



分布式计算实验报告

题目：实验课作业
(二)

姓 名	张瑞程
学 号	22354189
院 系	智能工程学院
专 业	智能科学与技术
指导教师	余成韵

2024 年 12 月

问题一：多生产者多消费者模式（20 分）

描述：实现一个多生产者和多消费者的程序，模拟数据流处理。

说明：分别创建超过一个 goroutine 作为生产者和消费者（生产者数量 ≥ 2 ，消费者数量 ≥ 2 ）。生产者不断将随机数发送到一个通道，而消费者从通道读取并输出该随机数。

要求：使用 channel 实现生产者和消费者的交互。生产或消费数据时，同时输出对应的生产者或消费者编号。

示例：`fmt.Printf("Producer %d produced %d\n", id, num)`

1.1 问题分析

本问题为了实现了一个多生产者、多消费者的并发数据流处理程序，关键目标是通过 channel 实现生产者与消费者之间的同步通信，同时使用 `sync.WaitGroup` 保证所有并发任务完成后程序能够正确退出。

1.2 代码设计

（1）生产者与消费者的并发管理

- 生产者通过随机数模拟数据生产，向通道中写入数据。代码段如下：

```
1. for i := 0; i < numItems; i++ {  
2.     num := rand.Intn(100)  
3.     ch <- num // 向通道发送数据  
4.     fmt.Printf("Producer %d produced %d\n", id, num)  
5.     time.Sleep(time.Millisecond * 500) // 模拟生产过程的延迟  
6. }
```

每个生产者向通道发送固定数量的随机数，配合延迟模拟实际生产的时间间隔。

- 消费者从通道中读取数据并处理，直到通道关闭。代码实现如下：

```
1. for num := range ch { // 从通道中读取数据，直到通道关闭  
2.     fmt.Printf("Consumer %d consumed %d\n", id, num)  
3.     time.Sleep(time.Millisecond * 500) // 模拟消费过程的延迟  
4. }
```

采用 `range` 语法遍历通道，优雅地处理通道关闭后自动退出的场景。

(2) 通道的使用与缓冲设置

- 通道被设计为带缓冲区，大小为 10:

```
1. ch := make(chan int, 10) // 带缓冲的通道, 缓冲大小为 10
```

设置缓冲区可以减少生产者与消费者的阻塞次数，从而提升性能，尤其在高并发情况下尤为重要。

(3) 生产者和消费者同步机制

- 使用两个 `sync.WaitGroup` 分别管理生产者和消费者的并发任务:

```
1. var producerWG sync.WaitGroup
2. var consumerWG sync.WaitGroup
```

其工作思路为:

- 在启动每个生产者和消费者时，增加计数 (Add)，在其完成后减少计数 (Done)。
- 通过 `Wait()` 保证主程序在所有任务完成后再退出。

(4) 关闭通道的逻辑

- 当所有生产者任务完成后，通道会被关闭，通知消费者不再有新数据:

```
1. go func() {
2.     producerWG.Wait() // 等待生产者完成
3.     close(ch)         // 关闭通道
4. }()
```

这里使用一个专门的协程完成此逻辑，避免通道关闭与消费者并发读取产生冲突。

1.3 运行与结果

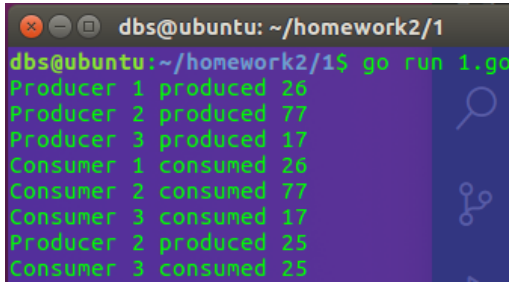
运行代码: `go run 1.go`。

程序首先进行初始化，创建生产者、消费者各三个，每个生产者生产的随机数个数 `numItems=10`。

```
1. const (
2.     numProducers = 3 // 生产者数量
3.     numConsumers = 3 // 消费者数量
4.     numItems     = 10 // 每个生产者生产的随机数个数
5. )
```

每个生产者会向通道发送 `numItems` 个随机数，每次生产时格式化打印输出;

