用golang开发系统软件的一些细节

作者:张富春(ahfuzhang)，转载时请注明作者和引用链接，谢谢!

- cnblogs博客
- zhihu
- Github
- 公众号:一本正经的瞎扯

众所周知，golang非常适合用于开发后台应用，但也通常是各种各样的应用层软件。

开发系统软件， 目前的首选还是C++, C, rust等语言。相比应用软件，系统软件需要更加稳定，更加高效。其维持自身运行的资源消耗要尽可能小，然后才可以把更多CPU、内存等资源用于业务处理上。简单来说，系统软件在CPU、内存、磁盘、带宽等计算机资源的使用上要做到平衡且极致。

golang代码经过写法上的优化，是可以达到接近C的性能的。现在早已出现了很多用golang完成的系统软件，例如很优秀的etcd, VictoriaMetrics等。VictoriaMetrics是Metric处理领域优秀的TSDB存储系统， 在阅读其源码后，结合其他一些golang代码优化的知识，我将golang开发系统软件的知识总结如下：

golang的第一性能杀手：GC

个人认为GC扫描对象、及其GC引起的STW，是golang最大的性能杀手。本小节讨论优化golang GC的各种技巧。

压舱物ballast

下面一段神奇的代码，能够减少GC的频率，从而提升程序性能：

```
func main(){
    ballast := make([]byte, 10*1024*1024*1024)
    runtime.KeepAlive(ballast)
    // do other things
}
```

其原理是扩大golang runtime的堆内存，使得实际分配的内存不容易超过堆内存的一定比例，进而减少GC的频率。GC的频率低了，STW的次数和时间也就更少，从而程序的性能也提升了。

具体的细节请参考文章：

- 一个神奇的golang技巧：扩大heap内存来降低gc频率 (本人)
- Go Ballast 让内存控制更加丝滑

堆外内存

众所周知，golang中分配太多对象，会给GC造成很大压力，从而影响程序性能。
那么，我在golang runtime的堆以外分配内存，就可以绕过GC了。
可以通过mmap系统调用来使用堆外内存，具体请见：《Go Mmap 文件内存映射简明教程》
对于堆外内存的应用，在此推荐一个非常经典的golang组件：fastcache。具体请看这篇我对fastcache的分析文章：《介绍一个golang库：fastcache 》。

> 也需要注意，这里有个坑：
> 如果使用mmap去映射一个文件，则某个虚拟地址没有对应的物理地址时，操作系统会产生缺页终端，并转到内核态执行，把磁盘的内容load到page cache。如果此时磁盘IO高，可能会长时间的阻塞.....进一步地，导致了golang调度器的阻塞。

对象复用

对象太多会导致GC压力，但又不可能不分配对象。因此对象复用就是减少分配消耗和减少GC的释放消耗的好办法。

下面分别通过不同的场景来讨论如何复用对象。

海量微型对象的情况

假设有很多几个字节或者几十个字节的，数以万计的对象。那么最好不要一个个的new出来，会有两个坏处：

- 对象的管理会需要额外的内存，考虑内存对齐等因素又会造成额外的内存浪费。因此海量微型对象需要的总内存远远大于其自身真实使用的字节数；
- GC的压力源于对象的个数，而不是总字节数。海量微型对象必然增大GC压力。

海量微型对象的影响，请看我曾经遇到过的这个问题：《【笔记】对golang的大量小对象的管理真的是无语了.....》

因此，海量微型对象的场景，这样解决：

- 分配一大块数组，在数组中索引微型对象
- 考虑fastcache这样的组件，通过堆外内存绕过GC

当然，也有缺点：不好缩容。

大量小型对象的情况

对于大量的小型对象，sync.Pool是个好选择。

推荐阅读这篇文章：《Go sync.Pool 保姆级教程》

sync.Pool不如上面的方法节省内存，但好处是可以缩容。

数量可控的中型对象

有的时候，我们可能需要一些定额数量的对象，并且对这些对象复用。

这时可以使用channel来做内存池。需要时从channel取出，用完放回channel。

slice的复用

fasthttp, VictoriaMetrics等组件的作者 valyala可谓是把slice复用这个技巧玩上了天，具体可以看fasthttp主页上的 Tricks with `[]byte` buffers这部分介绍。

