Debugging met Oracle

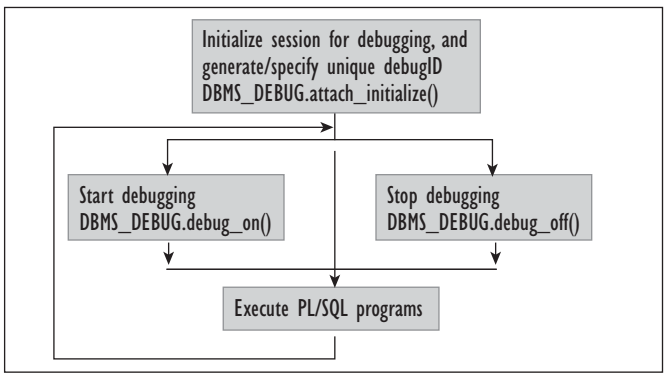
Toepassing van debugging-ondersteuning

Oracle biedt sinds Oracle 8i via het package DBMS_DEBUG ondersteuning voor debugging. Gert-Jan Paulissen zal dit package onder de loep nemen en de volgende toepassingen presenteren: een interactieve debugger in SQL*Plus en het automatisch tonen van waardes van in- en out-parameters bij het begin en het einde van stored (package) functions en procedures.

JDeveloper biedt standaardfunctionaliteit om PL/SQL server-side code te debuggen. Er wordt geen gebruik gemaakt van DBMS_DEBUG, maar van een nog ongedocumenteerd package DBMS_DEBUG_JDWP. Het voordeel van JDeveloper is de grafische user interface voor debuggen en het feit dat ook Java-code gedebugged kan worden; het nadeel is dat het lang duurt voor je aan het debuggen kan beginnen (JDeveloper opstarten, handmatig een connectie maken, enzovoorts) en dat het debuggen geen programmatische aanknopingspunten biedt: je kunt de debug-resultaten niet automatisch verwerken. Zie [1] in het kader 'Referenties' voor meer informatie.

Gebruik van DBMS_DEBUG

Het package DBMS_DEBUG is beschreven in de Oracle-documentatie (zie [2]). Ik geef hier een kleine samenvatting. Om server-side PL/SQL te debuggen zijn twee sessies nodig: een sessie om de code in debug-mode te draaien (de target sessie) en een tweede sessie (de debug sessie) die de target sessie monitort. Figuur 1 en figuur 2 tonen de operaties in de target en debug sessie:



Figuur 1. De target sessie: de database sessie waarin de PL/SQL-code draait die wordt gedebugged.

Een standaardscenario luidt als volgt: de target-sessie start en initialiseert de debug-sessie, Dan wordt het te debuggen programma gestart. Dit kan bijvoorbeeld gebeuren in een anoniem stukje PL/SQL-code (begin dbms_debug.initialize; dbms_debug.debug_on; execute_PLSQL; dbms_debug.debug_off; end;) maar bijvoorbeeld ook in een ON-LOGON trigger. Dat laatste betekent dat ook de PL/SQL-code die door client/server applicaties wordt uitgevoerd, kan worden gedebugged. Let op: de aanroep van dbms_debug.initialize levert een debugID op; dat id hebben we nodig in de debug-sessie om contact te leggen met de targetsessie. Vervolgens wordt de debug-sessie gestart; deze sessie legt een verbinding met de target-sessie door middel van een aanroep van dbms_debug.attach_session(debugID).