每个消费者从通道中读取数据，格式化打印消费信息。最后，生产者完成后，关闭通道；消费者检测到通道关闭后自动退出。



输出结果展示（部分）

1.4 完整代码

```
1 package main
2
3 import (
4     "fmt"
5     "math/rand"
6     "sync"
7     "time"
8 )
9
10 const (
11     numProducers = 3 // 生产者数量
12     numConsumers = 3 // 消费者数量
13     numItems      = 10 // 每个生产者生产的随机数个数
14 )
15
16 func producer(id int, ch chan<- int, wg *sync.WaitGroup) {
17     defer wg.Done()
18     for i := 0; i < numItems; i++ {
19         num := rand.Intn(100)
20         ch <- num // 向通道发送数据
21         fmt.Printf("Producer %d produced %d\n", id, num)
22         time.Sleep(time.Millisecond * 500) // 模拟生产过程的延迟
23     }
24 }
25
```

```
26 func consumer(id int, ch <-chan int, wg *sync.WaitGroup) {
27     defer wg.Done()
28     for num := range ch { // 从通道中读取数据, 直到通道关闭
29         fmt.Printf("Consumer %d consumed %d\n", id, num)
30         time.Sleep(time.Millisecond * 500) // 模拟消费过程的延迟
31     }
32 }
33
34 func main() {
35     rand.Seed(time.Now().UnixNano()) // 初始化随机数种子
36     ch := make(chan int, 10)         // 带缓冲的通道, 缓冲大小为10
37
38     var producerWG sync.WaitGroup
39     var consumerWG sync.WaitGroup
40
41     // 启动生产者
42     for i := 1; i <= numProducers; i++ {
43         producerWG.Add(1)
44         go producer(i, ch, &producerWG)
45     }
46
47     // 启动消费者
48     for i := 1; i <= numConsumers; i++ {
49         consumerWG.Add(1)
50         go consumer(i, ch, &consumerWG)
51     }
52
53     // 启动一个协程等待所有生产者完成, 并关闭通道
54     go func() {
55         producerWG.Wait() // 等待生产者完成
56         close(ch)         // 关闭通道, 通知消费者不再有新数据
57     }()
58
59     consumerWG.Wait() // 等待所有消费者完成
60 }
```

问题二：多路复用数据收集（20 分）

描述：从多个数据源中收集数据并合并结果。

说明：启动超过一个 goroutine 作为数据源，每个数据源定期发送数据到 channel，使用 select 同时监听多个数据源的输入并汇总结果。

要求：可以使用多个通道接收不同数据源的输出，并使用一个聚合通道收集最终结果。

2.1 问题分析

本问题可以理解为一个主线程接受多个分线程的信息的过程，如果有三个数据源：数据源 1：[10, 20, 30]，数据源 2：[40, 50]，数据源 3：[60, 70, 80]。那么

聚合通道收集到的结果将为:

Aggregated data: 10

Aggregated data: 40

Aggregated data: 60

Aggregated data: 20

Aggregated data: 50

Aggregated data: 30

Aggregated data: 70

Aggregated data: 80

本问题可以分解为如下三个子问题:

1. 创建多个独立的数据源, 每个数据源独立运行, 并定期产生数据。
2. 数据通过通道 `channel` 传递到聚合逻辑。
3. 使用 `select` 同时监听多个通道, 最终输出收集到的结果。

对应三个子问题, 我们可以分别给出设计思路:

1. **数据源部分:** 每个数据源运行在一个 `goroutine` 中 (独立运行), 定期向对应通道发送随机数据。
2. **聚合器部分:** 另启动一个 `goroutine` 使用 `select` 动态监听多个数据源的通道, 将结果合并到聚合通道。
3. **结果处理部分:** 从聚合通道收集数据并输出。

2.2 代码设计

1. **数据源的实现** 每个数据源通过 `generateData` 函数产生数据并发送到通道, 核心代码如下:

```
1. func generateData(sourceID int, ch chan<- int, wg *sync.WaitGroup) {  
2.     defer wg.Done()  
3.     for i := 0; i < dataCount; i++ {  
4.         data := rand.Intn(100) // 生成随机数据  
5.         ch <- data  
6.         fmt.Printf("DataSource %d produced %d\n", sourceID, data)  
7.         time.Sleep(time.Millisecond * 500) // 模拟延迟  
8.     }
```

```
9.     close(ch) // 数据源完成后关闭通道
10. }
```

其工作流程如下：

- i. 每个数据源固定产生 `dataCount` 个随机数。
- ii. 使用 `time.Sleep` 模拟数据产生的延迟。
- iii. 生产完成后通过 `close(ch)` 关闭通道，通知聚合器。

