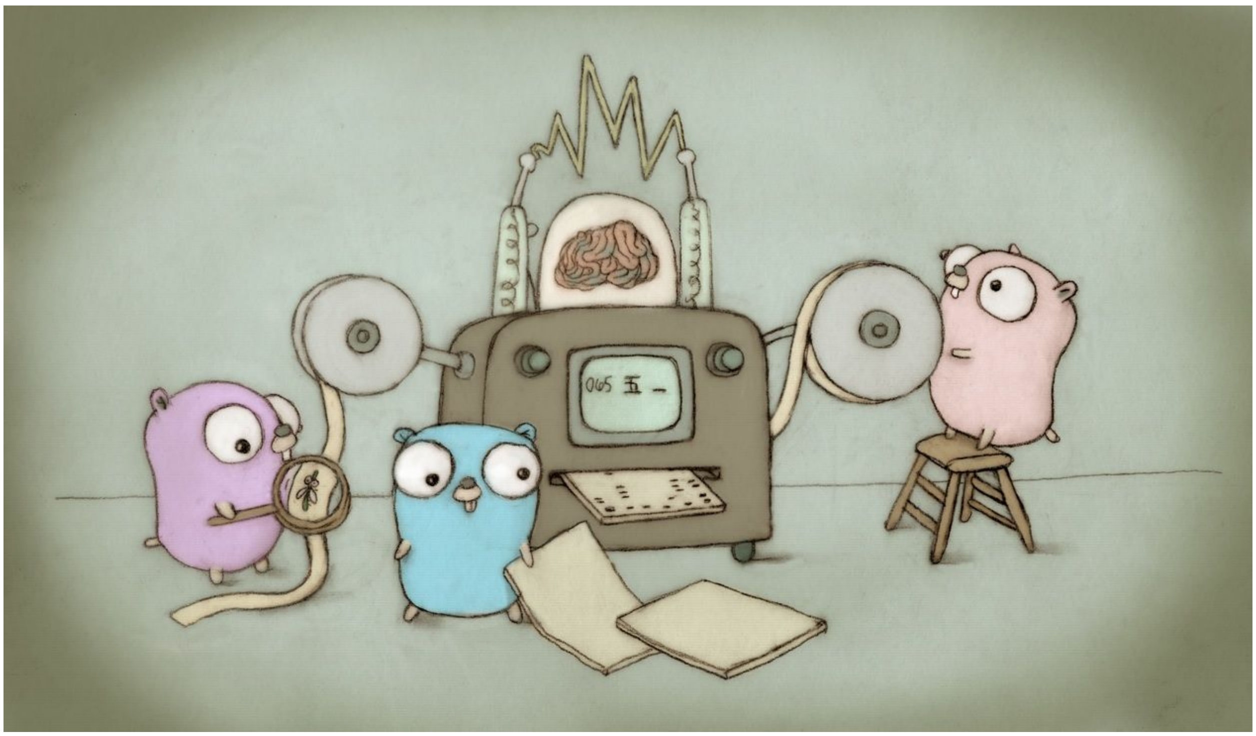
Concurrent Caching in Web Servers Using Go

A Technical Description



Gophers, The Mascots of Golang

Yiping Su
UWP 102E

Table of Contents

1.0 Introduction

This document is an overview of concurrent caching in web servers written in Golang (Go), a programming language developed by Google. Golang is a compiled language with C-like syntax. It is designed to have baseline support for concurrency, object-oriented programming, functional programming, and imperative programming. Its speed and efficiency has made it a valuable language in backend design because each program can be scaled for thousands of processes.

This document will go over the details of concurrent server caching in conjunction with a code example from Jon Calhoun in Go. The code example will explain how to retrieve top stories concurrently from the Hacker News API and store the data in the server cache.

2.0 Glossary

API - Application Programming Interface	A list or description of operations a programmer can use
Array	An array is a data structure which stores items of the same type in sequential order
Backend	The data access layer, physical infrastructure, or hardware portion of a software program
Call(s)	A function call is when you activate another function to complete a process
Data Structure(s)	A data structure is a method to organize data for effective usage
Function(s)	A function is a block of reusable code that performs a single or related action
Functional Programming	A type of programming paradigm which treats computations as a method of mathematical evaluation
HTTP Request(s)	A protocol a web browser uses to request data from a server
Imperative Programming	A type of programming paradigm which uses statements (instructions) to change a program's state
Method(s)	A method is a function which is associated with an object
Object-Oriented Programming	A type of programming paradigm which defines data and functions as objects
Parallel Computing	A type of computation in which many processes are carried out at the same time
Slice	A slice is a dynamic array which can grow in size when the initial size limit has been reached
URL(s)	Web addresses

3.0 Components of Concurrency and Servers

Concurrency in web servers requires three main parts: concurrent method, server cache, and web server. The concurrent method and server cache are both used in the software portion of the web server. This section will go over the description and application of each sub-process.

