Coroutines for Go

Russ Cox
July 17, 2023

research.swtch.com/coro

This post is about why we need a coroutine package for Go, and what it would look like. But first, what are coroutines?

Every programmer today is familiar with function calls (subroutines): F calls G, which stops F and runs G. G does its work, potentially calling and waiting for other functions, and eventually returns. When G returns, G is gone and F continues running. In this pattern, only one function is running at a time, while its callers wait, all the way up the call stack.

In contrast to subroutines, coroutines run concurrently on different stacks, but it's still true that only one is running at a time, while its caller waits. F starts G, but G does not run immediately. Instead, F must explicitly *resume* G, which then starts running. At any point, G may turn around and *yield* back to F. That pauses G and continues F from its resume operation. Eventually F calls resume again, which pauses F and continues G from its yield. On and on they go, back and forth, until G returns, which cleans up G and continues F from its most recent resume, with some signal to F that G is done and that F should no longer try to resume G. In this pattern, only one coroutine is running at a time, while its caller waits on a different stack. They take turns in a well-defined, coordinated manner.

This is a bit abstract. Let's look at real programs.

Coroutines in Lua

To use a venerable example*, consider comparing two binary trees to see if they have the same value sequence, even if their structures are different. For example, here is code in Lua 5* to generate some binary trees:

```
function T(l, v, r)
    return {left = l, value = v, right = r}
end

e = nil
t1 = T(T(T(e, 1, e), 2, T(e, 3, e)), 4, T(e, 5, e))
t2 = T(e, 1, T(e, 2, T(e, 3, T(e, 4, T(e, 5, e)))))
t3 = T(e, 1, T(e, 2, T(e, 3, T(e, 4, T(e, 6, e)))))
```

The trees `t1` and `t2` both contain the values 1, 2, 3, 4, 5; `t3` contains 1, 2, 3, 4, 6.

We can write a coroutine to walk over a tree and yield each value:

```
function visit(t)
    if t ~= nil then  -- note: ~= is "not equal"
        visit(t.left)
        coroutine.yield(t.value)
        visit(t.right)
    end
end
```

Then to compare two trees, we can create two visit coroutines and alternate between them to read and compare successive values:

```
function cmp(t1, t2)
    co1 = coroutine.create(visit)
    co2 = coroutine.create(visit)
    while true
    do
        ok1, v1 = coroutine.resume(co1, t1)
        ok2, v2 = coroutine.resume(co2, t2)
        if ok1 ~= ok2 or v1 ~= v2 then
            return false
        end
        if not ok1 and not ok2 then
            return true
        end
    end
end
```

The `t1` and `t2` arguments to `coroutine.resume` are only used on the first iteration, as the argument to `visit`. Subsequent resumes return that value from `coroutine.yield`, but the code ignores the value.

A more idiomatic Lua version would use `coroutine.wrap`, which returns a function that hides the coroutine object:

```
function cmp(t1, t2)
    next1 = coroutine.wrap(function() visit(t1) end)
    next2 = coroutine.wrap(function() visit(t2) end)
    while true
    do
        v1 = next1()
        v2 = next2()
        if v1 ~= v2 then
            return false
        end
        if v1 == nil and v2 == nil then
            return true
        end
    end
end
```

When the coroutine has finished, the `next` function returns `nil` (full code*).

Generators in Python (Iterators in CLU)

Python provides generators that look a lot like Lua's coroutines, but they are not coroutines, so it's worth pointing out the differences. The main difference is that the "obvious" programs don't work. For example, here's a direct translation of our Lua tree and visitor to Python:

```
def T(l, v, r):
    return {'left': l, 'value': v, 'right': r}

def visit(t):
    if t is not None:
        visit(t['left'])
        yield t['value']
        visit(t['right'])
```

But this obvious translation doesn't work:

```
>>> e = None
>>> t1 = T(T(T(e, 1, e), 2, T(e, 3, e)), 4, T(e, 5, e))
>>> for x in visit(t1):
...     print(x)
...
4
>>>
```

We lost 1, 2, 3, and 5. What happened?

