LRU CACHE LIBRARY

Link: [Go Mini-Project]

LIBRARY SETUP INSTRUCTIONS:
Find the library from library branch
Create a new Go Project, with a go.mod file and a .go file.
Go mod can be initialised with:

go mod init <module_name>

In the terminal of that folder, to get the library in your go module use:

go get github.com/pragadeesh-mcw/Go-Mini-Project@library

Now you can import this library as a package in your .go file/
Example:

```
import (
    cache "github.com/pragadeesh-mcw/Go-Mini-Project"
)
func main() {
    r := cache.Entry()
    r.Run(":8080")
}
```

IN-MEMORY CACHE
These functions collectively manage an in-memory cache with LRU eviction policy, ensuring thread-safe operations and periodic cleanup of expired items.

Functions and methods for in-memory caching:

LRUCache: A structure representing the cache, including capacity, time-to-live (TTL), items map, order list for LRU, a mutex for concurrency, and a channel for eviction.

NewLRUCache: Initializes and returns a new `LRUCache` instance with specified capacity and TTL. Starts the eviction routine in a separate goroutine.

startEvictionRoutine: Periodically checks for expired items based on TTL and removes them. Also handles manual eviction through a channel.

evictExpired: Removes expired items from the cache based on the current time.

Set: Adds or updates an item in the cache. Moves updated items to the front of the LRU list and evicts items if the capacity is exceeded.
Get: Retrieves an item from the cache if it exists and hasn't expired. Updates the item's position in the LRU list.

deletekey: Internally deletes an item from the cache without locking. Used in the eviction routine and `Get` method.
GetAll: Retrieves all non-expired items from the cache. Initiates eviction for expired items.
Delete: Removes a specific item from the cache if it exists.
DeleteAll: Clears all items from the cache.
evict: Removes the least recently used (LRU) item from the cache when capacity is exceeded.

REDIS CACHE
These functions collectively provide a Redis-based cache with LRU eviction policy and CRUD operations.

NewCache: Initialises a new `RedisCache` instance with a connection to the Redis server using the provided address, password, database number, and maximum size for the cache.
redis.Client: Redis instance provided by go-redis package for connection parameters.
Interface: Used to store values of any data type.

Redis List Operations:

- **LPush**: Adds an element to the start of a list.
- **RPop**: Removes and returns the last element of a list.
- **LRem**: Removes elements from a list.
- **LLen**: Gets the length of a list.
- **LRange**: Retrieves elements from a list.

Functions and methods for redis caching:

Set: Adds a key-value pair to the cache with a specified time-to-live (TTL). Updates the access order for LRU and evicts elements if necessary.
Get: Retrieves the value for a given key from the cache and updates the access order for LRU.
GetAll: Retrieves all key-value pairs currently stored in the cache.
Delete: Removes a specific key from the cache and the LRU list.
DeleteAll: Clears the entire cache by deleting all keys and the LRU list.

updateAccessOrder: Maintains the LRU order by removing the key from the list if it exists and then pushing it to the front of the list.

evictIfNecessary: Checks if the cache size exceeds the maximum size and evicts the least recently used items if necessary.

API

A **Gin Engine** in Gin allows HTTP request mapping in Go.
A **Gin Context** encapsulates HTTP requests and provides methods for HTTP response.
`ShouldBindJSON`: used to parse and bind JSON data from the body of an HTTP request to a Go struct.

Cache Interface: This interface defines the methods that in-memory cache implementation should provide. It includes methods for setting, getting, deleting, and retrieving all items, as well as deleting all items in the cache.

UNIFIED API:

SET: `POST /cache`
Description: Sets a key-value pair in the cache with an expiration time.
Parameters: JSON object containing `key`, `value`, and `expiration` (in seconds).
Response: Returns a success message upon setting the value.

GET: `GET /cache/:key`
Description: Retrieves the value associated with the specified key from the cache.
Parameters: `key` - The key to look up in the cache.
Response: Returns the value if the key is found; otherwise, returns a 404 error.

GETALL: `GET /cache`
Description: Retrieves all key-value pairs from the cache.
Response: Returns a JSON object containing all items in the cache.

DELETE: `DELETE /cache/:key`
Description: Deletes the specified key from the cache.
Parameters: `key` - The key to delete from the cache.
Response: Returns a success message if the key is deleted; otherwise, returns a 404 error.

