22 | 优雅的离场: Context超时控制与原理

2022-11-29 郑建勋 来自北京

《Go进阶 · 分布式爬虫实战》

课程介绍 >

讲述：郑建勋

时长 13:26　大小 12.27M

你好，我是郑建勋。

在 Go 语言的圈子里有一句名言：

> **Never start a goroutine without knowing how it will stop。**

意思是，如果你不知道协程如何退出，就不要使用它。

如果想要正确并优雅地退出协程，首先必须正确理解和使用 Context 标准库。Context 是使用非常频繁的库，在实际的项目开发中，有大量第三方包的 API（例如 Redis Client、MongoDB Client、标准库中涉及到网络调用的 API）的第一个参数都是 Context。

复制代码

```
1 // net/http
```

```
func (r *Request) WithContext(ctx context.Context) *Request
// sql
func (db *DB) BeginTx(ctx context.Context, opts *TxOptions) (*Tx, error)
// net
func (d *Dialer) DialContext(ctx context.Context, network, address string) (Con
```


那么 Context 的作用是什么？应该如何去使用它？Context 的最佳实践又是怎样的？让我们带着这些疑问开始这节课的学习。

我们为什么需要 Context？

协程在 Go 中是非常轻量级的资源，它可以被动态地创建和销毁。例如，在典型的 HTTP 服务器中，每个新建立的连接都会新建一个协程。我们之前介绍 HTTP 请求时就说过，标准库内部在处理时创建了多个协程。当请求完成后，协程也随之被销毁。但是，请求连接可能临时终止也可能超时。这个时候，我们希望安全并及时地停止协程和与协程关联的子协程，避免白白消耗资源。

在没有 Context 之前我们一般会怎么做呢？我们需要借助通道的 close 机制，这个机制会唤醒所有监听该通道的协程，并触发相应的退出逻辑。写法大致如下：

复制代码

```
select {
  case <-c:
    // 业务逻辑
  case <-done:
    fmt.Println("退出协程")
  }
```

随着 Go 语言的发展，越来越多的程序都开始需要进行这样的处理。然而不同的程序，甚至同一程序的不同代码片段，它们的退出逻辑的命名和处理方式都会有所不同。例如，有的将退出通道命名为了 done，有的命名为了 closed，有的采取了函数包裹的形式←g.dnoe()。如果有一套统一的规范，代码的语义将会更加清晰明了。例如，引入了 Context 之后，退出协程的规范写法将是“←ctx.Done() ”。

复制代码

```
func Stream(ctx context.Context, out chan<- Value) error {
  for {
    v, err := DoSomething(ctx)
    if err != nil {
```

