Обработка ошибок — общие соображения и грязные подробности

Дмитрий Грошев

Application Developer Days

11.05.2012

Вступление

Ошибки

Все делают ошибки, однако некоторые думают об ошибках так:

Ошибки

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
  - снова о статике и немного о монадах
  - велосипеды
  - пример кода

Часть 1: общие соображения

Классификация

Наиболее очевидная классификация:

- времени компиляции
- времени выполнения

Ремарка о балансе

Когда ловить ошибки?

- во время компиляции — код либо сложнее, либо многословнее
- во время выполнения — падает надёжность

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?»


```java
public class AddNumbers {
    public static int add() {
        int a = 1;
        String b = "b";
        return a + b;
    }
}
```

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

«Где типы?»


```haskell
my_sum = a + b
   where a = 1
         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


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

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

Оптимистичная типизация

```
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

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


```
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

Зависимые типы


```
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
- сложные компиляторы, очень сложно писать

«Сложные» ошибки: резюме

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

Good enough

Нужно:

- признать неизбежность ошибок
- проектировать весь стек технологий с учётом неизбежности *неожиданных* ошибок
- разделять обработку ошибок для отображения и для сохранения работоспособности системы в целом

Ремарка о работоспособности

- сначала работоспособность
- потом отображение

Хороший пример — CGI и HTTP 500 вместо падения сервера

Хьюстон, у нас проблема

Ошибка произошла. Что делать?

- отбросить испорченные данные вместо сохранения
- перезапуститься
- let it crash (it will crash anyway)

Цена перезапуска

Необходимо минимизировать цену перезапуска

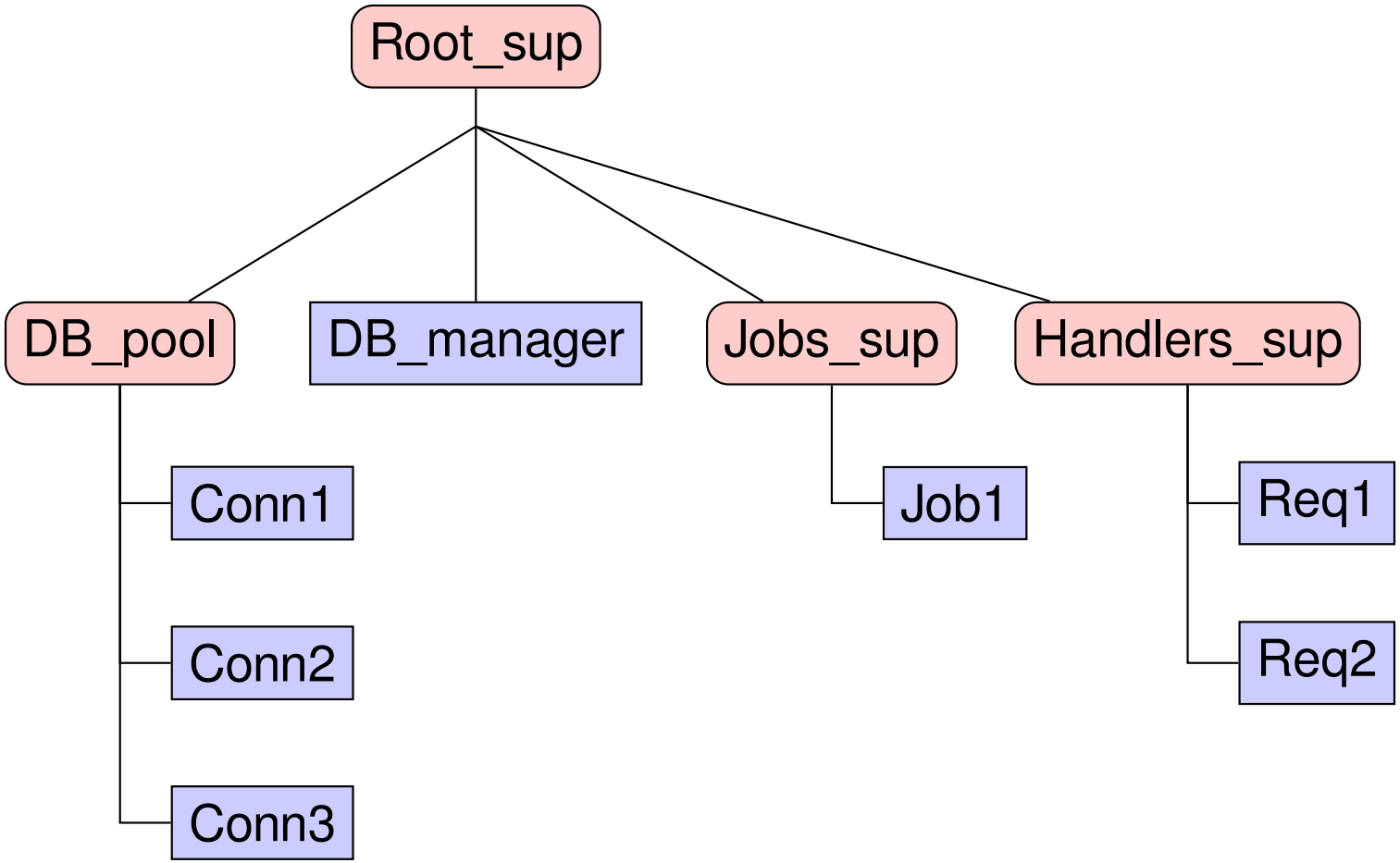
- изоляция потоков исполнения
- изоляция данных
- асинхронный message-passing
- никаких глобальных event loop'ов
- никому нельзя доверять

