第三节课 The 3rd class - Go 并发编程

前置知识

理解串行(Serial)、并行(Parallel)及并发(Concurrent)

串行：串行是指多个任务时，各个任务按顺序执行，完成一个之后才能进行下一个。

> 一个人写三份作业，一份一份地写完

并行：并行是指两个或多个事件在同一时间间隔内发生，并行的关键是你有**同时处理**多个任务的能力。

> 三个人同时写三份作业

并发：并发是指两个或多个事件在同一时间间隔发生，并发的关键是你有处理多个任务的能力，**不一定要同时。**

> 一个人同时写三份作业，并且每次写一道题，三份作业交替进行

深入理解

上述栗子中，“人”可以看作是cpu，具有处理任务的能力。

我们在之前的学习中，当我们执行我们写的main.go中的代码时，我们的程序是像**串行**这样**顺序逐过程执行**的。

而**并行**指**同一时刻同时**执行多个任务。但是在同一时刻，cpu只能干其中的一件事，所以要想并行的处理多个任务，就要**多个cpu(逻辑核)**参与（相对于多个人同时作业）。并且，**无论从微观上看还是宏观上看**，并行是可以看到两个任务在同时进行的。

并发是指**同一时间段处理多个任务。**在上述例子中，在一段时间内一个人做了三份作业。如果把作业之间切换的时间缩短，宏观上是不是就可以认为三份作业是一起进行的呢？这就不得不提cpu可以做到**极短时间内**切换一次任务了。所以，并发在**宏观**上可以认为两个或多个是同时进行的，但是在**微观**上它们在一个时刻只有一个任务在执行。

所以我认为并行和并发最关键的点就是：是否**任务在微观上是否同时进行。**

在并发程序中可以**同时拥有两个或者多个线程。**这意味着，如果程序在**单核处理器上运行**，那么这两个线程将**交替地换入或者换出内存。**这些线程是同时“存在”的——每个线程都处于执行过程中的某个状态。如果程序能够**并行执行**，那么就一定是运行在**多核处理器上。**此时，程序中的每个线程都将分配到一个独立的处理器核上，因此可以同时运行。

> 在现在的大多数操作系统中，进程调度的过程往往更加复杂，一个CPU也可以在短时间片段内快速切换上下文并行启动多个进程

进程，线程及协程

程序(Program)

程序是指令的有序集合，它本身没有任何运行的含义，只是一个**静态的实体**，一般对应于操作系统中的一个**可执行文件**

进程(Process)

进程是一种**抽象**的概念，从来没有统一的标准定义。

> 假如我们**深夜emo**想要打开网抑云音乐，然后边听歌边玩**原神**，则要分别启网易云和原神的**可执行程序**，然后加载到内存中运行，这就产生了进程。网易云是一个进程，原神也是一个进程

它有自己的生命周期。反映了一个程序在一定的数据集上运行的全部动态过程。

1. 进程是程序的一次动态执行过程， 占用特定的地址空间。
2. 每个进程由3部分组成：CPU、data、code。每个进程都是**独立的**，但也可使用IPC，socks等方式进行通信，保有自己的CPU时间，代码和数据，即便用同一份程序产生好几个进程，它们之间还是**拥有自己的这3样东西**，但这样仍有缺点：浪费内存，CPU的负担较重。
3. 多任务(Multitasking)操作系统将CPU时间**动态地划分给每个进程**，操作系统同时执行多个进程，**每个进程独立运行。**以进程的观点来看，它会以为自己**独占CPU的使用权。**

正在运行的进程可以通过我们电脑上的任务管理器上看见（ctrl+shift+esc）

线程(Thread)

进程可以产生多个线程。同多个进程可以共享操作系统的某些资源一样，同一进程的多个线程也可以**共享此进程的某些资源(比如：代码、数据)**，所以线程又被称为**轻量级进程(lightweight process)**

而对于线程，有这么几个特点

1. 一个进程内部的一个**执行单元**，它是程序中的一个单一的顺序控制流程。
2. 一个进程可拥有多个**并行的(concurrent)线程。**（并发是什么，后面会做出解释）
3. 一个进程中的**多个线程共享相同的内存单元/内存地址空间**，可以访问相同的变量和对象，而且它们从同一堆中分配对象并进行通信、数据交换和同步操作。
4. 由于线程间的通信是在同一地址空间上进行的，所以**不需要额外的通信机制**，这就使得通信更简便而且信息传递的速度也更快。
5. 线程的启动、中断、消亡，**消耗的资源非常少。**

