Обработка ошибок — общие соображения и грязные подробности

Дмитрий Грошев

Application Developer Days
12.05.2012

Вступление

Все делают ошибки, однако некоторые думают об ошибках так:

Но лучше делать это так:

Ошибки

Для этого нужно знать о враге больше!

Короткий план доклада

- общие соображения
  - классификация ошибок
  - о балансе
  - простые и сложные ошибки
  - третий путь
  - всё ещё хуже — ошибки перегрузки, ошибки параллелизма
- грязные подробности
  - исторически сложившиеся методы обработки ошибок
  - велосипеды
  - пример кода

Часть 1: общие соображения

Классификация

Наиболее очевидная классификация по времени появления:

- времени компиляции
- времени выполнения

Ремарка о балансе

Когда ловить ошибки?

- во время компиляции — код либо сложнее, либо многословнее
- во время выполнения — падает надёжность, ведь запуск кода не гарантирует его работоспособности

Необходим баланс между этими крайностями

«Простые» ошибки

Ошибки бывают очевидными:

```
1 a = 1
2 b = "b"
3 a + b
```

- JS: «это не ошибка»
- Python: «добавь try/except»
- Java: «где типы?»

«Это не ошибка»

Этот слайд оставлен пустым в память всех жертв плохого дизайна

«Добавь try/except»

```
1 a = 1
2 b = "b"
3 try:
4     a + b
5 except TypeError:
6     print "something bad happened"
```

- повседневная реальность большинства разработчиков
- требует юнит-тестов
- 100% покрытие ничего не гарантирует

«Где типы?»

```
1  public class AddNumbers {
2      public static int add() {
3          int a = 1;
4          String b = "b";
5          return a + b;
6      }
7  }
```

- этот код даже не скомпилируется
- этого кода слишком много
- люди не любят писать много и отказываются от типов вообще

«Где типы?»

```
1  my_sum = a + b
2     where a = 1
3           b = "b"
```

- этот код тоже не скомпилируется
- типы a и b однозначно вытекают из соответствующих литералов — зачем их указывать?
- компилятор может пытаться выводить типы сам
- не все корректные программы могут пройти проверку типов

Проблема статической типизации

Ещё раз: *не все корректные программы статически типизируемы*

Или: *любая система типов может мешать программисту*

Проблема статической типизации

Пример:

```
1  public class test {
2      public static int add() throws IOException {
3          int a = 1;
4          int b = 1;
5          return a + b;
6      }
7  }
```

Оптимистичная типизация

```erlang
type_error() ->
    A = 1,
    B = "b",
    A + B.
```

- этот код скомпилируется
- тайпчекер (отдельная программа в случае Erlang'а) укажет на ошибку

Оптимистичная типизация

```erlang
no_type_error() ->
    A = 1,
    B = "b",
    try throw(B)
    catch _:T -> A + T
    end.
```

- тайпчекер не найдёт ошибки в этом коде
- «оптимистичная» = если тайпчекер не может вывести тип, считается, что всё хорошо

Оптимистичная типизация

Система типов:

- оптимистичная — пропускает часть ошибок (но все найденные ошибки существуют в реальности)
- пессимистичная — отклоняет часть корректных программ (но скомпилированная программа точно не содержит ошибок типов)

«Простые» ошибки: резюме

Проблема ошибок типов («простых» ошибок) более-менее решена

- статическая типизация с выводом типов
- оптимистичная типизация для динамических языков

«Сложные» ошибки

```java
public class FindMean {
    public static float mean(String[] args) {
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        return (a + b) / 2;
    }
}
```

- этот код скомпилируется
- где ошибка?

«Сложные» ошибки

- в выражении [(a + b) / 2] a + b может быть больше, чем int
- это сложно увидеть глазами
- это не проверит компилятор
- 100% coverage не поможет
- эти ошибки связаны со значениями (а не с типами)

Property-based тестирование

Тесты для JSON-библиотеки:

```
prop_encode_decode() ->
    ?FORALL(Data, json(),
            Data == decode(encode(Data))).
```

- тестирующая система сама может генерировать тесты
- предполагается, что в определении json нет ошибки
- вероятность найти «сложную» ошибку выше

Зависимые типы

```
fun add {m,n:int}
    (a: int m, b: int n): int (m+n) =
    a + b
```

- конкретные значения и их соотношения являются параметрами типов (*1: int(1)*)
- сложные компиляторы, очень сложно писать

«Сложные» ошибки: резюме

- проблема не имеет общепринятого решения
- ошибки значений во время выполнения практически неизбежны
- что делать?

