**Seed帮助文档**

# “Seed”名字由来

seed名字的由来，用过多种测试框架的人都会了解，不同的框架，如：cucumber、lettuce、PyUnit、nose、Robot Framework等等在使用时需要严格遵守这些框架所规定的规则才能使用，例如cucumber、lettuce框架在使用时需要准备feature文件，然后编写setp脚本，PyUnit等类似单元测试框架，需要集成TestCase基类，编写test开头的方法，Robot Framework定义了类似关键词【可嵌套】驱动的规则，将每种框架当作是水果的话（因为cucumber、lettuce就是水果蔬菜的意思，假设其他框架都是如此），不同的框架有不同的规则，可以看成这些框架就是各式各样的水果一样，这些框架的核心是什么，有哪些？核心就是去掉框架所规定的各种规则，剩下的就是自动化测试框架所具有的核心特性，例如用例驱动（每种都具备）、参数化（有些框架没有）、用例根据属性等规则过滤执行，错误用例的重复执行、不同类型测试报告输出（xml、json、html等）等等，每种核心特性都是用来满足我们对自动化测试框架具有某些核心特性的要求。

试着想一想将这些框架的果肉【规则】去掉，留下种子【核心特性】，将不同的种子【核心特性】融合，形成一枚功能强大的种子【seed】，将更多的决策权交给使用者，例如用例的形式【文本、xml、excel、csv、数据库等等】，除了规定的核心功能之外的一切都可由你去掌控。如此，Seed才可与你珠联璧合。

你可以使用这枚种子打造属于你自己独特的框架【“水果”】

# 为什么使用Seed

seed可以使你的注意力集中在测试方法的编写上，而不用考虑实现自动化框架所具有的一些特性，顶多再编写一个用例的装载器【将载体（excel、txt、csv、数据库等等）中的用例装载到seed框架中】即可让你简单的开始自动化测试之旅。

# Seed快速入门

## 一个最小的案例

程序员开发了一个add接口，这个接口接收两个参数，返回这两个数的和，如下：

def add(a,b):

return 1+1

补充：大家可以将该接口想象成java api、dll、http接口等等

测试人员若要对该接口进行测试时就需要设计测试用例，假设我们设计了如下的用例

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编号 | 描述 | 加数1 | 加数2 | 预期结果 |
| Add\_1001 | 1加1用例 | 1 | 1 | 2 |
| Add\_1002 | 0加0用例 | 0 | 0 | 0 |
| Add\_1003 | 1加0用例 | 1 | 0 | 1 |

想象下这些用例你写在excel中或者txt文本或者csv里，或者数据库中等等

现在我们需要从用例的载体中将这些用例提取出来，如何提取，取决于你的用例载体以及存储方式seed不去为你决定用例的外在表现形式。这里会涉及到excel、txt、csv、数据库操作，都是属于固定知识点，百度或者在python的公共代码库中查找。

读取用例，并放到dict容器中，如下：

case1\_dict = {'id':'Add\_1001','desc':'1加1用例','n1':1,'n2':1,'exp\_res':2}

case2\_dict = {'id':'Add\_1002','desc':'0加0用例','n1':0,'n2':0,'exp\_res':0}

case3\_dict = {'id':'Add\_1003','desc':'1加0用例','n1':1,'n2':0,'exp\_res':1}

补充：dict容器的value值可以是任意类型，可以是dict、list、函数名等等，从而导致我们的用例可以非常的灵活，后面会有一些常见的案例参考，大家也可发挥自己的想象。

注意：关于测试用例的dict容器，seed框架要求用例的dict必须具有id与desc属性，也就是每条用例需要具有用例编号与用例描述，主要目的是为了后期的报告展示，而不是提示10001用例 Fail，10002用例 Pass。

现在我们有了这些用例，但还缺少一个执行这些用例的函数，编写如下：

#假设这是我们的测试方法，用来测试add这个接口的测试方法

def test\_add(test\_case):

