基于 XML 的自优化数据库连接池的设计与实现*

魏常辉[1]，夏克俭[1]，曾德华[2]
(1.北京科技大学 信息工程学院，北京 100083；2.教育部信息中心 网络处，北京 100816)

摘 要：提出一种自优化数据库连接池的设计方式。它可根据应用规模动态地调整设置参数，进而选取对应的管理策略，其中策略可以存储在连接池的配置文件中或作为知识库保存。在对连接资源进行有效管理的同时，可以对连接池的最大连接数与最大等待时间进行动态调整，并在每次初始化时将优化的配置参数读入，实现连接池的自优化，提高数据库运行效率。

关键词：XML；自优化；数据库连接池；资源池；连接复用
中图法分类号：TP311.11 文献标识码：A 文章编号：1001-3695(2005)07-0206-04

Design and Implementation of Self-optimized Database Connection Pool Based on XML

WEI Chang-hui[1], XIA Ke-jian[1], ZENG De-hua[2]
(1. *School of Information Engineering, Beijing University of Science & Technology, Beijing* 100083, *China*; 2. *Network Department, Information Center in Educational Ministry, Beijing* 100816, *China*)

Abstract: A new design approach on database connection pool, self-optimized connection pool, is proposed in the paper. By using this method, connection pool can dynamic adjust configuration parameters and choose accordingly management strategies, which can be stored in configuration file or repository. Furthermore, this paper has implemented a case about database connection pool, which is of the function of adjusting preferences. The case of self-optimized connection pool improves greatly the performance for connection pool as a result of loading the optimized configuration parameters when every initializing the pool.

Key words: XML; Self-Optimized; Database Connection Pool; Resource Pool; Connection Reuse

开发基于 Web 的应用，不可避免地要频繁地访问数据库，然而数据库资源管理器进程频繁创建和删除连接对象会影响系统的性能。基于 Java 技术开发的电子政务系统中，由于要频繁地对数据库进行查询、插入、更新等操作，使得连接数据库需消耗大量时间。每次直接的数据库访问所需的连接开销包括：建立通信、内存资源占用(DBS 为每个物理连接分配一定的内存资源)、用户验证过程(注册过程)、安全上下文配置等任务。因此对于连接的使用成为整个系统的瓶颈，因而有必要建立“资源池”以提高访问性能。资源池是一种为解决资源频繁分配、释放所造成问题的设计模式。将该模式应用于数据库连接管理，就是通过建立一个数据库连接池来提供一套连接分配、使用策略，以达到实现高效连接、安全复用的目标。在基于 JDBC 的数据库应用开发中，数据库连接池的管理策略是一个难点，而连接池的配置参数选择对整个应用的性能尤为关键[1]。

1 Tomcat 数据库连接池使用分析及问题的提出

Tomcat 是 Apache Jakarta 的子项目之一，是 SUN 公司推荐的 JSP, Servlet 容器[2]。作为一款优秀的应用服务器，Tomcat 提供了数据库连接池、SSL、Proxy 等许多通用组件功能，其中连接池是 4.0 以上版本的新增功能，应用非常广泛。Tomcat 数据库连接池有两种典型的配置方法：手动修改配置文件和图形化界面配置。手动配置主要是修改 Server.xml 文件和某一具体应用下的 Web.xml 文件。Server.xml 中主要是设定连接池名称和连接池初始化时所需要的各个参数。Web.xml 文件中是加入池资源的引用，以便于利用 JNDI 进行查找使用，获得数据库连接。图形化界面配置相对较为简单，只需要在管理员界面，按照向导进行添加操作即可。无论是哪种配置方式，主要任务都是将启动连接池的参数填入指定文件，以便启动连接池时调用。

目前，无论是 Tomcat 连接池还是其他大型应用服务器的数据库连接池，在设定 max Active Connections, max Idle Connections, max Wait for Connections 等初始化参数时，其配置策略大都是静态的，即在连接池初始化时进行一次设定，不能根据应用需求的变化而动态调整。另外，设置配置策略时没有可参照的量化标准，如果要选取一个适合实际应用的优化参数不得不使用压力测试工具，花费大量的人力、物力进行实际应用的模拟，进而来选定参数值。从某种程度上来说，这样的测试还存在许多不确定因素，如测试工具的局限性、对应用规模的模拟程度、测试人员的技术水平、模拟操作和实际操作的差别等。因而，选定优化连接池配置参数，对于商业服务器连接池或是个人开发的数据库连接池都是一个需要解决的关键问题。

