The basics of programming in Go

```
type ReliableWriter interface {
        WriteAt(…) error
        Complete(…) error
        Abort(…)
}

type ReliableWriterImpl struct {
    …
}

…
```

```
type ReliableWriter interface {
        WriteAt(…) error
        Complete(…) error
        Abort(…)
}

type ReliableWriterImpl struct {
    …
}

…
```

Avoid interfaces with only one implementation.

Just make `struct ReliableWriter`.

```
rw := NewReliableWriterImpl(ctx, unreliableWriter)
rw.MaxCacheSize = 64 * 1024 * 1024
rw.MaxChunkSize = 16 * 1024 * 1024
```

```
rw := NewReliableWriterImpl(ctx, unreliableWriter)
rw.MaxCacheSize = 64 * 1024 * 1024
rw.MaxChunkSize = 16 * 1024 * 1024
```

Make sure constructors return objects that are constructed and ready to use.

A typical approach is to have

```
rw := NewReliableWriter(ctx, ReliableWriterParams{
    …
})
```

```
rw := &ReliableWriterImpl{
    data:    *NewScatterGatherBuffer(),
    ...
```

```
rw := &ReliableWriterImpl{
    data:    *NewScatterGatherBuffer(),
    …
```

Typically, we have two kinds of structs:

1. Smaller ones that are ok to move or copy.
2. Structs that must not move. These
   - may have fields that other structs point to,
   - may have embedded locks,
   - may be too big to copy efficiently.

Do not mix the two kinds.

```
for !rw.data.IsEmpty() {
    rw.mutex.Lock()
    buf, err := rw.data.TakeBytes(rw.MaxChunkSize)
    rw.mutex.Unlock()
    …
}
```

```
for !rw.data.IsEmpty() {
    rw.mutex.Lock()
    buf, err := rw.data.TakeBytes(rw.MaxChunkSize)
    rw.mutex.Unlock()
    ...
}
```

Some accesses to `rw.data` are not protected by a mutex.

```
for !rw.data.IsEmpty() {
    rw.mutex.Lock()
    buf, err := rw.data.TakeBytes(rw.MaxChunkSize)
    rw.mutex.Unlock()
    …
}

% go build -v -race
```

Some accesses to `rw.data` are not protected by a mutex.

```
for !rw.data.IsEmpty() {
   rw.mutex.Lock()
   buf, err := rw.data.TakeBytes(rw.MaxChunkSize)
   rw.mutex.Unlock()
   …
}
```

Some accesses to `rw.data` are not protected by a mutex.

```
% go build -v -race

% ./awesomeProject
==================
WARNING: DATA RACE
Write at 0x00c000206cb8 by goroutine 18:
 awesomeProject/writers.(*ScatterGatherBuffer).TakeBytes()
     /Users/artem/dev/students/Google-Cloud-Storage-Client-Project/writers/ScatterGatherBuffer.go:58 +0x320
     …

Previous read at 0x00c000206cb8 by main goroutine:
 awesomeProject/writers.(*ReliableWriterImpl).WriteAt()
     /Users/artem/dev/students/Google-Cloud-Storage-Client-Project/writers/ReliableWriter.go:94 +0x32c
     …

Goroutine 18 (running) created at:
 awesomeProject/writers.(*ReliableWr  awesomeProject/writers.NewReliableWriterImpl()
     /Users/artem/dev/students/Google-Cloud-Storage-Client-Project/writers/ReliableWriter.go:36 +0x1ec
     …
==================
```

```
func (rw *ReliableWriter) WriteAt(…) error {

    select {
    case <-ctx.Done():
        return ctx.Err()
    default:
    }

    rw.mutex.Lock()
    rw.data.AddBytes(buf)
    rw.writtenBytes += uint64(len(buf))
    rw.mutex.Unlock()

    …
```

```
func (rw *ReliableWriter) WriteAt(…) error {

   select {
   case <-ctx.Done():
      return ctx.Err()
   default:
   }

   rw.mutex.Lock()
   rw.data.AddBytes(buf)
   rw.writtenBytes += uint64(len(buf))
   rw.mutex.Unlock()

   …
```

See https://pkg.go.dev/context.

