知名 IT 图书作者杨中科的又一扛鼎力作

Chinapub 在线购买地址：http://www.china-pub.com/301651

当当网在线购买地址：http://product.dangdang.com/product.aspx?product_id=20368319

第一本专门为程序员编写的数据库图书《程序员的 SQL 金典》

- 将子查询、表连接、数据库语法差异等用通俗易懂、诙谐幽默的语言讲解出来
- 配合大量真实案例，学了就能用，在短时间内成为数据库开发高手

- 高度提取不同数据库的共同点，仔细分析不同点，并给出解决方案，同时学会 MSSQLServer、MYSQL、Oracle、DB2 数据库不再是梦
- 国内第一本讲解开窗函数实际应用的图书

轻举技术之“纲”，张合用之“目”，锻造 SQL 高可用性数据库应用指南从理论到实践，凝聚 SQL 主流数据库最前沿的技术要领。

本书特色：主要介绍 SQL 的语法规则及在实际开发中的应用，并且对 SQL 在 MySQL、MS SQL Server、Oracle 和 DB2 中的差异进行了分析；详细讲解数据库对增、删、改、查等 SQL 的支持并给出了相应的 SQL 应用案例；透彻分析函数、子查询、表连接、不同 DBMS 中的 SQL 语法差异、SQL 调优、NULL 值处理、事务、开窗函数等高级技术；通过对实际案例开发过程的详细分析，使读者掌握 SQL 的综合应用技巧。

内容提要

本书主要介绍 SQL 以及在实际开发中的应用，并且对 SQL 在 MYSQL、MSSQLServer、Oracle 和 DB2 中的差异性进行了分析。本书分为三部分：第一部分为基础篇，主要讲解数据库对增删改查等 SQL 的支持，给出了这些 SQL 的应用案例；第二部分为进阶篇，讲解了函数、子查询、表联接、不同 DBMS 中 SQL 语法差异、SQL 调优、NULL 值处理、事务、开窗函数等高级技术；第三部分为案例篇，讲解了前两部分的知识的综合运用。此书适用于学习数据库编程的读者，对于有一定数据库开发经验的读者也有非常大的参考价值。

前言

市面上讲解数现在据库的书都花了很多篇幅讲解数据库的备份、授权、调优、修复、监控等内容，这些内容是数据库管理员（DBA）应该掌握的，而对于程序员来说更需要去掌握的则是 SQL 语句的使用。但是市面上专门讲解 SQL 语句的书非常少，初学者学习数据库开发过程中常常被那些写给 DBA 的书弄的晕头转向，无法真正快速的掌握 SQL 技术；而且这些书中讲解的 SQL 也常常是针对特定数据库系统的专有实现，无法很容易的在其他数据库系统中运行，读者需要阅读大量的书和查阅大量的文档才能掌握不同数据库系统的使用。

本书是专门写给程序员的，因此没有讲解备份、授权、调优、修复、监控等开发人员不关心的内容，直接从 SQL 语句入手让读者快速的掌握数据库开发的技能。“面向开发人员，讲解对开发人员最有用的知识”是本书的编写宗旨。

MYSQL、MSSQLServer、Oracle 和 DB2 等都是非常流行的数据库管理系统（DBMS），虽然在大部分 SQL 语法上这些 DBMS 实现都是一致的，不过在实现细节以及高级语法方面这些 DBMS 的实现差异还是非常大的。如果编写能够兼容这些 DBMS 的 SQL 语句是开发人员经常需要面对的问题。本书将帮助读者从根本上解决这个问题。

很多开发人员对于 SQL 语句的掌握只限于简单的 SELECT、UPDATE 语句，对于稍微复杂的逻辑经常需要编写程序代码来完成，这不仅无法发挥数据库的优势，而且开发出的系统性能非常低，而如果能够使用数据库函数、子查询、表联接、开窗函数等高级的 SQL 特性则可以大大简化系统开发的难度，并且提高系统的性能。本书将对这些高级特性进行详细的讲解。

本书第 1、2 章介绍数据库系统的基本知识以及基本操作；第 3 章介绍 Insert、Delete 和 Update 语句的基本应用；第 4 章对 Select 语句进行全面的介绍，对模糊匹配、分组、限制数据条数、计算字段、组合查询等高级内容进行了重点的讲解；第 5 章介绍常用的数据库函数以及它们在主流 DBMS 中的实现差异；第 6 章介绍索引与约束等知识点；第 7、8 章分别介绍表连接、子查询等高级查询技术；第 9 章对主流 DBMS 的语法差异进行分析，并且给出了解决方案；第 10 章介绍注入漏洞攻击、SQL 调优、事务、自动增长字段、NULL 值处理、开窗函数等高级话题；第 11 章以一个案例讲解书中知识点在实际开发中的应用。

在此，我要感谢为这本书的诞生给于我帮助的所有人。首先我要感谢 CowNew 开源团队的朋友们一直以来的无私奉献；感谢 KingChou 在开发 CowNewSQL 过程中的卓越工作，没有 CowNewSQL 也就不会有这本书的问世；还要感谢 EasyJF 的蔡世友，他一直以来对开源事业的奉献是值得我学习的；最后我要感谢电子工业出版社的田小康经理，他的高效工作使得本书能够顺利的完成和出版。

如果您对我的书有任何意见和建议，您可以给我发送邮件：about521@163.com，本书相关的后续资料将会发布到 CowNew 开源团队网站（http://www.cownew.com ）中。

第一章 数据库入门

本章介绍数据库的入门知识，首先介绍什么是数据库，然后介绍数据库中的一些基本概念，接着介绍操纵数据库的不同方式，最后介绍操纵数据库时使用的语言 SQL，在章节中我们还将穿插一些非常有趣的话题。

1.1 数据库概述

广义上来讲，数据库就是“数据的仓库”，计算机系统经常用来处理各种各样大量的数据，比如使用计算机系统收集一个地区的人口信息、检索符合某些条件的当地人口信息、当一个人去世后还要从系统中删除此人的相关信息。我们可以自定义一个文件格式，然后把人口数据按照这个格式保存到文件中，当需要对已经存入的数据进行检索或者修改的时候就重新读取这个文件然后进行相关操作。这种数据处理方式存在很多问题，比如需要开发人员熟悉操作磁盘文件的函数、开发人员必须编写复杂的搜寻算法才能快速的把数据从文件中检索出来、当数据格式发生变化的时候要编写复杂的文件格式升级程序、很难控制并发修改。

在计算机系统在各个行业开始普遍应用以后，计算机专家也遇到了同样的问题，因此他们提出了数据库理论，从而大大简化了开发信息系统的难度。数据库理论的鼻祖是 Charles W.Bachman，他也因此获得了 1973 年的图灵奖。IBM 的 Ted Codd 则首先提出了关系数据库理论，并在 IBM 研究机构开发原型，这个项目就是 R 系统，并且使用 SQL 做为存取数据表的语言，R 系统对后来的 Oracle、Ingres 和 DB2 等关系型数据库系统都产生了非常重要的影响。

1.1.1 “数据库”与“数据库管理系统”

前面我们讲到数据库就是“数据的仓库”，我们还需要一套系统来帮助我们管理这些数据，比如帮助我们查询到我们需要的数据、帮我们将过时的数据删除，这样的系统我们称之为数据库管理系统（Database Management System，DBMS）。有时候很多人也将 DBMS 简称为“数据库”，但是一定要区分“数据库”的这两个不同的意思。

数据库管理系统是一种操纵和管理数据库的系统软件，是用于建立、使用和维护数据库。

它对数据库进行统一的管理和控制，以保证数据库的安全性和完整性。用户通过 DBMS 访问数据库中的数据，数据库管理员也通过 DBMS 进行数据库的维护工作。它提供多种功能，可使多个应用程序和用户用不同的方法在同时或不同时刻去建立，修改和询问数据库。它使用户能方便地定义和操纵数据，维护数据的安全性和完整性，以及进行多用户下的并发控制和恢复数据库。通俗的说，DBMS 就是数据库的大管家，需要维护什么数据、查找什么数据的话找它告诉他了，它会帮你办的干净利落。

1.1.2 数据库能做什么

数据库能够帮助你储存、组织和检索数据。数据库以一定的逻辑方式组织数据，当我们要对数据进行增删改查的时候数据库能非常快速的完成所要求的操作；同时数据库隐藏了数据的组织形式，我们只要对数据的属性进行描述就可以了，当我们要对数据库中的数据进行操作的时候只要告诉“做什么”（What to do）就可以了，DBMS 会决定一个比较好的完成操作的方式，也就是我们无需关心“怎么做”（How to do），这样我们就能从数据存储的底层中脱身出来，把更多精力投入到业务系统的开发中。

数据库允许我们创建规则，以确保在增加、更新以及删除数据的时候保证数据的一致性；数据库允许我们指定非常复杂的数据过滤机制，这样无论业务规则多么复杂，我们都能轻松应对；数据库可以处理多用户并发修改问题；数据库提供了操作的事务性机制，这样可以保证业务数据的万无一失。

1.1.3 主流数据库管理系统介绍

目前有许多 DBMS 产品，如 DB2、Oracle、MicrosoftSQL Server、Sybase SQLServer、Informix、MySQL 等，它们在数据库市场上各自占有一席之地。下面简要介绍几种常用的数据库管理系统。

（1）DB2

DB2 第一种使用使用 SQL 的数据库产品。DB2 于 1982 年首次发布，现在已经可以用在许多操作系统平台上，它除了可以运行在 OS/390 和 VM 等大型机操作系统以及中等规模的 AS/400 系统之外，IBM 还提供了跨平台（包括基于 UNIX 的 LINUX，HP-UX，Sun Solaris，以及 SCO UnixWare；还有用于个人电脑的 Windows 2000 系统）的 DB2 产品。应用程序可以通过使用微软的 ODBC 接口、Java 的 JDBC 接口或者 CORBA 接口代理来访问 DB2 数据库。

DB2 有不同的版本，比如 DB2 Everyplace 是为移动用户提供的一个内存占用小且性能出色的版本；DB2 for z/OS 则是为主机系统提供的版本；Enterprise Server Edition(ESE) 是一种适用于中型和大型企业的版本； Workgroup Server Edition(WSE) 主要适用于小型和中型企业，它提供除大型机连接之外的所有 ESE 特性；而 DB2 Express 则是为开发人员提供的可以免费使用的版本。

IBM 是最早进行关系数据库理论研究和产品开发的公司，在关系数据库理论方面一直走在业界的前列，所以 DB2 的功能和性能都是非常优秀的，不过对开发人员的要求也比其他数据库系统更高，使用不当很容易造成宕机、死锁等问题；DB2 在 SQL 的扩展方面比较保守，很多其他数据库系统支持的 SQL 扩展特性在 DB2 上都无法使用；同时 DB2 对数据的类型要求也非常严格，在数据类型不匹配的时候会报错而不是进行类型转换，而且如果发生精度溢出、数据超长等问题的时候也会直接报错，这虽然保证了数据的正确性，但是也使得基于 DB2 的开发更加麻烦。因此，很多开发人员称 DB2 为“最难用的数据库系统”。

（2）Oracle

Oracle 是和 DB2 同时期发展起来的数据库产品，也是第二个采用 SQL 的数据库产品。Oracle 从 DB2 等产品中吸取到了很多优点，同时又避免了 IBM 的官僚体制与过度学术化，大胆的引进了许多新的理论与特性，所以 Oracle 无论是功能、性能还是可用性都是非常好的。

（3）Microsoft SQL Server

Microsoft SQL Server 是微软推出的一款数据库产品。细心的读者也许已经发现我们前面提到了另外一个名字非常相似的 Sybase SQLServer，这里的名字相似并不是一种巧合，这还要从 Microsoft SQL Server 的发展史谈起。

微软当初要进军图形化操作系统，所以就开始和 IBM“合作”开发 OS/2，最终当然无疾而终，但是微软就很快的推出了自己的新一代视窗操作系统；而当微软发现数据库系统这块新的市场的时候，微软没有自己重头开发一个数据库系统，而是找到了 Sybase 来“合作”开发基于 OS/2 的数据产品，当然微软达到目的以后就立即停止和 Sybase 的合作了，于 1995 年推出了自己的 Microsoft SQL Server 6.0，经过几年的发展终于在 1998 年推出了轰动一时的 Microsoft SQL Server 7.0，也正是这一个版本使得微软在数据库产品领域有了一席之地。正因为这段“合作”历史，所以使得 Microsoft SQL Server 和 Sybase SQLServer 在很多地方非常类似，比如底层采用的 TDS 协议、支持的语法扩展、函数等等。

微软在 2000 年推出了 Microsoft SQL Server 2000，这个版本继续稳固了 Microsoft SQL Server 的市场地位，由于 Windows 操作系统在个人计算机领域的普及，Microsoft SQL Server 理所当然的成为了很多数据库开发人员的接触的第一个而且有可能也是唯一一个数据库产品，很多人甚至在“SQL Server”和“数据库”之间划上了等号，而且用“SQL”一次来专指 Microsoft SQL Server，可见微软的市场普及做的还是非常好的。做足足够的市场以后，微软在 2005 年“审时度势”的推出了 Microsoft SQL Server 2005，并将于 2008 年发布新一代的 Microsoft SQL Server 2008。

Microsoft SQL Server 的可用性做的非常好，提供了很多了外围工具来帮助用户对数据库进行管理，用户甚至无需直接执行任何 SQL 语句就可以完成数据库的创建、数据表的创建、数据的备份/恢复等工作；Microsoft SQL Server 的开发者社区也是非常庞大的，因此有众多可以参考的学习资料，学习成本非常低，这是其他数据库产品做不具有的优势；同时从 Microsoft SQL Server 2005 开始开发人员可以使用任何支持.Net 的语言来编写存储过程，这进一步降低了 Microsoft SQL Server 的使用门槛。

不过正如微软产品的一贯风格，Microsoft SQL Server 的劣势也是非常明显的：只能运行于 Windows 操作系统，因此我们无法在 Linux、Unix 上运行它；不管微软给出什么样的测试数据，在实际使用中 Microsoft SQL Server 在大数据量和大交易量的环境中的表现都是不尽人意的，当企业的业务量到达一个水平后就要考虑升级到 Oracle 或者 DB2 了。

（4）MySQL

MySQL 是一个小型关系型数据库管理系统，开发者为瑞典 MySQL AB 公司。目前 MySQL 被广泛地应用在中小型系统中，特别是在网络应用中用户群更多。MySQL 没有提供一些中小型系统中很少使用的功能，所以 MySQL 的资源占用非常小，更加易于安装、使用和管理。

由于 MySQL 是开源的，所以在 PHP 和 Java 开发人员心中更是首选的数据库开发搭档，目前 Internet 上流行的网站构架方式是 LAMP（Linux+Apache+MySQL+PHP），即使用 Linux 作为操作系统，Apache 作为 Web 服务器，MySQL 作为数据库，PHP 作为服务器端脚本解释器。

MySQL 目前还很难用于支撑大业务量的系统，所以目前 MySQL 大部分还是用来运行非核心业务；同时由于 MySQL 在国内没有足够的技术支持力量，所以对 MySQL 的技术支持工作是由 ISV 或者系统集成商来承担，这也导致部分客户对 MySQL 比较抵制，他们更倾向于使用有更强技术支持力量的数据库产品。

1.2 数据库基础概念

要想使用数据库，我们必须熟悉一些基本概念，这些概念包括：Catalog、表、列、数据类型、记录、主键以及表关联等等。

1.2.1 Catalog

数据库就是数据的仓库，而 DBMS 是数据库的“管理员”。一些企业即生产食品又生产农用物资，这些产品都要保存到仓库中，同时企业内部也有一些办公用品需要保存到仓库中。如果这些物品都保存到同一个仓库中的话会造成下面的问题：

- 不便于管理。食品的保存和复印纸的保存需要的保存条件是不同的，食品需要低温保鲜而复印纸则需要除湿，不同类的物品放在一起加大了管理的难度；
- 可能会造成货位冲突。食品要防止阳光直射造成的变质，因此要摆放到背阴面，同时为了防止受潮，也要把它们摆放到高处；办公用胶片也要避免阳光直射，所以同样要摆放到背阴面，而且胶片也要防潮，所以同样要把它们摆放到高处。这就造成两种货物占据的货位相冲突了。
- 会有安全问题。由于所有物品都放到一个仓库中没有进行隔离，所以来仓库领取办公用品的人员可能会顺手牵羊将食品偷偷带出仓库。

既然都是“仓库”，那么数据库系统也存在类似问题。如果企业将人力资源数据和核心业务数据都保存到一个数据库中同样会造成下面的问题：

- 不便于管理。为了防止数据丢失，企业需要对数据进行定期备份，不过和核心业务数据比起来人力资源数据的重要性要稍差，所以人力资源数据只要一个月备份一次就可以了，而核心业务数据则需要每天都备份。如果将这两种数据保存在一个数据库中会给备份工作带来麻烦。
- 可能会造成命名冲突。比如人力资源数据中需要将保存员工数据的表命名为 Persons，而核心业务数据也要将保存客户数据的表也命名为 Persons，这就会相冲突了。
- 会有数据安全问题。由于所有的数据都保存在一个数据库中，这样人力资源系统的用户也可以访问核心业务系统中的数据，很容易造成数据安全问题。

显而易见，对于上边提到的多种物品保存在一个仓库中的问题，最好的解决策略就是使用多个仓库，食品保存在食品仓库中，农用物资保存在农用物资仓库中，而办公用品则保存在办公用品仓库中，这样就可以解决问题了。问了解决同样的问题，DBMS 也采用了多数据库的方式来保存不同类别的数据，一个 DBMS 可以管理多个数据库，我们将人力资源数据保存在 HR 数据库中，而将核心业务数据保存在 BIZ 数据库中，我们将这些不同数据库叫做 Catalog（在有的 DBMS 中也称为 Database，即数据库）。采用多 Catalog 以后可以给我们带来如下好处：

- 便于对各个 Catalog 进行个性化管理。DBMS 都允许我们指定将不同的 Catalog 保存在不同的磁盘上，由于人力资源数据相对次要一些，因此我们可以将 HR 保存在普通硬盘上，而将 BIZ 保存在 RAID 硬盘上。我们还可以对每个 Catalog 所能占据的最大磁盘空间、日志大小甚至优先级进行指定，这样就可以针对不同的业务数据进行个性化定制了。
- 避免了命名冲突。同一个 Catalog 中的表名是不允许重复的，而不同 Catalog 中的表名则是可以重复的，这样 HR 中可以有 Persons 表，而 BIZ 中也可以有 Persons 表，二者结构可以完全不相同，保存的数据也不会互相干扰。
- 安全性更高。DBMS 允许为不同的 Catalog 指定不同的用户，并且可以限定用户能访问的 Catalog。比如用户 hr123 只能访问 HR，而用户 sales001 只能访问 BIZ。这就大大加强了系统数据的安全性。

1.2.2 表（Table）

虽然我们已经将不同用途的物品保存在不同的仓库中了，但是在同一个仓库中数据的保存仍然存在问题。比如食品分为熟食、生肉、大米等，如果把他们随意的堆放在一起，就会造成我们无法很容易的对这些食品进行管理，当要对大米进行提货的话就必须在一堆的食品

中翻来翻去。解决这个问题的方法就是将仓库划分为不同的区域，熟食保存在熟食区，生肉保存在生肉区，而大米则保存在大米区。

DBMS 中也存在类似的问题，虽然我们将核心业务数据保存在 BIZ 数据库中了，但是核心业务数据也有很多不同类型的数据，比如客户资料、商品资料、销售员资料等，如果将这些数据混杂在一起的话将会管理起来非常麻烦，比如我们要查询所有客户资料的话就必须将所有数据查询一遍。解决这个问题的方法就是将不同类型的资料放到不同的“区域”中，我们将这种区域叫做“表”（Table）。客户资料保存到 Customers 表中，将商品资料保存在 Goods 表中，而将销售员资料保存在 SalesMen 表中，这样当需要查找商品的时候只要到 Goods 表中查找就可以了。

1.2.3 列（Column）

同样是生肉，不同的生肉又有不同的特性，有的生肉是里脊肉，有的生肉是前臀尖，这块生肉是 18 公斤，而那块生肉是 12 公斤，这块生肉是 12.2 元/公斤，而那块生肉是 13.6 元/公斤。每块肉都有各自的不同的特性，这些特性包括取肉部位、重量、单价。如果不对每块肉标注这些特性数据的话，当提货人要我们将所有里脊肉出库的话我们就非常麻烦了。解决这个问题的方法就是制作一些标签，在这个标签上标明取肉部位、重量、单价，这样要提取货物就会非常方便了。

不仅如此，标签的格式也要统一，如果第一块生肉的标签内容是：

这块肉是 15.6 公斤的里脊肉，13.2 元/公斤。

另一块生肉的标签内容是：

每市斤 8.6 元，前臀尖，13.6 公斤的，。

采用这种标签由于没有统一的格式，所以阅读起来非常麻烦，要靠人工去分辨，错误率非常高。如果我们规定一个统一的标签格式，比如下面的标签：

取肉部位	
重量	
单价（元/公斤）	

这样每块肉的标签就可以按照这个格式来填写了：

取肉部位	里脊肉
重量	15.6
单价（元/公斤）	13.2

这种格式阅读起来非常方便，如果引入自动识别设备的话，甚至可以实现自动化的物品分拣。

在数据库的表中保存的数据也有类似问题，如果不规定格式的话，表中的数据也会非常阅读，如果一个员工的资料在表中保存的内容为：

2003 年 5 月入职，是产品开发部的，姓名马小虎。

另外一个员工的资料在表中保存的内容为：

王二小，技术支持部，入职是 2005 年 7 月。

通常，以这种不标准的格式保存造成数据十分混乱，想要从数据库中取出合适的数据仍然非常麻烦。为了解决这个问题，我们规定下面这种标准的格式：

姓名	
部门	
入职时间	

这里的“姓名”、“部门”和“入职时间”就被称为员工表的列（Column），有时候也叫做字段（Field），每个列描述了数据的一个特性。

1.2.4 数据类型（DataType）

上面我们为员工表规定了“姓名”、“部门”和“入职时间”三个列，这样只要按照这个格式进行数据填写就可以了，但是这里仍然有一个问题，那就是我们没法限定用户向表中填写什么数据，比如用户填写成下面的格式：

姓名	33
部门	12.3
入职时间	信息中心

显然姓名不应该为一个数字 33；不可能有一个名称为“12.3”的部门；入职时间更不可能是“信息中心”。因此我们必须规则每一列中填写的数据的格式：姓名必须填写汉字，最短 2 个汉字，最长 5 个汉字；部门必须填写“产品开发部”、“技术支持部”、“产品实施部”、“人力资源部”中的一个；入职时间必须填写为正确的时间格式。

这里就规定了各个列的数据类型（DataType），数据类型规定了一个列中能填写什么类型的数据，减少了不规范数据出现的几率。

除了可以对数据进行规范之外，数据类型还有下面的作用：

- 提高效率。对不同的数据赋予不同的类型能够使得数据库更好的对数据进行存储和管理，从而减少空间占用并且提供数据的访问速度。比如，如果将数字 123454321 以文本类型存储的话将会占用 9 字节的存储空间，而以整数类型保存的话将只需要占用 4 字节的存储空间。
- 能够确定对数据进行操作所需要的正确处理方式。比如如果是整数类型，那么 123+234 被解释为两个整数的加法运算，所以其结果是 357；如果是文本类型，那么 123+234 则会被解释为两个字符串的相连操作，所以其结果是 123234。

1.2.5 记录（Record）

记录有可以被称为行（Row），可以通俗的认为它是数据表中的一行数据。以员工表为例，一个公司的员工表中的数据是这样的：

姓名	部门	入职时间
马小虎	产品开发部	2003 年 5 月 22 日
王二小	技术支持部	2005 年 7 月 17 日
白展堂	后勤部	1998 年 3 月 27 日
钱长贵	销售部	2001 年 3 月 3 日
李达最	后勤部	2005 年 11 月 11 日

这里每一行数据就代表一个员工的资料，这样的一行数据就叫做一条记录。表是由行和列组成的一张二维表，这就是关系数据库中最基本的数据模型。

1.2.6 主键（PrimaryKey）

员工表中的每一行记录代表了一个员工，一般员工的名字就能唯一标识这一个员工，但是名字也是有可能重复的，这时我们就要为每一名员工分配一个唯一的工号：

工号	姓名	部门	入职时间
001	马小虎	产品开发部	2003 年 5 月 22 日
002	王二小	技术支持部	2005 年 7 月 17 日
003	白展堂	后勤部	1998 年 3 月 27 日
004	钱长贵	销售部	2001 年 3 月 3 日
005	李达最	后勤部	2005 年 11 月 11 日
006	王二小	产品开发部	2005 年 3 月 22 日

这样就可以通过这个工号来唯一标识一名员工了。当老板下令说“把王二小提升为副总”的时候，我们就要问“公司有两个王二小，您要提升哪一个？”，老板可以说“技术支持部的王二小”，但是更好的方式，那就是说“提升工号为的 002 员工为副总”，因为只有 002 这个工号才能唯一标识一名员工。这里的“工号”被称为员工表的“主键”（PrimaryKey），所以我们可以说能唯一标识一行记录的字段就是此表的主键。

有的公司比较懒惰，不想为员工分配工号，只是硬性规定：一个部门中员工的姓名不能重复，有姓名重复的必须调换到其它部门。这样“部门”和“姓名”这两个字段加在一起就能唯一标识一名员工了，这里的“部门”和“姓名”两个字段就被称为“复合主键”，也就是任何一个字段都不能唯一标识一行数据，只有构成“复合主键”的所有字段组合起来才能唯一标识这一行数据。

在大多数 DBMS 中并没有强制规定一个表必须有主键，也就是一个表可以没有主键，但是为一个数据表指定一个主键是一个非常好的习惯。在后边的章节我们将提到用一个无意义的字段做主键将会更加有利于系统的可扩展性。

1.2.7 索引（Index）

无索引的表就是一个无序的行集。比如下面的人员表中有一些数据：

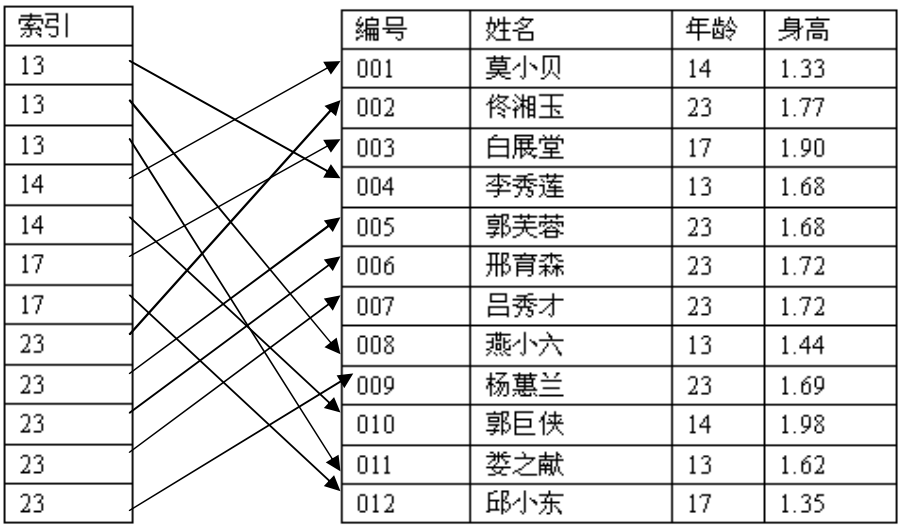
编号	姓名	年龄	身高
001	莫小贝	14	1.33
002	佟湘玉	23	1.77
003	白展堂	17	1.90
004	李秀莲	13	1.68
005	郭芙蓉	23	1.68
006	邢育森	23	1.72
007	吕秀才	23	1.72
008	燕小六	13	1.44
009	杨蕙兰	23	1.69

010	郭巨侠	14	1.98
011	娄之献	13	1.62
012	邱小东	17	1.35

这个表上没有索引，因此如果我们查找年龄等于 17 的人员时，必须查看表中的每一行，看它是否与所需的值匹配。这是一个全表扫描，很慢，如果表中只有少数几个记录与搜索条件相匹配，则其效率是相当低的。

如果我们经常要查询某个年龄的人员，必须查看表中的每一行，看它是否与所需的值匹配。这是一个全表扫描，很慢，如果表中只有少数几个记录与搜索条件相匹配，则其效率是相当低的。

如果我们为年龄列创建一个索引，注意这里的索引所采用的值是排序的：



索引
13
13
13
14
14
17
17
23
23
23
23
23

编号	姓名	年龄	身高
001	莫小贝	14	1.33
002	佟湘玉	23	1.77
003	白展堂	17	1.90
004	李秀莲	13	1.68
005	郭芙蓉	23	1.68
006	邢育森	23	1.72
007	吕秀才	23	1.72
008	燕小六	13	1.44
009	杨蕙兰	23	1.69
010	郭巨侠	14	1.98
011	娄之献	13	1.62
012	邱小东	17	1.35

假如我们要查找年龄为 13 岁的所有人员，那么可以扫描索引，结果得出前 3 行，当到达年龄为 14 的行的时候，我们发现它是一个比我们正在查找的年龄要大的人员。索引值是排序的，因此在读到包含 14 的记录时，我们知道不会再有匹配的记录，可以退出了。如果查找一个值，它在索引表中某个中间点以前不会出现，那么也有找到其第一个匹配索引项的定位算法，而不用进行表的顺序扫描（如二分查找法）。这样，可以快速定位到第一个匹配的值，以节省大量搜索时间。

可以把索引想像成汉语字典的的按笔画查找的目录。汉语字典中的汉字是按拼音的顺序排列在书页中的，如果要查询笔画数为 18 的汉字的话就必须挨个查找每个汉字来比较每个汉字的笔画数，这种速度是让人无法忍受的。而如果我们创建一个按笔画查找的目录：将笔画为 5 的汉字列出来，将笔画为 6 的汉字列出来……，这样当我们要查询笔画数为 18 的汉字的话只要来查找这个目录就可以非常快速的查找到需要的数据了。

虽然索引可以提高数据查询的速度，但是任何事物都是双刃剑，它也有一些缺点：
索引占据一定磁盘空间，就像有按笔画查找的目录的书会比没有这种目录的书页数要多一些。

索引减慢了数据插入和删除的速度。因为每次插入和删除的时候都需要更新索引，一个表拥有的索引越多，则写操作的平均性能下降就越大。

1.2.8 表关联

我们来为货物建一张表，其中包含规格、名称、生产厂家等等信息，如下：

编号	名称	规格	生产厂家	厂家地址	厂家电话
001	生肉	优质	七侠镇肉联厂	西凉河路 3 号	5555-123456
002	玉米肠	简装	七侠镇肉联厂	西凉河路 3 号	5555-123456
003	尿素	60 公斤装	六扇门化工厂	汉中工业区	5555-654321
004	打印纸	16 开	钱氏纸业	县政府对过	5555-123654
005	磷酸二铵	30 公斤装	六扇门化工厂	汉中工业区	5555-654321

可以看到这里存在大量冗余信息，比如厂家的名称、地址、电话等就在表中重复多次，这会带来如下的问题：

- 信息冗余占据空间。数据的存储是占据一定的空间的，如果存在过多冗余信息将会使得存储系统的利用率过低。
- 信息冗余使得新数据的加入变得麻烦。每次录入新的货物的话必须把厂家地址、厂家电话等信息重新录入一次。
- 信息冗余使得维护数据的正确性变得困难。如果七侠镇肉联厂迁址了，那么必须将表中所有七侠镇肉联厂的厂家地址都要更新一遍。

解决的方法就是即将厂家的信息在一个新的表中维护。我们创建下边的厂家表：

厂家编号	厂家名称	厂家地址	厂家电话
001	七侠镇肉联厂	西凉河路 3 号	5555-123456
002	六扇门化工厂	汉中工业区	5555-654321
003	钱氏纸业	县政府对过	5555-123654

这里我们为每个厂家指定了一个厂家编号做为主键，这个编号就可以唯一标识一个厂家。

有了厂家信息表，货物表就可以修改成如下的新的格式了：

编号	名称	规格	生产厂家编号
001	生肉	优质	001
002	玉米肠	简装	001
003	尿素	60 公斤装	002
004	打印纸	16 开	003
005	磷酸二铵	30 公斤装	002

在货物表中只保留了指向厂家表的主键的字段“生产厂家编号”，这样就避免了数据冗余的问题。当进行查询的时候，只要根据“生产厂家编号”到厂家信息表中查询就可以知道厂家的详细信息了；当厂家迁址的时候，只要修改厂家信息表中的一条数据就可以了。

这种将两张表通过字段关联起来的方式就被称为“表关联”，关联到其他表主键的字段被称为“外键”，上边例子中货物表中的“生产厂家编号”字段就是外键。表关联也是关系数据库的核心理念，它使得数据库中的数据不再互相孤立，通过表关联我们可以表达非常复杂的数据关系。

1.2.9 数据库的语言——SQL

DBMS 是一种系统软件，我们要与它交互的时候就必须使用某种语言，在数据库发展初期每一种 DBMS 都有自己的特有的语言，不过逐渐的 SQL 成为了所有 DBMS 都支持的主流语言。SQL 是专为数据库而建立的操作命令集，是一种功能齐全的数据库语言。在使用它时，只需要发出“做什么”的命令，“怎么做”是不用使用者考虑的。SQL 功能强大、简单易学、使

用方便，已经成为了数据库操作的基础，并且现在几乎所有的数据库均支持 SQL。

SQL 的英文全称是 Structured Query Language，它是 1974 年由 Boyce 和 Chamberlin 提出的，并且首先在 IBM 的关系数据库原型产品 R 系统（SYSTEM R）上实现。它的前身是 1972 提出的 SQUARE（Specifying Queries As Relational Expressesion ）语言，在 1974 年做了修改，并且改名为 SEQUEL（Structured English Query Language）语言，后来 SEQUEL 简化为 SQL。

SQL 是高级的非过程化编程语言，允许用户在高层数据结构上工作。使用它，用户无需指定对数据的存放方法，也不需要用户了解具体的数据存放方式，所以具有完全不同底层结构的不同数据库系统可以使用相同的 SQL 语言作为数据输入与管理的接口。它以记录集合作为操纵对象，所有 SQL 语句接受集合作为输入，返回集合作为输出，这种集合特性允许一条 SQL 语句的输出作为另一条 SQL 语句的输入，所以 SQL 语言可以嵌套，这使它具有极大的灵活性和强大的功能，在多数情况下，在其他语言中需要一大段程序实现的一个单独事件只需要一个 SQL 语句就可以达到目的，这也意味着用 SQL 语言可以写出非常复杂的语句。

SQL 具有下面 4 个主要的功能：创建数据库并定义表的结构；查询需要的数据；更新或者删除指定的数据；控制数据库的安全。使用 SQL 我们可以完成和 DBMS 的几乎所有交互任务。

比如我们要查找年龄小于 18 岁的员工信息，那么我们只要执行下面的 SQL 就可以：

```
SELECT * from Employees where age<18
```

比如我们要将所有职位为“名誉总裁”的员工删除，那么就可以执行下面的 SQL：

```
DELETE from Employees where position=’ 名誉总裁’
```

可以看到我们只是描述了我们要做什么，至于怎么去做则由 DBMS 来决定。可以想想如果要是自己编程去实现类似的功能，则需要编写非常复杂的算法才能完成，而且性能也不一定会非常好。

我们可以通过三种方式执行 SQL：

- 在工具中执行。各个 DBMS 几乎都提供了工具用于执行 SQL 语句，比如 Microsoft SQL Server 的 Management Studio、DB2 的命令中心、Oracle 的 SqlPlus 或者 MySQL 的 Query Browser。在这些工具中我们只要输入要执行的 SQL 然后点击【执行】按钮就可以得到执行结果。
- 以编译的方式嵌入到语言中。在这种方式中我们可以把 SQL 直接写到代码中，在编译的时候由编译器来决定和数据库的交互方式。比如 PowerBuild、C 等就采用这种方式。
- 以字符串的形式嵌入到语言中。在这种方式中 SQL 语句只是以字符串的形式写到代码中，然后由代码将其提交到 DBMS，并且分析返回的结果。目前这是大部分支持数据库操作的语言采用的方式，比如 C#、Java、Python、Delphi 和 VB 等。

由于嵌入到语言中的执行方式是严重依赖宿主语言的，而本书不假定用户使用任何编程语言，为了能够使得使用任何语言的读者都能学习本书中的知识点，本书将主要以在工具中执行的方式来执行 SQL 语句，读者可以根据自己使用的编程语言来灵活运用这些知识点。不熟悉用工具执行 SQL 的读者可以参考附录 A 中的介绍。

IBM 是 SQL 语言的发明者，但是其他的数据库厂商都在 IBM 的 SQL 基础上提出了自己的扩展语法，因此形成了不同的 SQL 语法，对于开发人员来说，使用这些有差异的语法是非常头疼的时候。因此在 1986 年美国国家标准化协会（ANSI）为 SQL 制定了标准，并且在 1987 年国际标准化组织（ISO）也为 SQL 指定了标准，迄今为止已经推出 SQL-86、SQL-89、SQL-92、SQL-99、SQL-2003 等版本的标准。

虽然已经有了国际标准，但是由于种种原因，各个数据库产品的 SQL 语法仍然有着很大差异，在数据库 A 上能成功执行的 SQL 放到数据库 B 上就会执行失败。为了方便使用不

同数据库产品的读者都能成功运行本书中的例子，我们会介绍各种数据库 SQL 的差异性，并且给出解决方案，而且本书将会安排专门章节讲解跨数据库程序开发的技术。

1.2.10 DBA 与程序员

如果你是一个数据库开发技术的初学者的话，你会发现到了书店里有很多数据库相关的书你看不懂，你会发现互联网有一些搞数据库的人的 Blog 上说的东西你感觉很陌生，他们都是在谈论数据库的恢复、数据库的调优、调整数据库的安全性，难道他们搞的是更深层次的东西吗？不是的，他们就是数据库系统管理员（Database Administrator，DBA）。围绕在 DBMS 周围的技术人员有两类：数据库系统管理员和开发人员。使用数据库进行程序开发的人员是程序员，而对数据库系统进行管理、维护、调优的则是数据库系统管理员。

作为一名开发人员，我们不必知道如何安装和配置数据库系统，这应该是 DBA 的任务；当规划数据库的备份策略的时候，不要去问开发人员，这也是 DBA 的任务；当数据库系统崩溃的时候，请立即给 DBA 打电话，如果打给开发人员的话，你得到的回答通常是“怎么会呢？天知道怎么恢复!”。正如一个公司的网络系统是由网管来负责的一样，一个公司的数据库系统也是由 DBA 来进行管理的，它们的主要工作如下：

- 安装和配置数据库，创建数据库以及帐户；
- 监视数据库系统，保证数据库不宕机；
- 收集系统统计和性能信息以便进行调整；
- 发现性能糟糕的 SQL，并给开发人员提出调优建议；
- 管理数据库安全性；
- 备份数据库，当发生故障时要及时恢复；
- 升级 DBMS 并且在必要时为系统安装补丁；
- 执行存储和物理设计，均衡设计问题以完成性能优化；

DBA 大部分时间是在监视系统、备份/恢复系统、优化系统，而开发人员则无需精通这些技能；开发人员大部分时间是在用 SQL 实现业务逻辑。二者知识的重合点就是 SQL，一个开发人员如果不熟悉 SQL 的话就无法很好的实现业务逻辑，而一个 DBA 如果不熟悉 SQL 的话就无法完成数据库的调优工作。所以无论你是想成为开发人员还是成为 DBA，那么都首先来学好 SQL 吧!

