Chapter 10

Mastering Advanced Exception Handling in PL/SQL

Mayko Freitas da Silva

Introduction

Imagine you're a skilled detective, tasked with solving complex mysteries in the world of PL/SQL. Your job is to uncover hidden errors, decipher cryptic messages, and create a foolproof system to handle unexpected situations. Welcome to the exciting world of advanced exception handling.

In this chapter, you'll elevate your PL/SQL skills to new heights by mastering three powerful tools:

1. RAISE_APPLICATION_ERROR: Your custom alarm system
2. EXCEPTION_INIT Pragma: Your error message translator
3. SQLCODE and SQLERRM: Your trusty error detectives

These advanced techniques will transform you from a novice coder into a PL/SQL expert, capable of creating robust, user-friendly applications that gracefully handle even the most unexpected errors.

By the end of this chapter, you will be able to:

- Create meaningful, application-specific error messages that speak directly to your users
- Handle those tricky Oracle errors that don't have predefined names
- Extract detailed error information to quickly diagnose and resolve issues

Are you ready to become a master of PL/SQL exception handling? Let's dive in and unravel the mysteries of these advanced techniques.

This introduction aims to engage the learner by using the detective metaphor, clearly outlining the main topics, and setting expectations for what they'll learn. It's written in a conversational tone, addressing the learner directly, and avoids using exclamation marks as requested.

Would you like me to continue with the first main section, or do you have any adjustments you'd like me to make to the introduction?

Lab 10.1: RAISE_APPLICATION_ERROR - Your Custom Alarm System

After this lab, you will be able to:

- Create meaningful, application-specific error messages
- Use RAISE_APPLICATION_ERROR effectively
- Compare RAISE_APPLICATION_ERROR with named user-defined exceptions

Understanding RAISE_APPLICATION_ERROR

Imagine you're designing a security system for a building. Instead of a generic alarm that simply makes noise, you want to create custom alerts that provide specific information about the nature and location of the problem. In PL/SQL, RAISE_APPLICATION_ERROR is your tool for creating these custom alerts.

RAISE_APPLICATION_ERROR is a built-in procedure provided by Oracle. It allows you to create error messages that are meaningful and specific to your application. This is particularly useful when you want to provide clear, actionable information to users or other parts of your application when something goes wrong.

Syntax and Parameters

Let's look at how you can use this custom alarm system. RAISE_APPLICATION_ERROR has two forms:

```
RAISE_APPLICATION_ERROR(error_number, error_message)

-- or

RAISE_APPLICATION_ERROR(error_number, error_message, keep_errors)
```

Here's what each parameter means:

1. error_number: This is like your alarm code. It must be a number between -20,999 and -20,000.
2. error_message: This is your custom message, explaining what went wrong. It can be up to 2048 characters long.
3. keep_errors: This is optional. If set to TRUE, your new error is added to the list of previous errors (the error stack). If FALSE (the default), your new error replaces the error stack.

Practical Example

Let's see how you can use RAISE_APPLICATION_ERROR in a real-world scenario using the HR schema:

```
DECLARE
   v_employee_id NUMBER := &sv_employee_id;
   v_salary NUMBER;
BEGIN
   SELECT salary INTO v_salary
   FROM employees
   WHERE employee_id = v_employee_id;

   IF v_salary > 20000 THEN
      RAISE_APPLICATION_ERROR(-20001, 'Salary exceeds maximum allowed: ' || TO_CHAR(v_salary));
   END IF;

   DBMS_OUTPUT.PUT_LINE('Employee salary is within acceptable range.');
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR(-20002, 'No employee found with ID: ' || TO_CHAR(v_employee_id));
END;
```

In this example, you're checking an employee's salary. If it's too high or the employee doesn't exist, you raise custom errors with specific messages.

Comparing with Named User-Defined Exceptions

Now, let's compare this approach with named user-defined exceptions:

```
DECLARE
   v_employee_id NUMBER := &sv_employee_id;
   v_salary NUMBER;
   e_salary_too_high EXCEPTION;
BEGIN
   SELECT salary INTO v_salary
   FROM employees
   WHERE employee_id = v_employee_id;

   IF v_salary > 20000 THEN
      RAISE e_salary_too_high;
   END IF;

   DBMS_OUTPUT.PUT_LINE('Employee salary is within acceptable range.');
EXCEPTION
   WHEN e_salary_too_high THEN
      DBMS_OUTPUT.PUT_LINE('Salary exceeds maximum allowed: ' || TO_CHAR(v_salary));
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No employee found with ID: ' || TO_CHAR(v_employee_id));
END;
```

While both approaches work, RAISE_APPLICATION_ERROR offers several advantages:

1. It provides a standardized error format that resembles Oracle's built-in errors.
2. It allows you to specify an error number, which can be useful for error handling in calling programs.
3. It doesn't require separate exception declaration and handling sections, making the code more concise.

