The Case for `interface{}`

FOSDEM'18

Sam Whited

XMPP: `sam@samwhited.com`

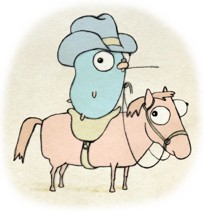
2017–02–03

`interface{}`

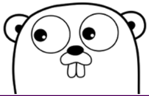
`interface{}`

FEAR! THRILLS! HORROR!

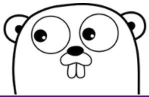
The following requirements keywords as used in this document are to be interpreted as described in RFC 2119: "MUST", "SHALL", "REQUIRED"; "MUST NOT", "SHALL NOT"; "SHOULD", "RECOMMENDED"; "SHOULD NOT", "NOT RECOMMENDED"; "MAY", "OPTIONAL".



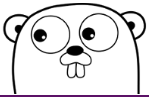
1. In Go, interfaces should describe behavior, not data



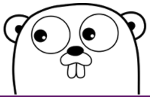
2. `interface{}` is easy to abuse (and thus, is abused; widely and often)



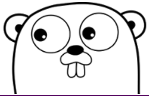
3. `interface{}` is code for “dynamic typing”



4. If you can describe your behavior with a more specific type, you should

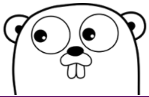


5. Heavy use of reflection leads to difficult to maintain code



5. Heavy use of reflection leads to difficult to maintain code

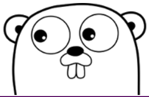
`encoding/{json,xml}`



5. Heavy use of reflection leads to difficult to maintain code

`encoding/{json,xml}`

Q.E.D.



encoding/xml

```go
package xml

// Marshal returns the XML encoding of v.
func Marshal(v interface{}) ([]byte, error) { /* … */ }
```

Empty interface says nothing

Rob Pike, Gopherfest 2015

```go
package xml

// Marshal returns the XML encoding of v.
func Marshal(v interface{}) ([]byte, error) { /* … */ }
```

```go
package xml

// Marshal returns the XML encoding of v.
func Marshal(v interface{}) ([]byte, error) { /* … */ }
```

1. 1 → <int>1</int>

```go
package xml

// Marshal returns the XML encoding of v.
func Marshal(v interface{}) ([]byte, error) { /* … */ }
```

1. 1 → <int>1</int>
2. “To sit in solemn silence” → <string>To sit in solemn silence</string>

```
package xml

// Marshal returns the XML encoding of v.
func Marshal(v interface{}) ([]byte, error) { /* … */ }
```

1. 1 → <int>1</int>
2. “To sit in solemn silence” → <string>To sit in solemn silence</string>
3. `xml.Marshaler` → `T.MarshalXML(e, start)`

```go
package xml

// Marshal returns the XML encoding of v.
func Marshal(v interface{}) ([]byte, error) { /* … */ }
```

1. 1 → <int>1</int>
2. "To sit in solemn silence" → <string>To sit in solemn silence</string>
3. `xml.Marshaler` → `T.MarshalXML(e, start)`
4. struct{ Name string } → ???

```go
package xml

// Marshal returns the XML encoding of v.
func Marshal(v interface{}) ([]byte, error) { /* … */ }
```

1. 1 → <int>1</int>
2. "To sit in solemn silence" → <string>To sit in solemn silence</string>
3. `xml.Marshaler` → `T.MarshalXML(e, start)`
4. struct{ Name string } → ??? (reflection!)

"When the *producer* of some data does not care about the type, but the *consumer* does, the library becomes difficult to maintain."

Rule №1

The producer of the `interface{}` must also be the consumer of the `interface{}`.

context

```
package context

// WithValue returns a copy of parent in which the value
// associated with key is val.
func WithValue(
        parent Context, key, val interface{},
) Context

type Context interface {
        // Value returns the value associated with this
        // context for key, or nil if no value is
        // associated with key.
        Value(key interface{}) interface{}
}
```

```
// Package context defines the Context type, which carries
// deadlines, cancelation signals, and other request-scoped
// values across API boundaries and between processes.
// Package context defines the Context type, which carries
// deadlines, cancelation signals, and other request-scoped
// values across API boundaries and between processes.
```

1. Session ID

1. Session ID
2. Request ID

1. Session ID
2. Request ID
3. Trace ID

```
func AddErrorLogger(
        ctx context.Context,
        logger log.Logger,
) context.Context {
        /* … */
}

func AddDebugLogger(
        ctx context.Context,
        logger log.Logger,
) context.Context {
        /* … */
}
```

```
func AddMetrics(
        ctx context.Context,
        metrics prometheus.Regi
) context.Context {
        /* … */
}

func AddDatabase(
        ctx context.Context,
        db *sql.DB,
) context.Context {
        /* … */
}
```

```
// LogKey is a context key that can be used for
// getting a log.Logger from a request.
// Don't do this.
type LogKey struct{}

// AddLogger adds a log.Logger to a request.
// No really, Don't do this.
func AddLogger(next Handler, l *log.Logger) HandlerFunc {
        return func(w ResponseWriter, r *Request) {
                ctx := r.Context()
                ctx = context.WithValue(
                        ctx, LogKey{}, logger)
                r = r.WithContext(ctx)
                h.ServeHTTP(w, r)
        }
}
```

Rule №2

`interface{}` should not cross package boundaries.

sasl

```go
// Mechanism represents an auth mechanism
// (eg. plain, scram, or oauth2).
type Mechanism struct {
        Next func(data interface{}) (cache interface{})
}

// Negotiator is a state machine that handles
// requests and responses in the auth flow.
type Negotiator struct{
        cache interface{}
}

// Step advances the state machine.
func (c *Negotiator) Step(challenge []byte) (resp []byte)
func Next(step int, data interface{}) interface{} {
        // State machine will always advance "step"
        switch step {
        case 0:
                // Do stuff
                // Return a "random" integer ID:
                return 4
        case 1:
                // We know it's an int!
                id := data.(int)
                // Do more stuff
                return nil
        }
        panic("the state machine is broken!")
}
```

Rule №3

You must always be able to assert the type of the `interface{}`.

$$\overbrace{\underbrace{\text{sam}}_{\text{me}}@\underbrace{\underbrace{\text{SamWhited}}_{\text{twitter \& freenode}}.\text{com}}_{\text{blog}}}^{\text{email \& xmpp}}$$