The Go Blog

Go Concurrency Patterns: Context

29 July 2014

Introduction

In Go servers, each incoming request is handled in its own goroutine. Request handlers often start additional goroutines to access backends such as databases and RPC services. The set of goroutines working on a request typically needs access to request-specific values such as the identity of the end user, authorization tokens, and the request's deadline. When a request is canceled or times out, all the goroutines working on that request should exit quickly so the system can reclaim any resources they are using.

At Google, we developed a `context` package that makes it easy to pass request-scoped values, cancelation signals, and deadlines across API boundaries to all the goroutines involved in handling a request. The package is publicly available as context. This article describes how to use the package and provides a complete working example.

Next article

Go at OSCON

Previous article

Go will be at OSCON 2014

Links

golang.org
Install Go
A Tour of Go
Go Documentation
Go Mailing List
Go on Google+
Go+ Community
Go on Twitter

Blog index

Context

The core of the `context` package is the `Context` type:

```
// A Context carries a deadline, cancelation signal, and request-scoped
values
// across API boundaries. Its methods are safe for simultaneous use by
multiple
// goroutines.
type Context interface {
    // Done returns a channel that is closed when this Context is
```

```
canceled
    // or times out.
    Done() <-chan struct{}

    // Err indicates why this context was canceled, after the Done
channel
    // is closed.
    Err() error

    // Deadline returns the time when this Context will be canceled, if
any.
    Deadline() (deadline time.Time, ok bool)

    // Value returns the value associated with key or nil if none.
    Value(key interface{}) interface{}
}
```

(This description is condensed; the godoc is authoritative.)

The `Done` method returns a channel that acts as a cancelation signal to functions running on behalf of the `Context`: when the channel is closed, the functions should abandon their work and return. The `Err` method returns an error indicating why the `Context` was canceled. The Pipelines and Cancelation article discusses the `Done` channel idiom in more detail.

A `Context` does *not* have a `Cancel` method for the same reason the `Done` channel is receive-only: the function receiving a cancelation signal is usually not the one that sends the signal. In particular, when a parent operation starts goroutines for sub-operations, those sub-operations should not be able to cancel the parent. Instead, the `WithCancel` function (described below) provides a way to cancel a new `Context` value.

A `Context` is safe for simultaneous use by multiple goroutines. Code can pass a single `Context` to any number of goroutines and cancel that `Context` to signal all of them.

The `Deadline` method allows functions to determine whether they should start work at all; if too little time is left, it may not be worthwhile. Code may also use a deadline to set timeouts for I/O operations.

`Value` allows a `Context` to carry request-scoped data. That data must be safe for simultaneous use by multiple goroutines.

Derived contexts

The `context` package provides functions to *derive* new `Context` values from existing ones. These values form a tree: when a `Context` is canceled, all `Contexts` derived from it are also canceled.

`Background` is the root of any `Context` tree; it is never canceled:

```
// Background returns an empty Context. It is never canceled, has no
deadline,
// and has no values. Background is typically used in main, init, and
tests,
// and as the top-level Context for incoming requests.
func Background() Context
```

`WithCancel` and `WithTimeout` return derived `Context` values that can be canceled sooner than the parent `Context`. The `Context` associated with an incoming request is typically canceled when the request handler returns. `WithCancel` is also useful for canceling redundant requests when using multiple replicas. `WithTimeout` is useful for setting a deadline on requests to backend servers:

```
// WithCancel returns a copy of parent whose Done channel is closed as
soon as
// parent.Done is closed or cancel is called.
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)

// A CancelFunc cancels a Context.
type CancelFunc func()

// WithTimeout returns a copy of parent whose Done channel is closed as
soon as
// parent.Done is closed, cancel is called, or timeout elapses. The new
// Context's Deadline is the sooner of now+timeout and the parent's
deadline, if
// any. If the timer is still running, the cancel function releases its
// resources.
func WithTimeout(parent Context, timeout time.Duration) (Context,
CancelFunc)
```

`WithValue` provides a way to associate request-scoped values with a `Context`:

```
// WithValue returns a copy of parent whose Value method returns val for
key.
```

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

The best way to see how to use the `context` package is through a worked example.

