链滴

go 语言中 for 循环的并发安全问题

作者：zhengliwei

原文链接：https://ld246.com/article/1602923833058

来源网站：链滴

hello，大家好，欢迎来到银之庭。我是Z，一个普通的程序员。今天我们来看一个我在工作中刚发现go语言里for循环的一个问题。

1. 结论

先说结论，用尽可能简练的语言描述就是：**在go语言中，用for循环创建子协程，用errgroup管理这协程，并会向子协程中传递参数的情况下，有可能产生并发安全问题，需要复制一下循环中的临时变传给子协程，而不能直接把临时变量传过去。**

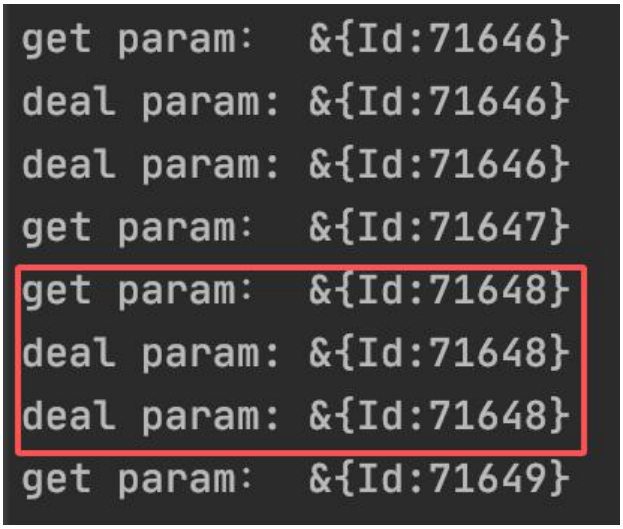
2. 现象

在某个go语言开发的工作项目中，我实现了个小需求，代码逻辑抽象出来大概是这样：

```
package main

import (
    "context"
    "fmt"
    "golang.org/x/sync/errgroup"
    "time"
)

type Param struct {
    Id int64
}

func test1() {
    // 传递参数的管道
    ch := make(chan *Param)

    // 创建errgroup，便于实现主协程等待子协程的功能
    gCtx, _ := context.WithTimeout(context.Background(), 1*time.Hour)
    g, gCtx := errgroup.WithContext(gCtx)

    // 起子协程往管道里不断传递参数
    g.Go(func() error {
        return send(ch)
    })

    // 在主协程里从管道中取出参数，然后对每个参数起一个子协程去处理
    for param := range ch {
        fmt.Printf("get param:  %+v\n", param)
        g.Go(func() error {
            return dealParam(param)
        })
    }

    // 等待所有子协程处理完毕
    if err := g.Wait(); err != nil {
        fmt.Printf("err: %+v\n", err)
    }

}
```

```
func send(ch chan *Param) error {
    for i := 1; ; i++ {
        p := Param{Id: int64(i)}
        ch <- &p
    }
}

func dealParam(param *Param) error {
    fmt.Printf("deal param: %+v\n", param)

    return nil
}
```

业务逻辑在代码中都已经注释出了，这里为了方便演示，直接死循环生产参数流，实际的业务代码里数流是有限的。在从管道中接到参数和处理参数的地方我打了日志，尝试运行一下后通过看这两处的志可以发现问题。我的一次运行的输入如下：



从上面红框里的内容可以很明显地看到，id为71648的参数从管道中获取了一次，但却被两个子协程理了，而且再仔细观察会发现，71646和71648都被处理了两次，而71647明明从管道中取出来了，一次都没被处理。可能有些小伙伴已经猜到原因了，这不是重复执行的问题，而是主协程和子协程并读写了某个变量，导致子协程读取到了被覆盖后的内容。

3. 本质

问题实际出在for循环这里：

```
for param := range ch {
    fmt.Printf("get param:  %+v\n", param)
    g.Go(func() error {
        return dealParam(param)
    })
}
```