收稿日期：2004-07-05；修返日期：2004-09-21
基金项目：国家教育部教育信息化电子政务平台系统

2　自调整数据库连接池的设计模式及其实现

针对以上问题, 本文提出一种新的设计数据库连接池的方式——自适应数据库连接池。主要设计思想为: 数据库连接池采用 XML 格式文件作为配置文件, 为用户提供一个访问连接池的管理类 DBConnPoolAgent, 并将数据库的物理连接封装在连接的代理类 ConnectionAgent 中。管理类包括了对连接池资源及策略的管理, 通过这个类我们才能进一步对数据库的物理连接进行分类管理, 并按照预定的策略进行分配、释放连接以及对连接的异常处理。在管理类中, 有两个用来记录获取连接时间和释放连接时间的方法 writeStartInfo( ), writeEndInfo( ), 并在撤销连接池时将信息写入 XML 格式的日志文件, 便于随后的分析得出优化参数(如最大连接数、最小连接数、最大空闲时间等) , 供下一次启动连接池时读取, 从而有效地避免用户盲目的设置连接池参数。

2.1　设计模式

设计模式是近年来面向对象软件设计领域的研究热点之一[3]。其目的主要在于忽略具体应用环境中的细节, 在比较高的抽象层次上应用已被证明是行之有效的方法来解决在软件设计中重复出现的同类问题。不同的设计模式具有不同的抽象粒度和水平, 但它们之间存在共性, 可以选取一种通用的描述机制来进行模式描述。图 1 给出自适应数据库连接池在 UML 建模中的对象图。

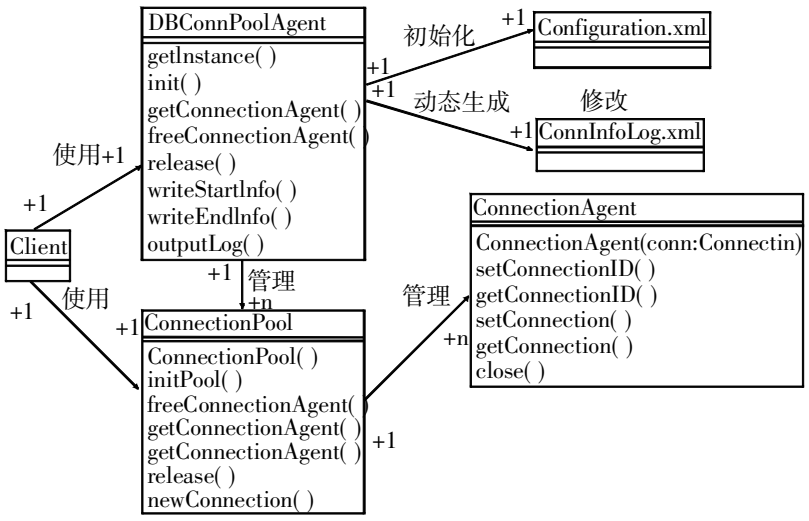

图 1　自调整数据库连接池对象图

2.2　具体实现

基于上面所述的设计思想和设计模式, 本文实现了一个自调整数据库连接池。它可以完成普通数据库连接池的基本资源管理功能, 实现数据库连接安全、高效的复用。复用有两层含义: ①指数据库物理连接建立并放入连接池中以后, 对于用户的每次连接请求连接池把该物理连接以独享的方式分配给用户, 当用户使用完还回到连接池后才能继续分配给其他请求的用户使用; ②对于用户的每次连接请求, 连接池并不是以独享的方式分配给用户, 即在一个连接被其他用户使用时, 就可以将这一连接再次分配给请求用户使用。

连接池的实现主要包括: 连接池智能管理类 DBConnPoolAgent、连接池类 ConnectionPool 和封装物理连接的连接代理类 ConnectionAgent。其中连接池智能管理类 DBConnPoolAgent 是实现整个连接池的主控类, 包括管理策略的选择、日志文件的分析、优化参数的生成及读取都是由这个类调用不同的功能模块来完成的。下面分别介绍对象模型图中各个对象的实现。

