文章编号：1005-8451（2006）02-0001-04

基于J2EE多层架构中Web层对数据库访问的优化

范后军，魏慧琴
（北京交通大学 计算机与信息学院，北京 100044）

摘 要：分析基于J2EE多层架构B/S应用系统对数据库访问的优化问题，着重研究Web组件层的利用缓存优化数据访问，并给出实际项目中应用的具体解决方案，该方案从两个方面实现对数据访问的优化：（1）有效缓存从数据库中读取的数据，提高已获取数据的利用率；（2）缓存对数据库的更新操作，实现一定的批量更新。同时给出该方案的测试，验证该方案所取得的优化效果。

关键词：JavaBean；J2EE；数据库；Web服务器；B/S模式
中图分类号：TP39 文献标识码：A

Optimizing of data access in web layer of multilayer based on J2EE

FAN Hou-jun, WEI Hui-qin
(School of Computer and Information Technology, Beijing Jiaotong University, Beijing 100044, China)

Abstract: It was analyzesd how to optimize data access in B/S model Multilayer Application System based on J2EE, and stressed the study of utilizing cache to optimize data access in the web layer, and detailedly presented a practicable solution that optimizes data access in two aspects: (1) effectively cache data gotten from the system database, increasing utilization of the data. (2) caching the operation of updating database to realize updating in batch. At the same time, a testing was given, which proved the optimal effect gotten from the solution.

Key words: JavaBean; J2EE; Database; Web Sever; B/S Model

企业应用开发中，基于B/S模式由于自身的优点逐步取代了传统的C/S模式，J2EE技术在B/S模式开发中广泛应用并得到认可，相对其他技术更加成熟、安全、开放，基于J2EE技术的多层架构所带来的良好的移植性、可维护性和复用性给开发者和使用者带来了巨大的方便。

J2EE平台利用数据库存储业务数据，该数据库通过JDBC API进行访问，这样，数据库能够被JavaBean、EJB等Web组件访问，在整个应用系统中，数据库访问调用最为昂贵，尤其是Web服务器与数据库服务器不在同一台机器上时，大、中型应用系统通常如此。因此，对数据库访问的优化可以在很大程度上提高应用系统性能。

基于J2EE的多层架构不仅仅带来了良好的移植性和复用性，同时也提供了更多空间进行优化数据访问，在各层几乎都可以采取相应对策，减少数据的访问次数和节约通信费用，以提高整个系统的性能。

收稿日期：2005-09-22
作者简介：范后军，在读硕士研究生；魏慧琴，副教授。

基于J2EE的MVC结构如图1所示。

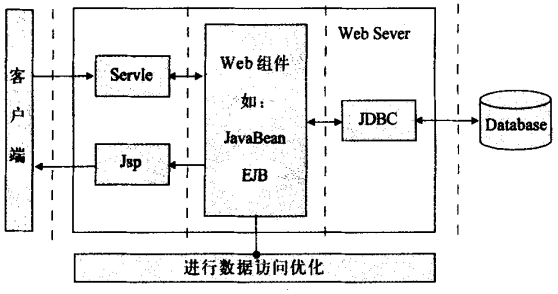

图1 基于J2EE的MVC结构图

整个系统可以大致分为如图1中虚线所隔离的5层，除了客户端即浏览器（主要功能是浏览）之外，在其他各层都可以对数据访问进行优化，例如，数据库层使用的存储过程，连接数据库时采用连接池，在JSP页面层的页面缓存技术等。

其中，Web组件层集中了应用系统各种业务逻辑，因此，在这层里可以充分利用业务逻辑的具体情况，对一些数据信息进行可靠和有效地缓存，实现优化。下面详细地讨论Web组件层中对数据访问的优化。

1 应用方案

B/S模式应用一般都是多个客户端访问同一个服务器，往往很多客户需要访问同一份数据，或者多个页面频繁地调用相同的数据，例如，在很多网站中的导航菜单信息、最新信息、分类信息和一些有必要突出展现给用户的信息等，每一个用户来访问或者每跳转另一个页面，可能都需要同样的某份数据，这时就不应该都去数据库中读取这些数据。如果能把这些被频繁访问的数据缓存起来，就能达到优化数据访问的目的。与次类似，可以把零散用户对数据库更新操作缓存起来，实现批量对数据库更新操作，从而达到提高系统性能的目的。在实际项目开发中，采用JavaBean技术就可以实现对数据结果集的缓存。以缓存导航栏主菜单信息为例，说明在实际项目开发中应用的解决方案。演示代码如下：

```
package datacache;
import java.util.Collection;
import java.util.ArrayList;
// 自定义JavaBean，主要属性：id，url，name
import beans.Menu;
public class MyDataCache{
  //menus 用来缓存主菜单结果集
  private static Collection menus=null;
  // 实现批量往数据库中写入数据
  private static Collection addedMenus =new
ArrayList();
  public static Collection getMenus(){
    if (menus = = null){
menus=getMenusFromDatabase();
    }
  // 获得缓存数据集的副本，如果不考虑外
  // 部对 menus 的非法修改
  // 可以直接 return menus;
  return menus.clone();
}
private static Collection
                getMenusFromDatabase(){
    // 连接数据库，查询数据库
    // 获取 menus 信息，代码省略
          ...
  }
public synchronized static void
          addMenu(Menu menu){
  // 即刻更新缓存中的 menu
    menus.add(menu);
    // 缓存需要添加的 menu
  addedMenus.add(menu);
  //n 的值根据实际情况设定批量大小
  if (tmpMenus.size()>n){
  // 批量写入数据到数据库中
  addMenusToDatabase(addedMenus);
  // 清空 tmpMenus
    addedMenus =new ArrayList();
  }
  }
  private synchronized static void
        addMenusToDatabase(Collection
          tmpMenus){
        // 连接数据库，将 tmpMenus 中多个
menu
      // 一次写入数据库中，代码省略
            ...
}
}
```

