Chapter 5: Advanced SQL

Database System Concepts, 6th Ed.

Outline

- Accessing SQL From a Programming Language
- Functions and Procedural Constructs
- Triggers
- Recursive Queries
- Advanced Aggregation Features
- OLAP

- 如何使用通用程序设计语言来访问SQL
- 介绍两种在数据库中执行程序代码的方法：
- 通过扩展SQL语言来支持程序的操作；
- 在数据库中，执行程序语言中定义的函数。
- 触发器，用于说明当特定事件（例如，在某个表上进行元组插入、删除或更新操作）发生时，自动执行的操作
- 递归查询
- SQL支持的高级聚集特性
- 联机分析处理(OLAP)系统，用于海量数据的交互分析。

5.1 Accessing SQL From a Programming Language

- SQL提供了一种强大的声明性查询语言。
- 数据库程序员必须能够使用通用程序设计语言，原因至少有以下两点：
  - 因为，SQL没有提供通用程序设计语言那样的表达能力，所以，SQL并不能表达所有查询要求。因而，要写这样的查询，我们可以将SQL嵌入到一种更强大的语言中。
  - 非声明性的动作（例如，打印一份报告、和用户交互，或者把一次查询的结果送到一个图形用户界面中）都不能用SQL实现。

- 一个应用程序通常包括很多部分，
  - 查询或更新数据只是其中之一，
  - 而其他部分，则用通用程序设计语言实现。
- 对于集成应用来说，必须用某种方法，把SQL与通用编程语言结合起来。
- 通用编程语言中访问SQL的方法:
  - 动态SQL:通用程序设计语言，可以通过函数（对于过程式语言）或者方法（对于面向对象的语言）来连接数据库服务器，并与之交互。

动态SQL

- 利用动态SQL，可以在运行时，以字符串形式构建SQL查询，提交查询
- 然后，把结果存入程序变量中，每次一个元组。
- 动态SQL的SQL组件，允许程序在运行时，构建和提交SQL查询。
- 两种用于连接到SQL数据库，并执行查询和更新的标准。
  - Java语言的应用程序接口JDBC
  - ODBC，语言如C、C++、C#和Visual Basic。

嵌入式SQL

- 嵌入式SQL语句，必须在编译时全部确定，并交给预处理器。
- 预处理程序，将SQL语句提交到数据库系统，进行预编译和优化，
- 然后，它把应用程序中的SQL语句替换成相应的代码和函数，
- 最后，调用程序语言的编译器进行编译。

- 把SQL与通用程序语言相结合的主要挑战是：
- 这些语言处理数据的方式互不兼容。
  - 在SQL中，数据的主要类型是关系。程序设计语言通常一次操作一个变量。
- 因此，为了在同一应用程序中，整合这两类语言，必须提供一种转换机制，使得程序语言可以处理查询的返回结果。

Language

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables

- Various tools:
  - ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic. Other API's such as ADO.NET sit on top of ODBC
  - JDBC (Java Database Connectivity) works with Java
  - Embedded SQL

5.1.1 JDBC

- JDBC is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for
  - querying
  - updating data,
  - and for retrieving query results.
- JDBC also supports metadata retrieval, such as
  - querying about relations present in the database
  - the names and types of relation attributes.

- Model for communicating with the database:
  - Open a connection
  - Create a “statement” object
  - Execute queries using the Statement object to send queries and fetch results
  - Exception mechanism to handle errors
  - 最后，关闭statement，关闭连接
- 注意，Java程序必须引用java.sql.*，它包含了，JDBC所提供功能的接口定义。
  - 即：import java.sql.*

JDBC Code

```
public static void JDBCexample(String dbid,
   String userid, String passwd)
 {
    try (
Connection
   conn=DriverManager.getConnection(
   "jdbc:oracle:thin:@db.yale.edu:2000:univdb",
   userid, passwd);
Statement stmt = conn.createStatement();
        )
```

```
{
        … Do Actual Work ….
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " +
sqle);
    }
}
```

- NOTE: Above syntax works with Java 7, and JDBC 4 onwards.
  Resources opened in "try (….)" syntax ("try with

JDBC Code for Older Versions of Java/JDBC

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
        … Do Actual Work ….
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

NOTE: Classs.forName is not required from JDBC 4 onwards. The try with resources syntax in prev slide is preferred for Java 7 onwards.

5.1.1.1 连接到数据库

- Connection conn=DriverManager.getConnection("jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
- 要在Java程序中访问数据库，首先，要打开一个数据库连接。
  - 这一步，需要选择要使用哪个数据库，可以是本机上的，也可以是远程数据库。
- 只有在打开数据库连接以后，Java程序才能执行SQL语句。

- 通过调用DriverManager类（在java.sql包中）的getConnection方法来打开一个数据库连接。
- 该方法有三个参数：
  - 第一个参数是，以字符串类型表示的URL，
    - 指明服务器所在的主机名称(@db.yale.edu)
    - 与数据库通信的协议（jdhc:oracle:thin:）
    - 数据库系统用来通信的端口号（2000），
    - 服务器端使用的特定数据库(univdb)。

- 注意，JDBC只是指定API，而不指定通信协议。
- 一个JDBC驱动器，可能支持多种协议，有多种版本的getConnection函数，它们所接受的参数各不相同。
  - getConnection(String url)
  - getConnection(String url, String user, String password)
  - getConnection(String url, Properties info)

- 例子中，我们已经建立了一个Connection对象，其句柄是conn
- 每个支持JDBC的数据库产品，都会提供一个JDBC驱动程序(JDBC driver)
- 该驱动程序，必须被动态加载才能实现Java对数据库的访问。
- 即，必须在连接数据库之前，完成驱动程序的加载。

- The Java SQL framework allows for multiple database drivers.
- Each driver should supply a class that implements the Driver interface.
- The DriverManager will try to load as many drivers as it can find and then for any given connection request, it will ask each driver in turn to try to connect to the target URL.
- It is strongly recommended that each Driver class should be small and standalone so that the Driver class can be loaded and queried without bringing in vast quantities of supporting code.

