Table of Contents

序

最近在总结以往的工作: Robot Framework作为在诺西工作时期接触到的框架，其不仅仅是自己最早接触到的大型测试框架，同时还引导自己进入了Python世界。 感情深厚，所以第一个整理工作就从此开始。

网上简单搜索了一下Robot Framework的资料，英文资料居多，中文资料多为用户手册的翻译，并且翻译的内容篇幅不大。 也找到一篇代码阅读解析，不过是比较老版本的代码，看了一些出入颇大。 再次阅读Robot Framework框架过程中，突发奇想，觉得将学习成果整理为文档，定是一件极好的事情。 如果自己的整理能成为填补空白的中文资料，也不枉一份极有意义的工作。

这应该算是自己第一本开源书的写作，大致思路如下:
介绍框架，整理一遍全面但又不失简洁的用户手册；
然后介绍一些进阶的框架使用帮助，主要为测试库的扩展；
最后对一些重要的流程，进行代码的阅读分析。

希望本文既能帮助到他人，也为自己留下重要的学习笔记。

虞科敏
2016/7

---

gitbook.com只会更新第1章节，即简明使用手册。
完整**pdf**版本，将会放在github.com

第1章 简明使用手册

作者:虞科敏

本节内容可以理解为官方文档《用户手册》的摘录和笔记，主要包括重要的语法，使用方法，以便为后续部分进行重要的测试情景的代码分析做好框架知识准备。

后续章节，请在本开源书的完整版中获得

测试数据, Test Data

作者:虞科敏

文件和目录

1. 测试用例文件, Test Case File
   测试用例创建在用例文件中.
   用例文件会自动创建一个包含文件中所定义用例的测试套件, TestSuite.
2. 目录
   包含多个测试文件的目录， 形成一个更高一级的测试套件。
   Suite目录拥有从测试文件创建的套件，将它们作为目录Suite的子suite。
   suite目录还可以包含其他的suite目录，这种层级结构可以递归嵌套。
   suite目录可以拥有一个特别的__init__文件。
3. 特别的测试文件 Test Libraries, 包含低级别的关键字
   Resource File, 包含变量Variables，高级别的用户自定义关键字
   Variable File, 提供资源文件外更灵活的创建变量的手段

支持的文件格式

1. HTML
2. TSV
3. Plain TEXT
4. reStructuredText

以Plain Text举例，其他格式请参考用户手册

空格分隔

```
*** Settings ***
Library       OperatingSystem

*** Variables ***
${MESSAGE}    Hello, world!

*** Test Cases ***
My Test
    [Documentation]    Example test
    Log    ${MESSAGE}
    My Keyword    /tmp

Another Test
    Should Be Equal    ${MESSAGE}    Hello, world!

*** Keywords ***
My Keyword
    [Arguments]    ${path}
    Directory Should Exist    ${path}
```

管线和空格分隔

```
| *Setting*  |     *Value*     |
| Library    | OperatingSystem |

| *Variable* |     *Value*     |
| ${MESSAGE} | Hello, world!   |

| *Test Case*  | *Action*        | *Argument*   |
| My Test      | [Documentation] | Example test |
|              | Log             | ${MESSAGE}   |
|              | My Keyword      | /tmp         |
| Another Test | Should Be Equal | ${MESSAGE}   | Hello, world!

| *Keyword*  |
| My Keyword | [Arguments] | ${path}
|            | Directory Should Exist | ${path}
```

数据表格

共计4种表格，以表格中的第一个cell在标识： Settings, Variables, Test Cases, Keywords.
大小写敏感； 单数形式也可以接受，比如Setting, Variable, Test Case, Keyword.

1. Settings
   1) 引入test libraries, resource files, variable files.
   2) 定义suite和case的元数据metadata
2. Variables
   定义变量，可被其他测试数据使用
3. Test Cases
   使用可用的关键字创建测试用例
4. Keywords
   使用已有的低级关键字创建更高级的用户关键字

测试数据解析的规则

1. RF框架解析测试数据时，会忽略一些信息, 细节请参考用户手册。
2. 处理空白字符Whitespace, 转义字符Escaping, 细节请参考用户手册。
3. 空cell的技巧： \ 或者 ${EMPTY}
4. 空格的技巧： \ 或者 ${SPACE}
5. 多行技巧： ...

Tips: 会被忽略的信息

```
在第一个cell中表名字非法的表格
第一行中第一个cell后的其他内容
第一个表前面的所有内容（如果允许，表之间的内容也会被忽略）
所有的空行（空行一般用来提高可读性）
每行末尾的空cell（除非被转义）
所有单个的\（当不用做转义时）
当#时一个cell中的第一个字符，#后面的所有字符（#可被用来作为注释之用）
在HTML或reStructuredText中的所有格式信息
```

测试用例, **Test Case**

作者:虞科敏

基础语法

测试用例在test case table中创建，使用各种合法可用的关键字。
关键字的来源： 从test libraries或者resource file中导入； 在用例文件自身的keyword table中创建。

样例

```
*** Test Cases ***
Valid Login
    Open Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open

Setting Variables
    Do Something    first argument    second argument
    ${value} =    Get Some Value
    Should Be Equal    ${value}    Expected value
```

用例表中的**Settings**

Force Tags, Default Tags
Test Setup, Test Teardown
Test Template
Test Timeout

样例

```
*** Test Cases ***
Test With Settings
    [Documentation]    Another dummy test
    [Tags]    dummy    owner-johndoe
    Log    Hello, world!
```

参数

关键字的参数，可以类比Python参数来理解，实际上其实现即为Python语言实现，难怪行为也如此相似。 在关键字的文档中也会用类似语法说明出来。 使用Python的同学理解RF中关键字的参数，对比Python的相关行为很容易理解。 将关键字参数和python中的参数进行对比如下。

必选参数, **Mandatory arguments**

=> 类比python的位置参数，positional arg

```
*** Test Cases ***
Example
    Create Directory    ${TEMPDIR}/stuff
    Copy File    ${CURDIR}/file.txt    ${TEMPDIR}/stuff
    No Operation
```

参数的缺省值, **Default values**

=> 类比python的默认参数 (arg=value)

```
*** Test Cases ***
Example
   Create File    ${TEMPDIR}/empty.txt
    Create File    ${TEMPDIR}/utf-8.txt         Hyvä esimerkki
    Create File    ${TEMPDIR}/iso-8859-1.txt    Hyvä esimerkki    ISO-8859-1
```

可变长参数, **Variable number of arguments**

=> 类比python的非关键字可变长参数(元组)（*arg)

```
*** Test Cases ***
Example
    Remove Files    ${TEMPDIR}/f1.txt    ${TEMPDIR}/f2.txt    ${TEMPDIR}/f3.txt
    @{paths} =    Join Paths    ${TEMPDIR}    f1.txt    f2.txt    f3.txt    f4.txt
```

命名参数, **Named arguments**

=> 类比python的关键字参数，可以直接指定参数名来进行赋值

样例*1* 正常使用举例

```
*** Settings ***
Library    Telnet    prompt=$    default_log_level=DEBUG

*** Test Cases ***
Example
    Open connection    10.0.0.42    port=${PORT}    alias=example
    List files    options=-lh
    List files    path=/tmp    options=-l

*** Keywords ***
List files
    [Arguments]    ${path}=.    ${options}=
    List files    options=-lh
    Execute command    ls ${options} ${path}
```

样例*2* 特殊使用举例

```
*** Test Cases ***
Example
    Run Program    shell=True    # This will not come as a named argument to Run Process!
    Log    foo\=quux  # this will get the value "foo=quux"

*** Keywords ***
Run Program
    [Arguments]    @{args}
    Run Process    program.py    @{args}    # Named arguments are not recognized from inside @{args}!
```

自由参数, **Free keyword arguments**

=> 类比python的关键字可变长参数(字典) (**arg)，其会收集所有使用name=value语法，并且不匹配任何其他类型参数的内容作为kwargs

样例*1*
Run Process关键字(来自Process库), signature: command, *arguments, **configuration
command: 将执行的命令*, its arguments as variable number of arguments (* arguments: 可变长参数
**configuration: 自由参数

```
*** Test Cases ***
Using Kwargs
    Run Process    program.py    arg1    arg2    cwd=/home/user
    Run Process    program.py    argument    shell=True    env=${ENVIRON}
```

样例*2*
对Run Process关键字的包裹Run Program

```
*** Test Cases ***
Using Kwargs
    Run Program    arg1    arg2    cwd=/home/user
    Run Program    argument    shell=True    env=${ENVIRON}

*** Keywords ***
Run Program
    [Arguments]    @{arguments}    &{configuration}
    Run Process    program.py    @{arguments}    &{configuration}
```

失败

直接使用fails关键字，可以让用例失败: 然后执行test teardown，然后执行下一个用例
使用continuable failures关键字，如果不希望停止用例执行

失败消息: 在fails关键字中可以直接指定失败消息内容，有些关键字也可以指定失败消息内容

样例

```
*** Test Cases ***
Normal Error
    Fail    This is a rather boring example...

HTML Error
    ${number} =    Get Number
    Should Be Equal    ${number}    42    *HTML* Number is not my **MAGIC** number.
```

用例名字和文档串

直接阅读样例理解即可，不作过多说明

样例

```
*** Test Cases ***
Simple
    [Documentation]    Simple documentation
    No Operation

Formatting
    [Documentation]    *This is bold*, _this is italic_  and here is a link: http://robotframework.org
    No Operation

Variables
    [Documentation]    Executed at ${HOST} by ${USER}
    No Operation

Splitting
    [Documentation]    This documentation    is split    into multiple columns
    No Operation

Many lines
    [Documentation]    Here we have
    ...                an automatic newline
    No Operation
```

标签

标签用来分类用例，方便有选择的执行和查看。
标签会出现在报告，日志，统计信息中，提供用例的元数据信息。
使用标签，include和exclude可以容易选择用例，也方便指定关键用例Critical.

Setting Table之强制标签, Force Tags

在测试文件中，本文件中的所有用例都将加上此标签
在suite的__init__文件中，所有的子suite中的用例都将加上此标签

Setting Table之缺省标签, Default Tags

没有Tags定义的用例将加上此标签
不支持suite的__init__文件

Test Case Table之[Tags]

指定本测试用例的标签
同时，它会覆盖缺省标签

命令行中设定标签

--settag

BuiltIn关键字可以在执行过程中动态操纵标签

Set Tags
Remove Tags
Fail
Pass Execution

标签定义时候是自由无限制的； 但是标签有一个normalized过程，将转为全小写，并且移除空格，这个在阅读框架代码是可以看到。
标签可以使用变量来定义。

样例

```
*** Settings ***
Force Tags      req-42
Default Tags    owner-john    smoke

*** Variables ***
${HOST}         10.0.1.42

*** Test Cases ***
No own tags
    [Documentation]    This test has tags owner-john, smoke and req-42.
    No Operation

With own tags
    [Documentation]    This test has tags not_ready, owner-mrx and req-42.
    [Tags]    owner-mrx    not_ready
    No Operation

Own tags with variables
    [Documentation]    This test has tags host-10.0.1.42 and req-42.
    [Tags]    host-${HOST}
    No Operation

Empty own tags
    [Documentation]    This test has only tag req-42.
    [Tags]
    No Operation

Set Tags and Remove Tags Keywords
    [Documentation]    This test has tags mytag and owner-john.
    Set Tags    mytag
    Remove Tags    smoke    req-*
```

保留标签: RF的预留标签都使用**"robot-"**作为前缀，用户不要自己定义以此为前缀的标签。 当前，只有在正常停止测试执行时， "robot-exit"标签会被自动添加到测试用例。

测试固定件**Fixture: setup**和**teardown**

Fixture: setup和teardown语义等同于其他测试框架中相似的部件，如果不清楚请自己阅读用户手册。

Fixture: setup和teardown中一般使用单个的关键字。 如果需要关注多个任务，也请创建高级的用户关键字来进行。 另外， 你也可以使用BuiltIn中**Run Keywords**关键字。

teardown的2点特别说明：
即时测试失败， teardown也会被执行， 这里是进行必行执行的clean-up活动的最佳地方
teardown中的关键字，就算其中某些关键字执行失败，其他关键字也会被执行。 这个行为也可以通过**Continue on faliure**系列关键字(比如，Run Keyword And Ignore Error, Run Keyword And Expect Error)来得到。

最方便指定setup和teardown的地方是测试文件的Setting Table中使用Test Setup和Test Teardown进行设置。 T 单独的测试用例， 也可以有它自己的[Setup]和[Teardown]。

样例

```
*** Settings ***
**Test Setup**       Open Application    App A
**Test Teardown**    Close Application

*** Test Cases ***
Default values
    [Documentation]    Setup and teardown from setting table
    Do Something

Overridden setup
    [Documentation]    Own setup, teardown from setting table
    **[Setup]**    Open Application    App B
    Do Something

No teardown
    [Documentation]    Default setup, no teardown at all
    Do Something
    **[Teardown]**

No teardown 2
    [Documentation]    Setup and teardown can be disabled also with special value NONE
    Do Something
    **[Teardown]**    **NONE**

Using variables
    [Documentation]    Setup and teardown specified using variables
    **[Setup]**    **${SETUP}**
    Do Something
    **[Teardown]**    **${TEARDOWN}**
```

测试模板, **Test Template**

作者:虞科敏

测试模板，关键字驱动 => 数据驱动

关键字驱动, 用例主体由若干关键字+参数构成
vs.
数据驱动, 用例主体只由Template关键字的参数构成

用途举例:
对每个测试用例，或者一个测试文件中的所有用例，重复执行同一个关键字多次（使用不同数据）
也可以只针对测试用例，或者每个测试文件只执行一次

模板关键字可以接受普通的位置参数，命名参数
关键字名中可以使用参数
不可以使用变量定义模板关键字

样例*1*
[Template]会覆盖Setting Table中的template设置
如果[Template]为空值，意味着没有模板

```
*** Test Cases **
Normal test case
    Example keyword    first argument    second argument

Templated test case
    [Template]    Example keyword
    first argument    second argument
```

样例*2*
对于多行数据，模板关键字会逐行调用执行，一次一行 如果其中有些失败，其他也会执行。 对于普通用例的continue on failure模式，对于模板关键字是缺省行为。

```
*** Settings ***
Test Template    Example keyword

*** Test Cases ***
Templated test case
    first round 1     first round 2
    second round 1    second round 2
    third round 1     third round 2
```

样例*3*
模板关键字支持嵌入参数的语法
关键字名字就作为参数的持有者，在实际执行中这些参数会被模板关键字解析出实际的参数，传递给低级的底层关键字作为参数

```
*** Test Cases ***
Normal test case with embedded arguments
    The result of 1 + 1 should be 2
    The result of 1 + 2 should be 3

Template with embedded arguments
    [Template]    The result of ${calculation} should be ${expected}
    1 + 1    2
    1 + 2    3

*** Keywords ***
The result of ${calculation} should be ${expected}
    ${result} =    Calculate    ${calculation}
    Should Be Equal    ${result}    ${expected}
```

样例*4*
模板关键字名字中的参数个数必须匹配它将使用的参数数量
参数名不需要匹配原始关键字的参数

```
*** Test Cases ***
Different argument names
    [Template]    The result of ${foo} should be ${bar}
    1 + 1    2
    1 + 2    3

Only some arguments
    [Template]    The result of ${calculation} should be 3
    1 + 2
    4 - 1

New arguments
    [Template]    The ${meaning} of ${life} should be 42
    result    21 * 2
```

样例*5* 带有for循环的模板关键字

```
*** Test Cases ***
Template and for
    [Template]    Example keyword
    :FOR    ${item}    IN    @{ITEMS}
    \    ${item}    2nd arg
    :FOR    ${index}    IN RANGE    42
    \    1st arg    ${index}
```

不同的测试用例风格

- 关键字驱动
  - 描述工作流
  - 若干关键字和他们必要的参数
- 数据驱动
  - 针对相同工作流，执行不同的输入数据
  - 只使用一个高级的用户关键字，其中定义了工作流，然后使用不同的输入和输出数据测试相同的场景
  - 每个测试中可以重复同一个关键字，但是test template功能只允许定义以此被使用的关键字
- 行为驱动：
  - 描述工作流
  - Acceptance Test Driven Development, ATDD
  - Specification by Example
  - BDD's Given-When-Then

- And or But，如果测试步骤中操作较多
- 支持嵌入数据到关键字名

样例*1*

```
*** Settings ***
Test Template    Login with invalid credentials should fail

*** Test Cases ***                USERNAME         PASSWORD
Invalid User Name                 invalid          ${VALID PASSWORD}
Invalid Password                  ${VALID USER}    invalid
Invalid User Name and Password    invalid          invalid
Empty User Name                   ${EMPTY}         ${VALID PASSWORD}
Empty Password                    ${VALID USER}    ${EMPTY}
Empty User Name and Password      ${EMPTY}         ${EMPTY}

*样例2*
*** Test Cases ***
Invalid Password
    [Template]    Login with invalid credentials should fail
    invalid          ${VALID PASSWORD}
    ${VALID USER}    invalid
    invalid          whatever
    ${EMPTY}         ${VALID PASSWORD}
    ${VALID USER}    ${EMPTY}
    ${EMPTY}         ${EMPTY}
```

样例1和样例2都是数据驱动的test template样例。
样例1有命令列，方便阅读理解; test template在setting table中定义; 每行有名字也方便查看结果（如果行数不是太多的话）
样例2在一个用例中完成所有的事情

样例*3*
搜索关键字的时候， 如果full name没有搜索到， Given-When-Then-And-But等前缀会被忽略

```
*** Test Cases ***
Valid Login
    Given login page is open
    When valid username and password are inserted
    and credentials are submitted
    Then welcome page should be open
```

测试套件, **Test Suite**

作者**:**虞科敏

测试用例 => Test File => 以目录进行组织

Test Case Files

一个文件中用例数建议 <10; 如果用例数过多建议考虑数据驱动

用例文件会自动创建一个包含文件中所定义用例的测试套件, TestSuite

Test Suite Dirctories

test case files被组织为目录， 这些目录形成更高级的suite。

由目录创建的test suite不能有任何直接的测试用例(Test Cases), 但可以包含其他带有测试用例的suite。

这些目录又可以被放进其他目录，形成更高级的suite。 包含的层级深度有没有特别的限制。

当一个测试目录被执行，它包含的文件和目录按以下方式递归处理：

1. 以"."和"_"开始的文件和目录被跳过
2. 目录名为CVS(大小写敏感)被跳过
3. 扩展名非法（有效扩展名: .html, .xhtml, .htm, .tsv, .txt, .rst, or .rest）的文件被跳过
4. 其他文件和目录被处理

   Tips: 如果文件或目录不包含任何测试用例Test Case，会被静默地跳过; 如果需要产生警告，可以在命令行使用 --warnonskippedfiles

初始化文件

目录创建的Suite可以拥有和Test Case File创建的suite相似的设置，这些设置信息被放在Suite初始化文件中。
格式: __init__.ext （Pythoner们是不是觉得很熟悉，RF借用了python的命名方式，不过做的事情和python的__init__.py有些不同)
合法的.ext:

- .html, .htm and .xhtml for HTML
- .tsv for TSV
- .txt and special .robot for plain text
- .rst and .rest for reStructuredText

除了不能拥有Test Case Table，Setting内容有一定限制外， __init__.ext和test case file的结构和语法都保持一致.

在__init__中创建或引入的变量Variables和关键字Keywords，对于其子suite并不可用;如果你需要共享这些变量和关键字， 你需要考虑使用resource file, 它们可以被__init__和test case file同时使用。

__init__.ext的主要用途在于指定类似在test case file中进行的suite相关设置； 另外一些case相关设置也可能在这里进行:

- Documentation, Metadata, Suite Setup, Suite Teardown: 参考test case file部分

- Force Tags: 无条件的指定本目录下的所有用例（直接或递归地）标签
- Test Setup, Test Teardown, Test Timeout: 指定本目录下的所有用例（直接或递归地），可以被更低级别层面的定义覆盖
- Default Tags, Test Template: 不支持！！！！！！！！

样例

```
*** Settings ***
Documentation    Example suite
Suite Setup      Do Something    ${MESSAGE}
Force Tags       example
Library          SomeLibrary

*** Variables ***
${MESSAGE}       Hello, world!

*** Keywords ***
Do Something
    [Arguments]    ${args}
    Some Keyword    ${arg}
    Another Keyword
```

定制化**Test Suite**

名字

Suite Name来自测试文件名或者目录名。 参考举例来理解，不复杂:
some_tests.html => suite name: Some Tests
My_test_directory => suite name: My test directory

可以添加前缀(pre__, 2个下划线)来指定执行顺序，但是前缀不会被包含在suite name中。
01__some*tests.txt* => *suite name: Some Tests*
*02*_more_tests.txt => suite name: More Tests
Some Tests在More Tests前被执行

命令行选项--name，可以覆盖top-level suite的name定义

文档

可以在Setting Table里用**Documentation**设置: 可以在test case file中，或者高级suite的__init__文件中

命令行选项--doc，可以覆盖top-level suite的documentation定义

样例

```
*** Settings ***
Documentation    An example test suite documentation with *some* _formatting_.
...              See test documentation for more documentation examples.
```

元数据

其他元数据可以在Setting Table里用**Metadata**设置: 以此方式定义的元数据会显示在测试报告和日志中。

命令行选项--metadata，可以覆盖top-level suite的metadata定义

```
*** Settings ***
Metadata    Version        2.0
Metadata    More Info      For more information about *Robot Framework* see http://robotframework.org
Metadata    Executed At    ${HOST}
```

Suite Setup和Suite Teardown

测试固定件Fixture: setup和teardown的语义，和其他测试框架没有太多不同。 很多情况也和Tese Case的fixture差不多，可以参考Test Case的相关章节。
也是推荐只使用一个关键字

如果suite setup失败， suite下的所有测试用例和它的子suite都会被标记为失败，并且不会再被执行。 所以suite setup适合用于检查必须满足的前提条件。

suite teardown和test case teardown很类似，也满足continue on failure模式。 如果suite teardown失败, suite下的所有用例被标记为失败, 不管之前这些用例的执行结果如何.

