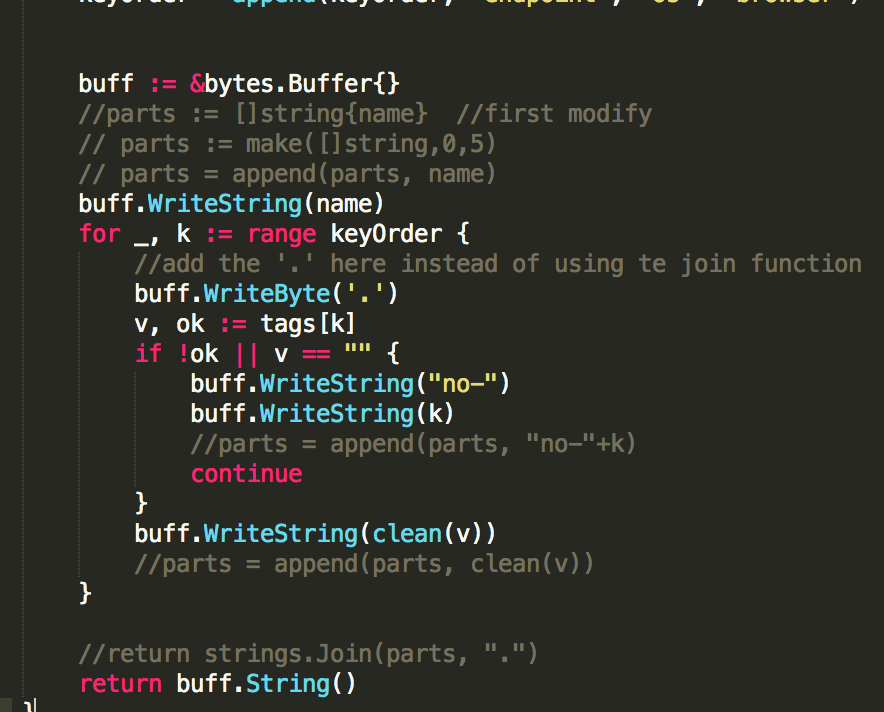
after reading the document, I modified the program like this:



I used a bytes.Buffer for string, just like the documents states above. Then I simply use writeString() instead of append so that there is no new intermediate object during this process.

Lest see how it perform:

ShuyanmatoMacBook-Pro:stats shuyanli$ go test -bench . -benchmem -cpuprofile prof.cpu

BenchmarkAddTagsToName-8    3000000       450 ns/op     208 B/op       10 allocs/op

We can found that the time reduce from 0.58us to 0.45us. The allocation time is not changed.

If we want to get a bigger performance, we need to reduce the allocation time.

    20ms      120ms     59: v, ok := tags[k]

         .          .     60: if !ok || v == "" {

         .          .     61: buff.WriteString("no-")

         .          .     62: buff.WriteString(k)

         .          .     63: //parts = append(parts, "no-"+k)

         .          .     64: continue

         .          .     65: }

      10ms      710ms     66: buff.WriteString(clean(v))

         .          .     67: //parts = append(parts, clean(v))

         .          .     68: }

         .          .     69:

         .          .     70: //return strings.Join(parts, ".")

         .      110ms     71: return buff.String()

         .          .     72:}

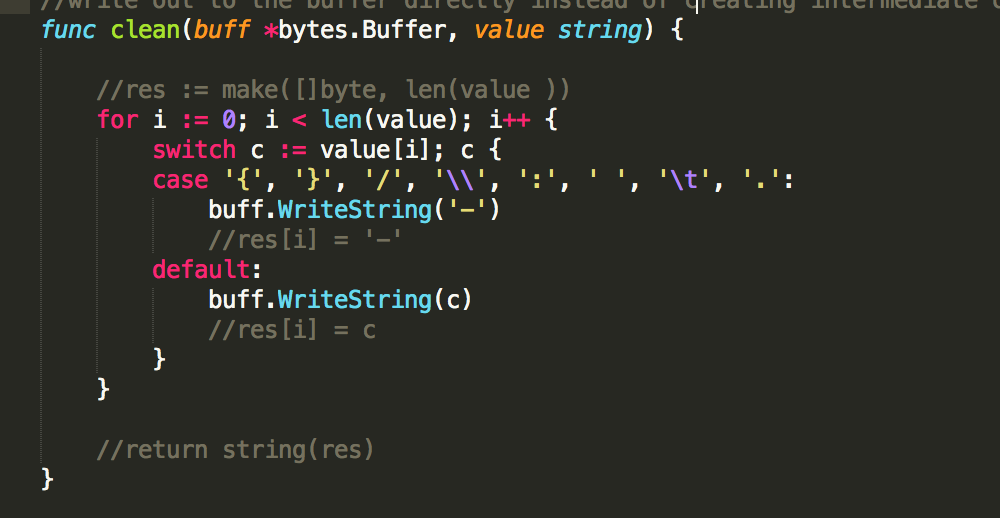
         .          .     73:

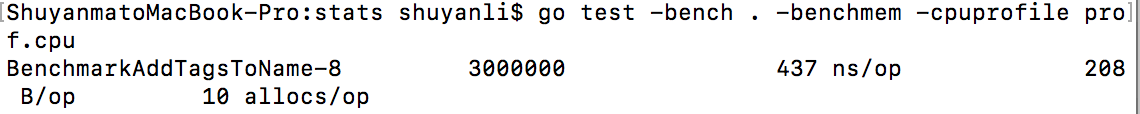
         .          .     74:

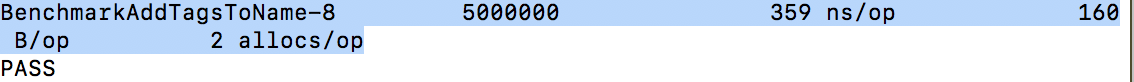
         .          .     75:

Now that the return function has significantly reduced its time. We might need to get back to clean function again.

In the clean function, I created a new slice type string, as a result, the program keep creating the new intermediate slice. Just like what I did before, I want to use buff.writebyte(), we don’t even need new string at all:



Lets check the performance:



It is amazing to find out that the allocation drop from 10 to 2!!

Then lets make it perfect: can we even make the allocation number to 0?

We need the help of memory allocation information, so instead of using –cpuprofile for the benchmark, we need –memprofile flag and a prof.mem file that will created by the runtime.

After searching golang allocation documents, I found the –alloc\_object flag:

“The parameter -alloc\_objects will visualize the number of allocated objects across the lifetime of the application.”

So we need –alloc\_objects flag as well:

5795873   11628844 (flat, cum)   100% of Total

         .          .     46: keyOrder = append(keyOrder, "host")

         .          .     47: }

         .          .     48: keyOrder = append(keyOrder, "endpoint", "os", "browser")

         .          .     49:

         .          .     50:

   5795873    5795873     51: buff := &bytes.Buffer{}

         .          .     52: //parts := []string{name}  //first modify

         .          .     53: // parts := make([]string,0,5)

         .          .     54: // parts = append(parts, name)

         .          .     55: buff.WriteString(name)

         .          .     56: for \_, k := range keyOrder {

         .          .     57: //add the '.' here instead of using te join function

..

..

..

         .          .     71:

         .          .     72: //return strings.Join(parts, ".")

         .    5832971     73: return buff.String()

         .          .     74:}

we can see that there are two places for the allocation

buff := &bytes.Buffer{} is what we use to create the buffer and buff.String() is we conver the buffer into a string. I don’t know how to get rid of the allocation here, even though I spent lots of time googled it.

I’ll do it tomorrow.

=============================update for buffer============================

After googled more, I found a way to optimize the buffer:

### sync.Pool:

var getBufferPool = sync.Pool{

New: func () interface{} {

return &Bytes.buffer{}

},

}

Truth be told, it took us a while to figure out what sync.Pool was actually for and why we would want to use it. It’s a free list that reuses allocations between garbage collection cycles, so that you don’t have to allocate another object that’s going to have to be collected by the garbage collector later. Each time a garbage collection cycle starts, it clears items out of the pool.

An example of how to use sync.Pool:

buf := getBufferPool.Get().(\*getBuffer)

defer getBufferPool.Put(buf)

key := append(buf.key[0:0], …)

First you declare a global sync.Pool object with a factory function, which in this case allocates a getBuffer struct and returns it. Instead of making a new getBuffer, we can get one from the pool. Pool.Get returns an empty interface, which we then type assert to the correct pointer type. When we’re done with it, we put it back in the pool. The result is that we don’t have to do even the one allocation to get the Buffer struct.

