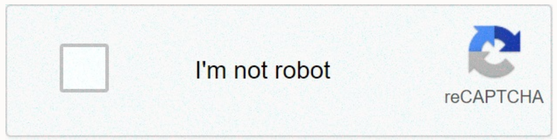

Continue

Golang anonymous function performance

Memory management can be difficult, to say the least. However, after reading the literature, one might be led to believe that all problems are solved: sophisticated automated systems that manage the life cycle of memory allocation free us from these burdens. However, if you've ever tried to fix the garbage collector of a JVM program or optimized the allocation model for a Go code base, you understand that this is far from a solved problem. Automated memory management usefully excludes a large class of errors, but that's only half the story. The hot paths of our software must be built so that these systems can work effectively. We found inspiration to share our learning in this area while building a high-speed service in Go called Centrifuge, which handles hundreds of thousands of events per second. The centrifuge is an essential part of Segment's infrastructure. Consistent and predictable behaviour is a requirement. An orderly, efficient and accurate use of memory is an important part of achieving this consistency. In this post we will cover common patterns that lead to inefficiency and production surprises related to memory allocation as well as practical ways to blunt or eliminate these issues. We will focus on the key mechanisms of the allocator that provide developers with a way to get a handle on their memory usage. Our first recommendation is to avoid premature optimization. Go provides excellent profiling tools that can point directly to heavy allocation parts of a code base. There's no reason to reinvent the wheel, so instead of taking readers through it here, let's refer to this excellent post on the official Go blog. It has a solid step-by-step procedure of using pprof for both CPU and allocation profiling. These are the same tools we use at Segment to find bottlenecks in our production GB code, and should be the first thing you achieve as well. Use the data to boost your optimization! The analysis of our EscapeGo automatically manages the memory allocation. This prevents a whole class of potential bugs, but it does not completely release the reasoning programmer on the mechanics of allocation. Since Go does not provide a direct way to manipulate allocation, developers need to understand the rules of this system so that it can be maximized for our own benefit. If you remember one thing from this whole post, that would be it: stack allocation is cheap and heap allocation is expensive. Now let's dive into what it really means. Go allocates memory two places: a global heap for dynamic allocations and a local stack for each goroutine. Go prefers allocation on the stack — most allocations in a given Go program will be on the stack. This is cheap because it only requires two CPU instructions: one to push on the stack for allocation, and the other to release from the stack. Unfortunately, not all data can use the memory allocated to the battery. The battery allocation requires that the life and memory footprint of a variable can be determined at the time of compiling. Compile. a dynamic on-the-job allocation occurs at the time of execution. malloc should look for a piece of free memory large enough to hold the new value. Later in the line, the garbage collector scans the pile for objects that are no longer referenced. It goes without saying that it is much more expensive than the two instructions used by the battery allocation. The compiler uses a technique called escape analysis to choose between these two options. The basic idea is to do the garbage collection work when compiling. The compiler tracks the scope of variables between code regions. It uses this data to determine which variables cling to a set of controls that prove that their lifespan is fully knowable at the time of execution. If the variable passes these controls, the value can be assigned to the stack. Otherwise, it is said to escape, and must be allocated heap. Escape analysis rules are not part of the Go language specification. For Go programmers, the easiest way to learn more about these rules is through experimentation. The compiler will give the results of the escape analysis by building with go build -gcflags '-m'. Let's take an example: package mainimport fmtfunc main (x: 42 fmt. Println(x) $go build -gcflags '-m'./main.go -order-line-arguments ./main./hand.go:7: x escapes to the heap ./hand.go:7: main ... Here argument does not escape that the variable x escapes the heap, which means that it will be dynamically assigned on the job at the execution time. This example is a bit confusing. In the eyes of man, it is immediately obvious that x will not escape the main function.) The compiler output does not explain why it thinks the value is escaping. For more details, pass the option -m several times, making the output more wordy:$ go build -gcflags '-m-m'./main.go-command-line-arguments./main./go:5: can't in main line: non-leaf function ./hand.go:7: x's escape to the