SCALA + AKKA

programação multithread de maneira fácil

Introdução ao AKKA, framework que torna a programação multithread simples e fácil.

Hoje em dia, visto que processadores ganham cada vez mais núcleos, é de extrema importância que, além de criar sistemas multithread eficientes para nos beneficiarmos de todo o poder dos processadores modernos e, desse modo, criar sistemas com maior capacidade de processamento, também criemos sistema de fácil manutenção e com o menor número possível de erros. Vamos mostrar como escrever um programa multithread de análise de dados de forma simples, porém robusta.

Sabemos que criar programas multithread não é uma tarefa simples. Pelo contrário, a complexidade de se fazer esse tipo de sistema é enorme e é capaz de tornar o dia-a-dia de desenvolvedores em uma batalha épica, tanto para conseguir pensar em como os threads interagem entre si e evitar memory leaks e deadlocks e outros problemas que possam aparecer, como para, quando necessário, debugar e corrigir problemas.

Akka, segundo os seus criadores, é um framework para facilitar essa tarefa e assim diminuir o número de erros que surgem em consequência de implementações erradas, sejam elas devido à complexidade da tarefa ou por qualquer outro motivo.

A grande força do framework vem do fato que, além de facilitar a programação multithread e permitir construir sistemas que escalem tanto horizontal como verticalmente. Pois podemos criar mais atores (ver caixa: Entendendo atores) dentro de um mesmo computador, utilizando dessa maneira toda capacidade do processador e de que podemos, caso seja necessário, criar um cluster de atores utilizando vários computadores (ver figura 1) como do fato de que podemos utilizá-lo dentro de vários cenários diferentes (ver caixa: Diferentes cenários para o uso do Akka) há também a possibilidade de ser utilizado em projetos Java (Ver caixa: Java + Akka).

Neste artigo, irei mostrar com utilizar o framework para criar um sistema multithread em Scala, porém, não faz parte do escopo mostrar funcionalidades mais avançadas, como, por exemplo, a clusterização de atores. Para mais informações, o site do framework possui uma documentação completa e de fácil entendimento.

Entendendo atores

Para facilitar a compreensão do que são atores, podemos imaginá-los como threads, isto é, cada ator seria uma thread. Diferentemente do que, normalmente, fazemos em Java, os atores não compartilham estado entre si. Eles se comunicam, entre si, através de mensagens as quais, normalmente são classes case. Portanto, quando trabalhamos com atores, nós não utilizamos o modelo de estado compartilhado (shared-state), das linguagens imperativas tradicionais.

Implementando o sistema

Iremos construir um sistema que analisa notas dos alunos de uma escola e nos dá as seguintes informações:

1. Lista de alunos aprovados
2. Lista de alunos reprovados
3. Maior média
4. Menor média

Esse exemplo servirá para mostrar como podemos otimizar a análise de dados, que é uma tarefa que consome muita memória, processamento e costuma ser demorada, criando um sistema paralelizado de maneira simples e fácil (ver caixa: Imagem não é tudo).

A arquitetura do sistema consiste, basicamente, em um objeto Analisador, que será o responsável por criar os atores, passar as tarefas a serem executas para os mesmos e consolidar o resultado e, em vários atores que farão o trabalho duro (analisar os dados) (ver figura 2).

Gilberto T. Garcia Jr. | ggarcia@eureka.inf.br

Formado em Filosofia pela USP, trabalha com Java e atualmente estuda linguagens funcionais. Há 12 anos no mercado de TI, já atuou em diversos segmentos do mercado, como telefonia, financeiro e seguros. Desde 2010 pesquisa linguagens funcionais para descobrir técnicas que gerem aumento de produtividade no desenvolvimento de sistemas, além de ser o criador do site Scalado (site voltado ao Scala e Lift).

