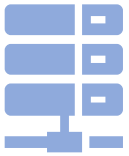
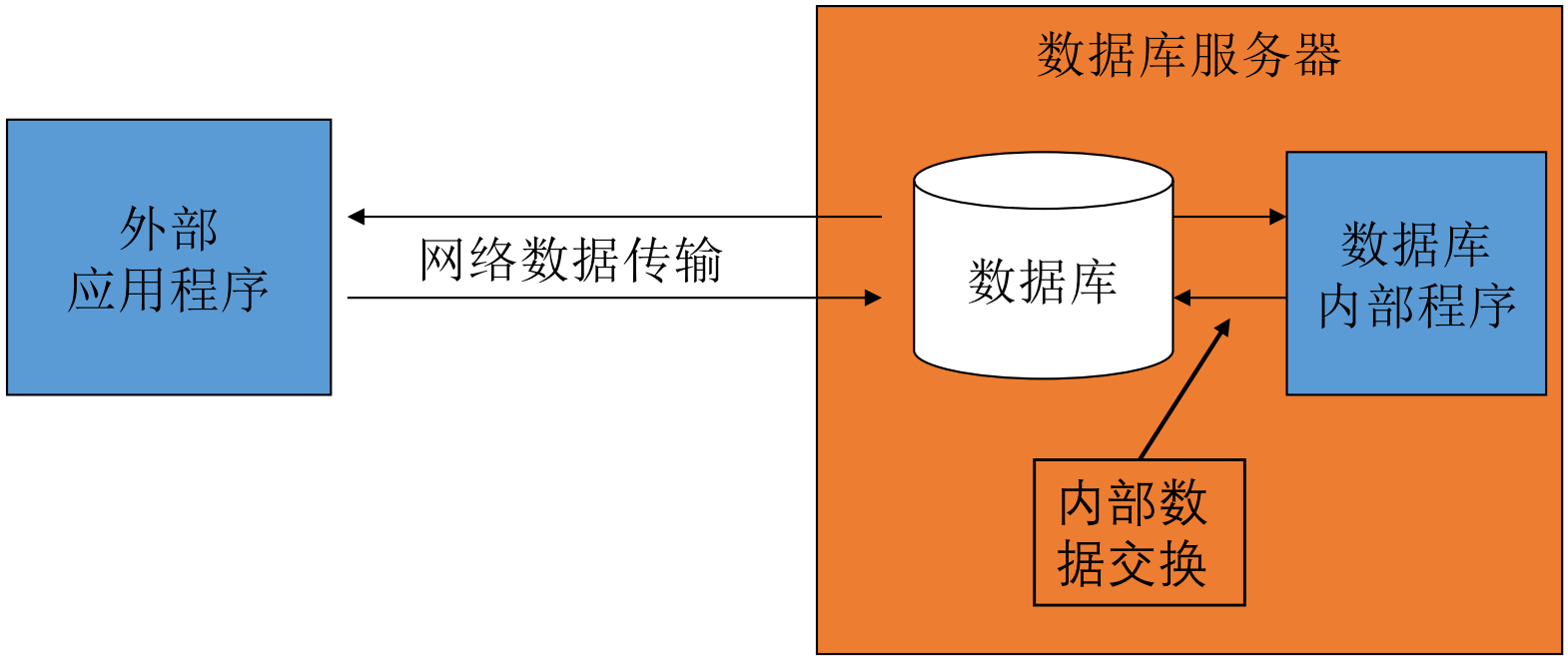


第四章　数据库编程

数据库程序开发

- SQL语言主要用于数据查询和更新
  - ✓由于缺乏变量定义、流程控制等高级编程语言特性，难以担负复杂的数据分析处理任务
  - ✓代码重用性差，运行效率低
- 对数据的分析处理主要由数据库应用程序来完成
- 数据库应用程序的两种形式：
  - ✓数据库内部应用程序
  - ✓数据库外部应用程序

数据库程序开发



外部应用程序开发灵活，功能与可移植性更强

数据库内部程序执行效率更高，适用于例行的数据分析维护功能

数据库内部编程

- 数据库内部程序的常见形式：
  - ✓存储过程
  - ✓函数
  - ✓触发器
- 下面以存储过程为例了解数据库编程要点

数据库内部编程

存储过程

- ✓由过程化SQL语句书写的过程，经编译和优化后存储在数据库服务器中，使用时只要调用即可。
- ✓通常将多次重复执行的代码段编写成一个过程（procedure or function），保存在数据库中

存储过程的优点

- ✓是SQL和模块化编程的结合，能够完成复杂业务功能
- ✓在创建的时候进行预编译，可以提高SQL执行效率
- ✓位于数据库服务器上，调用的时候无需通过网络传输大量数据
- ✓可以做为一种安全机制来加以充分利用。例如参数化的存储过程可以防止SQL注入式的攻击

数据库内部编程

存储过程

创建（MySQL）

```
CREATE PROCEDURE procedure_name (
        [ IN | OUT | INOUT ] param_name type [ ,... ]
)
[ BEGIN ]
            sql_statement
[ END ]
```

- procedure_name：数据库服务器合法的对象标识
- param_name：用名字来标识调用时给出的参数值，必须指定值的数据类型。参数也可以定义输入参数、输出参数或输入/输出参数。默认为输入参数。
- sql_statement：是一个过程化SQL块。包括声明部分和可执行语句部分

数据库内部编程

存储过程

删除与调用（MySQL）

DROP PROCEDURE prodedure_name

CALL procedure_name (param_name type [,...])