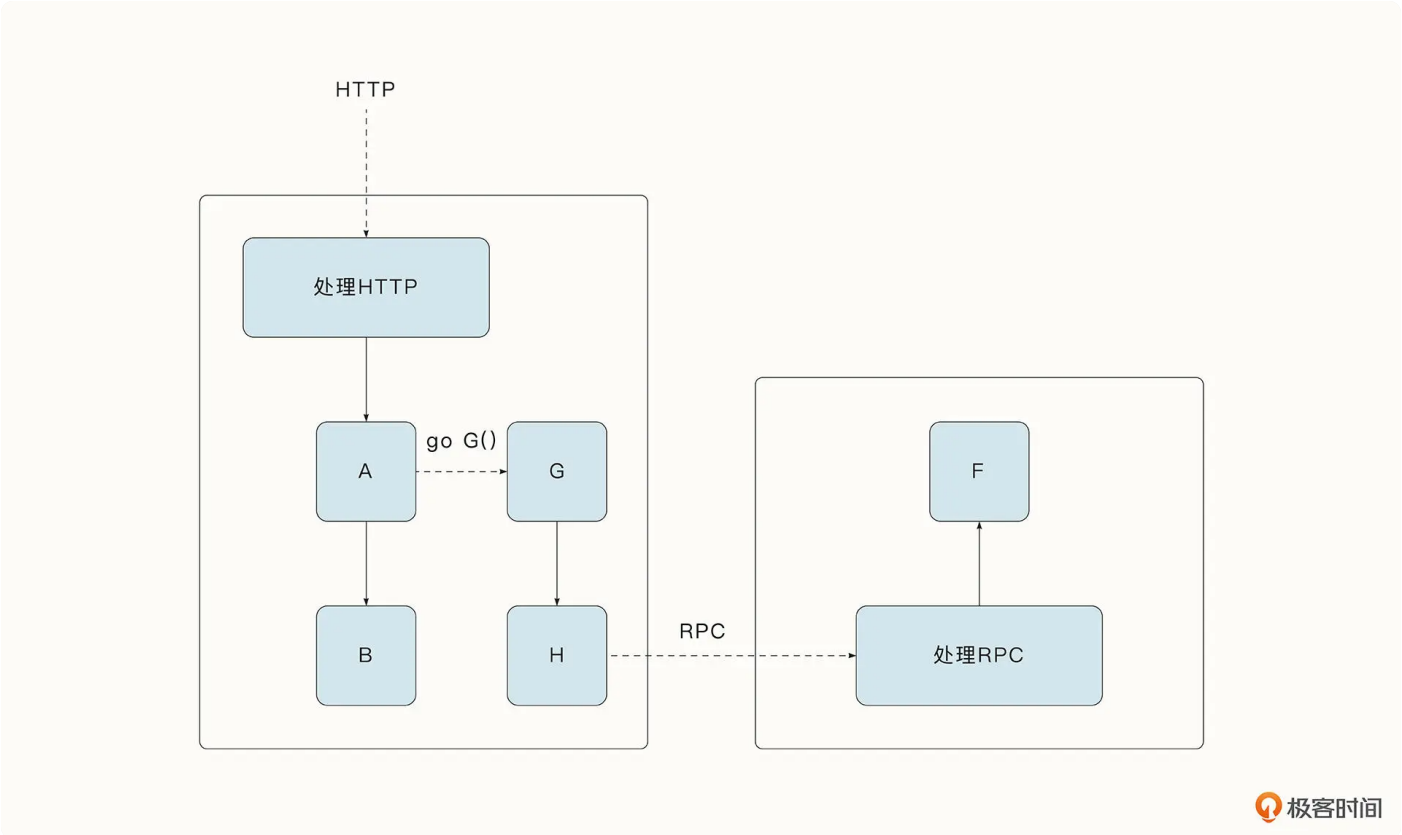
```
      return err
    }
    select {
    case <-ctx.Done():
      return ctx.Err()
    case out <- v:
    }
  }
}
```


为了对超时进行规范处理，在 Go 1.7 之后，Go 官方引入了 Context 来实现协程的退出。不仅如此，Context 还提供了跨协程、甚至是跨服务的退出管理。

Context 本身的含义是上下文，我们可以理解为它内部携带了超时信息、退出信号，以及其他一些上下文相关的值（例如携带本次请求中上下游的唯一标识 trace_id）。 由于 Context 携带了上下文信息，父子协程就可以“联动”了。

我们举个例子来看看 Context 是怎么处理协程级联退出情况的。

如下图所示，服务器处理 HTTP 请求一般会单独开辟一个协程，假设该处理协程调用了函数 A，函数 A 中也可能创建一个新的协程。假设新的协程调用了函数 G，函数 G 中又有可能通过 RPC 远程调用了其他服务的 API，并最终调用了函数 F。



假设这个时候上游将连接断开，或者服务处理时间超时，我们希望能够立即退出函数 A、函数 G 和函数 F 所在的协程。

在实际场景中可能是这样的，上游给服务的处理时间是 500ms，超过这一时间这一请求就无效了。 A 服务当前已经花费了 200ms 的时间，G 又用了 100ms 调用 RPC，那么留给 F 的处理时间就只有 200ms 了。 如果远程服务 F 在 200ms 后没有返回，所有协程都需要感知到并快速关闭。

而使用 Context 标准库就是当前处理这种协程级联退出的标准做法。让我们先看一看 Context 的使用方法。在 Context 标准库中重要的结构 context.Context 其实是一个接口，它提供了 Deadline、Done、Err、Value 这 4 种方法：