Diferentes cenários para o uso do Akka

Você pode utilizar o framework como se fosse uma biblioteca, caso esteja escrevendo uma aplicação para a Web ou, se quiser, sua aplicação web poderia invocar os atores como se fossem serviços externos à aplicação. Ou ainda, poderíamos utilizá-lo como um microkernel stand-alone.

Java + Akka

A utilização do framework em Java é quase igual a utilização em Scala. A diferença é que, ao utilizar em projetos Scala, devemos importar Actor, ActorFactory e LoadBalancer, enquanto em projetos Java, devemos importar UntypedActor, UntypedActorFactory e UntypedLoadBalancer. Para mais detalhes, ver referência no final do artigo.

Imagem não é tudo

O leitor que não se engane, o exemplo pode ser simples, mas o conceito é muito poderoso. O Google criou um framework chamado MapReduce para suportar computação distribuída de enormes massas de dados. Esse framework é inspirado nas funções map e reduce, comumente encontradas em linguagens funcionais.

Bancos utilizam esse mesmo conceito para, por exemplo, fazer análise de riscos de seus clientes.

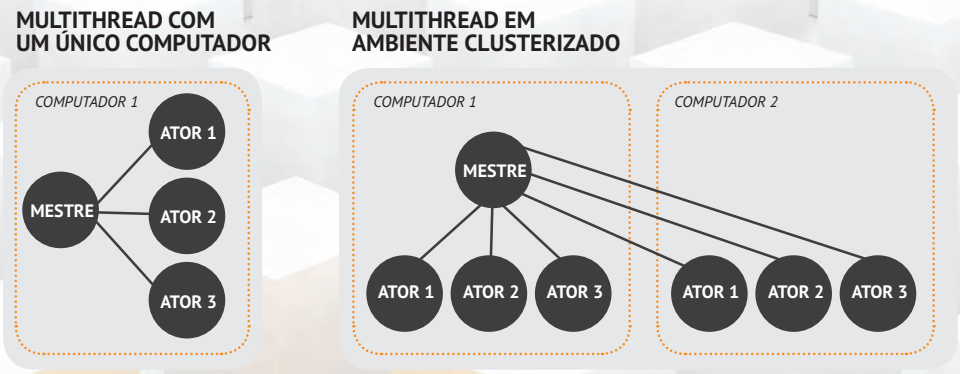

Figura 1. *Possíveis arquiteturas.*

Figura 2. *Arquitetura do sistema.*

Atores como roteadores

Um trabalhador pode atuar como roteador e gerar mais trabalhadores, criando, assim, uma verdadeira cadeia de processamento (ver figura 3).

O trabalhador

A classe Trabalhador define as nossas regras de negócio. Será ela a responsável por analisar os dados e devolver o resultado para o analisador. Trabalhador estende do trait Actor do framework. O trait Actor define o método abstrato receive, o qual devemos implementar em nosso ator (ver Listagem 1 – Implementação da classe trabalhador). Esse método é o responsável por receber as mensagens do analisador, processá-las e devolver o resultado, ou seja, ele é o ponto de entrada de mensagens para os atores.

Durante a transmissão da mensagem, o framework passa uma referência implícita do ator mestre (chamada de self) para os trabalhadores, para que possam utilizá-lo para dar a resposta do processamento ou repassá-lo para outros atores, caso exista uma cadeia de trabalhadores (ver caixa: Atores como roteadores).

É por causa dessa referência implícita, que podemos chamar o método reply e assim passar o resultado da análise para o ator mestre (analisador).

O método analise utiliza a classe helper Dados (ver caixa: Obtendo dados) para obter a lista de alunos que deverá ser analisada (ver Listagem 1 – Implementação da classe trabalhador).

Obtendo dados

A classe dados gera uma sequência de alunos. Poderia ser uma classe que obtivesse os dados de um banco, ou de um XML, ou de qualquer outra forma. Nós a criamos desta maneira para simplificar o exemplo e poder focar na construção do sistema.

