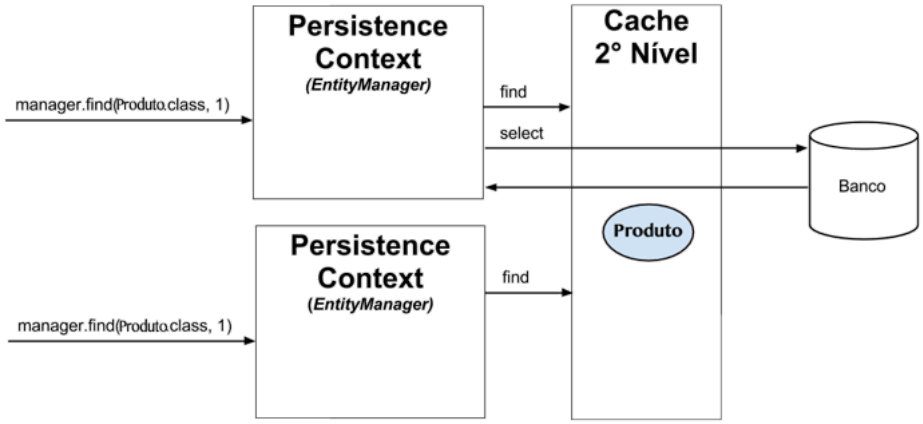
06

Conhecendo o cache de segundo nível

Transcrição

O que precisamos é de um espaço de "cache" que seja compartilhado entre os vários *EntityManagers* da nossa aplicação e que seja utilizado quando o cache de primeiro nível não detiver a informação desejada. Esse espaço chamamos de **cache de segundo nível**.



Em geral, lidar com um cache de segundo nível é bem mais complexo do que tratar com um de primeiro, uma vez que a possibilidade de trabalhar com dados desatualizados (*stale*) é bem maior. Os objetos desse cache são invalidados quando há alguma operação de escrita na entidade (como update). Se houver algum outro sistema atualizando os dados no banco sem passar pela JPA seu uso pode se tornar inexecutável.

Vamos compreender, na sequência, como configuramos o cache de segundo nível na nossa aplicação!

Configurando o EhCache

Por padrão, o cache de segundo nível vem desabilitado. Para ativá-lo, precisamos adicionar uma chave a mais na configuração do Hibernate. Em nosso caso, na classe `JpaConfigurator`:

(OBS: Se você mudou a estratégia do Hibernate Mapping pra **update**, altere novamente para **create-drop**)

```
public class JpaConfigurator {
    ...
    @Bean
    public LocalContainerEntityManagerFactoryBean getEntityManagerFactory(DataSource dataSource

        ...
        props.setProperty("hibernate.cache.use_second_level_cache", "true");
        ...
    }
}
```

Se estivermos utilizando o `persistence.xml`, adicionamos uma *tag* com a mesma propriedade:

```
<property name="hibernate.cache.use_second_level_cache" value="true" />
```

Conhecendo o provedor EhCache

Além disso, é necessário informar ao Hibernate qual será o provedor de *cache* que usaremos. O **JBoss Wildfly** já possui um *provider* embarcado, ele se chama `infinispan`. Em nosso projeto, usaremos o *EhCache* que é um dos *providers* mais comuns de se trabalhar com *Hibernate*.

Para configurá-lo como `provider`, adicionaremos a propriedade `hibernate.cache.region.factory_class`:

```
public class JpaConfigurator {
    ...
    @Bean
    public LocalContainerEntityManagerFactoryBean getEntityManagerFactory(DataSource dataSource

        ...
        props.setProperty("hibernate.cache.use_second_level_cache", "true");
        props.setProperty("hibernate.cache.region.factory_class", "org.hibernate.cache.ehcache.S
        ...
    }
}
```

Ou no `persistence.xml`:

```
<property name="hibernate.cache.region.factory_class" value="org.hibernate.cache.ehcache.Singlet
```

Com essa configuração já podemos começarmos a desfrutar do poder do cache de segundo nível. Porém, precisamos configurar nossas classes para usar *cache*.

Usando o cache de segundo nível na aplicação

Antes de realmente configurarmos o cache, vamos analisar o comportamento atual do nosso projeto. Para isso, reiniciamos o Tomcat e abrimos nossa aplicação. Para testar, acessamos a página de edição de algum produto, por exemplo: [http://localhost:8080/projeto_jpa_avancado/produto/1/form][1] (http://localhost:8080/projeto_jpa_avancado/produto/1/form%5D%5B1%5D).

