TURING 图灵程序设计丛书 数据库系列

Apress®

Expert PL/SQL Practices
for Oracle Developers and DBAs

Oracle PL/SQL实战

[美] John Beresniewicz
[美] Melanie Caffrey
[美] Dominic Delmolino
[美] Connor McDonald
[乌克兰] Michael Rosenblum
[英] Adrian Billington
[美] Ron Crisco
[印度] Sue Harper
[美] Arup Nanda
[美] Robyn Sands
[瑞士] Martin Büchi
[美] Lewis Cunningham
[丹] Torben Holm
[瑞士] Stephan Petit
[美] Riyaj Shamsudeen
著

卢涛 译
苏旭晖 李颖 贾书民 审

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书 数据库系列

Expert PL/SQL Practices
for Oracle Developers and DBAs

Oracle PL/SQL实战

[美] John Beresniewicz
[美] Melanie Caffrey
[美] Dominic Delmolino
[美] Connor McDonald
[乌克兰] Michael Rosenblum
[英] Adrian Billington
[美] Ron Crisco
[印度] Sue Harper
[美] Arup Nanda
[美] Robyn Sands
[瑞士] Martin Büchi
[美] Lewis Cunningham
[丹] Torben Holm
[瑞士] Stephan Petit
[美] Riyaj Shamsudeen
著

卢涛 译
苏旭晖 李颖 贾书民 审

人民邮电出版社
北京

图书在版编目（CIP）数据

Oracle PL/SQL实战 / (英) 比林顿(Billington,A.) 等著 ; 卢涛译. -- 北京 : 人民邮电出版社, 2012.11
（图灵程序设计丛书）
书名原文: Expert PL/SQL Practices: for Oracle Developers and DBAs
ISBN 978-7-115-29485-2

Ⅰ. ①O… Ⅱ. ①比… ②卢… Ⅲ. ①关系数据库－数据库管理系统 Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2012)第232411号

内 容 提 要

本书共 15 章，分别由 15 位业内顶级专家担纲撰写。一条 SQL 语句的误用，可能导致作业的运行时间增加几百倍，Riyaj Shamsudeen 巧妙地回答了何时使用 PL/SQL 这一问题；Michael Rosenblum 说明了如果只有到最终运行时，才知道所要运行的 SQL 语句到底是什么，那该如何编写代码；Dominic Delmolino 介绍了用 PL/SQL 进行并行处理的方法，以及这样能获得的益处和产生的额外负担。本书还有更多精彩内容等待读者一探究竟。

本书适合有一定 PL/SQL 基础的读者学习。

图灵程序设计丛书

Oracle PL/SQL实战

◆ 著 [美] John Beresniewicz [英] Adrian Billington
[瑞士] Martin Büchi [美] Melanie Caffrey
[美] Ron Crisco [美] Lewis Cunningham
[美] Dominic Delmolino [印度] Sue Harper
[丹麦] Torben Holm [美] Connor McDonald
[美] Arup Nanda [瑞士] Stephan Petit
[乌克兰] Michael Rosenblum [美] Robyn Sands
[美] Riyaj Shamsudeen
译 卢 涛
审 苏旭晖 李颖 贾书民
责任编辑 王军花
执行编辑 李 静

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：27.25
字数：650千字 2012年 11 月第 1 版
印数：1－3 500册 2012年 11 月北京第 1 次印刷
著作权合同登记号 图字：01-2011-8034号
ISBN 978-7-115-29485-2

定价：89.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

译　者　序

2010年，我参与编写了一本有关Oracle开发的书《剑破冰山：Oracle开发艺术》，这本书的重点是SQL编程，因此曾想过再写一本关于PL/SQL开发的书。2011年10月，在《剑破冰山：Oracle开发艺术》上市大约1周年的时候，我在图灵公司的网站发现了本书英文版正在征集译者，看了下目录，感觉它提到了很多我想表达的内容，而且比较全面、系统。于是，就萌生了翻译的想法。本书由15位作者共同编写，每人负责一章，所以每章都能独立成篇。而把各章内容结合到一起，又基本上涵盖了PL/SQL开发的各个方面。需要指出一点，本书不是PL/SQL入门教材，读者需要有一定PL/SQL基础。实际上，本书大部分内容在Oracle官方的培训计划中属于高级课程*Oracle Database 11g: Advanced PL/SQL*的范畴。

决定挑战这个任务后，我通过图灵副总经理谢工联系到了傅志红副总经理，有幸通过了试译。这本书英文原作有500多页，加上我之前还没有翻译过这么大规模的文字，所以觉得一人翻译在时间上有些吃紧，因此想邀请上本书的几位合著者共同翻译。不过由于各种原因没有成功，最终我还是独自承担了这个任务，在5个月的时间里，完成了初译、复查、修改等工作。

如果没有大家的帮助，我一个人无法完成这本译作。

首先感谢老战友苏旭晖（ITPUB Oracle开发版版主，网名newkid），他现定居加拿大多伦多，从事数据库应用系统的设计与开发工作。虽未见面，但他在我翻译的5个多月时间里，一直不辞辛劳地帮助我解决多处语言和技术难点，并给出了更好的译法。更难得的是，对原书个别不合理的叙述，提出了中肯的意见，使读者不仅能通过本书了解原作者的观点，还能引发进一步的思考。

其次感谢我妻子李颖，她是英语专业毕业生，作为本书的第一读者，帮助我检查出很多生硬的译文，并把它们修改得更加通顺。得益于她的修改，本书才有现在的可读性。同时，她还承担了培养教育孩子的重任，使我可以专心翻译本书。

还要感谢老大哥贾书民，他有10多年的Oracle开发经验，做过许多基于Oracle的大型项目，他通读了我的译文，提出了许多中肯的修改建议。在我试译时，他也帮助我对译文做了很多修改，对我顺利通过试译起到了很大的作用。

感谢图灵公司李鑫编辑、李静编辑、李松峰主任，他们对译文进行把关，并对语言进行润色，使表达更加地道。

最后希望这本书能帮助读者提高Oracle PL/SQL开发水平。由于译者经验和水平有限，译文中难免有不妥之处，恳请读者批评指正!

序

作为图书编辑，我一般满足于在幕后做好自己的编辑工作，很少站到前台来为我负责编辑的图书作序。但是，这次我却破例了。因为我也曾做过开发，这本书的内容勾起了我的很多回忆。

本书旨在教你如何有效地使用PL/SQL。它不是描述语法的书，而是介绍如何把语法及其特性与良好的开发实践相结合，创建更具有可靠性、可扩展性的应用，并使之具备长期可维护性。

不管使用什么工具，首先需要明白的是在什么情况下使用它比较合适。在本书的开篇章节“避免误用”中，Riyaj Shamsudeen巧妙地回答了何时使用PL/SQL这一问题。我把这一章放在本书的开始，也是因为我个人的经历：那是在20世纪90年代末，我进行了最成功的一次性能优化，那次我把客户端PC上的一堆过程代码改写为一个SQL语句以后，一个作业的运行时间从超过36小时缩短到仅仅几分钟。虽然PL/SQL不是导致性能问题的罪魁祸首，但是我得到了一条经验——基于集合的方法（如果有可能）通常优于过程性代码。

Michael Rosenblum紧接着写了动态SQL那一章，其内容相当精彩。他说明了如果只有到最终运行时才知道所要运行的SQL语句到底是什么，那么该如何编写代码。这使我想起20世纪90年代初的一段经历，那时我在陶氏化学公司利用Rdb的扩展动态游标功能集，为一个医疗记录系统写数据装载应用程序。我感觉那是我开发过的最有趣的应用程序之一。

Dominic Delmolino介绍了用PL/SQL 进行并行处理的方法，以及这样能获得的益处和产生的额外负担。谨小慎微，永不为过！我在当数据库管理员时犯下的最大错误之一就是，有一次头脑一热，在一个关键的应用表上设置了一个并行度，其目的是为了想让其中的一个报表程序运行得更快。可是，事与愿违。好像键盘上的回车键连着我的电话似的，修改后大约不到一分钟，我的电话铃就响了，电话线另一端的主管对我的修改非常不满。无须多言，从此我就决定在实施并行前一定要深思熟虑。Dominic写的那章可以帮助我们避免陷入这样的窘境。

本书的多章内容涵盖了代码规范和极其实用的编程经验。Stephan Petit向我们介绍了一套实用的命名和编码规范。Torben Holm讲述了PL/SQL警告和条件编译。Lewis Cunningham阐述了代码分析，使我们认识到真正理解自己所写的代码及其工作原理的重要性，发人深思。Robyn Sands的“渐进式数据建模”一章，令我们对渐进式数据模型的良好设计及灵活性有了更多的思考。Melanie Caffrey介绍了经常使用的各种游标类型，帮助我们在不同条件下正确地选择游标。

其他章节介绍如何进行调试和故障排除。Sue Harper介绍了PL/SQL单元测试，特别是一些如今已经集成在SQL Developer中的功能集（这使我想起了在纸上写单元测试脚本的日子），它可以帮助避免犯回归错误。利用自动化单元测试，能够容易且方便地验证程序，以确保自己不会在

修复一个错误的同时制造了更多的错误。

还有John Beresniewicz介绍“合同导向编程”那一章。John给出的方法的一个关键部分是，在代码的各种特定地方使用断言来验证条件。记得我首次了解断言技术是在做PowerBuilder编程时，那已经是很久很久以前的事了。我很高兴看到John把这项技术和PL/SQL联系起来，将其发扬光大。

Arup Nanda的论述能够帮助我们控制依存和失效问题。依存问题可能会导致发生类似随机的、难以重现的应用程序错误。Arup展示了如何完全掌控那些必将发生的事情，这样你才不会落入意外错误的陷阱。

一般情况下，我们不得不考虑性能和扩展性。Ron Crisco告诉我们通过剖析代码，找到尽可能优化代码的方法。Adrian Billington讨论了从SQL语句中调用PL/SQL的性能问题。Connor McDonald论述了批量SQL操作惊人的性能优势。

关于可扩展性，通常会被遗漏的一面是应用程序的规模和参与开发的人数。PL/SQL是否适合数十到数百位程序员的大规模开发？Martin Büchi在“大规模PL/SQL开发”那章，描述了他在管理由170多位开发者维护的具有1100万行代码的应用中的成功经验，说明了PL/SQL是非常适合于这种任务的。

不难看出我为这本书感到兴奋。作者都是顶级专家，分别介绍了自己热衷的并且特别在行的PL/SQL的某个方面。如果你已经学习了语法，那么坐下来读一读这本书，充分利用PL/SQL和Oracle数据库的强大功能，投身到开发应用程序的事业中来吧！

Jonathan Gennick
Apress编辑主任助理

作 者 简 介

- John Beresniewicz（约翰·贝雷斯尼维奇）是位于加州红木城红木岸（Redwood Shores）的Oracle总部技术团队的一名咨询顾问。他于2002年加入Oracle，负责企业管理器的数据库性能领域，他对诊断和调优包、实时应用测试、支持工作台和Exadata的设计作出了重要贡献。多年以来，他经常在Oracle全球大会和其他会议上发言，发言主题包括数据库性能和PL/SQL编程。他与Steven Fellerstein合著了*Oracle Built-in Packages*（O'Reilly & Associates，1998年）一书，并且是OakTable网络的创始人之一。
- Adrian Billington（阿德里安·比林顿）是应用设计、开发和性能调优方面的顾问。自1999年以来，一直从事Oracle数据库方面的工作。他是www.oracle-developer.net网站的发起人，这个网站为Oracle开发人员提供各种SQL和PL/SQL功能、实用工具和技术。阿德里安还是Oracle ACE，同时也是OakTable网络的成员。现在，他与妻子安吉和三个孩子：格鲁吉亚、奥利弗和伊莎贝拉一起居住在英国。
- Martin Büchi（马丁·步琪）自2004年以来，任Avaloq公司首席软件架构师。该公司是一个标准化的银行软件供应商，其产品基于Oracle RDBMS构建，包含1100万行PL/SQL代码。他与两位同事一起设计了系统架构，并评审了170名全职PL/SQL开发人员的设计和代码，以追求软件的简明、效率和健壮性。马丁经常在Oracle大会上发言。2009年，他被*Oracle Magazine*评选为PL/SQL年度开发人员。从事Oracle数据库工作之前，马丁曾在面向对象的系统、形式化方法和近似记录匹配等领域工作。他拥有瑞士联邦技术研究所的硕士学位和芬兰土尔库计算机科学中心的博士学位。业余时间，马丁喜欢与他的家人一起进行各种户外运动。
- Melanie Caffrey（梅拉妮·卡弗里）是Oracle公司高级开发经理，为不同客户的业务需求提供前端和后端的Oracle解决方案。她是多部技术出版物的合著者，包括*Oracle Web Application Programming for PL/SQL Developers*、*Oracle DBA Interactive Workbook*、*Oracle Database Administration: The Complete Video Course*等，这些书全部由Prentice Hall出版。她在纽约哥伦比亚大学的计算机技术与应用课程中指导学生，教授先进的Oracle数据库管理和PL/SQL开发。她也经常在Oracle会议上发言。
- Ron Crisco（罗恩·克里斯科）28年来分别担任软件设计师、开发人员和项目负责人，并有21年的Oracle数据库工作经验。他在R方法（Method R）公司从事软件设计和开发、软件产品管理（如R方法剖析器、MR工具和MR跟踪）、咨询、教授课程等工作。他的特长

