Advanced generics patterns

Axel Wagner

https://blog.merovius.de/

@Merovius@chaos.social

2024-07-09

r/golang generics

Posts Comments Media

Relevance All time

Show results from all of Reddit →

r/golang · 1mo ago

Generics on member funcs

13 votes · 19 comments

r/golang · 3mo ago

What is the current state of Go's generics?

41 votes · 42 comments

r/golang · 4y ago

The Next Step for Generics - The Go Blog

401 votes · 172 comments

r/golang · 1y ago

Do you use generics?

60 votes · 78 comments

r/golang · 3y ago

Never felt need for generics, did you?

99 votes · 190 comments

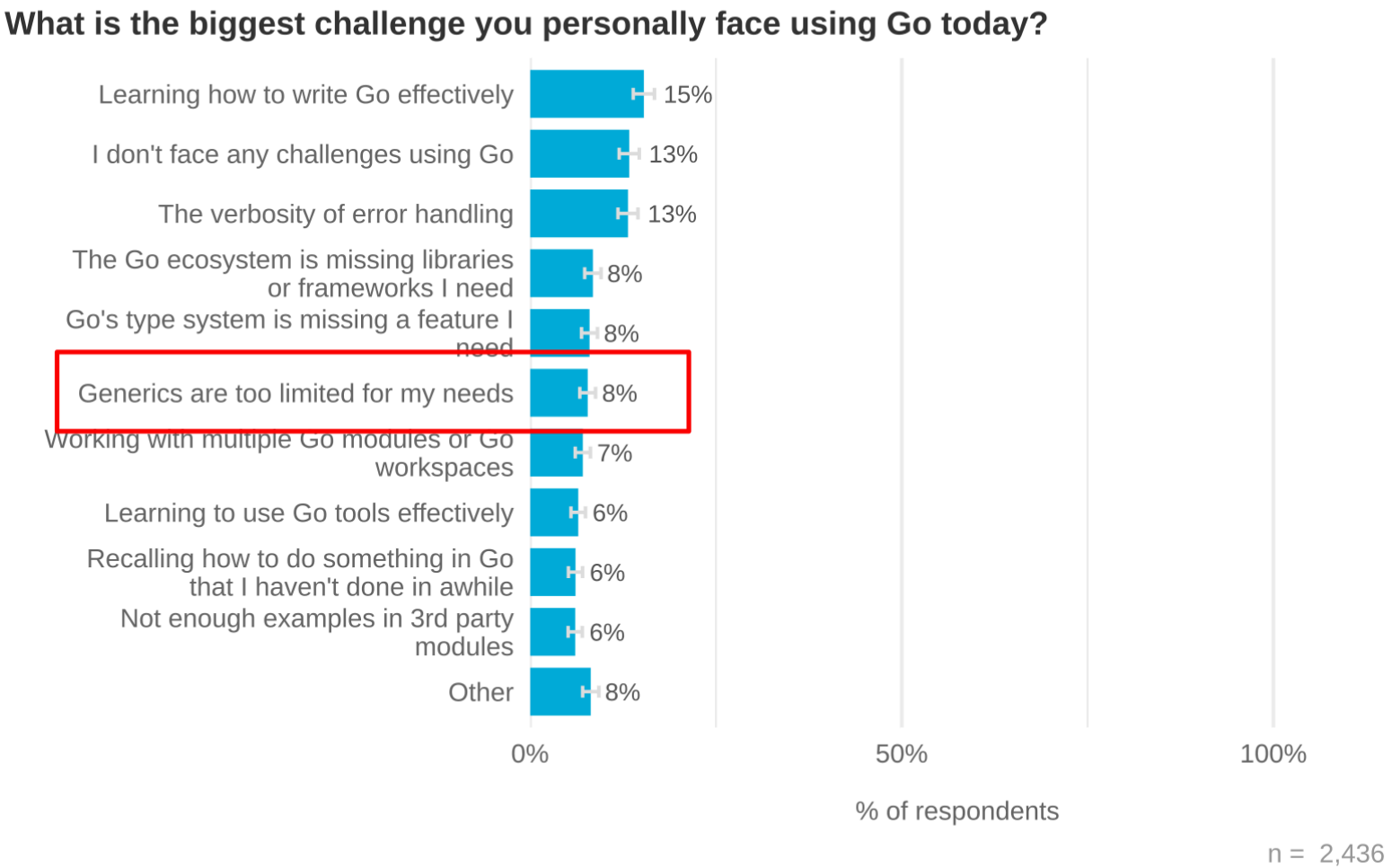
Introduction

Go generics were released in Go 1.18, over two years ago. We're using Go to write Dolt, the world's first version-controlled SQL database, and while we have hundreds of thousands of lines of Go code, we haven't used generics very much. There are a couple places we use them to make high-traffic parts of our code faster, but for the most part, we haven't really found a good reason for them, outside of the useful library methods in the slices and maps packages.

Unfortunately, some of these features start appearing in recent Go releases:

- Generics have been added in Go1.18. Many software engineers wanted generics in Go because they were thinking this will significantly improve their productivity in Go. Two years passed since Go1.18 release, but there is no sign in the increased productivity. The overall adoption of generics in Go remains low. Why? Because **generics aren't needed in most of practical Go code**. On the other hand, generics significantly increased the complexity of Go language itself. Try, for example, understanding all

Go Developer Survey 2024 H1 Results




“
WALLACE D. WATTLES
It is essential to have good tools, but it is also essential that the tools should be used in the right way.
