Deterministic testing in Go

랜덤하게 실패하지 않는 테스트 방법

정겨울 / 당근

Speaker

정겨울

당근, Software Engineer

- me@winterjung.dev
- www.winterjung.dev
- x.com/res_tin
- 뱅크샐러드 Go 코딩 컨벤션

이번 발표는

아래 분들에게 도움이 돼요

- 테스트를 더 잘 짜고싶은 분들
- 고랭의 테스트 문법에 익숙하신 분들
- `time.Now()` 때문에 테스트가 어려우셨던 분들

이번 발표는

아래 내용을 얘기해요

- deterministic testing이란?
- 비 결정적 요소를 더 잘 테스트 하는 법
- 시간을 더 잘 테스트 하는 법
- 고루틴을 더 잘 테스트 하는 법
- (유닛 테스트에서)

이번 발표는

아래 내용은 다루지 않아요

- 테스트를 작성해야하는 이유
- 유닛 테스트와 통합 테스트, e2e 테스트의 차이
- db, 서버같은 외부 의존성을 테스트하는 법
- TDD

이번 발표를
스포일러 해보자면

- 의존성을 잘 주입해 쓰자는 얘기

아마 다들 해봤을 경험

```go
func newUser(name, email string) *user {
    return &user{
        id: uuid.New(),
        name: name,
        email: email,
        joinedAt: time.Now(),
    }
}
```

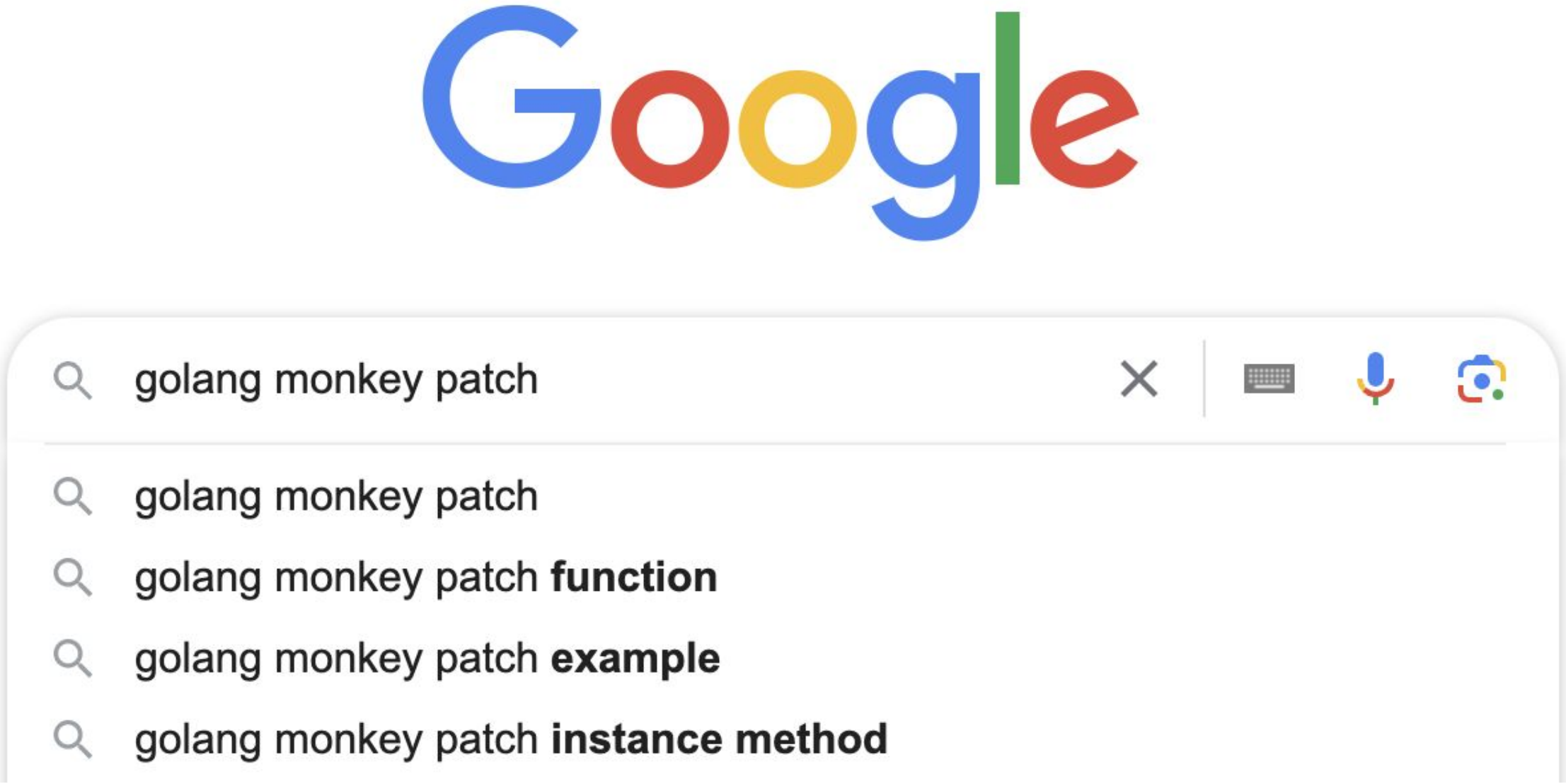
```go
func Test_newUser(t *testing.T) {
    u := newUser("test", "test@example.com")
    assert.Equal(t, "test", u.name)
    assert.Equal(t, "test@example.com", u.email)
    assert.Equal(t, /* ??? */, u.id)
    assert.Equal(t, /* ??? */, u.joinedAt)
}
```

```go
func sampling(rate float64) bool {
    // rand.Float64() returns half-open interval [0.0,1.0)
    // if rate is 0.0, never sample
    // if rate is 1.0, always sample
    return rand.Float64() < rate
}
```

```go
func TestPublisher_Publish(t *testing.T) {
    p := NewPublisher()
    evt := &Event{...}

    p.Publish(evt)
    // 이벤트가 flush 되길 기다림
    time.Sleep(100 * time.Millisecond)

    assert.Equal(t, 1, len(p.processedEvents))
}
```


