project construct:
H:\Go\gopath\src\golang.org\x\xerrors
```
├──── LICENSE
├──── PATENTS
├──── README
├──── adaptor.go
├──── codereview.cfg
├──── doc.go
├──── errors.go
├──── errors_test.go
├──── example_As_test.go
├──── example_FormatError_test.go
├──── example_test.go
├──── fmt.go
├──── fmt_test.go
├──── fmt_unexported_test.go
├──── format.go
├──── frame.go
├──── go.mod
├──── stack_test.go
├──── wrap.go
├──── wrap_113_test.go
├──── wrap_test.go
└──── internal
    └──── internal.go
```

H:\Go\gopath\src\golang.org\x\xerrors\fmt_unexported_test.go:

```
// Copyright 2018 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package xerrors

import "testing"

func TestParsePrintfVerb(t *testing.T) {
    for _, test := range []struct {
        in       string
        wantSize int
        wantW    bool
    }{
        {"", 0, false},
        {"%", 1, false},
        {"%3.1", 4, false},
        {"%w", 2, true},
        {"%v", 2, false},
        {"%3.*[4]d", 8, false},
    } {
        gotSize, gotW := parsePrintfVerb(test.in)
        if gotSize != test.wantSize || gotW != test.wantW {
            t.Errorf("parsePrintfVerb(%q) = (%d, %t), want (%d, %t)",
```

```
                test.in, gotSize, gotW, test.wantSize, test.wantW)
        }
    }
}

func TestParsePercentW(t *testing.T) {
    for _, test := range []struct {
        in         string
        wantIdx    int
        wantFormat string
        wantOK     bool
    }{
        {"", -1, "", true},
        {"%", -1, "%", true},
        {"%w", 0, "%v", true},
        {"%w%w", 0, "%v%v", false},
        {"%3.2s %+q %% %w %#v", 2, "%3.2s %+q %% %v %#v", true},
        {"%3.2s %w %% %w %#v", 1, "%3.2s %v %% %v %#v", false},
    } {
        gotIdx, gotFormat, gotOK := parsePercentW(test.in)
        if gotIdx != test.wantIdx || gotFormat != test.wantFormat || gotOK
!= test.wantOK {
            t.Errorf("parsePercentW(%q) = (%d, %q, %t), want (%d, %q, %t)",
                test.in, gotIdx, gotFormat, gotOK, test.wantIdx,
test.wantFormat, test.wantOK)

        }
    }
}
```

H:\Go\gopath\src\golang.org\x\xerrors\format.go:

```
// Copyright 2018 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package xerrors

// A Formatter formats error messages.
type Formatter interface {
    error

    // FormatError prints the receiver's first error and returns the next
error in
    // the error chain, if any.
    FormatError(p Printer) (next error)
}

// A Printer formats error messages.
//
// The most common implementation of Printer is the one provided by package
fmt
```

```
// during Printf (as of Go 1.13). Localization packages such as
golang.org/x/text/message
// typically provide their own implementations.
type Printer interface {
    // Print appends args to the message output.
    Print(args ...interface{})

    // Printf writes a formatted string.
    Printf(format string, args ...interface{})

    // Detail reports whether error detail is requested.
    // After the first call to Detail, all text written to the Printer
    // is formatted as additional detail, or ignored when
    // detail has not been requested.
    // If Detail returns false, the caller can avoid printing the detail at
all.
    Detail() bool
}
```

H:\Go\gopath\src\golang.org\x\xerrors\frame.go:

```
// Copyright 2018 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package xerrors

import (
    "runtime"
)

// A Frame contains part of a call stack.
type Frame struct {
    // Make room for three PCs: the one we were asked for, what it called,
    // and possibly a PC for skipPleaseUseCallersFrames. See:
    // https://go.googlesource.com/go/+/032678e0fb/src/runtime/extern.go#169
    frames [3]uintptr
}

// Caller returns a Frame that describes a frame on the caller's stack.
// The argument skip is the number of frames to skip over.
// Caller(0) returns the frame for the caller of Caller.
func Caller(skip int) Frame {
    var s Frame
    runtime.Callers(skip+1, s.frames[:])
    return s
}

// location reports the file, line, and function of a frame.
//
// The returned function may be "" even if file and line are not.
func (f Frame) location() (function, file string, line int) {
```

```
    frames := runtime.CallersFrames(f.frames[:])
    if _, ok := frames.Next(); !ok {
        return "", "", 0
    }
    fr, ok := frames.Next()
    if !ok {
        return "", "", 0
    }
    return fr.Function, fr.File, fr.Line
}

// Format prints the stack as error detail.
// It should be called from an error's Format implementation
// after printing any other error detail.
func (f Frame) Format(p Printer) {
    if p.Detail() {
        function, file, line := f.location()
        if function != "" {
            p.Printf("%s\n    ", function)
        }
        if file != "" {
            p.Printf("%s:%d\n", file, line)
        }
    }
}
```