是简化复杂的工作，这在帮助他身边的人完成非凡工作时尤显宝贵。

- Lewis Cunningham（刘易斯·坎宁安）在IT领域已经工作了20多年。自1993年以来一直与Oracle数据库打交道。他的专长是应用程序设计、数据库设计，以及大容量的VLDB数据库编码。目前他任职于佛罗里达州圣彼得堡的一家金融服务公司，担任高级数据库架构师，负责超大规模、高事务率分析型数据库和应用程序的工作。他花了大量时间来与最新的技术和趋势保持同步，并在用户组发表演讲，举办网络研讨会。刘易斯也是一位Oracle ACE总监和Oracle认证专家。他在Oracle技术网发表了数篇文章，并在http://it.toolbox.com/blogs/oracle-guide维护一个Oracle技术博客。刘易斯写了两本书：*EnterpriseDB: The Definitive Reference*（Rampant Tech press，2007年）和*SQL DML: The SQL Starter Series*（CreateSpace，2008年）。他与他的妻子及两个儿子起住在佛罗里达州。可以通过电子邮件lewisc@databasewisdom.com与他联系。
- Dominic Delmolino（多米尼克·德莫里诺）是Agilex技术公司首席Oracle和数据库技术专家，这是一家专门协助政府和私营企业实现信息价值的咨询公司。多米尼克拥有24年以上的数据库经验，其中担任过20多年的Oracle数据库工程和开发专家。他是OakTable网络的成员，并定期出席各种学术会议、研讨会，以及欧洲和美国的用户组会议。他还维护www.oraclemusings.com网站，该网站专注于与数据库应用程序开发相关的数据库编码和设计实践。多米尼克拥有纽约州伊萨卡康奈尔大学的计算机科学学士学位。
- Sue Harper（苏·哈珀）是数据库开发工具组中的Oracle SQL Developer和SQL Developer数据建模器的产品经理。她自1992年以来一直在Oracle公司工作，目前在伦敦办事处工作。苏是一些杂志的特约撰稿人，维护着一个技术博客，并在世界各地的许多会议上发言。她撰写了技术书籍*Oracle SQL Developer 2.1*（Packt，2009），业余时间，苏喜欢步行和摄影。同时，她还花时间到新德里的贫民区做慈善工作，帮助那里的妇女和儿童。
- Torben Holm（托尔·霍尔姆）自1987年以来一直从事开发工作。自1992年以来，他一直致力于与Oracle相关的工作，前四年担任系统分析师和应用程序开发人员（Oracle 7、Forms 4.0/Reports 2.0和DBA），然后做了两年开发（ORACLE6/7、Forms 3.0和RPT以及DBA）。他在Oracle丹麦公司的高级服务组工作了数年，担任首席高级顾问，执行应用程序开发和DBA任务。他还担任过PL/SQL、SQL和DBA课程的讲师。现在，托尔在Miracle A/S（www.miracleas.dk）工作，担任顾问，负责应用开发（PLSQL、`mod_plsql`、Forms、ADF）和数据库管理。10年来他一直在Miracle A/S公司工作。他是Oracle认证开发人员，并且也是OakTable网络成员。
- Connor McDonald（康纳·麦当劳）自20世纪90年代初一直从事Oracle相关工作，他非常熟悉Oracle 6.0.36和Oracle 7.0.12。在过去11年中，康纳曾为位于西欧、东南亚、澳大利亚、英国和美国的公司开发过系统。他已经认识到，虽然世界各地的系统及方法非常多样，但开发在Oracle上运行的系统往往有两个共同的问题：要么避免使用Oracle特定的功能，要么就是采取不太理想的用法或随意乱用它们。正是这种观察，促使他创建了一个提示和技巧的个人网站（www.oracledba.co.uk），并努力在Oracle演讲者组织中发表更多演讲，

以提高PL/SQL的业内认知度和普及度。

- Arup Nanda（奥雅纳·南大）自1993年以来，一直是Oracle DBA，他熟悉数据库管理的所有方面，从建模到灾难恢复。目前，他在纽约州白原市的喜达屋酒店（即喜来登、威斯汀等连锁酒店的母公司）领导全球DBA团队。他是独立Oracle用户协会（IOUG）旗下出版物*SELECT Journal*的特约编辑，在许多Oracle技术盛会，如Oracle全球和本地用户组（如纽约Oracle用户组）中发表演讲，并为印刷出版物如*Oracle Magazine*和网络出版物如*Oracle Techndogy Network*撰写了许多文章。奥雅纳与他人合著了两本书：*Oracle Privacy Security Auditing*（Rampant，2003年）和*Oracle PL/SQL for DBAs*（O'Reilly，2005年）。由于他的专业成就和对用户社区的贡献，Oracle评选他为2003年年度DBA。奥雅纳与他的妻子Anindita和儿子阿尼什住在康涅狄格州的丹伯里。可以通过arup@proligence.com联系他。
- Stephan Petit（斯蒂芬·佩蒂特）于1995年在位于瑞士日内瓦的欧洲粒子物理实验室（CERN）开始了他的职业生涯。他现在是一个软件工程师和学生团队的负责人，负责为实验室和其他部门提供应用程序和工具。工程和设备数据管理系统是这些工具之一，也称为CERN EDMS。像CERN的大型强子对撞机（LHC）项目有40年或以上的生命周期。EDMS是实验室的数字化工程的内存/记忆体。电子文件管理系统中存储了与一百多万件设备有关的一百多万份文件，EDMS也供CERN的产品生命周期管理（PLM）和资产跟踪系统使用。 EDMS几乎完全是基于PL/SQL的，并旨在拥有一个至少与LHC一样长的生命周期。

  斯蒂芬和他的团队一直在完善PL/SQL编码规范和最佳实践，以满足他们非常有趣的各种挑战的组合：几十年的可维护性、可靠性、高效的错误处理、可扩展性、模块的可重用性。团队成员的频繁轮换，其中大部分只是暂时在CERN实习的学生，加剧了这些挑战。最古老的一段代码是在1995年写的，现在仍然在使用——并且成功地运行！除了完善PL/SQL，斯蒂芬还喜欢不时登台表演，比如担任CERN摇滚夏季音乐节的摇滚乐队歌手，以及在多部戏中出演角色。
- Michael Rosenblum（迈克尔·罗森布鲁姆）是Dulcian公司的软件架构师/开发DBA，他负责系统调优和应用程序架构。迈克尔通过编写复杂的PL/SQL例程和研究新功能支持Dulcian开发人员。他是*PL/SQL for Dummies*（Wiley，2006年）一书的作者之一，并在IOUG *Select Journal*和ODTUG *Tech Journal*发表了许多篇与数据库相关的文章。迈克尔是一位Oracle ACE，也经常出席不同地区和国家的Oracle用户组大会（Oracle OpenWorld大会、ODTUG、IOUG Collaborate、RMOUG、NYOUG等），他是ODTUG万花筒2009年"最佳演讲奖"得主。在他的家乡乌克兰，他获得了乌克兰总统奖学金，并拥有信息系统理学硕士学位并以优异成绩获得基辅国立经济大学毕业证书。
- Robyn Sands（罗宾·桑兹）是思科系统公司的软件工程师，她为思科的客户设计和开发嵌入式Oracle数据库产品。自1996年以来，她一直使用Oracle软件，并在应用开发、大型系统实现和性能测量方面具有丰富经验。罗宾的职业生涯始于工业工程和质量工程，她

将自己对数据的挚爱结合到以前接受的教育和工作经验中，寻找新方法来建立性能稳定、易于维护的数据库系统。她是OakTable网络成员，并是下面两本Oracle书籍的作者之一：*Expert Oracle Practices*和*Pro Oracle SQL*（都由Apress出版，2010）。罗宾偶尔在http://adhdocddba.blogspot.com发表一些博客。

- Riyaj Shamsudeen是OraInternals公司首席数据库管理员和主席，这是一家从事性能调优/数据库恢复/EBS1*1i*等领域的咨询公司。他专门研究真正的应用集群（RAC）、性能调优和数据库内部结构。他还经常在其博客http://orainternals.wordpress.com上发表这些技术领域的文章。他也经常出席许多国际会议，如HOTSOS、COLLABORATE、RMOUG、SIOUG、UKOUG等，他是OakTable网络的骄傲一员。他拥有16年以上使用Oracle技术产品的经验，并担任了15年以上的Oracle/Oracle应用程序数据库管理员。

目　录

第 1 章　避免误用 …… 1
1.1　逐行处理 …… 1
1.2　嵌套的逐行处理 …… 3
1.3　查找式查询 …… 5
1.4　对 DUAL 的过度访问 …… 8
1.4.1　日期的算术运算 …… 8
1.4.2　访问序列 …… 9
1.4.3　填充主–从行 …… 9
1.5　过多的函数调用 …… 10
1.5.1　不必要的函数调用 …… 10
1.5.2　代价高昂的函数调用 …… 12
1.6　数据库链接调用 …… 14
1.7　过度使用触发器 …… 15
1.8　过度提交 …… 15
1.9　过度解析 …… 16
1.10　小结 …… 16

第 2 章　动态 SQL：处理未知 …… 18
2.1　动态 SQL 的三种方式 …… 19
2.1.1　本地动态 SQL …… 19
2.1.2　动态游标 …… 21
2.1.3　DBMS_SQL …… 25
2.2　动态思考的样例 …… 26
2.3　安全问题 …… 30
2.4　性能和资源利用率 …… 33
2.4.1　反模式 …… 34
2.4.2　比较动态 SQL 的实现 …… 35
2.5　对象的依赖关系 …… 37
2.5.1　负面影响 …… 37
2.5.2　正面影响 …… 37
2.6　小结 …… 38

第 3 章　PL/SQL 和并行处理 …… 39
3.1　为什么需要并行处理 …… 39
3.2　影响并行处理的定律 …… 40
3.3　大数据的崛起 …… 40
3.4　并行与分布式处理 …… 41
3.5　并行硬件体系结构 …… 41
3.6　确定目标 …… 42
3.6.1　加速 …… 42
3.6.2　按比例扩展 …… 43
3.6.3　并行度 …… 43
3.7　用于并行处理的候选工作负载 …… 43
3.7.1　并行和 OLTP …… 43
3.7.2　并行和非 OLTP 工作负载 …… 44
3.8　MapReduce 编程模型 …… 44
3.9　在使用 PL/SQL 之前 …… 45
3.10　可用于并行活动的进程 …… 45
3.11　使用 MapReduce 的并行执行服务器 …… 46
3.11.1　管道表函数 …… 46
3.11.2　指导 …… 60
3.11.3　并行管道表函数小结 …… 61
3.12　小结 …… 61

第 4 章　警告和条件编译 …… 62
4.1　PL/SQL 警告 …… 62
4.1.1　基础 …… 62
4.1.2　使用警告 …… 63
4.1.3　升级警告为错误 …… 67
4.1.4　忽略警告 …… 68
4.1.5　编译和警告 …… 69
4.1.6　关于警告的结束语 …… 72

4.2 条件编译……72
4.2.1 基础……72
4.2.2 正在运行代码的哪部分……75
4.2.3 预处理代码的好处……76
4.2.4 有效性验证……78
4.2.5 控制编译……80
4.2.6 查询变量……81
4.2.7 关于条件编译的结束语……82
4.3 小结……84

