Goroutine并发调度模型深入之实现一个协程池

原创：潘少 傲来三少 6月4日

> 并发（并行），一直以来都是一个编程语言里的核心主题之一，也是被开发者关注最多的话题；Go语言作为一个出道以来就自带 『高并发』光环的富二代编程语言，它的并发（并行）编程肯定是值得开发者去探究的，而Go语言中的并发（并行）编程是经由goroutine实现的，goroutine是golang最重要的特性之一，具有使用成本低、消耗资源低、能效高等特点，官方宣称原生goroutine并发成千上万不成问题，于是它也成为Gopher们经常使用的特性。
>
> Goroutine是优秀的，但不是完美的，在极大规模的高并发场景下，也可能会暴露出问题，什么问题呢？又有什么可选的解决方案？本文将通过runtime对goroutine的调度分析，帮助大家理解它的机理和发现一些内存和调度的原理和问题，并且基于此提出一种个人的解决方案 — 一个高性能的Goroutine Pool（协程池）。

Goroutine&Scheduler

Goroutine，Go语言基于并发（并行）编程给出的自家的解决方案。goroutine是什么？通常goroutine会被当做coroutine（协程）的 golang实现，从比较粗浅的层面来看，这种认知也算是合理，但实际上，goroutine并非传统意义上的协程，现在主流的线程模型分三种：内核级线程模型、用户级线程模型和两级线程模型（也称混合型线程模型），传统的协程库属于**用户级线程模型**，而goroutine和它的 `Go Scheduler` 在底层实现上其实是属于**两级线程模型**，因此，有时候为了方便理解可以简单把goroutine类比成协程，但心里一定要有个清晰的认知 — goroutine并不等同于协程。

线程那些事儿

互联网时代以降，由于在线用户数量的爆炸，单台服务器处理的连接也水涨船高，迫使编程模式由从前的串行模式升级到并发模型，而几十年来，并发模型也是一代代地升级，有IO多路复用、多进程以及多线程，这几种模型都各有长短，现代复杂的高并发架构大多是几种模型协同使用，不同场景应用不同模型，扬长避短，发挥服务器的最大性能，而多线程，因为其轻量和易用，成为并发编程中使用频率最高的并发模型，而后衍生的协程等其他子产品，也都基于它，而我们今天要分析的 goroutine 也是基于线程，因此，我们先来聊聊线程的三大模型：

线程的实现模型主要有3种：内核级线程模型、用户级线程模型和两级线程模型（也称混合型线程模型），它们之间最大的差异就在于用户线程与内核调度实体（KSE，Kernel Scheduling Entity）之间的对应关系上。而所谓的内核调度实体 KSE 就是指可以被操作系统内核调度器调度的对象实体（这说的啥玩意儿，敢不敢通俗易懂一点？）。简单来说 KSE 就是**内核级线程**，是操作系统内核的最小调度单元，也就是我们写代码的时候通俗理解上的线程了（这么说不就懂了嘛！装什么13）。

用户级线程模型

用户线程与内核线程KSE是多对一（N ：1）的映射模型，多个用户线程的一般从属于单个进程并且多线程的调度是由用户自己的线程库来完成，线程的创建、销毁以及多线程之间的协调等操作都是由用户自己的线程库来负责而无须借助系统调用来实现。一个进程中所有创建的线程都只和同一个KSE在运行时动态绑定，也就是说，操作系统只知道用户进程而对其中的线程是无感知的，内核的所有调度都是基于用户进程。许多语言实现的 **协程库** 基本上都属于这种方式（比如python的gevent）。由于线程调度是在用户层面完成的，也就是相较于内核调度不需要让CPU在用户态和内核态之间切换，这种实现方式相比内核级线程可以做的很轻量级，对系统资源的消耗会小很多，因此可以创建的线程数量与上下文切换所花费的代价也会小得多。但该模型有个原罪：并不能做到真正意义上的并发，假设在某个用户进程上的某个用户线程因为一个阻塞调用（比如I/O阻塞）而被CPU给中断（抢占式调度）了，那么该进程内的所有线程都被阻塞（因为单个用户进程内的线程自调度是没有CPU时钟中断的，从而没有轮转调度），整个进程被挂起。即便是多CPU的机器，也无济于事，因为在用户级线程模型下，一个CPU关联运行的是整个用户进程，进程内的子线程绑定到CPU执行是由用户进程调度的，内部线程对CPU是不可见的，此时可以理解为CPU的调度单位是用户进程。所以很多的**协程库**会把自己一些阻塞的操作重新封装为完全的非阻塞形式，然后在以前要阻塞的点上，主动让出自己，并通过某种方式通知或唤醒其他待执行的用户线程在该KSE上运行，从而避免了内核调度器由于KSE阻塞而做上下文切换，这样整个进程也不会被阻塞了。

内核级线程模型

用户线程与内核线程KSE是一对一（1 ：1）的映射模型，也就是每一个用户线程绑定一个实际的内核线程，而线程的调度则完全交付给操作系统内核去做，应用程序对线程的创建、终止以及同步都基于内核提供的系统调用来完成，大部分编程语言的线程库(比如Java的java.lang.Thread、C++11的std::thread等等)都是对操作系统的线程（内核级线程）的一层封装，创建出来的每个线程与一个独立的KSE静态绑定，因此其调度完全由操作系统内核调度器去做。这种模型的优势和劣势同样明显：优势是实现简单，直接借助操作系统内核的线程以及调度器，所以CPU可以快速切换调度线程，于是多个线程可以同时运行，因此相较于用户级线程模型它真正做到了并行处理；但它的劣势是，由于直接借助了操作系统内核来创建、销毁和以及多个线程之间的上下文切换和调度，因此资源成本大幅上涨，且对性能影响很大。

两级线程模型

两级线程模型是博采众长之后的产物，充分吸收前两种线程模型的优点且尽量规避它们的缺点。在此模型下，用户线程与内核KSE是多对多（N ： M）的映射模型：首先，区别于用户级线程模型，两级线程模型中的一个进程可以与多个内核线程KSE关联，于是进程内的多个线程可以绑定不同的KSE，这点和内核级线程模型相似；其次，又区别于内核级线程模型，它的进程里的所有线程并不与KSE一一绑定，而是可以动态绑定同一个KSE， 当某个KSE因为其绑定的线程的阻塞操作被内核调度出CPU时，其关联的进程中其余用户线程可以重新与其他KSE绑定运行。所以，两级线程模型既不是用户级线程模型那种完全靠自己调度的也不是内核级线程模型完全靠操作系统调度的，而是中间态（自身调度与系统调度协同工作），也就是 — 『薛定谔的模型』（误），因为这种模型的高度复杂性，操作系统内核开发者一般不会使用，所以更多时候是作为第三方库的形式出现，而Go语言中的runtime调度器就是采用的这种实现方案，实现了Goroutine与KSE之间的动态关联，不过Go语言的实现更加高级和优雅；该模型为何被称为两级？**即用户调度器实现用户线程到KSE的『调度』，内核调度器实现KSE到CPU上的『调度』**。

G-P-M 模型概述

每一个OS线程都有一个固定大小的内存块(一般会是2MB)来做栈，这个栈会用来存储当前正在被调用或挂起(指在调用其它函数时)的函数的内部变量。这个固定大小的栈同时很大又很小。因为2MB的栈对于一个小小的goroutine来说是很大的内存浪费，而对于一些复杂的任务（如深度嵌套的递归）来说又显得太小。因此，Go语言做了它自己的『线程』。

在Go语言中，每一个goroutine是一个独立的执行单元，相较于每个OS线程固定分配2M内存的模式，goroutine的栈采取了动态扩容方式， 初始时仅为2KB，随着任务执行按需增长，最大可达1GB（64位机器最大是1G，32位机器最大是256M），且完全由golang自己的调度器 **Go Scheduler** 来调度。此外，GC还会周期性地将不再使用的内存回收，收缩栈空间。 因此，Go程序可以同时并发成千上万个goroutine是得益于它强劲的调度器和高效的内存模型。Go的创造者大概对goroutine的定位就是屠龙刀，因为他们不仅让goroutine作为golang并发编程的最核心组件（开发者的程序都是基于goroutine运行的）而且golang中的许多标准库的实现也到处能见到goroutine的身影，比如net/http这个包，甚至语言本身的组件runtime运行时和GC垃圾回收器都是运行在goroutine上的，作者对goroutine的厚望可见一斑。

任何用户线程最终肯定都是要交由OS线程来执行的，goroutine（称为G）也不例外，但是G并不直接绑定OS线程运行，而是由Goroutine Scheduler中的 P - Logical Processor （逻辑处理器）来作为两者的『中介』，P可以看作是一个抽象的资源或者一个上下文，一个P绑定一个OS线程，在golang的实现里把OS线程抽象成一个数据结构：M，G实际上是由M通过P来进行调度运行的，但是在G的层面来看，P提供了G运行所需的一切