Ele utiliza os métodos drop e take do objeto Seq para eliminar os alunos que estão fora do range (início, fim), ficando assim só com os alunos que interessam.

Ele chama, então, o método partition e passa uma função '() => Boolean' que servirá para separar os alunos aprovados dos reprovados. O método partition pega uma lista e quebra em duas. A primeira lista conterá os elementos da lista original que fizerem com que a função passada como parâmetro retorne true e os elementos que fizerem a função retornar false serão alocados na segunda lista.

Após separar os alunos aprovados dos reprovados, precisamos achar a maior e a menor média. Para isso, chamamos o métodos acharMedia (ver Listagem 1 – Implementação da classe trabalhador).

Este método recebe uma lista de alunos, um valor base para ser usado no cálculo e uma função. Para achar a maior média utilizamos como valor base 0.0, pois estamos interessados no maior valor e, por isso, precisamos passar o menor valor possível. E como função de comparação, passamos '(i,m) => i.max(m)', ou seja, uma função com dois parâmetros que retorna o maior deles. Analogamente, para calcular a menor média, precisamos passar a maior média possível como valor base e como função de comparação precisamos passar uma que dado dois valores ela nos retorne o menor e por isso passamos '(i,m) => i.min(m)'

Por fim, criamos o objeto Resultado que será retornado para o analisador (ver Listagem 1 – Implementação da classe trabalhador). Com isso, encerramos a implementação da classe Trabalhador. Podemos ir além e ver com mais detalhes a classe Analisador.

Listagem 1. Implementação da classe Trabalhador.

```
class Trabalhador extends Actor {
```

```
class Trabalhador extends Actor {

def receive = {
    case Trabalho(inicio, numElementos) =>
        self reply analise(inicio, numElementos)
}

    private def analise(inicio: Int, numElementos: Int):
    Resultado = {
        val lista = Dados.carregaDados.drop(inicio).
            take(numElementos)
        val (aprovados, reprovados) = lista.partition(_.media
            >= 5)

        val maiorMedia = acharMedia(lista, 0.0, (i, m) =>
            m.max(i))
        val menorMedia = acharMedia(lista, 10.0, (i, m) =>
            m.min(i))

        new Resultado(maiorMedia, menorMedia,
            aprovados, reprovados)
    }

    private def acharMedia(alunos: Seq[Aluno], base:
    Double, func: (Double, Double) => Double): Double =
    alunos.map(_.media).foldLeft(base)(func)
}
```

O analisador

O analisador é mais uma classe que estende do trait Actor e por isso deve implementar o método receive (ver Listagem 2: Implementação da classe Analisador). Porém, antes de definir o método receive, nós precisamos preparar o que for necessário para a construção do objeto Analisador.

Para isso, criamos um vetor de Trabalhadores utilizando o método fill, o qual recebe dois parâmetros, o primeiro é um inteiro que informa quantas posições deverão ser criadas e o segundo recebe o objeto que será alocado em cada posição, nesse caso utilizamos a factory actorOf (ver caixa: Tipos de factories). Esse método cria um ator do tipo especificado. No nosso caso, atores do tipo Trabalhador.

Devemos notar que além de fazer a chamada à factory para criar o ator, nós também invocamos o método start, o qual deixa o ator recém-criado pronto para receber mensagens (ver caixa: Ciclo de vida dos atores).

O próximo passo é a criação do roteador, o qual será responsável por balancear o trabalho. Para isso utilizamos o método loadBalanceActor do objeto Routing que recebe como parâmetro um InfiniteIterator. Iremos utilizar, como InfiniteIterator, o CyclicIterator que, como o próprio nome diz, fará o balanceamento do trabalho de forma cíclica, como na algoritmo round-robin.

Utilizaremos uma variável para controlar o tempo de execução e outra para saber se todo o trabalho foi efetuado ou não. São elas, inicio e numResultado, respectivamente.