第 5 章 PL/SQL 单元测试……85
5.1 为什么要测试代码……85
5.2 什么是单元测试……86
5.2.1 调试还是测试……86
5.2.2 建立测试的时机……86
5.3 单元测试构建工具……87
5.3.1 utPLSQL：使用命令行代码……87
5.3.2 Quest Code Tester for Oracle……87
5.3.3 Oracle SQL Developer……88
5.4 准备和维护单元测试环境……88
5.4.1 创建单元测试资料档案库……89
5.4.2 维护单元测试资料档案库……90
5.4.3 导入测试……91
5.5 构建单元测试……91
5.5.1 使用单元测试向导……91
5.5.2 创建第一个测试实施……92
5.5.3 添加启动和拆除进程……93
5.5.4 收集代码覆盖率统计信息……93
5.5.5 指定参数……93
5.5.6 添加进程验证……94
5.5.7 保存测试……95
5.5.8 调试和运行测试……95
5.6 扩大测试的范围……95
5.6.1 创建查找值……96
5.6.2 植入测试实施……97
5.6.3 创建动态查询……98
5.7 支持单元测试功能……99
5.7.1 运行报告……99
5.7.2 创建组件库……100
5.7.3 导出、导入和同步测试……103
5.7.4 构建套件……105
5.8 从命令行运行测试……105
5.9 小结……106

第 6 章 批量 SQL 操作……107
6.1 五金商店……107
6.2 设置本章的例子……108
6.3 在 PL/SQL 中执行批量操作……108
6.3.1 批量获取入门……110
6.3.2 三种集合风格的数据类型……112
6.3.3 为什么要自找麻烦……114
6.3.4 监控批量收集的开销……116
6.3.5 重构代码以使用批量收集……119
6.4 批量绑定……127
6.4.1 批量绑定入门……127
6.4.2 度量批量绑定性能……128
6.4.3 监视内存的使用……131
6.4.4 11g 中的改进……133
6.5 批量绑定的错误处理……134
6.5.1 SAVE EXCEPTIONS 和分批操作……137
6.5.2 LOG ERRORS 子句……138
6.5.3 健壮的批量绑定……139
6.6 大规模集合的正当理由……143
6.7 真正的好处：客户端批量处理……145
6.8 小结……149

第 7 章 透识你的代码……151
7.1 本章内容取舍……152
7.2 自动代码分析……153
7.2.1 静态分析……154
7.2.2 动态分析……154
7.3 执行分析的时机……154
7.4 执行静态分析……156
7.4.1 数据字典……156
7.4.2 PL/Scope……163
7.5 执行动态分析……175
7.5.1 DBMS_PROFILER 和 DBMS_TRACE……175
7.5.2 DBMS_HPROF……184
7.6 小结……189

第 8 章 合同导向编程 ……190
8.1 契约式设计……190
8.1.1 软件合同 ……190
8.1.2 基本合同要素 ……191
8.1.3 断言……192
8.1.4 参考文献……192
8.2 实现 PL/SQL 合同……192
8.2.1 基本的 ASSERT 程序 ……192
8.2.2 标准的包本地断言……194
8.2.3 使用 ASSERT 执行合同 ……196
8.2.4 其他改进……198
8.2.5 合同导向函数原型……199
8.3 示例：测试奇数和偶数……200
8.4 有用的合同模式 ……202
8.4.1 用 NOT NULL 输入且输出 NOT NULL ……202
8.4.2 函数返回 NOT NULL ……203
8.4.3 布尔型函数返回 NOT NULL……203
8.4.4 检查函数：返回 TRUE 或 ASSERTFAIL……204
8.5 无错代码的原则 ……205
8.5.1 严格地断言先决条件……205
8.5.2 一丝不苟地模块化……206
8.5.3 采用基于函数的接口 ……206
8.5.4 在 ASSERTFAIL 处崩溃……207
8.5.5 对后置条件进行回归测试 ……207
8.5.6 避免在正确性和性能之间取舍……207
8.5.7 采用 Oracle 11g 优化编译 ……208
8.6 小结 ……209

第 9 章 从 SQL 调用 PL/SQL ……210
9.1 在 SQL 中使用 PL/SQL 函数的开销……210
9.1.1 上下文切换 ……211
9.1.2 执行……216
9.1.3 欠理想的数据访问……218
9.1.4 优化器的难点 ……222
9.1.5 读一致性陷阱 ……226
9.1.6 其他问题……228
9.2 降低 PL/SQL 函数的开销 ……228
9.2.1 大局观……229
9.2.2 使用 SQL 的替代品 ……230
9.2.3 减少执行……236
9.2.4 协助 CBO ……244
9.2.5 调优 PL/SQL ……255
9.3 小结……257

第 10 章 选择正确的游标 ……258
10.1 显式游标 ……258
10.1.1 解剖显式游标 ……260
10.1.2 显式游标和批量处理……261
10.1.3 REF 游标简介……262
10.2 隐式游标 ……263
10.2.1 解剖隐式游标 ……264
10.2.2 隐式游标和额外获取的理论 ……265
10.3 静态 REF 游标……267
10.3.1 详细的游标变量限制清单 ……269
10.3.2 客户端和 REF 游标 ……270
10.3.3 有关解析的话题 ……271
10.4 动态 REF 游标……273
10.4.1 例子和最佳用法 ……273
10.4.2 SQL 注入的威胁……275
10.4.3 描述 REF 游标中的列……276
10.5 小结……277

第 11 章 大规模 PL/SQL 编程 ……279
11.1 将数据库作为基于 PL/SQL 的应用服务器……279
11.1.1 案例研究：Avaloq 银行系统 ……279
11.1.2 在数据库中使用 PL/SQL 实现业务逻辑的优势……281
11.1.3 用数据库作为基于 PL/SQL 的应用程序服务器的限制 ……283
11.1.4 软因素……284
11.2 大规模编程的要求……284
11.3 通过规范实现一致性 ……285
11.3.1 缩写词……286
11.3.2 PL/SQL 标识符的前缀和后缀 ……289

11.4 代码和数据的模块化……291
11.4.1 包和相关的表作为模块……293
11.4.2 含有多个包或子模块的模块……297
11.4.3 模式作为模块……299
11.4.4 在模式内部模块化……303
11.4.5 用模式模块化与在模式内模块化的比较……306
11.5 使用 PL/SQL 面向对象编程……306
11.5.1 使用用户定义类型的面向对象编程……307
11.5.2 使用 PL/SQL 记录面向对象编程……310
11.5.3 评估……316
11.6 内存管理……317
11.6.1 测量内存使用……317
11.6.2 集合……322
11.7 小结……325

第 12 章 渐进式数据建模……326
12.1 从二十年的系统开发中总结的经验……327
12.2 数据库和敏捷开发……328
12.3 渐进式数据建模……329
12.4 重构数据库……331
12.5 通过 PL/SQL 创建访问层……335
12.6 敏捷宣言……347
12.7 用 PL/SQL 进行渐进式数据建模……349
12.7.1 定义接口……349
12.7.2 思考可扩展性……349
12.7.3 测试驱动开发……350
12.7.4 明智地使用模式和用户……350
12.8 小结……351

第 13 章 性能剖析……352
13.1 何谓性能……353
13.1.1 功能需求……353
13.1.2 响应时间……353
13.1.3 吞吐量……354
13.1.4 资源利用率……354
13.1.5 性能是功能的一种……355
13.2 什么是剖析……356
13.2.1 顺序图……356
13.2.2 概要文件之神奇……357
13.2.3 性能剖析的好处……357
13.3 性能测量……358
13.3.1 这个程序为什么慢……358
13.3.2 测量嵌入……360
13.3.3 识别……360
13.3.4 条件编译……364
13.3.5 内建的剖析器……365
13.3.6 扩展的 SQL 跟踪数据（事件 10046）……365
13.3.7 针对 Oracle 的测量工具库（ILO）……366
13.4 问题诊断……368
13.4.1 R 方法……369
13.4.2 ILO 示例……371
13.4.3 剖析示例……373
13.5 小结……376

第 14 章 编码规范和错误处理……378
14.1 为什么要制订编码规范……378
14.2 格式化……379
14.2.1 大小写……379
14.2.2 注释……380
14.2.3 比较……380
14.2.4 缩进……380
14.3 动态代码……383
14.4 包……384
14.5 存储过程……385
14.5.1 命名……385
14.5.2 参数……386
14.5.3 调用……386
14.5.4 局部变量……386
14.5.5 常量……386
14.5.6 类型……387
14.5.7 全局变量……387
14.5.8 本地存储过程和函数……387
14.5.9 存储过程元数据……388

14.6 函数 ····· 388
14.7 错误处理 ····· 389
14.7.1 错误捕获 ····· 389
14.7.2 错误报告 ····· 390
14.7.3 错误恢复 ····· 391
14.7.4 先测试再显示 ····· 392
14.8 小结 ····· 392
第 15 章 依赖关系和失效 ····· 395
15.1 依赖链 ····· 395
15.2 缩短依赖链 ····· 401
15.3 数据类型引用 ····· 406
15.4 用于表修改的视图 ····· 407
15.5 把组件添加到包 ····· 410
15.6 依赖链中的同义词 ····· 413
15.7 资源锁定 ····· 414
15.8 用触发器强制执行依赖 ····· 415
15.9 创建最初禁用的触发器 ····· 418
15.10 小结 ····· 420

第 1 章 避免误用

Riyaj Shamsudeen

感谢您购买本书。PL/SQL是你值得拥有的好工具，不过，你应当理解PL/SQL并不适用于所有场景。本章将讲述何时使用PL/SQL编写应用程序，怎样编写可扩展的代码，更重要的是，何时不使用PL/SQL编写代码。滥用某些PL/SQL的结构会导致不可扩展的代码。本章将讨论大量的案例，它们因误用PL/SQL而不可扩展。

PL/SQL 与 SQL

SQL是一门集合处理的语言,如果以集合级别的思维编写SQL语句,则会使其具有更好的可扩展性。PL/SQL是过程语言，而SQL语句可以嵌入到PL/SQL代码中。

SQL语句在SQL执行器（也就是通常说的SQL引擎）中执行。PL/SQL语句则由PL/SQL引擎执行。PL/SQL的威力在于能将PL/SQL的过程化数据处理能力和SQL的集合处理能力结合起来。

1.1 逐行处理

在一个典型的逐行处理程序中，代码首先打开一个游标，然后遍历从游标返回的行，并且处理每一行数据。这种基于循环的处理方式是非常不赞成使用的，它会导致不可扩展的代码。代码清单1-1显示了一个使用这种结构的例子。

代码清单1-1 逐行处理

```
DECLARE
  CURSOR c1 IS
    SELECT prod_id, cust_id, time_id, amount_sold
    FROM sales
    WHERE amount_sold > 100;
  c1_rec c1%rowtype;
  l_cust_first_name customers.cust_first_name%TYPE;
  l_cust_lasT_name customers.cust_last_name%TYPE;
BEGIN
  FOR c1_rec IN c1
```

```
  LOOP
    -- 查询客户的详细信息
    SELECT cust_first_name, cust_last_name
    INTO l_cust_first_name, l_cust_last_name
    FROM customers
    WHERE cust_id=c1_rec.cust_id;
    --
    -- 插入数据到目标表
    --
    INSERT INTO top_sales_customers①(
      prod_id, cust_id, time_id, cust_first_name, cust_last_name,amount_sold
      )
      VALUES
      (
        c1_rec.prod_id,
        c1_rec.cust_id,
        c1_rec.time_id,
        l_cust_first_name,
        l_cust_last_name,
        c1_rec.amount_sold
      );
  END LOOP;
  COMMIT;
END;
/

PL/SQL procedure successfully completed.

Elapsed: 00:00:10.93
```

在代码清单1-1中，程序声明了一个游标c1，然后用游标for循环隐式地打开了这个游标，对从游标c1取出的每一行，程序查询customers表，并把first_name和last_name的值填充到变量，随后插入一行数据到top_sales_customers表。