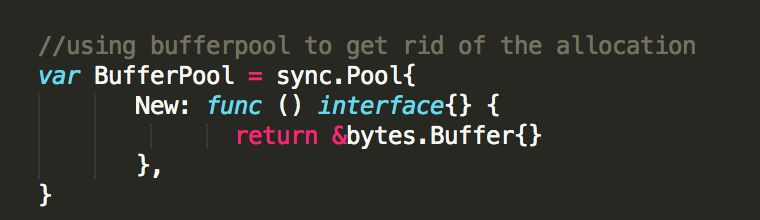
In conclusion, Pool can help to avoid the Buffer allocation.

One note: when using the pool, make sure it’s for the object that is very cheap to create!

As a result, it’s ok we use pool here.

Let’s try that out:

Definition:



Use it in function



And it failed ☹

--- FAIL: TestAddTagsToName (0.00s)

reporter\_test.go:41: addTagsToName(recvd, map[endpoint:hello os:OS X browser:Chrome]) got recvd.no-endpoint.no-os.no-browserrecvd.hello.OS-X.Chrome, expected recvd.hello.OS-X.Chrome

reporter\_test.go:41: addTagsToName(r.call, map[host:my-host-name endpoint:hello os:OS{}/ X browser:Chro\:me]) got recvd.no-endpoint.no-os.no-browserrecvd.hello.OS-X.Chromer.call.my-host-name.hello.OS----X.Chro--me, expected r.call.my-host-name.hello.OS----X.Chro--me

FAIL

exit status 1

=============================update=================================

it takes me a long time to fix this problem. Finally I found a very tiny line of code in golang files:

func Log(w io.Writer, key, val string) {

b := bufPool.Get().(\*bytes.Buffer)

b.Reset()

// Replace this with time.Now() in a real logger.

b.WriteString(timeNow().UTC().Format(time.RFC3339))

b.WriteByte(' ')

b.WriteString(key)

b.WriteByte('=')

b.WriteString(val)

w.Write(b.Bytes())

bufPool.Put(b)

}

Note that in this file, it has a reset() function that is not even introduced in sync(). I don’t know why we need that, but after I add it to my code:

ShuyanmatoMacBook-Pro:stats shuyanli$ go test -bench . -benchmem -memprofile prof.mem

BenchmarkAddTagsToName-8    5000000       395 ns/op       48 B/op       1 allocs/op

PASS

We can see here that the allocation time drop to 1!!

This shows one disadvantages of using benchmark, that it didn’t show the detail or any hint to my mistakes. Missing one line of code could simply crush all the function, as well as the benchmark

lets check the cpu svg file for handler only instead of the whole program. We find out that there is no function or library that takes majority of our time. This indicates that the function has been improved.

Lets compare with our original program(data from week 4 and 5)

258429 requests in 4.884710008s, 27.85MB read

Requests/sec: 48752.70

Transfer/sec: 5.70MB

Avg Req Time: 189.015µs

Fastest Request: 58.198µs

Slowest Request: 151.564736ms

Number of Errors: 0

And now we have:

ShuyanmatoMacBook-Pro:go-torch shuyanli$ go-wrk -d 5 http://localhost:9090/

Running 5s test @ http://localhost:9090/hello

  10 goroutine(s) running concurrently

260225 requests in 4.881132558s, 28.04MB read

Requests/sec: 56975.42

Transfer/sec: 5.75MB

Avg Req Time: 169.573µs

Fastest Request: 57.476µs

Slowest Request: 4.062266ms

Number of Errors: 0

Some conclusion:

Tools like go-torch is very easy to use in go profiling. The most important thing in profiling, just as the professor suggested in the lecture, is finding out the hot function. All the tools I used during the process makes it easy to find the hot function.

Lesson learnt:

I started to panic after saw so many functions in stackImpect and go svg statistics and graphs. Some tips for finding the hot function:

1. Check the svg file or FrameGraph, ignore all the read and write function, since that’s all modified by the runtime system designer already. There is nothing we can do about them. Instead, focus on some functions that are called by a lot of other functions. For example in our case the handler and addTagsToName function(that is design by the programmer).

3. using memory profile and cpu profile flag can help find out the memory/CPU occupation of the overall or some specific functions. Using a list func\_name can show the detail of how each line work for one function.

