14.8 Fernaufrufe

Die Go-Bibiothek bietet im Paket `net/rpc` eine Konstruktion von Anbietern und Kunden für Fernaufrufe an. Sie ist allerdings recht vertrackt und in Bezug auf die Möglichkeiten ihrer Verwendung in abstrakten Schichten nach dem Prinzip des „*information hiding*“ m. E. suboptimal.

Die einfachste allgemeine Form eines Fernaufrufs ist eine einstellige Funktion, im allgemeinen Fall also vom Typ `func (a Any) Any`. (Dieser Typ wird unter dem Namen `Func` im Paket `obj` untergebracht.)

Die Realisierung ist durch das Kunden-Anbieter-Paradigma gegeben: Die Möglichkeit zu einem Fernaufruf wird von einem Anbieter zur Verfügung gestellt, die von Kunden benutzt werden kann.

Weil das die gleiche Grundidee wie bei fernen Monitoren ist, sieht die Spezifikation eines entsprechenden Pakets fast genauso aus wie die der fernen Monitore:

```
package rpc
import . "nU/obj"

type RPC interface {
  F (a Any) Any
  Fin()
}
```

Der einzige Unterschied zu den fernen Monitoren besteht darin, dass bie ihnen die Monitorfunktionen auf Objekten *eines Typs* operieren, hier dagegen in der Regel der Typ des Ergebnisses vom Typ des Arguments verschieden ist. Daher muss der Konstruktor um einen Parameter des Ergebnistyps erweitert werden, in dem ein Objekt des Ergebnistyps übergeben wird. Als letzter Parameter wird die vom Kunden definierte Funktion übergeben, die auf dem rpc-Anbieter ausgeführt werden soll.

```
func New (a, b Any, h string, port uint16, s bool, f Func) RPC {
          return new_(a,b,h,port,s,f) }
```

Allerdings brauchen wir für die Implementierung noch eine Erweiterung der fernen Monitore, die den zweiten Parameter des Konstruktors – also einen weiteren Typ – berücksichtigt; also einen zweiten Konstruktor

```
func New2 (a, b Any, n uint, fs FuncSpectrum, ps PredSpectrum,
           h string, p uint16, s bool) FarMonitor {
  return new2(a,n,fs,ps,h,p,s) }
```

Der zweite Typ muss natürlich in der Repräsentation erfasst werden:

```
package fmon
import ("time"; . "nU/obj"; "nU/nchan")
```

```
type farMonitor struct {
  Any "Musterobjekt für das Argument"
  result Any "Musterobjekt für das Ergebnis"
  ... // weiter wie in der obigen Implementierung
}
```

Die Implementierung des Konstruktors `New2` ist ganz ähnlich wie die von `new_`. Da die Größen der Objekte verschieden sein können, wird den Konstruktoren für die benötigten Netzwerkkanäle als erster Parameter `nil` übergeben. Der hintere Teil – in der Methode `common` – ist mit dem des hinteren Teils von `new_` identisch:

```
func new2 (a, b Any, n uint, fs FuncSpectrum, ps PredSpectrum,
           h string, p uint16, s bool) FarMonitor {
  x := new(farMonitor)
  x.Any, x.result = Clone(a), Clone(b)
  x.uint = n
  x.ch = make([]nchan.NetChannel, x.uint)
  x.bool = s
  for i := uint(0); i < x.uint; i++ {
    x.ch[i] = nchan.NewN (nil, h, p + uint16(i), s)
  }
  return x.common (fs, ps)
}

func (x *farMonitor) common (fs FuncSpectrum, ps PredSpectrum) FarMonitor {
  in, out := make([]chan Any, x.uint), make([]chan Any, x.uint)
  for i := uint(0); i < x.uint; i++ {
    in[i], out[i] = x.ch[i].Chan()
  }
  if ! x.bool {
    return x // x ist ein Kunde
  }
  x.FuncSpectrum, x.PredSpectrum = fs, ps // x ist der Server
  any := make([]Any, x.uint)
  for i := uint(0); i < x.uint; i++ {
    ... // weiter wie in der Implementierung von new_
```