代码清单1-1的编程方法很有问题。即使在循环中调用的SQL语句是高度优化的，程序的执行还是会消耗大量时间。假设查询customers表的SQL语句消耗0.1秒，INSERT语句也消耗0.1秒，那么在循环中每次执行就要0.2秒。如果游标c1取出了100 000行，那么总时间就是100 000乘以0.2秒，即20 000秒，也就是大约5.5小时。很难去优化这个程序的结构。基于显而易见的理由，Tom Kyte把这种处理方式定义为慢之又慢的处理（slow-by-slow processing）。

注意　本章的例子采用SH模式（schema），这是Oracle公司提供的范例模式之一。要安装这个范例模式，Oracle提供了安装软件。可以从http://download.oracle.com/otn/solaris/oracle11g/R2/solaris.sparc64_11gR2_examples.zip下载Solaris平台11gR2版本的样例软件。软件解压缩后的目录中含有安装说明。Oracle网站也提供了其他平台和版本的范例软件的Zip压缩包。

代码清单1-1的代码还有一个固有的问题。从PL/SQL的循环中调用的SQL语句会反复在PL/SQL引擎和SQL引擎之间切换执行，这种两个环境之间的切换称作上下文切换。上下文切换

① 这里的top_sales_customers表需要有6个列。——译者注（以下如未做特殊说明的均为译者注）

增加了程序运行的时间，并增加不必要的CPU开销。你应当通过消除或减少这种两个环境之间的切换来减少上下文切换的次数。

一般应当禁止逐行处理，更好的编程实践是把代码清单1-1的程序转换成一个SQL语句。代码清单1-2重写了代码，完全避免了PL/SQL。

代码清单1-2 重写逐行处理的代码

```
--
-- 插入数据到目标表
--
INSERT
INTO top_sales_customers
  (
    prod_id,
    cust_id,
    time_id,
    cust_first_name,
    cust_last_name,
    amount_sold
  )
SELECT s.prod_id,
  s.cust_id,
  s.time_id,
  c.cust_first_name,
  c.cust_last_name,
  s.amount_sold
FROM sales s,
     customers c
WHERE s.cust_id    = c.cust_id and
s.amount_sold> 100;

135669 rows created.

Elapsed: 00:00:00.26
```

代码清单1-2除了解决逐行处理的缺陷以外，还有更多的优势。重写后的SQL语句可以使用并行执行来调优，使用多个并行执行进程可以大幅地减少执行时间。并且，代码变得简明且可读性强。

注意 如果将PL/SQL的循环代码重写为连接（join），需要考虑重复问题。如果在customers表中cust_id列有重复值，那么重写后的SQL语句取回的行比你想要的行多。不过，在这个特定的例子中，customers表的cust_id列上有主键，因此在cust_id列上使用等值连接不存在重复的危险。

1.2 嵌套的逐行处理

在PL/SQL语言中游标可以嵌套使用。首先将数据从一个游标中取出，然后将这些值传给另一个游标，即二级游标，再从二级游标取值传给三级游标，以此类推，这是很常见的编程方法。但是如果游标嵌套很深，这种基于循环的代码导致的性能问题就会加剧。由于游标嵌套，SQL执

行的次数大量增加，导致程序运行时间更长。

在代码清单1-3中，c1、c2和c3是嵌套游标。游标c1是顶级游标，从表t1取得数据，c2是开放游标，传递从游标c1取得的值，c3也是开放游标，传递游标c2取得的值。有一个UPDATE语句对游标c3返回的每一行执行一次。尽管UPDATE语句已经优化为执行一次只要0.01秒，但程序的性能还是会由于深度嵌套游标而难以忍受的。假设游标c1、c2和c3分别返回20、50和100行，那么上述代码需要循环100 000行，程序的总执行时间超过了1000秒。对这类程序的调优通常需要完全重写它。

代码清单1-3　用嵌套游标逐行处理

```
DECLARE
  CURSOR c1 IS
    SELECT n1 FROM t1;
  CURSOR c2 (p_n1) IS①
    SELECT n1, n2 FROM t2 WHERE n1=p_n1;
  CURSOR c3 (p_n1, p_n2) IS②
    SELECT text FROM t3 WHERE n1=p_n1 AND n2=p_n2;
BEGIN
  FOR c1_rec IN c1
  LOOP
    FOR c2_rec IN c2 (c1_rec.n1)
    LOOP
      FOR c3_rec IN c3(c2_rec.n1, c2_rec.n2)
      LOOP
        -- 在此执行一些sql语句③
        UPDATE … SET ..where n1=c3_rec.n1 AND n2=c3_rec.n2;
      EXCEPTION
      WHEN no_data_found THEN
        INSERT into… END;
      END LOOP;
    END LOOP;
  END LOOP;
 COMMIT;
END;
/
```

代码清单1-3中代码的另一个问题在于先执行一个UPDATE语句。如果UPDATE 语句产生了no_data_found异常④，那么再执行一个INSERT语句。这种类型的问题可以利用MERGE语句从PL/SQL转到SQL引擎处理。

从概念上讲，代码清单1-3中的三重循环表示表t1、t2和t3之间的等值连接。代码清单1-4展示了根据上述逻辑改写的使用表别名t的SQL语句。UPDATE和INSERT逻辑的结合用MERGE语句代替，MERGE语法提供了更新存在的行和插入不存在的行的功能。

---

① 这里缺少游标参数的数据类型，应改为：CURSOR c2 (p_n1 number) IS。

② 这里缺少游标参数的数据类型。

③ 需要一个SELECT 语句触发no_data_found异常，前1行需要一个begin，将异常处理放在其后。

④ 这个说法是错的，select 语句才能触发no_data_found异常，update语句只能用if sql%notfound 去判断是否更新成功，update语句不会触发该异常。

代码清单1-4 用MERGE语句重写逐行处理

```
MERGE INTO fact1 USING
(SELECT DISTINCT c3.n1,c3.n2 ①
 FROM t1, t2, t3
 WHERE t1.n1      = t2.n1
 AND t2.n1        = t3.n1
 AND t2.n2        = t3.n2
) t
ON (fact1.n1=t.n1 AND fact1.n2=t.n2)
WHEN matched THEN
  UPDATE SET .. WHEN NOT matched THEN
  INSERT .. ;
  COMMIT;
```

不要在PL/SQL语言中编写深度嵌套游标的代码。审查这类代码的逻辑，看是否能用SQL语句来代替。

1.3 查找式查询

查找式查询（lookup query）一般用于填充某些变量或执行数据的合法验证。但在循环中执行查找式查询会导致性能问题。

在代码清单1-5中，高亮显示的部分就是使用查找式查询来得到country_name值。程序对游标c1中的每一行都要执行一个查询来取得country_name的值。当从游标c1中取得的行数增加时，执行查找式查询的次数也增加了，这导致代码的效率低下。

代码清单1-5 查找式查询，对代码清单1-1的副本的修改

```
DECLARE
  CURSOR c1 IS
    SELECT prod_id, cust_id, time_id, amount_sold
    FROM sales
    WHERE amount_sold > 100;
  l_cust_first_name customers.cust_first_name%TYPE;
  l_cust_last_name customers.cust_last_name%TYPE;
  l_Country_id countries.country_id%TYPE;
  l_country_name countries.country_name%TYPE;
BEGIN
 FOR c1_rec IN c1
 LOOP
  -- 查询客户的详细信息
  SELECT cust_first_name, cust_last_name, country_id
  INTO l_cust_first_name, l_cust_last_name, l_country_id
  FROM customers
  WHERE cust_id=c1_rec.cust_id;

  -- 用来得到country_name的查询
  SELECT country_name
  INTO l_country_name
  FROM countries WHERE country_id=l_country_id;
  --
  -- 插入数据到目标表
```

① 这里表名有错，应为：t3.n1，t3.n2，按题目意思还缺一个要更新的值。

```
    --
    INSERT
    INTO top_sales_customers①
      (
        prod_id, cust_id, time_id, cust_first_name,
        cust_last_name, amount_sold, country_name
      )
      VALUES
      (
        c1_rec.prod_id, c1_rec.cust_id, c1_rec.time_id, l_cust_first_name,
        l_cust_last_name, c1_rec.amount_sold, l_country_name
      );
  END LOOP;
COMMIT;
END;
/
PL/SQL procedure successfully completed.
Elapsed: 00:00:16.18
```

代码清单1-5的代码是过分简化的，对country_name的查找式查询实际上可以重写为主游标c1本身中的一个连接。第一步，应将查找式查询修改为连接，可是在实际的应用程序中，并不一定可以实现这种改写。

如果无法利用改写代码来减少查找式查询的执行次数，那么还有另一个选择。你可以定义一个关联数组来缓存查找式查询的结果，以便在随后的执行中重用这个数组，这样也能有效地减少查找式查询的执行。

代码清单1-6演示了数组缓存技术。不必再在游标c1返回的每一行中执行查询来得到country_name，而是用一个名为l_country_names的关联数组来存储本例中的country_id和country_name键–值对。关联数组和索引类似，任意给定的值都可以通过一个键值来访问。

在执行查找式查询前，通过EXISTS操作对一个数组中是否存在一个匹配country_id键值的元素做一个存在性验证，如果数组中存在这么一个元素，那么country_name就从数组中获取而不需要执行查找式查询。如果没有这样的元素，那么就执行查找式查询，并且把查到的结果作为一个新元素存入数组。

你还需要理解，这种技术非常适用于不同的键值很少的语句，在本例中，当country_id列的唯一值个数越少时，查找式查询的执行次数可能也越少。如使用示例模式，执行查找式查询的次数最多是23，因为country_id列只有23个不同的值。

代码清单1-6　用关联数组实现的查找式查询

```
DECLARE
  CURSOR c1
  IS
    SELECT prod_id, cust_id, time_id, amount_sold
    FROM sales WHERE amount_sold > 100;
    l_country_names country_names_type;②
    l_Country_id countries.country_id%TYPE;
```

① 这里的top_sales_customers表需要有7个列。

② 这个变量定义是错误的，需要先定义类型，后边还有一处定义，因此需删除。

```
    l_country_name countries.country_name%TYPE;
    l_cust_first_name customers.cust_first_name%TYPE;
    l_cust_lasT_name customers.cust_last_name%TYPE;
   TYPE country_names_type IS
   TABLE OF VARCHAR2(40) INDEX BY pls_integer;
   l_country_names country_names_type;
BEGIN
  FOR c1_rec IN c1 LOOP
  -- 查询客户的详细信息
  SELECT cust_first_name, cust_last_name, country_id
  INTO l_cust_first_name, l_cust_last_name, l_country_id
  FROM customers
  WHERE cust_id=c1_rec.cust_id;
  -- 在执行SQL语句前检查数组

  IF ( l_country_names.EXISTS(l_country_id)) THEN
    l_country_name := l_country_names(l_country_id);
  ELSE
    SELECT country_name INTO l_country_name
    FROM countries
    WHERE country_id               = l_country_id;
  -- 保存到数组以便将来重用
    l_country_names(l_country_id) := l_country_name;
  END IF;
  --
  -- 插入数据到目标表
  --
  INSERT
  INTO top_sales_customers①
    (
      prod_id, cust_id, time_id, cust_first_name,
      cust_last_name, amount_sold, country_name
    )

    VALUES
    (
      c1_rec.prod_id, c1_rec.cust_id, c1_rec.time_id, l_cust_first_name,
      l_cust_last_name, c1_rec.amount_sold, l_country_name
    );
END LOOP;
COMMIT;
END;
/
PL/SQL procedure successfully completed.

Elapsed: 00:00:10.89
```

注意 关联数组所需内存是在数据库服务器中专用服务器进程的PGA（Program Global Area，程序全局区）中分配的，如果数千个连接都要把程序的中间结果缓存到数组中，那么内存的占用将会大幅增加。你应当掌握每个进程的内存使用增加情况，并设计数据库服务器以适应内存的增长。

① 这里的top_sales_customers表需要有7个列。

利用基于数组的技术也可在其他场景中消除不必要的工作。例如，可以通过把函数结果值存入关联数组来减少执行代价高昂的函数调用。1.5节讨论了减少执行次数的另一种技术。

注意　将函数结果存入一个关联数组仅适用于确定性函数。也就是说，这种函数对给定的输入集合，总是返回相同的输出。

1.4　对 DUAL 的过度访问

PL/SQL代码中对DUAL表的过度访问情况并不罕见。应当避免对DUAL表的过度访问。因为从PL/SQL中访问DUAL表会导致上下文切换，而这会损害性能。这一节总结了一些PL/SQL程序中过度访问DUAL表的常见原因，并讨论了减少这种访问的方案。

