

1. 什么是context

2. 使用场景

2.1 Http server

2.1.1 数据共享

2.1.2 取消请求

2.1.2 超时时间控制

总结

3. 用法样例

3.1 传递共享的数据

3.2 取消 goroutine

3.3 超时取消

4. 原理解析

4.1 context创建方式

4.2 源码赏析

4.2.1 emptyCtx

4.2.2 valueCtx

4.2.3 cancelCtx

4.2.3 WithDeadline、WithTimeout

4.3 源码总结

5. 最佳实践

5.1 优缺点

5.2 最佳实践

6. 评价

7. 参考资料

1. 什么是context

最早 `context` 是独立的第三方库，后来才移到标准库里。Go 1.7 标准库引入 `context`，中文译作“上下文”，准确说它是 `goroutine` 的上下文，包含 `goroutine` 的运行状态、环境、现场等信息

`context` 主要用来在 `goroutine` 之间传递上下文信息，包括：取消信号、超时时间、`k-v` 等。

随着 `context` 包的引入，标准库中很多接口因此加上了 `context` 参数，例如 `database/sql` 包、`http` 标准库，第三方库如 `gin`。`context` 几乎成为了并发控制和超时控制的标准做法。

`context.Context` 类型的值可以协调多个 `goroutine` 中的代码执行“取消”操作，并且可以存储键值对。最重要的是它是并发安全的，可同时传给多个协程。

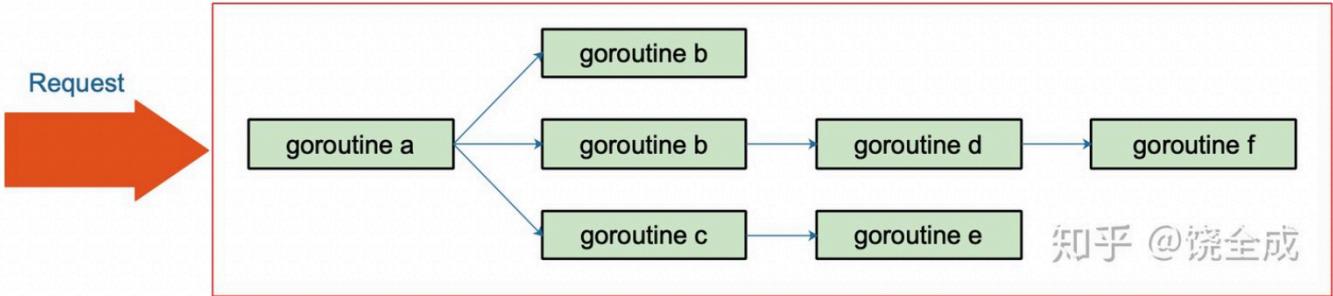
与它协作的 API 都可以由外部控制执行“取消”操作，例如：取消一个 HTTP 请求的执行。

2. 使用场景

2.1 Http server

Go 常用来写后台服务，通常只需要几行代码，就可以搭建一个 `http server`。

在 Go 的 `server` 里，通常每来一个请求都会启动若干个 `goroutine` 同时工作：有些去数据库拿数据，有些调用下游接口获取相关数据……



2.1.1 数据共享

这些 goroutine 需要共享这个请求的基本数据，例如登陆的 token、traceId

2.1.2 取消请求

可能是使用者关闭了浏览器或是已经超过了请求方规定的超时时间（Http client超时时间），请求方直接放弃了这次请求结果，此时需要停止这次请求相关的所有goroutine

