第 32 卷 第 19 期 计 算 机 工 程 2006 年 10 月
Vol.32 № 19 Computer Engineering October 2006
·软件技术与数据库· 文章编号：1000—3428(2006)19—0068—03 文献标识码：A 中图分类号：TP301.6

基于 LRU 算法的 Web 系统缓存机制

张震波，杨鹤标，马振华
(江苏大学计算机科学与通信工程学院，镇江 212013)

摘 要：Web 系统嵌入缓存机制将被访问的对象保存在内存缓冲区中，在频繁创建和销毁对象时，降低了系统开销、提高了系统的整体快速响应能力、避免了频繁的数据交互。该文分析了 Cache 技术的设计原理和实现策略、LRU 算法的设计模式，构造了缓存机制的基本框架。
关键词：J2EE；LRU 算法；缓存机制；设计模式

Web Cache Mechanism Based on LRU Algorithm

ZHANG Zhenbo, YANG Hebiao, MA Zhenhua
(School of Computer Science & Communication Engineering, Jiangsu University, Zhenjiang 212013)

【Abstract】 The cache mechanism of Web system that stores object in the memory buffer, lowers the systematic expenses produced by frequently creation and destruction of the object, avoids frequently data exchange with database and improves the fast response ability of system. By analyzing the design and implementation principle of cache, the cache framework based on J2EE with LRU algorithm is implemented.
【Key words】 J2EE; LRU arithmetic; Cache mechanism; Design patterns

缓冲机制是衡量 Web 系统是否成熟的重要标志。引入缓冲机制可以缩短信号请求的处理时间，提高系统整体响应能力。Web 系统通常有大量经常被访问的对象，为了降低创建和销毁对象的开销，避免频繁和数据库进行交互，将经常被访问的对象保存在内存缓冲中，可以大幅度地提高系统的性能。

1 Cache 设计原理

缓存机制就是在原始数据第 1 次读取后保存在内存中，第 2 次读取时，就直接从内存中读取。原始数据可能保存在持久化介质或网络上。缓存机制从原理上讲比较简单，但在具体实现时需要考虑的技术指标是比较多的。

1.1 缓存命中率

命中率是指从缓存中得到请求对象的百分比，它是考察缓存机制优劣的关键参数。

1.2 对象生命周期

缓存对象每天甚至几分钟就要更新(如论坛)，有的却不需要更新。有生命周期的对象不管访问频度如何，生命周期结束时它必须替换出缓存。因此缓存对象何时该消亡，以什么样策略被更新，也是缓存系统必须要解决的问题。

1.3 对象的一致性

缓存对象是原对象映射，设计缓存时必须考虑如何确保缓存对象和原对象的一致性，即当原对象发生变化时，缓存象也必须立即更新。

此外，设计缓存时还需要综合考虑缓存大小和缓存对象大小，缓存对象更新和清除以及系统时间戳等一系列问题。

2 Cache 应用策略

Cache 应用策略最常用的有 LRU(Least Recently Used)策略、LFU(Least Frequently Used)策略和 Size 策略。

2.1 LRU 策略

LRU 算法广泛应用于 CPU 缓存及虚拟内存中，本文基于 WWW 访问中存在的时间局部性考虑，应用 LRU 策略将最不常被访问的对象替换出缓存。

该算法描述如下：用户访问的对象大小为 L，如果缓存命中，直接返回该对象并将之调整到引用序列表首位；否则从存储介质读取，加入缓存列表，若缓存空闲空间小于 L，重复将最近最少使用对象替换出缓存，直到空闲空间至少为 L 为止。当引用的序列表现出明显的时间局部性时，曾引用过的对象可能不久再次被引用，该算法便有出色表现。LRU 用于 Web 缓存时，需要记录下每一个引用过对象的存入缓存时间和最后一次访问时间。

2.2 LFU 策略

LFU 是基于 Web 页面的访问特性，将访问频率最低的对象替换出去。

存入缓存的对象都内置有计数器，每发生一次缓存命中，相应对象的计数器加 1。如果缓存失效，触发缓存替换，访问次数少的对象被替换出缓存。如果两个以上对象有相同的访问次数，使用 LRU 算法打破僵局。LFU 的一个潜在弱点就是缓存中的部分对象积累访问次数很大，即使不再被引用而成为“死对象”，它们也不会出现在替换的候选名单中，缓存就发生了“污染”。显然，“污染”会导致缓存效率降低。

