01

Entendendo Deadlock

Transcrição

Neste capítulo vocês verão um problema que pode acontecer quando threads precisam acessar mais de um recurso/objeto sincronizado ao mesmo tempo. O nosso exemplo aqui é algo tipicamente usado em aplicações web. Quando usamos um banco de dados, normalmente criamos e administramos as conexões através de um pool de conexões. O pool de conexões é responsável por abrir e fechar as conexões. Além disso, é preciso gerenciar uma transação com banco para garantir que os dados realmente serão inseridos corretamente. E aí encontramos nossos dois recursos/objetos: um `PoolDeConexao` e um `GerenciadorDeTransacao`.

Criação do ambiente

Vamos simular o acesso aos objetos pelos threads para mostrar um problema. As duas classes são apenas um esboço das classes reais, muito simplificado para o nosso objetivo. Você pode copiar as classes abaixo:

A classe `PoolDeConexao`:

```
public class PoolDeConexao {

    public String getConnection() {

        System.out.println("Emprestando conexão");

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        return "connection";
    }
}
```

E o `GerenciadorDeTransacao`:

```
public class GerenciadorDeTransacao {

    public void begin() {

        System.out.println("Começando a transação");

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Ambas as classes usam o método `Thread.sleep(5000)` para atrasar a execução por 5 segundos, para simular o acesso.

Além disso, preparamos uma classe com o método `main()` que já instancia o `GerenciadorDeTransacao` e o `PoolDeConexao`:

```
public class PrincipalBanco {

    public static void main(String[] args) {

        GerenciadorDeTransacao tx = new GerenciadorDeTransacao();
        PoolDeConexao pool = new PoolDeConexao();

    }
}
```

Vários threads, dois objetos

Uma vez preparadas as classes, podemos começar a trabalhar com threads. Em uma aplicação web, cada requisição será executada usando um thread dedicado. Vamos pensar que dois desenvolvedores acessam a nossa aplicação para gravar um dado no banco de dados. Isso significa que temos duas requisições, ou seja, dois threads rodando ao mesmo tempo.

O primeiro passo é criar uma tarefa para acessar o banco de dados (por exemplo, para executar algum comando SQL):

```
public class PrincipalBanco {

    public static void main(String[] args) {

        GerenciadorDeTransacao tx = new GerenciadorDeTransacao();
        PoolDeConexao pool = new PoolDeConexao();

        new Thread(new TarefaAcessaBanco(pool, tx)).start();

    }
}
```

Assim, já podemos criar a classe que representa a tarefa usando a interface `Runnable`:

```
public class TarefaAcessaBanco implements Runnable {

    private PoolDeConexao pool;
    private GerenciadorDeTransacao tx;

    public TarefaAcessaBanco(PoolDeConexao pool, GerenciadorDeTransacao tx) {
        this.pool = pool;
        this.tx = tx;
    }

    @Override
    public void run(){
```

```
    }
}
```

Dentro do nosso método `run`, vamos utilizar os dois objetos `pool` e `tx`. Como os objetos serão compartilhados entre threads, é preciso sincronizar o acesso através de um bloco `synchronized`:

```
synchronized (pool) {

    System.out.println("Peguei a chave do pool");
    pool.getConnection();

}
```

Não basta só pegar a chave do pool, é preciso também ter a chave do gerenciador da transação. Ou seja, vamos ter um bloco `synchronized` dentro do outro. No método `run`, adicionaremos o seguinte código:

```
synchronized (pool) {

    System.out.println("Peguei a chave do pool");
    pool.getConnection();

    synchronized (tx) {

        System.out.println("Peguei a chave da tx");
        tx.begin();

    }
}
```

A primeira tarefa está completa. Agora vamos atacar a segunda. Na classe `PrincipalBanco`, copiamos a linha que inicializa a thread, criando uma nova tarefa:

```
public class PrincipalBanco {

    public static void main(String[] args) {

        GerenciadorDeTransacao tx = new GerenciadorDeTransacao();
        PoolDeConexao pool = new PoolDeConexao();

        new Thread(new TarefaAcessaBanco(pool, tx)).start();
        new Thread(new TarefaAcessaBancoProcedimento(pool, tx)).start();

    }
}
```

Com a nova tarefa, escrita de maneira correta (no vídeo escrevi errado, desculpe!):

```
public class TarefaAcessaBancoProcedimento implements Runnable {

    private PoolDeConexao pool;
    private GerenciadorDeTransacao tx;
```

```
    public TarefaAcessaBancoProcedimento(PoolDeConexao pool,
            GerenciadorDeTransacao tx) {
        this.pool = pool;
        this.tx = tx;
    }

    @Override
    public void run() {

    }
}
```

No entanto, o desenvolvedor muda a ordem nos blocos `synchronized`. Primeiro opta em pegar a chave da tx e depois do pool:

```
// dentro do método run da classe TarefaAcessaBancoProcedimento
synchronized (tx) {

    System.out.println("Peguei a chave da tx");
    tx.begin();

    synchronized (pool) {

        System.out.println("Peguei a chave do pool");
        pool.getConnection();
    }
}
```

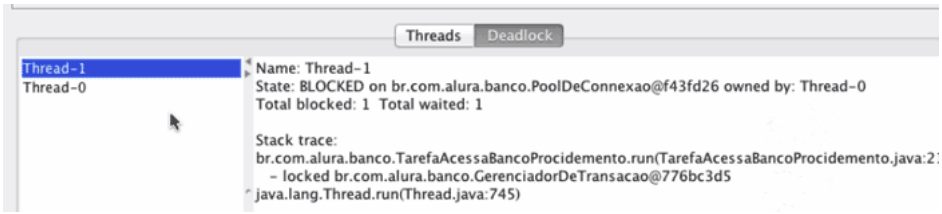
Entendendo Deadlock

Invertemos a ordem de propósito e, ao executar a nossa classe `PrincipalBanco`, percebemos que a execução nunca termina. Analisando a saída no console, podemos ver que ambos os threads executaram o início do primeiro bloco `synchronized` mas nunca entraram no segundo!

Um thread impede a execução do outro, pois cada um possui a chave de um objeto que o outro precisa para continuar. Temos um impasse e esse impasse é chamado de **deadlock**.

Podemos visualizar o problema através da ferramenta *jconsole*. Ao inicializar e se conectar com a JVM em execução, podemos encontrar os nossos dois threads na aba *Threads*:

Até existe um botão específico para descobrir se existe um *deadlock*:



Resolvendo o problema

Óbvio que causamos o problema de propósito, mas em algum projeto real com mais desenvolvedores envolvidos o perigo é real. Para resolver o problema, devemos sempre obter as chaves na mesma ordem. Ambos os blocos devem primeiro tentar obter a chave do `pool` e só depois a chave do gerenciador de transação.

```
// dentro do método run da classe TarefaAcessaBancoProcedimento

synchronized (pool) {  //primeiro pool

    System.out.println("Peguei a chave do pool");
    pool.getConnection();

    synchronized (tx) {

        System.out.println("Peguei a chave da tx");
        tx.begin();
    }
}
```

Ao executar novamente o código, uma tarefa será executada depois da outra, sem ter o perigo de *deadlock*.