Google
golang monkey patch
golang monkey patch
golang monkey patch **function**
golang monkey patch **example**
golang monkey patch **instance method**

몽키 패치

저도 좋아했는데..

- 최신 버전에선 안되거나
- 병렬 테스트에서 안되거나
- m1 mac 이상의 arm64 아키텍처에선 안되거나

Deterministic testing?

Non-deterministic testing

- 결과를 예측할 수 없는 테스트
- 네트워크 호출에 의존하거나: 언제나 실패할 수 있음
- 파일을 읽고 쓰거나: 실패할 수 있음
- 임의의 값을 사용하거나: 어떤 값이 생성될 지 (정해져 있지만) 우린 예측할 수 없음
- 동시에 실행하거나: 순서가 일관되지 않음

Flaky testing

왜 피해야 할까

- Non-deterministic 요소 때문에 때론 성공, 때론 실패하는 테스트
- 깨진 유리창: '원래 종종 ❌ 뜨고 실패하니까'
- 불필요하게 테스트 시간을 늘어뜨림
- 프로덕션 배포의 잠재 위협 요소

비 결정적 요소를 고정하기

랜덤 값을 사용할 땐

- 단순 샘플링 때문에 사용할 수도 있고 다양한 사례에서 랜덤 사용
- `rand.Seed`로 시드를 고정시켜볼 수 있었으나

랜덤 값을 사용할 땐

- 단순 샘플링 때문에 사용할 수도 있고 다양한 사례에서 랜덤 사용
- `rand.Seed`로 시드를 고정시켜볼 수 있었으나
- 1.20부턴 `rand.New(rand.NewSource(...))`로 반환된 rander를 써야함
- 결국 rander를 주입받아 쓰는게 속 편함

