我们重点探索一下go test的实现原理。

首先，我们会先从数据结构入手，查看测试是如何被组织起来的。其次，我们会关注测试的关键实现方法，尽量呈现源码并配以示例来了解其实现原理。

为了叙述方便，本节部分源码隐去了部分与话题无关的代码，更多的源码解释，可以通过源码注释来了解。

# testing.common公共类

## 简介

我们知道单元测试函数需要传递一个testing.T类型的参数，而性能测试函数需要传递一个testing.B类型的参数，该参数可用于控制测试的流程，比如标记测试失败等。

testing.T和testing.B属于testing包中的两个数据类型，该类型提供一系列的方法用于控制函数执行流程，考虑到二者有一定的相似性，所以Go实现时抽象出一个testing.common作为一个基础类型，而testing.T和testing.B则属于testing.common的扩展。

本节，我们重点看testing.common，通过其成员及方法，来了解其实现原理。

## 数据结构

// common holds the elements common between T and B and  
// captures common methods such as Errorf.  
type common struct {  
 mu sync.RWMutex // guards this group of fields  
 output []byte // Output generated by test or benchmark.  
 w io.Writer // For flushToParent.  
 ran bool // Test or benchmark (or one of its subtests) was executed.  
 failed bool // Test or benchmark has failed.  
 skipped bool // Test of benchmark has been skipped.  
 done bool // Test is finished and all subtests have completed.  
 helpers map[string]struct{} // functions to be skipped when writing file/line info  
  
 chatty bool // A copy of the chatty flag.  
 finished bool // Test function has completed.  
 hasSub int32 // written atomically  
 raceErrors int // number of races detected during test  
 runner string // function name of tRunner running the test  
  
 parent \*common  
 level int // Nesting depth of test or benchmark.  
 creator []uintptr // If level > 0, the stack trace at the point where the parent called t.Run.  
 name string // Name of test or benchmark.  
 start time.Time // Time test or benchmark started  
 duration time.Duration  
 barrier chan bool // To signal parallel subtests they may start.  
 signal chan bool // To signal a test is done.  
 sub []\*T // Queue of subtests to be run in parallel.  
}

### common.mu

读写锁，仅用于控制本数据内的成员访问。

### common.output

存储当前测试产生的日志，每产生一条日志则追加到该切片中，待测试结束后再一并输出。

### common.w

子测试执行结束需要把产生的日志输送到父测试中的output切片中，传递时需要考虑缩进等格式调整，通过w把日志传递到父测试。

### common.ran

仅表示是否已执行过。比如，跟据某个规范筛选测试，如果没有测试被匹配到的话，则common.ran为false，表示没有测试运行过。

### common.failed

如果当前测试执行失败，则置为true。

### common.skipped

标记当前测试是否已跳过。

### common.done

表示当前测试及其子测试已结束，此状态下再执行Fail()之类的方法标记测试状态会产生panic。

### common.helpers

标记当前为函数为help函数，其中打印的日志，在记录日志时不会显示其文件名及行号。

### common.chatty

对应命令行中的-v参数，默认为false，true则打印更多详细日志。

### common.finished

如果当前测试结束，则置为true。

### common.hasSub

标记当前测试是否包含子测试，当测试使用t.Run()方法启动子测试时，t.hasSub则置为1。

### common.raceErrors

竞态检测错误数。

### common.runner

执行当前测试的函数名。

### common.parent

如果当前测试为子测试，则置为父测试的指针。

### common.level

测试嵌套层数，比如创建子测试时，子测试嵌套层数就会加1。

### common.creator

测试函数调用栈。

### common.name

记录每个测试函数名，比如测试函数TestAdd(t \*testing.T), 其中t.name即“TestAdd”。 测试结束，打印测试结果会用到该成员。

### common.start

记录测试开始的时间。

### common.duration

记录测试所花费的时间。

### common.barrier

用于控制父测试和子测试执行的channel，如果测试为Parallel，则会阻塞等待父测试结束后再继续。

### common.signal

通知当前测试结束。

### common.sub

子测试列表。

## 成员方法

### common.Name()

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

func (c \*common) Name() string {

return c.name

}

该方法直接返回common结构体中存储的名称。

### common.Fail()

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

func (c \*common) Fail() {

if c.parent != nil {

c.parent.Fail()

}

c.mu.Lock()

defer c.mu.Unlock()

// c.done needs to be locked to synchronize checks to c.done in parent tests.

if c.done {

panic("Fail in goroutine after " + c.name + " has completed")

}

c.failed = true

}

Fail()方法会标记当前测试为失败，然后继续运行，并不会立即退出当前测试。如果是子测试，则除了标记当前测试结果外还通过c.parent.Fail()来标记父测试失败。

### common.FailNow()

func (c \*common) FailNow() {

c.Fail()

c.finished = true

runtime.Goexit()

}

FailNow()内部会调用Fail()标记测试失败，还会标记测试结束并退出当前测试协程。 可以简单的把一个测试理解为一个协程，FailNow()只会退出当前协程，并不会影响其他测试协程，但要保证在当前测试协程中调用FailNow()才有效，不可以在当前测试创建的协程中调用该方法。

### common.log()

func (c \*common) log(s string) {

c.mu.Lock()

defer c.mu.Unlock()

c.output = append(c.output, c.decorate(s)...)

}

common.log()为内部记录日志入口，日志会统一记录到common.output切片中，测试结束时再统一打印出来。 日志记录时会调用common.decorate()进行装饰，即加上文件名和行号，还会做一些其他格式化处理。 调用common.log()的方法，有Log()、Logf()、Error()、Errorf()、Fatal()、Fatalf()、Skip()、Skipf()等。

注意：单元测试中记录的日志只有在执行失败或指定了-v参数才会打印，否则不会打印。而在性能测试中则总是被打印出来，因为是否打印日志有可能影响性能测试结果。

### common.Log(args ...interface{})

func (c \*common) Log(args ...interface{}) {

c.log(fmt.Sprintln(args...))

}

common.Log()方法用于记录简单日志，通过fmt.Sprintln()方法生成日志字符串后记录。

### common.Logf(format string, args ...interface{})

func (c \*common) Logf(format string, args ...interface{}) {

c.log(fmt.Sprintf(format, args...))

}

common.Logf()方法用于格式化记录日志，通过fmt.Sprintf()生成字符串后记录。

### common.Error(args ...interface{})

// Error is equivalent to Log followed by Fail.func (c \*common) Error(args ...interface{}) {

c.log(fmt.Sprintln(args...))

c.Fail()

}

common.Error()方法等同于common.Log()+common.Fail()，即记录日志并标记失败，但测试继续进行。

### common.Errorf(format string, args ...interface{})

// Errorf is equivalent to Logf followed by Fail.func (c \*common) Errorf(format string, args ...interface{}) {

c.log(fmt.Sprintf(format, args...))

c.Fail()

}

common.Errorf()方法等同于common.Logf()+common.Fail()，即记录日志并标记失败，但测试继续进行。

### common.Fatal(args ...interface{})

// Fatal is equivalent to Log followed by FailNow.

func (c \*common) Fatal(args ...interface{}) {

c.log(fmt.Sprintln(args...))

c.FailNow()

}

common.Fatal()方法等同于common.Log()+common.FailNow()，即记录日志、标记失败并退出当前测试。

### common.Fatalf(format string, args ...interface{})

// Fatalf is equivalent to Logf followed by FailNow.

func (c \*common) Fatalf(format string, args ...interface{}) {

c.log(fmt.Sprintf(format, args...))

c.FailNow()

}

common.Fatalf()方法等同于common.Logf()+common.FailNow()，即记录日志、标记失败并退出当前测试。

### common.skip()

func (c \*common) skip() {

c.mu.Lock()

defer c.mu.Unlock()

c.skipped = true

}

common.skip()方法标记当前测试为已跳过状态，比如测试中检测到某种条件，不再继续测试。该函数仅标记测试跳过，与测试结果无关。测试结果仍然取决于common.failed。

### common.SkipNow()

func (c \*common) SkipNow() {

c.skip()

c.finished = true

runtime.Goexit()

}

common.SkipNow()方法标记测试跳过，并标记测试结束，最后退出当前测试。

### common.Skip(args ...interface{})

// Skip is equivalent to Log followed by SkipNow.

func (c \*common) Skip(args ...interface{}) {

c.log(fmt.Sprintln(args...))

c.SkipNow()

