Go Containers Explained In Color

REVISION 1

Hawthorne-Press.com

Go Containers Explained in Color

Published by

Hawthorne-Press.com
916 Adele Street
Houston, Texas 77009, USA

Table of Contents

Go Containers Explained in Color

Introduction

Containers are integral to modern programming practice. They take many forms. A container is generally described as object that has methods for manipulating container's contents. The Go package *container* provides three basic containers. They are *list*, *ring*, and *heap*. The Go language itself provides maps and slices. This document will only cover the container package and maps.

The container elements manipulated by container methods are always of type *interface{}*. Thus, all Go language containers can handle any type of value as long as it supports the specific container interface methods.

Container List

The list container is implemented as a classic doubly linked list. While it is not necessary to understand the internal implementation of this container to use it, it does help to understand in theory how it is constructed. Even if the internal workings change, it is helpful to have a visual picture of how this container works in theory.

A list must have a *Root Pointer.* It either points to the initial element of the list or is nil. It is created by the **list.New** method. Each Element of the list contains not only the data, but also a pointer to the previous and next elements in the list. This allows the user step both backward and forward through the list elements. The package provides a variety of methods to move to various elements within the list.

The following short program an example from the container list package:

```
package main

import (
        "container/list"
        "fmt"
)

func main() {
        // Create a new list and put some numbers in it.
        l := list.New()
        e4 := l.PushBack(4)
        e1 := l.PushFront(1)
        l.InsertBefore(3, e4)
        l.InsertAfter(2, e1)

        // Iterate through list and print its contents.
        for e := l.Front(); e != nil; e = e.Next() {
                fmt.Println(e.Value)
        }
}
```

The following results are printed one per line: **1 2 3 4.**

The four insertion methods in the above program allow for the insertion of elements in any position in the list. Two other methods, **PushBackList** and **PushFrontList** allow for appending or prepending one list to another.

```
    m := list.New()
    m.PushFront(0)
    m.PushBackList(l)
```

These results are from printing list 'm' once per line: **0 1 2 3 4.**

Positioning methods **Back**, **Front**, **Next**, and **Prev** methods allow moving through the list. We saw Front and Next used in previous program. Back and Prev can be used to step through a container in the reverse direction.

```
    for f := m.Back(); f != nil; f = f.Prev() {
        fmt.Println(f.Value)
    }
```

The **Init** method initializes a list, or if it already exists, clears the list.

If we add the following code, in red, to the original program as shown above, the result are**: 1 2 3**

```
    l := list.New()
    e4 := l.PushBack(4)
    e1 := l.PushFront(1)
    l.InsertBefore(3, e4)
    l.InsertAfter(2, e1)
    l.Remove(e4)                    // Removes element "e4" if it is in the list

// Iterate through list and print its contents.
    for e := l.Front(); e != nil; e = e.Next() {
        fmt.Println(e.Value)
    }
```

There are four methods to allow elements to be moved to other locations within the container. These are **MoveAfter, MoveBefore, MoveToBack,** and **MoveToFront**. For example, replacing the line in red above with the following:

```
    l.MoveToFront(e4)               // Move last element to front
```

Results in the following change in results: **4 1 2 3.**

Finally, the **Len** method will return the number of elements in the container.

The container/list and its companion container/ring are the simplest of the containers provided by the package. The container/heap package is a more complicated and will be discussed last.

Container Ring

A "ring" is a circular list of elements, with no beginning and no end. A *zero value* ring is a one-element ring with a nil value. An *empty ring* is represented by a nil ring pointer. Since rings do not have a beginning or end, any pointer to any ring element is a pointer to the entire ring.

Think of it as a doubly linked list where the front and back elements are cross-connected and you can use a pointer to any element to move forward and backward through the container.

Unlike container/list, the range and type of methods are different. A ring buffer is useful is situations where elements can be consumed without shuffling the containers contents. A FIFO queue is a perfect example.

