Posted on **2013/03/04**

← Previous

goroutine背后的系统知识

Go语言从诞生到普及已经三年了，先行者大都是Web开发的背景，也有了一些普及型的书籍，可系统开发背景的人在学习这些书籍的时候，总有语焉不详的感觉，网上也有若干流传甚广的文章，可其中或多或少总有些与事实不符的技术描述。希望这篇文章能为比较缺少系统编程背景的Web开发人员介绍一下goroutine背后的系统知识。

1. 操作系统与运行库
2. 并发与并行 (Concurrency and Parallelism)
3. 线程的调度
4. 并发编程框架
5. goroutine

1. 操作系统与运行库

对于普通的电脑用户来说，能理解应用程序是运行在操作系统之上就足够了，可对于开发者，我们还需要了解我们写的程序是如何在操作系统之上运行起来的，操作系统如何为应用程序提供服务，这样我们才能分清楚哪些服务是操作系统提供的，而哪些服务是由我们所使用的语言的运行库提供的。

除了内存管理、文件管理、进程管理、外设管理等等内部模块以外，操作系统还提供了许多外部接口供应用程序使用，这些接口就是所谓的“系统调用”。从DOS时代开始，系统调用就是通过软中断的形式来提供，也就是著名的INT 21，程序把需要调用的功能编号放入AH寄存器，把参数放入其他指定的寄存器，然后调用INT 21，中断返回后，程序从指定的寄存器(通常是AL)里取得返回值。这样的做法一直到奔腾2也就是P6出来之前都没有变，譬如windows通过INT 2E提供系统调用，Linux则是INT 80，只不过后来的寄存器比以前大一些，而且可能再多一层跳转表查询。后来，Intel和AMD分

别提供了效率更高的SYSENTER/SYSEXIT和SYSCALL/SYSRET指令来代替之前的中断方式，略过了耗时的特权级别检查以及寄存器压栈出栈的操作，直接完成从RING 3代码段到RING 0代码段的转换。

系统调用都提供什么功能呢？用操作系统的名字加上对应的中断编号到谷歌上一查就可以得到完整的列表 (Windows, Linux)，这个列表就是操作系统和应用程序之间沟通的协议，如果需要超出此协议的功能，我们就只能在自己的代码里去实现，譬如，对于内存管理，操作系统只提供进程级别的内存段的管理，譬如Windows的virtualmemory系列，或是Linux的brk，操作系统不会去在乎应用程序如何为新建对象分配内存，或是如何做垃圾回收，这些都需要应用程序自己去实现。如果超出此协议的功能无法自己实现，那我们就说该操作系统不支持该功能，举个例子，Linux在2.6之前是不支持多线程的，无论如何在程序里模拟，我们都无法做出多个可以同时运行的并符合POSIX 1003.1c语义标准的调度单元。

可是，我们写程序并不需要去调用中断或是SYSCALL指令，这是因为操作系统提供了一层封装，在Windows上，它是NTDLL.DLL，也就是常说的Native API，我们不但不需要去直接调用INT 2E或SYSCALL，准确的说，我们不能直接去调用INT 2E或SYSCALL，因为Windows并没有公开其调用规范，直接使用INT 2E或SYSCALL无法保证未来的兼容性。在Linux上则没有这个问题，系统调用的列表都是公开的，而且Linus非常看重兼容性，不会去做任何更改，

glibc里甚至专门提供了syscall(2)来方便用户直接用编号调用，不过，为了解决glibc和内核之间不同版本兼容性带来的麻烦，以及为了提高某些调用的效率(譬如__NR_ gettimeofday)，Linux上还是对部分系统调用做了一层封装，就是VDSO (早期叫linux-gate.so)。

可是，我们写程序也很少直接调用NTDLL或者VDSO，而是通过更上一层的封装，这一层处理了参数准备和返回值格式转换、以及出错处理和错误代码转换，这就是我们所使用语言的运行库，对于C语言，Linux上是glibc，Windows上是kernel32(或调用msvcrt)，对于其他语言，譬如Java，则是JRE，这些“其他语言”的运行库通常最终还是调用glibc或kernel32。

