



Contextual Talk. Разбираемся в устройстве пакета **context**



Олег Сидоренков
Разработчик информационных систем
OzonTech



Привет!

Я – Олег,

Go-разработчик в команде селлерских акций.

Люблю разбираться, как устроены различные инструменты под капотом

11:25 ✓

AGENDA

01 Функционал контекста



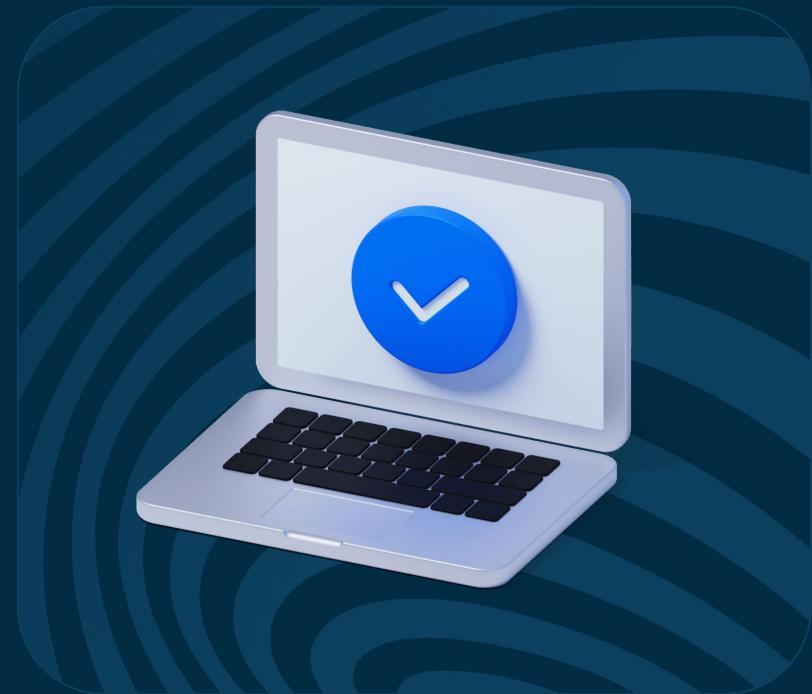
02 История появления



03 Виды контекстов и их реализация



04 Best Practices



01



Функционал
контекста



Context

```
1 type Context interface {
2     // Deadline returns context's deadline.
3     Deadline() (deadline time.Time, ok bool)
4     // Done returns channel which locks until the context is finished.
5     // The types of finish usually are:
6     // 1. Context has been cancelled;
7     // 2. Context's deadline has been exceeded;
8     Done() <-chan struct{}
9     // Err returns cause of context finish.
10    Err() error
11    // Returns some stored value.
12    Value(key any) any
13 }
```



Получение дедлайна

Context

```
1 type Context interface {
2     // Deadline returns context's deadline.
3     Deadline() (deadline time.Time, ok bool)
4     // Done returns channel which locks until the context is finished.
5     // The types of finish usually are:
6     // 1. Context has been cancelled;
7     // 2. Context's deadline has been exceeded;
8     Done() <-chan struct{}
9     // Err returns cause of context finish.
10    Err() error
11    // Returns some stored value.
12    Value(key any) any
13 }
```

➡ Получение дедлайна

➡ Синхронизация
при помощи каналов

Context

```
1 type Context interface {
2     // Deadline returns context's deadline.
3     Deadline() (deadline time.Time, ok bool)
4     // Done returns channel which locks until the context is finished.
5     // The types of finish usually are:
6     // 1. Context has been cancelled;
7     // 2. Context's deadline has been exceeded;
8     Done() <-chan struct{}
9     // Err returns cause of context finish.
10    Err() error
11    // Returns some stored value.
12    Value(key any) any
13 }
```