Tipos de factories

O framework Akka possui duas versões da factory, actorOf, para criar atores. A primeira versão utiliza um tipo de ator (actorOf[MeuTipo]) e é utilizada quando o construtor do ator não possui parâmetros e a segunda recebe uma instância de um ator (actorOf(new MeuAtor(...)) e deve ser utilizada quando o construtor do ator possui parâmetros.

Ciclo de vida dos atores

Os atores possuem três estados.
1. Criado
2. Iniciado
3. Parado

Um ator só pode receber mensagens quando está no estado iniciado. Nos outros dois estados, o ator não pode receber mensagens.

O ator está no estado 'criado' logo após a sua criação através de uma das duas versões da factory actorOf (ver caixa: Tipos de factories).

O ator passa ao estado iniciado após a chamada ao método start e parado quando chamamos o método stop.

Uma vez parado, um ator não pode mais receber mensagens e não pode voltar ao estado iniciado.

Mensagens

Mensagens são a forma como os atores se comunicam entre si. É uma forma poderosa de comunicação, pois evita o compartilhamento de estado entre os atores, o que levaria as complicações clássicas de uma implementação multithread.

Nosso sistema utiliza três tipos de mensagens, Analise, Trabalho e Resultado. Todas elas utilizam o trait Mensagem que é definido como sealed para evitar que mensagens sejam criadas de forma espalhada pela estrutura do sistema.

Com tudo preparado, podemos criar o método receive do analisador, que está capacitado a lidar com dois tipos de mensagens (ver caixa: Mensagens), Analise e Resultado. Analise é a mensagem que será passada pelo método main da aplicação e será responsável por iniciar todo o processo. E a mensagem Resultado é a responsável por disparar o processo de consolidação dos dados.

Vamos primeiro olhar para o caso da mensagem Analise. Criamos um for que passará, para o roteador, o número de mensagens definidas quando criarmos o objeto Analisador. Para cada mensagem, o for cria um trabalhador e passa como parâmetros o número do elemento do início e o número de elementos que o trabalhador deverá processar. Após passar todas as mensagens, devemos passar duas mensagens PoisonPill para o roteador. A primeira é para que o roteador desligue todos os atores e por isso a passamos utilizando a mensagem de Broadcast. Ao utilizar a mensagem de Broadcast, o roteador irá pegar a mensagem que foi passada como parâmetro e irá retransmiti-la para todos os atores, fazendo com que todos eles sejam desligados. A outra PoisonPill é para que o próprio roteador se desligue.

Para cada mensagem de Resultado que o Analisador recebe, nós chamamos o método consolidar do objeto Consolidado, veremos como ele funciona mais a frente. Após invocar o método consolidar, incrementamos a variável numResultados e verificamos se é igual ao número de mensagens. Se for, ou seja, se processamos todas as mensagens, invocamos o método stop do próprio analisador, utilizando a variável de referência self e, dessa forma, encerramos o processo de análise.

Antes de passar para o objeto Consolidado, temos, ainda, dois métodos para analisar, o primeiro é o preStart e o segundo é o postStop. Esses métodos estão intimamente relacionados com o ciclo de vida dos atores e como os nomes deixam claro, o primeiro método serve para executar qualquer coisa que seja necessária antes da inicialização do ator e o postStop é utilizado quando queremos executar coisas após o desligamento do ator.

Com essas características eles se tornam o melhor lugar para iniciar a nossa variável início e depois imprimi-la no final, e assim saber exatamente o tempo que demorou para efetuar a análise.