复制代码

```
type Context interface {
  Deadline() (deadline time.Time, ok bool)
  Done() <-chan struct{}
  Err() error
  Value(key interface{}) interface{}
}
```

这 4 种方法的功能如下。

- Deadline 方法用于返回 Context 的过期时间。Deadline 第一个返回值表示 Context 的过期时间，第二个返回值表示是否设置了过期时间，如果多次调用 Deadline 方法会返回相同的值。
- Done 是使用最频繁的方法，它会返回一个通道。一般的做法是调用者在 select 中监听该通道的信号，如果该通道关闭则表示服务超时或异常，需要执行后续退出逻辑。多次调用 Done 方法会返回相同的通道。
- 通道关闭后，Err 方法会返回退出的原因。
- Value 方法返回指定 key 对应的 value，这是 Context 携带的值。key 必须是可比较的，一般的用法 key 是一个全局变量，通过`context.WithValue` 将 key 存储到 Context 中，并通过`Context.Value` 方法取出。

Context 接口中的这四个方法可以被多次调用，其返回的结果相同。同时，Context 的接口是并发安全的，可以被多个协程同时使用。

context.Value

因为在实践中 Context 携带值的情况并不常见，所以这里我们单独讲一讲 context.Value 的适用场景。

context.Value 一般在远程过程调用中使用，例如存储分布式链路跟踪的 traceId 或者鉴权相关的信息，并且该值的作用域在请求结束时终结。同时 key 必须是访问安全的，因为可能有多个协程同时访问它。

如下所示，withAuth 函数是一个中间件，它可以让我们在完成实际的 HTTP 请求处理前进行 hook。 在这个例子中，我们获取了 HTTP 请求 Header 头中的鉴权字段 Authorization，并将其存入了请求的上下文 Context 中。而实际的处理函数 Handle 会从 Context 中获取并验证用户的授权信息，以此判断用户是否已经登录。

复制代码

```go
const TokenContextKey = "MyAppToken"

// 中间件
func WithAuth(a Authorizer, next http.Handler) http.Handler {
  return http.HandleFunc(func(w http.ResponseWriter, r *http.Request) {
    auth := r.Header.Get("Authorization")
    if auth == "" {
      next.ServeHTTP(w, r) // 没有授权
      return
    }
    token, err := a.Authorize(auth)
    if err != nil {
      http.Error(w, err.Error(), http.StatusUnauthorized)
      return
    }
    ctx := context.WithValue(r.Context(), TokenContextKey, token)
    next.ServeHTTP(w, r.WithContext(ctx))
  })
}

// HTTP请求实际处理函数
func Handle(w http.ResponseWriter, r *http.Request) {
  // 获取授权
  if token := r.Context().Value(TokenContextKey); token != nil {
    // 用户登录
  } else {
    // 用户未登录
  }
}
```

Context 是一个接口，这意味着需要有对应的具体实现。用户可以自己实现 Context 接口，并严格遵守 Context 接口规定的语义。当然，我们使用得最多的还是 Go 标准库中的实现。

当我们调用 context.Background 函数或 context.TODO 函数时，会返回最简单的 Context 实现。context.Background 返回的 Context 一般是作为根对象存在，不具有任何功能，不可以退出，也不能携带值。

复制代码

```
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
}
```

因此，要具体地使用 Context 的功能，需要派生出新的 Context。配套的函数有下面这几个，其中，前三个函数都用于派生出有退出功能的 Context。

复制代码

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

- WithCancel 函数会返回一个子 Context 和 cancel 方法。子 Context 会在两种情况下触发退出：一种情况是调用者主动调用了返回的 cancel 方法；另一种情况是当参数中的父 Context 退出时，子 Context 将级联退出。
- WithTimeout 函数指定超时时间。当超时发生后，子 Context 将退出。因此，子 Context 的退出有三种时机，一种是父 Context 退出；一种是超时退出；最后一种是主动调用 cancel 函数退出。

- WithDeadline 和 WithTimeout 函数的处理方法相似，不过它们的参数指定的是最后到期的时间。
- WithValue 函数会返回带 key-value 的子 Context。

