Golang 之协程详解

liang1101 · 2019-05-25 10:45:09 · 11198 次点击 · 预计阅读时间 6 分钟 · 大约1分钟之前 开始浏览

一、Golang 线程和协程的区别

备注：需要区分进程、线程(内核级线程)、协程(用户级线程)三个概念。

进程、线程 和 协程 之间概念的区别

对于 **进程、线程**，都是有内核进行调度，有 CPU 时间片的概念，进行 **抢占式调度**（有多种调度算法）

对于 **协程**(用户级线程)，这是对内核透明的，也就是系统并不知道有协程的存在，是完全由用户自己的程序进行调度的，因为是由用户程序自己控制，那么就很难像抢占式调度那样做到强制的 CPU 控制权切换到其他进程/线程，通常只能进行 **协作式调度**，需要协程自己主动把控制权转让出去之后，其他协程才能被执行到。

goroutine 和协程区别

本质上，goroutine 就是协程。 不同的是，Golang 在 runtime、系统调用等多方面对 goroutine 调度进行了封装和处理，当遇到长时间执行或者进行系统调用时，会主动把当前 goroutine 的CPU (P) 转让出去，让其他 goroutine 能被调度并执行，也就是 Golang 从语言层面支持了协程。Golang 的一大特色就是从语言层面原生支持协程，在函数或者方法前面加 go关键字就可创建一个协程。

其他方面的比较

1. 内存消耗方面

   每个 goroutine (协程) 默认占用内存远比 Java 、C 的线程少。
   goroutine： 2KB
   线程：8MB

2. 线程和 goroutine 切换调度开销方面

   线程/goroutine 切换开销方面，goroutine 远比线程小
   线程： 涉及模式切换(从用户态切换到内核态)、16个寄存器、PC、SP...等寄存器的刷新等。
   goroutine： 只有三个寄存器的值修改 - PC / SP / DX.

二、协程底层实现原理

线程是操作系统的内核对象，多线程编程时，如果线程数过多，就会导致频繁的上下文切换，这些 cpu 时间是一个额外的耗费。所以在一些高并发的网络服务器编程中，使用一个线程服务一个 socket 连接是很不明智的。于是操作系统提供了基于事件模式的异步编程模型。用少量的线程来服务大量的网络连接和I/O操作。但是采用异步和基于事件的编程模型，复杂化了程序代码的编写，非常容易出错。因为线程穿插，也提高排查错误的难度。

协程，是在应用层模拟的线程，他避免了上下文切换的额外耗费，兼顾了多线程的优点。简化了高并发程序的复杂度。举个例子，一个高并发的网络服务器，每一个socket连接进来，服务器用一个协程来对他进行服务。代码非常清晰。而且兼顾了性能。

那么，协程是怎么实现的呢?

他和线程的原理是一样的，当 a线程 切换到 b线程 的时候，需要将 a线程 的相关执行进度压入栈，然后将 b线程 的执行进度出栈，进入 b线程 的执行序列。协程只不过是在 应用层 实现这一点。但是，协程并不是由操作系统调度的，而且应用程序也没有能力和权限执行 cpu 调度。怎么解决这个问题?

答案是，协程是基于线程的。内部实现上，维护了一组数据结构和 n 个线程，真正的执行还是线程，协程执行的代码被扔进一个待执行队列中，由这 n 个线程从队列中拉出来执行。这就解决了协程的执行问题。那么协程是怎么切换的呢？答案是：golang 对各种 io函数 进行了封装，这些封装的函数提供给应用程序使用，而其内部调用了操作系统的异步 io函数，当这些异步函数返回 busy 或 bloking 时，golang 利用这个时机将现有的执行序列压栈，让线程去拉另外一个协程的代码来执行，基本原理就是这样，利用并封装了操作系统的异步函数。包括 linux 的 epoll、select 和 windows 的 iocp、event 等。

由于golang是从编译器和语言基础库多个层面对协程做了实现，所以，golang的协程是目前各类有协程概念的语言中实现的最完整和成熟的。十万个协程同时运行也毫无压力。关键我们不会这么写代码。但是总体而言，程序员可以在编写 golang 代码的时候，可以更多的关注业务逻辑的实现，更少的在这些关键的基础构件上耗费太多精力。