- 使用CALL方式激活存储过程的执行；
- 数据库服务器支持在过程体中调用其他存储过程

数据库内部编程

- 存储过程
  - ✓变量
  - ✓流程控制
  - ✓常用数据库函数与命令…

  自学
  - ✓嵌入式SQL
  - ✓动态SQL
  - ✓游标编程

数据库内部编程

存储过程

- ✓数据库提供过程化编程语言
  - 变量
  - 流程控制
  - 常用数据库函数与命令…

  自学
- ✓在过程化编程中嵌入SQL语句

```
declare v_score int;
select score into v_score from sc where cno=55 and stuid=001;
Set v_score=v_score+1;
Update sc set score=v_score where cno=55 and stuid=001;
```

数据库内部编程

存储过程

✓动态SQL

- 根据用户输入参数和/或数据库状态，动态确定程序中的SQL语句内容
- Prepare：组装SQL语句
- Exceute：动态执行SQL语句

```
CREATE PROCEDURE count_field(IN FIELDNAME VARCHAR(255), IN
FIELDVALUE INT)
BEGIN
        SET @sql = CONCAT('SELECT COUNT(*) FROM stu WHERE ‘,
                FIELDNAME, ' = ?’);
        PREPARE stmt FROM @sql;
        SET @fieldvalue = FIELDVALUE;
        EXECUTE stmt USING @fieldvalue;
        DEALLOCATE PREPARE stmt;
END;
```

数据库内部编程

存储过程

✓游标编程

- SQL语句以关系为操作对象和操作结果
- 例如由 SELECT 语句返回的行集包括满足该语句的 WHERE 子句中条件的所有行
- 应用程序有时需要遍历关系表，对每行数据进行单独处理。SQL语言不能实现这样的操作要求。
- 应用程序需要一种机制以便每次处理一行或一部分行。游标就是提供这种机制的对SELECT查询结果集的一种扩展

数据库内部编程

存储过程

✓游标可以对查询语句返回的行结果集中的每一行进行操作。

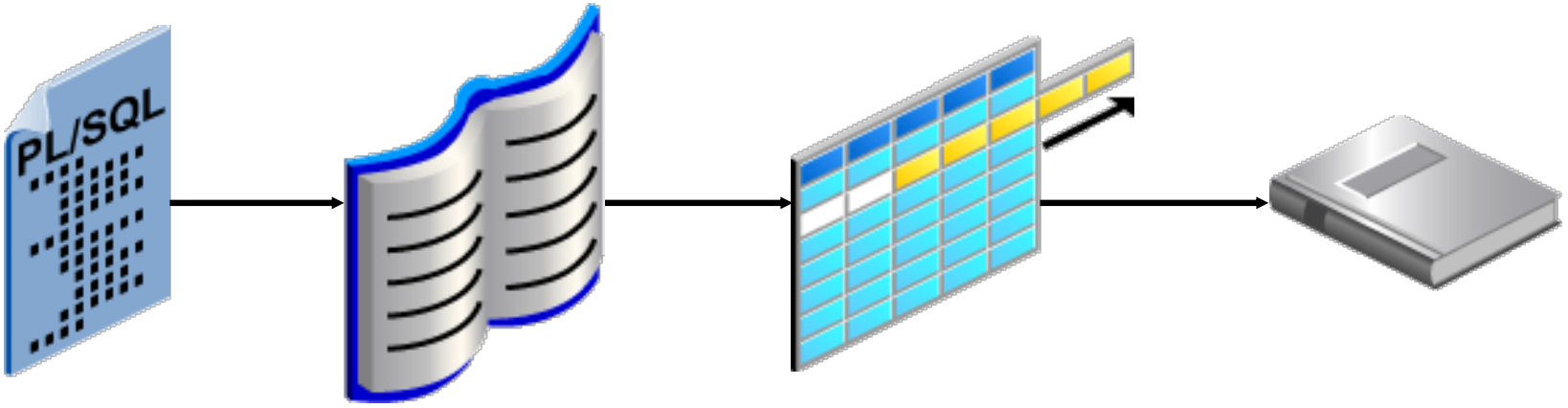
主要功能包括：

- 定位到结果集中的指定行
- 从结果集的当前位置检索一行或多行
- 可对结果集中当前位置的行进行数据修改
- 可以显示其它用户对结果集中的数据库数据进行的数据更改

数据库内部编程

存储过程

✓游标编程



定义并声明游标变量
declare

打开游标变量
open

从结果集里抽取记录
fetch

关闭游标变量
close

1 2 3 4

游标编程示例

```
CREATE PROCEDURE stu_count(c_age int)
BEGIN
        DECLARE p_age int; # 声明变量
        DECLARE p_c int;
        -- 声明游标结束判断变量，默认值为0;
        DECLARE fetchSeqOk boolean DEFAULT 0;
        DECLARE my_cursor CURSOR for select age FROM t_user; -- 定义游标
        -- 游标执行结束时将会设置fetchSeqOk 变量为1
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET fetchSeqOk = 1;
        -- 在MySql中，造成游标溢出时会引发mysql预定义的NOT FOUND错误
        SET p_c=0;
        # 打开游标
        OPEN my_cursor;
                WHILE fetchSeqOk=0 DO. -- 判断是不是到了最后一条数据
                        fetch my_cursor into p_age; -- 游标改变位置指向下一行
，取下一行数据
                        IF p_age<c_age THEN
                           SET p_c=p_c+1;
                        END IF;
                END WHILE;
                Select p_c as concat('小于', c_age, '岁的总人数'); -- 输出结果
        CLOSE my_cursor;  -- 关闭游标，释放内存
END
```

数据库内部编程示例

- 自定义函数

需求：

输入：字符串

输出：第一个字母大写，后面的字母小写

实现：

```
CREATE FUNCTION capitalize(input_string VARCHAR(255))
RETURNS VARCHAR(255)
BEGIN
   DECLARE output_string VARCHAR(255);
   SET output_string = CONCAT(UPPER(LEFT(input_string, 1)),LOWER(SUBSTRING(input_string, 2)));
   RETURN output_string;
END
```

使用： select capitalize(’samPLeSTring’);

select capitalize(stu_name) from stu;

数据库应用系统开发

- 数据库访问与Python MySQL Client
- 对象关系映射--ORM

数据库访问

■各种高级编程语言都提供访问和操作数据库的类库

■使用类库进行数据库开发的基本过程：

JDBC:

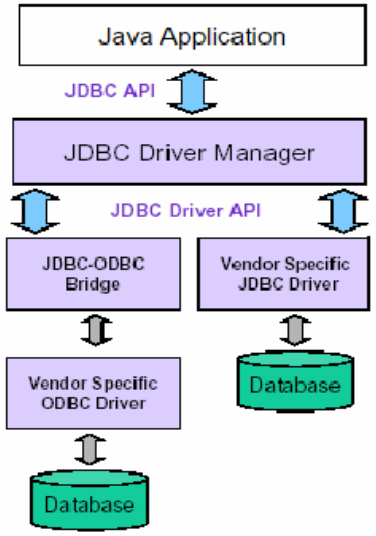
1.Establish a **Connection**

2.Create SQL **Statements**

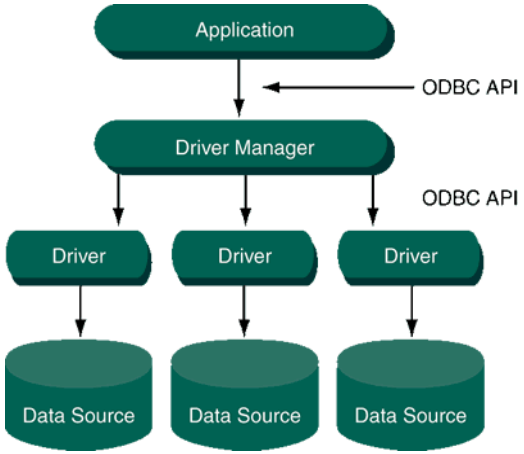
3.Execute **SQL** Statements

4.GET **ResultSet**

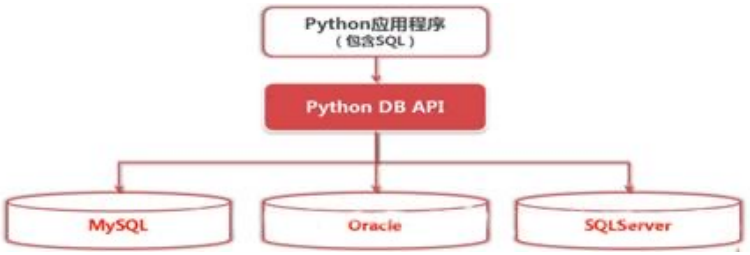
5.**Close** connections



JDBC

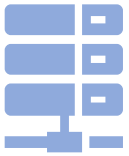


ODBC

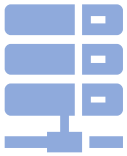


Python

DB API



```
import java.sql.*;
class Test  {
   public static void main(String[] args)  {
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); //dynamic loading of driver
         String filename = "c:/db1.mdb"; //Location of an Access database
         String database = "jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=";
         database+= filename.trim() + ";DriverID=22;READONLY=true}"; //add on to end
         Connection con = DriverManager.getConnection( database ,"","");
         Statement s = con.createStatement();
         s.execute("create table TEST12345 ( firstcolumn integer )");
         s.execute("insert into TEST12345 values(1)");
         s.execute("select firstcolumn from TEST12345");
         ResultSet rs = s.getResultSet();
         if (rs != null) // if rs == null, then there is no ResultSet to view
         while ( rs.next() ) // this will step through our data row-by-row
         {   /* the next line will get the first column in our current row's ResultSet
            as a String ( getString( columnNumber) ) and output it to the screen */
            System.out.println("Data from column_name: " + rs.getString(1) );
         }
         s.close(); // close Statement to let the database know we're done with it
         con.close(); //close connection
       }
      catch (Exception err) { System.out.println("ERROR: " + err);  }
    }
 }
```

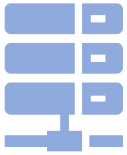


Python3 MySQL Client

■什么是PyMySQL

- PyMySQL 是在 Python3.x 版本中用于连接 MySQL 服务器的一个库，Python2 中则使用 mysqldb。
- PyMySQL 遵循 Python 数据库 API v2.0 规范，并包含了 pure-Python MySQL 客户端库。
- PyMySQL安装：

```
pip3 install PyMySQL
```



数据库连接

■连接数据库前，请先确认以下事项

- 在本地或者云数据库已经创建了数据库 TESTDB.
- 记得自己数据库的用户名和密码
- Python环境已经安装了PyMySQL库
- 下面我们以TESTDB数据库为示例演示数据库的连接、建表、插入、查询、更新、删除等方法。

数据库连接

■连接Mysql的TESTDB数据库

- 新建一个python文件，写入如下内容

```python
import pymysql

# 打开数据库连接
db = pymysql.connect(host='localhost',
                     user='root',
                     password='password', # 填你自己的password
                     database='testdb',
                     port=3360 # 端口
                     )

# 使用 cursor() 方法创建一个游标对象 cursor
cursor = db.cursor()

# 使用 execute()  方法执行 SQL 查询
cursor.execute("SELECT VERSION()")

# 使用 fetchone() 方法获取单条数据.
data = cursor.fetchone()

print ("Database version : %s " % data)

# 关闭数据库连接
db.close()
```

✓ 0.0s Python

- 执行以上代码输出结果如下（具体版本可能会有所不同）：

```
Database version : 8.0.32
```

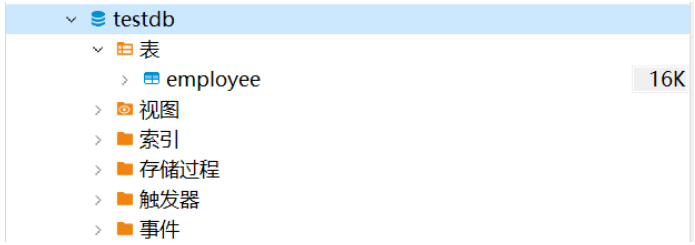
创建数据库表

- 如果数据库连接存在我们可以使用execute()方法来为数据库创建表，如下所示创建表EMPLOYEE：

```python
import pymysql

# 打开数据库连接
db = pymysql.connect(host='localhost',
                     user='root',
                     password='password', # 填你自己的password
                     database='testdb',
                     port=3360 # 端口
                     )

# 使用 cursor() 方法创建一个游标对象 cursor
cursor = db.cursor()

# 使用 execute() 方法执行 SQL，如果表存在则删除
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# 使用预处理语句创建表
sql = """CREATE TABLE EMPLOYEE (
         FIRST_NAME  CHAR(20) NOT NULL,
         LAST_NAME  CHAR(20),
         AGE INT,
         SEX CHAR(1),
         INCOME DOUBLE )"""

cursor.execute(sql)

# 关闭数据库连接
db.close()
```