“运行库”这个词其实不止包括用于和编译后的目标执行程序进行链接的库文件，也包括了脚本语言或字节码解释型语言的运行环境，譬如Python，C#的CLR，Java的JRE。

对系统调用的封装只是运行库的很小一部分功能，运行库通常还提供了诸如字符串处理、数学计算、常用数据结构容器等等不需要操作系统支持的功能，同时，运行库也会对操作系统支持的功能提供更易用更高级的封装，譬如带缓存和格式的IO、线程池。

所以，在我们说“某某语言新增了某某功能”的时候，通常是这么几种可能：

1. 支持新的语义或语法，从而便于我们描述和解决问题。譬如Java的泛型、Annotation、lambda表达式。
2. 提供了新的工具或类库，减少了我们开发的代码量。譬如Python 2.7的argparse
3. 对系统调用有了更良好更全面的封装，使我们可以做到以前在这个语言环境里做不到或很难做到的事情。譬如Java NIO

但任何一门语言，包括其运行库和运行环境，都不可能创造出操作系统不支持的功能，Go语言也是这样，不管它的特性描述看起来多么炫丽，那必然都是其他语言也可以做到的，只不过Go提供了更方便更清晰的语义和支持，提高了开发的效率。

2. 并发与并行 **(Concurrency and Parallelism)**

并发是指程序的逻辑结构。非并发的程序就是一根竹竿捅到底，只有一个逻辑控制流，也就是顺序执行的(Sequential)程序，在任何时刻，程序只会处在这个逻辑控制流的某个位置。而如果某个程序有多个独立的逻辑控制流，也就是可以同时处理(deal)多件事情，我们就说这个程序是并发的。这里的“同时”，并不一定要是眞正在时钟的某一时刻(那是运行状态而不是逻辑结构)，而是指：如果把这些逻辑控制流画成时序流程图，它们在时间线上是可以重叠的。

并行是指程序的运行状态。如果一个程序在某一时刻被多个CPU流水线同时进行处理，那么我们就说这个程序是以并行的形式在运

行。（严格意义上讲，我们不能说某程序是“并行”的，因为“并行”不是描述程序本身，而是描述程序的运行状态，但这篇小文里就不那么咬文嚼字，以下说到“并行”的时候，就是指代“以并行的形式运行”）显然，并行一定是需要硬件支持的。

而且不难理解：

1. 并发是并行的必要条件，如果一个程序本身就不是并发的，也就是只有一个逻辑控制流，那么我们不可能让其被并行处理。

2. 并发不是并行的充分条件，一个并发的程序，如果只被一个CPU流水线进行处理(通过分时)，那么它就不是并行的。

3. 并发只是更符合现实问题本质的表达方式，并发的最初目的是简化代码逻辑，而不是使程序运行的更快；

这几段略微抽象，我们可以用一个最简单的例子来把这些概念实例化：用C语言写一个最简单的HelloWorld，它就是非并发的，如果我们建立多个线程，每个线程里打印一个HelloWorld，那么这个程序就是并发的，如果这个程序运行在老式的单核CPU上，那么这个并发程序还不是并行的，如果我们用多核多CPU且支持多任务的操作系统来运行它，那么这个并发程序就是并行的。

还有一个略微复杂的例子，更能说明并发不一定可以并行，而且并发不是为了效率，就是Go语言例子里计算素数的sieve.go。我们从小到大针对每一个因子启动一个代码片段，如果当前验证的数能被当前因子除尽，则该数不是素数，如果不能，则把该数发送给下一个因子的代码片段，直到最后一个因子也无法除尽，则该数为素数，我们再启动一个它的代码片段，用于验证更大的数字。这是符合我们计算素数的逻辑的，而且每个因子的代码处理片段都是相同的，所以程序非常的简洁，但它无法被并行，因为每个片段都依赖于前一个片段的处理结果和输出。