}

common.Skip()方法等同于common.Log()+common.SkipNow()。

### common.Skipf(format string, args ...interface{})

// Skipf is equivalent to Logf followed by SkipNow.

func (c \*common) Skipf(format string, args ...interface{}) {

c.log(fmt.Sprintf(format, args...))

c.SkipNow()

}

common.Skipf()方法等同于common.Logf() + common.SkipNow()。

### common.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.

func (c \*common) Helper() {

c.mu.Lock()

defer c.mu.Unlock()

if c.helpers == nil {

c.helpers = make(map[string]struct{})

}

c.helpers[callerName(1)] = struct{}{}

}

common.Helper()方法标记当前函数为help函数，所谓help函数，即其中打印的日志，不记录help函数的函数名及行号，而是记录上一层函数的函数名和行号。

# testing.TB 接口

TB接口，顾名思义，是testing.T(单元测试)和testing.B(性能测试)共用的接口。

TB接口通过在接口中定义一个名为private(）的私有方法，保证了即使用户实现了类似的接口，也不会跟testing.TB接口冲突。

其实，这些接口在testing.T和testing.B公共成员testing.common中已经实现。

## 接口定义

在src/testing/testing.go中定义了testing.TB接口：

// 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()  
  
 // A private method to prevent users implementing the  
 // interface and so future additions to it will not  
 // violate Go 1 compatibility.  
 private()  
}

其中对外接口需要testing.T和testing.B实现，但由于testing.T和testing.B都继承了testing.common，而testing.common已经实现了这些接口，所以testing.T和testing.B天然实现了TB接口。

其中私有接口private()用于控制该接口的唯一性，即便用户代码中某个类型实现了这些方法，由于无法实现这个私有接口，也不能被认为是实现了TB接口，所以不会跟用户代码产生冲突。

## 接口分类

我们在testing.common部分介绍过每个接口的实现，我们接下来就从函数功能上对接口进行分类。

以单元测试为例，每个测试函数都需要接收一个testing.T类型的指针做为函数参数，该参数主要用于控制测试流程（如结束和跳过）和记录日志。

### 记录日志

* Log(args ...interface{})
* Logf(format string, args ...interface{})

Log()和Logf()负责记录日志，其区别在于是否支持格式化参数；

### 标记失败+记录日志

* Error(args ...interface{})
* Errorf(format string, args ...interface{})

Error()和Errorf()负责标记当前测试失败并记录日志。 只标记测试状态为失败，并不影响测试函数流程，不会结束当前测试，也不会退出当前测试。

### 标记失败+记录日志+结束测试

* Fatal(args ...interface{})
* Fatalf(format string, args ...interface{})

Fatal()和Fatalf()负责标记当前测试失败、记录日志，并退出当前测试。

### 标记失败

* Fail()

Fail()仅标记录前测试状态为失败。

### 标记失败并退出

* FailNow()

FailNow()标记当前测试状态为失败并退出当前测试。

### 跳过测试+记录日志并退出

* Skip(args ...interface{})
* Skipf(format string, args ...interface{})

Skip()和Skipf()标记当前测试状态为跳过并记录日志，最后退出当前测试。

### 跳过测试并退出

* SkipNow()

SkipNow()标记测试状态为跳过，并退出当前测试。

## 私有接口避免冲突

接口定义中的private()方法是一个值得学习的用法。其目地是限定testing.TB接口的全局唯一性，即便用户的某个类型实现了除private()方法以外的其他方法，也不能说明实现了testing.TB接口，因为无法实现private()方法，private()方法属于testing包内部可见，外部不可见。

# 单元测试实现原理

## 简介

在了解过testing.common后，我们进一步了解testing.T数据结构，以便了解更多单元测试执行的更多细节。

## 数据结构

源码包src/testing/testing.go:T定义了其数据结构：

type T struct {  
 common  
 isParallel bool  
 context \*testContext // For running tests and subtests.  
}

其成员简单介绍如下：

* common： 即前面绍的testing.common
* isParallel： 表示当前测试是否需要并发，如果测试中执行了t.Parallel()，则此值为true
* context： 控制测试的并发调度

因为context直接决定了单元测试的调度，在介绍testing.T支持的方法前，有必要先了解一下context。

### testContext

源码包src/testing/testing.go:testContext定义了其数据结构：

type testContext struct {  
 match \*matcher  
  
 mu sync.Mutex  
  
 // Channel used to signal tests that are ready to be run in parallel.  
 startParallel chan bool  
  
 // running is the number of tests currently running in parallel.  
 // This does not include tests that are waiting for subtests to complete.  
 running int  
  
 // numWaiting is the number tests waiting to be run in parallel.  
 numWaiting int  
  
 // maxParallel is a copy of the parallel flag.  
 maxParallel int  
}

testContext成员简单介绍如下：

* match：匹配器，用于管理测试名称匹配、过滤等。
* mu：互斥锁，用于控制testContext成员的互斥访问；
* startParallel： 用于通知测试可以并发执行的控制管道，测试并发达到最大限制时，需要阻塞等待该管道的通知事件；
* running： 当前并发执行的测试个数；
* numWaiting：等待并发执行的测试个数，所有等待执行的测试都阻塞在startParallel管道处；
* maxParallel：最大并发数，默认为系统CPU数，可以通过参数-parallel n指定。

testContext实现了两个方法用于控制测试发调度。

### 等待并发执行：testContext.waitParallel()

如果一个测试使用t.Parallel()启动并发，这个测试并不是立即被并发执行，需要检查当前并发执行的测试数量是否达到最大值，这个检查工作统一放在testContext.waitParallel()实现的。

testContext.waitParallel()函数的源码如下：

func (c \*testContext) waitParallel() {  
 c.mu.Lock()  
 if c.running < c.maxParallel {  
 c.running++  
 c.mu.Unlock()  
 return  
 }  
 c.numWaiting++  
 c.mu.Unlock()  
 <-c.startParallel  
}

函数实现比较简单，如果当前运行的测试数未达最大值，将c.running++后直接返回即可，否则将c.numWaiting++并阻塞等待其他并发测试结束。

这里有个小细节，阻塞等待后面并没有累加c.running，因为其他并发的测试结束后也不会递减c.running，所以这里阻塞返回时也不用累加，一个测试结束，随即另一个测试开始，c.running个数没有变化。

### 并发测试结束：testContext.release()

当并发测试结束后，会通过release()方法释放一个信号，用于启动其他等待并发测试的函数。

testContext.release()函数的源码如下：

func (c \*testContext) release() {  
 c.mu.Lock()  
 if c.numWaiting == 0 {  
 c.running--  
 c.mu.Unlock()  
 return  
 }  
 c.numWaiting--  
 c.mu.Unlock()  
 c.startParallel <- true // Pick a waiting test to be run.  
}

## 测试执行：tRunner()

函数tRunner用于执行一个测试，在不考虑并发测试、子测试场景下，其处理逻辑如下：

unc tRunner(t \*T, fn func(t \*T)) {  
 t.runner = callerName(0)  
  
 // When this goroutine is done, either because fn(t)  
 // returned normally or because a test failure triggered  
 // a call to runtime.Goexit, record the duration and send  
 // a signal saying that the test is done.  
 defer func() {  
 if t.Failed() {  
 atomic.AddUint32(&numFailed, 1)  
 }  
  
 if t.raceErrors+race.Errors() > 0 {  
 t.Errorf("race detected during execution of test")  
 }  
  
 // Check if the test panicked or Goexited inappropriately.  
 //  
 // If this happens in a normal test, print output but continue panicking.  
 // tRunner is called in its own goroutine, so this terminates the process.  
 //  
 // If this happens while fuzzing, recover from the panic and treat it like a  
 // normal failure. It's important that the process keeps running in order to  
 // find short inputs that cause panics.  
 err := recover()  
 signal := true  
  
 t.mu.RLock()  
 finished := t.finished  
 t.mu.RUnlock()  
 if !finished && err == nil {  
 err = errNilPanicOrGoexit  
 for p := t.parent; p != nil; p = p.parent {  
 p.mu.RLock()  
 finished = p.finished  
 p.mu.RUnlock()  
 if finished {  
 t.Errorf("%v: subtest may have called FailNow on a parent test", err)  
 err = nil  
 signal = false  
 break  
 }  
 }  
 }  
  
 if err != nil && t.context.isFuzzing {  
 prefix := "panic: "  
 if err == errNilPanicOrGoexit {  
 prefix = ""  
 }  
 t.Errorf("%s%s\n%s\n", prefix, err, string(debug.Stack()))  
 t.mu.Lock()  
 t.finished = true  
 t.mu.Unlock()  
 err = nil  
 }  
  
