CRM

1、CRM系统是什么

CRM系统，是Customer Relationship Management 的简称，其中文名称为“客户关系管理”。CRM系统既是一套管理制度，也是一套软件和技术。能够为企业提供全方位的管理视角，赋予企业完善的客户交流能力，最大化客户的收益率。它的目标是缩减销售周期和销售成本，增加收入，寻找扩展业务所需的新的市场和渠道，提高客户的价值、满意度、盈利性和忠实度，为企业创造更多的收益。

2. CRM系统的起源

CRM系统的起源，在于企业对客户长期管理观念的重视，这种观念认为客户是企业最重要的资产，具有忠诚度的客户，能够提升企业的竞争优势，最终提高企业的利润率。因此，20世纪90年代初，第一代传统安装部署型CRM诞生，以流程管控为中心，其作用主要是降低销售成本。而第二代CRM系统是基于PC端的SaaS型CRM，诞生于21世纪初，通过对客户关系全流程进行管理，实现内部资源优化。第三代移动CRM系统于21世纪10年代发起，融合云、移动、社交、大数据等前沿技术，实现了企业业务在移动端的拓展。第四代智能CRM的崛起，融合了数据挖掘和机器学习技术，能够智能识别重要客户、建议联系潜在客户最佳时间、协助企业进行科学战略制定等。

3.CRM系统的作用

1. **CRM整合客户、企业、员工资源、优化业务流程 。**CRM可以根据需要千变万化地、围绕某个方面去整合资源，并可同时从其它多个角度探寻事物的相关属性，优化业务流程。
2. **减少销售环节，降低销售成本。**企业员工通过系统所给出客户信息，全面地了解客户的情况，同时将自身所得到的客户信息添加进系统，这样会使销售渠道更为畅通，信息的中间传递环节减少，销售环节也相应地减少，销售费用、销售成本也随之降低。
3. **提升员工生产力，提高企业的销售收入。**CRM在现场销售/服务，销售/服务自动化、协同工作、客户关怀等方面，有效地提高了员工和企业的生产力，从而提高企业的销售收入。
4. **通过数据分析提升企业运行效率。**对每个客户的数据进行整合及智能化分析，向客户提供个性化的产品及良好的售后服务，最终达到提升企业运行效率的目的。

展览行业是一个无比重视用户数据的行业，无论是展商、观众还是潜在的用户数据量非常庞大，以简单的EXCEL表格根本无法处理如此庞大的数据。而CRM系统能够很好的整合数据，同时方便销售进行用户管理，信息共享变得实时而便捷。

单元测试框架

单元测试简介

单元测试负责对最小的软件设计单元（模块）进行验证，它使用软件设计文档中对模块的描述作为指南，对重要的程序分支进行测试以发现模块中的错误。由于软件模块并不是一个单独的程序，为了进行单元测试还必须编写大量的额外代码，从而无形中增加了开发人员的工作量，目前解决这一问题比较好的方法是使用测试框架。测试框架在需要构造大量测试用例时尤为有效，因为如果完全依靠手工的方式来构造和执行这些测试，花费的成本是十分高的，而测试框架则可以很好地解决这些问题。

单元测试框架简介

单元测试是软件测试的一种类型，是对程序中最小单元进行的测试。程序的最小单元可以是一个函数、一个类，也可以是函数的组合或类的组合。

单元测试是软件测试中最低级别的测试活动，与之相对应的更高级别的测试有模块测试、集成系统和系统测试等。单元测试一般由软件开发者而不是独立的测试工程师完成。另外，单元测试有一个隐含的性质，那就是单元测试属于自动化测试。

软件测试分为手工测试和自动化测试。自动化测试中才有框架的概念。自动化测试框架需要提供自动化测试用例编写、自动化测试用例执行、自动化测试报告生成等基础功能。有了测试框架，只需要完成和业务高度相关的测试用例设计和实现即可。

另外，框架会处理好复杂度与扩展性的问题。目前较流行的Python单元测试框架是UnitTest、Pytest和Nose。

UnitTest框架

Python语言中有很多单元测试框架，UnitTest框架作为标准Python语言中的一个模块，是其他框架的基础。

简介

UnitTest是Python标准库中自带的单元测试框架，它有时候也被称为PyUnit。就像JUnit是Java语言的标准单元测试框架一样，UnitTest（PyUnit）则是Python语言的标准单元测试框架。UnitTest可以组织执行测试用例，并且提供丰富的断言方法，可以判断测试用例是否通过，并最终生成测试结果。用UnitTest单元测试框架可以进行Web自动化测试。