3.1 Concurrency

Concurrency is a parallel computing process in which a program, algorithm, or CPU executes multiple processes at the same time. The CPU is the central processing unit of the device; it executes the instructions of a given program. Golang supports concurrency through the use of goroutines. Goroutines are functions or methods which run at the same time as other functions and methods. Concurrent functions and methods are prefixed with the keyword *go*. Goroutines are lightweight and only take a few kB in size, thus, a Go application can run thousands of goroutines at the same time.

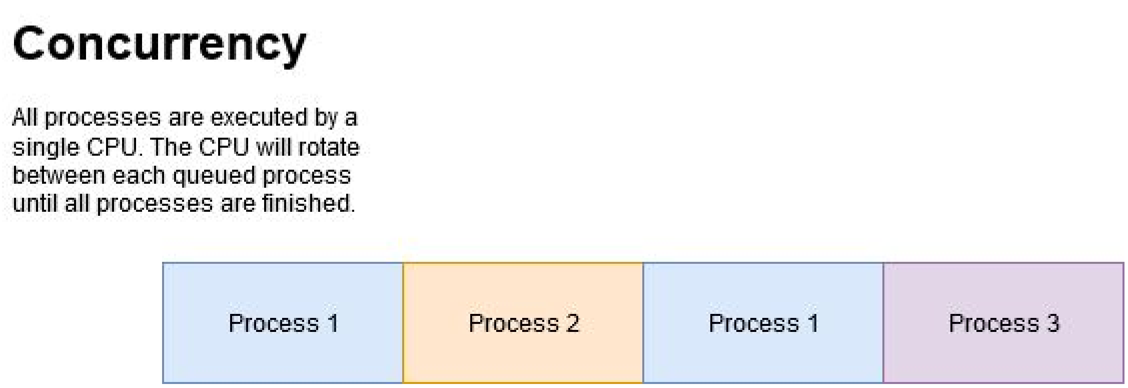


Figure 1

3.2 Cache

A cache is a software or hardware component which temporarily stores information so requests for the data can be retrieved faster.

3.2.1 Web Cache

A web cache is a temporary storage located on a user's web browser. It usually stores webpage data such as images, static HTML files, and other media files.

3.2.2 Server Cache

A server cache is a temporary storage located on a web server. Server caches are usually directly inaccessible to users. It usually stores temporary user data and data retrieved from backend processes.

3.3 Web Server

A web server is either a software, hardware, or both software and hardware component dedicated to sending and displaying data of web pages to users.

3.3.1 Hardware

The hardware portion of a web server is a physical computer which stores the data of a website's component files. Some example component files include HTML pages, CSS stylesheets, images, and JavaScript files.

3.3.2 Software

The software portion of a web server controls how the user accesses hosted files from the physical server. A software server is able to understand HTTP requests and URLs.

4.0 Design and Process

A concurrent web caching system is designed to reduce user wait time while a web application retrieves latest information from other data sources and updates it to the user's page view. The web server contains a timer which notifies another server component to update the server cache every ten minutes. The new data replaces the old data in the server cache and is displayed to the user after the next browser refresh. The data is gathered from other sources concurrently which speeds up retrieval time, especially when the amount of data scales exponentially higher. Figure 2 shows the graphical overview of the entire process.