资源和环境，因此在G看来P就是运行它的“CPU”，由 G、P、M 这三种由Go抽象出来的实现，最终形成了Go调度器的基本结构：

- G: 表示Goroutine，每个Goroutine对应一个G结构体，G存储Goroutine的运行堆栈、状态以及任务函数，可重用。G并非执行体，每个G需要绑定到P才能被调度执行。
- P: Processor，表示逻辑处理器， 对G来说，P相当于CPU核，G只有绑定到P(在P的local runq中)才能被调度。对M来说，P提供了相关的执行环境(Context)，如内存分配状态(mcache)，任务队列(G)等，P的数量决定了系统内最大可并行的G的数量（前提：物理CPU核数 >= P的数量），P的数量由用户设置的GOMAXPROCS决定，但是不论GOMAXPROCS设置为多大，P的数量最大为256。
- M: Machine，OS线程抽象，代表着真正执行计算的资源，在绑定有效的P后，进入schedule循环；而schedule循环的机制大致是从Global队列、P的Local队列以及wait队列中获取G，切换到G的执行栈上并执行G的函数，调用goexit做清理工作并回到M，如此反复。M并不保留G状态，这是G可以跨M调度的基础，M的数量是不定的，由Go Runtime调整，为了防止创建过多OS线程导致系统调度不过来，目前默认最大限制为10000个。

关于P，我们需要再絮叨几句，在Go 1.0发布的时候，它的调度器其实G-M模型，也就是没有P的，调度过程全由G和M完成，这个模型暴露出一些问题：

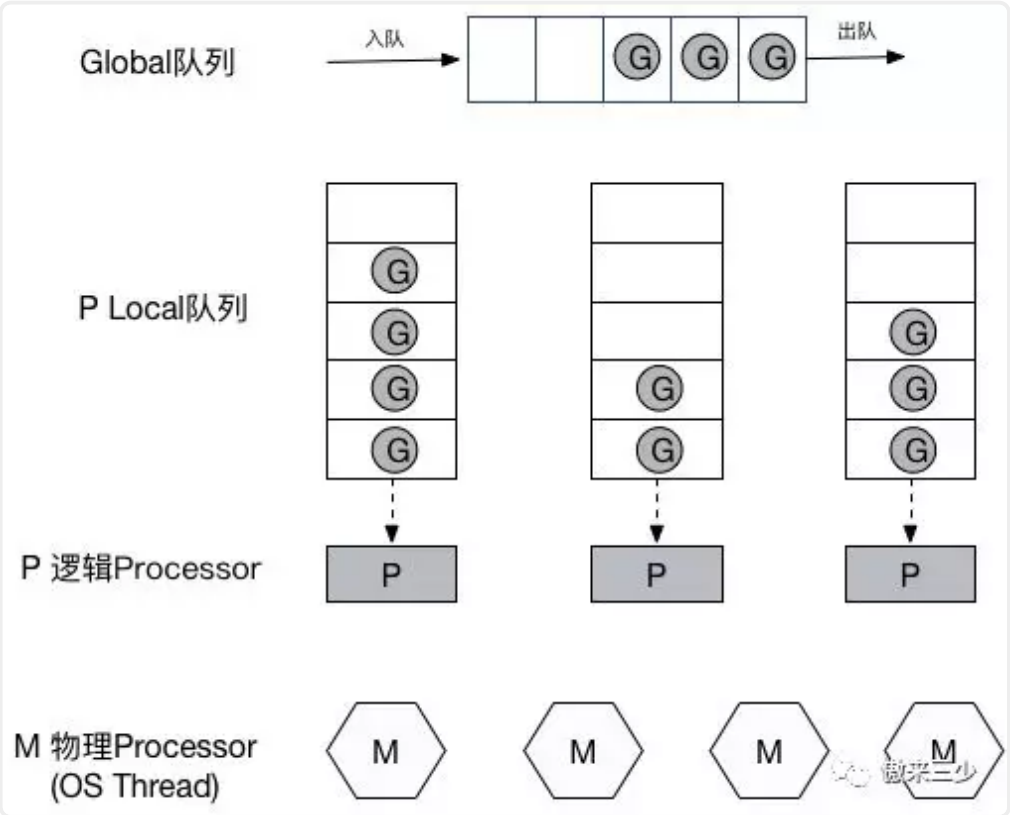
- 单一全局互斥锁(Sched.Lock)和集中状态存储的存在导致所有goroutine相关操作，比如：创建、重新调度等都要上锁；
- goroutine传递问题：M经常在M之间传递『可运行』的goroutine，这导致调度延迟增大以及额外的性能损耗；
- 每个M做内存缓存，导致内存占用过高，数据局部性较差；
- 由于syscall调用而形成的剧烈的worker thread阻塞和解除阻塞，导致额外的性能损耗。

这些问题实在太扎眼了，导致Go1.0虽然号称原生支持并发，却在并发性能上一直饱受诟病，然后，Go语言委员会中一个核心开发大佬看不下了，亲自下场重新设计和实现了Go调度器（在原有的G-M模型中引入了P）并且实现了一个叫做 work-stealing 的调度算法：

- 每个P维护一个G的本地队列；
- 当一个G被创建出来，或者变为可执行状态时，就把他放到P的可执行队列中；
- 当一个G在M里执行结束后，P会从队列中把该G取出；如果此时P的队列为空，即没有其他G可以执行， M就随机选择另外一个P，从其可执行的G队列中取走一半。

该算法避免了在goroutine调度时使用全局锁。

至此，Go调度器的基本模型确立：



G-P-M模型

G-P-M 模型调度

Go调度器工作时会维护两种用来保存G的任务队列：一种是一个Global任务队列，一种是每个P维护的Local任务队列。

当通过 `go` 关键字创建一个新的goroutine的时候，它会优先被放入P的本地队列。为了运行goroutine，M需要持有（绑定）一个P，接着M会启动一个OS线程，循环从P的本地队列里取出一个goroutine并执行。当然还有上文提及的 `work-stealing` 调度算法：当M执行完了当前P的Local队列里的所有G后，P也不会就这么在那躺尸啥都不干，它会先尝试从Global队列寻找G来执行，如果Global队列为空，它会随机挑选另外一个P，从它的队列里中拿走一半的G到自己的队列中执行。

如果一切正常，调度器会以上述的那种方式顺畅地运行，但这个世界没这么美好，总有意外发生，以下分析goroutine在两种例外情况下的行为。

Go runtime会在下面的goroutine被阻塞的情况下运行另外一个goroutine：

- blocking syscall (for example opening a file)
- network input

- channel operations
- primitives in the sync package

这四种场景又可归类为两种类型：

用户态阻塞/唤醒

当goroutine因为channel操作或者network I/O而阻塞时（实际上golang已经用netpoller实现了goroutine网络I/O阻塞不会导致M被阻塞，仅阻塞G，这里仅仅是举个栗子），对应的G会被放置到某个wait队列(如channel的waitq)，该G的状态由 `_Gruning` 变为 `_Gwaitting` ，而M会跳过该G尝试获取并执行下一个G，如果此时没有runnable的G供M运行，那么M将解绑P，并进入sleep状态；当阻塞的G被另一端的G2唤醒时（比如channel的可读/写通知），G被标记为runnable，尝试加入G2所在P的runnext，然后再是P的Local队列和Global队列。

系统调用阻塞

当G被阻塞在某个系统调用上时，此时G会阻塞在 `_Gsyscall` 状态，M也处于 block on syscall 状态，此时的M可被抢占调度：执行该G的M会与P解绑，而P则尝试与其它idle的M绑定，继续执行其它G。如果没有其它idle的M，但P的Local队列中仍然有G需要执行，则创建一个新的M；当系统调用完成后，G会重新尝试获取一个idle的P进入它的Local队列恢复执行，如果没有idle的P，G会被标记为runnable加入到Global队列。

以上就是从宏观的角度对Goroutine和它的调度器进行的一些概要性的介绍，当然，Go的调度中更复杂的抢占式调度、阻塞调度的更多细节，大家可以自行去找相关资料深入理解，本文只讲到Go调度器的基本调度过程，为后面自己实现一个Goroutine Pool提供理论基础，这里便不再继续深入上述说的那几个调度了，事实上如果要完全讲清楚Go调度器，一篇文章的篇幅也实在是捉襟见肘，所以想了解更多细节的同学可以去看看Go调度器 G-P-M 模型的设计者 Dmitry Vyukov 写的该模型的设计文档《Go Preemptive Scheduler Design》以及直接去看源码，G-P-M模型的定义放在 `src/runtime/runtime2.go` 里面，而调度过程则放在了 `src/runtime/proc.go` 里。

大规模Goroutine的瓶颈

既然Go调度器已经这么~~牛逼~~优秀了，我们为什么还要自己去实现一个golang的 Goroutine Pool 呢？事实上，优秀不代表完美，任何不考虑具体应用场景的编程模式都是耍流氓！有

基于G-P-M的Go调度器背书，go程序的并发编程中，可以任性地起大规模的goroutine来执行任务，官方也宣称用golang写并发程序的时候随便起个成千上万的goroutine毫无压力。

然而，你起1000个goroutine没有问题，10000也没有问题，10w个可能也没问题；那，100w个呢？1000w个呢？（这里只是举个极端的例子，实际编程起这么大规模的goroutine的例子极少）这里就会出问题，什么问题呢？