2. After finding the hot function, use the cpu and memory tools to check how it performance. There are some aspects that we can modify:

A. intermediate object.

For example, int a = 10;

b = a;

c = return\_value(b)

Bar(c)

Sometime the intermediate objects are obvious, but sometimes it’s hard to find them out, since we need to know the data structure of the language. For example, in go when we declare a slice, or call the slice library, like append, the system simply genarate a new slice and copy all the slice with smaller capacity to that new address. This will cause a worse GC performance as well as longer execution time. We need to get rid of all these intermediate objects.

B. cache

Cache is important for optimizing our code.

For example, making some hot local objects global is one easy approach to do the caching.

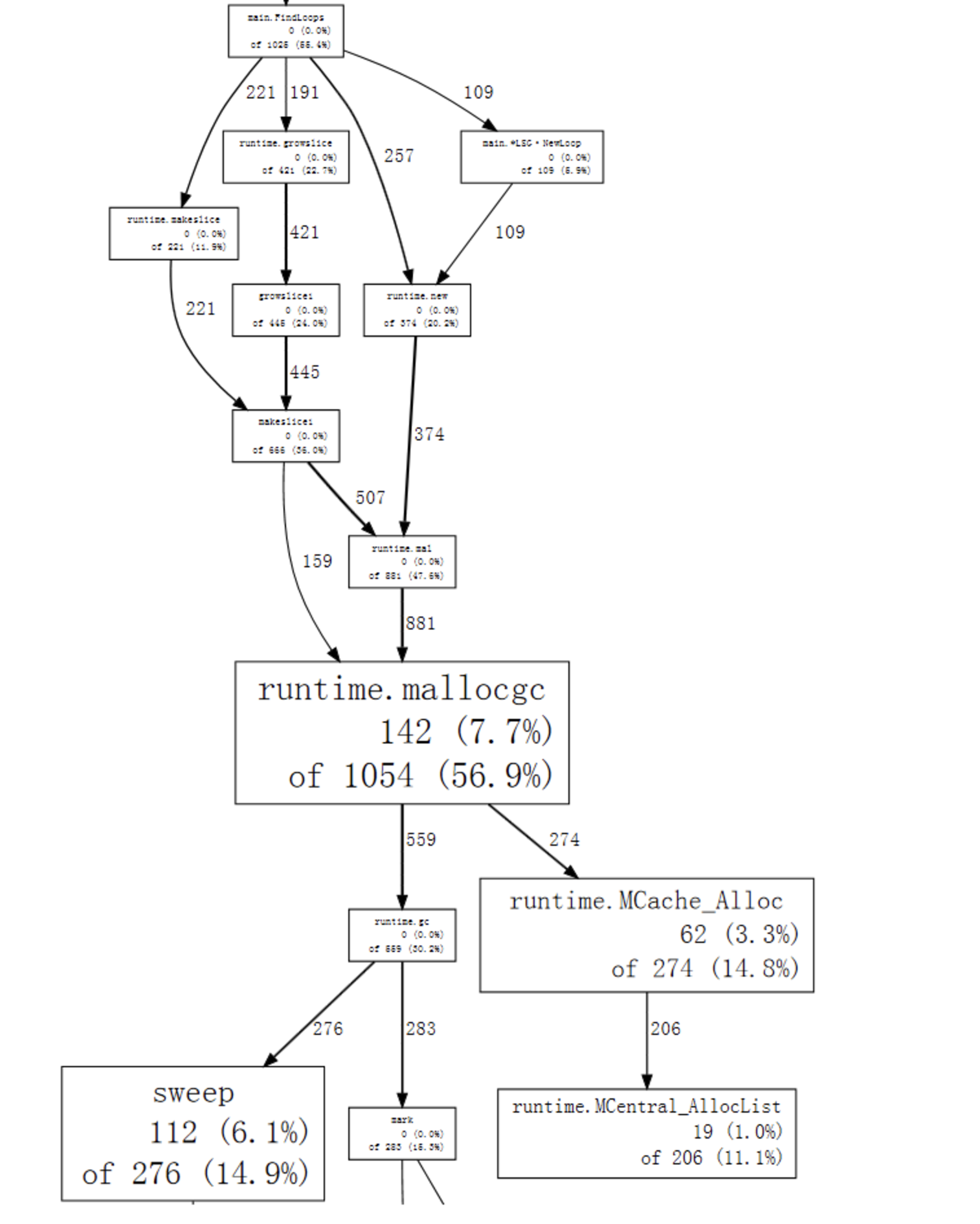
C. data structure

Sometime even if the algorithm is the same, which means the big O of one function is the same, the performance of this function varies a lot due to different data structures the function use. For example, using a map or an array(slice in go) can all store some data. However, in some cases, using an array instead of array, if possible, can reduce time the functions consume.

