🌊 Streaming-first

A Evolução do elixir-grpc

Adriano Santos

Agenda

O que vamos ver hoje:

Tópico	Descrição
📚 **Contexto**	O que é elixir-grpc
💡 **Conceito**	Por que Streaming-first?
🔧 **Fundamentos**	API GRPC.Stream em detalhes
💻 **Prática**	Exemplos reais e casos de uso
🚀 **Evolução**	Mudanças e próximos passos

elixir-grpc em poucas palavras

🎯 Implementação nativa de gRPC em Elixir

Fundação:

- HTTP/2 (multiplexing, compressão)
- Protocol Buffers (serialização tipada)
- TLS e autenticação

Quatro tipos de RPC:

`Unary` · `Server Stream` · `Client Stream` · `Bidirectional`

Características: Idiomático · Escalável · Fault-tolerant

Tipos de streaming no gRPC

Tipo	Descrição	Fluxo
Unary	Request/Response único	Cliente → Servidor → Cliente
Server Streaming	Servidor envia stream	Cliente → Servidor ⇉ Cliente
Client Streaming	Cliente envia stream	Cliente ⇉ Servidor → Cliente
Bidirectional	Ambos streamam	Cliente ⇄ Servidor

Modelo único no elixir-grpc

O problema do modelo tradicional

- gRPC tratado como request/response
- Streams como exceção
- APIs imperativas
- Baixa composabilidade

Streaming-first

Streams não são um detalhe

- Unary = stream de 1 elemento
- Tudo é fluxo
- Composição funcional

Tipos de streaming no gRPC

- Unary
- Server streaming
- Client streaming
- Bidirectional streaming

Modelo único no elixir-grpc

O papel do GRPC.Stream

- **Abstração baseada em Flow**
- Suporte a backpressure via GenStage
- Transformações funcionais
- Composição de pipelines
- Integração com produtores externos (RabbitMQ, Kafka)

Estrutura da API

Funções de Criação:

- `from/2` - Converte input em Flow com backpressure
- `unary/2` - Converte request única em Flow

Transformadores:

- `map/2`, `filter/2`, `flat_map/2`
- `map_with_context/2`, `map_error/2`
- `reduce/3`, `partition/1`
- `uniq/1`, `uniq_by/2`

Materializadores:

- `run/1` - Executa stream unary

Unary como stream

```elixir
def say_hello(request, materializer) do
  GRPC.Stream.unary(request, materializer: materializer)
  |> GRPC.Stream.map(fn %HelloRequest{name: name} ->
    %HelloReply{message: "Hello, #{name}!"}
  end)
  |> GRPC.Stream.run()
end
```

Conceito: Unary é apenas um stream de 1 elemento

materializer: GRPC.Server.Stream que representa o contexto da requisição

Server-side streaming

```
def stream_numbers(request, materializer) do
  Stream.repeatedly(fn ->
    %NumberReply{value: :rand.uniform(100)}
  end)
  |> Stream.take(5)
  |> GRPC.Stream.from()
  |> GRPC.Stream.run_with(materializer)
end
```

Servidor envia múltiplas mensagens, cliente recebe

Client-side streaming

```
def sum_numbers(input_stream, materializer) do
  GRPC.Stream.from(input_stream)
  |> GRPC.Stream.reduce(
    fn -> 0 end,
    fn %Number{value: v}, acc -> acc + v end
  )
  |> GRPC.Stream.map(fn sum ->
    %SumReply{total: sum}
  end)
  |> GRPC.Stream.run_with(materializer)
end
```

Cliente envia múltiplas mensagens, servidor responde uma vez

Bidirectional streaming

```elixir
def route_chat(input, materializer) do
  # Stream independente do servidor (pode ser RabbitMQ, Kafka, etc)
  {:ok, server_producer} = MyApp.EventProducer.start_link([])

  # Combina input do cliente + eventos independentes do servidor
  GRPC.Stream.from(input,
    join_with: server_producer,
    max_demand: 10
  )
  |> GRPC.Stream.map(fn
    # Mensagens do cliente
    %RouteNote{} = note ->
      %RouteNote{message: "Echo: #{note.message}"}

    # Eventos independentes do servidor
    {:server_event, data} ->
      %RouteNote{message: "Server says: #{data}"}
  end)
  |> GRPC.Stream.run_with(materializer)
end
```