并发可以通过以下方式做到：

1. 显式地定义并触发多个代码片段，也就是逻辑控制流，由应用程序或操作系统对它们进行调度。它们可以是独立无关的，也可以是相互依赖需要交互的，譬如上面提到的素数计算，其实它也是个经典的生产者和消费者的问题：两个逻辑控制流A和B，A产生输出，当有了输出后，B取得A的输出进行处理。线程只是实现并发的其中一个手段，除此之外，运行库或是应用程序本身也有多种手段来实现并发，这是下节的主要内容。

2. 隐式地放置多个代码片段，在系统事件发生时触发执行相应的代码片段，也就是事件驱动的方式，譬如某个端口或管道接收到了数据(多路IO的情况下)，再譬如进程接收到了某个信号(signal)。

并行可以在四个层面上做到：

1. 多台机器。自然我们就有了多个CPU流水线，譬如Hadoop集群里的MapReduce任务。

2. 多CPU。不管是真的多颗CPU还是多核还是超线程，总之我们有了多个CPU流水线。

3. 单CPU核里的ILP(Instruction-level parallelism)，指令级并行。通过复杂的制造工艺和对指令的解析以及分支预测和乱序执行，现在的CPU可以在单个时钟周期内执行多条指令，从而，即使是非并发的程序，也可能是以并行的形式执行。

4. 单指令多数据(Single instruction, multiple data. SIMD)，为了多媒体数据的处理，现在的CPU的指令集支持单条指令对多条数据进行操作。

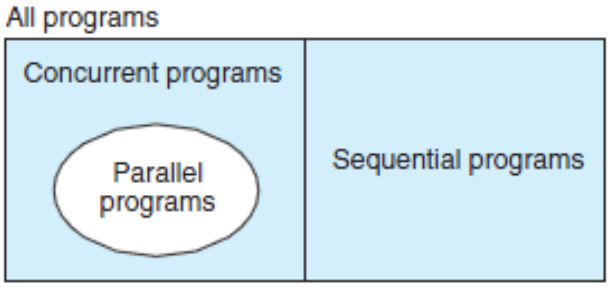
其中，1牵涉到分布式处理，包括数据的分布和任务的同步等等，而且是基于网络的。3和4通常是编译器和CPU的开发人员需要考虑的。这里我们说的并行主要针对第2种：单台机器内的多核CPU并行。

关于并发与并行的问题，Go语言的作者Rob Pike专门就此写过一个

幻灯片：http://talks.golang.org/2012/waza.slide

在CMU那本著名的《Computer Systems: A Programmer's Perspective》里的这张图也非常直观清晰：

Figure 12.30
Relationships between the sets of sequential, concurrent, and parallel programs.



3. 线程的调度

上一节主要说的是并发和并行的概念，而线程是最直观的并发的实现，这一节我们主要说操作系统如何让多个线程并发的执行，当然在多CPU的时候，也就是并行的执行。我们不讨论进程，进程的意义是"隔离的执行环境"，而不是"单独的执行序列"。

我们首先需要理解IA-32 CPU的指令控制方式，这样才能理解如何在多个指令序列(也就是逻辑控制流)之间进行切换。CPU通过CS:EIP寄存器的值确定下一条指令的位置，但是CPU并不允许直接使用MOV指令来更改EIP的值，必须通过JMP系列指令、CALL/RET指令、或INT中断指令来实现代码的跳转；在指令序列间切换的时

候，除了更改EIP之外，我们还要保证代码可能会使用到的各个寄存器的值，尤其是栈指针SS:ESP，以及EFLAGS标志位等，都能够恢复到目标指令序列上次执行到这个位置时候的状态。

