



(12) 发明专利

(10) 授权公告号 CN 101981545 B

(45) 授权公告日 2014. 08. 20

(21) 申请号 200980111713. 2

(51) Int. Cl.

(22) 申请日 2009. 01. 30

G06F 9/46(2006. 01)

(30) 优先权数据

G06F 12/08(2006. 01)

61/025, 165 2008. 01. 31 US

(56) 对比文件

61/025, 171 2008. 01. 31 US

WO 0205134 A2, 2002. 01. 17, 说明书第 1 页  
第 15-18 行, 第 3 页第 3-4 行, 第 4 页第 25-26  
行, 第 6 页第 7-9, 19-23 行, 第 9 页第 27-31 行.

61/025, 176 2008. 01. 31 US

WO 0205134 A2, 2002. 01. 17, 说明书第 1 页  
第 15-18 行, 第 3 页第 3-4 行, 第 4 页第 25-26  
行, 第 6 页第 7-9, 19-23 行, 第 9 页第 27-31 行.

61/025, 185 2008. 01. 31 US

US 6704735 B1, 2004. 03. 09, 说明书第 8 栏  
第 6-9 行.

(85) PCT 国际申请进入国家阶段日

审查员 李伟华

2010. 09. 29

(86) PCT 国际申请的申请数据

PCT/US2009/032742 2009. 01. 30

(87) PCT 国际申请的公布数据

W02009/097586 EN 2009. 08. 06

(73) 专利权人 甲骨文国际公司

地址 美国加利福尼亚

(72) 发明人 N·雷瓦那鲁

(74) 专利代理机构 中国国际贸易促进委员会专

利商标事务所 11038

代理人 袁玥

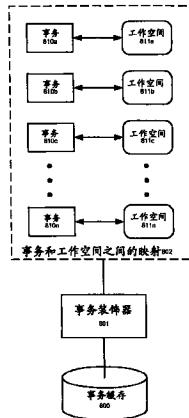
权利要求书2页 说明书28页 附图9页

(54) 发明名称

用于事务缓存的系统和方法

(57) 摘要

一种支持事务缓存服务的计算机实现的方法和系统包含:配置与一个或多个事务和一个或多个工作空间相关联的事务缓存;在事务装饰器中维持该一个或多个事务与该一个或多个工作空间之间的内部映射;获取具有一个或多个操作的事务;利用事务装饰器中的内部映射找到用于该事务的工作空间;以及向与该事务相关联的工作空间应用该事务的一个或多个操作。



1. 一种支持事务缓存服务的计算机实现的方法,包括 :

在事务装饰器处维持多个事务与多个工 作空间之间的映射,其中在所述多个事务中的至少一个被成功提交时,每个所述工作空间能够被并入事务缓存的备份数据中;

在所述事务装饰器处接收所述多个事务内的具有一个或多个操作的事务;

利用所述事务装饰器中的映射找到与该事务相关联的工作空间;以及

向与该事务相关联的工作空间应用与该事务相关联的所述一个或多个操作。

2. 根据权利要求 1 的所述方法,其中 :

使用隔离等级配置该事务缓存。

3. 根据权利要求 1 所述的方法,还包含 :

基于用于悲观事务的配置的隔离等级获得与操作的事务相关联的条目上的锁定。

4. 根据权利要求 1 所述的方法,还包含 :

记录与操作的事务相关联的条目的版本。

5. 根据权利要求 1 所述的方法,还包含 :

在用户进行提交或回滚时或在当前事务结束时发起事务完成进程。

6. 根据权利要求 5 所述的方法,其中 :

该事务完成进程使用标准同步回叫 beforeCompletion 和 afterCompletion。

7. 根据权利要求 1 所述的方法,还包含 :

将一组缓存对象分组以实现事件处理,其中,对于该组缓存对象中的相关对象自动执行缓存行为。

8. 根据权利要求 1 所述的方法,还包含 :

仅在会话结束时更新缓存对象。

9. 根据权利要求 1 所述的方法,还包含 :

从缓存管理器创建事务缓存。

10. 根据权利要求 1 所述的方法,其中 :

该事务缓存能够自加载。

11. 根据权利要求 1 所述的方法,其中 :

该事务缓存使用可插拔消息传送器来与跨越一个或多个虚拟机实例的一个或多个分布式缓存通信。

12. 根据权利要求 1 所述的方法,还包含 :

经由前向策略和 / 或监听方案,相对于所述事务缓存中的第一缓存的更新和 / 或查询而更新和 / 或查询所述事务缓存中的第二缓存。

13. 根据权利要求 1 所述的方法,还包含 :

存储一个或多个可变缓存对象,其中该一个或多个可变缓存对象中每一个是经由一个或多个缓存的对象图的一条或多条检索路径可达的;以及

维持在该一个或多个缓存的对象图与保存在缓存空间中的所述一个或多个可变缓存对象之间透明地转换的内部实例映射。

14. 一种支持事务缓存服务的计算机实现的系统,包括 :

用于在事务装饰器处维持多个事务与多个工 作空间之间的映射的装置,其中在所述多个事务中的至少一个被成功提交时,每个所述工作空间能够被并入事务缓存的备份数据

中；

用于在所述事务装饰器处接收所述多个事务内的具有一个或多个操作的事务的装置；

用于利用所述事务装饰器中的映射找到与该事务相关联的工作空间的装置；以及

用于向与该事务相关联的工作空间应用与该事务相关联的所述一个或多个操作的装置。

## 用于事务缓存的系统和方法

[0001] 版权声明

[0002] 本专利文件的一部分公开包含受著作权保护的材料。如出现在专利商标局专利记录档案中，著作权拥有人并不反对复制专利文件或专利公开内容，否则著作权拥有人保留对其的任何著作权。

[0003] 优先权声明

[0004] 本申请权利要求 2008 年 1 月 31 日提交的名为 SYSTEM AND METHOD FOR TRANSACTIONAL CACHE 的美国临时专利申请 No. 61/025, 176 ;2008 年 1 月 31 日提交的名为 SYSTEM AND METHOD FOR DISTRIBUTED CACHE 的美国临时专利申请 No. 61/025, 165 ;2008 年 1 月 31 日提交的名为 SYSTEM AND METHOD FOR TIERED CACHE 的美国临时专利申请 No. 61/025, 171 ;2008 年 1 月 31 日提交的名为 SYSTEM AND METHOD FOR MUTABLE OBJECT HANDLING 的美国临时专利申请 No. 61/025, 185 ,其公开内容通过引用结合于此。

### 技术领域

[0005] 本发明一般涉及缓存服务领域，且尤其涉及分布式缓存。

### 背景技术

[0006] 缓存是指在软件系统中为获得更高的性能而临时存储记录的拷贝。缓存由条目池 (pool of entries) 组成。每个条目具有数据 (数据块)，这些数据是一些后备存储器中数据的拷贝。每个条目还具有标签，该标签指定在后备存储器中的条目为拷贝的数据的身份。当缓存客户端 (CPU、网络浏览器、操作系统) 希望访问可推测在后备存储器中的数据时，它首先检查缓存。如果可以找到具有与所需数据的标签匹配的标签的条目，则使用该条目中的数据作为替代。这种情形已知为缓存命中。所以，例如，网络浏览器程序可以检查硬盘上的本地缓存以查看它是否具有特定 URL 的网页内容的本地拷贝。在该示例中，URL 是标签，且网页内容是数据。导致缓存命中的访问百分比已知为缓存的命中率或命中比例。在备选情形中，当查询缓存且发现不包含具有所需标签的数据时，已知为缓存缺失。在缺失处理中从后备存储器获取的数据通常插入到缓存中，准备用于下一次访问。

### 附图说明

[0007] 图 1 是根据本发明的一个实施例的缓存系统的示例性说明。

[0008] 图 2 是根据本发明的一个实施例的分层缓存的示例性说明。

[0009] 图 3 是根据本发明的一个实施例的分布式缓存的示例性框架的说明。

[0010] 图 4 是根据本发明的一个实施例的用于弱一致性复制缓存的缓存更新操作的示例性说明。

[0011] 图 5 是根据本发明的一个实施例的用于同步弱一致性复制缓存的陈旧缓存条目操作的示例性说明。

[0012] 图 6 是根据本发明的一个实施例的用于弱一致性复制缓存的针对节点关闭操作

的缓存更新的示例性说明。

[0013] 图 7 是根据本发明的一个实施例的用于强一致性复制缓存的变异运算 (mutation operation) 序列的示例性说明。

[0014] 图 8 是根据本发明的一个实施例的用于强一致性复制缓存的附属操作序列的示例性说明。

[0015] 图 9 是根据本发明的一个实施例的用于事务缓存的示例性框架的说明。

[0016] 图 10 是根据本发明的一个实施例的用于可变对象处理的示例性框架的说明。

## 具体实施方式

[0017] 通过在附图中举例而非限制的方式说明本发明，附图中相似的附图标记表示相似的元件。应当注意，本公开中的“一”或“一个”或“一些”实施例并不一定表示相同的实施例，且这些引用表示至少一个。

[0018] 缓存是指在软件系统中为获得更高的性能而临时存储记录的拷贝。可以通过减小对数据的原始记录的访问次数实现性能提升，这可能涉及创建原始记录时的数据库访问或计算密集的操作，或者它可以是对访问来说（在资源方面）昂贵的其他信息源。通过在一个进程、多个进程以及可能跨在多个机器上分布的进程中在线程上共享这种缓存对象，也可以实现性能提升。与之相连的是提供各种缓存特征——增强诸如事务、分区缓存的缓存存取特征以及诸如复制的可用性特征。

[0019] 本发明的一个实施例是可以在 Java 应用中部署和使用的用于静态和动态 Java 对象的缓存系统。尽管该缓存系统可以与比如 J2EE 容器中存在的特征相结合，本发明不强制要求 J2EE 容器的存在以使用 Java 缓存系统。

[0020] 根据一个实施例，缓存是被缓存对象的命名容器。缓存系统或缓存空间具有一个或多个这种容器。缓存可以从缓存系统重载一些属性。在一个示例中，缓存主要向用户呈现 java.util.Map 接口。

[0021] 对象明确地布置在缓存 (cache aside caches, 旁路缓存) 中或者在访问与缓存相关的键时暗示地布置在缓存中 (cache-through caches, 贯穿缓存)。在一些情况下，缓存的对象可以由特定键识别。

[0022] 缓存键是在缓存容器内用来引用缓存对象的键。键是重载针对对象定义的 hashCode( ) 和 equals( ) 方法的 Java 对象。它们还是用于支持查询 (queries) 和留存 (persistence) 的可序列化和可比较接口。java.lang.String 类型的对象被典型地使用。

[0023] 根据一个实施例，缓存键针对非本地的缓存对象是可序列化的。键的类可以重载 java.lang.Object 类的 hashCode( ) 和 equals( ) 方法。基本类型也可以用作缓存键。

[0024] 图 1 是根据本发明的一个实施例的缓存系统的示例性说明。

[0025] 根据一个实施例，区域 110a 或 110b 是缓存内的容器。它可以提供缓存 100 中的分层对象存储。区域 110a 和 110b 可以重载一些缓存属性（比如范围）。区域 110a 可以具有子区域或缓存对象 110c。

[0026] 根据一个实施例，为了实现复杂事件处理，一组缓存对象可以被分组。分组允许对相关对象自动执行某些缓存行为，诸如缓存失效。缓存失效是由于对缓存对象的更新或从缓存去除缓存对象而使得缓存对象失效的行为。

[0027] 例如,组对象的失效使得从属于该对象的所有对象都失效。类似地,在贯穿缓存中,如果被缓存的对象不存在其它依赖性,则组用于使得所有相关缓存对象无效(fault-in)。在另一示例中,组对象被限制为仅在容器缓存对象内具有依赖性。

[0028] 根据一个实施例,应用域无关缓存配置。在一个示例中,域无关缓存配置是配置分布式缓存的集群的域无关方式。另外,具有一个域范围内分布的缓存也是有用的,其可以不限制分布式缓存为集群。缓存服务还利用不同的安全措施。作为实现缓存的安全性的一个步骤,修改 API 以基于角色限制对于缓存的访问是有用的。本发明的一个实施例可以利用 OpenJPA 中的缓存实现方式。

[0029] 根据一个实施例,缓存管理器 101 是缓存服务的控制器。缓存服务通过具有动态管理和支配设备的永久配置描述。初始化时,缓存管理器 101 创建配置后的缓存且初始化它们。缓存管理器提供的生命周期服务可以类似于应用服务器中的服务器服务,但是它们被独立地定义以避免对于这种应用服务器的依赖性。

[0030] 缓存管理器 101 提供配置、管理、支配和性能监控服务。缓存管理器 101 还可以为比如许可和缓存会话的辅助服务提供支持。缓存管理器 101 的另一重要方面是作为一个整体进行容量管理和实施缓存服务的各种标准属性。

[0031] 根据一个实施例,会话可以提供视图管理和隔离。在一个示例中,一直到会话结束之前,在会话 104 内完成的对缓存对象的更新都并不传播。会话 104 还可以异常结束,意味着没有做出对缓存的其他用户可见的会话语境内的更新。会话 104 本质上可以提供缓存更新的批处理。

[0032] 除会话支持之外,缓存服务还提供事务支持以管理缓存中的对象。缓存事务通过由事务缓存暴露的方法手动控制或者通过 Java 事务 API (JTA) 自动控制。在一个示例中,事务 105 内做出的变化根据所配置的隔离等级被充分隔离。提交可以是原子的、一致的和可持久的。回滚可以被充分支持且可以不影响缓存的完整性。

