The **Go** Programming Language
Go
▽

Documents Packages The Project Help Blog Play Search

```
package main
```

Run Format Share

Package testing

```
import "testing"
```

Overview ▹

Overview ▾

Package **testing** provides support for automated **testing** of **Go** packages. It is intended to be used in concert with the “**go test**” command, which automates execution of any **function** of the form

```
func TestXxx(*testing.T)
```

where Xxx does not start with a lowercase letter. The **function** name serves to identify the **test** routine.

Within these functions, use the Error, Fail or related methods to signal failure.

To write a new **test** suite, create a file whose name ends _test.**go** that contains the TestXxx functions as described here. Put the file in the same package as the one being tested. The file will be excluded from regular package builds but will be included when the “**go test**” command is run. For more detail, run “**go** help **test**” and “**go** help testflag”.

Tests and benchmarks may be skipped if not applicable with a call to the Skip method of *T and *B:

```
func TestTimeConsuming(t *testing.T) {
    if testing.Short() {
        t.Skip("skipping test in short mode.")
    }
    ...
}
```

Benchmarks

Functions of the form

```
func BenchmarkXxx(*testing.B)
```

are considered benchmarks, and are executed by the "**go test**" command when its -bench flag is provided. Benchmarks are run sequentially.

For a description of the **testing** flags, see https://golang.org/cmd/**go**/#hdr-Description_of_**testing**_flags.

A sample benchmark **function** looks like this:

```
func BenchmarkHello(b *testing.B) {
    for i := 0; i < b.N; i++ {
        fmt.Sprintf("hello")
    }
}
```

The benchmark **function** must run the target code b.N times. During benchmark execution, b.N is adjusted until the benchmark **function** lasts long enough to be timed reliably. The output

```
BenchmarkHello    10000000    282 ns/op
```

means that the loop ran 10000000 times at a speed of 282 ns per loop.

If a benchmark needs some expensive setup before running, the timer may be reset:

```
func BenchmarkBigLen(b *testing.B) {
    big := NewBig()
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        big.Len()
    }
}
```

If a benchmark needs to **test** performance in a parallel setting, it may use the RunParallel helper **function**; such benchmarks are intended to be used with the **go test** -cpu flag:

```
func BenchmarkTemplateParallel(b *testing.B) {
    templ := template.Must(template.New("test").Parse("Hello, {{.}}!"))
    b.RunParallel(func(pb *testing.PB) {
        var buf bytes.Buffer
        for pb.Next() {
            buf.Reset()
            templ.Execute(&buf, "World")
        }
    })
}
```

Examples

The package also runs and verifies example code. Example functions may include a concluding line comment that begins with "Output:" and is compared with the standard output of the **function** when the tests are run. (The comparison ignores leading and trailing space.) These are examples of an example:

```
func ExampleHello() {
    fmt.Println("hello")
    // Output: hello
}

func ExampleSalutations() {
    fmt.Println("hello, and")
    fmt.Println("goodbye")
    // Output:
    // hello, and
    // goodbye
}
```

The comment prefix "Unordered output:" is like "Output:", but matches any line order:

```
func ExamplePerm() {
    for _, value := range Perm(4) {
        fmt.Println(value)
    }
    // Unordered output: 4
    // 2
    // 1
    // 3
    // 0
}
```

Example functions without output comments are compiled but not executed.

The naming convention to declare examples for the package, a **function** F, a type T and method M on type T are:

```
func Example() { ... }
func ExampleF() { ... }
func ExampleT() { ... }
func ExampleT_M() { ... }
```

Multiple example functions for a package/type/**function**/method may be provided by appending a distinct suffix to the name. The suffix must start with a lower-case letter.

```
func Example_suffix() { ... }
func ExampleF_suffix() { ... }
func ExampleT_suffix() { ... }
func ExampleT_M_suffix() { ... }
```

The entire **test** file is presented as the example when it contains a single example **function**, at least one other **function**, type, variable, or constant declaration, and no **test** or benchmark functions.

