Go Context介绍


Dakini_Wind 关注 IP属地: 河北
0.106 2021.04.27 19:43:29 字数 795 阅读 2,840

`context`包是Go 语言中用来设置截止日期、同步信号，传递请求相关值的结构体，是开发常用的并发控制技术。
与`WaitGroup`的不同在于`context`可以控制多级的`goroutine`。

1. 接口定义

```
type Context interface {
    Deadline()  (deadline time.Time, ok bool)

    Done() <- chan struct{}

    Err() error

    Value(key interface{}) interface{}
}
```

> Deadline(): 工作的截止时间，没有设置`deadline`则`ok==false`。
> Done(): 需要在`select-case`语句中使用（`case <-context.Done():`）。当`context`被关闭后，`Done()`返回一个被关闭的通道（关闭的通道依然是可以读的，所以`goroutine`可以收到关闭请求）；当`context`还未关闭时，`Done()`返回`nil`。
> Err(): 描述`context`关闭的原因，其原因由`context`实现控制。例如：因`deadline`关闭：`context deadline exceeded`；因主动关闭：`context canceled`。没有关闭时，返回`nil`。
> Value(): 特别的用于一种`context`：不用于控制呈树状分布的`goroutine`，而是用于在树状分布的`goroutine`之间传递信息。
> `Value()`方法根据`key`值查询`map`中的`Value`。

2. 使用

一个案例来展示`context`的使用，它做了2件事：1. 创建过期时间为1s的上下文。 2. 将`context`传入`handle`函数中，函数使用500ms的时间处理请求。

```
func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 1*time.Second)
    defer cancel()

    go handle(ctx, 500*time.Millisecond)
    select {
    case <-ctx.Done():
        fmt.Println("main", ctx.Err())
    }
}

func handle(ctx context.Context, duration time.Duration) {
    select {
    case <-ctx.Done():
        fmt.Println("handle", ctx.Err())
    case <-time.After(duration):
        fmt.Println("process request with", duration)
    }
}
```

output:

```
process request with 500ms
main context deadline exceeded
```

分析: `context` 过期时间为1s，处理时间为0.5秒(`select` 中的过期时间)，函数有足够的时间完成处理，也就是 `<-time.After(duration):` 会在 `<-ctx.Done()` 之前完成，故输出 `process request with 500ms` 。再过0.5s， `<-ctx.Done()` 完成，这时候输出 `main context deadline exceeded` 。

倘若，代码中的``改为 `400*time.Millisecond` ，会输出什么呢?
A:

```
main context deadline exceeded
```

B:

```
main context deadline exceeded
handle context deadline exceeded
```

C:

```
process request with 500ms
main context deadline exceeded
```

D:

```
process request with 500ms
```

E:

```
handle context deadline exceeded
main context deadline exceeded
```

答案是：A、B、E
可能出现这3种，而不是1种的原因是和调度器有关。

`context` 的一些方法:

- **默认上下文**: 以下两个方法都会返回预先初始化好的私有变量 `background` 和 `todo` ，它们会在同一个 Go 程序中被复用。这两个私有变量都是通过 `new(emptyCtx)` 语句初始化的，它们是指向私有结构体 `context.emptyCtx` 的指针，这是最简单、最常用的上下文类型。
  - `context.Background()` : 是上下文的默认值，所有其他的上下文都应该从它衍生出来。
  - `context.TODO()` : 应该仅在不确定应该使用哪种上下文时使用。
- **取消信号**：前两个创建的是 `context.WithCancel` ，最后一个创建的是 `context.timerCtx` 。
  - `context.WithCancel()` : 由 `context.Context()` 衍生出的特殊的子上下文。一旦它的返回函数被执行，其所有子 `context` 将都会被返回。
  - `context.WithDeadline()` : 在某个时间点进行返回。
  - `context.WithTimeout()` : 某个时间段过后进行返回。
- **传值方法**:

- `context.WithValue()`：从父上下文中创建一个子上下文，传值的子上下文使用 `context.valueCtx`。

  需要注意的是这个方法是递归的根据`Key`来获取`Value`的。

```
func (c *valueCtx) Value(key interface{}) interface{} {
    if c.key == key {
        return c.val
    }
    return c.Context.Value(key)
}
```

go context

tracy_668 关注 IP属地: 北京

0.785 2022.05.18 07:20:09 字数 2,224 阅读 746

相信大家在日常工作开发中一定会看到这样的代码：

```
func a1(ctx context ...){
  b1(ctx)
}
func b1(ctx context ...){
  c1(ctx)
}
func c1(ctx context ...)
```

context被当作第一个参数（官方建议），并且不断透传下去，基本一个项目代码中到处都是context，但是你们真的知道它有何作用吗以及它是如何起作用的吗？我记得我第一次接触context时，同事都说这个用来做并发控制的，可以设置超时时间，超时就会取消往下执行，快速返回，我就单纯的认为只要函数中带着context参数往下传递就可以做到超时取消，快速返回。相信大多数初学者也都是和我一个想法，其实这是一个错误的思想，其取消机制采用的也是通知机制，单纯的透传并不会起作用，比如你这样写代码：

```go
func main()  {
 ctx,cancel := context.WithTimeout(context.Background(),10 * time.Second)

 go Monitor(ctx)
 cancel()

 time.Sleep(20 * time.Second)
}

func Monitor(ctx context.Context)  {
 for {
  fmt.Print("monitor")
 }
}
```