三、协程的历史以及特点

协程（Coroutine）是在1963年由Melvin E. Conway USAF, Bedford, MA等人提出的一个概念。而且协程的概念是早于线程（Thread）提出的。但是由于协程是**非抢占式**的调度，无法实现公平的任务调用。也无法直接利用多核优势。因此，我们不能武断地说协程是比线程更高级的技术。

尽管，在任务调度上，协程是弱于线程的。但是在资源消耗上，协程则是极低的。一个线程的内存在 MB 级别，而协程只需要 KB 级别。而且线程的调度需要内核态与用户的频繁切入切出，资源消耗也不小。

我们把协程的基本特点归纳为：

1. 协程调度机制无法实现公平调度
2. 协程的资源开销是非常低的，一台普通的服务器就可以支持百万协程。

那么，近几年为何协程的概念可以大热。我认为一个特殊的场景使得协程能够广泛的发挥其优势，并且屏蔽掉了劣势 --> 网络编程。与一般的计算机程序相比，网络编程有其独有的特点。

1. 高并发（每秒钟上千数万的单机访问量）
2. Request/Response。程序生命期端（毫秒，秒级）
3. 高IO，低计算（连接数据库，请求API）。

最开始的网络程序其实就是一个线程一个请求设计的（Apache）。后来，随着网络的普及，诞生了C10K问题。Nginx 通过单线程异步 IO 把网络程序的执行流程进行了乱序化，通过 IO 事件机制最大化的保证了CPU的利用率。

至此，现代网络程序的架构已经形成。基于IO事件调度的异步编程。其代表作恐怕就属 NodeJS 了吧。

异步编程的槽点

异步编程为了追求程序的性能，强行的将线性的程序打乱，程序变得非常的混乱与复杂。对程序状态的管理也变得异常困难。写过Nginx C Module的同学应该知道我说的是什么。我们开始吐槽 NodeJS 那恶心的层层Callback。

Golang

在我们疯狂被 NodeJS 的层层回调恶心到的时候， Golang 作为名门之后开始走入我们的视野。并且迅速的在Web后端极速的跑马圈地。其代表者 Docker 以及围绕这 Docker 展开的整个容器生态圈欣欣向荣起来。其最大的卖点 – 协程 开始真正的流行与讨论起来。

我们开始向写PHP一样来写全异步IO的程序。看上去美好极了，仿佛世界就是这样了。

在网络编程中，我们可以理解为 Golang 的协程本质上其实就是对 IO 事件的封装，并且通过语言级的支持让异步的代码看上去像同步执行的一样。

四、Golang 协程的应用

我们知道，协程（coroutine）是Go语言中的轻量级线程实现，由Go运行时（runtime）管理。

在一个函数调用前加上go关键字，这次调用就会在一个新的goroutine中并发执行。当被调用的函数返回时，这个goroutine也自动结束。需要注意的是，如果这个函数有返回值，那么这个返回值会被丢弃。

先看一下下面的程序代码：

```
func Add(x, y int) {
    z := x + y
    fmt.Println(z)
}

func main() {
    for i:=0; i<10; i++ {
        go Add(i, i)
    }
}
```

执行上面的代码，会发现屏幕什么也没打印出来，程序就退出了。

对于上面的例子，main()函数启动了10个goroutine，然后返回，这时程序就退出了，而被启动的执行 Add() 的 goroutine 没来得及执行。我们想要让 main() 函数等待所有 goroutine 退出后再返回，但如何知道 goroutine 都退出了呢？这就引出了多个goroutine之间通信的问题。