 // Use a deferred call to ensure that we report that the test is  
 // complete even if a cleanup function calls t.FailNow. See issue 41355.  
 didPanic := false  
 defer func() {  
 if didPanic {  
 return  
 }  
 if err != nil {  
 panic(err)  
 }  
 // Only report that the test is complete if it doesn't panic,  
 // as otherwise the test binary can exit before the panic is  
 // reported to the user. See issue 41479.  
 t.signal <- signal  
 }()  
  
 doPanic := func(err any) {  
 t.Fail()  
 if r := t.runCleanup(*recoverAndReturnPanic*); r != nil {  
 t.Logf("cleanup panicked with %v", r)  
 }  
 // Flush the output log up to the root before dying.  
 for root := &t.common; root.parent != nil; root = root.parent {  
 root.mu.Lock()  
 root.duration += time.Since(root.start)  
 d := root.duration  
 root.mu.Unlock()  
 root.flushToParent(root.name, "--- FAIL: %s (%s)\n", root.name, fmtDuration(d))  
 if r := root.parent.runCleanup(*recoverAndReturnPanic*); r != nil {  
 fmt.Fprintf(root.parent.w, "cleanup panicked with %v", r)  
 }  
 }  
 didPanic = true  
 panic(err)  
 }  
 if err != nil {  
 doPanic(err)  
 }  
  
 t.duration += time.Since(t.start)  
  
 if len(t.sub) > 0 {  
 // Run parallel subtests.  
 // Decrease the running count for this test.  
 t.context.release()  
 // Release the parallel subtests.  
 close(t.barrier)  
 // Wait for subtests to complete.  
 for \_, sub := range t.sub {  
 <-sub.signal  
 }  
 cleanupStart := time.Now()  
 err := t.runCleanup(*recoverAndReturnPanic*)  
 t.duration += time.Since(cleanupStart)  
 if err != nil {  
 doPanic(err)  
 }  
 if !t.isParallel {  
 // Reacquire the count for sequential tests. See comment in Run.  
 t.context.waitParallel()  
 }  
 } else if t.isParallel {  
 // Only release the count for this test if it was run as a parallel  
 // test. See comment in Run method.  
 t.context.release()  
 }  
 t.report() // Report after all subtests have finished.  
  
 // Do not lock t.done to allow race detector to detect race in case  
 // the user does not appropriately synchronize a goroutine.  
 t.done = true  
 if t.parent != nil && atomic.LoadInt32(&t.hasSub) == 0 {  
 t.setRan()  
 }  
 }()  
 defer func() {  
 if len(t.sub) == 0 {  
 t.runCleanup(*normalPanic*)  
 }  
 }()  
  
 t.start = time.Now()  
 t.raceErrors = -race.Errors()  
 fn(t)  
  
 // code beyond here will not be executed when FailNow is invoked  
 t.mu.Lock()  
 t.finished = true  
 t.mu.Unlock()  
}

tRunner传一个经调度者设置过的testing.T参数和一个测试函数，执行时记录开始时间，然后将testing.T参数传入测试函数并同步等待其结束。

tRunner在defer语句中记录测试执行耗时，并上报日志，最后发送结束信号。

测试执行结束，进入defer后需要启动子测试，启动方法为关闭t.barrier管道，然后等待所有子测试执行结束。

需要注意的是，关闭t.barrier管道，阻塞在t.barrier管道上的协程同样会被唤醒，也是发送信号的一种方式，关于管道的更多实现细节，请参考管道实现原理相关章节。

defer中，如果检测到当前测试本身也处理并发中，那么结束后需要释放一个信号（t.context.release()）来启动一个等待的测试。

## 启动子测试：Run()

Run()函数用于启动一个子测试，这个子测试可以是用户的测试函数中主动调用Run()方法启动的，如果用户没有主动调用Run()方法，那么用户的测试函数也是被调度程序以Run()方法启动的。可以说，所有的测试都是Run()方法启动的。

按照惯例，隐去部分代码后的Run()方法如下所示：

// 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 \*T) Run(name string, f func(t \*T)) bool {  
 atomic.StoreInt32(&t.hasSub, 1)  
 testName, ok, \_ := t.context.match.fullName(&t.common, name)  
 if !ok || shouldFailFast() {  
 return true  
 }  
 // Record the stack trace at the point of this call so that if the subtest  
 // function - which runs in a separate stack - is marked as a helper, we can  
 // continue walking the stack into the parent test.  
 var pc [*maxStackLen*]uintptr  
 n := runtime.Callers(2, pc[:])  
 t = &T{  
 common: common{  
 barrier: make(chan bool),  
 signal: make(chan bool, 1),  
 name: testName,  
 parent: &t.common,  
 level: t.level + 1,  
 creator: pc[:n],  
 chatty: t.chatty,  
 },  
 context: t.context,  
 }  
 t.w = indenter{&t.common}  
  
 if t.chatty != nil {  
 t.chatty.Updatef(t.name, "=== RUN %s\n", t.name)  
 }  
 // Instead of reducing the running count of this test before calling the  
 // tRunner and increasing it afterwards, we rely on tRunner keeping the  
 // count correct. This ensures that a sequence of sequential tests runs  
 // without being preempted, even when their parent is a parallel test. This  
 // may especially reduce surprises if \*parallel == 1.  
 go tRunner(t, f)  
 if !<-t.signal {  
 // At this point, it is likely that FailNow was called on one of the  
 // parent tests by one of the subtests. Continue aborting up the chain.  
 runtime.Goexit()  
 }  
 return !t.failed  
}

每启动一个子测试都会创建一个testing.T变量，该变量继承当前测试的部分属性，然后以新协程去执行，当前测试会在子测试结束后返回子测试的结果。

子测试退出条件要么是子测试执行结束，要么是子测试设置了Paraller()，否则是异常退出。

## 启动并发测试：Parallel()

Parallel()方法将当前测试加入到并发队列中，其实现方法如下所示：

// 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 \*T) Parallel() {  
 if t.isParallel {  
 panic("testing: t.Parallel called multiple times")  
 }  
 if t.isEnvSet {  
 panic("testing: t.Parallel called after t.Setenv; cannot set environment variables in parallel tests")  
 }  
 t.isParallel = true  
 if t.parent.barrier == nil {  
 // T.Parallel has no effect when fuzzing.  
 // Multiple processes may run in parallel, but only one input can run at a  
 // time per process so we can attribute crashes to specific inputs.  
 return  
 }  
  
 // We don't want to include the time we spend waiting for serial tests  
 // in the test duration. Record the elapsed time thus far and reset the  
 // timer afterwards.  
 t.duration += time.Since(t.start)  
  
 // Add to the list of tests to be released by the parent.  
 t.parent.sub = append(t.parent.sub, t)  
 t.raceErrors += race.Errors()  
  
 if t.chatty != nil {  
 // Unfortunately, even though PAUSE indicates that the named test is \*no  
 // longer\* running, cmd/test2json interprets it as changing the active test  
 // for the purpose of log parsing. We could fix cmd/test2json, but that  
 // won't fix existing deployments of third-party tools that already shell  
 // out to older builds of cmd/test2json — so merely fixing cmd/test2json  
 // isn't enough for now.  
 t.chatty.Updatef(t.name, "=== PAUSE %s\n", t.name)  
 }  
  
 t.signal <- true // Release calling test.  
 <-t.parent.barrier // Wait for the parent test to complete.  
 t.context.waitParallel()  
  
 if t.chatty != nil {  
 t.chatty.Updatef(t.name, "=== CONT %s\n", t.name)  
 }  
  
 t.start = time.Now()  
 t.raceErrors += -race.Errors()  
}

关于测试耗时统计，看过前面的testContext实现我们知道，启动一个并发测试时，当并发数达到最大时，新的并发测试需要等待，那么等待期间的时间消耗不能统计到测试的耗时中，所以需要先计算当前耗时，在真正被并发调度后才清空t.start以跳过等待时间。

看过前面的Run()方法实现机制后，我们知道一旦子测试以并发模式执行时，需要通知父测试，其通知机制便是向t.signal管道中写入一个信号，父测试便从Run()方法中唤醒，继续执行。