Transformadores: map, filter, flat_map

```elixir
def process_events(input, materializer) do
  GRPC.Stream.from(input)
  |> GRPC.Stream.filter(fn event -> event.priority >= 5 end)
  |> GRPC.Stream.map(fn event ->
    %Event{id: event.id, data: String.upcase(event.data)}
  end)
  |> GRPC.Stream.flat_map(fn event ->
    # Emite múltiplas respostas para cada entrada
    [event, %Event{id: "#{event.id}-copy", data: event.data}]
  end)
  |> GRPC.Stream.run_with(materializer)
end
```

Tratamento de erros no gRPC

Desafio: Streams podem falhar de diversas formas

- Exceções durante processamento
- Validações que falham
- Timeouts de serviços externos
- Erros de parsing/codificação

Solução: `map_error/2` captura e transforma erros

Tipos de erro capturados:

```
{:error, reason}                        # Erros explícitos
{:error, {:exception, exception}}       # Exceções lançadas
{:error, {kind, reason}}                # throw/exit
```

map_error: Transformando exceções em erros gRPC

```elixir
def safe_processing(input, materializer) do
  GRPC.Stream.from(input)
  |> GRPC.Stream.map(fn data ->
    # Validação pode lançar exceção
    if data.value < 0, do: raise("Negative value!")

    # Processamento pode falhar
    case process(data) do
      {:ok, result} -> result
      {:error, reason} -> {:error, reason}
    end
  end)
  |> GRPC.Stream.map_error(fn
    # Captura exceções
    {:error, {:exception, %RuntimeError{message: msg}}} ->
      GRPC.RPCError.exception(
        status: :invalid_argument,
        message: "Validation failed: #{msg}"
      )

    # Captura erros explícitos
    {:error, :not_found} ->
      GRPC.RPCError.exception(status: :not_found)

    # Qualquer outro erro
    {:error, reason} ->
      GRPC.RPCError.exception(
        status: :internal,
        message: "Processing error: #{inspect(reason)}"
      )
```

Estratégias de tratamento de erros

1. Fail-fast - Para o stream no primeiro erro:

```
# Sem map_error - primeira exceção interrompe tudo
GRPC.Stream.from(input)
|> GRPC.Stream.map(&may_fail/1)
|> GRPC.Stream.run_with(materializer)
```

2. Transformar erros em respostas válidas:

```
# Continua processando mesmo com erros
GRPC.Stream.from(input)
|> GRPC.Stream.map(&may_fail/1)
|> GRPC.Stream.map_error(fn {:error, _} ->
  %Response{status: "error", data: nil}
end)
|> GRPC.Stream.run_with(materializer)
```

Side-effects: effect

```elixir
def stream_with_effects(input, materializer) do
  # Pode ser um GenServer, Agent, ou qualquer processo
  {:ok, metrics_collector} = MetricsCollector.start_link()
  {:ok, notifier} = EventNotifier.start_link()

  GRPC.Stream.from(input)
  |> GRPC.Stream.effect(fn msg ->
    # Logging
    Logger.info("Processing: #{inspect(msg)}")
  end)
  |> GRPC.Stream.map(&transform/1)
  |> GRPC.Stream.effect(fn result ->
    # Envia mensagem para processo coletor de métricas
    send(metrics_collector, {:metric, :processed, 1})

    # Notifica outros processos
    send(notifier, {:event, :item_processed, result})
  end)
  |> GRPC.Stream.run_with(materializer)
end
```

effect: Casos de uso comuns

```elixir
def monitored_stream(input, materializer) do
  GRPC.Stream.from(input)
  # 1. Debug/Observabilidade
  |> GRPC.Stream.effect(fn item ->
    IO.inspect(item, label: "Stream item")
  end)

  # 2. Telemetry events
  |> GRPC.Stream.effect(fn item ->
    :telemetry.execute([:app, :stream, :item], %{count: 1}, %{type: item.__struct__})
  end)

  # 3. Notificar processo externo
  |> GRPC.Stream.effect(fn item ->
    send(some_pid, {:item_seen, item.id})
  end)

  # 4. Cachear valores
  |> GRPC.Stream.effect(fn item ->
    Cache.put("item:#{item.id}", item)
  end)

  |> GRPC.Stream.map(&process/1)
  |> GRPC.Stream.run_with(materializer)
end
```