heap./hand.go:7: argument (arg to ...) to ./main.go:7 ./main.go:7: from .... (indirection) to ./main.go:7 ./main.go:7: de ... argument (past to call [argument content escapes]) to ./main.go:7 ./main.go:7: main ... argument does not escapeAh, yes! This shows that x escapes because it is transmitted to a function argument that escapes - more on this later. The rules may continue to seem arbitrary at first, but after some trial and error with these tools, patterns begin to emerge. For those who are running out of time, here is a list of some models we found that usually cause variables to escape the heap: Sending pointers or values containing pointers to the channels. When compiling, there is no way to know which goroutine will receive the data on a channel. As a result, the compiler determine when this data will no longer be referenced. Store pointers or values containing pointers in a slice. An example of this is a type like [] chain. This always causes the contents of the slice to escape. Although the slice support panel may still be on the stack, the data escapes the heap. Back up tables of slices that are reassigned because a joint would exceed their capacity. In cases where the initial size of a slice is known at the time of compiling, it will begin its allocation on the stack. If the underlying storage of this instalment is to be expanded based on the data only known at the time of execution, it will be allocated on the job. Call methods on a type of interface. Method calls on interface types are a dynamic distribution — the actual concrete implementation to be used is only determinable at the time of execution. Consider a r variable with a type of io interface. Player. A call to r.Read(b) will cause both the r value and the support table of the bracket of the b-in to escape and thus be allocated on the job. In our experience, these four cases are the most common sources of mysterious dynamic allocation in Go programs. Fortunately, there are solutions to these problems! Then we'll go further in some concrete examples of how we've addressed memory inefficiencies in our production software. A few pointersThe basic rule is this: Pointers point to the data allocated on the job. Ergo, reducing the number of pointers in a program reduces the number of heap allocations. This is not an axiom, but we found that this was the common case in real-world go programs. It has been our experience that developers become competent and productive in Go without understanding the performance characteristics of values versus pointers. A common hypothesis derived from intuition goes something like this: copying values is expensive, so instead I'll use a pointer. However, in many cases, copying a value is much less expensive than overhead using a pointer. Why you might ask? The compiler generates controls when a pointer is carried over. The goal is to avoid memory corruption by running panic () if the pointer is zero. This is an additional code that must be run at the execution time. When data is transmitted by value, it cannot be zero. Pointers often have a bad location of reference. All the values used in a function are stored in memory on the stack. The locality of reference is an important aspect of the effective code. It greatly increases the chances of a value being hot in CPU caches and reduces the risk of a missed penalty during pre-locking. Copying objects in a cache line is roughly equivalent to copying a single pointer. Processors move memory between caching layers and the main memory on constant-sized cache lines. On x86 it's 64 bytes. In addition, Go uses a technique called Duff device to make common memory operations as very effective copies. The pointers should mainly be used to reflect property semantics and mutability. In practice, the use of pointers to avoid copies should be infrequent. Don't fall into the trap of premature optimization. It is good to develop a habit of transmitting data by value, only to return to the crossing pointers if necessary. An additional bonus is the increased security of eliminating zero. Reducing the number of pointers in a program can result that the garbage collector will skip the areas of memory that it can prove will not contain any pointers. For example, areas of the pile where the []byte rear slices are not digitized at all. This also applies to tables of struct types that do not contain fields with pointer types. Not only does reducing pointers result in less work for the garbage man, but it also produces a more cache-friendly code. Reading the memory moves the data from the main memory into the CPU caches. The caches are finished, so that another piece of data must be expelled to make room. The expelled data may still be relevant to other parts of the program. The resulting cover beat can cause unexpected and sudden changes in the behaviour of production services. Digging to point GoldReducing point use often means digging into the source code of the types used to build our programs. Our service, Centrifuge, maintains