进行数据库的备份/恢复、权限管理等操作也经常需要使用 SQL 命令来完成，不过这些 SQL 命令都是与特定的 DBMS 产品相关的，而且不同产品的使用方式也是差别很大的，所以本书不会讲解数据库的备份/恢复、权限管理相关的 SQL，有兴趣的读者可以去参考相关的资料。

第三章 数据的增删改

上一章中介绍了创建和管理数据表的方法，数据表只是数据的容器，没有任何数据的表是没有任何意义的。主流的数据库系统都提供了管理数据库的工具，使用这些工具可以查看表中的数据，还可以添加、修改和删除表中的数据，但是使用工具进行数据的增删改通常只限于测试数据库时使用，更常见的方式时通过程序或者 Web 页面来向数据库发出 SQL 语句指令来进行这些操作，因此本章将介绍通过 SQL 语句增删改表中数据的方法。

本章中我们将使用一些数据表，为了更容易的运行本章中的例子，必须首先创建所需要的数据表，因此下面列出本章中要用到数据表的创建 SQL 语句：

MYSQL：

```
CREATE TABLE T_Person (FName VARCHAR(20),FAge INT,FRemark VARCHAR(20),PRIMARY KEY (FName));
```

```
CREATE TABLE T_Debt (FNumber VARCHAR(20),FAmount DECIMAL(10,2) NOT NULL,
    FPerson VARCHAR(20),PRIMARY KEY (FNumber),
    FOREIGN KEY (FPerson) REFERENCES T_Person(FName)) ;
```

MSSQLServer：

```
CREATE TABLE T_Person (FName VARCHAR(20),FAge INT,FRemark VARCHAR(20),PRIMARY KEY (FName));
CREATE TABLE T_Debt (FNumber VARCHAR(20),FAmount NUMERIC(10,2) NOT NULL,
    FPerson VARCHAR(20),PRIMARY KEY (FNumber),
    FOREIGN KEY (FPerson) REFERENCES T_Person(FName)) ;
```

Oracle：

```
CREATE TABLE T_Person (FName VARCHAR2(20),FAge NUMBER (10) ,FRemark VARCHAR2(20),PRIMARY KEY (FName)) ;
CREATE TABLE T_Debt (FNumber VARCHAR2(20),FAmount NUMERIC(10,2) NOT NULL,
    FPerson VARCHAR2(20),PRIMARY KEY (FNumber),
    FOREIGN KEY (FPerson) REFERENCES T_Person(FName)) ;
```

DB2：

```
CREATE TABLE T_Person (FName VARCHAR(20) NOT NULL,FAge INT,FRemark VARCHAR(20),PRIMARY KEY (FName));
CREATE TABLE T_Debt (FNumber VARCHAR(20) NOT NULL,FAmount DECIMAL(10,2) NOT NULL,
    FPerson VARCHAR(20),PRIMARY KEY (FNumber),
    FOREIGN KEY (FPerson) REFERENCES T_Person(FName)) ;
```

请在不同的数据库系统中运行相应的 SQL 语句。T_Person 为记录人员信息的数据表，其中主键字段 FName 为人员姓名，FAge 为年龄，而 FRemark 则为备注信息；T_Debt 记录了债务信息，其中主键字段 FNumber 为债务编号，FAmount 为欠债金额，FPerson 字段为欠债人姓名，FPerson 字段与 T_Person 中的 FName 字段建立了外键关联关系。

3.1 数据的插入

数据表是数据的容器，没有任何数据的数据表是没有意义的，数据表创建完成以后比如向其中插入有用的数据才能使得系统运转起来。

3.1.1 简单的 INSERT 语句

INSERT INTO 语句用来向数据表中插入数据，比如执行下面的语句就可以向 T_Person 表中插入一条数据：

```
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('Tom',18,'USA')
```
[1]

这句 SQL 向 T_Person 表中插入了一条数据，其中 FName 字段的值为'Tom'，FAge 字段的值为 18，而 FRemark 字段的值为'USA'。VALUES 前边的括号中列出的是要设置字段的字段名，字段名之间用逗号隔开；VALUES 后边的括号中列出的是要设置字段的值，各个值同样用逗号隔开。需要注意的是 VALUES 前列出的字段名和 VALUES 后边列出的字段值是按顺序一一对应的，也就是第一个值'Tom'设置的是字段 FName 的值，第二个值 18 设置的是字段 FAge 的值，第三个值'USA'设置的是字段 FRemark 的值，不能打乱它们之间的对应关系，而且要保证两边的条数是一致的。由于 FName 和 FRemark 字段是字符串类型的，所以需要用单引号[2]将值包围起来，而整数类型的 FAge 字段的值则不需要用单引号包围起来。

---

[1] 需要注意，这里的单引号是半角字符，如果使用全角字符将会导致执行错误。

[2]有的数据库系统也支持用双引号来包围，不过为了使得我们编写的 SQL 更容易的在主流数据库系统中运行，本书将一律采用单引号来包围字符串类型数据。

我们来检验一下数据是否真的插入数据表中了，执行下面的 SQL 语句：

SELECT * FROM T_Person[3]

执行完毕我们将会看到如下的输出结果（在不同的数据库系统以及管理工具下的显示效果会略有不同）：

FName	FAge	FRemark
Tom	18	USA

可以看到插入的数据已经保存在 T_Person 表中了，我们还可以运行多条 SQL 语句来插入多条数据：

```
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('Jim',20,'USA');
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('Lili',22,'China') ;
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('XiaoWang',17,' China ') ;
```

再次执行 SELECT * FROM T_Person 来查看表中的数据：

FNAME	FAGE	FREMARK
Tom	18	USA
Jim	20	USA
Lili	22	China
XiaoWang	17	China

INSERT 语句中列的顺序可以是任意的，比如我们也可以用下面的 SQL 来插入数据：

```
INSERT INTO T_Person(FAge,FName,FRemark) VALUES(21,'Kimisushi','Korea')
```

执行 SELECT * FROM T_Person 来查看表中的数据：

FNAME	FAGE	FREMARK
Tom	18	USA
Jim	20	USA
Lili	22	China
XiaoWang	17	China
Kimisushi	21	Korea

可见 INSET 语句中列的顺序不会影响数据插入的结果。

3.1.2 简化的 INSERT 语句

INSERT 语句中也并不需要我们指定表中的所有列，比如在插入数据的时候某些字段没有值，我们可以忽略这些字段。下面我们插入一条没有备注信息的数据：

```
INSERT INTO T_Person(FAge,FName) VALUES(22,'LXF')
```

执行 SELECT * FROM T_Person 来查看表中的数据：

FName	FAge	FRemark
Tom	18	USA
Jim	20	USA
Lili	22	China
XiaoWang	17	China
Kimisushi	21	Korea
LXF	22	<NULL>

INSERT 语句还有另一种用法，可以不用指定要插入的表列，这种情况下将按照定义表中字段顺序来进行插入，我们执行下面的 SQL：

```
INSERT INTO T_Person VALUES('luren1',23,'China')
```

[3]先不用管这句 SQL 语句的具体语法，只要知道它是用来查看表 T_Person 中的数据即可。

这里省略了 VALUES 前面的字段定义，VALUES 后面的值列表中按照 CREATE TABLE 语句中的顺序排列。执行 SELECT * FROM T_Person 来查看表中的数据：

FNAME	FAGE	FREMARK
Tom	18	USA
Jim	20	USA
Lili	22	China
XiaoWang	17	China
Kimisushi	21	Korea
LXF	22	<NULL>
luren1	23	China

这种省略字段列表的方法可以简化输入，不过我们推荐这种用法，因为省略字段列表之后就无法很容易的弄清楚值列表中各个值到底对应哪个字段了，非常容易导致程序出现 BUG 并且给程序的调试带来非常大的麻烦。

3.1.3 非空约束对数据插入的影响

正如“非空约束”表达的意思，如果对一个字段添加了非空约束，那么我们是不能向这个字段中插入 NULL 值的。T_Debt 表的 FAmount 字段是有非空约束的，如果我们执行下面 SQL：

```
INSERT INTO T_Debt (FNumber, FPerson) VALUES ('1', 'Jim')
```

这句 SQL 中没有为字段 FAmount 赋值，也就是说 FAmount 为空值。我们执行这句 SQL 以后数据库系统会报出类似如下的错误信息：

不能将值 NULL 插入列 'FAmount'，表 'demo.dbo.T_Debt'；列不允许有空值。INSERT 失败。

如果我们为 FAmount 设置非空值的话，则会插入成功，执行下面的 SQL：

```
INSERT INTO T_Debt (FNumber,FAmount, FPerson) VALUES ('1',200, 'Jim')
```

此句 SQL 则可以正常的执行成功。执行 SELECT * FROM T_Debt 来查看表中的数据：

FNumber	FAmount	FPerson
1	200.00	Jim

可以看到数据已经被正确的插入到表中了。

3.1.3 主键对数据插入的影响

主键是在同一张表中必须是唯一的，如果在进行数据插入的时候指定的主键与表中已有的数据重复的话则会导致违反主键约束的异常。T_Debt 表中 FNumber 字段是主键，如果我们执行下面 SQL：

```
INSERT INTO T_Debt (FNumber,FAmount, FPerson) VALUES ('1',300, 'Jim')
```

由于在上一节中我们已经向表中插入了一条 FNumber 字段为 1 的记录，所以运行这句 SQL 的时候会报出类似如下的错误信息：

```
不能在对象 'dbo.T_Debt' 中插入重复键。
```

而如果我们为 FNumber 设置一个不重复值的话，则会插入成功，执行下面的 SQL：

```
INSERT INTO T_Debt (FNumber,FAmount, FPerson) VALUES ('2',300, 'Jim')
```

此句 SQL 则可以正常的执行成功。执行 SELECT * FROM T_Debt 来查看表中的数据：

FNumber	FAmount	FPerson
1	200.00	Jim
2	300.00	Jim

可以看到数据已经被正确的插入到表中了。

3.1.4 外键对数据插入的影响

外键是指向另一个表中已有数据的约束，因此外键值必须是在目标表中存在的。如果插入的数据在目标表中不存在的话则会导致违反外键约束异常。T_Debt 表中 FPerson 字段是指向表 T_Person 的 FName 字段的外键，如果我们执行下面 SQL：

```
INSERT INTO T_Debt (FNumber,FAmount, FPerson) VALUES ('3',100, 'Jerry')
```

由于在 T_Person 表中不存在 FName 字段等于“Jerry”的数据行，所以会数据库系统会报出类似如下的错误信息：

```
INSERT 语句与 FOREIGN KEY 约束"FK__T_Debt__FPerson__1A14E395"冲突。该冲突发生于数据库"demo"，表"dbo.T_Person", column 'FName'。
```

而如果我们为 FPerson 字段设置已经在 T_Person 表中存在的 FName 字段值的话则会插入成功，执行下面的 SQL：

```
INSERT INTO T_Debt (FNumber,FAmount, FPerson) VALUES ('3',100, 'Tom')
```

此句 SQL 则可以正常的执行成功。执行 SELECT * FROM T_Debt 来查看表中的数据：

FNumber	FAmount	FPerson
1	200.00	Jim
2	300.00	Jim
3	100.00	Tom

可以看到数据已经被正确的插入到表中了。

3.2 数据的更新

录入到数据表中的数据很少有一成不变的，随着系统的运行经常需要更新表中的某些数据，比如 Tom 的家庭住址变化了我们就要在数据库中将他的家庭住址更新、新年度到来的时候我们就要将所有人员的年龄增加一岁，类似需求都要求对数据库中现有的数据进行更新。

3.2.1 简单的数据更新

UPDATE 语句用来对数据表中的数据进行更新。下边的语句用来将表 T_Person 中所有人员的 FREMARK 字段值更新为“SuperMan”：

```
UPDATE T_Person
SET FRemark = 'SuperMan'
```

执行 SELECT * FROM T_Person 来查看表中的数据：

FName	FAge	FRemark
Jim	20	SuperMan
Kimisushi	21	SuperMan
Lili	22	SuperMan
luren1	23	SuperMan
LXF	22	SuperMan
Tom	18	SuperMan
XiaoWang	17	SuperMan

可以看到所有行的 FRemark 字段值都被设置成了“SuperMan”。

来看一下刚才执行的 SQL 语句，首先它声明了要更新的表为 T_Person：

```
UPDATE T_Person
```

在 SET 子句中，我们指定将 FRemark 字段更新为新值'SuperMan'：

```
SET FRemark = 'SuperMan'
```

我们还可以在 SET 语句中定义多个列，这样就可以实现多列同时更新了，比如下面的 UPDATE 语句用来将所有人员的 FRemark 字段更新为“Sonic”，并且将年龄更新为 25：

```
UPDATE T_Person
SET FRemark = 'Sonic',
```

FAge=25

多个列之间需要使用逗号分隔开。执行完此 SQL 语句后执行 SELECT * FROM T_Person 来查看表中的数据的变化：

FName	FAge	FRemark
Jim	25	Sonic
Kimisushi	25	Sonic
Lili	25	Sonic
luren1	25	Sonic
LXF	25	Sonic
Tom	25	Sonic
XiaoWang	25	Sonic

3.2.2 带 WHERE 子句的 UPDATE 语句

目前演示的几个 UPDATE 语句都是一次性更新所有行的数据，这无法满足只更新符合特定条件的行的需求，比如“将 Tom 的年龄修改为 12 岁”。要实现这样的功能只要使用 WHERE 子句就可以了，在 WHERE 语句中我们设定适当的过滤条件，这样 UPDATE 语句只会更新符合 WHERE 子句中过滤条件的行，而其他行的数据则不被修改。

执行下边的 UPDATE 语句：

```
UPDATE T_Person
SET FAge = 12
WHERE FNAME='Tom'
```

执行完此 SQL 语句后执行 SELECT * FROM T_Person 来查看表中的数据的变化：

FName	FAge	FRemark
Jim	25	Sonic
Kimisushi	25	Sonic
Lili	25	Sonic
luren1	25	Sonic
LXF	25	Sonic
Tom	12	Sonic
XiaoWang	25	Sonic

可以看到只有第一行中的 FAGE 被更新了。WHERE 子句“WHERE FNAME='Tom'”表示我们只更新 FNAME 字段等于'Tom'的行。由于 FNAME 字段等于'Tom'的只有一行，所以仅有一行记录被更新，但是如果有多个符合条件的行的话将会有多行被更新，比如下面 UPDATE 语句将所有年龄为 25 的人员的备注信息修改为“BlaBla”：

```
UPDATE T_Person
SET FRemark = 'BlaBla'
WHERE FAge =25
```

执行完此 SQL 语句后执行 SELECT * FROM T_Person 来查看表中的数据的变化：

FName	FAge	FRemark
Jim	25	BlaBla
Kimisushi	25	BlaBla
Lili	25	BlaBla
luren1	25	BlaBla
LXF	25	BlaBla

Tom	12	Sonic
XiaoWang	25	BlaBla

目前为止我们演示的都是非常简单的 WHERE 子句，我们可以使用复杂的 WHERE 语句来满足更加复杂的需求，比如下面的 UPDATE 语句就用来将 FNAME 等于'Jim'或者'LXF'的行的 FAge 字段更新为 22：

```
UPDATE T_Person
SET FAge = 22
WHERE FName ='jim' OR FName='LXF'
```

执行完此 SQL 语句后执行 SELECT * FROM T_Person 来查看表中的数据的变化：

FName	FAge	FRemark
Jim	22	BlaBla
Kimisushi	25	BlaBla
Lili	25	BlaBla
luren1	25	BlaBla
LXF	22	BlaBla
Tom	12	Sonic
XiaoWang	25	BlaBla

这里我们使用 OR 逻辑运算符来组合两个条件来实现复杂的过滤逻辑，我们还可以使用 OR、NOT 等运算符实现更加复杂的逻辑，甚至能够使用模糊查询、子查询等实现高级的数据过滤，关于这些知识我们将在后面的章节专门介绍。

3.2.3 非空约束对数据更新的影响

正如“非空约束”表达的意思，如果对一个字段添加了非空约束，那么我们是不能将这个字段中的值更新为 NULL 的。T_Debt 表的 FAmount 字段是有非空约束的，如果我们执行下面 SQL：

```
UPDATE T_Debt set FAmount = NULL WHERE FPerson='Tom'
```

这句 SQL 为 FAmount 设置空值。我们执行这句 SQL 以后数据库系统会报出类似如下的错误信息：

不能将值 NULL 插入列 'FAmount'，表 'demo.dbo.T_Debt'；列不允许有空值。UPDATE 失败。

如果我们为 FAmount 设置非空值的话，则会插入成功，执行下面的 SQL：

```
UPDATE T_Debt set FAmount =123 WHERE FPerson='Tom'
```

此句 SQL 则可以正常的执行成功。执行 SELECT * FROM T_Debt 来查看表中的数据：

FNumber	FAmount	FPerson
1	200.00	Jim
2	300.00	Jim
3	123.00	Tom

可以看到数据已经被正确的更新到表中了。

3.2.3 主键对数据更新的影响

主键是在同一张表中必须是唯一的，如果在进行数据更新的时候指定的主键与表中已有的数据重复的话则会导致违反主键约束的异常。T_Debt 表中 FNumber 字段是主键，如果我们执行下面 SQL：

```
UPDATE T_Debt set FNumber = '2' WHERE FPerson='Tom'
```

由于表中已经存在一条 FNumber 字段为 2 的记录，所以运行这句 SQL 的时候会报出类似如下的错误信息：

违反了 PRIMARY KEY 约束 'PK__T_Debt__1920BF5C'。不能在对象 'dbo.T_Debt' 中插入重复键。

而如果我们为 FNumber 设置一个不重复值的话，则会插入成功，执行下面的 SQL：

```
UPDATE T_Debt set FNumber = '8' WHERE FPerson='Tom'
```

此句 SQL 则可以正常的执行成功。执行 SELECT * FROM T_Debt 来查看表中的数据：

FNumber	FAmount	FPerson
1	200.00	Jim
2	300.00	Jim
8	123.00	Tom

可以看到数据已经被正确的更新到表中了。

3.2.4 外键对数据更新的影响

外键是指向另一个表中已有数据的约束，因此外键值必须是在目标表中存在的。如果更新后的数据在目标表中不存在的话则会导致违反外键约束异常。T_Debt 表中 FPerson 字段是指向表 T_Person 的 FName 字段的外键，如果我们执行下面 SQL：

```
UPDATE T_Debt set FPerson = 'Merry' WHERE FNumber='1'
```

由于在 T_Person 表中不存在 FName 字段等于“Merry”的数据行，所以会数据库系统会报出类似如下的错误信息：

UPDATE 语句与 FOREIGN KEY 约束"FK__T_Debt__FPerson__1A14E395"冲突。该冲突发生于数据库"demo"，表"dbo.T_Person", column 'FName'。

而如果我们为 FPerson 字段设置已经在 T_Person 表中存在的 FName 字段值的话则会插入成功，执行下面的 SQL：

```
UPDATE T_Debt set FPerson = 'Lili' WHERE FNumber='1'
```

此句 SQL 则可以正常的执行成功。执行 SELECT * FROM T_Debt 来查看表中的数据：

FNumber	FAmount	FPerson
1	200.00	Lili
2	300.00	Jim
8	123.00	Tom

可以看到数据已经被正确的更新到表中了。

3.3 数据的删除

数据库中的数据一般都有一定的生命周期，当数据不再需要的时候我们就要将其删除，执行 DELETE 语句就可以将数据从表中删除。不过需要注意的就是如果被删除的数据行是某个外键关联关系中的被引用数据的话，则进行删除的时候会失败，如果要删除成功则必须首先删除引用者才可以。

3.3.1 简单的数据删除

删除数据的 SQL 语句非常简单，我们只要指定要删除的表就可以了，比如我们要将 T_Debt 和 T_Person 表中的数据删除，那么执行下面的 SQL 语句即可：

```
DELETE FROM T_Debt;
DELETE FROM T_Person;
```

由于 T_Debt 表中 FPerson 字段是指向表 T_Person 的 FName 字段的外键，所以必须首先删除 T_Debt 表中的数据后才能删除 T_Person 中的数据。

执行 SELECT * FROM T_Debt 查看 T_Debt 表中的数据变化：

FNumber	FAmount	FPerson

执行完此 SQL 语句后执行 SELECT * FROM T_Person 来查看 T_Person 表中的数据变化：

FName	FAge	FRemark

可以见表中所有的数据行都被删除了，T_Debt 和 T_Person 中没有任何数据。

初学者往往容易把 DROP TABLE 语句和 DELETE 混淆，虽然二者名字中都有“删除”两个

字，不过 DELETE 语句仅仅是删除表中的数据行，而表的结构还存在，而 DROP TABLE 语句则不仅将表中的数据行全部删除，而且还将表的结构也删除。可以形象的比喻成 DELETE 语句仅仅是“吃光碗里的饭”，而 DROP TABLE 语句则是“吃光碗里的饭还将碗砸碎”。如果我们执行“DROP TABLE T_Person”的话，那么再次执行“SELECT * FROM T_Person”的时候数据库系统就会报告“数据表 T_Person 不存在”。

上边介绍的 DELETE 语句将表中的所有数据都删除了，如果我们只想删除我们指定的数据行怎么办呢？和 UPDATE 语句类似，DELETE 语句也提供了 WHERE 语句进行数据的过滤，这样只有符合过滤条件的数据行才会被删除。

3.3.2 带 WHERE 子句的 DELETE 语句

由于前面我们执行“DELETE FROM T_Person”语句将数据表 T_Person 中的数据全部删除了，为了演示带 WHERE 子句的 DELETE 语句，我们需要重新插入一些数据到 T_Person 中。请执行下面的 SQL 语句：

```
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('Jim',20,'USA');
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('Lili',22,'China') ;
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('XiaoWang',17,' China ') ;
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('Sam',16,'China') ;
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('BlueFin',12,'Mars') ;
```

执行完此 SQL 语句后执行 SELECT * FROM T_Person 来查看 T_Person 表中新插入的数据：

FNAME	FAGE	FREMARK
Jim	20	USA
Lili	22	China
XiaoWang	17	China
Sam	16	China
BlueFin	12	Mars

我们要删除年龄大于 20 岁或者来自火星（Mars）的人员，因此使用带复合逻辑 WHERE 子句，如下：

```
DELETE FROM T_Person WHERE FAge > 20 or FRemark = 'Mars'
```

执行完此 SQL 语句后执行 SELECT * FROM T_Person 来查看表中的数据的变化：

FNAME	FAGE	FREMARK
Jim	20	USA
XiaoWang	17	China
Sam	16	China

可以看到年龄为 22 岁的 Lili 和来自火星的 BlueFin 被删除了。

本章已经结束，我们不再需要 T_Person、T_Debt 这两张表，因此需要将它们删除，执行下面的 SQL 即可：

```
DROP TABLE T_Debt;
DROP TABLE T_Person;
```

第四章 数据的检索

到目前为止，我们已经学习了如何创建数据表、如何修改数据表以及如何删除数据表，我们还学习了如何将数据插入数据表、如何更新数据表中的数据以及如何数据删除。创建数据表是在创建存放数据的容器，修改和删除数据表是在维护数据模型的正确性，将数据插入

数据表、更新数据表以及删除数据表中的数据则是在维护数据库中数据与真实业务数据之间的同步，这些操作都不是经常发生的，它们只占据数据库操作中很小的一部分，我们大部分时间都是在对数据库中的数据进行检索，并且基于检索结果进行响应的分析，可以说数据的检索是数据库中最重要的功能。

与数据表结构的管理以及数据表中数据的管理不同，数据检索所需要面对的问题是非常复杂的，不仅要求能够完成“检索出所有年龄小于 12 岁的学生”、“检索出所有旷工时间超过 3 天的职工”等简单的检索任务，而且还要完成“检索出本季度每种商品的出库入库详细情况”、“检索出所有学生家长的工作单位信息”等复杂的任务，甚至还需要完成其他更加复杂的检索任务。数据检索面对的场景是异常复杂的，因此数据检索的语法也是其他功能所不能比的，不仅语法规则非常复杂，而且使用方式也非常灵活。本书中大部分内容都是讲解数据检索相关知识的，为了降低学习的梯度，本章我们将讲解基本的数据检索语法，这些语法是数据检索功能中最基础也是最核心的部分，因此只有掌握我们才能继续学习更加复杂的应用。

本章中我们将使用一些数据表，为了更容易的运行本章中的例子，必须首先创建所需要的数据表，因此下面列出本章中要用到数据表的创建 SQL 语句：

MYSQL:

```
CREATE TABLE T_Employee (FNumber VARCHAR(20),FName VARCHAR(20),FAge INT,FSalary DECIMAL(10,2),PRIMARY KEY (FNumber))
```

MSSQLServer:

```
CREATE TABLE T_Employee (FNumber VARCHAR(20),FName VARCHAR(20),FAge INT,FSalary NUMERIC(10,2),PRIMARY KEY (FNumber))
```

Oracle:

```
CREATE TABLE T_Employee (FNumber VARCHAR2(20),FName VARCHAR2(20),FAge NUMBER (10),FSalary NUMERIC(10,2),PRIMARY KEY (FNumber))
```

DB2:

```
CREATE TABLE T_Employee (FNumber VARCHAR(20) NOT NULL,FName VARCHAR(20),FAge INT,FSalary DECIMAL(10,2),PRIMARY KEY (FNumber))
```

请在不同的数据库系统中运行相应的 SQL 语句。T_Employee 为记录员工信息的数据表，其中主键字段 FNumber 为员工工号，FName 为人员姓名，FAge 为年龄，FSalary 为员工月工资。

为了更加直观的验证本章中检索语句的正确性，我们需要在 T_Employee 表中预置一些初始数据，请在数据库中执行下面的数据插入 SQL 语句：

```
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('DEV001','Tom',25,8300);
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('DEV002','Jerry',28,2300.80);
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('SALES001','John',23,5000);
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('SALES002','Kerry',28,6200);
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('SALES003','Stone',22,1200);
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('HR001','Jane',23,2200.88);
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('HR002','Tina',25,5200.36);
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('IT001','Smith',28,3900);
```

4.1 SELECT 基本用法

SELECT 是实现数据检索的 SQL 语句，本节我们学习 SELECT 语句最基本的用法。

4.1.1 简单的数据检索

“取出一张表中所有的数据”是最简单的数据检索任务，完成这个最简单任务的 SQL

语句也是最简单的，我们只要执行“SELECT * FROM 表名”即可。比如我们执行下面的 SQL 语句：

```
SELECT * FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT001	Smith	28	3900.00
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00
SALES003	Stone	22	1200.00

执行结果中列出了表中的所有行，而且包含了表中每一列的数据。

4.1.2 检索出需要的列

上面的 SQL 语句执行的结果中包含了表中每一列的数据，有的时候并不需要所有列的数据。比如我们只需要检索所有员工的工号，如果我们采用“SELECT * FROM T_Employee”进行检索的话，数据库系统会将所有列的数据从数据库中取出来，然后通过网络发送给我们，这不仅会占用不必要的 CPU 资源和内存资源，而且会占用一定的网络带宽，这在我们这种测试模式下不会有影响，但是如果是在真实的生产环境中的话就会大大降低系统的吞吐量，因此最好在检索的之后只检索需要的列。那么如何只检索出需要的列呢？

检索出所有的列的 SQL 语句为“SELECT * FROM T_Employee”，其中的星号“*”就意味着“所有列”，那么我们只要将星号“*”替换成我们要检索的列名就可以了。比如我们执行下面的 SQL 语句：

```
SELECT FNumber FROM T_Employee
```

这就表示我们要检索出表 T_Employee 中的所有数据，并且只取出 FNumber 列。执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber
DEV001
DEV002
HR001
HR002
IT001
SALES001
SALES002
SALES003

可以看到只有 FNumber 列中的数据被检索出来了。

上面的 SQL 语句列出了 FNumber 列中的数据，那么如果想列出不止一个列中的数据呢？非常简单，只要在 SELECT 语句后列出各个列的列名就可以了，需要注意的就是各个列之间要用半角的逗号“,”分隔开。比如我们执行下面的 SQL 语句：

```
SELECT FName,FAge FROM T_Employee
```

这就表示我们要检索出表 T_Employee 中的所有数据，并且只取出 FName 和 FAge 两列的内容。执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FAge
Tom	25
Jerry	28
Jane	23
Tina	25
Smith	28
John	23
Kerry	28
Stone	22

可以看到，执行结果中列出了所有员工的姓名和他们的年龄。

如果要用这种显式指定数据列的方式取出所有列，我们就可以编写下面的 SQL：

```
SELECT FNumber,FName,FAge,FSalary FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT001	Smith	28	3900.00
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00
SALES003	Stone	22	1200.00

这和“`SELECT * FROM T_Employee`”的执行结果是一致的，也就是说“`SELECT FNumber,FName,FAge,FSalary FROM T_Employee`”和“`SELECT * FROM T_Employee`”是等价的。

4.1.3 列别名

由于编码命名规范、编程框架要求等的限制，数据表的列名有的时候意思并不是非常易读，比如 `T_Employee` 中的姓名字段名称为 `FName`，而如果我们能用 `Name` 甚至“姓名”来代表这个字段就更清晰易懂了，可是字段名已经不能更改了，那么难道就不能用别的名字来使用已有字段了吗？

当然不是！就像可以为每个人取一个外号一样，我们可以为字段取一个别名，这样就可以使用这个别名来引用这个列了。别名的定义格式为“列名 `AS` 别名”，比如我们要为 `FNumber` 字段取别名为 `Number1`[4]，`FName` 字段取别名为 `Name`、`FAge` 字段取别名为 `Age`、为 `FSalary` 字段取别名为 `Salary`，那么编写下面的 SQL 即可：

```
SELECT FNumber AS Number1,FName AS Name,FAge AS Age,FSalary AS Salary FROM
T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

Number1	Name	Age	Salary
DEV001	Tom	25	8300.00
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88

[4]由于 `Number` 在 `Oracle` 中为关键字，所以如果在为 FNumber 字段取别名为 `Number`，那么将会在 `Oracle` 中运行失败，所以这里取别名为 `Number1`。

HR002	Tina	25	5200.36
IT001	Smith	28	3900.00
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00
SALES003	Stone	22	1200.00

这里的执行结果和“`SELECT FNumber,FName,FAge,FSalary FROM T_Employee`”执行结果一样，唯一不同的地方就是表头中的列名，这里的表头的列名就是我们为各列设定的别名。

定义别名的时候“AS”不是必须的，是可以省略的，比如下面的 SQL 也是正确的：

```
SELECT FNumber Number1,FName Name,FAge Age,FSalary Salary FROM T_Employee
```

如果数据库系统支持中文列名，那么还可以用中文来为列设定别名，这样可读性就更好了，比如在 MSSQLServer 中文版上执行下面的 SQL：

```
SELECT FNumber 工号,FName 姓名,FAge 年龄,FSalary 工资 FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

工号	姓名	年龄	工资
DEV001	Tom	25	8300.00
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT001	Smith	28	3900.00
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00
SALES003	Stone	22	1200.00

4.1.4 按条件过滤

前面演示的例子都是检索出表中所有的数据，不过在很多情况下我们需要按照一定的过滤条件来检索表中的部分数据，这个时候可以先检索出表中所有的数据，然后检查每一行看是否符合指定的过滤条件。比如我们要检索出所有工资少于 5000 元的员工的姓名，那么可以编写下面的代码来处理[5]：

```
result = executeQuery(“SELECT FName, FSalary FROM T_Employee”);
for(i=0;i<result.count;i++)
{
    salary = result[i].get(“FSalary”);
    if(salary<5000)
    {
        name = result[i].get(“FName”);
        print(name+”的工资少于 5000 元，为:”+salary);
    }
}
```

这种处理方式非常清晰简单，在处理小数据量以及简单的过滤条件的时候没有什么不妥的地方，但是如果数据表中有大量的数据（数以万计甚至百万、千万数量级）或者过滤条件非常复杂的话就会带来很多问题：

[5]为了不涉及具体宿主语言的细节，这里采用的是实例性的类 C 伪代码，如果需要您可以将其翻译成对应宿主语言的代码。本书其他部分也将采用相同的伪代码来表示宿主语言无关的一些算法。

- 由于将表中所有的数据都从数据库中检索出来，所以会有非常大的内存消耗以及网络资源消耗。
- 需要逐条检索每条数据是否符合过滤条件，所以检索速度非常慢，当数据量大的时候这种速度是让人无法忍受的。
- 无法实现复杂的过滤条件。如果要实现“检索工资小于 5000 或者年龄介于 23 岁与 28 岁之间的员工姓名”这样的逻辑的话就要编写复杂的判断语句，而如果要关联其他表进行查询的话则会更加复杂。

数据检索是数据库系统的一个非常重要的任务，它内置了对按条件过滤数据的支持，只要为 SELECT 语句指定 WHERE 语句即可，其语法与上一章中讲的数据更新、数据删除的 WHERE 语句非常类似，比如完成“检索出所有工资少于 5000 元的员工的姓名”这样的功能可以使用下面的 SQL 语句：

```
SELECT FName FROM T_Employee
WHERE FSalary<5000
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName
Jerry
Jane
Smith
Stone

WHERE 子句还支持复杂的过滤条件，下面的 SQL 语句用来检索出所有工资少于 5000 元或者年龄大于 25 岁的员工的所有信息：

