链滴

如何理解 channel?

作者：xhaoxiong

原文链接：https://ld246.com/article/1526452881104

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

- goroutine :是一个通过go关键字起起来的独立的执行某个function的过程，它拥有独立的可以自管理的调用栈。
- channels: 用于 goroutine 之间的通讯、同步

一个简单的事务处理的例子

对于下面这样的非并发的程序：

```
package main

import "fmt"

func main() {
    tasks := getTask()

    for _,task:=range tasks{

        process(task)
    }
}

func getTask() []int {

    t := []int{1, 2, 3, 4}

    return t
}

func process(task int)  {
    fmt.Printf("this is running %d task \n",task)
}
```

将其转换为 Go 的并发模式很容易，使用典型的 Task Queue 的模式：

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    tasks := getTasks()

    //创建带缓冲的channel
    ch := make(chan int, 3)

    //运行固定数量的workers
    wg := &sync.WaitGroup{}
    wg.Add(4)
    for i := 0; i < 4; i++ {
```

```
        go worker(ch, wg)
    }

    for _, task := range tasks {
        ch <- task

    }

    wg.Wait()

}

func getTasks() []int {

    t := []int{1, 2, 3, 4}

    return t
}

func worker(task chan int, wg *sync.WaitGroup) {

    for {
        process(task, wg)

    }
}

func process(task chan int, wg *sync.WaitGroup) {

    fmt.Printf("this is running %d task \n", <-task)
    wg.Done()
}
```

channels 的特性

- goroutine-safe，多个 goroutine 可以同时访问一个 channel 而不会出现竞争问题
- 可以用于在 goroutine 之间存储和传递值
- 其语义是先进先出（FIFO）
- 可以导致 goroutine 的 block 和 unblock

构造 channel

```
// 带缓冲的 channel
ch := make(chan Task, 3)
// 无缓冲的 channel
ch := make(chan Tass)
```

原文链接：如何理解 channel?

如果忽略内置的channel，让我们自己设计一个具有goroutines-safe 并且可以用来存储、传递值的西,该如何做？或许可以用一个带锁的队列来实现。channel内部就是一个带锁的队列

源码

```
type hchan struct {
    qcount   uint           // total data in the queue
    dataqsiz uint           // size of the circular queue[环形队列的大小]
    buf      unsafe.Pointer // points to an array of dataqsiz elements[// 指向一个环形队列]
    elemsize uint16
    closed   uint32
    elemtype *_type // element type
    sendx    uint   // send index[// 发送 index]
    recvx    uint   // receive index[// 接收 index]
    recvq    waitq  // list of recv waiters
    sendq    waitq  // list of send waiters

    // lock protects all fields in hchan, as well as several
    // fields in sudogs blocked on this channel.
    //
    // Do not change another G's status while holding this lock
    // (in particular, do not ready a G), as this can deadlock
    // with stack shrinking.
    lock mutex //  互斥量
}
```

buf具体实现就是一个环形队列的实现，sendx和recvx分别用来纪录发送、接收的位置然后用一个lock 斥锁来确保无竞争冒险。

对于每一个ch:=make(chan int,3)这类操作都会在堆中分配一个空间，简历并且初始化一个hchan str ut 而ch则是指向这个hchan struct的指针

因为ch本身是个指针，所以我们才可以在gotoutine函数调用的时候将ch传递过去，而不用再&ch取针了，所以所有使用同一个ch的goroutine都指向了同一个实际的内存空间

发送、接收

我们用 G1 描述main()函数的goroutine,G2表示worker的goroutine

```
// G1
func main() {
  ...
  for _, task := range tasks {
    ch <- task
  }
  ...
}
```

```
// G2
```

```
func worker(task  chan int) {
  for {
    process(task)
  }
}
```

简单的发送接收

G1中的ch<-task[0]具体是怎么是怎么做的呢?

1、获取锁

2、enqueue(task[0]) （这里是内存赋值task[0]）

3、释放锁

``` G2 ```中```fmt.Printf("this is running %d task \n", <-task)```是如何读取数据的

1、获取锁

2、dequeue()(<-task)（同样，这里也是内存复制）

3、释放锁

我们从这个操作中可以看到，所有 ```goroutine``` 中共享的部分只有这个```hchan```的结构体，而所通讯的数据都是内存复制。这遵循了 Go 并发设计中很核心的一个理念：```
  “Do not communicate by sharing memory;instead, share memory by communicating.” ```