根据现象可以想象到，go语言对for循环的实现是：

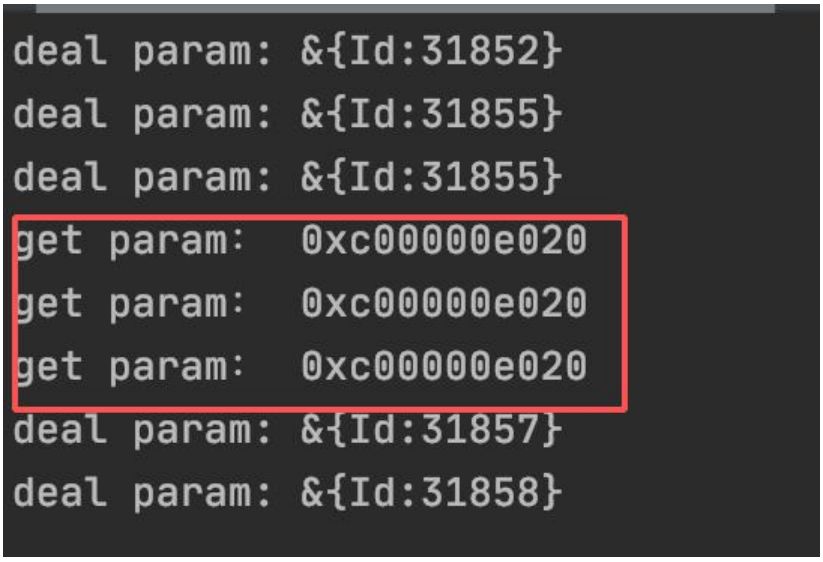
1. 开辟出一块内存空间，作为临时变量param；
2. 每次从管道中取到一个值，覆盖到param这块内存空间里。

而g.Go()函数内部是起了个子协程去运行我们传入的函数的，这样一来，在我们的函数运行时，para的值如果已经被后续的值覆盖了，那么前后两个子协程就拿到相同的param的值了，而之前那个没来及处理的param值，就没机会得到处理了。

下面我们来验证一下。主要是验证param这个临时变量的内存地址在每次循环中是否是同一个，可以打印语句中把&param的值打印出来，如下：

```
for param := range ch {
    fmt.Printf("get param:  %+v\n", &param)
    g.Go(func() error {
        return dealParam(param)
    })
}
```

我一次运行的结果如下：



可以看出，临时变量param确实是同一块内存地址，不断覆盖。这里额外说一点，我们的函数dealParam里接收到的param参数，是复制了一个传进来的，并不是原来的内存地址。

4. 解决方案

知道了原因，解决就比较简单了，只要在g.Go()前，把param临时变量复制一份出来，把复制的变量到子协程里就行了，因为复制的变量不会被覆盖，所以就避免了这个问题，代码如下：

```
for param := range ch {
    // 复制param，防止param的值被后续的新值覆盖
    tempParam := param
    fmt.Printf("get param:  %+v\n", tempParam)
    g.Go(func() error {
        return dealParam(tempParam)
    })
}
```

再次运行，就可以发现没有上面的重复处理参数和漏处理参数的问题了：

```
get param:  &{Id:22606}
deal param: &{Id:22606}
deal param: &{Id:22605}
get param:  &{Id:22607}
get param:  &{Id:22608}
deal param: &{Id:22608}
deal param: &{Id:22607}
get param:  &{Id:22609}
get param:  &{Id:22610}
```

