Ошибки при выполнении групповых задач

В этом модуле мы познакомимся с **golang.org/x/sync/errgoup** (и вскользь с его аналогами), позволяющим упростить написание кода по конкурентной обработке групповых задач.

Будем посмотреть.

golang.org/x/sync/errgoup

Одна из главных фишек Go – это легкость в написании конкурентного кода.

Нередко перед разработчиком возникает необходимость решить некоторую задачу группой воркеров (такие задачи мы и называем **групповыми**), синхронизируя их с помощью `sync.WaitGroup`.

Более того, часто оказывается так, что результат выполняемых действий сам по себе нам неинтересен, а интересен только факт, выполнились ли они все успешно или нет (т.е. если хотя бы одна из конкурентных операций завершилась с ошибкой, то мы не смотрим на результат работы остальных воркеров).

Для решения подобной проблемы может подойти следующий код (исходник примера):

```
// https://goplay.tools/snippet/6TuIa2Z-Az6

import (
    "errors"
    "fmt"
    "sync"
)

func main() {
    if err := work(); err != nil {
        fmt.Println(err) // something bad has happened
    }
}

func work() error {
    // Будем выполнять два параллельных действия.
    var wg sync.WaitGroup
    errsCh := make(chan error, 2)

    wg.Add(1)
    go func() {
        defer wg.Done()
        // Выполняем какую-то операцию, завершившуюся с ошибкой.
        // ...
        errsCh <- errors.New("something bad has happened")
    }()

    wg.Add(1)
    go func() {
        defer wg.Done()
        // Выполняем какую-то операцию, завершившуюся без ошибки.
        // ...
    }()

    wg.Wait() // Дожидаемся окончания работ.

    // Возвращаем ошибку от любой из операций (если ошибка
произошла).
    select {
    case err := <-errsCh:
        return err
    default:
```

```
        return nil
    }
}
```

Здесь на арену выходит модуль **golang.org/x/sync/errgoup**, позволяющий сильно упростить код выше (исходник примера):

```
// https://goplay.tools/snippet/084Moc7AiwV
import (
    "errors"
    "fmt"

    "golang.org/x/sync/errgroup"
)

func main() {
    if err := work(); err != nil {
        fmt.Println(err) // something bad has happened
    }
}

func work() error {
    var eg errgroup.Group

    eg.Go(func() error {
        // Выполняем какую-то операцию, завершившуюся с ошибкой.
        // ...
        return errors.New("something bad has happened")
    })

    eg.Go(func() error {
        // Выполняем какую-то операцию, завершившуюся без ошибки.
        // ...
        return nil
    })

    // Дожидаемся окончания работ.
    // Возвращаем ошибку от любой из операций (если ошибка
произошла).
    return eg.Wait()
```

```
}
```

Выглядит неплохо.

Осталось глянуть на внутреннее устройство **errgroup** и решить парочку задач.

Препарируем golang.org/x/sync/errgoup

Если посмотреть на реализацию `errgroup.Group`, то можно увидеть, что она достаточно простая:

type Group

- func WithContext(ctx context.Context) (*Group, context.Context)
- func (g *Group) Go(f func() error)
- func (g *Group) Wait() error

errgroup.WithContext

Конструктор, позволяющий нам создать новую группу на основе контекста:

```
// WithContext returns a new Group and an associated Context derived
from ctx.
//
// The derived Context is canceled the first time a function passed
to Go
// returns a non-nil error or the first time Wait returns, whichever
occurs
// first.
```

```
func WithContext(ctx context.Context) (*Group, context.Context) {
    ctx, cancel := context.WithCancel(ctx)
    return &Group{cancel: cancel}, ctx

}
```

Возвращаемый контекст отменяется, когда одна из функций, переданных в метод `(*error.Group).Go`, завершается с ошибкой. Как этим можно пользоваться – узнаем в следующих шагах.

errgroup.Group

Основной тип модуля:

```
type Group struct {
    cancel func() // Функция, отменяющая контекст группы.

    wg sync.WaitGroup // Группа ожидания для синхронизации горутинок из Go.

    errOnce sync.Once // Чтобы единожды записать первую ошибку и отменить контекст.
    err     error     // Первая возникшая ошибка.

}
```