A context should be regarded as a way to signal “the result of this computation is no longer needed”.

A typical use:

1. HTTP requests have a context that is passed to their handlers.
2. When a request is cancelled, or a connection to a client is broken, the context is cancelled.
3. The handler detects the cancelation, cancels all requests that it issued, and exits.

```
func (rw *ReliableWriter) WriteAt(…) error {

   select {
   case <-ctx.Done():
      return ctx.Err()
   default:
   }

   rw.mutex.Lock()
   rw.data.AddBytes(buf)
   rw.writtenBytes += uint64(len(buf))
   rw.mutex.Unlock()

   …
```

See https://pkg.go.dev/context.

A context should be regarded as a way to signal “the result of this computation is no longer needed”.

A typical use:

1. HTTP requests have a context that is passed to their handlers.
2. When a request is cancelled, or a connection to a client is broken, the context is cancelled.
3. The handler detects the cancelation, cancels all requests that it issued, and exits.

Note: there are more uses of contexts. For example, they may carry some implicit state around like the current tracing span, the current logger to use, etc. These are not good usages. First, this state is hidden and there is no way to discover it, and no way to verify that all implicit parameters are present, etc. Second, retrieving such parameters from a context turns out to be a costly operation.

```
func (rw *ReliableWriter) WriteAt(…) error {

   select {
   case <-ctx.Done():
      return ctx.Err()
   default:
   }

   rw.mutex.Lock()
   rw.data.AddBytes(buf)
   rw.writtenBytes += uint64(len(buf))
   rw.mutex.Unlock()

   …
```

See https://pkg.go.dev/context.

A context may be regarded as a way to signal “the result of this computation is no longer needed”.

A typical use:

1. HTTP requests have a context that is passed to their handlers.
2. When a request is cancelled, or a connection to a client is broken, the context is cancelled.
3. The handler detects the cancelation, cancels all requests that it issued, and exits.

```
func (rw *ReliableWriter) waitWriteSpace(ctx) error {
   select {
   case <-rw.writeSpace:
      return nil
   case <-ctx.Done():
      return ctx.Err()
   }
}
```

← This is a typical and correct use of a context. When a goroutine blocks, it also arranges to wake up when a context is cancelled.

```
func (rw *ReliableWriter) WriteAt(…) error {

    select {
    case <-ctx.Done():
        return ctx.Err()
    default:
    }

    rw.mutex.Lock()
    rw.data.AddBytes(buf)
    rw.writtenBytes += uint64(len(buf))
    rw.mutex.Unlock()

    …
```

This check, on the contrary, is just unneeded complexity.

Cancelation is asynchronous. It is racy by definition. It may happen a millisecond earlier or a millisecond later, so it makes no sense to check context cancelation in short sequences of code that do not sleep.

Just remove this check.

```
func (rw *ReliableWriter) launchWriting(…) {
   var bytesWritten int64 = 0
   go func() {
      for {
         select {
         case <-rw.writeEventsChan:
            rw.handleWriteEvents(bytesWritten, ctx)

         case <-ctx.Done():
            fmt.Println(”shutting down")
            return
         }
      }
   }()
}
```

```
func (rw *ReliableWriter) launchWriting(…) {
   var bytesWritten int64 = 0
   go func() {
      for {
         select {
         case <-rw.writeEventsChan:
            rw.handleWriteEvents(bytesWritten, ctx)

         case <-ctx.Done():
            fmt.Println(”shutting down")
            return
         }
      }
   }()
}
```

This goroutine has a complex lifecycle. It receives commands over `writeEventsChan` and executes them, and also tracks the context cancelation.

This is typical for a thread in a thread pool. Creating a thread is a costly operation: one needs to allocate a big stack, allocate multiple other data structures like `struct task`, switch contexts several times, etc. That is why it makes sense to create a thread and then have it run for a long time and process many commands.