看过前面的tRunner()方法实现机制后，不难理解，父测试唤醒后继续执行，结束后进入defer流程中，在defer中将启动所有子测试并等待子测试执行结束。

## 完整的测试执行：tRunner()

测试执行结束，进入defer后需要启动子测试，启动方法为关闭t.barrier管道，然后等待所有子测试执行结束。

需要注意的是，关闭t.barrier管道，阻塞在t.barrier管道上的协程同样会被唤醒，也是发送信号的一种方式，关于管道的更多实现细节，请参考管道实现原理相关章节。

defer中，如果检测到当前测试本身也处理并发中，那么结束后需要释放一个信号（t.context.release()）来启动一个等待的测试。

# 性能测试实现原理

## 简介

跟据前面章节，我们可以快速的写出一个性能测试并执行，最令我感到神奇的是b.N的值，虽然官方资料中说b.N会自动调整以保证可靠的计时，可还是想了解具体的实现机制。

本节，我们先分析testing.B数据结构，再看几个典型的成员函数，以期从源码中寻找以下问题的答案：

* b.N是如何自动调整的？
* 内存统计是如何实现的？
* SetBytes()其使用场景是什么？

## 数据结构

源码包src/testing/benchmark.go:B定义了性能测试的数据结构，我们提取其比较重要的一些成员进行分析：

// 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 output 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 {  
 common  
 importPath string // import path of the package containing the benchmark  
 context \*benchContext  
 N int  
 previousN int // number of iterations in the previous run  
 previousDuration time.Duration // total duration of the previous run  
 benchFunc func(b \*B)  
 benchTime durationOrCountFlag  
 bytes int64  
 missingBytes bool // one of the subbenchmarks does not have bytes set.  
 timerOn bool  
 showAllocResult bool  
 result BenchmarkResult  
 parallelism int // RunParallel creates parallelism\*GOMAXPROCS goroutines  
 // The initial states of memStats.Mallocs and memStats.TotalAlloc.  
 startAllocs uint64  
 startBytes uint64  
 // The net total of this test after being run.  
 netAllocs uint64  
 netBytes uint64  
 // Extra metrics collected by ReportMetric.  
 extra map[string]float64  
}

其主要成员如下：

* common： 与testing.T共享的testing.common，管理着日志、状态等；
* N：每个测试中用户代码执行次数
* benchFunc：测试函数
* benchTime：性能测试最少执行时间，默认为1s，可以通过能数-benchtime 2s指定
* bytes：每次迭代处理的字节数
* timerOn：计时启动标志，默认为false，启动计时为true
* startAllocs：测试启动时记录堆中分配的对象数
* startBytes：测试启动时记录堆中分配的字节数
* netAllocs：测试结束后记录堆中新增加的对象数，公式：结束时堆中分配的对象数-
* netBytes：测试对事后记录堆中新增加的字节数

## 关键函数

### 启动计时：B.StartTimer()

StartTimer()负责启动计时并初始化内存相关计数，测试执行时会自动调用，一般不需要用户启动。

// StartTimer starts timing a test. This function is called automatically  
// before a benchmark starts, but it can also be used to resume timing after  
// a call to StopTimer.  
func (b \*B) StartTimer() {  
 if !b.timerOn {  
 runtime.ReadMemStats(&memStats)  
 b.startAllocs = memStats.Mallocs  
 b.startBytes = memStats.TotalAlloc  
 b.start = time.Now()  
 b.timerOn = true  
 }  
}

StartTimer()负责启动计时，并记录当前内存分配情况，不管是否有“-benchmem”参数，内存都会被统计，参数只决定是否要在结果中输出。

### 停止计时：B.StopTimer()

StopTimer()负责停止计时，并累加相应的统计值。

// StopTimer stops timing a test. This can be used to pause the timer  
// while performing complex initialization that you don't  
// want to measure.  
func (b \*B) StopTimer() {  
 if b.timerOn {  
 b.duration += time.Since(b.start)  
 runtime.ReadMemStats(&memStats)  
 b.netAllocs += memStats.Mallocs - b.startAllocs  
 b.netBytes += memStats.TotalAlloc - b.startBytes  
 b.timerOn = false  
 }  
}

需要注意的是，StopTimer()并不一定是测试结束，一个测试中有可能有多个统计阶段，所以其统计值是累加的。

### 重置计时：B.ResetTimer()

ResetTimer()用于重置计时器，相应的也会把其他统计值也重置。

// ResetTimer zeroes the elapsed benchmark time and memory allocation counters  
// and deletes user-reported metrics.  
// It does not affect whether the timer is running.  
func (b \*B) ResetTimer() {  
 if b.extra == nil {  
 // Allocate the extra map before reading memory stats.  
 // Pre-size it to make more allocation unlikely.  
 b.extra = make(map[string]float64, 16)  
 } else {  
 for k := range b.extra {  
 delete(b.extra, k)  
 }  
 }  
 if b.timerOn {  
 runtime.ReadMemStats(&memStats)  
 b.startAllocs = memStats.Mallocs  
 b.startBytes = memStats.TotalAlloc  
 b.start = time.Now()  
 }  
 b.duration = 0  
 b.netAllocs = 0  
 b.netBytes = 0  
}

ResetTimer()比较常用，典型使用场景是一个测试中，初始化部分耗时较长，初始化后再开始计时。

### 设置处理字节数：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 \*B) SetBytes(n int64) { b.bytes = n }

这是一个比较含糊的函数，通过其函数说明很难明白其作用。

其实它是用来设置单次迭代处理的字节数，一旦设置了这个字节数，那么输出报告中将会呈现“xxx MB/s”的信息，用来表示待测函数处理字节的性能。待测函数每次处理多少字节数只有用户清楚，所以需要用户设置。

举个例子，待测函数每次执行处理1M数据，如果我们想看待测函数处理数据的性能，那么我们在测试中设置SetByte(1024 \*1024)，假如待测函数需要执行1s的话，那么结果中将会出现 “1 MB/s”（约等于）的信息。示例代码如下所示：

func BenchmarkSetBytes(b \*testing.B) {  
 b.SetBytes(1024 \* 1024)  
 for i := 0; i < b.N; i++ {  
 time.Sleep(1 \* time.*Second*) // 模拟待测函数  
 }  
}

打印结果：

E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test -bench SetBytes benchmark\_test.go  
BenchmarkSetBytes-4 1 1010392800 ns/op 1.04 MB/s  
PASS  
ok command-line-arguments 1.412s

可以看到测试执行了一次，花费时间约1S，数据处理能力约为1MB/s。

### **报告内存信息：**

// 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 \*B) ReportAllocs() {  
 b.showAllocResult = true  
}

ReportAllocs() 用于设置是否打印内存统计信息，与命令行参数“-benchmem”一致，但本方法只作用于单个测试函数。

## 性能测试是如何启动的

性能测试要经过多次迭代，每次迭代可能会有不同的b.N值，每次迭代执行测试函数一次，跟据此次迭代的测试结果来分析要不要继续下一次迭代。

我们先看一下每次迭代时所用到的方法，runN():

// runN runs a single benchmark for the specified number of iterations.  
func (b \*B) runN(n int) {  
 benchmarkLock.Lock()  
 defer benchmarkLock.Unlock()  
 defer b.runCleanup(*normalPanic*)  
 // Try to get a comparable environment for each run  
 // by clearing garbage from previous runs.  
 runtime.GC()  
 b.raceErrors = -race.Errors()  
 b.N = n  
 b.parallelism = 1  
 b.ResetTimer()  
 b.StartTimer()  
 b.benchFunc(b)  
 b.StopTimer()  
 b.previousN = n  
 b.previousDuration = b.duration  
 b.raceErrors += race.Errors()  
 if b.raceErrors > 0 {  
 b.Errorf("race detected during execution of benchmark")  
 }  
}