1.4.1　日期的算术运算

在PL/SQL中，没有理由通过访问DUAL表来执行算术运算或日期运算。因为大部分操作都能用PL/SQL语言本身的结构完成。甚至SYSDATE也能脱离开SQL引擎直接用PL/SQL访问。在代码清单1-7中加粗的SQL语句用于计算Unix新纪元时间（从1970年1月1日午夜到当前时间经历的秒数），使用了SELECT…from DUAL语法。虽然访问DUAL表很快，但执行这个语句仍然会导致SQL和PL/SQL引擎之间的上下文切换。

代码清单1-7　过度访问DUAL——算术运算

```
DECLARE
  l_epoch INTEGER;
BEGIN
  SELECT ((SYSDATE-TO_DATE('01-JAN-1970 00:00:00', 'DD-MON-YYYY HH24:MI:SS'))
         * 24 *60 *60 )
  INTO l_epoch
  FROM DUAL;
  dbms_output.put_line(l_epoch);
END;
```

可以使用如下的PL/SQL结构执行算术运算来避免对DUAL表的不必要访问。

```
l_epoch := (SYSDATE- TO_DATE('01-JAN-1970 00:00:00', 'DD-MON-YYYY HH24:MI:SS'))
         * 24 *60 *60 ;
```

完全不必调用SQL执行算术运算，简单地让PL/SQL做这项工作，代码会更易读，也更高效。

过度访问当前日期或时间戳是增加DUAL表访问的另一个原因。考虑在一个SQL语句中直接调用SYSDATE而不是先用SELECT语句将SYSDATE保存到一个变量，然后再把这个变量传递给SQL引擎。如果需要访问刚刚插入的列值，那么用returning子句来取出该列值就能实现。如果你需要在PL/SQL本身中访问SYSDATE，只需要用PL/SQL结构将当前日期取出并赋给一个变量即可。

1.4.2 访问序列

对DUAL表进行不必要的访问的另一种原因，是要从一个序列中获得下一个值。代码清单1-8显示的代码片段通过访问DUAL表从cust_id_seq[①]序列中取出下一个值，并赋给一个变量，然后把这个变量的值插入customers表[②]。

代码清单1-8 过度访问DUAL——序列

```
DECLARE
  l_cust_id NUMBER;
BEGIN
  FOR c1 in (SELECT cust_first_name, cust_last_name FROM customers
             WHERE cust_marital_status!='married')
  LOOP
    SELECT cust_hist_id_seq.nextval INTO l_cust_id FROM dual;
    INSERT INTO customers_hist
      (cust_hist_id, first_name, last_name )
    VALUES
      (l_cust_id, c1.cust_first_name, c1.cust_last_name)
      ;
  END LOOP;
END;
/
PL/SQL procedure successfully completed.

Elapsed: 00:00:01.89
```

更好的方法是不要将序列的值取出并存入变量，而是直接在INSERT语句本身中获得序列的值。下面的代码片段演示了INSERT语句怎么将序列产生的值插入customers表。用这种编码方法，就避免了对DUAL表的访问，因此也避免了引擎之间的上下文切换。

```
Insert into customers (cust_id, ...)
Values (cust_id_seq.nextval,...);
```

这个问题还有更好的解决办法：将整个PL/SQL块改写为一个SQL语句。改写后的语句0.2秒就执行完了，而基于PL/SQL循环处理的运行时间需要1.89秒。如下所示：

```
INSERT INTO customers_hist
SELECT
   cust_hist_id_seq.nextval, cust_first_name, cust_last_name
FROM customers
WHERE cust_marital_status!='married';

23819 rows created.
Elapsed: 00:00:00.20
```

1.4.3 填充主-从行

另一个过度访问DUAL表的原因，是要把数据插入具有主从关系的几个表中。这种编码方法的典型例子是从序列中得到主表的主键值并赋给一个局部变量，再把这个局部变量插入主表和从

① 从代码看是cust_hist_id_seq序列。

② 从代码看是customers_hist表。

表。这样做的原因是需要把主表的主键插入从表[①]。

Oracle DataBase 9*i*引入了一项SQL新功能，它允许从插入的行中返回插入值，从而为这类问题提供了更好的解决方案。你可以用DML RETURNING子句取得新插入主表行的键值，然后就能将这些键值插入从表。例如：

```
INSERT INTO customers (cust_id, ...)
VALUES (cust_id_seq.nextval,...)
RETURNING cust_id into l_cust_id;
...
INSERT INTO customer_transactions (cust_id, ...)
VALUES (l_cust_id,...)
...
```

1.5　过多的函数调用

认识到一个设计良好的应用程序会用到函数、过程和包很重要。但本节不是讨论应该怎么使用模块化的方法编程，而是讨论在编程中如何避免不必要的函数调用。

1.5.1　不必要的函数调用

函数调用通常意味着将指令集的不同部分要装载到CPU。执行时需要从指令的一个部分跳转到另一个部分。这种跳转执行增加了性能问题，因为这引起了指令管道的大量转储（dump）和重新填充，结果是加重了CPU的负担。

通过避免不必要的函数执行，可以避免不必要的指令管道刷新和重新填充，这样就使得对CPU的需求降到最低。再次声明，我并不反对模块化编程，而只反对过度和不必要的函数调用。最好用例子来解释。

在代码清单1-9中，log_entry是一个调试函数，在每次合法性验证中都会调用它。但这个函数本身有一个对v_debug的检查，并且只有在调试标记设为“真”时才将消息插入。假设有个程序的循环中包括执行数百个这样复杂的业务合法值验证，那么实际上，即使v_debug标志的值设为“假”，log_entry函数也会被不必要地调用数百万次。

代码清单1-9　不必要的函数调用

```
create table log_table ( message_seq number, message varchar2(512));
create sequence message_id_seq;

DECLARE
 l_debug BOOLEAN := FALSE;
 r1 integer;
 FUNCTION log_entry( v_message IN VARCHAR2, v_debug in boolean)
  RETURN number
  IS
  BEGIN
    IF(v_debug) THEN
      INSERT INTO log_table
        (message_seq, MESSAGE)
```

① 虽然每次从序列取出的nextval值都不同，但取出的currval值是相同的，后者可以用于insert从表。

```
        VALUES
        (message_id_seq.nextval, v_message
        );
    END IF;
    return 0;
  END;
BEGIN
 FOR c1 IN (
        SELECT s.prod_id, s.cust_id,s.time_id,
               c.cust_first_name, c.cust_last_name,
               s.amount_sold
        FROM sales s,
             customers c
        WHERE s.cust_id    = c.cust_id and
        s.amount_sold> 100)
  LOOP
    IF c1.cust_first_name IS NOT NULL THEN
      r1 := log_entry ('first_name is not null ', l_debug );
    END IF;
    IF c1.cust_last_name IS NOT NULL THEN
      r1 := log_entry ('Last_name is not null ', l_debug);
    END IF;
  END LOOP;
END;
/

PL/SQL procedure successfully completed.
Elapsed: 00:00:00.54
```

代码清单1-9中的代码可以改写为仅当标志变量l_debug设为“真”时才调用log_entry函数。这个改写减少了不必要的log_entry函数执行。改写后的程序在0. 43秒内执行完成。如果执行次数更多，那么性能的提高会更明显。

```
...
IF first_name IS NULL AND l_debug=TRUE THEN
  log_entry('first_name is null ');
END IF;
...
/
PL/SQL procedure successfully completed.
Elapsed: 00:00:00.43
```

如果还嫌不够好，可以考虑用条件编译[①]结构来完全避免代码片段的执行。在代码清单1-10中，加粗的代码使用$IF-$THEN结构和条件变量$debug_on。如果条件变量$debug_on是“真”，那么代码块得到执行。在一个生产环境中，变量 $debug_on会设为“假”，这就消除了函数执行。注意，程序的执行时间进一步减少到了0.34秒。

代码清单1-10 用条件编译来避免不必要的函数调用

```
DECLARE
 l_debug BOOLEAN := FALSE;
 r1 integer;
 FUNCTION log_entry( v_message IN VARCHAR2, v_debug in boolean)
  RETURN number
  IS
```

① 条件编译的详细用法见第4章。

```
  BEGIN
    IF(v_debug) THEN
      INSERT INTO log_table
        (message_seq, MESSAGE)
        VALUES
        (message_id_seq.nextval, v_message
        );
    END IF;
    return 0;
  END;
BEGIN
 FOR c1 IN (
       SELECT s.prod_id, s.cust_id,s.time_id,
              c.cust_first_name, c.cust_last_name,
              s.amount_sold
       FROM sales s,
            customers c
       WHERE s.cust_id    = c.cust_id and
       s.amount_sold> 100)
  LOOP
   $IF $$debug_on $THEN
    IF c1.cust_first_name IS NOT NULL  THEN
      r1 := log_entry ('first_name is not null ', l_debug );
    END IF;
    IF c1.cust_last_name IS NOT NULL   THEN
      r1 := log_entry ('Last_name is not null ', l_debug);
    END IF;
   $END
    null;
  END LOOP;
END;
/
PL/SQL procedure successfully completed.
Elapsed: 00:00:00.34
```

不必要的函数调用问题通常在从一个模板程序复制再修改的程序中频繁出现。观察程序，如果没必要调用某个函数，就不要调用它。

解释和本地编译的对比

PL/SQL代码默认是解释执行的。在编译PL/SQL的过程中，代码被转化为一种中间格式并存储在数据字典中。在执行时，引擎执行中间代码。

Oracle Database 9*i*引入了一项新功能：本地编译，即把PL/SQL代码编译成机器指令并作为共享库存储。过度函数调用对本地编译的影响可能更小，因为现代编译器能够将子程序展开而避免指令的跳转。

1.5.2　代价高昂的函数调用

如果某个函数需要几秒的运行时间，那么在循环中调用此函数将导致代码的性能低劣。你应当优化频繁执行的函数以使它尽可能有效率地运行。

在代码清单1-11中，如果函数calculate_epoch在一个循环中被调用数百万次。即使每次执行

此函数只耗费0.01秒，一百万次执行此函数也要耗费2.7小时。解决这个性能问题的办法之一是优化此函数使它执行一次只要几毫秒，不过这种深度的优化很难做到。

代码清单1-11 代价高昂的函数调用

```
CREATE OR REPLACE FUNCTION calculate_epoch (d in date)
 RETURN NUMBER DETERMINISTIC IS
 l_epoch number;
BEGIN
 l_epoch := (d - TO_DATE('01-JAN-1970 00:00:00', 'DD-MON-YYYY HH24:MI:SS'))
          * 24 *60 *60 ;
 RETURN l_epoch;
END calculate_epoch;
/

SELECT  /*+ cardinality (10) */ max( calculate_epoch (s.time_id))  epoch
FROM sales s
 WHERE s.amount_sold> 100 and
       calculate_epoch (s.time_id) between 1000000000 and 1100000000;

     EPOCH
----------
1009756800

Elapsed: 00:00:01.39
```

另一个可选方案是把函数执行的结果预存起来，这样就避免了在查询语句中执行函数。可以用一个基于函数的索引做到这一点。代码清单1-12创建了一个基于calculate_epoch函数的索引。SQL语句的性能从1.39秒提高到了0.06秒。

代码清单1-12 用基于函数的索引代替执行代价高昂的函数调用

```
CREATE INDEX compute_epoch_fbi ON sales
(calculate_epoch(time_id))
Parallel (degree 4);

SELECT  /*+ cardinality (10) */ max( calculate_epoch (s.time_id))  epoch
FROM sales s
 WHERE s.amount_sold> 100 and
       calculate_epoch (s.time_id) between 1000000000 and 1100000000;
     EPOCH
----------
1009756800

Elapsed: 00:00:00.06
```

你应该明白基于函数的索引也是有代价的。INSERT语句和更新time_id列的UPDATE语句会导致调用函数和维护索引的开销。需要认真地权衡DML操作中的函数执行开销和SELECT语句中的函数执行开销，并选择一个开销较低的方法。

注意 从Oracle Database11g开始，可以创建一个虚拟列，然后再在虚拟列上创建一个索引。虚拟列上索引的影响和基于函数的索引的影响是一样的。虚拟列优于基于函数的索引的优点是，你可以基于虚拟列对一个表进行分区，而这是基于函数的索引所做不到的。