在工程上，有两种最常见的并发通信模型：**共享内存** 和 **消息**。

下面的例子，使用了锁变量（属于一种共享内存）来同步协程，事实上 Go 语言主要使用消息机制（channel）来作为通信模型

```
package main

import (
        "fmt"
        "sync"
        "runtime"
)

var counter int = 0

func Count(lock *sync.Mutex) {
        lock.Lock()     // 上锁
        counter++
        fmt.Println("counter =", counter)
        lock.Unlock()   // 解锁
}

func main() {
        lock := &sync.Mutex{}

        for i:=0; i<10; i++ {
                go Count(lock)
        }
        for {
                lock.Lock()     // 上锁
                c := counter
                lock.Unlock()   // 解锁

                runtime.Gosched() // 出让时间片

                if c >= 10 {
                        break
                }
        }
}
```

channel

消息机制认为每个并发单元是自包含的、独立的个体，并且都有自己的变量，但在不同并发单元间这些变量不共享。每个并发单元的输入和输出只有一种，那就是消息。

channel 是 Go 语言在语言级别提供的 goroutine 间的通信方式，我们可以使用 channel 在多个 goroutine 之间传递消息。channel是进程内的通信方式，因此通过 channel 传递对象的过程和调用函数时的参数传递行为比较一致，比如也可以传递指针等。channel 是类型相关的，一个 channel 只能传递一种类型的值，这个类型需要在声明 channel 时指定。

channel的声明形式为：

```
var chanName chan ElementType
```

举个例子，声明一个传递int类型的channel：

```
var ch chan int
```

使用内置函数 make() 定义一个channel：

```
ch := make(chan int)
```

在channel的用法中，最常见的包括写入和读出：

```
// 将一个数据value写入至channel，这会导致阻塞，直到有其他goroutine从这个channel中读取数据
ch <- value

// 从channel中读取数据，如果channel之前没有写入数据，也会导致阻塞，直到channel中被写入数据为止
value := <-ch
```

默认情况下，channel的接收和发送都是阻塞的，除非另一端已准备好。

我们还可以创建一个带缓冲的channel：

```
c := make(chan int, 1024)

// 从带缓冲的channel中读数据
for i:=range c {
    ...
}
```

此时，创建一个大小为1024的int类型的channel，即使没有读取方，写入方也可以一直往channel里写入，在缓冲区被填完之前都不会阻塞。

可以关闭不再使用的channel：

```
close(ch)
```

应该在生产者的地方关闭channel，如果在消费者的地方关闭，容易引起panic；

现在利用channel来重写上面的例子：

```
func Count(ch chan int) {
    ch <- 1
    fmt.Println("Counting")
}

func main() {

    chs := make([] chan int, 10)

    for i:=0; i<10; i++ {
        chs[i] = make(chan int)
        go Count(chs[i])
    }

    for _, ch := range(chs) {
        <-ch
    }
}
```

在这个例子中，定义了一个包含10个channel的数组，并把数组中的每个channel分配给10个不同的goroutine。在每个goroutine完成后，向goroutine写入一个数据，在这个channel被读取前，这个操作是阻塞的。在所有的goroutine启动完成后，依次从10个channel中读取数据，在对应的channel写入数据前，这个操作也是阻塞的。这样，就用channel实现了类似锁的功能，并保证了所有goroutine完成后main()才返回。

另外，我们在将一个channel变量传递到一个函数时，可以通过将其指定为单向channel变量，从而限制该函数中可以对此channel的操作。

select

在UNIX中，select()函数用来监控一组描述符，该机制常被用于实现高并发的socket服务器程序。Go语言直接在语言级别支持select关键字，用于处理异步IO问题，大致结构如下：

```
select {
    case <- chan1:
    // 如果chan1成功读到数据

    case chan2 <- 1:
    // 如果成功向chan2写入数据

    default:
    // 默认分支
}
```

select默认是阻塞的，只有当监听的channel中有发送或接收可以进行时才会运行，当多个channel都准备好的时候，select是随机的选择一个执行的。

Go语言没有对channel提供直接的超时处理机制，但我们可以利用select来间接实现，例如：

```
timeout := make(chan bool, 1)

go func() {
    time.Sleep(1e9)
    timeout <- true
}()

switch {
    case <- ch:
    // 从ch中读取到数据

    case <- timeout:
    // 没有从ch中读取到数据，但从timeout中读取到了数据
}
```

这样使用select就可以避免永久等待的问题，因为程序会在timeout中获取到一个数据后继续执行，而无论对ch的读取是否还处于等待状态。