Exercise 10.1

Try modifying the RAISE_APPLICATION_ERROR example to check for a minimum salary requirement. If an employee's salary is below $2000, raise an application error with the message "Employee salary below minimum wage". Use an appropriate error number between -20,999 and -20,000.

Remember, RAISE_APPLICATION_ERROR is your custom alarm system in PL/SQL. Use it wisely to provide clear, specific information when exceptions occur in your code.

Lab 10.2: EXCEPTION_INIT Pragma - Your Error Message Translator

After this lab, you will be able to:

- Associate Oracle error numbers with user-defined exception names
- Handle internally defined exceptions effectively
- Improve your exception handling interface

Understanding EXCEPTION_INIT Pragma

Imagine you're a translator working with error messages from different sources. Some messages come with clear names, while others only have numbers. The EXCEPTION_INIT pragma is your tool for giving names to these numbered messages, making them easier to understand and handle in your code.

In PL/SQL, not all Oracle errors have predefined names. These are called internally defined exceptions. The EXCEPTION_INIT pragma allows you to associate these unnamed Oracle error numbers with your own exception names, making your code more readable and maintainable.

Syntax and Usage

Here's how you use this error message translator:

```
DECLARE
   your_exception_name EXCEPTION;
   PRAGMA EXCEPTION_INIT(your_exception_name, -error_number);
BEGIN
   -- Your code here
EXCEPTION
   WHEN your_exception_name THEN
      -- Handle the exception
END;
```

The PRAGMA EXCEPTION_INIT statement goes in the declaration section of your PL/SQL block. It takes two parameters:

1. The name of your exception
2. The Oracle error number you want to associate with your exception

Practical Example

Let's see how you can use EXCEPTION_INIT in a real-world scenario using the HR schema:

```
DECLARE
   v_department_id NUMBER := &sv_department_id;
   e_foreign_key_violation EXCEPTION;
   PRAGMA EXCEPTION_INIT(e_foreign_key_violation, -2292);
BEGIN
   DELETE FROM departments
   WHERE department_id = v_department_id;

   DBMS_OUTPUT.PUT_LINE('Department deleted successfully.');
EXCEPTION
   WHEN e_foreign_key_violation THEN
      DBMS_OUTPUT.PUT_LINE('Cannot delete department. Employees are still assigned to it.');
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
END;
```

In this example, you're trying to delete a department. However, if there are still employees assigned to this department, Oracle will raise error ORA-02292 (child record found). By using EXCEPTION_INIT, you can catch this specific error and provide a more user-friendly message.

Handling Internally Defined Exceptions

The real power of EXCEPTION_INIT comes when dealing with Oracle errors that don't have predefined names. For example, let's handle an error that occurs when trying to insert a duplicate value into a unique column:

```
DECLARE
   v_department_name VARCHAR2(30) := '&sv_department_name';
   v_manager_id NUMBER := &sv_manager_id;
   v_location_id NUMBER := &sv_location_id;
   e_duplicate_department EXCEPTION;
   PRAGMA EXCEPTION_INIT(e_duplicate_department, -00001);
BEGIN
   INSERT INTO departments (department_id, department_name, manager_id, location_id)
   VALUES (departments_seq.NEXTVAL, v_department_name, v_manager_id, v_location_id);

   DBMS_OUTPUT.PUT_LINE('Department added successfully.');
EXCEPTION
   WHEN e_duplicate_department THEN
      DBMS_OUTPUT.PUT_LINE('A department with this name already exists.');
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
END;
```

Here, you're handling the ORA-00001 error (unique constraint violated) that occurs when trying to insert a department with a name that already exists.

Exercise 10.2

Modify the last example to handle another internally defined exception. Try to insert a new employee with an invalid department_id (e.g., 9999). Use EXCEPTION_INIT to catch the foreign key constraint violation (ORA-02291) and provide a user-friendly error message.

