Go打动服务端开发者：协程（goroutine 、channel）+编译快+中规中矩的垃圾自动回收。

常见面试题：

1. nil = 类型nil+值nil
2. Golang 为什么不能直接将任意类型数组赋值给 []interface{}

每个interface{}接口占用两个字（一个字表示所包含内容的类型，另一个字表示所包含的数据或指向数据的指针），于是[]interface{}的内存即2\*N的空间

同等长度的[]MyType 切片，其内存空间为N \* sizeof(MyType)

1. context作用：可以cancel（遵循父子关系）；服务内/间传递信息
2. slice扩容：不同go版本还不一样，简而言之，小于一定值2倍扩容否则小点的倍数，目标容量超过2倍就直接目标容量。
3. 协程
   1. 线程比的优势：用户态调度，更加轻量（4K占用堆栈，创建与销毁不需要线程的syscall）
   2. defer&recover原理

协程g结构体中字段有panic链表、recover链表

当panic中继续调用panic会导致panic链表多个元素

按照defer链表顺序执行defer，最里面的defer在head

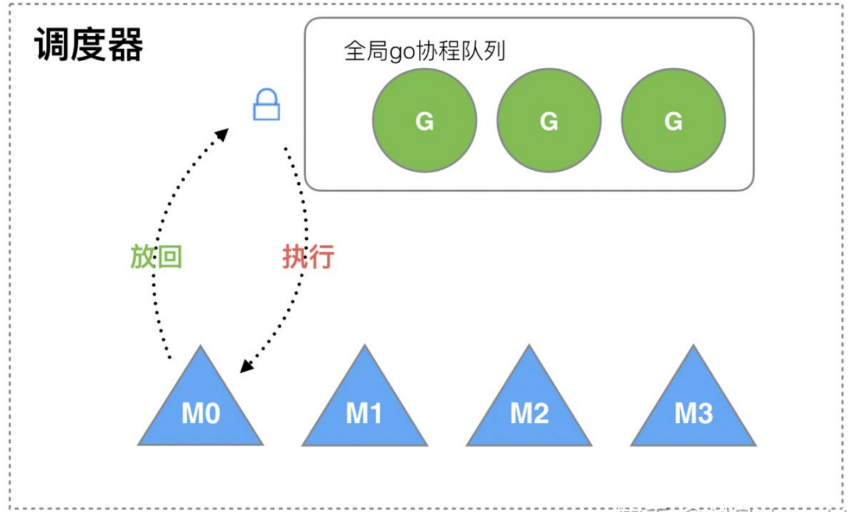
因此main defer recover不会recover住协程内部panic

recover函数：取出当前 goroutine 的 \_panic 链表最新的一个非空 \_panic对象，对其recovered字段赋值为true。

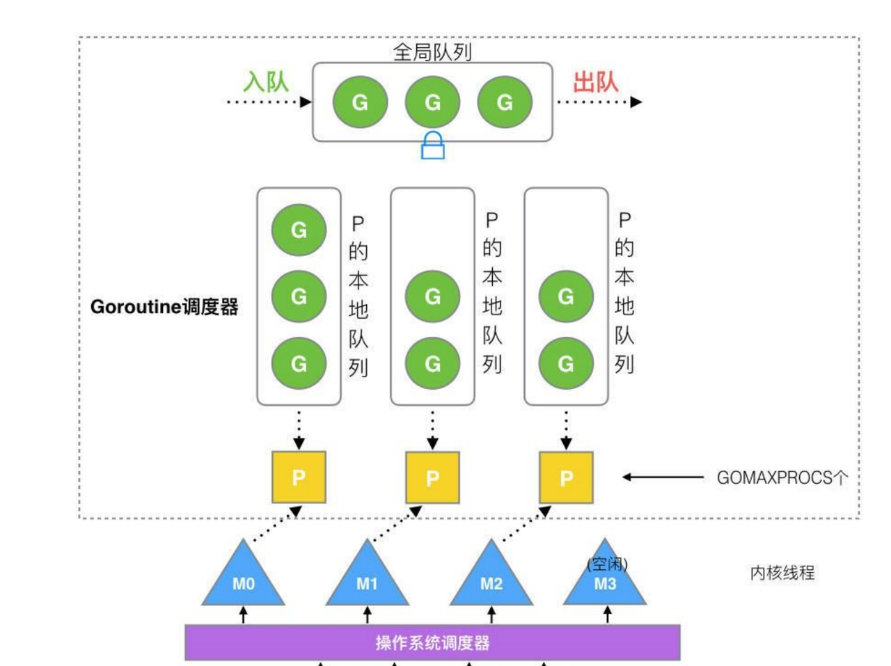
panic处理函数发现recovered=true，goroutine回到调度队列，进程不退出。

