Entendendo o NoSuchMethodError e o ClassLoader hell

É frequente que diversas aplicações Web sejam implantadas em um mesmo Servlet Container, em uma mesma máquina virtal, e também é usual que muitas delas usem bibliotecas e frameworks em comum. Imagine duas aplicações, a **APP1** e a **APP2**, que utilizam a mesma versão do Hibernate: onde devemos colocar os JARs?

Muitos dirão para colocar os JARs em algum diretório `common/lib` do Servlet Container; outros dirão para jogar diretamente na variável de ambiente `CLASSPATH`. Esses eram conselhos frequentes, em especial quando era raro ter mais de uma aplicação Java implantanda em um mesmo servidor. Mas são boas práticas? Para entender os graves problemas que pode surgir nessa situação, é preciso primeiro entender como funciona o carregamento das classes pela JVM.

O **classloader** [60] é o objeto responsável pelo carregamento de classes Java para a memória a partir de um array de bytes contendo seus bytecodes. É representado pela classe abstrata `java.lang.ClassLoader` e existem já diversas implementações concretas, como a `java.net.URLClassLoader`, responsável por carregar classes buscando em determinadas `URL`s.

Classloaders são um ponto chave da plataforma Java. Com eles podemos carregar diferentes classes com **exatamente o mesmo nome** e mantê-las de forma distinta, em diferentes **espaços de nomes** (*namespaces*). Basta que usemos classloaders diferentes. Uma classe é **unicamente identificada** dentro da JVM através de seu nome completo (incluindo o nome do pacote, o chamado *fully qualified name*), mais o classloader que a carregou.

É desta forma que servidores de aplicação conseguem manter separadas versões de bibliotecas diferentes: podemos ter o Hibernate 3.3 em uma aplicação Web e o Hibernate 3.1 em outra. Para isso, dois classloaders diferentes são necessário para carregar classes de duas diferentes aplicações. Como um classloader carrega uma classe uma única vez, se nesse caso fosse utilizado o mesmo classloader, haveria conflito de classes com o mesmo nome (por exemplo a `org.hibernate.Session` das duas versões de Hibernate diferente), sendo carregada apenas a primeira que fosse encontrada.

Porém, o mecanismo de carregamento de classes não é tão simples. Por uma medida de segurança, os classloaders são criados e consultados de maneira hierárquica.

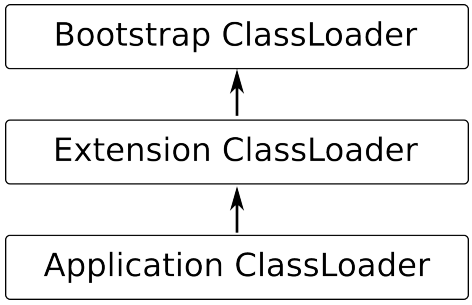

Figura 6.1: Hierarquia de ClassLoaders

A figura mostra os três classloaders que a JVM sempre irá utilizar por padrão em qualquer aplicação.

Antes de um classloader iniciar um carregamento, ele pergunta ao seu pai (*parent classloader*) se este não consegue carregar a classe em questão. Se o *parent* classloader conseguir, ela será carregada através deste. Esse funcionamento hierárquico é o que garante **segurança** à plataforma Java. É assim que classes importantes, como as do `java.lang`, serão sempre lidas pelo *bootstrap* classloader. Isso é essencial: caso contrário poderíamos, por exemplo, adicionar uma classe `java.net.Socket` à nossa aplicação, e quando esta fosse implantada no servidor, todas as outras classes dessa aplicação que fizessem uso da `Socket` estariam usando agora um provável cavalo de tróia.

Logo, é vital um certo conhecimento desses classloaders fundamentais:

- **Bootstrap ClassLoader**:

  Carrega classes do `rt.jar` e é o único que realmente não tem `parent` algum: é o pai de todos os outros, evitando que qualquer outro classloader tente modificar as classes do rt.jar (pacote `java.lang`, por exemplo);

- **Extension ClassLoader**:

  Carrega classes de jars dentro do diretório na propriedade `java.ext.dirs`, que por padrão tem o valor `$JAVA_HOME/lib/ext`. Na JVM da Sun, é representado pela classe interna `sun.misc.Launcher$ExtClassLoader`. Tem como pai o *Bootstrap ClassLoader*;

- **Application ClassLoader** ou **System ClassLoader**:

  Carrega tudo definido no `CLASSPATH`, tanto o da variável de ambiente, quanto o da linha de comando (passado como argumento por `-cp`). Representado pela classe interna `sun.misc.Launcher$AppClassLoader`. Tem como pai o *Extensions ClassLoader*.