→ Получение дедлайна

→ Синхронизация
при помощи каналов

→ Получение причины завершения

Примеры: отмена, истечение времени

Context

```
1 type Context interface {
2     // Deadline returns context's deadline.
3     Deadline() (deadline time.Time, ok bool)
4     // Done returns channel which locks until the context is finished.
5     // The types of finish usually are:
6     // 1. Context has been cancelled;
7     // 2. Context's deadline has been exceeded;
8     Done() <-chan struct{}
9     // Err returns cause of context finish.
10    Err() error
11    // Returns some stored value.
12    Value(key any) any
13 }
```

→ Получение дедлайна

→ Синхронизация при помощи каналов

→ Получение причины завершения

Примеры: отмена, истечение времени

→ Хранение произвольных данных

Примеры: HTTP-заголовки, логгеры, иные метаданные

02



История появления



Раньше

```
1  func longOperationContext(ctx context.Context) error {
2      for i := 0; i < bigNumber; i++ {
3          select {
4              case <-ctx.Done():
5                  return errors.New("deadline")
6              default:
7                  }
8          }
9      return nil
10 }
11
12 func ntfr(timeout time.Duration) {
13     notifier := make(chan struct{})
14     go func() {
15         time.Sleep(timeout)
16         notifier <- struct{}{}
17     }()
18     fmt.Println(longOperationNotifier(notifier))
19 }
```

До появления контекста использовались каналы, сигнализирующие о завершении



Сейчас

```
1 func longOperationContext(ctx context.Context) error {
2     for i := 0; i < bigNumber; i++ {
3         select {
4             case <-ctx.Done():
5                 return errors.New("deadline")
6             default:
7                 }
8         }
9     return nil
10 }
11
12 func ctx(timeout time.Duration) {
13     ctx, cancel := context.WithTimeout(context.Background(), timeout)
14     defer cancel()
15     fmt.Println(longOperationContext(ctx))
16 }
```

В 2014

Появился пакет golang.org/x/net/context
Стал очень популярен, особенно среди
инженеров Google

Сейчас

```
1 func longOperationContext(ctx context.Context) error {
2     for i := 0; i < bigNumber; i++ {
3         select {
4             case <-ctx.Done():
5                 return errors.New("deadline")
6             default:
7                 }
8         }
9     return nil
10 }
11
12 func ctx(timeout time.Duration) {
13     ctx, cancel := context.WithTimeout(context.Background(), timeout)
14     defer cancel()
15     fmt.Println(longOperationContext(ctx))
16 }
```

В 2014

Появился пакет golang.org/x/net/context
Стал очень популярен, особенно среди инженеров Google

В 2016

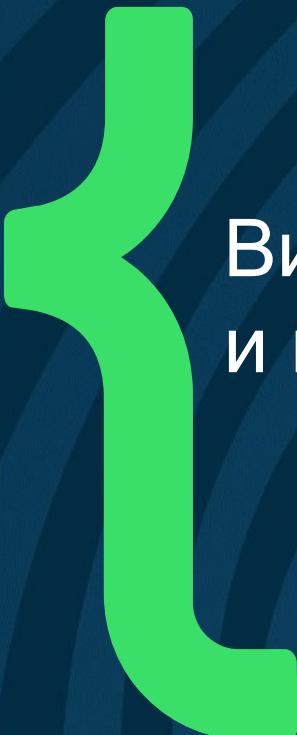
Был добавлен в STL в версии Go 1.7

История появления



Получается, контекст - это
просто удобная обёртка
над каналами?

03



Виды контекстов и их реализация

Empty Context

Context With Cancel

Context Without
Cancel

After Func

Context With
Deadline

Context With
Timeout

Cause

Context With Value

Empty Context

```
1 type emptyContext struct{}
2
3 func (emptyContext) Deadline() (deadline time.Time, ok bool) {
4     return time.Time{}, false
5 }
6
7 func (emptyContext) Done() <-chan struct{} {
8     return nil
9 }
10
11 func (emptyContext) Err() error {
12     return nil
13 }
14
15 func (emptyContext) Value(key any) any {
16     return nil
17 }
```

Самая простая реализация
во всём пакете



Empty Context

Основные назначения:

Плейсхолдер (context.TODO)

Корневая нода (context.Background)

