时可借助 IDE 内置的代码生成功能,但一方面局限 性很大,另一方面无法自动化和版本化。"

问号插问: "什么叫版本化?"

冒号解释: "理想情况下,一个程序员对程序的贡献都应该保存在版本控制系 统(version control system)中,以便跟踪、比较、改进、借鉴和再生成。在 IDE 下 自动生成的代码本身可以被记录,但产生代码时的行为却不能被记录,几次简单的 鼠标动作就能产生较大的代码差别, 使得版本比较的意义大打折扣。顺便说一句, 离开 IDE 就无法编写、编译或调试的程序员,如同卸盔下马后便失去战斗力的武 十,是残缺和孱弱的。"

问号有些明白了: "这是因为鼠标行为本身在代码中是没有痕迹的。"

"不仅是鼠标行为,有些需要键盘交互的行为也是没有痕迹的。比如在命令行下用debugger来调试的行为无法被记录,也难以重复和自动化,只能作为权宜之策。相比之下,日志(logging)和单元测试(unit test)具有明显的优势^[2]。"冒号答完,立马重返主题,"回到上面的问题,既然有重复的代码,不能从语法上提炼,不妨退一步从文字上提炼。我们可以利用AWK、Perl之类的擅长文字处理的脚本语言,当然也可以用Java、C等非脚本语言,再辅以XSLT之类的模板语言,自动生成重复代码。这样不仅灵活性强,而且生成代码的代码——也就是元程序代码可以被重用,元程序的数据来源也能版本化。"

句号深得要领:"就像 Hibernate 中的 antlr 包一样,真正的源码反而不在版本控制中了。一方面没有保存的必要——可以自动生成;另一方面没有比较的必要——元程序的数据来源的变化比实际源码的变化更简明、更直观。"

冒号继续推进: "另外,有时程序的结构需要动态改变,而 C++、Java、C#等静态语言是不允许动态变更类的成员或实现代码的,利用元编程便可突破这种限制。"

逗号恍然大悟: "原来元编程就是编写能自动生成源代码的程序。"

"也不尽然。"冒号马上修正道,"自动生成源代码的编程也属于另一种编程范式——产生式编程(Generative Programming)^[3]的范畴。区别在于后者更看重代码的生成,而元编程看重的是生成代码的可执行性。另外,除了在编译期间生成源代码的静态元编程,还有能在运行期间修改程序的动态元编程。从低级的汇编语言到一些高级的动态语言如Perl、Python、Ruby、JavaScript、Lisp、Prolog等均支持此类功能。比如,许多脚本语言都提供eval函数,可以在运行时将字符串作为表达式来运算^[4]。"

问号突然问道: "编写病毒算不算元编程?"

"编写一个只是删除或感染文件的病毒,不必用到元编程。但如果要开发一个能自我变异的智能病毒,那就需要元编程了。不过你要是把元编程用在这方面,可别说是我教的。"冒号开了个玩笑。

引号自言自语: "程序的程序,就是程序的平方。"

"也可以是程序的立方,4次方……理论上是无限次方。在传统的编程中,运算是动态的,但程序本身是静态的,在元编程中,二者都是动态的。元程序将程序



插语

[2]虽然调试与日志和测试不是一码事,但合理的日志和单元测试能大量减少调试工作。

[3]也译作"生成式编程",属于自动编程 (Automatic Programming) 范畴。

[4]考虑到 eval 过于广泛和强大,有些动态语言还提供其他更明确和更安全的元编程机制,如 JavaScript 可用字符串来构建 Function, Ruby 更是提供了define_method、instance_eval、class_eval和 module_eval等诸多元编程方法。

作为数据来对待,能自我发现、自我赋权和自我升级,有着其他程序所不具备的自 觉性、自适应性和智能性,可以说是一种最高级的程序。它要求编程者超越常规的 编程思维,在一种崭新的高度上理解编程。想象一下吧!"冒号激情勃发,"如果 有一天机器人能自我学习、自我完善,甚至能生产新的机器人,实现'智能繁衍', 是不是很美妙?"

"我怎么觉得特恐怖呢?岂止是程序员,所有地球人的饭碗都会被它们砸光 了。"叹号此言一出,众皆忍俊不禁。

总结

- 元编程是编写、操纵程序的程序。在传统的编程中,运算是动态的,但 程序本身是静态的;在元编程中,二者都是动态的。
- 元编程能减少手工编程、突破原语言的语法限制、提升语言的抽象级别 与灵活性,从而提高程序员的生产效率。
- 元编程有诸多应用:许多开发工具、框架引擎之类的基础软件都有自动 生成源代码的功能; 创造 DSL 以便更高效地处理专门领域的业务; 自动 生成重复代码: 动态改变程序的语句、函数, 类, 等等。
- IDE 下自动生成的代码通常局限性大且可读性差, 小操作可能造成的源 码上的大差异, 削弱了版本控制的意义。用自编的无需人机交互的元程 序来生成代码,只须将元程序的数据来源版本化,简明而直观。同时由 于元程序可以随时修改,因此局限性小,更加灵活。
- ▶ 语言导向式编程(LOP)通过创建一套专用语言 DSL 来编写程序。相比 通用语言, DSL 更简单、更抽象、更专业、更接近自然语言和声明式语 言、开发效率更高,同时有助于专业程序员与业务分析员之间的合作。
- 语言导向式编程一般通过元编程将专用语言转化为通用语言。
- 产生式编程与静态元编程都能自动生成源代码。产生式编程强调代码的 生成, 元编程强调生成代码的可执行性。此外, 动态元编程并不生成源 代码, 但能在运行期间修改程序。
- ▶ 元程序将程序作为数据来对待,有着其他程序所不具备的自觉性、自适 应性和智能性,可以说是一种最高级的程序。

③ 参考

- [1] Martin Ward. Language Oriented Programming. http://www.cse.dmu.ac.uk/~mward/martin/papers/middle-out-t.pdf
- [2] Sergey Dmitriev. Language Oriented Programming: The Next Programming Paradigm.

http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf

[3] Wikipedia. Metaprogramming. http://en.wikipedia.org/wiki/Metaprogramming

3.3 切面范式——多角度看问题

横看成岭侧成峰。

——《苏轼·题西林壁》

关键词:编程范式; SoC; DRY; AOP; Aspect; join point; pointcut;

advice; OOP

摘 要: 切面式编程简谈



预览

从宏观角度看,太阳底下没有新鲜事——AOP 无非是 SoC 原理和 DRY 原则的一种应用。

从微观角度看,太阳每天都是新的——AOP 虽自 OOP 的土壤中长出,却脱离藩篱自成一体。

抽象是前提,分解是方式,模块化是结果。

在常人眼中复杂的牛体, 庖丁经过抽象,已目无全牛,及至提刀分解,自是游 刃有余。待牛如土委地,模块化即成。

两条(抽象与分解的原则):单一化,正交化。每个模块职责明确专一,模块 之间相互独立,即高内聚低耦合。

对程序员来说,英语也是一门计算机语言,而且是必修的语言。

OOP 只能沿着继承树的纵向方向重用,而 AOP 则弥补了 OOP 的不足,可以在横向方向重用。

如果一个程序是一个管道系统,AOP 就是在管道上钻一些孔,在每个孔中注入新的代码流。

00

提问

什么是 SoC 和 DRY?

如何有效地避免紊乱、松散、重复的代码?

抽象与分解的原则是什么?

什么是横切关注点?

接入点与切入点有何区别?

什么是编织? 有哪些不同的编织方法?

实施 AOP 有哪些步骤?

为什么说 AOP 是 OOP 的一种补充?

为什么提倡尽可能地阅读原文的书籍和资料?

讲解

课间休息刚一结束,引号便重开话题: "OOP 方兴未艾,AOP 又开始崭露头角。AOP 算是 OOP 的一种补充、一种分支,还是一种超越?"

叹号故作捶胸顿足状: "OOP还没有完全吃透,又来了个什么 AOP。"

"不同的人对新生事物采取不同的态度。"冒号王顾左右而言他,"追星族倾向于盲目追捧,唯恐落伍,他们信奉新潮的、流行的就是好的;守旧派倾向于本能抗拒,回避求新,他们认为经典的、传统的才是好的。"

引号和叹号互视一眼,不情愿地戴上了老冒派发的帽子。

冒号续道: "从宏观角度看,太阳底下没有新鲜事——AOP 无非是 SoC 原理和 DRY 原则的一种应用;从微观角度看,太阳每天都是新的——AOP 虽自 OOP的土壤中长出,却脱离藩篱自成一体,并且嫁接到非 OOP 的领地,不仅在纯过程

式语言、函数式语言,甚至逻辑式语言中得到发展,而且本身也具备了一定的声明 式语言特征,成为一种新的软件模块化方式。"

问号举手: "什么是 SoC 和 DRY?"

引号代答: "SoC 就是 Separation of Concerns,即关注点分离; DRY 是 Don't Repeat Yourself,即尽量减少重复代码。"

"答案正确,加十分!"冒号戏赞道,"不良代码通常有两种病征:一是结构混乱,或聚至纠缠打结、或散至七零八落;二是代码重复,叠床架屋、臃肿不堪。治疗此类病症一个有效的方法是抽象与分解:从问题中抽象出一些关注点,再以此为基础进行分解。分解后的子问题主题鲜明且独立完备,既不会牵一发而动全身,也不会四分五裂、支离破碎。同时具有相同特征的部分可以像代数中的公因子一样提取出来,提高了重用性,减少了重复性。"

句号醒悟道:"这不就是模块化吗?"

"准确地说,抽象是前提,分解是方式,模块化是结果。"冒号很讲究精确, "大家记得庖丁解牛的故事吧?在常人眼中复杂的牛体,庖丁经过抽象,已目无全 牛,及至提刀分解,自是游刃有余。待牛如土委地,模块化即成。"

句号举一反三:"前面提到的编程范式的基本思想大多不也如此?将程序分别抽象分解为过程、函数、断言、对象和进程,就依次成为过程式、函数式、逻辑式、对象式和并发式。至于泛型式——"

句号讲不下去了。

"泛型式虽未引入新类型的模块,其核心也是抽象出算法后与数据分解。"冒号为其解围,"以此类推,切面式的 AOP 将程序抽象分解为切面。"

问号提问:"抽象与分解的原则是什么?"

冒号作了个V字: "两条:单一化,正交化。每个模块职责明确专一,模块之间相互独立,即高内聚低耦合(high cohesion & low coupling)^[1]。此原则相当普适,是分析复杂事物的一种基本方法,在数学和物理中应用得尤为广泛,如质因式分解、正交分解、谱分解,等等。"

逗号调皮地抬杠: "为什么称为正交化呢? 斜交化不行吗?"

冒号呵呵一笑: "在数学中互为正交的两个向量在彼此方向上投影为零,意味着彼此独立、互不影响,斜交可不行。"



插语

[1]耦合 (coupling) 用来 衡量模块之间的依赖程 度, 内聚 (cohesion) 用 来衡量模块内在的关联 强度。它们常用来作为软 件质量的评判标准,耦合 度宜低, 内聚度宜高。 逗号吐了吐舌头。

"诚如前述,AOP 以切面为模块。"冒号返回主题,"切面 Aspect 常直译为'方面',但它描述的是横切关注点(cross-cutting concerns),故'切面'更准确生动,而'方面'则失之空泛呆板。何谓横切关注点?顾名思义,乃是与程序的纵向主流执行方向横向正交的关注焦点。不妨回顾一下,无论是过程式的函数,还是对象式的方法,都包含了完整的执行代码。但有些代码横跨多个模块,以片断的形式散落在各处,虽具有相似的逻辑,却无法用传统的方式提炼成模块,难以实现SoC 与 DRY。典型的例子如:在调用某些对象的方法、读写某些对象的域、抛出某些异常等前后需要用到统一的业务逻辑,诸如日志输出、代码跟踪、性能监控、异常处理、安全检查、事务管理,等等。为解决此类问题,AOP 应运而生。它将每类横切关注点封装到单独的 Aspect 模块中,将程序中的一些执行点与相应的代码绑定起来。单个的执行点称为接入点(join point)。例如,调用某个对象的方法前后;符合预先指定条件的接入点集合称为切入点(pointcut)。再如,所有以 set 为命名开头的方法;每段绑定的代码称为一个建议(advice)。"

问号有点疑问:"接入点与切入点有何区别?"

冒号释疑:"望文生义,接入处是点,切入处是面,面由点组成。advice 定义于切入点上,执行于接入点处。换言之,共享一段附加代码的接入点组成了一个切入点。切入点一般用条件表达式来描述,不仅有广泛性,还有预见性——以后新增的代码如果含有满足切入点条件的接入点,advice 中的代码便自动附着其上。这是AOP的成力所在,但有时也是麻烦所在。"

引号很较真: "好像一些书上把 join point 译作连接点,把 advice 译作通知。"

"误导,完全是误导!"冒号有些痛心疾首,"何谓 join point?是 advice 中额外代码接入之处,join 显为'参加'、'加入'之意。如果说翻作'连接'只是因缺乏动感和方向性而不够贴切的话,将 advice 译作'通知'则近乎荒谬了。advice是在原有程序流程中加入的额外流程,可理解为建议采取的措施,而'通知'强调的是一种信息,难道是程序运行到 join point 的信息?抑或采取某种行动的信息?简直不知所云。"

顿了一会,冒号仍意犹未尽: "英文好的技术不好,技术好的英文不好,两者都好的不屑去翻译,导致市面上的译书虽汗牛充栋,然佳作寥寥。这里奉劝各位,如果真想成为优秀的程序员,一定要尽可能地读原文的书籍、文章和文档。事实上,凡是科学和艺术方面的专业人员,要想专业水平上一层台阶,都应读该专业权威经

典原文。要知道,语言之间的天堑原本难以弥合,译者的专业水准、语言功底和严 谨程度更是参差不齐。纵使万事俱备,一年半载后的译书便如隔夜的饭菜,虽刚出 炉,已然不新鲜了。"

逗号抱怨: "英文虽读得懂,但太慢、太费劲了。"

"多读,读多了就习惯了。"冒号鼓励着,"对程序员来说,英语也是一门计算机语言,而且是必修的语言。"

课堂上有这么一个规律,越是题外的话大家讨论得越欢。下面果然开始嘈杂起来,冒号也乐得乘机歇歇嘴。

等大伙渐渐把视线重新聚焦到讲台上,冒号才继续讲课: "从软件重用的角度看,可以这么理解 AOP 与 OOP 的关系: OOP 只能沿着继承树的纵向方向重用,而 AOP 则弥补了 OOP 的不足,可以在横向方向重用。这算是回答了引号开始提出的问题: AOP 不是 OOP 的分支,也不能说是超越了 OOP,而是 OOP 的一种补充——尽管 AOP 并不局限于 OOP 语言。"

问号的求知欲很强: "AOP 实现的机理是什么?"