分析上述代码，MyDataCache类中有一系列静态成员变量和静态方法，这里的静态成员变量就充当了缓存的角色，静态方法就是对这些缓存的业务操作，把这些成员和方法定义为静态的，很方便程序中直接通过类名来调用方法，确保同一业务访问的是同一缓存。第1个用户来访问系统时，需要从数据库中读取菜单信息，在页面上显示导航菜单。Web组件层调用：MyDataCache.getMenus()，由于是第1个用户访问，MyDataCache.menus为null，需要调用MyDataCache.GetMenusFromDatabase()去数据库中读取数据，而后续的用户来访问或访问其他页面再次调用MyDataCache.getMenus()时，MyDataCache.menus中已经读入了数据库中的数据，直接返回它的副本。这样就可以减少对数据库的访问次数，有效地改善了系统性能。

读者还可以在这段代码中看到添加菜单的操作：addMenu()，在增加导航菜单时，将新的menu添加到MyDataCache.menus时，并不立刻写入数据库，而是用MyDataCache.tmpMenus又做了缓存，当新增menu达到一定量时才一次性写入数据库，在这里，批量也可以同时根据时间来决定何时批量写

入数据库，如每天都写一次数据到数据库。相比传统的方式：每次新增一个menu时，都要立刻写入数据库，然后又要重新读出数据库中所有的菜单信息，大大减少了对数据库的访问。类似的操作修改菜单（updateMenu）和删除菜单（deleteMenu）代码与增加菜单（addMenu）代码类似，读者可以在实际应用开发中自行补全。如果把删除、增加和更新3个操作再批量地结合起来，可以进一步优化。

概括起来，Web组件层数据缓存有两方面：（1）有效缓存从数据库中读取的数据，提高已获取数据的利用率；（2）缓存对数据库的更新操作，实现一定的批量更新，从而达到减少数据库访问次数，节省通信带来的开销，以提高整个系统的性能。

2　性能测试

为了充分说明上述方案的优化效果，下面做一个测试。

navaigator.jsp产生导航菜单的JSP代码，可以被其他页面包含显示。

```
<@page contentType="text/html,charset=GBK">
<@page import="java.util.* ">
<@page import="datacache. MyDataCache">
<@page import="beans.Menu">
<%
// 服务器端读取数据开始时刻
long begin_time= System.curentTimeMillis();
System.out.println(
      "server begin: "+begin_time);
// 从Web组件层获取导航菜单信息
Collection menus=MyDataCache.getMenus();
// 服务器端读取数据结束时刻
long end_time= System.curentTimeMillis();
System.out.println("server end:"+end_time);
System.out.println(
    "server cost:"+(end_time-begin_time));
Iterator it=menus.iterator();
While(it.hasNext()){
     Menu menu=(Menu)it.next();
%>
    <a href="<%=menu.getUrl()%>">
    <%=menu.getName()%></a>
<%}%>
```

该JSP代码中包含了测试服务器的业务处理时间，客户端测试可以通过JavaScript来实现，JavaScript是运行在客户端的脚本语言，几乎所有的浏览器都支持。写一个静态页面连接到该JSP页面，同时记录客户端点击的时间，然后再在navaigator.jsp中添加一些JavaScript代码，输出该JSP页面在客户端显示完成的时间，代码省略。用于测试的机器CPU赛扬900 MH，内存256 M，Web服务器为tomcat，数据库mysql，本地机测试该页面：

（1）不使用缓存结果集，修改方法MyDataCache.getMenus()，

```
public static Collection getMenus(){
    // 从数据库中读取数据
    menus=getMenusFromDatabase();
    return menus;
}
```

4次访问的客户端响应时间和WebServer处理时间测试数据分别如下表1和表2。

表1　不要使用缓存第1次测试数据表

访问	client begin	client end	cost
1	12:9:58:483	12:10:7:556	9.073 s
2	12:12:3:643	12:12:4:494	0.851 s
3	12:13:41:283	12:13:41:924	0.641 s
4	12:14:33:188	12:14:33:949	0.761 s

表2　不要使用缓存第2次测试数据表

访问	server begin	server end	cost
1	1126930204381	1126930207045	2.664 s
2	1126930324023	1126930324063	0.040 s
3	1126930421654	1126930421714	0.060 s
4	1126930473528	1126930473628	0.100 s

