Golang 高性能编程指南

ChrisLinn

https://github.com/ChrisLinn

注意: 本文说的所有优化技巧只有在有需要时才有意义，如果性能瓶颈不在这里那么就不要做没有意义的优化。应该通过分析性能瓶颈和热点代码来决定是否要优化。

1 GC

1.1 避免指针

指针 GC 影响性能。[1]

什么时候会出现这种情况?

- 堆上申请了大量内存。(栈分配便宜、堆分配昂贵。[2])
- 就算想不在堆上申请, 把数据挪到堆下 (那就会不可避免的需要很多指针)。

什么场景需要警觉?

Strings, slices 和 `time.Time` 都含有指针。所以以下场景需要小心:

- 很多 strings
- `time.Time`
- Maps with slice values
- Maps with string keys

解决办法

- 如果这个 string 只可能是几个固定值的其中之一的话，不妨就用整数来表示。
- 如果要用 string 来存时间，不妨 parse 成整数来存储。
- 如果真的需要存很多字符串:
  - 如果在内存中连续，我们可以用 offset 来访问，就不需要指针了。

1.2 strings vs []bytes

string 的值是不可变的，`[]byte` 是可变的。string 可读性强，但是大多数 IO 是使用 `[]byte` 来完成的。

尽可能避免 *`[]byte`* 和 *string* 之间的转换。

[1] https://blog.gopheracademy.com/advent-2018/avoid-gc-overhead-large-heaps/
[2] go 中使用逃逸分析

使用 *string* 作为映射键 是很常见的，但我们通常有的是一个 `[]byte`. 编译器针对这种情况实现了特定的优化:

```
var m map[string]string
v, ok := m[string(bytes)]
```

但不会对下列代码进行优化:

```
key := string(bytes)
val, ok := m[key]
```

避免 *string concatenation* :

```
// version 1
s := request.ID
s += " " + client.Addr().String()
s += " " + time.Now().String()
r = s
```

```
// version 2
var b bytes.Buffer
fmt.Fprintf(&b, "%s %v %v", request.ID, client.Addr(), time.Now())
r = b.String()
```

```
// version 3
r = fmt.Sprintf("%s %v %v", request.ID, client.Addr(), time.Now())
```

```
// version 4
b := make([]byte, 0, 40)
b = append(b, request.ID...)
b = append(b, ' ')
b = append(b, client.Addr().String()...)
b = append(b, ' ')
b = time.Now().AppendFormat(b, "2006-01-02 15:04:05.999999999 -0700 MST")
r = string(b)
```

```
// version 5
var b strings.Builder
b.WriteString(request.ID)
b.WriteString(" ")
b.WriteString(client.Addr().String())
b.WriteString(" ")
b.WriteString(time.Now().String())
r = b.String()
```

bench test 发现 version 4 最快最省，version 1&5 一般，version 2&3 最差。

1.3 预分配 slice

`append` 很方便，但很浪费。

```
func main() {
       b := make([]int, 1024)
       b = append(b, 99)
       fmt.Println("len:", len(b), "cap:", cap(b))
}
```

会复制大量数据并创建大量垃圾。

如果知道事先知道 slice 的长度可以预先分配 slice ，并使用 index 赋值, 以避免复制。

1.4 减少 allocation

方法一 (bad):

```
func (r *Reader) Read() ([]byte, error)
```

始终分配缓冲区，从而给 GC 带来压力。

方法二 (good):

```
func (r *Reader) Read(buf []byte) (int, error)
```

接收一个 `[]byte` 缓冲区，填充给定的缓冲区，并返回读取的字节数。

1.5 *sync.Pool

`sync.Pool` 类型可以用来重用公共对象, **避免 new**，减少内存分配，降低 GC 压力。没有固定大小或最大容量。(注意：里面的值可能随时被回收。)

```
var pool = sync.Pool{
   New: func() interface{} {
      return make([]byte, 4096)
   },
}

func fn() {
       buf := pool.Get().([]byte) // takes from pool or calls New
       // do work
       pool.Put(buf) // returns buf to the pool
}
```

1.6 GC 参数调优

调整环境变量 `GOGC`。

1.7 避免 finalizer

2 goroutine

goroutine 创建成本低廉，Go 语言设计时就考虑到了成千上万个 goroutine 的情况。

但是，每个 goroutine 确实会消耗 goroutine 堆栈的内存，目前至少为 2k。就算什么也不做，1,000,000 个 goroutines 也要消耗 2GB 的内存。

