Передача ошибок между горутинами

Сама по себе конкурентность и как её готовить в Go темы интересные, но они в основном про работу с памятью, синхронизацию горутин, различные **concurrency patterns** и так далее.

Мы же с вами говорим про ошибки и их обработку в конкурентном коде, и здесь возникает два краеугольных вопроса: нет, не "Что делать?" и "Кто виноват?", а "**Кто обрабатывает ошибки?**" и "**Как доставить ошибки к месту обработки?**".

Кто обрабатывает ошибки?

Поясним, что вообще имеется ввиду под этим вопросом.

Представим типичную, казалось бы, ситуацию. Есть горутина-прораб, запускающая N горутин-воркеров, ожидая, что они выполнят полезную работу.

Полезная работа в примере ниже – это то, чем большинство из нас занимается, а именно перекладывание котиков, сериализованных в JSON, туда и обратно (исходник примера):

```
// https://goplay.tools/snippet/fsUMwLj3_lk

type Cat struct {
    Name string `json:"name"`
}

func main() {
```

```
    catsJSONs := []string{`{"name": "Bobby"}`, `"name": "Billy"`,
`{"name": "Васёк"}`}
    catsCh := make(chan Cat)

    var wg sync.WaitGroup
    wg.Add(len(catsJSONs))

    for _, catData := range catsJSONs {
        go func(catData string) { // Разбираем котиков в несколько
горутин.
            defer wg.Done()

            var cat Cat
            if err := json.Unmarshal([]byte(catData), &cat); err !=
nil {
                // Случилась ошибка, что делать?
            }
            catsCh <- cat
        }(catData)
    }

    go func() {
        wg.Wait()
        close(catsCh)
    }()

    for cat := range catsCh {
        fmt.Println(cat)
    }
}
```

Всё бы хорошо, но может произойти ошибка, если нам подсунут невалидного с точки зрения JSON котика.

И тут мы подошли к сути: **если случилась ошибка, то кто её обрабатывает**?

Варианта у нас два:

- ошибку обрабатывает горутина-воркер прямо в месте возникновения ошибки;

- ошибку обрабатывает родительская горутина (в данном случае `main`) в удобном ей месте.

Не будем сходу авторитетно одобрять один из вариантов и отрицать другой, а сравним их.

Обработка ошибки в месте её возникновения (in-place)

Прямо на месте ошибку удобно обработать, потому что на первый взгляд в месте её возникновения мы обладаем наибольшим контекстом случившегося. Чем-то это напоминает структурированное логирование ошибок, помните?

Например, мы можем попытаться выполнить операцию ещё раз или прервать свою работу или просто проигнорировать ошибку и т.д.

Обработка ошибки не в месте её возникновения

Но бывают ситуации, когда рядовой горутине не стоит пытаться принимать решение, является ли случившаяся ошибка критической для неё или нет. Что же делать? Нужно дать знать "родительской" горутине о том, что случилась ошибка, и глядишь, батя уже разберётся, что с ней делать и как красиво обработать ошибку в контексте всей задачи в целом.

Резюме

Обработка ошибок в конкурентном коде по своей сути не отличается от обработки в последовательном:

Ошибку нужно обработать в ближайшем возможном месте к месту возникновения,

но таком, где хватает контекста и полномочий для этого.

Весь вопрос заключается в том, как при необходимости передать ошибку между горутинами.

Техники, чтобы проворачивать подобное, мы и обсудим далее.

Тест "Какой вывод в предыдущем примере?"

Для кошки с JSON `` `"name": "Billy"` ``.

Тест "Где обрабатывать возникающую ошибку?"

При условии, что она случилась в одной из горутин-воркеров, конкурентно выполняющих общую задачу.

Выберите один вариант из списка

- ☐ В воркере, в месте возникновения
- ☐ В родительской горутине
- ☐ В месте, где хватает для этого контекста и полномочий
- ☐ В main-горутине

Передача ошибок с помощью типа-контейнера

Errors are values.

Перед нами стоит задача по передаче ошибки в "родительскую" горутину.