#从用例中获取数值1

num1 = test\_case.n1

#从用例中获取数值2

num2 = test\_case.n2

#从用例中获取预期结果

exp\_res = test\_case.res

#调用开发的add接口，传入测试用例中的两个数字，获取调用接口后的实际结果

act\_res = add(num1,num2)

#调用接口后的实际值与预期值进行比较断言

assert exp\_res == act\_res , "{0} add {1} not equal {2}".format(num1,num2,exp\_res)

#不懂什么是断言，请花5分钟百度下

测试方法讲解：

函数接收一个参数，该参数就是我们的测试用例【见上面的dict用例】，可通过访问属性的方式去访问用例中的属性，例如获取用例中的n1属性，使用test\_case.n1 即可访问n1的值。

假想下，我们上面整理的几个dict的用例一一传给这个测试方法，让该方法来执行我们准备好的用例，如果测试方法执行通过，表示用例成功，如果断言时失败，表示用例执行失败，最后报告会自动统计，并记录错误详细信息。

有了测试用例dict，有了测试用例执行方法，接下来我们来装载用例到seed框架中

首先定义个测试集suit【对suit概念不了解的后期会解释】：

#省略了from seed import TestSuit导入TestSuit

add\_suit = TestSuit("add\_suit")

接下来将用例装载到suit对象中：

add\_suit.add\_case(case1\_dict,test\_add)

add\_suit.add\_case(case2\_dict,test\_add)

add\_suit.add\_case(case3\_dict,test\_add)

补充：add\_case函数参数解释，参数1表示导入的用例dict，参数2表示用哪个测试方法来执行该用例

依次装载好我们的用例，有人在想如果有100条用例是不是得写100行？在实际使用的过程中，用例装载的装载器由你们来决定，因为seed框架并不规定你的用例是用何种形式去存储，有人会说这增加了我们的工作量，但是我想说这提高了灵活性，一个框架如何知道用何种外部形式来存储用例最合适，他不知道。我们可以用最适合的用例存储方式【txt、excel、数据库等】，并编写少量的装载代码将用例提取装载到seed即可，对于后期方便用例的管理，这一点点的工作量投入是值得的。

装载完用例后，开始执行：

seed.run()

#-------完整代码------

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import seed

from seed import TestSuit

def add(a,b):

return 1+1

def test\_add(test\_case):

num1 = test\_case.n1

num2 = test\_case.n2

exp\_res = test\_case.res

act\_res = add(num1,num2)

assert exp\_res == act\_res , "{0} add {1} not equal {2}".format(num1,num2,exp\_res)

case1\_dict = {'id':'10001','desc':'test case 10001 desc','name':'test case 10001 name','n1':1,'n2':1,'res':2}

case2\_dict = {'id':'10002','desc':'test case 10002 desc','name':'test case 10002 name','n1':1,'n2':2,'res':3}

case3\_dict = {'id':'10003','desc':'test case 10003 desc','name':'test case 10003 name','n1':1,'n2':3,'res':4}

add\_suit = TestSuit("add\_suit")

add\_suit.add\_case(case1\_dict,test\_add)

add\_suit.add\_case(case2\_dict,test\_add)

add\_suit.add\_case(case3\_dict,test\_add)

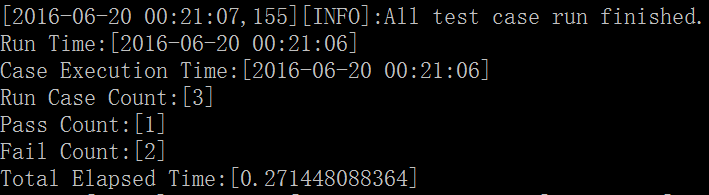
seed.run()

