



Go Context 并发编程简明教程

Go 简明教程系列文章链接:

- [Go 语言简明教程](#) (Aug 6, 2019)
- [Go Gin 简明教程](#) (Aug 7, 2019)
- [Go2 新特性简明教程](#) (Aug 15, 2019)
- [Go Protobuf 简明教程](#) (Jan 11, 2020)
- [Go RPC & TLS 鉴权简明教程](#) (Jan 13, 2020)
- [Go WebAssembly \(Wasm\) 简明教程](#) (Jan 23, 2020)
- [Go Test 单元测试简明教程](#) (Feb 10, 2020)
- [Go Mock \(gomock\)简明教程](#) (Feb 14, 2020)
- [Go Mmap 文件内存映射简明教程](#) (Apr 20, 2020)
- [Go Context 并发编程简明教程](#) (Apr 20, 2020)

1 为什么需要 Context

WaitGroup 和信道(channel)是常见的 2 种并发控制的方式。

如果并发启动了多个子协程，需要等待所有的子协程完成任务，WaitGroup 非常适合于这类场景，例如下面的例子：

```
1  var wg sync.WaitGroup
2
3  func doTask(n int) {
4      time.Sleep(time.Duration(n))
5      fmt.Printf("Task %d Done\n", n)
6      wg.Done()
7  }
8
9  func main() {
10     for i := 0; i < 3; i++ {
11         wg.Add(1)
12         go doTask(i + 1)
13     }
14     wg.Wait()
15     fmt.Println("All Task Done")
16 }
```



```
1 Task 3 Done
2 Task 1 Done
3 Task 2 Done
4 All Task Done
```

WaitGroup 只是傻傻地等待子协程结束，但是并不能主动通知子协程退出。假如开启了一个定时轮询的子协程，有没有什么办法，通知该子协程退出呢？这种场景下，可以使用 `select+chan` 的机制。

```
1 var stop chan bool
2
3 func reqTask(name string) {
4     for {
5         select {
6             case <-stop:
7                 fmt.Println("stop", name)
8                 return
9             default:
10                fmt.Println(name, "send request")
11                time.Sleep(1 * time.Second)
12            }
13        }
14    }
15
16 func main() {
17     stop = make(chan bool)
18     go reqTask("worker1")
19     time.Sleep(3 * time.Second)
20     stop <- true
21     time.Sleep(3 * time.Second)
22 }
```

子协程使用 `for` 循环定时轮询，如果 `stop` 信道有值，则退出，否则继续轮询。

```
1 worker1 send request
2 worker1 send request
3 worker1 send request
4 stop worker1
```

更复杂的场景如何做并发控制呢？比如子协程中开启了新的子协程，或者需要同时控制多个子协程。这种场景下，`select+chan` 的方式就显得力不从心了。



- 通知子协程退出（正常退出，超时退出等）；
- 传递必要的参数。

2 context.WithCancel

`context.WithCancel()` 创建可取消的 Context 对象，即可以主动通知子协程退出。

2.1 控制单个协程

使用 Context 改写上述的例子，效果与 `select+chan` 相同。

```
1 func reqTask(ctx context.Context, name string) {
2     for {
3         select {
4             case <-ctx.Done():
5                 fmt.Println("stop", name)
6                 return
7             default:
8                 fmt.Println(name, "send request")
9                 time.Sleep(1 * time.Second)
10        }
11    }
12 }
13
14 func main() {
15     ctx, cancel := context.WithCancel(context.Background())
16     go reqTask(ctx, "worker1")
17     time.Sleep(3 * time.Second)
18     cancel()
19     time.Sleep(3 * time.Second)
20 }
```

- `context.Background()` 创建根 Context，通常在 main 函数、初始化和测试代码中创建，作为顶层 Context。
- `context.WithCancel(parent)` 创建可取消的子 Context，同时返回函数 `cancel`。
- 在子协程中，使用 `select` 调用 `<-ctx.Done()` 判断是否需要退出。
- 主协程中，调用 `cancel()` 函数通知子协程退出。

2.2 控制多个协程

```
1 func main() {
2     ctx, cancel := context.WithCancel(context.Background())
```



```
5      go reqTask(ctx, "worker2")
6
7      time.Sleep(3 * time.Second)
8      cancel()
9      time.Sleep(3 * time.Second)
10 }
```

为每个子协程传递相同的上下文 ctx 即可，调用 cancel() 函数后该 Context 控制的所有子协程都会退出。

```
1  worker1 send request
2  worker2 send request
3  worker1 send request
4  worker2 send request
5  worker1 send request
6  worker2 send request
7  stop worker1
8  stop worker2
```

3 context.WithValue

如果需要往子协程中传递参数，可以使用 context.WithValue()。

```
1  type Options struct{ Interval time.Duration }
2
3  func reqTask(ctx context.Context, name string) {
4      for {
5          select {
6              case <-ctx.Done():
7                  fmt.Println("stop", name)
8                  return
9              default:
10                 fmt.Println(name, "send request")
11                 op := ctx.Value("options").(*Options)
12                 time.Sleep(op.Interval * time.Second)
13             }
14         }
15     }
16
17     func main() {
18         ctx, cancel := context.WithCancel(context.Background())
19         vCtx := context.WithValue(ctx, "options", &Options{1})
```



```
22     go reqTask(vCtx, "worker2")
23
24     time.Sleep(3 * time.Second)
25     cancel()
26     time.Sleep(3 * time.Second)
27 }
```

