20

Java et Oracle 10*g*

Dans ce chapitre :

- l'environnement Java d'Oracle 10*g* ;
- les objets Java stockés dans Oracle 10*g* ;
- les procédures Java stockées.

Ce chapitre aborde les fonctionnalités Java intégrées dans le serveur de base de données Oracle 10*g*. Il ne s'agit en aucun cas d'un guide d'apprentissage du langage Java dont la connaissance est cependant nécessaire pour une bonne compréhension.

Java a d'abord été conçu pour développer des interfaces graphiques de petite taille facilement téléchargeables sur Internet. Il offre en effet des possibilités intéressantes pour améliorer l'ergonomie et les fonctionnalités des sites Internet, notamment l'interactivité avec les utilisateurs. Néanmoins, Java n'est pas uniquement destiné au développement d'applications client : il a rendu possible la conception d'applications distribuées complexes et multi-plates-formes. Grâce à sa portabilité et à sa nature orientée objet, Java est maintenant à la base de très nombreuses applications métier déployées sur des serveurs.

Oracle 10*g* fournit une plate-forme de mise en œuvre destinée aux applications réseau (intranet ou Internet), qui intègre une machine virtuelle Java au sein du serveur de base de données. Elle constitue un complément du moteur PL/SQL. Avec Java et Oracle 10*g*, il devient possible de réaliser des procédures stockées écrites en Java.

Après avoir décrit comment fonctionne la machine virtuelle Java Oracle et la configuration de l'environnement Java, nous décrirons le déploiement d'objets Java et de procédures stockées Java dans Oracle 10*g*. À la fin de ce chapitre, la machine virtuelle Java d'Oracle 10*g* et ses nombreuses fonctionnalités n'auront plus de secret pour vous.

L'environnement Java d'Oracle 10*g*

Oracle a développé sa propre machine virtuelle Java, compatible avec les spécifications de SUN Microsystems. Dans la version Oracle 10*g*, l'environnement Java est conforme aux spécifications Java 2. Par rapport aux machines virtuelles destinées aux postes client, la JVM d'Oracle 10*g* est capable de hautes performances dans le domaine transactionnel ; elle peut s'adapter automatiquement à un nombre grandissant d'utilisateurs et communiquer facilement avec la base de données Oracle, notamment avec le moteur PL/SQL d'Oracle 10*g*.

L'environnement Java d'Oracle 10*g* se caractérise par :

- une machine virtuelle Java performante ;
- une intégration étroite à la base de données.

Les nombreuses applications développées en PL/SQL n'ont pas besoin d'être réécrites en Java. Grâce à l'intégration de la JVM dans la base de données, une véritable interopérabilité est possible entre le moteur PL/SQL, le moteur SQL et la JVM d'Oracle 10*g*. La base de données peut exécuter des déclencheurs (triggers), des fonctions, des procédures et des méthodes de type objet écrits en Java, tout comme elle peut exécuter du code PL/SQL.

*Évolutions d'Oracle 10*g

La machine virtuelle Java interne d'Oracle 10*g* est en version 1.4.

Les technologies Java J2EE ne sont plus supportées dans la version Oracle 10*g*.

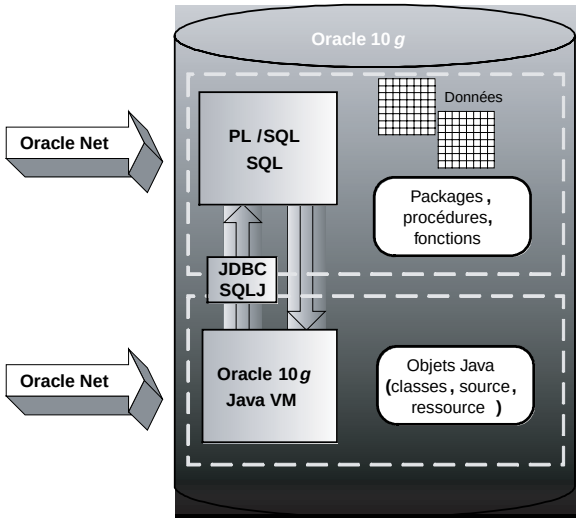
Ces technologies regroupent :

- le conteneur pour Enterprise Java Beans (EJB) ;
- le moteur Java Server Pages (JSP) ;
- le moteur Oracle Servlet Engine (OSE) ;
- l'environnement CORBA intégré dans la base Oracle.

Dans la gamme des produits Oracle, les standards J2EE sont désormais disponibles avec Oracle 10*g* Application Server (10*g* AS), le serveur d'application Oracle. Le module OC4J de 10*g* AS constitue le moteur de déploiement des applications J2EE.

Nous venons de présenter l'environnement Java intégré dans Oracle 10*g ;* apprenons maintenant à l'exploiter.

Figure 20-1
Java dans Oracle 10g



Configuration Java dans Oracle 10*g*

Le fonctionnement de la JVM d'Oracle dans une instance Oracle 10*g* est déterminé par les paramètres de configuration suivants :

- SHARED_POOL_SIZE;
- JAVA_POOL_SIZE ;
- JAVA_SOFT_SESSIONSPACE_LIMIT ;
- JAVA_MAX_SESSIONSPACE_SIZE.

Les paramètres SHARED_POOL_SIZE et JAVA_POOL_SIZE agissent directement sur les performances de la JVM d'Oracle 10*g*.

Les deux autres paramètres fixent des seuils de consommation mémoire Java : le premier est un seuil d'avertissement, le second un seuil d'erreur.

Ces quatre paramètres sont spécifiés dans le fichier d'initialisation de l'instance (local ou serveur).

Paramètre SHARED_POOL_SIZE

La zone mémoire dont la taille est spécifiée par le paramètre SHARED_POOL_SIZE sert, dans le cadre de la JVM, pour le chargement des classes Java par l'utilitaire *loadjava* par exemple. En effet le chargement d'objets Java dans la base de données

nécessite des traitements lourds en terme de ressource mémoire. Le mécanisme de chargement et de résolution des dépendances entre classes Java est décrit plus loin dans ce chapitre.

La valeur minimum conseillée pour ce paramètre est environ 50 Mo.

Il faut noter que la zone mémoire du pool partagé n'est pas dédiée uniquement à la JVM d'Oracle 10*g*.

Paramètre JAVA_POOL_SIZE

Le paramètre JAVA_POOL_SIZE spécifie la taille de la zone mémoire utilisée pour le stockage des objets Java pendant leur utilisation.

La taille minimum conseillée est 20 Mo.

Depuis la version Oracle 10*g* (release 1 et 2), le paramètre JAVA_POOL_SIZE est ajusté automatiquement par les processus de la base.

Paramètre JAVA_SOFT_SESSIONSPACE_LIMIT

Ce paramètre indique le seuil de mémoire Java consommée par une session Oracle au-delà duquel un message d'avertissement est enregistré dans un fichier de trace Oracle.

Paramètre JAVA_SOFT_SESSIONSPACE_SIZE

Ce paramètre indique la quantité de mémoire Java maximale que peut consommer une session Oracle. Si la mémoire Java utilisée par la session dépasse cette valeur, Oracle renvoie au programme appelant une erreur de dépassement de mémoire (« out of memory »).

Par défaut, la valeur de ce paramètre est fixée à 4 Go. S'il existe un risque que la consommation mémoire des classes Java n'est pas maîtrisée, il est conseillé de fixer une valeur moins élevée à ce paramètre.

Sécurité Java dans Oracle 10*g*

Oracle 10*g* a des fonctionnalités de gestion de la sécurité pour tous les objets de la base de données. Elle gère la sécurité sur les objets des schémas par type d'accès (lecture, exécution, modification, suppression, etc.). La sécurité de l'environnement Java 2 repose initialement sur l'octroi de permissions sur les ressources système.

Dans ce livre, nous n'expliquerons pas en détail les permissions Java 2.

Cette section décrit l'implémentation et la gestion de la sécurité Java 2 dans Oracle 10*g*.

Gérer la sécurité Java 2 dans Oracle 10g

Dans les JVM externes, les règles de sécurité Java 2 sont décrites dans un fichier (Security policy file).

Dans Oracle 10*g*, les règles de sécurité Java 2 sont enregistrées dans une table (Java policy table).

La consultation de la table des permissions Java 2 se fait à travers des vues du dictionnaire Oracle 10*g*. Ces vues décrivent la configuration des permissions Java 2 paramétrées dans la base de données :

- DBA_JAVA_POLICY ;
- USER_JAVA_POLICY.

Dans ces vues on trouve les informations suivantes :

Colonne	Information
Kind	GRANT ou RESTRICT (octroi ou restriction)
Grantee	schéma ou rôle auquel la permission est accordée
Type_schema	schéma propriétaire de la classe de la permission
Type_name	nom de la classe de la permission
Name	nom de la permission
Action	actions associées à la permission (read, write par exemple)
Enabled	permission active ou inactive
Seq	clé identifiant la permission

Un enregistrement dans la table des permissions Java 2 ne signifie pas forcément un octroi de privilèges.
Le type de la permission est GRANT ou RESTRICT.
Un enregistrement de la table peut donc indiquer une interdiction.

Comme pour les privilèges standard Oracle (système et objets), les permissions Java 2 peuvent être accordées, soit à des schémas, soit à des rôles.

Sécurité Oracle 10g préconfigurée

Dans Oracle 10*g*, l'octroi de permissions Java 2 est facilité par l'existence de rôles Oracle standard déjà configurés à la création de la base de données :

- JAVA_ADMIN ;
- JAVA_DEPLOY ;
- JAVASYSPRIV ;
- JAVAUSERPRIV ;

- JAVADEBUGPRIV.

Le rôle JAVA_ADMIN accorde des droits privilégiés sur la gestion de la sécurité Java (PolicyTablePermission). Par défaut, il est accordé au rôle DBA.

Le rôle JAVA_DEPLOY accorde des droits pour le déploiement des librairies de classes Java compilées nativement.

Le rôle JAVASYSPRIV accorde des droits privilégiés comme :

- l'administration des classes de la JVM d'Oracle 10*g* ;
- l'utilisation de sockets (client et serveur) ;
- l'accès complet aux fichiers du système d'exploitation.

