|  |
| --- |
| <https://blog.csdn.net/puss0/article/details/80627753> |
| <https://blog.csdn.net/xclyfe/article/details/50563048> |
| https://www.zhihu.com/question/20862617# |
| https://blog.csdn.net/justaipanda/article/details/44064811 |

特性：

1. 是最小的并发执行单元
2. 最小的调度单元
3. Goroutine非常轻量，数量很多
4. Goroutine 是协作式任务，不是抢占式多任务

进程：

分配完整独立的地址空间，拥有自己独立的堆栈，既不共享堆，也不共享栈。进程的切换只发生在内核态，有操作系统调度。

进程见通信开销较大，进程间的切换开销也大：Context Switch，上下文包括代码，数据，堆栈，处理器状态，资源。

线程：

和其本进程的线程的共享地址空间，拥有自己独立的栈和共享堆，不共享栈。线程的切换一般也是由操作系统调度

现代操作系统中，是进程中程序执行流的最小单元。标准线程由线程ID，当前指令指针，寄存器集合和堆栈组成

协程：

和线程类似，共享堆，不共享栈。轻量级线程，可以并发执行的函数。有变异或用户指定将控制权交给协程调度程序执行的方式。他是非抢占式，避免反复系统调用，还有进程切换造成的开销。

Work – stealing算法：

当P中的runqueue 里面的G执行完了以后，它可以从全局列表中获取，如果全局列表没有G，那么此时可以随机向另一个P的runqueue中偷取一般的G执行，这样提高了Go语言的并发性，同时避免了goroutine调度时的全局锁。

Goroutine调度器原理：

主要有4个重要结构，分别是M，G，P，Sched

Sched:结构是调度器，他维护有存储M和G队列以及调度器的一些状态信息

M:代表内核级线程，一个M就是一个线程，goroutine就是跑在M之上的，M里面维护小对象内存cache，当前执行的goroutine，随机数发生器等等非常多的信息

P全程是Processeor,处理器，主要用来执行goroutine，其维护了一个goroutine队列，里面存储了所有需要它来只执行的goroutine

G就是groutine实现的核心结构，G维护了goroutine需要的栈，程序计数器以及它所在的M等信息

工人（M） 用小车（P） 运输代加工的砖头（G）

启动过程：

调度器初始化runtime-schedinit后，调用runtime-newproc创建出第一个goroutine，这个goroutine将执行的函数是 runtime-main，即第一个goroutine就是所谓的主goroutine

启动过程中调度器初始化runtime-schedinit函数主要是根据用户设置的GOMAXPROCS来创建一批小车，最多256个（P）。这些P创建好友都是空置状态，所以将他们放置在调度器结构（Sched）的pidle字段维护的链表中存储起来。

主goroutine开始执行后，第一件事就是创建一个新的内核线程，不过是一个特殊线程，其在运行期间只做-系统监控（sysmon）

到此go程序被启动起来了，一个真正干活的Go程序，一定创建有不少的goroutine，所以在Go程序开始运行后，就会向调度器添加goroutine，调度器就要负责维护好这些goroutine的正常执行。

Go 关键字就是用来创建一个goroutine，go关键字对应调度器结构就是run-newproc.其主要作用是制造一块砖头（G）,将砖头（G）放入到但钱工人（M）的小车（P）中。每个新goroutine都有自己的栈，G结构的Sched字段维护了栈地址以及程序计数器等信息，这是基本的调度信息，goroutine放弃啊CPU的时候需要保存这些信息。

创建内核线程：

Go程序中没有关键字让你创建一个内核线程，你只能创建goroutine，内核线程只能由runtime根据实际情况创建，当砖头（G）太多，工人（M）太少，实在忙不过来，刚好还有空闲的小车（P）没有使用，就会创建一个内核线程（M）

|  |
| --- |
| void newm(void (\*fn)(void), P \*p)  newm函数的核心行为就是调用clone系统调用创建一个内核线程，每个内核线程的开始执行位置都是runtime·mstart函数。参数p就是一辆空闲的小车(p)。  每个创建好的内核线程都从runtime·mstart函数开始执行了，它们将用分配给自己小车去搬砖了 |

调度核心：

Newm接口只是给M分配一个空闲P，即告诉M你将使用1号车搬砖，待会自己区停车场那车。M去拿P的过程就是 acquirep

|  |
| --- |
| static void schedule(void) { G \*gp; gp = runqget(m->p); if(gp == nil) gp = findrunnable(); if (m->p->runqhead != m->p->runqtail && runtime·atomicload(&runtime·sched.nmspinning) == 0 && runtime·atomicload(&runtime·sched.npidle) > 0) // TODO: fast atomic wakep(); execute(gp); } |