举一个例子来说明一下 Context 中的级联退出。下面的代码中 childCtx 是 preCtx 的子 Context，其设置的超时时间为 300ms。但是 preCtx 的超时时间为 100 ms，因此父 Context 退出后，子 Context 会立即退出，实际的等待时间只有 100ms。


```
func main() {
  ctx := context.Background()
  before := time.Now()
  preCtx, _ := context.WithTimeout(ctx, 100*time.Millisecond)
  go func() {
    childCtx, _ := context.WithTimeout(preCtx, 300*time.Millisecond)
    select {
    case <-childCtx.Done():
      after := time.Now()
      fmt.Println("child during:", after.Sub(before).Milliseconds())
    }
  }()
  select {
  case <-preCtx.Done():
    after := time.Now()
    fmt.Println("pre during:", after.Sub(before).Milliseconds())
  }
}
```

这时的输出如下，父 Context 与子 Context 退出的时间差接近 100ms：


```
pre during: 104
child during: 104
```

当我们把 preCtx 的超时时间修改为 500ms 时：


```
preCtx ,_:= context.WithTimeout(ctx,500*time.Millisecond)
```

从新的输出中可以看出，子协程的退出不会影响父协程的退出。


```
child during: 304
pre during: 500
```

从上面这个例子可以看出，父 Context 的退出会导致所有子 Context 的退出，而子 Context 的退出并不会影响父 Context。

Context 最佳实践

了解了 Context 的基本用法，接下来让我们来看看 Go 标准库中是如何使用 Context 的。

对 HTTP 服务器和客户端来说，超时处理是最容易犯错的问题之一。因为在网络连接到请求处理的多个阶段，都可能有相对应的超时时间。以 HTTP 请求为例，http.Client 有一个参数 Timeout 用于指定当前请求的总超时时间，它包括从连接、发送请求、到处理服务器响应的时间的总和。

复制代码

```
c := &http.Client{
    Timeout: 15 * time.Second,
}
resp, err := c.Get("<https://baidu.com/>")
```

标准库 client.Do 方法内部会将超时时间换算为截止时间并传递到下一层。setRequestCancel 函数内部则会调用 context.WithDeadline ，派生出一个子 Context 并赋值给 req 中的 Context。

复制代码

```
func (c *Client) do(req *Request) (retres *Response, reterr error) {
  ...
  deadline      = c.deadline()
  c.send(req, deadline);
}

func setRequestCancel(req *Request, rt RoundTripper, deadline time.Time) {
    req.ctx, cancelCtx = context.WithDeadline(oldCtx, deadline)
    ...
}
```

在获取连接时，正如我们前面课程中讲到的，如果从闲置连接中找不到连接，则需要陷入 select 中去等待。如果连接时间超时，req.Context().Done() 通道会收到信号立即退出。在实际发送数据的 transport.roundTrip 函数中，也有很多类似的例子，它们都是通过在 select 语句中监听 Context 退出信号来实现超时控制的，这里就不再赘述了。

复制代码

```
func (t *Transport) getConn(treq *transportRequest, cm connectMethod) (pc *pers
...
select {
  case <-w.ready:
    return w.pc, w.err
  case <-req.Cancel:
    return nil, errRequestCanceledConn
  case <-req.Context().Done():
    return nil, req.Context().Err()
    return nil, err
  }
}
```

获取 TCP 连接需要调用 sysDialer.dialSerial 方法，dialSerial 的功能是从 addrList 地址列表中取出一个地址进行连接，如果与任一地址连接成功则立即返回。代码如下所示，不出所料，该方法的第一个参数为上游传递的 Context。

复制代码

```
// net/dial.go
func (sd *sysDialer) dialSerial(ctx context.Context, ras addrList) (Conn, error
  for i, ra := range ras {
    // 协程是否需要退出
    select {
    case <-ctx.Done():
      return nil, &OpError{Op: "dial", Net: sd.network, Source: sd.LocalAddr, A
    default:
    }

    dialCtx := ctx

     // 是否设置了超时时间
    if deadline, hasDeadline := ctx.Deadline(); hasDeadline {
      // 计算连接的超时时间
      partialDeadline, err := partialDeadline(time.Now(), deadline, len(ras)-i)
      if err != nil {
        // 已经超时了.
```

```
19          if firstErr == nil {
20            firstErr = &OpError{Op: "dial", Net: sd.network, Source: sd.LocalAddr
21          }
22          break
23        }
24        // 派生出新的context，传递给下游
25        if partialDeadline.Before(deadline) {
26          var cancel context.CancelFunc
27          dialCtx, cancel = context.WithDeadline(ctx, partialDeadline)
28          defer cancel()
29        }
30      }
31
32      c, err := sd.dialSingle(dialCtx, ra)
33      ...
34  }
```