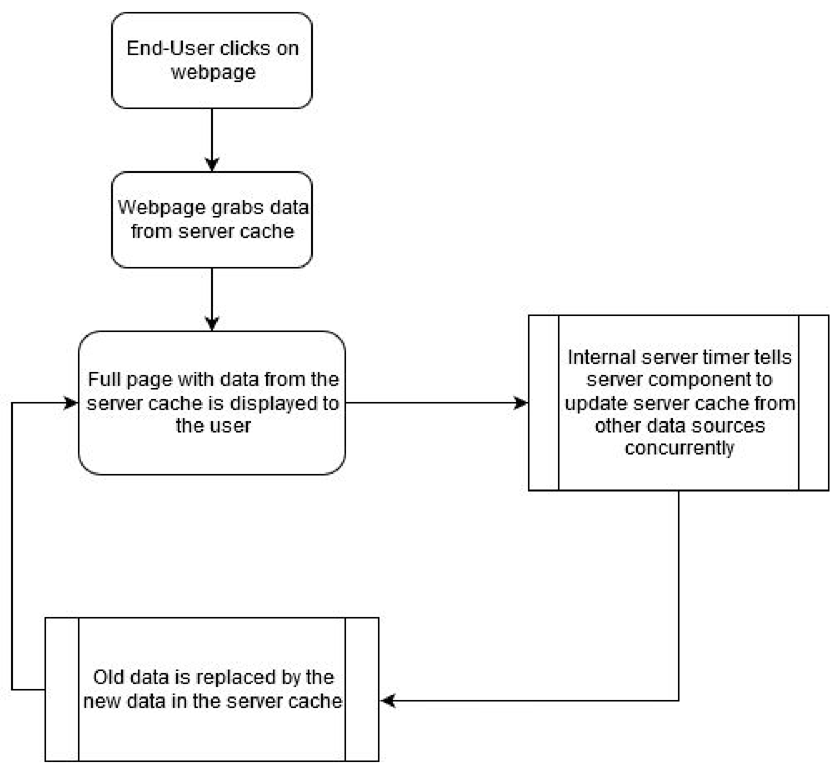


Figure 2

4.1 Concurrent Process in Go

Web servers written in Go are able to run concurrent processes using goroutines. As stated in the design overview, the data is gathered concurrently. The concurrent process is demonstrated below in figure 3.

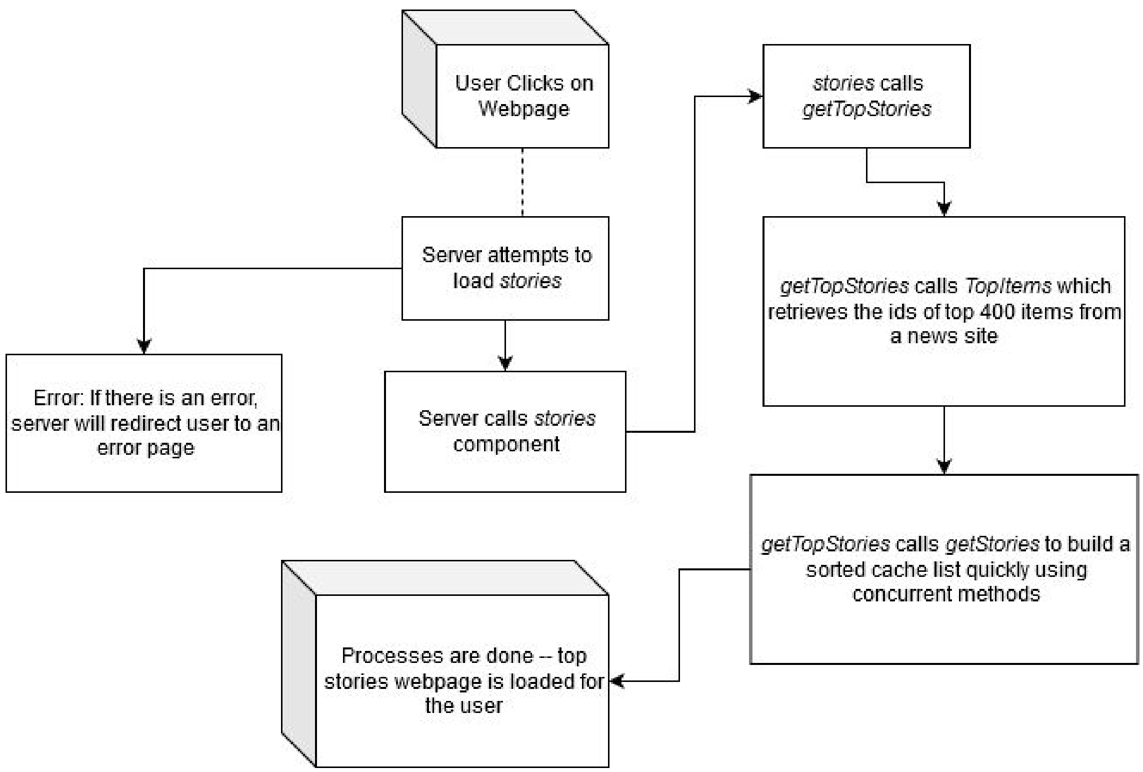


Figure 3

Each sub-process will be explained in detail in the following sections. From this point on, function names will be italicized, and data structure objects will be bolded.