H:\Go\gopath\src\golang.org\x\xerrors\wrap_test.go:

```
// Copyright 2018 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package xerrors_test

import (
    "fmt"
    "os"
    "testing"

    "golang.org/x/xerrors"
)

func TestIs(t *testing.T) {
    err1 := xerrors.New("1")
    erra := xerrors.Errorf("wrap 2: %w", err1)
    errb := xerrors.Errorf("wrap 3: %w", erra)
    erro := xerrors.Opaque(err1)
    errco := xerrors.Errorf("opaque: %w", erro)
    err3 := xerrors.New("3")

    poser := &poser{"either 1 or 3", func(err error) bool {
        return err == err1 || err == err3
```

```
    }}

    testCases := []struct {
        err    error
        target error
        match  bool
    }{
        {nil, nil, true},
        {nil, err1, false},
        {err1, nil, false},
        {err1, err1, true},
        {erra, err1, true},
        {errb, err1, true},
        {errco, erro, true},
        {errco, err1, false},
        {erro, erro, true},
        {err1, err3, false},
        {erra, err3, false},
        {errb, err3, false},
        {poser, err1, true},
        {poser, err3, true},
        {poser, erra, false},
        {poser, errb, false},
        {poser, erro, false},
        {poser, errco, false},
        {errorUncomparable{}, errorUncomparable{}, true},
        {errorUncomparable{}, &errorUncomparable{}, false},
        {&errorUncomparable{}, errorUncomparable{}, true},
        {&errorUncomparable{}, &errorUncomparable{}, false},
        {errorUncomparable{}, err1, false},
        {&errorUncomparable{}, err1, false},
    }
    for _, tc := range testCases {
        t.Run("", func(t *testing.T) {
            if got := xerrors.Is(tc.err, tc.target); got != tc.match {
                t.Errorf("Is(%v, %v) = %v, want %v", tc.err, tc.target, got,
tc.match)
            }
        })
    }
}

type poser struct {
    msg string
    f   func(error) bool
}

func (p *poser) Error() string     { return p.msg }
func (p *poser) Is(err error) bool { return p.f(err) }
func (p *poser) As(err interface{}) bool {
    switch x := err.(type) {
    case **poser:
        *x = p
    case *errorT:
```

```
        *x = errorT{}
    case **os.PathError:
        *x = &os.PathError{}
    default:
        return false
    }
    return true
}

func TestAs(t *testing.T) {
    var errT errorT
    var errP *os.PathError
    var timeout interface{ Timeout() bool }
    var p *poser
    _, errF := os.Open("non-existing")

    testCases := []struct {
        err    error
        target interface{}
        match  bool
    }{{
        nil,
        &errP,
        false,
    }, {
        xerrors.Errorf("pittied the fool: %w", errorT{}),
        &errT,
        true,
    }, {
        errF,
        &errP,
        true,
    }, {
        xerrors.Opaque(errT),
        &errT,
        false,
    }, {
        errorT{},
        &errP,
        false,
    }, {
        errWrap{nil},
        &errT,
        false,
    }, {
        &poser{"error", nil},
        &errT,
        true,
    }, {
        &poser{"path", nil},
        &errP,
        true,
    }, {
        &poser{"oh no", nil},
```

```
        &p,
        true,
    }, {
        xerrors.New("err"),
        &timeout,
        false,
    }, {
        errF,
        &timeout,
        true,
    }, {
        xerrors.Errorf("path error: %w", errF),
        &timeout,
        true,
    }}
    for i, tc := range testCases {
        name := fmt.Sprintf("%d:As(Errorf(..., %v), %v)", i, tc.err,
tc.target)
        t.Run(name, func(t *testing.T) {
            match := xerrors.As(tc.err, tc.target)
            if match != tc.match {
                t.Fatalf("xerrors.As(%T, %T): got %v; want %v", tc.err,
tc.target, match, tc.match)
            }
            if !match {
                return
            }
            if tc.target == nil {
                t.Fatalf("non-nil result after match")
            }
        })
    }
}

func TestAsValidation(t *testing.T) {
    var s string
    testCases := []interface{}{
        nil,
        (*int)(nil),
        "error",
        &s,
    }
    err := xerrors.New("error")
    for _, tc := range testCases {
        t.Run(fmt.Sprintf("%T(%v)", tc, tc), func(t *testing.T) {
            defer func() {
                recover()
            }()
            if xerrors.As(err, tc) {
                t.Errorf("As(err, %T(%v)) = true, want false", tc, tc)
                return
            }
            t.Errorf("As(err, %T(%v)) did not panic", tc, tc)
        })
```

```
    }
}

func TestUnwrap(t *testing.T) {
    err1 := xerrors.New("1")
    erra := xerrors.Errorf("wrap 2: %w", err1)
    erro := xerrors.Opaque(err1)

    testCases := []struct {
        err  error
        want error
    }{
        {nil, nil},
        {errWrap{nil}, nil},
        {err1, nil},
        {erra, err1},
        {xerrors.Errorf("wrap 3: %w", erra), erra},

        {erro, nil},
        {xerrors.Errorf("opaque: %w", erro), erro},
    }
    for _, tc := range testCases {
        if got := xerrors.Unwrap(tc.err); got != tc.want {
            t.Errorf("Unwrap(%v) = %v, want %v", tc.err, got, tc.want)
        }
    }
}

func TestOpaque(t *testing.T) {
    got := fmt.Sprintf("%v", xerrors.Errorf("foo: %v",
xerrors.Opaque(errorT{})))
    want := "foo: errorT"
    if got != want {
        t.Errorf("error without Format: got %v; want %v", got, want)
    }

    got = fmt.Sprintf("%v", xerrors.Errorf("foo: %v",
xerrors.Opaque(errorD{})))
    want = "foo: errorD"
    if got != want {
        t.Errorf("error with Format: got %v; want %v", got, want)
    }
}

type errorT struct{}

func (errorT) Error() string { return "errorT" }

type errorD struct{}

func (errorD) Error() string { return "errorD" }

func (errorD) FormatError(p xerrors.Printer) error {
    p.Print("errorD")
```

```
    p.Detail()
    p.Print("detail")
    return nil
}

type errWrap struct{ error }

func (errWrap) Error() string { return "wrapped" }

func (errWrap) Unwrap() error { return nil }

type errorUncomparable struct {
    f []string
}

func (errorUncomparable) Error() string {
    return "uncomparable error"
}

func (errorUncomparable) Is(target error) bool {
    _, ok := target.(errorUncomparable)
    return ok
}
```

H:\Go\gopath\src\golang.org\x\xerrors\doc.go:

```
// Copyright 2019 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

// Package xerrors implements functions to manipulate errors.
//
// This package is based on the Go 2 proposal for error values:
//
//    https://golang.org/design/29934-error-values
//
// These functions were incorporated into the standard library's errors
package
// in Go 1.13:
// - Is
// - As
// - Unwrap
//
// Also, Errorf's %w verb was incorporated into fmt.Errorf.
//
// Use this package to get equivalent behavior in all supported Go versions.
//
// No other features of this package were included in Go 1.13, and at
present
// there are no plans to include any of them.
package xerrors // import "golang.org/x/xerrors"
```