冒号回答: "如果一个程序是一个管道系统,AOP 就是在管道上钻一些孔,在每个孔中注入新的代码流。因此 AOP 实现的关键是将 advice 的代码嵌入到主体程序之中,术语称编织(weaving)。这是很自然的——将问题分解之后再合成,问题才得以还原。编织可分两种: 一种是静态编织,通过修改源码或字节码(bytecode)在编译期(compile-time)、后编译期(post-compile)或加载期(load-time)嵌入代码——请注意,这里涉及刚才提到的元编程和产生式编程; 另一种是动态编织,通过代理(proxy)等技术在运行期(run-time)实现嵌入。具体的工具包括一些扩展性语言如 AspectJ、AspectC++、Aspect#等和一些框架如 AspectWerkz、Spring、Jboss AOP等。"

叹号搔着头: "听起来怪复杂的。"

引号倒不在乎:"这些机理是 AOP 的实现者需要操心的,使用者只需关心 AOP 是否好用,性能如何,等等。"

"为了让你们有更直观的印象,我们借用光学原理来类比。"冒号用幻灯片展示(如图 3-3 所示)——

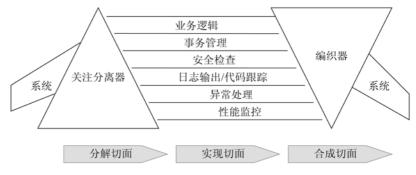


图 3-3 切面式编程

"众所周知,白光经过三棱镜的折射而分解为七色光,是谓光的色散。再经过一个倒置的三棱镜,七色光又重新会聚为白光。"冒号简述中学的物理知识,"如果把一个复杂的系统看作复合色的白光,经过第1个三棱镜——关注分离器,系统被分解为不同的切面,如同不同的单色的彩光。这些切面经过第2个三棱镜——编织器,再度合成为原系统。"

叹号脸上的迷惘之色渐去: "这下清楚多了。"

句号积极发言: "从中看出, AOP 的实施分 3 步: 切面分解、切面实现和切面合成。其中第 1 步是在设计者的头脑中进行的, 第 3 步是通过 AOP 的工具实现的,真正须要程序员编码的部分在第 2 步,即分别实现各切面的 advice。"

冒号赶紧补漏: "你好像忽略了切面的另一要素 pointcut,如果程序员不指明 advice 挂靠的切入点,系统如何知道该何时何处调用他编写的执行代码呢?"

句号的嘴张成 O 状: "是哦,我怎么把这茬给忘了?"

在 AOP 的议题结束前,冒号不忘指出: "与 OOP 一样,AOP 在带来便利的同时,也增加了一定的复杂度和性能损耗。它们更适用于大中型程序,用在小型程序中则不啻牛刀杀鸡。"

🕒 总结

- SoC 是 Separation of concerns 的缩写,指应将关注点分离; DRY 是 Don't Repeat Yourself 的缩写,指应尽量减少重复代码。
- ▶ 抽象与分解是治愈代码紊乱、松散、重复的良方。
- ▶ 抽象与分解的原则是单一化和正交化,以保障软件系统符合"高内聚、

低耦合"的要求。

- ▶ 横切关注点指与程序的纵向主流执行方向横向正交的关注焦点。
- ▶ 接入点是附加行为——建议(advice)的执行点,切入点(pointcut)是指定的接入点(join point)集合,这些接入点共享一段插入代码。切入点与建议组成了切面(aspect),是模块化的横切关注点。
- 编织是将附加的切面逻辑嵌入到主体应用程序之中的过程。编织分静态 编织和动态编织两种。静态编织在编译期、后编译期或加载期嵌入代码, 动态编织则在运行期嵌入。
- ▶ AOP的实施分 3 步: 切面分解、切面实现和切面合成。
- ▶ OOP 只能沿继承树的纵向方向重用, AOP 可以沿横向方向重用。
- ▶ 语言之间的天然差别、译者的专业水准、语言功底和严谨程度及时间上的滞后决定了阅读原文书籍和资料的必要性。

🥯 参考

- [1] Wikipedia. Aspect-oriented programming. http://en.wikipedia.org/wiki/Aspect-oriented_programming
- [2] Ramnivas Laddad . I want my AOP!. http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html

3.4 事件驱动——有事我叫你,没事别烦我

劳心者治人,劳力者治于人。

——《孟子·滕文公上》

关键词:编程范式;事件驱动式编程;回调函数;框架;控制反转; 观察者模式

摘 要: 事件驱动式编程简谈



预 览

它们(同步回调和异步回调)都使调用者不再依赖被调者,将二者从代码上解 耦,异步调用更将二者从时间上解耦。

它们(控制反转、依赖反转和依赖注射)的主题是控制与依赖,目的是解耦, 方法是反转,而实现这一切的关键是抽象接口。

"回调"强调的是行为方式——低层反调高层,而"抽象接口"强调的是实现方式——正是由于接口具有抽象性,低层才能在调用它时无须虑及高层的具体细节,从而实现控制反转。

控制反转导致了事件驱动式编程的被动性。

事件驱动式还具有异步性的特征,这是由事件的不可预测性与随机性决定的。

独立是异步的前提, 耗时是异步的理由。

发行/订阅模式正是观察者模式的别名,一方面可看作简化或退化的事件驱动式,另一方面可看作事件驱动式的核心思想。



提问

什么是事件? 有哪些不同类型的事件?

什么是回调函数? 什么是异步回调? 它们有什么用处?

控制反转的目的是什么? 它是如何实现的? 它在框架设计中起什么作用?

控制反转、依赖反转原则和依赖注射的共同点是什么?

事件驱动式编程有哪些关键步骤?

异步过程特点和作用是什么?

事件驱动式编程最重要的特征是什么?它们是如何实现的?

事件驱动式与观察者模式、MVC 架构有何关系?

讲解

逗号渐觉睡虫上脑, 开始闭目点头。正神游之际, 忽觉腰间一阵酥麻。惺眼微

睁,原是被引号的胳膊肘给捅的,顿时警醒。抬头见讲台上的老冒正目光灼灼地盯着自己,不禁脸颊微烫,嗫嚅道: "不好意思,昨晚睡得太晚了。"

冒号却不以为意:"正愁找不到新话题呢,你倒启发了我。话说课堂上睡觉大抵有3种方式——"

话音未落,有人已笑不自禁。

"第一种是警觉式:想睡可又担心被老师发现,不时睁眼查看周围的变化。同时双耳保持警戒,一有异动立刻挺直身板。"冒号有板有眼地形容,"第2种是宽心式:俯桌酣睡,如处无人之境。境界至高者或可雷打不动,或可鼾声如雷。"

"总之是很雷人。"叹号的网络新语再度引发笑声。

冒号继续分析: "第3种是托付式:请人放哨,非急勿扰。遂再无顾忌,大可封目垂耳,安心入眠。请问你们乐意采用哪种方式?"

"第1种方式睡不踏实,不得已而为之。敢用第2种方式的人多半没心没肺,估计IT人都达不到那种境界。只要有同伴在身旁,我想大家都会选第3种方式的。"句号的回答获得一致认同。

冒号续问: "好,抛开第2种方式不谈,为什么第3种要比第1种优越呢?"

句号回答:"犯困者既要打盹又要警戒,必然苦不堪言。如果把警戒的任务委 托同伴,两人分工合作,自然愉快得多。"

冒号再问: "他们是如何合作的呢?"

"放哨者一旦发现有情况,立即通知犯困者采取行动——睁眼坐直,作认真听讲状。"句号说得是绘声绘色。

除了两位当事人略显尴尬外,其他人均乐不可支。

眼见时机成熟,冒号不再兜圈: "采用警觉式者主动去轮询(polling),行为取决于自身的观察判断,是流程驱动的,符合常规的流程驱动式编程(Flow-Driven Programming)的模式。采用托付式者被动等通知(notification),行为取决于外来的突发事件,是事件驱动的,符合事件驱动式编程(Event-Driven Programming,简称 EDP)的模式。下面我们就来说说这种编程范式。"

逗号瓮声瓮气道:"没想到打瞌睡打出了个范式。"

冒号瞥了他一眼,继续说下去:"为完成一样事,既可以采用流程驱动式,也可以采用事件驱动式。这样的例子在生活中可谓俯拾即是,刚才逗号同学为大家现

场示范了一个, 谁还能举出其他范例?"

叹号抢先举例:"与客户打交道,推销员主动打电话或登门拜访,他的工作是 流程驱动的:接线员坐等电话,他的工作是事件驱动的。"

问号也说: "同样是交通工具,公共汽车主要是流程驱动的,它的路线已预先设定;出租车主要是事件驱动的,它的路线基本上由随机搭载的乘客所决定。"

引号以个人经验作例: "购买喜爱的杂志可以选择频繁光顾报刊亭,也可以选择一次性订阅。浏览关注的新闻网站或博客,可以直接访问站点,也可以订阅相应的 RSS。主动检查所关心的内容是否更新是流程驱动的,用订阅的方式是事件驱动的。"

句号回到本行:"Windows下的许多工作既可以在 DOS 下用批处理程序实现,也可以在图形界面下完成。前者不需人工干预,显然是流程驱动的;后者毫无疑问是事件驱动的。"

"看来你们对这种范式很熟悉嘛。不过,它原理虽简单,威力却无穷。看似一招,实则暗藏百式,甚可幻化千招。个中精妙之处,断非一时可以尽述。"冒号不知不觉中又走进了武侠的世界。

众人听了, 暗疑老冒有些言过其实。

冒号正式入题: "首当其冲的问题是:何谓事件?通俗地说,它是已经发生的某种令人关注的事情。在软件中,它一般表现为一个程序的某些信息状态上的变化。基于事件驱动的系统一般提供两类的内建事件(built-in event):一类是底层事件(low-level event)或称原生事件(native event),在用户图形界面(GUI)系统中这类事件直接由鼠标、键盘等硬件设备触发;一类是语义事件(semantic event),一般代表用户的行为逻辑,是若干底层事件的组合。比如鼠标拖放(drag-and-drop)多表示移动被拖放的对象,由鼠标按下、鼠标移动和鼠标释放三个底层事件组成。"

问号推想: "编程人员应该还能创造新的事件类型吧?"

"那是当然。"冒号点点头,"还有一类用户自定义事件(user-defined event)。它们可以是在原有的内建事件的基础上进行的包装,也可以是纯粹的虚拟事件(virtual event)。除此之外,编程者不但能定义事件,还能产生事件。虽然大部分事件是由外界激发的自然事件(natural event),但有时程序员需要主动激发一些事件,比如模拟用户鼠标点击或键盘输入等,这类事件被称为合成事件(synthetic event)^[1]。这些都进一步丰富完善了事件体系和事件机制,使得事件驱动式编程更



[1]许多基于事件驱动的 系统都提供了 createEvent之类的API、 授权编程者自行产生事 件。 具渗透性。"

叹号嘟哝了一句: "看来这里边还有点名堂。"

"名堂多着呢!"冒号回应,"事件固然是事件驱动式编程的核心概念,但一个编程范式的独特之处绝不仅仅是一些概念,更重要的是建立于这些概念之上的思维模式。为了了解这种范式与众不同的特点,我们先看看如何利用 win32 的 API 在 windows 下创建一个简单的窗口——"

```
/** 一个win32窗口程序 */
...WinMain(...) // windows应用程序的主函数
   // 第一步——注册窗口类别
   windowClass.lpfnWndProc = WndProc; // 指定该类窗口的回调函数
   // 指定该类窗口的名字
   windowClass.lpszClassName = windowClassName;
   RegisterClassEx(&windowClass);
   //第二步——创建一个上述类别的窗口
   CreateWindowEx(..., windowClassName, ...);
   ...;
   // 第三步——消息循环
   while (GetMessage(&msg, NULL, 0, 0) > 0) // 获取消息
      TranslateMessage(&msg); // 翻译键盘消息
      DispatchMessage(&msg); // 分派消息
}
// 第四步---窗口过程(处理消息)
...WndProc(..., msg,...)
   switch (msg)
      case WM_SIZE: ...; // 用户改变窗口尺寸
      case WM_MOVE: ...;
                       // 用户移动窗口
                      // 用户关闭窗口
      case WM_CLOSE: ...;
      ...;
   }
}
```

"没有选用 Java、Visual C++、C#、VB 或 Delphi 来实现窗口,是因为它们高度的封装和强大的 IDE 掩盖了部分事件机制。如果你们对 win32 API 不太熟悉,没有关系。为了减少语言和 API 上的障碍,同时突出重点,这里最大限度地省略了次要的过程和参数等,仅保留脉络主干。"冒号解释,"从中可看出,创建一个能响应用户操作的 win32 窗口共分 4 步:注册窗口类别、创建窗口、消息循环和窗口过

程。"

问号对概念很敏感: "消息与事件是一回事吗?"

"严格说来它们不是一回事,但如果你不想深究,不加区分也无大碍。概略地说,消息是Windows内部最基本的通信方式,事件需要通过消息来传递,是消息的主要来源。每当用户触发一个事件,如移动鼠标或敲击键盘,系统都会将其转化为消息并放入相应程序的消息队列(message queue)中^[2]。"冒号解答着,"明白了这一点,上面的代码就不难理解了——在消息循环中,程序通过GetMessage不断地从消息队列中获取消息,经过TranslateMessage预处理后再通过DispatchMessage将消息送交窗口过程WndProc处理。"

逗号琢磨了一会,不解地问:"窗口过程应该是在分派消息时被调用的,但我怎么想不出 DispatchMessage 是如何联系到 WndProc 的?"

冒号为其解惑: "DispatchMessage的消息参数含有事发窗口的句柄(handle),从而可以得到窗口过程WndProc^[3]。至于窗口与窗口过程之间是如何建立联系的,回看前面两步就一目了然了: 当初在创建窗口时指明了窗口类别名windowClassName,而窗口类别windowClass又绑定了窗口过程。"

叹号有点纳闷: "干嘛要绕这么大的弯子,直接调用 WndProc 不就得了?"

