电子科技大学计算机科学与工程学院

**实 验 报 告**

**课程名称： 分布式系统**

**实验名称： 简易分布式缓存系统的设计与实现**

**学 号： 202320000392**

**姓 名： 于清源**

**电 子 科 技 大 学**

**实 验 报 告**

一、课程名称： 分布式系统

二、 实验项目名称：简易分布式缓存系统的设计与实现

三、 实验目的：

完成一个简易分布式缓存系统

四、 实验内容：

实现分布式系统，该系统满足：

1. Cache数据以Key-value形式存储在缓存系统节点内存中（不需要持久化）；
2. Cache数据以既定策略（round-robin或hash均可，不做限定）分布在不同节点（不考虑副本存储）；
3. 服务至少启动3个节点，不考虑节点动态变化；
   1. 所有节点均提供HTTP访问入口；
   2. 客户端读写访问可从任意节点接入，每个请求只支持一个key存取；
   3. 若数据所在目标存储服务器与接入服务器不同，则接入服务器向目标存储服务器发起相同操作请求，并将目标服务器结果返回客户端。
4. HTTP API约定
   1. Content-type: application/json; charset=utf-8
   2. 写入/更新缓存：POST /。使用HTTP POST方法，请求发送至根路径，请求体为JSON格式的KV内容
   3. 读取缓存 GET /{key}。使用HTTP GET方法，key直接拼接在根路径之后。为简化程序，对key格式不做要求。正常：返回HTTP 200，body为JSON格式的KV结果。错误：返回HTTP 404，body为空
   4. 删除缓存 DELETE /{key}。永远返回HTTP 200，body为删除的数量。
5. 程序基于docker打包，并通过docker compose启动运行（每个cache server为一个docker实例）。compose.yaml能直接启动不少于规定数量的cache server。每个server将内部HTTP服务端口映射至Host，外部端口从9527递增

五、 实验原理：

本实验旨在通过构建一个高性能的分布式缓存系统，深入探讨和展示高效缓存管理以及在分布式环境下对数据的快速检索技术。该系统基于Go语言构建，融合了多项关键技术和组件，以优化数据的存储和访问过程。

3.1关键技术和组件

1. 高效缓存策略：系统采用“最近最少使用”（LRU）算法作为缓存管理的核心策略。该策略通过优先淘汰长时间未被访问的缓存项，确保缓存空间的有效利用，同时提升数据检索的效率。此外，系统还集成了缓存删除机制，允许用户手动清除指定的缓存数据。
2. 一致性哈希算法：为了实现缓存数据在各个节点间的均匀分布，系统采用了一致性哈希算法。这种方法不仅优化了负载平衡，还提高了缓存数据的可访问性和检索效率。
3. Singleflight机制：系统引入了Singleflight组件来减少对相同数据的重复请求。这种机制通过聚合并合并相同的数据请求，显著提高了系统的整体性能和响应速度。
4. HTTP通信模块：该模块负责实现系统节点间的数据交换和通信。它支持分布式架构中的数据一致性和同步，确保了系统的稳定运行和高效性能。
5. 不可变数据视图：ByteView组件提供了一种安全且不可变的方式来访问缓存数据。这种设计保障了数据的完整性和一致性，避免了潜在的数据竞争和不一致问题。
6. 缓存基本操作：系统实现了添加、查询和自动淘汰等基本的缓存操作。这些功能的集成为用户提供了一个高效、易于操作的缓存数据处理接口。

3.2主要部分具体实现

**1. 缓存淘汰策略**

在本实验中，我们采用了LRU（最近最少使用）策略进行缓存淘汰，这是一种高效的数据管理方法，用于在有限的缓存空间内优化数据存储。LRU策略的核心在于优先淘汰那些长时间未被访问的数据项，从而为新的数据腾出空间。此外，实验还扩展了LRU缓存的功能，增加了删除特定缓存节点的能力，使得缓存管理更加灵活和高效。

实验中LRU缓存的实现细节如下：

双向链表结构：使用Go语言的container/list包实现了一个双向链表，以此来维护缓存项。每个缓存项包含键值对信息，链表的头部表示最近被访问的项，而尾部则代表最久未访问的项。

添加操作：当新数据项被添加到缓存时，它被放置于链表的头部。若缓存空间已满，链表尾部的项（即最久未访问的数据项）将被移除以腾出空间。

查询与删除操作：在访问某个缓存项时，若该项存在，则会被移到链表头部，这表示该项最近被访问。在删除操作中，目标项同样首先被移到链表头部，然后执行删除操作，以确保快速高效地移除目标数据。

