Can we panic yet?

Error handling in Go

Go Systems Conf SF ‘20

Raphael ‘kena’ Poss

Raphael / kena / knz

Netherlands

@knz

@kena42

https://dr-knz.net

@kena42

Background

Since **Go 1.0:** errors are objects

```
type error interface { Error() string }
```

“Idiomatic” error checking and propagation:

```
if err := myfunc(); err != nil {
   return err
}
```

This is **still advertised** in the Tour, guides, tutorials, etc.

Can we do better?

In this talk:
code performance and programmer productivity.

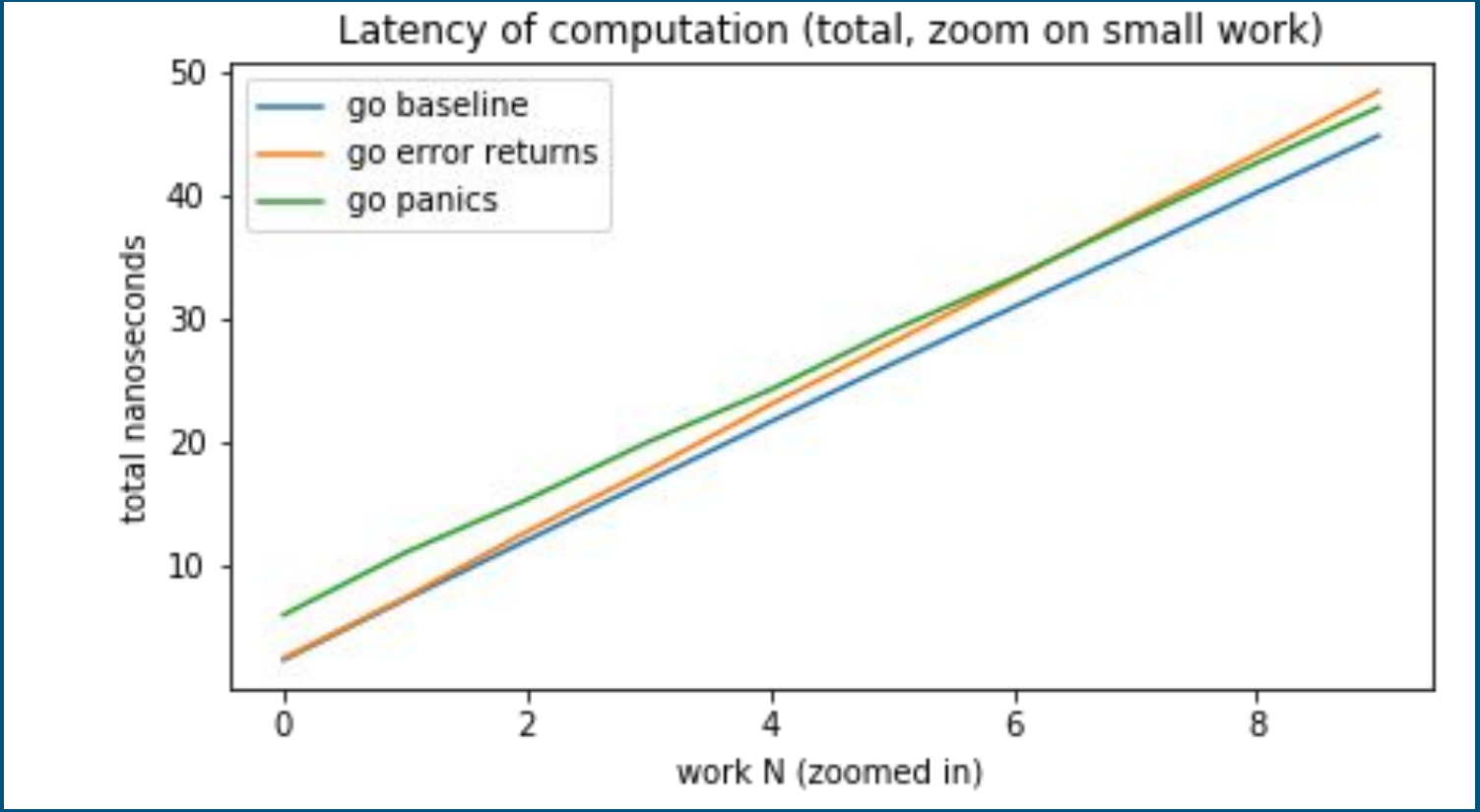
An experiment

Errors and exceptions

