Go のための Go

motemen <https://motemen.github.io/>

目次

はじめに

なぜ **Go for Go** なのか

WIP

- Goの言語仕様そのままだと表現力は（設計により）貧弱
- 一方でそれを補完するオフィシャルなエコシステムが充実している、例: go generate
- また、標準パッケージのAPIも。多くのツールが書かれ、提供されている 例: `goimports`

この本のねらい

WIP

- ドキュメントとソースコードの中間
- API同士の関連について書く
- すべてを網羅するのではなく、書きはじめるために必要な情報を提供する

対象読者

WIP

Chapter 1. 構文解析

そのままではただの文字列であるだけのGoのソースコードをプログラムにとって意味のある操作可能な対象とするには、まずソースコードを構文解析して抽象構文木に変換する必要があります。いったん抽象構文木を手元に得てしまえば、任意のソースコードをプログラムから扱うのはとても簡単です。ここでは、

- Goのソースコードの抽象構文木がどのようにして得られるのか、
- 構文要素がどのように表現されているのか

といったことについて見ていきます。

1.1. 式の構文解析

Goのソースコードの構文解析を行うには、標準パッケージの `go/parser` を使用します。 まずはGoの式（expression）を解析するところからはじめましょう。

リスト *1. parseexpr.go*

```
package main

import (
    "fmt"

    "go/parser"
)

func main() {
    expr, _ := parser.ParseExpr("a * -1")
    fmt.Printf("%#v", expr)
}
```

簡単のため、サンプルコードではエラーを無視することがあります。

`go/parser.ParserExpr` はGoの式である文字列を構文解析し、式を表現する抽象構文木である `ast.Expr` を返します。

godoc: go/parser.ParseExpr

```
func ParseExpr(x string) (ast.Expr, error)
```

実行すると以下のように、式 `a * -1` に対応する抽象構文木が `*ast.BinaryExpr` として得られたことが分かります。

```
&ast.BinaryExpr{X:(*ast.Ident)(0xc82000eb60), OpPos:3, Op:14,
Y:(*ast.UnaryExpr)(0xc82000eba0)}
```

二項演算子 * の左の項である a が X（*ast.Ident）として、右の項である -1 が Y（*ast.UnaryExpr）として表現されていそうだ、ということが見て取れます。

1.1.1. ast.Print

%#v による表示でも大まかには構文木のノードの様子を知ることができますが、定数値の意味やさらに深いノードの情報には欠けています。構文木をさらに深く見ていくには、ast.Print 関数が便利です:

リスト 2. *parseexpr-print.go*

```
package main

import (
    "go/ast"
    "go/parser"
)

func main() {
    expr, _ := parser.ParseExpr("a * -1")
    ast.Print(nil, expr)
}
```

```
     0  *ast.BinaryExpr {
     1  .  X: *ast.Ident {
     2  .  .  NamePos: 1
     3  .  .  Name: "a"
     4  .  .  Obj: *ast.Object {
     5  .  .  .  Kind: bad
     6  .  .  .  Name: ""
     7  .  .  }
     8  .  }
     9  .  OpPos: 3
    10  .  Op: *
    11  .  Y: *ast.UnaryExpr {
    12  .  .  OpPos: 5
    13  .  .  Op: -
    14  .  .  X: *ast.BasicLit {
    15  .  .  .  ValuePos: 6
    16  .  .  .  Kind: INT
    17  .  .  .  Value: "1"
    18  .  .  }
    19  .  }
    20  }
```

X.Name が "a" であることや Op が "*" であることなど、先ほどの式 a * -1 を表す抽象構文木の構造がより詳細に理解できます。

ast.Print は抽象構文木を人間に読みやすい形で標準出力に印字します。便利な関数ですがあくまで開発中やデバッグに便利な関数であって、実際にコードを書いて何かを達成するために直接これを使うことは

ないでしょう。

godoc: go/ast.Print

```
func Print(fset *token.FileSet, x interface{}) error
```

第一引数 `fset` に関しては、ソースコード中の位置 で触れます。ここでは `nil` を渡すので十分です。

1.1.2. 構文ノードのインターフェース

`ast.ParseExpr` の返り値となっている `ast.Expr` はインターフェース型であり、先ほどの例で得られたのは具体的には `*ast.BinaryExpr` 構造体でした。これは二項演算に対応する構文ノードです。

godoc: go/ast.BinaryExpr

```
type BinaryExpr struct {
    X     Expr        // left operand
    OpPos token.Pos   // position of Op
    Op    token.Token // operator
    Y     Expr        // right operand
}
```

二項演算の左右の式である `X` や `Y` も `ast.Expr` として定義されていることがわかります。先ほどの例では `*ast.Ident` や `*ast.UnaryExpr` がその具体的な値となっていました。

これらの構造体を含め、すべてのGoの式に対応する構文ノードは `ast.Expr` インターフェースを実装しています。

godoc: go/ast.Expr

```
type Expr interface {
    Node
    exprNode()
}
```

`ast.Expr` は（埋め込まれている `ast.Node` を除けば）外部に公開されないメソッドで構成されています。そのため、これを実装する型は `ast` パッケージに定義されているものに限られます。