The following example explores some the primary container/ring methods.

```
// Example From Github fatih/ring.go
package main

import (
    "container/ring"
    "fmt"
    "time"
)

func main() {
    coffee := []string{"kenya", "guatemala", "ethiopia"}

    // create a ring and populate it with some values
    r := ring.New(len(coffee))
    for i := 0; i < r.Len(); i++ {
        r.Value = coffee[i]
        r = r.Next()
    }

    // print all values of the ring, easy done with ring.Do()
    r.Do(func(x interface{}) {
        fmt.Println(x)
    })

    // .. or each one by one.
    for _ = range time.Tick(time.Second * 1) {
        r = r.Next()
        fmt.Println(r.Value)
    }
}
```

The first thing a developer must do using container/ring is build the initial ring. The size of a ring is initially set by the **New** method. It is loaded by stepping through the length of the container loading the elements. The element data is represented by a **r.Value** of type interface{}.

In our example, we create a string slice containing the string data we want in our ring. The length of slice is used to set the ring size and we simply iterate over the ring length, using **r.Next**, loading element data from that slice.

The only pure positioning methods are **r.Next** and **r.Prev**. Other than the **r.Len** method, the other ring methods are very different from container/list methods discussed in previous section.

The **r.Do** method will perform a specified function on element of the ring in forward order. Starting at the position of the current ring pointer and executing the function on each element in the container.

The results displayed one per line are**:** **Kenya Guatemala Ethiopia**

Since the ring has no start or end position, the container does not a have a standard append operation. However, you can join two rings into one ring, effectively expanding a list and depending how the link is performed the list can be shortened and sub-lists can be extracted. This is all accomplished with the **r.Link** and **r.Unlink** methods.

To explore these methods the initial program was modified as follows:

```
package main

import (
    "container/ring"
    "fmt"
)

func main() {
    coffee := []string{"Kenya", "Quatemala", "Ethiopia"}
    softdrink := []string{"Coke", "Pepsi", "DrPepper", "Slice"}   // Add soft drink list

    // create a ring and populate it with coffee values
    r := ring.New(len(coffee))
    for i := 0; i < r.Len(); i++ {
        r.Value = coffee[i]
        r = r.Next()
    }

    // create a ring and populate it with softdrink values
    s := ring.New(len(softdrink))
    for i := 0; i < s.Len(); i++ {
        s.Value = softdrink[i]
        s = s.Next()
    }

    // print all values in both rings with ring.Do()
    r.Do(func(x interface{}) {
        fmt.Println(x)
    })

    s.Do(func(x interface{}) {
        fmt.Println(x)
    })
}
```

The output displayed one per line is: **Kenya Guatemala Ethiopia Coke Pepsi DrPepper Slice.**

Merging Two Rings

Merging these two rings requires they be "linked". The linking process essentially breaks ring **'r'** at the pointer location and inserts ring **'s'** at that location. When done, **r.Next** points at the **'s'** ring element at the specified **'s'** pointer position and the returned value is a pointer to the original **r.Next** position.

By adding the following code:

```
    z := r.Link(s)
    z.Do(func(x interface{}) {
        fmt.Println(x)
    })
```

Printing **'z'** results in the same output shown above, the **'r'** values plus the **'s'** values. *The exact same results are a function of using the same ring pointers*. Changing the ring pointer positions would of course change the results printed. Remember, rings have no fixed beginning or end, just the position of the current ring pointer.

Removing A Section of A Ring

If the **'r'** and **'s'** point at the same ring, the **Link** function will remove the elements *between* **'r'** and **'s'** and the resulting sub-ring is returned as a result. The start of the sub-string is **r.Next** and the end is **s.Prev**. The following is the modified link code:

```
    z := r.Link(s)
    z1 := z.Next().Next()
    z2 := z1.Next().Next().Next()
    zz := z1.Link(z2)
    zz.Do(func(x interface{}) {
        fmt.Println(x)
    })
```

The results printed one per line are: **Coke Pepsi.**

If the link process does not remove any elements, the result is the still the original **r.Next** not **nil.** For example, changing the line:

```
        z2 := z1.Next().Next().Next()
```

To the following:

```
        z2 := z1.Next()
```

Prints the results one per line: **Coke Pepsi Dr Pepper Slice Kena Quatemala Ethiopia**

All we have actually done is change the start of the '**zz'** pointer to "Coke". If this result confuses you, you need to reread the container/ring package documentation.

Moving The Ring Position Pointer

In the previous examples we used the **Next** method to change the position of the ring pointer. While this illustrated the ability of using basic methods together by chaining, it is obviously an inefficient way to change a ring position. The more idiomatic way is the use the **Move** method as shown below in red.

```
    z := r.Link(s)
    z1 := z.Move(2)
    z2 := z1.Move(3)
    zz := z1.Link(z2)


    zz.Do(func(x interface{}) {
        fmt.Println(x)
    })
```

The Move method will accept both positive and negative values. Since the ring is circular, offset values for **z2** of 3, 10, and -11 all result in the same ring pointer value.

