



下载APP



18 | 分组操作：处理一组子任务，该用什么并发原语？

2020-11-20 晁岳攀

Go 并发编程实战课

[进入课程 >](#)

讲述：安晓辉

时长 23:26 大小 21.47M



你好，我是鸟窝。

共享资源保护、任务编排和消息传递是 Go 并发编程中常见的场景，而**分组执行一批相同的或类似的任务则是任务编排中一类情形**，所以，这节课，我专门来介绍一下分组编排的一些常用场景和并发原语，包括 ErrGroup、gollback、Hunch 和 schedgroup。

我们先来学习一类非常常用的并发原语，那就是 ErrGroup。

ErrGroup



🔗**ErrGroup**是 Go 官方提供的一个同步扩展库。我们经常会碰到需要将一个通用的父任务拆成几个小任务并发执行的场景，其实，将一个大的任务拆成几个小任务并发执行，可以

有效地提高程序的并发度。就像你在厨房做饭一样，你可以在蒸米饭的同时炒几个小菜，米饭蒸好了，菜同时也做好了，很快就能吃到可口的饭菜。

ErrGroup 就是用来应对这种场景的。它和 WaitGroup 有些类似，但是它提供功能更加丰富：

和 Context 集成；

error 向上传播，可以把子任务的错误传递给 Wait 的调用者。

接下来，我来给你介绍一下 ErrGroup 的基本用法和几种应用场景。

基本用法

golang.org/x/sync/errgroup 包下定义了一个 Group struct，它就是我们要介绍的 ErrGroup 并发原语，底层也是基于 WaitGroup 实现的。

在使用 ErrGroup 时，我们要用到三个方法，分别是 WithContext、Go 和 Wait。

1.WithContext

在创建一个 Group 对象时，需要使用 WithContext 方法：

```
1 func WithContext(ctx context.Context) (*Group, context.Context)
```

[复制代码](#)

这个方法返回一个 Group 实例，同时还会返回一个使用 context.WithCancel(ctx) 生成的新 Context。一旦有一个子任务返回错误，或者是 Wait 调用返回，这个新 Context 就会被 cancel。

Group 的零值也是合法的，只不过，你就没有一个可以监控是否 cancel 的 Context 了。

注意，如果传递给 WithContext 的 ctx 参数，是一个可以 cancel 的 Context 的话，那么，它被 cancel 的时候，并不会终止正在执行的子任务。

2.Go

我们再来学习下执行子任务的 Go 方法：

```
1 func (g *Group) Go(f func() error)
```

[复制代码](#)

传入的子任务函数 f 是类型为 func() error 的函数，如果任务执行成功，就返回 nil，否则就返回 error，并且会 cancel 那个新的 Context。

一个任务可以分成好多个子任务，而且，可能有多个子任务执行失败返回 error，不过，Wait 方法只会返回第一个错误，所以，如果想返回所有的错误，需要特别的处理，我先留个小悬念，一会儿再讲。

3.Wait

类似 WaitGroup，Group 也有 Wait 方法，等所有的子任务都完成后，它才会返回，否则只会阻塞等待。如果有多个子任务返回错误，它只会返回第一个出现的错误，如果所有的子任务都执行成功，就返回 nil：

```
1 func (g *Group) Wait() error
```

[复制代码](#)

ErrGroup 使用例子

好了，知道了基本用法，下面我来给你介绍几个例子，帮助你全面地掌握 ErrGroup 的使用方法和应用场景。

简单例子：返回第一个错误

先来看一个简单的例子。在这个例子中，启动了三个子任务，其中，子任务 2 会返回执行失败，其它两个执行成功。在三个子任务都执行后，group.Wait 才会返回第 2 个子任务的错误。

```
1 package main
2
3
4 import (
5     "errors"
6     "fmt"
7     "time"
8
9     "golang.org/x/sync/errgroup"
10 )
11
12 func main() {
13     var g errgroup.Group
14
15
16     // 启动第一个子任务,它执行成功
17     g.Go(func() error {
18         time.Sleep(5 * time.Second)
19         fmt.Println("exec #1")
20         return nil
21     })
22     // 启动第二个子任务, 它执行失败
23     g.Go(func() error {
24         time.Sleep(10 * time.Second)
25         fmt.Println("exec #2")
26         return errors.New("failed to exec #2")
27     })
28
29     // 启动第三个子任务, 它执行成功
30     g.Go(func() error {
31         time.Sleep(15 * time.Second)
32         fmt.Println("exec #3")
33         return nil
34     })
35     // 等待三个任务都完成
36     if err := g.Wait(); err == nil {
37         fmt.Println("Successfully exec all")
38     } else {
39         fmt.Println("failed:", err)
40     }
41 }
```

