Go语言 标准库 Context包

`context.Context`是一个非常抽象的概念，中文翻译为”上下文“，可看做为`goroutine`的上下文。`Context`是线程安全的，所以可以在多个`goroutine`之间传递上下文信息，包括信号、超时时间、`K-V`键值对等，同时它也可以用作并发控制。

Goroutine和Channel

在理解context包之前，应该首先熟悉Goroutine和Channel，能加深对context的理解。

Goroutine

Goroutine是一个轻量级的执行线程，多个Goroutine比一个线程轻量，所以管理Goroutine消耗的资源相对更少。Goroutine是Go中最基本的执行单元，每一个Go程序至少有一个Goroutine：主Goroutine。程序启动时会自动创建。为了能更好的理解Goroutine，先来看一看线程Thread与协程Coroutine的概念。

线程（Thread）

线程是一种轻量级进程，是CPU调度的最小单位。一个标准的线程由线程ID，当前指令指针（PC），寄存器集合和堆栈组成。线程是进程中的一个实体，是被系统独立调度和分派的基本单位，线程自己不拥有系统资源，只拥有一点在运行中必不可少的资源，但它可与同属于一个进程的其他线程共享进程所拥有的全部资源。线程拥有自己独立的栈和共享的堆，共享堆，不共享栈。

线程的切换一般由操作系统调度。

协程（Coroutine）

协程又称为微线程，与子例程一样，协程也是一种程序组建，相对子例程而言，协程更为灵活，但在实践中使用没有子例程那样广泛。和线程类似，共享堆，不共享栈，**协程的切换一般由程序员在代码中显式控制**。他避免了上下文切换的额外耗费，兼顾了多线程的优点，简化了高并发程序的复杂。

Goroutine和其他语言的协程（coroutine）在使用方式上类似，但从字面意义上来看不同（一个是Goroutine，一个是coroutine），再就是协程是一种协作任务控制机制，在最简单的意义上，协程不是并发的，而Goroutine支持并发的。因此Goroutine可以理解为一种Go语言的协程。同时，Gorotine可以运行在一个或多个线程上。

使用Goroutine示例

```
 func Hello()  {
    fmt.Println("hello everybody , I'm baozi")
}

func main()  {
    go Hello()
    fmt.Println("Golang-Gorontine Example")
}
#结果
Golang-Gorontine Example
```

从执行结果上，直观的看，我们的程序似乎没有执行Goroutine的Hello方法。出现这个问题的原因是我们启动的主Goroutine在main执行完就退出了，所以为了main等待Hello-Goroutine执行完，就需要一些方法，让Hello-Goroutine告诉main执行完了，这里就需要通道Channel了。

Channel

Channel就是多个Goroutine 之间的沟通渠道。当我们想要将结果或错误，或任何其他类型的信息从一个 goroutine 传递到另一个 goroutine 时就可以使用通道。通道是有类型的，可以是 int 类型的通道接收整数或错误类型的接收错误等。

假设有个 int 类型的通道 ch，如果想发一些信息到这个通道，语法是 ch <- 1，如果想从这个通道接收一些信息，语法就是 var := <-ch。这将从这个通道接收并存储值到 var 变量。

通过改善使用Goroutine示例的代码片段，证明通道的使用确保了 goroutine 执行完成并将值返回给 main 。

```
func Hello(ch chan int)  {
    fmt.Println("hello everybody , I'm baozi")
    ch <- 1
}

func main()  {
    ch := make(chan int)
    go Hello(ch)
    <-ch
    fmt.Println("Golang-Gorontine Example")
}
#结果
hello everybody , I'm baozi
Golang-Gorontine Example
```

这里我们使用通道进行等待，这样main就会等待Hello-Goroutine执行完。熟悉了Goroutine、Channel的概念了，就很好理解Context了。

