passando o número desejado. Muitas pessoas perguntam-se por que o JIT não compila todos os métodos, rodando tudo de forma nativa e otimizada. É fácil testar este caso, passando 1 na opção anterior, e logo você notará uma perda de performance em muitas aplicações pequenas ou com pouca lógica repetida. Neste caso, o custo de compilação e otimização é muito alto. Em outros, principalmente em aplicações mais longas, é possível até ter algum ganho, mas logo percebe-se que os números calculados por padrão pela JVM acabam sendo valores bem razoáveis para a maioria das aplicações. Outro teste interessante é usar a opção `–Xint`, que desabilita completamente o JIT e faz a VM rodar apenas em modo interpretado. Em aplicações um pouco mais longas, logo se percebe uma imensa perda de performance.

De modo geral, o que se percebe é que não ter um JIT é uma péssima ideia, quase tão ruim quanto compilar todos os métodos existentes. Na prática, o melhor acaba sendo o comportamento padrão do JIT de compilar apenas os pontos quentes. Nossa aplicação, porém, deve, sim, estar preparada para usufruir o melhor do JIT. Em geral, isso significa as boas práticas de design com vários pequenos métodos sendo reaproveitados por todo o código.

Pode ser necessário monitorar a execução de uma aplicação, como quando há a necessidade de encontrar gargalos específicos de performance e vazamentos de memória. A Oracle disponibiliza diversas ferramentas para facilitar o *profiling*, como `jmap`, `jstat` e `jconsole`.[17] Plugins para IDEs também existem em abundância, como o TPTP do Eclipse. Dentre as ferramentas pagas, destacam-se o TestKit e o JProfiler. Todos eles permitem a um desenvolvedor extrair informações de uma aplicação rodando na plataforma Java para entender melhor como ela está funcionando em produção, até mesmo aquelas escritas em outras linguagens, mas interpretadas na VM.

2.4. Carregamento de classes e classloader hell

É frequente diversas aplicações Web serem implantadas em um mesmo Servlet Container, em uma mesma máquina virtual, e também que muitas delas utilizem bibliotecas e frameworks em comum. Imagine duas aplicações que utilizam a mesma versão do Hibernate: onde devemos colocar os JARs?

Muitos sugeriram colocá-los em algum diretório `common/lib` do Servlet Container; outros disseram para jogar diretamente na variável de ambiente `CLASSPATH`. Esses eram conselhos frequentes, em especial quando era raro ter mais

de uma aplicação Java implantada em um mesmo servidor. Para entender os graves problemas que podem surgir nesta situação, é preciso primeiro compreender como é o funcionamento do carregador de classes da JVM.

Classloader é o objeto responsável pelo carregamento de classes Java para a memória a partir de um array de bytes que contém seus bytecodes.[18] É representado pela classe abstrata `java.lang.ClassLoader`, e existem diversas implementações concretas, como a `java.net.URLClassLoader`, responsável por carregar classes buscando em determinadas URLs.

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 por 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 diferentes de uma mesma biblioteca; podemos ter o Hibernate 3.6 em uma aplicação Web e o Hibernate 3.3 em outra. Para isso, dois classloaders diferentes são necessários para carregar as classes das duas diferentes aplicações. Como um classloader carrega uma classe uma única vez, se neste 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 diferentes), prevalecendo 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 2.4).

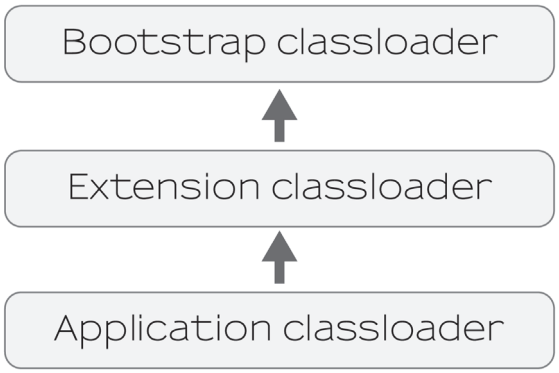

Figura 2.4 – Hierarquia de classloaders.