协程(Coroutine)

协程是一种比线程更轻量级的并发单元。它们在同一个线程中执行，但可以在执行过程中暂停和恢复，从而实现并发执行，相对于以上两者适合**IO密集型任务。**协程的特点和优势如下：

1. **轻量级**：协程的启动和切换开销非常小，通常只需要几个 CPU 指令。相比之下，线程的启动和切换开销要大得多。
2. **非抢占式调度**：协程由程序显式地控制何时暂停和恢复，而不是由操作系统调度。这使得协程的执行更加可预测和高效。
3. **共享内存**：同一线程中的多个协程共享相同的内存空间，可以直接访问和修改共享数据，而不需要像进程间通信那样复杂的机制。
4. **易于管理**：协程的生命周期由程序控制，开发者可以更灵活地管理协程的创建、暂停、恢复和销毁。

三者的区别

常规的进程、线程和协程

小结

1. 进程要分配一大部分的内存，而线程只需要分配一部分栈就可以了.
2. 一个程序至少有一个进程,一个进程至少有一个线程.
3. 进程是资源分配的最小单位，线程是程序执行的最小单位。
4. 一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以**并行**执行

正文

Go并发基础

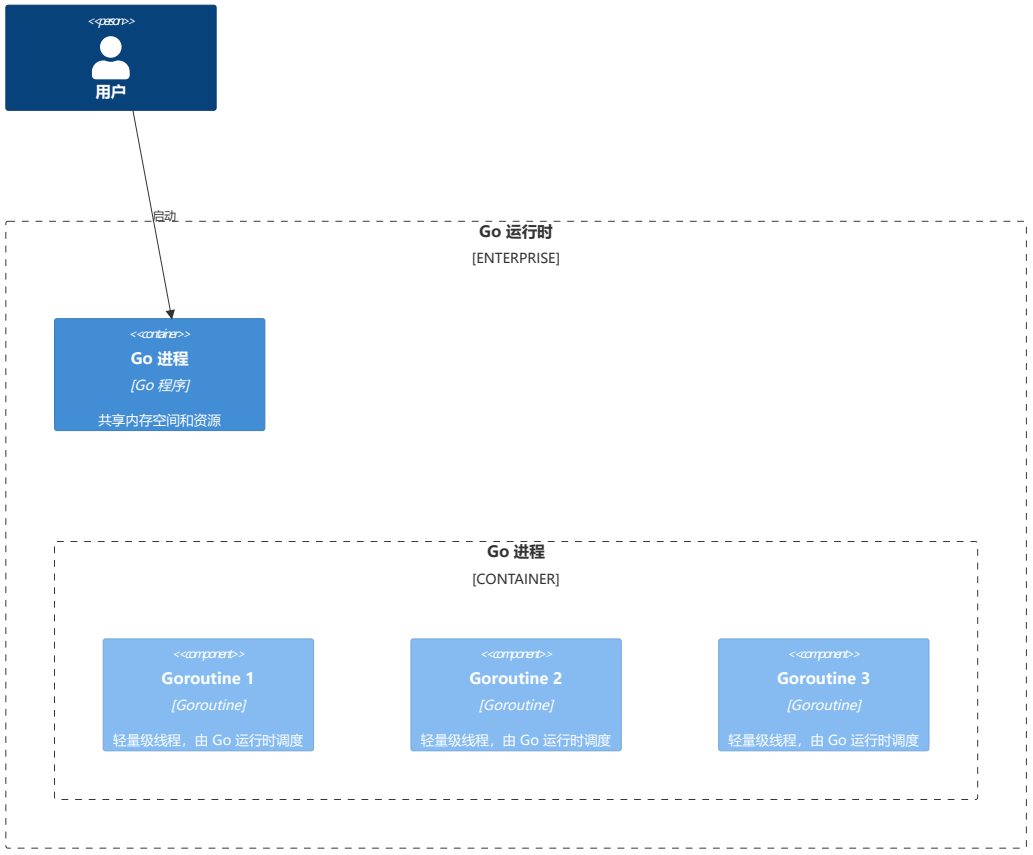
> 高并发性是go的优势之一，理解go的并发后有助于开发io密集型的应用程序

Goroutine(Go 协程)

Goroutine 是 Go 语言中的一种轻量级**线程**--(Go协程)，与传统的线程及协程有区别，粒度小于线程。它们由 **Go 运行时管理**，具有以下特点：

