

Обработка ошибок — общие соображения и грязные подробности

Дмитрий Грошев

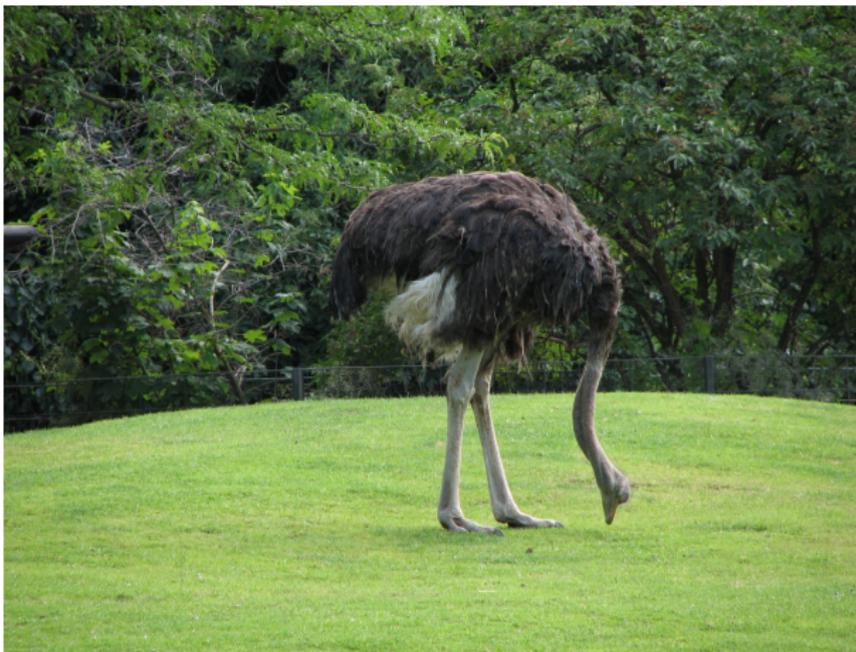


Application Developer Days

12.05.2012

Вступление

Все делают ошибки, однако некоторые думают об ошибках так:



Но лучше делать это так:



Для этого нужно знать о враге больше!

- ▶ 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

Наиболее очевидная классификация по времени появления:

- ▶ времени компиляции
- ▶ времени выполнения

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

Ошибки бывают очевидными:

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

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

Этот слайд оставлен пустым в память всех жертв плохого дизайна

```
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

```
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

```
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

Ещё раз: не все корректные программы статически типизируемы

Или: любая система типов может мешать программисту

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 }
```

```
1 type_error() ->  
2   A = 1,  
3   B = "b",  
4   A + B.
```

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

```
1 no_type_error() ->
2   A = 1,
3   B = "b",
4   try throw(B)
5   catch _:T -> A + T
6   end.
```

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

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)

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

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

```
1 public class FindMean {  
2     public static float mean(String[] args) {  
3         int a = Integer.parseInt(args[0]);  
4         int b = Integer.parseInt(args[1]);  
5         return (a + b) / 2;  
6     }  
7 }
```

- ▶ ЭТОТ КОД СКОМПИЛИРУЕТСЯ
- ▶ где ошибка?

- ▶ 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)

Тесты для JSON-библиотеки:

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

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

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

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

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

Оставить надежду найти и обработать все ошибки

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)

Ошибка произошла. Что делать?

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

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

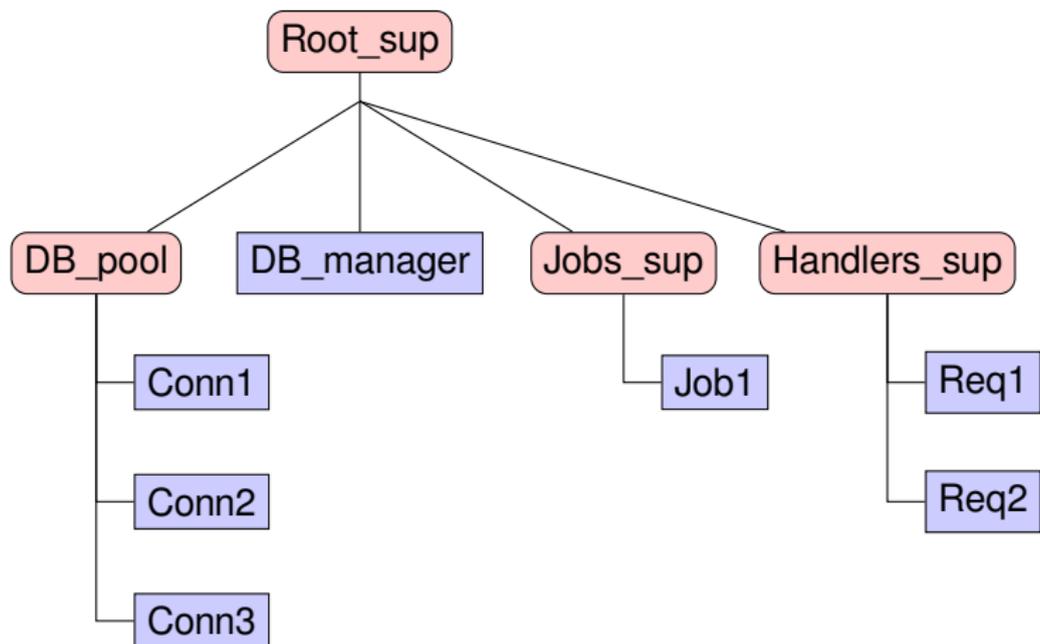
- ▶ изоляция потоков исполнения
- ▶ изоляция данных и состояния, явное выделение состояния
- ▶ асинхронный message-passing с копированием (Akka не Erlang)
- ▶ никаких глобальных event loop'ов