[0033] 根据一个实施例,缓存映射 (Cache Map) 103 是缓存的基本接口。作为示例,缓存映射 103 是基于映射的接口。像监控和存储意识性这样的附加功能性可以添加到缓存映射 103。

[0034] 在一个示例中,在用户所关心的范围内,缓存可以仅是 java.util.Map。该 Map 从配置推导其属性。例如,如果该 Map 被配置成要被复制,则对该 Map 的所有更新被传播到集群中的其他节点。可以支持所有映射操作。缓存的典型用法如下所示:

[0035]

```
Map myCache = CacheManager.getCache("mycache");
myCache.put(mySerializableKey, mySerializableValue);
java.io.Serializable value = (Serializable) myCache.get(mySerializableKey);
// modify value
myCache.put(mySerializableKey, value);
```

[0036] 清单 -1 :典型的 CacheMap 用法

[0037] 在一些示例中,本地缓存不需要可序列化的键或值。像复制和分区缓存的其他形

式要求可序列化的缓存条目。在一个示例中,缓存 API 的用户可以通过利用缓存管理器提供的如下操作来获得缓存实例 :

[0038]

```
CacheMap getCache(String name);
```

[0039] 清单 -2 :CacheMap 查找

[0040] 在另一示例中,缓存可以通过缓存管理器 101 创建。在一个示例中,缓存管理器 101 具有下面的创建方法 :

[0041]

```
// Creates a cache with the given name and provided properties
```

```
CacheMap createCache(String name, Map properties);
```

[0042] 清单 -3 :缓存创建

[0043] 该方法允许客户传递一组名称 - 值对且基于此构建缓存实例。

[0044] 根据一个实施例,存在用于缓存管理的两个接口。CacheFactory 可用于创建系统中的新缓存。CacheManager 对于在系统中查找缓存是有用的。Cache 和 CacheEntry 是 JCache 的两个主要工件。它们扩展 Map&Map. Entry 接口以提供额外缓存相关的 API。

[0045] 根据一个实施例,可以扩展 API 以提供增强的功能性。例如,可以向缓存添加额外 API 以帮助用户获得关于该缓存的统计、改变行为且监听变化。在本发明的一个实施例中, CacheStatistics 提供缓存实例的各种统计。另外,用户可以使用 CacheListener 接口的实现来监听缓存的变化。用户还可以插入 CacheLoader 的实现来指定在缓存中没有发现被查找的键时该怎样载入缓存条目。用户还可以指定定制 EvictionStrategy 来定义如何从缓存删去缓存条目。而且, CacheException 可用于表示缓存使用中的问题和例外条件。

[0046] 根据一个实施例,缓存映射 103 是 Map 的实例。除了通过 Map 接口支持的基本 CRUD 操作之外,下面的操作对于缓存映射用户是可用的。

[0047] (1) 如果缺少,则输入

[0048] (2) 去除

[0049] (3) 替代

[0050] (4) 将键集合载入到缓存

[0051] 根据一个实施例,缓存映射 103 提供附加的操作以检索包括以下信息的监控数据 :

[0052] (1) 缓存命中 & 缺失

[0053] (2) 存储器中的元素计数

[0054] (3) 清除统计且再度开始监控

[0055] (4) 检查是否当前通过缓存装载器装载键

[0056] 根据一个实施例,也可以在单个缓存条目等级展现监控信息。然而,该信息可能不需要在跨 Java 虚拟机 (JVM) 的缓存实例之间共享。例如,可用的缓存条目等级统计是 :

[0057] (1) 创建时间

[0058] (2) 过期时间

- [0059] (3) 命中数
- [0060] (4) 最后访问时间
- [0061] (5) 最后更新时间
- [0062] (6) 当前状态 (有效或过期)
- [0063] 根据一个实施例,缓存映射 103 还展现如下管理操作 :
  - [0064] (1) 立即驱除过期元素
  - [0065] (2) 清洗所有缓存条目到持久存储器
  - [0066] (3) 在开始操作之前引导 / 初始化缓存
  - [0067] (4) 关闭缓存的本地实例,由此从内存清除元素到持久存储器
  - [0068] (5) 强制关闭缓存的所有实例。该操作的效果与本地缓存的规则关闭相同。对于分布式缓存,这还关闭该缓存的远程实例。
  - [0069] (6) 在给定缓存中锁定 / 解锁键的能力。如果该缓存是分布式的,锁定缓存中的键将给予跨分区的缓存条目排他的访问。锁操作将采用超时参数,该超时参数指定时间期以等待实现锁定。
  - [0070] (7) 注册和注销事件监听器的能力
- [0071] 根据一个实施例,缓存管理器 101 是缓存的工厂 (factory) 和注册表。缓存管理器 101 还可以支持下面的缓存管理操作 :
  - [0072] (1) 查找 :给定缓存名称,返回缓存。
  - [0073] (2) 列表 :获取所有创建的缓存的列表。
  - [0074] (3) 启动 :利用配置文件创建缓存。
  - [0075] (4) 关闭 :关闭本地 JVM 中的所有缓存。
  - [0076] (5) 强制关闭 :强制关闭本地 JVM 中的所有缓存实例。而且还关闭其他 JVM 上的实例。
  - [0077] (6) 缓存生命周期监听器 :附加缓存创建和关闭监听器。
  - [0078] (7) 缓存时间监听器 :向所有注册了且尚未创建的缓存附加监听器。
- [0079] 根据一个实施例,缓存管理器实例被创建且对于比如 WebLogic Server (WLS) 的被管理环境可用。然而,在用于不被管理的环境的另一实施例中,利用缓存管理器工厂创建缓存管理器。该工厂是用于创建缓存管理器的无状态 JVM 单例 (singleton)。
- [0080]

```
CacheManager manager =
```

```
    CacheManagerFactory.getInstance().createCacheManager();
```

- [0081] 清单 -4 :CacheManager 创建
- [0082] 根据一个实施例,客户实现 EventListener 接口以获得缓存变化通知。注意,事件监听器需要与缓存并置。缓存事件监听器可以不远程注册。
- [0083] 根据本发明的一个实施例,当底层缓存变化时生成变化事件。事件监听器可应用于下列事件。
  - [0084] (1) 输入
  - [0085] (2) 更新

- [0086] (3) 删除
- [0087] (4) 将条目调页到临时存储器
- [0088] (5) 从后备存储器载入键
- [0089] (6) 向后备存储器存储条目
- [0090] 根据一个实施例,该事件返回已经改变的一组缓存值。该组中的每个元素可包含以下信息:
  - [0091] (1) 对缓存的引用
  - [0092] (2) 键
  - [0093] (3) 改变的时间
  - [0094] (4) 更新的类型
- [0095] 根据一个实施例,在变化事件中可以不返回缓存值。可以这样做以改善性能且保留数据传输带宽。
- [0096]

```
CacheMap cache = CacheManager.getCache("mycache");
```

- [0097]

```
cache.addListener(EventListener.PUT | EventListener.UPDATE,  
myListener);  
...  
cache.removeListener(myListener);
```

- [0098] 清单 -5 :监听器注册
- [0099] 根据一个实施例,针对需要异步处理事件监听器的某些系统实现异步事件监听器。针对这种情况提供称为异步监听器的工具类。这种事件监听器可以是用于包装任意客户端事件监听器的装饰器(decorator)。
- [0100] 缓存载入器 102 是在启动或缓存缺失故障中可用于填充缓存的外部信息源。
- [0101] 根据一个实施例,缓存载入器 102 是实现 CacheLoader 接口的用户定义的类。如果配置,缓存载入器 102 被调用以在有缓存缺失时载入缓存条目。在一个示例中,缓存载入器 102 可仅在缓存创建时被配置。
- [0102] 根据一个实施例,缓存载入器 102 不做出关于调用线程的任何假设。它可以从用户线程调用或者可以异步调用。缓存载入器 102 实现下面的方法:
- [0103] (1) 给定一个键的情况下载入值
- [0104] (2) 给定键的集合的情况下载入多个值
- [0105] 根据一个实施例,存在很多方法在自加载缓存中调用缓存载入器:
- [0106] (1) 直读(Read Through):在直读自加载缓存时,当做出用于键的第一请求时调用载入器。在载入器试图从后备存储器获得条目时,该请求阻断。同时,如果其他线程做出对于相同键的请求,它们也可以被阻断。
- [0107] (2) 后读(Read Behind):如果键不在后读缓存中存在,对于 Map.get() 的调用立

即返回空。在单独的线程中，缓存载入器被调用且它试图载入值。同时，如果其他线程试图对相同的键调用 Map.get()，则它们也可以获得空值。然而，在这种情况下，不产生新的缓存加载线程。这通过维持载入的键的内部列表完成。可以利用缓存映射的监控 API 查询该列表。

[0108] 根据一个实施例，缓存存储是实现 CacheStore 接口的用户定义的类。在一个示例中，缓存存储仅在创建时配置。用户定义的类用于使改变持续，只要缓存值更新。在一个实例中，缓存存储不做出哪个线程征调用它的任何假设。它可以由用户线程调用或者可以异步调用。缓存存储实现下面的方法：

[0109] (1) 存储键值对。

[0110] (2) 存储键值对的映射。

[0111] (3) 从存储去除键。

[0112] (4) 从存储去除键的集合。

[0113] 根据一个实施例，定义写策略。有三种不同的方法可以向后备存储器写入脏条目。

[0114] (1) 直写：直写策略表示同步向后备存储器写入条目。

[0115] (2) 后写：在后写方案中，更新条目异步地写入到后备存储器。

[0116] (3) 回写：使用回写的写入策略，脏条目不在后备存储器中更新，直到它从缓存被驱除。

[0117] 根据一个实施例，当使用缓存会话时，向后备存储器的写操作被批处理。

[0118] 另一方面，驱除策略处理三种问题：

[0119] (1) 何时从缓存驱除条目？

[0120] (2) 哪些条目被驱除？

[0121] (3) 驱除到哪里？

[0122] 何时驱除条目可以由缓存的大小决定。客户设置缓存中内存中条目的数目限制。一旦到达该限制，发生驱除。用户可以设置两种限制：

[0123] (1) 内存中条目的数目的限制

[0124] (2) 内存中条目以及永久保存在磁盘上的条目的总数的限制

[0125] 要驱除的条目可以基于以下算法之一：

[0126] (1) 最久未使用 (LRU)

[0127] (2) 最不常用 (LFU)

[0128] (3) 先入先出 (FIFO)

[0129] (4) 最近未使用 (NRU)

[0130] (5) 随机驱除策略

[0131] (6) 定制驱除算法 - 在一个示例中，可被暴露以用于定制驱除的接口。客户可以实现下面的方法：

[0132] 1. 给定映射，返回键的有序列表。该顺序支配驱除的顺序。

[0133] 驱除到何处可以通过两个策略其中之一决定：

[0134] (1) 丢弃它——驱除的条目被丢弃。

[0135] (2) 调页到次级缓存。

[0136] 根据一个实施例，用户可以保持对于一个或多个缓存条目的锁定。缓存映射 103

可以具有规定来获得对其执行锁定和解锁操作的锁管理器 106。在分布式缓存的情况下，锁可以是集群范围的。所有锁与超时值一起获得。如果提供所有者，在不同的线程中获取和释放锁。锁管理器 106 提供对于共享和排他锁定的支持。在一个示例中，锁管理器 106 允许对所有者从排他锁定降级到共享锁定。

[0137] 根据一个实施例，在复制缓存的情况下锁所有者是事务且在本地缓存的情况下它可以是当前线程。另外，如果映射被复制，锁所有者是可序列化的对象。锁定缓存映射中不存在的键可以为客户保留该键。这防止其他客户使用相同的键创建条目。客户可以锁定整个映射以用于排他访问。这阻止其他客户执行关于该映射的任何操作。而且，客户可以在给定配置的时间期内保持锁定。锁可以通过锁管理器 106 强制解锁且一旦保持时间消逝，其被交付给其他等待者。

[0138] 表 1 描述了当获取和请求不同类型的锁时的各种情景。

[0139]

	请求共享锁定	请求排他锁定
实现共享锁定	是 (对于全部所有者)	否(对于全部所有者)
实现排他锁定	是 (如果所有者相同)	否
	否 (如果所有者不同)	

[0140] 表 -1

[0141] 根据一个实施例，提供的两个主要操作是：

[0142] (1) tryLocks (设置 <K> 键、对象所有者、RWLock. Mode 模式、长超时) – 以所提供的所有者和模式（共享或排他）以及超时，获取对于给定一组键的锁定。

[0143] (2) releaseLocks (设置 <K> 键、对象所有者、RWLock. Mode 模式) – 以所提供的所有者和模式（共享或排他）释放对于给定一组键的锁定。

[0144] 根据一个实施例，锁管理器 106 支持其他方便的操作：

[0145] (1) tryExclusiveLocks (设置 <K> 键) – 获取对该组键的排他锁定，无超时，以当前线程作为所有者

[0146] (2) trySharedLocks (设置 <K> 键) – 获取对该组键的共享锁定，无超时，以当前线程作为所有者

[0147] (3) releaseExclusiveLocks (设置 <K> 键) – 释放对该组键的排他锁定，以当前线程作为所有者

[0148] (4) releaseSharedLocks (设置 <K> 键) – 释放对该组键的共享锁定，以当前线程作为所有者

[0149] 清单 -6 示出了怎样锁定缓存条目的示例：

[0150]

```
LockManager lockManager = cacheMap.getLockManager();

if (lockManager != null) {
    // this map supports locking support

    if (lockManager.tryExclusiveLocks(keySubSet)) {
        try {
            // do some operations here. The current thread is the owner
            // of the locks

        } finally {
            // release all the locks

            lockManager.releaseExclusiveLocks(keySubSet)
        }
    }
}
```

