2025/5/20

1.接口测试中区分前端和后端问题的具体方法包括以下几种：

1. **检查接口请求和响应**：通过抓包工具（如Fiddler、Charles）或浏览器自带的开发者工具（F12），检查接口请求和响应的数据。如果请求的数据与接口文档不符，通常是前端问题；如果返回的数据与预期不符，则是后端问题12。
2. **查看状态码**：接口响应的状态码可以帮助判断问题来源。通常，状态码以4开头的错误是前端问题，以5开头的错误是后端问题，状态码200表示请求成功1。
3. **检查数据一致性**：如果接口返回的数据正常，但页面显示不一致或无法显示，通常是后端问题；如果前端显示的数据与后台返回的数据格式不一致，也是后端问题1。
4. **日志和错误信息**：查看后台日志和错误信息，确认是否发送了正确的通知或数据。如果后台没有发送正确的通知或数据，通常是后端问题；如果前端没有正确处理接收到的数据，则是前端问题13。
5. **代码审查和调试**：通过查看开发代码、日志、浏览器控制台输出等，可以进一步定位问题是出在前端还是后端。这需要测试人员对代码有一定的了解1。

接口测试的基本流程包括以下几个步骤：

1. **需求沟通与任务接收**：明确测试内容和截止时间，与相关开发人员沟通接口业务逻辑和需求细节。
2. **测试准备阶段**：查看接口文档，确认测试环境，编写测试用例，选择合适的测试工具。
3. **测试执行阶段**：使用选定工具进行接口测试，关注请求发送、响应接收和结果验证。查询后台日志和数据库，验证数据的正确性。
4. **问题管理与沟通**：及时提报bug并跟踪修复进度，确认问题解决情况。
5. **测试报告撰写与通知**：撰写测试报告，包括测试用例数量、bug数量、测试结论等信息

2025/6/4

一、Python

Python 是一种解释型、面向对象的语言

Python是一种**解释型**语言。这意味着Python代码在运行时会被一个解释器逐行解释执行，而不是像编译型语言那样，在程序运行之前先编译成机器码。解释型语言的这种特性使得它们通常更适合快速开发和原型设计，因为开发者可以立即看到代码的运行结果，而不必等待整个程序编译完成。然而，这也意味着解释型语言可能在执行效率上略低于编译型语言。

1、数据类型

类型	描述	说明
数字`Number`	整数(int)、浮点数(float)、复数(complex)、布尔(bool)	
字符串`String`	描述文本的一种数据类型	由任意数量的字符组成
列表`List`	**有序**的**可变**序列	`eg:[1, 'a']`
元组`Tuple`	**有序**的**不可变**序列	`eg:(1, 'a')`
集合`Set`	**无序**的**不重复**集合	`eg:{1, 2, 3}`或`set([1,2,3])`
字典`Dictionary`	**无序 ** **Key-Value **集合	`eg:{'name': 'Alcie'}`

- **类型**
  - **不可变数据类型：**`Number、String、Tuple`
  - **可变数据类型：**`List、Set、Dictionary`
  - **其他类型：**
    - `NoneType:`表示空值，唯一实例为`None`
    - 字节 `bytes` 与字节数组 `bytearray`：用于二进制数据处理，`eg:b"python"`

1.1 Number

数字数据类型，如果改变Number类型的值、将重新分配内存空间。

- **int** 整型，是正或负整数，不带小数点。Python3 整型是没有限制大小
- **float** 浮点型，由整数部分与小数部分组成，浮点类型不精确存储，可用科学计数法表示（2.5e2 = 2.5 x 102 = 250）
- **bool** 布尔型，true的整型值是1，false的整型值是0。a = True + 1的值为2
- **complex**（复数） 复数类型，复数由实部（real）和虚部（imag）构成

1.2 String

字符串类型可以使用单引号、双引号、三个单引号和三个双引号，其中三引号可以多行定义字符串，Python 不支持单字符类型，单字符也在Python也是作为一个字符串使用

