实验五

王为 2311605

2024 年 10 月 20 日

1 实验目标

实现一个简易的用户注册功能，具体功能如下：

1. 使用字典存储用户信息，其中键为用户名，值为一个包含用户电子邮件和密码的字典。这种结构便于快速访问和管理用户数据。
2. 使用正则表达式验证电子邮件格式是否正确。电子邮件的格式应符合以下规则：
   - 包含一个`'@'` 符号，且`'@'` 符号前后各有字符。
   - `'@'` 后必须有一个域名，且域名必须至少包含一个'.'，以确保有效性。
3. 验证输入的用户名是否已经存在于字典中。如果已存在，则提示用户重新输入，确保每个用户名的唯一性。
4. 使用正则表达式验证密码的复杂度。密码必须至少包含 8 个字符，并且至少包含以下四种类型的字符：
   - 一个大写字母（A-Z）
   - 一个小写字母（a-z）
   - 一个数字（0-9）
   - 一个特殊字符（如 `'@'`, $, !, 等）
5. 将用户输入的密码进行哈希加密，使用 `bcrypt` 库存储加密后的密码，增强用户信息的安全性。

6. 将字典中存储的用户信息（包括用户名、电子邮件和加密密码）插入到 MySQL 数据库中，以实现数据的持久化存储。用户信息存储在名为 `users` 的表中，确保字段的唯一性和完整性。

通过实现上述功能，我们能够建立一个基本的用户注册系统，既能进行输入验证，又能确保数据安全性和持久性。

程序具体流程见图一。

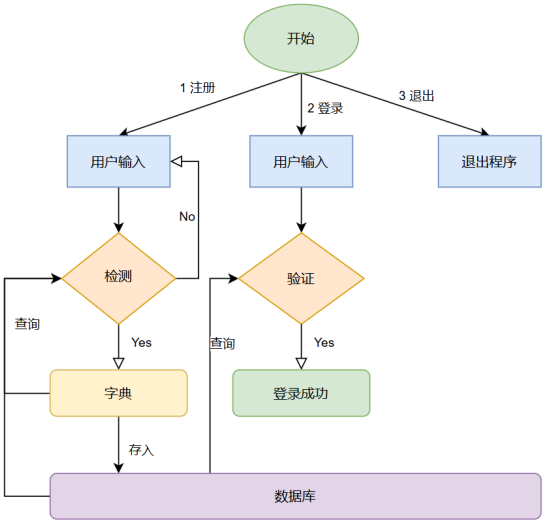

图 1: 程序流程图

2 正则表达式

在本节中，我们将详细说明如何使用正则表达式来验证用户的电子邮件格式和密码复杂度。正则表达式是一种强大的文本处理工具，通过模式匹配来检查字符串的格式和内容。

2.1 电子邮件验证

电子邮件的验证使用以下函数实现：

```python
def validate_email(email):
    """ 验证电子邮件格式 """
    email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return re.match(email_pattern, email) is not None
```

在此函数中，email_pattern 是一个正则表达式，具体含义如下：

- ^ 表示字符串的开始，确保整个字符串符合模式。
- [a-zA-Z0-9._%+-]+ 匹配电子邮件的用户名部分，允许字母、数字以及一些特定符号（如点、下划线、百分号、加号和减号）。+ 表示前面的字符组可以出现一次或多次。
- @ 代表电子邮件的分隔符，标识用户名和域名的边界。
- [a-zA-Z0-9.-]+ 匹配域名部分，允许字母、数字、点和破折号。
- \. 确保紧接在域名之后的是一个点，表明顶级域名的开始。
- [a-zA-Z]{2,} 确保顶级域名至少有两个字母，以避免不合法的顶级域名（如.com）。
- $ 表示字符串的结束，确保模式匹配到字符串的末尾。

函数使用 re.match 来匹配输入的电子邮件。如果匹配成功，返回 True，否则返回 False。这种方法有效地防止了格式不正确的电子邮件地址被接受。

2.2 密码验证

密码验证的函数实现如下：

```python
def validate_password(password):
    """ 验证密码复杂度 """
    password_pattern = r'^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za
        -z\d@$!%*?&]{8,}$'
    return re.match(password_pattern, password) is not None
```

