Robot Framework 用户指南

Version 2.5.3

•• 翻译 ••

黄志中
任星伟
黎守秀
耿立直
国庆波

•• 校对 ••

张晓红
黎守秀
代锦秀
耿立直
任星伟

•• 整理 ••

耿立直

• 译者注 •

由于译者水平有限且有可能对Robot Framework理解不够深入，因此不能保证译文准确无误，敬请各位读者谅解。如果您发现了译文中有错漏之处，也请您指出，哪怕是错别字也好，请将错漏之处发送至huangzz_8011@163.com，谢谢。

2010 年 12 月 29 日

目录

1 开始

1.1 简介

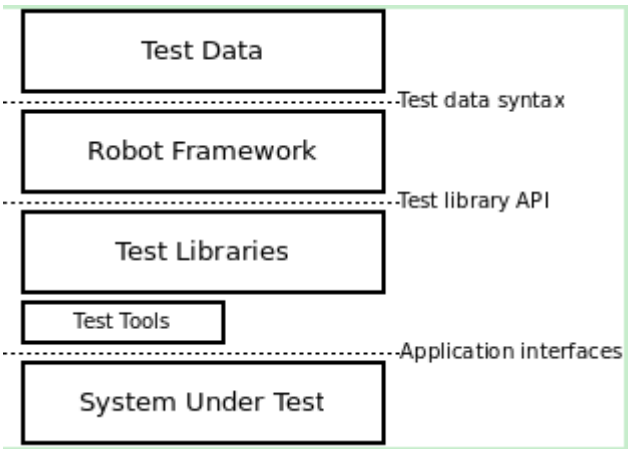
Robot Framework 是一种基于 Python 的可扩展关键字驱动自动化测试框架，通常用于端到端的可接收测试和可接收测试驱动的开发。可以用于测试声明涉及到多种技术和接口的分布式的，异构的应用系统。

1.1.1 为什么使用 Robot Framework

- 易于使用：它采用一种表格式的语法，易于创建统一格式的测试用例。
- 重用性好：可以重复利用已经存在的关键字来创建高层次关键字。
- 结果报告和日志采用 HTML 格式，易于阅读。
- 平台与被测系统应用程序相互独立。
- 提供了简单的库函数 API 用于创建用户自定义的基于 Python 或者 Java 的测试库，这些测试库可以被应用于本地应用。
- 为已经存在的构建设施（持续集成系统）提供了命令行接口和基于 XML 的输出文件。
- 为 Selenium 提供支持用户 WEB 测试，Java GUI 测试，启动进程，终端，SSH 等等。
- 支持创建基于数据驱动的测试用例。
- Built-in 支持不同环境下的特殊变量。
- 提供标签以分类和选择将被执行的测试用例。
- 源文件易于集成：测试集就是指可以被产品代码描述的文件和路径。
- 提供测试用例和测试集级别的 setup 和 teardown。
- 模块化的结构甚至支持为有多种接口的应用程序创建测试数据。

1.1.2 高层的架构

Robot Framework 是一种通用的，应用和技术相互独立的框架。是一种如下图所示的模块化结构。

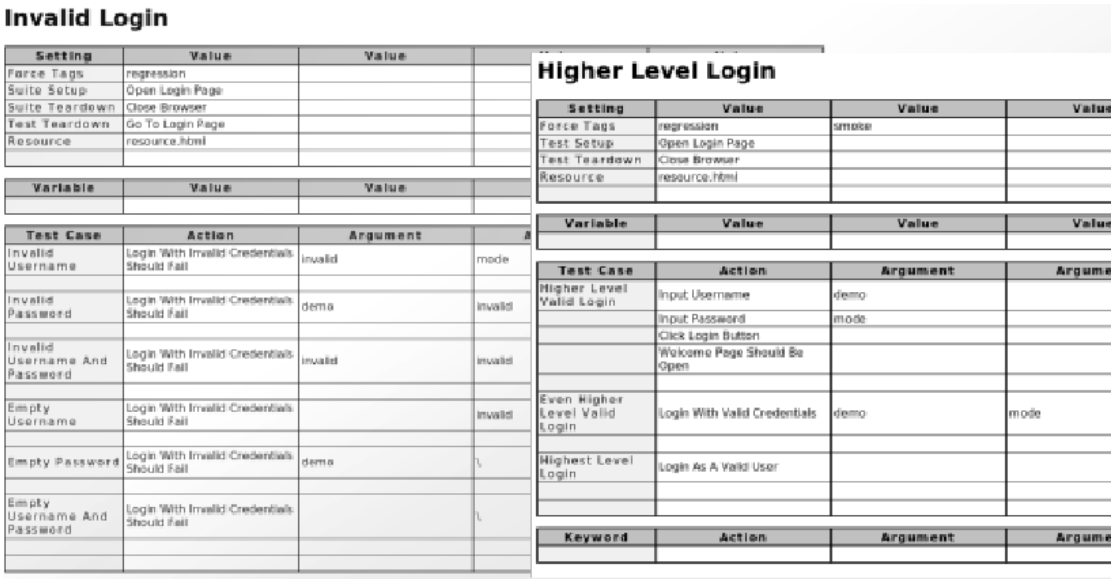


Robot Framework architecture

测试数据以一种简单易于编辑的表格格式。当 Robot Framework 启动的时候，启动测试数据，执行测试用例并且生成日志和报告。核心框架不知道任何关于被测目标系统的细节，核心框架句柄与被测系统通过测试库进行交互。测试库能够直接使用应用程序接口或者使用更低层次的测试工具作为驱动。

1.1.3 屏幕截图

以下屏幕截图展示了测试数据和创建的报告和日志的例子。



测试用例文件

报告和日志

1.1.4 获取更多信息

工程页面

获取更多关于 Robot Framework 信息的网址 http://robotframework.org.

用户指导手册当然能够在哪里获取，你也可以找到更多的文档，一个版本跟踪器，下载，源代码和其他相关工程的一些链接。Robot Framework 被托管于为开源项目提供优秀免费服务的谷歌代码。

邮件列表

这里有大量你可以获取更多关于 Robot Framework 信息的邮件列表。邮件列表档案对每个人都开放（包括搜索引擎）并且每个人都可以免费加入这个列表中。只有列表中的成员能够发送邮件，然而为了阻止垃圾邮件，新用户注册比较缓慢，这就意味着在你的第一条消息通过之前会花一定的时间。不要害怕发送问题到邮箱列表中，但是记住怎么用一种巧妙的方式咨询问题。

Robot Framework-用户

通常讨论关于 Robot Framework 相关的问题。疑问和难题都可以发送到这个邮件列表。所有用户将分享这些信息。

Robot Framework-公告

一个只读公告邮件列表仅有管理员能够发送消息。所有公告被发送到 Robot Framework 的用户邮件列表当中，所有没必要同时加入两个列表中。

Robot Framework-开发

讨论关于 Robot Framework 的开发。

Robot Framework-承诺

自动生成关于提交到版本控制系统的邮件，编译结果，新编辑的出版物等等。能够用于跟踪 Robot Framework 的发展。

1.2 版权与许可证

Robot Framework 自身，测试库和支持工具与版权和许可证没联系，和用户指导手册一样其他提供的文档有以下版权申明。

1.3 安装和卸载

1.3.1 简介

有几种方法安装 Robot Framework：

源文件安装

你可以直接从版本控制器或者从某个地方导出的分包上获取源文件代码。在前一种情况下，首先在某个路径下解压该文件，所以你有了一个名为 robotframwork-<version>的路径。详细的指令也在制定的路径下面，但是简而言之，你需要做的就是到创建的路径下运行 python setup.py install 命令。

利用窗体安装

有一种为 windows 提供的特别的图形化安装。

简易安装

如果 Python 包管理工具的简易安装可用，安装 Robot Framework 就是运行一个 easy_install robotframework 命令那么简单。

利用一键安装

如果你使用的是 windows XP 并且你没安装好先前的运行条件（Python 和可选的 Jython），你可以使用一键安装来安装你需要的安装程序。

单独的 Jar 分布

如果你仅仅想用 Jython 运行测试，安装所有的东西最方便的方法就是下载只包含 Jython 和 Robot Framework 的安装包 robotframework-<version>.jar。

从 http://downloads.robotframework.org 可以获取安装包，从 http://source.robotframework.org 获取源代码。

1.3.2 安装必备条件

Robot Framework 运行于 Python 或者 Jython，你至少需要会使用两者中的一种。然而，一些提供的安装程序必须有 Python 的时候才能运行，所以总是推荐安装 Python。

安装 Python

从 Robot Framework 2.5 开始，Python2.5 是最低的支持版本。早期的版本支持 Python2.3 或者更新的。在很多 UNIX 类似的操作系统，已经默认安装了 Python。如果是在 windows 操作系统上你需要自己安装 Python，你最好的寻找资源的地方应该是 Python homepage。在 Python 主页你能够下载适合的安装程序并且获得更多的关于安装源程序和 Python 的信息。

> 小贴士
>
> Robot Framework 与 Python 3.X 版本不匹配，因为 Python 3.X 不向后兼容支持以前发布的版本。

> 小贴士

在 windows 操作系统上，特别是在 windows vista 操作系统上，推荐所有用户安装 Python，并且作为管理员运行安装程序。

安装 Jython

运行由需要 Java 实时运行环境的 Java 开发测试库或者 Java 工具直接需要在 Jython 上运行 Robot Framework，Jython2.5 需要 Java1.5 或者更新的版本。SUN 公司和 IBM 公司的 Java 安装都支持。

从 Robot Framework 2.5 开始，最低支持 Jython 版本是 2.5。早期的 Robot Framework 版本也支持 Jython 2.2。

安装 Jython 是一个非常简单的过程，第一步是在 Jython homepage 获取安装源文件。注意安装源程序是一个可执行的 JAR 包，你需要运行 java -jar jython_installer-<version>.jar。依赖于你的系统，安装源程序可以是图形化的或者文本的模式，但是在两种情况下，安装程序都非常的简单。

以下有几种能够运行 Jython 的 Robot Framework 安装方法

1、所有支持 Jython 的 robot framework 的版本都可以用 python 来安装，安装后可以使用 Jython 来启动。当利用 Python 安装 Robot Framework，安装源文件试图在系统中寻找可以运行的 Jython，如果找到 Jython 就能正确创建 jybot 运行脚本。安装源文件按以下方式寻找 Jython：

- Jython 能在系统中直接运行(i.e 在环境变量中)
- 环境变量 JYTHON_HOME 需要设置，并且指向 Jhthon 的安装路径。
- 安装源文件从系统中寻找 Jhthon 安装路径。在 windows 系统中，从 *C:* and *D:* drives 下寻找，其他的操作系统从*/usr/local* and */opt* 路径下寻找。如果文件在前面提到的路径中会在该路径或者路径的下一层被找到。例如，如下所示的 Jython 安装源路径会被安装程序找到：

```
C:\APPS\Jython2.5.1
D:\Jython251
/usr/local/jython2.5.1
/opt/whatever/Jython251
```

2、利用 Jython 2.5，可以在没有从源文件安装 Python 的情况下安装 Robot Framework。

3、可以执行手动安装。

1.3.3 安装

源文件安装

你可以直接从版本控制器或者从某个地方导出的分包上获取源文件代码。在这两种情况下你需要有一个文件夹来存放源代码，文档，工具，例子等等。

你需要安装 Robot Framework 在 Python 源代码能够运行的环境中。在之前导入 Robot Framework 的路径下使用如下命令将完成安装：

```
python setup.py install
python install.py install
```

setup.py 是标准的 Python 安装脚本，这个脚本能够运行很多允许的参数，例如，安装到默认的路径不需要管理员权限，也可以用于创建不同的分包。

install.py 是典型的 Robot Framework 卸载脚本。也可以用于安装，但是通常还是使用 **setup.py**，然而这几种操作都等价于以上提到的命令行操作。

安装 Robot Framework 2.5 也可能用到 Jython。在这种情况下 **pybot** 安装脚本将不会被生成。

安装完成后你将得到一个相当长的输出，并且在最后会出现如下文档所示的东西。实际输出明显的取决于你的环境。

```
Creating Robot Framework runner scripts...
Installation directory: /usr/lib/python2.5/site-packages/robot
Python executable: /usr/bin/python
Jython executable: /cygdrive/c/jython2.5/jython.bat (found from system)
Pybot script: /usr/bin/pybot
Jybot script: /usr/bin/jybot
Rebot script: /usr/bin/rebot
```

采用 windows 安装程序安装

安装源程序仅仅需要双击提供的窗体安装程序根据向导进行安装。安装完成后你需要设置环境使得 Robot

Framework 的运行脚本能够被很容易的执行。请注意框架将被自动的安装到 Python 安装文件所在的路径，并且该路径不能更改。

小贴士

使用 Python 2.6，运行 Robot Framework 安装程序之前，需要将 Python 的安装路径设置到 PATH 这个环境变量中。设置环境部分由很多关于设置环境变量的信息。

小贴士

在 Windows Vista 操作系统上安装 Robot Framework 需要管理员权限。启动安装的时候从上下文的菜单中选择作为管理员进行安装。

使用简易安装

简易安装的一个很明显的先决条件就是你已经安装过 Robot Framework，并且你能够参阅一些相关的文档在你的操作系统上应该如何安装。安装 Robot Framework 的命令取决于你想安装的最近的版本还是一些特殊的版本：

```
easy_install robotframework              # latest version
easy_install robotframework==2.1.3    # specified version
```

如果你需要使用代理登陆 Internet，你可以告诉简易安装程序使用已经设置好的 **http_proxy** 环境变量。

在 Windows 操作系统上，使用简易安装 **pybot**，**jybot** 和 **rebot** 运行脚本不会被更新。简易安装完成后需要在工作区手动运行 **robot_postinstall.py** 脚本。安装程序的输出会告诉我们后来的简易安装运行完成后需要运行的脚本路径，你可以双击该脚本或者从命令行执行该脚本。脚本所在的路径取决于 Python 的安装程序，但是命令行运行必须是这样的格式 **python C:\Python25\Scripts\robot_postinstall.py.**

通常情况下更新安装脚本是一件很繁琐的事情，并且采用简易安装的时候它不是在所有的环境下都能工作。如果安装完成后运行脚本不起作用，需要进行手动的修改。

使用一键安装

一键安装 Robot Framework 的前提条件是已经安装了 Python 和 Jython(可选的)。这种安装方式自动设置环境

使得 Robot Framework 运行脚本，同事 Python 和 Jyhton 可执行文件在路径当中。

小贴士

一键安装仅仅适合于 windows XP 操作系统。

运行一键安装需要下载完整的安装程序，并且所有的安装程序都在同一个路径下。更多关于安装程序的详细介绍可以在 One Click Installer's documentation 中获取。

备注

只是在没安装 Python 或者 Jython 的情况下，可以使用这种安装方式。在这种情况下，如果你想定制安装程序，可以分别安装各个组件。

手动安装

如果你不想安装 Python，或者其他原因使得你不想使用 Robot Framework 的自动化安装方式，你可以按照以下步骤进行手动安装：

1、获取源代码。所有的代码都在一个叫做 robot 的目录下。如果你有一个源分布或者版本控制校验，你可以在 src 目录下找到，但是你也可以从更早的安装中获取到。

2、复制源代码到你需要的地方。

3、创建需要的运行脚本，如果你有源代码包或者校验，可以从 src/bin 目录下获得模板。

单独发布的 Jar

从 Robot Framework 2.5.2 开始，Robot Framework 可以以独立的 robotframework-<version>.jar 的方式发布，这个包里面包含了 Jython 和 Robot Framework。如果你不想用 Python 执行测试，那么这是一种一次性准备好所有文件的好方法。下载 jar 文件之后，你可以这样执行：

```
java -jar robotframework-2.5.2.jar --help
java -jar robotframework-2.5.2.jar mytests.txt
java -jar robotframework-2.5.2.jar --variable name:value mytests.txt
```

如果你需要输出运行过程中的结果，rebot 必须作为第一个参数传递给 jar 文件：

```
java -jar robotframework-2.5.2.jar rebot --help
java -jar robotframework-2.5.2.jar rebot output.xml
java -jar robotframework-2.5.2.jar rebot --name Combined outputs/*.xml
```

文件应该安装在哪里

当使用自动安装的时候，Robot Framework 源代码被复制到包含外部申明的 Python 模块所在的目录下。手动安装位置根据平台各不相同，但是像 UNIX 一类的操作系统，通常是在/usr/lib/[PythonVer]/site-packages 这样的目录下，在 windows 一类操作系统上一般是在[PythonInstallationDir]\Lib\site-packages 这样的目录下。手动安装的 Robot Framework 源代码在名为 robot 的目录下，当采用简易安装的时候在 obotframework[RobotVer].py[PythonVer].egg/robot 目录下。

Robot Framework 运行脚本(**pybot, jybot and rebot**)被创建后同时辅以到一个平台限定的路径。在 UNIX 一类的操作系统中，通常被拷贝到/usr/bin 下，所以能立即从命令行执行。在 windows 操作系统中，不提供相类似的路径，Python 拷贝这些脚本到[PythonInstallationDir]\Scripts 下。

设置环境

安装完成以后，你可能需要使得 Robot Framework 的脚本很容易的从命令行启动。在 UNIX 一类的操作系统中，是自动完成的，但是在 windows 操作系统中，不是自动完成的。包含安装文件的目录必须设置在 **PATH** 环境变量中。

在 windows 操作系统中设置 PATH 环境变量：

Open Start > Settings > Control Panel > System > Advanced > Environment Variables.有用户环境变量和系统环境变量，两者的不同点在于**用户环境变量**只是当前用户能使用，**系统变量**则是所有的用户都可以使用。

编辑 **PATH**，选择“**编辑**”并且将;[PythonInstallationDir]\Scripts 添加在行尾。注意以（;）开始非常重要，它是用来分隔不同的条目的。添加新变量，选择“新建”输入名称和值，此时输入的时候不需要输入分号。

为新的修改开启一个命令提示符使其生效。

在 windows 上安装 Python 不会创建[PythonInstallationDir]\Scripts 目录，但是在安装 Robot Framework 过程中会自动创建。

安装验证

验证安装和环境是否设置完成，输入：

```
$ pybot --version
Robot Framework 2.5 (Python 2.6.5 on darwin)
```

验证基于 Jython 的 Robot Framework 是否运行正常，输入：

```
$ jybot --version
Robot Framework 2.5 (Jython 2.5.1 on java1.6.0_07)
```

在这两种情况下。具体的版本号可以与上面的不同。在 Jython 上，第一次运行的时候可能会得到一些来自 Jython 包管理器的信息。

1.3.4 卸载

如果 Robot Framework 是通过源文件安装版的形式安装的，可以通过命令行卸载：

```
python install.py uninstall
```

如果 Robot Framework 是通过二进制安装包安装的，可以通过操作系统的进程清除。例如，在 window 操作系统中只需要到控制面板中执行添加/删除程序。

如果卸失败或者你采用的是简易安装，Robot Framework 的卸载可以通过手动清除框架运行代码和运行脚本来卸载。

1.3.5 升级

Robot Framework 的升级或者降级取决于所用的版本：

- 如果从一个较小的 Robot Framework 版本升级到其他的（例如，从 2.5 到 2.5.1），用新版本覆盖旧版本是比较安全的。
- 如果从一个主要的 Robot Framework 版本升级到其他的（例如，从 2.1.3 to 2.5），此时强烈建议安装前卸载掉旧版本。

- 如果您正在进行降级操作，请遵循和升级相同的准则。

对于源安装包，首先需要获得一个新的程序包，然后运行以下命令，这些命令会自动的运行卸载：

```
python install.py reinstall
```

对于简易安装您可以直接运行：

```
easy_install robotframework==<new-version>
```

不管版本或者安装方法，你不需要重新安装之前已经安装好的环境或者从新设置 PATH 环境变量。

1.4 范例

Robot Framework 快速启动指导作为一个单独的 demo。也是安装源文件的一部分（在路径 doc/quickstart 下），快速启动指导可以从 project web pages 上单独下载。

另外，两个外部库 SeleniumLibrary 和 SwingLibrary 有易于执行的示例。前者有一个简单、独立的 HTTP 服务器和一个被测系统应用程序，后者有一个 Swing 应用。也为不同的系统提供了准确的测试用例文件和脚本示例。

2 创建测试文件

2.1 测试文件语法

本节涵盖 robot framework 的整体测试数据的语法。以下各节将说明如何实际创建测试用例，测试集等。

2.1.1 文件和目录

测试用例的组织层次结构如下：

- 在测试用例文件（test case file）中建立测试用例
- 一个测试文件自动的建成一个包含了这些测试用例的测试集（test suite）
- 一个包含多个测试用例文件（test case file）的目录构成更高级别的测试集，即这个测试集目录包含多

个子测试集，而子测试集即是由测试文件创建而来。

- 一个测试集目录可以包含其他测试集目录，这种层次结构可以满足层嵌套的需要
- 测试集目录可以有一个特殊的初始文件（initialization file）

此外，还有：

- 测试库（Test libraries）包含最低级别的关键字（keywords）
- 资源文件（Resource files）包含变量和更高层次的用户自定义关键字
- 变量文件（Variable files）可以提供比资源文件更灵活的方式创建变量

2.1.2 支持的文件格式

Robot Framework 的测试数据被定义为表格形式，无论是使用超文本标记语言（HTML），制表符分隔值（TSV），纯文字，或 reStructuredText 格式。Robot Framework 通过文件扩展名为其选择一个测试数据解释器。扩展名不区分大小写，以其公认的扩展名为准，.html, .htm and .xhtml 是 HTML 格式，.tsv 是 TSV 格式，.txt 是纯文本格式，.rst 或.rest 是 reStructuredText 格式。只为 HTML 和 TSV 2 种文件格式提供了模板，模板的提供使得用例编写更简单。

HTML 格式

在 HTML 文件，测试数据定义在分开的表中（见下面的例子）。Robot Framework 认识这些测试数据是基于表中第一个单元格中的文字。被认可的表之外的任何东西都将被忽略。

Using the HTML format

Setting	Value	Value	Value
Library	OperatingSystem		

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

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

Keyword	Action	Argument	Argument
My Keyword	[Arguments]	${path}	
	Directory Should Exist	${path}	

编辑测试数据

你可以使用任何你喜欢的编辑器编辑 HTML 格式的测试数据文件，但是有一个图形化的编辑器可以使你真实的看到表格，这个编辑器我们推荐使用。我们提供一个这样的图形化工具 RIDE 来编辑测试数据。

编码和实体引用

HTML 的实体引用（如：ä）是支持的。此外，可以使用任何编码，假设它是在数据文件中被指定。普通的 HTML 文件必须使用如下列所示的 META 元素：

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

XHTML 文件需要使用如下列所示的 XML preamble：

```
<?xml version="1.0" encoding="Big5"?>
```

假设没有指定编码，Robot Framework 将默认使用 ISO-8859-1。

TSV format

同 HTML 一样，TSV 格式也可以用于 Robot Framework 的测试数据。在 TSV 文件中，所有数据是在一个大的表格中。测试数据表从一个或多个*开始被识别的，紧跟着一个普通的表名和一个可选的关闭的*。同 HTML 数据一样，在第一个被识别的表格之外的将被忽略。

Using the TSV format

Setting	*Value*	*Value*	*Value*
Library	OperatingSystem		
Variable	*Value*	*Value*	*Value*
${MESSAGE}	Hello, world!		
Test Case	*Action*	*Argument*	*Argument*
My Test	[Documentation]	Example test	
	Log	${MESSAGE}	
	My Keyword	/tmp	
Another Test	Should Be Equal	${MESSAGE}	Hello, world!
Keyword	*Action*	*Argument*	*Argument*
My Keyword	[Arguments]	${path}	
	Directory Should Exist	${path}	

编辑测试数据

你可以在任何电子表格程序中创建和编辑 TSV 文件，如微软的 Excel。在保存文件的时候选择制表符分割格式并将文件的扩展名设置为.tsv。这也是个好主意，将工具的所有自动修正关闭使其像处理纯文本文件一样。TSV 文件对于任何文本编辑器是比较容易编辑的，尤其是该编辑器支持可视化的制表符分割。RIDE 同样支持 TSV 格式。

Robot Framework 解析 TSV 数据首先根据表格的特点是将所有内容分裂为多行，然后再分裂为单元格。电子表格程序有时会用引号将单元格引上（如：“**my value**”），Robot Framework 会去除这些引号。有时引号在数据中是成对的这种也能被正确处理（如：**"my ""quoted"" value"**）。假如你是使用电子表格程序生成 TSV 文件，你可以不用关心这些，但如果你编程创建数据，则必须遵循同电子表格一样的引号约定。

Encoding

TSV 文件要求使用 UTF-8 编码。由于 ASCII 是 UTF-8 的子集，所以也可以使用无格式 ASCII 编码。

Plain text format

纯文本格式在技术上同 TSV 格式相似，只是单元格之间的分隔符不同。TSV 格式使用 Tab 分割，但在纯文

本格式中可以使用两个或多个空格以及前后有空格的管道字符（ | ）。在所有用例的测试数据表格中必须同 TSV 格式类似其前面必须有 1 个或多个星号，并且在第一个表格之前的一切会被忽略。

小贴士

Robot Framework 2.1.1 及其后续版本才支持纯文本格式。

从 Robot Framework 2.1.2 开始，在纯文本中的 tab 自动被转换为两个空格。纯文本格式允许同 TSV 格式类似使用单个 tab 做为分隔符。但是请注意，在纯文本格式中多个 tab 被认为是一个分隔符，但在 TSV 格式中每个 tab 被认作一个分隔符。

空格分隔格式

做为分隔符的空格数量可以不同，要求最少 2 个空格，这样可以很好的使数据排列整齐。这同在文本编辑器中编辑 TSV 格式有个明显的好处，TSV 格式的对齐不能被控制。

```
***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}
```

因为空格可以作为分隔符，所以所有的空单元格必须使用${EMPTY}或者反斜杠。另外，对空格（包括空格 spaces、换行 newlines 和制表符 tabs）的处理与其他测试数据是相同的，即：对于首尾空格、连续的空格都需要进行转义。

Pipe and space separated format

使用空格分隔格式的最大问题是在视觉上分隔关键字形式的参数很困难。当关键字接多个参数或者参数包含空格时，这种问题尤其困难。在这种情况下使用管道符和空格分隔会工作的更好，因为这样可以使单元

格的边界更明显。

```
| *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}
```

在一个纯文本文件中可以包含空格分隔和空格与管道符组合分隔，但是在一行中必须只使用同一种分隔符。用空格和管道符分隔的行，其开头的管道符必须有，这是强制规定，但其结尾的管道符可以省略，是可选的。在管道符的两侧都必须有至少一个空格（开头和结尾的管道符除外），但是没有必要校正管道符的位置除非它能使数据更清晰。

在使用管道和空格分隔符格式时没有必要转义空白单元格（尾部空格单元格除外）。唯一需要注意的是如果在真实的测试数据中包含管道符必须使用反斜杠进行转义，如：

```
| ${file count} = | Execute Command | ls -1 *.txt \| wc -l |
| Should Be Equal | ${file count}   | 42                   |
```

Editing and encoding

使用纯文本格式比 TSV 格式和 HTML 格式更好的最大的一个优点是使用普通文本编辑器更容易编辑。Emacs 集成的 robot-mode.el 插件提供了语法高亮和关键字完成。纯文本格式同样支持 RIDE，但仅支持空格分隔变量。

同 TSV 格式类似，纯文本文件要求使用 UTF-8 编码，因此 ASCII 文件也能被支持。

reStructuredText format

reStructuredText 的（reST）是一个易于阅读的纯文本标记语法，在 python 项目中被普遍使用（包括 Python 自身以及该用户指南）。

通过 Robot Framework 使用 reST 可以允许用户混合多种格式文档和表格，这种特定测试数据在简明文本格

式中易于被简单文本编辑器所编辑，易于被比较工具和代码控制系统所使用。Robot Framework V2.1 版本开始支持 reStructuredText 格式。

处理 reStructuredText 的工具作为 docutils 项目的一部份可以被自由的使用，并且这里有快速参考指南展示最常用的格式结构其中包括 Robot Framework 使用的表格。

需要注意的是 Robot Framework 在真正开始解析测试数据之前需要将 reST 格式转换为 HTML。这些数据必须严格遵循 reST 的语法，否则将不能处理成功。

小贴士

如果 Robot Framework 使用 reST 需要先安装 Python docutils 模块。

在 reST 文件中，同 HTML 格式类似测试数据被定义在文件的表格中。Robot Framework 确是基于第一个单元格中的文字来识别测试数据表的，在认可的表格之外的所有内容都将被忽略。

以下例子是使用 reST 简单表格格式的四种测试数据表格。需要注意的是如果表格第一列是一个空的单元格，需要用“\”或者“..”来标识：

```
============  ================  =======  =======
  Setting          Value          Value    Value
============  ================  =======  =======
Library       OperatingSystem
============  ================  =======  =======


============  ================  =======  =======
  Variable         Value          Value    Value
============  ================  =======  =======
${MESSAGE}    Hello, world!
============  ================  =======  =======


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


============  ======================  ============  ==========
 Keyword            Action              Argument     Argument
============  ======================  ============  ==========
My Keyword    [Arguments]             ${path}
\             Directory Should Exist  ${path}
============  ======================  ============  ==========
```

Editing test data

在 reST 文件中的测试数据可以被任何文本编辑器所编辑。建议将编辑器使用等宽字体以帮助表元素对齐。Robot IDE 不支持在 reST 源文件中直接编辑测试数据。

Temporary files when using reST

不同于 HTML 和 TSV 格式，Robot Framework 不能直接使用 reST 文件。相反，docutils 将自动将 reST 源文件转换为临时 HTML 文件供后续 Robot 使用。这些临时 HTML 文件会在使用后及时被清理。这些 HTML 文件的产生和清理是由 Robot Framework 内部控制的，不需要用户直接调用 docutils 工具。

Syntax errors in reST source files

假如 reST 文件不是正确的语法格式（如包含畸形的表格），reST 到 HTML 的转换将不会发生并且没有一个测试用例可以从该文件中被读出来。当这种情况发生时，Robot Framework 将在控制台输出文件中显示 docutils 的错误信息，包括文件名、行号、资源上下文和错误类型。

2.1.3 测试数据表格

测试数据被组织在以下四种表格中。这些测试数据表格通过表格的第一个单元格被识别，下列表格的最后一列列出了可以做为表名称使用的不同别名。

Table name	Used for	Aliases
Setting table	1）导入测试库，资源文件和变量文件 2）为测试集和测试用例定义元数据	Setting Settings Metadata
Variable table	定义可以用在其他地方的测试数据的变量。	Variable Variables
Test case table	通过可用的关键字创建测试用例。	Test Case Test Cases
Keyword table	通过低级别关键字创建用户自定义关键字	Keyword Keywords User Keyword User Keywords

2.1.4 解析测试数据的规则

数据忽略

当 Robot Framework 在解析测试数据的时候，以下数据将会被忽略：

- 第一个单元格不是以可识别的表名开头的所有表格。
- 表格第一行中除了第一个单元格的所有东西。
- 在 HTML 或 reST 文件中表格外的数据和 TSV 文件中第一个表格之前的数据。
- 所有空行，这种方式意味着可以使表格更具有可读性。
- 每行结尾处的所有空单元格，你必须添加一个反斜杠（\）以防止这种类型的单元格被忽略。
- 所有单反斜杠（\），这被用来作为转义字符。
- 所有跟随在哈希字符（#）后的字符，假如这个哈希字符是单元格的第一个字符。这意味着可以用于测试数据的注释。
- 在 HTML 或 reST 测试数据中的所有格式。

当 robot framework 忽略这些数据后，这些数据也不会出现在测试结果报告中，此外大部分使用了 robot framework 的工具也会忽略这些数据。为了使一些信息对 robot framework 的输出或是 RIDE 可见，需要将这些信息写入说明书或测试用例/测试集的元数据中，或是使用 BuiltIn library 库中的 Log 或 Comment 关键字来记录日志。

转义

Robot framework 解析器的转义字符是反斜杠（\），这个转义字符可以在以下内容中使用：

- 转义特殊字符，从而使用它们的原义：
  - \${notvar}：意味着一个字面上的字符串${notvar}虽然看起来像个变量
  - \\：意味着一个单斜杠(例如, C:\\Temp)
  - \#：意味着哈希符(#), 常被用于一个单元格的开始
- 影响空字符的解析
- 防止忽视一行结尾空的单元格(这需要“\”在相应的单元格)，另一种处理方法是使用内建变量${EMPTY}。

空格处理

Robot framework 处理空格，例如：空格，换行或者 tab 键，其处理方式与 HTML 处理空格的方式是一致的，即：

- 去除所有单元格中开头和结尾的空格
- 将多个连续空格变为一个空格
- 将所有的换行和 tab 转换为空格

如果希望 robot framework 在解析数据的时候不遵照以上原则，就需要使用转义字符反斜杠：

- 开头的空格之前，例如：\ sometext
- 连续空格之间，例如：text \\ more text
- 结尾的空格之后，例如：text \\
- 使用\n 创建新一行，例如：first line\n2nd line

- 使用\t 创建 tab 键，例如：text\tmore text
- 使用\r 创建回车键，例如：text\tmore text

另外，通常使用内建变量${SPACE}代表开始、结尾或连续的空格。扩展变量语法可以使用${SPACE*8}这种格式，这样对于使用连续空格非常方便。

将测试数据分为多行

如果测试数据超过一行，可以使用省略号（...）连接前一行

在测试用例表和用户自定义表中，省略号前面必须有一个空单元格。该方法对于设置（setting）和变量（variable）也一样有效，只是省略号可以直接放在 setting 和 variable 名称下面。

在所有表格中，省略号前的所有空格将被忽略。

此外，在 setting 中的一个值（主要是注释）可以被分割到多列。这些测试数据在解析时将会把这些值通过空格连接在一起。

所有这些语法将在下面的例子中说明。在第一个三个测试数据表中，测试数据没有被分裂为多行，在随后的三个表中，测试数据被分裂为多行和多列。

测试数据未被分行：

Setting	Value	Value	Value	Value	Value	Value
Default Tags	tag-1	tag-2	tag-3	tag-4	tag-5	tag-6

Variable	Value	Value	Value	Value	Value	Value
@{LIST}	this	list	has	quite	many	items

Test Case	Action	Argument	Arg	Arg	Arg	Arg	Arg	Arg
Example	[Documentation]	Documentation for this test case. This can get quite long...						
	[Tags]	t-1	t-2	t-3	t-4	t-5		
	Do X	one	two	three	four	five	six	
	${var} =	Get X	1	2	3	4	5	6

测试数据被分为多行：

Setting	Value	Value	Value
Default Tags	tag-1	tag-2	tag-3
...	tag-4	tag-5	tag-6

Variable	Value	Value	Value
@{LIST}	this	list	has
...	quite	many	items

Test Case	Action	Argument	Argument	Argument
Example	[Documentation]	Documentation	for this	test case.
	...	This can get	quite	long...
	[Tags]	t-1	t-2	t-3
	...	t-4	t-5	
	Do X	one	two	three
	...	four	five	six
	${var} =	Get X	1	2
	...	3	4	5
	...	...	6	

小贴士

在省略号之前的空单元格只有在 robot framework2.5.2 及其以后的版本才被允许。更早的版本只有在循环中才允许以空单元格开头。

在 reST 表格中分裂数据

在纯文本文件的 reST 表格中，这里有两种表格语法用于创建测试数据。当使用简单表格语法，\和..被允许使用在第一个单元格，附加...用于连接前一行，例如：

```
===========  ================  ==============  ==========  ==========
 Test Case        Action           Argument      Argument    Argument
===========  ================  ==============  ==========  ==========
Example      [Documentation]   Documentation   for this    test case.
\            ...               This can get    quite       long...
\            [Tags]            t-1             t-2         t-3
\            ...               t-4             t-5
\            Do X              one             two         three
\            ...               four            five        six
\            ${var} =          Get X           1           2
\            ...               3               4           5
\            ...               6
===========  ================  ==============  ==========  ==========
```

网格表格语法，建议改为：空格被允许使用在第一个单元格，附加...用于连接前一行：

Test Case	Action	Argument	Argument	Argument
Example	[Documentation]	Documentation	for this	test case.
	...	This can get	quite	long...
	[Tags]	t-1	t-2	t-3
	...	t-4	t-5	
	Do X	one	two	three
	...	four	five	six
	${var} =	Get X	1	2
	...	3	4	5
	...	6		

2.2 创建测试用例

这一章节将描述创建测试用例的语法。在下一章节将介绍如何使用测试用例文件将测试用例组织进测试集以及测试集文件夹的组织。

2.2.1 测试用例语法

基本语法

测试用例是使用可用的关键字在测试用例表格创建而成。关键字可以通过测试库和资源文件引入，也可以在测试用例文件中的关键字表中创建。

测试用例表格的第一列是测试用例名称。一个测试用例的开始是从包含测试用例名词的这一行到下一个用例名称或表格的结束。在表格表头到第一个测试用例中存在东西是错误的。

第二列通常是关键字名称。这种规则的例外是设置关键字的返回值，在第二个或后续的列可能会包含变量名并且关键字跟随其后。在这两种用例中，列的内容可能是参数或是特定的关键字。

测试用例例子：

Test Case	Action	Argument	Argument
Valid Login	Open Login Page		
	Input Name	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

测试用例表格的设置

测试用例可以有他自己的设置。与关键字的位置一样,设置名称也是在表格的第二列，设置的值在后续的列中。设置名称由方括号包围，以区别于关键字。以下是可用的设置名称，并将在本章的后面进行解释。

[Documentation]

用于说明测试用例的注释

[Tags]

用于标记测试用例

[Setup], [Teardown]

用于指定测试用例的 Setup 和 Teardown。其同义词是[Precondition]和[Postcondition]

[Template]

用于指定测试用例模板。这个测试本身将只包含数据，这被用于关键字的参数。

[Timeout]

设置测试用例超时。超时会在相应的章节阐述

测试用例的设置例子

Test Case	Action	Argument	Argument
Test With Settings	[Documentation]	Another dummy test	
	[Tags]	dummy	owner-johndoe
	Log	Hello, world!	