✓ 0.0s Python

- 执行以上代码，发现数据库表已经创建好：



数据库插入操作

- 以下实例使用执行 SQL INSERT 语句向表 EMPLOYEE 插入记录：

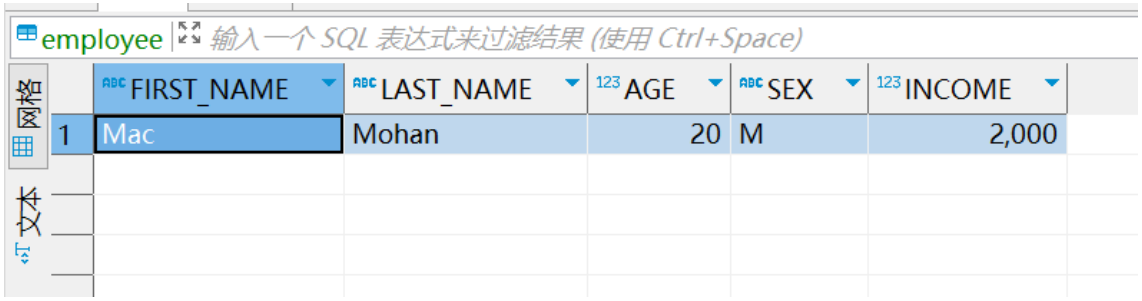
```python
import pymysql

# 打开数据库连接
db = pymysql.connect(host='localhost',
                     user='root',
                     password='password', # 填你自己的password
                     database='testdb',
                     port=3360 # 端口
                     )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 插入语句
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
         LAST_NAME, AGE, SEX, INCOME)
         VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""

# 以上语句也可以写成这样
"""
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \
       LAST_NAME, AGE, SEX, INCOME) \
       VALUES ('%s', '%s',  %s,  '%s',  %s)" % \
       ('Mac', 'Mohan', 20, 'M', 2000)
"""

try:
   # 执行sql语句
   cursor.execute(sql)
   # 提交到数据库执行
   db.commit()
except:
   # 如果发生错误则回滚
   db.rollback()

# 关闭数据库连接
db.close()
```

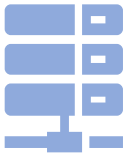
✓ 0.0s Python

数据库插入操作

- 以下实例使用执行 SQL INSERT 语句向表 EMPLOYEE 插入记录：

- 执行以上代码，发现数据已经插入到表中：





数据库查询操作

■Python查询Mysql使用 fetchone() 方法获取单条数据, 使用fetchall() 方法获取多条数据。

- **fetchone()**: 该方法获取下一个查询结果集。结果集是一个对象；
- **fetchall()**: 接收全部的返回结果行；
- **rowcount**: 这是一个只读属性，并返回执行execute()方法后影响的行数。

数据库查询操作

■示例：查询EMPLOYEE表中salary（工资）字段大于1000的所有数据

```python
import pymysql

# 打开数据库连接
db = pymysql.connect(host='localhost',
                     user='root',
                     password='password', # 填你自己的password
                     database='testdb',
                     port=3360 # 端口
                     )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 查询语句
sql = "SELECT * FROM EMPLOYEE \
       WHERE INCOME > %s" % (1000)
try:
   # 执行SQL语句
   cursor.execute(sql)
   # 获取所有记录列表
   results = cursor.fetchall()
   for row in results:
      fname = row[0]
      lname = row[1]
      age = row[2]
      sex = row[3]
      income = row[4]
       # 打印结果
      print ("fname=%s,lname=%s,age=%s,sex=%s,income=%s" % \
             (fname, lname, age, sex, income ))
except:
   print ("Error: unable to fetch data")

# 关闭数据库连接
db.close()
```

Python

```
fname=Mac,lname=Mohan,age=20,sex=M,income=2000.0
```

数据库更新操作

- 更新操作用于更新数据表的数据，以下实例将 TESTDB 表中 SEX 为 'M' 的 AGE 字段递增 1：

```python
import pymysql

# 打开数据库连接
db = pymysql.connect(host='localhost',
                     user='root',
                     password='password', # 填你自己的password
                     database='testdb',
                     port=3360 # 端口
                     )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 更新语句
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = '%c'" % ('M')
try:
   # 执行SQL语句
   cursor.execute(sql)
   # 提交到数据库执行
   db.commit()
except:
   # 发生错误时回滚
   db.rollback()

# 关闭数据库连接
db.close()
```

Python

	FIRST_NAME	LAST_NAME	AGE	SEX	INCOME
1	Mac	Mohan	20	M	2,000

	FIRST_NAME	LAST_NAME	AGE	SEX	INCOME
1	Mac	Mohan	21	M	2,000

数据库删除操作

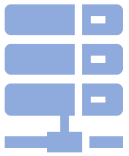
- 删除操作用于删除数据表中的数据，以下实例演示了删除数据表 EMPLOYEE 中 AGE 大于 20 的所有数据：

```python
import pymysql

# 打开数据库连接
db = pymysql.connect(host='localhost',
                     user='root',
                     password='password', # 填你自己的password
                     database='testdb',
                     port=3360 # 端口
                     )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 删除语句
sql = "DELETE FROM EMPLOYEE WHERE AGE > %s" % (20)
try:
   # 执行SQL语句
   cursor.execute(sql)
   # 提交修改
   db.commit()
except:
   # 发生错误时回滚
   db.rollback()

# 关闭连接
db.close()
```