Le schéma PUBLIC reçoit par défaut les droits de chargement et d'exécution des classes. Il ne peut en revanche pas modifier les classes des packages Java système telles que oracle.aurora, java et jdbc.

Le rôle JAVAUSERPRIV accorde des droits d'exécution supplémentaires (par rapport à PUBLIC), comme :

- l'utilisation de sockets (client) ;
- l'accès aux fichiers en lecture seule ;
- la gestion des threads.

Le rôle JAVADEBUGPRIV accorde des droits pour utiliser les fonctions de débogage de la JVM d'Oracle 10*g*.

*Gérer les permissions Java 2 dans Oracle 10*g

La gestion des permissions Java 2 est effectuée via une API PL/SQL ou Java.

Le package PL/SQL s'appelle DBMS_JAVA.

Il comprend les méthodes pour accorder ou supprimer des privilèges Java 2 dans la JVM Oracle 10*g*.

Types de permissions

Aux types de permissions Java 2 communs, Oracle 10*g* ajoute deux types propres à la JVM interne de la base de données :

- oracle.aurora.rdbms.security.PolicyTablePermission;
- oracle.aurora.security.JserverPermission.

Le premier permet d'accorder les droits sur la table des permissions Java 2.

Le second permet d'accorder des droits sur l'utilisation de la JVM interne d'Oracle 10*g*.

Types de permission dans Oracle 10*g*
java.util.PropertyPermission
java.io.SerializablePermission
java.io.FilePermission
java.net.NetPermission
java.net.SocketPermission
java.lang.RuntimePermission
java.lang.reflect.ReflectPermission
java.security.SecurityPermission
oracle.aurora.rdbms.security.PolicyTablePermission
oracle.aurora.security.JServerPermission

Accorder des permissions

Pour accorder des permissions, la méthode PL/SQL de l'API est grant_permission.

Les informations à fournir à cette méthode sont :

Paramètre	Information
Grantee	schéma ou rôle auquel la permission est accordée
Permission_type	classe correspondant à la permission
Permission_name	attribut de la permission (dépend de la permission)
Permission_action	actions associées à cette permission
Key	(optionnel) identifiant de la permission dans la table

À partir d'un programme Java, il faut appeler la méthode Grant de la classe oracle.aurora.rdbms.security.PolicyTableManager.

Restreindre des permissions

Pour restreindre des permissions, la méthode PL/SQL de l'API est restrict_permission.

Les informations à fournir à cette méthode sont :

Paramètre	Information
Grantee	schéma ou rôle auquel la permission est accordée
Permission_type	classe correspondant à la permission
Permission_name	attribut de la permission (dépend de la permission)
Permission_action	actions associées à cette permission
Key	(optionnel) identifiant de la permission dans la table

À partir d'un programme Java, il faut appeler la méthode Restrict de la classe oracle.aurora.rdbms.security.PolicyTableManager.

Accorder des permissions du type PolicyTablePermission

Précédemment, nous avons décrit JAVA_ADMIN comme un rôle ayant les privilèges nécessaires à la modification de la table des permissions Java d'Oracle 10*g*. Un utilisateur ayant reçu ce rôle peut modifier n'importe quelle permission.

Oracle 10*g* permet d'accorder des droits de modifications sur la table des permissions Java à d'autres utilisateurs que ceux possédant le rôle JAVA_ADMIN. Chaque utilisateur peut recevoir le droit de modifier un ou plusieurs types de permissions dans la table des permissions.

Par exemple, on pourra autoriser le schéma SCOTT à mettre à jour uniquement la sécurité sur les accès aux fichiers (java.io.FilePermission).

La méthode PL/SQL s'appelle grant_policy_permission.

Les informations à fournir à cette méthode sont :

Paramètre	Information
Grantee	schéma ou rôle auquel la permission est accordée
Permission_schema	schéma propriétaire de la classe de la permission
Permission_type	classe du type de permission sur laquelle on accorde la permission
Permission_name	attribut de la permission (dépend de la permission)
Key	(optionnel) identifiant de la permission dans la table

À partir d'un programme Java, il faut appeler la méthode grantPolicyPermission de la classe oracle.aurora.rdbms.security.PolicyTableManager.

Les objets Java dans Oracle 10*g*

La version d'Oracle 10*g* est fournie avec une machine virtuelle Java pour utiliser et exécuter des classes Java depuis le moteur de la base de données. À l'image des autres types d'objets pouvant être créés et stockés dans la base de données (tables, index, séquences, procédures, packages, etc.), Oracle 10*g* permet d'y stocker les objets Java. La JVM interne d'Oracle 10*g* n'exécute pas des classes rangées parmi les fichiers du système d'exploitation : elle accède aux classes chargées et stockées dans des schémas de la base de données.

Objets Java stockés dans la base de données

Les différents types d'objets Java sont :

- des fichiers source Java (.java, .sqlj) ;
- des fichiers de classe Java (.class) ;

- des fichiers de ressource Java (.ser, .properties, etc.) ;
- des archives compressées de fichiers Java (.jar, .zip).

Dans la base de données, ces objets sont rangés dans trois catégories de fichiers :

- les sources Java (fichiers source Java) ;
- les classes Java (fichiers de classe Java) ;
- les ressources Java.

Un schéma est propriétaire de chaque objet Java. Ainsi, la même classe peut être chargée plusieurs fois dans des schémas séparés mais dans un schéma ne peut exister qu'un exemplaire de l'objet Java.

De plus, il est possible de créer des synonymes privés ou publics sur les objets Java stockés dans la base.

Pendant le chargement des fichiers d'archives, comprimés, il est possible de les conserver en tant que fichiers d'archives ou de charger séparément les fichiers qu'ils contiennent. Dans le deuxième cas, les fichiers contenus dans les archives sont extraits et triés en source, classe et ressource selon leur extension.

Lancer la suppression d'une archive compressée dans la base de données revient à supprimer les objets Java de la base de données correspondant aux fichiers présents dans cette archive. La suppression concerne uniquement les objets inclus dans le schéma sous lequel la suppression est déclenchée.

Déployer des programmes Java dans la base Oracle

Une différence majeure, par rapport aux programmes Java qui s'exécutent en dehors de la base de données, réside dans la manière de les déployer et de les lancer.

Pour déployer des classes Java dans un système d'exploitation, il faut compiler puis copier des fichiers d'archives, compressées, ou des répertoires référencés dans la variable d'environnement CLASSPATH. Pour lancer une application Java, sa classe principale doit comporter une méthode statique Main.

Dans la base de données Oracle 10*g*, la notion de programme Java n'existe pas. L'accès aux classes Java se résume à l'appel des méthodes statiques et publiques des classes. Il n'y a pas d'équivalent au programme java (ou java.exe) des machines virtuelles Java externes.

Développer, déployer et exécuter une procédure stockée Java comprend quatre étapes, à savoir :

- écrire le code Java ;
- charger le code Java dans la base de données ;
- publier les procédures Java dans le dictionnaires de données Oracle ;
- utiliser les procédures stockées Java.

Gestion des références entre objets Java

Autre différence significative entre une machine virtuelle Java externe et la JVM interne de la base : la localisation des packages, des classes et des fichiers de ressource.

Dans les JVM externes, la portée de la recherche est déterminée par le paramètre CLASSPATH. Dans la JVM de la base de données, c'est un module de résolution (resolver) qui prend en charge les références entre classes, packages et fichiers de ressources.

Pour une JVM externe, la variable d'environnement CLASSPATH indique les chemins d'accès disques des classes, des packages ou des archives, compressées, contenant les classes et packages. Quand une classe est appelée, la JVM externe recherche cette classe dans les emplacements spécifiés par la variable CLASSPATH.

La notion de répertoire (conteneur de fichiers) ou de fichier n'existe pas dans la base Oracle 10*g*. À la place, on trouve des schémas (propriétaires d'objets) et des objets. De même, Oracle 10*g* ne permet pas de spécifier des variables d'environnement internes dans la base. Le paramètre CLASSPATH est donc remplacé par le resolver. La JVM d'Oracle 10*g* utilise son resolver pour trouver les classes Java référencées par les autres classes.

Si les classes références appartiennent à des schémas Oracle différents du schéma dans lequel les classes vont être chargées, il faut explicitement indiquer le nom des autres schémas au resolver.

La base de données maintient automatiquement à jour dans son dictionnaire les dépendances entre les classes Java chargées dans chaque schéma.

Ainsi, comme une JVM externe sait manipuler des packages ou des classes isolées, stockées dans des arborescences différentes, la base de données peut manipuler sans erreur des packages ou des classes isolées dont les schémas propriétaires sont différents.

Contexte d'exécution d'une classe Java

À l'instar des applications clientes Oracle exécutant des ordres SQL ou appelant du code PL/SQL, une session Oracle est nécessaire pour utiliser les classes Java déployées dans la base de données Oracle.

Implicitement, toutes les manipulations de données sur les objets de la base de données sont effectuées dans le cadre de la session Oracle ouverte pour l'exécution du code Java.

Chaque session Oracle possède son propre contexte d'exécution en termes de gestion de la mémoire, de gestion des instances de classes et des variables statiques. Par exemple, le mécanisme de nettoyage (garbage collector) de la machine virtuelle Java traite les instances de classes par session.

Par analogie avec une machine virtuelle Java externe à la base, le contexte d'exécution d'une classe Oracle est comparable à un process de JVM. La base de données gère autant de contextes Java qu'il existe de sessions faisant appel à des programmes Java.

Néanmoins, pour ne pas pénaliser les performances du système hébergeant la base de données, la base de données Oracle 10*g* n'alloue pas à chaque session toutes les ressources nécessaires à un process d'une JVM traditionnelle. La JVM d'Oracle 10*g* est développée pour que chaque session ne consomme qu'un faible surcoût de mémoire. La première session invoquant une classe Java provoque l'octroi des ressources pour démarrer la machine virtuelle Java de la base de données. Puis cet espace mémoire commun est partagé pour chaque session. Seule la mémoire nécessaire à l'exécution des méthodes des classes Java est allouée à chaque session.

Compiler une classe Java

Avant d'utiliser une classe Java déployée ou non dans la base de données Oracle 10*g*, le code source de cette classe doit être transformé en bytecodes : ce traitement est réalisé par la compilation Java.