In Python, that `def visit` does not define an ordinary function. Because the body contains a `yield` statement, the result is a generator instead:

```
>>> type(visit(t1))
<class 'generator'>
>>>
```

The call `visit(t['left'])` doesn't run the code in `visit` at all. It only creates and returns a new generator, which is then discarded. To avoid discarding those results, you have to loop over the generator and re-yield them:

```
def visit(t):
    if t is not None:
        for x in visit(t['left']):
            yield x
        yield t['value']
        for x in visit(t['right'])
            yield x
```

Python 3.3 introduced `yield from`, allowing:

```
def visit(t):
    if t is not None:
        yield from visit(t['left']):
        yield t['value']
        yield from visit(t['right'])
```

The generator object contains the state of the single call to `visit`, meaning local variable values and which line is executing. That state is pushed onto the call stack each time the generator is resumed and then popped back into the generator object at each `yield`, which can only occur in the top-most call frame. In this way, the generator uses the same stack as the original program, avoiding the need for a full coroutine implementation but introducing these confusing limitations instead.

Python's generators appear to be almost exactly copied from CLU, which pioneered this abstraction (and so many other things), although CLU calls them iterators, not generators. A CLU tree iterator looks like:

```
visit = iter (t: cvt) yields (int):
    tagcase t
        tag empty: ;
        tag non_empty(t: node):
            for x: int
                in tree$visit(t.left) do
                    yield(x);
                    end;
            yield(t.value);
```

```
            for x: int
                in tree$visit(t.right) do
                    yield(x);
                    end;
        end;
    end visit;
```

The syntax is different, especially the `tagcase` that is examining a tagged union representation of a tree, but the basic structure, including the nested `for` loops, is exactly the same as our first working Python version. Also, because CLU was statically typed, `visit` is clearly marked as an iterator (`iter`) not a function (`proc` in CLU). Thanks to that type information, misuse of `visit` as an ordinary function call, like in our buggy Python example, is something that the compiler could (and I assume did) diagnose.

About CLU's implementation, the original implementers wrote, "Iterators are a form of coroutine; however, their use is sufficiently constrained that they are implemented using just the program stack. Using an iterator is therefore only slightly more expensive than using a procedure." This sounds exactly like the explanation I gave above for the Python generators. For more, see Barbara Liskov *et al.*'s 1977 paper "Abstraction Mechanisms in CLU*", specifically sections 4.2, 4.3, and 6.

Coroutines, Threads, and Generators

At first glance, coroutines, threads, and generators look alike. All three provide concurrency* in one form or another, but they differ in important ways.

- Coroutines provide concurrency without parallelism: when one coroutine is running, the one that resumed it or yielded to it is not.

  Because coroutines run one at a time and only switch at specific points in the program, the coroutines can share data among themselves without races. The explicit switches (`coroutine.resume` in the first Lua example or calling a `next` function in the second Lua example) serve as synchronization points, creating happens-before edges*.

  Because scheduling is explicit (without any preemption) and done entirely without the operating system, a coroutine switch takes at most around ten nanoseconds, usually even less. Startup and teardown is also much cheaper than threads.

- Threads provide more power than coroutines, but with more cost. The additional power is parallelism, and the cost is the overhead of scheduling, including more expensive context switches and the need to add preemption in some form. Typically the operating system provides threads, and a thread switch takes a few microseconds.

  For this taxonomy, Go's goroutines are cheap threads: a goroutine switch is closer to a few hundred nanoseconds, because the Go runtime takes on some of the scheduling work, but goroutines still provide the full parallelism and preemption of threads. (Java's new lightweight threads are basically the same as goroutines.)

- Generators provide less power than coroutines, because only the topmost frame in the coroutine is allowed to yield. That frame is moved back and forth between an object and the call stack to suspend and resume it.