(*errgroup.Group).Go

Самая "сложная" функция из пакета:

- В качестве аргумента принимает некую функцию `f`, которая возвращает ошибку. Это как раз действие, которое мы и хотим выполнить.
- Вызов `f` происходит в отдельной горутине. Предварительно увеличивается счетчик запущенных горутин в `wg` на единицу.
- Если `f` возвращает ошибку, то сохраняем её и отменяем контекст, вызывая тот самый `cancel`. Причем делаем это один раз через `errOnce` – берём на заметку пример использования `sync.Once`. Сохранённую ошибку мы вернём в `(*error.Group).Wait`.

```
// Go calls the given function in a new goroutine.
//
// The first call to return a non-nil error cancels the group; its
error will be
// returned by Wait.
func (g *Group) Go(f func() error) {
    g.wg.Add(1)

    go func() {
        defer g.wg.Done()

        if err := f(); err != nil {
            g.errOnce.Do(func() {
                g.err = err
                if g.cancel != nil {
                    g.cancel()
                }
            })
        }
    }()

}
```

Из кода выше можно сделать вывод, что ошибка от выполняемой функции записывается единожды и **не перезаписывается** остальными функциями. Очень часто, по незнанию, разработчики ожидают там последнюю случившуюся ошибку (исходник примера):

```
// https://goplay.tools/snippet/fAqjkbxg5JF

func main() {
    var eg errgroup.Group

    eg.Go(func() error {
        return errors.New("first error")
    })

    eg.Go(func() error {
        time.Sleep(time.Second)
        return errors.New("second error")
    })

    eg.Go(func() error {
        time.Sleep(2 * time.Second)
        return errors.New("third error")
```

```
    })

    fmt.Println(eg.Wait()) // first error
}
```

(*errgroup.Group).Wait

Позволяет дождаться выполнения всех действий, запущенных в горутинах, и получить первую возникшую ошибку, если она, конечно, была:

```
// Wait blocks until all function calls from the Go method have
returned, then
// returns the first non-nil error (if any) from them.
func (g *Group) Wait() error {
    g.wg.Wait()
    if g.cancel != nil {
        g.cancel()
    }
    return g.err

}
```

Также на всякий случай (если ни одна из функций не вернула ошибку и контекст не был отменён в `(*errgroup).Go`) отменяется контекст группы.

Тест "Основы errgroup - 1"

Постарайтесь, не запуская программы, ответить – **Что будет выведено на экран?**

```
import (
    "fmt"

    "golang.org/x/sync/errgroup"
)

func main() {
    var eg errgroup.Group

    eg.Go(func() error {
        fmt.Println("1")
        return nil
    })
```

```
    eg.Go(func() error {
        fmt.Println("2")
        return nil
    })

    fmt.Println("3")

}
```

Выберите один вариант из списка

Ничего
123
213
321
Вывод недетерминирован
3

Тест "Основы errgroup - 2"

Постарайтесь, не запуская программы, ответить – **Что будет выведено на экран?**

```
import (
    "errors"
    "fmt"

    "golang.org/x/sync/errgroup"
)

var (
    errNotFound     = errors.New("not found")
    errUnauthorized = errors.New("unauthorized")
    errUnknown      = errors.New("unknown error")
)

func main() {
    var eg errgroup.Group

    eg.Go(func() error { return errNotFound })
    eg.Go(func() error { return errUnauthorized })
    eg.Go(func() error { return errUnknown })
```

```
    switch err := eg.Wait(); {
    case errors.Is(err, errNotFound):
        fmt.Println("1")

    case errors.Is(err, errUnauthorized):
        fmt.Println("2")

    case errors.Is(err, errUnknown):
        fmt.Println("3")
    }

}
```

Выберите один вариант из списка

Ничего
1
2
3
Вывод недетерменирован

errgroup.WithContext

Возникает вопрос, как лучше создавать `errgroup.Group`:

```
// Просто.
var eg errgroup.Group
// eg := errgroup.Group{}
// eg := new(errgroup.Group)

// Или на базе контекста?
eg, ctx := errgroup.WithContext(ctx)
```

Рассмотрим пример.

