Go の interface と Generics の内部構造と進化

Go Conference 2025

@turbofish_

自己紹介

富永 良子

@turbofish

- バックエンドエンジニア
- 仕事では株式会社 ZOZO でマーケティング配信基盤システムを開発しています
- 最近は Go, Google Cloud
- 旅行好きな人におすすめのクレカ教えてください

本テーマについて話そうと思ったきっかけ

複数の型を扱える汎用的な変数に any 型（interface{}）を使用する実装を見かけた。

```
func DoSomething(x any) {
    if x == nil {
        return
    }
    ...
}
```

any 型として扱われた引数が、具象型がnilのポインタ（またはスライス、マップ、チャネル）だった場合には、引数 x == nil は false と判定される（Typed-nil という現象（後述））。そのため、このようなユースケースでは Generics を使用した方が安全。

※ Typed-nil の サンプルコード（Go Playground）

このセッションについて

目的

Goの **型 / interface / Generics の内部構造** を理解し、正しく使い分けること。

注意点

※ ある程度 Go を書いたことがある人を対象としています。インターフェースなどの基礎的な用語の解説や使用方法ついては説明しません。
※ 本セッションで話す内容とセッション後に公開する資料は、2025年9月中旬時点のGo 1.25.1 のソースコードに基づいて作成されています。
※ できるだけ内部実装のソースコードを用いて説明をします。Go アセンブリを追う必要がある挙動についてはコードコメントなどで挙動を説明している箇所を掲載します。

アジェンダ

- 内部実装
  - 型
  - interface{} / any 型
  - interface
  - Generics
- interface と Generics の進化
- まとめ

型システムに関係する主要な Go の内部パッケージ

src/runtime

Go プログラムの実行時の振る舞いを支えるパッケージ。スケジューラやガーベジコレクタ、インターフェースの実体表現などを提供する。

src/cmd/compile

Go コンパイラ本体のパッケージ群。ソースコードを解析・最適化し、中間表現を経て実行可能なバイナリを生成する。

src/internal/abi

コンパイラとランタイムが共有する「Application Binary Interface」を定義するパッケージ。型情報やインターフェース表現など低レベルのデータ構造を記述する。

内部実装

1-1. 型の内部実装

src/runtime/type.go

```
type Type struct {
        Size_       uintptr
        PtrBytes    uintptr
        Hash        uint32
        TFlag       TFlag
        Align_      uint8
        FieldAlign_ uint8
        Kind_       Kind    // string, int, ...
        Equal       func(unsafe.Pointer, unsafe.Pointer) bool
        GCData      *byte   // GCがスキャンすべき場所を示すポインタ
        Str         NameOff
        PtrToThis   TypeOff
}
```

1-2. 派生型の内部実装

配列、スライス、マップ、チャネル、関数、構造体などの派生型は、Type 構造体を埋め込んだ構造体で表現される。

src/internal/abi/type.go

```
type SliceType struct {
        Type
        Elem *Type // 要素のスライス
}

type StructType struct {
        Type
        PkgPath Name
        Fields  []StructField
}
```

2. interface{} / any 型の内部実装

src/runtime/type.go

```
type _type = abi.Type
```

src/runtime/runtime2.go

```
type eface struct {
    _type *_type         // 型のメタ情報
    data  unsafe.Pointer // 値のポインタ
}
```

eface 構造体は、_type と data の両方が nil になる時だけ、nil とイコールであると判断される。data が nil でも、_type が有意であれば nil ではない（Typed-nil）。

3-1. interface の内部実装

interface 値は、内部的にはインターフェースが持つメソッドの情報（itab）と値の2つの要素を持つ構造体として実装されている。

src/runtime/runtime2.go

```
type itab = abi.ITab   // 後述
...
type iface struct {
    tab  *itab          // interface テーブル（後述）
    data unsafe.Pointer
}
```

eface と同様、data が nil でも、itab に含まれる型情報が有意であれば nil ではない（Typed-nil）。

（参考）interface の Typed-nil の落とし穴

例：error interface を実装した定義型を返す関数の挙動

```
type MyError struct{}
func (e MyError) Error string { return "error" }

func main() {
        fmt.Println(returnsNilError() == nil) // false
}

func returnsNilError(bad bool) error {
        var p *MyError = nil
        return p
}
```

Go Playground

出処：Go Documentation Frequently Asked Questions (FAQ): Why is my nil error value not equal to nil?