设置表格（Setting Table）中的与测试用例相关的设置

以下是设置表中测试用例的相关设置。在测试用例指定设置之前，这些值是设置的默认值。

Force Tags, Default Tags

Tag 的强制和默认值

Test Setup, Test Teardown

Test Setup，Test Teardown 的默认值。其同义词是 Test Precondition 和 Test Postcondition。

Test Template

默认模板关键字

Test Timeout

测试用例超时的默认值。

2.2.2 变量使用

前面章节已经展示了使用不同参数的关键字，在这一个章节将更进一步阐述这个重要的功能。如何真正实现拥有不同参数的用户关键字和库关键字将分几个部分介绍。关键字可以接受零或多个参数，并且有些参数可能会有默认值。关键字能接受那种参数取决于其实现，通常可以搜索关键字注释获取这些信息。在本节中的例子里，注释是用 libdoc.py 工具生成的，但同样的信息在诸如 **javadoc** 之类的生成工具生成的注释中也能获取到。

必选参数

大多数的关键字通常有指定数目的参数，这些是必须的参数。在关键字文档中，通过逗号分隔的特定参数名词来制定其参数，如：first，second，third。这些参数的名称不重要，除非需要解释这个参数的作用，但重要的是参数的数目必须同文档中规定的一致。使用过多或过少的参数都将是错误的。下面的例子用到了 *OperatingSystem* 库中的关键字 *Create Directory* 和 *Copy File*。第一个关键字有 1 个参数，用来指定路径；第二个关键字有 2 个参数，分别用来指定源文件和目标文件夹；最后一个关键字是来自 *BuiltIn 库*的 *No Operation*，它不需要参数。

Test Case	Action	Argument	Argument
Example	Create Directory	${TEMPDIR}/stuff	
	Copy File	${CURDIR}/file.txt	${TEMPDIR}/stuff
	No Operation		

默认值

有默认值的参数通常可以提供或不提供。在本文档中，参数名和默认值之间由等号分隔，如 name = default value，但是对使用 Java 实现的关键字来说，同一个关键字可能会有多种实现方法，只是它们的参数不同。所有的参数都可以有默认值，但是有默认值的参数后面不能再有参数了。

下面的例子说明了默认值的使用，这个例子使用了 *Create File* 关键字，参数包括：**path, content=, encoding=UTF-8**。如果试图不使用任何参数或者是超过三个参数，这个关键字都不能工作。

Test Case	Action	Argument	Argument
Example	Create Directory	${TEMPDIR}/stuff	
	Copy File	${CURDIR}/file.txt	${TEMPDIR}/stuff
	No Operation		

可变数目的参数

可以创建接受任意数目参数的关键字。这些参数可以由强制参数和有默认值的参数组成，但可变参数必须在最后。在本文档中，这种参数通常是在参数名前面有个星号（*），如：*varargs。

Remove Files 和 *Join Paths* 关键字在下面例子中分别使用这些参数：*paths 和 base、*parts。前者可以使用任意数目的参数，后者则至少需要一个参数。

使用可变数目参数的关键字

Test Case	Action	Argument	Argument	Argument
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

当关键字接受多于一个具有默认值的参数时，如果只对最后一个参数进行重新赋值是不可能的。例如：有个关键字接受参数 rg1=a, arg2=b, arg3=c，其中参数 rg1=a, arg2=b 将得到空字符串而不是他们的默认值。

Overriding default values with positional arguments

Test Case	Action	Argument	Argument	Argument
Positional Arguments	[Documentation]	1st and 2nd	argument get	empty strings
	Example Keyword			value

为了让只给某些参数覆盖默认值更容易，在 Robot Framework2.5 中增加了命名参数的新语法。有了这种参数语法，对于需要覆盖默认值的参数只要按照如下格式即可 argname=value。这样使用默认值的参数就可以被排除在外。下面的例子说明了这种语法是如何实现的，这个例子使用了与上一个例子相同的关键字。在这个例子中，没有写明的参数将使用他们的默认值。

Test Case	Action	Argument	Argument	Argument
Named Arguments	[Documentation]	Not specified	arguments get	default values
	Example Keyword	arg3=value		
	Example Keyword	arg2=xxx	arg3=yyy	

这个命名参数的语法能够很自然的用于接收默认值，当参数没有被排除。这种语法相比较只显示参数值来说使得参数含义更清晰。

这种语法最大的限制就是仅能用于用户自定义关键字和使用 python 的静态库（static library API）和混合库（hybrid library API）实现的关键字。在未来将支持 java 库和动态库实现的关键字。

小提示

当用户自定义关键字使用命名参数语法时，这个命名参数没有使用${}符号。例如，用户自定义关键字参数${arg1}=default，${arg2}=second 必须使用如下格式 arg2=override。

这种命名参数语法仅用于当部分前面的参数需要默认值。这种语法是从给定的参数列表的末尾开始匹配，直到没有可以匹配的。在一些少有的例子中，可以使用“\”用于排除不需要的匹配，如 nomatch\=here

下面例子展示命名参数语法如何在不同场景下使用命名参数，包括如何在引入测试库文件时使用命名参数：

Setting	Value	Value	Value
Library	Telnet	prompt=$	

Test Case	Action	Argument	Argument	Argument
Example	Open connection	10.0.0.42	port=${25}	
	List files	options=-lh		
	List files	path=/tmp	options=-l	

Keyword	Action	Argument	Argument	Argument
List files	[Arguments]	${path}=.	${options}=	
	Execute command	ls ${options} ${path}		

关键字名称嵌入参数

这种完全不同的方法使用关键字名称去说明参数。这种语法在当前只支持用户自定义关键字。

2.2.3 测试用例的名称和文档注释

测试用例名称直接来自测试用例表格：它是开始于测试用例那一列。在同一个测试集中只测试用例名称是唯一的。关于这一点，可以使用自动变量${TEST_NAME}在测试用例自身中引用测试用例名称。这个在测试执行的任何时候都是可用的，包括所有用户自定义关键字，setup 和 teardown。

[Documentation]设置可以用于设置测试用例的注释。这些文字将在命令行输出中显示，也会显示在测试日志和测试报告中。假如注释较长，这些注释会被分割到多个通过空格连接起来的单元格中。可以用简单的 HTML 格式、或是变量来创建动态注释。从 Robot Framework 2.1 版本开始，注释中用到的并不存在的变量将原样保持。

Test Case	Action	Argument	Argument
Simple	[Documentation]	Simple documentation	
	No Operation		
Splitting	[Documentation]	This documentation is a bit longer and	it has been split into several columns.
	No Operation		
Formatting	[Documentation]	*This is bold*, _this italic_ and	here is a link: http://robotframework.org
	No Operation		
Variables	[Documentation]	Executed at ${HOST} by ${USER}	
	No Operation		

重要的是如果测试用例有清晰和具有描述性的名称，则没有必要对用例添加注释。如果测试用例的逻辑需要注释，这表明测试用例中的关键字需要更明确的名称而不是添加额外的注释。最后，元数据例如最后一个例子中的环境变量和用户信息最好指明使用的标签。

2.2.4 给测试用例打标签

在 Robot Framework 中使用 tag 是一个简单但强大的对测试用例进行分类的机制。Tag 可以是任意文字，可以用于以下用途：

- Tag 可以在测试报告和日志中显示，当然，在测试数据中可以向测试用例提供原数据。
- 测试用例的统计（可以自动通过 tag 收集总数，通过的，失败的测试用例）。
- 使用 tag，可以包含和排除测试用例的执行。
- 使用 tag，可以指定哪些测试用例是严重级别的。

在本章节将要解释如何对测试用例设置 tag，后续介绍不同的方法。这些方法可以一起使用。

Force Tags in the Setting table

在设置表格中设置 Force Tags

使用该设置的测试用例文件中的所有测试用例都将使用这指定的 tag。如果是在测试集文件中设置的该 tag，那么该测试集下的所有测试用例都将得到该 tag。

Default Tags in the Setting table

在设置表格中添加 Default Tags

测试用例如果没有[tags]设置，将获取该 Default Tags。但在 Robot Framework2.5 之后，测试集初始化文件中不再有该设置了。

[Tags] in the Test Case table

测试用例最常用到的 tag。此外有该 tag，就不能获得指定的 Default Tags

--settag command line option

所有执行的测试用例都将获得这命令行设置的 tag 和在其他地方设置的 tag。

Set Tags and *Remove Tags* keywords

这些内建的关键字可以在测试用例执行时更改其 tag。

Tag 可以是任意文字，但他们都会被转换为小写字母，并且空格会被删除掉。假如一个测试用例获取了多次相同的 tag，除了第一个其他的 tag 都将被删除。Tag 可以用变量创建，假设这些变量是存在的。

Setting	Value	Value	Value
Force Tags	req-42		
Default Tags	owner-john	smoke	

Variable	Value	Value	Value
${HOST}	10.0.1.42		

Test Case	Action	Argument	Argument
No own tags	[Documentation]	This test has tags	owner-john, smoke, req-42
	No Operation		
With own tags	[Documentation]	This test has tags	not_ready, owner-mrx, req-42
	[Tags]	owner-mrx	not_ready
	No Operation		
Own tags with variables	[Documentation]	This test has tags	host-10.0.1.42, req-42
	[Tags]	host-${HOST}	
	No Operation		
Empty own tags	[Documentation]	This test has tags	req-42
	[Tags]		
	No Operation		
Set Tags and Remove Tags Keywords	[Documentation]	This test has tags	mytag, owner-john
	Set Tags	mytag	
	Remove Tags	smoke	req-*

2.2.5 测试用例的 setup 和 teardown

Robot Framework 同其他测试框架相同有类似的 setup 和 teardown 方法。简言之，setup 是在测试用例执行之前执行的，teardown 是在测试用例执行之后执行的。Robot Framework 的 setup 和 teardown 可以使用带有参数的普通关键字。Setup 和 teardown 通常是单个关键字。如果你需要使用多个分开的任务，则需要创建更高级别的用户自定义关键字。另一种解决方案是使用 Robot Framework2.5 后添加的内建关键字 run keywords。

测试用例中的 teardown 通常有两种用途。首先，它在测试用例失败后也能被执行，所以能用来执行必须清理的动作，无论测试用例的状态如何。从 Robot Framework2.5 开始，teardown 中的所有关键字都将被执行，即使其中有一个失败的。对于普通关键字也可以在失败后继续执行，但在 teardown 中这功能是默认的。

在测试用例中使用 setup 和 teardown 的最简单方法是在设置表中设置[Test setup]和[Test teardown]。单独的用例也可以有 setup 和 teardown。是在用例中设置[setup]和[teardown]，并且会覆盖 Test setup 和 Test teardown 设置。在[setup]和[teardown]之后没有关键字，则意味着该用例没有 setup 和 teardown。

Setting	Value	Value	Value
Test Setup	Open Application	App A	
Test Teardown	Close Application		

Test Case	Action	Argument	Argument
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]		
Using variables	[Documentation]	Setup and teardown	given as variables
	[Setup]	${SETUP}	
	Do Something		
	[Teardown]	${TEARDOWN}	

在测试用例中，*precondition* 和 *postcondition* 放在前面更好是 setup 和 teardown。Robot Framework 支持这种术语，*precondition* 就是 setup，*postcondition* 就是 teardown。

Test Setup	Test Precondition
Test Teardown	Test Postcondition
[Setup]	[Precondition]
[Teardown]	[Postcondition]

作为 Setup 和 teardown 执行的关键字名称可以是一个变量。这个优点在于：可以在命令行下给定关键字名字变量，从而使得在不同的环境下使用不同的 setup 和 teardown。

小提示

测试用例有他们自己的 setup 和 teardown。一个测试用例的 setup 是在该测试集的所有测试用例和子测试集之前执行，类似 teardown 是在所有测试用例执行完成后再执行。

2.2.6 测试模板

测试模板可以让关键字驱动测试用例转换为数据驱动测试用例。鉴于普通测试用例是由关键字和可能的参数组成，使用了模板的测试用例只需要定义模板关键字的参数即可。下列 2 个测试用例实例功能是完全一致的。

Test Case	Action	Argument	Argument
Normal test case	Example keyword	first argument	second argument
Templated test case	[Template]	Example keyword	
	first argument	second argument	

如上例所示，可以去掉为单个测试用例指定模板，通过*[Template]*设置为单个测试用例指定模板。另一种方法是在设置表中设置测试模板，在这种情况下，模板应用于该测试用例文件中的所有测试用例。这个[Template]设置会覆盖去掉设置表中的模板，如果[Template]的值为空则意味着这个测试不使用模板即设置了测试模板。

假如一个模板测试用例同下面例子一样有多行测试数据，那这个模板将所有行的数据有效。这意味着相同的关键字将被执行多次，每行数据执行一次。模板测试是一种特殊的形式，所有的轮次都将被执行，即使出现执行失败。普通测试用例使用 continue on failure 模式也能达到同样的效果，但是模板测试自动就按照该模式执行。

Test Case	Action	Argument	Argument
Templated test case	[Template]	Example keyword	
	first round 1	first round 2	
	second round 1	second round 2	
	third round 1	third round 2	

假如模板使用循环，则模板将执行循环中的所有步骤。continue on failure 模式在这种模式下也可以使用，这意味着循环中的步骤都将执行即使出现执行失败。

Test Case	Action	Argument	Argument	Argument
Template and for	[Template]	Example keyword		
	:FOR	${item}	IN	@{ITEMS}
		${item}	2nd arg	
	:FOR	${index}	IN RANGE	42
		1st arg	${index}	

测试模板主要的使用目的是减少数据驱动测试的冗余。在设置表中使用一次，就可以代替文件中测试用例的所有相同关键字。在下一章节将有更详细的说明。

小提示

测试模板这种新特性在 Robot Framework2.5 中支持

小提示

当前可能不支持模板关键字使用变量，这一限制将在后续版本中解决。

2.2.7 不同的测试用例模式

这里有几种不同编写用例的方法。测试用例是工作流（workflow）的描述，可以使用关键字驱动模式（keyword-driven）也可以使用行为驱动模式（behavior-driven）。数据驱动模式可以在用变化的数据测试相同的工作流时使用。

关键字驱动模式

工作流测试，如前面所示的有效登录测试（*Valid Login*），它由一些关键字和参数所组成。他们的正常结构是首先让系统进入初始状态（如：有效登陆中的打开登陆页面），然后是对系统进行操作（输入名字，输入密码，提交认证），最后验证系统行为表现是否同预期一致（如：欢迎页面是否打开）。

数据驱动模式

另一种测试用例编写的风格是数据驱动，即测试用例只使用一个更高级别的关键字。通常这个关键字是用户自定义关键字以隐藏实际的测试工作流。当用于测试相同的脚本有不同的输入或输出数据，这些测试是非常有用的。这将使得每次测试都重复相同的关键字，但测试模板的功能只允许指定的关键字执行一次。

数据驱动测试例子：

Setting	Value	Value	Value
Test Template	Login with invalid credentials should fail		

Test Case	User Name	Password	
Invalid User Name	invalid	${VALID PASSWORD}	
Invalid Password	${VALID USER}	invalid	
Invalid User Name And Password	invalid	whatever	
Empty User Name	${EMPTY}	${VALID PASSWORD}	
Empty Password	${VALID USER}	${EMPTY}	
Empty User Name And Password	${EMPTY}	${EMPTY}	

上面的例子有六个独立的测试用例，每一个是无效的用户名和密码组合。在下面的例子中，将展示如何使用一个用例测试所有的组合。当使用测试模板，测试中的所有轮都将被执行，即使中间出现失败，因此这两种风格是没有真正的功能差异。上面的例子按照不同的组合进行命名，这样能更清晰的看出在测试什么，

但在后续统计用例数量时可能存在巨大的潜在混乱。使用哪一种方法，取决于测试的上下文和个人喜爱。

数据驱动测试使用变化的数据

Test Case	User Name	Password	
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}	

小提示

在上述的两个例子中，列标题已经更改为匹配的数据，这是因为在第一行中除了第一个单元格以外的都将被忽略。

行为驱动模式

这种编写用例的需求是为了让非技术的项目利益相关者也能读懂用例。这种需求是验收测试驱动开发（ATDD）的基石。

编写这种测试的一种方法是使用 *Given-When-Then* 模式的行为驱动开发（BDD）。当使用该模式编写测试用例，初始化状态通常表现为由 Given 开头的关键字，后续动作的执行表现为由 when 开头的关键字，预期结果由 then 开头的关键字表示。由 and 开头的关键字通常表示有超过一个以上的行为。

行为驱动测试用例

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

忽略 Given/When/Then/And 前缀

如果全名匹配没有找到时，前缀 Given、When、and、Then 将在搜索关键字时被忽略。这对所有用户自定义关键字和库关键字都是适用的。例如：上例中的 *Given login page is open* 能做为带有或不带有 Given 的用户关键字被执行。忽略前缀还支持相同的关键字使用不同的前缀。例如 *Welcome page should be open* 也如作为 And *Welcome page should be open* 使用。

嵌入数据到关键字中

当编写具体的例子，它是一种非常有用的传递实际数据到关键字实现方法。用户关键字支持通过嵌入参数到关键字名称中，见 2.6.4。

2.3 创建测试集

Robot Framework 测试用例被创建在测试用例文件中，文件组织在目录结构中。这些目录和文件构成了测试集框架。

2.3.1 测试用例文件

Robot Framework 使用测试用例表格将测试用例创建在用例文件中。这个文件自动创建一个测试集将所有的测试用例包含其中。这里的测试集没有测试用例个数的限制，但建议不要超过 10 个测试用例，除非是使用数据驱动模式，这种模式的测试用例仅包含一个高级别关键字。

以下设置表格可以用于用户自定义测试集：

Documentation

用于解释测试集

Metadata

使用键值对设置测试集的元数据

Suite Setup, Suite Teardown

指定测试集的 setup 和 teardown。也可以使用其同义词 *Suite Precondition* 和 *Suite Postcondition*

2.3.2 测试集目录

测试用例文件可以组织在目录中，这些目录就构成了更高级别的测试用例。一个目录结构创建的测试集不能直接包含任何测试用例，但是可以包含其他有测试用例的测试集。这些目录可以被放置到另一些目录中，那些目录就构成了更高级别的测试集。这种结构是没有限制的，因此测试用例的组织是很有必要的。

当一个目录被执行时，以下所示的该目录下的文件和目录都将被递归执行：

- 名称以“.”和“_”开头的目录和文件将被忽略
- 目录名称是 CVS 的将被忽略（大小写敏感）

- 文件没有包含任何一个被认可的后缀将被忽略(*.html*, *.xhtml*, *.htm*, *.tsv*, *.txt*, *.rst*, or *.rest*)。大小写不敏感
- 除上所说的其他文件和目录都将被执行

假如一个文件或目录没有包含一个用例，将被静默忽略（仍会在日志中记录），并且其他的用例将被继续执行。

初始化文件

目录结构的测试集可以同文件结构的测试集有类似的设置。由于一个单独结构的目录不能包含这类信息，因此这些信息必须被存放在一个特殊的初始化文件中。

初始化文件同测试用例文件一样具有相同的结构和语法，但是不能有测试用例表并且有些设置是不支持的。

初始化文件的文件名必须是*__init__.ext*，其中扩展名必须是它支持的文件格式（如：*__init__.html* or *__init__.txt*）。这种名字格式是来自于 Python，这种命名格式的文件表示该目录是一个模块。

初始化文件的主要用途是指定测试集的相关设置同测试文件中的设置类似，但是在测试文件中的相关设置也是可用的。

在初始文件中引入或创建的变量和关键字在低级别的测试集中是不可用的，如果引入的资源文件需要在低级别的测试集中共享的话，资源文件是可以使用的。

如何在初始文件中使用不同的设置：

Documentation, *Metadata*, *Suite Setup*, *Suite Teardown*

这些测试集设置的使用同测试文件中的设置相同

Force Tags

无条件的指定该测试集下的所有测试用例的 tag

Test Setup, *Test Teardown*

为该测试集下的所有测试用例设置默认的 setup 和 teardown。可以在低级别的用例中被覆盖。

Default Tags, *Test Template*, *Test Timeout*

Robot Framework 2.5 后的版本在测试集初始文件中不再支持这些设置

测试集初始文件示例

Setting	Value	Value
Documentation	Example suite	
Suite Setup	Do Something	${MESSAGE}
Force Tags	example	
Library	SomeLibrary	

Variable	Value	Value
${MESSAGE}	Hello, world!	

Keyword	Action	Argument	Argument
Do Something	[Arguments]	${arg}	
	Some Keyword	${arg}	
	Another Keyword		

2.3.3 测试集的名称和文档注释

测试集名称由目录名或文件名构成。这样产生的测试集名称将会把扩展名忽略，并且会使用空格代替下划线，完全由小写字母组成的名称将按照标题样式进行转换。例如：*some_tests.html* 变为 *Some Tests* ，*My_test_directory* 变为 *My test directory*。

> 小提示
>
> 在 Robot Framework2.5 后，创建测试集名称的规则有微小改变。

文件和目录名称可以包含一个前缀用于控制这个测试集执行的顺序。前缀通过两个下划线与基本名称分隔，当创建真实的测试集名称时，前缀和下划线都将被删除。例如：文件 *01__some_tests.txt* 和 *02__more_tests.txt* 将被创建为测试集名称 *Some Tests* 和 *More Tests*。并且前者将先于后者被执行。

测试集的注释是在测试集的设置表格中使用 *Documentation* 标示。它可以用于在测试文件中，或者是更高级的测试集的初始文件中。测试集的注释与测试用例文件的注释相同，关于在哪显示和如何创建都有相同的规格参数表。

测试集注释如下：

Setting	Value	Value	Value
Documentation	An example test suite	documentation with	*some* _formatting_.
...	See test documentation	for more documentation	examples.

顶级测试用例的名称和注释，在测试集执行时可以被复写。这需要在命令行中使用*--name* 和*--doc* 命令，这

将在设置元数据章节解释。

2.3.4 自由的测试集元数据

测试集可以有除注释之外的其他元数据。这些元数据的设置可以使用测试集表格中的 *Metadata 设置*。这种方式的元数据将在测试报告和日志中显示出来。

元数据的名称和值跟随在 *Metadata* 后的列中。这些值的处理同注释相似，这意味着可以被分裂到多个单元格中（通过空格连接到一起），简单的 HTML 格式可以使用变量。

元数据示例：

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

对于最高级别的测试集，可以使用命令行*--metadata* 进行设置。详细信息将在设置元数据章节解释。

在 Robot Framework2.5 之前的设置元数据使用格式如：*Meta: <name>*，*<name>*是元数据的名称，其值在其后的列中。Robot Framework2.5 支持但不赞同使用老的格式。

2.3.5 测试集的 setup 和 teardown

并不是只有测试用例有 setup 和 teardown，测试集同样也有。一个测试集的 setup 将在所有测试用例和子测试集之前被执行，并且 teardown 将在他们所有之后被执行。所有的测试集都可以有 setup 和 teardown，通过目录创建的测试集的 setup 和 teardown 放在测试集初始化文件中。

类似于测试用例的 setup 和 teardown，测试集的 setup 和 teardown 都可以使用参数。他们被定义在设置表中，使用 *Suite Setup* 和 *Suite Teardown* 表示。他们有相似的同义词 *Suite Precondition* 和 *Suite Postcondition*,。关键字名称和可能的参数在设置后面的列。

假如测试集的 setup 执行失败，所有的测试用例和子测试集都将被标识为失败并不会被执行。这使得 setup 用于检测用例执行之前的必要条件成为可能。

Teardown 用于所有测试用例执行完之后的清理。即使 setup 执行失败，它也将被执行。假如测试集的 teardown 执行失败，所有用例都将被标识为失败，不管他们实际执行的结果如何。从 Robot Framework2.5 之后，setup 和 teardown 中的所有关键字都将被执行，即使出现失败。Setup 和 teardown 中执行的关键字名称可以是变量。这有利于在不同的环境下执行不同的 setup 和 teardown，通过命令行赋值。

2.4 使用测试库

测试库包含最低级别的关键字，称为库关键字（*library keywords*），这些关键字才实际同被测系统交互。所有的测试用例常常是通过高级别的用户自定义关键字来使用库关键字。该章节将展示如何使用库关键字以及提供了哪些库关键字。创建测试库将在独立的章节中介绍。

2.4.1 引入测试库

以下章节将介绍如何引入测试库。

使用库的设置

测试库的引入通常是在设置表中使用 *Library* 标识，并在其后的列中加入库名称。库名称是大小写敏感的（该名称是库的模块或类的实现名，并且必须严格匹配），但是其中的任何空格都将被忽略。使用模块中的 python 库或包中的 Java 库，名称必须使用包括模块名或包名的全名。

在一些情况下，测试库可能会需要参数，这些参数可以紧跟在库名后面的列中。测试库引入时使用参数同关键字的参数类似可以使用默认值，可变数量参数，命名参数。库名和参数都可以使用变量。

在设置表中引入参数

Setting	Value	Value	Value
Library	OperatingSystem		
Library	com.company.TestLib		
Library	MyLibrary	arg1	arg2
Library	${LIBRARY}		

在测试用例文件、资源文件、测试集初始文件中都可以引入测试库。在这些情况下，被引入库中的所有关键字在这些文件中都是可用的。在资源文件中，所有使用了这些资源文件的文件也可以引入的关键字。

使用 Import Library 关键字

另一种引入测试库的方式是使用内建库的关键字 *Import Library*。与库设置类似，这个关键字在引入测试库的时候可以携带参数，被引入库的关键字在使用了 *Import Library* 关键字的测试集中是可用的。

这种引入方式在以下情况非常有用：在测试开始执行的时候该测试库是获取不到的，只有通过其它关键字的调用执行才能获取到该测试库

使用 *Import Library* 关键字

Test Case	Action	Argument	Argument	Argument
Example	Do Something			
	Import Library	MyLibrary	arg1	arg2
	KW From Mylibrary			

测试库搜索路径

大多数正确引入测试库的方法是使用测试库的名称，如在本章节展示的所有示例。在这种情况下 Robot Framework 会试图在库搜索路径（library search path）下去查找实现测试库的模块和类。从根本上说，这意味着库代码和所有可能的依赖关系必须加入到 PYTHONPATH，如果是在 jython 上运行，则需要加入到 CLASSPATH 中。设置库的搜索路径，将在对应章节进行讲解。库可以自动设置搜索路径，也可以特别指定其如何工作。所有的标准库都库代码自动设置搜索路径。

这种方法的最大好处是当库的搜索路径已经配置，经常使用用户的 start-up 脚本，普通用户不用关心这些库是如何安装的。很简单其缺点是，库的搜索路径可能会需要一些额外配置。

使用测试库的物理路径

另一种指明库的机制是引入库时在文件系统中指定库的路径。这个路径是相对于当前测试数据文件的相对目录类似于资源文件和变量文件。这种方法最主要的好处是不需要配置库的搜索路径。

假如这个库是一个文件，则路径必须包含文件的后缀名。对于 Python 库扩展名自然是.py，对于 Java 库可以是.class 和.java，但这类文件必须是可用的。如果 Python 库的实现是一个目录，则其路径必须以反斜杠结尾（/）。下面将展示不同的用法：

Setting	Value	Value	Value
Library	PythonLib.py		
Library	/absolute/path/JavaLib.java		
Library	relative/path/PythonDirLib/	possible	arguments
Library	${RESOURCES}/Example.class		

这种方法对于用 Python 类实现的库的限制是这个类的名称必须同模块的名称一样。另外，这种方法不支持导入 JAR 或 ZIP 格式的库。

2.4.2 给测试库定制名称

在测试日志中测试库的名称显示在关键字的前面，假如有多个关键字有相同的名称，则必须在关键字名称前面加上测试库名称。测试库的名称通常是实现他们的类名或模块名称，但是也有一些情况需要改变库名

称，，例如：

有时需要使用不同的参数多次引入同一个测试库。没有这种功能是不可能实现的。

测试库的名称过长。例如，有个 Java 库的包名过长。

想使用变量在不同的时候引入不同的库，但是它们具有相同的名字。

这些库的名字可能会误导或很差。这种情况下改变其名字将是更好的解决办法。

这种基本的语法是在库名称后面使用 WITH NAME 来指定其新名字。这个指定的名字将在测试日志中显示，并且当在测试数据中使用关键字全名（LibraryName.Keyword Name）时必须使用这个新的库名。

使用用户自定义库名称导入库文件

Setting	Value	Value	Value
Library	com.company.TestLib	WITH NAME	TestLib
Library	${LIBRARY}	WITH NAME	MyName

库文件的参数放置在库的原名称和 WITH NAME 之间的单元格中。下面的例子展示了如何使用不同的参数多次引入同一个测试库。

相同的库文件使用不同的参数并对其进行不同的命名：

Setting	Value	Value	Value	Value	Value
Library	SomeLibrary	localhost	1234	WITH NAME	LocalLib
Library	SomeLibrary	server.domain	8080	WITH NAME	RemoteLib

Test Case	Action	Argument	Argument
My Test	LocalLib.Some Keyword	some arg	second arg
	RemoteLib.Some Keyword	another arg	whatever
	LocalLib.Another Keyword		

可以在设置表格中和 Import Library 关键字引入测试库时使用对测试库设置新的名字。

2.4.3 标准库

有一些测试库是随 Robot Framework 一起发布的，这些库被叫做标准库。以下是一些可用的标准库：

- BuiltIn
- OperatingSystem

- Telnet
- Collections
- String
- Dialogs
- Screenshot
- Remote

其中内建库 BuiltIn 是特殊的，因为他是考虑到自动使用的，所以它的关键字是始终可用的。其他的标准库则同其他库一样需要引入，但不需要安装。另外，他们能工作在 Python 和 jython 模式下（其中 Screenshot 库除外）。

新的标准库会在未来加入。假如你对新的标准库有想法或已经有了一个可以立即合入的新库，请及时联系 Robot Framework 的开发组。通常，判断一个库是否可以加入标准库的标准是：这个库是通用的，可以在没有其他额外依赖的情况下工作在 Python 和 jython 上，并且通过充分测试有充分的注释。

BuiltIn library

内建库提供了一系列常用的关键字集合。它是自动引入并且关键字始终是可用的。它提供的关键字可以用在诸如处理验证（Should Be Equal, Should Contain），处理转换（Convert To Integer）和其它各种其它目的中（Log, Sleep, Run Keyword If, Set Global Variable）。

内建库中的关键字在 Robot Framework v1.8 后进行过重命名。所有老的关键字都可以使用，但是关键字的长名称（在日志文件中展示）不赞成使用，它显示为以 DeprecatedBuiltIn 开头（例如：DeprecatedBuiltIn.Equals）。强烈建议使用新的关键字名称，旧版本中的关键字将在未来被完全去除。

更多信息请参考 BuiltIn library documentation.