2. **聚合器的实现** 聚合器使用 `select` 同时监听所有数据源的通道，并将数据转发到聚合通道：

```
1. go func() {
2.     for {
3.         activeSources := 0
4.         for _, ch := range channels {
5.             select {
6.                 case data, ok := <-ch:
7.                     if ok {
8.                         aggregateChannel <- data
9.                         activeSources++
10.                    }
11.                }
12.            }
13.            if activeSources == 0 {
14.                break // 所有通道关闭后退出
15.            }
16.        }
17.        close(aggregateChannel) // 关闭聚合通道
18.    }()
```

其工作流程如下：

- i. 遍历所有数据源通道，使用 `select` 动态收集数据。
- ii. 当所有数据源通道都关闭时，退出循环并关闭聚合通道。

3. 结果处理的实现 从聚合通道读取最终合并的数据并输出：

```
1. go func() {  
2.     for data := range aggregateChannel {  
3.         fmt.Printf("Aggregated result: %d\n", data)  
4.     }  
5. }()
```

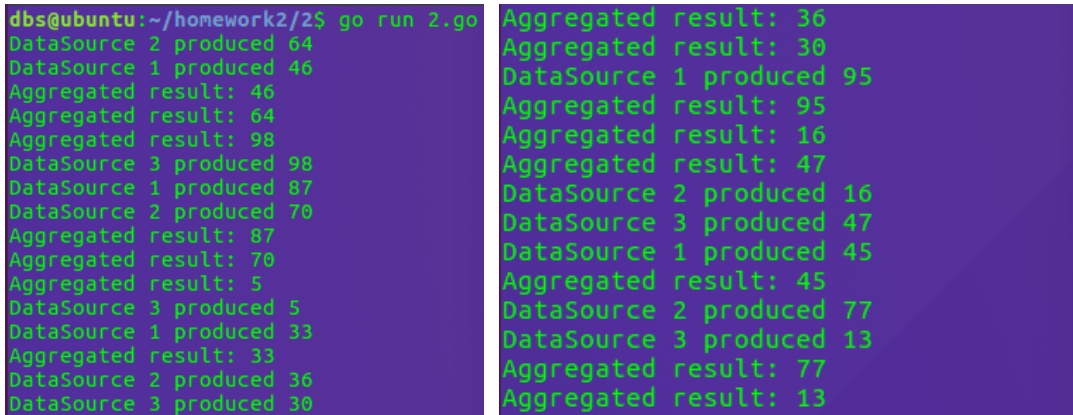
其工作流程如下：

- i. 通过 `range` 遍历聚合通道，直到通道关闭。
- ii. 将结果格式化输出。

4. 通道的关闭逻辑 主程序通过 `sync.WaitGroup` 等待所有数据源完成，并在聚合逻辑中保证关闭通道的顺序：

```
1. wg.Wait() // 等待所有数据源完成  
2. time.Sleep(time.Second * 1) // 确保数据处理完成后主程序退出
```

2.3 运行与结果



```
db@ubuntu:~/homework2/2$ go run 2.go  
DataSource 2 produced 64  
DataSource 1 produced 46  
Aggregated result: 46  
Aggregated result: 64  
Aggregated result: 98  
DataSource 3 produced 98  
DataSource 1 produced 87  
DataSource 2 produced 70  
Aggregated result: 87  
Aggregated result: 70  
Aggregated result: 5  
DataSource 3 produced 5  
DataSource 1 produced 33  
Aggregated result: 33  
DataSource 2 produced 36  
DataSource 3 produced 30  
Aggregated result: 36  
Aggregated result: 30  
DataSource 1 produced 95  
Aggregated result: 95  
Aggregated result: 16  
Aggregated result: 47  
DataSource 2 produced 16  
DataSource 3 produced 47  
DataSource 1 produced 45  
Aggregated result: 45  
DataSource 2 produced 77  
DataSource 3 produced 13  
Aggregated result: 77  
Aggregated result: 13
```

运行 `go run 2.go`，数据源按顺序产生数据并通过聚合器输出到终端，所有数据源的结果成功合并到聚合通道并输出。代码采用了异步输出的方式，聚合到的数据集随机就被输出，这样的设计减小了缓存所有数据所带来的不必要额外开销。