- When a Driver class is loaded, it should create an instance of itself and register it with the DriverManager. This means that a user can load and register a driver by calling:
  - Class.forName("foo.bah.Driver")
- A JDBC driver may create a DriverAction implementation in order to receive notifications when DriverManager.deregisterDriver(java.sql.Driver) has been called.

- Class.forName函数完成驱动程序的加载，
- 在调用时，需要通过一个参数来指定，实现java.sql.Driver接口的实体类。
- 这个接口的功能是：
  - 为了实现不同层面的操作之间的转换，
  - 一边是，与数据库产品类型无关的JDBC操作，
  - 另一边是，与数据库产品相关的、在所使用的特定数据库管理系统中完成的操作。

- 本例采用了Oracle的驱动程序，
  - oracle.jdbe.driver.OracleDriver。
- 该驱动程序包含在一个.jar文件里，可以从提供商的网站下载，
- 然后,放在Java的类路径(classpath)里，用于Java编译器访问。

- IBM DB2: coru.ibm.db2.jdbc.app.DB2Driver
- Micmsoft SQL Server: rom.microsoft.sqlserver.jdbe.SQLServerDriver
- PostgreSQL: org.postgresql.Driver
- MySQL: com.mysql.jdbc.Driver
- Sun公司，还提供了一种“桥接驱动器”，可以把JDBC调用转换成ODBC。

- 用来与数据库交换信息的具体协议，并没有在JDBC标准中定义，而是由所使用的驱动程序决定的。
- 有些驱动程序，支持多种协议，使用哪一种更合适，取决于所连接的数据库支持什么协议。
- 我们的例子里，在打开一个数据库连接时，字符串jdbc:oracle:thin:指定了Oracle支持的一个特定协议。

5.1.1.2 向数据库系统中传递SQL语句

- 一旦打开了一个数据库连接，程序就可以利用该连接来向数据库发送SQL语句用于执行。
- 这是通过Statement类的一个实例来完成的。
- Statement并不是SQL语句本身，而是一个对象，
- 通过Java程序调用该对象中的一些方法，可以实现传递SQL语句，
  - 在方法中，通过参数来传递SQL语句,并被数据库系统所执行。
- 例子中，在连接对象conn上，创建了一个Statement句柄(stmt)。
  - Statement stmt =conn.createStatement();

JDBC Code (Cont.)

- Update to database

```
try {
    stmt.executeUpdate(
        "insert into instructor values(’77987’, ’Kim’, ’Physics’, 98000)");
} catch (SQLException sqle)
{
    System.out.println("Could not insert tuple. " + sqle);
}
```

- executeQuery函数、executeUpdate，决于这条SQL语句
  - update、insert、delete、create table等
- stmt.executeUpdate执行了一条更新语句，向instructor关系中插人数据。
  - 它返回一个整数，表示被插入、更新或者删除的元组个数。
  - 对于DDL语句，返回值是0。
- try{…}catch{…}结构让我们可以捕捉JDBC调用产生的异常（错误情况），并显示给用户适当的出错信息。

5.1.1.3 获取查询结果

- stmt.executeQuery执行一次查询。
  - 它把结果中的元组集合，提取到ResultSet对象rset中，并每次取出一个进行处理。
- ResultSet中的next方法，
  - 用来查看在集合中，是否还存在至少一个尚未取回的元组，如果存在的话就取出。
  - next方法的返回值，是一个布尔变量，表示是否从结果集中取回了一个元组。

- 以get为前缀的方法，来得到所获取元组的各个属性。
  - 方法getString，可以返回所有的基本SQL数据类型的属性（被转换成Java中的String类型的值）。
  - getFloat，约束性更强的方法。
- 这些不同的get方法的参数，既可以是一个字符串类型的属性名称，又可以是一个整数，用来表示所需获取的属性在元组中的位置。

- Execute query and fetch and print results
  - ResultSet rset = stmt.executeQuery(
    "select dept_name, avg (salary)
    from instructor
    group by dept_name");

```
while (rset.next()) {
System.out.println(rset.getString("dept_name") +
" " + rset.getFloat(2));
}
```

JDBC Code Details

- Getting result fields:
  - rs.getString(“dept_name”) and rs.getString(1) equivalent
  - if dept_name is the first argument of select result.
- Dealing with Null values
  - int a = rs.getInt(“a”);
  - if (rs.wasNull()) Systems.out.println(“Got null value”);

- Java程序结束的时候，Statement和Connection都将被关闭。
  - stmt.close();
  - conn.close();
- 注意，关闭Connection是很重要的，因为数据库连接的个数是有限制的；未关闭的连接可能导致超过这一限制。
- 如果发生这种情况，应用将不能再打开任何数据库连接。

5.1.1.4 Prepared Statement

- 利用JDBC，我们可以创建一个预备语句
- 预备语句中，“？”来代表以后再给出的实际值
- The database system compiles the query when it is prepared.
- 在每次执行该语句时（executeUpdate()，用新值替换“？”），数据库可以重复使用预先编译的查询的形式，应用新值进行查询。

- PreparedStatement pStmt = conn.prepareStatement(
  "insert into instructor values(?,?,?,?)");
  pStmt.setString(1, "88877");
  pStmt.setString(2, "Perry");
  pStmt.setString(3, "Finance");
  pStmt.setInt(4, 125000);
  pStmt.executeUpdate();
  pStmt.setString(1, "88878");
  pStmt.executeUpdate();

- 使用Connection类的prepareStatement方法，来提交SQL语句用于编译。
- 它返回一个PreparedStatement类的对象。
- 此时还没有执行SQL语句。
- 执行需要PreparedStatement类的两个方法executeQuery和executeUpdate。
- 但是，在它们被调用之前，我们必须使用PteparedStatement类的方法来为”？”参数设定具体的值。
- setString方法、setInt方法等

- 在同一查询编译一次，然后，设置不同的参数值执行多次的情况下，预备语句使得执行更加高效。
- 预备语句，使得只要在查询中使用了用户输入值，即使是只运行一次，预备语句都是执行SQL查询的首选方法。
  - 如果用户输入了某些特殊字符，例如一个单引号，除非我们采取额外工作对用户输入进行检查，否则，生成的SQL语句会出现语法错误。
  - setStrng方法，为我们自动完成检查，并插入需要的转义字符，以确保语法的正确性。

- WARNING: always use prepared statements when taking an input from the user and adding it to a query
  - NEVER create a query by concatenating strings
  - "insert into instructor values(’ " + ID + " ’ , ’ " + name + " ’ ," + " ’ " + dept name + " ’ ," + balance + ") “
  - What if name is “D’ Souza”?