В нём нам необходимо сходить по N ссылкам в M горутинах и получить данные по ним, используя вспомогательную функцию. При этом, если случается хотя бы одна ошибка, то необходимо прервать выполнение программы, а не продолжать обрабатывать оставшиеся ссылки.

Основная проблема здесь состоит в том, чтобы грамотно завершить все горутины при ошибке хотя бы в одной из них. Ведь по умолчанию `errgroup.Group` запишет ошибку, но в `(*errgroup).Wait` мы **заблокируемся**, пока не закончат свою работу все воркеры (т.е. пока не будут обработаны все ссылки).

Здесь на помощь приходит работа с контекстом как в горутинах-писателях, так и горутинах-читателях, а также создание группы через `errgroup.WithContext`.

Резюме

~~Контекст рулет!~~

Создание `errgroup` не на основе контекста может привести к неприятным ошибкам вплоть до **заблокированного** приложения (особенно актуально в случае использования неопытными разработчиками голого `time.Sleep`).

Если же вы не хотите, чтобы остановка работы одного из воркеров влияла на другие воркеры, то можете не использовать `errgroup.WithContext`, но обязательно "сидите" в горутинах на другом контексте!

Пример на это утверждение.

Тест "errgroup.WithContext"

Постарайтесь, не запуская программы, ответить – **Что будет выведено на экран?**

```
import (
    "context"
    "errors"
    "fmt"
    "time"

    "golang.org/x/sync/errgroup"
)
```

```
func main() {
    ctx, cancel := context.WithCancel(context.Background())
    go func() {
        select {
        case <-ctx.Done():
            fmt.Print("5")
        case <-time.After(3 * time.Second):
            cancel()
        }
    }()

    eg, egCtx := errgroup.WithContext(ctx)

    eg.Go(func() error {
        select {
        case <-egCtx.Done():
            fmt.Print("1")

        case <-time.After(10 * time.Millisecond):
            return errors.New("2")
        }

        return nil
    })

    eg.Go(func() error {
        select {
        case <-egCtx.Done():
            fmt.Print("3")

        case <-time.After(time.Second):
            fmt.Print("4")
        }

        return nil
    })

    fmt.Print(eg.Wait())

}
```

Напишите текст

_____

Тест "errgroup и отмена контекста"

Постарайтесь, не запуская программы, ответить – **Что будет выведено на экран?**

```
package main

import (
    "context"
    "fmt"

    "golang.org/x/sync/errgroup"
)

func main() {
    ctx := context.TODO()
    action(ctx, "initial action")

    g, ctx := errgroup.WithContext(ctx)

    g.Go(func() error {
        action(ctx, "concurrent action 1")
        return nil
    })
    g.Go(func() error {
        action(ctx, "concurrent action 2")
        return nil
    })

    _ = g.Wait()
    action(ctx, "final action")
}

func action(ctx context.Context, s string) {
    if ctx.Err() != nil {
        fmt.Println(ctx.Err())
        return
    }
    fmt.Println(s)
}
```

Не спешите "угадывать" ответ случайным протыкиванием. **Понимание этого примера крайне важно!**

Выберите один вариант из списка

- ☐ initial action
  concurrent action 1
  concurrent action 2
  final action

- ☐ initial action
  concurrent action 2
  concurrent action 1
  final action

- ☐ initial action
  "concurrent action 1" и "concurrent action 2" в неопределённом порядке
  final action

- ☐ initial action
  concurrent action 1
  concurrent action 2
  context canceled

- ☐ initial action
  concurrent action 2
  concurrent action 1
  context canceled

- ☐ initial action
  "concurrent action 1" и "concurrent action 2" в неопределённом порядке

context canceled

Больше примеров

Вдохновение можно (и нужно) черпать не только из наших примеров, но и из официальной документации **golang.org/x/sync/errgoup**:

- errgroup.Group: Examples: JustErrors – фетчим URL'ы в разных горутинках.
- errgroup.Group: Examples: Parallel – синхронизируем параллельный поиск `query` в разных категориях.
- errgroup.Group: Examples: Pipeline – самый интересный пример: реализуем функцию `MD5All`, считающую хеши файлов в директории с помощью ограниченного числа воркеров.