作为setup或者teardown的被执行的关键字名可以为变量Varialbe。

```
小技巧： 可以通过命令行来指定变量，来实现不同环境执行不同的setup和teardown
```

测试库, **Test Library**

作者:虞科敏

测试库包含底层关键字，库关键字Library Keywords.

导入库

- 使用Setting Table中的Library Setting导入
  库名字是大小写和空格敏感的
  如果库存在一个包Package中，应提高包含包的全名Full Name
  库可能需要参数进行初始化
  库导入时支持参数缺省值，可变长参数，命名参数等语法，这和关键字的参数语法很像 库名和参数都可以使用变量

样例

可以在Resource File, Test Case File, Suite __init__ File中导入库，库中的关键字在导入库的文件中可用

```
*** Settings ***
Library    OperatingSystem
Library    my.package.TestLibrary
Library    MyLibrary    arg1    arg2
Library    ${LIBRARY}
```

- 使用关键字**Import Library**

导入的方法和Library Setting很相似
导入的关键字在Import Library关键字被使用的suite中可用

样例

```
*** Test Cases ***
Example
    Do Something
    Import Library    MyLibrary    arg1    arg2
    KW From MyLibrary
```

指定导入的库

使用库的名字

RF框架会在[module search path][1]中根据库名字找寻实现库的类或者模块

理解Module Search Path对于理解库的查找非常重要，这个实际就是使用Python的解释器进行模块搜索的概念，了解Python的同学不难理解。

- Python解释器的安装目录，三方库的安装目录，自动就在搜索路径中。 在Python的搜索路径中的模块，不需要经过其他配置，就能被导入
- 更多的路径信息，可以通过PYTHONPATH, JYTHONPATH, IRONPYTHONPATH环境变量进行设置
- 命令行选项 --pythonpath (-P) 也可以添加更多的地址到搜索路径中
- 通过编程控制sys.path属性

- 如果使用Jython，还可以使用Jython的模块搜索路径或者Java Classpath

样例

```
--pythonpath libs
--pythonpath /opt/testlibs:mylibs.zip:yourlibs
--pythonpath mylib.jar --pythonpath lib/STAR.jar --escape star:STAR
```

[1] 路径搜索和搜索路径
模块的导入需要一个叫做"路径搜索"的过程。 即在文件系统"预定义区域"中查找 mymodule.py 文件(如果你导入 mymodule 的话)。 这些预定义区域只不过是你的 Python 搜索路径的集合。路径搜索和搜索路径是两个不同的概念, 前者是指查找某个文件的操作, 后者是去查找一组目录。
默认搜索路径是在编译或是安装时指定的。 它可以在一个或两个地方修改。
一个是启动Python的shell或命令行的PYTHONPATH环境变量。 该变量的内容是一组用冒号分割的目录路径。 如果你想让解释器使用这个变量, 那么请确保在启动解释器或执行Python脚本前设置或修改了该变量。
解释器启动之后, 也可以访问这个搜索路径, 它会被保存在 sys 模块的 sys.path 变量里。不过它已经不是冒号分割的字符串, 而是包含每个独立路径的列表。 这只是个列表, 所以我们可以随时随地对它进行修改。 如果你知道你需要导入的模块是什么,而它的路径不在搜索路径里, 那么只需要调用列表的 append() 方法即可, 就像这样: sys.path.append('/home/wesc/py/lib'). 修改完成后, 你就可以加载自己的模块了。 只要这个列表中的某个目录包含这个文件, 它就会被正确导入。

使用库的物理路径

另一种思路是使用库在系统中的路径来指定库
路径以相对于当前test data file所在目录的相对地址来给出，这和Resource Files, Vairable Files很相似
如果库是文件，那么.ext，文件的扩展名必须被包括在路径中，比如Java的.class或.java，Python的.py等
如果库是目录，那么路径应该以"/"结尾

样例

局限: 实现为Python类的库，Python类所在的module名必须和类名相同

```
*** Settings ***
Library    PythonLibrary.py
Library    /absolute/path/JavaLibrary.java
Library    relative/path/PythonDirLib/    possible    arguments
Library    ${RESOURCES}/Example.class
```

库别名

有很多场景，我们使用客户定制化的库名（库别名）会更加方便，或不得不使用定制化的库名:

- 使用不同的参数导入同一个库多次
- 库名太长，比如很多包名很长
- 使用变量在不同的环境中导入不同的包， 但你想使用相同的名字进行引用
- 库名很糟糕，很容易误导人

Library Setting或者关键字Import Keyword都支持库别名

样例*1*
WITH NAME 大小写敏感
指定的名字会在Log中显示
使用关键字全名时，必须使用库别名

```
*** Settings ***
Library    com.company.TestLib    WITH NAME    TestLib
Library    ${LIBRARY}             WITH NAME    MyName
```

样例*2*
使用不同的参数引入同一个库多次

```
*** Settings ***
Library    SomeLibrary    localhost        1234    WITH NAME    LocalLib
Library    SomeLibrary    server.domain    8080    WITH NAME    RemoteLib

*** Test Cases ***
My Test
    LocalLib.Some Keyword     some arg       second arg
    RemoteLib.Some Keyword    another arg    whatever
    LocalLib.Another Keyword
```

标准库, **Standard Library**

RF框架提供如下标准库。

```
BuiltIn
Collections
DateTime
Dialogs
OperatingSystem
Process
Screenshot
String
Telnet
XML
```

另外， 除了普通标准库，还存在另一种完全不同的远程库, Remote Library.
远程库不提供任何关键字， 它在RF框架和远程的库实现之间以代理的形式工作.
远程的库实现可以运行在和RF框架不同的其他机器上，甚至可以被并非RF支持的编程语言实现。

外部库, **External Library**

任何非标准库的测试库，称为外部库。
RF开源社区已经一些常用的测试库，没有打包到框架核心之中. 更多公共外部库，可以访问 http://robotframework.org

```
Selenium2Library
SwingLibrary
```

变量, Variable

作者: 虞科敏

很多测试数据场景会使用变量。 最常见的，比如 用作关键字的参数， Setting的值等。
普通关键字名中不能使用变量，但内建关键字Run Keyword可以用来达到相似的效果。

3种变量类型

- 标量scalars - ${SCALAR}
- 列表lists - @{LIST}
- 字典dictionaries - &{DICT}

另外，环境变量可以直接使用语法 %{ENV_VAR} 进行访问

变量的使用场景举例： 在测试数据中，某字符串经常改变： 使用变量你只需要在一个地方进行改变
创建系统独立或者操作系统独立的测试数据： 变量可以使用命令行进行指定
关键字中需要对象参数，而不仅仅是字符串
不同的库，不同的关键字需要通信： 返回值可以付给变量，然后传给另一个关键字
测试数据的值太长，太复杂

如果测试数据使用了一个不存在的变量，关键字执行会失败
如果不希望获取变量的值，可以进行转义 ${NAME}

变量类型

变量和关键字一样，大小写敏感
空格和下划线会被忽略

建议:
全局变量使用大写，如${PATH} or ${TWO WORDS}
只在某用例或关键字中使用的变量使用小写，如 ${my var} or ${myVar}
最重要的，大小写的风格要保持一致
变量名使用字母表字符(a-z, A-Z)，数字(0-9), 下划线(_)和空格，这也是扩展变量的所明确要求的

标量, scalars - ${SCALAR}

不只是字符串，任何对象实例都可以付给标量变量，比如list
当标量变量是cell中的唯一值时， 变量变量会被它所有的值所代替
当标量变量在cell中和其他元素一起存在时，变量变量会被转换为unicode，然后和其他元素进行合并

```
Tips: 当参数使用命名参数语法被传递给关键字(argname=${var})，标量变量值被用作as-is值， 不会进行转换，
```

样例*1*
假设 ${GREET} = Hello and ${NAME} = world， 以下两个关键字的效果是一样的

```
*** Test Cases ***
Constants
    Log    Hello
    Log    Hello, world!!

Variables
    Log    ${GREET}
    Log    ${GREET}, ${NAME}!!
```

样例2 假设 ${STR} = Hello, world!
${OBJ} = new MyObj()

```
public class MyObj {
    public String toString() {
        return "Hi, tellus!";
    }
}
```

KW1得到字符串 Hello, world!
KW2得到MyObj对象
KW3得到字符串 I said "Hello, world!"
KW4得到字符串 You said "Hi, tellus!"

```
*** Test Cases ***
Objects
    KW 1    ${STR}
    KW 2    ${OBJ}
    KW 3    I said "${STR}"
    KW 4    You said "${OBJ}"
```

列表，lists - @{LIST}

标量变量${EXAMPLE}， 值以"as-is"的方式被直接使用
列表变量@{EXAMPLE}, 单个的list item被分别传递给关键字

样例1

假设 @{USER} = ['robot', 'secret']，关键字Constants和List Variable的效果是一样的

```
*** Test Cases ***
Constants
    Login    robot    secret

List Variable
    Login    @{USER}
```

RF框架存储变量在一个内部存储中，允许使用这些变量作为scalars, lists 或者 dictionaries。

和其他数据一起使用lists

lists可以和其他数据一起，包括和其他list参数一起使用。

样例

```
*** Test Cases ***
Example
    Keyword    @{LIST}    more    args
    Keyword    ${SCALAR}    @{LIST}    constant
    Keyword    @{LIST}    @{ANOTHER}    @{ONE MORE}
```

访问**list**中的**item**

@{NAME}[index]访问列表中的单个元素
此处的语法和Python中的list访问是相似的，比如从0开始，比如-1访问倒数第1个元素

样例

```
*** Test Cases ***
List Variable Item
    Login    @{USER}
    Title Should Be    Welcome @{USER}[0]!

Negative Index
    Log    @{LIST}[-1]

Index As Variable
    Log    @{LIST}[${INDEX}]
```

list在**setting**中的使用

导入libraries和variable files： list变量不能为库名和变量文件名，但可以用作参数
setups和teardowns： list变量不能为关键字名，但可以用作参数
tags： 可以自由使用

```
*** Settings ***
Library         ExampleLibrary      @{LIB ARGS}    # This works
Library         ${LIBRARY}          @{LIB ARGS}    # This works
Library         @{NAME AND ARGS}                   # This does not work
Suite Setup     Some Keyword        @{KW ARGS}     # This works
Suite Setup     ${KEYWORD}          @{KW ARGS}     # This works
Suite Setup     @{KEYWORD}                         # This does not work
Default Tags    @{TAGS}                            # This works
```

字典**, dictionaries - &{DICT}**

字典变量&{EXAMPLE}, 单个的dict item被分别以命名参数形式()传递给关键字

样例
假设&{USER} = {'name': 'robot', 'password': 'secret'}, 关键字Constants和Dict Variable是等价的

```
*** Test Cases ***
Constants
    Login    name=robot    password=secret

Dict Variable
    Login    &{USER}
```

和其他数据一起使用**dict**

dict可以和其他数据一起，包括和其他dict参数一起使用。

样例 语法要求，位置参数必须在命名参数之前，所以dict后面只能跟dict或者命名参数

```
*** Test Cases ***
Example
    Keyword    &{DICT}    named=arg
    Keyword    positional    @{LIST}    &{DICT}
    Keyword    &{DICT}    &{ANOTHER}    &{ONE MORE}
```

访问**dict**中的**item**

&{NAME}[key]访问字典中的单个元素
此处的语法和Python中的dict访问是相似的
key被认为为字符串，但是非字符串的key也可以被用作变量

样例

```
*** Test Cases ***
Dict Variable Item
    Login    &{USER}
    Title Should Be    Welcome &{USER}[name]!

Variable Key
    Log Many    &{DICT}[${KEY}]    &{DICT}[${42}]
```

dict在**setting**中的使用

字典变量通常不能用在setting中，除了imports, setups, teardowns，字典可以用作参数

样例

```
*** Settings ***
Library        ExampleLibrary    &{LIB ARGS}
Suite Setup    Some Keyword      &{KW ARGS}     named=arg
```

环境变量**, Environment Variables - %{ENV_VAR}**

%{ENV_VAR_NAME}直接访问环境变量
环境变量都被认为为字符串
环境变量在测试执行之前，在操作系统中被设置，在执行过程中一直可用

环境变量是全局的，在一个用例中被设置，后续执行的用例都可以使用
对于环境变量的改变，在测试结束后将失效
OperatingSystem.Set Environment Variable
OperatingSystem.Delete Environment Variable

样例

```
*** Test Cases ***
Env Variables
    Log    Current user: %{USER}
    Run    %{JAVA_HOME}${/}javac
```

Java系统属性

如果使用Jython，Java系统属性可视为环境变量被使用
如果Java系统属性和环境变量同名，环境变量将被使用

样例

```
*** Test Cases ***
System Properties
    Log    %{user.name} running tests on %{os.name}
```

创建变量, Creating Variables

作者: 虞科敏

有以下途径可以创建变量：

- 通过Vairable Table
- 在Vairable File中定义
- 使用命令行选项
- 来自关键字的返回值
- 使用内建关键字设置

变量表, Variable Table

最常见的创建变量的地方，就是在Test Case File和Resource File中的Variable Table中。
在Variable Table中创建变量有诸多好处： 和其他测试数据在同一个地方，语法也非常简单； 不足在于变量值可能只能为字符串，并且不能动态创建。

```
Tips: 如果需要克服此问题，可以考虑Variable File
```

创建**scalar**变量

样例*1*
如果第2列为空，那么空字符串被赋值给变量

```
*** Variables ***
${NAME}         Robot Framework
${VERSION}      2.0
${ROBOT}        ${NAME} ${VERSION}
${ZERO}
```

样例*2*
也支持中间添加"="的语法，但这不是强制要求的

```
*** Variables ***
${NAME} =       Robot Framework
${VERSION} =    2.0
```

样例*3*
如果值太长，可以分为多行和多列
多行和多列会被框架合并起来
缺省的，合并中间会使用空格(等效于" ".join())，也可以通过在第一个cell中使用SEPARATOR=来改变连接字符

```
*** Variables ***
${EXAMPLE}      This value is joined    together with a space
${MULTILINE}    SEPARATOR=\n    First line
...             Second line     Third line
```

创建**list**变量

样例*1*
值从第2列开始
也支持空值，list作为元素，多行等语法
列表的下标从0开始

```
*** Variables ***
@{NAMES}        Matti       Teppo
@{NAMES2}       @{NAMES}    Seppo
@{NOTHING}
@{MANY}         one         two      three      four
...             five        six      seven
```

创建**dict**变量

创建dict变量的语法: "name=value" 或者 已有的Dict变量赋值给新变量
同样key的value，后出现的覆盖先出现的
如果key或value中存在"="，需要进行转义"\="

样例
访问元素的语法(使用Python的同学一定不会陌生): ${VAR.key} 或者 &{USER}[name]
&{MANY}[${3}] 不等价于 ${MANY.3}
字典的key是有顺序的，按照其被定义的顺序
字典按照列表的语法进行使用，实际使用的是字典的key集合: @{MANY}变量 == ['first', 'second', 3]

```
*** Variables ***
&{USER 1}       name=Matti    address=xxx         phone=123
&{USER 2}       name=Teppo    address=yyy         phone=456
&{MANY}         first=1       second=${2}         ${3}=third
&{EVEN MORE}    &{MANY}       first=override      empty=
...             =empty        key\=here=value
```

变量文件**, Vairable Files**

变量文件支持各种变量的创建: 任何对象被指派给变量; 动态创建变量等
变量文件典型使用Python模块来实现
更多关于变量文件的介绍，请参考下一个章节。

命令行**, Command Line Option**

命令行选项可以设置变量:

- 单个设置变量: --variable (-v)
- 使用Variable File: --variablefile (-V)

  Tips: --variable (-v) 优先于 --variablefile (-V)

通过命令行设置的变量，对于所有测试执行是全局可用的。 它们会覆盖通过Variable Table创建或者导入的Variable File创建的同名变量。

样例*1*
语法： --variable name:value
只能设置scalar变量，值只能为字符串
可以使用--escape对字符进行转义

以下样例的结果如下:
${EXAMPLE} 值: value
${HOST} 和 ${USER} 值分别为: localhost:7272 和 robot
${ESCAPED} 值: "quotes and spaces"

```
--variable EXAMPLE:value
--variable HOST:localhost:7272 --variable USER:robot
--variable ESCAPED:Qquotes_and_spacesQ --escape quot:Q --escape space:_
```

返回值, **Return Value from Keyword**

关键字返回值可以被赋值给变量。这个变量又可以被传递给其他关键字。 这样实现不同关键字之间的相互通信。
这种变量的设置和使用，和其他方式创建的变量是一样的。只是这种变量的作用域范围只局限在它们被创建的局部作用域。即在一个Test Case里创建的变量，不能在另一个Test Case中被使用。

scalar变量的赋值

样例*1*
"="不是强制要求的。
这种创建变量的方式，在定义用户关键字时也是一样的。

```
*** Test Cases ***
Returning
    ${x} =    Get X    an argument
    Log    We got ${x}!
```

样例*2*
尽管被赋值给scalar变量，但是，如果数据为list-like的，那么你可以将其按照list变量进行使用。
同样地，如果数据为dict-like的，也可以将其按照dict变量进行使用。

```
*** Test Cases ***
Example
    ${list} =    Create List    first    second    third
    Length Should Be    ${list}    3
    Log Many    @{list}
```

list变量的赋值

样例
如果关键字返回值为list或者list-like的值，可以将其赋值给list变量。

```
*** Test Cases ***
Example
    @{list} =    Create List    first    second    third
    Length Should Be    ${list}    3
    Log Many    @{list}
```

```
因为所有的变量都被存储在同一个名字空间(namespace)，值被赋值给scalar变量或者list变量实际上是没有太大区别。
区别在于:
当创建一个list变量时，RF框架会验证值是否为list或list-like的，存储的值将会是从返回值创建的新的list对象。
当创建一个scalar变量时，RF框架不会进行任何验证，返回的值将会原样被存储起来。
```

dict变量的赋值

样例
如果关键字返回值为dict或者dict-like的值，可以将其赋值给dict变量。

```
*** Test Cases ***
Example
    &{dict} =    Create Dictionary    first=1    second=${2}    ${3}=third
    Length Should Be    ${dict}    3
    Do Something    &{dict}
    Log    ${dict.first}
```

因为所有的变量都被存储在同一个名字空间(namespace)，值被赋值给scalar变量，之后又按照dict变量来进行使用，这样做也是可以的。
但赋值给dict变量有这些好处：
RF框架会像验证list变量一样，执行dict变量的相应验证。
RF框架会将值转换为特殊的dict：可排序的；可以使用语法${dict.first}访问单个元素。

多个变量的同时赋值

如果关键字返回值为list或list-like的对象，可以将单个值赋值给不同的scalar变量，或者scalar变量+list变量。

样例
假设关键字"Get Three"返回[1, 2, 3], 创建的变量和值如下情况:
${a}, ${b}, ${c} 值分别为: 1, 2, 3
${first} 值: 1, @{rest} 值: [2, 3]
@{before} 值: [1, 2], ${last} 值: 3
${begin} 值: 1, @{middle} 值: [2], ${end} 值: 3

```
*** Test Cases ***
Assign Multiple
    ${a}    ${b}    ${c} =    Get Three
    ${first}    @{rest} =    Get Three
    @{before}    ${last} =    Get Three
    ${begin}    @{middle}    ${end} =    Get Three
```

内建变量设置关键字, Set Test/Suite/Global Variable

内建变量设置关键字，可以在用例执行期间动态设置关键字
如果变量已经存在，值将会被覆盖;
如果变量不存在，新变量会被创建

- Set Test Variable
  作用域: 当前测试用例中
- Set Suite Variable
  作用域: 和测试文件中Variable Table或者从Variable File导入等效;
  其他Suite, 包括子Suite, 不可见
- Set Global Variable
  作用域: 设置后的所有用例和Suite都可见;
  和--variable 或 --variablefile等效

  关键字"Set Xxx Variable"直接设置变量到相应的作用域，没有返回值
  关键字"Set Variable"使用返回值设置本地变量

变量文件, Vairable Files

作者: 虞科敏

变量文件提供了强大的变量创建和共享的机制。 它支持各种变量的创建: 任何对象被指派给变量; 动态创建变量等
变量文件强大的功能，是因为典型地，它是使用Python模块(或者Python类，Java类)来实现的

在Variable File创建变量的2种方法

- 直接创建变量

模块的属性，直接成为变量。比如在模块中定义

```
MY_VAR = 'my value'
```

创建${MY_VAR}， 值为'my value'

- 使用特殊函数

特殊的获取变量函数, 返回作为字典的变量。方法可以带有参数，此机制创建变量非常灵活。

```
get_variables
getVariables
```

另外， 除了Python模块，也可以使用Python类或者Java类实现， 框架会实例化这些类作为变量。 创建这种对象实例的变量，方法也和上面创建变量的2种方法一致。

导入和使用Varialbe File

在Setting中导入

Variable File的导入和Resource File的导入相似
Path先以相对要求导入的文件所在目录的相对路径进行； 如果没有找到，会在Python的模块搜索路径中查找
导入的路径和参数，都支持使用变量