概要的总结起来就是：[]byte这样的数组分配后，不要释放，然后下次使用前，用`slice=slice[:0]`来清空，继续使用其上次分配好的cap指向的空间。

这篇中文的总结也非常不错：《fasthttp对性能的优化压榨》

valyala大神还写了个 bytebufferpool，对`[]byte`重用的场景进行了封装。

避免容器空间动态增长

对于slice和map而言，在预先可以预估其空间占用的情况下，通过指定大小来减少容器操作期间引起的空间动态增长。特别是map，不但要拷贝数据，还要做rehash操作。

```
func xxx(){
  slice := make([]byte, 0, 1024)  // 有的时候，golangci-lint会提示未指定空间的情况
  m := make(map[int64]struct{}, 1000)
}
```

大神技巧：用slice代替map

此技巧源于valyala大神。

假设有一个很小的map需要插入和查询，那么把所有key-value顺序追加到一个slice中，然后遍历查找——其性能损耗可能比分配map带来的GC消耗还要小。

1. map变成slice，少了很多动态调整的空间
2. 如果整个slice能够塞进CPU cache line，则其遍历可能比从内存load更加快速

具体请见这篇：《golang第三方库fasthttp为什么要使用slice而不是map来存储header？》

避免栈逃逸

golang中非常酷的一个语法特点就是没有堆和栈的区别。编译器会自动识别哪些对象该放在堆上，哪些对象该放在栈上。

```
func xxx() *ABigStruct{
  a := new(ABigStruct)  // 看起来是在堆上的对象
  var b ABigStruct      // 看起来是栈上的对象
  // do something
  // not return a   // a虽然是对象指针，但仅限于函数内使用，所以编译器可能把a放在栈上
  return &b   // b超出了函数的作用域，编译器会把b放在堆上。
}
```

valyala大神的经验：先找出程序的hot path，然后在hot path上做栈逃逸的分析。尽量避免hot path上的堆内存分配，就能减轻GC压力，提升性能。

> fasthttp首页上的介绍：
>
> Fast HTTP package for Go. Tuned for high performance. **Zero memory allocations in hot paths.** Up to 10x faster than net/http

这篇文章介绍了侦测栈逃逸的方法：

> 验证某个函数的变量是否发生逃逸的方法有两个：
>
> - go run -gcflags "-m -l" (-m打印逃逸分析信息，-l禁止内联编译)；例：
>
> ```
> → testProj go run -gcflags "-m -l" internal/test1/main.go
> # command-line-arguments
> internal/test1/main.go:4:2: moved to heap: a
> internal/test1/main.go:5:11: main make([]*int, 1) does not escape
> ```
>
> - go tool compile -S main.go | grep runtime.newobject（汇编代码中搜runtime.newobject指令，该指令用于生成堆对象）,例：
>
> ```
> → testProj go tool compile -S internal/test1/main.go | grep newobject
>  0x0028 00040 (internal/test1/main.go:4) CALL  runtime.newobject(SB)
> ```
>
> ——《golang 逃逸分析详解》

逃逸的场景，这篇文章有详细的介绍：《go逃逸场景有哪些》

CPU使用层面的优化

声明使用多核

强烈建议在main.go的import中加入下面的代码：

```
import _ "go.uber.org/automaxprocs"
```

特别是在容器环境运行的程序，要让程序利用上所有的CPU核。

在k8s的有的版本(具体记不得了)，会有一个恶心的问题：容器限制了程序只能使用比如2个核，但是`runtime.GOMAXPROCS(0)`代码却获取到了所有的物理核。这时就导致进程的物理线程数接近逻辑CPU的个数，而不是容器限制的核数。从而，大量的CPU时间消耗在物理线程切换上。我曾经在腾讯云上测试过，这种现象发生时，容器内单核性能只有物理机上单核性能的43%。