Cancel Context

```
1 type cancelCtx struct {
2     // Parent context.
3     Context
4     // Chan struct.
5     done atomic.Value
6     // Omit mutex for simplicity.
7     children map[canceler]struct{}
8     err, cause string
9 }
10
11 type canceler interface {
12     cancel(removeFromParent bool, err, cause error)
13     Done() <-chan struct{}
14 }
```

Cancel Context включает **родителя**, **канал для done** и **набор потомков**, которых можно отменить, а также информацию об отмене



Cancel Context

```
1 var closedchan = make(chan struct{})  
2  
3 func init() {  
4     close(closedchan)  
5 }  
6  
7 var cancelCtxKey int
```

Существуют 2 магических значения:

1. Заранее закрытый канал
2. Переменная int с дефолтным значением

Cancel Context

```
1  func (c *cancelCtx) Done() <-chan struct{} {
2      d := c.done.Load()
3      if d == nil {
4          d = make(chan struct{})
5          c.done.Store(d)
6      }
7      return d.(chan struct{})
8  }
9
10 func (c *cancelCtx) Err() error {
11     return c.err
12 }
13
14 func (c *cancelCtx) Value(key any) any {
15     if key == &cancelCtxKey {
16         return c
17     }
18     return value(c.Context, key)
19 }
```

➡ Deadline возвращается
от родителя

Cancel Context

```
1 func (c *cancelCtx) Done() <-chan struct{} {
2     d := c.done.Load()
3     if d == nil {
4         d = make(chan struct{})
5         c.done.Store(d)
6     }
7     return d.(chan struct{})
8 }
9
10 func (c *cancelCtx) Err() error {
11     return c.err
12 }
13
14 func (c *cancelCtx) Value(key any) any {
15     if key == &cancelCtxKey {
16         return c
17     }
18     return value(c.Context, key)
19 }
```

➡ **Deadline** возвращается от родителя

➡ **Done** инициализируется лениво

Cancel Context

```
1  func (c *cancelCtx) Done() <-chan struct{} {
2      d := c.done.Load()
3      if d == nil {
4          d = make(chan struct{})
5          c.done.Store(d)
6      }
7      return d.(chan struct{})
8  }
9
10 func (c *cancelCtx) Err() error {
11     return c.err
12 }
13
14 func (c *cancelCtx) Value(key any) any {
15     if key == &cancelCtxKey {
16         return c
17     }
18     return value(c.Context, key)
19 }
```

➡ **Deadline** возвращается от родителя

➡ **Done** инициализируется лениво

➡ **Err** просто возвращает сохранённое значение

Cancel Context

```
1  func (c *cancelCtx) Done() <-chan struct{} {
2      d := c.done.Load()
3      if d == nil {
4          d = make(chan struct{})
5          c.done.Store(d)
6      }
7      return d.(chan struct{})
8  }
9
10 func (c *cancelCtx) Err() error {
11     return c.err
12 }
13
14 func (c *cancelCtx) Value(key any) any {
15     if key == &cancelCtxKey {
16         return c
17     }
18     return value(c.Context, key)
19 }
```

➡ **Deadline** возвращается от родителя

➡ **Done** инициализируется лениво

➡ **Err** просто возвращает сохранённое значение

➡ **Value** может возвращать сам контекст при получении магического значения

Cancel Context

```
1 func WithCancel(parent Context) (
2     ctx Context,
3     cancel CancelFunc,
4 ) {
5     c := &cancelCtx{}
6     c.propagateCancel(parent, c)
7
8     return c, func() {
9         c.cancel(true, Canceled, nil)
10    }
11 }
```

При инициализации происходит 2 основных действия:

1. «Распространение» отмены
2. Создание контекста и ручки отмены

Cancel Context

Распространение добавляет вызов cancel для потомков



```
1  func (c *cancelCtx) propagateCancel(parent Context, child canceler) {
2      c.Context = parent
3
4      // Check for parent valid cancellation ...
5
6      if p, ok := parentCancelCtx(parent); ok {
7          // Parent is a *cancelCtx, or derives from one.
8          if p.err != nil {
9              // Parent has already been canceled.
10             child.cancel(false, p.err, p.cause)
11         } else {
12             p.children[child] = struct{}{}
13         }
14     }
15     return
16 }
17
18 if a, ok := parent.(afterFuncer); ok {
19     // parent implements an AfterFunc method.
20     stop := a.AfterFunc(func() {
21         child.cancel(false, parent.Err(), Cause(parent))
22     })
23     c.Context = stopCtx{
24         Context: parent,
25         stop:    stop,
26     }
27     return
28 }
29
30 go func() {
31     select {
32         case <-parent.Done():
33             child.cancel(false, parent.Err(), Cause(parent))
34         case <-child.Done():
35     }
36 }
```

Cancel Context

Распространение добавляет вызов cancel для потомков



```
1  func (c *cancelCtx) propagateCancel(parent Context, child canceler) {
2      c.Context = parent
3
4      // Check for parent valid cancellation ...
5
6      if p, ok := parentCancelCtx(parent); ok {
7          // Parent is a *cancelCtx, or derives from one.
8          if p.err != nil {
9              // Parent has already been canceled.
10             child.cancel(false, p.err, p.cause)
11         } else {
12             p.children[child] = struct{}{}
13         }
14     }
15
16     if a, ok := parent.(afterFuncer); ok {
17         // parent implements an AfterFunc method.
18         stop := a.AfterFunc(func() {
19             child.cancel(false, parent.Err(), Cause(parent))
20         })
21         c.Context = stopCtx{
22             Context: parent,
23             stop:    stop,
24         }
25     }
26
27     return
28 }
29
30     go func() {
31         select {
32             case <-parent.Done():
33                 child.cancel(false, parent.Err(), Cause(parent))
34             case <-child.Done():
35         }
36     }()
37 }
```

Cancel Context

Распространение добавляет вызов cancel для потомков



```
1  func (c *cancelCtx) propagateCancel(parent Context, child canceler) {
2      c.Context = parent
3
4      // Check for parent valid cancellation ...
5
6      if p, ok := parentCancelCtx(parent); ok {
7          // Parent is a *cancelCtx, or derives from one.
8          if p.err != nil {
9              // Parent has already been canceled.
10             child.cancel(false, p.err, p.cause)
11         } else {
12             p.children[child] = struct{}{}
13         }
14     }
15     return
16 }
17
18 if a, ok := parent.(afterFuncer); ok {
19     // parent implements an AfterFunc method.
20     stop := a.AfterFunc(func() {
21         child.cancel(false, parent.Err(), Cause(parent))
22     })
23     c.Context = stopCtx{
24         Context: parent,
25         stop:    stop,
26     }
27     return
28 }
29
30 go func() {
31     select {
32         case <-parent.Done():
33             child.cancel(false, parent.Err(), Cause(parent))
34         case <-child.Done():
35     }
36 }
```

Cancel Context

```
1 func (c *cancelCtx) cancel(removeFromParent bool, err error, cause error) {
2     if c.err != nil {
3         // Already cancelled.
4         return
5     }
6     c.err = err
7     c.cause = cause
8
9     d, _ := c.done.Load().(chan struct{})
10    if d == nil {
11        c.done.Store(closedchan)
12    } else {
13        close(d)
14    }
15
16    for child := range c.children {
17        child.cancel(false, err, cause)
18    }
19    c.children = nil
20
21    if removeFromParent {
22        removeChild(c.Context, c)
23    }
24 }
```



Происходит отмена как самого контекста, так и его ПОТОМКОВ

Cancel Context

```
1 func (c *cancelCtx) cancel(removeFromParent bool, err error, cause error) {
2     if c.err != nil {
3         // Already cancelled.
4         return
5     }
6     c.err = err
7     c.cause = cause
8
9     d, _ := c.done.Load().(chan struct{})
10    if d == nil {
11        c.done.Store(closedchan)
12    } else {
13        close(d)
14    }
15
16    for child := range c.children {
17        child.cancel(false, err, cause)
18    }
19    c.children = nil
20
21    if removeFromParent {
22        removeChild(c.Context, c)
23    }
24 }
```