该方法指定b.N的值，执行一次测试函数。

与T.Run()类似，B.Run()也用于启动一个子测试，实际上用户编写的任何一个测试都是使用Run()方法启动的，我们看下B.Run()的伪代码：

// 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 \*B) Run(name string, f func(b \*B)) bool {  
 // Since b has subbenchmarks, we will no longer run it as a benchmark itself.  
 // Release the lock and acquire it on exit to ensure locks stay paired.  
 atomic.StoreInt32(&b.hasSub, 1)  
 benchmarkLock.Unlock()  
 defer benchmarkLock.Lock()  
  
 benchName, ok, partial := b.name, true, false  
 if b.context != nil {  
 benchName, ok, partial = b.context.match.fullName(&b.common, name)  
 }  
 if !ok {  
 return true  
 }  
 var pc [*maxStackLen*]uintptr  
 n := runtime.Callers(2, pc[:])  
 sub := &B{  
 common: common{  
 signal: make(chan bool),  
 name: benchName,  
 parent: &b.common,  
 level: b.level + 1,  
 creator: pc[:n],  
 w: b.w,  
 chatty: b.chatty,  
 bench: true,  
 },  
 importPath: b.importPath,  
 benchFunc: f,  
 benchTime: b.benchTime,  
 context: b.context,  
 }  
 if partial {  
 // Partial name match, like -bench=X/Y matching BenchmarkX.  
 // Only process sub-benchmarks, if any.  
 atomic.StoreInt32(&sub.hasSub, 1)  
 }  
  
 if b.chatty != nil {  
 labelsOnce.Do(func() {  
 fmt.Printf("goos: %s\n", runtime.*GOOS*)  
 fmt.Printf("goarch: %s\n", runtime.*GOARCH*)  
 if b.importPath != "" {  
 fmt.Printf("pkg: %s\n", b.importPath)  
 }  
 if cpu := sysinfo.CPU.Name(); cpu != "" {  
 fmt.Printf("cpu: %s\n", cpu)  
 }  
 })  
  
 fmt.Println(benchName)  
 }  
  
 if sub.run1() {  
 sub.run()  
 }  
 b.add(sub.result)  
 return !sub.failed  
}

所有的测试都是先使用run1()方法执行一次测试,run1()方法中实际上调用了runN(1)，执行一次后再决定要不要继续迭代。

测试结果实际上以最后一次迭代的数据为准，当然，最后一次迭代往往意味着b.N更大，测试准确性相对更高。

## B.N是如何调整的？

B.launch()方法里最终决定B.N的值。我们看下伪代码：

// launch launches the benchmark function. It gradually increases the number  
// of benchmark iterations until the benchmark runs for the requested benchtime.  
// launch is run by the doBench function as a separate goroutine.  
// run1 must have been called on b.  
func (b \*B) launch() {  
 // Signal that we're done whether we return normally  
 // or by FailNow's runtime.Goexit.  
 defer func() {  
 b.signal <- true  
 }()  
  
 // Run the benchmark for at least the specified amount of time.  
 if b.benchTime.n > 0 {  
 // We already ran a single iteration in run1.  
 // If -benchtime=1x was requested, use that result.  
 // See https://golang.org/issue/32051.  
 if b.benchTime.n > 1 {  
 b.runN(b.benchTime.n)  
 }  
 } else {  
 d := b.benchTime.d  
 for n := int64(1); !b.failed && b.duration < d && n < 1e9; {  
 last := n  
 // Predict required iterations.  
 goalns := d.Nanoseconds()  
 prevIters := int64(b.N)  
 prevns := b.duration.Nanoseconds()  
 if prevns <= 0 {  
 // Round up, to avoid div by zero.  
 prevns = 1  
 }  
 // Order of operations matters.  
 // For very fast benchmarks, prevIters ~= prevns.  
 // If you divide first, you get 0 or 1,  
 // which can hide an order of magnitude in execution time.  
 // So multiply first, then divide.  
 n = goalns \* prevIters / prevns  
 // Run more iterations than we think we'll need (1.2x).  
 n += n / 5  
 // Don't grow too fast in case we had timing errors previously.  
 n = min(n, 100\*last)  
 // Be sure to run at least one more than last time.  
 n = max(n, last+1)  
 // Don't run more than 1e9 times. (This also keeps n in int range on 32 bit platforms.)  
 n = min(n, 1e9)  
 b.runN(int(n))  
 }  
 }  
 b.result = BenchmarkResult{b.N, b.duration, b.bytes, b.netAllocs, b.netBytes, b.extra}  
}

不考虑程序出错，而且用户没有主动停止测试的场景下，每个性能测试至少要执行b.benchTime长的秒数，默认为1s。先执行一遍的意义在于看用户代码执行一次要花费多长时间，如果时间较短，那么b.N值要足够大才可以测得更精确，如果时间较长，b.N值相应的会减少，否则会影响测试效率。

最终的b.N会被定格在某个10的指数级，是为了方便阅读测试报告。

## 内存是如何统计的？

我们知道在测试开始时，会把当前内存值记入到b.startAllocs和b.startBytes中，测试结束时，会用最终内存值与开始时的内存值相减，得到净增加的内存值，并记入到b.netAllocs和b.netBytes中。

每个测试结束，会把结果保存到BenchmarkResult对象里，该对象里保存了输出报告所必需的统计信息：

// BenchmarkResult contains 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.  
  
 // Extra records additional metrics reported by ReportMetric.  
 Extra map[string]float64  
}

其中MemAllocs和MemBytes分别对应b.netAllocs和b.netBytes。

那么最终统计时只需要把净增加值除以b.N即可得到每次新增多少内存了。

每个操作内存对象新增值：

// AllocsPerOp returns the "allocs/op" metric,  
// which is calculated as r.MemAllocs / r.N.  
func (r BenchmarkResult) AllocsPerOp() int64 {  
 if v, ok := r.Extra["allocs/op"]; ok {  
 return int64(v)  
 }  
 if r.N <= 0 {  
 return 0  
 }  
 return int64(r.MemAllocs) / int64(r.N)  
}

每个操作内存字节数新增值：

// AllocedBytesPerOp returns the "B/op" metric,  
// which is calculated as r.MemBytes / r.N.  
func (r BenchmarkResult) AllocedBytesPerOp() int64 {  
 if v, ok := r.Extra["B/op"]; ok {  
 return int64(v)  
 }  
 if r.N <= 0 {  
 return 0  
 }  
 return int64(r.MemBytes) / int64(r.N)  
}

# 示例测试实现原理

## 简介

示例测试相对于单元测试和性能测试来说，其实现机制比较简单。它没有复杂的数据结构，也不需要额外的流程控制，其核心工作原理在于收集测试过程中的打印日志，然后与期望字符串做比较，最后得出是否一致的报告。

## 数据结构

每个测试经过编译后都有一个数据结构来承载，这个数据结构即InternalExample:

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

比如，示例测试如下：

// 检测乱序输出

func ExamplePrintNames() {

gotest.PrintNames()

// Unordered output:

// Jim

// Bob

// Tom

// Sue

}

该示例测试经过编译后，产生的数据结构成员如下：

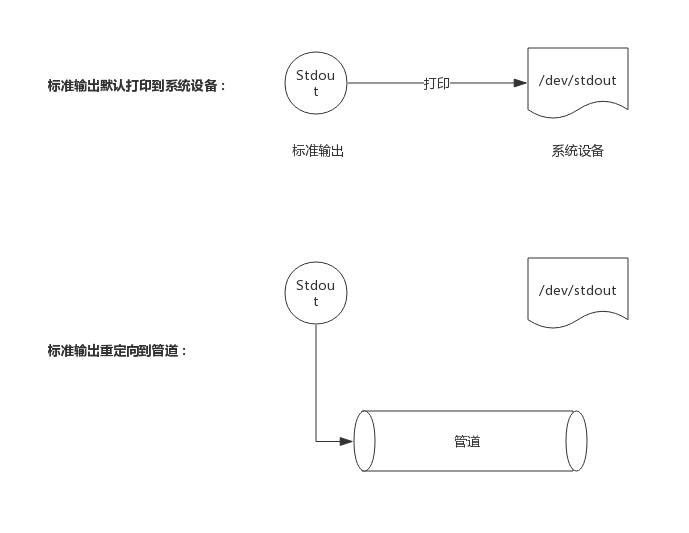
* InternalExample.Name = "ExamplePrintNames";
* InternalExample.F = ExamplePrintNames()
* InternalExample.Output = "Jim\n Bob\n Tom\n Sue\n"
* InternalExample.Unordered = true;

其中Output是包含换行符的字符串。

## 捕获标准输出

在示例测试开始前，需要先把标准输出捕获，以便获取测试执行过程中的打印日志。

捕获标准输出方法是新建一个管道，将标准输出重定向到管道的入口(写口)，这样所有打印到屏幕的日志都会输入到管道中，如下图所示：



测试开始前捕获，测试结束恢复标准输出，这样测试过程中的日志就可以从管理中读取了。

## 测试结果比较

测试执行过程的输出内容最终也会保存到一个string类型变量里，该变量会与InternalExample.Output进行比较，二者一致即代表测试通过，否则测试失败。