因此，发现性能问题时，可以通过`ls /proc/$(pidof xxx)/tasks | wc`来查看进程的物理线程数，如果这个数量远远高于从容器要求的核数，那么在部署的时候建议加上环境变量来解决：`export -p GOMAXPROC=2`

golang不适合做计算密集型的工作

协程的调度，本质上就是一个一直在运行的循环，不断的调用各个协程函数。然后协程函数在适当的时机保存上下文，放弃执行，把程序流程再转回到主循环。

这里有几个要点：

- 主循环来负责唤起每个协程函数，如果存在很多协程函数，轮一遍的周期很长。
- 协程函数一定不能阻塞
- 协程函数也不能阻塞太长的时间
- 主循环唤起协程函数，以及协程函数切换回主循环是有开销的。协程越多，开销越大

因此，每个协程函数：在做IO操作的时候一定会切换回主循环，编译器也会在协程函数内编译进去可以切换上下文的代码。新版的golang runtime还存在强制调度的机制，如果某个正在执行的协程不会退出，会强制进行切换。

由于存在协程切换的调度机制，golang是不适合做计算密集型的工作的。例如：音视频编解码，压缩算法等。以zstd压缩库为例，golang版本的性能不如cgo的版本，即便cgo调用存在一定开销。(我举的例子比较极端，当需要让golang的性能达到与C同一个级别时，标题的结论才成立。)

克制使用协程数

由runtime的调度器原理可知，协程数不是越多越好，过多的协程会占用很多内存，且占用调度器的资源。

如何克制的使用协程，请参考我的这篇文章：《VictoriaMetrics中的golang代码优化方法》

总结起来就是：

- 最合适情况：核心的工作协程的数量，与可用的CPU核数相当。
- 区分IO协程和工作协程，把繁重的计算任务交给工作协程处理。

协程优先级机制

关于优先级的案例，请参考我写的这篇文章：《VictoriaMetrics中协程优先级的处理方式》

当业务环境需要区分重要和不太重要的情况时，要通过一定的机制来协调协程的优先级。比如存贮系统中，写入的优先级高于查询，当资源受限时，要让查询的协程主动让出调度。

不能让调度器来均匀调度，不能创建更多的某类协程来获得争抢优势。

> 要深入理解golang的runtime，推荐阅读yifhao同学的这篇文章：《万字长文带你深入浅出 Golang Runtime》

并发层面

```
并发层面的问题是通用性的知识，与语言的特性并无直接的关系。本节列出golang中处理并发的惯用方法，已经对golang的并发处理很熟悉的同学可以跳过本小节。
```

锁

关于锁的使用，VictoriaMetrics这个开源组件中有很多经典的案例。也可以移步参考这篇文章的总结：《VictoriaMetrics中的golang代码优化方法》(本人)

尽量不加锁

以生产者-消费者模型为例：如果多个消费者之间可以做到互不关联的处理业务逻辑，那么应该尽量避免他们之间产生关联。其业务处理过程中需要的各个对象，宜各自一份。

对数据加锁，而不是对过程加锁

拥有JAVA经验的同学要特别小心这一点：JAVA中，在方法上加上个关键字就能实现互斥，但这是非常不好的设计方式。只需要对并发环境下产生冲突的变量加锁即可，代码及其不冲突的变量都是不必要加锁的。

更进一步，如果存在多个冲突的变量，且在程序中不同的位置发生冲突，那么可以对特定的一组变量定义一个特定的锁，而不是使用一把统一的大锁来进行互斥——**尽量使用多个锁，让冲突进一步减小**。

读多写少的场景考虑读写锁

某些读写的场景下，读是可以并发的，而写是互斥的。这种场景下，读写锁是比互斥锁更好的选择。

原子操作

基础的原子操作技巧

```
var value int64 = 0

atomic.AddInt64(&value, 1)            // 原子加
atomic.AddInt64(&value, -1)           // 原子减

var n uint64 = 1
atomic.AddUint64(&n, 1)
atomic.AddUint64(&n, ^uint64(0))    // 原子减1，无符号类型，使用反码来减

newValue := atomic.LoadInt64(&value) // 内存屏障，避免乱序执行，并且同步CPU cache和内存
atomic.StoreInt64(&value, newValue)

oldValue := atomic.SwapInt64(&value, 0) // 获取当前值，并清零
```

原子操作就能搞定的并发场景，就不要再使用锁。

自旋锁

golang里面哪来的自旋锁?

