文章编号:1001 - 9081(2005)01 - 0069 - 04

轻型永久对象管理器缓存机制的设计与实现

贺 庆,卢显良,宋 杰
(电子科技大学 计算机科学与工程学院,四川 成都 610054)
(qinghe@uestc.edu.cn)

摘 要:在信息系统中,理想的应用程序结构应该是基于对象模型的,但是数据几乎总是保存在关系数据库中。永久对象管理器的位置处于对象模型和关系模型之间,它将应用程序的对象映射到关系数据库的表中,从而对应用程序开发者隐藏了下层的实现,简化了应用程序的开发。缓存机制是构成永久对象管理器的重要组成部分,它通过缓存最近被访问过的对象来减少对数据库的访问。文中分析了一个对象缓存机制的设计和实现,并通过测试证明引入该缓存机制可以极大地提高系统的性能。

关键词:永久对象;对象模型;关系模型;对象-关系映射;对象缓存机制;缓存视图;Java 反射机制;Java 垃圾收集器

中图分类号: TP331 **文献标识码**:A

Design and implementation of a cache mechanism for a lightweight persistent object manager

HE Qing, LU Xian-liang, SONG Jie
(College of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu Sichuan 610054, China)

Abstract: Enterprise information systems, e-business systems and other WEB application software should base on the object model ideally, but the data are almost always stored in relational databases. A persistent object manager is between the object model and the relational model. It conceals the implementation of the lower layers from the application programmers by mapping objects from the applications to the tables of relational databases, and so simplifies the development of application programs. A cache mechanism is a critical component of a persistent object manager. It caches recently accessed objects, and therefore, decreases access to the database. In this paper, the design and implementation of an object cache mechanism were analyzed and this mechanism was proved to improve performance of the system greatly with a test.

Key words: persistent object; object model; relational model; object-relational mapping; object cache mechanism; cache view; Java reflection; Java garbage collector

0 引言

目前,无论是信息系统、电子商务系统或其他的 Web 应用软件,大都是基于对象模型的。这也是此类系统理想的结构。然而,由于目前对象数据库实现还不完备,性能和可扩展性方面都不能满足要求,大多数系统是将数据存储在关系数据库中的。关系数据库所使用的关系模型和对象模型从本质上讲是不相关的,也不具备面向对象的继承、关联等概念,这使得应用程序开发者不得不同时处理两种截然不同的模型,系统开发的复杂性大大增加。永久对象管理器在对象模型和关系模型之间搭起了一座桥梁,它负责把对象转换成关系数据库中的记录加以存储,以及根据记录重构对象;它为应用程序员提供了一个一致的面向对象的接口,而隐藏了下层的具体实现,使得应用程序的开发被大大简化[1-3]。

对象缓存机制(以下简称缓存机制)是构成永久对象管理器的重要组成部分,通过缓存最近被访问过的对象,它可以减少对数据库的访问,同时也避免了每次重构对象的额外开销,从而大大提高系统性能[4,5]。目前应用比较广泛的对象-关系映射模型之一 Hibernate 就可内建 EHCache 或 JCS 来进行对象的缓存。

我们的轻型对象管理器最初是为一个远程技术支持系统设计的。与大型的信息系统不同,这个系统需要处理的对象种类不多,对象与对象之间的关系也比较简单,但它有自己特殊的事务特性,同时系统吞吐量要求比较高,这些使得 JDO 和 Hibernate 这类复杂的 ORM 并不适用[6]。基于以上原因,我们也必须自己来实现对象管理器的缓存机制。虽然我们的初衷只是针对一个专用的对象管理器实现对象缓存,但是它的设计思想和算法是可以为类似系统提供有益的借鉴的;并且在开发完成以后,我们也发现,无论是对象管理器还是其中的缓存机制的实现,都依然具有一定的通用性。

1 设计思想

1.1 对象管理器结构

本文集中讨论对象管理器的缓存机制的设计和实现,所涉及的对象管理器采用了文献[6]中所述结构,在本文中不再做专门的描述,但在涉及缓存机制的部分以及有改动的部

收稿日期:2004 - 06 - 28;**修订日期**:2004 - 11 - 29 **基金项目**:信息产业部电子发展基金资助项目(信部运[2004])

作者简介:贺庆(1980 -),男,四川成都人,硕士研究生,主要研究方向:计算机网络、操作系统、数据库; 卢显良(1944 -)男,河北人,教授,博士生导师,主要研究方向:计算机网络、操作系统; 宋杰(1972 -)男,四川成都人,博士研究生,主要研究方向:计算机网络、操作系统.