- **字符串切片** 字符串索引index取值范围：`-len()`到`len()-1`，越界会报错
- **字符串替换** `s = """hello""" q = s.replace('l', 'p') => q = "heppo"`
- **字符串查找** `find()、index()、rfind()、rindex()`
  - **find()** `str.find(sub[, start[, end]])` ：返回子字符串 `sub` 在原字符串 `str` 中**第一次出现**的起始索引；若未找到，返回-1
  - **index()** `string.index(value[, start[, end]])` 未找到，返回异常
  - `rfind()、rindex()` 返回字符串**最后一次出现**的索引
- **字符串转大小写** `upper()、lower()、swapcase()、capitalize()、istitle()、isupper()、islower()`
  - `str.swapcase()`： 大小写反转
  - `str.capitalize()`：第一个单词首字母大写
  - `str.title()` ：每个单词首字母大写
- **字符串去空格**
  - `string.strip([characters])` ：要删除的前导/尾随字符的一个或多个字符，默认空格、`\n、\t`
  - `string.lstrip([characters])` ：去掉字符串左边的空格或指定字符
  - `string.rstrip([characters])` ：去掉字符串右边的空格或指定字符
- **字符串格式化** 推荐使用`format`格式化字符串
- **字符串连接与分割**

  使用 + 连接字符串，每次操作会重新计算、开辟、释放内存，效率很低，所以推荐使用`join`

- **分割** `split()` eg: `s = 'a-b-c' s.split('-') => ['a', 'b', 'c']`

1.3 List

元素可以重复、可以修改、无数据类型限制、有序的元素集 中括号[ ]

- **定义** `list1 = ['a', 'b']` 或者 `list1 = list('ab')` # `['a', 'b']`
- **搜索** `list1.index('b')` # 1
- **添加** `list1.append('c')` # ['a', 'b', 'c']
- **删除元素** `list1.remove('b')` # ['a', 'c']
- **删除元素** `del list1[1]` # ['a', 'c']
- **删除列表** `del list1` # 删除后不可再用`list1`
- **列表的截取（切片）：** `str[start: end: step]`
  - 正数：左闭右开；负数：左闭右闭
  - ```
    list1 = list('engli') #['e', 'n', 'g', 'l', 'i']
    # 切片赋值
    list1[5:] = 'hs' #['e', 'n', 'g', 'l', 'i', 'h', 's']
    # 切片替换
    list1[5:] = list('sh!') #['e', 'n', 'g', 'l', 'i', 's', 'h', '!']
    # 切片删除
    list1[-1:] = '' # ['e', 'n', 'g', 'l', 'i', 's', 'h']
    # 负索引操作
    list1[-1:-1] = [1,2,3]  # ['e', 'n', 'g', 'l', 'i', 's', 1, 2, 3, 'h']
    ```

    > **注：**`[-1:-1]`解释为**空切片**，在索引前插入
    >
    > 尾部追加：`append()`(用于单元素)、`extend()`（用于多元素）、+、`lsit1[len(list1):]`
- **列表遍历**
  - `for i in list1: print(i)` 或者 `for i in range(len(list1)): print(list1[i])`
- **len()**：返回集合中元素个数 `eg:len(list1)`
- **max、min函数**：返回集合中的最值 `max(list1)`
- **sorted 排序：**`sorted(iterable[, key=None[, reverse=False]])`
  - 不修改原对象
  - `Flase`默认升序，`True`表示降序
  - `list1.sort()`：仅适用于列表，直接修改且无返回值

1.4 Tuple
    ```

**元素可以重复、不能修改、无数据类型限制、有序的集合 ** 小括号（ ）

- **元组运算符**
  - **乘号*、加号+**

```
t = ('a', 'b')
print((t,) * 2) # (('a', 'b'), ('a', 'b'))
print(t * 2) # ('a', 'b', 'a', 'b')
```

  - **将列表转换为元组** `tuple(list)`
  - **max min len**

1.5 Set

(集合、容器) 无序不重复元素

使用大括号 {} 或 set() 函数创建（创建空set集合不能用 {}， {} 是创建空字典）

成员检测、消除重复元素 支持像并集、交集、差集、对称差分等**数学运算** set集合**不能被切片也不能被索引**，除了做集合运算之外，集合元素可以被添加还有删除

- **定义** `set1 = {'a', 'b', 'a', 'b'}` # {'b', 'a'}
- **添加单个元素** `set1.add('e')` #{'a', 'e', 'b'}
- **添加元素**（参数可以是列表、元组、字典等）

