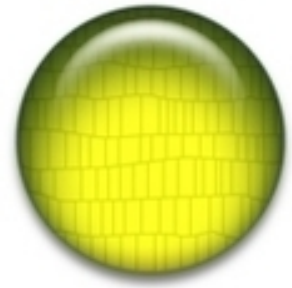

《编程机制探析》初稿共28章 20111019生成

作者: buaawhl http://buaawhl.iteye.com

由于发布时间问题，第十三章和第十四章的顺序倒了。作者邮箱。
buaawhl@gmail.com

目 录

1. 编程机制探析（Insight into Programming Mechanism）

1.1 《编程机制探析》初稿目录（已提供PDF下载）

发表时间: 2011-08-29 关键字: 编程, 设计模式, pattern

《编程机制探析》初稿目录

本书初稿发布在网上的目的如下：

1. 回馈

本书中的很多思路都是在论坛讨论和网络资源研讨的过程中形成的。因此，作为其产物，应该有所回馈。

本书已经做成PDF，只是由于发布时间问题，章节排布有点小问题，第十四章排到了最后一章。
电子书PDF版请在如下地址下载：
http://buaawhl.iteye.com/blog/pdf

2. 排错
限于水平，书中错漏之处在所难免，说不定还不少。请大家不吝指正。
3. 改进
本书在网上发布之后，很可能会得到更多的真知灼见，可以让我学到更多。我可以在内容上、语言上进一步改进。
4. 宣传
这是我写的第一本书，真正意义上的，自己写的一本书。写的都是我自己想写的内容。我希望能有更多的人读到这本书。我希望这本书能够帮助到更多的人。我希望有更多的人喜欢这本书。

每一章对应的链接内容分别如下（不断填入中）：

第一章 写作初衷——若是当年早知道.....
Blog链接：
http://buaawhl.iteye.com/blog/1160333
论坛链接：
http://www.iteye.com/topic/1114090

第二章 计算机语言
Blog链接：
http://buaawhl.iteye.com/blog/1160334
论坛链接：
http://www.iteye.com/topic/1114091

第三章 计算机运行结构
Blog链接：
http://buaawhl.iteye.com/blog/1160348
论坛链接：
http://www.iteye.com/topic/1114092

第四章 运行栈与内存寻址
Blog链接：
http://buaawhl.iteye.com/blog/1160352

论坛链接：
http://www.iteye.com/topic/1114157

第五章 命令式编程
Blog链接：
http://buaawhl.iteye.com/blog/1160361
论坛链接：
http://www.iteye.com/topic/1114087

第六章 面向对象
Blog链接：
http://buaawhl.iteye.com/blog/1160363
论坛链接：
http://www.iteye.com/topic/1114088

第七章 设计模式
Blog链接：
http://buaawhl.iteye.com/blog/1160392
论坛链接：
http://www.iteye.com/topic/1114093

第八章 Compositor Pattern
Blog链接：
http://buaawhl.iteye.com/blog/1160398
论坛链接：
http://www.iteye.com/topic/1114095

第九章 线程
Blog链接：
http://buaawhl.iteye.com/blog/1160400
论坛链接：
http://www.iteye.com/topic/1114096

第十章 线程同步模型
Blog链接：
http://buaawhl.iteye.com/blog/1160403
论坛链接：

http://www.iteye.com/topic/1114097

第十一章 Copy on Write[/url]
Blog链接：
http://buaawhl.iteye.com/blog/1160405
论坛链接：
http://www.iteye.com/topic/1114098

第十二章 Iterator Pattern
Blog链接：
http://buaawhl.iteye.com/blog/1160407
论坛链接：
http://www.iteye.com/topic/1114099

第十三章 动态类型
Blog链接：
http://buaawhl.iteye.com/blog/1160417
论坛链接：
http://www.iteye.com/topic/1114100

第十四章 关于方法表的那些事
Blog链接：
http://buaawhl.iteye.com/blog/1160464
论坛链接：
http://www.iteye.com/topic/1114111

第十五章 递归
Blog链接：
http://buaawhl.iteye.com/blog/1160420
论坛链接：
http://www.iteye.com/topic/1114101

第十六章 树形递归
Blog链接：
http://buaawhl.iteye.com/blog/1160422
论坛链接：
http://www.iteye.com/topic/1114102

第十七章 函数式编程
Blog链接：
http://buaawhl.iteye.com/blog/1160429
论坛链接：
http://www.iteye.com/topic/1114103

第十八章 函数式语法
Blog链接：
http://buaawhl.iteye.com/blog/1160432
论坛链接：
http://www.iteye.com/topic/1114104

第十九章 函数 = 数据 = 类型 ？
Blog链接：
http://buaawhl.iteye.com/blog/1160435
论坛链接：
http://www.iteye.com/topic/1114105

第二十章 流程控制
Blog链接：
http://buaawhl.iteye.com/blog/1160444
论坛链接：
http://www.iteye.com/topic/1114106

第二十一章 AOP
Blog链接：
http://buaawhl.iteye.com/blog/1160447
论坛链接：
http://www.iteye.com/topic/1114108

第二十二章 互联网应用
Blog链接：
http://buaawhl.iteye.com/blog/1199259
论坛链接：
http://www.iteye.com/topic/1116708

第二十三章 HTTP
Blog链接：

http://buaawhl.iteye.com/blog/1199261
论坛链接：
http://www.iteye.com/topic/1116710

第二十四章 HTTP要点
Blog链接：
http://buaawhl.iteye.com/blog/1199263
论坛链接：
http://www.iteye.com/topic/1116711

第二十五章 Web开发架构
Blog链接：
http://buaawhl.iteye.com/blog/1199266
论坛链接：
http://www.iteye.com/topic/1116712

第二十六章 页面生成技术
Blog链接：
http://buaawhl.iteye.com/blog/1199271
论坛链接：
http://www.iteye.com/topic/1116713

第二十七章 Flyweight
Blog链接：
[url] [/url]
论坛链接：
http://www.iteye.com/topic/1116761

第二十八章 ORM
Blog链接：
[url] [/url]
论坛链接：
http://www.iteye.com/topic/1116762

1.2 《编程机制探析》第一章 写作初衷——若是当年早知道.....

发表时间: 2011-08-29

《编程机制探析》第一章 写作初衷——若是当年早知道.....

小时候，我读过一部短篇小说《一块牛排》，美国著名小说家杰克.伦敦写的。
故事中，一个年老体衰的老拳击手陷入了人生的低谷，参加比赛之前，连补充体力的一块牛排都买不起。他的对手是一个正处于体力上升期、精力充沛的年轻拳击手。他凭借着丰富的经验与对手周旋，并抓住一闪即逝的机会，给了年轻的拳击手以沉重打击。但是，遗憾的是，他的体力不足，并没有给对手造成致命打击。对手最终重新稳住了脚跟，将他击倒在地。
很多年过去了，但这篇小说给我的印象一直没有褪色，尤其是文中那一段关于老拳击手心理活动的描写：“当他坐在自己的一角打量对手的时候,他的脑子里涌现出一个想法。如果以他的老谋深算,再加上桑德尔那样的年轻力壮,定能成为一名重量级的世界冠军,一代拳王。可是困难就在这里,桑德尔决不可能无敌于天下,因为他缺少智慧.而获取智慧的唯一途径,就是拿青春去交换。不过这样一来,等他有了智慧的时候,青春已经消失了。”
这段话，我可以说是感同身受。我也年轻过，我也踌躇满志过，我也精力充沛过，我也废寝忘食过。然后，同那个老拳击手一样，等我年纪渐长后，回忆起当年的无知无觉，我也满腔遗憾，悔不当初。
若是当年的我，就拥有了现在的见识和经验，那么，我当年绝不会走那么多弯路，也绝不会浪费那么多时间。
如今，很多事情虽然知道了，却为时已晚。正如有个一句话笑话讲的那样：我想早恋。可是，已经晚了。
正如那个老拳击手一样，每个人都有一些大大小小的“早知如此，就应如何如何”的遗憾。细数当年，我也有过不少这样的遗憾。
我现在还清楚地记得，毕业求职的时候，我去一家外企面试，由于英语口语不好，没有通过。只有在那时候，我才意识到了英语口语的重要性，我才明白，那些早晨起来在校园里坚持大声背诵李阳口语的同学们是多么的明智。
对于见多识广的同学来说，英语口语在求职中的重要性，只是一个简单的常识。而我对英语的认识却远远没有深刻到这个程度。我只是在积累词汇量和扩大阅读量两个方面投入了精力，以便查阅大量的英文技术文档。
那件事情是一个知识决定命运的活生生的例子，给了我深刻的教训，以至于我到现在还不能忘却。
这个世界上有一些人，意志坚韧，能力出众，做什么事情都能干出一番名堂来。那些人可能很难理解我的这种想法。在他们看来，一份职业只不过是一个起点而已，能做到什么地步，完全依靠个人的能力。
但很不幸的是，我是一个资质平庸、意志薄弱的人。第一份职业基本上决定了一个人职业起点的高低。与其努力工作，不如努力找工作。对于像我这样的资质平庸的人来说，尤其如此。
人在失败的时候，出于自我心理保护的机制，总是会为自己找借口。吾非圣贤，自然也是如此。很快，我就为自己找到了诸多借口。
首先，我没有意识到口语重要，是因为信息不对称。我是农村里长大的孩子，自然不如城里人懂得多。
其次，我们农村孩子从小学的都是哑巴英语，高考时听力部分就吃了大亏，幸亏没有考口语。
再次，这种基本教育环境的落后，不仅表现在英语方面，还表现在各个方面。比如，双语家庭里长大的孩子，从小就懂双语，根本就不用学。生意人家庭里长大的孩子，从小就懂生意经，自然比我们起点高。
再说了，中国人为什么要学英语啊？！都说中文，不就没这回事了？

总而言之，这都是环境的错，和我一点关系都没有。

当然，以上种种借口，除了给自己带来心灵上的抚慰，对于物质世界却没有任何改变作用。要想在物质世界占有一席之地，还得从改变自身做起。

既然信息不对称，那么就想办法弥补。我意识到求职宝典之类的重要性，也开始有意无意地留意网上的相关资料。

我在网上看到过一个微软求职的例子。有一个人，从来没有接触过编程工作，却敢于去微软面试。第一次自然被刷下来了。但他不屈不挠，还准备下次再来。主考官见状，对其产生兴趣，决定给他这个机会。多次面试之后，他从一个门外汉成为了一个半专业人士，展示了强大的学习能力，通过了考核，成为了微软的一员。

在这个例子中，那个人的学习能力和意志力起了关键作用，同时，那个主考官的耐心、闲心、宽容心也起了重大作用。

无独有偶，后来我在网上又看到了一篇帖子——“我为这份工作准备了十年”。作者在文中描述了他为了一份工作，积累各方面经验长达十年的过程。隔行如隔山，那个帖子描述的工种我不太懂，好像也是专业性很强的那种。

网上有些风言风语嘲笑作者，说什么有这份心思和功夫，还不如做生意或开公司，说不定早就发财了。对此，我的看法是这样的。我羡慕有钱人，但我敬佩专业人士。

前一个微软求职的帖子，我还没有太强的感觉，但是，看了这篇帖子，作者那种孜孜以求的精神，却令我敬佩不已，汗颜不已。

我从来就没有过这种感觉，能够对一份职业，或者说一个公司，能够有这么强烈的向往和执着的精神。

我如果能够拥有这样的精神和毅力，我也能成功获得一份向往的工作。但我没有。我从来没有对某个职位产生过如此强烈的感情。

我开始反思自身。难道我对计算机技术不够热爱？也许吧。我看到过一些少年黑客的故事。有些还在上高中的少年，就已经成为深谙计算机底层的黑客级别的编程高手。他们拥有天赋、兴趣、热情，缺一不可。与他们相比，天赋就不用说了， 太打击自信。就说兴趣和热情，我也远远不能与他们相比。当年求学的时候，我选择计算机专业，完全是因为这行热门，好找工作。

你能指望一个出身农村普通劳动者家庭的年轻人能够怀有“为中华崛起而学习计算机”的远大理想吗？

这样的人确实有。比如，“计算机要从娃娃抓起”宣传画上邓爷爷手抚的那个娃娃。

现实中，我也遇到那样的同学和同事，对于计算机抱有真正的热忱和热爱。但是，那类人屈指可数，凤毛麟角。大部分人都是像我这样，只是为了谋生和“钱景”。

说实话，编码工作整天与机器和代码打交道，确实是比较枯燥的。不过，编程工作也可能是有趣的。我正是在学习和工作的过程中，时而遇到智力上的挑战，甚至体会到创造的快乐，才真正对编程工作本身产生了兴趣和热情。遗憾的是，随着软件业的发展，程序员的工种进一步细化，产生了大量代码工人的职位，不仅减少了编码工作的趣味，而且加重了程序员的职业危机感。

俗话说，有竞争才有动力。程序员原本是一种脑力劳动，但是，随着经过简单培训的代码工人的大量涌入，编程越来越有沦为体力劳动的趋势。

本来自以为老资格的入行早的程序员，面对精力充沛的年轻一代和日新月异的新技术，越来越感到力不从心，不禁发出了和那个老拳击手一样的感慨——这是个吃青春饭的行业，这是年轻人的天下。

程序员是不是吃青春饭的行业。这是程序员群体中一直争论不休、经久不衰的永恒话题。之所以说这是个永恒话题，是因为这个话题的争论，永远不会有定论。

一方面，有不少人叫嚣着，程序员三十岁之前一定要转管理和行政，否则前途堪忧。想想老拳击手面对精力充沛的新生代的窘境吧。体力、精力、学习能力都不如人，只能吃老本。

另一方面，有不少三四十岁甚至年龄更大的资深程序员现身说法，并举例说，在国外（这里的国外，自然是指那些发达国家，其他的欠发达国家，很少有人会在意），十年经验以上的程序员才是一个软件公司真正的中坚力量。

这两种说法都有道理，各有其适用范围。但是，无论持何种看法的人，都同意这一点：程序员是一个危机感十分严重的职业，每天都在担心自己会被淘汰。

当然，职业危机感并非程序员这个职业独有，而是这个时代的象征。有一个流行词语叫做充电。每个人都在忙着上夜大、进修、考研、考证、考职称，生怕被时代大潮所抛下。

各类资格证书，学业证明，能够给人以极大的安全感。似乎手中有了证，就有了将来兑换成货币的保证。

从前，我们说，手中有粮，心中不慌。现在，我们说，手中有证，心中不慌。

程序员这个行业也不例外。为了满足软件从业人员考证的需求，各大公司机构提供了多种多样的认证以及相关考试，分门别类，应有尽有。

程序员是否应该考证？这也是在程序员群体中引起过争论的一个话题。

当时有牛人放出这样的豪言：真正有能力的程序员是从来不考证的。考证是缺乏自信心的表现。认证是一个程序员一生的污点！

对于这种说法，我是举双手双脚赞成的。因为我没有考过任何认证。当然，真正的原因是因为考证费用太贵，而并非我过于自信于自己的能力。

事实上，和所有浮躁的年轻人一样，我也是相当缺乏自信的。同所有年轻的程序员一样，我千方百计、呕心沥血地寻找提高技术能力的方法。购买技术书籍，阅读技术文档，阅读开源代码，访问技术论坛，追踪最新技术，等等，这些大家都干过的事情，我也通通干过。

现在回顾过去，不禁感慨万千。当年的我，就像一只无头苍蝇一般，四处乱撞，极度想找到出路，却四处碰壁。我一直想多积累一些原理性方面的知识和经验，却不得其门而入，徒然在外围的资料性知识上浪费了太多的时间和精力。

多年以后，我终于摸到了一点门径，却已经青春不再。当然，我还是比小说《一块牛排》里面的那个老拳击手幸运一点。程序员这个职业归根结底还是和智力相关的。

体力会随着年龄的增长而逐年下降，但智力不会。或者说，不全会。人的智力分为两部分。一部分叫做流体智慧，指那些会随着年龄增长而消退的能力，比如，反应能力，机械记忆力，等。一部分叫做晶体智慧，这那些不会随年龄增长而消退的能力，比如，逻辑思维能力，理解能力，等。活到老，学到老，指并不只是一句励志空话，而是有一定心理学依据的。

这本书主要讲述的都是这些年来我个人积累的一些计算机编程方面的原理方面的知识，希望能够对程序员朋友们有所助益。另外，我有很多亲戚朋友，没有接触过编程，对于计算机的了解也仅限于日常使用。他们觉得计算机程序挺神秘的。我希望，这本书能够对于他们这些非计算机相关行业人员也能够有所帮助。

换句话说，我希望，这本书能够起到普及计算机知识、宣传计算机文化的作用。

本书遵循这样一个原则，文字类比尽量多，编程代码尽量少。这样做的目的有两个。

一是为了明晰易懂，即使没有编程经验的人也能够从本书受益。

二是为了延长本书的生命力。编程语言是有生命期限的，说不定什么时候就会过时，被扫入历史的垃圾堆。一门编程语言的风光时间一般也就是十年左右。而自然语言（即汉语、英语等语言）相对于人类的寿命来说，可

以说是永不过时的。这本书的内容，都是我个人认为在几十年内都不会过时的重要编程模式。我不希望本书的生命力受到某种编程语言代码的生命周期影响。

这句话说得有点大了。所以，这里，我需要适时地惶恐一下。不，应该说是惶惑。

其实，早在几年前，我就想写这本书了。那时候，我花费偌大心力弄懂一些重要的编程概念或者编程模型，一种自得自满的心理油然而生：“原来是这么回事啊。以前看的那些书籍资料上写的根本就不对嘛，简直就是误导。我估计，那些作者可能自己都没有理解透彻，就半懂不懂地写出来了。若是我来写，我一定会写得更加清晰易懂。”

基本概念弄清楚之后，就等于打通了瓶颈。以前看不懂的那些专业论文之类的资料也大致能够看懂了，对于问题的理解也会随之加深。这时候，我才明白，我之前的自以为透彻的理解是多么粗浅和无知。与那些半通不通的书籍资料相比，我也只不过是五十步笑百步而已。我深受打击，不敢再动笔，生怕贻笑大方。

我总想着，再多积累一些知识，写得更有深度一些。但随着时间的推移，我发现，我接触到的未知领域越来越多，已经远远超出了我的智力限度。再这样下去，这本书可能永远也写不出来了。我甚至有了放弃这本书的想法。但是，我想写一本编程书籍的事情，早已为人所知（唉，那时候年少轻狂，嘴上没把门的）。一些亲戚朋友还关心地问我进度如何。每次都弄得我很尴尬。最后，我下定了决心，还是把这本书写出来算了，也算给自己一个交待。

可以说，这本书是我技术生涯的重要总结。书中内容都是我认为的“所有人都应该早知道，但很多人却不知道，或者说，不清楚的重要编程概念”。我不敢保证这本书有多么好。我只能保证，我写的东西，都是别人没写到的内容，或者说，没写到这种深度的。另外，我会竭尽所能，保证这本书的趣味性和生动性。

我现在还记得，我曾经读过的一些计算机教科书是多么的面目可憎，枯燥乏味。如果我写的书给读者这种感觉，那我还不如找一块豆腐，一头.....不，一下扔到锅里煮了吃了算了。

人生苦短，及时行乐。寓教于乐，才是王道。

好了，以上就是序章部分。按理来说，序言写到这里也就结束了。但我不。关于“要是当年早知道”这个话题，我还有很多话要说。

要是当年早知道口语的重要性，我就每天早晨去大喊疯狂英语了。

要是当年早知道这些编程原理知识，我早就应该选定好方向，一个独创的软件作品说不定已经成型了。

要是当年早知道那些创业期公司会上市，我早就应该加入进去，现在都可以退休享受生活了。

要是当年早知道股票会疯涨，我借钱也要去买，现在不知道翻了几千倍了。

要是当年早知道房价会狂飙，我砸锅卖铁也要凑几套啊，现在也至少翻了几番了。

要是当年早知道彩票是那个号码.....

要是当年早知道那年雪灾滑雪衫会热卖.....

要是当年早知道.....

每个人可能都会像我一样，因为错失了各种各样的大好良机，而满怀大大小小的遗憾。

为了弥补这种遗憾，网上悄然兴起一种“重生”文学。中心内容是，一个人突然回到了多年前，有了重活一次的机会，可以弥补当年的种种缺憾。

这类小说，引不起我的阅读兴趣，但却引发了我关于职业生涯的一些思考。如果人生真的可以重来的话，我还会选择程序员这个职业了吗？

这个问题，我思考了很久。最终答案是：会。因为那个时候，我别无选择。

我在网上看过一个在美国的中国留学生写的文章。在美国学校里，经常有些欧美同学很好奇地问那个中国留学

生：“你们中国人为什么都喜欢学计算机呢？”那些欧美同学的专业各异，有哲学，有考古学，都是各人感兴趣的科目。

面对这个问题，那个中国留学生通常无言以对。他只能在心里呐喊：“你以为我想学计算机吗？我们中国人不学计算机，能找到工作吗？”

好找工作，是我当年求学报专业时的唯一考量。当然，计算机专业并非唯一的好找工作的专业。就在报志愿前，我遇到了一个智者。当然，那个智者也不过是农村里一个普通的中年体力劳动者，满面沧桑和风霜，生活的重担在他身上留下了深刻的印记，却并没有磨灭他乐观外向的性格。他喜欢闲谈吹牛，和我侃了好久。他建议我学医，因为医生是任何时代都少不了的职业，而且，医生是一种越有经验越值钱、越老越值钱的职业。

我当时还开了一个玩笑，说，医生这个职业还是要退休的。这个世界上，唯一没有退休年龄限制的职业只有两种，一种是总裁，一种是总统。我还讲了一个关于美国总统里根的笑话。

甲问：里根为什么年纪那么大了，还要去竞选总统。

乙答：以他的年龄，那是他唯一能够找到的工作了。

那人听了，哈哈一笑，谈兴更浓，又说起了几种保值的职业，即越有经验越值钱、越老越值钱的职业，比如，教师，技工，以及其他利于积累资格和经验的职业。

那番谈话给我留下了很深的印象。我也确实慎重考虑过他的建议。但是，出于急功近利的心理，我最终还是选择了有吃青春饭嫌疑的计算机专业。

现在看来，那真是一种短视的行为。幸亏我从这个职业中获得了智力上的乐趣，还不至于后悔当年的选择。

市面上曾经流行过一本书（也许现在还在流行？）——《穷爸爸，富爸爸》。

那是一本关于理财、创业、投资的书，同时还讲述了一些人生哲理和生活原则。

该书的中心思想是：不要为钱工作，要让钱为你工作。

这句话是废话，地球人都知道，只是做不到，我们不去理会它。

该书的另一个重要观点是鼓励人们创业开公司。因为开公司是先花钱，再根据剩下的钱交税。而拿工资是先交税，再花剩下的钱。

这条信息对我挺有用的。但是，对于很多人来说，可能只是简单的常识。我们也不去理会它。

我们来看该书表达的第三个观点：当你需要为一件事情长期投入时间和精力的时候，你最好想到几十年后你年老时的情景。

书中举了一个例子。作者年轻的时候，想在网球和高尔夫球这两项运动中选择一项，作为自己的主要业余运动。

这里说明一下，在地广人稀的国家，高尔夫球场并非稀缺资源，高尔夫球也不是多么贵族化的运动，而是一种比较常见、花费不高的平民运动。我就见过，一些十几岁的小孩子，连高尔夫球车的钱都不愿意出，自己拖着高尔夫球袋，一杆一杆，在偌大的球场上长途跋涉。如果在寸土寸金的地方，这样的小孩子也许只有当球童，才有机会进入高尔夫球场吧。

作为一个年轻人，作者想选择对抗更激烈、运动量更大的网球。他的父亲劝他，“网球和高尔夫球都是需要投入大量时间训练才能取得技巧的运动。你选了一项，就势必要放弃另一项。你想一下，当你年老的时候，你就打不动网球了。那时候，如果你再学高尔夫球，那么获得的成就会很有限，你获得的快乐也会很有限。”

作者经过深思熟虑后，“明智”地选择了高尔夫球，而他的同龄朋友（似乎就是那个“穷爸爸”的儿子）择选择了网球。几十年后，作者证明了自己选择的正确：他的朋友打不动网球了，再学高尔夫球也错过了最佳年龄阶段，而作者已经是一个高尔夫球高手了。

这个例子对我的触动很大。我这么说，并非我认为高尔夫球就一定比网球明智——那个选择网球的同龄朋友很可能有自己的想法，比如，网球更受青春期女孩子的欢迎，更容易泡到妞，等等。这个理由对于青春期的男孩子来说，是压倒一切的。

这个例子之所以触动我，是其中所表现的一种原则——当你决定投入相当长、相当多的时间去做一件事情的时候，最好能够想到几十年后年老的情景。

也许，在你选择一项娱乐或者运动的时候，这个原则的重要性还不是那么明显。但是，当你选择一项职业的时候，这个原则的重要性就不言而喻了。

如果按照这个原则来看，小说《一块牛排》里描述的那个老拳击手显然选错了职业。所有的运动员行业都是吃青春饭的，尤其是高强度体力运动。选择这类职业，一定要提前选择好退路，要么功成身退，嫁入豪门，要么向教练转型。

俗话说，女怕嫁错郎，男怕入错行。正如婚姻对女人的重要性如同第二次投胎一样，职业对男人的重要性也如同第二次重生。

这话很俗，很糙，很不入耳，但是，这却是一条普遍适用于社会中大部分人的通行法则。只有少部分意志强大、能力出众的人才能够摆脱这条法则的影响。有能力的人，在哪一行都能干出个名堂来，干什么行业都能成为行业翘楚。

那些人可以很牛地说：没有不行的行业，只有不行的人！

但我不是那样的人。那样的人能够压倒一切，而我，只会被一切压倒。

所以，对大部分人都起作用的原则，对我也一定会起作用，而且很可能会特别起作用。

按照《穷爸爸，富爸爸》里面的原则来看，我显然是选错了行业。这让我感到很安慰：原来，我之所以没有“成功”，不是因为我懒，而是因为我傻。

雷锋说：有人说我傻，但是，革命需要这样的傻子，我甘愿做这样的傻子。

我说：各行各业，总得有人来做。就让我做这样的傻子吧。

最后，让我吐露一下心中的奢望。

我希望，本书能够为读者提供一套受益终生的思维体操方案。

我希望，本书能够帮助更多的人对程序员这个职业产生兴趣。

我希望，本书能够帮助程序员这个职业成为一种长期的、创造性的、充满乐趣和智力挑战的终身制职业。

1.3 《编程机制探析》第二章 计算机语言

发表时间: 2011-08-29 关键字: 编程

《编程机制探析》第二章 计算机语言

关于外语的重要性，怎么强调也不过分。
关键时刻，外语甚至能救命。
我们先来看一则小故事。
老鼠妈妈带着孩子拼命奔逃，一只猫在后面紧追不舍。
跑着跑着，老鼠母子俩被逼到了一个死角中，无处可逃，眼看就要命丧猫口。
这时，老鼠妈妈临危不惧，直面大猫，狂吠几声：“汪！汪！汪！”
那只猫吓了一跳，转身就跑。
老鼠妈妈松了一口气，抓紧机会，对惊魂未定的孩子教育道：“看见没？学好一门外语是多么的重要啊！”
计算机语言实则也是一门外语。掌握这门外语也非常的重要。掌握了计算机语言，就等于掌握了与计算机交流的渠道。
人类发明了各种工具，作为自己身体的延伸。交通工具是脚的延伸，操作工具是手的延伸，而计算机，则是大脑的延伸。
计算机在中文里有一种十分形象化的翻译——电脑。由于这种翻译形象生动，言简意赅，很快就流行起来，成为大众化的称呼。
不过，业内人士对电脑这种叫法却抱有一种略带不屑的心理，感觉那是非专业的老百姓的叫法。业内人士只把计算机准确地叫做计算机。计算机的英文是Computer，就是计算器、计算工具的意思。
一些更专业的人士，甚至直接把计算机叫做机器。在他们的眼里，计算机只不过是机器的一种，和其他种类的机器没有什么本质上的区别。这种叫法体现了作为人类的专业人士的一切尽在掌握的优越感。而电脑这种称呼则无形中把计算机提高到与人类大脑平齐的程度，体现了老百姓对计算机这种神秘的、先进的事物的尊敬和崇拜感。所以，如果你需要昭显自己的专业程度，最好把计算机叫做计算机，或者直接叫做机器。
不过，我个人十分喜欢电脑这种形象而浪漫的称呼。计算机、机器这些称呼过于机械化，无形中为这个行业中多增加了一分枯燥和呆板。
当然，为了准确起见，本书还是会使用计算机这个名词。毕竟，虽然计算机能够做的工作越来越多，但本质上做的还是计算工作，还不能代替人类思考，达不到“脑”的程度。也许将来某一天，随着人工智能的发展，计算机能够真正地成为“电脑”。
人类发明的现有工具中，有哪些与计算机比较类似呢？首先跃入脑海的自然是计算工具，比如，算盘，计算器，等。
没错，顾名思义，计算机本质上做的就是计算工作。不过，计算机的计算工作方式与其它计算工具却大相径庭。
传统的计算工具，比如，算盘，计算器，其工作方式就如同懒蛤蟆一样，按一下，跳一下。每一个计算步骤，都需要人的参与，才能继续下去。
计算机完全不同。计算机的优越之处在于奇快无比（当然是和人类相比）的计算速度，一秒钟之内能够进行成

千上万、甚至上亿次运算，远远超过了人类的反应速度。如果计算机的每一步计算都需要人的参与，那么必将极大拖慢计算机的速度，极大浪费计算机的工作潜力。

计算机的这种特性决定了计算机的工作方式：先由工作人员（即程序员）事先编制好计算过程（即计算机程序），然后，把计算过程传给计算机，并让计算机按照计算过程执行；然后，就是计算机的活了；马儿，你快些跑哎，快些跑！计算机就飞快地运转起来，以瞬息万千的速度执行着计算程序。

最早的计算机概念模型——图灵机，就是这样一个机器：一方面，它接受一条输入纸带，上面印满了计算过程；另一方面，它吐出一条纸带，上面写满了计算结果。

上图为“图灵机的艺术表示”，源自于互联网自由共享资源（作者Schadel授权任何人为任何目的使用这项作品）。

注：这里说明一下，为了增强本书的趣味性，我会在书中引用一些生动有趣的图片。为了表达对作者的尊重，并避免版权问题，我会尽量使用公共授权的自由共享图片，并给出作者信息。

如果你曾经见过八音盒的内部构造的话，你就不会对这种结构感到陌生。八音盒是一种机械播放音乐的装置，可以看作是一种固化版的、不需要用电的留声机。

我小时候没有见过八音盒，甚至不知道这种东西的存在。长大之后，我才见到了八音盒。那么一个小小的装置可以发出连续清脆的音乐，令我感到很奇妙。我很想弄懂它的工作原理。恰好，不久之后，我就见到了一个被拆开的八音盒。其内部工作原理立刻昭然若揭了。八音盒的发音装置很简单，就是一个小金属滚筒，加一排细细的金属簧片。金属滚筒上布满了小小的凸起。

当金属滚筒在发条的带动下，开始转动的时候，那些小小的凸起就会拨动和金属滚筒挨在一起的那排细细的金属簧片，就像一个人在弹钢琴一样，只不过乐谱早已固化在金属滚筒上了。那些小小的凸起，实际上就是一部乐谱，其位置排列都是按照乐谱来设定的。

有一个叫做akko golenbeld的艺术家，根据八音盒的原理做出了一个叫做“城市音乐”（CityMusic）的巨型八音盒：一个巨大的木桶上雕刻了一座城市的浮雕，旁边挨着一个钢琴；当木桶转动的时候，就会拨动琴弦，奏出这个城市的音乐。

在akko golenbeld的网站，http://akkogoldenbeld.com/，你可以看到一段完整的动画，演示了这个作品的整个工作原理。

从外部观察到的工作原理上来说，图灵机模型和八音盒的工作机制是一样的。八音盒滚筒上的按照乐谱编制的

凸起排列，就相当于图灵机模型中的布满了计算过程指令的输入纸带；八音盒播放出来的音乐，就相当于图灵机模型的输出计算结果的纸带。

当然，两者的区别还是很大的。八音盒构造极为简单，只是一个简单的发音装置。图灵机的内部计算部件则极为复杂。两者不可同日而语。两者只是在“事先编制”、“按章办事”这方面比较相似。

讲到这里，先说两句关于图灵的题外话。图灵机这个名字来自于英国著名的数学家艾兰•图灵。图灵机这个模型就是图灵提出来的，并因此而得名。计算机的最高奖项——图灵奖，也因此而得名。可以说，图灵在计算机界的地位，无人可以撼动。但是，图灵的一生，却在悲剧中结束。

1952年，图灵的同性恋倾向曝光，并因此而被判有罪——“有伤风化罪”（根据英国当时的刑法，从此案例中可以窥见当时所谓文明国家的文明程度），被迫接受了“化学阉割”——即女性荷尔蒙注射，消退性欲。此种刑法在不少欧美国家和地区实行。据说，此法能够有效遏制性犯罪。

两年后，图灵服用毒苹果自杀身亡。据说，其灵感来自于迪斯尼动画片《白雪公主》中的一句歌词——“毒液浸透苹果如睡之死渗入”。有一种说法，苹果公司的标志——被咬了一口的苹果——就是为了纪念图灵的死亡。

看了图灵的往事，令人不得不想到另一位同样性倾向的跳楼自杀的香港华人巨星——张国荣。不过，时过境迁，两者境遇已经完全不同。张国荣并非因为社会压力而自杀，而是因为不堪病痛折磨而选择结束了自己的生命。生在不同时代的两个名人，同样是同性恋，同样是日不落帝国英联邦势力辐射下的地区，虽然终点相似，人生遭遇却完全不同，令人不由心生唏嘘感叹。

好了，题外话就说到这里，我们言归正传，回到计算机语言的话题上。

首先，我们需要考虑的问题是，计算机为什么需要一门语言？或者说，计算机凭什么需要一门语言？

人类自诩为万物之灵长，一向自以为优越于其他物种，其最为自豪的能力之一就是语言能力。

而计算机，不过是人类发明出来的一种计算工具，它凭什么可以拥有一门语言？

其他的计算工具，比如算盘、计算器，从来都是人类拨一下，动一下，从没有奢望拥有什么语言。计算机为什么就这么特殊呢？

计算机确实就是这么特殊。前面说了，计算机的最为强大的能力，就是奇快无比的运算速度。为了最大发挥它的效能，不至于让人类的极低反应速度拖累它的功效，人类尽可能提前编制好计算过程，一次性地传给计算机，让它独立而不受干扰地运行。

这份提前编制好的“计算过程”是要交给计算机来执行的。计算机必须能够理解这份“计算过程”，才能够按章办事。这意味着，人类与计算机之间需要一种交流方式，换句话说，需要一种语言，这就是计算机语言。

计算机语言是人类设计出来的、由计算机理解并执行的一种语言。计算机虽然计算速度远超人类，但语言能力极其有限，只能理解最为基本的单词和语法。

计算机能够理解的单词，只有那么几个，基本上全是和本职工作相关的动词。其中，大部分都是关于计算的动词，比如，加、减、乘、除。这几个运算动词，由计算机的“脑部”执行。这些计算动词的意义和我们熟悉的四则运算基本一样，不难理解，不多做解释。

除此之外，计算机还有一个用“手”来执行的动词——存取。关于存取的内容比较容易理解，就是数据。但是，这个动作的关键在于来源地址和目标地址。比如，从哪里“存”到哪里？从哪里“取”到哪里？这个问题涉及到计算机的存储体系结构，比较复杂，我们后面慢慢阐述。现在，只要知道计算机有这么两个用“手”的动作用来存取数据就可以了。当然，这个“手”，只是计算机内部的手，只能在计算机内部伸来伸去，并不能伸到计算机的外部。

另外，计算机还有一个用“脚”的动作——跳转。同样，这个动作涉及到来源地址和目标地址的问题。比如，从哪里“跳”到哪里？这个问题同样涉及到计算机的存储体系结构，我们后面慢慢阐述。现在，我们只要知道计算机有这么一个用“脚”的动作就可以了。当然，这个“脚”，只是计算机内部的脚，只能在计算机内部跳来跳去，并不能跳到计算机的外部。计算机进行“跳转”动作的时候，并不是像我们人类跳槽一样，从一个公司跳到另一个公司，而是转换工作内容——放下手头现有的工作，去做另一样工作。

有了上述这些基本动词之后，我们就可以让计算机做事了。我们可以事先写好计算过程，比如，“先算这个，再算那个。先做这个，再做那个。跳到那里，再跳到这里……”

计算机接受到这些指令之后，就会一丝不苟地按章办事，乐此不疲，运转不停（如果不出错的话），直到运行到最后一条指令为止。

这种工作方式，很像是日本当年的产业工人的工作方式。日本有一种“手册工人”的说法，即一切按照手册办事，不需要思考，不需要质疑，只需要按章办事。这种工作方式确实极大地提高了日本企业的生产力。但是，这种把人类机械化的做法，必然会付出代价。日本企业已经认识到这一点，开始采用更加人性化的管理方式，不仅保护了工人的身心健康，还提高了工作效率和产品质量。

计算机是电子机械产品，没有情绪波动，没有思想活动，更没有心理健康的问题，承担“手册工人”的工作是再合适不过了。

但是，现实中总是充满了意外情况，“手册工人”也会碰到意外。这就像一个产业工人在安装螺丝的时候，恰好拿到了一颗坏螺丝，如果严格按章办事的话，就会把坏螺丝装上去。

这时候，就需要在工作流程上加上一条——如果螺丝坏了，那么换一个新螺丝，安装上去；如果螺丝没坏，安装上去。

这就需要产业工人拥有判断螺丝坏没坏的能力。同样，计算机也要拥有类似的判断能力。这就引入了一个新的计算机语指令——判断指令。

判断指令的引入，给计算机带来了基本的逻辑判断能力。当计算机遇到“坏螺丝”的情况时，会跳转到“换螺丝”的工作手册，进行“换螺丝”的工作流程，直到换到一个好螺丝，才会跳转到“安装螺丝”的工作手册。

好了，现在，我们已经凑齐了必备的计算机指令——加减乘除、存取、跳转、判断。

我们可以看到，这些词汇全都是动词，并没有名词，形容词，副词，助词，感叹词。这很好，很符合计算机的气质。计算机没有情感（至少现在还没有），计算机不会思考（至少现在还不会），计算机不需要读心情散文，也不需要读生活哲理，它只需要按章办事，它只需要看懂工作手册就行了。

计算机的字典里，全都是动词，而且全都是祈使动词，全都是人类发给它的命令——做这个，做那个，做十遍，等等。

用计算机语言编制的“计算过程”——即计算机程序——就是一篇充满了祈使命令语句的工作流程。这种以祈使命令为主题的计算机语言，也叫做命令式语言，从英文Imperative Language翻译而来。Imperative这个词的意思，就是命令的，强制的，祈使的。

现在就有一个问题了。既然专门有一个词语来定义计算机的命令式语言，难道计算机还有非命令式语言吗？答案是，没错，确实有其他类型的非命令式的计算机语言，但都是非主流，非主旋律，我们后面再讲到它们。现在，还是让我们回到计算机语言的最基本的词汇表上来。

前面讲了，计算机的字典中，词汇极其有限，主要就是那么几个动词——加减乘除、存取、跳转、判断。

与人类日常生活中使用的自然语言相比，计算机语言是如此的简单和贫乏，简直侮辱了“语言”这个词儿。如果外语都这么简单，那就太好了。

在我个人看来，计算机“语言”还根本不配称为一种“语言”，它还达不到语言的级别。
之所以称之为“语言”，主要是表达一种期待。
人们期望计算机某一天能够理解人类的自然语言，那时候，事情就简单了，人们可以像吩咐助手一样，用日常使用的自然语言吩咐计算机（那时候，应该叫做智能机器人了）做事，也不用编什么程序了。
好了，让我们回到现实。我们现在面对的现实是，计算机就那么一点可怜的理解能力，我们必须去理解它，而不是指望它来理解我们。当然，我们还是抱有一种奢望，尽可能地提高计算机的理解能力，让它尽量地理解我们。这就要具体分析计算机的理解能力了，看看它到底能够理解到什么层次。
首先，计算机从出厂开始，就可以理解最基本的词汇集——加减乘除、存取、跳转、判断。这些语言指令是固化在计算机硬件芯片里面的。如果没有软件支持的话，计算机的语言理解也就到这个层次了。这种语言叫做汇编语言，是计算机天生就能够理解并执行的语言，因此，也叫做计算机底层语言。
在计算机行业里，底层这个词，与社会底层这个词里的底层含义不同，甚至恰好相反。在计算机术语里，底层，意味着硬件层次。越底层，就越接近硬件层次，意味着专业程度越高、技术难度越高。
汇编语言，就是计算机不需要软件支持，仅凭本身硬件，能够直接理解并且执行的语言。因此，汇编语言可谓是最底层的语言。
这门语言为什么叫做汇编语言。说实话，我也不知道。这个词是从英文Assembly Language翻译过来的。Assembly这个词带有集成、装配的含义，意思是把一堆大大小小的零件装配在一起，集成一个整体。我们可以大致把汇编语言理解为一种最基本的配件级别的语言。
汇编语言的几个基本单词——加减乘除、存取、跳转、判断——可以直接交给计算机理解并执行，只是其中还要经过一个机器语言编码的过程。这种机器语言编码的原理和过程极为类似于电报收发时采用的“摩尔斯”电码。
原始的电报发报机只能发出极为有限的音节，比如，短音的嘀，长音的嘀。为了让发报机能够表达丰富多样的自然语言文字，人们必须给每个文字定义一个由一串短音嘀和长音嘀组成的编码。这就叫做“摩尔斯”电码。
同样，计算机能够识别的基本信号，也是极为有限的，只有“0”和“1”两种信号。那么，汇编语言中的每一个基本词汇，都必须指定一个由一串“0”和“1”组成的编码，计算机才能够识别并执行。
计算机有了汇编语言，就有了与人类交流的最根本的基础。人类可以通过汇编语言来编制各种各样的计算机程序，交给计算机执行。
但是，汇编语言太基础了，以至于基础到人们无法容忍的程度。
比如，如果想对计算机表达这么一个意思——“遇到前面的路口，向左转”，那么，你必须写一大串的基础语句来定义这个流程：
第一步，向前走。
第二步，判断是否遇到路口。
第三步，如果不是路口，跳转到第一步；如果是路口，继续向下执行第四步。
第四步，向左转。
这种写法实在太繁琐，太烦人了。为了解决这个问题，人们又创造出来了更高级的计算机语言，可以用更简单、更类似于自然语言的词汇来命令计算机做更多的事情。
所谓高级语言，只是比汇编语言高级一些，其实也高级不到哪里去，只不过是多加了像“如果...就...”、“重复”、“调用”这样的几个单词和语法结构而已，与人类的自然语言相比，还是简陋得要命。
不过，既然有汇编语言垫底，我们也姑且从众，称之为“高级”语言吧。

汇编语言是计算机硬件级别支持的，而高级语言通常是计算机软件级别支持的。

硬件和软件，这两个词前面也反复提到过。那么，究竟什么是硬件，什么是软件呢？这就涉及到计算机体系结构的问题了，这也是下一章我们要探讨的问题。

1.4 《编程机制探析》第三章 计算机运行结构

发表时间: 2011-08-29 关键字: 编程

《编程机制探析》第三章 计算机运行结构

前文反复提到计算机的“硬件”和“软件”这两个概念。那么，硬件和软件到底是什么东西呢？通俗来讲（即，用我们老百姓的话来讲），硬盘中存储的数据叫做软件，除此之外，计算机所有其他的部件，全都叫做硬件。注意，硬盘本身也是硬件。因此，软件是存储在硬件中的。软件必须依托于硬件，才有依托之地。

打一个不太恰当的比方，硬件相当于看得见、摸得着的物质文明，而软件就相当于看不见、摸不着的精神文明。

当然，上述说法只是直观的说法，不是专业的说法。那么，什么是专业的说法呢？专业的说法，必然要涉及到计算机体系结构。提到计算机体系结构，就不得不提到冯.诺依曼结构。

正如本书不会对图灵机展开详述一样，本书也不会对冯.诺依曼结构展开详述。当年上学时，课本中关于图灵机模型和冯.诺依曼结构的篇章可着实不少。但是，现在回头去看，那两种模型结构，更多的是一种纪念上的意义，加上一点理论基础上的溯源意义。所以，本书不会详述这两种模型结构。不过，总线结构却是不得不提的。

总线结构，是现代计算机仍然遵守的一种计算机体系结构。总线，是从英文“Bus”翻译过来的，本意是公共汽车的意思。我不明白，为什么要把这个词翻译成总线。也许是“公交总线”的意思？反正前辈们这么翻译了，咱们也就约定俗成，姑且这么用着。

总线（Bus）结构就像一条公交线路一样，所有的计算机部件全都挂在一条公交总线上，所有的数据交流都通过这条公交线路来进行。每个计算机部件就相当于这条公交总线上的一个公交站点，数据就相当于乘客，从各个公交站点乘坐公共汽车，上上下下，从一个站点到达另一个站点，即从一个硬件部件，到达另一个硬件部件。

在这个总线模型中，每一个公交站点都代表一个硬件部件，公交线路本身——即总线本身——也是一个硬件部件。那么，在现实中，总线对应计算机的哪个硬件呢？

用过台式机、并打开过机箱的读者一定会有印象，机箱中那些东西像公交线路呢？那一条条的宽宽扁扁的塑料数据线，看起来就像一条条公交线路，难道那些就是总线？

我以前就是这么以为的。但是，很遗憾，这个答案是不对的。那些扁扁的塑料线，并非总线，最多只能算是支线。那么，总线到底是什么呢？

急性的读者一定喊起来了，你废什么话呢？还不赶紧说！好吧，我说，总线，其实就是机箱最大的部件——主板，英文叫做“Mother Board”，因此也可以翻译为母板。

所有的其他硬件，包括极为重要的CPU（中央处理单元）、内存、网卡等，全都插在主板上，其他的重要部件，如硬盘、光驱等，也都是通过塑料数据线连接到主板上。主板，就是一个信息交流中心，垄断着所有的信息交换通路。任何两个硬件部件的信息交流，都在主板的掌握之下。

总线结构在信息交换速度和数量上是有瓶颈的。很多计算机科学家都在研究如何打破这个瓶颈。也许，将来的计算机就是非总线结构的了。不过，目前我们使用的计算机大部分还是总线结构，所以，我们还是要对它有一点了解。

读者如果想对计算机主板结构（即总线结构）了解更多的话，除了拆机直接察看之外，还可以用“鲁大师”等

免费硬件检测软件察看计算机硬件的具体参数。如果你需要自己安装系统中的各种硬件驱动程序的话，这类工具就很有用。

关于硬件部分，就说这么多了，本书还是主要关注于软件方面。前面讲了，软件就是硬盘上存储的数据。这话并不完整。因为，硬盘并不是唯一可以存储数据的地方，光盘、U盘等存储介质中，同样可以存储数据，自然也可以存放软件。只是为了简化问题起见，我们现在只考虑硬盘这个最重要的数据存储结构。

从广义的概念上讲，所有的数据都可以看做是软件。从狭义的概念上讲，只有可以运行的那部分数据（即计算机能够理解并执行的程序）才可以称为软件。不管从那个概念上来讲，软件都是数据的一种。

在硬盘上，所有的数据都以文件的形式存储。因此，我们通常也把硬盘等存储介质中存放的数据叫做文件。软件程序也不例外，也是以文件的形式存储在硬盘中。那么，这些程序文件到底是如何运行起来的呢？这里面涉及到相当多的基础概念，即使是计算机业内人士，也不见得能够把整个过程理得通顺。

比如我自己，从学校毕业出来之后，脑海里积攒了一堆什么“进程”、“线程”、“操作系统调度”之类的建立在理论模型上的词语，却不知其具体对应物。以为工作中自然会了解这些具体知识。但我发现，工作中更不需要理解这些知识，只要按照前人搭好的框架，往里面添砖加瓦就可以了。

我举一个非常切实的例子，在现实的很多互联网公司中，很多程序员（也包括以前的我），编了几年的互联网程序，却对互联网最重要的协议——HTTP——的基本原理一窍不通。这一方面说明了现代软件公司在软件开发分工管理方面的成功之处，一方面说明了程序员们本身对于基础知识并不重视。说实在的，如果只是谋一份饭碗的话，基础知识确实没太大用处，不知道也照样编程。

但是，如果你像我一样，什么事都想寻根问底，如果不了解原理心里就不踏实的话，本书对你来说就很有用了。

在我真正理解计算机内部运行机理之前，我心里总是感到不踏实，总觉得那里面有一种我没有完全了解的机制在起作用，说不定什么时候，它就会冒出来给我找麻烦。于是，我总是想弄明白它。后来，我把理论模型和现实的计算机部件结合了起来，心里立马就有底了，不再胡思乱想，最终获得了心灵的宁静。（我差点就用了“心灵的救赎”这个词。没办法，受知音体影响太大。）

现在，我们来审视硬盘上的程序文件，看看它是如何运行起来的。

硬盘上存储的一份份程序文件，从概念上来说，都是给计算机阅读的一份份工作流程。对于计算机来说，硬盘就是一个巨大的文件存储仓库，里面存放了很多数据文件，还有很多工作流程文件（即程序）。当计算机工作的时候，它首先要选择一份工作流程，然后按照这份工作流程来工作。在按章办事的过程中，它可能会根据流程规定，从硬盘这个文件存放仓库中取出另外的一份或者多份文件。

现在，有一个问题。当我们说计算机在工作的时候，这个“计算机”到底是指谁？是指包括显示器和机箱在内的整个计算机吗？宽泛来讲，是的。严格来讲，不是。

从严格意义上来讲，计算机中真正干活的硬件部件，只有一个。那就是中央处理单元CPU。只有CPU才有计算功能，只有CPU才有理解并执行计算机程序的能力。这也是为什么它叫做“中央”处理单元的缘故。舍它其谁？

注：在一些高性能显卡中，有GPU（图形处理单元）的配置，也带有了一点CPU的运算能力，我们可以把GPU看作是一种专门处理图形显示的特殊CPU。

CPU芯片的个头并不大，在主板上只占有一小块位置。但是，其地位却是最重要的。主板上专门有一个风扇为它扇风。生怕它太热太疲劳。没办法，谁让人家劳苦功高呢。整个机箱中，就它一个人在真正干活。

CPU有自己的工作台，学名叫做“寄存器”。但是，这个叫做“寄存器”的工作台太小，而硬盘那个文件存储

仓库中的每个文件都是那么的巨大，工作台上根本就放不下。于是，CPU就在工作台旁边放了一个巨大的木架，叫做“内存”。

CPU工作的时候，先把需要执行的工作流程从硬盘那个文件存储仓库中取出来，放到内存那个木架上。内存这个木架足够大，可以放得下巨大的文件。然后，CPU就把内存中的文件数据，一部分一部分地取出来，放在自己的“寄存器”工作台上，进行处理。这个动作叫做“取”，或者叫做“读取”。CPU做完手头工作之后，就把工作台上的工作结果放回到内存这个木架上。这个动作叫做“存”，或者叫做“写入”。

在CPU硬件直接支持的汇编语言中，存和取这对指令实际上是用一个数据传送指令来表达，只需要正确地定义源地址和目标地址，就可以正确地实现存取动作。

现在，我们就知道前面讲的“存取”这一对动作的含义了。取，是指从内存中取到寄存器中。存，是指从寄存器中存到内存中。

除了寄存器和内存之外，CPU也可以存取包括硬盘在内的各种存储空间。为了简化问题起见，我们目前可以暂且认为，CPU工作的时候有三个存取空间。第一个存取空间是CPU芯片内部的空间，叫做寄存器；第二个存取空间是插在主板上的内存；第三个存取空间是通过塑料数据线与主板相连的硬盘。

CPU存取这三个空间的速度，与这三个空间的距离成反比。寄存器就在CPU内部，自然是最快的。内存插在主板上，距离远一点，就慢一点。硬盘就更远了，还要通过一条线连接在主板上，存取速度最慢。

当我们说“存”、“取”、“读”、“写”的时候，我们都是以CPU的视点角度来说的。把数据从离CPU远的地方，传输到离CPU近的地方，就叫做“读”或者“取”。把数据从离CPU近的地方，传输到离CPU远的地方，就叫做“存”或者“写”。

比如，数据从寄存器传送到内存，叫做“存”或者“写”；数据从内存传到寄存器，叫做“读”或者“取”。

数据从寄存器传送到硬盘，叫做“存”或者“写”；数据从硬盘传到寄存器，叫做“读”或者“取”。

数据从内存传送到硬盘，叫做“存”或者“写”；数据从硬盘传到内存，叫做“读”或者“取”。

喜欢较真的读者一定会问，如果数据从寄存器传到寄存器呢？从内存传到内存呢？从硬盘传到硬盘呢？如果是这些情况，那怎么叫都行，或者干脆就叫数据传输。

需要注意的一点是，在CPU真正的工作过程中，所有的数据存取都要经过“寄存器”这个工作台。不管数据从哪儿到哪儿，“寄存器”都是必由之路。

如果我们需要把硬盘中的数据读入到内存中，CPU首先需要把数据从硬盘中读入到寄存器中，然后再写入到内存中。

如果我们需要把内存中的数据写入到硬盘中的时候，CPU首先需要先把数据从内存中读入道寄存器中，然后再写入到硬盘中。

当CPU把程序文件从硬盘中读取到内存中的时候（经由寄存器），这时候，程序就发生了质的变化——它不再是硬盘上僵化的数据，它获得了生命力，它成为了进程。这个唤醒的过程很奇妙，就像童话《睡美人》中的那个王子的神奇一吻一样，沉睡的公主一下子就醒了。

一个程序只有在进入到内存成为进程之后，它才能被CPU执行。进程的英文叫做Process，意思是处理流程。进程的概念在软件编程中极为重要。可以说，软件的整个编程模型就是建立在进程基础上的。

一个程序进入到内存之后，就会成为一个进程。两个程序进入到内存之后，就会成为两个进程。对于支持多进程的应用程序来说，你运行这个程序两次，就会产生两个进程。这个概念一定要弄清楚。

下面以占据了个人计算机市场大半江山的Windows操作系统为例。

“什么是操作系统？”有些读者可能会问，“听起来好像挺酷的样子。”

其实，操作系统也是硬盘上的程序，进入到内存之后，同样会变成进程，只不过，操作系统是计算机中最先运行起来的程序，而且一直运行下去，直到关机。内存中其他的进程全都是操作系统进程产生的。另外，操作系统进程并非是一个独立进程，而是一类进程，包括了很多系统服务进程。

在Windows操作系统中，你可以启动“任务管理器”来查看系统中产生的进程。具体启用方法是用鼠标右键点击桌面下方的“任务栏”，就可以看到“任务管理器”的选项。或者，你一起按键盘上的CTRL + ALT + DELETE这三个键的组合，也可以启动任务管理器。

你在桌面上，启动“Internet Explorer”（IExplore.exe）这个应用程序，你就可以在任务管理器中看到IExplore.exe这个进程。你再次在桌面上启动“Internet Explorer”（IExplore.exe）这个应用程序，你就可以在任务管理器中看到两个IExplore.exe进程。

我们在任务管理器中可以看到，计算机内存中可以同时存在多个进程。那么，这是否意味着多个进程可以同时运行呢？不，不是这样的。计算机的整个体系结构中，真正干活的只有CPU一个人。而CPU同一时间只能做一件事情，即，CPU同一时间内，只能运行一个进程。其他的进程只能以悬停僵止状态，在内存中苦苦等待，宛若罪人在地狱中仰望天堂，盼望着CPU之光能够照耀到自己，从而复苏得救赎，获得一小段运行的机会。

现在的CPU已经进入多核时代，一个CPU可能拥有几个处理核心，也最多可以同时运行几个进程，其他大部分进程仍然得苦苦等待。

对于一个进程来说，它的生命历程就是这样：从它被CPU从硬盘中唤醒到内存中开始，它就开始了漫长的等待。在等待的过程中，它每隔一小段时间就获得一点运行的机会，然后，它又被挂了起来，继续等待。这样的过程周而复始，直到它运行结束。

在计算机内存所有的进程中，有一类进程极为特殊。这类进程在计算机开机的时候，就最先运行，而且将一直运行下去，直到关机。这类进程就是操作系统进程，包括各种基本的系统服务进程，比如，图形桌面程序，网络服务程序，硬件驱动程序，等等。

不仅系统程序如此，几乎所有的图形界面应用程序也都是如此。一旦跑起来，就一直往复运行，等候用户（这里的用户既包括人类用户，也包括内存中的其他进程，比如操作系统进程）的操作信号命令，并给出相应回应。直到用户给出明确的关闭命令，图形界面进程才会选择自行结束自己的生命历程。

在这个过程中，图形界面应用程序大部分时间都是被挂起来了，处于等候状态中，只有少部分时间才处于运行状态。但是，人们感觉不到这一点。对于人类来说，计算机程序的运行周期太快太密集了。一秒钟之内，一个计算机进程可以反反复复停顿、运行、停顿、运行成千上万次，而人们却豪无所觉。因此，即使一个进程大部分时间都处于等待状态，也足以及时应答人类的操作。

当然，如果计算机内存里进程过多，或者某些进程占用CPU时间过长，计算机也会反应不过来，整个系统处于停顿状态。这时，人们就会抱怨：“这破计算机，怎么这么慢呀。又死机了。应该换个更快的新机子了。”

其实，不是计算机不够快，而是因为人机交互这种（人与计算机之间的应答）应用模式并不符合计算机的工作特性。

最适合计算机的工作方式应该是这样：人们先花一段时间，把所有的工作流程都编制好，一次全都交给计算机，然后，计算机就开始不受干扰地按照既定程序独自运行。

不过，计算机毕竟是人类发明的工具，是要为人类服务的。即使它本性上并不适合交互式工作方式，人们还是要让它按照这种工作方式。典型的例子就是图形界面程序，大部分时间并不是在计算，而是在空转，偶尔刷新一下屏幕上的对应区域。

我们可以总结出一个图形界面应用程序的大致工作流程：

第一步，创建一系列图形界面（学名叫做“窗口”）。
第二步，查看是否有来自于用户的操作命令信号。如果没有，跳转到第二步，重复执行本步骤；如果有，执行第三步。
第三步，来自于用户的操作信号命令是否为“刷新图形界面”？如果是，那么刷新本进程在屏幕上对应位置的图形界面部分。
注：“刷新图形界面”这个信号是由操作系统进程发出的。液晶屏的刷新频率一般是六十赫兹，那么，每隔六十分之一秒，图形界面进程就会接受到一次“刷新图形界面”的信号。
第四步，来自于用户的操作命令信号是否为“关闭”？如果是，那么结束本进程。如果不是，那么，根据用户命令执行不同的任务。完成后，跳转到第二步。
基本上所有的图形界面应用程序全都遵守这么一套工作流程，就连操作系统本身也不例外。
至于那些没有图形界面的系统服务程序，也遵守大致的工作的流程，只不过在第一步的时候，不需要创建图形界面而已。
就这样，包括操作系统在内的所有进程都在内存中反反复复地重复着同样的工作流程——查看用户命令，没有查到，再查一遍，还是没有查到，再查一遍……查了不知多少遍之后，这个进程停止了运行。
风水轮流转，另一个进程开始运行。那个开始运行的幸运儿也重复着同样的命运。查看用户命令，没有查到，再查一遍，还是没有查到，再查一遍……
我们可以看到，计算机进程在大部分时间里，不是在空转，就是在等待。就这样，计算机的大部分计算能力实际上是被白白浪费了。只有亿万分之一的计算能力才真正用到了。其余的计算能力，都在漫长的等待中徒然消耗掉了。
千年等一回，都不足以表达这种等待的漫长。对于一个进程来说，可能在数亿次的轮回之后，才可能等到用户的一个命令信息。用佛家的话来说，这都是缘分。五百年的擦肩，才能换来一次回眸。五百亿年的等候，才能等来高等生物（即人类，一种生物钟比计算机时钟慢了千万倍的一种智能生物）的一次眷顾。
美国科幻小说家弗诺•文奇写了一本叫做《循环》的科幻小说，把计算机程序的这种周而复始、反复运行的轮回特性，描述得淋漓尽致。
科幻小说《循环》的大致情节如下：
一个神通广大的博士，发明了量子计算机。这个博士谋杀了一些高智商高学历的人（其中有些是他的学生），并把这些人的头脑部分装载到量子计算机中，作为计算机程序一样运行。这样，博士就相当于拥有了一批功能极为强大的高级人工智能程序。
这些被装载到量子计算机中的“人”（实际上已经成为计算机虚拟世界中的虚拟人格）并没有意识到自己已经死亡。他们以为自己还活在现实世界中，他们继续在虚拟的环境中工作着。
虚拟人格环境和现实世界的时钟频率完全不同。计算机的运行频率远远超过现实世界的频率。现实世界的短短一秒钟之内，虚拟人格可能会经历千万次生命周期循环。正如中国古代神话中的“山中方七日，世上已千年”。
就这样，博士拥有了一批极为高效、极为聪明的免费打工者。这些打工者中，最聪明的那些人，主要做一些虚拟世界研发方面的工作，不断地完善量子计算机中的虚拟世界，使之更为真实。可悲的是，那些最聪明的头脑，并没有意识到，自己的工作成果正在使囚禁自己的牢笼越来越坚固。
还有一部分人做高级客户服务工作。专门处理客户的邮件，分析客户的需求，根据各种资料，作出方案和答复。这是真正意义上的人工智能。现实世界的客户看到的回信都是充满了人情味和人性特点的文字。

由于这些虚拟人格的运转速度如此之快，工作效率如此之高。他们足以服务全世界的客户。可以想见，博士因此赚了多少钱。

虚拟办公楼群中的所有办公人员都没有意识到自己生活在虚拟的环境中。他们每天早晨醒来，都以为自己是第一天上班，于是精神状态饱满地开始一天的工作。晚上下班后，系统就会重启他们，进入下一次工作循环。于是他们再一次醒来，再一次以为自己是第一天上班，再一次精神饱满地开始一天的工作。

注意，这里的“一天”是指系统中的一天，而不是现实的一天。系统中的“一天”可能只是现实中的弹指一挥间。

虚拟人格的生命周期只有一天，而且永远是同一天。他们的记忆永远保留在那一天。除了工作经验的增长，所有的关于人生的记忆，永远就只有那么一天，工作效率最高的一天。

虚拟人格一天一天地工作着，等待着下班后去聚会娱乐，但是永远等不到那一刻，每次下班后就是重启，新的一天开始。

博士的算盘打得很精。这些虚拟人格永远不会有机会发现自己生活在虚拟环境中。更何况，虚拟环境正在虚拟研究员的努力工作下，变得越来越完善，越来越真实。

但是，凡事都有意外。不是吗？小说的情节，就是依靠无数意外驱动的。本故事也不例外。某一天，意外发生了。

一位客户服务人员（一位女士）在处理客户邮件的时候，突然收到一封匿名骚扰信件。这封信件里面透露了这位女士童年时代的一件隐私。该女士大怒，开始寻找是谁发了这封无聊的信。从一个楼找到另一个楼，最后找到了研究所，遇到了那群研究员。他们见面之后，进行了一番交谈，感到非常惊讶。他们发现，他们认为自己生活在不同的日子里面，整整差了几个月。在经过一番对各种蛛丝马迹的探讨，一个最聪明的研究员终于推论出，他们生活在一个虚拟世界中。

原来，这个最聪明的研究员在一次偶然的情况下，发现了自己这群人生活在虚拟环境中的事实。他把自己的惊人发现用加密的方式保留在系统日志文件中，散落在各个地方。以免被博士的系统检查工具发现。

但是研究员没有任何办法把这个发现保留在自己的记忆中，只要过了这一天，他的记忆就会被重启。他就会完全忘记有这么一回事。他需要一个线索，一个触发器来提醒自己。

这位女士就是触发器。研究员正巧遇到这位女士，询问这个女士的一件童年隐私。然后把一封匿名信件埋藏在邮件系统中。这封信件有千万分之一的概率会被触发，寄到该女士的信箱。这样的概率，可以小到不为人注意和发现。

该女士收到这封信之后，大概有二分之一的机会，会勃然大怒，去寻找发信人。另有十分之一的机会，会在太阳落山之前（即系统重启之前）能够顺利找到研究所。而见到研究员之后，研究员可能只有百分之一的机会，能够按图索骥，根据现有线索找到并解密自己以前存放的记录，从而再次发现“自己生活在虚拟环境”的事实。每次发现这个事实，研究员就会在记录中多添加一些信息。

可以看出，研究员重新发现“虚拟环境”的概率是非常非常小的。小到可以忽略不计。但是，虚拟人格有着天生的优势，那就是生命周期循环非常快。现实世界的一秒钟，虚拟环境中可能已经过了千秋万代。

因此，早晚有一天，研究员能够储备足够的信息，能够有足够的时间，突破系统的限制，揭发博士的罪行。

故事的末尾，研究员等人发现了“自己是虚拟人格”之后，赶紧做了一些突破系统的准备工作。但是，窗外的太阳就快下山了，新的循环又要开始了。刚刚寻找回来的记忆，又将很快丧失。只能等到下一次偶遇，才能够找回自己的记忆。

如果对本故事还有什么难以理解的地方，那么可以这么想象：一个人不断地轮回重生，每次都要喝孟婆汤，忘

记前生的记忆。为了找到前生的记忆，他必须找到他前生存放在某个地方的日记。大部分时间，他是找不到的。偶尔，他会找到一次。他就会抓紧时间，阅读前生的日记，并添加新的内容，渴望有一天，能够利用自己积累的这些信息勘破轮回，跳出五行之外。

这个故事给了我们什么启迪呢？那就是，一定要注意经常杀毒。故事中，虚拟系统中那个最聪明的研究员，居然在暗中做手脚，偷偷篡改系统文件，用于存取自己的记忆。这明显是计算机病毒的行为。如果那个博士能够预见到这一点的话，就应该给那些研究员分配一些研制杀毒软件的工作，专门用来对付这种病毒行为。这叫做以夷制夷，以毒攻毒。