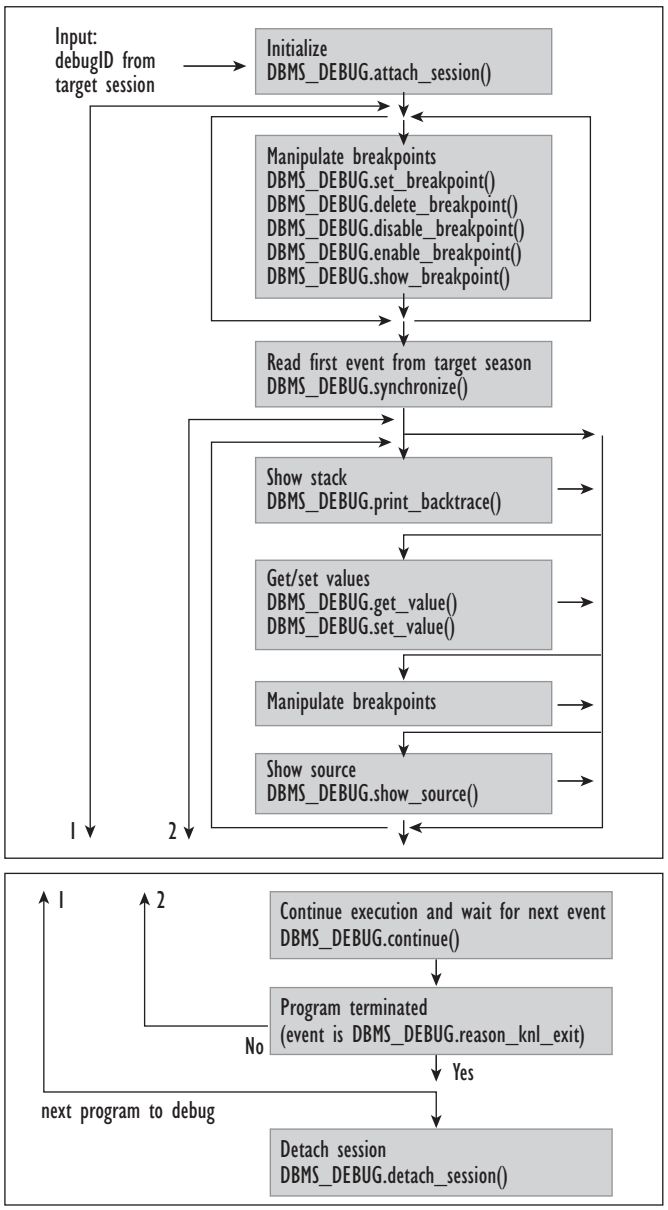
Een voorwaarde om waarden van parameters en variabelen te tonen of modificeren is dat de te debuggen PL/SQL-code met debug-informatie wordt gecompileerd. Op de volgende manieren kan met debug-informatie gecompileerd worden:

- ALTER SESSION SET PLSQL_DEBUG = TRUE;
- ALTER [PROCEDURE | FUNCTION | PACKAGE | TRIGGER | TYPE] <name> COMPILE DEBUG;
- ALTER [PACKAGE | TYPE] <name> COMPILE DEBUG BODY;

Als je een PL/SQL-block in SQL*Plus wilt debuggen, dan moet PLSQL_DEBUG op TRUE staan via het eerste commando. Dit is niet zo duidelijk beschreven in de Oracle-documentatie.

Interactieve debugger

Informatie op Internet (zie [3]) heeft mij geïnspireerd tot het maken van een laag op DBMS_DEBUG. De extra laag moet voldoen aan de volgende eisen:



Figuur 2. De debug-sessie.

1. De targetsessie moet automatisch contact maken met de debug sessie;
2. Standaard debug-functionaliteit zoals stappen door de code moet eenvoudig zijn;
3. Het moet mogelijk zijn om te tracen: het begin en einde van een functie- of procedureaanroep tonen met de inputparameters respectievelijk de outputparameters.
4. De extra laag moet eenvoudig uitbreidbaar zijn;
5. Er hoeven geen andere programma's dan SQL*Plus gebruikt te worden om te kunnen debuggen.

Automatisch contact maken

Een nadeel van de standaard DBMS_DEBUG functionaliteit is dat de debug-sessie eerst het debugID van de target-sessie moet weten voordat gestart kan worden. Het is veel gemakkelijker wanneer je dit debugID automatisch krijgt en dat vervolgens de debug-sessie start. Dit kan gerealiseerd worden door het debugID via een database pipe te sturen naar de debug-sessie. De debug-sessie kan hiermee dbms_debug.attach_session() aanroepen. De database-pipe is publiek toegankelijk met als naam PLDBG.

Als de target-sessie klaar is, dan moet de debug-sessie ook een signaal krijgen dat het werk erop zit. Ook dit gebeurt weer via een database-pipe, echter nu via een privé database-pipe, die is genoemd naar de debugID: PLDBG$<debugID>.