```
set1.update([1,2])
set1.update((1,2))
set1.update({1,2})
set1.update([1,2],[3,5])
```

- **删除指定元素** `set1.discard('e')` 或者`set1.remove('e')`

> **区别：**`remove`元素不存在会报错，`discard`元素不存在不会报错

- **随机删除一个元素** `set1.pop()` `set1`不为空
- **去重** `list1 = ['a', 'b', 'a']` `set2 = set(list1)` # {'a', 'b'}
- **长度len()** `eg:len(set1)`
- **清空集合** `set1.clear()`
- **判断元素是否存在** `'a' in set1`
- **集合之间的运算**

- **差集 - 或者 difference()**：代表前者中包含后者中不包含的元素
- **并集 | 或者 union()**：代表两者中全部元素聚在一起去重后的结果
- **交集 & 或者 intersection()**：两者中都包含的元素
- **对称差集 ^ 或者 symmetric_difference()**：不同时包含于两个集合中的元素
- ```
  set1 = {'a', 'b'}
  set2 = {'a', 'e'}
  print(set1-set2) # {'b'}
  print(set1 | set2) # {'b', 'e', 'a'}
  print(set1.union(set2))
  print(set1 & set2) # {'a'}
  print(set1 ^ set2) # {'b', 'e'}
  ```

1.6字典

字典是一种元素为**key-value**映射类型（即**键值对映射**，其它变成语言中的Map集合）。 字典的**key（键）必须为不可变**类型，且**不能重复**，如果键重复、则最后的一个键值对会替换前面的键值对。值可以取任何数据类型。键必须是不可变的，如字符串，数字或元组等，用列表就不行。 字典是**不排序**的，所以不能像列表那样切片。如果访问字典中不存在的键，将导致 `KeyError` 出错信息。 **创建空字典使用 {}**

- **字典的定义**

  ```
  dic = {} # 创建空字典
  stu = {'stu1': 'tree', 'stu2': 'flower'}
  print(stu['stu1'])  # tree
  print(stu.keys())  # dict_keys(['stu1', 'stu2'])
  print(stu.values())  # dict_values(['tree', 'flower'])
  print(len(stu))  # 2
  ```

- **字典的遍历** `dict.items()`

  ```
  for k, v in stu.items():
      print(k, v)
  ```

- **字典的操作**
  - **更新/修改字典** `stu['stu3'] = 'wind'`
  - **删除字典元素** `del stu['stu1']`
  - **删除字典** `del stu` 不可再访问`stu`
  - **计算元素（键）总数** `len()`
  ```

- **转换成字符串** `str(dict)`
- **判断数据类型** `type(elem)`

2.库

分为三类：

1. **测试框架与工具库**（直接支撑测试工作）
   - `pytest`：主力测试框架，常用于编写自动化测试用例，配合`pytest-html`生成报告
   - `unittest`：在部分遗留项目中维护单元测试
   - `selenium`：`Web UI`自动化测试，配合`WebDriverManager`管理浏览器驱动
   - `requests`：接口测试，验证`RESTful API`功能及性能
2. **测试辅助工具库**（提升测试效率）
   - `mock`/`unittest.mock`：隔离被测代码的依赖项（如模拟数据库连接）
   - `pymysql`：数据库操作库，用于测试数据准备与结果验证
   - `allure-pytest`：生成可视化测试报告，增强缺陷定位能力
   - `faker`：生成仿真测试数据（如用户注册信息）
3. **通用工具库**（支持测试脚本开发）
   - `pandas`：分析测试日志或性能数据（如解析`JMeter`输出）
   - `openpyxl`：读写`Excel`测试用例
   - `logging`：管理测试脚本的日志记录
   - `json`/`re`：处理`API`响应及数据清洗

二、操作系统

1.进程和线程

1.1线程和进程的定义

- 进程（`Process`） 进程是计算机中运行的一个程序的实例。它是操作系统分配资源和调度的基本单位，每个进程都有自己的内存空间、文件描述符和其他资源。
- 特点：
  - 每个进程都有独立的地址空间。
  - 进程之间的通信需要通过进程间通信（`IPC`）机制，例如管道、信号量或消息队列。
  - 进程是重量级的，创建和销毁成本较高。
- 线程（`Thread`）

线程是进程中的一个执行单元，是CPU调度的基本单位。一个进程可以包含多个线程，它们共享进程的资源。

- 特点：
  - 线程之间共享同一个进程的内存空间和资源（例如堆和全局变量）
  - 线程的切换开销比进程小，效率更高
  - 一个线程的崩溃可能导致整个进程的崩溃

1.2 线程和进程的区别

比较维度	线程	进程
定义	进程中的执行单元	程序的实例
地址空间	共享进程的地址空间	独立的地址空间
资源共享	共享进程的资源（如内存、文件句柄等）	进程间不共享资源，需要IPC机制
创建开销	较低，创建速度快	较高，需要分配独立的内存和资源
通信方式	通过共享内存直接通信	需要使用IPC机制（如管道、队列）
独立性	一个线程崩溃可能影响整个进程	一个进程崩溃通常不会影响其他进程
上下文切换	开销小，速度快	开销大，速度慢

