一、Context的作用（What）

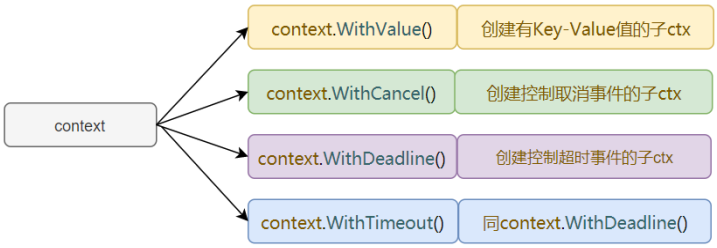
Context主要表示上下文，其控制一个请求的生命周期。在并发程序中，超时、定时、取消、或者一些异常操作，通常需要中断当前任务的后续操作。

引入Context的原因主要是我们不能从外部终止正在执行的goroutine。当然我们也可以使用channel+select方法，但是当多个goroutine出现的时候，就得维护大量的协程与channel的关系。一句话，context用来解决goroutine之间的退出通知以及元元素传递的功能。

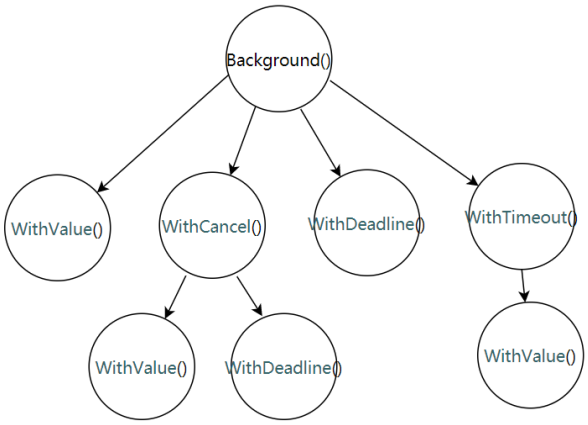
Context可以在多个goroutine控制上下文，收到控制信号的时候，可以终止goroutine树，也就是上层任务中断后，其子任务也将被取消，且不影响同级任务以及上级任务。每次创建一个goroutine，要么将原有的Context传递给goroutine，要么创建一个子Context并传递给Goroutine

Context在gRPC收发消息最为常用(因为每一个RPC调用都应该有超时退出的能力)，gRpc使用Context来终止某个请求产生的goroutine树。同时我们也要养成关闭Context的习惯，特别是超时的Context，创建子Context后紧接着defer cancel()。

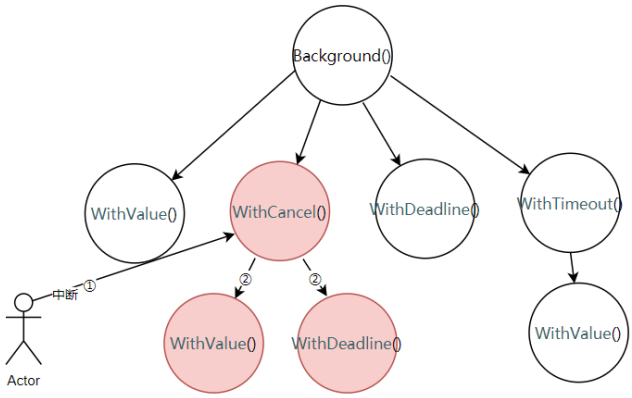
Context共有4个常用接口：可以通过第二部分的几个例子学习怎么使用这几个接口



这里每增加一个context.WithXXX就增加一个子ctx，形成一个树状图



若其中一个节点中断，不会影响同级以及上级任务。如下图：



当收到一个中断后，基于该Context节点所派生的子Context也都会被关闭，并且会将自己从父Context中移除，停止和它相关的timer。如上图。

Context是协程安全的，所以在WithXXX时无需加锁，即使是多个协程访问也可以保证资源安全。

二、Context几个例子(When)