Coroutines are a useful building block for writing programs that want concurrency for program structuring but not for parallelism. For one detailed example of that, see my previous post, "Storing Data in Control Flow*". For other ex-

amples, see Ana Lúcia De Moura and Roberto Ierusalimschy's 2009 paper "Revisiting Coroutines*". For the original example, see Melvin Conway's 1963 paper "Design of a Separable Transition-Diagram Compiler*".

Why Coroutines in Go?

Coroutines are a concurrency pattern not directly served by existing Go concurrency libraries. Goroutines are often close enough, but as we saw, they are not the same, and sometimes that difference matters.

For example, Rob Pike's 2011 talk "Lexical Scanning in Go*" presents the original lexer and parser for the text/template package*. They ran in separate goroutines connected by a channel, imperfectly simulating a pair of coroutines: the lexer and parser ran in parallel, with the lexer looking ahead to the next token while the parser processed the most recent one. Generators would not have been good enough—the lexer yields values from many different functions—but full goroutines proved to be a bit too much. The parallelism provided by the goroutines caused races and eventually led to abandoning the design in favor of the lexer storing state in an object, which was a more faithful simulation of a coroutine. Proper coroutines would have avoided the races and been more efficient than goroutines.

An anticipated future use case for coroutines in Go is iteration over generic collections. We have discussed adding support to Go for ranging over functions*, which would encourage authors of collections and other abstractions to provide CLU-like iterator functions. Iterators can be implemented today using function values, without any language changes. For example, a slightly simplified tree iterator in Go could be:

```
func (t *Tree[V]) All(yield func(v V)) {
    if t != nil {
        t.left.All(yield)
        yield(t.value)
        t.right.All(yield)
    }
}
```

That iterator can be invoked today as:

```
t.All(func(v V) {
    fmt.Println(v)
})
```

and perhaps a variant could be invoked in a future version of Go as:

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

Sometimes, however, we want to iterate over a collection in a way that doesn't fit a single `for` loop. The binary tree comparison is an example of this: the two iterations need to be interlaced somehow. As we've already seen, coroutines would provide an answer, letting us turn a function like `(*Tree).All` (a "push" iterator) into a function that returns a stream of values, one per call (a "pull" iterator).

How to Implement Coroutines in Go

If we are to add coroutines to Go, we should aim to do it without language changes. That means the definition of coroutines should be possible to implement and understand in terms of ordinary Go code. Later, I will argue for an optimized implementation provided directly by the runtime, but that implementation should be indistinguishable from the pure Go definition.

Let's start with a very simple version that ignores the yield operation entirely. It just runs a function in another goroutine:

```
package coro

func New[In, Out any](f func(In) Out) (resume func(In) Out) {
    cin := make(chan In)
    cout := make(chan Out)
    resume = func(in In) Out {
        cin <- in
        return <-cout
    }
    go func() { cout <- f(<-cin) }()
    return resume
}
```

`New` takes a function `f` which must have one argument and one result. `New` allocates channels, defines `resume`, creates a goroutine to run `f`, and returns the `resume` funtion. The new goroutine blocks on `<-cin`, so there is no opportunity for parallelism. The `resume` function unblocks the new goroutine by sending an `in` value and then blocks receiving an `out` value. This send-receive pair makes a coroutine switch. We can use `coro.New` like this (full code*):

```
func main() {
    resume := coro.New(strings.ToUpper)
    fmt.Println(resume("hello world"))
}
```

So far, `coro.New` is just a clunky way to call a function. We need to add `yield`, which we can pass as an argument to `f`:

```
func New[In, Out any](f func(in In, yield func(Out) In) Out) (
        resume func(In) Out) {
    cin := make(chan In)
    cout := make(chan Out)
    resume = func(in In) Out {
        cin <- in
        return <-cout
    }
    yield := func(out Out) In {
        cout <- out
        return <-cin
    }
    go func() { cout <- f(<-cin, yield) }()
    return resume
}
```

Note that there is still no parallelism here: `yield` is another send-receive pair. These goroutines are constrained by the communication pattern to act indistinguishably from coroutines.