SQL Injection

- Suppose query is constructed using
  - "select * from instructor where name = ’ " + name + "’ "
- Suppose the user, instead of entering a name, enters:
  - X’ or ’ Y’ = ’ Y
- then the resulting statement becomes:
  - "select * from instructor where name = ’ " + "X’ or ’ Y’ = ’ Y" + "’ "
  - which is:
    - select * from instructor where name = ’ X’ or ’ Y’ = ’ Y’

- User could have even used
  - X’ ; update instructor set salary = salary + 10000;
- Prepared stament internally uses:
  "select * from instructor where name = ’ X\’ or \’ Y\’ = \’ Y’
  - Always use prepared statements, with user inputs as parameters
- 比较老的系统，允许多个由分号隔开的语句在一次调用里被执行。
- 此功能正逐渐被淘汰，因为恶意的黑客会利用SQL注入技术插入整个SQL语句。

5.1.1.5 可调用语句

- JDBC还提供了CallableStatement接口，允许用户调用SQL存储的过程和函数。
- CallableStatement cStmtl= conn.prepareCall(“{?=call some_function(?,?,...)}”);
- CallableStatement cStmt2= conn. prepareCall(“{call some_procedure(?,? ,...)}”):
- 函数返回值和过程的对外参数的数据类型，必须先用方法registeroutParameter()注册，它们可以用get方法获取，与结果集用的方法类似。

Example to call the function using JDBC

- CallableStatement stmt=con.prepareCall("{?= call sum4(?,?)}");
- stmt.setInt(2,10);
- stmt.setInt(3,43);
- stmt.registerOutParameter(1,Types.INTEGER);
- stmt.execute();
- The Types class defines many constants such as INTEGER, VARCHAR, FLOAT, DOUBLE, BLOB, CLOB etc.

Example to call the stored procedure using JDBC

- CallableStatement stmt=con.prepareCall("{call insertR(?,?)}");
- stmt.setInt(1,1011);
- stmt.setString(2,"Amit");
- stmt.execute();

5.1.1.6 获取元数据的功能

- 一个Java应用程序，不包含数据库中存储的数据的声明。
- 这些声明是SQL数据定义语言(DDL)的一部分。
- 因此，使用JDBC的Java程序，必须
  - 要么，将关于数据库模式的假设，硬编码到程序中，
  - 要么，直接在运行时，从数据库系统中得到那些信息。
- 后一种方法更可取，因为，它使得应用程序可以更健壮地处理数据库模式的变化。

- 使用executeQuery方法的查询时，查询结果被封装在一个ResultSet对象中。
- ResultSet，有一个getMetaData()方法，它返回一个包含结果集元数据的ResultSetMetaData对象。
- ResultSetMetaData中，又包含查找元数据信息的方法，例如
  - 结果集的列数、某个特定列的名称，或者某个特定列的数据类型。

- ResultSetMetaData rsmd = rs.getMetaData();
- for(int i = 1; i <= rsmd.getColumnCount(); i++) {
  - System.out.println(rsmd.getColumnName(i));
  - System.out.println(rsmd.getColumnTypeName(i));
- }
- 其中，rs是执行查询后，所获得的一个ResultSet实例
- getColumnCount()方法，返回结果关系的元数（属性个数）。
- getColumnName(i)获得属性的名称

DatabaseMetaData 接口

- DatabaseMetaData接口提供了，查找数据库元数据的机制。
- Connection包含一个getMetaData方法，用于返回一个。
- DatabaseMetaData对象，又含有大量的方法，可以用于获取程序所连接的数据库和数据库管理系统的元数据。例如，
  - 数据库管理系统的产品名称和版本号。
  - 查询数据库系统所支持的功能
  - 数据库本身信息的方法。

Metadata (Cont)

- DatabaseMetaData dbmd = conn.getMetaData();
- // Arguments to getColumns（ Catalog, Schema-pattern, Table-pattern, and Column-Pattern ）
  // Returns: One row for each column; row has a number of attributes
  // such as COLUMN_NAME, TYPE_NAME
  // The value null indicates all Catalogs/Schemas.
  // The value “” indicates current catalog/schema
  // The value “%” has the same meaning as SQL like clause，“%”匹配所有的名字，”_”单个字符
- ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");

- while(rs.next()) {
- System.out.println(rs.getString("COLUMN_NAME"), rs.getString("TYPE_NAME"));
- }

- DatabaseMetaData dbmd = conn.getMetaData();
- // Arguments to getTables（Catalog, Schema-pattern, Table-name_pattern, Table-Type）
  // Returns: One row for each table; row has a number of attributes
  // such as TABLE_NAME, TABLE_CAT, TABLE_TYPE, ..
  // The value null indicates all Catalogs/Schemas.
  // The value “” indicates current catalog/schema
  // The value “%” has the same meaning as SQL like clause
  // The last attribute is an array of types of tables to return.

- ResultSet getTables(String catalog,
- String schemaPattern,
- String tableNamePattern,
- String[] types)
- throws SQLException
- Retrieves a description of the tables available in the given catalog.
- Only table descriptions matching the catalog, schema, table name and type criteria are returned. They are ordered by TABLE_TYPE, TABLE_CAT, TABLE_SCHEM and TABLE_NAME.

- Each table description has the following columns:
- TABLE_CAT String => table catalog (may be null)
- TABLE_SCHEM String => table schema (may be null)
- TABLE_NAME String => table name
- TABLE_TYPE String => table type. Typical types are "TABLE", "VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", "SYNONYM".