`exprNode()` は実際にはどこからも呼ばれないメソッドなので、`ast.Expr` はその振る舞いに関する情報を提供しない、分類用のインターフェースであるといえます（後で見ますが、`ast.Node` も構文木に対する情報を含んだメソッドを提供しません）。同様に、文や宣言に対応するインターフェース（`ast.Stmt` と `ast.Decl`）も定義されています。埋め込まれている `ast.Node` インターフェースも含め、これらについて詳しくは構文ノードの実装で見ます。

1.2. ファイルの構文解析

実践においては、Goのソースコードはファイルかパッケージの単位で扱うことが普通です。ここからはファイル全体を構文解析する方法を見ていきます。

1.2.1. ファイルの構造

まず、Goのソースコードファイルの構造を確認しておきましょう。

The Go Programming Language Specification - Source file organization によれば、ひとつのファイルの中には

1. パッケージ名
2. `import` 節
3. 値や関数などトップレベルの宣言

の順番で現れることになっています。

1.2.2. parser.ParseFile

Goのソースコードファイルの構文解析を行うには `parser.ParseFile` を使用します。

godoc: go/parser.ParseFile

```
func ParseFile(fset *token.FileSet, filename string, src interface{}, mode Mode) (f
*ast.File, err error)
```

第二引数の `filename` と第三引数の `src` はふたつで一組になっていて、構文解析するソースコードを指定します。`src != nil` であるときは `filename` に指定されたファイルを読み込み、それ以外の場合は `src` をソースコードとして読み込み、`filename` はソースコードの位置情報にだけ使われます。`src` に指定できるのは `string` か `[]byte` か `io.Reader` のいずれかのみです。

第一引数の `fset` は構文解析によって得られた構文木のノードの詳細な位置情報を保持する `token.FileSet` 構造体へのポインタです。詳しくはソースコード中の位置で説明しますが、基本的に、`token.NewFileSet()` で得られるものを渡せば十分です。

最後の引数 `mode` では構文解析する範囲の指定などが行えます。後でコメントとドキュメントを扱うときに少し触れます。

リスト *3. parsefile.go*

```
package main

import (
    "fmt"
    "go/ast"
    "go/parser"
    "go/token"
)

func main() {
    fset := token.NewFileSet()
    f, _ := parser.ParseFile(fset, "example.go", src, parser.Mode(0))

    for _, d := range f.Decls {
        ast.Print(fset, d)
        fmt.Println()
    }
}

var src = `package p
import _ "log"
func add(n, m int) {}
`
```

```
     0  *ast.GenDecl {
     1  .  TokPos: example.go:2:1
     2  .  Tok: import
     3  .  Lparen: -
     4  .  Specs: []ast.Spec (len = 1) {
     5  .  .  0: *ast.ImportSpec {
     6  .  .  .  Name: *ast.Ident {
     7  .  .  .  .  NamePos: example.go:2:8
     8  .  .  .  .  Name: "_"
     9  .  .  .  }
    10  .  .  .  Path: *ast.BasicLit {
    11  .  .  .  .  ValuePos: example.go:2:10
    12  .  .  .  .  Kind: STRING
    13  .  .  .  .  Value: "\"log\""
    14  .  .  .  }
    15  .  .  .  EndPos: -
    16  .  .  }
    17  .  }
    18  .  Rparen: -
    19  }

     0  *ast.FuncDecl {
     1  .  Name: *ast.Ident {
     2  .  .  NamePos: example.go:3:6
```

```
     3  .  .  Name: "add"
     4  .  .  Obj: *ast.Object {
     5  .  .  .  Kind: func
     6  .  .  .  Name: "add"
     7  .  .  .  Decl: *(obj @ 0)
     8  .  .  }
     9  .  }
    10  .  Type: *ast.FuncType {
    11  .  .  Func: example.go:3:1
    12  .  .  Params: *ast.FieldList {
    13  .  .  .  Opening: example.go:3:9
    14  .  .  .  List: []*ast.Field (len = 1) {
    15  .  .  .  .  0: *ast.Field {
    16  .  .  .  .  .  Names: []*ast.Ident (len = 2) {
    17  .  .  .  .  .  .  0: *ast.Ident {
    18  .  .  .  .  .  .  .  NamePos: example.go:3:10
    19  .  .  .  .  .  .  .  Name: "n"
    20  .  .  .  .  .  .  .  Obj: *ast.Object {
    21  .  .  .  .  .  .  .  .  Kind: var
    22  .  .  .  .  .  .  .  .  Name: "n"
    23  .  .  .  .  .  .  .  .  Decl: *(obj @ 15)
    24  .  .  .  .  .  .  .  }
    25  .  .  .  .  .  .  }
    26  .  .  .  .  .  .  1: *ast.Ident {
    27  .  .  .  .  .  .  .  NamePos: example.go:3:13
    28  .  .  .  .  .  .  .  Name: "m"
    29  .  .  .  .  .  .  .  Obj: *ast.Object {
    30  .  .  .  .  .  .  .  .  Kind: var
    31  .  .  .  .  .  .  .  .  Name: "m"
    32  .  .  .  .  .  .  .  .  Decl: *(obj @ 15)
    33  .  .  .  .  .  .  .  }
    34  .  .  .  .  .  .  }
    35  .  .  .  .  .  }
    36  .  .  .  .  .  Type: *ast.Ident {
    37  .  .  .  .  .  .  NamePos: example.go:3:15
    38  .  .  .  .  .  .  Name: "int"
    39  .  .  .  .  .  }
    40  .  .  .  .  }
    41  .  .  .  }
    42  .  .  .  Closing: example.go:3:18
    43  .  .  }
    44  .  }
    45  .  Body: *ast.BlockStmt {
    46  .  .  Lbrace: example.go:3:20
    47  .  .  Rbrace: example.go:3:21
    48  .  }
    49  }
```