2.3 Size 策略

Size 策略多用于代理服务器的缓存设计。当发生缓存替换时，缓存中最大的文档被替换出缓存，以便容纳更多小文档。如果出现僵局，使用其它策略来打破。

Size 策略通过替换较大的对象来最小化失误率，但有时会出现一些小对象一直没有被访问，却长时间滞留在缓存中，造成缓存“污染”。鉴于此，可结合使用 LRU 策略进行优化，

基金项目：国家高新技术研究发展计划基金资助项目(2004AA414031)
作者简介：张震波(1976－)，男，硕士生，主研方向：现代软件工程，模型驱动，Web 框架技术；杨鹤标，副教授；马振华，硕士生
收稿日期：2005-10-19 **E-mail：**zzb205@163.com

综合考虑局域性、对象大小、延迟代价等因素，把使用率最低的替换出缓存。

3 设计模式

Cache 框架实现时采用了单态模式、工厂模式、观察者模式和(动态)代理模式等多种设计模式。

设计模式是对被用来在特定场景下解决一般设计问题的类和相互通信的对象的描述[1]。设计模式其实是在开发中被反复凝练，再经过分类编目的代码设计经验的结晶。通过运用设计模式，可以让系统更加具有可复用性和可伸缩性。

4 基于 J2EE 架构 Web Cache 框架的实现

Cache 框架主要类图如图 1 所示。为了实现它，需要解决时间戳、缓存对象大小、对象一致性、生命周期及对象过期清除和缓存命中率等问题。

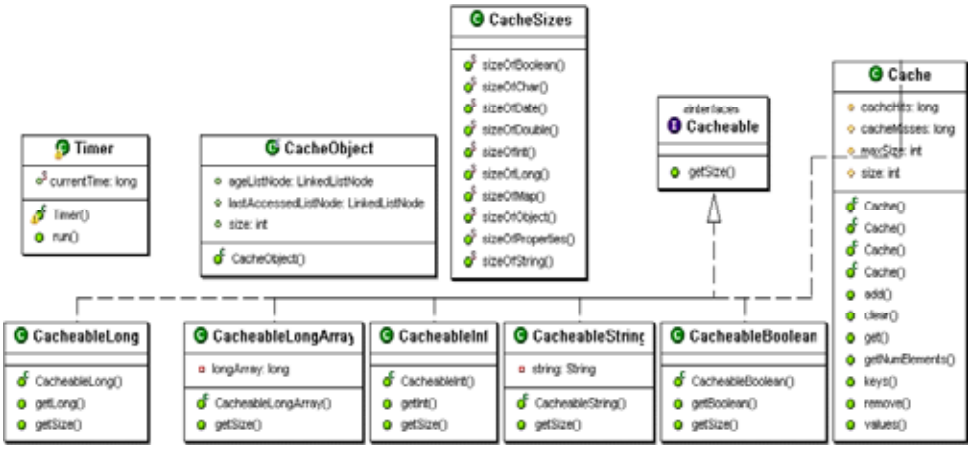


图 1 基于 J2EE 架构下的缓存机制

4.1 时间戳的设计

缓存需要对对象进行更新和消亡等处理，因此需要一个时间戳。而 Java 中执行 System.currentTimeMillis()很耗费性能，如果 get 操作都执行该语句将会形成性能瓶颈，因此考虑设置一个全局时间戳来实现秒的更新。

Timer 类主要功能就是每间隔 1 s 更新当前静态时间变量 curTime。设计时采用单态(SingleTon)模式，保证系统中任意时刻仅有一个对象实例，在其构造函数中使用 setDaemon 方法将自己设置为守护进程，以便在主进程结束后自动退出。这种设计会使得缓存过期时间有 1 秒到几秒的误差，但系统性能得到了提高，而且从实际应用中来看，没有对系统功能造成影响，程序如下：

```
public class Timer extends Thread{
private static long curTime;
private static int interval=1000;
static{getInstance();}
private Timer(){this.setDaemon(true);start();}
private static Timer instance=new Timer();
public static Timer getInstance(){return instance;}
public void run(){
while(true){
  curTime=System.currentTimeMillis();
  Cache.curTime=curTime;
  try{sleep(interval);}
  catch (InterruptedException ie){}
}}}
```