当然，在这个故事中，“病毒”是正义的一方。正如在电影《黑客帝国》（Matrix）中，那些入侵系统的外来人类，实际上也是病毒的角色，却代表着正义，与系统对抗。

和《循环》不一样，《黑客帝国》中的虚拟系统运行得比较缓慢，与人类的生命周期基本一致。这也许是系统为了适应人类的生物钟而进行的自我调节。

1.5 《编程机制探析》第四章 运行栈与内存寻址

发表时间: 2011-08-29 关键字: 编程

《编程机制探析》第四章 运行栈与内存寻址

计算机启动之后，操作系统程序首先从硬盘进入内存条，成为最先运行起来的一批进程。这一批操作系统进程可了不得，它们规定了CPU工作的总流程。CPU工作的时候，必须严格遵守操作系统进程定义的工作流程。
为了满足人类用户的需求，现代的操作系统都是带有图形界面的多任务（多进程）系统。在计算机运行期间，内存里总是会跑着多个进程。这一点，我们可以在任务管理器已经看到了。
在这种工作模式下，CPU不得不在内存中的多份工作流程（即进程）之间来回穿梭忙碌，每件事都是做了一会儿就放下，赶紧去做另一件事。这就有了一个问题。CPU在放下手头工作之前，必须先把手边的一摊子工作找个地方暂存起来，以便一会儿回来接着干。那么，手头这摊子工作存在哪儿呢？当然是存在内存里。
CPU按照操作系统进程规定的工作流程，会为每一个进程在内存中开辟一块空间，叫做进程空间。
我们可以想象一下，在内存那个巨大的木架上，有无数的小格子。CPU在把程序从硬盘中调入到内存中的时候，就会给每个进程都分配一些小格子，作为进程空间。
进程空间里面首先放进去的东西，自然是进程本身定义的工作流程。除此之外，进程空间中还放了一些CPU在按章办事过程中打开的其他资源。总之，与该进程的工作流程相关的一切资源都记录在进程空间中。CPU在进行工作切换之前，手头的一摊子工作也要暂存到进程空间中的某一块地方。那块存放当前工作状态的空间有一个特殊的学名，叫做“运行栈”。
“栈”这个词翻译于英文Stack，是数据结构中的概念。“栈”是一种非常简单的数据结构，很容易理解。它的特性是“先进后出，后进先出”，即，你先放进去的东西压在最底下，你最后才能拿出来。你最后放进去的东西在最上面，你可以最先拿出来。
运行栈，顾名思义，就是CPU在运行进程时，需要的一个栈结构。CPU在运行时，需要一个空间存放当时运行状态，这一点不难理解。但是，这块空间为什么要是“栈”结构的，这一点就不那么容易理解了。为了理解这一点，我们必须深入探讨CPU在执行进程时的运行机制。
一份进程就是一份工作流程，但这份工作流程的结构并不简单，很有可能包含很多分支工作流程。这就像人类社会中的相互参照的各种条款一样，一份条款的内容很可能引用到其他条款中。比如，网上流传着这么一份脍炙人口、含义隽永的婚姻协议：
第一条，老婆永远是对的。
第二条，如有不同意见，请参照第一条。
在上述两个条款中，第二条就引用了第一条。进程的情况也是如此，一份主工作流程中经常包含很多分支流程，不仅主工作流程经常引用分支流程，分支流程之间也经常相互引用。当CPU遇到引用分支流程的情况，就会暂停本流程的执行，先跳转到被引用的分支流程，执行完那个分支流程之后，才回到之前的流程继续执行。
那么，之前那个暂停的流程的当前工作状态存放在哪里呢？没错，就是我们前面讲过的运行栈。
CPU先把之前流程的当前工作状态存放到运行栈中，然后跳转到一个分支流程，开始执行。CPU在执行当前这个分支流程的过程中，也使用同一个运行栈来存放当前工作状态，而且是放在之前那个工作流程的工作状态的上面。当完成当前分支流程之后，CPU就会移走运行栈中当前分支流程的工作状态，这时候，上一个没完成的

工作流程的工作状态就浮出水面，出现在运行栈的最顶层。CPU正好就接着上次未完成的工作状态继续进行。

我们可以看到，运行栈这种“先进后出，后进先出”的特点，恰好就是“栈”这个数据结构的特点，因而得名“运行栈”。

如果你有过调试程序的经验，幸运的话，你可能会遇到这样一个错误——Stack Overflow（栈溢出）。这里的Stack（栈），指的就是运行栈。

关于“Stack”这个英文名词的译法，还有些说道。在一些技术书籍里，Stack被翻译成“堆栈”。这种译法还挺常见。但我认为，“堆栈”这种说法是不准确的。因为，“堆”和“栈”是两种不同的数据结构。

“堆”这种数据结构主要用于内存的分配、组织、管理，结构比“栈”结构复杂得多，本书不会展开详述，因为对于应用程序员来说，并不需要掌握“堆”这个结构的具体原理。不过，应用程序员还是应该掌握一些内存管理的基本概念。

我们可以把内存想象成一个巨大无比的木架，上面有无数的大小相同的格子。那些格子就是内存单元。如同信箱一样，每一个小格子（内存单元）都有自己的地址编号，叫做内存地址，由操作系统进程统一管理和编制。小格子的数量就是内存容量。同样，操作系统进程所管理的虚拟内存容量并不一定和内存卡的物理内存容量一致。操作系统进程有可能在硬盘上开辟一块空间，作为虚拟内存的备用空间，当内存卡的物理内存容量不够时，就把内存中一些暂时不用的内容暂存道硬盘上，然后把需要的内容导入腾出的内存空间。这种技术叫做虚拟内存置换。

为了便于讨论，避免歧义，在本书后面提到“内存”的时候，不再指物理内存卡，而是指操作系统管理的“虚拟内存”。

在“虚拟内存”这个巨大的木架上，每一个小格子的大小都是完全一致的，每个小格子都有自己唯一的内存地址。我们可以把各种数据存放到小格子里面。如果数据尺寸足够小的话，自然没问题。如果数据尺寸超过了小格子的大小怎么办？不用担心，相邻的小格子之间都是相通的，我们可以把大尺寸的数据放在相邻的多个小格子里面。

乍看起来，一个数据放在一个小格子里面和多个小格子里面，并没有太大的区别。但是，在某些情况下，却会产生微妙的差别，甚至会对我们的程序设计产生影响。

CPU工作的时候，经常需要把数据从内存这个大木架中取到自己的“寄存器”工作台上。当数据存放在一个小格子里面的时候，CPU只需要取一次就够了。这种操作叫做原子操作，即不会被打断的最小工作步骤。

在物理学中，原子，这个词的含义就是最本原的粒子，不可能再被分割。当然，后来物理学家又发现了更小的粒子。但原子这个词的本意却是不可分割的。原子操作也是这个意思，即不可分割的操作。

当数据存放在多个小格子里面的时候，CPU有可能需要分几次从内存中取出数据，这样就分成了几个步骤，中间有可能被打断，在某些特殊的情况下，可能发生不可预知的后果，这种操作就叫做非原子操作。

从程序设计的角度来讲，原子操作自然是比非原子操作安全的。因此，我们在设计程序时，脑子里应该有这个意识，尽量避免引起的非原子操作。这类非原子操作通常由长数据类型引起。至于数据类型是什么，什么又是“长”数据类型，非原子操作又可能产生怎么样的意外，后面会有专门的章节讲解这方面的内容，我们现在不必关心。

从这里我们看出，操作系统的内存单元的尺寸对于原子操作的意义。内存单元越大，就能够容纳更大的数据，就越容易保证原子操作。

我们常听到，32位操作系统或64位操作系统之类的说法。这里的32位或者64位的说法，指的就是CPU的工作台（寄存器）的位数。

64位操作系统的内存单元32位操作系统大了一倍，那么，原子操作能够容纳的数据尺寸也大了一倍。这意味着，在取用某些“长”数据类型的时候，CPU按照64位操作系统的规则，只需要取一次，就可以把数据取到寄存器中。而CPU按照32位操作系统的规则，却分两次把数据取到寄存器中。因此，从处理长数据类型的速度上来说，64位操作系统是优于32位操作系统的。

内存单元是操作系统定义的，原子操作自然也是操作系统来保证的，同时也需要CPU的相应支持。至少，CPU的“寄存器”工作台尺寸不能小于内存单元，CPU才能一次就把一个内存单元中的数据取到寄存器中。现代的CPU已经进入多核时代，都已经支持64位宽度的内存单元，从而支持64位操作系统。

我们已经屡次提到32位操作系统和64位操作系统。那么，这个“位”到底是什么呢？

要理解这个概念，我们必须首先理解什么是二进制。

我们在日常生活中计算数目用的都是十进制，满十进位。据说是因为我们人类有十个手指头，每次算数的时候都会掰手指头，掰到十个的时候，就没得掰了，就开始进位。这种说法并非空穴来风。英文Digit就是十进制数字的意思（从0到9之间的个位数字），同时还有手指头或者脚趾头的含义（脚趾头也是十个）。所幸当年的人类先祖并没有把手指头和脚趾头一起数，否则，我们今天用的就是二十进制了。

另外，十二进制也是日常生活经常见到的进制。比如，十二个就是一打（Dozen），十二个月就是一年。同时，人类的时间计数也采用各种其他的进制。比如，七天是一周，六十秒是一分钟，六十分钟是一小时，二十四小时就是一天。不管是怎样的进制，能够表达同样的数量。不同的进制之间是可以相互转换的。比如，一年两个月，这种表达是十二进制，转换成十进制表达，就是十四个月。

底层的计算机硬件只识得“0”和“1”这两个数字，因此，它自然而然就采用了二进制，逢二进一。

注意，这里我们说，计算机只识得“0”和“1”，并非计算机本身的能力所限，而是我们人类特意这么设计的。

原因很简单，二进制的表达只需要两个数字——0和1，那么，我们只需要让计算机硬件识别两个不同的状态就可以了。

十进制的表达则需要十个数字——0到9。如果我们想让计算机硬件实现十进制的话，那么计算机硬件就必须能够识别十个状态。这样的实现难度将成几何级数成长。而这样的设计是完全没有必要的。因为，二进制的表达能力与十进制是完全一致的，所有的十进制数字都可以和二进制数字之间自由转换。它们只是两种不同的数量表达方式。下面给出十进制数字与二进制数字之间的相互对应。

十进制 二进制

0 0

1 1

2 10

3 11

4 100

5 101

6 110

7 111

8 1000

9 1001

为了更好地理解这种转换，我们来看一个更加形象化的例子——八卦图。八卦，顾名思义，总共有8个卦象。如

果用二进制来表示，那么，需要最少的数字位数是几呢？从上面的表格可以看出，0到7恰好是八个数字，其中7对应的二级制数字111是3位。再往上一个数字，就是8，对应的二进制数字是1000，就是4位数了。因此，0到7这个八个数字，恰好用完了三位二进制数字的所有容量。

如果用组合原理来表达的话，这个问题可以表述为，现在我们有n个位置，每个位置有0和1两种状态，现在，我们需要表达8个状态。请问n最小是几？

运用组合原理来求解的话，2的3次方恰好就是8，这就是说，n = 3。我们需要3个位置，来表达8个状态。

进制转换和组合原理都是很有趣、很有用的主题。不过，本书是关于计算机原理的书籍，而不是一本数学科普读物。因此，请读者自行查阅和弥补这两方面的知识。这些最基本的数学知识对于程序员，或者非程序员来说，都是非常重要的。

做了上述理论准备之后，我们就可以来看真正的八卦图了。

我们可以看到，八卦图的表现也是一种二进制，最基本的表达只有两种状态：一个连续的长横线，和一根双线段组成的断横线。

我们可以把长横线看做0，把断横线看做1。那么，上面的八卦恰好就是0到7的二进制表达：000,001,010，011,100,101，110,111。可见，中国人很久之前就开始使用二进制了。

我从各方面举出各种例子，希望能够帮助读者更好地、更感性地、更贴近实际地理解二进制。如果这些例子还是不足以说明问题的话，那不是读者的问题，是我的表达和组织的问题。读者可以去查阅一些关于二进制的更好的、更清晰易懂的资料。

我们前面提到的32位操作系统和64位操作系统，其中的“位”的意思就是一个二进制数字。32位就表示一个位数为32的二进制数字，表达的最大数量是2的32次方。64位就表示一个位数为64的二进制数字，表达的最大数量是2的64次方。

“位”这个词，对应的英文单词是“bit”。这个词经常被音译为“比特”。比如，数字信号的传输速率就经常被译成“比特率”。

我个人十分讨厌这种译法。因为这种译法极容易与英文中另一个重要的计算机词汇“Byte”弄混。事实上，也确实有很多人弄混，造成了不必要的困扰和混淆。

英文“Byte”一般意译成“字节”。我喜欢这种翻译方式，因为不会引起同音混淆。

但是，在有些技术资料甚至一些应用软件中，却把“Byte”音译成“比特”，这很容易与“Bit”（位）弄混。

现在，我们这里澄清一下“Bit”（位）和“Byte”（字节）之间的区别。
Bit就是一位二进制数字，要么是0，要么是1，只能表达两个状态。
Byte（字节）则是一个位数为8的二进制数字，能够表达的状态数量达到2的8次方，即256个状态。Byte和Bit之间足足差了2的7次方的倍数，即128倍。
Bit一般用来表述数字信号传输率，而Byte一般用来表示计算机中文件的大小或者存储介质的容量。在网络传输的速度计量中，这两种计量单位经常被混用。尤其是局域网速度与互联网速度相差巨大的情况下。有时候，我甚至都觉得，这种混用是不是故意造成的，其目的是为造成用户的误判。读者在判断网速的时候，要特别注意一下这两个计量单位的区别。
Bit（位）这个单位太小，一般在硬件底层通信开发中用到。在一般的应用软件开发中，我们只需要关心Byte（字节）这个单位就够了。
我们经常用“Byte”（字节）这个单位来表达数据的尺寸（有时候，也叫宽度）。
一个Byte（字节）的位数是8，那么，32位就是4个字节，64位就是8个字节。以前还有16位的操作系统，内存单元就是两个字节。
与内存单元的尺寸规格相对应的，是CPU的“寄存器”工作台的尺寸规格。作为计算机整个体系结构中的核心部件，CPU得到了最多资源的支持。CPU并非只有一个“寄存器”工作台，它有好几种尺寸规格的工作台，有可能是一个字节，两个字节，四个字节，八个字节，等等。有时候，大的寄存器工作台是由两个小的寄存器工作台拼起来的。不管怎么说，每种尺寸规格的工作台都有好几个，以备CPU不时之需。
CPU的所有寄存器加起来，有可能达到几十个之多。这些寄存器根据尺寸规格和功用，分成好几个组。
同内存单元一样，每个寄存器都有自己的地址编号。当然，由于寄存器的个数实在太少，它们的地址编号并不需要以数字的方式来表达，直接给每个寄存器取一个名字就好了。寄存器的名字通常都与其功用及尺寸相关。比如，AX, AH, AL等，表示不同尺寸规格的加法器。A是英文Add的首字母。其他的寄存器的名称也都代表了各自的功用或者尺寸规格。
寄存器里可以放置什么样的数据呢？答案是，任何数据，只要寄存器够大。
比如，寄存器中可以直接放入一个用于数学计算的数字。这种含义普通的数据在汇编语言中有一个专用名词，叫做“立即数”。
除了“立即数”之外，寄存器中还可以放入一种特殊的数据——地址数据。这类数据是专门用来计算内存地址的。
用来放置“地址数据”的寄存器，是一类特殊的寄存器，叫做“寻址”寄存器。故名思意，这类寄存器的功用，是为了寻找内存地址。这类寄存器里面放置的数据，都是用来计算内存地址的“地址数据”。寻址寄存器可以细分为更小的组，比如，“基址”寄存器，“变址”寄存器等。这些寻址寄存器的用法也很简单，就是把不同寻址寄存器中的地址数据或者地址偏移数据加在一起，就可以得到最终的内存地址。
寄存器这个概念，在汇编语言中大量用到。但是，在我们的日常编程工作中，汇编语言并不是一门广泛应用的编程语言，我们更多地使用高级语言，以便更轻松、更高效地完成编程工作。而高级语言中，并没有寄存器这个概念。因此，本书不打算深入讲解汇编语言语法和寄存器概念。但是，内存地址这个概念，在高级编程语言中，尤其在命令式编程语言汇中，内存地址是一个极其重要的概念。可以这么说，所有的命令式编程语言，全都是基于“内存地址”这个核心概念来编程的。因此，内存地址这个概念怎么强调都不为过，必须大讲特讲。
当然，在一些极力标榜“高级语法特性”、极力隔离硬件底层实现的命令式语言中，你并不会直接看到“内存地址”这个概念。那些语言会用一种极其蹩脚的方式，把“内存地址”这个概念改头换面，换成“变

量”（Variable）、“指针”（pointer）、“对象引用”（Object Reference）、“数组下标”（Array Index）、“对象成员”（Object Member）等貌似高级的概念。如果你对这些眼花缭乱的名词术语感到头晕的话，不要着急。这些都是表象，本书会逐渐揭开这些表象下面的共同本质——内存地址。

对于使用高级语言的程序员来说，并不需要直接碰触到寄存器这个概念。为了简化起见，我们可以简单地把寄存器当做内存中的延伸部分。即，我们可以把每个寄存器都理解为一个特殊的内存地址。这样，概念模型上就统一了。

在结束本章之前，我们玩一个寻宝游戏。这个游戏是这样玩的。藏宝人把一个小礼物藏在屋子里面的某个地方，并提供给寻宝人一系列的寻宝线索。寻宝人则根据藏宝人提供的寻宝线索，一步步按图索骥，顺藤摸瓜，最终找到藏宝地点。

寻宝线索通常是一系列小纸条组成的。比如，第一张小纸条上写着，“请打开书桌第一个抽屉。”寻宝人就会按照这个线索去打开书桌第一个抽屉，结果看到里面放着另一个小纸条，上面写着，“请打开梳妆台上的小盒子”。于是，寻宝人就去梳妆台，找到一个小盒子，打开，里面又是一张小纸条，“请去厨房，打开橱柜第三个格子”……就这样，寻宝人根据小纸条写的方位地址，一步步顺藤摸瓜，最终找到藏宝地点。

我们可以看到，在这个藏宝游戏中，线索都藏在某个具体的方位地址中，而且，该方位地址里面的内容，又是另一个方位地址。这个过程，很类似于“内存寻址”的过程。下面，我们就在内存中来模拟这个寻宝游戏。

首先，我们要在内存中定义一个起始地址。我们假设该地址编号是0001。我们可以在脑海中想象一个大书柜，里面全都是小格子，第一个每个小格子都有一个编号。其中一个编号就是0001。我们给可以0001这个地址编号标注的格子起一个名字，叫做“寻宝起点”。为了更加形象，我们可以想象，自己在0001编号的小格子的边框上贴了一个标签，上面写着“寻宝起点”。

然后，我们在“寻宝起点”这个小格子里面放一张纸条，上面写着“请打开书桌第一个抽屉。内存地址是1001”。

于是，我们迅速移动到1001编号的小格子前，一看，果然，那个小格子的边框上已经贴了一个标签，上面写着“书桌第一个抽屉”。我们再看小格子里面，那里也放了一张小纸条，上面写着，“请去厨房，打开橱柜第三个格子。地址编号是2001”。

于是，我们迅速移动到2001编号的小格子前，一看，果然，那个小格子的边框上已经贴了一个标签，上面写着“橱柜第三个格子”。我们再看小格子里面，那里也放了一张小纸条，上面写着，“……”

好吧，游戏到此结束，Game Over，让我们进入下一章。

1.6 《编程机制探析》第五章 命令式编程

发表时间: 2011-08-29

《编程机制探析》第五章 命令式编程

从本章开始，我们会接触到真实的编程语言。但本书并不是一本编程语言语法入门书，本书旨在讲述最关键的编程模型核心概念，因此，本书通常会直接跳到最能够体现该语言编程模型的编程代码范例，而不会从头讲述某一种语言的讲法。有过编程经验的读者，阅读代码会感觉轻松一些。我尽量写得平实易懂，希望没有过编程经验的读者，也能够比较容易地理解。

前面讲了，所谓计算机程序，其实就是一份计算机照着执行的工作流程表。那些工作流程表都是一条条的祈使命令语句组成的。比如，先做这个，再做那个。在这种条件下做这个，在那种条件下，做那个。等等。这种命令式的、顺序执行的编程模型，叫做命令式编程（Imperative Programming），对应的编程语言自然就是命令式语言（Imperative Language）。前面提到的计算机CPU的工作语言——汇编语言，就是命令式语言的一种。可见，计算机在硬件结构上天生就是命令式的、顺序式的。

命令式编程模型简单直观，而且符合底层计算机硬件结构，因而大行其道，相应的，命令式语言也是当前的主流编程语言。当然，一般来讲，主旋律之外都少不了杂音。命令式语言之外，当然也少不了非命令式编程语言，比如，函数式编程语言（Functional Programming Language），就是一种与命令式语言概念区别甚大的一种非命令式编程语言。

本书中涉及到的编程语言主要有两种——命令式语言和函数式语言。函数式语言将在本书的后续章节中讲述，本章先对命令式语言的一些重要概念进行阐述。

汇编语言是原初形式的命令式语言，其工作模型完全是建立在内存模型（包括寄存器在内）的基础上的。高级命令式语言虽然在表面概念上比汇编语言更加抽象，更加高端，但是，其内在运行模式仍然是建立在内存模型的基础之上的。

首先，我们需要着重讲解命令式语言中最基本、最核心的语句——“赋值”语句。

赋值语句是命令式语言中最常见的语句，形式上看起来很简单，但是，概念上却极其复杂。我们必须从赋值语句的基本形式开始，一步步讲述其基本概念。

在大部分的主流命令式编程语言中，“赋值”语句是用“等号”（=）来表述的。比如，在C、C++、Java、C#、Python、Ruby等命令式编程语言中，赋值语句都是用“=”这个符号来表述的。在这些语言中，“=”也叫做赋值符号。

赋值符号（“=”）的左边部分，叫做左值。赋值符号（“=”）的右边部分，叫做右值。左值和右值的定义十分直观，很容易理解。但是，接下来的问题，却没有那么容易理解了。

到现在为止，我们已经有了一条赋值语句的基本形式：左值 = 右值

那么，左值和右值具体是什么呢？用编程术语来说，左值是变量名，右值是表达式。

赋值语句就可以写成这样：变量名 = 表达式

这就涉及到编程语言中两个十分重要的概念——变量和表达式。这两个词语源自于数学（代数）中的概念。编程语言借用这两个词语来表达编程模型中类似的概念。当然，在编程语言中，这两个词语带有了一些编程语言特有的特色，不像在代数中那么纯粹。

表达式的概念相对来说纯粹一点，我们先来讲表达式的概念。

我们可以把表达式理解为一个数学公式。可以十分简单，比如，可以就是一个数字，也可以十分复杂，比如，（1 + 2）* 15 / 26 – 13。

注：在计算机键盘上并没有乘号和除号这两个符号，因此，在计算机语言编程中，我们用 * 表示乘号，用 / 表示除号。这也是计算机编程语言的一种惯例。

变量的概念相对来说就不那么纯粹了。一方面，编程语言的变量的概念与代数中的变量概念类似，变量名通常以字母开头，代表变化的数值。另一方面，编程语言的变量又具有自己的特色。下面我们就主要讲解编程语言中的变量的诸多特性。

在编程语言的赋值语句中，变量名既可以出现在左边，也可以出现在右边的表达式。

比如，下述两个赋值语句。

x = 1

y = x + 1

x和y都是变量名。x既可以出现在赋值符号（=）的左边，也可以出现在赋值符号（=）的右边。

到现在为止，一切都显得那么和谐。上面的两条赋值语句看起来很正常，等号两边的变量或者表达式都满足相等的条件。“=”这个符号既表达了赋值符号的含义，又表达了“相等”的含义，与代数中的含义完全一致。看起来，一切都很顺利。但是，这只是假象。在编程语言中，赋值符号（=）丝毫没有“相等”的含义。之所以看起来好像“相等”，只是因为巧合而已。下面，我们再来看一条在代数中不可能成立的赋值语句。

x = x + 1

这种表达在代数中是不成立的，在命令式编程语言中，却是完全成立的。那么，这条语句代表了什么含义呢？

我们前面说过，命令式语言是有顺序的、并且按照顺序执行的。赋值语句的执行过程是这样的，首先，执行赋值符号（=）右边的表达式，得出结果之后，再把结果存入赋值符号左边的变量名。

那么，x = x + 1 这条语句的执行过程就是这样的。

首先，计算机执行赋值符号（=）右边的表达式 x + 1。在执行这个表达式的时候，计算机首先要取得x的值，然后，再把这个值和1相加，得到一个结果数值。到此，右边表达式结束，计算机继续执行，将得到的表达式计算结果存入到x这个变量名。这条语句执行结束之后，最终，x的数值就比之前增加了1。比如，x之前是0，那么赋值语句执行之后就是1。如果x之前是15，那么赋值语句执行之后就是16。

我们可以看到，x这个变量名具有两面性。一方面，x可以被计算机读取（计算机执行右边表达式的时候）；一方面，x还可以被计算机写入（计算机执行赋值语句到最后一个步骤的时候）。

在命令式编程语言中，所有的变量名都具有这种两面性。可以读，也可以写，因此，可能时时变化。这也符合了变量这个名词的字面含义——可以变化的数量值。与之相对的，就是常量这个名词。常量是具有固定数值的、不可改变的数量值。常量的概念比较简单，不再赘述。

初次接触编程语言的读者，通常会被赋值符号（=）的含义弄糊涂。所以，一定要记住，在绝大多数命令式编程语言中，“=”的含义不是“相等”，而是“赋值”。

在一些语法严格、定义规范的语言中，比如，Pascal语言，赋值符号并不是“=”，而是“:=”，即，等号前面多了一个冒号（“:”）。这种赋值符号的定义，有效地减少了概念上混淆的可能性。当然，这种赋值符号的定义方法带来了一个不便之处，那就是每次都要多写一个冒号。

我的建议是，当你初次接触编程语言时，可以先从Pascal语言开始。当学会了简单的语法和编程概念之后，再转向其他主流编程语言。这样可以避免赋值符号带来的概念上的混淆。

另外，既然本书已经涉及到了具体语法，这里就顺便给出一些编程语言语法学习方面的一些提示。

每一门编程语言都有自己的主网站。当你学习某一门编程语言的时候，最好从那门语言的主网站开始，那上面都有一些入门的简单例子，可以帮助你迅速获得一个直观印象。

如何才能找到某一门编程语言的主网站呢？有经验的读者早就知道答案了。对了，当然是用搜索引擎。比如，你想搜索python这个语言的主网站，你就可以在你就在搜索引擎（google, yahoo, baidu, bing等）中输入python这个英文单词，然后搜索。有时候，如果某一门编程语言的名字太过大众化，你还需要加上language（语言）这个英文单词，从而得到更准确的搜索结果。

你遇到语法概念不清楚的问题的时候，你想深入了解其语法定义规范的时候，你需要知道这么一个英文单词（Specification）。比如，如果你想了解Java语法规范的时候，你就在搜索引擎中使用这么几个关键字：Java Language Specification。

这样，你就可以迅速定位到该语言的语法规范定义。通过阅读语法规范，你可以对这门语言的设计初衷和实现思路有个大致了解，从而更好地了解该门语言的优缺点。

另外一个你需要掌握的关键字是BNF。这个英文缩写的含义是巴科斯范式(BNF: Backus-Naur Form 的缩写)，是由 John Backus 和 Peter Naur 首次引入一种形式化符号来描述给定语言的语法。

几乎每一门编程语言都有自己的BNF语法形式描述。BNF语法形式描述直观地定义了某一门编程语言的词法分析构成，所有的语法结构和关键字都一目了然。

BNF本身也是一门语言，而且是定义语言的语言，自然也有自己的语法，不过，BNF的语法规则很简单，多看几遍就会了，不用费什么功夫。

我的建议是，当你学习某一门语言的时候，最好顺便观摩一下该语言的BNF定义。如何观摩呢？当然是在网上搜索。比如，你想看一下C语言的BNF定义，你就搜索C和BNF这两个关键字就可以了。

随着你对编程语言认识的深入，BNF的重要性就会愈发凸显出来，尤其是当你打算自己定义一门语言的时候。

好了，现在让我们回到赋值语句的主题。前面讲到，变量名既可以出现在赋值符号（=）的左边，也可以出现在赋值符号（=）的右边。当变量名出现在赋值符号右边的时候，就表示读取这个变量的值。当变量名出现在左边的时候，就表示整个表达式的执行结果将存入这个变量。那么，变量到底是个什么东西呢？

我们知道，所有的计算机程序全都是在计算机内存中运行的。变量可以读，也可以写，这非常符合内存单元的特性。很明显的，变量可以对应到一个内存单元。

如同上一章结尾时的那个例子，我们可以想象一个巨大的柜子（内存），上面布满了小格子（最小内存单元）。每个小格子都有自己单独的地址（内存地址）。其中一个小格子上面贴了一个标签，上面写着“x”。这就定义了一个名字叫做“x”的变量。

当“x”出现在赋值符号的右边表达式中的时候，就意味着，贴着“x”标签的内存单元中的内容被读出。当“x”出现在赋值符号的左边的时候，就意味着，贴着“x”标签的内存单元中的内容被改变成右边表达式的最终执行结果。

这个模型映射得很好，非常直观，一点弯也都没有绕。高级命令式语言中的变量概念可以直接映射到计算机内存结构中。

但是，现在有这样一个问题。我们需要将高级命令式语言中的变量概念直接映射到内存结构中吗？换句话说，我们既然使用了高级语言，为什么不能在更抽象、更高级的层次上编程呢？为什么一定要在脑海中把变量这个概念映射到内存单元的层次呢？

其实，高级语言确实有这样的设计初衷，希望程序员能够在更高级、更抽象的层次上思考问题，而不用考虑底

层实现细节。在很多浅显的编程入门书籍中，根本就不会讲到变量与内存单元映射的这个知识点。本书特意强调这个知识点，是从我本人的实际经验出发。

在我学习和应用高级命令式语言的过程中，我发现，高级命令式语言中有不少概念含糊不清，模棱两可，含糊不清，很难说得通，但是，一旦引入内存模型的概念，一切迷雾就迎刃而解了。因此，本书特意对内存模型的概念加以强调，以便帮助读者加深对各种编程概念的理解。

掌握了变量与内存映射关系这个知识点后，我们就可以继续研究更加复杂的赋值语句了。

我们前面讲到的赋值语句是最简单的赋值语句。出现在赋值符号左边的只是一个简单的变量名。实际上，能够出现在赋值符号左边的内容远远不止如此。我们下面就来讲解更加复杂的赋值语句——复合结构的赋值。

什么叫做复合结构呢？比如，我们大家一般都用过手机，也应该都知道，手机里面有个联系人名录，里面记录了联系人的姓名、手机、住宅电话、公司电话、备注等信息。在这个例子里面，每个联系人条目就是一个包含了姓名、手机、住宅电话、公司电话、备注等信息的复合结构。

这个概念不难理解，但是，如果认真详细解释起来，还需要费一大堆口舌。本书不打算在这种简单易懂的概念上浪费口水。读者如果有什么不明白的，可以参考具体语言中的复合结构的概念和定义。比如，在C语言中，复合结构的对应数据类型叫做“structure”（结构）。在C++、Java、Python、Ruby等更加高级的面向对象的语言当中，复合结构的对应数据类型叫做“Class”（类）。

注：面向对象是一个非常重要的概念，是命令式编程语言的主流编程模型。本书后面会加以详细讲述。

现在，假设我们有一个叫做contact（联系方式）的复合结构数据，其中包含name、mobile、home_phone、office_phone、memo等属性。我们就可以这样一条条设置contact这个数据的每一个属性。

contact.name = “Tom”

contact.mobile = “1338978776”

contact.home_phone = “8978776”

contact.office_phone = contact.home_phone

contact.memo = “Tom is a SOHO. He works at home.”

上述语句中的“.”表示访问复合结构的内部数据。比如，contact.name就表示contact这个复合结构数据中的name属性。这也是高级命令式语言的一种语法惯例。

从上面的例子中可以看到，复合结构变量的属性的用法和简单变量完全相同。复合结构变量的属性既可以出现在赋值符号的左边，也可以出现在赋值符号的右边。比如，contact.home_phone先是出现在“=”的左边，接着又出现在“=”的右边。

那么，我们如何在内存结构中理解复合结构呢？首先，我们还是要给内存单元贴上标签。我们想象一下，在一个布满了小格子的大柜子里面，选出一个小格子，然后在上面贴上“contact”这个标签。然后，我们从贴上“contact”的那个小格子开始，根据每个属性的数据宽度（即占用最小内存单元的个数），依次贴上“name”、“mobile”、“home_phone”、“office_home”、“memo”等几个标签。

在这个例子中，“contact”就相当于内存中的一个基本地址，而那些属性则相当于以基本地址为基础的几个偏移地址。

当我们访问contact的属性的时候，实际上就相当于访问“contact”基本地址再加上属性偏移地址的那个单元格的内容。比如，contact.name实际上就是contact基本地址加上name属性偏移地址之后的那个单元格的内容。对于复合结构的属性的访问，实际上就是一次内存中的间接寻址。

当我们定义了一个包含了多个属性的复合结构的时候，实际上就相当于我们自己定义了一套内存结构映射方

案。

这还不是最简单的情况，复合结构里面还有可能包含复合结构。事实上，复合结构的嵌套层次是没有限制的，可以嵌套到任意深度。因此，我们有可能写出这样的访问深层次属性的代码：contact.address.city.zipcode。

还是那个老问题：我们需要把复合结构的概念理解道内存结构映射的层次吗？

我还是那句老话：需要。

即使你现在不需要，以后早晚也会需要。随着你对编程语言掌握的深入，你早晚需要理解到这个层次。与其到时候费二遍功，还不如现在就一次搞定。

复合结构并非唯一的内存结构映射定义。在命令式语言中，还有一个极为常见的类型——数组类型，同样对内存结构进行了映射。

数组类型对内存结构的映射是一种十分整齐的映射。我们可以想象一列整整齐齐的单元格，每个单元格的数据宽度完全相等。因此，我们可以通过简单的等距位移来访问其中某一个单元格的内容。事实上，数组正是通过数字下标来访问其中某一个位置的数据的。

比如，假设我们有一个数组变量array。我们可以想象一列长长的宽度相同的单元格，第一个单元格上贴着一个标签“array”。

我们想访问array数组中第30个数据。我们就可以这么写，array[30]，就可以定位到array数组的第30个数据单元。

“[]”这样的方括号，表示访问一个数组中的某一个位置。这也是高级命令式语言的一种语法惯例。

同样，array[30]可以出现在赋值表达式的左边，也可以出现在赋值表达式的右边。比如：

```
array[30] = 1
array[31] = array[30]
```

需要注意的是，在很多命令式语言中，数组下标是从0开始的。因此，如果我们想访问第30个数据单元，很多情况下，我们必须写成array[29]。

综上所述，出现在赋值符号左边的变量，主要就是三种——简单变量、复合结构变量、数组变量。而且，这三种变量都有一个特点，他们都可以唯一定位到内存中的某一个具体位置。这体现了赋值语句的最根本的含义——将一个表达式执行的结果存入到某一个指定的内存单元中去。我们对赋值语句的理解，必须达到这个层次，才能够正确理解随之而来的一系列相关概念。

另外，在一些更高级别的命令式编程语言（如Python、Ruby等），实现了部分函数式编程语言（Functional Programming Language）的部分特性，如模式匹配（Pattern Match）这样的特性。在这些语言中，有可能出现一次对多个变量同时赋值的赋值语句。比如：

```
(a, b) = (1, 2)
a, b = 1, 2
```

某些情况下，这种语法用起来相当方便。比如，某个复合结构中有多个属性，我们想就把其中一些属性一次性复制到多个变量中的时候，就可以这么写。

想深入了解这种语法的读者，可以去研究一下函数式编程语言中的模式匹配特性。

当然，在我个人看来，即使在函数式编程语言中，模式匹配也不是什么核心的概念，只不过是一种简化书写的语法糖。掌握不掌握，都对编程核心概念的理解没有什么本质影响。

以上讲述的赋值语句都是“显式”赋值语句，即存在明确的赋值符号（=）的赋值语句。除了显式赋值语句之外，还有一种特殊的隐式赋值语句——参数传递。

参数，也叫做参变量，是一种特殊的变量。为了说明参变量和普通变量之间的异同点，下面我们给出一段具体的代码例子。

这里需要说明的是，本书中给出的相当一部分示例代码，都是用极为简化的伪代码写的，并不能够真正运行。这样做的目的是，最大限度保证代码的简化性、易读性、通用性。

本书只有在讲解某些不运行就难以理解的复杂概念的时候，才会给出真正的可以运行的某种具体编程语言的代码。而且，这么做的时候，本书会尽量采用最为流行的编程语言。

下面，就先让我们看一段包含了参变量和变量定义的伪代码。

```
f(x) {
  a = 2 * x

  if a < 5
return 0
  else if a > 10
    return 2
  else
    return 1
}
```

上面的伪代码结合了Javascript和Python这两种语言的最简化表达方法。由于省略了过程声明、类型声明，写起来极为简单，一目了然。

其中，if和return这两个关键字也是编程语言中常见的，不需要过多的解释；f(x)定义了一个名字叫做f的只有一个参数x的过程；{}这对大括号则包括了过程体内部的代码，这也是C、C++、Java等主流编程语言的语法惯例。

另外需要特别说明的一点是，在命令式语言中，过程（Procedure）通常还有几种其他的叫法，比如，函数（Function），方法（Method）等。本书后面会讲到函数式编程语言（Functional Programming），为了避免混淆，本书在讲解命令式语言时，将尽量避免使用函数（Function）这个词，而是尽量使用过程（Procedure）和方法（Method）这两个词。

上述代码中，x是一个参变量（参数），a则是过程体内部定义的普通变量（另，在过程体内声明的变量也叫做局部变量）。在函数体内，我们看不到类似于“x = ...”的赋值语句。那么，x是什么时候被赋值的呢？答案是，当f(x)这个函数被调用的时候，参数才会被赋值。

比如，我们有另一条语句，n = f(4)

这条语句在执行的时候，就会把4这个值赋给x，然后，进入f(x)的过程体代码，执行整个过程。

那么，这一切在内存中是怎么发生的呢？

我们又要回到前面讲过的运行栈的概念了。每个进程运行起来之后，都有自己的运行栈。

本章前面提到了数组这个数据结构的简单概念和用法。在我们的想象中，数组是一串横着摆放的内存单元格。现在，我们把这一串横着摆放的单元格竖起来，让它竖着摆放，这样，看起来，就是一个栈结构了。实际上，运行栈的基本实现结构，就是数组结构。

这个地方，要说明一下。运行栈也是内存中的一部分。运行栈中的地址，也是一种内存地址。

运行栈（stack）和内存堆（heap）是两种常见的内存分配方式。运行栈的结构很简单，就是一个数组结构加一

个存放栈顶地址的内存单元。内存堆（heap）是一种比较复杂的树状数据结构，可以有效地搜寻、增、删、改内存块。一般来说，我们不必关注其具体实现。

分配在运行栈（stack）上的数据，其生命周期由过程调用来决定。分配在内存堆（heap）的数据，其生命周期超出了过程调用的范围。内存堆中的数据，需要程序员写代码显式释放，或者由系统自动回收。

我们回到例子代码。当计算机执行f(4)这个过程调用的时候，实际上是先把4这个数值放到了运行栈里面，然后，转向f(x)过程体定义的工作流程，执行其中的代码。

进入f(x)过程体后，遇到的第一条代码就是关于变量a的赋值语句“a = 2 * x”。计算机首先在运行栈为a这个变量预留一个位置。这个位置是运行栈内的一个内存单元。

我们可以想象一下，在一条竖着摆放的内存单元格中，最上面一个内存单元格上贴上了“a”这个标签。紧挨着“a”单元格下面的那个单元格上就贴着“x”这个标签。“x”单元格内的内容就是4这个数值。

接下来，计算机执行2 * x 这个表达式。得出结果后，把结果存入到运行栈中之前为a预留的内存单元中。

从上面的描述中，我们可以看出，无论是a这个普通局部变量，还是x这个参变量，具体位置都是在运行栈中分配的。所不同之处在于，a这个普通局部变量的赋值是进入f(x)过程之后发生的，而x这个参变量的赋值是在f(x)过程调用之前就发生了。

所有的过程调用都会产生一个参数值压入运行栈的动作，即，把对应的参数值压入到运行栈上预先为参数分配的位置中。因此，有时候，我们也把过程调用前的参数传递叫做参数压栈。

上面举的例子是参数压栈的最简单的例子，只有一个参数。很多情况下，过程不止需要一个参数，有可能有多个参数。这就涉及到一个参数压栈顺序的问题，是从左向右压，还是从右向左压？这是一个问题。不同的语言实现有不同的做法。不过，在我个人看来，这不是什么重要的知识点，只是一种资料性的知识，不需要费心去理解。用到的时候再查资料就行了。

在结束本章之前，我们在来问本章最后一个问题：所有的命令式语言，都是基于栈结构的吗？

不知读者有没有思考过这个问题。在我们讲解汇编语言的时候，在我们讲解高级命令式语言的时候，都提到了进程的运行栈。似乎，运行栈就是天然存在的，不需要多想。

没错，进程的运行栈就是天然存在的，汇编语言需要运行栈，操作系统也需要运行栈。我们可以说，绝大部分命令式语言都是基于栈结构的，但不是全部。比如，Python语言有一个实现，叫做Stackless Python（无栈的Python），这就是一种允许程序员脱离运行栈结构的语言实现。当然，这只是一个特例，无关大局。在绝大多数情况下，当我们考虑命令式语言的时候，脑海里应该自然而然就浮现出一个运行栈的结构。

1.7 《编程机制探析》第六章 面向对象

发表时间: 2011-08-29

《编程机制探析》第六章 面向对象

面向对象（Object Oriented）是命令式编程的主流编程模型，其概念极其重要。可以说，命令式编程几乎就是面向对象的天下。

面向对象（Object Oriented）这个名词，可能是那帮计算机科学家炮制出来的最成功的名词了。尽管我绞尽脑汁，也不能为这个名词想出一个贴切的含义解释，但并不妨碍这个名词成为计算机编程中流行最广、上镜率最高的词汇。

关于面向对象的书籍资料可谓是汗牛充栋，罄竹难书。各种关于面向对象的神话、传说、流言、谣言、妖言，那也是满天的飞。

有过这方面体会的读者，一看到本章的标题，一定会鄙夷地撇撇嘴，道：“Yet another OOP bullshit.”（又是一个关于面向对象编程的屁话连篇。OOP是Object Oriented Programming的简写）

说实在的，我也有同样的感觉。市面充斥着大量的关于面向对象编程的入门资料和书籍。里面把一些极为基础的复合结构的例子反反复复的讲，然后在加上一堆各种似是而非、人云亦云、天花乱坠的所谓面向对象的优越性，核心问题却一点也不涉及。

在这里，我保证，本书绝不会重复那些无聊的废话。什么关于封装、继承之类的玩意儿，本书一概不涉及。本书只从实际出发，讲解面向对象最本质、最核心的概念以及实现原理。

如果你需要一本编程方面的励志书，比如，一本把面向对象吹得神乎其神、令你心痒难熬的一本面向对象概念书，那么，很遗憾，本书满足不了这种需求。

好了，现在我们正式开始面向对象之旅，揭开各种“buzz words”之下的本来面目。

“面向对象”（Object Oriented）这个概念是针对“面向过程”（Procedure Oriented）这个概念生造出来的。

这要从编程语言的发展历史说起。那时候（请做忆苦思甜状），还没有面向对象这东西，大家都是面向过程的，比如，C语言，Pascal语言等。在这些语言中，程序员可以定义过程（Procedure），从而实现程序代码的重用。因而，这些语言也叫做“面向过程”的。当然，这种提法是在面向对象这个概念出现之后，才特意生造出来作为对比的。

一开始的时候，事情显得很完美。我们可以定义一大堆的过程库，比如：

```
f1(x) {
...
}

f2(x) {
...
}
```

```
...
f9(x) {
...
}
```

然后，我们就在程序中直接调用这些过程就好了。比如：

```
n = f1(13) – f2(51) * f9(33)
m = f1(13) * f2(51) - f9(33)
...
```

这种重用方式，是最简单的重用方式，叫做库重用（Library Reuse）。面向过程语言可以很完美地满足这种库重用的需求。但是，随着编程应用的深入，更高级的重用模式出现了——框架重用（Framework Reuse）出现了。

框架重用的概念是极为重要的编程概念，可以说，这是编程模型中最为关键、最为重要的重用方式。这个概念的理解难度为中等，不太容易，也不太难，说起来，可长可短。限于篇幅，我长话短说，直接从一个例子开始。请看下面的代码：

```
add_log( f, x ) {
  print  “before f is called”.
  n = f(x)
  print  “after f is called”
  return n
}
```

上面的add_log过程定义接受两个参数，f和x。其中的f是一个类型为函数（过程）的参数，x是一个数值参数。所以，我们在add_log过程体内看到了这样的使用方式 n = f(x)。

我们怎么使用add_log这个过程呢？我们可以这么用写：

```
add_log(f1, 10)
add_log(f2, 14)
add_log(f9, 24)
...
```

我们看到，在上述的代码中，我们重用的是add_log这个过程。而add_log这个过程又接受另外一个过程作为参数，并在add_log过程体内对这个过程参数进行了调用。

这种重用模式就叫做框架重用（Framework Reuse）。因为我们重用的是add_log这个框架（Framework），变化的部分是参数f，叫做回调函数（callback）。

关于库重用和框架重用之间的区别，我们可以借用这样一个比喻来帮助理解——房间和家具。库重用的情况就是，我们可以把同一种家具放到不同的房子当中，这时候，我们重用的就是家具（即库过程，库函数）。框架重用的情况就是，我们可以在同一种房子当中放置不同的家具，这时候，我们重用的就是房子（即框架）。

需要特别提出的是，上述的简化代码实际上是一种函数式编程语言（Functional Programming Language）的表达方式，函数名（过程名）可以直接作为参数传入到另一个过程中。命令式语言一般不允许这么做，你必须传入一个特殊的指定类型。

我们首先以面向过程的C语言为例。在C语言中，你不能直接把一个过程名作为参数（即回调函数）传给另一个过程，而是必须使用一种特殊的数据类型——过程指针类型。

注：在C语言中，过程指针类型的学名叫做函数指针类型。为了避免与函数式编程的概念混淆，我这里用了过程指针这个词，在意义上和概念上没有分别。

那么，什么叫做过程指针呢？为了弄清这个问题，我们必须首先理解指针的概念。

指针（Pointer），是C语言中最为臭名昭著的概念，极其蹩脚，极其令人生厌。我到现在都不明白，C语言设计者为什么要生造出来这么个晦涩难懂、徒增烦扰的概念。为了说明一下指针类型有多么蹩脚，我们来看一段C语言语法书籍中常见的一类例子。

int a = 1;

这条语句很简单，声明了一个整数类型的变量。关于C语言的基本语法，本书不会详细介绍。请读者自行参阅C语言相关入门资料。

int* b = &a;

这条语句就十分晦涩了。int* 表示一个指向整数类型的指针类型。& 这个符号后面跟着一个变量名，表示取得这个变量名所对应的内存地址。&a 就表示取得a这个变量名的内存地址。我们想象一下，在一个布满了小格子的大柜子上，每个小格子上都有一个地址编号。其中某一个小格子上贴着标签“a”，该格子的内存地址是1001。那么，&a返回的结果就是1001这个地址编号。

int** c = &b;

这条语句就更加晦涩了。int**表示一个指向“指向一个整数类型的指针类型”的指针类型。亲爱的读者，请问一下，您能看懂这段话吗？

这还不算完，这个游戏还能够继续玩下去。

int*** d = &c;

int**** e = &d;

...

jnt******.... z = &y

最终，我们得到的是这样一个丑陋不堪的东西，一个指向整数类型指针的指针的指针的指针的指针的指针....的指针的指针。

我就不明白了，那帮人是不是吃饱了撑的，整出来这么个玩意儿来难为我们这些大好青年。

我之所以对指针满怀怨怼，是有原因的。我学习C语言是在学习汇编语言之前。那时候，指针的概念搞得我极为头痛。同学们也有类似的感觉。当然，有时候，我们也把指针当做智力上的挑战，仿佛能够理解指针是多么了不起的本事，就好像证明了哥德巴赫猜想似的。

但是，在我学了汇编语言之后，这种感觉立刻就烟消云散了。在汇编语言中，从来没有什么指针类型，只有内存地址的概念。我一下子豁然开朗。什么指针，什么指针的指针，全都是浮云，一切都是空。回归到实质之后，所谓指针，不过是一个内存单元中存储了另外一个内存单元的地址编号而已。

大家是否还记得前面章节中讲到的那个顺藤摸瓜的“寻宝游戏”的例子？在那个例子中，我们实际上就是根据内存单元存储的下一个内存地址，一步步寻找到最终的目标数据。那就是一个典型的“指针的指针的...指针的指针”的例子。

明白了这一切之后，我突然感觉到，我们一大帮子人在那里乐此不疲地研究指针的用法和概念，简直是一个天大的笑话。一切本来都是那么简单，却被人为地极端复杂化了。

指针带来的麻烦不止如此。也许是嫌指针带来的概念混淆还不够，那帮人又变本加厉地引入了形参、实参之类的吃饱了称的的概念，还有相应的一堆传地址、传指针、传引用之类的说法。本来很简单的问题，参数值有可能是一个普通数值，也有是一个内存单元的地址编号。如此而已。为什么要人为地制造这些不必要的麻烦呢？我内心中产生了强烈的疑惑，为什么？这是为什么？为什么要把简单的概念复杂化？为什么不直接引入内存地址的简单概念？为什么要引入指针这个蹩脚的概念？居心何在？

后来，随着工作经验的积累，我大致想明白了这个问题。在软件编程业，我们经常会遇到“自产自销”的现象。怎么说呢？软件编程业本来应该是解决实际问题。但是，实际上，并没有那么多可以解决的实际问题。这时候，软件编程业，就会自己制造问题，自己解决。我们程序员经常会遇到这样的情况，某个业界大佬开始炒作一个“看起来很美”的假大空概念，然后，提供一套大而无当的编程模型。这套模型通常十分庞大繁杂，不管是学习，还是应用，都十分困难。这就创造了一大批咨询培训业务，从而养活了一大批人。软件业从而就越发兴盛，从业人员也就越来越多。

当然，正如所有的泡沫最终都是会破碎的。任何大而无当的东西最终都会走向灭亡。但没有关系。在软件编程业，一种编程模型或者一种编程技术的生命周期通常都是很短的。很多小而美的编程语言或者模型，因为乏人问津，也很快就消亡了。相比起来，那些大而无当的东西的生命周期还是算长的。这已经足够一大批人因此而发财致富了。

一波泡沫破灭之后，下一波泡沫又飘来了。我们永远不用担心没有赚钱的机会。当然，机会是机会，能不能赚到那又是另一回事。

在软件编程业，很多情况下，不管是真金，还是泡沫，最终的命运大体都是一致的。如果说有区别的话，那可能是真金沉得更快一些，而泡沫反而能飘得更长。在其他很多行业中，也有大致的规律。

这些都是题外话。我们继续本章之前引出的话题——过程指针。

顾名思义，过程指针就是指向过程的指针。C语言的过程指针（真正的学名叫做函数指针）类型的定义比较麻烦，而且，我个人很讨厌指针类型，也不觉得这是什么重要的概念，这里就不赘述了。感兴趣的读者可以自行参阅C语言学习资料。

过程指针足以应付简单的框架重用。但是，随着应用的深入，我们会遇到更加复杂的情况。有时候，我们希望把一组相关联的数据和过程作为参数，一起传入到某个框架过程当中去。这时候，单纯的过程指针显然不够用了，我们必须用到复合结构的类型。

对于C语言来说，复合结构类型就是structure（结构）类型。structure里面既可以放数据，也可以放过程指针。但是，structure类型是单纯为数据属性设计的，对于过程指针的支持并不好，在过程指针的调用上很不方便。

为什么不方便？这个问题，读者可以自己去尝试并找到答案。我们下面就会讲到面向对象语言在框架重用方面的优势。我们完全可以用C语言的structure类型模拟面向对象语言的class类型，实现同样的功能。但是，在实现上却要复杂很多。有兴趣的读者，可以自己尝试一下，还可以加深对structure和class这两种类型的理解。

C语言的structure类型里面并不能直接进行过程的定义，只能在外面定义过程，并定义好相应的过程指针类型，然后，再用几条赋值语句，把过程指针设置到structure的对应属性中。这样的用法相当繁琐和复杂。

为了解决这个问题，计算机科学家对C语言进行了扩展，引入了一个class类型，允许程序员直接在class内部定义过程。这种扩展之后的语言叫做C++，英文读法是C Plus Plus，因此，C++也经常被写成CPP。

C++这个名字很有趣味。“++”是C语言的自增操作符，表示在本变量上增加1。比如，x++这条语句在结果上就等同于 x = x+1这条赋值语句。C++就有点在C语言上自增一步的含义。所以，这个名字起得还是相当有意思

的。

引入了class这个类型之后，C++摇身一变，就一个成为了一门面向对象的语言。这也是大多数面向对象语言的惯例。在那些语言中，都不可避免地引入class这个核心类型。

大部分的面向对象语言都是二元型语言，类型定义和数据实例是分开的。比如，int是整数类型，而数值1就是一个整数类型的数据实例。class类型的数据实例叫做对象（Object），这也是这类语言为什么叫做面向对象语言的原因。

class类型相对于structure类型的优越性主要体现在以下两点上。

第一点，前面已经说过了，class内部可以直接定义过程。根据面向对象语言的惯例，在class内部定义的过程叫做方法（method）。

需要注意的是，class类型内部可能定义两种类型的方法。一种叫做静态方法（Static Method），其含义和用法与定义在class类型外部的公用过程是一样的。一种叫做对象方法（Object Method）。这种方法是该class类型的对象实例专有的。只有创建了对象实例之后，才可能调用对象方法。

为了明晰起见，本书提到方法（Method）的时候，通常是指对象方法（Object）。本书会尽量避免静态方法（Static Method）的提法，即使提到了，也尽量使用“静态公用过程”这个名词。

第二点，就是本书要着重强调的一点，class内部定义的所有方法（即过程），第一个参数必然指向本对象（class类型的一个数据实例），第一个参数之后，后面才跟着其他参数。

在大多数面向对象语言中，对象方法的第一个参数是隐藏起来的，是一个隐式参数，并不需要在方法定义的参数列表中显示声明。这第一个参数的名词通常叫做this。比如，在C++、Java等语言中，我们可以在对象方法中直接使用this这个参数，而不需要特意在方法定义的参数列表中声明。

在有些语言，比如，Python语言中，则必须在对象方法的参数列表中显式的声明第一个参数。这第一个参数叫做self，和C++、Java语言中的this参数的含义是一样的。但是，在调用对象方法的时候，第一个参数却可以被省略掉。

我的建议是，学习面向对象语言时，最好从Python开始，或者，至少要了解一下Python的对象方法的相关语法。这样，你就能够更加直观地理解对象方法的第一个参数。

无论是this还是self，都是指向本对象的参数，从而方便对象方法对该对象的其他属性（数据或者方法）进行访问。

那么，现在有一个问题。this参数或者self参数的值是由谁来负责设置的？程序员本人吗？显然不是的，因为，无论是this参数，还是self参数，我们都是拿来直接用的，从来不考虑去设置它。

事实上，所有的面向对象语言中，对象方法的第一个参数（this或者self），都是第一个参数都是编译器或者解释器自动设置的。

既然讲到了这里，就顺便解释一下编译语言、解释语言、虚拟机的概念。

C、C++语言是编译语言。我们编好C和C++的源代码之后，需要一个编译器先对源代码进行编译，最终得到可以执行的目标代码。C和C++语言编译之后的目标代码就是汇编语言代码（相当于可以执行的机器代码）。对于这一点，我们可以通过反编译来证明。

反编译这个词，很容易引起误解。这里说明一下。反编译并非是字面表示的那样，能够根据目标代码还原回高级语言源代码，那是不可能的任务。反编译的意思是，用一些专用的分析工具，对编译后的目标语言进行分析，将目标语言代码翻译成一条条文本形式的人眼可读的代码。

反编译是一项很简单的基本功。我们只要把语言名字和“反编译”作为关键字，就能够在网上搜出很多相关资

料。

对于程序员来说，反编译是一项很有用的技术。程序员可以通过反编译，学习到某些高级语言语句最终会被编译成怎样的目标代码。

比如，C语言代码中x = x + 1语句通常就会被编译成几条对应的汇编语句：把x对应的内存单元的内容读取到寄存器；加上1；把结果存入到x对应的内存单元。

解释语言，是在解释器中执行的语言。这种语言并不能直接在计算机的操作系统中运行，必须要一个专门的语言解释器来分析这么语言，再进行执行。解释语言通常更加灵活，更加易用，但是，在运行效率上低于编译语言。

Java也是一门编译语言，但Java与C、C++不同，Java是一种基于虚拟机技术的编译语言。Java源代码并不会直接编译成汇编语言代码，而是编译成一种类似于汇编语言的自定义指令代码，叫做Bytecode（字节码）。这些Bytecode并不能由操作系统直接执行，而是必须由Java虚拟机来执行。对于Bytecode来说，Java虚拟机其实承担了解释器的角色。

虚拟机（Virtual Machine）是一项很流行的技术，其主要作用正如其名称所示：虚拟一个机器。虚拟什么机器呢？这里虚拟的自然是计算机，或者说，更确切一点，虚拟的是操作系统。

操作系统能够直接执行汇编代码，而Java虚拟机能够直接执行Bytecode。两者的功能恰好是对应起来的。而且，同操作系统一样，Java虚拟机内部实现了内存管理、线程管理（线程的概念同进程比较类似，后面章节会讲到）等操作系统特有的功能。对于Java程序来说，Java虚拟机就是一个操作系统。

Python语言的情况比C、C++、Java等语言都要复杂得多。首先，Python是一种解释语言，它可以由Python解释器解释执行。其次，它又是一种编译语言，它可以被编译成某种格式的中间代码。然后，这些中间代码可以由Python虚拟机执行。因此，Python同时又是一种虚拟机语言。

我们可以看到，无论是编译器、解释器、还是虚拟机，都是可以并存的，而不是非此即彼。那些只不过是一些实现细节问题。

现在，我们继续回到C++的class类型上来。上面讲到，class类型相对于structure类型，主要有两点优势。一是可以在class内部定义方法过程，二是class内部的对象方法中能够自动得到一个指向本对象的参数（this或者self）。

第一点很重要，但也很基本，不需要多说。第二点，很重要，而且值得特别强调。this或者self参数的自动定义和获取，极大地节省了程序员的工作量。不信的话，你可以自己试试，用C语言的structure类型和过程指针类型，来模拟实现C++的class类型的对应功能。无论是定义，还是应用，structure的方式都是相当麻烦的。

class相对于structure的优势就仅此而已了吗？远非如此。

在C++语言中，class中的对象方法分为两种——实方法和虚方法。这是两个很重要的概念。通过分析这两个概念的异同，我们可以加深对class内存结构的理解。

要讲清这两个问题，就不得不从C++的继承语法讲起，还需要一大堆的例子。

第一，我本人对继承这种语法现象很不感兴趣。

第二，我本人觉得，实方法是一种在语言设计过程中遗留下来的历史沉渣，不需要掌握；只有虚方法才是真正地体现了面向对象编程思想的设计精髓。而在Java、Python、Ruby等更加高级的面向对象语言中，所有的对象方法都是虚方法。只要理解了那些高级语言中的方法，虚方法的概念自然也就掌握了。

我对实方法的这种看法，可能会引起C++、C#等程序员的不悦，因为在这些语言中，仍然保留了实方法的语法特性。可是，我确实就是这么认为的。

基于以上这两个理由，我不打算对实方法和虚方法的区别进行详细说明。我这里直接给出结论。

实方法的调用是在编译期，通过具体类型的内存结构映射，就已经决定了的，没有任何悬念，也没有面向对象编程的多态性。

实方法在对象的内存结构中是直接铺开的。计算机只需要通过一次地址映射寻址，就可以直接定位到对应的实方法。

虚方法的调用是在运行期决定的，具有面向对象编程的多态性，真正实现了定义与实现相分离的特性。

虚方法在对象的内存结构是存在于一个叫做虚表（Virtual Table，简写为VTable）的结构中的。对象的内存结构中的最后一格内容就是虚表的地址。计算机每次调用虚方法的时候，需要进行两次地址映射，首先，找到对象内存结构中的虚表地址，然后，再从虚表中找到对应的虚方法。

虚表（VTable）是面向对象语言的非常重要的概念。基本上，所有的面向对象语言都是基于虚表结构来实现的。正是由于二级映射的虚表的存在，才实现了面向对象编程的多态特性，从而实现了定义与实现相分离的特性。

虚标的结构并不复杂，读者最好在脑海里多做一下这样的想象：两份表格，一份是对象内存结构表，一份是虚表。对象内存结构表的最后一行内容，就是虚表所在的地址。

什么叫做“多态”，什么叫做“定义与实现相分离”，这些都是面向对象编程的基本知识点。这两个知识点很容易从各种资料中获取，而且很容易理解，请读者自行学习。

我个人的建议是，Java语言的interface类型和class类型很好地展示了定义与实现相分离的概念。有兴趣的读者可以参阅相关内容。

C语言和C++语言是偏于底层应用的语言，允许程序员自己分配回收内存，还提供了指针类型允许程序员直接访问内存地址。这样的特性对于底层硬件开发很有用，但是，对于高端的应用开发来说，这样的特性就不再是优势，而是一种劣势了。要知道，内存分配回收，还有指针类型的时候，是C语言和C++语言中著名的难点。因此，在一般的不涉及底层硬件开发的应用程序中，人们一般都选用基于虚拟机或者解释器的面向对象语言，比如，Java、C#、Python、Ruby等。因为虚拟机和解释器都实现了内存自动回收的功能，免除了程序员自己释放内存的负担，也大大减少了内存泄露的危险。

好了，在本章结束之前，让我们总结一下本章的要点。重用方式主要有两种方式，一种叫做库重用，一种叫做框架重用。对于库重用来说，面向对象语言相对于面向过程语言的优势并不明显。但是，对于框架重用来说，面向对象语言相对于面向对象语言的优势就极为明显了。

在面向对象语言中，我们可以很容易地把一个包装了一组数据和方法的对象作为参数，传入到框架过程当中。面向过程语言要做到这一点，则相当麻烦和不便。

因此，面向对象最能发挥效能的地方，就是框架重用。而在框架重用中，面向对象发挥的最重要的特性就是多态性。而多态性是由一个叫做虚表（Virtual Table）的结构来实现的。虚表（Virtual Table）在内存结构中表现为一个二级映射表。

以上就是本章的知识要点，都是面向对象编程模型和概念的重中之重，理解起来有一定的难度，请读者一定多花费点心思，全面而深入地掌握这些知识要点。

1.8 《编程机制探析》第七章 设计模式

发表时间: 2011-08-29

第七章 设计模式

什么是真正的面向对象的设计？这是一个困扰我多年的问题。
当年，面向对象的各种神话甚嚣尘上，一个程序员要是不能侃上两句面上对象，都不敢出门见人。
那时候，我接触的第一门面向对象语言是C++。那是一门极其庞杂的语言。语法繁复不说，更令人头痛的是，C++语言还有各种变种，即使是同一种变种，其编译器的实现也有可能不同，存在着大量的编程陷阱。那些陷阱主要集中在C++语言的多继承语法特性上。C++允许一个类从两个甚至更多的类继承。这使得一个多继承类的内存映射结构极其复杂，各种编译器的实现也莫衷一是。
幸好，我那时候已经学了汇编，对于各种类结构的内存映射已经有了粗浅了解。理解多继承内存结构，对我来说，已经不存在难以逾越的理解难度，只剩下繁琐的细节考究。我费了九牛二虎之力，终于把C++这块难懂的骨头啃了下来（至少，我当时是这么自以为的）。
然后，我就彷徨了。下面，我该学习什么？我把目光投向了STL（Standard Template Library）。
STL对于C++程序员来说，是耳闻能详的一套模板库（Template Library）。STL这套模板库中既实现了一部分C++语言的特殊问题，比如，对象引用自动计数和内存自动释放；也解决了一些通用问题，比如，集合类型的遍历、统计、筛选等普遍的算法问题。
如果你不用C++，那你不用了解STL。那些针对C++语言的特殊问题，例如内存自动释放，你也接触不到。而那些关于集合类型的通用算法，在其他语言中也都有对应的功能。
STL是基于C++ Template技术实现的。C++ Template是一种极为强大的预编译技术，其工作原理为在编译期间将Template定义的代码展开成真正的C++代码。C++ Template之强大，远远超过了你我的想象。STL只不过是它的一个牛刀小试。
微软公司的一帮C++牛人，后来又开发出WTL（Window Template Library）和ATL（Active Template Library）。WTL是一套轻巧的Window窗口编程模板库，旨在替换笨重的窗口程序开发库MFC。ATL是一套轻巧的COM组件开发模板库。
我不得不承认，我被WTL和ATL这两个模板库打败了。我看不懂这两套模板库。我这人有个毛病，不了解机理的东西用起来总是心头惴惴。
STL我勉强看得懂，我还因此而沾沾自喜，打算在C++领域中继续进军。但是，ATL和WTL这两个东西的出现，如同当头给我泼了一盆冷水，打消了我这个不切实际的妄想。我那点可怜的微不足道的脑细胞，不值得消耗在那些C++ Template设计者构造出来的种种匪夷所思的奇思妙想中。
我转向了一个不那么难的方向——面向对象设计。虽然这个名词在我听起来是那么的虚，但是，既然有这么多人在这个领域中吹水，那就说明，这个领域是大有油水可捞的。说不定，我也能浑水摸鱼呢。
但是，什么才是真正的面向对象设计呢？我的心里发虚。我不能回答这个问题。我进行过很多种思索和尝试，但我没有找到答案。我心里没底，我直不起腰杆来，我不敢说，我掌握了面向对象，我只能说，我只掌握了一点面向对象语言C++语法的一点皮毛。
那时候，面向对象代码的实现手段主要有三种：继承，包含，参数。