Stappen door de code

Met de functie DBMS_DEBUG.CONTINUE kun je de volgende breakflags zetten om op bepaalde punten te stoppen:

Breakflag	Betekenis	Overeenkomstige functie in JDeveloper
break_next_line	Ga naar de volgende regel, maar stap niet in aanroepen.	Next
break_any_call	Ga naar de volgende regel in deze methode of in een aangeroepen methode.	Step
break_any_return	Stop pas bij het verlaten van deze methode.	Step to end of method
break_return	Stop bij het eerste verlaten van deze methode of enig andere aangeroepen methode.	Step out
break_exception	Stop wanneer een exceptie wordt 'geraised'.	Niet aanwezig
break_handler	Stop wanneer een 'exception handler' wordt uitgevoerd.	Niet aanwezig

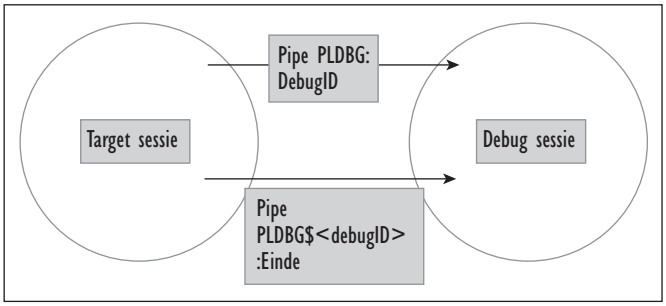
Tabel 1. Lijst van breakflags.

Deze breakflags worden gebruikt om te stoppen op genoemde punten en om vervolgens de code te tonen.

Traceerbaarheid

Functionaliteit die handig kan zijn, is om te kunnen zien waar een programma zich bevindt: welke aanroep is gestart of beëindigd en met welke parameterwaardes. Met behulp van het package DBMS_DEBUG en de data dictionary view ALL_ARGUMENTS is het mogelijk om het volgende te realiseren:

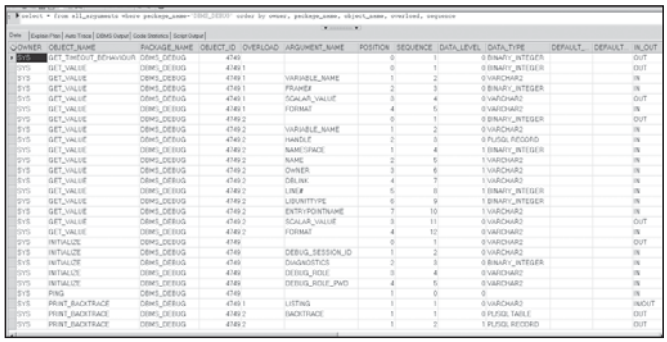
• Stoppen bij het begin en einde van elke PL/SQL methode;



Figuur 3. Berichten tussen sessies via database pipes.

• De input (bij het begin) en output (bij het einde) tonen.

De view ALL_ARGUMENTS bevat de stored (package) procedures/functies met hun argumenten en returnwaarden. De opbouw van de view ALL_ARGUMENTS is te zien in Figuur 4. Tabel 2 toont een beschrijving van kolommen die toelichting vereisen:



Figuur 4. Informatie over package DBMS_DEBUG in view ALL_ARGUMENTS.

Kolom	Opmerking
OBJECT_NAME	Naam van stored (package) functie/procedure.
PACKAGE_NAME	Naam van package. Leeg voor stand-alone functies/procedures.
OVERLOAD	Gevuld indien het een overloaded functie/procedure is.
ARGUMENT_NAME	Naam van argument. Leeg voor de return waarde van een functie (als DATA_TYPE niet leeg is).
SEQUENCE	Volgorde binnen functie/procedure.
DATA_LEVEL	0 voor het laagste niveau. PL/SQL records en tables worden met een hoger niveau getoond.
POSITION	Volgorde binnen een DATA_LEVEL.

Tabel 2. Beschrijving van ALL_ARGUMENTS.

Ik heb me beperkt tot het tonen van parameters met DATA_LEVEL 0.

