Golang 高性能编程分享

郑文峰

引言：为什么要关注Go性能?

- 性能优化能够减少响应时间，提高系统吞吐量，为用户提供更流畅、更快速的交互体验，尤其在高并发场景下至关重要。
  - Logrich服务对系统吞吐量有不小的要求。
- 降低系统资源消耗，包括CPU使用率、内存占用和网络带宽，从而节约运营成本，提高系统整体效率。
  - 最近发生了因为logrich和grpc-server服务 OOM 导致熬夜加班。

内存管理与数据结构优化

堆栈分配与逃逸分析

- 栈分配 VS 堆分配
  - 栈分配：轻量快速，函数结束时自动释放，不产生垃圾
  - 堆分配：需要垃圾回收(GC)参与，开销较大
- 逃逸分析是Go编译器在编译时进行的一种优化技术，它会分析变量的生命周期和使用方式，自动决定将变量分配在栈上还是堆上。
- 通过添加参数 -gcflags=“-m" 可以看到逃逸情况
- 常见逃逸的场景
- 优化栈分配场景
  - GC 压力大的时候
  - 对于短期的小对象
  - 高频调用的函数内部
- 不必强求栈分配的场景
  - 工厂方法返回对象指针(Go惯用法)
  - 对象需要跨函数生命周期。
  - 不频繁创建的小对象
  - 优化会影响代码可读性时

GC

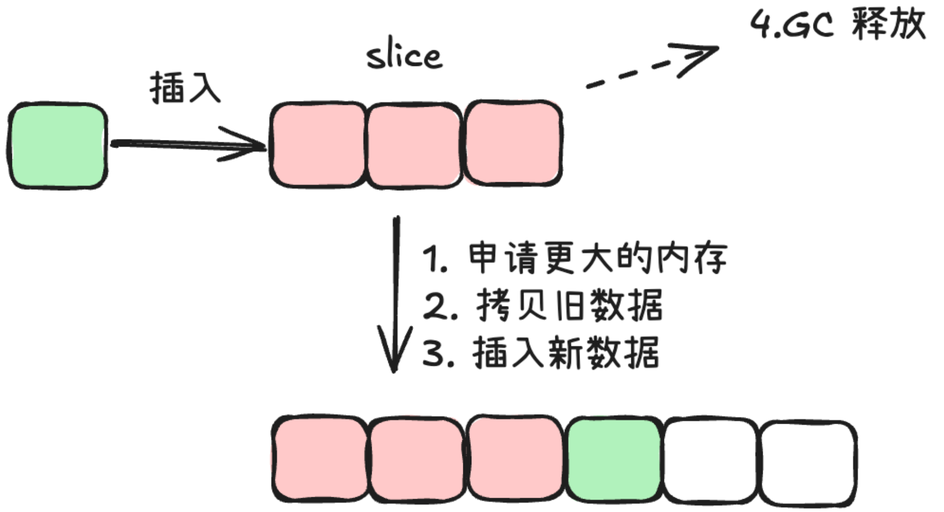
- 自动识别内存中哪些部分不再需要，代表应用程序回收内存。
- 从根结点开始遍历所有的对象进行标记与清除。
- GC开销
  - 需要启动专门的后台 goroutine 执行并发的标记与清除操作。
  - 当应用程序分配新内存的速度过快，后台 GC 标记工作跟不上，则会征用业务 goroutine 来帮助 GC 完成标记工作。
  - Stop-the-world，GC在标记阶段和清扫阶段之间会有短暂的全局暂停，是完全的停止。
- GC 参数调优
  - GOGC
    - Target heap memory = Live heap + (Live heap + GC roots) * GOGC / 100
    - 调高 GOGC → 减少 GC 频率，但内存占用更大。
    - 调低 GOGC → 更频繁 GC，减少内存但可能影响性能。
    - 项目案例
  - GOMEMLIMIT
    - 当前程序接近 GOMEMLIMIT 时，GC 的频率会更加的激进，增加 CPU 的负载。
    - 避免因内存超限而被k8s给oom kill，设置成容器内存限制的 90%-95%
  - GODEBUG=gctrace=1 打印 GC 日志，用于监控和分析性能。

空结构体零消耗

- 空结构体不占用任何内存空间，所有空结构体实例共享相同的内存地址。
- 项目案例
- 应用场景
  - 实现 Set 数据结构，项目案例
  - 作为channel的信号，项目案例