[0151]

```
}
```

[0152] 清单 -6 :锁定条目

[0153] 根据一个实施例，会话 104 是基于映射的接口以用于在缓存上批处理操作。会话既可以用在事务设置又可以用在非事务设置中。在非事务设置中使用会话的主要优点可以在涉及后备存储时批处理更新操作。这在减小复制 / 分区缓存中的网络流量方面也可以是有用的。

[0154]

```
CacheMap cache = CacheManager.getCache("mycache");
CacheMap session = CacheManager.createSession(cache);
// Could have alternatively used
// CacheMap session = CacheManager.createSession("mycache");

// perform a series up update operations on the session map
session.put(mykey, myvalue);
...
session.flush();
...
session.put(mykey1, myvalue1);
...
session.flush();

// stop using session when done
```

[0155] 清单 -7 :缓存会话用法

[0156] 根据一个实施例，支持不同类型的调页：

[0157] (1) 调页到永久存储，诸如永久存储 - 本地磁盘的快速存取

[0158] (2) 文件存储

[0159] 根据本发明的一个实施例，对于上述两种临时存储其中每一个，形成可用的适配器缓存映射作为工具类。

[0160] 根据本发明的一个实施例，销住 (pinning) 键是重载驱除策略的一种方式且为应用程序员提供更好的控制。在缓存映射接口上可需要下面的 API 从而为程序员提供更好的控制。

[0161] (1) 销住键

[0162] (2) 销住键集合

[0163] (3) 拔去键

[0164] (4) 拔去键集合

[0165] 分层缓存

[0166] 图 2 是根据本发明的一个实施例的分层缓存的示例性说明。分层缓存是将两个或更多代表性缓存联系在一起且使得它们用作一个缓存的工具类。

[0167] 在如图 2 所示的双层示例中，L1 映射 201 比 L2 映射 202 更小且更快，且因而在 L2 202 之前查询 L1 201。两个缓存使用一致性同步策略同步。在该示例中，存在用于一致同步策略的两个部分。L2 映射 202 可以经由前向策略 203 相对于 L1 201 中的变化被更新。前向策略 203 的示例包括：

[0168] (1) 只读 - 禁止对更新的任何尝试。

[0169] (2) 直写 - 对 L1 201 的每次更新立即被应用于 L2 202。

[0170] (3) 调页 - 对 L1 201 的每次更新仅在 L1 201 驱除一些条目时被应用于 L2 202。

[0171] (4) 后写 - 对 L1 201 的每次更新异步地应用于 L2 202。

[0172] 另一方面, 在相同的示例中, 可以使用监听方案 204 相对于 L2 缓存 202 中的变化而更新 L1 映射 201。监听方案 204 的选项是:

[0173] (1) 不监听

[0174] (2) 监听全部

[0175] (3) 选择性监听

[0176] 根据一个实施例, 支持缓存服务的计算机实现的方法包含: 更新或查询第一缓存 201; 以及经由前向策略 203 或监听方案 204, 相对于第一缓存 201 的更新或查询, 更新或查询第二缓存 202。例如, 前向策略 203 可以包括以下至少之一: 禁止任何更新尝试; 立即向第二缓存应用第一缓存的每个更新; 仅当第一缓存驱除条目时向第二缓存应用第一缓存的更新; 以及异步地向第二缓存应用第一缓存的每个更新。监听方案 204 可以包括以下至少之一: 不监听; 监听全部; 以及选择性监听。根据本发明的一个实施例, 第一缓存 201 可以比第二缓存 202 更小且更快。

[0177] 根据一个实施例, 该计算机实现的方法可以包括保持对一个或多个缓存条目上锁的进一步的步骤。可以存在将一组缓存对象分组以实现复杂事件处理的另一进一步的步骤, 其中对该组缓存对象中的相关对象自动执行缓存动作。可以存在仅在会话结束时更新缓存对象的另一进一步的步骤。可以存在从缓存管理器 101 创建第一缓存和第二缓存的另一进一步的步骤。

[0178] 根据一个实施例, 第一缓存 201 和第二缓存 202 可以是能够自加载的。而且, 可以利用分层映射实现调页构造, 其中以客户缓存作为 L1 缓存 201 且以一个缓存映射作为 L2 缓存 221。可以利用调页一致性策略同步这两个缓存。

#### [0179] 分布式缓存

[0180] 图 3 是根据本发明的一个实施例的分布式缓存的示例性框架的说明。

[0181] 根据本发明的一个实施例, 缓存可以分类为以下广义的类别:

[0182] (1) 本地缓存 - 整个缓存可以托管在一个服务器实例中。

[0183] (2) 全复制缓存 - 每个集群成员可以托管整个缓存内容的拷贝。这有利于大多数都是读取的用途的情况。

[0184] (3) 分区缓存 - 缓存数据可以分割在多个缓存分区上, 每个分区具有缓存的一部分。每个分区可以具有一个或多个备份以支持故障转移。

[0185] 另外, 全复制缓存还基于缓存拷贝之间的数据一致性被划分为两个大类:

[0186] ● 弱一致性 - 缓存拷贝中的信息是陈旧的。弱一致性复制缓存可以使用异步机制来向其他集群成员传播缓存改变。组消息基础设施(多播或单播)可用于完成异步通信。

[0187] ● 强一致性 - 缓存信息在集群的所有成员之间都一致。可以通过对缓存条目在集群范围上锁且然后在锁定情况下向其他集群成员进行复制来实现强一致性。强一致性依赖于集群首领的存在, 该集群首领进而使用租赁基础。

[0188] 在一些情况中, 缓存类型的选择是管理选项, 而不是开发者选项。换句话说, 应用代码以完全的位置透明性继续工作。

[0189] 根据一个实施例，支持分布式缓存服务的计算机实现的系统可以包含跨越一个或多个虚拟机（VM）实例的一个或多个分布式缓存 212a、212b 和 212c；以及可插拔消息传送器 211。对于一个或多个分布式缓存 212a、212b 和 212c 其中的每一个分布式缓存，可以存在单独的缓存载入器 213a、213b 或 213c、容量控制器 214a、214b 或 214c 以及子系统管理器 215a、215b 或 215c。另外，该计算机实现的系统还可以包含一个或多个本地缓存 210。

[0190] 根据一个实施例，该计算机实现的系统中的一个或多个分布式缓存 212a、212b 和 212c 可以是全复制缓存，其中一个或多个分布式缓存的每个集群成员托管整个缓存内容的拷贝。

[0191] 在一个示例中，一个或多个分布式缓存 212a、212b 和 212c 是使用异步通信机制来向其他集群成员传播缓存变化的弱一致性全复制缓存。在另一示例中，一个或多个分布式缓存 212a、212b 和 212c 是对缓存条目使用集群范围的锁定且然后在锁定条件下复制到其他集群成员的强一致性全复制缓存。

[0192] 根据一个实施例，一个或多个分布式缓存 212a、212b 和 212c 是分区缓存，其中分区缓存可以将缓存条目分割到多个分区中且在一个或多个 VM 实例中托管该多个分区。在一个示例中，通过利用 CacheFactory 程序地创建分区缓存。在另一示例中，分区缓存是动态分区的缓存。在另一示例中，每个分区是一组键的排他所有者。在另一示例中，用于分区缓存的分区映射使用 Java 命名和目录接口（JNDI）查找，其使用分区映射的 JNDI 名称，其中 JNDI 查找返回智能代理，该智能代理保存分区路由表的拷贝且能够将缓存操作路由到正确分区。在另一示例中，分区配置成在托管在其他集群成员上的一个或多个副本上备份其数据。

[0193] 根据一个实施例，一种支持分布式缓存服务的计算机实现的方法包含以下步骤：在跨越一个或多个虚拟机（VM）实例的一个或多个分布式缓存 212a、212b 和 212c 中存储一个或多个缓存对象；以及使用可插拔消息传送器 211 维持该一个或多个分布式缓存 212a、212b 和 212c。根据本发明的一个实施例，该计算机实现的方法还可以包括在一个或多个本地缓存 210 上存储一个或多个缓存对象的另一步骤。

[0194] 根据一个实施例，实现弱一致性复制缓存。缓存映射的弱一致性副本存在于所有集群节点上。当该映射更新时，该映射使用不可靠传输（比如多播）来更新副本。当任意副本检测到映射陈旧时，它通过可靠传输与另一节点同步。对于大多数只读操作，异步地应用这种映射尺度与映射更新。当副本检测到陈旧时，它们倾向于不急切地同步。副本可以维持一致性；但是程度较弱。

[0195] 用户可以配置多于一个的弱一致性分布式映射，每个映射由唯一名称在系统中识别。用户可以使用 CacheFactory 接口查找或创建这种映射，如下：

[0196]

```
Map properties = new HashMap();
properties.put(CacheConstants.CACHE_TYPE,
CacheConstants.MULTICAST_DISTRIBUTED_MAP);
cacheManager.createCache("IncoherentCache", properties);
```

[0197] 清单 -8 :弱一致性映射创建

[0198] 弱一致性复制缓存呈现下面的功能性行为 :

[0199] (1) 初始化

[0200] (2) 缓存更新

[0201] (3) 用于同步的 HTTP

[0202] (4) 监听器

[0203] (5) 陈旧检测

[0204] (6) 从关闭节点更新传播

[0205] 根据一个实施例,所有集群节点通过多播监听缓存事件。集群节点初始化用于处理通过HTTP对等通信的特定小服务程序 (servlet)。这是当缓存检测到陈旧时使用的备份同步通信。

[0206] 当集群节点首先接收关于特定缓存的通信时,它初始化空缓存且在接收消息时更新该空缓存。该缓存还被标记为陈旧的。

[0207] 图 4 是根据本发明的一个实施例用于弱一致性复制缓存的缓存更新操作的示例性说明。

[0208] 在图 3 中示出的示例中,对于一个节点 301 上的对缓存的更新 300(输入、去除或清除操作)触发对所有节点 302、303 和 309 的消息广播 310。

[0209] 当节点 302、303 和 309 接收消息时,如果如预期一样版本升级,它们,302、303 和 309 更新其本地拷贝有效性。否则该条目被标记为陈旧的。

[0210] 循环计时器事件可以周期性触发在可靠传输上同步陈旧的缓存条目。

[0211] 在另一示例中,通过多播会话(诸如具有用于大于多播数据报大小的消息的恢复和分段的内置功能的多播会话)提供消息排序和更新的可靠性。

[0212] 根据一个实施例,在陈旧的情况下,缓存通过 HTTP 同步自身。选择 HTTP 的决定基于这一事实:当空闲时 HTTP 连接超时,且这防止 N 乘 N 连接网格。

[0213] 根据一个实施例,弱一致性映射的实现作为监控映射变化监听器提供,该监听器仅通过在监听器内包装任何映射而使得该映射是弱一致性的。

[0214] 图 5 是根据本发明的一个实施例用于弱一致性复制缓存的陈旧缓存条目的同步的示例性说明。

[0215] 根据一个实施例,缓存条目的每一次改变使用版本 id 标记。当应用更新时缓存保持跟踪版本变化。如果缓存发现它们具有最新版本,更新被丢弃。如果缓存发现它们具有应用更新的较早版本,该条目被标记为陈旧的且准备通过触发器 400 被同步。

[0216] 图 6 是根据本发明的一个实施例用于弱一致性复制缓存的对关闭节点的缓存更新的示例性说明。

[0217] 在图 5 中示出的示例中,当节点被关闭 510 时,缓存更新 500 通过可靠传输 510 发送到另一健康的集群节点 502,且目标节点 502 然后代表关闭节点 501 广播 511 变化。

[0218] 根据本发明的一个实施例,用于弱一致性复制缓存的本地缓存配置选项的行为包括:

[0219] (1) 驱除策略 - 驱除策略局部地应用于每一个节点。由于驱除导致的对于底层缓存的任何改变可以传播到所有节点。

[0220] (2) 缓存更新监听器 - 用户在任意成员上注册更新监听器且获得对复制映射做出的更新。

[0221] (3) 缓存统计是本地的 - 可以不复制关于缓存条目的统计信息。

[0222] 另外,还可以对弱一致性复制缓存实现延迟的一致性和冲突解决方案。

[0223] 根据一个实施例,在同步周期之间节点具有不一致的数据。与其他节点的及时节点同步可以解析陈旧数据。提供配置参数以配置同步间隔。

[0224] 对于多个节点上相同键的并发更新可以导致在不同节点上具有不同值的键。集群中的节点基于服务器名称、服务器身份和一些其他恒定参数被给予优先权。在所有节点上服务器的优先顺序与基于常量计算的顺序相同。在一个示例中,当节点接收用于具有相同版本的相同键的多个更新时,来自最高优先级服务器的更新获胜。最终,在所有节点中实现一致性。

[0225] 根据一个实施例,如下所示,通过利用 CacheFactory 程序地创建强一致性复制缓存:

[0226]

```
Map properties = new HashMap();
properties.put(CacheConstants.CACHE_TYPE,
```

[0227]

```
CacheConstants.COHERENT_REPLICATED_CACHE_TYPE);
cacheManager.createCache("ReplicatedCache", properties);
```