分会有提及。

需要强调的是每个永久对象类都是 PersistentObject 的子类,而我们的系统又采用了紧耦合的结构,这意味着缓存机制是永久对象类的共有属性,应该被实现于基类 PersistentObject 中。然而,每个永久对象类有权利选择是否使用缓存机制以及如何使用。简单地说就是缓存机制是共有的,但是缓存却是每个永久对象类所独有的,并以 Singleton 模式来被实现。这样,可以认为每个永久对象类都有自己独立的缓存管理器,这与 EHCache 每个类装载器只有一个缓存管理器的结构是不同的[7]。

1.2 设计目标

仅在使用一个属性值进行查询时检索缓存。但是很显然,任何从数据库获得的永久对象都要进入缓存(在我们的改进算法中,有例外产生,但并不破坏缓存的一致性)。在复杂条件查询时,跳过缓存直接进入数据库查找,但是永久对象依然被装入缓存,然后再被返回给应用程序。

在使用可能具有重复值的属性进行查询时可以利用缓存,返回结果为该属性值等于指定值的所有对象。

1.3 基本思想

如前所述,我们的缓存是基于每一个对象类的,各个对象类的缓存之间并没有联系,本文接下来的所有内容也是基于这个上下文的。

任何一个对象在缓存中都最多允许一个拷贝,因为缓存中的永久对象实际上就是数据库中永久对象的映像。要保证这种唯一性的最简单有效的办法就是以永久对象的 oid 为键值来在缓存中保存和检索对象(缓存的基本结构应该是 Hash,这是毫无疑问的)。

但是,仅仅通过 oid 值来检索对象是不够的。oid 只是在系统的内部被使用,对于系统的使用者来说,oid 并不是查找对象时有意义的关键字。为了能使用其他一些属性检索对象,引入一个缓存视图(cache view)的概念。这些缓存视图建立在缓存之上,实际上只是用需要检索的属性值作为键值将缓存中的永久对象重新组织。这与数据库中的视图概念有相似之处。所以缓存机制是由缓存和缓存视图两个部分构成的。

与文献[6]中描述的结构有所不同的是,永久对象的多内存拷贝不再被采用。这主要是基于性能上的考虑——构造同一个对象的多个拷贝在时间和空间上都会造成很大的开销。并且,缓存机制的加入解决了文献[6]中所提到的由单拷贝造成的不得不显式维护拷贝引用计数的问题,这使多拷贝不再有必要。原因就是前面提到的,应用程序获得的永久对象始终都是从缓存中获得的,因而不用再考虑对于某一个对象,内存中是否已经有了其他的拷贝的问题。而在处理多个线程同时更新同一对象造成的冲突时,保留了 Multiversion 算法[6]——当修改对象时,将对象的版本号加 1;若版本号不一致,则说明有冲突。有所变化的是,当冲突发生时,不需要通过数据库来返回冲突信息,而是通过与缓存中的对象(如果它此时不在缓存中也会立刻被装入)的版本号进行比较。当然对缓存的加锁是不可避免的。

2 实现

2.1 缓存

本小节所指的缓存不再是广义上的缓存机制,而是以永久对象的 oid 为键值建立的用于存储对象的容器。实际上在 Java 中,容器中存储的永远是对象的引用,下面不再重复。