淘汰机制：一旦缓存达到其预设的容量限制，最久未访问的项（位于链表尾部的项）会被自动淘汰，确保缓存空间的有效利用。

线程安全与同步：考虑到缓存可能会被多个协程（goroutine）同时访问，我们在实现中引入了互斥锁（mutex）。这保证了各个操作在并发环境下的原子性和一致性，避免了潜在的数据竞争问题。

通过上述设计和实现，LRU缓存在本实验中不仅实现了基本的数据存储和检索功能，还提供了高效的数据管理和淘汰策略，满足了分布式缓存系统中对性能和灵活性的需求。

2. 一致性哈希算法

在本实验中，我们采用了一致性哈希算法来实现分布式缓存系统中的数据分配和定位。一致性哈希算法通过创建一个连续的哈希空间环来优化节点增加或删除时数据的重映射需求，从而提高系统的稳定性和效率。

一致性哈希的关键组成部分和步骤如下：

哈希函数的选择：作为哈希算法的核心，本实验中选用了crc32.ChecksumIEEE函数。该函数将输入的字节数据映射成一个uint32类型的哈希值。此哈希函数的选择确保了良好的分布特性和计算效率。

添加键（Add）操作的具体实现：

副本的生成：对于每个添加到缓存中的键，系统会生成预定数量的副本。这些副本的生成是通过在原键的基础上添加不同的前缀或后缀并计算它们的哈希值来完成的。

存储哈希键值对：每个副本的哈希值及其对应的原始键都被存储在Map结构中，这个结构包括了一个哈希映射（hashMap）和一个有序的键切片（keys）。

排序：为了实现高效的查找操作，所有的哈希值（副本的哈希）在keys切片中按顺序存储。这样可以在后续查找操作中使用二分查找算法，提高查找效率。

查找键（Get）操作的实现：

计算哈希值：首先，对给定的键计算出其哈希值。

二分查找：使用二分查找法在有序的keys切片中找到距离计算出的哈希值最近的节点。该节点即为应当定位到的目标节点，也就是拥有或应当存储该键值对的节点。

通过上述设计和实现，一致性哈希在本实验中提供了一种有效的方法来分布和定位数据，特别是在处理节点的动态变化时，如节点的添加和移除。这种方法显著减少了因节点变化而导致的数据重新分配，提升了整个分布式缓存系统的效率和稳定性。

3. Singleflight 机制

在本实验中，我们采用了Singleflight机制来有效管理并发请求，确保对于同一个键（key），相关的处理函数（func）在任何给定时刻仅被执行一次。这种机制在处理高并发环境中的重复请求时特别有效，可以显著减少不必要的资源消耗和处理时间。

Singleflight机制的核心特点和实现如下：

互斥锁的应用：通过互斥锁（Mutex）控制对共享资源的访问，确保并发访问中对于每个特定的键，相关的处理函数只执行一次。这个锁机制避免了多个相同请求导致的函数重复执行，减少了计算资源的浪费。

等待组（WaitGroup）的使用：当有一个请求正在处理特定的键时，其他请求同样请求这个键的操作将被暂时挂起。为此，我们使用sync.WaitGroup来管理这些等待中的调用。这意味着，当第一个请求执行处理函数时，其他请求会进入等待状态。

共享结果：一旦被执行的请求完成其处理函数，其结果将被共享给所有在等待中的请求。这意味着，无论有多少并发请求，对于同一键的处理函数只执行一次，其结果被所有请求者共享。

提高效率与降低负载：Singleflight机制显著提高了处理效率，特别是在高并发场景下。通过减少重复的数据处理，不仅降低了服务器的负载，也加快了响应速度。

通过实现Singleflight机制，本实验的分布式缓存系统能够更加高效地处理并发请求，减少了不必要的数据处理，提高了系统的整体性能。这是构建高效且可扩展的分布式系统的关键技术之一。

4. HTTP通信

在本实验的分布式缓存系统中，我们采用了HTTP通信协议来实现节点间的高效通信，这对于数据的查询、删除和更新至关重要。通过HTTP通信，我们能够在分布式环境中实现快速且可靠的数据交换。

HTTP通信模块的关键实现细节如下：

服务端实现（ServeHTTP函数）：我们使用了ServeHTTP函数来构建服务器端的功能。这个函数对HTTP请求进行处理，并根据请求类型（GET、POST、DELETE）执行相应的逻辑。例如，在GET请求中，它检索并返回请求的数据；在POST请求中，它接受并存储新的数据；在DELETE请求中，它负责删除指定的数据。

客户端功能实现：对于客户端功能，我们实现了一个具体的HTTP客户端类httpGetter，它实现了PeerGetter接口。该接口定义了三个关键方法：Get、Delete和Update，分别用于从其他节点获取数据、删除数据以及更新数据。

