1. 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)

设计思路：

在生产者-消费者问题中，最重要的是设计合理的同步机制，避免死锁和资源竞争。

1. 生产者和消费者的分离：

生产者：多个生产者并发运行，不断生成数据并将数据放入一个共享的缓冲通道中。

消费者：多个消费者从共享通道中获取数据进行消费。

2. 使用通道（Channel）作为缓冲区：

使用 chan 来作为生产者和消费者之间的共享资源。Go 的 channel 本身就是一个高效的线程安全的数据结构，它能够自动处理并发访问问题。

生产者向通道发送数据，消费者从通道读取数据。生产者和消费者并不会直接竞争内存或资源，而是通过通道进行交互。

3. 使用 sync.WaitGroup 等待所有任务完成：

sync.WaitGroup 用于等待所有的生产者和消费者 goroutine 完成任务。通过 wg.Add(1) 增加计数器，表示要等待的 goroutine 数量，wg.Done() 表示每个任务完成时减少计数器。

在所有任务完成后，wg.Wait() 会阻塞，直到所有的生产者和消费者都结束执行。

4. 这里避免死锁的关键点：

确保通道的关闭：生产者将数据推入通道，而消费者从通道中获取数据。为避免死锁，必须确保在所有生产者完成数据生产后关闭通道。否则，消费者可能会永远阻塞等待数据。

防止消费者等待生产者：消费者应该可以在通道中获取数据时正常退出，不需要等待某个生产者完成任务。如果在生产者结束后没有更多数据可消费，消费者应及时退出，避免死锁。

避免无限等待：当所有生产者都完成后，消费者应该能够检查通道是否关闭并退出，而不需要等待“空”通道的进一步数据。

5. 通道的缓冲区控制：

如果通道有缓冲区（make(chan int, bufferSize)），消费者不必每次都等待生产者生产新的数据，缓冲区可以临时存储数据，避免生产者过于依赖消费者的消费速度。

缓冲区的大小控制了生产者和消费者之间的并发流量，避免了过多的生产者或消费者同时工作导致的阻塞或死锁。

```go
package main

import (
	"fmt"
	"math/rand"
	"sync"
	"time"
)

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

func producer(id int, ch chan int, wg *sync.WaitGroup) {
	defer wg.Done()
	for i := 0; i < numItems; i++ {
		num := rand.Intn(100)
		ch <- num
		fmt.Printf("Producer %d produced %d\n", id, num)
		time.Sleep(time.Millisecond * 500)
	}
}

func consumer(id int, ch chan int, wg *sync.WaitGroup) {
	defer wg.Done()
	for num := range ch { // 从通道中读取，直到通道关闭
		fmt.Printf("Consumer %d consumed %d\n", id, num)
		time.Sleep(time.Millisecond * 500)
	}
}

func main() {
	rand.Seed(time.Now().UnixNano())
	ch := make(chan int, 10)

	var producerWG sync.WaitGroup // 专门用于等待生产者
	var consumerWG sync.WaitGroup // 专门用于等待消费者

	// 启动生产者
	for i := 1; i <= numProducers; i++ {
		producerWG.Add(1)
		go producer(i, ch, &producerWG)
	}