* 1. channel以循环数组+mutex互斥锁实现
     1. 对已经关闭的 channel 进行读取操作会返回剩余数据 或 一个零值和一个表示channel 是否关闭的标志位。 对已经关闭的 channel 进行写入操作会导致 panic 异常。
  2. 模型

GM 从全局队列队列获取资源需要锁保护



GPM G-goroutine；P-processor；M-thread



全局队列：存放等待运行的G；被全局锁保护。

G

在Go中，一个goroutine最多占用CPU 10ms，防止其他goroutine被饿死

P

本地队列：与全局队列类似，存放的也是等待运行的G，数量不超过256个。新建G'时，G'优先加入到P的本地队列，如果队列满了，则将本地队列中一半的G移动到全局队列；

创建时机：在程序启动时创建，并保存在数组中，最多有GOMAXPROCS个线程分布在多个CPU上同时运行。

M：

创建时机：没有足够的M来关联P并运行其中的可运行的G。比如所有的M此时都阻塞住了，而P中还有很多就绪任务，就会去寻找空闲的M，而没有空闲的，就会去创建新的M。

在循环：运行golang调度器schedule() 或 执行用户线程goroutine()

线程想运行任务就得获取P，从P的本地队列获取G，P队列为空时，M会尝试从全局队列拿一批G放到P的本地队列，或从其他P的本地队列窃取一半放到自己P的本地队列。M运行G，G执行之后，M会从P获取下一个G，不断重复下去。

work stealing机制：当本线程无可运行的G时，尝试从其他线程绑定的P偷取G，而不是销毁线程。

hand off机制：当本线程因为G进行系统调用阻塞时，runtime会把P和M分离，并绑定新M。

1. 变量存放位置

Go的变量逃逸：本应该分配在栈上面的变量，却分配在堆上面了，说明发生了逃逸

golang通过逃逸分析来确定变量是分配在栈上还是堆上

- 如果函数外部没有引用，则优先放到栈中；不涉及GC

- 如果函数外部存在引用，则必定放到堆中；GC负责回收，比如返回指针

- 大内存变量可能分配在堆上

1. GC三色标记法

三色：白色默认值，灰色遍历起点，黑色遍历过

- 白色对象 — 潜在的垃圾，其内存可能会被垃圾收集器回收；

- 黑色对象 — 活跃的对象，包括不存在任何引用外部指针的对象以及从根对象可达的对象；

- 灰色对象 — 活跃的对象，存在外部指针，垃圾收集器会扫描这些对象的子对象

标记：

step1. 根对象被标记为灰色，并放入处理队列

step2. 从处理队列里面选出一个灰色对象，标记为黑色，使用广度优先遍历，将可以遍历到的对象标记为灰色放入处理队列

step3. 不断重复上述过程，直至队列中不再存在灰色对象

goroutine

在用户空间存在一个调度器，一旦进行go库函数的调用，runtime就有机会进行调度。如果线程编写了一个单纯的死循环，虽然有CPU调度但基本会持续占用掉一个核心。goroutine是非常轻量级的,它就是一段代码,一个函数入口,以及在堆上为其分配的一个堆栈(初始大小为4K,会随着程序的执行自动增长与删除)。

runtime.GOMAXPROCS(number) 绑定CPU

package

vendor：govendor godep

将所有的外部依赖打包到项目的vendor目录，编译时路径优先级还高于$GOPATH/src，但是vendor中没有版本概念，如果出现不同的库依赖于同一个库但是不向前兼容的两个版本，会GG。

govendor init

govendor add +e(相当于+external，所有外部依赖)

govendor update +v(相当于+vendor)