3-2. interface の内部実装（itab）

`itab` はメソッドポインタの配列になっており、実体型と interface のメソッドセットを対応付けする。メソッド呼び出しは `itab` 経由で実行時に動的ディスパッチ（実行時に多態的な操作（メソッドまたは関数）のどの実装を呼び出すかを選択するプロセス）を行う。

src/internal/abi/type.go

```
type ITab struct {
        Inter *InterfaceType // 実装しているインターフェース型
        Type  *Type          // 基底となる具象型
        Hash  uint32         // Type.Hash のコピー。型スイッチに使用される
        Fun   [1]uintptr // 実際には可変長サイズ。fun[0]==0 の場合、Type が Inter を実装していないことを意味する
}

type InterfaceType struct {
        Type
        PkgPath Name      // import path
        Methods []Imethod // sorted by hash
}
```

（参考）空 interface と具象 interface のコンパイル時型情報

src/internal/abi/type.go

```
// EmptyInterface describes the layout of a "interface{}" or a "any."
// These are represented differently than non-empty interface, as the first
// word always points to an abi.Type.
type EmptyInterface struct {
        Type *Type
        Data unsafe.Pointer
}

// NonEmptyInterface describes the layout of an interface that contains any methods.
type NonEmptyInterface struct {
        ITab *ITab
        Data unsafe.Pointer
}
```

（小まとめ）iface 構造体と eface 構造体

src/runtime/runtime2.go

```
// interface{} / any 型
type eface struct {
    _type *_type          // 型のメタ情報
    data  unsafe.Pointer // 値のポインタ
}

// interface 型
type iface struct {
    tab  *itab          // interface テーブル
    data unsafe.Pointer
}
```

（参考）Typed-nil を引き起こす変換処理

コンパイラから呼ばれる、型 t の値を interface に詰めるときの変換処理。
godocコメント要約：convT関数は、vが指す型tの値を取り、それをインターフェース値の2番目の要素（data）として使用できるポインタに変換する。この関数は v が nil でも成功する。

src/runtime/iface.go

```
func convT(t *_type, v unsafe.Pointer) unsafe.Pointer {
        if raceenabled {
                raceReadObjectPC(t, v, sys.GetCallerPC(), abi.FuncPCABIInternal(convT))
        }
   ...
        x := mallocgc(t.Size_, t, true)
        typedmemmove(t, x, v)
        return x
}
```

4-1. Generics の内部実装（方針）

Go の Generics は、**辞書 と Gcshape Stenciling** によって実装されている。

Generics のコンパイラ実装は、主に具体的な型を持つ引数で実行されるジェネリック関数およびメソッドのインスタンス化を行う。
さらに、辞書を使って純粋なモノモーフィゼーションを避けて1つの関数インスタンスを複数の型で動かせるようにしている（コードサイズの節約などのメリットのため）。

参考：Go 1.18 Implementation of Generics via Dictionaries and Gcshape Stenciling

明日16時スタートのセッション**「なぜGoのジェネリクスはこの形なのか？ - Featherweight Goが明かす設計の核心」**で詳しい話が聞けそう！

4-2. Generics の内部実装（型制約チェック）

コンパイラの Instantiate 関数でジェネリック型や関数に具体的な型引数を適用してインスタンス化する。インスタンス化に失敗した場合には panic を起こす。

src/cmd/compile/internal/types2/instantiate.go

```
// orig（オリジナルの型）は、型エイリアス、定義型、関数型のいずれかである必要がある
func Instantiate(ctxt *Context, orig Type, targs []Type, validate bool) (Type, error) {
    ...
        if validate {
                tparams := orig_.TypeParams().list()
     ...
                if i, err := (*Checker)(nil).verify(nopos, tparams, targs, ctxt); err != nil {
                        return nil, &ArgumentError{i, err}
                }
        }

        inst := (*Checker)(nil).instance(nopos, orig_, targs, nil, ctxt)
        return inst, nil
}
```