线程是操作系统对外提供的服务，应用程序可以通过系统调用让操作系统启动线程，并负责随后的线程调度和切换。我们先考虑单颗单核CPU，操作系统内核与应用程序其实是也是在共享同一个CPU，当EIP在应用程序代码段的时候，内核并没有控制权，内核并不是一个进程或线程，内核只是以实模式运行的，代码段权限为RING 0的内存中的程序，只有当产生中断或是应用程序呼叫系统调用的时候，控制权才转移到内核，在内核里，所有代码都在同一个地址空间，为了给不同的线程提供服务，内核会为每一个线程建立一个内核堆栈，这是线程切换的关键。通常，内核会在时钟中断里或系统调用返回前(考虑到性能，通常是在不频繁发生的系统调用返回前)，对整个系统的线程进行调度，计算当前线程的剩余时间片，如果需要切换，就在“可运行”的线程队列里计算优先级，选出目标线程后，则保存当前线程的运行环境，并恢复目标线程的运行环境，其中最重要的，就是切换堆栈指针ESP，然后再把EIP指向目标线程上次被移出CPU时的指令。Linux内核在实现线程切换时，耍了个花枪，它并不是直接JMP，而是先把ESP切换为目标线程的内核栈，把目标线程的代码地址压栈，然后JMP到__switch_to()，相当于伪造了一个CALL __switch_to()指令，然后，在__switch_to()的最后使用RET指令返回，这样就把栈里的目标线程的代码地址放入了EIP，接下来CPU就开始执行目标线程的代码了，其实也就是

上次停在switch_to这个宏展开的地方。

这里需要补充几点：(1) 虽然IA-32提供了TSS (Task State Segment)，试图简化操作系统进行线程调度的流程，但由于其效率低下，而且并不是通用标准，不利于移植，所以主流操作系统都没有去利用TSS。更严格的说，其实还是用了TSS，因为只有通过TSS才能把堆栈切换到内核堆栈指针SS0:ESP0，但除此之外的TSS的功能就完全没有被使用了。(2) 线程从用户态进入内核的时候，相关的寄存器以及用户态代码段的EIP已经保存了一次，所以，在上面所说的内核态线程切换时，需要保存和恢复的内容并不多。(3) 以上描述的都是抢占式(preemptively)的调度方式，内核以及其中的硬件驱动也会在等待外部资源可用的时候主动调用schedule()，用户态的代码也可以通过sched_yield()系统调用主动发起调度，让出CPU。

现在我们一台普通的PC或服务里通常都有多颗CPU (physical package)，每颗CPU又有多个核 (processor core)，每个核又可以支持超线程 (two logical processors for each core)，也就是逻辑处理器。每个逻辑处理器都有自己的一套完整的寄存器，其中包括了CS:EIP和SS:ESP，从而，以操作系统和应用的角度来看，每个逻辑处理器都是一个单独的流水线。在多处理器的情况下，线程切换的原理和流程其实和单处理器时是基本一致的，内核代码只有一份，当某个CPU上发生时钟中断或是系统调用时，该CPU的CS:EIP

和控制权又回到了内核，内核根据调度策略的结果进行线程切换。但在这个时候，如果我们的程序用线程实现了并发，那么操作系统可以使我们的程序在多个CPU上实现并行。

这里也需要补充两点：(1) 多核的场景里，各个核之间并不是完全对等的，譬如在同一个核上的两个超线程是共享L1/L2缓存的；在有NUMA支持的场景里，每个核访问内存不同区域的延迟是不一样的；所以，多核场景里的线程调度又引入了“调度域”(scheduling domains)的概念，但这不影响我们理解线程切换机制。(2) 多核的场景下，中断发给哪个CPU？软中断(包括除以0，缺页异常，INT指令)自然是在触发该中断的CPU上产生，而硬中断则又分两种情况，一种是每个CPU自己产生的中断，譬如时钟，这是每个CPU处理自己的，还有一种是外部中断，譬如IO，可以通过APIC来指定其送给哪个CPU；因为调度程序只能控制当前的CPU，所以，如果IO中断没有进行均匀的分配的话，那么和IO相关的线程就只能在某些CPU上运行，导致CPU负载不均，进而影响整个系统的效率。