Die Implementierung unseres Fernaufruf-Pakets mit einem fernen Monitor ist damit trivial:

```
package rpc
import (. "nU/obj"; "nU/fmon")

type rpc struct {
  fmon.FarMonitor
}

func new_(a, b Any, h string, port uint16, s bool, g Func) RPC {
  x := new(rpc)
  f := func (a Any, i uint) Any { return g (a) }
```

```
  x.FarMonitor = fmon.New2 (a, b, 1, f, AllTrueSp, h, port, s)
  return x
}

func (x *rpc) F (a Any) Any {
  return x.FarMonitor.F (a, 0)
}

func (x *rpc) Fin() {
  x.FarMonitor.Fin()
}
```

14.8.1 Beispiel eines Fernaufrufs

Mit dem Typ `Intstream = []Int` im Paket `obj` und dazu passender Erweiterungen der Implementierungen der Funktionen in `obj/equaler.go` und `obj/coder.go` (s. Abschnitte 3.3.2.1 und 3.3.2.4) lässt sich das in in der Datei `net/rpc/server.go` von den Go-Autoren gegebene Beispiel, in dem das Produkt von 7 und 8 per Fernaufruf berechnet wird, wie folgt realisieren:

```
package main
import (. "nU/obj"; "nU/ego"; "nU/rpc")

func f (a Any) Any {
  p := IntStream{0, 0}
  p = Decode (p, a.(Stream)).(IntStream)
  return p[0] * p[1]
}

func main() {
  me := ego.Me()
  input, output := IntStream{7, 8}, 0
  hostname, port := ..., ...
  r := rpc.New (input, output, hostname, port, me == 0, f)
  if me == 0 { // Fernaufruf-Anbieter ist aufgerufen
    for { }
  } else { // Kunde
    output = Decode (output, r.F (input).(Stream)).(int)
    println ("7 * 8 =", uint(output))
  }
}
```

Streng genommen widerspricht die explizite Angabe der Konstruktion des Rückgabewerts des Fernaufrufs in der Funktion `f` der durchaus sinnvollen Forderung, dass dieser Quelltext eigentlich nur auf dem Anbieter-Rechner (d. h. dem Rechner, auf dem der

Anbieter-Prozess läuft) implementiert ist. Das ist aber ganz leicht dadurch zu erreichen, dass die Funktion `f` in ein eigenes Paket `f` mit der Spezifikation

```
package f
import "nU/obj"

func f (a Any) Any { return f(a) }
```

ausgelagert wird und ihre Implementierung

```
func f (a Any) Any {
  if ego.Me() == 0 {
    p := IntStream{0, 0}
    p = Decode (p, a.(Stream)).(IntStream)
    return p[0] * p[1]
  }
  return 0
}
```

nur auf dem Anbieter-Rechner diese Form hat; auf allen anderen Rechnern, auf denen `f` aufgerufen werden könnte, dagegen einfach so implementiert wird, dass sie nur einen bedeutungslosen Wert – z. B. den entsprechenden *zero-value* – liefert:

```
func f (a Any) Any {
  return 0
}
```

Damit lautet das Beispiel wie folgt:

```
package main
import (. "nU/obj"; "nU/ego"; "nU/rpc"; "f")

func main() {
  ... // die ersten drei Quelltext-Zeilen wie oben
  r := rpc.New (input, output, hostname, port, me == 0, f.F)
  ... // Rest wie oben
```

Literaturverzeichnis

1. Andrews, G. R.: Foundations of Multithreaded, Parallel and Distributed Programming. Addison-Wesley Reading (2000)
2. Ben-Ari, M.: Principles of Concurrent and Distributed Programming. Prentice Hall Hemel Hempstead (1990)
3. Hoare, C. A. R.: Monitors: An Operating Systems Structuring Concept. Commun. ACM 17 (1974) 549–557 doi: 10.1145/355620.361161
4. Raynal, M.: Concurrent Programming: Algorithms, Principles and Foundations Springer-Verlag Berlin Heidelberg (2013)