1. **自动调度**：相较于传统线程由内核管理，Goroutine 的调度由 Go 运行时自动管理，工作在用户态，开发者不需要显式地控制调度。
2. **并发执行**：多个 goroutine 可以在同一个进程中**并发**执行，Go 运行时会在多个 goroutine 之间切换，使得它们看起来像是同时运行。
3. **共享内存**：多个 goroutine 共享同一个进程的内存空间，可以直接访问和修改共享数据。
4. **通道通信**：Goroutine 之间可以通过通道（channel）进行通信和同步

Go 的 goroutine



在go中启动一个goroutine非常简单，只需用go `function_name`即可启动一个goroutine

```go
package main

import (
    "fmt"
    "time"
)

func keepAlive() {
    for {
        time.Sleep(5 * time.Second)
        fmt.Println("Heartbeat")
    }
}

func main() {
    go keepAlive()  // 此处启动一个goroutine
    fmt.Println("Goroutine started")
    select {}   // 保持主程序不退出
}
```

运行输出

```
Goroutine started
Heartbeat
Heartbeat
Heartbeat
Heartbeat
...
```

可以看到函数keepAlive已经被「挂起」运行了，每5秒输出一个心跳，这个无限循环并不会阻塞我们的主程序继续向下执行代码，并且由go runtime自动管理

通信

如何在两个goroutine之间通信呢?

- 全局变量

```go
package main

import (
    "fmt"
    "time"
)

var shared int

func g1() {
    for i := 0; i < 10; i++ {
        fmt.Println("当前shared的值为", shared)
        time.Sleep(1 * time.Second)
    }
}

func g2() {
    for i := 0; i < 10; i++ {
        shared++
        time.Sleep(1 * time.Second)
    }
}

func main() {
    go g1()
    time.Sleep(20 * time.Millisecond)   // 等待0.02s执行下一语句，手动调控并发时间差
    go g2()

    time.Sleep(20 * time.Second)
}
```

- 通道通信
  - 通道是双工的，但是同时只能有一方发送，一方接收，也可看作队列，先进先出
  - 通道缓冲区满后，发送操作将被阻塞；无缓冲区的通道需要发送方和接收方**都就绪后**才能进行收发操作
  - 通道空后接收操作会被阻塞

使用`通道变量名 := make(chan 类型, 缓冲区)`来创建一个基本通道

- 类型：通道传输的数据类型
- 缓冲区：通道可容纳的数据数量最大值

```
// 创建无缓冲区通道
ch1 := make(chan int)

// 创建有缓冲通道
ch2 := make(chan int, 100)
```

使用`通道名 <- 对象`往通道内传递对象

```
ch1 <- 1
```

使用`<- 通道名`来接收从通道传递来的对象，该表达式可赋值给变量

```
...
// 使用
recv := <-ch1

// 或
var recv int
recv = <-ch1
```

关闭通道

```
close(chan)
```

预习作业：计时器

使用你已经学过的知识，尝试用Go编写一个计时器程序

当用户：

- 输入0时，重置计时器
- 输入1时，开始计时
- 输入2时，暂停或继续计时

如果暂时做不出来也没关系，分析一下可能的问题

```
package main

import (
    "fmt"
    "time"
)

// 定义全局变量
var ch chan int                // 用于goroutine间通信
var startTime time.Time        // 计时器开始时间
var elapsedTime time.Duration // 计时器已运行时间
var running bool               // 计时器是否在运行

// Timer 函数，接收通道中的命令并执行相应的操作
```

```go
func Timer(ch chan int) {
    for {
        switch <-ch {
        case 0: // 重置计时器
            startTime = time.Time{}
            elapsedTime = 0
            running = false
            fmt.Println("计时器已重置")
        case 1: // 开始计时
            if !running {
                startTime = time.Now()
                running = true
                fmt.Println("计时器已开始")
            } else {
                fmt.Println("计时器已在运行")
            }
        case 2: // 暂停/继续计时
            if running {
                elapsedTime += time.Since(startTime)
                running = false
                fmt.Println("计时器已暂停，已运行时间：", elapsedTime)
            } else {
                startTime = time.Now()
                running = true
                fmt.Println("计时器已继续")
            }
        }
    }
}

// Input 函数，接收用户输入的命令并发送到通道
func Input(ch chan int) {
    for {
        var cmd int
        fmt.Scan(&cmd)
        ch <- cmd
    }
}

func main() {
    fmt.Println("请输入命令：0-重置计时器，1-开始计时，2-暂停/继续计时")

    // 初始化通道
    ch = make(chan int)

    // 启动 Timer goroutine
    go Timer(ch)

    // 启动 Input goroutine
    go Input(ch)

    // 主 goroutine 等待
    select {}
}
```