GO

The Basics

```
type Slice[E any] []E
```

```
type Slice[E any] []E

func (s Slice[E]) Filter(keep func(E) bool) Slice[E]
```

```
type Slice[E any] []E

func (s Slice[E]) Filter(keep func(E) bool) Slice[E] {
  var out Slice[E]
  for i, v := range s {
    if keep(v) { out = append(out, v) }
  }
  return out
}
```

```
type Slice[E any] []E

func (s Slice[E]) Filter(keep func(E) bool) Slice[E] {
  var out Slice[E]
  for i, v := range s {
    if keep(v) { out = append(out, v) }
  }
  return out
}

func Map[A, B any](s Slice[A], f func(A) B) Slice[B]
```

```
type Slice[E any] []E

func (s Slice[E]) Filter(keep func(E) bool) Slice[E] {
  var out Slice[E]
  for _, v := range s {
    if keep(v) { out = append(out, v) }
  }
  return out
}

func Map[A, B any](s Slice[A], f func(A) B) Slice[B] {
  out := make(Slice[B], len(s))
  for i, v := range s {
    out[i] = f(v)
  }
  return out
}
```

```
func usage() {
  primes := Slice[int]{2, 3, 5, 7, 11, 13}
```

```
func usage() {
  primes := Slice[int]{2, 3, 5, 7, 11, 13}

  strings := Map(primes, strconv.Itoa)
```

```
func usage() {
  primes := Slice[int]{2, 3, 5, 7, 11, 13}

  strings := Map(primes, strconv.Itoa)

  fmt.Printf("%#v", strings)
  // Slice[string]{"2", "3", "5", "7", "11", "13"}
```

```go
func usage() {
  primes := Slice[int]{2, 3, 5, 7, 11, 13}

  strings := Map(primes, strconv.Itoa)

  fmt.Printf("%#v", strings)
  // Slice[string]{"2", "3", "5", "7", "11", "13"}

  // package reflect
  // func TypeFor[T any]() Type
  intType := reflect.TypeFor[int]()
}
```

A type parameter can be inferred if and only if it appears in an argument.

A type parameter can be inferred if and only if it appears in an argument.

Corollary: If you want a type parameter to be inferrable, make sure it appears as an argument.

