

Язык программирования Elixir



- Конкурентность / параллелизм
- Выполнили Фролов Иван и Ткачев Михаил, гр. 5030102/20202

Модель акторов и легковесные процессы

- Ключевые преимущества:

- - □ Легковесность: 2-3 КБ памяти против 1-2 МБ у потоков
- - □ Изоляция: Нет разделяемой памяти — нет race conditions
- - □ Асинхронность: Общение только через обмен сообщениями
 - # Создаем процесс - это дешево и быстро!
 - ```
pid = spawn(fn ->
```
  - ```
  IO.puts("Я независимый процесс! Мой
```
 - ```
 идентификатор: #{inspect(self())}")
```
  - ```
  end)
```
 -
 - # Пояснение: spawn создает новый процесс и сразу возвращает управление.
 - # Процесс работает асинхронно, не блокируя основной поток.



Параллелизм. Пример

- Вместо сложных циклов с пулами потоков - `Task.async`` и `Task.await``
 - `defmodule ImageProcessor do`
 - `def process_images(urls) do`
 - `urls`
 - `# Шаг 1: Запускаем все задачи параллельно`
 - `|> Enum.map(&Task.async(fn -> download_and_process(&1) end))`
 - `# Шаг 2: Ждем результаты всех задач`
 - `|> Enum.map(&Task.await/1)`
 - `end`
 -
 - `defp download_and_process(url) do`
 - `# Имитация тяжелой операции (загрузка + обработка)`
 - `:timer.sleep(1000)`
 - `"#{url} - обработан"`
 - `end`
 - `end`



Кооперативная vs Вытесняющая многозадачность

Кооперативная многозадачность (JavaScript async/await)

Характеристики:

- ☐ Процессы добровольно отдают управление (`await``, `yield``)
- ☐ Если процесс "жадный" — вся система зависает
- ☐ Нужно явно писать `async/await``
- ☐ Один зависший Promise блокирует event loop

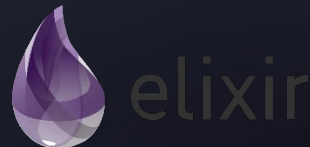


Кооперативная vs Вытесняющая многозадачность

Вытесняющая многозадачность (Elixir Processes)

Характеристики:

- □ Планировщик принудительно переключает процессы
- □ Нельзя "заблокировать" систему одним процессом
- □ Прозрачно для программиста — не нужно явно отдавать управление
- □ Каждый процесс получает квант времени (~2000 редукций)



Кооперативная vs Вытесняющая многозадачность

Вытесняющая многозадачность (Elixir Processes)

Elixir - вытесняющая модель

```
defmodule Worker do
```

```
  def loop do
```

```
    receive do
```

```
      msg -> process_message(msg)
```

```
    end
```

```
    loop() # BEAM CAM решает, когда переключиться на другой процесс
```

```
  end
```

```
  defp process_message(_) do
```

```
    # Даже если здесь бесконечный цикл...
```

```
    # BEAM принудительно переключит на другой процесс через ~2000 инструкций
```

```
  end
```

```
end
```



Планировщик BEAM vs Планировщик ОС

BEAM имеет собственные планировщики (обычно 1 на ядро CPU)

`:erlang.system_info(:schedulers) # => 8` (на 8-ядерном процессоре)

- Каждый планировщик управляет своей очередью процессов
- Переключение происходит ПОСЛЕ 2000 редукций (инструкций BEAM)

Ключевое отличие:

- ОС планировщик: управляет десятками/сотнями потоков
- BEAM планировщик: управляет миллионами виртуальных процессов



Сколько процессов создавать

1. "Процесс на подключение" (Web серверы) - сколько одновременных подключений (Типично: 10,000 - 100,000 процессов)
2. "Процесс на задачу" (Фоновая обработка) - количество CPU ядер, умноженное на 2 – 4 (Пример: 8 ядер \times 3 = 24 параллельных задачи)
3. "Процесс на сущность" (Игровые серверы, чаты) - столько процессов, сколько активных сущностей (Может быть: тысячи-десятки тысяч)

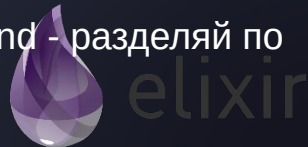


Когда НЕ создавать много процессов

- CPU-bound задачи
- # ПЛОХО: Слишком мелкое разделение
- ```
defmodule BadCalculator do
```
- ```
  def compute(a, b, c) do
```
- ```
 # Создаем процессы для КАЖДОЙ операции - накладные расходы!
```
- ```
    task1 = Task.async(fn -> a + b end)
```
- ```
 task2 = Task.async(fn -> b * c end)
```
- ```
    {result1, result2} = {Task.await(task1), Task.await(task2)}
```
- ```
 result1 + result2
```
- ```
  end
```
- ```
end
```

- Практическое правило:

- Если задача выполняется  $< 1\text{ms}$  - объединяй
- Если задача I/O bound - разделяй на процессы
- Если задача CPU bound - разделяй по количеству ядер



- # ХОРОШО: Крупные блоки работы

# ВЫВОДЫ

- ☐ **Web серверы:** 10,000+ одновременных подключений
- ☐ **Чаты:** 100,000+ параллельных пользователей
- ☐ **Data processing:** Параллельная обработка потоков данных
- ☐ **Background jobs:** Тысячи фоновых задач

# Источники

- [elixir-lang.org](https://elixir-lang.org) — официальный сайт языка
- Wikipedia — статья Elixir (programming language)