其实我们可以自己写一个:

```
var globalValue int64 = 0
func xxx(newValue int64){
  oldValue := atomic.LoadInt64(&globalValue)  // 相当于使用 memory barrier 指令，避免指令乱序
  for !atomic.CompareAndSwapInt64(&globalValue, oldValue, newValue) {  // 自旋等待，直到成功
    oldValue = atomic.LoadInt64(&globalValue)  // 失败后，说明那一瞬间值被修改了。需要重新获取最新的值
    // 其他数值操作的准备
  }
}
```

以上是无锁数据结构的经典套路。

atomic.Value: 用于并发场景下需要切换的对象

有的对象很基础，可能需要频繁访问，且有时又会发生引用的切换。比如程序中的全局配置，很多地方都会引用，有时配置更新后，又会切换为最新的配置。

这种情况下，加锁的成本太高，不加锁又会带来风险。因此，使用sync.Value来保存全局配置的数据是个不错的选择。

```
type Configs map[string]string

var globalConfig atomic.Value

func GetConfig() Configs {
  v, ok := globalConfig.Load().(Configs)
  if ok{
    return v
  }
  return map[string]string{}
}

func SetConfig(cfg Configs){
  globalConfig.Store(cfg)
}
```

并发容器

sync.Map

并发map设计得很精巧，用起来也很简单。不过很可惜，sync.Map没有那么快，要避免将sync.Map用在程序的关键路径上。

当然，我上述的观点的区分点是：这是业务程序还是系统程序，如果是系统程序，尽量不要用。我实际使用中发现，sync.Map会导致CPU消耗高，且GC压力增大。

RoaringBitmap(或类似实现)

对某些特定的场景，可以做到很少的锁，很小的内存，比如存储大量UINT64类型的集合这一点，RoaringBitmap是个非常好的选型。

VictoriaMetrics中有一个RoaringBitmap实现的组件，叫做uint64set。具体介绍请见：《vm中仿照RoaringBitmap的实现：uint64set》(本人)。

channel

channel当然也算一种并发容器，其本质上是无锁队列。

需要注意两点：

- 为了在多读多写条件下维持队列的数据结构，通常通过CAS+自旋等待来操作关键数据。

  因此在大并发下，入队出队操作是串行化的，CAS失败+自旋重试又会带来cpu使用率升高。

  同样的，channel没有那么快。要避免在剧烈竞争的环境下使用channel。

- 通常会使用channel来做生产者-消费者模式的并发结构。数据数据可以按照一定的规律分区，则可以考虑每个消费者对应一个channel，然后生产者根据数据的key来决定放到哪个channel。这样本质上减缓了锁的竞争。

其他

用sync.Once来懒惰初始化

有的运算结果，有一定概率用到，但是又不必每次都计算。这种情况下，使用sync.Once来懒惰初始化是个好办法：

```
var once sync.Once
var globalXXX *XXX
func GetXXX() *XXX{
  once.Do(func(){
    globalXXX = getXXX()
  })
  return globalXXX
}
```

不安全代码

string与[]byte的转换

string与slice的结构本质上是一样的，可以直接强制转换：

```
import (
  "reflect"
  "unsafe"
```

```
)

// copy from prometheus source code

// NoAllocString convert []byte to string
func NoAllocString(bytes []byte) string {
  return *(*string)(unsafe.Pointer(&bytes))
}

// NoAllocBytes convert string to []byte
func NoAllocBytes(s string) []byte {
  strHeader := (*reflect.StringHeader)(unsafe.Pointer(&s))
  sliceHeader := reflect.SliceHeader{Data: strHeader.Data, Len: strHeader.Len, Cap:
strHeader.Len}
  return *(*[]byte)(unsafe.Pointer(&sliceHeader))
}
```