Context应用场景

熟悉了Goroutine、Channel的概念，先看一个请求服务场景示例：

```
func main()  {
    http.HandleFunc("/", SayHello) // 设置访问的路由
    log.Fatalln(http.ListenAndServe(":8080",nil))
}

func SayHello(writer http.ResponseWriter, request *http.Request)  {
    fmt.Println(&request)
    writer.Write([]byte("Hi, New Request Comes"))
}
```

上述代码，每次请求，Handler会创建一个Goroutine为其提供服务；
如果连续请求3次，request的地址也是不同的：

```
0xc0000b8030
0xc000186008
0xc000186018
```

在真实应用场景中，每个请求对应的Handler，常会启动额外的的goroutine进行数据查询或PRC调用等。这里可以理解为每次请求的业务处理逻辑中，需要多次访问其他服务，而这些服务是可以并发执行的，即主Gorontine内的多个Goroutine并存。而且，当请求返回时，这些额外创建的goroutine需要及时回收。此外，一个请求对应一组请求域内的数据可能会被该请求调用链条内的各goroutine所需要。

现在对上面代码在添加一点东西，当请求进来时，Handler创建一个监控goroutine，这样就会每隔1s打印一句Current request is in progress：

```
func main()  {
    http.HandleFunc("/", SayHello) // 设置访问的路由

    log.Fatalln(http.ListenAndServe(":8080",nil))
}

func SayHello(writer http.ResponseWriter, request *http.Request)  {
    fmt.Println(&request)

    go func() {
        for range time.Tick(time.Second) {
            fmt.Println("Current request is in progress")
        }
    }()

    time.Sleep(2 * time.Second)
    writer.Write([]byte("Hi, New Request Comes"))
}
```

这里假定请求需要耗时2s，在请求2s后返回，我们期望监控goroutine在打印2次Current request is in progress后即停止。但运行发现，监控goroutine打印2次后，其仍不会结束，而会一直打印下去。

问题出在创建监控goroutine后，未对其生命周期作控制，下面我们使用context作一下控制，即监控程序打印前需检测request.Context()是否已经结束，若结束则退出循环，即结束生命周期。

```
func main()  {
    http.HandleFunc("/", SayHello) // 设置访问的路由

    log.Fatalln(http.ListenAndServe(":8080",nil))
}

func SayHello(writer http.ResponseWriter, request *http.Request)  {
    fmt.Println(&request)

    go func() {
        for range time.Tick(time.Second) {
            select {
            case <- request.Context().Done():
                fmt.Println("request is outgoing")
                return
            default:
                fmt.Println("Current request is in progress")
```

```
            }
        }
    }()

    time.Sleep(2 * time.Second)
    writer.Write([]byte("Hi, New Request Comes"))
}
```

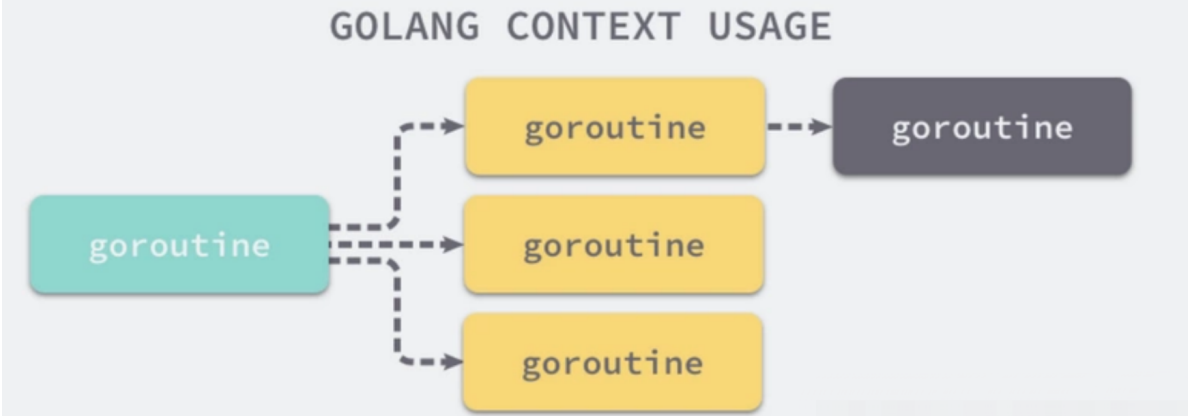
基于如上需求，context包应用而生。
context包可以提供一个请求从API请求边界到各goroutine的请求域数据传递、取消信号及截至时间等能力。