Третий путь

Оставить надежду найти и обработать все ошибки

Good enough

Нужно:

- признать неизбежность ошибок
- проектировать весь стек технологий с учётом неизбежности *неожиданных* ошибок (функция в используемой библиотеке изменила интерфейс, сложение вернуло exception)
- разделять обработку ошибок для отображения и для сохранения работоспособности системы в целом (CGI и HTTP 500 вместо падения сервера)

Хьюстон, у нас проблема

Ошибка произошла. Что делать?

- отбросить «испорченное» состояние вместо сохранения
- перезапуститься, начав с чистого листа
- let it crash (it will crash anyway)

О падениях

Необходимо минимизировать цену падения и облегчить падение (вместо продолжения работы в неправильном режиме)

- изоляция потоков исполнения
- изоляция данных и состояния, явное выделение состояния
- асинхронный message-passing с копированием (Akka не Erlang)
- никаких глобальных event loop'ов

Никому нельзя доверять!

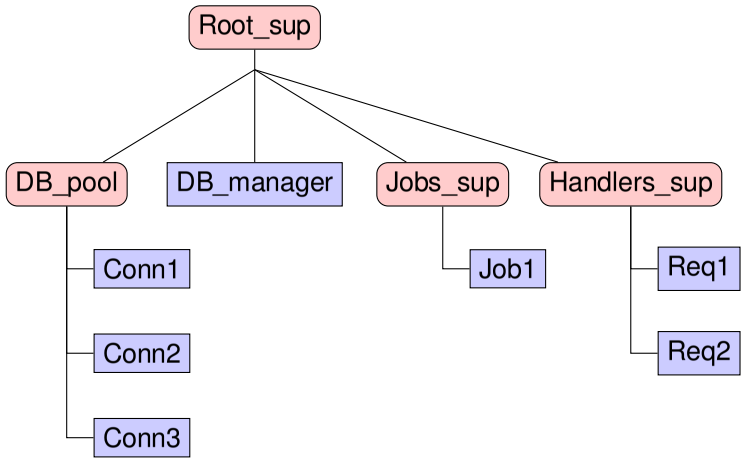
Логика перезапуска

Необходимо контролировать логику перезапуска

- ошибка может произойти для группы процессов
- если ошибка происходит слишком часто, нет смысла перезапускать процесс ()
- то, что перезапускает (supervisor) само может содержать ошибку

Достаточно общим решением является supervision tree

Supervision tree



Ремарка

Кстати, мы только что изобрели Erlang

Ремарка

Каждый раз, когда кто-то говорит о поддержке многоядерных сред Erlang'ом как о главном его плюсе, Бог убивает котёнка

Не только неожиданные ошибки

Подход let it crash можно расширить на известные программисту ошибки:

```
1 assert_tuple(X) ->
2     {_, _} = X.
```

Иногда можно описывать только happy path:

```
1 read_input(Str) ->
2     {ok, X} = parse_input(Str),
3     ok = do_something(X).
```


Chaos monkey

- в 2010 Netflix переехал на AWS
- стоимость перезапуска инстанса упала
- Netflix создал Chaos monkey — процесс, убивающий случайный инстанс
- цена перезапуска должна быть низкой

Load hell

Высокая нагрузка это не
«мой магазин виагры держит 10к хитов в сутки»

Load hell

Ошибки, связанные с высокой нагрузкой:

- переполнение mailbox'ов в случае message passing
- переполнение числа открытых файловых дескрипторов
- невозможность сделать malloc
- медленные дисковые операции (нет записи в лог)
- ...

Техники борьбы:

- back pressure
- back pressure
- back pressure

Компилятор не помогает, нагрузочное тестирование может не содержать все «опасные» паттерны активности

Hell of parallel execution

- паралеллизм при исполнении программ делает всё ещё хуже
- правильные программы при параллельном исполнении становятся неправильными
- подробнее в докладе Евгения Кирпичёва на ADD-2010

Что дальше

Мы поговорили о:

- ошибках времени компиляции и времени выполнения
- ошибках типов и термов
- property-based тестировании
- зависимых типах
- принципе let it crash
- цене перезапуска
- супервизорах
- нерешённых проблемах

Ближе к коду!