缓存结构被定义为 CacheHashMap,是采用 access-order 的 LinkedHashMap 的子类,以 oid 作为键值。通过覆盖 LinkedHashMap 的 removeOldestEntry( ) 方法,就构成了一个采用 LRU 算法的缓存。除此之外还定义了一个 lastRemoved 成员,用以存储上一个由于缓存已满而被淘汰的永久对象,以便随后能从缓存视图中清除这个对象。作为参考之一的 EHCache 也是建立在 LinkedHashMap 的子类上,使用这个 Java 标准类大大简化了缓存的实现[7]。

2.2 缓存视图

缓存视图也是一个容器,它只是将缓存中的永久对象根据除 oid 之外的某一属性重新组织起来。它在缓存发生变化时及时反映变化,本身并不需要具有缓存的 LRU 属性,所以简单的 HashMap 就满足要求了。但是我们定义了两个子类 UniqueCacheView 和 NotUniqueCacheView,它们并没有增加新的方法,只是起到类似于 Java 的标志接口的作用,以方便根据不同的类型进行相应的操作。

为了更清楚地描述缓存及缓存视图,我们以一个永久对象类 Company 来举例,显然,它也是 PersistetObject 的子类。图 1 至图 4 描述了这个示例。

UniqueCacheView 是以具有非重复值的属性(如公司名)为键值建立的视图。

NotUniqueCacheView 是以可能具有多个相同值的属性(如注册年份)为键值建立的视图。由于多个对象可能有相同的属性值,NotUniqueCacheView 以属性值为键值存储 NotUniqueCacheList,每个这样的列表中存储了此属性具有该值的对象。

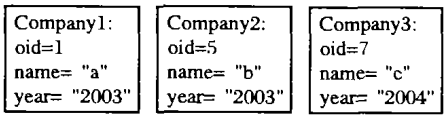

图 1 永久对象示例

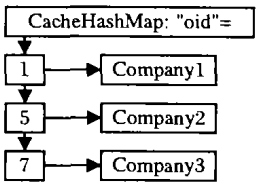

图 2 CacheHashMap 示例

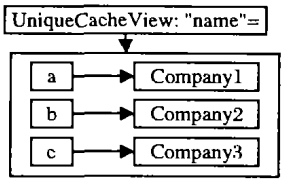

图 3 UniqueCacheView 示例

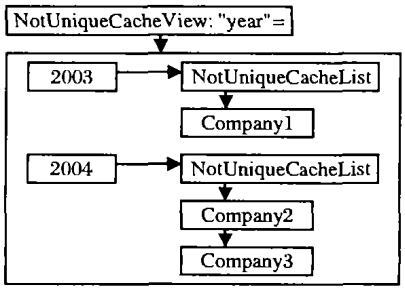

图 4 NotUniqueCacheView 示例

NotUniqueCacheList 是 LinkedList 的子类,但增加了成员和方法以检测是否具有特定值的对象全部在缓存中,只有这样才能利用缓存,否则需要进入数据库查询。

2.3 Java 反射机制(Reflection)的使用

Java 反射机制是实现我们的缓存机制的要点之一。无论是缓存的淘汰还是对象的更新,都要引发缓存视图的相应变化,此时就需要使用反射机制了——在缓存视图中,我们是以对象的属性值作为键值的,要从缓存视图中增删对象,也需要首先得到相应属性的值。

同样以前面一节的 Company 类为例,假设需要更新 Company1。首先查找 Company1 对象,如果此时它不在缓存中则会立即被装入;然后,需要首先把对象从缓存视图中清除(而不是首先更新对象本身,因为当对象的属性值发生变化时,缓存视图中原有的键值-对象映射可能已经不存在了,比如把 Company1 的 name 属性值从"c"改为"d",d- > Company1 还没有被添加,而 c- > Company1 也无法被删除了),这时就需要使用反射机制构造事先定义的 bean 方法来获得 Company1 的作为键的属性(这里是 name 和 year)的值("c"和"2003"),然后才能从缓存视图中删除旧的映射关系;随后,对象被更新(oid 是始终不变的);最后,新的映射关系被增加到缓存视图中。当然,数据库中的数据也会相应地更新。

