



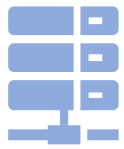
北京航空航天大学
BEIHANG UNIVERSITY

数据管理技术

北京航空航天大学

周号益

2024年



数据库编程-1

数据库程序开发

□ SQL语言主要用于数据查询和更新

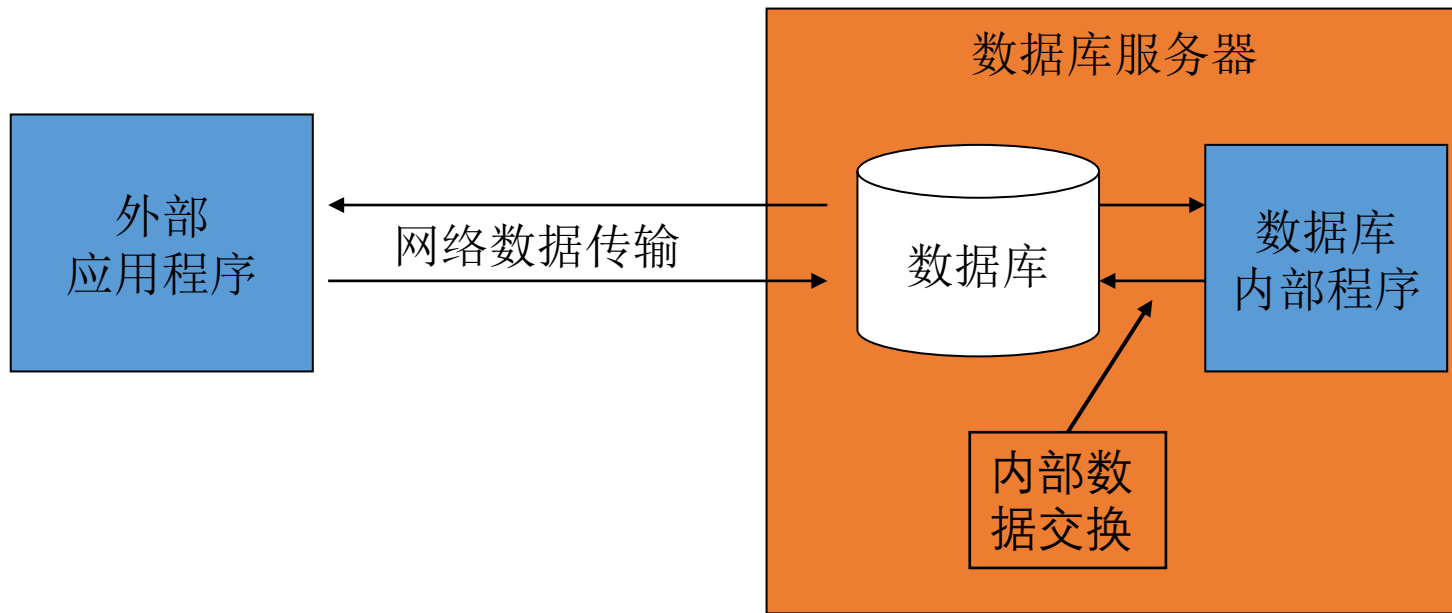
- ✓ 由于缺乏变量定义、流程控制等高级编程语言特性，难以担负复杂的数据分析处理任务
- ✓ 代码重用性差，运行效率低

□ 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
- ✓可以作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...

✓ 嵌入式SQL

✓ 动态SQL

✓ 游标编程

自学

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。

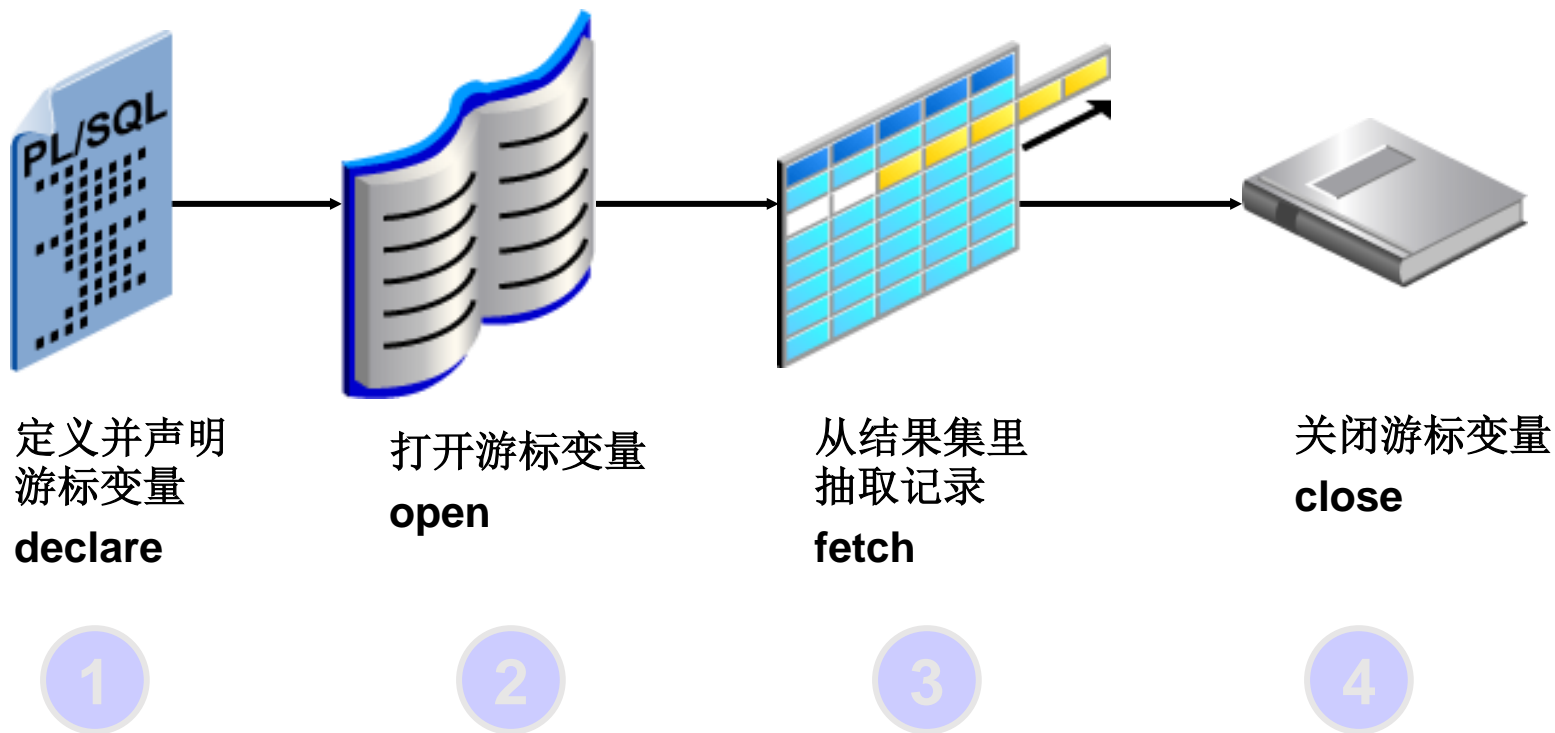
主要功能包括：

- 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

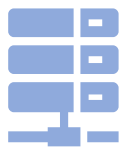


游标编程示例

```
CREATE PROCEDURE my_cursor()  
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<18 THEN  
                SET p_c=p_c+1;  
            END IF;  
        END WHILE;  
        Select p_c as '小于18岁的总人数'; -- 输出结果  
    CLOSE my_cursor; -- 关闭游标，释放内存
```

```
END
```



数据库编程-2

数据库系统开发

- 数据库访问与Python MySQL Client

- ORM

- 延伸阅读：开发模式

- Docker usage

- 作业介绍

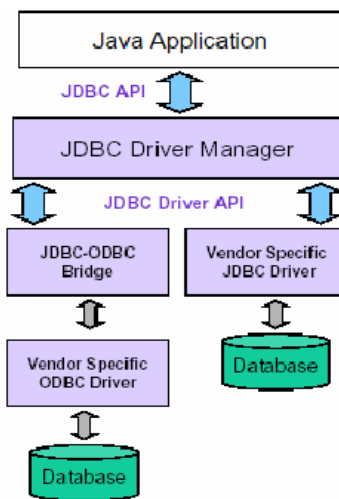


数据库访问

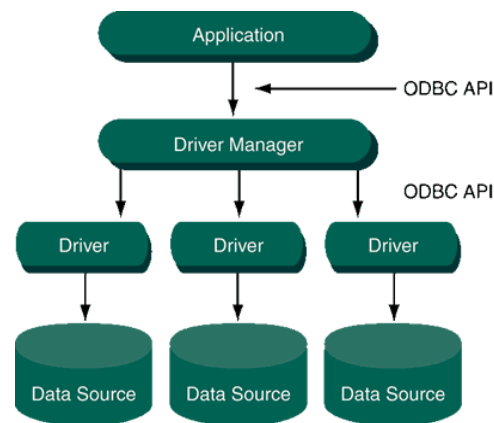
- 各种高级编程语言都提供访问和操作数据库的类库
- 使用类库进行数据库开发的基本过程：

JDBC:

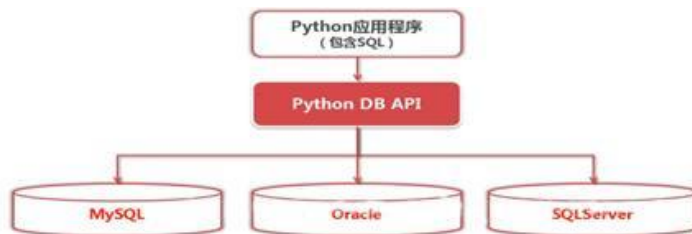
1. Establish a **Connection**
2. Create SQL **Statements**
3. Execute **SQL** Statements
4. GET **ResultSet**
5. **Close** connections



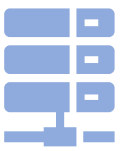
JDBC



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( columnName ) ) 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 mysqlldb。
- 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

```
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()

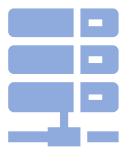
print ("Database version : %s " % data)

# 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:

```
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")

