讲完Go并发控制，讲讲并发抑制

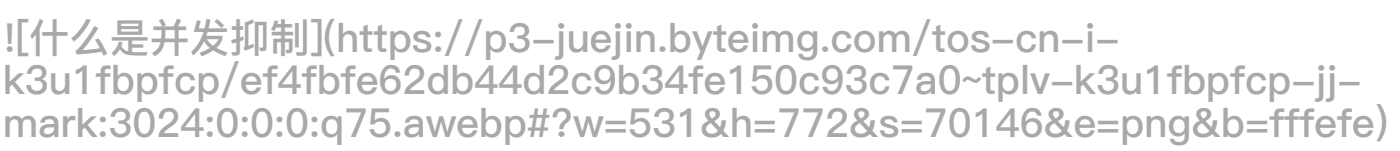
在一个例子，给你讲透典型的Go并发控制这篇文章中，我介绍了如何正确并优雅的并发执行多个goroutine，这篇文章来聊一聊另外一种并发控制的模型

为了更容易理解，我们沿用上一次的

已知有一个函数`search`，能够按照关键词执行搜索，`coSearch`能够批量并发查询。

让我们把目光定位到`search`上，`search`通过查询数据库或者调用其他api来完成搜索，这是一个相对耗时和消耗资源的操作。

当多个相同的关键词并发查询（调用`search`函数）时，我们希望只产生一次数据库调用（调用`query`），第一个查询未完成时后续的重复查询会等待，当第一个查询完成时则会与其他查询分享结果，这样一来虽然只执行了一次数据库调用但是所有查询都拿到了最终的结果。




```
package main

import (
”context”
”fmt”
”sync”
”time”

”golang.org/x/sync/errgroup”
)

func query(ctx context.Context, word string) (string, error) {
fmt.Println(”searching: ”, word)
time.Sleep(5 * time.Second)
```

```
return fmt.Sprintf("result: %s", word), nil // 模拟结果
}

// 实现search，在重复并发调用下仅执行一次query
// 其他并发共享这次query的结果
func search(ctx context.Context, word string) (string, error) {
    return query(ctx, word)
}

func coSearch(ctx context.Context, words []string) ([]string, error) {
g, ctx := errgroup.WithContext(ctx)
g.SetLimit(10)

results := make([]string, len(words))

for i, word := range words {
i, word := i, word

g.Go(func() error {
result, err := search(ctx, word)
if err != nil {
return err
}

results[i] = result
return nil
})
}

err := g.Wait()

return results, err
}

func main() {
words := []string{"Go","Go", "Go", "Rust", "PHP", "JavaScript", "Java"}
results, err := coSearch(context.Background(), words)
if err != nil {
fmt.Println(err)
return
}

fmt.Println(results)
}

```
```
```

好了，可以先暂停想想该如何实现`search`函数了

一步一步实现并发抑制

我们先假设所有查询关键词都一样，那么问题简化成并发执行`search`时，只在第一次`search`时调用`query`，其他的`search`并发调用等待并共享这次的查询结果

通过`waiting`变量，其他goroutine等待第一个goroutine数据库调用完成，那么如何让其他goroutine等待在这个位置呢?

```
func main() {
words := []string{”Go”, ”Go”, ”Go”, ”Go”, ”Go”}
results, err := coSearch(context.Background(), words)
if err != nil {
fmt.Println(err)
return
}

fmt.Println(results)
}

var (
waiting bool
resp    string
err     error
)

func search(ctx context.Context, word string) (string, error) {
  if waiting {
    // 等待resp, err被赋值，即第一个query完成后再返回
    // ...?
      return resp, err
  }

  waiting = true
  resp, err = query(ctx, word)
  waiting = false

  return resp, err
}
```

```
func query(ctx context.Context, word string) (string, error) {
fmt.Println("searching: ", word)
time.Sleep(5 * time.Second)

return fmt.Sprintf("result: %s", word), nil // 模拟结果
}
```

`sync.WaitGroup{}`并发控制

`sync.WaitGroup{}`是并发控制的核心，这里再次重申下用法

* 当新运行一个goroutine时，我们需要调用`wg.Add(1)`
* 当一个goroutine运行完成的时候，我们需要调用`wg.Done()`
* `wg.Wait()`让程序阻塞在此处，直到所有的goroutine运行完毕

利用 `sync.WaitGroup{}`便可实现上文代码中等待的效果

```
var (
wg      sync.WaitGroup
waiting bool
resp    string
err     error
)

func search(ctx context.Context, word string) (string, error) {
if waiting {
// 其他goroutine等待第一个goroutine执行完成
wg.Wait()
return resp, err
}

waiting = true

wg.Add(1)
resp, err = query(ctx, word)
    wg.Done() // 第一个goroutine执行完成

waiting = false
```

return resp, err
}

```

### 并发安全

当多个goroutine对同一个内存区域进行读写时，就会产生并发安全的问题，它会导致程序运行的结果不符合预期，而上文的程序并发的读写了`waiting`变量，需要给`waiting`变量加把锁。

释放锁的位置非常的有技巧，如果在在`wg.Add(1)`之前`mu.Unlock()`，可能`wg.Add(1)`还未来得执行其他goroutine已经执行了`wg.Wait()`，并获取到了错误的数据