+all 会包含所以依赖，包含了当前GO SDK中的库

vendor->go.mod->go.work

数组

package main

import "fmt"

func appendChange(buf []int) {

    buf = append(buf, 0)

    buf = append(buf, 0)

    buf[0] = 1

    fmt.Printf("%p: %v\n", buf, buf)

}

func main() {

    buffer := make([]int, 0, 2)

    buffer = append(buffer, 0)

    buffer = append(buffer, 0)

    appendChange(buffer)

    fmt.Printf("%p: %v\n", buffer, buffer)

}

lipeinandeMacBook-Pro:test lipeinan$ go run main.go

0xc0000120e0: [1 0 0 0]

0xc000014070: [0 0]

因为原先限制大小为2，所以超过之后就会被append重新创建。

语法

条件时，临界区要尽可能的小，为此可以尽量使用基于 atomic.StorePointer 的无锁结构

Go 的 Context 可以存取一些全局变量，其存储方式是一个树状结构，每一次取值的时候就会从当前节点一路遍历到根节点，查找是否有对应的 Key。因此有http框架，将取参数逻辑增加了一个map参数 func xxx(ctx context.Context, newCtx map[string]interface{})

(\*Type)(nil) ≠ nil

语言内的 nil 可以理解为一个 Type 和 Value 均为空的 interface

代码里面返回的 p 虽然 Value 为空，但是 Type 是 \*MyError

结构体interface: Comparable才可判断是否等于， Ordered才可比较大小

Go 当中绝大多数类型都是可以使用运算符相互比较的，唯独不包含 Slice，Map 和 Func，也有容器类型 Struct、Array 本身的 Comparable 取决于成员的类型或自身实现。

x, ok := err.(interface{ Is(error) bool }); ok && x.Is(target)

Go优化

**内存分配大的自己搞分配**

**inline减少函数调用、参数存寄存器、去除多余指令、部分场景SIMD**

**内存分配**：程序分配的大部分小对象都是短生命周期的。在使用Go语言的微服务上做profiling的时候经常能观察到mallocgc()及其关联的内存管理方法占据了较多的CPU资源。

- Go程序频繁堆内存分配动作，比如大量json和thrift 序列化/反序列化在微服务中处理请求参数

- Go语言目前的堆内存分配实现比较重，消耗较多CPU资源

因此预先分配一块较大的buffer，进行小对象分配时优先采用buffer内，因此不用执行完整的mallocgc逻辑。Go 的GC goroutine会把整个buffer看作一个完整的对象来。当GAB之外的对象引用了GAB内部的一个对象时，效果上等同于一个对象直接引用了另外一个对象内部的某个field。

- go stack copy优化。golang的栈的扩栈缩栈操作在gonotice中占比比较高，约为5.6%左右。

新创建的goroutine，虽然默认分配了2K的栈大小，但是，由于有stackguard的存在(go1.16 linux默认的stackgurad是928)，所以实际使用的时候，栈的长度达到1120字节就会触发栈的拷贝了。

栈的拷贝步骤有以下几步，比较复杂，所以栈的拷贝开销比较大

1. 计算要扩栈后的栈的大小（一般为2倍，但是如果当前函数较大，也会增加到4倍甚至8倍...）

2. 申请新的内存，并把当前栈的内容拷贝过去

3. 遍历栈帧，逐帧的修改指向旧栈的指针

4. 销毁并回收旧栈

把default goroutine大小设成4k，应该能提升gonotice的性能大约3%左右。goprofile可能只有0.5%左右的提升。 内存可能会有小幅度增长。函数在运行时，在入口会去检查当前栈的剩余空间是否满足需求。如果不满足，则执行栈的扩容操作。

