MAC 5742 - Computação Paralela e Distribuída
Processamento Concorrente com *Promises*

Prof. Alfredo Goldman, Rogério A. Gonçalves,
Daniel Cordeiro e Marcos Amarís

Mateus Espadoto
mespadoto@yahoo.com
9395501

20 de Junho de 2015

1 Introdução

O objetivo deste trabalho é apresentar o conceito de *promises* como técnica de programação concorrente, bem como exemplos de seu uso em linguagens de programação atuais.

Computadores com múltiplos processadores (SMP, ou *Symmetric Multiprocessing*), que possibilitam a computação concorrente, existem desde os anos 1960, e durante muitos anos ficaram restritos às grandes instituições que podiam arcar com o seu alto custo. Nos anos 2000, com o surgimento dos processadores *multicore* de baixo custo, a computação concorrente ficou acessível a todos, e portanto houve a necessidade de se repensar as técnicas de programação concorrente existentes, de modo a tornar mais simples o seu uso.

Técnicas como programação concorrente com *threads* e subprocessos vêm sendo utilizadas há anos com sucesso, mas por se tratar de um paradigma de baixo nível, onde é preciso definir explicitamente a lógica de controle e divisão de tarefas, é sujeita a erros como condições de corrida e *deadlocks*.

Há diversas iniciativas de se simplificar a programação concorrente, como os *frameworks* OpenMP (OPENMP, 2015) e MPI (OPENMPI, 2015), que abstraem parte da lógica de controle das tarefas, deixando o programador livre para concentrar-se no algoritmo em questão. Mesmo com o benefício evidente, estes *frameworks* introduzem dificuldades próprias, como por exemplo, o uso de uma sintaxe peculiar que nem sempre é familiar ou totalmente integrada à linguagem em questão.

As *promises* surgem nesse contexto como alternativa a estes *frameworks*, proporcionando comandos e declarações mais integradas às linguagens alvo, de modo a facilitar a vida do programador. Não se trata de uma ideia nova: foi proposta originalmente em (FRIEDMAN; WISE, 1976), mas ganhou força no início dos anos 2000, coincidentemente com a disseminação dos processadores *multicore*.

2 Definição

Como definição geral de *promise* pode-se dizer que se trata de uma abstração para programação concorrente que permite a realização de chamadas assíncronas de funções, composição de chamadas e controle de exceções.

Não há padronização quanto à terminologia utilizada pelas implementações nas diferentes linguagens de programação. O padrão Promises/A+ (PROMISES/A+, 2015) usa os seguintes termos:

- *Promise*: uma referência para um valor que ainda não é conhecido, do resultado de uma computação;
- *Deferred*: uma computação não concluída, que cumprirá a promessa com um valor ou uma exceção.

De acordo com essa definição, pode-se dizer que *promise* é uma referência que armazenará o resultado da execução de um *deferred*, que é uma computação assíncrona ainda não concluída.

Outras linguagens de programação utilizam nomes diferentes para as mesmas funcionalidades em suas implementações, causando certa confusão para o leitor:

- Javascript: Promise e Deferred;
- Java: FutureTask (Runnable) e Callable;
- Scala: Future e Promise;
- C++11: Future e Promise;
- Python: Future e callable (qualquer função/método);
- Microsoft: Task e Action.

Algumas linguagens, como Java por exemplo, possuem tipos diferentes de *promises*, como *FutureTask* e *CompletableFuture*, o que torna a confusão ainda maior. No entanto, apesar da inconsistência de terminologia, as implementações mencionadas possuem funcionalidade equivalente entre elas.

3 *Promises* e seus Estados

Promises podem ter os seguintes estados:

- Pendente: em execução;
- Resolvida: concluída com sucesso;
- Rejeitada: concluída com falha.

Quando um *deferred* conclui a sua execução, a *promise* é cumprida, seja ela resolvida, quando obrigatoriamente deve conter o valor final do *deferred*, ou rejeitada, quando obrigatoriamente deve conter o motivo da falha.

A composição de tarefas se utiliza destes estados para identificar quando uma *promise* foi cumprida e a próxima tarefa da cadeia pode ser executada, e qual o valor a ser passado, dependendo do estado de resolvida ou rejeitada. O valor final da *promise* precedente é sempre passado para a *promise* subsequente.