- REMARKS String => explanatory comment on the table (may be null)
- TYPE_CAT String => the types catalog (may be null)
- TYPE_SCHEM String => the types schema (may be null)
- TYPE_NAME String => type name (may be null)
- SELF_REFERENCING_COL_NAME String => name of the designated "identifier" column of a typed table (may be null)
- REF_GENERATION String => specifies how values in SELF_REFERENCING_COL_NAME are created. Values are "SYSTEM", "USER", "DERIVED". (may be null)

Finding Primary Keys

- DatabaseMetaData dmd = connection.getMetaData();
  // Arguments getPrimaryKeys are(Catalog, Schema, Table)
  // The value “” for Catalog/Schema indicates current catalog/schema
  // The value null indicates all catalogs/schemas
  ResultSet rs = dmd.getPrimaryKeys(“”, “”, tableName);
  while(rs.next()){
  // KEY_SEQ indicates the position of the attribute in the primary key, which is required if a primary key has multiple attributes

- System.out.println(rs.getString(“KEY_SEQ”), rs.getString("COLUMN_NAME"));
  }

5.1.1.7其他特性

- 可更新的结果集(updatahle result sets)
- 它可以从一个在数据库关系上执行的查询中，创建一个可更新的结果集。
- 然后，通过对结果集中的元组进行更新，引起对数据库关系中相应元组的更新。

Transaction Control in JDBC

- By default, **each** SQL statement is treated as a separate **transaction** that is committed automatically
  - **bad idea** for transactions with multiple updates
- Can **turn off** automatic commit on a **connection**
  - conn.**setAutoCommit(false)**;
- Transactions must then be committed or rolled back explicitly
  - conn.**commit**(); conn.**rollback**();
- conn.**setAutoCommit(true)** turns on automatic commit.

- JDBC提供，处理大对象的接口，而不要求在内存中创建整个大对象。
- 为了获取大对象，ResultSet接口提供方法getBlob()和getCloh()，分别返回类型为Blob和Clob的对象。
- 这些对象，并不存储整个大对象，而是存储这些大对象的指针，
  - 即，指向数据库中实际大对象的逻辑指针。
- 从这些对象中获取数据，可以采用getBytes和getSubString方法来实现。

向数据库里存储大对象

- 可以用PreparedStatement类的方法
  - setBlob(int parameterIndex，InputStream inputStream)
- 把一个类型为二进制大对象(blob)的数据库列与一个输入流关联起来。
- 当预备语句被执行时，数据从输入流被读取，然后被写入数据库的二进制大对象中。
- 使用方法setClob，可以设置字符大对象(clob)列，该方法的参数包括，该列的序号和一个字符串流。

行集(row set)功能

- JDBC允许把结果集打包起来，发送给其他应用程序。
- 行集，既可以向后又可以向前扫描，并且可被修改。
- 行集一旦被下载下来，就不再是数据库本身的内容了

JDBC Resources

- JDBC Basics Tutorial
  - https://docs.oracle.com/javase/tutorial/jdbc/index.html

5.1.2 ODBC

- **Open DataBase Connectivity** (ODBC) standard
  - standard for *application* program to *communicate* with a database server.
  - a**pplication program interface** (API) to
    - open a connection with a database,
    - send queries and updates,
    - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC

- 每一个支持ODBC的数据库系统，都提供一个和客户端程序相连接的库
- 当客户端发出一个ODBC API请求，库中的代码就可以和服务器通信，来执行被请求的动作并取回结果。
- 利用ODBC和服务器通信的第一步是
  - 与服务器的连接。
- 为了实现这一步，程序先分配一个SQL的环境，然后，是一个数据库连接句柄。

- void ODBCexample()
  - {
  - RETCODE error;
  - HENV env; /* environment */
  - HDBC conn; /* database connection */
  - SQLAllocEnv (&env);
  - SQLAllocConnect (env, &conn);
  - SQLConnect (conn, "db.yale.edu", SQL_NTS, "avi", SQL_NTS, "avipasswd", SQL_NTS);

- ODBC定义了HENV、HDBC和RETCODE几种类型。
- 程序随后利用SQLConnect打开和数据库的连接，参数包括：
  - 数据库的连接句柄、
  - 要连接的服务器、
  - 用户的身份和密码等。
  - 常数SQL_NTS，表示前面参数是一个以null结尾的字符串。

- 一旦一个连接建立了，C语言就可以通过SQLExecDirect语句,把SQL命令发送到数据库。
- 因为，C语言的变量可以和查询结果的属性绑定，所以，当一个元组被SQLFetch语句取回的时候，结果中相应的属性的值，就可以放到对应的C变量里了。

- {
  - char deptname[80];
  - float salary;
  - int lenOut1, lenOut2;
  - HSTMT stmt;
  - char * sqlquery = "select dept name, sum (salary)
    - from instructor
    - group by dept name";
  - SQLAllocStmt(conn, &stmt);
  - error = SQLExecDirect(stmt, sqlquery, SQL NTS);

- SQLBindCol做这项工作；
- 在SQLBindCol函数里面，
  - 第二个参数，代表选择属性中哪一个位置的值，
  - 第三个参数，代表SQL应该把属性值转化成什么类型的C变量。
  - 再下一个参数，给出了存放变量的地址。
  - 对于变长类型,如字符数组，最后两个参数还要给出变量的最大长度和一个存放取回元组实际长度的地址。
  - 如果长度域返回一个负值，那么代表着这个值为空（null）。

- 对于定长类型的变量，如整型或浮点型，最大长度的域被忽略，
- 然而，当长度域返回一个负值，时表示该值为空值。

- if (error == SQL_SUCCESS) {
  - SQLBindCol(stmt, 1, SQL_C_CHAR, &deptname , 80, &lenOut1);
  - SQLBindCol(stmt, 2, SQL_C_FLOAT, &salary, 0 , &lenOut2);
  - while (SQLFetch(stmt) == SQL SUCCESS) {
    - printf (" %s %g\n", depthname, salary);
  - }
- }

- 在会话结束的时候，
  - 程序释放语句的句柄，
  - 断开与数据库的连接，
  - 释放连接和SQL环境句柄。
- 好的编程风格要求，检查每一个函数的结果，确保它们没有错误。