在此函数中，password_pattern 是一个正则表达式，具体含义如下：

- ^ 表示字符串的开始，确保整个字符串符合模式。

- `(?=.*[a-z])` 确保密码至少包含一个小写字母。这是一个前瞻性断言，允许我们检查条件但不消耗字符。
- `(?=.*[A-Z])` 确保密码至少包含一个大写字母。
- `(?=.*\d)` 确保密码至少包含一个数字。
- `(?=.*[@$!%*?&])` 确保密码至少包含一个特殊字符，增加密码的复杂度和安全性。
- `[A-Za-z\d@$!%*?&;:"#_]{8,}` 确保密码长度至少为 8 个字符，且只能包含字母、数字或特殊字符。
- `$` 表示字符串的结束，确保模式匹配到字符串的末尾。

同样，函数使用 `re.match` 来验证密码的复杂度，并返回验证结果。通过这种方式，确保用户设置的密码既安全又符合要求。

2.3 正则表达式的优势

正则表达式在数据验证方面具有以下优势：

- **简洁性**：通过简单的模式表达，可以实现复杂的验证规则，减少了代码量。
- **灵活性**：可以轻松调整模式，以满足不同的需求。
- **效率**：在处理大量数据时，正则表达式能够快速匹配和验证，提高了程序的效率。

正则表达式为数据验证提供了一种强大而灵活的方式，使得程序能够有效地处理用户输入，提高了安全性和用户体验。

2.4 正则表达式的深入理解

正则表达式是一个强大的工具，能够处理各种复杂的文本匹配需求。以下是一些常用的正则表达式元素及其用途：

- `.`：匹配除换行符以外的任意单个字符。例如，`a.b` 可以匹配 `acb`、`axb` 等。

- `*`：匹配前一个字符零次或多次。例如，`ab*` 可以匹配 `a`、`ab`、`abb` 等。
- `+`：匹配前一个字符一次或多次。例如，`ab+` 可以匹配 `ab`、`abb` 等，但不能匹配 `a`。
- `?`：匹配前一个字符零次或一次。例如，`ab?` 可以匹配 `a` 或 `ab`。
- `|`：表示“或”，例如 `abc|def` 匹配”abc” 或”def”。
- `( ... )`：分组，用于捕获匹配的子表达式。例如，`(abc)+` 表示一个或多个连续的 `abc`。
- `[abc]`：匹配字符集中的任意一个字符。例如，`[aeiou]` 匹配任意元音字母。
- `{n}`：精确匹配 n 次。例如，`[0-9]{3}` 匹配任意三个数字。
- `\d`、`\w`、`\s`：分别匹配数字、字母或数字的组合和空白字符。

正则表达式的这些基本元素允许我们构造复杂的匹配模式，从而处理各种文本处理需求，如验证输入、搜索特定字符串等。

2.5 正则表达式的应用示例

以下是一些实际应用正则表达式的示例：

1. ** 匹配电话号码 **：

```
phone_pattern = r'^\+?[\d\s-]{7,15}$'
```

该正则表达式用于匹配国际电话号码。解释如下：

- `^\+?`：可选的加号开头，表示国际区号。
- `[\d\s-]{7,15}`：匹配 7 到 15 位的数字、空格或破折号，确保电话号码的长度合理。
- `$`：表示字符串的结束，确保匹配整个输入。

2. ** 匹配日期格式 **：

```
date_pattern = r'^\d{4}-\d{2}-\d{2}$'
```

该正则表达式用于匹配日期格式（YYYY-MM-DD）。解释如下：

- \d{4}：表示年份，必须是四位数字。
- -：分隔符，必须为连字符。
- \d{2}-\d{2}$：表示月份和日期，必须是两位数字。

通过这些示例，可以看到正则表达式在处理字符串数据时的灵活性和强大功能。使用正则表达式，程序可以更高效地验证和处理用户输入，从而提高用户体验和数据质量。

3 字典的使用

在代码中，我们使用了一个名为 user_data 的字典来存储用户信息。字典的键是用户名，而值是一个包含用户电子邮件和密码的字典。这种数据结构使得用户信息的存储和访问更加简洁和高效。

```
# 存储用户信息的字典
user_data = {}
```