#----------------------

运行：python demo.py

执行就是这么简单，简单说下run方法执行的顺序：会遍历所有的suit，依次执行每个suit下的测试用例，使用装载时对应的测试方法。

默认时，会在标准输出中进行打印执行过程中的日志以及结果：



详细信息为粘贴，大家可实际执行看看。

seed框架提供了各种特性，例如html结果报告、xml结果报告：

运行：python demo.py -o html

执行完成后会在当前目录下生成report.html文件，如下：



果你需要对报告进行定制，可根据提供的所有执行结果获取接口，自行封装。

有关seed其他的高级特性与用法，见seed进阶篇。

# Seed进阶

## setup与teardown说明

关于这两个名词简单的解释下：

使用过单元测试框架的童鞋应该不陌生了，呵呵，请飘过！

setup：主要用来处理测试前的初始化操作

teardown：主要处理测试完成后的收尾工作

## suit级别的setup与teardown的demo

首先创建setup与teardown两个方法【方法名没有要求】

#定义setup方法

def add\_suit\_setup():

print ("call add\_suit\_setup fun")

#定义teardown方法

def add\_suit\_teardown():

print ("call add\_suit\_teardown fun")

在创建suit时，添加这两个方法，如下：

add\_suit = TestSuit("add\_suit",add\_suit\_setup,add\_suit\_teardown)

以快速入门中的例子来说，这时初始化了setup与teardown后执行顺序如下：

1、调用add\_suit\_setup方法

2、调用test\_add（case1\_dict）方法

3、调用test\_add（case2\_dict）方法

4、调用test\_add（case3\_dict）方法

5、调用add\_suit\_teardown方法

每个suit都可以设置自己的setup与teardown，到这里你可以将一些初始化的内容放到setup与teardown中了，还少了一点东西，需要将setup与teardown中的变量传递到测试方法test\_add中，想象下，如果在setup与teardown中定义了数据库连接（变量），就需要在用例方法中使用这个连接变量，seed提供了一个全局级别的全局变量，方便你在整个运行周期中不同模块之间的通讯，请查看seed中world变量章节。

## global级别的setup与teardown的demo

基于以上的suit级别的setup与teardown的demo继续编写，如下：

def global\_setup\_fun():

print ("call global\_setup\_fun")

def global\_teardown\_fun():

print ("call global\_teardown\_fun")

定义了两个方法，具体不解释了。

在seed运行之前，给global级别的setup与teardown进行赋值，如下：

seed.seed\_setup = global\_setup\_fun

seed.seed\_teardown = global\_teardown\_fun

这时候，代码的执行顺序如下：

1、调用global\_setup\_fun

2、调用add\_suit\_setup方法

3、调用test\_add（case1\_dict）方法

4、调用test\_add（case2\_dict）方法

5、调用test\_add（case3\_dict）方法

6、调用add\_suit\_teardown方法

7、调用global\_teardown\_fun方法

非常容易理解吧，大家可试试多个suit时setup与teardown的调用情况，加深seed中setup与teardown的理解。

## seed中的全局变量world

seed中为大家准备了一个全局级别【整个生命周期】的全局变量world。

什么时候会用到全局变量？例如数据库连接，或者用例之间的数据通讯【例如1001用例中会产生一个订单编号，1002用例中会使用这个订单编号】，后期慢慢体会。

seed中提供的world全局变量实际上就是一个dict类型变量，在使用过程中注意KeyError问题【后期会优化自动处理不存在的key】。

比如我们在全局初始化时，设置几个全局变量：

def global\_setup\_fun():

seed.world["order\_no"] = "2013030510001"

seed.world["msg"] = "haha I am global"

print ("call global\_setup\_fun")

在《global级别的setup与teardown的demo》章节中的执行顺序部分，global\_setup\_fun设置了全局变量以后，后面的方法都可以通过seed.world["msg"]来进行访问，比如：

def test\_add(test\_case):

print "Order no:{0}".format(seed.world["order\_no"])

print "msg:{0}".format(seed.world["msg"])

num1 = test\_case.n1

num2 = test\_case.n2

exp\_res = test\_case.res

act\_res = add(num1,num2)

assert exp\_res == act\_res , "{0} add {1} not equal {2}".format(num1,num2,exp\_res)