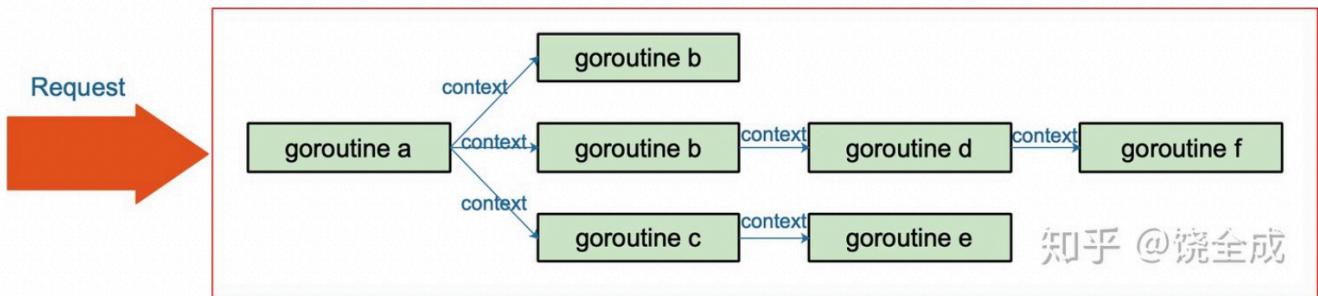
2.1.2 超时时间控制

Go 语言中的 server 实际上是一个“协程模型”，也就是说一个协程处理一个请求，这个协程可以根据需要开启更多的协程处理各项任务，等待所有协程处理完毕后一起返回响应。例如在业务的高峰期，某个下游服务的响应变慢，而当前系统的请求又没有超时控制，或者超时时间设置地过大，那么等待下游服务返回数据的协程就会越来越多

协程是要消耗系统资源的，后果就是协程数激增，内存占用飙升，甚至导致服务不可用。更严重的会导致雪崩效应通过设置“允许下游最长处理时间”就可以避免。例如，给下游设置的 timeout 是 50 ms，如果超过这个值还没有接收到返回数据，就直接向客户端返回一个默认值或者错误

总结

context 包就是为了解决上面所说的这些问题而开发的：在一组 goroutine 之间传递共享的值、取消信号、**deadline**.....



3. 用法样例

3.1 传递共享的数据

对于 Web 服务端开发，往往希望将一个请求处理的整个过程串起来，如通过在日志中加入 traceId 是很常见的做法，可通过传递 context 的方式来实现这一点

携带 traceId 样例：

```
package main

import (
    "context"
    "fmt"
)

func main() {
    // 生成一个初始 context
    ctx := context.Background()
    process(ctx)

    // contextWithValue 基于已有 context 和指定要携带的 key-value 生成新的 context，将在下一章原理解析部分详细说明
    // 这里用 string traceId 作为 key，不是最佳实践，详情见下面的章节
    ctx = context.WithValue(ctx, "traceId", "123456")
    go process(ctx)

    time.Sleep(2 * time.Second)
    fmt.Println("main: bye bye!")
}

func process(ctx context.Context) {
    traceId, ok := ctx.Value("traceId").(string)
    if ok {
        fmt.Printf("process over. trace_id=%s\n", traceId)
    } else {
        fmt.Printf("process over. no trace_id\n")
    }
}
```

运行结果：

```
process over. no trace_id
process over. trace_id=123456
```

3.2 取消 goroutine

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 goroutine 去做一些事情，这就导致我们会在一次请求中开了多个 goroutine 却无法控制他们

代码样例：

```
package main

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

func main() {
    // 返回取消方法cancel
    ctx, cancel := context.WithCancel(context.Background())
    go Speak(ctx)
    time.Sleep(10 * time.Second)
    cancel()

    time.Sleep(2 * time.Second)
    fmt.Println("main: bye bye!")
}

func Speak(ctx context.Context) {
    for range time.Tick(time.Second) {
        select {
        // 要监听ctx.Done()返回的channel中的数据，当有数据被读到说明context被取消了
        case <-ctx.Done():
            fmt.Println("Speak: bye bye!")
            return
        default:
            fmt.Println("hello")
        }
    }
}
```

运行结果：

```
hello
Speak: bye bye!
main: bye bye!
```

3.3 超时取消

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

代码样例：

```
package main

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

func main() {
    // 设置3秒后超时，同时返回了cancel方法，支持提前取消
    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    // 建议在自动取消后也调用cancelFunc去停止定时任务减少不必要的资源浪费
    defer cancel()
    go deal(ctx)

    time.Sleep(5 * time.Second)
    fmt.Println("main: bye bye!")
}

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)
        }
    }
}
```

程序输出：

```
deal time is 0
deal time is 1
context deadline exceeded
```

4. 原理解析

4.1 context创建方式

context 包主要提供了两种方式创建根 context：

- `context.Background()`
- `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：