1、先举个小栗子，使用Context来控制一个goroutine。发送停止信号通知协程退出

```go
// 控制一个goroutine
func oneGoContext() {
    ctx, cancel := context.WithCancel(context.Background())
    go func(ctx context.Context) {
        for {
            select {
            case <- ctx.Done(): // 接收到停止信号，立即退出
                fmt.Println("goroutine stop...")
                return
            default:
                fmt.Println("goroutine running...")
                time.Sleep(1*time.Second)
            }
        }
    }(ctx)
    // 10s 后发送停止信号
    time.Sleep(10*time.Second)
    fmt.Println("call goroutine stop !!! ")
    cancel() // 发送停止信号
    time.Sleep(5 * time.Second)
}
```

2、使用Context控制多个goroutine，主动发出停止信号，所有ctx相关的协程取消。

```go
func mulGoContext() {
    ctx, cancel := context.WithCancel(context.Background())
    // 多个协程
    go procCtx(ctx,"Test1")
    go procCtx(ctx,"Test2")
    go procCtx(ctx,"Test3")

    time.Sleep(5*time.Second)
    fmt.Println("Call goroutines stop")
    cancel()
    time.Sleep(5*time.Second)
```

```
}

func procCtx(ctx context.Context, str string) {
    for {
        select {
        case <- ctx.Done(): // 接收到停止信号，立即退出
            fmt.Printf("%s out...\n",str)
            return
        default:
            fmt.Printf("%s running...\n",str)
            time.Sleep(1*time.Second)
        }
    }
}
/*
    Test1 running...
    Test2 running...
    Test3 running...
    Test1 running...
    Test3 running...
    Test2 running...
    Test3 running...
    Test2 running...
    Test1 running...
    Test2 running...
    Test3 running...
    Test1 running...
    Test2 running...
    Test3 running...
    Test1 running...
    Call goroutines stop
    Test2 out...
    Test3 out...
    Test1 out...
*/
```

可以看到，三个goroutine使用同一个ctx进跟踪监控，当ctx关闭的时候，三个goroutine会随着关闭。使用cancel()通知ctx关闭释放当前的goroutine，就这样控制了多个goroutine的执行。

3、创建子context并附加一个Key-Value键值对，在子context贯穿所有的函数都可以使用。这里的Key-Value值只能查询自己和父节点的值，不能查询兄弟节点的值。若找不到Key对应的值，会递归查找父ctx的值。

```
// 附加一个Key-value值
var key string = "KEY"
func valueGoContext() {
    ctx, cancel := context.WithCancel(context.Background())
    ctx = context.WithValue(ctx,key,"Test") // 在子context附加一个键值对
    go procCtxTest(ctx)
    // 10s 后发送停止信号
    time.Sleep(10*time.Second)
    fmt.Println("call goroutine stop !!! ")
    cancel() // 发送停止信号
    time.Sleep(5 * time.Second)
}

func procCtxTest(ctx context.Context) {
```

```
    for {
        select {
        case <- ctx.Done():
            // 通过Value方法获取value值
            fmt.Println(ctx.Value(key),"goroutine out...")
            return
        default:
            fmt.Println(ctx.Value(key),"goroutine running...")
            time.Sleep(1*time.Second)
        }
    }
}
```

```
// 附加多个Key-value值
var key string = "KEY"
var key1 string = "KEY1"
func testMulValueContext() {

    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()
    // 对于ctx生成2个Key-Value
    ctx = context.WithValue(ctx,key,"valueOne") // ctx->chaild1
    ctx2 := context.WithValue(ctx,key1,"valueTwo") // ctx->chaild2

    var deadline time.Time = time.Now().Add(5 * time.Second)
    ctxde,cancel:= context.WithDeadline(ctx,deadline) // ctx->chaild1-
>chaild
    defer cancel()

    go procCtxTest(1,ctxde)// ctx->chaild1->chaild
    go procCtxTest(2,ctx2)// ctx->chaild2

    time.Sleep(10*time.Second)
}

func procCtxTest(num int,ctx context.Context) {
    for {
        select {
        case <- ctx.Done():
            fmt.Println(num,"goroutine out...")
            return
        default:
            fmt.Println(num,"goroutine running...")
            time.Sleep(1*time.Second)
        }
    }
}

/*
2 goroutine running...
1 goroutine running...
1 goroutine running...
2 goroutine running...
1 goroutine running...
2 goroutine running...
2 goroutine running...
1 goroutine running...
```

```
1 goroutine running...
2 goroutine running...
1 goroutine out...   // 1已经退出 但是不影响2的执行
2 goroutine running...
2 goroutine running...
2 goroutine running...
2 goroutine running...
2 goroutine running...
*/
```