Para os casos de rejeição, é possível definir lógica de tratamento de erro, que pode ser executada localmente, diretamente associada à rejeição de uma *promise* específica, ou globalmente, associada à rejeição de qualquer *promise* da cadeia.

4 Casos de Uso

Promises podem ser vistas como uma forma implícita de criação de *threads*, com código mais enxuto. Chamadas assíncronas e barreiras, que são idiomas comuns da programação com *threads*, podem ser realizados com *promises* de forma mais simples e clara.

Uma vantagem das *promises* em relação às *threads* pode ser encontrada na possibilidade de composição de tarefas fornecida pela primeira: em casos onde é necessário realizar chamadas a funções de bibliotecas definidas externamente, a capacidade de compor cadeias de chamadas e realizar o tratamento de exceções externamente torna o código mais claro e direto.

Particularmente no caso de programação em Javascript do lado do cliente, onde boa parte da codificação é realizada de forma assíncrona devido à necessidade de não bloquear a *thread* principal para não impedir o usuário de realizar outras tarefas, é muito comum o uso de *callbacks*, que são funções registradas para serem executadas ao término de uma chamada de função assíncrona. O uso excessivo de *callbacks* recebe o nome

de *"callback-hell"*, por ser de difícil compreensão e manutenção. Acredita-se que as *promises* contribuem positivamente neste caso, tornando o código mais legível e simples de manter.

Em casos de divisão de trabalho, como por exemplo, um laço *for* paralelo, em que cada *thread* executa paralelamente uma iteração do laço, o uso de *promises* não apresenta grandes vantagens, dado que o particionamento dos dados deve ser feito manualmente, como na programação com *threads*. Acredita-se que *frameworks* como OpenMP apresentam alternativas mais apropriadas para esses casos, bem como linguagens desenvolvidas mais recentemente, como Julia (JULIA, 2015) e Chapel (CRAY, 2015), que possuem funcionalidades de programação concorrente implementadas nativamente.

5 Exemplos

Nas seções abaixo serão apresentados exemplos de uso de *promises* em Javascript e C#. Foram escolhidas duas linguagens com o objetivo de permitir comparação entre as diferentes implementações. A biblioteca utilizada para os exemplos em Javascript foi a Q.js, a mesma utilizada em (RATHBUN, 2015), e para os exemplos em C# foram utilizadas as classes do *namespace* System.Threading.Tasks (MICROSOFT, 2015). Os exemplos em C# são mais extensos devido às particularidades da linguagem, que usa um sistema de tipos estático e que necessita que um programa possua uma estrutura mínima, com uma classe e um método *Main* como ponto de entrada, de forma similar ao que se tem em Java.

5.1 Chamada assíncrona

Uma chamada assíncrona, não-bloqueante, que retorna uma *promise* para que seja possível obter o valor final após o término da execução. Exemplo em Javascript:

```
var A = function() {...}

var promise = Q.fcall(A);
```

Exemplo em C#, lembrando que a Microsoft utiliza o termo *Task* no lugar de *promise*:

```
using System;
using System.Threading.Tasks;

namespace Promises
{
    class Program
    {
        static int A(int val)
        {
            ...
            return result;
        }

        static void Main(string[] args)
        {
            Task<int> promise = Task.Run<int>(() => A(1));
            ...

            //chamada bloqueante, obtem valor final
            Console.WriteLine(promise.Result);
        }
    }
}
```

O valor final da execução é obtido por meio da propriedade *Result* do objeto *promise*.

5.2 Encadeamento de chamadas

Em Javascript, o método *then* fornece um meio de encadear chamadas assíncronas, com a seguinte estrutura:

```
promise.then(onFulfilled1, onRejected1).then(onFullfilled2, onRejected2);
```

Onde "onFullfilled" indica o nome da função ou método a ser chamado caso a *promise* seja cumprida com sucesso, e "onRejected" indica o nome da função ou método a ser chamado caso a *promise* seja rejeitada, o que pode ser visto como uma forma de tratamento de exceção *local*, que pode, entre outras coisas, realizar uma nova tentativa de execução da *promise* precedente. O valor final da *promise* precedente será passado à função ou método correspondente, dependendo do seu estado.