"对于这个简单的程序来说,的确区别不大。但假如再增添其他菜单、按钮、 文本框之类的控件,每个控件都可绑定自己的窗口过程,那么到底该调用哪个才对 呢?"冒号反问。

叹号虽有所悟,但仍有心结: "总觉得窗口过程的用法有些怪怪的。"

冒号一敲桌案: "没错!怪就怪在编程者自己写了一个应用层的函数,却不直接调用它,而是通过库函数间接调用。这类函数有个专用名称:回调函数(callback)。"

引号忍不住插话: "回调函数我知道,在 C 和 C++中就是函数指针嘛。"

"确切地说,函数指针是 C 和 C++用来实现 callback 的一种方式。此外,抽象类(abstract class)、接口(interface)、C++中的泛型函子(generic functor)和 C#中的委托(delegate)都可实现 callback。我们先图解一下回调机制。"冒号调出一张图(如图 3-4 所示)——



[2]更准确地说,Windows 先把所有的硬件事件存 入系统消息队列(system message queue),然后再 放入应用程序消息队列 (application message queue)。

[3]比如可以这样从 msg 中 得 到 窗 口 过 程: (WNDPROC)GetWindow Long(msg.hwnd, GWL_WNDPROC)。

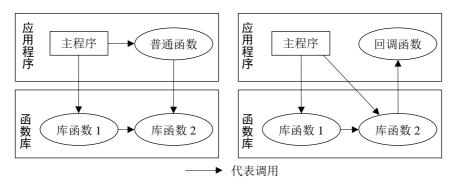


图 3-4 普通函数与回调函数的对比

"如果我们把系统划分为两层^[4]:低层的函数库和高层的应用程序。同样作为 主函数的辅助函数,左图中的普通函数直接被主函数调用,然而右图中的回调函数 却是通过库函数间接被主函数调用的。"冒号的手影在幻灯下上下翻飞。

[4]后面的论述同样适用 于其他形式的软件分层 结构。

句号点出要害: "一般都是高层代码调用低层代码, callback 反其道而行之, 因此显得与众不同。"

"所言极是。一方面,在软件模块分层中,低层模块为高层模块提供服务,并且不能依赖高层模块,以保证其可重用性;另一方面,通常被调者(callee)为调用者(caller)提供服务,调用者依赖被调者。两相结合,决定了低层模块多为被调者,高层模块多为调用者。但这种惯例并不总是合适的——低层模块为了追求更强的普适性和可扩展性,有时也有调用高层模块的需求,于是便邀 callback 前来相助。我们看一个简单的例子。"冒号写下一段 Java 代码——

```
String[] strings = {"Please", "sort", "the", "strings", "in",
    "REVERSE", "order"};
Arrays.sort(strings, new Comparator<String>() {
  public int compare(String a, String b)
  { return -a.compareToIgnoreCase(b); }
});
```

引号很快读懂了代码: "这是将字符串组不区分大小写地逆序排列。其中 Comparator的匿名类实现了callback,因为它的方法compare是在类库中被调用的。"

"此处 callback 的好处是显而易见的——它使得 Arrays.sort 不再局限于自然排序,允许用户自行定制排序规则,大大提高了算法的重用性。"冒号说着将幻灯片又翻到前页,"回头再看 win32 窗口程序的例子,其中第三步消息循环那段代码不依赖应用程序代码,完全可以提炼出来作为 library 的一部分。事实上,在 Visual C++里这段代码就'下放'到 MFC 类库中去了。假设窗口过程由应用程序直接调用,

那么消息循环中的代码将不再具有独立性,无法作为公因子分解出来。"

叹号块垒顿消,畅然无比: "终于搞清那个怪异的窗口过程了!每个窗口在创建时就携带了一个 callback,以后每当系统侦查到事件时,都能轻易地从事发窗口身上找到它的 callback,然后调用它以响应事件。"

"这等于将侦查事件与响应事件两项任务进行了正交分解,降低了软件的耦合度和复杂度。"句号言犹未尽,又加了一句,"就像刚才,引号负责侦查事件——警戒,逗号负责响应事件——警醒。想法很好,可惜配合不够默契,还是给人逮住了。"

逗、引二人大窘,余者大笑。

"仔细比较,以上两个 callback 的用法还是稍有不同的。在字符串组排序中,callback 在作为参数传入低层的函数后,很快就在该函数体中被调用;在窗口程序中,callback 则先被储存起来,至于何时被调用完全是未定之数。用一句话概括:前者属同步(synchronous)回调,后者属异步(asynchronous)回调。它们都使调用者不再依赖被调者,将二者从代码上解耦,异步调用更将二者从时间上解耦。"冒号显示出一副新图(如图 3-5 所示)——

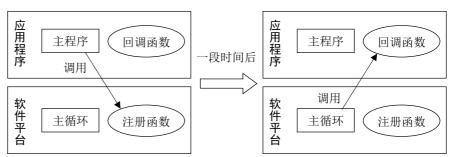


图 3-5 异步回调: 先注册, 后被调

"图中处于低层的软件平台是在 win32 API 的基础上的改进。不仅把主循环从应用程序中沉淀下来,而且将储存 callback 的过程封装在一个注册函数中,使得应用程序代码变得更简洁、健壮。同时我们看到,整个流程的控制权已经从应用程序的主程序转移到底层平台的主循环中,符合好莱坞原则。"冒号道。

逗号好奇地问: "什么是好莱坞原则?"

"Don't call us, we'll call you." 冒号难得甩出一句洋文,"我很想画蛇添足地在末尾加上单词'back',这样更容易理解 callback 的含义: 'call you back'。此话的背景大约是这样的:一个艺人要想演出,须与好莱坞的经纪公司联系。由于幻想一朝成名的人太多,经纪人总是牛气十足,他们的口头禅是: '别打电话给我们,

留下你的电话,有活干我们会打给你的'。"

引号认真地解析: "好莱坞经纪公司相当于一个背后运作的软件平台,艺人相当于一个 callback, '留下你的电话'就是注册 callback, '我们会打给你的'就是异步调用 callback。"

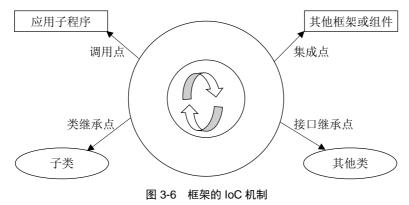
冒号接着补充: "'别打电话给我们'意味着经纪公司处于主导地位,艺人们处于受控状态,这便是控制反转(Inversion of Control,简称 IoC)。"

问号听着耳熟: "控制反转?第1课谈到框架时似乎提到过。"

"没错,正是它!"冒号谈兴愈浓,"一般 library 中用到 callback 只是局部的控制反转,而 framework 将 IoC 机制用到全局。程序员牺牲了对应用程序流程的主导权,换来的是更简洁的代码和更高的生产效率。如果将编程譬比命题作文,不用framework 的程序是一张可以自由写作的白纸,library 是作文素材库;采用framework 的程序是一篇成型的作文,作者只须填写空白的词语和段落即可。"

叹号为之一叹: "唉,编程序变成了做填空题,真没劲!"

"那你就多努力,争取以后出填空题吧。"冒号笑着鼓励他,"控制反转不仅增强了 framework 在代码和设计上的重用性,还极大地提高了 framework 的可扩展性。这是因为 framework 的内部运转机制虽是封闭的,但也开放了不少与外部相连的扩展接口点,类似插件(plugin)体系(如图 3-6 所示)——"



引号联想到另一个名词: "我知道有个依赖反转,与控制反转是一回事吗?"

冒号简答: "虽然不少人把它们看成同义词,但依赖反转原则(Dependency-Inversion Principle,简称 DIP)更加具体——高层模块不应依赖低层模块,它们都应依赖抽象:抽象不应依赖细节,细节应依赖抽象。经常相提并论的

还有依赖注射(Dependency Injection,简称 DI)——动态地为一个软件组件提供外部依赖。由于时间关系,它们之间的区别容后再叙。有一点可以看出,它们的主题是控制与依赖,目的是解耦,方法是反转,而实现这一切的关键是抽象接口。"

"为什么说是抽象接口而不是前面所说的回调函数?"打过瞌睡的逗号现在似 乎变得特别清醒。

冒号予以说明: "回调函数的提法较为古老,多出现于过程式编程,抽象接口是更现代、更 OO 的说法。另外,从字面上看,'回调'强调的是行为方式——低层反调高层,而'抽象接口'强调的是实现方式——正是由于接口具有抽象性,低层才能在调用它时无须虑及高层的具体细节,从而实现控制反转。"

众人细细品味着冒号的这番话。

问号忽然惊觉: "我们是不是跑题了?本来是谈事件驱动式编程的,结果从 callback 谈到控制反转,再到框架,现在又说起了抽象接口。"

"事物是普遍联系的嘛。"冒号扯了句哲学套话,"不谙熟 callback 和 IoC 机制,就不可能真正领会事件驱动式编程的精髓。不过,也该回到中心主题了。我们通过 win32 API 用 4 步实现了一个简单的窗口程序,与事件直接相关的有 3 步:实现事件处理器(event handler)或事件监听器(event listener);注册事件处理器;实现事件循环(event loop)。具体说来,事件处理器负责处理事件,经注册方能在事发时收到通知;事件循环负责侦查事件、预处理事件、管理事件队列和分派事件等,无事时默默等待,有事时立即响应,生命不息工作不止。在整个事件机制中,主循环好比心脏,事件处理器好比大脑,是最重要的两类模块。"

句号指出: "在支持事件驱动的开发环境中,主循环是现成的。许多 IDE 的图形编辑器在程序员点击控件后,还能自动生成事件处理器的骨架代码,连注册的步骤也免除了。"

冒号提醒他: "并不是总有这样的好事,要知道事件驱动式并不局限于GUI应用,支持事件驱动的开发环境也未必唾手可得。程序员有时必须自行设计整个事件系统,他须要决定:采用事件驱动式是否合适?如果合适,如何设计事件机制?其中,包括事件定义、事件触发、事件侦查、事件转化、事件合并、事件调度、事件传播、事件处理、事件连带(event cascade)^[5]等一系列问题。"

逗号扮着苦相说:"我的脑袋就是一个事件监听器,在听到要面临这么多的事件后,迅速作出反应——大了一圈。"



[5]指事件处理器在处理 过程中又产生新的事件, 从而再次触发事件处理 器。 众皆弯腰捧腹。

"脑袋能变大是件好事啊,说明它伸缩性强,相信用它来编的程序也是一样。" 冒号打着哈哈, "事件驱动式的程序可伸缩性就很强,知道为什么吗?"

叹号随口说道: "不是因为利用回调函数实现了控制反转吗?"

"非也,非也。"冒号文绉绉地说,"软件的可伸缩性(scalability)一般指从容应对工作量增长的能力,常与性能(performance)等指标一并被考量。而控制反转的主要作用是降低模块之间的依赖性,从而降低模块的耦合度和复杂度,提高软件的可重用性、柔韧性和可扩展性,但对可伸缩性并无太大帮助。我们已经看到,控制反转导致了事件驱动式编程的被动性(passivity)。此外,事件驱动式还具有异步性(asynchrony)的特征,这是由事件的不可预测性与随机性决定的。如果一个应用中存在一些该类特质的因素,比如频繁出现堵塞呼叫(blocking call),不妨考虑将其包装为事件。"

问号打岔道: "什么是堵塞呼叫?"

冒号作了个比方:"在高速公路上一辆车突然出故障停在路途,急调维修人员。如果现场修理,在修好之前所在车道是堵塞的,后面车辆无法通行。类似地,在程序中一些函数须要等待某些数据而不能立即返回^[6],从而堵塞整个进程。"

引号道出常识: "显然更可取的修车做法是:先把车拖到路边,修完后向其他车辆发出信号,以便重回车道。"

冒号趁热打铁: "同理,我们可以让堵塞呼叫暂时脱离主进程,事成之后再利用事件机制申请重返原进程。相比第1种同步流程式的方案,这种异步事件式将连续的进程中独立且耗时的部分抽取出来,从而减少随机因素造成的资源浪费,提高系统的性能和可伸缩性。"

问号听得仔细:"为什么抽取的部分是'独立且耗时',而不是'随机且耗时'?"

"问得好!"冒号很欣赏他严谨的学风,"再拿修车来说,第2种方案之所以可行有两方面原因:一是修车耗时,二是修车独立。所谓独立又有两层含义:与车道独立——修车时不必占用车道;与后车独立——后面车辆不必恭候该车。如果一分钟内能修好,或者路边没有足够空位,再或者后面车辆是故障车的随行车,那么拖车方案均不成立。大家可以自己类比堵塞呼叫的情形,我就不再饶舌了。总之,独立是异步的前提,耗时是异步的理由。至于随机嘛,只是副产品,一个独立且耗时的子过程,通常结束时间也是不可预期的。"



插语

[6]比如套接字(socket) 中的 accept 函数。

注册/注销 事件管理器 事件处理器 通知事件 事件源 发表事件 事件转化 通知事件 事件合并 事件处理器 通知事件 事件排队 发表事件 事件分派 事件处理器 事件源 注册/注销

眼见天色已晚,冒号赶忙换上最后一页幻灯片(如图 3-7 所示)——

图 3-7 事件驱动式模型

"图 3-7 为一个典型的事件驱动式模型。事件处理器事先在关注的事件源上注册,后者不定期地发表事件对象,经过事件管理器的转化(translate)、合并(coalesce)、排队(enqueue)、分派(dispatch)等集中处理后,事件处理器接收到事件并对其进行相应处理。请注意事件处理器随时可以注册或注销事件源,意味着二者之间的关系是动态建立和解除的。"冒号在幻灯屏上指指点点,"通过事件机制,事件源与事件处理器之间建立了松耦合的多对多关系:一个事件源可以有多个处理器,一个处理器可以监听多个事件源。再换个角度,把事件处理器视为服务方,事件源视为客户方,便是一个 client-server 模式。每个服务方与其客户方之间的会话(session)是异步的,即在处理完一个客户的请求后不必等待下一请求,随时可切换(switch)到对其他客户的服务。更有甚者,事件处理器也能产生事件,实现处理器接口的事件源也能处理事件,它们可以角色换位,于是又演化为peer-to-peer 模式。"

叹号抱怨: "有点眼花缭乱了。"

为湿润枯燥的理论,冒号再次举例: "你们不是很喜欢在QQ上聊天吗? QQ服务器是事件管理器,每个聊天者既是事件源又是事件处理器,这正是事件驱动式的P2P模式啊^[7]。此外,聊天时不等对方回答,就可与另一网友交谈,这就是会话切换带来的异步效果。不过同样是聊天,改用电话就稍有不同了。"

冒号扫了众人一眼,果见有人皱起了眉头。

"当你正用座机通话时, 手机响了。你会怎么做?"冒号提示。



[7]真正的 P2P 网络是不 需要中心服务器的,此处 P2P 指聊天双方是不分 主客的对等关系。 逗号本能地回答: "要么挂掉电话再接手机,要么让打手机的人迟些打来。"