```
SELECT * FROM T_Employee
WHERE FSalary<5000 OR FAge>25
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
IT001	Smith	28	3900.00
SALES002	Kerry	28	6200.00
SALES003	Stone	22	1200.00

使用 WHERE 子句只需指定过滤条件就可以，我们无需关心数据库系统是如果进行查找的，数据库会采用适当的优化算法进行查询，大大降低了 CPU 资源的占用。

4.1.5 数据汇总

有时需要对数据库中的数据进行一些统计，比如统计员工总数、统计年龄大于 25 岁的员工中的最低工资、统计工资大于 3800 元的员工的平均年龄。SQL 中提供了聚合函数来完成计算统计结果集条数、某个字段的最大值、某个字段的最小值、某个字段的平均值以及某个字段的合计值等数据统计的功能，SQL 标准中规定了下面几种聚合函数：

函数名	说明
MAX	计算字段最大值
MIN	计算字段最小值
AVG	计算字段平均值
SUM	计算字段合计值
COUNT	统计数据条数

这几个聚合函数都有一个参数，这个参数表示要统计的字段名，比如要统计工资总额，那么就需要把 FSalary 做为 SUM 函数的参数。通过例子来看一下聚合函数的用法。第一个例子是查询年龄大于 25 岁的员工的最高工资，执行下面的 SQL：

```
SELECT MAX(FSalary) FROM T_Employee
WHERE FAge>25
```

执行完毕我们就能在输出结果中看到下面的执行结果：

6200.00

为了方面的引用查询的结果，也可以为聚合函数的计算结果指定一个别名，执行下面的 SQL：

```
SELECT MAX(FSalary) as MAX_SALARY FROM T_Employee
WHERE FAge>25
```

执行完毕我们就能在输出结果中看到下面的执行结果：

MAX_SALARY
6200.00

第二个例子我们来统计一下工资大于 3800 元的员工的平均年龄，执行下面的 SQL：

```
SELECT AVG(FAge) FROM  T_Employee
WHERE FSalary>3800
```

执行完毕我们就能在输出结果中看到下面的执行结果：

25

第三个例子我们来统计一下公司每个月应支出工资总额，执行下面的 SQL：

```
SELECT SUM(FSalary) FROM  T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

34302.04

我们还可以多次使用聚合函数，比如下面的 SQL 用来统计公司的最低工资和最高工资：

```
SELECT MIN(FSalary),MAX(FSalary) FROM  T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1200.00	8300.00

最后一个介绍的函数就是统计记录数量的 COUNT，这个函数有一点特别，因为它的即可以像其他聚合函数一样使用字段名做参数，也可以使用星号“*”做为参数。我们执行下面的 SQL：

```
SELECT COUNT(*),COUNT(FNumber) FROM  T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

8	8

可以看到 **COUNT**(*)、**COUNT**(FNumber)两种方式都能统计出记录的条数，据此为数不少的开发人员都认为 **COUNT**(*)、**COUNT**(字段名)这两种使用方式是等价的。下面通过例子来说明，为了看到两种使用方式的区别需要首先向表 T_Employee 中插入一条新的数据，执行下面的 SQL：

```
INSERT INTO T_Employee(FNumber,FAge,FSalary) VALUES('IT002',27,2800)
```

需要注意的就是这句 INSERT 语句没有为 FName 字段赋值，也就是说新插入的这条数据的 FName 字段值为空，可以执行 **SELECT** * **FROM** T_Employee 来查看表 T_Employee 中的内容：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36

IT001	Smith	28	3900.00
IT002	<NULL>	27	2800.00
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00
SALES003	Stone	22	1200.00

可以看到 FNumber 为 IT002 的行的 FName 字段是空值。接着执行下面的 SQL：

```
SELECT COUNT(*),COUNT(FNumber),COUNT(FName) FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

9	9	8

可以看到**COUNT**(*)、**COUNT**(FNumber)两个表达式的计算结果都是9，而**COUNT**(FName)的计算结果是8。也就反应出了两种使用方式的区别：**COUNT**(*)统计的是结果集的总条数，而COUNT(FName)统计的则是除了结果集中 FName 不为空值（也就是不等于 NULL）的记录的总条数。由于 FNumber 为 IT002 的行的 FName 字段是空值，所以 **COUNT**(FName)的计算结果是 8。因此在使用聚合函数 COUNT 的时候一定要区分两种使用方式的区别，以防止出现数据错误。

4.1.6 排序

到目前为止，数据检索结果的排列顺序取决于数据库系统所决定的排序机制，这种排序机制可能是按照数据的输入顺序决定的，也有可能是按照其他的算法来决定的。在有的情况下我们需要按照某种排序规则来排列检索结果，比如按照工资从高到低的顺序排列或者按照姓名的字符顺序排列等。SELECT 语句允许使用 ORDER BY 子句来执行结果集的排序方式。

ORDER BY 子句位于 SELECT 语句的末尾，它允许指定按照一个列或者多个列进行排序，还可以指定排序方式是升序（从小到大排列）还是降序（从大到小排列）。比如下面的 SQL 语句演示了按照年龄排序所有员工信息的列表：

```
SELECT * FROM T_Employee
ORDER BY FAge ASC
```

执行完毕我们就能在输出结果中看到下面的执行结果，可以看到输出结果已经按照 FAge 字段进行升序排列了：

FNumber	FName	FAge	FSalary
SALES003	Stone	22	1200.00
SALES001	John	23	5000.00
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
DEV001	Tom	25	8300.00
IT002	<NULL>	27	2800.00
SALES002	Kerry	28	6200.00
DEV002	Jerry	28	2300.80
IT001	Smith	28	3900.00

这句 SQL 中的“**ORDER BY** FAge ASC”指定了按照 FAge 字段的顺序进行升序排列，其中 ASC 代表升序。因为对于 ORDER BY 子句来说，升序是默认的排序方式，所以如果要采用升序的话可以不指定排序方式，也就是“ASC”是可以省略的，比如下面的 SQL 语句具有和上面的 SQL 语句等效的执行效果：

```
SELECT * FROM T_Employee
ORDER BY FAge
```

执行完毕我们就能在输出结果中看到下面的执行结果，可以看到输出结果同样按照

FAge 字段进行升序排列了：

FNumber	FName	FAge	FSalary
SALES003	Stone	22	1200.00
SALES001	John	23	5000.00
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
DEV001	Tom	25	8300.00
IT002	<NULL>	27	2800.00
SALES002	Kerry	28	6200.00
DEV002	Jerry	28	2300.80
IT001	Smith	28	3900.00

如果需要按照降序排列，那么只要将 ASC 替换为 DESC 即可，其中 DESC 代表降序。执行下面的 SQL 语句：

```
SELECT * FROM  T_Employee
ORDER BY FAge DESC
```

执行完毕我们就能在输出结果中看到下面的执行结果，可以看到输出结果已经按照 FAge 字段进行降序排序了：

FNumber	FName	FAge	FSalary
DEV002	Jerry	28	2300.80
IT001	Smith	28	3900.00
SALES002	Kerry	28	6200.00
IT002	<NULL>	27	2800.00
DEV001	Tom	25	8300.00
HR002	Tina	25	5200.36
HR001	Jane	23	2200.88
SALES001	John	23	5000.00
SALES003	Stone	22	1200.00

可以看到上面的检索结果中有几组年龄相同的记录，这些年龄相同的记录之间的顺序是由数据库系统决定的，但是有时可能需要需要完成“按照年龄从大到小排序，如果年龄相同则按照工资从大到小排序”之类的排序功能。这可以通过指定多个排序规则来完成，因为 ORDER BY 语句允许指定多个排序列，各个列之间使用逗号隔开即可。执行下面的 SQL 语句：

```
SELECT * FROM  T_Employee
ORDER BY FAge DESC,FSalary DESC
```

FNumber	FName	FAge	FSalary
SALES002	Kerry	28	6200.00
IT001	Smith	28	3900.00
DEV002	Jerry	28	2300.80
IT002	<NULL>	27	2800.00
DEV001	Tom	25	8300.00
HR002	Tina	25	5200.36
SALES001	John	23	5000.00
HR001	Jane	23	2200.88
SALES003	Stone	22	1200.00

可以看到年龄相同的记录按照工资从高到低的顺序排列了。

对于多个排序规则，数据库系统会按照优先级进行处理。数据库系统首先按照第一个排序规则进行排序；如果按照第一个排序规则无法区分两条记录的顺序，则按照第二个排序规则进行排序；如果按照第二个排序规则无法区分两条记录的顺序，则按照第三个排序规则进行排序；……以此类推。以上面的 SQL 语句为例，数据库系统首先按照 FAge 字段的降序进行排列，如果按照个排序规则无法区分两条记录的顺序，则按照 FSalary 字段的降序进行排列。

ORDER BY 子句完全可以与 WHERE 子句一起使用，唯一需要注意的就是 ORDER BY 子句要放到 WHERE 子句之后，不能颠倒它们的顺序。比如我们尝试执行下面的 SQL 语句：

```
SELECT * FROM T_Employee
ORDER BY FAge DESC,FSalary DESC
WHERE FAge>23
```

执行以后数据库系统会报错提示此语句有语法错误，如果我们颠倒 ORDER BY 和 WHERE 子句的位置则可以执行通过：

```
SELECT * FROM T_Employee
WHERE FAge>23
ORDER BY FAge DESC,FSalary DESC
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
SALES002	Kerry	28	6200.00
IT001	Smith	28	3900.00
DEV002	Jerry	28	2300.80
IT002	<NULL>	27	2800.00
DEV001	Tom	25	8300.00
HR002	Tina	25	5200.36

前面我们提到，如果没有为 SELECT 语句指定 ORDER BY 子句，数据库系统会按照某种内置的规则对检索结果进行排序。如果您对检索结果的前后排列顺序有要求，那么即使数据库系统返回的检索结果符合要求也最好显式的指定 ORDER BY 子句，因为这种系统提供的排序方式是不稳定的，不仅在不同数据库系统之间存在差异，而且即使对同一种数据库系统来说在不同的条件下这种排序方式也是有可能发生改变的。

4.2 高级数据过滤

数据检索是数据库系统中最复杂的功能，而数据过滤则是数据检索中最核心的部分，到目前为止我们讲解的数据过滤都是“过滤某字段等于某个值的所有记录”、“过滤某字段小于某个值或者大于某个值的所有记录”等简单的数据过滤方式，这显然是无法满足真实业务系统中的各种数据过滤条件的，因此本节我们将介绍一些单表查询时的高级数据过滤技术。需要注意的是，本节讲解的高级数据过滤技巧几乎同样适用于 Update 语句和 Delete 语句中的 Where 子句。

4.2.1 通配符过滤

到目前为止，我们讲解的数据过滤方式都是针对特定值的过滤，比如“检索所有年龄为 25 的所有员工信息”、“检索所有工资介于 2500 元至 3800 元之间的所有记录”，但是这种过滤方式并不能满足一些模糊的过滤方式。比如，检索所有姓名中含有“th”的员工或者检索所有姓“王”的员工，实现这样的检索操作必须使用通配符进行过滤。

SQL 中的通配符过滤使用 LIKE 关键字，可以像使用 OR、AND 等操作符一样使用它，它是一个二元操作符，左表达式为待匹配的字段，而右表达式为待匹配的通配符表达式。通配符表达式由通配符和普通字符组成，主流数据库系统支持的通配符有单字符匹配和多字符匹

配，有的数据库系统还支持集合匹配。

4.2.1.1 单字符匹配

进行单字符匹配的通配符为半角下划线“_”，它匹配单个出现的字符。比如通配符表达式“b_d”匹配第一个字符为 b、第二个字符为任意字符、第三个字符为 d 的字符串，“bed”、“bad”都能匹配这个表达式，而“bd”、“abc”、“build”等则不能匹配这个表达式；通配符表达式“_oo_”匹配第一个字符为任意字符、第二个字符为 o、第三个字符为 o、第四个字符为任意字符的字符串，“look”、“took”、“cool”都能匹配这个表达式，而“rom”、“todo”等则不能匹配这个表达式。

下面来演示一下单字符匹配的用法。我们来检索 T_Employee 表中 FName 字段匹配如下规则的数据行：以任意字符开头，剩余部分为“erry”。根据通配符表达式语法，我们得知这个匹配规则对应的通配符表达式为“_erry”，因此编写如下的 SQL：

```
SELECT * FROM T_Employee
WHERE FName LIKE '_erry'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV002	Jerry	28	2300.80
SALES002	Kerry	28	6200.00

“Jerry”、“Kerry”两个字符串能够匹配通配符表达式“_erry”，所以被显示到了结果集中，而其他数据行则由于不匹配此通配符表达式，所以被过滤掉了。

单字符匹配在通配符表达式中可以出现多次，比如我们要检索长度为 4、第三个字符为“n”、其它字符为任意字符的姓名。根据通配符表达式语法，我们得知这个匹配规则对应的通配符表达式为“__n_”（注意前两个字符为连续的两个下划线），那么需要编写如下的 SQL：

```
SELECT * FROM T_Employee
WHERE FName LIKE '__n_'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36

4.2.1.2 多字符匹配

使用下划线可以实现“匹配长度为 5、以 ab 开头、剩余字符任意”的功能，而对于“匹配以 k 开头，长度不限，剩余字符任意”这样的需求则无法满足，这时就需要使用多字符匹配了。进行多字符匹配的通配符为半角百分号“%”，它匹配任意次数（零或多个）出现的任意字符。比如通配符表达式“k%”匹配以“k”开头、任意长度的字符串，“k”、“kerry”、“kb” 都能匹配这个表达式，而“ark”、“luck”、“3kd”等则不能匹配这个表达式；配符表达式“b%t”匹配以“b”开头、以“t”结尾、任意长度的字符串，“but”、“bt”、“belt” 都能匹配这个表达式，而“turbo”、“tube”、“tb”等则不能匹配这个表达式。

下面来演示一下多字符匹配的用法。我们来检索 T_Employee 表中 FName 字段匹配如下规则的数据行：以“T”开头长度，长度任意。根据通配符表达式语法，我们得知这个匹配规则对应的通配符表达式为“T%”，因此编写如下的 SQL：

```
SELECT * FROM T_Employee
WHERE FName LIKE 'T%'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00

HR002	Tina	25	5200.36

接着我们来检索姓名中包含字母“n”的员工信息，编写如下 SQL：

```
SELECT * FROM T_Employee
WHERE FName LIKE '%n%'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
SALES001	John	23	5000.00
SALES003	Stone	22	1200.00

单字符匹配和多字符匹配还可以一起使用。我们来检索 T_Employee 表中 FName 字段匹配如下规则的数据行：最后一个字符为任意字符、倒数第二个字符为“n”、长度任意的字符串。根据通配符表达式语法，我们得知这个匹配规则对应的通配符表达式为“%n_”，因此编写如下的 SQL：

```
SELECT * FROM T_Employee
WHERE FName LIKE '%n_'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
SALES003	Stone	22	1200.00

4.2.1.3 集合匹配

集合匹配只在 MSSQLServer 上提供支持，在 MYSQL、Oracle、DB2 等数据库中不支持，必须采用变通的手段来实现。

进行集合匹配的通配符为“[]”，方括号中包含一个字符集，它匹配与字符集中任意一个字符相匹配的字符。比如通配符表达式“[bt]%”匹配第一个字符为 b 或者 t、长度不限的字符串，“bed”、“token”、“t”都能匹配这个表达式，而“at”、“lab”、“lot”等则不能匹配这个表达式。

下面来演示一下多字符匹配的用法。我们来检索 T_Employee 表中 FName 字段匹配如下规则的数据行：以“S”或者“J“开头长度，长度任意。根据通配符表达式语法，我们得知这个匹配规则对应的通配符表达式为“[SJ]%”，因此编写如下的 SQL：

```
SELECT * FROM T_Employee
WHERE FName LIKE '[SJ]%'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
IT001	Smith	28	3900.00
SALES001	John	23	5000.00
SALES003	Stone	22	1200.00

还可以使用否定符“^”来对集合取反，它匹配不与字符集中任意一个字符相匹配的字符。比如通配符表达式“[^bt]%”匹配第一个字符不为 b 或者 t、长度不限的字符串，“at”、“lab”、“lot”都能匹配这个表达式，而 “bed”、“token”、“t”等则不能匹配这个表达式。

我们来检索 T_Employee 表中 FName 字段匹配如下规则的数据行：不以“S”或者“J“开

头长度，长度任意。根据通配符表达式语法，我们得知这个匹配规则对应的通配符表达式为“[^SJ]%”，因此编写如下的 SQL：

```
SELECT * FROM T_Employee
WHERE FName LIKE '[^SJ]%'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
HR002	Tina	25	5200.36
SALES002	Kerry	28	6200.00

集合匹配只在 MSSQLServer 上提供支持，不过在其他数据库中我们可以通过变通手段来实现相同的效果。比如下面的 SQL 可以实现和本节第一个例子相同的效果：

```
SELECT * FROM T_Employee
WHERE FName LIKE 'S%' OR FName LIKE 'J%'
```

而下面的 SQL 可以实现和本节第二个例子相同的效果：

```
SELECT * FROM T_Employee
WHERE NOT(FName LIKE 'S%') AND NOT(FName LIKE 'J%')
```

通配符过滤一个非常强大的功能，不过在使用通配符过滤进行检索的时候，数据库系统会对全表进行扫描，所以执行速度非常慢。因此不要过分使用通配符过滤，在使用其他方式可以实现的效果的时候就应该避免使用通配符过滤。

4.2.2 空值检测

没有添加非空约束列是可以为空值的（也就是 NULL），有时我们需要对空值进行检测，比如要查询所有姓名未知的员工信息。既然 NULL 代表空值，有的开发人员试图通过下面的 SQL 语句来实现：

```
SELECT * FROM T_Employee
WHERE FNAME=null
```

这个语句是可以执行的，不过执行以后我们看不到任何的执行结果，那个Fnumber为“IT002”的数据行中Fname字段为空，但是没有被查询出来。这是因为在SQL语句中对空值的处理有些特别，不能使用普通的等于运算符进行判断，而要使用IS NULL关键字，使用方法为“待检测字段名IS NULL”，比如要查询所有姓名未知的员工信息，则运行下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FNAME IS NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
IT002	<NULL>	27	2800.00

如果要检测“字段不为空”，则要使用IS NOT NULL，使用方法为“待检测字段名IS NOT NULL”，比如要查询所有姓名已知的员工信息，则运行下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FNAME IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36

IT001	Smith	28	3900.00
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00
SALES003	Stone	22	1200.00

IS NULL/IS NOT NULL可以和其他的过滤条件一起使用。比如要查询所有姓名已知且工资小于5000的员工信息，则运行下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FNAME IS NOT NULL AND FSalary <5000
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
IT001	Smith	28	3900.00
SALES003	Stone	22	1200.00

4.2.3 反义运算符

“=”、“<”、“>”等运算符都是用来进行数值判断的，有的时候则会想使用这些运算符的反义，比如“不等于”、“不小于”或者“不大于”，MSSQLServer、DB2提供了“!”运算符来对运算符求反义，也就是“!=”表示“不等于”、“!<”表示“不小于”，而“!>”表示“不大于”。

比如要完成下面的功能“检索所有年龄不等于22岁并且工资不小于2000元”，我们可以编写下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FAge!=22 AND FSALARY!<2000
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNUMBER	FNAME	FAGE	FSALARY
DEV001	Tom	25	8300.00
DEV002	Jerry	28	2300.80
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT001	Smith	28	3900.00

“!”运算符能够把“不等于”、“不大于”、“不小于”这样的语义直接翻译成SQL运算符，不过这个运算符只在MSSQLServer和DB2两种数据库系统上提供支持，如果在其他数据库系统上则可以用其他的变通的方式实现，最常用的变通实现方式有两种：使用同义运算符、使用NOT运算符。

否定的语义都有对应的同义运算符，比如“不大于”的同义词是“小于等于”、而“不小于”的同义词是“大于等于”，同时SQL提供了通用的表示“不等于”的运算符“<>”，这样“不等于”、“不大于”和“不小于”就分别可以表示成“<>”、“<=”和“>=”。 因此要完成下面的功能“检索所有年龄不等于22岁并且工资不小于2000元”，我们可以编写下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FAge<>22 AND FSALARY>=2000
```

NOT运算符用来将一个表达式的值取反，也就是将值为“真”的表达式结果变为“假”、 将

值为“假”的表达式结果变为“真”，使用方式也非常简单“NOT (表达式)”，比如要表达“年龄不小于20”，那么可以如下使用“NOT(Fage<20)”。因此要完成下面的功能“检索所有年龄不等于22岁并且工资不小于2000元”，我们可以编写下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE NOT(FAge=22) AND NOT(FSALARY<2000)
```

使用“!”运算符的方式由于只能运行在MSSQLServer和DB2两种数据库系统上，所以如果应用程序有移植到其他数据库系统上的需求的话，就应该避免使用这种方式；使用同义运算符的方式能够运行在所有主流数据库系统上，不过由于粗心等原因，很容易将“不大于”表示成“<”，而忘记了“不大于”是包含“小于”和“等于”这两个意思的，这样就会造成检索数据的错误，造成应用程序的Bug；而采用NOT运算符的方式能比较容易的表达要实现的需求，而且能够实现复杂的嵌套，最重要的是避免了潜在的应用程序的Bug，所以除了“<>”这种方式之外，我们推荐使用NOT运算符的方式来表示“非”的语义。

4.2.4 多值检测

“公司要为年龄为23岁、25岁和28岁的员工发福利，请将他们的年龄、工号和姓名检索出来”，要完成这样的功能，我们可以使用OR语句来连接多个等于判断。SQL语句如下：

```
SELECT FAge,FNumber,FName FROM T_Employee
WHERE FAge=23 OR FAge=25 OR FAge=28
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge	FNumber	FName
25	DEV001	Tom
28	DEV002	Jerry
23	HR001	Jane
25	HR002	Tina
28	IT001	Smith
23	SALES001	John
28	SALES002	Kerry

这里要检索的年龄值是很少的，只有3个，如果要求我们“检索年龄为21岁、22岁、25岁、28岁、30岁、33岁、35岁、38岁、46岁的员工信息”，那么我们就要用OR连接九个等于判断：

```
SELECT FAge,FNumber,FName FROM T_Employee
WHERE FAge=21 OR FAge=22 OR FAge=25
OR FAge=28 OR FAge=30 OR FAge=33
OR FAge=35 OR FAge=38 OR FAge=46
```

这不仅写起来是非常麻烦的，而且维护的难度也相当大，一不小心就会造成数据错误。为了解决进行多个离散值的匹配问题，SQL提供了IN语句，使用IN我们只要指定要匹配的数据集合就可以了，使用方法为“IN (值1,值2,值3……)”。要完成“公司要为年龄为23岁、25岁和28岁的员工发福利，请将他们的年龄、工号和姓名检索出来”这样功能的话，可以使用下面的SQL语句：

```
SELECT FAge,FNumber,FName FROM T_Employee
WHERE FAge IN (23,25,28)
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge	FNumber	FName
25	DEV001	Tom
28	DEV002	Jerry
23	HR001	Jane

25	HR002	Tina
28	IT001	Smith
23	SALES001	John
28	SALES002	Kerry

可以看到执行结果和使用OR语句来连接多个等于判断的方式是一样的。

使用IN我们还可以让字段与其他表中的值进行匹配，比如“查找所有姓名在迟到记录表中的员工信息”，要实现这样的功能就需要IN来搭配子查询来使用，关于这一点我们将在后面的章节介绍。

4.2.5 范围值检测

使用IN语句只能进行多个离散值的检测，如果要实现范围值的检测就非常麻烦甚至不可能了。比如我们要完成下面的功能“检索所有年龄介于23岁到27岁之间的员工信息”，如果用IN语句来实现的话就必须列出此范围内的所有可能的值，SQL如下：

```
SELECT * FROM T_Employee
WHERE FAGE IN(23,24,25,26,27)
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT002	<NULL>	27	2800.00
SALES001	John	23	5000.00

当范围内的值比较多的时候使用这种方式非常麻烦，比如“检索所有年龄介于20岁到60岁之间的员工信息”就要列出20到60之间的每一个值，这个工作量是非常大的。而且这种方式也无法表达非离散的范围值，比如要实现“检索所有工资介于3000元到5000元之间的员工信息”的话就是不可能的，因为介于3000到5000之间的值是无数的。

这种情况下我们可以使用普通的“大于等于”和“小于等于”来实现范围值检测，比如完成下面的功能“检索所有年龄介于23岁到27岁之间的员工信息”，可以使用下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FAGE>=23 AND FAGE <=27
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT002	<NULL>	27	2800.00
SALES001	John	23	5000.00

这种方式能够实现几乎所有的范围值检测的功能，不过SQL提供了一个专门用语范围值检测的语句“BETTWEEN AND”，它可以用来检测一个值是否处于某个范围中（包括范围的边界值，也就是闭区间）。使用方法如下“字段名 BETTWEEN 左范围值 AND 右范围值”，其等价于“字段名>=左范围值 AND 字段名<=右范围值”。 比如完成下面的功能“检索所有年龄介于23岁到27岁之间的员工信息”，可以使用下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FAGE BETWEEN 23 AND 27
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT002	<NULL>	27	2800.00
SALES001	John	23	5000.00

使用“BETTWEEN AND”我们还能够进行多个不连续范围值的检测，比如要实现“检索所有工资介于2000元到3000元之间以及5000元到8000元的员工信息”，可以使用下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE (FSalary BETWEEN 2000 AND 3000)
OR (FSalary BETWEEN 5000 AND 8000)
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT002	<NULL>	27	2800.00
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00

数据库系统对“BETTWEEN AND”进行了查询优化，使用它进行范围值检测将会得到比其他方式更好的性能，因此在进行范围值检测的时候应该优先使用“BETTWEEN AND”。需要注意的就是“BETTWEEN AND”在进行检测的时候是包括了范围的边界值的（也就是闭区间），如果需要进行开区间或者半开半闭区间的范围值检测的话就必须使用其他的解决方案了。

4.2.6 低效的“WHERE 1=1”

网上有不少人提出过类似的问题：“看到有人写了WHERE 1=1这样的SQL，到底是什么意思？”。其实使用这种用法的开发人员一般都是在使用动态组装的SQL。

让我们想像如下的场景：用户要求提供一个灵活的查询界面来根据各种复杂的条件来查询员工信息，界面如下图：

界面中列出了四个查询条件，包括按工号查询、按姓名查询、按年龄查询以及按工资查询，

每个查询条件前都有一个复选框，如果复选框被选中，则表示将其做为一个过滤条件。比如上图就表示“检索工号介于DEV001和DEV008之间、姓名中含有J并且工资介于3000元到6000元的员工信息”。如果不选中姓名前的复选框，比如下图表示“检索工号介于DEV001和DEV008之间并且工资介于3000元到6000元的员工信息”：

如果将所有的复选框都不选中，则表示表示“检索所有员工信息”，比如下图：

这里的数据检索与前面的数据检索都不一样，因为前边例子中的数据检索的过滤条件都是确定的，而这里的过滤条件则随着用户设置的不同而有变化，这时就要根据用户的设置来动态组装SQL了。当不选中年龄前的复选框的时候要使用下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FNumber BETWEEN 'DEV001' AND 'DEV008'
AND FName LIKE '%J%'
AND FSalary BETWEEN 3000 AND 6000
```

而如果不选中姓名和年龄前的复选框的时候就要使用下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FNumber BETWEEN 'DEV001' AND 'DEV008'
AND FSalary BETWEEN 3000 AND 6000
```

而如果将所有的复选框都不选中的时候就要使用下面的SQL语句：

```
SELECT * FROM T_Employee
```

要实现这种动态的SQL语句拼装，我们可以在宿主语言中建立一个字符串，然后逐个判断各个复选框是否选中来向这个字符串中添加SQL语句片段。这里有一个问题就是当有复选框被选中的时候SQL语句是含有WHERE子句的，而当所有的复选框都没有被选中的时候就没有WHERE子句了，因此在添加每一个过滤条件判断的时候都要判断是否已经存在WHERE语句了，如果没有WHERE语句则添加WHERE语句。在判断每一个复选框的时候都要去判断，这使得用起来非常麻烦，“聪明的程序员是会偷懒的程序员”，因此开发人员想到了一个捷径：为SQL语句指定一个永远为真的条件语句（比如“1=1”），这样就不用考虑WHERE语句是否存在的问题了。伪代码如下[6]：

```
String sql = " SELECT * FROM T_Employee WHERE 1=1";
if(工号复选框选中)
{
    sql.appendLine("AND FNumber BETWEEN '"+工号文本框1内容+"' AND '"+工号文本框2内容+"'");
}
if(姓名复选框选中)
{
    sql.appendLine("AND FName LIKE '%"+姓名文本框内容+"%'");
}
if(年龄复选框选中)
{
    sql.appendLine("AND FAge BETWEEN "+年龄文本框1内容+" AND "+年龄文本框2内容);
}
executeSQL(sql);
```

这样如果不选中姓名和年龄前的复选框的时候就会执行下面的SQL语句：

```
SELECT * FROM T_Employee WHERE 1=1
AND FNumber BETWEEN 'DEV001' AND 'DEV008'
AND FSalary BETWEEN 3000 AND 6000
```

而如果将所有的复选框都不选中的时候就会执行下面的SQL语句：

```
SELECT * FROM T_Employee WHERE 1=1
```

这看似非常优美的解决了问题，殊不知这样很可能会造成非常大的性能损失，因为使用添加了“1=1”的过滤条件以后数据库系统就无法使用索引等查询优化策略，数据库系统将会被迫对每行数据进行扫描（也就是全表扫描）以比较此行是否满足过滤条件，当表中数据量比较大的时候查询速度会非常慢。因此如果数据检索对性能有比较高的要求就不要使用这种“简便”的方式。下面给出一种参考实现，伪代码如下：

```
private void doQuery()
{
    Bool hasWhere = false;
    StringBuilder sql = new StringBuilder(" SELECT * FROM T_Employee");
    if(工号复选框选中)
    {
        hasWhere = appendWhereIfNeed(sql, hasWhere);
```

[6] 这里演示的将检索参数值直接拼接到 SQL 中的做法是有一定的问题的，会造成性能问题以及注入漏洞攻击。为了降低问题的复杂度，这里规避了这个问题，在本书的后续章节将会详细讲解。

```
        sql.appendLine("FNumber BETWEEN '"+工号文本框1内容+"' AND '"+工号
        文本框2内容+"'");
    }
    if(姓名复选框选中)
    {
        hasWhere = appendWhereIfNeed(sql, hasWhere);
        sql.appendLine("FName LIKE '%"+姓名文本框内容+"%'");
    }
    if(年龄复选框选中)
    {
        hasWhere = appendWhereIfNeed(sql, hasWhere);
        sql.appendLine("FAge BETWEEN "+年龄文本框1内容+" AND "+年龄文本框2
    内容);
    }
    executeSQL(sql);
}
private Bool appendWhereIfNeed(StringBuilder sql,Bool hasWhere)
{
    if(hasWhere==false)
    {
        sql. appendLine("WHERE");
    }
    else
    {
        sql. appendLine("AND");
    }
}
```

4.3 数据分组

前面我们讲解了聚合函数的使用，比如要查看年龄为23岁员工的人数，只要执行下面的SQL就可以：

```
SELECT COUNT(*) FROM T_Employee
WHERE FAge=23
```

可是如果我们想查看每个年龄段的员工的人数怎么办呢？一个办法是先得到所有员工的年龄段信息，然后分别查询每个年龄段的人数，显然这样是非常低效且烦琐的。这时候就是数组分组开始显现威力的时候了。

为了更好的演示本节中的例子，我们为T_Employee表增加两列，分别为表示其所属分公司的FSubCompany字段和表示其所属部门的FDepartment，在不同的数据库下执行相应的SQL语句：

```
MYSQL,MSSQLServer,DB2:
ALTER TABLE T_Employee ADD FSubCompany VARCHAR(20);
ALTER TABLE T_Employee ADD FDepartment VARCHAR(20);
Oracle:
ALTER TABLE T_Employee ADD FSubCompany VARCHAR2(20);
ALTER TABLE T_Employee ADD FDepartment VARCHAR2(20);
```

两个字段添加完毕后还需要将表中原有数据行的这两个字段值更新，执行下面的SQL语句：

```
UPDATE T_Employee SET FSubCompany='Beijing',FDepartment='Development'
WHERE FNumber='DEV001';
UPDATE T_Employee SET FSubCompany='ShenZhen',FDepartment='Development'
WHERE FNumber='DEV002';
UPDATE T_Employee SET
FSubCompany='Beijing',FDepartment='HumanResource'
WHERE FNumber='HR001';
UPDATE T_Employee SET
FSubCompany='Beijing',FDepartment='HumanResource'
WHERE FNumber='HR002';
UPDATE T_Employee SET FSubCompany='Beijing',FDepartment='InfoTech'
WHERE FNumber='IT001';
UPDATE T_Employee SET FSubCompany='ShenZhen',FDepartment='InfoTech'
WHERE FNumber='IT002';
UPDATE T_Employee SET FSubCompany='Beijing',FDepartment='Sales'
WHERE FNumber='SALES001';
UPDATE T_Employee SET FSubCompany='Beijing',FDepartment='Sales'
WHERE FNumber='SALES002';
UPDATE T_Employee SET FSubCompany='ShenZhen',FDepartment='Sales'
WHERE FNumber='SALES003';
```

4.3.1 数据分组入门

数据分组用来将数据分为多个逻辑组，从而可以对每个组进行聚合运算。SQL语句中使用GROUP BY子句进行分组，使用方式为“GROUP BY 分组字段”。分组语句必须和聚合函数一起使用，GROUP BY子句负责将数据分成逻辑组，而聚合函数则对每一个组进行统计计算。

虽然GROUP BY子句常常和聚合函数一起使用，不过GROUP BY子句并不是不能离开聚合函数而单独使用的，虽然不使用聚合函数的GROUP BY子句看起来用处不大，不过它能够帮助我们更好的理解数据分组的原理，所以本小节我们将演示GROUP BY子句的分组能力。

我们首先来看一下如果通过SQL语句实现“查看公司员工有哪些年龄段的”，因为这里只需要列出员工的年龄段，所以使用GROUP BY子句就完全可以实现：

```
SELECT FAge FROM T_Employee
GROUP BY FAge
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge
22
23
25
27
28

这个SQL语句处理表中的所有记录，并且将FAge相同的数据行放到一组，分组后的数据可以看作一个临时的结果集，而SELECT FAge语句则取出每组的FAge字段的值，这样我们就得到上表的员工年龄段表了。

GROUP BY子句将检索结果划分为多个组，每个组是所有记录的一个子集。上面的SQL例子在执行的时候数据库系统将数据分成了下面的分组：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment	分组
SALES003	Stone	22	1200.00	ShenZhen	Sales	22 岁组
SALES001	John	23	5000.00	Beijing	Sales	23 岁组
HR001	Jane	23	2200.88	Beijing	HumanResource	
HR002	Tina	25	5200.36	Beijing	HumanResource	25 岁组
DEV001	Tom	25	8300.00	Beijing	Development	
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech	27 岁组
SALES002	Kerry	28	6200.00	Beijing	Sales	28 岁组
DEV002	Jerry	28	2300.80	ShenZhen	Development	
IT001	Smith	28	3900.00	Beijing	InfoTech	

需要注意的是GROUP BY子句的位置，GROUP BY子句必须放到SELECT语句的之后，如果SELECT语句有WHERE子句，则GROUP BY子句必须放到WHERE语句的之后。比如下面的SQL语句是错误的：

```
SELECT FAge FROM T_Employee
GROUP BY FAge
WHERE FSubCompany = 'Beijing'
```

而下面的SQL语句则是正确的：

```
SELECT FAge FROM T_Employee
WHERE FSubCompany = 'Beijing'
GROUP BY FAge
```

需要分组的所有列都必须位于GROUP BY子句的列名列表中，也就是没有出现在GROUP BY子句中的列（聚合函数除外）是不能放到SELECT语句后的列名列表中的。比如下面的SQL语句是错误的：

```
SELECT FAge,FSalary FROM T_Employee
GROUP BY FAge
```