No exemplo a seguir, em Javascript, são encadeadas as chamadas às funções A, B e C, e são mapeadas as funções de tratamento de erro falhaA e falhaB. Além do exemplo de encadeamento de chamadas, pode-se ver um exemplo de tratamento de erro *local*, onde o eventual erro gerado por uma *promise* precedente é tratado em uma *promise* subsequente:

```
var A = function() {...}
var B = function() {...}
var C = function() {...}
var falhaA = function() {...}
var falhaB = function() {...}

var promise = Q.fcall(A).then(B, falhaA).then(C, falhaB);
```

A seguir um exemplo de encadeamento de tarefas em C#:

```
using System;
using System.Threading.Tasks;

namespace Promises
{
    class Program
    {
        static int A(int val)
        {
            ...
            return result;
        }

        static int B(int val)
        {
            ...
            return result;
        }

        static void Main(string[] args)
        {
            Task<int> promise1 = Task.Run<int>(() => A(1));
            Task<int> promise2 = promise1.ContinueWith<int>((antecedent) => B(antecedent.Result));
            ...

            //chamada bloqueante, obtem valor final
            Console.WriteLine(promise2.Result);
```

```
        }
    }
}
```

O método *ContinueWith* em C# tem papel idêntico ao método *then* em Javascript, fazendo o encadeamento das chamadas às funções *A* e *B*. O tratamento de exceções com *promises* em C# é realizado por meio de um bloco *try/catch*. Na seção seguinte será exemplificada esta opção.

5.3 Tratamento de Erro Global

No exemplo a seguir, em Javascript, é apresentado o método *catch*, que serve para registrar uma função de tratamento de exceção global para toda a cadeia de chamadas:

```
var A = function() {...}
var B = function() {...}
var C = function() {...}
var D = function() {...}
errorHandler = function() {...}

var promise = Q.fcall(A).then(B).then(C).then(D).catch(errorHandler);
```

Em C# tem-se o mesmo efeito com o uso de um bloco *try/catch*:

```
using System;
using System.Threading.Tasks;

namespace Promises
{
    class Program
    {
        static int A(int val)
        {
            ...
            return result;
        }

        static int B(int val)
        {
            ...
            return result;
        }

        static void Main(string[] args)
        {
            Task<int> promise1 = Task.Run<int>(() => A(1));
            Task<int> promise2 = promise1.ContinueWith<int>((antecedent) => B(antecedent.Result));

            try
            {
                //chamada bloqueante, obtem valor final
                Console.WriteLine(promise2.Result);
            }
            catch (AggregateException e)
            {
                //obtem excecoes, em caso de erro
```

```
            Console.WriteLine(e.Message);
        }
    }
  }
}
```

5.4 Barreira *all*

O método *all* em Javascript cria uma *promise* que é cumprida quando todas as suas sub-*promises* forem cumpridas. Na prática funciona como uma barreira, que aguarda o término da execução de todas as chamadas, e possui a seguinte estrutura:

```
all([f1, f2, ..., fn])
```

Onde "f1", "f2", etc, indicam os nomes das *promises* cujos resultados serão aguardados pelo método *all*.

No exemplo a seguir, em Javascript, são chamadas as funções A, B e C de forma concorrente e assíncrona, e a *promise* criada pelo método *all* bloqueará a chamada até que todas as funções terminem sua execução:

```
var A = function() {...}
var B = function() {...}
var C = function() {...}

var promise = Q.all([A, B, C]);
```

A seguir um exemplo de barreira do tipo *all* em C#, implementada pelo método *WaitAll*:

```
using System;
using System.Threading.Tasks;

namespace Promises
{
    class Program
    {
        static int A(int val)
        {
            ...
            return result;
        }

        static int B(int val)
        {
            ...
            return result;
        }

        static void Main(string[] args)
        {
            Task[] promises = new Task[2];
            promises[0] = Task.Run<int>(() => A(1));
            promises[1] = Task.Run<int>(() => B(1));

            try
            {
                //barreira: aguarda termino de todas as execucoes
```

```
            Task.WaitAll(promises);

            foreach(Task<int> p in promises)
            {
                //obtem todos os valores finais
                Console.WriteLine(p.Result);
            }
        }
        catch (AggregateException e)
        {
            //obtem excecoes, em caso de erro
            Console.WriteLine(e.Message);
        }
    }
  }
}
```

5.5 Barreira *any*

O método *any* cria uma *promise* que é cumprida quando a primeira sub-*promise* for cumprida, com estrutura parecida à do método *all*:

```
all([f1, f2, ..., fn])
```

Onde "f1", "f2", etc, indicam os nomes das *promises* cujos resultados serão aguardados pelo método *any*.