A goroutine is designed to be very lightweight. Typically, there are no context switches, and a newly spawned goroutine even runs in the same OS thread.

That is why it is preferable to make goroutines that do one simple action and have a very simple lifecycle.

It is fine to keep creating goroutines that only do `rw.handleWriteEvents()`.

```
func (rw *ReliableWriter) launchWriting(…) {
   var bytesWritten int64 = 0
   go func() {
      for {
         select {
         case <-rw.writeEventsChan:
            rw.handleWriteEvents(bytesWritten, ctx)

         case <-ctx.Done():
            fmt.Println(”shutting down")
            return
         }
      }
   }()
}
```

The lifetime of the goroutine is not limited by the lifetime of ReliableWriter. It keeps running even after a call to `rw.Commit()` or `rw.Abort()`.

More generally, all resources must be accounted and must be properly released.

See https://pkg.go.dev/sync#WaitGroup

```
func (rw *ReliableWriter) launchWriting(…) {
   var bytesWritten int64 = 0
   go func() {
      for {
         select {
         case <-rw.writeEventsChan:
            rw.handleWriteEvents(bytesWritten, ctx)

         case <-ctx.Done():
            fmt.Println(”shutting down")
            return
         }
      }
   }()
}
```

The lifetime of the goroutine is not limited by the lifetime of ReliableWriter. It keeps running even after a call to `rw.Commit()` or `rw.Abort()`.

More generally, all resources must be accounted and must be properly released.

See https://pkg.go.dev/sync#WaitGroup

A typical use of a waitgroup is:

```
var wg sync.WaitGroup

wg.Add(1)
go func() {
   defer wg.Done()
   …
}()

…

wg.Wait()
```

```
func (rw *ReliableWriter) launchWriting(…) {
   var bytesWritten int64 = 0
   go func() {
      for {
         select {
         case <-rw.writeEventsChan:
            rw.handleWriteEvents(bytesWritten, ctx)

         case <-ctx.Done():
            fmt.Println(”shutting down")
            return
         }
      }
   }()
}
```

The lifetime of the goroutine is not limited by the lifetime of ReliableWriter. It keeps running even after a call to `rw.Commit()` or `rw.Abort()`.

More generally, all resources must be accounted and must be properly released.

See https://pkg.go.dev/sync#WaitGroup

Better yet, resources must be accounted and have upper bounds on their usage.

```
func (rw *ReliableWriter) launchWriting(…) {
   var bytesWritten int64 = 0
   go func() {
      for {
         select {
         case <-rw.writeEventsChan:
            rw.handleWriteEvents(bytesWritten, ctx)

         case <-ctx.Done():
            fmt.Println(”shutting down")
            return
         }
      }
   }()
}
```

The lifetime of the goroutine is not limited by thelife time of ReliableWriter. It keeps running even after a call to `rw.Commit()` or `rw.Abort()`.

More generally, all resources must be accounted and must be properly released.

See https://pkg.go.dev/sync#WaitGroup

Better yet, resources must be accounted and have upper bounds on their usage.

Exercise: implement struct Workgroup that combines

1. sync.WaitGroup to wait for the completion of member goroutines,
2. a semaphore to limit the number of goroutines in a workgroup.

Suggested interface:

1. NewWorkgroup(cfg WorkgroupConfig) *Workgroup,
2. func (wg *Workgroup) Go(ctx, f func(ctx) error)
3. func (wg *Workgroup) Wait()
4. func (wg *Workgroup) Error() error

See also: https://pkg.go.dev/golang.org/x/sync/errgroup

```
func (ulw *UnreliableLocalWriter) WriteAt(…) {
    …
    if rand.Intn(100) == 42 {
        return totalWritten, errors.New("Bad Luck")
    }
    …
}
```

```
func (ulw *UnreliableLocalWriter) WriteAt(…) {
    …
    if rand.Intn(100) == 42 {
        return totalWritten, errors.New("Bad Luck")
    }
    …
}
```

This is a plain string. There is no way for callers to verify whether this particular error happened, and there is no way to flag it as “retryable” or “definitely not retryable”.

```
func (ulw *UnreliableLocalWriter) WriteAt(…) {
    …
    if rand.Intn(100) == 42 {
        return totalWritten, errors.New("Bad Luck")
    }
    …
}
```

This is a plain string. There is no way for callers to verify whether this particular error happened, and there is no way to flag it as “retryable” or “definitely not retryable”.