Constraints

```
// StringifyAll converts the elements of a slice to strings and returns the
// resulting slice.
func StringifyAll[E any](s []E) []string
```

```
// StringifyAll converts the elements of a slice to strings and returns the
// resulting slice.
func StringifyAll[E any](s []E) []string {
  out := make([]string, len(s))
  for i, v := range s {
    out[i] = ???
  }
  return out
}
```

```
// StringifyAll converts the elements of a slice to strings and returns the
// resulting slice.
func StringifyAll[E ~string|~[]byte](s []E) []string
```

```
// StringifyAll converts the elements of a slice to strings and returns the
// resulting slice.
func StringifyAll[E ~string|~[]byte](s []E) []string {
  out := make([]string, len(s))
  for i, v := range s {
    out[i] = string(v)
  }
  return out
}
```

```go
// StringifyAll converts the elements of a slice to strings and returns the
// resulting slice.
func StringifyAll[E ~string|~[]byte](s []E) []string {
  out := make([]string, len(s))
  for i, v := range s {
    out[i] = string(v)
  }
  return out
}


func usage() {
  type Path string
  s := []Path{"/usr", "/bin", "/etc", "/home", "/usr"}
  fmt.Printf("%#v", StringifyAll(s))
  // []string{"/usr", "/bin", "/etc", "/home", "/usr"}
}
```

```
// StringifyAll converts the elements of a slice to strings and returns the
// resulting slice.
func StringifyAll[E Bytes](s []E) []string {
  out := make([]string, len(s))
  for i, v := range s {
    out[i] = string(v)
  }
  return out
}

type Bytes interface {
  ~string | ~[]byte
}
```

```
// StringifyAll converts the elements of a slice to strings and returns the
// resulting slice.
func StringifyAll[E fmt.Stringer](s []E) []string
```

```
// StringifyAll converts the elements of a slice to strings and returns the
// resulting slice.
func StringifyAll[E fmt.Stringer](s []E) []string {
  out := make([]string, len(s))
  for i, v := range s {
    out[i] = v.String()
  }
  return out
}
```

```go
// StringifyAll converts the elements of a slice to strings and returns the
// resulting slice.
func StringifyAll[E fmt.Stringer](s []E) []string {
  out := make([]string, len(s))
  for i, v := range s {
    out[i] = v.String()
  }
  return out
}


func usage() {
  durations := []time.Duration{time.Second, time.Minute, time.Hour}
  fmt.Printf("%#v", StringifyAll(durations))
  // []string{"1s", "1m0s", "1h0m0s"}
}
```

```
// StringifyAll converts the elements of a slice to strings and returns the
// resulting slice.
func StringifyAll[E any](s []E, stringify func(E) string) []string
```

```
// StringifyAll converts the elements of a slice to strings and returns the
// resulting slice.
func StringifyAll[E any](s []E, stringify func(E) string) []string {
  out := make([]string, len(s))
  for i, v := range s {
    out[i] = stringify(v)
  }
  return out
}
```

```
// StringifyAll converts the elements of a slice to strings and returns the
// resulting slice.
func StringifyAll[E any](s []E, stringify func(E) string) []string {
  out := make([]string, len(s))
  for i, v := range s {
    out[i] = stringify(v)
  }
  return out
}


func usage() {
  // time.Time.String has type func(time.Time) string
  strings := StringifyAll(times, time.Time.String)
  // strconv.Itoa has type func(int) string
  strings = StringifyAll(ints, strconv.Itoa)
}
```

```
package slices

func Compact[E comparable](s []E) []E
func CompactFunc[E any](s []E, eq func(E, E) bool) []E

func Compare[E cmp.Ordered](s1, s2 S) int
func CompareFunc[E1, E2 any](s1 []E1, s2 []E2, cmp func(E1, E2) int) int

func Sort[E cmp.Ordered](x []E)
func SortFunc[E any](x []E, cmp func(a, b E) int)

// etc.
```

```
func Sort[E cmp.Ordered](x []E) {
  SortFunc(x, cmp.Compare[E])
}

func SortFunc[E any](x []E, cmp func(a, b E) int) {
  // sort in terms of cmp
}
```

```
// Heap implements a Min-Heap using a slice.
type Heap[E cmp.Ordered] []E
```

```
// Heap implements a Min-Heap using a slice.
type Heap[E cmp.Ordered] []E

func (h *Heap[E]) Push(v E) {
  *h = append(*h, v)
  // […]
  if (*h)[i] < (*h)[j] {
    // […]
  }
}
```

```
// HeapFunc implements a Min-Heap using a slice and a custom comparison.
type HeapFunc[E any] struct {
  Elements []E
  Compare func(E, E) int
}
```

```
// HeapFunc implements a Min-Heap using a slice and a custom comparison.
type HeapFunc[E any] struct {
  Elements []E
  Compare func(E, E) int
}

func (h *HeapFunc[E]) Push(v E) {
  h.Elements = append(h.Elements, v)
  // […]
  if h.Compare(h.Elements[i], h.Elements[j]) < 0 {
    // […]
  }
}
```

