golang runtime 的理解

作者：cuua
原文链接：https://ld246.com/article/1610696930954
来源网站：链滴

runtime 运行时到底是个什么东西?
Go 的调度为什么说是轻量的?
Go 调度都发生了啥?
Go 的网络和锁会不会阻塞线程?
什么时候会阻塞线程?
Go 的对象在内存中是怎样的?
Go 的内存分配是怎样的?
栈的内存是怎么分配的?
GC 是怎样的?
GC 怎么帮我们回收对象?
Go 的 GC 会不会漏掉对象或者回收还在用的对象?
Go GC 什么时候开始?
Go GC 啥时候结束?
Go GC 会不会太慢, 跟不上内存分配的速度?
Go GC 会不会暂停我们的应用? 暂停多久? 影不影响我的请求?
带着这些问题，我们来一起研究 golang 的 runtime

Golang Runtime 简介

Golang Runtime 是 go 语言运行所需要的基础设施

协程调度, 内存分配, GC;
操作系统及 CPU 相关的操作的封装(信号处理, 系统调用, 寄存器操作, 原子操作等), CGO;
pprof, trace, race 检测的支持;
map, channel, string 等内置类型及反射的实现.

与 Java, Python 不同, Go 并没有虚拟机的概念, Runtime 也直接被编译

成 native code.
<ol start="2">
Go 的 Runtime 与用户代码一起打包在一个可执行文件中
用户代码与 Runtime 代码在执行的时候并没有明显的界限, 都是函数调用
go 对系统调用的指令进行了封装, 可不依赖于 glibc
一些 go 的关键字被编译器编译成 runtime 包下的函数.

Runtime 发展历程

注: GC STW 时间与堆大小, 机器性能, 应用分配偏好, 对象数量均有关. 较早的版本来自网络上的据. 1.4-1.9 数据来源于 twitter 工程师. 这里是以较大的堆测试, 数据仅供参考. 普通应用的情况好于述的数值.

Golang 调度简述

PMG 模型, M:N 调度模型.
调度在计算机中是分配工作所需资源的方法. linux 的调度为 CPU 找到可运行的线程. 而 Go 的调是为 M(线程)找到 P(内存, 执行票据)和可运行的 G.
轻量级协程 G, 栈初始 2KB, 调度不涉及系统调用

用户函数调用前会检查栈空间是否足够, 不够的话, 会进行栈扩容.
用户代码中的协程同步造成的阻塞, 仅仅是切换协程, 而不阻塞线程.
网络操作封装了 epoll, 为 NonBlocking 模式, 未 ready, 切换协程, 不阻塞线程.
每个 p 均有 local runq, 大多数时间仅与 local runq 无锁交互. 实现 work stealing.
用户协程无优先级, 基本遵循 FIFO.
目前(1.12), go 支持协作的抢占调度, 还不支持非协作的抢占调度.

协程结构体和切换函数

GM 模型

一开始, 实现一个简单一点的, 一个全局队列放待运行的 g.新生成 G, 阻塞的 G 变为待运行, M 寻可运行的 G 等操作都在全局队列中操作, 需要加线程级别的锁。

调度锁问题. 单一的全局调度锁(Sched.Lock)和集中的状态, 导致伸缩性下降.
G 传递问题. 在工作线程 M 之间需要经常传递 runnable 的 G, 会加大调度延迟, 并带来额外的性损耗
Per-M 的内存问题. 类似 TCMalloc 结构的内存结构, 每个 M 都需要 memory cache 和其他类的 cache(比如 stack alloc), 然而实际上只有 M 在运行 Go 代码时才需要这些 Per-M Cache, 阻塞在统调用的 M 并不需要这些 cache. 正在运行 Go 代码的 M 与进行系统调用的 M 的比例可能高达 1:100 这造成了很大的内存消耗.

是不是可以给运行的 M 加个本地队列?
是不是可以剥夺阻塞的 M 的 mcache 给其他 M 使用?