Задача "errgroup: заполнение структуры"

Ссылка на заготовку.

golang.org/x/sync/errgoup частенько используют для того, чтобы конкуретно наполнить поля структуры данными.

Дана структура `Order`:

```
type Order struct {
    BuyerName    string
    BuyerAddress string
    SellerName   string

}
```

Её наполнением занимается функция `GetOrder`, которую, собственно, и надо реализовать:

```
type StringValueGetter func(ctx context.Context) (string, error)

func GetOrder(
    ctx context.Context,
```

```
    getBuyerName, getBuyerAddress, getSellerName StringValueGetter,

) (Order, error)
```

К сожалению, мы не можем проверить эту задачу средствами Stepik, поэтому она остаётся на самостоятельное изучение, совесть и желание студента:

```
$ cd
tasks/07-working-with-errors-in-concurrency/errgroup-structure-fillin
g
$ go test -race -v  .
=== RUN   TestGetOrder
=== RUN   TestGetOrder/no_errors
=== RUN   TestGetOrder/buyer_name_error
=== RUN   TestGetOrder/slow_buyer_name
=== RUN   TestGetOrder/slow_buyer_name_and_error
=== RUN
TestGetOrder/total_functions_execution_time_greater_than_test_timeout
--- PASS: TestGetOrder (0.16s)
    --- PASS: TestGetOrder/no_errors (0.00s)
    --- PASS: TestGetOrder/buyer_name_error (0.00s)
    --- PASS: TestGetOrder/slow_buyer_name (0.10s)
    --- PASS: TestGetOrder/slow_buyer_name_and_error (0.00s)
    --- PASS:
TestGetOrder/total_functions_execution_time_greater_than_test_timeout
(0.06s)
PASS
ok
tasks/07-working-with-errors-in-concurrency/errgroup-structure-fillin
g      0.436s
```

Чтобы увидеть авторское решение просто нажмите "Отправить". Чтобы опубликовать своё решение, вставьте кодовую вставку в комментарий к оставленному решению (по аналогии с авторским).

P.S. Подумайте, не нужны ли здесь ещё какие-нибудь примитивы синхронизации для конкурентной работы с полями структуры?

Задача "errgroup: обработка очереди задач"

Ссылка на заготовку.

Наша цель – реализовать обработчик очереди задач, используя **golang.org/x/sync/errgoup**.

Обработчиком очереди будем считать функцию `Run`:

```
// Run выполняет задачи из очереди tasks с некоторыми условиями:
// - параллельно обрабатываются workersCount задач;
// - если workersCount <= 0, то функция возвращает
ErrInvalidWorkersCount;
// - для обработки задачи Task вызывается функция process;
// - если во время работы process возникла ошибка ErrFatal, то
обработка очереди
//   завершается с возвратом этой ошибки;
// - при любой другой ошибке обработка очереди продолжается.

func Run(ctx context.Context, workersCount int, tasks <-chan Task)
error
```

Под капотом `Run` вызывает вспомогательную функцию `process`:

```
// process выполняет задачу task с некоторыми условиями:
// - задача task выполняется не дольше Task.ExecutionTimeout();
// - если во время выполнения задачи возникает ошибка, то она
возвращается наружу;
// - при возникновении паники функция возвращает ErrFatal.

func process(ctx context.Context, task Task) (err error)
```

Очередь представлена в виде канала `tasks`, в который попадают экземпляры интерфейса `Task`:

```
type Task interface {
    // Handle выполняет задачу.
    Handle(ctx context.Context) error

    // ExecutionTimeout возвращает промежуток времени,
    // в течение которого задача должна быть выполнена.
    ExecutionTimeout() time.Duration
```

```
}
```

`Task` может сам себя выполнить, а также может сказать о том, сколько времени обычно отводится ему на выполнение.

Больше подробностей, как всегда, в наших тестах.