（2）使用缓存，4次访问，客户端响应时间和WebServer处理时间测试数据分别如表3和表4。

表3　使用缓存第1次测试数据表

访问	client begin	client end	cost
1	11:54:26:963	11:54:33:703	6.740 s
2	11:59:38:251	11:59:38:671	0.420 s
3	12:1:41:999	12:1:42:580	0.581 s
4	12:4:28:919	12:4:29:650	0.731 s

表4　使用缓存第2次测试数据表

访问	server begin	server end	cost
1	1126929271069	1126929273483	2.414 s
2	1126929578631	1126929578631	0.000 s
3	1126929702349	1126929702349	0.000 s
4	1126929869259	1126929869259	0.000 s

分析上述测试数据，客户端的时间是通过javascript代码输出的，与WebServer通过Java代码

输出时间格式不一样。客户端响应时间从总体上看，用缓存比没用缓存很明显快得多。再仔细观察Web服务器的处理花费，用缓存与没有使用缓存相比，第1次访问花费的时间相差不大，这是由于两者都需要去数据库中读取数据。而后续的访问差别就相当明显了，前者没有使用缓存，每次访问都需要去数据库中读取数据，花费时间40 s、60 s、100 s不等，后者由于对数据进行了缓存，重新显示页面的时候直接从内存中读取数据非常之快，以至于测试时间花费为0 s（太快了），可以看出性能上差别非常明显。如果是多个用户同时访问，所表现出来的优越性能将更加明显。

3 需要解决的问题

在使用该方案时，需要注意着重解决如下两个问题：

（1）缓存要占用内存，如果当需要缓存的数据量很大，例如一些大的论坛中大量不断更新的帖子，如果缓存帖子数据将带来的系统过大的内存开销，似乎行不通。但是完全没必要缓存所有的帖子，对于每个论坛板块只需要缓存一定量（根据实际的硬件条件来决定该数量）最新的帖子，大多数用户主要还是看最新的帖子，解决了大问题，可以大大改善系统的性能了。在实际开发的应用系统中，一般来说，业务数据中被经常访问的数据相对整体来说往往是很小的一部分，例如前述论坛中最新的帖子和所有的贴子，只要对这些最常用的少量数据进行有效地缓存，就可以实现很可观的数据访问优化。在Web层根据业务执行的具体情况确定哪些数据是最常用的，可以动态地对这些最常用数据进行有效缓存。缓存数据不管如何总是要多占些内存，当前硬件设备内存一般几百M甚至几个G，利用几十M内存进行缓存一些经常使用的数据来提高整体性能是值得的。

（2）缓存数据，实现批量对数据库更新的时候，如果系统出现异常，如何确保把缓存中尚未完成数据库的更新操作写入数据库，利用Java的IO流很容易解决这个问题，动态地在Web服务器硬盘中保存缓存中尚未完成的批量更新操作，若系统异常，待系统重启时，读取这些尚未完成的更新操作对数据库进行实际的更新。

（3）如果是一个数据库对应多个应用系统，就会出现如何确保两个应用系统访问数据的一致性，解决方案：利用到触发器，将缓存数据的操作与对应表相应的操作建立对应关系以确保同步，这就涉及更多的JDBC技术，实现确实比较有难度，但是这样能够确保各个应用系统较强的独立性；如果条件允许，可以让多个应用系统共用这些缓存数据的Web组件，在不同体系之间可采用WebService来实现，这将又会增加系统开销，如何取舍需要根据实际应用情况来决定。因此该方案主要适合一个应用系统独占一个数据库的情况。

4 利用EJB缓存数据

无状态会话Bean也为缓存数据提供了一个比较好的解决方案，EJB容器维护着一些可用的bean实例，bean实例的生命周期由EJB容器根据使用情况来决定，通常，一个bean实例供某个用户使用后并不是立刻销毁，而是回收到EJB所维护的池中，为可用状态，因此可以将一些大的数据对象作为Bean的成员对象进行缓存，当第1个用户访问时去数据库中读取数据，以后使用该bean实例的用户不需要再次去数据库中读取数据，可以直接利用bean的成员，从而实现优化。

5 结束语

在基于J2EE整个应用系统中，Web组件层集中了应用系统的主要业务，根据业务具体情况，可以更好地选择数据，进行有效缓存，提高数据利用率，同时，还可以在业务层缓存一些更新数据库的操作，实现一定的批量更新数据库，从而减少数据访问次数和数据通信的开销，实现系统性能优化。

参考文献：

[1] Gregory Nyberg Robert Patrick. 精通BEA WebLogic Server 构建与部署J2EE应用的最佳策略[M]. 北京：电子工业出版社，2004.

[2] 飞思科技产品开发中心. Oracle 9i J2EE应用开发指南[M]. 北京：电子工业出版社，2003.

[3] Mihir Kulkarni. Performance Tuning EJB Applications [N] http://dev2dev.bea.com/pub/a/2005/02/perf_tune_session_beans.html

[4] Bruce Eckel. Thinking in Java[M]. (3rd ed) 北京：机械工业出版社，2004.