03

Entendendo Volatile

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/threads2/capitulo-4.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

Revisão do servidor

Nesse capítulo vamos voltar a mexer no nosso servidor. Mas vamos revisar rapidamente o código, estamos criando o nosso servidor usando a classe `ServerSocket`, além de inicializar o pool de threads. Após isso, executamos um laço para aceitar novas conexões. Cada novo cliente será atendido através de uma nova thread:

```
public class ServidorTarefas {

    public static void main(String[] args) throws Exception {

        System.out.println("---- Iniciando Servidor ----");
        ServerSocket servidor = new ServerSocket(12345);
        ExecutorService threadPool = Executors.newCachedThreadPool();

        while (true) {
            Socket socket = servidor.accept();
            System.out.println("Aceitando novo cliente na porta " + socket.getPort());

            DistribuirTarefas distribuirTarefas = new DistribuirTarefas(socket);
            threadPool.execute(distribuirTarefas);
        }
    }
}
```

Aparentemente tudo está funcionando e realmente está. No entanto, não estamos fechando os recursos devidamente. Após o laço, devemos desligar o pool e fechar o `ServerSocket`.

```
threadPool.shutdown();
servidor.close();
```

Infelizmente o código não compila, pois o compilador alega que nunca irá executar esse pedaço de código. Isto é por causa do laço `while(true)`, que nunca terminará.

Para resolver este problema, vamos refatorar o nosso servidor. A ideia é que o servidor possua um atributo `estaRodando`, que define se o laço deve ser executado ou não:

```
private boolean estaRodando = true;
```

Colocaremos todo o nosso código do método `main` dentro de um método com o nome `rodar`. Para chamar o método, devemos criar uma instância da classe `ServidorTarefas`.

Segue o código após refatoração:

```
public class ServidorTarefas {

    private boolean estaRodando = true;

    private void rodar() throws IOException {
        System.out.println("---- Iniciando Servidor ----");
        ServerSocket servidor = new ServerSocket(12345);
        ExecutorService threadPool = Executors.newCachedThreadPool();

        while (this.estaRodando) { //usando o atributo
            Socket socket = servidor.accept();
            System.out.println("Aceitando novo cliente na porta " + socket.getPort());

            DistribuirTarefas distribuirTarefas = new DistribuirTarefas(socket);
            threadPool.execute(distribuirTarefas);
        }

        threadPool.shutdown();
        servidor.close();
    }

    public static void main(String[] args) throws Exception {
        ServidorTarefas servidor = new ServidorTarefas();
        servidor.rodar();
    }
}
```

Mudou muito pouco mas já estamos programando um pouco mais orientado a objeto.

Parando o servidor

Agora imagine que o nosso servidor possa ser terminado através de um comando do cliente. Ou seja, o cliente envia um comando e o servidor se desligará automaticamente.

Sabendo disso, já podemos melhorar o nosso servidor mais ainda. A ideia é inicializar todos os recursos dentro do construtor e fechar os recursos dentro de um método dedicado.

```
public class ServidorTarefas {

    private ServerSocket servidor;
    private ExecutorService threadPool;
    private boolean estaRodando;

    public ServidorTarefas() throws IOException {
        System.out.println("---- Iniciando Servidor ----");
        this.servidor = new ServerSocket(12345);
        this.threadPool = Executors.newCachedThreadPool();

        // remova as três linhas anteriores do método rodar()

        this.estaRodando = true;
    }
}
```

E um método para fechar os recurso, que chamaremos de `parar`:

```
public void parar() throws IOException {
    this.estaRodando = false;
    this.threadPool.shutdown();
    this.servidor.close();
}
```