例では `src` 変数のもつソースコードを構文解析し、トップレベルの宣言を印字します。今回は `import` 宣言が `*ast.GenDecl` として、関数 `func f` が `*ast.FuncDecl` として得られました。

1.2.3. ast.File

ファイルは `ast.File` 構造体で表現され、パッケージ名やトップレベルの宣言の情報を含んでいます。

godoc: go/ast.File

```
type File struct {
    Doc        *CommentGroup   // associated documentation; or nil
    Package    token.Pos       // position of "package" keyword
    Name       *Ident          // package name
    Decls      []Decl          // top-level declarations; or nil
    Scope      *Scope          // package scope (this file only)
    Imports    []*ImportSpec   // imports in this file
    Unresolved []*Ident        // unresolved identifiers in this file
    Comments   []*CommentGroup // list of all comments in the source file
}
```

他にもいろいろなフィールドがありますが、

- `Package` はソースコード中の位置、
- `Scope` と `Unresolved` はスコープ、
- `Doc` と `Comments` はコメントとドキュメント

で解説します。

1.2.4. 構文木の探索

先に述べたように構文木のノードは `ast` パッケージのインターフェースとして得られるので、具体的な内容を知るには type assertion や type switch を用いなければなりません。これを手で書いていくのは大変で間違いも起きがちですが、`ast.Inspect` 関数を使えば構文ノードに対する深さ優先探索を行えます。

godoc: go/ast.Inspect

```
func Inspect(node Node, f func(Node) bool)
```

各ノードを引数に呼ばれるコールバック関数 `f` が `true` を返すと、さらにその下のノードを探索していきます。

以下は先ほどのソースコードファイル中の識別子を一覧する例です。`ast.Node` の具体的な型を知るために、type assertion をおこなっています。

リスト *4. listidents.go*

```
package main

import (
    "fmt"
    "go/ast"
    "go/parser"
    "go/token"
)

func main() {
    fset := token.NewFileSet()
    f, _ := parser.ParseFile(fset, "example.go", src, parser.Mode(0))

    ast.Inspect(f, func(n ast.Node) bool {
        if ident, ok := n.(*ast.Ident); ok {
            fmt.Println(ident.Name)
        }
        return true
    })
}

var src = `package p
import _ "log"
func add(n, m int) {}
`
```

```
p
_
add
n
m
int
```

パッケージ名や変数名などの識別子が構文木に含まれていることが確認できます。

もうひとつの方法として、`ast.Visitor` インターフェースを実装して `ast.Walk(v ast.Visitor, node ast.Node)` を使うこともできます。実際 `ast.Inspect` の内部では `ast.Walk` が使われています。

1.3. 構文ノードの実装

1.3.1. ast.Node

抽象構文木のノードに対応する構造体は、すべて `ast.Node` インターフェースを実装しています。

godoc: go/ast.Node

```
type Node interface {
    Pos() token.Pos // position of first character belonging to the node
    End() token.Pos // position of first character immediately after the node
}
```

定義を見れば分かる通り、ast.Node インターフェース自身はそのソースコード中の位置を提供するだけであり、このままでは構文木に関する情報を得ることはできません。構文木を探索・操作するには type assertion や type swtich による具体的な型への変換が必要になります。

構文木のノードを大別するため、ast.Node を実装するサブインターフェースが定義されています:

ast.Decl
: 宣言（declaration）。import や type など

ast.Stmt
: 文（statement）。if や switch など

ast.Expr
: 式（expression）。識別子や演算、型など

以下でこれらのうち主要なものを見て行きます。

ast.Node の階層で ast.Node を実装する型の完全な一覧を確認できます。

1.3.2. ast.Decl

ast.Decl インターフェースはGoソースコードにおける宣言（declaration）に対応する構文木のノードを表します。Goの宣言には

- パッケージのインポート（import）
- 変数および定数（var、const）
- 型（type）
- 関数およびメソッド（func）

といったものがありますが、ast.Decl インターフェースを実装している構造体は *ast.GenDecl と *ast.FuncDecl の2つのみです。後者は関数及びメソッドの宣言に相当し、前者が残りすべてをカバーします。

ast.FuncDecl

godoc: go/ast.FuncDecl

```
type FuncDecl struct {
    Doc  *CommentGroup // associated documentation; or nil
    Recv *FieldList    // receiver (methods); or nil (functions)
    Name *Ident        // function/method name
    Type *FuncType     // function signature: parameters, results, and position of
"func" keyword
    Body *BlockStmt    // function body; or nil (forward declaration)
}
```