即使context透传下去了，没有监听取消信号也是不起任何作用的。所以了解context的使用还是很有必要的，本文就先从使用开始，逐步解析Go语言的context包，下面我们就开始喽！！！

context 包的起源与作用

看官方博客我们可以知道context包是在go1.7版本中引入到标准库中的：

context可以用来在goroutine之间传递上下文信息，相同的context可以传递给运行在不同goroutine中的函数，上下文对于多个goroutine同时使用是安全的，context包定义了上下文类型，可以使用background、TODO创建一个上下文，在函数调用链之间传播context，也可以使用WithDeadline、WithTimeout、WithCancel 或 WithValue 创建的修改副本替换它，听起来有点绕，其实总结起就是一句话：*context的作用就是在不同的goroutine之间同步请求特定的数据、取消信号以及处理请求的截止日期。*

目前我们常用的一些库都是支持context的，例如gin、database/sql等库都是支持context的，这样更方便我们做并发控制了，只要在服务器入口创建一个context上下文，不断透传下去即可。

context 的使用

创建context

context包主要提供了两种方式创建context:

- context.Backgroud()
- context.TODO()

这两个函数其实只是互为别名，没有差别，官方给的定义是：

- context.Background 是上下文的默认值，所有其他的上下文都应该从它衍生（Derived）出来。
- context.TODO 应该只在不确定应该使用哪种上下文时使用；