Composição de streams

```elixir
def pipeline(input, materializer) do
  input
  |> GRPC.Stream.from(max_demand: 20)
  |> validate_input()
  |> enrich_with_data()
  |> persist_to_db()
  |> format_response()
  |> GRPC.Stream.run_with(materializer)
end

defp validate_input(stream) do
  GRPC.Stream.filter(stream, &valid?/1)
end

defp enrich_with_data(stream) do
  GRPC.Stream.map(stream, &fetch_metadata/1)
end
```

Backpressure e max_demand

```elixir
def controlled_stream(input, materializer) do
  # Limita demanda para evitar sobrecarga
  GRPC.Stream.from(input, max_demand: 10)
  |> GRPC.Stream.map(&expensive_operation/1)
  |> GRPC.Stream.run_with(materializer)
end
```

max_demand controla quantos itens são solicitados por vez

- Evita sobrecarga de memória
- Controla taxa de processamento
- GenStage gerencia automaticamente

Integração com produtores externos

```elixir
def stream_from_queue(input, materializer) do
  # Inicia producer externo (RabbitMQ, Kafka, etc)
  {:ok, queue_pid} = MyApp.RabbitMQ.Producer.start_link([])

  GRPC.Stream.from(input,
    join_with: queue_pid,
    max_demand: 10
  )
  |> GRPC.Stream.map(&handle_message/1)
  |> GRPC.Stream.run_with(materializer)
end

defp handle_message({_, msg}), do: msg
defp handle_message(event), do: %Event{data: inspect(event)}
```

:join_with permite mesclar streams gRPC com fontes externas

Contexto e Headers

```elixir
def authenticated_stream(input, materializer) do
  headers = GRPC.Stream.get_headers(materializer)

  GRPC.Stream.from(input, materializer: materializer)
  |> GRPC.Stream.map_with_context(fn headers, msg ->
    user = headers["user-id"]
    authorize(user, msg)
  end)
  |> GRPC.Stream.run_with(materializer)
end
```

Acesso a headers HTTP/2 e contexto da requisição

ask: Comunicação com GenServer

```elixir
defmodule PricingService do
  use GenServer

  def start_link(_opts) do
    GenServer.start_link(__MODULE__, %{}, name: __MODULE__)
  end

  def init(_), do: {:ok, %{}}

  # Recebe {:request, item, from} e responde {:response, result}
  def handle_info({:request, %Product{id: id}, from}, state) do
    # Consulta preço no banco, cache, API externa, etc.
    price = calculate_price(id)
    send(from, {:response, price})
    {:noreply, state}
  end

  defp calculate_price(id), do: :rand.uniform(100) * 1.0
end
```

ask: Integrando com o stream

```elixir
def enrich_with_prices(input, materializer) do
  # isto provavelmente estaria em outro lugar
  {:ok, pricing_pid} = PricingService.start_link([])

  GRPC.Stream.from(input)
  # Consulta o GenServer para cada produto
  |> GRPC.Stream.ask(pricing_pid, timeout: 5000)
  |> GRPC.Stream.map(fn
    # Sucesso: enriquece o produto com preço
    {:response, price} ->
      %ProductReply{price: price, status: :available}

    # Timeout: serviço não respondeu
    {:error, :timeout} ->
      %ProductReply{price: 0.0, status: :unavailable}

    # Processo morreu
    {:error, :process_not_alive} ->
      %ProductReply{price: 0.0, status: :error}
  end)
  |> GRPC.Stream.run_with(materializer)
end
```

ask vs ask!: Quando usar cada um

ask/3 - Retorna {:error, reason}:

```
GRPC.Stream.ask(stream, service_pid, timeout: 5000)
|> GRPC.Stream.map(fn
  {:response, data} -> process(data)
  {:error, :timeout} -> fallback_value()
  {:error, :process_not_alive} -> default_value()
end)
```

✅ **Recomendado:** Permite tratamento de erro gracioso

ask!/3 - Levanta exceção:

```
GRPC.Stream.ask!(stream, service_pid, timeout: 5000)
|> GRPC.Stream.map(fn {:response, data} ->
  process(data)  # Se falhar, CRASH no worker!
end)
```

