Futures and Promises

Martin Sulzmann, Bas van den Heuvel

2024-06-05

Overview

- Futures and promises are a high-level concurrency construct to support asynchronous programming.
- A *future* can be viewed as a placeholder for a computation that will eventually become available.
- A *promise* is a placeholder for a computation that is explicitly provided by the programmer.
- A future is then a specific kind of promise where the computation is provided at initialization.
- For a high-level overview, see here.

Lecture plan

1. We first implement futures via channels. We use `interface{}` to represent a value of arbitrary type.
2. We increase expressivity by considering how we can simultaneously wait for multiple futures.
3. Using `interface{}` entails explicit type assertions. We avoid this using *generics*.
4. We implement promises, and redefine futures in terms of promises. In the process, we reduce the number of goroutines to optimize performance.

Channel-based futures

```go
// Result of a computation.
type Comp struct {
    val    interface{} // value of any type
    status bool        // success or not
}
```

```go
// A future is a channel that provides the result of a computation.
type Future chan Comp

// At initialization, the future channel is created and the computation runs as a goroutine
// Once the computation is finished, the result is continuously sent on the future channel.
func future(f func() (interface{}, bool)) Future {
    ch := make(chan Comp)
    go func() {
        r, s := f()
        v := Comp{r, s}
        for {
            ch <- v
        }
    }()
    return ch
}

// The future getter blocks until the result of the computation is available.
func (ft Future) get() (interface{}, bool) {
    c := <- ft
    return c.val, c.status
}

// When the computation is successfull, the callback function is called with the computatio
// Used to register a callback, so should not be blocking!
// Possible to register multiple callbacks, because the future keeps sending the result.
func (ft Future) onSuccess(cb func(interface{})) {
    go func() {
        v, o := ft.get()
        if o {
            cb(v)
        }
    }()
}

// When the computation fails, the callback function is called.
func (ft Future) onFailure(cb func()) {
    go func() {
        _, o := ft.get()
        if !o {
            cb()
        }
    }()
}
```

Example

Here is an example application where we asynchronously execute some http request. We don't have to wait for the request to complete.

```
func getSite(url string) Future {
    return future(func() (interface{}, bool) {
        resp, err := http.Get(url)
        if err == nil {
            return resp, true
        }
        return nil, false
    })
}

func printResponse(response *http.Response) {
    fmt.Println(response.Request.URL)
    date := response.Header.Get("Date")
    fmt.Println(date)
}

func example1() {
    stern := getSite("http://www.stern.de")

    stern.onSuccess(func(result interface{}) {
        response := result.(*http.Response) // Type cast!
        printResponse(response)
    })

    stern.onFailure(func() {
        fmt.Println("failure")
    })

    fmt.Printn("do something else")
    time.Sleep(2 * time.Second)
}
```

More expressive functionality for futures

Suppose we make several http requests (say stern and spiegel) and would like to retrieve the first available result.

A naive (inefficient) solution would check for each result one after the other (via `get`). Can we be more efficient? Yes: we use `select` to check for the first available future result.

```
// Method for Futures.
```

```go
// Which of ft1 and ft2 is available first?
func (ft1 Future) first(ft2 Future) Future {
    return future(func() (interface{}, bool) {
        var v interface{}
        var o bool

        // Check for either result to become available.
        select {
        case x1 := <-ft1:
            v = x1.val
            o = x1.status

        case x2 := <-ft2:
            v = x2.val
            o = x2.status
        }

        return v, o
    })
}

// Which of ft1 and ft2 returns a successful result first?
// Returns failure if neither returns success.
func (ft1 Future) firstSucc(ft2 Future) Future {
    return future(func() (interface{}, bool) {
        var v interface{}
        var o bool

        select {
        case x1 := <-ft1:
            if x1.status {
                v = x1.val
                o = x1.status
            }
            else {
                v, o = ft2.get()
            }

        case x2 := <-ft2:
            if x2.status {
                v = x2.val
                o = x2.status
            }
            else {
                v, o = ft.get()
            }
        }

        return v, o
    })
}
```

Example

We extend the previous example to three simultaneous http requests. We only care about the first available result.

```
func example2() {
    spiegel := getSite("http://www.spiegel.de")
    stern := getSite("http://www.stern.de")
    welt := getSite("http://www.welt.com")

    // stern.first(welt) creates a future that returns the first available result between st
    // Hence, this line creates a future that returns the first available result between ste
    req := spiegel.first(stern.first(welt))

    req.onSuccess(func(result interface{}) {
        response := result.(*http.Response)
        printResponse(response)
    })

    req.onFailure(func() {
        fmt.Println("failure")
    })

    fmt.Println("do something else")
    time.Sleep(2 * time.Second)
}
```

Another example (holiday booking)

```
func example3() {
    // Book some hotel.
    // Report price (int) and potential failure (bool).
    booking := func() (int, bool) {
        time.Sleep((time.Duration)(rand.Intn(999)) * time.Millisecond)
        return rand.Intn(50), true
    }

    // Can't simply do:
    // ft1 := future(booking)
    // because we use interface{}...
    ft1 := future(func() (interface{}, bool) {
        return booking()
    })

    ft2 := future(func() (interface{}, bool) {
        return booking()
    })

    ft := ft1.firstSucc(ft2)

    ft.onSuccess(func(result interface{}) {
        quote := result.(int)
        fmt.Printf("Hotel asks for %d Euros\n", quote)
    })

    time.Sleep(2 * time.Second)
}
```