输出有序的情况下，比较很简单只是比较两个String内容是否一致即可。无序的情况下则需要把两个String变量排序后再进行对比。

比如，期望字符串为："Jim\n Bob\n Tom\n Sue\n"，排序后则变为："Bob\n Jim\n Sue\n Tom\n"

## 测试执行

一个完整的测试，过程将分为如下步骤：

1. 捕获标准输出
2. 执行测试
3. 恢复标准输出
4. 比较结果

下面，由于源码非常简单，下面直接给出源码：

func runExample(eg InternalExample) (ok bool) {  
 if \*chatty {  
 fmt.Printf("=== RUN %s\n", eg.Name)  
 }  
  
 // Capture stdout.  
 stdout := os.Stdout  
 r, w, err := os.Pipe()  
 if err != nil {  
 fmt.Fprintln(os.Stderr, err)  
 os.Exit(1)  
 }  
 os.Stdout = w  
 outC := make(chan string)  
 go func() {  
 var buf strings.Builder  
 \_, err := io.Copy(&buf, r)  
 r.Close()  
 if err != nil {  
 fmt.Fprintf(os.Stderr, "testing: copying pipe: %v\n", err)  
 os.Exit(1)  
 }  
 outC <- buf.String()  
 }()  
  
 finished := false  
 start := time.Now()  
  
 // Clean up in a deferred call so we can recover if the example panics.  
 defer func() {  
 timeSpent := time.Since(start)  
  
 // Close pipe, restore stdout, get output.  
 w.Close()  
 os.Stdout = stdout  
 out := <-outC  
  
 err := recover()  
 ok = eg.processRunResult(out, timeSpent, finished, err)  
 }()  
  
 // Run example.  
 eg.F()  
 finished = true  
 return  
}

示例测试执行时，捕获标准输出后，马上启动一个协程阻塞在管道处读取数据，一直阻塞到管道关闭，管道关闭也即读取结束，然后把日志通过channel发送到主协程中。

主协程直接执行示例测试，而在defer中去执行关闭管道、接收日志、判断结果等操作。

# 模糊测试实现原理

## 什么是模糊测试

模糊测试是一项测试技术，通过测试基础架构用随机生成的输入调用代码，以检查它是否产生正确的结果或合理的错误。模糊测试是对单元测试的补充，在单元测试中，你给定一组静态输入来测试你的代码是否产生正确的输出。单元测试的局限性在于你只能用预期的输入进行测试；模糊测试擅长发现暴露奇怪行为的\_非预期\_输入。一个好的模糊测试系统还可以利用被测试的代码，这样它就可以有效地生成扩大代码覆盖率的输入。

模糊测试**通常用于检查解析器和验证器**，尤其是安全上下文中使用的任何东西。模糊测试非常擅长发现导致安全问题的错误，如二进制编码中的无效长度、截断输入、整数溢出、无效Unicode等。

还有其他使用模糊的方法。例如，差分模糊通过向两个实现输入相同的随机输入并检查输出是否匹配来验证同一事物的两个实现具有相同的行为。您也可以使用模糊来进行用户界面“猴子”测试：模糊引擎可以产生随机点击、按键和点击，测试验证应用程序没有崩溃。

## Go中的模糊测试是什么情况

模糊测试对 Go 来说并不新鲜，go-fuzz 可能是当今使用最广泛的工具，我们在开发原生模糊时也借鉴了它的设计。Go 1.18 中的新情况是，模糊直接集成到 go test 和 testing 包中，接口与 testing 接口非常相似，testing.T。

例如，如果你有一个名为 ParseSomething 的函数，就可以编写一个如下所示的模糊测试。这将检查对于任何随机输入，ParseSomething 要么成功，要么返回一个 ParseError。

package parser  
import (  
 "errors"  
 "testing"  
)  
var seeds = [][]byte{  
 nil,  
 []byte("123"),  
 []byte("(12)"),  
}  
func FuzzParseSomething(f \*testing.F) {  
 for \_, seed := range seeds {  
 f.Add(seed)  
 }  
 f.Fuzz(func(t \*testing.T, input []byte) {  
 err := ParseSomething(input)  
 if err == nil {  
 return  
 }  
 if parseErr := (\*ParseError)(nil); !errors.As(err, &parseErr) {  
 t.Fatal(err)  
 }  
 })  
}

当正常运行 go test 时（没有 -fuzz 标签），FuzzParseSomething 会按单元测试来对待。提供给 F.Fuzz 的模糊测试函数会使用种子语料中的输入来调用：使用 F.Add注册的输入和 testdata/corpus/FuzzParseSomething 文件夹中读取文件的输入。如果模糊测试函数 panic 或调用 T.Fail，测试会失败，而 go test 会以非零的状态退出。

模糊测试可以在运行 go test 时增加 -fuzz 标签来开启，如下所示：

go test -fuzz=FuzzParseSomething

在这种模式下，模糊测试系统将用随机生成的输入调用模糊测试函数，使用来自种子语料库和一个缓存语料库的输入作为起点。生成的扩大覆盖范围的输入被最小化并添加到缓存语料库中。生成的导致错误的输入被最小化并添加到种子语料库中，有效地成为新的回归测试用例。以后的 go test 在问题被修复前都会执行失败，即使没有开启模糊测试。

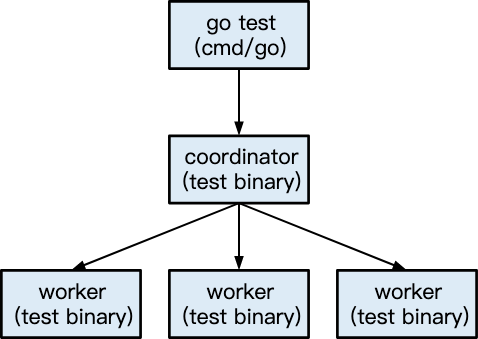
同样，与其他系统相比，这里没有什么真正的新颖之处。优势在于接口的熟悉性和易用性。编写你的第一个模糊测试很容易，因为模糊测试遵循 testing 包的约定。无需让团队中的每个人都安装和学习一个新工具。

## 模糊测试系统如何工作

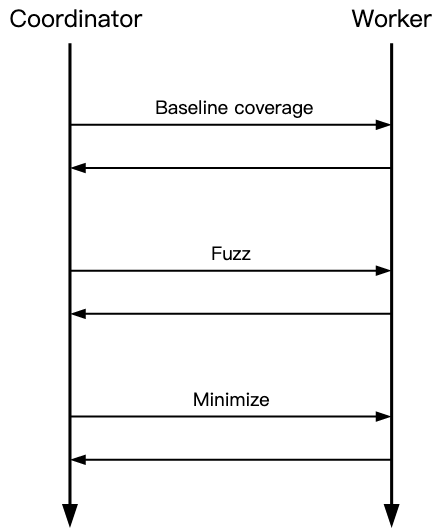
你可能已经知道 go test 为每个被测试的包构建一个测试可执行文件，然后运行这些可执行文件以获得测试和基准测试结果。模糊测试遵循这种模式，尽管有一些不同。

当使用 -fuzz 标志调用 go test时，go test 使用额外的覆盖率测量来编译测试可执行文件。Go 编译器已经对 libFuzzer 提供了测量支持，所以我们重用了它。编译器为每个基本块添加了一个 8 位计数器。计数器快速且近似：它包装了溢出，并且没有跨线程的同步。（我们不得不告诉竞争检测器不要检测这些计数器的写入）。计数器数据在运行时由内部internal/fuzz包使用，其中包括了大部分的模糊逻辑。

在 go test 构建一个可测量的可执行文件后，它会像往常一样运行它。这被称为协调进程。这个进程以传递给 go test 的大部分标志开始，包括-fuzz=pattern，用来识别要模糊测试的目标；目前，每次 go test 调用只能模糊测试一个目标（#46312）。当目标调用 F.Fuzz时，控制被传递给 [fuzz.CoordinateFuzzing] (https://pkg.go.dev/internal/fuzz#CoordinateFuzzing)，它将初始化模糊测试系统并启动协调器的事件循环。协调器启动几个辅助进程，运行相同的测试可执行文件并执行实际的模糊测试。辅助进程使用一个未记录的命令行标志启动，该标志告诉它们成为辅助进程。模糊测试必须在不同的进程中进行，这样如果辅助进程完全崩溃，协调器仍然可以找到并记录导致崩溃的输入。



