o")`直接在main goroutine中执行。由于goroutines的执行调度由Go运行时管理，因此"hello"和"world"的打印顺序不确定。

二、Channels：安全的数据共享

概念介绍： Channel是Go中用于goroutines之间通信的桥梁，它允许我们安全地发送和接收数据。Channel本身是类型化的，这意味着它只能传输特定类型的数据，并且支持同步和缓冲两种模式。

基本使用：

1. **无缓冲Channel**：同步发送和接收，发送者必须等待接收者准备好接收数据。
2. **带缓冲Channel**：允许一定数量的数据暂存，从而解耦发送者和接收者的执行。

示例代码：

```
package main

import "fmt"

func main() {
// 创建一个无缓冲的channel
ch := make(chan int)

// 启动一个goroutine发送数据
go func() {
for i := 0; i < 5; i++ {
ch <- i     // 发送数据到channel
}
close(ch) // 发送完毕后关闭channel
}()

// 主goroutine接收数据
for data := range ch { // 使用for-range循环自动接收直到channel被关闭
fmt.Println(data)
```

}
}

```

### 三、Select：多路复用

**概念介绍**：`select`语句用于监控多个channel的操作，类似于在其他语言中的事件监听或多路复用。它可以等待多个channel的操作完成，并根据完成的channel执行相应的代码块。

**示例代码**：

```
package main

import (
”fmt”
”time”
)

func main() {
// 创建两个channel
c1 := make(cha 个读goroutine并发执行，提高了效率。

五、Context：优雅地取消和超时

概念介绍：`context`包提供了context类型，用于在API之间传递请求的截止时间、取消信号等信息。这对于管理长时间运行的goroutines特别有用，可以优雅地取消操作并清理资源。

示例代码：

```
package main

import (
”context”
”fmt”
```

```
"time"
)

func longRunningTask(ctx context.Context, id int) {
for {
select {
case <-ctx.Done(): // 检查是否被取消
fmt.Printf("Task %d was canceled\n", id)
return
default:
fmt.Printf("Task %d is running...\n", id)
time.Sleep(500 * time.Millisecond)
}
}
}

func main() {
ctx, cancel := context.WithTimeout(context.Background(),
2*time.Second) // 设置2秒超时
defer cancel() // 在结束时取消

go longRunningTask(ctx, 1)
time.Sleep(3 * time.Second) // 等待一段时间以观察输出
}
```

六、总结

Go语言通过goroutines和channels提供了一种简洁而强大的并发编程模型，使得开发者能够轻松地编写出高并发、高性能的应用程序。掌握这些基本概念和技巧，是进行Go异步编程的基础。实践这些示例，并尝试将它们应用到实际项目中，将有助于深入理解Go的并发机制，进而构建更加复杂和高效的系统。

原文链接: https://juejin.cn/post/7363650007269392399