如果执行下面的这个程序，会显示三个任务都执行了，而 Wait 返回了子任务 2 的错误：

```


smallnest > ...kshop/9.group/errgroup > ? master ● go run errg.go
exec #1
exec #2
exec #3
failed: failed to exec #2
smallnest > ...kshop/9.group/errgroup > ? master ● |

```

更进一步，返回所有子任务的错误

Group 只能返回子任务的第一个错误，后续的错误都会被丢弃。但是，有时候我们需要知道每个任务的执行情况。怎么办呢？这个时候，我们就可以用稍微有点曲折的方式去实现。我们使用一个 result slice 保存子任务的执行结果，这样，通过查询 result，就可以知道每一个子任务的结果了。

下面的这个例子，就是使用 result 记录每个子任务成功或失败的结果。其实，你不仅可以使 result 记录 error 信息，还可以用它记录计算结果。

 复制代码

```

1 package main
2
3 import (
4     "errors"
5     "fmt"
6     "time"
7
8     "golang.org/x/sync/errgroup"
9 )
10
11 func main() {
12     var g errgroup.Group
13     var result = make([]error, 3)
14
15     // 启动第一个子任务,它执行成功
16     g.Go(func() error {
17         time.Sleep(5 * time.Second)
18         fmt.Println("exec #1")
19         result[0] = nil // 保存成功或者失败的结果
20         return nil
21     })
22
23
24     // 启动第二个子任务,它执行失败
25     g.Go(func() error {
26         time.Sleep(10 * time.Second)
27         fmt.Println("exec #2")
28
29         result[1] = errors.New("failed to exec #2") // 保存成功或者失败的结果

```

```
30         return result[1]
31     })
32
33     // 启动第三个子任务，它执行成功
34     g.Go(func() error {
35         time.Sleep(15 * time.Second)
36         fmt.Println("exec #3")
37         result[2] = nil // 保存成功或者失败的结果
38         return nil
39     })
40
41     if err := g.Wait(); err == nil {
42         fmt.Printf("Successfully exec all. result: %v\n", result)
43     } else {
44         fmt.Printf("failed: %v\n", result)
45     }
46 }
```

任务执行流水线 Pipeline

Go 官方文档中还提供了一个 pipeline 的例子。这个例子是说，由一个子任务遍历文件夹下的文件，然后把遍历出的文件交给 20 个 goroutine，让这些 goroutine 并行计算文件的 md5。

这个例子中的计算逻辑你不需要重点掌握，我来把这个例子简化一下（如果你想看原始的代码，可以看[这里](#)）：

```
1 package main
2
3 import (
4     .....
5     "golang.org/x/sync/errgroup"
6 )
7
8 // 一个多阶段的pipeline.使用有限的goroutine计算每个文件的md5值.
9 func main() {
10     m, err := MD5All(context.Background(), ".")
11     if err != nil {
12         log.Fatal(err)
13     }
14
15     for k, sum := range m {
16         fmt.Printf("%s:\t%x\n", k, sum)
17     }
18 }
```

[复制代码](#)

```
19 type result struct {
20     path string
21     sum  [md5.Size]byte
22 }
23
24 // 遍历根目录下所有的文件和子文件夹,计算它们的md5的值.
25 func MD5All(ctx context.Context, root string) (map[string][md5.Size]byte, error,
26     ctx := errgroup.WithContext(ctx)
27     paths := make(chan string) // 文件路径channel
28
29     g.Go(func() error {
30         defer close(paths) // 遍历完关闭paths chan
31         return filepath.Walk(root, func(path string, info os.FileInfo, err error)
32             ..... //将文件路径放入到paths
33             return nil
34         })
35     })
36
37 // 启动20个goroutine执行计算md5的任务, 计算的文件由上一阶段的文件遍历子任务生成.
38 c := make(chan result)
39 const numDigesters = 20
40 for i := 0; i < numDigesters; i++ {
41     g.Go(func() error {
42         for path := range paths { // 遍历直到paths chan被关闭
43             ..... // 计算path的md5值, 放入到c中
44         }
45         return nil
46     })
47 }
48 go func() {
49     g.Wait() // 20个goroutine以及遍历文件的goroutine都执行完
50     close(c) // 关闭收集结果的chan
51 }()
52
53
54 m := make(map[string][md5.Size]byte)
55 for r := range c { // 将md5结果从chan中读取到map中,直到c被关闭才退出
56     m[r.path] = r.sum
57 }
58
59 // 再次调用Wait, 依然可以得到group的错误信息
60 if err := g.Wait(); err != nil {
61     return nil, err
62 }
63 return m, nil
64 }
65 }
```