字典的主要特点是它是一个无序的、可变的、且通过键访问的集合，这使得它非常适合用于存储和管理具有唯一标识符的对象。在我们的示例中，用户名作为键能够确保每个用户信息的唯一性。

3.1 字典的基本操作

字典支持丰富的操作，包括添加、更新、删除和查询。以下是一些基本操作的示例：

```
# 添加新用户信息
user_data['alice'] = {'email': 'alice@example.com', 'password': 'hashed_pw'}

# 更新用户电子邮件
user_data['alice']['email'] = 'alice_new@example.com'

# 删除用户信息
del user_data['alice']

# 查询用户信息
if 'alice' in user_data:
    print(user_data['alice'])
```

这些基本操作展示了字典在信息管理中的灵活性，允许开发者快速修改和访问用户数据。

3.2 用户注册

在 register_user 函数中，首先检查用户名是否已存在于字典中：

```
if username in user_data:
    print("用户名已存在，请重新输入。")
```

如果用户名不存在，则将用户的电子邮件和哈希密码存入字典：

```
user_data[username] = {
    'email': email,
    'password': hashed_password.decode('utf-8')
}
```

这种方式使得用户信息的存储结构清晰，易于扩展和维护。字典的嵌套特性允许我们轻松地为每个用户添加更多信息，例如注册时间、用户角色和用户状态：

```
user_data[username]['registration_date'] = registration_date
user_data[username]['role'] = 'user'
user_data[username]['is_active'] = True
```

通过这种结构，我们能够在一个地方集中管理所有与用户相关的信息。

3.3 用户登录

在 login_user 函数中，用户信息并不直接从字典中检索，而是从数据库中查询。这是因为在实际应用中，用户信息往往需要持久化存储。因此，在用户成功注册后，其信息被上传到数据库。虽然字典在登录过程中并不直接使用，但它在注册过程中为信息的暂存提供了便利。

字典的这种临时存储功能对于开发过程中的调试和测试尤为重要。开发者可以迅速验证用户信息是否被正确存储和处理，而无需频繁访问数据库。此外，使用字典可以轻松实现用户信息的批量操作，例如：

```
# 批量更新用户状态
for user in user_data:
    user_data[user]['is_active'] = True
```

这显示了字典在管理复杂数据时的便利性。

3.4 总结

通过使用字典来存储用户信息，我们能够简化代码逻辑，提高信息管理的灵活性。这种方法特别适合于处理小规模的数据存储，在用户注册阶段提供了有效的信息验证和存储方式。然而，在生产环境中，持久化存储（如数据库）仍然是必需的，以确保数据的安全和完整性。

总体而言，字典的使用展示了 Python 中数据结构的强大和灵活性，使得开发者能够快速实现复杂的数据管理需求。通过合理地运用字典，开发者可以创建出更为高效和易于维护的代码。字典不仅能存储简单的数据，还能处理复杂的数据关系，为应用程序提供了强大的数据管理能力。

4 数据库的使用

在本节中，我们将分析如何在用户注册和登录系统中使用数据库，以持久化存储用户信息。

4.1 数据库连接

在代码中，我们使用了 `mysql.connector` 库来连接到 MySQL 数据库。通过配置数据库连接信息（如用户名、密码、主机和数据库名），可以建立与数据库的连接：

```
# 数据库配置
DB_CONFIG = {
    'user': 'root',
    'password': 'root',
    'host': 'localhost',
    'database': 'test_db'
}
conn = mysql.connector.connect(**DB_CONFIG)
```

这里，`DB_CONFIG` 是一个字典，包含连接所需的所有信息。

4.2 创建用户表

在用户注册过程中，首先检查用户表是否存在。如果表不存在，则创建一个新的用户表以存储用户信息。相关的 SQL 语句如下：

```
create_table_query = '''
CREATE TABLE IF NOT EXISTS users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) NOT NULL UNIQUE,
    email VARCHAR(255) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL
)
'''
cursor.execute(create_table_query)
```