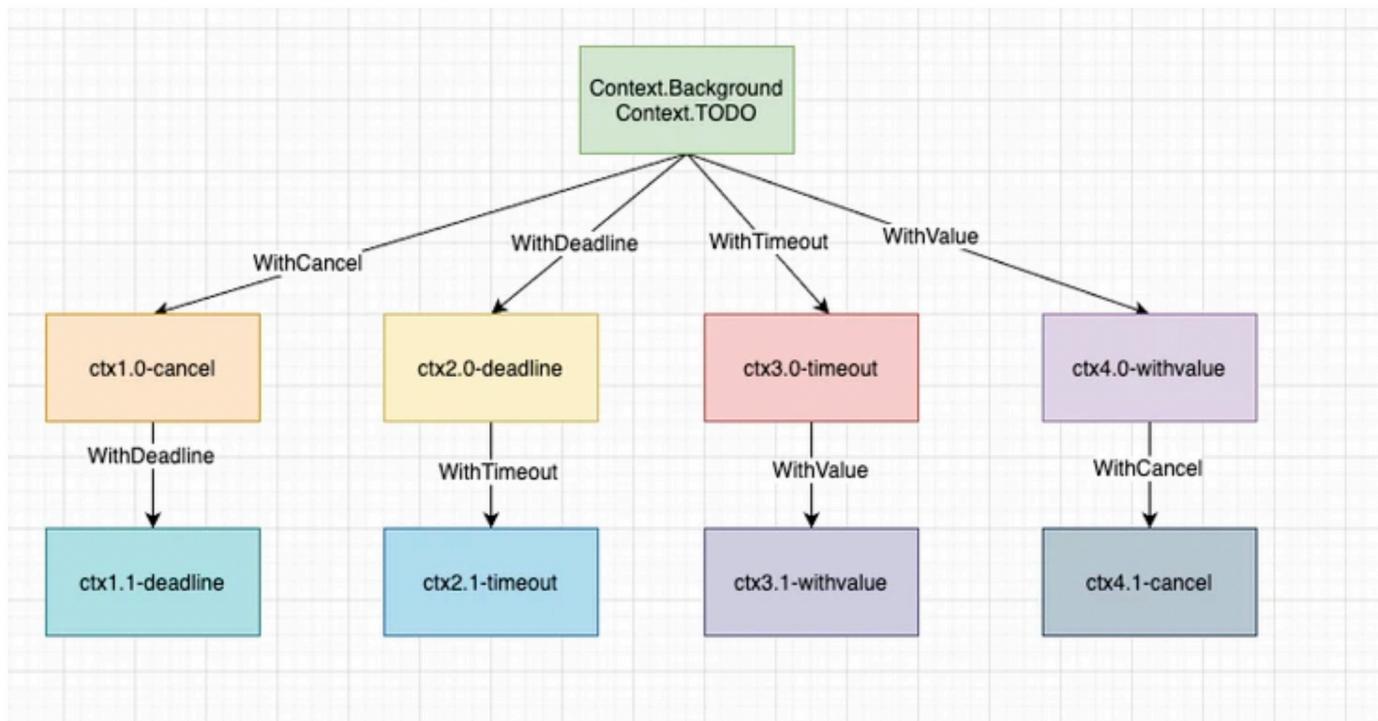
- `context.Background` 是上下文的默认值，所有其他的上下文都应该从它衍生（Derived）出来。
- `context.TODO` 应该只在不确定应该使用哪种上下文时使用；

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，比如在写单元测试时。

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：



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，当父 `context` 被取消时，子 `context` 也会被取消。

4.2 源码赏析

`Context` 其实就是一个接口，定义了四个方法：

```

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

```

1. `Deadline` ctx 如果在某个时间点关闭的话，返回该值。否则 `ok` 为 `false`
2. `Done` 返回一个 channel, 如果超时或是取消就会被关闭，实现消息通讯
3. `Err` 如果当前 ctx 超时或被取消了，那么 `Err` 返回错误
4. `Value` 根据某个 key 返回对应的 value, 功能类似字典

这个接口主要被三个类继承实现，分别是 `emptyCtx`、`ValueCtx`、`cancelCtx`，采用匿名接口的写法，这样可以对任意实现了该接口的类型进行重写。

4.2.1 emptyCtx

空上下文，通用用于初始化一个context

```

// An emptyCtx is never canceled, has no values, and has no deadline. It is not
// struct{}, since vars of this type must have distinct addresses.
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
}

func (e *emptyCtx) String() string {
    switch e {
    case background:
        return "context.Background"
    case todo:
        return "context.TODO"
    }
    return "unknown empty Context"
}