Oracle 10*g* offre plusieurs alternatives pour la compilation des classes Java stockées dans la base :

- compilation préalable au chargement dans la base ;
- compilation au moment du chargement ;
- compilation au moment de l'exécution.

Pour compiler une classe Java avant son chargement dans la base Oracle 10*g*, l'outil standard javac peut être invoqué dans le système d'exploitation. Ensuite, au lieu d'indiquer le nom du fichier source Java, il faut charger directement la classe compilée dans la base de données.

La compilation au moment du chargement est réalisée par l'outil loadjava décrit dans la section suivante.

L'outil loadjava peut également charger un fichier source Java sans le compiler.

Dans la suite du chapitre, nous présenterons la compilation native des classes Java qui a pour but d'accélérer les temps d'exécution des classes Java déployées dans la base Oracle 10*g*.

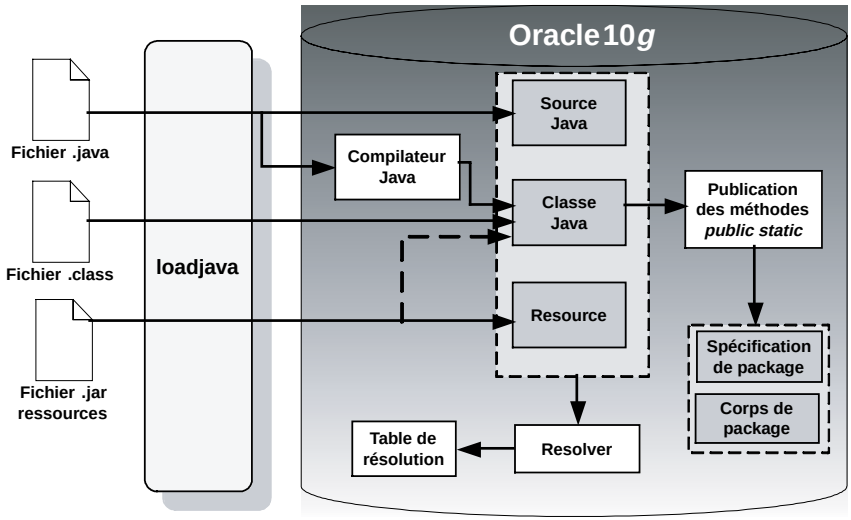
Charger des objets Java avec l'utilitaire loadjava

Pour que la base de données Oracle 10*g* soit capable d'exécuter une méthode Java, il faut charger la classe ou les classes correspondantes dans la base de données.

À cet effet, il convient de recourir à l'utilitaire loadjava fourni avec Oracle 10*g* et qui réalise les opérations suivantes :

- chargement des fichiers source Java (ou des classes Java compilées) et des fichiers de ressources ;
- chargement effectif des classes Java dans la base de données en tant qu'objets de schémas.

Figure 20-2
Principe de l'utilitaire loadjava



L'utilitaire loadjava charge chacun des fichiers qui lui sont spécifiés et exécute les traitements opportuns. Les fichiers source Java sont compilés par le compilateur Java d'Oracle 10*g*, puis enregistrés comme objets compilés Java. Les fichiers de classes Java sont enregistrés directement comme objets Java. Les fichiers de ressources (comme les fichiers de propriétés) sont stockés comme objets ressources Java. Une table d'options contient des informations supplémentaires sur les fichiers. Si le fichier à charger est une archive JAR ou ZIP, alors loadjava charge séparément chaque fichier de l'archive.

Au moment de l'exécution d'une application Java, les classes sont chargées dynamiquement. Pour remplacer la variable d'environnement CLASSPATH, la machine virtuelle Java d'Oracle 10*g* utilise un resolver (l'outil resolver). La résolution d'une classe consiste à définir la liste des objets nécessaires pour l'exécuter. Le resolver enregistre les dépendances pour chaque classe chargée dans la JVM d'Oracle 10*g*. Le resolver marque ensuite une classe comme valide si toutes les classes dont elle dépend sont valides. Dès qu'une classe devient invalide (rechargement par exemple), les classes qui appellent cette classe deviennent invalides. Les classes invalides ne sont plus exécutables ou appelables jusqu'à ce que le *r*esolver les qualifie de valides. L'action du resolver a lieu, soit au moment du chargement de la classe, soit ultérieurement au moment de l'exécution d'une classe. Par défaut, le resolver recherche les classes dans le schéma courant de l'utilisateur connecté et dans le schéma PUBLIC. Le schéma SYS contient tous les packages Java standard comme java.lang et les packages système Java d'Oracle 10*g* comme oracle.aurora. Des synonymes publics sont automatiquement créés pour les classes standard Java et les classes système Java d'Oracle 10*g*. Si les classes, ou packages, appelées ont été chargées dans un autre schéma Oracle, il faut spécifier le paramètre resolver de la commande loadjava.

Oracle 10*g* inclut la génération automatique de classes Java « vides » à l'utilitaire loadjava. Les classes Java générées automatiquement remplacent les classes inexistantes

dans la base de données mais obligatoires pour la compilation des classes Java chargées. L'analyse des classes manquantes n'est appliquée que pour les fichiers Java chargés sous forme de classes. L'analyse n'est pas effectuée pour les fichiers source Java car la détection des références manquantes dans les fichiers sources Java est compliquée.

L'option associée à cette fonctionnalité de l'utilitaire loadjava est genmissing.

La syntaxe de loadjava est la suivante :

```
loadjava <options> <nom des classes>
<nom des classes> : liste des classes à charger
```

Voici quelques options de l'outil loadjava :

Option	Description
user ou u	chaîne de connexion à la base de données JDBC OCI : user/password[@tns] JDBC Thin : user/password@serveur:port:SID
Thin	utilisation du pilote JDBC Thin
oci ou oci8	utilisation du pilote JDBC OCI
grant userA, userB	accorde le privilège EXECUTE sur les classes en cours de chargement aux comptes Oracle userA et userB ; équivalent à *grant execute* pour les procédures PL/SQL
resolve	demande la résolution pendant le chargement de la classe
resolver [(« package ou classe » « schema ») (« package ou classe » « schema »)...]	paramètre de la résolution de classes
synonym	crée un synonyme public pour les classes à charger ; la résolution de la classe sera possible depuis n'importe quel schéma
genmissing	génération des classes manquantes (l'implémentation des méthodes est vide)
genmissingjar	En plus des actions de genmissing, création d'un jar qui contient les classes générées
publish « package »	crée ou remplace le package « package » pour écrire des méthodes wrapper pour les méthodes Java (des classes à charger) éligibles à la publication sous forme de procédures stockées, utilisable seulement avec les fiches '.class'. La méthode Java doit appartenir à une classe *public*, doit être déclarée *public* et *static*, et ne doit comporter que des types de paramètres convertibles directement dans des types de valeurs PL/SQL.
synonym	crée des synonymes publics pour les classes chargées (nécessite le privilège CREATE PUBLIC SYNONYM)
schema	indique dans quel schéma les objets Java sont à charger (nécessite les privilèges CREATE ANY PROCEDURE, CREATE ANY TABLE)

Option	Description
tableschema	Schéma dans lequel seront stockées les tables utilisées par loadjava. Par défaut, elles sont stockées dans le schema où sont chargées les classes.
verbose	Affiche des informations au chargement des fichiers. Recommandé pour trouver les classes manquantes plus tôt !
order	Résout les classes dans l'ordre croissant
noverify	Ne vérifie pas le bytecode
force	Force le chargement de toutes les classes. Normalement, les classes identiques aux classes déjà chargées ne sont pas rechargées.
definer	Spécifie que les méthodes des classes chargées seront exécutées avec les privilèges de créateur, et non de simple utilisateur. Par défaut, les méthodes s'exécutent avec les privilèges de leur utilisateur. Des créateurs différents peuvent avoir des privilèges différents, et une application peut avoir de nombreuses classes, donc assurez-vous que les méthodes d'une classe donnée s'exécutent avec seulement les privilèges requis.

Ci-dessous un exemple pour spécifier de prendre en compte toutes les classes et packages Java du schéma USER1 et toutes les classes et packages du schéma PUBLIC :

```
-resolver '((* USER1) (* PUBLIC))'
```

Chargeons le fichier Salarie.class dans le schéma SCOTT, avec la commande shell suivante :

```
loadjava -oci8 -user scott/tiger Salarie.class
```

La classe Salarie possède une méthode public static augmenteSalaire qui prend comme paramètre le nom d'un employé et un taux d'augmentation de salaire. Elle augmente le salaire de l'employé du taux indiqué.

Nous allons charger la classe Salarie dans la base de données et publier automatiquement la méthode augmenteSalaire en tant que procédure stockée. Le principe des procédures stockées Java est présenté plus loin dans ce chapitre.

Le paramètre publish de loadjava permet de spécifier le nom du package PL/SQL à créer pour contenir une procédure d'appel de la méthode Java augmenteSalaire.

La signature de la méthode augmenteSalaire est la suivante :

```
public static void augmenteSalaire(String p_salarie, float p_taux)
```

Il n'est pas nécessaire d'indiquer à loadjava quelle méthode publier en tant que procédure stockée : l'utilitaire détecte automatiquement les méthodes qui satisfont aux critères pour être publiées en tant que procédures stockées. La méthode Java doit appartenir à une classe public, doit être déclarée public et static et ne doit comporter que des types de paramètres convertibles directement dans des types de valeurs PL/SQL.

```
loadjava -publish pSalarie -oci8 -user scott/tiger Salarie.class
```

Cette commande va charger la classe Java Salarie en tant que classe dans la base de données puis créer le package pSalarie avec une méthode augmenteSalaire.

Elle est équivalente à :

- charger la classe Java dans la base de données :

```
loadjava -oci8 -user scott/tiger Salarie.class
```

- créer le package pSalarie avec SQL*Plus :

```
>sqlplus scott/tiger
SQL> CREATE OR REPLACE PACKAGE P_SALARIE AUTHID CURRENT_USER AS
 PROCEDURE AUGMENTE_SALAIRE(arg0 VARCHAR,arg1 NUMBER) AS LANGUAGE JAVA NAME
 ➥'entreprise.Salarie.augmenteSalaire(java.lang.String ,float )';
END;
/
SQL>
```

Supprimer des objets Java avec l'utilitaire dropjava

L'utilitaire dropjava est utilisé pour supprimer une classe, des packages ou des fichiers de ressources Java chargés dans la base de données Oracle 10*g*. Si la classe a été chargée à partir d'une archive JAR, les fichiers contenus dans cette archive ne sont pas forcément supprimés : seules les classes qui dépendent de la classe supprimée sont signalées comme invalides. Les classes qui héritent de la classe supprimée le sont également.