4-3. Generics の内部実装（辞書）

src/cmd/compile/internal/noder/reader.go

```
type readerDict struct {
    shaped                  bool
    baseSym                 *types.Sym
    shapedObj               *ir.Name
    targs                   []*types.Type // 型パラメータに対応する具体型のリスト
    implicits               int
    derived                 []derivedInto
    derivedTypes            []*types.Type
    typeParamMethodExprs []readerMethodExprInfo
    subdicts                []objInfo
    rtypes                  []typeInfo
    itabs                   []itabInfo
}
```

interface と Generics の進化

Generics 導入前の型の抽象化

`interface{} + 型アサーション` を使用するのが一般的だった。
interface{} に格納された値の具体的な型情報はランタイムで解決されるため、型安全性の担保はコンパイル時ではなく実装者の責任に委ねられていた。

```
type MyType struct {}
type AnotherType struct {}

a := MyType{}
// 型アサーションで確認（2つ目の返り値を取る場合は panic しない）
a.(AnotherType) // panic
```

Generics の登場（Go1.18）

インターフェースを用いた型制約により、コンパイル時に型チェックが実行されるため、安全に型を抽象化できるようになった。

例：Generics を使用した関数と型定義の実装例

```
func Min[T constraints.Ordered](a, b T) T {
  ...
}

x := Min(3, 5)      // T=int
y := Min(2.1, 4.2) // T=float64

// 型定義の例
type Stack[T any] struct {
    items []T
}
```

Generics の進化

- **Go 1.18（2022年）**: 型パラメータと constraints 導入、any 登場
  - interface に型を並べて型集合として定義できるようになった
  - インターフェースはメソッドの集合から型の集合、つまりそれらのメソッドを実装する型を定義していると言う考え方に
- **Go 1.19–1.21**: 標準ライブラリの Generics 対応拡大
- **Go 1.21–1.22**: 制約の表現力強化 (`~`, `comparable` など)、型推論の改善
- **Go 1.23以降**: コンパイル効率改善、エラーメッセージ強化、ジェネリック型エイリアス完全サポート
- **Go 1.25（2025年）**: コア型の削除
  - 将来的な言語仕様の柔軟性向上の可能性

Generics の仕様について議論されていることの例

Go の Generics は、他言語のものと比較するとシンプルだが、ユースケースが少ないとの声も。（Go 開発者アンケート 2024 で回答者の 8%）

過去に拒否された仕様案の例

- イシュー #21659：関数のオーバーロード
- イシュー #49085：メソッド内で型パラメータを定義する

議論中の仕様案の例

- イシュー #48522：型集合に共通するフィールドを参照可能にする仕様変更
  - 動的ディスパッチでは対応できないパフォーマンス要件を持つシステムにも対応できるようになる
  - 実現可能性が高い

Generics 利用の一般的なガイドライン

Generics を使うべき状況

- 言語定義のコンテナ型（スライス、マップ、チャンネル）を使用する操作
- 汎用データ構造（連結リストなど言語に組み込まれていないもの）
- 異なる型で共通のメソッドを実装する必要があり、異なる型の実装がすべて同じように見える場合（例：slices パッケージでの要素のソートなど）

Generics を使うべきではない状況

- ある型の値のメソッドを呼び出すだけで済む場合は interface 型を使用する
- メソッドの実装が型ごとに異なる場合は、interface 型を使用する
- メソッドを持たない型で操作をサポートする必要がある場合（例：encoding/json パッケージ)

出処：When To Use Generics (The Go Blog)

まとめ

- Goの型システムは、interface{} / any 型はランタイム構造、interface はメソッドテーブル（itab）を介した動的ディスパッチ、Generics はコンパイル時展開（モノモーフィゼーション）+ ランタイムでの辞書渡し で実装されている
- interface{} / any 型は一種のジェネリックプログラミングを可能にするが、ランタイムエラーを完全に回避するためには実装者が注意する必要がある
- Generics は型安全性をコンパイル時に担保できるので、再利用性の高いデータ構造やアルゴリズムを実装する場合はGenericsを使用するのが良い
    - 処理が異なる場合は普通にその型を使った関数もしくはメソッドにしよう

ご清聴ありがとうございました 🙏

本登壇のスライドと参考資料はセッション後に公開予定です。