道理非常简单，因为采用分组以后的查询结果集是以分组形式提供的，由于每组中人员的员工工资都不一样，所以就不存在能够统一代表本组工资水平的FSalary字段了，所以上面的SQL语句是错误的。不过每组中员工的平均工资却是能够代表本组统一工资水平的，所以可以对FSalary使用聚合函数，下面的SQL语句是正确的：

```
SELECT FAge,AVG(FSalary) FROM T_Employee
GROUP BY FAge
```

GROUP BY子句中可以指定多个列，只需要将多个列的列名用逗号隔开即可。指定多个分组规则以后，数据库系统将按照定义的分组顺序来对数据进行逐层分组，首先按照第一个分组列进行分组，然后在每个小组内按照第二个分组列进行再次分组……逐层分组，从而实现“组中组”的效果，而查询的结果集是以最末一级分组来进行输出的。比如下面的SQL语句将会列出所有分公司的所有部门情况：

```
SELECT FSubCompany,FDepartment FROM T_Employee
GROUP BY FSubCompany,FDepartment
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FSubCompany	FDepartment
Beijing	Development
Beijing	HumanResource

Beijing	InfoTech
Beijing	Sales
ShenZhen	Development
ShenZhen	InfoTech
ShenZhen	Sales

上面的SQL例子在执行的时候数据库系统将数据分成了下面的分组：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment	FSubCompany 分组	FDepartment 分组
DEV001	Tom	25	8300.00	Beijing	Development	Beijing 组	Development 组
HR001	Jane	23	2200.88	Beijing	HumanResource		HumanResource 组
HR002	Tina	25	5200.36	Beijing	HumanResource		InfoTech 组
IT001	Smith	28	3900.00	Beijing	InfoTech		Sales 组
SALES001	John	23	5000.00	Beijing	Sales		
SALES002	Kerry	28	6200.00	Beijing	Sales		
DEV002	Jerry	28	2300.80	ShenZhen	Development	ShenZhen 组	Development 组
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech		InfoTech 组
SALES003	Stone	22	1200.00	ShenZhen	Sales		Sales 组

4.3.2 数据分组与聚合函数

到目前为止我们使用的聚合函数都是对普通结果集进行统计的，我们同样可以使用聚合函数来对分组后的数据进行统计，也就是统计每一个分组的数据。我们甚至可以认为在没有使用GROUP BY语句中使用聚合函数不过是在一个整个结果集是一个组的分组数据中进行数据统计分析罢了。

让我们来看一下“查看每个年龄段的员工的人数”如何用数据分组来实现，下面是实现此功能的SQL语句：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
GROUP BY FAge
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge	CountOfThisAge
22	1
23	2
25	2
27	1
28	3

GROUP BY子句将检索结果按照年龄划分为多个组，每个组是所有记录的一个子集。上面的SQL例子在执行的时候数据库系统将数据分成了下面的分组：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment	分组
SALES003	Stone	22	1200.00	ShenZhen	Sales	22 岁组
SALES001	John	23	5000.00	Beijing	Sales	23 岁组
HR001	Jane	23	2200.88	Beijing	HumanResource	
HR002	Tina	25	5200.36	Beijing	HumanResource	25 岁

DEV001	Tom	25	8300.00	Beijing	Development	组
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech	27 岁组
SALES002	Kerry	28	6200.00	Beijing	Sales	28 岁组
DEV002	Jerry	28	2300.80	ShenZhen	Development	
IT001	Smith	28	3900.00	Beijing	InfoTech	

可以看到年龄相同的员工被分到了一组，接着使用“**COUNT**(*)”来统计每一组中的条数，这样就得到了每个年龄段的员工的个数了。

可以使用多个分组来实现更精细的数据统计，比如下面的SQL语句就可以统计每个分公司的年龄段的人数：

```
SELECT FSubCompany,FAge,COUNT(*) AS CountOfThisSubCompAge FROM
T_Employee
GROUP BY FSubCompany,FAge
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FSubCompany	FAge	CountOfThisAge
ShenZhen	22	1
Beijing	23	2
Beijing	25	2
ShenZhen	27	1
Beijing	28	2
ShenZhen	28	1

上面的执行结果是按照数据库系统默认的年龄进行排序的，为了更容易的按照每个分公司进行查看，我们可以指定按照FSubCompany字段进行排序，带ORDER BY的SQL语句如下：

```
SELECT FSubCompany,FAge,COUNT(*) AS CountOfThisSubCompAge FROM
T_Employee
GROUP BY FSubCompany,FAge
ORDER BY FSubCompany
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FSubCompany	FAge	CountOfThisSubCompAge
Beijing	23	2
Beijing	25	2
Beijing	28	2
ShenZhen	22	1
ShenZhen	27	1
ShenZhen	28	1

上面的SQL语句中，GROUP BY子句将检索结果首先按照FSubCompany进行分组，然后在每一个分组内又按照FAge进行分组，数据库系统将数据分成了下面的分组：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment	FSubCompany 分组	FAge 分组
HR001	Jane	23	2200.88	Beijing	HumanResource	Beijing 组	23 岁组
SALES001	John	23	5000.00	Beijing	Sales		
DEV001	Tom	25	8300.00	Beijing	Development		25 岁组
HR002	Tina	25	5200.36	Beijing	HumanResource		

IT001	Smith	28	3900.00	Beijing	InfoTech		28 岁组
SALES002	Kerry	28	6200.00	Beijing	Sales		
SALES003	Stone	22	1200.00	ShenZhen	Sales	ShenZhen 组	22 岁组
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech		27 岁组
DEV002	Jerry	28	2300.80	ShenZhen	Development		28 岁组

“COUNT(*)”对每一个分组统计总数，这样就可以统计出每个公司每个年龄段的员工的人数了。

SUM、AVG、MIN、MAX也可以在分组中使用。比如下面的SQL可以统计每个公司中的工资的总值：

```
SELECT FSubCompany,SUM(FSalary) AS FSalarySUM FROM T_Employee
GROUP BY FSubCompany
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FSubCompany	FSalarySUM
Beijing	30801.24
ShenZhen	6300.80

下面的SQL可以统计每个垂直部门中的工资的平均值：

```
SELECT FDepartment,SUM(FSalary) AS FSalarySUM FROM T_Employee
GROUP BY FDepartment
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FDepartment	FSalarySUM
Development	10600.80
HumanResource	7401.24
InfoTech	6700.00
Sales	12400.00

下面的SQL可以统计每个垂直部门中员工年龄的最大值和最小值：

```
SELECT FDepartment,MIN(FAge) AS FAgeMIN,MAX(FAge) AS FAgeMAX FROM
T_Employee
GROUP BY FDepartment
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FDepartment	FAgeMIN	FAgeMAX
Development	25	28
HumanResource	23	25
InfoTech	27	28
Sales	22	28

4.3.3 HAVING 语句

有的时候需要对部分分组进行过滤，比如只检索人数多余1个的年龄段，有的开发人员会使用下面的SQL语句：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
GROUP BY FAge
WHERE COUNT(*)>1
```

可以在数据库系统中执行下面的SQL的时候，数据库系统会提示语法错误，这是因为聚合函

数不能在WHERE语句中使用，必须使用HAVING子句来代替，比如：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
GROUP BY FAge
HAVING COUNT(*)>1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge	CountOfThisAge
23	2
25	2
28	3

HAVING语句中也可以像WHERE语句一样使用复杂的过滤条件，比如下面的SQL用来检索人数为1个或者3个的年龄段，可以使用下面的SQL：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
GROUP BY FAge
HAVING COUNT(*) =1 OR COUNT(*) =3
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge	CountOfThisAge
22	1
27	1
28	3

也可以使用IN操作符来实现上面的功能，SQL语句如下：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
GROUP BY FAge
HAVING COUNT(*) IN (1,3)
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge	CountOfThisAge
22	1
27	1
28	3

HAVING语句能够使用的语法和WHERE几乎是一样的，不过使用WHERE的时候GROUP BY子句要位于WHERE子句之后，而使用HAVING子句的时候GROUP BY子句要位于HAVING子句之后，比如下面的SQL是错误的：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
HAVING COUNT(*) IN (1,3)
GROUP BY FAge
```

需要特别注意，在HAVING语句中不能包含未分组的列名，比如下面的SQL语句是错误的：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
GROUP BY FAge
HAVING FName IS NOT NULL
```

执行的时候数据库系统会提示类似如下的错误信息：

HAVING 子句中的列 'T_Employee.FName' 无效，因为该列没有包含在聚合函数或 GROUP BY 子句中。

需要用WHERE语句来代替HAVING，修改后的SQL语句如下：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
WHERE FName IS NOT NULL
GROUP BY FAge
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge	CountOfThisAge
22	1
23	2
25	2
28	3

4.4 限制结果集行数

在进行数据检索的时候有时候需要只检索结果集中的部分行，比如说“检索成绩排前三名的学生”、“检索工资水平排在第3位到第7位的员工信息”，这种功能被称为“限制结果集行数”。在虽然主流的数据库系统中都提供了限制结果集行数的方法，但是无论是语法还是使用方式都存在着很大的差异，即使是同一个数据库系统的不同版本（比如MSSQLServer2000和MSSQLServer2005）也存在着一定的差异。因此本节将按照数据库系统来讲解每种数据库系统对限制结果集行数的特性支持。

4.4.1 MYSQL

MYSQL中提供了LIMIT关键字用来限制返回的结果集，LIMIT放在SELECT语句的最后位置，语法为“LIMIT 首行行号，要返回的结果集的最大数目”。比如下面的SQL语句将返回按照工资降序排列的从第二行开始（行号从0开始）的最多五条记录：

```
SELECT * FROM T_Employee ORDER BY FSalary DESC LIMIT 2,5
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment
HR002	Tina	25	5200.36	Beijing	HumanResource
SALES001	John	23	5000.00	Beijing	Sales
IT001	Smith	28	3900.00	Beijing	InfoTech
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech
DEV002	Jerry	28	2300.80	ShenZhen	Development

很显然，下面的SQL语句将返回按照工资降序排列的前五条记录：

```
SELECT * FROM T_Employee ORDER BY FSalary DESC LIMIT 0,5
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment
DEV001	Tom	25	8300.00	Beijing	Development
SALES002	Kerry	28	6200.00	Beijing	Sales
HR002	Tina	25	5200.36	Beijing	HumanResource
SALES001	John	23	5000.00	Beijing	Sales
IT001	Smith	28	3900.00	Beijing	InfoTech

4.4.2 MSSQLServer2000

MSSQLServer2000中提供了TOP关键字用来返回结果集中的前N条记录，其语法为“SELECT TOP 限制结果集数目 字段列表 SELECT语句其余部分”，比如下面的SQL语句用来检索工资水平排在前五位（按照工资从高到低）的员工信息：

```
select top 5 * from T_Employee order by FSalary Desc
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment
DEV001	Tom	25	8300.00	Beijing	Development
SALES002	Kerry	28	6200.00	Beijing	Sales

HR002	Tina	25	5200.36	Beijing	HumanResource
SALES001	John	23	5000.00	Beijing	Sales
IT001	Smith	28	3900.00	Beijing	InfoTech

MSSQLServer2000没有直接提供返回提供“检索从第5行开始的10条数据”、“检索第五行至第十二行的数据”等这样的取区间范围的功能，不过可以采用其他方法来变通实现，最常使用的方法就是用子查询[7]，比如要实现检索按照工资从高到低排序检索从第六名开始一共三个人的信息，那么就可以首先将前五名的主键取出来，在检索的时候检索排除了这五名员工的前三个人，SQL如下：

```
SELECT top 3 * FROM T_Employee
WHERE FNumber NOT IN
(SELECT TOP 5 FNumber FROM T_Employee ORDER BY FSalary DESC)
 ORDER BY FSalary DESC
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech
DEV002	Jerry	28	2300.80	ShenZhen	Development
HR001	Jane	23	2200.88	Beijing	HumanResource

4.4.3 MSSQLServer2005

MSSQLServer2005兼容几乎所有的MSSQLServer2000的语法，所以可以使用上个小节提到的方式来在MSSQLServer2005中实现限制结果集行数，不过MSSQLServer2005提供了新的特性来帮助更好的限制结果集行数的功能，这个新特性就是窗口函数ROW_NUMBER()。

ROW_NUMBER()函数可以计算每一行数据在结果集中的行号（从1开始计数），其使用语法如下：

```
ROW_NUMBER OVER(排序规则)
```

比如我们执行下面的SQL语句：

```
SELECT ROW_NUMBER() OVER(ORDER BY FSalary),FNumber,FName,FSalary,FAge
FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

	FNumber	FName	FSalary	FAge
1	DEV001	Tom	8300.00	25
2	SALES002	Kerry	6200.00	28
3	HR002	Tina	5200.36	25
4	SALES001	John	5000.00	23
5	IT001	Smith	3900.00	28
6	IT002	<NULL>	2800.00	27
7	DEV002	Jerry	2300.80	28
8	HR001	Jane	2200.88	23
9	SALES003	Stone	1200.00	22

可以看到第一列中的数据就是通过ROW_NUMBER()计算出来的行号。有的开发人员想使用如下的方式来实现返回第3行到第5行的数据（按照工资降序）：

```
SELECT ROW_NUMBER() OVER(ORDER BY FSalary
```

[7] 在本书的后面章节将会详细介绍子查询。对于子查询不熟悉的读者可以暂时认为子查询就是将另外一个查询结果当作一个新的表进行检索。

```
DESC),FNumber,FName,FSalary,FAge
FROM T_Employee
WHERE (ROW_NUMBER() OVER(ORDER BY FSalary DESC))>=3
AND (ROW_NUMBER() OVER(ORDER BY FSalary DESC))<=5
```

但是在运行的时候数据库系统会报出下面的错误信息：

开窗函数只能出现在 SELECT 或 ORDER BY 子句中。

也就是说ROW_NUMBER()不能用在WHERE语句中。我们可以用子查询来解决这个问题，下面的SQL语句用来返回第3行到第5行的数据：

```
SELECT * FROM
(
SELECT ROW_NUMBER() OVER(ORDER BY FSalary DESC) AS rownum,
FNumber,FName,FSalary,FAge FROM T_Employee
) AS a
WHERE a.rownum>=3 AND a.rownum<=5
```

执行完毕我们就能在输出结果中看到下面的执行结果：

rownum	FNumber	FName	FSalary	FAge
3	HR002	Tina	5200.36	25
4	SALES001	John	5000.00	23
5	IT001	Smith	3900.00	28

4.4.4 Oracle

Oracle中支持窗口函数ROW_NUMBER()，其用法和MSSQLServer2005中相同，比如我们执行下面的SQL语句：

```
SELECT * FROM
(
SELECT ROW_NUMBER() OVER(ORDER BY FSalary DESC) row_num,
FNumber,FName,FSalary,FAge FROM T_Employee
) a
WHERE a.row_num>=3 AND a.row_num<=5
```

执行完毕我们就能在输出结果中看到下面的执行结果：

ROW_NUM	FNUMBER	FNAME	FSALARY	FAGE
3	HR002	Tina	5200.36	25
4	SALES001	John	5000	23
5	IT001	Smith	3900	28

注意：rownum在Oracle中为保留字，所以这里将MSSQLServer2005中用到的rownum替换为row_num；Oracle中定义表别名的时候不能使用AS关键字，所以这里也去掉了AS。

Oracle支持标准的函数ROW_NUMBER()，不过Oracle中提供了更方便的特性用来计算行号，也就在Oracle中可以无需自行计算行号，Oracle为每个结果集都增加了一个默认的表示行号的列，这个列的名称为rownum。比如我们执行下面的SQL语句：

```
SELECT rownum,FNumber,FName,FSalary,FAge FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

ROWNUM	FNUMBER	FNAME	FSALARY	FAGE
1	DEV001	Tom	8300	25
2	DEV002	Jerry	2300.8	28
3	SALES001	John	5000	23

4	SALES002	Kerry	6200	28
5	SALES003	Stone	1200	22
6	HR001	Jane	2200.88	23
7	HR002	Tina	5200.36	25
8	IT001	Smith	3900	28
9	IT002	<NULL>	2800	27

使用rownum我们可以很轻松的取得结果集中前N条的数据行，比如我们执行下面的SQL语句可以得到按工资从高到底排序的前6名员工的信息：

```
SELECT * FROM T_Employee
WHERE rownum<=6
ORDER BY FSalary Desc
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNUMBER	FNAME	FAGE	FSALARY	FSUBCOMPANY	FDEPARTMENT
DEV001	Tom	25	8300	Beijing	Development
SALES002	Kerry	28	6200	Beijing	Sales
SALES001	John	23	5000	Beijing	Sales
DEV002	Jerry	28	2300.8	ShenZhen	Development
HR001	Jane	23	2200.88	Beijing	HumanResource
SALES003	Stone	22	1200	ShenZhen	Sales

看到这里，您可能认为下面的SQL就可以非常容易的实现“按照工资从高到低的顺序取出第三个到第五个员工信息”的功能了：

```
SELECT rownum,FNumber,FName,FSalary,FAge FROM T_Employee
WHERE rownum BETWEEN 3 AND 5
ORDER BY FSalary DESC
```

执行完毕我们就能在输出结果中看到下面的执行结果：

ROWNUM	FNUMBER	FNAME	FSALARY	FAGE

检索结果为空！！！这非常出乎我们的意料。让我们来回顾一下rownum的含义：rownum为结果集中每一行的行号（从1开始计数）。对于下面的SQL：

```
SELECT * FROM T_Employee
WHERE rownum<=6
ORDER BY FSalary Desc
```

当进行检索的时候，对于第一条数据，其rownum为1，因为符合“WHERE rownum<=6”所以被放到了检索结果中；当检索到第二条数据的时候，其rownum为2，因为符合“WHERE rownum<=6”所以被放到了检索结果中……依次类推，直到第七行。所以这句SQL语句能够实现“按照工资从高到低的顺序取出第三个到第五个员工信息”的功能。

而对于这句SQL语句：

```
SELECT rownum,FNumber,FName,FSalary,FAge FROM T_Employee
WHERE rownum BETWEEN 3 AND 5
ORDER BY FSalary DESC
```

当进行检索的时候，对于第一条数据，其rownum为1，因为不符合“WHERE rownum BETWEEN 3 AND 5”，所以没有被放到了检索结果中；当检索到第二条数据的时候，因为第一条数据没有放到结果集中，所以第二条数据的rownum仍然为1，而不是我们想像的2，所以因为不符合“WHERE rownum<=6”，没有被放到了检索结果中；当检索到第三条数据的时候，因为第一、二条数据没有放到结果集中，所以第三条数据的rownum仍然为1，而不是我们想像

的3，所以因为不符合“WHERE rownum<=6”，没有被放到了检索结果中……依此类推，这样所有的数据行都没有被放到结果集中。

因此如果要使用rownum来实现“按照工资从高到低的顺序取出第三个到第五个员工信息”的功能，就必须借助于窗口函数ROW_NUMBER()。

4.4.5 DB2

DB2中支持窗口函数ROW_NUMBER()，其用法和MSSQLServer2005以及Oracle中相同，比如我们执行下面的SQL语句：

```
SELECT * FROM
(
SELECT ROW_NUMBER() OVER(ORDER BY FSalary DESC) row_num,
FNumber,FName,FSalary,FAge FROM T_Employee
) a
WHERE a.row_num>=3 AND a.row_num<=5
```

执行完毕我们就能在输出结果中看到下面的执行结果：

ROW_NUM	FNUMBER	FNAME	FSALARY	FAGE
3	HR002	Tina	5200.36	25
4	SALES001	John	5000.00	23
5	IT001	Smith	3900.00	28

除此之外，DB2还提供了FETCH关键字用来提取结果集的前N行，其语法为“FETCH FIRST 条数 ROWS ONLY”，比如我们执行下面的SQL语句可以得到按工资从高到底排序的前6名员工的信息：

```
SELECT * FROM T_Employee
ORDER BY FSalary Desc
FETCH FIRST 6 ROWS ONLY
```

需要注意的是FETCH子句要放到ORDER BY语句的后面，执行完毕我们就能在输出结果中看到下面的执行结果：

FNUMBER	FNAME	FAGE	FSALARY	FSUBCOMPANY	FDEPARTMENT
DEV001	Tom	25	8300.00	Beijing	Development
SALES002	Kerry	28	6200.00	Beijing	Sales
HR002	Tina	25	5200.36	Beijing	HumanResource
SALES001	John	23	5000.00	Beijing	Sales
IT001	Smith	28	3900.00	Beijing	InfoTech
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech

DB2没有直接提供返回提供“检索从第5行开始的10条数据”、“检索第五行至第十二行的数据”等这样的取区间范围的功能，不过可以采用其他方法来变通实现，最常使用的方法就是用子查询，比如要实现检索按照工资从高到低排序检索从第六名开始一共三个人的信息，那么就可以首先将前五名的主键取出来，在检索的时候检索排除了这五名员工的前三个人，SQL如下：

```
SELECT * FROM T_Employee
WHERE FNumber NOT IN
(
SELECT FNumber FROM T_Employee
ORDER BY FSalary DESC
FETCH FIRST 5 ROWS ONLY
```

```
)
ORDER BY FSalary DESC
FETCH FIRST 3 ROWS ONLY
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNUMBER	FNAME	FAGE	FSALARY	FSUBCOMPANY	FDEPARTMENT
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech
DEV002	Jerry	28	2300.80	ShenZhen	Development
HR001	Jane	23	2200.88	Beijing	HumanResource

4.4.6 数据库分页

在进行信息检索的时候，检索结果的数量通常会非常多，达到成百上千条，甚至更多，这么多的检索结果同时显示在同一个界面中，不仅查看起来非常麻烦，而且过多的数据显示在界面上也会造成占用过多的系统资源。解决这个问题的最常用方案就是数据库分页，就像我们在论坛上查看帖子的时候，一个网页中只显示50条帖子，论坛会提供【上一页】、【下一页】、【首页】以及【尾页】等按钮用来显示不同的页。实现数据库分页的核心技术就是“限制结果集行数”。

假设每一页显示的数据条数为PageSize，当前页数（从0开始技术）为CurrentIndex，那么我们只要查询从第PageSize*CurrentIndex开始的PageSize条数据得到的结果就是当前页中的数据；当用户点击【上一页】按钮的时候，将CurrentIndex设置为CurrentIndex-1，然后重新检索；当用户点击【下一页】按钮的时候，将CurrentIndex设置为CurrentIndex+1，然后重新检索；当用户点击【首页】按钮的时候，将CurrentIndex设置为0，然后重新检索；当用户点击【首页】按钮的时候，将CurrentIndex设置为“总条数/PageSize”，然后重新检索。

下面我们将要用伪代码来演示数据库分页功能，其中的数据库系统使用的MYSQL：

```
int CurrentIndex=0;
PageSize=10;
//按钮【首页】被点击
private void btnFirstButtonClick()
{
    CurrentIndex=0;
    DoSearch();
}

//按钮【尾页】被点击
private void btnLastButtonClick()
{
    CurrentIndex=GetTotalCount()/PageSize;
    DoSearch();
}

//按钮【下一页】被点击
private void btnNextButtonClick()
{
    CurrentIndex= CurrentIndex+1;
    DoSearch();
}
```

```
//按钮【上一页】被点击
private void btnNextButtonClick( )
{
    CurrentIndex= CurrentIndex-1;
    DoSearch();
}

//计算表中的总数据条数
private int GetTotalCount()
{
    ResultSet rs = ExecuteSQL("SELECT COUNT(*) AS TOTALCOUNT FROM T_Employee ");
    return rs.getInt("TOTALCOUNT");
}

//查询当前页中的数据
private void DoSearch()
{
    //计算当前页的起始行数
    String startIndex = (CurrentIndex* PageSize).ToString();
    String size = PageSize.ToString()
    ResultSet rs = ExecuteSQL("SELECT * FROM T_Employee   LIMIT "
                                + startIndex +","+ size);
    //显示查询结果
    DisplayResult(rs);
}
```

4.5 抑制数据重复

如果要检索公司里有哪些垂直部门，那么可以执行下面的SQL语句：

```
SELECT FDepartment FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FDepartment
Development
Development
HumanResource
HumanResource
InfoTech
InfoTech
Sales
Sales
Sales

这里列出了公司所有的垂直部门，不过很多部门名称是重复的，我们必须去掉这些重复的部门名称，每个重复部门只保留一个名称。**DISTINCT关键字是用来进行重复数据抑制的最简单的功能，而且所有的数据库系统都支持DISTINCT，DISTINCT的使用也非常简单，只要在SELECT之后增加DISTINCT即可。比如下面的SQL语句用于**检索公司里有哪些垂直部门，并且

抑制了重复数据的产生：

```
SELECT DISTINCT FDepartment FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FDepartment
Development
HumanResource
InfoTech
Sales

DISTINCT是对整个结果集进行数据重复抑制的，而不是针对每一个列，执行下面的SQL语句：

```
SELECT DISTINCT FDepartment,FSubCompany FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FDepartment	FSubCompany
Development	Beijing
Development	ShenZhen
HumanResource	Beijing
InfoTech	Beijing
InfoTech	ShenZhen
Sales	Beijing
Sales	ShenZhen

检索结果中不存在FDepartment和FSubCompany列都重复的数据行，但是却存在FDepartment列重复的数据行，这就验证了“**DISTINCT是对整个结果集进行数据重复抑制的**”这句话。

4.6 计算字段

存在数据库系统中的数据的默认展现方式不一定完全符合应用的要求，比如：

- 数据库系统中姓名、工号是单独存储在两个字段的，但是在显示的时候想显示成“姓名+工号”的形式。
- 数据库系统中金额的显示格式是普通的数字显示方式（比如668186.99），但是显示的时候想以千分位的形式显示（比如668,186.99）。
- 数据库系统中基本工资、奖金是单独存储在两个字段的，但是希望显示员工的工资总额。
- 要检索工资总额的80%超过5000元的员工信息。
- 要升级员工工号，需要将所有员工的工号前增加两位0。

所有这些功能都不能通过简单的SQL语句来完成的，因为需要的数据不是数据表中本来就有的，必须经过一定的计算、转换或者格式化，这种情况下我们可以在宿主语言中通过编写代码的方式来进行这些计算、转换或者格式化的工作，但是可以想象当数据量比较大的时候这样处理的速度是非常慢的。计算字段是数据库系统提供的对数据进行计算、转换或者格式化的功能，由于是在数据库系统内部进行的这些工作，而且数据库系统都这些工作进行了优化，所以其处理效率比在宿主语言中通过编写代码的方式进行处理要高效的多。本节将介绍计算字段的基本使用以及在SELECT、Update、Delete等语句中的应用。

4.6.1 常量字段

软件协会要求各个公司提供所有员工的资料信息，其中包括公司名称、注册资本、员工姓名、年龄、所在子公司，而且出于特殊考虑，要求每个员工都列出这些资料信息。对于单个公司而言，公司名称、注册资本这两部分信息不是能从现有的T_Employee，但是它们是

确定的值，因此我们编写下面的SQL语句：

```
SELECT 'CowNew集团',918000000,FName,FAge,FSubCompany FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

		FName	FAge	FSubCompany
CowNew 集团	918000000	Tom	25	Beijing
CowNew 集团	918000000	Jerry	28	ShenZhen
CowNew 集团	918000000	Jane	23	Beijing
CowNew 集团	918000000	Tina	25	Beijing
CowNew 集团	918000000	Smith	28	Beijing
CowNew 集团	918000000	<NULL>	27	ShenZhen
CowNew 集团	918000000	John	23	Beijing
CowNew 集团	918000000	Kerry	28	Beijing
CowNew 集团	918000000	Stone	22	ShenZhen

这里的'CowNew集团'和918000000并不是一个实际的存在的列，但是在查询出来的数据中它们看起来是一个实际存在的字段，这样的字段被称为“常量字段”（也称为“常量值”），它们完全可以被看成一个值确定的字段，比如可以为常量字段指定别名，执行下面的SQL语句：

```
SELECT 'CowNew集团' AS CompanyName,918000000 AS RegAmount,FName,FAge,FSubCompany
FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CompanyName	RegAmount	FName	FAge	FSubCompany
CowNew 集团	918000000	Tom	25	Beijing
CowNew 集团	918000000	Jerry	28	ShenZhen
CowNew 集团	918000000	Jane	23	Beijing
CowNew 集团	918000000	Tina	25	Beijing
CowNew 集团	918000000	Smith	28	Beijing
CowNew 集团	918000000	<NULL>	27	ShenZhen
CowNew 集团	918000000	John	23	Beijing
CowNew 集团	918000000	Kerry	28	Beijing
CowNew 集团	918000000	Stone	22	ShenZhen

4.6.2 字段间计算

人力资源部要求统计全体员工的工资指数，工资指数的计算公式为年龄与工资的乘积，这就需要计算将FAge和FSalary的乘积做为一个工资指数列体现到检索结果中，执行下面的SQL语句：

```
SELECT FNumber,FName,FAge * FSalary FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	
DEV001	Tom	207500.00
DEV002	Jerry	64422.40
HR001	Jane	50620.24
HR002	Tina	130009.00
IT001	Smith	109200.00
IT002	<NULL>	75600.00

SALES001	John	115000.00
SALES002	Kerry	173600.00
SALES003	Stone	26400.00

同样，这里的FAge * FSalary并不是一个实际的存在的列，但是在查询出来的数据中它们看起来是一个实际存在的字段，它们完全可以被看成一个普通字段，比如可以为此字段指定别名，执行下面的SQL语句：

```
SELECT FNumber,FName,FAge * FSalary AS FSalaryIndex FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FSalaryIndex
DEV001	Tom	207500.00
DEV002	Jerry	64422.40
HR001	Jane	50620.24
HR002	Tina	130009.00
IT001	Smith	109200.00
IT002	<NULL>	75600.00
SALES001	John	115000.00
SALES002	Kerry	173600.00
SALES003	Stone	26400.00

前面提到常量字段完全可以当作普通字段来看待，那么普通字段也可以和常量字段进行计算，甚至常量字段之间也可以进行计算。比如人力资源部要求统计每个员工的工资幸福指数，工资幸福指数的计算公式为工资/（年龄-21），而且要求在每行数据前添加一列，这列的值等于125与521的和。我们编写下面的SQL：

```
SELECT 125+521,FNumber,FName,FSalary/(FAge-21) AS FHappyIndex
FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果[8]：

	FNumber	FName	FHappyIndex
646	DEV001	Tom	2075.0000000000000
646	DEV002	Jerry	328.6857142857142
646	HR001	Jane	1100.4400000000000
646	HR002	Tina	1300.0900000000000
646	IT001	Smith	557.1428571428571
646	IT002	<NULL>	466.6666666666666
646	SALES001	John	2500.0000000000000
646	SALES002	Kerry	885.7142857142857
646	SALES003	Stone	1200.0000000000000

计算字段也可以在WHERE语句等子句或者UPDATE、DELETE中使用。比如下面的SQL用来检索所有资幸福指数大于1000的员工信息：

```
SELECT * FROM T_Employee
WHERE FSalary/(FAge-21)>1000
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment
DEV001	Tom	25	8300.00	Beijing	Development

[8] 由于不同的数据库系统对于除法的精度处理不同，所以 FHappyIndex 的显示精度也不一样。

HR001	Jane	23	2200.88	Beijing	HumanResource
HR002	Tina	25	5200.36	Beijing	HumanResource
SALES001	John	23	5000.00	Beijing	Sales
SALES003	Stone	22	1200.00	ShenZhen	Sales

4.6.3 数据处理函数

像普通编程语言一样，SQL也支持使用函数处理数据，函数使用若干字段名或者常量值做为参数；参数的数量是不固定的，有的函数的参数为空，甚至有的函数的参数个数可变；几乎所有函数都有返回值，返回值即为函数的数据处理结果。

其实在前面的章节中我们已经用到函数了，最典型的就是“聚合函数”， “聚合函数”是函数的一种，它们可以对一组数据进行统计计算。除了“聚合函数”，SQL中还有其他类型的函数，比如进行数值处理的数学函数、进行日期处理的日期函数、进行字符串处理的字符串函数等。

我们来演示几个函数使用的典型场景。

主流数据库系统都提供了计算字符串长度的函数，在MYSQL、Oracle、DB2中这个函数名称为LENGTH，而在MSSQLServer中这个函数的名称则为LEN。这个函数接受一个字符串类型的字段值做为参数，返回值为这个字符串的长度。下面的SQL语句计算每一个名称不为空的员工的名字以及名字的长度：

MYSQL、Oracle、DB2：

```
SELECT FName, LENGTH(FName)  AS namelength FROM T_Employee
WHERE FName IS NOT  NULL
```

MSSQLServer：

```
SELECT FName, LEN(FName)  AS namelength FROM T_Employee
WHERE FName IS NOT  NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	namelength
Tom	3
Jerry	5
Jane	4
Tina	4
Smith	5
John	4
Kerry	5
Stone	5

主流系统都提供了取得字符串的子串的函数，在MYSQL、MSSQLServer中这个函数名称为SUBSTRING，而在Oracle、DB2这个函数名称为SUBSTR。这个函数接受三个参数，第一个参数为要取的主字符串，第二个参数为字串的起始位置（从1开始计数），第三个参数为字串的长度。下面的SQL语句取得每一个名称不为空的员工的名字以及名字中从第二个字符开始、长度为3的字串：

MYSQL、MSSQLServer：

```
SELECT FName, SUBSTRING(FName,2,3)  FROM T_Employee
WHERE FName IS NOT NULL
```

Oracle、DB2：

```
SELECT FName, SUBSTR(FName,2,3)  FROM T_Employee
WHERE FName IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	namelength
Tom	om
Jerry	er
Jane	an
Tina	in
Smith	mi
John	oh
Kerry	er
Stone	to

多个函数还可以嵌套使用。主流系统都提供了计算正弦函数值的函数SIN和计算绝对值的函数ABS，它们都接受一个数值类型的参数。下面的SQL语句取得每个员工的姓名、年龄、年龄的正弦函数值以及年龄的正弦函数值的绝对值，其中计算“年龄的正弦函数值的绝对值”时就要使用嵌套函数，SQL语句如下：

```
SELECT FName,FAge, SIN(FAge) , ABS(SIN(FAge)) FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FAge		
Tom	25	-0.13235175009777303	0.13235175009777303
Jerry	28	0.27090578830786904	0.27090578830786904
Jane	23	-0.8462204041751706	0.8462204041751706
Tina	25	-0.13235175009777303	0.13235175009777303
Smith	28	0.27090578830786904	0.27090578830786904
<NULL>	27	0.956375928404503	0.956375928404503
John	23	-0.8462204041751706	0.8462204041751706
Kerry	28	0.27090578830786904	0.27090578830786904
Stone	22	-0.008851309290403876	0.008851309290403876

数据库系统提供的函数是非常丰富的，而且不同的数据库系统提供的函数差异也非常大，本书后面章节将对这些函数进行详细讲解。

4.6.4 字符串的拼接

SQL允许两个或者多个字段之间进行计算，字符串类型的字段也不例外。比如我们需要以“工号+姓名”的方式在报表中显示一个员工的信息，那么就需要把工号和姓名两个字符串类型的字段拼接计算；再如我们需要在报表中在每个员工的工号前增加“Old”这个文本。这时候就需要我们对字符串类型的字段（包括字符串类型的常量字段）进行拼接。在不同的数据库系统下的字符串拼接是有很大差异的，因此这里我们将讲解主流数据库下的字符串拼接的差异。

需要注意的是，在Java、C#等编程语言中字符串是用半角的双引号来包围的，但是在有的数据库系统的SQL语法中双引号有其他的含义（比如列的别名），而所有的数据库系统都支持用单引号包围的形式定义的字符串，所以建议读者使用以单引号包围的形式定义的字符串，而且本书也将使用这种方式。

4.6.4.1 MYSQL

在Java、C#等编程语言中字符串的拼接可以通过加号“+”来实现，比如："1"+"3"、"a"+"b"。在MYSQL中也可以使用加号“+”来连接两个字符串，比如下面的SQL：

```
SELECT '12'+'33',FAge+'1' FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

'12'+'33'	FAge+'1'
45	26
45	29
45	24
45	26
45	29
45	28
45	24
45	29
45	23

仔细观察第一列，惊讶吗？这个列的显示结果并不是我们希望的“1233”，而是把“12”和“33”两个字符串当成数字来求两个数的和了；同样将一个数字与一个字符串用加号“+”连接也是同样的效果，比如这里的第二列。