К сожалению, мы не можем проверить эту задачу средствами Stepik, поэтому она остаётся на самостоятельное изучение, совесть и желание студента:

```
$ cd tasks/07-working-with-errors-in-concurrency/errgroup-task-runner
$ go test -race -v .
...
--- PASS: TestRun (8.50s)
    --- PASS: TestRun/invalid_workers_count (0.00s)
    --- PASS: TestRun/no_workers (0.00s)
    --- PASS: TestRun/1_worker_X_60_tasks_(50ms)_=_3s (3.25s)
    --- PASS: TestRun/2_workers_X_60_tasks_(50ms)_=_1.5s (1.63s)
    --- PASS: TestRun/60_workers_X_600_tasks_(50ms)_=_500ms (0.54s)
    --- PASS:
TestRun/16_workers,_200_long_tasks,_cancellation_by_root_ctx (1.01s)
    --- PASS:
TestRun/16_workers,_16_long_tasks,_cancellation_by_self_timeout
(1.01s)
    --- PASS: TestRun/16_workers,_15_long_tasks_&_1_task_with_panic
(1.01s)
    --- PASS: TestRun/16_workers,_16_errored_tasks,_no_error_from_Run
(0.05s)
PASS

ok
tasks/07-working-with-errors-in-concurrency/errgroup-task-runner
8.629s
```

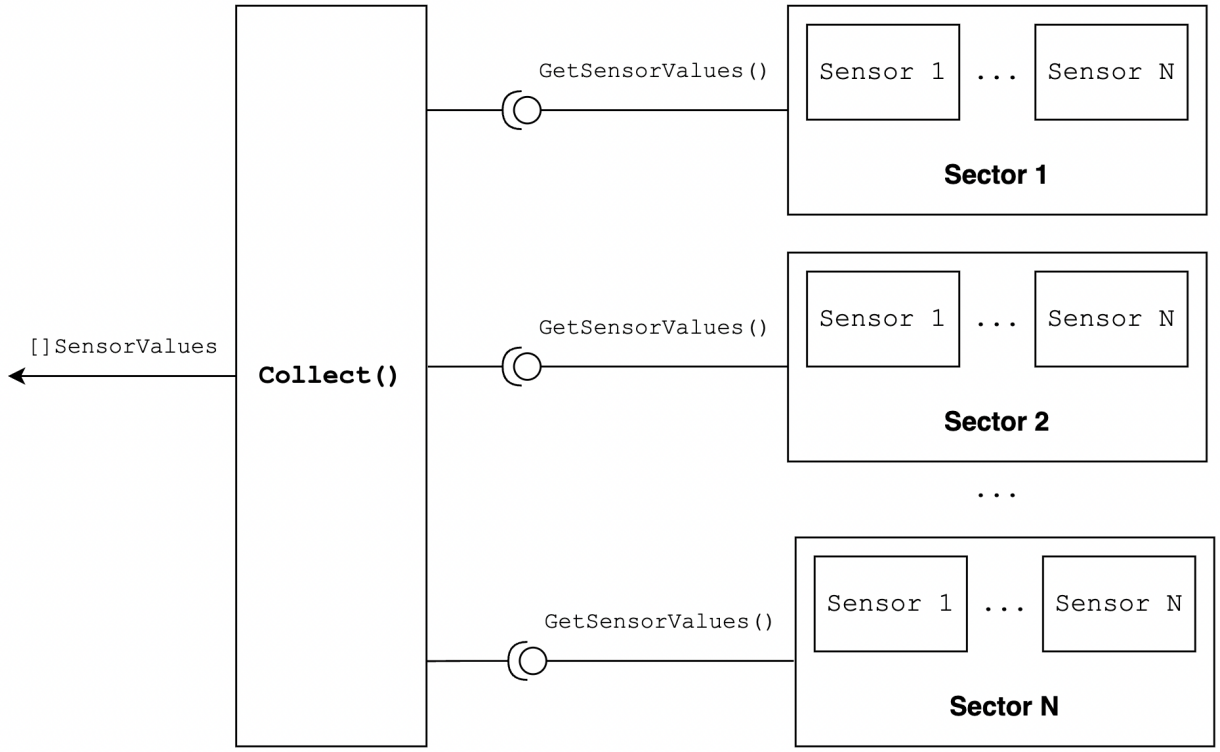
Задача "errgroup: Collect"

Ссылка на заготовку.