Одним из первых решений, приходящих в голову, является создание определённого типа-контейнера, хранящего как результат операции, так и ошибку от неё (исходник примера):

```
// https://goplay.tools/snippet/vuFrDhhHbe6

type Cat struct {
    Name string `json:"name"`
}

type CatContainer struct {
    Cat Cat
    Err error // Сюда складываем ошибку, если что-то пошло не так.
}

func main() {
    catsJSONs := []string{`{"name": "Bobby"}`, `"name": "Billy"`,
`{"name": "Васёк"}`}
    catsCh := make(chan CatContainer)

    var wg sync.WaitGroup
    wg.Add(len(catsJSONs))

    for _, catData := range catsJSONs {
        go func(catData string) {
            defer wg.Done()

            var cat Cat
```

```
            if err := json.Unmarshal([]byte(catData), &cat); err !=
nil {
                catsCh <- CatContainer{Err: err} // Случилась ошибка.
                return
            }
            catsCh <- CatContainer{Cat: cat} // Всё прошло хорошо.
        }(catData)
    }

    go func() {
        wg.Wait()
        close(catsCh)
    }()

    for catContainer := range catsCh {
        if catContainer.Err != nil {
            fmt.Println("ERROR:", catContainer.Err)
            continue
        }
        fmt.Println(catContainer.Cat)
    }
}

/*
ERROR: invalid character ':' after top-level value
{Васёк}
{Bobby}

*/
```

Какие тут встречаются лучшие практики?

---

1) Следует гарантировать наличие результата в контейнере при отсутствии ошибки и наоборот (уже знакомое нам правило).

Исходя из этого, следующий код не является лучшим вариантом:

```
go func(catData string) {
    defer wg.Done()

    var cat Cat
    catsCh <- CatContainer{
        Cat: cat,
```

```
        Err: json.Unmarshal([]byte(catData), &cat),
    }

}(catData)
```

Да, вышло лаконично, но теоретически мы не знаем, что окажется в `cat` при ошибке десериализации. С одной стороны наш потребитель и не должен смотреть на потенциальный мусор в контейнере, если видит в нём `Err`, а с другой – зачем вводить людей в заблуждение и создавать возможность потенциальной баги?

Пример из той же оперы:

```
type Result struct {
    Error    error
    Response *http.Response
}

// ...

var result Result

result.Response, result.Error = http.Get(url)
```

2) Инкапсуляция контейнера.

Если мы хотим переиспользовать контейнер на нескольких уровнях нашего приложения, то можно защититься от его возможного изменения ([исходник примера]):

```
// https://goplay.tools/snippet/zNIm8FFqEaa

type Cat struct {
    Name string `json:"name"`
}

type CatEnvelope struct {
    cat Cat
    err error // Сюда складываем ошибку, если что-то пошло не так.
}

func NewCatEnvelope(c Cat) CatEnvelope {
```

```
    return CatEnvelope{cat: c}
}

func NewCatEnvelopeWithErr(err error) CatEnvelope {
    return CatEnvelope{err: err}
}

func (c CatEnvelope) Unpack() (Cat, error) {
    return c.cat, c.err
}

func main() {
    catsJSONs := []string{`{"name": "Bobby"}`, `"name": "Billy"`,
`{"name": "Васёк"}`}
    catsCh := make(chan CatEnvelope)

    var wg sync.WaitGroup
    wg.Add(len(catsJSONs))

    for _, catData := range catsJSONs {
        go func(catData string) {
            defer wg.Done()

            var cat Cat
            if err := json.Unmarshal([]byte(catData), &cat); err !=
nil {
                catsCh <- NewCatEnvelopeWithErr(err) // Случилась
ошибка.
                return
            }
            catsCh <- NewCatEnvelope(cat) // Всё прошло хорошо.
        }(catData)
    }

    go func() {
        wg.Wait()
        close(catsCh)
    }()

    for catEnvelope := range catsCh {
        // Мы можем как угодно менять копии, полученные после
распаковки,
        // не влияя на оригинальный контейнер.
        c, err := catEnvelope.Unpack()
        c = Cat{Name: "Hacked"}
```

```
        err = errors.New("hacked")

        c, err = catEnvelope.Unpack()
        if err != nil {
            fmt.Println("ERROR:", err)
            continue
        }
        fmt.Println(c)
    }
}

/*
ERROR: invalid character ':' after top-level value
{Васёк}
{Bobby}
*/
```