预分配容量

- Slice 和 Map 会动态的扩展来适用新的元素数量，空间不足时是会进行分配新的内存、复制、以及旧内存的回收操作，而频繁的调整大小的操作会显著的降低性能。
- 预分配就是提前设置好需要的大小，从而避免了动态扩容。

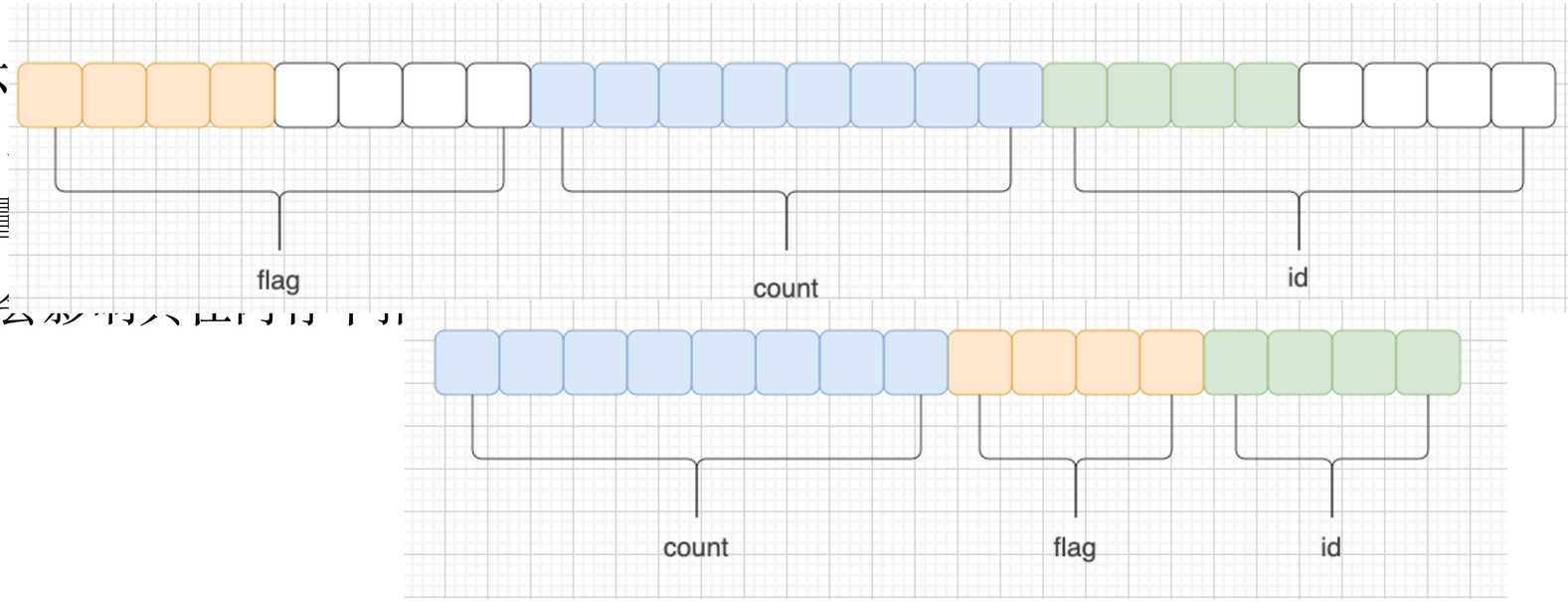


- 举个栗子
  - 容量随着插入的元素不断地增大。容量增长规则
  - 有无预分配的benchmark比较

- 项目案例

struct 内存对齐

- CPU 访问内存时是根据字长来访问而不
  CPU 字长是 4 字节，64 位的 CPU 时 8
  减少 CPU 访问内存的次数，提高吞吐量
- 而结构体中不同类型的变量排列顺序则
  序，从而影响 CPU 访问内存的效率。
- 举个栗子
- 最佳实践
  - 通过对字段的排序来减少内部无意义的填充。
  - 尽可能将类型大小相同的字段放在一起，避免大小交替。
  - 使用 fieldalignment linter 对代码进行校验，通过工具自动捕获



运行时性能优化

最优遍历方式

- 遍历的方式主要有两种
  - 索引遍历和值遍历
- 两者的benchmark比较
  - 取值时会有进行数据的复制，而索引取值不会，所以索引遍历更快。
- 项目案例
- 最佳实践
  - 大结构体切片优先使用索引遍历
  - 基础类型切片，按编码习惯选择即可