例外场景：频繁申请很小的内存放到GAB里面，并且会把其中一定比例的内存作为local cache起来不释放。这样就会导致很多GAB的buffer无法被GC回收，这种场景不是GAB目标支持的。

**编译优化**：Go语言的一个卖点在于编译速度，所以go在编译优化时使用了一些比较快速但是相对比较简单的算法，比如寄存器分配采用线性分配算法，内联优化实现比较简陋。增加一些编译优化，或者进行一些编译优化策略的调整，提供一个侧重于服务端长期运行的代码的编译器，以编译时时间开销为代价达到生成代码运行速度提升的目的。

- 常调用路径进行inline调整

对于一些newobject类型的对象，在编译时期，对象的大小和类型信息是可知的。所以可以选择在编译时期直接生成gab的快速路径。这样相比原有的实现方式可以节省掉runtime.object 和mallocgc的调用和类型信息和大小的判断。

根据Inline收益静态分析再探发现，在gonotice和goprofile中，大多热函数不能inline的原因有两个 1. Inline budget太小 2. 存在for/range循环 所以做了一个commit, 增加inline budget， 同时把包含for/range的函数inline进去。在goprofile上压测有约5%左右的性能提升。

inline实际上“相当于”宏替换，就是把函数的二进制代码直接复制到调用的地方，因而inline代码不应该有跳转。而循环结构无法避免条件跳转，所以有循环的代码无法inline。

一个函数里面如果有defer语句，这个函数是不能被inline的。要不然会影响到原有的执行逻辑。 比如函数callee panic，会被defer recover()给捕捉到

Binary size 会略微增加5%~10%左右，编译时长增加。主要是受inline策略调整的影响。运行时内存可能会稍微增加，但是影响会比较小。

- Aggresive Boundcheck Elimnation

go是内存安全的语言，在操作数组或切片的索引时，用户不用手动检查是否出现了数组越界，go的编译器会在生成代码中插入边界检查的指令，如果出现数组越界，会抛出panic。编译器在优化的过程中，会进行bound check elimination的优化，这个优化可以用来减少一些非常明显的不必要的边界检查。

对于runtime的代码，经过我们仔细检查，我们认为不会出现越界访问的情况。此外runtime中的panic无recover机制，所以一旦runtime中出现越界情况，程序必然会出现crash。

GC控制：Golang目前的release版本只有GOGC一个参数来调整GC，这个参数实际上控制的是触发下一次gc时堆内存的目标增长率。更好的方法是像Java一样，设置一个预期的内存上限，让Golang自动的控制GC的频率。

Go compiler本身的问题。golang为了保证编译速度，在编译阶段进行的优化工作较少，并且无法直接使用诸如LLVM（Low Level Virtual Machine）等编译器后端进行优化。将一些关键的计算函数用某种执行效率更高的语言编写：C/Clang是一个理想的编译体系（内部集成LLVM），充分编译优化后的x86 asm转译成plan9并嵌入到go代码

json https://github.com/bytedance/sonic

json-iterator库的做法是：基于codec-assembly机制，把schema拆解成一个个字段编解码函数，然后组装并缓存，最小化了反射和分支选择带来的性能损耗。在实际测试中我们发现，随着json嵌套层数加深，json-iterator与其它库差距逐渐缩小甚至被反超，这是因为这种实现实际转化成了大量的interface封装和function调用，随之而来的是function-call损耗：​调用interface涉及itab动态寻址​、组装函数无法内联，而golang的函数调用性能较差（没有寄存器传参）。优化办法：JIT(just in-time compile）。由于编译出来的codec函数是一个整体，能极大减少函数调用同时又保证业务灵活性。