Example: String Parser

Before we build up to iterator conversion, let's look at a few simpler examples. In "Storing Data in Control Flow*," we considered the problem of taking a function

```
func parseQuoted(read func() byte) bool
```

and running it in a separate control flow so that bytes can be provided one at a time to a `Write` method. Instead of the ad hoc channel-based implementation in that post, we can use:

```
type parser struct {
    resume func(byte) Status
}

func (p *parser) Init() {
    coparse := func(_ byte, yield func(Status) byte) Status {
        read := func() byte { return yield(NeedMoreInput) }
        if !parseQuoted(read) {
            return BadInput
        }
        return Success
    }
    p.resume = coro.New(coparse)
    p.resume(0)
}

func (p *parser) Write(c byte) Status {
    return p.resume(c)
}
```

The `Init` funtion does all the work, and not much. It defines a function `coparse` that has the signature needed by `coro.New`, which means adding a throwaway input of type `byte`. That function defines a `read` that yields `NeedMoreInput` and then returns the byte provided by the caller. It then runs `parseQuoted(read)`, converting the boolean result to the usual status code. Having created a coroutine for `coparse` using `coro.New`, `Init` calls `p.resume(0)` to allow `coparse` to advance to the first read in `parseQuoted`. Finally the `Write` method is a trivial wrapper around `p.resume` (full code*).

This setup abstracts away the pair of channels that we maintained by hand in the previous post, allowing us to work at a higher level as we write the program.

Example: Prime Sieve

As a slightly larger example, consider Doug McIlroy's concurrent prime sieve*. It consists of a pipeline of coroutines, one for each prime `p`, each running:

```
loop:
    n = get a number from left neighbor
    if (p does not divide n)
        pass n to right neighbor
```

A counting coroutine on the leftmost side of the pipeline feeds the numbers 2, 3, 4, ... into the left end of the pipeline. A printing coroutines on the rightmost side can read primes out, print them, and create new filtering coroutines. The first filter in the pipeline removes multiples of 2, the next removes multiples of 3, the next removes multiples of 5, and so on.

The `coro.New` primitive we've created lets us take a straightforward loop that yields values and convert it into a function that can be called to obtain each value one at a time. Here is the counter:

```
func counter() func(bool) int {
    return coro.New(func(more bool, yield func(int) bool) int {
        for i := 2; more; i++ {
            more = yield(i)
        }
        return 0
    })
}
```

The counter logic is the function literal passed to `New`. It takes a yield function of type `func(int) bool`. The code yields a value by passing it to `yield` and then receives back a boolean saying whether to continue generating more numbers. When told to stop, either because `more` was false on entry or because a `yield` call returned false, the loop ends. It returns a final, ignored value, to satisfy the function type required by `New`.

`New` turns this into loop a function that is the inverse of `yield`: a `func(bool) int` that can be called with true to obtain the next value or with false to shut down the generator. The filtering coroutine is only slightly more complex:

```
func filter(p int, next func(bool) int) (filtered func(bool) int) {
    return coro.New(func(more bool, yield func(int) bool) int {
        for more {
            n := next(true)
            if n%p != 0 {
                more = yield(n)
            }
        }
        return next(false)
    })
}
```

It takes a prime `p` and a `next` func connected to the coroutine on the left and then returns the filtered output stream to connect to the coroutine on the right.

Finally we have the printing coroutine:

```
func main() {
    next := counter()
    for i := 0; i < 10; i++ {
        p := next(true)
        fmt.Println(p)
        next = filter(p, next)
    }
    next(false)
}
```

Starting with the counter, `main` maintains in `next` the output of the pipeline constructed so far. Then it loops: read a prime `p`, print `p`, and then add a new filter on the right end of the pipeline to remove multiples of `p` (full code*).