La syntaxe de dropjava est similaire à celle de loadjava :

```
dropjava -user <utilisateur>/<mot de passe>[@<connexion>] <options> <nom des classes>
<utilisateur>/<mot de passe> : nom et mot passe pour se connecter à la base de données;
<connexion> : chaîne de connexion à la base; voir la syntaxe de loadjava
<nom des classes> : liste des classes à supprimer dans le schéma de l'utilisateur
➥qui ouvre la connexion
```

Par exemple, pour supprimer les classes contenues dans le fichier archive unearchive.jar qui ont été chargées dans le schéma SCOTT :

```
dropjava -verbose -oci8 -user scott/tiger unearchive.jar
```

Charger des objets Java en PL/SQL

La méthode PL/SQL loadjava du package dbms_java joue un rôle identique à l'utilitaire loadjava qui est lancé à l'extérieur de la base.

Elle charge des objets Java dans la base de données, à la différence près qu'on peut l'exécuter depuis un bloc PL/SQL.

Par exemple :

```
begin
dbms_java.loadjava('e:\digora\deploy\unearchive.jar','-resolve');
end;
```

Supprimer des objets Java en PL/SQL

La méthode PL/SQL dropjava du package dbms_java joue un rôle identique à l'utilitaire dropjava qui est lancé à l'extérieur de la base.

Elle supprime des objets Java dans la base de données, à la différence près qu'on peut l'exécuter depuis un bloc PL/SQL.

```
begin
dbms_java.dropjava('e:\digora\deploy\unearchive.jar');
end;
```

Options de compilation par défaut

Si aucune option de compilation n'est indiquée pendant le chargement d'une classe, la JVM prend des valeurs par défaut. Les valeurs par défaut du compilateur sont enregistrées dans la table JAVA$OPTIONS du schéma où sont chargées les classes. Si la table JAVA$OPTIONS n'existe pas, elle est automatiquement créée.

Le package PL/SQL DBMS_JAVA fournit trois méthodes pour modifier les valeurs des options de compilation par défaut pour chaque package.

Méthode	Paramètres	Description
set_compiler_option	name, option, value	affecte la valeur d'une option
get_compiler_option	name, option	récupère la valeur d'une option sous forme d'une chaîne de caractères
reset_compiler_option	name, option	réaffecte la valeur initiale d'une option

Le paramètre name spécifie pour quels packages s'applique l'option par défaut. Si le paramètre name est une chaîne vide, alors l'option s'applique pour tous les packages.

Le paramètre option indique le nom de l'option de configuration. C'est une valeur parmi les trois possibles : online, debug ou encoding.

Le paramètre value contient la valeur de l'option de configuration.

Consulter les objets Java stockés dans Oracle 10g

Comme pour tous les objets créés et stockés dans Oracle 10*g*, des vues d'administration sont fournies en standard pour consulter les objets Java stockés dans la base.

Les informations sur les objets Java (fichiers source, classes Java, fichiers de ressource) sont accessibles dans les vues DBA_OBJECTS et USER_OBJECTS.

La vue USER_OBJECTS contient les informations sur les objets Java mis en œuvre dans la base de données, dans le schéma de l'utilisateur courant. La vue DBA_OBJECTS élargit la visibilité des objets à tous les schémas de la base de données. La vue DBA_OBJECTS n'est accessible que par des utilisateurs Oracle DBA.

Par défaut, ces vues affichent le nom des objets Java sous une forme codifiée. Pour rendre lisible le nom des objets, il faut utiliser la fonction dbms_java.longname qui transforme

le nom d'objet codifié en un nom d'objet textuel lisible. La fonction réciproque de dbms_java.longname est dbms_java.shortname qui renvoie le nom d'une classe sous forme codifiée.

Les types d'objets renvoyés dans ces vues pour les objets Java sont :

- JAVA SOURCE pour les codes source Java ;
- JAVA CLASS pour les classes Java ;
- JAVA RESOURCE pour les autres objets Java.

Le statut des objets Java est, soit VALID, soit INVALID. Si le statut d'un objet Java est INVALID, cela signifie qu'il n'a pas pu être compilé, soit pour des erreurs de syntaxe, soit pour des erreurs de dépendances non trouvées.

La commande suivante extrait de cette vue le nom des objets Java, leur type (classe Java, fichiers de ressource Java) et leur statut (valide ou invalide).

```
select replace(dbms_java.longname(object_name),'/','.') as
➥" OBJET JAVA ",object_type,status from user_objects where object_type like 'JAVA%';
```

Par exemple, la requête précédente produit un affichage de type :

```
OBJET JAVA                    OBJECT_TYPE        STATUS
----------------------------- ------------------ -------
Entreprise                    JAVA CLASS         VALID
META-INF.MANIFEST.MF          JAVA RESOURCE      VALID
connections.properties        JAVA RESOURCE      VALID
3 rows selected.
```

Cet exemple montre que le schéma de la base possède :

- une classe Java intitulée ENTREPRISE ;
- deux fichiers de ressources, *meta-inf.manifest.mf* et *connections.properties*.

L'interface shell ojvmjava

La base de données Oracle 10*g* n'est plus livrée avec cet utilitaire pour accéder à l'espace des objets Java d'un schéma sous forme d'un environnement shell.

Le script ojvmjava.bat offre notamment la possibilité d'exécuter des applications Java, c'est-à-dire d'invoquer la méthode static main des classes Java.

> Le script ojvmjava.bat était livré avec la version Oracle9*i*.
>
> Pour le recréer en environnement MS Windows :
>
> 1) recopier le fichier loadjava.bat sous le nom ojvmjava.bat ;
>
> 2) ouvrir le fichier ojvmjava.bat ;
>
> 3) remplacer la chaîne oracle.aurora.server.tools.loadjava.LoadJavaMain par la chaîne : *oracle.aurora.server.tools.shell.ShellClient.*

La liste des commandes disponibles est limitée :

Commandes	Description
Echo	affiche un message sur la sortie standard
Java	programme Java similaire au programme Java livré dans le JDK par exemple : Java Uneclasse invoque la méthode main de la classe Uneclasse
Exit	quitte l'environnement ojvmjava
Help	Aide
Version	version de l'utilitaire
Whoami	nom de l'utilisateur connecté

Compilation native des classes Java dans Oracle 10*g*

La portabilité des applications Java repose sur le principe suivant. À la différence des applications écrites en C par exemple, les compilateurs Java ne créent pas des programmes directement exécutables sur une plate-forme particulière. Le code source Java est transformé en bytecodes qui sont indépendants de toute plate-forme d'exécution. Ces bytecodes sont traduits par une machine virtuelle Java au moment de leur exécution. La machine virtuelle Java est propre à chaque environnement d'exécution (système d'exploitation et matériel). En terme de performance, la version interprétée d'une application Java est plus lente que la version compilée de la même application.

Pour pallier cet inconvénient, les machines virtuelles Java peuvent intégrer des compilateurs natifs. Ces compilateurs natifs traduisent les bytecodes Java dans du code natif à la façon d'un compilateur C par exemple. Les performances s'en trouvent améliorées car le mécanisme d'interprétation des bytecodes est supprimé.

Deux stratégies sont possibles pour la compilation native des bytecodes Java :

- la compilation dite JIT (Just-In-Time) ;
- la compilation statique.

Avec la compilation JIT, la machine virtuelle Java détecte les blocs d'instructions exécutés le plus fréquemment et transforme ces instructions en code natif. Cette compilation a lieu pendant l'exécution de l'application Java et aucun programme exécutable n'est créé.

La compilation statique consiste à traduire les bytecodes Java en code C puis à compiler ce code C avant l'exécution de l'application. L'application Java ainsi compilée peut exploiter au mieux les capacités de la plate-forme d'exécution.

À l'installation des fichiers binaires de la base de données Oracle 10*g*, les librairies Java standard fournies avec Oracle 10*g* sont compilées nativement.

Tous les programmes Java peuvent profiter de cette fonctionnalité de la machine virtuelle d'Oracle 10*g*.

Configuration de la compilation native Oracle 10g

Plusieurs étapes de configuration sont nécessaires avant de pouvoir utiliser la compilation native d'Oracle 10*g* sur le serveur hébergeant la base de données.

La configuration concerne, d'une part l'environnement logiciel du système d'exploitation, d'autre part l'octroi de privilèges particuliers au sein de la base de données.

Outil de génération des librairies dynamiques DLL

Pendant la compilation native Oracle 10*g*, l'outil ncomp génère des librairies dynamiques Windows (DLL). Oracle 10*g* n'est pas livré avec un outil permettant la génération de librairies dynamiques DLL, donc un produit tiers doit être installé. Le logiciel Visual C++ de Microsoft est un exemple de produit qui peut être utilisé pour la compilation native d'Oracle 10*g*.

Une partie des paramètres de la compilation native d'Oracle 10*g* est spécifiée dans le fichier Settings_os.properties. Ce fichier se trouve dans le répertoire du module Java Accelerator d'Oracle 10*g* : %ORACLE_HOME%\javavm\jahome.

Le fichier des paramètres de compilation est à modifier en fonction du produit choisi pour compiler des fichiers source C et générer des DLL. Le contenu du fichier Settings_os.properties est préconfiguré pour s'exécuter avec Microsoft Visual C++.

Un paramètre reste cependant à adapter en fonction du répertoire d'installation de Visual C++ sur le serveur : visual.c.home. La valeur de ce paramètre est le chemin complet du produit Visual C++ sur le serveur.

Par exemple :

```
visual.c.home = c:/Program Files/MSVisualStudio60/VC
```

Environnement Java du système d'exploitation

Pour utiliser les utilitaires Oracle 10g de compilation native (ncomp, deployne et statusnc), un environnement JDK 1.4 doit être présent sur le serveur.