Обратим внимание на конструкторы, которые в целом не позволяют нам работать одновременно и со значением и с ошибкой, дабы удовлетворить первому пункту рекомендаций, описанному выше:

```
func NewCatEnvelope(c Cat) CatEnvelope {
    return CatEnvelope{cat: c}
}

func NewCatEnvelopeWithErr(err error) CatEnvelope {
    return CatEnvelope{err: err}

}
```

Также очевидно, что метод, подобный нашему `Unpack`

```
func (c CatEnvelope) Unpack() (Cat, error) {
    return c.cat, c.err

}
```

сработает только для не-указателей и интерфейсов, т.е. типов, которые можно копировать, не боясь получить ссылку на оригинальную область памяти. В обратном же случае придётся делать что-то вроде

```
func (c CatEnvelope) Unpack() (Cat, error) {
    return c.cat.Copy(), c.err
```

```
}
```

Резюме

Как мы помним, **ошибки – это значения**, и мы можем делать с ними, что угодно, вплоть до отправки их своей маме.

Здесь как раз этот случай! Описываем контейнер, дающий полное представление о результатах работы горутины-воркера, тем самым организовывая способ сообщить ей о результатах своей работы "наверх". Profit.

Тест "И так сойдёт (нет)"

Исходник примера.

Ниже пример банальной gotcha, которая ~~к нашему стыду~~ иногда пролезала на прод.

Глядя на пример из предыдущего урока, разработчик задаётся вопросом – а зачем воротить какие-то контейнеры, если можно объявить специальную переменную, отвечающую за ошибку во всех воркерах?

```
func main() {
    catsJSONs := []string{`{"name": "Bobby"}`, `"name": "Billy"`,
`{"name": "Васёк"}`}
    catsCh := make(chan Cat, len(catsJSONs))

    var wg sync.WaitGroup
    wg.Add(len(catsJSONs))

    var err error // Заводим специальную переменную для хранения
ошибки.

    for _, catData := range catsJSONs {
        go func(catData string) {
            defer wg.Done()

            var cat Cat
            err = json.Unmarshal([]byte(catData), &cat)
            catsCh <- cat
        }(catData)
```

```
    }

    wg.Wait()
    fmt.Println(err) // Выводим ошибку на экран.

}
```

Что ожидается на экране после выполнения данного кода?

Выберите один вариант из списка

- ☐ nil
- ☐ invalid character ':' after top-level value
- ☐ Вывод недетерминирован
- ☐ <nil>

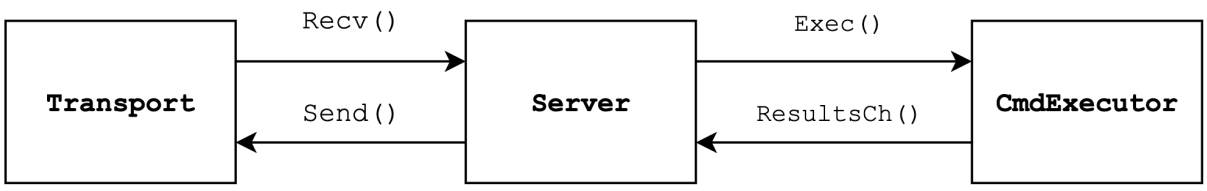
Задача "Command Executor"

Ссылка на заготовку.

Вам необходимо реализовать сервер, который:

- принимает команды;
- выполняет их с помощью стороннего executor'а;
- конкуретно отправляет результаты команд обратно.