➡ Происходит отмена как самого контекста, так и его ПОТОМКОВ

➡ Предусмотрено поле `removeFromParent`, чтобы предотвратить множественные удаления

Cancel Context

Основные назначения:

Отмена запросов

Освобождение ресурсов

Context without Cancel

```
1 type withoutCancelCtx struct {
2     c Context
3 }
4
5 func WithoutCancel(parent Context) Context {
6     return withoutCancelCtx{parent}
7 }
8
9 func (withoutCancelCtx) Deadline() (deadline time.Time, ok bool) {
10    return
11 }
12
13 func (withoutCancelCtx) Done() <-chan struct{} {
14     return nil
15 }
16
17 func (withoutCancelCtx) Err() error {
18     return nil
19 }
20
21 func (c withoutCancelCtx) Value(key any) any {
22     return value(c, key)
23 }
```

Without Cancel просто «забывает»
о своём завершении



Context without Cancel

Основные назначения:

Создание нового корневого контекста
без потери метаданных
(За исключением Cause)

AfterFunc

```
1 func AfterFunc(ctx Context, f func()) (stop func() bool) {
2     a := &afterFuncCtx{
3         f: f,
4     }
5     a.cancelCtx.propagateCancel(ctx, a)
6     return func() bool {
7         stopped := false
8         a.once.Do(func() {
9             stopped = true
10        })
11        if stopped {
12            a.cancel(true, Canceled, nil)
13        }
14        return stopped
15    }
16 }
17
18 type afterFuncer interface {
19     AfterFunc(func()) func() bool
20 }
21
22 type afterFuncCtx struct {
23     cancelCtx
24     // Either starts running f or stops f from running.
25     once sync.Once
26     f   func()
27 }
28
29 func (a *afterFuncCtx) cancel(removeFromParent bool, err, cause error) {
30     a.cancelCtx.cancel(false, err, cause)
31     if removeFromParent {
32         removeChild(a.Context, a)
33     }
34     a.once.Do(func() {
35         go a.f()
36     })
37 }
```



Редко встречается из-за узконаправленности и новизны.

Появился в 2023 году на версии Go 1.21

AfterFunc

```
1 func AfterFunc(ctx Context, f func()) (stop func() bool) {
2     a := &afterFuncCtx{
3         f: f,
4     }
5     a.cancelCtx.propagateCancel(ctx, a)
6     return func() bool {
7         stopped := false
8         a.once.Do(func() {
9             stopped = true
10        })
11        if stopped {
12            a.cancel(true, Canceled, nil)
13        }
14        return stopped
15    }
16 }
17
18 type afterFuncer interface {
19     AfterFunc(func()) func() bool
20 }
21
22 type afterFuncCtx struct {
23     cancelCtx
24     // Either starts running f or stops f from running.
25     once sync.Once
26     f   func()
27 }
28
29 func (a *afterFuncCtx) cancel(removeFromParent bool, err, cause error) {
30     a.cancelCtx.cancel(false, err, cause)
31     if removeFromParent {
32         removeChild(a.Context, a)
33     }
34     a.once.Do(func() {
35         go a.f()
36     })
37 }
```

- ➡ Редко встречается из-за узконаправленности и новизны.

Появился в 2023 году на версии Go 1.21

- ➡ Благодаря механизмам синхронизации обещает, что функция f либо исполнится ровно 1 раз, либо будет отменена

AfterFunc

```
1 func AfterFunc(ctx Context, f func()) (stop func() bool) {
2     a := &afterFuncCtx{
3         f: f,
4     }
5     a.cancelCtx.propagateCancel(ctx, a)
6     return func() bool {
7         stopped := false
8         a.once.Do(func() {
9             stopped = true
10        })
11        if stopped {
12            a.cancel(true, Canceled, nil)
13        }
14        return stopped
15    }
16}
17
18 type afterFuncer interface {
19     AfterFunc(func()) func() bool
20 }
21
22 type afterFuncCtx struct {
23     cancelCtx
24     // Either starts running f or stops f from running.
25     once sync.Once
26     f   func()
27 }
28
29 func (a *afterFuncCtx) cancel(removeFromParent bool, err, cause error) {
30     a.cancelCtx.cancel(false, err, cause)
31     if removeFromParent {
32         removeChild(a.Context, a)
33     }
34     a.once.Do(func() {
35         go a.f()
36     })
37 }
```

- ➡ Редко встречается из-за узконаправленности и новизны.