1.3 线程和进程的应用场景

- 使用进程的场景：
  - 需要更高的独立性和隔离性，例如运行多个服务
  - 程序之间的相互影响需要最小化
  - 系统级任务，例如启动数据库服务或运行一个单独的守护程序
- 使用线程的场景
  - 同一任务中需要并发执行，例如多线程处理请求
  - 性能要求高且需要频繁切换上下文
  - 程序中需要共享大量数据。

三、数据结构

1.数组

1.1 数组的特点

> **1.在内存中，数组是一块连续的区域**
>
> **2.数组需要预留空间**
>
> 在使用前需要提前申请所占内存的大小，这样不知道需要多大的空间，就预先申请可能会浪费内存空间，即数组空间利用率低 ps：数组的空间在编译阶段就需要进行确定，所以需要提前给出数组空间的大小（在运行阶段是不允许改变的）
>
> **3.在数组起始位置处，插入数据和删除数据效率低**
>
> 插入数据时，待插入位置的的元素和它后面的所有元素都需要向后搬移 删除数据时，待删除位置后面的所有元素都需要向前搬移
>
> **4.随机访问效率很高，时间复杂度可以达到O(1)**
>
> 因为数组的内存是连续的，想要访问那个元素，直接从数组的首地址处向后偏移就可以访问到了

> **5.数组开辟的空间，在不够使用的时候需要扩容，扩容的话，就会涉及到需要把旧数组中的所有元素向新数组中搬移**
>
> **6.数组的空间是从栈分配的**

1.2 数组的优点

> 随机访问性强，查找速度快，时间复杂度为O(1)

1.3 数组的缺点

> 1.头插和头删的效率低，时间复杂度为O(N)
>
> 2.空间利用率不高
>
> 3.内存空间要求高，必须有足够的连续的内存空间
>
> 4.数组空间的大小固定，不能动态拓展

2.链表

2.1 链表的特点

> **1.在内存中，元素的空间可以在任意地方，空间是分散的，不需要连续**
>
> **2.链表中的元素都会两个属性，一个是元素的值，另一个是指针，此指针标记了下一个元素的地址**
>
> 每一个数据都会保存下一个数据的内存的地址，通过此地址可以找到下一个数据
>
> **3.查找数据时效率低,时间复杂度为O(N)**
>
> 因为链表的空间是分散的，所以不具有随机访问性，如要需要访问某个位置的数据，需要从第一个数据开始找起，依次往后遍历，直到找到待查询的位置，故可能在查找某个元素时，时间复杂度达到O(N)
>
> **4.空间不需要提前指定大小，是动态申请的，根据需求动态的申请和删除内存空间，扩展方便，故空间的利用率较高**
>
> **5.任意位置插入元素和删除元素效率较高，时间复杂度为O(1)**
>
> **6.链表的空间是从堆中分配的**

2.2 链表的优点

> 1.任意位置插入元素和删除元素的速度快，时间复杂度为O(1)
>
> 2.内存利用率高，不会浪费内存
>
> 3.链表的空间大小不固定，可以动态拓展

2.3 链表的缺点

> 随机访问效率低，时间复杂度为0(N)

3.栈

概念与结构

栈：一种特殊的**线性表**，其只允许在固定的一端进行插入和删除元素操作。进行数据插入和删除操作的一端称为**栈顶**，另一端称为**栈底**。栈中的数据元素遵守**后进先出**LIFO（Last In First Out）的原则。

- **压栈**：栈的插入操作叫做进栈/压栈/入栈，入数据在栈顶。
- **出栈**：栈的删除操作叫做出栈。出数据也在栈顶。

4.队列

概念与结构

概念：只允许在一端进行插入数据操作，在另一端进行删除数据操作的特殊**线性表**，队列具有**先进先出** FIFO(First In First Out)

- **入队列**：进行插入操作的一端称为队尾
- **出队列**：进行删除操作的一端称为队头

四、计网

1.网络分层架构

OSI/RM（理论上的标准）	TCP/IP 事实上的标准	
应用层	应用层（应用程序间）	HTTP
表示层		
会话层		
传输层	传输层（进程间）	TCP、UDP
网络层	网络层（主机间）	IP
数据链路层	链路层（设备间）	
物理层		