样例

```
*** Settings ***
Variables    myvariables.py
Variables    ../data/variables.py
Variables    ${RESOURCES}/common.py
Variables    taking_arguments.py    arg1    ${ARG2}
```

```
Tips:
导入的变量在执行导入操作的测试文件中有效
如果多个文件导入存在重名变量情况，最早导入的变量有效
在Variable Table或者命令行选项创建的变量，可能覆盖Variable File创建的变量
```

通过命令行, Command Line导入

命令行选线 --variablefile 也可以使用Variable File
从命令行导入的Varialbe File，作用域范围是全局可用的。这个通过--varialbe选项设置的变量情况是相似的。 如果通过--variablefile 和 --varialbe 创建的变量名存在冲突，--variable选项创建的变量将会被保留。

样例
文件通过path被引用
如果需要参数，使用":"将参数添加在path后面; 也可以使用";"分隔path和参数

```
--variablefile myvariables.py
--variablefile path/variables.py
--variablefile /absolute/path/common.py
--variablefile taking_arguments.py:arg1:arg2
--variablefile "myvariables.py;argument:with:colons"
--variablefile C:\path\variables.py;D:\data.xls
```

```
Tips
从命令行导入与从Setting中导入Variable File，路径搜索的规则是一致的
```

直接创建变量

如果熟悉Python的同学，对于Variable的导入可能不难理解。 Varialbe File就是以导入Python模块的机制在进行导入。 Variable File导入时候，模块中除了以"_"开始的属性，其他所有全局属性都会被导入作为变量。

```
Tips
变量名是大小写敏感的
建议：全局变量请使用全大写名称
```

样例*1*
在此定义的所有变量都可以被当做scalar变量来进行使用 @{STRINGS}为list变量， $STRINGS为一个list
&{MAPPIN}为Gdict变量， $MAPPING为一个包含dict所有key的list

```
VARIABLE = "An example string"
ANOTHER_VARIABLE = "This is pretty easy!"
INTEGER = 42
STRINGS = ["one", "two", "kolme", "four"]
NUMBERS = [1, INTEGER, 3.14]
MAPPING = {"one": 1, "two": 2, "three": 3}
```

样例*2*
为了更明确的定义list变量或者dict变量，可以使用前缀"LIST__" 或者 "DICT__"。 前缀不会不会作为变量名的一部分，它们的作用是告诉RF框架，变量将会是"list-like"或者"dict-like"类型的，框架会执行相应的验证检查。
访问dict变量的值: ${FINNISH.cat}

```
from collections import OrderedDict

LIST\_\_ANIMALS = ["cat", "dog"]
DICT\_\_FINNISH = OrderedDict([("cat", "kissa"), ("dog", "koira")])
```

样例*3*
样例1和样例2创建的变量，可以采用样例3的Variable Table来创建。

```
*** Variables ***
${VARIABLE}           An example string
${ANOTHER VARIABLE}   This is pretty easy!
${INTEGER}            ${42}
@{STRINGS}            one          two          kolme        four
@{NUMBERS}            ${1}         ${INTEGER}   ${3.14}
&{MAPPING}            one=${1}     two=${2}     three=${3}
@{ANIMALS}            cat          dog
&{FINNISH}            cat=kissa    dog=koira
```

使用对象实例作为值

在Variable Table创建变量，变量值局限于字符串或其他基本类型
在Variable File中创建变量，没有这个局限

样例*1*
${MAPPING}的值为Hashtable对象，其中有2个值。

```
from java.util import Hashtable

MAPPING = Hashtable()
MAPPING.put("one", 1)
MAPPING.put("two", 2)
```

样例*2* 创建了和样例1相似的${MAPPING}，本样例中为python字典实例。
另外, ${OBJ1}和${OBJ2}，值为Variable File中定义的MyObject类的实例。

```
MAPPING = {'one': 1, 'two': 2}

class MyObject:
    def __init__(self, name):
        self.name = name

OBJ1 = MyObject('John')
OBJ2 = MyObject('Jane')
```

动态创建变量

由于Variable File本质上是编程语言进行变量的创建，所以可以达到动态创建变量的效果。

样例*1*
使用python库中随机函数，每次设置不同的值

```
import os
import random
import time

USER = os.getlogin()                # current login name
RANDOM_INT = random.randint(0, 10)  # random integer in range [0,10]
CURRENT_TIME = time.asctime()       # timestamp like 'Thu Apr  6 12:45:21 2006'
if time.localtime()[3] > 12:
    AFTERNOON = True
else:
    AFTERNOON = False
```

样例*2*
可以使用外部数据源(比如数据库，文件，甚至询问用户)来设置变量值

```
import math

def get_area(diameter):
    radius = diameter / 2
    area = math.pi * radius * radius
    return area

AREA1 = get_area(1)
AREA2 = get_area(2)
```

选择导入的变量

为了避免不被需要的变量被导入，有如下手段。 其实熟悉Python的同学可以看出来，都是Python的菜啊。

样例*1*
以"_"开头的属性将不会被导入

```
import math

def _get_area(diameter):
    radius = diameter / 2
    area = math.pi * radius * radius
    return area

AREA1 = _get_area(1)
AREA2 = _get_area(2)
```

样例*2*
熟悉Python的同学，一定不会陌生，这和__init__.py 中加入 __all__ 变量的语法很像. 该变量包含执行"from-import all"语句时应该导入的模块名字. 它由一个模块名字符串列表组成.

```
import math

__all__ = ['AREA1', 'AREA2']

def get_area(diameter):
    radius = diameter / 2.0
    area = math.pi * radius * radius
    return area

AREA1 = get_area(1)
AREA2 = get_area(2)
```

特殊函数

如果在Variable File中存在特殊函数(get*variables*或者*getVariables)*，可以通过特殊函数来得到变量。
如果特殊函数存在，*RF*框架会调用此函数，来获取函数。 函数的预期返回应该是*Python*的*dict* 或者 *Java*的*Map*， 其中*key*为*Variable*名称， *value*为*Variable*值。 其他的规则，和直接创建变量的情形一样*:* 可以创建*scalar, list, dict*各种类型变量*;* 也支持特殊前缀*"LIST*_" 和 "DICT__"，等。

样例*1*

```
def get_variables():
    variables = {"VARIABLE ": "An example string",
                 "ANOTHER VARIABLE": "This is pretty easy!",
                 "INTEGER": 42,
                 "STRINGS": ["one", "two", "kolme", "four"],
                 "NUMBERS": [1, 42, 3.14],
                 "MAPPING": {"one": 1, "two": 2, "three": 3}}
    return variables
```

样例*2*
特殊函数也支持参数
在测试数据中，参数在path cell后面的cells中给出; 在命令行中，参数通过":"或者";"给出

```
variables1 = {'scalar': 'Scalar variable',
    'LIST\_\_list': ['List','variable']}
variables2 = {'scalar' : 'Some other value',
              'LIST\_\_list': ['Some','other','value'],
              'extra': 'variables1 does not have this at all'}

def get_variables(arg):
    if arg == 'one':
        return variables1
    else:
        return variables2
```

使用**Python**类或者**Java**类来实现**Variable File**

由于Variable File导入需要使用文件系统的path进行，使用类来实现Varialbe File有一些限制:

- Python类必须和它所在的module同名
- Java类必须存在于缺省包
- Java类的路径必须以.java或.class结尾，类文件必须存在

无论使用哪种语言实现，框架都会使用无参构造函数创建类实例，通过类实例得到变量。
和直接使用模块一样，变量可以在实例中直接定义，也可以通过特殊方法(get_variables或者getVariables)得到。

如果变量在类实例中直接被定义，为了避免实例方法用来创建变量，全部可调用的属性将会被忽略。

实例*1*
2个版本都创建了如下变量:
来自类属性的 ${VARIABLE} 和 @{LIST}
来自实例属性的 ${anotherVariable}

Python版本

```
class StaticPythonExample(object):
    variable = 'value'
    LIST__list = [1, 2, 3]
    _not_variable = 'starts with an underscore'

    def __init__(self):
        self.another_variable = 'another value'
```

Java版本

```
public class StaticJavaExample {
    public static String variable = "value";
    public static String[] LIST__list = {1, 2, 3};
    private String notVariable = "is private";
    public String anotherVariable;

    public StaticJavaExample() {
        anotherVariable = "another value";
    }
}
```

实例*2*
2个版本都使用动态方法创建了变量${dynamic variable}

Python版本

```python
class DynamicPythonExample(object):

    def get_variables(self, *args):
        return {'dynamic variable': ' '.join(args)}
```

Java版本

```java
import java.util.Map;
import java.util.HashMap;

public class DynamicJavaExample {

    public Map<String, String> getVariables(String arg1, String arg2) {
        HashMap<String, String> variables = new HashMap<String, String>();
        variables.put("dynamic variable", arg1 + " " + arg2);
        return variables;
    }
}
```

内建变量, Build-in Variables

作者: 虞科敏

OS变量

${CURDIR} - Test Data File所在目录的绝对路径
${TEMPDIR} - 系统临时目录的绝对路径(Linux "/tmp"; Window "c:\Documents and Settings\\Local Settings\Temp")
${EXECDIR} - 测试执行开始目录的绝对路径
${/} - 操作系统目录路径分隔符(Linux "/"; Window "\")
${:} - 操作系统路径环境变量的分隔符(Linux ":"; Window ";")
${\n} - 操作系统文件行分隔符(Linux "\n"; Window "\r\n")

样例

```
*** Test Cases ***
Example
    Create Binary File    ${CURDIR}${/}input.data    Some text here${\n}on two lines
    Set Environment Variable    CLASSPATH    ${TEMPDIR}${:}${CURDIR}${/}foo.jar
```

数字变量

数字变量语法，用来创建整数或者浮点数。
当关键字希望得到一个实际的数字，而不是字符串时，应该使用数字变量语法。

样例*1*

```
*** Test Cases ***
Example 1A
    Connect    example.com    80       # Connect gets two strings as arguments

Example 1B
    Connect    example.com    ${80}    # Connect gets a string and an integer

Example 2
    Do X    ${3.14}    ${-1e-4}        # Do X gets floating point numbers 3.14 and -0.0001
```

样例*2*
数字进制的前缀(注: 本语法不是大小写敏感的)
0b或者0B - 2进制
0o或者0O - 8进制
0x或者0X - 16进制

```
*** Test Cases ***
Example
    Should Be Equal    ${0b1011}    ${11}
    Should Be Equal    ${0o10}      ${8}
    Should Be Equal    ${0xff}      ${255}
    Should Be Equal    ${0B1010}    ${0XA}
```

布尔变量

样例
布尔变量不是大小写敏感的: ${true} == ${True}, ${false} == ${False}

```
*** Test Cases ***
Boolean
    Set Status    ${true}              # Set Status gets Boolean true as an argument
    Create Y    something   ${false}    # Create Y gets a string and Boolean false
```

空值变量

样例
空值变量不是大小写敏感的, Null和None同义: ${null} == ${Null} == ${none} == ${None}

```
*** Test Cases ***
None
    Do XYZ    ${None}                  # Do XYZ gets Python None as an argument

Null
    ${ret} =    Get Value    arg        # Checking that Get Value returns Java null
    Should Be Equal    ${ret}    ${null}
```

空格和空字符串变量

空格变量: ${SPACE} == " "
多个空格语法: ${SPACE * 2} == " \ "

空字符串变量: ${EMPTY} = \

样例*1*
空格和空字符串样例

```
*** Test Cases ***
One Space
    Should Be Equal    ${SPACE}          \ \

Four Spaces
    Should Be Equal    ${SPACE * 4}      \ \ \ \ \

Ten Spaces
    Should Be Equal    ${SPACE * 10}     \ \ \ \ \ \ \ \ \ \ \

Quoted Space
    Should Be Equal    "${SPACE}"        " "

Quoted Spaces
    Should Be Equal    "${SPACE * 2}"    " \ "

Empty
    Should Be Equal    ${EMPTY}          \
```

样例*2*
空list和空dict样例

```
*** Test Cases ***
Template
    [Template]    Some keyword
    @{EMPTY}

Override
    Set Global Variable    @{LIST}    @{EMPTY}
    Set Suite Variable     &{DICT}    &{EMPTY}
```

自动变量

自动变量被RF框架创建和修改，在测试执行过程中值可能会变量; 另外，某些自动变量在执行过程中并非总是可用。
修改自动变量，并不能对变量的初始值产生影响。
但可以用某些内建关键字来修改某些自动变量的值。

变量名	含义	可用范围
${TEST NAME}	当前测试用例的名字	Test case
@{TEST TAGS}	当前测试用例的标签(按字母序)。可以使用"Set Tags"和"Remove Tags"关键字修改	Test case
${TEST DOCUMENTATION}	当前测试用例的文档说明。可以使用"Set Test Documentation"关键字修改	Test case
${TEST STATUS}	当前测试用例的状态: Pass或FAIL	Test teardown
${TEST MESSAGE}	当前测试用例的消息	Test teardown
${PREV TEST NAME}	前一个测试用例的名字。如果还没有用例被执行，值为空字符串	Everywhere
${PREV TEST STATUS}	前一个测试用例的状态: Pass或FAIL。如果还没有用例被执行，值为空字符串	Everywhere
${PREV TEST MESSAGE}	前一个测试用例的错误消息	Everywhere
${SUITE NAME}	当前Suite的全名	Everywhere
${SUITE SOURCE}	Suite的文件或目录的绝对路径	Everywhere
${SUITE DOCUMENTATION}	当前测试Suite的文档说明。可以使用"Set Suite Documentation"关键字修改	Everywhere
&{SUITE METADATA}	当前测试Suite的元数据。可以使用"Set Suite Metadata"关键字修改	Everywhere
${SUITE STATUS}	当前测试Suite的状态: Pass或FAIL	teardown
${SUITE MESSAGE}	当前测试Suite的消息, 包括统计信息	Suite teardown
${KEYWORD STATUS}	当前测试关键字的状态: Pass或FAIL	User keyword teardown
${KEYWORD MESSAGE}	当前测试关键字的错误消息	User keyword teardown
${LOG LEVEL}	当前的日志级别	Everywhere
${OUTPUT FILE}	输出(output)文件的绝对路径	Everywhere
${LOG FILE}	日志(log)文件的绝对路径。如果没有日志文件，值为空字符串	Everywhere
${REPORT FILE}	报告(report)文件的绝对路径。如果没有报告文件，值为空字符串	Everywhere
${DEBUG FILE}	调试(debug)文件的绝对路径。如果没有调试文件，值为空字符串	Everywhere
${OUTPUT DIR}	输出(output)文件所在目录的绝对路径	Everywhere

```
Tips:
${SUITE SOURCE}, ${SUITE NAME}, ${SUITE DOCUMENTATION}, &{SUITE METADATA}在libraries和variable files被导入时已可用
```

变量的属性和作用范围

作者: 虞科敏

属性, **Properties**

命令行设置的变量

- 命令行设置的变量拥有在测试开始执行前，所有能设置变量的最高优先级: 可以覆盖Test Case File中
- Variable Table, 导入的Resource File和Variable File的相关设置
- --variable (-v) 优先于 --variablefile (-V)
- 如果设置同一个变量多次，后面的设置会覆盖前面的设置
- 可以在启动脚本中设置变量，这种情况会覆盖命令行选项的设置同名变量
- 如果在多个变量文件中设置同名变量，第一次设置的变量具有最高优先级(先导入优先原则)

在**Test Case File**的**Vaiable Table**中设置的变量

- 对于本文件中的所有Test Case可用
- 会覆盖文件中通过导入Reousrce File和Variable File创建的同名变量
- 本文件中其他的表(如Setting Table可以使用本文件中Vairable Table中创建的变量)

通过导入**Resource File**和**Variable File**设置的变量

- 在Test Data中设置的变量中，优先级最低
- 来自Resource File的变量 和 来自Variable File的变量，优先级相同
- 对于来自Resource File和Variable File的同名变量，采用"先导入优先原则"
- 如果一个Resource File导入其他Rsource File和Variable File, 采用"本地优先于导入"的原则
- 从Resource File和Variable File导入的变量，对于导入它们的文件中的Variable Table不可用

  Tips: Variable Table的处理先于Setting Table的处理(Resource File和Variable File在Setting Table中被导入)

在执行过程中设置的变量

在执行过程中设置的变量(通过关键字返回值设置，或者使用内建变量设置关键字)会覆盖其作用域内的同名变量
但其不会影响作用域外的变量

内建变量

内建变量拥有最高优先级，不能被Vairable Table或者命令行所覆盖； 但它们可能被框架重置
数字变量是个例外: 当变量不能被找到时，它们会被动态解析； 这样它们可能被覆盖，但不推荐这样做！

作用范围, **Scope**

全局范围

以下变量具有全局范围:

```
通过命令行选项创建的变量
内建关键字"Set Global Variable"设置(和修改)全局变量
内建变量
```

全局范围变量，建议使用全大写命名

Test Suite范围

以下变量具有Test Suite范围:

```
Varaible Table创建的变量
通过Resource File和Variable File导入的变量
内建关键字"Set Suite Variable"设置(和修改)的变量
```

Test Suite范围变量，建议使用全大写命名

Test Case范围

以下变量具有Test Case范围:

```
内建关键字"Set Case Variable"设置(和修改)的变量
```

Test Case范围变量，建议使用全大写命名

本地(Local)范围

以下变量具有Local范围:

```
通过扩展关键字的返回值创建的变量，用户自定义关键字使用其作为参数
```

Local范围变量，建议使用全小写命名

变量的高级特性

作者: 虞科敏

暂缺

用户关键字, User Keyword

作者: 虞科敏

基础语法

在Keyword Table中， 组合使用已有的关键字，创建更高一级的关键字，被称为用户关键字。
区别于在测试库中实现的低级库关键字。
创建用户关键字的语法，和创建Test Case很相似。

用户关键字可以在以下位置被创建:

- Test Case File
- Resource File
- Suite的__init__ File

样例
使用底层库关键字，或者其他用户关键字，定义新的用户关键字
通常，关键字名位于第2个cell中； 对于返回值赋值语法，关键字名位于返回值变量后1个cell(第3个cell）中
用户关键字可以带参数，也可以不带参数

```
*** Keywords ***
Open Login Page
    Open Browser    http://host/login.html
    Title Should Be    Login Page

Title Should Start With
    [Arguments]    ${expected}
    ${title} =    Get Title
    Should Start With    ${title}    ${expected}
```

Keywork Table的设置项(Setting)

样例

```
[Documentation]
Used for setting a user keyword documentation.
[Tags]
Sets tags for the keyword.
[Arguments]
Specifies user keyword arguments.
[Return]
Specifies user keyword return values.
[Teardown]
Specify user keyword teardown.
[Timeout]
Sets the possible user keyword timeout. Timeouts are discussed in a section of their own.
```

关键字名

用户关键字的名称，在定义的第一行中指定
关键字名可以很长，以描述关键字主要功能为目标

文档串

关键字文档串，和用例文档串很相似
文档工具Libdoc能从Resource File中提取出正式的关键字文档，文档串会作为重要的说明性信息
文档串的第一行也会出现在测试日志中
以 "*DEPRECATED*" 作为文档串的开头，可以标记关键字过期。

标签

关键字标签使用 "[Tags]" 进行设置，这和用例标签很相似。
设置(Setting)中的Force Tags和Default Tags不会影响关键字的标签。

文档工具Libdoc提取出的正式关键字文档中，关键字标签也会被显示。
命令行选项 --removekeywords 和 --flattenkeywords 支持根据标签选择关键字。
和用例标签一样，RF框架预留标签前缀 **"robot-"** 作为特殊特性的用途。除非用户希望激活这些特性，否则请不要自己定义以此为前缀的标签。

样例
在文档串的最后一行，可以指定标签。
样例中定义的2个用户关键字，标签是一样的，都是3个标签: my, fine, tags。

```
*** Keywords ***
Settings tags using separate setting
    [Tags]    my    fine    tags
    No Operation

Settings tags using documentation
    [Documentation]    I have documentation. And my documentation has tags.
    ...                Tags: my, fine, tags
    No Operation
```

用户关键字参数

作者: 虞科敏

参数使用 "[Arguments]" 进行设置， 参数名设置使用变量的语法，如 ${arg}
参数命名和框架无关，应该能对参数作用具有很好的描述性
参数建议使用小写形式，如 ${my_arg}, ${my arg}, ${myArg}

位置参数

位置参数的格式在调用时候的数量，必须和关键字签名的参数数量相同

样例

```
*** Keywords ***
One Argument
    [Arguments]    ${arg_name}
    Log    Got argument ${arg_name}

Three Arguments
    [Arguments]    ${arg1}    ${arg2}    ${arg3}
    Log    1st argument: ${arg1}
    Log    2nd argument: ${arg2}
    Log    3rd argument: ${arg3}
```

缺省值

缺省值语法: ${arg}=default
缺省值中可以包含test范围，suite范围，global范围的变量，但不可以包含关键字local范围的变量

```
Tips:
缺省值语法是空格敏感的，在"="前不允许空格；在"="后的空格被认为是值的一部分
```

样例
如果几个带有缺省值的参数，只有部分参数值需要被重置，可以使用命名参数语法: arg=newvalue
命名参数语法中，参数名不要使用${}进行修饰; 而值可以是字符串，也可以是变量。