1. 首先，即便每个goroutine只分配2KB的内存，但如果是恐怖如斯的数量，聚少成多，内存暴涨，就会对GC造成极大的负担，写过java的同学应该知道jvm GC那万恶的STW（Stop The World）机制，也就是GC的时候会挂起用户程序直到垃圾回收完，虽然Go1.8之后的GC已经去掉了STW以及优化成了并行GC，性能上有了不小的提升，但是，如果太过于频繁地进行GC，依然会有性能瓶颈；
2. 其次，还记得前面我们说的runtime和GC也都是goroutine吗？是的，如果goroutine规模太大，内存吃紧，runtime调度和垃圾回收同样会出问题，虽然G-P-M模型足够优秀，韩信点兵，多多益善，但你不能不给士兵发口粮（内存）吧？巧妇难为无米之炊，没有内存，Go调度器就会阻塞goroutine，结果就是P的Local队列积压，又导致内存溢出，这就是个死循环...，甚至极有可能程序直接Crash掉，本来是想享受golang并发带来的*快感*效益，结果却得不偿失。

一个http标准库引发的血案

我想，作为golang拥趸的Gopher们一定都使用过它的net/http标准库，很多人都说用golang写web server完全可以不用借助第三方的web framework，仅用net/http标准库就能写一个高性能的web server，的确，我也用过它写过web server，简洁高效，性能表现也相当不错，除非有比较特殊的需求否则一般的确不用借助第三方web framework，但是天下没有白吃的午餐，net/http为啥这么快？要搞清这个问题，从源码入手是最好的途径。孔子曾经曰过：源码面前，如同裸奔。所以，*高清*无码是阻碍程序猿发展大大滴绊脚石啊，源码才是我们进步阶梯，切记切记！

接下来我们就来先看看net/http内部是怎么实现的。

net/http接收请求且开始处理的源码放在 `src/net/http/server.go` 里，先从入口函数 `ListenAndServe` 进去：

```
func (srv *Server) ListenAndServe() error {
  addr := srv.Addr
  if addr == "" {
    addr = ":http"
  }
  ln, err := net.Listen("tcp", addr)
  if err != nil {
    return err
  }
  return srv.Serve(tcpKeepAliveListener{ln.(*net.TCPListener)})
}
```

看到最后那个srv.Serve调用了吗？没错，这个 `Serve` 方法里面就是实际处理http请求的逻辑，我们再进入这个方法内部：

```
func (srv *Server) Serve(l net.Listener) error {
  defer l.Close()
  ...
  // 不断循环取出TCP连接
  for {
    // 看我看我！！！
    rw, e := l.Accept()
    ...
    // 再看我再看我！！！
    go c.serve(ctx)
  }
}
```

首先，这个方法的参数 `(l net.Listener)` ，是一个TCP监听的封装，负责监听网络端口， `rw, e := l.Accept()` 则是一个阻塞操作，从网络端口取出一个新的TCP连接进行处理，最后 `go c.serve(ctx)` 就是最后真正去处理这个http请求的逻辑了，看到前面的go关键字了吗？没错，这里启动了一个新的goroutine去执行处理逻辑，而且这是在一个无限循环体里面，所以意味着，每来一个请求它就会开一个goroutine去处理，相当任性粗暴啊...，不过有Go调度器背书，一般来说也没啥压力，然而，如果，我是说如果哈，突然一大波请求涌进来了（比方说黑客搞了成千上万的肉鸡DDOS你，没错！就这么倒霉！），这时候，就很成问题了，他来10w个请求你就要开给他10w个goroutine，来100w个你就要老老实实开给他100w个，线程调度压力陡升，内存爆满，再然后，你就跪了...

釜底抽薪

有问题，就一定有解决的办法，那么，有什么方案可以减缓大规模goroutine对系统的调度和内存压力？要想解决问题，最重要的是找到造成问题的根源，这个问题根源是什么？goroutine的数量过多导致资源侵占，那要解决这个问题就要限制运行的goroutine数量，合理复用，节省资源，具体就是 — goroutine池化。

超大规模并发的场景下，不加限制的大规模的goroutine可能造成内存暴涨，给机器带来极大的压力，吞吐量下降和处理速度变慢还是其次，更危险的是可能使得程序crash。所以，goroutine池化是有其现实意义的。

首先，100w个任务，是不是真的需要100w个goroutine来处理？未必！用1w个goroutine也一样可以处理，让一个goroutine多处理几个任务就是了嘛，池化的核心优势就在于对goroutine的复用。此举首先极大减轻了runtime调度goroutine的压力，其次，便是降低了对内存的消耗。

有一个商场，来了1000个顾客买东西，那么该如何安排导购员服务这1000人呢？有两种方案：

第一，我雇1000个导购员实行一对一服务，这种当然是最高效的，但是太浪费资源了，雇1000个人的成本极高且管理困难，这些可以先按下不表，但是每个顾客到商场买东西也不是一进来就马上买，一般都得逛一逛，选一选，也就是得花时间挑，1000个导购员一对一盯着，效率极低；这就引出第二种方案：我只雇10个导购员，就在商场里待命，有顾客需要咨询的时候招呼导购员过去进行处理，导购员处理完之后就回来，等下一个顾客需要咨询的时候再去，如此往返反复...

第二种方案有没有觉得很眼熟？没错，其基本思路就是模拟一个I/O多路复用，通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。关于多路复用，不在本文的讨论范围之内，便不再赘述，详细原理可以参考 I/O多路复用。

第一种方案就是net/http标准库采用的：来一个请求开一个goroutine处理；第二种方案就是Goroutine Pool（I/O多路复用）。

实现一个Goroutine Pool

因为上述陈列的一些由于goroutine规模过大而可能引发的问题，需要有方案来解决这些问题，上文已经分析过，把goroutine池化是一种行之有效的方案，基于此，可以实现一个

Goroutine Pool，复用goroutine，减轻runtime的调度压力以及缓解内存压力，依托这些优化，在大规模goroutine并发的场景下可以极大地提高并发性能。

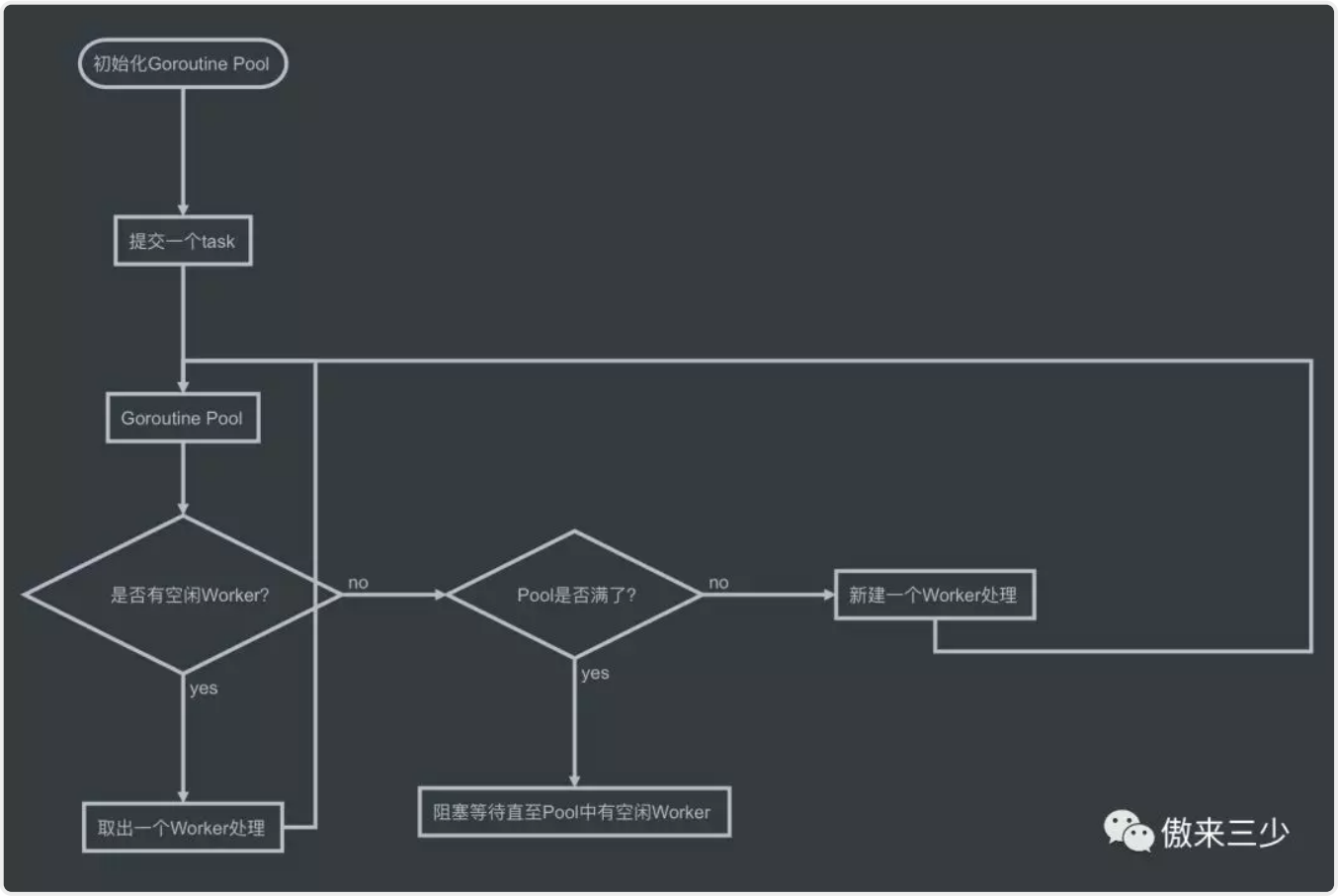
> 哎玛！前面絮絮叨叨了这么多，终于进入正题了，接下来就开始讲解如何实现一个高性能的Goroutine Pool，秒杀原生并发的goroutine，在执行速度和占用内存上提高并发程序的性能。好了，话不多说，开始*装逼*分析。