- `context.WithValue()` 创建了一个基于 `ctx` 的子 `Context`，并携带了值 `options`。
- 在子协程中，使用 `ctx.Value("options")` 获取到传递的值，读取/修改该值。

4 context.WithTimeout

如果需要控制子协程的执行时间，可以使用 `context.WithTimeout` 创建具有超时通知机制的 `Context` 对象。

```
1 func main() {
2     ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
3     go reqTask(ctx, "worker1")
4     go reqTask(ctx, "worker2")
5
6     time.Sleep(3 * time.Second)
7     fmt.Println("before cancel")
8     cancel()
9     time.Sleep(3 * time.Second)
10 }
```

`WithTimeout()` 的使用与 `WithCancel()` 类似，多了一个参数，用于设置超时时间。执行结果如下：

```
1 worker2 send request
2 worker1 send request
3 worker1 send request
4 worker2 send request
5 stop worker2
6 stop worker1
7 before cancel
```

因为超时时间设置为 2s，但是 `main` 函数中，3s 后才会调用 `cancel()`，因此，在调用 `cancel()` 函数前，子协程因为超时已经退出了。

5 context.WithDeadline

超时退出可以控制子协程的最长执行时间，那 `context.WithDeadline()` 则可以控制子协程的最迟退出时间。



```
2      for {
3          select {
4              case <-ctx.Done():
5                  fmt.Println("stop", name, ctx.Err())
6                  return
7              default:
8                  fmt.Println(name, "send request")
9                  time.Sleep(1 * time.Second)
10         }
11     }
12 }
13
14 func main() {
15     ctx, cancel := context.WithDeadline(context.Background(), time.Now().Add(1*time.Second))
16     go reqTask(ctx, "worker1")
17     go reqTask(ctx, "worker2")
18
19     time.Sleep(3 * time.Second)
20     fmt.Println("before cancel")
21     cancel()
22     time.Sleep(3 * time.Second)
23 }
```

- WithDeadline 用于设置截止时间。在这个例子中，将截止时间设置为1s后，cancel() 函数在 3s 后调用，因此子协程将在调用 cancel() 函数前结束。
- 在子协程中，可以通过 ctx.Err() 获取到子协程退出的错误原因。

运行结果如下：

```
1  worker2 send request
2  worker1 send request
3  stop worker2 context deadline exceeded
4  stop worker1 context deadline exceeded
5  before cancel
```

可以看到，子协程 worker1 和 worker2 均是因为截止时间到了而退出。

专题: [Go 简明教程](#)

本文发表于 2020-04-20，最后修改于 2021-02-06。

本站永久域名「geektutu.com」，也可搜索「极客兔兔」找到我。

[赞赏支持](#)

推荐阅读

Go 语言陷阱 - 数组和切片

发表于2020-12-07, 阅读约9分钟

机器学习笔试面试题 11-20

发表于2019-08-06, 阅读约15分钟

TensorFlow 2 中文文档 - 卷积神经网络分类 CIFAR-10

发表于2019-07-19, 阅读约22分钟

[#关于我 \(9\)](#) [#Go \(48\)](#) [#百宝箱 \(2\)](#) [#Cheat Sheet \(1\)](#) [#Go语言高性能编程 \(20\)](#) [#友链 \(1\)](#) [#Pandas \(3\)](#)

[#机器学习 \(9\)](#) [#TensorFlow \(9\)](#) [#mnist \(5\)](#) [#Python \(10\)](#) [#强化学习 \(3\)](#) [#OpenAI gym \(4\)](#) [#DQN \(1\)](#)

[#Q-Learning \(1\)](#) [#CNN \(1\)](#) [#TensorFlow 2 \(10\)](#) [#官方文档 \(10\)](#) [#Rust \(1\)](#)

0 条评论

未登录用户



说点什么

支持 Markdown 语法

使用 GitHub 登录

预览

来做第一个留言的人吧!

字符串拼接性能及原理

4 评论 • 6天前



[l-xue-yu](#) —— 写的不错, 详略得当, 有理有据。好好和大佬学习。

动手写RPC框架 - GeeRPC第七天 服务发现与注册中心(registry)


4 评论 • 29天前






Go 空结构体 struct{} 的使用

2 评论 • 5天前

 **xiezhenyu19970913** —— 学到了很多细节，感谢!


动手写ORM框架 - GeeORM第五天 实现钩子(Hooks)

3 评论 • 24天前

 **shiluoye** —— 用MethodByName(method)实在太骚了，为什么不用interface，gorm的

Gitalk Plus

© 2021 - 极客兔兔 - 沪ICP备18001798号-1

Powered by [Hexo](#) | Theme [Geektutu](#)  [Star](#)

👁996643 📄4206