No exemplo a seguir, em Javascript, são chamadas as funções A, B e C de forma concorrente e assíncrona, e a *promise* criada pelo método *any* bloqueará a chamada até que uma das funções termine sua execução:

```
var A = function() {...}
var B = function() {...}
var C = function() {...}

var promise = Q.any([A, B, C]);
```

A seguir um exemplo de barreira do tipo *any* em C#, implementada pelo método *WaitAny*:

```
using System;
using System.Threading.Tasks;

namespace Promises
{
    class Program
    {
        static int A(int val)
        {
            ...
            return result;
        }

        static int B(int val)
        {
            ...
            return result;
        }
```

```
        static void Main(string[] args)
        {
            Task[] promises = new Task[2];
            promises[0] = Task.Run<int>(() => A(1));
            promises[1] = Task.Run<int>(() => B(1));

            try
            {
                //barreira: aguarda o termino da primeira chamada
                int taskid = Task.WaitAny(promises);

                //obtem o id da promise que terminou primeiro
                Task<int> promise = (Task<int>)promises[taskid];

                //obtem valor final
                Console.WriteLine(promise.Result);

            }
            catch (AggregateException e)
            {
                //obtem excecoes, em caso de erro
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

6 Disponibilidade

As principais linguagens de programação da atualidade possuem implementações de *promises*. Abaixo são listados alguns exemplos com os nomes das respectivas bibliotecas:

- Javascript: Q.js, when.js, jQuery;
- Java: java.util.concurrent.Future (ORACLE, 2015), JDeferred (JDEFERRED, 2015);
- C#: System.Threading.Tasks.Task (MICROSOFT, 2015);
- C++11: std::promises e std::futures;
- Python: concurrent.futures (PYTHON, 2015), aplus (APLUS, 2015);
- Scala: scala.concurrent (SCALA, 2015).

A existência de implementações para todas estas linguagens indica a crescente popularidade das *promises* como recurso para programação concorrente.

7 Conclusão

Promise é uma abstração para programação concorrente que busca tornar simplificar a execução de chamadas assíncronas e a composição e tarefas. Pode ser utilizada em praticamente todos os contextos em que *threads* explícitas seriam utilizadas com benefícios, como código mais simples de manter e mais legível.

As principais linguagens de programação da atualidade dão suporte a *promises*, de forma similar em termos de funcionalidade.

No momento, percebe-se que um fator de risco para o seu uso é que, devido à falta de padronização, há certa confusão na terminologia utilizada pelas diferentes implementações, o que pode causar certa confusão.

Referências

APLUS. **aplus**. 2015. https://github.com/xogeny/aplus. Acesso em: 18 jun. 2015.

CRAY. **The Chapel Parallel Programming Language**. 2015. http://chapel.cray.com/. Acesso em: 18 jun. 2015.

FRIEDMAN, D. P.; WISE, D. S. **The impact of applicative programming on multiprocessing**. Indiana, USA: Indiana University, Computer Science Department, 1976.

JDEFERRED. **JDeferred**. 2015. http://jdeferred.org. Acesso em: 18 jun. 2015.

JULIA. **Julia Programming Language**. 2015. http://julialang.org/. Acesso em: 18 jun. 2015.

MICROSOFT. **Microsoft .NET Documentation**. 2015. https://msdn.microsoft.com/en-us/library/system.threading.tasks(v=vs.110).aspx. Acesso em: 18 jun. 2015.

OPENMP. **OpenMP**. 2015. http://openmp.org/wp/. Acesso em: 18 jun. 2015.

OPENMPI. **OpenMPI**. 2015. http://www.open-mpi.org/. Acesso em: 18 jun. 2015.

ORACLE. **Oracle Java SE 8 Documentation**. 2015. http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html. Acesso em: 18 jun. 2015.

PROMISES/A+. **Promises/A+**. 2015. http://promisesaplus.com. Acesso em: 18 jun. 2015.

PYTHON. **Python Documentation**. 2015. https://docs.python.org/3.4/library/concurrent.futures.html. Acesso em: 18 jun. 2015.

RATHBUN, S. Parallel processing with promises. **Commun. ACM**, ACM, New York, NY, USA, v. 58, n. 5, p. 42–47, abr. 2015. ISSN 0001-0782.

SCALA. **Scala Documentation**. 2015. http://docs.scala-lang.org/overviews/core/futures.html. Acesso em: 18 jun. 2015.