Generic interfaces

```
type Comparer interface {
  Compare(Comparer) int
}
```

```
type Comparer interface {
  Compare(Comparer) int
}

// Does not implement Comparer: Argument has type time.Time, not Comparer
func (t Time) Compare(u Time) int
```

```
type Comparer[T any] interface {
  Compare(T) int
}
```

```
type Comparer[T any] interface {
  Compare(T) int
}

// implements Comparer[Time]
func (t Time) Compare(u Time) int
```

```
type Comparer[T any] interface {
  Compare(T) int
}

// implements Comparer[Time]
func (t Time) Compare(u Time) int

// E must have a method Compare(E) int
type HeapMethod[E Comparer[E]] []E
```

```
type Comparer[T any] interface {
  Compare(T) int
}

// implements Comparer[Time]
func (t Time) Compare(u Time) int

// E must have a method Compare(E) int
type HeapMethod[E Comparer[E]] []E

func (h *HeapMethod[E]) Push(v E) {
  *h = append(*h, v)
  // […]
  if (*h)[i].Compare((*h)[j]) < 0 {
    // […]
  }
}
```

```
func push[E any](s []E, cmp func(E, E) int, v E) []E {
  // […]
  if cmp(s[i], s[j]) < 0 {
    // […]
  }
}
```

```
func push[E any](s []E, cmp func(E, E) int, v E) []E {
  // […]
  if cmp(s[i], s[j]) < 0 {
    // […]
  }
}

func (h *Heap[E]) Push(v E) {
  *h = push(*h, cmp.Compare[E], v)
}
```

```
func push[E any](s []E, cmp func(E, E) int, v E) []E {
  // […]
  if cmp(s[i], s[j]) < 0 {
    // […]
  }
}

func (h *Heap[E]) Push(v E) {
  *h = push(*h, cmp.Compare[E], v)
}

func (h *HeapFunc[E]) Push(v E) {
  h.Elements = push(h.Elements, h.Compare, v)
}
```

```
func push[E any](s []E, cmp func(E, E) int, v E) []E {
  // […]
  if cmp(s[i], s[j]) < 0 {
    // […]
  }
}

func (h *Heap[E]) Push(v E) {
  *h = push(*h, cmp.Compare[E], v)
}

func (h *HeapFunc[E]) Push(v E) {
  h.Elements = push(h.Elements, h.Compare, v)
}

func (h *HeapMethod[E]) Push(v E) {
  *h = push(*h, E.Compare, v)
}
```