通过这个例子，你可以学习到多阶段 pipeline 的实现（这个例子是遍历文件夹和计算 md5 两个阶段），还可以学习到如何控制执行子任务的 goroutine 数量。

很多公司都在使用 ErrGroup 处理并发子任务，比如 Facebook、bilibili 等公司的一些项目，但是，这些公司在使用的时候，发现了一些不方便的地方，或者说，官方的 ErrGroup 的功能还不够丰富。所以，他们都对 ErrGroup 进行了扩展。接下来呢，我就带你看看几个扩展库。

扩展库

[bilibili/errgroup](#)

如果我们无限制地直接调用 ErrGroup 的 Go 方法，就可能会创建出非常多的 goroutine，太多的 goroutine 会带来调度和 GC 的压力，而且也会占用更多的内存资源。就像 [go#34457](#) 指出的那样，当前 Go 运行时创建的 g 对象只会增长和重用，不会回收，所以在高并发的情况下，也要尽可能减少 goroutine 的使用。

常用的一个手段就是使用 worker pool(goroutine pool)，或者是类似 [containerd/stargz-snapshotter](#) 的方案，使用前面我们讲的信号量，信号量的资源的数量就是可以并行的 goroutine 的数量。但是在这一讲，我来介绍一些其它的手段，比如下面介绍的 bilibili 实现的 errgroup。

bilibili 实现了一个扩展的 ErrGroup，可以使用一个固定数量的 goroutine 处理子任务。如果不设置 goroutine 的数量，那么每个子任务都会比较“放肆地”创建一个 goroutine 并发执行。

这个链接里的文档已经很详细地介绍了它的几个扩展功能，所以我不通过示例的方式进行讲解了。

除了可以控制并发 goroutine 的数量，它还提供了 2 个功能：

1. cancel，失败的子任务可以 cancel 所有正在执行任务；
2. recover，而且会把 panic 的堆栈信息放到 error 中，避免子任务 panic 导致的程序崩溃。

但是，有一点不太好的地方就是，一旦你设置了并发数，超过并发数的子任务需要等到调用者调用 Wait 之后才会执行，而不是只要 goroutine 空闲下来，就去执行。如果不注意这一点的话，可能会出现子任务不能及时处理的情况，这是这个库可以优化的一点。

另外，这个库其实是有一个并发问题的。在高并发的情况下，如果任务数大于设定的 goroutine 的数量，并且这些任务被集中加入到 Group 中，这个库的处理方式是把子任务加入到一个数组中，但是，这个数组不是线程安全的，有并发问题，问题就在于，下面图片中的标记为 96 行的那一行，这一行对 slice 的 append 操作不是线程安全的：

```
90 func (g *Group) Go(f func(ctx context.Context) error) {
91     g.wg.Add(1)
92     if g.ch != nil {
93         select {
94             case g.ch <- f:
95             default:
96                 g.chs = append(g.chs, f)
97         }
98         return
99     }
100     go g.do(f)
101 }
```

我们可以写一个简单的程序来测试这个问题：

[复制代码](#)


```
1 package main
2
3 import (
4     "context"
5     "fmt"
6     "sync/atomic"
7     "time"
8
9     "github.com/bilibili/kratos/pkg/sync/errgroup"
10 )
11
12 func main() {
13     var g errgroup.Group
14     g.GOMAXPROCS(1) // 只使用一个goroutine处理子任务
15
16     var count int64
17     g.Go(func(ctx context.Context) error {
18         time.Sleep(time.Second) //睡眠5秒，把这个goroutine占住
19         return nil
20     })
21
22     total := 10000
23
24     for i := 0; i < total; i++ { // 并发一万个goroutine执行子任务，理论上这些子任务都
```

```
25     go func() {
26         g.Go(func(ctx context.Context) error {
27             atomic.AddInt64(&count, 1)
28             return nil
29         })
30     }()
31 }
32
33 // 等待所有的子任务完成。理论上10001个子任务都会被完成
34 if err := g.Wait(); err != nil {
35     panic(err)
36 }
37
38 got := atomic.LoadInt64(&count)
39 if got != int64(total) {
40     panic(fmt.Sprintf("expect %d but got %d", total, got))
41 }
42 }
```

运行这个程序的话，你就会发现死锁问题，因为我们的测试程序是一个简单的命令行工具，程序退出的时候，Go runtime 能检测到死锁问题。如果是一直运行的服务器程序，死锁问题有可能是检测不出来的，程序一直会 hang 在 Wait 的调用上。