✓ 0.0s

	FIRST_NAME	LAST_NAME	AGE	SEX	INCOME
1	Mac	Mohan	21	M	2,000
2	John	Doe	31	M	700
3	Jane	Smith	15	F	8,000
4	Alice	Johnson	35	F	6,000
5	Bob	Brown	41	M	10,000
6	Emma	Davis	18	F	5,500
7	Michael	Wilson	46	M	9,000
8	Sophia	Martinez	20	F	7,500
9	William	Anderson	39	M	8,500
10	Olivia	Taylor	13	F	6,200
11	James	Thomas	28	M	5,800
12	Ella	Jackson	29	F	6,700

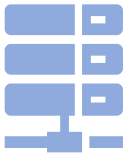
	FIRST_NAME	LAST_NAME	AGE	SEX	INCOME
1	Jane	Smith	15	F	8,000
2	Emma	Davis	18	F	5,500
3	Sophia	Martinez	20	F	7,500
4	Olivia	Taylor	13	F	6,200



执行事务

■事务机制可以确保数据一致性。事务应该具有4个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为ACID特性。

- **原子性（atomicity）**。一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。
- **一致性（consistency）**。事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。
- **隔离性（isolation）**。一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
- **持久性（durability）**。持续性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响
- Python DB API 2.0 的事务提供了两个方法 commit 或 rollback。

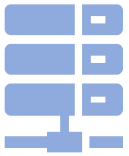


执行事务

- 对于支持事务的数据库， 在Python数据库编程中，当游标建立之时，就自动开始了一个隐形的数据库事务。
- commit()方法游标的所有更新操作，rollback（）方法回滚当前游标的所有操作。每一个方法都开始了一个新的事务。

```python
# SQL删除记录语句
sql = "DELETE FROM EMPLOYEE WHERE AGE > %s" % (20)
try:
   # 执行SQL语句
   cursor.execute(sql)
   # 向数据库提交
   db.commit()
except:
   # 发生错误时回滚
   db.rollback()
```

Python



错误处理

- DB API中定义了一些数据库操作的错误及异常，下表列出了这些错误和异常:

异常	描述
Warning	当有严重警告时触发，例如插入数据是被截断等等。必须是 StandardError 的子类。
Error	警告以外所有其他错误类。必须是 StandardError 的子类。
InterfaceError	当有数据库接口模块本身的错误（而不是数据库的错误）发生时触发。 必须是Error的子类。
DatabaseError	和数据库有关的错误发生时触发。 必须是Error的子类。
DataError	当有数据处理时的错误发生时触发，例如：除零错误，数据超范围等等。 必须是DatabaseError的子类。
OperationalError	指非用户控制的，而是操作数据库时发生的错误。例如：连接意外断开、 数据库名未找到、事务处理失败、内存分配错误等等操作数据库是发生的错误。 必须是DatabaseError的子类。
IntegrityError	完整性相关的错误，例如外键检查失败等。必须是DatabaseError子类。
InternalError	数据库的内部错误，例如游标（cursor）失效了、事务同步失败等等。 必须是DatabaseError子类。
ProgrammingError	程序错误，例如数据表（table）没找到或已存在、SQL语句语法错误、 参数数量错误等等。必须是DatabaseError的子类。
NotSupportedError	不支持错误，指使用了数据库不支持的函数或API等。例如在连接对象上 使用.rollback()函数，然而数据库并不支持事务或者事务已关闭。 必须是DatabaseError的子类。

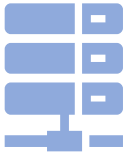
错误处理

```
CREATE PROCEDURE delete_table_proc ( IN table_name VARCHAR(100) )
BEGIN
  DECLARE error_msg VARCHAR(255) DEFAULT NULL;
START TRANSACTION; --开始事务
BEGIN SET @sql_statement = CONCAT('DROP TABLE ', table_name); -- 尝试删除表
    PREPARE stmt FROM @sql_statement;
    EXECUTE stmt;
    DEALLOCATE PREPARE stmt;
 EXCEPTION
    WHEN others THEN -- 获取错误信息
     SET error_msg = CONCAT('Error deleting table ‘, table_name, ': ', GET_DIAGNOSTICS CONDITION 1);
      ROLLBACK; -- 回滚事务
     END;
IF error_msg IS NULL THEN -- 提交事务 COMMIT;
    SELECT 'Table deleted successfully.' AS message;
ELSE -- 返回错误信息
  SELECT error_msg AS message;
 END IF;
END$$
```

1.SQLEXCEPTION：捕获所有SQL异常，相当于通配符。
2.SQLWARNING：捕获SQL警告。
3.NOT FOUND：捕获未找到数据的异常，通常在游标操作中使用。
4.DUP_VAL_ON_INDEX：捕获在唯一索引上插入重复值的异常。
5.PROGRAM_ERROR：捕获编程错误，例如除零异常等。
6.STORAGE_ERROR：捕获存储异常，例如内存不足等。
7.DATA_ERROR：捕获数据异常，例如数据类型不匹配等