a queue of failed operations to try again like a circular buffer with a set of data structures that look like this: type retryQueue struct -[]retryItem // each bucket represents an interval of 1 second currentTime time. Time currentOffset int -type retryItem struct - id ksuid. KSUID // Element ID to try the time again. Exact time/time at which the item should be rejudged - The size of the outer table in the buckets is constant, but the number of items in the []retryItem slice contained will vary at the time of execution. The more retries there are, the more these slices will grow. By delving into the implementation details of each field of a retryItem, we learn that KSUID is a typical alias for [20]byte, which has no pointers, and can therefore be excluded. currentOffset is an int, which is a primitive fixed size, and can also be excluded. Then, looking at the implementation of time. Type of time[1]:type Struct of weather - dry int64 nsec int32 loc - Location // time zone structure - Time. The time struct contains an internal pointer for the location field. Its use in the retryItem type causes the GC to hunt pointers on these structs every time it crosses this area of the pile. We found that this is a typical case of cascading effects in unexpected circumstances. During normal operating failures are rare. Only a small amount of memory is used to store the retries. When failures suddenly increase, the number of items in the retry queue can increase by thousands per second, significantly increasing the garbage collector's workload. For this particular use case, time zone information on time. Time not necessary. These timestamps are kept in memory and are never serialized. Therefore, these data structures can be refactored to avoid this type entirely: type retryItem struct - id ksuid. KSUID nsec uint32 sec int64 'func' (article 'retryItem) time () time. Time to return time. Unix (item.sec, int64 (item.nsec)) -func makeRetryItem (id ksuid. KSUID is time. Time) retryItem - back retryItem id: id, nsec: uint32 (time. Nanosecond ()), dry: weather. Unix(), 'Now, the retryItem does not all pointers. This greatly reduces the load on the garbage collector because the total footprint of retryItem is known when compiling. Pass Me a SliceSlices are fertile ground for ineffective allocation behavior in hot code paths. Unless the compiler knows the size of the slice when compiling, the support tables for the slices (and cards!) are assigned on the job. Let's explore a few ways to keep the slices on the stack and avoid heap allocation. Centrifuge uses MySQL extensively. The overall effectiveness of the program depends heavily on the effectiveness of the MySQL pilot. After using pprof to analyze the behavior of the allocator, we found that the code that serializes time. The time values in the Go MySQL driver were particularly expensive. The profiler showed a large percentage of heap allocations were in code that serialized for a time. Time value so it can be sent on the thread to the MySQL server. This particular code called the format method () on time. Time, which returns a rope. Wait, aren't we talking about slices? Well, according to the official blog Go, a channel is just a read only slices of bytes with a little extra syntactic support of the language. Most of the same rules around the allowance apply! The profile tells us that a whopping 12.38% of the allocations took place during the execution of this format method. What does Format do? It turns out that there is a much more efficient way to do the same thing that uses a common model across the standard library. Although the format () method is easy and convenient, the code using AppendFormat() can be much easier on the allocator. Looking in the source code for the time package, we notice that all internal uses are AppendFormat () and non-format (). This is a pretty strong clue that AppendFormat () will give a more efficient behavior. In fact, the format method simply envelops the AppendFormat method: func (t Time) format (layout chain) string - const bufSize - 64 var b []byte max: - len(layout) - 10 if max - bufSize - var buf [bufSize]byte b-b-b-b[:0] - else - b - make([]byte, 0, max) - b - t.AppendFormat(b, layout) return chain (b) More importantly, AppendFormat() gives the programmer much more control over the allocation. You have to pass the slice to mutate rather than flip a chain that it allocates internally as a format.) The use of AppendFormat () instead of the format () allows the same operation to use a fixed size allocation[3] and is therefore eligible for battery placement. Let's look at the change we have upstream for go's MySQL driver in this PR. The first thing to notice is that var a [64]byte is a fixed size table. Its size is at the time of compiling and its use is fully extended to this function, so we can deduce that this will be assigned on the stack. However, this type cannot be forwarded to AppendFormat(), which accepts the type []byte. The use of the a[:0] notation