La base de données Oracle 10*g* est installée avec un environnement JDK 1.4 dans le répertoire %ORACLE_HOME%\jdk.

Le chemin d'accès complet au JDK 1.4 doit être affecté à la variable d'environnement JAVA_HOME.

Par exemple :

```
JAVA_HOME = c:\oracle\ora10g\jdk
```

La variable d'environnement JAVA_HOME peut être positionnée :

- directement dans les paramètres système de Windows ;
- dans les scripts deployne.bat, ncomp.bat et statusnc.bat ;
- dans le script appelant les utilitaires Oracle 10*g*.

Configuration dans la base de données

Avant de pouvoir utiliser la compilation native des classes Java, l'utilisateur Oracle 10*g* propriétaire des classes Java doit recevoir des privilèges spécifiques. En effet, pour la compilation native d'Oracle 10*g*, il est nécessaire d'accéder avec tous les droits aux fichiers du système d'exploitation (génération de fichiers temporaires, des DLL, déplacement de fichiers, etc.) et de lancer des programmes externes (compilation, génération des DLL).

Ces privilèges sont accordés via des rôles standard Oracle :

- JAVA_DEPLOY ;
- JAVASYSPRIV.

L'octroi est réalisé avec un client SQL, par exemple SQL*Plus, en étant connecté DBA.

```
c:\digora\ncomp> sqlplus system/manager
SQL>grant java_deploy,javasyspriv to scott ;
Autorisation de privilèges (GRANT) acceptée.
```

Utilitaires Oracle 10*g* pour la compilation native

Oracle 10*g* est livré avec trois utilitaires pour utiliser la compilation native des classes Java chargées dans la base de données :

- ncomp, pour la compilation et le déploiement des classes Java compilées nativement dans la base de données ;
- deployнc, pour le déploiement des classes Java compilées nativement dans la base de données ;
- statusnc, pour la vérification du statut des classes Java compilées nativement déployées dans la base de données.

Les utilitaires sont lancés par des scripts shell Windows (ncomp.bat, deployнc.bat, statusnc.bat) stockés dans le répertoire %ORACLE_HOME%\bin.

Lancer la compilation native avec l'utilitaire Oracle ncomp

La compilation native d'Oracle 10*g* est réalisée par l'utilitaire ncomp. Cet utilitaire opère de la façon suivante :

- traduction des bytecodes en fichiers source C génériques ;
- compilation des fichiers source C ;
- génération de librairies dynamiques DLL sur le serveur hébergeant la base de données ;
- installation des librairies dynamiques DLL dans l'arborescence Oracle et mise à jour du dictionnaire de la base de données.

L'utilitaire ncomp ne peut compiler que des classes déjà chargées dans la base de données.

Si les classes à compiler nativement ne sont pas déjà chargées dans la base de données, l'utilitaire ncomp peut charger (option load) ces classes Java en invoquant implicitement loadjava. En effet, la ou les classes à compiler nativement sont indiquées à ncomp sous la forme d'un fichier de classe ou d'un fichier archive JAR contenant des classes. L'utilitaire loadjava est présenté en détail dans la partie traitant le chargement des objets Java.

Syntaxe de l'utilitaire ncomp :

```
ncomp <options> <fichier de classes>

<options>
<fichier de classes> : soit une archive compressée JAR ou ZIP contenant les classes
➥à compiler ou soit une classe Java à compiler.
```

Options	Signification
user ou u	chaîne de connexion à la base de données JDBC OCI : user/password[@tns] JDBC Thin : user/password@serveur:port:SID
force	force la compilation native de toutes les classes
load	charge les classes spécifiées dans la base
Thin	utilisation du pilote JDBC Thin
oci8	utilisation du pilote JDBC OCI (par défaut)
noDeploy	uniquement génération d'un fichier archive JAR avec les classes compilées
outputJarFile	nom du fichier archive JAR (avec chemin absolu) généré par *ncomp* et qui contient les fichiers pour le déploiement des classes compilées nativement
lightweightDeployment	déploiement séparé des librairies de la compilation native et des informations de la compilation. Les librairies ne sont pas stockées dans le fichier archive JAR avec les informations de compilation.
projectDir ou d	chemin absolu du projet pour la génération des fichiers source, la compilation et la génération des librairies dynamiques
update	met à jour les librairies et le fichier de déploiement avec les nouvelles classes non encore compilées nativement
verbose	affiche tous les messages de la compilation native

Les noms des options de ncomp sont sensibles à la casse.

Si vous spécifiez une valeur au paramètre projectDir, le répertoire de travail de ncomp est la valeur du paramètre, sinon le répertoire de travail est le répertoire courant d'où est lancé ncomp. Tous les fichiers temporaires de ncomp sont stockés dans le répertoire de travail (fichiers d'include, fichiers source C, fichiers make, fichiers de log, librairies, etc.). Si le paramètre ProjectDir n'est pas spécifié, alors tous les fichiers temporaires de ncomp sont enregistrés dans le répertoire courant.

Si le mode verbose n'est pas explicitement activé avec l'option verbose, les messages de ncomp sont enregistrés dans le fichier de log ncomp.log du répertoire de travail (répertoire courant ou valeur de projectDir).

Avec l'option noDeploy, les fichiers de déploiement et les librairies dynamiques sont générés, mais ces dernières ne sont pas déployées sur le serveur et dans la base de données Oracle 10*g*. Il est conseillé de l'utiliser avec l'option outputJarFile qui spécifie le nom de l'archive JAR contenant tous les fichiers nécessaires au déploiement futur des classes compilées.

Prenons l'exemple d'un fichier JAR monjar.jar contenant une classe monpackage/Maclasse.class.

L'utilitaire ncomp va charger la classe monpackage/Maclasse.class dans Oracle 10*g* puis créer une librairie dynamique DLL sur le serveur hébergeant la base de données.

```
c:\digora\ncomp> ncomp -u scott/tiger -load monjar.jar

#
# this list is produced by query
#   select status, class_name  from jaccelerator$status;
#

NEED_NCOMPING monpackage/Maclass
NEED_NCOMPING monpackage/
# Deployment History, produced by query:
# select timestamp, status, dll_name  from jaccelerator$dlls order by dll_name
Sat Sep 01 16:29:47 GMT 2005 installed /orajox9_xxxxxx_scott_monpackage.dll
```

Les erreurs de compilation sont accessibles, soit en interrogeant la table JACCELERATOR$DLL_ERRORS, soit avec l'utilitaire Oracle statusnc. L'utilitaire statusnc est présenté plus loin dans ce chapitre.

La librairie dynamique qui vient d'être générée par l'utilitaire ncomp se trouve dans le répertoire %ORACLE_HOME%\javavm\admin.

```
c:\digora\ncomp>cd %ORACLE_HOME%\javavm\admin
c:\digora\ncomp>dir orajox10_*_scott_monpackage.dll
orajox10_xxxxxxxx_scott_monpackage.dll
```

Vous remarquerez que les deux dernières parties du nom de la librairie sont le nom du schéma où est chargée la classe Java suivi du nom du package de cette classe Java. Si la classe Java n'appartient à aucun package, la librairie est nommée orajox10_xxxxxxx_scott_UnnamedPackage.dll.

Pour générer uniquement un fichier archive JAR de déploiement (depmonjar.jar) :

```
c:\digora\ncomp> ncomp -oci8 -user scott/tiger -noDeploy -outputJarFile
➥c:\digora\depmonjar.jar -load monjar.jar

#
```

```
# this list is produced by query
#   select status, class_name  from jaccelerator$status;
#

NEED_NCOMPING monpackage/Maclass
NEED_NCOMPING monpackage/
```

Le fichier depmonjar.jar est créé dans le répertoire c:\digora. Les librairies dynamiques des classes Java ne sont pas déployées dans la base de données.

Déployer des librairies dynamiques de classes Java avec l'utilitaire deploync

Précédemment, nous avons décrit l'utilitaire ncomp qui compile nativement des classes Java. Cet utilitaire offre la possibilité de déployer automatiquement ou non les librairies des classes compilées nativement. Si la seconde alternative (option noDeploy) a été choisie, seul un fichier archive JAR est construit : cette archive JAR contient les classes compilées nativement prêtes à être déployées.

Le déploiement des classes compilées nativement doit alors être réalisé par un autre utilitaire d'Oracle 10*g* : deploync.

L'utilitaire deploync prend le fichier archive JAR des classes compilées nativement comme source de données.

```
deploync <options> <fichier archive>

<fichier archive> : une archive compressée JAR contenant les classes compilées nativemen
```

Options	Signification
user ou u	chaîne de connexion à la base de données JDBC OCI : user/password[@tns] JDBC Thin : user/password@serveur:port:SID
Thin	utilisation du pilote JDBC Thin
oci ou oci8	utilisation du pilote JDBC OCI

Pour déployer les classes compilées du package monpackage :

```
c:\digora\ncomp> deploync -oci8 -user scott/tiger -noDeploy -outputJarFile
➥c:\digora\depmonjar.jar -load monjar.jar

# Deployment History, produced by query:
# select timestamp, status, dll_name  from jaccelerator$dlls order by dll_name
Sat Sep 01 16:29:47 GMT 2005 installed /orajox9_xxxxxx_scott_monpackage.dll
```

La librairie dynamique DLL qui vient d'être déployée est copiée dans le répertoire %ORACLE_HOME%\javavm\admin.

Vérifier la compilation native avec l'outil statusnc

L'outil statusnc permet de valider la compilation native d'une ou plusieurs classes Java.

Syntaxe de l'utilitaire statusnc :

```
statusnc -u <utilisateur>/<mot de passe>[@<alias TNS>] <classes>

<utilisateur>/<mot de passe> : connexion à la base de données.
<alias TNS> : alias TNS pour accéder à la base de données.
<fichier de classes> : soit une archive compressée JAR ou ZIP contenant les classes
➥à compiler ou soit une classe Java à compiler.
```

Options	Signification
user ou u	chaîne de connexion à la base de données JDBC OCI : user/password[@tns] JDBC Thin : user/password@serveur:port:SID
Thin	utilisation du pilote JDBC Thin
oci ou oci8	utilisation du pilote JDBC OCI (par défaut)
output	fichier où diriger le flux de sortie de l'utilitaire
projectDir ou d	chemin absolu du projet pour la génération des fichiers source, la compilation et la génération des librairies dynamiques

Si vous spécifiez une valeur au paramètre projectDir, le répertoire de travail de statusnc est la valeur du paramètre, sinon le répertoire de travail est le répertoire courant d'où est lancé statusnc. Tous les fichiers temporaires de statusnc sont stockés dans le répertoire de travail. Si le paramètre projectDir n'est pas spécifié, alors tous les fichiers temporaires de statusnc sont enregistrés dans le répertoire courant.