Упрощённо процесс, описанный выше, выглядит следующим образом:



Транспорт

Позволяет принимать команды и отправлять статус их выполнения обратно:

```
type ProtoCommand struct {
    ID string
}

type ProtoCommandResult struct {
    ID     string
    Status ProtoCommandStatus
}

type ITransport interface {
    // Context возвращает контекст текущего соединения.
    Context() context.Context

    // Recv позволяет принять команду. При успешном отсоединении
клиента вернёт ошибку io.EOF.
    // Любая другая ошибка будет постоянной и при её появлении нужно
завершать обработчик
    // транспорта в принципе. Вызов Recv блокирующий.
    //
    // Безопасно иметь одну горутину, вызывающую Recv и другую -
вызывающую Send.
    // Но опасно вызывать Recv в разных горутинах.
    Recv() (*ProtoCommand, error)

    // Send позволяет отправить результат команды. Любая ошибка будет
постоянной и
    // аналогичной той, что придёт в очередном Recv. Вызов Send
блокирующий.
    //
    // Безопасно иметь одну горутину, вызывающую Recv и другую -
вызывающую Send.
    // Но опасно вызывать Send в разных горутинах.
    Send(c *ProtoCommandResult) error

}
```

Сервер

Вся основная работа сервера заключается в его методе `(*Server).ProcessCommandsStream`:

```
type CommandResult struct {
    ID   CommandID
```

```
    Data any
    Err  error
}

type ICmdExecutor interface {
    // ResultsCh возвращает канал с потоком результатов выполненных
команд.
    ResultsCh() <-chan CommandResult
    // Exec выполняет команду, результат приходит в ResultsCh.
    Exec(cid CommandID) error
}

type IMetricsCollector interface {
    CountCommand(cid CommandID) error
    CountError(err error) error
}

type Server struct {
    executor ICmdExecutor
    metrics  IMetricsCollector
    wg       sync.WaitGroup
}

// ProcessCommandsStream получает команды из транспорта, отдаёт их на
выполнение,
// а параллельно с этим: слушает канал результатов команд, считает по
ним метрики,
// на основе CommandResult формирует ProtoCommandResult и отправляет
назад в транспорт.

func (s *Server) ProcessCommandsStream(t ITransport) error
```

- Если результат команды содержит ошибку, то сервер вызывает для неё метод коллектора метрик `CountError`, иначе, если результат команды содержит данные, то сервер вызывает метод `CountCommand`.

- Таблица соответствий ошибок в `CommandResult` и `ProtoCommandStatus`:

```
ErrCommandTimeout     - ProtoCommandStatusTimeoutError
ErrUnsupportedCommand - ProtoCommandStatusUnsupportedCommandError
<nil>                 - ProtoCommandStatusSuccess
```

```
Любая другая ошибка   - ProtoCommandStatusUnknownError
```

Executor команд

Как и коллектор метрик для вас всего лишь интерфейс – реализовывать их не нужно.

Тесты

Для большего понимания обратитесь к тестам (хотя они получились достаточно сложными и интересными).

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/command-executor
$ go test -v -race .
=== RUN   TestServer
=== RUN   TestServer/TestErrFromCountCommand
=== RUN   TestServer/TestErrFromCountError
=== RUN   TestServer/TestExecError
=== RUN   TestServer/TestFlow
=== RUN   TestServer/TestFlow/commands_was_executed_concurrently
=== RUN   TestServer/TestFlow/status_processing_is_ok
=== RUN   TestServer/TestRecvError
--- PASS: TestServer (1.02s)
    --- PASS: TestServer/TestErrFromCountCommand (0.00s)
    --- PASS: TestServer/TestErrFromCountError (0.00s)
    --- PASS: TestServer/TestExecError (0.00s)
    --- PASS: TestServer/TestFlow (1.01s)
        --- PASS:
TestServer/TestFlow/commands_was_executed_concurrently (0.00s)
        --- PASS: TestServer/TestFlow/status_processing_is_ok (0.00s)
    --- PASS: TestServer/TestRecvError (0.00s)
PASS

ok      tasks/07-working-with-errors-in-concurrency/command-executor
1.333s
```

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

Передача ошибок с помощью отдельного канала

Отсутствие "из коробки" возможности передать через канал и валидное значение и ошибку – фундаментальный вопрос, который многие разработчиками считают проблемой и недостатком архитектуры каналов в Go.

В предыдущих уроках мы обошли это с помощью введения дополнительного типа-контейнера:

```
type CatContainer struct {
    Cat Cat
    Err error

}
```

Вторым очевидным решением является введение канала для ошибок (исходник примера):

```
// https://goplay.tools/snippet/x867b_LwmsD

type Cat struct {
    Name string `json:"name"`
}

func main() {
    catsJSONs := []string{`{"name": "Bobby"}`, `"name": "Billy"`,
`{"name": "Васёк"}`}

    done := make(chan struct{})
    catsCh := make(chan Cat)
    errsCh := make(chan error)

    for _, catData := range catsJSONs {
        go func(catData string) { // Разбираем котиков в несколько
горутин.
            var cat Cat
            if err := json.Unmarshal([]byte(catData), &cat); err !=
nil {
                errsCh <- err // Случилась ошибка.
                return
            }
            catsCh <- cat // Всё прошло хорошо.
        }(catData)
    }
```

```
    var wg sync.WaitGroup
    wg.Add(len(catsJSONs))

    go func() {
        wg.Wait()
        close(done)
    }()

    for {
        select {
        case <-done:
            return

        case c := <-catsCh:
            wg.Done()
            fmt.Println(c)

        case err := <-errsCh:
            wg.Done()
            fmt.Println(err)
        }
    }
}
```

Обратите внимание на следующие непривычные особенности примера выше:

- Введён третий канал `done`, служащий сигналом к завершению программы.

- `wg.Done` перенесён из писателей в читателя. Таким образом, финальный цикл завершает сам себя через сторонний сигнал и только после того, как результаты из всех горутин будут обработаны.

Тест "Что будет при чтении из nil-канала?"

Выберите один вариант из списка

Паника
Непрерываемая блокировка
Получим значение по умолчанию для типа элемента канала (channel element type)
Undefined behaviour

Задача "Hashing Pipeline"

Ссылка на заготовку.

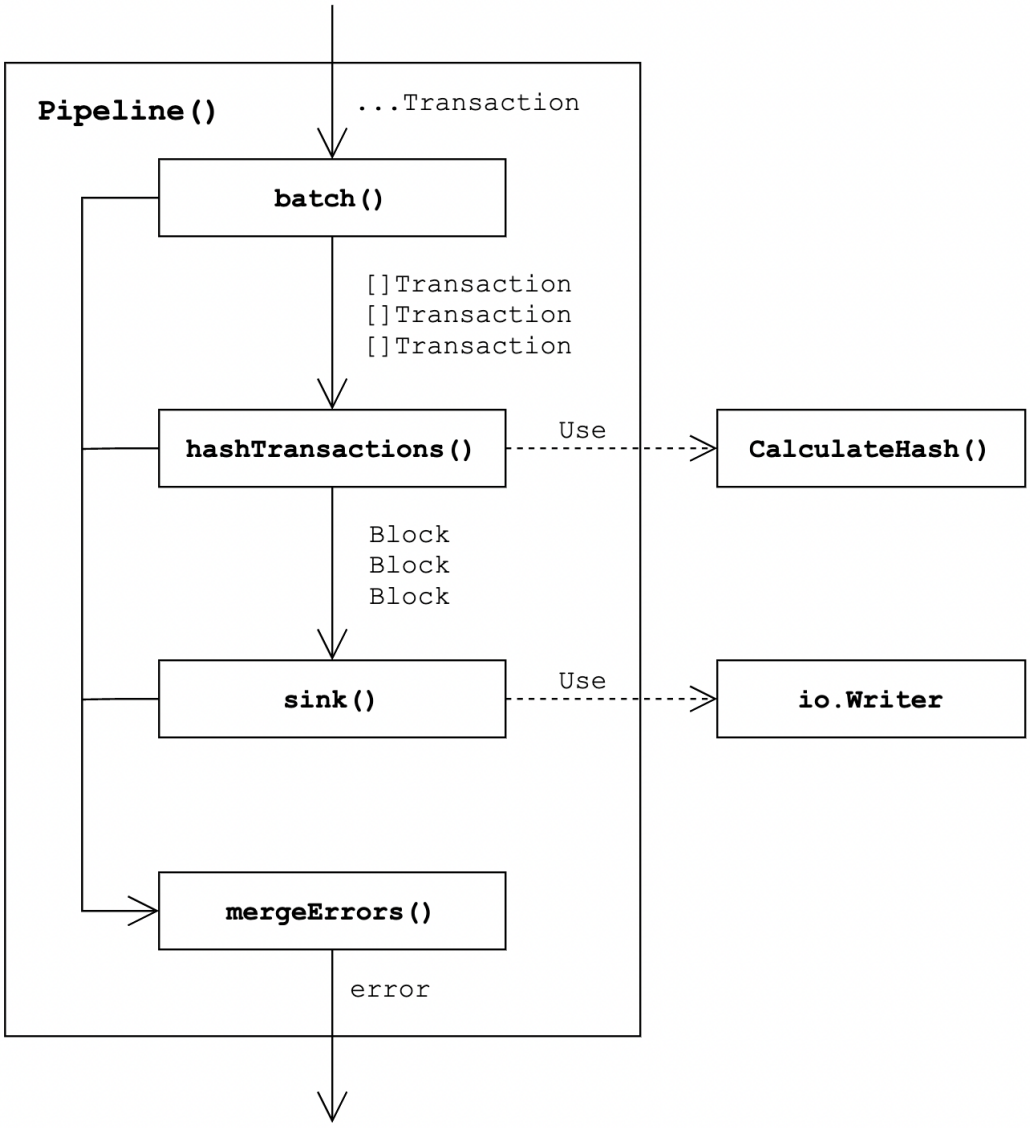
Трубопроводные войска всегда готовы для броска.

Pipeline

Вам необходимо реализовать функцию

