LOGGER – Open Source zum Protokollieren von PL/SQL-Programmen

Andreas Wismann
WHEN OTHERS
D-41564 Kaarst

Schlüsselworte
LOGGER 3.0, Open Source, PL/SQL

Einleitung
Fehlerbehandlung. Ein Aspekt, der in vielen Projekten erst zum Ende der Entwicklungsphase thematisiert wird – wenn der Großteil der Programme "ordentlich" läuft und man sich (vermeintlich) nur noch um vereinzelte Ausreißer kümmern muss. Warum wird das Exception Handling so stiefmütterlich behandelt?

Häufige Antworten:

- weil es im Unternehmen keine anwendungsübergreifende Standard-Verfahrensweise gibt, wie überhaupt mit Laufzeitfehlern zu verfahren ist
- weil genau deshalb der einzelne Entwickler keine Anhaltspunkte hat, wie er im Falle eines Programmfehlers mit der Situation umgehen soll
- weil unklar ist, welche Seite der Schnittstelle für die Daten verantwortlich zeichnet
- weil die bestehenden Programme zu verschachtelt und gegenseitige Abhängigkeiten schwierig zu analysieren sind
- weil es an Kenntnisse mangelt, wie sauberes Exception Handling funktioniert
- weil Performance-Einbußen befürchtet werden, wenn Fehlercode hinzugefügt wird
- weil das Kodieren einer leistungsfähigen Fehlerbehandlung in PL/SQL recht umständlich ist und wertvolle Projektzeit verbraucht
- weil schon seit geraumer Zeit alles fehlerfrei läuft

Um diese Probleme zu adressieren und ein leistungsfähiges Logging und Handling von Fehlersituationen zu ermöglichen, gibt es das Open-Source-Projekt LOGGER, sowie zwei darauf aufbauende Utilities, die das Arbeiten mit dem LOGGER-Package vereinfachen und beschleunigen.

PL/SQL bietet ein zuverlässiges "automatisches" Exception Handling, so dass beim Auftreten einer Fehlersituation die Konsistenz der zu verarbeitenden Daten stets gewährleistet ist (doch Vorsicht - auch dieses Prinzip lässt sich durch bewusste programmatische Eingriffe oder durch Unkenntnis aushebeln). Jedoch, einer der wenigen Schwachpunkte der Programmiersprache PL/SQL besteht darin, dass nicht alle Informationen im Zusammenhang mit der Fehlerentstehung im Nachhinein verfügbar sind. Das betrifft vor allem die Werte von Parametern, die einer Routine übergeben worden sind. Ein Beispiel: die allseits "gefürchtete" Fehlermeldung

```
ORA-01722 Ungültige Zahl / ORA-01722 invalid number
```

Viel Erfolg (oder besser: viel Glück), wenn diese Fehlermeldung nach Monaten anstandsloser Programmausführung eines Morgens in der Produktion aufschlägt, während sämtliche Abteilungen auf die Daten warten...

Wenn kein Logging existiert, müssen nun (unter Zeitdruck!) sämtliche Stammdaten, Bewegungsdaten, SQL-Statements und Verarbeitungsschritte im gesamten Programmverlauf daraufhin überprüft werden,

1. wo genau der Fehler eigentlich passiert ist,
2. welcher konkrete Wert nicht akzeptiert wurde,
3. welche Relevanz dieser Fehler für die weitere Ausführung des Programms hat (wie würde sich eine "schnelle" Fehlerbehandlung auswirken?)
4. ob dieser Fehler bereits früher aufgetreten ist, und ober er nach seiner Korrektur später nicht mehr auftritt

Zu 1.: Der vollständige Call Stack ist nur dann bis zur Quelle zurückverfolgbar, wenn er beim Auftreten des Fehlers in irgendeiner Weise „gerettet“ worden ist. Falls kein dediziertes Exception Handling und Logging existiert, geht diese Information verloren und kann somit nicht mehr zur Fehlersuche verwendet werden.

Zu 2.: Der konkrete problematische Wert, der zu diesem Laufzeitfehler geführt hat, ist nicht Bestandteil der Oracle-Fehlermeldung und wird auch sonst nirgends „automatisch“ protokolliert. Ohne Logging kann das Aufspüren der Störung zum Kaffeesatzlesen ausarten.

Zu 3.: Solange die Fehlerquelle nicht exakt lokalisiert ist, kann auch keine Aussage darüber getroffen werden, mit welchen Mitteln das Problem behoben werden kann. Das Programm bricht ja in jedem Fall mit einem Laufzeitfehler ab, egal ob der Fehler bei der Verarbeitung von wichtigen Geschäftsdaten oder vielleicht nur bei der nachlässigen Programmierung eines Fortschrittbalkens auftritt!