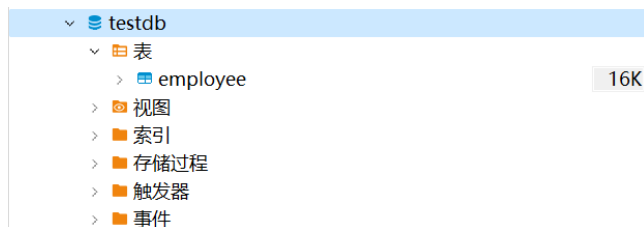
# 使用预处理语句创建表
sql = """CREATE TABLE EMPLOYEE (
        FIRST_NAME  CHAR(20) NOT NULL,
        LAST_NAME   CHAR(20),
        AGE         INT,
        SEX         CHAR(1),
        INCOME      DOUBLE )"""

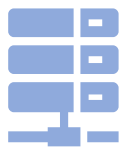
cursor.execute(sql)

# 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:

```
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:

employee 输入一个 SQL 表达式来过滤结果 (使用 Ctrl+Space)					
网格	ABC FIRST_NAME	ABC LAST_NAME	123 AGE	ABC SEX	123 INCOME
1	Mac	Mohan	20	M	2,000
文本					



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

```
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
                    )

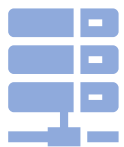
# 使用cursor()方法获取操作游标
cursor = db.cursor()

# 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：

```
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()

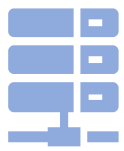
# 关闭数据库连接
db.close()
```

Python

	ABC FIRST_NAME	ABC LAST_NAME	123 AGE	ABC SEX	123 INCOME
1	Mac	Mohan	20	M	2,000

↓

	ABC FIRST_NAME	ABC LAST_NAME	123 AGE	ABC SEX	123 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：

```
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。

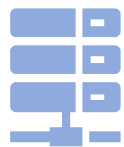
- **原子性 (atomicity)** 。一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。
- **一致性 (consistency)** 。事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。
- **隔离性 (isolation)** 。一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
- **持久性 (durability)** 。持续性也称永久性 (permanence) ， 指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其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

```
# 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()
```



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。

数据库系统开发

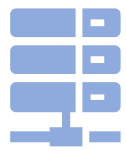
- 数据库访问与Python MySQL Client

- ORM

- 延伸阅读：开发模式

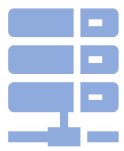
- Docker usage

- 作业介绍

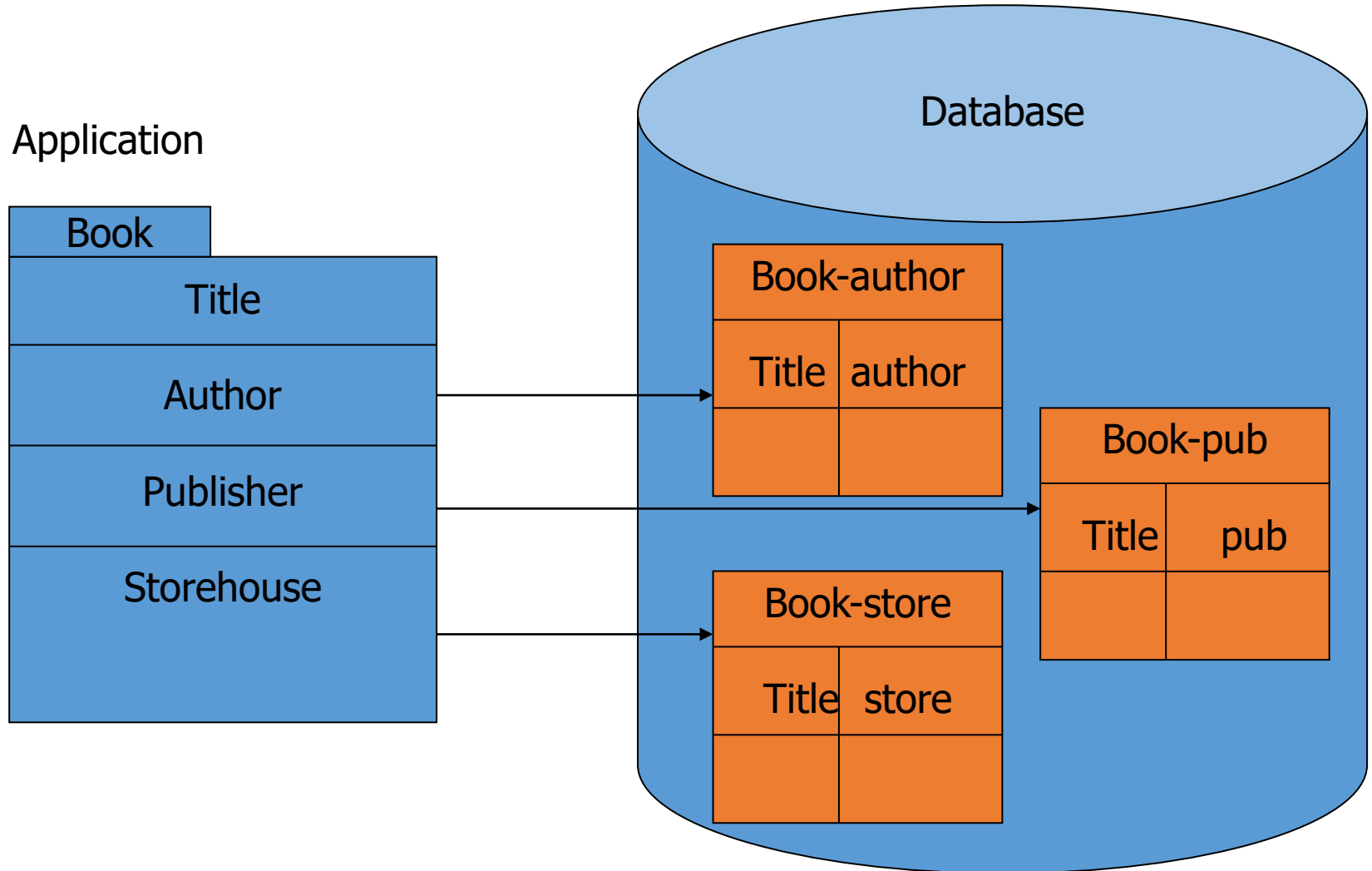


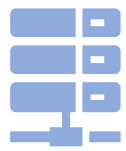
ORM—对象-关系映射

- ❑ 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





ORM—对象-关系映射

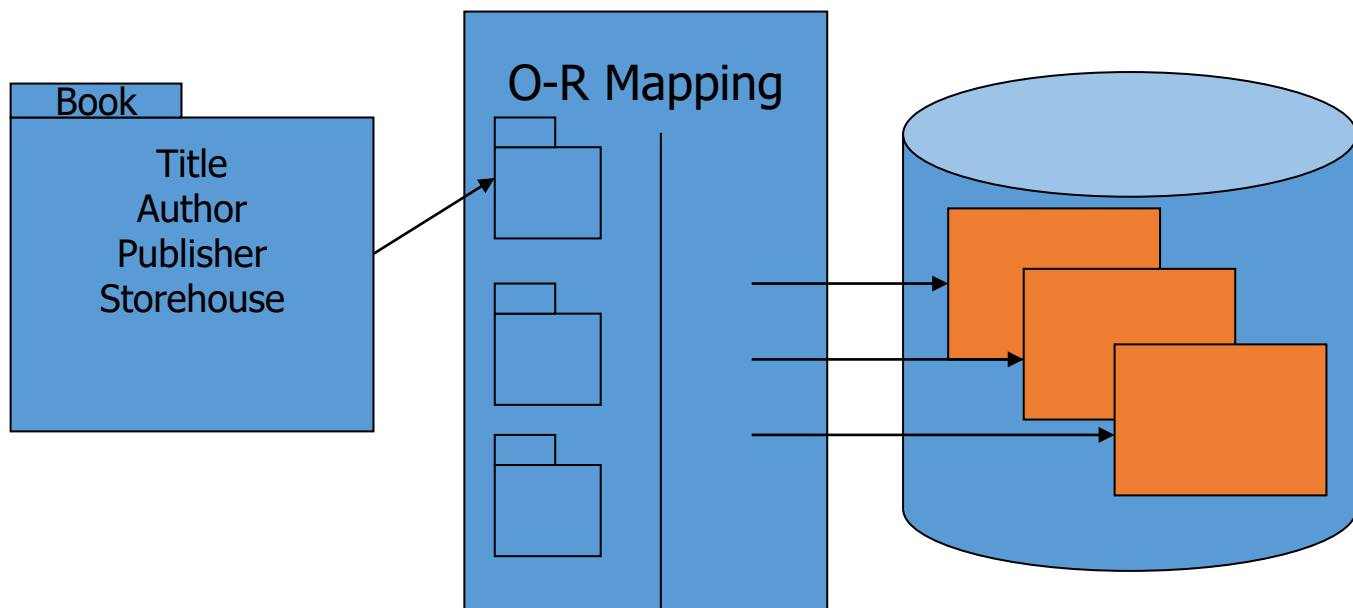
- 面向对象的设计方法的目标是对处理过程建模
- 关系数据库设计的目标是使数据符合范式要求
- 将对象映射到表会遇到很大困难，特别是：
 - ✓ 对象含有复杂结构
 - ✓ 存在大的非结构化的对象
 - ✓ 存在类继承
- 映射的结果很可能是：表存取的效率很差；或在表中检索对象很困难
- 一种有效的解决方案： Object-Relation Mapping



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

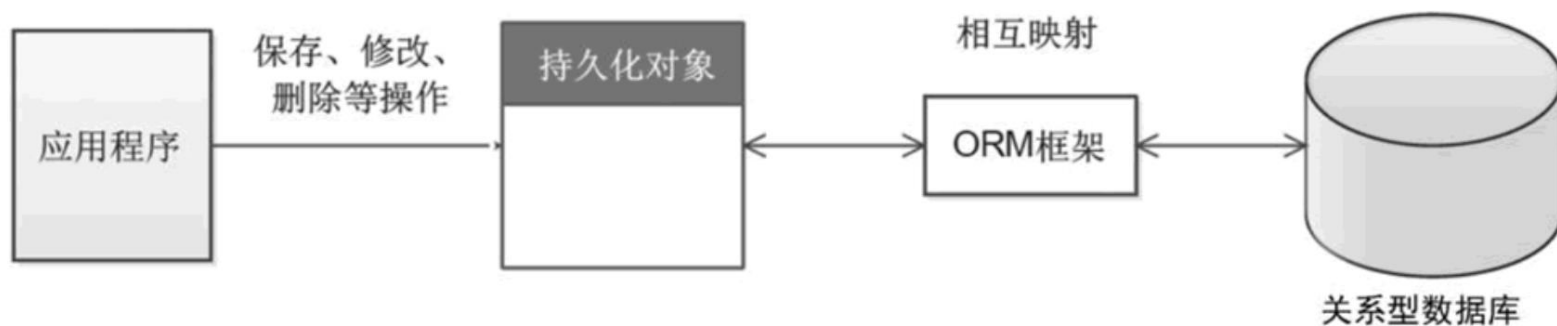


图1: ORM 映射关系

ORM 实现方案

□ Active Record

□ Data Mapper

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——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：