最优字符串拼接

- 常见的六种字符串拼接性能大比拼
  - +号拼接：最简单的拼接方式
  - fmt.Sprintf：格式化拼接
  - strings.Builder：专门优化的字符串构建器
  - bytes.Buffer：字节缓冲区
  - []byte转换：字节切片转换
  - 预分配[]byte：预先分配足够空间的字节切片
- 经过测试结果如下：
  - 预分配[]byte性能最佳，适合高性能场景
  - strings.Builder是通用场景的最佳选择
  - +号和fmt.Sprintf在循环拼接中性能极差

IO缓冲

- 在计算机系统中，I/O操作（如文件读写、网络通信）是性能瓶颈的主要来源之一。主要原因包括：
  - 系统调用开销：每次直接I/O操作都涉及用户态和内核态的上下文切换
  - 硬件限制：磁盘和网络设备更适合大块数据传输
  - 频繁小数据操作：大量小数据写入会显著降低性能
- 有无使用IO缓冲的benchmark对比
- 应用场景
  - 频繁的执行小数据量的 I/O。
  - 减少系统调用。
  - 高吞吐量比延迟更重要。
- 不适用场景
  - 实时性要求高。
  - 过度缓冲导致内存使用不受控制。

Interface Boxing

- 在Go语言中，interface{} 是一种强大的抽象机制，但将具体类型赋值给 interface{} (称为Boxing)会带来一定的性能开销
- 将具体类型的值赋值给interface{}的过程称为Boxing。在这个过程中：
  - 值会在堆上分配新内存并拷贝
  - 将指针及对应类型赋值给interface{}变量
  - 这会带来额外性能开销并增加GC压力
- 举个栗子
  - 基础类型的逃逸分析及对应的benchmark比较
  - 值和指针类型的结构体逃逸及对应的benchmark比较
- 项目案例
- 项目案例2
- 项目案例3
- 最佳实践
  - 传递给接口时使用指针。可以避免内存的重复复制与申请。
  - 如果设计 API 时，类型已经确定并且是稳定的，尽可能避免使用 interface 。
  - 尽可能使用特定类型的容器。