Часть 2: грязные подробности

ПРЕДУПРЕЖДЕНИЕ
далее следует значительное количество
неидиоматичного Python-кода

Обработка кодов ошибок

```
const int CODE_ONE = 1;
const int CODE_TWO = 2;

int dummy() {
  if (make_test()) {
      return CODE_ONE;
    } else {
      return CODE_TWO;
  }
}
```

- мы все это видели
- компилятор не контролирует обработку возвращаемых значений
- код превращается в лапшу из if'ов/case'ов

Null

```
Exception in thread "main"
java.lang.NullPointerException
```

- null *гораздо* хуже кодов возврата — это нетипизированный терм в типизированной среде (в C все привыкли к содомии кодов возврата)
- компилятор контролирует обработку возвращаемых значений, но не null
- Тони Хоар (создатель Algol'a) считает введение null своей худшей ошибкой

Ремарка: тотальность

Тотальность — свойство функции всегда делать что-то осмысленное, несмотря на вход.

Плохо:

```
def connect_bad(db):
    return get_connection(db) if good(db) else None
```

Хорошо:

```
def connect_better(db):
    if not good(db): log_and_raise(DbException(db))
    return get_connection(db)
```

Пример:

```
def test_bad(maybe_db_data):
    connect = connect_bad(maybe_db_data)
    if not connect:
        raise Exception("ALARM")

def test_good(maybe_db_data):
    connect = connect_good(maybe_db_data)
```

Тотальность помогает изолировать ошибки и отлаживать код

О потоках

- control flow: обычно виден, локален и очевиден
- error flow: может быть абсолютно неочевидным

req_handlers.py:

```
def handle_req(req):
    try:
        data_handlers.handle(req.data)
    except SomeException:
        do_something()
```

data_handlers.py:

```
def handle(data):
    if not test(data):
        raise SomeException()
    else:
        store(data)
```


Error flow

- Exception'ы делают error flow нелокальным и независимым от control flow
- checked exceptions в Java помогает частично решить эту проблему
- альтернатива — метки успешности/неуспешности выполнения

Метки успешности

req_handlers.py:

```
def handle_req(req):
    (is_ok, res) = data_handlers.handle(req.data)
    if not is_ok: do_something()
```

data_handlers.py:

```
def handle(data):
    if not test(data): return (False, "failed")
    return (True, store(data))
```


Метки успешности

Непривычно, но:

- error flow полностью соответствует control flow
- тайпчекер может проверять обработку ошибок без поддержки checked exceptions

Проблема

```python
def handle(data):
    (is_ok, value) = parse(data)
    if not is_ok:
        return (False, value)
    (is_ok, value) = process(value)
    if not is_ok:
        return (False, value)
    return finalize(value)
```


Двойственная природа Exception'a

Exception выполняет 2 функции:

- оповещение вызывающего об ошибке
- прерывание исполнения

Можно ли решить вторую проблему с метками успешности?

Ремарка о pattern matching

```erlang
A = {1, 2, 3},
{B1, B2, B3} = A,
{C1, C2} = A,   %% ERROR
{D1, D2, 0} = A %% ERROR
```


Pattern matching

```erlang
handle(Request) ->
    {ok, Prepared} = prepare(Request),
    {ok, Result} = process(Prepared),
    show(Result).
```

```
** exception error: no match of right hand side
value {error,some_error_exception}
```

Ошибка информативнее, но это exception со всеми его минусами

Смысл точки с запятой

```
usr_id = auth();
status = send(usr_id, "logged");
```

- ";" можно воспринимать как «безусловно перейти к следующему выражению»
- можно заменить данный переход на условный

Функция bind

```
bind(auth(),
     lambda user_id: bind(send(usr_id, "logged"),
                          lambda status: status))
```

- функция bind принимает решение, вызвать ли свой второй аргумент
- в любой момент вся цепочка выражений может вернуть значение без вычисления остальных выражений
- если *auth* и *send* возвращают метки успешности, конструкция аналогична использованию Exception
- тайпчекер, если он есть, может контролировать возврат *auth* и *send*

Функция bind

```
def auth():
    if authed():
        return (True, "usr42")
    else:
        return (False, "not_authed")

def send(usr, msg):
    return (True, do_send(usr, msg))

def bind((is_ok, value), f):
    if is_ok:
        return f(value)
    else:
        return (False, value)
```