Example: Google Web Search

Our example is an HTTP server that handles URLs like `/search?q=golang&timeout=1s` by forwarding the query "golang" to the Google Web Search API and rendering the results. The `timeout` parameter tells the server to cancel the request after that duration elapses.

The code is split across three packages:

- server provides the `main` function and the handler for `/search`.
- userip provides functions for extracting a user IP address from a request and associating it with a `Context`.
- google provides the `Search` function for sending a query to Google.

The server program

The server program handles requests like `/search?q=golang` by serving the first few Google search results for `golang`. It registers `handleSearch` to handle the `/search` endpoint. The handler creates an initial `Context` called `ctx` and arranges for it to be canceled when the handler returns. If the request includes the `timeout` URL parameter, the `Context` is canceled automatically when the timeout elapses:

```
func handleSearch(w http.ResponseWriter, req *http.Request) {
    // ctx is the Context for this handler. Calling cancel closes the
    // ctx.Done channel, which is the cancellation signal for requests
    // started by this handler.
    var (
        ctx    context.Context
        cancel context.CancelFunc
    )
    timeout, err := time.ParseDuration(req.FormValue("timeout"))
    if err == nil {
        // The request has a timeout, so create a context that is
        // canceled automatically when the timeout expires.
        ctx, cancel = context.WithTimeout(context.Background(), timeout)
    } else {
        ctx, cancel = context.WithCancel(context.Background())
    }
    defer cancel() // Cancel ctx as soon as handleSearch returns.
```

The handler extracts the query from the request and extracts the client's IP address by calling on the `userip` package. The client's IP address is needed for backend requests, so `handleSearch` attaches it to `ctx`:

```
// Check the search query.
query := req.FormValue("q")
if query == "" {
    http.Error(w, "no query", http.StatusBadRequest)
    return
}

// Store the user IP in ctx for use by code in other packages.
userIP, err := userip.FromRequest(req)
if err != nil {
    http.Error(w, err.Error(), http.StatusBadRequest)
    return
}
ctx = userip.NewContext(ctx, userIP)
```

The handler calls `google.Search` with `ctx` and the `query`:

```
// Run the Google search and print the results.
start := time.Now()
results, err := google.Search(ctx, query)
elapsed := time.Since(start)
```

If the search succeeds, the handler renders the results:

```
if err := resultsTemplate.Execute(w, struct {
    Results          google.Results
    Timeout, Elapsed time.Duration
}{
    Results: results,
    Timeout: timeout,
    Elapsed: elapsed,
}); err != nil {
    log.Print(err)
    return
}
```

Package userip

The userip package provides functions for extracting a user IP address from a request and associating it with a `Context`. A `Context` provides a key-value mapping, where the keys and values are both of type `interface{}`. Key types must support equality, and values must be safe for simultaneous use by multiple goroutines. Packages like `userip` hide the details of this mapping and provide strongly-typed access to a specific `Context` value.

To avoid key collisions, `userip` defines an unexported type `key` and uses a value of this type as the context key:

```
// The key type is unexported to prevent collisions with context keys
defined in
// other packages.
type key int

// userIPkey is the context key for the user IP address.  Its value of
zero is
// arbitrary.  If this package defined other context keys, they would
have
// different integer values.
const userIPKey key = 0
```

`FromRequest` extracts a `userIP` value from an `http.Request`:

```
func FromRequest(req *http.Request) (net.IP, error) {
    ip, _, err := net.SplitHostPort(req.RemoteAddr)
    if err != nil {
        return nil, fmt.Errorf("userip: %q is not IP:port",
req.RemoteAddr)
    }
```

`NewContext` returns a new `Context` that carries a provided `userIP` value:

```
func NewContext(ctx context.Context, userIP net.IP) context.Context {
    return context.WithValue(ctx, userIPKey, userIP)
}
```

`FromContext` extracts a `userIP` from a `Context`:

```
func FromContext(ctx context.Context) (net.IP, bool) {
    // ctx.Value returns nil if ctx has no value for the key;
```

```
    // the net.IP type assertion returns ok=false for nil.
    userIP, ok := ctx.Value(userIPKey).(net.IP)
    return userIP, ok
}
```

Package google

The google.Search function makes an HTTP request to the Google Web Search API and parses the JSON-encoded result. It accepts a `Context` parameter `ctx` and returns immediately if `ctx.Done` is closed while the request is in flight.