```
func Pipeline(ctx context.Context, batchSize int, out io.Writer, txs ...Transaction) error
```

Внутреннее устройство которой схематично можно представить следующим образом:



Таким образом, в наш пайплайн входят следующие этапы:

- группирование входных транзакций в пачки размером `batchSize`;
- хеширование пачек транзакций функцией `CalculateHash` и получение так называемых **блоков** (каждый из которых хранит свой хеш);
- вывод хешей блоков в `io.Writer`;
- слияние потоков ошибок из каждого этапа в один поток и организация прерывания всего пайплайна при ошибке в одном из этапов.

Этапы выполняются последовательно друг за другом, но внутри каждый из них работает конкуретно, т.е. самый первый этап не дожидается завершения самого последнего, а сразу выдаёт полезные данные последующему этапу.

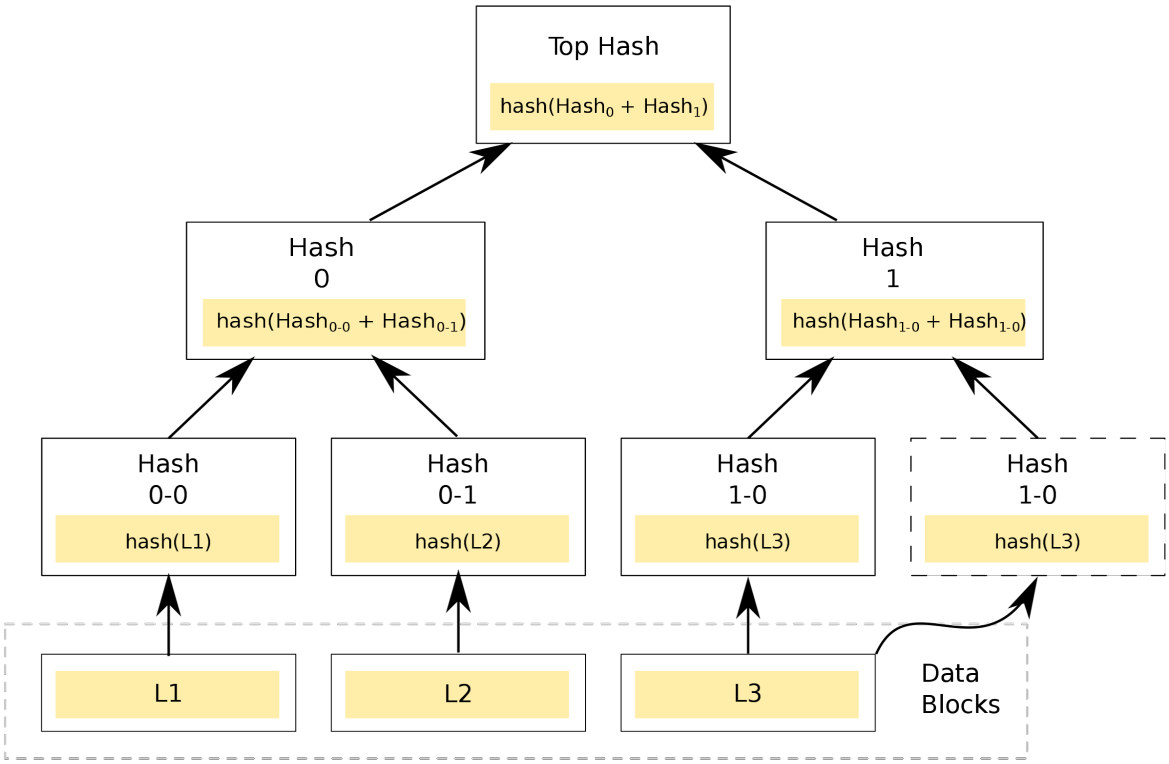
Подробнее о работе каждой из функций читайте в документации к ней.

CalculateHash