Subtests and Sub-benchmarks

The Run methods of T and B allow defining subtests and sub-benchmarks, without having to define separate functions for each. This enables uses like table-driven benchmarks and creating hierarchical tests. It also provides a way to share common setup and tear-down code:

```
func TestFoo(t *testing.T) {
    // <setup code>
    t.Run("A=1", func(t *testing.T) { ... })
    t.Run("A=2", func(t *testing.T) { ... })
    t.Run("B=1", func(t *testing.T) { ... })
    // <tear-down code>
}
```

Each subtest and sub-benchmark has a unique name: the combination of the name of the top-level **test** and the sequence of names passed to Run, separated by slashes, with an optional trailing sequence number for disambiguation.

The argument to the -run and -bench command-line flags is an unanchored regular expression that matches the **test**'s name. For tests with multiple slash-separated elements, such as subtests, the argument is itself slash-separated, with expressions matching each name element in turn. Because it is unanchored, an empty expression matches any string. For example, using "matching" to mean "whose name contains":

```
go test -run ''      # Run all tests.
go test -run Foo     # Run top-level tests matching "Foo", such as "TestFooBar".
go test -run Foo/A=  # For top-level tests matching "Foo", run subtests matching "A=".
go test -run /A=1    # For all top-level tests, run subtests matching "A=1".
```

Subtests can also be used to control parallelism. A parent **test** will only complete once all of its subtests complete. In this example, all tests are run in parallel with each other, and only with each other, regardless of other top-level tests that may be defined:

```
func TestGroupedParallel(t *testing.T) {
    for _, tc := range tests {
        tc := tc // capture range variable
        t.Run(tc.Name, func(t *testing.T) {
            t.Parallel()
            ...
        })
    }
}
```

Run does not return until parallel subtests have completed, providing a way to clean up after a group of parallel tests:

```
func TestTeardownParallel(t *testing.T) {
    // This Run will not return until the parallel tests finish.
    t.Run("group", func(t *testing.T) {
        t.Run("Test1", parallelTest1)
        t.Run("Test2", parallelTest2)
        t.Run("Test3", parallelTest3)
    })
    // <tear-down code>
}
```

Main

It is sometimes necessary for a **test** program to do extra setup or teardown before or after **testing**. It is also sometimes necessary for a **test** to control which code runs on the main thread. To support these and other cases, if a **test** file contains a **function**:

```
func TestMain(m *testing.M)
```

then the generated **test** will call TestMain(m) instead of running the tests directly. TestMain runs in the main goroutine and can do whatever setup and teardown is necessary around a call to m.Run. It should then call os.Exit with the result of m.Run. When TestMain is called, flag.Parse has not been run. If TestMain depends on command-line flags, including those of the **testing** package, it should call flag.Parse explicitly.

A simple implementation of TestMain is:

```
func TestMain(m *testing.M) {
        // call flag.Parse() here if TestMain uses flags
        os.Exit(m.Run())
}
```

Index ▹

Index ▾

Examples

B.RunParallel

Package files

allocs.go benchmark.go cover.go example.go match.go **testing.go**

func AllocsPerRun ¶

```
func AllocsPerRun(runs int, f func()) (avg float64)
```

AllocsPerRun returns the average number of allocations during calls to f. Although the return value has type float64, it will always be an integral value.

To compute the number of allocations, the **function** will first be run once as a warm-up. The average number of allocations over the specified number of runs will then be measured and returned.

AllocsPerRun sets GOMAXPROCS to 1 during its measurement and will restore it before returning.

func CoverMode ¶

```
func CoverMode() string
```

CoverMode reports what the **test** coverage mode is set to. The values are "set", "count", or "atomic". The return value will be empty if **test** coverage is not enabled.

func Coverage ¶

```
func Coverage() float64
```

Coverage reports the current code coverage as a fraction in the range [0, 1]. If coverage is not enabled, Coverage returns 0.

