基于Java的数据库连接池技术*

苏水根，王平，焦方源
（重庆邮电学院 电子商务研发中心，重庆 400065）

摘　要：介绍了Java的数据库访问机制，对实际应用中出现的问题进行了分析，提出了能改善数据库应用程序性能的连接池技术，深入分析了连接池的管理策略，并构造了一个连接池实例。

关键词：Java；JDBC；数据库；连接池
中图分类号：TP391　　**文献标识码**：A　　**文章编号**：1004-5694(2003)02-0050-04

Database Connection-Pool Technology Based on Java

SU Shui-gen, WANG Ping, JIAO Fang-yuan
(*E-Commerce Research & Development Center, CUPT, Chongqing* 400065, *P. R. China*)

Abstract: This article describes the Java-based database access mechanism and analyzes the problems in applications, and then puts forward the database connection-pool technology that can improve the performance of applications, and analyzes the management strategy of connection-pool and constructs an instance of connection-pool.

Key words: Java, JDBC, database, connection-pool

0 引　言

Java语言具有坚固、安全、易于使用、易于理解和可从网络上自动下载等特性，是编写数据库应用程序的杰出语言。在使用Java语言进行数据库应用开发中，一般都使用JDBC(Java database connectivity)来进行和数据库的交互，其中一个关键的概念就是连接，它在Java中是一类，代表了一个通道。通过它，使用数据的应用就可以从数据库访问数据了。对于一般的数据库应用来说，访问数据库不是很频繁，可以在需要访问数据库时新建一个连接，用完后直接关闭它，这不会对系统性能有太大的影响。但对于一个复杂的数据库应用来说，频繁地建立、关闭连接会极大地降低系统性能。本文讨论的Java中的数据库连接池技术可以有效地解决这个问题。

1 基于Java的数据库访问机制

Java语言的跨平台、可移植以及安全等特性使其成为开发数据库应用的一种优秀语言。JDBC是Java应用程序与数据库的沟通桥梁。JDBC主要提供2种API，分别是面向开发人员的JDBC API和面向底层的JDBC驱动程序管理器API，底层的驱动程序主要有2种类型：JDBC-ODBC桥驱动和直接的JDBC驱动。上层JDBC API和JDBC驱动程序管理器API通信，向它发送各种不同的SQL语句；管理器(对程序员是透明的)和各种不同的第三方驱动程序通信，由它们负责连接数据库，返回查询信息或执行查询语句指定的动作。图1说明了JDBC和数据库的通信路径。通过JDBC提供的API，应用程序能完成与数据库的连接和通信。

* 收稿日期：2002-06-21
作者简介：苏水根(1978-)，男，江西抚州人，硕士研究生。研究方向为数据库，电子商务；王平，博士，教授。

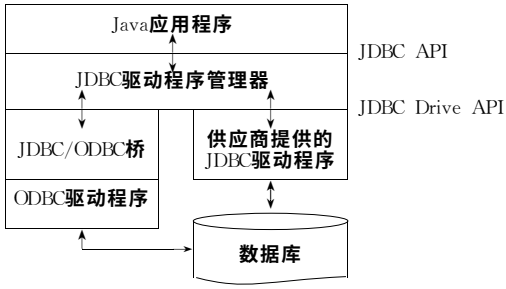

图1 JDBC和数据库的通信路径

Fig. 1 JDBC and Communication Path of the Database

2 连接池的工作原理

2.1 问题分析及解决方案

应用程序与数据库的交互过程中,通过 JDBC 与数据库建立连接是一项相当消耗系统资源的连接过程。在系统中简单地利用 JDBC 提供的 API 频繁地与数据库建立和关闭连接,会导致系统整体性能严重下降,因为连接频繁使用成了系统性能的瓶颈。试考虑一个需几百人同时访问数据库的应用,如果按传统的方法,数据库服务器必须建立大量的连接,这样必然会使服务器性能和效率极其低下,甚至还可能导致系统的崩溃。

问题的根源在于对连接资源的低效管理,并给出一个有效的解决方法:通过建立一个数据库连接池以及一套高效的连接使用管理策略使得一个数据库连接可以得到高效安全的复用,避免了数据库连接的频繁建立和关闭的开销,提高了系统的效率。