2.4 完整代码



```

1 package main
2
3 import (
4     "fmt"
5     "math/rand"
6     "sync"
7     "time"
8 )
9
10 const (
11     numDataSources = 3 // 数据源数量
12     dataCount      = 5 // 每个数据源发送的数据个数
13 )
14
15 func generateData(sourceID int, ch chan<- int, wg *sync.WaitGroup) {
16     defer wg.Done()
17     for i := 0; i < dataCount; i++ {
18         data := rand.Intn(100) // 随机数据
19         ch <- data
20         fmt.Printf("DataSource %d produced %d\n", sourceID,
21 data)
22         time.Sleep(time.Millisecond * 500) // 模拟数据源产生数据的
23         时间
24     }
25     close(ch) // 数据源完成后关闭自己的通道
26 }
27
28 func main() {
29     rand.Seed(time.Now().UnixNano()) // 初始化随机数种子
30
31     var wg sync.WaitGroup
32     channels := make([]chan int, numDataSources) // 用于存放每个数据源
33     的通道
34     aggregateChannel := make(chan int)           // 聚合通道，用于收集
35     所有数据源的结果
36
37     // 启动多个数据源 (goroutines)
38     for i := 0; i < numDataSources; i++ {
39         channels[i] = make(chan int)
40         wg.Add(1)
41         go generateData(i+1, channels[i], &wg)
42     }
43
44     // 启动一个协程来聚合数据
45     go func() {
46         // 使用select从多个通道收集数据
47         for {
48             activeSources := 0
49             for _, ch := range channels {
50                 select {
51                     case data, ok := <-ch:
52                         if ok {
53                             aggregateChannel <-

```

```

data
50                                     activeSources++
51                                     }
52                                     }
53                                     }
54                                     if activeSources == 0 {
55                                         break // 如果所有通道都关闭, 则退出循环
56                                     }
57                                 }
58                                 close(aggregateChannel) // 所有数据处理完成后关闭聚合通道
59                             }()
60
61                             // 从聚合通道收集最终结果
62                             go func() {
63                                 for data := range aggregateChannel {
64                                     fmt.Printf("Aggregated data: %d\n", data)
65                                 }
66                             }()
67
68                             wg.Wait() // 等待所有数据源的生产者完成
69                             time.Sleep(time.Second * 1) // 确保所有数据处理完成后主程序退出
70 }

```

问题三：计数器实现 (20 分)

描述：实现一个安全的并发计数器。

说明：多个 goroutine 可以同时读取计数器值，但只有一个 goroutine 能修改计数器。

要求：代码中分别启用超过一个并发读取器和并发写入器，涉及 sync.Mutex 和 sync.RWMutex 的使用。

提示：计数器可以定义为如下结构，并实现它的取值函数和增长函数。

```

type Counter struct {
    sync.RWMutex
    count int
}

```

3.1 问题分析

本问题要求实现一个支持多读单写的并发访问模式的计数器。

首先，需要定义一个计数器结构（题目中提示），其中包含一个 sync.RWMutex 作为锁和一个 int 类型的 count 变量用于存储计数器的值。

然后，要保证读和写操作的安全性，可以通过两个函数分别实现。对于读操作，使用 RLock 来获取读锁，以允许多个读操作同时进行，确保读取操作的安全性。在读取完计数器值后，通过 defer 释放读锁。对于写操作，使用 Lock

获取写锁，以确保在增加计数器值时没有其他读或写操作。增加完毕后，释放写锁。

在有了正确的读和写操作函数后，我们就可以创建读取器和写入器了。**读取器**使用多个 `goroutine` 调用读函数，该函数循环读取计数器值（模拟并发），并输出当前的计数器值。**写入器**使用多个 `goroutine` 将调用写函数，该函数每隔一段时间就循环增加计数器值，并输出写入信息。这样，既满足了“代码中分别启用超过一个并发读取器和并发写入器”的要求。

最后，通过 `sync.WaitGroup` 等待所有 `goroutine` 完成。

3.2 代码设计

我的代码完全按照上面的思路设计，下面为详细解析：

(1) 计数器：Counter 结构体

```
1. type Counter struct { sync.RWMutex; count int }
```

它包含一个读写锁 `RWMutex` 和一个整数 `count` 来保存计数器的值。

(2) 读操作函数：GetValue 方法

```
1. func (c *Counter) GetValue() int {  
2.     c.RLock() // 使用读锁  
3.     defer c.RUnlock()  
4.     return c.count  
5. }
```