```

`emptyCtx` 主要是给我们创建根 `Context` 时使用的

```
var (
    background = new(emptyCtx)
    todo       = new(emptyCtx)
)

// Background returns a non-nil, empty Context. It is never canceled, has no
// values, and has no deadline. It is typically used by the main function,
// initialization, and tests, and as the top-level Context for incoming
// requests.
func Background() Context {
    return background
}

// TODO returns a non-nil, empty Context. Code should use context.TODO when
// it's unclear which Context to use or it is not yet available (because the
// surrounding function has not yet been extended to accept a Context
// parameter).
func TODO() Context {
    return todo
}
```

`Background` 和 `TODO` 还是一模一样的，官方说：`background` 它通常由主函数、初始化和测试使用，并作为传入请求的顶级上下文；`TODO` 是当不清楚要使用哪个 `Context` 或尚不可用时，代码应使用 `context.TODO`，后续再进行替换掉，归根结底就是语义不同而已。

4.2.2 valueCtx

`valueCtx` 目的就是为 `Context` 携带键值对，`context` 是一个继承结构，会一直往上找 `key` 对应的 `value`

```
// A valueCtx carries a key-value pair. It implements Value for that key and
// delegates all other calls to the embedded Context.
type valueCtx struct {
    Context
    key, val interface{}
}

// stringify tries a bit to stringify v, without using fmt, since we don't
// want context depending on the unicode tables. This is only used by
// *valueCtx.String().
func stringify(v interface{}) string {
    switch s := v.(type) {
    case Stringer:
        return s.String()
    case String:
        return s
    }
```

```

    return "<not Stringer>"
}

func (c *valueCtx) String() string {
    return contextName(c.Context) + ".WithValue(type " +
        reflectlite.TypeOf(c.key).String() +
        ", val " + stringify(c.val) + ")"
}

func (c *valueCtx) Value(key interface{}) interface{} {
    if c.key == key {
        return c.val
    }
    // 从这里也可以知道，context是一个继承结构，会一直往上找key对应的value
    return c.Context.Value(key)
}

```

可使用WithValue方法得到valueCtx对象

注意看方法注释：

1. context values用于存放请求范围的数据，不要用于传递方法的可选参数
2. key必须是可比较的类型，并且不应该是string或者其他内置的类型，以避免key冲突，用户应该自定义类型作为key
3. key需要定义为具体的类型struct{}

```

// WithValue returns a copy of parent in which the value associated with key is
// val.
//
// Use context Values only for request-scoped data that transits processes and
// APIs, not for passing optional parameters to functions.
//
// The provided key must be comparable and should not be of type
// string or any other built-in type to avoid collisions between
// packages using context. Users of WithValue should define their own
// types for keys. To avoid allocating when assigning to an
// interface{}, context keys often have concrete type
// struct{}. Alternatively, exported context key variables' static
// type should be a pointer or interface.
func WithValue(parent Context, key, val interface{}) Context {
    if parent == nil {
        panic("cannot create context from nil parent")
    }
    if key == nil {
        panic("nil key")
    }
    // key必须是可比较的类型，因为value方法中需要用==进行对比
    if !reflectlite.TypeOf(key).Comparable() {
        panic("key is not comparable")
    }
}

```

```
    }
    return &valueCtx{parent, key, val}
}
```

正确使用valueCtx的样例：

```
package cache

import (
    "context"
)

type cacheKey struct{}


// FromContext returns cache from context
func FromContext(ctx context.Context) (Cache, bool) {
    c, ok := ctx.Value(cacheKey{}).(Cache)
    return c, ok
}

// NewContext returns new context with cache
func NewContext(ctx context.Context, c Cache) context.Context {
    if ctx == nil {
        ctx = context.Background()
    }
    return context.WithValue(ctx, cacheKey{}, c)
}
```