[0228] 清单 -9 :强一致性映射创建

[0229] 在一个示例中,使用下面的片段创建一致的复制映射的服务器形成一个组且保持缓存条目同步。这意味着集群中的每个服务器使用相同的名称创建一致的复制缓存类型来连接该组。未局部创建缓存的集群成员能够获得更新。

[0230] 强一致的复制缓存显示出下面的功能性行为:

[0231] (1) 利用 CacheFactory API 局部地创建缓存的服务器可以招收具有相同类型和名称的缓存的其他集群成员。

[0232] (2) 服务器可以竞争以变成复制缓存的主服务器。

[0233] (3) 一个服务器变成主服务器且其他组成员用作从服务器。

[0234] (4) 利用 Leasing API 完成主服务器选择。

[0235] (5) 在当前主服务器死掉时选择另一主服务器。

[0236] 图 7 是根据本发明的一个实施例用于强一致的复制缓存的变异操作序列的示例性说明。

[0237] 在如图 6 所示的示例中,组成员 602 对其本地映射的变异操作导致对主服务器 601 做出远程调用。主服务器 601 执行下面的操作:

[0238] (1) 获得对于映射条目的锁定

[0239] (2) 在锁定条件下将变化复制到所有组成员 603

[0240] (3) 响应请求变化的组成员 602

[0241] 在一个示例中,当主服务器 602 出现故障时,可以使用定义好的异常防止对映射的变异操作。

[0242] 图 8 是根据本发明的一个实施例用于强一致的复制缓存的附属操作序列的示例性说明。

[0243] 在图 8 所示的示例中,组成员 702 对其本地映射的附属操作导致对主服务器 701 做出远程调用。主服务器 701 执行以下操作 :

[0244] (1) 获得对映射条目的锁定

[0245] (2) 在锁定条件下反回所访问的映射条目

[0246] 在一个示例中,对映射的附属操作在缺少主服务器 701 时继续工作。组成员 702 首先试着联系主服务器 701。如果主服务器 701 不可用则返回本地映射条目。

[0247] 具有强一致的复制缓存的本地缓存配置选项的行为包括 :

[0248] (1) 驱除策略——可以支持所有本地缓存驱除策略。驱除策略可以仅在主映射上有效。主映射可以与从映射通信。

[0249] (2) 缓存更新监听器——用户可以在任意成员上注册更新监听器且获得对于复制映射作出的更新。

[0250] 根据一个实施例,实现分区缓存以包括分割在多个缓存分区上的缓存数据,其中每个分区拥有该缓存的一部分。每个分区具有一个或多个备份以支持故障转移。分区缓存分为两个类型 :

[0251] (1) 静态分区缓存——与分区相关的键可以不改变。键不能动态地从一个分区迁移到另一个分区。

[0252] (2) 动态分区缓存——在集群的生命周期中分区相关的键可以改变。

[0253] 根据一个实施例,分区缓存将缓存条目分割到多个分区且在不同的 JVM 实例上托管这些分区。当内存数据集很大且不能托管在单个 JVM 实例上时,这是有用的。

[0254] 通过使用 CacheFactory 程序地创建分区缓存,比如 :

[0255]

```
Map properties = new HashMap();
properties.put(CacheConstants.CACHE_TYPE,
CacheConstants.PARTITIONED_CACHE_TYPE);
ArrayList list = new ArrayList(4);
// create named partitions and optionally assign preferred servers to them
list.add(new Partition("p1", "managed1"));
list.add(new Partition("p2", "managed1"));
list.add(new Partition("p3", "managed2"));
list.add(new Partition("p4", "managed3"));
properties.put(CacheConstants.PARTITIONS, list);
```

[0256]

```
cacheManager.createCache("PartitionedCache", properties);
```

[0257] 清单 -10 :分区缓存创建

[0258] 在上述示例中,用户请求具有 4 个分区—p1、p2、p3、p4 的分区映射。而且,每个分区具有分配给它的优选服务器。集群尽可能尝试在优选服务器上托管这些分区。如果优选服务器不可用,则在非优选服务器上托管分区。

[0259] 用户配置用于每个分区映射的分区数目。在一些情况下,推荐多配置分区的数目以具有未来集群尺寸增长的优势。集群在其现有成员之间分布分区。如果在未来添加更多的服务器,分区可以重新分布以减小现有服务器上的负载。

[0260] 根据一个实施例,每个分区是一组键的排他所有者。整个键范围在多个分区之间划分,使得仅一个分区是任意给定键的有效所有者。

[0261] 可以以两种方式配置分区的键 :

[0262] (1) 缺省映射——缺省地,该实现方式依赖于键的哈希码以进行路由。利用哈希码和分区尺寸的简单模操作可以判断该键的托管分区。

[0263] (2) 定制映射——用户可以通过提供 KeyToPartitionLocator 接口的实现来提供定制键给分区映射。给定键,该实现方式可以返回分区。

[0264] 根据一个实施例,客户端能够使用分区映射的 JNDI 名称从 JNDI 查找分区映射。查找涉及任何集群成员。JNDI 查找可以返回智能代理,该智能代理能够将缓存操作路由到正确的分区。该智能代理保持分区路由表的拷贝,该分区路由表帮助它将调用路由到正确的分区。分区路由表可以在客户端上随分区从一个服务器迁移到另一个服务器而动态地变化。当分区路由表变化时,客户端可不需要再次查找该分区映射。

[0265] 分区被配置成在托管在其他集群成员上的一个或多个副本上备份其数据。这允许分区高度可用。以下面的方式配置副本 :

[0266] (1) 同步地复制分区数据到集群中的一个或多个备份服务器。

[0267] (2) 异步地复制分区数据到集群中的一个或多个备份服务器。

[0268] (3) 使用 (1) 和 (2) 的组合来同步地复制到“N”个成员且还异步地复制到集群中的“M”个成员。

[0269] 根据一个实施例,副本具有决定副本何时变为主副本的等级顺序。当主副本不再存在时,排行最高的存活副本可以变成下一个主副本。客户端可以在它们不能到达主副本时尝试该最高等级副本。

[0270] 根据一个实施例,分区缓存呈现以下行为 :

[0271] (1) 只要一个集群成员存活,所有分区都是有效的。

[0272] (2) 在试图向优选服务器分配分区时,分区缓存可以在集群启动期间等待一个暖机期。如果优选服务器在暖机期之后不运行,分区可以被分配到非优选服务器。

[0273] (3) 在优选服务器启动时,分配给非优选服务器的分区可以不被分配回优选服务器,当需要再次平衡分区时,给予优选服务器以优先权。

[0274] 根据一个实施例,分区向服务器的分配按以下方式完成 :

[0275] (1) 分区可以分配给优选服务器

- [0276] (2) 其余分区可以均匀地分布在集群上
- [0277] (3) 当新成员添加到集群时可以重新平衡分区。
- [0278] 根据一个实施例，重新平衡按以下方式发生：
- [0279] (1) 新启动的服务器可以试图托管指定它们为优选主机的分区。
- [0280] (2) 在第二步骤，可以重新平衡分区，使得运行服务器将试图托管相等数目的分区。
- [0281] (3) 当服务器关闭后，分配给它的分区一律迁移到现有服务器。
- [0282] 在一个实施例中，分区映射客户端可能不需要知道分区的物理位置来访问缓存条目。客户端可以获得完全的位置透明度。仅可序列化的条目被放置在分区映射中。
- [0283] 根据本发明的一个实施例，具有分区缓存的本地缓存配置选项的行为包括：
- [0284] (1) 驱除策略——可以支持所有本地缓存驱除策略。每个分区将独立于它拥有的键集而应用驱除策略。
- [0285] (2) 缓存更新监听器——用户可以在分区映射上注册更新监听器且获得来自正确分区的更新。
- [0286] 根据一个实施例，全复制缓存将其更新传播到所有存活的集群成员。可选地，客户可以指定将托管该复制缓存的服务器列表。候选列表是整个集群的子集。这类似于单例服务或可迁移目标的候选列表。
- [0287] 在一个示例中，可能需要特别注意应用范围的复制缓存。如果应用同质地部署在集群中，则整个集群可以参与该复制缓存。但是在应用并非同质地部署的特殊情况中，复制可以限于已经部署了应用的集群成员。
- [0288] 事务缓存
- [0289] 根据一个实施例，当创建一个缓存时，它是事务性的。CacheFactory 提供基于属性 Map 来创建缓存的方法。该属性 Map 包括以下与事务相关的属性：
- [0290] (1) 要使用的事务的类型——悲观或乐观。
- [0291] (2) 事务隔离等级。
- [0292] (3) 描述是否使用手工事务或 JTA 集成以及怎样与 JTATransactionManager 集成的属性
- [0293] 清单 11 示出示例：
- [0294]

```
Map properties = new HashMap();
properties.put(CacheConstants.TRANSACTION_TYPE,
CacheConstants.PESSIMISTIC);
properties.put(CacheConstants.ISOLATION_LEVEL,
CacheConstants.REPEATABLE_READ);
Map transCache = cacheManager.createCache("TransactionalCache",
properties);
```

[0295] 清单 -11 :事务缓存创建

[0296] 根据一个实施例,该缓存支持具有如下标准隔离等级的本地悲观事务 :

[0297] (1)NONE

[0298] (2)READ\_UNCOMMITTED

[0299] (3)READ\_COMMITTED

[0300] (4)REPEATABLE\_READ

[0301] 如果隔离等级设置为 NONE,则缓存不是事务性的。而是,内部缓存被包装在事务装饰器中,使用装饰器模式提供对所关心内容的强隔离,允许缓存保持集中于基本数据存储和检索,而装饰器处理事务性行为。装饰器模式的实现细节可以隐藏到外部代码。

[0302] 根据一个实施例,缺省悲观事务隔离可以是 REPEATABLE\_READ。利用读锁和写锁的组合实现事务隔离等级以隔离变化。

[0303] 根据本发明的另一实施例,尽管悲观事务被设置为缺省,缓存还支持本地乐观事务。所有乐观事务具有 NONE 或 REPEATABLE\_READ 隔离;其他配置的隔离等级可以自动升级为 REPEATABLE\_READ。

[0304] 乐观事务的动机是改善并发性。当存在对缓存数据的严重争用时,在延长的时间期中锁定部分缓存是不够的,而悲观事务常常如此。乐观事务通过利用数据版本控制代替锁定允许更大的并发性。

[0305] 数据版本控制通过在每个缓存条目贯穿事务被访问时记录其版本而工作。当事务提交时,每个条目的记录版本与缓存中条目的当前版本进行比较。在第一事务记录条目版本之后但在第一事务提交之前,如果并发事务变更条目 - 由此增加其版本,则在版本失配的情况下,提交失败。

[0306] 乐观事务使用悲观事务中使用的相同的读锁和写锁,但是锁定仅保持极短的时间期:当条目被检索时以及当提交时事务将其隔离的变化融入回主缓存时。所以尽管乐观事务可以在版本验证失败时偶尔强制回滚或者由于记录数据版本和在提交时验证造成的不可避免的开销而可能比悲观锁定运行得稍慢,但它们在维持极高度并发的同时允许高度隔离。

[0307] 图 9 是根据本发明的一个实施例用于事务缓存的示例性框架的说明。

[0308] 本发明的一个实施例是支持事务缓存服务的计算机实现的方法,包含:配置与一个或多个事务 810a-n 和一个或多个空间 811a-n 相关联的事务缓存;在事务装饰器 801 内维持一个或多个事务 810a-n 与一个或多个空间 811a-n 之间的内部映射 802;获得具有一个或多个操作的事务 810a;利用事务装饰器 801 中的内部映射 802 找到用于该事务的工作空间 811a;以及向与事务 810a 相关联的工作空间 811a 应用事务 810a 的一个或多个操作。根据本发明的一个实施例,使用隔离等级配置事务缓存。

[0309] 根据一个实施例,该计算机实现的方法包含创建内部映射 802 的另一步骤。可以存在基于用于悲观事务的所配置的隔离等级获得对受影响条目的锁定的另一步骤。可以存在记录受影响条目的版本的另一步骤。可以存在在用户命令提交或回滚时或者在当前事务结束时发起事务完成进程的另一步骤。在一个示例中,事务完成进程使用标准同步回叫: beforeCompletion 和 afterCompletion。

[0310] 根据一个实施例,该计算机实现的方法包含将一组缓存对象分组以实现复制事务

处理的另一步骤，其中，对于该组缓存对象中的相关对象自动执行缓存动作。可以存在仅在会话结束时更新缓存对象的另一步骤。另外，该计算机实现的方法可以包含从缓存管理器创建事务缓存的另一步骤。而且，该事务缓存能够自加载且提供 XA 事务支持。

[0311] 根据一个实施例，事务缓存作为单一 JVM 映射出现，就像非事务缓存一样。然而当在不同事务语境中访问时，事务缓存可以呈现隔离视图。它可以通过维持每个事务和其改变的数据或其工作空间之间的内部映射来完成。在一些示例中，工作空间仅可以在成功提交时并入缓存的备份数据。

[0312] 根据一个实施例，这种用法模式最大化对用户的透明度且允许我们自然地对事务强加严格的线程关联性。对于实现事务隔离所必须的锁定通常需要将请求锁定的客户端与客户端可能已经持有的任何锁进行匹配，且通过线程识别客户端提供这样做的简单方式。没有线程关联性，我们将必须通过将客户端标识符或者已经持有的锁传送到所有映射操作来复杂化我们的整个 API 栈。

