Erlang – concorrência

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos Departamento de Informática Universidade do Minho



Criação de processos

- Um processo em Erlang é uma entidade auto-contida;
- Processos não partilham memória; toda a comunicação é por mensagens;
- Um processo pode ser criado com variantes de spawn:
 - passando nome do módulo, da função e argumentos da função;

```
Pid = spawn (Modulo, NomeFuncao, Args)
```

passando closure:

```
F = fun() \rightarrow ... end,
Pid = spawn(F)
```

 Um processo termina quando a função especificada termina, sendo perdido o seu valor de retorno;



Envio de mensagens

- Erlang segue o modelo de actores:
 - cada processo tem uma mailbox fila de mensagens;
 - mensagens são enviadas especificando o processo de destino não é usado o conceito de canal.
- O envio é feito com a primitiva send, com a sintaxe:

Pid! Msg

- Pid é o identificador do processo de destino;
- Msg é um termo Erlang arbitrário (incluíndo closures);
- O envio é assincrono e não causa erro no emissor:
 - não espera que a mensagem chegue ao destino ou seja entregue;
 - mesmo que o processo destino já tiver terminado, o emissor não é notificado e a mensagem perde-se.



Recepção de mensagens

- Cada processo tem uma mailbox: fila de mensagens;
- As mensagens são armazenadas pela ordem de chegada;
- Uma mensagem pode ser removida da mailbox com:

```
receive
  Pattern1 [when Guard1] ->
    Seq1;
    ...
  PatternN [when GuardN] ->
    SeqN
end
```

- O mecanismo utiliza pattern matching:
 - é removida a primeira mensagem da mailbox que faça match com um padrão, desde que a guarda seja satisfeita, quando utilizada;
 - com sucesso, a sequência de acções correspondente é executada;
 - caso contrário, o processo bloqueia até chegar mensagem;



Ordem de entrega e ordem das guardas

- Para escolher uma mensagem a entregar:
 - primeiro é considerada a ordem das mensagens na mailbox;
 - para cada mensagem é considerada a ordem das guardas;
- Exemplo:
 - se a mailbox tiver as mensagens:

```
[msg1, msg2, msg4]
```

e executarmos o código:

```
receive
msg4 ->
msg2 ->
msg2 ->
```

- é feito match com msg2, apesar de a guarda ser a última;
- Conclusão: a ordem das guardas não podem ser usadas directamente para dar prioridade a mensagens;



Receber mensagens de um dado processo

- Não há mecanismo especial para escolher a origem;
- A facilidade geral de pattern matching pode ser usada, enviando mensagens etiquetadas com Pids;
- Quem envia pode mandar o seu Pid na mensagem:

```
Pid ! {self(), ... } % self() - Pid do processo corrente
```

 O receptor faz receive, especificando numa variável previamente atribuída o Pid da origem:

```
From = ...
receive
  {From, Msg} ->
    doSomething(Msg)
end
```



Timeouts na recepção

 A primitiva receive pode ter opcionalmente especificado um timeout:

```
receive
...
after TimeoutExpr ->
   Seq
end
```

- A expressão representa um tempo, em milisegundos;
- Se nenhuma mensagem for seleccionada, passado esse tempo as acções em Seq são executadas;
- Casos especiais para o valor de timeout:
 - o átomo infinity significa esperar para sempre;
 - O valor 0 significa timeout imediato, depois de consideradas as mensagens já na mailbox;



Usos para timeout

• Função sleep, para adormecer processo T milisegundos:

```
sleep(T) ->
  receive
  after T ->
    true
end.
```

Função para esvaziar a mailbox de um processo:

```
flush() ->
  receive
    -->
     flush()
  after 0 ->
     true
end.
```



Usos para timeout – prioridade de mensagens

- Suponhamos que queremos ter prioridades na recepção de diferentes mensagens;
- Para dar maior prioridade a msg1, depois msg2, ...:

```
receive
   msg1 ->
        Seq1
   after 0 ->
        receive
        msg2 ->
        Seq2
   after 0 ->
        end
end
```



Processos registados

- Muitas vezes é conveniente poder usar um nome para contactar um processo, sem necessitar de saber o seu Pid;
- Um exemplo é para comunicar com processos de outro nó, não conhecendo o Pid mas apenas a identificação do nó;
- Erlang fornece, em cada nó, um pequeno serviço de nomes, que mantém um registo com associações de nomes a Pids;
- As primitivas disponibilizadas são:
 - register (Name, Pid) associa o átomo Name a Pid;
 - unregister (Name) remove a associação;
 - whereis (Name) devolve o Pid correspondente, ou undefined;
 - registered() devolve a lista dos nomes registados;
- A primitiva send permite o uso de nomes registados:

```
Pid = spawn(...),
register(server, Pid),
Msg = ...,
server ! Msg
```



Exemplo: contador

- Queremos implementar um processo "contador";
- Este processo deve atender dois tipos de mensagens:
 - incrementar o contador:
 - saber o valor do contador;
- Função counter especifica o comportamento do processo:

```
counter(Val) ->
  receive
  increment ->
    counter(Val + 1);
  {value, From} ->
    From ! Val,
    counter(Val)
end.
```

- Val mantém o valor actual;
- Quem quer saber o resultado passa o seu Pid na mensagem para lhe ser enviada resposta;
- É usada tail recursion;



Exemplo: contador - cliente

- Função client incrementa um contador algumas vezes;
- Tem dois parâmetros:
 - qual o contador;
 - quantas vezes pede ao contador para para se incrementar;

```
client(_, 0) ->
    true;
client(Counter, N) ->
    Counter ! increment,
    client(Counter, N - 1).
```



Exemplo: contador – outro cliente

- Função printVals pergunta o valor do contador, num ciclo infinito;
- Envia o seu Pid, para lhe poder ser enviada a resposta;

```
printVals(Counter) -> % infinite loop
  Counter ! {value, self()},
  receive
    Value ->
        io:format("Counter_value:~w~n", [Value])
  end,
  printVals(Counter).
```

- Não estamos a discriminar a mensagen recebida;
- Não funcionaria se houvesse a possibilida de serem recebidas mensagens diferentes, ou de diferentes processos;
- Deve ser usada mensagem estruturada.