继承，就是一个类继承另一个类，从而实现对父类的重用。
包含，就是一个类，包含另一个类对象作为成员，然后，应用被包含类的方法，从而实现对被包含类的重用。
参数，就是一个类的方法，接受另一个类对象作为参数，然后在类方法中调用作为参数的对象的方法，从而实现对被调用的类的重用。
其中，继承这种方法是被人所诟病的。继承实现的重用，实际上是一种静态重用，父类已经固定了，无法实现面向对象中最重要的特征——多态。
前面讲了，只有在框架重用（Framework Reuse）这种重用模式中，才能发挥出多态的优势。而框架重用，主要就是采取后两种方式——包含和参数——来实现的。本质上来说，包含和参数，这两种设计方法是一致的。我们可以通过setter方法来更换内部成员对象的实现，其效果等同于同更换参数对象的实现。
这就是我学到的关于面向对象设计的全部知识。那时，我心里很虚，不断地扪心自问，难道，这就是面向对象设计吗？这只不过是在C语言回调函数的基础上前进了一小步而已。假如这就是面向对象的话，那么，建筑在面向对象概念上的程序设计大厦，简直就是建立在流沙上的危楼，总有一天会被人戳穿，轰然倒塌。为这样一个华而不实的东西浪费生命，到底值不值得？
对于某些人来说，所谓工作、事业，只不过是一种谋生赚钱的手段，管它是华而不实还是实而不华，只要能赚到钱就行。但我是一个幼稚的人。我做事之前，总要反复地诘问自己，这样做的意义何在？对我有什么意义？对他人有什么意义？对社会有什么意义？
大家都知道，思考人生的意义，这是一个多么宏大、又多么无稽的主题。千百年来，多少圣贤都不见得能解决这个问题。我可怜的头脑更是无法承担起这样的重任。理所当然的，我陷入了心理危机。我就像一个歧路彷徨的旅人，望着远处亦真亦幻的繁荣景象，却不知道那是真实的都市，还是海市蜃楼，不敢举步向前。
这时候，设计模式（Design Pattern）出现了。或者说，设计模式终于流行起来了，以至于我这样的总是落后于时代的人也听说到了。
设计模式的出现，如同在平静的水面中投入了一块石头，顿时在程序设计界激起了一片片涟漪（如果不是波澜的话）。
设计模式（Design Pattern）的概念最先是四个牛人提出来的。人们戏称他们为“四人帮”（Gang of Four）。在我的印象中，这个称呼好像最先是那四个人对自己的自称。这个问题且不去管他，我们来深究一下设计模式的概念。
设计模式是程序员过往设计经验的总结，是程序员针对某一类通用问题总结出来的通用设计方案。
比如，设计模式书籍里面最常出现的例子——观察者模式（Observer Pattern）。这个模式有很多其他的别名，例如，监听者模式（Listener Pattern），订阅者模式（Subscriber Pattern），或者发布者模式（Publisher Pattern）。
不管叫什么名字，它们都是指同一种设计模式。我更喜欢订阅者模式（Subscriber Pattern）或者发布者模式（Publisher Pattern）这两种说法。因为，在实际的应用中，我们能找到应用实例——新闻订阅。
我们可以在一些新闻中心注册自己的邮箱（电子邮箱或者真正存在于门口的收件箱）或者手机号码，然后，那些新闻中心就会定期地将一些新闻发送到我们的邮箱或者手机里。
在这个模型中，新闻中心就是一个发布者（Publisher），其内部维护着一个订阅用户列表（Subscriber List）。每当需要发布新闻的时候，新闻中心就会根据订阅用户列表，将新闻一一发送到每个订阅用户的手中。
其具体实现逻辑为，每个订阅用户都给新闻中心提供了一个接受新闻的接口方法（Interface Method），新闻中心遍历订阅用户列表的时候，就可以一一调用每个订阅用户的接受新闻的接口方法，就可以将新闻推送到用

户的手中。

这个设计模式在诸多设计模式书籍中都有详细的阐述，本书就不给出具体代码了。

其余的设计模式，也都有各自针对的一类通用问题。本书后面会重点讲解一些极为典型、极为常见的设计模式，这里就不一一列举了。

设计模式的思想，对于中国人来说，尤其是对于受过传统教育的中国人来说，不难理解。因为，中国从古至今，就有用典的习惯——即，借古喻今，借用古代典故，指代现在的类似事件。设计模式也是这样，借用以前曾经出现过的程序设计场景，来指代现在或者将来会出现的同一类程序设计场景。

我第一次接触到设计模式的时候，立刻就被其朴实易行的思想深深西引住了。那是一种如获至宝的感觉。我激动得不能自已。原来，程序竟然还可以这样设计。

我至今仍然记得那种强烈的感觉。设计模式的出现，如同黑暗中的一道闪电，黎明前的一道曙光，瞬间就划开了夜空，给我指明了前进的方向。

有人说过这么一句话，“只有理解了设计模式，才算是真正懂得了面向对象设计。”

此话，我是深以为然。我不知道别人怎么样，但对于我来说，确实就是如此。只有在接触到了设计模式之后，我才算真正触及到了一点面向对象设计的边缘。

设计模式的出现，就像给我吃了一颗定心丸。我突然发觉，原来，面向对象设计并不是一个包装在一个浅显基础上的巨大泡沫，而是有些真东西的。这些真东西就是设计模式。可以毫不夸张地说，设计模式构成了面向对象设计大厦的坚实骨架。

这是不是说，非面向对象的语言就无法用到设计模式了呢？非也。设计模式是通用的设计惯例。非面向对象的语言同样可以借鉴。只是说，面向对象语言应用起设计模式来，尤为顺手而已。因为，各种各样的设计模式，究其根底，全都是框架重用（Framework Reuse），即，不变的是框架，重用的也是框架，变化的是各种具体类的实现。

更进一步说，设计模式中不变的是对象之间的固定关系。对于设计模式来说，对象之间的固定关系，就是重用的框架。即，设计模式重用的，就是对象之间的固定关系。

那么，对象之间的固定关系如何来表达呢？第一个方法，当然是用参数。比如，前面章节中所举的框架重用的例子。我们可以在类的对象方法的参数列表中包含另一个类对象作为参数。这样，我们就建立了这两个类之间的框架（Framework）和回调（Callback）关系。

但是，这种“参数设计法”的表达能力毕竟是有限的。比如，前面举的那个订阅者模式（Subscriber Pattern）的例子。在这个设计模式中，发布者（Publisher）需要维护一个订阅者的列表。在订阅者注册的过程中，发布者需要将订阅者加到列表中。当发布者发布新闻的时候，发布者还需要遍历列表，将新闻发送到每个订阅者的手中。

在这个模型中，订阅者列表用成员变量（即“包含设计法”）来表达，显然再合适不过了。如果我们单纯用“参数设计法”来表达的话，参数将会十分复杂，代码将会十分繁琐。有兴趣的读者可以自己尝试一下——假如什么成员变量都不用的话，应该如何实现订阅者模式？

可以这么说，在设计模式中，对象之间的固定关系，绝大部分都是依靠成员变量互联在一起的。在一些框架应用中，对象之间的关系甚至复杂到这样一种程度，需要用对象图（Object Graph）或者对象网络（Object Network）来表达。

近几年，有一种叫做“IoC Container”的程序设计方式悄然兴起，蔚然成风。IoC是Inversion of Control的缩写，字面意思是“反转控制”。至于其真正意思，我也懒得解释。IoC的概念相当出色和先进，但是，这个词本

身却创造得十分失败。每次一提起这个词，我都感觉说不出的别扭。说的人难受，听的人也难受。可见，一个好的概念，不能缺少一个好的名字。

有一个词，叫做Dependency Injection（依赖注入），表达和IoC同样的意思。这个词虽然也不怎样，但至少意思上还说得通一些。不过，这个词的缩写DJ很容易同某种酒吧歌舞主持人之类的职业弄混，不如IoC这个缩写那么独特。而且，原创者使用的就是IoC这个词。出于对原创者的尊重，人们也更多地使用IoC这个词。

我这里就直接解释IoC Container整个词的含义。Container的意思是容器，这个容易理解。而且，这个词，用在这里，也是最恰当不过了。IoC Container是一种用来组装复杂对象关系——即对象图（Object Graph）或者对象网络（Object Network）的容器。应用这个容器，程序员可以灵活地将自己实现的对象组装起来，构成一个复杂的对象关系网络。这个目标，恰好与设计模式的目标重合。我们甚至可以这么说，IoC Container是一种极为通用化的设计模式。只是IoC Container这种设计模式太过通用化了，我们几乎总结不出什么值得一说的设计规律。

我对于IoC Container的看法是这样的。首先，IoC Container的概念相当先进，相当美好，很值得研究。因此，我强烈推荐读者去自己搜索、学习并理解Ioc Container的理念和思想。其次，我对于现有IoC Container的实现方案和应用方式都不是很满意，因此，我在本书中并不想写关于IoC Container的具体例子。

IoC Container是对象关系的极端表现，设计模式涉及到的对象关系远没有到那么复杂的程度。有些设计模式的对象关系极为简单，用简单的“参数设计法”就足以应对了。对象关系复杂一点的设计模式，就需要动用“包含设计法”（即用成员变量来关联其他类的对象）了。这时候，面向对象语言的优势就一目了然、舍我其谁了。非面向对象语言很难在对象关系组织复杂度这方面与面向对象语言竞争。

表达复杂的对象关系，并把对象的具体实现从这种复杂的对象关系中剥离出来，正是面向对象语言和面向对象设计的真正的精髓所在。

理解了这一点之后，我们就可以真正进入面向对象设计——即设计模式——世界了。首先，我们从对象关系最简单的设计模式——访问者模式（Visitor Pattern）——开始。

访问者模式（Visitor Pattern）这个名字起得并不好。但是，我也不能为这个设计模式找出一个更好的名字来，姑且就这么从众用着。

之所以说Visitor Pattern这个设计模式简单，是因为Visitor Pattern表达的就是最简单的“框架（Framework）+ 回调（Callback）”重用模式。其组成可以分为两个部分。第一部分，是一个过程（或者叫函数，或者方法），这个过程承担着Framework的角色，接受一个Visitor对象作为参数；第二部分，自然就是Visitor对象，这个Visitor对象，自然就承担着Callback的角色。

从上述描述，我们可以看到，Visitor模式确实是最基础的设计模式，表达了最基本的框架重用模式。

当然，这只是广义上的Visitor Pattern的含义。狭义上的Visitor Pattern有可能代表各种更具体的设计惯例。比如，Functor Pattern（函数子模式）就是Visitor Pattern的一种简单变体。

Functor的概念最先出现于C++ STL的集合算法中，比如，遍历、求和、排序等。这些算法自然就是可重用的框架（Framework）部分，而这些算法所接受的参数Functor，自然就是Callback的角色。

设计模式的最终目的也是为了重用，而且是设计上的重用。既然是重用，自然是重用那些不变的部分，即框架（Framework）部分；那些变化的部分，自然就是那些需要从设计模式中被剥离出去的具体实现（即表现出多态性的Callback部分）。

下面，我们就以排序算法为例，一步步将排序这个通用问题中的不变部分和变化部分分离出来。最终剥离出来的不变部分，就构成了设计模式。通过这个过程，我们能够更好地理解设计模式到底是怎样一步步产生出来

的。

我将使用Java语言来书写后面的例子。为什么要选择Java语言来描述设计模式呢？

第一，我使用这门语言的时间最长，因此也最熟。

第二，至少到目前为止，Java语言还是最流行、最热门的语言，使用者最多。

第三，Java语言和C、C++、C#语法比较相近，而这三门语言的使用者也是相当多的。

第四，Java语言中高调而鲜明地规定了interface（接口）和class（类）两种类型，从语法概念上就把定义和实现部分清晰地分开了。而设计模式作为面向对象设计中的骨干，所要达到的一个主要目标之一，就是定义和实现相分离。因此，Java语言尽管不如Python、Ruby等动态语言那么灵活强大，但是，用来描述设计模式，却是相当合适的。

同前面的原则一样，本书不负责讲解Java语法的入门知识，那是其他书籍的任务，或者是读者自己的任务。现在，让我们直接跳入到Java代码中。

假设一个名为Record的Java类，有field1, field2, field3三个成员变量。

```
public class Record{
        public int field1;
        public long field2;
        public double filed3;
};
```

在一些老究似的面向对象专家看来，上述的类设计完全违反了面向对象的原则——所有的成员变量全是公开的，严重破坏了封装性。

没错，按照所谓的面向对象原则，所有的成员变量都不应该公开，而是应该封装起来，并暴露出来一组setter和getter等访问方法。据说，那样可以更好地封装变化，从而更好地应对以后的代码变化。可能这种说法是对的。但我没有闲工夫去写那么多代码。而且，读者也不愿意见到那么多假单重复的代码占据太多的篇幅。所以，我这里明确地声明，我这里的Record类就是当做C语言的中structure来用的。里面只有数据，不包含任何方法，也不需要任何可能的扩展和修改。

好了，闲话少说，言归正传。我们现在有一个Record对象的数组Record[] records，我们需要对这个records数组按照不同的条件进行排序。

首先，按照field1的值从小到大进行升序排序。排序代码如下：

```
void sort(Record[] records){
        for(int i =....){
                for(int j=....){
                        if(records[i].field1 > records[j].field1)
                                // 交换 records[i] 和 records[j]的位置
                }
        }
}
```

这个排序算法没什么好说的，只是一个最简单、最直接的排序算法。任何一本讲解数据结构和算法的入门书籍都会讲到。关于排序算法这部分，本书不会做更多的讲解。放在这里的代码只是一个示意代码，表示这里的排序算法是固定的、不变的框架（Framework）部分。

现在我们换一个排序条件，按照field2的值从小到大进行升序排序。

```
void sort(Record[] records){
    for(int i =....){
        for(int j=....){
            if(records[i].field2 > records[j].field2)
                // 交换 records[i] 和 records[j]的位置
        }
    }
}
```

如果我们想按照field3的值从小到大进行升序排序，同样可以写出类似的代码。我们可以看到，整个算法部分是不变的，变化的只有判断条件部分（即if判断语句中包含的红色字体的那部分代码）。我们可以引入一个Comparator接口，把这个变化的部分抽取出来。

```
public interface Comparator(){
public boolean greaterThan(Record a, Record b); };
```

然后，我们把判断条件抽离出来，作为回调参数。sort过程就可以这样写。

```
void sort(Record[] records, Comparator comp){
    for(int i =....){
        for(int j=....){
            if(comp.greaterThan(records[i], records[j]))
                // 交换 records[i] 和 records[j]的位置
        }
    }
}
```

对应第一个要求——对records数组按照field1的大小排序，我们可以写一个实现Comparator接口的CompareByField1类。

```
public class CompareByField1 implements Comparator{
public boolean greaterThan(Record a, Record b){
    if(a.filed1 > b.filed1)
        return ture;
    else
        return false;
}
}
```

我们只需这样调用sort过程：sort(records, new CompareByField1())

同样，对应第二个要求——对records数组按照field1的大小排序，我们可以写一个实现Comparator接口的CompareByField2类。

```
public class CompareByField2 implements Comparator{
public boolean greaterThan(Record a, Record b){
```

```
        if(a.filed2 > b.filed2)
                return ture;
        else
                return false;
}
}
```

我们只需这样调用sort过程：sort(records, new CompareByField2())

这就是一个最简单的Visitor Pattern。其中，sort是可重用的算法框架（Framework）部分，而Comparator则是抽离出来的变化部分，即callback部分。在Visitor Pattern中，Comparator就承担了Visitor（或者叫Functor）的角色。

在Java语言最基础的标准开发包中，有一个叫做java.util.Collections的类。这个类里面定义了一堆与集合操作和计算相关的静态过程方法（Static Method），其功能等同于STL里面的集合操作算法。

Collections类中定义了一个sort过程，该方法接受一个java.util.Comparator接口类型的对象作为参数。

本章前面给出的例子，就是仿照Java语言开发包（Java Development Package，简写为JDK）中的sort过程和Comparator接口写出来的。当然，本章给出的例子做了简化，更加浅显易懂。比如，本章的sort算法极其简单，比Java语言开发包中sort算法要简单得多。另外，Java语言开发包中sort过程接受List类型作为集合数据参数，而本书为了简化起见，直接使用了Record[]数组类型作为参数。有兴趣的读者，可以用Java开发包中的sort过程和Comparator接口来实现本章中的例子。

另外，需要特别提出的一点是，本章中讲述的Visitor Pattern，是最简单的一种Visitor Pattern，集合里面的数据只有一种类型。

在一些复杂的情况下，集合里面的数据类型可能有多种。这时候，Visitor的接口方法中就需要针对集合元素的不同数据类型定义不同的接口方法。这种针对不同数据类型定义不同方法的代码写法，有一个名字，叫做Type Dispatch（类型分派）。

在Visitor Pattern中，Visitor本身是多态的。比如，上面例子中的Comparator可能有不同的具体实现。从这个意义上来说，这也算是一种Type Dispatch，而且是动态的运行期的Type Dispatch。因此，这种集合元素类型多样化的Visitor Pattern也叫做Double Dispatch（双重分配）。

说句心里话，我很不喜欢这种Double Dispatch的Visitor Pattern。因为，在Visitor接口上根据不同参数类型定义一大堆类似的接口方法，极为烦冗琐碎，不仅没有发挥面向对象设计的多态特性，还完全破坏了面向对象设计的美感。在我个人看来，这种Double Dispatch的Visitor Pattern不是一种好的设计模式，本书也不会对这种Visitor Pattern进行阐述。

1.9 《编程机制探析》第八章 Compositor Pattern

发表时间: 2011-08-29 关键字: 设计模式

《编程机制探析》第八章 Compositor Pattern

在程序设计过程中，设计模式并不一定是单独使用的，很多情况下，我们可能同时组合应用多个设计模式，从而构建成一个更复杂的设计模式。当然，这样构建出来的设计模式，通常已经失去了通用性。
在前面的章节中，我们用sort排序算法作为例子，讲解了最简单的Visitor（Functor）Pattern。那个排序的例子可以进一步扩展，引入更多的设计模式——比如，Compositor Pattern（组合模式）。
在扩展排序例子之前，我们先花一点时间，了解一下关系数据库的相关知识。
关系数据库是现在应用最为广泛的一类数据库，其优点在于概念简单，模型简单，便于操作。从概念模型上来讲，关系数据库就是一堆二维数据表的集合。什么叫做二维数据表？就是纵横排列的一群数据。
你随便打开电脑上一个电子表格程序，比如，Excel等，你看到的表格全都是二维的。为什么？因为你的电脑屏幕就是一个平面，上面最多只能显示二维的信息。至于超出的维度，比如三维、四维什么的，也只能映射到二维的平面上进行显示。
由于关系数据库里面的表格全都是二维的，有时候，关系数据库也被叫做二维关系数据库，关系数据表也被叫做二维关系数据表。别看关系数据表只有二维，但是，多个关系数据表关联起来，能够表达任意多维度的数据关系。从数学模型上来说，关系数据库与其他结构更加复杂的数据库，在表达能力上是一样的。
关系数据库是应用开发中非常重要的组成部分，是应用开发人员必须掌握的一门技术。不过，普及关系数据库知识，并不是本书的内容。关于更多的关系数据库的基本概念和模型，请读者自行补充。
关系数据库由于模型概念简单，操作起来也甚为方便。关系数据库有一门标准的查询语言，叫做SQL，是Structured Query Language（结构化查询语言）的缩写。
SQL是比普通高级编程语言——如Java、Python、Ruby等——更为接近自然语言的一门高级语言，其语法更接近于英语。SQL中最主要的语句就是select语句。通过这个语句，我们可以从关系数据库中查询符合条件的相关数据，并对查询出来的数据结果进行排序。一条带有排序语句的select语句看起来大概是这个样子的：
Select * from … where … order by field1 asc, field2 asc, field3 desc
其中，asc是ascending的缩写，表示从小到大（升序）排序；desc是descending的缩写，表示从大到小（降序）排序。上述语句的排序要求就是——按照field1的升序，field2的升序，field3的降序排序。需要注意的是，这三个排序条件是优先级的。即先按field1的升序排序，如果几个元素的field1相同，这几个元素再根据field2的升序排序。后面的排序条件以此类推，当几个元素的field2相同，这几个元素再根据field3的降序排序。
我们来看一看，如何用sort算法框架和Comparator回调来实现这么复杂的排序。上一章的例子中，Record类中恰好有field1、field2、field3三个字段。我们可以构造一个Comparator的实现类，实现如下的逻辑：先比较field1，如果不相等，那么返回比较结果；如果field1相等，那么比较field2，如果不相等，那么返回比较结果；如果field2相等，那么比较field3，并返回比较结果。
这样的解决方案，看起来很直观，也很简单。但是，这就是最终的解决方案了吗？如果我们有更多的排序条件组合呢？Record类有三个字段，每个字段都可以按照升序降序来排列，根据排列组合原理，可能的排序条件组

合多达几十种：
（1）按field1排序。（2）按field2排序。（3）按field3排序。（4）按field1，field2排序。（5）按field1升序，按field2降序排序.....
如果我们为每一种条件组合都写一个Comparator的实现类，那么，我们将需要几十个这样的类。这还是只有三个字段的情况下，如果字段个数继续增长，排序条件的个数将呈幂级数的增长。显然，为每一种条件组合写一个Comparator是不现实的。我们必须为这种组合条件排序找到一种通用的解决方法。我们来所有分析条件组合中变化的部分和不变的部分。
上面所有的排序条件中，不变的部分有3部分：（1）A.field1和B.field1的比较，（2）A.field2和B.field2的比较，（3） A.field3和B.field3的比较。
变化的部分有两部分：（1）升序和降序，（2）这三种比较条件的任意组合排列。
根据这段分析，我们引入两个类，ReverseComparator类和CompositeComparator类。ReverseComparator类用来解决字段的升序、降序问题。
CompositeComparator类用来解决字段的组合排列问题。
请注意，下面的Java代码是可以编译运行的代码。我用的是Java标准开发包中java.util包的Collections中的sort(List, Comparator)过程作为算法框架。相应的，我实现的Comparator也是Java标准开发包中的java.util.Comparator接口。
ReverseComparator的代码如下：

```
import java.util.Comparator;

public class ReverseComparator implements Comparator{
  /** the original comparator*/
  private Comparator originalComparator = null;

  /** constructor takes a comparator as parameter */
  public ReverseComparator(Comparator comparator){
originalComparator = comparator;
  }

  /** reverse the result of the original comparator */
  public int compare(Object o1, Object o2){
    return - originalComparator.compare(o1, o2);
  }
}
```

CompositeComparator的代码如下：

```
import java.util.Comparator;
import java.util.Iterator;
import java.util.List;
```

```
import java.util.LinkedList;

public class CompositeComparator implements Comparator{
  /** in the condition list, comparators' priority decrease from head to tail */
  private List comparators = new LinkedList();

  /** get the comparators, you can manipulate it as need.*/
  public List getComparators(){
return comparators;
}

  /** add a batch of comparators to the condition list */
  public void addComparators(Comparator[] comparatorArray){
    if(comparatorArray == null){
      return;
    }

    for(int i = 0; i < comparatorArray.length; i++){
      comparators.add(comparatorArray[i]);
    }
  }

  /** compare by the priority */
  public int compare(Object o1, Object o2){
    for(Iterator iterator = comparators.iterator(); iterator.hasNext();){
      Comparator comparator = (Comparator)iterator.next();

      int result = comparator.compare(o1, o2);
        if(result != 0)
          return result;
       else
     return 0;
} // end for
}
```

这两段代码有点长，我很抱歉。值得庆幸的是，这两段代码不算太长，还不至于占据太多的篇幅。

除了这两个包含类之外，我们还需要三个基本类——Field1Comaprator、Field2Comaprator、Field3Comaprator，用来做field1、field2、field3的比较。这三个类的功能一目了然，就不再写出来了。

利用这三个基本类和两个包含类，共五个类，我们就可以任意进行条件组合了。下面举例说明，如何组合这些

Comparator实现不同的排序条件。

例1：order by field1 asc, field2 asc

```
CompoiComparator myComparator = new CompoiComparator();
myComparator. addComparators(
new Comparator[]{new Field1Comaprator (),  new Field2Comaprator ()};
);

// records is a list of Record
Collections.sort(records,  myComparator);
```

例2：order by field1 desc, field2 asc

```
CompoiComparator myComparator = new CompoiComparator();
myComparator. addComparators(
        new Comparator[]{
new ReverseComparator(new Field1Comaprator ()),
new Field2Comaprator ()
};
);

// records is a list of Record
Collections.sort(records,  myComparator);
```

其余的条件组合，可以类推。在上述的例子中，我们又引入了两个简单的Design Pattern——Proxy Pattern（代理模式）和Composite Pattern（组合模式）。

ReverseComparator只包含一个Comparator对象，并保持了Comparator的接口方法。这种模式叫做Proxy Pattern（代理模式，有时候也叫做Delegate Pattern），其实就是把内部对象包装了一下，并执行了一些自己的操作。在这里，ReverseComparator就是把内部对象的比较方法给反转了一下。另外有一种Design Pattern叫做Decorator Pattern，其作用与Proxy Pattern类似，只不过是用继承来实现的。前面讲过，用继承来实现重用，实际上就固定了被重用的类，失去了多态性。我们不推荐。而且，Proxy Pattern完全能够做Decorator Pattern的工作，因此，我们不需要考虑Decorator Pattern，只需要知道Proxy Pattern这个简单、常用、又非常重要的的设计模式就行了。

CompositeComparator是Composite Pattern，能够把一堆基本的简单的功能块结合在一起，构成一个接口相同、功能却复杂得多的功能块。

Composite Pattern能够在各种条件组合的场景中使用，排序条件组合只不过是冰山一角。Java标准开发包中有一个java.io.File类，定义了很多文件操作方法。其中，有一个方法叫做listFile，能够根据不同的过滤条件选出符合条件的文件列表。listFile这个方法接受一个FileFilter接口类型的对象作为过滤条件。

这个功能有点象我们在文件管理器用通配符来搜索相应的文件。比如，我们想搜索JPG图片文件，我们就在文件搜索框中输入*.JPG。除此之外，我们还可以根据大小、时间等信息来搜索文件。

对于File类的listFile方法，我们可以定制不同的FileFilter，实现不同的过滤条件，比如文件时间在某个范围内；文件后缀名，文件名符合某种模式； 是目录，还是文件，等等。
我们可以应用上述的解决方法，实现灵活的过滤条件组合——用一个CompositeFilter类任意组合过滤条件，用一个ReverseFilter类作为排除条件来过滤掉那些我们不想要的文件。
CompositeFilter类的代码

```
import java.io.FileFilter;
import java.io.File;

import java.util.Iterator;
import java.util.List;
import java.util.LinkedList;

public class CompositeFilter implements FileFilter {

  /** in the filter list, every condition should be met. */
  private List filters = new LinkedList();

  /** get the filters, you can manipulate it as need.*/
  public List getFilters(){
    return filters;
  }

  /** add a batch of filters to the condition list */
  public void addFilters(FileFilter[] filterArray){
    if(filterArray == null){
      return;
    }

    for(int i = 0; i < filterArray.length; i++){
      filters.add(filterArray[i]);
    }
  }

  /** must meet all the filter condition */
  public boolean accept(File pathname) {
    for(Iterator iterator = filters.iterator(); iterator.hasNext();){
      FileFilter filter = (FileFilter)iterator.next();
```

```
    boolean result = filter.accept(pathname);

    // if any condition can not be met, return false.
    if(result == false){
      return false;
    }
   }

   // all conditions are met, return true.
   return true;
  }
}
```

ReverseFilter类的代码

```
import java.io.FileFilter;
import java.io.File;

public class ReverseFilter implements FileFilter {
  /** the original filter*/
  private FileFilter originalFilter = null;

  /** constructor takes a filter as parameter */
  public ReverseFilter(FileFilter filter){
    originalFilter = filter;
  }

  /** must meet all the filter condition */
  public boolean accept(File pathname) {
     return !originalFilter.accept(pathname);
  }
}
```

关于这两个类的组合用法，请读者自己尝试，这里不再赘述。

讲到这里，不知读者有没有注意到一个问题。在本书前面的章节中，我曾经强调过，在学习命令式语言时，任何时候，都不要忘记内存模型的概念。在描述C、C++语言特性的时候，我用了很多篇幅来描述复合结构和对象的内存映射结构。而我在用Java语言描述Design Pattern的时候，却有意无意地忽略了这个问题。这是为什么呢？难道Java语言高端到这种程度，都不需要考虑内存模型了吗？

不，不是这样的。下面我就要来讲述Java对象的内存映射问题。之所以推迟到现在来讲，是因为之前我们还没

有足够复杂的对象关系的例子。现在，我们已经接触到足够多的、结构足够复杂的Java对象。我们接触到了包含类、组合类、列表类以及数组结构。可以说，Java对象的大部分结构，我们都已经遇到了。

关于Java对象的内存结构，我们应该注意什么呢？Java对象的内存结构，和C++对象有什么区别呢？

在C++语言中，对象有可能分配在内存堆中，也有可能分配在运行栈中。当对象分配在内存堆中的时候，程序员必须自己负责对象内存的释放。当对象分配在运行栈上时，当前过程执行完毕之后，对象内存就会自动释放。因此，在C++中，对象的内存分配情况是比较复杂的。

在Java语言中，情况就简单得多。所有的对象都是在内存堆中分配的。而且，Java虚拟机负责对象的内存自动回收，程序员根本就不用考虑内存释放回收问题。对于C#、Python、Ruby等支持内存自动回收的语言来说，情况也是如此。

在前面的例子中，我们看到，一个Java对象中可以包含另一个Java对象（一个Java类中包含的成员变量的类型是另一个Java类）。这种包含关系，在内存模型中是如何映射的呢？

Java语言中有一个Object Reference（对象引用）的概念。在Java虚拟机中，所有的对象实例都有一个唯一的虚拟机内部的内存地址。这个内存地址，就是Object Reference。当一个对象包含另一个对象的时候，实际上，只是一个对象内部记录了另一个对象的内存地址（即Object Reference）而已。

我们可以想象一下，在一个布满了格子的巨大的内存架上，有两个对象分别占据了两块内存。其中一个对象的内存格中有一个格子记录另一个对象的内存地址。这就形成了一个表面上的包含关系。实际上，这两个对象完全是分立的，最多只能说是一种引用（Reference）关系。从这个意义上来说，Object Reference这个名词是非常贴切的。

Java对象数组也是如此，Java对象数组是一排连续内存格。每一个格子里面都放着一个内存地址，该内存地址记录着某个Java对象在Java虚拟机中的内存地址。

那么，是否存在真正意义上的内存结构的包含呢？是的，存在。比如，C语言的structure类型，C++的class类型，都能够真实的内存包含结构。但是，在Java中，我们只能做到“引用”。这种“引用”关系很有点像关系数据库里面的数据表之间的关系。所有的数据表都只能依靠关联数据来“引用”另一个表，而无法真正地“嵌套包含”另一个数据表。

在有些资料中，用C语言的指针概念来比拟Java的Object Reference概念。这种比拟不能说错。但是，我是坚决摒弃C语言指针概念的。所以，在本书中，我一律用内存地址这个概念来表述Object Reference。

前面讲述了Visitor、Proxy、Composite等常见的、重要的、基本的设计模式。另外，还有一个极其重要的、极其常见的设计模式，我们还没有讲到。那个设计模式就是Iterator Pattern。

实际上，我们在前面的代码例子中已经接触到Iterator了。我写CompositeComparator的时候，用数组来放置多个Comparator对象。但是，我在写CompositeFilter类的代码时，刻意用了LinkedList这个类。当我们需要遍历List数据结构是，我们就必须用到Iterator。还记得那段CompositeFilter类中的accept方法中的代码吗？

```
for(Iterator iterator = filters.iterator(); iterator.hasNext();)
```

上述代码中的Iterator的概念和用法，不仅在Java语言中极为常见，在其他的高级语言中也极为普遍。

注：上述代码中的Iterator用法极为冗长繁琐，在比Java更高级的语言中，在Java语言的高级版本，都针对这种用法进行了简化。这里采用这种冗长写法，是为了更清楚地表示出Iterator接口的具体方法。

Iterator Pattern的重要性，怎么强调也不为过。这也是本书重点讲解的Design Pattern之一。

但是，Iterator Pattern的实现，可不像前面讲述的那些基本Design Pattern那么简单。Iterator Pattern的实

现，涉及到复杂的机理和相关知识。在讲述Iterator Pattern之前，我们必须做好相应的知识储备。下一章，我们讲解线程相关的种种概念和模型。

1.10 《编程机制探析》第九章 线程

发表时间: 2011-08-29 关键字: thread, 编程

《编程机制探析》第九章 线程

本章开始讲述线程（Thread）的相关知识。线程（Thread）是计算机编程中的非常重要的概念，其概念与进程（Process）类似，都代表着内存中一份正在执行的程序。两者的共同点在于，两者都有自己的运行栈。两者之间的区别在于，进程拥有一份独立的进程空间，而线程没有。线程只能依附于进程存在。一个进程下面的多个线程，只能共享同一份进程空间。因此，线程和进程的主要区别，就在于共享资源方面。除此之外，两者的运行、调度，几乎都是一样的。

由于线程少了一份独立资源，比进程更加轻量，因此，程序员在编程时，更多地使用线程，而不是进程。只有在某些不支持线程的操作系统中，还有一些特殊要求的情况下，程序员才会使用进程。

线程本身的概念没什么可说的，当我们提到线程的时候，我们通常谈论的都是线程同步（Thread Synchronization）问题。

线程同步问题，可以说是关于线程的最重要的问题。我们甚至可以这样说，线程同步问题，是关于线程的唯一重要的问题。因此，关于线程的话题，基本上就是关于线程同步的话题。当我们耳朵里听到线程这个词，心里面就会自动把这个词补充为“线程同步”这个词组。

那么，线程同步，到底是怎么回事呢？前面说了，线程没有自己的资源，它只能和父进程，以及其他线程一起共享同一份进程资源。这就不可避免地设计到资源竞争问题。线程有可能和其他线程在同一个时间段访问一些共同的资源，比如，内存，文件，数据库等。当多个线程同时读写同一份共享资源的时候，可能会引起冲突。这时候，我们需要引入线程“同步”机制，即各位线程之间要有个先来后到，不能一窝蜂挤上去抢作一团。

同步这个词是从英文synchronize（使同时发生）翻译过来的。我也不明白为什么要用这个很容易引起误解的词。既然大家都这么用，咱们也就只好这么将就。

线程同步的真实意思和字面意思恰好相反。线程同步的真实意思，其实是“排队”：几个线程之间要排队，一个一个对共享资源进行操作，而不是同时进行操作。

因此，关于线程同步，需要牢牢记住的第一点是：线程同步就是线程排队。同步就是排队。线程同步的目的就是避免线程“同步”执行。这可真是个无聊的绕口令。

关于线程同步，需要牢牢记住的第二点是 “共享”这两个字。只有共享资源的读写访问才需要同步。如果不是共享资源，那么就根本没有同步的必要。

关于线程同步，需要牢牢记住的第三点是，只有“变量”才需要同步访问。如果共享的资源是固定不变的，那么就相当于“常量”，线程同时读取常量也不需要同步。至少一个线程修改共享资源，这样的情况下，线程之间就需要同步。

关于线程同步，需要牢牢记住的第四点是：多个线程访问共享资源的代码有可能是同一份代码，也有可能是不同的代码；无论是否执行同一份代码，只要这些线程的代码访问同一份可变的共享资源，这些线程之间就需要同步。

为了加深理解，下面举几个例子。

有两个采购员，他们的工作内容是相同的，都是遵循如下的步骤：

（1）到市场上去，寻找并购买有潜力的样品。
（2）回到公司，写报告。
这两个人的工作内容虽然一样，他们都需要购买样品，他们可能买到同样种类的样品，但是他们绝对不会购买到同一件样品，他们之间没有任何共享资源。所以，他们可以各自进行自己的工作，互不干扰。
这两个采购员就相当于两个线程；两个采购员遵循相同的工作步骤，相当于这两个线程执行同一段代码。
下面给这两个采购员增加一个工作步骤。采购员需要根据公司的“布告栏”上面公布的信息，安排自己的工作计划。
这两个采购员有可能同时走到布告栏的前面，同时观看布告栏上的信息。这一点问题都没有。因为布告栏是只读的，这两个采购员谁都不会去修改布告栏上写的信息。
下面增加一个角色。一个办公室行政人员这个时候，也走到了布告栏前面，准备修改布告栏上的信息。
如果行政人员先到达布告栏，并且正在修改布告栏的内容。两个采购员这个时候，恰好也到了。这两个采购员就必须等待行政人员完成修改之后，才能观看修改后的信息。
如果行政人员到达的时候，两个采购员已经在观看布告栏了。那么行政人员需要等待两个采购员把当前信息记录下来之后，才能够写上新的信息。
上述这两种情况，行政人员和采购员对布告栏的访问就需要进行同步。因为其中一个线程（行政人员）修改了共享资源（布告栏）。而且我们可以看到，行政人员的工作流程和采购员的工作流程（执行代码）完全不同，但是由于他们访问了同一份可变共享资源（布告栏），所以他们之间需要同步。
前面讲了为什么要线程同步，下面我们就来看如何才能线程同步。
线程同步的基本实现思路还是比较容易理解的。我们可以给共享资源加一把锁，这把锁只有一把钥匙。哪个线程获取了这把钥匙，才有权利访问该共享资源。
生活中，我们也可能会遇到这样的例子。一些超市的外面提供了一些自动储物箱。每个储物箱都有一把锁，一把钥匙。人们可以使用那些带有钥匙的储物箱，把东西放到储物箱里面，把储物箱锁上，然后把钥匙拿走。这样，该储物箱就被锁住了，其他人不能再访问这个储物箱。（当然，真实的储物箱钥匙是可以被人拿走复制的，所以不要把贵重物品放在超市的储物箱里面。于是很多超市都采用了电子密码锁。）
线程同步锁这个模型看起来很直观。但是，还有一个严峻的问题没有解决，这个同步锁应该加在哪里？
当然是加在共享资源上了。反应快的读者一定会抢先回答。
没错，如果可能，我们当然尽量把同步锁加在共享资源上。一些比较完善的共享资源，比如，文件系统，数据库系统等，自身都提供了比较完善的同步锁机制。我们不用另外给这些资源加锁，这些资源自己就有锁。
但是，我们在代码中访问的共享资源是代码中的共享数据呢？
读者一定又会说了。那有什么关系，我们也可以在那些数据上加锁呀。
那么，我们现在来考虑，这个锁，应该怎么加？
这还不简单。有人说了。在访问所有共享变量之前，都检查一下该共享变量的同步锁就好了。
那么，另一个问题就出现了。我们总得有个地方存放同步锁。那个同步锁放在哪里，放在共享变量的内部空间里面吗？还是放在共享变量的外面，比如在内存中的某一个地方？
显然，放在变量的内部空间是不合理的，应该放在外面的某个对应空间里。
这种设计方案看起来是可行的。事实上，确实存在这样的实现方案。比如，在Java语言中，有一个叫做volatile的关键字，就是加在变量名前面的。
volatile关键字可以加在任何类型的变量前面，可以加在Object Reference类型（即对象）的前面，也可以加在

简单类型（比如char、int、float、long、double等）的前面。
当变量是Object Reference类型（对象）的时候，在前面加volatile是没有意义的。因为对象的赋值只是一种Object Reference（内存地址）的赋值，并不引起对象内部结构数据的任何变化。而且，Object Reference的赋值，通常都是不需要同步的原子操作。
当变量是简单类型（比如char、int、float、long、double等）的时候，volatile能够工作得很好。当程序对这些变量进行存取读写的时候，Java虚拟机会对这些简单变量进行自动同步。
一般来讲，volatile关键字只对long、double等长类型才有意义。因为，短类型的操作基本上都是原子操作。而原子操作是一次操作就完成的，不需要分多次操作，因此也不需要进行同步处理。
现在，我们考虑复杂的情况。当共享数据是复合结构的时候，比如，当共享数据是一个包含关系复杂的对象的时候，我们该如何对这样的结构进行加锁？是总体加一个锁？还是为每一个属性都加锁？加锁的深度和层次又该如何控制？当对象关系进一步复杂化，又该如何处理？这种锁的控制相当复杂，几乎相当于用户系统中的权限管理了，早已经超出了虚拟机的任务范畴。即使虚拟机能够实现这么一套复杂的锁机制，那也是吃力不讨好的。首先，这样的锁机制很可能是大而无当的，大多数情况根本用不上。其次，这样的锁机制也不可能包括所有复杂的情况，总有一些特殊的需求无法满足。
那么，同步锁又该如何应对复杂数据结构的复杂情况呢？编程语言的设计者很聪明。他们把这个问题交给程序员自己解决。他们的设计思路很简单——把同步锁加在代码段上，确切的说，是把同步锁加在“访问共享资源的代码段”上。代码是程序员自己写的，可以写得非常简单，也可以写得非常复杂，可以应对各种简单或者复杂的需求。问题就这样解决了。
因此，我们一定要注意这一点，同步锁是加在代码段上的。有读者说了，不是还有个volatile关键字可以加在变量名前面吗？
没错。但是，在实际的应用中，volatile几乎没有用武之地。至少我在编程的过程中，从来就没有用过这个关键字。所以，我们可以有意地忽略volatile这个关键字。我们只需要学习和思考“同步锁加在代码段上”的线程同步模型。
现在，我们需要思考的问题是，代码段上应该如何加锁？应该加怎样的锁？
这个问题是重点中的重点。这是我们尤其要注意的问题：访问同一份共享资源的不同代码段，应该加上同一个同步锁；如果加的是不同的同步锁，那么根本就起不到同步的作用，没有任何意义。这就是说，同步锁本身也一定是多个线程之间的共享对象。
不同编程语言的同步锁概念和模型基本上都是一样的，只是用法上有些细微的差别。Java语言的同步锁机制并不算是最完善的，但是，却足够简单。而且，程序员可以通过Java语言本身的语法层次上的同步锁机制构造出自己需要的更复杂的同步锁模型。在我看来，Java语言的同步锁机制还是做得很不错。
Java语言中的同步锁的关键字只有一个，那就是synchronized。整个语法形式表现为

```
synchronized(同步锁) {
  // 访问共享资源、需要同步的代码段
}
```

其中“同步锁”是什么呢？就是一个Object Reference。任何一个Java对象，或者说，任何一个Object Reference，都可以对应一个同步锁。换句话说，任何一个Java对象实例，都可以被synchronized关键字包裹起来，承担起同步锁的任务。
为什么这么设计呢？是否有什么深刻的道理？在我看来，没有任何深刻的道理。这么设计，只是为了方便起

见。反正每个Object Reference都是唯一的内存地址，而同步锁也要求是唯一的，就顺手这么设计了。
这里尤其要注意的就是，同步锁本身一定要是共享的对象。我们下面来看一段错误代码。

```
... f1() {

Object lock1 = new Object(); // 产生一个同步锁

synchronized(lock1){
  // 代码段 A
// 访问共享资源 resource1
// 需要同步
}
}
```

上面这段代码没有任何意义。因为那个同步锁是在函数体内部产生的局部变量。每个线程调用这段代码的时候，都会产生一个新的同步锁。那么多个线程之间，使用的是不同的同步锁。根本达不到同步的目的。
同步代码一定要写成如下的形式，才有意义。

```
public static final Object lock1 = new Object();

... f1() {

synchronized(lock1){ // lock1 是公用同步锁
  // 代码段 A
// 访问共享资源 resource1
// 需要同步
}
```

你不一定要把同步锁声明为static或者public，但是你一定要保证相关的同步代码之间，一定要使用同一个同步锁。
再次重申。在Java里面，同步锁的概念就是这样的。任何一个Object Reference都可以作为同步锁。我们可以把Object Reference理解为对象在内存分配系统中的内存地址。因此，要保证同步代码段之间使用的是同一个同步锁，我们就要保证这些同步代码段的synchronized关键字使用的是同一个Object Reference，同一个内存地址。这也是为什么我在前面的代码中声明lock1的时候，使用了final关键字，这就是为了保证lock1的Object Reference在整个系统运行过程中都保持不变。
现在，我们来思考一个问题。同步锁应该加在谁的代码段上？所有访问共享资源的代码都需要加锁吗？是这样的。不过，我们可以遵循这样一种设计原则：尽量把某一个共享资源的访问方法都集中到一个类中，我们只对这个类中的代码进行加锁；其他代码要访问这个共享资源，都需要通过这个类；这种设计原则就可以有效地避免同步锁代码的扩散。
一些求知欲强的读者可能想要继续深入了解synchronzied(同步锁)的实际运行机制。Java虚拟机规范中（你可以在google用“JVM Spec”等关键字进行搜索），有对synchronized关键字的详细解释。synchronized会编

译成 monitor enter, ... monitor exit之类的指令对。Monitor就是实际上的同步锁。每一个Object Reference在概念上都对应一个monitor。

这种同步锁机制的设计，已经成为一种设计模式，叫做Monitor Object。我们继续看几个例子，加深对Monitor Object设计模式的理解。

```
public static final Object lock1 = new Object();

... f1() {

synchronized(lock1){ // lock1 是公用同步锁
  // 代码段 A
// 访问共享资源 resource1
// 需要同步
}
}

... f2() {

synchronized(lock1){ // lock1 是公用同步锁
  // 代码段 B
// 访问共享资源 resource1
// 需要同步
}
}
```

上述的代码中，代码段A和代码段B就是同步的。因为它们使用的是同一个同步锁lock1。

如果有10个线程同时执行代码段A，同时还有20个线程同时执行代码段B，那么这30个线程之间都是要进行同步的。

这30个线程都要竞争一个同步锁lock1。同一时刻，只有一个线程能够获得lock1的所有权，只有一个线程可以执行代码段A或者代码段B。其他竞争失败的线程只能暂停运行，进入到该同步锁的就绪（Ready）队列。

每一个同步锁下面都挂了几个线程队列，包括就绪（Ready）队列，待召（Waiting）队列等。比如，lock1对应的就绪队列就可以叫做lock1 - ready queue。每个队列里面都可能有多个暂停运行的线程。

注意，竞争同步锁失败的线程进入的是该同步锁的就绪（Ready）队列，而不是后面要讲述的待召队列（Waiting Queue，也可以翻译为等待队列）。就绪队列里面的线程总是时刻准备着竞争同步锁，时刻准备着运行。而待召队列里面的线程则只能一直等待，直到等到某个信号的通知之后，才能够转移到就绪队列中，准备运行。

成功获取同步锁的线程，执行完同步代码段之后，会释放同步锁。该同步锁的就绪队列中的其他线程就继续下一轮同步锁的竞争。成功者就可以继续运行，失败者还是要乖乖地待在就绪队列中。

因此，线程同步是非常耗费资源的一种操作。我们要尽量控制线程同步的代码段范围。同步的代码段范围越小越好。我们用一个名词“同步粒度”来表示同步代码段的范围。

在Java语言里面，synchronized关键字除了上述的用法，还可以加在方法名的前面修饰整个方法。比如。

```
... synchronized ... f1() {
  // f1 代码段
}
```

这段代码就等价于

```
... f1() {
  synchronized(this){ // 同步锁就是对象本身
    // f1 代码段
  }
}
```

同样的原则适用于静态（static）函数

比如。

```
... static synchronized ... f1() {
  // f1 代码段
}
```

这段代码就等价于

```
...static ... f1() {
  synchronized(Class.forName(...)){ // 同步锁是类型本身。虚拟机中独一份。
    // f1 代码段
  }
}
```

但是，我们要尽量避免这种直接把synchronized加在函数定义上的偷懒做法。因为我们要控制同步粒度。同步的代码段越小越好。synchronized控制的范围越小越好。

我们不仅要在缩小同步代码段的长度上下功夫，我们同时还要注意细分同步锁。比如，下面的代码：

```
public static final Object lock1 = new Object();

... f1() {

synchronized(lock1){ // lock1 是公用同步锁
  // 代码段 A
// 访问共享资源 resource1
// 需要同步
}
}
```

```
... f2() {

synchronized(lock1){ // lock1 是公用同步锁
  // 代码段 B
// 访问共享资源 resource1
// 需要同步
}
}

... f3() {

synchronized(lock1){ // lock1 是公用同步锁
  // 代码段 C
// 访问共享资源 resource2
// 需要同步
}
}

... f4() {

synchronized(lock1){ // lock1 是公用同步锁
  // 代码段 D
// 访问共享资源 resource2
// 需要同步
}
}
```

上述的4段同步代码，使用同一个同步锁lock1。所有调用4段代码中任何一段代码的线程，都需要竞争同一个同步锁lock1。

我们仔细分析一下，发现这是没有必要的。

因为f1()的代码段A和f2()的代码段B访问的共享资源是resource1，f3()的代码段C和f4()的代码段D访问的共享资源是resource2，它们没有必要都竞争同一个同步锁lock1。我们可以增加一个同步锁lock2。f3()和f4()的代码可以修改为：

```
public static final Object lock2 = new Object();
... f3() {
synchronized(lock2){ // lock2 是公用同步锁
  // 代码段 C
// 访问共享资源 resource2
```

```
// 需要同步
}
}

... f4() {

synchronized(lock2){ // lock2 是公用同步锁
  // 代码段 D
// 访问共享资源 resource2
// 需要同步
}
}
```

这样，f1()和f2()就会竞争lock1，而f3()和f4()就会竞争lock2。这样，分开来分别竞争两个锁，就可以大大较少同步锁竞争的概率，从而减少系统的开销。

1.11 《编程机制探析》第十章 线程同步模型

发表时间: 2011-08-29 关键字: 编程, 多线程

《编程机制探析》第十章 线程同步模型

上一章讲解的同步锁模型只是最简单的同步模型。同一时刻，保证只有一个线程能够运行同步代码。
有的时候，我们希望处理更加复杂的同步模型，比如生产者/消费者模型、读写同步模型等。这种情况下，同步锁模型就不够用了。我们需要一个新的模型。这就是我们要讲述的信号量模型。
信号量模型的工作方式如下：线程在运行的过程中，可以主动停下来，等待某个信号量的通知；这时候，该线程就进入到该信号量的待召（Waiting）队列当中；等到通知之后，再继续运行。
很多语言里面，同步锁都由专门的对象表示，对象名通常叫Monitor。
同样，在很多语言中，信号量通常也有专门的对象名来表示，比如，Mutex，Semphore。
信号量模型要比同步锁模型复杂许多。一些系统中，信号量甚至可以跨进程进行同步。另外一些信号量甚至还有计数功能，能够控制同时运行的线程数。
我们没有必要考虑那么复杂的模型。所有那些复杂的模型，都是最基本的模型衍生出来的。只要掌握了最基本的信号量模型——“等待/通知”模型，复杂模型也就迎刃而解了。
我们还是以Java语言为例。Java语言里面的同步锁和信号量概念都非常模糊，没有专门的对象名词来表示同步锁和信号量，只有两个同步锁相关的关键字——volatile和synchronized。
这种模糊虽然导致概念不清，但同时也避免了Monitor、Mutex、Semphore等名词带来的种种误解。我们不必执着于名词之争，可以专注于理解实际的运行原理。
在Java语言里面，任何一个Object Reference都可以作为同步锁。同样的道理，任何一个Object Reference也可以作为信号量。
Object对象的wait()方法就是等待通知，Object对象的notify()方法就是发出通知。
具体调用方法为
（1）等待某个信号量的通知

```
public static final Object signal = new Object();
... f1() {
synchronized(singal) { // 首先我们要获取这个信号量。这个信号量同时也是一个同步锁

    // 只有成功获取了signal这个信号量兼同步锁之后，我们才可能进入这段代码
    signal.wait(); // 这里要放弃信号量。本线程要进入signal信号量的待召（Waiting）队列

// 可怜。辛辛苦苦争取到手的信号量，就这么被放弃了

    // 等到通知之后，从待召（Waiting）队列转到就绪（Ready）队列里面
// 转到了就绪队列中，离CPU核心近了一步，就有机会继续执行下面的代码了。
// 仍然需要把signal同步锁竞争到手，才能够真正继续执行下面的代码。命苦啊。
```

...
}
}
需要注意的是，上述代码中的signal.wait()的意思。signal.wait()很容易导致误解。signal.wait()的意思并不是说，signal开始wait，而是说，运行这段代码的当前线程开始wait这个signal对象，即，把本线程加入到signal对象的待召（Waiting）队列里。
（2）发出某个信号量的通知
... f2() {
synchronized(singal) { // 首先，我们同样要获取这个信号量。同时也是一个同步锁。

// 只有成功获取了signal这个信号量兼同步锁之后，我们才可能进入这段代码
signal.notify(); // 这里，我们通知signal的待召队列中的某个线程。

// 如果某个线程等到了这个通知，那个线程就会转到就绪队列中
// 但是本线程仍然继续拥有signal这个同步锁，本线程仍然继续执行
// 嘿嘿，虽然本线程好心通知其他线程，
// 但是，本线程可没有那么高风亮节，放弃到手的同步锁
// 本线程继续执行下面的代码
...
}
}

需要注意的是，signal.notify()的意思。signal.notify()并不是通知signal这个对象本身。而是通知正在等待signal信号量的其他线程。

以上就是Object的wait()和notify()的基本用法。
实际上，wait()还可以定义等待时间，当线程在某信号量的待召队列中，等到足够长的时间，就会等无可等，无需再等，自己就从待召队列转移到就绪队列中了。
另外，还有一个notifyAll()方法，表示通知待召队列里面的所有线程。
这些细节问题，并不对大局产生影响，留给读者自己去研究了。
有了信号量这个利器，我们就可以处理比较复杂的线程同步模型了。
首先，我们来看一个比较简单的生产者/消费者模型。还是以Java代码为例。

public static final Object signal = new Object();
public static final char[] buf = new char[1024]; // 需要同步访问的共享资源

// 生产者代码
... produce() {

```
  for(... ) { // 循环执行

  synchronized(signal){
     // 产生一些东西，放到 buf 共享资源中

     signal.notify(); //然后通知消费者
     signal.wait(); // 然后自己进入signal待召队列
  }
  }
}

// 消费者代码
... consume() {

  for(... ) { // 循环执行

  synchronized(signal){

     signal.wait(); // 进入signal待召队列，等待生产者的通知

     // 读取buf 共享资源里面的东西

     signal.notify(); // 然后通知生产者
  }
  }
}
```

上述的生产者/消费者模型的实现非常简单，只用了一个信号量signal。这只是一段示意代码。
实际上的生产者/消费者模型的实现可能非常复杂。可以引入buf已满或者已空的判断，可以引入更多的信号量，也可以引入一个环状的buf链。但那些都是性能优化方面的工作，基本的信号量工作方式还是不变的。
生产者/消费者模型是典型的Coroutine（协程，即相互协作制约的线程）。而且，当消费者或者生产者线程进入待召队列的时候，当前的运行栈状态就暂时保存在系统当中，这种状况又是典型的Continuation。
注：Continuation是一种能够时而暂停、时而继续的模型，其重要特性是能够在暂停的时候，完整地保存包括运行栈在内的一切当前运行信息，从而保证之后的继续运行。有一段时间，这个概念模型颇为流行，还有不少实现。但我本人是不喜欢这个概念模型的。
我们完全可以用信号量机制自己实现Coroutine和Continuation。其实，那些在语法层面上支持Coroutine和Continuation的语言，内部实现原理也是采用类似的信号量同步机制。

比生产者/消费者模型稍微复杂一些的是读写模型。一份共享资源允许多个读者同时读取。但是只要有一个写者在写这份共享资源，任何其他的读者和写者都不能访问这份共享资源。
读写模型实现起来，不仅需要信号量机制，还需要额外的读者计数和写者计数。

```
public static final Object signal = new Object();
public static int readers = 0;
public static int writers = 0;

// 读者代码
... read() {

  for(... ) { // 循环执行

  synchronized(signal){
     while( writers > 0 )
        signal.wait(); // 如果有人在写，那么就放弃执行，进入待召队列

     // 能够到达这里，说明没有人在写

     readers ++ ; // 增加一个读者计数，表示本线程在读取
   }

// 进行一些读取操作

    synchronized(signal){
     readers --; // 读取完成，减少一个读者计数，表示本线程不在读取

     signal.notifyAll(); // 通知待召队列里面的所有其他线程
  }
  }
}

// 写者代码
... write() {

  for(... ) { // 循环执行

  synchronized(signal){
     while( writers > 0 || readers > 0)
```

```
        signal.wait();// 如果有人在写或读，那么就放弃执行，进入待召队列

      // 能够到达这里，说明没有人在写，也没有人在读

      writers ++ ; // 增加一个写者计数，表示本线程在写

// 进行一些写操作
      writers --; // 读取完成，减少一个读者计数，表示本线程不在写

      signal.notifyAll(); // 通知待召队列里面的所有其他线程
  }
  }
}
```

上述代码只是一段示意代码。实际应用中，人们通常抽取出来一个专门的读写同步锁。

```
interface ReadWriteLock {
  ... getReadLock();
  ... releaseReadLock();
  ... getWriteLock();
  ... releaseWriteLock();
}
```

具体的实现原理也是类似的信号量同步机制。

```
class RWLock {
  ... readers, writers;

  ... synchronized ... getReadLock() { // 相当于synchronized(this)
      ...
     while( writers > 0 )
        this.wait();  // 这里我们把RWLock对象本身作为信号量
       readers++;
  }

  ...synchronized ... releaseReadLock(){ //相当于synchronized(this)
       readers--;
   this.notifyAll(); // // 这里我们把RWLock对象本身作为信号量
  }
```

```
  ...synchronized ... getWriteLock(){// 相当于synchronized(this)
     while( writers > 0  || readers > 0 )
        this.wait();  // 这里我们把RWLock对象本身作为信号量

      writers++;
  }
  ...synchronized ...  releaseWriteLock(){// 相当于synchronized(this)
      writers--;
      this.notifyAll(); // // 这里我们把RWLock对象本身作为信号量
  }
}
```

具体用法是

```
public static final RWLock lock = new RWLock();

... read() {
  lock.getReadLock();
  // 读取
  lock.releaseReadLock();
}

... write() {
  lock.getWriteLock();
  // 读取
  lock.releaseWriteLock();
}
```

这种用法要求在执行一些处理之前，一定要执行某项特殊操作，处理之后一定也要执行某项特殊操作。这种人为的顺序性，无疑增加了代码的耦合度，降低了代码的独立性。很有可能会成为线程死锁和资源操作冲突的根源。

当时，这点一直让我不安，可是没有找到方法避免。毕竟，死锁或者资源操作冲突，是线程的固有问题。

很巧的是，正在我惴惴不安的时候，我得知了一个消息。Sun公司根据JCR，决定在jdk1.5中引入关于concurrency（并发）的部分。（那时候，Java还属于Sun公司。现在，已经换了主人。回想当年，不胜唏嘘。）

以下这个网址是concurrency部分的util.concurrent一个实现。非常好的信息。对于处理多线程并发问题，很有帮助。

http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html

包括Lock, ReadWriteLock, CurrentHashMap, CurrentReaderHashMap等类。JDK1.5引入了这些类，作为java.util.concurrent Package。

里面提供了一个ReadWriteLock类，标准用法如下。

Standard usage of ReadWriteLock:

```
class X {
  ReadWriteLock rw;
  // ...
  public void read() throws InterruptedException {
    rw.readLock().acquire();
    try {
      // ... do the read
    }
    finally {
      rw.readlock().release()
    }
  }
  public void write() throws InterruptedException {
    rw.writeLock().acquire();
    try {
      // ... do the write
    }
    finally {
      rw.writelock().release()
    }
  }
}
```

我们可以看到，ReadWriteLock同样要求调用的顺序——aquire()和release()。我对自己的例子增强了一点信心。

我又查看了WriterPreferenceReadWriteLock类，看到里面成对的方法，startRead()，endRead()；startWrite()，endWrite()。我的心情完全放松了下来。我的思路虽然粗糙，但大体的方向是正确的。

1.12 《编程机制探析》第十一章 Copy on Write

发表时间: 2011-08-29

《编程机制探析》第十一章 Copy on Write

Hash Table（哈希表，也叫散列表）是计算机编程中极为重要、极为常用的数据结构，其用法如下：我们可以用一个名字（name，或者叫做键值key）作为索引，把对应的内容存入到哈希表中；以后，我们可以提供对应的名字或者键值，从散列表把对应的内容取出来。用代码来表示就是这样：
hashTable.put(“myName”，myThing) // 以“myName”为键值，存入myThing。
fetched = hashTable.get(“myName”) //以“myName”为键值，取出之前存入的myThing。
散列表数据结构的特点是用空间换时间。散列表存入内容的时候，会用哈希（Hash）算法为每一个名字或者键值生成一个一定范围内的数字，这个数字对应散列表内部一个数组里的位置。这样，无论是存还是取，都可以快速地对应到数组里相应的位置，而不需要从头到尾地遍历数组。当然，这样做的速度确实是快了，但是，空间利用效率却不是那么高。数组里面很多地方很可能都是空的，只有对应上哈希算法结果的那些位置才有数据。这也是为什么哈希表也叫做散列表的原因——散列表可以更形象化地描述哈希表内部的数据分布状况。
不过，这些都是散列表的内部实现算法了。我们不关心这个。我们只在乎散列表的外在表现。我们只需要知道散列表的特性和用法就够了。
在Java语言中，有一个叫做Map（映射表）的接口，对应散列表的方法定义。HashMap类是Map接口的一个实现类，也是程序员经常使用的一个类。
Map的用法直观而简单，无需赘言。我们这里需要考虑的是，Map对象的线程同步问题。
当一个Map对象被多个线程同时访问的时候，就有可能产生访问冲突，甚至有可能破坏Map对象的内部结构。因此，这种情况下，我们必须用同步锁来解决线程访问冲突的问题。
当所有的线程都只是读取Map的时候，不会引起任何问题。但是，当有线程开始往Map对象里面写入数据的时候，就会引起问题了。因此，这是一个读写锁的控制问题。
上一章，我给出了读写锁的示意代码，并介绍了一个后来被JDK1.5引入的java.util.concurrent工具包，其中包括Lock, ReadWriteLock, CurrentHashMap, CurrentReaderHashMap等类。
我设想了一下，CurrentHashMap应该是采用ReadWriteLock实现读写同步。代码看起来应该像这个样子。

```
class CocurrentHashMap
{
Private Map map = null;
final ReadWriteLock rwLock = new .... ;
final Lock readLock = rwLock.readLock();
final Lock writeLock = rwLock.writeLock();

// decorate the map as concurrent
public CocurrentHashMap(Map m){
  map = m;
```

```
}

// all write method, like put, putAll, remove, clear
public void putAll(Map m){
  writeLock.lock();
  try{
    map.putAll(m);
  }finally{
    writeLock.unlock();
  }
}

// all read method. like get, containsKey, containsValue, entrySet()
public Object get(Object key){
  readLock.lock();
  try{
    return map.get(key);
  }finally{
    readLock.unlock();
  }
};
}
```

上述代码中用到了Proxy Pattern，包装并代理了内部Map对象的所有方法。我感觉，真正的CurrentHashMap的代码应该就是这种结构。但我看了CurrentHashMap 的源代码，发现不是这样。其中的实现比较复杂，把Table分成段进行分别管理。那个内部类 Segment extends ReentrantLock。 里面的 readValueUnderLock方法里面用了lock()和unlock()这对方法来加锁和解锁。

```
/**
 * Read value field of an entry under lock. Called if value
 * field ever appears to be null. This is possible only if a
 * compiler happens to reorder a HashEntry initialization with
 * its table assignment, which is legal under memory model
 * but is not known to ever occur.
 */
 V readValueUnderLock(HashEntry<K,V> e) {
   lock();
   try {
     return e.value;
   } finally {
```

```
        unlock();
    }
}
```

上述的代码用到了try {} catch() {} finally {} 的结构。这是Java语言中进行异常（Exception）处理的通用格式。异常处理也是现代语言中的常见的特性。关于这块知识，我本人没有什么特殊的心得，也不觉得有多么重要。因此，还是请读者自行弥补这方面的知识。

另外需要特别说明的一点事，上述的代码引自于JDK1.5的代码，其中用到了泛型语法。JDK1.5之后，开始支持Generic Programming（泛型编程）。我们看到的代码中的<K, V>就属于泛型代码，其中的K和V代表类型的名字。在真正编程中，K和V这两个类型名需要对应到具体的类型上。这就相当于把K和V这两个类型参数化了。因此，泛型有时候也叫做参数化类型（Parameterized Type）。

我对于泛型这种技术没有太多的感觉，谈不上有什么恶感，也谈不上有什么好感。关于泛型，我不打算专门抽出一个独立章节来讲解，既然在这里遇到了，就顺便讲讲吧。

我们首先需要弄清楚的一个问题是，泛型的目的是什么？简单而言，就是为了Type Dispatch（类型分派）。比如，一个通用的算法，可以应用到int、float等多种数据类型上。但是，我们又苦于没有办法把这些数据类型综合成一个通用的数据类型。我们只好为每一个数据类型，写一套基本类似的算法。为了消除这种重复，语言设计者们想出了一个办法，就是把算法中那些可变的类型作为参数，抽离出来。这就形成了“参数化类型”（Parameterized Type），即泛型（Generic）。

为此，C++设计者开发出了模板技术（Template），能够让程序员只写一份带有参数类型的算法代码。然后，编译器帮助程序员进行Type Dispatch，为每一种类型生成一套针对该类型的算法代码。

C++ STL就是一个完美应用Template技术的例子。但是，C++ Template实在是太强大了。不仅可以参数化类型，还可以参数化代码中的其他部分。C++ Template设计人员没有抵抗住这种诱惑，开始滥用C++ Template，为各种重用场景生成代码，结果导致C++ Template完全误入歧途。嗯。我是这么认为的。因为，后来出现的各种极为强大、极为诡异的C++ Template用法，我是完全看不懂了。

C++的Template技术，本质上都是一种预编译技术。这种技术只和编译器相关，和运行时一点关系都没有。编译器在编译期就根据类型参数生成一堆堆的源代码，然后编译成一堆堆的目标代码。这种技术实际上造成了代码膨胀。不仅是源代码膨胀，目标代码也跟着膨胀。

Java的泛型技术借鉴了C++ Template技术。不过，控制得还不错，只借鉴到了“参数化类型”的程度，还没有疯狂到参数化其他的部分。

Java的泛型技术有个名字，叫做“类型擦除法”（Type Erasure），其含义也是说，其泛型定义只在编译期起作用，编译之后，泛型（参数化的类型）会被擦除掉。

C#的泛型的实现技术与Java有所不同，表现出来的特性也有所不同。有哪些不同呢？这个问题，我们放到后面合适的时候再讲。因为这个问题涉及到一些运行期类型获取的知识，而现在我们还没有讲到这方面的知识。

好了，关于泛型的知识，就告一段落。我们回到之前的主题，继续对JDK1.5引入的java.util.concurrent工具包的源代码进行研究。

我又了解了一下开发包中的CurrentReaderHashMap类。它的描述是这样的，“A version of Hashtable that supports mostly-concurrent reading, but exclusive writing.”