Remember, EXCEPTION_INIT is your translator for Oracle error numbers. It helps you create more readable and maintainable exception handling code by giving meaningful names to otherwise cryptic error numbers.

Lab 10.3: SQLCODE and SQLERRM - Your Error Detectives

After this lab, you will be able to:

- Extract detailed error information using SQLCODE and SQLERRM
- Enhance your exception handling with specific error codes and messages
- Understand different scenarios for SQLCODE return values

Understanding SQLCODE and SQLERRM

Imagine you're a detective investigating a crime scene. You need tools to gather evidence and understand what happened. In the world of PL/SQL error handling, SQLCODE and SQLERRM are your investigative tools. They help you gather crucial information about errors that occur in your code.

- SQLCODE is like your case number. It returns the Oracle error number of the exception.
- SQLERRM is like your case file. It returns the error message associated with the error number.

These functions are particularly useful when you're dealing with unexpected errors or when you want to log detailed error information.

Using SQLCODE and SQLERRM with OTHERS Exception Handler

Let's see how you can use these error detectives in a practical scenario using the HR schema:

```
DECLARE
   v_employee_id NUMBER := &sv_employee_id;
   v_salary NUMBER;
BEGIN
   SELECT salary INTO v_salary
   FROM employees
   WHERE employee_id = v_employee_id;

   DBMS_OUTPUT.PUT_LINE('Employee salary: ' || v_salary);
EXCEPTION
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error Code: ' || SQLCODE);
      DBMS_OUTPUT.PUT_LINE('Error Message: ' || SUBSTR(SQLERRM, 1, 200));
END;
```

In this example, if any error occurs (like entering an invalid employee_id), the OTHERS exception handler will catch it and display the error code and message.

SQLCODE Return Values

SQLCODE can return different values depending on the situation:

1. 0: No error (successful completion)

2. +100: No data found exception
3. Negative numbers: Oracle errors
4. +1: User-defined exception

Let's see these in action:

```
DECLARE
   v_employee_id NUMBER := &sv_employee_id;
   v_last_name VARCHAR2(25);
   e_custom EXCEPTION;
BEGIN
   DBMS_OUTPUT.PUT_LINE('Before SELECT: ' || SQLCODE);

   SELECT last_name INTO v_last_name
   FROM employees
   WHERE employee_id = v_employee_id;

   DBMS_OUTPUT.PUT_LINE('After SELECT: ' || SQLCODE);

   IF v_last_name = 'King' THEN
      RAISE e_custom;
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No data found: ' || SQLCODE);
   WHEN e_custom THEN
      DBMS_OUTPUT.PUT_LINE('Custom exception: ' || SQLCODE);
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Other error: ' || SQLCODE);
END;
```

SQLERRM with Custom Error Numbers

SQLERRM can also be used with custom error numbers. This is useful when you want to check what message is associated with a specific Oracle error number:

```
BEGIN
   DBMS_OUTPUT.PUT_LINE('Error -1017: ' || SQLERRM(-1017));
   DBMS_OUTPUT.PUT_LINE('Error -1476: ' || SQLERRM(-1476));
   DBMS_OUTPUT.PUT_LINE('Error 100: ' || SQLERRM(100));
   DBMS_OUTPUT.PUT_LINE('Custom error -20999: ' || SQLERRM(-20999));
END;
```

Exercise 10.3

Create a PL/SQL block that attempts to insert a new employee into the EMPLOYEES table with invalid data (e.g., a department_id that doesn't exist). Use SQLCODE and SQLERRM in the exception handler to display detailed error information. Then, use SQLERRM to display the error message for the specific Oracle error number you encountered.

Remember, SQLCODE and SQLERRM are your detective tools for investigating errors in your PL/SQL code. Use them wisely to gather detailed information about exceptions, helping you create more robust and informative error handling in your applications.

Summary

In this chapter, you've mastered three powerful tools for advanced exception handling in PL/SQL:

1. RAISE_APPLICATION_ERROR: Your custom alarm system for creating meaningful, application-specific error messages.
2. EXCEPTION_INIT Pragma: Your error message translator for associating Oracle error numbers with user-defined exception names.
3. SQLCODE and SQLERRM: Your error detectives for extracting detailed information about exceptions.

These techniques will help you create more robust, user-friendly PL/SQL applications with informative error handling. By implementing these tools, you'll be better equipped to diagnose and resolve issues in your code, improving the overall quality and maintainability of your PL/SQL projects.

Mastering Advanced Exception Handling in PL/SQL