句号听出了门道:"这说明电话的通话过程是同步而非异步的,原因是打电话双方的交流是连贯的、非堵塞式的(non-blocking),与 QQ 聊天正好相反。"

冒号点头称许。

虽然早已过了下课时间,引号仍是好学不倦:"我觉得观察者模式与事件驱动式很像啊。"

"你开始不是还举了订阅杂志和RSS的例子吗?发行/订阅模式(publish-subscribe pattern)^[8]正是观察者模式(observer pattern)的别名,一方面可看作**简化** 或退化的事件驱动式,另一方面可看作事件驱动式的核心思想。该模式省略了事件管理器部分,由事件源直接调用事件处理器的接口。这样更加简明易用,但威力有所削弱,缺少事件管理、事件连带等机制。著名的MVC(Model-View-Controller)架构正是它在架构设计上的一个应用^[9]。"冒号长舒了一口气,准备收工,"事件驱动式的应用极广,变化极多,还涉及框架、设计模式、架构,以及其他的编程范式,本身也可作为一种架构模型。今天我们仅仅是蜻蜓点水,更深入、更具体的内容只能留后探讨了。时候不早,你们也该饿了,赶快回家吧!范式可不能当饭吃哦。"



插语

[8]有人将发行-订阅模式 视为事件驱动设计的同 义词,这是有道理的:在 实际生活中,处于出版商 与订阅者之间的邮局可 作为事件管理器。

[9]MVC也可作为一种设 计模式,同样是观察者模 式的应用。

众人笑作鸟兽散。

总结

- 事件是程序中令人关注的信息状态上的变化。在基于事件驱动的系统中, 事件包括内建事件与用户自定义事件,其中内建事件又分为底层事件和 语义事件。此外,事件还有自然事件与合成事件之分。
- Callback 指能作为参数传递的函数或代码,它允许低层模块调用高层模块,使调用者与被调者从代码上解耦。异步 callback 在传入后并不立即被调用,使调用者与被调者从时间上解耦。
- 控制反转一般通过 callback 来实现,其目的是降低模块之间的依赖性,从 而降低模块的耦合度和复杂度。
- 在框架设计中,控制反转增强了软件的可重用性、柔韧性和可扩展性, 减少了用户的负担,简化了用户的代码。
- 控制反转、依赖反转原则和依赖注射是近义词,它们的主题是控制与依赖,目的是解耦,方法是反转,而实现这一切的关键是抽象接口(包括

- 函数指针、抽象类、接口、C++中的泛型函子和 C#中的委托)。
- 事件驱动式编程的 3 个步骤:实现事件处理器;注册事件处理器;实现事件循环。
- ▶ 异步过程在主程序中以非堵塞的机制运行,即主程序不必等待该过程的 返回就能继续下一步。异步机制能减少随机因素造成的资源浪费,提高 系统的性能和可伸缩性。
- ▶ 独立是异步的前提,耗时是异步的理由。
- 事件驱动式最重要的两个特征是被动性和异步性。被动性来自控制反转, 异步性来自会话切换。
- 》 观察者模式又名发行/订阅模式,既是事件驱动式的简化,也是事件驱动式的核心思想。MVC 架构是观察者模式在架构设计上的一个应用。

6 参考

- [1] Wikipedia. Event-driven programming. http://en.wikipedia.org/wiki/Event-driven
- [2] Wikipedia. Callback (computer science). http://en.wikipedia.org/wiki/Callback_(computer_science)
- [3] Charles Petzold. Programming Windows, 5th ed.. Redmond: Microsoft Press, 1999. 41-70
- [4] Robert C. Martin. Agile Software Development: Principles, Patterns, and Practices (影印版). 北京: 中国电力出版社, 2003. 127-134
- [5] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html
- [6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley, 1994. 293-299

课后思考

- 03-01 了解 C++中的 STL、Java 中的 Collections Framework 和 C#中的 Collection Classes。
- 03-02 了解 C++、Java 和 C#中的泛型机制,比较它们之间的异同,以及各自

- 在集合(collection)中的应用。
- 03-03 在你成功构想并实现了一个算法后,是否考虑过利用泛型编程来扩大其 适用范围以提高其重用性?
- 03-04 当你发觉几个模块中有类似的算法时,是否考虑过利用泛型思想进行重构?
- 03-05 当你发觉程序中有大量类似的代码时,是否考虑过用产生式编程来自动生成它们?
- 03-06 试着利用编译器生成器(如 ANTLR)自定义一种 DSL,并用它来解决问题。
- 03-07 你采用过 AOP 吗? 它有哪些优缺点?
- 03-08 如何合理地抽象出系统的横切关注点?
- 03-09 请对比流程驱动式编程与事件驱动式编程之间的差异,它们各自适合哪些应用?
- 03-10 你编写的代码是否有足够的灵活性和可扩展性?能否利用控制反转原理?
- 03-11 你在程序中是如何处理堵塞呼叫的? 是否考虑过引入异步机制?

第10课 多态机制

课前导读

本课通过实例编程和对抽象类型的解读,显示了OOP中多态机制和抽象 类型的重要性,有助于培养和加深读者的OOP语感。



本课共分两节——

- 1. 多态类型——静中之动
- 2. 抽象类型——实中之虚

10.1 多态类型——静中之动

动静屈伸, 唯变所适。

——《王弼·周易略例》》

关键词: 多态;继承;参数多态;包含多态;子类型多态;模板方法模式;策略模式

摘 要: 通过实例展示多态类型的 3 种用法



预 览

继承是多态的基础,多态是继承的目的。

多态是动静结合的产物,将静态类型的安全性和动态类型的灵活性融为一体。

前者(参数多态)是发散式的,让相同的实现代码应用于不同的场合。

后者(包含多态)是收敛式的,让不同的实现代码应用于相同的场合。

模板方法模式突出的是稳定坚固的骨架,策略模式突出的是灵活多变的手腕。



提问

多态与继承有何关系?

多态的重要意义何在?

多态有哪几种形式?它们各自有什么特点?

什么是策略模式? 它与模板方法模式有何相同点和不同点? 多态在其中起到 了什么作用?

讲解

当冒号迈着不变的步伐出现在教室时,手上有了一点变化:左手仍拎着笔记本包,右手却多了一样东西。大家定睛一看,原来是个电脑主板,不由得暗自纳闷:难道软件课改成了硬件课?

冒号照例直入主题:"上节课我们对继承的利弊作了详细的分析,其中最重要的观点是:继承的主要用途不是代码重用,而是代码被重用。这依赖于两个前提,一个是在语义上遵循里氏代换原则,另一个是在语法上支持多态(polymorphism)机制。因此不妨说,对于静态类型语言来说,继承是多态的基础,多态是继承的目的。"

问号忍不住问: "为什么要强调静态类型呢?"

"还记得鸭子类型[1]吗?那就是一种不依赖于继承的多态类型,也是动态类型 语言一大优劣参半的特色。"冒号提醒道,"静态类型语言中的多态是动静结合的 产物,将静态类型的安全性和动态类型的灵活性融为一体。它一般有两种实现方式: 一种利用GP(泛型编程)中的参数多态(parametric polymorphism),一种利用OOP 中的包含多态(inclusion polymorphism)或称子类型多态(subtyping polymorphism)。 从实现机制上看,二者的不同之处在于何时将一个变量与其实际类型所定义的行为 挂钩。前者在编译期,属于早绑定 (early binding) 或静态绑定 (static binding) [2]; 后者在运行期,属于迟绑定 (late binding)或动态绑定(dynamic binding)。从应 用形式上看,前者是发散式的,让相同的实现代码应用于不同的场合;后者是收敛 式的, 让不同的实现代码应用于相同的场合。从思维方式上看, 前者是泛型式编程 风格,看重的是算法的普适性;后者是对象式编程风格,看重的是接口与实现的分 离度。尽管二者从范式到语法、语义都大相径庭,但都是为着同一个目的:在保证 必要的类型安全的前提下,突破编译期间过于严苛的类型限制。对于既是静态类型 语言又是静态语言、既支持OOP又支持GP的C++、Java和C#而言,多态机制是保证 代码的灵活性、可维护性和可重用性的终极武器。为了说明问题,我们看一个简单 而实用的例子:编写一个类,让它能储存用户名和密码,以作今后验证之用。"

叹号一愣: "这题是不是太简单了?还有别的要求吗?"

冒号摇摇头。

引号却认为: "要求太少反而不好做。比如把数据放在内存还是文件,或者数据库?密码以明文还是密文的形式存储?"



插语

[1]参见§5.2。

[2]虽然 C#具体的泛型类型是在运行期间实例化的,但每类泛型对应相同的实现代码,故变量的行为仍是在编译期间决定的。

句号提出: "无论是数据的存放方式还是密码的加密方式,都不应该硬编码。"

"循此思路,我们就来编写一个可重用的抽象类。"冒号投放了一段 Java 代码——

```
/** 一个可以验证用户名和密码的类 */
abstract class Authenticator
   /** 保存用户名和密码 */
   final public void save(String user, String password)
      if (password == null)
         password = "";
      store(user, encrypt(password));
   /** 验证用户名和密码 */
   final public boolean authenticate(String user, String
    password)
      String storedPassword = retrieve(user);
      if (storedPassword == null) return false; // 无此用户
      if (password == null)
         password = "";
      return storedPassword.equals(encrypt(password));
   /** 保存用户名和加密过的密码 */
   protected abstract void store(String user, String
encryptedPassword);
   /** 从用户名获取相应的加密过的密码 */
   protected abstract String retrieve(String user);
   /** 给明文单向(one-way)加密, 默认不加密 */
   protected String encrypt(String text) { return text; }
```

冒号解说道: "该抽象类有两个 public 接口,一个用来保存,一个用来验证。它们用 final 修饰符来禁止子类覆盖,因为真正的扩展点是 3 个 protected 方法。其中 store 和 retrieve 是抽象的,encrypt 有一个平凡实现。以此为基础,再根据实际需要来编写子类,具体实现这 3 个方法。"

幻灯片转到下一页——

```
import java.util.Map;
import java.util.HashMap;
/** 一个简单的验证类,数据放在内存,密码保持明文 */
class SimpleAuthenticator extends Authenticator
```

"我们利用 HashMap 来储存数据,密码保持明文。这大概是最简单的一种子类了。"冒号仿佛在轻轻地把玩着一件小物什,"为安全起见,最好还是将密码加密。于是我们设计了稍微复杂一点的子类——"

```
import java.security.MessageDigest;
/** 一个安全的验证类,数据放在内存,密码经过SHA-1加密 */
class ShalAuthenticator extends SimpleAuthenticator
   // SHA-1算法
   private static final String ALGORITHM = "SHA-1";
   // 避免依赖平台
   private static final String CHARSET = "UTF-8";
   @Override protected String encrypt(String plainText)
      try
         MessageDigest md =
          MessageDigest.getInstance(ALGORITHM);
         md.update(plainText.getBytes(CHARSET));
         byte digest[] = md.digest();
         // BASE64编码比十六进制编码节省空间
         //为简便起见用到了非标准的API, 因此以下代码有警告
         return (new
           sun.misc.BASE64Encoder()).encode(digest);
      catch (java.security.NoSuchAlgorithmException e)
         // 不可能发生
         throw new InternalError(e.getMessage());
      catch (java.io.UnsupportedEncodingException e)
         throw new InternalError(e.getMessage());
                                               // 不可能发生
```

}

逗号质疑道: "不是具体类不宜被继承的吗? 怎么 Sha1Authenticator 类却继承了具体类 SimpleAuthenticator?"

冒号略表赞许: "很高兴你没有忘记这个原则。不过考虑到 Sha1Authenticator 类需要覆盖父类的 encrypt 方法,这么做也是情有可原的。当然,最好选择让该类直接继承抽象类 Authenticator,但作为示例代码,我们还是希望它简洁一些,不想让过多的细枝末节掩盖核心主干。下面是测试代码——"

```
public class TestAuthenticator
{ // 为避免额外依赖,没有采用JUnit等单元测试工具
   public static void main(String[] args)
      test(new SimpleAuthenticator());
      test(new ShalAuthenticator());
   // 测试给定的Authenticator
   private static void test(Authenticator authenticator)
   // 子类型多态
      test(authenticator, "user", "password");
      test(authenticator, "user", "newPassword");
      test(authenticator, "admin", "admin");
      test(authenticator, "guest", null);
      test(authenticator, null, "pass");
      authenticator.save("scott", "tiger");
      // 大小写敏感
      assert(!authenticator.authenticate("scott", "TIGER"));
      // 大小写敏感
      assert(!authenticator.authenticate("SCOTT", "tiger"));
   private static void test(Authenticator authenticator,
    String user, String password)
      authenticator.save(user, password);
      assert(authenticator.authenticate(user, password));
}
```

引号觉得眼熟: "这不是上节课讲的模板方法模式吗?"