由于UnitTest是Python标准库中自带的单元测试框架，因此安装完Python后就已存在，而无须再单独安装。

UnitTest的核心要素

UnitTest的核心要素有TestCase、TestSuite、TextTestRunner、TextTestResult和Fixture，共5个。

1. TestCase
    - 一个TestCase（测试用例）就是一个测试用例。
    - 一个测试用例就是一个完整的测试流程，包括测试前的环境准备（SetUp）、执行测试代码（run），以及测试后的环境还原（tearDown）。
    - 一个测试用例就是一个完整的测试单元，通过运行这个测试单元，可以对某一个问题进行验证，用户定义测试用例需要继承TestCase类。
    - 一个测试用例是在UnitTest中执行测试的最小单元。它通过UnitTest提供的assert方法，来验证一组特定的操作和输入所得到的具体响应。UnitTest提供了一个名称为TestCase的基础类（unittest.TestCase），可以用来创建测试用例。
2. TestSuite
    - 一个TestSuite（测试套件）是多个测试用例的集合，是针对被测程序对应的功能和模块所创建的一组测试。一个测试套件内的所有测试用例将一起执行。
    - TestSuite()是测试用例集合。
    - 通过addTest()方法可以手动把TestCase添加到TestSuite中，也可以通过TestLoader把TestCase自动加载到TestSuite（TestCases之间不存在先后顺序）中。
3. TextTestRunner
    - TextTestRunner（测试执行器）负责测试执行调度并且为用户生成测试结果。它是运行测试用例的驱动类，其中的run方法可以执行TestCase和TestSuite。
4. TextTestResult
    - TextTestResult（测试报告）用来展示所有执行用例成功或者失败状态的汇总结果、执行失败的测试步骤的预期结果与实际结果，以及整体运行状况和运行时间的汇总结果。
5. Fixture
    - 通过使用Fixture（测试夹具），可以定义在单个或多个测试执行之前的准备工作，以及测试执行之后的清理工作。

- 一个测试用例环境的搭建和销毁就是一个Fixture，通过覆盖TestCase的setUp()和tearDown()方法来实现。
- 如果在测试用例中需要访问数据库，那么就可以在setUp()中建立数据库连接并进行初始化，测试用例执行后需要还原环境。tearDown()的过程很重要，要为以后的TestCase留下一个干净的环境，例如在tearDown()中需要关闭数据库连接。

工作流程

UnitTest的整个流程如下：

1. 编写TestCase。
2. 把TestCase添加到TestSuite中。
3. 由TextTestRunner来执行TestSuite。
4. 将运行的结果保存在TextTestResult中。将整个过程集成在unittest.main模块中

UnitTest练习

创建基础待测方法

```
# mathfun.py

# 加法，返回a+b的值
def add(a,b):
    return a+b

# 减法，返回a-b的值
def minus(a,b):
    return a-b

# 乘法，返回a*b
def multi(a,b):
    return a*b

def divide(a,b):
    return a/b
```

设计测试用例

UnitTest框架下测试用例必须以test为开头，必须为小写

```
# test_mathfunc.py

import unittest
from mathfunc import *

"""
.asserEqual断言函数，判断a,b两个值是否相等
.asserNotEqual，判断a,b两个值是否不相等

将mathfunc.py的函数导入进test_mathfunc.py中

编写TestMathFunc()类调用mathfunc.py函数，通过unittest中.asserEqual断言函数，判断预设值与实际值是否相等
"""

class TestMathFunc(unittest.TestCase):
```

```
    """测试mathfunc.py"""

    def test_add(self):
        """测试加法add"""
        self.assertEqual(3,add(1,2))
        self.assertNotEqual(3,add(2,2))

    def test_minus(self):
        """测试减法minus"""
        self.assertEqual(1,minus(3,2))

    def test_multi(self):
        """测试乘法multi"""
        self.assertEqual(6,multi(2,3))

    def test_divide(self):
        """测试除法divide"""
        self.assertEqual(2,divide(6,3))
        self.assertEqual(2.5,divide(5,2))
```

封装登录

多次运用的函数可以封装起来，也避免了多次初始化driver不在一个浏览器中

```
import time

class Mylogin(object):
    def __init__(self,driver):
        self.driver = driver

    def login(self):
        driver.find_element_by_name('username').send_keys('17610832710')
        driver.find_element_by_name('password').send_keys('123456')
        driver.find_element_by_xpath('//button[@type="button"]').click()
        time.sleep(2)
```