OperatingSystem library

该操作系统库中的各种操作系统相关的任务将在 Robot Framework 运行时被执行。它可以执行命令（如：Run）、创建和移除文件及文件夹（Create File, Remove Directory）、检查文件或文件夹是否存在或是否包含其他东西（File Should Exist, Directory Should Be Empty），以及处理环境变量（Set Environment Variable）。

操作系统库中关键字名称在 Robot Framework V1.8 后进行了重命名，处理方式同内建库关键字处理方式相似。

更多信息请参考 OperatingSystem library

Telnet library

Telnet 库可以用于连接 Telnet 服务器并执行一些操作命令。

更多信息请参考 Telnet library

Collections library

集合库提供了一系列关键字用于处理 Python 的链表和字典。，如：从链表和字典中获取和修改值（Append To List, Get From Dictionary）以及验证其内容（Lists Should Be Equal, Dictionary Should Contain Value）。

更多信息请参考 Collections library

String library

字符串库可以处理字符串（Replace String With Regexp, Split To Lines）和验证其中的内容（Should Be String）。

更多信息请参考 String library。该库是在 Robot Framework 2.1 中新增的。

Dialogs library

对话框库提供了一种暂停测试执行并从用户处获取输入的方式。运行在 Python 和 jython 上的对话框会略有不同，但他们的功能是相同的。

更多信息请参考 Dialogs library。该库是在 Robot Framework 2.1 中新增的。

Screenshot library

Screenshot library 提供关键字用于对整个桌面进行截取和保持屏幕画面。该关键字库是使用 Java AWT 的 API 实现的，因此只能在 jython 上使用。

更多信息请参考 Screenshot library。

Remote library

远程控制库同其他标准库存在较大的不同。它没有任何的关键字，但是像代理一样工作于 Robot Framework 和具体的测试库之间。这些库可以运行在其他机器上，甚至可以使用 Robot Framework 本身不支持的语言。

关于本章节的更多信息请参考 Remote library interface 章节。该库是在 Robot Framework 2.1 中新增的。

2.4.4 扩展库

任何不是标准库的测试库，顾名思义，叫做扩展库（External libraries）。Robot Framework 开发者提供了一些库，如 seleniumLibrary 和 SwingLibrary，他们没有同 Robot Framework 一起打包，因为他们需要外部的依赖。通用库也可以由其他组织提供，大多数的组织都有他们自己专属的自定义库。

不同的扩展库可以用完全不同的方式来安装和引入它们。大多数时候，它们需要单独安装其他的依赖。所有的库都需要明确的描述，最好是可以自动安装的。

创建自定义测试库请参考 Creating test libraries 章节。

2.5 变量

2.5.1 介绍

变量是 Robot Framework 的完整特征，它们能在测试数据的大多数地方被使用。最常见的是，在测试用例表和关键字表中被用于关键字的参数，但是所有设置的值中都允许使用变量。

一个普通的关键字名称不能使用变量来指定，但使用内建关键字 Run keyword 可以获得这种效果。

Robot Framework 有两种不同的变量，标量和链表。它们的语法分别是${SCALAR}和 @{LIST}。除了这些，环境变量可以直接使用%{VARIABLE}。

> 小提示
>
> 在 Robot Framework2.5 中不赞同使用标量变量和列表变量使用相同的名称，如${VAR}和@{VAR}。更多信息请参考 issue 484。

在以下几种情况下推荐使用变量：

- 测试数据中字符串经常改变。使用变量后，你只需要在一个地方更改就可以改变其所有。
- 在创建 system-independent 和 operating-system-independent 的测试数据。使用变量代替硬编码（例如：${RESOURCES}代替 c:\resources 或者${HOST}代替 10.0.0.1:8080）。因为变量可以在测试执行时通过命令行来设置，改变系统指定变量（system-specific）很容易（例如：--variable HOST:10.0.0.2:1234 --variable RESOURCES:/opt/resources）。这有利于本地化测试，这往往涉及到相同的测试使用不同的字符串。
- 当需要对象而不是字符串做为关键字的参数。
- 当不同的关键字在不同的测试库中需要通信，你可以将一个关键字的返回值赋给一个变量，并将该变量做为另一个关键字的参数。
- 当测试数据中的值过长或过于复杂。例如：${URL} 比 http://long.domain.name:8080/path/to/service?foo=1&bar=2&zap=42.更短。

如果在测试数据中使用不存在的变量，关键字将会失败。如果需要使用同变量语法相同的字符串必须使用反斜杠进行转义如：\${NAME}。

2.5.2 变量类型

本章节将介绍不同变量类型的优点。在以下的子章节中，将介绍变量的创建和使用详情。Robot Framework 变量，同关键字类似，不区分大小写，所有的空格和下划线都将被忽略。然而，对于全局变量推荐所有字母大写（如：${PATH}或${TWO_WORDS}），如只在当前测试用例或关键字有效的变量建议使用小写（如：${my_var}或${myVar}）。更重要的是，在用例中应该使用一致。

不同于其他编程语言中类似的变量语法，花括号（{和}）在 Robot Framework 的测试数据中是被强制使用的。基本上，变量名可以是花括号中间的任意字符。然而，只有 a-z 的字母、数字、下划线和空格被推荐使用，甚至对扩展变量语法来来说，这个限制是必要条件。

标量变量

当在测试数据中使用标量变量时，他们将被分配的值所代替。当标量变量用作最普通的简单的字符串使用时，你可以将任何对象包括链表分配给它。这种标量变量语法，如${NAME}，对于大多数用户是熟悉的，并且已使用过它们，如在 Shell 脚本或 Perl 编程中。

下列将展示标量变量的使用。假设变量${GREET}和${NAME}可用，并且分配给它们的字符串分别是 hello 和

world。下面展示的 2 个测试用例例子是等效的。

字符串标量变量

Test Case	Action	Argument	Argument
Strings	Log	Hello	
	Log	Hello, world!!	
Variables	Log	${GREET}	
	Log	${GREET}, ${NAME}!!	

当标量变量只是在测试数据单元格中做为一个值使用时，这个标量变量将被分配的值所代替，这个值可以是任何对象。.当一个标量变量在测试数据单元格中与任何其它内容一起使用时（常量字符串或其他变量），它的值首先被转成 Unicode 字符串，然后与该单元格中的其它内容连接起来。将这个值转换为字符串意味着这个对象的方法__unicode__（在 Python 中，__str__做为备用）或 toString（在 Java 中）将被调用。

小提示

变量值是 as-is 不是转换，象使用如 argname=${var}的命名参数语法向关键字传递参数一样。

下面的示例展示了单元格中仅有一个变量和有其他内容的差异。首先，让我们假设变量${STR}被赋值为一个字符串 Hello, world!和${OBJ}被赋值为以下 Java 对象的实例：

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

对于这两个变量设置，我们有以下的测试数据：

以对象为值的标量变量

Test Case	Action	Argument	Argument
Objects	KW 1	${STR}	
	KW 2	${OBJ}	
	KW 3	I said "${STR}"	
	KW 4	You said "${OBJ}"	

最后，当这些测试数据被执行，不同的关键字接受参数如下：

- KW 1 得到字符串 Hello, world!
- KW 2 得到一个存储在${OBJ}中的对象
- KW 3 得到字符串 I said "Hello, world!"
- KW 4 得到字符串 You said "Hi, tellus!"

小提示

假如变量不能被 Unicode 表示，则转换变量到 Unicode 将会失败。例如，假如你试图使用字节序列做为关键字的参数从而可以连接字符串如${byte1}${byte2}可能发生。一种解决办法是创建一个变量包含所有值并在一个单独的单元格中使用（如${byte}），因为该值将被做为 as-is 使用。

列表变量

列表变量是复合变量，可以分配多个值给它。总之，他们可以列出和包含不受限制的条目（空链表也是允许的）。列表变量的最大好处是允许为大数据集合分配一个名称。虽然列表变量通常只包含字符串，但是其他内容也是允许的。

当在测试数据中使用列表变量，那么列表中的元素作为新单元格被插入到测试数据中。因此,假如列表变量包含两个元素,则包含列表变量的单元格将分为两个单元格，且分别赋值为该列表的 2 个元素的内容。注意，包含了列表变量的单元格不能再包含其他内容。列表变量的语法@{name}是来自于 Perl。

假设列表变量@{user}被赋值['robot','secret']，下面两个测试用例是等价的。

使用列表变量

Test Case	Action	User Name	Password
Strings	Login	robot	secret
List Variable	Login	@{USER}	

访问链表中的单个元素

也可以使用如下语法访问列表变量中某个确定的元素@{list}[i]。索引是从 0 开始的，如果试图访问超过最大索引的值将引起错误。通过这种方法访问链表，同标量变量的使用类似。

访问列表变量项

Test Case	Action	Argument	Argument
Strings	Login	robot	secret
	Title Should Be	Welcome robot!	
List Variable	Login	@{USER}	
	Title Should Be	Welcome @{USER}[0]!	

像标量变量一样使用链表

通过将列表变量的@替换为$，可以像标量变量一样简单的使用链表。这样可以使用来自内建库和集合库的链表相关的关键字操作列表变量。这种功能只有在不存在与链表的基础名称相同的标量变量时才可用。这种情况下，标量变量有更高的优先级，将被他的值所替换。

像标量变量一样使用链表

Test Case	Action	Argument	Argument	Argument
Example	@{items} =	Create List	first	second
	Length Should Be	${items}	2	
	Append To List	${items}	third	
	Length Should Be	${items}	3	
	Remove From List	${items}	1	
	Length Should Be	${items}	2	
	Log Many	@{items}		

在设置中使用链表

列表变量仅能用于某些设置中。它们可以作为参数用于引入库和变量文件，但是库和变量文件名称不能是列表变量。同样列表变量不能作为 setup 和 teardown 的关键字名称，只能用做参数。对于 tag 的相关设置则可以随便使用。在那些不支持列表变量的地方，使用标量变量是允许的。

在设置中使用链表

Settings	Value	Value	Comment
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

环境变量

Robot Framework 允许在测试数据中使用环境变量，语法是%{ENV_VAR_NAME}。但仅限于字符串值。环境变

量是在测试执行之前被设置到操作系统中的，也可以使用关键字 Set Environment Variable 创建一个新环境变量或使用关键字 Delete Environment Variable 删除一个已存在的环境变量，这些关键字都是在操作系统库（OperatingSystem library）中。因为环境变量是全局的，只要环境变量在一个测试用例中被设置，则其后的用例都可以使用。但是，测试开始执行之后，对环境变量的改变却不能生效。

Test Case	Action	Argument	Argument
Env Variables	Log	Current user: %{USER}	
	Run	%{JAVA_HOME}${/}javac	

2.5.3 创建变量

在以下章节将介绍变量存在于多个资源中。

变量表格

最常见的变量来源是测试用例文件和资源文件中的变量表格。变量表格是方便的，因为它允许在同一个地方创建一个变量，并且所需的语法非常简单。他们最大的缺点是只能给其赋值为字符串或字符串链表。假如需要其他类型值，变量文件将是更好的选择。

创建标量变量

最简单的变量赋值是对标量变量设置一个字符串值。在变量表格的第一列填入变量名（包含${}），在其后的第二列填入赋予的值。假如第二列为空，空字符串被赋值给该变量。同时，一个已经定义的变量也可以作为新变量的值。

创建标量变量

Variable	Value	Value
${NAME}	Robot Framework	
${VERSION}	2.0	
${ROBOT}	${NAME} ${VERSION}	
${EMPTY}		

假如可能，但不是必须的，使用等号（=）在变量名后面可以使变量赋值更清楚。

Variable	Value	Value
${NAME} =	Robot Framework	
${VERSION} =	2.0	

创建列表变量

创建列表变量同创建标量变量一样容易。同样，变量名称在第一列，值跟随在其后的列中。一个列表变量可以有任意多个值，也可以有 0 个，假如需要很多值，它们可以被分割成多行。

Variable	Value	Value	Value
@{NAMES}	Matti	Teppo	
@{NAMES2}	@{NAMES}	Seppo	
@{NOTHING}			
@{MANY}	one	two	three
...	four	five	six
...	seven		

变量文件

变量文件是一种非常强大的方式用于创建各种类型变量的机制。它可以为变量赋值任何对象，也能动态的创建变量。变量文件的语法以及如何使用变量文件将在资源和变量文件章节介绍。

在命令行中设置变量

变量可以在命令行中设置，个别变量设置使用--variable (-v)选项，变量文件的选择使用--variablefile (-V)选项。通过命令行设置的变量是全局变量，对其所有执行的用例都有效。它们将覆盖变量表格中的同名变量或是通过变量文件引入到测试数据中的同名变量。

设置单个变量的语法是--variable name:value，“name”是不使用${}的变量名称，“value”则是其赋予的值。可以多次使用该选项设置多个变量。仅有标量变量可以使用该语法且只能赋值字符串。许多特殊字符在命令行中很难表示，但可以使用转义字符转义它们，使用—escape 选项。

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

在上面的例子中，变量获取的值如下：

- ${EXAMPLE} gets the value value
- ${HOST} and ${USER} get the values localhost:7272 and robot
- ${ESCAPED} gets the value "quotes and spaces"

在命令行中使用变量文件的基本语法是--variablefile path/to/variables.py，更多信息将在 Taking variable files into use 章节中介绍。哪些变量实际被创建依赖于引用的变量文件中存在哪些变量。

对于变量文件和通过命令行设置的单个变量来说后者有更高的优先级。另外，如果多个变量文件有相同的变量，第一个指定的变量将被使用。

从关键字中返回值

通过关键字返回的值也可以赋值给变量。这就使得不同的关键字之间可以通信，即使它们在不同的测试库中。下面列子将简单介绍这种语法

Test Case	Action	Argument	Argument
Returning	${x} =	Get X	an argument
	Log	We got ${x}!	

在上面例子，通过 Get X 关键字返回的值将赋值给${x}，并且在 Log 关键字使用它。这种语法适用于所有关键字返回值，这个变量可以赋予关键字返回的任何值。在变量名称后面的等号（=）不是必须的，但是我们推荐，因为可以使赋值更清晰。

假如关键字返回一个链表，它可以被赋值给多个标量变量和/或一个列表变量。从 Robot Framework2.5 开始，它可以工作于所有类似链表的对象，但是在之前它仅支持 Python 链表、tuples 和 Java 的队列。

Test Case	Action	Argument	Argument	Argument
Return Multiple	${scalar} =	Get 3		
	${a}	${b}	${c} =	Get 3
	${first}	@{rest} =	Get 3	
	@{list} =	Get 3		

假设关键字 Get 3 返回一个链表[1, 2, 3]，以下变量将被创建：

- ${scalar}其值是[1, 2, 3]
- ${a},${b},${c}的值分别是 1,2,3
- ${first}值是 1，@{list}值是[2,3]
- @{list}值是[1,2,3]

通过这种方式创建的变量同其他任何变量是类似的，但他们的作用域范围仅是该测试用例和关键字范围。因此，举例来说，如果在一个测试用例中设置一个变量而在另一个用例中使用，这是不可能的。这是因为，在一般情况下，自动化测试用例不会互相依赖，意外的设置了一个在别处使用的变量将引起难以调试的错

误。假如需要在一个用例中设置变量而在另一个用例中使用，需要使用下一章节中的内建关键字。

使用 built-in 关键字： Set Test/Suite/Global Variable

在内建库中有关键字 Set Test Variable, Set Suite Variable 和 Set Global Variable 用于在测试过程中动态设置变量。假如一个变量已经存在于新的作用域中，它的值将被覆盖，否则将创建一个新的变量。

使用 Set Test Variable 创建的变量，可以在该测试用例范围内的任何位置有效。例如，你在一个用户关键字中创建了变量，它将在测试用例级别有效及当前测试中的所有其他用户自定义关键字中有效。其他测试用例将看不到该变量。

使用 Set Suite Variable 创建的变量，可以在当前测试集范围内的任何位置有效。使用该关键字创建变量同使用测试数据文件的设置表格设置变量和引入变量文件相同。其他的关键字，包括可能的子测试集，都将不能看到该关键字创建的变量。

使用 Set Global Variable keyword 创建的变量，在所有测试用例和测试集中有效。通过该关键字创建的变量同使用命令行选项-variable 或—variablefile 创建的变量相同。因为该关键字可以在任何地方改变变量值，所有需要小心使用。

2.5.4 内建变量

Robot Framework 提供了一些内建变量，这些变量是自动可以使用的。

操作系统相关的变量

操作系统相关的内建变量使其可以轻松使用操作系统。

有效的操作系统内建变量

Variable	Explanation
${CURDIR}	提供当前测试文件存放的绝对路径。该变量是大小写敏感的。
${TEMPDIR}	获取操作系统临时文件夹的绝对路径。在UNIX 系统是在/tmp，在windows 系统是在c:\Documents and Settings\<user>\Local Settings\Temp.
${EXECDIR}	获取测试执行开始目录的绝对路径。该变量是在 Robot Framework2.1 新加的。

Variable	Explanation
${/}	操作系统的路径分隔符。在 UNIX 系统是“/”，在 windows 系统是“\”。
${:}	系统路径元素分隔符。在 UNIX 中是“:”在 windows 系统是“;”。

使用操作系统相关的内建变量

Test Case	Action	Argument	Argument
Example	Create File	${CURDIR}${/}input.data	Some text here
	Set Environment Variable	CLASSPATH	${TEMPDIR}${:}${TEMPDIR}${/}foo.jar

数字变量

变量语法可以用来创建一个全是整型和浮点型的数字，如下列所示。当关键字需要真实数字的时候这是非常有用的。它不是看起来像数字的字符串，而是一组数字。

Test Case	Action	Argument	Argument	Comment
Example 1A	Connect	example.com	80	Connect 获得了两个字符串参数
Example 1B	Connect	example.com	${80}	Connect 获得了一个字符串和一个整数
Example 2	Do X	${3.14}	${-1e-4}	Do X 获得了浮点数 3.14 和 -0.0001

Boolean 和 None/null 变量

就像创建数字变量一样，也可以使用变量语法创建布尔值、Python 语法中的 None 和 Java 语法中 Null。

使用 Boolean 和 None/null 变量

Test Case	Action	Argument	Argument	Comment
Boolean	Set Status	${true}		设置 Status 值为布尔值的 true。
	Create Y	something	${false}	创建变量 Y 获得字符串变量和布尔值 false。
None	Do XYZ	${None}		获取参数 Python 中的 None

Test Case	Action	Argument	Argument	Comment
Null	${ret} =	Get Value	arg	检查变量获取的值是否同 Java 的 null 是否相同
	Should Be Equal	${ret}	${null}	

这些参数对大小写不敏感。例如${TRUE}和${true}是相同的，另外，${none}和${null}是同义词，因为在 jython 解释器上运行测试用例时，jython 根据需要自动将 None 和 null 转换成正确的形式。

空格和空变量

可以使用${SPACE}和${EMPTY}创建空格和空变量。这些变量非常有用，否则需要使用反斜杠用于转义空格或空单元格。当需要使用多个空格时，可以使用该语法${SPACE*5}。如下面的例子，should be equal 使用了同样的参数，但使用该语法比使用反斜杠更简单清晰。

使用空格和空变量

Test Case	Action	Argument	Argument
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}	\

自动变量

一些自动变量可以在测试用例中使用。这些变量在执行时会有不同的值，并且他们不是在任何时候都是有效的。

Variable	Explanation	Available
${TEST NAME}	获取当前测试用例名称	Test case
@{TEST TAGS}	包含当前测试集中的 tag，按字母排序	Test case
${TEST STATUS}	获取当前测试用例的状态，PASS 或 FAIL 中的一种。只能在测试用例的 Teardown 中使用	Test teardown
${TEST MESSAGE}	当前测试用例可能的错误信息。只能在测试用例的 Teardown 中使用	Test teardown

Variable	Explanation	Available
${PREV TEST NAME}	前一个测试用例的名称，或者为空字符串【当没有任何测试用例执行过】。可以在任何地方使用	Everywhere
${PREV TEST STATUS}	前一个测试用例的状态，PASS，FAIL 或者未空字符串，【当没有任何用例执行过】。可以在任何地方使用	Everywhere
${PREV TEST MESSAGE}	前一个测试用例可能的错误信息。可以在任何地方使用	Everywhere
${SUITE NAME}	当前测试集的全名称。可以在任何地方使用。	Everywhere
${SUITE SOURCE}	测试集文件或目录的绝对路径，Robot Framework2.5 新增变量。可以在任何地方使用。	Everywhere
${SUITE STATUS}	当前测试集的状态，PASS 或 FAIL。只能在测试集的 teardown 中使用。	Suite teardown
${SUITE MESSAGE}	测试集的所有消息，包括统计。只能在测试集的 teardown 中使用。	Suite teardown
${OUTPUT FILE}	当前输出文件的绝对路径。当输出文件被分割会有不同的值。可以在任何地方使用。	Everywhere
${LOG FILE}	当前日志文件的绝对路径或字符串 None【当没有日志文件时】。当输出被分割时，将可能有不同的值。	Everywhere
${REPORT FILE}	当前报告文件的绝对路径或字符串 NONE【当没有报告文件生成时】。	Everywhere
${SUMMARY FILE}	摘要文件的绝对路径或字符串 NONE【当没有摘要文件时】。	Everywhere
${DEBUG FILE}	调试文件的绝对路径或字符串 NONE【当没有调试文件时】。	Everywhere
${OUTPUT DIR}	输出目录的绝对路径	Everywhere

2.5.5 变量的优先级和范围

变量优先级

来自命令行的变量

通过命令行设置的变量具有最高的优先级。它将可能覆盖测试文件中变量表中设置的变量，以及在资源文

件和变量文件中引入的变量。

单个设置的变量（--variable option)将覆盖使用引入变量文件中的变量（(--variablefile option）。另外，当多个变量文件中具有相同的变量，则第一次指定的变量将具有最高的优先级。

测试用例文件中的变量表

在测试用例文件中变量表创建的变量在该文件中的所有测试用例有效。这些变量将覆盖通过资源文件和变量文件引入的变量。

在测试用例文件中变量表创建的变量在该文件中所有表格有效。这意味着它可以用于设置表格，例如通过资源文件和变量文件引入更多变量。

引入资源文件和变量文件

通过资源文件和变量文件引入的变量在测试数据创建的所有变量中具有最低的优先级。资源文件和变量文件引入的变量具有相同的优先级。如果多个资源文件或变量文件存在相同的变量，则第一个指定的变量将被使用。

假如资源文件又引入了另外的资源文件或变量文件，该资源文件的变量表中的变量比引入的变量具有更高的优先级。所有引入资源变量的这些文件中的变量都可用。
注意，从资源文件和变量文件引入的变量在变量表中是不可用的，因为变量表的处理是早于设置表的。

在执行过程中设置的变量

在执行过程中通过关键字返回值或关键字 Set Test/Suite/Global Variable 设置的变量将覆盖他们作用域的相同变量。在某种意义上，这些变量具有最高的优先级，但另一方面，他们在作用域之外是无效的。

内置变量

内置变量如${TEMPDIR}和${TEST_NAME}在所有变量中具有最高的优先级。他们不能被变量表或命令行变量覆盖，但可以在测试执行时被重置。该规则的一种例外是数字变量，如果在别处没有变量被找到，它将被动态执行。因此它们也可能被覆盖，但这不是个好习惯。另外${CURDIR}是个例外，在测试执行过程中它可能已经被改变了。

变量范围

变量范围需要依赖该变量是在哪里创建的，以及是如何创建的，变量范围有全局的、测试集、测试用例和

用户关键字范围。

全局范围

全局变量在测试数据的任何地方都是有效的。这些变量通常是通过命令行中的选项—variable 和—variablefile 设置，但是也可以通过内建关键字 Set Global Variable 创建新的全局变量或更改已存在的变量。另外，内建变量也是全局变量。建议全局变量全部使用大写字母。

测试集范围

这些变量在定义了或引入了它们的测试集范围内有效。他们可以是在变量表中创建或通过资源和变量文件引入，或者是在测试执行时使用内建关键字 Set Suite Variable 在执行时创建。

测试集范围不是递归的，这意味着这些变量只在高级别的测试集中有效，在更低级别的测试集中是不可用的。如果需要，资源和变量文件可以用来共享变量。

因为这种变量可以是测试集中全局有效，因此推荐使用全部大写字母。

测试用例范围

通过测试用例中关键字返回值创建的变量只有测试用例范围，仅在测试用例中有效。通过内建关键字 Set Test Variable 创建的变量在特定的测试用例的任何位置都有效。测试用例变量是局部变量，使用小写字母。

用户关键字范围

用户关键字从参数获取他们的变量和使用关键字返回值。这些变量也是局部的，使用小写字母。

2.5.6 高级变量特性

扩展变量语法

扩展变量语法可以在标量变量中使用对象。它们可以访问对象的属性（如：${obj.name}或${obj.some_attr}），和调用对象的方法（如：${obj.get_name()}或${obj.getSomething('arg')}）。

扩展变量语法是很强大的特性，但是在使用上需要注意。访问属性通常不是问题，相反一个变量对应一个有多个属性值的对象比有多个变量要更好。另一方面，调用方法将使测试数据变得复杂，特别是方法使用参数的时候。假如发生这些，建议将这段代码移动到测试库中。

下面的例子将介绍扩展变量语法的最普通用法。首先假设我们有以下的变量文件和测试用例：

```
class MyObject:

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

    def greet(self, who):
        return '%s says hello to %s' % (self.name, who)

    def __str__(self):
        return self.name

OBJECT = MyObject('Robot')
DICTIONARY = { 1: 'one', 2: 'two', 3: 'three'}
```

Test Case	Action	Argument	Argument
Example	KW 1	${OBJECT.name}	
	KW 2	${OBJECT.greet('Fit')}	
	KW 3	${DICTIONARY[2]}	

当这个测试数据被执行，以下关键字将得到如下的参数：

- KW1 获得字符串 Robot
- KW2 获得字符串 Robot says hello to Fit
- KW3 获得字符串 two

扩展变量语法是按照以下顺序运算的：

1、通过变量的全名称搜索变量。只有在没有找到匹配的变量时扩展变量语法才被运算。

2、创建基础变量的真实名称。该名称体包括在${之后到第一个非字母字符或空格之间的所有字符。（如，OBJECT 在${OBJECT.name}中和 DICTIONARY 在${DICTIONARY[2]中}）。

3、搜索与该名称匹配的变量。一个变量如果没有匹配，将会出现一个异常且测试用例将失败。

4、在一个大括号中的表达式计算同 Python 表达式类似，因此这个基础，与上面对应变量名称将被它的值所替代。如果由于非法的格式或者这个查询属性不存在导致计算失败，这次测试将失败并产生一个异常。

5、用计算得到的值替换整个扩展变量。

假如被使用的对象是 Java 的实现，这个扩展变量语法允许使用 so-called Bean 属性访问属性。本质上，这意味着假如你有一个对象${OBJ}且其中有个一个方法 getName，这时语法${OBJ.name}与${OBJ.getName（）}是等价的，但是比${OBJ.getName（）}更清晰。因此，前面例子中的 Python 对象使用可以被以下的 Java 实现所替代：

```
public class MyObject:

    private String name;

    public MyObject(String name) {
        name = name;
    }

    public String getName() {
        return name;
    }

    public String greet(String who) {
        return name + " says hello to " + who;
    }

    public String toString() {
        return name;
    }
}
```

许多标准的 Python 对象，包括字符串和数字，都可以使用扩展变量语法中的方法。有些时候这是非常有用的，可以减少临时变量的使用，但是也很容易造成过度使用，创建一些隐秘的测试数据。下面例子展示了一些不错的用法：

使用字符串和数字的方法

Test Case	Action	Argument	Argument
String	${string} =	Set Variable	abc
	Log	${string.upper()}	# Logs 'ABC'
	Log	${string * 2}	# Logs 'abcabc'
Number	${number} =	Set Variable	${-2}
	Log	${number * 10}	# Logs -20
	Log	${number.__abs__()}	# Logs 2

注意，在标准 Python 代码中，更推荐使用 abs(number)而不是 number.__abs__()，使用${abs(number)}将不能正常工作，因为在扩展语法中变量名称必须是开头的。在测试数据中使用_xxx_方法是有争议的，更好的做法是将这部分逻辑移到测试库中。

嵌套变量

变量被允许插入到变量中，这种语法是可以使用的，变量是由内到外解析的。例如，你有一个变量${var${x}}，则${x}首先被解析。假如它有一个值 name，则该变量的最终值是${varname}。这里可以嵌套多个变量，但是假如有任何一个变量值不存在则全部失败。

如下列所示，Do X 获取一个值${JOHN_HOME}或者${JANE_HOME}，依靠 Get Name 关键字返回值 john 或者

jane。假如返回其他的值，处理变量${${name}_HOME}则会失败。

使用嵌套了变量的变量

Variable	Value	Value	Value
${JOHN_HOME}	/home/john		
${JANE_HOME}	/home/jane		

Test Case	Action	Argument	Argument
Example	${name} =	Get Name	
	Do X	${${name}_HOME}	

2.6 创建用户关键字

关键字表格使用已经存在的关键字组合来创建新的更高级别的关键字。这些关键字被叫做用户自定义关键字，以区别于低级别的通过测试库实现的库关键字。创建用户关键字的语法同创建测试用例的语法非常类似，这更易于学习。

2.6.1 使用关键字的语法

基本语法

在许多方面，用户关键字语法同测试用例语法相同。用来创建用户关键字的关键字表格不同于由名字唯一标识的测试用例表格。位于第一列的用户关键字名称同测试用例名称类似。用户关键字可以通过关键字创建，可以是测试库中的关键字也可以是其它的用户关键字。关键字名称通常在第二列中，但是当设置关键字返回值时，他们在其后的列中。

Keyword	Action	Argument	Argument
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}

大多数用户关键字有多个参数。这重要的特性已经在上面的第二个例子中使用了，本节将详细介绍，与前一句换一下顺序。

用户关键字可以在测试用例文件、资源文件和测试集初始化文件中被创建。在资源文件中创建的关键字可以在使用了该资源文件的文件中有效，其它关键字只能在创建他们的文件中有效。

关键字表格的设置

用户关键字和测试用例有相似的设置，它们具有相同的被关键字名称分开的方括号语法。以下是所有有效地设置，将在本章节后续介绍

[Documentation]

用于关键字的注释

[Arguments]

指明用户关键字的参数

[Return]

指明用户关键字的返回值

[Timeout]

设置用户关键字的可能超时。超时将在它们自己的章节中介绍

2.6.2 用户关键字名称和注释

用户关键字名称被定义在用户关键字表的第一列。当然，这名称应该是描述性的，可以接受较长的关键字名称。事实上，当创建 use-case-like 的测试用例时，最高级别的关键字通常是通过句子甚至段落表示。

用户关键字的注释是使用[Documentation]设置，同测试用例的注释类似。这个在测试数据中设置用户关键字注释。它可以在较正式的关键字文件中显示，通过 libdoc.py 工具可以从资源文件中创建。最后，注释的第一行直到第一个\n 序列之间的所有东西，在测试日志中作为关键字的注释显示。

有些时候关键字需要被删除、被新的所替换或者由于其他原因被不赞成使用，用户关键字可以通过在注释的开头使用*DEPRECATED*表示，当该关键字被使用时将引起一个警告。更多信息请参考 Deprecating 关键字章节。