O **Bootstrap** classloader é quem carrega todas as classes da biblioteca padrão (o `rt.jar` inclusive). Depois dele sua aplicação pode carregar classes usando outros classloaders, já que a maioria das classes estão fora desse JAR.

As aplicacões podem criar novos classloader e incrementar essa hierarquia. Os containers, por exemplo, costumam criar muitos outros classloaders além dos padrões da máquina virtual. Em geral, criam um **Container ClassLoader**, responsável pelos JARs do container e de uma certa pasta, como uma *common/lib*. Além deste, o container vai precisar de um classloader específico para cada aplicação (contexto) implantada(o). Esse **WebApplication ClassLoader** é responsável pelo carregamento das classes do *WEB-INF/classes* e do *WEB-INF/lib* em aplicações Web, por exemplo.

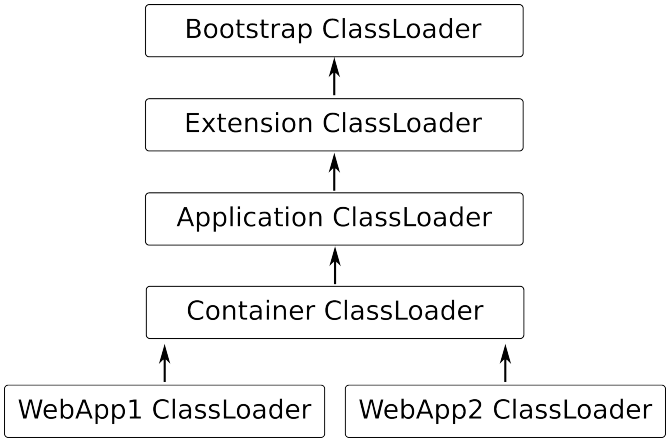

Figura 6.2: Hierarquia usual de ClassLoaders no Container

Conhecendo essa hierarquia podemos agora perceber o problema quando desenvolvedores desavisados jogam JARs em diretórios compartilhados por todas as aplicações de um determinado servidor, como o caso do antigo diretório `common/lib` do Apache Tomcat.

Enquanto tanto a aplicação **APP1** e **APP2** usam a mesma versão do Hibernate, tudo ocorrerá bem. No momento que a **APP1** usar outra versão, como fazemos? Jogar a versão mais nova no `WEB-INF/lib` da **APP1** não resolve o problema, pois a hierarquia de classloaders forçará que o carregamento de `org.hibernate.Session` seja feito pelo classloader que carrega classes de `common/lib`!

Esse problema é muito mais grave do que aparenta: o tipo de erro que pode surgir é de difícil interpretação. O menor dos problemas seria ter o comportamento da versão antiga do Hibernate. Algo mais grave é quando há funcionalidades novas no JAR mais recente: a interface `Session` do Hibernate antigo seria carregada do `common/lib`, e quando sua aplicação **APP1** invocar um método que só existe na versão nova, um `NoSuchMethodError` será lançado! Isso causa uma enorme dor de cabeça, pois o desenvolvedor fica confuso ao ver que o código compilou perfeitamente, mas na hora de executar o Java diz que aquele método não existe.

`NoSuchMethodError` **é um forte indicador de que o código foi compilado esperando uma versão diferente de uma biblioteca que a encontrada em tempo de execução**.

Há ainda um problema mais grave: imagine que a aplicação **APP1** utiliza uma classe do Hibernate que só existe na versão nova. O classloader que verifica o `common/lib` não achará essa classe, e então ela será corretamente carregada do `WEB-INF/lib`. Porém, essa classe precisa de algumas classes básicas do Hibernate, como a `Session`, e esta será carregada da versão antiga, do `common/lib`, fazendo com que o Hibernate seja carregado parte de uma versão, parte de outra; resultando em efeitos colaterais inesperados e desastrosos! Essa confusão é similar ao **DLL Hell**, e muitos conhecem esses problemas como **ClassLoader hell**.

Isto também pode resultar em um `ClassCastException` curioso e de difícil discerni-

mento: se uma referência a um objeto do tipo `AlgumTipo` for passado como argumento para um método que recebe um objeto também do tipo `AlgumTipo`, porém essa classe foi carregada por um classloader diferente, a exceção será lançada. O desenvolvedor fica confuso ao ver uma `ClassCastException` sendo lançada em uma invocação de método em que nem mesmo há um *casting*.

Isso acontece porque, como vimos, em tempo de execução a identidade de uma classe não é só seu nome completo (o *Fully Qualified Name*) mas também conta o classloader que a carregou. Assim uma classe com mesmo nome, mas carregada por classloaders diferentes, é considerada diferente. Isso é chamado de **runtime identity** e é muito importante para permitir que containers tenham mais de uma versão da mesma classe na memória (como o exemplo das `Session`s do Hibernate). Porém, isso pode causar dores de cabeça com os `ClassCastException` aparentemente inexplicáveis.