测试套件（TestSuite）是多个测试用例的集合，是针对被测程序的对应的功能和模块创建的一组测试。

通过TestSuite()的addTest()方法手动把TestCase添加到TestSuite中，或通过TestLoader把TestCase自动加载到TestSuite中。

```
# test_suite.py
"""
TestSuite是多个测试用例的集合，是针对被测程序的对应的功能和模块创建的一组测试

通过TestSuite()的addTest()方法手动把TestCase添加到TestSuite中，或通过TestLoader把
TestCase自动加载到TestSuite中

"""

import unittest
from test_mathfunc import TestMathFunc

if __name__ == "__main__":
    suite = unittest.TestSuite()
    # addTest()添加单个TestCase
    # suite = addTest(TestMathFunc("test_add"))
```

```
    # 执行多条测试用例
    tests=
[TestMathFunc("test_add"),TestMathFunc("test_divide"),TestMathFunc("test_minus")
]
    suite.addTests(tests)
    runner = unittest.TextTestRunner()
    runner.run(suite)
```

测试结果

TextTestRunner测试执行器负责测试执行调度并生成测试结果给用户
可以将测试结果直接在控制台中输出，也可以将测试结果输出到外部文件中

想要很清楚地看到每条用户执行的详细信息，可以通过设置verbosity参数实现。verbosity默认值为1，可以设置为0和2

- 0（静默模式）：只能获得总的测试用例数和总的测试结果
- 1（默认模式）：非常类似于静默模式，只是在每个成功的用例前面有个"."，每个失败的用例前面有个E
- 2（详细模式）：测试结果会显示每个测试用例的所有相关信息，并且在命令行里加入不同的参数可以起到一样的效果

```
# test_suite.py

import unittest
from test_mathfunc import TestMathFunc

if __name__ == "__main__":
    suite = unittest.TestSuite()
    # addTest()添加单个TestCase
    # suite = addTest(TestMathFunc("test_add"))
    # 执行多条测试用例
    tests=
[TestMathFunc("test_add"),TestMathFunc("test_divide"),TestMathFunc("test_minus")
]
    suite.addTests(tests)
    runner = unittest.TextTestRunner(verbosity=2)
    runner.run(suite)

    # 保存测试结果
    with open("E://result.txt",'a') as f:
        runner = unittest.TextTestRunner(stream=f, verbosity=2)
        runner.run(suite)
```

测试初始化与还原

通过使用Fixture，可以定义测试执行之前的准备工作和测试执行之后的清理工作。

setUp()：执行用例的前置条件，如建立数据库连接；

tearDown()：执行完用例后，为了不影响下一次用例的执行，一般有一个数据还原的过程，tearDown()是执行用例的后置条件，如关闭数据库连接。

```
# 修改test_mathfunc.py

import unittest
```

```
from mathfunc import *

class TestMathFunc(unittest.TestCase):
    """测试mathfunc.py"""

    # 在每条测试用例执行之前准备好测试环境
    def setUp(self):
        print("do something before test!")

    def test_add(self):
        """测试加法add"""
        self.assertEqual(3,add(1,2))
        self.assertNotEqual(3,add(2,2))

    def test_minus(self):
        """测试减法minus"""
        self.assertEqual(1,minus(3,2))

    def test_multi(self):
        """测试乘法multi"""
        self.assertEqual(6,multi(2,3))

    def test_divide(self):
        """测试除法divide"""
        self.assertEqual(2,divide(6,3))
        self.assertEqual(2.5,divide(5,2))

    def tearDown(self):
        print("do something after test!")
```

```
from selenium import webdriver
import unittest
import time

class TestIndex(unittest.TestCase):
    # 初始化一个webdriver,打开被测网站，最大化屏幕，打印开始时间
    def setUp(self):
        self.driver = webdriver.Chrome()
        self.driver.get("http://101.133.169.100/yuns/index.php")
        self.driver.maximize_window()
        time.sleep(2)
        print("StartTime:" + time.strftime('%Y-%m-%d-%H-%M-
%S',time.localtime(time.time())))
```

```
from selenium import webdriver
import unittest
import os
import time

class TestIndex(unittest.TestCase):
    def setUp(self):
        ...
        ...
    def tearDown(self):
        # 图片路径赋值
        filedir = "D:/screenshot"
```

```
        # 如果路径下没有文件夹，则创建文件夹
        if not os.path.exiss(filedir):
            os.makedirs(os.path.join('D:/','screenshot'))
        # 打印结束日期
        print("endTime:"+time.strftime('%Y-%m-%d-%H-%M-
%S',time.localtime(time.time())))
        # 定义图片名称
        screen_name = filedir + time.strftime('%Y-%m-%d-%H-%M-%S',
time.localtime(time.time())) + ".png"
        # 按日期截图
        self.driver.get_screenshot_as_file(screen_name)
        # 退出浏览器，释放进程
        self.driver.quit()
```