랜덤 값을 사용할 땐

- 단순 샘플링 때문에 사용할 수도 있고 다양한 사례에서 랜덤 사용
- `rand.Seed`로 시드를 고정시켜볼 수 있었으나
- 1.20부턴 `rand.New(rand.NewSource(...))`로 반환된 rander를 써야함
- 결국 rander를 주입받아 쓰는게 속 편함
- rander를 그대로 쓰기보단 id generator, logging decider처럼 그 역할을 인터페이스로 빼서 주입하길 권장

```go
// ❌
func sampling(rate float64) bool {
    return rand.Float64() < rate
}

// ✅
func sampling2(r rand.Rand, rate float64) bool {
    return r.Float64() < rate
}
```

```go
// ✅
func sampling3(r rand.Rand) func(float64) bool {
    return func(rate float64) bool {
        return r.Float64() < rate
    }
}

// ✅
func sampling4(randFn func() float64, rate float64) bool {
    return randFn() < rate
}
```

```go
// ✅
type sampler interface {
    Sample(float64) bool
}

type randSampler struct {
    randFn func() float64
}

func (s *randSampler) Sample(rate float64) bool {
    return s.randFn() < rate
}
```

```go
type neverSampler struct {}

func (s *neverSampler) Sample(float64) bool { return false }

type alwaysSampler struct {}

func (s *alwaysSampler) Sample(float64) bool { return true }
```

무언가를 생성할 땐

- 해시처럼 input으로 넣었던 인자의 expected를 계산하기 쉬운 경우도 있음

```go
func TestHash(t *testing.T) {
    hashed := sha256.Sum256([]byte("hello"))
    s := hex.EncodeToString(hashed[:])
    assert.Equal(t, "...", s)
}

// Error:
//     Not equal:
//     expected: "..."
//     actual  : "2cf24dba5fb0a30e...425e73043362938b9824"
```

무언가를 생성할 땐

- 문제는 uuid, nonce generator 류 함수
- new func든 factory든 생성 함수를 인자로 받기
- atomic을 사용하는 방법

```go
// ❌
func TestUUIDEventLogger(t *testing.T) {
    logger := NewEventLogger()

    logger.Log()
    // Output:
    // 8a18ead2-c292-4998-be08-ce0f1b5936c5
    // 2885f037-494e-4910-89fe-c7160ebf5e61
}

// ✅
func TestFixedEventLogger(t *testing.T) {
    logger := NewEventLogger(func() string {
        return "00000000-0000-0000-0000-123456789012"
    })

    logger.Log()
    // Output:
    // 00000000-0000-0000-0000-123456789012
}
```

```go
// ❌
func TestUUIDEventLogger(t *testing.T) {
    logger := NewEventLogger()

    logger.Log()
    // Output:
    // 8a18ead2-c292-4998-be08-ce0f1b5936c5
    // 2885f037-494e-4910-89fe-c7160ebf5e61
}

// ✅
func TestFixedEventLogger(t *testing.T) {
    logger := NewEventLogger(func() string {
        return "00000000-0000-0000-0000-123456789012"
    })

    logger.Log()
    // Output:
    // 00000000-0000-0000-0000-123456789012
}
```

```go
// ✅
func TestAtomicEventLogger(t *testing.T) {
    var cnt int32
    mockUUIDFunc := func() string {
        atomic.AddInt32(&cnt, 1)
        return
fmt.Sprintf("00000000-0000-0000-0000-%012d", cnt)
    }
    logger := NewEventLogger(mockUUIDFunc)
    logger.Log()
    // Output:
    // 00000000-0000-0000-0000-000000000001
    // 00000000-0000-0000-0000-000000000002
}
```