```

```
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          // 等待所有生产者完成，然后关闭通道
54          go func() {
55                  producerWG.Wait() // 等待所有生产者完成
56                  close(ch)         // 关闭通道，通知消费者结束
57          }()
58
59          consumerWG.Wait() // 等待所有消费者完成
60  }
61
```

```
haiden@haiden-virtual-machine: ~/go/code3
haiden@haiden-virtual-machine:~/go/code3$ go run 1.go
Producer 1 produced 74
Producer 2 produced 22
Producer 3 produced 11
Consumer 1 consumed 74
Consumer 2 consumed 22
Consumer 3 consumed 11
Producer 1 produced 40
Producer 2 produced 91
Consumer 2 consumed 91
Producer 3 produced 57
Consumer 1 consumed 40
Consumer 3 consumed 57
Producer 1 produced 95
Consumer 3 consumed 95
Producer 2 produced 9
Consumer 2 consumed 9
Producer 3 produced 92
Consumer 1 consumed 92
Producer 1 produced 64
Consumer 1 consumed 64
Producer 2 produced 17
Consumer 3 consumed 17
Producer 3 produced 11
Consumer 2 consumed 11
Producer 1 produced 35
Consumer 2 consumed 35
Producer 2 produced 0
Consumer 1 consumed 0
Producer 3 produced 2
Consumer 3 consumed 2
Producer 1 produced 45
Consumer 3 consumed 45
Producer 2 produced 56
Consumer 2 consumed 56
Producer 3 produced 64
Consumer 1 consumed 64
Producer 1 produced 17
Consumer 1 consumed 17
Producer 2 produced 44
Consumer 3 consumed 44
Producer 3 produced 45
Consumer 2 consumed 45
Producer 1 produced 40
Consumer 2 consumed 40
Producer 2 produced 26
Consumer 1 consumed 26
Producer 3 produced 23
Consumer 3 consumed 23
Producer 3 produced 93
Consumer 3 consumed 93
Producer 1 produced 67
Consumer 2 consumed 67
Producer 2 produced 42
Consumer 1 consumed 42
Producer 3 produced 43
Consumer 1 consumed 43
Producer 1 produced 85
Consumer 3 consumed 85
Producer 2 produced 77
Consumer 2 consumed 77
haiden@haiden-virtual-machine:~/go/code3$
```