此 SQL 查询创建一个名为 users 的表，其中包括用户的唯一标识符 (ID)、用户名、电子邮件和哈希密码。

4.3 插入用户信息

当用户注册成功后，使用以下 SQL 语句将用户信息插入数据库：

```
insert_query = '''
INSERT INTO users (username, email, password)
VALUES (%s, %s, %s)
'''
cursor.execute(insert_query, (username, email, user_data[username]['password
    ']))
```

使用 execute 方法，将用户名、电子邮件和哈希密码插入到 users 表中。通过 conn.commit() 提交更改以确保数据持久化。

4.4 查询用户信息

在用户登录过程中，系统根据输入的用户名从数据库中查询相应的密码，以验证用户身份。相关 SQL 语句如下：

```
query = "SELECT password FROM users WHERE username = %s"
cursor.execute(query, (username,))
```

如果查询成功，则可以对比输入的密码和数据库中存储的哈希密码。

4.5 释放数据库资源

在注册或登录操作结束后，无论是否发生异常，`finally` 块都会确保数据库游标和连接被正确关闭：

```
finally:
    cursor.close()
    conn.close()
```

`cursor.close()` 关闭游标，释放查询相关的资源；`conn.close()` 关闭数据库连接，避免占用过多的数据库资源。这种方式保证了在数据库操作后资源得到正确释放，即使操作过程中发生了错误。使用 `finally` 块能够确保在任何情况下，游标和连接都会被关闭，避免资源泄漏。

4.6 总结

通过使用数据库，我们能够有效地持久化存储用户信息，确保数据的安全性和完整性。虽然字典在注册过程中提供了临时存储，但数据库的使用是实现持久化和高效查询的关键。使用 SQL 语言与数据库交互，简化了数据管理，使得系统在用户注册和登录过程中能够可靠地处理用户信息。

5 程序测试

在本节中，我们将对用户注册和登录系统进行测试，以确保功能的正确性和稳定性。

5.1 注册测试

注册测试的目的是验证用户信息的输入、格式检查和存储功能是否正常。

5.1.1 测试内容

- **有效输入测试**：输入符合要求的用户名、电子邮件和密码，检查系统是否能够成功注册用户，并将信息存入数据库。例如：

  输入用户名：`testuser`
  输入电子邮件：`testuser@example.com`

输入密码：Password123!

- **用户名重复测试**：尝试注册已存在的用户名，系统应提示用户名已存在。例如：

  输入用户名：testuser

  系统应返回：用户名已存在，请重新输入。

- **电子邮件格式测试**：输入无效的电子邮件格式，系统应提示电子邮件格式不正确。例如：

  输入电子邮件：testuser@com

  系统应返回：电子邮件格式不正确，请重新输入。

- **密码复杂度测试**：输入不符合复杂度要求的密码，系统应提示密码不符合要求。例如：

  输入密码：12345678

  系统应返回：密码不符合复杂度要求，请重新输入。

5.1.2 测试结果

如图二所示，所有测试样例按预期返回。如图三所示，符合要求的用户名、电子邮件和密码被正确存入数据库，且密码已经被哈希加密。

5.2 登录测试

登录测试的目的是验证用户身份验证的正确性以及错误处理的有效性。

图 2: 测试代码

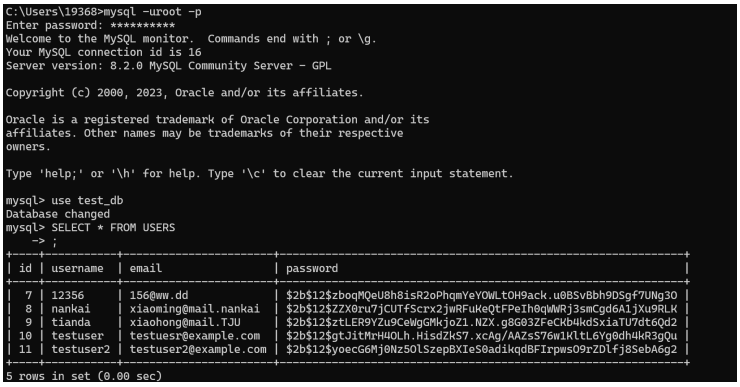

图 3: 数据库查询结果

5.2.1 测试内容

- **有效登录测试**：使用正确的用户名和密码登录，系统应返回登录成功的消息。例如：

  输入用户名：`testuser`
  输入密码：`Password123!`

  系统应返回：登录成功!