对于缓存淘汰的情况是类似的,在对象被从缓存中删除后,也需要使用反射机制来得到作为键值的属性值,以便从缓存视图中删除映射关系。

2.4 性能测试和分析

2.4.1 测试方法和结果

我们设计了一个测试来评估这个缓存机制实现的性能。

整个测试都是在轻型对象管理器使用或不使用缓存情况下的对比测试,并没有与 EHCache 或 JCS 进行横向比较,除了缺乏有效的测试用例之外,还在于后两者并不是专门为对象-关系映射设计的,可比性并不强;而如果直接使用 Hibernate 来和我们的轻型对象管理器比较,其中又会引入对象-关系映射实现本身的差异。

为了使测试数据能反映不同条件下缓存的性能,而不依赖于某一个特定的上层应用程序,同时也是为了测试的便利,我们没有把对象管理器集成在远程支持系统中进行测试,而是编写了一个独立的 Java 程序来模拟一个应用服务器,通过产生一定量的线程随机存取永久对象来模拟用户访问,主要收集了响应时间和内存使用两项数据。

在这个虚拟的应用系统中,我们建立了 10 个永久对象类。每个对象类建立可容纳 100 个对象的缓存;每个对象类有 2 个具有非重复值的属性和 4 个具有重复值的属性,均对其建立缓存视图。我们在数据库中为每个对象类建立 400 个对象的数据。

在对象访问方面,我们设定只读查询操作占总访问量的 90%,其余为更新操作。为了模拟不同的情况,除了随机访问之外,还通过人为设定访问特征制造了不同的缓存命中率:最坏的情况下,以具有重复值的属性进行的查询占总查询数的 75%,此时缓存的命中率不到 15%;最好的情况下,所有查询均以具有非重复值的属性为键值,缓存命中率接近 80%。另外,我们还模拟了不同的负载——在测试中就是单位时间内并发访问永久对象的线程数,低负载时为 10 个/秒,中负载和高负载分别设定为低负载的 2 倍和 4 倍。

图 5 和图 6 分别显示了响应时间和内存使用的测试结果。NC、LH、RAND、HH 分别表示无缓存、缓存低命中率、对象随机访问和缓存高命中率。

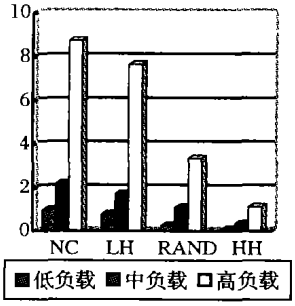

图 5　响应时间对比

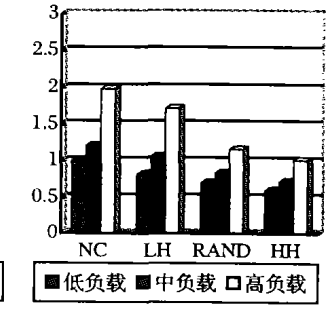

图 6　内存使用对比

2.4.2 缓存带来的性能提升

由图 5 可见,缓存机制的加入使响应时间的大幅度缩短,而在负载较高的条件下更为明显(有更多的查询从缓存获益)。在我们的测试中,缓存机制没有对性能造成任何负面影响。

不妨从结构上对缓存机制的实现做一些分析。它没有大多数缓存的通病——在大量数据交换时由于缓存频繁失效而造成二次拷贝,引起性能降低。原因就在于通常任何对象在内存中至多只有一个拷贝(特例是当应用程序还在使用一个对象时,这个对象就已经被缓存淘汰,然后又被第二次装入了,但是通常这是极少发生的;另外还有在改进算法中也产生了例外),应用程序得到的是该唯一拷贝的引用,而没有系统缓存和应用程序之间的拷贝过程;所以与文献[6]的多内存拷贝相比,既减少了对数据库的访问,也没有因为缓存失效引入额外的开销(实际上相对于文献[6]会有缓存操作,但基本上是可以忽略的),因而在最坏的情况下,性能也比没有缓存时有了提高,在高负载时平均响应时间缩短了约 12%。最好的情况下,高负载平均响应时间则相对于无缓存时缩短了约 87%。