4.2 缓存大小和缓存对象大小的设计

在 Cache 类中可以定义：int maxSize=64*1 024;用来表示缓存的最大限制，默认为 64kB，原则上 maxSize 越大性能越高，使用中应视实际内存大小设定。进行 setMaxSize 时需要检查缓存大小是否超过设置的最大值，若超过则必须调用删除清理的方法。

进行缓存操作时，必须知道存入对象的大小，这个前提条件的设定可以保证缓存增长时不会超过规定的最大缓存值。因此需要编写一个 Sizes 类，用来计算存入对象的大小。

对象引用占 4B；基本类型如 int 的 Size 值为 4；boolean 的 Size 为 1；Date 的 Size 为 12 等，字符串可以通过求其长度换算得到，而复杂对象可以通过基本类型组合得到：

```
public class Sizes{
public static int sizeOfObject(){return 4;}
public static int sizeOfInt(){return 4;}
public static int sizeOfString(String str){
return (str==null) ? 0 :
(4+str.length()*2);}
public static int
sizeOfDate(){return 12;}  …}
```

4.3 缓存对象生命周期的设计

如果缓存增长时空闲空间不足，则发生缓存替换，不被经常访问的对象首先从缓存中删除。如果设置了对象最大生命周期，那么即使这个对象被频繁访问，也将从缓存中删除。这个特性适用于一些周期性需要刷新的数据，如来自数据库的数据。

实现方式是在 Cache 中定义 long maxLifetime=-1；其值为-1 时，视为不设最大生命周期。

4.4 缓存击中率的设计

缓存击中率用来判断缓存设计的优劣。在 Cache 类中定义：long cacheHits = cacheMisses = 0L；每发生缓存命中时，cacheHits++，而当访问对象不在缓存中时，就从物理设备(数据库、网络或 XML 文件等)读取，cacheMisses++，这两个变量用于评测系统缓存效率。

4.5 Cache 类的实现

Cache 类实现了缓存机制的大部分行为，它将要缓存的对象加到 HashMap 映射表中，用时启用两个 LinkedListed 双向链表，保存对象的访问顺序和对象加入缓存顺序，当然链表中保存的是对象引用，真正的对象存放在 Hash 表中。替换策略采用 LRU 算法，缓存对象被访问，就把它放到链表最前端，不定期地把要缓存的对象加入链表中，把过期对象删除，如此反复。

Cache 类中有两个重要方法 add 和 get。add 方法是向 Hash 表中添加关键词为 Key 的缓存对象 Object，加入前需判断该对象大小，太大则不存入缓存，存入后还要删除过期对象和判断缓存状态等操作，如果缓存太满则重复删除不常访问对象直至空闲空间可以存放新加入对象；get 方法是从 Hash 表中得到关键词为 Key 的缓存对象，同时调整缓存命中率参数 CacheHits 和 CacheMisses，并将该缓存对象从 LeastRecentList 链表中取下插到链表头部。

```
public class Cache implements Cacheable{
protected static long curTime=Timer.curTime;
protected HashMap objectInCache;
```

```
protected LinkedList leastRecentList,ageList;
…
public synchronized void add(Object key, Cacheable object){
remove(key);
int objSize=object.getSize();
if(objSize>maxSize*.90){return;}
size+=objSize;
CacheObject obj=new CacheObject(object, objSize);
objectInCache.put(key, obj);
LinkedListNode
leastRecentlyNode=leastRecentList.addFirst(key);
obj.leastRecentListNode=leastRecentlyNode;
LinkedListNode ageNode=ageList.addFirst(key);
ageNode.timestamp=System.currentTimeMillis();
obj.ageListNode=ageNode;
cutCache;//Cache 满则删除最近最少使用对象}
public synchronized Cacheable get(Object key){
deleteExpiredEntries();
CacheObject obj = (CacheObject)objectInCache.get(key);
if(obj==null){cacheMisses++;return null;}
cacheHits++;
cacheObject.leastRecentListNode.remove();
leastRecentList.addFirst(obj.leastRecentListNode);
return cacheObject.object;}}
```

4.6 对象的一致性设计

缓存中对象是原对象映射，系统实现必须确保缓存中对象和原对象的一致性，原对象发生变化时，缓存中的对象也必须立即更新。因此，设计时采用了行为模式中的观察者模式，Cache 中一个对象状态发生改变，所有依赖于它的对象都得到通知并被自动更新，实现了对象间的低耦合。具体做法是将被观察者继承至 java.util.Observable，观察者需要实现 java.util.Observer 接口中的 update(Observerable thing, Object obj)方法。