Já podemos parar o servidor através de uma chamada de um método. Ótimo. Para acompanhar mais fácil vamos mostrar o código inteiro da classe `ServidorTarefas`:

```
public class ServidorTarefas {

    private ServerSocket servidor;
    private ExecutorService threadPool;
    private boolean estaRodando;

    public ServidorTarefas() throws IOException {
        System.out.println("---- Iniciando Servidor ----");
        this.servidor = new ServerSocket(12345);
        this.threadPool = Executors.newCachedThreadPool();
        this.estaRodando = true;
    }

    public void rodar() throws IOException {

        while (this.estaRodando) {

            Socket socket = this.servidor.accept();
            System.out.println("Aceitando novo cliente na porta " + socket.getPort());

            DistribuirTarefas distribuirTarefas = new DistribuirTarefas(socket);

            this.threadPool.execute(distribuirTarefas);
        }
    }

    public void parar() throws IOException {
        this.estaRodando = false;
        this.threadPool.shutdown();
        this.servidor.close();
    }

    public static void main(String[] args) throws Exception {
        ServidorTarefas servidor = new ServidorTarefas();
        servidor.rodar();
    }
}
```

Novo comando

Agora vamos implementar o novo comando. Lembrando que nossos comandos são analisados dentro da classe `DistribuirTarefas`. Nela vamos então adicionar mais um `case` para poder desligar o servidor:

```
switch(comando) {
    case "c1" : {
        saidaCliente.println("Confirmação do comando: " + comando);
        break;
    }
    case "c2" : {
        saidaCliente.println("Confirmação do comando: " + comando);
        break;
    }
    case "fim" : {
        saidaCliente.println("Desligando o servidor");
        //o que vamos fazer agora?
        break;
    }
    default : {
        saidaCliente.println("Comando não encontrado");
    }
}
```

O `case` já está pronto, no entanto é preciso ter o servidor em mãos para poder desligá-lo. Por isso passaremos no construtor da classe mais uma argumento, o nosso servidor:

```
//na classe ServidorTarefas, no método rodar
//passando this
DistribuirTarefas distribuidor = new DistribuirTarefas(socket, this);
```

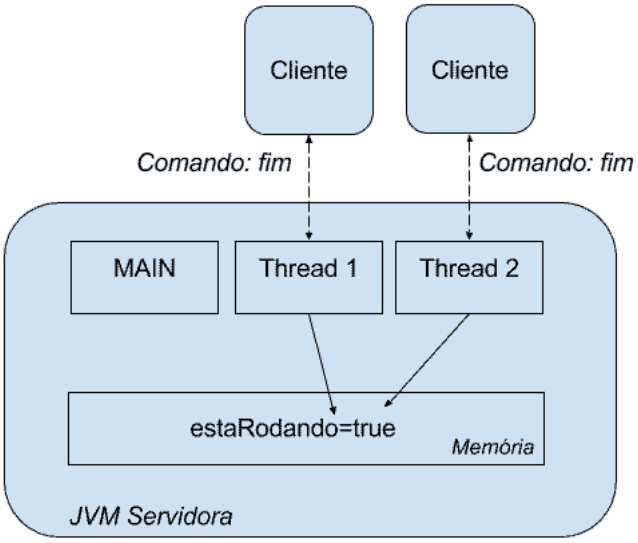
E a classe `DistribuirTarefas`, recebemos no construtor o `ServidorTarefas`;

```
public class DistribuirTarefas implements Runnable {

    private Socket socket;
    private ServidorTarefas servidor; //novo atributo

    // recebendo servidor
    public DistribuirTarefas(Socket socket, ServidorTarefas servidor) {
        this.socket = socket;
        this.servidor = servidor; //novo
    }

    // método run omitido
}
```

E dentro do método `run`, dentro do `case` adicionaremos a chamada do método `parar()`:

```
case "fim" : {
    saidaCliente.println("Desligando o servidor");
    servidor.parar();
    return;//sainda do método, break só sairia do switch
}
```

Agora temos a seguinte situação: o nosso cliente envia o comando *fim*, que é recebido através de uma thread dedicada e chama o método `parar()` que por sua vez manipula o atributo `estaRodando` .



Repare que o acesso ao atributo acontece em uma outra thread, e até poderia acontecer em paralelo. Será que isso não pode gerar um problema?

Simulando o problema

O acesso ao atributo através de várias threads pode sim criar problemas inesperados. No nosso exemplo é um pouco difícil de mostrar o problema acontecendo, pois temos clientes remotos, mas vamos simular o problema em um novo projeto.