M拿到自己的小车P就进入工厂开始干活了，即上面的Schedule过程

1. M试图从自己的小车P取出一块砖G，当然可能失败（小车空了）
2. M自己车没有砖会去仓库（全局队列）取砖头，如果仓库夜空了，则会试图从其他M的P里偷一半的砖头，如果多次都失败，则M会把小车还回去，然后sleep(线程sleep)
3. 当M发现自己小车砖头有很多，根本处理不过来，而还有空闲P，并且还有空闲工人M1，则会调度M1调用，分担工作。
4. Execute M拿着砖头放入火种 欢快的烧起来

假设P里有很多砖头，将一个G执行后，何时将其取出，放入下一块。

调度点：

当我们看channel的实现时，发现Channel读写操作都会出发调用runtime-park函数，即park后，这个G会被设置为waiting状态，放弃CPU，并且这个G不会在这个P中，如果不对其调用runtime-ready，其永远不会被执行。

除了park可以放弃cpu外，调用runtime·gosched函数也可以让当前goroutine放弃cpu，但和park完全不同；gosched是将goroutine设置为runnable状态，然后放入到调度器全局等待队列（也就是上面提到的工场仓库，这下就明白为何工场仓库会有砖块(G)了吧）。

若P状态被改为syacall，sysmon监控线程会扫描发现P处于该状态，于是会创建一个新线程把处于syscall的P抢过来，开始干活。而被抢走P的线程等待系统调用返回后，发现没有P了，会将当前G放如仓库（待商榷）

|  |
| --- |
| * goroutine在cpu上换入换出，不断上下文切换的时候，必须要保证的事情就是保存现场和恢复现场，保存现场就是在goroutine放弃cpu的时候，将相关寄存器的值给保存到内存中；恢复现场就是在goroutine重新获得cpu的时调用runtime·gosched函数。goroutine主动放弃CPU，该goroutine会被设置为runnable状态，然后放入一个全局等待队列中，而P将继续执行下一个goroutine。使用runtime·gosched函数是一个主动的行为，一般是在执行长任务时又想其它goroutine得到执行的机会时调用。 * 调用runtime·park函数。goroutine进入waitting状态，除非对其调用runtime·ready函数，否则该goroutine将永远不会得到执行。而P将继续执行下一个goroutine。使用runtime·park函数一般是在某个条件如果得不到满足就不能继续运行下去时调用，当条件满足后需要使用runtime·ready以唤醒它（这里唤醒之后是否会加入全局等待队列还有待研究）。像channel操作，定时器中，网络poll等都有可能park goroutine。 * 慢系统调用。这样的系统调用会阻塞等待，为了使该P上挂着的其它G也能得到执行的机会，需要将这些goroutine转到另一个OS线程上去。具体的做法是：首先将该P设置为syscall状态，然后该线程进入系统调用阻塞等待。之前提到过的sysmom线程会定期扫描所有的P，发现一个P处于了syscall的状态，就将M和P分离（实际上只有当 Syscall 执行时间超出某个阈值时，才会将 M 与 P 分离）。RUNTIME会再分配一个M和这个P绑定，从而继续执行队列中的其它G。而当之前阻塞的M从系统调用中返回后，会将该goroutine放入全局等待队列中，自己则sleep去。   候，需要从内存把之前的寄存器信息全部放回到相应寄存器中去。  goroutine在主动放弃cpu的时候(park/gosched)，都会涉及到调用runtime·mcall函数，此函数也是汇编实现，主要将goroutine的栈地址和程序计数器保存到G结构的sched字段中，mcall就完成了现场保存。恢复现场的函数是runtime·gogocall，这个函数主要在execute中调用，就是在执行goroutine前，需要重新装载相应的寄存器 |

|  |
| --- |
|  |

<https://blog.csdn.net/zdy0_2004/article/details/78624025>

<https://blog.csdn.net/liangzhiyang/article/details/52669851>

**[XiuXiuBlaza](https://my.csdn.net/u013884727" \t "_blank)**2017-12-30 15:45:52#3楼

总结的灰常不错 。 纠正几个地方， （1）什么时候会去创建M：当用go创建一个新的goroutine时，即调用newproc时，会检查是否有idle p，如果有那么就会为这个p找个idle M，如果没有idle M，则会新建M。 （2）P find runnable G：调度时，首先会以1/61的概率从global runq获取G(避免饥饿)，如果没有则从local runq获取G，local runq获取不到就去global runq获取（因为前面仅仅是1/61的概率），如果还没有拿到，就执行netpoll，检查是否有io就绪的G，如果还是没有，那么就只好steal from other p

https://blog.csdn.net/d\_guco/article/details/79179573goroutine最大的价值是其实现了并发协程和实际并行执行的线程的映射以及动态扩展

**https://blog.csdn.net/d\_guco/article/details/79179573**