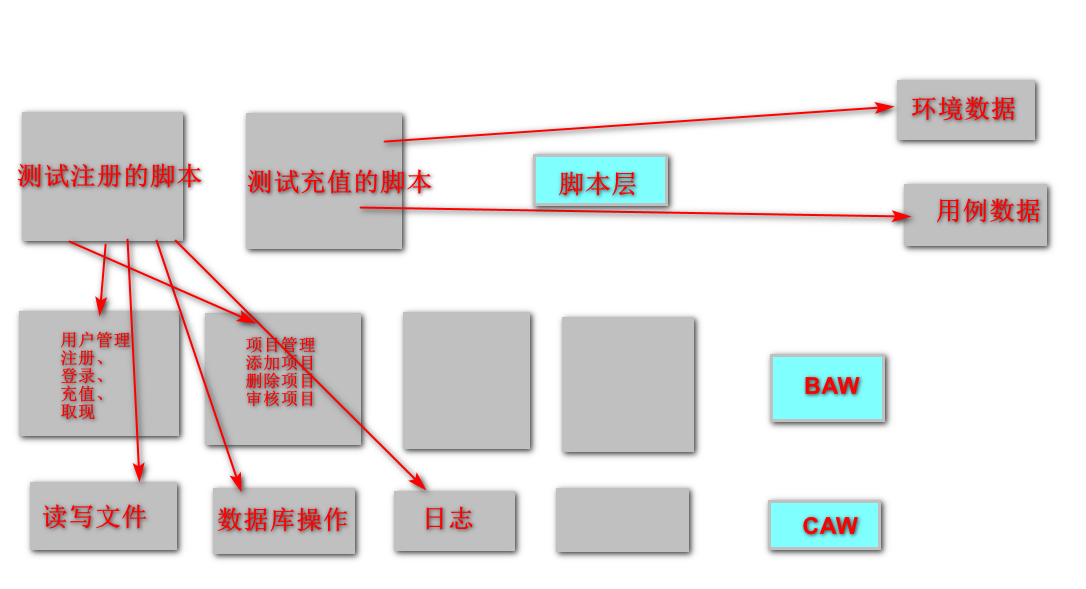
# Web/App接口自动化测试

1. 基于HTTP协议的接口
   1. 无状态，无连接
   2. get/post
      1. get参数是拼接到url中，不安全，?key=value&key=value
      2. post参数是放到body中的
   3. 通过cookie、session、token识别用户
2. 接口测试测的是什么？
   1. 单接口：通过构造不同的测试数据，检查对应场景的响应结果对不对。
   2. 接口之间有交互：登录->下单->支付； 下单
3. 接口测试与界面测试有什么差别？
   1. 接口测试不用界面
   2. 收益比较高
      1. 接口比界面稳定，代码的维护成本低。
      2. 接口比界面测试较早，修复缺陷的成本比较低。
      3. 接口测试自动率90%，界面自动化率60~70%
         1. 界面定位元素比较复杂，一些页面元素定位不到。
         2. 一些测试场景比较难构造。
         3. 界面变化比较快，修改老的代码+开发新功能的代码，导致人力不足。
   3. 接口代码执行比界面代码执行稳定性高。
4. 接口测试工具
   1. Postman（图形化工具）
   2. Jmeter（图形化工具，后面会讲）
   3. Python+Requests
   4. Java+HttpClient
   5. Ruby+。。。
5. 接口测试时，如果开发没有提供文档，怎么办？
   1. 界面已经有一部分功能，可以通过抓包的方式抓接口。
   2. 百格活动有界面，但是没有接口文档，
6. cookies
   1. 登录（用户名、密码）->服务器校验用户名密码
   2. 服务器把识别用户的关键信息，放到set-cookie中
   3. 下单接口，需要携带服务器返回的cookie信息。
   4. 评价接口，需要携带服务器返回的cookie信息。
7. pytest
   1. 测试前置和后置
      1. 测试前置：测试准备的工作，比如打开浏览器，比如准备测试数据，登录
      2. 测试后置：测试完成后，要做的清理的动作。比如关闭浏览器，比如恢复环境（自动化执行时在系统增加了什么数据，测试完成后要清理数据）
8. 接口自动化的演进
   1. 第一版：线性脚本
      1. 缺点：
         1. 执行失败时，中断，后面的执行不了。
         2. 统计不了有多少用例，无法统计用例通过率。
         3. 重复的代码比较多
         4. 硬编码测试数据
      2. 优点：实现起来简单
   2. 第二版：使用pytest框架来实现，用pytest组织测试用例
      1. 缺点：
         1. 重复的代码比较多
         2. 硬编码测试数据，不够灵活，修改起来不方便。
      2. 优点：相对于第一个版本，简单的封装
   3. 第三版：使用pytest-fixture来实现，数据驱动的模型。
      1. 缺点：
         1. 没有清理环境，部分脚本是半自动化的。比如注册成功，下次要手动清理环境。
      2. 优点：
         1. 测试数据跟测试用例是分离的
         2. 减少了重复代码
   4. 第四版：使用pytest-fixture来实现，数据驱动的模型，代码分层，数据分层。
      1. 优点：
         1. 测试数据跟测试用例是分离的
         2. 减少了重复代码
         3. 代码中自动清理环境
         4. 方便维护，比如接口增加参数/修改了返回值，更方便维护。
9. 自动化测试框架
   1. 线性脚本框架
      1. 录制回放
   2. 数据驱动自动化框架
      1. 测试数据放到csv、xml、yaml、数据库，同一个测试用例，调不同的数据执行。
   3. 关键字驱动自动化框架
      1. selenium属于典型的关键字驱动的框架
      2. 把功能封装成函数/方法，
         1. 打开浏览器(OpenBrowser)