[0313] 根据一个实施例，事务装饰器向标准 CacheMap 添加如下主要 API：

[0314]

```
package weblogic.cache.transaction;  
/**  
 * A {@link CacheMap} with transactional capabilities.  
 */  
  
public interface TransactionalCacheMap  
    extends CacheMap {  
    /**  
     * The transaction for the calling context.  
     */  
  
    public CacheTransaction getTransaction();  
    /**  
     * Immutable view of the transactional workspace for  
     * the calling context.  
     */  
  
    public Workspace getWorkspace();  
    /**  
     * Set a value update for the given key. During  
     * transactions, value updates are buffered until  
     * commit.  
     * @return the previously-buffered update for this key  
     */  
  
    public ValueUpdate setValueUpdate(Object key,  
                                     ValueUpdate update);
```

[0315]

```
}

/**
 * Current transaction.
 */

public interface CacheTransaction {
    /**
     * Whether a transaction is in progress.
     */

    public boolean isActive();

    /**
     * Begin a transaction.
     */

    public void begin();

    /**
     * Commit the current transaction.
     */

    public void commit();

    /**
     * Rollback the current transaction.
     */

    public void rollback();

    /**
     * Mark the current transaction for rollback.
     */

    public void setRollbackOnly();
}
```

[0316] 清单 -12 : 事务映射 API

[0317] 根据一个实施例, 对事务缓存的每个操作导致下面的一系列事件 :

[0318] (1) 获取用于当前线程的事务。如果没有正在进行的事务, 则将操作传递到底层缓存。否则 :

[0319] (2) 使用事务和工作空间之间的事务装饰器的内部映射来找到事务的工作空间。如果不存在, 创建一个。

[0320] (3) 如果是悲观事务, 基于配置的隔离等级获得对受影响条目的锁定。如果是乐观事务, 记录受影响条目的版本。

[0321] (4) 向工作空间应用操作。

[0322] 根据一个实施例, 当用户命令提交或回滚时或者在当前 JTA 事务结束时, 可以发起事务完成进程。缓存事务完成使用标准同步回叫 :beforeCompletion 和 afterCompletion。在 beforeCompletion 中, 工作空间变化被批处理到 action——对于缓存执行一系列操作的对象。这种行为具有两个阶段。Prepare 阶段获得受到所配置的事务隔离的所有修改条目上的写锁。如果事务是乐观的, 该阶段还相对于事务的记录版本验证所有条目的版本。事务变化然后被应用于底层缓存, 且之前的状态被保存以用于回滚。Run 阶段在必要时执行回滚。在一个示例中, beforeCompletion 回叫仅执行 prepare 阶段。

[0323] 根据一个实施例, 在 afterCompletion 回叫中采取的步骤依赖于事务是被回滚还是提交。如果事务被回滚, 则 afterCompletion 执行在 beforeCompletion 中准备的行为的 run 阶段以回滚缓存变化。然而如果事务提交, 则行为简单地关闭。在两种情况下, 所有锁都被释放且事务工作空间被清除。

[0324] 因而, 实现 ACID 事务通过工作空间、读 - 写锁和动作对象的组合达成。事务装饰器维持工作空间且定义动作。然而实现读 - 写锁和运行动作可以是底层缓存的功能。

[0325] 逻辑上, 每一个缓存条目可以具有支持多个读取器的读锁, 以及排他写锁既不允许读取器也不允许记录器。实际上, 缓存可以锁定区域或其他粗粒度结构而不是各个条目。读锁可升级成写锁。锁所有权是基于线程的, 且请求当前线程已经保持的锁是没有效果的。即使在缺乏事务的情况下, 读取条目可以隐含地获取并立即释放其读锁, 且改变条目获取且立即释放其写锁。对于本地事务, 仅需要本地锁。为了附加地支持在集群上的完全 ACID 事务, 需要集群范围的锁。然而重要的是, 不需要对事务装饰器做出改变。在一些示例中, 集群式缓存可能仅需要实现其锁 API 来使用集群范围的锁而不是本地锁。

[0326] Action 用于批处理操作。动作可以被给予对缓存的引用且可以执行它希望的任意系列的操作。动作是可序列化的, 使得它们可以以单程被发送以在单独的 JVM 中的缓存上运行, 而不招致很多各远程呼叫的开销。

[0327]

```
package weblogic.cache;

/**
 * A bulk operation.
 */

public interface Action
    extends Serializable {

    /**
     * The system will set the target cache before the
     * action is used.
     */

    public void setTarget(CacheMap cache);

    /**
     * Perform the action.
     */

    public Object run(Object arg);

    /**
     * Clean up the resources held by this action. This
     * method is automatically called if any method throws
     * an exception, or after {@link #run}. It can also be
     */
}
```

[0328]

```
* manually invoked via the action's trigger.

}

public void close();
```

[0329] 清单 -13 :Action 接口

[0330] 运行一个动作是一个两阶段进程。要求缓存准备一个动作返回一触发,该触发可用于运行或关闭所准备的动作。

[0331]

```

package weblogic.cache;

public interface CacheMap ... {

    public ActionTrigger prepare(Action action);

    ...

}

/***
 * Trigger or close an action.
 */

public interface ActionTrigger {

    /***
     * Run the action, returning the result.
     */

    public Object run(Object arg);

    /***
     * Manually close the action. Actions are closed
     * automatically if they throw an exception or after
     * they are run.
     */

    public void close();
}

```

[0332] 清单 -14 :ActionTrigger 接口

[0333] 具有单独准备阶段的价值是双重的。首先,它允许多步骤进程的封装和简单取消,该多步骤进程中,在诸如上述 beforeCompletion、afterCompletion 提交进程的步骤之间保存资源。而且,更重要地,正确实现的准备阶段增加了动作可以完全成功或安全失败的可能性。这在集群式缓存中尤其有用。例如,同步复制集群式缓存使用准备阶段来确保动作在试图运行之前在每个节点中被成功地发送和准备。像集群式锁定一样,这对事务装饰器是透明的。

[0334] 可变对象处理

[0335] 被缓存对象一般对于缓存服务是不透明的。在一些情况中,对其内部状态和关系的变化可能不被检测,对象身份可以不被保留,且它们可以经由序列化在 JVM 边界上传输。

[0336] 这种不透明性对于不可变类型可以是可接受的,但是对于具有可变状态和关系的类型引入了编程困难和性能瓶颈。首先,它强制用户微观管理缓存:只要对象更新,用户就可以将该变化告知缓存。对象更新还可以不服从所配置的事务隔离等级:对于实例的内部

状态的改变可以是对所有本地缓存客户可见的。在性能前端，序列化的使用可能极其低效。对于大对象或大对象图中的节点，即使单个字段的更新也可能触发整个对象或者整个图的序列化以用于集群上的复制。最后，Java 对象序列化不能保留对象身份。相同远程缓存的对象的多次查找可以返回不同的 JVM 实例，且在多个键（可能作为多个对象图的一部分）下缓存相同的对象可导致在检索时用于每个键的不同的 JVM 实例。

[0337] 根据一个实施例，用户可以配置用于可变对象处理的任意或全部的他们的定制类。可变对象是其内部状态可以由用户改变的对象。这包括被识别为经由配置可变的用户定义的类的实例以及诸如 collection、map 和 date 的标准可变类型的实例。在可变对象处理中，对于实例的内部状态和关系的改变被自动地检测。对象身份可以在多个缓存键和多次查找中保留。在一些情况中，可变对象可以不被序列化。除了所配置的用户定义的类，缓存系统作为可变对象处理的很多标准类型，包括 collection、map 和 date。

[0338] 根据一个实施例，缓存系统包括用于缓存值的专用可变对象处理。诸如 collection、map 和 date 的标准可变类型的实例可以自动被处理为可变对象。用户还可以设定他的用于可变对象处理的任意或所有类。

[0339] 根据一个实施例，在可变对象处理下，可变对象存储在多个缓存对象图中且经由相同对象图中的多个路径可达。相同可变对象的检索总是返回相同的 JVM 实例，而不管用于导航到对象的检索路径 - 缓存键和关系 - 遍历如何。无论对象是图根或者缓存对象图中的任意节点，对可变对象的改变，包括可变对象之间的关系的变化，被自动检测。这些变化还可以根据配置的事务语义被隔离。最后，当一次缓存的可变对象不再从任何缓存的对象图引用时，该缓存可以检测出它不再被需要且要求归还它占用的任意空间。另外，在本发明的一个实施例中可以支持作为缓存键的可变对象。

[0340] 根据一个实施例，可变对象处理对于外部代码是透明的。像其他缓存一样，可变处理缓存的行为一般表现得像扩展映射那样，且其包含的可变对象图是从正常 Java 对象的图中不可区分的。对于这种透明性的唯一主要例外是事务隔离。当缓存的实例在事务内变化时，变化可以仅对于该事务的语境可见。仅当事务成功地提交时，变化才对其他缓存客户可见。这通过向可变对象状态应用在事务缓存中使用的相同工作空间模式实现。即，事务可变对象可以从该事务的隔离工作空间而不是它们自己的实例字段汲取其状态。

[0341] 根据一个实施例，所描述的用法模式在维持正确的事务语义的同时最大化对用户的透明度。用户可以正常地操作可变对象图且可以在线程之间共享它们，尽管事务线程中做出的变化直到提交时对于其他线程都可以是不可见的。工作空间模式的使用最小化排他锁定。没有它，即使在乐观事务内，事务对象在修改时立即被排他地锁定以防止脏读。

[0342] 根据一个实施例，可变对象处理被实现为会话或事务缓存之上的装饰器。缓存保持集中于基本数据存储和检索以及工作空间维持，而可变处理装饰器关注于上面列举的需求。为此，可变处理装饰器透明地在用户可见的对象图和平坦缓存空间之间转换，且维持内部实例映射。该实例映射是分配给可变对象（参加下文）的唯一 id 和为该对象实例化的 JVM 实例之间的存储器敏感的映射。该实例映射负责维持查找和对象图上的可变对象身份。

[0343] 图 10 是根据本发明的一个实施例用于可变对象处理的示例性框架的说明。

[0344] 本发明的一个实施例是支持可变对象处理的计算机实现的系统其包含：缓存空间 900，能够存储一个或多个可变对象；一个或多个缓存的对象图 902、903、904、905，其中该

一个或多个可变对象其中每一个经由一个或多个缓存的对象图 902、903、904 和 905 的一个或多个检索路径 912a、912b、912c 和 912d；913a、913b、913c 和 913d；914a、914b、914c 和 914d；以及 915a、915b、915c 和 915d 可达；以及可变处理装饰器 901，其维持在一个或多个缓存对象图 902、903、904 和 905 以及存储在缓存空间中的一个或多个可变缓存对象之间透明地转变的内部实例映射 910。

[0345] 根据一个实施例，缓存空间 900 可以包括一个或多个缓存。根据本发明的一个实施例，该一个或多个缓存中的一个缓存是会话或事务缓存。根据本发明的一个实施例，该一个或多个缓存的对象图 902、903、904 和 905 对用户可见。

[0346] 根据一个实施例，实例映射 901 是分配给可变缓存对象的唯一 id 和为可变缓存对象发起的 Java 虚拟机 (JVM) 实例之间的存储器敏感的映射。

[0347] 根据一个实施例，一个或多个检索路径 912a、912b、912c 和 912d；913a、913b、913c 和 913d；914a、914b、914c 和 914d；以及 915a、915b、915c 和 915d 包括缓存键和关系遍历。可变处理装饰器 901 自动检测可变对象之间的关系的变化。可变对象作为不可变状态的结构存储在缓存空间中，包括不可变字段值和相关可变值的 id。钩子用于构建用户定义的类的实例且向 / 从缓存的结构传递其状态。而且，可变对象图中存在的标准可变类型的实例在清洗时被子类的实例代替。

[0348] 根据一个实施例，支持可变对象处理的计算机实现的方法包含：维持在一个或多个缓存的对象图 902、903、904 和 905 与存储在缓存空间中的一个或多个缓存对象之间透明转换的可变处理装饰器 901 中的内部实例映射 910；以及从一个或多个缓存的对象图到达可变对象，其中每个缓存的对象图包括一个或多个检索路径 912a、912b、912c 和 912d；913a、913b、913c 和 913d；914a、914b、914c 和 914d；以及 915a、915b、915c 和 915d。

[0349] 根据一个实施例，还包括以下步骤：通过一个或多个缓存的对象图 902、903、904 和 905 在缓存空间 900 中存储可变对象；以及通过一个或多个缓存的对象图 902、903、904 和 905 从缓存空间检索可变对象。

[0350] 当用户添加缓存条目 (K, V) 时，其中 V 是可变对象，可变处理装饰器可以在清洗时执行以下步骤：

[0351] (1) 为 V 分配全局唯一 id。

[0352] (2) 在装饰器的内部实例映射中添加将该 id 映射到 V 的条目。

[0353] (3) 创建结构以代表缓存中的 V。该结构保存 V 的不可变字段状态、V 的引用计数（见步骤 5）以及 V 的关系的唯一 id（见步骤 6）。

[0354] (4) 将 V 结构添加到缓存，以 V 的 id 为键。

[0355] (5) 设置 V 的初始引用计数为 1。

[0356] (6) 遍历从 V 可达的整个对象图递归地向 V 的关系应用步骤 1-5。当我们发现图中对象之间的交叉关联时适当地递增引用计数。

[0357] (7) 添加映射 K 到 V 的 id 的缓存条目。

[0358] 根据一个实施例，当用户检索键 K 时，装饰器可以进行如下处理：

[0359] (1) 检查以查看 K 的值是否是可变对象的 id。如果不是，直接返回该值，否则：

[0360] (2) 检查实例映射以查看是否已经为匹配该 id 的结构构建了 JVM 实例。如果是，则返回该实例。否则：

[0361] (3) 找到缓存中用于给定 id 的结构, 为该结构构建正确的用户可见类的 JVM 实例, 且使用该结构的所有存储的数据填充它。将映射该 id 到该 JVM 实例的条目添加到该实例映射中。