在该方法中，使用 `c.RLock()` 获取读锁，确保在读取计数器值时，其他读取操作可以并发进行。通过 `defer c.RUnlock()` 确保在方法结束时释放读锁，从而避免死锁。

(3) 写操作函数：Increment 方法

```
1. func (c *Counter) Increment() {  
2.     c.Lock() // 使用写锁  
3.     defer c.Unlock()  
4.     c.count++  
5. }
```

在该方法中，使用 `c.Lock()` 获取写锁，以确保在增加计数器值时没有其他读或写操作。通过 `defer c.Unlock()` 在增加操作完成后释放写锁。

(4) 读取器：reader

```
1. func reader(id int, counter *Counter, wg *sync.WaitGroup) {  
2.     defer wg.Done()
```

```
3.     for i := 0; i < 5; i++ {
4.         value := counter.GetValue()
5.         fmt.Printf("Reader %d: Counter value = %d\n", id, value)
6.         time.Sleep(time.Millisecond * 200)
7.     }
8. }
```

reader 循环调用 GetValue 获取计数器值并输出，使用 wg.Done() 在任务完成后减少 WaitGroup 计数。time.Sleep 用于模拟读取过程的延迟。

(5) 写入器: writer

```
1. func writer(id int, counter *Counter, wg *sync.WaitGroup) {
2.     defer wg.Done()
3.     for i := 0; i < 3; i++ {
4.         counter.Increment()
5.         fmt.Printf("Writer %d: Incremented counter\n", id)
6.         time.Sleep(time.Millisecond * 500)
7.     }
8. }
```

writer 循环调用 Increment 增加计数器值并输出，使用 wg.Done() 在任务完成后减少 WaitGroup 计数，同样的，time.Sleep 用于模拟写入过程的延迟。

(6) main 函数

在主函数中启动三个读取器和一个写入器，并为每个读取任务添加到 WaitGroup 中。在所有 goroutine 完成后，程序结束。

```
1. for i := 1; i <= 3; i++ {
2.     wg.Add(1)
3.     go reader(i, &counter, &wg)
4. }
```

启动三个并发读取 goroutine 并添加到 WaitGroup 中

```
1. for i := 1; i <= 3; i++ {
2.     wg.Add(1)
3.     go writer(i, &counter, &wg)
4. }
```

启动三个写入 goroutine 并添加到 WaitGroup 中。

```
1. wg.Wait()
```

等到所有 goroutine 完成后，程序结束。

3.3 运行与结果

运行 go run 2.go, reader 和 writer 分别被操作并开始执行。输出结果如下：



```
dbz@ubuntu:~/homework2/3$ go run 3.go
Reader 2: Counter value = 0
Writer 3: Incremented counter
Writer 2: Incremented counter
Writer 1: Incremented counter
Reader 3: Counter value = 3
Reader 1: Counter value = 3
Reader 3: Counter value = 3
Reader 1: Counter value = 3
Reader 2: Counter value = 3
Reader 1: Counter value = 3
Reader 3: Counter value = 3
Reader 2: Counter value = 3
Writer 1: Incremented counter
Writer 2: Incremented counter
Writer 3: Incremented counter
Reader 2: Counter value = 6
Reader 3: Counter value = 6
Reader 1: Counter value = 6
Reader 2: Counter value = 6
Reader 1: Counter value = 6
Reader 3: Counter value = 6
```

3.4 完整代码

```
1 package main
2
3 import (
4     "fmt"
5     "sync"
6     "time"
7 )
8
9 // 计数器可以定义为如下结构，并实现它的取值函数和增长函数
10 type Counter struct {
11     sync.RWMutex
12     count int
13 }
14
15 // 获取计数器值
16 func (c *Counter) GetValue() int {
17     c.RLock() // 使用读锁
18     defer c.RUnlock()
19     return c.count
20 }
21
22 // 增加计数器值
23 func (c *Counter) Increment() {
24     c.Lock() // 使用写锁
25     defer c.Unlock()
26     c.count++
27 }
```

```
29 func reader(id int, counter *Counter, wg *sync.WaitGroup) {
30     defer wg.Done()
31     for i := 0; i < 5; i++ {
32         value := counter.GetValue()
33         fmt.Printf("Reader %d: Counter value = %d\n", id, value)
34         time.Sleep(time.Millisecond * 200)
35     }
36 }
37
38 func writer(id int, counter *Counter, wg *sync.WaitGroup) {
39     defer wg.Done()
40     for i := 0; i < 3; i++ {
41         counter.Increment()
42         fmt.Printf("Writer %d: Incremented counter\n", id)
43         time.Sleep(time.Millisecond * 500)
44     }
45 }
46
47 func main() {
48     var counter Counter
49     var wg sync.WaitGroup
50
51     // 启动多个读取任务
52     for i := 1; i <= 3; i++ {
53         wg.Add(1)
54         go reader(i, &counter, &wg)
55     }
56
57     // 启动3个写入任务
58     for i := 1; i <= 3; i++ {
59         wg.Add(1)
60         go writer(i, &counter, &wg)
61     }
62
63
64     // 等待所有任务完成
65     wg.Wait()
66 }
```