Notice that the calling relationship between coroutines can change over time: any coroutine C can call another coroutine D's `next` function and become the coroutine that D yields to. The counter's first `yield` goes to `main`, while its subsequent `yield`s go to the 2-filter. Similarly each `p`-filter `yield`s its first output (the next prime) to `main` while its subsequent `yield`s go to the filter for that next prime.

Coroutines and Goroutines

In a certain sense, it is a misnomer to call these control flows coroutines. They are full goroutines, and they can do everything an ordinary goroutine can, including block waiting for mutexes, channels, system calls, and so on. What `coro.New` does is create goroutines with access to coroutine switch operations inside the `yield` and `resume` functions (which the sieve calls `next`). The ability to use those operations can even be passed to different goroutines, which is happening with `main` handing off each of its `next` streams to each successive `filter` goroutine. Unlike the `go` statement, `coro.New` adds new concurrency to the program *without* new parallelism. The goroutine that `coro.New(f)` creates can only run when some other goroutine explicitly loans it the ability to run using `resume`; that loan is repaid by `yield` or by `f` returning. If you have just one main goroutine and run 10 `go` statements, then all 11 goroutines can be running at once. In contrast, if you have one main goroutine and run 10 `coro.New` calls, there are now 11 control flows but the parallelism of the program is what it was before: only one runs at a time. Exactly which goroutines are paused in coroutine operations can vary as the program runs, but the parallelism never increases.

In short, `go` creates a new concurrent, *parallel* control flow, while `coro.New` creates a new concurrent, *non-parallel* control flow. It is convenient to continue to talk about the non-parallel control flows as coroutines, but remember that exactly which goroutines are "non-parallel" can change over the execution of a program, exactly the same way that which goroutines are receiving or sending from channels can change over the execution of a program.

Robust Resumes

There are a few improvements we can make to `coro.New` so that it works better in real programs. The first is to allow `resume` to be called after the function is done: right now it deadlocks. Let's add a bool result indicating whether `resume`'s result came from a yield. The `coro.New` implementation we have so far is:

```
func New[In, Out any](f func(in In, yield func(Out) In) Out) (
        resume func(In) Out) {
    cin := make(chan In)
    cout := make(chan Out)
    resume = func(in In) Out {
        cin <- in
        return <-cout
    }
    yield := func(out Out) In {
        cout <- out
        return <-cin
    }
    go func() {
        cout <- f(<-cin, yield)
    }()
    return resume
}
```

To add this extra result, we need to track whether `f` is running and return that result from `resume`:

```
func New[In, Out any](f func(in In, yield func(Out) In) Out) (
        resume func(In) (Out, bool)) {
    cin := make(chan In)
    cout := make(chan Out)
    running := true
    resume = func(in In) (out Out, ok bool) {
        if !running {
            return
        }
        cin <- in
        out = <-cout
        return out, running
    }
    yield := func(out Out) In {
        cout <- out
        return <-cin
    }
    go func() {
        out := f(<-cin, yield)
        running = false
        cout <- out
    }()
    return resume
}
```

Note that since `resume` can only run when the calling goroutine is blocked, and vice versa, sharing the `running` variable is not a race. The two are synchronizing by taking turns executing. If `resume` is called after the coroutine has exited, `resume` returns a zero value and false.

Now we can tell when a goroutine is done (full code*):

```
func main() {
    resume := coro.New(func(_ int, yield func(string) int) string {
        yield("hello")
        yield("world")
        return "done"
    })
    for i := 0; i < 4; i++ {
        s, ok := resume(0)
        fmt.Printf("%q %v\n", s, ok)
    }
}
```

```
$ go run cohello.go
"hello" true
"world" true
"done" false
"" false
$
```

Example: Iterator Conversion

The prime sieve example showed direct use of `coro.New`, but the `more bool` argument was a bit awkward and does not match the iterator functions we saw before. Let's look at converting any push iterator into a pull iterator using `coro.New`. We will need a way to terminate the coroutine running the push iterator if we want to stop early, so we will add a boolean result from `yield` indicating whether to continue, just like in the prime sieve:

```
push func(yield func(V) bool)
```