大家试试下具体会打印什么吧。

## seed运行时参数

### TAG参数

tag参数【-t 或者 --tag】

解释：过滤用例中的tag标签进行过滤

场景：测试用例除了最基本的id与desc以外，通常还分为，用例属于XX模块、优先级、冒烟等等都是属于用例的分类

注意：使用tag参数时，用例的dict里面必须包含tag属性【详见下面的dict例子】

例如用例的dict如下：

case1\_dict = {'id':'10001','desc':'test case 10001 desc','tag':'mod1,zhong','name':'test case 10001 name','n1':1,'n2':1,'res':2}

case2\_dict = {'id':'10002','desc':'test case 10002 desc','tag':'mod2,zhong','name':'test case 10002 name','n1':1,'n2':2,'res':3}

case3\_dict = {'id':'10003','desc':'test case 10003 desc','tag':'mod1,gao','name':'test case 10003 name','n1':1,'n2':3,'res':4}

添加了tag标签，分类之间使用英文逗号隔开。

这些用例如果装载到seed中后，如果运行时，只想运行mod1（模块1）的用例时，命令如下

python demo.py -t mod1

或者

python demo.py --tag mod1

如果进行多个tag的过滤

python demo.py -t mod1 -t gao

意思是：执行tag中包含mod1或者tag中包含gao的用例

### failfast参数

failfast参数【-f 或者 --failfast】

使用该参数时，如果遇到异常就立马结束执行。

场景：有时候我们的脚本写好了，用例也都装载结束了，这时候需要进行脚本的试运行，看看可有什么异常出现，这时候可以使用这个参数，能在遇到的第一个异常就停下

命令如下：

python demo.py -f

或者

python demo.py –failfast

### rerun参数

rerun参数【-r 或者 --rerun】

该参数会对上一次执行失败的用例重新再执行一遍，需要基于运行一次之后

场景：假设我们有100条用例，第一次执行后发现了有20个失败了，开发修改后，并确认对之前的用例没有影响，这时我们可以快速的对20条失败的用例进行重新运行看看修复情况，而不需要对以前正确的80条用例重新执行而浪费大量的时间。

实现原理：seed在执行一次后会在demo.py当前目录下创建一个《last\_time\_fail.txt》文档，并在其中写入执行失败的用例id，-r参数会从这个文档中读取用例id并执行对应的用例。

注意：如果《last\_time\_fail.txt》文档中没有失败的用例id，或者没有《last\_time\_fail.txt》文档时，使用-r参数时，则会重新执行整个用例。

小技巧：你可以对《last\_time\_fail.txt》文档进行编辑，配合-r参数来进行用例执行

命令如下

python demo.py -r

或者

python demo.py –rerun

### out\_type参数

out\_type参数【-o 或者 --out\_type】

该参数用来指定seed运行完所有的测试后，最后结果的输出形式

out\_type参数最后的可选参数有cmd【默认】、html、xml，后期了解了seed中的total\_result的结构后可自己定制，非常灵活【因为seed使用的基本上都是平民化的代码来编写】

注意：不加参数时，默认使用cmd方式

使用html、xml参数时，会在demo.py目录下生成对应的report文件

命令如下：

python demo.py -o html

或者

python demo.py --out\_type html

## seed初始化导入case\_fun

略

(当你的测试方法越来越多时，seed的初始化导入可以使你按照目录以及文件的方式来管理你的脚本)

## total\_result结构

略

(seed运行完后通过seed.total\_res获取所有的suit以及case执行后的结果，采用json的方式返回，方便获取后进行定制化的处理)

## xml结果转html插件

略