Si le mode verbose n'est pas explicitement activé avec l'option verbose, les messages de statusnc sont enregistrés dans le fichier de log ncomp.log du répertoire de travail (répertoire courant ou valeur de projectDir).

L'utilitaire statusnc indique le statut des classes Java compilées nativement :

- ALREADY_NCOMPED si la classe a été compilée nativement avec succès ;
- NEED_NCOMPING si la classe est chargée dans la base de données mais n'a pas encore été compilée nativement ;
- INVALID si la classe est invalide.

Reprenons l'exemple de la classe Java monpackage/Maclasse.class.

```
c:\digora\ncomp> statusnc -user scott/tiger monjar.jar

#
# this list is produced by query
#   select status, class_name from jaccelerator$status;
#
```

```
ALREADY_NCOMPED monpackage/Maclass
ALREADY_NCOMPED monpackage/
```

Le fichier exact_class_list.txt créé dans le répertoire de travail liste un état à la fois des classes Java et des librairies des classes Java compilées nativement. Par exemple :

```
# Windows NT x86

# Classes that are already ncomped

# ALREADY_NCOMPED monpackage/Maclass

# Ncomp libraries are ncomped

# ALREADY_NCOMPED monpackage/
```

Les procédures stockées Java dans Oracle 10*g*

Dans les versions antérieures à Oracle8*i* du serveur de base de données Oracle, on recourait uniquement au langage PL/SQL pour développer des applications s'exécutant dans la base Oracle. Le code créé se présentait sous forme de sous-programmes que la base stockait en tant qu'objets : les procédures stockées. Ces objets pouvaient être des procédures, des fonctions ou des déclencheurs de base de données. Avec Oracle 10*g*, les développeurs peuvent écrire la logique métier d'une application en langage PL/SQL ou en langage Java. Comme pour le code PL/SQL, le code Java déployé dans Oracle 10*g* peut être composé de procédures, de fonctions et de déclencheurs. Dans Oracle 10*g*, les procédures stockées Java sont exécutées comme les procédures stockées PL/SQL. Une procédure PL/SQL peut appeler une procédure stockée Java et réciproquement.

Méthodes Java et procédures stockées Java

Déclarer une procédure stockée Java revient à écrire une classe comprenant une méthode static et à déclarer une méthode wrapper PL/SQL pour appeler cette méthode. En effet, le moteur PL/SQL d'Oracle 10*g* ne fournit pas d'API pour instancier directement des classes. Les seuls composants Java que le moteur PL/SQL peut utiliser sont les méthodes public static de classes Java déclarées public. En revanche, rien n'empêche la méthode static appelée en tant que procédure stockée d'instancier elle-même des classes Java.

Pour qu'une méthode Java puisse être publiée en tant que procédure stockée, elle doit respecter une autre règle : les types de tous les paramètres de la méthode Java, y compris le type de retour pour une fonction, doivent être convertibles en type de paramètre PL/SQL.

Écrire une procédure stockée Java

Nous allons créer une procédure stockée qui renvoie le nom d'un employé et met à jour son salaire. Comme nous l'avons expliqué précédemment, une procédure stockée Java est en réalité une méthode static membre d'une classe Java.

Nous définissons donc une classe Java Salarie, avec une méthode getEmpGrade qui renvoie le nom d'un employé, recherché à partir de son identifiant. Cet exemple utilise l'API JDBC pour exécuter les requêtes SQL sur la table EMP du schéma SCOTT.

Le code source montre un exemple de chaîne de connexion pour ouvrir une connexion JDBC depuis une classe Java stockée dans la base de données. Cet exemple utilise le pilote JDBC Oracle disponible dans la machine virtuelle Java d'Oracle 10*g*.

Comme les ordres SQL sont exécutés dans le contexte de la session Oracle ouverte pour lancer la classe, il n'est pas nécessaire de spécifier des paramètres de connexion.

```
import java.sql.*;
import oracle.jdbc.driver.*;
import oracle.*;

class Salarie
{

public static String getEmpGrade (int empid) throws SQLException
  {
String nom = null;
Connection conn = null;
Statement stmt = null;

    try
    {
DriverManager.registerDriver
(new oracle.jdbc.driver.OracleDriver ());
/* la chaîne de connexion utilise le pilote JDBC kprb intégré dans Oracle 10g
il n'est pas nécessaire de spécifier d'utilisateur et de mot de passe car la requête
➥est exécutée dans la session courante */
      conn = DriverManager.getConnection ("jdbc:oracle:kprb:");
      stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery(
        "SELECT JOB FROM EMP WHERE EMPNO = " + empid);
      while (rset.next())
      {
      nom = rset.getString (1);
      }
    }
    finally
    {
stmt.close();
return nom;
    }
  }
```

Déployer la procédure stockée Java

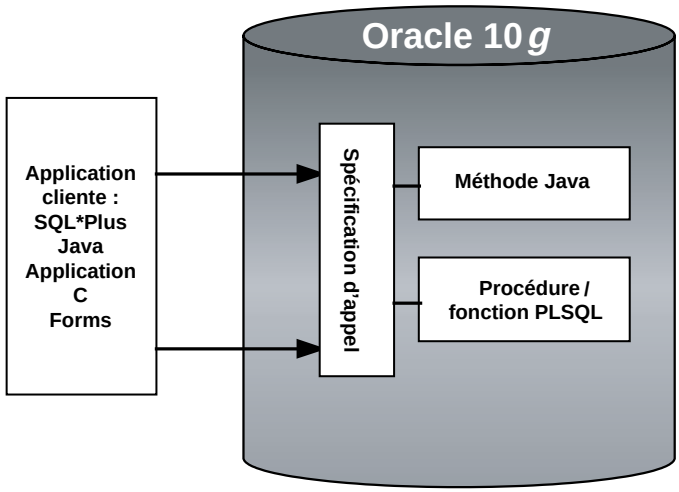
La première étape de déploiement d'une procédure stockée Java consiste à charger la classe Java dans Oracle 10*g*.

Cette opération peut être réalisée par l'utilitaire loadjava.

Publier manuellement une procédure stockée Java

Avant d'appeler des méthodes Java à partir d'instructions SQL, il faut publier ces méthodes dans le dictionnaire de données Oracle. Le schéma suivant montre comment une application cliente accède à une méthode Java dans Oracle 10*g*. Pour appeler la méthode Java, l'application utilise la spécification d'appel associée à cette méthode. Le dictionnaire de données d'Oracle 10*g* recherche alors la méthode Java associée à la spécification d'appel et exécute cette méthode Java.

Figure 20-3
Principe de publication d'une méthode



Au moment du chargement d'une classe Java à l'intérieur de la base, les classes ne sont pas publiées, parce qu'Oracle ne connaît pas les méthodes qui seront accessibles à partir des instructions SQL. La publication d'une méthode Java consiste à définir des spécifications d'appel (*call specifications*). Elles indiquent la correspondance entre :

- les noms des méthodes SQL et les noms des méthodes Java ;
- les types des arguments SQL et Java ;
- les types des valeurs de retour SQL et Java.

En résumé, publier une méthode revient à écrire une signature de sous-programme PL/SQL, analogue à la signature de la méthode écrite en Java.

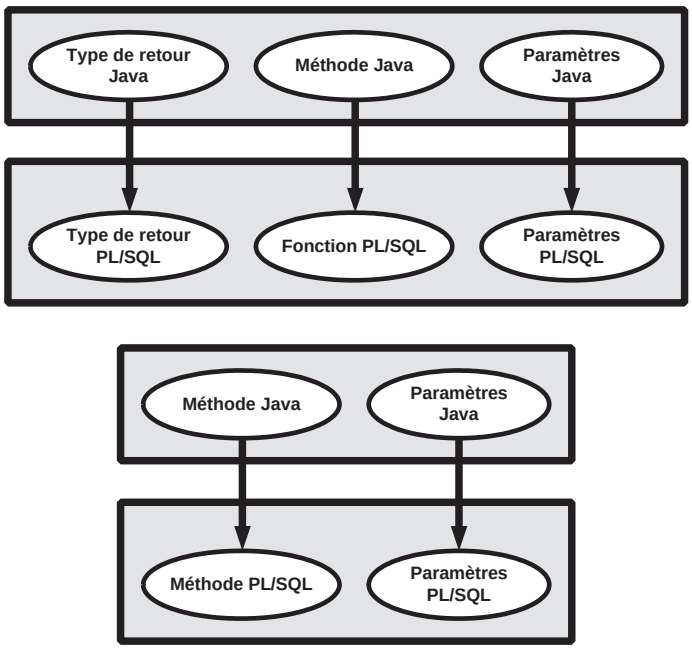


Figure 20-4
Publication des méthodes Java en procédures stockées

Une méthode Java sans type de retour est publiée en tant que procédure PL/SQL. Une méthode Java qui retourne une valeur est publiée en tant que fonction PL/SQL. Le nom des sous-programmes PL/SQL peut être différent de celui des méthodes Java. Concernant le type des paramètres, la classe STRING de Java devient une variable VARCHAR2 dans PL/SQL ; de même, un argument float en Java devient un NUMBER dans PL/SQL. Cette transposition de type de données entre Java et PL/SQL montre la perte de précision des valeurs numériques quand elles sont converties plusieurs fois d'un système Java vers un système PL/SQL. Il ne faut pas négliger cet aspect quand une application travaille à la fois avec des procédures stockées PL/SQL et des méthodes Java.

Définir les spécifications d'appel d'une procédure stockée Java

Une spécification d'appel doit appartenir au même schéma que la procédure stockée équivalente.

Pour écrire une spécification d'appel, il faut créer :

- soit une fonction ou une procédure PL/SQL, dans un package ou non ;
- soit une méthode membre d'un type objet SQL.