设计思路

Goroutine Pool 的实现思路大致如下：

> 启动服务之时先初始化一个 Goroutine Pool 池，这个Pool维护了一个类似栈的FILO队列 ，里面存放负责处理任务的Worker，然后在client端提交task到Pool中之后，在Pool内部，接收task之后的核心操作是：
>
> 1. 检查当前Worker队列中是否有空闲的Worker，如果有，取出执行当前的task；
> 2. 没有空闲Worker，判断当前在运行的Worker是否已超过该Pool的容量，是 — 阻塞等待直至有Worker被放回Pool；否 — 新开一个Worker（goroutine）处理；
> 3. 每个Worker执行完任务之后，放回Pool的队列中等待。

调度过程如下：



按照这个设计思路，我实现了一个高性能的Goroutine Pool，较好地解决了上述的大规模调度和资源占用的问题，在执行速度和内存占用方面相较于原生goroutine并发占有明显的优势，尤其是内存占用，因为复用，所以规避了无脑启动大规模goroutine的弊端，可以节省大量的内存。

完整的项目代码可以在我的github上获取：https://github.com/panjf2000/ants，也欢迎提意见和交流。

实现细节

Goroutine Pool的设计原理前面已经讲过了，整个调度过程相信大家应该可以理解了，但是有一句老话说得好，空谈误国，实干兴邦，设计思路有了，具体实现的时候肯定会有很多细节、难点，接下来我们通过分析这个Goroutine Pool的几个核心实现以及它们的联动来引导大家过一遍Goroutine Pool的原理。

首先是 `Pool struct`：

```
1  type sig struct{}
2
3  type f func() error
4
```

```
// Pool accept the tasks from client,it limits the total
// of goroutines to a given number by recycling goroutines.
type Pool struct {
    // capacity of the pool.
    capacity int32

    // running is the number of the currently running goroutines.
    running int32

    // freeSignal is used to notice pool there are available
    // workers which can be sent to work.
    freeSignal chan sig

    // workers is a slice that store the available workers.
    workers []*Worker

    // release is used to notice the pool to closed itself.
    release chan sig

    // lock for synchronous operation
    lock sync.Mutex

    once sync.Once
}
```

`Pool` 是一个通用的协程池，支持不同类型的任务，亦即每一个任务绑定一个函数提交到池中，批量执行不同类型任务，是一种广义的协程池；本项目中还实现了另一种协程池 — 批量执行同类任务的协程池 `PoolWithFunc` ，每一个 `PoolWithFunc` 只会绑定一个任务函数 `pf` ，这种Pool适用于大批量相同任务的场景，因为每个Pool只绑定一个任务函数，因此 `PoolWithFunc` 相较于 `Pool` 会更加节省内存，但通用性就不如前者了，为了让大家更好地理解协程池的原理，这里我们用通用的 `Pool` 来分析。

`capacity` 是该Pool的容量，也就是开启worker数量的上限，每一个worker绑定一个goroutine； `running` 是当前正在执行任务的worker数量； `freeSignal` 是一个信号，因为Pool开启的worker数量有上限，因此当全部worker都在执行任务的时候，新进来的请求就需要阻塞等待，那当执行完任务的worker被放回Pool之时，如何通知阻塞的请求绑定一个空闲的worker运行呢？ `freeSignal` 就是来做这个事情的； `workers` 是一个slice，用来存放空闲worker，请求进入Pool之后会首先检查 `workers` 中是否有空闲worker，若有则取出绑定任务执行，否则判断当前运行的worker是否已经达到容量上限，是—阻塞等待，否—新开一个worker执行任务； `release` 是当关闭该Pool支持通知所有worker退出运行以防goroutine泄露； `lock` 是一个锁，用以支持Pool的同步操作； `once` 用在确保Pool关闭操作只会执行一次。

提交任务到Pool

`p.Submit(task f)` 如下：

```
// Submit submit a task to pool
func (p *Pool) Submit(task f) error {
    if len(p.release) > 0 {
```

```
4         return ErrPoolClosed
5       }
6       w := p.getWorker()
7       w.sendTask(task)
8       return nil
9    }
```

第一个if判断当前Pool是否已被关闭，若是则不再接受新任务，否则获取一个Pool中可用的worker，绑定该 `task` 执行。

获取可用worker（核心）

`p.getWorker()` 源码：

```
1   // getWorker returns a available worker to run the tasks.
2   func (p *Pool) getWorker() *Worker {
3     var w *Worker
4     // 标志，表示当前运行的worker数量是否已达容量上限
5     waiting := false
6
7     // 涉及从workers队列取可用worker，需要加锁
8     p.lock.Lock()
9     workers := p.workers
10    n := len(workers) - 1
11    // 当前worker队列为空(无空闲worker)
12    if n < 0 {
13       // 运行worker数目已达到该Pool的容量上限，置等待标志
14       if p.running >= p.capacity {
15          waiting = true
16       // 否则，运行数目加1
17       } else {
18          p.running++
19       }
20    // 有空闲worker，从队列尾部取出一个使用
21    } else {
22       w = workers[n]
23       workers[n] = nil
24       p.workers = workers[:n]
25    }
26    p.lock.Unlock()
27
28    // 阻塞等待直到有空闲worker
29    if waiting {
30       // 队列有空闲worker通知信号
31       <-p.freeSignal
32       for {
33          p.lock.Lock()
34          workers = p.workers
35          l := len(workers) - 1
36          if l < 0 {
37             p.lock.Unlock()
38             continue
39          }
40          w = workers[l]
41          workers[l] = nil
42          p.workers = workers[:l]
43          p.lock.Unlock()
44          break
45       }
46    // 当前无空闲worker但是Pool还没有满，
47    // 则可以直接新开一个worker执行任务
```

```
48      } else if w == nil {
49          w = &Worker{
50              pool: p,
51              task: make(chan f),
52          }
53          w.run()
54      }
55      return w
56  }
```

上面的源码中加了较为详细的注释，结合前面的设计思路，相信大家应该能理解获取可用worker绑定任务执行这个协程池的核心操作，这里主要关注一个地方：达到Pool容量限制之后，额外的任务请求需要阻塞等待idle worker，这里是为了防止无节制地创建goroutine，事实上Go调度器有一个复用机制，每次使用 `go` 关键字的时候它会检查当前结构体M中的P中，是否有可用的结构体G。如果有，则直接从中取一个，否则，需要分配一个新的结构体G。如果分配了新的G，需要将它挂到runtime的相关队列中，但是调度器却没有限制goroutine的数量，这在瞬时性goroutine爆发的场景下就可能来不及复用G而依然创建了大量的goroutine，所以 `ants` 除了复用还做了限制goroutine数量。

其他部分可以依照注释理解，这里不再赘述。

任务执行

```
1   // Worker is the actual executor who runs the tasks,
2   // it starts a goroutine that accepts tasks and
3   // performs function calls.
4   type Worker struct {
5       // pool who owns this worker.
6       pool *Pool
7
8       // task is a job should be done.
9       task chan f
10  }
11
12  // run starts a goroutine to repeat the process
13  // that performs the function calls.
14  func (w *Worker) run() {
15      go func() {
16          for f := range w.task {
17              if f == nil || len(w.pool.release) > 0 {
18                  atomic.AddInt32(&w.pool.running, -1)
19                  return
20              }
21              f()
22              w.pool.putWorker(w)
23          }
24      }()
25  }
26
27  // stop this worker.
28  func (w *Worker) stop() {
29      w.task <- nil
30  }
31
32  // sendTask sends a task to this worker.
33  func (w *Worker) sendTask(task f) {
```