使用setUpClass()与tearDownClass()方法

- setUpClass()：必须使用@classmethod 装饰器，初始化操作在所有case运行前只运行
- tearDownClass()：必须使用@classmethod装饰器，还原操作在所有case运行后只运行一次

```
# 修改test_mathfunc.py

import unittest
from mathfunc import *

class TestMathFunc(unittest.TestCase):
    """测试mathfunc.py"""

    # 在每条测试用例执行之前准备好测试环境
    @classmethod
    def setUp(self):
        print("do something before test!")

    def test_add(self):
        """测试加法add"""
        self.assertEqual(3,add(1,2))
        self.assertNotEqual(3,add(2,2))

    def test_minus(self):
        """测试减法minus"""
        self.assertEqual(1,minus(3,2))

    def test_multi(self):
        """测试乘法multi"""
        self.assertEqual(6,multi(2,3))

    def test_divide(self):
        """测试除法divide"""
        self.assertEqual(2,divide(6,3))
        self.assertEqual(2.5,divide(5,2))

    @classmethod
    def tearDown(self):
        print("do something after test!")
```

测试用例跳过(skip)

在执行测试用例时，有时候有些用例是不需要执行的，UnitTest提供了跳过用例的方法。

- @unittest.skip(reason)：强制跳过，不需要判断条件。reason参数是跳过原因的描述，必须填写。
- @unittest.skipIf(condition, reason)：condition为True时将跳过用例。
- @unittest.skipUnless(condition, reason)：当condition为False时将跳过用例。
- @unittest.expectedFailure：如果test失败了，这个test不计入失败的case数目。

```
# 修改test_mathfunc.py

import unittest
from mathfunc import *

class TestMathFunc(unittest.TestCase):
    """测试mathfunc.py"""

    @unittest.skipUnless(1>2,"跳过这个测试用例")
    def test_add(self):
        """测试加法add"""
        self.assertEqual(3,add(1,2))
        self.assertNotEqual(3,add(2,2))

    def test_minus(self):
        """测试减法minus"""
        self.assertEqual(1,minus(3,2))

    def test_multi(self):
        """测试乘法multi"""
        self.assertEqual(6,multi(2,3))

    def test_divide(self):
        """测试除法divide"""
        self.assertEqual(2,divide(6,3))
        self.assertEqual(2.5,divide(5,2))
```

HTML测试报告

测试脚本执行后测试结果均是以命令结果的形式展现出来，可读性较差。如果将测试脚本执行的测试结果以样式丰富的形式呈现出来，会大大提升测试结果的可读性

HTMLTestRunner简介

HTMLTestRunner是Python标准库中UnitTest模块的一个扩展，它可以生成HTML测试报告。

HTMLTestRunner的下载地址为http://tungwaiyip.info/software/HTMLTestRunner.html。

下载后，将下载的HTMLTestRunner.py文件放到Python安装路径下的Lib文件中。

修改HTMLTestRunner

大部分开发者现在使用的Python版本可能是Python 3，而前面我们下载的HTMLTest- Runner.py是基于Python 2的版本，所以有些地方需要修改成符合Python 3版本的规范要求

```
# 94行
import StringIO      修改为      import io

# 539行
self.outBuffer=StringIO.String()     修改为      self.outBuffer=io.StingIO()
```

```
# 631行
print >>sys.stderr, '\nTime Elapsed: %s' % (self.stopTime-self.startTime)    修改为
print(sys.stderr, '\nTime Elapsed: %s' % (self.stopTime-self.startTime))

# 642行
if not map.has_key(cls)      修改为      if not cls in map

# 766行
uo=o.decode("latin-1")       修改为      uo=e

# 772行
ue=e.decode("latin-1")       修改为      ue=e
```

测试结果在开启详细模式后，无法输出到html中