[0362] (4) 递归地向保存在该结构中的关系的 id 应用此过程, 使得重新组建整个对象图。这种递归处理可以立即执行, 或者可以在用户遍历关系时慢慢完成。

[0363] (5) 返回构建的实例。

[0364] 根据一个实施例, 可变对象不作为其原始类的实例而是作为不可变状态的结构被存储在缓存中, 该不可变状态的结构包括不可变字段值和相关可变值的 id。每个这种结构 - 以及因此每一个可变对象 - 可以通过唯一 id 识别, 使得图可以被重建为引用正确的实例, 包括对象图内和对象图之间的共享引用。结构可以独立地关于其 id 被缓存在平坦缓存空间内以用于简单的查找。可变处理装饰器可以从用户隐藏这些缓存条目。引用数可用于判断结构何时不再被任何值图引用且可以从缓存安全去除。

[0365] 根据一个实施例, 构建用户定义的类的实例且从 / 向被缓存结构传递其状态可以通过添加到类字节码的钩子实现。附加的钩子可用于检测对于实例字段的访问。这可允许可变处理装饰器跟踪脏字段且在事务语境中访问实例时重定向状态访问到事务工作空间。所需的字节码操作可以在建设进程期间执行或者在类加载时动态地执行。

[0366] 根据一个实施例, 诸如列表、映射和日期的标准可变类的字节码可以不修改。但是, 如有需要可以通过字节码产生子类 (例如 CacheableArrayList, CacheableHashMap)。作为修改的用户定义的类, 这些子类可以呈现相同的脏跟踪和状态截取行为。对象图中存在的标准可变类型的实例可以在清洗时使用这些子类的实例代替。因而, 清洗代表一个边界, 该边界之后, 对于标准可变对象保持的引用可变得无效。注意, 这仅应用于新的, 即缓存的标准可变的实例, 从缓存对象图中检索的实例可以总是已经可缓存的子类的实例。

[0367] 根据一个实施例, 如果底层缓存是会话, 对于事务内的 beforeCompletion 或者各个更新, 可以明确地发生清洗。清除可以执行以下步骤:

[0368] (1) 基于会话、事务或各个更新中的关系的再分配计算引用计数的变化。

[0369] (2) 如上所述, 使用可缓存的子类实例代替对新缓存的标准可变类型的引用。

[0370] (3) 清洗引用计数变化以及底层工作空间的明显脏可变状态。

[0371] (4) 如果任何结构引用计数达到零, 则计划去除结构的条目。

[0372] (5) 底层会话或事务缓存可以处理与特定缓存相关的其余操作。

[0373] 根据一个实施例, Java 注释可用于允许用户识别其可变对象类型。另外, 配置文件构造可以允许用户识别其可变对象类型。

[0374] 一个实施例包括计算机程序产品, 该计算机程序产品是存储介质, 该存储介质上具有存储的指令 / 该指令可用于对计算机进行编程以执行此处提出的任意特征。存储介质可以包括但不限于任何类型的磁盘, 包括软盘、光盘、DVD、CD-ROM、微硬盘以及磁光盘、ROM、RAM、EPROM、EEPROM、DRAM、闪存介质或者适于存储计算机可读介质其中任意一个上存储的指令和 / 或数据的设备, 本发明可以包括用于控制通用目的 / 专用计算机或微处理器的硬件且使得计算机或微处理器能够与人类用户交互的软件或者利用本发明的结果的其他机制。这种软件可以包括但不限于设备驱动器、操作系统、执行环境 / 容器和用户应用。

[0375] 本发明的实施例可以包括提供用于实现本发明的处理的代码。这种提供可以包括

以任意方式向用户提供代码。例如,这种提供可以包括向用户发射包含代码的数字信号;向用户提供物理介质上的代码;或者使得代码可用的任意其他方法。

[0376] 本发明的实施例可以包括用于发射可以在计算机处执行的代码以执行本发明的实施例的任意处理的计算机实现的方法。这种发射可以包括通过诸如因特网的网络的任意部分、通过布线、空气或空间的传输、或者任意其他类型的传输。这种传输可以包括发起代码的传输;或者导致代码从任何区域或国家传输到另一区域或国家。例如,传输包括导致作为原先寻址的结果通过一部分网络传送代码以及向用户发送包括代码的数据。向用户的传输可以包括在任何区域或国家中由用户接收的任何传输,不管发送传输的位置如何。

[0377] 本发明的实施例可以包括包含代码的信号,该代码可以在计算机执行的代码以执行本发明的实施例的任意处理。该信号可以通过诸如因特网的网络;通过布线、空气或空间;或者任意其他类型的传输发射。整个信号不需要同时被传播。信号可以在其传送期上扩展时间。该信号不被认为是当前被传播的快照。

[0378] 用于说明和描述目的,已经提供了本发明的优选实施例的上述描述。其并不旨在是排他性的或将本发明限制于公开的特定形式。相关技术领域的技术人员显见很多修改和变型。例如,公开的本发明的实施例中执行的步骤可以以备选顺序执行,可以省略某些步骤,且可以添加附加步骤。选择和描述的实施例是为了更好地解释本发明的原理及其实际应用,由此使得本领域技术人员能够理解本发明的各种实施例以及尤其适于预期使用的各种修改。旨在表明,本发明的范围由所附权利要求及其等价物限定。