The Google Web Search API request includes the search query and the user IP as query parameters:

```
func Search(ctx context.Context, query string) (Results, error) {
    // Prepare the Google Search API request.
    req, err := http.NewRequest("GET",
"https://ajax.googleapis.com/ajax/services/search/web?v=1.0", nil)
    if err != nil {
        return nil, err
    }
    q := req.URL.Query()
    q.Set("q", query)

    // If ctx is carrying the user IP address, forward it to the server.
    // Google APIs use the user IP to distinguish server-initiated
requests
    // from end-user requests.
    if userIP, ok := userip.FromContext(ctx); ok {
        q.Set("userip", userIP.String())
    }
    req.URL.RawQuery = q.Encode()
```

`Search` uses a helper function, `httpDo`, to issue the HTTP request and cancel it if `ctx.Done` is closed while the request or response is being processed. `Search` passes a closure to `httpDo` handle the HTTP response:

```
    var results Results
    err = httpDo(ctx, req, func(resp *http.Response, err error) error {
        if err != nil {
            return err
        }
```

```
        defer resp.Body.Close()

        // Parse the JSON search result.
        // https://developers.google.com/web-search/docs/#fonje
        var data struct {
            ResponseData struct {
                Results []struct {
                    TitleNoFormatting string
                    URL               string
                }
            }
        }
        if err := json.NewDecoder(resp.Body).Decode(&data); err != nil {
            return err
        }
        for _, res := range data.ResponseData.Results {
            results = append(results, Result{Title:
res.TitleNoFormatting, URL: res.URL})
        }
        return nil
    })
    // httpDo waits for the closure we provided to return, so it's safe
to
    // read results here.
    return results, err
```

The `httpDo` function runs the HTTP request and processes its response in a new goroutine. It cancels the request if `ctx.Done` is closed before the goroutine exits:

```
func httpDo(ctx context.Context, req *http.Request, f
func(*http.Response, error) error) error {
    // Run the HTTP request in a goroutine and pass the response to f.
    tr := &http.Transport{}
    client := &http.Client{Transport: tr}
    c := make(chan error, 1)
    go func() { c <- f(client.Do(req)) }()
    select {
    case <-ctx.Done():
        tr.CancelRequest(req)
        <-c // Wait for f to return.
        return ctx.Err()
    case err := <-c:
        return err
```

```
        }
    }
```

Adapting code for Contexts

Many server frameworks provide packages and types for carrying request-scoped values. We can define new implementations of the `Context` interface to bridge between code using existing frameworks and code that expects a `Context` parameter.

For example, Gorilla's github.com/gorilla/context package allows handlers to associate data with incoming requests by providing a mapping from HTTP requests to key-value pairs. In gorilla.go, we provide a `Context` implementation whose `Value` method returns the values associated with a specific HTTP request in the Gorilla package.

Other packages have provided cancelation support similar to `Context`. For example, Tomb provides a `Kill` method that signals cancelation by closing a `Dying` channel. `Tomb` also provides methods to wait for those goroutines to exit, similar to `sync.WaitGroup`. In tomb.go, we provide a `Context` implementation that is canceled when either its parent `Context` is canceled or a provided `Tomb` is killed.

Conclusion

At Google, we require that Go programmers pass a `Context` parameter as the first argument to every function on the call path between incoming and outgoing requests. This allows Go code developed by many different teams to interoperate well. It provides simple control over timeouts and cancelation and ensures that critical values like security credentials transit Go programs properly.

Server frameworks that want to build on `Context` should provide implementations of `Context` to bridge between their packages and those that expect a `Context` parameter. Their client libraries would then accept a `Context` from the calling code. By establishing a common interface for request-scoped data and cancelation, `Context` makes it easier for package developers to share code for creating scalable services.

By Sameer Ajmani

Related articles

- Go Concurrency Patterns: Pipelines and cancellation

- Introducing the Go Race Detector
- Advanced Go Concurrency Patterns
- Concurrency is not parallelism
- Go videos from Google I/O 2012
- Go Concurrency Patterns: Timing out, moving on
- Share Memory By Communicating