SIMD在json处理上有用武之地的（itoa、char-search都是适合的场景）。我们看到在>100KB的大型json场景下，simdjson-go是非常有竞争力的。但是对于一些极小或不规律的字符串，SIMD需要的额外load操作反而导致性能下降。因此我们需要做好前置处理，决定哪些场景该使用SIMD，哪些场景不需要SIMD。针对单key查找和多key遍历两种不同场景，分别设计一个利用SIMD加速的查找器和一个基于lazy-load模式的解析器。

日志库优化：减低CPU消耗

Logs 的日志文本格式如下：{Level} {Date} {Time} {Version}({NumHeaders}) {Location} {HostIP} {PSM} {LogID} {Cluster} {Stage} {RawLog}

使用已有的 Log SDK Logs 打印单条日志往往需要付出数微秒的开销。Log SDK V2 旨在通过在日志生成时尽量避免所有的：

- 对象逃逸

- 系统调用

- 堆内存分配

- 内存的拷贝与移动，传递指针而不是拷贝内容

- 反射

- C++私有协议提升了解析与反解析性能

最终将生成单条日志的性能开销降低到至 100～250 纳秒，相比 Logs 原生接口 (CtxInfo) 提升 12 倍的性能，相比开启全部优化的 Logs 接口 (Ctx\*sffl) 减少约 70% 以上的性能开销。

写法上从函数调用加参数，变成函数链加函数。对于新的升级与改动，只需要对实例增加新的方法即可，而无须改动整体的参数与返回值。

每次文件写入的大小总是大于等于 4KB 有助于提升写入性能；

优化：

对于 []byte 与 string 类型的互相转换，如果可以确认对内存地址只读，使用 unsafe.Pointer 可以避免发生可能的堆内存分配与拷贝。

使用 fmt.Sprintf 对日志文本进行拼接，这带来了两个问题：因此，在做日志的文本拼接时，应该尽量避免使用 fmt 库。

1. 没有采用追加方式对字符串进行拼接，从而需要更多的内存整理与移动 (runtime.memmove)；

2. fmt.Sprintf 的参数类型为 interface{}，任何结构体在作为 interface{} 类型的参数传入函数时，都会在逃逸分析中被判断为需要逃逸，最终造成不必要的垃圾回收开销与堆内存分配。

使用 time.Time.AppendFormat 对时间进行序列化而不使用 time.Time.Format 可以避免新的堆内存分配；

存在竞态

库

SDK写法：client.xxModule.xxFunc

json https://github.com/bytedance/sonic

singleflight 保证单机内同一时间相同请求处理只有一个，其余陷入等待，一起返回

golang.org/x/sync/errgroup 协程池+错误处理取消协程，每一个协程内函数需要监听select

结构体复制 https://github.com/jinzhu/copier

ORM https://github.com/go-gorm/gorm

并发map github.com/puzpuzpuz/xsync

RPC框架 https://github.com/cloudwego/kitex

HTTP框架 https://github.com/cloudwego/hertz

mod可视化分析 <https://github.com/PaulXu-cn/go-mod-graph-chart>

Local command UI: https://charm.sh

gorm库，sql record not find: errors.Is(err, gorm.ErrRecordNotFound) || errors.Is(err, sql.ErrNoRows)

in-memory cache

https://github.com/allegro/bigcache

https://github.com/patrickmn/go-cache

sort.pdqsort 是一种新型的混合排序算法，在所有场景相比于 Go 语言内置排序算法sort.Sort会快 2 ~ 60 倍。

sync.Map

https://github.com/bytedance/gopkg/tree/develop/collection/skipset

https://github.com/bytedance/gopkg/tree/develop/collection/skipmap

xxhash3 是目前业界中速度以及 hash 质量排第一序列的 hash 算法 https://github.com/bytedance/gopkg/tree/develop/util/xxhash3

伪随机rand库 <https://github.com/bytedance/gopkg/tree/develop/lang/fastrand>

HTTP https://github.com/go-resty/resty

moul.io/http2curl