上面的代码可以避免string和[]byte在转换的时候发生拷贝。

> 注意：转换后的对象一定要立即使用，不要进一步引用到更深的层次中去。牢记这是不安全代码，谨慎使用。

强制类型转换

懂C的人，请绕过……

例如一个[]int64的数组要转换为[]uint64的数组，使用个指针强制转换就行了。

```
package main

import (
  "testing"
  "unsafe"
)

func TestConvert(t *testing.T) {
  int64Slice := make([]int64, 0, 100)
  int64Slice = append(int64Slice, 1, 2, 3)
  uint64Slice := *(*[]uint64)(unsafe.Pointer(&int64Slice))
  t.Logf("%+v", uint64Slice)
}
```

还有一种使用场景，要比较两个大数组是否完全一样：可以把数组强制转换为[]byte，然后使用bytes.Compare()。相当于C中的memcmp()函数。

类似的操作还很多，推荐这篇文章：《深度解密Go语言之unsafe》

```
模糊记得一个golang（或是rust）的原则：
普通开发者可以使用安全代码来无顾虑的使用，高手把不安全代码包装成安全代码来提供高性能组件。
```

数组越界检查的开销

相比C的数组访问，为什么golang可以做到很安全?

答案是编译器加了两条越界检查的指令。每次通过下标访问数组，就像这样：

```go
if index<0 || index>=len(slice){
  panic("out of index")
}
return slice[index]
```

这两条越界检查指令是有开销的，请看我的测试：《golang中数组边界检查的开销大约是1.87%~3.12%》

所以，当某些位置使用类似查表法的时候，可以用不安全代码绕过越界检查：

```go
slice := make([]byte, 1024*1024)
offset = 100
b := (*(*byte)(unsafe.Pointer(uintptr(unsafe.Pointer(&arr[0])) + uintptr(offset))))
```

编译/链接阶段

使用尽量新的golang版本

理论上，每个新版的golang，都有一定编译器优化的提升。

编译参数

> - -X importpath.name=value 编译期设置变量的值
> - -s disable symbol table 禁用符号表
> - -w disable DWARF generation 禁用调试信息
>
> ——《golang编译参数ldflags》

理论上说 -s -w加上后，代码段的长度会减小，理论上会提高CPU代码cache的利用率。(还未亲自测试过)

使用runtime中的非导出函数

runtime中有的底层函数是汇编实现的，性能很高，但是不是export类型。

这时候可以用链接声明来使用这些函数：

```
//go:noescape
//go:linkname memmove runtime.memmove
//goland:noinspection GoUnusedParameterfunc memmove(to unsafe.Pointer, from
unsafe.Pointer, n uintptr)
func memmove(to, from unsafe.Pointer, n uintptr)

// 通过上面的声明后，就可以在代码中使用底层的memmove函数了。这个函数相当于c中的memcpy()
```

更具体的细节请参考文章：

- 《Go的2个黑魔法技巧》(腾讯 pedrogao)
- How to call private functions (bind to hidden symbols) in GoLang

函数内联

golang的小函数默认就是内联的。

可以通过函数前的注释 `//go:noinline` 来取消内联，不过似乎没有理由这么做。

关于函数内联的深层知识还是值得学习的，推荐这篇文章：《详解Go内联优化》

可以关注文章中的这个内联优化技巧：

> 可通过 `-gcflags="-l"` 选项全局禁用内联，与一个 `-l` 禁用内联相反，如果传递两个或两个以上的 `-l` 则会打开内联，并启用更激进的内联策略。

泛型

golang 1.18正式发布了泛型。

泛型可以让之前基于反射的代码变得更加简单，很多type assert的代码可以去掉；基于interface的运行期动态分发，也可以转成编译期决定。

由于对具体的类型产生了具体的代码，理论上指令cache命中会提高，分支预测失败会降低，

不过，对于有一定体量的golang团队而言，泛型的引入要考虑的问题比较多：如何避免滥用，如何找到与之匹配的基础库?