(1) 连接池类 ConnectionPool

连接池类 ConnectionPool 是连接池管理类的内部类, 它的所有方法只供管理类来调用, 本身不提供向外的接口。下面介绍它的参数意义及初始化连接池的过程:

```
private int checkedOut ;
//连接池中的连接总数, 包括使用的和空闲的
private Vector freeConnections = new Vector( ) ;
//存放连接代理类 ConnectionAgent
private int minConn; //最小连接数
private int maxConn; //最大连接数
private long waitingTime;
//请求连接的最长等待时间, 超出将抛出异常
private String name; //连接池类型
private String password; //数据库密码
private String url; //数据库地址
private String user; //数据库用户名
private synchronized void initPool( ) {
for( int i = 0; i < minConn; i ++ )
{ con = newConnection( ) ;
if( con!  = NULL) {
//如果物理连接可用, 将其放入代理类并存于缓存中
ConnectionAgent conbean  =  new ConnectionAgent ( con) ;
freeConnections. addElement( conbean) ; } } }
```

(2) 连接代理类 ConnectionAgent

该类封装了数据库的物理连接、标志连接状态的属性及相应的存取方法。

```
private Connection conn = NULL; //存放物理连接
private boolean inUse = false; //标志该连接是否被使用
private int usedCount = 0; //标志连接使用次数
    private int connectionID = 0; //连接被分配的 ID 号
    public Connection getConnection( ) { if( conn!  = NULL) return conn;
else return NULL; }
    public void close( ) { }
```

(3) 连接池管理类 DBConnPoolAgent

DBConnPoolAgent 类实际上是一个控制类, 它控制着数据库连接池的实现细节。同时, 它也是开发人员访问数据库的接口, 无论连接池的实现细节如何变化, 开发人员只需简单地调用其获得连接 getConnectionAgent 方法、释放连接 freeConnectionAgent 方法来建立和释放连接代理对象 ConnectionAgent, 并调用 ConnectionAgent 的 getConnection( ) 方法, 即可获得物理连接, 进行有关数据库操作。

①使用连接池的 Java 应用程序在初始化时调用 DBConnPoolAgent 的 init( ) 方法实现连接池的初始化(static synchronized public DBConnPoolAgent getInstance( )) 。该方法是公有的静态、同步方法, 采用“单态”模式用来返回管理类的唯一实例。执行 init( ) 方法时, 采用 JDOM 读取连接池的 XML 配置文件, 完成驱动注册和初始化连接池进入 Hashtable 类型变量中, 文件格式如下:

```
< ? xml version = "1. 0 "encoding = "UTF-8 " ? >
< ConnPoolPara >
< Pool >
< PoolTypeName > mysql < /PoolTypeName >
< Driver > org. gjt. mm. mysql. Driver < /Driver >
< Url > < ! [ CDATA [ jdbc: mysql: //localhost/edoas? user =
root&password = root&useUnicode = true&characterEncoding = gb2312 ] ] >
< /Url >
< User > < /User >
< PassWord > < /PassWord >
< MaxCon > 20 < /MaxCon >
< MinCon > 5 < /MinCon >
```

```
< WaitingTime > 10000 < /WaitingTime >
< /Pool >
< /ConnPoolPara >
```

②开发人员得到连接池管理类的实例后即可调用该类的 getConnectionAgent( ) 方法得到连接代理类对象,同时根据 MaxCon 的预设值进行不同管理策略转换也是在 getConnectionAgent( ) 方法中完成。对管理类 DBConnPoolAgent 的调用都是通过对 ConnectionPool 对象中的对应方法来间接完成操作的。

```
public ConnectionAgent getConnectionAgent( String poolTypeName) {
ConnectionPool myConnPool = ( ConnectionPool) pools. get ( poolTypeName) ;
if( myConnPool! = null) {
if( maxcon > "CONST") {
//返回内部类中以第二种复用方式分配连接的方法, CONST 用来切换两种复用方式的参数
return myConnPool. getConnectionAgent2( ) ; } else{
return myConnPool. getConnectionAgent 1( ) ;
//返回内部类中第一种复用方式分配连接的方法
} } return null; }
```