2.2 工作机制

连接池是众多连接对象的"缓冲存储池",连接池提供一种管理机制来控制连接池内部连接对象的个数,提供应用程序使用的获取、释放连接接口。连接池主要由 3 部分组成:连接池的建立、连接池中连接的使用管理、连接池的关闭。下面就着重讨论这 3 部分及连接池的配置问题。

(1) 连接池的建立。应用程序中要建立的是一个静态连接池,所谓静态是指连接池中的连接在系统初始化时就已分配好,且不能随意关闭连接。连接池建立时,根据配置,连接池从数据库中一次性获取预设数目的连接对象。这些连接对象作为系统可分配的自由连接,以后所使用的连接都从连接池中获取,这样可避免随意建立、释放连接所带来的开销。

(2) 连接池管理策略。连接池管理策略是连接池机制的核心。当客户需访问数据库时,不是直接同数据库建立连接,而是向连接池申请一个连接。同样,当客户访问数据库完毕,释放连接时,并不是直接关闭连接,而是向连接池释放连接。下面介绍连接池中连接的分配、释放策略:当客户应用连接池请求数据库连接时,先查看池中是否有没有被分配的空闲连接,如果存在,则把空闲连接分配给客户,并作相应处理;如果池中没有空闲连接,则等待直到有空闲连接分配给客户,此时该连接被多个用户复用。当客户释放连接时,唤醒所有等待连接的客户线程并做相应的处理。如果连接释放后没有等待连接的客户线程,则把它重新放回连接池中,并不关闭连接。由此可知:连接池能保证数据库连接的有效复用,避免频繁地建立、释放连接所带来的系统资源开销。

(3) 连接池的关闭。当应用程序退出时,应关闭连接池,此时应把在连接池建立时向数据库申请的连接对象统一归还给数据库(即关闭所有数据库连接),这与连接池的建立正好是一个相反过程。

(4) 连接池的配置。数据库连接池中到底要放置多少个连接,连接耗尽后该如何处理呢?这是一个配置策略。这里的配置策略是:根据具体的应用需求,给出一个初始的连接池连接数目以及一个连接池可以扩张到的最大连接数目。