```
func CalculateHash(hh []Hashable) (Hash, error)
```

Функция должна реализовывать хэширование входящих элементов по принципу [дерева хешей (дерева Меркла)](#).

Принцип хеширования представлен ниже. Обратите внимание, что если у элемента нет пары, то в качестве пары нужно использовать самого себя (таким образом, хеши для массива из одного элемента и для массива из двух таких же элементов будут равны):



(изображение взято с Вики, создано пользователем MrTsepa)

В нашем случае блоками данных являются транзакции в составе очередной пачки транзакций.

Напутствия

Мы рекомендуем выполнять задачу последовательно: реализовать шаг за шагом отдельные функции пайплайна (запуская тесты только для них), а затем уже взяться за корневую функцию `Pipeline`. Будьте готовы, что тесты могут блокироваться при неверной реализации (или её отсутствии в принципе).

Обратите внимание, что недопустимо менять сигнатуры функций, а также предоставленные структуры и интерфейсы, равно как и наши тесты.

По завершению работы не забудьте прогнать своё решение через **race detector:**

```
$ cd tasks/07-working-with-errors-in-concurrency/hashing-pipeline
$ go test -v -race .
...
PASS
ok      tasks/07-working-with-errors-in-concurrency/hashing-pipeline
16.870s
```

Если вы до этого никогда не сталкивались с паттерном пайплайна или задача вводит вас в ступор, то рекомендуем пройти модуль до конца, ознакомиться со списком **литературы**, а затем вернуться сюда.

Также в комментариях ниже оставлен универсальный рецепт для реализации очередного этапа пайплайна.

Удачи!

P.S. Приводим время исполнения тестов для авторского решения (без рейс детектора):

```
$ cd tasks/07-working-with-errors-in-concurrency/hashing-pipeline
```

```
$ go test -run=TestCalculateHash/different_1000000_elements -count 1
.
ok      tasks/07-working-with-errors-in-concurrency/hashing-pipeline
0.558s

$ go test -run=TestPipeline/big_data_processing -count 1 .
ok      tasks/07-working-with-errors-in-concurrency/hashing-pipeline
2.216s

$ go test -count 1 .

ok      tasks/07-working-with-errors-in-concurrency/hashing-pipeline
5.848s
```

Чтобы ваше решение не было отброшено Stepik по таймауту, оно должно занимать плюс-минус такое же время. Обращайте внимание на эффективность своих операций, особенно на процесс создания новых слайсов.