Context

Context

Writing highly concurrent code can get complex, and controlling that concurrent systems can be even harder.

In Go 1.7 the `context` package was introduced, designed to solved exactly these problems.

The Context Interface

The `Context` interface consists of just 4 methods that need to be implemented to fulfill the interface.

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

The `context` package has default implementations for almost all use cases.

Cancellation

One of the most useful, and common, uses for `context.Context` is cancellation of multiple Goroutines.

Using a `context.Context` and the `select` statement we can control a Goroutine.

Problem

We want to launch `N` number of Goroutines and search for a particular number, `42`. How do we tell all the Goroutines to stop searching once one of them has found the magic number?

Managing With A WaitGroup

One option would be use a `sync.WaitGroup` to tell the main process to wait while all of the Goroutines search for the number.

```
var found string
var moot = &sync.Mutex{}

func main() {
    wg := &sync.WaitGroup{}
    for i := 0; i < N; i++ {
        wg.Add(1)
        go search(i, wg)
    }
    wg.Wait()

    fmt.Print(found)
}
                                                                    Ø
```

Managing With A WaitGroup

This solution requires the extensive use of `sync.Mutex` as well to prevent multiple Goroutines from deadlocking on writing/reading the `found` variable.

```
func search(i int, wg *sync.WaitGroup) {
    for {
        fmt.Print(".")
        moot.Lock()
        if found != "" {
            moot.Unlock()
            wg.Done()
            break
        }
        moot.Unlock()
        x := rand.Intn(N)
        if x == 42 {
            moot.Lock()
            fmt.Print("found")
            found = fmt.Sprintf("routine %d: found 42!", i)
            moot.Unlock()
            wg.Done()
            break
        }
        time.Sleep(10 * time.Millisecond)
    }
}
```

Managing With Channels

A better solution, would be to use Channels to manage the Goroutines

```
const N = 50

var found = make(chan string)

func main() {
    for i := 0; i < N; i++ {
        go search(i)
    }
    fmt.Print(<-found)
    time.Sleep(3 * time.Second)
}
```

Ø

Managing With Channels

The drawback with this solution is that, while the code is cleaner, we are never properly shutting down the Goroutines, they are being killed when the program exits.

```
func search(i int) {
    for {
        fmt.Print(".")
        x := rand.Intn(N)
        if x == 42 {
            fmt.Print("found")
            found <- fmt.Sprintf("routine %d: found 42!", i)
            break
        }
        time.Sleep(10 * time.Millisecond)
    }
}
```

Ø

Managing With Channels (output)

This code will keep running until our program finally exits, despite the fact that it found the number it was looking for.

```
.....................found.routine 13: found 42!.........................
.....found.............................................found.....................
....found.............found.....................................................
found.....................................................found.....................
............found..found.................................found.....................
.................................found.......................found...........
.....................................................................................
.....................................................found.................
.....................................found.................................
.....................................................................................
.....................................................found.................
.............................found.....................found.................
.......found.....found.......foundfound.....................................found
nd.....................................................found........
.....................................................found.....................
.....................................................................................
.....................................................................found....
...........found..found.....................................found.................
..found.........found.....................................................found.........
..found.....................................found.................................
.....................................................found.....................
.....................................................................................
```

Creating A New Context

All contexts start with a `context.Background()` context. This is just an "empty" context onto which we can create new "child" contexts.

Contexts behave like trees with a top node, `context.Background()`, and then further sub-nodes.

Cancelling With Contexts

To help solve the problem of cancelling Goroutines we can create a new context that also gives us a cancellation function we can call when we are ready to stop our Goroutines.

```
ctx, cancel := context.WithCancel(context.Background())
```

When the `cancel` function is a called a message is sent to the `ctx.Done()` channel that we can listen to and stop our Goroutines safely.

Cancelling With Contexts

Updating the `search` function to take `context.Context` we can use a `select` statement to listen to the `ctx.Done()` channel on the context and break out of the loop that is searching for the number.

```
func search(ctx context.Context, i int) {
Loop:
    for {
        select {
        case <-ctx.Done():
            fmt.Print("x")
            break Loop
        default:
            fmt.Print(".")
            x := rand.Intn(N)
            if x == 42 {
                fmt.Print("found")
                found <- fmt.Sprintf("routine %d: found 42!", i)
                break Loop
            }
            time.Sleep(10 * time.Millisecond)
        }
    }
}
```

Ø

Cancelling With Contexts

Putting it altogether once we receive a message on the `found` channel we can call the `cancel` function and terminate all of the outstanding Goroutines.

```
func main() {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()
    for i := 0; i < N; i++ {
        go search(ctx, i)
    }
    m := <-found
    cancel()
    fmt.Print(m)
    time.Sleep(3 * time.Second)
}
```

Ø

Cancelling With Contexts (output)

Notice in the output that all of the Goroutines are safely shutdown before the program exits.

```
.......found.............................routine 10: found 42!xxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Cancelling With Timeouts

The current program has a subtle bug in it. What if the number we are searching for is never found? The program will run forever. That is probably not the behavior we want to have.

The `context` package gives us two different types of contexts we can use to fix this bug.

