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：

姓　　名：

分布式系统实验报告

2023 年 11 月 25 日

1 实验目的

完成一个简易分布式缓存系统。

2 实验内容

实现一个简易分布式缓存系统，该系统应满足以下条件：

1. 缓存数据以 Key-Value 形式存储在缓存系统节点的内存中（不需要持久化）。
2. 缓存数据根据一定策略（如 round-robin 或 hash）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。
   (a) 所有节点均提供 HTTP 访问入口
   (b) 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；
   (c) 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
   (a) Content-type: application/json; charset=utf-8
   (b) 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
   (c) 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
   (d) 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

3 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. 缓存机制: 采用最近最少使用（LRU）算法为基础管理缓存数据，确保高效利用缓存空间，优先移除长时间未被访问的项。在此基础上增加了删除功能，能够删除缓存的指定节点。
2. 一致性哈希算法: 使用一致性哈希来均匀分配缓存键到各个节点，实现负载平衡和有效的数据检索。
3. Singleflight 机制: 通过 Singleflight 组件减少重复的数据请求，提高了系统的整体效率和响应速度。
4. HTTP 通信模块: 实现节点间的数据交换和通信，支持分布式结构中的数据同步。
5. 不可变数据视图: ByteView 组件提供了对缓存数据的安全和不可变的访问，保障数据完整性。
6. 缓存基本操作: 实现了缓存的添加、查询和自动淘汰功能，支持高效的数据处理。

3.2 主要部分具体实现

3.2.1 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。

3.2.2 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。我们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。

3.2.3 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。

3.2.4 HTTP 通信模块

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。

另外我还设计了节点选择器（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。

具体的接口设计代码如下：

```
type PeerPicker interface {
      PickPeer(key string) (peer PeerGetter, nowport string, nowpeer
          string)
}

// PeerGetter is the interface that must be implemented by a peer.
type PeerGetter interface {
      Get(in *Request, out *Response) error
      Delete(in *Request) bool
      Update(in *Request, data string) error
}
```

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。

3.2.5 程序实例运行设计

1. 缓存服务启动:

   使用形如 go build -o server /server -port=9527 的命令启动一个缓存服务。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。

2. 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。

3. 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。

4. 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。

4 实验步骤

4.1 写入/更新

命令行输入示例：

```
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}'
```

分别测试在三个不同端口进行缓存的写入

4.2 读取

```
curl http://server2/myname
curl http://server1/tasks
curl http://server1/notexistkey
```

分别测试在不同端口进行读取，以及测试查找不存在的缓存

4.3 删除缓存

```
curl -XDELETE http://server3/myname
curl http://server1/myname
curl -XDELETE http://server3/myname
```

测试在其他端口能否删除缓存，以及删除后再次查找的情况

4.4 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，综合分析系统对数据的处理功能。

5 实验数据及结果分析

5.1 写入/更新

通过在终端输入指令，返回的值均为”save current port “，说明写入成功

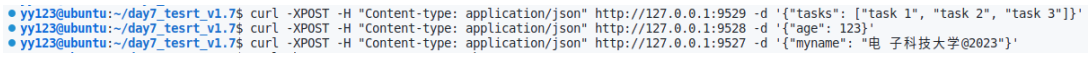

图 1: POST 输入指令

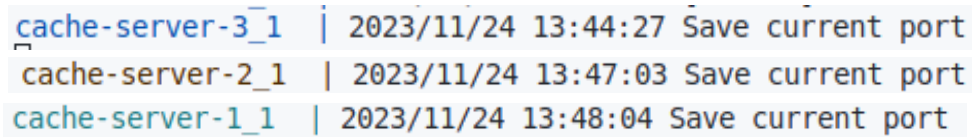

图 2: POST 返回值

5.2 读取

如下图所示，输入 GET 命令后，均可以正常返回输入的键值对。

图 3: GET 命令执行情况

5.3 删除缓存

如下图所示，首先删除本地缓存再进行查找；接下来在其他端口进行两次删除后进行查找。

第一次的 delete 返回结果为 1，说明删除成功；查找结果为 Not Found，说明已成功删除

第二次结果 delete 返回为 1，说明删除成功，第二次 delete 返回 0，说明内存中已没有该缓存。最后通过 GET 命令进行查找，查找结果为 Not Found，该缓存已被成功删除。

```
yy123@ubuntu:~/day7_tesrt_v1.7$ curl -XDELETE http://127.0.0.1:9529/tasks
1yy123@ubuntu:~/day7_tesrt_v1.7$ curl "http://127.0.0.1:9527/tasks"
Not Found

yy123@ubuntu:~/day7_tesrt_v1.7$ curl -XDELETE http://127.0.0.1:9529/age
1yy123@ubuntu:~/day7_tesrt_v1.7$ curl -XDELETE http://127.0.0.1:9527/age
0yy123@ubuntu:~/day7_tesrt_v1.7$ curl "http://127.0.0.1:9527/age"
Not Found
```

图 4: DELETE 命令执行情况

5.4 测试脚本执行结果

将脚本中 MAX_ITER 变量设为 500 进行测试，结果如下图所示：上图结果表明可以成功通

```
yy123@ubuntu:~/day7_tesrt_v1.7$ sudo ./sdcs-test.sh 3
test_set ..... PASS
test_get ..... PASS
test_set again ..... PASS
test_delete ..... PASS
test_get_after_delete ..... PASS
test_delete_after_delete ..... PASS
====================================
Run 6 tests in 122.097 seconds.
6 passed, 0 failed.
```

图 5: 脚本执行结果

过该测试脚本，该系统的主要功能通过测试。

6 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。

7 总结及心得体会

在进行分布式系统实验的过程中，我选择了 Go 语言作为开发工具，这一决定在很大程度上提升了系统处理高并发的能力，得益于 Go 语言对并发的原生支持。Goroutines 和 Channels 的使用使得并发编程变得直观而高效，这对于分布式系统的性能至关重要。Go 的简洁性和其强大的网络库也为系统的开发与维护带来了便利。

尽管如此，Go 语言的一些限制也逐渐显现。在实验过程中，Go 的错误处理机制要求开发者付出更多的精力去细致处理，同时，泛型的缺乏（直到最近的更新）也是其一大劣势。这些限制在某些情况下增加了开发的复杂度。此外，Go 的第三方库生态系统还在不断成熟中，这意味着在某些特定场景下，我需要自行实现一些功能，而不是依赖现成的解决方案。

在本次实验中，我也遇到了分布式通信的挑战。初始阶段由于缺少本地查询标识符，导致节点间通信循环，服务器处理陷入死锁。通过持续的日志审查和调试，我最终识别并解决了这一问题。

另外在实验初期，由于缺乏对代码结构和设计模式的深思熟虑，我的代码显得过于复杂且难以维护。这不仅影响了代码的可读性，也增加了后续调整和扩展功能的难度。

随着实验的深入，我开始意识到编写清晰、结构化的代码的重要性。我投入了大量时间来重构代码，通过提取重复代码段、利用更合适的设计模式、以及清晰地注释和文档化，以增强代码的可读性和维护性。外，由于程序编写的不规范性，使自己的代码十分臃肿，可读性十分低。在更改代码函数调用使其变得更加规范，也花费了很多时间。

整体来看，这次实验不仅加深了我对分布式系统架构和 Go 语言的理解，也让我认识到了编程规范对于构建一个高效、可维护的软件系统的重要性。这不仅有助于个人的代码质量提升，也为未来在更大规模的软件开发中合作提供了坚实的基础。