Unlinking A Section Of A Ring

The method **Unlink** also removes a section of a ring and returns it as a sub-ring. The difference is how you specify the section to remove. While the **Link** method specifies non-inclusive the start and end positions. The **Unlink** method specifies a non-inclusive starting point, **'r',** and the number of elements to unlink!

The method will start the unlinking process at **r.Next** and continue for **'n'** elements. The effect is that these elements are removed and placed in a sub-ring. The sub-ring is returned as the result.

This allows us to simply the removal of a sub-string. See the modified statements in red.

```
    z := r.Link(s)
    z1 := z.Move(2)
    zz := z1.Unlink(2)
    zz.Do(func(x interface{}) {
        fmt.Println(x)
    })
```

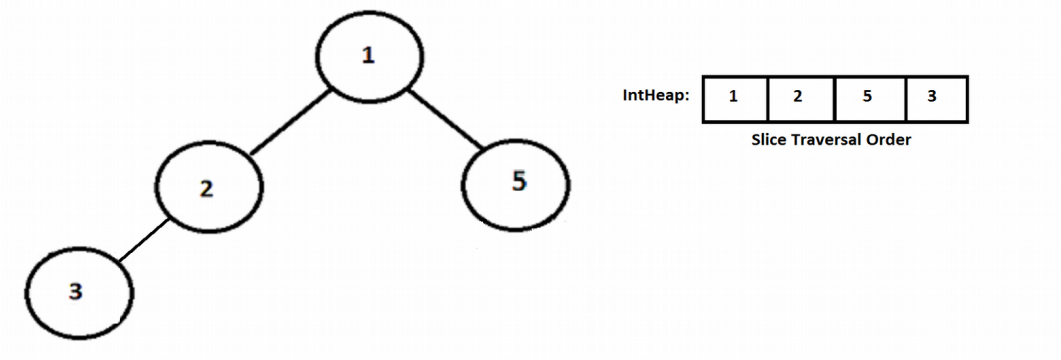
Again the results one per line are: **Coke Pepsi.**

Container Heap

The heap is setup as a tree structure where each node is has the minimum value of its sub-tree. This container is often used to support priority queues of various types. An extended example of a priority queue can be found in the document ***Go Concurrency and Load Balancing Explained in Color***. It is available at: http://www.hawthorne-press.com/GOConcurrencyLoadBalancing.pdf .

In this document, we will explore both examples shown in the container/heap package documentation, and make clear how the heap works. The following program implements a simple integer heap.

The main function inserts, in the following order, the values 1, 2, 5, and 3 into the heap. After loading the values, the heap's tree representation and the contents of the slice **IntHeap** look as follows:




```
// This example demonstrates an integer heap built using the heap interface.

package main

import (
        "container/heap"
        "fmt"
)

// An IntHeap is a min-heap of int
type IntHeap []int

func (h IntHeap) Len() int           { return len(h) }
```

```
func (h IntHeap) Less(i, j int) bool { return h[i] < h[j] }
func (h IntHeap) Swap(i, j int)      { h[i], h[j] = h[j], h[i] }

func (h *IntHeap) Push(x interface{}) {
	// Push and Pop use pointer receivers because they modify the slice's length,
	// not just its contents.
	*h = append(*h, x.(int))
}

func (h *IntHeap) Pop() interface{} {
	old := *h
	n := len(old)
	x := old[n-1]
	*h = old[0 : n-1]
	return x
}

// This example inserts several ints into an IntHeap, checks the minimum,
// and removes them in order of priority.
func main() {
	h := &IntHeap{2, 1, 5}
	heap.Init(h)
	heap.Push(h, 3)
	fmt.Printf("minimum: %d\n", (*h)[0])
	for h.Len() > 0 {
		fmt.Printf("%d ", heap.Pop(h))
	}
}
```

The first thing to understand is that the heap totally contained in slice **IntHeap**. The heap interface functions provided by the user manipulate the values in that slice. Keep that in mind as we go through this example.

Heap Structure

The heap is stored in a slice of the appropriate type, in our case it is a simple integer. For programs that are more complex it is generally a structure. Whatever the type, these elements are always stored in the slice in *Level Traversal Order*. This means that each level of the tree is stored sequentially in the slice.