🔗 [neilotoole/errgroup](#)

neilotoole/errgroup 是今年年中新出现的一个 ErrGroup 扩展库，它可以直接替换官方的 ErrGroup，方法都一样，原有功能也一样，只不过**增加了可以控制并发 goroutine 的功能**。它的方法集如下：

 复制代码

```
1 type Group
2 func WithContext(ctx context.Context) (*Group, context.Context)
3 func WithContextN(ctx context.Context, numG, qSize int) (*Group, context.Context)
4 func (g *Group) Go(f func() error)
5 func (g *Group) Wait() error
```

新增加的方法 WithContextN，可以设置并发的 goroutine 数，以及等待处理的子任务队列的大小。当队列满的时候，如果调用 Go 方法，就会被阻塞，直到子任务可以放入到队列中才返回。如果你传给这两个参数的值不是正整数，它就会使用 runtime.NumCPU 代替你传入的参数。

当然，你也可以把 bilibili 的 recover 功能扩展到这个库中，以避免子任务的 panic 导致程序崩溃。

🔗 [facebookgo/errgroup](https://github.com/facebookgo/errgroup)

Facebook 提供的这个 ErrGroup，其实并不是对 Go 扩展库 ErrGroup 的扩展，而是对标准库 WaitGroup 的扩展。不过，因为它们的名字一样，处理的场景也类似，所以我把它也列在了这里。

标准库的 WaitGroup 只提供了 Add、Done、Wait 方法，而且 Wait 方法也没有返回子 goroutine 的 error。而 Facebook 提供的 ErrGroup 提供的 Wait 方法可以返回 error，而且可以包含多个 error。子任务在调用 Done 之前，可以把自己的 error 信息设置给 ErrGroup。接着，Wait 在返回的时候，就会把这些 error 信息返回给调用者。

我们来看下 Group 的方法：

```
1 type Group
2 func (g *Group) Add(delta int)
3 func (g *Group) Done()
4 func (g *Group) Error(e error)
5 func (g *Group) Wait() error
```

[📄 复制代码](#)

关于 Wait 方法，我刚刚已经介绍了它和标准库 WaitGroup 的不同，我就不多说了。这里还有一个不同的方法，就是 Error 方法，

我举个例子演示一下 Error 的使用方法。

在下面的这个例子中，第 26 行的子 goroutine 设置了 error 信息，第 39 行会把这个 error 信息输出出来。

```
1 package main
2
3 import (
4     "errors"
5     "fmt"
6     "time"
```

[📄 复制代码](#)

```
7
8     "github.com/facebookgo/errgroup"
9 )
10
11 func main() {
12     var g errgroup.Group
13     g.Add(3)
14
15     // 启动第一个子任务,它执行成功
16     go func() {
17         time.Sleep(5 * time.Second)
18         fmt.Println("exec #1")
19         g.Done()
20     }()
21
22     // 启动第二个子任务, 它执行失败
23     go func() {
24         time.Sleep(10 * time.Second)
25         fmt.Println("exec #2")
26         g.Error(errors.New("failed to exec #2"))
27         g.Done()
28     }()
29
30     // 启动第三个子任务, 它执行成功
31     go func() {
32         time.Sleep(15 * time.Second)
33         fmt.Println("exec #3")
34         g.Done()
35     }()
36
37     // 等待所有的goroutine完成, 并检查error
38     if err := g.Wait(); err == nil {
39         fmt.Println("Successfully exec all")
40     } else {
41         fmt.Println("failed:", err)
42     }
43 }
```

关于 ErrGroup, 你掌握这些就足够了, 接下来, 我再介绍几种有趣而实用的 Group 并发原语。这些并发原语都是控制一组子 goroutine 执行的面向特定场景的并发原语, 当你遇见这些特定场景时, 就可以参考这些库。

其它实用的 Group 并发原语

SizedGroup/ErrSizedGroup

[go-pkgz/syncs](#)提供了两个 Group 并发原语，分别是 SizedGroup 和 ErrSizedGroup。


SizedGroup 内部是使用信号量和 WaitGroup 实现的，它通过信号量控制并发的 goroutine 数量，或者是不控制 goroutine 数量，只控制子任务并发执行时候的数量（通过）。

它的代码实现非常简洁，你可以到它的代码库中了解它的具体实现，你一看就明白了，我就不多说了。下面我重点说说它的功能。

默认情况下，SizedGroup 控制的是子任务的并发数量，而不是 goroutine 的数量。在这种方式下，每次调用 Go 方法都不会被阻塞，而是新建一个 goroutine 去执行。