2. 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。

设计思路：

数据源生成器：generateData 函数模拟一个数据源，它定期生成数据并发送到指定通道。

每个数据源会生成 dataCount 个数据，并通过通道传送到主程序。

多个通道和聚合通道：

channels[]：用来存放每个数据源的通道。

aggregateChannel：用于收集所有数据源的结果。

使用 select 聚合数据：

我们使用一个 select 语句监听多个数据源的通道，并将接收到的数据传送到聚合通道中。

主程序中的并发控制：

使用 sync.WaitGroup 确保所有数据源的生产者完成后，才开始关闭聚合通道。

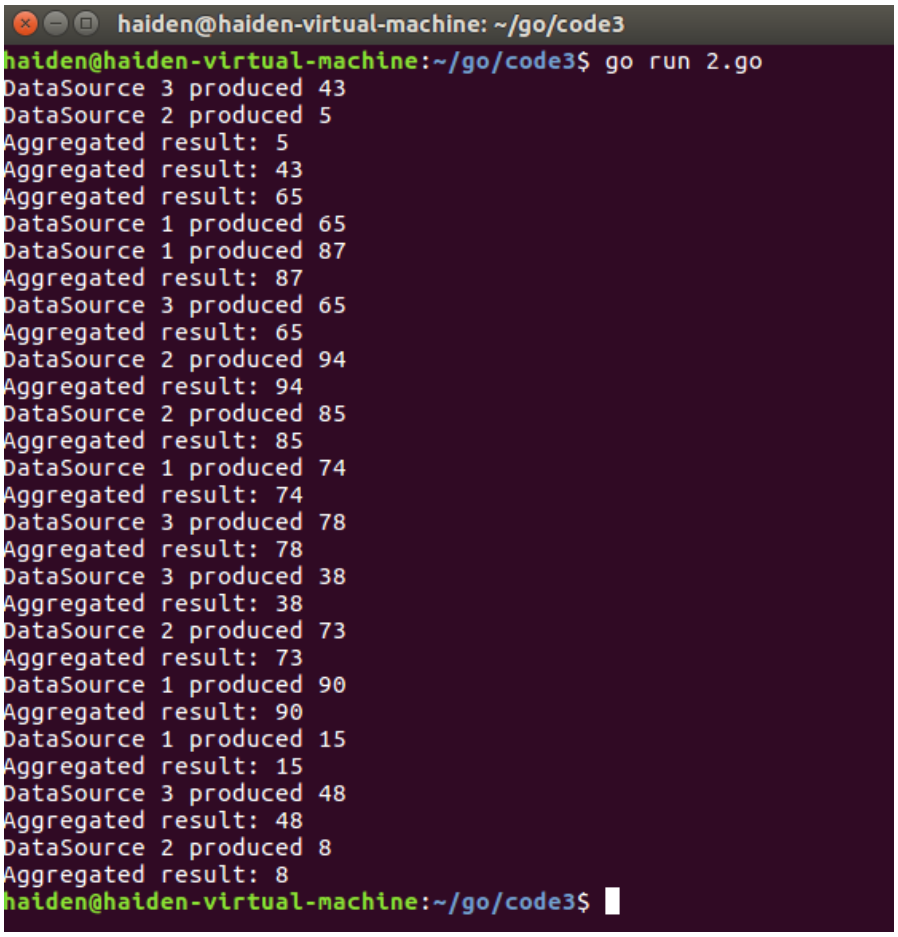
数据聚合：

在聚合通道中，我们从不同数据源接收数据，并汇总结果。

退出条件：

使用 time.Sleep(time.Second * 5)确保程序运行 5 秒钟以收集所有数据。

```go
package main

import (
	"fmt"
	"math/rand"
	"sync"
	"time"
)

const (
	numDataSources = 3    // 数据源数量
	dataCount      = 5    // 每个数据源发送的数据个数
)

func generateData(sourceID int, ch chan int, wg *sync.WaitGroup) {
	defer wg.Done()
	for i := 0; i < dataCount; i++ {
		data := rand.Intn(100) // 随机数据
		ch <- data
		fmt.Printf("DataSource %d produced %d\n", sourceID, data)
		time.Sleep(time.Millisecond * 500) // 模拟数据源产生数据的时间
	}
}

func main() {S
	rand.Seed(time.Now().UnixNano())

	var wg sync.WaitGroup
	channels := make([]chan int, numDataSources) // 用于存放每个数据源的通道
	aggregateChannel := make(chan int)           // 聚合通道，用于收集所有数据源的结果

	// 启动多个数据源（goroutines）
	for i := 0; i < numDataSources; i++ {
		channels[i] = make(chan int)
		wg.Add(1)
		go generateData(i+1, channels[i], &wg)
	}

	// 启动一个goroutine来聚合数据
	go func() {
		wg.Wait()    // 等待所有数据源的生产者完成
		close(aggregateChannel) // 关闭聚合通道
	}()


	// 使用select从多个通道收集数据
	go func() {
		for {
			select {
			case data, ok := <-channels[0]:
				if ok {
					aggregateChannel <- data
				}
			case data, ok := <-channels[1]:
				if ok {
					aggregateChannel <- data
				}
			case data, ok := <-channels[2]:
				if ok {
					aggregateChannel <- data
				}
			}
		}
	}()

	// 从聚合通道收集最终结果
	go func() {
		for data := range aggregateChannel {
			fmt.Printf("Aggregated result: %d\n", data)
		}
	}()

	// 等待所有goroutine完成
	time.Sleep(time.Second * 5)
}
```



3. 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}
```

设计思路：

1. **数据结构**:

   设计一个结构体 Counter，其中包含一个整型 count 作为计数器的值。

   使用 sync.RWMutex 来为 Counter 提供读写锁支持。sync.RWMutex

是 Go 中的读写互斥锁，它允许多个 goroutine 同时读共享资源，但在写操作时会阻塞所有的读操作和其他写操作，从而保证了对共享资源的独占访问。

2. **并发操作**:

   读取操作（GetValue）：使用 RLock() 和 RUnlock() 来实现读锁，多个读取 goroutine 可以并发读取计数器的值。

   修改操作（Increment）：使用 Lock() 和 Unlock() 来实现写锁，确保只有一个 goroutine 可以修改计数器的值，避免多个 goroutine 同时修改计数器时发生冲突。

3. **并发控制**:

   使用 sync.WaitGroup 来等待所有的并发任务完成。wg.Add(1) 用来增加任务计数，wg.Done() 用来表示任务完成，wg.Wait() 会等待所有任务完成。

4. **避免并发问题**:

   保证读操作和写操作的安全性：通过 sync.RWMutex 的读写锁机制，保证多个读操作并发执行的同时，也保证写操作的独占性，避免了读写冲突。

   并发任务完成的同步：通过 sync.WaitGroup 保证所有并发的读取器和写入器执行完毕后，主程序才能退出，避免主程序提前退出导致未完成的任务。