When running a large set of sequential **test** cases, checking Coverage after each one can be useful for identifying which **test** cases exercise new code paths. It is not a replacement for the reports generated by '**go test** -cover' and '**go** tool cover'.

func Main ¶

```
func Main(matchString func(pat, str string) (bool, error), tests []InternalTest, benchmarks []InternalBenchmark, examples []InternalExample)
```

Main is an internal **function**, part of the implementation of the "**go test**" command. It was exported because it is cross-package and predates "internal" packages. It is no longer used by "**go test**" but preserved, as much as possible, for other systems that simulate "**go test**" using Main, but Main sometimes cannot be updated as new functionality is added to the **testing** package. Systems simulating "**go test**" should be updated to use MainStart.

func RegisterCover ¶

```
func RegisterCover(c Cover)
```

RegisterCover records the coverage data accumulators for the tests. NOTE: This **function** is internal to the **testing** infrastructure and may change. It is not covered (yet) by the **Go** 1 compatibility guidelines.

func RunBenchmarks ¶

```
func RunBenchmarks(matchString func(pat, str string) (bool, error), benchmarks []InternalBenchmark)
```

An internal **function** but exported because it is cross-package; part of the implementation of the "**go test**" command.

func RunExamples ¶

```
func RunExamples(matchString func(pat, str string) (bool, error), examples []InternalExample) (ok bool)
```

An internal **function** but exported because it is cross-package; part of the implementation of the "**go test**" command.

func RunTests ¶

```
func RunTests(matchString func(pat, str string) (bool, error), tests []InternalTest) (ok bool)
```

An internal **function** but exported because it is cross-package; part of the implementation of the "**go test**" command.

func Short ¶

```
func Short() bool
```

Short reports whether the -**test**.short flag is set.

func Verbose ¶

```
func Verbose() bool
```

Verbose reports whether the -**test**.v flag is set.

type B ¶

B is a type passed to Benchmark functions to manage benchmark timing and to specify the number of iterations to run.

A benchmark ends when its Benchmark **function** returns or calls any of the methods FailNow, Fatal, Fatalf, SkipNow, Skip, or Skipf. Those methods must be called only from the goroutine running the Benchmark **function**. The other reporting methods, such as the variations of Log and Error, may be called simultaneously from multiple goroutines.

Like in tests, benchmark logs are accumulated during execution and dumped to standard error when done. Unlike in tests, benchmark logs are always printed, so as not to hide output whose existence may be affecting benchmark results.

```
type B struct {
        N int
        // contains filtered or unexported fields
}
```

func (*B) Error ¶

```
func (c *B) Error(args ...interface{})
```

Error is equivalent to Log followed by Fail.

func (*B) Errorf ¶

```
func (c *B) Errorf(format string, args ...interface{})
```

Errorf is equivalent to Logf followed by Fail.

func (*B) Fail ¶

```
func (c *B) Fail()
```

Fail marks the **function** as having failed but continues execution.

func (*B) FailNow ¶

```
func (c *B) FailNow()
```

FailNow marks the **function** as having failed and stops its execution by calling runtime.Goexit (which then runs all deferred calls in the current goroutine). Execution will continue at the next **test** or benchmark. FailNow must be called from the goroutine running the **test** or benchmark **function**, not from other goroutines created during the **test**. Calling FailNow does not stop those other goroutines.

func (*B) Failed ¶

```
func (c *B) Failed() bool
```

Failed reports whether the **function** has failed.

func (*B) Fatal ¶

```
func (c *B) Fatal(args ...interface{})
```

Fatal is equivalent to Log followed by FailNow.

func (*B) Fatalf ¶

```
func (c *B) Fatalf(format string, args ...interface{})
```

Fatalf is equivalent to Logf followed by FailNow.

func (*B) Helper ¶

```
func (c *B) Helper()
```

Helper marks the calling **function** as a **test** helper **function**. When printing file and line information, that **function** will be skipped. Helper may be called simultaneously from multiple goroutines. Helper has no effect if it is called directly from a TestXxx/BenchmarkXxx **function** or a subtest/sub-benchmark **function**.

func (*B) Log ¶

```
func (c *B) Log(args ...interface{})
```

Log formats its arguments using default formatting, analogous to Println, and records the text in the error log. For tests, the text will be printed only if the **test** fails or the -**test**.v flag is set. For benchmarks, the text is always printed to avoid having performance depend on the value of the -**test**.v flag.