如果想控制 goroutine 的数量，你可以使用 syncs.Preemptive 设置这个并发原语的可选项。如果设置了这个可选项，但在调用 Go 方法的时候没有可用的 goroutine，那么调用者就会等待，直到有 goroutine 可以处理这个子任务才返回，这个控制在内部是使用信号量实现的。

我们来看一个使用 SizedGroup 的例子：

 复制代码

```
1 package main
2
3 import (
4     "context"
5     "fmt"
6     "sync/atomic"
7     "time"
8
9     "github.com/go-pkgz/syncs"
10 )
11
12 func main() {
13     // 设置goroutine数是10
14     swg := syncs.NewSizedGroup(10)
15     // swg := syncs.NewSizedGroup(10, syncs.Preemptive)
16     var c uint32
17
18     // 执行1000个子任务，只有10个goroutine去执行
19     for i := 0; i < 1000; i++ {
20         swg.Go(func(ctx context.Context) {
```

```
21         time.Sleep(5 * time.Millisecond)
22         atomic.AddUint32(&c, 1)
23     })
24 }
25
26 // 等待任务完成
27 swg.Wait()
28 // 输出结果
29 fmt.Println(c)
30 }
```

ErrSizedGroup 为 SizedGroup 提供了 error 处理的功能，它的功能和 Go 官方扩展库的功能一样，就是等待子任务完成并返回第一个出现的 error。不过，它还提供了额外的功能，我来介绍一下。

第一个额外的功能，就是可以控制并发的 goroutine 数量，这和 SizedGroup 的功能一样。


第二个功能是，如果设置了 termOnError，子任务出现第一个错误的时候会 cancel Context，而且后续的 Go 调用会直接返回，Wait 调用者会得到这个错误，这相当于是遇到错误快速返回。如果没有设置 termOnError，Wait 会返回所有的子任务的错误。

不过，ErrSizedGroup 和 SizedGroup 设计得不太一致的地方是，**SizedGroup 可以把 Context 传递给子任务，这样可以通过 cancel 让子任务中断执行，但是 ErrSizedGroup 却没有实现。我认为，这是一个值得加强的地方。**

总体来说，syncs 包提供的并发原语的质量和功能还是非常赞的。不过，目前的 star 只有十几个，这和它的功能严重不匹配，我建议你 star 这个项目，支持一下作者。

好了，关于 ErrGroup，你掌握这些就足够了，下面我再来给你介绍一些非 ErrGroup 的并发原语，它们用来编排子任务。

gollback

 **gollback** 也是用来处理一组子任务的执行的，不过它解决了 ErrGroup 收集子任务返回结果的痛点。使用 ErrGroup 时，如果你要收到子任务的结果和错误，你需要定义额外的变量收集执行结果和错误，但是这个库可以提供更便利的方式。

我刚刚在说官方扩展库 ErrGroup 的时候，举了一些例子（返回第一个错误的例子和返回所有子任务错误的例子），在例子中，如果想得到每一个子任务的结果或者 error，我们需要额外提供一个 result slice 进行收集。使用 gollback 的话，就不需要这些额外的处理了，因为它的方法会把结果和 error 信息都返回。

接下来，我们看一下它提供的三个方法，分别是 **All**、**Race** 和 **Retry**。

All 方法

All 方法的签名如下：

[复制代码](#)

```
1 func All(ctx context.Context, fns ...AsyncFunc) ([]interface{}, []error)
```

它会等待所有的异步函数（AsyncFunc）都执行完才返回，而且返回结果的顺序和传入的函数的顺序保持一致。第一个返回参数是子任务的执行结果，第二个参数是子任务执行时的错误信息。

其中，异步函数的定义如下：

[复制代码](#)

```
1 type AsyncFunc func(ctx context.Context) (interface{}, error)
```

可以看到，ctx 会被传递给子任务。如果你 cancel 这个 ctx，可以取消子任务。

我们来看一个使用 All 方法的例子：

[复制代码](#)

```
1 package main
2
3 import (
4     "context"
5     "errors"
6     "fmt"
7     "github.com/vardius/gollback"
8     "time"
```



```
9 )
10
11 func main() {
12     rs, errs := gollback.All( // 调用All方法
13         context.Background(),
14         func(ctx context.Context) (interface{}, error) {
15             time.Sleep(3 * time.Second)
16             return 1, nil // 第一个任务没有错误, 返回1
17         },
18         func(ctx context.Context) (interface{}, error) {
19             return nil, errors.New("failed") // 第二个任务返回一个错误
20         },
21         func(ctx context.Context) (interface{}, error) {
22             return 3, nil // 第三个任务没有错误, 返回3
23         },
24     )
25
26     fmt.Println(rs) // 输出子任务的结果
27     fmt.Println(errs) // 输出子任务的错误信息
28 }
```