问题四：并发计算的 MapReduce 模型（20 分）

描述：使用并发方式实现一个简单的 MapReduce。

说明：设计一个函数处理字符串数组，将每个字符串映射为其长度，然后使用 goroutine 并发计算每个长度，并将结果汇总。

要求：使用 sync.WaitGroup 控制任务完成，并用 channel 收集结果。

示例：words := []string{"Where", "did", "I", "put", "my", "lighter"}

4.1 问题分析

本问题需要通过 map 和 reduce 两个阶段完成字符串的长度和计算。

Map 阶段并发地将字符串映射为其长度，通过 results 通道传递；**Reduce 阶段**从通道中读取长度并计算总和。

这两个阶段通过 `sync.WaitGroup` 协调，确保同步执行，最终返回总长度。

4.2 代码设计

Map 阶段

```
1. func mapPhase(words []string, results chan<- int, wg *sync.WaitGroup) {
2.     defer wg.Done()
3.     for _, word := range words {
4.         results <- len(word)
5.     }
6. }
```

`mapPhase` 以字符串数组 `words` 为输入。遍历 `words`，计算每个字符串的长度，将计算结果发送到 `results` 通道。

Reduce 阶段

```
1. func reducePhase(results <-chan int, wg *sync.WaitGroup) int {
2.     total := 0
3.     for length := range results {
4.         total += length
5.     }
6.     wg.Done()
7.     return total
8. }
```

`reducePhase` 以通道 `results` 中的长度数据为输入，将所用单词的长度进行累加，返回累加后的结果。并在处理完成后，调用 `wg.Done()` 通知任务完成。

mapReduce 函数

```
1. func mapReduce(words []string) int {
2.     results := make(chan int, len(words))
3.     var mapWg sync.WaitGroup
4.     var reduceWg sync.WaitGroup
5.
6.     // Map 阶段
7.     mapWg.Add(1)
8.     go mapPhase(words, results, &mapWg)
9.
10.    // Reduce 阶段
11.    reduceWg.Add(1)
12.    var total int
13.    go func() {
14.        total = reducePhase(results, &reduceWg)
15.    }()
```

```
16.  
17.     mapWg.Wait() // 等待 Map 阶段完成  
18.     close(results) // 关闭通道  
19.     reduceWg.Wait() // 等待 Reduce 阶段完成  
20.     return total  
21. }
```

mapReduce 函数用于执行 map 和 reduce 两个阶段的逻辑流，保证两任务的衔接和同步，其运行步骤具体如下：

1. 启动 Map 阶段，计算字符串长度并写入通道。
2. 启动 Reduce 阶段，从通道中读取数据并累加。
3. 等待 Map 阶段完成后，关闭通道。
4. 等待 Reduce 阶段完成后，返回总长度。

其中，保持 map 和 reduce 的工作协调一致是十分重要的，results 通道用于在 Map 和 Reduce 阶段传递数据，而 sync.WaitGroup 确保任务完成同步。

4.3 运行与结果

在主函数中设定一组字符串，输入 mapReduce 函数，totalLength 接受总长度的结果。

```
1. words := []string{"Where", "did", "I", "put", "my", "lighter"}  
2. totalLength := mapReduce(words)  
3. fmt.Printf("Total length of all words: %d\n", totalLength)
```

运行 go run 4.go，终端输出如下：

```
dbx@ubuntu:~/homework2/4$ go run 4.go  
Total length: 21
```