Laag op DBMS_DEBUG

De laag op DBMS_DEBUG moet eenvoudig zijn uit te breiden. Dit is als volgt gerealiseerd:

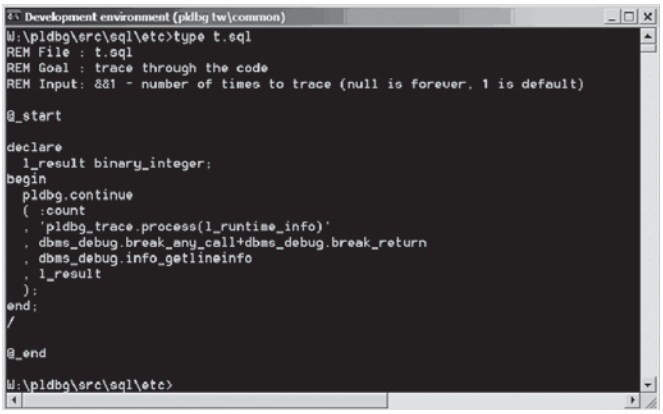
- Er is een package pldbg dat voorziet in het starten/stoppen van target en debug sessies en dat debug events kan verwerken. Bij de verwerking wordt dan een callback (bijvoorbeeld een functieaanroep in de vorm van dynamisch PL/SQL) aangeroepen, die het debug event dient te verwerken;
- Er bestaat een package pldbg_trace dat de tracing implementeert;
- Er is een package pldbg_show dat de broncode toont van het huidige breekpunt.

Debuggen met SQL*Plus

Naast de pldbg packages zijn er scripts gemaakt, die kunnen worden gebruikt in SQL*Plus:

Script	Omschrijving
bds.sql	Begin van een debug sessie.
bts.sql	Begin van een target sessie.
eds.sql	Einde van een debug sessie.
ets.sql	Einde van een target sessie.
c.sql	Continue: ga door tot het volgende breekpunt .
n.sql	Next: ga naar de volgende regel (sla aanroepen over).
r.sql	Run: ga naar het einde van de huidige procedure.
s.sql	Step: stap door de code (sla aanroepen niet over).
t.sql	Trace: toon begin en einde van functies/procedures.
p.sql	Print: toon de waarde van een parameter of variabele met als scope de huidige procedure.

Tabel 3. Lijst van debugger scripts.



Figuur 5. Bestand t.sql met als callback pldbg_trace.process(l_runtime_info).

Valkuilen

Bij elk anoniem PL/SQL-block wordt een nieuwe sessie van de debug interpreter gestart. Dat betekent bijvoorbeeld dat als je serveroutput aan hebt staan in de target-sessie, dat dan de debug-sessie de aanroep naar dbms_output.get_lines laat zien. Als je een PL/SQL-block wilt ingaan, dan moet een van je breakflags break_any_call zijn, anders wordt het hele blok overgeslagen. Gebruik dus bij het starten bijvoorbeeld eerst het script s.sql (step) of t.sql (trace).

Voorbeeld

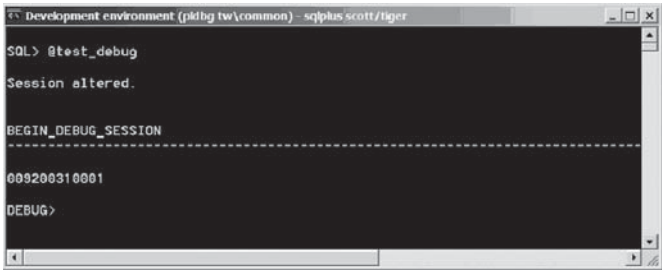
Als voorbeeld heb ik een tweetal faculteitsfuncties gemaakt: een recursieve en een iteratieve. Verder heb ik een package gemaakt die procedures heeft die beide functies aanroepen. Hiermee kan het gedrag van packages en stand-alone functies bekeken worden. Start een target sessie met het testscript test_target.sql als volgt:

De target-sessie wacht tot de debug-sessie zich koppelt aan de target-sessie.

Figuur 6. Start een target-sessie.