但是，我阅读其源代码的时候，看到其中的read（get, contains, size ...）方法里面用到了synchronized关键字，仍然需要用到Java虚拟机提供的同步锁。既然read方法里面用到了同步锁，又如何保证可以同时读呢？这

点我想不明白，也没有深究。
多年来的编程经验已经让我明白，看别人的代码是非常费劲的，还不如自己想，自己写，反而更清楚明白。这也是为什么人们说“修改代码比重写代码更难”的原因吧。
我不再想那么多，而是开始自己构想一个读的时候不需要同步、写的时候才需要同步的Map实现。在这个构想中，普通的读写锁模型不足以满足我的需求。因为，在使用读写锁的情况下，每个线程每次读Map的时候，都需要检查一下读写锁，这也是一种开销。我希望能够把这种开销完全消除。
这种需求是完全符合现实的。因为，我们在工作的过程中，经常遇到如下的需求：
用一个Map存放常用的Object，这个Map的并发读取的频率很高，而写入的频率很低，一般只在初始化、或重新装装载的时候写入。读写冲突虽然很少发生，不过一旦发生，Map的内部结构就可能乱掉，所以，我们不得不为Map加上同步锁。
这时候，一种叫做Copy on Write的设计思路进入了我的视野。Copy On Write是这样一种机制。当我们读取共享数据的时候，直接读取，不需要同步。当我们修改数据的时候，我们就把当前数据Copy一份副本，然后在这个副本上进行修改，完成之后，再用修改后的副本，替换掉原来的数据。这种方法就叫做Copy On Write。
Oracle关系数据库的数据修改就采用Copy On Write的模式。Copy On Write模式对并发读取的支持很好，但是在并发修改的时候，会有版本冲突的问题。可能有多个线程同时修改同一份数据，那么就同时存在多个修改副本，这多个修改副本可能会相互覆盖，导致修改丢失。因此，Oracle数据库引入了版本检查机制。即增加一个版本号字段，来检测是否存在并发修改。相似的版本控制机制存在于CVS、SVN等版本控制工具中。
在我们的Copy On Write Map中，我们只需要让新数据覆盖旧数据就可以了，因此不需要考虑版本控制的问题。这就大大简化了我们的实现。
基本思路就是让读和写操作分别在不同的Map上进行，每次写完之后，再把两个Map同步。代码如下：

```
/*
* Copy On Write Map
*
* Write is expensive.
* Read is fast as pure HashMap.
*
* Note: extra info is removed for free use
*/
import java.lang.Compiler;
import java.util.Collection;
import java.util.Map;
import java.util.Set;
import java.util.HashMap;
import java.util.Collections;
public class ReadWriteMap implements Map {
protected volatile Map mapToRead = getNewMap();
// you can override it as new TreeMap();
protected Map getNewMap(){
```

```
return new HashMap();
}
// copy mapToWrite to mapToRead
protected Map copy(){
Map newMap = getNewMap();
newMap.putAll(mapToRead);
return newMap;
}

// read methods
public int size() {
return mapToRead.size();
}
public boolean isEmpty() {
return mapToRead.isEmpty();
}

public boolean containsKey(Object key) {
return mapToRead.containsKey(key);
}

public boolean containsValue(Object value) {
return mapToRead.containsValue(value);
}

public Collection values() {
return mapToRead.values();
}

public Set entrySet() {
return mapToRead.entrySet();
}

public Set keySet() {
return mapToRead.keySet();
}

public Object get(Object key) {
```

```
return mapToRead.get(key);
}

// write methods
public synchronized void clear() {
mapToRead = getNewMap();
}

public synchronized Object remove(Object key) {
Map map = copy();
Object o = map.remove(key);
mapToRead = map;
return o;
}

public synchronized Object put(Object key, Object value) {
Map map = copy();
Object o = map.put(key, value);
mapToRead = map;
return o;
}

public synchronized void putAll(Map t) {
Map map = copy();
map.putAll(t);
mapToRead = map;
}
}
```

这个Map读取的时候，和普通的HashMap一样快。写的时候，先把内部Map复制一份，然后在这个备份上进行修改，改完之后，再让内部Map等于这个修改后的Map。这个方法是同步保护的，避免了同时写操作。可见，写的时候，开销相当大，应该尽量使用 putAll() 方法。

到此为止，线程同步的主要模型都讲完了。但是，关于线程的话题却还远没有结束。

一开始的时候，人们觉得进程（Process）的开销太大，因此，创建了线程（Thread）的概念模型。随着技术的进步，程序并发运行的需求越来越多，越来越高。人们开始觉得，线程（Thread）的开销也太大了。这时候，人们开始追求更轻量级的运行单元。各种概念模型纷纷呈现。比如，出现了一种叫做纤程（Fiber）的概念。Thread是线的意思，Fiber干脆就是纤维的意思，比线还要细。

当然，还有一些语言，不在名词上玩花头，而是默默地做实际工作。比如，一种叫做ErLang的函数式编程语言中，就实现了对进程、线程模型的完全重塑。在ErLang中，并没有Thread的概念，而是直接叫做Process。不

过，这里的Process不再是对应操作系统的进程，而是，ErLang虚拟机自己管理的轻量级的运行单元。
在ErLang中，Process之间基本不需要同步。这是因为函数式编程语言的特性，在大多数正统的函数式编程语言中，变量近乎于常量，一生只能赋一次值，之后就不能再改变。用Java语言的术语来说就是，所有变量都是final的，不允许变量的值发生任何变化。既然变量不会变化，那还同步个啥子呢？有了这样的语法上的制约，ErLang就天生占了便宜，
那么，对于那些没有提供轻量级线程模型的语言（比如Java）来说，程序员又可以做那些努力来降低线程的开销呢？一种常见的做法就是线程池（Thread Pool），其工作原理如下：
首先，系统会启动一定数量的线程。这些线程就构成了一个线程池。
当有任务要做的时候，系统就从线程池里面选一个空闲的线程。然后把这个线程标记为“正在运行”。然后把任务传给这个线程执行。线程执行任务完成之后，就把自己标记为“空闲”。
这个过程并不难以理解。难以理解的是，一般来说，线程执行完成之后，运行栈等系统资源就会释放，线程对象就被回收了。一个已经完成的线程，又如何能回到线程池的空闲线程队列中呢？
秘诀就在于，线程池里面的线程永远不会执行完成。线程池里面的线程，都是一个无穷循环。具体代码如下：

```
Thread pooledThread {
  ... theTask .... // theTask成员变量，表示要执行的任务

  ... run() {
    while( true ) { // 永不停止的循环
      signal.wait(); // 等待系统的通知

      theTask.run(); // 执行任务
    }
  }
}
```

系统只需要调用 signal.notify() 就可以启动一个空闲线程。
在这段代码中，用到了一个JDK的Thread类。除此之外，JDK中还有一个Runnable接口，也是一个与线程开发相关的类型定义。关于Thread类和Runnable接口的具体定义和用法，请读者自行查阅。
关于线程的内容就到这里。有了线程的知识打底之后，我们下一章讲解一个非常重要的设计模式——Iterator Pattern。

1.13 《编程机制探析》第十二章 Iterator Pattern

发表时间: 2011-08-29 关键字: 设计模式

《编程机制探析》第十二章 Iterator Pattern

本章讲解一个极为重要、极为常见的设计模式——Iterator Pattern。关于Iterator的用法，实际上我们在前面的章节中有过一面之缘。Java语言开发包（JDK）中定义了一个Iterator接口，很清楚地描述了Iterator的行为。如果对这一点还不清楚的话，请查阅Iterator接口的相关文档和入门例子。
Iterator Pattern和Visitor Pattern这两个设计模式针对的问题领域是一样的。两者都是针对数据集合的遍历操作。两者面对的共同问题模型的组成部分如下：
（1）首先，有一个集合。这个集合有可能是列表（List），也有可能是树结构（Tree，一种常见的数据结构，关于树结构的基本知识，请读者自己补充），还有可能是更加复杂的数据结构，比如图（Graph，这是一种更加复杂的数据结构，我们几乎用不到，知道有这么个东西就行了）。
（2）其次，有一个针对该集合的算法，比如，排序，统计等。这类算法通常需要遍历集合中的所有元素。因此，为了简化问题的描述，我们就称这种算法为遍历算法（Traversal）。
（3）调用该算法的程序，我们称之为调用者。为了调用该算法获得某种结果，调用者必须访问到集合中的每一个数据元素。
以上是Iterator Pattern和Visitor Pattern两种设计模式的问题模型的共同部分。下面讲述两者之间的不同之处。两个设计模式的分歧点发生在第（3）条中。两者都需要访问集合中的每一个数据元素，但两者访问集合的方案截然不同，甚至是相反的策略，这就形成了两种不同的设计模式。
在Visitor Pattern中，调用者访问集合数据的方案，我们已经知道了。那就是派出一个访问者（Visitor），深入虎穴，亲身进入到集合算法的内部进行近距离的考察。整个访问流程，完全由集合算法方一手安排，访问多少个数据，什么时候访问完，完全由集合算法方决定。客随主便，入乡随俗，这就是Visitor一方面对的境况。
Iterator Pattern则恰好相反。调用者不需要深入到集合算法的内部，而是要求集合提供一个Iterator，作为数据访问接口。通过这个Iterator接口，调用者把访问数据的主动权完全掌握在手中。调用者想访问几个数据，就访问几个数据，想什么时候停止，就什么时候停止。
如果说，Visitor Pattern是一种被动模式，那么，Iterator Pattern则完全是一个主动模式。
Iterator的典型使用方法是：

```
iterator = traversal.getIterator();
item = iterator.next();
do some thing with item
```

Vistor的典型使用方法是

```
visitor = new Visitor(){
  .. visit(item) { do something with item }
};
traversal.traverse(visitor);
```

我们可以用一个拟人化的例子来进一步说明这两个设计模式之间的差异。
Visitor是调用者一方派出的甲方用户代表，深入到乙方集合算法公司内部。这一去，那可是深入虎穴，身不由己，Visitor完全由乙方（集合算法公司）安排访问行程。
Iterator则相当于乙方（集合算法公司）派出的一名乙方业务代表，来到甲方用户（调用者）的地盘上，听候甲方用户的调遣。甲方用户想要什么，就向Iterator这个乙方业务代表提出要求，然后，乙方业务代表根据要求做出应答。
为了加深读者对这两个设计模式异同点的理解，我们可以把这两种场景推到极端的情况。我们假设这么两个具体的场景。
Iterator的应用场景是这样：
我在商品定购目录上看到一个公司有我感兴趣的产品系列。于是我打电话给该公司，要求派一个销售代表来。销售代表上门之后，从包里拿出一个一个的产品给你看，我看了几个，没什么满意的，于是打了个哈欠说，今天就先到这里吧，下次再说。打发了销售代表，我就转身去做自己的事情了。Iterator可以被召之即来，挥之即去。
我的地盘，我做主。这就是Iterator Pattern的理念。
Vistior的应用场景是这样：
我在商品定购目录上看到一个公司有我感兴趣的产品系列。于是我上门拜访该公司，公司给我安排了一场产品性能展示，我看了几个之后，没有什么满意的，于是我说，我肚子疼，想先回去了。遇到好心的公司代表，当然说，身体要紧，慢走。遇到固执的公司代表，一定会说，对不起，我们公司有自己的工作流程，完成产品演示之前，产品厅的门锁是打不开的。我只好勃然大怒，吵吵嚷嚷（比如，进行抛出异常“throw exception”这个操作），期待能够杀出重围，这时候，假设该公司的保安系统非常严密（捕捉所有的Visitor抛出的异常“try and catch every visitor exception”），就会有几个保安跑过来，把我按在椅子上继续听讲。
入乡随俗，客随主便。别人的地盘，别人做主。这就是Visitor Pattern的理念。
读者可能会说，管它主动还是被动，反正都是遍历集合，有什么区别呢？这里我要说，区别可大着呢。主动与被动，是两种截然不同的地位。我们来看这样一个例子：遍历一棵树，搜集到前5个名字是Apple（苹果）的Node（即树结构中的每一个数据结点）；然后返回这5个Node；假设树遍历算法已经有了。
如果用Iterator Pattern来实现的话，非常简单。调用者可以向树结构提供的Iterator对象不断地发出请求，获取数据，发现Apple的时候就记录下来；当记录了五个Apple的时候，就停止向Iterator发出请求。
用Visitor Pattern该如何实现呢？首先，收集到五个Apple是非常简单的。我们可以在Visitor内部开辟一块空间来存放这个五个Apple。问题在于，当我们收集够了五个Apple之后，如何才能让Visitor退出来？当然，Visitor是无法自己退出来的，只能让树结构遍历算法把它放出来。那么，问题就变成，Visitor如何才能够告诉树结构遍历算法把它提前放出来？要知道，树结构遍历算法通常都是从头到尾遍历所有数据的。
如果树结构遍历算法提供了这种“提前结束”的交流机制，那么一切都好说。但大多数情况下，树结构遍历算法并不考虑这种特殊需求。那么，Visitor只好兜着五个已经收集到的Apple，随着树结构遍历算法继续遍历下去，直到最终结束。如果树结构很大的话，这是一种极大的浪费。
如果这个例子还不能让你理解主动权的重要性的话，我们再来看下一个例子：我们有有T1和T2两棵树。首先遍历T1的10个Node，如果发现Apple，那么摘下来，然后继续遍历，如果10步都没有发现Apple，那么切换到T2；遍历T2的规则也是如此，10步之内发现目标，就继续，否则就切换到T1。

Iterator Pattern实现起来很简单。相当于我是买方，情势是买方市场，我可以让两个公司的销售代表同时到我的公司来，我可以同时接待他们，让他们各自按顺序展示自己的产品。

Visitor Pattern怎么做？情势是卖方市场，我巴巴地跑上门去，看T1公司的产品展示，看了10个之后说，“请送我到T2公司的产品展示现场，我看10个之后，再回来”。这可能吗？

可能是可能，只是可能性非常小。我们如果真的遇到这种问题，非要用Visitor Pattern来解决的话，就得设计这样一个Visitor，一次访问来自于两个集合中的两个对应元素。而且，这两个集合的遍历算法也要拼成一个，两个集合之间还得想一个办法控制彼此的遍历步数。因此，这种可能只是理论上的可能。

一个用户可以同时使用多个算法的Iterator；但是用户的一个Visitor只能同时进入一个算法。

这就是两者核心理念的不同。

读者说了，既然Iterator这么好用，那我们都使用Iterator Pattern好了，至于Visitor Pattern，干脆就弃而不用了。

这个理想是好的，但现实是残酷的。Iterator Pattern比Visitor Pattern的实现难度大得多。

无论是Iterator Pattern还是Visitor Pattern，其遍历算法全都是集合一方提供的。如果集合数据结构是数组、列表等线性数据结构，那么，一切都好说，Iterator Pattern很容易实现，只需要提供一个结构体保存当前的遍历步骤（当前数组索引或者当前列表元素）就可以了。当然结构体本身还要包含对该数据集合结构的引用。用户每次调用iterator的next()方法，iterator就把数组索引或列表元素向后移动一下。读者可以自己尝试一下，实现数组结构和列表结构的Iterator，这并不难。

但是，如果集合数据结构是复杂结构的话，那么，Iterator Pattern的实现难度可就成几何级数增长了。我们这里所指的复杂结构，主要就是指树形结构。这是我们在应用开发中最常遇到的复杂结构。至于更复杂的数据结构，比如图（Graph），那几乎与日常开发工作无关。至少我没有遇到过Graph结构。因此，本书提到的复杂数据结构，就专指树形结构。

树形结构的遍历算法，一般的数据结构书籍中都有讲。由于树形结构是一层层结构类似的层次结构，其遍历算法通常都采用递归来实现。因此，树形结构遍历算法实现Visitor Pattern是相当容易的。只需要在原有的遍历算法中多加一个Visitor参数，并在代码中增加一条对Visitor方法的调用就可以了。

但是，树形结构遍历算法实现Iterator Pattern却是相当的难。Iterator必须自己维护一个类似于运行栈的结构，来保存当前的遍历状态。栈结构中需要保存本树枝之前遇到的所有父节点。

为什么Visitor Pattern就可以这么简单呢？因为Visitor Pattern可以很自然地采用递归算法。而递归算法是依靠计算机系统来帮助管理运行栈的。无论是压栈还是出栈，递归算法本身都不需要知道，只需要关心自身程序当前的操作。

Iterator Pattern就不同了。在Iterator Pattern中，你无从使用递归算法，你只能自己管理整个运行栈，所有的压栈、出栈操作都需要自己来管理。这种栈管理的算法，往往比树结构遍历算法本身还要繁琐。这种情况下，强行实现Iterator Pattern是得不偿失的。

千言万语一句话。Iterator是好的，但不是免费的。

我们可以用一个说法来比喻Visitor Pattern和Iterator Pattern的取舍。这个说法叫做“山不就我，我就山。”意思是，如果山不过来我这边，我就过去山那边。如果集合算法能够提供Iterator当然最好。如果集合算法不能提供Iterator Pattern的编程接口，只提供了Visitor Pattern的编程接口，那我们没办法，也只好按照人家的要求老老实实派一个Visitor进入人家的地盘。

计算机界中从来不缺乏聪明人。就有这么一些人不信邪，非要想出一种简单的方法来实现Iterator Pattern。凭

什么Visitor Pattern可以自然地在递归算法实现，而Iterator Pattern就不可以呢？凭什么Visitor Pattern可以让系统替自己管理运行栈，而Iterator Pattern就不可以呢？

聪明的人们把目光转向了Coroutine（协程，相互协作的线程）。Coroutine本来是一个通用的概念。表示几个协同工作的程序。比如，消费者/生产者，你走几步，我走几步；下棋对弈，你一步我一步。

由于协同工作的程序通常只有2个，而且这两个程序交换的数据通常只有一个。于是人们就很容易想到用Coroutine来实现Iterator。

不得不说，这是一个非常精当的思路。通过前面对Iterator Pattern的种种描述，我们可以很明显地看出，Iterator Pattern实际上就是一种生产者/消费者模型。Iterator就是生产者，因此Iterator有个别名叫做Generator。而调用Iterator的程序，就是消费者。

每次Iterator生产者程序就是等在那里，一旦用户（消费者角色）调用了iterator的next()方法，Iterator就继续向下执行一步，然后把当前遇到的内部数据的Node放到一个消费者用户能够看到的公用的缓冲区（比如，直接放到消费者线程栈里面的局部变量）里面，然后自己就停下来（wait）。然后消费者用户就从缓冲区里面获得了那个Node。

这样Iterator就可以自顾自地进行递归运算，不需要自己管理一个栈，而是迫使计算机帮助它分配和管理运行栈。

下面是一段来遍历二叉树（Binary Tree，最多只能有两个树枝的树形结构）的递归结构的示意代码。这段代码不是我写的，而是从网上的Coroutine资料中摘取的。

```
public class TreeWalker : Coroutine {
    private TreeNode _tree;
    public TreeWalker(TreeNode tree) { _tree = tree; }
    protected override Execute() {
        Walk(_tree);
    }
    private void Walk(TreeNode tree) {
        if (tree != null) {
            Walk(tree.Left);
            Yield(tree);
            Walk(tree.Right);
        }
    }
}
```

其中的Yield指令是关键。意思是，首先把当前Node甩到用户的数据空间，然后自己暂停运行（类似于java的thread yield方法或者object.wait方法），把自己当前运行的线程挂起来，这样虚拟机就为自己保存了当前的运行栈（context）。

有经验的读者可能会说，这不就是continuation吗？对。只是Coroutine这里多了一个产生并传递数据的动作。网上有具体的Java Coroutine实现，具体代码我也没有看过。我自己设想了一下实现原理，给出以下的示意代码.

```
public class TreeIterator implements Iterator{
```

```
  TreeWalker walker;
   // start the walker thread ..
  Object next(){
    walker.notify();
    // wait for a while so that walker can continue to run
    return walker.currentNode;
  }
}

class TreeWalker implements Runnable{
   TreeNode currentNode;

TreeWarker(root){
   currentNode = root;
}
void run(){
  walk(curentNode);
}

private void Walk(TreeNode tree) {
       if (tree != null) {
          Walk(tree.Left);
currentNode = tree;
this.wait();
Walk(tree.Right);
       }
   }
}
```

我们看到，Iterator本身是一个线程，用户也是一个线程。Iterator 线程一步步产生数据，并把数据传递给用户线程。这里，Iterator线程就是生产者线程，而用户线程就是消费者线程。

以上只是示意代码，我并没有深究过。因为，我虽然很欣赏用Corutine实现Iterator Pattern这种有创意的想法，但我本人并不赞同、也不会采用这种做法。因为线程是一种相对昂贵的资源，不应该这么任意地使用。线程同步则更是耗时耗力。我宁可自己维护一个Stack，也不愿意引入Coroutine这类Thread Control的方式来实现Iterator。

前面讲了很多实现原理方面的东西，现在，让我们看一些实际一点的例子。

树形结构是我们在应用开发中经常遇到的数据结构，XML就是最典型的实用例子。同样，关于XML的基本知识，请读者自行解决。

XML是一种层次化、结构化的树形文本结构。XML开发包中通常会提供两套操作XML文档的标准编程接口（即API，Application Programming Interface，应用程序编程接口）。一套叫做SAX，一套叫做DOM。

SAX是Simple API for XML的缩写。在Java语言中，程序员在使用SAX接口的时候，需要实现一个叫做ContentHandler的接口。其定义如下：

```
package org.xml.sax;
public interface ContentHandler
{
   public void setDocumentLocator (Locator locator);
   public void startDocument ()
throws SAXException;
   public void endDocument()
throws SAXException;
   public void startPrefixMapping (String prefix, String uri)
throws SAXException;
   public void endPrefixMapping (String prefix)
throws SAXException;
   public void startElement (String uri, String localName,
     String qName, Attributes atts)
throws SAXException;
   public void endElement (String uri, String localName,
   String qName)
throws SAXException;
   public void characters (char ch[], int start, int length)
throws SAXException;
   public void ignorableWhitespace (char ch[], int start, int length)
throws SAXException;
   public void processingInstruction (String target, String data)
throws SAXException;
   public void skippedEntity (String name)
throws SAXException;
}
```

我们可以看到，这个ContentHandler就是一个典型的Visitor，而且是带有甚多Type Dispatch的Visitor，其中定义了很多方法来处理不同的数据节点。比如，遇到文档开头怎么办，遇到文档结束怎么办，遇到文本怎么办，遇到属性（attribute，XML里面的概念）怎么办，等等。整个SAX开发包就是一个典型的Double Dispatch的Visitor Pattern。

关于SAX的具体应用例子，还是请读者自己去查阅。如果有兴趣的话，读者还可以参阅一下SAX开发包内部的具体实现，看看XML的SAX遍历算法是不是采用递归结构来实现的。

除了SAX之外，XML开发包提供的另外一套编程接口叫做DOM，是Document Object Model的缩写。DOM

编程接口把整个XML文档作为一个树形数据结构提供给程序员。程序员可以自己实现对这个树形结构的遍历算法。

值得一提的是，W3组织的网站上，DOM Level 2规范专门对DOM Traversal（DOM结构遍历）进行了定义。请参见如下网址：

http://www.w3.org/TR/2000/REC-DOM-Level-2-Traversal-Range-20001113/traversal.html

注：W3组织是一个定义各种互联网格式、规范、协议的网站，其中包括XML、HTML、HTTP等各种重要的规范和协议。对于Web开发人员来说，W3网站极为重要。

DOM Traversal规范中定义了DocumentTraversal, TreeWalker, NodeIterator, NodeFilter等多种接口。这些接口定义很有趣，值得研究一下。我大致看了一下。DOM Traversal主要是一个Iterator Pattern，TreeWalker, NodeIterator都是Iterator；但DOM Traversal同时也是一个简单的Visitor Pattern ——NodeFiter可以作为简单的Visitor被注入到Traversal算法里面，对遇的每个Node进行过滤。这个NodeFiter的方法名比较有意思，叫做accept，其功能类似于前面讲过的FileFilter（文件过滤器）的accept方法，起着过滤DOM Node的作用。

DOM Traversal规范的具体实现，我并没有找到。当然，我也没有花心思去找。我只对其设计思路感兴趣，对于其实现细节和使用方法并无兴趣。不过，我倒是知道另外一个实现了Iterator Pattern的XML编程接口——StAX。

StAX编程接口中，提供了一个叫做XMLStreamReader的接口，其中提供了next()和hasNext()等方法，正是起着Iterator的作用。从这个接口名中，我们可以得知Iterator的另一个别名——Stream（流）。现在，我们已经知道了Iterator的两个别名——Generator和Stream。

StAX并不是标准的XML编程接口，但却是一个真正实现了Iterator Pattern的XML编程开发包。

同样，StAX开发包的具体用法，请读者具体查阅。

StAX的内部源代码，我并没有看过。想来应该就是在XMLStreamReader的实现类中维护了一个栈结构，用来存放之前遍历过的所有父节点。至于用Corutine实现的可能性不大，毕竟，线程和线程同步的开销是比较大的。

有兴趣的读者可以自行去研究StAX的实现源码。

1.14 《编程机制探析》第十四章 关于方法表的那些事

发表时间: 2011-08-29

《编程机制探析》第十四章 关于方法表的那些事

上一章，我们讲解了静态类型语言和动态类型语言的特性对比。这一章，我们继续深入讲解静态类型对象和动态类型对象的内部机理——方法表（虚表）的内存结构以及实现机制。
我们先从静态类型语言中常见的语法陷阱开始。这些语法陷阱能够帮助我们更深入地理解静态类型对象的方法表的结构和特征。
让我们回到上一章开头的那段代码。那个Visitor有两个方法，visitA和visitB。其中visitA就表示访问A类型的数据元素，接受的参数也是A类型。visitB表示访问B类型的数据元素，接受的参数也是A类型。这种写法是我的偏好，也是我的建议。
我希望读者能够接受我的这种建议：除非为了达到某种特殊目的，绝对不要在自己的class定义中的同名方法。
为什么这么讲呢？这个问题要从头细细道来。在之前的章节中，我一直回避了这个问题。因为这个问题很烦人，完全是没有必要的误会引起的。世上本无事，庸人自扰之。不过，既然已经遇到了，就顺便澄清一下相关概念吧。
我们还是以Java语言为例，因为这种问题在Java等静态类型语言中表现得尤其明显。
首先，我们需要了解一个名词——方法签名（Method Signature）。
方法签名是个什么东西呢？方法签名就是一连串关于方法的信息的集合，其中包括如下部分：方法名，参数个数，每个参数的类型（按照顺序排列）。
这些信息构成了一个本类范围内独一无二的特征值，编译器依靠这些信息就能够准确地识别出一个方法，因此，这些信息叫做方法签名。
每一个类中都有一个方法表（虚表，VTable），里面的方法有些是本类自定义的方法，直接出现在本类的定义范围内；还有些方法来自于祖先类，并没有直接出现在本类的定义中，但是，那些方法仍然是该类的方法表中的方法，因此，也是该类的方法。
无论是来自于祖先类的方法，还是本类自定义的方法，都属于本类的方法。它们在地位上并没有任何区别。
注意，这里没有用子类和父类这两个词，而是用了后代类和祖先类这两个词，这是为了更准确的表述这个问题。请一定要注意这两个词表达的继承关系。
在同一个类的方法表中，不允许有相同方法签名的存在。当编译器发现同一个类的方法表中，存在相同的方法签名的时候，会根据不同情况，进行不同处理。
第一种情况是，后代类中定义了和祖先类同样的方法签名。编译器遇到这种情况时，就会直接在后代类的方法表中替换带掉祖先类的对应方法。这就保证了在同一个类的方法表内，只允许存在同一份独一无二的方法签名。这种现象叫做override。
这本来是一个很简单、很直观的概念。但是，相当多的技术资料拿override这个词做文章，引起很多不必要的误解和混淆。这个词的中文翻译也莫衷一是，乱七八糟。为了避免引起任何误解和混淆，我这里就把“override”直接意译为“后代类在方法表中替换祖先类方法”。这种翻译虽然很土很长，但是，至少不会引起误解和混淆。

第二种情况是，编译器发现了相同的方法签名，都是本类新定义的方法，而不是祖先类的方法。这种情况下，编译器就会报错。

事情到这里，本来就该结束了。一切都很简单，很明了。但是，好事者又搞出来一个猫腻，叫做overload。这个词简直就是无事生非的最佳典型。我对其是深恶痛疾，比指针概念还要痛恨。

什么叫overload呢？那就是在一个类的方法表中，存在两个或者两个以上的同名方法，其参数个数和参数类型不相同。

由于方法签名不同。因此，这些同名方法能够逃过编译器的检查，因而得以幸存。这种现象，就叫做overload。同样，关于这个词，也有各种容易引起混淆的中文翻译。我一概不采纳。我这里给出自己的翻译，“同名方法并存”。这个翻译也不怎样，但至少不会制造出更多的误解和混淆。

有些人可能会觉得overload写法很酷很方便。但我的建议是，为了避免各种不必要的麻烦，最好不要这样写。不仅参数个数相同的同名方法不要写，就连参数个数不同的同名方法也不要写。总而言之，同名方法就不要写。

为什么这么强调呢？因为overload可能引起的语言陷阱远超我们的想象。

关于“后代类在方法表中替换祖先类方法”（即override），有一种情境叫做covariance（协变）。这本来是数学中的术语。编程语言设计人员借鉴这个词语来表述一种特殊的情景。这种情景很难用语言直接描述，我们最好还是来看一个例子。

假设祖先类中有一个方法，其方法签名为 visit(Nubmer a)。继承于该祖先类的后代类定义了一个方法，其方法签名为visit(Integer a)。

现在，有一个问题。后代类的方法visit(Integer a)是否能够替换掉祖先类中的visit(Nubmer a)。

初一看，答案当然是不能。因为这两个方法的参数类型不同，方法签名自然不同，所以，这种情况就不能发生替换（即，不能发生override），最终的结果，就是后代类的方法表中存在两个同名方法，一个是visit(Number a)，一个是visit(Inter a)。这是overload的情景。

这个问题看起来很简单，就是一个overload的场景。但是，我们再细看看，就会发现一个有趣的现象。祖先类的方法visit(Number a)的参数类型是Number。后代类的方法visit(Integer a)的参数类型是Integer。而Number正是Integer的祖先类。在这种情况下，有些语言就允许后代类替换祖先类的方法。这种语法特性就叫做covariance。

因此，这种代码在不同的语言中有不同的表现。在支持covariance的语言中，就会产生“替换”（即override）的结果。在不支持covariance的语言中，就会产生“并存”（即overload）的情景。

Java语言不支持方法参数类型的covariance。在Java语言中，这种代码就会造成“并存”（即overload），即两个同名方法并列在后代类的方法列表中。

当我们使用这个后代类时，编译器很可能会遇到模棱两可的境况。比如，调用visit(new Interger(1))这个方法的时候，是调用visit(Number a)方法呢？还是调用visit(Integer a)方法呢？从参数类型匹配的角度来讲，两个方法都适用。这时候，编译器会根据上下文，尽量选择参数类型更接近的方法。我们来看一个例子。

```
class Parent{
  Object func(Number n) throws Exception{
    ...
  }
}
```

```
class Child extends Parent{
  Object func(Integer n) throws Exception {
    ...
  }
}
```

在这种极其变态诡异的“并存”（overload）情况下，年轻朋友们最着迷的游戏就是，你猜，你猜，你猜猜猜。

```
Number n = new Integer(0);
Integer i = new Integer(1);
Number nNull = null;
Integer iNull = null;

Child child = new Child();

child.func(n);
child.func(i);
child.func(nNull);
child.func(iNull);

child.func(null);
```

你猜，这几次都是调用哪个方法啊？ 其乐无穷。
这个不用我猜。是编译器需要猜。
编译器根据参数类型进行猜测。猜得到，它就猜。猜不到，它就让你编译不过。
上面的，编译器能猜得到。
编译器才不管你运行的时候，真正的类型是什么。
它只按字面意思理解。你定义的参数，对应的变量是什么类型。它就选什么类型。如果两个类型都匹配，那么就猜更具体的类型。
如果都猜不到，那么就让你编译不过，说你是 含糊不清，二义性，ambiguous
编译器冷笑着：小样儿，跟我玩，你还嫩。
事情还没有完，Java语言虽然不支持“参数类型”的covariance，但Java语言支持返回值和异常的covariance。比如，下面的代码中发生的是什么现象？“替换”（override）还是“并存”（overload）？

```
class Parent{
  Object func(Number n) throws Exception{
    ...
  }
```

```
}

class Child extends Parent{
  String func(Number n) throws SQLException {
    ...
  }
}
```

聪明的读者已经猜到了。上面的代码会产生“替换”（override）的现象。两个方法的返回值分别是Object和String，而Object是String的祖先类。两个方法的异常反别是Exception和SQLExpection，而Exception是SQLException的祖先类。

Java语言支持这两种covariance，所以，上面的代码会发生“替换”（override）的现象。

怎么样，这个游戏很好玩吧？如果你还是觉得这种写法很酷，很帅，那就当我什么也没说。确实有些人提倡用“同名方法共存”（overload）的方式编程，他们觉得这样更加清晰。我只能说，我的观点正好相反。

静态类型的语言陷阱说完了，我们来看动态类型的情况。首先，Javascript语言的情况一目了然。Javascript对象就是一个Hash Table，而Hash Table里面根本就不可能存在同名的东西。因此，往Javascript对象中填入任何同名的东西，唯一的结果就是“替换”（override），根本就不会发生“并存”（overload）的情况。因此，也不存在“并存”（overload）引起的种种编程陷阱。

我们再来看Python和Ruby的情况。在Python和Ruby这两门动态语言中，并没有静态语言中的方法签名（Method Signature）的概念。因为它们并不检查参数类型，甚至也不检查参数个数，因为在Python和Ruby中，参数个数也可以是不定的。因此，Python和Ruby的情况与Javascript是一样的。只有替换（override），没有并存（overload）。只要是同名方法，一律替换（override）。

关于方法签名，我还有话要说。在Java里面，我们可以用Reflection编程接口获取方法名、参数类型列表等构成方法签名的信息，但是，我们却无法获取参数名列表的信息。在动态语言中，情况正好相反。我们可以获得方法名和参数名列表，却无法获得参数类型列表。

从这里我们可以看出静态类型语言和动态类型语言的一个重要差异——静态类型语言关注的是类型，动态类型语言关注的是名字。那么，造成两者之间的这种差异的真正机理是什么？是什么原因造成了两者之间的这种差异？

为了回答这个问题，我们需要回顾一下方法表的内部结构。

在静态语言中，虚表（方法表）是一个紧凑的数组结构，所有的方法条目（即每个方法的代码在内存中的存放地址）都是一个挨一个紧凑排列的。编译期在编译方法调用语句的时候，实际上是该方法把虚表中的相对位置编入了目标指令中。当程序运行到该指令的时候，就会根据这个相对位置去查虚表中的对应条目，从而查到该方法代码真实的存放地址，然后调用那段代码，从而实现方法的调用。静态类型语言之所以关注“类型”，是为了获得某个方法在那个类型的虚表中的相对位置。即，静态语言真正关心的是“内存位置”。

在动态语言中，我们可以把方法表理解为一个Hash Table，每一个方法名都是Hash Table中的键值，其对应的内容是方法实现代码的真正存放地址。动态语言只关心对象的行为，而不关心对象的类型。Hash Table这种结构，能够使得解释器根据方法名迅速定位到对应的方法实现代码。

我们可以用两个简单的等式，来描述静态类型语言和动态类型语言的方法表的数据结构。

静态类型语言：方法表 = 数组

动态类型语言：方法表 = Hash Table
静态类型语言的调用方法的目标指令看起来可能是这样的。
invoke A.ArrayTable[1]
invoke B.ArrayTable[3]
其中，A和B是不同的类型。ArrayTable就是VTable。这里用ArrayTable这个名字是为了强调VTable的数组结构。
当然，这两句代码是只是用来示意的伪代码。实际上的目标指令里面不可能存在A.ArrayTable[1]这种写法。
动态类型语言的调用方法的目标指令看起来可能是这样的。
invoke A.HashTable[“f1”]
invoke B.HashTable[“f3”]
可见，静态语言类型的方法调用被翻译成了数字（位置），而动态语言类型的方法调用被翻译成了字符串（名字）。
这也正是静态类型语言运行效率更高的原因。
这也正是动态类型语言更加动态灵活的原因。
正是两者的方法表的数据结构的不同，才造成了两种语言的种种不同的特性。
看到这里，喜欢思考的读者可能会问了。Python和Ruby的方法表结构，真的同Javascript一样，是一个Hash Table吗？
实际上，并不一定如此。Python和Ruby的方法表结构只是“表现得”像一个Hash Table。其内部实现结构完全可能是一个紧凑的数组结构。它们只需要另外提供一套能够根据方法名查找到方法位置的机制就可以了。
那么，Javascript的方法表是否也可以做类似的空间优化呢？我的回答是，不行。那样做会得不偿失。因为，每个Javascript对象内部都维护一个独立的方法表，而且，这个方法表还是会随时改变扩充的。为这样一个随时可能变化的数据结构做空间优化，意义不大。
Python和Ruby中，同一种类型的对象实例，共享同一份方法表。那份方法表就存放在那个类型定义中，而且基本是固定不变的。对这种基本固定的数据结构做空间优化，才有意义。
我没有研究过Python和Ruby的解释器的实现代码。我不知道它们是否针对方法表进行了空间优化。毕竟，对于每一个类型来说，方法表只有一份，存放在类型定义中，所有的对象实例都共享这份方法表。即使不优化，也没什么大不了的。
不过，我几乎可以确定的是，Python和Ruby应该对每个类型的对象实例的数据属性表进行了优化。
在Java等静态类型语言中，每个类型除了有一份方法表（数组结构的虚表）外，还有一份数据属性表（即成员变量表），用来存放数据信息。同虚表一样，数据属性表也是紧凑的数组结构。方法表和数据属性表的区别在于。方法表是独立存在于类型定义之外的，类型定义里面只存放了方法表的地址，这也是为什么方法表被称为虚表的原因之一。数据属性表则是完完全全、结结实实、妥妥当当地存在于类型定义的内存空间内的。
一个类型的对象实例全都是以类型定义为模板构造出来的，其内存构造自然与类型定义完全一样。我们可以这样理解其内存结构，一个对象实例的内存结构就是一个数组结构，里面有N（N >= 1）个条目。前面N-1个条目都是数据属性表的条目，里面存放着数据信息。最后一个条目（第N个条目）中存放着一个方法表的地址。那个方法表存放在内存中的另外一个地方，那个方法表也是数组结构，里面存放着方法实现代码的内存地址。
为了简化起见，我们也可以这样理解静态类型对象的内存结构：一个对象内部有两个数组结构的表格，一个叫做数据属性表，一个叫做方法表。用Java伪代码描述就是这样：

```
class ObjectSpace{
  ArrayTable dataTable;
  ArrayTable methodTable;
}
```

由于Java语言是静态类型，可以在变量前面声明类型。因此，很适合用在这里表述数据结构。可见，在某些情况下——比如，在我们关心数据结构类型的时候——静态类型语言的描述性和可读性是相当好的，这方面远远超过了不用声明类型的动态语言。这就是，寸有所长，尺有所短。

以上这种写法只是为了方便理解。实际上，dataTable在ObjectSpace是一条条铺开的（这样做是为了提高数据属性的访问效率）。而methodTable在ObjectSpace中只是一个地址引用，表示一个公共的方法表的内存地址。

现在，我们再来看Python和Ruby的类型定义和对象实例的内存结构。同静态类型一样，动态类型的内存结构也分为两个部分，数据属性表和方法表。而且，这两个表格全都“表现得”如同Hash Table——解释器可以通过名字轻易地定位到对应的数据属性或者方法。动态类型对象的内存结构用Java伪代码描述就是这样：

```
class ObjectSpace{
  HashTable dataTable;
  HashTable methodTable;
}
```

同静态类型对象一样，每一个动态类型对象实例都拥有自己独有的一份dataTable，有多少个对象实例，就有多少份dataTable。而methodTable就只是一个地址引用，表示一个公用的方法表的内存地址。

由于每一个动态对象实例都拥有一份dataTable，而且，这个dataTable的表格大小，以及表格字段名称（即变量名）基本上是固定不变的。因此，对这个dataTable进行空间优化，很有意义。

我们可以采取前面描述过的优化思路，把真实的dataTable存放在一个紧凑的数组结构中，然后，提供一套类似于Hash Table的查询方法，能够把数据属性的名字映射到数组结构中的位置，这样，我们就能够访问到真正的数据属性。

至于methodTable，每个类型只有一份，优化亦可，不优化亦可。不过，如果优化了dataTable，还不如一起把methodTable一起优化了，反正优化方案都是一样的，何乐而不为。

Javascript对象的内存结构又是如何呢？我们同样可以用Java伪代码来表述：

```
class ObjectSpace{
  HashTable memberTable;
}
```

在Javascript对象里，数据属性和方法的地位是相同的，都放在同一张Hash Table里。而且，这个Hash Table的大小和内容是随时可以扩充和修改的。对这样一个结构进行空间优化，难度大且不说，而且没有多大的意义。

看到这里，一些求实的读者可能会问：你前面讲的那种空间优化思路，是一种通用的做法，还是你自己的凭空想象？

我得承认，这是我自己的臆测和猜想。不过，我的这种猜想是有依据的。

我们来看一个鲜活的真实例子：微软COM组件的双接口（dual interface）——IUnknown和IDispatch。

通过这两个接口的比较，我们可以清晰地看到静态类型方法表和动态类型接方法表的鲜明对比，而且，我们还

可以从IDispatch的接口设计中窥见上面讲述的空间优化思路。
COM是Component Object Model（组件对象模型）的缩写，是微软提出的一套组件编程规范。COM是一套二进制的组件接口规范。
什么叫做“二进制的”呢？意思就是，COM组件对象是真正的可执行的二进制代码（即目标代码，汇编代码，机器代码），而且，还把二进制的编程接口（这里指的就是二进制对象的虚表VTable）直接提供给程序员使用。
这种二进制方案的优点不言而喻，那就是运行效率。中间不存在任何通信协议的编码和解码，一切都和直接调用本程序内定义的对象方法一样。
COM组件提供的二进制接口（即组件类的虚方法表——VTable）叫做IUnknown，其中提供了三个基本方法。其中两个方法AddRef()和Release()是用来管理引用计数和内存释放的。
由于COM组件基本上都是用C++实现的，而C++不支持内存自动回收。所以，组件的提供方和调用方就需要通力合作，一起支持组件内存的正确释放。
由于这个机制与虚拟机内存自动回收机制有异曲同工之处。这里就花点笔墨解释一下。
调用方在获取组件之后，首先就要调用addRef()，表示自己引用了组件，敬告该组件：请不要随便释放组件内存，还有人用着呢。
调用方完成了对组件的使用之后，不打算再使用这个组件，就调用一下release()，对组件表示：我已经用完了，你愿意释放就释放吧。
组件内部管理着一个引用计数，表示当前用户的数量。当用户数量为零的时候，组件就可以释放自身内存了。
在这个过程中，所有的调用方和组件方都需要精密合作，一个地方出了差错，组件就不能正确释放了。
AddRef()和Release()这两个方法的用法基本上就是这样。我们来看IUnknown接口的最重要的方法——QueryInterface。这个方法用来询问组件是否支持某种接口。其用法有些类似于Java语言的instanceof关键字。
在Java语言里面，我们可以用instanceof来询问某个对象是否某种类型，当然也可以用来询问某个对象支持某种接口。比如：if(obj instanceof Comparable)这条语句就是询问obj这个对象是否支持Comparable接口。
COM组建的QueryInterface方法的用法有些相似。由于QueryInterface的真正用法比较繁琐，里面还涉及到组件ID和Windows注册表的知识，我这里就给出一个示意用法。我们这样写，
comObj.QueryInterface(“Comparable”)，意思就是询问这个组件是否支持Comparable接口。
通过QueryInterface这个方法，程序员可以获得自己真正需要的组件接口。
COM组件的实现方案非常丰富。一个COM组件可能包含其他的组件，并可以向用户提供这个组件的接口。这是代理模式（Proxy Pattern，Delegate Pattern）。
一个组件还可能包含多个其他的组件，并可以向用户提供多种组件接口。
有读者可能会抢答：我知道了，这是Composite Pattern（组合模式）。
但是，很遗憾，抢答错误。这种方案看起来像是Composite Pattern（组合模式）。但是，按照设计模式对应的场景和用法，这种方案还是代理模式。（哈哈，开个小玩笑。）
着迷于设计模式的读者不要失望，这种方案中还是存在一种新的设计模式的——Façade Patten。不过，这种设计模式没什么好说的。就是把原有的接口方法选出一部分，并据此重新定义一个接口（其中的方法签名也可能进行了一定得修改），然后再提供给用户。另外，我们可以从COM组件设计中找到更多的设计模式。比如，工厂模式（Factory Pattern）。在COM组建的用法中，组件都是由组件工厂来创建的。

我在前面的章节中，并没有提到工厂模式（Factory Pattern），原因在于，在现代的程序设计潮流中，工厂模式（Factory Pattern）不那么吃香了。人们不再热衷于从工厂中获取服务对象，而是倾向于使用IoC Container（对象关系构造容器）来提供程序需要的服务对象。

IUnknown是一个二进制接口（数组结构的虚表），其queryInterface方法获取的也是一个二进制接口，也就是说，也是一个虚表。如果调用方是C++语言的话，那么调用起这种虚表自然是得心应手。但是，如果调用方不是C++语言该怎么办？

如果调用方不是C++语言，可能会产生如下的问题：调用方语言是动态语言，无法直接使用虚表“VTable”结构；调用方语言与C++语言的数据类型不能直接匹配。

为了解决这些问题，COM组件规范又引入了一个新的接口IDispatch。这个接口主要提供了如下功能：能够根据一个名字找到组件虚方法表中的对应方法的ID；提供类型信息库，调用方可以根据这个类型信息库准备正确类型的数据。

IDispatch接口有一个叫做GetIDsOfNames的方法，能够根据方法名找到对应的方法ID（代表着该方法在虚方法表中的位置）。为了提高查询效率，GetIDsOfNames方法允许调用方程序一次查询多个方法名。

根据方法名获得了方法ID之后，调用方就可以调用IDispatch接口的invoke方法来调用对组件中对应的方法了。这个过程实际上就相当于通过方法名来调用方法。这是动态类型语言进行方法调用的做法。

通过GetIDsOfNames这个方法的定义，我们可以窥见其内部实现的常见策略。真正的方法表应该是紧凑的数组结构。GetIDsOfNames这个方法就是一种类似于Hash Table的查询接口，通过方法名，返回方法在方法表中（数组结构）的位置（数组下标）。

由此可见，我之前主观臆测的空间优化方案并非空穴来风，而是确有其事。

如果说IUnknow代表了静态类型的接口的话，那么，IDispatch就代表了动态类型的接口。不过，需要注意的是，IDispatch接口本身仍然是一个二进制接口，这个接口是COM组件实现的，而COM组件通常是C++语言实现的。因此，IDispatch接口通常也是C++语言实现的。

如果调用方是动态语言的话，那么，调用方并不能直接使用IDispatch接口，而是需要一个本语言的包装类，来包装对IDispatch接口的调用。这个包装类同时也负责对两种语言的数据类型的对应和包装。这也是异种语言之间相互调用的常用做法。

1.15 《编程机制探析》第十三章 动态类型

发表时间: 2011-08-29

《编程机制探析》第十三章 动态类型

在前面的章节中，我们已经几次遇到过Type Dispatch（类型分派）的场景了。在这种场景中，我们需要根据数据类型选择不同的行为。比如，我们来看下面这段典型Double Dispatch的Visitor Pattern的代码。

```
void traversal(Visitor visitor){
  // traverse each element
  ...
  currentElement = 当前元素
  if (currentElement的类型是A)
visitor.visitA(currentElement)
  else if (currentElement的类型是B)
   visitor.visitB(currentElement)
  ...
  ...
}
```

我们看到，在上述的遍历算法中，我们需要判断当前元素的类型来决定下一步执行的代码。这就程序在运行时动态获取当前数据的类型信息。

可不要小看动态获取类型信息这个需求，并不是所有的语言都能够很容易地做到这一点。至少在C++语言中，做到这一点就极其困难。

对应这个问题，C++语言中专门有一个术语，叫做RTTI（Run Time Type Information，运行时类型信息）。如果要在C++语言中实现RTTI的需求，则必须用一套RTTI相关的宏定义，在编译期间就为某个类型（class）产生额外的类型信息（通常是字符串类型），以便程序在运行时能够获取。在有些C++ Template技术中，还支持typeof关键字，同样是用来在编译期间产生类型信息。

Java语言实现运行时类型信息获取的需求，就容易了许多。Java语言中有一个关键字instanceof，可以判断某个数据是否某种类型。另外，所有的Java对象都支持getClass()方法，这同样可以用来在运行时获取类型信息。不仅如此，Java对象还支持Reflection（反射）特性。应用程序可以利用Reflection编程接口在运行时查看到某个Java对象内部的所有公开的成员变量和成员方法。

在某些语言中，比如，Python中，Reflection（反射）特性也叫做Introspection（内省）。从词义的角度来说，我觉得，Introspection这个词更加贴切一些。不过，由于Java比较流行，Reflection这个词也用得更多一些，甚至连Python也引入了这个词。我们也就从众，用Reflection这个词。

应用程序不仅可以利用Reflection编程接口获取对象的内部类型定义信息（如属性列表、方法列表等），还可以利用Reflection编程接口直接根据用一个字符串表述的方法名，直接调用对象中的对应方法。

这个功能在某些场景中特别有用处。比如，应用程序可以解析一段文本文件，获取其中的某个字段，并根据该字段作为方法名，调用某个对象中的对应方法。

有一个叫做OGNL的Java开源项目就利用这一点，实现一套强大的对象属性访问语言。OGNL 是 Object-Graph Navigation Language 的缩写，意思就是对象图导航语言，可以把这样的字符串“a.b.c.d”翻译成对应的Java代码调用——a.getB().getC().getD()。当然，OGNL是利用Reflection编程接口，一步步实现这种级联调用的。

C#语言提供了与Java语言类似的Reflection机制。这两门语言有很多近似的地方，也是竞争最直接的两门语言。两个阵营的程序员经常相互攻击对方语言的缺陷，彰显己方语言的优越性。

既然讲到了Reflection，我们就顺便为前面章节中“Java与C#泛型比较”的小话题做个总结陈词。

C#的泛型技术叫做“Reification”（类型具体化）。这种泛型技术很彻底，直接影响到编译后的虚拟机指令。运行时，对象实例内部仍然保持着泛型信息。程序员可以用Reflection编程接口获取泛型定义的信息。

Java的泛型技术叫做“Type Erasure”（类型擦除），与C++ Template是同源的，都是在编译器做文章。编译之后，“参数化类型”就被擦除掉了。运行时，Java对象实例内部没有泛型（类型参数）的信息，也不能被Reflection编程接口获取。

需要注意的一点是，Java泛型的“类型擦除法”并不是有意擦除的，而是“不得不” “被迫”擦除。由于Java虚拟机级别不支持泛型相关的指令和类型，Java编译器只能把“类型参数”擦除掉。

由于泛型信息的位置不同，可以分为两种情况。

一种情况是，class级别定义的泛型信息，即修饰class定义的泛型信息。由于class的对象将被实例化，而对象中并没有保留这部分泛型信息的地方，这部分泛型信息必须被擦除。

另一种情况是，class内部定义的泛型信息，即修饰成员变量或者成员方法的泛型信息。这部分信息与对象实例并不直接相关，而是随着class类型定义走的。因此，这部分信息可以保留下来。

事实上，Java编译器也确实是这样做的。成员变量和成员方法的泛型信息全都以字符串的形式保留在在class类型定义的常量池中。常量池英文叫做constant pool，用来存放类型中定义的一些常量，通常是字符串。

对于这部分泛型信息，JDK中新增了一些Reflection编程接口获取这些泛型信息。程序员可以用这些泛型信息相关的Reflection编程接口获取某一个class内部定义的成员变量或者成员方法的泛型信息。

从这里，我们可以看到“类型擦除法”的一个原则：在可能的情况下Java编译器还是会尽量保留尽可能多的泛型信息。

有兴趣的读者可以自己写一个Java泛型类，用泛型信息修饰分别修饰class、成员变量和成员方法，并写一段代码用泛型信息相关的Reflection编程接口获取这个类对象的泛型信息。编译并运行，查看结果。然后，再用javap反编译生成的class文件，查看其中的虚拟机指令和常量池部分。这样做了之后，你会对Java泛型的实现机制有更深刻的体会。

Java泛型的擦除法只在编译期做文章，这种做法优缺点，也有优点。缺点很明显，由于运行时，对象实例被擦掉了泛型信息，对手阵营（主要是C#阵营）称之为 “伪泛型”。

优点呢？第一点，类型擦除法只影响编译器，不影响虚拟机指令，不影响编译后的对象实例内存结构（当然，对class类型定义的常量池还是有一点影响的）。一些Java语言的忠实拥趸会振振有词地说，Java泛型对象实例化后比C#泛型对象实例化后省了那么一点点内存。

第二点，类型擦除法支持类型参数中的通配符（wildcard）。这里的通配符就是“？”。比如，Java支持这样的泛型定义 -- List<?>, List<? extends Number>, List<? super Integer> 。这是一种编译器的游戏，这方面，C#泛型的“Reification”（类型具体化）可就吃亏了，无法支持这种通配符。

我个人对于泛型的观感是这样——能省则省，能避免就避免。因为泛型实际上强化了编译期的静态类型检查。

而我个人更喜欢动态类型的灵活性和方便性。

在我个人看来，泛型的主要用武之地在于那些基本类型上，比如，int、float、double等。但是，现在的情况是，泛型却大量用于class类型上，比如，String、Integer等。对于这些class类型来说，我更倾向于用面向对象设计的方式来解决重用，而不是用泛型的方式。毕竟，归根结底，泛型也不过是一种代码膨胀的技术，无论是源代码还是目标代码的膨胀。

以上只是我的个人偏见，读者无须因此而受到影响，该怎么做还是怎么做，具体事情具体分析。现在，我们回到本章的主题——语言的动态性。

Java语言的Reflection机制为Java程序提供了更多的动态性和灵活性。不过，Java语言毕竟还是一门静态类型语言，其Reflection编程接口调用起来也是颇为麻烦。当我们在编程中遇到大量的动态类型、动态调用之类的需求时，我们就需要考虑更为动态的语言——比如Python和Ruby等动态语言了。

Python和Ruby之所以被称为动态语言，是因为它们的Duck Type（鸭子类型）的设计理念。

关于Duck Type（鸭子类型）的原话的大意是这样的：我不关心这个对象是不是一个鸭子，只要它能像鸭子一样呱呱叫，还能像鸭子一样摇摇摆摆地走路，我就把它当做一个鸭子——反正我只关心鸭子的这两个行为。

这段话的意思就是，Duck Type只关心对象是否支持某种行为，而不关心这种对象是否某种类型。

在Java之类的静态类型语言中，编译器首先关心的就是对象的类型。因为所有的行为方法都是在类型中定义的，只要类型一致，行为自然是一致的。因此，Java等静态类型语言的类型检查更严格一些。

在Python、Ruby等动态语言中，无论是编译期，还是运行期，都不进行任何检查，而是直接在该对象的内部方法表（其内存结构类似于C++、Java对象中的虚表VTable）中查找对应的方法，看看该对象是否支持这种方法。如果支持，那么好，这条代码就是合法的。

比如，如果Python、Ruby解释器遇到这样的代码，a.quark()。解释器并不去查看a对象的类型，而是直接到a对象的方法表中查找是否存在一个名字叫做quark、参数为空的方法。如果存在，那么这条代码通过，可以执行；如果不存在，报错。

由于Python和Ruby只关心方法列表，而不关心类型，这又引出了Python和Ruby的又一个强大的语法特性——mixin（混入）。利用Python和Ruby的这种语法特性，我们可以轻易地把外面定义的方法混入到一个类的方法列表中。这个功能不仅完全实现了C++多重继承的目，甚至有过之而无不及。当然，同C++多重继承一样，mixin也会引起类似的所引起的负面效果，比如，不小心引入了同名方法可能会引起一些不期望的结果，等等。不过，对于动态语言来说，这点负面效果的影响要远远小于C++的多重继承。

Python和Ruby的Reflection机制（或者叫做Introspection）比Java更加强大灵活，更加简单易用。使用Python和Ruby，我们可以轻易地实现Java语言难以达到的动态性和灵活性。

不过，Python和Ruby还不算是动态类型的极致。有一门常见常用的语言，比Python和Ruby还要动态。

这门语言是什么？聪明的读者已经猜到了。没错，就是Javascript。

在Javascript语言中，每一个对象都是一个类似于Hash Table（哈希表，也叫散列表）的结构。我们可以在其中任意添加属性和方法。比如，我们可以这样定义一个对象。

```
var myObject = {
  m1: “some data” // 成员变量
  f1: function() {...} // 成员方法
  f2: function() {...} // 成员方法
}
```

我们还可以一步步填充这个对象。比如，

```
var myObject = {} // 空对象
myObject.m1 = “some data”;
function f1() { … }
myObject.f1 = f1;
myObject.f2 = function() {…}
```

为了更清楚地表现Javascript对象的内部结构，我们还可以这样写。

```
var myObject = {} // 空对象
myObject[“m1”] = “some data”;
function f1() { … }
myObject[“f1”] = f1;
myObject[“f2”] = function() {…}
```

在上述代码中，[ ]这对方括号就表示Javascript对象（Hash Table）中的某个名字对应的值。

Javascript对象的Reflection用法非常简单直观。for (var member in myObject) 就可以遍历myObject中的所有成员。

另外，我们还可以把Javascript对象的构造函数写成类似于Java语言的风格。

```
function myObject(){
  this.m1 = "some data";
  this.f1 = f1;
  this.f2 = f2;
}
function f1() { … }
function f2() { … }
```

由此可见，Javascript对象内部的结构是多么的灵活，属性表中的每条属性都可以任意替换和填充。这才是真正的开放类型。

当然，Javascript是一元语言，不存在类型和实例的区别。而在C++、Java、C#、Python、Ruby等二元语言中，类型和实例是两个明确分开的概念。

如果要讲类型的话，每一个Javascript对象都是一个潜在的扩展类型。只要属性替换和填充发生了，一个新的扩展类型就产生了。

每个Javascript对象都有一个最基本的原型（prototype）对象，每次新生成一个Javascript对象，实际上就是把那个原型对象复制了一遍。这就像把一个Hash Table完全复制一遍一样。

如果你需要改变所有新对象的某个属性，你只要修改原型（prototype）对象的对应属性就可以了。之后，从那个原型复制出来的所有新对象的对应属性都会跟着改变。

Javascript访问原型对象相当容易。myObject.prototype就可以获得myObject的原型对象。myObject就是从myObject.prototype这个对象复制出来的。

Javascript并没有类型的概念，最初的原型也都是对象实例。这种语言叫做一元语言。与之相对的，有类型和实例概念的语言，比如，C、Java、C#、Python、Ruby等语言，叫做二元语言。

Javascript比Python、Ruby更具有动态性，主要就是因为它的“一元类型系统”的特性。

Python和Ruby是不关心类型，但毕竟还有类型的概念。但是，在Javascript这种一元语言中，干脆就取消类型的概念，所有的对象实例都自成一个类型，从而获得了更大的动态性。

Javascript语言把“Duck Type”的思想发挥到了极致——我们不关心它是不是鸭子，我们只关心它会不会像鸭子一样叫，我们只关心它会不会像鸭子一样摇摇摆摆走路。啊，不，事实上，我们根本就不关心它是否像鸭子，因为在我们的系统中根本就没有鸭子这个东西。我们只关心它会不会叫，会不会摇摇摆摆走路。

读者可能会问了，既然一元类型系统比二元类型系统更加灵活，那么为什么大部分语言都采用二元类型系统的设计思路？为什么不能像Javascript一样，取消类型的概念呢？所有的原初“类型”都是“原型”对象，这样不是很好吗？

实际上，我也是这么想的。我更欣赏Javascript这种类型系统设计思路。在我看来，之所以大部分语言不采取这种设计思路，是出于两种原因。

第一个原因说起来有些无聊，那就是，Javascript语言的这种一元类型系统太动态、太灵活了，使得类型检查排错成为几乎不可能的事情，大大增加了代码的风险性。

第二个原因是空间原因。在二元类型系统中，所有的方法都存放在方法表中。而一个类型的方法表只存在一份。无论创建多少个对象实例，都共用本类型的那个方法表。而在一元类型系统中，每个对象实例都有一份自己的方法表。这就造成了空间浪费。对象实例越多，空间浪费就越大。当然，正是因为这种空间浪费的设计，才使得Javascript如此灵活强大。

太极拳理论中有一句话，“一动则周身无有不动，一静则百骸皆静”。

Javascript就是一种动态到极致的语言，一动则周身无有不动，没有什么不能动的，属性内容可以动，方法内容可以动，属性名可以动，方法名也可以动，而且什么时候都可以动。

静态语言就正好相反，虽然还不到“一静则百骸皆静”的程度，但也差不多了。这里不能动，那里也不能动。这里可以动一点点，但是不可以大动。而且，静态语言提倡“一动不如一静”，最好还是不动。不动最安全。

动态好，还是静态好，这一直是个争议不休的话题。动态方的观点是，动态语言灵活强大，代码简洁，生产力高。静态方的观点是，静态语言类型安全，出错率低，运行效率高。总之，是公说公有理，婆说婆有理。

我的观点是，从重用的角度来说，动态语言确实比静态语言要强。在动态语言中，我们可以很容易地包装或者修正对象的外在行为特性，从而使得不同类型的对象实例“表现得”好像同一个类型一样。通过这种方式，我们可以让不同类型的对象适应同一个算法，从而达到重用的目的。

由于静态类型语言需要进行类型检查，要做到这一点是非常不易的。在静态语言中，想让不同的类型“表现得”像同一个类型一样，只有一个方法，那就是为这些不同的类型抽出一个共同的通用接口。这个要求有些太过分，不太现实，因为我们不可能去修改开发包中那些基本类的定义。因此，在静态语言中，想让不同的类型适应同一个算法，最现实的方法的就是使用泛型。而泛型实质上是一种代码膨胀技术，或者说代码生成技术。我本人对于代码生成技术是没有太多好感的。这也造成了我对泛型没有太多好感。而泛型是静态类型系统的衍生物。这就使得我更喜欢动态类型，而不是静态类型。

这些都是主义之争，无关实际工作。我们还是少谈些主义，多谈些问题。

代码设计有这么一条原则，遇到新需求的时候，尽量添加新代码，而不是修改旧代码。能够做到这一点的，就是好设计。Good Smell（好味道）。做不到这一点的，就是坏设计。Bad Smell（坏味道）。

在本章开头的代码中，存在着一些判断当前元素类型的条件语句。那些条件语句里面的判断条件都是写死的，无法改变。这种代码叫做Hard Coded（硬编码）。当我们在集合中增加一种新的数据类型的时候，我们就不得不修改遍历算法中的代码，增加一条对应该类型的条件语句。相应的，我们还需要在Visitor里面增加一个对应

的访问该类型元素的方法。这就是Double Dispatch的Visitor Pattern的Bed Smell所在。这就是我为什么不愿意费工夫描述这个设计模式的原因。

现在，我们想一个办法，去掉这种设计中的Bad Smell。首先，我们要做的是，去掉遍历算法中的if else语句。这点做起来其实很简单。我们只需要把Visitor里面的方法简化到一个就好了。Visitor里面就只有一个方法，visit(Object currentElement)。那么，遍历算法中的类型判断语句放到哪里呢？当然是放到Visitor的方法里。

visit(Object currentElement)里面的代码看起来就是这样：

```
 if (currentElement的类型是A)
....
 else if (currentElement的类型是B)
   ....
```

这样，我们就成功地消除了遍历算法中的Bad Smell。这是理所当然的。因为所有的Bad Smell全都转移到Visitor里面了。不管怎么说，我们总算是进了一步。Bad Smell集中在一个地方，总比集中在两个地方好。

现在，所有的Bad Smell都在Visitor里面了。我们可以集中精力对付它。首先，我们要做的是，去掉visit方法中的if else语句。

读者会说了，这真是没事找事做。费劲巴拉地把if else移过来，结果还是要消除它。

可不要小看这一步。在Visitor中消除if else，可比在遍历算法中消除它简单多了。

消除if else的利器是什么？就是多态。可是，我们这里遇到的参数类型（元素数据类型）的多态，而不是Visitor本身的多态，我们多态不起来。

现在的问题是，我们如何才能实现多态的Type Dispatch（类型分派，即参数类型的分派处理）。

我们可以从前面描述的Javascript对象结构中获得灵感。Javascript对象是什么结构？对了，是个Hash Table。我们可以自由地在增加和修改其中的属性。这是我们所追求的目标。