在下面的测试用例中， arg2=new value即为命名参数语法.
关键字"Two Arguments With Defaults"的arg1参数值为缺省值， arg2参数的值为"new value "

```
*** Test Cases ***
Example
    Two Arguments With Defaults    arg2=new value

*** Keywords ***
One Argument With Default Value
    [Arguments]    ${arg}=default value
    [Documentation]    This keyword takes 0-1 arguments
    Log    Got argument ${arg}

Two Arguments With Defaults
    [Arguments]    ${arg1}=default 1    ${arg2}=${VARIABLE}
    [Documentation]    This keyword takes 0-2 arguments
    Log    1st argument ${arg1}
    Log    2nd argument ${arg2}

One Required And One With Default
    [Arguments]    ${required}    ${optional}=default
    [Documentation]    This keyword takes 1-2 arguments
    Log    Required: ${required}
    Log    Optional: ${optional}

 Default Based On Earlier Argument
    [Arguments]    ${a}    ${b}=${a}    ${c}=${a} and ${b}
    Should Be Equal    ${a}    ${b}
    Should Be Equal    ${c}    ${a} and ${b}
```

可变长参数

在关键字签名中，位置变量后面，跟上list变量 @{varargs} 作为可变长参数 也可以和缺省值语法混合使用，由后面的list变量接收所有未被匹配认领的位置参数(非命名参数)
list变量可以接受 0 到 多个参数值
以上这些用法， @{varargs} 其实和python里面的非关键字可变长参数(tuple, *args)和其他参数之间的交互是一致的

样例
对于关键字"Required, Default, Varargs":
如果超过1个参数值被指定，参数${opt}将总是使用指定值，而不会使用缺省值。即时给定值为${EMPTY}也会赋值给${opt}参数。
另外，请注意list参数如何在:FOR语法中被使用。

```
*** Keywords ***
Any Number Of Arguments
    [Arguments]    @{varargs}
    Log Many    @{varargs}

One Or More Arguments
    [Arguments]    ${required}    @{rest}
    Log Many    ${required}    @{rest}

Required, Default, Varargs
    [Arguments]    ${req}    ${opt}=42    @{others}
    Log    Required: ${req}
    Log    Optional: ${opt}
    Log    Others:
    : FOR    ${item}    IN    @{others}
    \    Log    ${item}
```

自由参数

在关键字签名中，位置变量和可变长变量后面，跟上dict变量 &{kwargs} 作为自由参数
自由参数应该是关键字签名中的最后一个参数
最后的dict变量，将会接收所有未被位置参数匹配的命名参数
以上这些用法，&{kwargs} 其实和python里面的关键字可变长参数(dict, **kwargs)和其他参数之间的交互是一致的。

样例

```
*** Keywords ***
Kwargs Only
    [Arguments]    &{kwargs}
    Log    ${kwargs}
    Log Many    @{kwargs}

Positional And Kwargs
    [Arguments]    ${required}    &{extra}
    Log Many    ${required}    @{extra}

Run Program
    [Arguments]    @{varargs}    &{kwargs}
    Run Process    program.py    @{varargs}    &{kwargs}
```

变量嵌入到关键字名中

作者: 虞科敏

除了常规的方法: 参数在关键字名后面的cell中指定，也可以将变量嵌入到关键字名之中。
这样做最大的好处是，真正意义上的并且简洁的语句，可以作为关键字名。

基础语法

嵌入变量语法，在关键字定义时，不在明确的使用[Arguments]指定参数。
在变量名中使用的参数，在关键字内部被解析出来使用，它们的值依赖于如何调用关键字的。

变量嵌入语法的关键字，使用和其他关键字区别不大。 除了:
在关键字名中，空格和"_"不会被忽略掉; 关键字名不是大小写敏感的

变量嵌入语法的关键字，不支持参数缺省值 和 可变长参数
变量嵌入语法，只支持用户关键字

样例*1*
理想的变量嵌入，如下的关键字，我们可以定义一个关键字，可以执行"Select cat from list"或者"Select dog from list"等关键字。
"Select cat from list"， 那么${anmial}=cat
"Select dog from list"， 那么${anmial}=dog

嵌入参数的关键字名非大小写敏感， select x from list == Select x fromlist

```
*** Keywords ***
Select ${animal} from list
    Open Page    Pet Selection
    Select Item From List    animal_list    ${animal}
```

嵌入参数匹配过多

确保调用关键字时候，使用的参数匹配正确的值，是最具挑战和需要一定技巧的。
"Select ${city} ${team}" 如果使用 "Select Los Angeles Lakers" 调用， 就不会工作得很好。

一种解决方法是使用引号: Select "${city}" "${team}" => Select "Los Angeles" "Lakers"

另一个更强大，但也更复杂的方法是使用 定制正则表达式
请参考后面章节

如果情况变得复杂，难于管理，使用普通的位置参数可能才是一个更好的选择。

用户正则表达式, **custom regular expression**

嵌入参数语法进行值匹配的内部机制，其实是 正则表达式匹配。
缺省的，每个参数位置是使用正则表达式".*"来进行替换，这个正则表达式会匹配所有的字符串。

使用正则表达式的语法: ${arg:regexp}
比如，参数值只能匹配数字，可以写为: ${arg:\d+}

样例*1*
使用引号"，有时候可以帮助解决匹配过多的问题。
但是下面定义的关键字，I execute "ls" with "-lh" 既可以匹配第1个用户关键字，也可以匹配第2个用户关键字。 匹配有歧义，会导致测试失败。

```
*** Test Cases ***
Example
    I execute "ls"
    I execute "ls" with "-lh"

*** Keywords ***
I execute "${cmd}"
    Run Process    ${cmd}    shell=True

I execute "${cmd}" with "${opts}"
    Run Process    ${cmd} ${opts}    shell=True
```

样例2

"${cmd:''+}" - cmd参数匹配非"的字符

${a:\d+} ${operator:[+-]} ${b:\d+} - a和b参数匹配数字，operator匹配+或-

${date:\d{4}-\d{2}-\d{2}} - date参数匹配 4数字-2数字-2数字

```
*** Test Cases ***
Example
    I execute "ls"
    I execute "ls" with "-lh"
    I type 1 + 2
    I type 53 - 11
    Today is 2011-06-27

*** Keywords ***
I execute "${cmd:[^"]+}"
    Run Process    ${cmd}    shell=True

I execute "${cmd}" with "${opts}"
    Run Process    ${cmd} ${opts}    shell=True

I type ${a:\d+} ${operator:[+-]} ${b:\d+}
    Calculate    ${a}    ${operator}    ${b}

Today is ${date:\d{4\}-\d{2\}-\d{2\}}
    Log    ${date}
```

支持的正则表达式语法

由于RF框架使用Python实现，Python的 re模块 支持标准的正则表达式语法，参数的正则表达式语法也继承了这些语法能力

正则表达式的 (?...) 扩展，在参数正则表达式语法中不被采用

如果正则表达式语法不合法，会导致相应的关键字失败

```
(pattern) 匹配pattern，并获取这一匹配
(?:pattern) 匹配pattern，但是不获取匹配结果，即为非获取匹配，不进行存储供以后使用
(?=pattern) 正向预查，在任何匹配 pattern 的字符串开始处匹配查找字符串，也为非获取匹配；  预查不消耗字符
(?!pattern) 反向预查，在任何不匹配 pattern 的字符串开始处匹配查找字符串，也为非获取匹配；  预查不消耗字符
```

特殊字符的转义

} => \} - 在正则表达式中定义出现次数需要{}语法，所以在正则表达式中出现，需要转义

\ => \\ , 更保险的是4个(即 \\\\) - \在Python正则表达式中有特殊含义，如果需要字面上的\字符，需要转义

在嵌入参数正则表达式使用变量

除了文本的匹配，RF框架会自动加强匹配能力，去匹配变量值

总是可以将变量 和 带正则表达式的嵌入参数关键字 一起使用

样例
以下测试用例Example的关键字，会匹配之前定义的关键字 I type ${a:\d+} ${operator:[+-]} ${b:\d+} 和 Today is ${date:\d{4}-\d{2}-\d{2}}

```
*** Variables ***
${DATE}    2011-06-27

*** Test Cases ***
Example
    I type ${1} + ${2}
    Today is ${DATE}
```

行为驱动开发范例, BDD Example

当进行BDD风格测试用例时，希望采用高级的语句风格的关键字

样例
如前所述，Given-When-Then-And-But 可以是，也可以不是关键字名的一部分

```
*** Test Cases ***
Add two numbers
    Given I have Calculator open
    When I add 2 and 40
    Then result should be 42

Add negative numbers
    Given I have Calculator open
    When I add 1 and -2
    Then result should be -1

*** Keywords ***
I have ${program} open
    Start Program    ${program}

I add ${number 1} and ${number 2}
    Input Number    ${number 1}
    Push Button     +
    Input Number    ${number 2}
    Push Button     =

Result should be ${expected}
    ${result} =    Get Result
    Should Be Equal    ${result}    ${expected}
```

```
Tips: RF中BDD编程风格的特性是受到Cucumber框架的启发添加的. 我的下一个开源书计划正好是Python-Cucumber: Lettuce :)
```

用户关键字返回值

作者: 虞科敏

和库关键字一样，用户关键字也可以带有返回值
最典型的返回值定义，是使用[Return]进行设置
也可以使用内建关键字: Return From Keyword 或者 Return From Keyword If
不管采用哪种方式返回值，在测试用例中，返回值都可以被赋值给变量，然后传递给其他关键字

[Return]

语法: [Return] cell 后紧跟 被返回值cell

关键字可以返回多个值，语法: [Return] cell 后紧跟 多个值的cells

这些值可以被赋值给:
一次性赋值给多个scalar变量
1个list变量
多个scalar变量+1个list变量

样例

```
*** Test Cases ***
One Return Value
    ${ret} =    Return One Value    argument
    Some Keyword    ${ret}

Multiple Values
    ${a}    ${b}    ${c} =    Return Three Values
    @{list} =    Return Three Values
    ${scalar}    @{rest} =    Return Three Values

*** Keywords ***
Return One Value
    [Arguments]    ${arg}
    Do Something    ${arg}
    ${value} =    Get Some Value
    [Return]    ${value}

Return Three Values
    [Return]    foo    bar    zap
```

使用特殊关键字返回

特殊关键字: Return From Keyword 或者 Return From Keyword If
可以在关键字中间有条件地返回值
这些关键字可以带可选的返回值，返回值的处理行为和[Return]一样

样例
关键字"Return One Value"展示和[Return]一样的功能
关键字"Find Index"展示在:For循环中有条件返回值的功能

```
*** Test Cases ***
One Return Value
    ${ret} =    Return One Value  argument
    Some Keyword    ${ret}

Advanced
    @{list} =    Create List    foo    baz
    ${index} =    Find Index    baz    @{list}
    Should Be Equal    ${index}    ${1}
    ${index} =    Find Index    non existing    @{list}
    Should Be Equal    ${index}    ${-1}

*** Keywords ***
Return One Value
    [Arguments]    ${arg}
    Do Something    ${arg}
    ${value} =    Get Some Value
    Return From Keyword    ${value}
    Fail    This is not executed

Find Index
    [Arguments]    ${element}    @{items}
    ${index} =    Set Variable    ${0}
    :FOR    ${item}    IN    @{items}
    \    Return From Keyword If    '${item}' == '${element}'    ${index}
    \    ${index} =    Set Variable    ${index + 1}
    Return From Keyword    ${-1}    # Could also use [Return]
```

用户关键字**Teardown**

作者: 虞科敏

可以使用[Teardown]设置用户关键字的Teardown过程。
Teardown一般是一个单一的关键，常常可能是另一个用户关键字。
即使用户关键字失败，Teardown过程也会被执行。
和其他的Teardown行为一样， 其中的步骤不管其他步骤是否成功，都会被执行。
用户关键字Teardonw任何步骤失败，都会导致所在的测试用例失败，后面的测试步骤将不会再被执行。
作为Teardown中被执行的用户关键字名，可以通过变量传入。

样例

```
*** Keywords ***
With Teardown
    Do Something
    [Teardown]    Log    keyword teardown

Using variables
    [Documentation]    Teardown given as variable
    Do Something
    [Teardown]    ${TEARDOWN}
```

资源文件, **Resource File**

作者: 虞科敏

在Test Case File和suite的__init__ File中定义的用户关键字和变量只能在创建它们的文件中被使用。
Resource File提供了一个共享用户关键字和变量的手段。
Resource File的结构和Test Case File非常相似，但其中不能有Test Case Table。

导入和使用**Resource File**

Resource File通过Setting Table中的Source命令进行导入，在Source Cell后，紧跟提供Resource File Path信息的cell。

如果path为绝对路径，将会被直接使用
如果path为相对路径，path先以相对要求导入的文件所在目录的相对路径进行； 如果没有找到，会在Python的模块搜索路径中查找
导入的路径支持使用变量

路径中可以包含参数，建议使用这项功能已使path尽量独立于系统，如 ${RESOURCES}/login_resources.html 或 ${RESOURCE_PATH}
另外，在windows系统中，路径中的/会被自动替换为\

Resource File定义的关键字和变量，只在导入它的文件中可用
在Resource File导入的库Libraries，Resource File, Variable File，这些文件定义的关键字和变量，也是可用的

Resource File结构

Resource File的结构和Test Case File非常相似，但其中不能有**Test Case Table**。
另外，Resource File中的Setting Table只能包含**import settings** (Library, Resource, Variables) 和 **Documentation**。

Variable Table和Keyword Table的使用，和Test Case File完全一样。

如果多个资源文件中存在同名的用户关键字，为了引用这些用户关键字，需要在关键字名前加上资源文件的名字(不带后缀)。 如，myresources.Some Keyword 和 common.Some Keyword。

如果多个资源文件中存在同名的变量，采用"先导入优先原则"。

文档串

Resource File中定义的用户关键字可以使用[Documentation]设置文档串。
Resource File本身也可以在Setting Table中定义自己的文档串 (和Suite中一样)。

工具Libdoc和RIDE都会使用这些文档串，当它们打开资源文件时，这些文档串就会被它们使用。

关键字的文档串的第1行在它们被执行时，会被log记录，但其他的Resource File文档串在执行时会被忽略。

样例

```
*** Settings ***
Documentation     An example resource file
Library           Selenium2Library
Resource          ${RESOURCES}/common.robot

*** Variables ***
${HOST}           localhost:7272
${LOGIN URL}      http://${HOST}/
${WELCOME URL}    http://${HOST}/welcome.html
${BROWSER}        Firefox

*** Keywords ***
Open Login Page
    [Documentation]    Opens browser to login page
    Open Browser    ${LOGIN URL}    ${BROWSER}
    Title Should Be    Login Page

Input Name
    [Arguments]    ${name}
    Input Text    username_field    ${name}

Input Password
    [Arguments]    ${password}
    Input Text    password_field    ${password}
```

第2章 高级特性

作者：虞科敏

本节内容为熟练使用框架后的进阶内容，包含使用频度略低的高级特性。

第3章 实现测试库, Implementing Test Library

作者:虞科敏

Robot Framework除了提供强大的功能，还可以对框架进行自己的定制和扩展，获取更强大的能力。

测试库

作者: 虞科敏

编程语言

- Python - RF框架是使用Python实现的，所以，自然地，扩展测试库可以使用Python来实现。Python代码可以在Python或Jython上运行。
- Java - 当RF框架运行在Jython上时，也可以使用Java来实现测试库。
- C - 在Python上运行时，通过Python C API，可以使用C语言来实现测试库。另外，使用ctypes模块，Python测试库可以很容易地和C交互。

以上3种语言，也可以作为打包器(Wrapper)支持使用其他语言来实现测试库。
另外，Remote Library作为代理，远端库也可以使用其他语言来实现。

测试库API

RF框架提供了3种Test Libaray API:

静态API

静态库是最简单的一种方法: Python模块, Python类 或 Java类，函数或方法会被直接映射为关键字。
关键字的参数会和函数或方法的参数保持一致。
抛出异常，关键字会报告失败；
写到标准输出，关键字会进行日志记录；
使用return语句，关键字也会返回值。

动态API

动态库，类中会实现一个特殊的方法，通过它获得被实现的关键字。
其他方法则成为带有给定参数的命名关键字。
关键字的名字，以及如何被执行，将在运行时动态决定。
报告状态，日志，返回值，行为和静态库相似。

混杂型API

这是一种动态库和静态库的混杂类型。
混杂库，类带有一个告知哪些关键字被实现的方法，但这些关键字必须是直接可用的。
除了被实现关键字的发现机制，其他都和静态库是一样的。

以上3种类型API都会在本章讨论。
静态API的工作机制是所有API的基础，更为关键，会被首先讨论。 然后再讨论动态API和混杂型API的区别。

创建测试库:类或模块

作者: 虞科敏

实现测试库的常见技术包括:

- Python模块(Python Module)
- Python类(Python Class)
- Java类(Java Class

测试库名

作者: 虞科敏

测试库的名字，在库导入过程中用来标识测试库，是一个很重要的属性。

一般地，测试库的名字规则很简单，就是实现该库的模块或类的名字。
例如，

```
1. Python模块 "MyLibrary" (MyLibrary.py)
=> 测试库名: "MyLibrary"， 导入语句"Library MyLibrary"
2. Java类 "YourLibrary" (在缺省包中，即不在任何包中）
=> 测试库名: "YouLibrary"，导入语句"Library YourLibrary"
```

是不是非常简单？

对于Python类，情况有一点点特别。
根据Python的语法， Python类必然是在某一个模块中定义的(虽然很多python IDE会有创建module或者class的不同选项，但在Python中创建的模板文件都是一个模块).
如果实现测试库的类名字 和 类所在模块文件的文件一样， RF框架允许这样的语法: 导入测试库时， 导入模块名就可以了，可以省略掉类名 (:) Python中好像不能这样，要么包.模块.名称; 要么只导入模块的话，使用名字需要加上模块的限定词）.
例如，

```
1. 假设: 类MyLib定义在模块文件MyLib.py中
"Library MyLib" 等效于 "Library MyLib.Mylib"
2. 假设: 类MyLib定义在parent的子模块MyLib中
"Library parent.MyLib"
3. 假设: 类MyLib定义在parent的子模块submodule中
"Library parent.submodule.Mylib"
```

对于非缺省包中的Java类，情况也有一些特殊之处: 需要使用全限定名
例如，

```
1. 假设: 存在于com.mycompany.myproject包中的MyLib类
"Library com.mycompany.myproject.MyLib"
```

Tips:

```
测试库名的别名
如果测试库名太长，建议给测试库取一个别名。
别名的语法，前面章节已经有提及"WITH NAME"，后面也有专门章节介绍。
(Pythener们，是不是依稀记得，from-import-as语法中，也具有相似的语义的as :) )
```

测试库的参数

作者: 虞科敏

以类为底层机制实现的测试库，可以接受参数. (Pytherner们是不是想起了 **init**()， Javaer们是不是想起了 constructor()，实际上传递给测试库的参数，会被传递到定义它们的类的构造函数中)

以模块为底层机制实现的测试库，因为模块不能接收参数，所以不支持传递参数的语法。 传如参数会报错。

参数的数量，和实现库的类构造函数一致。 缺省值和可变参数，也遵照普通关键字的参数规则(Java类实现的库不支持可变参数)。

参数名字和参数，都支持使用变量。

样例
测试库MyLibrary 采用Python类实现
测试库AnotherLib 采用Java类实现

```
*** Settings ***
Library    MyLibrary     10.0.0.1    8080
Library    AnotherLib    ${VAR}

Python Implementation
class MyLibrary:

    def __init__(self, host, port=80):
        self._conn = Connection(host, int(port))

    def send_message(self, message):
        self._conn.send(message)

Java Implementation
public class AnotherLib {

    private String setting = null;

    public AnotherLib(String setting) {
        setting = setting;
    }

    public void doSomething() {
        if setting.equals("42") {
            // do something ...
        }
    }
}
```

测试库的作用范围

作者: 虞科敏

类实现的库，一般都有内部状态。 状态信息可以使用关键字，或者通过传入构造函数的参数被改变。 因为大多数时候， 内部状态是会影响到关键字行为的，所以保证一个用例中的执行不会影响到其他用例的执行行为，就非常重要。否则，这种通过内部状态被影响到的行为，将是非常难于被发现的。

RF框架是这样保证测试用例执行之间的独立性的: 对于每一个测试用例，一个新的测试库的类对象(或叫实例)被创建来用于该测试用例。
但是，这样做也可能造成一些其他的问题: 比如有些时候测试用例之间是希望共享一些公共状态的； 对于没有内部状态的测试类，每个用例创建一个实例，既没有必要，也浪费资源。

解决方案是，测试库使用类属性"ROBOT_LIBRARY_SCOPE"来指定库的作用范围。

ROBOT_LIBRARY_SCOPE属性 值为字符串，有3种可能取值:

TEST CASE

每个用例创建一个新实例。
Suite Setup和Suite Teardown共享地使用另一个实例。
测试库的缺省作用范围为 Test Case

TEST SUITE

每个Suite创建一个新实例。
每个层级的Suite有它自己的库实例。 比如最低级的Test Suite(由Test Case File创建，直接包含测试用例的Suite)有它自己的库实例; 更高一级的Test Suite也有自己的库实例。 因为每个实例可能存在的Suite Setup和Suite Teardown需要使用自己的库实例。

GLOBAL

整个测试过程只有一个库实例被创建。
所有的Test Case和Test Suite共享同一个库实例。
使用模块创建的库，作用范围总是GLOBAL

```
Tips:
如果不同参数被传给给库，不管作用范围的值，每次一个新的库实例都会被创建。
```

如果Suite或Global库带有内部状态，建议库中定义一些特殊的关键字用来设置(或重置,或清除)状态。这样，在每一个Suite的Setup或Teardown中，可以使用这些特殊关键字来保证下一个Suite中的测试用例该库工作在一个可知的工作状态。
例如，SeleniumLibrary使用Global保证每个测试用例都使用同一个浏览器，不必每个用例都重新打开。 它也提供了关键字"Close All Browsers"，可以很容易地关闭所有打开的浏览器。