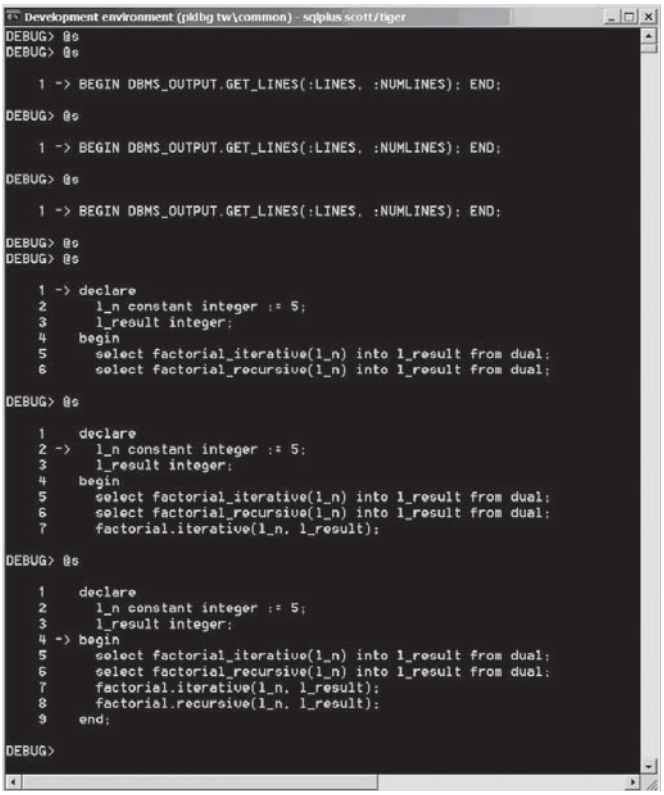
Stappen door de code

Start nu de debug sessie met testscript test_debug.sql als volgt:



Figuur 7. Starten van een debug-sessie.

De debugsessie heeft contact gemaakt met de target sessie door eerst de pipe uit te lezen, het debugID te verkrijgen en vervolgens met behulp van dbms_debug.attach_session() bij de target-sessie aan te haken. Nu kan er door de broncode heen gewandeld worden:



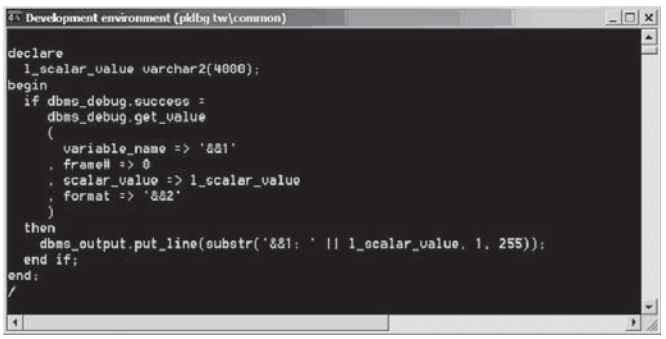
Figuur 8. Stappen door de code.

De volgende zaken vallen op:

1. Een aanroep van s.sql kan wel eens niets tonen omdat de debug interpreter start en daarna gelijk stopt;
2. Eerst wordt dbms_output.get_lines uitgevoerd door SQL*Plus zelf;
3. Regels met declare en begin zijn punten waar gestopt wordt. Een variabele declaratie met een assignment ook. Zonder een assignment wordt de regel niet getoond.

Het tonen van de waarde van een variabele of argument:

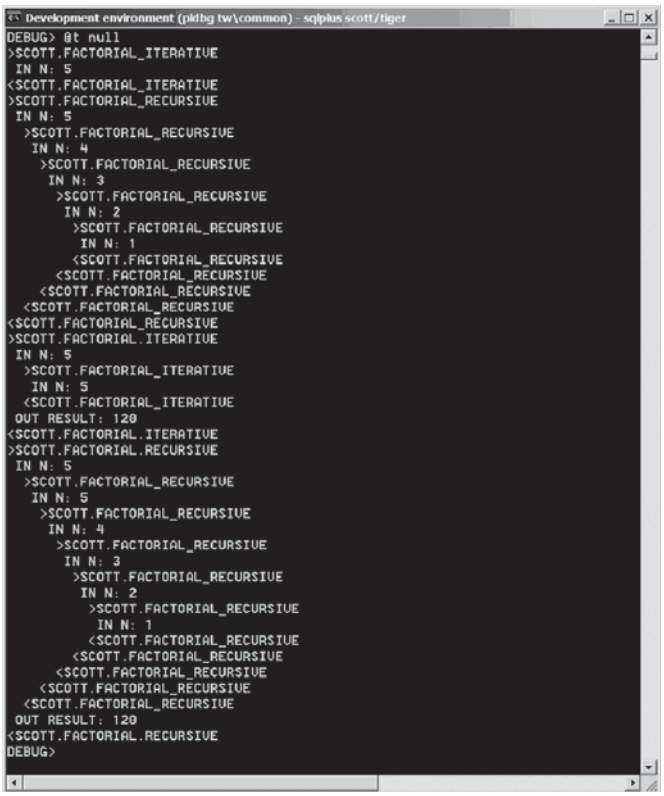
Figuur 9. Het tonen van een parameter met het p.sql script.