"正是此公。"冒号确认,"该模式的核心思想是:固定整体框架和流程以保证可重用性,留出一些子类定制点以保证可扩展性。在测试代码的两个 test 方法中,传入的参数是 Authenticator 类,但数据存放和密码加密的方式是在运行中才确定的,即先后遵照 SimpleAuthenticator 类和 Shal Authenticator 类的实现。这就是我

们所说的子类型多态的效果——让不同的实现代码应用于相同的场合。假设没有多态机制,这种效果就只能靠 if/else 或 switch 之类的条件语句才能实现,非常地痛苦。"

冒号的眉头皱成了粗体的"川"字。

"还有更好的方法吗?"句号察言观色,断定老冒还留有后手。

果不其然,冒号的眉毛立刻又舒展开来,中气充沛地应道: "有!诸位请看——"

```
// 键值对的存取接口
interface KeyValueKeeper
   public void store(String key, String value);
   public String retrieve(String key);
// 加密接口
interface Encrypter
   public String encrypt(String plainText);
class Authenticator
   private KeyValueKeeper keeper;
   private Encrypter encrypter;
   public Authenticator(KeyValueKeeper keeper, Encrypter
     encrypter)
      this.keeper = keeper;
      this.encrypter = encrypter;
   public void save(String user, String password)
      if (password == null)
          password = "";
      keeper.store(user, encrypter.encrypt(password));
   public boolean authenticate(String user, String password)
      String storedPassword = keeper.retrieve(user);
      if (storedPassword == null) return false;
      if (password == null)
         password = "";
      return storedPassword.equals(encrypter.
        encrypt(password));
```

```
}
```

冒号加以引导:"如果仔细比较两种设计,就会发现它们很相似。后者只不过把前者对子类开放的接口合成为自己的两个成员。再看接口的实现类——"

```
class MemoryKeeper implements KeyValueKeeper
   private Map<String, String> keyValue
    = new HashMap<String, String>();
   @Override public void store(String key, String value)
      keyValue.put(key, value);
   @Override public String retrieve(String key)
      return keyValue.get(key);
class PlainEncrypter implements Encrypter
   @Override public String encrypt(String plainText)
      return plainText;
class ShalEncrypter implements Encrypter
   private static final String ALGORITHM = "SHA-1";
   private static final String CHARSET = "UTF-8";
   @Override public String encrypt(String plainText)
      try
          MessageDigest md =
            MessageDigest.getInstance(ALGORITHM);
          md.update(plainText.getBytes(CHARSET));
          byte digest[] = md.digest();
          return(new sun.misc.BASE64Encoder()).encode(digest);
      catch (java.security.NoSuchAlgorithmException e)
          throw new InternalError(e.getMessage());
      catch (java.io.UnsupportedEncodingException e)
          throw new InternalError(e.getMessage());
   }
```

逗号比较后得出结论: "MemoryKeeper 与 SimpleAuthenticator、Sha1Encrypter 与 Sha1Authenticator 除了超类型和方法访问修饰符外,其他毫无二致。"

屏幕滚动出另一段代码——

"测试代码区别也不大,只是 Authenticator 的多态性更加隐蔽。"冒号如是说。叹号挑剔说: "后一种创建实例稍显麻烦一些。"

"但它是以小弊换大利。"冒号朗声而道,"首先,后者用的是合成与接口继承,比前者的实现继承更值得推荐,理由在上堂课业已阐明。其次,假设共有M种数据存取方式,包括内存、文件、数据库,等等;共有N种加密方式,包括明文、SHA-1、SHA-256、MD5,等等。按第 1 种设计,需要(M×N)个实现类;按第 2 种设计,只要(M+N)个实现类。这还只是两种变化因素,假如需要考虑更多的因素,二者差距将更大。比如增加编码方式:加密后的数据可以选择费空间省时间的十六进制编码、费时间省空间的BASE64编码、省时间省空间却包含非打印字符的原始形式等。比如,增加安全强度;引入salt、nonce或IV等^[3],比如,增加密码状态:已生效密码、未生效密码、已过期密码,等等。对比下面的UML类图,孰优孰劣更加一目了然。"

众人眼前出现了两幅图(图 10-1 和图 10-2) ——



插语

[3]salt、nonce 和 IV 都是 密码学中的术语,是在加 密过程中混入的一次性 数据,以增加预计算攻击 (如字典攻击)的难度。

Authenticator +save(user: String, password: String): void +authenticate(user: String, password: String): boolean #store(user: String, password: String): void #retrieve(user: String): String #encrypt(plainText: String): String SimpleAuthenticator DatabaseAuthenticator Md5Authenticator DatabaseMd5Authenticator FileAuthenticator FileSha1Authenticator FileMd5Authenticator

图 10-1 Authenticator 的 UML 类图 (模板方法模式)

(注: 与示例代码中稍有不同的是 Shal Authenticator 直接继承 Authenticator)

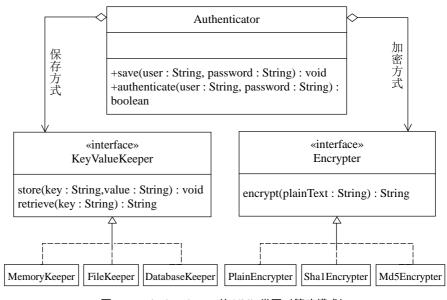


图 10-2 Authenticator 的 UML 类图 (策略模式)

冒号指着屏幕问: "图 10-2 不仅比图 10-1 少了 3 个实现类,而且可重用性也更高。大家说是为什么?"

引号应答: "图 10-1 中的 9 个 Authenticator 的子类只能作为验证类来重用,而图 10-2 中 6 个实现类不仅可以合作完成验证类的功能,还能分别单独提供键值存储和加密字符串的功能。"

冒号作出肯定:"这就是职责分离的好处。存储与加密本是两样不相干的工作,必要时可以合作,但平时最好分开管理,符合'低耦合、高内聚'的原则。"

问号注意到图中的注释,遂问: "第2种采用的是策略模式?"

冒号颔首: "简单地说,策略模式(strategy pattern 或 policy pattern)的基本 思想是: 把一个模块所依赖的某类算法委交其他模块实现。比如,Java 中的 Comparable 和 Comparator、C#中的 IComparer 就是比较算法的接口,当一个类的 某个方法接收了此种类型的参数,实质上就采用了策略模式。"

逗号不以为奇:"这岂非很平常?"

"你认为设计模式真的高不可攀吗?"冒号反问道,"包括模板方法模式,你们很可能也在编程实践中采用过,只不过相交不相识罢了。"

句号看出: "模板方法模式与策略模式非常神似,都是把一个类的可变部分移 交给其他类处理。"

"照你这么说,绝大多数设计模式都是神似的,这也是为什么我们不专门谈设计模式的缘故。GoF设计模式是OOP大树上结出的硕果,在你心中培养的OOP成熟之前,匆忙缔结的果实多半是青涩弱小的。"冒号忠告,"我们也不会对设计模式避而不谈,但凡提及都是水到渠成的产物。再说回这两种设计模式,虽然有相通的思想,也能解决相同的问题,在稳定性与灵活性之间都取得了某种平衡,但还是各有侧重的。模板方法模式突出的是稳定坚固的骨架,策略模式突出的是灵活多变的手腕。不妨拿国家政策作比:一个强调对内要稳,老一辈制订了大政方针,下一代必须在坚持原则的前提下进行完善;一个强调对外要活,不能或不便自行开发的技术不妨从国外引进。"

叹号一乐: "哈!设计模式上升到了政策模式。"

冒号抽丝剥茧: "正如模板方法模式可看作控制反转的特例,策略模式与依赖注射(Dependency Injection)也异曲同工。第 2 个Authenticator所依赖的两个功能 KeyValueKeeper和Encrypter,就是是通过构造方法'注射'进来的^[4]。当然策略只是一种特殊的依赖,是自内而外的——将算法抽出来外包;依赖注射的机制更复杂、涵盖面更广,是自外而内的——从外部嵌入定制功能。后者被广泛地用于框架应用



插语

[4]这被称为 constructor injection, 另外两种常用的 注射 方法 是 setter injection 和 interface injection。

之中, 尤以Spring Framework为代表。"

引号听得起劲:"这下热闹了,设计模式、框架与 OOP 范式全搅和到一块了。"

"还有 GP 范式呢。"冒号顺接话题,"让我们再用 C++的模板来实现一下 Authenticator 类吧。没有继续采用 Java,是因为它的泛型仍离不开子类型多态。"

说着,他换上了C++代码——

```
#include <string>
#include <map>
using namespace std;
template <typename KeyValueKeeper, typename Encrypter>
class Authenticator
   private:
      KeyValueKeeper keeper;
      Encrypter encrypter;
   public:
      void save(const string& user, const string& password)
          keeper.store(user, encrypter.encrypt(password));
      bool authenticate(const string& user,
        const string& password) const
          string storedPassword;
          if (!keeper.retrieve(user, storedPassword))
           return false;
         return storedPassword == encrypter.encrypt(password);
};
class MemoryKeeper
   private:
      map<string, string> keyValue;
   public:
      void store(const string& key, const string& value)
          keyValue[key] = value;
      bool retrieve(const string& key, string& value) const
          map<string, string>::const_iterator itr
           = keyValue.find(key);
          if (itr == keyValue.end()) return false;
          value = itr->second;
```

```
return true;
       }
};
class PlainEncrypter
   public:
      string encrypt(const string& plainText) const
       { return plainText; }
};
class ShalEncrypter
   public:
      string encrypt(const string& plainText) const
       { /* 省略代码 */ }
};
namespace
   template <typename K, typename E>
      void test(Authenticator<K, E> authenticator) // 参数多态
       { /* 省略代码 */ }
int main()
   test(Authenticator<MemoryKeeper, PlainEncrypter>());
   test(Authenticator<MemoryKeeper, ShalEncrypter>());
```

"以上代码与Java版的策略模式代码很相似,主要的区别是把KeyValueKeeper和Encrypter两个接口换成了模板参数。由于模板是在编译期间实例化的,因此没有动态绑定的运行开销,但缺点是不能动态改变策略^[5]。"冒号分析道,"至此,我们通过一个验证类的3种解法,分别展示了3种形式的多态:基于类继承的多态、基于接口继承的多态和基于模板的多态。它们殊途同归,都能让代码更简洁、更灵活、可重用性更高、更易维护和扩展。"

问号想到一个问题: "C语言既没有子类型多态也没有参数多态,又如何保证高质量的C程序呢?"

冒号眉梢轻挑: "C语言有指针啊,C++、Java和 C#的多态在底层就是用指针实现的。C中的函数指针比 Java中的接口更加灵活高效,当然对程序员的要求也更高。"

引号蓦地记起:"重载不也是一种多态吗?"

"刚才所说的多态都属于通用多态(universal polymorphism)。此外,还有一



插语

[5] 对用 Java 实现的 Authenticator 类 (策略模 式版) 稍加修改,就能让 客户动态改变策略。 类特别多态(ad-hoc polymorphism),常见有两种形式。一种是强制多态(coercion polymorphism),即一种类型的变量在作为参数传递时隐式转换成另一种类型,比如一个整型变量可以匹配浮点型变量的函数参数。另一种就是重载多态(overloading polymorphism),它允许不同的函数或方法拥有相同的名字。特别多态浅显易懂,其重要性与通用多态也不可同日而语,故不在我们关注之列。只是要注意一点,正如子类型应遵守超类型的规范,同名的函数或方法也应遵守相同的规范。如果为贪图取名方便而滥用重载,早晚因小失大。"冒号告诫道。

逗号突发奇论: "一个多态类型的对象可以在不同的类型之间变来变去,是不是叫'变态类型'更生动些?"

"我看你就属于典型的变态类型。"句号乘机拿他开涮。

全班哈哈大笑。

总结

- ▶ 在静态类型语言中,继承是多态的基础,多态是继承的目的。
- ▶ 多态结合了静态类型的安全性和动态类型的灵活性。
- 多态可分为通用多态和特别多态两种。
- 通用多态主要包括参数多态和包含多态(或子类型多态)。它们都是为了克服静态类型过于严格的语法限制。
- ▶ 特别多态主要包括强制多态和重载多态。
- 参数多态是静态绑定,重在算法的普适性,好让相同的实现代码应用于不同的场合。
- 包含多态是动态绑定,重在接口与实现的分离度,好让不同的实现代码应用于相同的场合。
- ▶ 策略模式授予客户自由选择算法(策略)的权力。
- ▶ 模板方法模式重在稳定坚固的骨架, 策略模式重在灵活多变的手腕。
- 合理地运用基于类继承的多态、基于接口继承的多态和基于模板的多态、 能增强程序的简洁性、灵活性、可维护性、可重用性和可扩展性。

③ 参考

- [1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA: Addison-Wesley, 1994. 315-323
- [2] Luca Cardelli, Peter Wegner. On understanding types, data abstraction, and polymorphism. Computing Surveys, 1985, 17(4): 471-522

10.2 抽象类型——实中之虚

有无相生,难易相成。

——《老子·道经》

关键词:继承;抽象类型;抽象数据类型;抽象类;接口; mixin; trait

摘 要: 介绍抽象类型的种类; 意义及其用法



预览

浅显的比方只是门槛前的台阶,借之或可拾级入门,却无法登堂入室。

具体类型是创建对象的模板,抽象类型是创建类型的模块。

抽象数据类型的核心是数据抽象,而抽象类型的核心是多态抽象。

必先以术养道, 而后以道御术。

以社会身份而非个人身份作为公民之间联系的纽带,正是针对接口而非实现来 编程的社会现实版。

个体身份对应的规范抽象借助封装,以数据抽象的形式出现。

家庭身份对应的规范抽象借助继承,以类型层级的形式出现。

社会身份对应的规范抽象借助多态,以多态抽象的形式出现。

提问

具体类型与抽象类型的区别是什么?

抽象数据类型与抽象类型的区别是什么?

除接口与抽象类外,还有其他抽象类型吗?它们各有何特点和意义?

抽象类型的主要作用是什么?

在系统中应采用何种类型作为模块之间通信的数据类型?

接口是为克服(JavaC#中)抽象类不能多重继承的缺点吗?

接口与抽象类在语法和语义上各有什么不同?

标记接口有何作用?

讲解

冒号调整了焦点: "鉴于目前专注的范式是 OOP,参数多态最好放在以后的 GP 专题再作探讨。除非特别说明,下面提到的多态专指子类型多态。谈到这类多 态,就不得不提及抽象类型。谁来说说,究竟什么是抽象类型?"

冒号抬手内扬,摆出了对练的姿势。

叹号率先抢攻:"抽象类型指的是至少含有一个抽象方法的类型。"

冒号轻松化解: "在 C++中这句话尚可勉强成立,但在 Java 和 C#中则大不尽 然: 一个类即使没有一个抽象方法也可以被申明为抽象的: 一个没有任何成员的空 接口或称标记接口同样属于抽象类型。"

"抽象类型是指无法实例化的类型。"逗号发起二次进攻。

冒号见招拆招: "Java 中的 Math 类也不能实例化,原因是它只有 private 构造 器,并且没有一个能返回实例的静态方法。C#中的 Math 类是静态类,同样不能实 例化。"

问号纵身而上:"抽象类型指能且只能通过继承来实例化的类型。Math 类是 final 类,无法被继承。最主要的是,它的价值体现在它的静态方法上,压根儿就没 有实例化的必要。"

冒号借力反打: "为什么要强调无法实例化呢?"

引号一旁助攻: "一个抽象类型代表着一个抽象概念,而抽象概念自然是无法 具化的。比如你无法实例化抽象的形状,但可以实例化长方形、三角形等具体的形状;无法实例化抽象的水果,但可以实例化苹果、桔子等具体的水果。"

"很官方的说法。这就好比将继承关系说成'is-a'关系一样,理论上虽通俗易懂,实践上却不足为训。"冒号收起架势,"要说抽象,Java和C#中的Object类可谓包罗万象,该够抽象了吧?不照样实例化?列表(list)与映射(map)是抽象的还是具体的?在C++中它们是具体类型,而在Java和C#中它们却是抽象类型[1]。这又是为什么?"