```
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 `DataManager::Resource` 的子类。使用了 `DataMapper` 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。

```
class User
  include DataManager::Resource

  property :id, Serial
  property :name, String

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

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

  def destroy
    DataManager::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

Advantage



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！

Disadvantage



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!



数据库系统开发

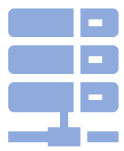
□数据库访问与Python MySQL Client

□ORM

□延伸阅读：开发模式

□Docker usage

□作业介绍



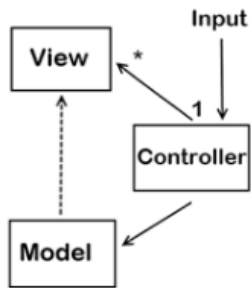
MVC/MVP/MVVM

■什么是MVC/MVP/MVVM

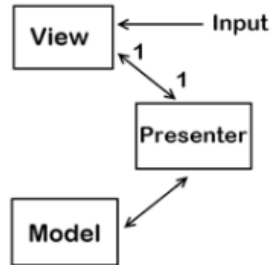
- MVC、MVP、MVVM 是开发中三个最常用的应用架构模式，目的都是为了将业务和视图的实现代码分离，从而使同一个程序可以使用不同的表现形式。
- MVC = **Model-View-Controller**, MVP = **Model-View-Presenter**, MVVM = **Model-View-ViewModel**。这三个架构模式，都分别有三个不同的部件，都有相同的 Model 层和 View 层。
- 三个架构模式目的都是为了**解耦 Model 和 View**，主要不同点就在于三者实现解耦的方案不同。Controller、Presenter、ViewModel，对应着三种不同的解耦方案，三种与 M 和 V 的连接方式。

MVC/MVP/MVVM

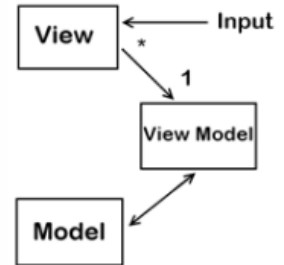
MVC VS MVP VS MVVM



MVC



MVP



MVVM



MVC

■MVC模式

- **视图 (View)** 表示用户界面，负责向用户展示数据和接受用户输入。视图能够读取模型中的数据，但不能够修改数据。视图通常只需要获取数据并以更美观的方式展示给用户即可。
- **控制器 (Controller)** 负责协调视图和模型之间的交互，处理用户输入并相应地更新视图和模型。控制器将用户输入和视图操作转化为适当的模型动作。控制器通过修改模型中的数据来改变应用程序的状态，并刷新视图。
- **模型 (Model)** 表示应用程序的业务数据和业务规则，负责读取和存储数据，控制业务逻辑。模型并不依赖于视图或控制器，所以模型能够独立于视图和控制器进行开发和测试。

■MVC通信方式

- View 传送指令到 Controller
- Controller 完成业务逻辑后，要求 Model 改变状态
- Model 将新的数据发送到 View，用户得到反馈

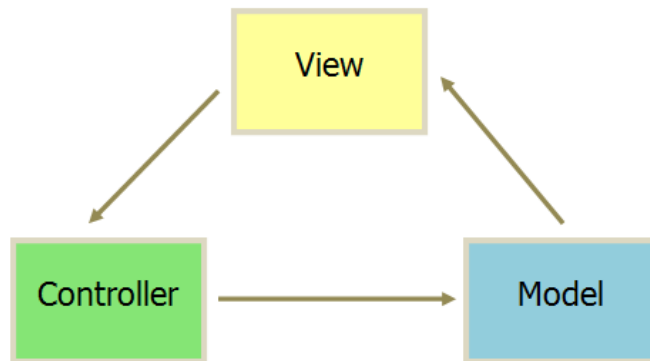


图1：MVC模式



MVC示例

■以 Backbone.js 为例

- 用户可以向 View 发送指令（DOM 事件），再由 View 直接要求 Model 改变状态。
- 用户也可以直接向 Controller 发送指令（改变 URL 触发 hashChange 事件），再由 Controller 发送给 View。
- Controller 非常薄，只起到路由的作用，而 View 非常厚，业务逻辑都部署在 View。所以，Backbone 索性取消了 Controller，只保留一个 Router（路由器）。

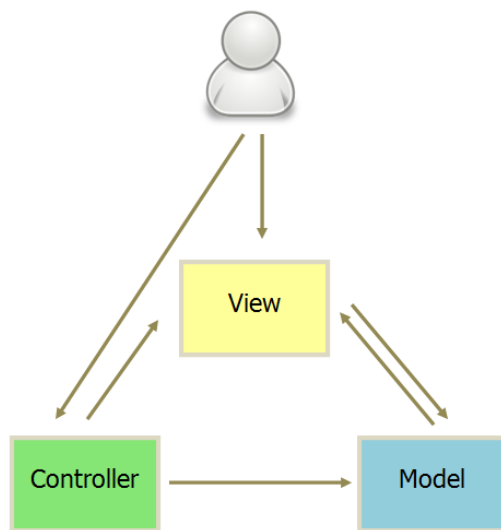


图1：Backbone.js 示例



MVC设计模式

■组合模式

- 组合模式用来实现 View 层。
- View 被设计为可嵌套的。比如，列表视图(ListView)或表格视图(Table View)由每个 Item 组成，每个 Item 又可以由图片、文本、按钮等组成。View 是倾向于可复用的，因此，在实际应用中，倾向于将 View 开发成相对通用的组件。



MVC设计模式

■策略模式

- 策略模式用于 View 与 Controller 之间。
- Controller 对象为一个或多个 View 对象实现了策略，View 对象仅限于保持其视觉外观，而与程序逻辑相关的所有决策都委托给 Controller，即 View 可以使用不同的 Controller 实现，得到不同的行为。



MVC设计模式

■ 观察者模式

- 观察者模式用于 Model 与 View 之间。
- View 会注册为 Model 的观察者，当 Model 有变化的时候，就能通知到 View。



MVC优点和缺点

■优点

- **耦合度低。** 视图层和业务层分离，这样就允许更改视图层代码而不用重新编译模型和控制器代码，同样，一个应用的业务流程或者业务规则的改变只需要改动MVC的模型层即可。因为模型与控制器和视图相分离，所以很容易改变应用程序的数据层和业务规则。
- **重用性高。** MVC模式允许使用各种不同样式的视图来访问同一个服务器端的代码，因为多个视图能共享一个模型，它包括任何WEB（HTTP）浏览器或者无线浏览器（wap）
- **生命周期成本低。** MVC使开发和维护用户接口的技术含量降低。
- **部署快。** 使用MVC模式使开发时间得到相当大的缩减，它使程序员（Java开发人员）集中精力于业务逻辑，界面程序员（HTML和JSP开发人员）集中精力于表现形式上。
- **可维护性高。** 分离视图层和业务逻辑层也使得WEB应用更易于维护和修改。
- **有利软件工程化管理。**



MVC优点和缺点

■缺点

- 没有明确的定义。完全理解MVC并不是很容易。使用MVC需要精心的计划，由于它的内部原理比较复杂，所以需要花费一些时间去思考。同时由于模型和视图要严格的分离，这样也给调试应用程序带来了一定的困难。每个构件在使用之前都需要经过彻底的测试。
- 不适合小型，中等规模的应用程序。花费大量时间将MVC应用到规模并不是很大的应用程序通常会得不偿失。
- 增加系统结构和实现的复杂性。MVC使开发和维护用户接口的技术含量降低。
- 视图与控制器间的过于紧密的连接。对于简单的界面，严格遵循MVC，使模型、视图与控制器分离，会增加结构的复杂性，并可能产生过多的更新操作，降低运行效率。
- 视图对模型数据的低效率访问。依据模型操作接口的不同，视图可能需要多次调用才能获得足够的显示数据。对未变化数据的不必要的频繁访问，也将损害操作性能。
- 一般高级的界面工具或构造器不支持模式。



MVP

■ MVP (Model-View-Presenter) 是从经典的模式MVC演变而来，它们的基本思想有相通的地方Controller/Presenter负责逻辑的处理，Model提供数据，View负责显示。

■ 区别

- MVP各部分之间的通信，都是双向的。
- View 与 Model 不发生联系，都通过 Presenter 传递。
- View 非常薄，不部署任何业务逻辑，称为"被动视图" (Passive View)，即没有任何主动性，而 Presenter非常厚，所有逻辑都部署在那里。
- MVP 之间的交互主要是通过接口实现的，Model、View、Presenter 都有各自的接口，定义各自的行为方法。

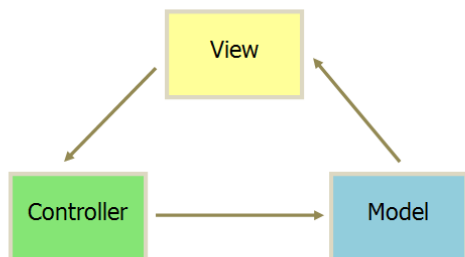


图1：MVC模式

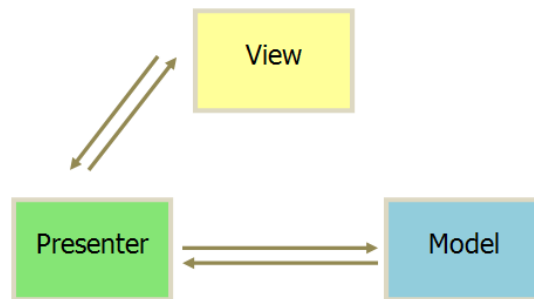
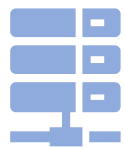


图2：MVP模式



MVP示例

■ 一个简单的登录案例：定义了 4 个接口和 3 个实现类，其中，**LoginActivity** 是 Android 的 Activity 类，在 iOS 中，则可定义为 **LoginViewController**。LoginActivity 实现了 LoginView 接口，同时还会持有一个 LoginPresenter 对象。LoginView 和 LoginActivity 都明确划分到 View 层，**LoginView** 定义了登录流程中涉及到的几个UI层的接口方法，包括显示和隐藏加载框，以及登录失败时的错误信息展示，和登录成功后的处理，这些方法都会在 **LoginPresenterImpl** 的方法中被调用。Presenter 层定义了两个接口，LoginPresenter 只有一个登录的接口方法，**OnLoginFinishedListener** 则定义了登录结果的回调方法，两个接口统一由 LoginPresenterImpl 来实现即可。另外，从图中也可看到，LoginPresenterImpl 既持有一个 LoginView 对象，也持有一个 LoginModel 对象，LoginPresenterImpl 其实就是 LoginView 和 LoginModel 之间交互的桥梁。而 Model 层的 LoginModel 则不直接持有 LoginPresenter 对象，只在登录接口中加了一个回调对象的参数。

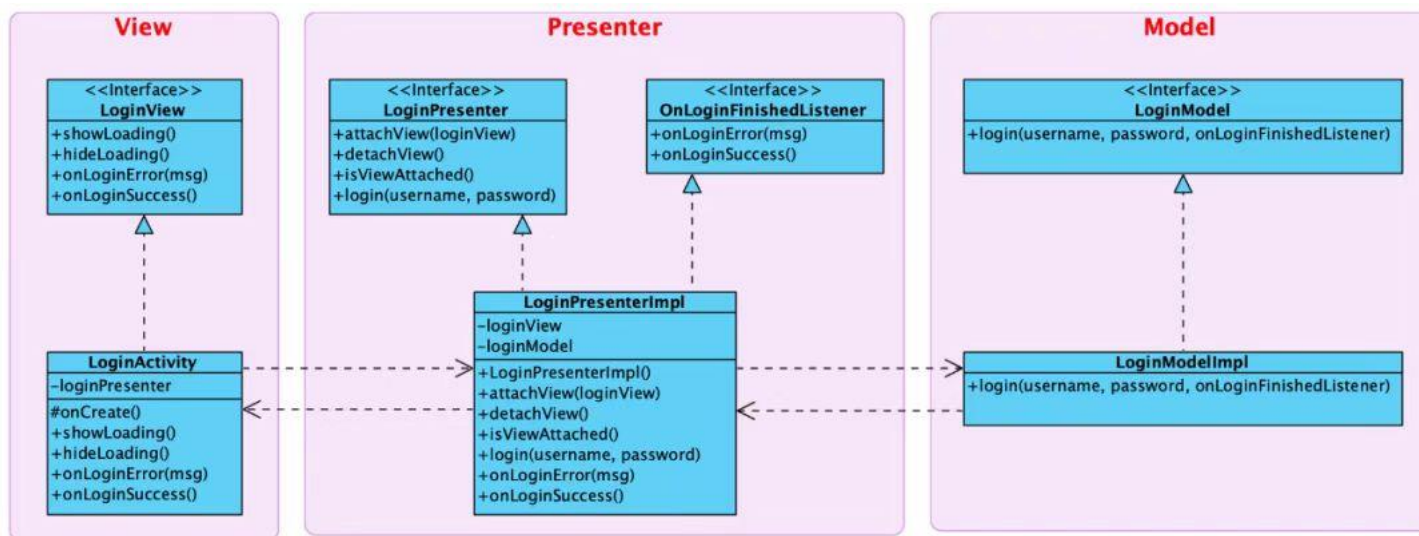


图1：登录案例类图



MVP优点和缺点

■优点

- 模型与视图完全分离，我们可以修改视图而不影响模型。
- 可以更高效地使用模型，因为所有的交互都发生在一个地方——Presenter内部。
- 我们可以将一个Presenter用于多个视图，而不需要改变Presenter的逻辑。这个特性非常的有用，因为视图的变化总是比模型的变化频繁。
- 如果把逻辑放在Presenter中，那么我们就可以脱离用户接口来测试这些逻辑（单元测试）



MVP优点和缺点

■缺点

- 由于对视图的渲染放在了Presenter中，所以视图和Presenter的交互会过于频繁。还有，如果Presenter过多地渲染了视图，往往会使得它与特定的视图的联系过于紧密。一旦视图需要变更，那么Presenter也需要变更了。
- 由于增加了很多接口的定义，需要编写的代码量暴增，增加了项目的复杂度。
- 需要对很多业务模块之间的交互抽象成接口定义，对开发人员的设计能力要求更高了。



MVVM

- MVVM = Model-View-ViewModel, 与 MVC、MVP 不同的就在于最后一个部件, 换成了 **ViewModel** (VM)
- MVVM 的关系图和 MVP 很相似, 最大的不同在于 View 和 ViewModel 之间主要是通过**数据绑定**的方案来实现交互的。
- View Model: Model 封装了**业务逻辑和数据**, 管理的是业务模型。而 ViewModel = Model of View, 即视图的模型, 封装的是**视图的表示逻辑和数据**, 是对视图的抽象, 包括视图的属性和命令, 或视图的状态和行为。

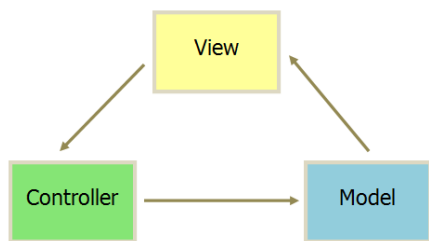


图1: MVC模式

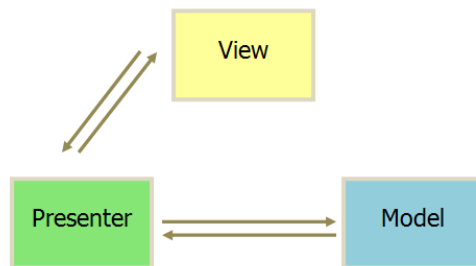


图2: MVP模式

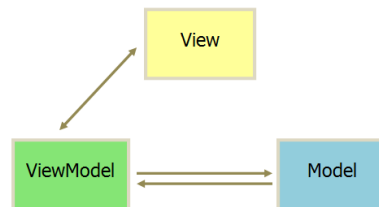
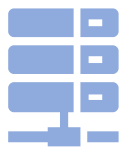


图3: MVVM模式



MVVM示例

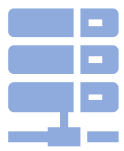
■ 比如我们要在页面中展示一个购物订单的信息，视图中要展示的内容包括订单号、订单状态、创建时间、成交时间、订单金额、商品名称、购买数量等。那么，在业务模型中，该订单的数据结构一般是这样的：

