LEC 2:Infrastructure :RPC and Threads

线程

是一个非常有用的机构化工作。

是基本的服务器构建工具。

将在实验中经常使用。

线程非常狡猾。

go 中使用 goroutines 代替线程。

线程是操作系统级别的并发机制，可以充分利用 CPU 的多个运算核心，在分布式系统中 IO 并发都会涉及多线程操作，比如启动一个线程等待其他 Server 返回 IO 信息时主线程可以继续处理下一个请求。多线程间可以共享数据信息，但是每个线程都会有自己的程序计数器（PC）、寄存器、栈等，线程是操作系统调度的最小资源单位。

为什么是线程?

它们表达并发性，这在分布式系统中很自然地表现出来。

输入/输出并发:在等待另一台服务器的响应时，处理下一个请求。

多核:线程在几个内核上并行运行。

线程="执行线程"

线程允许一个程序(逻辑上)同时执行许多事情。

线程共享内存。

每个线程包括一些每个线程的状态:程序计数器、寄存器、堆栈。

一个程序有几个线程?

有时由结构驱动，例如每个客户端一个线程，后台任务一个线程。

有时受多核并行性需求的驱动，所以每个内核有一个活动线程，Go 运行时会在可用内核上自动调度可运行的 goroutines。

有时是出于对输入/输出并发性的渴望，该数量由延迟和容量决定，不断增加，直到吞吐量停止增长。

Go 线程组的代价非常低。Go 语言支持程序多创建线程(在 Go 语言中表现为协程，即用户态线程，比操作系统级别的线程机制占用资源更少),100 或 1000 个线程的代价都很低(几百万代价可能就不低了)，通常线程数量比物理核数多，Go 运行环境会调度这些线程在多核上运行，要注意线程也并非免费的，创建线程要比一个函数方法调用更费资源。

线程的挑战

共享数据

一个线程读取数据而另一个线程修改数据?

例如，两个线程做计数加一:count = count+1

这是一场“比赛”,但通常是一个 bug

解决方案：使用互斥体(或其他同步)或避免共享

线程间的协调

如何等待所有映射线程完成？使用 Go channels 或 WaitGroup

并发粒度

粗粒度->简单，但很少并发/并行

细粒度->更多并发、更多竞争和死锁

什么是爬虫?

爬虫目标是获取所有网页，例如,把获取的网页提供给索引器。

将爬取的网页形成图表。

每页有多个链接。

图表有循环。

爬虫的挑战

分配 I/O 并发:同时获取多个网址;提升每秒获取的网址数量,因为网络延迟比网络容量更受限制。

只获取每个网址*一次*:避免浪费网络带宽;善待远程服务器,需要记住访问了哪些网址。

知道什么时候完成。

串行爬虫:

“爬取”的映射避免重复和循环中断。

这是一个单一的映射，通过引用递归调用来传递，但是:一次只取一页。

并发互斥爬虫:

每次爬取页面创建一个线程：越多的并发获取会带来更高的获取速率。

线程共享爬取到的映射。

为什么互斥锁(==锁)?

没有锁:

两个网页包含指向同一网址的链接。

两条线程同时获取这两页。

T1 检查获取了[url]，T2 检查获取了[url]。

两人都看到 url 还没有被提取。

两者都取，这是错误的。

同时读、写(或同时写、写)是一种“竞赛”

并且经常指示一个 bug

该错误可能只出现在不幸的线程交错中

如果我注释掉 Lock()/Unlock()调用，会发生什么情况?

go run crawler.go

go run -race crawler.go

lock 导致检查和更新是原子级的操作

它是如何决定完成的(线程结束爬取完毕)?

sync.WaitGroup。

隐式等待子线程完成递归提取。

ConcurrentChannel 爬虫

a Go channel:

//一个 channel 可以包含一个对象,可以用下面的方式创建多个

ch := make(chan int)

//一个 channel 可以由一个线程发送到另一个线程

ch <- x//直到 goroutine 接收消息之前 sender 都会阻塞

y := <- ch//直到 goroutine 接收消息之前接受者都会阻塞

for y := range ch

因此你可以使用 channel 进行通信和同步

几个线程也可以在一个通道上发送和接收

记住:发送者阻止，直到接收者收到！发送时持有锁可能很危险...

ConcurrentChannel master()

master() creates a worker goroutine to fetch each page

worker() sends URLs on a channel

多个 workers 会发生到同一个 channel 上

master()从 channel 上去读 urls

[diagram: master, channel, workers]

不需要锁定提取的 map，因为它不是共享的!