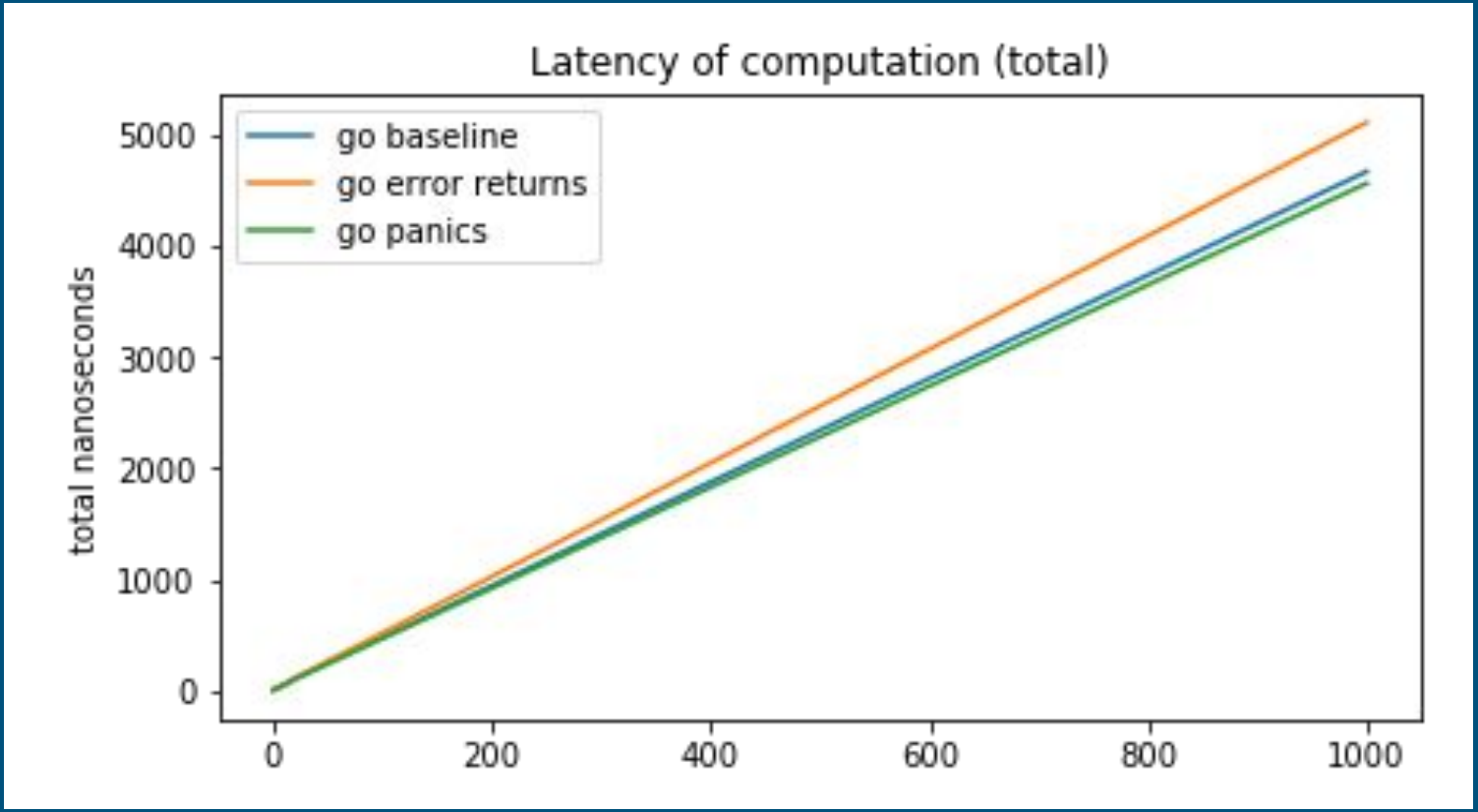
```
func unitOfWork(arg int) (int, error) {
   if arg == 0 { return -1, errObj }
   return arg
}

func doWork(work int) (r int, err error) {
   for i := 0; i < work; i++ {
      v, err := unitOfWork(work)
      if err != nil { return -1, err }
      r += v
   }
   return r, nil
}
```

```
func unitOfWork(arg int) int {
   if arg == 0 { panic(errObj) }
   return arg
}

func doWork(work int) (r int, err error) {
   defer func(){ err = recover().(error) }()

   for i := 0; i < work; i++ {
      r += unitOfWork(work)
   }
   return r, nil
}
```