阻塞和恢复

发送方阻塞

假设 ``` G2 ``` 需要很长时间的处理，在此期间，```G1``` 不断的发送任务：

- ```ch <- task1```

- ```ch <- task2```

- ```ch <- task3```
但是当再一次 ```ch <- task4``` 的时候，由于 ch 的缓冲只有 3 个，所以没有地方放了，于是 ```G1``` block 了，当有人从队列中取走一个 Task 的时候，```G1``` 才会被恢复。这是我们都知道的，不过我今天关心的不是发生了什么，而是如何做到的?

goroutine 的运行时调度

首先，goroutine 不是**操作系统线程**，而是**用户空间线程**。因此 goroutine 是由 Go runtime 来创建并管理的，而不是 OS，所以要比操作系统线程轻量级。

当然，goroutine 最终还是要运行于某个线程中的，控制 goroutine 如何运行于线程中的是 Go runtime 中的 scheduler （调度器）
```

Go 的运行时调度器是```M:N``` 调度模型，既 ```N```个 goroutine，会运行于 ```M```个 OS 线程中。句话说，一个 OS 线程中，可能会运行多个 goroutine。

Go 的 M:N 调度中使用了3个结构：

- ```M```: OS 线程
- ```G```: goroutine
- ```P```: 调度上下文

    - ```P``` 拥有一个运行队列，里面是所有可以运行的 ```goroutine``` 及其上下文

要想运行一个 goroutine - ```G```，那么一个线程 ```M```，就必须持有一个该 goroutine 的上下文 ```P```。

goroutine 被阻塞的具体过程

那么当 ```ch <- task4``` 执行的时候，```channel``` 中已经满了，需要**pause** ```G1```。这个时候：
1. ```G1``` 会调用运行时的 ```gopark```
2. 然后 ```Go``` 的运行时调度器就会接管
3. 将 ```G1``` 的状态设置为 ```waiting```
4. 断开 ```G1``` 和 ```M``` 之间的关系（switch out），因此 ```G1``` 脱离 ```M```，换句话说，```M``` 空了，可以安排别的任务了。
5. 从 ```P``` 的运行队列中，取得一个可运行的 ```goroutine G```
6. 建立新的 ```G``` 和 ```M``` 的关系（Switch in），因此 ```G``` 就准备好运行了。
7. 当调度器返回的时候，新的 ```G``` 就开始运行了，而 ```G1``` 则不会运行，也就是 ```block``` 了。

从上面的流程中可以看到，对于 goroutine 来说，```G1``` 被阻塞了，新的 G 开始运行了；而对于操系统线程``` M``` 来说，则根本没有被阻塞。

我们知道 OS 线程要比 goroutine 要沉重的多，因此这里尽量避免 OS 线程阻塞，可以提高性能。

###[插入相关知识]

与goroutine相关的调度逻辑:

- go(runtime.newproc)产生新的g，放到本地队列或全局队列
- gopark，g置为waiting状态，等待显示

- goready唤醒，在poller中用得较多
- goready，g置为runnable状态，放入全局队列
- gosched，g显示调用runtime.Gosched或被抢占，置为runnable状态，放入全局队列
- goexit，g执行完退出，g所属m切换到g0栈，重新进入schedule
- g陷入syscall:

    - net io和部分file io，没有事件则gopark;

    - 普通的阻塞系统调用，返回时m重新进入schedule

- g陷入cgocall: lockedm加上syscall的处理逻辑
- g执行超过10ms被sysmon抢占

#引用自知乎不一样的天空

goroutine 恢复执行的具体过程

前面理解了阻塞，那么接下来理解一下如何恢复运行。不过，在继续了解如何恢复之前，我们需要先一步理解
```hchan``` 这个结构。因为，当 ```channel``` 不在满的时候，调度器是如何知道该让哪个 ```goroutin``` 继续运行呢？而且 ```goroutine``` 又是如何知道该从哪取数据呢？