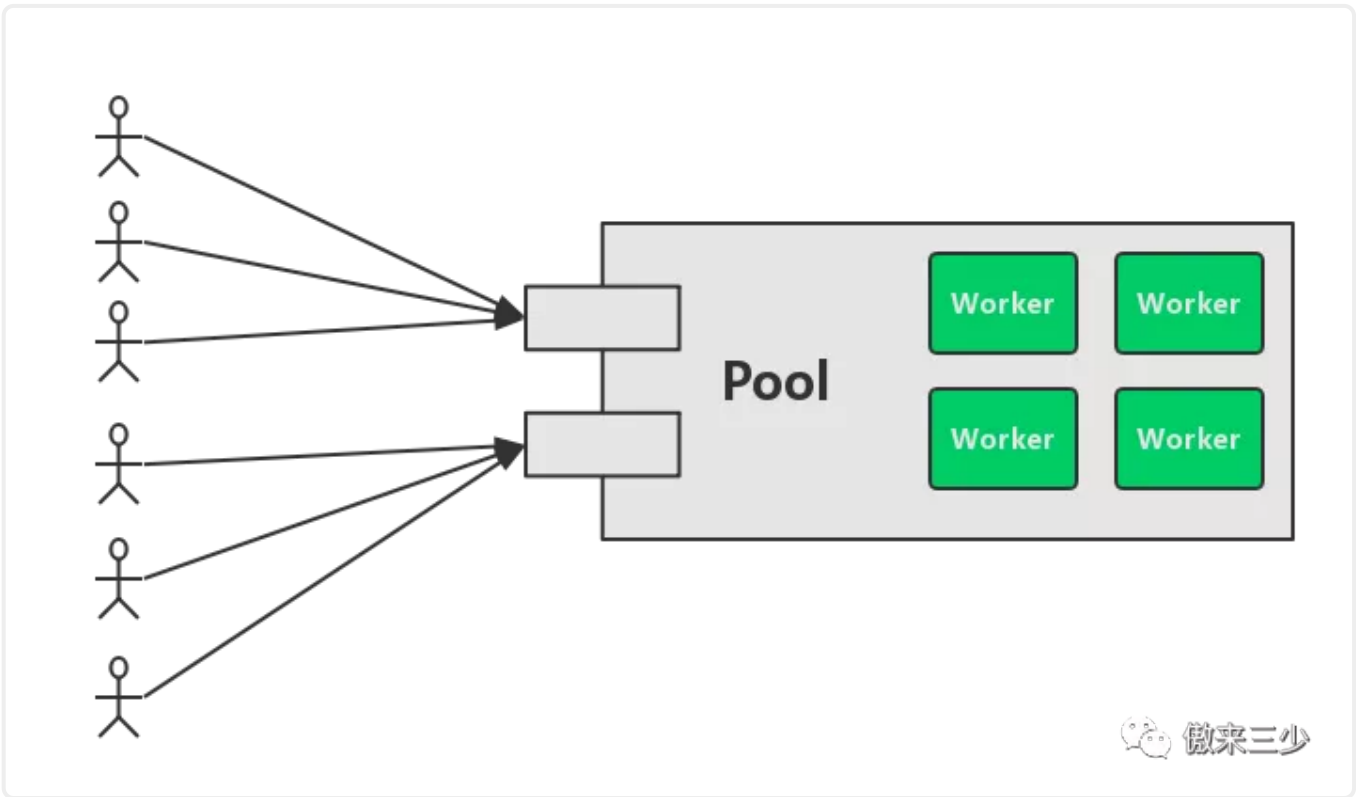
```
34      w.task <- task
35    }
```

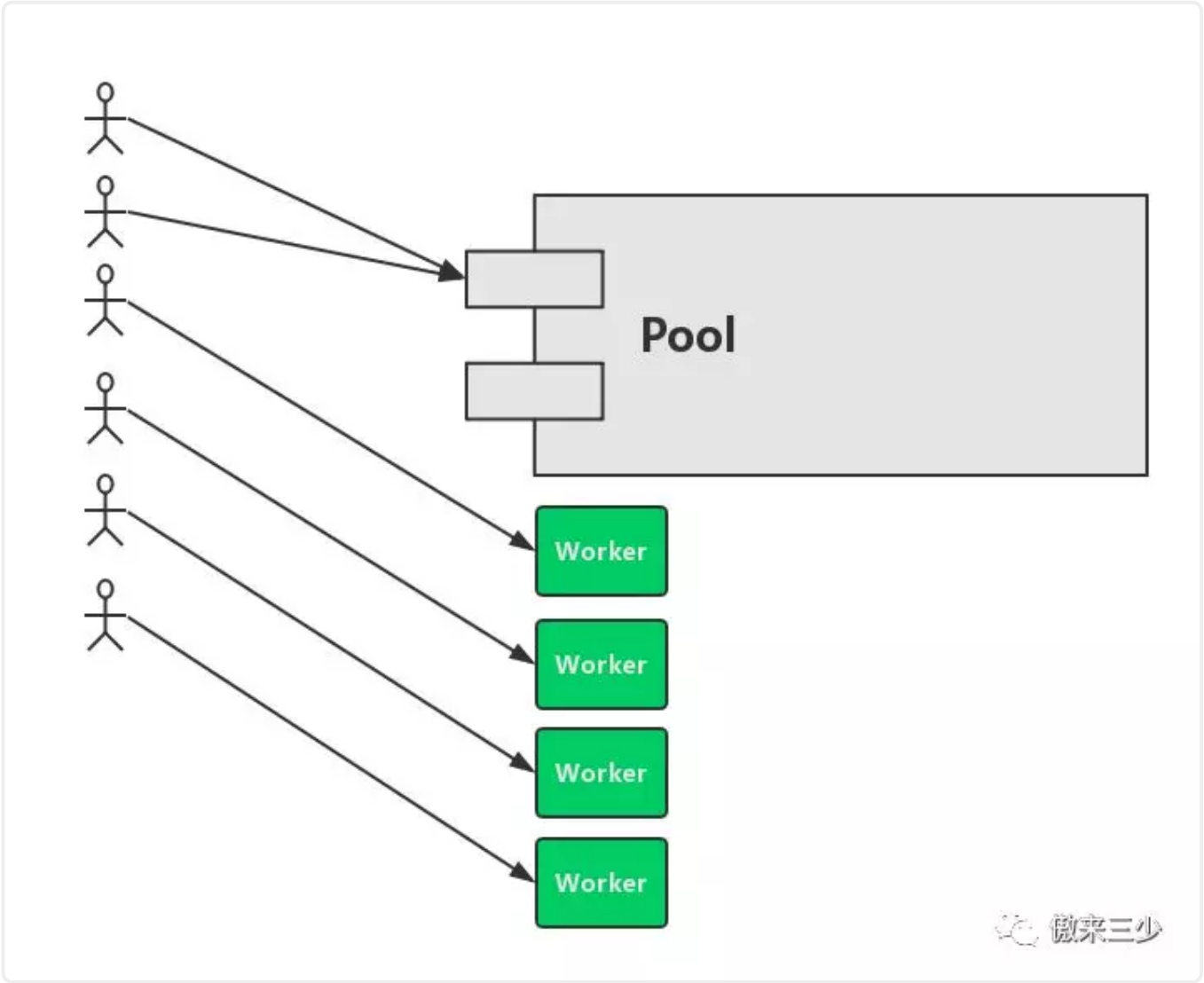
Worker回收（goroutine复用）

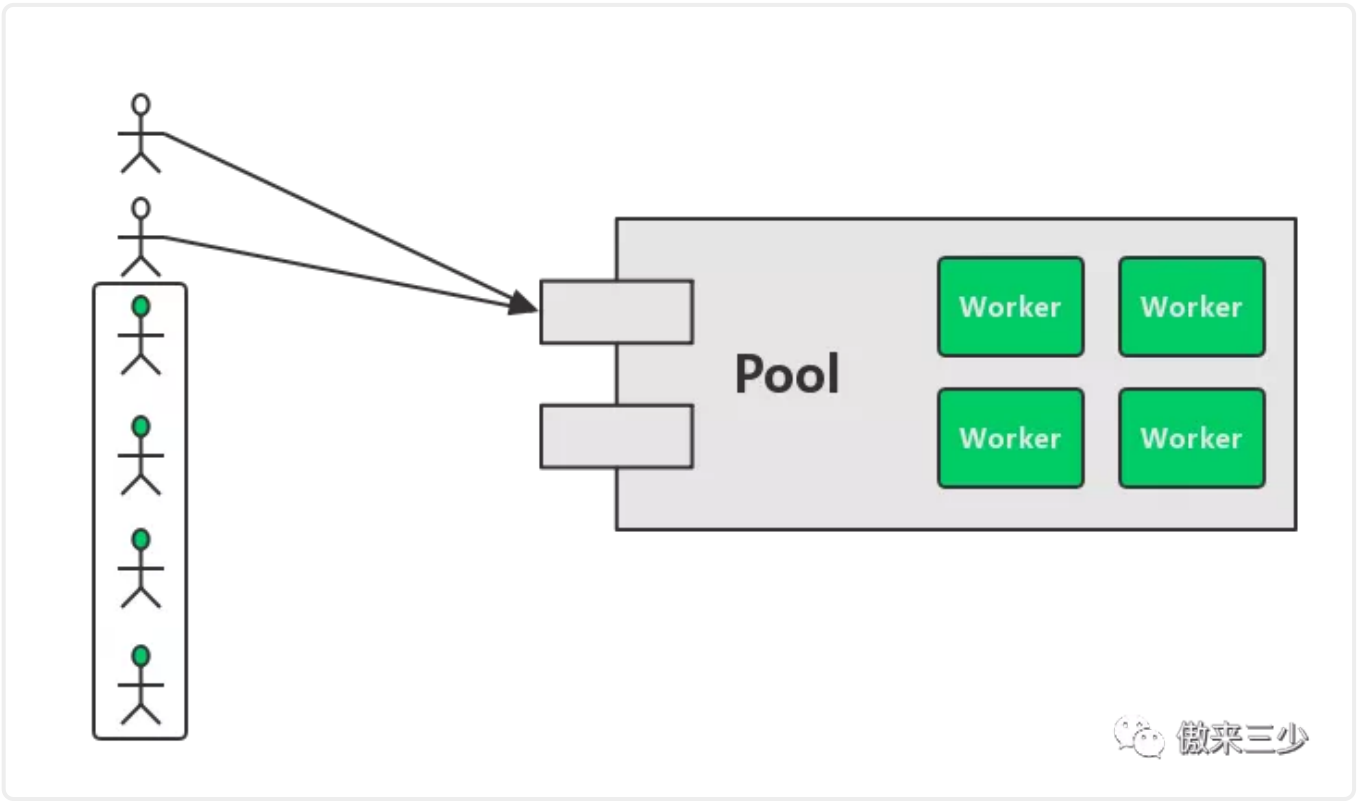
```
1   // putWorker puts a worker back into free pool, recycling the goroutines.
2   func (p *Pool) putWorker(worker *Worker) {
3     p.lock.Lock()
4     p.workers = append(p.workers, worker)
5     p.lock.Unlock()
6     p.freeSignal <- sig{}
7   }
```