2.6.3 用户关键字参数

大多数用户关键字都需要一些参数。用于指定他们的语法大概是 Robot Framework 中最复杂一种特性了，但是他们也是相对容易的，特别是在大多数用例中。参数通常是使用[Arguments]指定，参数的名称同变量具有相同的语法，如${arg}。

位置参数

指定参数（只需要部分不是全部参数）最简单的方法是只使用位置参数。在大多数情况，这个已经足以满足需要。

该语法是这样的，首先是设定[Arguments]，然后在后续的单元格填入参数名称。每个参数在它们各自的单元格中，使用同变量相同的语法。使用关键字时其参数个数必须与定义它时的相同。实际的参数名称对于框架是无关紧要的，但从用户的视角需要参数名尽量是有注释意义的。建议使用小写字母的变量，如${my_arg}, ${my arg} or ${myArg}。

不同个数参数的使用

Keyword	Action	Argument	Argument	Argument
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}		

参数默认值

在大多数情况下，位置参数已经充分够用了。但是，有时允许一个关键字有不同数量的参数，且对那些没有给定值的参数赋值为默认值是非常有用的。用户自定义关键字同样支持这种情况，且在已有的基础语法之上不用增加太多的新语法。简而言之，参数添加默认值，即在等号(=)后跟默认的值。例如：${arg}=default。这里可以有多个参数有默认值，但是他们必须在正常位置参数后面。

关键字使用有默认值的参数

Keyword	Action	Argument	Argument
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}=default 2
	[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}	

当一个关键字需要多个有默认值的参数并且只有其中一部分参数需要被覆盖，使用命名关键字语法很容易做到。这种语法可以用于用户自定义关键字，指定的参数不使用${}。例如，上例中的第二个关键字可以如下使用，${arg1}仍使用默认值。

用户关键字命名参数语法。

Test Case	Action	Argument	Argument
Example	Two Arguments With Defaults	arg2=new value	

所有的 Pythonistas 已经注意到了，这种指定默认参数的语法是来自于 python 函数的参数默认值。

可变数量参数

有时，但这种情况很少，默认值不能满足需要，关键字可能需要任意数量的参数。用户自定义关键字支持这种情况。这需要在关键字定义时使用列表变量如@{ varargs }。这种语法可以同位置参数和默认值参数组合使用，在最后的列表变量会获取剩余的没有被匹配的参数。列表变量可以有任意数量的条目，甚至 0。

关键字接收可变数量的参数

Keyword	Action	Argument	Argument	Argument
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}	

注意假如上例中的最后一个关键字使用的参数多余 1 个，那么第二个参数${opt}将始终获得赋值而不是默认值，即使给定的赋值是空。最后一个例子展示了在循环中用户关键字如何接收可变数量参数。

2.6.4 关键字名称嵌入参数

Robot Framework 从 2.1.1 版本开始，除了如上一节介绍的在单元格中的关键字名称后面指定参数外，还有另外的途径传递参数给用户自定义关键字。这种方法是基于在关键字名称中嵌入参数，这样最大的好处是可以使用真实的清晰的句子做为关键字。它使得一个关键字可以使用如 Select dog from list 和 Select cat from list 这样的句子，而其他的方法只能分别实现关键字。使用嵌入式参数只需要关键字的名字如 Select ${animal} from list 即可。

Keyword	Action	Argument	Argument
Select ${animal} from list	Open Page	Pet Selection	
	Select Item From List	animal_list	${animal}

使用嵌入式参数的关键字不能有任何的普通参数（使用[Arguments]设置的），但是其他同创建用户自定义关键字类似。在关键字名称中的参数在关键字内是有效的，它们获得的不同的值是依靠关键字如何被调用。例如：假如关键字是做为 Select dog from list 使用，则上例中的${animal}获得值 dog。显然它不是强制使用关键字中的所有参数，可以作为通配符使用。

这些关键字的用法与其它关键字相同，除了在这些名字里空格和下划线不被忽略。它们也是大小写不敏感的。例如上例中关键字可以这样使用 select x from list 而 Select x fromlist 就是不允许的。

嵌入式参数不支持默认值，但与传统参数一样支持可变数量参数。在调用这些关键字的时候使用变量是可行的，但是会降低可读性。注意，嵌入式参数仅能工作在用户自定义关键字中。

嵌入参数匹配过多

使用嵌入式参数时的一个棘手问题是调用关键字时匹配正确的参数值。这是一个问题，尤其是假如这里有多个参数和分开的字符。例如 Select ${city} ${team}假如城市 city 包含多个部分如 Select Los Angeles Lakers，将导致出现非预期的结果。一种解决办法是将参数用引号包围如 Select "${city}" "${team}"或者将他们互相分开，但是有时使用位置参数代替更简单些。

同样的问题经常出现，当试图创建忽略 given/when/then/and 前缀的关键字时。例如${name}goes home 配置 Given Janne goes home，因此${name}得到值 Given Janne。使用引号可以解决，如”${name}”goes home。

行为驱动开发例子

使用参数做为关键字名称的一部分最大的好处是在使用行为驱动模式开发测试用例时创建句子形式的高级关键字更容易。以下例子将介绍。注意前缀 Given, When 和 Then 将在关键字定义之外。

Test Case	Step
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

Keyword	Action	Argument	Argument
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}

2.6.5 用户自定义关键字返回值

同库关键字类似，用户关键字也可以返回值。返回值使用[Return]定义。该值可以赋给测试用例中的变量或其他关键字。

在典型例子中，一个用户关键字返回一个值，并被设置到标量变量中。这时将返回值放在[Return]之后的单元格。用户关键字也可以返回多个值，可以一次性赋给多个标量变量或一个列表变量或标量变量和列表变量。返回多个值是将指定的值放在[Return]后面的不同单元格。

Test Case	Action	Argument	Argument	Argument
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	

Keyword	Action	Argument	Argument	Argument
Return One Value	[Arguments]	${arg}		
	Do Something	${arg}		
	${value} =	Get Some Value		
	[Return]	${value}		
Return Three Values	[Return]	foo	bar	zap

2.7 资源和变量文件

测试用例文件和测试集初始化文件中的用户关键字和变量只能在创建它们的文件中使用，但是资源文件提供了共享他们的机制。由于资源文件很接近于测试用例文件，因此很容易创建它们。

变量文件提供了强有力的方法来创建和共享变量。例如，它允许除字符串之外的其它值，且可以动态创建变量。这种灵活性来自于 python，因此可以比变量表有更多的灵活性。

2.7.1 资源文件

使用资源文件

资源文件是在设置表格中被引入的。资源文件的路径放在设置名称的后面。

假如使用绝对路径，资源文件可以被直接使用。否则，首先是在导入该资源的文件的所在目录去搜索相关的路径，假如没找到，然后在 PYTHONPATH 路径下搜索。这个路径可以包含变量，建议使用操作系统无关的路径（例如：${RESOURCES}/login_resources.html 或${RESOURCE_PATH}）。这样，路径中的斜杠（/）在 Windows 中自动转变为反斜杠（\）。

导入资源文件

Setting	Value	Value
Resource	myresources.html	
Resource	../data/resources.html	
Resource	${RESOURCES}/common.tsv	

定义在资源文件中的用户关键字和变量在引入了资源文件的文件中是有效的。同样，该资源文件引入的所有库关键字库变量、资源文件好变量文件在引入了该资源文件的文件中也是有效的。

资源文件结构

资源文件的高层架构同测试用例文件是相同的，但是，它不包括测试用例表格。资源文件中的设置表格可以仅引入设置（(Library, Resource, Variables）和注释。变量表格和关键字表格同测试用例文件中一样使用。

假如多个资源文件包含多个相同名称的用户关键字，则他们的使用必须在关键字前面加上资源文件名称（例如：myresources.Some Keyword 和 common.Some Keyword）。而且，假如资源文件包含相同的变量，则第一个引入的将被使用。

注释资源文件

在资源文件中创建关键字可以使用[Documentation]注释。从 Robot Framework2.1 开始，资源文件自身可以在设置表格中进行注释，同测试集类似。

libdoc.py 和 RIDE 都可以使用这注释，他们对任何打开资源文件都是有效的。当它运行时关键字的注释的第一行将被日志记录，但是资源文件的其它注释将被忽略。

Setting	Value	Value	Value
Documentation	An example resource file		
Library	SeleniumLibrary		
Resource	${RESOURCES}/common.html		

Variable	Value	Value	Value
${HOST}	localhost:7272		
${LOGIN_URL}	http://${HOST}/		
${WELCOME_URL}	http://${HOST}/welcome.html		
${BROWSER}	Firefox		

Keyword	Action	Argument	Argument	Argument
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}	

2.7.2 变量文件

变量文件包含的变量在测试数据中可以被使用。使用变量表格或命令行设置可以创建变量，但是变量文件可以动态创建变量且，这些变量可以包含任何对象。

变量文件使用 Python 代码创建，技术上他们是 Python 模块。这里有两种创建变量的可能：

直接创建变量

变量做为模块属性被指定。在简单用例中，这种语法很简单，不需要实际的编程。例如：MY_VAR = 'my value' 创建一个变量${MY_VAR}，指定一个文本做为值。

通过指定函数创建变量

同 Python 的 dictionary 或 Java 的 map 一样，变量文件可以通过指定的 get_variables（或 getVariables）方法返回变量。这种方法甚至可以接收参数，相当灵活。

使用变量文件

设置表格

在设置表格中，所有的测试数据文件可以通过使用 Variables setting 引入变量，这种方法同使用 Resource

setting 引入资源文件相同。同资源文件类似，引入变量的文件的路径是相对于引入文件所在的目录，假如没有找到则在 PYTHONPATH 目录下寻找。这个路径可以包括变量，在 windows 中斜线将被转换为反斜线。假如一个参数文件包含参数，他们在路径后面的单元中被指定，同样可以包含变量。

引入变量文件

Setting	Value	Value	Value
Variables	myvariables.py		
Variables	../data/variables.py		
Variables	${RESOURCES}/common.py		
Variables	taking_arguments.py	arg1	${ARG2}

变量文件中的所有变量在引入它的测试数据文件中都是有效的。假如多个引入的变量文件包含有相同名称的变量，则最早引入的那个变量将被使用。 在变量表中创建的变量和通过命令行设置的变量将覆盖变量文件中的变量。

命令行

另一种使用变量文件的方法是在命令行中使用—variablefile 选项。变量文件使用路径引用，使用冒号（:）加入参数。如：

--variablefile myvariables.py

--variablefile path/variables.py

--variablefile /absolute/path/common.py

--variablefile taking_arguments.py:arg1:arg2

变量文件中的变量在所有测试数据文件中都是有效的，类似于使用—variable 选项设置的单个变量。假如 --variablefile 和—variable 都被使用了，且它们含有相同名称的变量，则使用—variable 设置的单个变量优先级更高。

直接创建变量

基本语法

当变量文件被使用，它们作为 Python 模块被引入，且它们的所有不以下划线（_）开头的全局属性都被看作是变量。因为变量名字是大小写不敏感的，小写和大写名字都有可能，但是通常推荐使用大写字母作为全局变量。

VARIABLE = "An example string"

ANOTHER_VARIABLE = "This is pretty easy!"

INTEGER = 42

STRINGS = ["one", "two", "kolme", "four"]

NUMBERS = [1, INTEGER, 3.14]

在上例中，创建了${VARIABLE}、 ${ANOTHER_VARIABLE}等多个变量。头两个变量是字符串，第三个是一个整型，后两个是链表。所有的变量都是标量变量，尽管有包含链表作为值的。创建列表变量，变量名称必须使用前缀 LIST__（注意是两个下划线）。

LIST__STRINGS = ["list", "of", "strings"]

LIST__MIXED = ["first value", -1.1, None, True]

上面例子中的变量同样可以使用下面的变量表格创建

Variable	Value	Value	Value	Value
${VARIABLE}	An example string			
${ANOTHER_VARIABLE}	This is pretty easy!			
${INTEGER}	${42}			
${STRINGS}	one	two	kolme	four
${NUMBERS}	${1}	${INTEGER}	${3.14}	
@{STRINGS}	list	of	strings	
@{MIXED}	first value	${-1.1}	${None}	${True}

使用对象作为值

在变量文件中的变量没有限制只能使用字符串或其他基本类型。这些变量可以包含任意的对象。如下面例子，变量${MAPPING}包含一个含有两个值的 Java 哈希表（这个例子只能在 jython 上运行）。

```
from java.util import Hashtable

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

第二个例子创建的变量${MAPPING}像 Python 的 dictionary 一样，同样也包含 2 个来自于同一个变量文件的自定义对象变量。

```
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

因为变量文件是使用真正的编程语言创建的，他们有动态的逻辑设置变量。

```
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
```

上面例子使用标准 Python 库设置了不同的变量，但是你可以使用已有的代码构建值。下面的例子说明了这个概念，同样地，你的代码可以从数据库中读取数据，或是从一个外部文件甚至让用户输入。

```
import math

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

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

选择变量引入

当 Robot Framework 处理变量文件时，所有的不以下划线开头的属性都被看作变量。这意味着不管是在变量文件中创建的或是从其他地方引入的函数还是类都被认为是变量。例如，最后一个例子包含变量${math}和${get_area}以及${AREA1}和${AREA2}。

通常额外的变量不会造成问题，但是他们可能会覆盖同名的其他变量导致调试困难的错误。忽略他们的一种方法是在前面增加前缀下划线（_）：

```
import math as _math

def _get_area(diameter):
    radius = diameter / 2.0
    area = _math.pi * radius * radius
    return area

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

假如这里有大量的其他属性都需要增加前缀，经常使用的简单方法是只有一个特别的属性_all_并给出需要做为变量处理的属性名称链表。

```
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)
```

小贴士

当使用 import *格式引入模块时，_all__属性也被 Python 用来决定哪些属性需要引入。

通过指定的方法获取变量

获取变量的另一个方法是变量文件中有指定的 get_variables 函数（getVariables 也可以）。在这种情况下，Robot Framework 调用方法并返回变量做为 Python 字典，子类或 Java map，变量名称和变量值作为键值对。（Robot Framework2.5 以前版本只支持 Python 字典）。变量被认为是标量，链表需要前缀 LIST__，值可以包含任何东西。下面的例子与“直接创建变量”的第一个例子相同。

```
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],
                  "LIST__STRINGS": ["list", "of", "strings"],
                  "LIST__MIXED": ["first value", -1.1, None, True] }
    return variables
```

get_variables 也可以获取参数，这使更改已经创建的变量更容易。方法的参数设置同 Python 的其他方法设置相同。当在测试数据中使用变量文件时，参数放在变量文件路径后面的单元格，在命令行中使用是在路

径后面使用冒号分隔。

以下虚构的例子展示变量文件如何使用参数。在更多的实际例子中，参数可以使用外部文本文件或数据库。

```
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 get_variables 而不是直接在变量文件中定义全局属性作为变量的最大的好处是可以使用参数。另一方面，最大的缺点是需要进行实际的编程。

2.8 高级特性

2.8.1 处理同名关键字

Robot Framework 中所用的关键字分为库关键字 和 用户自定义关键字. 前者出自 标准库 或者 扩展库, 后者既可以在关键字所在文件中创建也可以从源文件中引用。当一个测试用例中用到很多关键字时，难免这些关键字中会有重名的关键字，本节重点讲述如何处理由于关键字重名而可能造成的冲突。

关键字作用域

当用到一个同名关键字时，Robot Framework 将要判断哪个关键字在它的作用域中拥有最高的优先级。一个关键的作用域取决于它是如何创建的，有如下几种方式:

1、当前文件中创建的关键字。这类关键字最常用而且在所有同名关键字中拥有最高的优先级。

2、一个源文件创建的关键字且直接或间接地被其它源文件引用。这类关键字拥有第二高的优先级。

3、扩展测试库的关键字。当其它地方没有同名关键字时，这这些关键字将被使用。但是，如果在标准库中有同名的关键字的话将会有警告提示出现。

4、标准库中的关键字。这类关键字拥有最低的优先级。

显式指定关键字

仅仅靠关键字的作用域来判断并不是完美的解决方案，因为很有可能在几个不同的库或者源文件里面存有同名关键字，这样的话 Robot Framework 会机械地按最高优先级来选择关键字，但是有些情况很可能不是用户所希望看到的。这种情况下，一个可行的解决方案是使用关键字的全名，也就在关键字前用资源文件名或库名和点分界符作为前缀。

对于库关键字来说，长格式意味着使用“库名. 关键字名”的格式。例如，OperatingSystem library 中的 Run 关键字可以描述为 OperatingSystem.Run, 从而避免了与其它地方有 Run 关键字的冲突。如果一个测试库在一个模块或一个包中，那么模块或包的全名必须加上（例如，com.company.Library.Some Keyword）。如果一个自定义名字通过命名语法库给定，这个特定的名字也必须用关键字全名中。

像库名一样，也要在关键字全名中指定资源文件，这里资源文件名指的是除去扩展名以外的文件基本名。例如，myresources.html 源文件中关键字 Example 应该这样描述：myresources.Example。注意，当存在同名资源文件含有同名关键字时这种语法将不起作用。在这种情况下，同名资源文件和同名关键字至少改一个。像正常关键字一样，全名关键字对大小写、空格、下滑线不敏感。

指定关键字库的搜索顺序

Robot Framework 从 2.1 版起可以用 BuiltIn 库中的 Set Library Search Order 关键字来设定可用测试库的搜索优先级。当遇到多个关键字的情况，Robot Framework 按设定的搜索优先级来执行关键字。当优先级最高的库中含有所需关键字，则该库中的关键字被执行，若关键字未在所有库中找到，则测试执行失败。

在关键字库的搜索顺序被设定的情况下就没有必要再使用像 LibraryName.Keyword 长格式的关键字名了。这样就能方便地通过 Set Library Search Order 关键字修改测试库的优先级从而使用不同库中的关键字。这样就避免了因多个库中含有同名关键字而造成的冲突。

关键字库搜索顺序在测试集级别有效这意味着该搜索顺序只有在设定了的库中有效。

2.8.2 超时

关键字可能会在不可遇期的长期执行或不明原因挂起时出现问题。Robot Framework 允许用户为测试用例和用户自定义关键字设定一个超时时间，当一个测试用例或关键在超时时间段内未完成时则强制被停止。通过这种方式来停止关键字可能会使被测系统处于一个不稳定的状态，所以超时机制只有在没有安全选项可选的情况下才推荐使用。总的来说，测试库必须保证关键字不能挂起而且必要的话测试库本身也有超时机

制。

测试用例超时

测试用例超时既可以在测试集表通过超时时间项 *Test Timeout* 设置也可以通过测试用例表中的[Timeout]来设置。测试集表里的测试集超时为每个测试用例设置了一个默认的超时时间，而测试用例表里面的[Timeout]则为该用例设置超时时间并覆盖测试集的默认超时时间。

不论超时时间在哪定义，紧跟表名的第一个单元格里放的总是最大超时时间。最大超时时间一定要按 Robot Framework 的时间格式来给定，也就是说，直接定义为秒数或者像 1 minute 30 seconds 的格式。但值得注意的是，有一些 framework 本身作的限制，像超时时间小于 1 秒钟就是不推荐的。

当测试超时发生时会弹出“测试超时<时间值>到期”的默认错误提示信息。当然，也可以使用自定义超时错误提示信息，自定义的超时提示信息紧跟最大超时时间定义的单元格后面。提示信息像描述文档一样也可以分成多个单元格来写，而且超时时间和超时提示信息里面都可以包含变量。

如果设置了一个超时时间，在超时到期的时刻关键字执行会停止，测试用例会失败。但是， 即使超时发生的情况下，像 test teardown 的关键字也不会被中断，因为它们肩负着最后的清理环境的重任。如果有必要的话，用 user keyword timeouts 同样可以中止这些关键字的执行。

测试超时举例

Setting	Value	Value	Value
Test Timeout	2 minutes		

Test Case	Action	Argument	Argument	Argument
Default Timeout	[Documentation]	Timeout from the Setting table is used		
	Some Keyword	argument		
Override	[Documentation]	Override default, use 10 seconds timeout		
	[Timeout]	10		
	Some Keyword	argument		
Custom Message	[Documentation]	Override default and use custom message		

Test Case	Action	Argument	Argument	Argument
	[Timeout]	1min 10s	This is my custom error.	It continues here.
	Some Keyword	argument		
Variables	[Documentation]	It is possible to use variables too		
	[Timeout]	${TIMEOUT}		
	Some Keyword	argument		
No Timeout	[Documentation]	Empty timeout means no timeout even when	Test Timeout has been used	
	[Timeout]			
	Some Keyword	argument		

用户自定义关键字超时

用户自定义关键字超时可以在关键字表的[Timeout]表项设置。 这种语法设置用 test case timeouts 标识且包括超时时间值及可能的超时提示信息。如果没有设置超时提示信息则使用“关键字超时<时间值>到期”的默认提示信息。

用户关键字超时举例

Keyword	Action	Argument	Argument
Timed Keyword	[Documentation]	Set only the timeout value	and not the custom message.
	[Timeout]	1 minute 42 seconds	
	Do Something		
	Do Something Else		
Timed-out Wrapper	[Arguments]	@{args}	
	[Documentation]	This keyword is a wrapper	that adds a timeout to another keyword.
	[Timeout]	2 minutes	Original Keyword didn't finish in 2 minutes

在用户自定义关键字执行期间，如果应用了关键字超时机制，，当执行时间超过已设置的超时时间时，该关键字的执行就会终止。用户自定义关键字的超时机制同样可用于测试用例的 teardown 中，即使测试用例的

超时没到而自定义关键字的超时已到，关键字同样停止执行，测试用例失败。

如果测试用例和它内部的关键字都设置了超时时间值，那么实际用到的超时时间值是两者当中最小的一个。

> 警告
>
> 如果使用 Python 解释器的话，超时机制可能会使测试用例执行得比正常状态下慢些。慢的原因是由 Python 的线程实现机制引起的，详见 issue 497。

2.8.3 循环

重复执行几个相同的操作在自动化测试是很常见的。对于 Robot Framework 的测试库来说可能有各种各样的循环结构,这些循环结构大多数都是由库内部本身实现的。同时 Robot Framework 也提供了自己的 For 循环语法，这个语法非常有用，例如当需要重复执行来自不同测试库里的多个关键字时。

For 在测试用例和用户自定义关键字中都可以使用。除非是特别简单的测试用例，For 循环用在自定义关键字里要好一些，因为这些关键字隐藏了 For 循环语法的复杂性。最基本的 For 循环语法是 FOR item IN sequence,是从 Python 中提取出来的,但语法规则与 shell 脚本和 perl 非常类似。

普通 for 循环

在普通 for 循环中，每次循环被分配一系列值中的一个。循环语法从:FOR 开始，冒号用于 For 循环与普通关键字区分开来。接下来的单元格内放置循环变量，再接的单元格必须含有 IN,最后两个单元格则指定 For 循环的起点及结束点。

用于 For 循环里的关键字必须从下一行起，并且要向右缩进一个单元格。当关键字或其他元素不再缩进时说明 For 循环结束。并不支持直接的 For 循环嵌套,但我们可以通过把含有 For 循环的关键字再放置在另一个 For 循环里来实现 For 循环的嵌套。

普通 For 循环举例

Test Case	Action	Argument	Argument	Argument	Arguments
Example 1	:FOR	${animal}	IN	cat	dog
		Log	${animal}		
		Log	2nd keyword		
	Log	Outside loop			

Test Case	Action	Argument	Argument	Argument	Arguments
Example 2	:FOR	${var}	IN	one	two
	...	three	four	five	six
	...	seven			
		Log	${var}		

Example 1 中的 For 循环将要执行两次，因为循环变量${animal}可取 cat 和 dog 两个值。循环体包含两个 Log 关键字。循环中包含了 2 个 Log 关键字。第二个例子中循环序列被分成了很多行，循环体一共被执行了 7 次。

当用到列表变量时 For 循环将会更有用且更清晰。这种情况可用下面的事例来演示，例中@{ELEMENTS}去掉包含任意长度的${element}类型的元素，而关键字 Start Element 则对每个元素进行操作。

使用列表变量的 For 循环

Test Case	Action	Argument	Argument	Argument	Arguments
Example	:FOR	${element}	IN	@{ELEMENTS}	
		Start Element	${element}		

使用多个循环变量

在 For 循环中同样可以使用多个循环变量。这种形式的 For 循环与普通 For 循环语法上是类似的，强调的是所有循环变量必须放在:FOR 和 IN 之间。这里的变量数是任意的，但 IN 后面变量对应的值的个数必须是变量个数的倍数。

这种语法格式在有无列表变量时都起作用。在存在列表变量的情况下，常用于给定循环值少于循环变量的情况。实例如下：

多循环变量的 For 循环

Test Case	Action	Argument	Argument	Argument	Arguments
Example	:FOR	${index}	${english}	${finnish}	IN
	...	1	cat	kissa	
	...	2	dog	koira	
	...	3	horse	hevonen	
		Do X	${english}		
		Y Should Be	${finnish}	${index}	

Test Case	Action	Argument	Argument	Argument	Arguments
	:FOR	${name}	${id}	IN	@{EMPLOYERS}
		Create	${name}	${id}	

For in range 循环

早的 For 循环在一个序列上迭代执行，这也是 For 循环最常用的情况。很多场景下 For 循环执行一定数目的次数，Robot Framework 用 IN RANGE 限制语法来达到这个目的。这种语法格式是从 Python 中继承而来的。

和 For 循环很类似，For in range 同样以:FOR 开头，循环变量放在紧跟其后的单元格中。这种语法格式的循环只允许有一个循环变量，它包含当前循环索引。循环变量后面的单元格必须包含 IN RANGE 并且随后的单元格跟循环范围数。

下表中第一个是个最简单的例子，它只指定了循环变量的上限（10），实际上下限默认从 0 开始去掉，并且步长是 1。在 For 循环中也可以同时指定上限和下限，步长同样是 1。最后要指出的是，步长同样也可以自定义，但是如果步长是负的话，循环将递减进行。所有可能的情况详见下表的实例。

For in range 例子

Test Case	Action	Argument	Argument	Arg	Arg	Arg
Only upper limit	[Documentation]	Loops over	values	from 0	to 9	
	:FOR	${index}	IN RANGE	10		
		Log	${index}			
Start and end	[Documentation]	Loops over	values	from 1	to 10	
	:FOR	${index}	IN RANGE	1	11	
		Log	${index}			
Also step given	[Documentation]	Loops over	values	5, 15,	and 25	
	:FOR	${index}	IN RANGE	5	26	10
		Log	${index}			
Negative step	[Documentation]	Loops over	values	13, 3,	and -7	
	:FOR	${index}	IN RANGE	13	-13	-10
		Log	${index}			

退出 For 循环

正常情况下 For 循环只有在所有循环数据都完成或该测试用例或关键字失败的条件下才退出。但是有些情况下 For 循环在所有循环没有完成的情况退出， BuiltIn 关键字 Exit For Loop 可以实现这种功能。

Exit For Loop 关键字可以直接用在一个 For 循环里或 For 循环里面的关键字里。 这两种情况下测试执行在退出 for 循环后继续，但是如果关键字在 for 循环外面执行，则测试用例失败。

退出 for 循环举例

Test Case	Action	Argument	Argument	Argument
Exit	:FOR	${var}	IN	@{SOME LIST}
		Run Keyword If	'${var}' == 'EXIT'	Exit For Loop
		Do Something	${var}	

退出 for 循环也可以通过一个抛出 ROBOT_EXIT_FOR_LOOP 属性异常的关键字来发起。详见 停止测试执行中的举例说明如何在 Python 和 Java 实现停止 for 循环。

> 小贴士
>
> 退出 For 循环功能在 Robot Framework 2.5.2 版本中新引入。

重复执行单个关键字

当仅仅需要重复执行单个关键字时使用 For 循环显得大材小用。在这种用例中可以使用在 2.0.4 版刚引入的 BuiltIn 的关键字 *Repeat Keyword* 来处理。这个关键字后跟要执行的关键字及执行次数作为它的两个参数。这个次数参数有 x 或 times 作为可选后缀以增强它的可读性。

重复关键字举例

Test Case	Action	Argument	Argument	Argument	Argument
Example	Repeat Keyword	5	Some Keyword	arg1	arg2
	Repeat Keyword	42 times	My Keyword		
	Repeat Keyword	${var}	Another KW	argument	

Robot Framework 同样有个专用语法来重复执行单个关键字。在 2.04 版本中不赞同用这个语法取代 Repeat Keyword 关键字，并在 2.5 版本中被移除。

2.8.4 条件执行

通常来说，为了便于理解和掌握，不提倡测试用例甚至是关键字里面出现条件逻辑。相反，这种条件逻辑应该以自然编程语言的形式出现在测试库里。但是，有些条件逻辑在一些场合将会非常有用，尽管 Robot Framework 没有提供类似 if/else 结构的条件逻辑，但可以通过以下几种方式实现相同的功能：

- 用于测试用例或测试集 Setup 或 Teardown 的关键字名称可以用一个变量来指定，这样可以方便地改变关键字名称，例如在命令里改变它。
- BuiltIn keyword Run Keyword 可以将一个关键字作为 Run 参数的形式来执行，这个关键字其实就是 Run 的一个变量。这个变量的值（也就是某个关键字）可以动态地从前一个关键字中获得或是由命令行给出。
- BuiltIn keywords Run Keyword If and Run Keyword Unless 当该关键字的表达式分别为逻辑真和假时，跟在其后的关键字将被执行，它们可以很好地用于创建类似 if/else 逻辑结构表达式。参见前面文档的例子。
- BuiltIn keyword, Set Variable If 可以根据给定条件的逻辑真或假来设置某个变量的值。
- 还有几个 BuiltIn keywords 允许用户根据某个测试用例或测试集的失败或成功来执行某个命名关键字。

2.8.5 多个关键字的并行执行

早于 2.5 版本的 Robot Framework 提供了一种专用语法来并行执行多个关键字。这个功能后来因为很少用到并且效果不理想而被取消掉。

当需要并行执行的时候，为了能够让测试库在后台执行代码，并行机制必须在测试库级别上实现。不失一般性这实际上意味着这个库需要一个关键字来完成像开始某事件一样来开始执行并立即返回，另一个关键字则等到某种条件发生时返回某个结果。例如 OperatingSystem library 库中 Start Process 关键字和 Read Process Output 关键字。

3 执行测试用例

3.1 基本应用

RF 测试用例可从命令行执行，默认的结果为 XML 格式的 output 和 HTML 格式的 report 和 log。执行完之后，使用 Rebot 可以将多个输出文件进行合并，也可以进行其他的后续处理。

3.1.1 启动测试用例执行

大纲

```
pybot [options] datasources
jybot [options] datasources
interpreter /path/to/robot/runner.py [options] datasources
java -jar robotframework-<version>.jar [options] datasources
```

差异性的运行脚本

我们通常可以在命令行使用 pybot 或 jybot 命令来启动测试用例，在一定程度上来说，两个命令是一样的，但是，pybot 依靠 Python 解释器来执行测试用例，jybot 则使用 Jython。使用哪一个命令来启动测试用例取决于您使用的测试库。一些库使用 Python 的模块，或该库的语法在 Python 环境下可用，而另一些库使用以 JAVA 为基础的工具，需要使用 Jython，而有的库既依赖 Python 又依赖 Jython。您既可以使用 Python 也可以使用 Jython，我们推荐您使用 Python，因为 Python 的运行速度要快于 Jython。

从 RF2.5.2 版本开始，RF 可作为一个独立的 JAR 使用。Jython 内置在 JAR 中。通常 RF 的参数将跟在 JAR 文件后。

另一个启动测试用例执行的方式是直接运行 robot 模块中的 runner.py 脚本。这个方法可以自由的选择解释器和设置命令行参数。最普遍的方式是改变 JVM maximum memory .

无论哪种启动方式，都要设置测试数据的路径。另外，不同的命令行参数可以改变测试用例的执行和输出文件的产生。

指定测试数据执行

RF 的测试用例可以创建在文件或目录中，在运行脚本后接需要执行用例的名文件名或路径名。路径可以是绝对路径，更普遍的是相对路径。文件或目录构成顶级测试集，该测试集如果没有被--name option 所覆盖，它将使用文件或目录名作为自己的名字。不同的执行情况将在下面的例子中说明。本章节中只有 pybot 被用到，但 jybot 脚本的用法类似。

pybot test_cases.html

pybot path/to/my_tests/

pybot c:\robot\tests.txt

我们可以通过用空格分割来一次给出多个文件的路径，来运行多个文件或目录下的文件。RF 可以自动的产生顶级的测试集，具体的文件或目录将变成它的子测试集。产生的测试集的名称将由子测试集的名字和“&”组成。例如下面的第一个例子中，顶级的测试集名称为 My Tests & Your Tests。但是，有时自动产生的测试集的名字将很长很复杂。通常，更好的做法是使用—name option 来覆盖名称，正如第二个例子所示：

pybot my_tests.html your_tests.html

pybot --name Example path/to/tests/pattern_*.html

3.1.2 命令行参数

RF 提供了大量的命令行参数来控制测试用例的执行和输出文件的产生。这一章节将介绍参数的语法，和哪些参数是默认存在的。它是如何使用的将在其他章节介绍。

使用参数

当使用参数时，参数必须位于运行脚本和源数据中间。例如：

pybot -L debug my_tests.txt

jybot --include smoke --variable HOST:10.0.0.42 path/to/tests/

长短参数

参数总有一个长名字，例如“--name”，更普遍的需要一个短名字，例如“-N”，另外，只要长参数是唯一的，长参数就可以被缩短。例如：“--logle DEBUG”能起作用，但“--lo log.html”就不能起作用，原因是前者只能匹配“--loglevel”后者可以匹配多个参数。当手动的执行用例时，短参数和缩短后的长参数是实用的，但由于长参数容易理解，所以长参数推荐在启动脚本中实用。

长参数格式上是不区分大小写的，可以用一种便利的方式来书写。例如“-SuiteStatLevel”和“--suitestatlevel”是一样的，但比“--suitestatlevel”容易阅读。

设置参数值

大部分参数需要一个参数值，位于参数名后面。长参数和短参数都接受由空格分开的参数值，例如：“--include tag”或“-i tag”。使用长参数时，可以将等号“=”作为分隔符，例如“--include=tag”，使用短参数时，分隔符可以省略，例如“-itag”。

某些参数可以被具体赋值多次。例如，“--variable VAR1:value --variable VAR2:another”设置了两个变量。如果某个只有一个值的参数被赋值多次，则最后一次生效。

简单正则表达式

很多参数后可接正则表达式的参数值。“*”和“？”可替代具体的字符，“*”可匹配任何包括空值在内的字符串，“？”可以匹配任何单个字符。例如：“--include prefix-*”匹配所有以“prefix-”开头的 tag，“--include a???” 匹配所有以“a”开头 4 个字符的 tag。

3.1.3 测试结果

命令行输出

大部分在测试用例执行过程中可以看到的输出都显示在命令行中。所有执行的测试集、测试用例以及状态都在命令行中实时的显示。下面的例子显示了在执行一个具有两个用例的测试集的过程中，命令行中显示的输出。