Podemos observar, através do console, que o Hibernate logou duas queries. A primeira buscará o produto e a segunda as categorias relacionadas ao produto que procuramos.

```
Hibernate: select produto0_.id as id1_2_0_, produto0_.descricao as descrica2_2_0_, produto0_.lir

Hibernate: select categorias0_.Produto_id as Produto_1_2_0_, categorias0_.categorias_id as categ
```

Se atualizarmos a página veremos que as mesmas queries são disparadas novamente contra o banco. Agora, vamos configurar a classe `Produto` para ser armazenada no `Cache de segundo nível`. Para isso, basta anotá-la com `@Cache`:

```
@Entity
@Cache
public class Produto {
```

Ao escrevermos `Produto` para anotar a entidade, recebemos um erro de compilação! Isso acontece porque, ao usarmos dessa maneira, não especificamos qual a estratégia de concorrência que o cache deve adotar. E essa é uma informação obrigatória ao usar `@Cache` !

Existem algumas estratégias para se lidar com eventuais situações de concorrência pois, afinal, quando trabalhamos com dados em sistemas distribuídos temos várias formas de lidar com a concorrência. Um problema muito comum é manter a consistência do estado. Lembra-se que fizemos isso no capítulo de "Lock"? Repare que se o produto não pudesse ser alterado não precisaríamos do `lock` .

Se não fosse possível alterar o produto, poderíamos utilizar a estratégia **READ_ONLY** que abre mão dos `locks` e sincronizações por não permitir alterações no estado do objeto. Essa é forma mais barata, computacionalmente, de se trabalhar com *cache de segundo nível*.

Em situações em que alterações de estado são necessárias e há grandes chances de que elas ocorram **simultaneamente**, podemos adotar a estratégia **READ_WRITE** que consome muitos recursos para garantir que a última versão da entidade no banco seja **a mesma** que está no cache.

Porém, alterações ocorrendo ao mesmo tempo são incomuns. Ainda que não precisemos de todos os recursos usados pela estratégia **READ_WRITE**, faz-se necessário modificar o estado da entidade. Nessa situação, podemos usar a estratégia **NONSTRICT_READ_WRITE** ideal, ou seja, quando não há problemas em ler dados inconsistentes caso hajam alterações simultâneas.

Em ambientes JTA, por exemplo, os servidores de aplicação podem optar pela estratégia **TRANSACTIONAL**. Em nosso projeto iremos utilizar a estratégia `NONSTRICT_READ_WRITE` :

```
@Entity
@Cache(usage=CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Produto {
```

Vamos rodar nossa aplicação acessando a página de edição de produtos:
[http://localhost:8080/projeto_jpa_avancado/produto/1/form][1]
(http://localhost:8080/projeto_jpa_avancado/produto/1/form%5D%5B1%5D).

Repare que, novamente, os dois selects (um buscando o produto e um buscando as categorias) foram feitos. Agora, atualize a página e fique de olho no console. Quantos selects foram disparados?!

> **Apenas um**, que busca as categorias do produto! O primeiro `select` armazenou o produto no cache de segundo nível, portanto, não houve necessidade do segundo `select` .

Mas, agora, aconteceu um select extra que não esperávamos. Por mais que ele não busque novamente pelo produto no banco de dados, ele ainda dispara uma query para pegar as categorias daquele produto. Por que isso acontece?

Cache de Collections

Quando configuramos o cache para a entidade `Produto` não dissemos que queríamos cachear também suas associações. Assim, podemos passar para o Hibernate que desejamos armazenar também as categorias de cada produto anotando o seu relacionamento com @Cache:

```
@ManyToMany
@Cache(usage=CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
private List<Categoria> categorias = new ArrayList<>();
```

Vamos reiniciar o Tomcat e fazer mais um teste na página de edição do produto. No primeiro acesso recebemos no log os mesmos dois selects, porém ao acessarmos novamente receberemos, mais uma vez, os dois selects. E, mesmo que atualizemos novamente, veremos de novo os dois selects. O que será que aconteceu? Ao em vez de apenas um select extra ocorreram **dois**!

Isso acontece porque o cache do `collections` armazena apenas os `ids` das associações, ou seja, os `ids` das categorias associadas com o produto! O que de fato nos interessa, é o nome da categoria. Portanto, com esses `ids` em mãos o Hibernate busca no banco os dados das categorias relacionadas a esse produto (música e tecnologia, por isso dois selects!). Esse problema é bem parecido com o problema do "N + 1" onde para cada produto temos uma query que busca as categorias. Como evitar então que esses dois selects sejam disparados?

Armazenando a Categoria no cache

Como vimos, usando um cache de segundo nível uma query só é disparada ao banco quando a entidade não é encontrada no cache. No nosso caso, as categorias não estão no cache e por isso houve uma busca. Vamos adicionar a anotação `@Cache` também na entidade `Categoria` :

```
@Entity
@Cache(usage=CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Categoria {
```

Vamos testar se tudo funciona como esperado acessando novamente a página de produtos e atualizando a página. Repare que apenas os dois primeiros selects foram feitos!