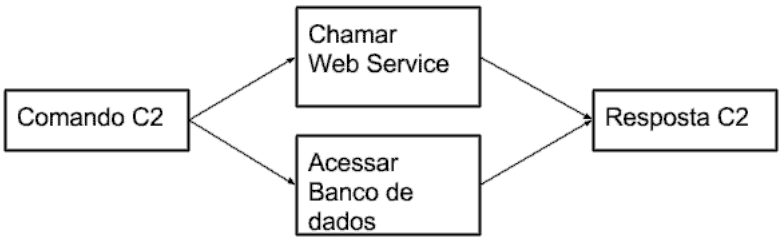
04

Retornos no Futuro

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/threads2/capitulo-6.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

Já falamos antes que o nosso `ComandoC2` fará algo diferente do que `ComandoC1`. Até agora ambos foram implementados de maneira igual. A nossa motivação é a seguinte: para executar o `ComandoC2` é preciso acessar dois recursos externos, um banco de dados e um serviço web. Ambos demoram e em alguns casos não terminam corretamente por causa de possíveis problemas na rede. É preciso lidar com esses problemas e o resultado de ambas as execuções deve ser combinado em um único resultado!



A nossa ideia é aproveitar as threads o máximo possível e executar cada acesso, banco e *web service*, em uma thread separada. Com isso não há problema nenhum, no entanto o resultado dessa execução deve ser combinado e devolvido ao cliente. Agora já temos um problema...

Tarefas com retorno, o Callable

Já vimos algumas formas de como threads podem compartilhar dados, por exemplo, usando objetos com acesso sincronizado, e poderíamos usar uma implementação baseada nessa ideia. Com a introdução do pacote `java.util.concurrent`, entraram outras possibilidades de resolver este problema, que vamos explorar nesse capítulo.

Até agora aprendemos que uma thread precisa de uma tarefa que deve implementar a interface `Runnable`. O problema da interface `Runnable` é que o método `run()` *não retorna nada*. Isso foi resolvido no Java 5 através de uma nova interface: `Callable`. Ao invés de criar uma tarefa com `Runnable`, podemos aproveitar a interface nova para definir a tarefa:

```
public class ComandoC2ChamaWS implements Callable<String> {

    @Override

    public String call() throws Exception {
    }
}
```

A interface `Callable` resolve dois problemas: a **execução pode retornar algo**, definido pelo parâmetro genérico `<String>`, e o método `call` joga uma `Exception` para **não precisar do** `try-catch`.

Callable precisa do pool

Diferente da `Runnable`, as tarefas com `Callable` precisam ser executadas pelo `ExecutorService`, ou seja, através do pool de threads, sempre. A classe `Thread` não pode receber um `Callable`, apenas `Runnable`. Sabendo disso, falta falar qual método do nosso pool pode receber um `Callable`. A resposta é `submit`:

```
ExecutorService threadPool = Executors.newCachedThreadPool(); //o nosso pool de threads
threadPool.submit(new ComandoC2AcessaBanco()); //usando submit que recebe um Callable
```

A pergunta ainda é: para onde vai o retorno do `Callable`? A resposta está no método `submit` do `ExecutorService`.

Submetendo Runnable e Callable

Aliás, talvez você já tenha percebido, quando passamos um `Runnable` usamos o método `execute` do `ExecutorService`

```
//usando CommandoC1 que implementa Runnable
threadPool.execute(new ComandoC1());
```

Mas quando usamos um `Callable` usamos o método `submit`:

```
//usando ComandoC2AcessaBanco que implementa Callable
threadPool.submit(new ComandoC2AcessaBanco());
```

Para ser correto, o método `submit` é sobrecarregado e **funciona com as duas interfaces**, `Runnable` e `Callable`:

```
//usando submit sempre
threadPool.submit(new ComandoC1());
threadPool.submit(new ComandoC2AcessaBanco());
```

A diferença do `submit`, comparado com `execute`, é que ele possui um retorno, **um `Future`**.