Появился в 2023 году на версии Go 1.21

- ➡ Благодаря механизмам синхронизации обещает, что функция f либо исполнится ровно 1 раз, либо будет отменена

AfterFunc

```
1 func AfterFunc(ctx Context, f func()) (stop func() bool) {
2     a := &afterFuncCtx{
3         f: f,
4     }
5     a.cancelCtx.propagateCancel(ctx, a)
6     return func() bool {
7         stopped := false
8         a.once.Do(func() {
9             stopped = true
10        })
11        if stopped {
12            a.cancel(true, Canceled, nil)
13        }
14        return stopped
15    }
16 }
17
18 type afterFuncer interface {
19     AfterFunc(func()) func() bool
20 }
21
22 type afterFuncCtx struct {
23     cancelCtx
24     // Either starts running f or stops f from running.
25     once sync.Once
26     f   func()
27 }
28
29 func (a *afterFuncCtx) cancel(removeFromParent bool, err, cause error) {
30     a.cancelCtx.cancel(false, err, cause)
31     if removeFromParent {
32         removeChild(a.Context, a)
33     }
34     a.once.Do(func() {
35         go a.f()
36     })
37 }
```

- ➡ Редко встречается из-за узконаправленности и новизны.

Появился в 2023 году на версии Go 1.21

- ➡ Благодаря механизмам синхронизации обещает, что функция f либо исполнится ровно 1 раз, либо будет отменена

AfterFunc

```
1 func read(ctx context.Context, conn net.Conn, b []byte) (n int, err error) {
2     stopc := make(chan struct{})
3     stop := context.AfterFunc(ctx, func() {
4         conn.SetReadDeadline(time.Now())
5         close(stopc)
6     })
7     n, err = conn.Read(b)
8
9     // Cancel.
10    if !stop() {
11        // The AfterFunc was started.
12        // Wait for it to complete, and reset the Conn's deadline.
13        <-stopc
14        conn.SetReadDeadline(time.Time{})
15        return n, ctx.Err()
16    }
17
18    // Read successfully.
19    return n, err
20 }
```

Пример для блокирующей функции Read



AfterFunc

Основные назначения:

Отмена блокирующих операций

Context with Deadline

```
1 type timerCtx struct {
2     cancelCtx
3     // Under cancelCtx.mu.
4     timer *time.Timer
5     deadline time.Time
6 }
7
8 func WithDeadline(parent Context, d time.Time) (Context, CancelFunc) {
9     if cur, ok := parent.Deadline(); ok && cur.Before(d) {
10         // The current deadline is already sooner than the new one.
11         return WithCancel(parent)
12     }
13     c := &timerCtx{
14         deadline: d,
15     }
16     c.cancelCtx.propagateCancel(parent, c)
17     dur := time.Until(d)
18     if dur <= 0 {
19         // Deadline has already passed.
20         c.cancel(true, DeadlineExceeded, nil)
21         return c, func() { c.cancel(false, Canceled, nil) }
22     }
23
24     if c.err == nil {
25         c.timer = time.AfterFunc(dur, func() {
26             c.cancel(true, DeadlineExceeded, nil)
27         })
28     }
29     return c, func() { c.cancel(true, Canceled, nil) }
30 }
```