         2. 最大化浏览器(Max...)
         3. 点击某个链接(Click)
   4. 模块化自动化框架
      1. 将系统按模块划分，管理自动化。
   5. 混合型
10. 数据分层
    1. 如下代码段，数据分几类？
       1. data = {**"mobilephone"**: 1801234567, **"pwd"**: 123456, **"regname"**: **"hello"**}
       2. expect = {**"status"**: 0, **"code"**: **"20109"**, **"data"**: **None**, **"msg"**: **"手机号码格式不正确"**}
       3. url = **"http://jy001:8081/futureloan/mvc/api/member/register"**
       4. r = [requests.post](http://requests.post)(url, data=data)
       5. **assert** real.json()[**'msg'**] == expect[**'msg'**]
    2. 用例数据（csv/yaml/xml）：测试数据、期望结果
    3. 接口地址（常量，定义到代码里）：futureloan/mvc/api/member/register
    4. 环境信息（变化，ini文件）：包括URL，http://jy001:8081；数据库信息（IP地址、端口号、用户名、密码）
11. 代码分层
    1. 租车系统（mysql+tomcat）、金融系统（mysql+tomcat）
    2. 自动化中可能有哪些内容？
       1. 用户管理模块的功能，包含注册、登录、删除用户
       2. 对数据库的操作（增删改查查）——两个项目通用
       3. 生成日志——两个项目通用
       4. 读写文件——两个项目通用
       5. 租车模块、客户管理、车辆管理
       6. 项目管理、审核
    3. CAW：业务无关的一层，Common Action Word；BAW：业务相关的一层，Business Action Word



* 1. 好处
     1. 方便维护，方便扩展
     2. 方便定位问题
     3. 减少手工的操作，提升自动化通过率。

1. 测试逻辑（测试步骤）：注册功能分三个测试逻辑。
   1. 注册成功
      1. post去发送注册的请求
      2. 检查结果（断言）：
         1. 1、检查接口的返回值是否正确
         2. 2、用户实际有没有注册成功（1、数据库中是否有注册的用户 2、是否能登录 3、调用查询接口能查到）
      3. 清理环境
         1. 删除注册的用户
   2. 注册失败
      1. post去发送注册的请求
      2. 检查结果（断言）：
         1. 检查接口的返回值是否正确
   3. 重复注册
      1. post去发送注册的请求（注册成功）
      2. post去发送注册的请求（重复注册）
      3. 检查结果（断言）：
         1. 检查接口的返回值是否正确
2. 自动化原则
   1. 脚本与脚本之间是独立的，没有依赖关系
   2. 脚本需要的环境，在执行前脚本中构造，测试完，清理环境。
3. 框架搭建的思路
   1. 根据分层创建好目录结构
   2. 编写环境文件（ini）
   3. 封装一个方法读环境文件
   4. 编写测试数据的文件（yaml）
   5. 封装一个方法读测试数据
   6. *对requests中的get、post的方法进行一层封装，BaseRequests*
      1. *1、增加异常处理*
      2. *2、增加日志打印（Log）*
      3. *3、创建一个session，确保能自动管理cookie*
   7. 封装业务层的注册方法register，
      1. 使用BaseRequests的实例发请求
   8. 实现注册的测试脚本 test\_register