Exemplo: Enriquecimento com múltiplos serviços

```elixir
def enrich_product_stream(input, materializer) do
  {:ok, pricing} = PricingService.start_link([])
  {:ok, inventory} = InventoryService.start_link([])

  GRPC.Stream.from(input)
  # Consulta preço
  |> GRPC.Stream.ask(pricing, timeout: 2000)
  |> GRPC.Stream.map(fn
    {:response, price} -> {:ok, price}
    {:error, _} -> {:ok, 0.0}  # fallback
  end)
  # Consulta estoque
  |> GRPC.Stream.ask(inventory, timeout: 2000)
  |> GRPC.Stream.map(fn
    {:response, stock} -> %ProductReply{price: price, stock: stock}
    {:error, _} -> %ProductReply{price: price, stock: 0}
  end)
  |> GRPC.Stream.run_with(materializer)
end
```

Exemplo completo: Chat bidirecional

```elixir
defmodule ChatService do
  use GRPC.Server, service: Chat.Service

  def route_chat(input, materializer) do
    # Stream de entrada com backpressure
    GRPC.Stream.from(input, max_demand: 10)
    # Log de mensagens recebidas
    |> GRPC.Stream.effect(fn msg ->
      Logger.info("Received: #{inspect(msg)}")
    end)
    # Validação
    |> GRPC.Stream.filter(&valid_message?/1)
    # Processamento
    |> GRPC.Stream.map(&process_chat_message/1)
    # Tratamento de erros
    |> GRPC.Stream.map_error(&handle_chat_error/1)
    # Envia respostas
    |> GRPC.Stream.run_with(materializer)
  end
end
```

Exemplo: Stream com Cache

```elixir
def cached_lookup(input, materializer) do
  GRPC.Stream.from(input)
  |> GRPC.Stream.map(fn %LookupRequest{key: key} ->
    case Cache.get(key) do
      {:ok, value} ->
        %LookupReply{value: value, from_cache: true}
      :miss ->
        value = Database.fetch(key)
        Cache.put(key, value)
        %LookupReply{value: value, from_cache: false}
    end
  end)
  |> GRPC.Stream.run_with(materializer)
end
```

Exemplo: Fan-out com múltiplos consumidores

```elixir
def broadcast_events(input, materializer) do
  GRPC.Stream.from(input)
  |> GRPC.Stream.flat_map(fn event ->
    # Cada evento gera múltiplas notificações
    subscribers = Subscribers.list(event.topic)

    Enum.map(subscribers, fn sub ->
      %Notification{
        subscriber_id: sub.id,
        event: event
      }
    end)
  end)
  |> GRPC.Stream.run_with(materializer)
end
```

Opções avançadas do from/2

```elixir
GRPC.Stream.from(input,
  # Backpressure
  max_demand: 50,

  # Producer externo
  join_with: external_producer_pid,

  # Dispatcher customizado
  dispatcher: GenStage.DemandDispatcher,

  # Propagar contexto para o Flow
  propagate_context: true,

  # Contexto do servidor
  materializer: materializer
)
```

Por que isso importa?

Modelo mental unificado:

- Unary, Server, Client e Bidirectional são todos streams
- Mesma API para todos os casos
- Composição funcional natural

Poder do BEAM:

- Backpressure automático via GenStage
- Processos isolados para cada stream
- Escalabilidade horizontal

Código idiomático:

- Pipeline operators (`|>`)

Quando usar streaming?

Casos de uso ideais:

- **Eventos em tempo real** - notificações, logs, telemetria
- **Processamento de grandes volumes** - ETL, agregações
- **Sistemas reativos** - dashboards, monitoramento
- **Integração com message queues** - Kafka, RabbitMQ
- **Upload/Download de arquivos** - chunked transfer

Quando evitar:

- Operações simples request/response (use unary)
- Dados pequenos que cabem em uma mensagem

Vantagens da abordagem Streaming-first

1. **Consistência**: Uma API para todos os tipos de RPC
2. **Composabilidade**: Combinar streams facilmente
3. **Testabilidade**: Streams são mais fáceis de testar
4. **Performance**: Backpressure evita sobrecarga
5. **Integração**: Compatível com Flow, GenStage, Broadway