4. 并发编程框架

以上大概介绍了一个用多线程来实现并发的程序是如何被操作系统调度以及并行执行(在有多个逻辑处理器时)，同时大家也可以看到，代码片段或者说逻辑控制流的调度和切换其实并不神秘，理论上，我们也可以不依赖操作系统和其提供的线程，在自己程序的代码段里定义多个片段，然后在我们自己程序里对其进行调度和切

换。

为了描述方便，我们接下来把“代码片段”称为“任务”。

和内核的实现类似，只是我们不需要考虑中断和系统调用，那么，我们的程序本质上就是一个循环，这个循环本身就是调度程序schedule()，我们需要维护一个任务的列表，根据我们定义的策略，先进先出或是有优先级等等，每次从列表里挑选出一个任务，然后恢复各个寄存器的值，并且JMP到该任务上次被暂停的地方，所有这些需要保存的信息都可以作为该任务的属性，存放在任务列表里。

看起来很简单啊，可是我们还需要解决几个问题：

(1) 我们运行在用户态，是没有中断或系统调用这样的机制来打断代码执行的，那么，一旦我们的schedule()代码把控制权交给了任务的代码，我们下次的调度在什么时候发生？答案是，不会发生，只有靠任务主动调用schedule()，我们才有机会进行调度，所以，这里的任务不能像线程一样依赖内核调度从而毫无顾忌的执行，我们的任务里一定要显式的调用schedule()，这就是所谓的协作式(cooperative)调度。(虽然我们可以通过注册信号处理函数来模拟内核里的时钟中断并取得控制权，可问题在于，信号处理函数是由内核调用的，在其结束的时候，内核重新获得控制权，随后返回用户

态并继续沿着信号发生时被中断的代码路径执行，从而我们无法在信号处理函数内进行任务切换)

(2) 堆栈。和内核调度线程的原理一样，我们也需要为每个任务单独分配堆栈，并且把其堆栈信息保存在任务属性里，在任务切换时也保存或恢复当前的SS:ESP。任务堆栈的空间可以是在当前线程的堆栈上分配，也可以是在堆上分配，但通常是在堆上分配比较好：几乎没有大小或任务总数的限制、堆栈大小可以动态扩展(gcc有split stack，但太复杂了)、便于把任务切换到其他线程。

到这里，我们大概知道了如何构造一个并发的编程框架，可如何让任务可以并行的在多个逻辑处理器上执行呢？只有内核才有调度CPU的权限，所以，我们还是必须通过系统调用创建线程，才可以实现并行。在多线程处理多任务的时候，我们还需要考虑几个问题：

(1) 如果某个任务发起了一个系统调用，譬如长时间等待IO，那当前线程就被内核放入了等待调度的队列，岂不是让其他任务都没有机会执行？

在单线程的情况下，我们只有一个解决办法，就是使用非阻塞的IO系统调用，并让出CPU，然后在schedule()里统一进行轮询，有数据时切换回该fd对应的任务；效率略低的做法是不进行统一轮询，让各个任务在轮到自己执行时再次用非阻塞方式进行IO，直到有数

据可用。

如果我们采用多线程来构造我们整个的程序，那么我们可以封装系统调用的接口，当某个任务进入系统调用时，我们就把当前线程留给它(暂时)独享，并开启新的线程来处理其他任务。