The goal of the new function `coro.Pull` is to turn that push function into a pull iterator. The iterator will return the next value and a boolean indicating whether the iteration is over, just like a channel receive or map lookup:

```
pull func() (V, bool)
```

If we want to stop the push iteration early, we need some way to signal that, so `Pull` will return not just the pull function but also a stop function:

```
stop func()
```

Putting those together, the full signature of `Pull` is:

```
func Pull[V any](push func(yield func(V) bool)) (
        pull func() (V, bool), stop func()) {
    ...
}
```

The first thing `Pull` needs to do is start a coroutine to run the push iterator, and to do that it needs a wrapper function with the right type, namely one that takes a `more bool` to match the bool result from `yield`, and that returns a final `V`. The `pull` function can call `resume(true)`, while the `stop` function can call `resume(false)`:

```
func Pull[V any](push func(yield func(V) bool)) (
        pull func() (V, bool), stop func()) {
    copush := func(more bool, yield func(V) bool) V {
        if more {
            push(yield)
        }
        var zero V
        return zero
    }
    resume := coro.New(copush)
    pull = func() (V, bool) {
        return resume(true)
    }
    stop = func() {
        resume(false)
    }
    return pull, stop
}
```

That's the complete implementation. With the power of `coro.New`, it took very little code and effort to build a nice iterator converter.

To use `coro.Pull`, we need to redefine the tree's `All` method to expect and use the new `bool` result from `yield`:

```
func (t *Tree[V]) All(yield func(v V) bool) {
    t.all(yield)
}

func (t *Tree[V]) all(yield func(v V) bool) bool {
    return t == nil ||
        t.Left.all(yield) && yield(t.Value) && t.Right.all(yield)
}
```

Now we have everything we need to write a tree comparison function in Go (full code*):

```
func cmp[V comparable](t1, t2 *Tree[V]) bool {
    next1, stop1 := coro.Pull(t1.All)
    next2, stop2 := coro.Pull(t2.All)
    defer stop1()
    defer stop2()
    for {
        v1, ok1 := next1()
        v2, ok2 := next2()
        if v1 != v2 || ok1 != ok2 {
            return false
        }
        if !ok1 && !ok2 {
            return true
        }
    }
}
```

Propagating Panics

Another improvement is to pass panics from a coroutine back to its caller, meaning the coroutine that most recently called `resume` to run it (and is therefore sitting blocked in `resume` waiting for it). Some mechanism to inform one goroutine when another panics is a very common request, but in general that can be difficult, because we don't know which goroutine to inform and whether it is ready to hear that message. In the case of coroutines, we have the caller blocked waiting for news, so it makes sense to deliver news of the panic.

To do that, we can add a `defer` to catch a panic in the new coroutine and trigger it again in the `resume` that is waiting.

```
type msg[T any] struct {
    panic any
    val   T
}

func New[In, Out any](f func(in In, yield func(Out) In) Out) (
        resume func(In) (Out, bool)) {
    cin := make(chan In)
    cout := make(chan msg[Out])
    running := true
    resume = func(in In) (out Out, ok bool) {
        if !running {
            return
        }
        cin <- in
        m := <-cout
        if m.panic != nil {
            panic(m.panic)
        }
        return m.val, running
    }
    yield := func(out Out) In {
        cout <- msg[Out]{val: out}
        return <-cin
    }
    go func() {
        defer func() {
            if running {
                running = false
                cout <- msg[Out]{panic: recover()}
            }
        }()
        out := f(<-cin, yield)
        running = false
        cout <- msg[Out]{val: out}
    }()
    return resume
}
```

