Marp

惰性求值

Rethink Lazy Evaluation

习以为常的即时求值

```
0..9
|> Enum.map(fn x -> x + 1 end)
|> Enum.map(fn x -> x * 2 == 0 end)
|> Enum.take(3)
```

计算发生的时刻

- 我们需要处理的数据在上一个函数中计算完成后，才会传入下一个函数
- 多个函数的组合帮助我们完成实际的需求

计算发生于每一次的函数调用的内部

惰性求值是什么

```
In programming language theory, lazy evaluation, or call-by-need,
is an evaluation strategy which delays the evaluation of an expression until its value is needed.
```

- 在编程语言理论中，惰性求值（或称为需要时调用）是一种求值策略。
- 它延迟对表达式的求值，直到其值被需要为止

与直觉相反的惰性求值

```
1..1000
|> Stream.map(fn x -> x * x end)
|> Stream.filter(fn x -> rem(x, 2) == 0 end)
|> Enum.take(5)
```

惰性求值中-计算发生的时刻

- 我们需要处理的数据在上一个函数中
  - 将函数存储到惰性的数据结构中
  - 但是不执行计算

计算发生于最终获取值的时刻

惰性求值的应用场景

无限序列

```
def addOne(n) = [n] + addOne(n + 1)
// [1, 2, 3, 4, 5, 6, ...]
infinite_list = addOne(1)
```

大数据集和流式数据集

按需处理数据集的部分内容，而不需要一次性加载整个数据集到内存中。

- 不需要手动的切分Batch
- 天然的并行执行
  - Elixir中的Flow
  - Java中的 Parallel Stream

惰性求值的优势

1. 节省内存(Memory)
   - 使用惰性求值，我们可以逐行读取日志文件并按需处理，
   - 不需要一次性加载整个文件到内存中。

2. 提高性能(CPU)
   - 通过按需计算和处理，惰性求值可以避免不必要的计算和处理的开销，
   - 并行执行提高处理速度和性能

实现惰性求值

函数式编程语言的天然优势

- 函数作为参数，返回值
- 函数的组合

只要计算能够存储就能实现

- 函数（Function）
- 方法 (Method)
- 表达式 (Expression)

一切都是数据

- 函数是数据
  - 只要我们能够将函数进行存储，我们就可以去做延迟计算
  - 我们可以用处理数据的方式来去处理函数

Q & A

- https://en.wikipedia.org/wiki/Lazy_evaluation#:~:text=10 External links-,History,a capability-limited address space.
- https://medium.com/luteceo-software-chemistry/can-programming-be-liberated-from-the-von-neumann-style-932ba107402b