Comparação: Antes vs Agora

Antes (imperativo)

```
def chat(stream) do
  receive_loop(stream, [])
end

defp receive_loop(stream, acc) do
  case receive_message(stream) do
    :end_stream -> send_final_reply(stream, acc)
    msg ->
      send_reply(stream, process(msg))
      receive_loop(stream, [msg | acc])
  end
end
```

Agora (declarativo)

```
def chat(input, materializer) do
  GRPC.Stream.from(input)
  |> GRPC.Stream.map(&process/1)
  |> GRPC.Stream.run_with(materializer)
end
```

Recursos da API GRPC.Stream

Funções de Criação:

- `from/2` - Stream com múltiplos elementos
- `unary/2` - Stream de elemento único

Materializadores:

- `run/1` - Para unary streams
- `run_with/3` - Para streaming responses

Transformadores:

- `map/2`, `filter/2`, `flat_map/2`, `reduce/3`
- `map_with_context/2`, `map_error/2`
- `uniq/1`, `uniq_by/2`, `partition/1`

Debugging e observabilidade

```
def observable_stream(input, materializer) do
  GRPC.Stream.from(input)
  |> GRPC.Stream.effect(fn msg ->
    :telemetry.execute([:grpc, :message, :received], %{}, %{msg: msg})
  end)
  |> GRPC.Stream.map(fn msg ->
    start_time = System.monotonic_time()
    result = process(msg)
    duration = System.monotonic_time() - start_time

    :telemetry.execute(
      [:grpc, :processing, :complete],
      %{duration: duration},
      %{}
    )

    result
  end)
  |> GRPC.Stream.run_with(materializer)
end
```

Testing streams

```elixir
defmodule ChatServiceTest do
  use ExUnit.Case

  test "processes messages correctly" do
    input = [
      %ChatMessage{text: "hello"},
      %ChatMessage{text: "world"}
    ]

    result =
      GRPC.Stream.from(input)
      |> ChatService.process_messages()
      |> GRPC.Stream.to_flow()
      |> Enum.to_list()

    assert length(result) == 2
    assert Enum.all?(result, &match?(%ChatReply{}, &1))
  end
end
```

Padrões comuns

1. Validação + Transformação + Persistência:

```
input
|> GRPC.Stream.from()
|> GRPC.Stream.filter(&valid?/1)
|> GRPC.Stream.map(&transform/1)
|> GRPC.Stream.effect(&persist/1)
|> GRPC.Stream.run_with(materializer)
```

2. Fan-out paralelo:

```
input
|> GRPC.Stream.from()
|> GRPC.Stream.partition(stages: 4)
|> GRPC.Stream.map(&expensive_operation/1)
|> GRPC.Stream.run_with(materializer)
```

Limitações e cuidados

⚠️ Partition com unbounded streams:

- Pode impactar memória
- Use apenas quando necessário
- Ver Flow.partition/2

⚠️ uniq com streams infinitos:

- Mantém histórico em memória
- Considere janelas de tempo

⚠️ ask!/3 vs ask/3:

- `ask!/3` pode crashar o worker
- Prefira `ask/3` em produção

Evolução da biblioteca

Versão atual (0.11.x):

- API streaming-first estável
- Suporte completo a Flow
- Backpressure nativo
- Integração com GenStage

Futuro:

- Mais operadores convenientes
- Melhor integração com Broadway
- Otimizações de performance
- Compatibilidade com HTTP/3 (QUIC)

Recursos e referências

Documentação:

- hexdocs.pm/grpc/GRPC.html
- hexdocs.pm/grpc/GRPC.Stream.html

Conceitos relacionados:

- GenStage - Backpressure
- Flow - Pipeline paralelo
- Broadway - Data ingestion

Repositório:

- github.com/elixir-grpc/grpc

Conclusão

Streaming-first não é apenas um nome:

- ✅ API unificada para todos tipos de RPC
- ✅ Composição funcional natural
- ✅ Backpressure automático
- ✅ Integração com ecossistema Elixir
- ✅ Código declarativo e testável

Mensagem principal:

> Todo RPC é um stream. Unary é apenas um stream de 1 elemento.

Resultado:

- Código mais limpo e idiomático

Obrigado!

GitHub: elixir-grpc/grpc

Perguntas?