- SQLFreeStmt(stmt, SQL DROP);
- }
- SQLDisconnect(conn);
- SQLFreeConnect(conn);
- SQLFreeEnv(env);

ODBC 预备语句

- ODBC可以创建带有参数的SQL语句，
  - 例如，insert into department values(?，？，？)；
- 问号是为将来提供值的占位符。
- 上面的语句可以先被“准备”，也就是在数据库中先编译，
- 然后，可以通过为占位符提供具体值，来反复执行。

ODBC 元数据操作

- ODBC为各种不同的任务定义了函数，例如
  - 查找数据库中所有的关系，
  - 查找数据库中某个关系的列的名称和类型，
  - 或者一个查询结果的列的名称和类型。

ODBC 对事务的操作

- 在默认情况下，每一个SQL语句，都被认为是一个自动提交的独立事务。
- 调用SQLSetConnectOption(conn, SQL_AUTOCOMMIT，0)，可以关闭连接conn的自动提交，
  - 事务必须通过显式地调用SQLTransact(conn，SQL_COMMIT)来提交
  - 或通过显式地调用SQLTransact(conn，SQL_ROLLBACK)来回滚。

符合性级别

- ODBC标准定义了符合性级别（conformance level），用于指定标准定义的功能的子集。
- 一个ODBC实现，可以仅提供核心级功能，也可以提供更多的高级功能（level l或level 2）。
  - level l，支持取得目录的有关信息，
    - 例如，什么关系存在，它们的属性是什么类型的等。
  - level 2，提供更多的功能，
    - 例如，发送和提取参数值数组，以及检索有关目录的更详细信息的能力。

ADO．NET

- ADO.NET API是为Visual Basic.NET和C#语言设计的，它提供了一系列访问数据的函数，
- ADO.NET API可以
  - 访问SQL查询的结果，
  - 以及元数据，
- 可以使用ADO.NET API来访问支持ODBC的数据库，这时．ADO.NET的函数调用被转成ODBC调用。
- ADO.NET API也可以用在某些非关系数据源上
- 如，微软的OLE-DB, XML

5.1.3 Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/I,
- A language to which SQL queries are embedded is referred to as a host language, and the SQL structures permitted in the host language comprise embedded SQL.
- The basic form of these languages follows that of the System R embedding of SQL into PL/I.

- 使用宿主语言，写出的程序，可以通过嵌入式SQL的语法访问和修改数据库中的数据。
- 一个使用嵌入式SQL的程序，在编译前，必须先由一个特殊的预处理器进行处理。
- The preprocessor replaces embedded SQL requests
  - with host-language declarations and procedure calls
  - that allow runtime execution of the database accesses.
- the resulting program is compiled by the host-language compiler.
- This is the main distinction between embedded SQL

- 在JDBC中，SQL语句是在运行时被解释的
- 当使用嵌入式SQL时，SQL请求被替换成宿主语言的声明和过程调用，一些SQL相关的错误（包括数据类型错误）可以在编译过程中被发现。
- 为使预处理器识别嵌入式SQL请求，我们使用EXEC SQL语句，格式如下：
  - EXEC SQL <embedded SQL statement >;
- 应用程序中，合适的地方插入SQL INCLUDE SQLCA语句，表示
  - 预处理器应该在此处，插入特殊变量以用于程序和数据库系统间的通信。

- EXEC SQL statement is used to identify embedded SQL request to the preprocessor
- EXEC SQL <embedded SQL statement >;
- Note: this varies by language:
  - In some languages, like COBOL, the semicolon is replaced with END-EXEC
  - In Java embedding uses # SQL { …. };

Embedded SQL (Cont.)

- Before executing any SQL statements, the program must first connect to the database.
- This is done using:
- EXEC-SQL connect to server user user-name using password;
  - Variables of the host language can be used within embedded SQL statements.
  - They are preceded by a colon (:) to distinguish from SQL variables (e.g., :credit_amount)

- Variables used as above must be declared within DECLARE section.
- The syntax for declaring the variables follows the usual host language syntax.
- EXEC-SQL BEGIN DECLARE SECTION;
- int credit-amount ;
- EXEC-SQL END DECLARE SECTION;

- 为了写出关系查询语句，我们使用声明游标(declare cursor)语句。
- 然而，这时并不执行查询，而程序必须用open和fetch语句得到查询结果元组。

Embedded SQL (Cont.)

- To write an embedded SQL query, we use the
- declare c cursor for <SQL query> statement.
- The variable c is used to identify the query
- Example:
  - Specify the query in SQL as follows:
  - EXEC SQL
  - declare c cursor for
    select ID, name
    from student
    where tot_cred > :credit_amount;
  - END_EXEC

Embedded SQL (Cont.)

- The open statement for our example is as follows:
- EXEC SQL open c ;
- This statement causes the database system to execute the query and to save the results within a temporary relation.
- The query uses the value of the host-language variable credit-amount at the time the open statement is executed.

- The fetch statement causes the values of one tuple in the query result to be placed on host language variables.
- EXEC SQL fetch c into :si, :sn ;
- END_EXEC
- Repeated calls to fetch get successive tuples in the query result

- 虽然关系在概念上是一个集合，查询结果中的元组还是有一定的物理顺序的。
- 执行SQL的open语句后，游标指向结果的第一个元组。
- 执行一条fetch语句后，游标指向结果中的下一个元组。
- 当后面不再有待处理的元组时，SQLCA中变量SQLSTATE被置为’02000’（意指“不再有数据”）；
- 访问该变量的确切的语法，依赖于所使用的特定数据库系统。

Embedded SQL (Cont.)

- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available
- The close statement causes the database system to delete the temporary relation that holds the result of the query.
- EXEC SQL close c ;
- Note: above details vary with language.
- For example, the Java embedding defines Java iterators to step through result tuples.