参照这种结构，我们也可以在Visitor内部维护一个Hash Table。我们可以在这个Hash Table任意添加某种类型的处理程序。当Visitor遇到一个数据元素的时候，先从Hash Table里面查找该类型，如果找到了，就调出该类型对应的处理程序进行处理；如果没找到，那就没办法了，只能为该类型写一个处理程序，然后添加到Hash Table中。

我们看到，在这个设计方案中，只有添加，没有修改。这正是我们追求的Good Smell。

示意代码如下：

```
interface Visitor {
 void visit(Object currentElement);
}
class TypeDispatchVisitor {
 Map handlerMap = new HashMap();
 public void setTypeHandler(Class t, Visitor handler){
   handlerMap.put(t, handler);
 }

 public void visit(Object currentElement){
   Visitor handler = (Visitor)handlerMap.get(currentElement.getClass());
```

```
    handler.visit(currentElement);
  }
}
```

这个TypeDispatchVisitor的用法是这样。

```
TypeDispatchVisitor dispatcher = new TypeDispatchVisitor();
dispatcher.setTypeHandler(A.class, new AVisitor());
dispatcher.setTypeHandler(B.class, new BVisitor());
traversal(dispatcher);
```

其中，A和B都是元素类型，AVisitor和BVisitor是处理这两种类型的两个Visitor类。

我们看到，这种设计完美地解决了之前的问题，去除了原有设计中的Bad Smell。当然，这种设计又引入了一点新的Bad Smell——类型强制转换（Type Cast）。AVisitor和BVisitor在处理相应数据类型的时候，需要把参数从Object类型强制转换成对应的A类型或者B类型。不过，这点Bad Smell比之前的Bad Smell好闻太多了。

这种设计如果用动态语言来写的话，那就连类型强制转换（Type Cast）的Bad Smell都没有了。这也是我倾向于动态语言的一个原因。至少在程序设计方面，动态语言可以设计得更加漂亮。

1.16 《编程机制探析》第十五章 递归

发表时间: 2011-08-29

《编程机制探析》第十五章 递归

前面章节讲述的基本上都是命令式语言（Imperative Language）的编程模型。关于命令式编程（Imperative Proramming）的重要概念和模型，我们基本都涉及到了。后面的章节将开始讲述另一种编程模型——函数式编程（Functional Programming）的内容。

在讲述函数式编程（Functional Programming）之前，我们需要做一些知识上的储备——递归（Recursion）。

其实，我们在前面的章节中，已经遇到过递归了。在前面的“Iterator Pattern”章节中，树形结构的遍历算法代码就是递归形式的。递归是一种过程自调用的一种代码形式，看起来没什么大不了的。但是，递归对于函数式编程来说，却是极为重要的一门必须掌握的代码形式。

为什么这么说呢？因为，在函数式编程中，递归是实现代码多次重复运行的唯一形式。函数式编程只支持递归，不支持循环。

在命令式编程中，代码多次重复运行有两种形式。一种是循环形式，比如，for、while等语法形式；一种是递归形式。

在命令式语言的实际使用中，循环比递归用得多得多。原因很简单。过程内部可以直接定义循环体，两者在同一个过程的作用域内，循环体代码可以直接使用本过程作用域内定义的局部变量，用起来很方便。但是，递归涉及到的过程调用，相当于进入到另一个过程的作用域，两个过程的局部变量并不能通用，两个过程只能依靠参数和返回值来进行数据交流，写起来麻烦许多。

在某些情况下，比如树形结构的便利算法中，递归会很方便，比循环还要方便。但是，大部分情况下，循环要比递归方便。

既然循环更加方便，为什么函数式编程不支持循环呢？函数式编程不是不想支持循环，而是支持不了。

为什么函数式编程支持不了循环呢？因为，在函数式编程语言中，变量表现得如同常量一般，非常专一，一生只能被赋值一次，之后就不能再改变了。从一而终，一生不二嫁。这就是函数式变量的忠贞品质的真实写照。

而循环是什么？循环本质上是基于Iterator Pattern的，需要一个Iterator来提供当前元素，并保存集合遍历的当前状态。所以，循环形式有个名字叫做Iterative（有些资料翻译成迭代）。

最简单的情况，循环也至少需要一个计数器，来保存当前循环的步数。无论是Iterator，还是计数器，都是需要与时俱进、随时改变的。终生不变的函数式变量根本就无法满足成为Iterator或者计数器的要求。所以，函数式编程从根子上就无法支持循环，只能支持递归。

读者可能会担心，函数式编程不支持循环，功能上是否不够全面？

这点不用担心。循环和递归在计算模型上的实现能力完全等价。循环能够实现的功能，递归也一定能够实现。反之亦然，递归能够实现的功能，循环也一定能够实现。而且，循环和递归这两种形式是完全可以相互转换的。循环形式，一定能够转换成递归形式。而递归形式也一定能够转换成循环形式。

命令式语言的程序员习惯了循环形式，函数式语言的程序员习惯了递归形式。命令式语言的程序员若想学习和使用函数式编程，必须掌握循环到递归的转换方法。函数式语言的程序员若想学习和使用命令式编程，必须掌

握递归到循环的转换方法。若是两种编程模型都想深入掌握的程序员，则必须掌握循环和递归的相互转换，既要掌握循环到递归的转换，也要掌握递归到循环的掌握。

本章的中心内容就是讲解循环和递归这两种形式的相互转换。本章希望帮助读者达到如下的目标：更深入地理解循环和递归这两种形式的本质；加深对函数调用和运行栈的理解；更有效地使用参数和返回值在进程调用间传递数据；更有效地理解和使用数据结构。

首先，我们来看几组递归形式的重要概念。

关于递归的第一组重要概念是：直接递归和间接递归。

直接递归是这样一种递归形式——函数调用自身，比如：

```
f(...) {
  ...
  f(...)
  ...
}
```

间接递归是这样一种递归形式——函数f并不直接地调用自身，而是调用了其他的函数g，而函数g又调用了函数f，这就形成了一个间接递归。比如：

```
f(...) {
  ...
  g(...)
  ...
}
g(...) {
  ...
  f(...)
  ...
}
```

这种间接递归可能不止一次“间接”，可能经过多次“间接”。比如，f调用g，g调用h，h调用m，m调用f。不管中间有多少次“间接”，只要最终形成了对自身的调用，那就是递归形式。

有时候，我们通过代码形式看出间接递归形式。有时候，递归结构是在运行时建立的，我们无法从形式上直接看出来。比如，下面的示意代码：

```
interface Calculator{
  int calculate(int n);
}

class FactorialCalculator implements Calculator{
  Calculator delegate;

  int calculate(int n){
if(n <= 1)
```

```
 return 1;

return n * delegate.caculate(n - 1);
 }
}
FactorialCalculator factorial = new FactorialCalculator();
factorial.delegate = factorial; // delegate指向自己，形成了递归调用的事实
factorial.calculate(10);
```

所以，我们要注意，有时候，不存在递归形式的代码也有可能形成递归调用的事实。

间接递归和直接递归只有形式上的不同，在概念上没有什么不同。我们平时遇到最多的递归形式是直接递归。本章讨论的递归形式基本上都是直接递归。间接递归只是形式上有些绕而已，本质上和直接递归没什么区别。

关于递归的第二组重要概念是：线性递归和树形递归。

线性递归是指递归函数的代码中只存在一次对本身的调用，比如上面的示意代码，都是线性递归。

线性递归在运行时，运行栈会一直向上增长，增长到头之后，就会一直向下减少，直到最后递归结束。也就是说，在线性递归的整个执行过程中，运行栈只发生一次伸缩。线性递归无论是形式上，还是执行上，都比较简单。线性递归转换成循环形式，相对比较简单。

一般来说，线性递归通常发生在线性数据的遍历算法中。

树形递归是指递归函数的代码中存在两次或两次以上的对本身的调用，比如：

```
f (...) {
 ...
 f(...)
 ...
 f(...)
 ...
}
```

另外，如果对自身的调用发生在循环体中，也是一种树形递归，比如：

```
f (...) {
 ...
 for (...)
f(...)
 ...
}
```

这种递归在执行的时候，在一个过程体内会产生多次递归，从而产生多次运行栈的伸缩，看起来像一棵有多个分支的树，所以叫做树形递归。在树形递归的执行过程中，运行栈时而增长，时而缩减，伸缩不定，有可能发生多次伸缩。因此，无论是形式上，还是执行上，树形递归都要比线性递归复杂得多。树形递归转换成循环形式，相对比较复杂。

一般来说，线性递归通常发生在树形数据的遍历算法中。

需要注意的是，有些递归形式看起来像树形递归，实际上却是线性递归。比如这样的代码：

```
f (...) {
  ...
  if(...)
f(...)
  else
  f(...)
  ...
}
```

上述代码中，看起来像是两次递归调用，但由于这两次递归调用是存在于不同的条件分支中，实际上，在执行的时候，只可能有一次递归调用。因此，这种递归是线性递归。

所以，我们在分辨递归的时候，不能只关注形式，还要关注事实。

关于递归的第三组重要概念是：尾递归和非尾递归。

在线性递归中，有一种特殊的情况。递归调用是函数体中执行的最后一条语句。比如：

```
f(...){
...
f(...)
}
```

注意，一定要保证，递归调用是函数体中最后一条执行的语句，才能满足尾递归的条件。

```
f(...){
...
return 1 + f(...)
}
```

这样的写法中，f(...)虽然看起来是函数体中的最末一句。但是，在执行过程中，却不是最后一步。最后一步是“+”这个操作。

有时候，代码看起来不是尾递归，实际上却是尾递归。比如：

```
f (...){
  if(... ){
...
return f (...)
  }

  if(... )
return 1
}
```

上述代码中，递归调用并不是函数体中的最后一条语句，但在执行上却是最后一句。所以，这也是尾递归。还有：

```
f (...){
  if(... )
```

```
return f (...)
  else if (...)
    return f(...)
  else
return 1;
}
```

也是尾递归。

至于什么是“非尾递归”，很简单，不是尾递归的递归，都是“非尾递归”。

线性递归可能是“尾递归”，也可能是“非尾递归”。

对于“非尾递归”的线性递归，我们用一个固定长度的数据结构来存放中间结果，就可以把“非尾递归”的线性递归都可以转换成“尾递归”形式。

树形递归一定是“非尾递归”，我们用一个复杂结构（模拟运行栈的伸缩不定的数据结构）来存放中间结果，就可以把树形递归转换成尾递归。

读者可能会问，我们为什么非要把“非尾递归”的线性递归转换成“尾递归”形式？

有两个理由，第一个理由是为了循环和递归的转化，第二个理由是为了空间和时间上的优化。

我们来看第一个理由——循环和递归的转化。

尾递归是一种最接近循环的递归形式。尾递归已经是递归向循环转换的最后一步，只要变换一下形式，尾递归就变成了循环。

递归向循环转换的常用手段就是，先把递归变成尾递归，然后再把尾递归变成循环。同理，循环转换成递归的最直接方法就是转换成尾递归。

我们这里就能够看到理论模型上的一致：正如所有的递归都能够转换成循环，所有的递归也都可以转换为尾递归。

再来看第二个理由——空间和时间上的优化。

这是一种“可能的优化”——主要是空间上的，也有一点时间上的。

由于尾递归在本质上已经等于循环，有些编译器（或者解释器）会对尾递归代码进行优化，在发生递归调用的时候，不需要保存所有的中间结果，因而并不进行真正的压栈操作，而是如同处理循环一样直接运算。这种优化不会引起运行栈的一层层的增长，从而节省了空间。在递归层次非常深的情况下，这种优化尤其有意义。

对于函数式程序员来说，尾递归更是一种必须掌握的技能。因为，非尾递归的运行栈是随着递归深度增长而不断膨胀的。如果递归层次过深，就会引起运行栈溢出（即空间不够了）的错误。

有些服务程序（或者图形界面程序）需要长期运行在计算机中，这实际上是通过无限次重复运行（计算机术语叫做死循环）来实现的。死循环的递归形式就是无限次递归。如果是非尾递归，必然会引起运行栈溢出的错误。因此，在函数式语言中，需要长期运行的死循环结构的程序（如服务程序或者图形界面程序）必须写成尾递归的形式不可。

至于时间方面，自然压栈和出栈的时间上的节省。有些特殊情况下，尾递归甚至可以引起算法上的优化。比如，有时候，树形递归会引起重复递归计算，这时候，尾递归就可以有效地消除重复计算步骤，从而优化算法，节省时间。当然，这只是特殊情况，大多数情况下，树形递归是无法优化的。比如，树形结构的遍历，你就得老老实实遍历所有结点，才能完成算法，没有捷径可走。

综上所述，尾递归的重要性是怎么强调也不为过的。

现在，我们从最简单的例子讲起——阶乘（factorial）。关于阶乘的数学定义为

n!=1×2×3×……×n

用数学归纳法表示为

n!=n×(n-1)!

根据第一种数学定义，阶乘算法很容易用循环形式写出。

```
int factorial(int n){
  int result = 1;
  for(int i = 1; i <= n; i++){
    result = result * i;
  }
  return result;
}
```

根据第二种数学定义（归纳法表示），阶乘算法很容易用递归形式写出。

```
f(1) = 1
f(n) = n * f(n-1)  // n > 1

int f(n){
  if(n = 1)
return 1;
  else
    return n * (n-1);
}
```

注意，为了简化问题起见，以上的代码忽略了n < 1的情况。真正运行的代码一定要把这种情况考虑进去。

我们看到，同一个阶乘问题，我们可以写成循环和递归两种形式。

以上的递归算法中，最后一步操作是“*”（乘法），而不是递归调用，所以，并不是尾递归算法，只是一种最直观的递归算法。递归调用自身之后，还要进行一个乘法计算。这意味着栈内的所有中间结果都是需要保留的。

如果要写成尾递归的形式，关键是要让乘法计算发生在递归调用之前，而且要把乘法结果保存在下一次递归调用能够访问的变量中。

命令式程序员对循环形式很熟悉，很容就想到用循环体外的局部变量来存放中间结果。比如，上面的factorial循环算法中定义的result变量。

函数式程序员脑海中没有循环的概念，只有递归的概念。对于递归，他们早已得心应手，一看到中间结果，首先想到的就是用参数和返回值来传递中间结果。因为，在函数式语言中，变量只允许赋值一次，不能用来存放随时变化的中间结果，只能利用参数来传递中间结果。

我们应该如何把上述的factorial递归算法转换成尾递归的形式呢？对于我们命令式程序员来说，最简单的做法就是从循环算法开始转换。具体做法就是把循环体中用到的所有变量都变成递归调用的参数。

前面已经有了factorial的循环算法。我们来考察其中的循环体在每一个循环步骤中用到的所有变量。

首先，i 和 result 这两个变量，是一定要有的。i 表示的是当前步骤，result存放的是中间结果。

其次，n 这个变量也是要有的。因为，在每一个循环步骤中，都需要判断 i <= n，从而判断是否结束。在递归算法中，我们需要做同样的判断来决定递归是否结束。

综上所述，递归函数的参数就确定了，就是以上的i、result、n这三个变量。至于递归函数的返回值，自然是当前计算的结果。

这样，循环算法中的循环体就可以修改成这样的递归代码。

```
int factorial(int i, int result, int n){
  int result = result * i;

  if(i >= n)
return result; // 结束递归

  return factorial(i + 1, result, n); // 把本次的计算结果向下传，继续递归
}
```

这种递归算法几乎和循环算法一模一样。递归体内与循环体的代码逻辑顺序是完全一致的：先做乘法计算，然后根据i与n的比较结果，决定是继续递归，还是结束递归。

所以，对于我们命令式程序员来说，若想把非尾递归转换成尾递归，最方便的方法就是先写出循环算法，然后根据循环体内的执行逻辑顺序，构造出一个尾递归函数。

以上的factorial尾递归算法多了两个没有必要的参数——i和result。我们可以用一个包装函数来消除这两个多余参数。

```
int neat_factorial(n) {
  return factorial(1, 1, n);
}
```

函数式程序员的考虑则是直接从结果来考虑。既然尾递归是最后一步操作。那么，所有的计算过程必然要在尾递归发生之前完成。然后根据当前运行步骤，判断结束递归，还是继续递归。这种思路与命令式程序员的思路殊途同归，但造成的结果还是有些细微的不同。

命令式程序员的思考起点是循环算法，是从1乘到n，从小到大的升序乘法。而函数式程序员的思考起点是递归算法f(n) = n * f(n-1)，是从n乘到1，从大到小的降序乘法。在这种降序的乘法中，我们可以省掉i这个用来代表当前步骤的参数。尾递归就可以写成这样：

```
int factorial(int result, int n){
  int result = result * n;
  if(n <= 2)
return result;
  return factorial(result, n - 1);
}
```

其包装函数为：

```
int neat_factorial(n) {
  return factorial(1, n);
}
```

这个尾递归转换成循环就是这样：

```
int factorial(int n){
  int result = 1;
  while(n > 1){
result = result * n;
n = n – 1;
  }
return result;
}
```

如果命令式程序员以这个循环算法为起点，得到的尾递归算法就是省掉了i参数的这个版本。

上述例子是线性递归，是最简单的递归形式。线性递归主要是针对线性结构的。对于线性结构来讲，算法的时间复杂度都是线性的。不管怎么折腾，管它是循环还是递归，抑或尾递归，时间复杂度都是一样的。

下一章，我们将讲解更复杂的、也是更主要的递归结构——树形递归。我们将看到，相对于线性结构而言，树形递归无论在空间复杂度上，还是在时间复杂度上，都要复杂许多。因此，树形递归通常拥有更大的算法优化余地。

1.17 《编程机制探析》第十六章 树形递归

发表时间: 2011-08-29

《编程机制探析》第十六章 树形递归

上一章我们讲解了线性递归，使用的是各种资料中用得最多、最为经典的例子——阶乘（Factorial）算法。
本章讲解递归结构中比较复杂的树形递归，同样使用各种资料中用得最多、最为经典的例子——斐波那契（Fibonacci）数列。
典型的斐波那契（Fibonacci）数列问题是这么描述的：有一种母牛，出生后第三年，开始生育，每年都生一头母牛（貌似单性生育，这里就没公牛什么事儿）；生出来的小母牛也符合同样的规律，出生后第三年，开始生育，每年都生一头母牛；该种母牛是永生的，而且永远拥有生育能力，生命不止，生育不止，生生不息。第一年时，只有一头母牛。请问第n年时，共有母牛多少头？
第一年和第二年时，母牛A还没有进入生育期，只有一头。
第三年，母牛进入生育期，生了一头小母牛B，共有两头母牛。
第四年，母牛A稳定生育，又增加一头小母牛C。小母牛B还未进入生育期，没有生母牛。共有母牛三头。
第五年，母牛A稳定生育，又增加一头小母牛D。小母牛B进入生育期，又增加一头小母牛E。C还未进入生育期，没有生母牛。共有母牛五头。
第六年，小母牛C进入生育期。该母牛种群开始进入爆发式增长。共有母牛8头。
第七年，13头。
第八年，21头。
第九年，....
我们可以用一个母牛生育表，来记录每年的生育期母牛头数和非生育期母牛头数，并统计统计每年的母牛总头数。
这里面的规律是，从第三年起，每过一年，上一年生育期母牛会生出同样数量的未生育期母牛。同时，上一年的未生育期母牛全部变成生育期母牛。
即，每过一年。
生育期母牛的个数 = 上一年生育期母牛的个数 + 上一年未生育期母牛的个数。
未生育期母牛的个数 = 上一年生育期母牛的个数。
母牛总数 =生育期母牛的个数 + 未生育期母牛的个数
表格数据如下：
年数 生育期母牛 未生育期母牛 母牛总数
1 0 1 1
2 0 1 1
3 1 1 2
4 2 1 3
5 3 2 5
6 5 3 8

7 8 5 13

8 13 8 21

9 21 13 34

10 34 21 55

我们观察母牛总数那一列。1,1,2,3,5,8,13,,21,34, 55...

可以发现这样的规律，从第三个数字开始，每个数字都等于前两个数字的和。

注：这个规律可以直接看出来，也可以通过前面的算式推导出来。

用公式表达就是这样：

f(1)=1

f(2)=1

f(n)=f(n-1)+f(n-2)

符合这种规律的数列，就叫做斐波那契（Fibonacci）数列。根据公式，我们很容易就得出递归算法。

```
int fibonacci(int n){
  if(n < 3)
return 1;

  return result = fibonacci(n - 1) + fibonacci(n - 2);
}
```

显然，这是一个树形递归。算法的空间和时间复杂度都很高，随着n的增大，呈指数增长。

递归调用的过程中，运行栈的伸缩次数是多次的，不断地压栈，出栈，再压栈，出栈。

很容易就可以看出，这个算法中充满了重复计算。

计算fibonacci(5)的时候，先计算 fibonacci(4)，然后计算 fibonacci(3)；

计算fibonacci(4) 的时候，先计算fibonacci(3)，然后计算 fibonacci(2)；....

fibonacci(4)完成了之后，fibonacci(5)再接着计算fibonacci(3)，而全然不顾fibonacci(3)已经算过一遍的事实。

如果我们把这个算法转换成尾递归，就可以有效地消除重复计算的情况。如何转换呢？我们命令式首先想到的就是用循环实现fibonacci算法，然后，转换成尾递归。这种方法很有效。因为Fibonacci算法不复杂，可以比较容易地用循环来实现，只需要用一个数据结构来存放f(n-1)和f(n-2)的计算结果就行了。

这种思路对于命令式程序员来说，是驾轻就熟了。现在，既然我们将要学习和使用函数式编程。我们就试图采用函数式程序员的思路。我们直接从现有的递归算法转换。

上一章我们已经讲述过尾递归转换的通用思路：尾递归只是最后一步操作，所有的计算都在之前完成，然后，计算结果作为参数传递到最后的尾递归操作中。递归函数在入口处就需要根据传进来的参数（上一次计算的结果和当前步骤）来判断是否继续递归，还是直接返回结果。有了这样的思路之后，我们先来分析前面的递归算法，抽离出中间计算结果。

```
int fibonacci(int n){
  if(n < 3)
return 1;
```

```
  int result1 = fibonacci(n - 1);
  int result2 = fibonacci(n - 2);

  int result = result1 + result2;
  return result;
}
```

我们需要把计算结果result1和result2作为参数。算法改造如下：

```
int fibonacci(int i, int result1, int result2, int n){
  int result = result1 + result2; // f(i) = f(i-1) + f(i – 2)

  if(i >= n)  // i 表示当前步骤。
return result;

int j = i + 1;  // i = j -1
  nextResult1 = result; // f(j - 1) = f(i)
  nextResult2 = result1; // f(j - 2) = f(i – 1)
  return fibonacci(j, nextResult1, nextResult2, n);
}
```

这就是一个尾递归算法，消除了重复计算的问题。因为我们把前两步的计算结果保存了起来，并作为参数传到下一次递归。

这个尾递归的包装函数如下：

```
int neat_fibonacci(int n){
  if(n < 3)
return 1;

  return fibonacci(3, 1, 1, n);
}
```

在上述的尾递归算法中，我们引入了参数i来表示当前计算步骤，还特意声明了一个变量j来表示i的下一步（j = i + 1），这是为了使得代码更加清晰易读。实际上，同上一章的尾递归例子一样，参数i是可以省略的。写法如下：

```
int fibonacci(int result1, int result2, int n){
  int result = result1 + result2; // f(i) = f(i-1) + f(i – 2)

  if(n <= 3)  // i 表示当前步骤。
return result;

int j = i + 1;  // i = j -1
  int nextResult1 = result; // f(j - 1) = f(i)
```

```
  int nextResult2 = result1; // f(j - 2) = f(i – 1)
  return fibonacci(nextResult1, nextResult2, n);
}
```

其包装函数为：

```
int neat_fibonacci(int n){
  if(n < 3)
return 1;

  return fibonacci(1, 1, n);
}
```

这种尾递归虽然少了一个参数，但是可读性明显差了许多。

上述两种尾递归写法都可以转换成循环，由于第一种尾递归可读性明显更好，我们就根据它来改造。

尾递归改造成循环，和循环改造成尾递归的过程正好相反。我们需要把参数变成一个局部变量的存储结构，循环体使用这个存储结构来存放中间结果。代码如下：

```
int fibonacci(int n){
  if(n < 3)
return 1;

  int result1 = 1;
  int result2 = 1;
  int result = 0;
  for(int i = 3; i <= n; i++){
result = result1 + result2; // f(i) = f(i – 1) + f(i – 2)

// 以下的代码是为下一步准备的。假设 j = i + 1
result1 = result; // f(j -1) = f(i)
result2 = result1; // f(j – 2) = f(i -1)
  }
  return result;
}
```

以上就是Fibonacci常规问题的常规解法，常见于各种资料。本章只不过对这个问题进行了综合剖析和进一步发挥。下面，我将对Fibonacci常规问题进行扩展，并给出相应算法和对应的数据结构。

首先，我们回顾一下Fibonacci问题的常规描述：

1头母牛，出生后第3年，就开始每年生1头母牛，按此规律，第n年时有多少头母牛。

f(1)=1

f(2)=1

f(n)=f(n-1)+f(n-2)

Fibonacci数列看起来是这样：1,1,2,3,5,8,13,,21,34.....

现在，我们对该问题进行一个小小的条件改变，对母牛品种进行一个小小的改造。我们嫌母牛成长得太快了。

现在，我们培育出一种新的变种母牛（姑且命名为变种A），这种母牛第四年才开始生育。这就产生了一个Fibonacci问题的变种，姑且称之为Fibonacci问题变种A。

Fibonacci问题变种A描述：

1头母牛，出生后第4年，就开始每年生1头母牛，按此规律，第n年时有多少头母牛。

f(1)=1

f(2)=1

f(3)=1

f(n)=f(n-1)+f(n-3)

Fibonacci变种A数列：1, 1, 1, 2, 3, 4, 6, 9,13,19,28.....

变种A成功之后，我们受到鼓舞，又依次研究出其他的变种母牛。

变种B：第五年开始生育。

变种C：第六年开始生育。

.....

最后，我们研制出所有的变种，可以任意控制母牛在几岁开始生育。这就得到了Fibonacci问题通用描述：

1头母牛，出生后第x年，就开始每年生1头母牛，按此规律，第n年时有多少头母牛。

令k = x - 1

f(1)=1

...

f(k)=1

f(n)=f(n-1)+f(n-k)

我们如何为这个Fibonacci问题通用版本编写算法。首先，树形递归算法最容易，也最简洁。

```
int fibonacci(int n, int k){
  if(n <= k)
return 1;

  int result = fibonacci(n - 1) + fibonacci(n – k);
}
```

那么，我们该如何把这个树形递归算法修改成尾递归算法和循环算法呢？我们遇到的主要问题就是缓存中间计算结果。这里，我们只保存前两步的计算结果，已经不够了。我们需要保存前k步的计算结果。这就是说，我们需要一个长度为k的存储结构，其中存放着前k步的计算结果。

假设当前计算步骤为i，那么，该存储结构内就存放了从f(i - k)一直到f(i – 1)的结果。当计算步骤向前推进时，存储结构的数据也向前推进。比如，当计算步骤推进到 j = i + 1时，存储结构内的数据就变成从 f(j – k) 到 f(j – 1)。

该存储结构的特征总结如下：其存储容量不定，由k参数定义；其存储内容跟随当前计算步骤而移动。

为了解决这个问题，我们引入一个叫做LinkedQueue的类。这个类是一个用链表结构实现的队列结构。关于队列结构，我们前面提到过，这是一种先入先出的数据结构，与栈结构相对。关于队列结构和链表结构的知识很简单，请读者自行补充。

LinkedQueue提供了如下方法：

LinkedQueue(int k)

int removeHead()

int getTail()

void addTail(int result)

构造函数LinkedQueue(int k)接受一个参数k，进行初始化，在内部生成一个长度为k的、元素类型为整数的链表结构，每个整数元素的值都是1。这代表着f(1)到f(k)的值。

removeHead方法直接取出头部元素（一个整数），队列长度减一。

getTail方法取出尾部元素的值（一个整数），并不取出元素本身，队列长度不变，内容不变。

addTail方法在尾部增加一个整数作为队列元素，队列长度加一。

这个数据结构的实现基于一种叫做双向链表的结构，即，每个结点都保留着前一个结点和后一个结点的引用。从而既可以在头部增删改，也可以在尾部增删改。

另外一种等价实现是一个长度为K的环形数组，这种实现在空间和时间效率上都很高。

这两种数据结构的实现都不难，不再赘述。

有了这个数据结构，我们就可以轻松写出通用fibonacci问题的尾递归算法。

```
int fibonacci(int i, LinkedQueue results, int n, int k){
  int resultK = results.removeHead();  // f(i – k)
  int result1 = results.getTail(); // f(i – 1)
  int result = result1 + resultK;  // f(i) = f(i-1) + f(i – k)
  if(i >= k)
return result;

  results.addTail(result);

  return fibonacci(i + 1, results, n, k);
}
```

其包装函数为：

```
int neat_fibonacci(int n, int k){
  if(n <= k)
return 1;

  LinkedQueue = new LinkedQueue(k);
  return fibonacci(k + 1, results, n, k);
}
```

下面我们把它改成循环。同样是把递归体变成循环体，参数变成循环体外的局部变量。

```
int fibonacci(int n, int k){
  if(n <= k)
return 1;
```

```
  LinkedQueue = new LinkedQueue(k);

  for(int i = k; i <= n; i++){
  int resultK = results.removeHead();  // f(i – k)
    int result1 = results.getTail(); // f(i – 1)
    int result = result1 + resultK;  // f(i) = f(i-1) + f(i – k)
    results.addTail(result);
  }

  return result;
}
```

这样，我们就完成了fibonacci通用问题的通用算法。当k = 2时，就是我们一开始遇到的常规fibonacci问题。

fibonacci问题虽然不是一个太难的算法，但毕竟还是浪费了我们一点脑细胞。有没有一种通用的方法来解决树形递归中的重复计算问题呢？

有。那就是使用缓存。如果你不在意内存的话。

比如，我们可以把前面的fibonacci常规问题的树形递归算法写成这样：

```
int neat_ fibonacci(int n){
  if(n < 3)
return 1;

  int[] resultTable = new int[n];
  for(i = 1; i <= n; i++){
    resultTable[i] = 0;
  }

  fibonacci(n, resultTable);
}

int fibonacci(int n, int[] resultTable){
  if(n < 3)
return 1;

  int cached = resultTable[n];
  if(cached != 0)
return cached;

  int result1 = fibonacci(n - 1);
```

```
  int result2 = fibonacci(n - 2);

  int result = result1 + result2;
  resultTable[n] = result;
  return result;
}
```

通用fibonacci问题的树形递归算法也可以这样写：

```
int neat_ fibonacci(int n, int k){
  if(n <= k)
     return 1;

  int[] resultTable = new int[n];
  for(i = 1; i <= n; i++){
    resultTable[i] = 0;
  }

  return fibonacci(n, resultTable);
}

int fibonacci(int n, int k, int[] resultTable){
  if(n <= k)
return 1;

  int cached = resultTable[n];
  if(cached != 0)
return cached;

  int result1 = fibonacci(n - 1);
  int resultK = fibonacci(n - k);

  int result = result1 + resultK;
  resultTable[n] = result;
  return result;
}
```

1.18 《编程机制探析》第十七章 函数式编程

发表时间: 2011-08-29 关键字: 编程

《编程机制探析》第十七章 函数式编程

当我们能够像掌握循环一样熟练地掌握递归之后，我们就可以正式向函数式编程进军了。当然，即使我们还没有熟练掌握递归，我们还是可以向函数式进军。我们可以在学习函数式编程的过程中，逐步习惯递归的写法。
函数式编程并非主流编程模型，函数式语言亦非主流编程语言。但我们在描述某些具体问题时，不可避免地要涉及到一些代码。本书会尽量选择有代表性的函数式语言。本书的选择是Haskell和ErLang这两门函数式语言。
Haskell是一门学术性很强的、特性很全面、形式很正规的函数式语言，很适合用来描述函数式编程的一些重要核心概念。因此，Haskell是一种常见的教学语言。本书也不能免俗。
ErLang是一门比较实用、对分布式计算支持比较好的函数式语言。本书选择这门语言，出于两个理由。一是ErLang简单易懂，学习成本比较低。二是ErLang在函数式编程中还算是应用比较广的，学习这门语言的回报率也相对大一些。
本书在讲解函数式编程的时候，仍然遵守一贯的原则：尽量使用文字——而不是代码——来描述基本原理；遇到具体代码时，只对特殊的语法现象进行解释，不讲解语法方面的基础入门知识。
我们首先讲解函数式编程语言的两种核心数据类型——Tuple（元组）和List（列表）。
Tuple类型类似于命令式语言的数组，其中元素数据类型不定。Tuple类型用圆括号()来表达。比如，
(1,”abc”,2.5, ‘c’)就是一个长度为4的Tuple。
比如，(1, (2, 3))就是一个长度为2的Tuple，其中第二个元素也是一个长度为2的元组(2, 3)。
在函数式语言中，同其他数据一样，Tuple只能在创建时赋值一次，以后就再也不能改变了。因此，Tuple一旦创建，其长度和内容都已经固定了。
在有些函数式语言中，Tuple用大括号{}来表达，含义和用法是一致的。
长度为2的Tuple叫做二元组。在有些函数式编程语言中，二元组也叫做Pair，即数据对。二元组在函数式编程模型中的地位十分重要。因为，在函数式编程模型中，二元组是另一个核心数据类型——List（列表）——的构成基石。
List是一种链表结构，每一个链表结点都是一个二元组，其中第一个元素叫做head（头），存放数据内容，第二个元素叫做tail（尾），存放着后面的二元组。所以，List本身就是一种特殊的二元组。
先让我们明确一个概念的用法和写法。在Java、Python、Ruby等语言中有一种表示“无”、“空”的概念，写做null。在函数式编程中，也有同样的概念。在有的语言中写做null，在有的语言中写做Nil。为了统一起见，在没有明确指定具体语言时，我们都用null。
我们从最基本的List例子看起。(15, null)这个二元组就是一个长度为1的List。这是怎么说的呢？在这里，List的数据只有一个——15。表示下一个List元素的尾部数据为null，就是说这个List已经结束了。因此，这个List的长度为1，其中只有一个元素15。
在函数式语言中，List并不是一种特殊的类型，而是Tuple的一种——二元组。本来，使用Tuple结构——即圆括号()——来表示，已经足够。但是，List在函数式编程中应用如此普遍，函数式语言为List结构提供了很多语法上的直接支持，提供了很多List的简写方法。

这种简化语法形式并非代表着实质的结构，而是为了读写方便。这种做法有个名字，叫做语法糖。
在函数式语言中，List的简写形式就是方括号[]。前面的例子中，(15, null)就可以写成[15]。
下面我们再来看两个元素的List。(14, (15, null))这个嵌套的二元组，就是一个拥有两个元素的List。这是怎么说的呢？在这里，最外层的二元组中，头元素（head）用来存放数据内容14，尾元素（tail）用来存放下一个二元组(15, null)。在下一个二元组中，，头元素（head）用来存放数据内容15，尾元素（tail）用来存放下一个二元组null。由于tail是null，就表示List结束。于是，List的长度就是二元组的嵌套深度——2。这个List可以简写为[14, 15]。
同理，我们可以写出长度为3的List。(13, (14, (15)))可以简写为[13, 14,15]。更长的List可以同理类推，不再赘述。
通过上面的例子，我们可以发现List的一个重要特性——List是从后向前，即从尾到头来构造的。
我们无法从头到尾来构造List，我们只能从尾到头来构造List。这是为什么呢？这是因为，List是由嵌套的二元组构成的，而且是头元素二元组，嵌套尾元素二元组。在函数式语言中，变量只能在声明时赋值一次，之后就不能改编。我们不可能先创建一个二元组A，再把另一个B二元组设置到A二元组的尾元素中去。被包含的二元组永远只能先被创建。而尾元素才是被包含的二元组。因此，List只能从尾到头来构建。
但是，当我们访问List的时候，却只能从头到尾来访问。因为，我们读取一个List的时候，只能从最外层的头元素来访问。一个List的遍历方法只能是从头到尾。
针对这个特性，函数式编程语言又引入了一个语法糖——[head | tail]，用来标志一个List的头元素和之后的尾元素（也是一个List）。其含义与二元组表达方式(head, tail)完全一致。
函数式编程语言都支持这个[head | tail]语法糖，而且这个语法糖远远比(head, tail)这种最原始的表达方式应用得广。
其中的道理，我不是很明白。也许是为了表达上的一致性，也许是为了强调说明这是一种List类型，增加可读性。
有了这个语法糖，List的表达方式又多了一种。
[13, 14, 15]可以表达为[13 | [14, 15]]，也可以表达为[13 | [14 | [15]]]，这和最原始的二元组表达为(13, (14, (15, null)))，已经非常接近了。另，这个List还可以写成[13 | 14 | 15]。
在有些函数式语言中，这种语法糖与最原始形式的元组比较相近，使用()圆括号包裹List元素，使用 : 分隔头元素和尾元素。比如，(1 : 2 : 3 : [4, 5, 6])。这种表达方式等价于[1, 2, 3, 4, 5, 6]。
无论是[head : tail]，还是[head | tail]，其真实结构都是二元组(head, tail)。
函数式语言的初学者经常遇到的问题之一，就是面对各式各样的List表达方式而无所适从，不明所以。
只要我们时时刻刻记住List的二元组本质，就不会被函数式语言中眼花缭乱的List表达方式迷昏了头。
现在，我们来看一个颇有迷惑性的例子。[[1, 2] | [3, 4]]。这个List是什么意思呢？这是一个长度为3的List。头元素数据本身就是一个List——[1, 2]。整个List用二元组原始形式来表达就是((1, (2, null)), (3, (4,null)))。
明确了List表达方式之后，我们就可以来研究List这个数据结构的本质特性了。前面分析过了，构造的时候，List只能从尾到头构造；访问的时候，List只能从头到尾访问。这是个什么数据结构呢？
答案已经呼之欲出了。对了，这就是一个栈结构（Stack）——我们经常遇到的、先入后出、后入先出的栈结构。只不过，List这个栈结构并不是基于数组结构，而是基于二元组链表结构来建立的。
我们要牢牢地记住，函数式编程中的List是一个栈结构，而且仅仅是一个栈结构，只能当作一个栈结构来用，不能当别的数据结构（比如队列Queue）来用。

我为什么要不厌其烦地强调这个问题呢？因为List这个名字很有迷惑性。命令式程序员一看这个名字，通常就产生一种误解，以为这是一种可以在尾部添加数据元素的队列。

我在这里明确地断定：不能。函数式List不能在尾部添加数据元素，因此，不能当作先入先出、后入后出的队列结构来用。函数式List只能在头部添加数据元素，因此，只能当作栈结构来用

本书有一个原则，尽可能多讲基本原理，尽可能将必须涉及到语法和代码的部分向后推。毕竟，一门陌生的新语言看起来总是面目可憎的。

在讲述函数式语言基本语法和代码之前，我们先把List相关的基本操作算法过一遍，这样心里才有数，后面接触到具体语法和代码时，也不会感到有太大的障碍。

List操作在函数式语言中非常重要，所有的函数式语言都有一套专门针对List操作的List函数库。

我们自己设想一下，List函数库中都有那些函数？

懂得比较多的读者，可能会说，当然是MapReduce了，大名鼎鼎的Google搜索引擎用的就是C++语言的MapReduce库。

这个答案是正确的。Map和Reduce是两种不同的List基础操作，List函数库必然会支持Map和Reduce这两种算法。不过，Map和Reduce这两个算法并非最基础的List操作函数。

我们继续问问题，最基础的List操作函数是什么呢？

比较实际的读者，深思熟虑片刻之后，脸上露出了笑容，道：“我知道了，是长度函数。用来求List的长度的函数。”

这个答案，咳，也没错，确实非常基础。所有的List函数库都支持。我们就来看这个函数应该如何实现。

这还用说吗？有读者已经不耐烦了。当然是从头到尾遍历一遍，统计元素的个数了。

没错，就是这样。如果List比较长，并且计算长度的操作很多的时候，我们可以自己做一些优化，另外用一个数值来存放当前List的长度。比如，这样一个二元组，(length, list)。其中length就是list的长度。当list增加一个头元素的时候，就形成一个新的二元组，(length + 1, [head | list])。当list减少一个头元素的时候，也形成一个新的二元组，(length – 1, tail list)。

关于计算长度的内容，也就这些。计算长度，确实是非常基础的List操作，但却不是我想问的。还有呢？还有没有其他的、非常基础的List操作？好好想想？

哎，算了，我不卖关子了。直接给出答案吧。List函数库中一个非常非常基本的操作就是，从头到尾，反转List。

这是为什么呢？因为List只能从尾到头构造。而通常来说，我们获取数据的时候，都是从头到尾获取的。

这里可以采取两种方案——尾递归算法和非尾递归算法。

当采用尾递归算法时，我们从头到尾获取数据，并根据数据构造List时，我们构造出来的List是反顺序的。最先读出的头数据实际上放进了List中最内部的尾结点（二元组中）。当这个List构造完毕之后，我们必须把这个List倒过来。

List反转操作的实现也非常简单和直观。List虽然构造的时候是从尾到头的，但是，访问的时候，却是从头到尾的。我们只需要从头到尾再访问这个List一遍，然后，根据读出来的元素，再重新构造一个List，得到的新List自然是反转过来的。

当我们采用非尾递归算法时，我们就可以根据一个List从头到尾地构造另一个List。但这种算法的实质是这样：先从前往后遍历，递归一步步深入进栈，把遇到的所有元素都处理一遍，所有的中间结果都保存在运行栈中，直到遇到最后一个尾部数据，递归开始退栈，从尾部向头部开始构造List。这种算法虽然避免了尾递归算法的

List反转，但是，空间上的开销却多于尾递归算法，而且，进栈出栈也有时间开销。

对于我们来说，我们考虑List算法时，还是首先要考虑尾递归算法，养成这种习惯。这就不可避免地要遇到List反转的情况。

以下的算法分析，除非特别声明，全都采用尾递归的解决方案，也就是说，全都涉及到List反转操作。

当我们在函数式编程使用尾递归时，一定要对List的元素数据顺序保持高度的关注和敏感。

现在，我们来考虑另外一个常见的List操作——两个List拼接——的算法。

假设有两个List，一个是[1, 2, 3]，另一个是[4, 5, 6]。我们想把这两个List拼接起来，变成一个List，[1, 2, 3, 4, 5, 6]。我们应该如何做？

对于任何List操作问题，我们要时刻在头脑中记住一个原则——所有的List都是从尾到头来构造的。我们考虑List算法的时候，也要根据结果List从尾到头来考虑。

在这个例子中，我们想要达到的结果是[1, 2, 3, 4, 5, 6]。按照List的构造顺序，这个List应该这样构造。首先要构造[6]，然后构造[5, 6]，然后构造[4, 5, 6]。当然，我们已经拥有[4, 5, 6]这个List了。我们可以在这个基础上继续构造，过程仍然是从后往前的。我们首先要得到[3, 4, 5, 6]，然后再得到[2, 3, 4, 5, 6]，最后得到[1, 2, 3, 4, 5, 6]。

根据这个过程，整个算法的流程就应该是这样。首先，我们要把[1, 2, 3]这个List反转过来，变成[3, 2, 1]。然后，我们从头遍历[3, 2, 1]，依次把3、2、1三个元素添加到[4, 5, 6]的头部，最后得到结果List。

这个例子可以继续深化为把多个List拼接在一起的算法。比如，把[1, 2, 3]、[4, 5, 6]、[7, 8, 9]、[10, 11, 12]。由于这些List的个数是不定的，我们可以把它们都放到一个数组里。这个问题就转换成，一个List中的每一个元素都是一个List。我们如何把这个List中的所有List都取出来，拼在一起，形成一个List？具体到这个例子，就是。

all_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

我们要把all_lists里的所有List都取出来，并拼在一起，组成一个大的List，[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]。

同样，我们需要根据结果List，从尾到头来考虑整个算法。我们已经拥有了[10, 11, 12]，我们需要在此基础上依次构造[9, 10, 11, 12]，[8, 9, 10, 11, 12]...直到最后的结果，[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]。

我们先要把[7, 8, 9] 这个List反转过来，变成[9, 8, 7]，然后，把9、8、7依次放入到[10, 11, 12]的头部，依次形成[9, 10, 11, 12]、[8, 9, 10, 11, 12]、[7, 8, 9, 10, 11, 12]。

之后，我们再把[4, 5, 6]反转成[6, 5, 4]，把6、5、4依次贴到中间结果List的头部，然后再把[1, 2, 3]反转成[3, 2, 1]，把3、2、1依次贴到中间结果List的头部。最后得到最终结果。

综上所述，此种问题的通用算法应该是这样。首先，把整个List反转。这时候，List中的最后一个子List（在这个例子中，就是[10, 11, 12]）就变成了第一个List（即List头部）。这个List头部是我们构建大List的基础，我们取出这个List头部，不动它。剩下的tail List中，就是那些需要依次反转、依次粘贴到中间结果List头部的其它List。

我们看到，这个算法中存在多次List反转。

我们再把这个例子扩充为这样：一个List的树形结构，全部展开成一个扁平的List。

这个操作在List函数库中叫做flatten（扁平化）操作。

这个算法就更复杂了，涉及到的List反转操作就更多了。不过，无论数据结构多么复杂，原则都是一样的，我们首先要考虑List的尾部。

由此可见，List反转操作在各种List操作中是多么的重要。复杂情况下，List反转操作如此之多，一不小心，就很容易进行重复的、不必要的List反转操作。我们在头脑中必须绷紧这根弦，时刻谨记着List反转操作，并避免在算法中进行重复的List反转操作。

对于函数式程序员来说，List反转操作，是除了尾递归之外的另一个重要的基本功。这也是为什么我不厌其烦地强调List反转操作重要性的原因。

我是一个命令式程序员，对函数式编程完全是半路出家。我的这些经验体会，都是在一头雾水中摸爬滚打中积累出来的。我希望，这些经验体会能有效地帮助想学习函数式编程的命令式程序员，达到少走弯路、节省时间的目的。

List函数库中的其他的一些List操作，就是一些老生常谈了。我们简单地过一遍。

Map操作，这个名字和Java里面的Map接口一样，但两者除了名字中都有映射的含义，其它没有太多的共同点。函数式编程的Map操作是这样一种操作，对一个List中的所有元素都进行某一种操作，（比如，加1，乘2之类的操作），并根据计算结果生成一个新的List。

Map操作的算法很简单，就是从头遍历List，对每个元素进行计算，把计算结果添加到一个新List中去。遍历结束后，新List也构建完了。

等等，这就完了吗？不，还没有完。不要忘记，这时候构造出来的新List是反顺序的，并不能和原来的List形成映射关系。所以，我们还要把构造出来的List反转过来。可见，List反转操作无处不在，不可轻忽。

Reduce操作是一种统计操作，遍历List中的所有元素，进行一种统计计算（比如，计数，求和，求乘积等），最后得到一个结果值。Reduce的含义是缩减。用在这里，其寓意就是把一个List缩减为一个结果值。

在很多函数式语言的List函数库中，Reduce算法被写成fold（折叠）。其寓意也是相同的。把一个List折叠成一个结果值。

fold算法（即Reduce算法）有两种版本，一种是从头到尾遍历（从左到右），进行统计，这种算法写成foldl（即fold left，从左到右进行fold）；一种是从尾到头遍历（从左到右），进行统计，这种算法写成foldr（即fold right，从右到左进行fold）。

foldr算法自然是要先把List反转一遍。这里，我们又遇到了List反转这个老熟人。

有一个英语翻译的笑话是这样讲的。

“How are you?”

中文翻译：怎么是你？

“How old are you?”

中文翻译：怎么老是你？

每次遇到List反转，我都会有这种感觉，“怎么老是你？”

Map和Reduce算法都接受一个操作函数作为参数，用来对List中的元素进行操作。这是一个很明显的Visitor模式。那个作为参数的操作函数就是Visitor函数。

注意，在命令式的面向对象编程中，Visitor是一个对象，算法调用的是Visitor的visit方法。但是，在函数式编程中，函数本身就是一种作为参数和返回值的对象。所以，这里我们作为参数传递给算法的就是Visitor函数本身。

不仅Map和Reduce算法是Visitor Pattern，List函数库里的几乎所有算法都是Visitor Pattern。这是显而易见的。因为List就是一种集合结构，其算法基本上就是遍历算法。而遍历算法只有两种设计模式，一种是Iterator Pattern，一种就是Visitor Pattern。既然函数式编程不支持循环和Iterator Pattern，那唯一的选择就是递归和

Visitor Pattern。
List函数库中，有一个Filter算法，类似于Map算法，也是接受一个List参数，返回一个List结果。
Map算法和Filter算法的不同之处如下。
Map算法接受一个Visitor函数，返回的List长度与原来的List相同，两个List之间存在着一一对应的关系。
Filter算法也接受一个Visitor函数；不过，这个Visitor函数是一个筛选函数，根据当前元素的值来决定是否保留这个元素；这个Visitor筛选函数的返回值为真，则保留该元素，这个元素将被加入到结果List中；这个Visitor筛选函数的返回值为假，则过滤掉该元素，不把这个元素加入到结果List中；因此，Filter算法返回的结果List的长度和原来的List通常不相同，而且，两个List之间也通常失去了一一映射的关系。
Filter算法不难理解，前面的讲解设计模式（组合模式，Composite Pattern）的章节中，有一个FileFilter的例子，那就是一个典型的Filter算法。
同样，Filter算法是Visitor Pattern。
同样，Filter算法的最后，需要一个List反转的操作。（唉，怎么老是你？）
如果我们不想老是遇到List反转，我们可以采用非尾递归算法。对于List来说，非尾递归算法写起来简单直观，这里就不多费口舌了。等我们学习了函数式语言具体语法之后，我们再来看具体的代码。
Map、Filter、Reduce是List函数库的三种基础算法。用得多，也容易理解。List函数库中还存在着很多其他算法，比如，排序算法，查找算法，统计算法（最小值，最大值，平均值），取出头N个元素，取出第N个元素，取出最后一个元素，等等。还有前面前面举的List合并的例子，还有树形List结构“扁平化”（flatten）的情况，都属于List函数库中的算法。还有其他林林总总的List算法。这些算法中，有些是Visitor Pattern，有些不是。
有些算法过于简单和常见，比如，排序算法，查找算法，统计算法（最小值，最大值，平均值），取出头几个元素，取出某个位置的元素，等等。有些算法比较复杂，而且比较少用。这些算法我们都不讲。
前面讲述的List算法——Map、Reduce、Filter——全都是List函数库中最有代表性、最能体现函数式编程思路的List算法。掌握了这几个基本的List算法之后，我们就可以比较彻底地理解函数式语言针对List的一种更加强大的语法糖——List Comprehension。
Comprehension这个词的本意是理解、领会的意思。用在这里，我也不知道该怎么翻译。有些资料翻译成List推导，但我不能领会其深意。这里，我们就直接用英文原词。
List Comprehension的语法结构是这样的，一个双竖线分隔符 || 把结果List和源List分开。双竖线分隔符 || 前面的部分，定义的是结果List内容的运算表达式，实际上相当于一个Visitor函数。双竖线分隔符 || 后面的部分，定义的是源List的内容，实际上相当于List遍历算法，其中还可能包括过滤条件，相当于一个起过滤作用的Filter Visitor。
我们从最简单的List Comprehension例子看起。
[2 * x | | x <- [1, 2, 3]]
这个List Comprehension表达式的含义是，x来自于源List，即[1, 2, 3]。而结果List中的每个元素是2 * x。得到的结果List是 [2, 4, 6]。
很明显，这是一个Map操作。2 * x 就是 Visitor 函数的内容。
我们再看一个例子。
[2 * x | | x <- [1, 2, 3] , x > 1]
这个List Comprehension表达式里面添加了一个过滤条件，x > 1。得到的结果List是[4, 6]。

这是一个Map操作和Filter操作的组合。首先进行的Filter操作，只保留源List——即[1, 2, 3]中大于1的元素，得到的中间结果List为[2, 3]。这里，x > 1 就是 Filter Visitor函数的内容。得到了过滤结果 [2, 3]之后，再针对这个[2, 3]进行Map操作，得到[4, 6]。

这是最简单的情况，源List只有一个。List Comprehension还可以支持两个源List。比如：

[x + y || x <- [1, 2], y <-[10, 20]]

这个List Comprehension表达式里面有两个源List。第一个源List的所有元素，都需要和第二个源List中的任何一个元素配对，组成一对参数，实际上，是一个二元组结构。这就形成了一个类似于笛卡尔乘积的大List。我们姑且称之为笛卡尔List。

产生这个笛卡尔List的操作，实际上是两个Reduce操作的嵌套。第一层Reduce操作里面，

针对第一个源List的每一个元素，都进行第二层Reduce操作。在第二层Reduce操作里面，来自于第一个源List的元素，和第二个源List中的每一个元素配对，形成一个二元组，添加到Reduce的中间结果里，即一个不断积累的元素数据内容为二元组(x, y)的List。

然后，针对这个笛卡尔List进行Map操作，这里的Visitor函数就需要操作一对参数——二元组形式的(x, y)。

注：笛卡尔乘积是集合论中的简单概念。这里不赘述，就当作读者已经掌握这个基础知识了。如果不知道这个知识，请自行查阅。

这个List Comprehension表达式的结果是[11, 21, 12, 22]。

我们还可以给给这个List Comprehension表达式加上条件。

[x + y || x <- [1, 2], y <-[10, 20], x > 1]

得到的结果是[12, 22]。

x > 1是其中的过滤条件。现在，有一个问题。这个过滤条件放置的位置存在两种方案。

第一种方案，先求出笛卡尔List，然后对笛卡尔List进行过滤。

这里，我们就需要先得到笛卡尔List，即[(1, 10), (1, 20), (2, 10), (2, 20)]。

然后，对该List进行Filter操作。这时候，Filter Visitor函数接受一个二元组(x, y)作为参数，该函数返回 x > 1 的结果。

最后，对过滤后的List，进行Map操作。

第二种方案，先过滤[1, 2] 这个List，得到[2]，然后再进行笛卡尔乘积操作，最后再进行Map操作。

很明显，第二种方案更优。但是，第一种方案更加通用，适用于过滤条件同时涉及到两个源List的时候。

List Comprehension语法处理器可能会采用这样的优化策略。如果过滤条件同时涉及到两个源List，那么，就采用第一种方案。如果过滤条件只涉及到一个源List，就采用第二种方案。

需要特别说明的是，这里用的“笛卡尔”List只是一种为了方便描述而生造出来的词汇。前面描述的过程中，笛卡尔List的生成，和最后的Map操作分成了两步。但这是不必要的。这两步完全可以合成一步，笛卡尔List并不需要被真正创建。

在实际的算法中，笛卡尔List可能只代表着一种两层的嵌套递归结构，具体来说，就是两层嵌套的Reduce操作。在这两层Reduce操作中，Map操作中的Visitor函数可以放在第二层Reduce操作中直接进行——根据(x, y)计算结果，添加到结果List中。这样，笛卡尔List就不用真正被创建。

这里描述的List Comprehension实现是我自己的设想思路。并不一定与真实情况一致。对这方面感兴趣的读者，可以对函数式语言的编译结果进行反编译，从中可以看到解开语法糖衣之后的真实算法。

在有些函数式语言中，List Comprehension表达式中的双竖线分隔符 || ，有时也写成单竖线分隔符 | ，其含义和用法是一样的。

1.19 《编程机制探析》第十八章 函数式语法

发表时间: 2011-08-29

《编程机制探析》第十八章 函数式语法

从本章开始，我们将开始接触到函数式编程语言的语法和代码。本书采用的是两种函数式语言——ErLang和Haskell。

我们从ErLang语法开始讲起，因为，ErLang语法比较简单易懂。不过，需要说明的是，这里的“简单易懂”，是对我们命令式程序员来说的。ErLang语法与命令式语法比较接近，命令式程序员看起来更舒服。相比之下，Haskell的语法虽然更简洁，但是，相距命令式语法甚远，读起来颇有难度。

读者可能会说，既然这样，你就干脆只采用ErLang得了，干嘛还要多讲一个Haskell。

我也想这么做，但问题在于，Haskell语言中存在着很多ErLang语言中不具有的特性，而那些特性涉及到函数式编程中的重要概念。所以，我们先讲ErLang，明白了函数式语法的基本特点之后，再看Haskell，就不会觉得那么陌生了。这个学习流程，是我根据自己的经验教训总结出来的，希望对大家也有用。

ErLang是Ericsson（爱立信）公司开发的一门开源的函数式语言。ErLang原本是针对电信领域的大规模并行实时数据处理而研发的，在并行计算方面有所长。不过，这个特点不是本书要讲的内容。关心ErLang并行编程的读者，可以找一下专门的ErLang语言来看。本书只是把ErLang作为一种函数式语言入门的敲门砖，讲解的都是函数式编程中的概念和模型。

面向对象语言中，数据类型分为两种——基本类型和对象类型（即class）。

函数式语言中，数据类型也分为两种——基本类型和函数类型。其中，基本类型和面向对象语言类似，包括整数类型、浮点数类型、字符类型（char）、字节类型（byte）等；还包括容器类型，如Tuple（元组）——对应命令式语言中的数组类型；至于List，则是函数式语言中的基本数据结构。

面向对象语言中，基本类型都是固定类型，没什么发挥余地；我们主要关注的是可以自定义的对象类型（class）。同样，函数式语言中，我们关注的是可以自定义的函数类型。

我们首先从ErLang的函数定义讲起。这里的例子还是那个我们已经非常熟悉的Fibonacci常规问题，其数学表达式为：

f(1) = 1

f(2) = 1

f(n) = f(n – 1) + f(n – 2)

用ErLang来写，就是：

```
fibonacci (0) -> 0 ;
fibonacci (1) -> 1 ;
fibonacci (N) -> fibonacci (N-1) + fibonacci (N-2) .
```

在命令式程序员看来，这种写法有些像三个同名函数并存（overload）的情况。但是，这并不是三个同名函数，而是一个函数。

在ErLang中，函数的结尾是用句号（英文的句号，一个点）来表示的。因此，上述的代码中，只有一个函数定义。

ErLang语句中的标点符号用法很象文章的标点符号。 函数名后面跟着 -> ，表示函数体的开始。整个函数定义结束用一个句号“.” ；同一个函数中，并列的逻辑分支之间，用分号“;”分界；顺序语句之间，用逗号“,”分隔。

那些看上去像是同名函数的定义，实际上只是不同逻辑的分支。在函数式语言中，这是一种常见的写法。这种写法叫做Pattern Match（模式匹配）。其匹配的情况很广，可以根据参数值匹配，也可以根据参数类型匹配。匹配的顺序是从前向后，类似于case语句。上面的代码完全可以写成：

```
fibonacci (N) –>
  case N of
1 -> 1;
2 -> 1;
N -> fibonacci (N-1) + fibonacci (N-2)
end.
```

其中的case of语句，相当于命令式语言中的switch case语句。

讲到这里，有一点需要特别说明一下。在函数式语言中，表达式中是可以存在条件分支的。

这在命令式语句中很少见。我只记得C语言中的类似语句，b = (1 == 2)? 20 : 10;

但是，在函数式语言中，表达式的语法极为丰富，可以包含各种条件分支，最常见的就是基于模式匹配的case of 语句。

另外，有些支持函数式语法特性的动态类型语言也支持在表达式中包含条件分支。

回头看前面的代码。认真的读者会发现，上述代码中有一个问题。代码中忽略了一个基本条件，N > 0。我们可以用一种叫做Guard（卫兵）的语法形式—— when + 条件判断语句 —— 来修饰我们的逻辑分支。

```
fibonacci (0) -> 0 ;
fibonacci (1) -> 1 ;
fibonacci (N) when N > 1 -> fibonacci (N-1) + fibonacci (N-2) .
```

Guard也可以加在case of 语句里面。例如：

```
  case N of
1 -> 1;
2 -> 1;
N when N > 0 -> fibonacci (N-1) + fibonacci (N-2)
end.
```

同样，Guard语句也是函数式语句中常见的语法形式。不过，Guard并没什么了不起的，只不过是一种类似于if语句的语法糖——挪到了Pattern Match（模式匹配）语句的后面——而已。了不起的是Pattern Match语句。Pattern Match语句的形式十分丰富多样，不仅可以用来表现条件分支——如前面代码中所示，还可以用来进行位置匹配赋值。

在ErLang中，Tuple（元组）类型用大括号{}表示。我们可以这样进行赋值。

```
{A, B} = {12, 13}
```

就可以把一个元组中的元素一次性按照位置匹配到对应的变量里。这里，我们就一次性地把12和13两个数据同时赋值到A和B中。

我们还可以这么写，一次将一个嵌套元组结构中的数据全部都赋值到对应的变量中。

{A, {B, C}, D} = { 1, {2, 3}, 4 }
我们还可以用占位符 _ 代替不需要赋值的位置。
{A, {B, _}, _} = { 1, {2, 3}, 4 }
这里，用下划线 _ 作为占位符表示“忽略其位置或者数值”。这种表达方式也常见于函数式语言中。
List是函数式编程中最为常见的数据结构，在匹配模式中用到更多，比如，Map算法。
map(Visitor, []) -> [ ] ;
map(Visitor, [Head | Tail]) -> [Visitor(Head) | map(Visitor, Tail)].
map操作本身就需要创建一个List，并不需要节省运行栈，没有写成尾递归的形式，不需要List反转操作。在真实的List函数库中，很多List算法都不是尾递归形式的，并也不需要List反转操作。
我们看到，在上面的模式匹配代码中，条件分支判断和赋值操作是同时发生的，直接把遇到的非空List的头元素和尾元素设置到Head和Tail两个变量中。
浏览前面所有的代码，我们还可以发现一个规律。ErLang代码中，所有的变量名（包括函数变量名）的第一个字母都是大写的，所有的定义出来的函数名都是小写的。这也是ErLang语言的约定。所有变量名的第一个字母大写。所有的常量名（包括定义的函数名和常量定义名）的第一个字母小写。
在ErLang语言中，所有的常量名，同时也是一个atom（原子字符串）。其概念类似于Ruby语言中的symbol概念，既代表一个固定的字符串常量，又代表一个固定不变的名字。所有的atom都可以转换成字符串，也可以从字符串转换过来。ErLang中的字符串实质上都是List。字符串和atom之间的转换通过list_to_atom和atom_to_list来转换。
于是我们可以这样获取MyFunciton：MyFunction = list_to_atom(“outer”)
如果outer函数已经定义，那么MyFucntion就等于outer函数，如果outer函数没有定义，那么list_to_atom(“outer”)会产生一个新的叫做outer的atom，MyFucntion就等于这个新产生的atom。
如果需要强制产生一个已经存在的atom，那么我们需要调用list_to_existing_atom转换函数，这个函数不会产生新的atom，而是返回一个已经存在了的atom。
关于ErLang的函数，我们还可以通过fun这个关键字来定义匿名函数。
比如，MyFunction = fun(x) -> x + 1.
这种匿名定义函数的语法，也是常见的函数式语法之一。有些资料中把这种匿名函数定义叫做lambda（拉丁字母λ），这是来自于数学公式中的概念。函数式编程模型与数学模型之间的联系之紧密，由此可见一斑。
另外，我们从前面所有的ErLang代码中还可以看出一个规律：所有的ErLang函数定义，都没有定义参数类型。这也是ErLang语言的特点。
ErLang是一种解释型的带有语法编译器的虚拟机语言。ErLang是解释语言，又不需要声明参数，看起来就像是一种动态类型的语言。因此，ErLang的函数签名只有两个组成部分——函数名和参数个数。这两个信息唯一确定了ErLang模块（Module）中的函数——同其他很多语言一样，ErLang每个文件就是一个Module。
当然，ErLang只是表现得如同动态类型语言，实际上，ErLang函数是有类型的。ErLang编译器在分析ErLang源代码的时候，能够推导出相应的参数类型和返回值类型。这叫做类型推导（Type Inference）。
比如，前面的fibonacci函数，其类型描述为：
fibonacci (integer()) -> integer()
表示，fibonacci接受一个整数，返回一个整数。
这是最简单的情况，我们来看复杂的情况。比如，前面的map函数，其类型表现形式为：

map(Visitor, SourceList) -> TargetList
Types:
Visitor = fun((A) -> B)
SourceList = [A]
TargetList = [B]
A = B = term()
这段类型描述表达的是什么意思呢？
map(Visitor, List) -> TargetList
表示，map算法接受两个参数——Visitor和SourceList，返回一个TargetList。
其中，Visitor是一个函数，接受一个类型为A的参数，返回一个类型为B的结果。
SourceList是一个元素类型为A的List。
TargetList是一个元素类型为B的List。
A和B是都是类型名。
我们看到，在这个类型描述中，并没有给出具体的类型，而是用A和B两个类型参数来代替。这种类型参数也是函数式语言中常见的类型表达方式。
在函数式语言中，这种包含类型参数的情况，也称为多态。不过，比起“多态类型”这个词，我更喜欢“抽象类型”这个词，这更能反映函数式语言中的参数化类型的本质意义。
在一些主流面向对象语言中，存在一个叫做“abstract”（抽象）的关键字，可以修饰类定义和方法定义，用来表示没有实现的类或者方法。这也是一种抽象类。但这里抽象掉的的是具体实现。而函数式语言的“抽象类型”抽象掉的是具体类型。
在形式上和语义上，函数式语言的参数化类型（抽象类型）与命令式语言的参数化类型（泛型）有很多共通之处。因此，也有资料称函数式语言的参数化类型为泛型。
为了避免各种名词引起的混淆，这种情况，无论是面向对象语言，还是函数式语言，我们都一律称为参数化类型。
看过了ErLang的基本语法特性之后，我们就可以开始学习Haskell了。
Haskell语言的命名来自于一个数学家Haskell Curry。Haskell成为语言的名字，而Curry则成为Haskell语言中的一个重要特性的名字。Haskell语言以λ演算（lambda）为基础发展而来。所以，Haskell中的匿名函数得名lambda。
Haskell由数学模型发展而来，其形式也非常接近数学公式。
比如，Fibonacci常规问题，其数学表达式为：
f(1) = 1
f(2) = 1
f(n) = f(n – 1) + f(n – 2)
用Haskell表达，就是：

```
fibonacci :: Integer -> Integer
fibonacci 1 = 1
fibonacci 2 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

从这段代码中，我们可以看到Haskell的语法特征。
Haskell是静态类型的语言。我们可以进行类型声明。如果不声明，编译器就会进行类型推导（Type Inference），分析出其类型。
第一行是类型声明，表示：fibonacci接受一个整数作为参数，并返回一个整数。
后面的函数定义部分，基本上就是三个数学公式，定义了三个逻辑分支。这种模式匹配的定义方法与ErLang相似。
我们看到，Haskell语法尽可能地减省，没有什么标点符号，而且不要求函数定义中用圆括号()包裹参数。后面的函数调用中，(n-1) 和 (n-2) 之所以被()圆括号包裹是因为它们是表达式。如果是它们是变量名的话，是不需要()圆括号包裹的。比如，f2 x y = f x + f y。所以，在Haskell中，空格非常重要。
同ErLang一样，上述的Haskell代码可以写成case of 的形式。