Race 方法

Race 方法跟 All 方法类似，只不过，在使用 Race 方法的时候，只要一个异步函数执行没有错误，就立马返回，而不会返回所有的子任务信息。如果所有的子任务都没有成功，就会返回最后一个 error 信息。

Race 方法签名如下：

```
1 func Race(ctx context.Context, fns ...AsyncFunc) (interface{}, error)
```

[复制代码](#)

如果有一个正常的子任务的结果返回，Race 会把传入到其它子任务的 Context cancel 掉，这样子任务就可以中断自己的执行。

Race 的使用方法也跟 All 方法类似，我就不再举例子了，你可以把 All 方法的例子中的 All 替换成 Race 方式测试下。

Retry 方法

Retry 不是执行一组子任务，而是执行一个子任务。如果子任务执行失败，它会尝试一定的次数，如果一直不成功，就会返回失败错误，如果执行成功，它会立即返回。如果 `retires` 等于 0，它会永远尝试，直到成功。

[复制代码](#)

```
1 func Retry(ctx context.Context, retires int, fn AsyncFunc) (interface{}, error
```

再来看一个使用 `Retry` 的例子：

[复制代码](#)


```
1 package main
2
3 import (
4     "context"
5     "errors"
6     "fmt"
7     "github.com/vardius/gollback"
8     "time"
9 )
10
11 func main() {
12     ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
13     defer cancel()
14
15     // 尝试5次，或者超时返回
16     res, err := gollback.Retry(ctx, 5, func(ctx context.Context) (interface{}, e
17         return nil, errors.New("failed")
18     })
19
20     fmt.Println(res) // 输出结果
21     fmt.Println(err) // 输出错误信息
22 }
```

Hunch

[Hunch](#)提供的功能和 `gollback` 类似，不过它提供的方法更多，而且它提供的和 `gollback` 相应的方法，也有一些不同。我来一一介绍下。

它定义了执行子任务的函数，这和 `gollback` 的 `AsyncFunc` 是一样的，它的定义如下：


```
1 type Executable func(context.Context) (interface{}, error)
```

 复制代码

All 方法

All 方法的签名如下：

```
1 func All(parentCtx context.Context, execs ...Executable) ([]interface{}, error)
```


 复制代码

它会传入一组可执行的函数（子任务），返回子任务的执行结果。和 `gollback` 的 `All` 方法不一样的是，一旦一个子任务出现错误，它就会返回错误信息，执行结果（第一个返回参数）为 `nil`。

Take 方法

Take 方法的签名如下：

```
1 func Take(parentCtx context.Context, num int, execs ...Executable) ([]interface{}, error)
```


 复制代码

你可以指定 `num` 参数，只要有 `num` 个子任务正常执行完没有错误，这个方法就会返回这几个子任务的结果。一旦一个子任务出现错误，它就会返回错误信息，执行结果（第一个返回参数）为 `nil`。

Last 方法

Last 方法的签名如下：

```
1 func Last(parentCtx context.Context, num int, execs ...Executable) (interface{}, error)
```


 复制代码

它只返回最后 num 个正常执行的、没有错误的子任务的结果。一旦一个子任务出现错误，它就会返回错误信息，执行结果（第一个返回参数）为 nil。

比如 num 等于 1，那么，它只会返回最后一个无错的子任务的结果。

Retry 方法

Retry 方法的签名如下：


 复制代码

```
1 func Retry(parentCtx context.Context, retries int, fn Executable) (interface{}
```

它的功能和 gollback 的 Retry 方法的功能一样，如果子任务执行出错，就会不断尝试，直到成功或者是达到重试上限。如果达到重试上限，就会返回错误。如果 retries 等于 0，它会不断尝试。

Waterfall 方法

Waterfall 方法签名如下：

 复制代码

```
1 func Waterfall(parentCtx context.Context, execs ...ExecutableInSequence) (inte
```

它其实是一个 pipeline 的处理方式，所有的子任务都是串行执行的，前一个子任务的执行结果会被当作参数传给下一个子任务，直到所有的任务都完成，返回最后的执行结果。一旦一个子任务出现错误，它就会返回错误信息，执行结果（第一个返回参数）为 nil。

gollback 和 Hunch 是属于同一类的并发原语，对一组子任务的执行结果，可以选择一个结果或者多个结果，这也是现在热门的微服务常用的服务治理的方法。

schedgroup

接下来，我再介绍一个和时间相关的处理一组 goroutine 的并发原语 schedgroup。

🔗 [schedgroup](#) 是 Matt Layher 开发的 worker pool，可以指定任务在某个时间或者某个时间之后执行。Matt Layher 也是一个知名的 Gopher，经常在一些会议上分享一些他的 Go 开发经验，他在 GopherCon Europe 2020 大会上专门介绍了这个并发原语：