一连串的反问让大家陷入沉思。

"相比其他编程范式,OOP 更贴合客观世界,人们经常用打比方的形式来描述和理解OOP 的一些概念和思想。这本身并无不妥,但一定要保持清醒的头脑:浅显的比方只是门槛前的台阶,借之或可拾级入门,却无法登堂入室。"冒号谆戒道,"天下之理皆同,天下之人皆同,故凡学问殿堂之前皆一般景象:入门者众,入室者寡。本班的目的便是,引导诸位从徘徊于编程之门左右的人群中越众而出,早达内室。"

"那就成了传说中的内室弟子吧?大伙在门边转悠很久了,头都发晕了,师父还是快些领我等入室吧。" 逗号近乎戏谑地恳求。

冒号一笑: "我可算不得你们的师父,只不过是个闻道在先的师兄而已。"

一直没有出手的句号忽然开腔:"抽象是个相对概念,一个类型是否是抽象的 完全取决于设计者对它的角色定位。如果想用它来创建对象,它就是可实例化的具 体类型;如果想用它来作为其他类型的基类,它就是不可实例化的抽象类型。"

"这才击中了要害!"冒号不禁喝彩道,"整理一下你的观点:具体类型是创建对象的模板,抽象类型是创建类型的模块。一个是为对象服务的,一个是为类型服务的。显然,后者的抽象性正是源自其服务对象的抽象性。就拿刚才的实例来说,模板方法模式中的Authenticator类是抽象的,是为创建子类型SimpleAuthenticator、Shal Authenticator等服务的;策略模式中的Authenticator类是具体的,是为创建对象服务的,但它合成的两个接口KeyValueKeeper和Encrypter又是为创建算法类型服务的。值得注意的是,不要把抽象类型与抽象数据类型(ADT)混为一谈,后者的抽象指的是类型的接口不依赖其实现。或者说,抽象数据类型的核心是数据抽象,而抽象类型^[2]的核心是多态抽象。"

问号想让概念更明确些:"抽象类型就只有接口(interface)和抽象类(abstract



🎐 插语

[1]C#中表示列表和映射 的抽象类型(具体类型) 分别是 IList (List) 和 IDictionary(Dictionary)。

[2]此处主要指以继承为 基础的抽象类型,如接口 与抽象类。



插语

[3]在 C++中,如果一个 类不含数据只含抽象的 成员函数(即 pure virtual function),则该类有时 被称为纯抽象类(pure abstract class),与 Java 和 C#中的 interface 的功 用大致相当。

[4] 参考文献[4] 和[5] 对 trait 有深入的介绍, 并与 mixin 作了详细的比较。 class)两种吗?"

"在Java和C#中基本上是这样,但在C++中这两种类型没有显式的区别^[3]。"冒号,"此外,动态OOP语言如Ruby、Python、Perl、Scala、Smalltalk等还至少支持mixin和trait中的一种类型。mixin直译为'混入',trait直译为'特质',为避免翻译上的问题,今后我们还是采用英文术语。这两种类型大同小异,为简便起见,下面以mixin类型为代表^[4]。它们的出现是为了弥补接口与抽象类的一些不足,更好地实现代码重用。我们知道,接口的主要目的是创建多态类型,本身不含任何实现。子类型通过接口继承只能让代码被重用,却无法重用超类型的实现代码。抽象类可以重用代码,可又有多重继承的问题。Java和C#不支持这种机制,C++虽支持但有不少弊端。"

引号奇道: "这个问题上节课不是已经解决了吗? 用合成来代替继承啊。"

冒号解释: "合成是一种解决办法,但也不是没有缺陷。首先,合成的用法不如继承那么简便优雅,这也是许多人喜欢用继承的主要原因;其次,合成不能产生子类型,而有时这正是设计者所需要的;再次,合成无法覆盖基础类的方法,也无法访问它的 protected 成员;最后,却可能算最大的缺点是:合成的基础类只能是具体类型,不能是抽象类型。"

逗号不明所以:"这能算是缺点吗?"

"如前所述,具体类型的主要任务是创造新对象,如果用作合成或继承的基础类,等于是又承担了原本抽象类型的任务——创造新类型。这不仅有越俎代庖之嫌,而且这两个任务往往也是冲突的。我们曾提出,一个类的服务应该有纯粹性和完备性。一方面,人们希望创造的新对象无所不能,因此更看重服务的完备性,倾向它包含尽可能多的功能;另一方面,人们又希望创造的新类型有所不依,因此更看重服务的纯粹性,倾向它包含尽可能少的功能。"冒号擘肌分理,"妥协的结果是,一个新类型往往只用到基础类型的部分功能,却可能受到其他功能变动的影响。虽然这种影响在良好的封装之下会大大削弱,但也难以完全消弭。"

句号思索片刻,已明其意: "换句话说,以具体类型为代码重用的基本单位, 难免颗粒度过大?"

"然也!"冒号的手在空中挽了个花,"其实作为抽象类型的接口也有类似的尴尬:对它的客户类来说,它承诺的服务是多多益善;对它的实现类来说,承诺越多负担却越重。如果能有这样一种可重用的模块,既不像具体类型那样面面俱到,又不像接口那样有名无实,也没有抽象类的多重继承之弊,岂不妙哉?"

"想必就是 mixin 了!" 叹号眼中闪过一道光芒,旋即又暗淡下来,"只可惜 Java 并不支持啊。"。

"Java 不支持就没兴趣了?" 冒号听出他的话里有话, "要成为优秀的程序员,千万不能画地为牢、自我禁锢。始终要保持一颗开放的心,不要拘于某些语言或范式,也不要囿于某些概念或技术。"

叹号的耳根有点发热。

"陌生的理论和技术开始总是拒人千里,不过一旦你了解其问题来源,它们会慢慢变得和蔼可亲起来。"冒号循循善诱,"既然具体类型和现存的两种抽象类型均有不足之处,mixin 的产生便合情合理了。它是具体类型与接口类型的一种折衷,既可有抽象方法,也可有具体方法。这一点类似抽象类,但又没有抽象类的多重继承问题。举例来说,Ruby中的 Comparable 就是一个简单却很典型的 mixin。"

问号插话: "Java 中也有 Comparable 接口啊。"

冒号道出其中差异: "Java 中的 Comparable 和 C#中的 IComparable 只有一个抽象的比较方法,而 Ruby 中的除了有类似的抽象方法——比较(<=>)之外,还提供了小于(<)、小于等于(<=)、等于(==)、大于(>)、大于等于(>=)和介于(between?)等 6 种具体方法。显而易见,多出的方法均可通过唯一抽象的比较方法来实现。"

引号一点即通: "如此一来,重用 Comparable 的类只需实现一个抽象方法,便可自动拥有另外 6 个有用的功能。这既满足了客户类的需求,又不增加实现类的负担。"

"买1送6,这买卖划算!"逗号来劲了。

冒号双眼微眯: "更划算的买卖是 Ruby 中 Enumerable。任何包含该 mixin 的 类只要实现一个遍历方法 each,便可免费得到 20 多个有关遍历和搜寻的方法。如果再实现比较方法<=>,还可获赠排序和最值方法。相比 Java 中 Enumeration 和 Iterator 接口,优势历然。"

问号很好奇: "为什么称为 mixin 呢?"

冒号述说由来: "冰淇淋中经常会掺混一些薄荷、香草、巧克力之类的调味料和花生、坚果之类的小零碎,人们管它们叫 mix-in。后来被借用来表示一种抽象类型,主要有如下特点。第一,抽象性和依赖性:本身没有独立存在的意义,必须融入主体类型才能发挥作用。第二,实用性和可重用性:不仅提供接口,还提供部分

实现。第三,专一性和细粒度性:提供的接口职责明确而单一。第四,可选性和边缘性:为主体类型提供非核心的辅助功能。"

"这些特点与风味添加料还真的颇为神似。"叹号想着想着,嘴里不自觉地咂摸了一下。

"虽然C++、Java和C#在语法上尚不支持mixin,但C++可通过多重继承、Java和C#可通过合成和接口来分别模拟mixin。不仅如此,借助切面式编程(AOP),Java和C#甚至可完全实现mixin;借助泛型式编程(GP),C++也能通过模板更好地实现mixin^[5]。"冒号点到为止,"就此我们重温前面提到的两个观点。一是编程范式之间的合作性:mixin属于OOP的范畴,但其他编程范式如切面式、泛型式,以及二者背后的元编程都能与之相通;二是设计与语言的相关性:C++、Java和C#,以及其他诸如Ruby、Python等动态语言对mixin有着不同的支持方式,这在一定程度上会影响系统的OOP设计。"

引号憧憬道: "语言是在发展的,说不定哪天 Java 也会支持 mixin 的。"

冒号以实相应: "Java的动态小兄弟Groovy在 1.6 版已经开始支持mixin ,而 C#3.0 也新引入了对mixin更友好的语法特性[6]。"

逗号提了一个长期困惑大家的问题:"每当一个新技术出现,我就觉得很矛盾: 不追怕落伍,追吧又怕落空。如何判断一个它是昙花一现,还是大势所趋呢?"

"任何技术都是在赞美与批判中成长起来的,预测它们是流星还是恒星绝非易事。就拿OOP来说,20 世纪 60 年代就出现了支持OOP的语言^[7],但直到 90 年代中后期它才真正成为主流的编程范式。这段时间恐怕比大多数人的程序员生涯还长吧。再说mixin,其实并非今日的重点,介绍它的目的不是盲目追新,而是希望透过其背后的需求驱动点,重新审视现有技术。至于它今后会不会为主流语言所接纳,反倒不是那么重要了。如果一定要我给个建议,那就是八个字: '不执一法,不舍一法'。"冒号以禅语作答,"软件技术这棵大树经过多年的快速成长,早已枝蔓丛生。欲臻不执不舍之境,当如开班导言中所说:究其根本以知过去,握其主干以知现在,察其生长点以知未来。我之所以倾向于用抽象的方式来谈论技术,正是因为抽象的东西更接近根、更接近干、更接近生长点,从而更普泛深刻,也更稳定持久。"

句号借机问道: "您认为抽象比具体更重要?"

"抽象与具体无所谓孰高孰低,它们只是功用不同而已。"冒号轻轻晃了晃脑袋,"正所谓:必先以术养道,而后以道御术。也就是说,在学习时应注重从具体知识中领悟抽象思想,在应用时应注重用抽象理论来指导具体实践。类似地,软件



[5]C++ 可 利 用 CRTP (Curiously Recurring Template Pattern)的惯用 法来实现 mixin。

[6]指 C#3.0 的扩展方法 (extension method)。

[7]第一个支持 OOP 的语言是 Simula 67 (1967年)。

开发也是如此:从具体需求中构建出抽象模型,再根据抽象模型来完成具体实现。 因此,在设计阶段抽象类型尤为关键,而在实现阶段则是具体类型更为重要。"

问号表示理解: "假如从具体需求直接跨到具体实现,省去中间的抽象建模过程,那还用得着架构师和分析师吗?"

"话虽不错,但疑似倒果为因。"冒号洞若观火,"是否有必要抽象建模,关键看项目需求。如果需求简单而稳定,一步到位又何尝不可?甚至软件的开发效率和运行效率还更高——为劈几根细柴而磨刀,值吗?如果需求复杂而多变,引入抽象方有'磨刀不误砍柴工'之效。毕竟抽象不是目的而是手段,对它片面的追求反会导致过度的设计。"

众人这才发现,给老冒戴顶"抽象派"的帽子是有些冤枉他了,应该是"抽象 现实派"的。

冒号续道: "为进一步认识抽象类型,我举个非常实用的例子。它只适用于 C++,而不适用于 Java 和 C#。如果你对这一点感到遗憾,不要忘记原则:具体实例永远是为抽象思想服务的。"

幻灯一闪,现出一段 C++代码——

```
/** 一个不可复制的类 */
class NonCopyable
   protected:
      // 非公有构造函数防止创建对象
     NonCopyable() {}
      // 非公有非虚析构函数建议子类非公有继承
      ~NonCopyable() {}
   private:
      // 私有复制构造函数防止直接的显式复制和通过参数传递的隐式复制
     NonCopyable(const NonCopyable&);
      // 私有赋值运算符防止通过赋值来复制
      // copy assignment
      const NonCopyable& operator=(const NonCopyable&);
};
/** NonCopyable的一个私有继承类 */
class SingleCopy : private NonCopyable {};
/** 测试代码 */
int main()
   SingleCopy singleCopy1;
   SingleCopy copy(singleCopy1); // 编译器报错: 企图复制singleCopy1
   SingleCopy singleCopy2;
```

```
singleCopy2 = singleCopy1; // 编译器报错: 企图复制singleCopy1 return 0; }
```

冒号讲解道: "有些对象是不希望被复制的。比如一些代表网络连接、数据库连接的资源对象,它们的复制要么意义不大,要么实现困难。由于 C++的编译器为每个类提供了默认的复制构造函数(copy constructor)和赋值运算符(assignment operator),要想阻止对象的复制,通常做法是将这两个函数私有化。引入NonCopyable 后,它的任何子类将自动拥有不可复制的特性。这样为开发者节省了代码编写量,还免掉了相应的文档说明,使用者也一望而知其意,可说是一石三鸟。虽然 NonCopyable 从语法上说不是抽象类,但从本质上看是一种类似 mixin 功能的抽象类型。"

引号考量一番后说道:"单就它的功效而言,的确非常符合 mixin 的 4 大特点,只是它的子类用的是私有继承,而不是类继承或接口继承。"

"你说得很对。可问题是,我们并没有要求 mixin 或 trait 一定要通过继承的方式来重用啊?事实上,有些 mixin 甚至可在运行期间产生,还能克服继承的静态缺陷。即使采用继承,一般也不满足'is-a'关系。你总不能说草莓冰淇淋是一种草莓吧?"冒号淡淡地说,"先前你们总结出抽象类型有两个特征:须要继承和无法实例化,但它们并非本质,关键还是它的目的——为类型服务。提供可被继承的超类型只是一种服务方式,却非唯一的方式;无法实例化只因它不是为对象服务的,禁止实例化不过是语法上的加强,目的是让用户在编译期间就能发现用法错误。其实,即便 NonCopyable 类的构造函数是公有的,也不会有人去实例化。原因很简单,它的价值只有通过子类才能体现,这是由其抽象的本性所决定的。"

逗号有些奇怪: "为什么在 Java 中就没有类似的对象复制问题呢?"