所以在大多数情况下，我们都使用context.Background作为起始的上下文向下传递。

上面的两种方式是创建根context，不具备任何功能，具体实践还是要依靠context包提供的With系列函数来进行派生：

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
func WithValue(parent Context, key, val interface{}) Context
```

这四个函数都要基于父Context衍生，通过这些函数，就创建了一颗Context树，树的每个节点都可以有任意多个子节点，节点层级可以有任意多个，画个图表示一下：

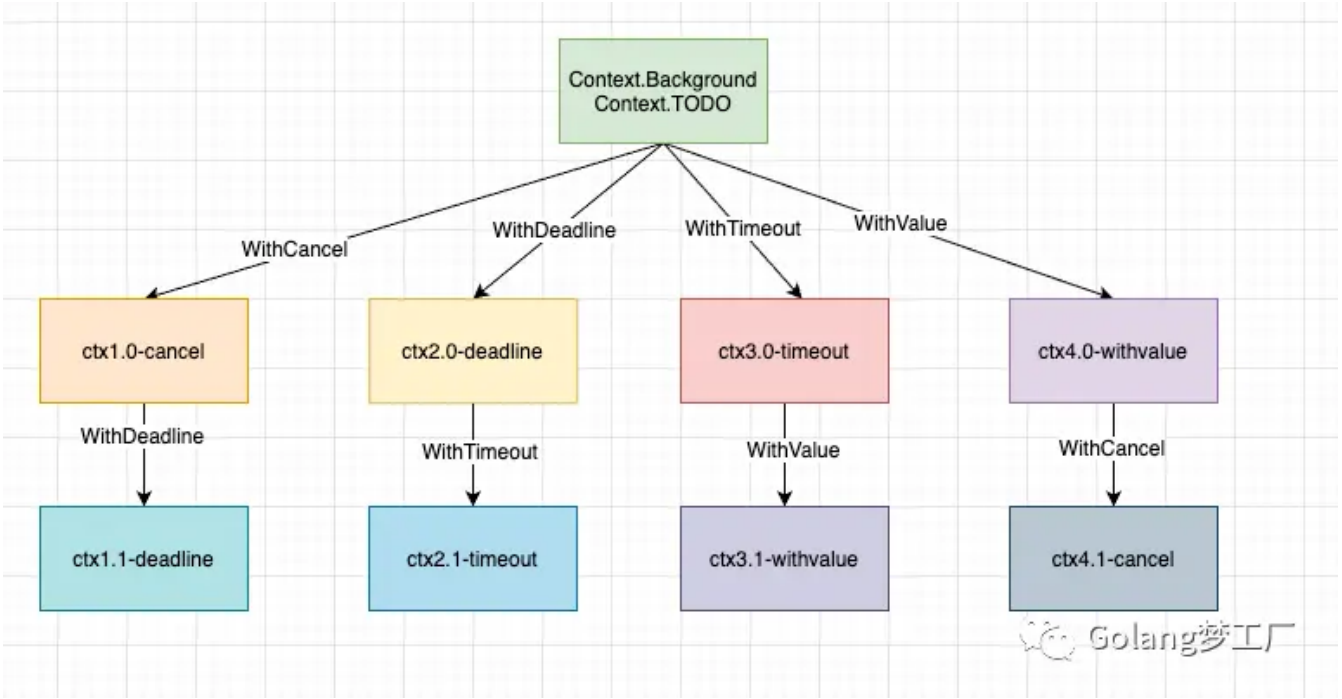

image.png

基于一个父Context可以随意衍生，其实这就是一个Context树，树的每个节点都可以有任意多个子节点，节点层级可以有任意多个，每个子节点都依赖于其父节点，例如上图，我们可以基于Context.Background衍生出四个子context：ctx1.0-cancel、ctx2.0-deadline、ctx3.0-timeout、ctx4.0-withvalue，这四个子context还可以作为父context继续向下衍生，即使其中ctx1.0-cancel 节点取消了，也不影响其他三个父节点分支。

创建context方法和context的衍生方法就这些，下面我们就一个一个来看一下他们如何被使用。

WithValue携带数据

我们日常在业务开发中都希望能有一个trace_id能串联所有的日志，这就需要我们打印日志时能够获取到这个trace_id，在python中我们可以用gevent.local来传递，在java中我们可以用ThreadLocal来传递，在Go语言中我们就可以使用Context来传递，通过使用WithValue来创建一个携带trace_id的context，然后不断透传下去，打印日志时输出即可，来看使用例子：

我们基于context.Background创建一个携带trace_id的ctx，然后通过context树一起传递，从中派生的任何context都会获取此值，我们最后打印日志的时候就可以从ctx中取值输出到日志中。目前一些RPC框架都是支持了Context，所以trace_id的向下传递就更方便了。

在使用withVaule时要注意四个事项：

不建议使用context值传递关键参数，关键参数应该显示的声明出来，不应该隐式处理，context中最好是携带签名、trace_id这类值。

因为携带value也是key、value的形式，为了避免context因多个包同时使用context而带来冲突，key建议采用内置类型。

上面的例子我们获取trace_id是直接从当前ctx获取的，实际我们也可以获取父context中的value，在获取键值对是，我们先从当前context中查找，没有找到会在从父context中查找该键对应的值直到在某个父context中返回 nil 或者查找到对应的值。

context传递的数据中key、value都是interface类型，这种类型编译期无法确定类型，所以不是很安全，所以在类型断言时别忘了保证程序的健壮性。

超时控制

通常健壮的程序都是要设置超时时间的，避免因为服务端长时间响应消耗资源，所以一些web框架或rpc框架都会采用withTimeout或者withDeadline来做超时控制，当一次请求到达我们设置的超时时间，就会及时取消，不在往下执行。withTimeout和withDeadline作用是一样的，就是传递的时间参数不同而已，他们都会通过传入的时间来自动取消Context，这里要注意的是他们都会返回一个cancelFunc方法，通过调用这个方法可以达到提前进行取消，不过在使用的过程还是建议在自动取消后也调用cancelFunc去停止定时减少不必要的资源浪费。

withTimeout、WithDeadline不同在于WithTimeout将持续时间作为参数输入而不是时间对象，这两个方法使用哪个都是一样的，看业务场景和个人习惯了，因为本质withTimout内部也是调用的WithDeadline。

现在我们就举个例子来试用一下超时控制，现在我们就模拟一个请求写两个例子：

达到超时时间终止接下来的执行

```
package main

import (
    "context"
    "fmt"
    "time"
)

func main()  {
    HttpHandler()
}

func NewContextWithTimeout() (context.Context,context.CancelFunc) {
    return context.WithTimeout(context.Background(), 3 * time.Second)
```

```
}

func HttpHandler()  {
    ctx, cancel := NewContextWithTimeout()
    defer cancel()
    deal(ctx)
}

func deal(ctx context.Context)  {
    for i:=0; i< 10; i++ {
        time.Sleep(1*time.Second)
        select {
        case <- ctx.Done():
            fmt.Println(ctx.Err())
            return
        default:
            fmt.Printf("deal time is %d\n", i)
        }
    }
}

```

没有达到超时时间终止接下来的执行

使用起来还是比较容易的，既可以超时自动取消，又可以手动控制取消。这里大家要记的一个坑，就是我们往从请求入口透传的调用链路中的context是携带超时时间的，如果我们想在其中单独开一个goroutine去处理其他的事情并且不会随着请求结束后而被取消的话，那么传递的context要基于context.Background或者context.TODO重新衍生一个传递，否决就会和预期不符合了，可以看一下我之前的一篇踩坑文章：context使用不当引发的一个bug。

withCancel取消控制

日常业务开发中我们往往为了完成一个复杂的需求会开多个gouroutine去做一些事情，这就导致我们会在一次请求中开了多个goroutine确无法控制他们，这时我们就可以使用withCancel来衍生一个context传递到不同的goroutine中，当我想让这些goroutine停止运行，就可以调用cancel来进行取消。

我们使用withCancel创建一个基于Background的ctx，然后启动一个讲话程序，每隔1s说一话，main函数在10s后执行cancel，那么speak检测到取消信号就会退出。