Pointer constraints

```
type Message struct {
  Price int // in cents
}

func (m *Message) UnmarshalJSON(b []byte) error {
  // { "price": 0.20 }
  var v struct {
    Price json.Number `json:"price"`
  }
  err := json.Unmarshal(b, &v)
  if err != nil {
    return err
  }
  m.Price, err = parsePrice(string(v.Price))
  return err
}
```

```
func Unmarshal[T json.Unmarshaler](b []byte) (T, error) {
  var v T
  err := v.UnmarshalJSON(b)
  return v, err
}
```

```
func Unmarshal[T json.Unmarshaler](b []byte) (T, error) {
  var v T
  err := v.UnmarshalJSON(b)
  return v, err
}


func usage() {
  input := []byte(`{"price": 13.37}`)
  // Message does not satisfy json.Unmarshaler
  //   (method UnmarshalJSON has pointer receiver)
  m, err := Unmarshal[Message](input)
  // …
}
```

```
func Unmarshal[T json.Unmarshaler](b []byte) (T, error) {
  var v T
  err := v.UnmarshalJSON(b)
  return v, err
}


func usage() {
  input := []byte(`{"price": 13.37}`)
  // panic: runtime error: invalid memory address or
  //     nil pointer dereference
  m, err := Unmarshal[*Message](input)
  // …
}
```

```
func Unmarshal[T any, PT json.Unmarshaler](b []byte) (T, error) {
  var v T
  err := v.UnmarshalJSON(b)
  return v, err
}
```

```
func Unmarshal[T any, PT json.Unmarshaler](b []byte) (T, error) {
  var v T
  err := v.UnmarshalJSON(b) // v.UnmarshalJSON undefined
  return v, err
}
```

```
func Unmarshal[T any, PT json.Unmarshaler](b []byte) (T, error) {
  var v T
  err := PT(&v).UnmarshalJSON(b) // cannot convert &v to type PT
  return v, err
}
```

```
func Unmarshal[T any, PT json.Unmarshaler](b []byte) (T, error) {
  var v T
  err := PT(&v).UnmarshalJSON(b) // cannot convert &v to type PT
  return v, err
}


type Unmarshaler[T any] interface{
  *T
  json.Unmarshaler
}
```

```
func Unmarshal[T any, PT Unmarshaler[T]](b []byte) (T, error) {
  var v T
  err := PT(&v).UnmarshalJSON(b)
  return v, err
}


type Unmarshaler[T any] interface{
  *T
  json.Unmarshaler
}
```

```
func Unmarshal[T any, PT Unmarshaler[T]](b []byte) (T, error) {
  var v T
  err := PT(&v).UnmarshalJSON(b)
  return v, err
}


type Unmarshaler[T any] interface{
  *T
  json.Unmarshaler
}


func usage() {
  input := []byte(`{"price": 13.37}`)
  m, err := Unmarshal[Message, *Message](input)
  // …
}
```

```
func Unmarshal[T any, PT Unmarshaler[T]](b []byte) (T, error) {
  var v T
  err := PT(&v).UnmarshalJSON(b)
  return v, err
}


type Unmarshaler[T any] interface{
  *T
  json.Unmarshaler
}


func usage() {
  input := []byte(`{"price": 13.37}`)
  m, err := Unmarshal[Message](input)
  // …
}
```

```go
func Unmarshal[T any, PT Unmarshaler[T]](b []byte, p *T) error {
  return PT(p).UnmarshalJSON(b)
}

type Unmarshaler[T any] interface{
  *T
  json.Unmarshaler
}

func usage() {
  input := []byte(`{"price": 13.37}`)
  var m Message
  err := Unmarshal(input, &m)
  // …
}
```

```
func Unmarshal[PT json.Unmarshaler](b []byte, p PT) error {
  return p.UnmarshalJSON(b)
}

func usage() {
  input := []byte(`{"price": 13.37}`)
  var m Message
  err := Unmarshal(input, &m)
  // …
}
```

```
func Unmarshal(b []byte, p json.Unmarshaler) error {
  return p.UnmarshalJSON(b)
}

func usage() {
  input := []byte(`{"price": 13.37}`)
  var m Message
  err := Unmarshal(input, &m)
  // …
}
```