In relation to our example, the first level is 1, the second level is 2 and 5, the third level is incomplete and contains only a 3. Our heap is represented as a binary tree, thus each level has twice as many entries as the previous level. A complete four level tree would store 1, 2, 4, and 8 items serially in the slice. Thus, if the value every node was the same as its level number, the heap would contain: 12244448 8 8 8 8 8 8 8.

One additional complication is that the internal processing for the **Pop** method swaps the zero and len-1 values and internally reorders the slice with the exception of the len-1 value before calling the *user defined Pop* method. This means the slice is correctly ordered when the user defined Pop method removes the len-1 value and shortens the slice by one.

The user Push routine appends to the heap slice and the user Pop removes the last element from the slice and shortens the slice by one. The heap processing takes care of the reordering by using various user defined processing methods.

User Supplied Methods

In order to satisfy the heap container interface, the developer must create the five methods defined by the interface. *These methods are not called by the user directly*; they are used internally by the heap processing.

For example, when you call **heap.Push**, the user method *Push* is called and when it returns the heap is reordered as necessary before returning the user.

Heap processing does not know the contents of the elements; it depends on the following methods to handle those elements. The user methods are responsible for performing the necessary element manipulations.

Method: len() int

This method simply returns an integer holding the number of elements in the heap.

Method: less(i, j int) bool

The less method compares two elements in the heap, identified by their heap index, and returns *true* if the element at index **'i'** is less than the element at index **'j'.**

Method: swap(i, j int)

The Swap method does what it implies, it swaps the elements at indexes **'i'** and **'j'.**

Method: Push(x interface())

This method appends an element to end of the slice.

```
func (h *IntHeap) Push(x interface{}) {
        *h = append(*h, x.(int))            // append new element to end of slice
}
```

Method: Pop() interface{}

This method is a little more involved. Given that the internal processing has swapped the zero and len-1 values and reordered the slice, this method must save the contents of the last element and shorten the slice by one. Finally returning saved element to the user.

```
func (h *IntHeap) Pop() interface{} {
        old := *h                           // Make copy of heap pointer
        n := len(old)                       // Get Length
        x := old[n-1]                       // Save len-1 value (This is the minimum value)
        *h = old[0 : n-1]                   // Overwrite Heap Pointer with the slice shortened by one
        return x                            // Return save value
}
```

Heap Supplied Methods

These methods are called by the user to manage the heap. The user must first define the element type that the heap will manage, in our example it is:

type IntHeap []int.

The user is responsible for creating the user-defined methods, described in the previous section to manipulate the elements of the defined type.

Given a slice **'h'** of the element type, the following methods are defined for the heap interface.

Method Fix(h Interface, i int)

Reorders the heap after an element at index **'i'** has changed it value. For elements that are complex, such as structures, the "changed" value is the item that specifies the heap ordering. Other values within the element may be changed without effect. A more expensive processing method of **remove(h, i)** followed by a **Push** of a new value is equivalent, but more time consuming, primarily because the heap will probably be reordered twice.

Method Init(h Interface)

A heap must be initialized before heap operations can commence. In our example program, a slice of the appropriate type is created and passed to the **Init** method. The result is an initialized heap of the correct order.

```
        h := &IntHeap{2, 1, 5}
        heap.Init(h)
```

The heap initialization is only to order the heap properly. There are no "control" codes or structure other than arranging the heap slice in *level traversal order*. In fact simply laying out the slice correctly, [1, 2, 5] in this case, would work without calling the **Init** method. However, this is not advisable.

Method Pop(h Interface) interface{}

Returns the minimum value element from the heap, as determined by the user *Less* method. Internally, the heap processing swaps the **zero** and **len-1** values in the heap, then reorders the heap and calls the user *Pop* method.

Method Push(h Interface, x interface{})

This method pushes the element **'x'** onto the heap. It expects that the user Push method simply appends the new element to the heap slice. It then reorders the heap.

Method Remove(h Interface, i int) interface{}

Removes element at index **'i'** from the heap, reorders the heap and returns the removed element.