func (*B) Logf ¶

```
func (c *B) Logf(format string, args ...interface{})
```

Logf formats its arguments according to the format, analogous to Printf, and records the text in the error log. A final newline is added if not provided. For tests, the text will be printed only if the **test** fails or the -**test**.v flag is set. For benchmarks, the text is always printed to avoid having performance depend on the value of the -**test**.v flag.

func (*B) Name ¶

```
func (c *B) Name() string
```

Name returns the name of the running **test** or benchmark.

func (*B) ReportAllocs ¶

```
func (b *B) ReportAllocs()
```

ReportAllocs enables malloc statistics for this benchmark. It is equivalent to setting -**test**.benchmem, but it only affects the benchmark **function** that calls ReportAllocs.

func (*B) ResetTimer ¶

```
func (b *B) ResetTimer()
```

ResetTimer zeros the elapsed benchmark time and memory allocation counters. It does not affect whether the timer is running.

func (*B) Run ¶

```
func (b *B) Run(name string, f func(b *B)) bool
```

Run benchmarks f as a subbenchmark with the given name. It reports whether there were any failures.

A subbenchmark is like any other benchmark. A benchmark that calls Run at least once will not be measured itself and will be called once with N=1.

func (*B) RunParallel ¶

```
func (b *B) RunParallel(body func(*PB))
```

RunParallel runs a benchmark in parallel. It creates multiple goroutines and distributes b.N iterations among them. The number of goroutines defaults to GOMAXPROCS. To increase parallelism for non-CPU-bound benchmarks, call SetParallelism before RunParallel. RunParallel is usually used with the **go test** -cpu flag.

The body **function** will be run in each goroutine. It should set up any goroutine-local state and then iterate until pb.Next returns false. It should not use the StartTimer, StopTimer, or ResetTimer functions, because they have global effect. It should also not call Run.

▹ Example

▾ Example

```
package main
```

Run Format Share

func (*B) SetBytes ¶

```
func (b *B) SetBytes(n int64)
```

SetBytes records the number of bytes processed in a single operation. If this is called, the benchmark will report ns/op and MB/s.

func (*B) SetParallelism ¶

```
func (b *B) SetParallelism(p int)
```

SetParallelism sets the number of goroutines used by RunParallel to p*GOMAXPROCS. There is usually no need to call SetParallelism for CPU-bound benchmarks. If p is less than 1, this call will have no effect.

func (*B) Skip ¶

```
func (c *B) Skip(args ...interface{})
```

Skip is equivalent to Log followed by SkipNow.

func (*B) SkipNow ¶

```
func (c *B) SkipNow()
```

SkipNow marks the **test** as having been skipped and stops its execution by calling runtime.Goexit. If a **test** fails (see Error, Errorf, Fail) and is then skipped, it is still considered to have failed. Execution will continue at the next **test** or benchmark. See also FailNow. SkipNow must be called from the goroutine running the **test**, not from other goroutines created during the **test**. Calling SkipNow does not stop those other goroutines.

func (*B) Skipf ¶

```
func (c *B) Skipf(format string, args ...interface{})
```

Skipf is equivalent to Logf followed by SkipNow.

func (*B) Skipped ¶

```
func (c *B) Skipped() bool
```

Skipped reports whether the **test** was skipped.

func (*B) StartTimer ¶

```
func (b *B) StartTimer()
```

StartTimer starts timing a **test**. This **function** is called automatically before a benchmark starts, but it can also used to resume timing after a call to StopTimer.

func (*B) StopTimer ¶

```
func (b *B) StopTimer()
```

StopTimer stops timing a **test**. This can be used to pause the timer while performing complex initialization that you don't want to measure.

type BenchmarkResult ¶

The results of a benchmark run.

```
type BenchmarkResult struct {
        N         int           // The number of iterations.
        T         time.Duration // The total time taken.
        Bytes     int64         // Bytes processed in one iteration.
        MemAllocs uint64        // The total number of memory allocations.
        MemBytes  uint64        // The total number of bytes allocated.
}
```

func Benchmark ¶

```
func Benchmark(f func(b *B)) BenchmarkResult
```

Benchmark benchmarks a single **function**. Useful for creating custom benchmarks that do not use the "**go test**" command.