Listagem 2. Implementação da classe Analisador.

```
class Analisador(numTrabalhadores: Int, numMensagens:
Int, numElementos: Int, latch: CountDownLatch) extends
Actor {

  val trabalhadores = Vector.fill(numTrabalhadores)(
    actorOf[Trabalhador].start())

  val roteador = Routing.loadBalancerActor(
    CyclicIterator(trabalhadores)).start()
var numResultado: Int = _
  var inicio: Long = _
  val consolidado = new Consolidado
  def receive = {
  case Analise =>
    for (i <- 0 until numMensagens) roteador !
      Trabalho(i * numElementos, numElementos)
roteador ! Broadcast(PoisonPill
    roteador ! PoisonPill
 case Resultado(maior, menor, aprov, reprov) =>
    consolidado.consolidar(maior, menor, aprov, reprov)
    numResultado += 1
    if (numResultado == numMensagens) self.stop()
  }
override def preStart() {
    inicio = System.currentTimeMillis
  }
```

```
    override def postStop() {
      consolidado.imprimir()
      println("Tempo estimado de processamento:
  \t\t%s millis".format((System.currentTimeMillis -
        inicio)))
      latch.countDown()
    }
}
```

Consolidando o relatório

Notem que, quando chamamos o método consolidar durante o processamento da mensagem de resultado, passamos quatro parâmetros. O primeiro é a maior média, o segundo a menor média e o terceiro e quarto são a lista de aprovados e a lista de reprovados, respectivamente.

O método consolidar (ver Listagem 3: Implementação da classe Consolidado) testa a variável maior para ver se o valor é maior que o valor da variável maiorMedia, se for, ele armazena o novo valor na variável maiorMedia, caso contrário ele mantém o valor atual. O mesmo ocorre com a variável menor só que, ao invés de testar o maior valor, ele testa para saber quem é o menor. Nas outras duas linhas do método, nós apenas pegamos as listas e a concatenemos com as listas que estão no objeto.

Com isso, ao processar todas as mensagens de Resultado, no final, teremos a maior e menor média e as listas de todos os alunos aprovados e reprovados só nos restando imprimir os dados.

O método imprimir basicamente em chamadas ao método println. Porém, vale notar que para imprimir cada aluno da lista, nós utilizamos o método foreach para invocar o método println para cada objeto da lista. Como utilizamos _ para passar o objeto aluno para o método println, este invocará o método toString na classe Aluno, da mesma maneira que aconteceria em um programa Java.

Listagem 3. Implementação da classe Consolidado.

```
class Consolidado {
   var maiorMedia = 0.0
   var menorMedia = 10.0
   var aprovados: Seq[Aluno] = Nil
   var reprovados: Seq[Aluno] = Nil
   def consolidar(maior: Double, menor: Double, aprov:
      Seq[Aluno], reprov: Seq[Aluno]) {

  maiorMedia = maiorMedia max maior
   menorMedia = menorMedia min menor
   aprovados = aprovados ++ aprov
   reprovados = reprovados ++ reprov
   }
 def imprimir() {
   .....
   aprovados.foreach(println _)
   ....
   reprovados.foreach(println _)
   }
}
```

Ilustrando as diferenças do modelo tradicional

Após a explicação de como implementar uma aplicação multithread com Scala e Akka, vamos mostrar as diferenças para o modelo tradicional (shared-state).

As principais diferenças são notadas nas classes Analisador e Trabalhador. Enquanto na versão em Scala nós não transmitimos nenhuma referência da classe Consolidado para a classe trabalhador, na versão em Java, no modelo tradicional, isto se faz necessário, pois como não temos as passagens de mensagens, precisamos criar uma maneira de ter acesso ao estado da aplicação (ver Listagem 4: Implementação da classe Analisador em Java).