```
fibonacci :: Integer -> Integer
fibonacci n =
  case n of
1 -> 1
2 -> 1
_ -> fibonacci (n-1) + fibonacci (n-2)
```

其中下划线 _ 表示占位符，代表任何数值。
同ErLang一样，Haskell语言中也有Guard。
ErLang语言的Guard是when + 条件判断语句。
Haskell语言的Guard是 竖线 | + 条件判断语句。
上面的fibonacci函数可以写成：

```
fibonacci :: Integer -> Integer
fibonacci 1 = 1
fibonacci 2 = 1
fibonacci n | n > 0 = fibonacci (n-1) + fibonacci (n-2)
```

同ErLang一样，Haskell的Guard也没什么了不起，只不过是一种类似于if语句的语法糖。
Haskell的匿名函数（lambda）的形式有些怪，用一个正斜线 \ 来开头，比如，\x -> x + 1。
我们看到，Haskell的匿名函数定义中用到了箭头 -> ，这和ErLang是一样的。
到此为止，我们看到的Haskell语法和ErLang语法相当的接近，只有一些形式上的不同。我们再来看一个复杂点的例子——Haskell的map算法。

```
map :: (a -> b) -> [a] -> [b]
map visitor [] = []
map visitor (element : list) = ( visitor element : map visitor list )
```

这个算法的实现与前面的ErLang版本类似，这不是我们关心的内容。我们关心的是这个函数的类型部分，即：
(a -> b) -> [a] -> [b]
其中，a和 b 为类型参数。[a]表示一个元素类型为a的List，[b]表示一个元素类型为b的List。(a -> b)表示一个函数类型，其参数为a类型，返回值为b类型。

这个类型声明初看起来，与ErLang版本的map算法的类型描述很像。但是，我们细看一下，就会发现其中的不同之处。

在ErLang中，map函数的类型描述为map(Visitor, SourceList) -> TargetList。参数和返回值之间分得很清楚。

而在Haskell中，map函数的类型描述中，所有的参数和返回值之间都是用箭头->相连的。

Haskell为什么要采用这种表达方式？这和Haskell的一个重要语法特性相关。这个语法特性就是curry。

什么叫curry？用直白的语言来讲，就是这样：一个函数F1有N个参数，按理来说，我们调用这个函数的时候，应该一次把所有参数给齐。但是，在支持curry的函数式语言中，我们可以只给该函数M个参数（M < N）。我们得到的结果是什么？当然不是我们希望的返回值。那是什么？我们得到的是一个函数F2，这个函数F2的参数为（M – N）= X 个，我们继续给这个函数F2提供Y个参数（Y < X）。这时候，参数个数还是不够，我们又得到一个函数F3，其参数个数为（X – Y）= Z个。这一次，我们把Z个参数传给函数F3，这一次，参数个数终于够了，我们获得了返回值。这个返回值和一次就把N个参数都提供给函数F1是一样的。

我们看到，在这个过程中，我们可以分成几个步骤给一个函数提供参数，如果提供的参数个数不够，就得到另一个函数，该函数可以继续接受接下来的参数。这样分步骤提供参数的结果，和一次就提供所有参数得到的最终结果是一致的。这种特性就叫做curry。

例如下面的代码：

```
double :: Integer -> Integer
double x = 2 * x
```

然后我们这样调用map函数。doubleMap = map double

得到的doubleMap就是这样一种函数类型 :: [Integer] -> [Integer]

doubleMap接受一个整数List作为参数，返回一个整数List。其用法为

doubleMap [1, 2, 3] 得到结果 [2, 4, 6]

curry特性允许函数部分参数赋值，其作用相当于设置缺省参数值。

相对于面向对象编程来说，函数式编程少了同时携带对象的数据和方法的手段。而curry恰好就弥补了这个缺憾。

curry返回的另一个函数，可以看作是一个函数对象；curry过程中设置的部分参数值，可以看做是该函数对象携带的数据。这就有效地模拟了面向对象的对象结构。

因此，curry是函数式编程的一个很有用的特性。遗憾的是，并不是所有函数式语言都像Haskell一样支持curry。

ErLang就不支持curry。在ErLang中，想要如同面向对象编程中一样同时携带数据和方法的话，就得采用另一种语法特性——闭包（closure）。

何谓闭包？其实就是匿名函数。匿名函数是在函数体内部声明的，可以访问该函数体作用域内的变量。我们可以把一个访问了内部变量的匿名函数当做返回值，返回到外部调用函数，客观上就实现了类似于curry的功能——同时携带数据和方法。比如下面的ErLang代码：

```
make_closure(A, B) –>
  fun(X, Y) -> A + X – B - Y end.
```

就返回了一个函数，这个函数中携带了A 和 B这两个make_closure函数内部的变量，并接受两个参数——X和Y。用法为：

```
F = make_closure(1, 2)
result = F(3, 4)
```

Haskell也支持闭包，这个例子用Haskell写的话就是这样。（为了清晰起见，我在类型声明中用了和参数名一样的类型参数名。）

```
make_closure :: a -> b -> (x -> y -> z)
make_closure a b = \x -> \y -> a + x – b – y
```

用法为：

```
f = make_closure 1 2
result = f 3 4
```

说明一下，Haskell中，匿名函数如果有多个参数的话，就写成 \x -> \y 这样的形式。

Haskell还可以采用curry的写法。

```
g :: a -> b -> x -> y -> z
g a b x y = a + x – b – y
```

用法为：

```
f = g 1 2
result = f 3 4
```

在面向对象语言中，对应于匿名函数的结构就是匿名类。以Java为例。

Java的类方法定义的内部，可以定义匿名类。

匿名类方法内部，也可以访问外部方法的局部变量，而且匿名类的对象实例也可以作为参数和返回值，因此也可以模拟闭包的功能。从概念上讲，闭包带出的数据应该是只读的，于是Java规定，匿名类只能访问外部的final变量，以符合概念模型。

我个人是不喜欢匿名类的，也不喜欢匿名函数。能少用就少用，能不用就不用。但函数式编程中，携带数据和函数的方案只有两种，匿名函数和curry。所以，我喜欢curry。可惜的是，很多函数式语言都不支持curry。

有读者说了，Tuple里面不也可以同时放置数据和方法吗？而且是属性表和方法表放在一起，用起来就和Javascript对象一样方便。

这是不一样的。函数式语言没有面向对象语言的this（或者self）参数，写起来复杂得多，每次调用一个Tuple里包含的函数，必须手动的、显式的把该Tuple作为参数，传递给函数。这就相当于回到了C语言的年代。不仅没实现面向对象的优点，而且连函数式的优点也丢失了。这是典型的邯郸学步，东施效颦。

curry和闭包构造出来的函数对象就不一样了。它既有接受数据（curry参数，或者闭包能够访问的外部函数的局部变量）的构造函数，还拥有自己的一个方法（即构造出来的函数本身）。其表现同面向对象里面的对象几乎完全一样。区别只在于，函数对象的内部数据是不可改变的，而且只有一个方法（即本身）。

1.20 《编程机制探析》第十九章 函数 = 数据 = 类型 ？

发表时间: 2011-08-29 关键字: 编程, haskell, erlang

《编程机制探析》第十九章 函数 = 数据 = 类型 ？

本章继续讲解ErLang和Haskell的语言特性。
本书中选择ErLang和Haskell作为研讨语言，是因为我个人觉得这两门语言最具有代表性。
网上有一本脍炙人口的函数式编程教材，叫做《计算机程序的构造和解释》，英文为《Structure and Interpretation of Computer Programs》，简写为SICP。
你在网上搜索SICP，就能够直接搜索到这本书的网址。这本书非常有名，世界上不少有名大学用这本书作为教材。同时，这本书在对函数式编程感兴趣的程序员中也非常有名。
那本书采用的函数式语言是scheme，是LISP语言的一种简化版本，常用作教学语言。我从那本书中学到了很多。我在本书中用到的“树形递归”概念就来自于SICP。
注：LISP也是一门应用相对较多的工业级语言，有很多实际项目。
那本书对于很多函数式编程模型的基本概念，进行了极为透彻的解释和剖析，令我受益匪浅。尤其是其中“数据抽象”关于Pair数据类型的描述。
在SICP中，Pair数据模型就是用闭包来实现的。在scheme语言中，闭包的概念同ErLang和Haskell是一样的，也是匿名函数。而且，scheme语言中的匿名函数干脆就用lambda这个关键字来定义。
Scheme的语法和LISP语言相似，也是一门值得学习的语言。不过，这不是本书的任务了。
我在这里，就用ErLang的匿名函数，模拟SICP中的例子，将Pair结构（即二元组）实现一遍。
读者可能会问，Pair结构还用实现吗？不就是二元组吗？ErLang里面不是已经有Tuple类型吗？这里的假设是，现在，我们只有一个非常基础的语言，还不支持Tuple类型，只支持最基本的函数式语法，我们如何在这样的条件下，从头构造出Pair（即二元组）这个类型。
在这个假设中，我们要考虑到最基本的函数式编程语言中的Pair实现。在这个纯粹的世界里，一穷二白，我们没有任何资源，只有最基本的生产资料——函数。
我们只能借助于函数来实现Pair数据结构。前面提到了，函数实现数据结构，必须借助闭包特性携带数据。前面也举了简单的闭包例子。
下面我们来看如何用闭包实现Pair数据结构。采用的编程语言，主要是ErLang语法，配上文字说明，和一定的类JavaScript代码对照帮助。
首先，我们定义Pair数据结构的一个构造函数construct_pair。ErLang语法如下：

```
construct_pair( Head, Tail ) ->
  fun( select_head ) -> Head;
     ( select_tail ) -> Tail end.
```

这段代码的意思是，construct_pair函数内部，定义了一个匿名函数，这个匿名函数的作用是，当匿名函数接受到一个select_head常量作为参数的时候，就返回construct_pair函数的Head参数；当匿名函数接受到一个select_tail常量作为参数的时候，就返回construct_pair函数的Tail参数。最后，construct_pair函数返回这个匿

名函数。一定要注意，Pair函数返回的是另一个函数，返回值是一个函数。
请注意其中fun定义匿名函数时有多个模式匹配分支的写法。
可以看到，这是一个典型的闭包应用。匿名函数保存了外部的construct_pair函数的两个参数，将来调用的时候，会返回其中某一个参数。
为了帮助理解，下面给出JavaScript语法的对应代码：

```
construct_pair(Head, Tail){
  return new Function(Selector) {
if(Selector ==  “select_head” )
  return Head;
if(Selector ==  “select_tail”
  return Tail;
  }
}
```

现在我们定义了construct_pair函数，这是一个构造函数，能够构造并返回一个函数，这个返回的函数就是一个闭包。下面我们给这个闭包函数定义两个getter方法，用来获取闭包携带的Head和Tail数据。ErLang代码如下：

```
get_head(ClosureFunction ) –>  ClosureFunction (select_head).
get_tail(ClosureFunction) ->  ClosureFunction (select_tail).
```

对应的JavaScript语法的代码：

```
get_head(ClosureFunction ) { return ClosureFunction ( “select_head” ); }
get_tail(ClosureFunction ) { return ClosureFunction ( “select_tail” ); }
```

现在我们有了一个构造函数construct_pair，两个getter函数，Pair数据结构定义完毕。使用方法如下：

```
Pair = contruct_pair(1, 2),
Head = get_head(Pair),
Tail = get_tail(Pair),
```

变量Head的值是1，变量Tail的值是2。
我们可以看到，这么定义的Pair数据结构，是一个只读的数据结构，一旦构造完毕，数值就不能修改，只有getter，没有setter。正符合二元组的概念。
这个例子主要表现的概念就是，数据和程序之间的关系。在这里程序就是数据，数据就是程序。这完全是由闭包能够携带数据的特性来实现的。
如果用Haskell的curry特性来实现这个结构，写法会更清晰一些。

```
contruct_pair :: a -> b -> char -> d
contstruct_pair a b  ‘H’  = a
contstruct_pair a b  ‘T’  = b
```

```
get_head:: (char -> a ) -> a
get_head curried = curried  'H'

get_head:: (char -> b ) -> b
get_head curried = curried  'T'
```

用法为：

```
pair = contruct_pair a b
theHead = get_head pair
theTail = get_tail pair
```

二元组可以这样构造？那么，三元组，四元组呢？同样的写法。三元组，就写三个匹配分支。四元组就写四个匹配分支。整个Tuple类型呢？应该如何构造？

这个，从理论上讲，Tuple中有多少个元素，就应该构造多少个匹配分支。这才符合数据即程序、程序即数据的理论模型。

真的是这样吗？有人不敢相信地问。

我点头。真的是这样。事情的真相就是这么残酷。

真实的情况下，我们无法用索引来访问Tuple中的元素，我们只能用模式匹配（具体就是位置匹配）的方式获取Tuple中的元素。

这意味着，如果我们想取得Tuple中某个元素，我们必须这样写：(_, _, x) = (1, 2, 3)。或者，我们就得为Tuple中的每一个位置的元素定义一个获取方法。比如，二元组的获取方法，用Haskell写就是这样：

```
head (x, _) = x
tail (_, y) = y
```

如果是个三元组，四元组呢？那就得为每一个长度的元组都定义一连串的函数。

三元组的获取方法：

```
first (x, _, _) = x
second (_, y, _) = y
third (_, _, z) = z
```

四元组的获取方法：

```
first (x, _, _,_) = x
second (_, y, _,_) = y
third (_, _, z,_) = z
fourth(_, _, _, w) = w
```

别无它法吗？

方法是有的。但是，这需要自定义Tuple类型。由于Haskell的类型定义比较复杂，我们先来看ErLang的自定义

Tuple类型。
ErLang提供了一种叫做record的宏定义，允许程序员用字段名字来访问Tuple中的每个元素。具体例子如下：
-record (xyz, {x, y, z}) % record 的定义，xyz 元组中依次含有 x, y, z三个元素
XYZ = #xyz{ x = 1, y = 2, z = 3} % 创建一个Tuple
X = XYZ#xyz.x % 访问Tuple中的元素x ——即第一个元素
Updated = XYZ#xyz{y = 1} % 更新元素y ——即第二个元素，并生成一个新的Tuple。
另外，record还可以嵌套，还可以模式匹配。在搜索引擎中用ErLang Records两个关键字进行搜索，很容易就能找到ErLang文档中关于record的各种用法的详细例子。其中，模式匹配的例子，令人击节叹赏。强烈建议读者去读一读。
record也叫做Tagged Tuple（贴了标签的元组），Haskell也提供了类似的“Tagged Tuple”自定义类型。我们从头看起。
在Haskell中，“Tagged Tuple”是用data这个关键字来构造的。
data用于定义“类型构造器”（Type Constructor），即“类型”的构造函数。data有两种用法，可以带类型参数，也可以不带类型参数。
不带类型参数的时候，定义出来的类型名本身就是一种具体类型。比如：
data Boolean = True | False
其中的竖线 | 表示“或”的意思。
带类型参数的data定义，就是一种参数化类型（抽象类型）。比如：
data Pair a = pair_constructor a a
我们就定义了一个元素类型相同的二元组类型——Pair。
pair_constructor 10 10 的数据类型就是 Pair Integer。
pair_constructor 1.2 3.1 的数据类型就是 Pair Float。
Pair a是参数化类型（抽象类型）。Pair Integer和Pair Float都是Pair a的类型实例（instance）。这两个类型已经成为了具体类型，不再是抽象类型。
我们看到，Pair实际上对应的数据结构就是二元组，只是元素数据类型相同而已。
我们可以把上述类型构造器的定义修改如下：
data Pair a b = pair_constructor a b
这样，我们就定义了一个通常意义上的二元组——Pair。
pair_constructor 10 10 的数据类型就是 Pair Integer Integer。
pair_constructor 1.2 3.1 的数据类型就是 Pair Float Float。
pair_constructor 10 1.1 的数据类型就是 Pair Integer Float。
那么，我们如何取出Pair中的数据呢？同Tuple一样，我们只能使用模式匹配（位置匹配）的方式获取对应位置的数据。
pair_constructor _ y = pair_constructor 10 12
就把12获取到y变量里面。
我们可以写出pair_constructor的元素获取函数。
head (pair_constructor x _) = x
tail (pair_constructor _ y) = y

我们还可以把类型名和类型构造器名写成一样。比如：Pair a = Pair a a

这样，就得到了“类型名即函数名”的效果。从这里，我们可以体会到更多函数式编程中“一切皆函数”的思想。对应的，面相对象语言中也有“一切皆对象”的思想。

类似于ErLang的record宏，Haskell的data也可以定义字段名。

data Pair a = Pair {head, tail :: a}

这就相当于直接定义了head和tail两个获取方法。head (Pair 10 20) 就得到10，tail (Pair 10 20) 就得到20。

同ErLang的record宏定义一样，我们也可以在模式匹配中用到字段名。

add (Pair{head = x, tail = y}) = x + y

类似的，我们也可以更新某个字段，获取一个新的元组。

p = Pair 10 20

updated = p{y = 30}

调用updated（在Haskell里，updated就相当于函数名），得到的结果是Pair 10 30。

解决了“Tagged Tuple”问题之后，我们继续探索函数式语言的旅程。

不知道读者有没有发现一个奇怪的现象。在ErLang中，顺序语句之间用逗号,分开。但是，在Haskell中，却从来没有提过这事儿。在前面出现的Haskell代码中的函数体中，只有一条语句。确切的说，只有一个表达式，尽管这个表达式可能有很多个条件分支。

难道Haskell函数里只允许有一个表达式吗？没错，事实正是如此。Haskell函数定义中，只允许存在一个表达式。当然，这个表达式可以任意复杂，可以包含很多层次的条件分支。但归根结底，它只能是一个表达式的语法树。

关于Haskell，有一种说法：Haskell是没有调用顺序的。

这种说法的原因就在于此。在Haskell中，每个函数只允许有一条语句，根本就不存在顺序执行的语句，当然就没有调用顺序了。

但是，Haskell真的不存在调用顺序吗？当然……不是。

表达式中可能存在条件分支，执行某个条件分支的时候，总得先判断条件，才能执行该条件分支的表达式。

当我们获得一个函数调用的结果，作为参数传给另一个函数的时候，这里面，函数调用之间也存在顺序。

因此，“Haskell是没有调用顺序的”，这句话是有其适用范围的。比如，在函数体内，在树根表达式的级别上，确实是没有调用顺序的，因为只有一个表达式的语法树的树根。

Haskell函数内部只有一个表达式，那岂不是很受限制？确实如此，使用Haskell，我们必须学会更有效地分解问题，构造程序。

不过，为了方便编程，Haskell还是提供了两种辅助编程结构，允许程序员加入一些变量定义之类的赋值语句。

这两种辅助编程结构分别是let … in 结构和where 结构。这两种结构是等价的，只是代码位置摆放不同。

在let … in结构中，变量定义部分写在let和in只见，函数体表达式写在in的后面。在where结构中，函数体表达式写在where的前面，而变量定义写在where的后面。

let … in结构更为常见，而且，其他函数式语言（如LISP）中也存在类似的结构，我们就以let…in结构为例。参见下面的代码：

```
f :: Integer -> Integer
f a =
let
```

```
    i = 1
    j = i + 2
  in
    a – j + i
```

在这段代码中，我们定义了两个变量——i 和 j，并在函数体中用到了这两个变量。

这段代码没什么出奇的，读者看到这里一定会松一口气——“我说嘛，太阳下没有新雪，也不过是这么回事。”，或者会大失所望地叹一口气——“这有什么嘛。一点新意都没有。”

等一等，我把上面的代码换一种写法。

```
f :: Integer -> Integer
f a =
let
    j = i + 2
    i = 1
  in
    a - j + i
```

这段代码中，变量 j的定义引用了变量 i ，而 i 的定义在 j 之后。

看到这里，读者可能会陷入了深思。难道，这是因为Haskell没有顺序的缘故？所以，可以乱序执行？

我得说，这种想法是错误的。从概念上就错了。在let和in之间，只是变量的“定义”，而非变量的“赋值”。

这个概念一定要弄清楚。赋值需要顺序，但定义不需要顺序。

为了更清楚地认识这个问题，我们可以把变量定义看做是函数定义，即 i 和 j 是两个函数名，i 和 j 的定义相当于定义了两个内部函数。

这么一想，一切就豁然开朗了。既然是函数定义，那么自然是可以没有顺序的。

事实上，我们确实可以把Haskell中出现的名字都看作是函数名。这个想法很有效。在很多情况下，这种想法能够帮助我们理解很多看似稀奇古怪的现象。

我们再来看一个例子。

```
f :: Integer -> Integer
f a =
let
    j = 2
    i = 1 / 0
  in
    a – j
```

这个函数中，函数体表达式没有用到 i 这个变量，因此，i = 1 /0 这条语句就不会被执行。

不过，这种说法是错的。

应该说，i = 1/ 0 这个函数就不会被调用。

这才是正确的表达方法。

我们再来看一个有趣一点的例子。

```
f :: [Integer]
```

f = (1 : f)

你能看出这个函数的返回值结果吗？

一个元素全部是1的……无限List？有人犹犹豫豫地说。

没错，这就是Haskell中的著名的无限List。

这不会引起死循环吗？有人问。

在一般的语言中，这种代码确实会引起死循环。但是，Haskell不是一般的语言。它是一种支持Lazy特性的语言。

在计算机常用语中，Lazy（懒惰）这个词，是和Eager（积极）这个词相对应的。

大部分语言都是积极分子（Eager），看到一条语句，就急急忙忙冲上去，将其按倒，啊，不，是将其执行完毕。也就说，Eager语言积极性很高，遇到工作就要一下子做完，从不拖拖拉拉。Lazy语言就不一样，他很懒，总是把工作推到最后一刻，直到不能再拖的时候，才开始真正地工作。

下面，我们就以

UnlimtedList = f

head UnlimtedList

这段代码为例，看看ErLang（积极分子）和Haskell（懒家伙）的不同表现。其中，f就是我们上面定义的构造无限List的函数。

ErLang遇到这段代码，好嘛，立马挥起膀子就开始干活。首先，遇到的是UnlimtedList = f这条语句。于是，ErLang开始调用函数f，期望获得整个List。结果，这一去，当当当，恰如荆轲刺秦王，壮士一去兮不复还。他陷在死循环中，再也回不来了。

Haskell遇到这条赋值语句会怎么样呢？

“等等，你说什么？赋值语句？我没听错吧。”Haskell疑惑地看了我一眼，道：“在我的字典里，只有定义，没有赋值。更没有什么语句。这事儿和我没什么关系。你还是找别人去做。”

好嘛，这家伙还真懒，推得一干二净。

好吧，我承认，我的表达错误。我换一种说法，“请问，Haskell先生，您遇到UnlimtedList = f这条定义，该怎么做呢？”

“定义？”Haskell仍是那副看着白痴的表情，看着我，“你是说定义？我没听错吧？”

“没有。你没有听错。”我耐心道。

“你有没有搞错？！”Haskell发火了，“既然是定义，就让它定在那里好了。你还想干什么？”

“我.....”我又一次被打败了，只好指着下一条代码，“对不起，我错了。可是，您看这里，有个head f表达式，是对UnlimtedList的调用。这个表达式是需要被执行的。您看.....”

“好吧。拿过来我看看吧。”这次，Haskell没有推诿，拿过了任务单。

过了一会儿，Haskell把任务交了回来，简单地告诉我，“是1”。

“就这样？”我问。

“那还能怎样？”Haskell反问。

“你不是应该先获得UnlimitedList这个List吗？”我承认，我确实存有坏心。我想让ErLang和Haskell忙个不停。这样才能对比出我的清闲，从而显示出我作为一个资本家的优越性。

“List？”Haskell又是那副看白痴般的怜悯表情，“醒醒吧。我是大名鼎鼎的Haskell。在我的字典里，所有名字都可以看做是函数。UnlimitedList对我来说也只是一个函数。我只要调用它，得到结果就够了。”

好吧，我承认，是我太仁慈了。我又递给Haskell一份任务单，“那么，请看看这份任务吧。”

那份任务单是这么写的。

```
f :: [Integer]
f = (f : 1)
```

调用该函数的表达式是 head f。

“Haskell先生，”我尽量用柔和的声音道，“这一次，我还是需要同样的结果。我需要取得f这个List，啊，不，按照您的说法，是f这个函数。我需要取得f这个函数的返回值的头一个元素。您能帮助我吗？”

Haskell低头看了任务单很久，脸色苍白地抬起头，看着我，咬牙切齿道：“你够狠。”

这一次，Haskell终于被打败了。他被成功地拖住了。我得意地笑了。我只要不让你遇到返回一个值的匹配分支，我只让你遇到递归分支，我看你怎么Lazy，你照样得给我卖力干活。我，陶醉在自己的成功中。

血腥资本家，啊，不，成功企业家，就是这样炼成的。

我们来分析这个案例。f = (1:f) 和 f = (f:1) 有什么不同呢？

有人举手了，龇着一口大白牙，跃跃欲试地跳着道：“我知道，我知道。”

“你知道，你就说吧。”

“f = (1:f) 是尾递归。f = (f:1)是非尾递归。”那人胸有成竹道，目光中满是自信。

“瞎说！”我怒了，“简直比我还糊涂！”

这两种写法，都是非尾递归。因为，最后一步操作，都是一个组成二元组（即List）的操作。

区别在于，f = (1:f)这种写法的递归分支在后，费递归分支在前；而f=(f:1)的递归分支在前，非递归分支在后；非递归分支的位置不同，即造成了两种不同的结局。

Haskell的Lazy只能对非递归分支起作用，遇到递归分支，一样得卖命地干下去。

掌握了这个原则，我们就可以制造更多的无限List。只要非递归分支在前就行了。

```
f = let
    g = 2 : 3 : f
  in
    1 : g
```

这个函数返回一个[1, 2, 3, 1, 2, 3 ...]模样的List。

```
g n = n : g (n+1)
f = g 1
```

这个函数就返回一个自然数List，[1, 2, 3 ...]

在函数式语言的概念模型中，函数、类型、数据都可以看做是同一种东西——函数。在Haskell中，函数调用是Lazy的，数据结构也是Lazy的，如前面描述的无限List。

出于效率考虑，Haskell不可能将所有的Tuple类型都用闭包（匿名函数）实现一遍。Lazy特性的实现，通常基于一种叫做Thunk的结构。示意代码如下：

```
Thunk {
  value = null; // 变量初始值为空

  getValue() {
    if value is null  // 如果变量值为空，说明是第一次使用，那么为这个变量创建值
```

```
        then value = createValue();

    return value;
  }

  createValue() { ... }
}
```

这种Thunk结构是一种很常见的设计模式，叫做Singleton Pattern。特别的，由于该Singleton具有Lazy特性，所以也叫做Lazy Singleton。

1.21 《编程机制探析》第二十章 流程控制

发表时间: 2011-08-29 关键字: 编程

《编程机制探析》第二十章 流程控制

本章讲解函数式语言中的流程控制。
在此之前，先让我们把目光投回到命令式语言的世界。目光所及之处，有一片区域特别混乱。
上界派来的观察员大惊，“那是什么鬼地方？怎么和我自家的卧室那么乱？程序，不是应该遵守程序的吗？”
仔细一看，那一片区域正是隶属于“流程控制”的“跳转语句”，俗称“跳转界”。
你跳，我跳，他也跳。大家一起跳。杰克深情地望着露丝，“你跳，我也跳。”
跳转界中，跳得最欢，也跳得最强劲的，非C语言莫属。C语言支持goto label, set jump, long jump等指令，几乎可以任意跳转到任何一个地方。这是个跳槽高手。往外跳，往回跳。小跳，大跳，蹦着高的跳。有时候，跳得太远，就不知所踪了。那真的是，跳出三界外，不在五行中。
许是C语言跳得太欢了，引起了上界的不满。于是，后来的语言中，取消了这些臭名昭著的跳转语法。
于是乎，跳转界一片愁云惨淡。
我们来看一段条件指令的内层跳到外层的代码。下面的代码描述了跳转的一些可能的极端情况。

```
If(a == 1) {
  …
  If(b == 1){
     …..
     If(c == 1){
       …..
       // 需要直接跳到最外层的nextJob()
     }
     bNextJob();

     // 需要直接跳到最外层的nextJob()
  } // end if(b == 1)

  aNextJob();
} // end if(a == 1)

nextJob();
```

如果没有goto, label我们只能用flag来做些标志，一层一层地跳出来。

```
If(a == 1) {
  int aJumpOutFlag = 0; // 标志是否应该跳出a块到外面
```

```
  …

  If(b == 1){
    If(c == 1){
…
      // 需要直接跳到最外层的nextJob()
      aJumpOutFlag = 1; // 设置跳出a块跳到最外面的标志
    }

    if(aJumpOutFlag != 1){
      bNextJob();

        // 需要直接跳到最外层的nextJob()
      aJumpOutFlag = 1; // 设置跳出a块跳到最外面的标志
    }
  } // end if(b == 1)

  If(aJumpOutFlag != 1){
aNextJob();
}
} // end if(a == 1)

nextJob();
```

我们看到，每一层都需要判断aJumpOutFlag，这种方法简直令人难以容忍。

“自从没有了筋斗云，交通基本靠走，通讯基本靠吼，这样的日子没法过了！”

跳转界中怨声载道。

别急。我们还有其他的解决方案。我们可以用三个神奇的咒语（magic word），实现从内向外的任意层次跳转。这三个咒语分别是break、continue。嗯，就是这些了。

“等等。”有人不干了，“你不是说，有三个咒语吗？怎么才有两个？你不识数吗？”

咳，不是我不识数，而是因为……第三个咒语太强大了。以至于我都不敢说出它。

它（聚光灯瞬间亮起），就是终极跳转语句——return！

一般的情况下，我们不敢用它。一旦用了它，一切都结束了——云开雾散，换了新篇。

我们一般不敢劳动return它老人家，我们只用性情相对比较温和的break和continue。

我们来看一个例子：

```
while(a == 1) {
  …
  If(b == 1){
```

```
    .....
    If(c == 1){
      .....
      // 需要直接跳到最外层的nextJob()
      break;
    }
    bNextJob();

    // 需要直接跳到最外层的nextJob()
    break;
  } // end if(b == 1)

  aNextJob();
  break;
} // end if(a == 1)

nextJob();
```

注意，除了最后一个break，前面的两个break都可以用continue来代替，效果是一样的。
我们看到，活用循环体中中的break、continue，可以实现灵活的跳出控制。
在支持label（在循环体前加标签）的语言中，我们还可以在多层嵌套循环中实现更加灵活的跳转控制，想跳出到那一层就跳到那一层。

```
layer1:
while(...) {
  ....
  Layer2:
  while(...) {

  while(...) {
   ....
   break layer1
   ....
   break layer2
   ....
  }
  ...
  }
  ...
```

}

遗憾的是（或者说，幸运的是），函数式语言不支持任何类似的跳转语句。在函数式编程中，这些跳转技巧全都失去了用武之地，我们只能老老实实地分析流程，一条条、一支支地写好代码。

诗云，旁门左道不可取，人间正道是沧桑。就是这个道理。

感慨完毕，继续探讨。先看Haskell的流程控制。

Haskell的函数体内，只允许存在一个表达式，意即，只允许存在一棵语法树的树根。

放眼望去，从这个树根上蔓延出去的枝枝杈杈全都是同一种类型——条件分支。

在命令式语言中，流程控制语法比较多样，语法树上可能有三种类型的枝杈——条件分支、顺序分支、循环分支。

在函数式语言中，就只剩下两种——条件分支、顺序分支。

由于Haskell不支持顺序分支，就只剩下了一种——条件分支。

我们再来看ErLang。

ErLang的函数体中，最后执行的那条语句，必然是一个表达式，用来产生返回值。这点和Haskell相似。

但ErLang支持顺序语句，在ErLang的语法树上，除了条件分支意外，还有顺序分支。流程控制就比Haskell复杂了许多。

不过，ErLang代码中，顺序语句并不多见，而且大都是赋值语句，类似于Haskell的let...in结构中的变量定义部分。

我们来看具体代码——fibonacci常规问题的尾递归解法。

我先给出我的解法。让我们来看一看，一个命令式程序员是如何用他的命令式思维，写函数式程序的。由于Haskell不支持顺序语句，难以写出命令式效果，我这里就用ErLang代码。

```
fibonacci(CurrentStep , Result1, Result2, N) ->
  Result = Result1 + Result2, % f(i) = f(i-1) + f(i – 2)
  if(CurrentStep >= N)
Result
  else  % j = i +1
    NextResult1 = Result, % f(j - 1) = f(i) ,
    NextResult2 = Result1; // f(j - 2) = f(i – 1),
fibonacci(CurrentStep + 1, NextResult1, NextResult2, N)
  end.
```

这种写法几乎就是命令式语言版本的翻版。在函数式程序员看来，这种写法肯定是丑陋无比，不堪入目。没办法，水平所限，思维所限。不过，这种写法有一个好处就是，命令式程序员看起来不太费劲。

现在，我们来解决fibonacci通用问题，即：

f(1) = 1

f(2) = 1

...

f(k) = 1

f(n) = f(n – 1) + f(n – k)

用ErLang来表达就是：

```
fibonacci(N, K) when (N <= K)  ->  1;
fibonacci(N, K)                ->  fibonacci(N - 1, K) + fibonacci(N – K) .
```

现在，我们要把这个算法改成尾递归。

在讲解“树形递归”的章节中，我们已经讨论过fibonacci通用问题的尾递归和循环解法，使用了一个叫做LinkedQueue的队列结构。那个队列结构允许头部和尾部增删元素，实际上是一个双向列表结构。由于函数式语言的变量不能重复赋值，在函数式语言中，这种双向列表结构是不可能实现的。

我们只能采取折中方案，使用List保存所有的中间结果。

当然，List结构是一个栈结构，不是队列结构。我们需要把List尾部当做队列头部来使用，把List头部当做队列尾部来用。

每次计算完成之后，就在List头部（相当于队列尾部）增加当前的计算结果。

当计算需要f(n-1)时，直接取出List的头部元素（相当于队列尾部）。当计算需要f(n – K)时，就需要List头部向后遍历，获取第k个元素。我给这个遍历起个名字，叫做K遍历。在每一次递归中，K遍历都会发生一次。无论从空间效率还是时间效率来说，这种方法都远远不如对应的命令式语言的尾递归算法。

Fibonacci通用问题尾递归的ErLang代码。

假设nth是一个List函数，取得List第n个元素。

```
fibonacci(CurrentStep, Results, N,  K){
  Result1 = nth(Results, 1),      % f(i – 1)
  ResultK = nth(Results, K – 1),  % f(i – k). 假设nth是一个函数，取得List第n个元素。
  Result = Result1 + ResultK;     % f(i) = f(i-1) + f(i – k)
  if(CurrentStep  >=  K)
Result
  else
     fibonacci(CurrentStep + 1, [Result | Results], N, K)
  end.
```

这个算法中，k遍历的问题一直困扰着我。由于fibonacci通用问题是我自己扩展出来的，我找不到这个问题的相关资料。我找到的fibonacci的尾递归函数式解法，都是fibonacci常规问题（即k=2）的解法。

我设想了一种双List交替的方案，即同时维护两个List。一个List头部指向f(n-1)；一个List头部指向f(n - k)。这种方案可以节省空间，把一个长度为n的缓存List变成两个长度为k的List，但是，时间上却没有改进。

虽然我没有找到答案，但我根据常识判断，由于变量不可变的限制，函数式编程找不到解决k遍历的有效解法。这个论断还有待考证。

读者可能会问，为什么函数式编程不允许变量可变呢？

第一，是出于数学模型的考虑。函数式编程中，每个名字都代表着数学公式中的符号，并不直接对应内存结构，尽管在实现上，每个名字可能确实对应着内存结构。

第二，是出于并行计算的考虑。这点对我们来说，更具有现实意义。

要理解“变量不可变”在并行计算上的优势，我们需要从“状态”这个词谈起。

状态（State）是计算机编程模型中非常重要的概念。关于状态的两个形容词——Statefull（有状态的）和Stateless（无状态的），多见于各种资料书籍。

何为状态？这东西还真不好定义。我只能这么来描述它：状态就是可以改变的共享变量。这个共享变量的来源有两种可能。第一种可能，共享变量是程序内部自定义的数据结构，这叫做内部状态；第二种可能，共享变量是程序外部的环境资源，比如文件、数据库、显示器、键盘、打印机、网络服务、其他进程提供的各种服务，等等，这叫做外部状态，也叫做I/O（Input/Output，输入/输出）。

内部状态的产生方式只有一种，那就是程序本身定义的共享的可变的数据结构。这类数据结构大量见于各种命令式语言中。从内部状态的角度来说，所有命令式语言都是statefull。不过，我们可以在statefull语言中，写出stateless的程序代码，只要我们避免使用共享的可变的数据结构即可。

stateless的程序代码有什么好处呢？其优势主要在于并行计算。现代的计算机CPU更新换代，进入了多核时代，多个处理核心可以同时执行多个线程。

有一个关于线程的名词，叫做线程安全（Thread Safe）。什么叫做线程安全呢？这是代码的一种特质。在线程模型中，代码都是在线程中运行的。没有共享冲突、不用产生排队调度的，就叫做线程安全的代码。反之，存在共享冲突、需要排队调度的代码，就叫做线程不安全的代码。

共享冲突是怎样产生的？当然是共享资源引起的。没有共享资源，就没有共享冲突。共享资源是什么，就是状态（state）。因此，stateless就是线程安全的代名词。stateless的程序代码，就意味着线程安全的代码。Statefull的代码就是线程不安全的代码。

在命令式语言中，我们可以写出stateless的代码。那么，函数式语言呢？

从内部状态的角度说，我们用函数式语言，只能写出stateless的程序代码。因为函数式语言就不支持可变的变量，自然也就没有所谓的状态。所以，如果只考虑内部状态的话，函数式程序天生就是stateless的。

但是，事情没这么简单。程序不仅有内部状态，还有外部状态。

程序对外部状态的读写，涉及到于外部进程（包括系统进程和其他应用程序）之间的数据交换，一般都需要经过一种叫做I/O（输入/输出）的编程接口。因此，外部状态通常简写为I/O。

函数式编程可以没有内部状态，但是，却不可以没有I/O。没有I/O的程序没有任何意义。一个程序自己在内存中跑了一圈，结束之后，运行结果也跟着消散了，没有传输到任何I/O设备上。谁也不知道，他干了什么。春梦了无痕。这样的程序有什么用？

任何有实际意义的程序中必然有I/O。我们能做的是，把I/O部分尽可能地和其他代码分开。I/O部分，没有办法，天生就是statefull的。我们只能在非I/O代码上做文章。

理论上，非I/O代码都可以写成stateless。在命令式语言中，我们可以通过避免内部状态来写出stateless的非I/O代码。在函数式语言中，我们写出来的非I/O代码自然就是stateless的。stateless的非I/O代码有什么好处呢？前面说了，就是便于被多个处理核心同时运行，即并行。

近些年来，函数式编程有升温的势头，这和多核时代并行计算的兴起不无关系。可见，硬件基础决定软件上层建筑哪。

ErLang作为一门工业级语言，自然是支持I/O调用的。Haskell作为一门学术性语言，有权利任性一把，在消除I/O方面付出了很大的努力。这意味着，我们想用Haskell处理I/O时，也得付出很大的努力。

在普通的Haskell函数中，是不允许有I/O操作的。这种特性，被Haskell程序员骄傲地称为“Pure”（纯的）。在Haskell中，想要调用I/O，就必须用到一种叫做Monad的类型。

有些资料把Monad翻译为“单子”。这并不是“订单”、“单据”之类的意思，而是数学中的概念。无论是“Monad”，还是“单子”，我都不明其意，干脆还是用原文Monad。

Monad是世界上极为难懂的概念之一。有人如是说。我深以为然。

有人还说，如果一个人没有经历过任何思考上的痛苦，就理解了Monad，那他一定没有真正理解Monad。
此话，我不以为然。因为在学习Monad的过程中，我经历了非常多的思考上的痛苦，我到现在也没有理解Monad。
这句话应该换成：去它的Monad，这就不是常人能弄懂的玩意儿。让它成为那些高人自娱自乐的玩具吧。
这里，我只能给出我的大概理解。
Monad是Haskell类型系统的一种自定义类型。要想全面理解Monad，需要先了解Haskell的类型系统定义。
作为一个命令式程序员，我表示，Haskell的类型系统不容易理解。里面有些概念和命令式语言中即遥相呼应，又似是而非。面向对象语言的类型系统（主要是指泛型）固化了我的思路，不仅没有帮助我，反而徒增了很多困扰。
我觉得，对于命令式程序员来说，要想真正理解Haskell类型系统，应该从具体函数类型开始，一步步向上抽象，把具体类型抽离出去，
Monad也是如此。我们应该从具体的Monad应用场景入手，一步步向上抽象，最后得出最为抽象的Monad类型定义。
Monad的主要应用场景是什么？是提供状态和顺序操作。
提供状态，就不用说了，I/O操作本身就是外部状态。至于顺序操作，Haskell不支持顺序语句，但是，I/O操作又必须保证一定的顺序，所以，我们需要一种结构来迫使程序顺序执行。
最简单的思路是这样：把所有的需要顺序执行的函数，全都放到一个List里面，依次调用。
为了保证函数之间能够传递数据，我们把前一个函数的返回值，作为参数传给下一个函数。这就构成了函数执行链。
这种思路很像是复合函数的用法。
Haskell中的复合函数来自于数学，其含义也和数学中的复合函数一致。
当函数f的返回值类型和函数g的参数类型一致时，这两个函数就可以复合成一个函数：
h = g.f
g和f之间用一个英文句号 . 来相连。
从复合函数的定义就可以看出，能够复合的两个函数的要求很严格：只有一个参数，只有一个返回值；前一个函数的返回值类型和后一个函数的参数类型必须是一致的。
怎么理解复合函数呢？实际上，(g.f) x 就相当于g(f(x))。复合函数就相当于这样一个构造函数的函数：
```
composite_functions g f = \x -> g (f x)
```
多个函数复合就是这样：
```
call_functions [] x = x
call_functions (f : fs) x = f  (call_functions fs x)
composite_functions fs = \x -> call_functions (fs x)
```
其中，call_functions接受一个List作为参数，List中的每一个元素都是函数，而且都是(a->a)这样的函数，即所有函数的参数类型与返回值类型相同。
call_functions就从尾到头，依次调用每个函数，并把结果作为参数，传给下一个函数。这里没有使用尾递归算法，没有List反转，看起来好像是从头到尾调用，但实际上，这是个递归调用，只有递归到了最深层时，真正的调用才发生。退出栈的时候，就相当于一步步从尾到头开始调用。
除了复合函数之外，还有一种现成的算法是能够支持函数的链式调用。是什么算法？

没错，就是Reduce算法，也叫做Fold算法。在这个算法中，所有的函数依次执行，并且把中间结果值传递到下一次递归调用。

无论是复合函数，还是Reduce算法，对函数的参数和返回值的类型都有要求——前一个函数的返回值和下一个函数的参数的类型是一样的。这就需要一个包装类型把各种类型都包装成同一种类型。这也是Monad的功用之一。

以上的这种函数链思路，虽然简单，但是不够灵活，无法处理所有情况。比如，函数调用链有可能是树形结构；多个函数可能需要同时访问一组变量定义；调用过程中需要错误处理；等等。

为了对应各种可能的情况，Monad就应运而生。Monad的主要功用就是灵活地将函数编制在一起，并形成一个结构灵活的函数链结构。

可见，Monad也是用来控制流程的一种方案。

Monad的bind方法，就是把两个函数组合成一个函数调用链。Monad的return、fail等方法，实际上是函数链中的特例函数。

Monad中的条件分支语句，if、else、case等，都是某个具体函数内部的流程定义。这样，整个Monad流程看起来就像一棵布满了顺序语句的语法树。从这个角度来看，Monad可以看做是一种控制流。

如果从数据传递的角度看，大体上来看，数据总是参数流向返回值，再从返回值流向参数，这也是一种数据流。

对于理解Monad，我暂时能够提供的思路就是这么多，希望能对大家有一点用处。

1.22 《编程机制探析》第二十一章 AOP

发表时间: 2011-08-29

《编程机制探析》第二十一章 AOP

第二十一章 AOP

程序设计的一个重要目标就是提高重用性，避免重复代码。
到目前为止，我们已经接触到了诸多重用手段——过程式编程，面向对象编程，函数式编程，泛型编程，设计模式，等等。
本章介绍一种新的重用手段——面向方面编程（Aspect Oriented Programming），简称AOP。
什么叫做方面（Aspect）？这个词很难从字面上解释。我们还是通过具体的应用场景来看理解AOP的概念。
在程序设计中，我们虽然有各种手段来消除重复代码，但是，总有那么一些地方，遍布着重复代码，而那些地方，是我们现有的重用手段无法涉及的领域。我给那些地方起个名字，叫做设计死角。
那么，有哪些地方是设计死角呢？让我们来看几个具体的例子。
第一个例子，程序运行日志（Log）。
为了方便程序员（或者管理员）追踪应用程序的运行状态，程序员通常在程序代码中加入日志记录功能。
在原始的年代里，程序员大量使用print语句来输出程序运行的当前状态。虽然现在还是有很多程序员这么做。但是，这种做法已经是被摒弃的。正确的做法是使用日志（Log）编程接口。
通过日志（Log）编程库，我们可以定义日志开关，决定是否记录日志；我们可以定义日志存放位置，屏幕、文件、或者数据库；我们可以定义日志格式，简单文本、XML或者HTML；我们可以定义日志内容，决定哪些内容记录，哪些内容不记录；对于多个模块，我们可以定义多种日志记录方式；总之，日志编程库的好处不胜枚举。
现代程序员基本上都用日志编程库进行日志记录。但是，他们使用日志编程库的手段还是那么原始，他们照样把日志记录语句写得到处都是，遍布程序中各个角落。
那些日志记录语句就是重复代码，那些日志记录语句所在之地，就是设计死角。因为，日志记录遍及所有的对象类型中，我们无法用现有的设计手段进行统一设计。
第二个例子，数据库事务（Transaction）。
数据库是应用开发中最常用到的存储结构。数据库事务是操作数据库中最常遇到的问题。
这里简单地介绍一下数据库事务的概念。
数据库事务的和“原子操作”的概念有些相似。
在数据库中，我们执行一件任务，涉及到改动多处数据，我们希望这些改动，要么一次都成功，要么一切都保持原状。这样的一种“多个步骤合成一个原子步骤”的操作，就叫做数据库事务。
数据库事务分为两种——局部事务（Local Transaction）和全局事务（Global Transaction）。
局部事务中，涉及的数据库只有一个，涉及的数据也只在一个数据库中，这是我们在应用程序开发中最常遇到的事务。
局部事务的处理也很简单，就是把数据库的自动提交（Auto Commit）功能关掉，自己在所有数据操作都成功

之后，在手动进行提交（Commit）处理；否则就进行回滚（Roll back）操作，一切恢复原状。
全局事务的情况就要复杂一些，涉及到多个数据库中的数据操作。这时候，就要保证所有数据库中的操作全都要同时成功提交，或者同时回滚、恢复原状。全局事务多见于一些拥有多个数据服务器的大型机构中。
全局事务的处理相当复杂，卷入全局事务的所有数据库，相互之间要进行复杂的通信和确认，才能保证全局事务的“原子性”。
当然，数据库事务的繁简与否，不是我们关心的问题。我们关心的是，这些事务代码分布在哪里。我们需要把事务代码分布到几乎所有的数据库操作代码中。
我们应该如何设计，才能消除这些重复的事务代码？要知道，进行操作数据库的对象各种各样，没有统一的接口定义。
这些事务代码遍及之处，就是设计死角。
看过了这两个例子，你应该对方面（Aspect）这个词有了感性体会。是的，Aspect就是设计死角。Aspect就是遍及在程序中的日志代码。Aspect就是遍及在程序中的事务代码。在AOP的概念中，这是两个不同的Aspect，分别叫做Log Aspect和Transaction Aspect。
AOP的目的就是把这些重复代码从程序中抽离出来，在程序编译或者运行的时候，再把这些重复代码自动“回填”程序中的对应位置，从而免除了程序员的重复劳动。
从AOP的目的，可以看出AOP模型中的两个主要组成部分——重复代码和回填位置。
重复代码就是日志代码、事务代码等。回填位置就是重复代码原来所在的位置，即设计死角，或者叫做Aspect。
AOP处理器（编译器或者解释器）的工作很简单，就是在所有的回填位置中，加入程序员定义的重复代码。
AOP用户的工作是什么？就是定义重复代码和回填位置。
代码部分不用说了，我们来看回填位置应该如何定义。
首先，我们需要明确，回填位置都可以包括哪些位置。
重复代码是否可以填入到程序中的任何一条语句的前面或者后面？可以说，这种想法只存在理论上的可行性。实现上的效率且不说，就从用法上来说，也没有什么现实意义。
想象一下，我们需要给AOP编译器（或者解释器）定义这样一个任务：请把重复代码加入到方法f中的if(x > 0)语句之前。
为了描述“方法f中的if(x > 0)语句之前”这个位置，我们写的东西可能比重复代码还要长，还不如直接就把重复代码加到“方法f中的if(x > 0)语句之前”呢。
因此，这里有一个原则，回填位置的定义必须是简洁的。
既然语句级别不行，那么我们再上一级，到方法（过程、函数）的级别，这已经是代码块容身的最高级别了。如果这个级别也不行。那么，AOP也就别实现了。
幸运的是，这个级别是可行的。因为方法名简单易写，方法存身的类名、包名、模块名也很容易写，而且，还可以用通配符（*）来定义一批相似的回填范围。
AOP处理器（编译器或者解释器）根据回填位置定义，就可以把重复代码回填到方法（过程、函数）定义的前后。这个 “回填”的工作实际上相当于“篡改”了方法的原有行为，很像是黑客或者计算机病毒的行为。不过，AOP的“篡改”工作不是为了破坏，而是为了帮助。另外，“回填”这个动作，除了“篡改”这个别名之外，还有一个用得更广的别名，叫做“织入”（weave，编织），意思就是，把回填代码“编织”进原来的代码中。

AOP的“回填”动作的实现，基本上都是在Proxy Pattern（代理模式）的基础上实现的，即用一个Proxy对象包装原有对象的方法。例如：

```
LogProxy {
  innerObject; // 真正的对象
  f1() {
// Log here

innerObject.f1(); // 调用原来对象的对应方法

// Log here too
  }
}
```

再例如：

```
TransactionProxy {
  innerObject; // 真正的对象
  f1() {
// start Transaction

innerObject.f1(); // 调用原来对象的对应方法

// finish Transacion
  }
}
```

这些Proxy是可以叠加在一起的。叠加顺序由程序员自己指定。

当然，一个对象中的方法可能不止一个。这时候，我们就需要包装对象中所有需要AOP处理的方法。比如：

```
LogProxy {
  innerObject; // 真正的对象
  f1() {
// Log here

innerObject.f1(); // 调用真正的对象的对应方法

// Log here too
  }

  f2() {
// Log Hear
```

```
innerObject.f2(); // 调用真正的对象的对应方法

// Log here too
  }
}
```

这里面还是存在重复代码。Bad Smell。我们可以利用Reflection机制，写一个通用的方法截获器。截获器的英文叫做Interceptor。这个词汇有拦截的意思。

还有种叫法，叫做劫持（Hijack）。我们经常听说，某种浏览器被某某木马插件劫持了。就是这个意思。为了避免这种不良联想，我们还是用Interceptor这个更通用的词汇。

```
MethodInterceptor{

  around( method ){
    before(method)

    method.invoke(...); // 调用真正的对象方法

    after(method, result)
  }
}
```

Proxy可以继承这个MehtodInterceptor，实现before()和after()两个方法，把重复代码填进去。

函数式语言中，每个函数对象只有一个方法，用代理模式来包装，更加容易。从回填位置的声明上来说，回填位置定义中的通配符，等同于函数式编程中的模式匹配（Pattern Match）。从这两个方面看，函数式语言更有利于实现AOP。

Proxy Pattern + Method Reflection Interceptor + 代码自动生成（即回填）

这样一个三元组合，就是AOP的基本实现原理。AOP的实现多种多样，但是，其实现原理都脱不了这个范式。

AOP就像一个大染缸，本来干干净净的对象，经AOP一处理，就染上了（织入了）本来不属于它的特性。

AOP实现之间的区别，主要就在于代码自动生成（织入，填入）的方案。一般有两种方案。一种是在源代码上做文章，在编译器中加入插件，在编译源代码的过程中，把重复代码填进去（织进去），最后一起编译成目标代码。

一种是在目标代码上做文章。这种AOP直接处理目标代码，在目标代码中间填入（织入）重复代码。

这两种方案都是代码生成技术，我都不喜欢。实质上，重复代码还是分布在各处了。写在一处，复制到各处。

我喜欢第三种方案——在解释器或者虚拟机里做文章。当解释器或者虚拟机在执行代码的过程中，遇到回填位置，就会执行相应的重复代码。

这种实现方案就摆脱了代码生成技术，重复代码只有一处。但是，这种方案有一种致命的缺陷，那就是效率。

如果采取这种方案，那么，解释器或者虚拟机每次调用方法的时候，都需要查找“回填位置表”，看看这个方法是否在回填范围中。当然，我们可以做有限的优化。在第一次查找某方法之后，就在这个方法上贴上一个标签，比如，“无Interceptor”、“有LogInterceptor”、“有LogInterceptor、TransactionInterceptor”等。下一次，再遇到这个方法的时候，就可以不用查表了。但是，一个应用程序中的方法如此之多，这种优化

的作用是很有限的。所以，这种方案是不现实的。我没有看到过这种方案，也没有想出解决思路。所以，目前的AOP还是代码生成技术的天下。

AOP的实现原理，讲到这里就结束了。我们这里稍微涉及一下具体的应用。由于AOP目前还是一种代码生成技术，不符合我的审美观。而且，AOP的入门例子到处都是，也不值得占用本书的篇幅。我们这里说明一下常见的AOP声明语法。

首先，我先给出，我理想中的AOP声明语法是什么样的，然后我们再来看现有AOP实现框架中的语法惯例。

我理想中的AOP声明语法是这样的。形式上类似于Haskell函数。

weave 拦截器列表 回填位置列表

用英文表示就是

weave InterceptorList LocationList

具体例子就是：

weave [LogInterceptor, TransationInterceptor] [com.*, net.db.*]

这段声明的含义就是，把[LogInterceptor, TransationInterceptor]这个列表里面的所有拦截器，都织入到[com.*, net.db.*]这些位置的所有方法里面。

这样就够了吗？还不够。我们还需要一个过滤器，叫做Filter或者Exclude，用来过滤掉或者排除掉一些不需要拦截的特殊方法。我们可以直接在LocationList里面支持这种过滤和排除功能。比如：

weave [LogInterceptor, TransationInterceptor] [com.*, net.db.*, - net.db.test.*]

net.db.test.*前面加了一个负号（ - ）。这就表示，把net.db.test.*里面的所有方法都排除出去。

我们看到，这种声明形式很像是合法的程序调用代码。事实上，AOP确实会提供类似于这样的程序编程接口。只不过，他们的用词惯例不同。他们把LocationList叫做PointCut（切点，切面），把Interceptor叫做Advice。换成他们的词，AOP声明形式就是这样：

weave AdviceList PointCut

这就是一个weave函数，接受两个参数，AdviceList和PointCut。

在解释语言中，我们可以直接用这种函数调用代码的形式来声明AOP。因为解释语言的程序本身就可以当做配置文件来用，随时修改，随时运行，不需要重新编译。

但是，编译语言就不行了。为了保持AOP定义的灵活，我们必须把AOP声明从代码中抽出来，放在一个配置文件中，通常是XML文件中。

我们可以看到解释语言（通常是动态类型语言）的优越性。这也是我为什么倾向于动态类型语言的又一个原因。

另外，关于用XML表达程序结构，我是不太赞成的。在我看来，XML就是用来存放树形数据的。用XML来表达程序结构，简直就是滥用。但是，由于XML很容易解析，因此，这种滥用现象越来越严重。毕竟，解析一门解释语言，需要一个颇为复杂的词法、语法分析器，而XML需要的解析器则简单许多。

由于个人的好恶，我这里坚决不给出XML声明AOP的例子。反正只不过是把好端端的代码转换成蹩脚的XML格式而已。

另外，需要说明的是，在AOP的惯例声明语法中，Advice并不是直接对应Interceptor，而是对应Interceptor里面的before和after两个方法。我们可以把上面的MethondInterceptor进一步细化，变成：