协调器通过一对管道使用基于JSON的简易 RPC 协议与每个辅助角色进行通信。该协议非常简单，因为我们不需要像 gRPC 这样复杂的东西，也不想在标准库中引入任何新的东西。每个辅助角色还在内存映射的临时文件中保留一些状态，与协调器共享。这主要是迭代计数和随机数生成器状态。如果辅助角色完全崩溃，协调器可以从共享内存中恢复其状态，而不需要辅助角色先通过管道发送消息。协调器启动辅助角色后，通过从种子语料库和模糊缓存语料库（在 $GOCACHE 的子目录中）发送给辅助角色的输入来收集基线覆盖率。每个辅助角色运行其给定的输入，然后用其覆盖率计数器的快照报告。协调器将这些计数器粗化并合并成一个组合的覆盖率数组。接下来，协调器从种子语料库和缓存的语料库中发送输入进行模糊测试：每个辅助角色都被赋予一个输入和基线覆盖率数组的副本。然后每个辅助角色随机地改变其输入（翻转位、删除或插入字节等）并调用模糊测试函数。为了减少通信开销，每个辅助角色可以在没有协调器进一步输入的情况下保持 100 ms 的改变和调用。每次调用后，辅助角色检查是否报告了错误（带有 T.Fail）或与基线覆盖率数组相比发现了新的覆盖。如果是，辅助角色立即向协调器报告“有趣”的输入。当协调器收到一个产生新覆盖率的输入时，它会将辅助角色的覆盖范围与当前的组合覆盖范围数组进行比较：可能另一个辅助角色已经发现了一个提供相同覆盖范围的输入。如果是这样，新的输入就会被丢弃。如果新的输入确实提供了新的覆盖范围，协调器会将其发送回一个辅助角色（可能是另一个辅助角色）进行最小化。最小化就像模糊测试，但是辅助角色会执行随机突变来创建一个更小的输入，这个输入仍然提供了至少一些新的覆盖范围。较小的输入往往会更快，所以值得花时间提前最小化，以便以后模糊测试过程更快。辅助进程在完成最小化后返回报告，即使它没有找到任何更小的内容。协调器将最小化的输入添加到缓存的语料库中并继续。稍后，协调器可能会将最小化的输入发送给辅助进程进行进一步的模糊处理。这就是模糊测试系统适应新覆盖范围的方式。当协调器收到导致错误的输入时，它会再次将输入发送回辅助角色以进行最小化。在这种情况下，辅助角色试图找到一个仍然会导致错误的较小输入，尽管不一定是相同的错误。输入最小化后，协调器将其保存到 testdata/corpus/$FuzzTarget 中，优雅地关闭辅助角色进程，然后以非零状态退出。



如果辅助进程在模糊处理时崩溃，协调器可以使用发送给辅助进程的输入、辅助进程的 RNG 状态和迭代计数（都留在共享内存中）恢复导致崩溃的输入。崩溃输入通常不会最小化，因为最小化是一个高度有状态的过程，每次崩溃都会将该状态清空。理论上这是可能的，但还没有完成。模糊测试通常会持续到发现错误或者用户通过按下 Ctrl-C 中断进程，或者通过设置有 -fuzztime 标志的截止日期。模糊测试引擎会优雅地处理中断，无论它们是传递给协调器还是辅助进程。例如，如果一个辅助进程在最小化导致错误的输入时被中断，协调器会保存未最小化的输入。

# 子测试

## 简介

简单的说，子测试提供一种在一个测试函数中执行多个测试的能力，比如原来有TestA、TestB和TestC三个测试函数，每个测试函数执行开始都需要做些相同的初始化工作，那么可以利用子测试将这三个测试合并到一个测试中，这样初始化工作只需要做一次。

除此之外，子测试还提供了诸多便利，下面我们逐一说明。

## **简单例子**

我们先看一个简单的例子，以便快速了解子测试的基本用法。

package gotest\_test  
  
import (  
 "testing"  
 "gotest"  
)  
  
// sub1 为子测试，只做加法测试  
func sub1(t \*testing.T) {  
 var a = 1  
 var b = 2  
 var expected = 3  
  
 actual := gotest.Add(a, b)  
 if actual != expected {  
 t.Errorf("Add(%d, %d) = %d; expected: %d", a, b, actual, expected)  
 }  
}  
  
// sub2 为子测试，只做加法测试  
func sub2(t \*testing.T) {  
 var a = 1  
 var b = 2  
 var expected = 3  
  
 actual := gotest.Add(a, b)  
 if actual != expected {  
 t.Errorf("Add(%d, %d) = %d; expected: %d", a, b, actual, expected)  
 }  
}  
  
// sub3 为子测试，只做加法测试  
func sub3(t \*testing.T) {  
 var a = 1  
 var b = 2  
 var expected = 3  
  
 actual := gotest.Add(a, b)  
 if actual != expected {  
 t.Errorf("Add(%d, %d) = %d; expected: %d", a, b, actual, expected)  
 }  
}  
  
// TestSub 内部调用sub1、sub2和sub3三个子测试  
func TestSub(t \*testing.T) {  
 // setup code  
  
 t.Run("A=1", sub1)  
 t.Run("A=2", sub2)  
 t.Run("B=1", sub3)  
  
 // tear-down code  
}

本例中TestSub()通过t.Run()依次执行三个子测试。t.Run()函数声明如下：

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

name参数为子测试的名字，f为子测试函数，本例中Run()一直阻塞到f执行结束后才返回，返回值为f的执行结果。 Run()会启动新的协程来执行f，并阻塞等待f执行结束才返回，除非f中使用t.Parallel()设置子测试为并发。

本例中TestSub()把三个子测试合并起来，可以共享setup和tear-down部分的代码。

我们在命令行下，使用-v参数执行测试：

从输出中可以看出，三个子测试都被执行到了，而且执行次序与调用次序一致。

## 子测试命名规则

通过上面的例子我们知道Run()方法第一个参数为子测试的名字，而实际上子测试的内部命名规则为："*<父测试名字>*/*<传递给Run的名字>*"。比如，传递给Run()的名字是“A=1”，那么子测试名字为“TestSub/A=1”。这个在上面的命令行输出中也可以看出。

## 过滤筛选

通过测试的名字，可以在执行中过滤掉一部分测试。

比如，只执行上例中“A=\*”的子测试，那么执行时使用-run Sub/A=参数即可：

E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test subunit\_test.go -v  
=== RUN TestSub  
=== RUN TestSub/A=1  
=== RUN TestSub/A=2  
=== RUN TestSub/B=1  
--- PASS: TestSub (0.00s)  
 --- PASS: TestSub/A=1 (0.00s)  
 --- PASS: TestSub/A=2 (0.00s)  
 --- PASS: TestSub/B=1 (0.00s)  
PASS  
ok command-line-arguments 0.354s

上例中，使用参数-run Sub/A=则只会执行TestSub/A=1和TestSub/A=2两个子测试。

对于子性能测试则使用-bench参数来筛选，此处不再赘述。

注意：此处的筛选不是严格的正则匹配，而是包含匹配。比如，-run A=那么所有测试（含子测试）的名字中如果包含“A=”则会被选中执行。

## 子测试并发

前面提到的多个子测试共享setup和teardown有一个前提是子测试没有并发，如果子测试使用t.Parallel()指定并发，那么就没办法共享teardown了，因为执行顺序很可能是setup->子测试1->teardown->子测试2...。

如果子测试可能并发，则可以把子测试通过Run()再嵌套一层，Run()可以保证其下的所有子测试执行结束后再返回。

为便于说明，我们创建文件subparallel\_test.go用于说明：

package gotest\_test  
  
import (  
 "testing"  
 "time"  
)  
  
// 并发子测试，无实际测试工作，仅用于演示  
func parallelTest1(t \*testing.T) {  
 t.Parallel()  
 time.Sleep(3 \* time.*Second*)  
 // do some testing  
}  
  
// 并发子测试，无实际测试工作，仅用于演示  
func parallelTest2(t \*testing.T) {  
 t.Parallel()  
 time.Sleep(2 \* time.*Second*)  
 // do some testing  
}  
  