H:\Go\gopath\src\golang.org\x\xerrors\errors_test.go:

```
// Copyright 2011 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package xerrors_test

import (
    "fmt"
    "regexp"
    "testing"

    "golang.org/x/xerrors"
)

func TestNewEqual(t *testing.T) {
    // Different allocations should not be equal.
    if xerrors.New("abc") == xerrors.New("abc") {
        t.Errorf(`New("abc") == New("abc")`)
    }
    if xerrors.New("abc") == xerrors.New("xyz") {
        t.Errorf(`New("abc") == New("xyz")`)
    }

    // Same allocation should be equal to itself (not crash).
    err := xerrors.New("jkl")
    if err != err {
        t.Errorf(`err != err`)
    }
}

func TestErrorMethod(t *testing.T) {
    err := xerrors.New("abc")
    if err.Error() != "abc" {
        t.Errorf(`New("abc").Error() = %q, want %q`, err.Error(), "abc")
    }
}

func TestNewDetail(t *testing.T) {
    got := fmt.Sprintf("%+v", xerrors.New("error"))
    want := `(?s)error:.+errors_test.go:\d+`
    ok, err := regexp.MatchString(want, got)
    if err != nil {
        t.Fatal(err)
    }
    if !ok {
        t.Errorf(`fmt.Sprintf("%%+v", New("error")) = %q, want %q"`, got,
want)
    }
}
```

```
func ExampleNew() {
    err := xerrors.New("emit macho dwarf: elf header corrupted")
    if err != nil {
        fmt.Print(err)
    }
    // Output: emit macho dwarf: elf header corrupted
}

// The fmt package's Errorf function lets us use the package's formatting
// features to create descriptive error messages.
func ExampleNew_errorf() {
    const name, id = "bimmler", 17
    err := fmt.Errorf("user %q (id %d) not found", name, id)
    if err != nil {
        fmt.Print(err)
    }
    // Output: user "bimmler" (id 17) not found
}
```

H:\Go\gopath\src\golang.org\x\xerrors\example_As_test.go:

```
// Copyright 2019 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package xerrors_test

import (
    "fmt"
    "os"

    "golang.org/x/xerrors"
)

func ExampleAs() {
    _, err := os.Open("non-existing")
    if err != nil {
        var pathError *os.PathError
        if xerrors.As(err, &pathError) {
            fmt.Println("Failed at path:", pathError.Path)
        }
    }

    // Output:
    // Failed at path: non-existing
}
```

H:\Go\gopath\src\golang.org\x\xerrors\example_FormatError_test.go:

```
// Copyright 2019 The Go Authors. All rights reserved.
```

```
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package xerrors_test

import (
    "fmt"

    "golang.org/x/xerrors"
)

type MyError2 struct {
    Message string
    frame   xerrors.Frame
}

func (m *MyError2) Error() string {
    return m.Message
}

func (m *MyError2) Format(f fmt.State, c rune) { // implements fmt.Formatter
    xerrors.FormatError(m, f, c)
}

func (m *MyError2) FormatError(p xerrors.Printer) error { // implements
xerrors.Formatter
    p.Print(m.Message)
    if p.Detail() {
        m.frame.Format(p)
    }
    return nil
}

func ExampleFormatError() {
    err := &MyError2{Message: "oops", frame: xerrors.Caller(1)}
    fmt.Printf("%v\n", err)
    fmt.Println()
    fmt.Printf("%+v\n", err)
}
```

H:\Go\gopath\src\golang.org\x\xerrors\fmt.go:

```
// Copyright 2018 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package xerrors

import (
    "fmt"
    "strings"
    "unicode"
```

```
    "unicode/utf8"

    "golang.org/x/xerrors/internal"
)

const percentBangString = "%!"

// Errorf formats according to a format specifier and returns the string as
a
// value that satisfies error.
//
// The returned error includes the file and line number of the caller when
// formatted with additional detail enabled. If the last argument is an
error
// the returned error's Format method will return it if the format string
ends
// with ": %s", ": %v", or ": %w". If the last argument is an error and the
// format string ends with ": %w", the returned error implements an Unwrap
// method returning it.
//
// If the format specifier includes a %w verb with an error operand in a
// position other than at the end, the returned error will still implement
an
// Unwrap method returning the operand, but the error's Format method will
not
// return the wrapped error.
//
// It is invalid to include more than one %w verb or to supply it with an
// operand that does not implement the error interface. The %w verb is
otherwise
// a synonym for %v.
//
// Note that as of Go 1.13, the fmt.Errorf function will do error
formatting,
// but it will not capture a stack backtrace.
func Errorf(format string, a ...interface{}) error {
    format = formatPlusW(format)
    // Support a ": %[wsv]" suffix, which works well with xerrors.Formatter.
    wrap := strings.HasSuffix(format, ": %w")
    idx, format2, ok := parsePercentW(format)
    percentWElsewhere := !wrap && idx >= 0
    if !percentWElsewhere && (wrap || strings.HasSuffix(format, ": %s") ||
strings.HasSuffix(format, ": %v")) {
        err := errorAt(a, len(a)-1)
        if err == nil {
            return &noWrapError{fmt.Sprintf(format, a...), nil, Caller(1)}
        }
        // TODO: this is not entirely correct. The error value could be
        // printed elsewhere in format if it mixes numbered with unnumbered
        // substitutions. With relatively small changes to doPrintf we can
        // have it optionally ignore extra arguments and pass the argument
        // list in its entirety.
        msg := fmt.Sprintf(format[:len(format)-len(": %s")],
a[:len(a)-1]...)
```

```
        frame := Frame{}
        if internal.EnableTrace {
            frame = Caller(1)
        }
        if wrap {
            return &wrapError{msg, err, frame}
        }
        return &noWrapError{msg, err, frame}
    }
    // Support %w anywhere.
    // TODO: don't repeat the wrapped error's message when %w occurs in the
middle.
    msg := fmt.Sprintf(format2, a...)
    if idx < 0 {
        return &noWrapError{msg, nil, Caller(1)}
    }
    err := errorAt(a, idx)
    if !ok || err == nil {
        // Too many %ws or argument of %w is not an error. Approximate the
Go
        // 1.13 fmt.Errorf message.
        return &noWrapError{fmt.Sprintf("%sw(%s)", percentBangString, msg),
nil, Caller(1)}
    }
    frame := Frame{}
    if internal.EnableTrace {
        frame = Caller(1)
    }
    return &wrapError{msg, err, frame}
}

func errorAt(args []interface{}, i int) error {
    if i < 0 || i >= len(args) {
        return nil
    }
    err, ok := args[i].(error)
    if !ok {
        return nil
    }
    return err
}

// formatPlusW is used to avoid the vet check that will barf at %w.
func formatPlusW(s string) string {
    return s
}

// Return the index of the only %w in format, or -1 if none.
// Also return a rewritten format string with %w replaced by %v, and
// false if there is more than one %w.
// TODO: handle "%[N]w".
func parsePercentW(format string) (idx int, newFormat string, ok bool) {
    // Loosely copied from
golang.org/x/tools/go/analysis/passes/printf/printf.go.
```

```
    idx = -1
    ok = true
    n := 0
    sz := 0
    var isW bool
    for i := 0; i < len(format); i += sz {
        if format[i] != '%' {
            sz = 1
            continue
        }
        // "%%" is not a format directive.
        if i+1 < len(format) && format[i+1] == '%' {
            sz = 2
            continue
        }
        sz, isW = parsePrintfVerb(format[i:])
        if isW {
            if idx >= 0 {
                ok = false
            } else {
                idx = n
            }
            // "Replace" the last character, the 'w', with a 'v'.
            p := i + sz - 1
            format = format[:p] + "v" + format[p+1:]
        }
        n++
    }
    return idx, format, ok
}

// Parse the printf verb starting with a % at s[0].
// Return how many bytes it occupies and whether the verb is 'w'.
func parsePrintfVerb(s string) (int, bool) {
    // Assume only that the directive is a sequence of non-letters followed
by a single letter.
    sz := 0
    var r rune
    for i := 1; i < len(s); i += sz {
        r, sz = utf8.DecodeRuneInString(s[i:])
        if unicode.IsLetter(r) {
            return i + sz, r == 'w'
        }
    }
    return len(s), false
}

type noWrapError struct {
    msg   string
    err   error
    frame Frame
}

func (e *noWrapError) Error() string {
```

```
    return fmt.Sprint(e)
}

func (e *noWrapError) Format(s fmt.State, v rune) { FormatError(e, s, v) }

func (e *noWrapError) FormatError(p Printer) (next error) {
    p.Print(e.msg)
    e.frame.Format(p)
    return e.err
}

type wrapError struct {
    msg   string
    err   error
    frame Frame
}

func (e *wrapError) Error() string {
    return fmt.Sprint(e)
}

func (e *wrapError) Format(s fmt.State, v rune) { FormatError(e, s, v) }

func (e *wrapError) FormatError(p Printer) (next error) {
    p.Print(e.msg)
    e.frame.Format(p)
    return e.err
}

func (e *wrapError) Unwrap() error {
    return e.err
}
```