两种返回连接代理类的方法中,均调用 writeStartInfo(connID, new Date( ) . getTime( )) 方法将返回的连接的 ID 号及分配连接的时间写入日志文件。

③获得连接进行数据库操作后,开发人员应调用 DBConnPoolAgent 的 freeConnectionAgent(Sting poolName, ConnectionAgent conbean) 方法将连接返回到连接池。该方法也是调用连接池中的同名方法将指定的连接代理类返回到连接池中,并将该连接的使用终止时间用 writeEndInfo(connID, new Date( ) . getTime( )) 方法写入日志文件。

④开发人员在将连接代理类返回到连接池后,还要有一步操作,就是关闭所有连接,释放驱动程序。一般的连接池没有这一步操作,而本文实现的自调整连接池的分析、调整优化参数都被设置在这一步完成。

```
public synchronized void release( ) { /* 当整个应用程序中最后一个访问用户调用该方法时才能完全关闭连接* /
if( --clients! = 0) { return; }
Enumeration allSortPools = pools. elements( ) ;
while( allSortPools. hasMoreElements( ) ) {
ConnectionPool myConnPool = ( ConnectionPool) allSortPools. nextElement( ) ; //调用连接池类的 release( ) 方法,真正的关闭数据库连接
myConnPool. release( ) ; }
//将记录连接起止时间的"连接日志"写入文件
File autoLogFile = new File( "connectionLog. xml") ;
outputLog( autoLogFile) ; /* 使用搜索算法分析日志文件,得出并发连接数及并发连接占用时间,并选择最大、最小并发连接数和最长连接时间作为优化参数, 将其写回连接池配置文件中* /
SearchAlgorithm sa = new SearchAlgorithm( ) ;
sa. parseLogFile( "connectionLog. xml") ; }
```

2.3 关键技术

上述连接池中的关键技术完成了连接池优化参数的调整,使连接复用策略得以具体实现。

2.3.1 搜索算法的实现

日志文件记录的是连接池自运行起到撤销连接池整个期间所分配出去的每一个连接的起始时间和终止时间,其内容结构如下:

```
< ? xml version = "1. 0" encoding = "UTF-8" ? >
< ConnectionInfo >
< connection >
< RecordID > 1 < /RecordID >
< StartTime > 1084180470312 < /StartTime > < EndTime > 1084180470500 < / EndTime >
< /connection >
…
< /ConnectionInfo >
```

