

Table of Contents

简介	0
Chapter 1 简明使用手册	1
1-1 测试数据, Test Data	1.1
1-2 测试用例, Test Case	1.2
1-3 测试模板, Test Template	1.3
1-4 测试套件, Test Suite	1.4
1-5 测试库, Test Library	1.5
1-6 变量, Variable	1.6
1-6-1 创建变量, Creating Variable	1.7
1-6-2 变量文件, Variable Files	1.8
1-6-3 内建变量, Build-in Variables	1.9
1-6-4 变量的属性和作用范围	1.10
1-6-5 变量的高级特性	1.11
1-7 用户关键字, User Keyword	1.12
1-7-1 用户关键字参数	1.13
1-7-2 变量嵌入到关键字名中	1.14
1-7-3 用户关键字返回值	1.15
1-7-4 用户关键字Teardown	1.16
1-8 资源文件, Resource File	1.17
Chapter 2 高级特性	2
Chapter 3 实现测试库	3
3-1 测试库	3.1
3-2 创建测试库:类或模块	3.2
3-2-1 测试库名	3.3
3-2-2 测试库的参数	3.4
3-2-3 测试库的作用范围	3.5
3-2-4 测试库版本	3.6
3-2-5 测试库的文档说明	3.7
3-2-6 测试库工作为Listener	3.8
3-3 静态API	3.9
3-3-1 关键字的名字	3.10
3-3-2 关键字的标签	3.11
3-3-3 关键字的参数	3.12
3-3-4 关键字的参数缺省值	3.13
3-3-5 关键字的可变长参数	3.14
3-3-6 关键字的自由参数	3.15
3-3-7 关键字的参数类型	3.16

3-3-8 使用修饰器	3.17
3-4 和RF框架通信	3.18
3-4-1 停止和继续测试执行	3.19
3-4-2 日志消息	3.20
3-4-3 Logging编程API	3.21
3-4-4 测试库初始化阶段Logging	3.22
3-4-5 返回值	3.23
3-4-6 线程间通信	3.24
3-5 测试库的发布	3.25
3-6 动态API	3.26
3-6-1 获取关键字名	3.27
3-6-2 关键字的运行	3.28
3-6-3 获取关键字参数	3.29
3-6-4 获取关键字的文档信息	3.30
3-6-5 命名参数语法	3.31
3-6-6 自由参数语法	3.32
3-6-7 小结	3.33
3-7 混合型API	3.34
3-8 扩展测试库	3.35
Chapter 4 常用测试库	4
4-1 RF内部模块和内建测试库	4.1
4-1-1 BuiltIn库	4.2
4-1-2 Collections库	4.3
4-1-3 DateTime库	4.4
4-1-4 Dialogs库	4.5
4-1-5 OperatingSystem库	4.6
4-1-7 Screenshot库	4.7
4-1-8 String库	4.8
4-1-9 Telnet库	4.9
4-1-10 XML库	4.10
4-2 第3方测试库	4.11
4-2-1 Web自动化	4.12
4-2-2 GUI自动化	4.13
4-2-3 DB自动化	4.14
4-2-4 接口自动化	4.15
4-2-5 移动端自动化	4.16
4-2-6 敏捷自动化	4.17
Chapter 5 框架分析	5
5-1 语言Tips	5.1
5-2 框架结构	5.2

5-3 主流程	5.3
Chapter 6 扩展框架	6
Chapter 7 工具	7

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

- [1-1 测试元素, Test Data](#)
- [1-2 测试用例, Test Case](#)
- [1-3 测试模板, Test Template](#)
- [1-4 测试套件, Test Suite](#)
- [1-5 测试库, Test Library](#)
- [1-6 变量, Variable](#)
 - [1-6-1 创建变量, Creating Variable](#)
 - [1-6-2 变量文件, Variable Files](#)
 - [1-6-3 内建变量, Build-in Variables](#)
 - [1-6-4 变量的属性和作用范围](#)
 - [1-6-5 变量的高级特性](#)
- [1-7 用户关键字, User Keyword](#)
 - [1-7-1 用户关键字参数](#)
 - [1-7-2 变量嵌入到关键字名中](#)
 - [1-7-3 用户关键字返回值](#)
 - [1-7-4 用户关键字 Teardown](#)
- [1-8 资源文件, Resource File](#)

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 Kwarg
  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 Kwarg
  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 failure**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 workflow
 - 若干关键字和他们必要的参数
- 数据驱动
 - 针对相同 workflow，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 workflow，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 workflow
 - 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 ***
Invalid User Name      USERNAME      PASSWORD
invalid                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 Directories

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__sometests.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Test 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Variable Table
- 在Variable 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 \${MANY3}

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, Variable 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, Variable 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Variable 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

命令行选项 --variablefile 也可以使用Variable File

从命令行导入的Variable File, 作用域范围是全局可用的。这个通过--variable选项设置的变量情况是相似的。如果通过--variablefile 和 --variable 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。Variable 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 &{MAPPING}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(getvariables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Variable 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