Listagem 4. Implementação da classe Analisador em Java.

```
public class Analisador implements Runnable {
   private int numTrabalhadores;
   private int numMensagens;
   private int numElementos;
   private Consolidado consolidado = new Consolidado();

   @Override
   public void run() {
      List<Thread> threads = new ArrayList<Thread>();
      int rodando;

      for (int i = 0; i < numTrabalhadores; i++) {
         Trabalhador trabalhador = new Trabalhador();
for (int j = 0; j < numMensagens; j++) {
            trabalhador.setInicio(i * numElementos);
            trabalhador.setNumElementos(numElementos);
            trabalhador.setConsolidado(consolidado);
         }
         Thread thread = new Thread(trabalhador);
         thread.start();
  threads.add(thread);
      }
 do {
         rodando = 0;
for (Thread thread : threads) {
            if (thread.isAlive()) {
               rodando++;
            }
         }

               } while (rodando > 0);
      consolidado.imprimir();
   }
```

Figura 3. *Atores como roteadores.*

```
    /* getters and setters */
}
```

Do mesmo modo, na versão em Scala, a classe trabalhado não tinha nenhuma referência a classe Consolidado, e agora, na versão em Java tem (ver Listagem 5: Implementação da classe Trabalhador em Java). Isto porque, ela precisa atualizar a classe com os dados processados de modo que, no final do processamento, a classe Analisador possa imprimir o resultado.

Listagem 5. Implementação da classe Trabalhador em Java.

```
public class Trabalhador implements Runnable {
    private int inicio;
    private int numElementos;
    private List<Aluno> alunos = Dados.carregaDados();
    private Consolidado consolidado;

    @Override
    public void run() {
        int counter = 0;
        List<Aluno> aprovados = new ArrayList<Aluno>();
        List<Aluno> reprovados = new ArrayList<Aluno>();
        Double maiorMedia = 0d;
        Double menorMedia = 10d;
        for (Aluno aluno : alunos) {
            if (counter >= inicio && counter <= (inicio +
                numElementos)) {
                Double media = aluno.calcularMedia();

                if (media >= 5) {
                    aprovados.add(aluno);
                } else {
                    reprovados.add(aluno);
                }
                            if (media > maiorMedia) {
                    maiorMedia = media;
                }
                            if (media < menorMedia) {
                    menorMedia = media;
                }
            }
        }
        consolidado.consolidar(maiorMedia, menorMedia,
                aprovados, reprovados);
    }
    /* getters and setters */
}
```

Considerações finais

Tentei mostrar, através de um exemplo simples, como é fácil implementar um sistema multithread utilizando o framework Akka. Como o leitor pôde perceber, não nos preocupamos com os detalhes corriqueiros de uma implementação multithread como notificações, deadlocks e outras complexidades que, frequentemente, os desenvolvedores são obrigados a lidar, muito pelo contrário, ao utilizar o framework, nós pudemos nos concentrar exclusivamente nas regras de negócio, tornando a tarefa de desenvolver um sistema multithread mais produtiva e menos suscetível a erros.

Do mesmo modo, para efeito de simplicidade, não nos preocupamos com notify, wait, synchronized e qualquer outro problema que poderíamos ter na implementação multithread tradicional. Porém, devemos ficar atentos ao fato de que muitos problemas podem e certamente ocorrerão devido ao compartilhamento do objeto Consolidado entre as diversas threads.

Poderíamos explorar outras funcionalidades do framework para criar sistemas maiores e complexos, como, por exemplo, o agendador de tarefas.

/referências

> *Scalado: http://scalado.com.br/*

> *Site oficial do framework: http://akka.io/*

> *Atores: http://www.scala-lang.org/node/242*

> *Java + Akka: http://akka.io/docs/akka/1.3-RC1/intro/getting-started-first-java.html*

> *Drop: http://www.scala-lang.org/api/2.7.6/scala/Seq.html#drop%28Int%29*

> *Take: http://www.scala-lang.org/api/2.7.6/scala/Seq.html#take%28Int%29*

> *Partition: http://www.scala-lang.org/api/2.7.6/scala/Iterable.html#partition%28%28A%29%3D%3EBoolean%29*

> *Round-robin: http://pt.wikipedia.org/wiki/Round-robin_%28algoritmo%29*

> *Classes seladas (sealed classes): http://www.scala-lang.org/node/123*