我们要求的是并发的连接数和并发时间。由于连接是顺序分配给用户的,因此 ID 号大的连接的起始时间一定大于 ID 号小的连接。基于这一特点的分析,如图 2 所示。

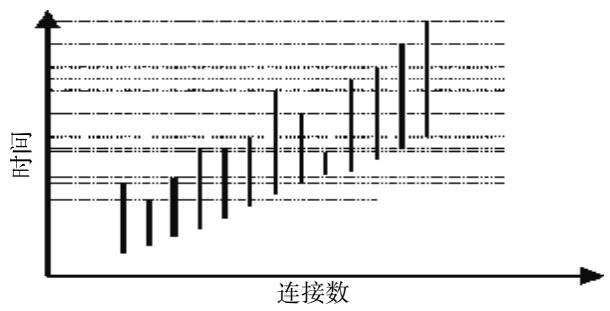

图 2 连拉池连接分配模型图

在图 2 中,水平轴代表连接数,纵轴表示时间;区域中的每条竖线代表连接池分配的一个物理连接,下端点指出该连接被分配的起始时间,上端点为该连接被释放的时间;垂直长度代表该连接被用户使用、占有的时间。下面给出对 XML 日志文件进行解析,并用优化的搜索算法进行参数的计算、选择的具体方法及其说明:

```
public class SearchAlgorithm { //类的属性略
public void parseLogFile( String filePath) {
/* 解析并遍历 XML 日志文档,将连接起始、终止时间写入指定数组( 省略代码) ,求并发连接数及并发的连接时间搜索算法实现。其主要思想是: 以每个连接的存活时间为基准,搜索其后的连接,如果其后的某一连接的起始时间小于该基准连接的终止时间,则计数器加 1,表明该连接为基准连接时间段的并发连接* /
for( int j = 1; j < = connNum; j + + ) {
nt count = 0; long tempEndTime = endTime[ j] ;
for( int k = j; k < = connNum; k + + ) {
if( startTime[ k] < endTime[ j] ) { count + + ;
/* 如果基准连接的终止时间在其后的某一连接的终止时间之后,则将用于计算并发时间的终止时间设定为该连接的终止时间,如"连接池连接分配模型图"前两个连接的情形* /
if( endTime[ k] < tempEndTime) { tempEndTime = endTime[ k] ; }
}
/* 判断基于一个连接时间基准的搜索是否完成,如果完成,则保存结果并发数及并发连接存活时间;退出本次循环,以下一个连接的生存时间为基准时间进行搜索,依次循环* /
if( ( startTime[ k] > = endTime[ j] ) | |( k = = connNum) ) {
long tempConcurrentTime = tempEndTime - startTime[ k] ;
if( tempConcurrentTime < 0) {
/* 该值小于 0,说明以 tempEndTime 为终止时间的连接与搜索的最后一个连接非并发,如图 2 中以第 6 个连接为基准连接时,第 10 个连接与第 13 个连接的关系。此时 count 值减 1; 并选取一种计算并发连接的方法* /
concurrentTime[ j] = endTime[ j] - startTime[ k - 1] ; } else{
concurrentTime[ j] = tempConcurrentTime; }
concurrentNum[ j] = --count; break; } } }
/* 本算法计算所得的并发连接数,及并发时间均为一组值;这组数值可以供连接池部署人员设定参数时参考。本文对两组数均选择其中最大值分别作为 maxConn, waitingtTime, 数组 concurrentNum 的最小值作为 minConn, 写入配置文件的对应节点中,便于下一次连接池启动时初始连接池之用,从而实现了连接池配置的自优化工作* /
//( 代码省略) 用 JDOM 解析配置文件,将 maxConn, minConn, waitingTime 写入配置文件}
```

2.3.2 第二种复用策略的实现

当从配置文件中读取的最大连接数 maxConn 大于一定值时(只可能出现在大规模的应用中) , 考虑到数据库本身的资

源管理器对连接分配的管理是多个用户进程共享有限的服务器进程, 因此可以将一个已分配的数据库连接, 在没有返回到空闲池时分配给其他请求用户, 进而使数据库服务器处理一些底层的并发操作。主要实现方法就是采用引用记数(Referencecounting) 的模式, 该模式在资源复用方面应用得非常广泛, 我们把该模式应用到连接的分配上。在 ConnectionAgent 类中使用 usedCount 属性标记连接池中的类使用的次数, 当连接池中无空闲连接时, 遍历使用池中的连接(已分配出去的连接) , 选择一个使用次数最小的连接分配给用户。搜索最小使用次数的方法如下:

```
public synchronized ConnectionAgent getConnectionFromUsed( ) {
int[ ] usedCount = new int[ maxConn] ;
for( int i = 0; i < maxConn; i ++ ) {
//遍历使用连接池中的代理类, 把每个连接的使用次数赋给数组
usedCount[ i] = ( ( ConnectionAgent) usedConnections. get ( i) ) .
getUsedCount( ) ; }
//调用 sortNo( ) 方法, 得到使用次数最小的连接的位置
int position = sortNo( usedCount) ; //得到连接代理类, 并返回
ConnectionAgent connBean = ( ConnectionAgent) usedConnections. get
( position) ;
return connBean; … }
// 输入: 整数数组, 输出: 该数组中最小数所在数组中的位置
public int sortNo( int[ ] a) { int min = a[ 0] ; int k = - 1;
for( int i = 0; i < a. length; i + + ) { if( min > a[ i] ) { min = a[ i] ;
k = i; } } return k; }
```

3　改进与展望

3.1　对事务处理的支持

可以为连接池加入事务处理机制, 作为进一步的改进。上面的论述是关于使用数据库连接进行常规操作, 对于事务处理情况比较复杂, 因为事务本身要求原子性保证。尽管 Connection 本身提供对事务的支持, 但要在自己开发的连接池中实现事务处理, 不得不提供相应的事务支持机制。比较实用的实现方法是: 采用显式的事务支撑方法, 每个事务独占一条连接, 这样可以降低对事务处理的复杂性[4] 。目前这一部分在本文的连接池中有待进一步完善。