No Eclipse, criaremos um *Java Project* com o nome **experimento**. Nele, criaremos a classe `ServidorDeTeste` , no pacote `br.com.alura.threads` . Copiaremos a classe abaixo, que é bem parecida com nossa classe `ServidorTarefas*` :*

```
package br.com.alura.threads;

public class ServidorDeTeste {

    private boolean estaRodando = false;

    public static void main(String[] args) throws InterruptedException {
        ServidorDeTeste servidor = new ServidorDeTeste();
        servidor.rodar();
        servidor.alterandoAtributo();
    }

    private void rodar() {
        new Thread(new Runnable() {

            public void run() {
                System.out.println("Servidor começando, estaRodando = " + estaRodando );

                while(!estaRodando) {}

                System.out.println("Servidor rodando, estaRodando = " + estaRodando );

                while(estaRodando) {}
```

```
                System.out.println("Servidor terminando, estaRodando = " + estaRodando );
            }
        }).start();
    }

    private void alterandoAtributo() throws InterruptedException {
        Thread.sleep(5000);
        System.out.println("Main alterando estaRodando = true");
        estaRodando = true;

        Thread.sleep(5000);
        System.out.println("Main alterando estaRodando = false");
        estaRodando = false;
    }
}
```

Repare que temos o mesmo atributo booleano, com a diferença que o atributo está sendo inicializado como `false`. No código, temos duas threads trabalhando, `Main` e `Servidor`. A thread `Main` dorme por um tempo, altera esse atributo para `true`, dorme novamente, e altere para `false`. Muito simples.

Quando o `Main` dorme, a outra thread `Servidor` tem tempo para se inicializar. No método `run`, testamos o atributo `estaRodando`, parando a thread enquanto estiver como `false`:

```
// enquanto estiver false, fica no laço
while(!estaRodando) {}
```

Ou seja, a thread fica rodando para sempre a não ser alguém altere o atributo. E como já falamos, isso é a tarefa do `Main`. Após alteração do atributo para `true` pela thread `Main`, paramos novamente a thread `Servidor` com um laço:

```
// enquanto estiver true, fica no laço
while(estaRodando) {}
```

O `Main` dorme novamente um pouco e altera o atributo `estaRodando` para `false`, para *liberar* a thread `Servidor`.

Resumindo, através do atributo `estaRodando`, a thread `Main` controla a execução da outra. Antes de rodar o código, simulamos uma vez a execução! A saída esperada está abaixo:

```
Servidor começando, estaRodando = false
Main alterando estaRodando = true
Servidor rodando, estaRodando = true
Main alterando estaRodando = false
Servidor terminando, estaRodando = false
```

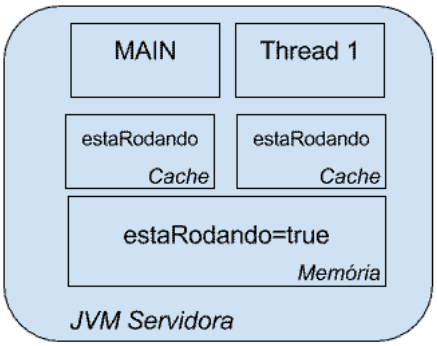
No entanto, ao testar e executar o código, a saída é outra e a JMV nunca desliga! Veja o que é impresso no console:

```
Servidor começando, estaRodando = false
Main alterando estaRodando = true
Main alterando estaRodando = false
```

Repare que a thread `Main` alterou o atributo como planejamos, mas mesmo assim não fez que a outra thread `Servidor` sai do laço! O que está acontecendo?

Entendendo o volatile

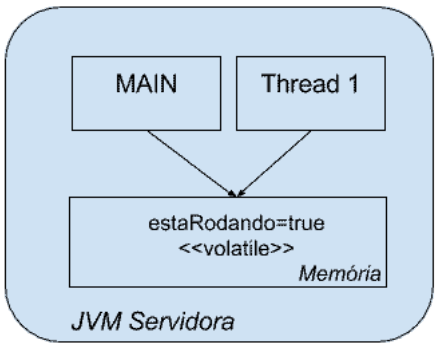
O problema é que uma thread pode cachear variáveis e é muito provável que isso acontecerá! Cada thread é mapeada para uma thread nativa do sistema operacional, e esses caches nativos vão aproveitar o cache da CPU. Enquanto o nosso atributo fica na memória principal, a thread o guarda no cache da CPU!