我们来看看 dialSerial 函数几个比较有代表性的 Context 用法。

- 首先，第 3 行代码遍历地址列表时，判断 Context 通道是否已经退出，如果没有退出，会进入到 select 的 default 分支。如果通道已经退出了，则直接返回，因为继续执行已经没有必要了。
- 接下来，第 14 行代码通过 ctx.Deadline() 判断是否传递进来的 Context 有超时时间。如果有超时时间，我们需要协调好后面每一个连接的超时时间。例如，我们总的超时时间是 600ms，一共有 3 个连接，那么每个连接分到的超时时间就是 200ms，这是为了防止前面的连接过度占用了时间。partialDeadline 会帮助我们计算好每一个连接的新的到期时间，如果该到期时间小于总到期时间，我们会派生出一个子 Context 传递给 dialSingle 函数，用于控制该连接的超时。
- dialSingle 函数中调用了 ctx.Value，用来获取一个特殊的接口 nettrace.Trace。nettrace.Trace 用于对网络包中一些特殊的地方进行 hook。dialSingle 函数作为网络连接的起点，如果上下文中注入了 trace.ConnectStart 函数，则会在 dialSingle 函数之前调用 trace.ConnectStart 函数，如果上下文中注入了 trace.ConnectDone 函数，则会在执行 dialSingle 函数之后调用 trace.ConnectDone 函数。


```
1  func (sd *sysDialer) dialSingle(ctx context.Context, ra Addr) (c Conn, err erro
2    trace, _ := ctx.Value(nettrace.TraceKey{}).(*nettrace.Trace)
3    if trace != nil {
4      raStr := ra.String()
5      if trace.ConnectStart != nil {
6        trace.ConnectStart(sd.network, raStr)
```

```
    }
    if trace.ConnectDone != nil {
      defer func() { trace.ConnectDone(sd.network, raStr, err) }()
    }
  }
  la := sd.LocalAddr
  switch ra := ra.(type) {
  case *TCPAddr:
    la, _ := la.(*TCPAddr)
    // tcp连接
    c, err = sd.dialTCP(ctx, la, ra)
    ...
}
```


到这里，我们就通过 Go 网络标准库中对 Context 的使用，将 Context 的使用场景和最佳实践方式都梳理了一遍。

由于标准库为我们提供了 Timeout 参数，我们在项目中实践超时控制就容易多了。只要在 BrowserFetch 结构体中增加 Timeout 超时参数，然后设置超时参数到 http.Client 中就大功告成了。

复制代码

```
type BrowserFetch struct {
   Timeout time.Duration
}

//模拟浏览器访问
func (b BrowserFetch) Get(url string) ([]byte, error) {
  client := &http.Client{
    Timeout: b.Timeout,
  }
...
}

func main() {
  url := "<https://book.douban.com/subject/1007305/>"
  var f collect.Fetcher = collect.BrowserFetch{
    Timeout: 300 * time.Millisecond,
  }
  body, err := f.Get(url)
  ...
  }
```

不过要提醒一下，如果我们设置的超时时间太短，会出现下面这样“Context 超时”的错误信息：

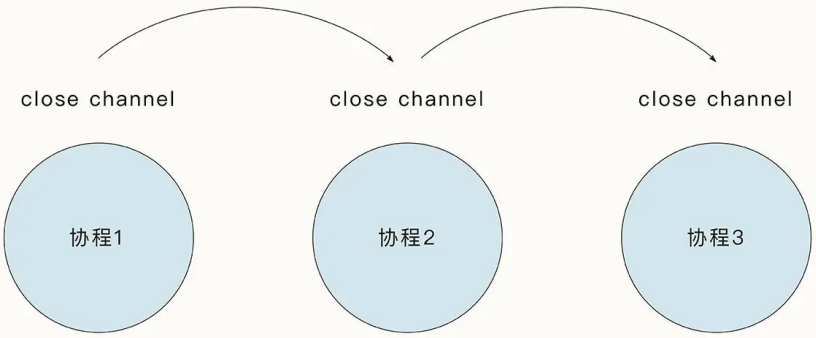

```
read content failed:Get "<https://book.douban.com/subject/1007305/>": context d
```