Let's test it out (full code*):

```
func main() {
    defer func() {
        if e := recover(); e != nil {
            fmt.Println("main panic:", e)
            panic(e)
        }
    }()
    next, _ := coro.Pull(func(yield func(string) bool) {
        yield("hello")
        panic("world")
    })
    for {
        fmt.Println(next())
    }
}
```

The new coroutine yields `hello` and then panics `world`. That panic is propagated back to the main goroutine, which prints the value and repanics. We can see that the panic appears to originate in the call to `resume`:

```
% go run coro.go
hello true
main panic: world
panic: world [recovered]
    panic: world

goroutine 1 [running]:
main.main.func1()
    /tmp/coro.go:9 +0x95
panic({0x108f360?, 0x10c2cf0?})
    /go/src/runtime/panic.go:1003 +0x225
main.coro_New[...].func1()
    /tmp/coro.go.go:55 +0x91
main.Pull[...].func2()
    /tmp/coro.go.go:31 +0x1c
main.main()
    /tmp/coro.go.go:17 +0x52
exit status 2
%
```

Cancellation

Panic propagation takes care of telling the caller about an early coroutine exit, but what about telling a coroutine about an early caller exit? Analogous to the `stop` function in the pull iterator, we need some way to signal to the coroutine that it's no longer needed, perhaps because the caller is panicking, or perhaps because the caller is simply returning.

To do that, we can change `coro.New` to return not just `resume` but also a `cancel` func. Calling `cancel` will be like `resume`, except that `yield` panics instead of returning a value. If a coroutine panics in a different way during cancellation, we want `cancel` to propagate that panic, just as `resume` does. But of course we don't want `cancel` to propagate its own panic, so we create a unique panic value we can check for. We also have to handle a cancellation in before `f` begins.

```
var ErrCanceled = errors.New("coroutine canceled")

func New[In, Out any](f func(in In, yield func(Out) In) Out) (
        resume func(In) (Out, bool), cancel func()) {
    cin := make(chan msg[In])
    cout := make(chan msg[Out])
    running := true
    resume = func(in In) (out Out, ok bool) {
        if !running {
            return
        }
        cin <- msg[In]{val: in}
        m := <-cout
        if m.panic != nil {
            panic(m.panic)
        }
        return m.val, running
    }
    cancel = func() {
        e := fmt.Errorf("%w", ErrCanceled) // unique wrapper
        cin <- msg[In]{panic: e}
        m := <-cout
        if m.panic != nil && m.panic != e {
            panic(m.panic)
        }
    }
    yield := func(out Out) In {
        cout <- msg[Out]{val: out}
        m := <-cin
        if m.panic != nil {
            panic(m.panic)
        }
        return m.val
    }
    go func() {
        defer func() {
            if running {
                running = false
                cout <- msg[Out]{panic: recover()}
            }
        }()
        var out Out
        m := <-cin
        if m.panic == nil {
            out = f(m.val, yield)
        }
        running = false
        cout <- msg[Out]{val: out}
    }()
    return resume, cancel
}
```

We could change `Pull` to use panics to cancel iterators as well, but in that context the explicit `bool` seems clearer, especially since stopping an iterator is unexceptional.

Example: Prime Sieve Revisited

Let's look at how panic propagation and cancellation make cleanup of the prime sieve "just work". First let's update the sieve to use the new API. The `counter` and `filter` functions are already "one-line" `return coro.New(...)` calls. They change signature to include the additional cancel func returned from `coro.New`:

```
func counter() (func(bool) (int, bool), func()) {
    return coro.New(...)
}

func filter(p int, next func(bool) (int, bool)) (
        func(bool) (int, bool), func()) {
    return coro.New(...)
}
```

Then let's convert the `main` function to be a `primes` function that prints `n` primes (full code*):

```
func primes(n int) {
    next, cancel := counter()
    defer cancel()
    for i := 0; i < n; i++ {
        p, _ := next(true)
        fmt.Println(p)
        next, cancel = filter(p, next)
        defer cancel()
    }
}
```

When this function runs, after it has gotten `n` primes, it returns. Each of the deferred `cancel` calls cleans up the coroutines that were created. And what if one of the coroutines has a bug and panics? If the coroutine was resumed by a `next` call in `primes`, then the panic comes back to `primes`, and `primes`'s deferred `cancel` calls clean up all the other coroutines. If the coroutine was resumed by a `next` call in a `filter` coroutine, then the panic will propagate up to the waiting `filter` coroutine and then the next waiting `filter` coroutine, and so on, until it gets to the `p := next(true)` in `primes`, which will again clean up the remaining coroutines.