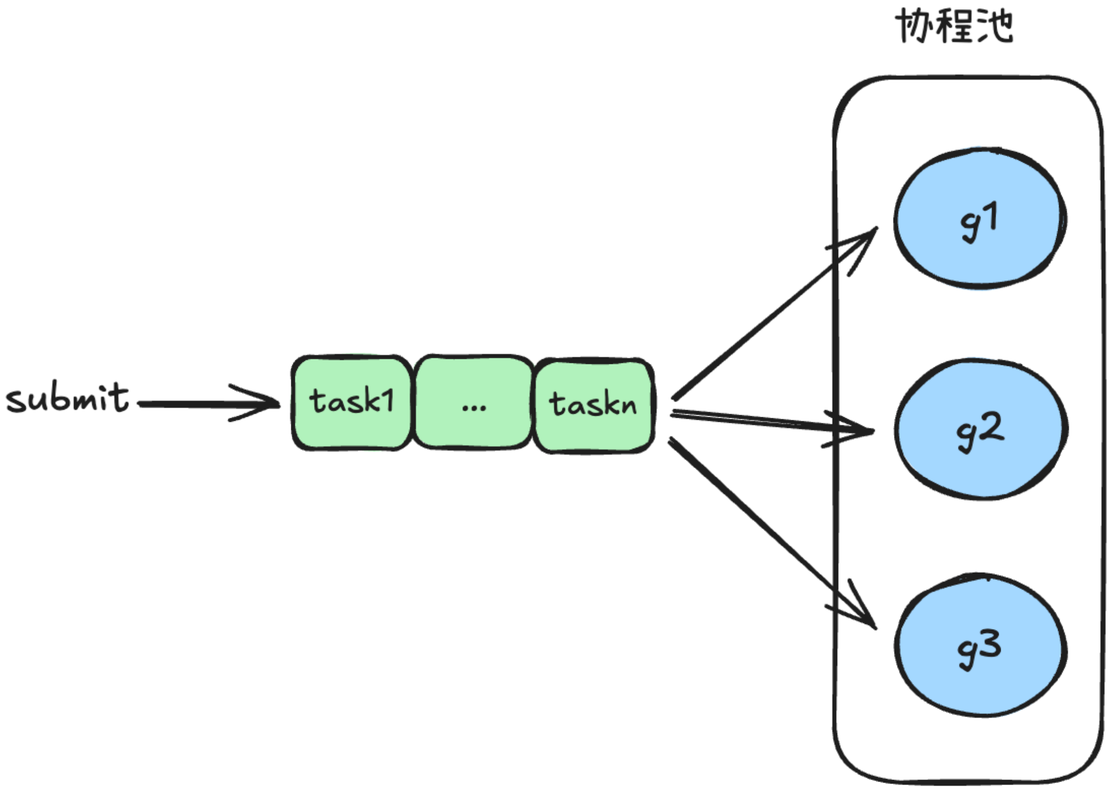
原子操作

- 原子操作是指不可中断的一个或一系列操作，这些操作要么全部执行成功，要么全部不执行。
- 允许在不适用互斥锁的情况下安全地并发访问共享数据，加锁会引入协调开销，性能可能会下降，而原子操作使用 CPU 指令直接在硬件层面进行操作，从而有更高的性能。
- 原子操作的应用场景
  - 计数器实现
  - 状态标志控制
    - 案例: sync.Once
  - 单次初始化(替代sync.Once)
- 栗子
  - 原子操作与互斥锁 benchmark对比

并发模型与资源调度

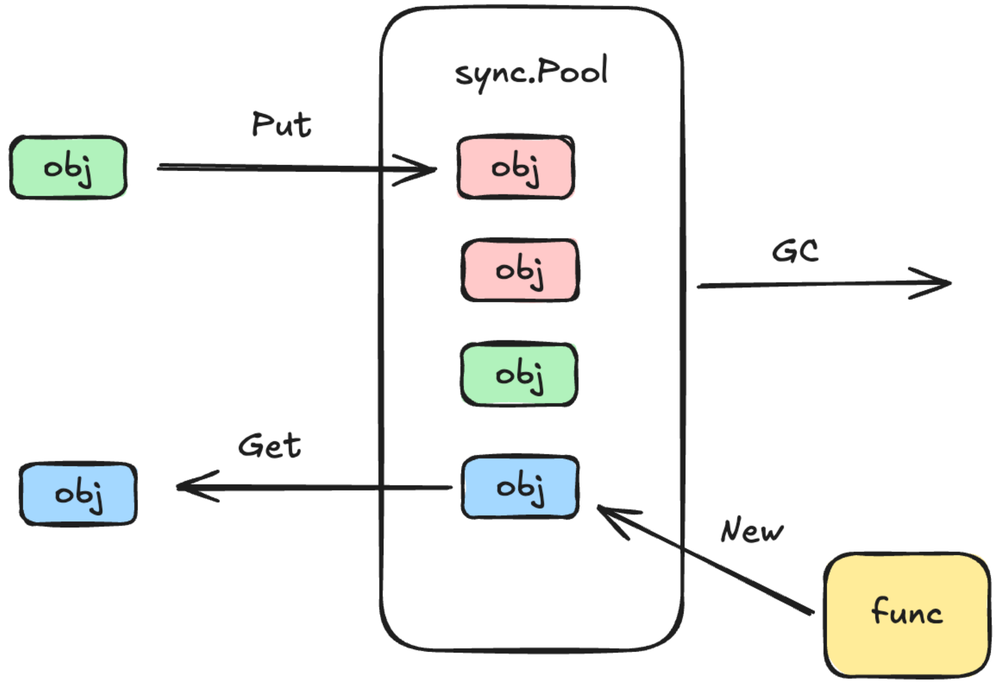
协程池

- goroutine 虽然是轻量级的并发模型，但是协程也是有栈空间的，并且有上下文切换的开销，当协程数量增加时，性能可能会急剧的下降，甚至导致程序崩溃。
- 而协程池限制 gorotinue 的数量，并从共享的任务队列中提取任务执行，从而让 goroutine可控，不会超过其处理的能力，保证服务的稳定性。
- 举个栗子
  - 协程池与非协程池的benchmark基准测试
- 项目案例



Sync.Pool

- 一个高性能对象池，用于复用临时对象。
- 减少 GC 压力
- 提升性能（减少内存分配）
- 适用于“临时对象”
- 举个例子
  - 有无使用 sycn.Pool 的 benchmark
- 项目案例



pprof 实战

- 服务启动时 OOM
  - Logrich，在服务启动时构造树时大量的创建node从而导致内存快速上升，导致 GC 不过来了，从而发生了 OOM。
  - 通过动态的调节 GC，在构造树时降低 GOGC的大小，提前进行 GC。
  - grpc-service-framework，在服务启动时会将大量的数据进行序列化，而过程中创建大量的内存与动态扩容，导致了 OOM。
  - 通过sync.Pool+预分配从而降低了使用的内存。

谢谢

- 博客网站：www.zhengwenfeng.com