"这是一个非常基础的问题,请容我下次再回答你。"冒号破天荒地没有立即解疑,"以下重点还是放在接口和抽象类上面,我们称之为基本抽象类型,以别于mixin、trait等其他抽象类型。我们先从语法上简单地对比一下这两种类型。"

屏幕上显示出一张表格(如表 10-1 所示) ——

表 10-1 Java/C#的抽象类与接口在语法上的区别

	抽象类	接口
提供实现代码	能	否
多重继承	否	能
拥有非 public 成员	能	否

续表

	抽象类	接口
拥有域成员	能	否(Java 中的 static final 域成员除外)
拥有 static 成员	能	否(Java 中的 static final 域成员除外)
拥有非 abstract 方法成员	能	否
方法成员的默认修饰符	无	public abstract(Java:可选。C#:不能含有任何修饰 符)
域成员的默认修饰符	无	Java: public static final。C#: 不允许域成员

冒号简明扼要地总结: "C#的语法与Java的稍有不同,但二者在接口与抽象类的关键区别上还是一致的:接口不能提供实现但能多重继承,抽象类则正相反;接口只能包含公有的、非静态的、抽象的方法成员^[8],抽象类则无此限制。"

问号言明难处: "从语法上区分它们并不难,难的是从设计上区分它们。"

逗号实话实说:"按照上节课'提倡接口继承,慎用实现继承'的方针,应该倾向用接口而非抽象类。但总觉得接口太虚了,没有抽象类实在。"

引号反驳: "要说实在,具体类型更实在啊。"

叹号坦言: "在编程中经常需要用到标准的或第三方的类库,可查起 API 来经常是左一个接口、右一个接口的,迟迟不见具体类型现身,心里哪个急啊!"

冒号打了个比方: "如果到包子铺买包子,作为客户你也许会认为包子是具体类型,但对提供包子的人来说它却是抽象类型。他一定会问你:是要肉包、菜包,还是豆沙包?是要蒸包、煎包,还是小笼包?他的铺子开得越专业,给你出的选择题越多,众口难调嘛。同样道理,要建一个高度可重用的类库,一些接口是必不可少的。"

句号悟道:"接口的意义就在于:提供者不是擅作主张,而是推迟决定,让客户选择实现方式。"

"言之有理!类似地,抽象类的意义就在于:父类推迟决定,让子类选择实现方式。'推迟'二字道出了抽象类型除创建类型之外的另一功用:提供动态节点。如果是具体类型,节点已经固定,没有太多变化的余地。反过来,要使节点动态化,一般通过多态来实现。由此可见,抽象类型常常与多态机制形影不离。"冒号稍加引申,"就说前面的验证类吧,用模板方法模式实现的 Authenticator 类将关键的方法交给子类 SimpleAuthenticator 或 Sha1Authenticator 处理,用策略模式实现的 Authenticator 类将关键的方法交给内嵌接口 KeyValueKeeper 和 Encrypter 的实现类处理。后者的两次接口继承比前者的一次实现继承多了一个动态节点,因而更加灵



插语

[8] 例外情形: Java 的 interface 可含 static final 域成员, C#的 interface 还可含 property、event 或 indexer 成员。 活。这也是为什么一个需要(M×N)个实现类,一个只要(M+N)个的原因。当然,这也不是完全没有代价的。比如要创建一个用 SHA-1 算法加密的验证类实例,两种方法对比如下——"

模板方法模式: new ShalAuthenticator() 策略模式: new Authenticator(new MemoryKeeper(), new ShalEncrypter())

冒号指点着黑板: "显然,后者无论是使用上还是性能上都比前者稍有不如。 但权衡利弊, 多数时候它仍是更好的选择。"

"包子铺的包子用料种类越多、做法越多,买一个包子越费事。但只要不到饿得发昏的地步,大家还是更喜欢花样更多的包子铺。看来我也不该再抱怨类库的接口过多了。"叹号心下释然。

"大家再看看这个电脑主板,开过机箱攒过机的人应该对它并不陌生。"冒号终于亮出了蓄藏已久的道具,"上面密密麻麻地布满了各种元件,那是它的实部,而我们关注的是它的虚部——各种插槽和接口,包括 CPU 插槽、内存插槽、PCI 插槽、AGP 插槽、ATA 接口、PS/2 接口、USB 接口,以及其他林林总总的扩展插槽等。这些接口的存在,使得主板与 CPU、内存条、外围设备及扩展卡等不必硬性焊接在一起,大大增强了电脑主机的可定制性。"

引号受到启发:"主板与其他硬件就好比一个个的具体类型,那些插槽和接口就相当于一个个的接口类型。所有的硬件以接口为桥来组装合成,以机箱为壳来封装隐藏,一个新的具体类型——具有完整功能的主机便产生了。"

"比喻非常到位!"冒号很满意,"不过准确地说,与接口类型对应的不是物理接口,而是接口规范。如果仅仅是物理接口,只能保证该接口适用于某种特定型号的硬件产品,却不能保证同时适用于其他型号或其他类型的硬件。以大家熟悉的USB(Universal Serial Bus)接口为例,它能接入各种外部设备,包括鼠标、键盘、打印机、外置硬盘、闪存和形形色色的数码产品。这当然不是偶然的,因为所有厂家在生产这些硬件时均遵循了相同的业界标准——USB协议规范。换言之,任何一个与USB接口兼容的设备,都可看作是实现了此接口的具体类型,而主机对该设备的自动识别能力则可看作一种多态机制。"

"这下我更深刻地理解那句话了:接口继承不是为了重用,而是为了被重用。"句号品味道,"比如一个鼠标,可以有串行接口、PS/2接口、USB接口或无线接口,还可以同时拥有多个不同类型的接口。无论怎样,它本身都是完整的产品,根本不需

要重用主机上的其他硬件,它实现某些接口的目的完全是为了能被主机所用。"

逗号意识到: "看样子,硬件设计也需要 OOP 思想呢。"

"相比软件设计师,硬件设计师往往能更好地贯彻 OOP 的理念。"冒号加强了语气,"他们的对象化概念更清晰、更自然,因为硬件模块比软件模块更实在、更具体;他们更注重设计,因为硬件比软件的修改成本大得多;他们更注重设计重用,因为硬件重新发明轮子的成本普遍很高;他们更注重实现重用,因为无法在举手之间完成'复制-粘贴'工作;他们更注重接口明确、封装完好,因为把内部的接口或结构暴露在外不仅难看,还容易带来缠绕、磨损、短路等问题;他们采用合成和接口来组装模块,因为硬件没有类似实现继承的机制。"

"看起来我们真得向硬件设计师取经了。"叹号有些信服了。

冒号旧话重提: "我们曾对 OOP 有过这样的描述:如果把 OOP 系统看作民主制社会,每个对象是独立而平等的公民,那么封装使得公民拥有个体身份,继承使得公民拥有家庭身份,多态使得公民拥有社会身份。补充一下,其中的继承主要指类继承,多态主要指接口继承带来的多态。经过这段时间的学习,大家对此有何见解?"

问号发表看法: "广义封装让每个类成为独立的模块,从而让每个对象具备了个体身份。狭义封装又进一步地把类的接口与实现分离,从而让每个对象具有显著的外在行为和隐藏的内在特性。继承机制可使一个类成为其他类的子类或父类,从而确立了对象在类型家族中的身份。至于多态嘛,嗯……"

问号努力想抓住若隐若现的头绪。

句号接过话头:"一个公民的社会身份是指他在社会中所处的地位和扮演的角色。比如,一个人在学校里是学生,在公司里是职员,在商店里是顾客,他真正的个体身份往往是被掩盖的。同样地,一个对象在与外界联系时,通常不以其实际类型的身份出现,而是在不同的场合下以不同的抽象类型的身份出现。我想,这大概就是多态带来的社会身份吧。"

"这种社会身份的意义何在?"冒号不动声色地问。

句号接着回答: "社会身份既是一种资格,也是一种义务。比如,在列车上有人得了急病,可以通过广播找医生。人们不用事先知道来者的具体个人身份,只要他是医生,就会放心地让他第一时间去救人。"

"这个比喻很恰当。"冒号赞道,"不用事先知道个人身份,不正说明广播呼叫的对象是一个多态的抽象类型吗?同理,当一个具体类型显式继承了一个接口,

它的对象便拥有了个体身份之外的社会身份:有资格以该接口的形式与外界打交 道, 也有义务履行该接口的职责。"

"咦,那为什么把社会身份归功于多态而不是继承呢?"问号发出疑问。

冒号释疑:"继承自然有功劳,毕竟子类型多态要建立在它的基础上。但如果 没有多态机制,要确保一个对象的实际方法而不是其超类型的方法被调用,必须将 其还原为具体类型,从而使社会身份变得几乎有名无实。"

问号憬然醒悟。

冒号继续深入: "对象每多一种社会身份,便多一条与外界交流的渠道。为什 么遮遮掩掩地不肯以本来面目示人呢? 非是羞于见人,盖因一般的具体类型在公共 场合是不为人知的,只有少数核心库里的核心类是例外。即使侥幸被认识,也难被 认可,因为那会以代码的复杂度和耦合度为代价。社会身份则不然,它远比一般的 个体身份更容易被接受。"

逗号举出例证:"这就好比上课得有学生证,上班得有工作证,上火车得有火 车票,上飞机得有登机牌。只要不是炙手可热的公众人物,很多场合都是认牌认证 不认人的。"

"道理人人都懂,可总有不少人以为自己编写的类都是明星大腕,大有'天下 谁人不识我'的豪迈,无牌无证就敢到处乱窜。更有甚者,不用多态就算了,连封 装也不要,简直是在裸奔嘛。"冒号揶揄道。

全班笑不可仰。

冒号恢复肃容:"谈到这里,我们不能不再次提到'针对接口编程'的基本原 则。它有一种建立于数据抽象之上的形式,能让用户只关心抽象数据类型的 API 接口而无视其具体实现。不过,它至少有两大局限。其一,虽然在接口不变的情况 下,实现代码的改变不会影响客户代码,但仍须要重新编译,对于需要头文件的 C++来说则需要更多的编译链接时间。其二,虽然相同的接口可以有多种实现方法, 但它们不能同时并存,更无法动态切换。于是,另一种建立于多态抽象之上的形式 应运而生。它把抽象数据类型隐藏在抽象类型的背后,从而提升了抽象接口。同一 个抽象接口允许有多种实现并存,目能动态切换,新增、删除或修改某种实现也不 会导致其他代码的修改或重新编译。方才我们从主体类的角度来看,它的对象尽量 以社会身份参与社会活动:现在再从客户类的角度看,它会尽量召集有社会身份的 对象。两相结合,以社会身份而非个人身份作为公民之间联系的纽带,正是针对接 口而非实现来编程的社会现实版。"

问号有所顾虑: "可是,有不少具体类型并没有实现任何接口,也就没有社会身份。"

"排除设计不良的因素,没有抽象超类型的具体类型最常见的有两种可能。一 种是与世隔绝,一辈子几乎足不出户,至多在小圈子里活动。典型的有非公有类、 内部类、局部类,等等。一种是名满天下,他的脸就是一张天然名片,他的个人身 份也就是社会身份。典型的有基本数据类型、字符串类型、日期类型等通用数据类 型及特定领域的通用数据类型。可见,个人身份与社会身份并无绝对的界限。同样, 家庭身份与社会身份也有交合之处,正如名门望族也可成为社会身份一样。典型的 有 Java IO 库中的 InputStream 和 OutputStream、Reader 和 Writer, 以及 UI 库中的 Component 和 JComponent, 等等。"冒号信手拈来,"因此,我们谈到的社会身 份,不必拘泥于接口,甚至不必限于抽象类型,关键是该类型是否具备了足够的通 用性和规范性、稳定性和独立性、灵活性和专业性。还是应了那句话:抽象不是目 的而是手段。再拿现实社会说事,每种社会身份都代表了个体与社会缔结的一种契 约,它有如下的特点:独立而稳定——先于个体而存在,且不随个体的变化而变化; 公开而权威——为人所知、为人所信;规范而开放——制定的协议标准明确,且允 许个体在遵守协议的前提下百花齐放。毫无疑问,推行契约制将使社会大受其惠。 首先,相同身份的个体可相互替换、新型个体可随时加入,而且不会影响整体框架 和流程,保证了系统的灵活性和扩展性。其次,整体不因某一个体的变故而受冲击, 保证了系统的稳定性和可靠性;最后,个体角色清晰、分工明确,保证了系统的规 范性和可读性。"

引号非常注重概念: "社会身份所代表的契约对应的正是规范抽象吧。"

"每种身份都是规范抽象的结果。"冒号推而广之,"具体地说,个体身份对应的规范抽象借助封装,以数据抽象(data abstraction)的形式出现;家庭身份对应的规范抽象借助继承,以类型层级(type hierarchy)的形式出现;社会身份对应的规范抽象借助多态,以多态抽象(polymorphic abstraction)的形式出现。至此,我们从行为和规范两个角度分别诠释了OOP的3大特征与公民的3大身份之间的关系。这也非常合乎情理:一个合理设计和实现的类,其对象的行为与规范本应保持一致。"

句号欲印证自己的想法: "我的理解是,接口是一个携带契约的角色标签,接口继承的作用就是静态地为对象贴上该标签,而多态机制的作用就是动态地让对象发挥该角色。因此,要赋予对象某个角色,就应该让相应的类去继承相应的接口。"

"你的前半部分表述得非常精当,后半部分则稍有瑕疵。"冒号评论道,"接

口可用来代表角色,但角色却不一定要通过接口。正如你提到的,接口继承是静态的,而角色却可能是动态的。比如,学生毕业后变成职员,职员升迁后变成经理,等等。对于静态类型语言来说,这类问题的解决单靠接口继承是不够的,还需要利用合成等手段,或者利用前面提到的其他抽象类型如 mixin 或 trait。"

叹号仍有疑惑:"接口的意义已经很清楚了,那抽象类呢?它们的区别真的很大吗?"

"我们已经从语法上比较了它们的区别,那些只是表象的东西。如果对语言规则的理解仅仅停留于语法层面,那么它更多体现为对实现的束缚。只有提升到语义层面,它才更多体现为对设计的保障。"冒号保持一贯高举高打的风格,"从语义上看,抽象类与接口的区别,并不比它与具体类的区别小多少。"

叹号错愕不已: "怎么可能?抽象类与接口好歹都是抽象类型啊。"

冒号反诘: "为什么不说抽象类与具体类好歹都是类呢?"

叹号一时无语。

"先看段历史吧。"冒号幽幽地说,"开始 C++是没有抽象类型的,直到 1989 年 C++ Release 2.0 发布前的最后一刻,Bjarne Stroustrup 才力排众议引入抽象类。从 C++的前身 C with Classes 开始算起,其间已经整整 10 年了。即便如此,它的意义在当时仍不为大多数人所认识。推出一个看似小小的语法特征竟会如此艰难,恐怕远远超出诸位的想象吧!有人幻想只通过看语法书就能完全领会语言的精髓,又与痴人说梦何异?"