GPM 模型

Golang 1.1 中调度为 GPM 模型. 通过引入逻辑 Processer P 来解决 GM 模型的几个问题.

mcache 从 M 中移到 P 中.
不再是单独的全局 runq. 每个 P 拥有自己的 runq. 新的 g 放入自己的 runq. 满了后再批量放入局 runq 中. 优先从自己的 runq 获取 g 执行
实现 work stealing, 当某个 P 的 runq 中没有可运行 G 时, 可以从全局获取, 从其他 P 获取
当 G 因为网络或者锁切换, 那么 G 和 M 分离, M 通过调度执行新的 G
当 M 因为系统调用阻塞或 cgo 运行一段时间后, sysmon 协程会将 P 与 M 分离. 由其他的 M 来合 P 进行调度

G 状态流转

调度

golang 调度的职责就是为需要执行的 Go 代码(G)寻找执行者(M)以及执行的准许和资源(P).
并没有一个调度器的实体, 调度是需要发生调度时由 m 执行 runtime.schedule 方法进行的.
调度时机:

channel, mutex 等 sync 操作发生了协程阻塞
time.sleep
网络操作暂时未 ready
gc
主动 yield
运行过久或系统调用过久
等等

调度流程:
实际调度代码复杂很多.
如果有分配到 gc mark 的工作需要做 gc mark.
local runq 有就运行 local 的,
没有再看全局的 runq 是否有,
再看能否从 net 中 poll 出来,
从其他 P steal 一部分过来.
....
实在没有就 stopm

sysmon 协程

P 的数量影响了同时运行 go 代码的协程数. 如果 P 被占用很久, 就会影响调度.sysmon 协程的一功能就是进行抢占.
sysmon 协程是在 go runtime 初始化之后, 执行用户编写的代码之前, 由 runtime 启动的不与何 P 绑定, 直接由一个 M 执行的协程. 类似于 linux 中的执行一些系统任务的内核线程.
可认为是 10ms 执行一次. (初始运行间隔为 20us, sysmon 运行 1ms 后逐渐翻倍, 最终每 10ms 运行一次. 如果有发生过抢占成功, 则又恢复成初始 20us 的运行间隔, 如此循环)

每 sysmon tick 进行一次 netpoll(在 STW 结束,和 M 执行查找可运行的 G 时也会执行 netpoll) 取 fd 事件, 将与之相关的 G 放入全局 runqueue
每次 sysmon 运行都执行一次抢占, 如果某个 P 的 G 执行超过 1 个 sysmon tick, 则执行抢占. 在执行系统调用的话, 将 P 与 M 脱离(handoffp); 正在执行 Go 代码,则通知抢占(preemptone).
每 2 分钟如果没有执行过 GC, 则通知 gchelper 协程执行一次 GC
如果开启 schdule trace 的 debug 信息(例如 GODEBUG=schedtrace=5000,scheddetail=1),

按照给定的间隔打印调度信息
每 5 分钟归还 GC 后不再使用的 span 给操作系统(scavenge)

协作式抢占

retake()调用 preemptone()将被抢占的 G 的 stackguard0 设为 stackPreempt,
被设置抢占标记的 G 进行下一次函数调用时, 检查栈空间失败. 进而触发 morestack()（汇编代
，位于
asm_XXX.s 中）然后进行一连串的函数调用，主要的调用过程如下：
morestack()（汇编代码）-> newstack() -> gopreempt_m() -> goschedImpl() -&gt
schedule()

网络

JavaScript 网络操作是异步非阻塞的, 通过事件循环, 回调对应的函数.一些状态机模式的框架, 每
网络操作都有一个新的状态.
代码执行流被打散.
用户态的协程: 结合 epoll, nonblock 模式的 fd 操作;
网络操作未 ready 时的切换协程和 ready 后把相关协程添加到待运行队列. 网络操作达到既不阻
线程, 又是同步执行流的效果.

封装 epoll, 有网络操作时会 epollcreate 一个 epfd.
所有网络 fd 均通过 fcntl 设置为 NONBLOCK 模式, 以边缘触发模式放入 epoll 节点中.
对网络 fd 执行 Accept(syscall.accept4),Read(syscall.read), Write(syscall.write)操作时, 相关</l
>

操作未 ready, 则系统调用会立即返回 EAGAIN; 使用 gopark 切换该协程
<ol start="4">
在不同的时机, 通过 epollwait 来获取 ready 的 epollevents, 通过其中 data 指针可获取对应的 g,
将其

置为待运行状态, 添加到 runq