样例*1*
Python实现的Suite范围测试库

```python
class ExampleLibrary:

    ROBOT_LIBRARY_SCOPE = 'TEST SUITE'

    def __init__(self):
        self._counter = 0

    def count(self):
        self._counter += 1
        print self._counter

    def clear_counter(self):
        self._counter = 0
```

样例2
Java实现的Global范围测试库

```java
public class ExampleLibrary {

    public static final String ROBOT_LIBRARY_SCOPE = "GLOBAL";

    private int counter = 0;

    public void count() {
        counter += 1;
        System.out.println(counter);
    }

    public void clearCounter() {
        counter = 0;
    }
}
```

测试库版本

作者: 虞科敏

RF框架使用测试库时，会去获取测试库的版本信息。版本信息也会作为重要调试信息被写入syslog；Libdoc工具也会将版本信息写入关键字的文档说明中。

测试库使用类属性"ROBOT_LIBRARY_VERSION"来指定库的作用范围。 如果类属性"ROBOT_LIBRARY_VERSION"不存在，会去尝试"__version__"属性。
根据测试库是有模块还是类实现的，这两个属性必须应该有一个存在。
另外，Java测试库版本信息属性需要被声明为"static final"。

实例
Python模板使用属性"__version__"

```
__version__ = '0.1'

def keyword():
    pass
```

Java类使用属性"ROBOT_LIBRARY_VERSION"

```
public class VersionExample {

    public static final String ROBOT_LIBRARY_VERSION = "1.0.2";

    public void keyword() {
    }
}
```

测试库的文档说明

作者: 虞科敏

测试库文档生成工具Libdoc执行多个格式的文档。
如果希望RF框架已支持格式之外的格式，可以在测试库源码中使用类属性"ROBOT_LIBRARY_DOC_FORMAT"来指定格式。

文档格式的可能取值(大小写敏感):

- ROBOT (default)
- HTML
- TEXT (plain text)
- reST (reStructuredText)

reST格式在生成文档时，需要docutils模块的支持。

测试库的说明文档，更多信息可以参考文档生成库，Libdoc工具等章节。

实例*1*
Python测试库中使用reStructuredText格式

```
"""A library for *documentation format* demonstration purposes.

This documentation is created using reStructuredText__. Here is a link
to the only \`Keyword\`.

__ http://docutils.sourceforge.net
"""

ROBOT_LIBRARY_DOC_FORMAT = 'reST'

def keyword():
    """Nothing to see here. Not even in the table below.

    =======  =====  =====
    Table    here   has
    nothing  to     see.
    =======  =====  =====
    """
    pass
```

实例*2*
Java测试库中使用HTML格式

```
/**
 * A library for <i>documentation format</i> demonstration purposes.
 *
 * This documentation is created using <a href="http://www.w3.org/html">HTML</a>.
 * Here is a link to the only `Keyword`.
 */
public class DocFormatExample {

    public static final String ROBOT_LIBRARY_DOC_FORMAT = "HTML";

    /\**<b>Nothing</b> to see here. Not even in the table below.
     \*
     \* <table>
     \* <tr><td>Table</td><td>here</td><td>has</td></tr>
     \* <tr><td>nothing</td><td>to</td><td>see.</td></tr>
     \* </table>
     \*/
    public void keyword() {
    }
}
```

测试库工作为**Listener**

Listner接口，允许外部Listners获取测试执行情况的通知。
当Suite，Test, Keyword开始和结束时, 它们被调用。
有些时候，测试库获取这类通知是非常有用的。

可以通过使用类属性"ROBOT_LIBRARY_LISTENER"来注册用户Listner。
本属性的值应该是用作Listener的类实例。 也可能就是库本身。

更多关于Listner的信息和样例，可以参考文Test libraries as listeners章节。

静态API

作者: 虞科敏

使用静态Library API技术创建静态关键字，是最常用的测试库实现手段。

成为关键字的方法

使用Static Library API技术时，RF框架利用反射技术，来查找类或模块的实现中的公共方法作为关键字:

- Python中以"-"开头的方法会被忽略。
- Java中非public方法，只在Object类中实现的方法会被忽略。
- 所有未被忽略的方法，将被识别为关键字。

样例*1*
Python测试库，实现了关键字"My Keyword"

```
class MyLibrary:

    def my_keyword(self, arg):
        return self._helper_method(arg)

    def _helper_method(self, arg):
        return arg.upper()
```

Java测试库，实现了关键字"My Keyword"

```
public class MyLibrary {

    public String myKeyword(String arg) {
        return helperMethod(arg);
    }

    private String helperMethod(String arg) {
        return arg.toUpperCase();
    }
}
```

以Python模块实现的测试库，也可以通过Python的"__all__"属性来指定实现关键字的方法。
如果"__all__"属性存在，只有其中所列的方法会被识别为关键字。

样例*2*
如果没有"__all__"属性, current*thread* 和 *not_exposed_as_keyword*都会被暴露为关键字*"Current Thread"*和*"Not Exposed As Keyword"*。
"_all__"属性最重要的作用，就是保证在实现文件中导入的帮助性工具性方法，不要被意外地暴露成关键字。

```
from threading import current_thread

__all__ = ['example_keyword', 'second_example']

def example_keyword():
    if current_thread().name == 'MainThread':
        print 'Running in main thread'

def second_example():
    pass

def not_exposed_as_keyword():
    pass
```

关键字的名字

作者: 虞科敏

关键字的名字，会和实现关键字的方法的方法名进行比较。

这个比较是大小写不敏感的，并且空格和下滑下_也会被忽略掉。
例如，在样例1和样例2中的方法，
方法名"hello" 可以被映射为关键字名 "Hello", "hello", 甚至"h e l l o" 方法名为"do_nothing"和"doNothing"的方法都会被用来作为关键字"Do Nothing"关键字的实现。

样例*1*
Python模块MyLibrary.py实现的测试库

```
def hello(name):
    print "Hello, %s!" % name

def do_nothing():
    pass
```

样例*2*
Java模块MyLibrary.java(MyLibrary类)实现的测试库

```
Example Java library implemented as a class in the MyLibrary.java file:

public class MyLibrary {

    public void hello(String name) {
        System.out.println("Hello, " + name + "!");
    }

    public void doNothing() {
    }

}
```

样例*3*
使用以上创建的测试库

```
*** Settings ***
Library    MyLibrary

*** Test Cases ***
My Test
    Do Nothing
    Hello    world
```

定制化关键字名

除了通过方法名直接映射生成的缺省关键字名，也可以为关键字设置定制化的名字。
这可以通过设置相应方法的属性"robot_name"来完成。
更方便一些，使用修饰器 robot.api.deco.keyword也可以定制化关键字名(原理也是设置属性"robot_name")。

样例
定制化关键字名

```
from robot.api.deco import keyword

@keyword('Login Via User Panel')
def login(username, password):
    # ...
```

测试数据表格

```
*** Test Cases ***
My Test
    Login Via User Panel    ${username}    ${password}
```

使用不带参数的 @keyword 修饰符，将不会对暴露的关键字名产生任何影响，仍旧为缺省的关键字名。
但这一语法仍然会创建方法的属性"robot_name"。空白的属性"robot_name"，在Dynamical Library API技术中，标明暴露的关键字也是有用的。

另外，之间讨论过的使用嵌入变量的语法让关键字接收参数，也可以定制化关键字名。

关键字的标签

作者: 虞科敏

库关键字 和 用户关键字 都可以设置标签。
库关键字通过设置方法的属性"rotot_tags" (值为tags的list)。
更方便一些，使用修饰器 robot.api.deco.keyword也可以设置标签(原理也是设置属性"robot_tags")。

样例*1*
修饰器 @keyword 的第2个位置参数为tags

```
from robot.api.deco import keyword

@keyword(tags=['tag1', 'tag2'])
def login(username, password):
    # ...

@keyword('Custom name', ['tags', 'here'])
def another_example():
    # ...
```

样例*2*
关键字文档串的最后一行，也可以设置标签: 以Tags开始，标签之间以","分隔

```
def login(username, password):
    """Log user in to SUT.

    Tags: tag1, tag2
    """
    # ...
```

关键字的参数

作者: 虞科敏

在静态API和混合型API中，关键字需要多少参数，直接通过实现关键字的方法来决定。
动态API中，有其他的方法来制定关键字需要的参数数量。

最常见，也是最简单的情况是， 一个关键字需要的参数数量已经明确。 这样，Python或Java方法只需要获取这些参数就可以。

样例

关键字No Arguments不需要参数，那么方法no_arguments签名中不需要任何参数。
关键字One Argument不需要参数，那么方法one_argument签名中指定一个参数。

```
def no_arguments():
    print "Keyword got no arguments."

def one_argument(arg):
    print "Keyword got one argument '%s'." % arg

def three_arguments(a1, a2, a3):
    print "Keyword got three arguments '%s', '%s' and '%s'." % (a1, a2, a3)
```

```
Tips: 因为Java不支持命名参数语法，所以Java测试库采用静态API实现时这是一个局限。
可以采用Python实现，或者动态API来实现。
```

关键字的参数缺省值

作者: 虞科敏

Python和Java实现参数缺省值，有不同的实现语法。

Python实现参数缺省值

使用Python，关键字的参数缺省值实现根本就不是个事，如果你已经是一个Pythoner，在函数或方法中指定参数缺省值，简直就是轻而易举的事情。 所以，没有太多说明，直接看例子 :)

样例
对于one_default, 调用时候可以是0或1个值。 如果无值，那么arg使用缺省值; 如果1个值, 那么arg使用这个值; 如果超过1个值, 会报错。
对于multiple_defaults, 至少1个值被需要，第2和第3个值有缺省值，所以调用时候可以是1-3个值。

Python实现

```
def one_default(arg='default'):
    print "Argument has value %s" % arg

def multiple_defaults(arg1, arg2='default 1', arg3='default 2'):
    print "Got arguments %s, %s and %s" % (arg1, arg2, arg3)
```

RF测试数据表格

```
*** Test Cases ***
Defaults
    One Default
    One Default    argument
    Multiple Defaults    required arg
    Multiple Defaults    required arg    optional
    Multiple Defaults    required arg    optional 1    optional 2
```

Java实现参数缺省值

使用Java，要达到以上效果，需要对同一个方法名，实现不同的签名: 参数个数不同。 这样, RF把这些不同的签名，认为是对于同一个关键字，不同参数个数的实现，达到对于参数缺省值的支持。

样例
以下是使用Java，实现上面Python样例同样的关键字 One Default 和 Multiple Defaults

```
public void oneDefault(String arg) {
    System.out.println("Argument has value " + arg);
}

public void oneDefault() {
    oneDefault("default");
}

public void multipleDefaults(String arg1, String arg2, String arg3) {
    System.out.println("Got arguments " + arg1 + ", " + arg2 + " and " + arg3);
}

public void multipleDefaults(String arg1, String arg2) {
    multipleDefaults(arg1, arg2, "default 2");
}

public void multipleDefaults(String arg1) {
    multipleDefaults(arg1, "default 1");
}
```

关键字的可变长参数 (*varargs)

作者: 虞科敏

Python和Java实现可变长参数，有不同的实现语法。

Python实现可变长参数

Python函数(或方法)本来就支持可变长参数的语法， 即非关键字可变长参数语法(*args，即tuple)，这对于Pythoner一定不会陌生。 使用本语法在测试库中，即能提供对于关键字的可变长参数的支持。

样例

Python实现

```
def any_arguments(*args):
    print "Got arguments:"
    for arg in args:
        print arg

def one_required(required, *others):
    print "Required: %s\nOthers:" % required
    for arg in others:
        print arg

def also_defaults(req, def1="default 1", def2="default 2", *rest):
    print req, def1, def2, rest
```

RF测试数据表格

```
*** Test Cases ***
Varargs
    Any Arguments
    Any Arguments    argument
    Any Arguments    arg 1    arg 2    arg 3    arg 4    arg 5
    One Required     required arg
    One Required     required arg    another arg    yet another
    Also Defaults    required
    Also Defaults    required    these two    have defaults
    Also Defaults    1    2    3    4    5    6
```

Java实现可变长参数

Java方法中对于可变数量参数，或者使用数组语法，或者使用"..."语法。 这些语法，也用在RF框架中对关键字的可变长参数的支持。 另外，RF框架还支持List类型参数作为最后1个值（或者倒数第2个值，如果最后1个是自由参数的话）来提供对关键字的可变长参数的支持。

样例*1*
使用"..."语法实现可变长参数

```
public void anyArguments(String... varargs) {
    System.out.println("Got arguments:");
    for (String arg: varargs) {
        System.out.println(arg);
    }
}

public void oneRequired(String required, String... others) {
    System.out.println("Required: " + required + "\nOthers:");
    for (String arg: others) {
        System.out.println(arg);
    }
}
```

样例2
使用数组和List实现可变长参数

```
public void anyArguments(String[] varargs) {
    System.out.println("Got arguments:");
    for (String arg: varargs) {
        System.out.println(arg);
    }
}

public void oneRequired(String required, List<String> others) {
    System.out.println("Required: " + required + "\nOthers:");
    for (String arg: others) {
        System.out.println(arg);
    }
}
```

Tips: 仅仅java.util.List类型的参数可实现可变长参数，子类型是不行的。
另外，Java实现不能同时支持缺省值和可变长参数。

关键字的自由参数 (**kwargs)

作者: 虞科敏

Python和Java实现可变长参数，有不同的实现语法。

Python实现可变长参数

Python函数(或方法)本来就支持自由参数的语法， 即关键字可变长参数语法(**kwargs，即dict)，这对于Pythoner也是经常使用的。 使用本语法在测试库中，即能提供对于关键字的自由参数的支持。

样例*1*
如果关键字参数，后面的命名参数name=value，如果不匹配任何其他参数，那么它们被传递给自由参数。

Python实现

```
def example_keyword(**stuff):
    for name, value in stuff.items():
        print name, value
```

RF测试数据表格

```
*** Test Cases ***
Keyword Arguments
    Example Keyword    hello=world        # Logs 'hello world'.
    Example Keyword    foo=1    bar=42    # Logs 'foo 1' and 'bar 42'.
```

```
Tips: 如果想使用"name=value"的字面量，传递给参数，而不是作为一个命名参数，那么需要对"="进行转义，即"name\=value"
```

样例*2*
普通位置参数，可变长参数，自由参数共存的情况

Python实现

```
def various_args(arg, *varargs, **kwargs):
    print 'arg:', arg
    for value in varargs:
        print 'vararg:', value
    for name, value in sorted(kwargs.items()):
        print 'kwarg:', name, value
```

RF测试数据表格

```
*** Test Cases ***
Positional
    Various Args    hello    world               # Logs 'arg: hello' and 'vararg: world'.

Named
    Various Args    arg=value                    # Logs 'arg: value'.

Kwargs
    Various Args    hello a=1    b=2    c=3            # Logs 'arg: hello' and 'kwarg: a 1', 'kwarg: b 2' and 'kwarg: c 3'
    Various Args    hello c=3    a=1    b=2            # Same as above. Order does not matter.

Positional and kwargs
    Various Args    1    2    kw=3               # Logs 'arg: 1', 'vararg: 2' and 'kwarg: kw 3'.

Named and kwargs
    Various Args    arg=value      hello=world    # Logs 'arg: value' and 'kwarg: hello world'.
    Various Args    hello=world    arg=value      # Same as above. Order does not matter.
```

Java实现自由参数

不同于Python，Java语言本身并不支持类似自由参数的语法，RF框架定义关键字的实现，最后一个参数使用java.util.Map类型，来实现自由参数。
RF框架会将name=value语法，转换为key和value，设置到Map类型参数中。

样例
使用Java实现和上面Python实现的Example Keyword和Various Args完全相同的关键。

```
public void exampleKeyword(Map<String, String> stuff):
    for (String key: stuff.keySet())
        System.out.println(key + " " + stuff.get(key));

public void variousArgs(String arg, List<String> varargs, Map<String, Object> kwargs):
    System.out.println("arg: " + arg);
    for (String varg: varargs)
        System.out.println("vararg: " + varg);
    for (String key: kwargs.keySet())
        System.out.println("kwarg: " + key + " " + kwargs.get(key));
```

Tips: 仅仅java.util.Map类型的参数可实现自由参数，子类型是不行的。
另外，Java实现不能同时支持缺省值和自由参数。

关键字的参数类型

作者: 虞科敏

关键字的参数值缺省一般为字符串。
如果关键字需要其他类型，那么可以使用变量，或者在关键字内部将字符串转换为需要的类型。
Java关键字，基本类型会进行自动的强制转换。

Python关键字的参数类型

Python是一种动态语言，变量在编码时是不会制定类型信息的，所以对于Python测试库，不可能自动转换字符串到其他类型。 调用Python函数或方法，只要参数数量合法，调用就总能成功； 但是参数如果不兼容，那么执行会失败。

样例
Python可以使用内建的工厂函数在进行显式地强制类似转换。

```
def connect_to_host(address, port=25):
    port = int(port)
```

Java关键字的参数类型

Java语言，编码时，方法的参数是有类型信息的；所有基本类型对于RF会被自动处理。
所有测试数据表格中的普通基本类型，在运行时会被自动强制转换为正确的类型。

能进行强制转换的内容包括:
整形 (byte, short, int, long)
浮点型 (float and double)
布尔型
上述基本类型的对象类，如 java.lang.Integer

样例1

当测试方法的签名中，所有的类型为相同或兼容的时，强制转换可以自动完成。
doubleArgument和compatibleTypes中强制转换可以完成；conflictingTypes中强制转换不能完成。

```
public void doubleArgument(double arg) {}

public void compatibleTypes(String arg1, Integer arg2) {}
public void compatibleTypes(String arg2, Integer arg2, Boolean arg3) {}

public void conflictingTypes(String arg1, int arg2) {}
public void conflictingTypes(int arg1, String arg2) {}
```

样例2

如果测试数据为数字的字符串，或者"true""false"的字符串，强制转换为数字或布尔型是可能的。
测试数据中的原始值为字符串，强制转换才可能发生。 当然，如果使用包含正确类型的变量，也是可行的。
如果存在冲突的数字签名，那么使用变量来指明类型，就是唯一的办法。

```
*** Test Cases ***
Coercion
    Double Argument     3.14
    Double Argument     2e16
    Compatible Types    Hello, world!    1234
    Compatible Types    Hi again!    -10    true

No Coercion
    Double Argument    ${3.14}
    Conflicting Types    1       ${2}    # must use variables
    Conflicting Types    ${1}    2
```

使用修饰器

作者: 虞科敏

在创建静态关键字时， 使用Python修饰器有时的确会非常方便。
但是，通过修饰器修改函数签名，可能会让RF框架工作出现一些问题，混淆其自检机制。 特别是使用文档工具Libdoc和 IDE工具RIDE时，可能会有问题。 建议，要么不用修饰器，要么使用 decorator模块。

嵌入参数到关键字名中

作者: 虞科敏

像用户关键字，库关键字也支持嵌入变量到关键字名中的语法。
修饰器 robot.api.deco.keyword , 可以创建支持预期格式的变量嵌入，得到定制化的关键字名。

样例

Python实现

```
from robot.api.deco import keyword

@keyword('Add ${quantity:\d+} Copies Of ${item} To Cart')
def add_copies_to_cart(quantity, item):
    # ...
```

测试数据表格

```
*** Test Cases ***
My Test
    Add 7 Copies Of Coffee To Cart
```

和RF框架通信

作者: 虞科敏

实现关键字的函数或方法被调用后， 存在一些机制来和RF系统进行通信:

- 发送消息给RF的log文件
- 返回值，被RF保存到变量中
- 报告关键字是否执行通过

报告关键字状态

使用异常(Exception)，可以报告关键字的执行状态。
如果执行过程中，函数和方法抛出异常, 关键字的执行状态为失败；如果正常返回，执行状态为通过。

在日志(log)，报告(report)和控制台(console)中显示的错误消息，来自抛出的异常类型信息和它所带的错误消息。
常见异常(AssertionError, Exception, RuntimeError)，只显示: 异常错误消息； 其他异常，显示信息的格式:
ExceptionType: 异常错误消息

也可以设置你的异常中特殊属性ROBOT_SUPPRESS_NAME为True，这样显示消息的格式也可以和常见异常一样，不会带上ExceptionType作为前缀。

样例

Python示范

```
class MyError(RuntimeError):
    ROBOT\_SUPPRESS\_NAME = True
```

Java示范

```
public class MyError extends RuntimeException {
    public static final boolean ROBOT_SUPPRESS_NAME = true;
}
```

错误消息中的HTML

使用"*HTML*"作为消息的开头，可以使用HTML格式的错误消息。
例如：
raise AssertionError("*HTML* Robot Framework rulez!")
除了在测试库的exception中，这种语法也可以使用测试数据表格中指定错误消息。

自动截断长消息

如果错误消息超过40行， 在报告中会从中间自动截断，避免报告太长影响阅读。
但在日志消息中，完整的错误消息会被打印出来。

Traceback

使用Debug级别的日志，异常的traceback信息会被打印出来。
这种级别的消息缺省时候不会打印出来，因为普通用户不可能关心这些信息。 开发测试库时，可以使用 --loglevel DEBUG 打印这些信息。

停止和继续测试执行

作者: 虞科敏

停止测试执行

可以通过设置，让一个测试用例失败，停止整个测试执行。
要达到这种效果，应该将关键字中抛出的异常，其中的特殊属性"ROBOT_EXIT_ON_FAILURE"设置为True。

样例

Python实现

```
class MyFatalError(RuntimeError):
    ROBOT_EXIT_ON_FAILURE = True
```

Java实现

```
public class MyFatalError extends RuntimeException {
    public static final boolean ROBOT_EXIT_ON_FAILURE = true;
}
```