在整个团队的能力还没准备好迎接泛型以前，使用工具生产代码的 `产生式编程` 或许是更容易驾驭的方法。

API使用

反射

编译期决定当然是好于运行期决定的。

我的建议是：

- 能不用就不用，可以用下面的方法代替：
  - 泛型
  - 代码生成(产生式编程)
- 非得要用
  - 缓存反射的到的结果

```
有的场景下，标准库提供的API不够好。下面列举一些自己认识的fast-xx组件。
```

fasttime组件，低精度的time.Now()

源码请见：https://github.com/VictoriaMetrics/VictoriaMetrics/blob/master/lib/fasttime/fasttime.go

原理就是创建协程每秒一次获取 time.Now()，然后一秒以内取时间戳就只是访问全局变量。

我测试过：性能比直接使用time.Now()快三倍左右。

fastrand，绕开rand库的锁

源码请见：https://github.com/valyala/fastrand

超长字符串输出的优化：quicktemplate

假设一次要输出几兆字节的JSON字符串，如何优化性能?

VictoriaMetrics中的vm-select就遇到了这个问题，当一个大查询需要返回很多的metrics数据的时候，其输出的json的体积非常可观。

如果把数据先放到一个大数组，再使用json.Marsharl，则一方面要频繁申请释放内存，另一方面会带来内存使用量的剧烈抖动。vm-select的解决方式是使用quicktemplate库——把json看成是字符串流的输出。

具体代码请看：https://github.com/valyala/quicktemplate

其他

总有很多人想把某个细分领域做到极致：

- fasthttp
- fastpb
- sonic - json解析

欢迎推荐更好好用的库给我，谢谢。

其他高级主题

汇编/SIMD

一些涉及大量计算的热点，可以采用汇编来优化。

golang使用plan 9汇编的语法，门槛还是比较高的。（经过半年断断续续的学习，我已经知道怎么看注释了）

所幸的是，懂C的人可以通过工具一步步把C代码翻译成plan 9汇编。

我自己做了个尝试：《玩一玩golang汇编》(师从于这篇：《Go的2个黑魔法技巧》)

```
注意：https://github.com/Maratyszcza/PeachPy这个库的代码翻译能力有限，我就发现有的代码无法翻译的情况。
且，只支持amd64平台下的翻译。
如果大家遇到更好的汇编翻译工具，请推荐给我。
```

使用汇编的最佳理由是SIMD指令集。

通常，一条指令只处理一条数据。而simd中，一条指令可以处理多条数据，当数据由多个128bit或者256bit构成的时候，使用SIMD指令可以取得较好的收益。

以strcmp()函数为例，传统的写法是逐个字符比较；而使用SIMD的话，可以把连续的16字节或者32字节(AVX2) load 到寄存器中，然后一次性比较。

这块知识体系较为庞大，有兴趣请自行搜索。

推荐文章：《golang 汇编》

JIT技术

当前流行的OLAP数据库clickhouse为何性能如此卓绝？其两个核心技术点就是SIMD和JIT。

> 在计算机技术中，**即时编译**（英语：just-in-time compilation，缩写为**JIT**；又译**及时编译**[1]、**实时编译**[2]），也称为**动态翻译**或**运行时编译**[3]，是一种执行计算机代码的方法，这种方法涉及在程序执行过程中（在执行期）而不是在执行之前进行编译。[4]通常，这包括源代码或更常见的字节码到机器码的转换，然后直接执行。实现JIT编译器的系统通常会不断地分析正在执行的代码，并确定代码的某些部分，在这些部分中，编译或重新编译所获得的加速将超过编译该代码的开销。
>
> JIT编译是两种传统的机器代码翻译方法——提前编译（英语：ahead-of-time compilation）（AOT）和解释——的结合，它结合了两者的优点和缺点。[4]大致来说，JIT编译，以解释器的开销以及编译和链接（解释之外）的开销，结合了编译代码的速度与解释的灵活性。JIT编译是动态编译的一种形式，允许自适应优化（英语：adaptive optimization），比如动态重编译和特定于微架构的加速[nb 1][5]——因此，在理论上，JIT编译比静态编译能够产生更快的执行速度。解释和JIT编译特别适合于动态编程语言，因为运行时系统可以处理后期绑定（英语：Late binding）的数据类型并实施安全保证。
>
> ——维基百科-即时编译