(2) 任务同步。譬如我们上节提到的生产者和消费者的例子，如何让消费者在数据还没有被生产出来的时候进入等待，并且在数据可用时触发消费者继续执行呢？

在单线程的情况下，我们可以定义一个结构，其中有变量用于存放交互数据本身，以及数据的当前可用状态，以及负责读写此数据的两个任务的编号。然后我们的并发编程框架再提供read和write方法供任务调用，在read方法里，我们循环检查数据是否可用，如果数据还不可用，我们就调用schedule()让出CPU进入等待；在write方法里，我们往结构里写入数据，更改数据可用状态，然后返回；在schedule()里，我们检查数据可用状态，如果可用，则激活需要读取此数据的任务，该任务继续循环检测数据是否可用，发现可用，读取，更改状态为不可用，返回。代码的简单逻辑如下：

```
struct chan {
    bool ready,
    int data
};
```

```
int read (struct chan *c) {
    while (1) {
        if (c->ready) {
            c->ready = false;
            return c->data;
        } else {
            schedule();
        }
    }
}

void write (struct chan *c, int i) {
    while (1) {
        if (c->ready) {
            schedule();
        } else {
            c->data = i;
            c->ready = true;
            schedule(); // optional
            return;
        }
    }
}
```

很显然，如果是多线程的话，我们需要通过线程库或系统调用提供的同步机制来保护对这个结构体内数据的访问。

以上就是最简化的一个并发框架的设计考虑，在我们实际开发工作中遇到的并发框架可能由于语言和运行库的不同而有所不同，在功能和易用性上也可能各有取舍，但底层的原理都是殊途同归。

譬如，glic里的getcontext/setcontext/swapcontext系列库函数可以

方便的用来保存和恢复任务执行状态；Windows提供了Fiber系列的SDK API；这二者都不是系统调用，getcontext和setcontext的man page虽然是在section 2，但那只是SVR4时的历史遗留问题，其实现代码是在glibc而不是kernel；CreateFiber是在kernel32里提供的，NTDLL里并没有对应的NtCreateFiber。

在其他语言里，我们所谓的“任务”更多时候被称为“协程”，也就是Coroutine。譬如C++里最常用的是Boost.Coroutine；Java因为有一层字节码解释，比较麻烦，但也有支持协程的JVM补丁，或是动态修改字节码以支持协程的项目；PHP和Python的generator和yield其实已经是协程的支持，在此之上可以封装出更通用的协程接口和调度；另外还有原生支持协程的Erlang等，笔者不懂，就不说了，具体可参见Wikipedia的页面：http://en.wikipedia.org/wiki/Coroutine

由于保存和恢复任务执行状态需要访问CPU寄存器，所以相关的运行库也都会列出所支持的CPU列表。

从操作系统层面提供协程以及其并行调度的，好像只有OS X和iOS的Grand Central Dispatch，其大部分功能也是在运行库里实现的。

5. goroutine

Go语言通过goroutine提供了目前为止所有(我所了解的)语言里对于

并发编程的最清晰最直接的支持，Go语言的文档里对其特性也描述的非常全面甚至超过了，在这里，基于我们上面的系统知识介绍，列举一下goroutine的特性，算是小结：

(1) goroutine是Go语言运行库的功能，不是操作系统提供的功能，goroutine不是用线程实现的。具体可参见Go语言源码里的pkg/runtime/proc.c

(2) goroutine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。所以它非常廉价，我们可以很轻松的创建上万个goroutine，但它们并不是被操作系统所调度执行

(3) 除了被系统调用阻塞的线程外，Go运行库最多会启动$GOMAXPROCS个线程来运行goroutine

(4) goroutine是协作式调度的，如果goroutine会执行很长时间，而且不是通过等待读取或写入channel的数据来同步的话，就需要主动调用Gosched()来让出CPU

(5) 和所有其他并发框架里的协程一样，goroutine里所谓“无锁”的优点只在单线程下有效，如果$GOMAXPROCS > 1并且协程间需要通信，Go运行库会负责加锁保护数据，这也是为什么sieve.go这样的例子在多CPU多线程时反而更慢的原因

(6) Web等服务端程序要处理的请求从本质上来讲是并行处理的问题，每个请求基本独立，互不依赖，几乎没有数据交互，这不是一个并发编程的模型，而并发编程框架只是解决了其语义表述的复杂性，并不是从根本上提高处理的效率，也许是并发连接和并发编程的英文都是concurrent吧，很容易产生“并发编程框架和coroutine可以高效处理大量并发连接”的误解。