Updates Through Embedded SQL

- Embedded SQL expressions for database **modification** (**update**, **insert**, and **delete**)
- Can **update** tuples **fetched** by *cursor* by **declaring** that the cursor is **for** *update*

```
EXEC SQL
    declare c cursor for
    select *
    from instructor
    where dept_name = 'Music'
    for update
```

- We then iterate through the tuples by performing fetch operations on the cursor , and after fetching each tuple， we execute the following code:
- EXEC SQL
- update instructor
  set salary = salary + 1000
  where current of c

- 可以用EXEC SQL COMMIT语句来提交事务，
- 或者用EXEC SQL ROLLBACK进行回滚。
- 嵌入式SQL的查询，一般是在编写程序时被定义的。
- 但是，在某些比较罕见的情况下，查询需要在运行时被定义。例如，
  - 一个应用程序可能会让用户来，指定某个关系的一个或多个属性上的选择条件，
  - 然后，在运行时，用用户选择的属性的条件，来构造SQL查询的where子句。

- 此种情况下，可以使用
  - EXEC SQL PREPARE<query-name> FROM:<variable>”
- 在运行时，构造和准备查询字符串；
- 并且，可以在查询名字上打开一个游标。

SQLJ

- JDBC is overly dynamic, errors cannot be caught by compiler
- SQLJ: embedded SQL in Java
- SQLJ使用句法#sql代替EXEC
- SQLJ并且不使用游标，用java循环体接口，来获取查询结果。
- 因此，执行查询的结果，被储存在Java循环体里，
- 然后，利用Java循环体接口中的next()方法，采逐步遍历结果元组。

```
#sql iterator deptInfoIter ( String dept_name, int avgSal);
  deptInfoIter iter = null;
  #sql iter = { select dept_name, avg(salary)
                        from instructor
                        group by dept_name };
while (iter.next()) {
          String deptName = iter.dept_name();
      int avgSal = iter.avgSal();
      System.out.println(deptName + " " + avgSal);
```

- IBM的DB2和Oracle都支持SQLJ
- 并且，提供从SQLJ代码到JDBC代码的转换器。
- 该转换器，可以在编译时，连接数据库，来检查查询的语法是否正确，
- 并用，来确保查询结果的SQL类型与所赋值的Java变量类型相一致。

5.2 Functions and Procedures

- SQL:1999 supports functions and procedures
- 开发者编写他们自己的函数和过程，把它们存储在数据库里，并在SQL语句中调用。
  - Functions/procedures can be written in SQL itself, or in an external programming language (e.g., C, Java).
  - Functions written in an external languages are particularly useful with specialized data types such as images and geometric objects.
    - Example: functions to check if polygons overlap, or to compare images for similarity.

- Some database systems support table-valued functions, which can return a relation as a result.

- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.

- SQL函数和过程，允许将“业务逻辑”作为数据库的存储过程，记录在数据库中，并在数据库内执行。
- 尽管业务逻辑能够被写成程序设计语言过程，并完全存储在数据库以外（存储在客户端）
- 但，把它们定义成，数据库中的存储过程（放在服务器），有几个优点。例如，
  - 它允许多个应用访问这些过程，
  - 允许当业务规则发生变化时，进行单个点的改变，而不必改变应用系统的其他部分。
- 客户端的应用程序代码，可以调用存储过程，而不是直接更新数据库关系。

5.2.1 Declaring and Invoking SQL Functions and Procedures

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.
-
```
create function dept_count (dept_name varchar(20))
        returns integer
        begin
        declare d_count  integer;
            select count (* ) into d_count
            from instructor
            where instructor.dept_name = dept_count.dept_name
        return d_count;
```

- The function dept_count can be used to find the department names and budget of all departments with more that 12 instructors.
- ```
  select dept_name, budget
  from department
  where dept_count (dept_name ) > 12
  ```
  ```

SQL functions (Cont.)

- Compound statement: **begin ... end**
  - May contain *multiple* SQL statements between **begin** and **end**.
- **returns** -- indicates the variable-type that is returned (e.g., integer)
- **return** **--** specifies the *values* that are to be returned as result of invoking the function
- in fact, SQL function are *parameterized views* that generalize the regular notion of views by allowing parameters.

Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all instructors in a given department
- create function instructor_of (dept_name varchar(20))
- returns table (
- ID varchar(5),
  name varchar(20),
  dept_name varchar(20),
  salary numeric(8,2))
- return table

- (select ID, name, dept_name, salary
  from instructor
  where instructor.dept_name = instructor_of.dept_name)
- 注意，使用函数的参数时,需要加上函数名作为前缀（instructor_of.dept_name）
- Usage
- select *
  from table (instructor_of (‘Music’))

SQL Procedures

- The *dept_count* **function** could instead be written as **procedure**:

```
create procedure dept_count_proc (in dept_name
varchar(20), out d_count integer)
   begin
      select count(*) into d_count
      from instructor
      where instructor.dept_name =
dept_count_proc.dept_name
    end
```

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the call statement.
- declare d_count integer;
  call dept_count_proc(‘Physics’, d_count);
- Procedures and functions can be invoked also from dynamic SQL
- SQL:1999 allows more than one function/procedure of the same name (called name overloading), as long as
  - the number of arguments differ,
  - or at least the types of the arguments differ

5.2.2 Language Constructs for Procedures & Functions

- SQL supports **constructs** that gives it almost all the *power* of a general-purpose programming language.
  - Warning: **most** database systems implement their own *variant* of the standard syntax below.
- *SQL标准*中，*处理*这些**constructs** 的部分，称为**持久存储模块**(**Persistent Storage Module**，PSM)。
- 变量声明，**declare**语句，变量可以是*任意*的合法SQL**类型**。
- 使用**set**语句进行*赋值*。

- Compound statement: begin … end,
  - May contain multiple SQL statements between begin and end.
  - Local variables can be declared within a compound statements
- begin atomic...end的复合语句
  - 可以确保其中包含的所有语句，作为单一的事务来执行。

- While and repeat statements:
  - while boolean expression do
    - sequence of statements ;
  - end while
  - repeat
    - sequence of statements ;
  - until boolean expression
  - end repeat

Language Constructs (Cont.)

- For loop
  - Permits iteration over all results of a query
- Example: Find the budget of all departments