在MYSQL中，当用加号“+”连接两个字段（或者多个字段）的时候，MYSQL会尝试将字段值转换为数字类型(如果转换失败则认为字段值为0)，然后进行字段的加法运算。因此，当计算的'12'+'33'的时候，MYSQL会将“12”和“33”两个字符串尝试转换为数字类型的12和33，然后计算12+33的值，这就是为什么我们会得到45的结果了。同样道理，在计算FAge+'1'的时候，由于FAge为数字类型，所以不需要进行转换，而'1'为字符串类型，所以MYSQL将'1'尝试转换为数字1，然后计算FAge+1做为计算列的值。

MYSQL会尝试将加号两端的字段值尝试转换为数字类型，如果转换失败则认为字段值为0，比如我们执行下面的SQL语句：

```
SELECT 'abc'+'123',FAge+'a' FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

'abc'+'123'	FAge+'a'
123	25
123	28
123	23
123	25
123	28
123	27
123	23
123	28
123	22

在MYSQL中进行字符串的拼接要使用CONCAT函数，CONCAT函数支持一个或者多个参数，参数类型可以为字符串类型也可以是非字符串类型，对于非字符串类型的参数MYSQL将尝试将其转化为字符串类型，CONCAT函数会将所有参数按照参数的顺序拼接成一个字符串做为返回值。比如下面的SQL语句用于将用户的多个字段信息以一个计算字段的形式查询出来：

```
SELECT CONCAT('工号为:',FNumber,'的员工的幸福指数:',FSalary/(FAge-21))
FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CONCAT('工号为:',FNumber,'的员工的幸福指数:',FSalary/(FAge-21))
工号为:DEV001 的员工的幸福指数:2075.000000
工号为:DEV002 的员工的幸福指数:328.685714
工号为:HR001 的员工的幸福指数:1100.440000

工号为:HR002 的员工的幸福指数:1300.090000
工号为:IT001 的员工的幸福指数:557.142857
工号为:IT002 的员工的幸福指数:466.666667
工号为:SALES001 的员工的幸福指数:2500.000000
工号为:SALES002 的员工的幸福指数:885.714286
工号为:SALES003 的员工的幸福指数:1200.000000

CONCAT支持只有一个参数的用法，这时的CONCAT可以看作是一个将这个参数值尝试转化为字符串类型值的函数。MYSQL中还提供了另外一个进行字符串拼接的函数CONCAT_WS，CONCAT_WS可以在待拼接的字符串之间加入指定的分隔符，它的第一个参数值为采用的分隔符，而剩下的参数则为待拼接的字符串值，比如执行下面的SQL：

```
SELECT CONCAT_WS(',',FNumber,FAge,FDepartment,FSalary) FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CONCAT_WS(',',FNumber,FAge,FDepartment,FSalary)
DEV001,25,Development,8300.00
DEV002,28,Development,2300.80
HR001,23,HumanResource,2200.88
HR002,25,HumanResource,5200.36
IT001,28,InfoTech,3900.00
IT002,27,InfoTech,2800.00
SALES001,23,Sales,5000.00
SALES002,28,Sales,6200.00
SALES003,22,Sales,1200.00

4.6.4.2 MSSQLServer

与MYSQL不同，MSSQLServer中可以直接使用加号“+”来拼接字符串。比如执行下面的SQL语句：

```
SELECT '工号为'+FNumber+'的员工姓名为'+Fname FROM T_Employee
WHERE FName IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

工号为 DEV001 的员工姓名为 Tom
工号为 DEV002 的员工姓名为 Jerry
工号为 HR001 的员工姓名为 Jane
工号为 HR002 的员工姓名为 Tina
工号为 IT001 的员工姓名为 Smith
工号为 SALES001 的员工姓名为 John
工号为 SALES002 的员工姓名为 Kerry
工号为 SALES003 的员工姓名为 Stone

4.6.4.3 Oracle

Oracle中使用“||”进行字符串拼接，其使用方式和MSSQLServer中的加号“+”一样。比如执行下面的SQL语句：

```
SELECT '工号为'||FNumber||'的员工姓名为'||FName FROM T_Employee
WHERE FName IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

工号为||FNUMBER||的员工姓名为||FNAME
工号为 DEV001 的员工姓名为 Tom
工号为 DEV002 的员工姓名为 Jerry
工号为 SALES001 的员工姓名为 John
工号为 SALES002 的员工姓名为 Kerry
工号为 SALES003 的员工姓名为 Stone
工号为 HR001 的员工姓名为 Jane
工号为 HR002 的员工姓名为 Tina
工号为 IT001 的员工姓名为 Smith

除了“||”，Oracle还支持使用CONCAT()函数进行字符串拼接，比如执行下面的SQL语句：

```
SELECT CONCAT('工号:',FNumber) FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CONCAT(工号:,FNUMBER)
工号:DEV001
工号:DEV002
工号:HR001
工号:HR002
工号:IT001
工号:IT002
工号:SALES001
工号:SALES002
工号:SALES003

如果CONCAT中连接的值不是字符串，Oracle会尝试将其转换为字符串，比如执行下面的SQL语句：

```
SELECT CONCAT('年龄:',FAge) FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CONCAT(年龄:,FAGE)
年龄:25
年龄:28
年龄:23
年龄:28
年龄:22
年龄:23
年龄:25
年龄:28
年龄:27

与MYSQL的CONCAT()函数不同，Oracle的CONCAT()函数只支持两个参数，不支持两个以上字符串的拼接，比如下面的SQL语句在Oracle中是错误的：

```
SELECT CONCAT('工号为',FNumber,'的员工姓名为',FName) FROM T_Employee
WHERE FName IS NOT NULL
```

运行以后Oracle会报出下面的错误信息：

```
参数个数无效
```

如果要进行多个字符串的拼接的话，可以使用多个CONCAT()函数嵌套使用，上面的SQL可以如

下改写：

```
SELECT CONCAT(CONCAT(CONCAT('工号为',FNumber),'的员工姓名为'),FName) FROM
T_Employee
WHERE FName IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CONCAT(CONCAT(CONCAT(工号为,FNUMBER),的员工姓名为),FNAME)
工号为 DEV001 的员工姓名为 Tom
工号为 DEV002 的员工姓名为 Jerry
工号为 SALES001 的员工姓名为 John
工号为 SALES002 的员工姓名为 Kerry
工号为 SALES003 的员工姓名为 Stone
工号为 HR001 的员工姓名为 Jane
工号为 HR002 的员工姓名为 Tina
工号为 IT001 的员工姓名为 Smith

4.6.4.4 DB2

DB2中使用“||”进行字符串拼接，其使用方式和MSSQLServer中的加号“+”一样。比如执行下面的SQL语句：

```
SELECT '工号为'||FNumber||'的员工姓名为'||FName FROM T_Employee
WHERE FName IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1
工号为 DEV001 的员工姓名为 Tom
工号为 DEV002 的员工姓名为 Jerry
工号为 SALES001 的员工姓名为 John
工号为 SALES002 的员工姓名为 Kerry
工号为 SALES003 的员工姓名为 Stone
工号为 HR001 的员工姓名为 Jane
工号为 HR002 的员工姓名为 Tina
工号为 IT001 的员工姓名为 Smith

除了“||”，DB2还支持使用CONCAT()函数进行字符串拼接，比如执行下面的SQL语句：

```
SELECT CONCAT('工号:',FNumber) FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1
工号:DEV001
工号:DEV002
工号:HR001
工号:HR002
工号:IT001
工号:IT002
工号:SALES001
工号:SALES002
工号:SALES003

与Oracle不同，如果CONCAT中连接的值不是字符串，则DB2不会尝试进行类型转换而是

报出错误信息，比如执行下面的SQL语句是错误的：

```
SELECT CONCAT('年龄:',FAge) FROM T_Employee
```

运行以后DB2会报出下面的错误信息：

```
未找到类型为 "FUNCTION" 命名为 "CONCAT" 且具有兼容自变量的已授权例程
```

与MYSQL的CONCAT()函数不同，DB2的CONCAT()函数只支持两个参数，不支持两个以上字符串的拼接，比如下面的SQL语句在Oracle中是错误的：

```
SELECT CONCAT('工号为',FNumber,'的员工姓名为',FName) FROM T_Employee
WHERE FName IS NOT NULL
```

运行以后Oracle会报出下面的错误信息：

```
未找到类型为 "FUNCTION" 命名为 "CONCAT" 且具有兼容自变量的已授权例程
```

如果要进行多个字符串的拼接的话，可以使用多个CONCAT()函数嵌套使用，上面的SQL可以如下改写：

```
SELECT CONCAT(CONCAT(CONCAT('工号为',FNumber),'的员工姓名为'),FName) FROM
T_Employee
WHERE FName IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1
工号为 DEV001 的员工姓名为 Tom
工号为 DEV002 的员工姓名为 Jerry
工号为 SALES001 的员工姓名为 John
工号为 SALES002 的员工姓名为 Kerry
工号为 SALES003 的员工姓名为 Stone
工号为 HR001 的员工姓名为 Jane
工号为 HR002 的员工姓名为 Tina
工号为 IT001 的员工姓名为 Smith

4.6.5 计算字段的其他用途

我们不仅能在SELECT语句中使用计算字段，我们同样可以在进行数据过滤、数据删除以及数据更新的时候使用计算字段，下面我们举几个例子。

4.6.5.1 计算处于合理工资范围内的员工

我们规定一个合理工资范围：上限为年龄的1.8倍加上5000元，下限为年龄的1.5倍加上2000元，介于这两者之间的即为合理工资。我们需要查询所有处于合理工资范围内的员工信息。因此编写如下的SQL语句：

```
SELECT * FROM T_Employee
WHERE Fsalary BETWEEN Fage*1.5+2000 AND Fage*1.8+5000
```

这里我们在**BETWEEN……AND……语句中使用了计算表达式。**执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment
DEV002	Jerry	28	2300.80	ShenZhen	Development
HR001	Jane	23	2200.88	Beijing	HumanResource
IT001	Smith	28	3900.00	Beijing	InfoTech
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech
SALES001	John	23	5000.00	Beijing	Sales

4.6.5.2 查询“工资年龄指数”

我们定义“工资年龄指数”为“工资除以年龄”。我们需要查询“工资年龄指数”的最

高值和最低值。因此编写如下的SQL语句：

```
SELECT MAX(FSalary/FAge) AS MAXVALUE,MIN(FSalary/FAge) AS MINVALUE
FROM T_Employee
```

这里我们在MAX、MIN函数中使用了计算字段。执行完毕我们就能在输出结果中看到下面的执行结果：

MAXVALUE	MINVALUE
332.0000000000000	54.5454545454545

4.6.5.3 年龄全部加1

新的一年到来了，系统需要自动将员工的年龄全部加1。这个工作如果使用代码来完成的话会是这样：

```
result = executeQuery("SELECT * FROM T_Employee");
for(i=0;i<result.count;i++)
{
    age = result[i].get("FAge");
    number = result[i].get("FNumber");
    age=age+1;
    executeUpdate("UPDATE T_Employee SET FAge="+age+" WHERE FNumber="+number);
}
```

这种方式在数据量比较大的时候速度是非常慢的，而在UPDATE中使用计算字段则可以非常简单快速的完成任务，编写下面的SQL语句：

```
UPDATE T_Employee SET FAge=FAge+1
```

这里在SET子句中采用计算字段的方式为FAge字段设定了新值。执行完毕后执行`SELECT * FROM T_Employee`来查看修改后的数据：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment
DEV001	Tom	26	8300.00	Beijing	Development
DEV002	Jerry	29	2300.80	ShenZhen	Development
HR001	Jane	24	2200.88	Beijing	HumanResource
HR002	Tina	26	5200.36	Beijing	HumanResource
IT001	Smith	29	3900.00	Beijing	InfoTech
IT002	<NULL>	28	2800.00	ShenZhen	InfoTech
SALES001	John	24	5000.00	Beijing	Sales
SALES002	Kerry	29	6200.00	Beijing	Sales
SALES003	Stone	23	1200.00	ShenZhen	Sales

4.7 不从实体表中取的数据

有的时候我们需要查询一些不能从任何实体表中能够取得的数据，比如将数字1作为结果集或者计算字符串“abc”的长度。

有的开发人员尝试使用下面的SQL来完成类似的功能：

```
SELECT 1 FROM T_Employee
```

可是执行以后却得到了下面的执行结果集

1
1
1

1
1
1
1
1
1

结果集中出现了不止一个1，这时因为通过这种方式得到的结果集数量是取决于T_Employee表中的数据条目数的，必须要借助于DISTINCT关键字来将结果集条数限定为一条，SQL语句如下：

```
SELECT DISTINCT 1 FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1

这种实现方式非常麻烦，而且如果数据库中一张表都没有的时候这样就不凑效了。主流数据库系统对这种需求提供了比较好的支持，这一节我们就来看一下主流数据库系统对此的支持。

MYSQL和MSSQLServer允许使用不带FROM子句的SELECT语句来查询这些不属于任何实体表的数据，比如下面的SQL将1作为结果集：

```
SELECT 1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1

还可以在不带FROM子句的SELECT语句中使用函数，比如下面的SQL将字符串“abc”的长度作为结果集：

MYSQL:

```
SELECT LENGTH('abc')
```

MSSQLServer:

```
SELECT LEN('abc')
```

执行完毕我们就能在输出结果中看到下面的执行结果：

3

还可以在SELECT语句中同时计算多个表达式，比如下面的SQL语句将1、2、3、’a’、’b’、’c’作为结果集：

```
SELECT 1,2,3,'a','b','c'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1	2	3	a	b	c

在Oracle中是不允许使用这种不带FROM子句的SELECT语句，不过我们可以使用Oracle的系统表来作为FROM子句中的表名，系统表是Oracle内置的特殊表，最常用的系统表为DUAL。比如下面的SQL将1以及字符串'abc'的长度作为结果集：

```
SELECT 1, LENGTH('abc')    FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1	LENGTH(ABC)
1	3

在DB2中也同样不支持不带FROM子句的SELECT语句，它也是采用和Oracle类似的系统表，最常用的系统表为SYSIBM.SYSDUMMY1。比如下面的SQL将1以及字符串'abc'的长度作为结果集：

```
SELECT 1, LENGTH('abc')  FROM SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1	2
1	3

4.8 联合结果集

有的时候我们需要组合两个完全不同的查询结果集，而这两个查询结果之间没有必然的联系，只是我们需要将他们显示在一个结果集中而已。在SQL中可以使用UNION运算符来将两个或者多个查询结果集联合为一个结果集中。

为了更好的讲解本节的内容，需要首先创建一张用来存储临时工信息的新表，在数据库系统下执行下面的SQL语句：

MYSQL:

```
CREATE TABLE T_TempEmployee (FIdCardNumber VARCHAR(20),FName VARCHAR(20),FAge INT ,PRIMARY KEY (FIdCardNumber))
```

MSSQLServer:

```
CREATE TABLE T_TempEmployee (FIdCardNumber VARCHAR(20),FName VARCHAR(20),FAge INT, PRIMARY KEY (FIdCardNumber))
```

Oracle:

```
CREATE TABLE T_TempEmployee (FIdCardNumber VARCHAR2(20),FName VARCHAR2(20),FAge NUMBER (10), PRIMARY KEY (FIdCardNumber))
```

DB2：

```
CREATE TABLE T_TempEmployee (FIdCardNumber VARCHAR(20) Not NULL,FName VARCHAR(20),FAge INT, PRIMARY KEY (FIdCardNumber))
```

由于临时工没有分配工号，所以使用身份证号码FIdCardNumber来标识一个临时工，同时由于临时工不是实行月薪制，所以这里也没有记录月薪信息。我们还需要一些初始数据，执行下面的SQL语句以插入初始数据：

```
INSERT INTO T_TempEmployee(FIdCardNumber,FName,FAge) VALUES('1234567890121','Sarani',33);
INSERT INTO T_TempEmployee(FIdCardNumber,FName,FAge) VALUES('1234567890122','Tom',26);
INSERT INTO T_TempEmployee(FIdCardNumber,FName,FAge) VALUES('1234567890123','Yalaha',38);
INSERT INTO T_TempEmployee(FIdCardNumber,FName,FAge) VALUES('1234567890124','Tina',26);
INSERT INTO T_TempEmployee(FIdCardNumber,FName,FAge) VALUES('1234567890125','Konkaya',29);
INSERT INTO T_TempEmployee(FIdCardNumber,FName,FAge) VALUES('1234567890126','Fotfa',46) ;
```

插入初始数据完毕以后执行SELECT * FROM T_TempEmployee查看表中的数据：

FIdCardNumber	FName	FAge
1234567890121	Sarani	33
1234567890122	Tom	26

1234567890123	Yalaha	38
1234567890124	Tina	26
1234567890125	Konkaya	29
1234567890126	Fotfa 4	6

4.8.1 简单的结果集联合

UNION运算符要放置在两个查询语句之间。比如我们要查询公司所有员工（包括临时工）的标识号码、姓名、年龄信息。

查询正式员工信息的SQL语句如下：

```
SELECT FNumber,FName,FAge FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge
DEV001	Tom	26
DEV002	Jerry	29
HR001	Jane	24
HR002	Tina	26
IT001	Smith	29
IT002	<NULL>	28
SALES001	John	24
SALES002	Kerry	29
SALES003	Stone	23

而查询临时工信息的SQL语句如下：

```
SELECT FIdCardNumber,FName,FAge FROM T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FIdCardNumber	FName	FAge
1234567890121	Sarani	33
1234567890122	Tom	26
1234567890123	Yalaha	38
1234567890124	Tina	26
1234567890125	Konkaya	29
1234567890126	Fotfa 4	6

只要用UNION操作符连接这两个查询语句就可以将两个查询结果集联合为一个结果集，SQL语句如下：

```
SELECT FNumber,FName,FAge FROM T_Employee
UNION
SELECT FIdCardNumber,FName,FAge FROM T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge
1234567890121	Sarani	33
1234567890122	Tom	26
1234567890123	Yalaha	38
1234567890124	Tina	26
1234567890125	Konkaya	29
1234567890126	Fotfa 4	6

DEV001	Tom	26
DEV002	Jerry	29
HR001	Jane	24
HR002	Tina	26
IT001	Smith	29
IT002	<NULL>	28
SALES001	John	24
SALES002	Kerry	29
SALES003	Stone	23

可以看到UNION操作符将两个独立的结果集联合成为了一个结果集。

UNION可以连接多个结果集，就像“+”可以连接多个数字一样简单，只要在每个结果集之间加入UNION即可，比如下面的SQL语句就连接了三个结果集：

```
SELECT FNumber,FName,FAge FROM T_Employee
WHERE FAge<30
UNION
SELECT FIdCardNumber,FName,FAge FROM T_TempEmployee
WHERE FAge>40
UNION
SELECT FIdCardNumber,FName,FAge FROM T_TempEmployee
WHERE FAge<30
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge
1234567890122	Tom	26
1234567890124	Tina	26
1234567890125	Konkaya	29
1234567890126	Fotfa 4	6
DEV001	Tom	26
DEV002	Jerry	29
HR001	Jane	24
HR002	Tina	26
IT001	Smith	29
IT002	<NULL>	28
SALES001	John	24
SALES002	Kerry	29
SALES003	Stone	23

4.8.2 联合结果集的原则

联合结果集不必受被联合的多个结果集之间的关系限制，不过使用UNION仍然有两个基本的原则需要遵守：一是每个结果集必须有相同的列数；二是每个结果集的列必须类型相容。

首先看第一个原则，每个结果集必须有相同的列数，两个不同列数的结果集是不能联合在一起的。比如下面的SQL语句是错误的：

```
SELECT FNumber,FName,FAge,FDepartment FROM T_Employee
UNION
SELECT FIdCardNumber,FName,FAge FROM T_TempEmployee
```

执行以后数据库系统会报出如下的错误信息：

使用 UNION、INTERSECT 或 EXCEPT 运算符合并的所有查询必须在其目标列表中有相同数目的表达式。

因为第一个结果集返回了4列数据，而第二个结果集则返回了3列数据，数据库系统并不会用空值将第二个结果集补足为4列。如果需要将未知列补足为一个默认值，那么可以使用常量字段，比如下面的SQL语句就将第二个结果集的与FDepartment对应的字段值设定为“临时工，不属于任何一个部门”：

```
SELECT FNumber,FName,FAge,FDepartment FROM T_Employee
UNION
SELECT FIdCardNumber,FName,FAge,'临时工，不属于任何一个部门' FROM
T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FDepartment
1234567890121	Sarani	33	临时工，不属于任何一个部门
1234567890122	Tom	26	临时工，不属于任何一个部门
1234567890123	Yalaha	38	临时工，不属于任何一个部门
1234567890124	Tina	26	临时工，不属于任何一个部门
1234567890125	Konkaya	29	临时工，不属于任何一个部门
1234567890126	Fotfa 4	6	临时工，不属于任何一个部门
DEV001	Tom	26	Development
DEV002	Jerry	29	Development
HR001	Jane	24	HumanResource
HR002	Tina	26	HumanResource
IT001	Smith	29	InfoTech
IT002	<NULL>	28	InfoTech
SALES001	John	24	Sales
SALES002	Kerry	29	Sales

联合结果集的第二个原则是：每个结果集的列必须类型相容，也就是说结果集的每个对应列的数据类型必须相同或者能够转换为同一种数据类型。比如下面的SQL语句在MYSQL中可以正确的执行：

```
SELECT FIdCardNumber,FAge,FName FROM T_TempEmployee
UNION
SELECT FNumber,FName,FAge FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FIdCardNumber	FAge	FName
1234567890121	33	Sarani
1234567890122	26	Tom
1234567890123	38	Yalaha
1234567890124	26	Tina
1234567890125	29	Konkaya
1234567890126	46	Fotfa
DEV001	Tom	26
DEV002	Jerry	29
HR001	Jane	24
HR002	Tina	26

IT001	Smith	29
IT002	<NULL>	28
SALES001	John	24
SALES002	Kerry	29
SALES003	Stone	23

可以看到MYSQL将FAge转换为了文本类型，以便于与FName字段值匹配。不过这句SQL语句在MSSQLServer、Oracle、DB2中执行则会报出类似如下的错误信息：

表达式必须具有与对应表达式相同的数据类型。

因为这些数据库系统不会像MYSQL那样进行默认的数据类型转换。由于不同数据库系统中数据类型转换规则是各不相同的，因此如果开发的应用程序要考虑跨多数据库移植的话最好保证结果集的每个对应列的数据类型完全相同。

4.8.3 UNION ALL

我们想列出公司中所有员工（包括临时工）的姓名和年龄信息，那么我们可以执行下面的SQL语句：

```
SELECT FName,FAge FROM T_Employee
UNION
SELECT FName,FAge FROM T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FAge
<NULL>	28
Fotfa 4	6
Jane	24
Jerry	29
John	24
Kerry	29
Konkaya	29
Sarani	33
Smith	29
Stone	23
Tina	26
Tom	26
Yalaha	38

仔细观察结果集，我们发现输出的结果和预想的是不一致的，在正式员工中有姓名为Tom、年龄为26以及姓名为Tina、年龄为26的两名员工，而临时工中也有姓名为Tom、年龄为26以及姓名为Tina、年龄为26的两名员工，也就是说正式员工的临时工中存在重名和年龄重复的现象，但是在查询结果中却将重复的信息只保留了一条，也就是只有一个姓名为Tom、年龄为26的员工和一个姓名为Tina、年龄为26的员工。

这时因为默认情况下，UNION运算符合并了两个查询结果集，其中完全重复的数据行被合并为了一条。如果需要在联合结果集中返回所有的记录而不管它们是否唯一，则需要在UNION运算符后使用ALL操作符，比如下面的SQL语句：

```
SELECT FName,FAge FROM T_Employee
UNION ALL
SELECT FName,FAge FROM T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FAge
Tom	26
Jerry	29
Jane	24
Tina	26
Smith	29
<NULL>	28
John	24
Kerry	29
Stone	23
Sarani	33
Tom	26
Yalaha	38
Tina	26
Konkaya	29
Fotfa 4	6

4.8.4 联合结果集应用举例

联合结果集在制作报表的时候经常被用到，使用联合结果集我们可以将没有直接关系的数据显示到同一张报表中。使用UNION运算符，只要被联合的结果集符合联合结果集的原则，那么被连接的两个SQL语句可以是非常复杂，也可以是非常简单的。本小节将展示几个实用的例子来看一下联合结果集在实际开发中的应用。

4.8.4.1 员工年龄报表

要求查询员工的最低年龄和最高年龄，临时工和正式员工要分别查询。

实现SQL语句如下：

```
SELECT '正式员工最高年龄',MAX(FAge) FROM T_Employee
UNION
SELECT '正式员工最低年龄',MIN(FAge) FROM T_Employee
UNION
SELECT '临时工最高年龄',MAX(FAge) FROM T_TempEmployee
UNION
SELECT '临时工最低年龄',MIN(FAge) FROM T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

临时工最低年龄	26
临时工最高年龄	46
正式员工最低年龄	23
正式员工最高年龄	29

4.8.4.2 正式员工工资报表

要求查询每位正式员工的信息，包括工号、工资，并且在最后一行加上所有员工工资额合计。

实现SQL语句如下：

```
SELECT FNumber,FSalary FROM T_Employee
UNION
SELECT '工资合计',SUM(FSalary) FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FSalary
DEV001	8300.00
DEV002	2300.80
HR001	2200.88
HR002	5200.36
IT001	3900.00
IT002	2800.00
SALES001	5000.00
SALES002	6200.00
SALES003	1200.00
工资合计	37102.04

4.8.4.3 打印5以内自然数的平方

要求打印出打印5以内自然数以及它们的平方数。

实现SQL语句如下：

MYSQL、MSSQLServer：

```
SELECT 1,1 * 1
UNION
SELECT 2,2 * 2
UNION
SELECT 3,3 * 3
UNION
SELECT 4,4 * 4
UNION
SELECT 5,5 * 5
```

Oracle：

```
SELECT 1,1 * 1 FROM DUAL
UNION
SELECT 2,2 * 2 FROM DUAL
UNION
SELECT 3,3 * 3 FROM DUAL
UNION
SELECT 4,4 * 4 FROM DUAL
UNION
SELECT 5,5 * 5 FROM DUAL
```

DB2:

```
SELECT 1,1 * 1 FROM SYSIBM.SYSDUMMY1
UNION
SELECT 2,2 * 2 FROM SYSIBM.SYSDUMMY1
UNION
SELECT 3,3 * 3 FROM SYSIBM.SYSDUMMY1
UNION
```

```
SELECT 4,4 * 4 FROM SYSIBM.SYSDUMMY1
UNION
SELECT 5,5 * 5 FROM SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1	1
2	4
3	9
4	16
5	25

4.8.4.4 列出员工姓名

要求列出公司中所有员工（包括临时工）的姓名，将重复的姓名过滤掉。

实现SQL语句如下：

```
SELECT FName FROM T_Employee
UNION
SELECT FName FROM T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName
<NULL>
Fotfa
Jane
Jerry
John
Kerry
Konkaya
Sarani
Smith
Stone
Tina
Tom
Yalaha

4.8.4.5 分别列出正式员工和临时工的姓名

要求分别列出正式员工和临时工的姓名，要保留重复的姓名。

实现SQL语句如下：

MYSQL、MSSQLServer：

```
SELECT '以下是正式员工的姓名'
UNION ALL
SELECT FName FROM T_Employee
UNION ALL
SELECT '以下是临时工的姓名'
UNION ALL
SELECT FName FROM T_TempEmployee
```

Oracle：

```
SELECT '以下是正式员工的姓名' FROM DUAL
UNION ALL
SELECT FName FROM T_Employee
UNION ALL
SELECT '以下是临时工的姓名' FROM DUAL
UNION ALL
SELECT FName FROM T_TempEmployee
```

DB2：

```
SELECT '以下是正式员工的姓名' FROM SYSIBM.SYSDUMMY1
UNION ALL
SELECT FName FROM T_Employee
UNION ALL
SELECT '以下是临时工的姓名' FROM SYSIBM.SYSDUMMY1
UNION ALL
SELECT FName FROM T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

以下是正式员工的姓名
Tom
Jerry
Jane
Tina
Smith
<NULL>
John
Kerry
Stone
以下是临时工的姓名
Sarani
Tom
Yalaha
Tina
Konkaya

本章即将结束，请执行下面的SQL语句将本章用到的数据表删除：

```
Drop TABLE T_Employee;
Drop TABLE T_TempEmployee;
```

第五章 函数

第四章中我们讲解了 SQL 中函数的用法。SQL 中可供使用的函数是非常多的，这些函数的功能包括转换字符串大小写、求一个数的对数、计算两个日期之间的天数间隔等，数量的掌握这些函数将能够帮助我们更快的完成业务功能。本章将讲解这些函数的使用，并且对它们在不同数据库系统中的差异性进行比较。

为了更容易的运行本章中的例子，必须首先创建所需要的数据表，因此下面列出本章中要用到数据表的创建 SQL 语句：

MYSQL:

```
CREATE TABLE T_Person (FIdNumber VARCHAR(20),
FName VARCHAR(20),FBirthDay DATETIME,
FRegDay DATETIME,FWeight DECIMAL(10,2))
```

MSSQLServer:

```
CREATE TABLE T_Person (FIdNumber VARCHAR(20),
FName VARCHAR(20),FBirthDay DATETIME,
FRegDay DATETIME,FWeight NUMERIC(10,2))
```

Oracle:

```
CREATE TABLE T_Person (FIdNumber VARCHAR2(20),
FName VARCHAR2(20),FBirthDay DATE,
FRegDay DATE,FWeight NUMERIC(10,2))
```

DB2:

```
CREATE TABLE T_Person (FIdNumber VARCHAR(20),
FName VARCHAR(20),FBirthDay DATE,
FRegDay DATE,FWeight DECIMAL(10,2))
```

请在不同的数据库系统中运行相应的 SQL 语句。T_Person 为记录人员信息的数据表，其中字段 FIdNumber 为人员的身份证号码，FName 为人员姓名，FBirthDay 为出生日期，FRegDay 为注册日期，FWeight 为体重。

为了更加直观的验证本章中函数使用方法的正确性，我们需要在 T_Person 表中预置一些初始数据，请在数据库中执行下面的数据插入 SQL 语句：

MYSQL、MSSQLServer、DB2：

```
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789120','Tom','1981-03-22','1998-05-01',56.67);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789121','Jim','1987-01-18','1999-08-21',36.17);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789122','Lily','1987-11-08','2001-09-18',40.33);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789123','Kelly','1982-07-12','2000-03-01',46.23);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789124','Sam','1983-02-16','1998-05-01',48.68);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789125','Kerry','1984-08-07','1999-03-01',66.67);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789126','Smith','1980-01-09','2002-09-23',51.28);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789127','BillGates','1972-07-18','1995-06-19',60.32);
```

Oracle：

```
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789120','Tom',TO_DATE('1981-03-22', 'YYYY-MM-DD HH24:MI:SS'),
```

```
TO_DATE('1998-05-01', 'YYYY-MM-DD HH24:MI:SS'),56.67);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789121','Jim',TO_DATE('1987-01-18', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('1999-08-21', 'YYYY-MM-DD HH24:MI:SS'),36.17);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789122','Lily',TO_DATE('1987-11-08', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('2001-09-18', 'YYYY-MM-DD HH24:MI:SS'),40.33);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789123','Kelly',TO_DATE('1982-07-12', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('2000-03-01', 'YYYY-MM-DD HH24:MI:SS'),46.23);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789124','Sam',TO_DATE('1983-02-16', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('1998-05-01', 'YYYY-MM-DD HH24:MI:SS'),48.68);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789125','Kerry',TO_DATE('1984-08-07', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('1999-03-01', 'YYYY-MM-DD HH24:MI:SS'),66.67);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789126','Smith',TO_DATE('1980-01-09', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('2002-09-23', 'YYYY-MM-DD HH24:MI:SS'),51.28);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789127','BillGates',TO_DATE('1972-07-18', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('1995-06-19', 'YYYY-MM-DD HH24:MI:SS'),60.32);
```

初始数据预置完毕以后执行 `SELECT * FROM T_Person` 来查看表中的数据，内容如下：

FIdNumber	FName	FBirthDay	FRegDay	FWeight
123456789120	Tom	1981-03-22 00:00:00.0	1998-05-01 00:00:00.0	56.67
123456789121	Jim	1987-01-18 00:00:00.0	1999-08-21 00:00:00.0	36.17
123456789122	Lily	1987-11-08 00:00:00.0	2001-09-18 00:00:00.0	40.33
123456789123	Kelly	1982-07-12 00:00:00.0	2000-03-01 00:00:00.0	46.23
123456789124	Sam	1983-02-16 00:00:00.0	1998-05-01 00:00:00.0	48.68
123456789125	Kerry	1984-08-07 00:00:00.0	1999-03-01 00:00:00.0	66.67
123456789126	Smith	1980-01-09 00:00:00.0	2002-09-23 00:00:00.0	51.28
123456789127	BillGates	1972-07-18 00:00:00.0	1995-06-19 00:00:00.0	60.32

5.1 数学函数

本节内容试读版不提供。请购买《程序员的 SQL 金典》。

5.2 字符串函数

本节内容试读版不提供。请购买《程序员的 SQL 金典》。

5.3 日期时间函数

日期时间类型的数据也是经常用到的，比如员工出生日期、结账日期、入库日期等等，而且经常需要对这些数据进行处理，比如检索所有超过保质期的商品、将结账日期向后延迟 3 天、检索所有每个月 18 日的入库记录，进行这些处理就需要使用日期时间函数。SQL 中提供了丰富的日期时间函数用于完成这些功能，本节将对这些日期时间函数进行详细讲解。

5.3.1 日期、时间、日期时间与时间戳

根据表示的类型、精度的不同，数据库中的日期时间数据类型分为日期、时间、日期时间以及时间戳四种类型。

日期类型是用来表示“年-月-日”信息的数据类型，其精度精确到“日”，其中包含了年、月、日三个信息，比如“2008-08-08”。日期类型可以用来表示“北京奥运会开幕式日期”、“王小明的出生年月日”等信息，但是无法表示“最近一次迟到的时间”、“徐总抵京时间”等精确到小时甚至分秒的数据。在数据库中，一般用 Date 来表示日期类型。

时间类型是用来表示“小时:分:秒” 信息的数据类型，其精度精确到“秒”，其中包含了小时、分、秒三个信息，比如“19:00:00”。时间类型可以用来表示“每天《新闻联播》的播出时间”、“每天的下班时间”等信息，但是无法表示“卢沟桥事变爆发日期”、“上次结账时间”等包含“年-月-日”等信息的数据。在数据库中，一般用 Time 来表示时间类型。

日期时间类型是用来表示“年-月-日 小时:分:秒”信息的数据类型，其精度精确到“秒”，其中包含了年、月、日、小时、分、秒六个信息，比如“2008-08-08 08:00:00”。日期时间类型可以用来表示“北京奥运会开幕式准确时间”、“上次迟到时间”等信息。在数据库中，一般用 DateTime 来表示日期时间类型。

日期时间类型的精度精确到“秒”，这在一些情况下能够满足基本的要求，但是对于精度要求更加高的日期时间信息则无法表示，比如“刘翔跑到终点的时间”、“货物 A 经过射频识别器的时间”等更高精度要求的信息。数据库中提供了时间戳类型用于表示这些对精度要求更加高的场合。时间戳类型还可以用于标记表中数据的版本信息，比如我们想区分表中两条记录插入表中的先后顺序，由于数据库操作速度非常快，如果用 DateTime 类型记录输入插入时间的话，若两条记录插入的时间间隔非常短的话是无法区分它们的，这时就可以使用时间戳类型。在有的数据库系统中，如果对数据表中的记录进行了更新的话，数据库系统会自动更新其中的时间戳字段的值。数据库中，一般用 TimeStamp 来表示日期时间类型。

不同的数据库系统对日期、时间、日期时间与时间戳等数据类型的支持差异性非常大，有的数据类型在有的数据库系统中不被支持，而有的数据类型的表示精度则和其类型名称所暗示的精度不同，比如 MSSQLServer 中不支持 Time 类型、Oracle 中的 Date 类型中包含时间信息。数据库中的日期时间函数对这些类型的支持差别是非常小的，因此在一般情况下我们将这些类型统一称为“日期时间类型”。

5.3.2 主流数据库系统中日期时间类型的表示方式

在 MYSQL、MSSQLServer 和 DB2 中可以用字符串来表示日期时间类型，数据库系统会自动在内部将它们转换为日期时间类型，比如“'2008-08-08'”、“2008-08-08 08:00:00”、“08:00:00” 、“2008-08-08 08:00:00.000000”等。

在 Oracle 中以字符串表示的数据是不能自动转换为日期时间类型的，必须使用 TO_DATE()函数来手动将字符串转换为日期时间类型的，比如 TO_DATE('2008-08-08', 'YYYY-MM-DD HH24:MI:SS') 、 TO_DATE('2008-08-08 08:00:00', 'YYYY-MM-DD HH24:MI:SS')、TO_DATE('08:00:00', 'YYYY-MM-DD HH24:MI:SS')等。

5.3.3 取得当前日期时间

在系统中经常需要使用当前日期时间进行处理，比如将“入库时间”字段设定为当前日期时间，在 SQL 中提供了取得当前日期时间的方式，不过各个数据库中的实现方式各不相

同。

5.3.3.1 MYSQL

MYSQL 中提供了 NOW()函数用于取得当前的日期时间，NOW()函数还有 SYSDATE()、CURRENT_TIMESTAMP 等别名。如下：