A figura mostra os três classloaders que a JVM sempre utilizará por padrão. Antes de um classloader iniciar um carregamento, ele pergunta ao seu pai (*parent classloader*) se não consegue carregar (ou já carregou) a classe em questão. Se o *parent classloader* conseguir, ele será responsável por esse carregamento. 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*, 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 troia.

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 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 da variável de ambiente, quanto da linha de comando (passado como argumento por `-cp`). Representado pela classe interna `sun.misc.Launcher$AppClassLoader`, tem como pai o *Extension classloader*.

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

As aplicações podem criar novos classloaders e incrementar essa hierarquia (Figura 2.5). Os containers 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 precisa de um classloader específico para cada aplicação (contexto) im-

plantada. Esse **WebApplication classloader** é responsável pelo carregamento das classes do *WEB-INF/classes* e do *WEB-INF/lib* em aplicações Web, por exemplo.

Conhecendo essa hierarquia, podemos perceber o problema quando desenvolvedores desavisados jogam JARs em diretórios compartilhados por todas as aplicações de um determinado servidor, como o caso do diretório `common/lib` do Apache Tomcat em versões muito antigas, ou utilizando a variável de ambiente `CLASSPATH`.

Considerando que temos duas aplicações, `APP-nova` e `APP-antiga`, que usam a mesma versão do Hibernate 3.3, elas funcionarão normalmente mesmo com os jars do Hibernate referenciados na variável `CLASSPATH`. No momento que a `APP-nova` precisar utilizar a versão 3.6, como proceder? Jogar a versão mais nova no `WEB-INF/lib` da `APP-nova` não resolve o problema, pois a hierarquia de classloaders faria com que a `org.hibernate.Session` e todas as outras classes do Hibernate sejam carregadas dos jars definidos na `CLASSPATH`, possivelmente gerando erros inesperados em tempo de execução (Figura 2.5).

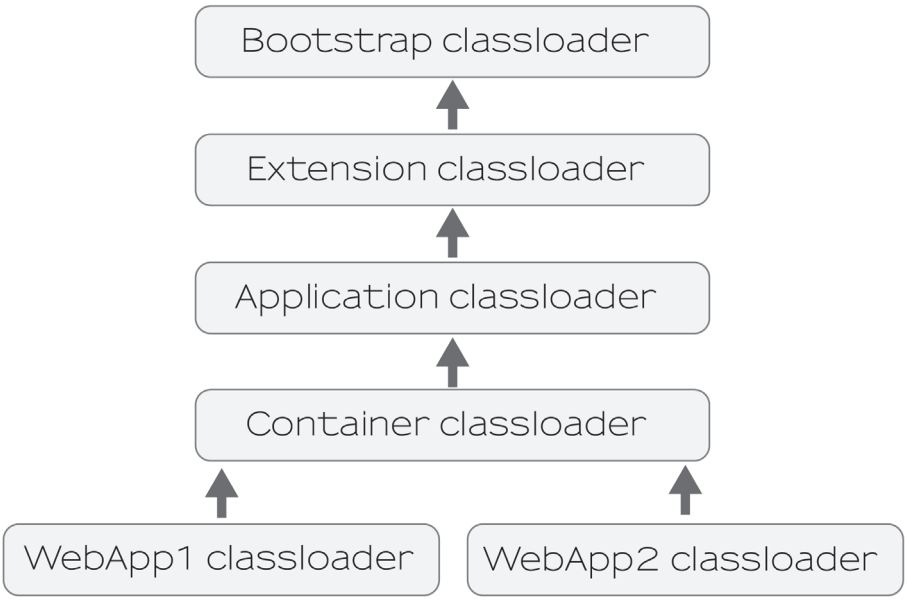

Figura 2.5 – Hierarquia usual de classloaders no container.