`ast.FuncDecl` 構造体は関数の宣言に対応します。`Recv` フィールドはそのレシーバを表しており、これが `nil` か否かで関数の宣言かメソッドの宣言かを区別できます。

`Recv` の型である `*ast.FieldList`は識別子と型の組のリストで、関数のパラメータや構造体のフィールドを表すのに使われます。`FieldList` はその名の通り複数の組を表すこともできますが、Goの文法上、レシーバとしてはただ1つの組のみが有効です。（ただし、`go/parser` は複数の組からなるレシーバをエラーなく解析します！）

ast.GenDecl

godoc: go/ast.GenDecl

```
type GenDecl struct {
    Doc    *CommentGroup // associated documentation; or nil
    TokPos token.Pos     // position of Tok
    Tok    token.Token   // IMPORT, CONST, TYPE, VAR
    Lparen token.Pos     // position of '(', if any
    Specs  []Spec
    Rparen token.Pos // position of ')', if any
}
```

`import`、`const`、`var`、`type` の宣言は `ast.GenDecl` がまとめて引き受けます。`Specs` フィールドは `ast.Spec` インターフェースのスライスと宣言されていますが、要素の型は `*ast.ImportSpec`、`*ast.ValueSpec`、`*ast.TypeSpec` のいずれかひとつであり、`Tok` フィールドの値によって決まります。`*ast.ValueSpec` は `const` と `var` の場合両方をカバーします。

これらの宣言は、以下のようにグループ化できるという共通点があります。グループ化された宣言のひとつが `Specs` のひとつの要素に対応します。

```
import (
    "foo"
    "bar"
)

const (
    a = 1
    b = 2
)

var (
    x int
    y bool
)

type (
    t struct{}
    y interface{}
)
```

1.3.3. ast.Stmt

ast.Decl インターフェースはGoソースコードにおける 文 に対応する構文木のノードを表します。文はプログラムの実行を制御するもので、go/ast パッケージの実装では以下のように分類されています:

ast.Decl の分類

- 宣言（ast.DeclStmt）
- 空の文（ast.EmptyStmt）
- ラベル付き文（ast.LabeledStmt）
- 式だけの文（ast.ExprStmt）
- チャンネルへの送信（ast.SendStmt）
- インクリメント・デクリメント（ast.IncDecStmt）
- 代入または定義（ast.AssignStmt）
- go （ast.GoStmt）
- defer（ast.DeferStmt）
- return（ast.ReturnStmt）
- break、continue、goto、fallthrough（ast.BranchStmt）
- ブロック（ast.BlockStmt）
- if（ast.IfStmt）
- 式による switch（ast.SwitchStmt）
- 型による switch（ast.TypeSwitchStmt）
- switch 中のひとつの節（ast.CaseClause）

- select（ast.SelectStmt）
- select 中のひとつの節（ast.CommClause）
- range を含まない for（ast.ForStmt）
- range を含む for（ast.RangeStmt）

ast.TypeSwitchStmt

- Assign

1.3.4. ast.Expr

ast.Expr インターフェースはおもにGoソースコードにおける 式 および 型 に対応する構文木のノードを表します。go/ast パッケージの実装では以下のように分類されています:

ast.Ellipsis や ast.KeyValueExpr のように、それ単体では式となり得ないノードも ast.Expr を実装していますが、このおかげでこれらを含むノードの実装が簡単になっているようです。

- 識別子（ast.Ident）
- …（ast.Ellipsis）
- 基本的な型のリテラル（ast.BasicLit）
- 関数リテラル（ast.FuncLit）
- 複合リテラル（ast.CompositeLit）
- 括弧（ast.ParenExpr）
- セレクタまたは修飾された識別子（x.y）（ast.SelectorExpr）
- 添字アクセス（ast.IndexExpr）
- スライス式（ast.SliceExpr）
- 型アサーション（ast.TypeAssertExpr）
- 関数またはメソッド呼び出し（ast.CallExpr）
- ポインタの間接参照またはポインタ型（*p）（ast.StarExpr）
- 単項演算（ast.UnaryExpr）
- 二項演算（ast.BinaryExpr）
- 複合リテラル中のキーと値のペア（key: value）（ast.KeyValueExpr）
- 配列またはスライス型（ast.ArrayType）
- 構造体型（ast.StructType）
- 関数型（ast.FuncType）
- インターフェース型（ast.InterfaceType）
- マップ型（ast.MapType）
- チャンネル型（ast.ChanType）

ast.Ident

godoc: go/ast.Ident

```
type Ident struct {
    NamePos token.Pos // identifier position
    Name    string    // identifier name
    Obj     *Object   // denoted object; or nil
}
```