Exatamente pelo mesmo motivo, jogar JARs na variável de ambiente `CLASSPATH` é uma idéia perigosa. **Mantenha as dependências das suas aplicações bem separadas, evitando utilizar diretórios comuns para tal**. Em outras palavras, o ideal é que cada aplicação seja auto suficiente, com todas as dependências de que necessita.

O padrão *singleton*, conforme implementado na literatura, é facilmente quebrado através do uso de diversos classloaders, já que a mesma classe pode ser carregada mais de uma vez, acabando com o senso comum de que "*atributos estáticos existem apenas um por virtual machine*". Aliás, o uso do singleton para centralizar o acesso a algumas informações é considerado por muitos um antipattern, como veremos no capítulo posterior [50, 55].

Criando seu ClassLoader

Podemos instanciar um `URLClassLoader` que é responsável por ler classes a partir de um conjunto dado de URLs. Considere que nossa classe `DAO` está dentro do diretório `bin`:

```
ClassLoader loader = new URLClassLoader(new URL[]{new URL("file:bin/")}, null);
Class<?> clazz = loader.loadClass("br.com.arquiteturajava.DAO");
```

Repare que passamos `null` como segundo argumento do construtor de `URLClassLoader`. Isso indica que o *parent* classloader dele será o *bootstrap* diretamente. Não há como criar um classloader sem *parent*: no mínimo é o bootstrap classloader.

Se também carregarmos a classe através do `Class.forName` e compararmos o resultado:

```
ClassLoader loader = new URLClassLoader(new URL[]{new URL("file:bin/")}, null);
Class<?> clazz = loader.loadClass("br.com.arquiteturajava.DAO");

Class<?> outraClazz = Class.forName("br.com.arquiteturajava.DAO");

System.out.println(clazz.getClassLoader());
System.out.println(outraClazz.getClassLoader());
```

```
System.out.println(clazz == outraClazz);
```

O `Class.forName()` carrega a classe usando o classloader que carregou o código sendo executado; no caso o *Application ClassLoader* se estivermos no método `main`. E o resultado com a JVM da Sun (Hotspot):

```
java.net.URLClassLoader@19821f
sun.misc.Launcher$AppClassLoader@11b86e7
false
```

Tentando executar esse mesmo código para uma classe da biblioteca padrão, digamos `java.net.Socket` ou `java.lang.String`, teremos um resultado diferente. Elas serão carregadas em ambos os casos pelo *bootstrap* classloader (sendo simbolizado pelo `null`), pois não há como evitar que esse classloader seja consultado pela questão de segurança já discutida.

Se removermos o `null` do construtor, nosso `URLClassLoader` terá como *parent* o classloader que carregou o código sendo executado (*Application ClassLoader*, neste caso), fazendo com que o carregamento da classe seja delegado para esse, antes de ser tentado pelo nosso próprio classloadder. Caso o diretório `bin` esteja no classpath, o *Application ClassLoader* consegue carregar a classe `DAO` e o nosso recém criado classloadder não vai tentar carregar a classe:

```
ClassLoader loader = new URLClassLoader(new URL[]{new URL("file:bin/")});
```

Dessa vez o resultado é:

```
sun.misc.Launcher$AppClassLoader@11b86e7
sun.misc.Launcher$AppClassLoader@11b86e7
true
```

O *parent* classloader vai ser sempre consultado antes do classloader de hierarquia "mais baixa" tentar carregar uma determinada classe. Lembre-se que esse é o motivo pelo qual jogar os JARs na variável de ambiente `CLASSPATH` pode acabar escondendo versões diferentes da mesma classe que estejam ao alcance de classloader mais "baixos" na hierarquia.

Endorsed jars

Esse funcionamento hierárquico causa problemas também com a biblioteca padrão: a cada versão nova do Java, algumas APIs, antes externas e opcionais, são incorporadas. Esse foi o caso, por exemplo, das APIS de parseamento de XML do Apache (Xerces e Xalan). Caso elas fossem incluídas dentro da biblioteca padrão, qualquer outra pessoa que necessitasse delas em uma versão diferente (tanto mais atual quanto antiga) teria sempre a versão do `rt.jar` carregada!

Para contornar esse problema, a Sun colocou as classes que seriam do pacote `org.apache` para dentro de `com.sun.org.apache` [40]. Sem dúvida uma maneira deselegante, mas que evitou o problema de versionamento.