这里还有个小细节：管道里传递的是参数对象的指针，而不是参数对象本身，这样在复制param时，可以值复制一个指针，而不是复制一个对象，减少复制的开销。

当然，只有使用errgroup时才会有这个问题（下面会讲到），所以也可以直接使用go关键字起子协，不用errgroup，不过就需要自己做协程间的同步了。

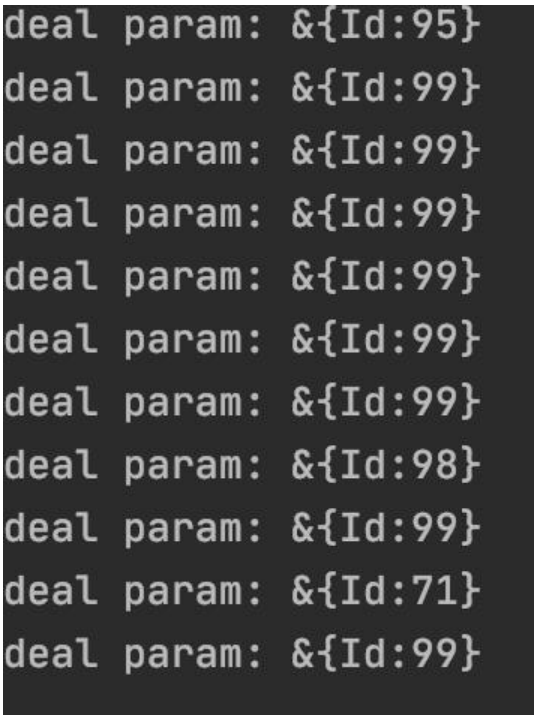
5. 扩展问题

5.1 普通循环是否有问题

上面的for循环是从管道里取数据，如果直接对一个列表进行遍历，还会不会有这种情况呢？我们来实一下，代码如下：

```
func test2() {
    params := []*Param{}
    for i := 0; i < 100; i++ {
        params = append(params, &Param{Id: int64(i)})
    }

    gCtx, _ := context.WithTimeout(context.Background(), 1*time.Hour)
    g, gCtx := errgroup.WithContext(gCtx)

    for _, param := range params {
        g.Go(func() error {
            return dealParam(param)
        })
    }

    if err := g.Wait(); err != nil {
        fmt.Printf("err: %+v\n", err)
    }
}
```

我一次运行的结果如下：



可以发现，问题更加严重，因为主协程执行更快，会有更多子协程拿到同样的param的值。所以，可确定，**这是fo语言里for循环实现的特性，和管道无关。只要是for循环创建协程，并传入了变量，就要注意变量并发读写的问题。**

5.2 go关键字是否有问题

上面的例子中，我们都使用了errgroup的API，如果不用errgroup，直接go出子协程呢？还会有这种题吗？我们来实验一下，代码如下：

```
func test3() {
    params := []*Param{}
    for i := 0; i < 100; i++ {
        params = append(params, &Param{Id: int64(i)})
    }

    for _, param := range params {
        go dealParam(param)
    }

    time.Sleep(10*time.Second)
}
```

这里为了方便，直接在主协程里休眠了10秒，确保子协程都执行完，实际上应该用锁来实现协程间同。运行之后会发现，没有上面的问题，可以合理猜测：**使用go dealParam(param)这句代码时，复一个新参数传给dealParam()的动作也是主协程里做的，所以没有并发读写param的问题，而用errgoup的APIg.Go()时，复制一个新参数传给dealParam()的动作是在子协程里做的，所以会产生并发写param的问题。**

5.3 Java是否有同样的问题

我写Java的时间更长，但好像从来没注意过这种问题，是有过这样的bug但我没注意，还是Java里就存在这样的问题呢？我们来验证一下，代码如下：

```
import lombok.Builder;
import lombok.Data;

import java.util.concurrent.CountDownLatch;

public class Test {

    public static void main(String[] args) throws InterruptedException {
        Param[] params = new Param[1000];
        for (int i = 0; i < 1000; i++) {
            params[i] = Param.builder().id(i).build();
        }

        CountDownLatch countDownLatch = new CountDownLatch(1000);
        for (Param param : params) {
        // System.identityHashCode可以打印对象的内存地址
            System.out.println("get param: " + System.identityHashCode(param));
            Worker worker = new Worker(param, countDownLatch);
            new Thread(worker).start();
        }

        countDownLatch.await();
    }
}

@Data
@Builder
class Param {
    private Integer id;
}

class Worker implements Runnable {

    private Param param;

    private CountDownLatch countDownLatch;

    public Worker(Param param, CountDownLatch countDownLatch) {
        this.param = param;
        this.countDownLatch = countDownLatch;
    }

    @Override
    public void run() {
        System.out.println("deal param: " + param.toString());
        countDownLatch.countDown();
    }
}
```

大家如果运行一下会发现，并没有上面的问题，而且打印出来的param临时变量的地址并不是同一个
可以合理猜测：**Java的for循环并不会创建一个临时变量接收循环到的值，而是直接用原对象做处理。**