🔗 [schedgroup: a timer-based goroutine concurrency primitive](#)，课下你可以点开这个链接看一下，下面我来给你介绍一些重点。

这个并发原语包含的方法如下：

📄 复制代码

```
1 type Group
2 func New(ctx context.Context) *Group
3 func (g *Group) Delay(delay time.Duration, fn func())
4 func (g *Group) Schedule(when time.Time, fn func())
5 func (g *Group) Wait() error
```

我来介绍下这些方法。

先说 Delay 和 Schedule。

它们的功能其实是一样的，都是用来指定在某个时间或者之后执行一个函数。只不过，Delay 传入的是一个 time.Duration 参数，它会在 time.Now()+delay 之后执行函数，而 Schedule 可以指定明确的某个时间执行。

再来说说 Wait 方法。

这个方法调用会阻塞调用者，直到之前安排的所有子任务都执行完才返回。如果 Context 被取消，那么，Wait 方法会返回这个 cancel error。


在使用 Wait 方法的时候，有 2 点需要注意一下。

第一点是，如果调用了 Wait 方法，你就不能再调用它的 Delay 和 Schedule 方法，否则会 panic。

第二点是，Wait 方法只能调用一次，如果多次调用的话，就会 panic。

你可能认为，简单地使用 timer 就可以实现这个功能。其实，如果只有几个子任务，使用 timer 不是问题，但一旦有大量的子任务，而且还要能够 cancel，那么，使用 timer 的话，CPU 资源消耗就比较大了。所以，schedgroup 在实现的时候，就使用 container/heap，按照子任务的执行时间进行排序，这样可以避免使用大量的 timer，从而提高性能。

我们来看一个使用 schedgroup 的例子，下面代码会依次输出 1、2、3：

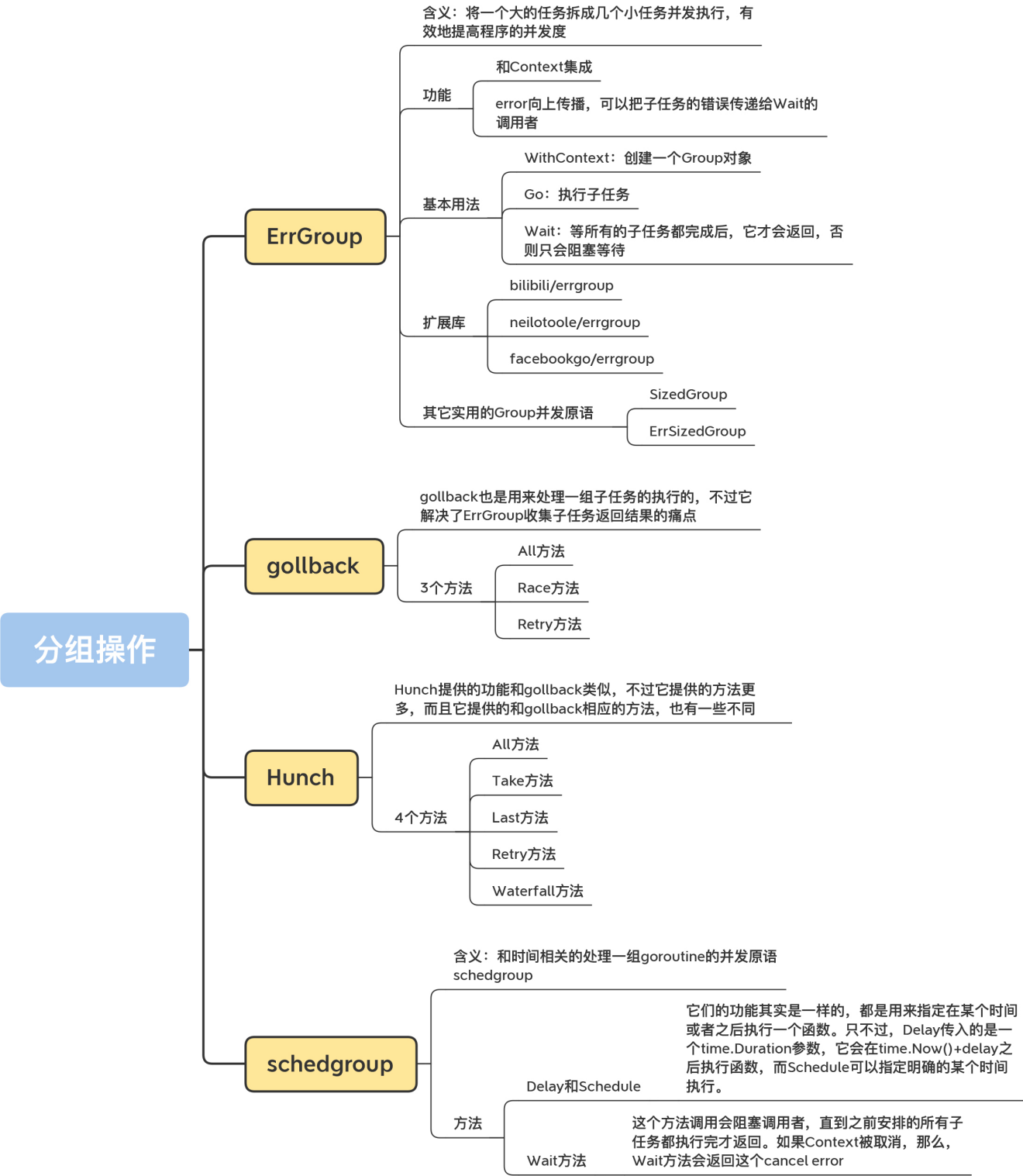
 复制代码

```
1 sg := schedgroup.New(context.Background())
2
3 // 设置子任务分别在100、200、300之后执行
4 for i := 0; i < 3; i++ {
5     n := i + 1
6     sg.Delay(time.Duration(n)*100*time.Millisecond, func() {
7         log.Println(n) //输出任务编号
8     })
9 }
10
11 // 等待所有的子任务都完成
12 if err := sg.Wait(); err != nil {
13     log.Fatalf("failed to wait: %v", err)
14 }
```