失败时也继续执行

可以通过设置，即时某一个测试用例失败，整个测试也将继续执行。
要达到这种效果，应该将关键字中抛出的异常，其中的特殊属性"ROBOT_CONTINUE_ON_FAILURE"设置为True。

样例

Python实现

```
class MyContinuableError(RuntimeError):
    ROBOT_CONTINUE_ON_FAILURE = True
```

Java实现

```
public class MyContinuableError extends RuntimeException {
    public static final boolean ROBOT_CONTINUE_ON_FAILURE = true;
}
```

日志消息(logging)

作者: 虞科敏

异常的错误消息不是提供信息给用户的唯一选择。
函数或方法也可以发送消息给日志文件: 只需要写信息到标准输出流或标准错误流。 你也可以设置不同的日志级别。

除了直接写信息到流， 使用编程logging API应该更地道的Pythonic一些。

缺省情况下， 方法中写到标准输出流的信息，以日志级别INFO写到日志文件。 写到标准错误流的信息，也会被相似地处理。但是在关键字执行完成后，这些信息也会被同时传递给最初的流。所以，如果希望在执行测试的控制台(console)这些日志消息可见，可以使用输出到标准错误流这个技巧。

日志级别

对于非INFO级别的日志消息，可以在实际的日志消息每行以 *LEVEL* 开头。
可用的日志级别有: TRACE, DEBUG, INFO, WARN, ERROR, HTML

错误(Errors)和警告(Warnings)

Error级别和Warn级别的日志，自动写到两个地方: 也打印在控制台(Console), 同时也会写到日志文件的"Test Execution Erros"部分。

如此，这两个级别的日志消息用户可见度更高，一般用来报告严重但非关键的问题。

HTML格式日志

测试库所有记录到日志的信息，都会被转换为 可供HTML呈现的格式。
例如，foo 在日志中将显示为 "foo", 而不是 "foo"。

如果测试库想要使用格式，链接，图片等语法，可以使用特殊的日志级别 HTML。
RF框架会将这些消息以INFO级别直接写入日志文件，这样它们就能使用HTML语法。
但是这个技巧应该谨慎使用，因为很容易破坏文件的显示格式。

当使用公共logging API时，各logging方法都有选择 "html"，设置为True，可以HTML格式输出日志。

时间戳

缺省地，当关键字执行完成时，输出到标准输出流和标准错误流的日志消息会记录时间戳。
如此记录的时间戳并不准确，用这些信息来调试可能会有问题。

更准确一些的时间戳, 使用Unix epoch时间毫秒数（即1970/1/1经过的秒数或毫秒数), 紧跟在日志级别后面，用":"分隔，如:
*INFO:1308435758660 *Message with timestamp*
\HTML:1308435758661* **HTML** message with timestamp

样例
实现以上时间戳的代码示例

Python示范

```
import time

def example_keyword():
    print '*INFO:%d* Message with timestamp' % (time.time()*1000)
```

Java示范

```
public void exampleKeyword() {
    System.out.println("*INFO:" + System.currentTimeMillis() + "* Message with timestamp");
}
```

日志输出到控制台

如前所述，警告 或 写到标准错误流的信息， 都会被同时写到 日志文件 和 控制台。
这两个方法的局限是，每次消息更新到控制台，都要等到当前执行的关键字完成后才能触发。(好处是，这两个方法对于Python和Java都适用)

另一个可选的方法，只能在Python中使用：将信息写到 sys.__stdout__ 或 sys.__stderr__
这个方法的效果是： 信息立刻被写到控制台，完全不会被写到日志文件中。

样例*1*
信息写到 sys.__stdout__ 或 sys.__stderr__

```
import sys

def my_keyword(arg):
   sys.__stdout__.write('Got arg %s\n' % arg)
```

还有一个可选方法是，使用公共logging API

样例*2*
使用公共logging API

```
from robot.api import logger

def log_to_console(arg):
   logger.console('Got arg %s' % arg)

def log_to_console_and_log_file(arg)
   logger.info('Got arg %s' % arg, also_console=True)
```

logging样例

对于日志的不同级别:
大多数时候，INFO级别已经足够
低于INFO级别的 DEBUG 和 TRACE，主要用来打印调试信息，帮助查找测试库自己的问题
WARN 和 ERROR 级别，用来让消息具有更多的可见性
HTML 用在需要使用某些格式的时候

样例

Python示范

```
print 'Hello from a library.'
print '\*WARN\* Warning from a library.'
print '\*ERROR\* Something unexpected happen that may indicate a problem in the test.'
print '\*INFO\* Hello again!'
print 'This will be part of the previous message.'
print '\*INFO\* This is a new message.'
print '\*INFO\* This is <b>normal text</b>.'
print '\*HTML\* This is <b>bold</b>.'
print '\*HTML\* <a href="http://robotframework.org">Robot Framework</a>'
```

日志级别和效果

```
16:18:42.123    INFO    Hello from a library.
16:18:42.123    WARN    Warning from a library.
16:18:42.123    ERROR    Something unexpected happen that may indicate a problem in the test.
16:18:42.123    INFO    Hello again!
This will be part of the previous message.
16:18:42.123    INFO    This is a new message.
16:18:42.123    INFO    This is normal text.
16:18:42.123    INFO    This is bold.
16:18:42.123    INFO    Robot Framework
```

Logging编程API

作者: 虞科敏

编程API 提供比 直接写到标准输出流或标准输出流 更为简洁方便的手段，目前编程API只在Python库的实现中可用。

Logging公共API

基于Python的Logging API，将直接输出消息到日志文件和控制台。
如: logger.info('My message') == 输出消息 "*INFO* My message" 到标准流
使用Logging API，更准确的时间戳也会被记录。

比较消息的文档资料在 https://robot-framework.readthedocs.org

样例
使用示范

```
from robot.api import logger

def my_keyword(arg):
    logger.debug('Got argument %s' % arg)
    do_something()
    logger.info('<i>This</i> is a boring example', html=True)
    logger.console('Hello, console!')
```

使用Logging公共API的测试库，存在对RF框架的依赖。

使用Python的标准库logging

RF框架也可以使用Python的内建库logging模块。
所有被logging模块的root logger接收到的消息，会被自动转播到RF日志文件。
在此方法中， 更准确的时间戳也会被记录。
但是不支持: 以HTML格式记录；输出到控制台。

此方法中，日志级别和RF框架中的日志级别有细微的差别:
DEBUG, INFO, WARNING, ERROR => 和RF框架中的日志级别一致
CRITICAL => 映射为ERROR级别
用户自定义级别 => 映射为低于本自定义级别的，最靠近的标准级别。 如一个介于INFO和WARNING之间的用户自定义级别，会被映射为INFO级别。

样例
使用示范

```
import logging

def my_keyword(arg):
    logging.debug('Got argument %s' % arg)
    do_something()
    logging.info('This is a boring example')
```

使用 Python的标准库logging模块 的测试库，不存在对RF框架的依赖。

测试库初始化阶段**Logging**

作者: 虞科敏

当测试库被导入，正在被初始化过程中，测试库已经可以开始输出日志。
这些日志消息不会被写到日志文件，而是输出到系统的syslog中。
这提供了 记录测试库初始化阶段所有有用信息 的手段。 同时， WARN和ERROR级别的日志消息，也会输出到日志文件的test execution erros部分。

测试库导入和初始化过程中，输出消息到标准输出流或标准出错流，logging编程API都可工作。

样例*1* Java测试库，在导入和和初始化过程中，输出消息到标准输出流或标准出错流

```
public class LoggingDuringInitialization {

    public LoggingDuringInitialization() {
        System.out.println("*INFO* Initializing library");
    }

    public void keyword() {
        // ...
    }
}
```

样例*2*
Python测试库，在导入和初始化过程中，使用logging编程API

```
from robot.api import logger

logger.debug("Importing library")

def keyword():
    # ...
```

如果在测试库导入和初始化过程中logging，如 Python __inin__.py 或者 Java构造函数，根据测试库的作用范围，日志消息可能会输出多次

返回值

作者: 虞科敏

关键字的返回值可以被保存在某变量中，然后传递给其他关键字作为参数输入。
这可以作为在不同关键字(不同测试库)之间的通信手段。

Python和Java语言中的return语句，将返回关键字的值。

样例*1*
可以返回对象作为返回值，赋值给1个scalar变量。
之后可以使用扩展变量语法访问对象的属性。

Java实现

```
from mymodule import MyObject

def return_string():
    return "Hello, world!"

def return_object(name):
    return MyObject(name)
```

测试数据表格

```
*** Test Cases ***
Returning one value
    ${string} =    Return String
    Should Be Equal    ${string}    Hello, world!
    ${object} =    Return Object    Robot
    Should Be Equal    ${object.name}    Robot
```

样例*2*
关键字返回多个值，这些值可以: 一次性付给多个scalar变量； 1个list变量； 若干scalar变量+1个list变量
要求返回的值类型: Python list, tuple 或者 Java array, list, Iterator

Python实现

```
def return_two_values():
    return 'first value', 'second value'

def return_multiple_values():
    return ['a', 'list', 'of', 'strings']
```

测试数据表格

```
*** Test Cases ***
Returning multiple values
    ${var1}    ${var2} =    Return Two Values
    Should Be Equal    ${var1}    first value
    Should Be Equal    ${var2}    second value
    @{list} =    Return Two Values
    Should Be Equal    @{list}[0]    first value
    Should Be Equal    @{list}[1]    second value
    ${s1}    ${s2}    @{li} =    Return Multiple Values
    Should Be Equal    ${s1} ${s2}    a list
    Should Be Equal    @{li}[0] @{li}[1]    of strings
```

线程间通信

作者: 虞科敏

如果测试库中使用了线程，通常，只有主线程会和框架进行通信。
如果某个工作线程需要报告错误，或者logging某事件等情况， 需要先将此信息传递给主线程， 然后主线程会使用抛出异常，或前面提到的通信机制，和RF框架通信。

和RF通信非常重要，特别是很多工作线程在后台运行，而其他关键字还在同时执行的时候。因为和RF框架的通信结果的不确定性，糟糕的情况下可能会crash掉整个框架，破坏输出文件等。
如果一个关键字开启了某后台任务，它应该提供其他关键字来检查工作线程的状态，以及报告收集到的相应信息。

非主线程的工作线程，使用logging编程API试图打印日志信息，会被RF框架已静默地方式悄悄丢弃掉。

独立项目"robot background logger"中提供了BackgroundLogger类，它的使用方法和logging公共API很相似。 除了非主线程的logging，此类会保存下这些后台消息，之后在可以logging的时候再输出到RF框架的日志中。

测试库的发布

作者: 虞科敏

测试库的文档信息

没有文档的测试库，实际意义不大，因为直接使用者很难了解测试库，它有哪些关键字，这些关键字能做些什么。
为了方便维护，最推荐的做法是： 测试库的文档信息嵌入于源代码中，使用工具提取产生测试库的文档；而不是单独写一个篇文档。
Python语言的文档串(docstrings)，Java语言的Javadoc，为这种手段提供了语言基础。

样例

Python示例

```python
class MyLibrary:
    """This is an example library with some documentation."""

    def keyword_with_short_documentation(self, argument):
        """This keyword has only a short documentation"""
        pass

    def keyword_with_longer_documentation(self):
        """First line of the documentation is here.

        Longer documentation continues here and it can contain
        multiple lines or paragraphs.
        """
        pass
```

Java示例

```java
/**
 *  This is an example library with some documentation.
 */
public class MyLibrary {

    /**
     * This keyword has only a short documentation
     */
    public void keywordWithShortDocumentation(String argument) {
    }

    /**
     * First line of the documentation is here.
     *
     * Longer documentation continues here and it can contain
     * multiple lines or paragraphs.
     */
    public void keywordWithLongerDocumentation() {
    }

}
```

Python和Java都有自己的语言工具，可以自动产生API技术文档。
RF框架也提供了自己的文档工具 Libdoc。 比较起来，它更适合用来生成RF测试库的文档，不仅仅支持静态Library API，也支持动态Libary API和混合型Library API。

关键字文档串的第1行有特殊的用途，应该包含本关键字的简单描述。
它被Libdoc工具作为特别的short documentation，也会显示在测试日志中。
(日志显示功能，对于Java静态API不适用，因为Javadoc在编译时会丢失，在运行时获取不到)

缺省地，文档串应该遵循RF框架的文档格式规则。 它们定义了一些常用格式，比如 *bold*, _italic_, tables, lists, links 等。
还有比如HTML, plain text, reStructuredText等格式也可被使用。

对测试库进行测试

为了避免Bug被发布出去，测试是必须的。
我们正在做的工作就是测试自动化，所以，对测试库的测试也尽量自动化起来吧，不然不够专业，是不 :)

Python和Java都有丰富而优秀的单元测试工具可供选择， 这些工具用来测试新开发的测试库都是很好的选择。
我用过的Python UT工具比如PyUnit(即Python发布已经自带的UnitTest), PyTest, Nose， Java UT工具比如JUnit 3，JUnit4都是不错的。 可以视自己的熟悉情况进行选择即可。

另外，你也可以使用RF框架来测试新开发的测试库，就像在进行一次端到端验收测试。 在BuiltIn Library中，有很多好用的关键字可用。 比如关键字Run Keyword And Expect Error，可以用来测试关键字是否正确的抛出错误(Errors)。

使用单元测试或是验收测试，应该是情况而定。
如果需要对实际的SUT(System Under Test)进行模拟，那么单元测试比较好；
而验收测试，可以确保关键字可以在RF框架中良好工作。

打包测试库

测试库实现，准备好相应文档，进行测试之后，就应该真正发布给使用者。
如果只是一个文件的简单测试库，那么直接拷贝给用户，设置好相应的模块搜索路径（或者就放在当前的模块搜索路径下），就可以使用了。

但是更复杂一些的测试库，为了便于安装，需要对测试库进行打包(packaging)。

测试库一般都是编程好的源代码，所以可以使用语言本身的打包分发工具来代劳。
比如，
Python可以使用distutils（Python发布自带的标准库），也可以使用功能更强大的三方工具，如 pip, setuptools。 使用这些分发工具，好处在于测试库模块被自动安装到位于模块搜索路径中的目录下，安装后就可以直接使用。
Java的分发一般使用jar工具将测试库代码打包为.jar文件。 将.jar文件放在模块搜索路径下，就可以使用测试库了。 如果先麻烦，写个脚本自动化这个过程也是常见的做法。

将关键字作废

如果需要用新的关键字替代已有关键字，或者删除某关键字，仅仅只是通知使用者这些改变，往往是不够的。 更好的做法是，在运行时直接提示警告信息。
RF框架有关键字作废机制，方便在测试数据中找出过时的关键字，进行替换或删除。

在关键字的文档串中，第1行以 "*DEPRECATED" (大小写敏感)开头, 以 "*"结束, 声明该关键字作废。
例如, *DEPRECATED*, *DEPRECATED.*, and *DEPRECATED in version 1.5.*

```
在最新的RF3.0中，只允许 \*DEPRECATED\*
```

已经作废的关键字被执行，作废警告信息将会在日志文件的Test Execution Errors部分中，控制台中被打印出来。
作废警告消息格式为: "Keyword " is deprecated." + 关键字的short documentation（即文档串第1行)

样例
关键字Example Keyword被执行，下面的作废警告会被打印。

关键字实现

```
def example_keyword(argument):
    """*DEPRECATED!!* Use keyword `Other Keyword` instead.

    This keyword does something to given ``argument`` and returns results.
    """
    return do_something(argument)  、
```

作废警告

```
20080911 16:00:22.650    WARN    Keyword 'SomeLibrary.Example Keyword' is deprecated. Use keyword `Other Keyword` instead.
```

如前所述，日志显示功能，对于Java静态API不适用。 所以作废警告对于Java静态API实现的关键字也不适用。如果需要作废警告，可以考虑使用用户关键字包装一下这类底层关键字。

动态**API**

作者: 虞科敏

动态Library API在很多地方，都和静态Library API相似。比如，报告关键字状态, 打印日志logging, 返回值都和静态API一样，请参考静态API的相应章节。
在导入动态测试库，和使用它们提供的关键字，和其他类型的测试库也是一样的。
也就是说，对于使用者，并不需要知道测试库是使用静态API技术，还是动态API技术实现的。

静态API和动态API的不同之处在于:
RF框架如何发现测试库实现了哪些关键字
RF框架如何发现这些关键字相应的参数和文档信息
RF框架如何执行这些关键字

对于静态API，以上这些行为都是通过反射技术实现（除了Java测试库的Javadoc）
对于动态API，有特殊的方法用来实现以上行为

动态API技术好处之一在于，在组织测试库是具有更大的灵活性。
静态API所有关键字必须放在同一个类或模块中；动态API则可以在不同的类中实现单个的关键字。

另一个好处在于，动态API可以实现一个测试库，它作为另一个真正意义上(真正实现功能)的功能测试库的代理来工作，而这些功能测试库可以运行为另一个进程，甚至运行在另一台机器上。
这种代理测试库可能会非常的简练短小，因为关键字名和相应的其他信息都是动态获取的，当功能测试库中添加新的关键字时也不需要更新代理测试库。

本章关注与动态API如何在RF框架和动态测试库之间工作。 动态库如何实现的，和RF框架没有太大关系。

获取关键字名

作者: 虞科敏

动态测试库，通过 get_keyword_names (getKeywordNames) 方法 申明关键字名。
此方法不带任何参数，返回字符串list或数组，包含本测试库实现的所有关键字的名字。

如果返回的关键字名，包含多个单词，可以使用空格，下划线"_"分隔它们，也可以使用驼峰格式。
如， ['first keyword', 'second keyword']
['first_keyword', 'second_keyword']
['firstKeyword', 'secondKeyword']
都对应关键字 First Keyword 和 Second Keyword

动态测试库，必须有此方法。 如果此方法缺失，或者调用失败，测试库将被认为是静态测试库。

标记将暴露为关键字的方法

动态测试库中，应该包含2类方法：一种是将作为关键字的关键字方法，另一种是私有的工具方法。
一个比较聪明的做法是，将关键字方法标记出来，这样通过筛选标记就可以很容易实现 get_keyword_names 方法。
修饰符 robot.api.deco.keyword 提供了一种很方便的方法来达到标记的目的，它在每个被它修饰的方法中，创造了自定义的属性 robot_name
这样，在动态测试库的get_keyword_names方法中的逻辑就是，检查测试库中的每一个方法是否有属性 robot_name。如果有，那么就是关键字方法； 否则就是工具方法。
在静态API中也用到了这个关键字，可以查看相关章节了解更多相关内容。

样例

```
from robot.api.deco import keyword

class DynamicExample:

    def get_keyword_names(self):
        return [name for name in dir(self) if hasattr(getattr(self, name), 'robot_name')]

    def helper_method(self):
        # ...

    @keyword
    def keyword_method(self):
        # ...
```

关键字的运行

作者: 虞科敏

动态API有1个特殊方法 run_keyword (或runKeyword)，负责运行关键字。
当执行测试数据表格中的动态关键字时, RF调用测试库的该方法来运行关键字。

该方法需要2-3个参数:
第1个参数: 字符串, 关键字的名字，应该和get_keyword_names返回的该关键字名相同
第2个参数: 列表或数组, 在测试数据表格中传递给关键字的参数
第3个参数: 可选, 字典或映射(Map), 传递给关键字的自由参数 (**kwargs)

获取关键字名和参数后，动态测试库开始执行关键字，它会使用和静态测试库相同的机制和RF框架通信:
抛出异常，告知框架关键字状态
打印到标准输出流，或使用logging API，输出日志
在run_keyword中的return语句，为关键字返回值

动态测试库必须包含方法: get_keyword_names 和 run_keyword。
其他方法是可选的。

样例

```
class DynamicExample:

    def get_keyword_names(self):
        return ['first keyword', 'second keyword']

    def run_keyword(self, name, args):
        print "Running keyword '%s' with arguments %s." % (name, args)
```

获取关键字参数

作者: 虞科敏

get_keyword_names 和 run_keyword方法并不能告知RF框架该关键字所需参数的信息。
例如，在样例中的First Keyword和Second Keyword可以接收任意数量的传输。
这在实际使用中是有些问题的，大多数关键字希望处理指定数量的参数。这种情况下，关键字需要自己来验证参数的数量。

动态API使用特殊方法 get_keyword_arguments (或getKeywordArguments)，告知RF框架该关键字希望传入的什么样的参数。
该方法的参数为关键字名，将返回该关键字希望的参数的(字符串)列表或数组。

和静态API关键字相同，动态API关键字可以接收任意数量的参数，支持缺省值，变长参数，自由参数。
下面以Python的语法来演示，如何表达各种不同的参数。 对于Java，请将Python list替换为Java List或Array。

参数	表达	样例	参数限制(最小/最大)
无参数	空list	[]	0//0
1个或多个	包含参数名的list	['one_argument'], ['a1', 'a2', 'a3']	1//1， 3//3
参数缺省值	List中的元素"name=value"	['arg=default value'], ['a', 'b=1', 'c=2']	0//1, 1//3
变长参数(varargs)	最后1个（或倒数第2个,如果有自由参数)参数名前加*	['*varargs'], ['a', 'b=42', 'rest']	0//any, 1//any
自由参数(kwargs)	最后1个参数名前加**	['**kwargs'], ['a', 'b=42', 'kws'], ['*varargs', '**kwargs']	0//any, 1//any, 0//any

当有get_keyword_arguments方法时，RF框架会自动计算 关键字需要的位置参数数量，是否支持变长参数或自由参数等。 如果所传入参数非法，会抛出错误，run_keyword方法不会被执行。