另一种方法是利用从Oracle Database11g开始提供的result_cache来优化代价高昂的函数调用。这种功能会将函数调用的结果记忆在一个实例的SGA所分配的结果缓存中。重复用同样的参数执行函数会从结果缓存中取值，而不需要重复执行函数。代码清单1-13显示的是利用函数result_cache功能来提高性能的例子。执行同样的SQL语句用了0.81秒。

代码清单1-13　带结果缓存的函数

```
DROP INDEX compute_epoch_fbi;
CREATE OR REPLACE FUNCTION calculate_epoch (d in date)
 RETURN NUMBER  DETERMINISTIC RESULT_CACHE IS
 l_epoch number;
BEGIN
 l_epoch := (d - TO_DATE('01-JAN-1970 00:00:00', 'DD-MON-YYYY HH24:MI:SS'))
          * 24 *60 *60 ;
 RETURN l_epoch;
END calculate_epoch;
/

SELECT  /*+ cardinality (10) */ max( calculate_epoch (s.time_id))  epoch
FROM sales s
 WHERE s.amount_sold> 100 and
       calculate_epoch (s.time_id) between 1000000000 and 1100000000;

     EPOCH
----------
1009756800

Elapsed: 00:00:00.81
```

总之，过度的函数执行导致性能问题。如果无法减少或消除函数执行，那么可以利用基于函数的索引或result_cache作为暂时的解决办法，从而将函数调用带来的影响降到最低。

1.6　数据库链接调用

过度使用数据库链接调用会影响应用程序的性能。在一个循环中通过数据库链接来访问远程表或修改远程表是一个不可扩展的方法。因为对远程表的每一次访问，都会导致本地数据库和数据库链接中包含的远程数据库之间的多个SQL*Net包的交换。如果数据库在地理上处于不同的数据中心，或者更糟糕一些，数据库是跨国的，那么等待SQL*Net的流量就会导致程序的性能问题。

在代码清单1-14中，游标每返回一行，远程数据库上的customer表都要被访问一次。我们假定一次网络往返需要0.1秒，那么100万次往返调用将需要大约27小时才能完成。而位于国内不同地区的数据库之间的响应时间为0.1秒是很常见的。

代码清单1-14　过度使用数据库链接调用

```
DECLARE
  V_customer_name VARCHAR2(32);
BEGIN
  ...
  FOR c1 IN (SELECT …)
  LOOP
```

```
    ...
    SELECT customer_name
    INTO v_customer_name
    FROM customers@remotedb
    WHERE account_id = c1.account_id;
    ...
  END LOOP;
END;
```

慎重地使用实体化视图能减少程序执行中的网络往返。在代码清单1-14的案例中，可以为远程customer表创建一个本地实体化视图。在程序执行前先刷新实体化视图，然后在程序中访问实体化视图，将表在本地实体化减少了SQL*Net往返调用次数。当然，作为应用程序的设计者，你需要比较实体化整个表和在循环中访问远程表的代价，选择最理想的解决方案。

另一种选择是将程序改写为使用远程表连接的SQL语句。Oracle数据库的查询优化器能优化这种语句，以减少SQL*Net往返的开销，而要使用这种技术，就需要重写程序，使这个SQL语句只执行一次，而不是循环执行。

调优代码清单1-14中程序的首选步骤是将数据实体化到本地或将代码改写为包含远程连接的SQL语句。不过，如果你无法做到这些，还有另外的解决办法。作为一个临时措施，你可以将程序转化为能够使用多进程架构。例如进程1处理1 ~ 100 000范围内的客户，进程2处理100 001 ~ 200 000之间的客户，等等。把这种逻辑应用到示例程序，创建10个并行执行进程，你就可以将程序的总运行时间减少到大约2.7小时。另一种可采取的方法是使用DBMS_PARALLEL_EXECUTE，将代码分解成多个任务并行处理。

1.7 过度使用触发器

尽管也能用Java语言来写触发器代码，但触发器通常还是用PL/SQL编写。由于性能的原因，过度使用触发器是不理想的。数据库表中行的变化在SQL引擎中执行而触发器是在PL/SQL引擎中执行，如果触发器频繁触发，程序就会遭遇到十分严重的上下文切换问题。

在一些情况下，触发器是无法避免的。例如，复杂业务中的合法性验证就不可避免要用到触发器。在这些情况下，你应当在合法性验证很复杂时才用PL/SQL编写，而要避免滥用触发器进行简单的合法性验证。例如，用check约束而不是用一个触发器来验证列的值是否在合法值代码清单之内。

还有，应避免对一个触发事件使用多个触发器。不要为一个事件编写两个触发器，而应当把它们合并为一个，以使上下文切换的次数最小化。

1.8 过度提交

在PL/SQL程序循环中经常见到插入、修改或删除每行后面紧接着就是提交。这种在每行后面提交的编码方法会导致程序执行得更慢。频繁的提交产生更多的重做（redo），需要日志写入器更频繁地将日志缓冲区的内容保存到日志文件。过度提交还会导致数据一致性问题，并且消耗

更多的资源。虽然Oracle已经优化了PL/SQL引擎以减少频繁提交的影响，但这无法替代由于自身编写良好从而减少了提交的代码。

你应当只在一项业务交易完成后提交，如果在到达业务交易的边界前就提交了，就会遇到数据一致性问题。如果必须用提交来提升中断后可重新开始的能力，那么建议使用批量提交。比起每行之后提交，每1000或5000行批量提交一次更好。每批的数量选择取决于应用程序。更少的提交会使程序的运行时间缩短。而且，从应用程序发出更少的提交也能提高数据库的性能。

1.9　过度解析

不要在PL/SQL循环中使用动态SQL语句，因为这么做会引起过度解析问题。反之，应通过使用绑定变量来减少硬解析的次数。

在代码清单1-15中，从游标c1取得cust_id，然后根据cust_id访问customers表，来获取客户的详细信息。构造出一个包含字面值的SQL语句，然后用本地动态SQL的EXECUTE IMMEDIATE结构来执行。问题在于每从游标c1取出一个不重复的cust_id值，就会构造一个新的SQL语句，然后将它发送到SQL引擎去执行。

执行共享池中不存在的SQL语句，就会触发一次硬解析。过度硬解析会对库缓存（library cache）产生压力，因而降低了应用程序的可扩展性和并发执行能力。当游标c1返回的行数增加时，硬解析的次数会线性增长。这个程序也许可以在只有少量行需要处理的开发数据库中正常运行，但用这个方法在生产环境运行就会很成问题。

代码清单1-15　过度解析

```
DECLARE
  ...
BEGIN
 FOR c1_rec IN c1
 LOOP
  -- 查询客户的详细信息
  EXECUTE IMMEDIATE
   'SELECT cust_first_name, cust_last_name, country_id
    FROM customers
    WHERE cust_id= ' || c1_rec.cust_id  INTO l_cust_first_name, l_cust_last_name,↩
 l_country_id;
  ...
 END LOOP;
COMMIT;
END;
/
```

应当尽可能减少硬解析。在循环中执行动态SQL将增加硬解析的影响，如果并发执行增多，这种影响就会放大。

1.10　小结

本章介绍了多个不适合使用PL/SQL的场景。请牢记，SQL是一种集合语言，而PL/SQL是过

程性语言。在设计程序时，可以考虑下列建议。

- 用SQL解决查询问题。用集合的方式进行思考，用SQL写的查询，比起用嵌套循环逐行处理的方式执行查询的PL/SQL程序更加易于调优。
- 如果必须用PL/SQL来编写代码，请尝试尽可能将较多的工作负担交给SQL引擎。这种技术随着Exadata等新技术的出现而变得更重要。Exadata数据库机中的智能扫描设备能将工作负担交给存储节点，从而提高用SQL编写的程序性能。而PL/SQL结构不能从Exadata数据库机获得同样的好处，至少在11gR2中是不能的。
- 如果必须使用循环，那么使用PL/SQL提供的批量处理功能。利用本章介绍的技术减少在PL/SQL中的不必要工作，比如不必要的函数执行或者对DUAL表的过度访问。
- 只把使用单行、基于循环的处理作为最后的手段。

没错，用PL/SQL处理所有的数据和业务，用Java或其他语言来展现界面和进行用户合法性验证。利用本章描述的方法可以编写出高度可扩展的PL/SQL语言程序。

第2章

动态SQL：处理未知

2

Michael Rosenblum

在过去的十年里，我参加过美国各地很多的Oracle讨论会。我一而再再而三地听到演讲嘉宾谈论构建“更好、更快、更便宜”的系统，但同样是这些人，当和你面对面地讨论同样的问题时，就变得很不乐观了。经常被提到的大型IT项目的失败率达75%仍然是客观的事实，如果再加上那些“实际上失败了但仍声称成功”的案例（现实中没有人敢于承认失败），可以说我们现代的软件开发过程确实处于严重的危机之中。

在这一章里，假定我们生活在一个稍微理想的世界中，没有团体间的勾心斗角，系统架构师们明白自己正在干的事，开发者至少了解OTN的含义。即使在这样一个改善了的世界中，系统开发过程仍然存在一些不可避免的固有风险。

- 正确无误地描述一个要构建的系统的需求是很有挑战性的。
- 构建一个能够满足规定的所有需求的系统是有一定难度的。
- 构建一个在短时间内不需要很多修改的系统是非常困难的。
- 构建一个永不会过期的系统是不可能的。

就算最后一条可被认为是常识，但是前三条肯定会遭到我很多同事的强烈反对。然而，在现实中永远不会有100%完美无缺的分析，不会有100%完备的需求集，也不会有100%满足使用需求且不需要再升级的硬件，等等。在IT行业，我们需要接受的事实是：无论何时，我们必须面对意料之外的事情。

开发者的信条 *整个开发过程的焦点应当从我们知道的事物转移到我们不知道的事物。*

令人遗憾的是，很多事情开发者并不知道。

- 涉及什么元素？比如，系统需要一个季度报表机制，但没有季度汇总表。
- 需要怎么处理？DBA的梦魇：如果有大量的来自不同表的评判标准，怎么确保全局搜索筛选能够充分完成？
- 究竟能否处理？对每个限制，通常会有至少一个替代方法或“后门”。但如果在下一个发行版本或版本更新后，“后门”的位置变了怎么办？

幸好，有多个不同的途径来回答这些（和类似的）问题。本章将讨论一个称作动态SQL的功能，它能够帮助我们解决前面提到的难题，并且能够避免一些可能导致系统失败或过时的主要陷阱。

2.1 动态 SQL 的三种方式

动态SQL的概念是明白而直截了当的。这项功能允许你以文本方式构建你的代码（可以是SQL和PL/SQL），然后在运行时处理它，仅此而已。动态SQL提供了一个程序在执行时写另一个程序的能力。话说起来简单，不过理解这个功能引入的潜在含义和可能性是极其重要的。本章将就这些问题展开讨论。

注意 我说的“处理”，指的是在任何程序语言执行程序过程中都需要经过的完整事件链——解析/执行/获取（最后一步是可选的）。这个主题的详细讨论超出了本章的范围，但了解每一个步骤的基础知识对正确运用动态SQL很关键。

实现动态SQL有下述我们要讨论的不同方式：

- 本地动态SQL；
- 动态游标；
- DBMS_SQL包。

有不少在线文档和正式出版物都很好地解释了每种方式的语法。虽然本书的目的是展示最佳实践而不是提供一个参考指南，但强调一下每种动态SQL的关键点，有利于下一步的深入讨论。

技术注解1 虽然术语“动态SQL”被整个Oracle社区接受，但它不是100%完备的，因为它涵盖了SQL语句**和PL/SQL块**。但“动态SQL**和PL/SQL**”听起来不够简洁，所以“动态SQL”实际表示的就是“动态SQL和PL/SQL”。

技术注解2 在写这些内容时，Oracle 10*g*和11*g*两个版本大家都普遍在用。这里的例子是Oracle 10*g*兼容的，如果描述的特性只在11*g*中存在，这些例子将会明确注明。

2.1.1 本地动态SQL

所有动态SQL的实现中大约95%采用如下EXECUTE IMMEDIATE 命令变体中的一种：

```
DECLARE
   v_variable_tx VARCHAR2(<Length>)|CLOB;
BEGIN
   v_variable_tx:='whatever_you_want';
   EXECUTE IMMEDIATE v_variable_tx [additional options];
END;
```