H:\Go\gopath\src\golang.org\x\xerrors\internal\internal.go:

```
// Copyright 2018 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package internal

// EnableTrace indicates whether stack information should be recorded in
errors.
var EnableTrace = true
```

H:\Go\gopath\src\golang.org\x\xerrors\codereview.cfg:

```
issuerepo: golang/go
```

H:\Go\gopath\src\golang.org\x\xerrors\example_test.go:

```
// Copyright 2012 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package xerrors_test

import (
    "fmt"
    "time"
)

// MyError is an error implementation that includes a time and message.
type MyError struct {
    When time.Time
    What string
}

func (e MyError) Error() string {
    return fmt.Sprintf("%v: %v", e.When, e.What)
}

func oops() error {
    return MyError{
        time.Date(1989, 3, 15, 22, 30, 0, 0, time.UTC),
        "the file system has gone away",
    }
}

func Example() {
    if err := oops(); err != nil {
        fmt.Println(err)
    }
    // Output: 1989-03-15 22:30:00 +0000 UTC: the file system has gone away
}
```

H:\Go\gopath\src\golang.org\x\xerrors\stack_test.go:

```
// Copyright 2018 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package xerrors_test

import (
    "bytes"
    "fmt"
    "math/big"
    "testing"
```

```
    "golang.org/x/xerrors"
    "golang.org/x/xerrors/internal"
)

type myType struct{}

func (myType) Format(s fmt.State, v rune) {
    s.Write(bytes.Repeat([]byte("Hi! "), 10))
}

func BenchmarkErrorf(b *testing.B) {
    err := xerrors.New("foo")
    // pi := big.NewFloat(3.14) // Something expensive.
    num := big.NewInt(5)
    args := func(a ...interface{}) []interface{} { return a }
    benchCases := []struct {
        name   string
        format string
        args   []interface{}
    }{
        {"no_format", "msg: %v", args(err)},
        {"with_format", "failed %d times: %v", args(5, err)},
        {"method: mytype", "pi: %v", args("myfile.go", myType{}, err)},
        {"method: number", "pi: %v", args("myfile.go", num, err)},
    }
    for _, bc := range benchCases {
        b.Run(bc.name, func(b *testing.B) {
            b.Run("ExpWithTrace", func(b *testing.B) {
                for i := 0; i < b.N; i++ {
                    xerrors.Errorf(bc.format, bc.args...)
                }
            })
            b.Run("ExpNoTrace", func(b *testing.B) {
                internal.EnableTrace = false
                defer func() { internal.EnableTrace = true }()

                for i := 0; i < b.N; i++ {
                    xerrors.Errorf(bc.format, bc.args...)
                }
            })
            b.Run("Core", func(b *testing.B) {
                for i := 0; i < b.N; i++ {
                    fmt.Errorf(bc.format, bc.args...)
                }
            })
        })
    }
}
```