如果不知道什么时候停止，就不要启动 goroutine，否则就是潜在的内存泄漏，因为 goroutine 会将其堆栈的内存以及可从堆栈访问的所有堆分配变量固定在堆栈上。

3 IO

3.1 本地文件 IO

Go 处理网络 IO 时会使用高效的操作系统轮询机制（kqueue, epoll, windows IOCP 等）。一个单一的操作系统线程将为许多等待的 goroutine 提供服务。**但对于本地文件 IO，Go 不会实现任何 IO 轮询**。`*os.File` 上的每个操作在进行中都会消耗一个操作系统线程。大量使用本地文件 IO 可能会导致您的程序产生数百或数千个线程，对磁盘系统**产生成百上千的并发 IO 请求**，可能超出您的操作系统所允许的范围。所以需要用 pool of worker goroutines 或 buffered channel 作为 semaphore。

```
var semaphore = make(chan struct{}, 10)

func processRequest(work *Work) {
       semaphore <- struct{}{} // acquire semaphore
       // process request
       <-semaphore // release semaphore
}
```

3.2 IO multipliers

写服务器代码要注意:

- 每个 request 产生多少 IO 事件? 且，是定值还是线性?
- 避免在请求的上下文中进行 IO，不要让用户等待读写磁盘。

3.3 流式 IO 接口

尽可能避免将数据读取到 `[]byte` 中并传递，不然可能将兆字节（或更多）的数据读取到内存中。这给 GC 带来了巨大压力，并会增加应用程序的平均延迟。

应该用 `io.Reader` 和 `io.Writer` 来构造处理管道，以限制每个请求使用的内存量。如果用了大量的 `io.Copy`，请考虑实现 `io.ReaderFrom`/`io.WriterTo`。这些接口效率更高，并且能避免将内存复制到临时缓冲区中。

3.4 超时

如果不知道所需的最长时间就不要 IO。用 `SetDeadline`, `SetReadDeadline`, `SetWriteDeadline` 对每个网络请求设置超时。

4 defer

`defer` 的开销很大, 因为它要记录 `defer` 参数的闭包。

```
defer mu.Unlock()
```

相当于

```
defer func() {
       mu.Unlock()
}()
```

如果计算量很小，defer 就会变得很昂贵，经典的例子是 mutex 解锁 struct 变量或 map 查找。在这种情况下，您可以选择避免 `defer`。(权衡性能 vs 可读性和维护性)。

5 重排 struct 中的成员

- 因为 padding 会影响 size [3]
- 另一篇相似的文章 [4]中也提到，Move hot fields to the top of the struct, 但是没有解释为什么，网上也没有相关的说法

6 CGO

cgo 调用类似于阻塞 IO，会消耗线程。不要大量循环调用 C 代码。

其实应该尽量避免使用 cgo:

- 如果 C 代码花费很长时间，那么 cgo 开销并不重要。
- 如果你用 cgo 调用非常短的 C 函数（其开销最明显），请在 Go 中重写该代码。
- 如果你大量循环调用了大量昂贵的 C 代码，那你为什么还要用 Go?

[3] https://dave.cheney.net/high-performance-go-workshop/gophercon-2019.html#rearrange_fields_for_better_packing

[4] https://dave.cheney.net/high-performance-go-workshop/gopherchina-2019.html#move_hot_fields_to_the_top_of_the_struct

7 其它

- 通过 `benchstat`, `pprof`, Execution Tracer 分析瓶颈。
- intrinsics。
- 始终使用最新发布的 Go 版本。
- 保持代码的简单性。
  - 编译器会对进行优化。
  - 代码越短，代码越小；这对于 CPU 的缓存很重要。
  - 可读意味着可靠。

A 延伸阅读

本文由以下文章总结而来:

- https://dave.cheney.net/high-performance-go-workshop/gophercon-2019.html
- https://blog.gopheracademy.com/advent-2018/avoid-gc-overhead-large-heaps/
- https://segment.com/blog/allocation-efficiency-in-high-performance-go-services/
- https://hackmd.io/@zkteam/goff[5]
- https://dave.cheney.net/2019/08/20/go-compiler-intrinsics
- https://blog.cloudflare.com/go-dont-collect-my-garbage/
- https://github.com/golang/go/wiki/SliceTricks[6]

[5] 介绍了一些位优化的小技巧。
[6] 介绍了一些过滤 & 去重的小技巧。