converts the fixed size table into a type of slice represented by b that is supported by this table. This will pass the compiler's checks and will be assigned to the stack. More memory that would otherwise be dynamically assigned is transmitted to AppendFormat(), a method that passes the compiler's battery allocation checks itself. In the previous version, format () is used, which contains size allocations that cannot be determined at the time of compiling and are therefore not eligible for the stack allocation. The result of this relatively small change has greatly reduced allocations in this code path! Using the append model in the MySQL driver, an Append() method was added to the KSUID type in this PR. Converting our hot paths to use Append() on KSUID against a fixed-size buffer instead of the String() method saved an equally large amount of dynamic allocation. It should also be noted that the strconv package has equivalent appendage methods for converting channels that contain numbers into numerical types. Interface types and YouIt is quite common knowledge that method calls on interface types are more expensive than those on struct types. Method calls on interface types are executed by dynamic shipping. This greatly limits the compiler's ability to determine how this code will be executed at the time of execution. So far, we have had extensive discussions about developing the code so that the compiler can better understand its behavior at the time of compilation. Interface types throw it all away! Unfortunately, interface types are a very useful abstraction - they allow us to write more flexible code. A common case of interfaces used in the hot path of a program is the hash functionality provided by the standard library hash package. The hash package defines a set of generic interfaces and provides several concrete implementations. Let's take an example: main import package (fmt hash/fnv) func hashIt (chain) uint64 - h: fnv. New64a() h.Write([[]byte(in)) out: 'h.Sum64'() return out 'func main's: 'hello' fmt. Printf (The FNV64a hash of '%v' is '%v', s, hashIt(s)) - Building this code with the escape analysis output gives the following results:./foo1.go:9:17: inlining call to fnv. New64a ./foo1.go:10:16: ([]byte)(in) escapes the heap ./foo1.go:9:17: hash. Hash64 ()/foo1.go:9:17: 'fnv.s'2 escapes the heap ./foo1.go:9:17: moved to the pile: fnv.s-2 ./foo1.go:go 8:24: hashIt in does not escape ./foo1.go:17:13: escapes the heap ./foo1.go:17:59: hashIt (s) escapes the heap ./foo1.go:17:12: main ... Argument does not escape This means that the hash object, entry chain, and the representation []byte of the entrance will all escape the heap. In human eyes, these variables obviously do not escape, but the type of interface binds the hands of compilers. And it has no way to safely use concrete implementations without going through the hash package interfaces. So what is an efficiency-conscious developer to do? We encountered this problem during the construction of Centrifuge, which performs non-cryptographic hashing on small ropes in its hot paths. So we built the fasthash library as an answer. It was simple to build — the code that does hard work is part of the Standard library. fasthash simply repackages the library's standard code with a usable API without heap allocations. Let's look at the fasthash version of our test program: package mainimport (fmt github.com/segmentio/fasthash/fnv1a) func hashIt (chain) uint64 - out: fnv1a. HashString64 (in) back on 'func main's: 'hello' fmt. Printf (The FNV64a hash of '%v' is '%v', s, hashIt(s)) - What about the escape analysis output?. /foo2.go:9:24: hashIt in does not escape ./foo2.go:16:13: escapes the heap ./foo2.go:16:59: hashIt(s) escapes the heap ./foo2.go:16:12: main ... the argument does not escapeThe only remaining escapes are due to the dynamic nature of the fmt. Printf function(). Although we strongly prefer to use the standard library from an ergonomic point of view, in some cases it is worth going to such efforts for the efficiency of the allocation. A weird thingOur final anecdote is more fun than practical. However, this is a useful example for understanding the mechanics of the compiler's evasion analysis. When reviewing the standard library for covered optimizations, we came across a rather curious piece of code.