Errors and exceptions - performance



Errors and exceptions - performance



Key points

- **Calling convention**: machine code inserted by compiler
  (for every function body and at every function call)
- Cannot be “optimized away” even by cleverest compiler
- So passing arguments and **returning values has a cost**
  - Moreso in Go which uses memory for arguments/returns instead of registers [cite]
- So do conditionals: all these “err != nil” are pure overhead

In comparison, the overhead of a well-placed defer is fixed and can be amortized

Other considerations: overall code size vs CPU I-caches; D-cache pressure

Go might get a
register-based calling convention

What happens next?

Ongoing project: new ABI

GitHub:

cmd/compile: switch to a register-based calling convention for Go functions #40724

Proposes register-based calling convention for x86-64, arm and possibly others.

Tentatively proposed for Go 1.16, likely available only later.

Support for all target architectures will also wait subsequent releases.

What we can expect

- Overhead of argument passing and return values comes back in line with equivalent C++ code
- Likely 1-4% performance improvement across all Go code

What will remain:

- Overhead of moving data around, albeit in registers
- Returning 1 error value on every call will remain more expensive than none

So exception-driven error handling will remain cheaper perf-wise!

Readability and correctness

Look again: errors and exceptions

```
func unitOfWork(arg int) (int, error) {
   if arg == 0 { return -1, errObj }
   return arg
}

func doWork(work int) (r int, err error) {
   for i := 0; i < work; i++ {
      v, err := unitOfWork(work)
      if err != nil { return -1, err }
      r += v
   }
   return r, nil
}
```

```
func unitOfWork(arg int) int {
   if arg == 0 { panic(errObj) }
   return arg
}

func doWork(work int) (r int, err error) {
   defer func(){ err = recover().(error) }()

   for i := 0; i < work; i++ {
      r += unitOfWork(work)
   }
   return r, nil
}
```

Look again: errors and exceptions

```
func myFunc() error {
   if err := stepA(); err != nil {
      return err
   }
   if err := stepB(); err != nil {
      return err
   }
   if err := stepC(); err != nil {
      return err
   }

   return nil
}
```

```
func myFunc() (err error) {
   defer func(){ err = recover().(error) }()

   stepA()
   stepB()
   stepC()

   return nil
}
```

In the real world - CockroachDB

- 14000+ occurrences of `err != nil` or `err == nil`
- in manually maintained code!

- We found bugs due to typing mistakes, multiple times:
  - "`if err == nil`" instead of "`if err != nil`" (or vice-versa)
  - "`return nil`" instead of "`return err`"
  - Now protected by linter, but the linter logic must also be developed and maintained

- Also, certain parts of CockroachDB are perf-sensitive too: SQL query planning and execution, low-level MVCC scans, etc.