2.3 构建连接池

以下用 Java 构建了一个连接池,并实现了连接池的基本功能。

```
import java.sql.*;
import java.util.*;
class ConnectionPool
{   private Vector freeConnections = new Vector();
    private int minConn = 0; //最小连接数,默认为0
    private int curConn = 0; //已使用的连接数
    private int maxConn; //连接池中的最大连接数
    private String drive; //数据库驱动程序类
    private String URL; //数据库的JDBC URL
    private String user; //数据库登录用户
    private String password; //数据库用户登录口令
    /** 初始化连接池
     * @param URL 数据库的JDBC URL
     * @param user 数据库登录用户
```

```
    * @param password 数据库用户登录口令
    * @param minConn 此连接池中最小的连接数
    * @param maxConn 此连接池允许建立的最大连接数
    */
    public ConnectionPool (String drive, String URL,
String user, String password,int minConn, int maxConn)
    {   this.drive = drive;
        this.URL = URL;
        this.user = user;
        this.password = password;
        this.minConn = minConn;
        this.maxConn = maxConn;
        iniConnection(); //初始化连接池中的初始连接
    }
    /** 初始化连接池,如未指定最小连接数,默认为0
    * 参见前一方法 */
    public ConnectionPool (String drive, String URL,
String user, String password, int maxConn)
    {   this(drive,URL,user,password,maxConn,0);
    }
    /** 在连接池中建立初始连接,放入空闲连接矢量中
    */
    private void iniConnection()
    {   Class.forName(drive);
        Connection con = null;
        for ( int i=0; i<minConn; )
        {   try
            {   if (user == null)
                {   con = DriverManager.getConnection
(URL);
                }
                else
                {   con= DriverManager.getConnection
(URL,user,password);
                }
            }
            catch (SQLException e)
            {   con = null;
            }
            if (con == null)
            {   continue;
            }
            else
            {   freeConnections.addElement(con);
                i++;
            }
        }
    }
    /**
    * 从连接池获得一个可用连接。如没有空闲的连接且
    * 当前连接数小于最大接数限制,则创建新连接。如原
    * 来登记为可用的连接不再有效,则从向量删除之,然
    * 后递归调用以试新的可用连接,如当前连接数已
    * 达到最大连接数则返回空
    */
    public synchronized Connection getConnection (long
timeout)
    {   Connection con = null;
        if (freeConnections.size() > 0)
        {   // 获取向量中第一个可用连接
            con = (Connection) freeConnections.firstEle-
ment();
            freeConnections.removeElementAt(0);
            try
            {   if (con.isClosed())
                {   // 递归调用自己,试再次获取可用连接
                    con = getConnection();
                }
            }
            catch (SQLException e)
            {   // 递归调用自己,尝试再次获取可用连接
                con = getConnection();
            }
        }
        else if (curConn < maxConn)
        {   //调用方法创建新连接
            con = newConnection();
        }
        else
        {   try
            {   //等待,如timeout 时间内仍没空闲连接,则
返回null wait(timeout);
            }
            catch(InterruptedException e)
            {   return null;
            }
        }
        if (con ! = null)
```

```
        {    curConn++;
        }
        return con;
    }
    /**
     * 创建新的连接 */
    private Connection newConnection()
    {    Connection con = null;
         try
        {   if (user == null)
              {    con = DriverManager. getConnection
(URL);
            }
            else
              {    con = DriverManager. getConnection
(URL, user, password);
            }
        }
        catch (SQLException e)
        {   return null;
        }
        return con;
    }
    /** * 将不再使用的连接返回给连接池
     * @param con 客户程序释放的连接 */
    public synchronized void freeConnection (Connection
con)
    {   // 将指定连接加入到向量末尾
         freeConnections. addElement(con);
         curConn--;
         notifyAll();
    }
    /*
     * 关闭所有连接 */
    public synchronized void release()
    {   Enumeration allConnections = freeConnections. el-
ements();
         while (allConnections. hasMoreElements())
         {   Connection con = (Connection) allConnec-
tions. nextElement();
            try
            {   con. close();
            }
            catch (SQLException e) {}
        }
        freeConnections. removeAllElements();
    }
}
```

这里，连接池经实例化后，一次性获得 minConn 数目的连接数，同时将连接对象存储在矢量对象 freeConnections 中。连接池提供 2 个同步方法 getConnection()和 freeConnection()来管理连接池中连接的分配与释放。如果向量中有多于一个的连接，getConnection()总是选取第一个。同时，由于新的可用连接总是从尾部加入向量，从而使得数据库连接由于长时间闲置而被关闭的风险减低到最小程度。第一个 getConnection()在返回可用连接给客户程序之前，调用了 isClosed()方法验证连接仍旧有效。如果该连接被关闭或触发异常，getConnection()递归地调用自己以尝试获取另外的可用连接。如果在向量 freeConnections 中不存在任何可用连接，getConnection()方法检查是否已经指定最大连接数限制。如已经指定，则检查当前连接数是否已经到达极限。如果当前连接数小于该值，该方法尝试创建新的连接。如创建成功，则增加已使用连接的计数并返回，创建新连接由 newConnection()方法实现。如果当前连接数已达到极限，则等待。如果在 timeout 时间内客户仍在等待连接，则返回 null，表示超时失败。release()提供了关闭连接池的方法，该方法遍历 freeConnections 向量并关闭所有连接，然后从向量中删除这些连接。

总之，由于 Java 有着其它语言不可比拟的优点，Java 的应用将越来越广泛，笔者着重阐述了 Java 数据库应用中连接池的基本工作机制，并实现了一个基本的连接池。对开发高性能的数据库应用程序有一定的实用和参考价值。

参 考 文 献

[1] HORSTMANN Cay S. Java2 核心技术(卷Ⅱ)：高级特性[M]. 朱志译. 北京：机械工业出版社，2000.

[2] PERRONE Paul J. J2EE 构建企业系统专家级解决方案[M]. 张志伟译. 北京：清华大学出版社，2001.

(编辑：刘勇)