Context详解

在 Go 语言中 context 包允许传递一个 “context” 到程序中。 Context 如超时或截止日期（deadline）或通道，来指示停止运行和返回。例如，如果正在执行一个 web 请求或运行一个系统命令，定义一个超时对生产级系统通常是个好主意。因为，如果依赖的API运行缓慢，不希望在系统上备份（back up）请求，因为它可能最终会增加负载并降低所有请求的执行效率。导致级联效应。这是超时或截止日期 context 派上用场的地方。

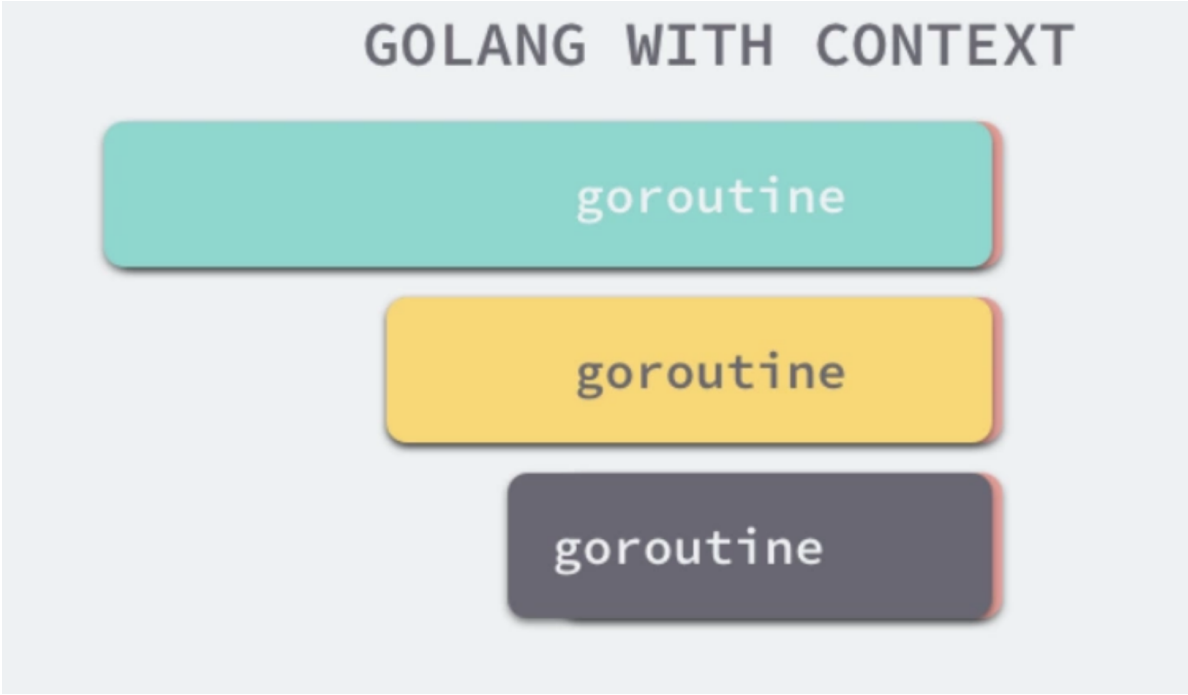
设计原理

Go 语言中的每一个请求的都是通过一个单独的 Goroutine 进行处理的，HTTP/RPC 请求的处理器往往都会启动新的 Goroutine 访问数据库和 RPC 服务，我们可能会创建多个 Goroutine 来处理一次请求，而 Context 的主要作用就是在不同的 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。