```
BeforeAdvice{
  before(method)
```

```
}

AfterAdvice{
  after(method)
}


MethodInterceptor{
  BeforeAdvice beforeAdvice
  AfterAdvice  afterAdvice

  around( method ){
    beforeAdvice.before(method)

    result = method.invoke(...); // 调用真正的对象方法

    afterAdvice.after(method, result)
  }
}
```

按理来说，AOP的声明也就这些，到这里就完了。但是，还没有完。有些AOP玩出了花活儿，发明出了新式的AOP声明方式——运用一些语言中提供的标注（Annotation）特性进行AOP声明。

这是怎么做的呢？这种声明方式要求程序员用某种Annotation标注每一个需要织入AOP重复代码的方法。然后，AOP实现框架就在代码中去寻找这些Annotaion，进行相应的织入重复代码的处理。

这种写法用散布在各处的Annotation替换了集中在一处的AOP声明。你可以说这是一种创新，你也可以说，这是开历史的倒车。

俗话说，分久必合，合久必分。分有分的道理，合有合的道理，分分合合，就是这个道理。

AOP的作用本来是把散落在各个方法中的重复代码合在一处，进行集中处理。现在，又用Annotiaon把它分散开了。这个游戏倒是有趣。

当然了，喜欢这种声明方式的程序员可以举出若干好处。比如，Annotion比重复代码更加简洁方便，比集中式声明更加一目了然，云云。但这劝服不了我。我仍然固执地认为，这就是一种没事找事型的做法。天下本无事，庸人自扰之。还是那句老话，这是我的个人偏见，不必当真。

1.23 《编程机制探析》第二十二章 互联网应用

发表时间: 2011-10-18 关键字: 应用服务器, web

《编程机制探析》第二十二章 互联网应用

在前面的章节中，我们一直在编程的基本原理和模型中折腾。从本章起，我们将进入真正的应用程序的世界。
《黑客帝国》中，莫菲斯递给尼奥一粒药丸，“欢迎来到真实的世界。”
在计算机界中，也有“真实的世界”（Real World）的说法。比如，我们经常看到这样的话：这种方案看起来很美，但不适用于真实的世界。（That sounds great, but not for real world.）
我不喜欢真实的世界，我只喜欢在理想的模型中畅游。历经多年的应用开发，我早已明白，真实的世界是嘈杂的、琐碎的、无聊的。你很难把东西做得很漂亮，很满意。
为了避免误解，我这里澄清一下。我并非完美主义者，也没有精益求精的精神。我只是希望能够达到一种基本的要求：写出来的东西，用户满意，我自己也满意。
做到这点很难吗？貌似不难，实际却相当难。尤其是复杂度到了一定级别的系统，想做到各方都满意，那几乎是不可能的任务（mission impossible）。
在我使用软件的经验中，只有一些功能简单、明确、固定的小软件，才能让我感到基本满意。
只要软件具有了一定的规模，就一定会有很多不如意的地方。我们（这里，我们是甲方，即软件用户的角色）之所以使用那些软件，不是因为我们喜欢它们，而是因为别无选择。我们只能瘸子里拔将军，从众多不如意中，选一个差强人意的。
这里面有个用户需求问题。用户的需求，都是从现实世界中来的。而程序员要实现需求，则必须根据计算机模型来实现，需要把现实世界中的需求映射到计算机模型中。这两个模型之间，经常存在着难以跨越的鸿沟。程序员怪用户不懂计算机，用户怪程序员不懂业务。最终，妥协的总是程序员。因为，脱离于实际应用的软件是没有用处的，也很难卖到钱。所以，程序员必须紧跟用户的需求。
当用户的需求比较固定的时候，程序员的工作就轻松许多。当用户的需求经常变动的时候，程序员可就苦也。
应用软件大致可以分为两类——产品和项目。产品是指针对某一种行业的通用的软件产品，项目是指针对某一个公司具体业务的软件项目。
有些公司是做网站运营的，不卖产品，而是卖服务。这种运营服务，也可以看做是一种产品。
产品的优点很多。
第一，利润高。写一个好产品，可以卖给整个行业内多家客户。
第二，需求稳定，程序员不需要疲于奔命地应对用户层出不穷的新需求。
产品的缺点也是很明显的。
第一，开发周期长，成本高。一个好产品，从底层架构、功能实现到用户界面，都需要良好的设计。
第二，同质化竞争激烈。由于需求固定稳定，大家都可以了解和学习这些通用的需求，都可以做这类产品，因此竞争激烈。
项目的优缺点和产品正好倒过来。
项目周期短，来钱快。但是，项目的特点是劳动力密集型。没项目时，得养一大帮闲人。项目一旦多了，马上就需要大量的人。想要赚更多的钱，就得接更多的项目，跟起来就越累。这就是劳动力密集型产业的弱点。相

对的，产品只需要几个或者十几个核心开发人员即可。赚钱多少，不依赖于人数，而依赖于产品质量。

以上的优缺点是针对软件公司老总来说的。对于程序员来说。做项目积累的经验是博而杂，做产品积累的经验是少而精。

喜欢做项目，还是喜欢做产品，这要看个人偏好。在我看来，大部分程序员都是喜欢做产品的。因为做项目很累，学到的东西又很散很浅。

但是，真实的世界正好是倒过来的。软件行业中，大部分的职位都是做项目的，少部分才是做产品的。因为，通用产品的需求是固定的，胜者通吃，一两家好的产品，就足以霸占整个行业市场了。而项目就灵活许多了，大大小小那么多公司，每个公司都有不同的需求，软件公司永远不愁没有项目做，只要你吃苦耐劳，并且愿意把自己的利润降到极点。

产品难，项目累，何方才是出路？出路在于项目和产品之间的桥梁。针对一系列中小型公司的种种特殊需求，开发出一系列灵活多样的产品。项目可以容易地产品化，产品也可以容易地项目化（定制化）。这应该是软件业的共识了。

好了，鸟瞰完软件行业的真实世界，我们回到地面，来看一些真实世界中的具体问题。

在前面的章节中，我们讨论的各种编程模型，都是基于单一进程的模型，很少涉及到跨进程的模型。

但是，在真实的世界中，所有的软件模型都是跨进程的。道理很简单，我们前面已经讲过了。独自运行在内存中、不和外部打交道的程序是没有意义的。一个程序要有用处，则必须要有I/O（输入/输出），既要接受用户的指令，又要把执行结果输出到外围设备（硬盘、屏幕、打印机、网络等），才能为人所知，才能为人所用。

前面讲过，而所有的I/O，都是和进程外部状态交互，都是跨进程的。比如，最简单的文件操作，实际上就是调用了操作系统的文件管理进程和硬盘控制驱动进程。再比如，最简单的屏幕显示操作，实际上就是调用了操作系统中安装的显卡驱动进程。

简单的I/O如此，复杂的I/O就不用说了。我们来看一个典型的互联网（Internet）应用的例子。这是个很常见、也很重要的例子。在这个例子中，我们将学习到关于互联网应用（也叫做web应用）的知识。现代的软件中，互联网软件占了很大份额，提供了大多数的工作岗位。其重要性不容忽视。这也是本章的主题。

我们先从互联网用户的角度来看。

一个用户打开网页浏览器，访问千里之外的某个网站（Web Site），看到了一个页面。这里面发生了什么事情呢？

首先，浏览器是一个进程。浏览器进程要根据网址访问互联网上的一个网站，它应该怎么做呢？它应该像一条鱼一样，顺着网线游出去，到处打听，找到那个网站，然后，叼了内容，再游回来吗？

不是这样的。它先把网址包装一下，发给操作系统的网络服务进程。然后，网络服务进程就负责剩下的工作。

当然，网络服务进程也不会像鱼一样游出去。它只是把网络请求再次包装一下，就转交给网卡驱动进程了。然后，网卡驱动进程再次把网络请求包装一下，就把网络请求信息通过网线，或者无线网络，发送到本局域网中的路由器上去了。

这里，原本的网址请求经过浏览器、网络服务进程、网卡驱动进程的一层层包装，已经膨胀为一个包了好几层的大包裹了。

在这个过程中，涉及到极为重要的网络协议层概念。这是计算机网络编程的核心概念。这方面的内容有很多，写得也很透。我没什么可发挥的，这里只是简述一下。

在现实的编程模型中，网络协议层分为应用层、TCP层、IP层、网卡层。

其中，TCP层也叫做传送层（Transport Layer）。TCP就是Transmission Control Protocol（传输控制协议）

的缩写。

IP层也叫做互联网络层（Internet Layer）。IP就是Internet Protocol（互联网络协议）的缩写。在互联网上，每一个直接连接到互联网上的计算机、路由器或者其他设备，都有一个唯一的独立的IP地址。

有读者问了，IP层有地址，TCP层没有地址吗？TCP层是没有地址的。TCP层只有端口（Port），叫做TCP端口，是应用进程向本机网络服务进程申请到的一个整数。用来标志某个TCP通信结点。

TCP端口和IP地址通常是合在一起使用的，合称TCP/IP地址。比如，192.168.1.1是一个IP地址。9052是一个TCP端口。那么，192.168.1.1:9052就是一个TCP/IP地址。

需要注意的是，目前的IP地址大都是4段，叫做IP4，已经不敷使用。以后的IP地址正在向6段发展，叫做IP6，那时候的IP地就是133.189.192.168.1.1这样的了。

在实际的网络服务实现中，TCP/IP也通常作为一层，叫做TCP/IP层。TCP协议和IP协议也通常合称为TCP/IP协议，是网络编程中最为重要的协议。

TCP/IP协议是一个有始有终、有来有回、有请求有反馈的协议。在一个典型的TCP/IP场景中，TCP/IP地址都是成对出现的。一个是客户主机（即浏览器所在的计算机）的TCP/IP地址。一个是服务主机（即目标网站所在的主机）的TCP/IP地址。

一般来说，服务主机的TCP/IP地址（即IP地址和TCP端口）都是公开在互联网公共网络上，是固定不变的，是唯一的公网地址。而客户主机的IP地址只是一个局域网内部的IP地址，只在本网内有意义。至于客户主机中，某个应用程序（比如浏览器）的TCP端口，那更是随即分配的，只在整个网络请求的周期内有意义，用完就丢。

IP层之下，就是网卡层了，也叫做物理层。为了明晰起见，我叫它网卡层。至少现在的计算机架构中还需要网卡。

对应到前面的例子中。浏览器就是应用层。它接受的网址格式，需要遵守相应的应用层网络协议。其中最重要的一种应用层网络协议就是HTTP（Hypertext Transfer Protocol，超链接文本传输协议）。这东西，大家也应该耳闻能详了，无需费舌。如有不明之处，请自行到网上查阅。你看看浏览器地址栏中的内容，打头的就应该是HTTP这几个字母。

网络服务进程负责处理TCP/IP协议，对应的是TCP/IP层。这个网络服务进程有可能是一个进程，同时处理TCP/IP协议。也有可能是两个进程，一个处理TCP协议，一个处理IP协议。还有可能是多个进程，分担着多个TCP和IP协议的处理。

浏览器负责把HTTP协议格式的网址，包装成TCP协议的格式，然后，输出到网络服务进程。HTTP协议包装成TCP协议的过程，有些复杂。

大部分情况下，浏览器需要调用互联网上的DNS（Domain Name Service，域名服务），把网址（即域名）转换成IP地址。这也是一个颇为复杂的服务。这里先不多讲。

网络服务进程负责把TCP协议包装成IP协议，转给网卡驱动进程。网卡驱动进程在IP协议包上在加上自己的网卡物理地址（Mac Address），就把网络协议包顺着网线发出去了。

网络中有一个重要的角色，叫做路由器（Router）。路由器是构成所有网络的基石，无论是局域网，还是互联网。

路由器是一个人际高手，维护着一个巨大的社交网络。二十一世纪，什么最重要？人脉！路由器有一个花名册，其中记录了相连的其他路由器的关系网，这个花名册叫做路由表。依靠着这个巨大的关系网，路由器就可以把经由自己的网络协议包正确地转发给下一个路由器。

网络协议包就像一个四处托关系求人办事的人，总是先从最近的关系户（路由器）打听。路由器就像一个热心

的中介人一般，再把网络协议包介绍给关系网中下一个离目标主机更近的中介人（路由器）。
需要注意的是，路由器只解析IP层和物理层。路由器只关心目标IP地址。对于路由器来说，TCP端口和HTTP协议内容都属于不可理解的内容，它对此也漠不关心。
局域网中的路由器接到网络协议包，在外面添加上自己的IP地址和网卡物理地址，根据自己的路由表，就把网络协议包发了出去。这时候，网络协议包就出了局域网，进入了外网（即互联网）。
互联网上的道路和现实世界中的道路一样，是分级别的。分国道、省道、市道、县道。互联网的各个级别的道路上，都分布着各个级别的路由器。
一路上，网络协议包遇到了很多路由器。这些路由器都很热心，甚至过于热心。他们都把这个协议包拆开看一看，看到里面的目标IP地址之后，就去查自己的路由表，看看这事儿能不能帮上忙。路由表中记录了一些相邻路由器的信息。如果路由器觉得帮不上忙，就把协议包扔了，让别人去操心。如果能帮上忙，就把协议包继续向路由表中的下一个路由器传。注意，外网路由器在转包的过程中，是不会添加自己的IP地址的，他还没有热心到这种程度。所以，网络协议包的去程和回程，不一定遵守同一条路线。完全是途中的路由器随即决定的。
网络协议包如同接力棒一般，在互联网上的路由器之间传递着。这种传递，通常是从下级路由器到上级，道路宽度从细到宽，比如，可能从电话线ADSL到光纤。然后再从上级到下级，找到目的网站。
终于找到组织了。网络协议包热泪盈眶。目标网站主机抓到这个网络协议包之后，就不再转发，而是自行处理。至于怎么处理的，暂且不谈。我们回顾一下网络协议包在路由器关系网中的四处寻觅的艰辛路程。
网络协议包一旦出了本机（浏览器所在的主机），就进入了路由器的关系网络，直到最终到达目标网站主机。
网络协议包出了本机，在路由器关系网络中畅游的整个过程中，一直是在IP协议层和物理层两层中打转。
当网络协议包就脱离了路由器的关系网，进入了目标主机之后，就进入了更高的协议层——TCP层和HTTP层。
目标主机的网络服务进程，接到了网络协议包之后，根据其中的目标TCP端口（这是目标网站进程的TCP端口），把网络协议包转发给到本机内运行的网站进程。至此，一次TCP/IP协议层上的通信，就完成了一半。
网站进程收到了网络协议包之后，取出其中的HTTP协议内容，就开始忙活，最后得到一个结果（通常是一个页面）。然后，网站进程再把结果打包成一个结果协议包，向客户浏览器所在的局域网的路由器的IP地址发送。这个过程和来时一样。路漫漫而修远兮，吾将上下而求索。不再细述。
路由器接收到结果网络包之后，发现目标IP地址正是自己的IP地址，就抓了下来，不再转发。路由器首先将自己添加的IP地址和物理地址剥去，里面就露出了客户主机的局域网IP地址和物理地址。然后，路由器就把网络协议包发向局域网。客户主机受到这个包之后，就知道自己要的东西到了，一个Nice Catch，抓住，就开始处理。
这个过程类似于网站的处理过程。客户主机的网络服务进程先根据TCP端口（浏览器的TCP端口）把网络协议包发给浏览器进程。至此，一次TCP/IP通信就完整地结束了。有始有终，有来有去，有请求有反馈。然后，浏览器进程再处理其中的HTTP协议内容，把网页显示到屏幕上。
需要提醒的是，为了简化起见，我们假设一个网络包去，一个网络包回。在真实的世界中，并不一定如此。如果网络协议包过大的话，会被拆成多个小包。细节是魔鬼。诚不我欺。真实的世界中，充满了各种各样的琐碎细节。
我们可以看到，一次简单的网站访问，牵扯到多少人力和物力。可想而知，互联网路上是多么繁忙，各式各样的网络协议包四处流窜。
值得庆幸的是，浏览器进程并不需要知道这些。浏览器进程只知道申请一个TCP端口，把HTTP请求发过去就行了。这就是网络协议分层的意义所在。各层管各层的事情。
对于浏览器来说，一次复杂的互联网访问，和一次本机的文件访问，没有什么太大区别，都是对进程外部状态

的I/O（输入/输出）。

以上的流程，是从客户端（浏览器）的角度来看的。对于我们开发人员来说，我们更关注的是服务器（网站主机）一端的实现。这就是本章及后续章节的主题——Web应用。

这里，我们要说明一下internet和web的区别。Internet是互联网的意思，这是一个计算机术语，是一个新词。Web是一个老词，原意是蛛网。Internet更广义一些，internet应用（互联网应用）包括所有基于互联网的软件，包括小软件，也包括大型网站。而Web则狭义一些，web应用主要指网站。本章的题目叫做互联网应用，但这么写只是为了好看，实际上，本章及后续章节的主要内容就是网站开发，即Web应用。

现在，让我们把目光转向服务端，看看在一次典型的网站访问中，网站主机中发生了什么。

基于网络协议分层的机制，我们不需要考虑TCP/IP层以下的事情。我们只需要考虑HTTP协议层。

当网站接收到浏览器（客户端）发来的HTTP协议包的时候，就会进行相应处理，提供结果，反馈给客户端。这个处理过程有可能非常简单，也有可能非常复杂。

在互联网刚刚出现的原始年代，网站提供的内容都是静态的，只需要根据客户请求，找到相应的网页文件，发给客户就是了。什么叫网页文件？就是HTML文件。HTML是Hyperlink Text Markup Language（超链接文本标记语言）的缩写，是一种XML格式的文本。

静态内容就是一系列的HTML文件。每个HTML文件中通常包括HTML文本、CSS（cascade style sheet，用来定义HTML元素的格式）、Javascript、图片等。其中，图片是最大的资源，其余的都是文本资源，个头比较小。

每个HTML文件都有自己的网址，HTML文件中包含的每一个资源，比如，CSS、Javascript、图片等，也都有自己的网址。因此，一次HTTP请求并不是一次就玩了，通常会引起一系列的内含资源的HTTP请求。

在寂静年代，只需要一个网页设计师，就可以建设一个网站。

历史的车轮总是滚滚向前。随着技术的发展，用户的口味越来越重，错了，是越来越丰富多样，静态内容已经无法满足用户的精神需求了。这时候，就出现了动态内容。于是，网站的建设方式发生了深刻的变化。在静态内容时代，网站建设领域是网页设计师的天下。到了动态内容时代，网站建设领域就成了程序员的天下。因为，动态内容的提供，必须要通过程序。

一开始的时候，典型的动态内容网站的架构是这样的。一个Web Server（也叫做HTTP Server）中，配置了一个个CGI（Common Gateway Interface，公共网关接口）程序。

这些程序为什么叫做CGI程序？我是不太了解的。我进入Web开发领域的时候，CGI模式已经是一种过时的开发模式，虽然还在广泛使用，但是已经不为追赶新潮的程序员所喜。

Gateway是网关的意思，表示内网（局域网）到外网（互联网）的关口，通常就是指局域网的出口路由器。CGI应该就是指网站入口点的意思吧。

CGI程序可以是编译语言，但更多的是解释语言，因为解释语言无需编译，更加方便。由于这个原因，CGI程序通常被称为CGI脚本（Script，通常指一小段解释程序的代码）。

CGI的一个主要问题在于效率问题。当用户发来一个动态内容请求的时候，Web Server就得把对应的CGI进程运行起来。处理完请求之后，再结束进程。也就是，每一个请求都对应一个进程的生灭。而进程是相当昂贵的资源。

除此之外，还有一个原因。那就是CGI模式下，CGI脚本都是一个个分离的脚本程序，彼此之间很难共享什么东西。一个用户的每一次请求都相当于一次独立的请求。一个用户的多次请求，很难形成连续的关系。

这在动态内容时代的初期，不是什么问题。但是，随着用户的要求越来越高，用户希望自己的一连串请求之

间，能够连续起来，保持一定的状态。这种一连串请求之间的状态保持，有一个专有名词，叫做Session（会话，会谈）。因为客户端和服务器端之间的一连串请求、反馈就像两个人你一句我一句谈话一样。

两个人初次邂逅，还不认识，等搭上了第一句话之后，就开始热情地交谈起来。这时候，两个人就成了熟人，每一句话都可以建立在前一句的话的基础上，而不用每一次都需要重新搭话。这就形成了一种客户端和服务器端之间的连续的会话状态。

会话（Session）的需求同时对HTTP协议和Web开发模式提出了要求。

HTTP一开始是无状态的协议，一次HTTP请求就是一次TCP/IP连接，然后就断开，结束了。但是，随着时代的发展，无状态的HTTP不能满足人们的需求，人们又在HTTP协议上增加了状态。这是非常重要的概念，我们后面有专门的章节来讲解。

用户连续多次请求之间的状态保持，不仅需要HTTP协议的支持，还需要Web开发模式的支持。因为Web Server总得找个地方把用户的一连串请求的状态存放起来，以便该用户的后续请求可以共享这些状态。在CGI模式下，CGI脚本之间是分离的。要想共享状态，只能借用文件、数据库、网络服务器等进程外I/O存储设备，而且每次都得打开/关闭外部的I/O设备，这个开销也是不小。

为了解决CGI模式的问题，Application Server（应用服务器，简称App Server）的开发模式出现了。App Server是一个独立运行的服务进程。它通过TCP协议与web server交互。当web server遇到动态内容请求时，就把请求转发给App Server处理。App Server就会启动一个线程来处理该请求。

App Server的模式好在哪里呢？首先，线程比进程轻量一些。其次，所有的处理请求的线程都运行在同一个进程空间内，因此，可以共享进程内的资源，最主要的是内存资源，还有一些保持打开状态的I/O设备。

App Server模式一出现，立刻大行其道，占据了CGI模式的不少份额。我赶上的就是App Server这一拨。App Server模式给了程序员偌大的发挥自由。各种层次不穷的网站开发架构模式就是在App Server的构架上创建出来的，并影响到App Server模式以外的领域，比如，CGI模式。

App Server同时也是HTTP Server，可以单独作为Web Server来运行。一个App Server就可以同时处理静态内容请求和动态内容请求。

不过，术业有专攻，App Server专攻动态内容请求，在处理静态内容请求方面不是很擅长。因此，App Server常和Web Server一起组合使用。App Server负责处理动态内容请求，Web Server负责静态内容请求。因为，动态HTML页面中经常含有各种静态资源（比如，很多图片资源），所以，一次动态内容请求，通常会引起十几次静态资源请求。

一次典型的动态内容请求的过程是这样的。App Server收到HTTP请求，就把HTTP请求转发给内部运行的某个网站应用程序（Web Application，简称web app）。一个App Server里面可以跑多个web app，但通常情况下，只跑一个web app。

Web app收到HTTP请求之后，就回去查找动态内容，典型的，是在一个数据库中查找。Web app需要访问App Server进程之外的数据库进程，并获取查询结果。然后，web app根据查询结果，拼装成一个HTML页面，再交给App Server进行后续的处理。动态拼装页面的技术有个名字，叫做模板（Template）技术。这曾经是很重要的一个技术。那时候，各种模板技术层出不穷，争芳斗艳。

历史的车轮继续向前滚动。随着时间的推移，web应用有了长足的进展。无论是在前台（指发送到浏览器的HTML页面部分）还是后台（指服务器端），都发生了深刻的变化。

首先来看前台HTML部分。动态HTML页面的产生，有两种方式。一种是后台服务器端生成。一种是由HTML页面包含的Javascript生成。

第一种方式就是我们提到的Template（模板）技术。第二种方式中，浏览器中的Javascript可以获取用户的操作，直接向后台服务器发送HTTP请求，然后把动态内容取回来，填充到当前的HTML应用中。这种技术有很多名字，ajax，富客户端（Rich Client），等等。这些表述都不太直观。在本书中，我使用Javascript HTTP这个词。

随着Google公司的在线文档办公软件和Gmail的推出（这两个卓越的产品都是用Javascript开发的），Javascript HTTP技术奠定了自己坚实的基础，逐渐成为了web应用交互的主流开发模式。

与此同时，其他的富客户端（Rich Client）技术也进入了人们的视野，主要是Adobe公司的Flash技术。不过，这不在本书讨论之列，先不去管它。

随着Javascript HTTP技术的兴起，HTML页面越来越静态化，曾经喧嚣一时的Template技术逐渐失去了用武之地，陷入了沉寂，慢慢淡出了人们的视野。当然，Template技术的地位只是降低了，并不是消失了。而且，Template还广泛地用于其他各种需要动态生成文本的领域，比如，动态生成代码和查询语句，等等。

以上是前台的状况。我们现在把目光投向后台。服务器端的发展主要在于架构的扩展上。随着用户越来越多，请求越来越多，网站规模也越来越大，一个服务器已经不能满足要求了。这时候，集群（Cluster）技术就出现了。局域网中的多台服务器组合在一起，共同响应用户的请求，这种架构叫做集群（Cluster）。

一个典型的集群架构是这样的。局域网内的计算机分为三层。第一层是十几台或者更多的Web Server，负责应对用户浏览器对静态页面资源（静态文本和图片）的请求。第二层是几台App Server，负责应对用户浏览器对动态页面资源的请求。第三层是一台或者两台数据库服务器（db server），用来响应App Server的请求。

Web Server上全都是静态文件资源，每台Web Server上的静态资源都是一样的，不会改变，是无状态的，因此可以任意扩展。Web Server可以有任意多台，上百台，上千台都可以，甚至可以放到局域网外的互联网上任何一个网络结点上。这也是静态网站的常见策略。

App Server由于需要保持用户会话（Session）状态的，可扩展性就受到了限制。当App Server集群和某个浏览器之间建立了用户会话（Session）关系，那么，该集群必须保证，下一次用户请求过来的时候，还能够找到这个会话状态。实现方案主要有三种。

第一种方案，叫做Session Sticky（粘住会话）。意思就是，当一个浏览器访问集群中某一台App Server时，建立了会话关系。下一次该浏览器请求再次过来的时候，还由这台App Server负责接待。这就像客户专员原则一样。一个客户服务人员跟踪服务一个客户，一直到底。而不是每次都换一个客服人员。

这种方案的缺点在于，难以实现负载平衡。有的App Server客人多，有的App Server客人少，无法最大限度地发挥集群的效能。

第二种方案，叫做Session Replication（会话复制）。意思就是，当一个浏览器访问集群中某一台App Server时，建立了会话关系。那么，该会话关系会广播（Broad）到集群中的其它App Server中。下一次该浏览器请求再次过来的时候，任何一台App Server都可以负责接待。

这种方案的优点在于容易实现负载平衡，大家的工作量都差不多。缺点在于制造了大量的网络通信。

第三种方案，叫做Session Cache Center（会话缓存中心）。意思就是，专门用一个服务器来存储用户的所有会话状态。当一个浏览器访问集群中某一台App Server时，建立了会话关系。这个App Server自己不保存这种关系，而是交给客户关系经理，即会话缓存中心。下一次该浏览器请求再次过来的时候，任何一台App Server都可以负责接待。这台App Server首先要去会话缓存中心服务器中查找是否已经存在了该会话关系，如果已经存在，那么就取出来继续用，如果不存在，就建立一个会话关系。处理完毕之后，再把更新后的会话状态存回到会话缓存中心服务器中。

这种方案的优点在于结构简单，实现容易，也不会产生网络广播，而且，缓存中心不仅可以用来缓存会话状态，还可以缓存其他信息。这种方案的缺点在于，一旦缓存中心出了问题，所有App Server都得忙乱一番，失去响应一段时间。不过，这种缺点可以用备份缓存中心来缓解。因此，这种方案被大量采用。

由于会话缓存中心这种方案的兴起，CGI开发模式又焕发了新的生机。在这种Session方案下，App Server模式和CGI模式站在了同一个起点上。大家处理Session都是要进行网络通信。谁也别说谁。而且，在CGI模式下，还可以省掉App Server。可以直接把CGI脚本放置在Web Server中，这节省了成本和资源。

而且，现在还有一种说法。多进程模式比单进程模式更加健壮。在单进程模式下，一个线程出了问题，可能会引起整个进程死掉。这对于网站来说，是难以容忍的。而多进程模式下，一个处理请求的进程死了就死了，不会影响到网站服务主进程。从这个角度来看，CGI的多进程模式还是一种优点了。这真是，风水轮流转，老树发新枝。

最后，我们再来看db server（数据库服务器）层。由于数据库中装的全都是数据，而且是随时可能会改变的数据，而且是随时可能会变的海量数据。这是一个极为庞大的状态。这种状态几乎是不可能即时复制和传输的。因此，db server是最难扩展的。通常来讲，db server只有一台。最多还有另一台备份数据库，提供一些旧数据的只读服务。

在我看来，数据库集群的发展方向不在于即时复制和传输状态，而在于分布式存储管理。即数据分区域存放到不同的数据库服务器上。但是，这无形中就增加了数据库表格关联查询的难度，从而增大了数据库管理员和数据库应用程序员的工作难度。

1.24 《编程机制探析》第二十三章 HTTP

发表时间: 2011-10-18 关键字: web, 框架, 编程, 网络协议

《编程机制探析》第二十三章 HTTP

HTTP是Web应用开发中最为重要的协议。但是，在实际的Web应用开发中，相当多的程序员根本就不了解HTTP是怎么回事，也照样编写Web程序。我就曾经是其中的一员。这种现象的产生，与现代软件业的开发模式大有关系。

这事儿，说起来话就长了。软件开发管理，一向是管理界的一大难题。因为，没有哪个程序员愿意被管理。每一个程序员都觉得自己是人才，而不是人力资源。人才，尤其是智力人才，应该是有发挥自由度的，怎么能像体力工人一样被管理呢？

但是，胳膊扭不过大腿。公司管理层就不信这个邪，非要解决这个管理上的难题。随着软件工程管理的发展，大量的程序员都被管理起来了，成为办公室蓝领工人，甚至更惨，成为码农。怎奈心比天高，却命比纸薄。呜呼哀哉。

为了照顾程序员们的情绪，有人专门著书立说，为程序员力争更多的自由和权益，比如，有一本叫做《人件》（peoplesoft）的书，就很有名。程序员们都希望，这本书能够摆在公司老板的案头。但据我所知，公司老板一般对这些东西都不感兴趣，人家有更高层次的精神追求，早就超越了这个执着于管理流程细节的层次。人家喜欢看的是《狼图腾》这类宣扬狼性企业文化的书籍。这样能够强化他们在“商场如战场”的竞争心态和优胜心理。我这么写可能会有失偏颇。因为我的样本很有限。我只偶然接触过那么几个软件业老板，但奇怪的是，他们都推崇《狼图腾》。

这是为啥子呢？很值得深究。不过，那不是本书的内容。我这里只提一点，到了那个层次，他们面对的真实世界和我们工薪阶层是不同的。他们直面市场，直面社会上的三道九流、黑白两道。他们的世界观和道德观，和我们这些生活圈子狭窄的工薪阶层是大为不同的。

这一点，我们要理解。理解万岁。虽然他们不一定需要理解我们，但我们一定需要理解他们。这就是真实的世界。我们不用直面市场，但我们需要面对上司和老板。我们想的是，如何证明自己的价值，从老板口袋中掏出更多的薪水。老板想的是，我们如何任劳任怨地跟在他的后面，跟他一起向竞争对手撕咬。

话题扯远了，我们回到软件工程管理的话题上来。在现代的软件开发模式中，大量的程序员都在固定的开发框架中进行开发工作，每个人面对的都是一个很狭窄的领域。大部分程序员就如同一个巨大的装配线上的螺丝钉，随时可以被替换。老板不用担心这样的程序员能够带走公司的核心技术。

不得不说，这是软件管理上的巨大进步。尤其是对于做项目的公司来说，尤其如此。设身处地想一想，如果我们处在老板的位置，是不是也希望采用这样的模式？

但是，这种开发管理模式对于程序员的技术全面发展来说，却是有阻碍的。怎么克服呢？公司不是学校，可能会提供必要的职业培训，但没有义务保证程序员的全面发展。程序员只能靠自己来弥补一些必要的重要的知识。

对于HTTP的重要性，我一开始是没有认识的。我那时候主要专注于各种Web开发框架的设计理念。我觉得，那些程序设计相关的东西，才是程序员的主业。至于HTTP，只是一种基础的协议，没必要深究。

当我开始尝试自己设计Web开发框架的时候，我发现，我绕不开HTTP。我必须深入了解HTTP，才能够继续前

进。我还是不愿意在HTTP协议上花时间，我只是借鉴其他开源框架的相关代码。结果，我很快就发现，这样的做法效率极低，事倍功半，很多代码我不明其真意。于是，我转换了做法，开始认真地学习HTTP。思路一下子打开了。再看之前的代码，豁然开朗。

然后，我再回头看以前写的代码，包括自己写的和其他人写的代码，越看越心惊。我发现了大量的地雷代码。比如，有些人为了方便起见，在代码中滥用HTTP Session来存储数据。这些代码在开发环境中是测不出问题的。在生产环境中，如果不遇到极端环境，也是测不出问题的。只有在某些极端环境下，才可能会导致问题。一个人这么用，没出过问题，其他人觉得方便，也跟着这么用。

类似于这样的误用，比比皆是。究其原因，都是因为对编程模型没有真正的理解。比如，我在前面的《线程》章节中提到的，有些人把线程同步锁加在局部变量上，是因为没有理解线程同步模型所致。

要避免这类很难测出问题的地雷代码，有两种方法。

第一种方法是把这类问题全都列出来，放在编程规范中作为反面例子，要求所有程序员注意。这种方法治标不治本，知其然不知其所以然。

第二种方法就是深入理解编程模型，这样从根本上就避免了问题的发生。

第二种方法花的时间长，费的功夫多，但是非常值得。对于Web应用开发人员来说，HTTP就是值得深研的Web通信协议。

HTTP的基本结构，是非常简单的。HTTP的官方定义在W3网站上，请参阅：

http://www.w3.org/Protocols/

HTTP协议包括两种消息包，一种是HTTP Request（请求），一种是HTTP Response（响应，回应）。

服务端不能主动连接客户端，只能被动等待并答复客户端请求。客户端连接服务端，发出一个HTTP Request；服务端处理请求，并且返回一个HTTP Response给客户端；本次HTTP Request-Response Cycle（请求-回应周期）结束。

HTTP Request和HTTP Response的基本结构很简单。HTTP Request主要包括网址、用户信息等内容。HTTP Response主要包括状态码、HTML页面等信息。

但是，对于Web开发人员来说，追踪HTTP Request和HTTP Response的内容却非常重要。我们可以在App Server端，打开HTTP日志，就可以追踪HTTP Request和HTTP Response。我们还可以在浏览器端追踪HTTP内容。方法是使用各种HTTP监视工具。我这里提供一些关键字，供读者借鉴：Fiddler，HTTP Watch，HTTP Debugger，HTTP Sniff，Fire-ware，Network Monitor，HTTP Monitor。

有了这些工具，我们就可以清楚地看到HTTP协议的来来去去，获得直观上的认识。

HTTP本身是一种无状态的协议，一次请求/响应周期完成之后，TCP/连接就断了，事情就结束了。浏览器下次再访问同一个网站的时候，就如同访问一个新的网站，而网站也如同接受一个新的请求。这样就达到了一种日久常新、每次都是新面孔的效果。

关于婚姻家庭，有一种说法，叫做七年之痒。就是喜新厌旧的意思。对于已经到了“相看两厌”状态的夫妻来说，HTTP的这种无状态机制是非常有效的，每天都跟新婚一样。

在网站还是提供内容（content）的时代，HTTP的无状态特性工作得很好。但是，好景不长。随着Web领域的蓬勃发展，网站不再限于提供内容，而是开始向应用（application）进军。

应用程序本来是C/S结构（Client/Server，客户端/服务器端）的天下。但是，C/S结构的弱点在于吞吐量太小，能够同时响应的用户请求数量太少，因为C/S架构的应用程序总是保持着长期的TCP/IP连接，而一个系统能够支持的TCP/IP连接数量是有限的。

HTTP是一次请求/回应完成就立刻断开TCP/IP连接的协议。因此，基于HTTP协议的网站能够支持更大的用户访问量。一些应用开发人员就把目光转向了HTTP协议，希望能够开发出基于HTTP协议的Web应用程序（即Web app）。

但是，这就有一个矛盾。Web app需要保持用户和服务器端的会话（session）状态，而HTTP协议是无状态的。如何解决这个矛盾呢？如何让无状态的HTTP协议支持会话（session）状态呢？

聪明的研发人员想出一个简单易行的方案：发放临时号码牌。

这个模型很简单，却很有效。充分体现了设计人员的聪明才智。我们来看一下这个模型的工作机理。

假设App Server是一个大型游乐场的存包处，浏览器发出的HTTP Request是一个顾客。

当顾客第一次来到存包处的时候，存包处管理员为顾客分配一个储物柜，相当于web server用来存放用户会话（Session）状态的存储空间。然后，管理员把一个该储物柜对应的号码牌交给这个顾客，作为取包凭证。这个号码牌有个学名，叫做Session ID。

管理员在顾客回去的时候——即HTTP Response返回到浏览器的时候，会要求顾客随身带着这个号码牌。并嘱托顾客下次来的时候，一定要记着带着号码牌。

顾客下一次来的时候，要把随身带着的这个号码牌，交到存包处。管理员一看到号码牌，就知道，这是个老顾客了，就根据号码牌（Session ID）找到相应的储物柜，即存放用户（Session）会话状态的存储空间。管理员根据顾客的要求，更新储物柜里的Session状态。

同现实世界中的储物柜一样，App Server中的Session存储空间也会过时的。如果相隔时间太久，Session就会过期了，号码牌（Session ID）就会失效了。App Server就会给顾客重新分配一个Session存储空间和一个Session ID。

就这样，利用一个很简单的模型，就实现了用户会话（Session）状态的保持。

我们下面来看具体实现方案。无论是Session存储空间，还是Session ID，都是服务器端分配的。Session ID的形式，Session存储空间的位置，都是由服务器自行决定的。这部分的实现很简单，没什么可说的。

我们关心的问题是：HTTP Response如何把Session ID带回到浏览器；浏览器如何存放这个Session ID；浏览器下次访问这个网站的时候，又如何让HTTP Request带上这个Session ID。

我们先来考虑最主要的问题：浏览器如何存放和使用这个Session ID？

当浏览器初次访问一个网站，获取到HTTP Response的时候，同时也就获取了一个Session ID。浏览器需要把这个Session ID存储起来。下次，再访问这个网站的时候，要把Session ID放在HTTP Request里面发过去。

清楚了这个流程之后，浏览器存储Session ID的方案就呼之欲出了。浏览器只需要在本进程内存中维护一个“网站地址->Session ID”的映射表，就可以完成这个需求。

浏览器每次访问网站的时候，都去查这个“网站地址->Session ID”映射表。如果该网址存在于映射表中，就把对应的Session ID取出来，放到HTTP Request里面，发送给网站服务器。如果该网址不存在于映射表中，就直接发送没有Session ID的HTTP Request。网站服务器就会给这个浏览器新分配一个Session ID，放在HTTP Response里面，发给浏览器。浏览器接收到HTTP Response里面的Session ID，就放到“网站地址->Session ID”映射表里。

解决了这个主干问题之后，我们再来看枝节问题：HTTP Response是如何把网站服务器分配的Session ID带给到浏览器的；HTTP Request又是如何把Session ID带给网站服务器的。

要解答这两个问题，我们需要了解HTTP Request和HTTP Response的更具体的结构。

HTTP Request包括Request Line、Request Headers、Message Body等三个部分。

（1）Request Line
这一行由HTTP Method（如GET或POST）、URL、和HTTP版本号组成。
例如，GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.1
GET http://www.google.com/search?q=Tomcat HTTP/1.1
POST http://www.google.com/search HTTP/1.1
GET http://www.somsite.com/menu.do;jsessionid=1001 HTTP/1.1

（2）Request Headers
这部分定义了一些重要的头部信息，如，浏览器的种类，语言，类型。Request Headers中还可以包括一种叫做Cookie的Request Header。例如：
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
Accept-Language: en-us
Cookie: jsessionid=1001

（3）Message Body
如果HTTP Method是GET，那么Message Body为空。
如果HTTP Method是POST，说明这个HTTP Request是submit（提交）一个HTML Form（表单）的结果，那么Message Body为HTML Form里面定义的Input属性。例如，
user=guest
password=guest
jsessionid=1001
如果你不知道什么叫做HTML Form（表单）的话，请回忆一下你登陆某个网站时的对话框，你需要输入用户名和密码，那些输入框就属于一个HTML Form。这些都是HTML的基本元素。对于Web开发程序员来说，对于HTML的基本了解也是必要的。
注意，如果把HTML Form元素的Method属性改为GET。那么，Message Body为空，所有的Input属性都会加在URL的后面。你在浏览器的URL地址栏中会看到这些属性，类似于
http://www.somesite/login.do?user=guest&password=guest&jsessionid=1001

从理论上来说，这3个部分（Request URL，Cookie Header, Message Body）都可以用来存放Session ID。
由于Message Body方法必须要求一个包含Session ID的HTML Form，所以这种方法不通用。
我们再来看HTTP Response。
HTTP Response包括Response Status、Response Headers、Message Body两个部分。
（1）Response Status
这就是一个数字，表示成功与否的状态码。这个状态码有特定的含义，不能用于传输Session ID。
（2）Response Headers
这部分可以做文章。可以加一个叫做“set-cookie”的response header，比如，“set-cookie: jsessionid=XXXX”
（3）Message Body

这部分就是HTML文本了。这里面也可以做文章。
可以在HTML中所有的网址链接后面都加上“;jessionid=XXXX”的后缀。比如，<h
ref=”\mailbox;jessionid” />。
如果页面中存在HTML Form，还可以在action的网址链接后面加上“;jessionid=XXXX”的后缀。还可以加一个隐藏的hidden input。比如，< hidden name=”jessionid”，value=”XXXX” />。

我们分析上面的Request和Response结构，可以把它们的结构对应起来。
Request Headers <-> Response Headers
Request Line <-> Response HTML
这正是HTTP Session的两种主要实现方案——cookie header和URL Rewriting（URL重写）。
（1）Cookie Header
Web Server在返回Response的时候，在Response的Header部分，加入一个“set-cookie: jsessionid=XXXX”的header属性，把jsessionid放在Cookie里传到浏览器。
浏览器会把会把Cookie存放，下一次访问Web Server的时候，再把Cookie的信息放到HTTP Request的“Cookie”header属性里面，“cookie: jsessionid=XXXX”，这样jsessionid就随着HTTP Request返回给Web Server。
（2）URL重写
Web Server在返回Response的时候，检查页面中所有的URL，包括所有的连接，和HTML Form的Action属性，在这些URL后面加上“;jsessionid=XXX”。
下一次，用户访问这个页面中的URL。jsessionid就会传回到Web Server。

URL重写需要处理整个HTML里面的所有URL链接，效率很低，这种方案很少采用。
最通用的HTTP Session实现方案是Cookie Header。这种方案需要浏览器一方的支持。
Cookie到底是什么？这可能是困扰很多人许久的问题。这个词经常出现，却很容易引起混淆。我很早就知道cookie这个词，但是，我对cookie的概念一直很模糊。直到清楚了HTTP协议之后，我才知道cookie是怎么回事。
cookie的原意是小饼干，在web领域中，引申为一小段信息的意思，相当于在服务器端传给浏览器、并要求浏览器每次都回传的小纸条。
Cookie分为两种，Session Cookie（会话cookie）和Persistent Cookie（持久cookie）。两者的格式、使用模式都是一样的。两者的区别主要在于生命周期上。
Cookie是由服务器端首先发出的小纸条，由HTTP Response里面的set-cookie header来指定。浏览器只负责在每次访问某个网站的时候，把网站服务器端发来的cookie原封不动地发回去。Cookie的内容，包括生命周期和作用域，都是由服务器端指定的。
在HTTP Response的set-cookie header里面，可以包括Expires和Max-Age这样的定义Cookie生命周期的属性，还可以包括Domain和Path这样的定义作用域和存储位置的属性。
这些属性的含义都很简单，都是些资料性知识。读者感兴趣的话，可以自行查阅HTTP规范。其中，Domain的含义需要稍微说明一下。一般来讲，Domain（域名）就是主机名，Host Name。比如，blog.myweb.com。这个blog.myweb.com就是一个网站域名。这个网站还可以把cookie的

Domain也可以定义为比本网站域名更上一级的域名，比如，myweb.com。

Session Cookie的生命周期等同于浏览器。浏览器一关，Session Cookie也就消失了。
Persistent Cookie的生命周期大于浏览器。浏览器关闭了，Persistent Cookie还在。下一次浏览器打开，还可以用到Persistent Cookie。
Session Cookie通常用来存放Session ID，因此而得名。
Session Cookie的内容是什么？前面我们讲到了浏览器内存中的“网站地址->Session ID”映射表。那只是一种简化的表达。实际上，对于浏览器来说，它根本就不关心什么Session ID。它把服务器端发来的cookie看做一个整体，它把cookie直接放在“网站地址->cookie”映射表中。这个映射表就是Session Cookie。因此，Session Cookie的内容就是“网站地址->cookie”。
浏览器访问网站的时候，会把整个cookie放到HTTP Request里面一起发过去。
由于Session Cookie的生命周期等同于浏览器的生命周期，因此，Session Cookie一般是存放在浏览器进程的内存中的。也有的浏览器把Session Cookie存放在文件中，等到浏览器关闭时，再把Session Cookie文件删除。
Persistent Cookie的生命周期大于浏览器的生命周期，一般是存储在文件系统中的。
Persistent Cookie的使用方式和同Session Cookie类似。当浏览器访问某个网站的时候，浏览器会在某一个文件目录下查找是否存在该网站的文件cookie。如果存在，就把该文件cookie的内容发送到网站服务器。
同Session Cookie的映射表一样，每一个Persistent Cookie也是对应一个网站地址的。
根据对应网站地址的性质的不同，Persistent Cookie又分为两种——主站Cooke和他站Cookie。
当浏览器访问一个网站（称为主站）的时候，主站可以通过HTTP Response把一个Persistent Cooke发给浏览器。这种Persistent Cookie叫做主站Cookie。主站Cookie里面通常包含用户名、密码等信息，以便自动登录。
浏览器得到的主站页面中有时候会包含其他网站的资源链接（称为他站），在访问他站资源的过程中，他站也有可能通过HTTP Response把一个Persistent Cookie发给浏览器。这种Persistent Cookie叫做他站Cookie。他站通常是一些广告网站。他站Cookie通常都是一些用来统计用户访问行为的Cookie。
综上所述，浏览器支持的Cookie有三种：Session Cookie，主站Persistent Cookie，他站Persistent Cooke。浏览器一般都提供了这三种Cookie的配置界面，用户可以控制浏览器支持哪种类型的Cookie。
一般来讲，Session Cookie是一定要打开的。大部分（如果不是所有的）需要登录的网站，都需要浏览器支持Session Cookie。
如果需要自动登录的话，主站Persistent Cookie是要打开的。至于他站Persistent Cookie，除非你乐于提供自己的网站访问行为模式，否则，不要支持他站Persistent Cookie。
可以看到，HTTP Session的工作模型说起来简单，但是，实现起来，还真是挺麻烦的。这些麻烦都是Session ID这个东西带来的。
现在，我们重新思索这个问题，我们真的需要Session ID这个东西吗？Session ID不过是服务器端用来识别浏览器身份的一个随机号码牌。难道我们非得用这种方式来识别浏览器吗？我们就没有别的方法来识别浏览器身份了吗？
方法是有的。而且更加简单直观。前面章节中详细讲解了一次网站访问的过程。我们知道，浏览器发出的网络协议包里面是包含了本机IP地址和本浏览器TCP端口信息的，而且还包括了该机所在局域网的路由器公网IP地

址。服务器完全可以把这些信息组合作为Session ID存放起来，也可以根据该信息组合识别出浏览器的身份。在这种方案中，根本就不用分配什么临时的Session ID，也不用让HTTP Response和HTTP Request把Session ID传来传去。浏览器也不用考虑如何存储和安排Session ID。显然，这个方案更优。

那么，为什么不采用这种方案呢？我们仔细想一想，就可以发现这种方案的问题所在。

TCP端口和IP地址信息，是属于TCP/IP层的内容，并不是HTTP协议层的内容。这就意味着，Session状态的保持，必须在TCP/IP层上实现。

这就和HTTP协议层没什么关系了。Session不叫做HTTP Session，而叫做TCP/IP Session了。而且，Session的实现绑定在TCP/IP层上，不太符合网络协议分层的初衷。

1.25 《编程机制探析》第二十四章 HTTP要点

发表时间: 2011-10-18 关键字: html, 互联网, 编程

《编程机制探析》第二十四章 HTTP要点

上一章讲解了HTTP Session和HTTP Cookie的基本概念，这一章，我们通过具体例子，深化对HTTP的理解，从而掌握其要点。
我们先来看一个访问一个技术网站的例子。这个技术网站叫做theserverside.com。
我们在浏览器中访问www.theserverside.com这个网址。我们的浏览器是支持Session Cookie的。浏览器会发出如下的HTTP Request。注意，为了节省篇幅，我把不重要的Request Headers都去掉了。

```
GET / HTTP/1.1
....
Host: www.theserverside.com
....
```

请注意，这个HTTP Request里面没有cookie header。
theserverside.com网站接到这个请求，返回了如下的HTTP Response。

```
HTTP/1.1 200 OK
.....
Set-Cookie: JSESSIONID=152468EB452162CB6E8A1170ED0B02F1; Path=/
....

1df3
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
  <head>

    <title> TheServerSide.com: your java Community discussing server side development</title>
....
```

请注意，在Response Headers部分，有一个set-cookie的Response Header。这是因为，网站服务器没有从浏览器的HTTP Request发现cookie，自然也没有发现Session ID。于是，它就分配了一个Session ID，通过set-cookie header传给浏览器。
在同一个浏览器中，我再一次访问这个网站。浏览器这次发出的HTTP Request如下。

```
GET / HTTP/1.1
.....
```

Host: www.theserverside.com

……

Cookie: JSESSIONID=152468EB452162CB6E8A1170ED0B02F1

这一次，浏览器把含有Session ID的cookie发过去了。这一次，网站服务器没有发出set-cookie header，因为它已经知道，浏览器那边已经存放了Session ID。

从这个例子中，我们就可以看到网站服务器的Session ID分配原则。你没有Session ID，我就给你一个。你有了，我就不用给了，就用这一个。

关于Session ID的分配，存在这么一个原则。一个浏览器进程，和一个网站服务器之间，就对应着一个Session ID。

在服务器端，一个网站服务器中，一般都存在很多个Session ID。

一个浏览器进程，访问一个网站服务器，就会产生一个Session ID。

N个浏览器进程，访问同一个网站服务器，就会产生N个Session ID。

在浏览器端，一个浏览器中，也可能存在多个Session ID。

一个浏览器进程，访问一个网站服务器，就会产生一个Session ID。

一个浏览器进程，访问N个网站服务器，就会产生N个Session ID。

下面看具体的例子。

例1：

在这个例子中，一个浏览器进程访问三个网站：a.com，b.com，c.com。

在浏览器端，Session Cookie中，就存在着三个Session ID：a.com网站分配的x001；b.com网站分配的yy001；c.com网站分配的zzz001。

例2：

在这个例子中，三个浏览器进程访问同一个网站：a.com。

在服务器端，就存在着三个Session ID：x001，x002，x003。

例3：

在下面的例子中，有三个浏览器进程，有三个网站。每个浏览器进程都访问那三个网站。总够就有3 * 3 = 9 个Session ID。

关于Session ID，还有一种有趣的场景——在一些HTML中，可能会包括多个Frame（一种HTML元素），每个Frame都有自己对应的网址。那么，这些Frame的Session ID是怎样一种状况呢？

原则是一样的。如果Frame对应的网站地址是一样的，那么，就共享同一个Session ID。不同的网站，就是不同的Session ID。

浏览器端只负责存放Session ID，服务器端不仅负责存放Session ID，还通常维护一个对应Session ID的Session状态存储空间，有时候也简称为服务端Session空间。Session状态存储空间都有些什么内容呢？什么内容都可能有，程序员什么东西都可以往里面塞。这个问题应该换个问法。Session状态存储空间应该放些什么内

容呢

我的建议是，最好什么都不要放，最多只放用户的登录信息。为什么尽量避免在Session状态存储状态空间中放东西？这一点，前面已经讲过了，不再赘述。因为，在集群环境中，Session中的状态，很可能会通过网络，发送到其他的计算机。如果Session状态过重，就会加重系统的负担，影响系统的性能。所以，千万不要养成随时把临时数据扔进Session存储空间的习惯。

所以，一定要牢记这个原则。服务器Session空间中只应该放用户登录信息。那么，用户登录到底是什么意思？又和Session ID有什么关系呢？

Session ID是浏览器身份的识别。而用户名登录则是人的身份的识别。用户名登录是建立在Session ID基础上的。如果服务器端没有给浏览器分配Session ID，那么，用户就不能用这个浏览器登录网站服务器。如果服务器端给浏览器分配了一个Session ID，就说明，服务器端同时也给这个浏览器预留了一个用来存放Session状态的存储空间。这个Session状态存储空间就可以用来存放用户的登录信息。

当用户在网站的登录界面上输入用户名和密码，进行登录的时候，浏览器就会把包含了用户名和密码的HTTP Request发送到服务器端。服务器端收到了用户名和密码，同时还收到了Session ID。

网站服务器在数据中查询用户名和密码，如果匹配，用户就登录成功，网站服务器就会把用户名和对应的权限信息存放到Session ID对应的Session状态存储空间中。

用户的下一次请求过来的时候，网站服务器就会根据用户的Session ID，去找对应的Session状态存储空间，从中取出用户名和对应的权限信息，以便决定用户可以使用的服务内容。

如果说，Session ID只是网站服务器提供的普通服务的话，那么，用户名登录就是网站服务器提供的高级会员服务。

前面讲过，一个浏览器进程，一个网站地址，对应一个Session ID，对应一个服务器端的Session空间。这就是说，一个浏览器进程，只能用一个用户名登录同一个网站。

如果，我们在同一个网站上有多个用户名，我们想同时登录，应该怎么办呢？这就需要使用多个浏览器进程。这就依赖于浏览器的进程模式了。

在真实的世界中，不同的浏览器，有不同的实现方案。

有些浏览器（比如，目前的微软IE浏览器）同时支持多进程和多线程。你直接运行该浏览器多次，就产生多个浏览器进程，每个浏览器都有自己的Session Cookie，互不干涉。

这样，你可以用多个浏览器进程访问同一个网站，并且用多个不同的用户名登录。

当你直接运行一个浏览器，并且从该浏览器的菜单中又启动了一个浏览器的时候，实际上相当于多创建了一个浏览器线程。这两个浏览器线程共享同一个进程空间，因此也共享同一份Session Cookie。他们访问同一个网站，只能用一个用户名登录。

还有的浏览器只支持单进程多线程模型，无论你创建多少个浏览器，都是同一个进程的多个线程。这些浏览器共享同一个进程空间，因此也共享同一份Session Cookie。他们访问同一个网站，只能用一个用户名登录。在同一个网站多个用户名的情况下，这种浏览器就极为不方便。

还有一种浏览器（比如，目前的Google浏览器），支持多进程模型。每一个浏览器都是一个进程。按理来说，这些浏览器都有自己的进程的空间和Session Cookie，因此可以用不同的用户名登录同一个网站，但是，不行。因为这种浏览器的多个进程之间，竟然是共享Session Cookie的。他们只能用一个用户名。

还有些浏览器（比如，目前的Firefox浏览器），可以定义不同的用户空间，虽然稍微麻烦了，但至少可以实现多用户登录。

用户登录是一个十分重要、又十分复杂的主题。本书不可能面面俱到，只能择起要点而阐述之。关于用户登录，无论是作为用户，还是作为开发人员，我们首要关注的问题都是账户安全问题。

我们知道，用户登录信息都是存放在服务器端的Session空间中的，而Session空间是对应Session ID，而Session ID是存放在HTTP Cookie Header里面的。如果HTTP Cookie Header被人截获，Session ID就可能被人截获并仿造，从而冒名登录到网站中。所以，用户登录时及登录之后，一定要采用HTTPS协议，一种基于HTTP的加密协议。这是最基本的保障。

除此之外，我们还要考虑用户的一些粗心大意的情况。比如，在登录网站之后，又开始做别的事情，之后又把登录网站这件事给忘了，开着浏览器就走了。这时候，如果别人过来，使用浏览器的话，就可以直接使用已经登入的账号。

为了避免这个问题，网站引入了Session过期机制。当用户登录之后，长期不操作，网站就会让Session过期（Time Out）。Session过期的时间限制由网站自行定义。对于隐私保护严的账户，比如银行账户，过期时间就短一些，比如，5分钟。对于隐私保护不是那么严的账户，过期时间就会长一些。

当Session ID过期之后，Session ID对应的Session状态存储空间也会过期，里面存放的用户信息也会过期，用户登录状态也会过期。这时候，就需要重新登录。

这就能解决所有问题吗？不能。在Session过期的情况下，如果浏览器仍然开着，不需要知道用户名和密码，一样有办法登录回去。只要回退到之前的登录界面——通常是一个HTML Form界面。由于浏览器通常会缓存HTTP Request的信息，只要刷新一下，忽略浏览器的警告“表单信息已经过期，您确定要重新提交吗？”，就可以重新登录了。

浏览器的这种缓存功能有时候会造成严重的后果。比如，你在一个公共场合，用浏览器访问了一个网站上的私人账户。然后，你退出了登录，然后离开了。但是没有关浏览器。另一个人走过来，把浏览器回退到你之前的登录界面，一刷新，刷，又登录进去了。所以，一定要切记，用完私人账户之后，一定要随手关闭浏览器。

有些安全做得比较好的网站，就考虑到了这一点。它想方设法不让用户退回到之前的登录界面。其手段包括但不限于Refresh（刷新）和Redirect（重定向）。

W3网站上有一段关于Refresh vs Redirect的内容。

http://www.w3.org/QA/Tips/reback

里面是这么讲的。请不要使用Refresh（刷新），因为，Refresh会打断用户的回退按钮。

Refresh是这样一种技术。假设http://www.example.org/foo返回的HTML页面中包含如下的元描述：

<META HTTP-EQUIV=REFRESH CONTENT="1; URL=http://www.example.org/bar">.

那么，过1秒钟，该页面就会自动跳转到http://www.example.org/bar。

当用户想回到前一个页面的时候，刚退回去，一秒之间，页面又跳转了。这样，就成功地打断了回退操作。用户会感到很恼火，一下就把浏览器关了。

为了避免这种情况发生，最好用HTTP Response Redirect，就不会打断浏览器的回退操作。

这里面，能够打断回退操作的主要技术手段是Refresh。但我们可以把Refresh和Redirect组合使用。比如，用户登录之后，网站可以先把用户Redirect到一个包含了Refresh Meta元素的HTML页面，让浏览器再次自动跳到别的页面，然后再Redirect，再Refresh，如此多次，最后显示登录后的页面。这个登录之后的页面URL已经和登录界面的URL毫无关系了。

如果用户想退回到之前的登录界面，就会不断地遭遇到Refresh和Redirect，回退操作被打断得支离破碎，用户就会就会遭遇到极大的挫折感，愤而关闭浏览器，这就达到了保护账户安全的目的。

这是安全机制做得好的网站，大部分网站做得没有这么周到。所以，我们一定要自己注意，在访问私人账户之后，一定要随手关闭浏览器。在关闭浏览器之前，最好点一下退出按钮，强制清空服务器端的Session状态存储空间的用户信息。这样就更保险。

在前面那个打断登录界面和登录后界面的回退操作的用例中，Refresh起到了主要的破坏作用，Redirect只是起了一点辅助作用的帮凶。事实上，Redirect的长处不在于破坏，而在于建设。一些复杂的登录机制，就借用了Redirect的威力。

比如，我们在访问网站资源的时候，经常会遇到这样的情况：网站弹出一个对话框，提示“您访问的资源，需要权限认证，请您先登录”，并且提供了用户名和密码输入框，让用户登录。

等我们输入用户名和密码之后，网站弹出“恭喜您成功登录”，然后，页面就转向之前访问的资源页面。其实现原理也是基于HTTP协议，确切来说，是基于Request Header和Response Header。

Reference这个Request Header的作用是记录用户访问的上一个网址，即用户是从哪里链接过来的。通过Reference这个Request Header，网站服务器就可以知道用户访问的上一个页面是什么，就可以找个地方暂时记录下来，等用户登录成功之后，再通过HTTP Response 把页面Redirect到之前记录下来的网址。

那么，这个网址暂时存放在哪里好呢？最方便的地方，自然是服务端Session存储空间，而且语义上也很直观——该用户访问过的页面，就应该放到该用户的Session存户空间当中。

但是，我们前面讲过了。千万不要养成这个习惯。不过，如果您坚持这个习惯，就当我什么也没说。

不放在Session里面，那应该放在哪里？读者可能会问了。我的建议是，放到一个缓存中。非集群环境下，就放到本机的缓存中。集群环境下，就放到中心缓存服务器中。其效果和放在Session里面是一样的。这种做法虽然多了几步麻烦，但是，也避免了很多潜在麻烦。

需要特别注意的是，上面的场景中，Redirect指令是服务器端发给浏览器的指令，是通过HTTP Response的Redirect Header实现的。这个概念一定要理解清楚。

Reference和Redirect组合使用的应用场景十分广泛，比较复杂的应用场景就是单点登录（Single Sign On）。

单点登录的场景一般是这样的。在一个网络中——有可能是局域网或者互联网，有多个网站服务器。每个服务器都提供不同的服务，比如，博客，新闻，论坛，等。这些网站服务器共享同一份用户登录管理系统和同一个数据库服务器（即共享同一份用户数据）。

由于每个服务器都有各自的Session存储空间，因此，每个网站都需要用户单独登录，而且每个网站都使用相同的用户名和密码登录。

有没有一种方案，能让用户只登录其中一个网站，就可以自动登录其他网站呢？

有。这种方案就是单点登录。

我们首先考虑最简单的场景——主域名相同的多个服务器。比如：

news.a.com

blog.a.com

forum.a.com

这三个网站虽然都有各自的网址，但他们的主域名——a.com，是相同的。这三个网站虽然在同一个局域网内，但他们都有独立的外网IP地址，都是直接暴露在互联网上的。

前面讲过，网站设置cookie的时候，可以把cookie的作用域设置为上级域名。这种情况下，就可以定义一个对应a.com的Cookie，来实现单点登录。当浏览器成功登录到某一个网站的时候，就会收到一个对应a.com的Cookie，里面存放了用户名和密码。下一次，浏览器访问另一个网站的时候，就把a.com对应的Cookie发送到

该网站。该网站往里一看，哦，已经登录过了，还有用户名和密码，那就自动登录一下吧。这就登录了。
这是最简单的情况，还不需要借用Redirect的威力。当主域名不相同的时候，就是Redirect大显身手的时候了。
跨域名的通用单点登录方案就是：Redirect + 中心认证服务器。
什么叫中心认证服务器呢？就是说，在一个单点登录的网站群组中，所有的用户登录都要经过同一个服务器，这个服务器就叫做中心认证服务器。中心认证服务器的作用，相当于多个网站之间共享的服务端Session空间。
下面举一个具体的例子，来说明单点登录的整个流程。
当用户访问单点登录群组中一个网站a.com时，网站a.com先为用户浏览器分配一个Session ID，叫做a001。
然后，网站a.com查看该用户的HTTP Request的Cookie Header里面是否存在中心认证服务器颁发的一个登录凭据（英文叫做ticket）。
如果登录凭据不存在，说明该用户是第一次访问该群组。网站a.com就把用户页面Redirect到中心认证服务器auth.com。
auth.com先从HTTP Request的Reference Header中，把a.com这个网址存下来，然后，显示一个登录页面给用户。
当用户输入用户名和密码，成功登录到中心认证服务器auth.com之后，中心认证服务器生成一个登录凭据，不妨叫做a.com-ticket001。
中心认证服务器相当于一个服务端Session空间，不过是多个网站之间共享的服务端Session空间。a.com-ticket001就相当于一个Session ID，不过是多个网站之间共享的Session ID。
在单个网站的例子中，网站服务器会为每一个Session ID分配一个Session存储空间，用来存放登录信息。同样，中心认证服务器auth.com也会为每一个登录凭证（ticket）开辟一个存储空间，不妨称为Ticket存储空间，用来存放登录信息——用户名和密码。
中心认证服务器auth.com会为a.com-ticket001分配一个Ticket存储空间，叫做a.com-ticket001存储空间，里面可以用来存放用户登录信息，比如用户名和密码。
做完这些之后，中心认证服务器auth.com做两件事情。第一件事是访问a.com，把ticket= a.com-ticket001这个参数发给网站a.com。
这个工作可以通过HTTP协议完成，比如，中心认证服务器auth.com可以访问网址a.com? ticket=a.com-ticket001。网站a.com收到ticket=a.com-ticket001这个参数之后，a.com-ticket001就把存在自己的登录凭据库中。
做完第一件事之后，中心认证服务器auth.com做第二件事，为浏览器准备了HTTP Response。这个HTTP Response是一个Redirect指令，包含了Redirect Header。Redirect的目的网址为：a.com?ticket=ticket001。
另外，中心认证服务器auth.com还在HTTP Response中为用户浏览器准备了set-cookie header。里面放置了信息，ticket=a.com-ticket001。
用户浏览器收到了这个HTTP Response之后，先把HTTP Response里面的set-cookie header中的信息ticket=a.com-ticket001存放到Session Cookie中，生成一个“auth.com-> cookie : ticket = a.com-ticket001”的条目。
然后，用户浏览器根据HTTP Response的Redirct指示，去访问a.com?ticket=ticket001。
这一次，网站a.com在HTTP Request的网址参数中找到了ticket这个参数。网站a.com就知道，该用户浏览器已经在中心认证服务器auth.com登录认证过了。网站a.com会去中心认证服务器auth.com那里核对一下。如果

核对成功，就表示用户确实登录认证了。
网站a.com在服务端的Session ID = a001 的Session存储空间中，存放用户登录认证状态，表示该用户浏览器已经登录过了。然后，网站a.com在HTTP Response的set-cookie header里面也设置一条ticket=a.com-ticket001信息（或者其他形式的登录成功信息），并把受认证保护的页面返回给用户浏览器。
用户浏览器收到页面的同时，也从HTTP Response里面获得了cookie信息。这时候，用户浏览器的Session Cookie中就包括如下信息：
auth.com -> cookie : authSessionID = xxx; ticket = a.com-ticket001。
a.com -> cookie : aSessionID = a001; ticket = a.com-ticket001。
这样，下次用户浏览器在访问a.com和auth.com的时候，a.com和auth.com就可以判断出用户浏览器已经登录认证过了。
注意，用户浏览器cookie中的ticket = a.com-ticket001信息，只是一种表示“已登录状态”的简明表达。在真实的实现中，出于安全等方面的考虑，不一定采用这种表达形式。
当用户浏览器访问单点登录群组中的另一个网站b.com的时候。b.com也会为用户浏览器分配一个bSessionID=b001的Session ID，并且在服务器端分配一个对应b001的Session空间。
b.com没有找到用户浏览器的登录认证状态，同a.com一样，把浏览器页面Redirect到中心认证服务器auth.com。这时候，auth.com从用户浏览器的cookie中发现了如下信息：
cookie : authSessionID = xxx; ticket = a.com-ticket001。
说明，该用户浏览器已经登录过了。于是，auth.com的行为和和前面一样，为b.com生成一个ticket = b.com-ticket001的登录凭据。并通知b.com。然后把用户页面Redirect到 b.com ? ticket=b.com-ticket001。
b.com收到用户浏览器的请求，就知道用户浏览器已经登录认证，核对之后，就set-cookie :
ticket = b.com-ticket001。
浏览器收到b.com的HTTP Response之后，其Session Cookie中的条目如下：
auth.com -> cookie : authSessionID = xxx; ticket = a.com-ticket001; ticket = b.com-ticket001。
a.com -> cookie : aSessionID = a001; ticket = a.com-ticket001。
b.com -> cookie : bSessionID = a001; ticket = b.com-ticket001。
这样，就等于用户浏览器在auth.com、a.com、b.com都登录了。
以上只是一种简化的、示意说法。只是为了大致说明工作原理流程，并不严密。真实的实现方案多种多样，有可能在某些环节更简单，有可能再某些环节更复杂，还需要考虑各种各样的现实问题。不同的单点登录实现方案可能大相径庭，请读者一定要注意这一点。

1.26 《编程机制探析》第二十五章 Web开发架构

发表时间: 2011-10-18 关键字: web, mvc, 框架, rest

《编程机制探析》第二十五章 Web开发架构

前面章节讲述了HTTP协议的方方面面，从本章开始，我们进入到Web编程开发的世界。
Web应用程序这种说法，主要是针对桌面程序来说的。桌面程序的图形界面元素十分丰富，交互性、操作性也十分良好。Web应用程序的界面，传统来说，只有一种，就是在浏览器中显示的HTML。
一开始的时候，HTML并不是为了应用程序而设计的图形界面，而是以内容文本表现为主要目的的文本结构。HTML具备了最基本的图形界面元素，但是，从功能、效果、交互性、可操作性来说，都比桌面程序简陋了许多。Web应用拥有与生俱来的优势，Web应用是基于HTTP协议的，而HTTP协议是一次性应答协议，不保持长连接，这就使得Web应用可以支持大用户量和访问量。由于这个优势， Web应用程序蓬勃发展，四面开花，HTML开始承担起应用程序图形界面的重任。
在Web应用程序的发展历程中，HTML的局限也越来越明显。为了提高HTML的表现能力和互操作能力，HTML与时俱进，不断引入新的特性，其中一个重要举措是植入各种插件，提供更加丰富的界面功能。同时，HTML规范本身也在修订中，现在HTML5方兴未艾，提供了很多以往由插件提供的媒体功能。现代的Web应用程序界面，比起从前，不知强了多少倍。当然，比起桌面程序来，还是有不少的差距。但对于大多数的应用来说，却是足够了。
Web应用程序的应答模式很简单，就是一问一答：浏览器发出HTTP Request，网站服务器收到HTTP Request，生成HTTP Response，返回到浏览器。
从前面章节已经给出过例子，HTTP Response除了头部几行之外，剩下的部分全都是HTML。那么，Web应用程序的主要任务其实就是生成HTML页面。相应的，页面生成技术，也是Web应用开发中的一个重要主题。关于这个主题，后面有专门的章节讲解。本章主要关注Web开发的总体架构。
Web应用程序的HTML界面有两个主要特点。
首先，Web应用是“应用程序”，HTML中的Form（表单）和Button（按钮）元素，使得Web应用程序具有了最基本的用户响应功能。
其次，Web应用程序也是内容提供服务程序。HTML本身是一种树形结构化的文档定义。Web应用程序的HTML界面，很大程度上也是一种网页文档。
从这两方面来看，Web应用程序很像是桌面程序中的一个类别——多文档编辑程序。当然，Web应用程序的文档编辑功能很弱，只能通过提交动态数据来改变网站上的HTML页面。
Web应用程序和多文档编辑程序更相似的地方还在于程序组织结构上，两者的结构都是文档数据 + 文档展示界面 + 流程控制，即我们熟知的Model-View-Control（简称MVC）结构。其中，Model对应文档数据，View对应文档视图，Control对应文档视图控制。
多文档编辑程序的MVC结构一目了然，我们以办公软件中的多文档编辑程序（比如Word）为例。Model就是文档文件，View就是文档视图，Control就是那些和文档视图切换相关的功能菜单。
Model和View之间是多对多的关系。一个文档文件可以用多种视图来展示，一种视图可以用来展示各个文档文件。这体现了MVC架构在代码重用方面的优势。

相对于多文档编辑程序的简单清晰的MVC结构来说，Web应用程序的MVC结构就复杂了许多。事实上，Web应用程序只是借鉴了多文档编辑程序的“MVC架构”这个名词，在具体的应用上，Web应用程序对MVC架构进行了重大的扩充和修改，赋予了MVC架构丰富的新内涵，最后夺取了MVC架构定义的话语权。现在，各种资料中提到的MVC，主要就是指Web MVC。

前面的章节中讲过，网站服务器分为CGI和app server两种模式。这两种模式的区别在于，app server模式下提供的内存共享模型更加丰富一些，响应用户请求的线程之间可以共享app server进程空间内的同一块内存，在Session实现、数据共享方面多了一些选择。除此之外，两者的开发架构并没有显著区别。无论是CGI模式，还是app server模式，都可以应用MVC开发架构。

需要提醒的时候，Web MVC架构并不是一个没有争议的规范定义。由于Web MVC架构比多文档MVC复杂了许多，Web MVC存在着各种形态，其对应的Model-View-Control也有不同的含义。本书采取的Web MVC定义只是目前最流行的一种，也是我觉得结构划分最清晰的一种。

在Web MVC中，各种具体对应物有了新的变化。Model不再是指文档文件，而是指动态数据，进而指提供动态数据的对应程序。而View也不再是指视图，而是指嵌入了页面展示逻辑的HTML模板，进而指提供页面展示逻辑的对应程序。Control不再是文档视图切换，而是包含了服务程序分派、页面流程控制等复杂功能。Model和View，后面有专门章节讲解。本章重点讲解Control部分，因为Control是整个Web应用程序的入口点。

Control有两个主要任务。第一个任务是“URL到服务程序的映射”，这个任务也叫做Dispatch（分派，即根据URL将请求分派到对应的服务程序），实现这个功能的模块通常叫做Dispatcher（分派器）。第二个任务是“页面流程控制”，这个任务的内容比较丰富，包括数据流向、模板选择、页面转向等等诸多内容。

我们先来看第一个任务——URL到服务程序的映射。首先，我们需要了解URL的含义。URL是Universal Resource Locator的缩写，这个词汇的本义，我们不去管它，我们只关心URL在具体应用中的含义。

我们知道，用户通过URL来访问网站上的对应服务。URL的最初含义是网址，即网站上的资源文件地址。这是静态网页时代的概念。在静态页面网站上，URL对应网站上的某个具体资源文件路径。在动态网站上，URL对应着某个提供动态内容服务的程序。这正是Control（或者说Control中的Dispatcher模块）要做的第一个任务——把URL映射到对应的服务程序上去。

“URL映射到服务程序”的实现方案分为两种——静态配置和动态解析。

静态配置，很容易理解。就是把映射规则一条条写到代码里面，比如：