Функция return

```
def ret(value):
    return (True, value)
```

- многие функции ничего не знают про наши метки успешности выполнения
- *return* позволяет использовать их

Функция return

```
def ignorant_auth():
    return "usr42"

bind(ret(ignorant_auth()),
     lambda user_id: bind(send(usr_id, "logged"),
                          lambda status: status))
```


Монада Either

- сочетание соглашения о метках успешности выполнения, *bind* и *return* образует монаду (в данном случае Either)
- в этой модели можно оперировать с любыми функциями
- *bind* обеспечивает прерывание потока выполнения
- подобную конструкцию можно создать в любом языке с первоклассными функциями
- error flow полностью совпадает с control flow
- тайпчекер укажет на ошибки
- счастье

Проблемы монадической обработки ошибок

- без оптимизирующего компилятора активное создание анонимных функций может быть проблемой
- без тайпчекера легко забыть вернуть значение с меткой успешности
- необходимы синтаксические извращения, чтобы вызовы *bind* выглядели менее страшно
- вызывающий код должен уметь обрабатывать ошибки вызываемого кода
- иногда при ошибке нужно передавать управление выше по стеку, а не непосредственно вызывающему, в этом случае код становится громоздким

Снова о балансе

- Exception'ы делают код более запутанным и менее предсказуемым, но удобны для передачи управления далеко по стеку
- отсутствие checked exceptions делают использование библиотек с exception'ами опасным либо трудноотлаживаемым (catch-all)
- метки успешности выполнения требуют либо развитого pattern matching'а, либо монад, но делают код понятнее
- pattern matching есть не везде и затрудняет перехват ошибок (если он нужен)
- монады сложно сделать быстрыми и удобными без поддержки языка

Нечто среднее

```python
def my_call(f, err, *args):
    try:
        return f(*args)
    except Exception as e:
        raise MyException(False, (err, e))

def my_return(x):
    raise MyException(True, x)

def test():
    try:
        conn = my_call(connect_db, "can't connect")
        data = my_call(make_req, "req error", conn)
        my_return(data)
    except MyException as e:
        return e.result if e.is_ok else e.error
```


Нечто среднее

- нет оверхеда на создание анонимных функций
- функция *test* тотальна
- catch-all малы и не затрудняют дебаг
- таким образом удобно писать handler'ы запросов пользователей

Почему это важно

- мы пишем на Erlang'e
- Erlang позволяет мало думать о влиянии ошибок на стабильность системы
- вместо размышлений о стабильности приходится много думать об отображении ошибок

Почему это важно

```erlang
try
    {Method, TaskName, VarSpecs} =
      ?Z_CATCH({_, _, _} = lists:keyfind(Method, 1, TaskSpecs),
               bad_method),
    TaskVarsRoute =
      ?Z_CATCH([fetch_var(RouteVar, RouteVarType, Bindings)
                || {RouteVar, RouteVarType} <- RouteVars],
               bad_route),
    TaskVars = [?Z_CATCH(fetch_var(Var, VarType, QSVals),
                         {bad_var, Var})
                || {Var, VarType} <- VarSpecs],
    z_return(rnbwdash_task:create(...))
catch
    ?Z_OK(Task) -> form_reply(run_task(Task), Errors, Req@);
    ?Z_ERROR(Err) -> form_error(Err, Req@)
end
```

Почему это важно

```
{Method, TaskName, VarSpecs} =
  ?Z_CATCH({_, _, _} = lists:keyfind(Method, 1, TaskSpecs)
           bad_method)
```


Почему это важно

```
TaskVarsRoute =
  ?Z_CATCH([fetch_var(RouteVar, RouteVarType, Bindings)
            || {RouteVar, RouteVarType} <- RouteVars],
           bad_route)
```


Почему это важно

```
TaskVars = [?Z_CATCH(fetch_var(Var, VarType, QSVals),
                     {bad_var, Var})
            || {Var, VarType} <- VarSpecs]
```


Ну и наконец…

Вопросы?

Мы ищем сотрудников! office@selectel.ru

Были использованы следующие картинки под CC:

- http://commons.wikimedia.org/wiki/File:Struthio-camelus-australis-grazing.jpg
- http://commons.wikimedia.org/wiki/File:%22Attack-Attack-Attack%22_-_NARA_-_513888.tif