4.2 Caching Process

The caching process starts when the user visits the webpage. The server will call the function, *stories*, to determine what data to show the user. If the cache is populated and the server timer has not run out, *stories* will provide the existing cache. If not, *stories* will transfer the work to another function that retrieves the updated **stories** list and reset the expiration timer.

```go
type storyCache struct {
	numStories int
	cache      []item
	expiration time.Time
	duration   time.Duration
	mutex      sync.Mutex
}

func (sc *storyCache) stories() ([]item, error) {
	sc.mutex.Lock()
	defer sc.mutex.Unlock()
	// if cache is not expired, return cache, nil
	if time.Now().Sub(sc.expiration) < 0 {
		return sc.cache, nil
	}
	stories, err := getTopStories(sc.numStories)
	if err != nil {
		return nil, err
	}
	sc.expiration = time.Now().Add(sc.duration)
	sc.cache = stories
	return sc.cache, nil
}
```

Figure 4, *Jon Calhoun*

Figure 1 shows the main overview of the concurrent process through the *stories* function. The cache on our server is made up of 5 components: total number of data to be shown (**numStories**), the cache as a data structure (**cache**), expiration time (**expiration**), duration of time to wait before retrieving new data (**duration**), and a mutual exclusion object which prevents concurrent processes from accessing a particular resource (**mutex**).

Lines 84 and 85 stops any other processes from accessing the *storyCache* until the current process is finished. Lines 87 to 93 tells the server to update the **cache** if the expiration time is up, otherwise, continue using the cache stored in the server. Lines 94 to 96 reconfigures the new expiration time if the cache is newly updated and replaces the outdated data with the new to the server cache.

4.3 Retrieving Top Stories

```
120
121  func getTopStories(numStories int) ([]item, error) {
122      var client hn.Client
123      ids, err := client.TopItems()
124      if err != nil {
125          return nil, errors.New("Failed to load top stories")
126      }
127      var stories []item
128      at := 0
129      for len(stories) < numStories {
130          need := (numStories - len(stories)) * 5 / 4
131          stories = append(stories, getStories(ids[at:at+need])...)
132          at += need
133      }
134      return stories[:numStories], nil
135  }
136
```

Figure 5, *Jon Calhoun*

Figure 2 shows how *getTopStories* builds the top story list. Lines 122 to 126 creates an object which retrieves the top 400 news links from Hacker News through the site's API.

Lines 127 to 134 creates a slice that holds the amount of top stories requested and stores the news article links inside it. All the concurrent filtering and sorting occur in *getStories* which is explained in the next section. After the list is built, *getTopStories* will return the information back to the function that called it so it is eventually displayed to the user.

4.4 Retrieving Stories Concurrently

Since the retrieval of large amounts of data can increase computation duration sharply, data retrieval is implemented concurrently. This process is shown at the bottom in figure 6.

getStories is a function which takes data from the API client and returns a data structure which contains the top N items the user specifies. Lines 138 to 142 define the resulting data structure to be made of three components: an index to track the item's position in the list, the item itself, and an error term to store errors if there were a problem extracting the item.

Lines 143 to 153 is the main concurrent process which starts multiple goroutines to find a specific record in the data which contains the same ID as the one requested. It is important to note that the function in lines 145 to 152 is completed using multiple goroutine threads. This causes the whole process to be faster because whichever process is finished first will be recorded before the others.

Lines 154 to 157 is where all the results are stored into a slice of **result**s. Lines 158 to 160 sorts the items in the array since concurrency guarantees that an element later in the list can be found before its previous element. Lines 162 to 171 checks if every **result** in the slice is valid. If it is a valid item, it will be kept in the final **stories** slice. This slice is then returned to the *stories* function where the data is ready to be displayed to the user.

```
137 func getStories(ids []int) []item {
138     type result struct {
139         idx  int
140         item item
141         err  error
142     }
143     resultCh := make(chan result)
144     for i := 0; i < len(ids); i++ {
145         go func(idx, id int) {
146             var client hn.Client
147             hnItem, err := client.GetItem(id)
148             if err != nil {
149                 resultCh <- result{idx: idx, err: err}
150             }
151             resultCh <- result{idx: idx, item: parseHNItem(hnItem)}
152         }(i, ids[i])
153     }
154     var results []result
155     for i := 0; i < len(ids); i++ {
156         results = append(results, <-resultCh)
157     }
158     sort.Slice(results, func(i, j int) bool {
159         return results[i].idx < results[j].idx
160     })
161 
162     var stories []item
163     for _, res := range results {
164         if res.err != nil {
165             continue
166         }
167         if isStoryLink(res.item) {
168             stories = append(stories, res.item)
169         }
170     }
171     return stories
172 }
```

Figure 6, *Jon Calhoun*

5.0 References

(n.d.). Retrieved from https://gobyexample.com/channels

Calhoun, J. (n.d.). Courses. Retrieved from https://www.calhoun.io/courses

Calhoun, J. (n.d.). Gophercises. Retrieved from https://gophercises.com/

Ramanathan, N. (2018, July 23). Understanding Concurrency in Golang. Retrieved from https://golangbot.com/concurrency/