JIT在JAVA圈耳熟能详，通常指把字节码编译为机器码。但是golang没有机器码，所以golang中JIT并不用于字节码翻译。

我觉得golang中的JIT可以这样定义：**为特定的功能点，动态生成特定的机器码，以提高程序性能。**

关于如何实现一个golang中的JIT，可以阅读这篇：《使用 Go 语言写一个即时编译器(JIT)》

```
像把大象放进冰箱里一样总结一下：
1.把一些机器码，放到一个数组中；（已经知道这些机器码是干啥的了）
2.使用mmap系统调用分配一块内存，把内存设置为可执行，把上面的机器码拷贝进去；（然后这片内存就成为了程序的代码段）
3.定义一个函数指针指向mmap的内存；
4.执行函数。
```

也有golang库提供动态生成机器码的能力：https://github.com/goccy/go-jit。支持的指令有限，而且，猜测没人愿意这么写代码。

（读者一定在想这么鸡肋的东西介绍给我干啥.....）

golang的JIT的一个精彩应用是bytedance开源的sonic库，从测试数据来看，应该是golang圈子里最快的JSON解析库。

怎么做到的呢?

例如有这样一个json:

```
{"a":123, "b":"abc"}
```

要把它解析到结构体：

```go
type Data struct{
  A int64
  B string
}
```

一般来说，这个过程需要很多的判断：源字段名是什么？源字段什么类型？目的字段名的反射对象在哪里？目的对象的内存指针在哪里？如果想要让解析过程变快，最好是直接去掉这些判断：遇到"a"， 在目的内存的偏移位置0，写入8字节整型值.....

但是上面的做法又没有通用性。如何直接的解析一个类型，又满足通用性？JIT就是个好办法。

针对类型`Data`，通过JIT产生一段最直接最高效的解析代码，并且以后都通过这段代码来解析。进而推演到每个类型都有专门的解析代码。如此：针对特定结构，有特定的最优解析代码。这样的做法绝对是最优的，无法被别的方法超越。

就像ClickHouse一样，相信未来会有越来越多的系统应用会添置JIT的能力。

CGO

关于cgo的性能，我认为主要是golang runtime中的物理线程(GMP模型中的M)，与运行CGO的物理线程之间的通讯造成了远高于直接函数调用的损耗。

> 内部显示 如果是单纯的 emtpy call，使用 cgo 耗时 55.9 ns/op， 纯 go 耗时 0.29 ns/op，相差了 192 倍。
>
> 而实际上我们在使用 cgo 的时候不太可能进行空调用，一般来说会把性能影响较大，计算耗时较长的计算放在 cgo 中，如果是这种情况，每次调用额外 55.9 ns 的额外耗时应该是可以接受的访问。
>
> ——CGO 和 CGO 性能之谜

可以说，CGO的调用本质上是线程间通讯，能否绕过这种开销呢？可以的！请看 fastcgo 这个项目——通过汇编来调用某个c的函数指针，从而避免了线程间通讯。但是缺点就是一旦C函数中存在阻塞，会导致调度器阻塞。

golang为了保障runtime的协程调度不被阻塞，就需要所有被调度的协程函数都是不阻塞的。一旦加入CGO，就无法保障函数不阻塞了，因此只有额外开辟物理线程来执行CGO的函数。

```
这里特别需要注意的一个坑是：
调用CGO的次数越多，时间越长，golang runtime开启的物理线程就越多。
我曾在VictoriaNetrics中的vm-storage中发现，因为大量调用ZSTD压缩库，导致物理线程数是允许核数的10倍。
并且，在目前的golang版本中，这些物理线程没有明确的销毁机制。
远多余可用核数的物理线程，会导致大量CPU时间消耗在无意义的线程切换上。建议运营中加上runtime的metric上报，一旦发现物理线程过多，定期重启来减少这种损耗。
```