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

Java版本

```
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
<code>\${TEST NAME}</code>	当前测试用例的名字	Test case
<code>@{TEST TAGS}</code>	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
<code>\${TEST DOCUMENTATION}</code>	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
<code>\${TEST STATUS}</code>	当前测试用例的状态: Pass 或 FAIL	Test teardown
<code>\${TEST MESSAGE}</code>	当前测试用例的消息	Test teardown
<code>\${PREV TEST NAME}</code>	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
<code>\${PREV TEST STATUS}</code>	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
<code>\${PREV TEST MESSAGE}</code>	前一个测试用例的错误消息	Everywhere
<code>\${SUITE NAME}</code>	当前Suite的全名	Everywhere
<code>\${SUITE SOURCE}</code>	Suite的文件或目录的绝对路径	Everywhere
<code>\${SUITE DOCUMENTATION}</code>	当前测试Suite的文档说明。可以使用" Set Suite Documentation "关键字修改	Everywhere
<code>&{SUITE METADATA}</code>	当前测试Suite的元数据。可以使用" Set Suite Metadata "关键字修改	Everywhere
<code>\${SUITE STATUS}</code>	当前测试Suite的状态: Pass 或 FAIL	teardown
<code>\${SUITE MESSAGE}</code>	当前测试Suite的消息, 包括统计信息	Suite teardown
<code>\${KEYWORD STATUS}</code>	当前测试关键字的状态: Pass 或 FAIL	User keyword teardown
<code>\${KEYWORD MESSAGE}</code>	当前测试关键字的错误消息	User keyword teardown
<code>\${LOG LEVEL}</code>	当前的日志级别	Everywhere
<code>\${OUTPUT FILE}</code>	输出(output)文件的绝对路径	Everywhere
<code>\${LOG FILE}</code>	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
<code>\${REPORT FILE}</code>	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
<code>\${DEBUG FILE}</code>	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
<code>\${OUTPUT DIR}</code>	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 **Variable 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 **Resource 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 **Variable 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 **Resource 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 **Variable 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:

Variable 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}
```

Keyword 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`${animal}=cat`

"`Select dog from list`", 那么`${animal}=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: `${arg:regex}`
比如, 参数值只能匹配数字, 可以写为: `${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Teardown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。

- [2-1 处理同名关键字](#)
- [2-2 超时机制](#)
- [2-3 For循环](#)
- [2-4 有条件地执行](#)
- [2-5 并行执行](#)

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。

- 3-1 测试库
- 3-2 创建测试库:类或模块
 - 3-2-1 测试库名
 - 3-2-2 测试库的参数
 - 3-2-3 测试库的作用范围
 - 3-2-4 测试库版本
 - 3-2-5 测试库的文档说明
 - 3-2-6 测试库工作为Listener
- 3-3 静态API
 - 3-3-1 关键字的名字
 - 3-3-2 关键字的标签
 - 3-3-3 关键字的参数
 - 3-3-4 关键字的参数缺省值
 - 3-3-5 关键字的可变长参数
 - 3-3-6 关键字的自由参数
 - 3-3-7 关键字的参数类型
 - 3-3-8 使用修饰器
- 3-4 和RF框架通信
 - 3-4-1 停止和继续测试执行
 - 3-4-2 日志消息
 - 3-4-3 Logging编程API
 - 3-4-4 测试库初始化阶段Logging
 - 3-4-5 返回值
 - 3-4-6 线程间通信
- 3-5 测试库的发布
- 3-6 动态API
 - 3-6-1 获取关键字名
 - 3-6-2 关键字的运行
 - 3-6-3 获取关键字参数
 - 3-6-4 获取关键字的文档信息
 - 3-6-5 命名参数语法
 - 3-6-6 自由参数语法
 - 3-6-7 小结
- 3-7 混合型API
- 3-8 扩展测试库

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: "YourLibrary", 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. (Pythener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

```

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

```

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() {
    }
}
```

测试库工作作为**Listener**

Listener接口，允许外部**Listeners**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**Listener**。

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**Listener**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, `currentthread` 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，并且空格和下划线_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，之前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"robot_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
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修饰器有时确实会非常方便。

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

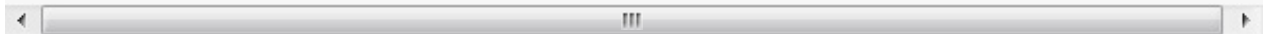
Python测试库，在导入和初始化过程中，使用logging编程API

```
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

```
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

```
/**
 * 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Library 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	<code>[]</code>	0//0
1个或多个	包含参数名的list	<code>[one_argument], [a1, 'a2', 'a3']</code>	1//1, 3//3
参数缺省值	List中的元素" name=value"	<code>[arg=default value], [a, 'b=1', 'c=2']</code>	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*	<code>[varargs], [a, 'b=42', rest]</code>	0//any, 1//any
自由参数 (kwargs)	最后1个参数名前加**	<code>[kwargs], [a, 'b=42', kws], [varargs, *kwargs]</code>	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

- 4-1 RF内部模块和内建测试库
 - 4-1-1 BuiltIn库
 - 4-1-2 Collections库
 - 4-1-3 DateTime库
 - 4-1-4 Dialogs库
 - 4-1-5 OperatingSystem库
 - 4-1-6 Process库
 - 4-1-7 Screenshot库
 - 4-1-8 String库
 - 4-1-9 Telnet库
 - 4-1-10 XML库
- 4-2 第3方测试库
 - 4-2-1 Web测试库
 - 4-2-2 GUI测试库
 - 4-2-3 数据库测试库
 - 4-2-4 接口测试库
 - 4-2-5 移动端测试库
 - 4-2-6 敏捷自动化

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。

请记住，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

Autolt 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。

- [5-1 语言Tips](#)
- [5-2 框架结构](#)
- [5-3 主流程](#)

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。

包(Package)的 `__init__.py`

修饰符

迭代器

包装和授权

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，转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: 将所有的更新的功能都是由新类的新方法来处理，但已存在的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

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...