Вам необходимо реализовать функцию `Collect`:

```
var ErrTooMuchSectors = errors.New("too much sectors")

type SensorValue struct {
    SensorID string
    Value    float64
}

type Sector interface {
    ID() string
    GetSensorValues(ctx context.Context) ([]SensorValue, error)
}

// Collect конкурентно собирает значения датчиков с секторов,
объединяет их в один
// слайс данных и выдаёт наружу.
// При превышении лимита на количество секторов возвращает ошибку
ErrTooMuchSectors.
// При возникновении ошибки во время опроса очередного сектора,
функция завершает
// свою работу и возвращает эту ошибку.

func Collect(ctx context.Context, sectors []Sector) ([]SensorValue,
error)
```

Схематично работу функции можно представить следующим образом:



Так же стоит отметить, что функция не может обрабатывать более чем 10 секторов:

```
const maxSectors = 10
```

К сожалению, мы не можем проверить эту задачу средствами Stepik, поэтому она остаётся на самостоятельное изучение, совесть и желание студента:

```
$ cd tasks/07-working-with-errors-in-concurrency/errgroup-collector
$ go test -v -race ./...
=== RUN   TestCollect
=== RUN   TestCollect/no_sectors_no_problems_1
=== RUN   TestCollect/no_sectors_no_problems_2
=== RUN   TestCollect/too_much_sectors
=== RUN   TestCollect/one_fast_sector
=== RUN   TestCollect/one_slow_sector
=== RUN   TestCollect/three_sectors_with_the_same_data
=== RUN   TestCollect/ten_sectors_with_diff_polling_time
=== RUN   TestCollect/poll_sensors_error
=== RUN   TestCollect/collect_timeout
=== RUN   TestCollect/large_amount_of_data
```

```
=== RUN   TestCollect/error_while_large_amount_of_data
--- PASS: TestCollect (5.78s)
    --- PASS: TestCollect/no_sectors_no_problems_1 (0.00s)
    --- PASS: TestCollect/no_sectors_no_problems_2 (0.00s)
    --- PASS: TestCollect/too_much_sectors (0.00s)
    --- PASS: TestCollect/one_fast_sector (0.00s)
    --- PASS: TestCollect/one_slow_sector (1.01s)
    --- PASS: TestCollect/three_sectors_with_the_same_data (0.51s)
    --- PASS: TestCollect/ten_sectors_with_diff_polling_time (1.01s)
    --- PASS: TestCollect/poll_sensors_error (1.01s)
    --- PASS: TestCollect/collect_timeout (1.01s)
    --- PASS: TestCollect/large_amount_of_data (0.23s)
    --- PASS: TestCollect/error_while_large_amount_of_data (1.01s)
PASS
ok
tasks/07-working-with-errors-in-concurrency/errgroup-collector
6.070s
```

multierror.Group

В начале данного урока мы говорили о том, что `errgroup.Group` сохраняет первую случившуюся ошибку (через `sync.Once`) и остальные ошибки её уже не интересуют.

А что если мы хотим накапливать ошибки и иметь возможность работать с ошибками из каждой горутины?

Здесь на помощь придёт **github.com/hashicorp/go-multierror**, с которым мы познакомились на уроке "**Прочие нестандартные модули**", а именно тип multierror.Group (исходник примера):

```
// https://goplay.tools/snippet/vO4-a9-XALB
import (
    "errors"
    "fmt"

    "github.com/hashicorp/go-multierror"
)
```

```
var (
    errNotFound     = errors.New("not found")
    errUnauthorized = errors.New("unauthorized")
    errUnknown      = errors.New("unknown error")
)

func main() {
    var eg multierror.Group

    eg.Go(func() error { return errNotFound })
    eg.Go(func() error { return errUnauthorized })
    eg.Go(func() error { return errUnknown })

    err := eg.Wait()
    fmt.Println(errors.Is(err, errNotFound))     // true
    fmt.Println(errors.Is(err, errUnauthorized)) // true
    fmt.Println(errors.Is(err, errUnknown))      // true
}
```

По сути это что-то среднее между `sync.WaitGroup` и `errgroup.Group`.

Поддержка контекста отсутствует (да и не очень понятно, нужна ли она и какое поведение ожидается).