或

```
BEGIN
   EXECUTE IMMEDIATE 'whatever_you_want' [additional options];
END;
```

或

```
BEGIN
   EXECUTE IMMEDIATE 'whatever_you_want1'||'whatever_you_want2'||… [additional options];
END;
```

直到（也包括）Oracle 10*g*，EXECUTE IMMEDIATE能处理的代码总长度（包括连接后的结果）是32 KB（更大的代码集要用DBMS_SQL包处理）。从Oracle 11*g*开始，可以使用CLOB作为输入参数。架构师可能会困惑，怎么有人试图动态处理超过32 KB的单条语句，但据我的经验，这种情况确实存在。

1. 本地动态SQL示例#1

具体的语法细节需要另外30页来解释，但既然本书是为有经验的用户写的，那就要求读者知道如何使用文档。因此我们不去深入PL/SQL编程的内部细节，而是用一个典型的例子来解释**为何**需要动态SQL。为此，假定有下面的需求。

- 期望系统有很多查找字段，这些字段和未来处理过程中会扩展的LOV（值的列表）查找表关联。不应该分别构建每个这样的LOV项，而应用一种集中的解决方案处理所有这种问题。
- 所有的LOV项应当遵从同样的格式，即两列（ID/DISPLAY），第一列是查找项，第二列是文字描述。

这些需求非常适合使用动态SQL来实现：都是在初始编码时不完全了解的运行时定义操作的重复模式。下列代码在这样的条件下是适用的。

```
CREATE TYPE lov_t IS OBJECT (id_nr NUMBER, display_tx VARCHAR2(256));

CREATE TYPE lov_tt AS TABLE OF lov_t;

CREATE FUNCTION f_getlov_tt  (
         i_table_tx    VARCHAR2,
         i_id_tx         VARCHAR2,
         i_display_tx VARCHAR2,
         i_order_nr   VARCHAR2,
         i_limit_nr   NUMBER:=100)
RETURN lov_tt
IS
    v_out_tt lov_tt := lov_tt();
    v_sql_tx VARCHAR2(32767);
BEGIN
    v_sql_tx:='SELECT lov_item_t '||
                    'FROM (SELECT lov_t(' ||
                                  dbms_assert.simple_sql_name(i_id_tx)||', '||
                      dbms_assert.simple_sql_name(i_display_tx)||') lov_item_t '||
                    ' FROM '||dbms_assert.simple_sql_name(i_table_tx)||
                       ' order by '||dbms_assert.simple_sql_name(i_order_nr)||
                ')' ||
             ' WHERE ROWNUM <= :limit';
    EXECUTE IMMEDIATE v_sql_tx BULK COLLECT INTO v_out_tt USING i_limit_nr;
    RETURN v_out_tt;
```

```
END;

SELECT * FROM TABLE(CAST(f_getlov_tt(:1,:2,:3,:4,:5) AS lov_tt))
```


这个例子包括了所有核心动态SQL语法元素。

- 用PL/SQL字符串变量表示要执行的代码。
- 使用绑定变量将额外的参数传递入语句或从语句中传出。绑定变量是逻辑占位符，且在执行步骤才与实际参数相链接。
- 绑定变量只能用于传值，它不是结构元素。这是表名和列名要用字符串拼接的方式加入到语句的原因。
- DBMS_ASSERT包有助于强制执行规则，防止代码注入（因为你必须使用字符串拼接），规则要求表名和列名必须是“简单SQL名称”（没有空格，没有分隔符等）。
- 由于SQL语句有输出，输出返回给类型匹配的PL/SQL变量（本地动态SQL允许用户自定义数据类型，包括集合）。

例子中的最后一条语句（SELECT语句中调用F_GETLOV_TT）是给予前端开发人员的。这是他们需要集成到应用程序中的唯一内容。其他的部分都可以让数据库开发人员处理，包括调优、授权、特别处理逻辑等。这些项目中的每一项都有一个修改的单点和控制的单点。这种“单点”的概念是最关键的功能之一，有助于减少未来的开发/调试/审计工作。

2. 本地动态SQL示例#2

示例 # 1针对正在构建新系统的开发者。示例 # 2针对现有系统的维护人员。

在删除基于函数的索引时，某些时候Oracle会改变对象的默认状态：所有引用了拥有这个索引的表的PL/SQL对象自动失效。可以想到，这种改变会给很多批处理例程造成严重破坏。因此，Oracle给出了一种设置跟踪事件的解决方法。该例程采取下述形式删除基于函数的索引：

```
CREATE PROCEDURE p_dropFBIndex (i_index_tx VARCHAR2) is
BEGIN
    EXECUTE IMMEDIATE
        'ALTER SESSION SET EVENTS ''10624 trace name context forever, level 12''';
    EXECUTE IMMEDIATE 'drop index '||i_index_tx;
    EXECUTE IMMEDIATE
        'ALTER SESSION SET EVENTS ''10624 trace name context off''';
END;
```

这个例子说明了动态SQL的另一个关键要素，即不是只限于动态运行DML，任何有效的PL/SQL匿名块或SQL语句，包括DDL或ALTER SESSION，都可以在运行时操作过程中执行。因此，DBA不再需要他们惯用的“用脚本生成脚本，然后再用生成的脚本生成其他脚本”的方法，因为所有这些事情，都可以直接在数据库中处理。

2.1.2 动态游标

动态游标的处理思路类似于执行动态SQL的思路。但是，动态游标不是运行一个完整的语句，而是打开一个游标，并将一个SELECT语句作为参数传递给它。例如：

```
DECLARE
    v_cur SYS_REFCURSOR;
    v_sql_tx VARCHAR2(<length>)|CLOB:='valid_SQL_query';
    v_rec ...<record type>;
        or

    v_tt ...<record collection>;
BEGIN
    OPEN v_cur FOR v_sql_tx [<additional parameters>];

    FETCH v_cur INTO v_rec;
          or
    FETCH v_cur BULK COLLECT INTO v_tt [<limit N>];

   CLOSE v_cur;
END;
```

据我以往的经验，有两种情况，动态游标可能会派上用场。例子如下。

1. 动态游标示例#1

第一种必须要使用动态游标处理的情况，是把很多引用游标（REF CURSOR）类型的变量作为应用程序的不同层之间的沟通机制的环境。可惜，大多数情况下，这些变量是从中间层创建的并且通常数据库端专家很少参与开发。我见过的许多案例是这样的，它们的业务用例是100%有效的，但是，解决一个直接的功能问题的代码却十分难以维护和调试，甚至对于安全性来说就是噩梦！

解决这个问题的正确方法，是将构建引用游标的过程下推到数据库，那样更容易控制所有创建的逻辑（尽管如此，正确关闭所有打开的游标仍需中间层开发人员把关，否则系统将出现“游标泄漏”）。

包装器函数的基本结构应该如下：

```
CREATE FUNCTION f_getRefCursor_REF (i_type_tx VARCHAR2)
RETURN SYS_REFCURSOR
IS
  v_out_ref SYS_REFCURSOR;
  v_sql_tx VARCHAR2(32767);
BEGIN
    if i_type_tx = 'A' then
        v_sql_tx:=<some code to build query A>;
    elsif i_type_tx = 'B' then
        v_sql_tx:=<some other code to build query B>;
  ...
    END IF;
    OPEN v_out_ref FOR v_sql_tx;
    RETURN v_out_ref;
END;
```

当任务不仅是构造和执行查询，而且要传递一些绑定变量到查询时，事情变得更有趣了。解决这个问题有几种不同的方法。其中最简单的办法是创建一个包，其中包括许多全局变量和相应返回这些变量的函数。获取正确结果的全过程包括设定所有适当的变量，并且立即在同一会话中调用F_GETREFCURSOR_REF函数。不过，上述方法并不完美，因为中间层经常用一种纯粹的无状态的方式和数据库对话。在这种情况下，是不可能用PL/SQL包变量的。

注意 关于中间层的无状态实现，指的是这样一个环境：每个数据库调用获得一个单独的数据库会话，即使是在作为同一个逻辑操作的上下文环境中。这个会话可能是从已有的连接池中选择的，也可能是直接连接数据库得到的。不过，在实际条件下，既然无法保证后一次调用能命中前一次调用的那个会话，应当认为所有会话级的资源在两次调用之间是丢失连接的。

但是，Oracle提供了足够的附加选项来克服这种限制，可以用对象集合或XMLType来解决，其中后者更灵活一些。当然，任何一种解决方法都需要对查询进行一些必要的修改（如下面的例子所示）。但得到的结果是一个100%抽象的查询构造器：

```
CREATE FUNCTION f_getRefCursor_ref
        (i_type_tx VARCHAR2:='EMP',
        i_param_xml XMLTYPE:= XMLTYPE(
                '<param col1_tx="DEPTNO" value1_tx="20" col2_tx="ENAME" value2_tx="KING"/>')
        )
RETURN SYS_REFCURSOR
IS
    v_out_ref SYS_REFCURSOR;
    v_sql_tx VARCHAR2(32767);
BEGIN
    IF i_type_tx = 'EMP' THEN
        SELECT
            'WITH param AS  ('||
            '    SELECT   TO_NUMBER(EXTRACTVALUE (in_xml, ''/param/@value1_tx'')) value1, '||
            '                       EXTRACTVALUE (in_xml, ''/param/@value2_tx'') value2 '||
            '    FROM (SELECT :1 in_xml FROM DUAL) '||
            '                                ) '||
            ' SELECT count(*)'||
            ' FROM scott.emp, '||
            '              param '||
            ' WHERE emp.'|| dbms_assert.simple_sql_name(
                                    EXTRACTVALUE (i_param_xml, '/param/@col1_tx')
                                                )||'=param.value1 '||
            'OR emp.'|| dbms_assert.simple_sql_name(
                                   EXTRACTVALUE (i_param_xml, '/param/@col2_tx')
                                    )||'=param.value2'
        INTO v_sql_tx  FROM DUAL;

    ELSIF i_type_tx = 'B' THEN
        v_sql_tx:='<queryB>';
  ...
    END IF;
    OPEN v_out_ref FOR v_sql_tx USING i_param_xml;
    RETURN v_out_ref;
END;
```

在这个例子中，单个XML变量的参数用一组值对表示，在每组中，第一个参数定义了应当被过滤的是哪些列（EMP.DEPTNO和EMP.ENAME），第二个参数是相对应的列要过滤的值（20和KING）。列（COL1_TX和COL2_TX）在查询构造时被计算，因为它们是结构元素。而取值是作为XMLType类型的一个输入参数传递给所创建的查询。显然，结果的语法看上去有点令人费解，但上述例子证明

了可以将任何属于数据库（即在数据库中）的系统元素输出。

2. 动态游标示例#2

如果我们承认EXECUTE IMMEDIATE的最大限制在于，它是一个单独的解析/执行/获取的完整序列，不可以在中间暂停或停止。那么，有效利用动态游标的另一种情况就变得不言而喻了。上述限制导致在通过BULK COLLECT获取多行时，总是存在一个严重风险，即试图将过多的元素装载到输出集合中。当然，这种风险可以通过增加WHERE ROWNUM<= :1条件来减轻，如最初的例子所示。但这种方法也不完美，因为它不允许从同一数据源持续读取，而动态游标可以解决这一问题。

对于这个例子，假定你需要编写一个模块。这个模块要求从有序表读取最前边的*N*个值，如果它还没有到达字母表的中部[①]则停止读取，否则继续读取直到表的结尾。

该解决方案如下所示。

```
CREATE FUNCTION f_getlov_tt  (
        i_table_tx    VARCHAR2,
        i_id_tx         VARCHAR2,
        i_display_tx VARCHAR2,
        i_order_nr   VARCHAR2,
        i_limit_nr     NUMBER:=50)
RETURN lov_tt
IS
    v_out1_tt lov_tt := lov_tt();
    v_out2_tt lov_tt := lov_tt();
    v_sql_tx VARCHAR2(32767);
    v_cur SYS_REFCURSOR;
BEGIN
    v_sql_tx:='SELECT lov_t(' ||
                            dbms_assert.simple_sql_name(i_id_tx)||','||
                dbms_assert.simple_sql_name(i_display_tx)||')'||
      ' FROM '||dbms_assert.simple_sql_name(i_table_tx)||
      ' ORDER BY '||dbms_assert.simple_sql_name(i_order_nr);

    OPEN v_cur FOR v_sql_tx;
    FETCH v_cur BULK COLLECT INTO v_out1_tt LIMIT i_limit_nr;
    IF v_out1_tt.count=i_limit_nr AND UPPER(v_out1_tt(i_limit_nr).display_tx)>'N' then
          FETCH v_cur BULK COLLECT INTO v_out2_tt;
          SELECT v_out1_tt MULTISET UNION v_out2_tt INTO v_out1_tt FROM DUAL;
    END IF;
    CLOSE v_cur;

    RETURN v_out1_tt;
END;
```