H:\Go\gopath\src\golang.org\x\xerrors\wrap.go:

```
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package xerrors

import (
    "reflect"
)

// A Wrapper provides context around another error.
type Wrapper interface {
    // Unwrap returns the next error in the error chain.
    // If there is no next error, Unwrap returns nil.
    Unwrap() error
}

// Opaque returns an error with the same error formatting as err
// but that does not match err and cannot be unwrapped.
func Opaque(err error) error {
    return noWrapper{err}
}

type noWrapper struct {
    error
}

func (e noWrapper) FormatError(p Printer) (next error) {
    if f, ok := e.error.(Formatter); ok {
        return f.FormatError(p)
    }
    p.Print(e.error)
    return nil
}

// Unwrap returns the result of calling the Unwrap method on err, if err
implements
// Unwrap. Otherwise, Unwrap returns nil.
//
// Deprecated: As of Go 1.13, use errors.Unwrap instead.
func Unwrap(err error) error {
    u, ok := err.(Wrapper)
    if !ok {
        return nil
    }
    return u.Unwrap()
}

// Is reports whether any error in err's chain matches target.
//
// An error is considered to match a target if it is equal to that target or
if
// it implements a method Is(error) bool such that Is(target) returns true.
//
// Deprecated: As of Go 1.13, use errors.Is instead.
```

```
func Is(err, target error) bool {
    if target == nil {
        return err == target
    }

    isComparable := reflect.TypeOf(target).Comparable()
    for {
        if isComparable && err == target {
            return true
        }
        if x, ok := err.(interface{ Is(error) bool }); ok && x.Is(target) {
            return true
        }
        // TODO: consider supporing target.Is(err). This would allow
        // user-definable predicates, but also may allow for coping with
sloppy
        // APIs, thereby making it easier to get away with them.
        if err = Unwrap(err); err == nil {
            return false
        }
    }
}

// As finds the first error in err's chain that matches the type to which
target
// points, and if so, sets the target to its value and returns true. An
error
// matches a type if it is assignable to the target type, or if it has a
method
// As(interface{}) bool such that As(target) returns true. As will panic if
target
// is not a non-nil pointer to a type which implements error or is of
interface type.
//
// The As method should set the target to its value and return true if err
// matches the type to which target points.
//
// Deprecated: As of Go 1.13, use errors.As instead.
func As(err error, target interface{}) bool {
    if target == nil {
        panic("errors: target cannot be nil")
    }
    val := reflect.ValueOf(target)
    typ := val.Type()
    if typ.Kind() != reflect.Ptr || val.IsNil() {
        panic("errors: target must be a non-nil pointer")
    }
    if e := typ.Elem(); e.Kind() != reflect.Interface &&
!e.Implements(errorType) {
        panic("errors: *target must be interface or implement error")
    }
    targetType := typ.Elem()
    for err != nil {
        if reflect.TypeOf(err).AssignableTo(targetType) {
```

```
            val.Elem().Set(reflect.ValueOf(err))
            return true
        }
        if x, ok := err.(interface{ As(interface{}) bool }); ok &&
x.As(target) {
            return true
        }
        err = Unwrap(err)
    }
    return false
}

var errorType = reflect.TypeOf((*error)(nil)).Elem()
```

H:\Go\gopath\src\golang.org\x\xerrors\adaptor.go:

```
// Copyright 2018 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package xerrors

import (
    "bytes"
    "fmt"
    "io"
    "reflect"
    "strconv"
)

// FormatError calls the FormatError method of f with an errors.Printer
// configured according to s and verb, and writes the result to s.
func FormatError(f Formatter, s fmt.State, verb rune) {
    // Assuming this function is only called from the Format method, and
given
    // that FormatError takes precedence over Format, it cannot be called
from
    // any package that supports errors.Formatter. It is therefore safe to
    // disregard that State may be a specific printer implementation and use
one
    // of our choice instead.

    // limitations: does not support printing error as Go struct.

    var (
        sep    = " " // separator before next error
        p      = &state{State: s}
        direct = true
    )

    var err error = f
```

```
    switch verb {
    // Note that this switch must match the preference order
    // for ordinary string printing (%#v before %+v, and so on).

    case 'v':
        if s.Flag('#') {
            if stringer, ok := err.(fmt.GoStringer); ok {
                io.WriteString(&p.buf, stringer.GoString())
                goto exit
            }
            // proceed as if it were %v
        } else if s.Flag('+') {
            p.printDetail = true
            sep = "\n  - "
        }
    case 's':
    case 'q', 'x', 'X':
        // Use an intermediate buffer in the rare cases that precision,
        // truncation, or one of the alternative verbs (q, x, and X) are
        // specified.
        direct = false

    default:
        p.buf.WriteString("%!")
        p.buf.WriteRune(verb)
        p.buf.WriteByte('(')
        switch {
        case err != nil:
            p.buf.WriteString(reflect.TypeOf(f).String())
        default:
            p.buf.WriteString("<nil>")
        }
        p.buf.WriteByte(')')
        io.Copy(s, &p.buf)
        return
    }

loop:
    for {
        switch v := err.(type) {
        case Formatter:
            err = v.FormatError((*printer)(p))
        case fmt.Formatter:
            v.Format(p, 'v')
            break loop
        default:
            io.WriteString(&p.buf, v.Error())
            break loop
        }
        if err == nil {
            break
        }
        if p.needColon || !p.printDetail {
            p.buf.WriteByte(':')
```

```
            p.needColon = false
        }
        p.buf.WriteString(sep)
        p.inDetail = false
        p.needNewline = false
    }

exit:
    width, okW := s.Width()
    prec, okP := s.Precision()

    if !direct || (okW && width > 0) || okP {
        // Construct format string from State s.
        format := []byte{'%'}
        if s.Flag('-') {
            format = append(format, '-')
        }
        if s.Flag('+') {
            format = append(format, '+')
        }
        if s.Flag(' ') {
            format = append(format, ' ')
        }
        if okW {
            format = strconv.AppendInt(format, int64(width), 10)
        }
        if okP {
            format = append(format, '.')
            format = strconv.AppendInt(format, int64(prec), 10)
        }
        format = append(format, string(verb)...)
        fmt.Fprintf(s, string(format), p.buf.String())
    } else {
        io.Copy(s, &p.buf)
    }
}

var detailSep = []byte("\n    ")

// state tracks error printing state. It implements fmt.State.
type state struct {
    fmt.State
    buf bytes.Buffer

    printDetail bool
    inDetail    bool
    needColon   bool
    needNewline bool
}

func (s *state) Write(b []byte) (n int, err error) {
    if s.printDetail {
        if len(b) == 0 {
            return 0, nil
```

```
        }
        if s.inDetail && s.needColon {
            s.needNewline = true
            if b[0] == '\n' {
                b = b[1:]
            }
        }
        k := 0
        for i, c := range b {
            if s.needNewline {
                if s.inDetail && s.needColon {
                    s.buf.WriteByte(':')
                    s.needColon = false
                }
                s.buf.Write(detailSep)
                s.needNewline = false
            }
            if c == '\n' {
                s.buf.Write(b[k:i])
                k = i + 1
                s.needNewline = true
            }
        }
        s.buf.Write(b[k:])
        if !s.inDetail {
            s.needColon = true
        }
    } else if !s.inDetail {
        s.buf.Write(b)
    }
    return len(b), nil
}

// printer wraps a state to implement an xerrors.Printer.
type printer state

func (s *printer) Print(args ...interface{}) {
    if !s.inDetail || s.printDetail {
        fmt.Fprint((*state)(s), args...)
    }
}

func (s *printer) Printf(format string, args ...interface{}) {
    if !s.inDetail || s.printDetail {
        fmt.Fprintf((*state)(s), format, args...)
    }
}

func (s *printer) Detail() bool {
    s.inDetail = true
    return s.printDetail
}
```