Type Interface interface {}

The following is the definition of type Interface. The methods Push and Pop are for internal use only.

```
type Interface interface {
        sort.Interface
        Push(x interface{})     // add x as element Len()
        Pop() interface{}       // remove and return element Len() - 1.
}
```

Priority Queue Example

The proceeding example was simple in nature. Now we shall tackle a more real-world example, where the elements are a structure containing items that are not directly linked to its "priority". We will examine various details of the priority queue example from the container/heap package. It can be found at: https://golang.org/pkg/container/heap/#example__intHeap .

Element Structure

The element in our example is a structure that contains a value, priority, and an index.

```
type Item struct {
        value    string      // The value of the item; arbitrary.
        priority int         // The priority of the item in the queue.
        index int            // The index of the item in the heap.
}
```

The *value* entry is only a placeholder for data in a real-world example. The *priority* entry controls the elements placement in the queue. Finally, the *index* value is needed by the **Update** function and is maintained by the user defined methods **Swap**, **Push**, and **Pop**. The primary need for an index value is that when updating a priority value, the index of that value is needed by the heap's **Fix** method.

Method Len()

This method simply returns the number of elements in the heap store.

Method Less(i , j int)

This method introduces our first oddity, *the Less method does not always mean less.* This method only determines the priority order. Whereas the previous example wanted the lowest priority, this program wants to select the highest priority. Thus, the comparison selects for **'i'** being greater than **'j'**.

```
        func (pq PriorityQueue) Less(i, j int) bool {
            return pq[i].priority > pq[j].priority              // Select for highest priority!
        }
```

Method Swap(i , j int)

The Swap method not only swaps the elements **'i'** and **'j'**, it must also set each of the indexes appropriately. The index in an element Item must match their heap index.

```
        func (pq PriorityQueue) Swap(i, j int) {
            pq[i], pq[j] = pq[j], pq[i]             // Swap Element 'i' and 'j'
            pq[i].index = i                         // Set element[i].index = 'i'
            pq[j].index = j                         // Set element[j].index = 'j'
        }
```

Method Push(x interface{})

As in the simpler integer case, we must append the new item to the heap. However, with the added complication of determining and storing the index value in the new element. After getting a reference to the actual item via type assertion, the index value is set the current length of the heap and the element is appended to the heap.

```
        func (pq *PriorityQueue) Push(x interface{}) {
            n := len(*pq)                       // Save Len of Heap
            item := x.(*Item)                   // Get reference to actual value of x
            item.index = n                      // Store index value
            *pq = append(*pq, item)             // Append new element to heap
        }
```

Method Pop() interface{}

When this method is called by the heap interface, the first and last values in the heap have been exchanged and the heap reordered excluding the last value. The user method is responsible for returning the last value on the heap as the result and shortening the heap by one element. As a safety measure, the index of the returned value is set to -1 to insure if it used as an element in the future its index will be invalid.

```
        func (pq *PriorityQueue) Pop() interface{} {
            old := *pq                      // Get Current heap pointer
            n := len(old)                   // Get Current length of heap
            item := old[n-1]                // Get reference to last element on the heap
            item.index = -1 // for safety   // Set its index to -1 (Index is invalid)
            *pq = old[0 : n-1]              // Shorten heap by one element
            return item                     // return previous last element
        }
```

Method update(item *item, value string, priority int)

This is a local method *called by the user*, not by the heap interface. Its purpose is to update the data and priority of an element in the heap. After changing the items value and priority, the heap is reordered by call the Fix Method. Notice, that in order for Fix to work, it must know the positional index of the item in the heap. This is the primary reason that the positional index is maintained by each element. By only having a pointer to an element, we can determine its positional index and update the heap order efficiently.

```
        func (pq *PriorityQueue) update(item *Item, value string, priority int) {
            item.value = value              // Insert New Value
            item.priority = priority        // Change Priority Value
            heap.Fix(pq, item.index)        // Reorder Priority Queue heap.
        }
```

Function Main()

The main function is similar to the previous example, with the exception of adding an **update** function. The slice that that becomes the heap is created, loaded with data, and initialized with the **Init** method. We then push a new element on the heap. Aside from how these actions are performed, these actions mirror the integer heap example.