```go
package main

import (
	"fmt"
	"sync"
	"time"
)

type Counter struct {
	sync.RWMutex
	count int
}

// 读取计数器值
func (c *Counter) GetValue() int {
	c.RLock() // 使用读锁
	defer c.RUnlock()
	return c.count
}

// 增加计数器值
func (c *Counter) Increment() {
	c.Lock() // 使用写锁
	defer c.Unlock()
	c.count++
}

func reader(id int, counter *Counter, wg *sync.WaitGroup) {
	defer wg.Done()
	for i := 0; i < 5; i++ {
		value := counter.GetValue()
		fmt.Printf("Reader %d: Counter value = %d\n", id, value)
		time.Sleep(time.Millisecond * 200)
	}
}

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


func main() {
	var counter Counter
	var wg sync.WaitGroup

	// 启动多个读取器
	for i := 1; i <= 3; i++ {
		wg.Add(1)
		go reader(i, &counter, &wg)
	}

	// 启动多个写入器
	for i := 1; i <= 2; i++ {
		wg.Add(1)
		go writer(i, &counter, &wg)
	}

	// 等待所有goroutine完成
	wg.Wait()
}

```

```
haiden@haiden-virtual-machine:~/go/code3$ go run 3.go
Writer 2: Incremented counter
Reader 1: Counter value = 1
Reader 2: Counter value = 1
Reader 3: Counter value = 1
Writer 1: Incremented counter
Reader 3: Counter value = 2
Reader 1: Counter value = 2
Reader 2: Counter value = 2
Reader 2: Counter value = 2
Reader 3: Counter value = 2
Reader 1: Counter value = 2
Writer 2: Incremented counter
Writer 1: Incremented counter
Reader 1: Counter value = 4
Reader 2: Counter value = 4
Reader 3: Counter value = 4
Reader 3: Counter value = 4
Reader 1: Counter value = 4
Reader 2: Counter value = 4
Writer 2: Incremented counter
Writer 1: Incremented counter
haiden@haiden-virtual-machine:~/go/code3$
```

4. 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"}

设计思路：

Map 阶段:

每个字符串都在单独的 goroutine 中进行处理，将字符串的长度发送到一个共享的通道。

使用 sync.WaitGroup 确保所有的 Map 阶段任务完成后才进行 Reduce 阶段。

Reduce 阶段:

在所有的 Map 阶段完成后，汇总通道中传递过来的所有长度，计算字符串长度的总和。

使用 range 来读取通道中的值，直到通道关闭。

并发控制:

在 Map 阶段，每个字符串的长度计算通过一个 goroutine 来并发执行，因此

可以并行地处理多个字符串，提升效率。

使用 sync.WaitGroup 来同步 Map 和 Reduce 阶段，确保所有的 Map 任务完成后再开始 Reduce 阶段。

避免并发问题：

通道关闭：当所有 Map 阶段任务完成后，通过 close(resultChannel) 关闭通道，这样可以通知消费者（Reduce 阶段）不再有新的数据进入通道，防止消费者一直等待。

同步与并发：通过 sync.WaitGroup 来保证 Map 阶段的并发任务完成后才开始 Reduce 阶段的汇总，避免在 Reduce 阶段出现数据未准备好的情况。

```go
package main

import (
	"fmt"
	"sync"
)

func mapReduce(words []string) int {
	var wg sync.WaitGroup
	resultChannel := make(chan int, len(words)) // 使用缓冲通道来存储每个字符串的长度