这个实现允许程序先获取第一组记录V_OUT1_TT，接着停下来，然后一边从已打开的游标中获取数据，一边应用逻辑规则：第二组记录V_OUT2_TT只有在满足定义的规则时才被填充，但它只需继续从同一查询获取数据即可填充，并无额外成本。这种“停止再继续”的方式体现了这个例子的价值。

① 即值转为大写后小于或等于字符'N'。

2.1.3 DBMS_SQL

DBMS_SQL内置包是动态SQL方法最早的典型工具。尽管传言说它可能会被弃用，但这个包随着RDBMS版本的不断更新还在持续地发展。显然，保留DBMS_SQL的一个良好理由是，它对运行时所定义语句的执行提供了最底层控制。你可以随心所欲地将SQL语句的解析、执行、获取、定义输出和处理绑定变量作为独立的命令来执行。当然，这样细化的代价是性能的降低（但Oracle在不断缩小这个差距）和复杂性的增加。通常，凭经验估计，除非你确实知道无法避免使用DBMS_SQL，否则不要使用它。

下面是一些非用DBMS_SQL不可的案例。

- 你需要在Oracle10*g*或更低的版本中打破EXECUTE IMMEDIATE的32 KB代码长度限制。
- 你有未知数目或类型的输入/输出参数（在所有版本中）。
- 你需要更精细地控制过程（所有版本，但在11*g*有更多的可能性）。

第一种情况是非常简单的。你要么超过了32 KB的限制，要么没有。不过，从开发的角度来看，后边两条更有趣。本节接下来要演示通过对系统逻辑的低级访问能做些什么。

在每个大量使用REF_CURSOR变量的环境中，迟早有人会提出以下问题：如何才能知道这些游标变量传送的确切查询是什么？直到Oracle11*g*出现前，只能从源代码或在DBA级别访问v$视图检查SGA获得。但从Oracle 11*g*开始，允许将DBMS_SQL游标和REF_CURSOR游标进行双向转换。这项功能产生了下列解决方案：

```
CREATE PROCEDURE p_explainCursor (io_ref_cur IN OUT SYS_REFCURSOR)
IS
    v_cur      INTEGER := dbms_sql.open_cursor;
    v_cols_nr NUMBER := 0;
    v_cols_tt dbms_sql.desc_tab;
BEGIN
    v_cur:=dbms_sql.to_cursor_number(io_ref_cur);
    dbms_sql.describe_columns (v_cur, v_cols_nr, v_cols_tt);
    FOR i IN 1 .. v_cols_nr LOOP
        dbms_output.put_line(v_cols_tt (i).col_name);
    END LOOP;
    io_ref_cur:=dbms_sql.to_refcursor(v_cur);
END;

SQL> DECLARE
  2    v_tx VARCHAR2(256):='SELECT * FROM dept';
  3    v_cur SYS_REFCURSOR;
  4  BEGIN
  5    OPEN v_cur FOR v_tx;
  6    p_explainCursor(v_cur);
  7    CLOSE v_cur;
  8  END;
  9  /
DEPTNO
DNAME
LOC
PL/SQL procedure successfully completed.
SQL>
```

本例利用过程DBMS_SQL.DESCRIBE_COLUMNS取得一个打开的游标，并分析它的结构，如果游标

是一个SQL查询打开的，输出的信息包括所输出的列数量、查询的列的名称和类型等。这使我们想起“处理未知”的概念。我们获取到一些未知的东西，然后通过采用适当的工具，将包含可能有用信息的原始材料转化为可用于解决现实业务需求的可靠数据。

2.2　动态思考的样例

在正式介绍完三种类型的动态SQL以后，现在是时候展示一个适合用动态思考的应用程序的“黄金标准”案例了。本节的例子来自实际的生产问题，它完美地演示了高级数据库开发人员要面对的概念级问题，对初学者来说，这个任务相当有挑战性。

- 在一个分层结构中包含大约100张表来描述一个人：顾客A有电话号码B，他的证明人是C，C的地址是D，等等。
- 顾客和相关的子数据有时需要被克隆。
- 所有表都有一个单列的人造主键，从一个共享的序列（OBJECT_SEQ）中产生。所有表之间通过外键关联。
- 数据模型经常变化，因此不允许硬编码。需求必须马上被处理，因此无法禁用约束或使用任何其他的数据转换解决方法。

这个案例的克隆过程包含什么内容？显然克隆根元素（顾客）需要一些硬编码，但其他所有信息都从依赖树上通过层次遍历得来。克隆过程无疑需要某种快速访问的存储（在本例是关联数组）来保存新旧ID对（用旧ID找到新ID）的信息，并且我们还需要用一个嵌套表数据类型来保存主键列表以便往下一级传递。因此，除了声明一个主过程之外，还创建了下面的类型：

```
CREATE PACKAGE clone_pkg
IS
    TYPE pair_tt IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    v_Pair_t pair_tt;

    PROCEDURE p_clone (in_table_tx VARCHAR2, in_pk_tx VARCHAR2, in_id NUMBER);
END;

CREATE TYPE id_tt IS TABLE OF NUMBER;
```

注意第二个类型（ID_TT）需要被创建为一个数据库对象，因为它要经常在SQL中使用。第一个类型（PAIR_TT）只需要在PL/SQL代码的环境中保存新值和旧值，因此，我在CLONE_PKG包中定义它，并且在创建包之后立即定义了一个这种类型的变量。

既然要试图确定可能的模式，首先就要忽略克隆顾客的根过程，因为它与其他事物不同。其次，要忽略向下的第一层（某个顾客的电话号码），因为它是个子事件（一个“父亲”的多个“孩子”）。我们从向下的第二层开始，描述如何克隆已确认电话号码的证明人（假定新电话号码已被创建）。逻辑上，操作流如下。

(1) 找到能确认现有电话号码的所有证明人。

(a) 现存的电话号码作为电话ID的集合传递（V_OLDPHONE_TT）。

(b) 所有找到的证明人装载到临时PL/SQL集合（V_ROWS_TT）。

(2) 处理每个找到过的证明人。

(a) 保存所有找到的证明人ID（V_PARENT_TT），以便在层次树中进一步使用。

(b) 从序列获取一个新ID，并把新/旧ID对记录在全局包变量中。

(c) 在临时集合（V_ROWS_TT）中，用新值代替主键（证明人ID）和外键（电话ID）。

(3) 遍历临时集合, 把新行插入到证明人表。

完成这些步骤的代码如下：

```
DECLARE
    TYPE rows_tt IS TABLE of REF%ROWTYPE;
    v_rows_tt rows_tt;
    v_new_id NUMBER;
    v_parent_tt id_tt:=id_tt();
BEGIN
    SELECT * BULK COLLECT INTO v_rows_tt
    FROM REF t WHERE PHONE_ID in
        (SELECT column_value FROM TABLE (CAST (v_oldPhone_tt AS id_tt)));

    FOR i IN v_rows_tt.first..v_rows_tt.last LOOP
      v_parent_tt.extend;
      v_parent_tt(v_parent_tt.last):=v_rows_tt(i).REF_ID;

      SELECT object_Seq.nextval INTO v_new_id FROM DUAL;
      clone_pkg.v_Pair_t(v_rows_tt(i).REF_ID):=v_new_id;

      v_rows_tt(i).REF_ID :=v_new_id;
      v_rows_tt(i).PHONE_ID:=clone_pkg.v_Pair_t(v_rows_tt(i).PHONE_ID);
    END LOOP;

    FORALL i IN v_rows_tt.first..v_rows_tt.last
      INSERT INTO REF VALUES v_rows_tt(i);
END;
```

有趣的是，如果你仔细检查创建的代码，就会明白它包括以下三类元素。

- 传入的父表主键的列表（V_OLDPHONE_TT）。
- 主逻辑过程。
- 功能标识符（functional identifier）定义要处理的对象（以粗体标记）。
  - 子表名（REF）。
  - 子表的主键列名（REF_ID）。
  - 子表的外键列名（PHONE_ID）。

我正在尝试构建一个具有结构参数（表名、列名）和数据参数（父ID）的模块，这是利用动态SQL的绝佳例子，因为它能处理这两种类型的参数。

既然层次结构的每级都能通过检查外键关系完全地表示，那么显然Oracle数据字典也可用于遍历“父子树”（这里假定系统中不存在循环依赖）。思路很简单：取得一个表名作为输入，然后返回它的“孩子”列表以及对应的“孩子”的主键列和指向“父亲”的外键列。虽然下面的代码不包含任何动态SQL，但用来说明已经足够。（同样也摘自CLONE_PKG）

```
TYPE list_rec IS RECORD
        (table_tx VARCHAR2(50), fk_tx VARCHAR2(50), pk_tx VARCHAR2(50));
TYPE list_rec_tt IS TABLE OF list_rec;

FUNCTION f_getChildrenRec (in_tablename_tx VARCHAR2)
RETURN list_rec_tt
IS
    v_out_tt list_rec_tt;
BEGIN
    SELECT fk_tab.table_name, fk_tab.column_name fk_tx, pk_tab.column_name pk_tx
    BULK COLLECT INTO v_Out_tt
    FROM
        (SELECT ucc.column_name, uc.table_name
         FROM user_cons_columns ucc,
              user_constraints uc
         WHERE ucc.constraint_name = uc.constraint_name
         AND   constraint_type = 'P') pk_tab,
         (SELECT ucc.column_name, uc.table_name
          FROM   user_cons_columns ucc,
                (SELECT constraint_name, table_name
                FROM user_constraints
                WHERE r_constraint_name = (SELECT constraint_name
                                           FROM user_constraints
                                           WHERE table_name = in_tablename_tx
                                           AND constraint_type = 'P'
                                           )
                    ) uc
         WHERE ucc.constraint_name = uc.constraint_name ) fk_tab
    WHERE pk_tab.table_name = fk_tab.table_name;
    RETURN v_out_tt;
END;
```

现在已具备所有必要的信息，可以构建通用处理模块了，它将会递归调用自己直到到达“父子链”的末端，如下面的代码所示。从此模块的定义可见，它取得要处理的“父亲”的主键ID集合和一个描述“父子链”的CLONE_PKG.LIST_REC类型对象作为输入。

```
PROCEDURE p_process (in_list_rec clone_pkg.list_rec, in_parent_list id_tt)
IS
    v_execute_tx VARCHAR2(32767);
BEGIN
    v_execute_tx:=
        'DECLARE '||
        '    TYPE rows_tt IS TABLE OF '||in_list_rec.table_tx||'%rowtype;'||
        '    v_rows_tt rows_tt;'||
        '    v_new_id number;'||
        '    v_list clone_pkg.list_rec_tt;'||
        '    v_parent_list id_tt:=id_tt();'||
        'BEGIN '||
        '    SELECT * BULK COLLECT INTO v_rows_tt '||
        '    FROM '||in_list_rec.table_tx||' t WHERE '||in_list_rec.fk_tx||
        '       IN  (SELECT column_value FROM TABLE (CAST (:1 as id_tt)));'||
        '    IF v_rows_tt.count()=0 THEN RETURN; END IF;'||
        '    FOR i IN v_rows_tt.first..v_rows_tt.last LOOP '||
        '       SELECT object_Seq.nextval INTO v_new_id FROM DUAL;'||
        '       v_parent_list.extend;'||
        '       v_parent_list(v_parent_list.last):=v_rows_tt(i).'||in_list_rec.pk_tx||';'||
        '       clone_pkg.v_Pair_t(v_rows_tt(i).'||in_list_rec.pk_tx||'):=v_new_id;'||
        '       v_rows_tt(i).'||in_list_rec.pk_tx||':=v_new_id;'||
```