Esse problema tornou-se ainda mais frequente. No Java 6, a API de mapeamento de XML, a JAXB 2.0, entrou para a biblioteca padrão. Quem necessita usar a versão 1.0, ou quem sabe uma futura versão 3.0, terá problemas: a versão 2.0 sempre será carregada. O JBoss 4.2 necessita da API do JAXB 1.0 para a sua implementação de Web Services, e sofre então com essa inclusão do Java 6 [51]. Aliás, o JBoss possui uma longa e interessante história de manipulação dos classloaders, sendo o primeiro a EJB container a oferecer *hot deployment*, e para isso teve de contornar uma série de restrições e até mesmo bugs da JVM [7].

O mesmo ocorre com o interpretador de Javascript que entrou junto com a Scripting API do Java 6 na JVM da Sun: o Mozilla Rhino. Quem precisa utilizar de versões diferentes do Rhino acaba padecendo dos problemas aqui citados [32].

Nesses dois casos, para contornar esse problema utilizamos do recurso de **endorsed JARs**. Através deste recurso a JVM deixa que, por linha de comando (`-Djava.endorsed.dirs=`), você especifique diretórios que devem ter prioridade na procura de classes antes de que o diretório `ext` seja consultado. É uma forma do administrador do sistema dizer que confia (endossa) determinados jars. Alguns servidores de aplicação já possuem uma pasta onde você pode jogar os JARs que você considera endossados. Vale notar o cuidado que deve ser tomado ao confiar em um JAR e endossá-lo, colocando-o no topo da hierarquia dos classloaders (mas sempre abaixo do bootstrap).

OutOfMemoryError no redeploy de aplicações

Vimos que a JVM da Sun (e outras) armazenam as classes carregadas na memória no espaço chamado **PermGen**. É um espaço de memória contado fora do heap onde apenas a JVM aloca seus objetos e classes carregadas. Muitas vezes, temos problemas com `OutOfMemoryError` acusando que esse espaço se esgotou, e isso em geral está ligado a fazer o carregamento de muitas classes na memória.

Em particular, existe um problema muito conhecido que se manifesta ao fazer alguns redeploys de aplicações no container. E isso está ligado diretamente a como é feito o carregamento pelos classloaders.

Toda classe carregada tem uma referência para o classloader que a carregou. Mas todo classloader tem uma referência para todas as classes que ele carregou (isso existe para que o classloader possa devolver a mesma classe sempre caso alguém peça mais de uma vez, e dessa forma carregando-a uma única vez). Esse relacionamento bidirecional entre `Class` e `ClassLoader` faz com que as classes só sejam coletadas pelo garbage collector junto com o classloader inteiro e todas as outras classes associadas. Logo, o único jeito de fazer um objeto `Class` ser coletado é liberando as referências para todas as classes daquele classloader (e também o objeto classloader em questão).

Quando fazemos o redeploy de uma aplicação, o próprio container libera as referências dele para as classes da aplicação antiga e para o WebApp ClassLoader. Isso deveria ser suficiente para que o contexto fosse inteiramente liberado da memória. Na prática isso quase nunca acontece pois sempre sobram referências para alguma classe do contexto, o que segura o classloader **inteiro** em memória.

O problema acontece quando alguma classe de outro classloader acima do WebApplication ClassLoader segura uma referência para alguma classe da aplicação. E há

vários culpados conhecidos hoje por terem esse tipo de comportamento. Um dos mais clássicos é o `DriverManager` do JDBC. A maioria das implementações do DriverManager guarda referências para as classes dos drivers registrados. Mas a classe `DriverManager` é carregada pelo bootstrap e o Driver em geral está dentro da aplicação, no *WEB-INF/lib*. O carregamento de um único driver JDBC portanto seria capaz de segurar na memória o contexto inteiro da aplicação.

Existem vários outros exemplos de bibliotecas e classes que causam memory leaks parecidos [3, 59]. No caso do JDBC, a solução é fazer um *context listener* que invoca o método `deregisterDriver` no `DriverManager`. Mas há várias situações não tão simples de se contornar e que envolvem versões específicas de bibliotecas com bugs de memory leaks. Na prática, é quase impossível uma aplicação Java conseguir se livrar desses leaks de classloaders, por isso acabamos tendo os `OutOfMemoryError` frequentes nos redeploys.

Aumentar a memória do *PermGen* não é uma solução, só vai adiar o momento em que a memória vai acabar. O ideal seria não existirem esses vazamentos de memórias, mas como isso não é possível, uma boa solução é evitar os hotdeploys. **Reiniciar o servidor e a JVM a cada novo deploy ajuda a evitar que o PermGen se esgote com o tempo.**