```
==============================================================Example test suite
==============================================================First test :: Possible test
documentation                                | PASS |
--------------------------------------------------------------Second test
| FAIL |
Error message is displayed here
==============================================================Example test suite
| FAIL |
2 critical tests, 1 passed, 1 failed
2 tests total, 1 passed, 1 failed
==============================================================Output:
/path/to/output.xml
```

```
Report:  /path/to/report.html
Log:     /path/to/log.html
```

输出文件合并

命令行中显示的输出结果是非常有限的，分开的输出文件通常需要被用来查看测试结果。在上面的例子中可以看到，默认会产生上述的三个文件。第一个是 XML 格式的文件，它包含了用例执行过程中全部的信息。第二个是一个更高级别的报告，第三个是更多细节的日志文件。这些文件以及其它可能产生的文件将在“差异化输出”章节中具体讨论。

返回码

运行脚本通过返回码使所有测试用例的状态和系统交互。如果所有的用例执行启动成功而且没有严重的失败，返回码将是 0，所有可能的返回码都列在下表中。

返回码

RC	Explanation
0	All critical tests passed.
1-249	Returned number of critical tests failed.
250	250 or more critical failures.
251	Help or version information printed.
252	Invalid test data or command line options.
253	Test execution stopped by user.
255	Unexpected internal error.

用例执行完之后，很容易通过返回码来确定所有用例执行的状态。例如，在 bash shell 中，返回码保存在具体的变量“ $?”中，在 Windows 系统中，返回码保存在变量% ERRORLEVEL %中。如果您使用外部的工具来运行用例，请查询相关的文档来了解如何获得返回值。

执行中的错误和警告

在用例执行过程中，可能会遇到一些意想不到的问题，例如不能导入库或资源文件或者是关键字已经不可用。取决于问题的严重程度，问题将被分类为错误或警告，通过标准的错误流它们将被写入控制台，它也

会在日志文件中单独的执行错误部分显示，同时，它也会写到 RF 的系统日志中。通常这些错误由 RF 的核心代码产生，但可以通过使用库的 log level WARN 来记录警告。下面的例子显示了在日志文件中错误和警告部分的样式。

```
20090322 19:58:42.528          ERROR  Error in file
                                      '/home/robot/tests.html' in table
                                      'Setting' in element on row 2:
                                      Resource file 'resource.html' does
                                      not exist

20090322 19:58:43.931          WARN   Keyword 'SomeLibrary.Example
                                      Keyword' is deprecated. Use keyword
                                      `Other Keyword` instead.
```

3.1.4 转义复杂字符

因为空格被用于分隔参数，所以使用空格在参数之中就会有问题。有些参数，例如，“--name”会自动转换下划线为空格，但是其他空格就必须要转意。还有，一些字符在命令行中使用会过于复杂。因为转意复杂字符为带有反斜线或引号的值并不是总奏效，所以 RF 拥有一套自己转意机制。另外的方式是使用参数文件，其中选项可以使用无格式文本文件的方式具体指定。当用例执行过程中或输出文件后处理时，这些机制将发挥作用，同时，一些外部工具同样具有类似的能力。

在 RF 的命令行转义机制中，问题字符被灵活的转意为其他字符。在命令行中需要使用到参数—escape（短参数为 -E），后接参数 what：with，这里 what 是指需要转意的字符名称，with 是转意后的字符串。下表中列出了可以转意的字符。

转意字符

Character	Name to use	Character	Name to use
&	amp	(	paren1
'	apos	)	paren2
@	at	%	percent
\	blash	|	pipe
:	colon	?	quest

Character	Name to use	Character	Name to use
,	comma	"	quot
{	curly1	;	semic
}	curly2	/	slash
$	dollar		space
!	exclam	[	square1
>	gt	]	square2
#	hash	*	star
<	lt		

下面的例子中使该语法更清晰。第一个例子中，源数据 X 将获得值为“Value with spaces”，第二个例子中，变量 ${VAR}设置为"Hello, world!"：

--escape space:_ --metadata X:Value_with_spaces

-E space:SP -E quot:QU -E comma:CO -E exclam:EX -v VAR:QUHelloCOSPworldEXQU

注意，所有命令行参数，包括路径，都是转意的，转意字符在使用时要仔细。

3.1.5 参数文件

Problematic characters can often be handled easily using argument files。这些文件可以包含命令行参数和测试数据路径，也可以逐行的包含测试数据。参数文件可以在“--argumentfile”（短参数为”-A”）参数后使用。参数文件可以包含任何除空格之外的字符，字符串首尾的空格将被忽略，另外，以字符“#”开始的行和空行将被忽略。

--doc This is an example (where "special characters" are ok!)

--metadata X:Value with spaces

--variable VAR:Hello, world!

This is a comment

path/to/my/tests

小贴士：

RF 从 2.5 版本开始，参数文件中可以包含非 ASCII 的字符。参数文件必须按照 UTF-8 的编码方式保存。

另外一个参数文件重要的应用是指明输入文件和目录的顺序。当按字母顺序不合适的时候，这种方式就显

得很有用。

--name My Example Tests

tests/some_tests.html

tests/second.html

tests/more/tests.html

tests/more/another.html

tests/even_more_tests.html

当参数文件在命令行中使用时，文件中的内容将会在命令行中列出来。参数文件既可以单独使用（内容中包含参数和测试数据的路径），又可以和其他的参数和路径一起使用。可以使用“—argumentfile +参数”多次，或者递归使用。

pybot --argumentfile all_arguments.txt

pybot --name example --argumentfile other_options_and_paths.txt

pybot --argumentfile default_options.txt --name example my_tests.html

pybot -A first.txt -A second.txt -A third.txt some_tests.tsv

3.1.6 获得帮助和版本信息

我们可以使用 pybot 或 jybot 来启动测试用例，使用 rebot 来后处理测试报告，可以通过—help 或-h 来获得帮助，各个参数将会有一个简明的解释。

所有的启动脚本都支持通过—version 来查看版本信息。信息中同时包含 Python 或 Jython 的版本信息。

$ pybot --version

Robot Framework 2.0.3 (Python 2.5.1 on cygwin)

C:\>jybot --version

Robot Framework 2.0.3 (Jython 2.2 on java1.5.0_16)

C:\>rebot --version

Rebot 2.0.3 (Python 2.5.2 on win32)

3.1.7 编写启动脚本

在一个集成的系统中，测试用例往往需要自动执行，在这种情况下，需要有一个启动脚本来启动测试用例和处理后续输出的文件。类似的脚本在手工启动时是有效的，特别是当大量的命令行参数需要设置，或者

搭建测试环境很复杂时。

在类 UNIX 环境中，Shell 脚本提供了一个简单但有效的机制来定制启动脚本。Windows 的批处理文件是有效的，但有更多的限制而且很复杂。一个和平台无关的方法是使用 Python 或其他高级别的编程语言来编写启动脚本。无论选择什么语言，我们都推荐使用长参数，因为它更好理解。

在第一个例子中，使用不同的浏览器做相同的 WEB 测试，并合并测试结果。使用 shell 脚本实现该功能就非常简单，只需要把需要使用到的命令逐行列出来。

```
#!/bin/bash
pybot --variable BROWSER:Firefox --name Firefox --log none --report none --output out/fx.xml login
pybot --variable BROWSER:IE --name IE --log none --report none --output out/ie.xml login
rebot --name Login --splitoutputs 1 --outputdir out --output login.xml out/fx.xml out/ie.xml
```

在 Windows 中使用批处理文件来实现上述功能也不是很复杂，有一点很重要，pybot 和 rebot 是由批处理文件实现的，当运行其他批处理文件时，必须要使用 call。当第一个批处理文件结束时，测试也将结束。

```
@echo off
call pybot --variable BROWSER:Firefox --name Firefox --log none --report none --output out\fx.xml login
call pybot --variable BROWSER:IE --name IE --log none --report none --output out\ie.xml login
call rebot --name Login --splitoutputs 1 --outputdir out --output login.xml out\fx.xml out\ie.xml
```

在下一个例子中，在测试用例执行前，将库路径下的 jar 文件设置到 CLASSPATH 路径中。在这个例子中，启动脚本需要测试数据的路径。我们同样可以自由的使用命令行参数，即使有些参数已经设置过。All this is relatively straight-forward using bash:

```
#!/bin/bash
cp=.
for jar in lib/*.jar; do
    cp=$cp:$jar
done
export CLASSPATH=$cp
jybot --ouputdir /tmp/logs --splitoutputs 2 $*
```

上述功能在 Windows 中实现有些复杂。困难的部分是在 for 循环中设置存放 jar 的变量，你将用到下面的函数。

```
@echo off
set CP=.
for %%jar in (lib\*.jar) do (
    call :set_cp %%jar
)
set CLASSPATH=%CP%
jybot --ouputdir c:\temp\logs --splitoutputs 2 %*
goto :eof

:: Helper for setting variables inside a for loop
:set_cp
    set CP=%CP%;%1
goto :eof
```

修改 JAVA 启动参数

有时，我们需要改变 JAVA 的启动参数。由于默认的大小不足以产生大的输出文件，更普遍的方式是增加 JVM 最大内存的大小，如何设置 JAVA 参数取决于 Jython 的版本。

在 Jython2.2 中，可以通过增加参数值来修改 Jython 启动脚本（jython shell 脚本或 jython.bat 文件），例如，-Xmx1024m 设置最大内存为 1024M。在 Windows 中，最终的命令行参数为：

"C:\Java\jre1.6.0\bin\java.exe" -Xmx1024m -Dpython.home="C:\Jython22" -classpath "C:\Jython22\jython.jar;%CLASSPATH%" org.python.util.jython %ARGS%

在 Jython2.5 中，设置参数将变得更容易，只需要使用-J 来设置 Jython 解释器。这意味着可以直接设置 Jybot 启动脚本，而不用设置 Jython 脚本。如下面例子所示：

```
%jython% -J-Xmx1024m %runner% %*
```

3.1.8 调试

在被测系统不正确工作的情况下，测试用例将会运行失败，可能是被测系统的问题或是自动化测试脚本的问题。在命令行和报告文件中的错误信息部分将会显示失败的原因，有时可以通过错误信息就足够查明问题的原因。使用更频繁的是查看日志文件来查明具体是哪个关键字失败。

当由于被测系统原因导致用例失败时，错误信息和日志信息应该足够查明失败的原因。如果这并不能发现原因，测试库并不能提供足够的信息，这是就需要增强它。这种情况下，运行相同的手工测试，这将可能发现导致该现象的更多信息。

失败往往由测试用例本身导致或是使用了很难调试的关键字。如果错误信息提示关键字的参数数量不正确，这样的问题很显而易见，但是如果关键字由于异常的原因失败，这样的根本原因就很难发现。查找更多信息的第一步应该是重新执行日志中错误的部分。例如，一个失败的测试库导入的错误的原因很可能是缺少关键字。

如果在默认的情况下，日志文件不能提供足够的信息，很可能是在一个低的级别执行用例。例如，显示着代码中失败的位置的 tracebacks 将会在调试级别被记录，当失败的原因是使用了无效的关键字时，这些记录的信息，将变得没有任何意义。

如果日志文件还是不能提供足够的信息，那么这时应该使用 RF 的系统日志（syslog）来查看具体的信息。也可以增加关键字来查看具体原因。特别是 BuiltIn 库中 Log 和 Log Variables 是有用的。如果没有别的可以提供帮助，可以给邮件列表中的地址发邮件来获得帮助。

3.2 测试用例执行

本章描述了由测试数据解析的测试集如何执行，当用例失败后如何继续执行，整个测试集如何优雅的停止。

3.2.1 执行流程

执行测试集和测试用例

测试用例总是在某个测试集中执行。由目录产生的测试集包含由测试用例文件或目录组成的子测试集，由测试用例文件组成的测试集直接包含测试用例。默认情况下，所有的测试用例都会执行，可以通过参数“--test”、“--suite、“--include”和“--exclude”来选择用例执行。测试集中如果没有用例，将不会执行。

RF 从顶级的测试集开始执行。如果测试集中包含用例，则顺序执行。如果测试集中再包含测试集，则按照深度优先的顺序依次执行。当执行单个用例时，关键字按顺序执行。通常用例中的任何一个关键字失败了，用例将停止执行，但是，我们可以做到让失败后继续执行。接下来的章节中将具体介绍准确的执行顺序和 setups 和 teardowns。

setups 和 teardowns

setups 和 teardowns 是在用例和测试集级别定义的。

Suite setup

如果测试集有 setup 关键字，在所有的用例和子测试集执行前将执行 setup 关键字，如果 setup 执行成功，将继续向下运行，如果失败了所有的用例、子测试集也将被标记为失败，具体信息为“Setup of the parent suite failed”。用例及子测试集的 setups 和 teardowns 也不会被执行。

Suite setup 通常被用来设置设置测试环境。由于如果 Suite setup 执行失败，用例将不会执行，可以很轻松的使用 Suite setup 来保证环境是满足测试条件的。

Suite teardown

如果有测试集的 teardown，在所有的用例和子测试集执行后将执行 teardown，不管用例的运行状况如何以及对应的 suite setup 是否执行成功，Suite teardown 都会执行，如果 Suite teardown 执行失败，所有用例和子测试集将标注了失败。

Suite teardown 通常被用来清理测试环境。即使 teardown 中的一些关键字执行失败，RF 也能保证 teardown 中的关键字都会被执行到，确保 Suite teardown 中的所有任务可以做完。

Test setup

在用例中的关键字执行前将执行 Test setup，如果执行失败，用例中的关键字将不会执行，Test setup 的主要作用是为具体的用例设置运行环境。

Test teardown

在用例执行后将执行 Test teardown。不管用例的运行状况如何以及对应的 test setup 是否执行成功，Test teardown 都会执行。这类似于 Suite teardown，Test teardown 的主要作用是清理环境。即使中间某个关键字执行失败，Test teardown 也会充分的执行完。

执行顺序

测试集中的用例按照用例文件中定义的顺序执行。测试集在高一级的测试集中按照名字在字母表的顺序执行。如果在命令行中给出多个文件或目录，将按顺序执行。

如果要确定目录中测试集的执行顺序，可以增加类似于 01、02 的前缀到文件名或路径名上。如果测试集的名字由两个下划线分开，这样的前缀将不会包含在产生的测试集名称中。例如：

01__my_suite.html -> My Suite

02__another_suite.html -> Another Suite

如果测试集中的测试集按照字母表的顺序执行存在问题，好的办法是对它们分别按照需要的顺序来执行。这将导致命令行参数过长，但是参数文件允许逐行列出参数。

可以使用参数“--runmode”来让用例按照随机的顺序运行。

3.2.2 失败继续执行

通常任何一个关键字执行失败了，整个测试用例将停止执行。这种行为缩短了测试执行时间，阻止后续的关键字被挂起，或者其他由被测程序本身状态不正确引发的问题出现。这或许是个缺点，因为后续的关键字也许会提供更多的被测系统的信息。

在 RF2.5 之前，在关键字执行失败后，只有使用 BuiltIn 库中的关键字 Run Keyword And Ignore Error 和 Run Keyword And Expect Error，测试执行才不会立即终止。使用这些关键字增加了用例的复杂性。在 RF2.5 中，增加了下面的特性，将使在关键字失败的情况下，用例可继续执行。

由关键字产生的特别的失败

库关键字使用异常的方式来上报失败，但它也可以使用具体的异常来通知 RF 核心框架当关键字执行失败的时候，测试执行继续进行。在测试库 API 章节，异常的产生方式将有介绍。

当关键字中有一个或多个执行失败，用例执行完后，整个测试用例将被标记失败，如果存在一个以上的失败，所有的失败信息都将会列举在最终失败信息中：

Several failures occurred:

1) First error message.

2) Second error message ...

在一个可继续的失败后，如果还有关键字执行失败，那么测试执行同样会停止。所有的失败信息也会列举在最终失败信息中。失败的关键字中可以得到返回值，将它付给变量后，是一个 Python 的空（None）。

关键字 Run Keyword And Continue On Failure keyword

BuiltIn 库中的关键字 Run Keyword And Continue On Failure 可以将任何失败都转换为可继续执行的失败。这是由框架本身的关键字所保证的。

在 teardowns 中，默认设置为可继续执行的失败

为了保证清理环境的操作能够执行到，所有用例和测试集的 teardown 中，都被设置为可继续执行的失败。这意味着，teardown 中的所有关键字都会被执行到。

用例使用模板时，所有顶级的关键字都会被执行

在使用测试模板时，为了确保所有的组合都要被测试到，数据行中的数据将会全部被执行。在这一应用中，可继续执行的失败将被限制在顶级的关键字中，如果内部的非可继续执行的失败产生，测试执行将停止。

3.2.3 停止测试执行

有时，我们需要在全部用例执行结束前停止在执行的用例，但日志和报告文件将会产生，剩余的用例将标记失败。不同的结束方式将在下面介绍。

小贴士：

大部分的特性是在 RF2.5 中新增。只有 ExitOnFailure 是以前版本提供的

使用键盘按键 Ctrl 和 C

当用例在执行过程中，在命令行就节面中使用键盘按键 Ctrl 和 C，自动化测试将停止执行。Python 环境下，执行将立即停止，在 Jython 环境下将会执行完当前的关键字后再停止。

如果再次按键盘按键 Ctrl 和 C，测试将立即停止，日志和报告也将不会产生。

使用信号

在类 UNIX 的机器上，停止测试执行可以使用信号量 INT 和 TERM,这些信号量可以在命令行通过 kill 命令来使用，信号是自动发送的。

信号量在 Jython 的环境中和按键盘按键 Ctrl 和 C 具有同样的限制。第二次发送信号量将会强制停止测试执行。

使用关键字

测试执行也可以被关键字来终止。BuiltIn 库中的关键字 Fatal Error 可以终止整个测试的执行，常规的关键

字失败时将使用 fatal exceptions。

ExitOnFailure

如果—runmode 参数后接值 ExitOnFailure (不区分大小写)，这时，如果有一个严重级别的测试失败，则整个测试的执行将停止，其余没有执行到的用例将标记为失败。

处理 Teardowns

默认情况下，在上述几种方法使测试执行停止前，用例和测试集的 teardown 方法都会执行到。无论测试停止与否，清理环境都是必要的。

从 RF2.5.2 开始，如果—runmode 参数后接值 SkipTeardownOnExit，测试执行停止时，teardown 方法将被跳过。如果清理环境比较耗时，这个方法就显的很有用了。

3.3 后处理输出文件

在测试执行过程中产生的 XML 格式的输出文件，在测试结束后，可以通过 rebot 工具进行后处理，rebot 是 RF 的一个组成部分。测试报告和日志在测试过程中自动生成，但还是有很多场景需要在测试执行完后再单独的产生报告。

3.3.1 使用 rebot 工具

Rebot 的基本语法是和启动测试完全一样的，返回码也是一样的，大部分命令行参数也是一致的。主要的差别在于 rebot 的源数据是 XML 格式的的输出文件而不是数据文件或目录。

3.3.2 产生不同的报告和日志文件

在测试执行过程中，robot 会自动产生相同报告和日志文件。但是，这并不能精确地相同，例如，它可能具有一个包含所有用例的报告文件或是包含测试子集的报告文件。另一个应用是，测试执行的过程中只产生 XML 格式的输出文件，报告文件和日志文件将通过设置命令行参数—log NONE 和--report NONE 而不起作用。测试用例可以在不同的环境执行，将所有的 XML 格式的输出文件收集在一起，然后产生一个整体的报告和日志文件。在 Jython 环境中产生报告和日志文件将花费较多的时间，最好是在用例执行完之后使用 rebot 工具产生报告和日志文件，因为 rebot 是运行在 Python 环境中，它运行更快。例如：

rebot output.xml

rebot path/to/output_file.xml

rebot --include smoke --name Smoke_Tests c:\results\output.xml

当使用 RF 的 jar 文件时，rebot 可以在命令行启动，如下：

java -jar robotframework-<version>.jar rebot

当使用 jar 的方式使用 rebot 时，所有的命令行参数都可以使用。

小贴士：

在 Jython 环境中运行用例时，如果报告和日志文件比较大，Jython 默认的最大 JVM 内存是不够用的。你应该增加最大 JVM 内存，或者是不产生报告和日志文件，在用例执行完之后，使用 rebot 产生。

3.3.3 合并输出文件

Rebot 的一个重要作用是它可以合并在不同环境下产生的 XML 格式的输出文件。你可以在不同的环境中运行用例，再产生一个整体的报告文件。合并输出文件非常简单，你只需要列出需要合并的文件名字。例如：

rebot output1.xml output2.xml

rebot outputs/*.xml

当输出文件合并后，它将产生一个最高级别的测试集，原来的测试集将作为它的子测试集。它和执行多个测试文件或测试集的原理相同，这样产生的最高等级的测试集的名称由所有子测试集的名称和&和空格组成。自动产生的名字很长并不理想，幸好我们可以使用--name 来赋予它一个有意义的名字，例如：

rebot --name Browser_Compatibility firefox.xml opera.xml safari.xml ie.xml

rebot --include smoke --name Smoke_Tests c:\results*.xml

3.4 配置测试用例执行

这一节将介绍如何在命令行设置测试的执行参数，和后处理输出文件。关于如何设置产生输出文件的参数将在下一章节中介绍。

3.4.1 选择测试用例

RF 提供了一组参数可以选择用例来执行。相同的参数在使用 rebot 工具产生输出文件时同样有效。

通过测试集和用例名来选用例

在命令行中可以分别使用参数--suite (-s)和--test(-t) 通过测试集和用例名称来选择用例或测试集来执行。这些参数可以使用多次来选择用例多次。这些参数都是不区分大小写和空格的，所以可以使用通配符的匹配方式来选择多个用例。如果参数--suite 和—test 都使用了，只有指定测试集中的指定用例将被选中。例如：

--test Example

--test mytest --test yourtest

--test example*

--suite example-??

--suite mysuite --test mytest --test your*

使用参数--suite 和直接选中某个用例文件或目录来执行类似。这样做主要的好处是可以基于父测试集来选择用例。指明具体的测试集时要在父测试集和子测试集中间加一个点“.”。这种情况下父测试集的 setup 和 teardown 方法会执行。

--suite parent.child

--suite myhouse.myhousemusic --test jack*

在开发用例时，选择单个用例执行可以使用参数--test，但执行多个用例时会比较受限制。在选择多个用例的情况下，使用—suite 比较合适，但通常情况下使用标签(tag)来选择用例会更加灵活。

使用标签选择用例

可以分别使用参数--include (-i)和—exclude(-e)来包含或排除一些用例。当前者使用时，只有 tag 名称匹配的用例才被选中，使用后者时情况刚好相反。如果两个参数一起使用，会对二者做一个交集。

--include example

--exclude not_ready

--include regression --exclude long_lastin

参数--include(-i)和—exclude(-e)都可以使用多次来匹配 tag 多次，参数中也都可以使用通配符。原则是在--include 选中的用例中排除掉—exclude 选中的用例。还可以选择由多个 tag 共同修饰的用例，多个 tag 间使用&或 AND（区分大小写）。从 RF2.1.3 以后，包含一个确定 tag 以上的用例可以通过 tag 间使用 NOT(区分大小写)来选择。只要有任何一个能匹配 NOT 的 tag，用例将不会被选择。例如：

--include req-*

--include regressionANDiter-42

--include tag1&tag2&tag3&tag4

--exclude regressionNOTowner-*

--include tag1NOTtag2NOTtag3&tag4 (includes tests which have `tag1`, but not tests which additionally have `tag2` or both tags `tag3` and `t

通过 tag 选用例具有非常灵活的机制，它将有下面几个有趣的可能：

- 测试用例的子集打上 tag——smoke，这样使用--incloud 就可以是该子集中的用例先执行。
- 未完成的用例仍然可以提交到版本控制库中，你只要为未完成的用例打上 tag——not_ready，执行用例时使用--exclude not_ready。
- 可以为用例打上 tag，iter-<数字>，当用例执行完之后，一个包含特定迭代次数的报告将会产生。例如，rebot --include iter-42 ouput.xml。

3.4.2 设置关键级别

最终的测试结果取决于关键级别的测试。如果一个关键测试失败了，则整个测试用例被认为失败。另一方面，非关键测试失败了，整个测试的状态仍然可以通过。

默认情况下，所有的测试用例都是关键级别的，但这可以通过参数--critical (-c)和 --noncritical (-n)来修改。这些参数通过 tag 来设置哪些用例是关键级别的，类似于--include 和--exclude 来选择用例。如果只使用了--critical 来设置，那么匹配的用例是关键级别的。如果只使用了--noncritical 来设置，那么不匹配的用例是关键级别的。如果两个参数都使用了，二者交集的用例被认为是关键级别的。这两个参数都可以通过通配符来设置，也可以设置多次来设置多次。

--critical regression

--noncritical not_ready

--critical iter-* --critical req-* --noncritical req-6??

最常见的使用方式是，用例还没开发完成或被测系统还没有开发完成，这部分用例还不能执行，你应该使用--exclude 来排除这部分用例而不是把它设置为非重要级别。enables you to see when they start to pass.

从 RF2.1 开始，当用例执行过程中，用例关键级别的设置并不总是有效。在使用 robot 后处理输出文件时，如果你想保持相同的关键级别，你需要使用--critical (-c)和--noncritical (-n)。

Use rebot to create new log and report from the output created during execution

pybot --critical regression --outputdir all my_tests.html

rebot --name Smoke --include smoke --critical regression --outputdir smoke all/output.xml

No need to use --critical/--noncritical when no log or report is created

jybot --log NONE --report NONE my_tests.html

rebot --critical feature1 output.xml

3.4.3 设置元数据

设置名称

当 RF 解析测试数据时，测试集的名称可以从测试用例文件和目录产生。顶级测试集的名称可以由命令行参数--name (-N)来复写，名称中的下划线将自动转换为空格，字符转换为大写字母。

设置文档

顶级测试集的注释文档可以由命令行参数--doc (-D)来复写，参数中的下划线将自动转换为空格，而且可以包含简单的 HTML 格式的文档。

设置元数据

命令行参数--metadata (-M)可以给所有执行的用例设置元数据。参数必须使用键值对的方式 name：vulue，其中 name 为元数据的名字，value 为元数据的值。参数中的下划线将自动转换为空格，而且可以包含简单的 HTML 格式的文档，这个参数可以多次使用来设置元数据多次。

设置 tag

命令行参数--settag (-G)可以给所有执行的用例设置 tag。这个参数可以多次使用来设置 tag 多次。

3.4.4 设置库的搜索路径

当使用某个测试库时，RF 使用 Python 或 Jython 解释器来导入模块，模块的搜索路径叫“PYTHONPATH”当用例运行在 Jython 环境时，还需要使用到 JAVA 的 CLASSPATH 路径。

为用例的顺利运行，需要修改库的搜索路径。In addition to find test libraries, the search path is also used to find listeners set on the command line.有多种方式可以修改 PYTHONPATH 和 CLASSPATH，无论选择哪种启动方式，我们都推荐使用用户自定义启动脚本。

在 PYTHONPATH 中自动定位

Python 和 Jython 安装程序自动的设置它自己的库路径到 PYTHONPATH 中，这意味着由 Python 封装的测试库会自动的安装到搜索路径中。RF 在自己的目录中包含测试要用到的标准库和执行用例的目录设置到 PYTHONPATH 中的目录。

设置 PYTHONPATH

有很多方法可以修改 PYTHONPATH,但常见的方式还是在执行用例前修改系统的环境变量。Jython 并不使用 PYTHONPATH 的环境变量，但 RF 还是要把 PYTHONPATH 加到搜索路径中去。

设置 CLASSPATH

CLASSPATH 是供 Jython 使用的，简单的修改方法也是修改系统的环境变量。在一些由 Java 实现的库的作用下，使用 Jython 时，可以只依赖 PYTHONPATH 而不依赖 CLASSPATH。

使用--pythonpath 参数

RF 有一个参数--pythonpath (-P)来增加 PYTHONPATH 中的路径，多个路径可由冒号分隔或多次使用该参数。路径可以匹配多个具体的路径，通常其中包含转意字符。例如：

--pythonpath libs/

--pythonpath /opt/testlibs:mylibs.zip:yourlibs

--pythonpath mylib.jar --pythonpath lib/STAR.jar --escape star:STAR

3.4.5 设置变量

在命令行中可以设置变量的值，可以通过--variable (-v)设置单个的值，也可以通过--variablefile (-V)设置变量文件。变量和变量文件在前面章节中介绍过，下面的例子也解释了如何使用它们。

--variable name:value

--variable OS:Linux --variable IP:10.0.0.42

--variablefile path/to/variables.py

--variablefile myvars.py:possible:arguments:here

--variable ENVIRONMENT:Windows --variablefile c:\resources\windows.py

3.4.6 静态检查（Dry Run）

RF 提供了一种叫做 Dry Run 的机制，运行用例但不执行关键字。这种机制常用来验证测试数据。如果 Dry Run 通过，那么测试数据的语法正确。这种模式可由命令行参数-runmode DryRun（不分大小写）来触发，这种功能在 RF2.5 中新增。

DryRun 运行失败可能有下面的原因：

- 使用不存在的关键字。
- 使用的关键字参数数量不对。
- 使用的关键字包含语法错误。

另外，一般运行的错误也能显示出来，例如，当测试库或测试源文件引用不正确时。

3.4.7 随机顺序执行

参数--runmode 可以设置用例以随机的顺序执行。它包含几种不同的参数值。

random:test

某个测试集中的测试用例以随机的顺序执行。

random:suite

测试集以随机顺序执行，但单个测试集中的用例顺序执行。

random:all

测试集和测试用例都以随机顺序执行。

例如：

pybot --runmode random:test my_test.txt

3.4.8 控制显示器输出

测试在命令行中的输出文字的宽度可以用参数--monitorwidth (-W)来设置。设置在显示器中的默认宽度为 78 个字符。参数--monitorcolors (-C)用来控制是否设置显示器中输出的颜色。ANSI 颜色在 Windows 系统中不起作用，只在类 UNIX 的系统中起作用。这个参数有三个可能的值，不区分大小写。

on

颜色在类 UNIX 的系统中起作用，在 Windows 系统中不起作用。这时默认设置。

off

不使用颜色，该情况适用于将输出重定向到文件中时。

force

颜色在类 UNIX 的系统中和 Windows 系统中都起作用。

例如：

pybot --monitorwidth 140 --monitorcolors OFF tests.html > output.txt

3.4.9 设置监听

所谓的侦听者用来监控用例的执行。可以使用参数--listener，侦听者必须位于模块搜索路径中，同测试库类似。

3.5 产生输出文件

许多输出文件在测试执行过程中产生，这些都和测试结果相关。本章节将介绍输出文件产生的是什么，如何配置让文件产生在哪，以及如何调整它的内容。

3.5.1 不同的输出文件

本章节将介绍产生不同的输出文件是什么，如何配置让文件产生在哪，输出文件在命令行中配置，把文件路径作为参数。使用 NONE(大小写不敏感)可以不产生输出文件。

输出目录

所有的输出文件的路径都可以设置为绝对路径，这样可以指明具体的位置，但是在另外一些情况还是考虑要设置为相对路径。默认的输出路径是执行测试所在的路径，它可以由--outputdir (-d)参数改变。设置为相对路径也是相对于执行测试的路径而言，相对路径会转换为绝对路径。不用考虑路径如何获得，当路径的上一级路径不存在时，系统会自动的创建该路径。

输出文件

输出文件是包含所有测试结果信息的 XML 格式的文件。日志文件、报告文件、摘要文件都是基于输出文件而产生，输出文件还可以在测试执行后合并或后处理。

命令行参数--output (-o)确定输出文件产生在哪里。输出文件是测试工程中产生，如果没有分割该文件，该文件的大小就会一直增长。默认的名称为 output.xml。如果分割输出文件，则产生一系列名称加数字的文件。当后处理输出文件时，只有显示的调用参数--output (-o) 才会产生新的输出文件。

日志文件

日志文件是包含执行用例细节的 HTML 文件。它具有分层结构显示测试集、测试用例、关键字细节。当想通过日志文件来查看测试执行的细节时，日志文件是必不可少的。虽然日志文件也包含统计数据，但报告文件和摘要文件是更好的查看高级别概况的手段。

命令行参数--log (-l)确定日志文件产生在哪里。如果没有显示的指定为 NONE，则日志文件会产生，名字为 log.html.

日志文件的开始

Selenium Demo Tests Log

Generated
20090414 14:51:46 GMT +03:00
1 year 167 days ago

Test Statistics

Total Statistics	Total	Pass	Fail	Graph
Critical Tests	10	10	0	
All Tests	10	10	0	

Statistics by Tag	Total	Pass	Fail	Graph
regression	10	10	0	
smoke	4	4	0	

Statistics by Suite	Total	Pass	Fail	Graph
Login Tests	10	10	0	
Login Tests . Higher Level Login	3	3	0	
Login Tests . Invalid Login	6	6	0	
Login Tests . Simple Login	1	1	0	

Test Execution Log

TEST SUITE: Login Tests — Expand All

Full Name:	Login Tests
Source:	/home/jth/workspace/seleniumlib/demo/login_tests
Start / End / Elapsed:	20090414 14:50:44.600 / 20090414 14:51:45.883 / 00:01:01.283
Overall Status:	PASS
Message:	10 critical tests, 10 passed, 0 failed 10 tests total, 10 passed, 0 failed

TEST SUITE: Higher Level Login — Expand All

TEST SUITE: Invalid Login — Expand All

TEST SUITE: Simple Login — Expand All

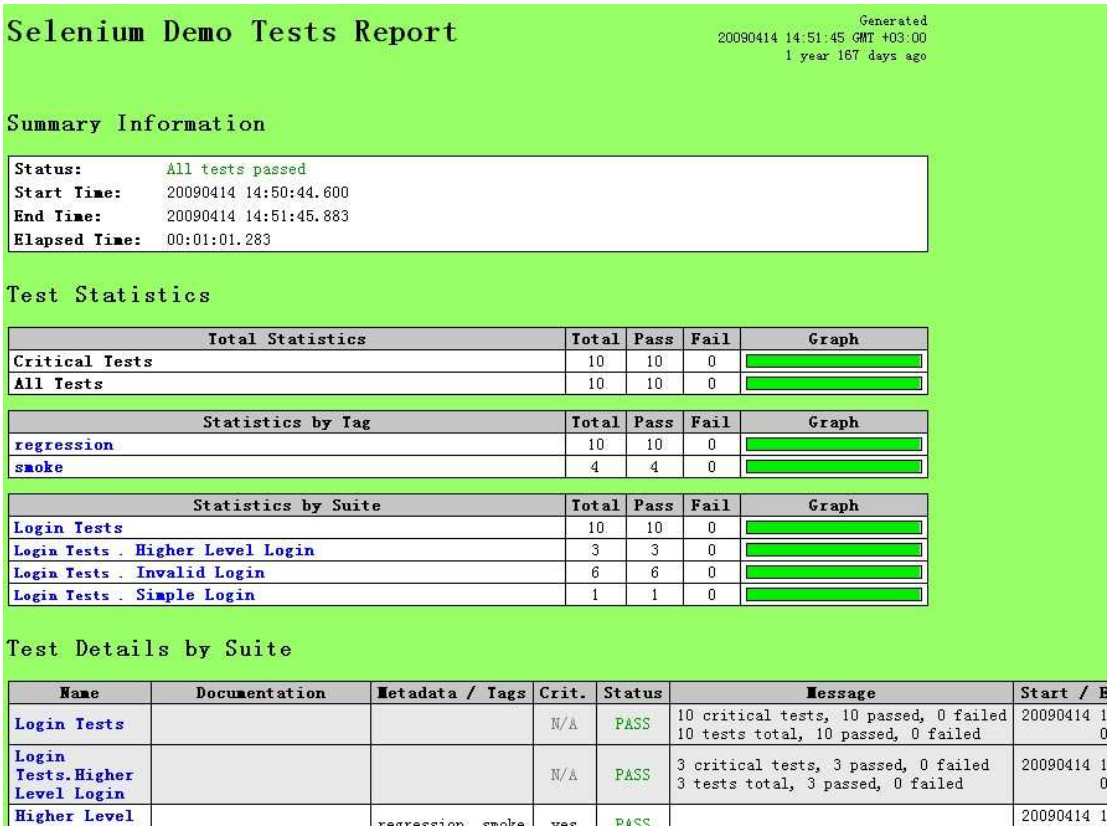
关键字细节视图

报告文件

报告文件是包含执行用例总体预览的 HTML 文件。它的统计信息包含 tag 和测试集，也展示的测试用例。当报告文件和日志文件都产生时，报告文件中包含日志文件的链接。我们很轻松的看到所有用例的执行情况，当所有关键测试都通过时，背景色为绿色，否则为红色。

命令行参数--report (-r) 确定报告文件产生在哪里。如果没有显示的指定为 NONE，则输出文件会产生，名字为 report.html.

测试执行成功的输出文件



Selenium Demo Tests Report

Generated
20090414 14:51:45 GMT +03:00
1 year 167 days ago

Summary Information

Status:	All tests passed
Start Time:	20090414 14:50:44.600
End Time:	20090414 14:51:45.883
Elapsed Time:	00:01:01.283

Test Statistics

Total Statistics	Total	Pass	Fail	Graph
Critical Tests	10	10	0	
All Tests	10	10	0	

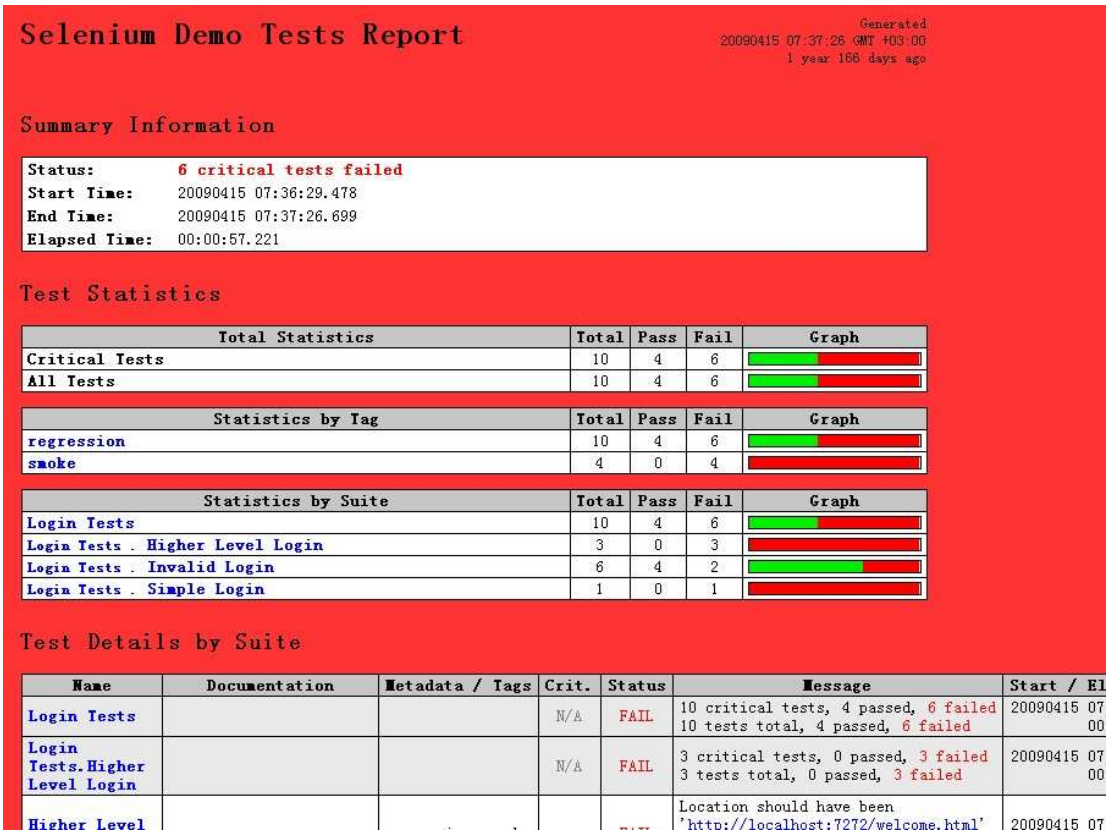
Statistics by Tag	Total	Pass	Fail	Graph
regression	10	10	0	
smoke	4	4	0	

Statistics by Suite	Total	Pass	Fail	Graph
Login Tests	10	10	0	
Login Tests . Higher Level Login	3	3	0	
Login Tests . Invalid Login	6	6	0	
Login Tests . Simple Login	1	1	0	

Test Details by Suite

Name	Documentation	Metadata / Tags	Crit.	Status	Message	Start / E
Login Tests			N/A	PASS	10 critical tests, 10 passed, 0 failed 10 tests total, 10 passed, 0 failed	20090414 1 0
Login Tests.Higher Level Login			N/A	PASS	3 critical tests, 3 passed, 0 failed 3 tests total, 3 passed, 0 failed	20090414 1 0
Higher Level		regression smoke	yes	PASS		20090414 1

测试执行失败的输出文件



Selenium Demo Tests Report

Generated
20090415 07:37:26 GMT +03:00
1 year 166 days ago

Summary Information

Status:	6 critical tests failed
Start Time:	20090415 07:36:29.478
End Time:	20090415 07:37:26.699
Elapsed Time:	00:00:57.221

Test Statistics

Total Statistics	Total	Pass	Fail	Graph
Critical Tests	10	4	6	
All Tests	10	4	6	

Statistics by Tag	Total	Pass	Fail	Graph
regression	10	4	6	
smoke	4	0	4	

Statistics by Suite	Total	Pass	Fail	Graph
Login Tests	10	4	6	
Login Tests . Higher Level Login	3	0	3	
Login Tests . Invalid Login	6	4	2	
Login Tests . Simple Login	1	0	1	

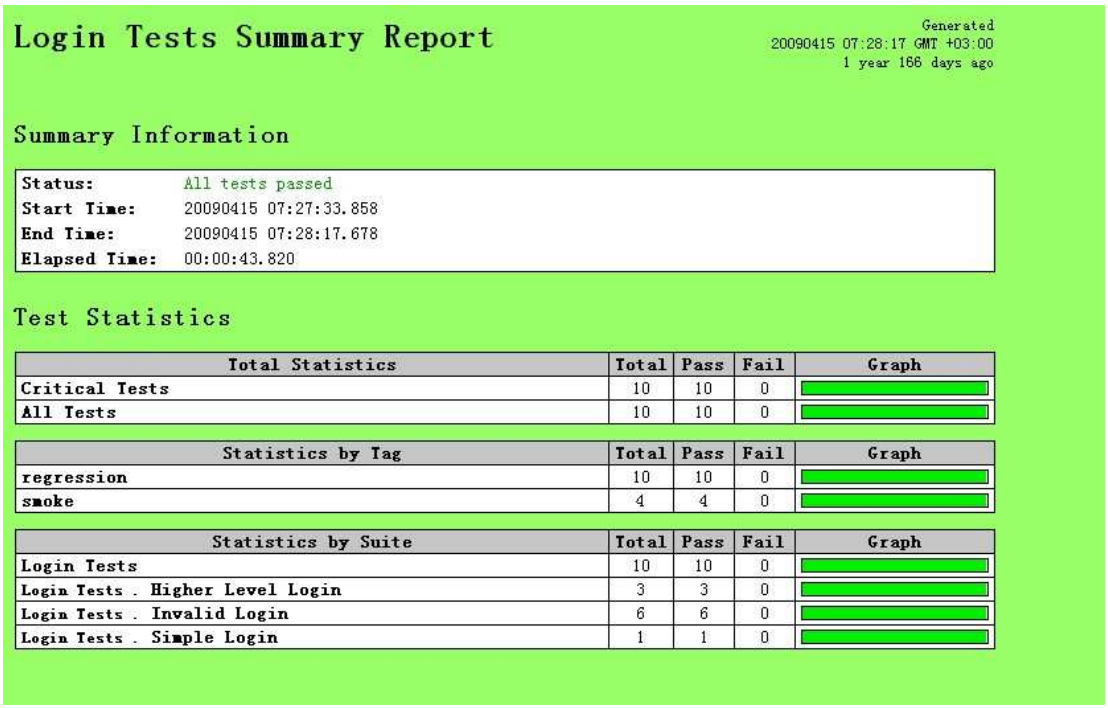
Test Details by Suite

Name	Documentation	Metadata / Tags	Crit.	Status	Message	Start / El
Login Tests			N/A	FAIL	10 critical tests, 4 passed, 6 failed 10 tests total, 4 passed, 6 failed	20090415 07 00
Login Tests.Higher Level Login			N/A	FAIL	3 critical tests, 0 passed, 3 failed 3 tests total, 0 passed, 3 failed	20090415 07 00
Higher Level					Location should have been 'http://localhost:7272/welcome.html'	20090415 07

总体摘要

摘要文件中包含和报告文件一样的统计信息，它的背景色取决于用例的执行状态来显示红色和绿色。但该文件中并不包含用例的细节，这使得该文件很小。当执行了大量的用例，需要一个高级别的统计视图时，该文件就显得很有价值。

摘要文件不是默认产生的，当需要时，可以使用命令行参数--summary (-S)来设置。



Login Tests Summary Report

Generated
20090415 07:28:17 GMT +03:00
1 year 166 days ago

Summary Information

Status:	All tests passed
Start Time:	20090415 07:27:33.858
End Time:	20090415 07:28:17.678
Elapsed Time:	00:00:43.820

Test Statistics

Total Statistics	Total	Pass	Fail	Graph
Critical Tests	10	10	0	
All Tests	10	10	0	

Statistics by Tag	Total	Pass	Fail	Graph
regression	10	10	0	
smoke	4	4	0	

Statistics by Suite	Total	Pass	Fail	Graph
Login Tests	10	10	0	
Login Tests . Higher Level Login	3	3	0	
Login Tests . Invalid Login	6	6	0	
Login Tests . Simple Login	1	1	0	

调试文件

调试文件是在测试执行过程中产生的纯文本文件。从测试库中得到的信息将都会写到调试文件中，包括测试集、测试用例、关键字的启动和停止。调试文件用来监控测试执行。可以通过一个独立的工具查看，在类 UNIX 系统中可以直接使用 tail –f 来查看。

调试文件要有命令行参数--debugfile (-b)显示调用.

输出文件时间戳

本节提到的所有文件，在使用参数--timestampoutputs (-T)后，都会自动的在文件名中加入当前时间，这是一个很少使用没有参数值的参数。时间值会加到基本名称之后。例如下面的例子将产生文件名为 output-20080604-163225.xml 和 mylog-20080604-163225.html 的文件。

pybot --timestampoutputs --log mylog.html --report NONE tests.html

设置标题

日志文件、报告文件和摘要文件产生时都以 Test Log, Test Report 或 Summary Report. 为前缀加顶级测试集命名。用户自定义标题可以使用命令行参数--logtitle,--reporttitle 和—summarytitle 来分别设置。在这个参数中下划线将自动转换为空格。

例如：

pybot --logtitle Smoke_Test_Log --reporttitle Smoke_Test_Report --include smoke mytests

rebot --summarytitle Overview --summary overview.html --log none --report none *.x

设置背景颜色

在默认的报告文件和摘要文件中，所有关键用例通过了背景色为绿色，否则为红色。可以使用参数—reportbackground 来自定义颜色，它需要两个或三个由冒号分开的颜色值作为参数：

--reportbackground blue:red

--reportbackground green:yellow:red

--reportbackground #00E

如果指明两种颜色，第一个颜色代替绿色，第二个代替红色。对色盲人群，最好将背景色设置为蓝色。

如果指明三种颜色，第一个颜色表示所有的测试都成功了，第二个颜色表示仅有非关键级别的测试失败了，第三个颜色表示存在关键测试失败。这一特性运行存在独立的背景色，例如黄色表示非关键测试失败。

具体的颜色值用例设置背景的 CSS 属性。这个值可以为 HTML 格式的颜色名（red），或十六进制的值（#F00），或是 RGB 三原色（rfb（255,0,0））。默认的绿色和红色使用十六进制表示为#9F6 和#F33。

3.5.2 日志级别

可用的日志级别

日志文件中记录的内容可以分为不同的级别。有一些信息是 RF 自己写入的，执行关键字时可以使用不同的级别来记录日志，可以的日志级别有：

FAIL

当关键字失败时使用，仅供 RF 自己使用。

WARN

用例显示警告，它会在控制台中记录也会在日志文件的执行错误部分记录。但它不会影响到测试的执行状态。

INFO

默认的日志级别。在日志文件中不作记录。

DEBUG

用于调试的目的，用于记录测试库的具体行为。当关键字失败时，自动记录代码中失败的位置。

TRACE

比调试级别更加详细，关键字的参数和返回值将自动记录。

设置日志级别

INFO 级别的日志中不会记录内容，但这一阈值可以由命令行参数来 --loglevel (-L)修改。这个参数后接任何可用的值作为参数值，这个值将变为新阈值。NONE 可以用来使该功能失效。

从 RF2.5.2 开始，使用 rebot 工具后处理输出文件是也可带上该参数。这就允许开始时以 TRACE 级别来运行用例，后来产生便于观察 INFO 级别的日志。默认情况下所有的测试信息都会包含进来。运行中忽略的信息不会被覆盖。

可以使用关键字 Set Log Level 来改变日志的级别。它和命令行参数--loglevel 等价，它可以返回旧的日志级别，便于将来恢复。例如在 test teardown 中。

3.5.3 分割输出文件

好处

当执行很多测试用例时，日志文件将会增大，在浏览器中打开将会很慢。另外，日志文件既然可以在输出文件准备好后产生，那么就可以在测试执行完后产生日志。对于一个长时间运行的测试来说在短时间内产生日志也不是一个难题，在其他测试还在运行时，我们就可以查看前面失败的细节。

输出文件的分割，为这些问题提供了解决办法。首先，分割日志意味着，单个日志文件将会更小，打开也越快。如果在测试执行过程中分割输出文件，那么低级别的日志文件将会在输出文件准备好后就能产生。输出文件通常由侦听接口产生，自动变量${LOG_FILE} 和 ${OUTPUT_FILE}中也包含着当前文件的路径。

指定分割级别

输出文件按照测试集的级别来分割。测试集拥有自己的输出文件，它将有一个索引文件，链接到低级别的输出文件中。只有输出文件和日志文件可以分割，并且它们的分割级别是一致的。当测试用例按照一个层次结构来组织时，输出文件分割将会给出一个最理想的测试结果。

可以设置命令行参数--splitoutputs 来设置文件的分割，后接分割级别作为具体的参数。索引文件将和输出文件具有相同的名字，日志文件或许不会分割，低级别的文件将由基本名字和运行的数量来命名。

样例

通过在下面的例子，我们将解释分割输出文件是多么的简单。在下面的的例子中，我们假设用例的组织为分层次的结构。