```
context.WithTimeout
context.WithDeadline
```

Both do *almost* the same thing. `WithTimeout` will cancel the context *after* a certain amount of time. `WithDeadline` will cancel the context *at* a certain time.

Cancelling With Timeouts

We can update the `main` function of the application to use a `select` statement to listen for the `found` channel, as well as the `ctx.Done()` channel.

```
func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
    defer cancel()
    for i := 0; i < N; i++ {
        go search(ctx, i)
    }
    select {
    case m := <-found:
        cancel()
        fmt.Print(m)
    case <-ctx.Done():
        cancel()
        if err := ctx.Err(); err != nil {
            fmt.Println(err)
        }
    }
    time.Sleep(3 * time.Second)
}
```

Ø

The `search` function in the application did not have to change to handle this new behavior.

Cancelling With Timeouts (output)

In this output the number wasn't found in the allotted time, so the Goroutines were all cancelled and the application was shutdown.

```
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................xxxxxxxxxxxxcontext deadline exceeded xxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxx
```

Context Values

Contexts also allow us to pass values through our applications easily, using `context.WithValue`. This can clean up an application and can help make sure that necessary information is passed along between concurrent processes.

What Values Should Be Passed?

Do pass anything that is "request" specific, such as the "current user" or a request ID.

Don't pass "global" values such as database connections or loggers.

Context Values

With `context.WithValue` we can set a key/value pair into a new context.

```
ctx = context.WithValue(ctx, "magic", 42)
```

```
// returns an interface or nil
i := ctx.Value("magic")
```

If the current context does not have the requested value it will keep looking up parent tree until it either finds the value, or hits the top most node.

Context Values

Using context values we can create a new context that has the "magic" value of `42`, and pass that to to our Goroutines that are searching for that number.

```
func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
    defer cancel()
    ctx = context.WithValue(ctx, "magic", 42)
    for i := 0; i < N; i++ {
        go search(ctx, i)
    }
    select {
    case m := <-found:
        cancel()
        fmt.Print(m)
    case <-ctx.Done():
        cancel()
        if err := ctx.Err(); err != nil {
            fmt.Println(err)
        }
    }
    time.Sleep(3 * time.Second)
}
```

Ø

Context Values

Updating the `search` function involves pulling the `magic` value out of the context and making sure to cast it to `int` from an interface.

```
func search(ctx context.Context, i int) {
    magic := ctx.Value("magic").(int)
Loop:
    for {
        select {
        case <-ctx.Done():
            fmt.Print("x")
            break Loop
        default:
            fmt.Print(".")
            x := rand.Intn(N)
            if x == magic {
                fmt.Print("found")
                found <- fmt.Sprintf("routine %d: found %d!", i, magic)
                break Loop
            }
            time.Sleep(10 * time.Millisecond)
        }
    }
}
```

Ø

Note: It is possible that `magic` is `nil` and should be checked for. We aren't checking it for simplicity in this example.

Exercises

At http://hamlet.gopherguides.com you will find a text copy of the play Hamlet.

Write a program using contexts that searches each line of text from the play to search for a particular word. Examples are `Hamlet`, `Mark`, `King` etc...

The program must:

- print out the line number the word was found, and the text of that line: `3545: QUEEN GERTRUDE O Hamlet, speak no more`
- stop when it finds 50 occurrences of the word
- stop after `5` seconds if it hasn't yet found 50 occurrences

Solution

```
type Result struct {
    LineNumber int
    Text       string
}

func (r Result) String() string {
    return fmt.Sprintf("%d: %s", r.LineNumber, r.Text)
}

func search(ctx context.Context, book []byte, results chan Result) {
Loop:
    for i, line := range bytes.Split(book, []byte("\n")) {
        select {
        case <-ctx.Done():
            break Loop
        default:
            if bytes.Contains(line, []byte("Hamlet")) {
                results <- Result{
                    LineNumber: i + 1,
                    Text:       string(line),
                }
            }
        }
    }
}
```

Ø

Solution

```go
func retrieveBook() ([]byte, error) {
    res, err := http.Get(hamlet)
    if err != nil {
        return nil, errors.WithStack(err)
    }
    if res.StatusCode != 200 {
        return nil, errors.Errorf("received a non-success message %d", res.StatusCode)
    }

    b, err := ioutil.ReadAll(res.Body)
    if err != nil {
        return b, errors.WithStack(err)
    }

    return b, nil
}
```

Ø

Solution

```
func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    book, err := retrieveBook()
    if err != nil {
        log.Fatal(err)
    }

    results := make(chan Result)

    go search(ctx, book, results)

Loop:
    for i := 0; i < 50; i++ {
        select {
        case r := <-results:
            fmt.Printf("[%d] %s\n", i, r)
        case <-ctx.Done():
            cancel()
            if err := ctx.Err(); err != nil {
                log.Fatal(err)
            }
            break Loop
        }
    }
}
```

Ø