Context 底层原理

了解了 Context 的价值和最佳实践，我们再来简单看一下它的底层原理。

Context 在很大程度上利用了通道的一个特性：通道在 close 时，会通知所有监听它的协程。

每个派生出的子 Context 都会创建一个新的退出通道，这样，只要组织好 Context 之间的关系，就可以实现继承链上退出信号的传递。如图所示的三个协程中，关闭通道 A 会连带关闭调用链上的通道 B，通道 B 会关闭通道 C。



前面我们说，Context.Background 函数和 Context.TODO 函数会生成一个根 Context。要使用 context 的退出功能，需要调用 WithCancel 或 WithTimeout，派生出一个新的结构 Context。 WithCancel 底层对应的结构为 cancelCtx，WithTimeout 底层对应的结构为 timerCtx，timerCtx 包装了 cancelCtx，并存储了超时时间。代码如下所示。

```
type cancelCtx struct {
  Context

  mu       sync.Mutex
  done     atomic.Value
  children map[canceler]struct{}
```

```
  err      error
}

type timerCtx struct {
  cancelCtx
  timer *time.Timer

  deadline time.Time
}
```


cancelCtx 第一个字段保留了父 Context 的信息。children 字段则保存了当前 Context 派生的子 Context 的信息，每个 Context 都会有一个单独的 done 通道。

而 WithDeadline 函数会先判断父 Context 设置的超时时间是否比当前 Context 的超时时间短。如果是，那么子协程会随着父 Context 的退出而退出，没有必要再设置定时器。

当我们使用了标准库中默认的 Context 实现时，propagateCancel 函数会将子 Context 加入父协程的 children 哈希表中，并开启一个定时器。当定时器到期时，会调用 cancel 方法关闭通道，级联关闭当前 Context 派生的子 Context，并取消与父 Context 的绑定关系。这种特性就产生了调用链上连锁的退出反应。

复制代码

```
func (c *cancelCtx) cancel(removeFromParent bool, err error) {

  ...
   // 关闭当前通道
  close(d)
  // 级联关闭当前context派生的子context
  for child := range c.children {
    child.cancel(false, err)
  }
  c.children = nil
  c.mu.Unlock()
  // 从父context中能够删除当前context关联
  if removeFromParent {
    removeChild(c.Context, c)
  }
}
```

总结

好了，这节课就讲到这里，我们总结一下。这节课，我们介绍了如何用 Context 安全而优雅地完成协程的退出。Context 是使用频率非常高的函数，它不仅为我们规范了处理协程退出的风格，而且它的一些特性，诸如并发安全、级联退出、携带上下文信息都比较好用。

自从 Context 出现之后，许多的包都相继完成了改造，开始在 API 的第一个参数中传递 Context，特别是涉及到需要跨服务调用的场景时。在 Go 网络处理中，我们可以设置很多超时时间来控制请求退出，这背后是离不开标准库对 Context 的巧妙使用的。

课后题

最后，我也给你留一道思考题。

Go HTTP 标准库中其实有多种类型的超时，你知道有哪些吗？

欢迎你在留言区与我交流讨论，我们下节课再见！

上一篇 21 | 采集引擎：实战接口抽象与模拟浏览器访问

下一篇 23 | 偷梁换柱：为爬虫安上代理的翅膀

精选留言 (1)

写留言

Geek_7ba156
2022-11-29 来自江苏

老师课程后面会有websocket相关的爬虫设计吗？毕竟网站数据也不只是restfulapi，现在很多数据都是wss了。对于wss的控制，keepalive，我觉得也很需要了解，gorilla自带的keepalive不是特别好用，如果有比较好的项目也可推荐下