H:\Go\gopath\src\golang.org\x\xerrors\errors.go:

```
// Copyright 2011 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package xerrors

import "fmt"

// errorString is a trivial implementation of error.
type errorString struct {
    s     string
    frame Frame
}

// New returns an error that formats as the given text.
//
// The returned error contains a Frame set to the caller's location and
// implements Formatter to show this information when printed with details.
func New(text string) error {
    return &errorString{text, Caller(1)}
}

func (e *errorString) Error() string {
    return e.s
}

func (e *errorString) Format(s fmt.State, v rune) { FormatError(e, s, v) }

func (e *errorString) FormatError(p Printer) (next error) {
    p.Print(e.s)
    e.frame.Format(p)
    return nil
}
```

H:\Go\gopath\src\golang.org\x\xerrors\fmt_test.go:

```
// Copyright 2018 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package xerrors_test

import (
    "fmt"
    "io"
    "os"
    "path"
    "reflect"
    "regexp"
```

```
    "strconv"
    "strings"
    "testing"

    "golang.org/x/xerrors"
)

func TestErrorf(t *testing.T) {
    chained := &wrapped{"chained", nil}
    chain := func(s ...string) (a []string) {
        for _, s := range s {
            a = append(a, cleanPath(s))
        }
        return a
    }
    testCases := []struct {
        got  error
        want []string
    }{{
        xerrors.Errorf("no args"),
        chain("no args/path.TestErrorf/path.go:xxx"),
    }, {
        xerrors.Errorf("no args: %s"),
        chain("no args: %!s(MISSING)/path.TestErrorf/path.go:xxx"),
    }, {
        xerrors.Errorf("nounwrap: %s", "simple"),
        chain(`nounwrap: simple/path.TestErrorf/path.go:xxx`),
    }, {
        xerrors.Errorf("nounwrap: %v", "simple"),
        chain(`nounwrap: simple/path.TestErrorf/path.go:xxx`),
    }, {
        xerrors.Errorf("%s failed: %v", "foo", chained),
        chain("foo failed/path.TestErrorf/path.go:xxx",
            "chained/somefile.go:xxx"),
    }, {
        xerrors.Errorf("no wrap: %s", chained),
        chain("no wrap/path.TestErrorf/path.go:xxx",
            "chained/somefile.go:xxx"),
    }, {
        xerrors.Errorf("%s failed: %w", "foo", chained),
        chain("wraps:foo failed/path.TestErrorf/path.go:xxx",
            "chained/somefile.go:xxx"),
    }, {
        xerrors.Errorf("nowrapv: %v", chained),
        chain("nowrapv/path.TestErrorf/path.go:xxx",
            "chained/somefile.go:xxx"),
    }, {
        xerrors.Errorf("wrapw: %w", chained),
        chain("wraps:wrapw/path.TestErrorf/path.go:xxx",
            "chained/somefile.go:xxx"),
    }, {
        xerrors.Errorf("wrapw %w middle", chained),
        chain("wraps:wrapw chained middle/path.TestErrorf/path.go:xxx",
            "chained/somefile.go:xxx"),
```

```
    }, {
        xerrors.Errorf("not wrapped: %+v", chained),
        chain("not wrapped: chained:
somefile.go:123/path.TestErrorf/path.go:xxx"),
    }}
    for i, tc := range testCases {
        t.Run(strconv.Itoa(i)+"/"+path.Join(tc.want...), func(t *testing.T)
{
            got := errToParts(tc.got)
            if !reflect.DeepEqual(got, tc.want) {
                t.Errorf("Format:\n got: %#v\nwant: %#v", got, tc.want)
            }

            gotStr := tc.got.Error()
            wantStr := fmt.Sprint(tc.got)
            if gotStr != wantStr {
                t.Errorf("Error:\n got: %#v\nwant: %#v", got, tc.want)
            }
        })
    }
}

func TestErrorFormatter(t *testing.T) {
    var (
        simple   = &wrapped{"simple", nil}
        elephant = &wrapped{
            "can't adumbrate elephant",
            detailed{},
        }
        nonascii = &wrapped{"café", nil}
        newline  = &wrapped{"msg with\nnewline",
            &wrapped{"and another\none", nil}}
        fallback  = &wrapped{"fallback", os.ErrNotExist}
        oldAndNew = &wrapped{"new style", formatError("old style")}
        framed    = &withFrameAndMore{
            frame: xerrors.Caller(0),
        }
        opaque = &wrapped{"outer",
            xerrors.Opaque(&wrapped{"mid",
                &wrapped{"inner", nil}})}
    )
    testCases := []struct {
        err    error
        fmt    string
        want   string
        regexp bool
    }{{
        err:  simple,
        fmt:  "%s",
        want: "simple",
    }, {
        err:  elephant,
        fmt:  "%s",
        want: "can't adumbrate elephant: out of peanuts",
```

```
}, {
    err:  &wrapped{"a", &wrapped{"b", &wrapped{"c", nil}}},
    fmt:  "%s",
    want: "a: b: c",
}, {
    err: simple,
    fmt: "%+v",
    want: "simple:" +
        "\n    somefile.go:123",
}, {
    err: elephant,
    fmt: "%+v",
    want: "can't adumbrate elephant:" +
        "\n    somefile.go:123" +
        "\n  - out of peanuts:" +
        "\n    the elephant is on strike" +
        "\n    and the 12 monkeys" +
        "\n    are laughing",
}, {
    err:  &oneNewline{nil},
    fmt:  "%+v",
    want: "123",
}, {
    err: &oneNewline{&oneNewline{nil}},
    fmt: "%+v",
    want: "123:" +
        "\n  - 123",
}, {
    err:  &newlineAtEnd{nil},
    fmt:  "%+v",
    want: "newlineAtEnd:\n    detail",
}, {
    err: &newlineAtEnd{&newlineAtEnd{nil}},
    fmt: "%+v",
    want: "newlineAtEnd:" +
        "\n    detail" +
        "\n  - newlineAtEnd:" +
        "\n    detail",
}, {
    err: framed,
    fmt: "%+v",
    want: "something:" +
        "\n    golang.org/x/xerrors_test.TestErrorFormatter" +
        "\n        .+/fmt_test.go:101" +