```
project
|-- component_a
|     |-- feature_a1
|     |     |-- a11.html
|     |     `-- a12.html
|     `-- feature_a2
|           `-- a21.html
|-- component_b
|     `-- feature_b1
|           |-- b11.html
|           `-- b21.html
`-- component_c
      |-- c1.html
      |-- c2.html
      `-- c3.html
```

在第一个例子中，所有的用例被执行，它们会在顶级测试集下正确的分割。命令行参数和输出结果如下所示。

pybot --splitoutputs 1 project

第一个例子中产生的所有输出文件

产生文件	内容
output.xml	测试集的信息，包括所有子测试集，和相关的低级别的输出文件。
output-001.xml	Component A 测试集的信息。
output-002.xml	Component B 测试集的信息。
output-003.xml	Component C 测试集的信息。
log.html	由 output.xml 产生的日志文件，包含所有低级别日志文件的链接。
log-001.html	由 output-001.xml 产生的日志文件。
log-002.html	由 output-002.xml 产生的日志文件。
log-003.html	由 output-003.xml 产生的日志文件。
report.html	报告文件。

事实上，所有的输出文件的分割对于输出文件的后处理来说是透明的，只要针对索引文件使用 rebot 命令即可。 使用 rebot 工具时不用考虑输出文件是否被分割。由第一个例子产生的输出文件还可以在分割，如下：

rebot --splitoutputs 2 output.xml

在这个例子中，采用第二个级别来分割。下表列出了所有的输出文件，由于 rebot 命令不能产生新的输出文件，输出文件就没有列在下表中。

第二个例子中产生的所有输出文件

产生文件	内容
log.html	整个 Project 和 Component 测试集的信息，但不包含子测试集。包括低级别的日志文件的链接。
log-001.html	整个 Feature A1 测试集的信息。
log-002.html	整个 Feature A2 测试集的信息。
log-003.html	整个 Feature B1 测试集的信息。
log-004.html	整个 C1 测试集的信息。
log-005.html	整个 C2 测试集的信息。
log-006.html	整个 C3 测试集的信息。
report.html	报告文件。

在最后的例子中，采用第三个级别来分割。同样显示了如何设置输出文件的名字。

rebot --splitoutputs 3 --log mylog.html --report none output.xml

第三个例子中产生的所有输出文件

产生文件	内容
mylog.html	整个 Project 测试集，并包含到下面两层的子测试集。包含 c?.html 中的测试用例。包含所有低级别日志文件的链接。
mylog-001.html	整个 A11 测试集。
mylog-002.html	整个 A12 测试集。
mylog-003.html	整个 A21 测试集。
mylog-004.html	整个 B11 测试集。
mylog-005.html	整个 B12 测试集。

如果分割级别高于三，所有信息都会在索引文件中，这种情况和没采用分割时一样的。

3.5.4 配置统计数据

有一些参数可以用例设置 tag 统计信息、测试集统计信息和 tag 表格中的测试细节数据。这些参数都可以在测试执行过程中和后处理输出文件时使用。

配置显示的测试集数据

当一个深层次的测试集在执行时，如果在统计数据表中显示所有测试集的数据，那么整个表格将变得非常难读。但是你可以使用命令行参数 –suitestatlevel 来控制测试集显示的层次。后接整数作为所需测试集的深度，如果输入 0，则整个表格不显示。

包含或排除 tag 数据

当使用很多 tag 时，tag 数据统计表将变的非常拥挤。可以使用命令行参数--tagstatinclude 和--tagstatexclude 来选择显示哪些 tag，方法类似于在选择用例时使用参数—include and—exclude 。

--tagstatinclude some-tag --tagstatinclude another-tag

--tagstatexclude owner-*

--tagstatinclude prefix-* --tagstatexclude prefix-13

这一设置将影响到 tag 表格中测试细节的显示，只有选中的 tag 才能看到细节。这将会使报告变得很小，所有我们推荐使用 tag。

合并 tag 统计数据

命令行参数--tagstatcombine 可以从多个 tag 中合并统计数据，产生新的 tag。这新产生的 tag 可以在 tag 数据表中显示，想对应匹配的测试数据也会显示在 tag 测试详细信息表中。可以用三种方式设置参数。

- 一个作为通配模式的 tag

  所有匹配模式的 tag 都被合并。

- 两个以上的 tag 由 AND 或&来分割

  合并的统计数据包含所有列出来的 tag。Tag 也可以按照通配模式来给出。

- 两个以上的 tag 由 NOT 来分割

  合并的统计数据包含第一个 tag 的数据，不包含第二个 tag 的数据。Tag 也可以按照通配模式来给出。

下面的例子将说明上述的使用方式，下图显示了当使用下面的参数时，tag 数据表中的一个片段。

--tagstatcombine owner-*

--tagstatcombine smokeANDmytag

--tagstatcombine smokeNOTowner-janne*

owner-*	4	4	0	
smoke & mytag	1	1	0	
smoke NOT owner-janne*	1	1	0	

从上面的例子中可以看到，新增加的统计数据的名称由给定的模式来产生。在这个方法中名称看起来有点奇怪，但从 RF2.0.2 开始，我们就可以为它起一个具体的名称了。具体的名字可以在模式后面接冒号来添加。下面的例子中产生了一个 tag 在报告文件和日志文件中显示为 Critical Tests：

--tagstatcombine *NOTnon-critical:Critical_Tests

创建 tag 链接

可以通过命令行参数--tagstatlink 为 tag 统计表的数据添加外部的链接。参数值以下面的格式给出 tag:link:name，其中，tag 为分配给链接的标记，link 为产生的链接，name 为链接指定的名称。

Tag 可以为一个单一的 tag,，也可以使用通配符，其中*代替任意字符串，？代替一个字符。当 tag 中是有了模式，在 link 中的语法%N，N 表示匹配的字符串，从 1 开始。

下面的例子将说明上述的使用方式，下图显示了当使用下面的参数时，tag 数据表中的一个片段。

--tagstatlink mytag:http://www.google.com:Google

--tagstatlink jython-bug-*:http://bugs.jython.org/issue_%1:Jython-bugs

--tagstatlink owner-*:mailto:%1@domain.com?subject=Acceptance_Tests:Send_Mail

jython-bug-1777567	[jython-bugs]	1	1	0
jython-bug-1778514	[jython-bugs]	1	1	0
mytag	[Google]	1	1	0
owner-janne.t.harkonen	[Send mail]	3	3	0
owner-laukpe	[Send mail]	1	1	0

增加 tag 注释

可以使用参数--tagdoc 来为 tag 添加注释，参数值以下面的格式给出 tag:doc。Tag 为准备添加注释的 tag 的名称，它可以采用通配符的方式匹配多个 tag。Doc 就是注释。这里的下划线会自动的转化为空格，它可以包含 html 格式的文件。注释会在 tag 详细信息表中显示，在 tag 统计信息表中以 tooltip 的方式显示。例如：

--tagdoc mytag:My_documentation

--tagdoc regression:*See*_http://info.html

--tagdoc owner-*:Original_author

3.5.5 后处理输出文件添加参数

下面这些参数只在使用 rebot 工具后处理输出文件时可用。

设置输出文件合并的时间

当合并输出文件时，我们可以使用参数--starttime 和—endtime 来设置合并测试集的开始和结束时间。默认的合并情况是没有这些数值的，这两个参数就使合并变的很方便。当给出开始和结束时间后，中间花费的时间会自动计算出来。另外也可以由所有子测试集所用的时间相加得到。

时间的参数值必须以时间戳(YYYY-MM-DD hh:mm:ss.mil)的方式给出，所有的分割字符都可以选择，毫秒到小时部分可以省略。例如 2008-06-11 17:59:20.495 和 20080611-175920.495 和 20080611175920495 是等价的，似乎只有 20080611 在其作用。

例如：

rebot --starttime 20080611-17:59:20.495 output1.xml output2.xml

rebot --starttime 20080611-175920 --endtime 20080611-180242 *.xml

在输出文件中移除关键字

输出文件的大部分内容来自关键字，特别是日志文件的信息。在产生高级别的报告时，日志文件并不是不可或缺的，所以日志文件中的关键字和信息仅仅是占不必要的空格。这种情况下，命令行参数--removekeywords 可以去除不必要的关键字。有两种可能的值。

- ALL

  所有关键字都无条件的删除。

- PASSED

  只删除通过的关键字，大多数情况下，用这种方式产生的日志文件都可以包含足够的信息来研究失败原因。

删除掉关键字使得输出文件非常小，产生起来也很快。当关键字被移除后，高级测试的测试集的名称、参数和状态将被保护，所以产生日志文件、查看测试用例高级结构是可能的。

3.5.6 系统日志

RF 拥有纯文本的系统日志，它包含下列信息：

- 处理和跳过的数据文件
- 导入测试库、源文件和变量文件
- 执行测试集和测试用例
- 产生输出文件。

通常用户不会需要这个信息，但是当查看测试库和 RF 本身的问题时，系统文件就变得很重要了。系统日志不会默认产生，它可以使用系统环境变量 ROBOT_SYSLOG_FILE 来激活，它包含选择文件的路径。

系统日志具有和普通的日志文件一样的日志级别，exception 代替了普通日志中的 FAIL，它拥有 ERROR 级别。使用的阈值级别可以通过时环境变量 ROBOT_SYSLOG_LEVEL 来改变，正如下面的例子所示。可能的非异常类的错误和警告将会现在系统文件中，也会显示在控制台和普通的日志文件中，例如：

```
#!/bin/bash
export ROBOT_SYSLOG_FILE=/tmp/syslog.txt
```

export ROBOT_SYSLOG_LEVEL=DEBUG

pybot --name Syslog_example path/to/tests

4 扩展 Robot Framework

4.1 创建测试库

Robot Framework 实际的测试能力是由测试库提供的，现在已经有不少测试库了，有些是直接绑定在 Robot Framework 核心中的，但是仍然经常有需要来创建一些新的测试库。创建测试库并不太复杂，就像本章介绍的例子一样，Robot Framework 的库 API 都是简单而直接的。

4.1.1 介绍

支持的编程语言

Robot Framework 自身是用 Python 编写的，那么很自然地，也能使用 Python 扩展测试库。如果在 Jython 运行 Robot Framework 的话，那么测试库也可以用 Java 来实现。只要 Python 代码没有使用在 Jython 中不可用的语法或者模块，那么它们就能够同时运行在 Python 和 Jython 中。当使用 Python 的时候，尽管可以很容易地在使用 ctypes 模块的 Python 库中来与 C 代码交互，但是也可以通过 Python C API 使用 C 语言来实现测试库。

用这些自然支持的语言实现的测试库也能作为使用其他编程语言实现的功能的包装者。这种方式一个很好的例子就是 Remote Library，另外一种广泛使用的方式就是作为独立的进程来运行外部脚本或者工具。

> 小贴士：
>
> 专为 Robot Framework 测试库开发者准备的 Python 指南包含了足够的 Python 语言信息来使用 Python 开始编写测试库，同时该指南也包含一个简单的测试库例子和一些测试用例，因此你可以在你的机器上运行或者研究它们。

不同的测试库 API

Robot Framework 有三种不同的测试库 API

静态 API

这是最简单的方法。采用一个 Python 的 module 或者 Python 和 Java 的 Class，提供一系列的关键字。module 或 class 的方法名和关键字相匹配，关键字和方法具有同样的参数。这些关键字可以通报异常，输出日志，返回需要的值等等。

动态 API

动态库是一些这样的 Class，它们要实现两个方法，一个用来获取它们自己实现了的关键字的名字，另外一个用来执行这些关键字。要实现哪些关键字以及怎样来运行关键字，都是在运行期决定的，但是可以像静态 API 一样通报异常，输出日志，返回需要的值。

混合 API

这种方式介于动态和静态 API 之间，这样的 Class 中有一个方法用来发现它们实现了哪些关键字，同时这些关键字也可以像静态 API 一样直接使用。所以，混合 API 除了多了一个方法来搜索它们实现的关键字以外，其他的一切都跟静态 API 一模一样。

以上这些 API 都将在本章描述，因为一切都依赖于静态 API 如何工作，所以静态 API 的功能我们首先来讨论。至于动态 API 和混合 API 跟静态 API 之间的差异将在它们自己的章节中进行讨论。

本章的例子主要使用的是 Python，但是只会 Java 的开发者也应该很容易理解。在少数有差别的 API 处，Python 和 Java 两种用法都有足够的例子。

4.1.2 创建测试库 Class 或者 Module

测试库名称

当一个测试库被引入到用例中的时候，它的名称跟实现该测试库的模块名或者类名是相同的。比如，如果有个 Python 模块叫 MyLibrary(同时，文件名为 MyLibrary.py)，它就创建了一个名为 MyLibrary 的库。类似地，一个名为 YourLibrary 的 Java 类，当它没有在任何包中时，它就创建了一个同名的库。

Python 类总是被写在一个模块中的。如果实现一个库的类的名称跟它所在的模块同名，则 Robot Framework 允许你引入它的时候去掉模块名。比如，MyLib.py 中有个实现类的名称为 MyLib，那么引入它的时候可以直接使用它的类名 MyLib。如果模块名和类名不同，使用测试库要同时使用模块名和类名，比如 mymodule.MyLib。

Java 类如果不是在默认包中，比如，MyLib 类在 com.mycompany.myproject 包中，那么它就要通过 com.mycompany.myproject.MyLib 这样的名字来引入。

> 小贴士：
>
> 如果测试库名称真的太长了，比如当 java 报名很长的时候，推荐你通过 WITH NAME 语法给测试库添加一个别名。

提供参数给测试库

所有实现测试库的类都可以带参数。在 Setting 表中，这些参数放到库名后面来指定，然后当 Robot Framework 创建被引入的库的实例时，就把这些参数传递给库的构造器。但是，如果库是用模块来实现的，则不能带任何参数。

测试库需要的参数个数跟它的构造器接收的参数个数相同。默认值和可变个数参数也能工作，就像关键字的参数一样，例外地，对于 Java 测试库，并没有可变参数支持。参数被传递给测试库时，可以通过变量，因此它们是可以修改的，比如通过命令行。

Importing a test library with arguments

Setting	Value	Value	Value
Library	MyLibrary	10.0.0.1	8080
Library	AnotherLib	${VAR}	

实现库的例子，第一个用 Python，第二个用 Java，这两个库的调用参见上面的表格。

