Язык программирования Elixir

- Виртуальная машина BEAM, управление памятью и обработка ошибокВыполнили Фролов Иван и Ткачев Михаил, гр. 5030102/20202

Сердце системы — Виртуальная машина BEAM

- Elixir компилируется в байт-код для виртуальной машины BEAM (Bogdan/Björn's Erlang
- Ключевые особенности BEAM:
- Abstract Machine). Важно понять цепочку:
-
- 1. Не стековая, а регистровая виртуальная машина (Использует виртуальные регистры (x0, x1, y0 и т.д.) для хранения аргументов и промежуточных значений)
- * Вы пишете код на Elixir.
- * Он транслируется в Erlang AST (абстрактное синтаксическое дерево).
- 2. Модель акторов в основе.
- * Компилятор Erlang превращает его в байт-код BEAM.
- 3. Планировщик (Scheduler) и вытесняющая многозадачность
- * BEAM выполняет этот байт-код
- 4. Менеджер ввода/вывода (I/O Manager).

Управление памятью в Elixir/BEAM

- 1. Изоляция и приватные кучи.
- * Каждый процесс имеет свою собственную кучу (private heap). Туда попадают все данные, которые создаёт процесс: кортежи, списки, структуры и т.д.
- * Память процессов НЕ пересекается. Один процесс не может «испортить» данные другого. Это железобетонная гарантия.
-
- 2. Сборка мусора (Garbage Collection, GC) — децентрализованная.
- * Сборщик мусора работает для каждого процесса независимо.
- * Простой и быстрый
- * Нет «стопа мира» (stop-the-world). Пока в одном процессе идёт сборка мусора, тысячи других процессов работают

Два типа данных

* Термы (Terms): Живут в приватной куче процесса. Копируются при отправке в сообщении (т.н. «copy semantics» — семантика копирования). Это быстро для небольших данных.

* Бинарные данные (Binaries) > 64 байт: Живут в общей куче (shared heap или binary heap). Это большие строки, картинки, файлы.

При отправке такого бинарного в сообщении, процесс получает только ссылку на него в общей куче.

Счётчик ссылок (reference counting) отслеживает, какие процессы используют бинар. Когда счётчик достигает нуля, память освобождается.

Иерархия памяти процесса BEAM:

Процесс BEAM

├── Stack (стек вызовов)

├── Private Heap (приватная куча)

│ ├── Малые термы (< 64 байт)

│ ├── Указатели на бинарные данные

│ └── Другие структуры данных

└── (ссылки) → Binary Heap (куча бинарных данных)

  ├── Binary 1 (большая строка, 1KB)

  ├── Binary 2 (картинка, 50KB)

  └── Binary 3 (файл, 5MB)

Разделение на приватные кучи и общие бинарии — это компромисс между изоляцией и эффективностью.

elixir

Обработка ошибок в Elixir

Три типа исключений

1. Ошибки (Errors) — неожиданные ситуации. Процесс умирает при ошибке (1 / 0 # (ArithmeticError) division by zero)

2. Выходы (Exits) — контролируемое завершение

```
# Явный выход процесса
Process.exit(pid, :shutdown)
# Связанные процессы умирают вместе
spawn_link(fn ->
  Process.exit(self(), :boom)  # Убьёт и родительский процесс
end)
```

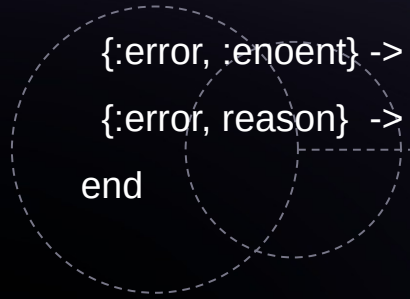
3. Вызов исключений (Throws) — нестандартный возврат (throw(:early_return))

elixir

Паттерны обработки исключений

Паттерн 1: {:ok, result} | {:error, reason}

```
with {:ok, content} <- File.read("file.txt"),
     {:ok, parsed}  <- parse_content(content),
     {:ok, result}  <- save_to_db(parsed) do
  {:ok, result}
else
  {:error, :enoent} -> {:error, "Файл не найден"}
  {:error, reason}  -> {:error, "Ошибка: #{reason}"}
end
```



Паттерны обработки исключений

Паттерн 2: Падение + супервизор

```
defmodule Worker do
  use GenServer

  def handle_call(:dangerous, _from, state) do
    # Если здесь ошибка — процесс УМРЁТ
    result = 1 / 0  # ArithmeticError!
    {:reply, result, state}
  end
end

# Супервизор в конфигурации
children = [
  {Worker, []}  # Автоматический перезапуск при падении
```

Паттерны обработки исключений

Паттерн 3: Мониторинг процессов

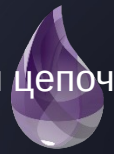
```
# Создаём и мониторим процесс
pid = spawn(fn -> :timer.sleep(1000) end)
ref = Process.monitor(pid)

# Получаем сообщение о смерти
receive do
  {:DOWN, ^ref, :process, ^pid, reason} ->
    IO.puts("Процесс умер: #{reason}")
    # Запускаем новый
end
```

Когда использовать try/rescue

- # ТОЛЬКО для интеграции или логирования
- try do
- external_library.call() # Может падать странным образом
- rescue
- error in [RuntimeError] ->
- # Логируем и поднимаем дальше
- Logger.error("Внешняя ошибка: #{Exception.message(error)}")
- raise error # Пусть падает дальше!
- end

- Ключевые принципы
-
- 1. Ошибки — падают, не ловятся (кроме интеграции)
- 2. Выходы — для контролируемого завершения
- 3. Вызов исключения — почти никогда не используется
- 4. Основная надёжность — через супервизоры, а не try/catch
- 5. Функции возвращают {:ok, result} | {:error, reason}
- 6. `with` — лучший друг для цепочек операций



Источники

- elixir-lang.org – официальный сайт языка
- Wikipedia – статья Elixir (programming language)