三、Context的设计原理（源码分析）（How）

现在你一定很好奇，Context到底是怎么实现超时和链式关闭的。那我们就来分析源码吧！

首先看下Context的数据结构：

```go
type Context interface {
    // 返回超时时间，
    Deadline() (deadline time.Time, ok bool)
    // Done若ctx被取消的时候，这个通道会关闭，对应的goroutine树结束并返回
    Done() <-chan struct{}
    // Err表示 取消的原因
    Err() error
    // 返回Key对应的Value值，goroutine共享的一些数据，获得数据是协程安全的
    Value(key interface{}) interface{}
}
```

值得注意的是Context是协程安全的，这样也就是可以把创建的子ctx让多个协程使用，且可以安全的访问共享数据

我们主要看一下其中一个Withcancel() 控制取消函数

```go
func WithCancel(parent Context) (ctx Context, cancel CancelFunc) {
    c := newCancelCtx(parent) // 创建cancelCtx类型的子ctx
    propagateCancel(parent, &c) // 将子ctx与父节点进行关系连接
    return &c, func() { c.cancel(true, Canceled) }  // 收到控制信号，执行cancel操作
}
```

再看下关键的propagateCancel函数

```go
// 在 parent 和 child 之间同步取消和结束的信号，保证在 parent 被取消时，child 也会收到对应的信号，不会出现状态不一致的情况
func propagateCancel(parent Context, child canceler) {
    done := parent.Done()
    // 父ctx不会触发取消信号（WithValue），例如定时的父ctx会触发取消信号，done!=nil
    if done == nil {
        return // parent is never canceled
    }

    select {
    case <-done: // 父ctx有取消信号,取消子节点
        // parent is already canceled
        child.cancel(false, parent.Err())
        return
    default:
```

```
    }
    // 确定parent最内层的cancel是否是内部实现的cancelCtx
    if p, ok := parentCancelCtx(parent); ok {
        p.mu.Lock()
        if p.err != nil {
            // parent has already been canceled
            child.cancel(false, p.err)
        } else {
            if p.children == nil {
                p.children = make(map[canceler]struct{})
            }
            // 将自己加入父节点的children，等到收到父ctx取消信号的时候可以取消
            p.children[child] = struct{}{}
        }
        p.mu.Unlock()
    } else {
        // 如果不是，开协程监听父节点是否有取消信号，若父节点有取消信号，则取消子节点
        atomic.AddInt32(&goroutines, +1)
        go func() {
            select {
            case <-parent.Done():
                child.cancel(false, parent.Err())
            case <-child.Done():
            }
        }()
    }
}
```

最后的取消操作

```
func (c *cancelCtx) cancel(removeFromParent bool, err error) {
    if err == nil {
        panic("context: internal error: missing cancel error")
    }
    c.mu.Lock()
    if c.err != nil {
        c.mu.Unlock()
        return // already canceled
    }
    c.err = err // 取消原因
    if c.done == nil {
        c.done = closedchan // 空的
    } else {
        close(c.done) // 关闭通道
    }
    // 将子节点依次取消
    // 依次遍历c.children，每个child分别cancel
    for child := range c.children {
        // NOTE: acquiring the child's lock while holding parent's lock.
        child.cancel(false, err)
    }
    c.children = nil
    c.mu.Unlock()

    if removeFromParent {
        // 移除子节点
        removeChild(c.Context, c)
```

```
28      }
29  }
```

这就是cancel主要的流程

主要的思想就是创建ctx与父ctx建立联系，使得子ctx能够监听到父ctx的消息，并将子ctx存在父ctx的children的map中，供后续删除节点的时候使用，这样就无需再创建新的功能来删除子ctx。取消信号可能会来自自己主动cancel或者父ctx取消信号。若监听到取消信号，遍历children_map，取消所有的子节点。

对于超时操作，取消信号除了自己主动cancel和父ctx取消信号，还有超时的取消信号，其他的操作基本类似。