The new feature added was updating the priority and value of an element in the heap using the user **update** method. The function main is shown below:

```
// This example creates a PriorityQueue with some items, adds and manipulates an item,
// and then removes the items in priority order.
func main() {
        // Some items and their priorities.
        items := map[string]int{
                "banana": 3, "apple": 2, "pear": 4,
        }


        // Create a priority queue, put the items in it, and
        // establish the priority queue (heap) invariants.
        pq := make(PriorityQueue, len(items))
        i := 0
        for value, priority := range items {   // Step thru map and create Priority Queue elements
                pq[i] = &Item{
                        value:    value,
                        priority: priority,
                        index:    i,
                }
```

```
                i++
        }

        heap.Init(&pq)                          // Initialize Priority Queue Heap

        // Create a new item
        item := &Item{
                value:    "orange",
                priority: 1,
        }

        heap.Push(&pq, item)                    // Push new item into heap

        pq.update(item, item.value, 5)          // Update Priority Queue!

        // Take the items out; they arrive in decreasing priority order.
        for pq.Len() > 0 {
                item := heap.Pop(&pq).(*Item)
                fmt.Printf("%.2d:%s ", item.priority, item.value)
        }
}
```

The resulting display is: **05:orange 04:pear 03:banana 02:apple**

Maps

Maps are Go's implementation of a *Hash Table*, one of the foundational data structures of computer science. Hash tables provide fast lookups, as well as adding and deleting table elements quickly. The internal structure is modern hash table implementation and can be counted on to be as efficient as any general hash table.

Maps, also called an *associative memory* or *associative array*, are often used for structures such a symbol tables, persistent data, and database indexes. A map associates a "key" with a "value". To access the map, the user specifies a key and the related value is returned. The efficiency of this transaction is created by storing the value in an array indexed by a hash value (constructed from the key). The average lookup time approaches O(1) in the average case and worst-case of O(n).

The Go map processing automatically takes care of hash function and sizing issues. There is no way to pre-specify the initial map size. The map is created with a default size, and the size doubles when the map approaches saturation. Retrieval times may start exceeding O(1) by various amounts as the map approaches saturation, which is when most of the slots in the map are filled. The internal mechanics of the Go map are not guaranteed to remain unchanged, only the defined behavior of the map.

Map Creation

A *map type* is created as follows: **map[KeyType]ValueType**

The **KeyType** may be any type that is *comparable,* and **ValueType** may be any type at all, including another map.

A map may be created and initialized in the following ways:

```
        var m map[string]int                // Create a uninitialized map
        m = make(map[string]int)            // Initialize map "m"

         n := map[string]int{}              // Uninitialized Composite literal construction
```

```
        f := map[string]int {               // Composite literal construction and initialization.
                "MainFloor": 1,
                "Penthouse": 14,
        }
```

Maps are reference types like slices and pointers, so the initial value of **"m"** is nil. Attempts to read an uninitialized map reference returns zero. However, writing to an uninitialized map causes a run-time error.

Maps are initialized with the **make** function or a composite literal statement. See examples above.

Using Maps

A developer may add and delete items in a map, and return the number of items in map. The following code illustrates various map operations:

```
        f["SecondFloor"] = 2                // Add new floor to map

        p := f["Penthouse"]                 // Return Penthouse floor

        b := f["Basement"]                  // Returns zero – No basement listed
        b, ok := f["Basement"]              // b equals 0  (ok will be False)
        p, ok := f["Penthouse"]             // p equals 14 (ok will be True)
        _, ok := f["ThirdFloor"]            // Test for existence only (ok will be False)

        l := len(f)                         // Length – Number of items in map "f"

        delete(f, "Penthouse")              // Delete element from map!
```

Key Types

Key types are restricted to types that are *comparable*, keys that can be compared for equality. These Include:

- Boolean
- Numeric
- Strings
- Pointers
- Channels
- Interface Types
- Structs and Arrays composed of only these types

Notice that slices, maps, and functions are not permitted, as these types cannot be compared with **"=="** operator.

Using Structures As Keys

It may be surprising that structures can be used as keys. Suppose a developer wants to map a two dimensional struct to a particular integer result. The work case may be counting the occurrences of combinations of items laid out on a two-dimensional grid, say items on the y-axis against prices on the x-axis. For example:

```
package main

import (
    "fmt"
)
```

```
// item / price grid

func main() {
    type grid struct {               // Defines grid structure
        item  string                 // Item
        price float64                // Value of item
    }

    hits := make(map[grid]int)

    hits[grid{"Doll", 1.00}]++       // Increment Dolls ordered
    hits[grid{"Doll", 1.99}]++       // Increment Dolls ordered
    hits[grid{"Hat",  2.99}]++       // Increment Cowboy Hats ordered

    cheapDoll := hits[grid{"Doll", 1.00}]

    fmt.Println("Cheap Dolls on Hand = ", cheapDoll)
}
```

This program displays the following results: Cheap Dolls on Hand = 1

Concurrency Issues

Maps are not inherently safe for concurrent use. They must be mediated by a synchronization mechanism. One method is to protect maps with **sync RWMutex**. A more idiomatic Go language method is use channels for serially accessing a map. Remember, the results of reading or writing a map from concurrently executing Goroutines is not defined without synchronization.