- **密码错误测试**：输入正确的用户名但错误的密码，系统应提示密码错

误。例如：

输入用户名：testuser
输入密码：WrongPassword!

系统应返回：密码错误。

- **用户不存在测试**：输入不存在的用户名，系统应提示用户名不存在。例如：

输入用户名：unknownuser
输入密码：Password123!

系统应返回：用户名不存在。

5.2.2 测试结果

图 4: 测试代码

如图四所示，对于所有样例系统返回正确结果。

5.3 总结

通过上述注册和登录测试，可以确保系统能够有效地处理用户输入，验证信息的正确性，并在不同情况下提供适当的反馈。这些测试有助于提高系统的稳定性和用户体验，确保用户注册和登录功能的可靠性。

6 遇到的困难及解决方案

6.1 用户输入验证

困难：用户输入的电子邮件或密码格式可能不符合要求，导致验证失败。

解决方案：在输入时，使用正则表达式进行格式验证，并在输入不符合要求时提供清晰的提示信息。对于电子邮件，正则表达式检查其结构是否包含“@”和有效的域名；对于密码，验证其复杂度（大写字母、小写字母、数字和特殊字符的组合）。使用循环结构确保用户反复输入直到格式正确为止。

6.2 使用正则表达式时的问题

困难：正则表达式可能过于严格或不准确，导致合法的输入被拒绝，或不符合标准的输入被接受。

解决方案：编写正则表达式时要综合考虑各种合法输入的形式，并确保表达式足够灵活。同时，在测试阶段应针对不同的输入情况进行全面测试，调整表达式以适应各种常见的格式。例如，确保支持常见的电子邮件格式，如包含子域名或短顶级域名的电子邮件。

6.3 密码哈希化和验证问题

困难：在处理密码时，可能会遇到哈希化或验证失败的问题，例如用户输入的密码与存储在数据库中的哈希值不匹配。

解决方案：确保使用相同的哈希算法（如 bcrypt）对密码进行加密和验证。哈希化时，必须将用户输入的明文密码加密后与数据库中的哈希值进行比较。在验证时，确保使用一致的编码方式（如 UTF-8）进行哈希处理，避免编码不一致导致的错误。

6.4 字典无法实现长期存储的问题

困难：在程序运行时，字典能有效地存储用户信息，但程序结束后，所有存储在字典中的数据将丢失，无法实现持久化存储。

解决方案：使用字典仅作为程序运行期间的数据缓存，真正的用户数据应存储在持久化的数据库中（如 MySQL）。在用户注册和登录的过程中，将

数据从字典中提取并通过 SQL 查询存入数据库，确保数据在系统重新启动后仍然可用。

6.5 数据库连接问题

困难：在连接到 MySQL 数据库时，可能会遇到连接失败或认证错误。

解决方案：确保数据库配置正确，包括用户名、密码、主机和数据库名。可以在数据库客户端中测试这些凭据，以验证其有效性。此外，确保数据库服务正在运行，并且允许外部连接。如果数据库连接频繁失败，可以设置数据库连接的超时时间，并在失败后进行重试。

6.6 SQL 语法错误

困难：在执行 SQL 查询时，可能会遇到语法错误，导致代码崩溃或返回错误信息。

解决方案：仔细检查 SQL 语句的语法，确保字段名和数据类型与数据库表的定义一致。可以通过数据库客户端执行相同的查询，以调试和确认语法的正确性。对于较复杂的查询，可以使用参数化查询避免 SQL 注入风险，同时减少手动拼接 SQL 语句时出错的几率。