// 并发子测试，无实际测试工作，仅用于演示  
func parallelTest3(t \*testing.T) {  
 t.Parallel()  
 time.Sleep(1 \* time.*Second*)  
 // do some testing  
}  
  
// TestSubParallel 通过把多个子测试放到一个组中并发执行，同时多个子测试可以共享setup和tear-down  
func TestSubParallel(t \*testing.T) {  
 // setup  
 t.Logf("Setup")  
  
 t.Run("group", func(t \*testing.T) {  
 t.Run("Test1", parallelTest1)  
 t.Run("Test2", parallelTest2)  
 t.Run("Test3", parallelTest3)  
 })  
  
 // tear down  
 t.Logf("teardown")  
}

上面三个子测试中分别sleep了3s、2s、1s用于观察并发执行顺序。通过Run()将多个子测试“封装”到一个组中，可以保证所有子测试全部执行结束后再执行tear-down。

命令行下的输出如下：

E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test subparallel\_test.go -v -run SubParallel  
=== RUN TestSubParallel  
=== RUN TestSubParallel/group  
=== RUN TestSubParallel/group/Test1  
=== RUN TestSubParallel/group/Test2  
=== RUN TestSubParallel/group/Test3  
--- PASS: TestSubParallel (3.01s)  
 subparallel\_test.go:25: Setup  
 --- PASS: TestSubParallel/group (0.00s)  
 --- PASS: TestSubParallel/group/Test3 (1.00s)  
 --- PASS: TestSubParallel/group/Test2 (2.01s)  
 --- PASS: TestSubParallel/group/Test1 (3.01s)  
 subparallel\_test.go:34: teardown  
PASS  
ok command-line-arguments 3.353s

通过该输出可以看出：

* 子测试是并发执行的（Test1最先被执行却最后结束）
* tear-down在所有子测试结束后才执行

## **总结**

* 子测试适用于单元测试和性能测试；
* 子测试可以控制并发；
* 子测试提供一种类似table-driven风格的测试；
* 子测试可以共享setup和tear-down；

# Main测试

## 简介

我们知道子测试的一个方便之处在于可以让多个测试共享Setup和Tear-down。但这种程度的共享有时并不满足需求，有时希望在整个测试程序做一些全局的setup和Tear-down，这时就需要Main测试了。

所谓Main测试，即声明一个func TestMain(m \*testing.M)，它是名字比较特殊的测试，参数类型为testing.M指针。如果声明了这样一个函数，当前测试程序将不是直接执行各项测试，而是将测试交给TestMain调度。

## **示例**

下面通过一个例子来展示Main测试用法：

// TestMain 用于主动执行各种测试，可以测试前后做setup和tear-down操作  
func TestMain(m \*testing.M) {  
 println("TestMain setup.")  
  
 retCode := m.Run() // 执行测试，包括单元测试、性能测试和示例测试  
  
 println("TestMain tear-down.")  
  
 os.Exit(retCode)  
}

上述例子中，日志打印的两行分别对应Setup和Tear-down代码，m.Run()即为执行所有的测试，m.Run()的返回结果通过os.Exit()返回。

如果所有测试均通过测试，m.Run()返回0，否同m.Run()返回1，代表测试失败。

有一点需要注意的是，TestMain执行时，命令行参数还未解析，如果测试程序需要依赖参数，可以使用flag.Parse()解析参数，m.Run()方法内部还会再次解析参数，此处解析不会影响原测试过程。

# Main测试实现原理

## 简介

每一种测试（单元测试、性能测试或示例测试），都有一个数据类型与其对应。

* 单元测试：InternalTest
* 性能测试：InternalBenchmark
* 示例测试：InternalExample

测试编译阶段，每个测试都会被放到指定类型的切片中，测试执行时，这些测试将会被放到testing.M数据结构中进行调度。

而testing.M即是MainTest对应的数据结构。

## 数据结构

源码src\testing/testing.go:M定义了testing.M的数据结构：

// M is a type passed to a TestMain function to run the actual tests.  
type M struct {  
 deps testDeps  
 tests []InternalTest  
 benchmarks []InternalBenchmark  
 fuzzTargets []InternalFuzzTarget  
 examples []InternalExample  
  
 timer \*time.Timer  
 afterOnce sync.Once  
  
 numRun int  
  
 // value to pass to os.Exit, the outer test func main  
 // harness calls os.Exit with this code. See #34129.  
 exitCode int  
}

单元测试、性能测试和示例测试在经过编译后都会被存放到一个testing.M数据结构中，在测试执行时该数据结构将传递给TestMain()，真正执行测试的是testing.M的Run()方法，这个后面我们会继续分析。

timer用于指定测试的超时时间，可以通过参数timeout <n>指定，当测试执行超时后将会立即结束并判定为失败。

## **执行测试**

TestMain()函数通常会有一个m.Run()方法，该方法会执行单元测试、性能测试和示例测试，如果用户实现了TestMain()但没有调用m.Run()的话，那么什么测试都不会被执行。

m.Run()不仅会执行测试，还会做一些初始化工作，比如解析参数、起动定时器、跟据参数指示创建一系列的文件等。

m.Run()使用三个独立的方法来执行三种测试：

* 单元测试：runTests(m.deps.MatchString, m.tests)
* 性能测试：runExamples(m.deps.MatchString, m.examples)
* 示例测试：runBenchmarks(m.deps.ImportPath(), m.deps.MatchString, m.benchmarks) 其中m.deps里存放了测试匹配相关的内容，暂时先不用关注。

# Go test 工作机制

## 前言

前面的章节我们分析了每种测试的数据结构及其实现原理，本节我们看一下go test的执行机制。

Go 有多个命令行工具，go test只是其中一个。go test命令的函数入口在src\cmd\go\internal\test\test.go:runTest()，这个函数就是go test的大脑。

## runTest()

runTest()函数场景如下：

func runTest(cmd \*base.Command, args []string)

GO 命令行工具的实现中，都遵循这种函数声明，其中args即命令行输入的全部参数。

runTest首先会分析所有需要测试的包，为每个待测包生成一个二进制文件，然后执行。

## 两种运行模式

go test运行时，跟据是否指定package分为两种模式，即本地目录模式和包列表模式。

### 本地目录模式

当执行测试并没有指定package时，即以本地目录模式运行，例如使用"go test"或者"go test -v"来启动测试。

本地目录模式下，go test编译当前目录的源码文件和测试文件，并生成一个二进制文件，最后执行并打印结果。

### 包列表模式

当执行测试并显式指定package时，即以包列表模式运行，例如使用"go test math"来启动测试。

包列表模式下，go test为每个包生成一个测试二进制文件，并分别执行它。 包列表模式是在Go 1.10版本才引入的，它会把每个包的测试结果写入到本地临时文件中做为缓存，下次执行时会直接从缓存中读取测试结果，以便节省测试时间。

## 缓存机制

当满足一定的条件，测试的缓存是自动启用的，也可以显式的关闭缓存。

### 测试结果缓存

如果一次测试中，其参数全部来自"可缓存参数"集合，那么本次测试结果将被缓存。

可缓存参数集合如下：

* -cpu
* -list
* -parallel
* -run
* -short
* -v

需要注意的是，测试参数必须全部来自这个集合，其结果才会被缓存，没有参数或包含任一此集合之外的参数，结果都不会缓存。

### 使用缓存结果

如果满足条件，测试不会真正执行，而是从缓存中取出结果并呈现，结果中会有"cached"字样，表示来自缓存。

使用缓存结果也需要满足一定的条件：

* 本次测试的二进制及测试参数与之前的一次完全一致；
* 本次测试的源文件及环境变量与之前的一次完全一致；
* 之前的一次测试结果是成功的；
* 本次测试运行模式是列表模式

下面演示一个使用缓存的例子：

前后两次执行测试，参数没变，源文件也没变化，第二次执行时会自动从缓存中获取结果，结果中“cached”即表示结果从缓存中获取。

### **禁用缓存**

测试时使用一个不在“可缓存参数”集合中的参数，就不会使用缓存，比较常用的方法是指定一个参数“-count=1”。

下面演示一个禁用缓存的例子：

E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src>go test gotest  
ok gotest 3.434s  
  
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src>go test gotest  
ok gotest (cached)  
  
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src>go test gotest -count=1  
ok gotest 3.354s

第三次执行使用了参数"-count=1"，所以执行时不会从缓存中获取结果。