方法中定义的参数名和缺省值，是非常重要的，因为命名参数需要这些信息，Libdoc工具创建用户文档也需要这些信息。

如果get_keyword_arguments方法针对某关键字缺失，或者返回None(或null)，该关键字可以接收所有的参数(没有希望传入参数的规则)。 这种情况下，自动产生的参数为 [*varargs, **kwargs] 或者 [*varargs]，这依赖于方法run_keyword中传入2个还是3个参数。

获取关键字的文档信息

作者: 虞科敏

动态API使用特殊方法 get_keyword_arguments (或getKeywordArguments)，告知RF框架该关键字的文档信息。
该方法的参数为关键字名，将返回该关键字的文档信息字符串。

返回的文档信息字符串，和Python静态测试库的关键字文档串，具有相似的作用。最主要的用途，就是Libdoc工具用来产生用户文档。 另外，文档信息的第1行是short documentation，会出现测试日志中。

获取测试库的文档信息

动态API使用特殊方法 get_keyword_arguments (或getKeywordArguments)，也被用来提供整个测试库的文档信息。
测试库的文档信息，对于测试执行没有任何影响；但能帮助Libdoc工具产生出更好的用户文档。

动态测试库提供2种测试库文档信息: 普通测试库信息(方法参数特殊值 "__intro__")，测试库使用帮助(方法参数特殊值 "__init__")。
如果源代码中的文档串，和get_keyword_documentation方法返回值都不为空，优先使用方法返回值。

获取关键字标签

动态关键字的标签，只能通过文档信息来定义。 类似静态API中通过文档串来指定标签。 单独的方法 get_keyword_tags 还在计划之中。

样例
通过文档信息定义标签

```
def login(username, password):
    """Log user in to SUT.

    Tags: tag1, tag2
    """
    ...
```

命名参数语法

作者: 虞科敏

动态API支持命名参数语法。
由测试库的 get_keyword_arguments 方法获取的参数名和缺省值，支持该语法的实现。

动态关键字的命名参数语法，和其中关键字的命名参数语法使用相同。
细微的不同在于，如果动态关键字有多个带缺省值的参数，只有后面的参数可以使用缺省值语法。 框架使用 get_keyword_arguments 方法返回的缺省值，填充跳过的未给值参数。

样例
Dynamic关键字的签名 Dynamic [arg1, arg2=xxx, arg3=yyy]
参考comments中关键字实际被调用的参数值

```
*** Test Cases ***
Only positional
    Dynamic    a                          # [a]
    Dynamic    a         b                # [a, b]
    Dynamic    a         b         c      # [a, b, c]

Named
    Dynamic    a         arg2=b           # [a, b]
    Dynamic    a         b         arg3=c    # [a, b, c]
    Dynamic    a         arg2=b    arg3=c    # [a, b, c]
    Dynamic    arg1=a    arg2=b    arg3=c    # [a, b, c]

Fill skipped
    Dynamic    a         arg3=c           # [a, xxx, c]
```

自由参数语法

作者: 虞科敏

动态API支持自由参数语法。
测试库使用自由参数语法的前提是，run_keyword 方法使用3个参数: 第3个参数将用来获取kwargs，即自由参数。
kwargs 以Python dict或Java Map形式传递给关键字。

如前所述，关键字接收什么样的参数，由 get_keyword_arguments方法的返回值定义。 如果返回值中最后一个参数以 ** 开头，该关键字就被认为可以接收自由参数，kwargs。

样例
Dynamic关键字的签名 Dynamic [arg1=xxx, arg2=xxx, **kwargs]
参考comments中关键字实际被调用的参数值

```
*** Test Cases ***
No arguments
    Dynamic                             # [], {}

Only positional
    Dynamic    a                        # [a], {}
    Dynamic    a         b              # [a, b], {}

Only kwargs
    Dynamic    a=1                      # [], {a: 1}
    Dynamic    a=1       b=2    c=3     # [], {a: 1, b: 2, c: 3}

Positional and kwargs
    Dynamic    a         b=2            # [a], {b: 2}
    Dynamic    a         b=2    c=3     # [a], {b: 2, c: 3}

Named and kwargs
    Dynamic    arg1=a    b=2            # [a], {b: 2}
    Dynamic    arg2=a    b=2    c=3     # [xxx, a], {b: 2, c: 3}
```

小结

作者: 虞科敏

动态API中的所有特殊方法 表中方法名以_格式给出，也支持驼峰格式。
RF框架中最典型的动态API测试库的代表是 Remote Library。

方法	参数	作用
get_keyword_names	无	返回实现关键字的名字
run_keyword	name, arguments, kwargs	使用给定arguments执行指定keyword，其中kwargs可选
get_keyword_arguments	name	返回关键字的参数规范。 可选方法
get_keyword_documentation	name	返回关键字或测试库的文档信息。 可选方法

样例
Java语言的动态API规范，DynamicAPI接口
因为RF框架使用反射技术对特殊方法进行检查，所以动态库并非一定需要显示的实现以下接口
除了驼峰格式，_格式也是可接受的。
getKeywordArguments和getKeywordDocumentation是可选方法。
除了List，使用Object[]或String[]也是可接受的。

```
public interface RobotFrameworkDynamicAPI {

    List<String> getKeywordNames();

    Object runKeyword(String name, List arguments);

    Object runKeyword(String name, List arguments, Map kwargs);

    List<String> getKeywordArguments(String name);

    String getKeywordDocumentation(String name);

}
```

混合型API

作者: 虞科敏

混合型API，Hybrid Library API，顾名思义，介于静态API和动态API之间的兼具二者特点的API。

获取关键字名

和动态API相同，通过 get_keyword_names (getKeywordNames) 方法 表明关键字名。

关键字的运行

在混合型API中，没有 run_keyword (或runKeyword) 来负责运行关键字。 而是像静态API一样，RF框架使用反射直接找到实现关键字的方法。
使用混合型API实现测试库，或者直接有实现关键字的方法，或者动态的处理找到关键字实现方法的过程。

作为Pythoner，不难理解通过 __getattr__ 特殊方法来找到类属性的行为。
Java缺乏这种授权机制，所以混合型API的这种能力，对于Java并不好用。

样例
在__getattr__ 特殊方法中，并不会像 run_keyword 一样执行关键字，它只是返回一个可调用对象给RF框架执行。
另外，RF框架会使用得到的方法名，直接去调用 get_keyword_arguments方法。 比如，如果get_keyword_names 返回"My Keyword", 测试库就不能正常工作。

```
from somewhere import external_keyword

class HybridExample:

    def get_keyword_names(self):
        return ['my_keyword', 'external_keyword']

    def my_keyword(self, arg):
        print "My Keyword called with '%s'" % arg

    def __getattr__(self, name):
        if name == 'external_keyword':
            return external_keyword
        raise AttributeError("Non-existing attribute '%s'" % name)
```

Python的 __getattr__ 特殊方法

作为授权机制的重要步骤， 一般我们会在 __getattr__()方法中，包含一个对 getattr()内建函数的调用。 调用 getattr()可以以得到默认对象属性（数据属性或者方法）并返回它以便访问或调用。 这样，即能实现授权过程: 所有更新的功能都是由新类的某部分来处理，但已存在的功能就授权给对象的默认属性。

特殊方法 __getattr__() 的工作方式是，当搜索一个属性时，本地局部对象首先被找到（定制的对象）。 如果搜索失败了，则 __getattr__() 会被调用，在其中调用 getattr()得到一个对象的默认行为。
换言之，当引用一个属性时，Python 解释器的查找顺序是这样:

1. 在局部名称空间中搜索该名字，比如自定义的方法或局部实例属性；
2. 如果没有在局部字典中找到，则搜索类名称空间，比如本类的类属性，父类的类属性；
3. 如果两类搜索都失败了，搜索则对原对象，调用 __getattr__()。

获取关键字参数和文档信息

获取关键字方法后，接着获取关键字参数和文档信息，混合型API的工作方式，和静态API完全一样，使用反射机制获取。 不会需要动态API中使用到特殊方法。

小结

在Python测试库中，混合型API具有和动态API一样的动态能力。
其最大的好处是，不用像动态API一样，提供 get_keyword_arguments 和 get_keyword_documentation 方法。 这在只需要对个别关键字的功能实现部分授权，而其他功能只需要使用主测试库提供的情况下，非常实用。
所以，混合型API大多数时候是动态API更好的替代方案，除了作为代理测试库的实现情况，因为真正的库功能在其他地方实现，代理测试库只能对关键字名和参数进行转发。

RF框架中最典型的混合API测试库的代表是 Telnet Library。

扩展测试库

作者: 虞科敏

本章节讲述如何增加新功能到已有的测试库，以及如何在自己的测试库中使用。

修改源代码

如果已经有了测试库的源代码，想扩展功能，当然可以直接修改源代码。
这样做，最大的坏处是不利于维护，你很难在影响你自己的改动的情况下更新原来的测试库。
对于测试库使用者，也会对改动的测试库和原来的测试库之间的功能差别产生混淆。

当然，如果你的修改是普遍适用的，你也准备将改动提交进原测试库，在原始库的下一次发布中就会带上你的改动，那么以上的问题也就不存在。 否则，后面的一些方法应该更好一些。

使用继承

另一个比较直接的想法，是使用继承来扩展测试库。

样例
对于Selenium Library添加新关键字 Title Should Start With

```
from SeleniumLibrary import SeleniumLibrary

class ExtendedSeleniumLibrary(SeleniumLibrary):

    def title_should_start_with(self, expected):
        title = self.get_title()
        if not title.startswith(expected):
            raise AssertionError("Title '%s' did not start with '%s'"
                                 % (title, expected))
```

通过继承进行扩展，新的测试库会有一个不同于原测试库的名字。
好处是，这显示地表示你正在使用一个定制化的测试库。
坏处是，在你的新测试库中有很多和原测试库一样的关键字名，这可能导致很多冲突。 库之间也不能共享状态。

这种方法适用的场景是，从一开始，你就准备添加定制化增强功能到原测试库中。 否则，建议采用后面的方法。

直接使用其他测试库

从技术层面来看，测试库就是类或者模块，可以导入类或者模块后，使用其中的方法，来达到使用其他测试库的目的。
这种方法使用的场景是，测试库方法是静态API实现的，并且不依赖于测试库中的状态。

如果原测试库有内部状态，事情就不会按你预期那样进行。 因为在你的测试库中实例，和RF框架使用的实例并不相同，通过关键字进行的状态改变，对于你的测试库不可见。

从RF框架获取被激活测试库的实例

BuiltIn Library关键字 Get Library Instance 可以从RF框架中获取正在被使用的测试库实例。
其返回的实例，和测试库正在使用的实例完全相同，因此不存在测试库内部状态不可见的问题。
这个关键字的功能(对应的方法)，也能通过导入BuiltIn Library类的方式，在你的测试库中被使用。

样例
使用BuiltIn.get_library_instance()重写上面的Selenium Library扩展

```
from robot.libraries.BuiltIn import BuiltIn

def title_should_start_with(expected):
    seleniumlib = BuiltIn().get_library_instance('SeleniumLibrary')
    title = seleniumlib.get_title()
    if not title.startswith(expected):
        raise AssertionError("Title '%s' did not start with '%s'"
                             % (title, expected))
```

测试数据表示示例
比较继承方法，你可以同时使用原测试库和新测试库

```
*** Settings ***
Library    SeleniumLibrary
Library    SeLibExtensions

*** Test Cases ***
Example
    Open Browser    http://example      # SeleniumLibrary
    Title Should Start With    Example  # SeLibExtensions
```

以动态API或混合型API实现的测试库

以动态API或混合型API实现的测试库，一般有自己的扩展方法。
咨询测试库的开发者，查询库文档或源代码，获取如何扩展的方法。

第4章 常用测试库

作者:虞科敏

BuiltIn库

作者: 虞科敏

作用范围: Global
是否支持命名参数: 支持

介绍

这是RF框架非常重要的标准库，里面很多关键字都是高频使用的。
BuiltIn库被自动导入，其中的关键字总是可用的。
本库提供的关键字用途覆盖：
验证 - 如 Should Be Equal, Should Contain 等
转换 - 如 Convert To Integer 等
其他框架功能 - 如 Log, Sleep, Run Keyword If, Set Global Variable 等

出错消息

许多BuiltIn关键字可以接收可选的参数: 出错消息。 用于在关键字失败时候显示。
支持HTML消息，需要在消息前缀 "*HTML*"。 这个语法不止局限于BuiltIn关键字，所有框架提供的关键字中的出错消息都支持。

Evaluating表达式

许多BuiltIn关键字可以一个表达式，使用Python进行评估(Evaluate)，比如 Evaluate, Run Keyword If, Should Be True 等。
这些表达式Evaluating的原理是，使用Python的 eval 函数。
类似地，很多Python内建函数都是可用的，比如 len(), int() 等。

样例*1*
Evaluating允许使用用户模块来配置执行环境的名字空间，其他关键字自动获取 os, sys等模块。
请记得，在Evaluating表达式中，${variable}被赋值为变量值的字符串表达，而不是值本身。

```
Run Keyword If    os.sep == '/'    Log    Not on Windows
${random int} =    Evaluate    random.randint(0, 5)    modules=random
```

样例*2*
字符串值需要使用引号('或")包围起来。
如果其中包含新行，需要使用3引号语法。

```
Should Be True    ${rc} < 10    Return code greater than 10
Run Keyword If    '${status}' == 'PASS'    Log    Passed
Run Keyword If    'FAIL' in '''${output}'''    Log    Output contains FAIL
```

样例*3*
在Evaluating表达式中，变量在名字空间中自动可用。 可以使用特殊的不需要{}的语法 ($variable)访问。
这些变量不应被引用。
如果它们存在于字符串中，也不会被替换。

```
Should Be True    $rc < 10    Return code greater than 10
Run Keyword If    $status == 'PASS'    Log    Passed
Run Keyword If    'FAIL' in $output    Log    Output contains FAIL
Should Be True    len($result) > 1 and $result[1] == 'OK'
```

布尔型参数

一些关键字接收值为布尔型(true或false)的参数。
当这种参数传入的值为字符串时，
如果字符串值为 空，或者 等于false或no(不考虑大小写)，其值被认为等于 false。
验证性的关键字，如果验证内容允许 将来自出错消息的实际值或预期值丢弃，没有值也被认为等于 false。
其他情况，不管字符串值为何值，都会被认为等于 true。

样例*1*
True值

```
Should Be Equal    ${x}    ${y}    Custom error    values=True    # Strings are generally true.
Should Be Equal    ${x}    ${y}    Custom error    values=yes    # Same as the above.
Should Be Equal    ${x}    ${y}    Custom error    values=${TRUE}    # Python True is true.
Should Be Equal    ${x}    ${y}    Custom error    values=${42}    # Numbers other than 0 are true.
```

样例*2*
False值

```
Should Be Equal    ${x}    ${y}    Custom error    values=False    # String false is false.
Should Be Equal    ${x}    ${y}    Custom error    values=no    # Also string no is false.
Should Be Equal    ${x}    ${y}    Custom error    values=${EMPTY}    # Empty string is false.
Should Be Equal    ${x}    ${y}    Custom error    values=${FALSE}    # Python False is false.
Should Be Equal    ${x}    ${y}    Custom error    values=no values    # no values works with values argument
```

多行字符串的比较

关键字 Should Be Equal 和 Should Be Equal As Strings，对于多行(超过2行)字符串，使用unified diff format格式报告失败情况。

样例
测试数据表格

```
${first} =    Catenate    SEPARATOR=\n    Not in second    Same    Differs    Same
${second} =    Catenate    SEPARATOR=\n    Same    Differs2    Same    Not in first
Should Be Equal    ${first}    ${second}
```

结果

```
Multiline strings are different:
--- first
+++ second
@@ -1,4 +1,4 @@
-Not in second
 Same
-Differs
+Differs2
 Same
+Not in first
```

关于unified diff format，请百度或Google Unix命令diff结果的上下文格式的diff

BuiltIn关键字一览

关键字名	参数	文档说明
Call Method	object, method_name, *args*, *kwargs	调用对象实例(object)的方法(method_name)，参数为(*args*, *kwargs)
Catenate	*items	结合给定内容(items)到一起，可以使用分隔符(SEPARATOR；缺省分隔符为空格)，返回结果字符串

样例*1*

Catenate

```
${str1} =    Catenate    Hello    world
${str2} =    Catenate    SEPARATOR=---    Hello    world
${str3} =    Catenate    SEPARATOR=    Hello    world

=>

${str1} = 'Hello world'
${str2} = 'Hello---world'
${str3} = 'Helloworld'
```

Web测试库

作者: 虞科敏

Selelium2 Library

建设中...

Django Library

建设中...

GUI库

作者: 虞科敏

AutoIt Library

建设中...

swing Library

建设中...

数据库测试库

作者: 虞科敏

DatabaseLibrary

建设中...

接口测试库

作者: 虞科敏

RequestLibrary

建设中...

移动端测试库

作者: 虞科敏

AndroidLibrary

建设中...

iOSLibrary

建设中...

appiumLibrary

建设中...

CI自动化

作者: 虞科敏

Jenkins集成

建设中...

第5章 框架分析

作者:虞科敏

本章主要记录自己阅读RF框架代码的心得，记录一些重要编程知识点，关键逻辑，重要方法等。

Tips

作者: 虞科敏

本章目的不在于对于Python语言进行一次全面的科普，而只是为了帮助阅读RF框架源码，对于关键知识点，或不是特别常用的知识，进行一个简单的说明，帮助更好的理解。
假设本章的阅读者，都是已经比较熟练使用Python语言的编程人员，已经能够完全掌握Python语言的基础语法。

RF框架主要使用Python实现，所以这里只会看到针对Python语言的一些Tips。

模块搜索路径, 包(Package) 和 __init__.py

作者: 虞科敏

RF框架中将各功能以不同的Package组织起来，大致理解框架每个package的作用，是了解框架一个很好的入手点。
而了解每个Package的很好的入手点，又是 __init__.py, 里面的注释和代码都是很有价值的信息。
对于模块，包，初始化文件这些基础知识，是必备的基础。

模块导入

Python使用 import 和 from-import 语句导入模块。

python在执行import语句时，按照python的文档说明，它会执行了如下操作： 第1步，创建一个新的，空的module对象(它可能包含多个module)
第2步，把这个module对象插入sys.module中
第3步，装载module的代码（如果需要，首先必须编译）
第4步，执行新的module中对应的代码。

在执行第3步时，首先要找到module程序所在的位置，这就是所谓的"模块搜索"： 如果需要导入的module的名字是m1，则解释器目标就是找到m1.py
首先, 在当前目录查找 然后, 在环境变量PYTHONPATH中查找 (PYTHONPATH可以视为系统的PATH变量一类的东西，其中包含若干个目录) 如果PYTHONPATH没有设定，或者找不到m1.py，则继续搜索 与python的安装设置相关的默认路径
总结搜索的顺序为: **1.** 当前路径（以及从当前目录指定的**sys.path**），**2. PYTHONPATH**，**3. python**的安装设置相关的默认路径。

按这样的搜索顺序，如果当前路径或PYTHONPATH中存在与标准module同样的module，则会覆盖标准module。 也就是说，如果当前目录下存在xml.py，那么执 行import xml时，导入的是当前目录下的module，而不是系统标准的xml。

包(Package) 和 __init__.py

为了像Java的Package一样，将多个.py文件组织起来，以便在外部统一调用，在内部可以互相调用，会用到python的包的概念，而 __init__.py 在包里起着很重要的作用。

我们可以先构建一个package，以普通module的方式导入，就可以直接访问此package中的各个module。

Python中的package定义很简单，其层次结构与程序所在目录的层次结构相同，这一点与Java类似，唯一不同的地方在于，python中的package必须包含一个**init**.py的文件。

样例*1*
定义包 package1:

```
package1/
    __init__.py
    subPack1/
        __init__.py
        module_11.py
        module_12.py
        module_13.py
    subPack2/
        __init__.py
        module_21.py
        module_22.py
    ……
```

__init__.py 可以为空，只要它存在，就表明此目录应被作为一个package处理。 当然，__init__.py中也可以设置相应的内容。

样例*2*
按照包层次关系调用模块中的函数

在module_11.py中定义一个函数：

```
def funA():
    print "funcA in module_11"
    return
```

在顶层目录运行python（也就是package1所在的目录，当然按照模块搜索的知识，必须将package1放在解释器能够搜索到的路径上）

```
>>>from package1.subPack1.module_11 import funcA
>>>funcA()
funcA in module_11
```

这样，我们就按照package1的层次关系，正确调用了module_11中的函数。

样例*3*
__init__.py 中的 __all__

经常，在"from-import"语法中会出现通配符*，导入某个module中的所有元素，所有元素的范围如何指定呢？ 就是靠 __init__.py 中的 __all__ 指定。 比如在subPack1的 __init__.py 文件中这样定义

```
__all__ = ['module_13', 'module_12']
```

然后在python中执行之前的导入: "from-import *"语法中，package内的module是受**init**.py限制的

```
from package1.subPack1 import *
module_11.funcA()
Traceback (most recent call last):
  File "", line 1, in
ImportError: No module named module_11
```

样例*4*
如何在package内部互相调用

如果希望调用同一个package中的module，则直接import即可。也就是说，在module_12.py中，可以直接使用

```
import module_11
```

如果不在同一个package中，例如我们希望在module_21.py中调用module_11.py中的FuncA，则应该这样：

```
from <module_11包名>.module_11 import funcA
```

修饰器

作者: 虞科敏

RF框架中一些关键语法支持使用到修饰器, 比如框架中大名鼎鼎的@keyword，这种关键字理解不了何谈理解框架的机制？
实际上，在诸多的Python框架中，对于修饰器的理解都是绕不开的基础知识。
此乃阅读理解，必备佳肴 :)