Создание включает проверки на отмену и достижение границы времени

Context with Deadline

```
1 func (c *timerCtx) Deadline() (deadline time.Time, ok bool) {
2     return c.deadline, true
3 }
4
5 func (c *timerCtx) cancel(removeFromParent bool, err, cause error) {
6     c.cancelCtx.cancel(false, err, cause)
7     if removeFromParent {
8         // Remove this timerCtx from its parent cancelCtx's children.
9         removeChild(c.cancelCtx.Context, c)
10    }
11
12    if c.timer != nil {
13        c.timer.Stop()
14        c.timer = nil
15    }
16 }
```

➡ При отмене
останавливаем таймер

Context with Deadline

Основные назначения:

Отмена при вызове `cancel` или
по наступлении дедлайна

Context with Timeout

```
1 func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc) {  
2     return WithDeadline(parent, time.Now().Add(timeout))  
3 }
```

➡ Полностью основан на With Deadline

Context with Timeout

Основные назначения:

Отмена при вызове `cancel` или
по истечении времени

Cause

```
1 var Canceled error
2 var DeadlineExceeded error
3
4 func Cause(c Context) error {
5     cc, ok := c.Value(&cancelCtxKey).(*cancelCtx)
6     if ok {
7         return cc.cause
8     }
9     return c.Err()
10 }
11
12 func WithCancelCause(parent Context) (ctx Context, cancel CancelCauseFunc) {
13     c := withCancel(parent)
14     return c, func(cause error) { c.cancel(true, Canceled, cause) }
15 }
```



Информация от ошибок достаточно ограничена

Cause

```
1 var Canceled error
2 var DeadlineExceeded error
3
4 func Cause(c Context) error {
5     cc, ok := c.Value(&cancelCtxKey).(*cancelCtx)
6     if ok {
7         return cc.cause
8     }
9     return c.Err()
10 }
11
12 func WithCancelCause(parent Context) (ctx Context, cancel CancelCauseFunc) {
13     c := withCancel(parent)
14     return c, func(cause error) { c.cancel(true, Canceled, cause) }
15 }
```

- ➡ Информация от ошибок достаточно ограничена

- ➡ Соответственно, Cause используется для уточнения причины отмены

Cause

```
1 var Canceled error
2 var DeadlineExceeded error
3
4 func Cause(c Context) error {
5     cc, ok := c.Value(&cancelCtxKey).(*cancelCtx)
6     if ok {
7         return cc.cause
8     }
9     return c.Err()
10 }
11
12 func WithCancelCause(parent Context) (ctx Context, cancel CancelCauseFunc) {
13     c := withCancel(parent)
14     return c, func(cause error) { c.cancel(true, Canceled, cause) }
15 }
```

- ➡ Информация от ошибок достаточно ограничена

- ➡ Соответственно, Cause используется для уточнения причины отмены

- ➡ Представлен для всех контекстов с отменой: Cancel, Deadline, Timeout

Cause

Основные назначения:

Возможность задать кастомное описание причины отмены

Context with Value

```
1 type valueCtx struct {
2     Context
3     key any
4     val any
5 }
6
7 func WithValue(parent Context, key, val any) Context {
8     if !reflectlite.TypeOf(key).Comparable() {
9         panic("key is not comparable")
10    }
11    return &valueCtx{parent, key, val}
12 }
```

- Единственный контекст, в котором можно хранить произвольные данные

Context with Value

```
1 func (c *valueCtx) Value(key any) any {
2     if c.key == key {
3         return c.val
4     }
5     return value(c.Context, key)
6 }
7
8 func value(c Context, key any) any {
9     for {
10         switch ctx := c.(type) {
11             case *valueCtx:
12                 if key == ctx.key {
13                     return ctx.val
14                 }
15             case *cancelCtx:
16                 if key == &cancelCtxKey {
17                     return c
18                 }
19             case withoutCancelCtx:
20                 if key == &cancelCtxKey {
21                     // This implements Cause(ctx) == nil.
22                     // When ctx is created using WithoutCancel,
23                     return nil
24                 }
25             case *timerCtx:
26                 if key == &cancelCtxKey {
27                     return &ctx.cancelCtx
28                 }
29             case backgroundCtx, todoCtx:
30                 return nil
31         }
32         c = c.Context
33     }
34 }
```