        "\n    something more",
    regexp: true,
}, {
    err:  fmtTwice("Hello World!"),
    fmt:  "%#v",
    want: "2 times Hello World!",
}, {
    err:  fallback,
    fmt:  "%s",
    want: "fallback: file does not exist",
```

```
}, {
    err: fallback,
    fmt: "%+v",
    // Note: no colon after the last error, as there are no details.
    want: "fallback:" +
        "\n    somefile.go:123" +
        "\n  - file does not exist",
}, {
    err:  opaque,
    fmt:  "%s",
    want: "outer: mid: inner",
}, {
    err: opaque,
    fmt: "%+v",
    want: "outer:" +
        "\n    somefile.go:123" +
        "\n  - mid:" +
        "\n    somefile.go:123" +
        "\n  - inner:" +
        "\n    somefile.go:123",
}, {
    err:  oldAndNew,
    fmt:  "%v",
    want: "new style: old style",
}, {
    err:  oldAndNew,
    fmt:  "%q",
    want: `"new style: old style"`,
}, {
    err: oldAndNew,
    fmt: "%+v",
    // Note the extra indentation.
    // Colon for old style error is rendered by the fmt.Formatter
    // implementation of the old-style error.
    want: "new style:" +
        "\n    somefile.go:123" +
        "\n  - old style:" +
        "\n    otherfile.go:456",
}, {
    err:  simple,
    fmt:  "%-12s",
    want: "simple      ",
}, {
    // Don't use formatting flags for detailed view.
    err: simple,
    fmt: "%+12v",
    want: "simple:" +
        "\n    somefile.go:123",
}, {
    err:  elephant,
    fmt:  "%+50s",
    want: "          can't adumbrate elephant: out of peanuts",
}, {
    err:  nonascii,
```

```
    fmt:  "%q",
    want: `"caf é "`,
}, {
    err:  nonascii,
    fmt:  "%+q",
    want: `"caf\u00e9"`,
}, {
    err:  simple,
    fmt:  "% x",
    want: "73 69 6d 70 6c 65",
}, {
    err: newline,
    fmt: "%s",
    want: "msg with" +
        "\nnewline: and another" +
        "\none",
}, {
    err: newline,
    fmt: "%+v",
    want: "msg with" +
        "\n    newline:" +
        "\n    somefile.go:123" +
        "\n  - and another" +
        "\n    one:" +
        "\n    somefile.go:123",
}, {
    err: &wrapped{"", &wrapped{"inner message", nil}},
    fmt: "%+v",
    want: "somefile.go:123" +
        "\n  - inner message:" +
        "\n    somefile.go:123",
}, {
    err:  spurious(""),
    fmt:  "%s",
    want: "spurious",
}, {
    err:  spurious(""),
    fmt:  "%+v",
    want: "spurious",
}, {
    err:  spurious("extra"),
    fmt:  "%s",
    want: "spurious",
}, {
    err: spurious("extra"),
    fmt: "%+v",
    want: "spurious:\n" +
        "    extra",
}, {
    err:  nil,
    fmt:  "%+v",
    want: "<nil>",
}, {
    err:  (*wrapped)(nil),
```

```
        fmt:  "%+v",
        want: "<nil>",
    }, {
        err:  simple,
        fmt:  "%T",
        want: "*xerrors_test.wrapped",
    }, {
        err:  simple,
        fmt:  "%",
        want: "%!(*xerrors_test.wrapped)",
        // For 1.13:
        //  want: "%!(*xerrors_test.wrapped=&{simple <nil>})",
    }, {
        err:  formatError("use fmt.Formatter"),
        fmt:  "%#v",
        want: "use fmt.Formatter",
    }, {
        err: fmtTwice("%s %s", "ok", panicValue{}),
        fmt: "%s",
        // Different Go versions produce different results.
        want:   `ok %!s\(PANIC=(String method: )?panic\)/ok
%!s\(PANIC=(String method: )?panic\)`,
        regexp: true,
    }, {
        err:  fmtTwice("%o %s", panicValue{}, "ok"),
        fmt:  "%s",
        want: "{} ok/{} ok",
    }, {
        err: adapted{"adapted", nil},
        fmt: "%+v",
        want: "adapted:" +
            "\n    detail",
    }, {
        err: adapted{"outer", adapted{"mid", adapted{"inner", nil}}},
        fmt: "%+v",
        want: "outer:" +
            "\n    detail" +
            "\n  - mid:" +
            "\n    detail" +
            "\n  - inner:" +
            "\n    detail",
    }}
    for i, tc := range testCases {
        t.Run(fmt.Sprintf("%d/%s", i, tc.fmt), func(t *testing.T) {
            got := fmt.Sprintf(tc.fmt, tc.err)
            var ok bool
            if tc.regexp {
                var err error
                ok, err = regexp.MatchString(tc.want+"$", got)
                if err != nil {
                    t.Fatal(err)
                }
            } else {
                ok = got == tc.want
```

```
            }
            if !ok {
                t.Errorf("\n got: %q\nwant: %q", got, tc.want)
            }
        })
    }
}

func TestAdaptor(t *testing.T) {
    testCases := []struct {
        err    error
        fmt    string
        want   string
        regexp bool
    }{{
        err: adapted{"adapted", nil},
        fmt: "%+v",
        want: "adapted:" +
            "\n    detail",
    }, {
        err: adapted{"outer", adapted{"mid", adapted{"inner", nil}}},
        fmt: "%+v",
        want: "outer:" +
            "\n    detail" +
            "\n  - mid:" +
            "\n    detail" +
            "\n  - inner:" +
            "\n    detail",
    }}
    for i, tc := range testCases {
        t.Run(fmt.Sprintf("%d/%s", i, tc.fmt), func(t *testing.T) {
            got := fmt.Sprintf(tc.fmt, tc.err)
            if got != tc.want {
                t.Errorf("\n got: %q\nwant: %q", got, tc.want)
            }
        })
    }
}

var _ xerrors.Formatter = wrapped{}

type wrapped struct {
    msg string
    err error
}

func (e wrapped) Error() string { return "should call Format" }

func (e wrapped) Format(s fmt.State, verb rune) {
    xerrors.FormatError(&e, s, verb)
}

func (e wrapped) FormatError(p xerrors.Printer) (next error) {
