The Go Programming Language

Part 3

Rob Pike
r@google.com
October, 2009

Today's Outline

Exercise
- any questions?

Concurrency and communication
- goroutines
- channels
- concurrency issues

Exercise

Any questions?

For a trivial HTTP server with various generators, see

`http://golang.org/src/pkg/http/triv.go`

Concurrency and communication: Goroutines

Goroutines

Terminology:

There are many terms for "things that run concurrently" - process, thread, coroutine, POSIX thread, NPTL thread, lightweight process, ..., but these all mean slightly different things. None means exactly how Go does concurrency.

So we introduce a new term: goroutine.

Definition

A goroutine is a Go function or method executing concurrently in the same address space as other goroutines. A running program consists of one or more goroutines.

It's not the same as a thread, coroutine, process, etc. It's a goroutine.

There are many concurrency questions. They will be addressed later; for now just assume it all works as advertised.

Starting a goroutine

Invoke a function or method and say `go`:

```
func IsReady(what string, minutes int64) {
    time.Sleep(minutes * 60*1e9);
    fmt.Println(what, "is ready")
}

go IsReady("tea", 6);
go IsReady("coffee", 2);
fmt.Println("I'm waiting....");
```

Prints:

```
I'm waiting....   (right away)
coffee is ready   (2 min later)
tea is ready      (6 min later)
```

Some simple facts

Goroutines are cheap.

Goroutines exit by returning from their top-level function, or just falling off the end. Or they can call `runtime.Goexit()`, although that's rarely necessary.

Goroutines can run concurrently on different processors, sharing memory.

You don't have to worry about stack size.

Stacks

In `gccgo`, at least for now, goroutines are pthreads. Stacks are big. We will fix this soon (our solution requires changes to `gcc` itself).

In `6g`, however, stacks are small (a few kB) and grow as needed. Thus in `6g`, goroutines use little memory, you can have lots of them, and they can dynamically have huge stacks.

The programmer shouldn't have to think about the issue.

Scheduling

Goroutines are multiplexed as needed onto system threads. When a goroutine executes a blocking system call, no other goroutine is blocked.

We will do the same for CPU-bound goroutines at some point, but for now, if you want user-level parallelism you must set `$GOMAXPROCS`. or call `runtime.GOMAXPROCS(n)`.

`GOMAXPROCS` tells the runtime scheduler how many non-syscall-blocked goroutines to run at once.

Concurrency and communication: Channels

Channels in Go

Unless two goroutines can communicate, they can't coordinate.

Go has a type called a channel that provides communication and synchronization capabilities.

It also has special control structures that build on channels to make concurrent programming easy.

Note: I gave a talk in 2006 about predecessor stuff in an older language. The talk is on Google video; if you want more background, look for "rob pike" and "newsqueak".

The channel type

In its simplest form the type looks like this:

```
chan element_type
```

Given a value of this type, you can send and receive items of `element_type`.

Channels are a reference type, which means if you assign one `chan` variable to another, both variables access the same channel. It also means you use `make` to allocate one:

```
var c = make(chan int)
```

The communication operator: <-

The arrow points in the direction of data flow.

As a binary operator, <- sends to a channel:

```
var c chan int;
c <- 1;   // send 1 on c (flowing into c)
```

As a prefix unary operator, <- receives from a channel:

```
v = <-c;  // receive value from c, assign to v
<-c;      // receive value, throw it away
i := <-c; // receive value, initialize i
```

Semantics

By default, communication is synchronous. (We'll talk about asynchronous communication later.) This means:

1) A send operation on a channel blocks until a receiver is available for the same channel.
2) A receive operation for a channel blocks until a sender is available for the same channel.

Communication is therefore a form of synchronization: two goroutines exchanging data through a channel synchronize at the moment of communication.

Let's pump some data

```
func pump(ch chan int) {
  for i := 0; ; i++ { ch <- i }
}
```

```
ch1 := make(chan int);
go pump(ch1);           // pump hangs; we run
fmt.Println(<-ch1);   // prints 0
```

Now we start a looping receiver.

```
func suck(ch chan int) {
  for { fmt.Println(<-ch) }
}
go suck(ch1);  // tons of numbers appear
```

You can still sneak in and grab a value:

```
fmt.Println(<-ch);  // Prints 314159
```

Google

Functions returning channels

In the previous example, `pump` was like a generator spewing out values. But there was a lot of fuss allocating channels etc. Let's package it up into a function returning the channel of values.

```
func pump() chan int {
  ch := make(chan int);
  go func() {
    for i := 0; ; i++ { ch <- i }
  }();
  return ch
}

stream := pump();
fmt.Println(<-stream);  // prints 0
```

This is a very important idiom.

Channel functions everywhere

I am avoiding repeating famous examples you can find elsewhere. Here are a couple to look up:

1) The prime sieve; in the language specification and also in the tutorial.