4.4 完整代码



```
1 package main
2
3 import (
4     "fmt"
5     "sync"
6 )
7
8 // Map 阶段: 将字符串映射为其长度
9 func mapPhase(words []string, results chan<- int, wg *sync.WaitGroup) {
10     defer wg.Done()
11     for _, word := range words {
12         results <- len(word)
13     }
14 }
15
16 // Reduce 阶段: 汇总所有长度
17 func reducePhase(results chan int, wg *sync.WaitGroup) int {
18     total := 0
19     for length := range results {
20         total += length
21     }
22     wg.Done()
23     return total
24 }
25
26 func mapReduce(words []string) int {
27     // 创建通道和 WaitGroup
28     results := make(chan int, len(words))
29     var mapWg sync.WaitGroup
30     var reduceWg sync.WaitGroup
31
32     // Map 阶段
33     mapWg.Add(1)
34     go mapPhase(words, results, &mapWg)
35
36     // Reduce 阶段
37     reduceWg.Add(1)
38     var total int
39     go func() {
40         total = reducePhase(results, &reduceWg)
41     }()
42
43     // 等待 Map 阶段完成并关闭通道
44     mapWg.Wait()
45     close(results)
46
47     // 等待 Reduce 阶段完成
48     reduceWg.Wait()
49     return total
50 }
51
52 func main() {
53     words := []string{"Where", "did", "I", "put", "my", "lighter"}
54     totalLength := mapReduce(words)
55     fmt.Printf("Total length: %d\n", totalLength)
56 }
```

问题五：超时与取消控制的多任务执行（20 分）

描述：设计一个支持超时和取消的多任务执行系统。

说明：启动若干 `goroutine` 作为任务，每个任务随机生成独立的执行时间。
要求在给定时间内完成全部任务，若超时则取消所有任务。

提示：可以使用 `context` 包提供的取消功能，实现对任务的超时控制。

参考资料：<https://zhuanlan.zhihu.com/p/626489437>

5.1 问题分析

在本问题中，我们首先要创建多个任务，每个任务都有独立的随机执行时间。然后需要时时监听总执行时间，确保其不能超过设定的超时时间（10 秒），否则需要取消所有已完成和未完成的任务。

具体解决思路如下：

1. 随机任务执行时间：

使用 `rand.Intn` 随机生成任务的执行时间（1 到 5 秒），为每个任务设置执行时间，并启动对应的 `goroutine`。

2. 总时间监控：

使用一个变量 `totalTime` 来累加所有任务的执行时间。如果任务的总执行时间超过设定的超时时间，调用 `context` 的取消方法终止所有未完成的任务。

3. 并发管理：

使用 `sync.WaitGroup` 确保所有任务被正确等待。使用 `sync.Mutex` 来保护对 `totalTime` 的并发访问，避免数据竞争。

4. 任务终止机制：

每个任务通过 `context.Context` 检测取消信号，一旦取消即提前终止任务。

5.2 代码设计

（1）任务函数（task）

`Task` 函数为了模拟一个任务，计算其运行时间，并将运行时间累加到总时间。

- 使用 `time.After` 模拟任务的完成时间。
- 监听 `context` 的取消信号，如果任务被取消，则提前退出。
- 使用 `sync.Mutex` 确保对共享变量 `totalTime` 的安全更新。

```

1. func task(id int, duration time.Duration, ctx context.Context, wg *sync.WaitGroup,
totalTime *time.Duration, mu *sync.Mutex) {
2.     defer wg.Done()
3.
4.     fmt.Printf("Task %d started, will run for %v\n", id, duration)
5.
6.     select {
7.     case <-time.After(duration): // 模拟任务执行完成
8.         mu.Lock()
9.         *totalTime += duration
10.        mu.Unlock()
11.        fmt.Printf("Task %d completed\n", id)
12.        case <-ctx.Done(): // 如果 context 被取消, 提前终止任务
13.            fmt.Printf("Task %d cancelled: %v\n", id, ctx.Err())
14.    }
15. }

```

(2) 主函数

Main 函数用于执行逻辑流, 包括任务创建、监控执行、输出结果。

- 创建任务: 使用随机数生成每个任务的执行时间, 并启动 goroutine 调用 task, 同时使用 WaitGroup 管理并发任务。

```

1. // 启动多个任务
2.     for i := 0; i < numTasks; i++ {
3.         wg.Add(1)
4.         go task(i+1, taskDurations[i], ctx, &wg, &totalTime, &mu)
5.     }