무언가를 생성할 땐

- 기존 코드 수정이 어렵다면 값 그 자체보다는 의도된 포맷인지 그 속성을 검사하는 방법도 존재 (e.g. 알파벳 조합인가, 정해진 길이인가)

```go
const charset = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"

func NewNonce() string {
    b := make([]byte, 16)
    for i := range b {
        b[i] = charset[rand.Intn(len(charset))]
    }
    return string(b)
}

func TestNewNonce(t *testing.T) {
    result := NewNonce() // Output: Imq61MBEGBVxXQ2l, eU1XBzYOqUFlTQeL

    assert.Len(t, result, 16)
    for _, r := range result {
        assert.Contains(t, charset, string(r))
    }
}
```

그 외 flaky test 피하기

- map을 순회할 때 순서가 보장되지 않음
- slice에 저장하고 sorting 하거나
- 아예 slice 자체를 돌며 map은 보조 룩업용으로만 쓰거나

```go
// ❌
func unstableUniq(all []string) []string {
    uniq := make(map[string]bool)
    for _, k := range all {
        uniq[k] = true
    }

    keys := make([]string, 0)
    for k := range uniq { // unstable
        keys = append(keys, k)
    }

    return keys
}
```

```go
// ✅
func stableSortUniq(all []string) []string {
    uniq := make(map[string]bool)
    for _, k := range all {
        uniq[k] = true
    }

    keys := make([]string, 0)
    for k := range uniq {
        keys = append(keys, k)
    }
    sort.Strings(keys) // stable
    return keys
}
```

```go
// ✅
func stableUniq(all []string) []string {
    keys := make([]string, 0)
    uniq := make(map[string]bool)
    for _, k := range all {
        if uniq[k] {
            continue
        }
        uniq[k] = true
        keys = append(keys, k)
    }

    return keys
}
```

그 외 flaky test 피하기

- 고루틴을 쓸 때 실행 순서가 보장되지 않음
- 값을 모두 수신할 수 있는 채널 여러개를 select 할 땐 랜덤: e.g. quit 시그널 받았을 때 채널을 drain 해주기
- protojson, prototext의 특이사항. marshal된 값은 그때그때 다르다 `{“a”: “b”}`일 수도 `{“a”:“b”}`일 수도

```go
// https://github.com/golang/protobuf/issues/1121
func mustMarshalJSON(m proto.Message) []byte {
    marshaler := protojson.MarshalOptions{}

    b, err := marshaler.Marshal(m)
    if err != nil {
        panic(err)
    }
    return b
}
```

```go
func TestPublishedProtoEvent(t *testing.T) {
    event := &proto.Event{
        Name: "hello",
    }
    publishedEvent := publish(event)
    // ❌
    assert.Equal(t, `{"source": {"name": "hello"}}`, publishedEvent)
    // ✅
    assert.JSONEq(t, `{"source": {"name": "hello"}}`, publishedEvent)
    assert.Equal(t, mustMarshalJSON(&proto.PublishedEvent{
        Source: &proto.Event{
            Name: "hello",
        },
    }), publishedEvent)
}
```