```
SELECT NOW(),SYSDATE(),CURRENT_TIMESTAMP
```

执行完毕我们就能在输出结果中看到下面的执行结果：

NOW()	SYSDATE()	CURRENT_TIMESTAMP
2008-01-12 01:13:19	2008-01-12 01:13:19	2008-01-12 01:13:19

如果想得到不包括时间部分的当前日期，则可以使用 **CURDATE()**函数，**CURDATE()**函数还有 **CURRENT_DATE** 等别名。如下：

```
SELECT CURDATE(),CURRENT_DATE
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CURDATE()	CURRENT_DATE
2008-01-12	2008-01-12

如果想得到不包括日期部分的当前时间，则可以使用 **CURTIME()**函数，**CURTIME ()**函数还有 **CURRENT_TIME** 等别名。如下：

```
SELECT CURTIME(),CURRENT_TIME
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CURTIME()	CURRENT_TIME
01:17:09	01:17:09

5.3.3.2 MSQLServer

MSSQLServer 中用于取得当前日期时间的函数为 GETDATE()。如下：

```
SELECT GETDATE() as 当前日期时间
```

执行完毕我们就能在输出结果中看到下面的执行结果：

当前日期时间
2008-01-12 01:02:04.78

可以看到 GETDATE()返回的信息是包括了日期、时间（精确到秒以后部分）的时间戳信息。MSSQLServer 没有专门提供取得当前日期、取得当前时间的函数，不过我们可以将 GETDATE()的返回值进行处理，这里需要借助于 Convert()函数，这个函数的详细介绍后面章节介绍，这里只介绍它在日期处理方面的应用。

使用 **CONVERT(VARCHAR(50)** ,日期时间值, **101)**可以得到日期时间值的日期部分，因此下面的 SQL 语句可以得到当前的日期值：

```
SELECT CONVERT(VARCHAR(50) ,GETDATE( ), 101) as 当前日期
```

执行完毕我们就能在输出结果中看到下面的执行结果：

当前日期
01/14/2008

使用 **CONVERT(VARCHAR(50)** ,日期时间值, **108)**可以得到日期时间值的日期部分，因此下面的 SQL 语句可以得到当前的日期值：

```
SELECT CONVERT(VARCHAR(50) ,GETDATE(), 108) as 当前时间
```

执行完毕我们就能在输出结果中看到下面的执行结果：

当前时间
21:37:19

5.3.3.3 Oracle

Oracle 中没有提供取得当前日期时间的函数，不过我们可以到系统表 DUAL 中查询

SYSTIMESTAMP 的值来得到当前的时间戳。如下：

```
SELECT SYSTIMESTAMP
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

SYSTIMESTAMP
2008-1-14 21.46.42.78000000 8:0

同样，我们可以到系统表 DUAL 中查询 SYSDATE 的值来得到当前日期时间。如下：

```
SELECT SYSDATE
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

SYSDATE
2008-01-14 21:47:16.0

同样，Oracle 中也没有专门提供取得当前日期、取得当前时间的函数，不过我们可以将 SYSDATE 的值进行处理，这里需要借助于 TO_CHAR()函数，这个函数的详细介绍后面章节介绍，这里只介绍它在日期处理方面的应用。

使用 TO_CHAR(时间日期值, 'YYYY-MM-DD') 可以得到日期时间值的日期部分，因此下面的 SQL 语句可以得到当前的日期值：

```
SELECT TO_CHAR(SYSDATE, 'YYYY-MM-DD')
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

TO_CHAR(SYSDATE,YYYY-MM-DD)
2008-01-14

使用 TO_CHAR(时间日期值, 'HH24:MI:SS') 可以得到日期时间值的时间部分，因此下面的 SQL 语句可以得到当前的日期值：

```
SELECT TO_CHAR(SYSDATE, 'HH24:MI:SS')
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

TO_CHAR(SYSDATE,HH24:MI:SS)
21:56:13

5.3.3.4 DB2

DB2 中同样没有提供取得当前日期时间的函数，不过我们可以到系统表 SYSIBM.SYSDUMMY1 中查询 CURRENT TIMESTAMP 的值来得到当前时间戳。如下：

```
SELECT CURRENT TIMESTAMP
FROM SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1
2008-01-14-21.58.20.01515000

从系统表 SYSIBM.SYSDUMMY1 中查询 CURRENT DATE 的值来得到当前日期值。如下：

```
SELECT CURRENT DATE
FROM SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1
2008-01-14

从系统表 SYSIBM.SYSDUMMY1 中查询 CURRENT **TIME** 的值来得到当前日期值。如下：

```
SELECT CURRENT TIME
FROM SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1
22:05:48

5.3.4 日期增减

有时我们需要在一个日期的基础上增加某个时间长度或者减去某个时间长度，比如我们知道每个员工的出生日期，而想计算出他出生后 10000 天的日期，再如我们想计算所有合同的到期日的三月后的日期。由于存在每个月天数不同、闰月等复杂的历法规则，所以不能使用简单的数字加减法进行计算，主流的数据库系统中都提供了对日期增减的计算，下面分别进行介绍。

5.3.4.1 MYSQL

MYSQL中提供了DATE_ADD()函数用于进行日期时间的加法运算，这个函数还有一个别名为ADDDATE()， DATE_ADD()函数的参数格式如下：

DATE_ADD (*date*,INTERVAL *expr type*)

其中参数date为待计算的日期；参数expr为待进行加法运算的增量，它可以是数值类型或者字符串类型，取决于type参数的取值；参数type则为进行加法运算的单位，type参数可选值以及对应的expr参数的格式如下表：

type参数值	expr参数的格式	说明
MICROSECOND	数值类型	以微秒为计算单位
SECOND	数值类型	以秒为计算单位
MINUTE	数值类型	以分钟为计算单位
HOUR	数值类型	以小时为计算单位
DAY	数值类型	以天为计算单位
WEEK	数值类型	以周为计算单位
MONTH	数值类型	以月为计算单位
QUARTER	数值类型	以季度为计算单位
YEAR	数值类型	以年为计算单位
SECOND_MICROSECOND	字符串类型，格式为：**'SECONDS.MICROSECONDS'**	以秒、微秒为计算单位，要求expr参数必须是“秒.微秒”的格式，比如“30.10”表示增加30秒10微秒。
MINUTE_MICROSECOND	字符串类型，格式为：**'MINUTES.MICROSECONDS'**	以分钟、毫秒为计算单位，要求expr参数必须是“分钟.微秒”的格式，比如“30.10”表示增加30分钟10微秒。
MINUTE_SECOND	字符串类型，格式为：**'MINUTES:SECONDS'**	以分钟、秒为计算单位，要求expr参数必须是“分钟:秒”的格式，比如“30:10”表示增加30分钟10秒。
HOUR_MICROSECOND	字符串类型，格式为：**'HOURS.MICROSECONDS'**	以小时、微秒为计算单位，要求expr参数必须是“小时.微秒”的格式，比如“30.10”表示增加30小时10微秒。

HOUR_SECOND	字符串类型，格式为：'HOURS:MINUTES:SECONDS'	以小时、分钟、秒为计算单位，要求expr参数必须是“小时:分钟:秒”的格式，比如“1:30:10”表示增加1小时30分钟10秒。
HOUR_MINUTE	字符串类型，格式为：'HOURS:MINUTES'	以小时、秒为计算单位，要求expr参数必须是“小时:秒”的格式，比如“30:10”表示增加30小时10秒。
DAY_MICROSECOND	字符串类型，格式为：'DAYS.MICROSECONDS'	以天、微秒为计算单位，要求expr参数必须是“天.微秒”的格式，比如“30.10”表示增加30天10微秒。
DAY_SECOND	字符串类型，格式为：'DAYS HOURS:MINUTES:SECONDS'	以天、小时、分钟、秒为计算单位，要求expr参数必须是“天 小时:分钟:秒”的格式，比如“1 3:28:36”表示增加1天3小时28分钟36秒。
DAY_MINUTE	字符串类型，格式为：'DAYS HOURS:MINUTES'	以天、小时、分钟为计算单位，要求expr参数必须是“天 小时:分钟”的格式，比如“1 3:15”表示增加1天3小时15分钟。
DAY_HOUR	字符串类型，格式为：'DAYS HOURS'	以天、小时为计算单位，要求expr参数必须是“天 小时”的格式，比如“30 10”表示增加30天10小时。
YEAR_MONTH	字符串类型，格式为：'YEARS-MONTHS'	以年、月为计算单位，要求expr参数必须是“年-月”的格式，比如“2-8”表示增加2年8个月。

表中前九种用法都非常简单，比如DATE_ADD(date,INTERVAL 1 HOUR)就可以得到在日期date基础上增加一小时后的日期时间，而DATE_ADD(date,INTERVAL 1 WEEK)就可以得到在日期date基础上增加一周后的日期时间。下面的SQL语句用来计算每个人出生一周、两个月以及5个季度后的日期：

```
SELECT FBirthDay,
DATE_ADD(FBirthDay,INTERVAL 1 WEEK) as w1,
DATE_ADD(FBirthDay,INTERVAL 2 MONTH) as m2,
DATE_ADD(FBirthDay,INTERVAL 5 QUARTER) as q5
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	w1	m2	q5
1981-03-22 00:00:00	1981-03-29 00:00:00	1981-05-22 00:00:00	1982-06-22 00:00:00
1987-01-18 00:00:00	1987-01-25 00:00:00	1987-03-18 00:00:00	1988-04-18 00:00:00
1987-11-08 00:00:00	1987-11-15 00:00:00	1988-01-08 00:00:00	1989-02-08 00:00:00
1982-07-12 00:00:00	1982-07-19 00:00:00	1982-09-12 00:00:00	1983-10-12 00:00:00
1983-02-16 00:00:00	1983-02-23 00:00:00	1983-04-16 00:00:00	1984-05-16 00:00:00
1984-08-07 00:00:00	1984-08-14 00:00:00	1984-10-07 00:00:00	1985-11-07 00:00:00
1980-01-09 00:00:00	1980-01-16 00:00:00	1980-03-09 00:00:00	1981-04-09 00:00:00
1972-07-18 00:00:00	1972-07-25 00:00:00	1972-09-18 00:00:00	1973-10-18 00:00:00

相对于前九种用法来说，后面几种用法就相对复杂一些，需要根据格式化的类型来决定

expr参数的值。比如如果想为日期增加3天2小时10分钟，那么就可以如下使用DATE_ADD()函数：

```
DATE_ADD(date,INTERVAL '3 2:10' DAY_MINUTE)
```

比如下面的SQL语句分别计算出生日期后3天2小时10分钟、1年6个月的日期时间：

```
SELECT FBirthDay,
DATE_ADD(FBirthDay,INTERVAL '3 2:10' DAY_MINUTE) as dm,
DATE_ADD(FBirthDay,INTERVAL '1-6' YEAR_MONTH) as ym
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	dm	ym
1981-03-22 00:00:00	1981-03-25 02:10:00	1982-09-22 00:00:00
1987-01-18 00:00:00	1987-01-21 02:10:00	1988-07-18 00:00:00
1987-11-08 00:00:00	1987-11-11 02:10:00	1989-05-08 00:00:00
1982-07-12 00:00:00	1982-07-15 02:10:00	1984-01-12 00:00:00
1983-02-16 00:00:00	1983-02-19 02:10:00	1984-08-16 00:00:00
1984-08-07 00:00:00	1984-08-10 02:10:00	1986-02-07 00:00:00
1980-01-09 00:00:00	1980-01-12 02:10:00	1981-07-09 00:00:00
1972-07-18 00:00:00	1972-07-21 02:10:00	1974-01-18 00:00:00

几乎所有版本的MYSQL都支持DATE_ADD()函数的前九种用法，但是MYSQL的早期版本不完全支持DATE_ADD()函数的后几种用法，不过在MYSQL的早期版本中可以嵌套调用DATE_ADD()函数来实现后几种用法的效果。下面的SQL语句使用嵌套函数的方式来分别计算出生日期后1年6个月的日期时间：

```
SELECT FBirthDay,
DATE_ADD(DATE_ADD(FBirthDay,INTERVAL 1 YEAR),INTERVAL 6 MONTH) as dm
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	dm
1981-03-22 00:00:00	1982-09-22 00:00:00
1987-01-18 00:00:00	1988-07-18 00:00:00
1987-11-08 00:00:00	1989-05-08 00:00:00
1982-07-12 00:00:00	1984-01-12 00:00:00
1983-02-16 00:00:00	1984-08-16 00:00:00
1984-08-07 00:00:00	1986-02-07 00:00:00
1980-01-09 00:00:00	1981-07-09 00:00:00
1972-07-18 00:00:00	1974-01-18 00:00:00

DATE_ADD()函数不仅可以用来在日期基础上增加指定的时间段，而且还可以在日期基础上减少指定的时间段，只要在expr参数中使用负数就可以，下面的SQL语句用来计算每个人出生一周、两个月以及5个季度前的日期：

```
SELECT FBirthDay,
DATE_ADD(FBirthDay,INTERVAL -1 WEEK) as w1,
DATE_ADD(FBirthDay,INTERVAL -2 MONTH) as m2,
DATE_ADD(FBirthDay,INTERVAL -5 QUARTER) as q5
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	w1	m2	q5
1981-03-22 00:00:00	1981-03-15 00:00:00	1981-01-22 00:00:00	1979-12-22 00:00:00
1987-01-18 00:00:00	1987-01-11 00:00:00	1986-11-18 00:00:00	1985-10-18 00:00:00
1987-11-08 00:00:00	1987-11-01 00:00:00	1987-09-08 00:00:00	1986-08-08 00:00:00
1982-07-12 00:00:00	1982-07-05 00:00:00	1982-05-12 00:00:00	1981-04-12 00:00:00
1983-02-16 00:00:00	1983-02-09 00:00:00	1982-12-16 00:00:00	1981-11-16 00:00:00
1984-08-07 00:00:00	1984-07-31 00:00:00	1984-06-07 00:00:00	1983-05-07 00:00:00
1980-01-09 00:00:00	1980-01-02 00:00:00	1979-11-09 00:00:00	1978-10-09 00:00:00
1972-07-18 00:00:00	1972-07-11 00:00:00	1972-05-18 00:00:00	1971-04-18 00:00:00

在MYSQL中提供了**DATE_SUB()**函数用于计算指定日期前的特定时间段的日期，其效果和在DATE_ADD()函数中使用负数的expr参数值的效果一样，其用法也和DATE_ADD()函数几乎相同。下面的SQL语句用来计算每个人出生一周、两个月以及3天2小时10分钟前的日期：

```
SELECT FBirthDay,
DATE_SUB(FBirthDay,INTERVAL 1 WEEK) as w1,
DATE_SUB(FBirthDay,INTERVAL 2 MONTH) as m2,
DATE_SUB(FBirthDay, INTERVAL '3 2:10' DAY_MINUTE) as dm
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	w1	m2	dm
1981-03-22 00:00:00	1981-03-15 00:00:00	1981-01-22 00:00:00	1981-03-18 21:50:00
1987-01-18 00:00:00	1987-01-11 00:00:00	1986-11-18 00:00:00	1987-01-14 21:50:00
1987-11-08 00:00:00	1987-11-01 00:00:00	1987-09-08 00:00:00	1987-11-04 21:50:00
1982-07-12 00:00:00	1982-07-05 00:00:00	1982-05-12 00:00:00	1982-07-08 21:50:00
1983-02-16 00:00:00	1983-02-09 00:00:00	1982-12-16 00:00:00	1983-02-12 21:50:00
1984-08-07 00:00:00	1984-07-31 00:00:00	1984-06-07 00:00:00	1984-08-03 21:50:00
1980-01-09 00:00:00	1980-01-02 00:00:00	1979-11-09 00:00:00	1980-01-05 21:50:00
1972-07-18 00:00:00	1972-07-11 00:00:00	1972-05-18 00:00:00	1972-07-14 21:50:00

5.3.4.2 MSSQLServer

MSSQLServer中提供了DATEADD()函数用于进行日期时间的加法运算， DATEADD ()函数的参数格式如下：

DATEADD (datepart , number, date)

其中参数date为待计算的日期；参数date制定了用于与 datepart 相加的值，如果指定了非整数值，则将舍弃该值的小数部分；参数datepart指定要返回新值的日期的组成部分，下表列出了 MicrosoftSQL Server 2005 可识别的日期部分及其缩写：

取值	别名	说明
year	yy,yyyy	年份
quarter	qq,q	季度
month	mm,m	月份
dayofyear	dy,y	当年度的第几天
day	dd,d	日
week	wk,ww	当年度的第几周
weekday	dw,w	星期几

hour	hh	小时
minute	mi,n	分
second	ss,s	秒
millisecond	ms	毫秒

比如 DATEADD(DAY, 3,date)为计算日期 date 的 3 天后的日期，而 DATEADD(**MONTH**,-8,date)为计算日期 date 的 8 个月之前的日期。

下面的 SQL 语句用于计算每个人出生后 3 年、20 个季度、68 个月以及 1000 个周前的日期：

```
SELECT FBirthDay, DATEADD (YEAR ,3,FBirthDay) as threeyrs,
DATEADD(QUARTER ,20,FBirthDay) as ttqutrs,
DATEADD(MONTH ,68,FBirthDay) as sxtmonths,
DATEADD(WEEK, -1000,FBirthDay) as thweeik
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	threeyrs	ttqutrs	xtmonths	thweeik
1981-03-22 00:00:00.0	1984-03-22 00:00:00.0	1986-03-22 00:00:00.0	1986-11-22 00:00:00.0	1962-01-21 00:00:00.0
1987-01-18 00:00:00.0	1990-01-18 00:00:00.0	1992-01-18 00:00:00.0	1992-09-18 00:00:00.0	1967-11-19 00:00:00.0
1987-11-08 00:00:00.0	1990-11-08 00:00:00.0	1992-11-08 00:00:00.0	1993-07-08 00:00:00.0	1968-09-08 00:00:00.0
1982-07-12 00:00:00.0	1985-07-12 00:00:00.0	1987-07-12 00:00:00.0	1988-03-12 00:00:00.0	1963-05-13 00:00:00.0
1983-02-16 00:00:00.0	1986-02-16 00:00:00.0	1988-02-16 00:00:00.0	1988-10-16 00:00:00.0	1963-12-18 00:00:00.0
1984-08-07 00:00:00.0	1987-08-07 00:00:00.0	1989-08-07 00:00:00.0	1990-04-07 00:00:00.0	1965-06-08 00:00:00.0
1980-01-09 00:00:00.0	1983-01-09 00:00:00.0	1985-01-09 00:00:00.0	1985-09-09 00:00:00.0	1960-11-09 00:00:00.0
1972-07-18 00:00:00.0	1975-07-18 00:00:00.0	1977-07-18 00:00:00.0	1978-03-18 00:00:00.0	1953-05-19 00:00:00.0

5.3.4.3 Oracle

Oracle中可以直接使用加号“+”来进行日期的加法运算，其计算单位为“天”，比如date+3就表示在日期date的基础上增加三天；同理使用减号“-”则可以用来计算日期前的特定时间段的时间，比如date+3就表示在日期date的三天前的日期。比如下面的SQL语句用于计算每个人出生日期3天后以及10天前的日期：

```
SELECT FBirthDay,
FBirthDay+3,
FBirthDay-10
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	FBIRTHDAY+3	FBIRTHDAY-10
1981-03-22 00:00:00.0	1981-03-25 00:00:00.0	1981-03-12 00:00:00.0
1987-01-18 00:00:00.0	1987-01-21 00:00:00.0	1987-01-08 00:00:00.0

1987-11-08 00:00:00.0	1987-11-11 00:00:00.0	1987-10-29 00:00:00.0
1982-07-12 00:00:00.0	1982-07-15 00:00:00.0	1982-07-02 00:00:00.0
1983-02-16 00:00:00.0	1983-02-19 00:00:00.0	1983-02-06 00:00:00.0
1984-08-07 00:00:00.0	1984-08-10 00:00:00.0	1984-07-28 00:00:00.0
1980-01-09 00:00:00.0	1980-01-12 00:00:00.0	1979-12-30 00:00:00.0
1972-07-18 00:00:00.0	1972-07-21 00:00:00.0	1972-07-08 00:00:00.0

可以使用换算的方式来进行以周、小时、分钟等为单位的日期加减运算，比如下面的SQL语句用于计算每个人出生日期2小时10分钟后以及3周后的日期：

```
SELECT FBirthDay,
FBirthDay+(2/24+10/60/24),
FBirthDay+(3*7)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	FBIRTHDAY+(2/24+10/60/24)	FBIRTHDAY+(3*7)
1981-03-22 00:00:00.0	1981-03-22 02:10:00.0	1981-04-12 00:00:00.0
1987-01-18 00:00:00.0	1987-01-18 02:10:00.0	1987-02-08 00:00:00.0
1987-11-08 00:00:00.0	1987-11-08 02:10:00.0	1987-11-29 00:00:00.0
1982-07-12 00:00:00.0	1982-07-12 02:10:00.0	1982-08-02 00:00:00.0
1983-02-16 00:00:00.0	1983-02-16 02:10:00.0	1983-03-09 00:00:00.0
1984-08-07 00:00:00.0	1984-08-07 02:10:00.0	1984-08-28 00:00:00.0
1980-01-09 00:00:00.0	1980-01-09 02:10:00.0	1980-01-30 00:00:00.0
1972-07-18 00:00:00.0	1972-07-18 02:10:00.0	1972-08-08 00:00:00.0

使用加减运算我们可以很容易的实现以周、天、小时、分钟、秒等为单位的日期的增减运算，不过由于每个月的天数是不同的，也就是在天和月之间不存在固定的换算率，所以无法使用加减运算实现以月为单位的计算，为此Oracle中提供了ADD_MONTHS()函数用于以月为单位的日期增减运算，ADD_MONTHS()函数的参数格式如下：

```
ADD_MONTHS(date,number)
```

其中参数date为待计算的日期，参数number为要增加的月份数，如果number为负数则表示进行日期的减运算。下面的SQL语句用于计算每个人的出生日期两个月后以及10个月前的日期：

```
SELECT FBirthDay,
ADD_MONTHS(FBirthDay,2),
ADD_MONTHS(FBirthDay,-10)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	ADD_MONTHS(FBIRTHDAY,2)	ADD_MONTHS(FBIRTHDAY,-10)
1981-03-22 00:00:00.0	1981-05-22 00:00:00.0	1980-05-22 00:00:00.0
1987-01-18 00:00:00.0	1987-03-18 00:00:00.0	1986-03-18 00:00:00.0
1987-11-08 00:00:00.0	1988-01-08 00:00:00.0	1987-01-08 00:00:00.0
1982-07-12 00:00:00.0	1982-09-12 00:00:00.0	1981-09-12 00:00:00.0
1983-02-16 00:00:00.0	1983-04-16 00:00:00.0	1982-04-16 00:00:00.0
1984-08-07 00:00:00.0	1984-10-07 00:00:00.0	1983-10-07 00:00:00.0
1980-01-09 00:00:00.0	1980-03-09 00:00:00.0	1979-03-09 00:00:00.0
1972-07-18 00:00:00.0	1972-09-18 00:00:00.0	1971-09-18 00:00:00.0

综合使用ADD_MONTHS()函数和加、减号运算符则可以实现更加复杂的日期增减运算，比如下面的SQL语句用于计算每个人的出生日期两个月零10天后以及3个月零10个小时前的日期时间：

```
SELECT FBirthDay,
ADD_MONTHS(FBirthDay,2)+10 as bfd,
ADD_MONTHS(FBirthDay,-3)-(10/24) as afd
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	BFD	AFD
1981-03-22 00:00:00.0	1981-06-01 00:00:00.0	1980-12-21 14:00:00.0
1987-01-18 00:00:00.0	1987-03-28 00:00:00.0	1986-10-17 14:00:00.0
1987-11-08 00:00:00.0	1988-01-18 00:00:00.0	1987-08-07 14:00:00.0
1982-07-12 00:00:00.0	1982-09-22 00:00:00.0	1982-04-11 14:00:00.0
1983-02-16 00:00:00.0	1983-04-26 00:00:00.0	1982-11-15 14:00:00.0
1984-08-07 00:00:00.0	1984-10-17 00:00:00.0	1984-05-06 14:00:00.0
1980-01-09 00:00:00.0	1980-03-19 00:00:00.0	1979-10-08 14:00:00.0
1972-07-18 00:00:00.0	1972-09-28 00:00:00.0	1972-04-17 14:00:00.0

5.3.4.4 DB2

DB2 中可以直接使用加减运算符进行日期的增减运算，只要在要增减的数目后加上单位就可以了。其使用格式如下：

date+length unit

其中 date 参数为待计算的日期；

length 为进行增减运算的日期，当 length 为正值的时候为向时间轴正向计算，而当 length 为负值的时候为向时间轴负向计算；

unit 为进行计算的单位，此参数可取值以及响应函数如下：

计算单位	说明
YEAR	年
MONTH	月
DAY	日
HOUR	小时
MINUTE	分
SECOND	秒

比如 date+3 DAY 为计算日期 date 的 3 天后的日期，而 date-8 MONTH 为计算日期 date 的 8 个月之前的日期，而且我们还可以连续使用加减运算符进行更加复杂的日期运算，比如 date+3 YEAR+10 DAY 用于计算 date 的 3 个月零 10 天后的日期。

下面的 SQL 语句用于计算每个人出生后 3 年、20 个季度、68 个月以及 1000 个周前的日期：

```
SELECT FBirthDay, FBirthDay+3 YEAR + 10 DAY,
FBirthDay-100 MONTH
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	2	3
1981-03-22	1984-04-01	1972-11-22
1987-01-18	1990-01-28	1978-09-18

1987-11-08	1990-11-18	1979-07-08
1982-07-12	1985-07-22	1974-03-12
1983-02-16	1986-02-26	1974-10-16
1984-08-07	1987-08-17	1976-04-07
1980-01-09	1983-01-19	1971-09-09
1972-07-18	1975-07-28	1964-03-18

5.3.5 计算日期差额

有时候我们需要计算两个日期的差额，比如计算“回款日”和“验收日”之间所差的天数或者检索所有“最后一次登录日期”与当前日期的差额大于100天的用户信息。主流的数据库系统中都提供了对计算日期差额的支持，下面分别进行介绍。

5.3.5.1 MYSQL

MYSQL中使用DATEDIFF()函数用于计算两个日期之间的差额，其参数调用格式如下：

DATEDIFF(date1,date2)

函数将返回date1与date2之间的天数差额，如果date2在date1之后返回正值，否则返回负值。

比如下面的SQL语句用于计算注册日期和出生日期之间的天数差额：

```
SELECT FRegDay,FBirthDay, DATEDIFF(FRegDay, FBirthDay) , DATEDIFF(FBirthDay ,FRegDay)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FRegDay	FBirthDay	DATEDIFF(FRegDay, FBirthDay)	DATEDIFF(FBirthDay ,FRegDay)
1998-05-01 00:00:00	1981-03-22 00:00:00	6249	-6249
1999-08-21 00:00:00	1987-01-18 00:00:00	4598	-4598
2001-09-18 00:00:00	1987-11-08 00:00:00	5063	-5063
2000-03-01 00:00:00	1982-07-12 00:00:00	6442	-6442
1998-05-01 00:00:00	1983-02-16 00:00:00	5553	-5553
1999-03-01 00:00:00	1984-08-07 00:00:00	5319	-5319
2002-09-23 00:00:00	1980-01-09 00:00:00	8293	-8293
1995-06-19 00:00:00	1972-07-18 00:00:00	8371	-8371

DATEDIFF()函数只能计算两个日期之间的天数差额，如果要计算两个日期的周差额等就需要进行换算，比如下面的SQL语句用于计算注册日期和出生日期之间的周数差额：

```
SELECT FRegDay,FBirthDay, DATEDIFF(FRegDay, FBirthDay)/7
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FRegDay	FBirthDay	DATEDIFF(FRegDay, FBirthDay)/7

1998-05-01 00:00:00	1981-03-22 00:00:00	892.7143
1999-08-21 00:00:00	1987-01-18 00:00:00	656.8571
2001-09-18 00:00:00	1987-11-08 00:00:00	723.2857
2000-03-01 00:00:00	1982-07-12 00:00:00	920.2857
1998-05-01 00:00:00	1983-02-16 00:00:00	793.2857
1999-03-01 00:00:00	1984-08-07 00:00:00	759.8571
2002-09-23 00:00:00	1980-01-09 00:00:00	1184.7143
1995-06-19 00:00:00	1972-07-18 00:00:00	1195.8571

5.3.5.2 MSSQLServer

MSSQLServer中同样提供了DATEDIFF()函数用于计算两个日期之间的差额，与MYSQL中的DATEDIFF()函数不同，它提供了一个额外的参数用于指定计算差额时使用的单位，其参数调用格式如下：

DATEDIFF (datepart , startdate , enddate)

其中参数datepart为计算差额时使用的单位，可选值如下：

单位	别名	说明
year	yy, yyyy	年
quarter	qq, q	季度
month	mm, m	月
dayofyear	dy, y	工作日
day	dd, d	天数
week	wk, ww	周
Hour	hh	小时
minute	mi, n	分钟
second	ss, s	秒
millisecond	ms	毫秒

参数startdate为起始日期；参数enddate为结束日期。

下面的SQL语句用于计算注册日期和出生日期之间的周数差额：

SELECT FRegDay,FBirthDay,DATEDIFF(WEEK, FBirthDay, FRegDay) FROM T_Person

执行完毕我们就能在输出结果中看到下面的执行结果：

FRegDay	FBirthDay	
1998-05-01 00:00:00.0	1981-03-22 00:00:00.0	892
1999-08-21 00:00:00.0	1987-01-18 00:00:00.0	656
2001-09-18 00:00:00.0	1987-11-08 00:00:00.0	723
2000-03-01 00:00:00.0	1982-07-12 00:00:00.0	920
1998-05-01 00:00:00.0	1983-02-16 00:00:00.0	793
1999-03-01 00:00:00.0	1984-08-07 00:00:00.0	760
2002-09-23 00:00:00.0	1980-01-09 00:00:00.0	1185
1995-06-19 00:00:00.0	1972-07-18 00:00:00.0	1196

5.3.5.3 Oracle

在Oracle中，可以在两个日期类型的数据之间使用减号运算符“-”，其计算结果为两个日期之间的天数差，比如执行下面的SQL语句用于计算注册日期FRegDay和出生日期FBirthDay之间的时间间隔：

```
SELECT FRegDay,FBirthDay,FRegDay-FBirthDay
```

```
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FREGDAY	FBIRTHDAY	FREGDAY-FBIRTHDAY
1998-05-01 00:00:00.0	1981-03-22 00:00:00.0	6249
1999-08-21 00:00:00.0	1987-01-18 00:00:00.0	4598
2001-09-18 00:00:00.0	1987-11-08 00:00:00.0	5063
2000-03-01 00:00:00.0	1982-07-12 00:00:00.0	6442
1998-05-01 00:00:00.0	1983-02-16 00:00:00.0	5553
1999-03-01 00:00:00.0	1984-08-07 00:00:00.0	5319
2002-09-23 00:00:00.0	1980-01-09 00:00:00.0	8293
1995-06-19 00:00:00.0	1972-07-18 00:00:00.0	8371

注意通过减号运算符“-”计算的两个日期之间的天数差是包含有小数部分的，小数部分表示不足一天的部分，比如执行下面的SQL语句用于计算当前时刻和出生日期FBirthDay之间的时间间隔：

```
SELECT SYSDATE,FBirthDay,SYSDATE-FBirthDay
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

SYSDATE	FBIRTHDAY	SYSDATE-FBIRTHDAY
2008-01-16 23:11:52.0	1981-03-22 00:00:00.0	9796.9665740740740740740740740740740740
2008-01-16 23:11:52.0	1987-01-18 00:00:00.0	7668.96657407407407407407407407407407407
2008-01-16 23:11:52.0	1987-11-08 00:00:00.0	7374.96657407407407407407407407407407407
2008-01-16 23:11:52.0	1982-07-12 00:00:00.0	9319.96657407407407407407407407407407407
2008-01-16 23:11:52.0	1983-02-16 00:00:00.0	9100.96657407407407407407407407407407407
2008-01-16 23:11:52.0	1984-08-07 00:00:00.0	8562.96657407407407407407407407407407407
2008-01-16 23:11:52.0	1980-01-09 00:00:00.0	10234.9665740740740740740740740740741
2008-01-16 23:11:52.0	1972-07-18 00:00:00.0	12965.9665740740740740740740740740741

可以看到天数差的小数部分是非常精确的，所以完全可以精确的表示两个日期时间值之间差的小时、分、秒甚至毫秒部分。所以如果要计算两个日期时间值之间的小时、分、秒以及毫秒差的话，只要进行相应的换算就可以，比如下面的SQL用来计算当前时刻和出生日期FBirthDay之间的时间间隔（小时、分以及秒）：

```
SELECT (SYSDATE-FBirthDay)*24,(SYSDATE-FBirthDay)*24*60,
(SYSDATE-FBirthDay)*24*60*60
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

(SYSDATE-FBIRTHDAY)*24	(SYSDATE-FBIRTHDAY)*24*60	(SYSDATE-FBIRTHDAY)*24*60*60

235127.2891666666666666666666666666667	14107637.35	846458241
184055.2891666666666666666666666666667	11043317.35	662599041
176999.2891666666666666666666666666667	10619957.35	637197441
223679.2891666666666666666666666666667	13420757.35	805245441
218423.2891666666666666666666666666667	13105397.35	786323841
205511.2891666666666666666666666666667	12330677.35	739840641
245639.2891666666666666666666666666656	14738357.34999999999999999999999999994	884301440.9999999999999999999999999996
311183.2891666666666666666666666666656	18670997.34999999999999999999999999994	1120259840.99999999999999999999999999996

下面的SQL语句用来计算当前时刻和出生日期FBirthDay之间的周间隔：

```
SELECT SYSDATE,FBirthDay,(SYSDATE-FBirthDay)/7
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

SYSDATE	FBIRTHDAY	(SYSDATE-FBIRTHDAY)/7
2008-01-16 23:22:17.0	1981-03-22 00:00:00.0	1399.567686838624338624338624338624338624
2008-01-16 23:22:17.0	1987-01-18 00:00:00.0	1095.567686838624338624338624338624338624
2008-01-16 23:22:17.0	1987-11-08 00:00:00.0	1053.567686838624338624338624338624338624
2008-01-16 23:22:17.0	1982-07-12 00:00:00.0	1331.424829695767195767195767195767195767
2008-01-16 23:22:17.0	1983-02-16 00:00:00.0	1300.139115410052910052910052910052910053
2008-01-16 23:22:17.0	1984-08-07 00:00:00.0	1223.28197255291005291005291005291005291
2008-01-16 23:22:17.0	1980-01-09 00:00:00.0	1462.139115410052910052910052910052910057
2008-01-16 23:22:17.0	1972-07-18 00:00:00.0	1852.281972552910052910052910052910052914

可以看到计算结果含有非常精确的小数部分，不过如果对这些小数部分没有需求的话则可以使用数值函数进行四舍五入、取最大整数等处理，比如下面的SQL用来计算当前时刻和出生日期FBirthDay之间的时间间隔（小时、分以及秒），并且对于计算结果进行四舍五入运算：