Figuur 10. Het bestand p.sql.

Tracen van de code

Nadat weer een target-sessie is gestart, kan ook getraced worden:



Figuur 11. Tracen van code.

Het is goed te zien dat de recursieve functie factorial_recursive zichzelf aanroept. Het resultaat (output) wordt alleen getoond bij de package-procedures omdat de functies geen output-parameters hebben.

Beschikbaarheid

Dit artikel, de broncode en de HTML-documentatie gegenereerd door PLDoc (zie [4]) zijn beschikbaar op de weblog van AMIS (zie [5]).

Conclusie

Een debugger kan helpen bij het opsporen van problemen en voorkomt dat je zelf debugging statements (bijvoorbeeld met dbms_output.put_line) moet toevoegen. Ook bij unit-testen kan debuggen van nut zijn, bijvoorbeeld ter controle of een aanroep wel bepaalde parameterwaardes heeft. Met de toepassing, zoals in dit artikel is beschreven, kan men eenvoudig met SQL*Plus debuggen en tracen. Er wordt standaard gebruik gemaakt van dbms_output, maar het is gemakkelijk om dit te implementeren met bijvoorbeeld UTL_FILE.

Referenties

[1] JDeveloper debugging: http://www.oracle.com/technology/tech/pl_sql/pdf/Paper_30720_Ppt.pdf.
[2] Oracle Database PL/SQL Packages and Types Reference 10g Release 2 (10.2): http://otn.oracle.com
[3] Debugging PL/SQL with DBMS_DEBUG: http://www.adp-gmbh.ch/ora/plsql/debug.html
[4] PLDoc: http://sourceforge.net/projects/pldoc
[5] AMIS weblog: http://technology.amis.nl/blog

Gert-Jan Paulissen is werkzaam bij Transfer Solutions.

NIEUWS

Oracle en Novell introduceren grid-ready toepassing voor datacenter

Novell en Oracle hebben de nieuwe Accelerator Service gepresenteerd voor stroomlijning van de implementatie van een grid-ready infrastructuur voor datacenters. Het gaat daarbij om toepassing van Oracle Database 10g, Oracle Real Application Clusters, Oracle Application Server 10g en Oracle Enterprise Manager 10g onder SUSE Linux Enterprise Server van Novell. Bij deze nieuwe opzet wordt gebruik gemaakt van Oracle Enterprise Manager 10g Grid Control (Oracle Grid Control) en de functie Automated Storage Management van Oracle Database 10g om een complete, gebruiksklare beheeroplossing tot stand te brengen voor monitoring en management van grid-applicaties, infrastructurele software en gegevensopslag.
De nieuwe Accelerator Service is een initiatief waarin deskundig personeel van Oracle gaat samenwerken met Novell, HP en Egenera om het traject van implementatie en ingebruikname van de infrastructurele Grid Computing-software van Oracle onder SUSE Linux Enterprise Server sneller te laten verlopen. Het is een inmiddels in de praktijk, bij diverse gezamenlijke klanten van de genoemde bedrijven, bewezen methodiek die risico's minimaliseert en de tijd voor ingebruikname van zakelijke grid-applicaties drastisch reduceert. De nieuwe Accelerator Service wordt aangeboden via een flexibel systeem van prijzen en opties, zodat de uiteindelijke diensten exact kunnen worden afgestemd op budget en behoeften van de klant.