Une méthode Java ne pourra être publiée que si elle est déclarée public et static (méthodes de classe). Les méthodes d'instance peuvent être publiées en tant que méthodes membres de type objet SQL.

Définir des spécifications d'appel SQL

SQL*Plus permet de définir manuellement des spécifications d'appel. Pour définir une spécification d'appel d'une procédure il faut utiliser l'instruction SQL standard CREATE OR REPLACE PROCEDURE.

```
CREATE OR REPLACE PROCEDURE <nom procedure>
AUTHID {DEFINER | CURRENT_USER}
AS LANGUAGE JAVA
NAME '<nom de la méthode>(<types des arguments>) <type de retour>' ;
Pour définir une spécification d'appel d'une fonction :
CREATE OR REPLACE FUNCTION <nom procedure>
AUTHID {DEFINER | CURRENT_USER}
AS LANGUAGE JAVA
NAME '<nom de la méthode>(<types des arguments>) <type de retour>' ;
```

La clause AUTHID détermine si une procédure stockée s'exécute avec les droits de son propriétaire ou les droits de l'appelant.

Par exemple, pour appeler la méthode getEmpGrade depuis du code SQL ou PL/SQL, il faut d'abord la publier.

Pour publier la méthode getEmpGrade, nous créons une fonction PL/SQL getEmpGrade qui renvoie une valeur VARCHAR2. La méthode Java getEmpGrade renvoie un objet String qui correspond à une valeur VARCHAR2 dans SQL. L'argument Java dans le type int devient un argument de type NUMBER dans SQL.

```
create or replace function getEmpGrade (empid IN number) return varchar2 is
     language java name 'Salarie.getEmpGrade(int)
     return java.lang.String';
```

Définir des spécifications d'appel dans des packages PL/SQL

Un package PL/SQL est un objet qui regroupe des types, des éléments et des sous-programmes. Il comprend deux parties : la spécification et le corps. La spécification correspond à l'interface pour les applications ; le corps implémente la spécification du package.

Dans l'exemple suivant, la classe Java Deptmanager est constituée de deux méthodes pour ajouter ou supprimer un employé dans la table EMP du schéma SCOTT.

Impossible de trouver les sources pour les classes ci dessous :

```
public class modifEmp {
public static void nvEmp(String empNom) throws SQLException {
 ...
}

public static void supEmp(int empID) throws SQLException {
 ...
}

}
```

Les deux méthodes sont liées car elles agissent sur la même table et font partie d'une même classe Java. Il est donc souhaitable de regrouper leurs spécifications d'appel dans un package PL/SQL.

Nous allons créer le package PL/SQL référençant les deux méthodes Java. Deux nouvelles méthodes PL/SQL sont nécessaires pour appeler ces méthodes depuis PL/SQL : chaque méthode PL/SQL servira à appeler la méthode homologue Java.

Pour définir le package PL/SQL il faut déterminer la spécification et le corps du package.

Spécification du package

La spécification du package déclare les méthodes qu'une application ou une procédure peut appeler à l'extérieur du package. Ici, nous déclarons deux méthodes.

```
CREATE OR REPLACE PACKAGE modifemp AS
PROCEDURE nvemp(empnom VARCHAR2) ;
PROCEDURE supemp(empid NUMBER) ;
END modifemp;
```

Définition du corps du package : les spécifications d'appel des méthodes Java

Le corps du package implémente les méthodes PL/SQL. Dans notre exemple, les méthodes mises en œuvre sont des spécifications d'appel de méthodes Java.

```
CREATE OR REPLACE PACKAGE BODY modifemp AS
PROCEDURE nvemp(empnom VARCHAR2);
AS LANGUAGE JAVA
NAME 'modifEmp.nvEmp(java.lang.String)';
PROCEDURE supemp(empid NUMBER) ;
AS LANGUAGE JAVA
NAME 'modifEmp.supemp(int)';
END modifemp;
```

Finalement, pour tester les procédures Java stockées, l'instruction CALL suffit. L'instruction ci-dessous appelle la procédure supemp du package PL/SQL modifemp que nous venons de créer.

```
CALL modifemp.supemp(1);
```

En réalité, l'appel de la procédure PL/SQL modifemp.supemp se transforme en un appel de la méthode de classe modifEmp.supemp écrite en Java. Les spécifications d'appel cachent l'implémentation Java de la méthode. L'utilisateur de la méthode PL/SQL supemp n'a pas besoin de connaître l'implémentation de la méthode. Il demande juste l'exécution du service Supprimer un employé. Cet exemple révèle la simplicité de remplacement des méthodes PL/SQL par des méthodes Java, et inversement.

Définir des spécifications d'appel dans des types objet

Le langage SQL est compatible avec le modèle objet en proposant les types objet. Un type objet (object type) est un type de données défini par l'utilisateur. Il est composé

d'attributs (les données) et de méthodes (fonctions et procédures) permettant de manipuler ces données.

Un type objet est défini par deux éléments : sa spécification et son corps. La spécification correspond à l'interface pour les applications : elle déclare les attributs et les méthodes de type objet. Le corps implémente la spécification en définissant les traitements des sous-programmes PL/SQL et les spécifications d'appel des méthodes Java. À chaque méthode précisée dans la spécification correspond un sous-programme PL/SQL ou une spécification d'appel dans le corps de type objet. Prenons l'exemple d'une entreprise. Un objet entreprise possède les attributs suivants :

- un nom ;
- un numéro ;
- une adresse.

Définissons une classe Java appelée Entreprise et comportant une méthode getNomEntr qui renvoie le nom d'une entreprise. Cette classe Java possède trois attributs : entrNom, entrId, entrAdresse qui représentent respectivement le nom, le numéro et l'adresse d'une entreprise.

```
import java.sql.*;
import java.io.*;
import oracle.sql.*;
import oracle.jdbc2.*;
import java.math.BigDecimal;

public class Entreprise implements SQLData{
private String entrNom;
private BigDecimal entrId;
private String entrAdresse;

public String getNomEntr() {
return entrNom;
 }

String typeSQL;
public String getSQLTypeName() throws SQLException {
return typeSQL;
};

public void readSQL(SQLInput flux, String nomType) throws SQLException {
typeSQL = nomType;
entrNom = flux.readString();
entrId = flux.readBigDecimal();
entrAdresse = flux.readString();
}
```

```
public void writeSQL(SQLOutput flux) throws SQLException {
flux.writeString(entrNom);
flux.writeBigDecimal(entrId);
flux.writeString(entrAdresse);
}
}
```

Dans la base, nous créons le type objet Entreprise : il possède les mêmes attributs que la classe Java Entreprise et une méthode getNom qui renvoie le nom de la société. Cette méthode est une spécification d'appel pour la méthode Java Entreprise.getNomEntr.

```
CREATE TYPE Entreprise AS OBJECT (
nom VARCHAR2(50),
entId NUMBER(2),
adresse VARCHAR2(50),
MEMBER FUNCTION getNom RETURN VARCHAR2
AS LANGUAGE JAVA
NAME 'Entreprise.getNomEntr() return java.lang.String'
);
```

Pour stocker les instances de type Entreprise, nous créons une table entreprises :

```
CREATE TABLE entreprises OF Entreprise;
```

Nous alimentons la table entreprises :

```
INSERT INTO entreprises VALUES('Cie des Ordinateurs',1, 'PARIS');
INSERT INTO entreprises VALUES('Ets AZERTY',2, 'STRASBOURG');
INSERT INTO entreprises VALUES('La boutique du CDROM',3, 'LYON');
INSERT INTO entreprises VALUES('SuperStore',4, 'TOULOUSE');
```

Il suffit d'écrire un bloc PL/SQL anonyme pour tester le type objet Entreprise. Ce bloc recherche une instance d'Entreprise et appelle sa méthode getNom.

```
declare
entr Entreprise;
id NUMBER(2);
begin
-- affecter à id le numéro de l'entreprise
id := 2;
-- entr pointe vers l'instance d'objet Entreprise
select value(e) into entr from entreprises e where entid=id;
-- affichage du nom de l'entreprise
dbms_output.put_line('Nom de la société : '||entr.getNom);
end;
```

Le bloc PL/SQL précédent doit produire un résultat de type :

```
Nom de la société : Ets AZERTY
```

Appeler une procédure stockée Java

Pour utiliser les procédures stockées Java publiées dans Oracle 10*g*, plusieurs solutions sont possibles :

- appeler des méthodes stockées directement depuis SQL*Plus ;
- appeler des méthodes Java depuis un déclencheur de base de données ;
- appeler des méthodes Java depuis des sous-programmes PL/SQL ;
- appeler des méthodes PL/SQL depuis des méthodes Java ;
- appeler des méthodes Java comme membres d'un type objet.

Appeler une méthode Java depuis SQL*Plus

L'instruction SQL CALL permet d'appeler des méthodes Java publiées dans des packages PL/SQL ou dans des types objet SQL.

Avec SQL*Plus, la syntaxe de la méthode CALL est la suivante :

```
CALL <schéma>.<package>.<procédure>(arguments);
CALL <schéma>.<package>.<fonction>(arguments) INTO :<variable>;
```

Par exemple :

```
CALL modifemp.nv_emp(:empNom);
```

Si le code Java envoie des informations sur la sortie standard, on peut rediriger cette sortie vers le buffer SQL*Plus avec la commande suivante :

```
SET SERVEROUTPUT ON
CALL dbms_java.set_output(2000) ;
```

Appeler une méthode Java depuis un déclencheur (trigger) de base de données

Un déclencheur de base de données est un sous-programme lié à une table ou une vue. Oracle 10*g* déclenche le trigger automatiquement quand une certaine opération SQL de manipulation de données est exécutée sur la table ou la vue. Les triggers PL/SQL sont décrits au chapitre 17, *Programmer avec PL/SQL*.

Exemple : nous allons créer un déclencheur qui permet de savoir si le salaire d'un employé dépasse 10 000 après augmentation.