数据库应用系统开发

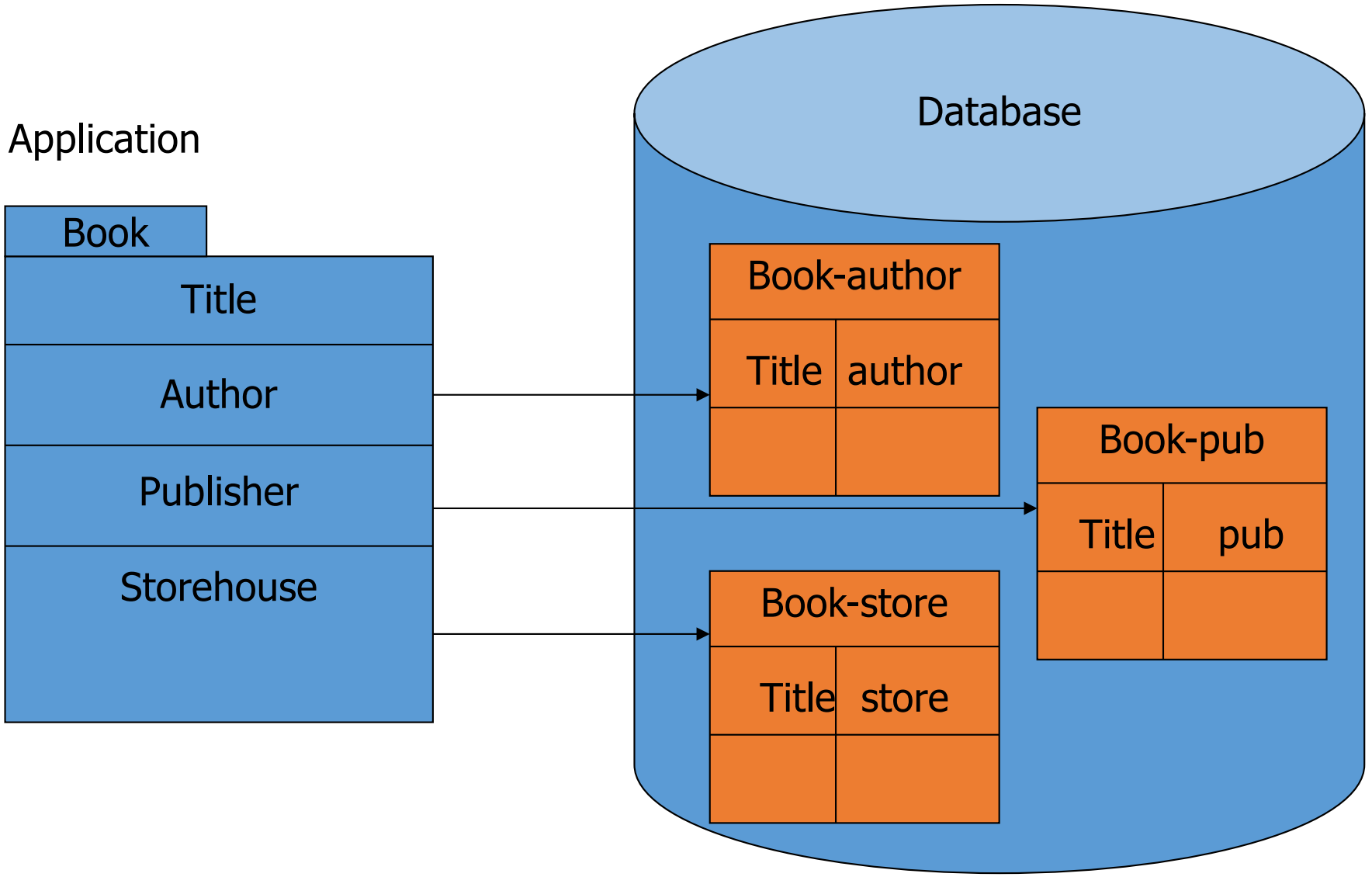
- 数据库访问与Python MySQL Client
- 对象关系映射--ORM

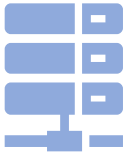


ORM—对象-关系映射

- 大多数系统使用关系数据库进行数据管理，并使用面向对象的语言开发上层应用
- 进行数据存取的时候必须在对象与二维表间建立映射关系。也就是说对象数据只能扁平化为关系表后才能存储
- 关系模型建模能力有限，面向对象方法无法贯彻到关系数据库中
- 应用程序必须嵌入SQL语言才能操纵数据库
- 以上不便被称为“Impedance mismatch”，即阻抗失配

ORM—对象-关系映射


Application
Book
Title
Author
Publisher
Storehouse
Database
Book-author
Title
author
Book-pub
Title
pub
Book-store
Title
store