시간에 구애받지 않는 테스트

시간을 테스트 할 땐

- 내부적으로 `time.Now()`를 쓰는 `time.Since(t)`, `time.Until(t)`은 피하고
- `time.Now` 대신 time func, now func을 인자로 전달받아 쓰기

```go
// ❌
func isExpired(t time.Time) bool {
    return t.Before(time.Now())
}

// ✅
func isExpired(t, now time.Time) bool {
    return t.Before(now)
}
```

```go
func handler(db *sql.DB, nowFunc func() time.Time) handlerFunc {
    return func(ctx context.Context, r http.Request) (http.Response, error) {
        token := getTokenFromDB(db)
        if isExpired(token.Expiry, nowFunc()) {
            // ...
        }
    }
}

func TestHandler(t *testing.T) {
    // ...
    mockNow := func() time.Time {
        return time.Date(2024, 7, 13, 0, 0, 0, 0, time.UTC)
    }
    resp, err := handler(mockDB, mockNow)(ctx, req)
}
```

```go
func TestExponentialBackoff(t *testing.T) {
    // Given
    // 실행 횟수에 따라 의도된 backoff 시간이 나오는지 검증하기 위한 sleep 함수
    var count int32
    sleepFunc := func() func(time.Duration) {
        expectedIntervals := []time.Duration{
            1 * time.Second, 2 * time.Second,
            4 * time.Second, 8 * time.Second,
        }
        return func(d time.Duration) {
            assert.Equal(t, expectedIntervals[count], d)
            count++
        }
    }()
    // ...
}
```

```go
func TestExponentialBackoff(t *testing.T) {
    // Given
    srv := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
        assert.Equal(t, "/users/1", r.URL.Path)
        w.WriteHeader(http.StatusServiceUnavailable)
    }))
    t.Cleanup(srv.Close)

    cli := NewUserServiceClient(
        srv.URL, // host
        sleepFunc, // sleepFunc
    )
    // ...
}
```

```go
func TestExponentialBackoff(t *testing.T) {
    // ...
    // When
    ctx := context.Background()
    resp, err := cli.GetUser(ctx, &GetUserRequest{
        UserID: 1,
    })

    // Then
    assert.EqualError(t, err, "503: Service Unavailable")
    assert.Equal(t, 5, count)
}
```

시간을 테스트 할 땐

- sleep, ticker, timer, after 등 좀 더 복잡해지면 jonboulle/clockwork 라이브러리처럼 clock 인터페이스를 정의해 사용하길 권장

```go
type Clock interface {
    After(d time.Duration) <-chan time.Time
    Sleep(d time.Duration)
    Now() time.Time
    Since(t time.Time) time.Duration
    NewTicker(d time.Duration) Ticker
    NewTimer(d time.Duration) Timer
    AfterFunc(d time.Duration, f func()) Timer
}
```

타임아웃 테스트

- 요청이 10초 이상 걸리면 취소하고 예전 stale 응답을 반환하는 예제
- 테스트에서 10초를 기다릴 순 없음
- context는 항상 상위 scope에서 전달 받아야하고
- 테스트 코드에선 `context.WithTimeout(ctx, 0)`으로 이미 타임아웃된 요청을 넘기는 방법

고루틴 잘 테스트하기

고루틴 타이밍

- send email 예제

```go
func handler(cli emailClient) {
    // ...
    go sendEmail(cli, newUser)
}

func TestIsEmailSent(t *testing.T) {
    cli := &mockEmailClient{}
    handler(cli)
    time.Sleep(100 * time.Millisecond)
    assert.Len(t, cli.sentEmails, 1)
}
```

고루틴 타이밍

- fire and forgot보다 더 관리의 영역으로 둬야함
- `runtime.Gosched()`로도 실행을 보장할 수 없음
- testify의 `assert.Eventually` 함수를 사용하거나
- 전달한 의존성의 채널을 소비하거나 wait group을 만료시키는 방식으로

```go
func TestIsEmailSent(t *testing.T) {
    cli := &mockEmailClient{}
    handler(cli)

    assert.Eventually(t, func() bool {
        return cli.sentEmails > 0
    }, time.Second, 100*time.Millisecond)
}
```

```go
func (c *mockEmailClient) SendEmail(title, body string) {
    c.sentEmails = append(c.sentEmails, title)
    c.sent <- struct{}{}
}

func TestIsEmailSent(t *testing.T) {
    cli := &mockEmailClient{sent: make(chan struct{})}
    handler(cli) // go sendEmail(cli, newUser) 수행
    <-cli.sent
    assert.Len(t, cli.sentEmails, 1)
}
```

고루틴 타이밍

- 고루틴 실행 순서에 민감하다면 go 키워드, sync/errgroup, sync.WaitGroup 대신 Group같은 인터페이스를 선언해 메커니즘 자체를 의존성으로 사용

```go
type Group interface {
    Go(f func() error)
    Wait() error
}

// using sync.WaitGroup, golang.org/x/sync/errgroup
type syncGroup struct {}

// for testing
type sequentialGroup struct {}
```

```go
func handler(g Group) {
    g.Go(func() error {
        return nil
    })
    if err := g.Wait(); err != nil {
        // ...
    }
}
```