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. Mais grave é quando há métodos novos no JAR mais recente; a interface `Session` do Hibernate antigo seria carregada, e

quando sua aplicação `APP-nova` invocar um método que só existe na versão nova, `NoSuchMethodError` será lançado! Isso causa uma enorme dor de cabeça, pois o desenvolvedor fica confuso ao ver que o código compilou perfeitamente, mas durante a execução a JVM indica 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**, provavelmente por causa de os jars estarem espalhados e compartilhados.

O comportamento da aplicação pode ser mais errático; imagine que a aplicação `APP-nova` utiliza diretamente uma classe do Hibernate que só existe na versão mais nova, como a `TypeResolver`. O classloader que verifica a variável `CLASSPATH` não a encontrará, e então ela será corretamente carregada do `WEB-INF/lib` pelo classloader específico do contexto. Porém, a `TypeResolver` do Hibernate 3.6 referencia outras classes básicas do Hibernate, e estas serão carregadas da versão antiga, o Hibernate encontrado na `CLASSPATH`, 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, frequentemente chamado de **Classloader hell** na plataforma Java.

Em algumas configurações de classloaders diferentes, que carregam classes de diretórios em comum, pode aparecer um `ClassCastException` curioso e de difícil discernimento. Se uma referência a um objeto do tipo `Produto` for passada como argumento para um método que recebe um objeto também deste tipo, mas esta classe tiver sido carregada por um classloader diferente, a exceção será lançada. O desenvolvedor ficará confuso ao ver uma `ClassCastException` 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 apenas seu *fully qualified name*, mas também o classloader que a carregou. Assim, uma classe com mesmo nome, mas carregada por classloaders diferentes, é outra. Isto é chamado de **runtime identity** e muito importante para permitir que containers tenham mais de uma versão da mesma classe na memória (como o exemplo das `Sessions` de diferentes versões do Hibernate).[19]

Muitos containers, incentivados pela própria especificação de Servlets, aplicam um modelo de classloaders invertidos para minimizar os problemas. Ao invés de deixar o WebApp classloader filho do Container classloader, ele é feito filho direto do Bootstrap classloader mas com uma referência simples para o Container classloader. A arquitetura tradicional dificulta que uma aplicação sobrescreva algum componente do Container classloader, mas esse modelo invertido (padrão no Tomcat, por exemplo) permite esse cenário. A ordem de resolução das classe

passa a ser: primeiro o Bootstrap, depois as classes do a aplicação e depois as compartilhadas do Container.

Criando seu ClassLoader

Podemos instanciar a `URLClassLoader` para carregar classes a partir de um conjunto dado de URLs. Considere que a classe `DAO` está dentro do diretório `bin`:

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

Repare que passamos `null` como segundo argumento do construtor de `URLClassLoader`. Isto indica que seu *parent* classloader será o *bootstrap* diretamente.