2) Doug McIlroy's power series work.
  Read This Paper!
Look for "Doug McIlroy" "Squinting at Power Series".
The Go version of this program is in available in the test suite as

`http://golang.org/test/chan/powser1.go`

Range and channels

The `range` clause on `for` loops accepts a channel as an operand, in which case the `for` loops over the values received from the channel. We rewrote `pump`; here's the rewrite for `suck`, making it launch the goroutine as well:

```
func suck(ch chan int) {
  go func() {
    for v := range ch { fmt.Println(v) }
  }()
}

suck(pump());  // doesn't block now
```

Closing a channel

What if we want to signal that a channel is done? We close it with a built-in function:

```
close(ch)
```

and test that state using `closed()`:

```
if closed(ch) { fmt.Println("done") }
```

Once a channel is closed and every sent value has been received, every subsequent receive operation will recover a zero value.

In practice, you rarely need `close` except in certain idiomatic situations.

When a channel closes

There are subtleties about races, so `closed()` succeeds only after you receive one zero value on the channel. The obvious loop isn't right. To use `closed()` correctly you need to say:

```
for {
  v := <-ch;
  if closed(ch) { break }
  fmt.Println(v)
}
```

But of course, the `range` clause does that for you.

```
for v := range ch {
  fmt.Println(v)
}
```

Iterators

Now we have all the pieces to write an iterator for a container. Here is the code for `Vector`:

```
// Iterate over all elements
func (p *Vector) iterate(c chan Element) {
  for i, v := range p.a {  // p.a is a slice
    c <- v
  }
  close(c);  // signal no more values
}

// Channel iterator.
func (p *Vector) Iter() chan Element {
  c := make(chan Element);
  go p.iterate(c);
  return c;
}
```

Using the iterator

Now that `Vector` has an iterator, we can use it:

```
vec := new(vector.Vector);
for i := 0; i < 100; i++ {
  vec.Push(i*i)
}

i := 0;
for x := range vec.Iter() {
  fmt.Printf("vec[%d] is %d\n", i, x.(int));
  i++;
}
```

Channel directionality

In its simplest form a channel variable is an unbuffered (synchronous) value that can be used to send and receive.

A channel type may be annotated to specify that it may only send or only receive:

```
var recv_only <-chan int;
var send_only chan<- int;
```

Channel directionality (II)

All channels are created bidirectional, but we can assign them to directional channel variables. Useful for instance in functions, for (type) safety:

```
func sink(ch <-chan int) {
  for { <-ch }
}
func source(ch chan<- int) {
  for { ch <- 1 }
}

var c = make(chan int);  // bidirectional
go source(c);
go sink(c);
```

Synchronous channels

Synchronous channels are unbuffered. Sends do not complete until a receiver has accepted the value.

```
c := make(chan int);
go func() {
  time.Sleep(60*1e9);
  x := <-c;
  fmt.Println("received", x);
}();

fmt.Println("sending", 10);
c <- 10;
fmt.Println("sent", 10);
```

Output:

```
sending 10  (happens immediately)
sent 10     (60s later, these 2 lines appear)
received 10
```

Asynchronous channels

A buffered, asynchronous channel can be created by giving `make` an argument, the number of elements in the buffer.

```
c := make(chan int, 50);
go func() {
  time.Sleep(60*1e9);
  x := <-c;
  fmt.Println("received", x);
}();

fmt.Println("sending", 10);
c <- 10;
fmt.Println("sent", 10);
```

Output:

```
sending 10  (happens immediately)
sent 10     (now)
received 10 (60s later)
```

Buffer is not part of the type

Note that the buffer's size, or even its existence, is not part of the channel's type, only of the value. This code is therefore legal, although dangerous:

```
buf   = make(chan int, 1);
unbuf = make(chan int);
buf   = unbuf;
unbuf = buf;
```

Buffering is a property of the value, not of the type.

Testing for communicability

Can a receive proceed without blocking? Need to find out and execute, or not, atomically.
"Comma ok" to the rescue:

```
v, ok = <-c; // ok=true if v received value
```

Can a send proceed without blocking? Need to find out and execute, or not, atomically. Use send as a boolean expression:

```
ok := c <- v;
```

or

```
if c <- v { fmt.Println("sent value") }
```

But usually you want more...

Google

Select

Select is a control structure in Go analogous to a communications switch statement. Each case must be a communication, either send or receive.

```
var c1, c2 chan int;

select {
case v := <-c1:
  fmt.Printf("received %d from c1\n", v)
case v := <-c2:
  fmt.Printf("received %d from c2\n", v)
}
```

Select executes one runnable case at random. If no case is runnable, it blocks until one is. A `default` clause is always runnable.

Select semantics

Quick summary:

- Every case must be a (possibly `:=`) communication
- All channel expressions are evaluated
- All expressions to be sent are evaluated
- If any communication can proceed, it does; others are ignored
- Otherwise:
  - If there is a `default` clause, that runs
  - If there is no `default`, select statement blocks until one communication can proceed; there is no re-evaluation of channels or values
- If multiple cases are ready, one is selected at random, fairly. Others do not execute.