2.4.3 缓存的效率

缓存确实带来了性能的提升,但是还有一些因素制约了缓存的效率。假设偶然出现一个以某个具有大量重复值的属性查询对象的访问,返回的大量对象可能会使缓存中原有的被频繁访问的对象也被淘汰,从而引起后续访问时的连续缓存失效。因此对这类查询,返回的对象不应进入缓存。在改进的实现中,当这类查询得到的永久对象数量超过缓存容量的 20% 时,不更新缓存。需要说明的是,之所以可以这样做,是基于我们的缓存并没有采用延迟写,因此数据库中的永久对象和缓存中的总是保持一致的。为了更彻底地解决这个问题,应用程序设计者还应在自己的永久对象类中避免针对有大量重复值的属性建立缓存视图,除了可以减少由此引发的缓存失效,还能减少不必要的缓存操作。

另一个影响缓存效率的因素是加锁策略[8]。由于任何一个对永久对象的操作都可能会引起缓存的改变,每个线程在查找或是更新对象时,都必须锁住整个缓存,势必对并发性造成影响。然而以目前缓存的结构也不可能通过减小锁粒度来加以改进。另外加锁过程本身也会有开销。在负载提高时,加锁带来的性能下降会更明显,但在我们的测试中,难于制造一个可以不加锁的特例,所以无法测定出加锁对响应时间的影响程度。不过可以肯定的是,缓存带来的性能提升显然是比加锁造成的性能损失大得多的。

2.4.4 内存使用的分析

空间方面,维护一个缓存并不一定会占用更多的内存。原因在于,Java 中释放无用对象的工作都是由垃圾收集器(Garbage Collector)来完成的,而文献[6]中的多拷贝方案非常依赖于这一机制;但是垃圾收集器并不是在对象引用数为0时马上就释放空间,大量还未来得及被释放的重复拷贝并不一定会比缓存占用的空间更少,更严重的是在一个很繁忙的系统中还会因为物理内存耗尽而引起换页操作。另一方面,即使垃圾处理器可以很快地释放空间,频繁地进行垃圾收集对 Java 虚拟机来说是有额外开销的,在高负载情况下必然引起显著的性能下降。缓存的引入,减少了给垃圾收集器造成的负担。当然在发生缓存淘汰时,还是需要垃圾收集器来回收被淘汰的对象的。

图6就是在系统相对稳定时的内存使用情况。有缓存时占用的内存发反而减少了 12% ~40%。另外,我们的缓存并没有超时淘汰的机制,在系统运行稳定后,无论负载如何,缓存占用的内存是不会变化的,负载升高引起的内存使用增加还是由于有更多的无用对象没有被及时释放造成的。从图6可以看出,缓存也使增幅有一定程度减小。最后我们尝试性地加入了缓存对象的超时淘汰,在低负载时确实减少了内存占用,但是在高负载时不仅对减少内存占用没有任何帮助,还造成了明显的性能损失。因此是否加入缓存超时淘汰机制以及如何高效地实现还有待进一步分析。

3 结语

该缓存机制的引入大大提高了系统的性能。在易用性方面,应用程序员也只需扩展 PersistentObject 类,遵循很简单的对象属性命名规则,提供相应的 Bean 方法就能构造使用缓存机制的永久对象类,并且也可以在自己的永久对象类中对缓存的使用方式进行配置。尽管该缓存机制是专为我们的远程支持系统所使用的轻型对象管理器设计的,但它可以被轻松地移植到其他类似的对象管理器中,同时,它的设计思想也能为其他的对象缓存机制的设计提供借鉴。

参考文献:

[1] TESCH T, VOLZ M. A Lightweight Object Manager for Group-Aware Applications[R]. GMD Report 47, 1999.