每一个 Context 都会从最顶层的 Goroutine 一层一层传递到最下层，这也是 Golang中上下文最常见的使用方式，如果没有 Context，当上层执行的操作出现错误时，下层其实不会收到错误而是会继续执行下去:

当最上层的 Goroutine 因为某些原因执行失败时，下两层的 Goroutine 由于没有接收到这个信号所以会继续工作；但是当我们正确地使用 Context 时，就可以在下层及时停掉无用的工作减少额外资源的消耗：



这其实就是 Golang 中上下文的最大作用，在不同 Goroutine 之间对信号进行同步避免对计算资源的浪费，与此同时 Context 还能携带以请求为作用域的键值对信息。

示例：

```
func main()  {
    ctx,cancel := context.WithTimeout(context.Background(),1 * time.Second)
    defer cancel()
    go HelloHandle(ctx,500*time.Millisecond)
    select {
    case <- ctx.Done():
        fmt.Println("Hello Handle ",ctx.Err())
    }
}
```

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

上面的代码，因为过期时间大于处理时间，所以我们有足够的时间处理改请求，结果所示：

```
process request with 500ms
Hello Handle  context deadline exceeded
```

HelloHandle函数并没有进入超时的select分支，但是main函数的select却会等待context.Context的超时并打印出Hello Handle context deadline exceeded。如果我们将处理请求的时间增加至2000ms，程序就会因为上下文过期而被终止。

```
context deadline exceeded
Hello Handle  context deadline exceeded
```

Context 接口

context.Context 是 Go 语言在 1.7 版本中引入标准库的接口1，该接口定义了四个需要实现的方法，其中包括：

- Deadline — 返回 context.Context 被取消的时间，也就是完成工作的截止日期；
- Done — 返回一个 Channel，这个 Channel 会在当前工作完成或者上下文被取消之后关闭，多次调用 Done 方法会返回同一个 Channel；
- Err — 返回 context.Context 结束的原因，它只会在 Done 返回的 Channel 被关闭时才会返回非空的值；如果 context.Context 被取消，会返回 Canceled 错误；如果 context.Context 超时，会返回 DeadlineExceeded 错误；
- Value — 从 context.Context 中获取键对应的值，对于同一个上下文来说，多次调用 Value 并传入相同的 Key 会返回相同的结果，该方法可以用来传递请求特定的数据。

```
type Context interface {
   // 返回当前 Context 被取消的时间（完成工作的截止时间）；如果没有设置时间，ok 返回 false
   Deadline() (deadline time.Time, ok bool)

   // 返回一个 Channel，该 Channel 会在当前工作完成或者上下文被取消之后关闭
   Done() <-chan struct{}

   // 返回当前 Context 结束的原因，它只会在 Done() 方法对应的 Channel 关闭时返回非 nil 的
值
   // 1. 如果当前 Context 被取消，返回 Canneled 错误
   // 2. 如果当前 Context 超时，返回 DeadlineExceeded 错误
   Err() error

   // 从当前 Context 中返回 key 对应的 value
   Value(key interface{}) interface{}
}
```

Background() 和 TODO() 函数

`context` 包提供 `Background()` 和 `TODO()` 函数，分别返回实现了 `Context` 接口的内置的上下文对象 `background` 和 `todo`。一般而言，我们代码最开始都是以这两个内置的上下文对象作为最顶层的 `partent context`，衍生出更多的子上下文对象。

- `Background()`：主要用于 `main` 函数、初始化以及测试代码中，作为 `Context` 这个树结构的最顶层的 `Context`，也就是根 `Context`。这个函数返回一个空context。这只能用于高等级（在 main 或顶级请求处理中）。
- `TODO()`：如果目前还不知道具体的使用场景，不知道该使用什么 `Context` 的时候，可以使用这个。这个函数返回一个空context。这只能用于高等级（在 main 或顶级请求处理中）。