	// 为每个字符串启动一个goroutine来计算其长度
	for _, word := range words {
		wg.Add(1)
		go func(w string) {
			defer wg.Done()
			resultChannel <- len(w) // 将字符串的长度发送到通道
		}(word)
	}

	// 等待所有goroutine完成
	wg.Wait()
	close(resultChannel)

	// 汇总所有结果
	totalLength := 0
	for length := range resultChannel {
		totalLength += length
	}

	return totalLength
}

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

```

```
haiden@haiden-virtual-machine:~/go/code3$ go run 4.go
Total length of all words: 21
haiden@haiden-virtual-machine:~/go/code3$
```

5.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

设计思路：

Map 阶段:

每个字符串都在单独的 goroutine 中进行处理，将字符串的长度发送到一个共享的通道。

使用 sync.WaitGroup 确保所有的 Map 阶段任务完成后才进行 Reduce 阶段。

Reduce 阶段:

在所有的 Map 阶段完成后，汇总通道中传递过来的所有长度，计算字符串长度的总和。

使用 range 来读取通道中的值，直到通道关闭。

并发控制:

在 Map 阶段，每个字符串的长度计算通过一个 goroutine 来并发执行，因此可以并行地处理多个字符串，提升效率。

使用 sync.WaitGroup 来同步 Map 和 Reduce 阶段，确保所有的 Map 任务完成后再开始 Reduce 阶段。

避免并发问题:

通道关闭：当所有 Map 阶段任务完成后，通过 close(resultChannel) 关闭通道，这样可以通知消费者（Reduce 阶段）不再有新的数据进入通道，防止消费者一直等待。

同步与并发：通过 sync.WaitGroup 来保证 Map 阶段的并发任务完成后才开始 Reduce 阶段的汇总，避免在 Reduce 阶段出现数据未准备好的情况。

```go
package main

import (
	"context"
	"fmt"
	"math/rand"
	"sync"
	"time"
)

const (
	numTasks = 5       // 任务数量
	timeout  = 2 * time.Second // 超时时间
)

// 模拟一个任务函数，任务随机执行 1 到 5 秒钟
func task(id int, ctx context.Context, wg *sync.WaitGroup) {
	defer wg.Done()

	// 随机任务执行时间
	taskDuration := time.Duration(rand.Intn(5)+1) * time.Second
	fmt.Printf("Task %d started, will run for %v\n", id, taskDuration)

	select {
	case <-time.After(taskDuration): // 模拟任务执行完成
		fmt.Printf("Task %d completed\n", id)
	case <-ctx.Done(): // 如果context被取消或超时，提前终止任务
		fmt.Printf("Task %d cancelled: %v\n", id, ctx.Err())
	}
}

func main() {
	rand.Seed(time.Now().UnixNano()) // 随机数种子

	// 创建一个带超时的 context
	ctx, cancel := context.WithTimeout(context.Background(), timeout)
	defer cancel() // 超时后自动取消所有任务

	var wg sync.WaitGroup

```

```
41          // 启动多个任务
42          for i := 1; i <= numTasks; i++ {
43                  wg.Add(1)
44                  go task(i, ctx, &wg)
45          }
46
47          // 等待所有任务完成或者超时
48          wg.Wait()
49
50          // 输出最终信息
51          if ctx.Err() == context.DeadlineExceeded {
52                  fmt.Println("Timed out: Not all tasks completed in time.")
53          } else {
54                  fmt.Println("All tasks completed successfully.")
55          }
56  }
```

```
haiden@haiden-virtual-machine:~/go/code3$ go run 5.go
Task 5 started, will run for 2s
Task 1 started, will run for 4s
Task 2 started, will run for 4s
Task 3 started, will run for 4s
Task 4 started, will run for 4s
Task 5 completed
Task 1 cancelled: context deadline exceeded
Task 4 cancelled: context deadline exceeded
Task 3 cancelled: context deadline exceeded
Task 2 cancelled: context deadline exceeded
Timed out: Not all tasks completed in time.
haiden@haiden-virtual-machine:~/go/code3$
```