```

其中, taskDurations 使用随机数生成不同任务的完成所需时间:

```

1. taskDurations := []time.Duration{
2.     time.Duration(rand.Intn(5)+1) * time.Second,
3.     time.Duration(rand.Intn(5)+1) * time.Second,
4.     time.Duration(rand.Intn(5)+1) * time.Second,
5. }

```

- 监控任务执行: 使用一个 goroutine 在 WaitGroup.Wait 后检查任务总时

间是否超出限制。如果超时，通过 `context.WithCancel` 的取消函数终止所有未完成任务。

```
1. // 监控总超时时间
2.     go func() {
3.         wg.Wait()
4.         if totalTime > totalTimeout {
5.             cancel() // 超过总时间，取消所有任务
6.         }
7.     }()
```

- 输出结果。

```
/1. / 输出最终信息
2.     if totalTime > totalTimeout {
3.         fmt.Printf("Tasks cancelled: Total time %v exceeded %v\n",
totalTime, totalTimeout)
4.     } else {
5.         fmt.Printf("All tasks completed successfully in %v\n", totalTime)
6.     }
```

5.3 运行与结果

运行 `go run 5.go`，终端输出如下：

当任务完成时间分别为 3s,1s,4s 时，总时长为 8s，那么所有任务执行成功。

当任务完成时间分别为 3s,4s,4s 时，总时长为 11s，那么所有任务取消。

```
dba@ubuntu:~/homework2/5$ go run 5.go
Task 1 started, will run for 3s
Task 2 started, will run for 1s
Task 3 started, will run for 4s
Task 2 completed
Task 1 completed
Task 3 completed
All tasks completed successfully in 8s
dba@ubuntu:~/homework2/5$ go run 5.go
Task 1 started, will run for 3s
Task 3 started, will run for 4s
Task 2 started, will run for 4s
Task 1 completed
Task 3 completed
Task 2 completed
Tasks cancelled: Total time 11s exceeded 10s
```

5.4 完整代码



```
1 package main
2
3 import (
4     "context"
5     "fmt"
6     "math/rand"
7     "sync"
8     "time"
9 )
10
11 const (
12     numTasks = 3           // 任务数量
13     totalTimeout = 10 * time.Second // 总超时时间
14 )
15
16 // 模拟一个任务函数, 任务随机执行 1 到 5 秒钟
17 func task(id int, duration time.Duration, ctx context.Context, wg
    *sync.WaitGroup, totalTime *time.Duration, mu *sync.Mutex) {
18     defer wg.Done()
19
20     fmt.Printf("Task %d started, will run for %v\n", id,
    duration)
21
22     select {
23     case <-time.After(duration): // 模拟任务执行完成
24         mu.Lock()
25         *totalTime += duration
26         mu.Unlock()
27         fmt.Printf("Task %d completed\n", id)
28     case <-ctx.Done(): // 如果 context 被取消, 提前终止任务
29         fmt.Printf("Task %d cancelled: %v\n", id, ctx.Err())
30     }
31 }
32
33 func main() {
34     rand.Seed(time.Now().UnixNano()) // 随机数种子
35
36     // 创建一个带取消功能的 context
37     ctx, cancel := context.WithCancel(context.Background())
38     defer cancel()
39
40     var wg sync.WaitGroup
41     var mu sync.Mutex
42     totalTime := time.Duration(0) // 记录所有任务的总时间
43
44     taskDurations := []time.Duration{
45         time.Duration(rand.Intn(5)+1) * time.Second,
46         time.Duration(rand.Intn(5)+1) * time.Second,
47         time.Duration(rand.Intn(5)+1) * time.Second,
48     }
49
50     // 启动多个任务
51     for i := 0; i < numTasks; i++ {
52         wg.Add(1)
53         go task(i+1, taskDurations[i], ctx, &wg, &totalTime,
    &mu)
54     }
```



```
56 // 监控总超时时间
57 go func() {
58     wg.Wait()
59     if totalTime > totalTimeout {
60         cancel() // 超过总时间, 取消所有任务
61     }
62 }()
63
64 // 等待所有任务完成
65 wg.Wait()
66
67 // 输出最终信息
68 if totalTime > totalTimeout {
69     fmt.Printf("Tasks cancelled: Total time %v exceeded %v\n", totalTime, totalTimeout)
70 } else {
71     fmt.Printf("All tasks completed successfully in %v\n", totalTime)
72 }
```