Entendendo o Future

O nome `Future` parece estranho, mas faz todo sentido se pensarmos que a tarefa passada para o pool *será executada em algum momento no futuro*. No entanto, o método `submit` não bloqueia a execução e podemos submeter quantas tarefas quisermos. Ou seja, o retorno pode ser algo que estará pronto no futuro, um `Future`:

```
Executor executor = Executors.newCachedThreadPool();
Future<String> future = executor.submit(new ComandoC2AcessaBanco(saida));
```

Esse `future` não é exatamente o que o `Callable` retorna e sim algo que terá o resultado desse `Callable`. Quando a tarefa terminar, o `Future` receberá o resultado. E como pegar esse resultado?

```
String resultadoDoCallable = future.get();
```

O método `get()` bloqueia a thread e espera até o resultado aparecer. Tem uma alternativa, podemos chamar o `get` e esperar por um tempo limitado. Isso é útil quando não podemos garantir que a execução realmente termine com

sucesso:

```
String resultadoDoCallable = future.get(10, TimeUnit.SECONDS); //esperando por 10s
```

Se não houver resultado em 10s, o `resultadoDoCallable` será nulo.

Implementando o Callable

Sabendo disso, podemos começar a implementar o nosso primeiro comando, que poderia fazer uma chamada a um *web service*:

```
public class ComandoC2ChamaWs implements Callable<String> {

    private PrintStream saida;

    public ComandoC2ChamaWs(PrintStream saida) {
        this.saida = saida;
    }

    @Override
    public String call() throws Exception {

        System.out.println("Servidor recebeu comando c2 - WS");

        saida.println("Processando comando c2 - WS");

        Thread.sleep(15000);

        int numero = new Random().nextInt(100) + 1;

        System.out.println("Servidor finalizou comando c2 - WS");

        return Integer.toString(numero);
    }
}
```

É apenas uma simulação e realmente não acontece uma chamada de um web service. No entanto, para ter *algum resultado*, estamos gerando um número aleatório, e uma outra tarefa, bem similar à primeira, para simular um acesso demorado ao banco de dados:

```
public class ComandoC2AcessaBanco implements Callable<String> {

    private PrintStream saida;

    public ComandoC2AcessaBanco(PrintStream saida) {
        this.saida = saida;
    }

    @Override
    public String call() throws Exception {

        System.out.println("Servidor recebeu comando c2 - Banco");
```

```
        saida.println("Processando comando c2 - Banco");

        Thread.sleep(15000);

        int numero = new Random().nextInt(100) + 1;

        System.out.println("Servidor finalizou comando c2 - Banco");

        return Integer.toString(numero);
    }
}
```

Novamente, a única coisa que realmente acontece é a geração de um número aleatório.

E para submeter as duas tarefas, precisamos mexer dentro do nosso `switch` da classe `DistribuirTarefa`:

```
switch (comando) {

    // case c1

    case "c2": {
        saidaCliente.println("Confirmação do comando c2");

        //criando os dois comandos
        ComandoC2ChamaWs c2WS = new ComandoC2ChamaWs(saidaCliente);
        ComandoC2AcessaBanco c2Banco = new ComandoC2AcessaBanco(saidaCliente);

        //passando os comandos para o pool, resultado é um Future
        Future<String> futureWS = this.threadPool.submit(c2WS);
        Future<String> futureBanco = this.threadPool.submit(c2Banco);

        break;
    }

    // restante do código omitido
}
```

Tarefa para juntar os resultados

Repare que no `case` acima temos dois `Future`, um para cada `Callable`. Lembrando, para pegar o resultado do `Future` usamos o método `get`:

```
String resultadoWS = futureWS.get();
```

Já falamos que, ao chamar o método `get()` do `Future`, vamos bloquear a thread (a thread fica travada) e não queremos isso, pois o `switch` executa a thread para receber qualquer comando, e não queremos travar o recebimento de outros comandos.

Consequentemente, usaremos uma nova thread para receber os resultados dos `Future`! Faz sentido?

Nesse caso, não seria preciso usar `Callable` pois a tarefa vai juntar os resultados, escrever na saída do cliente sem retornar nada. Poderíamos usar a interface `Runnable` então, mas nada impede praticar mais um pouco com `Callable` :)

O único truque é usar o tipo genérico `Void` (bizarro né?) e colocar `null` como retorno do método `call()` . Veja o código:

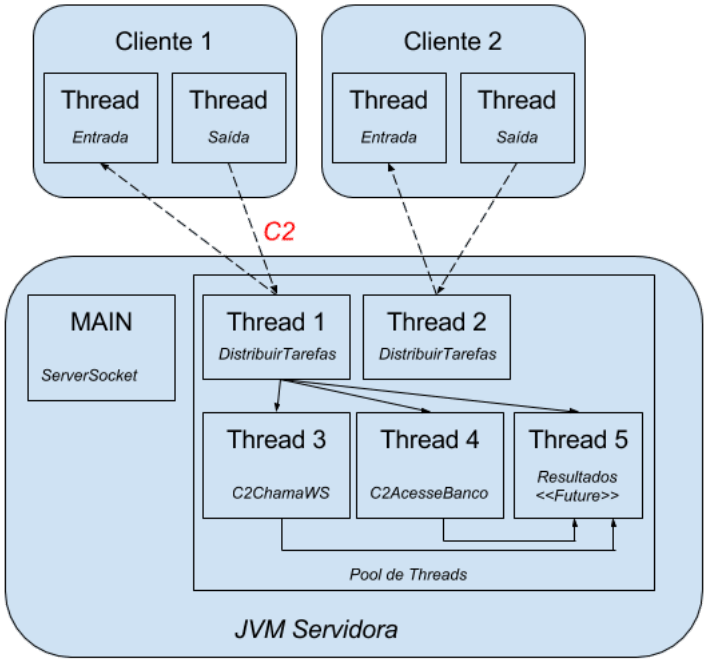
```
public class JuntaResultadosFutureWSFutureBanco implements Callable<Void> {

    private Future<String> futureWS;
    private Future<String> futureBanco;
    private PrintStream saidaCliente;

    public JuntaResultadosFutureWSFutureBanco(Future<String> futureWS,
            Future<String> futureBanco, PrintStream saidaCliente) {
        this.futureWS = futureWS;
        this.futureBanco = futureBanco;
        this.saidaCliente = saidaCliente;
    }

    //não queremos devolver nada, então usamos um tipo que representa nada: Void
    public Void call() {

        System.out.println("Aguardando resultados do future WS e Banco");

        try {
            String numeroMagico = this.futureWS.get(20, TimeUnit.SECONDS);
            String numeroMagico2 = this.futureBanco.get(20, TimeUnit.SECONDS);

            this.saidaCliente.println("Resultado do comando c2: "
                    + numeroMagico + ", " + numeroMagico2);

        } catch (InterruptedException | ExecutionException | TimeoutException e) {

            System.out.println("Timeout: Cancelando a execução do comando c2");

            this.saidaCliente.println("Timeout na execução do comando c2");
            this.futureWS.cancel(true);
            this.futureBanco.cancel(true);
        }

        System.out.println("Finalizou JuntaResultadosFutureWSFutureBanco");

        return null; //esse Callable não tem retorno, por isso null
    }
}
```

Repare que estamos aguardando os resultados de cada `Future` por 20 segundos. Após isso, estamos juntando os resultados em uma única resposta. Deixamos o tratamento de exceção explícito, mas na verdade não é obrigatório, pois o método `call` pode jogar qualquer *exception*!

Por fim, falta submeter a tarefa que junta os dois futures e devolve a resposta para o cliente. Isso fica também dentro do nosso `switch` , dentro do `case` do comando `c2` :

```
switch (comando) {

    // case c1

case "c2": {
        saidaCliente.println("Confirmação do comando c2");

        ComandoC2ChamaWs c2WS = new ComandoC2ChamaWs(saidaCliente);
        ComandoC2AcessaBanco c2Banco = new ComandoC2AcessaBanco(saidaCliente);

        Future<String> futureWS = this.threadPool.submit(c2WS);
        Future<String> futureBanco = this.threadPool.submit(c2Banco);

        //novo, passando a tarefa para juntar os resultados para o pool
        this.threadPool.submit(new JuntaResultadosFutureWSFutureBanco(futureWS, futureBanco, sa

        break;
    }

    // restante do código omitido
}
```

Segue uma visualização da nossa implementação quando o cliente 1 envia o comando `c2`:



Ufa, isso com certeza foi um desafio para você e é fundamental praticar bastante os exercícios!

O que aprendemos?

- A interface `Callable` é uma *alternativa* à interface `Runnable`
  - As interfaces `Runnable` e `Callable` servem para *definir uma tarefa* de uma thread.
  - A diferença do `Callable` é que pode *retornar algo* e jogar *qualquer exceção*.
- Para usar um `Callable` com threads, sempre precisamos de um pool de threads (`ExecutorService`).
  - O método `submit(..)` do pool recebe um `Callable` retorna um `Future`

- O `Future` representa o resultado da execução que talvez não tenha terminado ainda.