```
1 {  
2     "orderId": "107896581204552",  
3     "status": "paid",  
4     "createdTime": 1592826572000,  
5     "dealTime": 1592826691000,  
6     "amount": 105.28,  
7     "productName": "VicTsing MM057 2.4G 无线便携式光学鼠标，带 USB 接收器",  
8     "buyQty": 1  
9 }
```

■ 而要将这个订单的数据展示到界面，肯定还需要做一些数据转换，比如，需要将时间戳转换为「yyyy-MM-dd HH:mm:ss」格式的时间，paid 转换为「已付款」，金额前面再加个¥符号，数量前面加多个乘号。所以，对应到界面上的数据结构大概是这样的：

```
1 {  
2     "orderId": "107896581204552",  
3     "status": "已付款",  
4     "createdTime": "2020-06-22 19:49:32",  
5     "dealTime": "2020-06-22 19:51:31",  
6     "amount": "¥105.28",  
7     "productName": "VicTsing MM057 2.4G 无线便携式光学鼠标，带 USB 接收器",  
8     "buyQty": "X1"  
9 }
```

■ 这就是 ViewModel 中所指的视图数据，不过这里演示的只是视图的属性，也是视图状态。但 ViewModel 封装的除了属性，也包括命令，即视图行为，比如页面刚加载进来时发生什么，点击某个按钮发生什么，点击列表中的某个 item 又发生什么，这些都属于视图行为。



■ **数据绑定：** 通过将 View 的属性绑定到 ViewModel，可以使两者之间松耦合，也完全不需要在 ViewModel 里写代码去直接更新一个 View。数据绑定系统还支持输入验证，这提供了将验证错误传输到 View 的标准化方法。

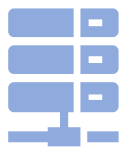
- **单向绑定：** ViewModel 与 View 绑定之后，ViewModel 变化后，View 会自动更新，但反之不然，即数据传递的方向是单向的。（ViewModel → View）
- **双向绑定：** ViewModel 与 View 绑定之后，如果 View 和 ViewModel 中的任何一方变化后，另一方都会自动更新，这就是双向绑定。（Model ↔ View）
- 一般情况下，在视图中只显示而无需编辑的数据用单向绑定，需要编辑的数据才用双向绑定。



MVVM优点

■优点

- 低耦合。视图 (View) 可以独立于Model变化和修改, 一个ViewModel可以绑定到不同的"View"上, 当View变化的时候Model可以不变, 当Model变化的时候View也可以不变。
- 可重用性。可以把一些视图逻辑放在一个ViewModel里面, 让很多view重用这段视图逻辑。
- 独立开发。开发人员可以专注于业务逻辑和数据的开发 (ViewModel), 设计人员可以专注于页面设计, 使用Expression Blend可以很容易设计界面并生成xaml代码。
- 可测试。界面素来是比较难于测试的, 而现在测试可以针对ViewModel来写。



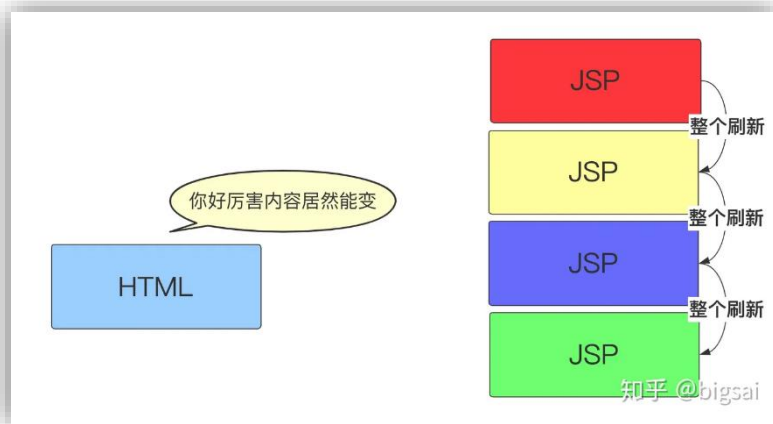
MVVM缺点

■缺点

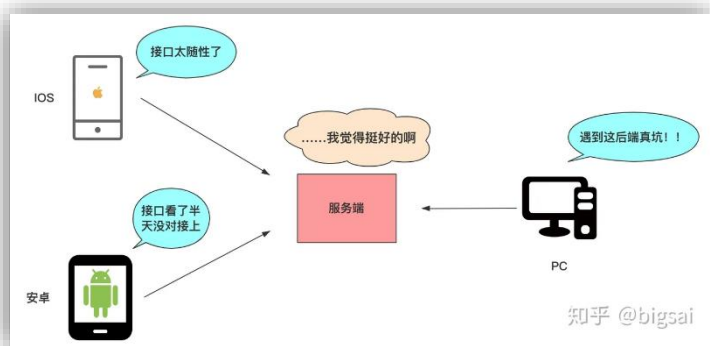
- **学习曲线较陡峭：** MVVM需要更多的编写代码和复杂的数据绑定配置，并且需要学习各种新的框架、库和技术，对开发者的学习和实践门槛较高。
- **复杂性增加：** MVVM通过精细的数据绑定机制，实现了Model, ViewModel, View三者之间的松耦合，但数据的双向绑定也增加了代码的复杂性。
- **内存开销增大：** MVVM模式下，因为每个页面都需要相应的ViewModel对象，所以随着页面数目的增加，内存开销也会相应增大，特别是在使用大量数据的情况下。
- **调试难度增加：** 因为MVVM模式下代码的组织方式相对复杂，这增加了调试时发现问题的难度，对于初学者来说可能不易排除问题。
- **可读性下降：** 使用其它框架，尤其是在数据绑定方面的框架具有语法糖和扩展特性，这些特性可能会增加代码的抽象程度和精髓度，但有些时候会降低代码的可读性和可维护性，尤其是在需要跨过多个相关代码段时。

RESTful API

❑在互联网并没有完全流行的初期，移动端也没有那么盛行，页面请求和并发量也不高，那时候人们对接口的要求没那么高，一些动态页面(jsp)就能满足绝大多数的使用需求。

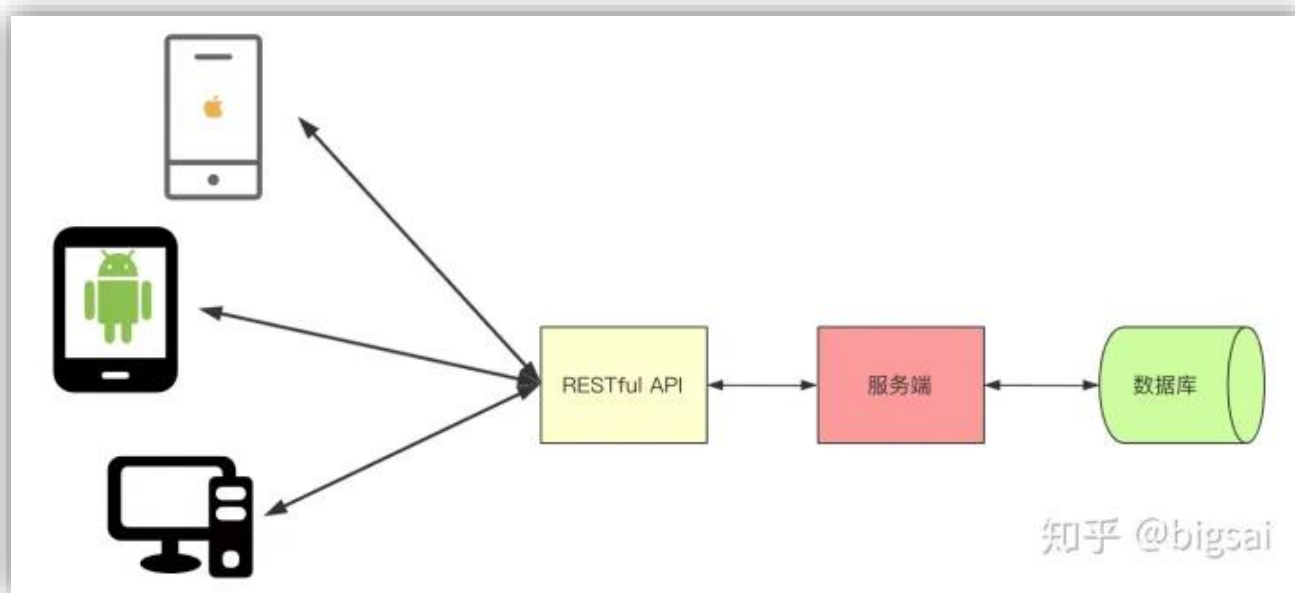


❑但是随着互联网和移动设备的发展，人们对Web应用的使用需求也增加，传统的动态页面由于低效率而渐渐被HTML+JavaScript(Ajax)的前后端分离所取代，并且安卓、IOS、小程序等形式客户端层出不穷，客户端的种类出现多元化，而客户端和服务端就需要接口进行通信，但接口的规范性就又成了一个问题：



RESTful API

- 所以一套**结构清晰、符合标准、易于理解、扩展方便**让大部分人都能够理解接受的接口风格就显得越来越重要，RESTful风格的接口(RESTful API)刚好有以上特点
- REST (Representational State Transfer, 是一种基于 HTTP 协议的网络服务架构风格, 通过使用如 GET、POST、PUT、DELETE 等 HTTP 方法来执行数据操作。**REST并没有一个明确的标准, 而更像是一种设计的风格, 基于这个风格设计的软件可以更简洁, 更有层次, 更易于实现缓存等机制。**



基本概念

- 资源与URI
- 统一资源接口
- 资源的表述
- 资源的链接
- 状态的转移

资源与URI

□ REST全称是表述性状态转移，其中的表述指的就是资源的表述。任何事物，只要有被引用到的必要，它就是一个资源。资源可以是实体，也可以只是一个抽象概念。下面是一些资源的例子：

- 某用户的手机号码
- 某用户的个人信息
- 最多用户订购的GPRS套餐
- 两个产品之间的依赖关系
- 某用户可以办理的优惠套餐
- 某手机号码的潜在价值

资源与URI

- ❑ 为了识别资源，定义URI (Uniform Resource Identifier) 来作为资源的唯一标记
- ❑ URI既可以看成是资源的地址，也可以看成是资源的名称。
- ❑ URI的设计应该遵循可寻址性原则，具有自描述性，需要在形式上给人以直觉上的关联。一些URL例子：
 - ✓ <https://github.com/git>
 - ✓ <https://github.com/git/git>
 - ✓ <https://github.com/git/git/blob/master/block-sha1/sha1.h>

资源与URI

□ URL设计技巧

✓ 使用_或-来让URI可读性更好