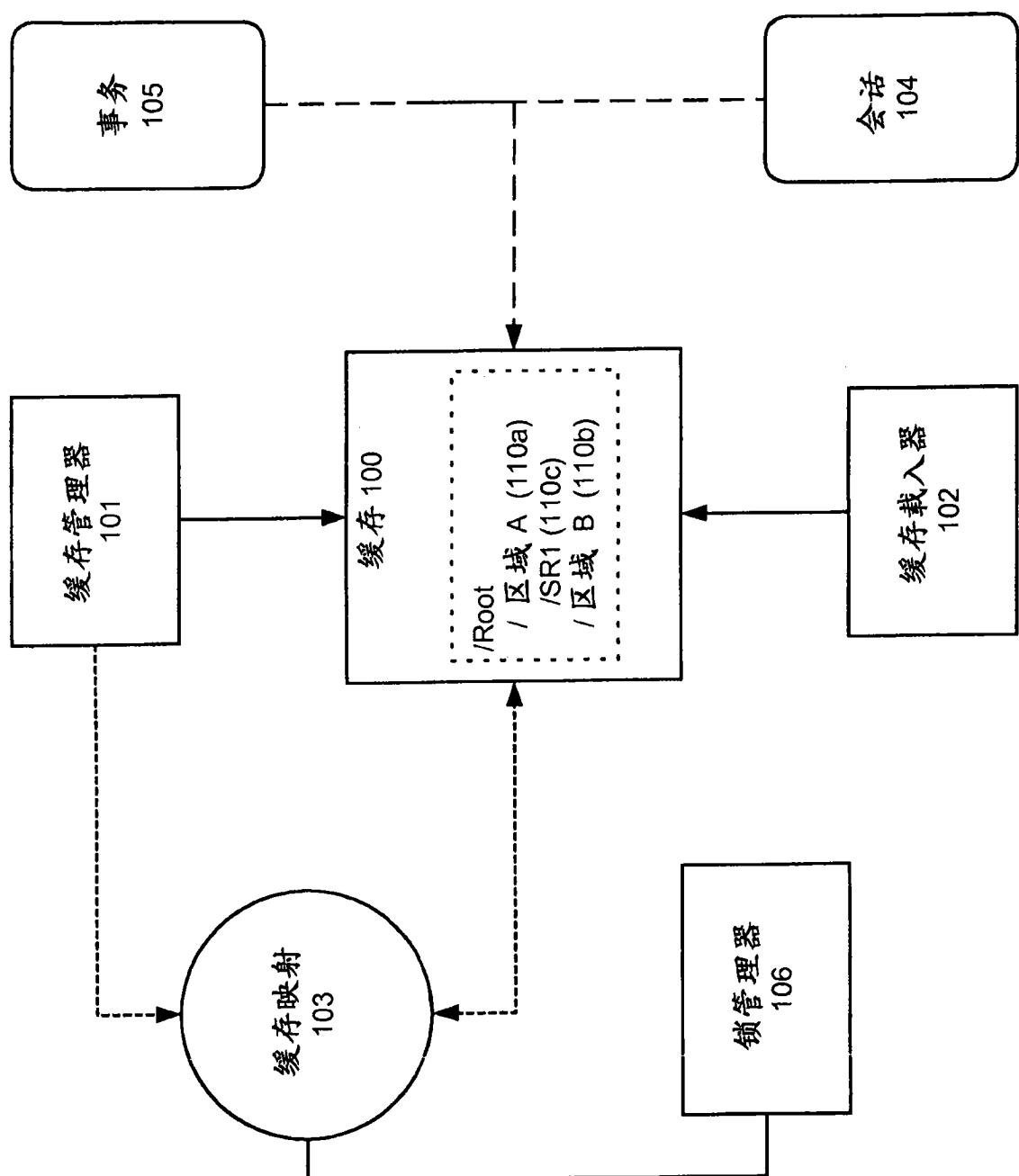


图 1

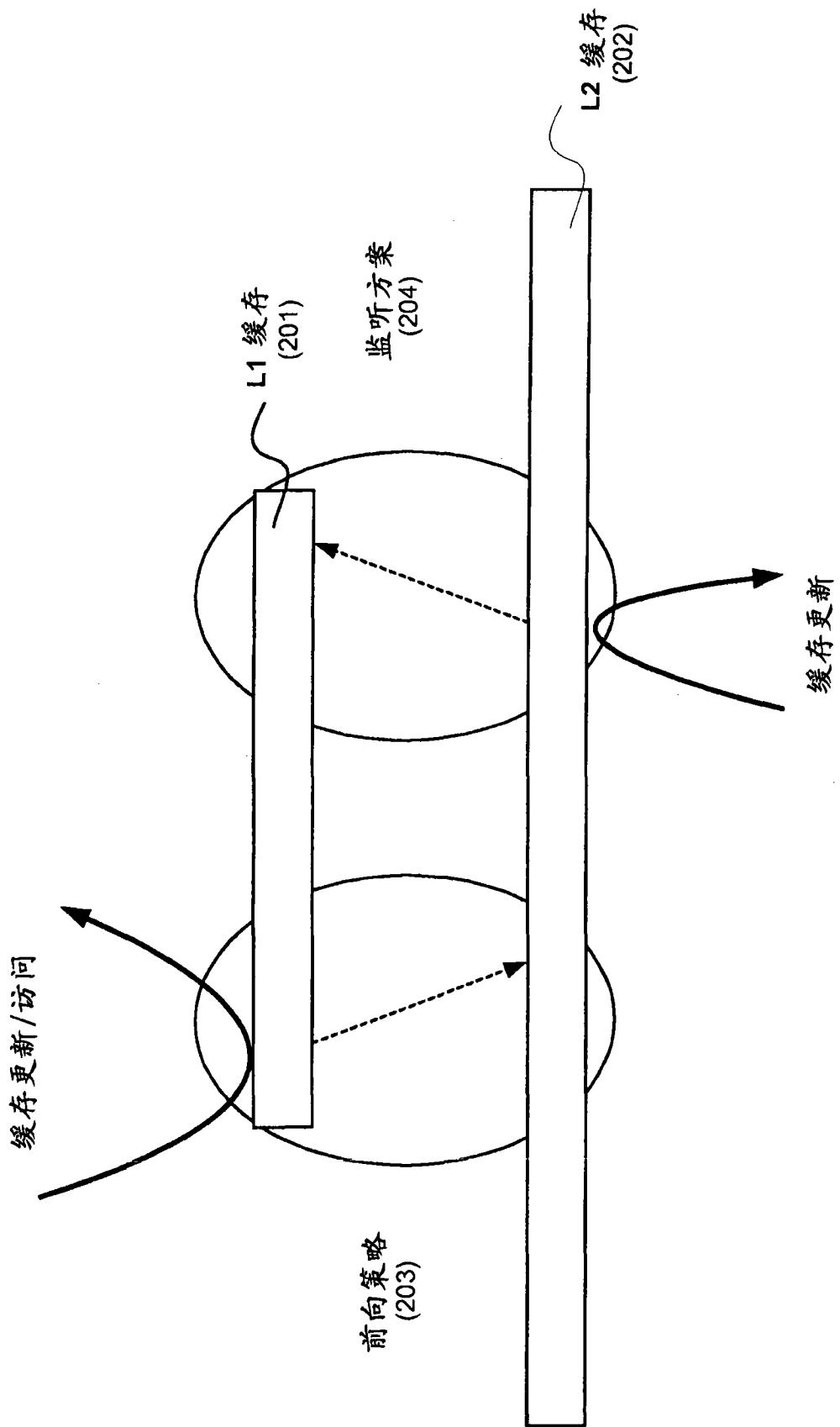


图 2

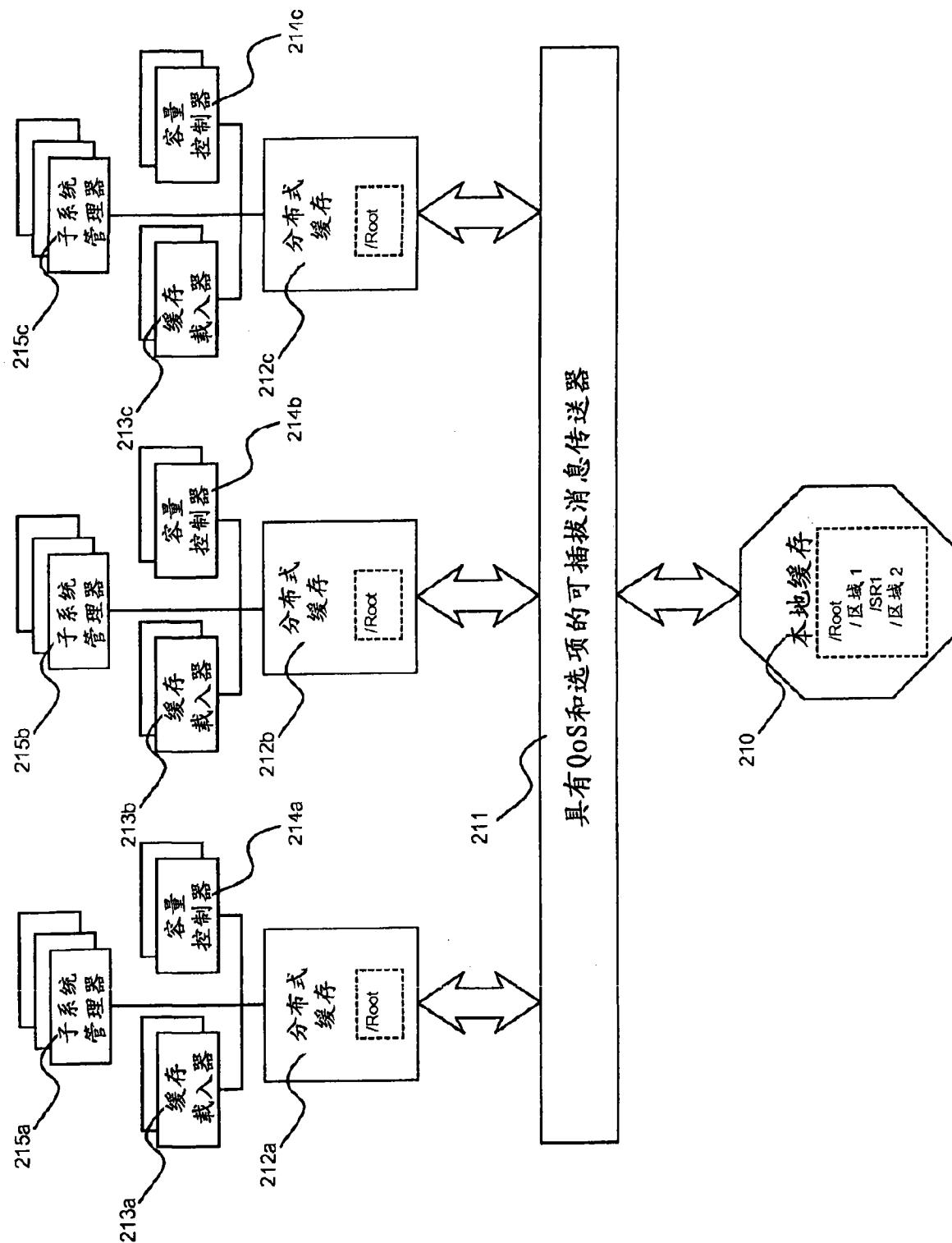


图 3

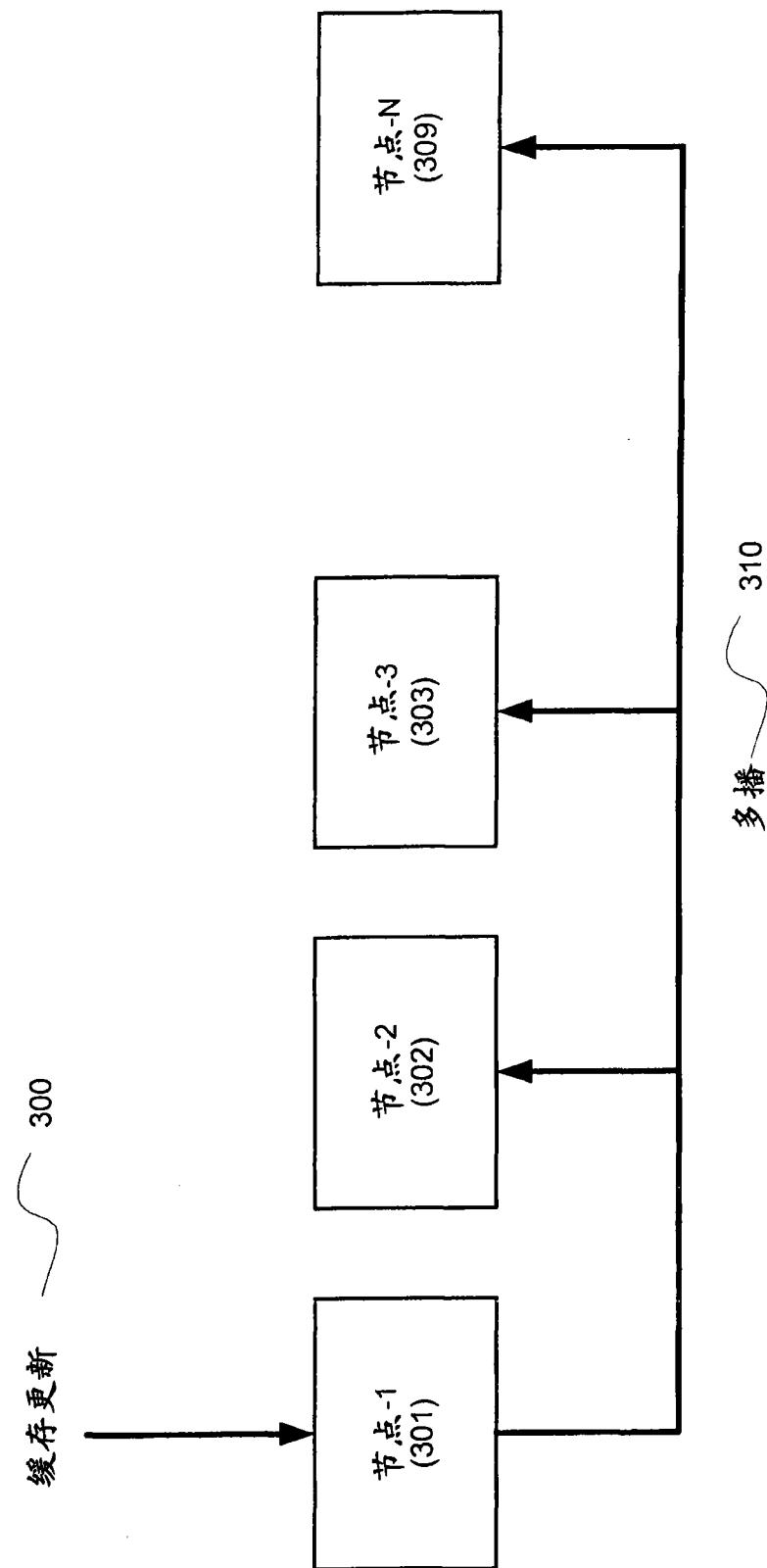


图 4

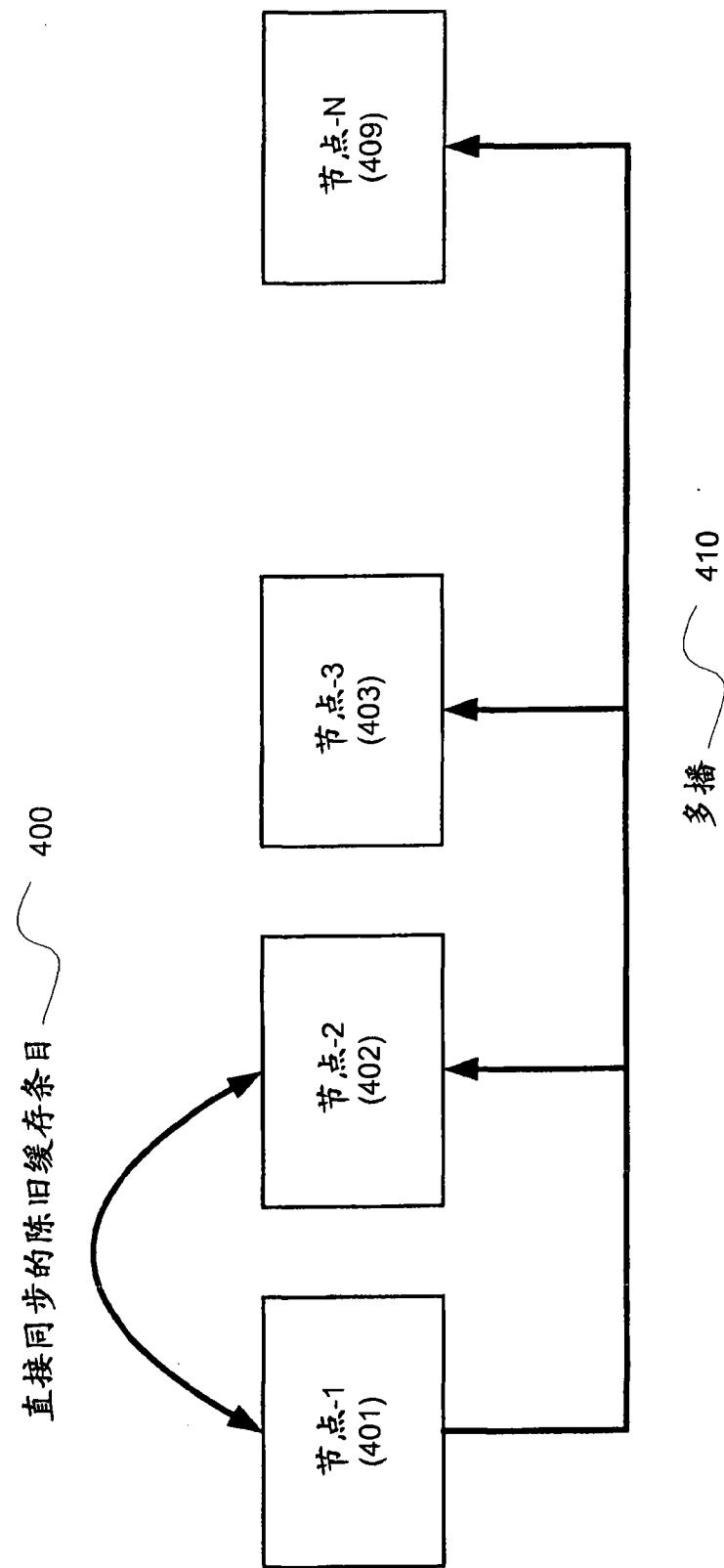


图 5