节点选择器（PeerPicker接口）：PeerPicker接口的职责是根据提供的数据来选择相应的节点（PeerGetter）。我们定义了Set()方法来初始化一致性哈希算法，并添加了传入的节点信息。此外，我们为每个节点创建了一个HTTP客户端实例httpGetter。PickPeer()方法封装了一致性哈希算法的Get()方法，根据特定的键（key）选择节点，并返回对应节点的HTTP客户端实例。

通过这种设计，我们的分布式缓存系统能够在不同节点间有效地分配和管理数据请求。这种基于HTTP通信的方法不仅提供了良好的扩展性，还确保了数据处理的一致性和可靠性。这一通信模块是构建分布式系统的关键组成部分，因为它允许系统的不同部分协调一致地工作，无论它们在物理上是否分散。

5. 程序实例运行设计

1. 缓存服务启动:

使用命令go build -o server /server -port=8001启动缓存服务。这个命令触发main.go文件的执行，从而创建并启动缓存服务器。

createGroup函数被调用来创建一个名为“scores”的缓存组，并为其指定一个最大大小。

程序接着从输入命令中提取端口号，并调用startCacheServer函数来注册节点（使用RegisterPeers函数）并启动缓存服务。

1. GET命令处理

在ServeHTTP函数中，首先从输入命令提取端口号和查询的key值。

Get函数被调用以先在本地查询数据。如果本地没有找到数据，将调用load函数以在其他节点间进行查询。

在load函数中，PickPeer函数用于确定向哪个端口发送查询请求。getFromPeer函数随后构造向其他节点查询该key值的请求，其中包含local参数作为标识符，用于区分是本地还是远程查询请求。远程查询请求将仅在当前节点上执行查询，并不触发load函数。

1. DELETE命令处理

DELETE命令处理逻辑与GET命令类似，先在本地执行删除操作。如果没有可删除的key，将调用deleteload函数向其他节点发送删除请求，同时使用local标识来确定请求的来源。与GET命令不同的是，DELETE命令的返回类型为int，表示被删除的数据数量。

1. POST命令处理

POST处理中，首先检查其他节点是否已缓存了待缓存的key值。这通过调用GET处理中的Get函数实现。如果此函数返回nil，则意味着该key值已在其他节点缓存，同时该函数还会返回缓存该key的节点端口号。此外，设置了一个localGet标志，若Get函数调用了load函数，则将localGet设置为false。若Get函数返回nil且localGet为false，则向返回的端口号发送POST请求以更新数据。

如果其他节点没有该key值，则在当前节点执行Add函数，进行数据的缓存。

通过这些步骤，我们的分布式缓存系统能够有效地管理数据，在分布式节点间实现数据的同步和一致性。每个操作都经过精心设计，以确保在保证数据一致性的同时，最大化系统的效率和响应速度。

六、 实验步骤：

1. 写入/更新

命令行输入;

curl -XPOST -H "Content-type: application/json" http://server1/ -d '{"myname": "电子科技大学@2023"}'

curl -XPOST -H "Content-type: application/json" http://server2/ -d '{"tasks": ["task 1", "task 2", "task 3"]}'

curl -XPOST -H "Content-type: application/json" http://server3/ -d '{"age": 123}'

1. 读取

命令行输入：

curl http://server2/myname

curl http://server1/tasks

curl http://server1/notexistkey

1. 删除缓存

curl -XDELETE http://server3/myname

curl http://server1/myname

curl -XDELETE http://server3/myname

1. 使用测试文件sdcs-test.sh对系统进行测试

该测试文件在

test\_set

test\_get

test\_set again

test\_delete

test\_get\_after\_delete

test\_delete\_after\_delete

六个部分进行测试

以上为整个实验步骤

七、 实验数据及结果分析：

1. 写入/更新







结果为：







1. 读取



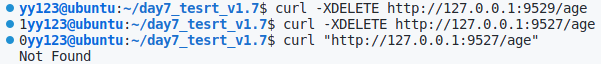




结果显示无论是本地查询还是跨端口查询都可以正常查询到

1. 删除缓存

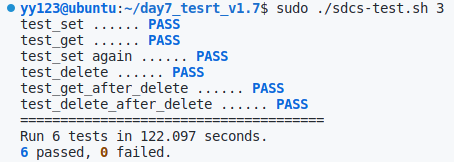




两个结果说明本地删除和跨端口删除都成功删除，返回1，删除后再次删除，因为没有可删除的缓存，返回0。最后查询删除的缓存，返回：Not Found

1. 执行测试脚本结果

将测试生成随机最大数量设为500.测试结果如下图所示：



八、 实验结论：

通过上述实验步骤测试，该分布式系统能够成功实现本地的数据缓存，获取，删除。另外在节点间也可以成功通过http进行数据缓存，获取和删除

九、 总结及心得体会：