The following simple program shows how this can be accomplished. First, we setup a map to maintain a count of items in our inventory. For this example, we are only counting the “Dolls” on hand. Second, we setup two concurrent Goroutines, one incrementing the number of dolls and one reading and printing the number of dolls on hand. When both routines complete we will print out the total number of dolls. Both Goroutines are sleeping a random time of approximately 300 to 800 milliseconds before each action is taken.

Since reads and writes are happening at random times, the display of count values may be the same. This occurs when the two sequential reads are executed without an intervening increment being executed. This all depends on the random delays in the Goroutines. In addition, since we are using a constant seed for the random number generator this values will probably remain the same from run to run.

Be aware that this program does not check for many real-world errors, such as messages without “Dolls” as the string.

```
// Hawthorne-press.com July 2016
package main

import (
    "fmt"
    "math/rand"
    "time"
)

// Generate a random integer between 300 and 800
func getRand() int64 {
    return int64(((rand.Float64() * 5) + 3) * 100)
}
```

```
// Sleep a random time and send increment message for "Dolls"
func cntDolls(ch chan string) {
    for i := 0; i < 10; i++ {
        time.Sleep((time.Millisecond) * time.Duration(getRand()))
        ch <- "Dolls"
    }
}

// Sleep a random time and send read message for "Dolls"
func readDolls(rd chan string) {
    for i := 0; i < 10; i++ {
        time.Sleep((time.Millisecond) * time.Duration(getRand()))
        rd <- "Dolls"
    }
}

// Main Function
func main() {
    rand.Seed(63)                       // Set Random Seed
    ch := make(chan string, 1)          // Create Increment Channel
    rd := make(chan string, 1)          // Create Read Channel
    inventory := make(map[string]int) // Create Inventory Map
    inventory["Dolls"] = 0              // Initialize Dolls Count

    go cntDolls(ch)                     // Launch Increment Goroutine
    go readDolls(rd)                    // Launch Read Goroutine

    for j := 0; j < 20; j++ {           // Process 20 Increments and Reads
        select {
        case msg := <-ch:               // Increment Message Handler
            if msg == "Dolls" {         // Count only Dolls
                inventory[msg]++
            }
        case item := <-rd:              // Read Message Handler
            if item == "Dolls" {        // Print only Dolls Count
                fmt.Println("Dolls = ", inventory[item])
            }
        }
    }
    fmt.Println("Total Dolls  = ", inventory["Dolls"])
}
```

An example of the output is as follows:

Dolls = 0
Dolls = 2
Dolls = 2
Dolls = 3
Dolls = 4
Dolls = 5
Dolls = 7
Dolls = 8
Dolls = 10
Dolls = 10
Total Dolls = 10

Iteration Order

When iterating over a map with a range loop, the order is not specified and is not guaranteed to be the same from on iteration to another. If you require a stable iteration order, you must maintain a different data structure that specifies the order.

This would entail creating a list of keys, sorting them, and using the sorted key list to access the map in the desired order.

This completes our discussion of maps.

A Final Word

Strive for simplicity in your code. The *Occam's Razor principle* applies as strongly for software as it does for many other walks of life. In most situations, giving up a little in efficiency to simplify your code is often the right decision. Code Simplicity reduces the chances of subtle errors.

As always, comments are welcome at hawthornepresscom@gmail.com.

As to why I have an email address on **gmail** instead of on my website, has to do with AT&T and their inability to handle the concept that not everyone is running a Microsoft OS and my stubborn insistence on running Linux.

Published by

Hawthorne-Press.com
10310 Moorberry Lane
Houston, Texas 77043, USA

Revision History

Date	By	Section	Changes
06/28/2016	C.E. Thornton	All	Initial Document
05/27/2018	C,E. Thornton	Copyright	Update Address