If f calls Run, the result will be an estimate of running all its subbenchmarks that don't call Run in sequence in a single benchmark.

func (BenchmarkResult) AllocedBytesPerOp ¶

```
func (r BenchmarkResult) AllocedBytesPerOp() int64
```

AllocedBytesPerOp returns r.MemBytes / r.N.

func (BenchmarkResult) AllocsPerOp ¶

```
func (r BenchmarkResult) AllocsPerOp() int64
```

AllocsPerOp returns r.MemAllocs / r.N.

func (BenchmarkResult) MemString ¶

```
func (r BenchmarkResult) MemString() string
```

MemString returns r.AllocedBytesPerOp and r.AllocsPerOp in the same format as '**go test**'.

func (BenchmarkResult) NsPerOp ¶

```
func (r BenchmarkResult) NsPerOp() int64
```

func (BenchmarkResult) String ¶

```
func (r BenchmarkResult) String() string
```

type Cover ¶

Cover records information about **test** coverage checking. NOTE: This struct is internal to the **testing** infrastructure and may change. It is not covered (yet) by the **Go** 1 compatibility guidelines.

```
type Cover struct {
        Mode            string
        Counters        map[string][]uint32
        Blocks          map[string][]CoverBlock
        CoveredPackages string
}
```

type CoverBlock ¶

CoverBlock records the coverage data for a single basic block. NOTE: This struct is internal to the **testing** infrastructure and may change. It is not covered (yet) by the **Go** 1 compatibility guidelines.

```
type CoverBlock struct {
        Line0 uint32
        Col0  uint16
        Line1 uint32
        Col1  uint16
        Stmts uint16
}
```

type InternalBenchmark ¶

An internal type but exported because it is cross-package; part of the implementation of the "**go test**" command.

```
type InternalBenchmark struct {
        Name string
        F    func(b *B)
}
```

type InternalExample ¶

```
type InternalExample struct {
        Name      string
        F         func()
        Output    string
        Unordered bool
}
```

type InternalTest ¶

An internal type but exported because it is cross-package; part of the implementation of the "**go test**" command.

```
type InternalTest struct {
        Name string
        F    func(*T)
}
```

type M ¶

M is a type passed to a TestMain **function** to run the actual tests.

```
type M struct {
        // contains filtered or unexported fields
}
```

func MainStart ¶

```
func MainStart(deps testDeps, tests []InternalTest, benchmarks []InternalBenchmark, examples []InternalExample) *M
```

MainStart is meant for use by tests generated by '**go test**'. It is not meant to be called directly and is not subject to the **Go** 1 compatibility document. It may change signature from release to release.

func (*M) Run ¶

```
func (m *M) Run() int
```

Run runs the tests. It returns an exit code to pass to os.Exit.

type PB ¶

A PB is used by RunParallel for running parallel benchmarks.

```
type PB struct {
        // contains filtered or unexported fields
}
```

func (*PB) Next ¶

```
func (pb *PB) Next() bool
```

Next reports whether there are more iterations to execute.

type T ¶

T is a type passed to **Test** functions to manage **test** state and support formatted **test** logs. Logs are accumulated during execution and dumped to standard output when done.

A **test** ends when its **Test function** returns or calls any of the methods FailNow, Fatal, Fatalf, SkipNow, Skip, or Skipf. Those methods, as well as the Parallel method, must be called only from the goroutine running the **Test function**.

The other reporting methods, such as the variations of Log and Error, may be called simultaneously from multiple goroutines.

```
type T struct {
        // contains filtered or unexported fields
}
```

func (*T) Error ¶

```
func (c *T) Error(args ...interface{})
```

Error is equivalent to Log followed by Fail.

func (*T) Errorf ¶

```
func (c *T) Errorf(format string, args ...interface{})
```

Errorf is equivalent to Logf followed by Fail.

func (*T) Fail ¶

```
func (c *T) Fail()
```

Fail marks the **function** as having failed but continues execution.