其他的不高级主题

panic

不要用panic来反馈异常，不要用recover()来接收异常。

除了程序初始化的错误，不要在业务的任何地方使用panic。

对于错误，存在可预见的error，和不可预见的panic。绝大多数情况都要通过error来针对性的识别并管理错误。recover()仅仅用于维护框架稳定的非预期的错误捕获。

```
目前还未测试过使用recover()是否会导致性能受损。
就我阅读VictoriaMetrics的源码看来，他们一个recover()都没用——也就是说，他们自信的认为组件只会产生可预见的error。
如果我们处处都想着加上recover()来捕获panic，是否意味着设计和测试上存在问题?
```

for循环避免拷贝

VictoriaMetrics中，几乎所有的for循环都是一种风格：

```go
var slice []int64
for i := range slice{
  item := &slice[i]
  // do something
}
```

我想这就是为了避免for循环中的第二个变量产生拷贝。就如同写C/C++的人，for循环中的循环变量要求写成 `++i` 而不是 `i++`。规范好写法，避免在细节之处有不必要的损耗。

内存对齐

golang中声明的每个变量默认都是字节对齐的，这点很好。

需要额外注意两点：

- 一个大的struct数组，要注意字节对齐带来的不必要消耗。内存敏感的话，调整字段的顺序以节约空间。
- 一个大的struct数组，可以故意加些padding的字段，然后item尽可能的按照cache line的长度对齐，可以提升访问性能。

分支预测优化

这种优化点很难找。

关于分支预测的案例，可以看看我写的这个分析文章：《用重复写入代替if判断，减少程序分支》

golang标准库中也有个很好的例子：《How does ConstantTimeByteEq work?》

一个简单的if x==y，考虑了攻击者对计算时间的猜测，考虑了分支预测的损耗。

其他的关于分支预测的优化技巧，这篇也不错：《浅谈利用分支预测提高效率》

在日常的开发中，换个写法是有可能会提高性能的：

```
switch variable{
  case "a":    // 根据业务特点，把最可能的分支放在最前。提高分支预测的成功率
     // do something
  case "b":
     // do something
}
```

根据可用内存来分配对象数量

以prometheus为例，一个突然的大查询会导致耗满容器内存，然后引发OOM导致崩溃。作为系统软件，因为一个无法预估容量的大查询而导致自身崩溃，这一点是非常糟糕的。

相比之下，VictoriaMetrics中会先读取容器的可用内存，然后根据可用内存来分配对象的数量：

```
  maxBufSizePerStorageNode = memory.Allowed() / 8 / len(storageNodes)
  if maxBufSizePerStorageNode > consts.MaxInsertPacketSize {
    maxBufSizePerStorageNode = consts.MaxInsertPacketSize
  }
// memory.Allowed() 获取了容器内的可用内存
// 除以8表示这个类型的对象，最多允许占用整个可用内存的八分之一(aka, 12.5%)
```

强烈建议系统应用中学习一下VictoriaMetrics。具体实现代码请见：https://github.com/VictoriaMetrics/VictoriaMetrics/blob/cluster/lib/cgroup/mem.go

池化

这个技术是老生常谈了——内存池、对象池、协程池、连接池等。

- 内存池：golang已经用了tcmalloc组件，我觉得没必要再自己做一次了。
- 对象池：参考第一章，不同的场景使用不同的对象池技巧。
- 协程池：我认为要看场景。对一个特定的生产者消费者模式而言，协程数量与可用核数对齐是个好办法。其他的场景，要考虑管理协程和创建/销毁协程哪个的成本更高。大多数情况下，协程池这个设计比较鸡肋。
- 连接池：似乎也没特别好说的，不过这篇分析文章让人耳目一新：《Golang 黑魔法之 4 倍性能提升》——每次都读完接收缓冲区的数据，使得连接池的复用率提升。

OK，文章到这里就结束了。

本人也才写了两年的golang，难免有很多错误之处，还请读者不吝赐教，谢谢！

（原文链接）