Tout d'abord, nous définissons une classe Java Triggerjava. Le fichier source triggerJava.sqlj est le suivant :

```
import sqlj.runtime.ref.*;
import java.sql.*;

public class triggerJava {
```

```
public static void testSalaire (int empID, float nvData) throws SQLException {
    try {
      #sql {INSERT INTO controle VALUES (:empID,:nvData)};
      }
    catch (SQLException e) {
      System.err.println(e.getMessage());
      }
    }
}
```

Cette classe comporte la méthode testSalaire qui va enregistrer dans la table Controle les employés dont le salaire dépasse 10 000 après augmentation.

Une requête SQL ne peut pas être exécutée si aucune session n'est ouverte. Comme la procédure s'exécute dans la base de données, la connexion à la base est liée à la session qui a permis de lancer la procédure stockée.

La classe Java est désormais créée. Il reste à définir la spécification d'appel que nous appelons test_salaire.

```
CREATE OR REPLACE PROCEDURE test_salaire (
  empid NUMBER, nvsalaire NUMBER)
AS LANGUAGE JAVA
NAME 'triggerJava.testSalaire(int,float)';
```

test_salaire est une procédure, car elle ne renvoie aucune valeur. La table Controle est créée par la commande SQL suivante :

```
CREATE TABLE controle (
empid   NUMBER,
salaire NUMBER);
```

Enfin, il faut définir le déclencheur, que nous appelons sal_trig. C'est un déclencheur de mise à jour qui est évalué à chaque mise à jour de la colonne sal.

```
CREATE OR REPLACE TRIGGER trigger_salaire
AFTER UPDATE OF sal ON emp
FOR EACH ROW
WHEN (new.sal > 10000)
CALL test_salaire(:new.empno, :new.sal)
```

L'appel de la procédure stockée test_salaire est réalisé par l'instruction CALL.

Quand l'instruction UPDATE affecte une valeur à la colonne SAL d'une ligne de la table EMP, le trigger vérifie la clause WHERE. Si la clause WHERE est valide, le trigger appelle la méthode test_salaire en lui fournissant les arguments nécessaires :

- le numéro de l'employé ;
- le nouveau salaire de cet employé.

test_salaire ajoute alors une nouvelle ligne dans la table Controle. Le trigger est exécuté pour chaque ligne modifiée par l'instruction UPDATE.

Vérifions maintenant que le trigger trigger_salaire et la procédure stockée test_salaire remplissent leur rôle. Le script suivant vide la table Controle puis augmente de 20 % le salaire de chaque employé. La table Controle contient alors tous les employés dont le salaire est supérieur à 10 000 après augmentation.

```
SQL> DELETE * FROM controle;
SQL> UPDATE emp SET sal = sal*1.2;
SQL> SELECT * FROM controle;
EMPID      SALAIRE
---------- ----------
      7499      10200
      7566      11850
      7698      11700
      7782      11220
      7788      11880
      7839      14280
      7844      10080
      7876      22680
      7902      11880
9 rows selected.
```

Appeler des fonctions Java depuis des instructions de manipulation de données SQL

Les instructions SELECT, INSERT, UPDATE et DELETE peuvent appeler les méthodes Java publiées en tant que fonctions.

L'exemple suivant montre comment formater la valeur d'une colonne SELECT avec une fonction de formatage écrite en Java.

Nous définissons la classe AFFEMP, qui comprend la méthode modifEmpNom. La méthode modifEmpNom reçoit le nom de l'employé et de sa fonction et renvoie une chaîne concaténant ces deux éléments. Voici le code source du fichier affEmp.java :

```
public class affEmp {
  public static String modifEmpNom (String empNom, String empFonction) {
  return (new String(empNom+" est un "+empFonction));
  }
}
```

Il faut publier la méthode formatEmp. Nous créons une spécification d'appel de type FUNCTION, car formatEmp renvoie une chaîne de caractères.

```
CREATE OR REPLACE FUNCTION aff_emp (nom VARCHAR2, job VARCHAR2)
RETURN VARCHAR2
AS LANGUAGE JAVA
NAME 'affEmp.modifEmpNom (java.lang.String, java.lang.String)
  return java.lang.String';
```

Nous allons tester la méthode Java affEmp.modifempnom avec une instruction SELECT sur la table EMP du schéma SCOTT. Pour chaque ligne renvoyée par l'instruction

SELECT, la fonction aff_emp est appelée avec indication du nom et de la fonction de l'employé :

- la colonne ename contient le nom de l'employé ;
- la colonne job contient la fonction de l'employé.

La requête SELECT et son résultat sont indiqués ci-dessous, à titre d'exemple :

```
SQL> SELECT aff_emp(ename,job) AS "Nom_Job" FROM emp;
Nom_Job
----------------------------------------
SMITH est un CLERK
ALLEN est un SALESMAN
WARD est un SALESMAN
JONES est un MANAGER
MARTIN est un SALESMAN
BLAKE est un MANAGER
CLARK est un MANAGER
SCOTT est un ANALYST
KING est un PRESIDENT
TURNER est un SALESMAN
ADAMS est un CLERK
JAMES est un CLERK
FORD est un ANALYST
MILLER est un CLERK
14 rows selected.
```

Appeler des méthodes Java depuis des blocs PL/SQL

Une procédure stockée Java peut être appelée depuis n'importe quel bloc PL/SQL. Définissons une classe Salaire qui comporte une méthode augmentant le salaire d'un employé d'une somme donnée. Cette méthode augSalaire utilise un argument qui identifie le salarié et la valeur de l'augmentation de salaire. Voici le code source du fichier SalaireMaj.sqlj :

```
import sqlj.runtime.ref.*;
import java.sql.*;

public class SalaireMaj {
 public static void augSalaire (int empid, float valeur) throws SQLException {
  #sql {UPDATE emp SET sal = sal+:valeur WHERE empno=:empid};
  }

}
```

Publions la méthode Java augSalaire en tant que procédure PL/SQL sous le nom augmente_salaire (la méthode Java augSalaire ne renvoie pas de valeur).

```
CREATE OR REPLACE PROCEDURE augmente_salaire (empid NUMBER, valeur NUMBER)
AS LANGUAGE JAVA
NAME 'SalaireMaj.augSalaire(int, float)';
```

On peut appeler la méthode augmente_salaire :

- à partir d'un bloc PL/SQL anonyme ;
- à partir d'une procédure PL/SQL ;
- à partir d'une méthode d'un type objet.

Par exemple, le bloc PL/SQL anonyme suivant appelle la procédure stockée augmente_salaire en lui indiquant l'identifiant de l'employé (7 788) et la hausse de salaire (1 000).

```
select sal from emp where empno=7788;
DECLARE
empid NUMBER;
valeur NUMBER;
BEGIN
empid:=7788;
valeur:=1000;
augmente_salaire(empid,valeur);
END;
/
select sal from emp where empno=7788;
```

Le résultat suivant est produit par cette séquence de commandes :

```
SAL
----------
     13 900
1 row selected.
```

Le salaire initial du salarié 7788 est 13 900. Nous exécutons le bloc PL/SQL anonyme et nous vérifions le nouveau salaire du salarié.

```
SQL> DECLARE
2> empid NUMBER;
3> valeur NUMBER;
4> BEGIN
5> empid:=7788;
6> valeur:=1000;
7> augmente_salaire(empid,valeur);
8> END;
9> /
Statement processed.
SQL> select sal from emp where empno=7788;
SAL
----------
     14 900
1 row selected.
```

Le salaire final du salarié 7 788 s'élève maintenant à 14 900.

Comment a été exécuté le bloc PL/SQL ?

L'appel de méthode augmente_salaire(empid,valeur) a réalisé les tâches suivantes :

- passage des valeurs empid et valeur à la méthode Java Salaire.augSalaire ;
- exécution de la méthode Java Salaire.augSalaire.

Appeler des sous-programmes PL/SQL depuis des méthodes Java

JDBC et SQLJ permettent d'appeler des procédures stockées PL/SQL depuis des méthodes Java.

La fonction PL/SQL empfonction renvoie la fonction d'un employé. Elle prend un argument de type NUMBER et retourne une valeur de type VARCHAR2.

```
CREATE OR REPLACE FUNCTION empfonction (empid NUMBER) RETURN VARCHAR2 IS
empGrade VARCHAR2(50);
BEGIN
SELECT job INTO empGrade FROM emp WHERE empno = empid;
RETURN empGrade;
END;
```

Testons empfonction depuis SQL*Plus :

```
SQL> select empfonction(7788) as Fonction from dual;
Fonction
--------
ANALYST
```

Le bloc d'instructions Java suivant remplit ces fonctions :

- il appelle la fonction PL/SQL empfonction ;
- il enregistre la valeur de retour dans une chaîne de caractères ;
- il affiche cette chaîne.

```
String chaine;
#sql chaine = {VALUES(empfonction(:empid))};
System.out.println("Fonction : "+chaine);
```

Le fichier SalInfo.sqlj suivant utilise ce bloc d'instructions :

```
import sqlj.runtime.ref.*;
import java.sql.*;

public class SalInfo {
  public SalInfo() throws SQLException {
  // Enregistrement du pilote JDBC Oracle
  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
  // Ouverture d'une session dans la base
  // Définition d'un contexte pour les instructions SQLJ
  try {
  Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@nomServeur:1521:orcl",
```

```
  ➥"scott","tiger");
  DefaultContext.setDefaultContext(new DefaultContext(conn));
  }
  catch (SQLException exception) {
System.out.println("Impossible d'ouvrir la session");
System.err.println(exception);
System.exit(1);
    }
  }


  public static void main (String [] args) throws SQLException {
  SalInfo application= new SalInfo();
  application.fonction(7788);
  }

  public static void fonction (int empid) throws SQLException {
  String chaine;
  #sql chaine = {VALUES(empfonction(:empid))};
  System.out.println("Fonction : "+chaine);
  }

}
```

L'exécution de ce code génère le message suivant :

```
Fonction : ANALYST
```

Cet exemple a montré comment invoquer simplement des procédures stockées PL/SQL à partir de méthodes Java grâce au standard SQLJ.

Résumé

Ce chapitre a présenté l'architecture Java d'Oracle 10*g*. Vous avez découvert :

- le fonctionnement de la machine virtuelle Java ;
- le déploiement des objets Java dans la base de données ;
- les possibilités d'utilisation de Java en complément du langage SQL.

Vous êtes désormais prêt à développer des logiciels évolutifs, sécurisés et portables.