Получение значения происходит с помощью рекурсивного обхода

Context with Value

Основные назначения:

Добавление метаданных. Например,
часто используется для HTTP-
заголовков

04



Best Practices



Best Practices

Лучше явно передавать в виде аргумента функции, причём первым и с названием `ctx`



```
1 type s struct {
2     ctx    context.Context
3     field any
4 }
5
6 func (s s) f(args ...any) {
7     // ...
8 }
```



```
1 type s struct {
2     field any
3 }
4
5 func (s s) f(ctx context.Context, args ...any) {
6     // ...
7 }
```



Best Practices

Всегда используйте `context.TODO()` вместо `nil`



```
1 func f(ctx context.Context) {  
2     // ...  
3 }  
4  
5 func main() {  
6     f(nil)  
7 }
```



```
1 func f(ctx context.Context) {  
2     // ...  
3 }  
4  
5 func main() {  
6     f(context.TODO())  
7 }
```



Best Practices

Не стоит использовать для передачи опциональных аргументов



```
1 func f(ctx context.Context) {
2     optStr, ok := ctx.Value("").(string)
3     if !ok {
4         // ...
5     }
6     // ...
7 }
```



```
1 func f(ctx context.Context, optStr *string) {
2     if optStr != nil {
3         // ...
4     }
5     // ...
6 }
```



Best Practices

В качестве ключей лучше использовать структуры, а не built-in типы
Структуры, по возможности, пустые



```
1 func f(ctx context.Context) {
2     metadata := ctx.Value("x-header")
3     // ...
4 }
```



```
1 type firstKey = struct{}
2 type secondKey = struct{}
3
4 func f(ctx context.Context) {
5     firstValue := ctx.Value(firstKey{})
6     secondValue := ctx.Value(secondKey{})
7     // ...
8 }
```



```
1 type firstKey struct{}
2 type secondKey struct{}
3
4 func f(ctx context.Context) {
5     firstValue := ctx.Value(firstKey{})
6     secondValue := ctx.Value(secondKey{})
7     // ...
8 }
```



Best Practices

Всегда добавляйте defer cancel(). **Двойная отмена не страшна, а утечка — да**



```
1 func f(ctx context.Context) {  
2     var cancel context.CancelFunc  
3     ctx, cancel = context.WithCancel(ctx)  
4     // ...  
5 }
```



```
1 func f(ctx context.Context) {  
2     var cancel context.CancelFunc  
3     ctx, cancel = context.WithCancel(ctx)  
4     defer cancel()  
5     // ...  
6 }
```





Заключение



Выводы

- ➡ Context, в сущности, — это удобная обёртка над каналом, предоставляющая дополнительный функционал



Выводы

- ➡ Context, в сущности, — это удобная обёртка над каналом, предоставляющая дополнительный функционал
- ➡ Сам пакет очень маленький, но написан слегка заумно



Выводы

- ➡ Context, в сущности, — это удобная обёртка над каналом, предоставляющая дополнительный функционал
- ➡ Сам пакет очень маленький, но написан слегка заумно
- ➡ Набор взаимосвязанных контекстов — это дерево
 - Вставка — $O(1)$
 - Поиск — $O(N)$



Выводы

- ➡ Context, в сущности, — это удобная обёртка над каналом, предоставляющая дополнительный функционал
- ➡ Сам пакет очень маленький, но написан слегка заумно
- ➡ Набор взаимосвязанных контекстов — это дерево
 - Вставка — $O(1)$
 - Поиск — $O(N)$
- ➡ Соблюдаем best practices: explicit лучше implicit



Источники



Документация пакета

<https://pkg.go.dev/context>



Статья на Хабре «Пакет context в Go: взгляд профессионала»

<https://habr.com/ru/companies/pt/articles/764850/>



Backend Podcast,
выпуск №9, «Пакет context в Go SDK»

<https://music.yandex.ru/album/28370806/track/119622159>



Спасибо
за внимание!



Олег Сидоренков

Разработчик информационных систем
OzonTech

t.me/olegdayo