Generics!

```
type Comp[T any] struct {
    val    T
    status bool
}

type Future[T any] chan Comp[T]
```

Holiday booking with generics

```
func example3b() {
    booking := func() (int, bool) {
        time.Sleep((time.Duration)(rand.Intn(999)) * time.Millisecond)
        return rand.Intn(50), true
    }

    // Provide return value at initialization.
    ft1 := future[int](booking)

    ft2 := future[int](booking)

    ft := ft1.firstSucc(ft2)

    ft.onSuccess(func(quote int) {
        fmt.Printf("Hotel asks for %d euros\n", quote)
    })
    time.Sleep(2 * time.Second)
}
```

Discussion

- Our current implementation uses a lot of goroutines:
  - Each future creates a goroutine.
  - Each `onSuccess`/`onFailure` call creates a goroutine.
  - `first`/`firstSucc` create further futures and therefore further goroutines.
- Goroutines in Go are relatively cheap. Still, we should avoid them if possible.
- Idea:
  - Each future maintains a list of callback functions.
    * One list for the success case.
    * Another list for the failure case.
  - Each `onSuccess`/`onFailure` call adds the callback to the respective list.
  - What if the "future" value is already present?
    * There's no need to register the callback.
    * We can immediately apply the "future" value and process the callback.

Promises

Instead of providing a computation at initialization, promises are futures that are set explicitly (at any time) by the user.

```
package main

import "fmt"
import "time"
import "math/rand"

type Promise[T any] struct {
    val           T          // return value
    status        bool       // success or not
    m             chan int   // mutex
```

```go
    succCallBacks []func(T)
    failCallBacks []func()
    empty         bool      // no value set yet?
}

func newPromise[T any]() *Promise[T] {
    // buffered channel as a mutex.
    p := Promise[T]{empty: true, m: make(chan int, 1), succCallBacks: make([]func(T), 0), fa
    // return a reference to prevent copying.
    return &p
}

// set successfull computation.
func (p *Promise[T]) setSucc(v T) {
    p.m <- 1 // lock.
    if p.empty { // not set yet.
        p.val = v
        p.status = true
        p.empty = false // not empty anymore.
        succs := p.succCallBacks
        p.succCallBacks = make([]func(T), 0) // reset callbacks.
        <- p.m // release.
        go func() {
            for _, cb := range succs { // call every current callback.
                cb(v)
            }
        }()
    }
    else { // already set, not allowed to overwrite.
        <- p.m // release.
    }
}

// set failed computation.
func (p *Promise[T]) setFail() {
    p.m <- 1
    if p.empty {
        p.status = false
        p.empty = false
        fails := p.failCallBacks
        p.failCallBacks = make([]func(), 0)
        <- p.m
        go func() {
            for _, cb := range fails {
                cb()
            }
        }()
    }
    else {
        <-p.m
    }
}

// future as special form of promise.
func future[T any](f func() (T, bool)) *Promise[T] {
    p := newPromise[T]()
    go func() {
        r, s := f()
        if s {
            p.setSucc(r)
        } else {
            p.setFail()
        }
    }()
    return p
}

// set promise computation, success or not.
func (p *Promise[T]) complete(f func() (T, bool)) {
    go func() {
        r, s := f()
        if s {
            p.setSucc(r)
        }
        else {
            p.setFail()
        }
    }()
}

// register callback for success.
func (p *Promise[T]) onSuccess(cb func(T)) {
    p.m <- 1 // lock.
    if p.empty { // not set yet, register callback.
        p.succCallBacks = append(p.succCallBacks, cb)
    }
    else if !p.empty && p.status { // already set and successfull, immediate call callback.
        go cb(p.val)
    }
    else { } // already set but failure, drop callback.
    <-p.m // release.
}
```

```go
func (p *Promise[T]) onFailure(cb func()) {
    p.m <- 1
    if p.empty {
        p.failCallBacks = append(p.failCallBacks, cb)
    }
    else if !p.empty && !p.status {
        go cb()
    }
    else {}
    <-p.m
}

// Try to complete p1 with p2.
func (p1 *Promise[T]) tryCompleteWith(p2 *Promise[T]) {
    p2.onSuccess(func(v T) {
        p1.setSucc(v)
    })
}

// Pick first successful promise.
func (p1 *Promise[T]) firstSucc(p2 *Promise[T]) *Promise[T] {
    p := newPromise[T]()
    p.tryCompleteWith(p1)
    p.tryCompleteWith(p2)
    return p
}

func example1() {
    // Book some hotel.
    booking := func() (int, bool) {
        time.Sleep((time.Duration)(rand.Intn(999)) * time.Millisecond)
        return rand.Intn(50), true
    }

    p1 := newPromise[int]()
    p1.complete(booking)

    p2 := newPromise[int]()
    p2.complete(booking)

    p := p1.firstSucc(p2)

    p.onSuccess(func(quote int) {
        fmt.Printf("Hotel asks for %d euros\n", quote)
    })
    time.Sleep(2 * time.Second)
}

func example2() {
    // Book some hotel.
    booking := func() (int, bool) {
        time.Sleep((time.Duration)(rand.Intn(999)) * time.Millisecond)
        return rand.Intn(50), true
    }

    ft1 := future[int](booking)

    ft2 := future[int](booking)

    ft := ft1.firstSucc(ft2)

    ft3.onSuccess(func(quote int) {
        fmt.Printf("Hotel asks for %d euros\n", quote)
    })

    time.Sleep(2 * time.Second)
}
```