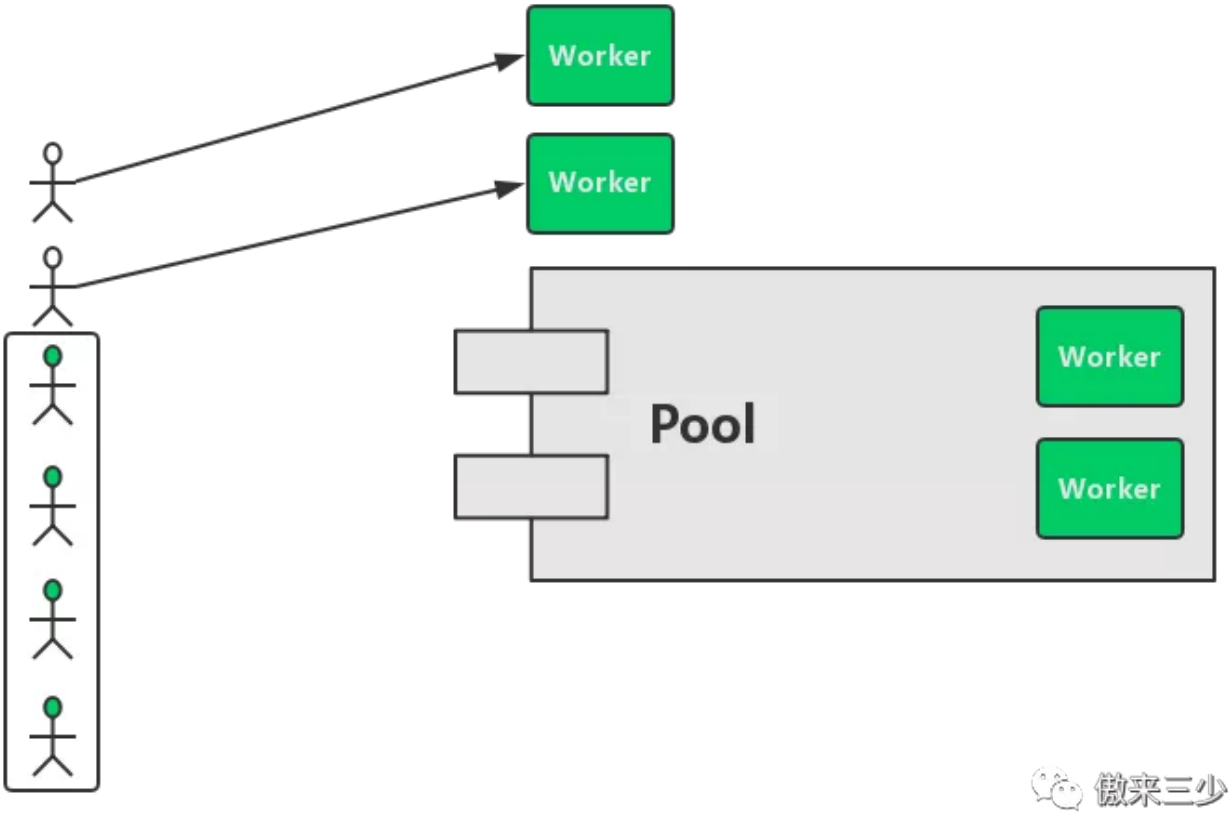
结合前面的 `p.Submit(task f)` 和 `p.getWorker()` ，提交任务到Pool之后，获取一个可用worker，每新建一个worker实例之时都需要调用 `w.run()` 启动一个goroutine监听worker的任务列表 `task` ，一有任务提交进来就执行；所以，当调用worker的 `sendTask(task f)` 方法提交任务到worker的任务队列之后，马上就可以被接收并执行，当任务执行完之后，会调用 `w.pool.putWorker(w *Worker)` 方法将这个已经执行完任务的worker从当前任务解绑放回Pool中，以供下个任务可以使用，至此，一个任务从提交到完成的过程就此结束，Pool调度将进入下一个循环。

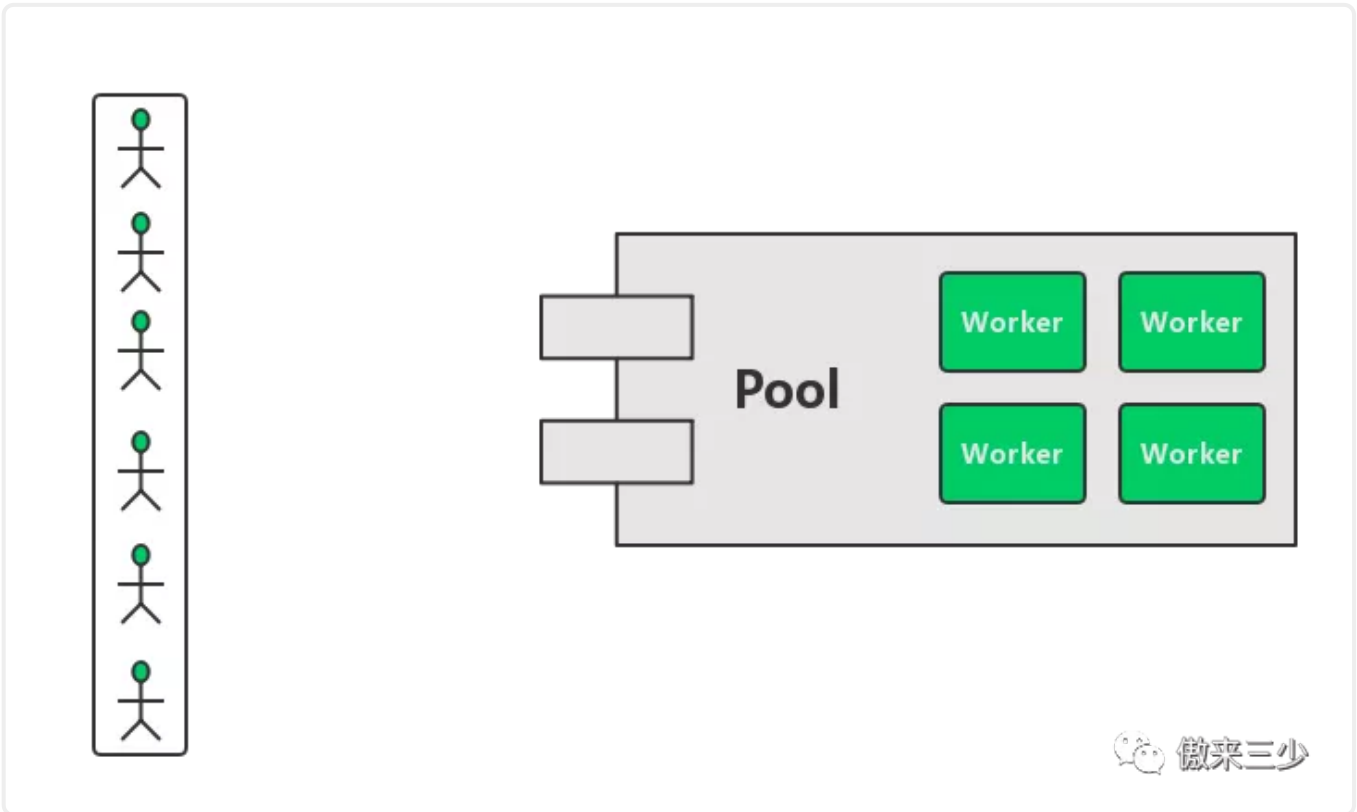
概括起来， `ants` Goroutine Pool的调度过程图示如下：




Pool
Worker
Worker
Worker
Worker
傲来三少


Pool
Worker
Worker
Worker
Worker
傲来三少


Worker
Worker
Pool
Worker
Worker
傲来三少


Pool
Worker
Worker
Worker
Worker
傲来三少

彩蛋

还记得前面我说除了通用的 `Pool struct` 之外，本项目还实现了一个 `PoolWithFunc struct` ——一个执行批量同类任务的协程池， `PoolWithFunc` 相较于 `Pool` ，因为一个池只绑定一个任务函数，省去了每一次task都需要传送一个任务函数的代价，因此其性能优势比起 `Pool` 更明显，这里我们稍微讲一下一个协程池只绑定一个任务函数的细节：