```
============================in Python=============================
from example import Connection

class MyLibrary:

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

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

```
============================in Java================================
public class AnotherLib {

    private String setting = null;

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

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

测试库的范围

被类实现的测试库有一个内部状态，该状态可以通过关键字和库构造器的参数来改变。因为该状态能够影响关键字的实际行为，所以确认它在一个测试用例中的改变并不会意外地影响到其他测试用例是非常重要的。这些依赖可能会导致难以调试的问题，比如，当新的测试用例被加入，它们使用这个测试库会出现不一致的情况。

Robot Framework 试图让测试用例之间保持独立：默认地，它为每个测试用例创建测试库的新实例。然而，这种方式并不总是合适的，因为有时候测试用例应该能够共享一个共通的状态。

通过类的一个属性 ROBOT_LIBRARY_SCOPE，测试库可以控制新的库什么时候被创建。这个属性必须是字符串，并且它只有以下三个值：

TEST CASE

> 为每个测试用例创建一个新的实例。A possible suite setup and suite teardown share yet another instance.这是默认的。

TEST SUITE

> 为每个测试集创建一个新的实例。The lowest-level test suites, created from test case files and containing test cases, have instances of their own, and higher-level suites all get their own instances for their possible setups and teardowns.

GLOBAL

在整个测试执行过程中，只有 1 个实例被创建。它被所有测试用例和测试集共享。由模块创建的库总是 global 的。

TEST SUITE 或者 GLOBAL 范围被测试库使用时，就有了一个状态，推荐测试库还要有个特殊的关键字来清除这个状态。这个关键字然后能被使用，比如，在一个测试集的 setup 或者 teardown 被使用，这样的话，就能保证下一个测试集的测试用例能在一个已知的状态下启动。比如，SeleniumLibrary 用 GLOBAL 范围来使不同的测试用例使用同一个浏览器而不需要每次都重新打开，同时，它也有 Close All Browsers 关键字来很容易地关闭所有浏览器。

Example Python library using the `TEST SUITE` scope:

```
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
```

Example Java library using the `GLOBAL` scope:

```
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;
    }
}
```

声明测试库的版本

当一个测试库被使用的时候，Robot Framework 试图判断它的版本。这个信息后面会被写入到 syslog 中来提供调试信息。测试库文档工具 libdoc.py 也把这个信息写入到它生成的关键字文档中。

版本信息是通过属性 ROBOT_LIBRARY_VERSION 读取的，类似于测试库范围从 ROBOT_LIBRARY_SCOPE 读取。如果 ROBOT_LIBRARY_VERSION 不存在，就尝试读取__version__属性。这些属性必须是类或者模块的属性。对于 Java 库来说，版本属性必须被声明为 static final 的。

An example Python module using `__version__`:

```
__version__ = '0.1'

def keyword():
    pass
```

A Java class using `ROBOT_LIBRARY_VERSION`:

```
public class VersionExample {

    public static final String ROBOT_LIBRARY_VERSION = "1.0.2";

    public void keyword() {
    }
}
```

4.1.3 创建静态关键字

关键字名称

当静态库的 API 被使用的时候，Robot Framework 通过反射找到测试库中对应的实现方法。而对于动态库 API 和混合库 API，关键字的名称是直接从测试库中获取的。自然地，Robot Framework 能够看见所有的 public 方法，它也能自动忽略以下划线开头的方法。对于 Java 库，java.long.Object 中实现的方法且没有被覆盖的，

它也会忽略。

在测试数据中使用的关键字名称被用来比较方法名， 名称的比较是不区分大小写的，空格和下划线也会被忽略。比如，有个名为 hello 的方法，可以匹配的关键字名称 Hello,hello,甚至 h e l l o。同样，do_nothing 和 doNothing 方法也能在测试数据中被用作 Do Nothing 关键字。

Example Python library implemented as a module in the *MyLibrary.py* file:

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

def do_nothing():
    pass
```

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() {
    }

}
```

下面的例子显示了怎么使用上面的测试库例子。如果你要自己尝试一下的话，必须保证测试库在库搜索路径中。

Using simple example library

Setting	Value	Value	Value
Library	MyLibrary		

Test Case	Action	Argument	Argument
My Test	Do Nothing		
	Hello	world	

关键字参数

在静态和混合库 API 中，一个关键字需要多少参数，取决于实现它的方法。使用动态库 API 的测试库有其他的方法来取得参数信息，本节不做相应的描述。

最通用也是最简单的情况是，当一个关键字需要的参数是固定个数的。在这种情况下，Python 和 Java 方法都带有同样的固定参数。

Example Python keywords taking different numbers of arguments:

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

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

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

带有默认值的参数

一个关键字的某些参数带有默认值，这种方式经常很好用。Python 和 Java 有不同的语法来处理带默认值参数的方法，并且当为 Robot Framework 创建测试库的时候，这些语言的自然语法可以使用。

- **Python 中带有默认值的参数**

在 Python 中，带有默认值的参数可以声明在方法签名中。如下例子：

```
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)
```

上面的第一个例子关键字可以用 0 个或者 1 个参数来调用。如果不提供参数，则 arg 取得默认值，如果有一个参数，arg 就取这个参数值，如果多于一个参数，将导致失败。第二个例子，第一个参数是必须输入的参数，但是第 2、3 个参数有默认值，所以使用该关键字可以有 1 到 3 个参数。

```
Using keywords with variable number of arguments
```

Test Case	Action	Argument	Argument	Argument
Defaults	One Default			
	One Default	argument		
	Multiple Defaults	required arg		
	Multiple Defaults	required arg	optional	
	Multiple Defaults	required arg	optional 1	optional 2

- **Java 中带有默认值的参数**

在 Java 中，一个方法可以根据不同的签名有不同的实现。Robot Framework 将这些参数不同的实现视为一个关键字。因此，这种语法可被用来提供默认值的参数。下面的 Java 代码例子展示了上面 Python 代码同样的功能：

```
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");
}
```

可变数量的参数

Robot Framework 也支持带有可变数量参数的关键字。类似地，Python 和 Java 各自的语法是不相同的。

- **Python 中带有可变数量的参数**

Python 支持方法接收可变数量的参数。同样的语法也可以在测试库中使用，如下面的列子：

```
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
```

Using keywords with a variable number of arguments

Test Case	Action	Argument	Argument	Argument
Varargs	Any Arguments			
	Any Arguments	argument		
	Any Arguments	arg 1	arg 2	arg 2
	...	arg 4	arg 5	
	One Required	required arg		
	One Required	required arg	another arg	yet another
	Also Defaults	required		
	Also Defaults	required	these two	have defaults
	Also Defaults	1	2	3
	...	4	5	6

- **Java 中带有可变数量的参数**

Robot Framework 支持 Java 可变参数语法。

Robot Framework 对 Java 可变参数的支持有一个限制：该方法只能有一个签名。因此，一个 Java 关键字既

拥有带默认值的参数，又拥有可变数量参数，是不可能的。当然，拥有一些必输的参数还是可以的，如下例子：

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

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

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

参数的类型

通常情况下，关键字参数是作为字符串传给 Robot Framework 的。如果关键字需要其他类型的参数，要么就使用变量，要么利用内建关键字将字符串转换成需要的类型。Java 关键字基础类型也自动被强制转换。

- **Python 中的参数类型**

因为在 Python 中的参数不需要任何类型信息，所以在使用 Python 库的时候不可能自动化地将字符串转换成其他类型。如果参数的数量正确，调用关键字时将会成功，如果参数类型不匹配将在执行关键字时失败。幸运的是，在 Python 中，可以很简单的将参数转换为合适的类型。

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

- **Java 中的参数类型**

Java 方法的参数具有类型，从 Robot Framework 2.1 开始，所有基础类型都能被自动处理，在测试数据中的

普通字符串参数能在运行时被强制转换成正确的类型。可以被转换成的类型如下：

- 整形 (byte, short, int, long)
- 浮点型 (float and double)
- 布尔型
- 上述类型的对象类型，比如 java.lang.Integer

参数的强制转换只有在参数的类型一致时才能进行，在下面的例子中，doubleArgument 和 compatibleTypes 关键字参数的强制转换能进行，而 confilctingTypes 的不能。

```
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) {}
```

> **译者注：**
>
> 是不是 conflictingTypes 关键字的参数 arg2 类型 int 是简单类型，改成 Integer 即可？

强制转换可以作用在数字类型上（如果测试数据中有字符串包含数字），也可以作用在布尔型上，只是字符串必须是 true 或者 false。强制转换只能在测试数据本来是一个字符串的时候才能进行，当然，使用变量（它包含了正确的数据类型）也可以。使用变量是当关键字具有冲突签名的时候，如(String arg1, int arg2)，能进行强制转换的唯一选择。

Using automatic type coercion

Test Case	Action	Argument	Argument	Argument
Coercion	Double Argument	3.14		
	Double Argument	2e16		# scientific notation
	Compatible Types	Hello, world!	1234	
	Compatible Types	Hi again!	-10	true

Using automatic type coercion

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

4.1.4 与 Robot Framework 通信

当一个实现关键字的方法被调用之后，它就可以使用任何机制来与被测系统通信。它也可以发送消息给 Robot Framework 的日志文件，返回变量存储的值，更重要的是报告关键字是 pass 还是 fail 之类的信息。

报告关键字的状态

报告关键字状态是通过简单的使用异常（exceptions）实现的。如果一个被执行的方法抛出一个异常，该关键字的状态就是 FAIL，如果它正常返回了，则状态为 PASS。

错误信息包含了异常的类型和信息，它被显示在日志、报告和控制台中。对于常见的异常（比如，AssertionError，Exception 和 RuntimeError），只显示异常信息，而对于其他的异常，信息以这样的格式“异常类型：实际信息”输出。不管哪种异常，对用户来说重要的还是其异常信息要尽可能的具有价值。

为了不让报告太长而导致阅读性差，如果错误信息长度超过了 40 行，则自动地将其从中间截断，再输出到报告中。但是，失败关键字的日志文件中还是会显示完整的错误信息。

异常的跟踪可以通过 DEBUG 日志级别来输出日志。这些信息在日志文件中默认是不可见的，因为普通用户很少会对它们有兴趣。当开发测试库的时候，使用—loglevel DEBUG 来运行测试通常是个好方法。

停止测试执行

从 Robot Framework 2.5 开始，可以通过让一个测试用例失败来停止整个测试运行。将一个特殊的属性 ROBOT_EXIT_ON_FAILURE（值为 True）设置到关键字抛出的异常中，就能很容易的达到这种目的。请看下面展示的例子：

Python:

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

Java:

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

不管失败继续执行

从 Robot Framework 2.5 开始，也可以让整个测试执行在出现了 failure 的时候继续执行。将一个特殊的属性 ROBOT_CONTINUE_ON_FAILURE（值为 True）设置到关键字抛出的异常中，就能很容易的达到这种目的。请看下面展示的例子：

Python:

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

Java:

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

输出日志信息

异常信息并不是输出信息给用户的唯一方式。方法也可以简单地通过写信息到标准输出（stdout）或者标准错误（stderr）的方式来输出信息到日志文件中，方法也可以使用不同的日志级别。

默认地，一个方法写入标准输出的一切信息都会被当作 INFO 级别输出到日志文件中。写入标准错误的信息也是被同样处理的，不同的是，在关键字执行完毕后，写入标准错误的信息还会被反馈给原始的标准错误。如果在测试被执行的时候，你需要一些信息显示控制台中，你就可以使用标准错误。

要使用其他的日志级别(非 INFO)或者创建一些信息，可以通过将级别绑定到信息中，格式如*LEVEL*Actual log message，这里*LEVEL*必须位于信息的头部，并且 LEVEL 的值必须是 TRACE、DEBUG、INFO、WARN 、HTML 其中之一。

警告

从 Robot Framework 开始，警告（WARN）级别的信息会自动被写入控制台和日志文件中单独的测试执行错误那一节。这样就使得警告比其他信息更加可见，可以使用警告来报告一下重要而不危险的问题给用户。

输出 HTML 格式的日志

测试库日志输出的一切都将被转换为一种可以安全的展现的格式，比如 HTML。比如，foo就会被原样显示，而不会显示粗体的 **foo**。如果测试库想要使用格式、超链接、图片等等，它们可以使用特殊的日志级别 HTML。Robot Framework 将这些信息以 INFO 级别直接写入日志文件，因此它们可以使用任何 HTML 语法。注意，这种特性要小心使用，因为在一个不合适的位置放置“</table>”标签会彻底地将这个日志文件打乱。

日志例子

大多数情况下，INFO 级别已经足够。它之下的级别，DEBUG 和 TRACE，在输出调试信息的时候比较有用。这些信息通常不会显示，但是它们可以很容易的调试测试库自身可能存在的问题。WARN 级别可以让信息更加可见，HTML 级别对于需要其他格式时很有用。

下面的例子澄清了怎样使用不同的日志级别。Java 程序员应该将 print ‘message’看成是 System.out.println(‘mssage’)的伪代码。

```
print 'Hello from a library.'
print '*WARN* Warning from a library.'
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>'
```

```
16:18:42.123    INFO    Hello from a library.
16:18:42.123    WARN    Warning from a library.
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 <b>normal text</b>.
16:18:42.123    INFO    This is bold.
16:18:42.123    INFO    Robot Framework
```

返回值

关键字与 Robot Framework 通信的最后一种方法就是返回信息，这些信息可能是从被测系统返回的或者其他方法创建的。返回的这些值可以赋给测试数据中的变量，然后这个变量又作为其他关键字的输入，甚至它们可以来自不同的测试库。

返回值在 Python 和 Java 中都采用 return 语句。如下面的例子所示，一般地，一个值被赋给一个标量变量。下面的例子也展示了如何返回对象，并通过扩展变量语法来访问对象的属性。

```
from mymodule import MyObject

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

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

Return one value from keywords

${string} =	Return String	
Should Be Equal	${string}	Hello, world!
${object} =	Return Object	Robot
Should Be Equal	${object.name}	Robot

关键字也能通过多个标量变量或者构成一个列表变量返回多个值。这种用法必须要求返回的值是 Python 的列表或者元组，Java 就是数组、列表、迭代器。

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

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

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	${v1} ${v2}	a list	
Should Be Equal	@{li}[0] @{li}[1]	of strings	

4.1.5 发布测试库

测试库文档

一个测试库如果没有文档来说明它有哪些关键字以及这些关键字能做什么，那么它基本上就是没用的。为了便于维护，测试库文档应该从源代码中生成，也就意味着需要使用 Python 的 docstrings 和 Java 的 Javadoc，如下：

```
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
/**
 *  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 都有工具来生成 API 文档。然而，这些工具的输出对于一些用户来说还是稍显技术化了。另外一个可选的工具就是 Robot Framework 自己的文档工具 libdoc.py。这个工具既能够从使用静态库 API 的 Python 和 Java 库中生成测试库文档，也可以处理动态库 API 和混合库 API。

一个关键字文档的第一行通常被用作一个特殊的目的，它应该包括该关键字的简要概述。这行内容一方面会被 libdoc.py 当作关键字的 tool tip，另一方面也会被显示到测试日志中，后者对使用静态 API 的 Java 测试库并不适用。原因是 Java 的文档会在编译期丢失，运行期是不可见的。

对测试库进行测试

任何有价值的测试库都需要彻底地进行测试以防止 bugs 存在。当然，这样的测试应该是自动化的，这样的话，测试库改变的时候重新运行才会很容易。

Python 和 Java 都有很好的单元测试工具，它们非常适合用来验证测试库。使用它们来验证测试库跟使用它们做其他测试并没有什么区别。熟悉这些工具的开发者基本不需要再学习任何新的东西，不熟悉的开发者

本来就需要学习这些工具的。

使用 Robot Framework 本身来验证测试库会很容易，对于它们来说，那种方式是真正的端到端的验收测试。在 BuiltIn 测试库中有很多有用的关键字可以达到这个目的。一个特别值得注意的是 Run Keyword And Expect Error 关键字，它能够验证关键字是否如预期地报告了错误。

使用单元测试方法还是验收测试方法取决与具体情况。如果需要模拟真实被测系统，那么通常单元测试比较容易。另一方面，验收测试允许关键字通过 Robot Framework 运行。如果你无法决定，当然也可以两种方法都用。

打包测试库

当一个测试库被实现、文档化、验证之后，还需要打包分发给用户。简单的库（只有一个文件），让用户拷贝到任意位置，并设置到测试库搜索路径就可以了。较为复杂的库则应该打包，以使安装更加容易。

因为测试库都是普通的程序代码，所以它们可以用普通的打包工具来打包。对于 Python 来说，一个很好的选择就是 Python 标准库中的 distutils，或者新的 setuptools。这两个工具的好处是测试库会被自动的安装到测试库搜索路径中。

当使用 Java 的时候，通常是将测试库打包成 Jar 文件。在运行测试之前，Jar 文件必须被设置到测试库搜索路径中去，不过这可以很容易的通过创建启动脚本来自动地实现。

不推荐的关键字

有时候，有必要将已经存在的关键字替换成新的，或者干脆删除某些关键字。只是告诉用户这样的变化可能还不够，更加有效的是在运行期给出警告。从 Robot Framework 2.0.2 开始，可以将关键字标识为不推荐的（deprecated）。这样的话，就更加容易从测试数据中发现老的关键字，并删除或者替换它们。

将*DEPRECATED*添加到关键字文档的开始处，就可以让该关键字变成不推荐的（deprecated）。当这种关键字被执行的时候，警告信息将会输出到控制台和单独的测试日志文件中。例如，如果下面的关键字被执行，日志文件中将会显示下面的警告信息。

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

    This keyword does something to given `argument` and returns the result.
    """
```

```
    return do_something(argument)
```

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

这种不推荐机制在绝大多数的测试库都能工作，也能在用户关键字工作。唯一的例外就是，关键字是在使用静态库接口的 Java 测试库中实现的时候，因为他们的文档在运行期是不可见的。对于这种关键字，可以使用用户关键字来包装，然后将对应的用户关键字设置为不推荐的。

There is a plan to implement a tool that can use the deprecation information for automatically replacing deprecated keywords. The tool will most likely get the name of the new keyword from the documentation so that it searches words inside backticks (`).

Thus it would find *Other Keyword* from the earlier example. Note that libdoc.py also automatically creates internal links using the same syntax.

> 译者注：
>
> 这里不翻译，大概意思是将有计划开发一个工具，这个工具可以从不推荐信息中自动地用新的关键字替换掉不推荐的关键字。

4.1.6 动态库 API

动态库 API 很多方式跟静态 API 类似。比如，报告一个关键字的状态、输出日志、返回值这些功能跟静态 API 处理的方式都是一样的。更加重要的是，引入动态库并使用它们的关键字，与其他测试库并没有不同。因此，你根本不用关心使用的是什么类型的 API（静态、动态、混合在使用上都是一样的）。

静态和动态测试库之间唯一的不同就是，Robot Framework 如何发现测试库实现了哪些关键字，包括这些关键字的参数和文档，以及这些关键字实际执行的方式。对于静态 API，这些动作都是通过反射（除了 Java 测试库的文档）来完成的。但是对于动态测试库，则有特殊的方法来实现这些目的。

动态 API 的一个好处就是你能更加灵活的组织你的库。对于静态 API，你要将所有的关键字放到一个类或者一个模块中，但是对于动态 API，只要你愿意，你可以让每一个关键字都作为一个单独的类来实现。对于

Python 而言，这种场景也许不是那么重要，因为它的动态特性和多继承就已经提供了丰富的灵活性，所以混合库 API 通常是更好的选择。

动态 API 另外一个主要的用处就是实现了一个测试库，而这个测试库仅仅只是位于其他电脑或者其他 JVM 的真实测试库的代理。这种代理测试库可以非常瘦，因为关键字的名称都是动态获取的，而且当有新的关键字添加到真实测试库的时候，并不需要更新这个代理测试库。

本节将阐述动态 API 是如何在 Robot Framework 和动态测试库之间工作的。对于 Robot Framework 而言，这些测试库具体是怎么实现的并不重要。然而，如果你使用 Java，在开始实现你自己的库的时候，你可能需要研究一下 JavaTools，这些可重用的 tools 集合支撑多种方式创建关键字，它提供了一种机制来适合您的需要。。

获取关键字的名称

动态库通过 get_keyword_names 方法来告诉我们它们实现了什么关键字。这个方法在 Java 中的方法名推荐为 getKeywordNames。这个方法不需要任何参数，它必须返回一个字符串的列表（在 Python 中）或者字符串数组（在 Java 中），返回值中包含了测试库实现的所有关键字的名称。

如果返回的关键字名称中包括了多个单词，它们可以用空格或者下划线来间隔开，或者使用骆驼拼写法，比如['first keyword', 'second keyword'], ['first_keyword', 'second_keyword'], 和 ['firstKeyword', 'secondKeyword'] 都将代表关键字 *First Keyword* 和 *Second Keyword*。

> 译者注：
>
> 骆驼拼写法就是每个单词首字母大写，其他字母小写的命名方法。

动态库必须要有这个方法。如果没有，或者因为某种原因调用该方法失败，这个测试库就会被视为静态库。

运行关键字

动态库用一个特殊的方法 run_keyword(也可以用 runKeyword)来运行它们的关键字。当一个来自于动态库的关键字在测试数据中被使用，Robot Framework 就用该测试库的 run_keyword 方法来让这个关键字运行。这个方法带有 2 个参数，第一个参数是需要运行的关键字名称，其格式必须与 get_keyword_names 中返回的值相同，第二个参数是将要传到该关键字的参数列表（在 Java 中是一个对象数组）。

当测试库已经取得关键字名称和它的参数后，它就能自由地运行这个关键字了，但是它必须使用同样的体制跟 Robot Framework 通信。这就意味着，它要用异常来报告关键字状态，通过写标准输出来打日志，在 run_keyword 中用 return 语句来返回值。

每个动态库必须同时具有 get_keyword_names 和 run_keyword 方法。在动态 API 中的其他方法都是可选的，如下：

```
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)
```

获取关键字的参数

如果一个动态库只是实现了 get_keyword_names 和 run_keyword 方法，Robot Framework 还不能获得关键字的参数信息。

通过 get_keyword_arguments 方法(也可以用 getKeywordArguments)，动态库能告诉 Robot Framework 它实现的关键字实际需要什么参数。这个方法将关键字的名称作为一个参数，并返回一个包含了该关键字接受的参数的字符串列表（在 Java 中是字符串数组）。

跟静态关键字类似，动态关键字可以要求任何数量的参数，也可以带具有默认值的参数，可以接受可变数量的参数。

Representing different arguments with `get_keyword_arguments`

Expected arguments	How to represent	Examples	Min / Max
No arguments	Empty list.	`[]`	0/0
One or more argument	List of strings containing argument names.	`['one_argument']`, `['a1', 'a2', 'a3']`	1/1, 3/3
Default values for arguments	Default values separated from names with `=`. Default values are always considered to be strings.	`['arg=default value']`, `['a', 'b=1', 'c=2']`	0/1, 1/3

Representing different arguments with `get_keyword_arguments`

Expected arguments	How to represent	Examples	Min / Max
Variable number of arguments	Last argument has `*` before its name.	`['*arguments']`, `['a', 'b=42', '*rest']`	0/any, 1/any

当 get_keyword_arguments 被使用的时候，Robot Framework 自动计算关键字需要多少个参数。如果一个关键字被调用时带了非法数量的参数，在需要调用 run_keyword 方法之前就已经出现错误，且 run_keyword 不会再被调用。如上图所示的表格最后一列中，就计算出了最少和最多参数个数。

当测试运行的时候，实际参数的名称并不重要，因为 Robot Framework 只关心参数的个数。另外一方面，如果在为库添加文档的时候使用了 libdoc.py 工具，则参数会被显示在文档中，在这时，参数就需要有意义的名称。

获取关键字的文档

动态库可能需要实现的最后一个特殊的方法是 get_keyword_documentation(也可以用 getKeywordDocumentation)。它带有一个参数，该参数是关键字的名称，并以字符串形式返回它的文档。

使用返回的文档与使用以 Python 实现的静态库的关键字文档字符串类似。最主要的用法就是取得关键字的文档，以放入用 libdoc.py 生成的库文档中。另外，文档的第一行（以第一个\n 结尾）会被显示在测试日志中。

小结

动态 API 中所有特殊的方法如下表格所示。这里以下划线格式显示，它们的骆驼拼写法也同样可以正确地工作。

All special methods in the dynamic API

Name	Arguments	Purpose
get_keyword_names		To return names of implemented keywords.
run_keyword	name, arguments	To execute the specified keyword with given arguments.

All special methods in the dynamic API

Name	Arguments	Purpose
get_keyword_arguments	name	To return keyword's argument specification. Optional.
get_keyword_documentation	name	To return keyword's documentation. Optional.

如下所示，在 Java 中，可以写成 interface 形式。然而，注意，测试库并不需要强制实现任何接口，因为 Robot Framework 直接通过反射判断测试库是否具有必须的 get_keyword_names 和 run_keyword 方法。另外，get_keyword_arguments 和 get_keyword_documentation 方法完全是可选的。

```
public interface RobotFrameworkDynamicAPI {

    String[] getKeywordNames();

    Object runKeyword(String name, Object[] arguments);

    String[] getKeywordArguments(String name);

    String getKeywordDocumentation(String name);

}
```

使用动态 API 最好的例子就是 Robot Framework 的 Remote 库。

4.1.7 混合库 API

混合库 API，顾名思义，它介于静态和动态 API 之间。就跟动态 API 一样，实现一个测试库，只需要一个使用混合 API 的类。

获取关键字的名称

关键字的名称的获取跟动态 API 完全一样，测试库需要有 get_keyword_names 或者 getKeywordNames 方法来返回一个它所实现的关键字的名称列表。

运行关键字

在混合 API 中没有 run_keyword 方法来执行关键字，而 Robot Framework 采用反射来发现实现关键字的方法，这点跟静态 API 类似。一个使用了混合 API 的库，既能直接实现一些方法，更重要的是，还能够动态的处理它们。

在 Python 中，通过__getattr__方法可以很容易的处理不存在的方法。这个特殊的方法对于大多数 Python 程序员来说可能非常熟悉，他们能马上理解下面的例子。其他不熟悉的人可以先参考 Python Reference Manual。

```
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)
```

注意，__getattr__并没有像动态 API 中的 run_keyword 一样执行实际的关键字，它只是返回了一个可供调用的对象，而这个对象才会在后面被 Robot Framework 所执行。

另外值得注意的一点是，Robot Framework 用相同的名字（从 get_keyword_names 中返回的）来发现实现它们的方法。因此，这个类自身中实现的方法的名称必须以它们被定义的名称完全一样的格式返回。比如，上面的例子中，如果 get_keyword_names 返回 My Keyword（代替 my_keyword），则该库就不能正确地工作了。

混合 API 对于 Java 来说并不是非常有用，因为 Java 不能处理不存在的方法。当然，你可以在库的类中将方法全部实现，但是这样的话跟静态 API 比较来说，并不能带来什么好处。

获取关键字的参数和文档

当混合 API 被使用的时候，Robot Framework 使用反射来发现实现关键字的方法，就跟静态 API 一样。跟静态 API 的方式一样，取得这个方法的引用之后，它就从引用中去搜索参数和文档信息。因此，这里并不需要像动态 API 那样有一个特殊的方法来取得参数和文档。

小结

当在 Python 中实现一个测试库的时候，混合 API 具有和动态 API 一样的动态特性。混合 API 一个很大的好处就是，它不需要特殊的方法来取得关键字的参数和文档。真正动态的关键字在__getattr__中处理，其他的

就直接在主库类中实现，这在实践中也经常用到。

因为上面的好处和同样的特性，当使用 Python 的时候，混合 API 在大多数情况下是更好的选择。One notable exception is implementing a library as a proxy for an actual library implementation elsewhere, because then the actual keyword must be executed elsewhere and the proxy can only pass forward the keyword name and arguments.

使用混合 API 的好例子就是 Robot Framework 自带的 Telnet 库。

4.1.8 使用 Robot Framework 内部模块

Python 实现的测试库可以使用 Robot Framework 的内部模块，比如，可以获取已经执行了的测试用例及其配置信息。这种强大的机制要小心使用，因为 Robot Framework 的 API 并不是都能从外部调用的，而且在不同的 Framework 版本之间这些 API 有些本质上的变化。

最安全的 API 就是实现 BuiltIn 库中关键字的那些方法。关键字的改变要非常谨慎，一般首先就要把以前的老用法废弃掉。最有用的方式之一就是 replace_variables，这个方法允许访问当前可用的变量。下面的例子展示了怎么取得 ${OUTPUT_DIR}，它是众多自动变量的其中一个。使用 set_test_variable, set_suite_variable 和 set_global_variable 可以设置新的变量。

```
import os.path
from robot.libraries.BuiltIn import BuiltIn

def do_something(argument):
    output = do_something_that_creates_a_lot_of_output(argument)
    outputdir = BuiltIn().replace_variables('${OUTPUTDIR}')
    path = os.path.join(outputdir, 'results.txt')
    f = open(path, 'w')
    f.write(output)
    f.close()
    print '*HTML* Output written to <a href="results.txt">results.txt</a>'
```

The only catch with using methods from BuiltIn is that all run_keyword method variants must be handled specially. Methods that use run_keyword methods have to be registered as *run keywords* themselves using register_run_keyword method in BuiltIn module. This method's documentation explains why this needs to be done and obviously also how to do it.

The plan is to document all the internal modules better so that it is easier to decide which can be used and how they should be used. If you are unsure is using some API safe, please send a question to either user or developer mailing list.