曾经Web上的URI都是冰冷的数字或者无意义的字符串，但现在越来越多的网站使用_或-来分隔一些单词，让URI看上去更为人性化。例如国内比较出名的开源中国社区，它上面的新闻地址就采用这种风格，如<http://www.oschina.net/news/38119/oschina-translate-reward-plan>。

✓ 使用/来表示资源的层级关系

例如/git/git/commit/e3af72cdafab5993d18fae056f87e1d675913d08就表示了一个多级的资源，指的是git用户的git项目的某次提交记录，又例如/orders/2012/10可以用来表示2012年10月的订单记录。

✓ 使用?用来过滤资源

可以把?用于对资源的过滤，例如/git/git/pulls用来表示git项目的所有推入请求，而/pulls?state=closed用来表示git项目中已经关闭的推入请求，这种URL通常对应的是一些特定条件的查询结果或算法运算结果。

统一资源接口

□ RESTful应该遵循统一的接口原则，包含一组受限的预定义的操作，主要包括以下四种

- ✓ GET：获取，幂等
- ✓ PUT：更新或者替换资源，幂等
- ✓ POST：创建新资源，非幂等
- ✓ DELETE：删除资源，幂等

□ 幂等：指对于同一个操作，无论执行多少次，结果都一样。具体来说，如果一个操作是幂等的，那么在执行完一次操作后再多次重复执行这个操作，得到的结果都应该与执行一次的结果相同。

Example

□ 我们以一个具体的例子来说明实现RESTful API的GET接口通常步骤

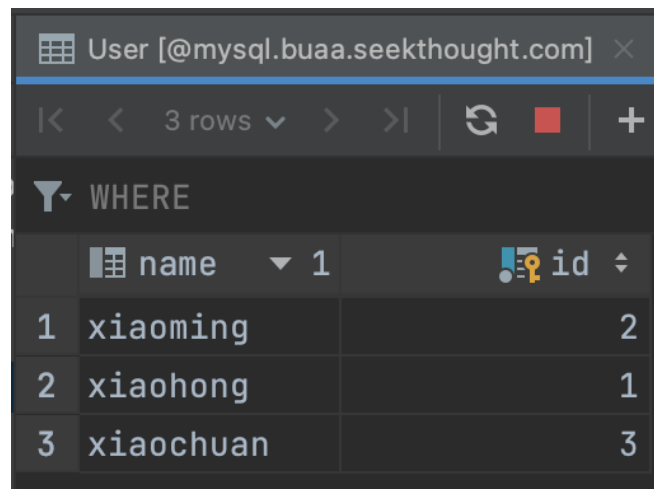
✓ 设计数据库表格

假设我们有一个名为User的表格（右图），其中包含id（作为主键）和name两个属性

✓ 创建RESTful路由

接下来，需要在你的应用程序中创建 RESTful 路由来处理 GET 请求。对于 User 表格，你可以创建以下路由：

- GET /User: 返回所有项的列表。
- GET /User/{id}: 返回特定 id 的项的详细信息。



The screenshot shows a MySQL database interface with a table named 'User'. The table has two columns: 'name' and 'id'. The 'id' column is marked as the primary key. There are three rows of data in the table.

	name	id
1	xiaoming	2
2	xiaohong	1
3	xiaochuan	3

Example

✓处理GET请求

对于每个路由，需要编写相应的处理函数来从数据库中获取数据并返回给客户端。

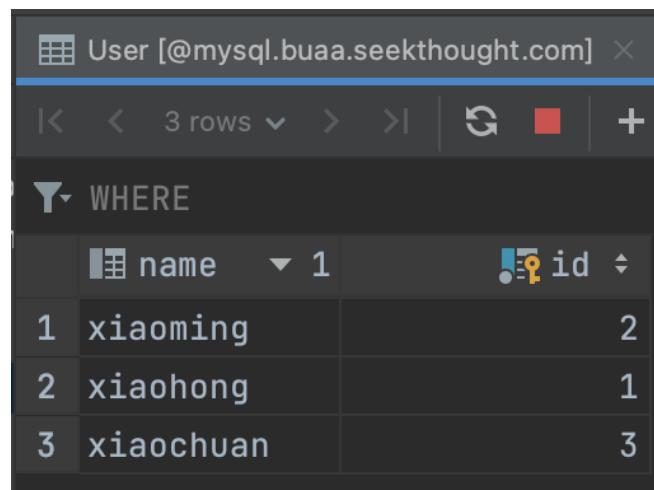
- 对于 GET /User，可以执行类似于 `SELECT * FROM Items` 的查询来获取所有项的信息，并将结果作为 JSON 格式返回给客户端。
- 对于 GET /User/{id}，可以执行类似于 `SELECT * FROM Items WHERE id = {id}` 的查询来获取特定 id 的项的信息，并将结果作为 JSON 格式返回给客户端。

✓处理异常情况

在处理 GET 请求时，还应该考虑到可能出现的异常情况，例如请求的资源不存在或者数据库查询出错等情况。对于这些情况，可以返回适当的 HTTP 状态码和错误信息^[1]给客户端。

✓测试接口

最后，可以使用工具如 Postman 或者 cURL 来测试你的 GET 接口，确保它们能够正确地返回所需的数据和状态码。



The screenshot shows a MySQL database interface with the title 'User [@mysql.buaa.seekthought.com]'. It displays a table with 3 rows. The table has two columns: 'name' and 'id'. The data is as follows:

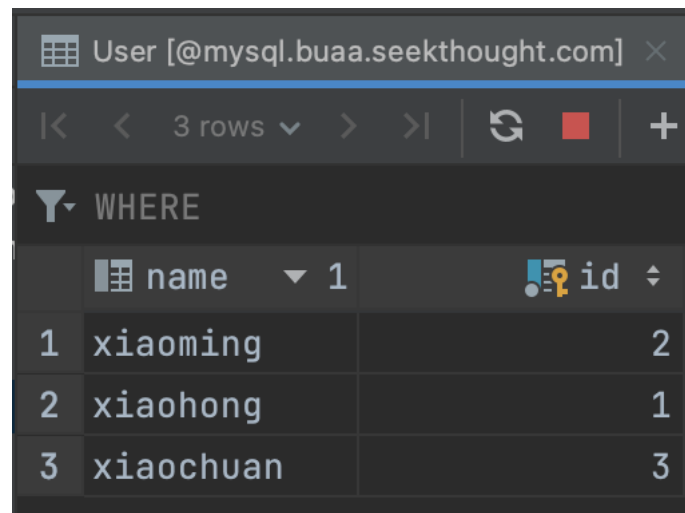
	name	id
1	xiaoming	2
2	xiaohong	1
3	xiaochuan	3

[1] <https://www.runoob.com/w3cnote/restful-architecture.html#:~:text=%E7%9A%84%E5%85%B8%E5%9E%8B%E7%94%A8%E6%B3%95%3A-,GET,-%E5%AE%89%E5%85%A8%E4%B8%94%E5%B9%82>

Example

□ 对于其他的接口，设计思想和步骤也是类似的

- ✓ GET /users/3: 请求id为3的user信息
- ✓ POST /users: 创建一个新的user，数据放在body中
- ✓ PUT /users/2: 更新user id=2的信息
- ✓ DELETE /users/1: 删除user id=1的用户



The screenshot shows a MySQL database client window titled 'User [@mysql.buaa.seekthought.com]'. The window displays a table with 3 rows. The table has two columns: 'name' and 'id'. The data is as follows:

	name	id
1	xiaoming	2
2	xiaohong	1
3	xiaochuan	3

资源的表述

- 资源是以一组资源表述形式（Resource Representation）与客户端交互，表述是资源在请求响应中的具体呈现形式，常用的资源表述形式一般有以下几种：
- ✓ 1. JSON格式：JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，通常用来表述Web APIs传输的数据。JSON利用JavaScript语法的对象表示法，描述类似于数组和字典的可嵌套结构数据。
 - ✓ 2. XML格式：XML（Extensible Markup Language）是一种标记语言，它和HTML一样，也是一种文本文件格式。但不同于HTML，XML并不预定义任何标签，而是允许用户根据需要自定义标签。
 - ✓ 3. HTML格式：HTML（Hypertext Markup Language）是一种标记语言，通常用于编写Web页面。虽然HTML最初是配合HTTP协议用来展示页面的，但也可以用于RESTful API中资源的表述。
 - ✓ 4. 图片、音频、视频等二进制格式：除了文本格式的表述外，因为有些资源的表述可能是图片、音频或者视频等二进制格式的，这时候可以将这些资源以二进制流的方式保存。
 - ✓ 5. 其他格式：比如格式化数据、CSV等。

Example

- 通过HTTP内容协商，客户端可以通过Accept头请求一种特定格式的表述，服务端则通过Content-Type告诉客户端资源的表述形式。

