Table of Contents

Panic

The `panic` and `recover` functions behave similarly to exceptions and try/catch in some other languages in that a `panic` causes the program stack to begin unwinding and `recover` can stop it. Deferred functions are still executed as the stack unwinds. If `recover` is called inside such a deferred function, the stack stops unwinding and `recover` returns the value (as an `interface{}`) that was passed to `panic`. The runtime will also panic in extraordinary circumstances, such as indexing an array or slice out-of-bounds. If a `panic` causes the stack to unwind outside of any executing goroutine (e.g. `main` or the top-level function given to `go` fail to recover from it), the program exits with a stack trace of all executing goroutines. A `panic` cannot be `recover`ed by a different goroutine.

Usage in a Package

By convention, no explicit `panic()` should be allowed to cross a package boundary. Indicating error conditions to callers should be done by returning error value. Within a package, however, especially if there are deeply nested calls to non-exported functions, it can be useful (and improve readability) to use panic to indicate error conditions which should be translated into error for the calling function. Below is an admittedly contrived example of a way in which a nested function and an exported function may interact via this panic-on-error relationship.

```
// A ParseError indicates an error in converting a word into an integer.
type ParseError struct {
    Index int    // The index into the space-separated list of words.
    Word  string // The word that generated the parse error.
    Error error  // The raw error that precipitated this error, if any.
}

// String returns a human-readable error message.
func (e *ParseError) String() string {
    return fmt.Sprintf("pkg: error parsing %q as int", e.Word)
}

// Parse parses the space-separated words in input as integers.
func Parse(input string) (numbers []int, err error) {
    defer func() {
```

```
        if r := recover(); r != nil {
            var ok bool
            err, ok = r.(error)
            if !ok {
                err = fmt.Errorf("pkg: %v", r)
            }
        }
    }()

    fields := strings.Fields(input)
    numbers = fields2numbers(fields)
    return
}

func fields2numbers(fields []string) (numbers []int) {
    if len(fields) == 0 {
        panic("no words to parse")
    }
    for idx, field := range fields {
        num, err := strconv.Atoi(field)
        if err != nil {
            panic(&ParseError{idx, field, err})
        }
        numbers = append(numbers, num)
    }
    return
}
```

To demonstrate the behavior, consider the following main function:

```
func main() {
    var examples = []string{
        "1 2 3 4 5",
        "100 50 25 12.5 6.25",
        "2 + 2 = 4",
        "1st class",
        "",
    }

    for _, ex := range examples {
        fmt.Printf("Parsing %q:\n  ", ex)
        nums, err := Parse(ex)
        if err != nil {
            fmt.Println(err)
            continue
        }
        fmt.Println(nums)
```

```
    }
}
```

References

Defer, Panic and Recover

https://go.dev/ref/spec#Handling_panics

https://go.dev/ref/spec#Run_time_panics