总结

这节课，我给你介绍了几种常见的处理一组子任务的并发原语，包括 ErrGroup、gollback、Hunch、schedgroup，等等。这些常见的业务场景共性处理方式的总结，你可以把它们加入到你的知识库中，等以后遇到相同的业务场景时，你就可以考虑使用这些并发原语。

当然，类似的并发原语还有别的，比如 [go-waitgroup](#) 等，而且，我相信还会有新的并发原语不断出现。所以，你不仅仅要掌握这些并发原语，而且还要通过学习这些并发原语，学会构造新的并发原语来处理应对你的特有场景，实现代码重用和业务逻辑简化。



思考题

这节课，我讲的官方扩展库 ErrGroup 没有实现可以取消子任务的功能，请你课下可以自己实现一个子任务可取消的 ErrGroup。

欢迎在留言区写下你的思考和答案，我们一起交流讨论。如果你觉得有所收获，也欢迎你 把今天的内容分享给你的朋友或同事。

提建议

Go 并发编程实战课

鸟窝带你攻克并发编程难题

晁岳攀 (鸟窝)

前微博技术专家

知名微服务框架 rpcx 的作者



新版升级：点击「👤请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 17 | SingleFlight 和 CyclicBarrier：请求合并和循环栅栏该怎么用？

下一篇 19 | 在分布式环境中，Leader选举、互斥锁和读写锁该如何实现？

精选留言 (6)

写留言



mbc

2020-11-28

老师，这种和用普通的go协程或者channel去完成一组任务的编写有啥不一样吗？好处是啥

作者回复：这一组并发原语只是提供了便利的方法，底层还是用gotoutinr和channel实现的

**伟伟**

2020-11-26

```
package main
```

```
import (  
    "context"  
    "fmt" ...
```

展开 ∨

**moooofly**

2020-11-24

```
```go
```

```
...
```

```
for i := 0; i < total; i++ { // 并发一万个goroutine执行子任务，理论上这些子任务都会加入到Group的待处理列表中
```

```
 go func() {...
```

展开 ∨

作者回复: 模拟一万个goroutine并发调用g.Go

**大漠胡萝卜**

2020-11-23

看了这篇文章还是收获巨大，以前不知道，也没用过。

首先是，ErrGroup，

官方的实现，bilibili/errgroup，neilotoole/errgroup，facebookgo/errgroup

```
...
```

展开 ∨

作者回复: 💎💎💎💎💎💎💎💎💎💎

**橙子888**

2020-11-20

打卡。

展开 ▾



皮卡丘

2020-11-20

cancel，失败的子任务可以 cancel 所有正在执行任务，这句话是不是有点绝对，ctx只是传递信号，如果cancel时，还有子任务的goroutine没有调度或者内部还没有运行到监听信号才会被cancel掉

作者回复: 所以说是“可以” 💎💎💎💎，这依赖子goroutine自己想不想cancel