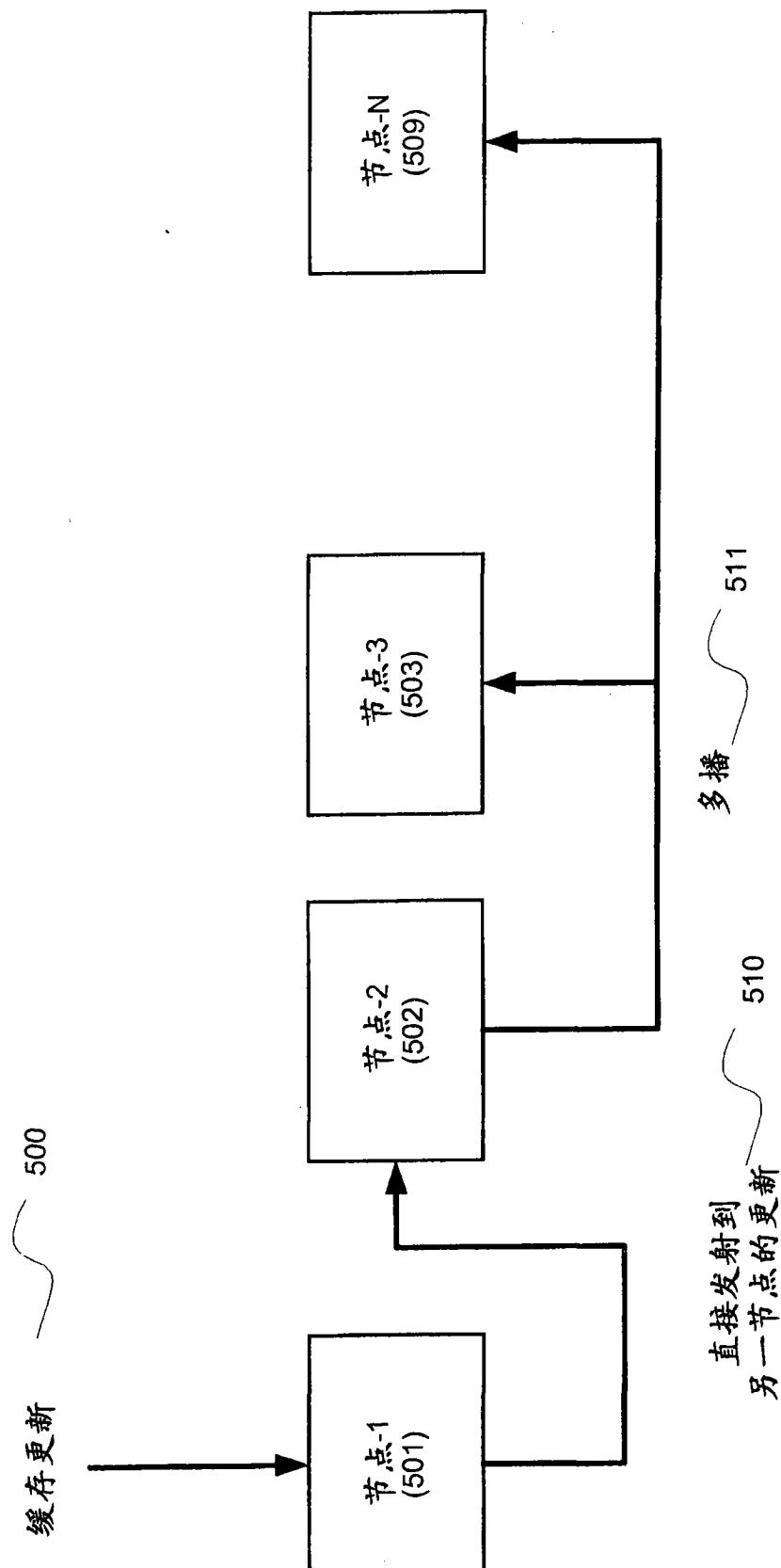


图 6

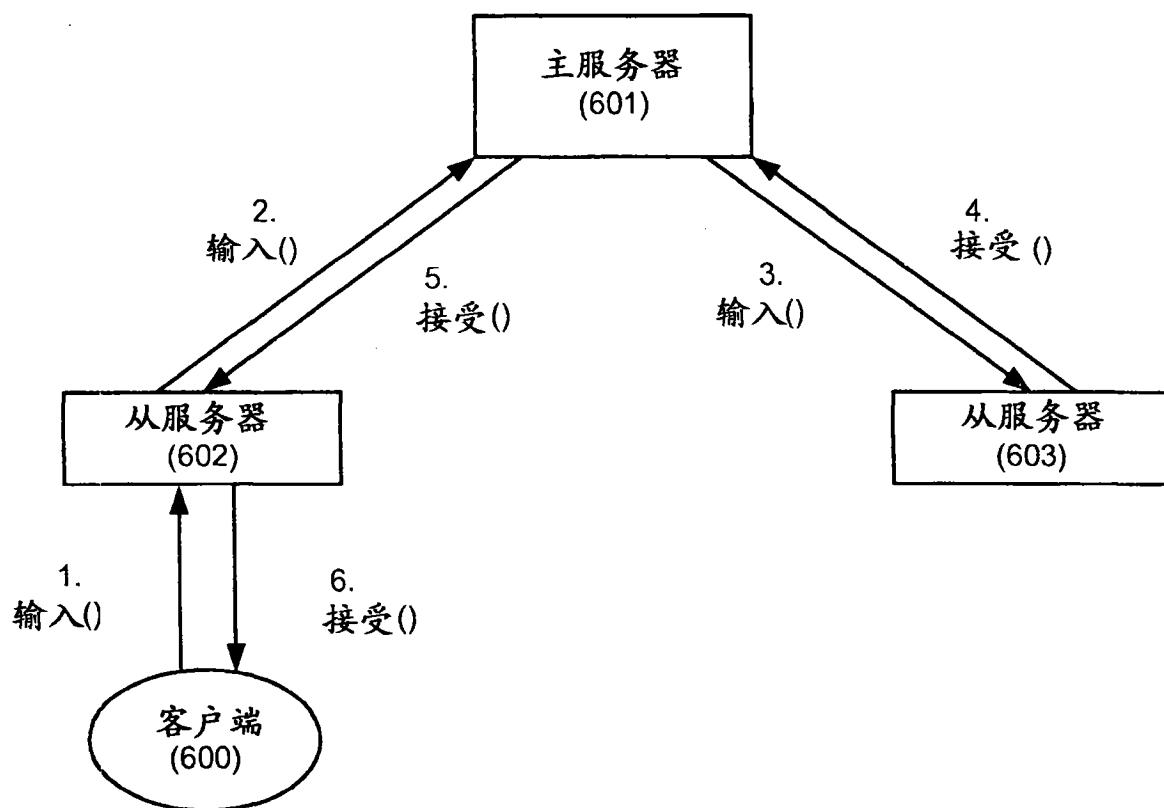


图 7

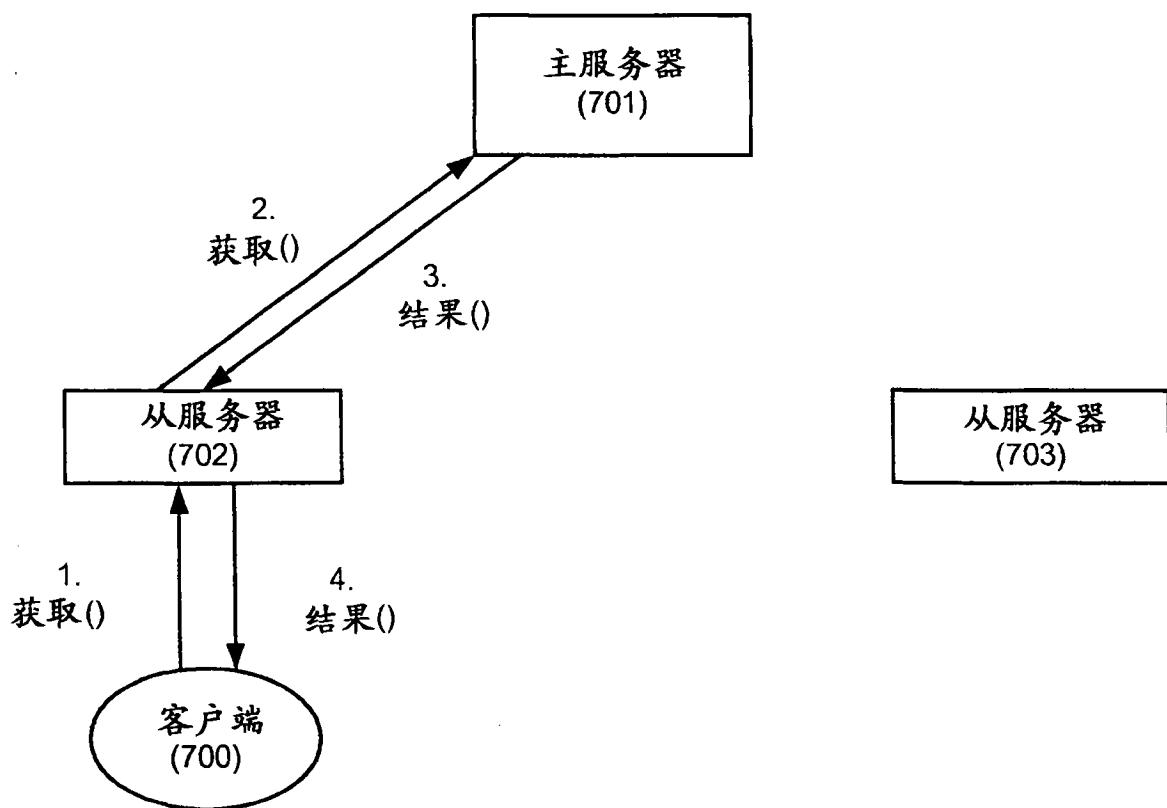


图 8

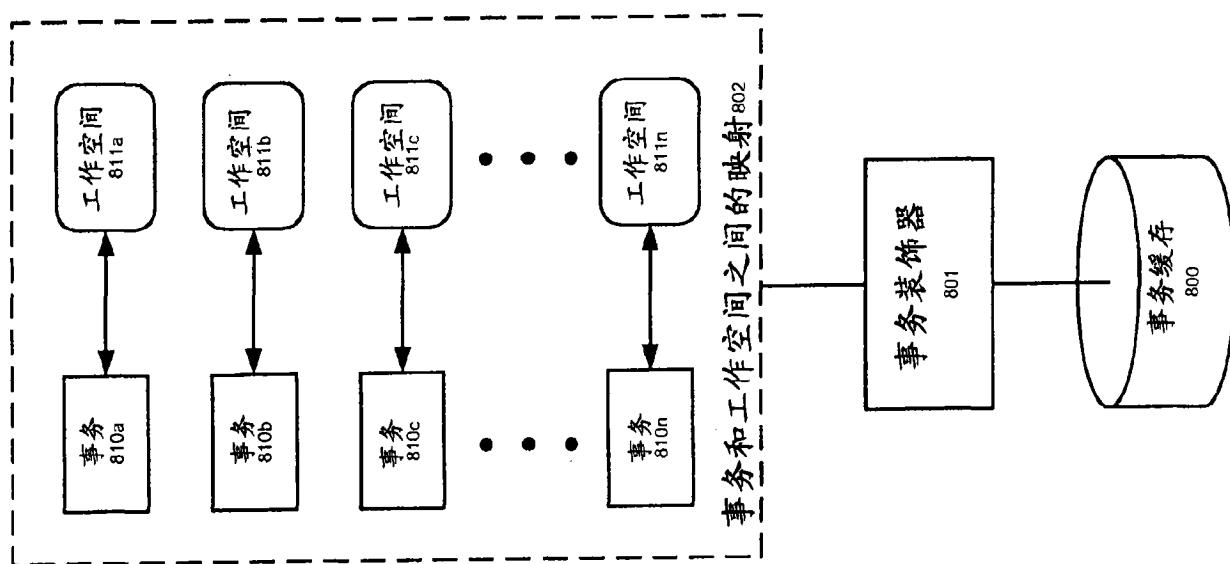


图 9

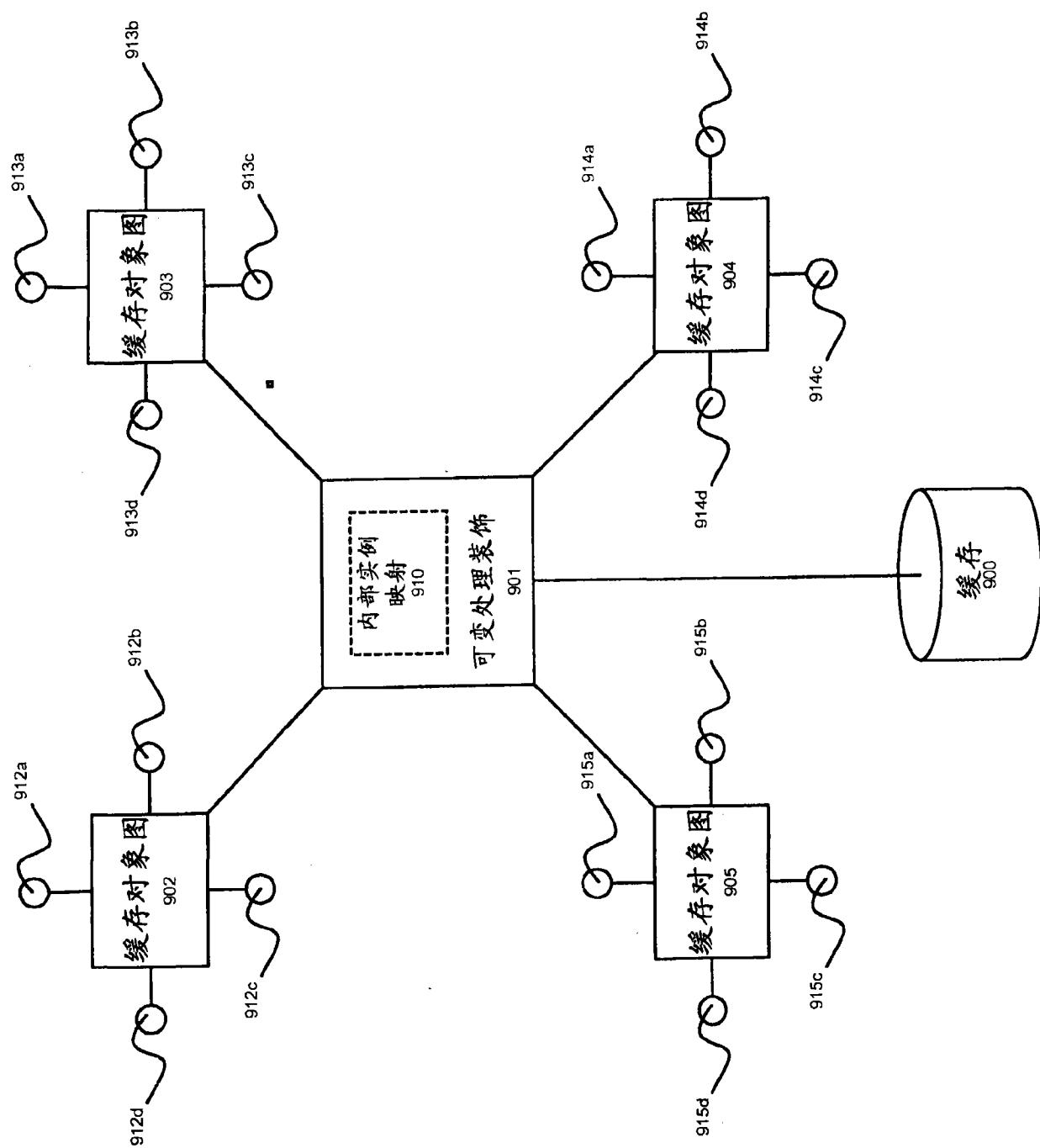


图 10