3.2　知识库及评测模块

要实现完全意义上的自适应、自优化数据库连接池, 不可或缺的是要加入知识库和一整套高效的评价策略, 用于连接池本身的自完善、自调整。本文实现的连接池只是在这一思想指导下进行的初步尝试。随着该连接池在教育电子政务系统中的实际应用, 不断改进, 以后还会将较为完善的知识体系加入, 从而达到不需人的干预就可自动调整参数, 选择最优策略, 以便高效、安全地使用数据库连接资源。

4　结论

本文设计并实现了一个自适应数据库连接池。通过管理类读入配置文件初始化值, 在池的具体使用过程中生成记录连接信息的日志文件, 并对日志文件进行分析, 得到较优的配置参数再写入池的配置文件。利用这种循环取优方式, 可以在每次启动连接池时读取较为合理的配置参数, 避免了使用者的盲目设置。同时, 给出了分配连接的两种策略以及在不同参数配置情况下进行的自动切换, 减少了人员的直接设置工作。最后对完全意义上的自适应数据库连接池进行了展望。

参考文献:

[1]　美] 格尔克, 周立柱. 数据库管理系统[M] . 北京: 清华大学出版社, 2002.

[2]　The Tomcat5 Servlet/JSP Container User Guide[EB/OL] . http: //jakarta. apache. org/tomcat/tomcat-5. 0-doc/jndi-datasource-examples-howto. html, 2004-06.

[3]　[美] Deepak Alur, *et al*. J2EE 核心模式[M] . 北京: 机械工业出版社, 2002.

[4]　孔哲, 孟丽容, 等. 数据库连接策略优化方法[J] . 山东大学学报(工学版) , 2003, 33(6) : 652-657.

作者简介:

魏常辉(1979-) , 男, 天津人, 硕士研究生, 研究方向为信息管理及 XML 在 Web 上应用与研究; 夏克俭(1955-) , 男, 湖北人, 教授, 研究方向为智能管理、数据库技术; 曾德华(1965-) , 男, 湖北人, 高工, 研究方向为网络技术、信息管理平台构建与实施。

(上接第 205 页) 调用不同的方法对要写入的值进行预处理; ④把数据和长度传给注册表; ⑤关闭主键。

上面介绍的是底层的 C 方法, 在 Java 平台中使用本地方法包装这些 C 方法就能够利用 Java 语言在 Windows 系统中实现添加桌面、程序组快捷方式和修改注册表等系统相关性的操作。同样我们可以根据不同的系统特性, 分别生成本地库函数文件供安装程序调用, 以适应不同的系统环境, 从而实现通用安装程序的跨平台。

4　结束语

通用安装程序的设计与开发是一种有益的尝试, 它的实现有助于软件的安装与部署。利用数据驱动技术本文提出了通用安装程序一种可能的解决方案, 但是应该看到不同的被安装软件对安装程序的要求是不一样的, 有些可能还有特殊性, 因此完全的通用性是不可能实现的。本文中介绍的安装程序还可以进一步的完善, 目前软件的安装在不同的操作系统平台要提供不同的安装程序, 利用 Java 语言的可移植性并结合数据驱动的方法, 只需为每种平台提供不同的本地方法库和驱动文件, 就可以使安装程序运行在多种操作系统平台上, 从而节约了软件的开发成本。目前我们实现了 UNIX 和 Windows 环境下的安装, 证明了 Java 跨平台安装程序的可行性和实用性。

参考文献:

[1]　orstmann. Java2 核心技术卷 2: 高级特性[M] . 朱志. 北京: 机械工业出版社, 2001.

作者简介:

贺惠萍(1978-) , 女, 河南郑州人, 硕士研究生, 研究方向为计算机软件, 现参与第三代移动通信 Cdma2000 网管系统的开发; 翟好(1977-) , 男, 河南信阳人, 硕士研究生, 研究方向为网络安全; 裘鸿林(1965-) , 男, 江苏盱眙人, 副教授, 硕士, 研究方向为计算机网络。