```
# Request
GET https://api.github.com/orgs/github HTTP/1.1
Accept: text/xml

# Response
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8

<org>
  <login>github</login>
  <id>1</id>
  <url>https://api.github.com/orgs/github</url>
  <avatar_url>https://github.com/images/error/octocat_hap
  <name>github</name>
  <company>GitHub</company>
  <blog>https://github.com/blog</blog>
  <location>San Francisco</location>
  <email>octocat@github.com</email>
  <public_repos>2</public_repos>
  <public_gists>1</public_gists>
  <followers>20</followers>
  <following>0</following>
  <html_url>https://github.com/octocat</html_url>
  <created_at>2008-01-14T04:33:35Z</created_at>
  <type>Organization</type>
</org>
```

```
# Request
GET https://api.github.com/orgs/github/repos HTTP/1.1
Accept: application/json

# Response
HTTP/1.1 Status: 200 OK
Link: <https://api.github.com/orgs/github/repos?page=2>; rel="next",
      <https://api.github.com/orgs/github/repos?page=3>; rel="last"
Content-Type: application/json; charset=utf-8

[
  {
    "id": 1296269,
    "owner": {
      "login": "octocat",
      "id": 1,
      "avatar_url": "https://github.com/images/error/octocat_happy.gif",
      "gravatar_id": "somehexcode",
      "url": "https://api.github.com/users/octocat"
    },
    "name": "Hello-World",
    "full_name": "octocat/Hello-World",
    "description": "This your first repo!",
    "private": false,
    "fork": false,
    "url": "https://api.github.com/repos/octocat/Hello-World",
    "html_url": "https://github.com/octocat/Hello-World",
    ...
  }
]
```

资源的链接

- ❑ RESTful API 中的资源通常包含一些超媒体链接，这些链接可以指向相关资源、资源操作的URI等信息。这种链接通常使用超文本标记语言（HTML）、Atom、Markdown、JSON等格式来表达。
- ❑ 一个资源的表述中的链接通常包括以下三种类型：
 - ✓ 1. 自关联链接（self-link）：指向该资源自身的URI，可以获取资源自身的详细信息。
 - ✓ 2. 相关链接（related-link）：指向与该资源相关的资源URI，这些资源与该资源相关联，可以帮助客户端构建更加精确的查询语句。
 - ✓ 3. 操作链接（operation-link）：指向对该资源进行操作的URI，包括读取、更新、删除、新建等操作。
- ❑ 通常，链接可以使用HTTP头部字段 Link 来添加到API中的响应中，例如：

```
Link: <http://example.com/self>; rel="self",  
<http://example.com/related>; rel="related",  
<http://example.com/operation>; rel="operation"
```
- ❑ 通过添加链接，RESTful API 可以更好地导航和组织资源，构建更加灵活的客户端。

Example

- 下面展示的是github获取某个组织下的项目列表的请求，可以看到在响应头里边增加Link头告诉客户端怎么访问下一页和最后一页的记录。而在响应体里边，用url来链接项目所有者和项目地址。