D. built-in library or packages

Sometimes when we call the library function, or some function from packages, the performance is not good, even if the library is a well-performance well-designed library. The reason is that most of the library are general. For some specific case, however, we might want to implement our own functions (or override the library functions) so that it can run better than the original library functions. They may not be widely-use, but they can be very fast.

E. avoid allocation

An allocation means a possible GC. This doesn’t mean GC is slow. GC is actually pretty fast. It’s just because every time we allocate, the run time has to do extra work to find some space, so allocations are not free. As a result, we want to minimize the allocation number. Check the path (hot path) where the number of allocation is high. Using the way introduced above to minimize the allocation number.

We can see here that the GC come from findloops function (find root).

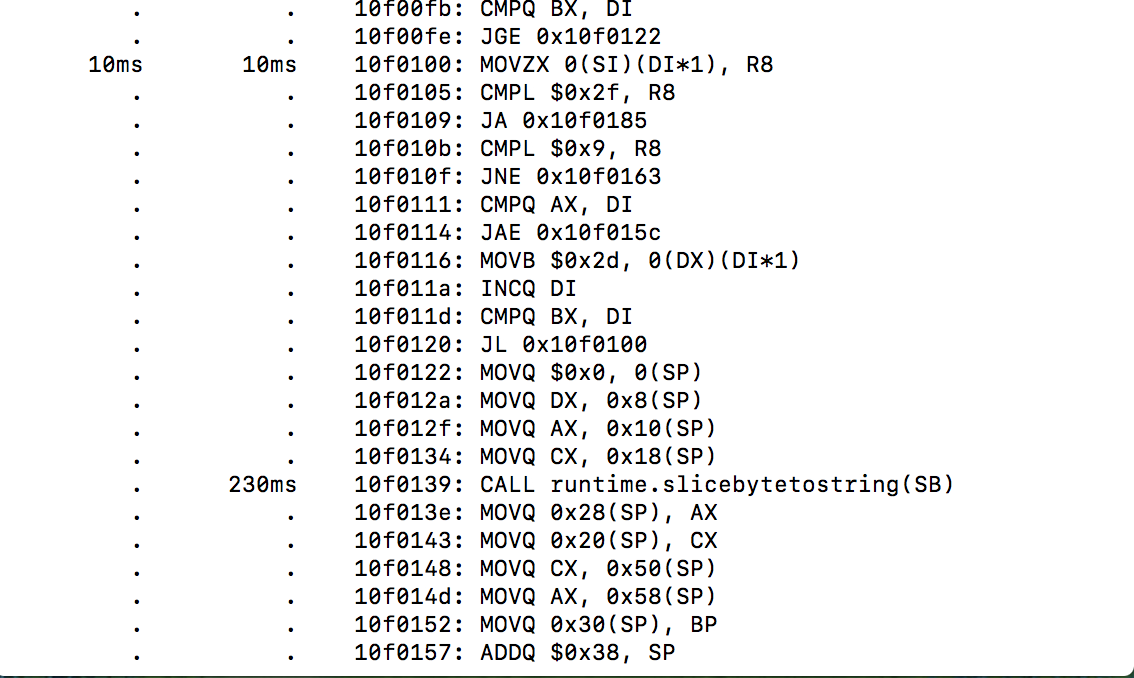
F. stackImpect

stackImpect is a very useful tool for the overall performance of a App or program. It can also show on the dashboard how the function or App is running during its process. However, when we want to test some small function inside the App, stackImpect doesn’t provide much help. Although it can show the details of the hierarchy structure (mostly the library the function calls), it’s very hard for a programmer to find out which functions call these library in the program.

G. try to tear the program apart.

Writing a benchmark can help figuring out the details of the program without keep running the program repeatedly. More important, sometime allocation number of one function is high due to the functions it calls. Sometimes these functions hide very deep. Using a stackImpect cannot help to find them out but using a benchmark can.

What’s more, disassemble the function can also help. For example:



We don’t need to understand every single line of code. All we need is to find CALL XXX. For example here is a CALL runtime.slicebytetostring() function that takes 230 ms to excecute. This is a very good initial point for us to start our optimization.