02

Caçando seus gargalos com o Hibernate Statistics

Transcrição

Coletando informações sobre a camada de persistência

Quando trabalhamos com um banco de dados, é importante sabermos se os recursos empregados estão sendo utilizados corretamente, principalmente, aqueles que podem trazer problemas de escalabilidade e performance. Como, por exemplo, conferir a existência de uma eventual *connection leak* devido a uma conexão que ficou aberta, ou, um número muito grande *miss* do cache que pode significar que devemos reconfigurar a estratégia de cache.

É difícil mensurar esses atributos olhando apenas a aplicação em execução e analisando logs. Vários desses problemas são descobertos realizando-se uma auditoria sobre o uso de recursos , tais como, conexões, transações e cache. Para nos ajudar nessa tarefa, o Hibernate possui uma ferramenta que fornece estatísticas do uso desses recursos.

Configurando o Hibernate Statistics

Esse recurso vem desabilitado por padrão, para ativá-lo, adicione uma propriedade na configuração da JPA. Na nossa classe `JpaConfigurator` acrescente:

```
@Bean
public LocalContainerEntityManagerFactoryBean getEntityManagerFactory(DataSource dataSource) {
    ...
    props.setProperty("hibernate.generate_statistics", "true");
    ...
}
```

Após ativar o *Hibernate Statistics*, precisamos acessar a API que fornece todos esses dados estatísticos. Como ela é ***específica do Hibernate*** e não faz parte da JPA, precisaremos pegar a instância de um objeto interno do Hibernate chamado `SessionFactory` e, a partir dele, teremos a instância do objeto `Statistics` :

```
SessionFactory sessionFactory =  emf.unwrap(SessionFactory.class);
Statistics statistics = sessionFactory.getStatistics();
```

A classe `Statistics` possui vários métodos úteis que usaremos para conseguir as informações. No nosso projeto, iremos usar essa classe em uma *view* específica para mostrar as estatísticas e, por isso, precisamos que ela seja acessível via *Expression Language*. O SpringMVC já está configurado para que seja possível acessar os objetos do seu contexto via EL, o que precisamos fazer é adicionar o `Statistics` no contexto do Spring. Faremos isso adicionando na classe `JPAConfigurator` o método abaixo:

```
@Bean
public Statistics statistics(EntityManagerFactory emf) {
    return emf.unwrap(SessionFactory.class).getStatistics();
}
```

Visualizando dados sobre o *cache de queries*:

Vamos testar se o *cache de queries* realmente funciona como esperado. No arquivo `WEB-INF/views/estatisticas/index.jsp` vamos exibir os relatórios do *Hibernate Statistics*:

```
<tr>
    <td>Cache</td>
    <!-- Hit -->
    <td></td>
    <!-- Miss -->
    <td></td>
    <! -- Conections -->
    <td></td>
</tr>
```

Entre as tags `<td>` vamos chamar os métodos da classe `Statistics` que mostram a quantidade de queries encontradas no cache (*hit*) e a quantidade de queries não encontradas (*miss*):

```
<tr>
    <td>Cache</td>
    <!-- Hit -->
    <td>${statistics.queryCacheHitCount}</td>
    <!-- Miss -->
    <td>${statistics.queryCacheMissCount}</td>
</tr>
```

Reinicie o *Tomcat*, acesse o projeto e faça uso do filtro para buscar um produto de tecnologia. Após a consulta, vamos exibir o relatório clicando no link *Estatísticas* na barra superior da tela.

Repare que houve um *miss* e nenhum *hit*. Isso ocorre porque na primeira vez que buscamos não há elementos no cache para reaproveitar, por isso, ele precisa realizar uma busca no banco de dados (*miss*).

Agora, volte a *Home* e realize novamente a pesquisa. Como já rodamos, anteriormente, essa *query* estão guardados os *ids* da *query*. Por isso, acontece um *hit*.

Outras estatísticas

Na classe `Statistics` há alguns outros métodos que fornecem dados interessantes. Vamos conhecer um que fornece a quantidade de conexões pedidas pelo `EntityManager` . Vamos mexer, novamente, no arquivo de relatório para editar a terceira linha da tabela (`<td></td>`) , assim, chamando o método `getConnectCount` :

```
<td>${statistics.connectCount}</td>
```

Para testar, reinicie o *Tomcat* e volte a essa página. Logo de cara conseguimos ver **três** conexões abertas. Por que será?

Na nossa aplicação usamos um *pool* de conexões, certo?! Há alguns capítulos configuramos esse *pool* para iniciar com **três** conexões. Vamos recordar? Na classe `JpaConfigurator` observe o método `getDataSource` :

```
@Bean()
public ComboPooledDataSource getDataSource() {
    ...
    ds.setMinPoolSize(3);
}
```