Zu 4.: Nicht jeder User meldet jeden Fehler. Wenn sich das Problem beispielsweise in einer APEX-Applikation manifestiert (mit einer Fehlermeldung am oberen Bildschirmrand), dann wird es typischerweise erst dann behandelt, wenn der Anwender ein Ticket erstellt. Die Frage, ob Ihre Anwendung überhaupt fehlerfrei läuft, lässt sich also ohne Fehler-Logging nur beantworten, wenn alle(!) Anwender alle(!) Fehler bemerken und weitergeben. Doch wie will man das jemals sicherstellen?

Installation von LOGGER
Das Open-Source-Projekt wird federführend geleitet von Martin d’Souza. Der Download-Link lautet

https://github.com/OraOpenSource/Logger

Zur Open Source gehören neben dem PL/SQL-Package einige Tabellen, die in jedem beliebigen Schema installiert werden können, auf das die zu überwachenden Schemata jeweils EXECUTE-Privilegien besitzen. Mit anderen Worten, man kann LOGGER entweder in das betreffende Schema installieren („neben“ die anderen bereits bestehenden Datenbankobjekte) oder in einem bestimmten Schema, welches zentral für das Logging zuständig sein soll. Wenn bereits ein Schema existiert, welches für allgemeine Anwendungssteuerung bereitsteht, wäre das der ideale Ort zur Installation von LOGGER.

Prinzipielle Funktionsweise
Von allein bewirkt die Installation von LOGGER gar nichts. Vielmehr müssen Sie ausdrücklich alle PL/SQL-Codeblöcke, die „LOGGER-fähig“ sein sollen (typischerweise also Packages, Procedures und Functions), mit zusätzlichem Code instrumentieren. Dahinter verbirgt sich natürlich etwas Arbeit.

Grundsätzlich ist das Vorgehen so, dass man LOGGER zu Beginn einer aufgerufenen Routine die Werte der übergebenen Parameter mitteilt, um sie dann im Falle eines Fehlers – per Exception-Handler – in die LOGGER_LOGS-Tabelle (und ggf. in Detailtabellen) zu schreiben und sie somit für die nachträgliche Analyse zu retten. Das heißt, man benötigt an mindestens zwei Stellen zusätzlichen Code:

- Im Initialisierungsabschnitt der Procedure oder Function und
- Im EXCEPTION-Abschnitt der Procedure oder Function

Die gute Nachricht: Es ist absolut praktikabel, zunächst nur wenige Stellen des vorhandenen Programmcodes mit den LOGGER-Aufrufen zu bestücken. Es besteht also kein zwingender Anlass für eine Komplettrenovierung sämtlicher Programme. Üblicherweise benötigt man den LOGGER-Code ja zunächst an wenigen, besonders fehlerträchtigen Stellen im Programm. Wenn man daraufhin feststellt, wie hilfreich die erzeugten zusätzlichen Informationen sein können, kann man Zug um Zug immer mehr Programmcode instrumentalisieren. Falls man sich (aus welchen Gründen auch immer) scheut, neuralgische Teile des bestehenden Codes zu verändern, so kann man immer eine (beliebige) Auswahl an Modulen treffen, die bearbeitet werden. Im Fall eines „Grüne-Wiese-Projekts“ ist es vorteilhaft, sämtliche Routinen direkt mit LOGGER-Calls zu bestücken. Man wird mit maximaler Kontrolle und Transparenz in Fehlersituationen belohnt.

Am Beispiel einer Dummy-Funktion zeige ich hier, wie Ihr Code vorher und nachher aussieht:

```
CREATE OR REPLACE FUNCTION beispiel_1a (i_count IN NATURAL)
RETURN VARCHAR2 DETERMINISTIC
IS
BEGIN
  CASE i_count
       WHEN 0 THEN RETURN 'Null';
       WHEN 1 THEN RETURN 'eins';
       WHEN 2 THEN RETURN 'zwei';
       WHEN 3 THEN RETURN 'drei';
   END CASE;
   RETURN 'mehr als drei';
END beispiel_1a;
```

Das geübte Auge erkennt, was in dieser Funktion früher oder später schieflaufen wird:

```
SELECT beispiel_1a(4) FROM dual;
-- ORA-06592: CASE bei Ausführung von CASE-Anweisung nicht gefunden
```

Mit LOGGER-Befehlen ausgerüstet, sähe die Funktion so aus:

```
CREATE OR REPLACE FUNCTION beispiel_1b (i_count IN NATURAL)
RETURN VARCHAR2 DETERMINISTIC
IS
  v_tab_param   LOGGER.tab_param;
BEGIN

  logger.append_param(v_tab_param, 'i_count', i_count);

  CASE i_count
       WHEN 0 THEN RETURN 'Null';
       WHEN 1 THEN RETURN 'eins';
```

```
      WHEN 2 THEN RETURN 'zwei';
      WHEN 3 THEN RETURN 'drei';
   END CASE;
   RETURN 'mehr als drei';
EXCEPTION
  WHEN OTHERS THEN
    logger.log_error(
       p_params => v_tab_param
    );
    RAISE;
END beispiel_1b;
```