API

The API we've arrived at is:

> New creates a new, paused coroutine ready to run the function f. The new coroutine is a goroutine that never runs on its own: it only runs while some other goroutine invokes and waits for it, by calling resume or cancel.
>
> A goroutine can pause itself and switch to the new coroutine by calling resume(in). The first call to resume starts f(in, yield). Resume blocks while f runs, until either f calls yield(out) or returns out. When f calls yield, yield blocks and resume returns out, true. When f returns, resume returns out, false. When resume has returned due to a yield, the next resume(in) switches back to f, with yield returning in.
>
> Cancel stops the execution of f and shuts down the coroutine. If resume has not been called, then f does not run at all. Otherwise, cancel causes the blocked yield call to panic with an error satisfying errors.Is(err, ErrCanceled).

> If f panics and does not recover the panic, the panic is stopped in f's coroutine and restarted in the goroutine waiting for f, by causing the blocked resume or cancel that is waiting to re-panic with the same panic value. Cancel does not re-panic when f's panic is one that cancel itself triggered.
>
> Once f has returned or panicked, the coroutine no longer exists. Subsequent calls to resume return zero, false. Subsequent calls to cancel simply return.
>
> The functions resume, cancel, and yield can be passed between and used by different goroutines, in effect dynamically changing which goroutine is "the coroutine." Although New creates a new goroutine, it also establishes an invariant that one goroutine is always blocked, either in resume, cancel, yield, or (right after New) waiting for the resume that will call f. This invariant holds until f returns, at which point the new goroutine is shut down. The net result is that coro.New creates new concurrency in the program without any new parallelism.
>
> If multiple goroutines call resume or cancel, those calls are serialized. Similarly, if multiple goroutines call yield, those calls are serialized.

```
func New[In, Out any](f func(in In, yield func(Out) In) Out) (
        resume func(In) (Out, bool), cancel func())
```

Efficiency

As I said at the start, while it's important to have a definition of coroutines that can be understood by reference to a pure Go implementation, I believe we should use an optimized runtime implementation. On my 2019 MacBook Pro, passing values back and forth using the channel-based `coro.New` in this post requires approximately 190ns per switch, or 380ns per value in `coro.Pull`. Remember that `coro.Pull` would not be the standard way to use an iterator: the standard way would be to invoke the iterator directly, which has no coroutine overhead at all. You only need `coro.Pull` when you want to process iterated values incrementally, not using a single for loop. Even so, we want to make `coro.Pull` as fast as we can.

First I tried having the compiler mark send-receive pairs and leave hints for the runtime to fuse them into a single operation. That would let the channel runtime bypass the scheduler and jump directly to the other coroutine. This implementation requires about 118ns per switch, or 236ns per pulled value (38% faster). That's better, but it's still not as fast as I would like. The full generality of channels is adding too much overhead.

Next I added a direct coroutine switch to the runtime, avoiding channels entirely. That cuts the coroutine switch to three atomic compare-and-swaps (one in the coroutine data structure, one for the scheduler status of the blocking coroutine, and one for the scheduler status of the resuming coroutine), which I believe is optimal given the safety invariants that must be maintained. That implementation takes 20ns per switch, or 40ns per pulled value. This is about 10X faster than the original channel implementation. Perhaps more importantly, 40ns per pulled value seems small enough in absolute terms not to be a bottleneck for code that needs `coro.Pull`.

* Asterisks mark hyperlinked text.