In the real world - CockroachDB

What we did:

```
// Build is the top-level function to build the memo structure inside
// Builder.factory from the parsed SQL statement in Builder.stmt. See the
// comment above the Builder type declaration for details.
//
// If any subroutines panic with a non-runtime error as part of the build
// process, the panic is caught here and returned as an error.
func (b *Builder) Build() (err error) {
        defer func() {
                if r := recover(); r != nil {
                        // This code allows us to propagate errors without adding lots of checks
                        // for `if err != nil` throughout the construction code. This is only
                        // possible because the code does not update shared state and does not
                        // manipulate locks.
                        if ok, e := errorutil.ShouldCatch(r); ok {
                                err = e
                        } else {
                                panic(r)
                        }
                }
        }()
```

In the real world - CockroachDB

```
// ShouldCatch is used for catching errors thrown as panics. Its argument is the
// object returned by recover(); it succeeds if the object is an error. If the
// error is a runtime.Error, it is converted to an internal error (see
// errors.AssertionFailedf).
func ShouldCatch(obj interface{}) (ok bool, err error) {
        err, ok = obj.(error)
        if ok {
                if errors.HasInterface(err, (*runtime.Error)(nil)) {
                        // Convert runtime errors to internal errors, which display the stack and
                        // get reported to Sentry.
                        err = errors.HandleAsAssertionFailure(err)
                }
        }
        return ok, err
}
```

In the real world - CockroachDB

What we did:

- All the SQL planning logic under a single API call uses no error returns, instead panic used with **error objects**
- defer/recover (i.e. try/catch) at API boundary

Same for vectorized/distributed query execution

Planning to extend this pattern to multiple other components in the project

Engineers report X% extra work satisfaction from increased maintainability

Recommendations

Avoid redundant effort, DRY

- The visual occurrences of “if err != nil …” when the handling is trivial must become more concise
  - To make the code faster to read, teach and maintain
  - Beware of fallacious arguments—ergonomics in toy example change at 10000+ occurrences
- Programmers should not have to manually type in the common pattern
  - To reduce the amount of typing work
  - To reduce the likelihood of mistakes and correctness bugs

How? Either keep error returns with syntactic sugar, like in Rust, or ….

Exceptions are better (inside API boundaries)

- Exceptions (a.k.a. controlled panics) yield **simpler function signatures**
  - Makes the code easier to read, teach and maintain
- Exceptions yield **better performance**
  - When errors are uncommon
  - And error handling happens at the “top” of multiple levels of computational calls

This is reliably achieved by keeping errors explicit at API boundaries, with panic-driven error handling inside the API boundaries

FWIW Go stdlib’s `encoding/json` already does it, but no one talks about it

Patterns

To report an error:

```
panic(errors.New(...))
```

To add context to an exception:

```
defer func() {
  if err, ok := recover().(error); ok {
    panic(errors.Wrap(err, …))
  }
}()
```

Useful: Go runtime throws **string** for most faults and its errors implement interface **runtime.Error**

To translate an exception to an error:

```
func myFunc() (err error) {
   defer func() {
       if r := recover(); r != nil {
          if rerr, ok := r.(error); ok &&
!errors.HasInterface(rerr, (*runtime.Error)(nil)) {
              err = rerr
          } else { panic(r) } // rethrow
       }
   }()
   …
}
```

Defer callbacks are just functions!

```
defer annotateErr(annot)
```

```
// this can be put in a library
func annotateErr(annot string) {
  if err, ok := recover().(error); ok {
    // rethrow, annotated
    panic(errors.Wrap(err, annot))
  }
}
```

```
func myFunc() (err error) {
   defer catchErr(&err)
   …
}
```

```
// this can be put in a library
func catchErr(err *error) {
   r := recover()
   if rerr, ok := r.(error); ok &&
  !errors.HasInterface(rerr, (*runtime.Error)(nil)) {
     *err = rerr
   } else if r != nil { panic(r) }
}
```

Can We Panic Yet?

How you can help

- Evaluate how robust your project is to mistakes around “err != nil” and “return err”
- Properly review what programmer productivity really means
- Petition for an update to the Tour of Go: **promote exceptions as a viable alternative to error returns**

Thank you

Work sponsored by

More reading

- The Go low-level calling convention on x86-64 (updated)
- Errors vs. exceptions in Go and C++ in 2020—Why and how exceptions are still better for performance, even in Go

Tangentially related (shameless plug):

- Documentation for cockroachdb/errors