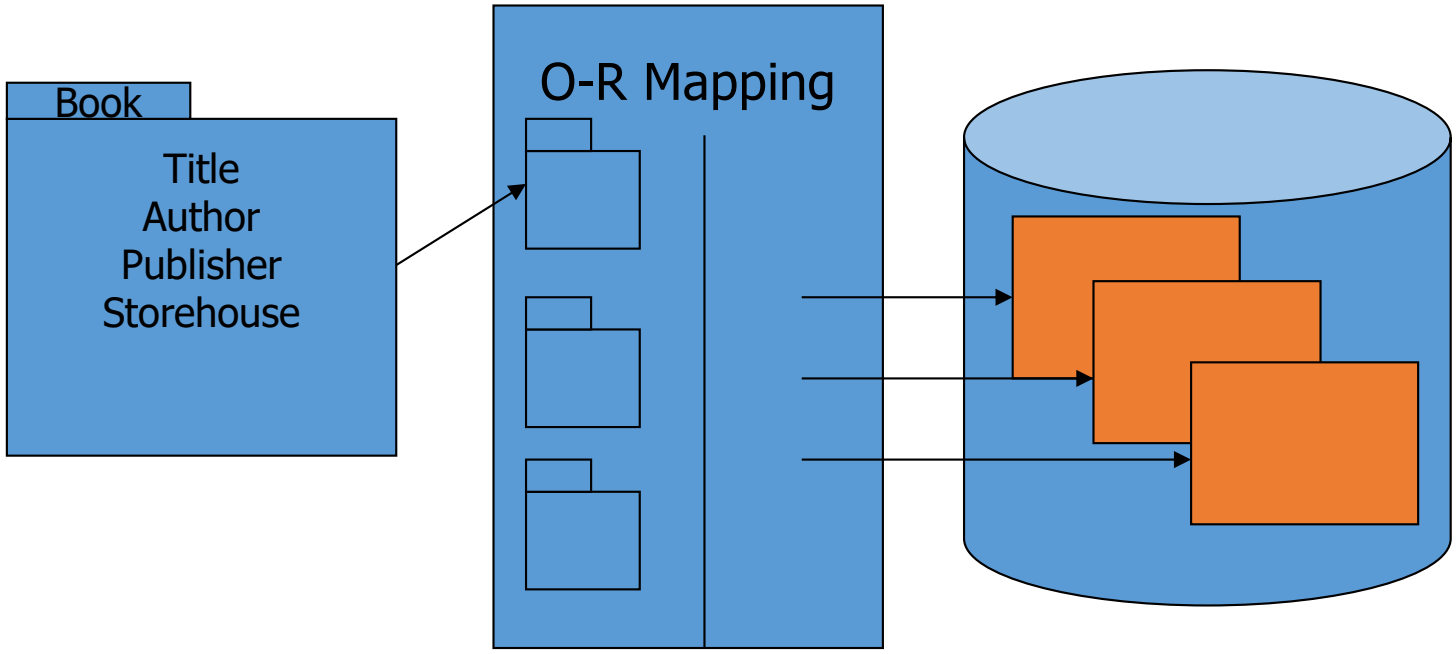
ORM—对象-关系映射

- 面向对象的设计方法的目标是对处理过程建模
- 关系数据库设计的目标是使数据符合范式要求
- 将对象映射到表会遇到很大困难，特别是：
  - ✓对象含有复杂结构
  - ✓存在大的非结构化的对象
  - ✓存在类继承
- 映射的结果很可能是：表存取的效率很差；或在表中检索对象很困难
- 一种有效的解决方案： Object-Relation Mapping

ORM—对象-关系映射

O-R映射

- ✓用户开发和维护一个中间件层，负责将对象数据映射到关系数据库的表中
- ✓系统中其它模块可以通过OR映射层以操作对象的方法操作关系表中的数据
- ✓OR映射对前端开发人员屏蔽了数据库底层细节，使得他们可以专注于业务流程的实现，极大提高了应用系统开发的生产率



ORM

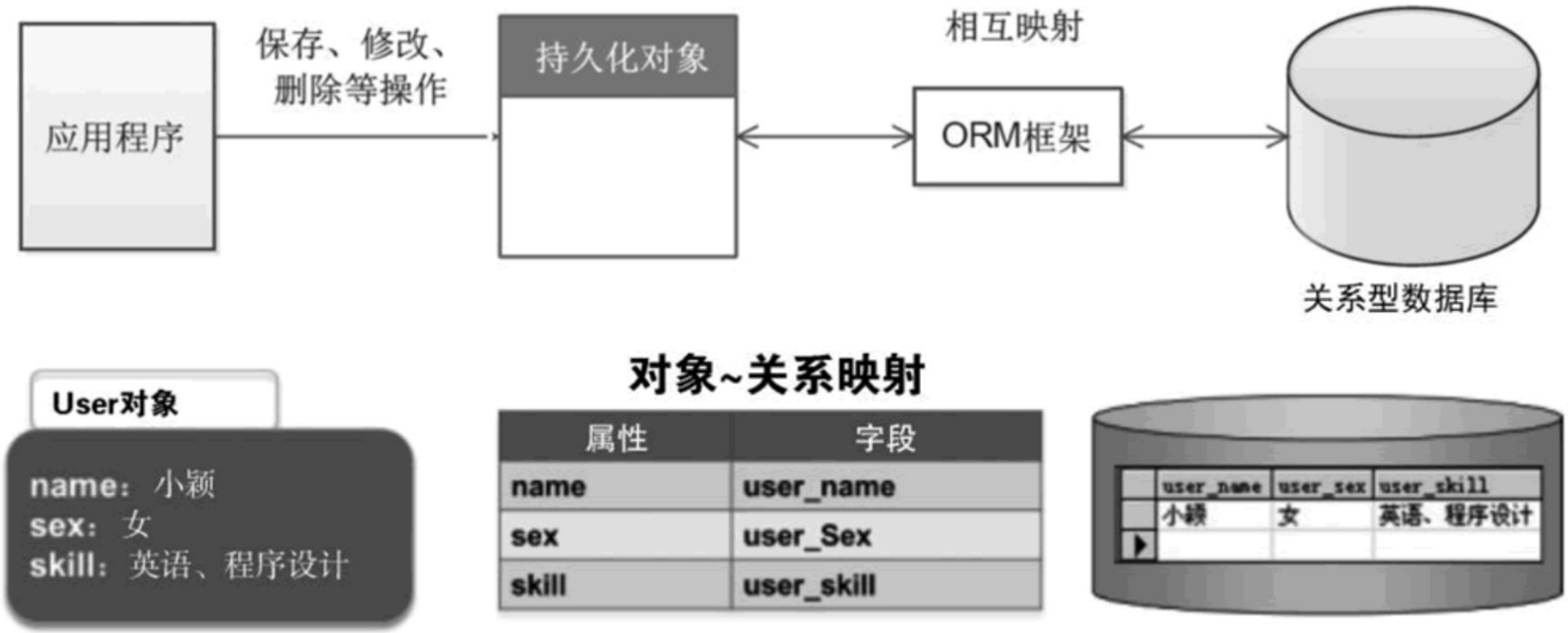


图1：ORM 映射关系

ORM 实现方案

- Active Record
- Data Mapper

List of object–relational mapping software - Wikipedia

Active Record

- Active record：“ *一个对象既包含数据又包含行为。这些数据大部分是持久性的，需要存储在数据库中。Active Record使用最明显的方法，将数据访问逻辑放在域对象中。这样，所有人都知道如何在数据库中读取和写入数据。*”

- 将对象和数据库表看作是一一对应的关系，一个对象对应一个数据库表中的一行数据，对象和行数据之间的映射关系由ORM框架自动维护。通过对象的属性来操作数据表，例如修改对象的属性后，直接通过ORM框架更新到数据库表中。