```
SELECT
ROUND((SYSDATE-FBirthDay)*24),ROUND((SYSDATE-FBirthDay)*24*60),
ROUND((SYSDATE-FBirthDay)*24*60*60)
```

```
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

ROUND((SYSDATE-FBIRTHDAY)*24)	ROUND((SYSDATE-FBIRTHDAY)*24*60)	ROUND((SYSDATE-FBIRTHDAY)*24*60*60)
235127	14107641	846458455
184055	11043321	662599255
176999	10619961	637197655
223679	13420761	805245655
218423	13105401	786324055
205511	12330681	739840855
245639	14738361	884301655
311183	18671001	1120260055

5.3.5.4 DB2

DB2中提供了DAYS()函数，这个函数接受一个时间日期类型的参数，返回结果为从0001年1月1日到此日期的天数，比如下面的SQL语句用于计算出生日期FBirthDay、注册日期FRegDay以及当前日期距0001年1月1日的天数差：

```
SELECT DAYS(FBirthDay),DAYS(FRegDay),DAYS(CURRENT DATE)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1	2	3
723261	729510	733057
725389	729987	733057
725683	730746	733057
723738	730180	733057
723957	729510	733057
724495	729814	733057
722823	731116	733057
720092	728463	733057

借助于DAYS()函数我们可以轻松计算两个日期之间的天数间隔，很显然DAYS(date1)-DAYS(date2)的计算结果就是日期date1和日期date2之间的天数间隔，比如下面的SQL语句用于计算出生日期FBirthDay与注册日期FRegDay之间的天数间隔：

```
SELECT FBirthDay,FRegDay,
DAYS(FRegDay)-DAYS(FBirthDay)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	FREGDAY	3
1981-03-22	1998-05-01	6249
1987-01-18	1999-08-21	4598
1987-11-08	2001-09-18	5063
1982-07-12	2000-03-01	6442
1983-02-16	1998-05-01	5553
1984-08-07	1999-03-01	5319
1980-01-09	2002-09-23	8293

1972-07-18	1995-06-19	8371

如果要计算两个日期时间值之间的周间隔的话，只要进行相应的换算就可以，比如下面的SQL用来计算出生日期FBirthDay和注册日期FRegDay之间的周数间隔：

```
SELECT FBirthDay,FRegDay,
(DAYS(FRegDay)-DAYS(FBirthDay))/7
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	FREGDAY	3
1981-03-22	1998-05-01	892
1987-01-18	1999-08-21	656
1987-11-08	2001-09-18	723
1982-07-12	2000-03-01	920
1983-02-16	1998-05-01	793
1984-08-07	1999-03-01	759
1980-01-09	2002-09-23	1184
1972-07-18	1995-06-19	1195

5.3.6 计算一个日期是星期几

计算一个日期是星期几是非常有用的，比如如果安排的报到日期是周末那么就向后拖延报到日期，在主流数据库中对这个功能都提供了很好的支持，下面分别进行介绍。

5.3.6.1 MYSQL

MYSQL中提供了DAYNAME()函数用于计算一个日期是星期几，比如下面的SQL语句用于计算出生日期和注册日期各是星期几：

```
SELECT FBirthDay,DAYNAME(FBirthDay),
FRegDay,DAYNAME(FRegDay)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	DAYNAME(FBirthDay)	FRegDay	DAYNAME(FRegDay)
1981-03-22 00:00:00	Sunday	1998-05-01 00:00:00	Friday
1987-01-18 00:00:00	Sunday	1999-08-21 00:00:00	Saturday
1987-11-08 00:00:00	Sunday	2001-09-18 00:00:00	Tuesday
1982-07-12 00:00:00	Monday	2000-03-01 00:00:00	Wednesday
1983-02-16 00:00:00	Wednesday	1998-05-01 00:00:00	Friday
1984-08-07 00:00:00	Tuesday	1999-03-01 00:00:00	Monday
1980-01-09 00:00:00	Wednesday	2002-09-23 00:00:00	Monday
1972-07-18 00:00:00	Tuesday	1995-06-19 00:00:00	Monday

注意MYSQL中DAYNAME()函数返回的是英文的日期表示法。

5.3.6.2 MSQLServer

MSQLServer中提供了DATENAME()函数，这个函数可以返回一个日期的特定部分，并且尽量用名称来表述这个特定部分，其参数格式如下：

DATENAME(datepart,date)

其中参数date为待计算日期，date 参数也可以是日期格式的字符串；参数datepart指定要返回的日期部分的参数，其可选值如下：

可选值	别名	说明

Year	yy、yyyy	年份
Quarter	qq, q	季度
Month	mm, m	月份
Dayofyear	dy, y	每年的某一日
Day	dd, d	日期
Week	wk, ww	星期
Weekday	dw	工作日
Hour	hh	小时
Minute	mi, n	分钟
Second	ss, s	秒
Millisecond	ms	毫秒

如果使用Weekday（或者使用别名dw）做为datepart参数调用DATENAME()函数就可以得到一个日期是星期几，比如下面的SQL语句用于计算出生日期和注册日期各是星期几：

```
SELECT FBirthDay,DATENAME(Weekday,FBirthDay),
FRegDay,DATENAME(DW, FRegDay)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay		FRegDay	
1981-03-22 00:00:00.0	星期日	1998-05-01 00:00:00.0	星期五
1987-01-18 00:00:00.0	星期日	1999-08-21 00:00:00.0	星期六
1987-11-08 00:00:00.0	星期日	2001-09-18 00:00:00.0	星期二
1982-07-12 00:00:00.0	星期一	2000-03-01 00:00:00.0	星期三
1983-02-16 00:00:00.0	星期三	1998-05-01 00:00:00.0	星期五
1984-08-07 00:00:00.0	星期二	1999-03-01 00:00:00.0	星期一
1980-01-09 00:00:00.0	星期三	2002-09-23 00:00:00.0	星期一
1972-07-18 00:00:00.0	星期二	1995-06-19 00:00:00.0	星期一

5.3.6.3 Oracle

Oracle中提供了TO_CHAR()函数用于将数据转换为字符串类型，当针对时间日期类型数据进行转换的时候，它接受两个参数，其参数格式如下：

TO_CHAR(date,format)

其中参数date为待转换的日期，参数format为格式化字符串，数据库系统将按照这个字符串对date进行转换，格式化字符串中可以采用如下的占位符：

占位符	说明
YEAR	年份（英文拼写），比如NINETEEN NINETY-EIGHT
YYYY	4位年份，比如1998

YYY	年份后3位，比如998
YY	年份后2位，比如98
Y	年份后1位，比如8
IYYY	符合ISO标准的4位年份，比如1998
IYY	符合ISO标准的年份后3位，比如998
IY	符合ISO标准的年份后2位，比如98
I	符合ISO标准的年份后1位，比如8
Q	以整数表示的季度，比如1
MM	以整数表示的月份，比如01
MON	月份的名称，比如2月
MONTH	月份的名称，补足9个字符
RM	罗马表示法的月份，比如VIII
WW	日期属于当年的第几周，比如30
W	日期属于当月的第几周，比如2
IW	日期属于当年的第几周（按照ISO标准），比如30
D	日期属于周几，以整数表示，返回值范围为1至7
DAY	日期属于周几，以名字的形式表示，比如星期五
DD	日期属于当月的第几天，比如2
DDD	日期属于当年的第几天，比如168
DY	日期属于周几，以名字的形式表示，比如星期五
HH	小时部分（12小时制）
HH12	小时部分（12小时制）
HH24	小时部分（24小时制）
MI	分钟部分
SS	秒部分
SSSSS	自从午夜开始的秒数

可以简单的将占位符做为参数传递给TO_CHAR()函数，下面的SQL语句用于计算出生日期的年份、月份以及周数：

```
SELECT FBirthDay,
TO_CHAR(FBirthDay, 'YYYY') as yyyy,
TO_CHAR(FBirthDay, 'MM') as mm,
TO_CHAR(FBirthDay, 'MON') as mon,
TO_CHAR(FBirthDay, 'WW') as ww
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	YYYY	MM	MON	WW
1981-03-22 00:00:00.0	1981	03	3 月	12
1987-01-18 00:00:00.0	1987	01	1 月	03
1987-11-08 00:00:00.0	1987	11	11 月	45
1982-07-12	1982	07	7 月	28

00:00:00.0				
1983-02-16 00:00:00.0	1983	02	2 月	07
1984-08-07 00:00:00.0	1984	08	8 月	32
1980-01-09 00:00:00.0	1980	01	1 月	02
1972-07-18 00:00:00.0	1972	07	7 月	29

同样还可以将占位符组合起来实现更加复杂的转换逻辑，比如下面的SQL语句用于以“2008-08-08”这样的形式显示出生日期以及以“31-2007-02”这样的形式显示注册日期：

```
SELECT FBirthDay,
TO_CHAR(FBirthDay, 'YYYY-MM-DD') as yyymmdd,
FRegDay,
TO_CHAR(FRegDay, 'DD-YYYY-MM') as ddyyyymm
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	YYYMMDD	FREGDAY	DDYYYYMM
1981-03-22 00:00:00.0	1981-03-22	1998-05-01 00:00:00.0	01-1998-05
1987-01-18 00:00:00.0	1987-01-18	1999-08-21 00:00:00.0	21-1999-08
1987-11-08 00:00:00.0	1987-11-08	2001-09-18 00:00:00.0	18-2001-09
1982-07-12 00:00:00.0	1982-07-12	2000-03-01 00:00:00.0	01-2000-03
1983-02-16 00:00:00.0	1983-02-16	1998-05-01 00:00:00.0	01-1998-05
1984-08-07 00:00:00.0	1984-08-07	1999-03-01 00:00:00.0	01-1999-03
1980-01-09 00:00:00.0	1980-01-09	2002-09-23 00:00:00.0	23-2002-09
1972-07-18 00:00:00.0	1972-07-18	1995-06-19 00:00:00.0	19-1995-06

我们前面提到了，当用“DAY”做为参数的时候就可以将日期格式化为名字的形式表示的星期几，比如下面的SQL语句用于计算出生日期以及注册日期各属于星期几：

```
SELECT
FBirthDay,TO_CHAR(FBirthDay, 'DAY') as birthwk,
FRegDay,TO_CHAR(FRegDay, 'DAY') as regwk
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	BIRTHWK	FREGDAY	REGWK
1981-03-22 00:00:00.0	星期日	1998-05-01 00:00:00.0	星期五

1987-01-18 00:00:00.0	星期日	1999-08-21 00:00:00.0	星期六
1987-11-08 00:00:00.0	星期日	2001-09-18 00:00:00.0	星期二
1982-07-12 00:00:00.0	星期一	2000-03-01 00:00:00.0	星期三
1983-02-16 00:00:00.0	星期三	1998-05-01 00:00:00.0	星期五
1984-08-07 00:00:00.0	星期二	1999-03-01 00:00:00.0	星期一
1980-01-09 00:00:00.0	星期三	2002-09-23 00:00:00.0	星期一
1972-07-18 00:00:00.0	星期二	1995-06-19 00:00:00.0	星期一

5.3.6.4 DB2

DB2中提供了DAYNAME()函数用于计算一个日期是星期几，执行下面的SQL语句我们可以得到出生日期和注册日期各是星期几：

```
SELECT
FBirthDay,DAYNAME(FBirthDay) as birthwk,
FRegDay,DAYNAME(FRegDay) as regwk
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	BIRTHWK	FREGDAY	REGWK
1981-03-22	星期日	1998-05-01	星期五
1987-01-18	星期日	1999-08-21	星期六
1987-11-08	星期日	2001-09-18	星期二
1982-07-12	星期一	2000-03-01	星期三
1983-02-16	星期三	1998-05-01	星期五
1984-08-07	星期二	1999-03-01	星期一
1980-01-09	星期三	2002-09-23	星期一
1972-07-18	星期二	1995-06-19	星期一

5.3.7 取得日期的指定部分

提取日期的特定部分是非常有必要的，比如检索本年的每个月的16日的销售量、检索访问用户集中的时间段，这些都需要对日期的特定部分进行提取，在主流数据库中对这个功能都提供了很好的支持，下面分别进行介绍。

5.3.7.1 MYSQL

MYSQL中提供了一个DATE_FORMAT()函数用来将日期按照特定各是进行格式化，这个函数的参数格式如下：

DATE_FORMAT(date,format)

这个函数用来按照特定的格式化指定的日期，其中参数date为待计算的日期值，而参数format为格式化字符串，格式化字符串中可以采用如下的占位符：

占位符	说明
%a	缩写的星期几(**Sun..Sat**)
%b	缩写的月份名(**Jan..Dec**)

%c	数字形式的月份(0..12)
%D	当月的第几天，带英文后缀(0th, 1st, 2nd, 3rd, ...)
%d	当月的第几天，两位数字形式，不足两位则补零(00..31)
%e	当月的第几天，数字形式(0..31)
%f	毫秒
%H	24小时制的小时 (00..23)
%h	12小时制的小时(01..12)
%I	12小时制的小时(01..12)
%i	数字形式的分钟(00..59)
%j	日期在当年中的天数(001..366)
%k	24进制小时(0..23)
%l	12进制小时(1..12)
%M	月份名(January..December)
%m	两位数字表示的月份(00..12)
%p	上午还是下午（AM.. PM）
%r	12小时制时间，比如08:09:29 AM
%S	秒数(00..59)
%s	秒数(00..59)
%T	时间，24小时制，格式为hh:mm:ss
%U	所属周是当年的第几周，周日当作第一天(00..53)
%u	所属周是当年的第几周，周一当作第一天(00..53)
%V	所属周是当年的第几周，周日当作第一天(01..53)
%v	所属周是当年的第几周，周一当作第一天(01..53)
%W	星期几(Sunday..Saturday)
%w	星期几，数字形式(0=Sunday..6=Saturday)
%X	本周所属年，周日当作第一天
%x	本周所属年，周一当作第一天
%Y	年份数，四位数字
%y	年份数，两位数字

组合使用这些占位符就可以实现非常复杂的字符串格式化逻辑，比如下面的SQL语句实现了将出生日期FBirthDay和注册日期FRegDay分别按照两种格式进行格式化：

```
SELECT
FBirthDay,
DATE_FORMAT(FBirthDay,'%y-%M %D %W') AS bd,
FRegDay,
DATE_FORMAT(FRegDay,'%Y年%m月%e日') AS rd
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	bd	FRegDay	rd
1981-03-22 00:00:00	81-March 22nd Sunday	1998-05-01 00:00:00	1998 年 05 月 1 日
1987-01-18 00:00:00	87-January 18th Sunday	1999-08-21 00:00:00	1999 年 08 月 21 日

1987-11-08 00:00:00	87-November 8th Sunday	2001-09-18 00:00:00	2001 年 09 月 18 日
1982-07-12 00:00:00	82-July 12th Monday	2000-03-01 00:00:00	2000 年 03 月 1 日
1983-02-16 00:00:00	83-February 16th Wednesday	1998-05-01 00:00:00	1998 年 05 月 1 日
1984-08-07 00:00:00	84-August 7th Tuesday	1999-03-01 00:00:00	1999 年 03 月 1 日
1980-01-09 00:00:00	80-January 9th Wednesday	2002-09-23 00:00:00	2002 年 09 月 23 日
1972-07-18 00:00:00	72-July 18th Tuesday	1995-06-19 00:00:00	1995 年 06 月 19 日

很显然，如果只使用单独的占位符那么就可以实现提取日期特定部分的功能了，比如DATE_FORMAT(date,'%Y')可以用来提取日期的年份部分、DATE_FORMAT(date,'%H')可以用来提取日期的小时部分、DATE_FORMAT(date,'%M')可以用来提取日期的月份名称。下面的SQL用于提取每个人员的出生年份、出生时是当年的第几天、出生时是当年的第几周：

```
SELECT
FBirthDay,
DATE_FORMAT(FBirthDay,'%Y') AS y,
DATE_FORMAT(FBirthDay,'%j') AS d,
DATE_FORMAT(FBirthDay,'%U') AS u
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	y	d	u
1981-03-22 00:00:00	1981	081	12
1987-01-18 00:00:00	1987	018	03
1987-11-08 00:00:00	1987	312	45
1982-07-12 00:00:00	1982	193	28
1983-02-16 00:00:00	1983	047	07
1984-08-07 00:00:00	1984	220	32
1980-01-09 00:00:00	1980	009	01
1972-07-18 00:00:00	1972	200	29

5.3.7.2 MSSQLServer

在5.3.6.2一节中我们介绍了DATENAME()函数，使用它就可以提取日期的任意部分，比如下面的SQL用于提取每个人员的出生年份、出生时是当年的第几天、出生时是当年的第几周：

```
SELECT
FBirthDay,
DATENAME(year,FBirthDay) AS y,
DATENAME(dayofyear,FBirthDay) AS d,
DATENAME(week,FBirthDay) AS u
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	y	d	u
1981-03-22 00:00:00.0	1981	81	13
1987-01-18 00:00:00.0	1987	18	4
1987-11-08 00:00:00.0	1987	312	46

1982-07-12 00:00:00.0	1982	193	29
1983-02-16 00:00:00.0	1983	47	8
1984-08-07 00:00:00.0	1984	220	32
1980-01-09 00:00:00.0	1980	9	2
1972-07-18 00:00:00.0	1972	200	30

在MSSQLServer中还提供了一个DATEPART()函数，这个函数也可以用来返回一个日期的特定部分，其参数格式如下：

DATEPART (datepart,date)

其中参数date为待计算日期，date 参数也可以是日期格式的字符串；参数datepart指定要返回的日期部分的参数，其可选值如下：

可选值	别名	说明
Year	yy、yyyy	年份
Quarter	qq, q	季度
Month	mm, m	月份
Dayofyear	dy, y	每年的某一日
Day	dd, d	日期
Week	wk, ww	星期
Weekday	dw	工作日
Hour	hh	小时
Minute	mi, n	分钟
Second	ss, s	秒
Millisecond	ms	毫秒

显然使用Dayofyear做为datepart参数调用DATEPART ()函数就可以得到一个日期是当年的第几天；使用Year做为datepart参数调用DATEPART ()函数就可以得到一个日期的年份数；以此类推……。下面的SQL语句用于计算出生日期是当年第几天以及注册日期中的年份部分：

```
SELECT FBirthDay, DATEPART(Dayofyear,FBirthDay),
FRegDay, DATEPART(Year, FRegDay)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay		FRegDay	
1981-03-22 00:00:00.0	81	1998-05-01 00:00:00.0	1998
1987-01-18 00:00:00.0	18	1999-08-21 00:00:00.0	1999
1987-11-08 00:00:00.0	312	2001-09-18 00:00:00.0	2001
1982-07-12 00:00:00.0	193	2000-03-01 00:00:00.0	2000
1983-02-16 00:00:00.0	47	1998-05-01 00:00:00.0	1998
1984-08-07 00:00:00.0	220	1999-03-01 00:00:00.0	1999
1980-01-09 00:00:00.0	9	2002-09-23 00:00:00.0	2002
1972-07-18 00:00:00.0	200	1995-06-19 00:00:00.0	1995

粗看起来，DATEPART()函数和DATENAME()函数完全一样，不过其实它们并不是只是名称不同的别名函数，虽然都是用来提取日期的特定部分的，不过DATEPART()函数的返回值是数字而DATENAME()函数则会将尽可能的以名称的方式做为返回值。

5.3.7.3 Oracle

在5.3.6.3一节中我们介绍了Oracle中使用TO_CHAR()函数格式化日期的方法，使用它就可

以提取日期的任意部分，比如下面的SQL用于提取每个人员的出生年份、出生时是当年的第几天、出生时是当年的第几周：

```
SELECT
FBirthDay,
TO_CHAR(FBirthDay,'YYYY') AS y,
TO_CHAR(FBirthDay,'DDD') AS d,
TO_CHAR(FBirthDay,'WW') AS u
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	Y	D	U
1981-03-22 00:00:00.0	1981	081	12
1987-01-18 00:00:00.0	1987	018	03
1987-11-08 00:00:00.0	1987	312	45
1982-07-12 00:00:00.0	1982	193	28
1983-02-16 00:00:00.0	1983	047	07
1984-08-07 00:00:00.0	1984	220	32
1980-01-09 00:00:00.0	1980	009	02
1972-07-18 00:00:00.0	1972	200	29

5.3.7.4 DB2

DB2中没有提供像MYSQL、Oracle中那样的日期格式化函数，也没有提供像MSSQLServer中DATENAME()那样通用的取日期的特定部分的函数，DB2中对于提取日期的不同的部分需要使用不同的函数，这些函数的列表如下：

函数名	功能说明
YEAR()	取参数的年份部分
MONTH()	取参数的月份部分，返回值为整数
MONTHNAME()	对于参数的月部分的月份，返回一个大小写混合的字符串（例如，January）。
QUARTER()	取参数的季度数
DAYOFYEAR()	返回参数中一年中的第几天，用范围在 1-366 的整数值表示。
DAY()	取参数的日部分
DAYNAME()	返回一个大小写混合的字符串，对于参数的日部分，用星期表示这一天的名称（例如，Friday）。
WEEK()	返回参数是一年中的第几周
DAYOFWEEK()	返回参数中的星期几，用范围在 1-7 的整数值表示，其中 1 代表星期日。
HOUR()	取参数的小时部分
MINUTE()	取参数的分钟部分
SECOND()	取参数的秒钟部分
MICROSECOND()	取参数的微秒部分

下面的SQL语句用于计算出生日期的年份部分并且计算注册日期的月份名以及是一年中的第几周：

```
SELECT
FBirthDay,
YEAR(FBirthDay),
```

```
FRegDay,
MONTHNAME(FRegDay),
WEEK(FRegDay)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	2	FREGDAY	4	5
1981-03-22	1981	1998-05-01	五月	18
1987-01-18	1987	1999-08-21	八月	34
1987-11-08	1987	2001-09-18	九月	38
1982-07-12	1982	2000-03-01	三月	10
1983-02-16	1983	1998-05-01	五月	18
1984-08-07	1984	1999-03-01	三月	10
1980-01-09	1980	2002-09-23	九月	39
1972-07-18	1972	1995-06-19	六月	25

5.4 其他函数

除了数学函数、字符串函数、日期函数之外，数据库中还有其他一些函数，比如进行类型转换的函数、进行非空逻辑判断的函数等，这些函数也是非常重要的，因此在本节中我们将对这些函数进行介绍。

5.4.1 类型转换

在使用SQL语句的时候，我们使用的数据的类型不一定符合函数或者运算符的需要，比如函数需要整数类型的数据而我们使用的则是一个字符串，在一些情况下数据库系统会替我们自动将字符串类型转换为整数类型，这种转换称为隐式转换。但是在有的情况下数据库系统不会进行隐式转换，这时就要使用类型转换函数了，这种转换称为显式转换。使用类型转换函数不仅可以保证类型转换的正确性，而且可以提高数据处理的速度，因此应该使用显式转换，尽量避免使用隐式转换。

在主流数据库系统中都提供了类型转换函数，下面分别进行介绍。

5.4.1.1 MYSQL

MYSQL中提供了CAST()函数和CONVERT()函数用于进行类型转换，CAST()是符合ANSI SQL99的函数，CONVERT() 是符合ODBC标准的函数，这两个函数只是参数的调用方式略有差异，其功能几乎相同。这两个函数的参数格式如下：

```
CAST(expression AS type)
CONVERT(expression,type)
```

参数expression为待进行类型转换的表达式，而type为转换的目标类型，type可以是下面的任一个：

可选值	缩写	说明
BINARY		BINARY字符串
CHAR		字符串类型
DATE		日期类型
DATETIME		时间日期类型
SIGNED INTEGER	SIGNED	有符号整数
TIME		时间类型
UNSIGNED INTEGER	UNSIGNED	无符号整数

下面的SQL语句分别演示以有符号整形、无符号整形、日期类型、时间类型为目标类型的数据转换：

```
SELECT
CAST('-30' AS SIGNED) as sig,
CONVERT ('36', UNSIGNED INTEGER) as usig,
CAST('2008-08-08' AS DATE) as d,
CONVERT ('08:09:10', TIME) as t
```

执行完毕我们就能在输出结果中看到下面的执行结果：

sig	usig	d	t
-30	36	2008-08-08	08:09:10

5.4.1.2 MSSQLServer

与MYSQL类似，MSSQLServer中同样提供了名称为CAST()和CONVERT()两个函数用于进行类型转换，CAST()是符合ANSI SQL99的函数，CONVERT() 是符合ODBC标准的函数。与MYSQL中的CONVERT()函数不同的是MSSQLServer中的CONVERT()函数参数顺序正好与MYSQL中的CONVERT()函数参数顺序相反。这两个函数的参数格式如下：

CAST (expression AS data_type)

CONVERT (data_type, expression)

参数expression为待进行类型转换的表达式，而type为转换的目标类型，与MYSQL不同，MYSQLServer中的目标类型几乎可以是数据库系统支持的任何类型。

下面的SQL语句分别演示以整形、数值、日期时间类型为目标类型的数据转换：

```
SELECT
CAST('-30' AS INTEGER) as i,
CONVERT(DECIMAL,'3.1415726') as d,
CONVERT(DATETIME,'2008-08-08 08:09:10') as dt
```

执行完毕我们就能在输出结果中看到下面的执行结果：

i	d	dt
-30	3	2008-08-08 08:09:10.0

下面的SQL语句用于将每个人的身份证后三位转换为整数类型并且进行相关的计算：

```
SELECT FIdNumber,
RIGHT(FIdNumber,3) as 后三位,
CAST(RIGHT(FIdNumber,3) AS INTEGER) as 后三位的整数形式,
CAST(RIGHT(FIdNumber,3) AS INTEGER)+1 as 后三位加1,
CONVERT(INTEGER,RIGHT(FIdNumber,3))/2 as 后三位除以2
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FIdNumber	后三位	后三位的整数形式	后三位加 1	后三位除以 2
123456789120	120	120	121	60
123456789121	121	121	122	60
123456789122	122	122	123	61
123456789123	123	123	124	61
123456789124	124	124	125	62
123456789125	125	125	126	62
123456789126	126	126	127	63
123456789127	127	127	128	63

5.4.1.3 Oracle

Oracle中也有一个名称为CONVERT()的函数，不过这个函数是用来进行字符集转换的。

Oracle中不支持用做数据类型转换的CAST()和CONVERT()两个函数，它提供了针对性更强的类型TO_CHAR()、TO_DATE()、TO_NUMBER()等函数，这些函数可以将数据显式的转换为字符串类型、日期时间类型或者数值类型。Oracle中还提供了HEXTORAW()、RAWTOHEX()、TO_MULTI_BYTE()、TO_SINGLE_BYTE()等函数用于存储格式的转换。下面我们将对这些函数进行分别介绍。

1) TO_CHAR()

TO_CHAR()函数用来将时间日期类型或者数值类型的数据转换为字符串，其参数格式如下：

TO_CHAR(expression,format)

参数expression为待转换的表达式，参数format为转换后的字符串格式，参数format可以省略，如果省略参数format将会按照数据库系统内置的转换规则进行转换。参数format的可以采用的格式非常丰富，具体可以参考Oracle的联机文档。

下面的SQL语句将出生日期和身高按照不同的格式转换为字符串类型：

```
SELECT FBirthDay,
TO_CHAR(FBirthDay,'YYYY-MM-DD') as c1,
FWeight,
TO_CHAR(FWeight,'L99D99MI') as c2,
TO_CHAR(FWeight) as c3
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	C1	FWEIGHT	C2	C3
1981-03-22 00:00:00.0	1981-03-22	56.67	￥56.67	56.67
1987-01-18 00:00:00.0	1987-01-18	36.17	￥36.17	36.17
1987-11-08 00:00:00.0	1987-11-08	40.33	￥40.33	40.33
1982-07-12 00:00:00.0	1982-07-12	46.23	￥46.23	46.23
1983-02-16 00:00:00.0	1983-02-16	48.68	￥48.68	48.68
1984-08-07 00:00:00.0	1984-08-07	66.67	￥66.67	66.67
1980-01-09 00:00:00.0	1980-01-09	51.28	￥51.28	51.28
1972-07-18 00:00:00.0	1972-07-18	60.32	￥60.32	60.32

2) TO_DATE()

TO_DATE()函数用来将字符串转换为时间类型，其参数格式如下：

TO_DATE (expression,format)

参数expression为待转换的表达式，参数format为转换格式，参数format可以省略，如果省略参数format将会按照数据库系统内置的转换规则进行转换。

下面的SQL语句用于将字符串形式的数据按照特定的格式解析为日期类型：

```
SELECT
```

```
TO_DATE('2008-08-08 08:09:10', 'YYYY-MM-DD HH24:MI:SS') as dt1,
TO_DATE('20080808 080910', 'YYYYMMDD HH24MISS') as dt2
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

DT1	DT2
2008-08-08 08:09:10.0	2008-08-08 08:09:10.0

3) TO_NUMBER()

TO_NUMBER()函数用来将字符串转换为数值类型，其参数格式如下：

```
TO_NUMBER (expression,format)
```

参数expression为待转换的表达式，参数format为转换格式，参数format可以省略，如果省略参数format将会按照数据库系统内置的转换规则进行转换。参数format的可以采用的格式非常丰富，具体可以参考Oracle的联机文档。

下面的SQL语句用于将字符串形式的数据按照特定的格式解析为数值类型：

```
SELECT
TO_NUMBER('33.33') as n1,
TO_NUMBER('100.00', '9G999D99') as n2
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

N1	N2
33.33	100.55

4) HEXTORAW()、RAWTOHEX()

HEXTORAW()用于将十六进制格式的数据转换为原始值，而RAWTOHEX()函数用来将原始值转换为十六进制格式的数据。例子如下：

```
SELECT HEXTORAW('7D'),
RAWTOHEX ('a'),
HEXTORAW(RAWTOHEX('w'))
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

HEXTORAW(7D)	RAWTOHEX(A)	HEXTORAW(RAWTOHEX(W))
}	61	w

5) TO_MULTI_BYTE()、TO_SINGLE_BYTE()

TO_MULTI_BYTE()函数用于将字符串中的半角字符转换为全角字符，而TO_SINGLE_BYTE()函数则用来将字符串中的全角字符转换为半角字符。例子如下：

```
SELECT
TO_MULTI_BYTE('moring'),
TO_SINGLE_BYTE('ｈｅｌｌｏ')
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

TO_MULTI_BYTE(MORING)	TO_SINGLE_BYTE(ＨＥＬＬＯ)
ｍｏｒｉｎｇ	hello

5.4.1.4 DB2

DB2中没有提供专门进行显式类型转换的函数，取而代之的是借用了很多高级语言中的强制类型转换的概念，也就是使用目标类型名做为函数名来进行类型转换，比如要将expr转换为日期类型，那么使用DATE(expr)即可。这种实现机制非常方便，降低了学习难度。

下面的SQL语句展示了DB2中类型转换的方式：

```
SELECT CHAR(FRegDay),
INT('33'),
DOUBLE('-3.1415926')
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1	2	3
1998-05-01	33	-3.1415926
1999-08-21	33	-3.1415926
2001-09-18	33	-3.1415926
2000-03-01	33	-3.1415926
1998-05-01	33	-3.1415926
1999-03-01	33	-3.1415926
2002-09-23	33	-3.1415926
1995-06-19	33	-3.1415926

5.4.2 空值处理

在数据库中经常需要对空值（NULL）做处理，比如“如果名称为空值则返回别名”，甚至还有更复杂的需求，比如“如果名称为空值则返回别名，如果别名也为空则返回‘佚名’两个字”、“如果名称为与别名相等则返回空值，否则返回名称”。这些需求已经带有流程控制的色彩了，一般来说需要在宿主语言中使用流程控制语句来进行处理，可是如果是在报表程序等大数据量的程序中把这些任务交给宿主语言的话会大大降低运行速度，因此我们必须想办法在SQL这一层进行处理。

为了更好的演示本节中的例子，我们需要对T_Person表中的数据进行一下修改，也就是将Kerry的出生日期修改为空值，将Smith的出生日期和注册日期都修改为空值，执行下面的SQL语句：

```
UPDATE T_Person SET FBirthDay=null WHERE FName='Kerry';
UPDATE T_Person SET FBirthDay=null AND FRegDay=null WHERE FName='Smith';
```

执行完毕我们查看T_Person表中中的数据如下：

FIDNUMBER	FNAME	FBIRTHDAY	FREGDAY	FWEIGHT
123456789120	Tom	1981-03-22	1998-05-01	56.67
123456789121	Jim	1987-01-18	1999-08-21	36.17
123456789122	Lily	1987-11-08	2001-09-18	40.33
123456789123	Kelly	1982-07-12	2000-03-01	46.23
123456789124	Sam	1983-02-16	1998-05-01	48.68
123456789125	Kerry	<NULL>	1999-03-01	66.67
123456789126	Smith	<NULL>	<NULL>	51.28
123456789127	BillGates	1972-07-18	1995-06-19	60.32

5.4.2.1 COALESCE()函数

主流数据库系统都支持COALESCE()函数，这个函数主要用来进行空值处理，其参数格式如下：

```
COALESCE ( expression,value1,value2……,valuen)
```

COALESCE()函数的第一个参数expression为待检测的表达式，而其后的参数个数不定。COALESCE()函数将会返回包括expression在内的所有参数中的第一个非空表达式。如果expression不为空值则返回expression；否则判断value1是否是空值，如果value1不为空值则返

回value1；否则判断value2是否是空值，如果value2不为空值则返回value3；……以此类推，如果所有的表达式都为空值，则返回NULL。

我们将使用COALESCE()函数完成下面的功能，返回人员的“重要日期”：如果出生日期不为空则将出生日期做为“重要日期”，如果出生日期为空则判断注册日期是否为空，如果注册日期不为空则将注册日期做为“重要日期”，如果注册日期也为空则将“2008年8月8日”做为“重要日期”。实现此功能的SQL语句如下：

MYSQL、MSSQLServer、DB2:

```
SELECT FName,FBirthDay,FRegDay,
COALESCE(FBirthDay,FRegDay,'2008-08-08')    AS ImportDay
FROM T_Person
```

Oracle:

```
SELECT FBirthDay,FRegDay,
COALESCE(FBirthDay,FRegDay,TO_DATE('2008-08-08', 'YYYY-MM-DD HH24:MI:SS'))    AS
ImportDay
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FBirthDay	FRegDay	ImportDay
Tom	1981-03-22 00:00:00	1998-05-01 00:00:00	1981-03-22 00:00:00
Jim	1987-01-18 00:00:00	1999-08-21 00:00:00	1987-01-18 00:00:00
Lily	1987-11-08 00:00:00	2001-09-18 00:00:00	1987-11-08 00:00:00
Kelly	1982-07-12 00:00:00	2000-03-01 00:00:00	1982-07-12 00:00:00
Sam	1983-02-16 00:00:00	1998-05-01 00:00:00	1983-02-16 00:00:00
Kerry	<NULL>	1999-03-01 00:00:00	1999-03-01 00:00:00
Smith	<NULL>	<NULL>	2008-08-08
BillGates	1972-07-18 00:00:00	1995-06-19 00:00:00	1972-07-18 00:00:00

这里边最关键的就是Kerry和Smith这两行，可以看到这里的计算逻辑是完全符合我们的需求的。

5.4.2.2 COALESCE()函数的简化版

COALESCE()函数可以用来完成几乎所有的空值处理，不过在很多数据库系统中都提供了它的简化版，这些简化版中只接受两个变量，其参数格式如下：

MYSQL:

```
IFNULL(expression,value)
```

MSSQLServer:

```
ISNULL(expression,value)
```

Oracle:

```
NVL(expression,value)
```

这几个函数的功能和COALESCE(expression,value)是等价的。比如SQL语句用于返回人员的“重要日期”，如果出生日期不为空则将出生日期做为“重要日期”，如果出生日期为空则返回NULL：

MYSQL:

```
SELECT FBirthDay,FRegDay,
IFNULL(FBirthDay,FRegDay)    AS ImportDay
FROM T_Person
```

MSSQLServer:

```
SELECT FBirthDay,FRegDay,
ISNULL(FBirthDay,FRegDay)    AS ImportDay
FROM T_Person
```

Oracle:

```
SELECT FBirthDay,FRegDay,
NVL(FBirthDay,FRegDay)    AS ImportDay
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	FREGDAY	IMPORTDAY
1981-03-22 00:00:00.0	1998-05-01 00:00:00.0	1981-03-22 00:00:00.0
1987-01-18 00:00:00.0	1999-08-21 00:00:00.0	1987-01-18 00:00:00.0
1987-11-08 00:00:00.0	2001-09-18 00:00:00.0	1987-11-08 00:00:00.0
1982-07-12 00:00:00.0	2000-03-01 00:00:00.0	1982-07-12 00:00:00.0
1983-02-16 00:00:00.0	1998-05-01 00:00:00.0	1983-02-16 00:00:00.0
<NULL>	1999-03-01 00:00:00.0	1999-03-01 00:00:00.0
<NULL>	<NULL>	<NULL>
1972-07-18 00:00:00.0	1995-06-19 00:00:00.0	1972-07-18 00:00:00.0

5.4.2.3 NULLIF()函数

主流数据库都支持NULLIF()函数，这个函数的参数格式如下：

```
NULLIF ( expression1 , expression2 )
```

如果两个表达式不等价，则 NULLIF 返回第一个 expression1的值。如果两个表达式等价，则 NULLIF 返回第一个 expression1类型的空值。也就是返回类型与第一个 expression 相同。

下面的SQL演示了NULLIF()函数的用法：

```
SELECT FBirthDay,FRegDay,
NULLIF(FBirthDay,FRegDay)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	FRegDay	
1981-03-22 00:00:00.0	1998-05-01 00:00:00.0	1981-03-22 00:00:00.0
1987-01-18 00:00:00.0	1999-08-21 00:00:00.0	1987-01-18 00:00:00.0
1987-11-08 00:00:00.0	2001-09-18 00:00:00.0	1987-11-08 00:00:00.0
1982-07-12 00:00:00.0	2000-03-01 00:00:00.0	1982-07-12 00:00:00.0
1983-02-16 00:00:00.0	1998-05-01 00:00:00.0	1983-02-16 00:00:00.0
<NULL>	1999-03-01 00:00:00.0	<NULL>
<NULL>	<NULL>	<NULL>
1972-07-18 00:00:00.0	1995-06-19 00:00:00.0	1972-07-18 00:00:00.0

5.4.3 CASE函数

COALESCE()函数只能用来进行空值的逻辑判断处理，如果要实现“如果年龄大于25则返回姓名，否则返回别名”这样的逻辑判断就比较麻烦了。在主流数据库系统中提供了CASE函数的支持，严格意义上来讲CASE函数已经是流程控制语句了，不是简单意义上的函数，不过为了方便，很多人都将CASE称作“流程控制函数”。

CASE函数有两种用法，下面分别介绍。

5.4.3.1 用法一

CASE函数的语法如下：

```
CASE expression
WHEN value1 THEN returnvalue1
WHEN value2 THEN returnvalue2
WHEN value3 THEN returnvalue3
……
ELSE defaultreturnvalue
END
```

CASE函数对表达式expression进行测试，如果expression等于value1则返回returnvalue1，如果expression等于value2则返回returnvalue2，expression等于value3则返回returnvalue3，……以此类推，如果不符合所有的WHEN条件，则返回默认值defaultreturnvalue。

可见CASE函数和普通编程语言中的SWITCH……CASE语句非常类似。使用CASE函数我们可以实现非常复杂的业务逻辑。下面的SQL用于判断谁是“好孩子”，我们比较偏爱Tom和Lily，所以我们将他们认为是好孩子，而我们比较不喜欢Sam和Kerry，所以认为他们是坏孩子，其他孩子则为普通孩子：

```
SELECT
    FName,
    (CASE FName
    WHEN 'Tom' THEN 'GoodBoy'
    WHEN 'Lily' THEN 'GoodGirl'
    WHEN 'Sam' THEN 'BadBoy'
    WHEN 'Kerry' THEN 'BadGirl'
    ELSE 'Normal'
    END) as isgood
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	ISGOOD
Tom	GoodBoy
Jim	Normal
Lily	GoodGirl
Kelly	Normal
Sam	BadBoy
Kerry	BadGirl
Smith	Normal
BillGates	Normal

CASE函数在制作报表的时候非常有用。比如表T_Customer中的FLevel字段是整数类型，它记录了客户的级别，如果为1则是VIP客户，如果为2则是高级客户，如果为3则是普通客户，在制作报表的时候显然不应该把1、2、3这样的数字显示到报表中，而应该显示相应的文字，这里就可以使用CASE函数进行处理，SQL语句如下：

```
SELECT
    FName,
    (CASE FLevel
    WHEN 1 THEN 'VIP客户'
    WHEN 2 THEN '高级客户'
```

```
    WHEN 3 THEN '普通客户'
    ELSE '客户类型错误'
    END) as FLevelName
FROM T_Customer
```

5.4.3.2 用法二

上边一节中介绍的CASE语句的用法只能用来实现简单的“等于”逻辑的判断，要实现“如果年龄小于18则返回‘未成年人’，否则返回‘成年人’”是无法完成的。值得庆幸的是，CASE函数还提供了第二种用法，其语法如下：

```
CASE
WHEN condition1 THEN returnvalue1
WHEN condition 2 THEN returnvalue2
WHEN condition 3 THEN returnvalue3
……
ELSE defaultreturnvalue
END
```

其中的condition1 、condition 2、condition 3……为条件表达式，CASE函数对各个表达式从前向后进行测试，如果条件condition1 为真则返回returnvalue1，否则如果条件condition2 为真则返回returnvalue2，否则如果条件condition3 为真则返回returnvalue3，……以此类推，如果不符合所有的WHEN条件，则返回默认值defaultreturnvalue。

这种用法中没有限制只能对一个表达式进行判断，因此使用起来更加灵活。比如下面的SQL语句用来判断一个人的体重是否正常，如果体重小于40则认为太瘦，而如果体重大于50则认为太胖，介于40和50之间则认为是正常：

```
SELECT
    FName,
    FWeight,
    (CASE
    WHEN FWeight<40 THEN 'thin'
    WHEN FWeight>50 THEN 'fat'
    ELSE 'ok'
    END) as isnormal
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FWEIGHT	ISNORMAL
Tom	56.67	fat
Jim	36.17	thin
Lily	40.33	ok
Kelly	46.23	ok
Sam	48.68	ok
Kerry	66.67	fat
Smith	51.28	fat
BillGates	60.32	fat

5.5 各数据库系统独有函数

本节内容试读版不提供。请购买《程序员的SQL金典》。

第十章 高级话题

本章将讨论一些数据库开发中的高级话题，包括 SQL 注入漏洞攻防、数据库调优、范式等。

10.4 自动增长字段

在设计数据库的时候，有时需要表的某个字段是自动增长的，最常使用自动增长字段的就是表的主键，使用自动增长字段可以简化主键的生成。不同的 DBMS 中自动增长字段的实现机制也有不同，下面分别介绍。

10.4.1 MYSQL 中的自动增长字段

MYSQL 中设定一个字段为自动增长字段非常简单，只要在表定义中指定字段为 AUTO_INCREMENT 即可。比如下面的 SQL 语句创建 T_Person 表，其中主键 FId 为自动增长字段：

```
CREATE TABLE T_Person
(
FId INT PRIMARY KEY AUTO_INCREMENT,
FName VARCHAR(20),
FAge INT
);
```

执行上面的 SQL 语句后就创建成功了 T_Person 表，然后执行下面的 SQL 语句向 T_Person 表中插入一些数据：

```
INSERT INTO T_Person(FName,FAge)
VALUES('Tom',18);

INSERT INTO T_Person(FName,FAge)
VALUES('Jim',81);

INSERT INTO T_Person(FName,FAge)
VALUES('Kerry',33);
```

注意这里的 INSERT 语句没有为 FId 字段设定任何值，因为 DBMS 会自动为 FId 字段设定值。执行完毕后查看 T_Person 表中的内容：

FId	FName	FAge
1	Tom	18
2	Jim	81
3	Kerry	33

可以看到 FId 中确实是自动增长的。

这个例子讲解完了，请删除 T_Person 表：

```
DROP TABLE T_Person;
```

10.4.2 MSSQLServer 中的自动增长字段

MSSQLServer 中设定一个字段为自动增长字段非只要在表定义中指定字段为 IDENTITY 即可，格式为 IDENTITY(startvalue,step)，其中的 startvalue 参数值为起始数字，step 参数值为步长，即每次自动增长时增加的值。

比如下面的 SQL 语句创建 T_Person 表，其中主键 FId 为自动增长字段，并且设定 100 为起始数字，步长为 3：

```
CREATE TABLE T_Person
(
FId INT PRIMARY KEY IDENTITY(100,3),
```

```
FName VARCHAR(20),
FAge INT
);
```

执行上面的 SQL 语句后就创建成功了 T_Person 表，然后执行下面的 SQL 语句向 T_Person 表中插入一些数据：

```
INSERT INTO T_Person(FName,FAge)
VALUES('Tom',18);

INSERT INTO T_Person(FName,FAge)
VALUES('Jim',81);

INSERT INTO T_Person(FName,FAge)
VALUES('Kerry',33);
```

注意这里的 INSERT 语句没有为 FId 字段设定任何值，因为 DBMS 会自动为 FId 字段设定值。执行完毕后查看 T_Person 表中的内容：

FId	FName	FAge
100	Tom	18
103	Jim	81
106	Kerry	33

可以看到 FId 中确实是 100 为起始数字、步长为 3 自动增长的。

这个例子讲解完了，请删除 T_Person 表：

```
DROP TABLE T_Person;
```

10.4.3 Oracle 中的自动增长字段

Oracle 中不像 MYSQL 和 MSSQLServer 中那样指定一个列为自动增长列的方式，不过在 Oracle 中可以通过 SEQUENCE 序列来实现自动增长字段。

在 Oracle 中 SEQUENCE 被称为序列，每次取的时候它会自动增加，一般用在需要按序列号排序的地方。

在使用 SEQUENCE 前需要首先定义一个 SEQUENCE，定义 SEQUENCE 的语法如下：

```
CREATE SEQUENCE sequence_name
INCREMENT BY step
START WITH startvalue;
```

其中 sequence_name 为序列的名字，每个序列都必须有唯一的名字；startvalue 参数值为起始数字，step 参数值为步长，即每次自动增长时增加的值。

一旦定义了 SEQUENCE，你就可以用 CURRVAL 来取得 SEQUENCE 的当前值，也可以通过 NEXTVAL 来增加 SEQUENCE，然后返回 新的 SEQUENCE 值。比如：

```
sequence_name.CURRVAL
sequence_name.NEXTVAL
```

如果 SEQUENCE 不需要的话就可以将其删除：

```
DROP SEQUENCE sequence_name;
```

下面举一个使用 SEQUENCE 序列实现自动增长的例子。

首先创建一个名称为 seq_PersonId 的 SEQUENCE：

```
CREATE SEQUENCE seq_PersonId
INCREMENT BY 1
START WITH 1;
```

然后创建 T_Person 表：

```
CREATE TABLE T_Person
(
FId NUMBER (10) PRIMARY KEY,
FName VARCHAR2(20),
FAge NUMBER (10)
);
```

执行上面的 SQL 语句后就创建成功了 T_Person 表，然后执行下面的 SQL 语句向 T_Person 表中插入一些数据：

```
INSERT INTO T_Person(FId,FName,FAge)
VALUES(seq_PersonId.NEXTVAL,'Tom',18);

INSERT INTO T_Person(FId,FName,FAge)
VALUES(seq_PersonId.NEXTVAL,'Jim',81);

INSERT INTO T_Person(FId,FName,FAge)
VALUES(seq_PersonId.NEXTVAL,'Kerry',33);
```

注意这里的 INSERT 语句没有为 FId 字段设定任何值，因为 DBMS 会自动为 FId 字段设定值。执行完毕后查看 T_Person 表中的内容：

FID	FNAME	FAGE
1	Tom	18
2	Jim	81
3	Kerry	33

使用 SEQUENCE 实现自动增长字段的缺点是每次向表中插入记录的时候都要显式的到 SEQUENCE 中取得新的字段值，如果忘记了就会造成错误。为了解决这个问题，我们可以使用触发器来解决，创建一个 T_Person 表上的触发器：

```
CREATE OR REPLACE TRIGGER trigger_personIdAutoInc
   BEFORE INSERT ON T_Person
   FOR EACH ROW
DECLARE
BEGIN
   SELECT seq_PersonId.NEXTVAL INTO:NEW.FID FROM DUAL;
END trigger_personIdAutoInc;
```

这个触发器在 T_Person 中插入新记录之前触发，当触发器被触发后则从 seq_PersonId 中取道新的序列号然后设置给 FID 字段。

执行下面的 SQL 语句向 T_Person 表中插入一些数据：

```
INSERT INTO T_Person(FAge)
VALUES('Wow',22);

INSERT INTO T_Person(FName,FAge)
VALUES('Herry',28);

INSERT INTO T_Person(FName,FAge)
VALUES('Gavin',36);
```

注意在这个 SQL 语句中无需再为 FId 字段赋值。执行完毕后查看 T_Person 表中的内容：

FID	FNAME	FAGE
1	Tom	18
2	Jim	81
3	Kerry	33
4	Wow	22
5	Herry	28
7	Gavin	36

这个例子讲解完了，请删除 T_Person 表以及 SEQUENCE：

```
DROP TABLE T_Person;
DROP SEQUENCE seq_PersonId;
```

10.4.4 DB2 中的自动增长字段

DB2 中实现自动增长字段有两种方式：定义带有 IDENTITY 属性的列；使用 SEQUENCE 对象。

10.4.4.1 定义带有 IDENTITY 属性的列

首先创建 T_Person 表，SQL 语句如下：

```
CREATE TABLE T_Person
(
FId INT PRIMARY KEY NOT NULL
GENERATED ALWAYS
     AS IDENTITY
     (START WITH 1
      INCREMENT BY 1
     ),
FName VARCHAR(20),
FAge INT
);
```

执行上面的 SQL 语句后就创建成功了 T_Person 表，然后执行下面的 SQL 语句向 T_Person 表中插入一些数据：

```
INSERT INTO T_Person(FName,FAge)
VALUES('Tom',18);

INSERT INTO T_Person(FName,FAge)
VALUES('Jim',81);

INSERT INTO T_Person(FName,FAge)
VALUES('Kerry',33);
```

注意这里的 INSERT 语句没有为 FId 字段设定任何值，因为 DBMS 会自动为 FId 字段设定值。执行完毕后查看 T_Person 表中的内容：

FId	FName	FAge
100	Tom	18
103	Jim	81
106	Kerry	33

这个例子讲解完了，请删除 T_Person 表：

```
DROP TABLE T_Person;
```

10.4.4.2 使用 SEQUENCE 对象

DB2 中的 SEQUENCE 和 Oracle 中的 SEQUENCE 相同，只是定义方式和使用方式略有不同。

下面创建了一个 SEQUENCE：

```
CREATE SEQUENCE seq_PersonId AS INT
INCREMENT BY 1
START WITH 1;
```

使用SEQUENCE的方式如下：

```
NEXT VALUE FOR sequence_name
```

这样就可以通过下面的SQL语句来使用SEQUENCE：

```
INSERT INTO T_Person(FId,FName,FAge)
VALUES(NEXT VALUE FOR seq_PersonId,'Kerry',33);
```

如果想在向表中插入记录的时候自动设定 FId 字段的值则同样要使用触发器，具体请参考相关资料，这里不再赘述。

这个例子讲解完了，请删除 seq_PersonId 序列：

```
DROP SEQUENCE seq_PersonId;
```

10.5 业务主键与逻辑主键

本节内容试读版不提供。请购买《程序员的 SQL 金典》。

10.6 NULL 的学问

本节内容试读版不提供。请购买《程序员的 SQL 金典》。

10.7 开窗函数

在开窗函数出现之前存在着很多用 SQL 语句很难解决的问题，很多都要通过复杂的相关子查询或者存储过程来完成。为了解决这些问题，在 2003 年 ISO SQL 标准加入了开窗函数，开窗函数的使用使得这些经典的难题可以被轻松的解决。目前在 MSSQLServer、Oracle、DB2 等主流数据库中都提供了对开窗函数的支持，不过非常遗憾的是 MYSQL 暂时还未对开窗函数给予支持，因此本节中的例子将无法在 MYSQL 中运行通过。

为了更加清晰的讲解开窗函数我们将创建一张表，执行下面的 SQL 语句：

MYSQL,MSSQLServer,DB2:

```
CREATE TABLE T_Person (FName VARCHAR(20),FCity VARCHAR(20),
FAge INT,FSalary INT)
```

Oracle:

```
CREATE TABLE T_Person (FName VARCHAR2(20),FCity VARCHAR2(20),
FAge INT,FSalary INT)
```

T_Person 表保存了人员信息，FName 字段为人员姓名，FCity 字段为人员所在的城市名，FAge 字段为人员年龄，FSalary 字段为人员工资。请在相应的 DBMS 中执行相应的 SQL 语句，然后执行下面的 SQL 语句向 T_Person 表中插入一些演示数据：

```
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Tom','BeiJing',20,3000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Tim','ChengDu',21,4000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Jim','BeiJing',22,3500);
```

```
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Lily','London',21,2000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('John','NewYork',22,1000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('YaoMing','BeiJing',20,3000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Swing','London',22,2000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Guo','NewYork',20,2800);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('YuQian','BeiJing',24,8000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Ketty','London',25,8500);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Kitty','ChengDu',25,3000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Merry','BeiJing',23,3500);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Smith','ChengDu',30,3000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Bill','BeiJing',25,2000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Jerry','NewYork',24,3300);
```

执行完毕查看 T_Person 表中的内容：

FNAME	FCITY	FAGE	FSALARY
Tom	BeiJing	20	3000
Tim	ChengDu	21	4000
Jim	BeiJing	22	3500
Lily	London	21	2000
John	NewYork	22	1000
YaoMing	BeiJing	20	3000
Swing	London	22	2000
Guo	NewYork	20	2800
YuQian	BeiJing	24	8000
Ketty	London 2	5 8	500
Kitty C	hengDu 2	5 3	000
Merry	BeiJing	23	3500
Smith	ChengDu	30	3000
Bill	BeiJing	25	2000
Jerry	NewYork	24	3300

10.7.1 开窗函数简介

与聚合函数一样，开窗函数也是对行集组进行聚合计算，但是它不像普通聚合函数那样每组只返回一个值，开窗函数可以为每组返回多个值，因为开窗函数所执行聚合计

算的行集组是窗口。在 ISO SQL 规定了这样的函数为开窗函数，在 Oracle 中则被称为分析函数，而在 DB2 中则被称为 OLAP 函数。

要计算所有人员的总数，我们可以执行下面的 SQL 语句：

```
SELECT COUNT(*) FROM T_Person
```

除了这种较简单的使用方式，有时需要从不在聚合函数中的行中访问这些聚合计算的值。比如我们想查询每个工资小于 5000 元的员工信息（城市以及年龄），并且在每行中都显示所有工资小于 5000 元的员工个数，尝试编写下面的 SQL 语句：

```
SELECT FCITY , FAGE , COUNT(*)
FROM T_Person
WHERE FSALARY<5000
```

执行上面的 SQL 以后我们会得到下面的错误信息：

```
选择列表中的列 'T_Person.FCity' 无效，因为该列没有包含在聚合函数或 GROUP BY 子句中。
```

这是因为所有不包含在聚合函数中的列必须声明在 GROUP BY 子句中，可以进行如下修改：

```
SELECT FCITY, FAGE, COUNT(*)
FROM T_Person
WHERE FSALARY<5000
GROUP BY FCITY , FAGE
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FCITY	FAGE	
BeiJing	20	2
NewYork	20	1
ChengDu	21	1
London	21	1
BeiJing	22	1
London	22	1
NewYork	22	1
BeiJing	23	1
NewYork	24	1
BeiJing	25	1
ChengDu	25	1
ChengDu	30	1

这个执行结果与我们想像的是完全不同的，这是因为 **GROUP BY 子句对结果集进行了分组，所以聚合函数进行计算的对象不再是所有的结果集，而是每一个分组。**

可以通过子查询来解决这个问题，SQL 如下：

```
SELECT FCITY , FAGE ,
(
    SELECT COUNT(* ) FROM T_Person
    WHERE FSALARY<5000
)
FROM T_Person
WHERE FSALARY<5000
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FCITY	FAGE	
BeiJing	20	13

ChengDu	21	13
BeiJing	22	13
London	21	13
NewYork	22	13
BeiJing	20	13
London	22	13
NewYork	20	13
ChengDu	25	13
BeiJing	23	13
ChengDu	30	13
BeiJing	25	13
NewYork	24	13

虽然使用子查询能够解决这个问题，但是子查询的使用非常麻烦，使用开窗函数则可以大大简化实现，下面的 SQL 语句展示了如果使用开窗函数来实现同样的效果：

```
SELECT FCITY , FAGE , COUNT(*) OVER()
FROM T_Person
WHERE FSALARY<5000
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FCITY	FAGE	
BeiJing	20	13
ChengDu	21	13
BeiJing	22	13
London	21	13
NewYork	22	13
BeiJing	20	13
London	22	13
NewYork	20	13
ChengDu	25	13
BeiJing	23	13
ChengDu	30	13
BeiJing	25	13

可以看到与聚合函数不同的是，开窗函数在聚合函数后增加了一个 OVER 关键字。开窗函数的调用格式为：

函数名(列) OVER(选项)

OVER 关键字表示把函数当成开窗函数而不是聚合函数。SQL 标准允许将所有聚合函数用做开窗函数，使用 OVER 关键字来区分这两种用法。

在上边的例子中，开窗函数 **COUNT(*) OVER()** 对于查询结果的每一行都返回所有符合条件的行的条数。OVER 关键字后的括号中还经常添加选项用以改变进行聚合运算的窗口范围。如果 OVER 关键字后的括号中的选项为空，则开窗函数会对结果集中的所有行进行聚合运算。

10.7.2 PARTITION BY 子句

开窗函数的 OVER 关键字后括号中的可以使用 PARTITION BY 子句来定义行的分区来供进行聚合计算。与 GROUP BY 子句不同，PARTITION BY 子句创建的分区是独立于结果

集的，创建的分区只是供进行聚合计算的，而且不同的开窗函数所创建的分区也不互相影响。下面的 SQL 语句用于显示每一个人员的信息以及所属城市的人员数：

```
SELECT FName,FCITY , FAGE , FSalary,
COUNT(*) OVER(PARTITION BY FCITY)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FCITY	FAGE	FSalary	
Tom	BeiJing	20	3000	6
Jim	BeiJing	22	3500	6
YaoMing	BeiJing	20	3000	6
YuQian	BeiJing	24	8000	6
Merry	BeiJing	23	3500	6
Bill	BeiJing	25	2000	6
Smith	ChengDu	30	3000	3
Kitty C	hengDu 2	5 3	000 3	
Tim	ChengDu	21	4000	3
Lily	London	21	2000	3
Ketty L	ondon 2	5 8	500 3	
Swing	London	22	2000	3
Guo	NewYork	20	2800	3
John	NewYork	22	1000	3
Jerry	NewYork	24	3300	3

`COUNT(*) OVER(PARTITION BY FCITY)`表示对结果集按照FCITY进行分区，并且计算当前行所属的组的聚合计算结果。比如对于FName等于Tom的行，它所属的城市是BeiJing，同属于BeiJing的人员一共有6个，所以对于这一列的显示结果为6。

在同一个SELECT语句中可以同时使用多个开窗函数，而且这些开窗函数并不会相互干扰。比如下面的SQL语句用于显示每一个人员的信息、所属城市的人员数以及同龄人的人数：

```
SELECT FName,FCITY, FAGE, FSalary,
COUNT(*) OVER(PARTITION BY FCITY),
COUNT(*) OVER(PARTITION BY FAGE)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FCITY	FAGE	FSalary		
Tom	BeiJing	20	3000	6	3
YaoMing	BeiJing	20	3000	6	3
Guo	NewYork	20	2800	3	3
Tim	ChengDu	21	4000	3	2
Lily	London	21	2000	3	2
Jim	BeiJing	22	3500	6	3
John	NewYork	22	1000	3	3
Swing	London	22	2000	3	3
Merry	BeiJing	23	3500	6	1
YuQian	BeiJing	24	8000	6	2

Jerry	NewYork	24	3300	3	2
Kitty C	hengDu 2	5 3	000 3	3	
Bill	BeiJing	25	2000	6	3
Ketty L	ondon 2	5 8	500 3	3	
Smith	ChengDu	30	3000	3	1

在这个查询结果中，可以看到同一城市中的**COUNT(*) OVER(PARTITION BY** FCITY) 计算结果相同，而且同龄人中的**COUNT(*) OVER(PARTITION BY** FAGE) 计算结果也相同。

10.7.2 ORDER BY子句

MSSQLServer中是不支持开窗函数中的ORDER BY子句的，因此本节演示的内容只适用于Oracle和DB2。开窗函数中可以在OVER关键字后的选项中使用ORDER BY子句来指定排序规则，而且有的开窗函数还要求必须指定排序规则。使用ORDER BY子句可以对结果集按照指定的排序规则进行排序，并且在一个指定的范围内进行聚合运算。ORDER BY子句的语法为：

ORDER BY 字段名 RANGE|ROWS BETWEEN 边界规则1 AND 边界规则2

RANGE表示按照值的范围进行范围的定义，而ROWS表示按照行的范围进行范围的定义；边界规则的可取值见下表：

可取值	说明	示例
CURRENT ROW	当前行	
N PRECEDING	前N行	2 PRECEDING
UNBOUNDED PRECEDING	一直到第一条记录	
N FOLLOWING	后N行	2 FOLLOWING
UNBOUNDED FOLLOWING	一直到最后一条记录	

“RANGE|ROWS BETWEEN 边界规则1 AND 边界规则2”部分用来定位聚合计算范围，这个子句又被称为定位框架。下面通过例子来展示ORDER BY子句的用法。

例1

```
SELECT FName, FSalary,
SUM(FSalary) OVER(ORDER BY FSalary ROWS BETWEEN UNBOUNDED PRECEDING
    AND CURRENT ROW)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	3
John	1000	1000
Lily	2000	3000
Swing	2000	5000
Bill	2000	7000
Guo	2800	9800
Tom	3000	12800
YaoMing	3000	15800
Kitty 3	000 1	8800
Smith	3000	21800
Jerry	3300	25100
Jim	3500	28600
Merry	3500	32100
Tim	4000	36100

YuQian	8000	44100
Ketty 8	500 5	2600

这里的开窗函数“**SUM(FSalary) OVER(ORDER BY FSalary ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)**”表示按照FSalary进行排序，然后计算从第一行（**UNBOUNDED PRECEDING**）到当前行（**CURRENT ROW**）的和，这样的计算结果就是按照工资进行排序的工资值的累积和。

例2

```
SELECT FName, FSalary,
SUM(FSalary) OVER(ORDER BY FSalary RANGE BETWEEN UNBOUNDED PRECEDING
    AND CURRENT ROW)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	3
John	1000	1000
Lily	2000	7000
Swing	2000	7000
Bill	2000	7000
Guo	2800	9800
Tom	3000	21800
YaoMing	3000	21800
Kitty 3	000 2	1800
Smith	3000	21800
Jerry	3300	25100
Jim	3500	32100
Merry	3500	32100
Tim	4000	36100
YuQian	8000	44100
Ketty 8	500 5	2600

这个SQL语句与例1中的SQL语句唯一不同的就是“**ROWS**”被替换成了“RANGE”。“**ROWS**”是按照行数进行范围定位的，而“RANGE”则是按照值范围进行定位的，这两个不同的定位方式主要用来处理并列排序的情况。比如Lily、Swing、Bill这三个人的工资都是2000元，如果按照“**ROWS**”进行范围定位，则计算从第一条到当前行的累积和，而如果如果按照“RANGE”进行范围定位，则仍然计算从第一条到当前行的累积和，不过由于等于2000元的工资有三个人，所以计算的累积和为从第一条到2000元工资的人员结，所以对Lily、Swing、Bill这三个人进行开窗函数聚合计算的时候得到的都是7000（“1000+2000+2000+2000”）。

例3

```
SELECT FName, FSalary,
SUM(FSalary) OVER(ORDER BY FSalary ROWS BETWEEN 2 PRECEDING AND 2
FOLLOWING)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	3
John	1000	5000

Lily	2000	7000
Swing	2000	9800
Bill	2000	11800
Guo	2800	12800
Tom	3000	13800
YaoMing	3000	14800
Kitty 3	000 1	5300
Smith	3000	15800
Jerry	3300	16300
Jim	3500	17300
Merry	3500	22300
Tim	4000	27500
YuQian	8000	24000
Ketty 8	500 2	0500

这里的开窗函数“SUM(FSalary) OVER(ORDER BY FSalary ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING)”表示按照FSalary进行排序，然后计算从当前行前两行（2 PRECEDING）到当前行后两行（2 FOLLOWING）的工资和，注意对于第一条和第二条而言它们的“前两行”是不存在或者不完整的，因此计算的时候也是要按照前两行是不存在或者不完整进行计算，同样对于最后两行数据而言它们的“后两行”也不存在或者不完整的，同样要进行类似的处理。

例4

```
SELECT FName, FSalary,
SUM(FSalary) OVER(ORDER BY FSalary ROWS BETWEEN 1 FOLLOWING AND 3
FOLLOWING)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	3
John	1000	6000
Lily	2000	6800
Swing	2000	7800
Bill	2000	8800
Guo	2800	9000
Tom	3000	9000
YaoMing	3000	9300
Kitty 3	000 9	800
Smith	3000	10300
Jerry	3300	11000
Jim	3500	15500
Merry	3500	20500
Tim	4000	16500
YuQian	8000	8500
Ketty 8	500 <	NULL>

这里的开窗函数“SUM(FSalary) OVER(ORDER BY FSalary ROWS BETWEEN 1

FOLLOWING AND 3 FOLLOWING)”表示按照FSalary进行排序，然后计算从当前行后一行（1 FOLLOWING）到后三行（3 FOLLOWING）的工资和。注意最后一行没有后续行，其计算结果为空值NULL而非0。

例5

```
SELECT FName, FSalary,
SUM(FSalary) OVER(ORDER BY FName RANGE BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	3
Bill	2000	2000
Guo	2800	4800
Jerry	3300	8100
Jim	3500	11600
John	1000	12600
Ketty 8	500 2	1100
Kitty 3	000 2	4100
Lily	2000	26100
Merry	3500	29600
Smith	3000	32600
Swing	2000	34600
Tim	4000	38600
Tom	3000	41600
YaoMing	3000	44600
YuQian	8000	52600

这里的开窗函数“SUM(FSalary) OVER(ORDER BY FName RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)”表示按照FName进行排序，然后计算从第一行（UNBOUNDED PRECEDING）到当前行（CURRENT ROW）的工资和。

“RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW”是开窗函数中最常使用的定位框架，为了简化使用，如果使用的是这种定位框架，则可以省略定位框架声明部分，也就是说“SUM(FSalary) OVER(ORDER BY FName RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)”等价于“SUM(FSalary) OVER(ORDER BY FName)”，所以这个SQL语句可以简写为：

```
SELECT FName, FSalary,
SUM(FSalary) OVER(ORDER BY FName)
FROM T_Person;
```

例6

```
SELECT FName, FSalary,
COUNT(*) OVER(ORDER BY FSalary ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	3
John	1000	1

Lily	2000	2
Swing	2000	3
Bill	2000	4
Guo	2800	5
Tom	3000	6
YaoMing	3000	7
Kitty 3	000 8	
Smith	3000	9
Jerry	3300	10
Jim	3500	11
Merry	3500	12
Tim	4000	13
YuQian	8000	14
Ketty 8	500 1	5

这里的开窗函数“COUNT(*) OVER(ORDER BY FSalary RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)”表示按照FSalary进行排序，然后计算从第一行（UNBOUNDED PRECEDING）到当前行（CURRENT ROW）的人员的个数，这个可以看作是计算人员的工资水平排名。

例7

```
SELECT FName, FSalary,FAge,
MAX(FSalary) OVER(ORDER BY FAge)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	FAGE	4
Tom	3000	20	3000
YaoMing	3000	20	3000
Guo	2800	20	3000
Tim	4000	21	4000
Lily	2000	21	4000
Jim	3500	22	4000
John	1000	22	4000
Swing	2000	22	4000
Merry	3500	23	4000
YuQian	8000	24	8000
Jerry	3300	24	8000
Ketty 8	500 2	5 8	500
Kitty 3	000 2	5 8	500
Bill	2000	25	8500
Smith	3000	30	8500

这里的开窗函数“MAX(FSalary) OVER(ORDER BY FAge)”是“MAX(FSalary) OVER(ORDER BY FAge RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)”的简化写法，它表示按照FSalary进行排序，然后计算从第一行（UNBOUNDED PRECEDING）到当前行（CURRENT ROW）的人员的最大工资值。

例8

```
SELECT FName, FSalary,FAge,
MAX(FSalary) OVER(PARTITION BY FAge ORDER BY FSalary)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	FAGE	4
Guo	2800	20	2800
Tom	3000	20	3000
YaoMing	3000	20	3000
Lily	2000	21	2000
Tim	4000	21	4000
John	1000	22	1000
Swing	2000	22	2000
Jim	3500	22	3500
Merry	3500	23	3500
Jerry	3300	24	3300
YuQian	8000	24	8000
Bill	2000	25	2000
Kitty 3	000 2	5 3	000
Ketty 8	500 2	5 8	500
Smith	3000	30	3000

从这个例子可以看出PARTITION BY子句和**ORDER BY可以共同使用，从而可以实现更加复杂的功能。**

10.7.3 高级开窗函数

本节内容试读版不提供。请购买《程序员的SQL金典》。

10.8 WITH子句与子查询

子查询可以简化SQL语句的编写，不过如果使用不当的话子查询会降低系统性能，为了避免子查询带来的性能问题，除了需要优化SQL语句之外还需要尽量降低使用子查询的次数。比如下面的子查询用来取得系统中所有年龄或者工资与Tom相同的人员：

```
SELECT * FROM T_Person
WHERE FAge=(SELECT FAge FROM T_Person WHERE FName='TOM')
OR FSalary=(SELECT FSalary FROM T_Person WHERE FName='TOM')
```

这个SQL语句可以完成要求的功能，不过可以看到类似的子查询被用到了两次，这会带来下面的问题：

- 同一个子查询被使用多次会造成这个子查询被执行多次，由于子查询是比较消耗系统资源的操作，所以这会降低系统的性能。

- 同一个子查询在多处被使用，这违反了编程中的DRY（Don't Repeat Yourself）原则，如果要修改子查询就必须对这些子查询同时修改，很容易造成修改不同步。

造成这种问题的原因就是子查询只能在定义的时候使用，这样如果多次使用就必须多次定义，为了解决这种问题，SQL提供了WITH子句用于为子查询定义一个别名，这样就可以通过这个别名来引用这个子查询了，也就是实现“一次定义多次使用”。

使用WITH子句来改造上面的SQL语句：

```
WITH person_tom AS
(
   SELECT * FROM T_Person
   WHERE FName='TOM'
)
SELECT * FROM T_Person
WHERE FAge=person_tom.FAge
OR FSalary=person_tom.FSalary
```

可以看到WITH子句的格式为：

```
WITH 别名 AS

(子查询)
```

定义好别名以后就可以在SQL语句中通过这个别名来引用子查询了，注意这个语句是一个SQL语句，而非存储过程，所以可以远程调用。

还可以在WITH语句中为子查询中的列定义别名，定义的方式就是在子查询别名后列出参数名列表。如下：

```
WITH person_tom(F1,F2,F3) AS
(
   SELECT FAge,FName,FSalary FROM T_Person
   WHERE FName='TOM'
)
SELECT * FROM T_Person
WHERE FAge=person_tom.F1
OR FSalary=person_tom.F3
```

Chinapub 在线购买地址：http://www.china-pub.com/301651

当当网在线购买地址：http://product.dangdang.com/product.aspx?product_id=20368319

第一本专门为程序员编写的数据库图书《程序员的 SQL 金典》

- 将子查询、表连接、数据库语法差异等用通俗易懂、诙谐幽默的语言讲解出来
- 配合大量真实案例，学了就能用，在短时间内成为数据库开发高手
- 高度提取不同数据库的共同点，仔细分析不同点，并给出

解决方案，同时学会 MSSQLServer、MYSQL、Oracle、DB2 数据库不再是梦

- 国内第一本讲解开窗函数实际应用的图书