```
var (
  wg   sync.WaitGroup
  mu   sync.Mutex
  waiting bool
  resp   string
  err   error
)

func search(ctx context.Context, word string) (string, error) {
mu.Lock()

if waiting {
mu.Unlock()
wg.Wait()
return resp, err
}

waiting = true

wg.Add(1)
    // 在wg.Add(1)之后释放锁，保证其他goroutine被wg.Wait()阻塞
mu.Unlock()
```
```

```
resp, err = query(ctx, word)
    wg.Done()

mu.Lock()
waiting = false
mu.Unlock()

return resp, err
}
```

完整版本

现在可以针对不同的关键词做区分了，使用一个map来代替原有的`waiting`，并将每一个关键词查询的WaitGroup和结果打包到map的value中

```
type call struct {
wg   sync.WaitGroup
resp string
err  error
}

var (
    mu sync.Mutex
m = make(map[string]*call)
)

func search(ctx context.Context, word string) (string, error) {
mu.Lock()

if c, ok := m[word]; ok {
mu.Unlock()
c.wg.Wait()
return c.resp, c.err
}

c := &call{}
m[word] = c

c.wg.Add(1)
// 在wg.Add(1)之后才释放锁，保证其他goroutine被wg.Wait()阻塞
mu.Unlock()
```

```
c.resp, c.err = query(ctx, word)
c.wg.Done()

mu.Lock()
delete(m, word)
mu.Unlock()

return c.resp, c.err
}
```

开源库 golang.org/x/sync/singleflight

上面一步一步教大家手搓了一个并发抑制的逻辑，我们的基本逻辑和开源库`golang.org/x/sync/singleflight`没有区别，只是singleflight内部实现更加严谨

直接使用singleflight非常简单的就可以实现我们的诉求

* `singleflight.Group` 创建一个需要并发控制的范围
* `Do`函数

+ 第一个参数接收一个key来判断否重复调用
+ 第二个参数为要执行的函数，函数可以返回正常值或者error
+ Do函数返回值除了闭包函数的返回值之外，还返回了此次返回值是否由其他goroutine共享

```
import (
”golang.org/x/sync/singleflight”
)

var g = new(singleflight.Group)

func search(ctx context.Context, word string) (string, error) {
resp, err, _ := g.Do(word, func() (interface{}, error) {
return query(ctx, word)
})

return resp.(string), err
```

}

```

### 错误处理

因为共享第一个goroutine的结果，因此如果第一次调用失败，那其他goroutine也都会失败

如果在某些场景下允许第一个调用失败后再次尝试调用该函数，那么可以通过调用`Forget`方法来忘记这个key

```
var g = new(singleflight.Group)

func search(ctx context.Context, word string) (string, error) {
resp, err, _ := g.Do(word, func() (interface{}, error) {
val, err := query(ctx, word)
// 当出错并且允许重试时
if err != nil && true {
g.Forget(word)
return "", err
}

return val, err
})

return resp.(string), err
}

```

### 超时控制

当使用Do函数时，如果query长时间未响应（这里假设qeury不具备超时能力），那么所有的goroutine都会被阻塞并等待，利用DoChan+select可以实现超时逻辑

```
var g = new(singleflight.Group)
```
```

```
func search(ctx context.Context, word string) (string, error) {
    ctx, cancel := context.WithTimeout(ctx, 2*time.Second)
defer cancel()

result := g.DoChan(word, func() (interface{}, error) {
return query(ctx, word)
})

select {
case r := <-result:
return r.Val.(string), r.Err
case <-ctx.Done():
return "", ctx.Err()
}
}
```

使用场景

预防缓存穿透

在高并发的状态下，一般会给热点数据设置缓存。但数据第一次访问或者缓存失效的状态下，如果直接去查询数据库，会给数据库造成极大压力，甚至直接打爆数据库。

以上各种分享中被反复提到的场景，但！注意！使用singleflight就一劳永逸了么，不是的，在大规模集群下可能有数百台机器，当处在高并发状态时，即使每台机器只发起一个请求，也足以打爆你的数据库！结合实际，搭配适当的缓存策略、数据预热、限流等手段才能避免潜在的风险。挖个坑，以后有机会聊聊这些问题

总结
--

本篇作为一个例子，给你讲透典型的Go并发控制的姊妹篇，讲述了另外一种并发控制模型，并介绍了开源库`golang.org/x/sync/singleflight`。

当由一个goroutine并发向下发展成多个goroutine时，使用`golang.org/x/sync/errgroup`

当多个goroutine并发向下抑制成一个goroutine时，使用
`golang.org/x/sync/singleflight`

公众号【**凉凉的知识库**】同步更新，欢迎获取最新最有用的知识

原文链接: https://juejin.cn/post/7380649024934576168