`ast.Ident` はコード中の識別子を表し、変数名をはじめパッケージ名、ラベルなどさまざまな場所に登場します。

`Obj` フィールドはその実体を表す `ast.Object` への参照になっています。詳しくは スコープとオブジェクトで触れます。

ast.StructType と ast.InterfaceType

godoc: go/ast.StructType

```
type StructType struct {
    Struct     token.Pos  // position of "struct" keyword
    Fields     *FieldList // list of field declarations
    Incomplete bool       // true if (source) fields are missing in the Fields list
}
```

godoc: go/ast.InterfaceType

```
type InterfaceType struct {
    Interface  token.Pos  // position of "interface" keyword
    Methods    *FieldList // list of methods
    Incomplete bool       // true if (source) methods are missing in the Methods list
}
```

これら2つの構造体は `Incomplete` フィールドを持っています。これは通常 `false` ですが、フィルタによってノードが書き換えられ、フィールドやメソッドの宣言が取り除かれた場合に `true` が設定され、ソースコードとノードに乖離があることを示します。`go doc` が出力する "// contains filtered or unexported fields" はこの値を参照しています。

リスト *5. structtypeincomplete.go*

```
package main

import (
    "fmt"
    "go/ast"
    "go/parser"
    "go/token"
)

func main() {
    fset := token.NewFileSet()
    f, _ := parser.ParseFile(fset, "example.go", src, parser.Mode(0))

    structType := f.Decls[0].(*ast.GenDecl).Specs[
0].(*ast.TypeSpec).Type.(*ast.StructType)

    fmt.Printf("fields=%#v incomplete=%#v\n", structType.Fields.List,
structType.Incomplete)

    ast.FileExports(f)

    fmt.Printf("fields=%#v incomplete=%#v\n", structType.Fields.List,
structType.Incomplete)
}

var src = `package p
type S struct {
    Public  string
    private string
}
`
```

```
fields=[]*ast.Field{(*ast.Field)(0xc820014540), (*ast.Field)(0xc8200145c0)}
incomplete=false
fields=[]*ast.Field{(*ast.Field)(0xc820014540)} incomplete=true
```

1.3.5. その他のノード

以上の3種類に分類されないノードもいくつか存在します。

ast.Comment と ast.CommentGroup

godoc: go/ast.Comment

```
type Comment struct {
    Slash token.Pos // position of "/" starting the comment
    Text  string    // comment text (excluding '\n' for //-style comments)
}
```

godoc: go/ast.CommentGroup

```
type CommentGroup struct {
    List []*Comment // len(List) > 0
}
```

ast.Comment はひとつのコメント（// … または /* … */）に、ast.CommentGroup は連続するコメントに対応します。コメントとドキュメントで詳しく見ます。

ast.Field と ast.FieldList

godoc: go/ast.Field

```
type Field struct {
    Doc     *CommentGroup // associated documentation; or nil
    Names   []*Ident      // field/method/parameter names; or nil if anonymous field
    Type    Expr          // field/method/parameter type
    Tag     *BasicLit     // field tag; or nil
    Comment *CommentGroup // line comments; or nil
}
```

godoc: go/ast.FieldList

```
type FieldList struct {
    Opening token.Pos // position of opening parenthesis/brace, if any
    List    []*Field  // field list; or nil
    Closing token.Pos // position of closing parenthesis/brace, if any
}
```

それぞれ、識別子と型の組ひとつと、そのリストに対応します。

ast.FieldList は以下の構造体に含まれています:

- ast.StructType ..... 構造体のフィールドのリストとして
- ast.InterfaceType ..... インターフェースのメソッドのリストとして
- ast.FuncType ..... 関数のパラメータおよび返り値として
- ast.FuncDecl ..... メソッドのレシーバとして

ast.Field の Tag は構造体のフィールドである場合のみ存在しえます。

Appendix A: ast.Node の階層

```
ast.Node
  ast.Decl
    *ast.BadDecl
    *ast.FuncDecl
    *ast.GenDecl
  ast.Expr
    *ast.ArrayType
    *ast.BadExpr
    *ast.BasicLit
    *ast.BinaryExpr
    *ast.CallExpr
    *ast.ChanType
    *ast.CompositeLit
    *ast.Ellipsis
    *ast.FuncLit
    *ast.FuncType
    *ast.Ident
    *ast.IndexExpr
    *ast.InterfaceType
    *ast.KeyValueExpr
    *ast.MapType
    *ast.ParenExpr
    *ast.SelectorExpr
    *ast.SliceExpr
    *ast.StarExpr
    *ast.StructType
    *ast.TypeAssertExpr
    *ast.UnaryExpr
  ast.Spec
    *ast.ImportSpec
    *ast.TypeSpec
    *ast.ValueSpec
  ast.Stmt
    *ast.AssignStmt
    *ast.BadStmt
    *ast.BlockStmt
    *ast.BranchStmt
    *ast.CaseClause
    *ast.CommClause
    *ast.DeclStmt
    *ast.DeferStmt
    *ast.EmptyStmt
    *ast.ExprStmt
    *ast.ForStmt
    *ast.GoStmt
    *ast.IfStmt
    *ast.IncDecStmt
    *ast.LabeledStmt
```

```
    *ast.RangeStmt
    *ast.ReturnStmt
    *ast.SelectStmt
    *ast.SendStmt
    *ast.SwitchStmt
    *ast.TypeSwitchStmt
  *ast.Comment
  *ast.CommentGroup
  *ast.Field
  *ast.FieldList
  *ast.File
  *ast.Package
```