Никому нельзя доверять!

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



Кстати, мы только что изобрели Erlang

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

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).
```

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

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

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

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

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

```
1  const int CODE_ONE = 1;
2  const int CODE_TWO = 2;
3
4  int dummy() {
5      if (make_test()) {
6          return CODE_ONE;
7      } else {
8          return CODE_TWO;
9      }
10 }
```

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

```
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

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

Плохо:

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

Хорошо:

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

Пример:

```
1 def test_bad(maybe_db_data):
2     connect = connect_bad(maybe_db_data)
3     if not connect:
4         raise Exception("ALARM")
5
6 def test_good(maybe_db_data):
7     connect = connect_good(maybe_db_data)
```

Тотальность помогает изолировать ошибки и отлаживать код

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

req_handlers.py:

```
1 def handle_req(req):
2     try:
3         data_handlers.handle(req.data)
4     except SomeException:
5         do_something()
```

data_handlers.py:

```
1 def handle(data):
2     if not test(data):
3         raise SomeException()
4     else:
5         store(data)
```

- ▶ 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

req_handlers.py:

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

data_handlers.py:

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

Непривычно, но:

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

```
1 def handle(data):
2     (is_ok, value) = parse(data)
3     if not is_ok:
4         return (False, value)
5     (is_ok, value) = process(value)
6     if not is_ok:
7         return (False, value)
8     return finalize(value)
```

Exception выполняет 2 функции:

- ▶ оповещение вызывающего об ошибке
- ▶ прерывание исполнения

Можно ли решить вторую проблему с метками успешности?

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

```
1 handle(Request) ->  
2   {ok, Prepared} = prepare(Request),  
3   {ok, Result} = process(Prepared),  
4   show(Result).
```

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

Ошибка информативнее, но это exception со всеми его минусами

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

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

```
1 bind(auth(),  
2     lambda user_id: bind(send(usr_id, "logged"),  
3     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`

```
1 def auth():
2     if authed():
3         return (True, "usr42")
4     else:
5         return (False, "not_authed")
6
7 def send(usr, msg):
8     return (True, do_send(usr, msg))
9
10 def bind((is_ok, value), f):
11     if is_ok:
12         return f(value)
13     else:
14         return (False, value)
```

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

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

```
1 def ignorant_auth():  
2     return "usr42"  
3  
4 bind(ret(ignorant_auth()),  
5     lambda user_id: bind(send(usr_id, "logged"),  
6         lambda status: status))
```

- ▶ 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

- ▶ 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

- ▶ 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'a, 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

```
1 def my_call(f, err, *args):
2     try:
3         return f(*args)
4     except Exception as e:
5         raise MyException(False, (err, e))
6
7 def my_return(x):
8     raise MyException(True, x)
9
10 def test():
11     try:
12         conn = my_call(connect_db, "can't connect")
13         data = my_call(make_req, "req error", conn)
14         my_return(data)
15     except MyException as e:
16         return e.result if e.is_ok else e.error
```

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

- ▶ 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

```
4  try
5      {Method, TaskName, VarSpecs} =
6          ?Z_CATCH({_, _, _} = lists:keyfind(Method, 1, TaskSpecs),
7                  bad_method),
8      TaskVarsRoute =
9          ?Z_CATCH([fetch_var(RouteVar, RouteVarType, Bindings)
10                  || {RouteVar, RouteVarType} <- RouteVars],
11                  bad_route),
12      TaskVars = [?Z_CATCH(fetch_var(Var, VarType, QSVals),
13                          {bad_var, Var})
14                  || {Var, VarType} <- VarSpecs],
15      z_return(rnbwdash_task:create(...))
16  catch
17      ?Z_OK(Task) -> form_reply(run_task(Task), Errors, Req@);
18      ?Z_ERROR(Err) -> form_error(Err, Req@)
19  end
```

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

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

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

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