`Background()` 和 `TODO()` 函数的源码如下：

```
// background 和 todo 本质是 emptyCtx 结构体类型，是一个不可取消，没有设置截止时间，没有携带任何值的Context
var (
    background = new(emptyCtx)
    todo       = new(emptyCtx)
)

func Background() Context {
    return background
}

func TODO() Context {
    return todo
}

type emptyCtx int

func (*emptyCtx) Deadline() (deadline time.Time, ok bool) {
    return
}

func (*emptyCtx) Done() <-chan struct{} {
    return nil
}

func (*emptyCtx) Err() error {
    return nil
}

func (*emptyCtx) Value(key interface{}) interface{} {
    return nil
}
```

context.Background 和 context.TODO 函数其实也只是互为别名，没有太大的差别。它们只是在使用和语义上稍有不同：

- 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；

在多数情况下，如果当前函数没有上下文作为入参，我们都会使用 context.Background 作为起始的上下文向下传递。

`Background()`和`TODO()`返回的上下文对象一般作为最顶层的根`Context`，然后通过调用`withCancel()`、`WithDeadline()`、`WithTimeout()`或`WithValue()`函数创建其派生的子上下文。当一个上下文被取消时，它派生的所有上下文也会被取消。

WithCancel() 函数

返回具有新的 Done 通道的父上下文的副本 ctx。传递一个父Context作为参数，返回子Context，以及一个取消函数用来取消Context。

当调用返回的 cancel 函数或关闭父上下文的 Done 通道时，返回的上下文 ctx 的 Done 通道也会被关闭。

示例：

```
package main

import (
    "context"
    "fmt"
)

func gen(ctx context.Context) <-chan int {
    dst := make(chan int)
    n := 1
    go func() {
        for {
            select {
            case <-ctx.Done():
                // return 结束该 goroutine，防止泄露
                return
            case dst <- n:
                n++
            }
        }
    }()
    return dst
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())
    // 获取完需要的整数后，调用 cancel
    defer cancel()

    for n := range gen(ctx) {
        fmt.Println(n)
        if n == 5 {
            break
        }
    }
}
#结果
// 1
// 2
// 3
// 4
```

```
// 5
```

```
package main

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

var wg sync.WaitGroup

func worker(ctx context.Context) {
    go worker2(ctx)
LOOP:
    for {
        fmt.Println("worker")
        time.Sleep(time.Second)
        select {
        case <-ctx.Done(): // 等待上级通知
            break LOOP
        default:
        }
    }
    wg.Done()
}

func worker2(ctx context.Context) {
LOOP:
    for {
        fmt.Println("worker2")
        time.Sleep(time.Second)
        select {
        case <-ctx.Done(): // 等待上级通知
            break LOOP
        default:
        }
    }
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())
    wg.Add(1)
    go worker(ctx)
    time.Sleep(time.Second * 3)
    cancel() // 通知子 goroutine 结束
    wg.Wait()
    fmt.Println("over")
}

// worker2
// worker
// worker
// worker2
```

```
// worker2
// worker
// over
```

WithDeadline() 函数

返回父上下文的副本 ctx，完成工作的截止时间（deadline）调整为不迟于 d。

如果父上下文的 deadline 早于 d，则 WithDeadline(parent, d) 在语义上等同于父上下文。

当遇到以下三种情况时，返回的上下文 ctx 的 Done 通道将被关闭，以最先发生的情况为准： 1. 设置的截止日期 d 过期；2. 调用返回的 cancal 函数；3. 父上下文的 Done 通道被关闭。

和WithCancel差不多，它会多传递一个截止时间参数，意味着到了这个时间点，会自动取消Context，当然我们也可以不等到这个时候，可以提前通过取消函数进行取消。

```
func WithDeadline(parent Context, d time.Time) (ctx Context, cancal CancelFunc)
```