在 ```hchan``` 中，除了之前提到的内容外，还定义有 ```sendq``` 和 ```recvq``` 两个队列，分别表示待发送、接收的 ```goroutine```，及其相关信息。

```
type hchan struct {
...
buf      unsafe.Pointer // 指向一个环形队列
...
sendq    waitq  // 等待发送的队列
recvq    waitq  // 等待接收的队列
...
lock     mutex  //  互斥量
}
```

其中 waitq 是一个链表结构的队列，每个元素是一个 sudog 的结构，其定义大致为：

waitq

```
type waitq struct {
first *sudog
last  *sudog
}
```

sudog struct

```
// sudog represents a g in a wait list, such as for sending/receiving
// on a channel.
//
// sudog is necessary because the g □□left_right_arrow
synchronization object relation
// is many-to-many. A g can be on many wait lists, so there may be
// many sudogs for one g; and many gs may be waiting on the same
// synchronization object, so there may be many sudogs for one object.
//
// sudogs are allocated from a special pool. Use acquireSudog and
// releaseSudog to allocate and free them.
type sudog struct {
```
```

// The following fields are protected by the hchan.lock of the
// channel this sudog is blocking on. shrinkstack depends on
// this.

g   *g [// 正在等候的 goroutine]
selectdone *uint32 // CAS to 1 to win select race (may point to stack)
next   *sudog
prev   *sudog
elem   unsafe.Pointer // data element (may point to stack)[// 指向需要接收、发送的元素]

// The following fields are never accessed concurrently.
// waitlink is only accessed by g.

acquiretime int64
releasetime int64
ticket   uint32
waitlink   *sudog // g.waiting list
c   *hchan // channel

}

源码

所以在之前的阻塞 G1 的过程中，实际上：

1. ```G1``` 会给自己创建一个 ```sudog``` 的变量
2. 然后追加到 ```sendq``` 的等候队列中，方便将来的 ```receiver``` 来使用这些信息恢复 ```G1```。

这些都是发生在***调用调度器之前***

恢复过程如下：

当 ```G2``` 调用 ```fmt.Printf("this is running %d task \n", <-task)``` 的时候，```channel``` 的状态，缓冲是满的，而且还有一个 ```G1``` 在等候发送队列里，然后 ```G2``` 执行下面的操作：

1. ```G2``` 先执行 ```dequeue()``` 从缓冲队列中取得 ```task1``` 给 ```t```
2. ```G2``` 从 ```sendq``` 中弹出一个等候发送的 ```sudog```
将弹出的 ```sudog``` 中的 ```elem``` 的值 ```enqueue()``` 到 ```buf``` 中
3. 将弹出的 ```sudog``` 中的 ```goroutine```，也就是 ```G1```，状态从 ```waiting``` 改为 ```runnable```
  1. 然后，```G2``` 需要通知调度器 ```G1``` 已经可以进行调度了，因此调用 ```goready(G1)```。
  2. 调度器将 ```G1``` 的状态改为 ```runnable```
  3. 调度器将 ```G1``` 压入 ```P ```的运行队列，因此在将来的某个时刻调度的时候，```G1``` 就会开始复运行。
  4. 返回到 ```G2```

++注意，这里是由 ```G2``` 来负责将 ```G1``` 的 ```elem``` 压入 ```buf``` 的，这是一个优化。这样将来 ```G1``` 恢复运行后，就不必再次获取锁、enqueue()、释放锁了。这样就避免了多次锁的开销++

如果接收方先阻塞呢?
更酷的地方是接收方先阻塞的流程。

1. 如果 ```G2``` 先执行了 ```fmt.Printf("this is running %d task \n", <-task)```，此时``` buf``` 是空，因此 ```G2``` 会被阻塞，他的流程是这样：

2. ```G2 ```给自己创建一个 ```sudog``` 结构变量。其中 ```g``` 是自己，也就是``` G2```，而``` elem``` 指向 ```t```
将这个 ```sudog``` 变量压入 ```recvq``` 等候接收队列
3. ```G2``` 需要告诉 ```goroutine```，自己需要``` pause``` 了，于是调用 ```gopark(G2)```
   1. 和之前一样，调度器将其 G2 的状态改为 ```waiting```
   2. 断开 ```G2``` 和``` M``` 的关系
   3. 从 ```P``` 的运行队列中取出一个 ```goroutine```
   4. 建立新的 ```goroutine``` 和``` M``` 的关系
   5. 返回，开始继续运行新的 ```goroutine```
这些应该已经不陌生了，那么当 ```G1``` 开始发送数据的时候，流程是什么样子的呢?