Specialization

```
// UnmarshalText implements the encoding.TextUnmarshaler interface. The time
// must be in the RFC 3339 format.
func (t *Time) UnmarshalText(b []byte) error {
  var err error
  *t, err = Parse(RFC3339, string(b))
  return err
}

// Parse parses a formatted string and returns the time value it represents.
func Parse(layout, value string) (Time, error) {
  // parsing code
}
```

```
// UnmarshalText implements the encoding.TextUnmarshaler interface. The time
// must be in the RFC 3339 format.
func (t *Time) UnmarshalText(b []byte) error {
  var err error
  *t, err = Parse(RFC3339, string(b))
  return err
}

// Parse parses a formatted string and returns the time value it represents.
func Parse(layout, value string) (Time, error) {
  // parsing code
}


func parse[S string|[]byte](layout string, value S) (Time, error) {
  // parsing code
}
```

```
// UnmarshalText implements the encoding.TextUnmarshaler interface. The time
// must be in the RFC 3339 format.
func (t *Time) UnmarshalText(b []byte) error {
  var err error
  *t, err = parse(RFC3339, b)
  return err
}

// Parse parses a formatted string and returns the time value it represents.
func Parse(layout, value string) (Time, error) {
  return parse(layout, value)
}


func parse[S string|[]byte](layout string, value S) (Time, error) {
  // parsing code
}
```

```
// error: cannot use value (variable of type S constarined by string|[]byte)
//   as string value in argument to strings.CutPrefix
rest, ok := strings.CutPrefix(value, month)
if !ok {
    return fmt.Errorf("can not parse %q as month name", value)
}
```

```
func cutPrefix[S string|[]byte](s, prefix S) (after S, found bool) {
  for i := 0; i < len(prefix); i++ {
    if i >= len(s) || s[i] != prefix[i] {
      return s, false
    }
  }
  return s[len(prefix):], true
}
```

```
func cutPrefix[S string|[]byte](s, prefix S) (after S, found bool) {
  switch s := any(s).(type) {
  case string:
    s, found = strings.CutPrefix(s, prefix)
    return S(s), found
  case []byte:
    s, found = bytes.CutPrefix(s, prefix)
    return S(s), found
  default:
    panic("unreachable")
  }
}
```

Phantom types

```
type X[T any] string
```

```
func Parse[T any](r io.Reader) (T, error)
```

```
func Parse[T any](r io.Reader) (T, error)
```

```
type buffer struct { /* … */ }
```

```
func Parse[T any](r io.Reader) (T, error)
```

```
type buffer struct { /* … */ }

var buffers = sync.Pool{
  New: func() any { return new(buffer) },
}
```

```
func Parse[T any](r io.Reader) (T, error) {
  b := buffers.Get().(*buffer)
  b.Reset(r)
  defer buffers.Put(b)
  // use the buffer
}

type buffer struct { /* … */ }

var buffers = sync.Pool{
  New: func() any { return new(buffer) },
}
```

```
func Parse[T any](r io.Reader) (T, error) {
  b := buffers.Get().(*buffer[T]) // panics
  b.Reset(r)
  defer buffers.Put(b)
  // use the buffer
}

type buffer[T any] struct { /* … */ }

var buffers = sync.Pool{
  // Can't set New: No known type argument
}
```

```
type key[T any] struct{}
```

```go
type key[T any] struct{}


func usage() {
  var (
    kInt    any = key[int]{}
    kString any = key[string]{}
  )
  fmt.Println(kInt == kInt) // true
  fmt.Println(kString == kString) // false
}
```

```
type key[T any] struct{}
```

```
type key[T any] struct{}

var bufferPools sync.Map // maps key[T]{} -> *sync.Pool
```

```
type key[T any] struct{}

var bufferPools sync.Map // maps key[T]{} -> *sync.Pool

func poolOf[T any]() *sync.Pool {
  k := key[T]{}
```

```
type key[T any] struct{}

var bufferPools sync.Map // maps key[T]{} -> *sync.Pool

func poolOf[T any]() *sync.Pool {
  k := key[T]{}
  if p, ok := bufferPools.Load(k); ok {
    return p.(*sync.Pool)
  }
```

```
type key[T any] struct{}

var bufferPools sync.Map // maps key[T]{} -> *sync.Pool

func poolOf[T any]() *sync.Pool {
  k := key[T]{}
  if p, ok := bufferPools.Load(k); ok {
    return p.(*sync.Pool)
  }
  pi, _ := bufferPools.LoadOrStore(k, &sync.Pool{
    New: func() any { return new(T) },
  })
  return pi.(*sync.Pool)
}
```

```
func Parse[T any](r io.Reader) (T, error)
```

```
func Parse[T any](r io.Reader) (T, error) {
  pool := poolOf[T]()
```

```
func Parse[T any](r io.Reader) (T, error) {
  pool := poolOf[T]()
  b := pool.Get().(*buffer[T])
  b.Reset(r)
  defer pool.Put(b)
  // use the buffer
}
```