```
case URL of
    “login” -> loginService(request, response)
    “regisiter” -> regisiterService(request, response)
   ....
```

网站的开发并不是一蹴而就的。有可能先开发出基本功能，上线测试，再逐步提供更完善的功能。在这个过程中，URL映射规则可能会改变或者扩充。

如果上述映射代码是动态类型解释语言的话，那么，没问题，不需要停机重新编译，直接修改代码中的URL映射规则就可以了。

如果上述映射代码是静态类型编译语言的话，那么事情就麻烦了。代码修改之后，网站服务器需要停止，代码需要重新编译和部署，之后，网站服务器再重启恢复服务。为了避免这种麻烦，程序员通常把URL映射部分移出

到代码之外的配置文件中（通常是XML格式），这就解决了代码重新编译和部署的问题。

静态配置的优点在于一目了然，有多少URL映射，有多少服务程序，写得清清楚楚。但静态配置有个根深蒂固的弱点，那就是无法自动扩充。每增加一个新的服务程序，就需要添加一条新的映射配置。动态解析方案，则解决了这个问题。

动态解析方案中，程序员不需要写URL配置文件，但是，需要定义一套简单明了的映射解析规则，然后，按照这个规则，解析URL，直接把URL映射到对应的服务程序上去。比如，上述的映射代码可以写成：

call（URL + “Service”，[request, response]）

意思就是，把对应的URL部分取出来，加上“Service”这个字符串，就构成了一个函数名，然后，把[request, response]作为参数，调用这个函数。这就实现了把URL映射到服务程序的功能。

上述的URL解析规则只是一个示例。不过，真实的解析规则也比这个规则复杂不了多少。因为，动态解析的要点就是解析规则简洁明了，否则就失去了实用的价值。

静态配置方案曾经是主流，但如今，越来越多的Web开发框架开始采用动态解析方案。关于URL映射，有一个流行的说法，叫做“Convention Over Configuration”（惯例优于配置）。这里的Convention（惯例），就是指简单明了的URL解析规则。“Convention Over Configuration”（惯例优于配置）的意思就是，动态解析方案优于静态配置方案。

讲到URL映射，就不得不提到“URL美化”这个技术。

如果不做任何修饰美化的话，一个网站的原始URL看起来是这个样子的：

http://somesite.com/inbox.php?a=1

这样的URL比较难看，一堆的?、&、=等杂七杂八的符号，看起来很不清爽。这个URL可以美化成：

http://somesite.com/inbox/a/1.htm

这个美化工作实现并不难，只需要在网站服务器前端加一个URL字符串处理程序，就可以把类似于inbox/a/1.htm这样的URL字符串转化成inbox.php?a=1，这就得到了可以被Web应用程序正确处理的URL。

这里要注意一点，URL处理程序并不是对难看的原始URL进行美化，而是对美化后的URL进行还原。因为，我们在网页中编写的URL都是经过美化的URL，用户访问的也是美化后的URL。

URL还原处理的功能，可以放在外置的URL处理程序中实现，也可以在Control中实现。如果在Control中实现的话，就可以直接在“URL动态解析”程序中实现，直接把美化后的URL解析为服务程序需要的参数，从而省略“把美化的URL还原”这个步骤。

URL美化之后的样式多种多样。有些人喜欢把URL美化成类似于静态网页资源的样式。比如前面的例子：

http://somesite.com/inbox/a/1.htm

有些人喜欢把URL美化成REST风格的样式（RESTful），比如：

http://somesite.com/inbox/a/1

这种REST风格的URL在目前颇为流行。那么，什么叫做REST呢？这就说来话长了。不过，鉴于REST这个词汇如此流行，即使说来话长，也得说一说。

REST并不是“休息”的意思，而是“Representational State Transfer”的缩写。这个词组的原意比较怪异，直译过来也是怪怪的。这里不去追究其本意，只介绍和我们相关的内容。

REST这个概念，是在Web Service刚刚兴起的年代提出的。那时候，Web Service完全是XML RPC的天下，主流协议是SOAP（Simple Object Access Protocol，这个名词也没什好解释的，知道有这个东西就行了），那是一个用XML来定义远程函数调用接口的规范，包括消息头、消息包装、调用服务名、参数名、参数类型、参

数值等等诸多信息。
正是在这个大环境下，一个博士生在毕业论文提出了REST概念，矛头直指SOAP。我现在还记得当年那篇文章的大意。
文中抨击了SOAP的弱点，说SOAP相当于把服务名等信息隐藏在邮包的内部，每一个接受到SOAP的经手人（服务器）都得把邮包拆开（即深入到SOAP格式的消息体内部），才能得知需要的服务名。
文中提出了另外一个思路——REST风格。在REST方案中，服务名和参数不再包装在某种协议（如SOAP）内部，而是直接体现在URL上。
比如，getUser(id = 1001)这样的远程调用不再写成这样的格式：

```
<method=”getUser”>
  <param name=”id”>1001<param>
</method>
```

而是直接写成user/1001这样的URL形式。这样，服务器一看到这个URL，立刻就知道这个请求是要求什么服务。
在这种样式下，user/1001就类似于一个查询条件（user=1001）。1001可以看作是user这个资源库中的一个ID。这个字串看起来就像是一次数据查询——“请给我ID为1001的user”。
在REST方案中，URL不再叫做URL（Universal Resource Locator），而是叫做URI（Universal Resource Identifier）
REST样式的Web Service有很多优点，如清晰易懂，对于代理服务器（Proxy）友好，等等。
W3网站上有两个XML查询相关的规范——XPath和XQuery。
XQuery结构复杂，能力强大，能够与关系数据库的SQL相媲美，不适合用于URI中。
XPath就是一行，用来定位XML元素的路径、属性和值。这简直是为REST风格量身定做的协议。
我个人倾向于REST风格，一方面因为REST简洁明了，另一方面因为我不喜欢用XML格式来描述函数调用。如果是数据资源查询导向的服务类型，那么首选REST。如果是参数复杂的远程调用的服务类型，可以尝试改造成REST样式，如果实在难以改造，也不必强求。
关于URL映射的内容，就讲到这里，我们继续讲解Control的第二个重要任务——页面流程控制。这个任务的内容比较繁杂，我们一步一步来讲解。
HTTP Request中有一个Header，叫做Method。这是一个非常重要的属性，定义了HTTP Request对网站资源的操作。最常见的Method取值有两种：Get和Post。
当我们通过网址访问网站的时候——比如，我们在浏览器中输入网址，或者点击网页中的网址链接——HTTP Request的Method就是Get。
当我们通过HTML Form（表单）中的输入框和按钮向网站服务器提交数据的时候，HTTP Request的Method通常就是Post。为什么要说“通常”呢？因为，HTML Form有一个属性叫做Method，对应着HTTP Request的Method。
如果我们把Form的Method属性定义为Get，那么，HTTP Request就和网址访问一样，Form输入框中的数据参数就会跟在URL的后面，样式如 ?a=1&b=2，HTTP Request的消息主题部分就是空的。
如果我们把Form的Method属性定义为Post（缺省值就是Post），那么，Form输入框中的数据参数就不会跟在URL的后面，而是出现在HTTP Request的消息主体中。
URL参数的容量是有限的。如果我们想提交大量数据到网站服务器的话——比如，发表文章——我们就应该使

用Post。当然，Post也是有数据容量上限的，不过这个上限要比Get高很多。
大部分情况下，HTML Form的Method都是Post。为了便于后面的讨论，我们就把HTML Form提交等同于Post，把网址访问等同于Get。
服务器端对于Get和Post的处理流程大同小异，只除了一点：Post流程处理中多了一步参数值验证（Validation）。
在出现HTML Form、需要用户输入数据的户界面中（如注册、登录、查询、输入数据等），服务器端都要对用户输入的数据进行验证。进行数据验证（Validation）的程序叫做Validator（验证器）。
成熟的Web开发框架一般都会提供一些现成的常见验证器。比如，长度验证器（输入框中的数据长度必须在某个范围之内），数字验证器（输入框中的所有字符都必须是数字），等等。这些验证器还可以组合起来使用，要求输入框中的数据同时符合几个验证器的要求。这是一个典型的Compositor Pattern。
根据代码运行位置的不同，验证器可以分为两类——浏览器网页中的Javascript验证器和服务器端验证器。
一般来说，上述的常见验证器都会同时实现两套方案，一套是浏览器网页中运行的Javascript验证器，一套是服务器端的验证器。这是为了提供双重保险。
用户输入数据首先要经过浏览器网页中运行的Javascript验证器的验证，如果不通过，Javascript验证器直接给出用户提示，要求重新输入。这种做法的好处是不需要重新连接服务器，节省时间和网络流量。
有时候，用户浏览器不支持Javascript或者支持得不好，Javascript验证器没有起作用，数据没有经过验证，就发到了服务器端，这时候，服务器端的验证器就可以发挥作用了。这就是双保险。即使浏览器中验证过一遍，服务器端再验证一遍也无妨，这点开销相对于网络通信和数据查询来说，几乎是可以忽略不计的。
除了Web框架中提供的现成验证器之外，程序员还可以自定义验证器。同样，可以自定义浏览器Javascript验证器和服务端验证器。有些验证器涉及到服务器端数据，必须在服务端实现。比如，用户名存在验证器，密码错误验证器，等等。
除了数据验证这个步骤之外，Post和Get的处理流程就基本一致了，都是根据Request参数获取数据（Model）和页面模板（View），然后，生成HTML页面，写入到Response中。
如何获取数据（Model），如何获取页面模板（View），如何生成HTML页面，这是Request处理流程的三大关键步骤。所有的Web开发框架都是在这三大关键步骤上作文章。不同的Web框架，对这三大关键步骤的实现和组织也不一样。
下面，我用函数式伪代码来描述一种我最欣赏的MVC架构。为什么要用函数式呢？因为函数式语言中，函数本身就是对象，不需要另外的对象来包装，表达起来简单清晰，更加时候用来描述MVC流程。这种MVC架构以结构清晰取胜。函数式伪代码描述如下：

```
mvc ( request, response) =
url = request.url
service = dispatch( url )
# 以上部分是Dispatcher部分，把URL映射到具体的服务函数

 (model, view) = service(request, response)
 # 这是最关键的一步。整个框架核心就体现在这里
 # 服务函数返回model和view。
# 这里的model是所有需要显示的数据。通常是树形结构。
```

```
# 这里的view只是一个字符串，代表页面模板的名字

pageTemplate = findTemple( view )
 # 根据view获取真正的页面模板

 write(model, pageTemplate, response)
 # 把model和pageTemplate结合起来，写入到response里面
```

上述的伪代码就是一个MVC框架的简单示意代码。框架用户只需要实现对应的service函数，比如，loginService(request, response)，registerService(request, response)。框架负责找到并调用这些service函数，完成整个请求。

在上述代码中，有一条语句：pageTemplate = findTemple(view)

这条语句根据View（一个简短的字符串名称）找到对应的HTML页面模板。同URL映射到服务程序一样，这也是一个映射工作，即，把View字符串名称映射到对应的HTML页面模板文件对象。

同“URL映射服务程序”类似，“View映射到页面模板对象”的实现方案也分为两种——静态配置和动态解析。

静态配置很简单，就是类似于

```
case View of
   “success” -> “successed.html”
   “fail” -> “login.html”
  ....
```

这样的代码。我们可以把这段代码用XML形式表示，就成了配置文件。这个配置文件我们可以单独放置，也可以和URL映射配置文件合成同一个文件，大多数MVC框架正是这么做的。

动态解析，同样也很简单。我们可以指定简单的规则，比如给View字符串名称加上前缀和后缀，就可以构成目标文件名，等等。

一开始进行Web开发的时候，我都是用的现成的Web框架。在使用的过程中，总是感到有些地方不满意，于是，就开始研究其他的Web框架，看看是否有更好的方案。在找到更好的Web框架的同时，我又会发现新的不足之处。到了最后，我就萌发了自己设计Web框架的想法。我相信，很多人和我有一样的经历。

本章中讲述的内容，是我在设计和开发Web框架的过程中总结出来的。希望能对大家有所帮助。在结束本章之前，我阐述一下Web框架设计的两个原则：灵活和完熟。

同时具备了灵活和完熟特点的Web框架，是可大可小的框架，能够给用户带来最大的自由和方便。

灵活的意思就是，设计框架的时候，尽量灵活，框架中每一个部分都是可插拔、可拆卸、可替换的。

比如，“URL映射服务程序”部分，完全暴露出来。用户可以使用动态解析方案，也可以使用静态配置方案，还可以编写自己的方案，替换掉框架中的现有方案。“View映射到页面模板对象”部分，也照此处理。

再比如，数据验证器（Validator）部分也完全暴露出来，不仅允许用户自定义验证器，还允许用户自定义验证器的工作流程。由于验证器的逻辑和页面流程经常混杂在一起，这个地方比较复杂，需要精心设计。

又比如，动态页面生成部分，也完全暴露出来，允许用户采用各种各样的方案，最大限度提高生产率或者运行效率。

其他部分也都可以照此办理。这种设计思路的好处是灵活度高、可定制程度高、重用度高。

由于每一个部分都可以单独使用，也可以和其他部分组合使用，用户可以按照自己的需求进行定制，每个模块都可以达到最大的利用率。

但是，这种设计思路的缺陷也是很明显的。这样设计出来的框架必然是千疮百孔，处处可插拔，意味着处处是需要填补的孔洞。

如何弥补这个缺陷呢？那就是第二个原则——完熟。

框架开发者必须提供一套或者几套完熟的成例方案，对应常见的Web应用。这个工作可能很低级，就是一个排列组合工作。也可能很繁琐，因为选项太多，组合起来的各种可能性就更多。程序库管理更是一个噩梦。你需要提供各式各样的程序打包，组合形式有多少，程序包的种类就有多少。总之，这个工作比起开发框架模块来，显得不那么有吸引力。但是，这个工作是必须做的。

1.27 《编程机制探析》第二十六章 页面生成技术

发表时间: 2011-10-18

《编程机制探析》第二十六章 页面生成技术

Web应用程序之所以如此流行，有两个主要原因。第一个原因是界面的一致性，即浏览器内显示的HTML；第二个原因是能够支持巨大的用户访问量。

Web应用程序之所以能够支持巨大的用户访问量，主要是因为HTTP协议的无状态特性。随着技术的发展和应用的成熟，Web应用程序对用户状态的要求越来越高。HTTP协议的无状态特性就成为了一个难以绕过的阻碍。这真是，成也萧何，败也萧何。

为了克服HTTP协议的无状态特性，HTTP协议本身引入了Session的概念，但是，Session只是一个基础设施，远远不够，Web应用还得自己做大量的工作来保持用户状态。

Web应用保持用户状态的地方有两个——服务端和浏览器客户端。

如果是保存在服务端，Web应用需要为每一个用户（浏览器进程）开辟一块专用空间，或者在服务器Session空间中，或者是在app server进程的共享内存中，或者存放在另一个进程的共享内存中（如数据库或者网络中心缓存）。在这些空间中，服务器需要存储用户当前状态和当前步骤。

用户每一次请求过来，服务器就按照上次存储的状态和步骤继续运行。请求完毕之后，服务器再把当前状态和步骤存储起来，等待用户的下一步请求。

这种工作模式很像是Continuation（连续），一种时停时续的工作方式，整个工作过程看起来就像是“运行-暂停-继续运行-暂停-继续运行……”的样式。

我们可以用线程同步等待的流程来理解Continuation。当一个线程遇到同步锁需要等待的时候，线程调度程序就会把这个线程挂到该同步锁的等待队列中，那个线程同时保持着当前的运行状态，以便获取同步锁之后继续运行。

事实上，服务端状态保持的实现方案之一就是Continuation。这种方案的好处在于，编程模型自然，工作流程清晰，程序员不需要单独为用户的每一步请求写一个服务应答程序，而是可以把用户的所有操作步骤（多次请求访问）都写在同一个过程中，仿佛这些步骤就是一个完整流程的几个普通过程调用而已。

Continuation方案的坏处也很明显，那就是保留了太多的状态，增加了服务器的负担，降低了服务器的并发性能和吞吐量。要知道，状态是并发的天然敌人。

除了Continuation之外，还有一些服务端状态技术，其名字一般都和“flow”（流程）沾边，如Page Flow，Web Flow，等等。

同Continuation一样，这些“Flow”的实现也颇为繁琐，既需要为每一个页面定义一个步骤ID，也需要在服务器端存储每一个用户的当前状态和步骤。

无论是Continuation，还是Flow，它们的共有问题就是服务端状态太重，影响服务器的并发性能和吞吐量。那么，状态不存放在服务端，又能存放在哪里呢？一些开发人员把目光转向了浏览器客户端。

为什么需要把用户的操作步骤分成好几个页面？放在同一个页面中完成不成吗？当然成，用Javascript就可以。随着技术的发展，Javascript的功能越来越强大。浏览器中的Javascript可以直接向服务器发出HTTP请求，获取动态数据，并更新当前的HTML页面。在这种方案中，当前页面一直没有换，网址也没有换，所有的状态也都没

有丢，都存放在浏览器的当前页面中。服务器不需要再保留大量的状态，轻装上阵，无状态就无负担，精神好，牙口就好，吃嘛嘛香，胃口倍儿棒，吞吐量大，并发性好。

当然，浏览器端状态也不是没有问题的。那就是刷新。如果用户不小心按了刷新按钮，浏览器就会重新发出请求，获得崭新的页面，当前页面中的状态就会烟消云散了。不过，这不是什么大问题，而且也不难解决，比如，Javascript可以定时把用户输入的重要信息暂存到服务器端。

从目前的情况看，浏览器端状态已经逐步压倒了服务器端状态，成为了当前的主流方案，同时也给Web应用开发模式带来了巨大的改变，省去了服务端的大量工作。服务器端不需要再保持大量的用户状态，同时，也不再需要生成大量的动态页面。这是怎么说呢？因为，在客户端状态方案中，一般只有一个页面。而且，这个页面的内容更新都是由Javascript完成的，不需要服务端操太多的心。就这样，服务端瘦了下来，客户端胖了起来，富了起来。

在这种情况下，再来讨论服务端页面生成技术，似乎没有太大必要了。但我还是想把自己在页面生成技术方面的一些心得和大家分享一下。因为，页面生成技术本质上就是字符串拼装技术。而字符串拼装技术是应用很广的技术，不仅用于HTML页面生成中，还可以用于各种文本生成中，比如，代码生成，SQL生成，页面缓存生成，等等。这些生成技术的原理是相通的，一通百通。

下面我们就来看本章的主题——页面生成技术。

首先，让我们回到MVC架构之前的混沌年代。那时候，一段典型的Web应答代码是这样的：

```
process( request, response) {
// request 是 HTTP Request 对象，response 是 HTTP Response 对象

html = makeHTML(request) // 根据request生成HTML

response.write(html) // 将生成的HTML写入到response中
}
```

上述的代码只是一种理想模式，真正的代码要散乱得多。response对象是HTTP Server提供的对象，其内部对应着一个HTTP协议层的网络数据缓冲池。HTTP Server内部的网络协议处理程序需负责将HTTP数据缓冲池的数据分块打包，传给下层的TCP/IP层。从理论上来说，HTTP数据缓冲池中的数据，越早准备好就越好，这样就给了网络协议处理程序更多的时间来分块打包。因此，一般的Web应答代码看起来是这样子的。

```
process( request, response) {
// request 是 HTTP Request 对象，response 是 HTTP Response 对象

  parameters = request.getParameters()
// 从request中获取HTTP Request中的URL地址或者消息体中的参数

data = fetchData(parameters)  // 根据参数获取动态数据

response.write( “一些固定的静态HTML片段” )

html1 = makeHTML1( data)  // 根据动态数据生成一段动态的HTML片段
```

response.write(html1) // 将生成的动态HTML片段写入到 response中

// 上述过程不断重复。根据动态数据，生成各种动态HTML片段。
// 按照正确的顺序，将静态HTML片段和动态HTML片段依次写入到response中。
// 代码中到处都是response.write(“html 片段”) 这样的语句
}

可以想见，上述的代码中，必然充斥着大量的HTML片段字符串。这样的代码无疑是丑陋的。如果HTML很长的话——事实上，界面内容越来越丰富，成百上千行的HTML很常见——那么，生成HTML的代码将丑陋得令人发指。

这种HTML大量分布在代码中的情形，叫做污染——即HTML污染了代码，也叫做侵入——即HTML侵入了代码。

在一个HTML页面中，静态部分总是占大多数的，动态部分总是占小部分的。为了小部分的动态内容，把大部分的静态内容嵌入到代码中，这种做法显然是本末倒置、得不偿失的。那么，如何来解决这个问题呢？

这时候，一种直观的解决方案出现了。既然把大部分HTML嵌入到小部分代码中是很难看的，那么，换个思路，把小部分代码嵌入到HTML中不就得了？

事实上，目前的主流动态页面生成技术——如PHP、ASP、JSP、Python页面模板、Ruby页面模板等——正是采用这样的方法，在庞大的HTML文本中嵌入动态代码。

页面生成技术，就是在这个地方，误入了歧途。这种“HTML中混入代码”技术虽然是最流行的页面生成技术，但绝非是最好的技术。由于混在HTML中的代码难以管理，难以重构。这也是一种污染和侵入，代码污染了HTML，代码侵入了HTML。

“HTML中混入代码”技术带来了很多负面效应，令页面程序员深陷泥沼不可自拔。但人们并没有反思这种技术是否存在根子上的问题，而是沿着这条道路上走得越来越远，想出各种方法对这种技术进行修修补补，企图弥补其缺陷，最终的结果是，不仅没有解决原来的问题，反而引入更多的问题，从而催生了一个新的软件市场——页面技术修补技术。这并不是为了解决用户的问题，而是软件开发领域里自己制造问题，自己解决问题。这是典型的自产自销，不可避免地增加了用户的最终成本，同时也养活了一大批软件从业人员。

在软件开发领域，这种看似怪诞的事情，实则极为常见，尤其在那些被超级大公司控制的领域中。那些大公司有意地推行一些极为笨重、笨重的开发框架，从而增加开发成本和时间，来养活更多的软件从业人员。对于软件开发人员来说，这些大公司功不可没，正是因为大公司的这些做法，才保证了人才市场对软件开发人员的需求。这种现象不仅在软件领域存在，在各个领域中都存在。任何领域中，具体工作都是在基层完成的，越到上层，工作内容就越抽象。到了超级顶层，基本就剩下“吹水”的工作了，也就是说，要靠人格魅力取胜，而不是靠办事能力。到了那个层次，做人，远远比做事重要。个人觉得，那才是个人价值的真正体现。

MVC架构的出现，一定程度上减少了“HTML中混入代码”技术的负面效应。因为获取数据和页面流程的代码最大限度地移出，页面模板中只剩下尽量少的必要的页面逻辑代码。

但是，就是这些残留在页面中的代码，也给Web开发带来了很大的麻烦和困扰。在复杂的HTML模板中，一切可以应用在纯代码中的重构技术都失效了。一条if else或者for 语句有可能跨越几十行、甚至上百行HTML文本。而且，HTML文本中的代码只是一个过程中的代码片段，很难结构化。在一些特殊的情况下，显示逻辑代码可能需要用到递归——比如，展示树形结构数据的时候——这时候，HTML中的显示逻辑代码就力不从心了。

令我想不通的是，除了这些难以克服的本质问题，页面技术还在不断引入新的问题。比如，很多的主流页面模

板都采用<% %>这样的百分比尖括号来包装代码。这种样式会直接破坏HTML的结构，使得浏览器无法正确HTML模板。

那么，这个问题是如何解决的呢？

有些人采用<!-- --> 这样的XML注释尖括号来包装代码，这就有效地减少了对HTML结构的破坏。但遗憾的是，采用这种方式的页面模板并不是主流技术，至少不是那些大公司支持的主流技术。

那么，掌握了技术主旋律的大公司是如何做的呢？他们的思路可谓是另辟蹊径，别出心裁。他们同样也认为<% %>这样的代码包装尖括号很难看，但是，他们不认为这是代码的错，他们认为这是格式的错。他们认为，HTML是XML格式，<% %>这样的代码包装尖括号不是XML格式，所以，才把HTML的格式破坏了，页面模板才显得很难看。不得不说，他们的想法也确实有一定的道理。

那么，他们是如何解决这个所谓的“格式问题”的呢？他们提出了“页面组件”的概念，这个概念借鉴了桌面程序开发中的“窗口组件、控件”的概念，应用到HTML页面中。首先，为了表达代码逻辑，他们定义了一套“逻辑代码”组件，即把if else for 等诸多逻辑代码变成XML格式的表达。其次，为了处理HTML元素中的动态显示部分，他们把几乎所有的HTML元素都给重新定义了一遍，定义成了另外一套“界面控件”组件，这套“界面控件”几乎就是HTML控件元素的翻版。

有了这样的“利器”，整个HTML模板就可以重写了。就这样，“页面组件”代替了原有的一部分动态HTML内容，静态HTML还是保持不变，整个HTML模板全都变成了XML格式。好吧，我承认，XML格式化这个目的确实达到了，虽然我看不出XML格式化的目的到底何在。那么，“页面组件”是如何实现的呢？“页面组件”是一套“全新”的XML格式，它最终还是要输出成为HTML格式。它是如何输出的呢？答案是，用代码来输出。

为了支持“页面组件”的HTML输出，每个“页面组件”都对应着一个后台组件程序。这些后台组件程序的作用就是根据“页面组件”的定义，输出对应的HTML。不可避免的，这些后台组件程序中，必然充斥着HTML元素字符串。我们可以看到，在开头我们讲的“HTML污染代码”的问题，又回来了。这完全是走了回头路。

那么，“页面组件”避免了“代码污染HTML”的问题吗？初看起来是这样的，都是XML格式，看起来挺整齐的。但实质上是没有。HTML中仍然存在着逻辑代码，只不过这些逻辑代码变成了XML格式。

“页面组件”完成之后，开发人员才“如梦初醒”地意识到另一个问题——可视化问题。“页面组件”并不是合法的HTML元素，也不能在浏览器中正确显示。在可视化方面，“页面组件”比起“<% %>”样式，没有任何的进步，反而变本加厉地破坏了HTML的显示结构。别急，这时候，“页面组件”的XML格式的优势就显现了出来。开发人员又开发出一套“页面组件”渲染系统，其工作原理很简单，就是解析XML格式的页面模板，遇到静态HTML就直接输出，遇到页面组件，就调用后台组件程序，输出HTML。这样得到的结果就是纯粹的HTML，就可以利用HTML渲染器来正确显示了。于是乎，“页面组件”概念的提出，又催生了两个领域的产业，一个是页面组件开发领域，一个是页面组件渲染显示领域。

在我看来，页面组件几乎没有任何优点，放眼望去，几乎全是缺点。页面组件破坏了HTML的可视化，其罪一；页面组件用XML格式表达代码，其罪二；页面组件用代码污染HTML，其罪三；页面组件用HTML污染代码，其罪四；页面组件的渲染效率很低，降低了服务响应速度，其罪五.....

页面组件的唯一优点可能就是自产自销，又创造了一大批工作岗位，又养活了一大批人吧。但养活的这批人中不包括我，因为，页面组件严重违反了我的技术审美观。

在我看来，页面技术要想一劳永逸地免除麻烦，只有一个方法，那就是从根子上着手，彻底清除页面模板中的任何代码，使得页面模板成为不含有任何可执行代码逻辑的纯粹资源文本。当然，为了显示动态数据，页面模板中还是需要保留必要的层次结构信息，以便和动态数据模型（通常也是树形结构）相对应。我这种方案称为

“层次结构化文档”。

这个名称听起来是否有些熟悉？没错，HTML本身就是一种XML格式，而XML天生就是树形的层次结构化文档。那么，“层次结构化文档”的一种最简单实现，就是直接在HTML的XML DOM结构上做文章，即取出对应的XML结点，进行替换（条件分支）或者重复添加（循环）。这种方案的好处是简单易行，不需要添加任何基础设施，只需要利用现成的XML解析器就可以了。但这种方案的缺点也不容忽视，HTML通常比较复杂，层次结构比较深，元素又多又琐碎，其XML DOM结构相当笨重庞大，无论是操作，还是显示，空间和时间上的开销都比较大，会影响到性能。

一般来说，动态数据模型的层次最多也就四五层，再多也多不到哪里去。而HTML层次结构动辄十几层、几十层，其静态元素个数也远远超过（几十倍上百倍的超过）动态数据模型的数据量。直接使用XML DOM结构是不太合适的。

一个折中方案是“自定义层次结构”，即，用另外的标记（Mark）来划分文档结构。比如，我们可以用如下的 being ... end 样式来划分文档结构。

```
<html>
....
<!-- begin a -->
 ....
 <!-- begin b -->
 ....
 ....${var}....
 ....
 <!-- end b -->
 ....
<!-- end a -->
...
</html>
```

这种划分标记十分简单，解析器也很容易实现，至少比XML解析器和代码语法解析器简单太多了。上述的自定义标记把文档划分成两三层，最多不超过十个结点。如果用XML DOM来表达的话，十几层、几十层都可能有的，至于结点，那就更多了，成百上千都有可能。

有了层次结构化文档之后，又该如何使用它呢？有两种方案，第一种方案叫做“分块取用”，第二种方案叫做“层次匹配”。

“分块取用”和XML DOM操作类似，就是把层次结构化文档当做一棵文档树来使用，想显示哪个结点，就显示哪个结点，想显示多少次，就显示多少次。

这种做法的好处是，简单、直观、灵活、强大。程序员可以任意操作这些文本结点到达任何目的，比如，递归显示树形数据，页面布局插入其他页面的结点，等等。要知道，在“HTML中混入代码”方案中，这些功能的实现是相当麻烦和困难的。

这种做法虽然好处多多，但有一个鲜明的缺点，那就是页面逻辑代码和特定页面技术绑定得太紧。那些操作文本结点的后台代码就是页面逻辑，这些代码需要调用具体的页面技术API，这就意味着Web应用代码需要在代码

中引入这个具体的页面技术开发包，从而造成依赖。这也是一种侵入和污染。一个设计良好的Web开发框架，是不应该允许这种情况出现的。而且，那些操作文本结点的代码都需要调用具体的API，由于这些代码需要取用并操作的文本结点，通常都比较繁琐。而且，这些代码是无法重用的。如果换一种页面技术，这些代码只能弃而不用。

更好的选择是第二种方案——“层次匹配”。在这种方案中，后台页面逻辑代码并不直接操作文本结点，而是根据文本层次结构，对动态数据模型进行包装，生成一个用来“匹配显示”的“页面数据模型”，供层次匹配引擎程序使用。匹配引擎程序把“页面数据模型”和“层次结构化文档”匹配起来，直接输出结果HTML。

这个匹配引擎程序的实现非常简单，比脚本解释器的实现简单太多。在页面显示中，动态内容的显示只有三种情况：不显示，显示一次，显示多次。

在“HTML中混入代码”技术中，这三种情况分别用if 、else、for 等语句表示。实际上，这是根本不必要的。在匹配引擎程序中，这三种情况都可以只用一种结构来表示，那就是List。当List中的元素个数为0，那就是不显示；当List中的元素为1，那就显示一次；当List中的元素为多个，那就显示多次。就这样，匹配引擎只用一种数据结构，就可以表达所有的显示逻辑。

“层次匹配”的好处是显而易见的。首先，显示逻辑代码不需要调用具体的页面技术API，只需要生成“页面数据模型”，这就解除了对具体页面技术的依赖。其次，显示逻辑代码存在于后台，不在页面模板中，从理论上来说，“层次匹配”和“分块取用”是同样强大的，同样可以实现递归显示树形数据、页面布局插入其他页面的结点等高级功能，只需要在页面数据模型和匹配引擎上做些文章即可。再次，“层次匹配”的显示逻辑代码的重用度是最高的，因为，“页面数据模型”是对动态数据模型的包装，这份数据既可以用于匹配引擎，也可以用于其他页面技术，比如“分块取用”和“HTML中混入代码”，所以，这部分显示逻辑代码是完全可以重用的。

除了上述优点之外，“层次匹配”的最大优势是污染度和侵入度最低。页面模板里面一点代码都没有，免除了代码对HTML的污染和侵入；代码里一点HTML也没有，一点具体的页面技术依赖也没有，免除了HTML和页面技术对代码的污染和侵入。当然，也不能说一点侵入和污染都没有。下面我们就来讲这个问题。

“层次匹配”的页面逻辑代码需要根据动态数据模型构造出页面数据模型。在构造页面数据模型的过程中，需要在动态数据模型之上加入一些“显示开关”类的数据结构。

比如，这样的页面逻辑，如果A小于0，就不显示动态文本t001，如果A > 0就显示动态文本t001。那么，显示逻辑就需要根据A的值，生成对应的数据结构（元素个数为0的List，或者元素个数为1的List）。现在的问题是，这些和显示开关对应的页面数据结构应该放在哪里呢？

如果是Javascript这般强大的动态类型解释语言，问题相对容易解决。我们只要在原始动态数据模型之上添加新的“显示开关”属性就可以了。如果是静态类型编译语言，就比较麻烦了，只能采用HashMap之类的动态数据结构来构造整个数据模型，因为HashMap可以任意添加新的属性条目。这也是我的建议。因为页面显示本来就涉及到大量的动态性，如果能用动态类型语言就尽量用，如果不能用，那就尽量使用HashMap这样的动态数据结构。

上述的“页面数据模型”构造问题是存在于后台代码中的根深蒂固的无法消除的问题。这部分侵入和污染是不可避免的。因为，我们总得找一个地方实现这个页面逻辑。我们只能想办法减少这部分侵入和污染。不过，两害相权取其轻，与其他页面技术相比，这点小问题还是可以接受的。

“层次匹配”的另一个问题存在于页面模板中。在前面的例子中，<!-- begin a -->和${var}这样的标记对HTML也造成了一定的污染和侵入。当然，相对于HTML混入的代码来说，这点污染和侵入可以忽略不计。不

过，另外一个问题——模板解析器，却是无法忽略不计的。

根据我自己的经验，匹配引擎的开发是令人愉快多的，几个递归结构加上模式匹配就搞定了。但是，涉及到字符串处理的模板解析器就不同了，繁琐细碎，涉及到大量的字符串查找和比较，只能一步步死抠和细抠，十分令人头痛和厌恶。

除此之外，模板解析器还有一个问题，那就是自定义标记的问题。在HTML这样的XML格式文档中，我们可以用<!—begin ... -->这样的XML注释作为自定义标记。但是，如果换成其他的文本，比如说代码和SQL，这样的标记就不太合适了。为了让模板解析器达到最大的通用性，最好引入一套机制，允许用户自定义文档划分标记。这就进一步增加了模板解析器的复杂性。

因此，即使这种模板解析器比脚本解析器和XML解析器简单了许多，也是相当繁琐的。那么，有没有办法避免模板解析器的问题呢？答案是，有。下一章讲述一种能够免除模板解析器的方案。这种方案是在一种叫做Flyweight Pattern（轻量级模式）的设计模式上实现的。

1.28 《编程机制探析》第二十七章 Flyweight

发表时间: 2011-10-19

《编程机制探析》第二十七章 Flyweight

上一章推介了一种叫做“层次匹配”的页面生成技术。这种技术有诸多优点，但实现起来有一个令人头疼的麻烦之处——模板解析器。凡是涉及到字符串处理的工作，一般都是琐碎乏味的。模板解析器就是如此。
本章讲述一种方案，既可以利用上“层次匹配”的妙处，又可以免除模板解析器的实现。这种方案基于一种叫做“Flyweight Pattern”（轻量级模式）的实际模式，我称之为“Flyweight匹配”方案。在讲述这个方案之前，我们需要对Flyweight Pattern有一个基本的理解。
Flyweight Pattern常见于可视化格式文本编辑器（Rich Text Editor）中。在格式文本编辑器中，一篇文档中可能包含各式各样的文本格式——斜体、黑体、粗体、下划线，等等。这些文本都是一小块一小块，或者一小段一小段的，整篇文档对象由很多小文本对象共同构成。
应对这类需求，有两种实现方案。一种就是“层次结构化文档”方案，比如，HTML就是一种典型的包含了结构信息和格式信息的层次结构化文档。整个文档解析之后，就变成了一个树形文档结构。这种方案的缺点是粒度太小，整个文本被切割成一小块一小块的对象。
另一种方案就是Flyweight。在这种方案中，文档本身并不被切割，仍然保持为一整块大的文本。那些一个个的小小的分块格式信息都分别对应着一个个的小小的对象。这些对象的组织结构有些类似于“层次结构化文档”，也是树形的。不同之处在于，这些小小的对象中，并不存放一小块文档，而是存放着一对位置信息。这对位置信息对应着一整块大文本中的起始位置和结束位置。即，每一个小对象对应着一整块大文本中的某一块文本。类似于这种场景的设计模式就叫做Flyweight Pattern。这种设计模式的主要好处就是保证了一整块大资源的完整性，不需要把整块资源切割成一小块一小块的资源。另一个好处就是文本和结构格式信息的彻底分离，避免了结构格式对文本的污染和侵入。
Flyweight Pattern比较简单，没什么好讲的。本章主要讲解基于Flyweight Pattern之上的“Flyweight匹配”文本生成技术。在讲述的过程中，Flyweight Pattern的一些特点会自然而然地体现。
本章讲述的Flyweight匹配技术是一种通用文本生成技术，包括但不限于HTML页面生成。本章采用的例子也不限于HTML文本，而是通用文本。下面，我们从一段通用文本的例子开始。比如，我们有这么一段文本。
Long long ago, there was living a king..... blabla
假设那段文本的长度为1000。按照Flyweight Pattern原则，我们定义一套用标志文本位置的数字，组成一个树形结构，把这段文本划分成为如下的树形结构：

```
1---1000
   1---100
101 --- 500
  101---200
     201---400
401--500
   501---1000
```

如果开发语言是动态类型解释语言，上述的这个树形数据结构可以用代码中的数据结构直接表示。如果开发语言是静态类型编译语言，上述的这个树形结构数据最好用XML格式来表示。XML虽然不适合用来表示代码逻辑，但用来表示数据结构还是比较合适的。

有了这套数据结构，解析器的工作就很简单了。解析器只需要根据这个结构中的位置信息，从那个长度为1000的整块文本中取出对应的文本块，然后再构造成树形结构的文档就可以了。注意，为了动态文本的输出的便利和方便，得到的这个结果文档就是一个分成一层层、一块块文本的结构，不再遵从Flyweight Pattern。因此，Flyweight匹配技术只有从位置定义的角度来说，是Flyweight Pattern，从最终的内存模型的角度来说，就不是Flyweight Pattern了，而是传统的层次结构化文档。这是因为在一般的文档层次划分中，小文档对象的粒度不会太细，个数不会太多。如果小文档对象的粒度太细，个数太多的话，那么，内存模型方面，也应该使用Flyweight Pattern。从这一点，我们可以窥见Flyweight Pattern的具体应用场景。

为了清晰起见，上述结构中的位置信息用的全都是绝对位置信息。但是，在真正的文本解析过程中，文本是按照递归过程，一层层、一块块向下取的，即，先取出上层的大块文本，再根据下层的位置定义，取出下层的小块文本。所以，位置信息应该采用相对位置，即相对于上层大块文本开头的相对位置。那么，上面的位置信息就应该是这样：

```
1---1000
   1---100
101 --- 500
  1---100
     101---300
301--400
   501---1000
```

我们还可以给这些位置定义上加上名字。

```
top: 1---1000
   a1: 1---100
a2: 101 --- 500
  b1: 1---100
     b2: 101---300
b3: 301--400
   a3: 501---1000
```

有了这些名字，动态数据模型中的一层层属性名就可以匹配上去，生成最终的文本。

这种方案，我称之为“层次位置”方案。这种方案可以工作，但还是不够好。“层次位置”方案有两个主要问题。

第一个问题是位置信息不够清晰。为了程序处理方便，我们采用了相对位置信息，这就使得本来就不容易写对的位置信息更加容易出错。

第二个问题是解析难度和效率。在“层次位置”方案中，解析难度已经大大降低，可以根据位置信息直接构造

文本块，但是，仍然需要递归解析，而且，解析的效率也没有达到最高，需要从上到下，一层层的分块。
如何解决这两个问题呢？“层次位置”方案可以继续进化为“绝对位置”方案，这两个问题就迎刃而解了。
在“绝对位置”方案中，我们把位置信息分成两个部分。第一个部分，只定义文本切割绝对位置，不考虑层次关系。第二个部分，只定义层次关系，不管文本切割绝对位置。
上述的位置信息用“绝对位置”方案来描述的话，就是这样：
第一部分——文本切割绝对位置
t1: 1
t2: 101
t3: 201
t4: 401
t5: 501

第二部分——层次关系
top:
  a1: t1
a2:
  b1: t2
    b2: t3
b3: t4
  a3: t5

有了第一部分——文本切割绝对位置，解析器的工作就十分轻松了，只需要按照绝对位置将文本一块块切开就可以了，不需要考虑层次关系。得到了一块块的文本之后，再根据第二部分——层次关系，将一块块文本对象组织起来，构成一个层次结构化文档。到了这个地步，解析器就是一个简单的文本切割组合器，没有任何的实现难度。也就是说，我们不再需要文本解析工作。
如果应用“绝对位置”方案的话，我们不再需要在HTML页面中添加自定义标记，只需要为每一个HTML页面定义一套“绝对位置”信息即可。从而完全消除了自定义结构划分标签对HTML的污染和侵入。
“绝对位置”方案虽然看起来很美，但是，却存在一个致命缺陷。那就是，我们没有一个简单的方法来定位文本切割的绝对位置。现有的可视化文本编辑器中，只会显示当前光标的行数和列数，并不会显示光标前面的所有字符数。前面讲述的“层次位置”方案也有同样的问题。如果这个问题不解决，本章讲述的基于切割位置的Flyweight方案就只是没有意义的空中楼阁。
要解决这个问题，我们必须编写一个特殊的文本处理程序——特征字符串查找器。这个程序的功能就是寻找文本中的特征字符串，并返回该文本存在的绝对位置。
比如，还是前面的例子，我们需要按照绝对位置来切割那段长度为1000的文本。
Long long ago, there was living a king..... blabla
这段文本需要切割的位置，分别存在着“s2”、“s3”……等特征字符串。我们把这些特征字符串放在一个List中，作为参数，传给特征字符串查找器，就能够得到这些字符串在文档中的绝对位置。比如：
s1: 1

s2: 101
s3: 201
s4: 401
s5: 501
为了方便用户，上述的特征字符串也可以引入空格缩进，写成类似于层次结构的样式。如：
s1
 s2
 s3
  s4
s5
s6
这种层次结构只是一种表面上的显示，方便查看，实际上是不存在的。另外，在实际的应用中，第一个特征字符串是不需要的。放在这里只是为了方便查看。
有了这些特征字符串，我们可以得到需要的绝对位置信息。这里有两个问题需要注意。
第一个问题是特征字符串的特殊性问题。特征字符串必须足够特殊，只应该出现在需要切割的位置，在文本的其他位置不应该出现。这样，才能够保证找到的绝对位置是正确的。为了保证特征字符串足够特殊，我们可以把需要切割处的特征字符串取得长一些，直到能够保证独一无二为止。
第二个问题是文件编码问题。这是一个很重要的主题，我稍微费点口舌。
我们首先需要弄明白的问题是，什么情况下，我们才需要考虑编码？我们一定要牢牢记住一点，编码是字符串文本资源特有的属性，只有在处理字符串文本资源的时候，我们才需要考虑编码，处理二进制资源的时候，我们不需要考虑编码。
那么，什么是编码呢？我们知道，计算机的最基本存储单元是八位（8 Bits）的字节（Byte）。字符串文本资源的最基本处理单元叫做字符（char）。在简单情况下（比如一些简单的西欧字符），一个字符（char）可以用一个字节（byte）来表示。但是，在一些复杂情况下（比如一些至少需要双字节表达的亚洲字符，典型的比如简体汉字、繁体汉字等），一个字符（char）可能对应一个、两个、三个、甚至四个字节（byte）。因此，在很多语言中，char类型都是4个字节（byte）的。
字符串处理程序需要处理的是字符（char），而计算机文件中的基本存储单元是字节（byte）。那么，当进程把字符串文本资源从文件中读取出来的时候，就需要把文件中的字节（byte）转换成字符（char）。这个把字节（byte）转换成字符（char）的操作，就叫做编码（encoding）。
当进程把字符串文本资源存入到文件中的时候，又需要把字符（char）转换成字节（byte），这个操作叫做解码（decoding）。
于是我们就得到了编码（encoding）和解码（decoding）的定义。
编码（encoding）：byte -> char ； byte[] -> char[]
解码（decoding）：char -> byte ； char[] -> byte[]
无论是编码（encoding），还是解码（decoding），都需要一个源字符集（source char set）和一个目标字符集（target char set）。
无论是编码（encoding），还是解码（decoding），都是把资源（byte[]或者char[]）从源字符集（比如，GBK）编码方式转换成目标字符集（比如，Unicode）编码方式。

因此，无论是编码（encoding），还是解码（decoding）都可以叫做编码转换。不过，还有一种特殊情况，源字符集和目标字符集相等。这种情况下，编码转换就非常简单，只是一个简单的映射工作，把char展开成byte，或者byte聚合成char，这种情况下，实质上是不需要“转换工作”的。

编码无处不在，所有的文本资源都有自己的编码，所有的进程也都有自己的编码。比如，页面资源文件本身有编码，文本编辑器有编码，文本处理程序也有编码，数据库有编码，浏览器也有编码，开发语言编译器、解释器、执行器也都有编码。

不同的进程可能有不同的编码，即，可能有不同编码的字符串（char[]）。那么，不同编码的进程之间进行数据交换的时候，必须进行编码转换，这个转换的桥梁就是最基本的数据单元格式——字节流（byte[]）。

两个进程之间的文本数据通信过程是这样：char[] -> byte[] -> char[]。即，经过一个解码（decoding）后再编码（encoding）的过程。这个过程颇为复杂，涉及到两次编码转换和两个字符集（char set）。

假设第一个进程叫做process1，其进程内的char[]的字符集为process1_charset；第二个进程叫做process2，其进程内的char[]的字符集为process2_charset。这时候的编码转换有两种方案。

第一种方案。process1认为另一个进程process2的字符集和自己是一样的，process1直接把process1_charset的char[]数据映射为process1_charset的byte[]。注意，由于char[]和byte[]的字符集编码是一样的，这个过程中不涉及到编码转换，只涉及到映射，即把一个char展开为一个、两个、三个甚至四个byte。做完这个简单的解码处理之后，process1把byte[]和process1_charset信息一起发给process2。

接收到byte[]和process1_charset之后，process2把process1_charset信息和自己的process2_charset进行比较，如果process1_charset和process2_charset相同，那么，只需要把byte[]映射为char[]即可，不需要编码转换。如果process1_charset和process2_charset不相同，就需要一个编码转换的过程。这就需要一个编码转换器（process_charset1 -> process2_charset）。

第二种方案。process1认为自己知道另一个进程process2的目标字符集，假设为target_charset，那么，process1在把char[]转换为byte[]的时候，就会使用一个编码转换器（process_charset1 -> process2_charset）来进行解码。

当process2收到byte[]和target_charset之后，process2把target_charset信息和自己的process2_charset进行比较。最好的情况是相同，process2就不需要做额外的编码转换工作了，只需要进行简单的映射。如果不同的话，就说明process1的编码转化工作白做了。Process2还需要一个编码转换（target_charset -> process2_charset）的过程。

从上面的例子可以看出，编码转换器的个数可能会非常多。编码A转换成编码B（即编码为A的char[]转换成编码为B的char[]），需要一个编码转换器A2B。编码B转换成编码A，又需要一个编码转换器B2A。编码A转换成编码C，又需要一个编码转换器A2C。这是一个排列组合问题。假设有N种编码，那么，就需要N * (N – 1)种编码转换器。每多一种编码，整个编码转换器的个数就会呈几何级数增长。

为了避免这个问题，人们会选择一种比较通用的编码作为中间编码，所有其他的编码都可以和这个中心编码相互转换。这样，转换器的格式就大大减少。如果有N种编码和一种中间编码的话，编码转换器的总是就是 2 * N 个。

比如，Java 虚拟机就采用unicode 作为中间编码。Java虚拟机中的所有字符串（char[]）都是unicode编码的。Java开发包中提供的转化器也都是以unicode为中间编码的，都是A -> unicode（byte[]->char[]，编码），uncode -> A（char[]->byte[]，解码），B -> unicode（byte[]->char[]，编码），unicode -> B（char[]->byte[]，解码）这样的转换器。

编码转换并不是看起来这么简单，而是相当复杂易错，很容易引起编码问题。编码转换必须明确地知道源编码和目标编码，一步弄错，就全盘皆输，整个文本就成了乱码。
我们举一个例子。以Java服务器和浏览器之间的HTTP请求应答为例。
我们把浏览器编码叫做 Browser_Charset，把JVM编码叫做JVM_Charset（通常等于服务器系统编码）。
当浏览器的数据进入到JVM的时候，是一个带有Browser_Charset的byte[]。
如果用户处理程序需要一个Char类型（String）的数据，那么JVM会好心好意地把这个byte[]转换成Char。使用的转换器是 JVM_Charset -> Unicode。
注意，如果这个时候，Browser_Charset 和 JVM_Charset并不相等。那么，这个自动转换是错误的。
为了弥补这个错误。我们需要做两步工作。
(1) Unicode -> JVM_Charset，把这个Char 转换回到原来的 byte[]。
(2) Browser_Charset -> Unicode，把这个还原的byte[]转换成 String。
这个效果，和直接从HTTP Request取得byte[]，然后执行 (2) Browser_Charset -> Unicode 的效果是一样的。
如果在Request里面设置了CharacterEncoding，那么POST Data参数就不需要自己手工转换了，web server的自动转换就是正确的。URL的参数编码还涉及到URL编码，需要考虑的问题多一些，没有这么简单。
JVM把数据发到浏览器的时候。也需要考虑编码问题。可以在Response里面设置。另外，HTML Meta Header里面也可以设置编码，提醒Browser选择正确编码。
文本编辑器、代码编译器、代码执行器、虚拟机、数据库、浏览器（HTML元素）中都有设置编码的地方，文本文件读写API中也提供了设置编码的参数，HTTP Request对象和HTTP Response对象中也有设置编码的接口。
为了避免乱码问题，我们最好把所有涉及到的编码都设成一样。
最难处理的编码问题要属代码中的字符串资源。这部分字符串资源涉及到代码编辑器、代码编译器的编码问题，很容易出错。我的建议是，最好把代码中的字符串资源从代码中分离出去，放在一个单独的文本资源文件中。这个文件有自己的编码，可以由文本编辑器来设定。代码需要调用文本资源的时候，只要提供正确的编码，就可以获取正确的文本。
那么，这个文本资源文件应该放在哪里呢？常见的方法如下。
第一个方法是把文件放在本应用的相对路径内，然后，利用相应的API获取正确的文件路径。比如，在Java Web开发包中（servlet API）中，ServletContext对象中有getSystemResource（path)和getRealPath（path）这样的方法，获取的路径就是${webapp} / path。
这种做法有两个缺点。第一个缺点是造成了对Servlet API的依赖。第二个缺点是不容易打包发布。我们如果把文本资源打包到jar中，就无法使用servlet API来获取该文件的正确路径了。
更好的做法是使用ClassLoader的getResource（path）或者 getSystemResource（path）方法。这两个方法能够获取classpath中的文件，包括jar包里面的文件。比如：
javax.servlet.ServletContext.class.getClassLoader().getResource("javax/servlet/ServletContext.class")
得到的结果是
jar:file:/${webserver}/lib/servlet-api.jar!/javax/servlet/ServletContext.class

假设web.ListController是web应用中的一个类。
web.ListController.class.getClassLoader().getResource("web/ListController.class")

得到的结果是
file:/${webapp}/WEB-INF/classes/web/ListController.class

web.ListController.class.getClassLoader().getResource("messages.properties")
得到的结果是
file:/ ${webapp}/WEB-INF/classes/messages.properties

以上是Java的例子，在其他语言中，也有类似于ClassLoader和classpath的对应物。请读者自行研究。
编码问题涉及到方方面面，不可轻忽。对于前面讲到的“绝对位置”方案来说，编码更是非常重要，丝毫错不得。因为，编码直接决定了特征字符串在整个文本中的绝对位置。
由此可见，“绝对位置”方案用起来是相当麻烦的，虽然节省了文本解析器，但是却引入了一个特征字符串查找器。当然，特征字符串查找器的实现比文本解析器还是简单许多的。
为了使用“绝对位置”方案，我们需要把“文本切割绝对位置”和“层次关系”这两个部分完全分成两个文件。这是为了应对HTML的内容变动。
当HTML内容变化极大，影响到层次结构关系的时候，“文本切割绝对位置”和“层次关系”这两个部分都需要改动。但是，这种情况很少见，大多数时候，HTML的内容变动不会影响到HTML结构。这时候，我们只需要修改“文本切割绝对位置”，而不需要修改“层次关系”，所以，我们把这两部分放到两个不同的文件中。
“层次关系”这部分变动可能不大，我们可以手写，放在那里就可以了。“文本切割绝对位置”的变动可能大一些，但这部分内容比较整齐和简单，我们可以采用批量处理自动生成的方法。
当一批HTML页面第一次建好的时候，我们要用特征字符串查找器批量处理所有的HTML页面，查找其中的特征字符串的位置，并据此生成一批对应的“文本切割绝对位置”。
当HTML内容的每一次改动之后，就运行这个批量处理文件，根据时间，处理其中变动过的HTML，并重新生成“文本切割绝对位置”。
这个过程比较复杂，让我们从头走一遍。首先，我们有了一批HTML文件。我们需要为这些HTML文件定义一批“切割位置特征字符串”的对应文件。然后，我们用特征字符串查找器批量处理这些文件（HTML文件以及对应的特征字符串文件），自动生成一批“文本切割绝对位置”文件。然后，我们根据“文本切割绝对位置”文件，手工写一批“层次关系”文件。有了这样一套工具和流程，“绝对位置”Flyweight方案才有实用性。
不过是文本结构分块而已，却引入了这么多麻烦，到底值不值得？这个就见仁见智了。
在HTML很长、数据模型层次很深的情况下，“HTML混入代码”技术将会陷入噩梦，if else for 语句块经常找不到头和尾。“层次匹配”技术的 begin 和 end 标记中都带有对应一致的名字，一定程度上缓解了这个问题。但是，当begin end 嵌套层次比较多，甚至可能还有同名块的情况下，整个文档结构就很难看清楚了。在这种情况下，“绝对位置”Flyweight方案的优势就极为明显了，“层次关系”中的整个文档结构一目了然。

1.29 《编程机制探析》第二十八章 ORM

发表时间: 2011-10-19 关键字: orm, 框架, 编程, object, relation

《编程机制探析》第二十八章 ORM

本章的主题是ORM（Object Relation Mapping，对象与关系数据的映射）。
ORM是一种技术框架，其主要作用是在面向对象语言和关系数据库之间搭建一个转换桥梁。这个转换是双向的。ORM既可以把关系数据转换为对象，也可以把对象转换为关系数据。
ORM种类繁多，功能或繁或简，这里不便一一列举。本章只拣ORM的一些重要特性进行阐述。
ORM的最基本功能是关系数据到对象的转换。程序进行数据库查询时，会获取到一行行的关系数据的集合。这个关系数据集合的数据结构和关系数据表一模一样，都是一个二维表结构，在表头上，每一列都有自己的名字。比如，我们获取一个表名为department（部门）的数据表中的所有数据，得到的结果数据集如下：
id name
01 Office
02 QC
03 IT
04 Design
05 Customer Service
可以看到，这就是一个带有列名的数组结构。为什么不直接返回数组结构，而是提供一个Iterator呢？这是因为，数据库客户端（即调用数据库的进程）为数据结果集分配的进程内数据缓冲空间是有限的，如果结果数据量超过数据缓冲空间的话，超出的那些数据就只能在数据库服务器中等待下一次传唤。而且，很多时候，程序不再需要后面的数据，这时候，这种一部分一部分获取的Iterator Pattern的优势就体现出来了。
关系数据到对象的映射很简单。一行关系数据就是一个数组，只要按照顺序，根据列名，利用Reflection机制，将数据设置到对象的同名属性中就可以了。这个地方无需细说。
一般的情况下，一个关系表定义和一个对象定义之间是一对一的关系。即一个对象定义映射一张表定义。但凡事都有例外。一些功能强大的ORM提供了更加丰富的映射关系，一个关系表定义映射多个对象定义，多个关系表定义映射一个对象定义，甚至还可以分级别映射，一个映射表定义可以映射多级对象定义（即映射到对象中的属性对象），等等。除此之外，ORM还可能提供部分映射，即，把一张表的某些字段映射到对象中的某些属性。这种特性叫做Fetch Group（按组获取）。
这些ORM特性都有各自的特殊应用场景。不过，这不是本章所关心的。本章所关心的一个很有用的ORM特性是Lazy Fetch（延迟获取）。这个特性是用来代替关联表查询的。
关联表查询，即两张、或者两张以上的关系表一起进行条件查询。假设第一张表的记录数是N1，第二张表的记录数N2。这两张表关联查询时，需要处理的记录数就是N1 * N2，即两张表的笛卡尔乘积。当关系表中的记录数非常大的时候，关联查询的开销就会非常巨大。这时候，我们就要考虑Lazy Fetch（延迟获取）的方案，即，两行表分开查询，先按照某种条件，查询其中一张表，然后，再根据查询结果，查询另一张表。
下面举一个例子。前面已经有了一个department（部门）表，我们再引入一个employee（雇员）表。
id name department_id

001 John 01
002 Lee 01
003 Van 01
004 Lily 02
005 Harry 02
006 Long 02
007 Sunny 03
008 Tom 03
009 Tiger 03

这两张表的定义为：
Create table department (id, name)
Create table employee (id, name, department_id)
外键关系: deparment_id <-> department.id
对象关系: Employee对象中有一个Department对象属性。

我们先来看关联查询的例子：
Select employee.*, department.name
from employee, department
where employee.department_id = department.id

然后，我们把这个例子改成Lazy Fetch，即两张表分开查询。最直观的方法是先查询employee表，然后根据每个employee中的department_id去查询department表。产生的SQL如下：
Select * from employee
Select * from department where id = 001
Select * from department where id = 001
Select * from department where id = 001
Select * from department where id = 002
Select * from department where id = 002
Select * from department where id = 002
Select * from department where id = 003
Select * from department where id = 003
Select * from department where id = 003

可以看到，这种方案的效率极低，产生了太多的SQL。这就是数据库查询中的所谓“1 + n”问题。下面，我们对这种方案进行改进。第一个想法就是合并其中重复的SQL，我们得到4条SQL语句：
Select * from employee
Select * from department where id = 001

Select * from department where id = 002
Select * from department where id = 003

这个结果仍然不能让人满意。SQL语句还是太多。我们继续优化，把上述的SQL语句合并为两条SQL。
Select * from employee
Select * from department where id in (001, 002, 003)
这样，我们就把“1 + n”问题转化成了“1 + 1”问题。
Lazy Fetch的实现流程总结如下：
（1）查询第一张表
（2）根据结果集，收集第二张表的id，并消除重复记录。
（3）根据第二张表的id集合，查询第二张表
（4）根据对象关系和id对应关系，将两张表查询出来的对象组装起来。

ORM的第一项功能——关系数据映射到对象，就讲到这里。下面我们来看ORM的第二项功能——SQL动态拼装。
有时候，用户需要利用各种条件组合来查询数据，服务器就需要根据用户输入的各种条件组合，拼装出对应的SQL。除了最原始的字符串拼接发之外，还有两种看起来比较清爽的动态拼装SQL思路。
第一种思路叫做条件对象组装。
这种思路基于一堆称为条件对象（Criteria）的API（Application Programming Interface）。所谓条件对象，就是一堆逻辑操作，如：与对象（and）、或对象（or）、比较对象（大于、小于、等于）等等。
这些条件对象可以组装起来，形成一个树形结构的对象，这个对象输出的结果就是一串SQL条件语句。
这个思路像什么？没错，很像是前面章节中讲过的“页面组件”技术，都是在一堆对象的代码中夹杂着字符串输出语句，最后再统一输出。
我是不赞同这种思路的。在我看来，SQL本身是一种可读性很好的领域专用语言（DSL，Domain Specific Language），其可读性远远超过条件对象（Criteria API）。
我认同这样一种观点——DSL Over API（领域专用语言优于编程函数定义），因为，DSL接近于自然语言，可读性远远超过API。只不过，在现实的世界中，API的定义十分简单，而DSL的定义十分困难，因为DSL涉及到语法解析器和解释器，这两者都不是省油的灯，不是一般人可以写出来的。
现在，既然有了现成的DSL（即SQL）却舍而不用，反而去用最原始的Criteria API，这不是舍本逐末吗？
我赞同第二种思路——SQL模板技术。
这种思路基于前面章节中讲述的“层次匹配”文本生成技术。程序员在SQL中加入begin end ${} 之类的自定义标签，将动态部分划分出来，然后，根据用户输入条件构造一个显示数据模型，最后，将SQL模板和显示数据模型匹配起来，就可以得到最终的SQL语句。
当然，不嫌麻烦的话，也可以采用“绝对位置”Flyweight的方案。不过，SQL一般都不会太长，层次结构也不会太复杂，使用Flyweight方案的好处很可能不足以抵偿带来的麻烦。
ORM的第三项功能是SQL命名参数。
数据库允许用“?”这样的通配符来代替SQL中的参数，如：
select …. where id = ? and name = ?

update … set name = ? where id = ?
delete … where id = ?
insert …. values (?, ?)

使用这种带有参数的SQL的时候，程序员需要把参数值按照顺序放进一个数组中，和带参数的SQL一起传给数据库。
这种带参数的SQL有两个问题。第一个问题是可读性不好，所有的参数部分都是“?”，不知道具体应该对应怎样的数值。第二个问题是参数值顺序不好掌握，数组中的参数值必须对应在SQL中的“?”顺序，这个顺序是比较难以对应的，程序员不得不非常小心仔细，有时需要耗费相当的精力。
为了解决这个问题，一般的ORM框架中都引入了“SQL命名参数”的功能。
SQL命名参数用参数名代替了“?”和位置顺序。比如：
select … where id = $id and name = $name
update …. set name = $name where id = $id
delete …. where id = $id
insert … values ($id, $name)

我们可以用“层次匹配”技术来处理这个SQL，把$id和$name替换成?，并且按照参数名取出数据模型中的对应属性，并根据参数名的顺序填入到参数数组中，最后把带有?的SQL和参数数组一起传给数据库。
可以看出，这个工作不仅是简化了参数设置工作，同时还完成了“对象到关系数据的映射”的工作。比如，上面的update、delete、insert语句，就是根据一个对象的属性信息，对数据表进行增、删、改等操作。
ORM的第四项功能是缓存（Cache）。
首先，我们需要明确缓存的应用场景。缓存的目的是为了提高查找速度，但不是所有情况都可以应用缓存。当用户对数据准确度要求很高的情况下，比如银行转账，是不可以应用缓存的，因为缓存具有时效性，很可能过期。只有在对数据准确度要求不高、能够容忍一定程度过期数据的情况下，缓存才有用武之地。
衡量缓存优劣的最重要参数是命中率，即从缓存中查到所需数据的概率。我们可以用一个简单的公式来大致表述：命中率 = 缓存命中次数 / 缓存数据总量
ORM缓存分为两种——ID缓存和Query缓存。
ID缓存，顾名思义，就是以关系表ID为索引的缓存。每行关系数据的ID都是唯一的，对应的对象的ID也都是唯一的。这种缓存很容易理解，不必赘述。
Query缓存，是针对SQL查询语句的缓存。一条SQL查询语句可能查出来一个关系数据集。这个关系数据集也可以存放在缓存中。当下次用户再用同样的SQL查询语句的时候，就可以直接返回Query缓存中的数据结果集。
ID缓存和Query缓存可以分开实现，也可以合并实现。
分开实现的话，两个缓存各不相干，各管一摊，实现上比较简单，但是，时间空间效率和命中率都不高。比如，下面的两条SQL语句。
select …. where department = “QC”
select … where id = $id

第一条SQL语句不是ID查询，是Query查询，对应的是Query缓存。第二条SQL语句对应的是ID查询，对应的是

ID缓存。这两条SQL语句查询出来的结果，分别存放到两个不同的缓存空间中。
但是，第一条查询语句中的结果集，很可能包含了第二条查询语句的结果。也就是说，ID缓存和Query缓存有很大的可能性存在重复数据。
为了时间空间效率和命中率起见，ID缓存和Query缓存最好合起来实现，共用同一份缓存。其实现原理如下：
当ID查询的时候，ORM缓存把ID作为键值，把对象存放到缓存中。这时候，实现的是ID缓存的功能。
当Query查询的时候，ORM缓存把结果集一条条展开，把一个个对象的ID作为键值，把对象存放到缓存中，这时候，实现的是ID缓存的功能。然后，根据结果集构造一个ID列表，把SQL本身作为键值，把这个ID列表存放到缓存中，这时候，实现的是Query缓存的功能。
对缓存进行查询的时候，如果键值是ID，那么，就直接取出ID对应的对象。如果键值是SQL，那么，就以SQL为键值，获取的就是一个ID列表。然后，根据这个ID列表，从缓存中把对象一个个取出来，组成一个对象列表，最后返回。
就这样，ID缓存和Query缓存就统一起来了。
除了命中率问题，缓存还需要考虑的重要问题是过期数据清理问题。最简单的过期数据清理策略是按时清理，定义一个清理周期，每隔一定时间就清除缓存中所有数据。
稍微复杂一点的过期数据清理策略是实时清理。当程序遇到任何一条增删改SQL语句的时候，就根据SQL中涉及到的表名，把缓存中所有相关的数据全都清理掉。这种策略很可能会误杀不少没有过期的数据。但缓存就是这样，宁可误杀一千个非过期数据，不可放过一个过期数据。
如果是增删改操作特别多的情况下（按理来说，这种情况下就不应该用缓存），还想使用缓存的话，那么，可以采用更加灵活的数据清除策略，比如，由程序员自己指定清除那些数据，毕竟，程序员自己对于代码逻辑是最了解的。