2.TCP协议

2.1、TCP协议：

位于**传输层**， 提供**可靠**的字节流服务。所谓的字节流服务（Byte Stream Service） 是指， 为了方便传输， 将大块数据分割成以**报文段**（segment） 为单位的数据包进行管理。 而**可靠的传输服务是指， 能够把数据准确可靠地传给对方。**即TCP 协议为了更容易传送大数据才把数据分割， 而且 TCP 协议能够确认数据最终是否送达到对方。所以，TCP连接相当于两根管道（一个用于服务器到客户端，一个用于客户端到服务器），管道里面数据传输是通过**字节码**传输，传输是**有序**的，每个字节都是一个一个来传输。

(1)、三次握手

握手过程中使用了 TCP 的标志（flag） —— SYN（synchronize） 和ACK（acknowledgement） 。

- **第一次**握手：建立连接时，客户端A发送SYN包（SYN=j）到服务器B，并进入SYN_SEND状态，等待服务器B确认。
- **第二次**握手：服务器B收到SYN包，必须确认客户A的SYN（ACK=j+1），同时自己也发送一个SYN包（SYN=k），即SYN+ACK包，此时服务器B进入SYN_RECV状态。
- **第三次**握手：客户端A收到服务器B的SYN＋ACK包，向服务器B发送确认包ACK（ACK=k+1），此包发送完毕，完成三次握手。

(2)、四次挥手

由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这个原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。先进行关闭的一方将执行主动关闭，而另一方被动关闭。

- 客户端A发送一个FIN，用来关闭客户A到服务器B的数据传送。
- 服务器B收到这个FIN，它发回一个ACK，确认序号为收到的序号加1。
- 服务器B关闭与客户端A的连接，发送一个FIN给客户端A。
- 客户端A发回ACK报文确认，并将确认序号设置为收到序号加1。

三次握手和四次挥手：在TCP连接中，服务器端的SYN和ACK向客户端发送是一次性发送的，而在断开连接的过程中， B端向A 端发送的ACK和FIN是分两次发送的。因为在B端接收到A端的FIN后， B端可能还有数据要传输，所以先发送ACK，等B端处理完自己的事情后就可以发送FIN断开连接了。

(3)、深入理解TCP连接：

由于TCP是全双工的，因此在每一个方向都必须单独关闭。

这原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个FIN只意味着这个方向上没有数据流动，一个TCP连接在接收到一个FIN后仍能发送数据。 首先进行关 闭的一方将执行主动关闭，而另一方执行被动关闭。 TCP协议的连接是全双工连接，一个TCP连接存在双向的读写通道。简单来说，是"先关读，再关写" ，总共需要4个阶段。

以客户机发起关闭连接为例：

1.服务器读通道关闭；

2.客户端写通道关闭；

3.客户端读通道关闭；

4.服务器写通道关闭。

关闭行为是在发起方数据发送完毕之后，给对方发出一个FIN（finish）数据段，直到接收到对方发送的FIN,且对方收到了接收确认的ACK之后，双方的数据通信完全结束，过程中每次都需要返回确认数据段ACK

3.UDP协议：

无连接协议，也称透明协议，也位于传输层。

4.两者区别：

1） TCP提供面向连接的传输，通信前要先建立连接（三次握手机制）；UDP提供无连接的传输，通信前不需要建立连接。

2） TCP提供可靠的传输（有序，无差错，不丢失，不重复）；UDP提供不可靠的传输。

3） TCP面向字节流的传输，因此它能将信息分割成组，并在接收端将其重组；UDP是面向数据报的传输，没有分组开销。

4） TCP提供拥塞控制和流量控制机制；UDP不提供拥塞控制和流量控制机制

5.长连接和短连接

HTTP的长连接和短连接本质上是TCP长连接和短连接。

HTTP属于应用层协议，在传输层使用TCP协议，在网络层使用IP协议。IP协议主要解决网络路由和寻址问题，TCP协议主要解决如何在IP层之上可靠地传递数据包，使得网络上接收端收到发送端所发出的所有包，并且顺序与发送顺序一致。TCP协议是可靠的、面向连接的。

在HTTP/1.0中默认使用短连接。也就是说，客户端和服务器每进行一次HTTP操作，就建立一次连接，任务结束就中断连接。当客户端浏览器访问的某个HTML或其他类型的Web页中包含有其他的Web资源（如JavaScript文件、图像文件、CSS文件等），每遇到这样一个Web资源，浏览器就会重新建立一个HTTP会话。

而从HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头加入这行代码

```
Connection:keep-alive
```

在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。实现长连接需要客户端和服务端都支持长连接。

HTTP协议的长连接和短连接，实质上是TCP协议的长连接和短连接