A typical approach is to have a package that defines your application-specific errors:

```
type Error struct {
    Code  string
    Msg   string
    Cause error
}
```

Such error type can properly implement `Is()`, `As()` and `Unwrap()` to integrate with stdlib’s `errors` package. See https://pkg.go.dev/errors

Also see https://github.com/pkg/errors for a possible implementation.

```
func (ulw *UnreliableLocalWriter) WriteAt(…) {
    …
    if rand.Intn(100) == 42 {
        return totalWritten, errors.New("Bad Luck")
    }
    …
}
```

This is a plain string. There is no way for callers to verify whether this particular error happened, and there is no way to flag it as “retryable” or “definitely not retryable”.

A typical approach is to have a package that defines your application-specific errors:

```
var ErrBadLuck = errors.Error {
    Code: “myproj.bad_luck”,
    Msg:  “a simulated fault”,
}

func (ulw *UnreliableLocalWriter) WriteAt(…) {
    …
    if rand.Intn(100) == 42 {
        return totalWritten, ErrBadLuck
    }
    …
}
```

```
for attempt := 0; attempt < 3; attempt++ {
    written, err := rw.unreliableWriter.WriteAt(...)
    totalWritten += written

    if err == nil {
        return totalWritten, nil
    }

    if written < int64(len(remaining)) {
        remaining = remaining[written:]
    }

    if ctx.Err() != nil {
        return totalWritten, ctx.Err()
    }
}
```

```
for attempt := 0; attempt < 3; attempt++ {
    written, err := rw.unreliableWriter.WriteAt(…)
    totalWritten += written

    if err == nil {
        return totalWritten, nil
    }

    if written < int64(len(remaining)) {
        remaining = remaining[written:]
    }

    if ctx.Err() != nil {
        return totalWritten, ctx.Err()
    }
}
```

This code retries **all** errors, for example, it will retry “access denied” which clearly makes no sense.

```
for attempt := 0; attempt < 3; attempt++ {
    written, err := rw.unreliableWriter.WriteAt(…)
    totalWritten += written

    if err == nil {
        return totalWritten, nil
    }

    if written < int64(len(remaining)) {
        remaining = remaining[written:]
    }

    if ctx.Err() != nil {
        return totalWritten, ctx.Err()
    }
}
```

This code retries **all** errors, for example, it will retry “access denied” which clearly makes no sense.

```
for {
    res, err := doSomething(ctx, …)
    if !IsRetryableError(err) {
        return nil, err
    }

    … ok to retry …
}
```

```
for attempt := 0; attempt < 3; attempt++ {
    written, err := rw.unreliableWriter.WriteAt(…)
    totalWritten += written

    if err == nil {
        return totalWritten, nil
    }

    if written < int64(len(remaining)) {
        remaining = remaining[written:]
    }

    if ctx.Err() != nil {
        return totalWritten, ctx.Err()
    }
}
```

This code has no delays between attempts. Imagine you get “network unreachable” because your WiFi connection temporarily went down.

```
for attempt := 0; attempt < 3; attempt++ {
    written, err := rw.unreliableWriter.WriteAt(…)
    totalWritten += written

    if err == nil {
        return totalWritten, nil
    }

    if written < int64(len(remaining)) {
        remaining = remaining[written:]
    }

    if ctx.Err() != nil {
        return totalWritten, ctx.Err()
    }
}
```

This code has no delays between attempts. Imagine you get “network unreachable” because your WiFi connection temporarily went down.

1. Any network-related call fails with “network unreachable” immediately because the OS does not need to communicate over the network to report this error.
2. This code retries a failed request immediately.
3. Thus, the whole loops exits very quickly.
4. Yet, it takes several seconds to reconnect to a WiFi network.