Então, como podemos dizer que não queremos usar esse cache? Isso é muito fácil e o Java possui uma palavra chave para tal: `volatile` .

O nosso atributo será `volatile` , que significa que cada thread deve acessar diretamente a memória principal.

```
private volatile boolean estaRodando = false;
```



Ao executar o nosso código novamente, realmente temos a saída esperada:

```
Servidor começando, estaRodando = false
Main alterando estaRodando = true
Servidor rodando, estaRodando = true
Main alterando estaRodando = false
Servidor terminando, estaRodando = false
```

Resumindo, quando declaramos uma variável como `volatile` , definimos:

- Que o valor dessa variável nunca será cacheado pela thread, todo o acesso vai diretamente na memória dessa variável.
- O acesso funciona de maneira atômica, como se fosse sincronizado.

Usando variáveis atômicas

Há uma alternativa ao uso da palavra chave `volatile`. Em vez de usar `volatile` diretamente, podemos utilizar classes do pacote `java.util.concurrent.atomic`. Nesse pacote encontraremos uma classe com o nome `AtomicBoolean` que garante que todas as threads usam essa variável de maneira atômica, sem cache.

Vamos voltar ao nosso projeto real, **servidor-tarefas**, na classe `ServidorTarefas`, e definir a variável `estaRodando` como o tipo `AtomicBoolean`.

```
private AtomicBoolean estaRodando;
```

E no construtor:

```
public ServidorTarefas() throws IOException {
    System.out.println("---- Iniciando Servidor ----");
    this.servidor = new ServerSocket(12345);
    this.threadPool = Executors.newCachedThreadPool();
    this.estaRodando = new AtomicBoolean(true); // devemos dar new em AtomicBoolean
}
```

Para recuperar o valor, devemos chamar um método `get`. Aplicaremos isso no nosso laço `while` do método `rodar()` da classe `ServidorTarefas`:

```
while (this.estaRodando.get()) {
```

E para alterar o valor, usamos o método `set` dentro do método `parar()`:

```
public void parar() throws IOException {
    this.estaRodando.set(false);
    this.threadPool.shutdown();
    this.servidor.close();
}
```

Assim garantimos que alterações nesse atributo pelas threads realmente são vistas através de todas as threads.

Segue uma vez o código completo da classe:

```
public class ServidorTarefas {

    private ServerSocket servidor;
    private ExecutorService threadPool;
    private AtomicBoolean estaRodando;

    public ServidorTarefas() throws IOException {
        System.out.println("---- Iniciando Servidor ----");
        this.servidor = new ServerSocket(12345);
        this.threadPool = Executors.newCachedThreadPool();
        this.estaRodando = new AtomicBoolean(true);
    }
```

```
    public void rodar() throws IOException {

        while (this.estaRodando.get()) {
            try {
                Socket socket = this.servidor.accept();
                System.out.println("Aceitando novo cliente na porta " + socket.getPort());

                DistribuirTarefas distribuirTarefas = new DistribuirTarefas(socket, this);

                this.threadPool.execute(distribuirTarefas);
            } catch (SocketException e) {
                System.out.println("SocketException, está rodando? " + this.estaRodando);
            }
        }
    }

    public void parar() throws IOException {
        this.estaRodando.set(false);
        this.threadPool.shutdown();
        this.servidor.close();
    }

    public static void main(String[] args) throws Exception {

        ServidorTarefas servidor = new ServidorTarefas();
        servidor.rodar();
    }
}
```

O que aprendemos?

- Threads possuem um cache.
  - Esse cache faz com que nem sempre todas as variáveis serão vistas e atualizadas de maneira atômica.
- A palavra chave `volatile` evita o uso desse cache e faz que as threads sempre acessem a memória principal.
- Como alternativa, podemos utilizar as classes do pacote `java.util.concurrent.atomic`
  - Vimos a classe `AtomicBoolean` como alternativa ao uso do `volatile`