Random bit generator

Silly but illustrative example.

```
c := make(chan int);
go func() {
  for {
    fmt.Println(<-c)
  }
}();
```

```
for {
  select {
  case c <- 0:  // no stmt, no fall through
  case c <- 1:
  }
}
```

Prints `0 1 1 0 0 1 1 1 0 1 ...`

Google

Multiplexing

Channels are first-class values, which means they can be sent over channels. This property makes it easy to write a service multiplexer since the client can supply, along with its request, the channel on which to reply.

```
chanOfChans := make(chan chan int)
```

Or more typically

```
type Reply struct { ... }
type Request struct {
  arg1, arg2, arg3 some_type;
  replyc chan *Reply;
}
```

Multiplexing server

```
type request struct {
    a, b    int;
    replyc  chan int;
}

type binOp func(a, b int) int

func run(op binOp, req *request) {
    req.replyc <- op(req.a, req.b)
}

func server(op binOp, service chan *request) {
    for {
     req := <-service; // requests arrive here
     go run(op, req);  // don't wait for op
    }
}
```

Starting the server

Use the channel function pattern to create a channel to a new server:

```
func startServer(op binOp) chan *request {
   req := make(chan *request);
   go server(op, req);
   return req
}

 var adderChan = startServer(
   func(a, b int) int { return a + b }
 )
```

The client

A similar example is worked in more detail in the tutorial, but here's a variant:

```
func (r *request) String() string {
  return fmt.Sprintf("%d+%d=%d",
         r.a, r.b, <-r.replyc)
}

req1 := &request{ 7, 8, make(chan int) };
req2 := &request{ 17, 18, make(chan int) };
```

Requests ready; send them:

```
adderChan <- req1;
adderChan <- req2;
```

Can retrieve results in any order; `r.replyc` demuxes:

```
fmt.Println(req2, req1);
```

Teardown

In the mux example, the server runs forever. To tear it down cleanly, signal with a channel. This server has the same functionality but with a `quit` channel:

```
func server(op binOp, service chan *request,
            quit chan bool) {
  for {
    select {
    case req := <-service:
      go run(op, req);  // don't wait for it
    case <-quit:
      return;
    }
  }
}
```

Starting the server

The rest of the code is mostly the same, with an extra channel:

```
func startServer(op binOp) (service chan *request,
                             quit chan bool) {
  service = make(chan *request);
  quit = make(chan bool);
  go server(op, service, quit);
  return service, quit;
}

var adderChan, quitChan := startServer(
  func(a, b int) int { return a + b }
)
```

Teardown: the client

The client is unaffected until it's ready to shut down the server:

```
req1 := &request{7, 8, make(chan int)};
req2 := &request{17, 18, make(chan int)};

adderChan <- req1;
adderChan <- req2;

fmt.Println(req2, req1);
```

All done, signal server to exit:

```
 quitChan <- true;
```

Chaining

```
package main

import ("flag"; "fmt")

var ngoroutine = flag.Int("n", 100000, "how many")

func f(left, right chan int) { left <- 1 + <-right }

func main() {
  flag.Parse();
  leftmost := make(chan int);
  var left, right chan int = nil, leftmost;
  for i := 0; i < *ngoroutine; i++ {
    left, right = right, make(chan int);
    go f(left, right);
  }
  right <- 0;  // bang!
  x := <-leftmost;  // wait for completion
  fmt.Println(x);   // 100000
}
```

Example: Leaky bucket

Queue of shared, reusable buffers

```
var freeList = make(chan *Buffer, 100)
var serverChan = make(chan *Buffer)

func server() {
  for {
    b := <-serverChan;  // wait for work
    process(b);
    ok := freeList <- b  // reuse buffer if room
  }
}
func client() {
  for {
    b, ok := <-freeList;  // grab one if available
    if !ok { b = new(Buffer) }
    load(b);
    serverChan <- b  // send to server
  }
}
```

Concurrency

Concurrency issues

Many issues, of course, but Go tries to take care of them. Channel send and receive are atomic. The `select` statement is very carefully defined and implemented, etc.

But goroutines run in shared memory, communication networks can deadlock, multithreaded debuggers suck, and so on.

What to do?

Go gives you the primitives

Don't program the way you would in C or C++ or even Java.

Channels give you synchronization and communication both, and that makes them powerful but also easy to reason about if you use them well.

The rule is:

> Do not communicate by sharing memory.
> Instead, share memory by communicating.

The very act of communication guarantees synchronization!

The model

For instance, use a channel to send data to a dedicated server goroutine. If only one goroutine at a time has a pointer to the data, there are no concurrency issues.

This is the model we advocate for programming servers, at least. It's the old "one thread per client" approach, generalized – and used by us since the 1980s. It works very well.

My old talk on Google video goes into this idea in more depth.

The memory model

The nasty details about synchronization and shared memory are written down at:

`http://golang.org/doc/go_mem.html`

But you rarely need to understand them if you follow our approach.

The
Go
Programming Language

Part 3

Rob Pike
r@google.com
October, 2009