```
1 # Request
2 GET https://api.github.com/orgs/github/repos HTTP/1.1
3 Accept: application/json
4
5 # Response
6 HTTP/1.1 Status: 200 OK
7 Link: <https://api.github.com/orgs/github/repos?page=2>; rel="next",
8       <https://api.github.com/orgs/github/repos?page=3>; rel="last"
9 Content-Type: application/json; charset=utf-8
10
11 [
12   {
13     "id": 1296269,
14     "owner": {
15       "login": "octocat",
16       "id": 1,
17       "avatar_url": "https://github.com/images/error/octocat_happy.gif",
18       "gravatar_id": "somehexcode",
19       "url": "https://api.github.com/users/octocat"
20     },
21     "name": "Hello-World",
22     "full_name": "octocat/Hello-World",
23     "description": "This your first repo!",
24     "private": false,
25     "fork": false,
26     "url": "https://api.github.com/repos/octocat/Hello-World",
27     "html_url": "https://github.com/octocat/Hello-World",
28     ...
29   }
30 ]
```

状态的转移

□ 服务端与客户端的状态管理

- ✓ 服务端维护资源状态
- ✓ 客户端维护应用状态

□ 客户端和服务端之间的通信要是无状态的，即每一次请求包含处理该请求的一切信息

□ 状态转移："会话"状态不是作为资源状态保存在服务端的，而是被客户端作为应用状态进行跟踪的。客户端应用状态在服务端提供的超媒体的指引下发生变迁。服务端通过超媒体告诉客户端当前状态有哪些后续状态可以进入。

Docker

❑容器：是一种轻量级的虚拟化技术，通过隔离应用程序和其依赖的环境来实现应用程序的快速交付和部署。与传统的虚拟化方式相比，Docker容器使用的底层资源更少，启动速度更快，因此可以更高效地利用计算机资源。

❑使用Docker部署MySQL：

- ✓环境隔离：避免不同版本的冲突
- ✓可靠性：可以快速恢复宕机的MySQL服务
- ✓方便测试：部署自动化测试
- ✓.....

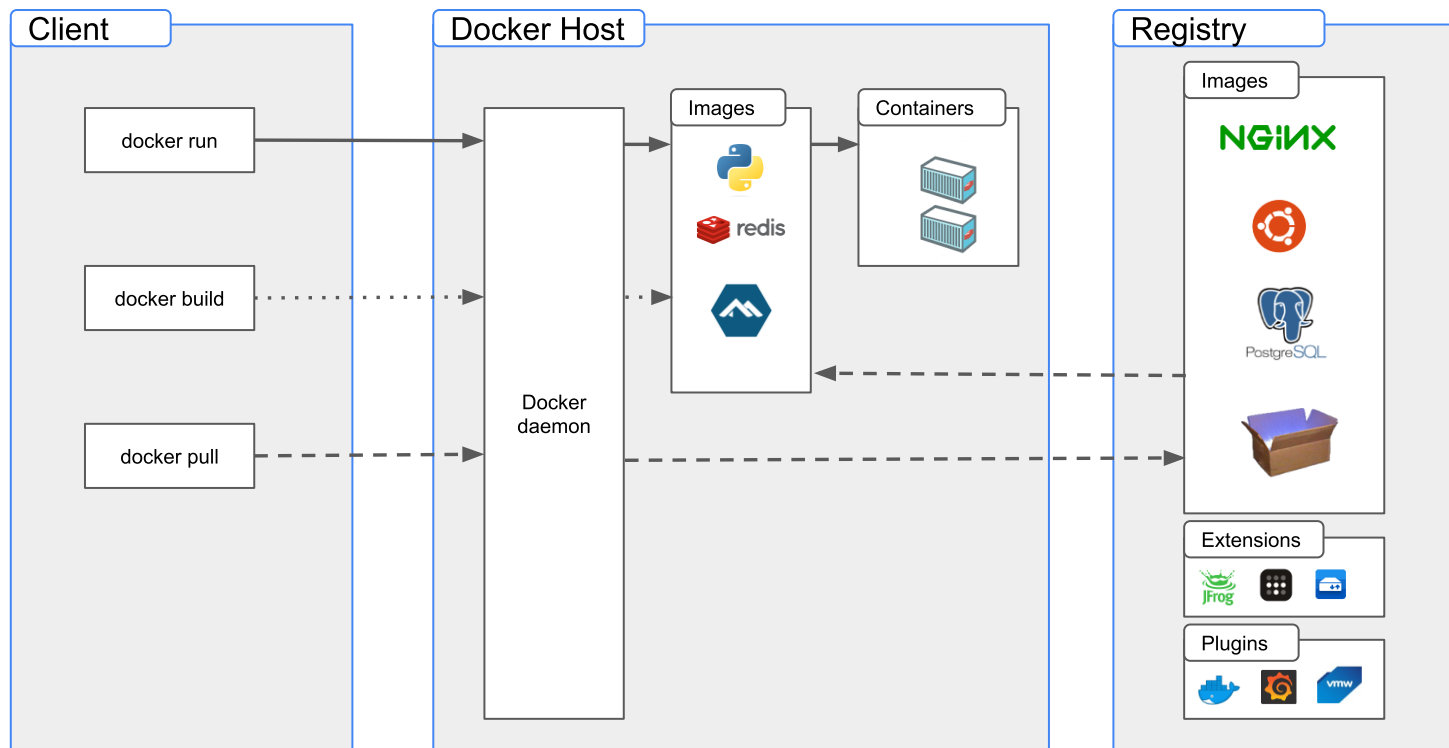
Docker architecture

❑ Docker daemon

❑ Docker Desktop

❑ Docker client

❑ Docker registry



Docker 安装

❑ Docker支持不同平台windows、Mac、Linux

- ✓ Mac: `brew cask install docker` 或者 直接下载dmg安装
- ✓ Windows: 下载exe程序
- ✓ Linux: 不同版本选择不同的package



Docker Desktop for Mac

A native application using the macOS sandbox security model which delivers all Docker tools to your Mac.



Docker Desktop for Windows

A native Windows application which delivers all Docker tools to your Windows computer.



Docker Desktop for Linux

A native Linux application which delivers all Docker tools to your Linux computer.

[\[1\] Docker Desktop](#)

Usage

- ❑ Docker pull/images
- ❑ Docker run
- ❑ Docker build

Basic

□ Container

- ✓ 实际运行应用程序的一个“虚拟机”，包含操作系统、用户文件和元数据

□ Image

- ✓ 只读的Docker容器模版，其实就是系统镜像文件

□ Dockerfile

- ✓ 脚本文件，用来创建image

Docker images

- ❑ image: Docker image 是一个只读的模板，用于创建 Docker 容器。一个 Docker image 包含了应用程序和运行时所需的所有依赖库、源代码、配置文件和其他文件。
- ❑ Image通常存储在仓库（docker hub），可以通过docker pull 拉取指定的image，就像从Github拉取仓库代码一样
 - ✓ `docker pull mysql`
- ❑ Pull后可以使用images查看本地的拉取image
 - ✓ `docker images`

Docker container

- ❑ container是一个实际运行的容器实例，属于runtime的范畴
- ❑ container和image可以类比为编程语言中的类定义和运行实例

image	container
静态的	动态可读写
定义的模版	实例
可以独立存在，不需要container	必须基于一个image存在
不需要内存运行	需要计算资源运行
一次创建就是一个独立的存在	一个image可以创建多个container

Docker pull

❑ `docker pull [选项] [Docker Registry 地址[:端口号]/]仓库名[:标签]`

❑ image在创建的时候可以添加标签，通常用来指定对应image的版本。
比如：

```
docker pull mysql:latest
```



```
docker pull mysql:oracle
```



```
docker pull mysql:8.0.32
```



这里需要注意，Mac m1架构暂时不支持5.7版本，因此我们要求统一使用8.0版本的MySQL

Docker run

❑ 当image拉取成功后，有了镜像文件，就可以启动一个容器来执行程序，此时需要用到的命令是 **docker run**

✓ `docker run [OPTIONS] IMAGE [COMMAND] [ARG...]`

❑ 这个命令启动一个容器，拥有操作系统的能力和内置的软件依赖。以MySQL服务为例，我们可以通过docker run启动刚才拉取的MySQL镜像，在本地部署docker MySQL服务。

✓ `docker run --name test-mysql -e MYSQL_ROOT_PASSWORD=pw -d mysql`

❑ 上面的命令会在本地启动MySQL服务，可以通过

✓ `docker ps -a` 命令查看此时在运行的容器(container)

```
(base) ~ % docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
8100f1d48230	mysql	"docker-entrypoint.s..."	21 seconds ago	Up 20 seconds	3306/tcp, 33060/tcp	test-mysql

Docker exec

- ❑ ``docker run`` 和 ``docker exec`` 都可以进入容器内部，但它们的作用不同。
- ❑ ``docker run`` 用于创建并启动一个新的容器，可以在启动容器时通过添加 ``-it`` 参数，使其在前台交互模式下运行，此时可以进入容器内部并执行你需要的命令，也可以通过 ``docker attach`` 命令重新连接已经在后台运行的容器，但这又只能将容器变为前台交互模式的状态。
- ❑ ``docker exec`` 则是在已经启动的容器内部运行一个新的进程，可以在执行 ``docker exec`` 命令时添加 ``-it`` 参数，使其在前台交互模式下运行，此时可以进入容器内部并执行你需要的命令。与 ``docker attach`` 不同的是，``docker exec`` 进入容器后不会影响以前容器内的运行状态，也不会影响容器的前台交互模式。
- ❑ 因此，如果需要创建新的容器来执行操作，可以使用 ``docker run``；如果需要进入已经运行的容器内部，可以使用 ``docker exec``。

Docker Network

- ❑ 这里还需要介绍一个概念：network
- ❑ Docker本身是一个隔离的环境，所以运行在本地上的docker容器和本地的网络其实是不互通的，需要我们手动加一下一些配置才可以使得在docker容器内也正常访问网络
- ❑ 上面部署的MySQL服务，实际测试就会发现只有在容器内部才可以访问，在本地暂时无法访问，下面介绍一下解决方案：
 - ✓ docker run 端口映射
 - ✓ docker 网络
 - ✓ docker 主机模式