DELETEALL: `DELETE /cache`
Description: Deletes all keys from the cache.
Response: Returns a success message if all keys are deleted; otherwise, returns a 404 error if no keys are found.

The API interaction can be done with Postman or using CURL commands as well.

BENCHMARK RESULTS:

This benchmark report provides a performance comparison between In-Memory, Redis and MultiCache that combines both In-Memory and Redis Caches. The tests were run on a system with the following specifications:

- OS: Windows
- Architecture: amd64
- CPU: AMD Ryzen 5 5500U with Radeon Graphics

Key Metrics

- ns/op: Nanoseconds per operation.
- B/op: Bytes allocated per operation.
- allocs/op: Allocations per operation.

```
PS C:\Users\praga\OneDrive\Desktop\McW\Go\MiniProject\New folder\Go-Mini-Project\test> go test -bench . -benchtime=1s -benchmem
goos: windows
goarch: amd64
pkg: unified/test
cpu: AMD Ryzen 5 5500U with Radeon Graphics
BenchmarkInMemory_Set-12                1542160              773.8 ns/op           155 B/op          7 allocs/op
BenchmarkInMemory_Get-12                6795085              167.0 ns/op             2 B/op          0 allocs/op
BenchmarkInMemory_Delete-12            11512944               95.84 ns/op            2 B/op          0 allocs/op
BenchmarkInMemory_GetAll-12                4564           256949 ns/op          166830 B/op         32 allocs/op
BenchmarkInMemory_DeleteAll-12         51647556               19.53 ns/op            0 B/op          0 allocs/op
BenchmarkMultiCacheSet-12                  790           1374812 ns/op             739 B/op         20 allocs/op
BenchmarkMultiCacheGet-12                 1052           1142540 ns/op             522 B/op         16 allocs/op
BenchmarkMultiCacheGetAll-12                 3         402743667 ns/op          527952 B/op       6076 allocs/op
BenchmarkMultiCacheDelete-12               426           2898230 ns/op            1100 B/op         31 allocs/op
BenchmarkMultiCacheDeleteAll-12              1        1960825400 ns/op          928072 B/op      23949 allocs/op
BenchmarkRedisSet-12                       524           1980547 ns/op             745 B/op         23 allocs/op
BenchmarkRedisGet-12                       813           1456708 ns/op             518 B/op         16 allocs/op
BenchmarkRedisGetAll-12                      3         454148500 ns/op          350824 B/op       6036 allocs/op
BenchmarkRedisDelete-12                   1224           1101863 ns/op             324 B/op         10 allocs/op
BenchmarkRedisDeleteAll-12                   1        1967611000 ns/op          718544 B/op      23804 allocs/op
PASS
ok      unified/test    72.658s
```

Performance Inference

In-Memory Cache

- **Performance**: The In-Memory Cache demonstrates the highest performance across all operations, with extremely low latency (ns/op) and minimal memory allocation (B/op and allocs/op).
- **Get Operation**: Exceptionally fast at 167.0 ns/op, making it ideal for high-frequency read operations.
- **Set and Delete Operations**: Efficient with sub-microsecond latencies.
- **GetAll and DeleteAll Operations**: While `GetAll` has higher latency and memory usage due to the nature of the operation, `DeleteAll` is extremely fast, showing the advantage of In-Memory operations.

Redis Cache

- **Performance**: Redis operations are significantly slower compared to In-Memory operations due to network overhead and the inherent latency of an external caching solution.
- **Set and Get Operations**: Higher latency (1-2 milliseconds) but consistent performance.
- **GetAll Operation**: Extremely high latency (454 milliseconds) and memory usage, reflecting the cost of fetching all data from Redis.
- **DeleteAll Operation**: Very high latency (1.97 seconds) and memory usage, indicating the overhead of bulk delete operations in Redis.

MultiCache

- **Set and Get Operations**: Latencies are lower than Redis alone but higher than In-Memory, as it performs operations on both caches.
- **GetAll Operation:** Slightly better than Redis but still costly (402 milliseconds) due to the need to **synchronise** data between both caches.

- **Delete and DeleteAll Operations**: Significantly higher latencies compared to either cache alone, due to the complexity of **maintaining consistency** between two caches.