context

Package context

go1.15.2 Latest

Published: **Sep 9, 2020** | License: BSD-3-Clause | Standard library

Overview

Package context defines the Context type, which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.

Incoming requests to a server should create a Context, and outgoing calls to servers should accept a Context. The chain of function calls between them must propagate the Context, optionally replacing it with a derived Context created using WithCancel, WithDeadline, WithTimeout, or WithValue. When a Context is canceled, all Contexts derived from it are also canceled.

The WithCancel, WithDeadline, and WithTimeout functions take a Context (the parent) and return a derived Context (the child) and a CancelFunc. Calling the CancelFunc cancels the child and its children, removes the parent's reference to the child, and stops any associated timers. Failing to call the CancelFunc leaks the child and its children until the parent is canceled or the timer fires. The go vet tool checks that CancelFuncs are used on all control-flow paths.

Programs that use Contexts should follow these rules to keep interfaces consistent across packages and enable static analysis tools to check context propagation:

Do not store Contexts inside a struct type; instead, pass a Context explicitly to each function that needs it. The Context should be the first parameter, typically named ctx:

```
func DoSomething(ctx context.Context, arg Arg) error {
	// ... use ctx ...
}
```

Do not pass a nil Context, even if a function permits it. Pass context.TODO if you are unsure about which Context to use.

Use context Values only for request-scoped data that transits processes and APIs, not for passing optional parameters to functions.

The same Context may be passed to functions running in different goroutines; Contexts are safe for simultaneous use by multiple goroutines.

See https://blog.golang.org/context for example code for a server that uses Contexts.

Variables

```
var Canceled = errors.New("context canceled")
```

Canceled is the error returned by Context.Err when the context is canceled.

```
var DeadlineExceeded error = deadlineExceededError{}
```

DeadlineExceeded is the error returned by Context.Err when the context's deadline passes.

func WithCancel

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
```

WithCancel returns a copy of parent with a new Done channel. The returned context's Done channel is closed when the returned cancel function is called or when the parent context's Done channel is closed, whichever happens first.

Canceling this context releases resources associated with it, so code should call cancel as soon as the operations running in this Context complete.

func WithDeadline

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

WithDeadline returns a copy of the parent context with the deadline adjusted to be no later than d. If the parent's deadline is already earlier than d, WithDeadline(parent, d) is semantically equivalent to parent. The returned context's Done channel is closed when the deadline expires, when the returned cancel function is called, or when the parent context's Done channel is closed, whichever happens first.

Canceling this context releases resources associated with it, so code should call cancel as soon as the operations running in this Context complete.

func WithTimeout

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

WithTimeout returns WithDeadline(parent, time.Now().Add(timeout)).

Canceling this context releases resources associated with it, so code should call cancel as soon as the operations running in this Context complete:

```
func slowOperationWithTimeout(ctx context.Context) (Result, error) {
    ctx, cancel := context.WithTimeout(ctx, 100*time.Millisecond)
    defer cancel()  // releases resources if slowOperation completes before timeout e
    return slowOperation(ctx)
}
```

type CancelFunc

```
type CancelFunc func()
```

A CancelFunc tells an operation to abandon its work. A CancelFunc does not wait for the work to stop. A CancelFunc may be called by multiple goroutines simultaneously. After the first call, subsequent calls to a CancelFunc do nothing.

type Context

```
type Context interface {
    // Deadline returns the time when work done on behalf of this context
    // should be canceled. Deadline returns ok==false when no deadline is
    // set. Successive calls to Deadline return the same results.
    Deadline() (deadline time.Time, ok bool)

    // Done returns a channel that's closed when work done on behalf of this
    // context should be canceled. Done may return nil if this context can
    // never be canceled. Successive calls to Done return the same value.
    // The close of the Done channel may happen asynchronously,
    // after the cancel function returns.
    //
    // WithCancel arranges for Done to be closed when cancel is called;
    // WithDeadline arranges for Done to be closed when the deadline
    // expires; WithTimeout arranges for Done to be closed when the timeout
    // elapses.
    //
    // Done is provided for use in select statements:
    //
    //  // Stream generates values with DoSomething and sends them to out
    //  // until DoSomething returns an error or ctx.Done is closed.
    //  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:
    //          }
    //      }
    //  }
    //
    // See https://blog.golang.org/pipelines for more examples of how to use
    // a Done channel for cancellation.
    Done() <-chan struct{}

    // If Done is not yet closed, Err returns nil.
    // If Done is closed, Err returns a non-nil error explaining why:
    // Canceled if the context was canceled
    // or DeadlineExceeded if the context's deadline passed.
    // After Err returns a non-nil error, successive calls to Err return the same err
    Err() error

    // Value returns the value associated with this context for key, or nil
    // if no value is associated with key. Successive calls to Value with
    // the same key returns the same result.
    //
    // Use context values only for request-scoped data that transits
    // processes and API boundaries, not for passing optional parameters to
    // functions.
    //
    // A key identifies a specific value in a Context. Functions that wish
    // to store values in Context typically allocate a key in a global
    // variable then use that key as the argument to context.WithValue and
    // Context.Value. A key can be any type that supports equality;
    // packages should define keys as an unexported type to avoid
    // collisions.
    //
    // Packages that define a Context key should provide type-safe accessors
    // for the values stored using that key:
    //
    //  // Package user defines a User type that's stored in Contexts.
    //  package user
    //
    //  import "context"
    //
    //  // User is the type of value stored in the Contexts.
    //  type User struct {...}
    //
    //  // key is an unexported type for keys defined in this package.
    //  // This prevents collisions with keys defined in other packages.
    //  type key int
    //
    //  // userKey is the key for user.User values in Contexts. It is
    //  // unexported; clients use user.NewContext and user.FromContext
    //  // instead of using this key directly.
    //  var userKey key
    //
    //  // NewContext returns a new Context that carries value u.
    //  func NewContext(ctx context.Context, u *User) context.Context {
    //      return context.WithValue(ctx, userKey, u)
    //  }
    //
    //  // FromContext returns the User value stored in ctx, if any.
    //  func FromContext(ctx context.Context) (*User, bool) {
    //      u, ok := ctx.Value(userKey).(*User)
    //      return u, ok
    //  }
    Value(key interface{}) interface{}
}
```

A Context carries a deadline, a cancellation signal, and other values across API boundaries.

Context's methods may be called by multiple goroutines simultaneously.

func Background

```
func Background() Context
```

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 TODO

```
func TODO() Context
```

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 WithValue

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

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.