- 优点：简单，容易理解
- 缺点：耦合度高，性能较差

Active Record

- 当涉及到使用ORM的Active Record模式时，在许多编程语言（例如Ruby on Rails和PHP）中，常见的模式是使用许多内置的ORM类和方法。以下是一个使用 Ruby on Rails 框架中的 Active Record 模式的示例：

```ruby
class Product < ActiveRecord::Base
  validates :name, presence: true
  validates :price, numericality: {
greater_than_or_equal_to: 0 }

  def on_sale?
    price < 10
  end
end
```

- 在上面的示例中，我们定义了一个 "Product" 类，该类使用 "ActiveRecord::Base" 派生自 ActiveRecord 模型的基类。我们还定义了两个验证，以确保在创建和更新产品时，“name” 字段不为空，而 “price” 字段为数字且大于或等于零。除此之外，我们还定义了一个简单的辅助方法 “on_sale?”，该方法检查产品价格是否低于 10 美元。

Active Record

- 当我们将此类链接到与数据库中的 “products” 表的记录时，Active Record 将处理将数据映射到对象属性，并提供了 CRUD（创建，读取，更新和删除）操作，使我们能够轻松与底层数据库进行交互。例如，我们可以使用以下方式创建新产品：

```
new_product = Product.new(name: 'New product', price: 5)
new_product.save
```

- 在上面的例子中，我们创建了一个新的 "Product" 对象，并将其 "name" 属性设置为 "New product"， "price" 属性设置为 5。然后，我们通过 “save” 方法存储新的产品记录。这将在数据库中插入新的 “products” 表记录，以及更新相应的 Product 对象。

Data Mapper

- Data Mapper：将对象和数据库表看作是两个独立的概念，对象类与数据库表之间没有必然的联系，需要手动定义对象属性和数据表的字段之间的映射关系。对象的读写操作不直接与数据库交互，而是通过数据访问对象（Data Access Object，DAO）来实现。

- 优点：
  - ✓提供更好的灵活性，选择需要的对象属性进行存储
  - ✓使得应用层和数据层相对独立，隐藏了彼此的细节
  - ✓性能较好
- 缺点：
  - ✓复杂，需要一定的学习成本，部署较为困难

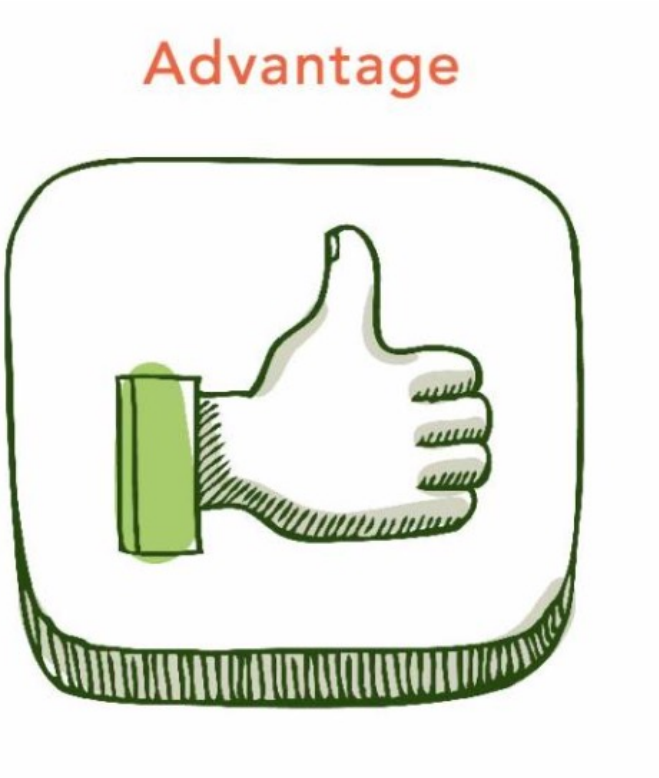
Data Mapper

- 在右边的例子中，定义了一个“User” 类，它是DataMapper::Resource 的子类。使用了 DataMappper 的 DSL（领域特定语言）来定义对象和数据表之间的映射关系
  - ✓例如，使用 “property” 方法定义了 “id” 和 “name” 属性，并指定它们所需的数据类型。
  - ✓此外还定义了几个方法，以便处理 User 记录的 CRUD 操作。
    - find_by_name：该方法通过 DataMapper 的 “first” 方法从数据库中检索符合条件的第一条记录。
    - save和destroy方法：它们分别将对象的更改保存到数据库中或删除其对应的记录。

```ruby
class User
  include DataMapper::Resource

  property :id, Serial
  property :name, String

  def self.find_by_name(name)
    first(name: name)
  end

  def save
    if valid?
      DataMapper::Model.save(self)
    else
      false
    end
  end

  def destroy
    DataMapper::Model.destroy(self)
  end
end
```

ORM 优点

- 提高开发效率
- 数据库平台透明
- 数据库结构自动维护
- 代码可读性高



ORM 缺点

- 需要一定的学习成本
- 性能问题
- 不适用于复杂场景
- 还有其他吗？自行思考！

关于ORM的思考

- ORM框架在复杂场景下，比如查询包含子查询，表现的并不是很好，甚至无法满足需求

- 对于真实的业务场景，数据变化快，如何提前确定一个合适的ORM呢?

- 对于我们的作业，ORM是否合适呢?

是否需要ORM?

- 你是否真的需要ORM?
- ORM本质上是为了将程序员从SQL语句中解放出来，但是代价是什么呢?
- 选择自己的需要!

延伸阅读

Web前后端开发

✓**W3Schools**：世界最大的Web开发人员网站

链接：https://www.w3schools.com

✓**Programming with Mosh**：YouTube速成教学视频

链接:

https://www.youtube.com/c/programmingwithmosh

大作业

- ✓ 要求同学们基于提供的MySQL数据库链接进行代码开发。
- ✓ 使用ORM框架搭建一个简易的“前”后端系统，提供数据插入、更新、删除、查询等4个接口，并将其部署为RESTful服务。