注意：cacheKey类型不要暴露出去，防止其他人拿到后可以覆盖context中对应的value

Horizon的错误用法：

```
package orm

import (
    "context"

    "errors \"g.hz.netease.com/horizon/core/errors\""
    "perror \"g.hz.netease.com/horizon/pkg/errors\""

    "gorm.io/gorm"
)

const ormKey = "ORM"

func Key() string {
    return ormKey
}
```

```

// FromContext returns orm from context
func FromContext(ctx context.Context) (*gorm.DB, error) {
    o, ok := ctx.Value(ormKey).(*gorm.DB)
    if !ok {
        return nil, perror.Wrap(errors.ErrFailedToGetORM, "cannot get the ORM from
context")
    }
    return o, nil
}

// NewContext returns new context with orm
func NewContext(ctx context.Context, o *gorm.DB) context.Context {
    if ctx == nil {
        ctx = context.Background()
    }
    return context.WithValue(ctx, ormKey, o) // nolint
}

```

4.2.3 cancelCtx

```

// A cancelCtx can be canceled. When canceled, it also cancels any children
// that implement canceler.
type cancelCtx struct {
    Context

    mu      sync.Mutex          // protects following fields
    done    atomic.Value         // of chan struct{}, created lazily, closed by first
cancel call
    children map[canceler]struct{} // set to nil by the first cancel call
    err     error                // set to non-nil by the first cancel call
}

```

字短解释：

- `mu`：就是一个互斥锁，保证并发安全的，所以 `context` 是并发安全的
- `done`：用来做 `context` 的取消通知信号，之前的版本使用的是 `chan struct{}` 类型，现在用 `atomic.Value` 做锁优化
- `children`：`key` 是接口类型 `canceler`，目的就是存储实现当前 `canceler` 接口的子节点，当根节点发生取消时，遍历子节点发送取消信号
- `error`：当 `context` 取消时存储取消信息

Done方法说明：

`cancelCtx`实现了 `Done` 方法，`Done()` 返回一个 channel，可以表示 `context` 被取消的信号：当这个 channel 被关闭时，说明 `context` 被取消了。注意，这是一个只读的channel。我们又知道，读一个关闭的 channel 会读出相应类型的零值。并且源码里没有地方会向这个 channel 里面塞入值。换句话说，这是一个 `receive-only` 的 channel。因此在子协程里读这个 channel，除非被关闭，否则读不出来任何东西。也正是利用了这一点，子协程从 channel 里读出了值（零值）后，就可以做一些收尾工作，尽快退出。

```

func (c *cancelCtx) Done() <-chan struct{} {
    d := c.done.Load()
    if d != nil {
        return d.(chan struct{})
    }
    c.mu.Lock()
    defer c.mu.Unlock()
    d = c.done.Load()
    if d == nil {
        d = make(chan struct{})
        c.done.Store(d)
    }
    return d.(chan struct{})
}

```

可通过WithCancel创建cancelCtx:

```

func WithCancel(parent Context) (ctx Context, cancel CancelFunc) {
    if parent == nil {
        panic("cannot create context from nil parent")
    }
    c := newCancelCtx(parent)
    // 重点是这个方法，构建了父子Context之间的关联，即child引用了parent，parent也会引用child
    // 实现了当父context取消时，取消掉所有的子context
    propagateCancel(parent, &c)
    return &c, func() { c.cancel(true, Canceled) }
}

// newCancelCtx returns an initialized cancelCtx.
func newCancelCtx(parent Context) cancelCtx {
    return cancelCtx{Context: parent}
}

```

4.2.3 withDeadline、WithTimeout

`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`。

```

func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc) {
    return WithDeadline(parent, time.Now().Add(timeout))
}

```

重点来看`WithDeadline`是如何实现的：

```

func WithDeadline(parent Context, d time.Time) (Context, CancelFunc) {
    // 不能为空`context`创建衍生context
    if parent == nil {

```

```

        panic("cannot create context from nil parent")
    }

// 当父context的结束时间早于要设置的时间，则不需要再去单独处理子节点的定时器了
if cur, ok := parent.Deadline(); ok && cur.Before(d) {
    // The current deadline is already sooner than the new one.
    return WithCancel(parent)
}

// 创建一个timerCtx对象
c := &timerCtx{
    cancelCtx: newCancelCtx(parent),
    deadline:  d,
}
// 将当前节点挂到父节点上
propagateCancel(parent, c)

// 获取过期时间
dur := time.Until(d)
// 当前时间已经过期了则直接取消
if dur <= 0 {
    c.cancel(true, DeadlineExceeded) // deadline has already passed
    return c, func() { c.cancel(false, Canceled) }
}
c.mu.Lock()
defer c.mu.Unlock()
// 如果没被取消，则直接添加一个定时器，定时去取消
if c.err == nil {
    c.timer = time.AfterFunc(dur, func() {
        c.cancel(true, DeadlineExceeded)
    })
}
return c, func() { c.cancel(true, Canceled) }
}