上码！

```
type pf func(interface{}) error

// PoolWithFunc accept the tasks from client,it limits the total
// of goroutines to a given number by recycling goroutines.
type PoolWithFunc struct {
   // capacity of the pool.
   capacity int32

   // running is the number of the currently running goroutines.
   running int32

   // freeSignal is used to notice pool there are available
   // workers which can be sent to work.
   freeSignal chan sig

   // workers is a slice that store the available workers.
   workers []*WorkerWithFunc

   // release is used to notice the pool to closed itself.
   release chan sig

   // lock for synchronous operation
   lock sync.Mutex

   // pf is the function for processing tasks
   poolFunc pf

   once sync.Once
}
```

`PoolWithFunc struct` 中的大部分字段和 `Pool struct` 基本一致，重点关注 `poolFunc pf` ，这是一个函数类型，也就是该Pool绑定的指定任务函数，而client提交到这种类型的Pool的数据就不再是一个任务函数 `task f` 了，而是 `poolFunc pf` 任务函数的形参，然后交由 `WorkerWithFunc` 处理：

```
// WorkerWithFunc is the actual executor who runs the tasks,
// it starts a goroutine that accepts tasks and
// performs function calls.
type WorkerWithFunc struct {
   // pool who owns this worker.
   pool *PoolWithFunc

   // args is a job should be done.
   args chan interface{}
}

// run starts a goroutine to repeat the process
// that performs the function calls.
func (w *WorkerWithFunc) run() {
   go func() {
      for args := range w.args {
         if args == nil || len(w.pool.release) > 0 {
```

```
            atomic.AddInt32(&w.pool.running, -1)
            return
        }
        w.pool.poolFunc(args)
        w.pool.putWorker(w)
    }
}()
}
```

上面的源码可以看到 `WorkerWithFunc` 是一个类似 `Worker` 的结构，只不过监听的是函数的参数队列，每接收到一个参数包，就直接调用 `PoolWithFunc` 绑定好的任务函数 `poolFunc pf` 任务函数执行任务，接下来的流程就和 `Worker` 是一致的了，执行完任务后就把worker放回协程池，等待下次使用。

至于其他逻辑如提交 `task` 、获取 `Worker` 绑定任务等基本复用自 `Pool struct` ，具体细节有细微差别，但原理一致，万变不离其宗，有兴趣的同学可以看我在github上的源码：Goroutine Pool协程池 ants：https://github.com/panjf2000/ants 。

Benchmarks

吹了这么久的Goroutine Pool，那都是虚的，理论上池化可以复用goroutine，提升性能节省内存，没有benchmark数据之前，好像也不能服众哈！所以，本章就来进行一次实测，验证一下再大规模goroutine并发的场景下，Goroutine Pool的表现是不是真的比原生Goroutine并发更好！

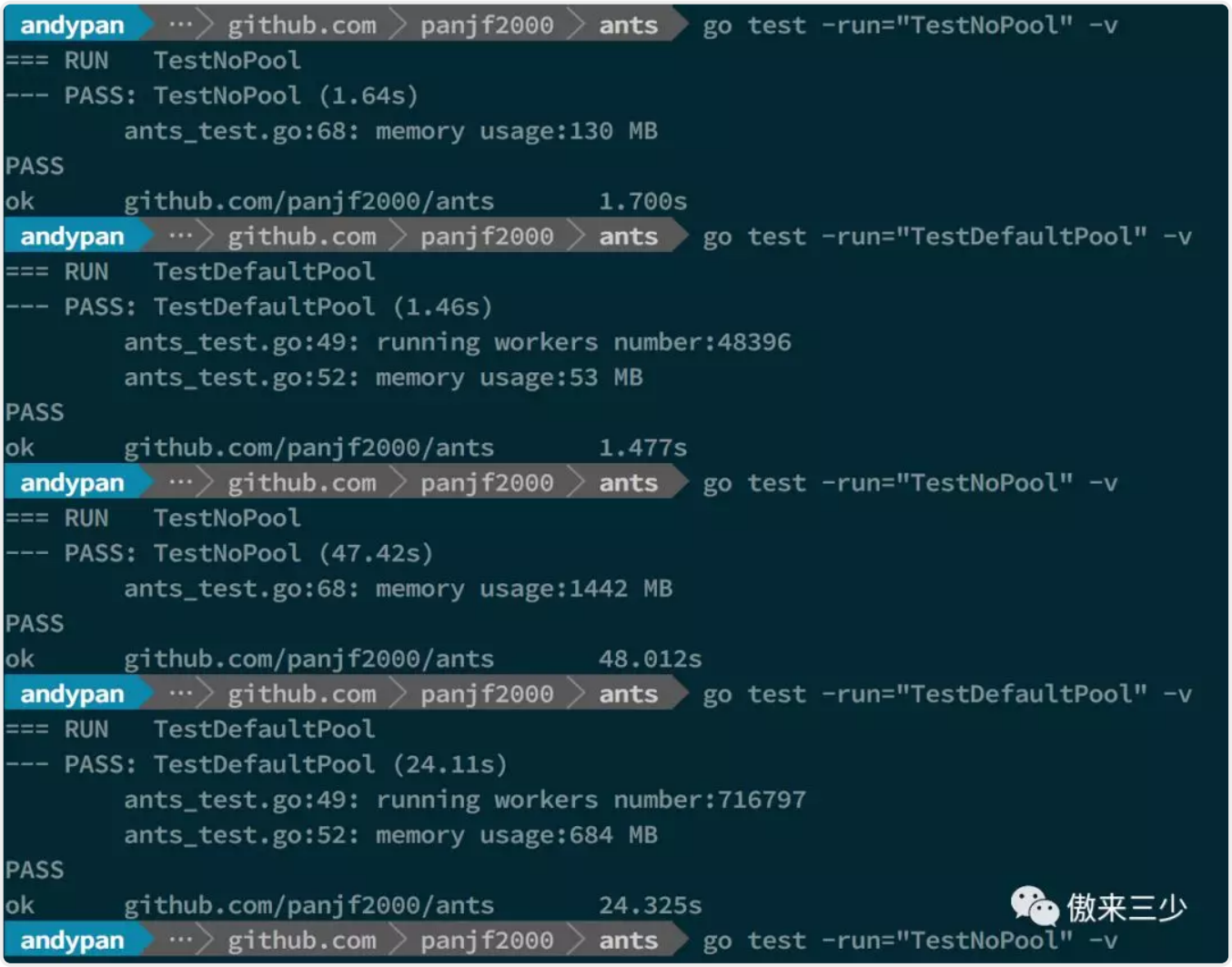
测试机器参数：

```
OS : macOS High Sierra
Processor : 2.7 GHz Intel Core i5
Memory : 8 GB 1867 MHz DDR3
```

Pool测试

测试代码传送门：https://github.com/panjf2000/ants/blob/master/ants_test.go

测试结果：



这里为了模拟大规模goroutine的场景，两次测试的并发次数分别是100w和1000w，前两个测试分别是执行100w个并发任务不使用Pool和使用了 ants 的Goroutine Pool的性能，后两个则是1000w个任务下的表现，可以直观的看出在执行速度和内存使用上，ants 的Pool都占有明显的优势。100w的任务量，使用 ants ，执行速度与原生goroutine相当甚至略快，但只实际使用了不到5w个goroutine完成了全部任务，且内存消耗仅为原生并发的40%；而当任务量达到1000w，优势则更加明显了：用了70w左右的goroutine完成全部任务，执行速度比原生goroutine提高了100%，且内存消耗依旧保持在不使用Pool的40%左右。

PoolWithFunc测试

测试代码传送门：

https://github.com/panjf2000/ants/blob/master/ants_benchmark_test.go

测试结果：

- Benchmarkxxx-4 格式为 基准测试函数名-GOMAXPROCS ，后面的-4代表测试函数运行时对应的CPU核数
- 1 表示执行的次数
- xx ns/op 表示每次的执行时间
- xx B/op 表示每次执行分配的总字节数（内存消耗）
- xx allocs/op 表示每次执行发生了多少次内存分配

因为 `PoolWithFunc` 这个Pool只绑定一个任务函数，也即所有任务都是运行同一个函数，所以相较于 `Pool` 对原生goroutine在执行速度和内存消耗的优势更大，上面的结果可以看出，执行速度可以达到原生goroutine的300%，而内存消耗的优势已经达到了两位数的差距，原生goroutine的内存消耗达到了 `ants` 的35倍且原生goroutine的每次执行的内存分配次数也达到了 `ants` 45倍，1000w的任务量， `ants` 的初始分配容量是5w，因此它完成了所有的任务依旧只使用了5w个goroutine！事实上， `ants` 的Goroutine Pool的容量是可以自定义的，也就是说使用者可以根据不同场景对这个参数进行调优直至达到最高性能。

吞吐量测试

上面的benchmarks出来以后，我当时的内心是这样的：

但是太顺利反而让我疑惑，因为结合我过去这20几年的坎坷人生来看，事情应该不会这么美好才对，果不其然，细细一想，虽然 ants Groutine Pool能在大规模并发下执行速度和内存消耗都对原生goroutine占有明显优势，但前面的测试demo相信大家注意到了，里面使用了WaitGroup，也就是用来对goroutine同步的工具，所以上面的benchmarks中主进程会等待所有子goroutine完成任务后才算完成一次性能测试，然而又有多少场景是单台机器需要扛100w甚至1000w同步任务的？基本没有啊！结果就是造出了屠龙刀，可是世界上没有龙啊！也是无情...