上述代码中，启动了两个goroutine，一个负责与用户交互，一个负责计时，两者通过通道进行通信；通过计算时间差来实现计时。这样就可以确保在等待和处理用户输入时不会阻塞计时器。

扩展及案例

调度

上述示例中，我们在主函数中都用了`select{}`或`time.Sleep(...)`等函数来让`main()`保持存活，这是因为如果main goroutine退出后，它启动的其他goroutine都会被退出，大部分情况下都是在`main()`函数尾写一个`select{}`来等待.

在主goroutine存活的情况下，启动一个goroutine1，在goroutine1里面启动goroutine1_1，goroutine1_2，即使goroutine1结束了，goroutine1_1及1_2都会继续运行，这是因为goroutine是由go运行时调度而不是依赖它们的父goroutine

单通道监听：`for ... range`语句:

使用`for 通道 range`语句可以接收单个通道的内容

```
for data := range ioSource {
    HandlerFunc(data)
}
```

多路IO复用：`select`语句:

select语句与switch语句非常相似，是一种控制结构

- 其case只能用于通道io操作，即每个case接一个通道操作(要么读要么写)
- select语句会监听case所有通道是否就绪
- 若其中一个case就绪就执行对应代码块
- 若多个通道都就绪则随机选择一个执行
- 若没有通道就绪，则会执行default块中的代码，若没有则会阻塞等待

```
select {
  case <- chan1:
    // 仅获取接收状态，无需消息内容
  case value := <- chan2:
    // 将消息内容赋值给value
  case chan3 <- value:
    // 将value传递给chan3
    ...
  default:
    // 所有通道都没有准备好，执行的代码
}
```

```
// 完整示例
package main

import (
```

```
    "fmt"
    "time"
)

func main() {
    // 创建两个channel
    ch1 := make(chan string)
    ch2 := make(chan string)

    // 启动goroutine来发送数据
    go func() {
        time.Sleep(1 * time.Second) // 模拟耗时操作
        ch1 <- "数据来自ch1"        // 向ch1发送数据
    }()

    go func() {
        time.Sleep(2 * time.Second) // 模拟耗时操作
        ch2 <- "数据来自ch2"        // 向ch2发送数据
    }()

    // 使用select来等待两个channel
    for i := 0; i < 2; i++ {
        select {
        case msg1 := <-ch1:
            fmt.Println(msg1) // 处理来自ch1的数据
        case msg2 := <-ch2:
            fmt.Println(msg2) // 处理来自ch2的数据
        case <-time.After(3 * time.Second):
            fmt.Println("超时，没有数据到达")
            return
        }
    }
}
```

案例

> go的并发编程中有很多内容，这里我们选择秒杀**案例**进行讲解

并发中要考虑很多衍生的问题，例如多个goroutine对同一个资源的竞争性，我们用一个变量来储存剩余物品数，如何保证多个人同时抢一个商品时，操作的正确性?

- 使用channel来实现消息队列，生产者(顾客)创造的请求依次传递给消费者(处理函数)，处理完后才处理下一个
- 使用锁(mutex)来控制goroutine对变量的竞争，当一个goroutine还未释放锁时，另一个goroutine是的上锁操作会被阻塞，等待锁被释放

> 查看vscode源代码

在大多数生产环境中，数据库的事务可以帮我们实现这些功能，我们就不用手动去造轮子了，因为造轮子的都是**

总结

这节课总的来说就学了三个东西：`goroutine` `channel` `select/for`

以及它们的一些常用操作

- goroutine的创建及管理
- channel的管理及读写
- select: 通道的多路操作

作业

Lv1

使用两个goroutine**轮流**打印100以内的数

比如一个goroutine打印1，另一个打印2，(即一个打印奇数，一个打印偶数)，这样打印下去到100

Lv2

完善上述计时器，添加新功能：添加flag，在结束时输出每个flag的序号和时间(就比如多人跑1000m时裁判在终点每经过一个人就添加一个flag，可以同时记录很多人的时间)

Lv3

实现一个经典的生产者-消费者模型，生产者生成数据，消费者处理数据，使用通道进行通信。（如果觉得太抽象可以看下lv5的题目，往这个方向靠）

Lv4 可选的

在Lv4基础上封装一个商品秒杀功能，有能力可以扩展：web服务器，数据库，多个用户进程等