```

`withDeadline`相较于`WithCancel`方法也就多了一个定时器去定时调用`cancel`方法，这个`cancel`方法在`timerCtx`类中进行了重写，我们先来看一下`timerCtx`类，他是基于`cancelCtx`的，多了两个字段：

```

type timerCtx struct {
    cancelCtx
    timer *time.Timer // Under cancelCtx.mu.

    deadline time.Time
}

// 重写了cancel方法
func (c *timerCtx) cancel(removeFromParent bool, err error) {
    // 调用cancelCtx的cancel方法取消掉子节点context
    c.cancelCtx.cancel(false, err)
    // 从父context移除放到了这里来做
}

```

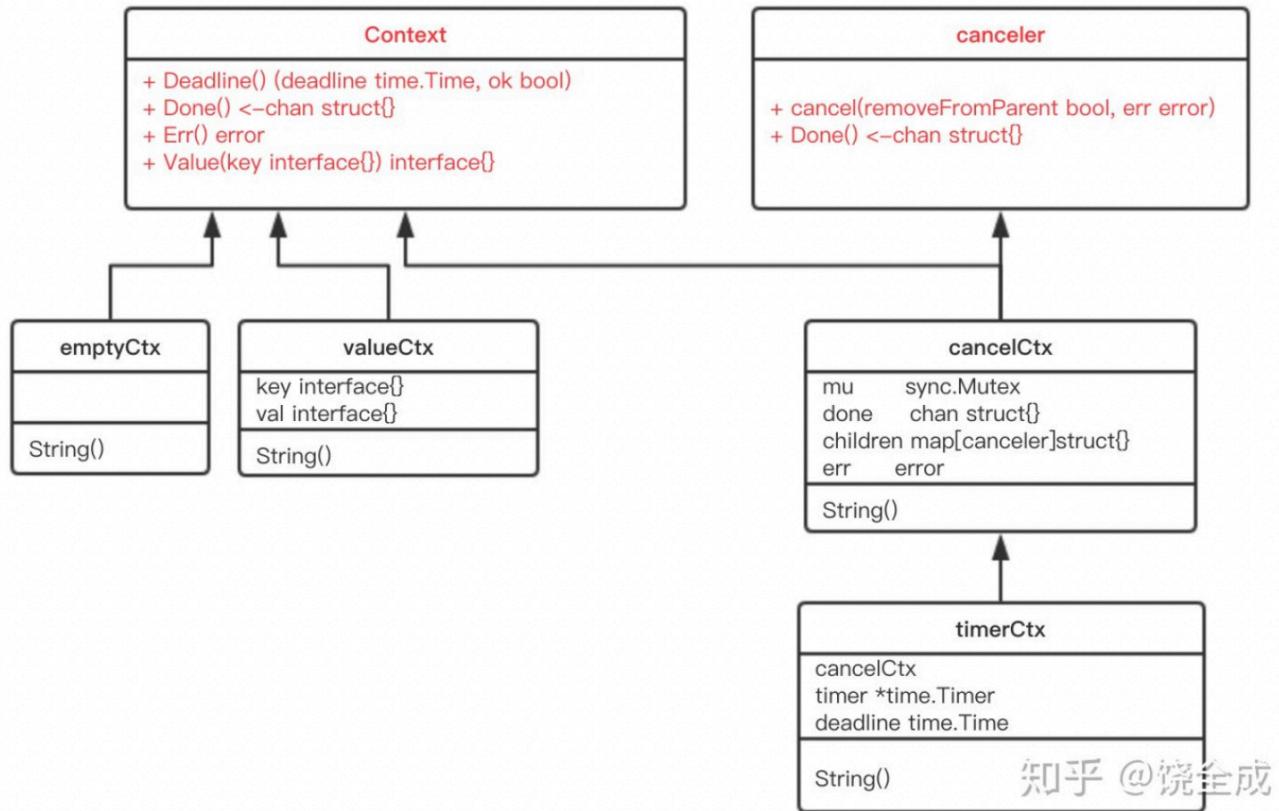
```

if removeFromParent {
    // Remove this timerCtx from its parent cancelCtx's children.
    removeChild(c.cancelCtx.Context, c)
}

// 停掉定时器，释放资源
c.mu.Lock()
if c.timer != nil {
    c.timer.Stop()
    c.timer = nil
}
c.mu.Unlock()
}