冒号的声音渐渐激昂起来。

逗号为自己找到了安慰:"难怪当初学到抽象类时,总感到只知其意而不知其用。"

冒号紧接着说:"抽象类的出现,让两种不同角色的类在语法上有了明确的界定:具体类描述对象,重在实现;抽象类描述规范,重在接口。这种分工降低了用户与实现者之间的耦合度,大大减少了代码的维护成本及编译时间^[9]。由于抽象类不是为了创建对象,它的实例化自然是没有意义的。又由于它是接口规范,在子类没有实现其所有规范之前,是不能实例化的,否则规范岂不成了一纸空文?在没有抽象类的语法之前,要实现类似的功能,唯一的办法是:在本该抽象的方法被调用时强行中止程序。烦琐丑陋不说,还只能在运行期间捕捉错误。在纯虚函数(pure virtual function)——相当于Java和C#中的抽象方法——被引入之后,任何含有抽象方法的类都是抽象类,编译器将保证它不会被实例化。"



[9]据参考文献[1]中介绍,一些大型系统在引入抽象类后,编译时间少了一个数量级。

问号连连点头:"从这个角度来理解抽象类的语法,一切都顺理成章了。不过, 抽象类与接口的区别好像还是没有看到。"

谈到兴头,冒号出言更如下阪走丸: "从具体类中分离出抽象类是一次质的飞 跃,从抽象类中进一步地分离出接口则是另一次飞跃。Java 推出接口类型之时同样 饱受质疑,最终还是经受了实践的考验,后又为 C#所采纳。其实最初 C++的抽象 类是为了定义一组协议并强令各子类遵守,实质上正是 Java 和 C#中的接口所起的 作用。但在协议规范的实现过程中,可能会产生一些不完全实现类。允许这种类的 存在固然是一种灵活的举措,但必须认识到它们与纯规范的抽象类已判若云泥。打 个比方,如果把对象看作产品,把具体类看作一个制作产品的模具,那么接口就是 模具的规格标准,而抽象类是在模具加工过程中产生的半成品。接口与抽象类无法 实例化,模具规格与模具半成品也不能直接制作产品:一个具体类可以有多个接口, 一个模具也可有多个不同方面的规格;一个具体类至多只能继承一个抽象类,一个 模具也至多只能在一种模具半成品的基础上直接加工。"

引号细加回味: "如果具体类、抽象类和接口分别对应于模具、模具半成品和 模具规格,那后两者的区别的确比前两者的区别还大。可是假如一个抽象类完全没 有任何实现呢? 抛开多重继承的限制,它与接口又有何区别呢?"

冒号辨析其别: "一个抽象类可以没有任何实现,但也随时可以加入实现。接 口则不同,永远都不能有实现代码。这正是引入关键字 interface 的目的,明明白白 地表明: 此乃规范集合所在, 杜绝任何自以为是、画蛇添足的实现。初看似乎不合 常理:这不是自缚手脚、自废武功吗?殊不知自由源于自制。许多人为了贪恋一点 点代码重用, 总忍不住把一些实现放在本该只是规范的地方。一来, 这模糊了规范 与实现的界限,背离了接口与实现相分离的设计初衷。要知道,再完美的实现都有 改动的余地,将其捆绑到规范中只会增加不稳定因素:再完美的实现也不应该影响 其他的实现,先入为主只会降低灵活性。二来,带有实现的抽象类无法用于合成, 必须通过类继承才能起作用,而实现继承的弊端我们已经见识过了。在有些情况下, 规范的实现比较复杂,需要渐进实现,保留一些中间状态的抽象类也是合理的,但 最初的接口最好保留。总不能因为有了模具半成品,就抛弃模具规格吧?以 Java Collections Framework 为例,既规范了 Collection、Set、List、Map 等接口,又为这 些接口提供了抽象类和具体类,从而给了用户3种选择:直接利用具体类、扩展抽 象类、直接实现接口,方便程度递减而灵活程度递增。"

句号进行反思:"我在想,为什么以前对接口总有本能的排斥心理?原因在于: 满脑子更多想的是怎么让程序工作,而不是想怎么让程序工作得更好。因此,更重 视代码实现,比较忽视规范设计。"

众人皆有同感。

"确实,在缺乏设计观念的人看来,使用接口和脱裤放屁差不多。"冒号轻笑 道, "特别需要注意一种常见的说法:接口是为了克服 Java 或 C#中抽象类不能多 重继承的缺点。这句话具有相当大的误导性, 因为该处的多重继承是指多重实现继 承,而接口甚至连单重实现继承都做不到!许多人对接口与抽象类的认识之所以模 糊不清,原因是他们习惯于从定义和语法中寻找表象的答案,不习惯从本源和语义 上进行本质的分析。然而不可否认,毕竟接口与抽象类提供了相似的抽象机制,在 实践中往往确难选择。因此光从语法上对比二者的差别是远远不够的,须要进一步 在语义上进行对比(如表 10-2 所示)——"

表 10-2 Java/C#的抽象类与接口在语义上的区别

		接口	抽象类			接口	抽象类
关	系	can-do	is-a	重	用	规范重用	代码重用
共	性	相同功能	相同种类	实	现	多种实现	多级实现
特	征	边缘特征	核心特征	重	点	可置换性	可扩展性
联	系	横向联系	纵向联系	演	变	新增类型	新增成员

冒号展开叙述: "先从本性上看:接口是一套功能规范集合,因此相同的接口代 表相同的功能,多表示'can-do'关系,常用后缀为'-able'的形容词命名,如Comparable、 Runnable、Cloneable,等等。接口一般表述的是对象的边缘特征^[10],或者说一个对象 在某一方面的特征,因此能在本质不同的类之间建立起横向联系。由于一个对象可拥 有多方面的角色特征,故而可有多种接口。与之相对地,抽象类是一类对象的本质属 性的抽象,因此相同的抽象基类代表相同的种类,多表示'is-a'关系,常用名词命名。 抽象类一般表述的是对象的核心特征,只能在本质相同的类之间沿着继承树建立起纵 向联系。由于一个对象通常只有一个核心,故而只能有一种基类。再从目的上看:接 口是为了规范重用,让一个规范有多种实现,看重的是可置换性;抽象类主要是为了

代码重用[11], 能逐级分步实现基类的抽象方法, 看重的是可扩展性。"

叹号追问: "演变指的又是什么呢?"

冒号答道: "严格说来,演变不属语义范畴,属于语法推论。在系统演变过程 中,接口与抽象类的表现差异很大。接口由于是被广泛采用的规范,相当于行业标 准,一经确立不能轻易改动。一旦被广泛采用,它的任何改动——包括增减接口、 修改接口的签名或规范——将波及整个系统,必须慎之又慎。抽象类的演变则没有 那么困难,一则它在系统中用得没有接口那么广泛,更多地是家庭身份而非社会身



[10]接口也可能描述对 象的核心特征, 但一个类 至多一个这样的接口。

[11]由于类继承同时也继 承了接口,抽象类也能规 范重用,但更侧重代码重 用。

份;二则它可随时新增域成员或有默认实现的方法成员^[12],所有子类将自动得以扩充。这是抽象类的最大优点之一。不过接口也有抽象类所不具备的优点,虽然自身难以演化,但很容易让其他类型演化为该接口的子类型。例如,JDK5.0 之前的StringBuffer、CharBuffer、Writer和PrintStream本是互不相关的,在引进了接口Appendable并让以上类实现该接口后,它们便有了横向联系,均可作为格式化输出类Formatter的输出目标。"

问号还留有一个疑点:"现在接口与抽象类之间的差异是越来越清晰了,我只是有一点一直没想通:标记接口究竟有什么用?它一个方法都没有,也就谈不上规范,也无法利用多态机制,继承这类接口又有何意义呢?"

冒号回应道: "先须澄清一点,一个类型的规范不限于单个的方法,类型整体上也有规范,比如主要目的、适用场合、限定条件、类不变量,等等。另外,接口的目的是为了产生多态类型,不能只看到'多态'而忽略'类型'。一个接口哪怕没有一个方法,也是有意义的。首先,接口是一种类型,有严格的语法保障和明确的语义提示,这也是静态类型的优势所在。让一个具体类继承特定接口,既凸显了设计者的用意,也授予用户针对性地处理该类型的权力。比如java.util.EventListener接口为所有的事件监听器提供了统一的根类型。其次,有时需要对某些类型提出特殊要求、提供特殊服务或进行特殊处理,而这些并不能通过公有方法来办到,也没有其他有效的语言支持。标记接口可担此任,成为类型元数据(metadata)的载体。比如给一个类贴上一个java.io.Serializable的标签,它的对象便能被序列化[13],具体工作由JVM来完成。用户也可以通过自定义私有的writeObject、readObject等方法来定制序列化方式。值得指出的是,当标记接口仅仅用于元数据时,更好的办法是采用属性导向式编程(@OP),Java中的annotation、C#中的attribute即作此用。"

逗号摸了摸后脑勺: "原来标记接口并非虚有其名,还是在偷偷地干实事呢。"冒号见时候已到,准备落下帷幕: "至此,我们探讨了 OOP 中最基本的机制——封装、最独特的机制——继承、最重要的机制——多态。在今天的课结束之前,请大家每人用一个关键词来形容自己眼中的 OOP,并作简要说明。"

引号说:"责任——在契约化的公民社会中,最重要的是对自己、对家庭、对社会的责任感。"

问号说:"变化——采用封装以防个人之变,慎用继承以防家庭之变,采用多态以防社会之变。"

逗号说: "分合——数据与运算结合,接口与实现分离。"



插语

[12]前提是新增的方法 成员不与子类型的方法 发生冲突。

[13]严格说来,还要求该 类 所 有 非 static 、 非 transient 的域都是可序列 化的。

句号说:"抽象——无论是封装、继承还是多态,都是施诸众对象之上的抽象 机制。"

叹号说: "虚伪——用封装来掩盖内心,用多态来掩盖外表,提倡继承责任却 不提倡继承财富!"

冒号欣赞道: "不错不错,虽然角度各异,但均深中肯綮。我也大可安心下课了!" 众人也乐得打道回府。

总结

- ▶ 具体类型是创建对象的模板,抽象类型是创建类型的模块。
- ▶ 抽象数据类型的核心是数据抽象,而抽象类型的核心是多态抽象。
- 抽象类型除了接口和抽象类外,还有 mixin、trait 等,它们用来克服以下 问题——

合成的缺陷:

用法不如继承那么简便优雅:

不能产生子类型:

无法覆盖基础类的方法,也无法访问它的 protected 成员;

不能以抽象类型为基础类。

具体类型的矛盾与缺陷:

作为创造对象的单位,功能越多越好;

作为可重用的单位,功能越少越好。

不官被继承。

接口的矛盾与缺陷:

客户类希望它提供尽可能多的服务:

实现类希望尽可能少的实现代码。

无法代码重用。

抽象类的缺陷:

接口无法代码重用, 多重类继承或复杂晦涩或未获支持。

- mixin 的特点:抽象性和依赖性;实用性和可重用性;专一性和细粒度性;可选性和边缘性。
- 在设计阶段,从具体需求中构建出抽象模型,此时抽象类型尤为关键; 在实现阶段,根据抽象模型来完成具体实现,此时具体类型更为重要。
- ▶ 抽象类型除了能创建类型外,还能提供动态节点,以增加软件的灵活性和可扩展性。
- 社会身份代表了个体与社会缔结的一种契约,具有独立、稳定、公开、权威、规范、开放等特点。
- 》 以社会身份作为公民之间联系的纽带,以接口类型作为对象之间联系的 纽带。
- 系统中广泛用于模块之间通信的数据类型相当于社会身份,提倡使用接口类型。但也不必拘泥,甚至不必限于抽象类型,关键是要确保该类型的通用性、规范性、稳定性、独立性、灵活性和专业性。
- 个体身份对应的规范抽象借助封装,以数据抽象的形式出现;家庭身份对应的规范抽象借助继承,以类型层级的形式出现;社会身份对应的规范抽象借助多态,以多态抽象的形式出现。
- 》 从具体类中分离出抽象类是一次质的飞跃,从抽象类中分离出接口则是 另一次飞跃。
- ▶ 接口与抽象类的语法区别:接口不能提供实现但能多重继承,抽象类则 正相反;接口只能包含公有的、非静态的、抽象的方法成员,抽象类则 无此限制。
- 》 接口与抽象类的语义区别:接口是一套功能规范集合,相同的接口代表相同的功能,多表示"can-do"关系,一般是对象的边缘特征,在本质不同的类型之间建立横向联系;抽象类是一类对象的本质属性的抽象,相同的抽象基类代表相同的种类,多表示"is-a"关系,一般是对象的核心特征,在本质相同的类型之间建立纵向联系。接口看重规范重用和可置换性;抽象类看重代码重用和可扩展性。
- 接口与抽象类的演变:抽象类的演变较为容易;接口自身很难演变,但很容易让其他类型演变为它的子类型。
- ▶ 标记接口除了能定义类型外,还可作为类型元数据的载体。

🥯 参考

- [1] Bjarne Stroustrup. The Design and Evolution of C++. Reading, MA: Addison-Wesley, 1994. 277-281
- [2] Joshua Bloch. Effective Java: Programming Language Guide. Boston, MA: Addison-Wesley, 2001. 84-88
- [3] Nathanael Schärli, St é phane Ducasse, Oscar Nierstrasz, Andrew P. Black. Traits: Composable Units of Behaviour. ECOOP, 2003, LNCS 2743: 248–274
- [4] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, Andrew P. Black. Traits: A Mechanism for Fine-grained Reuse. ACM Transactions, 2006, 28(2): 331-388
- [5] Wikipedia. Mixin. http://en.wikipedia.org/wiki/Mixin

课后思考

- 10-01 多态类型在何种程度上解放了静态类型的束缚?
- 10-02 请总结参数多态与子类型多态的特点和适用场合。
- 10-03 为什么抽象类型如此重要?
- 10-04 你认为有必要引入 mixin 或 trait 类型吗?
- 10-05 区分接口与抽象类的意义何在?
- 10-06 你常有往接口中放置代码的冲动吗?
- 10-07 如何理解文中"多态使得公民拥有社会身份"这句话?
- 10-08 "针对接口编程"与"公民之间以社会身份互相交流"有何相似之处?
- 10-09 你是如何理解 OOP 中抽象、封装、继承和多态的?
- 10-10 每当一项新技术出现时, 你通常抱什么态度?
- 10-11 你会在编程中对某些语法上的限制感到恼火吗?
- 10-12 在理解或比较一些编程概念时, 你是更习惯从定义和语法的角度, 还是更习惯从本源和语义的角度?
- 10-13 本课与前课均提到了编程与武术相通之处:攻守兼备,动静得宜,刚柔 并济,虚实结合。对此你有何心得体会?