[2] KELLER W. Object/Relational Access Layers - a Roadmap, Missing Links and More Patterns[A]. Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing [C], 1998.

[3] KELLER W. Mapping Objects to Tables - a Pattern Language[A]. Proceedings of the EuroPLoP[C], 1997.

[4] AMBLER SW. The Design of a Robust Persistence Layer For Relational Databases. White paper[Z]. An AmbySoft Inc White Paper, October 1999.

[5] BERGLAS A. SimpleORM White Paper/Simple Java Object Relational Mapping[M/OL]. http://www.uq.net.qu/~zzaberg/simpleorm/whitepaper.html, September 2003.

[6] SONG J. Spine: A Light Persistent Object Manager[J]. Computer Science, 2004, (3).

[7] http://ehcache.sourceforge.net/documentation/[EB/OL].

[8] Patterns for Object / Relational Mapping and Access Layers[EB/OL]. http://www.objectarchitects.de/ObjectArchitects/orpatterns/index.htm.

(上接第57页)

基于关键链技术的软件项目风险管理通过对缓冲区的监控进行。关键链技术消除了每项工作的开始日期、完成日期,取而代之的是每条链的起止时间。但是我们是以每项工作的进度风险量之和设置缓冲区的大小,因此要避免各项工作的实际工作时间超出(估计时间 + 风险时间)。我们为缓冲区设置了安全底线,缓冲区的安全底线反映的是项目过程中各时刻缓冲区大小的最小值。在项目进行过程中,定时观测缓冲区的大小,若缓冲区处于安全底线以上,我们认为工作情况正常,低于安全底线,则有必要采取风险措施。

由表1,得到图4的项目缓冲区划分。如图中所示,若项目过程中观察到缓冲区处于安全底线以上的区域,则工作执行情况良好;若处于安全底线以下的区域,则有必要根据风险计划,采取相应的风险措施。

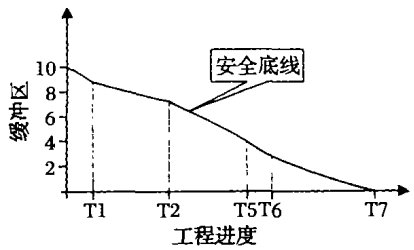

图4 缓冲区划分

5 结语

本文讨论了基于关键链的软件项目进度风险管理方法。关键链技术不仅考虑了工作间的紧前关系约束,还考虑了工作间的人力资源冲突。以理想工作条件下各个工作的执行时间建立工作节点网络图,考虑人力资源的冲突,确定关键链。在对各个工作进行风险分析的基础上,配置项目缓冲区和输入缓冲区,以消除不确定性,保证整个项目的按时完工。项目过程中,通过对缓冲区的监控和管理,实现对软件项目进度风险的管理。

但是,基于关键链的进度风险管理方法还存在一些问题。譬如,资源冲突时关键链的一般确定方法;存在多个资源约束时关键链的确定方法;基于缓冲区的进度风险的管理和监控等,这些都还有待作进一步研究。

参考文献:

[1] ROPPONEN J, LYYTINEN H. Component of Software Development Risk: How to Address Them? A Project Manager Survey[J]. IEEE Transactions on Software Engineering, 2000, 26(2).

[2] GOLDRATT EM. Critical Chain[M]. North River Press, Great Barrington, MA, 1997.

[3] PATRICK FS. Critical Chain and Risk Management - Protecting Project Value from Uncertainty[EB/OL]. http://www.focusedperformance.com/articles/ccrisk.pdf, 2001.

[4] 刘士新,宋健海,唐加福. 关键链———种项目计划与调度新方法[J]. 控制与决策,2003, 18(5).

[5] 李宁,吴之名. 网络计划技术的新发展——项目关键链管理(CCPM)[J]. 公路,2002,(10).

[6] MICHAELS JV. Technical Risk Management[M]. Prentice hall PTR, 1996.

[7] 徐哲,冯允成. 网络计划进度的风险管理[J]. 系统工程理论与实践,1998,(4).