```
declare n integer default 0;
 for r as
        select budget from department
 do
              set n = n + r.budget
 end for
```

- 程序每次获取查询结果的一行，并存入for循环变量（在上面例子中指r）中。
- 语句leave可用来退出循环，
- iterate表示，跳过剩余语句，从循环的开始进入下一个元组。

Language Constructs (Cont.)

- **Conditional** statements (**if-then-else**)
  SQL:1999 also supports a **case** statement similar to C case statement
- Example *function*: registers student after ensuring classroom capacity is not exceeded
  - Returns 0 on success and -1 if capacity is exceeded
  - See book (page 102) for details

- Signaling of exception conditions, and declaring handlers for exceptions
- declare out_of_classroom_seats condition
  declare exit handler for out_of_classroom_seats
  begin
  …
  .. signal out_of_classroom_seats
  end
  - The handler here is exit -- causes enclosing begin..end to be exited
  - Other actions possible on exception

- SQL程序语言支持发信号，通知异常条件(exception condition)，以及声明句柄(handler)来处理异常
- 在begin和end之间的语句，可以执行signaI out_of_classroom_seats来引发一个异常。
- 句柄说明，如果条件发生，将会采取动作终止begin end中的语句。
- 另一个可选的动作将是continue，它继续从引发异常的语句的下一条语句开始执行。
- 除了明确定义的条件，还有一些预定义的条件，比如，sqlexception、sqlwarning和not found.

过程和函数的非标准语法

- Oracle的PL/SQL与标准语法不同的一些方面
- create or replace function dept_count(dept_name in instructor.dept_name%type)
- return integer
- as
  - d_count integer;
  - begin
    - select count(*) into d_count
    - from instructor
    - where instructor.dept_name = dept_name;
    - return d_count;

- PL/SQL allows a type to be specified as the type of an attribute of a relation, by adding the suffix %type.
- PL/SQL does not directly support the ability to return a table,
- although there is an indirect way of implementing this functionality by creating a table type.

5.2.3 External Language Routines

- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- SQL:1999 allows the definition of procedures in an imperative programming language, (Java, C#, C or C++) which can be invoked from SQL queries.
- Functions defined in this fashion can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL can be executed by these functions.

- Declaring external language procedures and functions
- create procedure dept_count_proc(in dept_name varchar(20), out count integer)
  language C
  external name ’ /usr/avi/bin/dept_count_proc’
- create function dept_count(dept_name varchar(20))
  returns integer
  language C
  external name ‘/usr/avi/bin/dept_count’

- 外部语言过程，
  - 需要处理参数（包含In和out参数）和返回值中的空值，
  - 还需要，传递操作失败 / 成功的状态，以方便对异常进行处理。
- 这些信息可以通过几个额外的参数来表示：
  - 一个指明失败 / 成功状态的sqlstate值、
  - 一个存储函数返回值的参数，
  - 以及一些指明每个参数 / 函数结果的值是否为空的指示器变量。

External Language Routines (Cont.)

- **Benefits** of external language functions/procedures:
  - more *efficient* for many operations, and more *expressive* power.
- **Drawbacks**
  - *Code* to implement function may need to be **loaded** into *database system* and **executed** in the database system's *address space*.
    - **risk** of accidental *corruption* of database structures
    - **security** risk, allowing users access to **unauthorized** data

- There are alternatives, which give good security at the cost of potentially worse performance.
- Direct execution in the database system’s space is used when efficiency is more important than security.

Security with External Language Routines

- To deal with security problems, we can do on of the following:
  - Run external language functions/procedures in a separate process, with no access to the database process' memory.
    - Parameters and results communicated via inter-process communication
    - 进程间通信的时间代价相当高；
    - 在典型的CPU体系结构中，一个进程通信所需的时间，可以执行数万到数十万条指令。
    - 避免进程间通信，能够大大降低函数调用的时间代价。

- Use sandbox techniques
  - That is, use a safe language like Java, C# , which cannot be used to access/damage other parts of the database code.
  - 在数据库进程本身的沙盒(sandbox)内执行代码。
  - 沙盒允许Java或c#代码，访问它自己的内存区域，但阻止代码直接，在查询执行过程的内存中，做任何读操作或者更新操作，或者访问文件系统中的文件。
- Both have performance overheads

沙盒sandbox原理

- 沙盒也叫沙箱，sandbox。
- 在计算机领域指一种虚拟技术，且多用于计算机安全技术。
- 其原理是通过重定向技术，把程序生成和修改的文件定向到自身文件夹中。
- 当某个程序试图发挥作用时，安全软件可以先让它在沙盒中运行，如果含有恶意行为，则禁止程序的进一步运行，而这不会对系统造成任何危害。

- Many database systems support both above approaches， as well as direct executing in database system address space.
- 当今的一些数据库系统，支持外部语言例程，在查询执行过程中的沙盒里运行。例如，
  - Oracle和IBM DB2允许Java函数作为数据库过程中的一部分运行。
  - Microsoft SQL Server允许将外部语言过程编译成通用语言运行程序(CLR)，来在数据库过程中执行；这样的过程，可以用C#或Visual Basic编写。
  - PostgreSQL允许在Perl、Python和Tcl等多种语

5.3 Triggers

Triggers

- A trigger is a statement， that is executed automatically by the system， as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.它被分解为
    - 一个引起触发器被检测的事件
    - 一个触发器执行必须 满足的条件。
  - Specify the actions to be taken when the trigger executes.

- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  - Syntax illustrated here may not work exactly on your database system; check the system manuals
- 一旦我们把一个触发器输入数据库，只要指定的事件发生，相应的条件满足，数据库系统就有责任去执行它。

5.3.1 对触发器的需求

- 触发器可以用来实现，未被SQL约束机制指定的某些完整性约束。
- 当满足特定条件时，对用户发警报或自动开始执行某项任务。
- 例如，
  - 设计一个触发器，只要有元组被插入takes关系中，就更新student关系中选课的学生所对应的元组

- 另一个，假设一个仓库希望每种物品的库存保持一个最小量；当某种物品的库存少于最小值的时候，自动发出一个订货单。
  - 在更新某种物品的库存的时候，触发器会比较这种物品的当前库存和它的最小库存，
  - 如果库存数量等于或小于最小值，就会生成一个新的订单。

- 注意，触发器系统通常不能执行数据库以外的更新，
- 因此，在上面的库存补充的例子中，
  - 我们不能用一个触发器，去直接在外部世界下订单，
  - 而是，在存放订单的关系中，添加一个关系记录。
- 我们必须另外创建一个持久运行的系统进程，来周期性扫描该关系，并订购产品。
- 某些数据库系统，提供了内置的支持，可以使用上述方法，从SQL查询和触发器中发送电子邮件

5.3.2 Triggering Events and Actions in SQL

- **Triggering event** can be **insert**, **delete** or **update**
- Triggers on *update* can be restricted to **specific attributes**
  - For example, ***after*** **update** **of** *takes* **on** *grade*
- Values of attributes *before* and *after* an update can be referenced
  - **referencing old row as** : for deletes and updates
  - **referencing new row as** : for inserts and updates

触发器举例

- 如何使用触发器来确保关系section中属性time_slot_id的参照完整性
- create trigger timeslot_check1 after insert on section
  - referencing new row as nrow
  - for each row
  - when (nrow.time_slot_id not in (
    - select time_slot_id
    - from time_slot)) /* time slot id not present in time slot */
  - begin
    - rollback

- create trigger timeslot_check2 after delete on time_slot
- referencing old row as orow
- for each row
- when (orow.time_slot_id not in (
  - select time_slot_id
  - from time_slot) /* last tuple for time slot id deleted from time slot */
- and orow.time_slot_id in (
  - select time slot id
  - from section)) /* and time slot id still

- 一个SQL插入语句，可以向关系中，插入多个元组，
  - 在触发器代码中，for each row语句，可以显式地在每一个被插入的元组上，进行循环。
- referencing new row as语句，建立了一个行变量nrow(称为过渡变量(transition variable)。
  - 用来在插入完成后，存储所插入行的值。
- when语句指定一个条件。
  - 仅对于满足条件的元组，系统才会执行触发器中的其余部分。
  - begin atomic…end语句，用来将多行SQL语句集成为一个复合语句。

Trigger to Maintain credits_earned value

- create trigger credits_earned after update of takes on (grade)
  referencing new row as nrow
  referencing old row as orow
  for each row
  when nrow.grade <> 'F' and nrow.grade is not null
  and (orow.grade = 'F' or orow.grade is null)
  begin atomic
  update student
  set tot_cred= tot_cred +
  (select credits
  from course
  where course.course_id= nrow.course_id)
  where student.id = nrow.id;

- Triggers can be activated before an event, which can serve as extra constraints.
- 在事件之前被执行的触发器，可以作为避免非法更新、插入或删除的额外约束。
- 为了避免执行非法动作而产生错误，事件之前触发器，可以采取措施来纠正问题，使更新、插入或删除操作合法化。
- For example, convert blank grades to null.
- create trigger setnull_trigger before update of takes
  referencing new row as nrow
  for each row

小心设计触发器条件