Many other errors also make no sense to retry immediately. For example, if the remote service is restarting, then it is not going to become available immediately.

```
for attempt := 0; attempt < 3; attempt++ {
    written, err := rw.unreliableWriter.WriteAt(…)
    totalWritten += written

    if err == nil {
        return totalWritten, nil
    }

    if written < int64(len(remaining)) {
        remaining = remaining[written:]
    }

    if ctx.Err() != nil {
        return totalWritten, ctx.Err()
    }
}
```

This code has no delays between attempts. Imagine you get “network unreachable” because your WiFi connection temporarily went down.

There must be backoffs between retries.

```
for attempt := 0; attempt < 3; attempt++ {
    written, err := rw.unreliableWriter.WriteAt(…)
    totalWritten += written

    if err == nil {
        return totalWritten, nil
    }

    if written < int64(len(remaining)) {
        remaining = remaining[written:]
    }

    if ctx.Err() != nil {
        return totalWritten, ctx.Err()
    }
}
```

This code has no delays between attempts. Imagine you get “network unreachable” because your WiFi connection temporarily went down.

There must be backoffs between retries.

Moreover, backoffs must

1. start with short sleep intervals between attempts and then sleep longer,
2. include a random jitter to avoid a “thundering herd”.

Reminder: why do PAXOS and Raft add random delays during leader elections?

```
for attempt := 0; attempt < 3; attempt++ {
    written, err := rw.unreliableWriter.WriteAt(…)
    totalWritten += written

    if err == nil {
        return totalWritten, nil
    }

    if written < int64(len(remaining)) {
        remaining = remaining[written:]
    }

    if ctx.Err() != nil {
        return totalWritten, ctx.Err()
    }
}
```

Exercise: implement exponential backoffs:

```
b := backoff.New(backoff.Config{
    MinWait:   100*time.Millisecond,
    MaxWait :  10*time.Second,
    TotalWait: time.Hour,
})

for {
    res, err := doSomething(ctx, …)
    if !IsRetryableError(err) {
        return nil, err
    }

    if err = b.Wait(ctx); err != nil {
        return nil, err
    }
}
```

```
for attempt := 0; attempt < 3; attempt++ {
    written, err := rw.unreliableWriter.WriteAt(…)
    totalWritten += written

    if err == nil {
        return totalWritten, nil
    }

    if written < int64(len(remaining)) {
        remaining = remaining[written:]
    }

    if ctx.Err() != nil {
        return totalWritten, ctx.Err()
    }
}
```

This code places no upper bound on the duration of WriteAt(). If some packets are lost in the network, it may take a very long time to detect a connection failure.

```
for attempt := 0; attempt < 3; attempt++ {
    written, err := rw.unreliableWriter.WriteAt(…)
    totalWritten += written

    if err == nil {
        return totalWritten, nil
    }

    if written < int64(len(remaining)) {
        remaining = remaining[written:]
    }

    if ctx.Err() != nil {
        return totalWritten, ctx.Err()
    }
}
```

This code places no upper bound on the duration of WriteAt(). If some packets are lost in the network, it may take a very long time to detect a connection failure.

Often, we need to do the following:

```
for {
    opCtx, opCancel := context.WithTimeout(ctx, …)
    resp, err := httpClient.Do(opCtx, …)
    opCancel()

    if !IsRetryableError(err) {
        return nil, err
    }

    … backoff …
}
```

```
data := make([]byte, chunkSize)
for i := 0; i < len(data); i++ {
    data[i] = byte(i % 256)
}
```

```
data := make([]byte, chunkSize)
for i := 0; i < len(data); i++ {
    data[i] = byte(i % 256)
}
```

The test data is too regular. Suppose that `ReadAt()` that reads a file produces by this test is buggy and ignores the read offset. The following call will nevertheless read correct data:

```
in := make([]byte, 256)
n, err := r.ReadAt(in, 512)
```

Prefer randomly generated test data to avoid creating patterns that may conceal errors.