存在共享的数据吗?

The channel

The slices and strings sent on the channel

The arguments master() passes to worker()

什么时候使用共享和锁定，而不是 channel?

大多数问题都可以用这两种方式解决

用哪种方式解决问题依赖于程序员自身的思考

state -- sharing and locks

communication -- channels

waiting for events -- channels

使用 Go 竞赛检测器:

https://golang.org/doc/articles/race_detector.html

go test –race

Remote Proceduce Call(RPC)

RPC 是分布式系统的关键部分，后续的所有试验都使用 RPC，RPC 的目标是让客户端和服务器通讯部分更易于编程，让客户端和服务器调用更像本地调用。

RPC 消息图:

```
Client          Server
    request--->
        <---response
```

RPC 期望将网络通信做的跟函数调用一样:

```
Client:
    z=fn(x,y)
Server:
    fn(x,y) {
            compute
            return z
    }
```

但实际中很少能够做到如此简单。

软件架构:

```
client app          handlers
  stubs             dispatcher
 RPC lib             RPC lib
    net   ---------- net
```

具体查阅 go 案例，我们来看一些细节:

序列化: Go 的 RPC 库可以传递字符串，数组，对象，map，指针;Go 通过复制传递指针,但是服务端不能调用客户端的指针；有些东西你不能传递：比如 channels 和 function。

RPC 问题:

1.如何处理失败?例如丢包，网络掉线，服务器慢，服务器宕机

2.客户端如何对待 RPC 请求失败？也许客户端永远不会看到来自服务器的响应;也许客户端不知道服务器是否看到了请求； 也许服务器从未看到过请求；也许服务器根据请求已执行完毕，在发送回复之前崩溃了； 也许服务器根据请求已执行完毕，但是网络在传递回复之前就死了。

简单方案是实现“至少一次”机制：RPC 库等待回复一段时间，如果还是没有回复到达，重新发生请求。重复多次，如果还是没有回复，那么返回错误给应用程序。但是需要应用处理可能出现的多个写副本(因为多次请求导致)。

Q:”至少一次”每次都能成功吗?

是的：例如：只读操作，重复执行不执行的操作(数据库检查记录是否已插入)。

更好的方案是实现“至多一次”机制：服务器的 RPC 代码发现重复的请求，返回之前的回复，而不是重写运行。

Q：如何发现相同的请求? client 让每一个请求都带有唯一标示码 XID(unique ID),相同请求使用相同的 XID 重新发送。

server：

```
if seen[xid]:
   r = old[xid]
 else
   r = handler()
    old[xid] = r
   seen[xid] = true
```

实现“至多一次”机制的困难之处(陆陆续续会出现在 lab3 中)：

怎么确认 xid 是唯一的？使用很大的随机数或是将唯一的客户端 ID（比如 ip 地址）和序列号组合起来。

服务器最终必须丢弃关于旧 RPC 的信息，那么旧的 RPC 信息什么时候丢弃是安全的?

思路：

每个客户端都使用唯一 id(通过一个大的随机数)。

上一个客户端 rpc 请求的序列号。

客户端的每一个 RPC 请求包含"seen all replies <=X"

类似 tcp 中的 seq 和 ack。

或者保证每次只允许一个 RPC 调用，到达的是 seq+1，那么丢弃其他小于 seq。

客户端最多可以尝试 5 次，服务器会忽略大于 5 次的请求。

当原来的请求还在执行，怎么样处理相同 seq 的请求？服务器不想运行两次，也不想回复。需要给每个执行的 RPC 标识 pending；等待或者忽略。

如果“至多一次”服务器奔溃或者重启会怎么样?

如果服务器将副本信息保存在内存中，服务器会忘记请求，同时在重启之后接受相同的请求。

也许，你应该将副本信息保存到磁盘?

也许，副本服务器应该保存副本信息?

Go RPC 是"最多一次"的简单形式?

1.打开 TCP 连接。

2.向 TCP 连接写入请求。

3.TCP 也许会重传，但是服务器的 TCP 协议栈会过滤重复的信息，在 Go 代码里面不会有重试（即：不会创建第二个 TCP 连接）。

4.Go RPC 代码当没有获取到回复之后将返回错误。

也许是 TCP 连接的超时。

也许是服务器没有看到请求。

也许服务器处理了请求，但是在返回回复之前服务器的网络故障。

那"恰好一次"呢?

无限制的重试，重复检测以及容错服务(实验 3)。