Overengineering

```
type Client struct { /* … */ }

func (c *Client) CallFoo(req *FooRequest) (*FooResponse, error)
func (c *Client) CallBar(req *BarRequest) (*BarResponse, error)
func (c *Client) CallBaz(req *BazRequest) (*BazResponse, error)
```

```
type Client struct { /* … */ }

func Call[Req, Resp any](c *Client, name string, r Req) (Resp, error)
```

```
type Client struct { /* … */ }

func Call[Req, Resp any](c *Client, name string, r Req) (Resp, error)

const (
  Foo = "Foo"
  Bar = "Bar"
  Baz = "Baz"
)
```

```
type Client struct { /* … */ }

func Call[Req, Resp any](c *Client, name string, r Req) (Resp, error)

const (
  Foo = "Foo"
  Bar = "Bar"
  Baz = "Baz"
)

func usage() {
  resp, err := rpc.Call[*rpc.FooRequest, *rpc.FooResponse](c, rpc.Foo, req)
  // …
```

```go
type Client struct { /* … */ }

func Call[Req, Resp any](c *Client, name string, r Req) (Resp, error)

const (
  Foo = "Foo"
  Bar = "Bar"
  Baz = "Baz"
)

func usage() {
  resp, err := rpc.Call[*rpc.FooRequest, *rpc.FooResponse](c, rpc.Foo, req)
  // …
  resp, err := rpc.Call[*rpc.FooRequest, *rpc.BarResponse](c, rpc.Baz, req)
}
```

```
type Endpoint[Req, Resp any] string
```

```
type Endpoint[Req, Resp any] string

const (
    Foo Endpoint[*FooRequest, *FooResponse] = "Foo"
    Bar Endpoint[*BarRequest, *BarResponse] = "Bar"
    Baz Endpoint[*BazRequest, *BazResponse] = "Baz"
)
```

```go
type Endpoint[Req, Resp any] string

const (
    Foo Endpoint[*FooRequest, *FooResponse] = "Foo"
    Bar Endpoint[*BarRequest, *BarResponse] = "Bar"
    Baz Endpoint[*BazRequest, *BazResponse] = "Baz"
)

func Call[Req, Resp any](c *Client, e Endpoint[Req, Resp], r Req) (Resp, error)
```

```
type Endpoint[Req, Resp any] string

const (
    Foo Endpoint[*FooRequest, *FooResponse] = "Foo"
    Bar Endpoint[*BarRequest, *BarResponse] = "Bar"
    Baz Endpoint[*BazRequest, *BazResponse] = "Baz"
)

func Call[Req, Resp any](c *Client, e Endpoint[Req, Resp], r Req) (Resp, error)


func usage() {
  r1, err := rpc.Call(c, rpc.Foo, req) // r1 is inferred to be *FooResponse
```

```go
type Endpoint[Req, Resp any] string

const (
    Foo Endpoint[*FooRequest, *FooResponse] = "Foo"
    Bar Endpoint[*BarRequest, *BarResponse] = "Bar"
    Baz Endpoint[*BazRequest, *BazResponse] = "Baz"
)

func Call[Req, Resp any](c *Client, e Endpoint[Req, Resp], r Req) (Resp, error)


func usage() {
  r1, err := rpc.Call(c, rpc.Foo, req) // r1 is inferred to be *FooResponse

  // type *rpc.FooRequest of req does not match inferred type *rpc.BazRequest
  r2, err := rpc.Call(c, rpc.Baz, req)
```

```go
type Endpoint[Req, Resp any] string

const (
    Foo Endpoint[*FooRequest, *FooResponse] = "Foo"
    Bar Endpoint[*BarRequest, *BarResponse] = "Bar"
    Baz Endpoint[*BazRequest, *BazResponse] = "Baz"
)

func Call[Req, Resp any](c *Client, e Endpoint[Req, Resp], r Req) (Resp, error)


func usage() {
  r1, err := rpc.Call(c, rpc.Foo, req) // r1 is inferred to be *FooResponse