func (*T) FailNow ¶

```
func (c *T) FailNow()
```

FailNow marks the **function** as having failed and stops its execution by calling runtime.Goexit (which then runs all deferred calls in the current goroutine). Execution will continue at the next **test** or benchmark. FailNow must be called from the goroutine running the **test** or benchmark **function**, not from other goroutines created during the **test**. Calling FailNow does not stop those other goroutines.

func (*T) Failed ¶

```
func (c *T) Failed() bool
```

Failed reports whether the **function** has failed.

func (*T) Fatal ¶

```
func (c *T) Fatal(args ...interface{})
```

Fatal is equivalent to Log followed by FailNow.

func (*T) Fatalf ¶

```
func (c *T) Fatalf(format string, args ...interface{})
```

Fatalf is equivalent to Logf followed by FailNow.

func (*T) Helper ¶

```
func (c *T) Helper()
```

Helper marks the calling **function** as a **test** helper **function**. When printing file and line information, that **function** will be skipped. Helper may be called simultaneously from multiple goroutines. Helper has no effect if it is called directly from a TestXxx/BenchmarkXxx **function** or a subtest/sub-benchmark **function**.

func (*T) Log ¶

```
func (c *T) Log(args ...interface{})
```

Log formats its arguments using default formatting, analogous to Println, and records the text in the error log. For tests, the text will be printed only if the **test** fails or the -**test**.v flag is set. For benchmarks, the text is always printed to avoid having performance depend on the value of the -**test**.v flag.

func (*T) Logf ¶

```
func (c *T) Logf(format string, args ...interface{})
```

Logf formats its arguments according to the format, analogous to Printf, and records the text in the error log. A final newline is added if not provided. For tests, the text will be printed only if the **test** fails or the -**test**.v flag is set. For benchmarks, the text is always printed to avoid having performance depend on the value of the -**test**.v flag.

func (*T) Name ¶

```
func (c *T) Name() string
```

Name returns the name of the running **test** or benchmark.

func (*T) Parallel ¶

```
func (t *T) Parallel()
```

Parallel signals that this **test** is to be run in parallel with (and only with) other parallel tests. When a **test** is run multiple times due to use of -**test**.count or -**test**.cpu, multiple instances of a single **test** never run in parallel with each other.

func (*T) Run ¶

```
func (t *T) Run(name string, f func(t *T)) bool
```

Run runs f as a subtest of t called name. It runs f in a separate goroutine and blocks until f returns or calls t.Parallel to become a parallel **test**. Run reports whether f succeeded (or at least did not fail before calling t.Parallel).

Run may be called simultaneously from multiple goroutines, but all such calls must return before the outer **test function** for t returns.

func (*T) Skip ¶

```
func (c *T) Skip(args ...interface{})
```

Skip is equivalent to Log followed by SkipNow.

func (*T) SkipNow ¶

```
func (c *T) SkipNow()
```

SkipNow marks the **test** as having been skipped and stops its execution by calling runtime.Goexit. If a **test** fails (see Error, Errorf, Fail) and is then skipped, it is still considered to have failed. Execution will continue at the next **test** or benchmark. See also FailNow. SkipNow must be called from the goroutine running the **test**, not from other goroutines created during the **test**. Calling SkipNow does not stop those other goroutines.

func (*T) Skipf ¶

```
func (c *T) Skipf(format string, args ...interface{})
```

Skipf is equivalent to Logf followed by SkipNow.

func (*T) Skipped ¶

```
func (c *T) Skipped() bool
```

Skipped reports whether the **test** was skipped.

type TB ¶

TB is the interface common to T and B.

```
type TB interface {
        Error(args ...interface{})
        Errorf(format string, args ...interface{})
        Fail()
        FailNow()
        Failed() bool
        Fatal(args ...interface{})
        Fatalf(format string, args ...interface{})
        Log(args ...interface{})
        Logf(format string, args ...interface{})
        Name() string
        Skip(args ...interface{})
        SkipNow()
        Skipf(format string, args ...interface{})
        Skipped() bool
        Helper()
        // contains filtered or unexported methods
}
```

Subdirectories

Name	Synopsis
..	
iotest	Package iotest implements Readers and Writers useful mainly for **testing**.
quick	Package quick implements utility functions to help with black box **testing**.

Build version go1.10.2.