7 附录：源代码

```
    import mysql.connector
from mysql.connector import errorcode
import bcrypt
import re

# 数据库配置
DB_CONFIG = {
    'user': 'root',
    'password': 'zxc7777777',
    'host': 'localhost',
    'database': 'test_db'
}

# 存储用户信息的字典
```

```
15 user_data = {}
16
17 def validate_email(email):
18     """ 验证电子邮件格式 """
19     email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
20     return re.match(email_pattern, email) is not None
21
22 def validate_password(password):
23     """ 验证密码复杂度 """
24     password_pattern = r'^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za
           -z\d@$!%*?&;:"#_]{8,}$'
25     return re.match(password_pattern, password) is not None
26
27 def register_user():
28     """ 注册用户的函数 """
29     while True:
30         username = input("请输入用户名: ")
31         # 检查用户名是否已存在
32         if username in user_data:
33             print("用户名已存在，请重新输入。")
34         else:
35             break
36
37     email = input("请输入电子邮件: ")
38     while not validate_email(email):
39         print("电子邮件格式不正确，请重新输入。")
40         email = input("请输入电子邮件: ")
41
42     password = input("请输入密码: ")
43     while not validate_password(password):
44         print("密码不符合复杂度要求，请重新输入。")
45         password = input("请输入密码: ")
46
47     # 将密码哈希化
48     hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt
           ())
49
50     # 存储用户信息到字典
51     user_data[username] = {
```

```
52          'email': email,
53          'password': hashed_password.decode('utf-8')
54      }
55
56      # 将用户信息存入 MySQL 数据库
57      try:
58          conn = mysql.connector.connect(**DB_CONFIG)
59          cursor = conn.cursor()
60
61          # 创建用户表 (如果尚不存在)
62          create_table_query = '''
63          CREATE TABLE IF NOT EXISTS users (
64              id INT AUTO_INCREMENT PRIMARY KEY,
65              username VARCHAR(255) NOT NULL UNIQUE,
66              email VARCHAR(255) NOT NULL UNIQUE,
67              password VARCHAR(255) NOT NULL
68          )
69          '''
70          cursor.execute(create_table_query)
71
72          # 插入用户信息
73          insert_query = '''
74          INSERT INTO users (username, email, password)
75          VALUES (%s, %s, %s)
76          '''
77          cursor.execute(insert_query, (username, email, user_data[username]['
                password']))
78
79          # 提交更改
80          conn.commit()
81          print("用户注册成功，信息已存入数据库。")
82      except mysql.connector.Error as err:
83          if err.errno == errorcode.ER_DUP_ENTRY:
84              print("用户名或电子邮件已存在。")
85          else:
86              print(f"发生错误: {err}")
87      finally:
88          cursor.close()
89          conn.close()
```

```
90
91  def login_user():
92      """ 登录用户的函数 """
93      username = input("请输入用户名: ")
94      password = input("请输入密码: ")
95
96      # 连接到 MySQL 数据库
97      try:
98          conn = mysql.connector.connect(**DB_CONFIG)
99          cursor = conn.cursor()
100
101         # 查询用户信息
102         query = "SELECT password FROM users WHERE username = %s"
103         cursor.execute(query, (username,))
104         result = cursor.fetchone()
105
106         # 检查用户是否存在并验证密码
107         if result:
108             stored_password = result[0].encode('utf-8')  # 从数据库中获取的
                    密码
109             if bcrypt.checkpw(password.encode('utf-8'), stored_password):
110                 print("登录成功! ")
111             else:
112                 print("密码错误。")
113         else:
114             print("用户名不存在。")
115     except mysql.connector.Error as err:
116         print(f"发生错误: {err}")
117     finally:
118         cursor.close()
119         conn.close()
120
121 # 示例调用
122 if __name__ == "__main__":
123     while True:
124         action = input("请选择操作: 1. 注册用户 2. 登录用户 3. 退出\n")
125         if action == '1':
126             register_user()
127         elif action == '2':
```

```
128             login_user()
129         elif action == '3':
130             print("退出程序。")
131             break
132         else:
133             print("无效的选择，请重新输入。")
```