Se também carregarmos a classe através do `Class.forName` e compararmos a referência das duas classes:

```
URL[] bin = {new URL("file:bin/")};
ClassLoader loader = new URLClassLoader(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 foi responsável pela classe do código executado; no caso, o *Application classloader* se estivermos no método `main`. E o resultado com a JVM da Oracle/Sun é:

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

Executando este 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 (representado pelo `null`), pois não há como evitar que esse classloader seja consultado em razão da questão de segurança já discutida.

Ao remover o `null` do construtor, o `URLClassLoader` terá como *parent* o classloader que carregou o código sendo executado (o *Application classloader*), fazendo com que o carregamento da classe seja delegado para este, antes de ser tentado pelo nosso `URLClassLoader`. Caso o diretório `bin` esteja no classpath, o *Application classloader* consegue carregar a classe `DAO` e o nosso recém-criado classloader não será o responsável pelo carregamento:

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

Desta vez, o resultado é:

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

O *parent* classloader será sempre consultado antes de o classloader de hierarquia "mais baixa" tentar carregar uma determinada classe. Lembre-se de que este 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 classloaders 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. Este 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 outro projeto que necessitasse delas em uma versão diferente (tanto mais atual quanto mais antiga) teria sempre a versão do `rt.jar` carregada.

Para contornar o problema, a Sun colocou as classes que seriam do pacote `org.apache` para dentro de `com.sun.org.apache`.[20] Sem dúvida uma maneira deselegante, mas que evitou o problema de versionamento.

Este 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 ainda uma futura versão 3.0, terá problemas; a versão 2.0 sempre terá prioridade no carregamento. O JBoss 4.2 necessita da API do JAXB 1.0 para sua implementação de Web Services, e sofre então com essa inclusão do Java 6.[21] Aliás, o JBoss possui uma longa e interessante história de manipulação dos class-

loaders, sendo o primeiro EJB container a oferecer *hot deployment*, e para isto teve de contornar uma série de restrições e até mesmo bugs da JVM.[22]

Quando o interpretador de Javascript Rhino entrou junto com a Scripting API no Java 6 na JVM da Sun, passamos a ter o mesmo problema quando é necessário utilizar suas versões diferentes.[23]

Nesses dois casos, para contornar o problema, utilizamos o recurso de **endorsed JARs**. Através da linha de comando (`-Djava.endorsed.dirs=`), você especifica diretórios que devem ter prioridade na procura de classes antes que o diretório `ext` do Java SE seja consultado. É uma forma de o administrador do sistema dizer que confia em (endossa) determinados JARs. Alguns servidores de aplicação possuem um diretório especial onde você pode jogar os JARs a serem endossados. Vale notar o cuidado que deve ser tomado ao confiar em um JAR, colocando-o no topo da hierarquia dos classloaders.

OutOfMemoryError no redeploy de aplicações

Vimos que algumas JVMs armazenam as classes carregadas na memória no espaço chamado **PermGen**. É comum aparecerem problemas com `OutOfMemoryError`, acusando que este espaço se esgotou, dado um número muito grande de classes carregadas. Em particular, isso ocorre facilmente depois de alguns hot deploys em um container.

Toda classe carregada tem uma referência para o seu classloader, assim como todo classloader referencia todas as classes carregadas por ele. Isso para que possa devolver sempre a mesma classe no caso de outra invocação subsequente para o mesmo *full qualified name*, agindo como uma factory que cacheia suas instanciações. Esse relacionamento bidirecional entre `Class` e `ClassLoader` faz com que os objetos `Class` só sejam coletados pelo garbage collector junto com o classloader inteiro e todas as outras classes associadas. Logo, a única maneira de um objeto `Class` ser coletado é se todas as referências para todas as classes do seu classloader forem liberadas também.

Ao realizar o hot deploy de uma aplicação, o próprio container libera as referências das classes antigas, possibilitando a coleta do classloader do contexto. Isso deveria ser suficiente para que o contexto fosse inteiramente liberado da memória, mas, na prática, outro classloader acima daquele da WebApplication segura referências para classes da aplicação. E há várias bibliotecas que assim se comportam, como o próprio `DriverManager` do JDBC (*Java Database Conectivity*). Este guarda referências para as classes dos drivers registrados, mas elas são carregadas pelo bootstrap, e o driver, em geral, está dentro da aplicação,

no *WEB-INF/lib*. O carregamento de um único driver JDBC é capaz de segurar na memória o contexto inteiro de uma aplicação que não é mais necessária.

Existem vários outros exemplos de bibliotecas e classes que causam memory leaks parecidos.[24,25] No caso do JDBC, a solução é fazer um *context listener* que invoca o método `deregisterDriver` no `DriverManager`, mas há situações não tão simples de se contornar, que envolvem versões específicas de bibliotecas com seus próprios tipos de leaks de referências a classes que já não são mais utilizadas. Na prática, é quase impossível uma aplicação Java conseguir evitar esses leaks de classloaders, por isso em algum momento surgem os `OutOfMemoryError` frequentes nos hot deploys, sendo mais um motivo para evitá-los em produção.