  // type *rpc.FooRequest of req does not match inferred type *rpc.BazRequest
  r2, err := rpc.Call(c, rpc.Baz, req)

  r3, err := rpc.Call[int, string](c, "b0rk", 42) // compiles, but broken
}
```

```
type Endpoint[Req, Resp any] struct{ name string }
```

```
type Endpoint[Req, Resp any] struct{ name string }

var (
    Foo = Endpoint[*FooRequest, *FooResponse]{"Foo"}
    Bar = Endpoint[*BarRequest, *BarResponse]{"Bar"}
    Baz = Endpoint[*BazRequest, *BazResponse]{"Baz"}
)
```

```
type Endpoint[Req, Resp any] struct{ name string }

var (
    Foo = Endpoint[*FooRequest, *FooResponse]{"Foo"}
    Bar = Endpoint[*BarRequest, *BarResponse]{"Bar"}
    Baz = Endpoint[*BazRequest, *BazResponse]{"Baz"}
)

func Call[Req, Resp any](c *Client, e Endpoint[Req, Resp], r Req) (Resp, error)
```

```go
type Endpoint[Req, Resp any] struct{ name string }

var (
    Foo = Endpoint[*FooRequest, *FooResponse]{"Foo"}
    Bar = Endpoint[*BarRequest, *BarResponse]{"Bar"}
    Baz = Endpoint[*BazRequest, *BazResponse]{"Baz"}
)

func Call[Req, Resp any](c *Client, e Endpoint[Req, Resp], r Req) (Resp, error)


func usage() {
  // cannot use "b0rk" (untyped string constant) as Endpoint[int, string] value
  r1, err := rpc.Call[int, string](c, "b0rk", 42)
```

```
type Endpoint[Req, Resp any] struct{ name string }

var (
    Foo = Endpoint[*FooRequest, *FooResponse]{"Foo"}
    Bar = Endpoint[*BarRequest, *BarResponse]{"Bar"}
    Baz = Endpoint[*BazRequest, *BazResponse]{"Baz"}
)

func Call[Req, Resp any](c *Client, e Endpoint[Req, Resp], r Req) (Resp, error)


func usage() {
  // cannot use "b0rk" (untyped string constant) as Endpoint[int, string] value
  r1, err := rpc.Call[int, string](c, "b0rk", 42)

  e := rpc.Endpoint[int, string](rpc.Foo)
  r2, err := rpc.Call(c, e, 42)
}
```

```
type Endpoint[Req, Resp any] struct{ _ [0]Req; _ [0]Resp; name string }
```

```
type Endpoint[Req, Resp any] struct{ _ [0]Req; _ [0]Resp; name string }

var (
    Foo = Endpoint[*FooRequest, *FooResponse]{name: "Foo"}
    Bar = Endpoint[*BarRequest, *BarResponse]{name: "Bar"}
    Baz = Endpoint[*BazRequest, *BazResponse]{name: "Baz"}
)
```

```
type Endpoint[Req, Resp any] struct{ _ [0]Req; _ [0]Resp; name string }

var (
    Foo = Endpoint[*FooRequest, *FooResponse]{name: "Foo"}
    Bar = Endpoint[*BarRequest, *BarResponse]{name: "Bar"}
    Baz = Endpoint[*BazRequest, *BazResponse]{name: "Baz"}
)

func Call[Req, Resp any](c *Client, e Endpoint[Req, Resp], r Req) (Resp, error)
```

```go
type Endpoint[Req, Resp any] struct{ _ [0]Req; _ [0]Resp; name string }

var (
    Foo = Endpoint[*FooRequest, *FooResponse]{name: "Foo"}
    Bar = Endpoint[*BarRequest, *BarResponse]{name: "Bar"}
    Baz = Endpoint[*BazRequest, *BazResponse]{name: "Baz"}
)

func Call[Req, Resp any](c *Client, e Endpoint[Req, Resp], r Req) (Resp, error)


func usage() {
  // cannot convert rpc.Bar to rpc.Endpoint[int, string]
  e := rpc.Endpoint[int, string](rpc.Bar)
  resp, err := rpc.Call(c, e, req)
}
```

Go forth and experiment