1.4. ソースコード中の位置

ソースコードを対象とするプログラムがユーザにフィードバックを行う際は、ファイル名や行番号など、ソースコードにおける位置情報を含めるのが通常です。

go vet の出力

```
% go vet github.com/motemen/gore
quickfix.go:76: go/ast.ExprStmt composite literal uses unkeyed fields
exit status 1
```

この節では、ソースコード中の位置情報を扱うためのAPIを見ていきます。

1.4.1. token.Pos

さて、すべての抽象構文木のノードはast.Nodeインターフェースを実装しているのでした。`ast.Node` は `token.Pos` を返す `Pos()` と `End()` の2つのメソッドで構成されます。

godoc: go/ast.Node

```
type Node interface {
    Pos() token.Pos // position of first character belonging to the node
    End() token.Pos // position of first character immediately after the node
}
```

これらはその名が示すとおり、当該のノードがソースコード上に占める開始位置と終了位置を表しています。`token.Pos` の実体は基準位置からオフセットを示す `int` 型です。

godoc: go/token.Pos

```
type Pos int
```

オフセット値は `1` から始まるバイト単位の値です。特に、`token.Pos` の zero value（`= 0`）には `token.NoPos` という特別な名前が与えられています。

godoc: go/token.NoPos

```
const NoPos Pos = 0
```

token.Pos は単なる整数値でしかないので、ファイル名や行番号などの詳細な情報をこれだけから得ることはできません。実はノードの持つこれらの位置情報は token.FileSet を基準にした相対的なものとしてエンコードされていて、完全な情報を復元するには2つを組み合わせる必要があります。token.FileSet はこれまでの例にも登場してきた（そして無視されてきた）fset と呼ばれる変数です。

ここから分かるように、構文解析の際に与える token.FileSet によってノードの構造体の値は変化します。抽象構文木を扱うプログラムでは、構文解析によって得られたノードはその基準となる token.FileSet とともに保持しておく必要があります。

CallExpr.Ellipsis や GenDecl.Lparen など、token.NoPos はその位置情報を持つ要素がソースコード中に存在しないことを意味する場合もあります。

1.4.2. token.FileSet

godoc: go/token.FileSet

```
type FileSet struct {
    // Has unexported fields.
}
```

token.FileSet は、go/parser が生成する抽象構文木のノードの位置情報を一手に引きうけ、保持する構造体です。ノードの構造体が保持する位置情報は前項で述べたように token.FileSet を基準にした相対的なもので、整数値としてエンコードされています。

名前の通り、token.FileSet が表すのは複数のソースファイルの集合です。ここでのファイルとは概念上のもので、ファイルシステム上に存在する必要はなく、またファイル名が重複していても問題ありません。

興味あるソースファイル集合に対して1つあれば十分なので、いちど token.NewFileSet() で得られた参照を保持しておくのが普通です。

godoc: go/token.NewFileSet

```
func NewFileSet() *FileSet
```

token.FileSet は、構文要素の具体的な位置を参照するAPIで要求されます。

- 構文木のノードを生成する際に必要です。
- ソースコードの文字列化に必要です。
- ast.Printに渡すと、token.Pos がダンプされる際にファイル名と行番号、カラム位置が表示されます。

token.FileSet はファイルそれぞれについて、

- ファイルの開始位置のオフセット
- 各行の長さ

をバイト単位で保持しており、整数値にエンコードされた位置情報から次に見る完全な位置情報を復元できます。

1.4.3. token.Position

godoc: go/token.Position

```
type Position struct {
    Filename string // filename, if any
    Offset   int    // offset, starting at 0
    Line     int    // line number, starting at 1
    Column   int    // column number, starting at 1 (byte count)
}
```

`token.Position` 構造体はファイル名、行番号、カラム位置を持ち、ソースコード中の位置としては最も詳細な情報を含みます。`String()` メソッドによって human-readable な位置情報を得られます。

リスト *6. positionstring.go*

```
package main

import (
    "fmt"
    "go/token"
)

func main() {
    fmt.Println("Invalid position without file name:", token.Position{}.String())
    fmt.Println("Invalid position with file name:   ", token.Position{Filename:
"example.go"}.String())
    fmt.Println("Valid position without file name:  ", token.Position{Line: 2, Column:
3}.String())
    fmt.Println("Valid position with file name:     ", token.Position{Filename:
"example.go", Line: 2, Column: 3}.String())
}
```

```
Invalid position without file name: -
Invalid position with file name:    example.go
Valid position without file name:   2:3
Valid position with file name:      example.go:2:3
```

1.5. スコープとオブジェクト

Goのソースコードにおいて名前はレキシカルスコープを持ち、その有効範囲は静的に決まります。構文解

析のAPIにもスコープに関係するものがいくつか存在します。以下ではこれらについて簡単に見ていきます。

1.5.1. 構文解析だけでは不十分な例

ただし、構文解析だけでは全ての名前を正しく解決できるわけではありません。例えば以下のプログラムには `T{k: 0}` という同じ形をしたコードが出現しますが、ここで `k` が指すものは一方ではトップレベルの定数、もう一方では構造体のフィールドと、それぞれ違ったものになります。（go/types: The Go Type Checker より）