Python中所有名字都是对象，函数也是对象。
装饰器其实也就是一个函数，一个用来包装原始函数的函数(或类)，返回一个修改之后的函数(或类)对象。 将修改后的函数(或类)对象重新赋值给原来的标识符，这样原标识符就代表修改后的函数(或类) (如此，也永久丧失对原始函数(或类)对象的访问）。

无参数修饰器

按样例: 函数deco是装饰函数，它的参数就是对装饰的函数对象。
在函数内定义了一个函数deco，这个函数对象会被返回。 实际使用中传入的参数会被传给这个函数对象。
*deco*中可以访问func函数对象，这是一个闭包的语法。

样例 无参数装饰器

```
def deco(func):
    def deco_(*args, **kargs):
        print "do something."
        func(*args, **kargs)
        print "do something 2."
    return deco_

@deco
def func(a, b, c=3, *args, **kargs):
    print "a:", a
    print "b:", b
    print "c:", c
    print args
    print kargs

if __name__ == '__main__':
    func(1, 2, 3, 4, 5, 6, x='x', y='y', z='z')
```

带参数装饰器

按样例: 第一个函数deco_with_p是装饰函数，它的参数作用是用来"定义具体如何装饰"的。
由于这些参数并非被装饰的函数对象，所以在内部必须创建一个接受被装饰函数的函数，然后返回这个函数对象实际上
此时实际调用: deco_with_p(11, 12, 13, 14, 15, 16, u='u', v='v', w='w')(1, 2, 3, 4, 5, 6, x='x', y='y', z='z')

样例 带参数装饰器

```
def deco_with_p(a1, b1, c1=3, *args_d, **kargs_d):
    def deco_(func):
        def deco__(*args, **kargs):
            print "do something."
            func(*args, **kargs)
            print "decorator's a1: ", a1
            print "decorator's b1: ", b1
            print "decorator's c1: ", c1
            print "decorator's args: ", args_d
            print "decorator's kargs: ", kargs_d
            print "do something 2."
        return deco__
    return deco_

@deco_with_p(11, 12, 13, 14, 15, 16, u='u', v='v', w='w')
def func_with_p(a, b, c=3, *args, **kargs):
    print "a:", a
    print "b:", b
    print "c:", c
    print args
    print kargs

if __name__ == '__main__':
    func_with_p()
```

修饰类的修饰器

根据修饰器的定义，修饰器不但能修饰函数，也能修饰类。
当然，我们碰到的修饰函数的修饰器的确要更多一些。

> RF框架中，还没有使用类修饰器的情况，所以本章节可做基本了解。 当然，本人对RF框架的功力有限，所以情况不属实，还请指正。

按样例: log_method_calls， 是一个修饰类的修饰器（不是修饰函数） 类装饰器将一个class作为输入参数(Python中的类类型对象)，并且返回一个修改过的class。 另外，logged()修饰成员函数，在log_method_calls()中对类的成员函数进行了修改，以达到对成员函数调用打印时间信息的目的。

```
def logged(time_format, name_prefix=""):
    def decorator(func):
        if hasattr(func, "_logged_decorator") and func._logged_decorator:
            return func

        @wraps(func)
        def decorated_func(*args, **kwargs):
            start_time = time.time()
            print '- Running "%s"; on %s' % (name_prefix + func.__name__, time.strftime(time_format))
            result = func(*args, **kwargs)
            end_time = time.time()
            print '- Finished "%s", execution time = %0.3fs ' % (name_prefix + func.__name__,end_time - start_time)

            return result
        decorated_func._logged_decorator = True
        return decorated_func
    return decorator

def log_method_calls(time_format):
    def decorator(cls):
        for o in dir(cls):
            if o.startswith("__"):
                continue
            a = getattr(cls, o)
            if hasattr(a, "__call__"):
                decorated_a = logged(time_format, cls.__name__ + '.')(a)
                setattr(cls, o, decorated_a)
        return cls
    return decorator

@log_method_calls('%b %d %Y - %H:%M:%S')
class A(object):
    def test1(self):
        print 'test1'

@log_method_calls('%b %d %Y - %H:%M:%S')
class B(A):
    def test1(self):
        super(B, self).test1()
        print 'child test1'

    def test2(self):
        print 'test2'

if __name__ == '__main__':
    b = B()
    b.test1()
    b.test2()
```

修饰器参数为类

这是一个网络上非常出名的示例，看点除了 1. 类作为修饰器参数; 2. 静态方法的使用，同时它也是一个实现同步保护的很好框架.

> RF框架中，还没有修饰器参数为类的情况，所以本章节可做基本了解。 当然，本人对RF框架的功力有限，所以情况不属实，还请指正。

```python
class locker:
    def __init__(self):
        print("locker.__init__() should be not called.")

    @staticmethod
    def acquire():
        print("locker.acquire() called.（这是静态方法）")

    @staticmethod
    def release():
        print("  locker.release() called.（不需要对象实例）")

def deco(cls):
    '''cls 必须实现acquire和release静态方法'''
    def _deco(func):
        def __deco():
            print("before %s called [%s]." % (func.__name__, cls))
            cls.acquire()
            try:
                return func()
            finally:
                cls.release()
        return __deco
    return _deco

@deco(locker)
def myfunc():
    print(" myfunc() called.")

if __name__ == '__main__':
    myfunc()
    myfunc()
```

迭代器

作者: 虞科敏

以下是来自《Python核心编程》的描述:
迭代器，为类序列对象(sequence-like)提供了一个类序列的接口.
序列, 是一组数据结构, 可以利用它们的索引从 0 开始一直"迭代" 到序列的最后一个条目.
Python 的迭代器无缝地支持序列对象, 而且它还允许程序员迭代非序列类型, 包括用户定义的对象.
迭代器用起来很灵巧, 你可以迭代不是序列但表现出序列行为的对象, 例如字典的 key , 一个文件的行, 等等.
当你使用循环迭代一个对象条目时, 你几乎不可能分辨出它是迭代器还是序列.
你不必去关注这些, 因为 Python 让它象一个序列那样操作.

从实现层面来看，迭代器是一个容器对象，它实现了迭代器协议，有两个基本方法：

1. next方法 - 返回容器的下一个元素
2. __iter__方法 - 返回迭代器自身

样例*1*
一个自定义的迭代器类

```
class MyIterator(object):
  def __init__(self, step):
      self.step = step
  def next(self):
  """Returns the next element."""
      if self.step==0:
          raise StopIteration
      self.step-=1
      return self.step
  def __iter__(self):
  """Returns the iterator itself."""
      return self

for el in MyIterator(4):
    print el
```

样例*2*
使用内建的iter方法创建迭代器

```
>>> i = iter('abc')
>>> i.next()
'a'
>>> i.next()
'b'
>>> i.next()
'c'
>>> i.next()
Traceback (most recent call last):
  File "", line 1, in
StopIteration:
```

生成器

作者: 虞科敏

从句法上讲，生成器是一个带 yield 语句的函数。
一个函数或者子程序只返回一次，但一个生成器能暂停执行并返回一个中间的结果 ———— 那就是 yield 语句的功能，返回一个值给调用者并暂停执行。 当生成器的 next()方法被调用的时候，它会准确地从离开地方（当它返回[一个值以及]控制给调用者时）继续。

样例*1*
只要函数中包含yield关键字，该函数调用就是生成器对象。
gen()并不是函数调用，而是产生生成器对象。

```
def gen():
    for x in xrange(4):
        tmp = yield x
        if tmp == 'hello':
            print 'world'
        else:
            print str(tmp)

def demo():
    import types
    g=gen()
    print g   #<generator object gen at 0x02801760>
    print isinstance(g, types.GeneratorType) #True

demo()
```

生成器对象支持几个方法，如gen.next() ,gen.send() ,gen.throw()等。

样例*2*
next()示例
第1次调用next(), 将运行到yield位置，此时暂停执行环境，并返回yield后的值。 所以打印出的是0，暂停执行环境。 第2次调用next()，第1次调用next()后,执行到yield 0暂停，再次执行恢复环境，给tmp赋值(注意：这里的tmp的值并不是x的值，而是通过send方法接受的值)，由于我们没有调用send方法，所以tmp的值为None, 此时 "print str(tmp)" 语句输出None，并执行到下一次yield x, 返回值1，"print g.next()" 所以又输出1.

```
def demo2():
    g=gen()
    print g.next() # 0
    print g.next() #None  1
```

样例*3*
send()示例
第1次调用next(), 执行到yield 0后暂停 然后，调用send('hello')，程序将收到"hello"，并给tmp赋值为"hello",此时语句 "if tmp=='hello':"为真，所以 "print 'world'" 输出'world', 并执行到下一次yield 1, 返回值1，"print g.next()" 所以又输出1.
next()等价于send(None)

```
def demo3():
    g=gen()
    print g.next() # 0
    print g.send('hello') #world  1
```

样例*4*

throw()示例

第1次调用next(), 执行到yield 'something'暂停, "print g.next()" 输出'something'

然后，然后调用throw(ValueError)向g发送ValueError异常，恢复执行环境，except语句将会捕捉到ValueError，执行到yield 'value error'暂停, 执行finally语句。

```
def mygen():
    try:
        yield 'something'
    except ValueError:
        yield 'value error'
    finally:
        print 'clean'  #一定会被执行

def demo4():
    g=mygen()
    print g.next() #something
    print g.throw(ValueError) #value error  clean
```

包装和授权

作者: 虞科敏

包装(Wrap)是在Python世界经常被提到的一个术语。
其思想大致就是希望将某个对象（或类），进行重新打包，增加新功能，删除不需要的功能，或者修改已有的功能，装换成另外一种更适合当前使用场合的对外接口。
学习过设计模式的同学，这就是适配器的思想。
以"组合优先于继承" best practice来讲，如果我们要包装一个对象，最好选择组合的手段，比较典型的做法会是这样: 创建一个类，它往往只包含一个数据成员对象，就是它要封装的类的对象。 通俗的讲，就是让这个封装类包含这个被封装类的实例。 然后通过在这个封装类中各种方法的设置，实质上就是从这个包含对象的接口传递、转换其相应的功能到这个包装类上。 让外界感觉似乎就是在操作该被封装的类一样。

而授权是包装的一个特性，授权的过程，就是: 将所有更新的功能都由新类的新方法来处理，但已存在的其他功能就授权给原对象的默认属性。 而实现这一功能的关键就是 __getattr__() 特殊方法。
__getattr__() 的工作方式大致如下:
当引用一个属性时，Python 解释器的查找顺序是这样:

1. 在局部名称空间中搜索该名字，比如自定义的方法或局部实例属性；
2. 如果没有在局部字典中找到，则搜索类名称空间，比如本类的类属性，父类的类属性；
3. 如果两类搜索都失败了，搜索则对原对象，调用 __getattr__()。

因此这里就给我们提供了一个将属性访问重定向到被包装的类型（伪父类）的机会。 其具体的实现方法便是，在 __getattr__() 中调用内建的 getattr() 函数。

样例

```
from time import time,ctime
class TimedWrapMe(object):
    def __init__(self,obj):
        self.__data=obj
        self.__ctime=self.__mtime=self.__atime=time()
    def get(self):
        self.__atime=time()
        return self.__data
    def gettimeval(self,t_type):
        if not isinstance(t_type,str) or t_type[0] not in 'cma':
            raise TypeError("argument of 'c','m','a' req'd' ")
        print('_%s__%stime' % (self.__class__.__name__,t_type[0]))
        return getattr(self,'_%s__%stime' % (self.__class__.__name__,t_type[0]))#attention!get private data attribute
    def gettimestr(self,t_type):
        return ctime(self.gettimeval(t_type))
    def setr(self,obj):
        self.__data=obj
        self.__mtime=self.__atime=time()
    def __repr__(self):
        self.__atime=time()
        return 'self.__data'
    def __str__(self):
        self.__atime=time()
        return str(self.__data)
    def __getattr__(self, item):
        self.__atime=time()
        return getattr(self.__data,item)
a=TimedWrapMe(9)
print(getattr(a,'_TimedWrapMe__ctime'))
print(a.gettimestr('c'))
print(repr(a))
```

描述符

作者: 虞科敏

描述符是Python语言中一个深奥但却重要的一部分。
在RF框架中，不少地方用到了描述符技术。如果对描述符不理解，这些地方的代码将变得晦涩难懂。
以下内容大多摘录自《Python核心编程》。

描述符是 Python 新式类中的关键点之一。它为对象属性提供强大的 API。你可以认为描述符是表示对象属性的一个代理。当需要属性时，可根据你遇到的情况，通过描述符（如果有）或者采用常规方式（句点属性标识法）来访问它。

如你的对象有代理，并且这个代理有一个“get”属性(实际写法为**get**)，当这个代理被调用时，你就可以访问这个对象了。
当你试图使用描述符(set)给一个对象赋值或删除一个属性(delete)时，这同样适用。

__get__(), __set__(), __delete__() 特殊方法

严格来说，描述符实际上可以是任何（新式）类，这种类至少实现了三个特殊方法 __get__(), __set__(), __delete__() 中的一个 ———— 这三个特殊方法充当描述符协议的作用。
刚才提到过，__get__()可用于得到一个属性的值，__set__()是为一个属性进行赋值的，在采用 del 语句（或 其它，其引用计数递减）明确删除掉某个属性时会调__delete__()方法。三者中，后者很少被实现。
还有，也不是所有的描述符都实现了__set__()方法。它们被当作方法描述符，或更准确来说是，非数据描述符来被引用。
那些同时覆盖__get__()及__set__()的类被称作数据描述符，它比非数据描述符要强大些。

__get__(), __set__(), __delete__() 原型签名如下

```
def __get__(self, obj, typ=None) ==> value
def __set__(self, obj, val) ==> None
def __delete__(self, obj) ==> None
```

如果你想要为一个属性写个代理，必须把它作为一个类的属性，让这个代理来为我们做所有的工作。当你用这个代理来处理对一个属性的操作时，你会得到一个描述符来代理所有的函数功能。
现在让我们来处理更加复杂的属性访问问题，而不是将所有任务都交给你所写的类中的对象们。

__getattribute__() 特殊方法

使用描述符的顺序很重要，有一些描述符的级别要高于其它的。
整个描述符系统的心脏是 __getattribute__()，因为对每个属性的实例都会调用到这个特殊的方法。这个方法被用来查找类的属性，同时也是你的一个代理，调用它可以进行属性的访问等操作。

回顾一下上面的原型，如果一个实例调用了 __get__()方法，这就可能传入了一个类型或类的对象。
举例来说，给定类 X 和实例 x, x.foo 由 **getattribute**() 转化成：

```
type(x).__dict__['foo'].__get__(x, type(x))
```

如果类调用了 __get__() 方法，那么 None 将作为对象被传入(对于实例， 传入的是 self)：

```
X.__dict__['foo'].__get__(None, X)
```

最后，如果 super()被调用了，比如，给定 Y 为 X 的子类，然后用 super(Y,obj).foo 在 obj.**class**.**mro**中紧接类 Y 沿着继承树来查找类 X，然后调用：

```
X.__dict__['foo'].__get__(obj, X)
```

然后，描述符会负责返回需要的对象。

优先级别

由于 __getattribute__() 的实现方式很特别，我们在此对 __getattribute__() 方法的执行方式 做一个介绍。因此了解以下优先级别的排序就非常重要了：

1. 类属性
2. 数据描述符
3. 实例属性
4. 非数据描述符
5. 默认为**getattr**()

描述符是一个类属性，因此所有的类属性皆具有最高的优先级。你其实可以通过把一个描述符的引用赋给其它对象来替换这个描述符。
比它们优先级别低一等的是实现了**get**()和**set**()方法的描述符。如果你实现了这个描述符，它会像一个代理那样帮助你完成所有的工作！
否则，它就默认为局部对象的**dict**的值，也就是说，它可以是一个实例属性。
接下来是非数据描述符。可能第一次听起来会吃惊，有人可能认为在这条“食物链”上非数据描述符应该比实例属性的优先级更高，但事实并非如此。非数据描述符的目的只是当实例属性值不存在时，提供一个值而已。这与以下情况类似：当在一个实例的__dict__中找不到某个属性时，才去调用__getattr__()。
关于__getattr__()的说明，如果没有找到非数据描述符，那么__getattribute__()将会抛出一 个 AttributeError 异常，接着会调用**getattr**()做为最后一步操作，否则 AttributeError 会返回给用户。

样例

使用**Python**描述符实现对象属性的类型检查

```
class TypedProperty(object):
    def __init__(self, name, type, default=None):
        self.name = "_" + name
        self.type = type
        self.default = default if default else type()

    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default)

    def __set__(self,instance,value):
        if not isinstance(value,self.type):
            raise TypeError("Must be a %s" % self.type)
        setattr(instance,self.name,value)

    def __delete__(self,instance):
        raise AttributeError("Can't delete attribute")

class Foo(object):
    name = TypedProperty("name",str)
    num = TypedProperty("num",int,42)

acct = Foo()
acct.name = "obi"
acct.num = 1234
print acct.num #1234
print acct.name #obi
# trying to assign a string to number fails
acct.num = '1234' #TypeError: Must be a <type 'int'>
```

在这个例子中，我们实现了一个描述符TypedProperty，并且这个描述符类会对它所代表的类的任何属性执行类型检查。注意到这一点很重要，即描述符只能在类级别进行合法定义，而不能在实例级别定义。例如，在上面例子中的__init__方法里就不可以。

当访问类Foo实例的任何属性时，描述符会调用它的__get__方法。需要注意的是，__get__方法的instance参数是描述符代表的属性被引用的源对象。
当属性被分配时，描述符会调用它的__set__方法。为了理解为什么可以使用描述符代表对象属性，我们需要理解Python中属性引用解析的执行方式。对于对象来说，属性解析机制在object.**getattribute**()中。该方法将b.x转换成type(b).__dict__['x'].__get__(b, type(b))。然后，解析机制使用优先级链搜索属性，在优先级链中，类字典中发现的数据描述符的优先级高于实例变量，实例变量优先级高于非数据描述符，如果提供了getattr()，优先级链会为getattr()分配最低优先级。对于一个给定的对象类，可以通过自定义__getattribute__方法来重写优先级链。

深刻理解优先级链之后，就很容易想出针对前面提出的第二个和第三个问题的优雅解决方案了。那就是，利用描述符实现一个只读属性将变成实现数据描述符这个简单的情况了，即不带__set__方法的描述符。尽管在本例中不重要，定义访问方式的问题只需要在__get__和__set__方法中增加所需的功能即可。

类属性**property**

```
class Accout(object):
    def __init__(self):
        self._acct_num = None

    def get_acct_num(self): # self is instance argument in __get__(self, instance, cls)
        return self._acct_num

    def set_acct_num(self, value): # self, see get_...
        self._acct_num = value

    def del_acct_num(self): # self, see get_...
        del self._acct_num

    acct_num = property(get_acct_num, set_acct_num, \
                        del_acct_num, "Account number property.")
```

每次我们想使用描述符的时候都不得不定义描述符类，这样看起来非常繁琐。Python特性提供了一种简洁的方式用来向属性增加数据描述符。一个属性签名如下所示：

```
property(fget=None, fset=None, fdel=None, doc=None) -> property attribute
```

如果acct是Account的一个实例，acct.acct_num将会调用getter，acct.acct_num = value将调用setter，del acct_num.acct_num将调用deleter。

在Python中，属性对象和功能可以使用描述符协议来实现，如下所示:

```
class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)

    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel, self.__doc__)

    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel, self.__doc__)

    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel, self.__doc__)
```

@property装饰器

```
class C(object):
    def __init__(self):
        self._x = None

    @property
     # the x property. the decorator creates a read-only property
    def x(self):
        return self._x

    @x.setter
    # the x property setter makes the property writeable
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

Python也提供了@ property装饰器，可以用它来创建只读属性。
一个属性对象拥有getter、setter和deleter装饰器方法，可以使用它们通过对应的被装饰函数的accessor函数创建属性的拷贝。
如果我们想让属性只读，那么我们可以去掉setter方法。

__slots__类属性

RF框架中有不少类使用 __slots__，而不是__dict__。了解一下这是为什么吧。

字典是实例的“心脏”。
__dict__属性跟踪所有实例属性。
举例来说，你有一个实例 inst. 它有一个属性 foo，那使用 inst.foo 来访问它与使用 inst.__dict__['foo']来访问是一致的。
但字典 __dict__ 会占据大量内存，如果你有一个属性数量很少的类，但有很多实例，那么正好是这种情况。 为内存上的考虑，用户现在可以使用__slots__属性来替代__dict__。
基本上，__slots__是一个类变量，由一序列型对象组成，由所有合法标识构成的实例属性的集合来表示。它可以是一个列表，元组或可迭代对象。也可以是标识实例能拥有的唯一的属性的简单字符串。任何试图创建一个其名不在__slots__中的名字的实例属性都将导致 AttributeError 异常：

```
class SlottedClass(object):
__slots__ = ('foo', 'bar')
 c = SlottedClass()

 c.foo = 42
 c.xxx = "don't think so" Traceback (most recent call last):
File "", line 1, in ?
AttributeError: 'SlottedClass' object has no attribute
'xxx'
```

这种特性的主要目的是节约内存。其副作用是某种类型的"安全",它能防止用户随心所欲的动态增加实例属性。带__slots__属性的类定义不会存在__dict__了（除非你在__slots__中增加'__dict__'元素）。

CI自动化

作者: 虞科敏

Jenkins集成

建设中...

移动端测试库

作者: 虞科敏

AndroidLibrary

建设中...

iOSLibrary

建设中...

appiumLibrary

建设中...