고루틴 타이밍

- fanout 결과를 담을 땐 mutex + append, 채널보단 정해진 인덱스에 assign 하는 방식으로
- 로직에 따라 zero value 필터링이 필요해질 수 있음

```go
// ❌
func TestFanOutWrongWay(t *testing.T) {
    var wg sync.WaitGroup
    result := make([]int, 0)
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func(i int) {
            defer wg.Done()

            result = append(result, i)
        }(i)
    }
    wg.Wait()

    assert.Len(t, result, 10) // 10개가 아님
}
```

```go
// ⚠️
func TestFanOutNonDeterministic(t *testing.T) {
    var mu sync.Mutex
    var wg sync.WaitGroup
    result := make([]int, 0)
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func(i int) {
            defer wg.Done()
            mu.Lock()
            defer mu.Unlock()
            result = append(result, i)
        }(i)
    }
    wg.Wait()

    assert.Len(t, result, 10) // [1 9 5 6 7 8 0 2 3 4]
}
```

```go
// ✅
func TestFanOut(t *testing.T) {
    var wg sync.WaitGroup
    result := make([]int, 10)
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func(i int) {
            defer wg.Done()

            result[i] = i
        }(i)
    }
    wg.Wait()

    assert.Len(t, result, 10) // [0 1 2 3 4 5 6 7 8 9]
}
```

Flaky test 탐지하기

Flaky test 탐지하기

- 가끔 github actions가 실패했는데 re-run하니 성공하는 식
- `go test -count 10` 혹은 `100` 하다보면 관측됨
- 1.17부턴 `go test -shuffle on` 옵션으로 더 평소에 발견해볼 수 있음

Flaky test 탐지하기

- 수정이 어렵다면 gotestyourself/gotestsum같은 도구로 테스트 retry를 자동으로 시도해볼 수 있음
- 테스트에 불안정함을 표시할 수 있는 기능, 자동으로 재시도하는 기능관련 제안(golang/go#62244)이 수락되어 미래 릴리즈 버전에 자체 구현될수도 있음

cmd/go: add support for dealing with flaky tests #62244

Open **bradfitz** opened this issue on Aug 24, 2023 · 36 comments

bradfitz commented on Aug 24, 2023 • edited Contributor

Background

First off, flaky tests (a test that usually passes but sometimes fails) are the worst and you should not write them and fix them immediately.

That said, the larger teams & projects & tests get, the more likely they get introduced. And sometimes it's technically and/or politically hard to get them fixed. (e.g. the person who wrote it originally left the company or works on a new team) You're then left deciding whether to skip/delete the entire test (which might be otherwise super useful), or teach your team to become immune to and start ignoring test failures, which is the worst, when teams start submitting when CI's red, becoming blind to real test failures.

Google doesn't have this problem internally because Google's build system supports detecting & annotating flaky tests: https://bazel.build/reference/be/common-definitions

Tailscale has its own test wrapper that retries tests annotated with `flakytest.Mark` as flaky. We can't use `go test -exec=...` unfortunately, as that prevents caching (#27207). So instead we need a separate tool wraps `cmd/go` (kinda awkwardly, as it turns out).

Assignees

No one assigned

Labels

GoCommand Proposal Proposal-Accepted

Projects

Proposals

Status: Accepted

Milestone

Backlog

여기까지 드렸던 얘기들

한 줄로 요약해보자면

- 의존성은 인자로 잘 넘겨 쓰자는 얘기

이번 발표가

아래 분들에게 도움이 됐다면 좋겠습니다

- 테스트를 더 잘 짜고싶은 분들
- 고랭의 테스트 문법에 익숙하신 분들
- `time.Now()` 때문에 테스트가 어려우셨던 분들

Q&A

당근 홍보 살짝 👉 ml 데이터 플랫폼 팀 / 피드 인프라 팀

Thank you!