// noescape hides a pointer from the escape analysis. noescape is // identity function, but the escape analysis does not think that the // output depends on the entry. noescape is unlined and currently // compiles up to zero instructions. USE IT CAREFULLY! go: nosplit func noescape (p dangerous. Dangerous pointer. Pointer x: uintptr (p) dangerous return. Pointer (x 0) - This feature will hide the past pointer of the compiler's evasion analysis feature. What does that really mean if? Well, let's set up an experiment to see!package mainimport (dangerous) type Foo struct - S - string -func (f -Foo) String () string 'return 'f.S' type FooTrick struct 'S dangerous. Func (f -FooTrick) String () string -return -return -(f.S)(f.S)-func NewFoo(s string) Foo - return Foo-S: FooTrick (s) string Pointer (and) - func noescape (p dangerous. Dangerous pointer. Pointer x: uintptr (p) dangerous return. Pointer (x - 0) 'main func's: 'hello' f1: 'NewFoo(s) f2: NewFooTrick(s) s1: f1. Chain () s2: f2. Chain () - This code contains two implementations that perform the same task: they hold a string and flip the chain contained using the String method(). However, the release of the compiler's escape analysis shows us that the FooTrick version does not escape!. /foo3.go:24:16: Escapes to the heap ./foo3.go:23:23: moved to the heap: s ./foo3.go:27:28: NewFooTrick does not escape ./foo3.go:28:2845: NewFooTrick and doesn't escape ./foo3.go:31:33: noescape p does not escape ./foo3.go:38:14: hand and not escape ./foo3.go:39:19: It doesn't escape./foo3.go:40:17: main f1 does not escape ./foo3.go:41:17: main f2 does not escapeTh two lines are the most relevant:: ./foo3.go:24:16: Escapes to the heap ./foo3.go:23:23: moved to the heap: sC is the compiler acknowledging that the NewFoo() NewFoo() takes a reference to the chain and stores it in the structuring, causing it to escape. However, no such output appears for the function takes a reference to the chain and stores it in the struct, causing it to escape. However, no such output appears for the NewFooTrick feature. If the noescape call () is removed, the evasion analysis moves the data referenced by the FooTrick struct to the heap. What's going on here? NewFooTrick function.) If the noescape call () is removed, the evasion analysis moves the data referenced by the FooTrick struct to the heap. What's going on here?func noescape (p dangerous. Dangerous pointer. Pointer x: uintptr (p) dangerous return. Pointer (x -0) - The noescape function () masks the dependency between the entry argument and the return value. The compiler does not think that p escapes via x because the uintptr () produces an opaque reference to the compiler. The name of the uintptr builtin type may lead one to believe that it is a type of pointer in good faith, but the pointer's pointer is just a whole that happens to be large enough to store a pointer. The last line of code builds and returns a dangerous one. Pointer value from a seemingly arbitrary whole value. Nothing to see here people! compiler. The name of the uintptr builtin type may lead one to believe that it is a type of pointer in good faith, but the pointer's pointer is just a whole that happens to be large enough to store a pointer. The last line of code builds and returns a dangerous one. Pointer value from a seemingly arbitrary whole value. Nothing to see here folks!noescape() is used in dozens of functions in the runtime package that use dangerous. Pointer. It is useful in cases where the author knows with certainty that the data referenced by a dangerous. Pointer does not escape, but the compiler naively thinks otherwise. Just to be clear - we do not recommend using such a technique. There is a reason why the referenced package is called dangerous and the source code contains the comment USE NEONMENT! TakeawaysBuilding a state-intensive Go service that must be efficient and stable in a wide range of real-world conditions has been a great learning experience for our team. Let's review our main learnings: don't optimize prematurely! Use the data to conduct your optimization work. Battery allocation is cheap, heap allocation is expensive. Battery allocation is cheap, heap allocation is expensive. Understanding the rules for analyzing escapes allows us to write more effective code. Pointers make stack allocation mostly impossible. Look for APIs that provide allocation control in code sections performance. Use interface types sparingly in hot paths. Understanding the rules for analyzing escapes allows us to write more effective code. Pointers make stack allocation mostly impossible. Look for APIs that provide allocation control in performance-critical sections of code. Use interface types sparingly in hot paths. We have used these relatively relatively techniques to improve our own Go code, and I hope others find these hard-earned learnings useful in building their own Go programs. Good coding, fellow gophers! [1] Time. The type of struct timetime.time has changed in Go 1.9. type of struct has changed in Go 1.9. [2] You may also have noticed that we have changed the order of nsec and dry fields, the reason is that because of the alignment rules, Go would generate a 4-byte padding after the KSUID. The nanosecond field happens to be 4 bytes, so placing it after the KSUID Go doesn't need to add more padding because the fields are already aligned. This increased the size of the data structure from 40 to 32 bytes, reducing the memory used by the retry queue by 20%. [3] The fixed-size tables in Go are similar to the slices, but have their size coded directly into their type signature. While most APIs accept slices and not tables, slices can be made from tables! Tables!