```
create trigger reorder after update of amount on inventory
referencing old row as orow, new row as nrow
for each row
when nrow.level <= (select level
                    from minlevel
                    where minlevel.item = orow.item)
and orow.level > (select level
                  from minlevel
                  where minlevel.item = orow.item)
begin atomic
    insert into orders
        (select item, amount
         from reorder
         where reorder.item = orow.item);
end;
```

- 许多数据库系统，支持各种别的触发器事件，比如
  - 当一个用户（应用程序）登录到数据库（即打开一个连接）的时候，
  - 或者当系统停止的时候，
  - 或者当系统设置改变的时候。

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use for each statement instead of for each row
  - Use referencing old table or referencing new table to refer to temporary tables (called transition tables) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

- 过渡表不能用于before触发器，但是可以用于after触发器，
- 无论它们是语句触发器还是行触发器。
- 在过渡表的基础上，一个单独的SQL语句就可以用来执行多个动作。
- 触发器可以设为有效或者无效：
  - 默认情况下，它们在创建时是有效的，
  - 但是可以通过使用alter trigger trigger_name disable将其设为无效。
  - 设为无效的触发器，可以重新设为有效。
  - 通过使用命令drop trigger trigger_name，触发

5.3.3 When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - Maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called change or delta relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication

- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger

When Not To Use Triggers (Cont.)

- Risk of unintended execution of triggers, for example, when
  - Loading data from a backup copy，
  - Replicating updates at a remote site，
  - 在该情况下，触发器动作已经执行了，通常不应该再次执行。
- Trigger execution can be disabled before such actions.
- 对于要接管主系统的备份复制系统，
  - 触发器应该一开始就设为无效，而在备份站点接管了主系统的业务后，再设为有效。

- 作为取代的方法，一些数据库系统，允许触发器定义为not for replication，保证触发器不会在数据库备份的时候，在备份站点执行。
- 另一些数据库系统，提供了一个系统变量，用于指明该数据库是一个副本，数据库动作在其上是重放；
  - 触发器会检查这个变量，如果为真则退出执行
- 这两种解决方案，都不需要显式地将触发器设为失效或有效。

- Other risks with triggers Error :
  - leading to failure of critical transactions that set off the trigger
  - Cascading execution
- 一个触发器的动作可以引发另一个触发器。
- 在最坏的情况下，这甚至会导致一个无限的触发链。例如，
  - 假设在一个关系上的插入触发器里有一个动作引起在同一关系上的另一个（新的）插入，
  - 该新插入动作也会引起另一个新插入，如此无穷循环下去。

- 有些数据库系统，会限制这种触发器链的长度（例如16或32），把超过此长度的触发器链看作是一个错误。
- 另一些系统，把引用特定关系的触发器，标记为错误，对该关系的修改，导致了位于链首的触发器被执行。

- 触发器是很有用的工具，但是，如果有其他候选方法就最好别用触发器。
- 很多触发器的应用，都可以用适当的存储过程来替换