    p.Print(e.msg)
```

```
    p.Detail()
    p.Print("somefile.go:123")
    return e.err
}

var _ xerrors.Formatter = detailed{}

type detailed struct{}

func (e detailed) Error() string { panic("should have called FormatError") }

func (detailed) FormatError(p xerrors.Printer) (next error) {
    p.Printf("out of %s", "peanuts")
    p.Detail()
    p.Print("the elephant is on strike\n")
    p.Printf("and the %d monkeys\nare laughing", 12)
    return nil
}

type withFrameAndMore struct {
    frame xerrors.Frame
}

func (e *withFrameAndMore) Error() string { return fmt.Sprint(e) }

func (e *withFrameAndMore) Format(s fmt.State, v rune) {
    xerrors.FormatError(e, s, v)
}

func (e *withFrameAndMore) FormatError(p xerrors.Printer) (next error) {
    p.Print("something")
    if p.Detail() {
        e.frame.Format(p)
        p.Print("something more")
    }
    return nil
}

type spurious string

func (e spurious) Error() string { return fmt.Sprint(e) }

// move to 1_12 test file
func (e spurious) Format(s fmt.State, verb rune) {
    xerrors.FormatError(e, s, verb)
}

func (e spurious) FormatError(p xerrors.Printer) (next error) {
    p.Print("spurious")
    p.Detail() // Call detail even if we don't print anything
    if e == "" {
        p.Print()
    } else {
        p.Print("\n", string(e)) // print extraneous leading newline
```

```
    }
    return nil
}

type oneNewline struct {
    next error
}

func (e *oneNewline) Error() string { return fmt.Sprint(e) }

func (e *oneNewline) Format(s fmt.State, verb rune) {
    xerrors.FormatError(e, s, verb)
}

func (e *oneNewline) FormatError(p xerrors.Printer) (next error) {
    p.Print("1")
    p.Print("2")
    p.Print("3")
    p.Detail()
    p.Print("\n")
    return e.next
}

type newlineAtEnd struct {
    next error
}

func (e *newlineAtEnd) Error() string { return fmt.Sprint(e) }

func (e *newlineAtEnd) Format(s fmt.State, verb rune) {
    xerrors.FormatError(e, s, verb)
}

func (e *newlineAtEnd) FormatError(p xerrors.Printer) (next error) {
    p.Print("newlineAtEnd")
    p.Detail()
    p.Print("detail\n")
    return e.next
}

type adapted struct {
    msg string
    err error
}

func (e adapted) Error() string { return e.msg }

func (e adapted) Format(s fmt.State, verb rune) {
    xerrors.FormatError(e, s, verb)
}

func (e adapted) FormatError(p xerrors.Printer) error {
    p.Print(e.msg)
    p.Detail()
```

```
    p.Print("detail")
    return e.err
}

// formatError is an error implementing Format instead of xerrors.Formatter.
// The implementation mimics the implementation of github.com/pkg/errors.
type formatError string

func (e formatError) Error() string { return string(e) }

func (e formatError) Format(s fmt.State, verb rune) {
    // Body based on pkg/errors/errors.go
    switch verb {
    case 'v':
        if s.Flag('+') {
            io.WriteString(s, string(e))
            fmt.Fprintf(s, ":\n%s", "otherfile.go:456")
            return
        }
        fallthrough
    case 's':
        io.WriteString(s, string(e))
    case 'q':
        fmt.Fprintf(s, "%q", string(e))
    }
}

func (e formatError) GoString() string {
    panic("should never be called")
}

type fmtTwiceErr struct {
    format string
    args   []interface{}
}

func fmtTwice(format string, a ...interface{}) error {
    return fmtTwiceErr{format, a}
}

func (e fmtTwiceErr) Error() string { return fmt.Sprint(e) }

func (e fmtTwiceErr) Format(s fmt.State, verb rune) {
    xerrors.FormatError(e, s, verb)
}

func (e fmtTwiceErr) FormatError(p xerrors.Printer) (next error) {
    p.Printf(e.format, e.args...)
    p.Print("/")
    p.Printf(e.format, e.args...)
    return nil
}

func (e fmtTwiceErr) GoString() string {
```

```
    return "2 times " + fmt.Sprintf(e.format, e.args...)
}

type panicValue struct{}

func (panicValue) String() string { panic("panic") }

var rePath = regexp.MustCompile(`( [^ ]*)xerrors.*test\.`)
var reLine = regexp.MustCompile(":[0-9]*\n?$")

func cleanPath(s string) string {
    s = rePath.ReplaceAllString(s, "/path.")
    s = reLine.ReplaceAllString(s, ":xxx")
    s = strings.Replace(s, "\n    ", "", -1)
    s = strings.Replace(s, " /", "/", -1)
    return s
}

func errToParts(err error) (a []string) {
    for err != nil {
        var p testPrinter
        if xerrors.Unwrap(err) != nil {
            p.str += "wraps:"
        }
        f, ok := err.(xerrors.Formatter)
        if !ok {
            a = append(a, err.Error())
            break
        }
        err = f.FormatError(&p)
        a = append(a, cleanPath(p.str))
    }
    return a

}

type testPrinter struct {
    str string
}

func (p *testPrinter) Print(a ...interface{}) {
    p.str += fmt.Sprint(a...)
}

func (p *testPrinter) Printf(format string, a ...interface{}) {
    p.str += fmt.Sprintf(format, a...)
}

func (p *testPrinter) Detail() bool {
    p.str += " /"
    return true
}
```

H:\Go\gopath\src\golang.org\x\xerrors\wrap_113_test.go:

```
// Copyright 2019 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

//go:build go1.13
// +build go1.13

package xerrors_test

import (
    "errors"
    "testing"

    "golang.org/x/xerrors"
)

func TestErrorsIs(t *testing.T) {
    var errSentinel = errors.New("sentinel")

    got := errors.Is(xerrors.Errorf("%w", errSentinel), errSentinel)
    if !got {
        t.Error("got false, want true")
    }

    got = errors.Is(xerrors.Errorf("%w: %s", errSentinel, "foo"),
errSentinel)
    if !got {
        t.Error("got false, want true")
    }
}
```