示例：

```
package main

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

func main() {
    // 设置当前上下文 50ms 后过期
    d := time.Now().Add(50 * time.Millisecond)
    ctx, cancel := context.WithDeadline(context.Background(), d)

    // 尽管 ctx 会过期，但在任何情况下调用它的 cancel 函数都是很好的实践。
    // 如果不这样做，可能会使上下文及其父类存活的时间超过必要的时间。
    defer cancel()

    select {
    case <-time.After(1 * time.Second):
        // 等待 1 秒后打印 overslept 退出
        fmt.Println("overslept")
    case <-ctx.Done():
        // 等待 ctx 过期后，退出
        fmt.Println(ctx.Err())
    }
}

// context deadline exceeded
```

WithTimeout() 函数

等同于 WithDeadline(parent, time.Now().Add(timeout))。这个表示是超时自动取消，是多少时间后自动取消Context的意思。

```
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

示例：

```
package main

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

var wg sync.WaitGroup

func worker(ctx context.Context) {
LOOP:
    for {
        fmt.Println("db connecting...")
        time.Sleep(time.Millisecond * 10) // 假设正常连接数据库耗时 10 毫秒
        select {
        case <-ctx.Done(): // 50ms 后调用
            break LOOP
        default:
        }
    }
    fmt.Println("worker done!")
    wg.Done()
}

func main() {
    // 设置当前上下文 50ms 后过期
    ctx, cancel := context.WithTimeout(context.Background(),
50*time.Millisecond)
    wg.Add(1)
    go worker(ctx)
    time.Sleep(time.Second * 5)
    cancel() // 通知子 goroutine 结束
    wg.Wait()
    fmt.Println("over")
}

// db connecting...
// db connecting...
// db connecting...
// worker done!
// over
```

withValue() 函数

返回父节点的副本，可传递一个与 key 关联的 val 值。它是为了生成一个绑定了一个键值对数据的Context，这个绑定的数据可以通过Context.Value方法访问到.

```
func WithValue(parent Context, key, val interface{}) Context
```

示例:

```
package main

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

type TraceCode string

var wg sync.WaitGroup

func worker(ctx context.Context) {
    key := TraceCode("TRACE_CODE")
    traceCode, ok := ctx.Value(key).(string) // 在子 goroutine 中获取 trace code
    if !ok {
        fmt.Println("invalid trace code")
    }
LOOP:
    for {
        fmt.Printf("worker, trace code:%s\n", traceCode)
        time.Sleep(time.Millisecond * 10) // 假设正常连接数据库耗时 10 毫秒
        select {
        case <-ctx.Done(): // 50毫秒后自动调用
            break LOOP
        default:
        }
    }
    fmt.Println("worker done!")
    wg.Done()
}

func main() {
    // 创建一个过期时间为 50ms 的上下文
    ctx, cancel := context.WithTimeout(context.Background(),
50*time.Millisecond)
    // 在系统的入口中设置 trace code 传递给后续启动的 goroutine 实现日志数据聚合
    ctx = context.WithValue(ctx, TraceCode("TRACE_CODE"), "12512312234")
    wg.Add(1)
    go worker(ctx)
    time.Sleep(time.Second * 5)
    cancel() // 通知子goroutine结束
    wg.Wait()
    fmt.Println("over")
```

```
}

// worker, trace code:12512312234
// worker, trace code:12512312234
// worker, trace code:12512312234
// worker, trace code:12512312234
// worker, trace code:12512312234
// worker done!
// over
```

客户端超时取消示例

服务端代码：

```
package main

import (
    "fmt"
    "math/rand"
    "net/http"
    "time"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        random := rand.New(rand.NewSource(time.Now().UnixNano()))
        number := random.Intn(2)
        if number == 0 {
            time.Sleep(time.Second * 10) // 耗时 10 秒的慢响应
            fmt.Fprintf(w, "slow response")
            return
        }
        fmt.Fprintf(w, "quick response") // 快速响应
    })

    err := http.ListenAndServe(":8080", nil)
    if err != nil {
        panic(err)
    }
}
```

客户端代码