Docker Network

❑ 端口映射：对于一些网络应用，docker提供端口映射，将容器内的网络服务映射到本地的一个端口上，这样，访问端口的时候就等于访问容器内部的网络服务

✓ `docker run -d -p 80:80 nginx:alpine`

❑ Docker network：创建一个docker容器之间的局域网，使用服务的时候需要将容器添加到该网络中

✓ `docker network create -d bridge mysql-net`

✓ `docker run -it --rm -name xxx -network mysql-net ubuntu bash`

❑ Host模式：容器和主机共享网络。

✓ `docker run -it --net=host ubuntu bash`

Docker 数据持久化

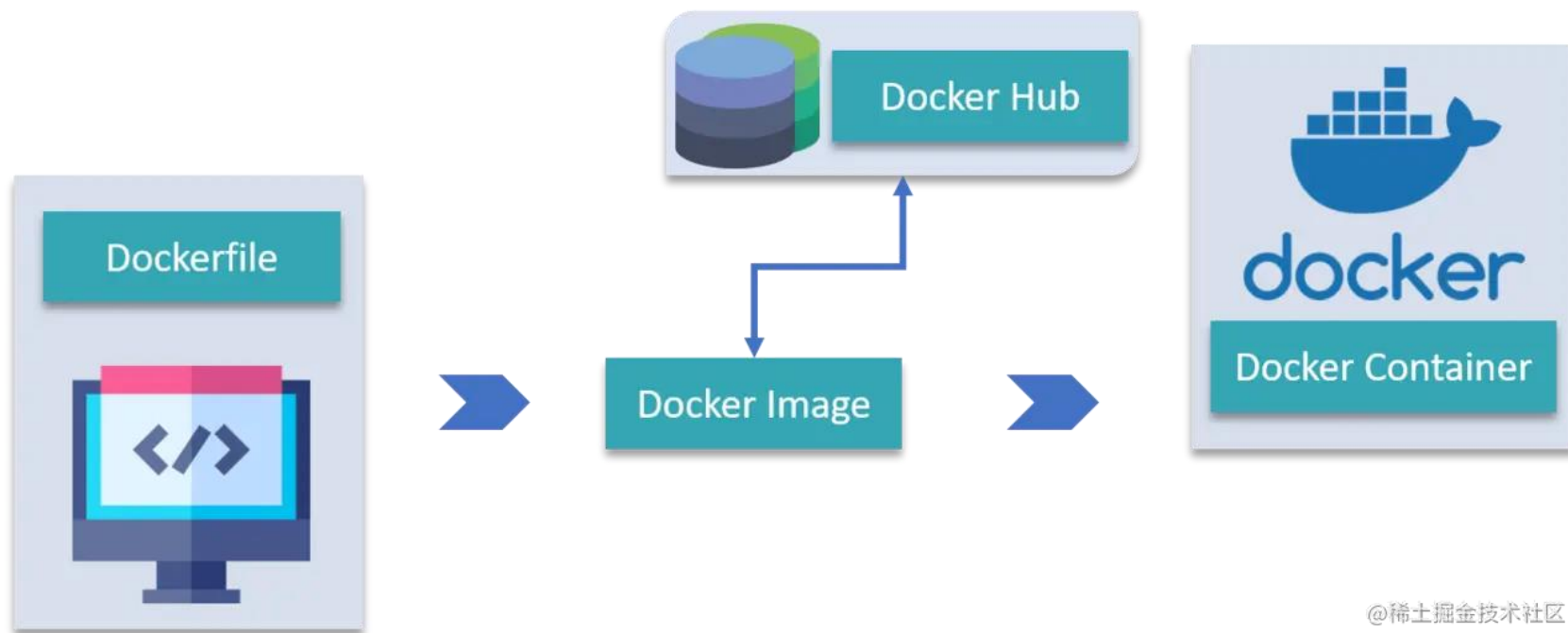
- ❑ Docker容器是一个隔离的环境，所有的读写都是被控制在自己的环境中，如果遇到系统故障，重新启动了容器，那么之前写的数据都会丢失，因此docker 容器的持久化也是一个需要注意的问题。
- ❑ 从原理上来讲：docker容器使用了union filesystem机制，将docker容器数据按照一层一层只读层的方式堆叠而来，在最上面一层是读写层。但是union fs是一个临时的存储，容器关闭后内部写入的数据就丢失了，对于MySQL服务而言，写入到数据库的表信息都会丢失，这就导致每一次启动MySQL服务都需要重新插入数据，显然这不是用户希望看到的。

Docker 挂载目录

- ❑ 这里只介绍挂载目录的方式，其他方法同学们可以自行学习。
- ❑ Docker run启动容器的时候可以挂载本地目录到容器内部，然后将数据都写入到这个指定的目录，这样即使容器失败，数据也不会丢失。
 - ✓ `docker run --name test-mysql -v /my/mysql/datadir:/var/lib/mysql`
 - ✓ -v参数指定挂载的目录和容器内挂载的位置，上面的例子是将本地/my/mysql/datadir挂载到容器内部系统的/var/lib/mysql目录

Docker build

❑ Dockerfile: 用于构建image的文本文件，内容包含了如何构建image的指令以及说明。



@稀土掘金技术社区

Dockerfile

❑ Dockerfile总是从一个base开始构建

- ✓ `FROM ubuntu:latest`: 意思是从ubuntu镜像开始构建自己的镜像, 通常选择是在一些已有的镜像服务基础上进行构建, 而不是从0开始

❑ RUN: 用于执行一个命令, 比如安装一个新的依赖, 解压缩等

- ✓ `RUN apt install python3`
- ✓ `RUN tar -xvf redis.tar.gz`

❑ COPY: 把本地文件拷贝到镜像中, 通常是一些本地的配置文件, 需要拷贝到镜像中作为服务的配置

- ✓ `COPY config.yaml /usr/share/nginx/config.yaml`

Dockerfile

❑ ENTRYPOINT: 用于设置容器启动时默认执行的命令和程序

✓ ENTRYPOINT ["echo", "hello world!"]

✓ Entrypoint

❑ `docker build -t homework:v1 -f my_dockerfile`

✓ 上面的命令用于从dockerfile构建my_dockerfile镜像, 版本由tag指定为v1

Example

```
1  # 使用Python作为基础映像
2  FROM python:3.8-slim-buster
3
4  # 设置工作目录
5  WORKDIR /code
6
7  # 将本地目录中的依赖项安装到容器中
8  COPY requirements.txt .
9  RUN pip install -r requirements.txt
10
11 # 复制应用程序代码到容器中
12 COPY . .
13
14 # 设置环境变量和容器启动时执行的命令
15 ENV FLASK_APP=app.py
16 ENTRYPOINT ["flask"]
17 CMD ["run", "--host=0.0.0.0"]
```

数据库系统开发

- 数据库访问与Python MySQL Client

- ORM

- 延伸阅读：开发模式

- Docker usage

- 作业介绍



数据库系统开发大作业

□ 要求

- ✓ 本次大作业详细要求请见《2024年数据管理技术大作业具体要求.pdf》。

□ 描述

- ✓ 本次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。
- ✓ 测试：
 - MySQL查询测试
 - 索引查询测试
 - 触发器测试
 - RESTful服务测试



1 数据库开发

□ 要求

- ✓ 使用ORM框架完成5个数据库模型的代码开发，索引和触发器定义以及数据注入

□ 描述

- ✓ 本次大作业所提交的目录文件如图1，数据库的开发部分位于Test1-ORM.py文件
- ✓ 我们已经提供了Departments数据库模型的定义代码（图2）以及Departments表的数据注入（图3）
- ✓ TODO：在Test1-ORM.py文件中，参照样例，完善实验全部内容

```
C:.\n|  insert.sh\n|  start.sh\n└─src\n\n    Test1-ORM.py\n    Test2-RESTful.py
```

图1 文件目录结构

```
class Departments(db.Model):
    dept_no = db.Column(db.CHAR(4), primary_key=True)
    dept_name = db.Column(db.String(40), nullable=False, unique=True)
    dept_ems = db.relationship('Dept_emp', backref='department', lazy=True, cascade='all, delete')
    dept_managers = db.relationship('Dept_manager', backref='department', lazy=True, cascade='all, delete')

    def to_dict(self):
        return {
            'dept_no': self.dept_no,
            'dept_name': self.dept_name
        }
```

图2 Departments数据库模型

```
# department注入
reader = read_csv_file('./departments.csv')
# data = [{'dept_no': row['dept_no'], 'dept_name': row['dept_name']} for row in reader]
rows = []
for row in reader:
    department = Departments(
        dept_no=row['dept_no'],
        dept_name=row['dept_name']
    )
    rows.append(department)

session.bulk_save_objects(rows)
session.commit()
```

图3 Departments数据注入



2 REST ful接口

■ 同学们需要提供REST ful接口给前端服务，具体而言，我们要求提供以下四个接口。

- 数据插入接口
- 数据更新接口
- 数据删除接口
- 数据查询接口

✓ 数据库的开发部分位于Test2-RESTful.py文件

✓ 我们已经提供了数据插入接口（图1）

✓ TODO：在Test2-RESTful.py文件中，参照样例，完善实验全部内容

```
# 插入数据
@app.route('/api/v1/<table_name>', methods=['POST'])
def insert_data(table_name):
    data = request.json
    rows = data['rows']
    cursor = db.cursor()
    for row in rows:
        keys = ', '.join(row.keys())
        values = ', '.join([f'"{value}"' for value in row.values()])

        sql = f'insert into {table_name}({keys}) values({values})'
        cursor.execute(sql)
    db.commit()
    return jsonify({'message': 'data inserted successfully'}), 201
```

图1 RESTful插入接口样例



3 数据库选择

■推荐选项

- ✓ 我们为所有同学提供插入的数据以及数据库链接，同学们可以直接使用助教提供的数据库
 - 数据库连接地址：119.3.248.160，用户名：学号，密码：学号
- ✓ 可视化连接方式：pymysql或mycli或者数据库可视化工具（DBeaver，DataGrip，Navicat等）

```
import pymysql

# 打开数据库连接
db = pymysql.connect(host='119.3.248.160',
                    user='2237xxxx', # 学号
                    password='2237xxxx', # 学号
                    port=3306 # 端口
                    )
```

图1 pymysql连接方式

```
pip install mycli # 按照mycli
mycli -h 119.3.248.160 -P 3306 -u <学号> -p <学号>
```

图2 mycli连接方式



图3 DBeaver连接方式

注意

- 请大家务必使用**代码连接数据库**，确保通过RESTful服务可以插入全部的数据，可视化连接仅作为辅助工具
- 请大家不要使用他人的账号连接数据库，因此造成的问题会受到惩罚！**
- 本次大作业的数据库提供的是MySQL 8.0及以上版本，同学们需要基于此版本进行代码开发。



4 查询测试

■这部分的测试分两类：

■SQL相关测试：

- 基本查询：针对数据表内容的MySQL语句查询；
- 索引查询：对特定的数据表进行索引查询（需要同学们预先使用代码对数据表进行索引建立）
- 触发器查询：针对触发器逻辑进行mysql查询

■ RESTful服务测试：

- 针对大家实现的RESTful服务进行测试

分数构成

□期中大作业总分10分 (5+2+3)

- ✓完成基本的CURD功能以及通过MySQL语句测试：5分。
- ✓通过索引测试和触发器：2分。
- ✓完成RESTful接口服务：3分。