4.7 对象的过期和清除

Remove 方法中需要从 Cache 中的 Hash 表中移除对象，从两个 LinkedList 链表中删除对象的引用，同时修改 Cache 中 Size 的值。为了确保在某一时刻，删除操作只有一个线程，其方法签名为 public synchronized void remove(Object key)。

5 结束语

缓存机制是提高 Web 系统运行性能必不可少的技术，设计优良的缓存机制可有效地缓解大量用户并行访问系统时对服务器造成的巨大压力。本文采用 LRU 算法结合设计模式，初步实现了一个性能优良的基于 J2EE 架构的缓存机制的基本框架。进一步研究内容主要包括：(1)结合 LFU 算法和 Size 策略，优化 Cache 实现策略；(2)考虑分布式集群 Cluster 环境下，Cache 的设计及 Cache 技术在多 JVM 环境中的应用。

参考文献

1 Gamma E, Helm R, Johnson R, et al. Design Patterns: Elements of Resuable Object-oriented Software[M]. Texas, USA: Addison Wesley Longman, 1995.

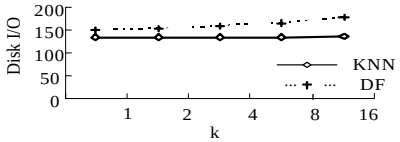
2 Bestavros A, Shudong J. Popularity-aware Greedy Dual-size Web Proxy Caching Algorithms[C]. Proc. of the 20th International Conference on Distributed Computing Systems, 2002: 254-261.

3 Nottingham M. Caching Tutorial for Web Authors and Webmasters[EB/OL]. http://www.mnot.net/ cache_docs/, 2003.

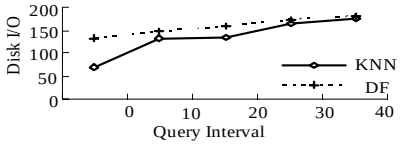
4 贺 琛, 陈肇雄, 黄河燕. Web 缓存技术综述[J]. 小型微型计算机系统, 2004, 25(5): 836-842.

5 王本年, 曹先彬. 一种域分布式合作 Web 缓存系统[J]. 计算机研究与发展, 2002, 39(3): 275-279.

（上接第 61 页）



(a) 当 k 改变时的查询性能



(b) 当查询时间段改变时的查询性能

图 3 实验结果

4 结论

本文基于 TPR 树这一动态时空索引结构，提出了一种可以通过一次查询遍历 TPR 树便可找到多个最近邻对象的查询方法，并针对数据更新的问题提出相应的解决办法。该方法能够支持动态条件下对多个最近邻移动对象的查询，并在性能上较已有算法有较大的提高。未来的工作将致力于对算法作进一步的改进以及在动态条件下对移动对象的连续最近邻查询的研究。

参考文献

1 Sistla A P, Wolfson O, Chamberlain S, et al. Modeling and Querying Moving Objects[C]. Proceedings of the 13th International Conference on Data Engineering, 1997: 422-432.

2 Hjaltason G R, Samet H. Distance Browsing in Spatial Databases[J]. ACM Transactions on Database Systems, 1999: 24(2): 265-318.

3 Cheung K L, Fu A W. Enhanced Nearest Neighbour Search on the R-tree[J]. ACM SIGMOD Record, 1998: 27(3): 16-21.

4 Roussopoulos N, Kelley S, Vincent F. Nearest Neighbor Queries[C]. Proceedings of the ACM SIGMOD International Conference on Management of Data, 1995: 71-79.

5 Benetis R, Jensen C S, Karciauskas G, et al. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects[C]. Proceedings of the International Data Engineering and Applications Symposium, 2002: 44-53.

6 Saltenis S, Jensen C S, Leutenegger S T, et al. Indexing the Positions of Continuously Moving Objects[C]. Proceedings of the ACM SIGMOD International Conference on Management of Data, 2000: 331-342.

7 Tao Y, Papadias D. Time-parameterized Queries in Spatio-temporal Databases[C]. Proceedings of the ACM SIGMOD Conference, 2002: 334-345.

8 Song Z, Roussopoulos N. K-Nearest Neighbor Search for Moving Query Point[C]. Proceedings of the 7th International Symposium on Spatial and Temporal Databases, 2001: 79-96.