Exemplo: contador - main

- Função principal do módulo:
 - cria um processo contador;
 - cria N processos clientes do contador;
 - executa a função printVals, entrando num ciclo infinito;

```
-module(counter).
-export([start/2]).

start(N, Iters) ->
   Counter = spawn(fun() -> counter(0) end),
   createClients(Counter, Iters, N),
   printVals(Counter).

createClients(_, _, 0) ->
   true;
createClients(Counter, Iters, N) ->
   spawn(fun() -> client(Counter, Iters) end),
   createClients(Counter, Iters, N - 1).
```



Exemplo: contador – nova versão com funções de interface

- Na versão anterior, os "clientes" do contador sabem, e usam, um protocolo envolvendo troca de mensagens;
- É útil encapsular o protocolo, escondendo-o dos clientes, e disponibilizar funções de interface para aceder ao contador;
- Tal permite mudar o protocolo sem reescrever código cliente;
- Vamos disponibilizar 4 funções:
 - criar contador: cria processo e devolve Pid;
 - incrementar contador leva Pid do processo contador;
 - obter valor do contador:
 - eliminar o contador, parando o processo;
- Vamos aproveitar para alterar o protocolo, tornando mais robusto o envio do valor do contador a quem faz o pedido;
- Vamos colocar cliente e contador em módulos separados;



Exemplo: contador – funções de interface

As funções de interface ficam:

```
-module (counter) .
-export([create/0, increment/1, value/1, stop/1]).
create() ->
  spawn(fun() -> counter(0) end).
increment (Counter) ->
  Counter ! increment.
value(Counter) ->
  Counter ! {value, self()},
  receive
    {Counter, Value} ->
      Value
  end.
stop(Counter) ->
  Counter ! stop.
```



Exemplo: contador – função do processo contador

A nova versão da função counter:

```
counter(Val) ->
  receive
  increment ->
    counter(Val + 1);
  {value, From} ->
    From ! {self(), Val},
    counter(Val);
  stop ->
    true
  end.
```

- Na resposta a value() é enviado o Pid; mais robustez;
- É acrescentada mensagem para parar processo;



Exemplo: contador – clientes

- Os clientes ficam agora mais simples;
- Protocolo é escondido;

```
start(N. Iters) ->
  Counter = counter:create(),
  createClients(Counter, Iters, N),
  printVals(Counter).
createClients(_, _, 0) -> true;
createClients(Counter, Iters, N) ->
  spawn(fun() -> client(Counter, Iters) end),
  createClients(Counter, Iters, N - 1).
client(, 0) -> true;
client (Counter, N) ->
  counter:increment (Counter),
  client (Counter, N - 1).
printVals(Counter) -> % infinite loop
  V = counter: value (Counter).
  io:format("Counter value:~w~n", [V]),
  printVals(Counter).
```



Exemplo: lock de leitura/escrita

- Suponhamos duas classes de processos:
 - readers: querem fazer operações de leitura sobre um recurso;
 - writers: querem fazer operações de escrita sobre um recurso;
- Um bloco de operações de leitura ou escrita é rodeado de código de sincronização; assim existem 4 operações:
 - acquireRead e releaseRead para rodear bloco de leitura;
 - acquireWrite e releaseWrite para rodear bloco de escrita.
- Ou melhor, duas operações:
 - acquire, em modo de read ou write
 - release; n\u00e3o faz sentido poder especificar o modo
- Requisitos de segurança:
 - podem estar vários processos a ler;
 - se um processo estiver a escrever, mais nenhum pode estar a ler ou escrever.
- Problema: implementar as operações de sincronização.



Leitores escritores em Erlang

Funções de interface:

```
create() ->
    spawn(fun released/0).

acquire(Pid, Mode) when Mode == read; Mode == write ->
    Pid ! {Mode, self()},
    receive acquired -> true end.

release(Pid) ->
    Pid ! {release, self()}.

destroy(Lock) ->
    Lock ! stop.
```



R/W Lock em Erlang – starvation de writers

```
released() ->
  receive
    {read, Pid} -> Pid ! acquired, reading([Pid]);
    {write, Pid} -> Pid ! acquired, writing(Pid)
  end.
reading([]) -> released();
reading(Readers) ->
  receive
    {read, Pid} -> Pid ! acquired, reading([Pid | Readers]);
    {release, Pid} -> reading(Readers -- [Pid])
  end.
writing(Pid) ->
  receive
    {release, Pid} -> released()
  end.
```



R/W Lock em Erlang – sem starvation

```
released() ->
  receive
    {read, Pid} -> Pid ! acquired, reading([Pid]);
    {write, Pid} -> Pid ! acquired, writing(Pid)
  end.
reading([]) -> released();
reading (Readers) ->
  receive
    {read, Pid} -> Pid ! acquired, reading([Pid | Readers]);
    {release, Pid} -> reading(Readers -- [Pid]);
    {write, Pid} -> reading(Readers, Pid)
  end.
reading([], Writer) -> Writer ! acquired, writing(Writer);
reading(Readers, Writer) ->
  receive
    {release, Pid} -> reading(Readers -- [Pid], Writer)
  end.
writing(Pid) ->
  receive
    {release, Pid} -> released()
  end.
```