```

4.3 源码总结



方法说明：

类型	名称	作用
Context	接口	定义了 Context 接口的四个方法
emptyCtx	结构体	实现了 Context 接口，它其实是个空的 context
CancelFunc	函数	取消函数
canceled	接口	context 取消接口，定义了两个方法
cancelCtx	结构体	可以被取消
timerCtx	结构体	超时会被取消
valueCtx	结构体	可以存储 k-v 对
Background	函数	返回一个空的 context，常作为根 context
TODO	函数	返回一个空的 context，常用于重构时期，没有合适的 context 可用
WithCancel	函数	基于父 context，生成一个可以取消的 context
newCancelCtx	函数	创建一个可取消的 context
propagateCancel	函数	向下传递 context 节点间的取消关系
parentCancelCtx	函数	找到第一个可取消的父节点
removeChild	函数	去掉父节点的孩子节点
init	函数	包初始化
WithDeadline	函数	创建一个有 deadline 的 context
WithTimeout	函数	创建一个有 timeout 的 context
WithValue	函数	创建一个存储 k-v 对的 context

知乎 @饶全成

5. 最佳实践

5.1 优缺点

缺点：

- 影响代码美观，现在基本所有 web 框架、RPC 框架都是实现了 context，这就导致我们的代码中每一个函数的一个参数都是 context，即使不用也要带着这个参数透传下去
- context 可以携带值，但是没有任何限制，类型和大小都没有限制，也就是没有任何约束，这样很容易导致滥用，程序的健壮很难保证；还有一个问题是通过 context 携带值不如显式传值舒服，可读性变差了。
- context 取消和自动取消的错误返回不够友好，无法自定义错误，出现难以排查的问题时不好排查。
- 创建衍生节点实际是创建一个个链表节点，其时间复杂度为 O(n)，节点多了会导致效率变低。

优点：

- `context` 的携带者功能没有任何限制，这样我们传递任何的数据，可以说这是一把双刃剑
- `context` 包解决了 `goroutine` 的 `cancelation` 问题

5.2 最佳实践

1. 除了框架层不要使用 `WithValue` 携带业务数据，key、value类型都是 `interface{}`，编译期无法确定，运行期需要断言，有性能和安全问题
2. 一定不要打印 `context`，尤其是从 http 标准库派生出来的，谁知道里面存了什么
3. `context` 做为第一个参数传给函数，而不是当成结构体的成员字段来使用。作为参数由调用者决定 `context` 的类型，作为结构体的成员，那么所有方法都共享这同一个 `context`
4. 异步 goroutine 逻辑使用 `context` 时要清楚谁还持有，会不会提前超时
5. 派生出来的 `child ctx` 一定要配合 `defer cancel()` 使用，释放资源

6. 评价

使用 `context` 带来的好处如上述章节所说，但也有另一种声音评价 `context`

在参考资料《Context should go away for Go 2》这篇英文博客里，作者甚至调侃说：如果要把 Go 标准库的大部分函数都加上 `context` 参数的话，

就给我来一枪吧！put a bullet in my head, please

为了表达自己对 `context` 并没有什么好感，作者接着又说了一句：

If you use `ctx.Value` in my (non-existent) company, you're fired.

简直太幽默了，哈哈

7. 参考资料

- 【1】 Context should go away for Go 2: <https://faiface.github.io/post/context-should-go-away-go2/>
- 【2】 也许是 Go Context 最佳实践：<https://mp.weixin.qq.com/s/8iBQWs7LAXEMCoXc5PpSaQ>
- 【3】 小白也能看懂的 `context` 包详解：从入门到精通：<https://segmentfault.com/a/1190000040917752>
- 【4】 深度解密 Go 语言之 `context`：<https://zhuanlan.zhihu.com/p/68792989>