```G1``` 可以将```enqueue(task)```，然后调用 ```goready(G2)```。不过，我们可以更聪明一些。

我们根据 ```hchan``` 结构的状态，已经知道 ```task``` 进入 ```buf``` 后，```G2``` 恢复运行后，会读取值，复制到 ```t``` 中。那么 ```G1``` 可以根本不走 ```buf```，```G1``` 可以直接把数据给 ```G2```。

```Goroutine``` 通常都有自己的栈，互相之间不会访问对方的栈内数据，除了 ```channel```。这里，于我们已经知道了 ```t``` 的地址（通过 ```elem```指针），而且由于 ```G2``` 不在运行，所以我们可以安全的直接赋值。当 ```G2``` 恢复运行的时候，既不需要再次获取锁，也不需要对 ```buf``` 进行操作从而节约了内存复制、以及锁操作的开销。

无缓冲 channel
无缓冲的``` channel``` 行为就和前面说的直接发送的例子一样:

- 接收方阻塞 → 发送方直接写入接收方的栈
- 发送方阻塞 → 接受法直接从发送方的 ```sudog``` 中读取

select

源码

1. 先把所有需要操作的 ```channel``` 上锁
2. 给自己创建一个 ```sudog```，然后添加到所有 channel 的 ```sendq```或```recvq``` （取决于是发送是接收）
3. 把所有的 ```channel``` 解锁，然后 ```pause``` 当前调用 ```select ```的 ```goroutine（gopark()）```
4. 然后当有任意一个 ```channel``` 可用时，```select ```的这个 ```goroutine``` 就会被调度执行。
5. resuming mirrors the pause sequence

为什么 Go 会这样设计?

Simplicity

更倾向于带锁的队列，而不是无锁的实现。

“性能提升不是凭空而来的，是随着复杂度增加而增加的。” - dvyokov

后者虽然性能可能会更好，但是这个优势，并不一定能够战胜随之而来的实现代码的复杂度所带来的势。

Performance

- 调用 ```Go``` 运行时调度器，这样可以保持 ```OS``` 线程不被阻塞
跨 ```goroutine``` 的栈读、写。

- 可以让``` goroutine``` 醒来后不必获取锁
- 可以避免一些内存复制

当然，**任何优势都会有其代价**。这里的代价是实现的复杂度，所以这里有更复杂的内存管理机制
垃圾回收以及栈收缩机制。

在这里性能的提高优势，要比复杂度的提高带来的劣势要大。

所以在 ```channel``` 实现的各种代码中，我们都可以见到这种 **simplicity vs performance** 的权
后的结果。

Gopher2017视频(需翻墙)

<a href="https://github.com/gophercon/2017-talks/blob/master/KavyaJoshi-Understanding
hannels/Kavya%20Joshi%20-%20Understanding%20Channels.pdf">幻灯片

<a href="https://blog.lab99.org/post/golang-2017-10-04-video-understanding-channels.htm
#go-de-bing-fa-te-xing">参考博文

在demo中用到了```sync.WaitGroup```这个包

先说说```WaitGroup```的用途：它能够一直等到所有的```goroutine```执行完成，并且阻塞主线程的
行，直到所有的```goroutine```执行完成。

WaitGroup总共有三个方法：```Add(delta int)```,```Done()```,```Wait()```。简单的说一下这三个方法
作用。

1. ```Add```:添加或者减少等待goroutine的数量

2. ```Done```:相当于Add(-1)

3. ```Wait```:执行阻塞，直到所有的```WaitGroup```数量变成0

WaitGroup的功能：它实现了一个类似队列的结构，可以一直向队列中添加任务，当任务完成后便从
列中删除，如果队列中的任务没有完全完成，可以通过Wait()函数来出发阻塞，防止程序继续进行，
到所有的队列任务都完成为止.

WaitGroup的特点是Wait()可以用来阻塞直到队列中的所有任务都完成时才解除阻塞，而不需要slee
一个固定的时间来等待

源码