Логика перезапуска

Необходимо контролировать логику перезапуска

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

Supervisor 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

Часть 2: грязные подробности

Обработка кодов ошибок


```
1  const int CODE_ONE = 1;
2  const int CODE_TWO = 2;
3
4  int foo() {
5    if (bar()) {
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

Null


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

- null *гораздо* хуже кодов возврата
- компилятор контролирует обработку возвращаемых значений, но не null
- Тони Хоар (создатель Algol'a) считает введение null своей худшей ошибкой
- null не является типом, и потому относится к «сложным» ошибкам значений термов

Ремарка: тотальность

Тотальность — свойство функции всегда возвращать что-то осмысленное.

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)
```

Тотальность помогает изолировать ошибки и отлаживать код

О потоках

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

Error flow

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)
```

Error flow

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

Метки успешности

req_handlers.py:

```
1  def handle_req(req):
2      (is_ok, result) = 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, 0)
3      return (True, store(data))
```

Метки успешности

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

Проблема


```
1  def handle(data):
2      (is_ok, foo_result) = foo(data)
3      if not is_ok:
4          return (False, data)
5      (is_ok, bar_result) = bar(foo_result)
6      if not is_ok:
7          return (False, data)
8      return baz(bar_result)
```

Двойственная природа Exception'a

Exception выполняет 2 функции:

- оповещение вызывающего об ошибке
- прерывание исполнения

Можно ли решить вторую проблему с метками успешности?

Pattern matching

```
1  def handle(data):
2      (is_ok, foo_result) = foo(data)
3      if not is_ok:
4          return False
5      (is_ok, bar_result) = bar(foo_result)
6      if not is_ok:
7          return False
8      return baz(bar_result)
```

```
TypeError: 'bool' object is not iterable
```

Поток выполнения прерывается, но ошибка неинформативна

Pattern matching


```
1 handle(Data) ->
2     {ok, FooResult} = foo(Data),
3     {ok, BarResult} = bar(FooResult),
4     baz(BarResult).
```

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

Ошибка информативнее, но это exception со всеми его минусами

Смысл точки с запятой


```
1 a = foo();
2 b = bar(a, "baz");
```

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

Функция bind

```
1 bind(foo(),
2      lambda a: bind(bar(a, "baz"),
3                     lambda b: b))
```

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

Функция bind


```
1  def foo():
2      return (True, "foo")
3
4  def bar(str1, str2):
5      return (True, str1 + str2)
6
7  def bind((is_ok, value), f):
8      if is_ok:
```

Функция return


```
    else:
        return False
```

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

Функция return


```
1  def ret(value):
2      return (True, value)
3
4  def ignorant_foo():
5      return "foo"
6
```

Монада Maybe

- сочетание соглашения о метках успешности выполнения, *bind* и *return* образует монаду (в данном случае Maybe)
- 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

Эмуляция монад

Промежуточный вариант:

```
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_request, "req error", conn)
14         my_return(data)
15     except MyException as e:
16         return e.result if e.is_ok else e.error
```

Эмуляция монад

- нет оверхеда на создание анонимных функций
- функция *test* тотальна
- catch-all малы и не затрудняют дебаг

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


```
handle(Req, #state{...}=State) ->
    {Bindings, Req@} = cowboy_http_req:bindings(Req),
    {QSVals, Req@} = cowboy_http_req:qs_vals(Req@),
    Req@ = 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,
    {ok, Req@, State}.
```

Почему это важно


```
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
```

Почему это важно


```
1  -define(Z_CATCH(EXPR, ERROR),
2          try
3              EXPR
4          catch
5              _:_ -> throw({z_throw, {error, ERROR}})
6          end).
```

Почему это важно


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

Почему это важно


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

Почему это важно


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