彼时，我内心变成了这样：

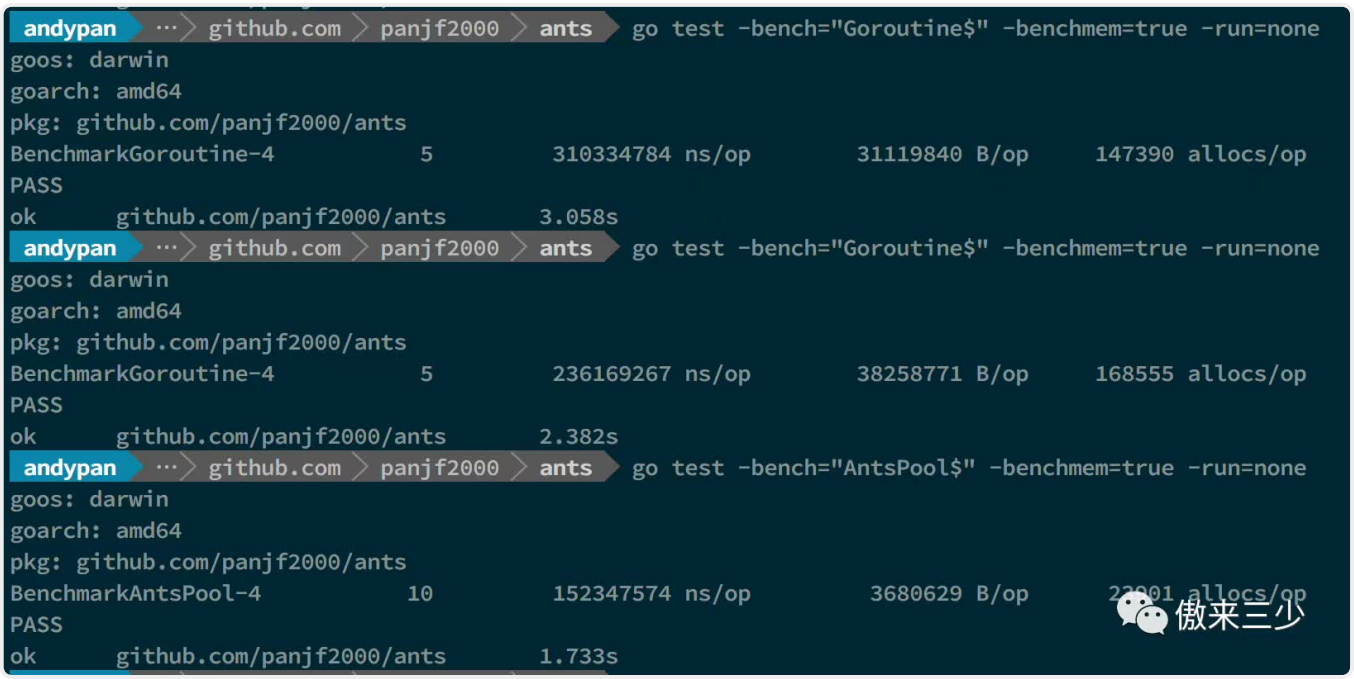
幸好， ants 在同步批量任务方面有点曲高和寡，但是如果是异步批量任务的场景下，就有用武之地了，也就是说，在大批量的任务无须同步等待完成的情况下，可以再测一下 ants 和原生goroutine并发的性能对比，这个时候的性能对比也即是吞吐量对比了，就是在相同大规模数量的请求涌进来的时候， ants 和原生goroutine谁能用更快的速度、更少的内存『吞』完这些请求。

测试代码传送门：

https://github.com/panjf2000/ants/blob/master/ants_benchmark_test.go

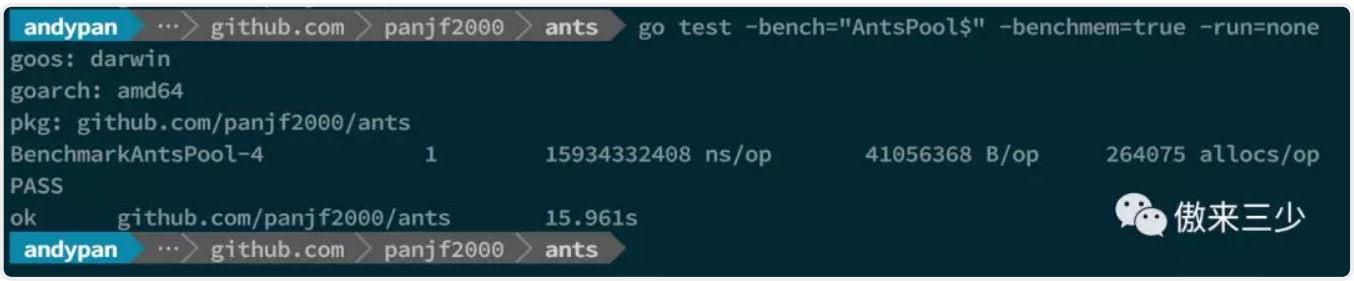
测试结果：

10w 吞吐量



100w 吞吐量

1000W 吞吐量



因为在我的电脑上测试1000w吞吐量的时候原生goroutine已经到了极限，因此程序直接把电脑拖垮了，无法正常测试了，所以1000w吞吐的测试数据只有 `ants` Pool的。

从该demo测试吞吐性能对比可以看出，使用 `ants` 的吞吐性能相较于原生goroutine可以保持在2~6倍的性能压制，而内存消耗则可以达到10~20倍的节省优势。

总结

至此，一个高性能的 Goroutine Pool 开发就完成了，事实上，原理不难理解，总结起来就是一个『复用』，具体落实到代码细节就是锁同步、原子操作、channel通信等这些技巧的使用， `ant` 这整个项目没有借助任何第三方的库，用golang的标准库就完成了所有功能，因为本身golang的语言原生库已经足够优秀，很多时候开发golang项目的时候是可以保持轻量且高性能的，未必事事需要借助第三方库。

关于 `ants` 的价值，其实前文也提及过了， `ants` 在大规模的异步&同步批量任务处理都有着明显的性能优势（特别是异步批量任务），而单机上百万上千万的同步批量任务处理现实意义不大，但是在异步批量任务处理方面有很大的应用价值，所以我个人觉得，Goroutine Pool真正的价值还是在：

1. 限制并发的goroutine数量；
2. 复用goroutine，减轻runtime调度压力，提升程序性能；
3. 规避过多的goroutine侵占系统资源（CPU&内存）。

本文项目源码：https://github.com/panjf2000/ants

后记

Go语言的三位最初的缔造者 — Rob Pike、Robert Griesemer 和 Ken Thompson 中，Robert Griesemer 参与设计了Java的HotSpot虚拟机和Chrome浏览器的JavaScript V8引擎，Rob Pike 在大名鼎鼎的bell lab侵淫多年，参与了Plan9操作系统、C编译器以及多种语言编译器的设计和实现，Ken Thompson 更是图灵奖得主、Unix之父、C语言之父。这三人在计算机史上可是元老级别的人物，特别是 Ken Thompson，是一手缔造了Unix和C语言计算机领域的上古大神，所以Go语言的设计哲学有着深深的Unix烙印：简单、模块化、正交、组合、pipe、功能短小且聚焦等；而令许多开发者青睐于Go的简洁、高效编程模式的原因，也正在于此。

Go语言的三个爸爸

本文从三大线程模型到Go并发调度器再到自定制的 Goroutine Pool，算是较为完整的窥探了整个Go语言并发模型的前世今生，我们也可以看到，Go的设计当然不完美，比如一直被诟病的error处理模式、不支持泛型、差强人意的包管理以及面向对象模式的过度抽象化等等，实际上没有任何一门编程语言敢说自己是完美的，还是那句话，任何不考虑应用场景和语言定位的争执都毫无意义，而Go的定位从出道开始就是系统编程语言&云计算编程语言

（这个有点模糊），而Go的作者们也一直坚持的是用最简单抽象的工程化设计完成最复杂的功能，所以如果从这个层面去看Go的并发模型，就可以看出其实除了G-P-M模型中引入的P，并没有太多革新的原创理论，两级线程模型是早已成熟的理论，抢占式调度更不是什么新鲜的调度模式，Go的伟大之处是在于它诞生之初就是依照Go在谷歌：以软件工程为目的的语言设计而设计的，Go其实就是将这些经典的理论和技术以一种优雅高效的工程化方式组合了起来，并用简单抽象的API或语法糖开放给使用者，Go一直致力于找寻一个高性能&开发效率的双赢点，目前为止，它做得远不够完美，但足够优秀。另外Go通过引入channel与goroutine协同工作，将一种区别于锁&原子操作的并发编程模式 — CSP 带入了Go语言，对开发人员在并发编程模式上的思考有很大的启发。

从本文中对Go调度器的分析以及 `ants` Goroutine Pool 的设计与实现过程，对Go的并发模型做了一次解构和优化思考，在 `ants` 中的代码实现对锁同步、原子操作、channel通信的使用也做了一次较为全面的实践，希望对Gopher们在Go语言并发模型与并发编程的理解上能有所裨益。

感谢阅读。

参考

- Go并发编程实战（第2版）
- Go语言学习笔记
- go-coding-in-go-way
- 也谈goroutine调度器
- [The Go scheduler]

阅读原文