Geben Sie nun ein:

```
SELECT beispiel_1b(4) FROM dual;
-- ORA-06592: CASE bei Ausführung von CASE-Anweisung nicht gefunden
```

In der Tabelle LOGGER_LOGS finden Sie nun äußerst hilfreiche Informationen zu diesem Fehler:

```
SELECT * FROM logger_logs
 ORDER BY ID DESC;
```

Row 1	Fields
ID	13835945
LOGGER_LEVEL	2
TEXT	ORA-06592: CASE bei Ausführung von CASE-Anweisung nicht
TIME_STAMP	22.10.15 15:32:59,060171
SCOPE	
MODULE	PL/SQL Developer
ACTION	Secondary Session
USER_NAME	AWISMANN
CLIENT_IDENTIFIER	
CALL_STACK	ORA-06592: CASE bei Ausführung von CASE-Anweisung nicht
UNIT_NAME	
LINE_NO	
SCN	
EXTRA	<CLOB>
SID	2193
CLIENT_INFO	

Neben zachlreichen hilfreichen Informationen wartet der eigentliche “Bonus“ im CLOB-Feld namens EXTRA:

```
*** Parameters ***

i_count: 4
```

Es wurde also zur Laufzeit genau derjenige problematische Wert aufgezeichnet, der im „Programm“ (hier repräsentiert durch die SELECT-Anweisung) letztendlich zum Fehler geführt hat. Ohne LOGGER wäre es äußerst schwierig bis unmöglich, diesen Wert im Nachhinein zu rekonstruieren, zumindest wenn wir von komplexen PL/SQL-Programmen sprechen. Solche Parameter-Informationen sind nämlich, vereinfacht gesagt, sofort nach dem Fehler „weg“.

Na schön, in diesem trivialen Beispiel war die Lösung augenscheinlich simpel genug. Doch stellen Sie sich eine beliebig komplexe Materie vor: Die Fehlersuche ist möglicherweise zum Scheitern verurteilt, wenn Sie nicht genau wissen, welche konkrete Eingabe der Benutzer in der Anwendung vorgenommen hat. Oder welche Fakten in Ihrem Data Warehouse verrückt gespielt haben. Um nur zwei Beispiele zu nennen.
Man ahnt auch, dass bei umfangreicheren Signaturen mit vielen Parametern eine Menge Tipparbeit auf die Programmierer wartet, um den vorhandenen Code nachträglich mit LOGGER-Aufrufen zu bestücken. Doch es gibt zwei Tools, die den notwendigen LOGGER-Code automatisch generieren können:

1. LOGGER UTIL
   Autor: Alex Nuijten
   Beschreibung: Das Package wird in Ihrem Arbeitsschema installiert. Dadurch, dass es Zugriff auf das Data Dictionary hat, generiert es Code für bereits vorhandene Programme. Die Code-Ausgabe dieses Tools kann per Template an den eigenen Bedarf angepasst werden.
   https://github.com/alexnuijten/loggerutil

2. PL/SQL Code Generator for LOGGER
   Autor: Andreas Wismann
   Beschreibung: Die APEX-Anwendung erwartet eine Funktions- oder Prozedursignatur per Copy/Paste, analysiert deren Aufbau und generiert daraus den LOGGER-Quellcode mit diversen Laufzeit- und Formatierungsoptionen sowie eine PLSQLDoc-kompatible Dokumentation. Eine Installation ist nicht erforderlich.
   https://apex.oracle.com/pls/apex/f?p=58849:10

Ausblick auf den Vortrag
In der Präsentation vertiefe ich anhand von Live-Beispielen das Package LOGGER und die beiden Tools. In diesem Manuskript habe ich nur an der Oberfläche dieser Utilities gekratzt und möchte die Neugier wecken, sich die Live-Demos anzuschauen. Es wird beispielsweise gezeigt, wie Sie elegant verhindern, dass der Befehl

```
logger.append_param(…);
```

welcher üblicherweise bei jedem Aufruf der Funktion ausgeführt wird, stets mitläuft, was die Performance Ihres Codes positiv beeinflusst. Mit hoher Wahrscheinlichkeit werden Sie nach der Präsentation das Projekt LOGGER ganz weit nach oben auf Ihre Prioritäten-Liste setzen!

Kontaktadresse:
Andreas Wismann
WHEN OTHERS Inh. Andreas Wismann
Hirschstr. 10
D-41564 Kaarst

Telefon: +49 (0) 2131 - 314 9966
mobil: +49 (0) 176 - 7800 3109
E-Mail: wismann@when-others.com
Internet: when-others.com