```
package main

import (
    "context"
    "fmt"
    "io/ioutil"
    "net/http"
    "sync"
    "time"
)
```

```
type respData struct {
    resp *http.Response
    err  error
}

func doCall(ctx context.Context) {
    client := http.Client{
        Transport: &http.Transport{
            // 设置为长连接
            DisableKeepAlives: true,
        },
    }

    respChan := make(chan *respData, 1)
    req, err := http.NewRequest("GET", "http://127.0.0.1:8080/", nil)
    if err != nil {
        fmt.Println("new request failed, err:%v\n", err)
        return
    }
    // 使用带超时的 ctx 创建一个新的 client request
    req = req.WithContext(ctx)

    var wg sync.WaitGroup
    wg.Add(1)
    defer wg.Wait()
    go func() {
        resp, err := client.Do(req)
        fmt.Printf("client.do resp:%v, err:%v\n", resp, err)
        respChan <- &respData{
            resp: resp,
            err:  err,
        }
        wg.Done()
    }()

    select {
    case <-ctx.Done():
        fmt.Println("call api timeout")
    case result := <-respChan:
        fmt.Println("call server api success")
        if result.err != nil {
            fmt.Printf("call server api failed, err:%v\n", result.err)
            return
        }
        defer result.resp.Body.Close()
        data, _ := ioutil.ReadAll(result.resp.Body)
        fmt.Printf("resp:%v\n", string(data))
    }
}

func main() {
    // 将当前 ctx 的超时时间设置为 100ms
    ctx, cancel := context.WithTimeout(context.Background(),
100*time.Millisecond)
    // 调用cancel释放子goroutine资源
```

```
    defer cancel()

    doCall(ctx)
}

// 超时的输出：
// call api timeout
// client.do resp:<nil>, err:Get "http://127.0.0.1:8080/": context deadline
exceeded

// 不超时的输出：
// client.do resp:&{200 OK 200 HTTP/1.1 1 1 map[Content-Length:[14] Content-Type:
[text/plain; charset=utf-8] Date:[Sun, 22 May 2022 07:50:29 GMT]] 0xc000206080 14
[] true false map[] 0xc00013a100 <nil>}, err:<nil> call server api success
// resp:quick response
```

总结

Context使用原则

1. context.Background 只应用在最高等级，作为所有派生 context 的根。
2. context取消是建议性的，这些函数可能需要一些时间来清理和退出。
3. 不要把Context放在结构体中，要以参数的方式传递。
4. 以Context作为参数的函数方法，应该把Context作为第一个参数，放在第一位。
5. 给一个函数方法传递Context的时候，不要传递nil，如果不知道传递什么，就使用context.TODO
6. Context的Value相关方法应该传递必须的数据，不要什么数据都使用这个传递。
7. context.Value应该很少使用，它不应该被用来传递可选参数。这使得 API 隐式的并且可以引起错误。取而代之的是，这些值应该作为参数传递。
8. Context是线程安全的，可以放心的在多个goroutine中传递。同一个Context可以传给使用其的多个goroutine，且Context可被多个goroutine同时安全访问。
9. Context 结构没有取消方法，因为只有派生 context 的函数才应该取消 context。

Go 语言中的 context.Context 的**主要作用还是在多个 Goroutine 组成的树中同步取消信号以减少对资源的消耗和占用**，虽然它也有传值的功能，但是这个功能我们还是很少用到。在真正使用传值的功能时我们也应该非常谨慎，使用 context.Context 进行传递参数请求的所有参数一种非常差的设计，比较常见的使用场景是传递请求对应用户的认证令牌以及用于进行分布式追踪的请求 ID。