リスト 7. *indeterminableident.go*

```
package p

const k = 0

func f1() {
    type T [1]int
    _ = T{k: 0}
}

func f2() {
    type T struct{ k int }
    _ = T{k: 0}
}
```

また、名前なしの `import` 文によって導入されたパッケージ名は構文解析だけでは判定できません。

```
import "github.com/motemen/go-gitconfig" // gitconfig という名前が導入される
```

これらも含めて正しく（Go言語の仕様通りに）名前のスコープを決定するには、意味解析の手続きを経なくてはなりません。

このように、`go/ast` のAPIで得られるスコープの情報は不完全なもので、ソースコードが構文的に一定の正しさを持っていることを保証するものです。より正確で詳しい情報が知りたい場合には型解析を行います。

1.5.2. スコープ

Go　のスコープはブロックにもとづいて作られます。ブレース（{…}）による明示的なブロックの他に作られるブロックもあります。Declarations and scope - The Go Programming Language Specification に述べられていますが、抄訳します:

- あらかじめ定義されている名前（`nil` など）の属するユニバースブロック（universe block）に属します。
- トップレベルの定数、変数、関数（メソッドは除く）はパッケージブロック（package block）に属します。

インポートされたパッケージの名前は、それを含むファイルブロック（file block）に属します。

go/ast のAPIを使用してアクセスできるのはファイルブロックのスコープとパッケージブロックのスコープ（パッケージ）のみです。

構文解析によって得られたスコープは ast.Scope として表現されます:

godoc: go/ast.Scope

```
type Scope struct {
    Outer   *Scope
    Objects map[string]*Object
}
```

スコープはその外側のスコープへの参照と、名前からオブジェクトへのマッピングで構成されています。

1.5.3. オブジェクト

ソースコード中の識別子を表す ast.Ident には、*ast.Object 型の Obj というフィールドが定義されていました。

godoc: go/ast.Ident

```
type Ident struct {
    NamePos token.Pos // identifier position
    Name    string    // identifier name
    Obj     *Object   // denoted object; or nil
}
```

go/ast や go/types では名前をつけられた言語上の実体（named language entity）をオブジェクト（object）と呼んでおり、構文上のオブジェクトはこの ast.Object によって表されています。

godoc: go/ast.Object

```
type Object struct {
    Kind ObjKind
    Name string      // declared name
    Decl interface{} // corresponding Field, XxxSpec, FuncDecl, LabeledStmt,
AssignStmt, Scope; or nil
    Data interface{} // object-specific data; or nil
    Type interface{} // placeholder for type information; may be nil
}
```

構文上同じオブジェクトを指すと思わしき識別子は、同じ ast.Object を共有します。

Kind フィールドは ObjKind 型に定義されている値のいずれかを取り、オブジェクトの種類を表します。

-

godoc: go/ast.Bad

```
const (
    Bad ObjKind = iota // for error handling
    Pkg                // package
    Con                // constant
    Typ                // type
    Var                // variable
    Fun                // function or method
    Lbl                // label
)
```

パッケージ名、定数名、型名、変数名、関数名またはメソッド名に加え、ラベル名もオブジェクトとして扱われることがわかります。

`ast.Object` の `Decl` フィールドはそのオブジェクトが宣言されたノードを表します。

`Data` や `Type` などさらに詳しい情報を保持するためのフィールドもありますが、先に述べたように構文解析だけでは完全な情報が得られないので、これらの詳しい情報が必要な場合には型解析のAPIを使用することになるでしょう。

1.5.4. パッケージ

Goでは、複数のソースファイルが集まってひとつのパッケージを構成します。コンパイルの成果物はパッケージ単位で生成され、パッケージを構成するソースコード中の名前はすべて解決できなければなりません。

ast.File.Scope と ast.File.Unresolved

godoc: go/ast.File

```
type File struct {
    Doc        *CommentGroup   // associated documentation; or nil
    Package    token.Pos       // position of "package" keyword
    Name       *Ident          // package name
    Decls      []Decl          // top-level declarations; or nil
    Scope      *Scope          // package scope (this file only)
    Imports    []*ImportSpec   // imports in this file
    Unresolved []*Ident        // unresolved identifiers in this file
    Comments   []*CommentGroup // list of all comments in the source file
}
```

`ast.File` 構造体の `Scope` フィールドは当該のソースファイルのファイルスコープ（ファイルブロック）を表します。ここで解決できなかったものは `Unresolved` フィールドに記録されます。正しくコンパイルできるソースコードであれば、ここに入るのは

- `imoport` によってファイルスコープに導入される名前
- 同パッケージの他ファイルのトップレベルに定義されている名前
- ユニバースブロックに定義されている名前

への参照です。

ast.Package

godoc: go/ast.Package

```
type Package struct {
    Name    string             // package name
    Scope   *Scope             // package scope across all files
    Imports map[string]*Object // map of package id -> package object
    Files   map[string]*File   // Go source files by filename
}
```

複数の `ast.File` をまとめたものが `ast.Package` です。先ほど触れたトップレベルの名前や外部から導入される名前が解決された結果が `Scope` や `Imports` フィールドに格納されます。