(7) Go语言运行库封装了异步IO，所以可以写出貌似并发数很多的服务端，可即使我们通过调整$GOMAXPROCS来充分利用多核CPU并行处理，其效率也不如我们利用IO事件驱动设计的、按照事务类型划分好合适比例的线程池。在响应时间上，协作式调度是硬伤。

(8) goroutine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，随着其运行库的不断发展和完善，其性能一定会越来越好，尤其是在CPU核数越来越多的未来，终有一天我们会为了代码的简洁和可维护性而放弃那一点点性能的差别。

This entry was posted in **golang** by **Zhennan**. Bookmark the **permalink**.

14 THOUGHTS ON “GOROUTINE背后的系统知识”

我要去桂林 on **2013/03/07 at 8:51 PM** said:

这么好的文章，怎么没有评论呢？

可能是文章太长，大家看到一半大脑int 21了。

Reply↓

Zhennan

on **2013/03/08 at 11:33 AM** said:

不好意思，可能是我权限设置的问题，之前都没人看，评论全是spam，就设置了需要批准。。。

Reply↓

prettykernel on **2013/03/07 at 9:12 PM** said:

内核只是以实模式运行的，代码段权限为RING 0的内存中的程序

====

内核初始化过程中就会进入保护模式，以后一直在保护模式运行

Reply↓

spin6lock on **2013/03/09 at 6:09 PM** said:

好文章，对于并发和并行区分的很清晰，golang最大的贡献感觉就是比python的gevent更进一步，为并发的协程引入了映射到并行线程的机制。对于新手来说，能够以串行的方式写并发，不需要经常考虑锁的问题，眞是一大福音啊。

Reply↓

尘泥之 on **2013/03/10 at 11:34 PM** said:

一直在找

> 从操作系统开始，

> 逐层讲解如何实现并发/并行/线程/协程

的文章

此文不顶，木有天理

Reply↓

轩脉刃 on **2013/03/13 at 12:22 AM** said:

这文章值得看好几遍啊。。。膜拜

此文不顶，木有天理

Reply↓

haohaoba on **2013/03/18 at 11:39 PM** said:

眞是好文，不过看完一遍只懂了个大概，眞是杯具呀。还得花时间多读几遍，主要是涉及内核、CPU指令的知识，除了让我想起“哦这些其实上学时好像都讲过不过现在好像都忘光了”外，完全不知所云

Reply↓

weager on **2013/03/21 at 4:55 PM** said:

我是golang的粉丝，文章写得很深入，学习了。文中有提到“在响应时间上，协作式调度是硬伤”。但是我看一些benchmark发现golang似乎并不慢，而且golang有互联网时代的c的称号，说明其性能是很不错的。难道这些所谓的性能好都是在指单线程的时候么，golang在多线程时如果使用很多的协程，性能是不是就不好了呢？

Reply↓

Pingback: Web服务的各种并发模型，以及适用场景 - Jagger Wang

Anonymous on **2013/04/01 at 10:54 AM** said:

好文,求作者.

Reply↓

xtaci on **2013/04/14 at 11:24 PM** said:

的确好文~~~

我看了下go的src, 文件IO部分我没有看到O_NONBLOCK的情况， net部分确实有。

Reply↓

roc on **2013/05/08 at 2:19 PM** said:

难得的好文章，谢谢。

Reply↓

est on **2013/05/20 at 10:35 AM** said:

好文啊，为啥顶的人少呢？

> 每个请求基本独立，互不依赖，几乎没有数据交互

不过现代web有点不同了。服务器端开始大量堆积state用来push到客户端。这种需求不小了。

Reply↓

异域幽灵地盘 on **2013/06/04 at 9:36 AM** said:

还在了解Go当中，但是觉得这是一篇不可多得的文章，收藏了，以备后续学习用。感谢版主的分享！！！

Reply↓

Leave a Reply

Your email address will not be published.

Name

Email

Website

Comment

用微博账号登录

Post Comment

Proudly powered by WordPress