`ast.Package` は `ast.NewPackage` で生成されます。

godoc: go/ast.NewPackage

```
func NewPackage(fset *token.FileSet, files map[string]*File, importer Importer,
universe *Scope) (*Package, error)
```

第4引数の `universe *ast.Scope` には、パッケージの外側のスコープであるユニバーススコープを渡します。

第3引数の `importer ast.Importer` は、`import` されるパスから、それが導入するオブジェクトを記録しつつ返す関数を渡します。

godoc: go/ast.Importer

```
type Importer func(imports map[string]*Object, path string) (pkg *Object, err error)
```

これらを正しく渡すことで完全な `ast.Package` を生成することができますが、（しつこいようですが）正しい情報が必要な場合には型解析を行うことを考えたほうがよいでしょう。

TODO: golang/gddo の例

1.5.5. parser.ParseDir

Goでは、ひとつのパッケージに属するソースコードファイルは同じディレクトリ直下に配置されます。これらを一度に構文解析し、`ast.Package` を生成するAPIもあります。

godoc: go/parser.ParseDir

```
func ParseDir(fset *token.FileSet, path string, filter func(os.FileInfo) bool, mode
Mode) (pkgs map[string]*ast.Package, first error)
```

この関数では `ast.NewPackage` で行われるような名前の解決は行われません。

`ParseDir` はひとつのディレクトリから複数のパッケージを返しうるAPIになっていますが、正常にコンパイルできるような構成においても、複数のパッケージがひとつのディレクトリに共在することはありえます（Test packages）。

Chapter 2. コメントとドキュメント

これまではプログラムの実行に関わるコード本体をプログラムから扱う方法について見てきました。この章ではGoソースコード中のコメントを扱っていきます。

コメントはドキュメントの記述にも使用されており、そのためのAPIも用意されています。

2.1. コメントの解析

`parser.ParseFile` の第4引数 `mode` に `parser.ParseComments` 定数を指定することで、構文解析の結果にコメントを含めることができます。

godoc: go/parser.ParseFile

```
func ParseFile(fset *token.FileSet, filename string, src interface{}, mode Mode) (f
*ast.File, err error)
```

リスト *8. parsecomment.go*

```
package main

import (
    "fmt"
    "go/parser"
    "go/token"
)

func main() {
    fset := token.NewFileSet()
    f, _ := parser.ParseFile(fset, "example.go", src, parser.ParseComments)

    for _, c := range f.Comments {
        fmt.Printf("%s: %q\n", fset.Position(c.Pos()), c.Text())
    }
}

var src = `// Package p provides Add function
// ...
package p

// Add adds two ints.
func add(n, m int) int {
    return n + m
}
`
```

```
example.go:1:1: "Package p provides Add function\n...\n"
example.go:5:1: "Add adds two ints.\n"
```

こうやって解析されたコメントは通常の構文木とは別に、`ast.File` 構造体の `Comments` フィールドに格納されます。`Comments` フィールドは `[]*ast.CommentGroup` として宣言されています。`ast.CommentGroup` は連続して続くコメントをひとまとめにしたもので、

- `/* … */` 形式のコメントなら `/*` から `*/` まで
- `// …` 形式なら `//` から行末まで

が、ひとつの `ast.Comment` に対応します。

godoc: go/ast.CommentGroup

```
type CommentGroup struct {
    List []*Comment // len(List) > 0
}
```

godoc: go/ast.Comment

```
type Comment struct {
    Slash token.Pos // position of "/" starting the comment
    Text  string    // comment text (excluding '\n' for //-style comments)
}
```

例えばコメントが以下のように書かれていた場合、それぞれ `CommentGroup` は2つ生成され、それぞれ2個の `Comment` を持ちます。

```
// foo
/* bar */

// baz
// quux
```

コメントも `ast.Node` インターフェースを実装し、位置情報を保持しています。ソースコードの文字列化の際は、この位置情報をもとにコメントが正しい位置に挿入されるようになっています。

2.2. Goにおけるドキュメント

Goではトップレベルの型や関数のすぐ直前のコメントがそのAPIのドキュメントである、と標準的に定められています（Godoc: documenting Go code）。標準の `go doc` コマンドもこのルールに則ってドキュメントを

```
% go doc go/parser.ParseFile
func ParseFile(fset *token.FileSet, filename string, src interface{}, mode Mode) (f
*ast.File, err error)
    ParseFile parses the source code of a single Go source file and returns the
    corresponding ast.File node. The source code may be provided via the
    filename of the source file, or via the src parameter.
...
```

2.3. doc.New

2.4. 例示コード

- doc.Examples

Chapter 3. ソースコードの文字列化

WIP

- go/printer
- go/format

Chapter 4. 型解析

WIP

- go/types

Chapter 5. ビルド情報

WIP

- go/build

Chapter 6. 高レベルのAPI

WIP

- tools/go/loader

Chapter 7. 標準コマンドを読む

WIP

- go doc
- go fmt
- stringer
- gorename
- guru
- goimports

Chapter 8. フロー解析

WIP

- tools/go/ssa