4.1.9 扩展已存在的测试库

本节将描述一些不同的方法来添加新的功能给已存在的测试库，以及怎么在你自己的库中使用它们。

修改原始代码

如果你能访问到你想扩展的库的源代码，你自然就能直接修改其源代码。这种方式最大的问题就是，当你升级原始库时要想不影响你做过的改变是非常困难的。对用户来说，使用一个跟原始库相比有不同的功能的测试库很容易产生混乱。另外，重新打包也是一个额外的很大的任务。

如果改变的内容是普通的，并且你打算将它们提交给原始开发者，那么这种方式也可以。如果你的改变应用到原始库中，它们将被包含在未来的发布中，那么上述讨论的所有问题都会得到减轻。如果改变是不一般的，或者你因为某种原因不能提交它们，那么后面章节要描述的方式可能会更好。

继承

另外一个简单的方式就是使用继承来扩展一个已存在的库。下面的例子展示的是在 SeleniumLibrary 中添加新关键字 Title Should Start With。例子中使用的是 Python，但是你显然可以在 Java 代码中使用同样的方式来扩展一个已存在的 Java 库。

```
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))
```

相对于修改原始库，这种方式最大的不同是新的测试库与原始的名称不相同。一个好处就是你能很容易地告诉你正在使用一个定制库，但是最大的问题是你不能很容易地将新库和原始库一起使用。首先，你的新库将和原始库拥有同样的关键字，这就是意味着会有冲突。另外一个问题是测试库不能共享它们的状态。

当你开始使用新库，并且从一开始就想要添加定制的增强功能给它的时候，这种方式能很好的工作。否则，

本章描述的其他机制可能更好。

直接使用其他测试库

因为测试库都是一些类和模块，使用另外的测试库的一个简单方法就是引入它，并使用它的方法。当这些方法是静态的，也不依赖测试库的状态的时候，这种方式就能很好的工作，这已经被上面那个使用 Robot Framework 的 BuiltIn 测试库的例子阐明了。

如果测试库具有状态，那么，也许就不能像你希望的那样运行了。你在自己的测试库中使用的这个库实例将与 Robot Framework 使用的不一样，因此执行关键字产生的改变对于你的测试库来说并不可用。下一节就会阐述如何访问 Robot Framework 正在使用的库实例。

从 Robot Framework 中获取活动的测试库实例

Robot Framework 2.5.2 添加一个新的 BuiltIn 关键字 Get Library Instance，它被用来从 Robot Framework 自身中获取当前活动的库实例。这个关键字返回的库实例，跟 Robot Framework 自己使用的完全一样，因此，这个实例能看见当前的库状态。虽然这个功能是作为一个关键字来提供的，但是它也能在其他测试库中通过直接引入 BuiltIn 库的类来使用（就像之前讨论的一样）。下面的例子展示了如何实现跟之前使用继承的例子中实现的 Title Should Start With 相同的关键字。

```
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))
```

这种方式相对于继承来说，最大的好处就是，你既能够正常的使用原来的测试库，也能在需要的时候使用新的测试库。如下例子：

Using library and another library that extends it

Settings	Value	Value	Value
Library	SeleniumLibrary		

Using library and another library that extends it

Settings	Value	Value	Value
Library	SeLibExtensions		

Test Case	Action	Argument	Argument
Example	Open Browser	http://example	# SeleniumLibrary
	Title Should Start With	Example	# SeLibExtensions

使用动态库或者混合库 API 的测试库

那些使用动态或者混合库 API 的测试库通常都有一套它们自己的扩展方式。对于这些测试库，你需要从库的开发者，或者参考库文档或源代码中获取相关指南。

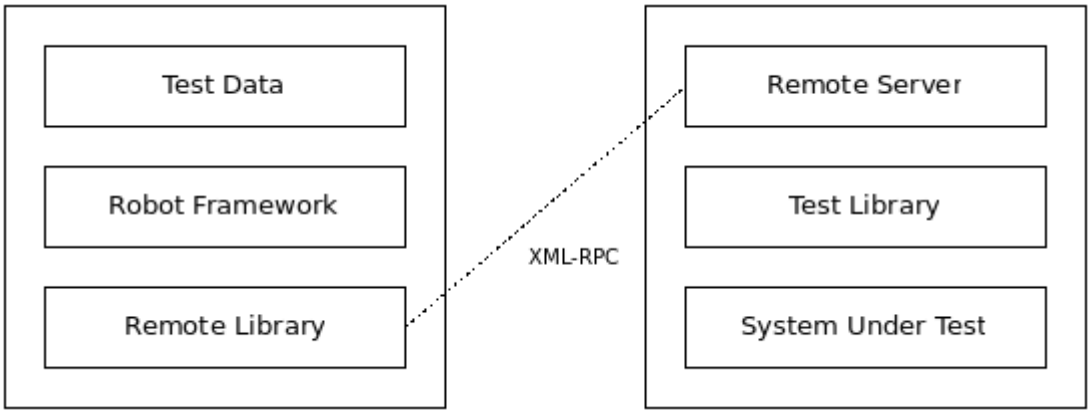
4.2 远程库接口

4.2.1 介绍

使用 remote 库 API 有两个主要的原因：

- 实际的测试库跟正在运行的 Robot Framework 不在同一台机器上。这样就产生了分布式测试的可能(This allows interesting possibilities for distributed testing)。
- 测试库可以用任何支持 XML-RPC 协议的语言来实现。Robot Framework 2.1 包含了针对 Python/Jython 和 Ruby 的通用 remote servers ， 未来还将实现针对其他语言比如 Java 和 Perl 的 generic servers。

从 Robot Framework 2.1 开始，Remote 库接口是被标准库之一的 Remote 库提供的，该库没有自己的任何关键字，但它是作为 Robot Framework 核心与在其他地方实现的关键字之间的一个代理方式来工作的。Remote 库通过 remote servers 与实际的库进行交互，Remote 库和服务器之间使用在 XML-RPC 通道上的简单 remote 协议来进行通信。所有这些高层的体系结构正如下面的图片所示：



Robot Framework architecture with Remote library

4.2.2 使用远程库

引入远程库

Remote 库需要知道远程服务器的地址，但是引入它和使用它提供的关键字跟其他测试库并没有什么区别。如果你需要在一个测试集中多次使用 Remote 库，或者只是想给它一个更容易理解的名字，你可以使用 WITH NAME 语法。

Importing Remote library

Setting	Value	Value	Value	Value
Library	Remote	http://localhost:8270	WITH NAME	Example1
Library	Remote	http://10.0.0.42:7777	WITH NAME	Example2

上面第一个例子的 URL 就是 Remote 库使用的默认地址。同样地，8270 端口也是远程服务器使用的默认端口（82 和 70 分别是 R 和 F 字母的 ASCII 码）。

启动或停止远程 Server

在 Remote 库被引入之前，提供实际关键字的远程服务器必须被启动。只有服务器在测试运行之前被启动，才能像上面例子中使用 Library 设置。另外，其他关键字，比如来自 OperatingSystem 或者 SSH 库，也能启动服务器，但是之后你可能需要使用 Import Library 关键字，因为这个库在测试执行开始的时候还不可用。

一个远程服务器如何停止取决于它是怎么实现的。按照下面的方法，可以跟 Robot Framework 提供的 Server 进行工作：

- 不管你使用哪个测试库，远程服务器提供 Stop Remote Server 关键字，这个关键字可以直接在测试脚本中使用。
- 远程服务器在它们的 XML-RPC 接口中，提供了 stop_remote_server 方法。
- 如果服务器正在一个终端窗口中运行，则可以使用 Ctrl-C 来停止 Server。不幸的是，并不是所有操作系统上的所有 server 都能如此。
- Server 进程可以被操作系统提供的工具来中止（比如 kill）

4.2.3 支持的参数和返回值的类型

因为 XML-RPC 协议不支持所有的对象类型，所以远程库与远程服务器之间传输的值必须转换成统一的类型。这一点对于远程库传给远程服务器的参数和远程服务器返回给远程库的值都适用。

转换采用下面的规则：

- 字符串、数字和布尔值不需要转换
- None、nil 值需要转换成空字符串
- 列表和元组的内容被递归地转换，仍然以列表来传输
- Dictionary 和 Map，它们的值被递归地转换成支持的类型，它们的键被转换成字符串
- 其他类型被转换成字符串

4.2.4 使用远程 Server

Robot Framework 2.1 包含了用 Python 和 Ruby 编写的远程服务器。这些服务器与后面展示的测试库例子、以及一个测试用例文件，都被包含在源代码发布中，在 tools/remoteserver 目录下，也可以参考

http://code.google.com/p/robotframework/wiki/RemoteLibrary

使用提供的服务器能够非常容易地创建测试库。下面是使用服务器的基本步骤：

- 用静态库 API 像创建普通测试库一样创建一个测试库模块或类。如果用 Python 服务器，也可以使用混合库 API。
- 引入远程服务器类，并创建一个它的实例，将测试库实例或者模块传递给它作为参数。监听地址

和端口，可以从命令行中获得。

以上步骤能够在同一个模块中完成，如下例子所示。从命令行将这些模块当作脚本来执行，就会启动远程服务器，这个服务器就会提供库中实现的关键字服务了。

Python 远程库例子

下面的例子展示了如何使用 Python 版的远程服务器。这个例子库实现了 Count Items In Directory 和 Strings Should Be Equal 关键字。

```
#!/usr/bin/env python

import os
import sys


class ExampleRemoteLibrary:

    def count_items_in_directory(self, path):
        return len([ i for i in os.listdir(path) if not i.startswith('.') ])

    def strings_should_be_equal(self, str1, str2):
        print "Comparing '%s' to '%s'" % (str1, str2)
        if str1 != str2:
            raise AssertionError("Given strings are not equal")


if __name__ == '__main__':
    from robotremoteserver import RobotRemoteServer
    RobotRemoteServer(ExampleRemoteLibrary(), *sys.argv[1:])
```

Ruby 远程库例子

下面的例子跟上面的例子实现同样的关键字，不过这次使用的是 Ruby 版的远程服务器。

```
#!/usr/bin/env ruby

class ExampleRemoteLibrary

  def count_items_in_directory(path)
    Dir.entries(path).find_all{|i| not i.match('^\.')}.length
  end
```

```
  def strings_should_be_equal(str1, str2)
    puts "Comparing '#{str1}' to '#{str2}'"
    if str1 != str2
      raise RuntimeError, "Given strings are not equal"
    end
  end

end


if __FILE__ == $0
  require "robotremoteserver"
  RobotRemoteServer.new(ExampleRemoteLibrary.new, *ARGV)
end
```

4.2.5 Remote 协议

本节描述在远程库与远程服务器之间使用的协议，本节内容主要是针对那些想要创建远程服务器的人。

该远程协议被 XML-RPC（通过 HTTP 使用 XML 进行远程过程调用的协议）所实现。大多数主流的语言（Python、Java、C、Ruby、Perl、JavaScript、PHP、.....）都提供 XML-RPC 支持。

需要的方法

一个远程服务器就是一个 XML-RPC 服务器，它必须在它的公共接口中提供跟动态库 API 所具有的相同的方法。虽然只有 get_keyword_names 和 run_keyword 是实际需要的，但是建议也提供 get_keyword_arguments 和 get_keyword_documentation。注意一点，方法名称使用骆驼拼写法是不行的。实际的关键字是如何实现的，对于远程库而言是没有关系的。一个远程服务器既可以作为真实测试库的包装器（就像 Robot 提供的 Python 和 Ruby 远程服务器一样），也可以自己实现关键字。

另外，远程服务器在它们的公共接口中应该提供一个 stop_remote_server 方法，以便于可以很容易地停止它们。它们也应该自动地将这个方法暴露为 Stop Remote Server 关键字，以便于它能在测试用例中进行使用。

获取远程关键字名称和其他信息

远程库可以通过 get_keyword_names 方法取得远程服务器提供的关键字列表。这个方法必须返回关键字名称的字符串列表。

远程服务器可以也应该实现 get_keyword_arguments 和 get_keyword_documentation 方法，用来提供关于关键字更多的信息。这两个方法都以关键字的名称作为参数。就跟动态库一样，get_keyword_arguments 返回的必须是字符串列表的参数， get_keyword_documentation 返回的是字符串文档。

执行远程关键字

当远程库想要服务器执行某个关键字的时候，它就调用远程服务器的 run_keyword 方法，并将关键字的名称和参数列表传过去。基本类型可以直接被用作参数，但是复杂类型就需要转换为支持的类型。远程服务器必须将执行结果返回为一个结果 dictionary 或者 map，内容包含如下表中所示的元素。

Entries in the remote result dictionary

Name	Explanation
status	Mandatory execution status. Either PASS or FAIL.
output	Possible output to write into the log file. Must be given as a single string but can contain multiple messages and different log levels in format *INFO* First message\n*INFO* Second\n*WARN* Another message.
return	Possible return value. Must be one of the supported types.
error	Possible error message. Used only when the execution fails.
traceback	Possible stack trace to write into the log file using DEBUG level when the execution fails.

4.3 使用监听器接口

Robot Framework 有一个监听器接口，可以用来接收关于测试执行的通知。监听器是带有一些特殊方法的类，它们既可以用 Python 也可以用 Java 实现。从 2.0.2 版本开始，它也可以将监听器做为 Python Module 来实现。监听器接口的常见用法包括外部的测试监控、当一个测试失败的时候发送一条邮件信息、跟其他系统沟通。

4.3.1 使用监听器

监听器可以通过命令行选项--listener 来使用,监视器的名称就作为这个选项的参数。监听器名称从实现这个监听器接口的类或者模块的名称中获取，就跟测试库名称是从它的实现类中获取的一样。当这些特殊的监听器被引入的时候，它们必须放在与搜索测试库时相同的模块搜索路径上。另外一个选择是，也跟测试库一样，取得监听器文件的绝对或者相对路径，使用这种方式可以多次引入多个监听器。

从 2.0.2 版本开始，可以从命令行给监听器类提供参数。参数定义在监听器名称或者路径后面，以冒号作为分隔符。这种方式仅适用于类型是字符串型的参数，并且参数中不能包含冒号。

Examples:

```
pybot --listener MyListener tests.html
jybot --listener com.company.package.Listener tests.html
pybot --listener path/to/MyListener.py tests.html
pybot --listener module.Listener --listener AnotherListener tests.html
pybot --listener ListenerWithArgs:arg1:arg2
pybot --listener path/to/MyListener.java:argument tests.html
```

4.3.2 可用的监听器接口方法

当测试执行开始的时候，Robot Framework 通过传递过来的参数为监听器类创建一个实例。在测试执行期间，Robot Framework 在测试集、测试用例、关键字开始和结束的时候调用监听器的方法。当输出文件准备好了之后，它也调用适当的方法，最后在结束时，它调用 close 方法。一个监听器没有必要实现任何官方接口，它只要实现它自己实际需要的方法即可。

监听器接口版本

在 Robot Framework 2.1 版本，方法签名跟测试执行过程有关这种状况已经改变了。这个改变使得可以添加新的信息给监听器接口，而不会打断已存在的监听器。老的签名会继续工作，但是它们在将来的版本中就会不被推荐了，所以所有新的监听器都应该带上后面表格中描述的签名。老的监听器接口的最近描述可以在 Robot Framework 2.0.4 的用户指南中找到。

【注意】一个监听器必须有 ROBOT_LISTENER_API_VERSION 属性，这样才能被识别为新样式的监听器。ROBOT_LISTENER_API_VERSION 属性的值必须是 2，字符串和数字都行。下面的例子就是以新样式来实现的。

监听器接口方法签名

从 Robot Framework 2.1 开始，所有与测试执行过程相关的方法都有了相同的签名 method(name,attributes)，这里的 attributes 是一个包含具体事件的 dictionary。下面的表中列出了在监听器接口中所有可用的方法和 attributes dictionary 中的内容。Dictionary 的键是字符串。所有这些方法也拥有骆驼样式的别名，比如，startSuite 跟 start_suite 就是相同的。

Available methods in the listener interface

Method	Arguments	Attributes / Explanation
start_suite	name, attributes	Keys in the attributes dictionary: • doc: test suite documentation • longname: suite name including parent suites • starttime: execution start time • metadata: a dictionary containing free test suite metadata • tests: names of tests directly in this suite as a list of strings • suites: names of suites directly in this suite as a list of strings • totaltests: total number of tests in this suite and all its sub-suites as an integer The last four were added in Robot Framework 2.5
end_suite	name, attributes	Keys in the attributes dictionary: • doc: test suite documentation • longname: test suite name including parents • starttime: execution start time • endtime: execution end time • elapsetime: execution time in milliseconds as an integer • status: either `PASS` or `FAIL` • statistics: suite statistics (number of passed and failed tests in the suite) as a string • message: error message, if the suite setup or teardown has failed, empty otherwise
start_test	name, attributes	Keys in the attributes dictionary: • doc: test case documentation • longname: test name including parent suites • starttime: execution start time • tags: test case tags as a list of strings
end_test	name, attributes	Keys in the attributes dictionary: • doc: test case documentation • longname: test name including parent suites • starttime: execution start time • endtime: execution end time

Available methods in the listener interface

Method	Arguments	Attributes / Explanation
		• elapsetime: execution time in milliseconds as an integer • tags: test case tags as a list of strings • status: either `PASS` or `FAIL` • message: contains an error message if the test has failed and is empty otherwise
start_keyword	name, attributes	Keys in the attributes dictionary: • doc: keyword documentation • starttime: execution start time • args: keyword arguments as a list of strings
end_keyword	name, attributes	Keys in the attributes dictionary: • doc: keyword documentation • starttime: execution start time • endtime: execution end time • elapsetime: execution time in milliseconds as an integer • args: keyword's arguments as a list of strings • status: either `PASS` or `FAIL`
log_message	message	Called when an executed keyword writes a log message. `message` is a dictionary with following keys: • message: the content of the message • level: log level used in logging the message • timestamp: message creation time, format is YYYY-MM-DD hh:mm:ss.mil • html: string `yes` or `no` denoting whether the message should be interpreted as HTML or not New in Robot Framework 2.1.3.
message	message	Called when the framework itself writes a `syslog` message. `message` is a dictionary with same keys as with `log_message`method. New in Robot Framework 2.1.3.
output_file	path	Called when writing to an output file is finished. The path is an absolute path to the file. When `outputs are split`, called for each finished file.
log_file	path	Called when writing to a log file is finished. The path is an absolute path to the file. When `outputs are split`, called for each finished file.

Available methods in the listener interface

Method	Arguments	Attributes / Explanation
report_file	path	Called when writing to a report file is finished. The path is an absolute path to the file.
summary_file	path	Called when writing to a summary file is finished. The path is an absolute path to the file.
debug_file	path	Called when writing to a debug file is finished. The path is an absolute path to the file.
close		Called after all test suites, and test cases in them, have been executed.

可用的方法和它们的参数也可以如下方式在 Java 接口中定义。java.util.Map 属性的内容跟上面的表格描述的一样。值得注意的是，一个监听器并不一定要实现所有的接口。

```
public interface RobotListenerInterface {
    public static final int ROBOT_LISTENER_API_VERSION = 2;
    void startSuite(String name, java.util.Map attributes);
    void endSuite(String name, java.util.Map attributes);
    void startTest(String name, java.util.Map attributes);
    void endTest(String name, java.util.Map attributes);
    void startKeyword(String name, java.util.Map attributes);
    void endKeyword(String name, java.util.Map attributes);
    void logMessage(java.util.Map message);
    void message(java.util.Map message);
    void outputFile(String path);
    void logFile(String path);
    void reportFile(String path);
    void summaryFile(String path);
    void debugFile(String path);
    void close();
}
```

4.3.3 监听器例子

第一个简单的例子是用 Python Module 实现的。它主要展示了使用监听器接口并不复杂。

```
ROBOT_LISTENER_API_VERSION = 2

def start_test(name, attrs):
    print 'Executing test %s' % name

def start_keyword(name, attrs):
```

```
    print 'Executing keyword %s with arguments %s' % (name, attrs['args'])

def log_file(path):
    print 'Test log available at %s' % path

def close():
    print 'All tests executed'
```

第二个例子，仍然使用 Python，稍微有点复杂。它将它获得的所有信息写入一个临时目录中一个文本文件中。文件名称可以从命令行传入，也可以有默认值。注意在真实的使用过程中，debug file 功能可以通过命令行选项--debugfile 来使用，可能比这个例子更有用一些。

```
import os.path
import tempfile


class PythonListener:

    ROBOT_LISTENER_API_VERSION = 2

    def __init__(self, filename='listen.txt'):
        outpath = os.path.join(tempfile.gettempdir(), filename)
        self.outfile = open(outpath, 'w')

    def start_suite(self, name, attrs):
        self.outfile.write("%s '%s'\n" % (name, attrs['doc']))

    def start_test(self, name, attrs):
        tags = ' '.join(attrs['tags'])
        self.outfile.write("- %s '%s' [ %s ] :: " % (name, attrs['doc'], tags))

    def end_test(self, name, attrs):
        if attrs['status'] == 'PASS':
            self.outfile.write('PASS\n')
        else:
            self.outfile.write('FAIL: %s\n' % attrs['message'])

     def end_suite(self, name, attrs):
         self.outfile.write('%s\n%s\n' % (attrs['status'], attrs['message']))

     def close(self):
         self.outfile.close()
```

第三个例子，实现了跟前一个例子相同的功能，不过使用的是 Java。

```
import java.io.*;
import java.util.Map;
import java.util.List;


public class JavaListener {

    public static final int ROBOT_LISTENER_API_VERSION = 2;
    public static final String DEFAULT_FILENAME = "listen_java.txt";
    private BufferedWriter outfile = null;

    public JavaListener() throws IOException {
        this(DEFAULT_FILENAME);
    }

    public JavaListener(String filename) throws IOException {
        String tmpdir = System.getProperty("java.io.tmpdir");
        String sep = System.getProperty("file.separator");
        String outpath = tmpdir + sep + filename;
        outfile = new BufferedWriter(new FileWriter(outpath));
    }

    public void startSuite(String name, Map attrs) throws IOException {
        outfile.write(name + " '" + attrs.get("doc") + "'\n");
    }

    public void startTest(String name, Map attrs) throws IOException {
        outfile.write("- " + name + " '" + attrs.get("doc") + "' [ ");
        List tags = (List)attrs.get("tags");
        for (int i=0; i < tags.size(); i++) {
           outfile.write(tags.get(i) + " ");
        }
        outfile.write(" ] :: ");
    }

    public void endTest(String name, Map attrs) throws IOException {
        String status = attrs.get("status").toString();
        if (status.equals("PASS")) {
            outfile.write("PASS\n");
        }
        else {
            outfile.write("FAIL: " + attrs.get("message") + "\n");
        }
    }
```

```
    public void endSuite(String name, Map attrs) throws IOException {
        outfile.write(attrs.get("status") + "\n" + attrs.get("message") + "\n");
    }

    public void close() throws IOException {
        outfile.close();
    }

}
```

4.4 使用内部 API

Robot Framework 有一些公共的 API，用来在开发支持工具中提供帮助，或者扩展输入输出数据的过程。这些 API 用 Python Module 实现，因此它们只能被 Python 和 Jython 脚本或程序使用。

不幸的是，这些 API 的文档并不具有什么好的文档和注释，所有目前来说要想获得更多的信息，只能从 Robot Framework 开发者那里获取了，或者查看源代码。

4.4.1 运行测试数据

这个 API 包含了一个工厂方法，将一个 Robot Framework 输出文件读取到 TestSuite 对象，该对象包含了所有有关测试执行的结果的信息。方法的签名是 TestSuite(outpath)，其中 outpath 是指向一个已存在的输出文件的路径。返回的 TestSuite 对象可以用来处理测试运行的结果。

这里有个例子，它读取了一个传入的输出文件，并将每一个执行时间超过 3 分钟的测试用例标识为 Failed。然后，这个 TestSuite 对象被序列化，日志和报告文件可以通过 rebot 来生成。

```
#!/usr/bin/env python

"""Usage: check_test_times.py inpath [outpath]

Reads result of a test run from Robot output file and checks that no test
took longer than 3 minutest to execute. If outpath is not given, the
result is written over the original file.
"""

import sys
from robot.output import TestSuite
```

```
def check_tests(inpath, outpath=None):
    if not outpath:
        outpath = inpath
    suite = TestSuite(inpath)
    _check_execution_times(suite)
    suite.write_to_file(outpath)

def _check_execution_times(suite):
    for test in suite.tests:
        if test.status == 'PASS' and test.elapsedtime > 1000 * 60 * 3:
            test.status = 'FAIL'
            test.message = 'Test execution time was too long: %s' % test.elapsedtime
    for suite in suite.suites:
        _check_execution_times(suite)


if __name__ == '__main__':
    try:
        check_tests(*sys.argv[1:])
    except TypeError:
        print __doc__
```

4.4.2 测试执行

这个 API 包含了 run 方法，该方法用来启动测试执行。这个方法的签名为 run(*datasources,**options)，其中 datasources 是指向被执行的文件和目录的路径，options 的内容跟命令行内容相同，只是格式不同。

```
from robot import run

run('tests.html', log='mylog.html', include=['tag1', 'tag2'])
```

相同效果的命令行输入为：

```
pybot --log mylog.html --include tag1 --include tag2 tests.html
```

【警告】这个方法在给定的上下文中只能使用一次。这个问题会在未来的版本中解决。

4.4.3 解析测试数据

robot.parsing 包中包含了解析和处理测试数据的工具。TestCaseFile 和 TestDataDirectory 类可以命名参数 source 来初始化，并开始解析已存在的测试数据。下面的例子展示了如何从一个测试文件中解析测试用例的名称。更多的细节请直接查看 Robot Framework 的源代码。

```
from robot.parsing import TestCaseFile

suite = TestCaseFile(source='path/to/tests.html')
print 'Suite: ', suite.name
for test in suite.testcase_table:
    print test.name
```

4.4.4 可运行的测试数据

这个 API 包含了一个工厂方法，用来解析给定的输入文件为一个可运行的 TestSuite 对象。这个 robot.runing.TestSuite 方法的签名为 TestSuite(datasources, settings)，datasources 是指向从命令行执行测试的文件和目录。

4.4.5 配置日志

Robot Framework 有一个全局的日志器，它负责处理错误的报告和平时的日志输出。这个 LOGGER 实例是一个代理，可以注册新的 logger 给它。当一些事件发生的时候，注册的 logger 能得到通知，它们就报告它们想要报告的事件。一个控制台 logger 被自动注册，它将警告和错误写入标准错误流中，但它可以禁用。更多细节请查看 robot.output.LOGGER 的源代码和文档。

4.5 通过 Java 使用 Robot Framework

从 Robot Framework 2.5.2 开始，Robot Framework 也作为单独的 Jar 文件来分发，它包含了 Jython 和 Robot Framework。只要安装了 Java，它就能从命令行执行，同时这个 Jar 文件也提供 JAVA 的 API。当前 API 文档可以在线看到。

4.5.1 通过 API 运行测试

下面这个简单的示例代码展示了 Robot Framework 如何从 Java 程序中启动测试执行。

```
import org.robotframework.RobotFramework;

public class Test {

    public void runTests() {
        int rc = RobotFramework.run(new String[] { "--outputdir", "/tmp", "mytests" });
        if (rc == 0)
            System.out.println("All tests passed");
        else if (rc <= 250)
```

```
            System.out.println(rc + " tests failed.");
        else
            System.out.println("Error occurred");
    }

}
```

RobotFramework.run 的返回值具有跟 Robot Framework 执行的返回码同样的含义。

5 附录

5.1 测试数据的变量设置

5.1.1 初始化设置表

初始化表用于为测试集和测试用例导入测试库、源文件、变量文件和定义元数据。初始化表可以包含在测试用例文件和测试源文件中。请注意，在源文件中，初始化表只包含导入库、源文件、变量的相关设置。

初始化设置表中的常用名称及含义

名称	作用
Library	用于引用测试库
Resource	用于引用源文件
Variables	用于引入环境变量
Documentation	用于注释测试集或测试用例
Metadata	用于设置测试集元数据
Suite Setup	用于为测试集指定 Setup
Suite Teardown	用于为测试集指定 Teardown
Suite Precondition	与 Suite Setup 的作用相同
Suite Postcondition	与 Suite Teardown 的作用相同
Force Tags	用于给整个测试集每个用例打上标签
Default Tags	每个测试用例的默认标签
Test Setup	用于为测试用例指定 Setup
Test Teardown	用于为测试用例指定 Teardown
Test Precondition	与 Test Setup 的作用相同
Test Postcondition	与 Test Teardown 的作用相同
Test Template	用于为测试用例设置一个默认关键字模板
Test Timeout	用于标识测试用例的默认超时时间

5.1.2 测试用例表

测试用例中的常用设置在测试用例表中设置，测试用例表的设置会屏蔽测试集用例表中的相应设置。

测试用例表中常用名称及含义

名称	作用
[Documentation]	用于注释测试用例的名称或作用
[Tags]	用于给测试用例打上标签
[Setup]	用于指定测试用例的 Setup
[Teardown]	用于指定测试用例的 Teardown
[Precondition]	与[Setup]的作用相同
[Postcondition]	与[Teardown]的作用相同
[Template]	用于标识一个关键字模板
[Timeout]	用于设定一个测试用例的超时时间

5.1.3 关键字表

关键字表用于设定关键所需要的参数等信息。

关键字表常用名称及含义

[Documentation]	用于注释关键字的名称或作用
[Arguments]	用于标识关键字的参数
[Return]	用于标识关键字的返回值
[Timeout]	用于标识关键字的超时时间

5.2 命令行选项

下面列出了用 pybot 和 jybot 执行测试用例和用 rebot 后处理输出结果时的所有命令行选项。

5.2.1 用例执行的命令行选项

-N,--name <name>

为顶级测试集设置名字

-D,--doc <document>

为顶级测试集添加注释

-M,--metadata <name:value>

为顶级测试集设置自由元数据

-G,--settag <tag>

为所有可执行测试用例设置标签

-t,--test <name>

通过名称选择测试用例

-s,--suite <name>

通过名称来选择测试集

-i,--include <tag>

通过标签来选择测试用例

-e,--exclude <tag>

通过标签来排除不执行的测试用例

-c, --critical <tag>

被打上相应标签的用例被认为是关键的用例

-n, --noncritical <tag>

被打上相应标签的用例被认为是非关键的用例

---runmode <mode>

为用例执行设置执行模式。有 ContinueOnFailure,ExitOnFailure,SkipTeardownOnExit,and Random:<what>.

-v, --variable <name:value>

设置有效变量

-V, --variablefile <path:args>

通过变量文件来设置变量

-d, --outputdir <dir>

指定创建输出文件的路径

-o, --output <file>

设置生成输出文件的路径

-l, --log <file>

设置生成日志文件的路径

-r, --report <file>

设置生成报告文件的路径

-S, --summary <file>

设置生成摘要文件的路径

-b, --debugfile <file>

用例执行的过程中写入 debug 文件

-T, --timestampoutputs

为所有的输出文件设置时间戳

---splitoutputs <title>

分离输出和日志文件

--logtitle <title>

为生成的日志文件设置标题

--reporttile <title>

为生成的报告文件设置标题

--reportbackground <colors>

为生成的报告和摘要文件设置背景色

--summarytitle <title>

为生成的摘要报告设置标题

-L, --loglevel <level>

为日志文件设置阈值级别

-suitestatlevel <level>

定义输出文件中统计出的用例级别

--tagstatinclude <tag>

在输出结果中仅统计被 Tag 和 Test Details by Tag 打上标签的用例

--tagstatexclude <tag>

在输出结果中排除被 Tag 和 Test Details by Tag 打上标签的用例

--tagstatcombine <tag>

创建基于标签的结果统计

--tagdoc <pattern:doc>

为特定的标签添加注释

--tagstatlink <pattern:link:tilte>

为输出结果的统计标签添加外部链接

--listener <name:args>

为监视用例执行设置监听器

--W, --monitorwidth <chars>

设置输出监视器的宽度

--C, --monitorcolors <on|off|force>

指定控制台设定的颜色

--P, --pythonpath <path>

添加搜索测试库的默认路径

--E, --escape <what:with>

执行遇到问题时的退出字符

--A, --argumentfile <path>

从一个文件中读出多个字符

--h, --help 打印用户帮助文档

--version 打印版本信息

5.2.2 后处理输出结果时的命令行选项

-N,--name <name>

为顶级测试集设置名字

-D,--doc <document>

为顶级测试集添加注释

-M,--metadata <name:value>

为顶级测试集设置自由元数据

-G,--settag <tag>

为所有可执行测试用例设置标签

-t,--test <name>

通过名称选择测试用例

-s,--suite <name>

通过名称来选择测试集

-i,--include <tag>

通过标签来选择测试用例

-e,--exclude <tag>

通过标签来排除不执行的测试用例

-c, --critical <tag>

被打上相应标签的用例被认为是关键的用例

-n, --noncritical <tag>

被打上相应标签的用例被认为是非关键的用例

-d, --outputdir <dir>

指定创建输出文件的路径

-o, --output <file>

设置生成输出文件的路径

-l, --log <file>

设置生成日志文件的路径

-r, --report <file>

设置生成报告文件的路径

-S, --summary <file>

设置生成摘要文件的路径

-T, --timestampoutputs

为所有的输出文件设置时间戳

---splitoutputs <title>

分离输出和日志文件

--logtitle <title>

为生成的日志文件设置标题

--reporttile <title>

为生成的报告文件设置标题

--reportbackground <colors>

为生成的报告和摘要文件设置背景色

--summarytitle <title>

为生成的摘要报告设置标题

-L, --loglevel <level>

为日志文件设置阈值级别

-suitestatlevel <level>

定义输出文件中统计出的用例级别

--tagstatinclude <tag>

在输出结果中仅统计被 Tag 和 Test Details by Tag 打上标签的用例

--tagstatexclude <tag>

在输出结果中排除被 Tag 和 Test Details by Tag 打上标签的用例

--tagstatcombine <tag>

创建基于标签的结果统计

--tagdoc <pattern:doc>

为特定的标签添加注释

--tagstatlink <pattern:link:tilte>

为输出结果的统计标签添加外部链接

--removekeywords <all|passwd>

从输出结果中删除关键字数据

--starttime <timestamp>

当创建联合报告时设置用例执行的开始时间

--endtime <timestamp>

当创建联合报告时设置用例执行的结束时间

--E, --escape <what:with>

执行遇到问题时的退出字符

--A, --argumentfile <path>

从一个文件中读出多个字符

--h, --help 打印用户帮助文档

--version 打印版本信息

5.3 测试数据模板

这些模板用于为 Robot Framework 创建测试数据。模板包括两种：测试用例模板和源文件模板，其中源文件模板同样可用于初始化测试集。模板有 HTML 和 TSV 两种，并且都可以随意地被自定义。

testcase_template.html

HTML 格式的测试用例文件模板

testcase_template.tsv

TSV 格式的测试用例文件模板

resource_template.html

HTML 格式的源文件模板

resource_template.tsv

在 ATDD 风格下创建测试由不用参数的高级关键字组成的用例的已经被简化的模板。

模板在整个用户模板都是可用的，它们是随着版本发布的，也可以在 http://code.google.com/p/robotframework/wiki/Templates 下载。

5.4 支持工具

Robot Framework 可用的工具如下：

5.4.1 内部工具

这些工具随着 Robot Framework 版本发布。它们被包含在版本包也可以从 http://code.google.com/p/robotframework/wiki/SupportingTools.获取。

Library documentation tool

libdoc.py 是一种由包含关键字的测试库或源文件生成 HTML 或 XML 文件的工具。

Test data documentation tool

testdoc.py 是一种由给定的测试集生成包含测试集名、高级关键字的高级文档描述的工具。

Historical reporting tool

risto.py 是一种由用例执行历史数据生成图形统计结果的工具。

test result diffing tool

robotdiff.py 是一种由 Robot Framework 输出文件生成不同报告的工具。

Execution time reporting tool

times2csv.py 是由包含测试集、测试用例、关键字的 CSV 格式文件生开始、结束和流逝时间信息的工具。

File viewing tool

fileviewer.py 是一种实现类似 UNIX tail 功能的图形化工具。它是专门为查看 debug files 而设计的。

One click installer

One click installer 是一种安装 Robot Framework 和它的依赖所用 AutoIT 脚本。

Test status checker tool

statuschecker.py 是确认用例状态、信息和关键字日志信息的工具。

5.4.2 扩展工具

这些工具是单独开发的。

Test data editing tool (RIDE)

RIDE 是独立的用编辑测试数据的工具。它可以用于创建、编辑、维护 Robot Framework 的测试数据。工程文件在 http://code.google.com/p/robotframework-ride/.

Manual test execution tool (mabot)

Mabot 是一种独立的生成手工执行用例结果报告的工具。它能够存储和生成通过自动化测试工具 Robot Framework 手工执行的报告。工程文件在 http://code.google.com/p/robotframework-mabot/.

Tools for creating Java test libraries

JavaTools 是一种简单创建提供动态关键字执行的大型 Java 测试库的工具集。工程文件在 http://code.google.com/p/robotframework-javatools/.

5.5 文档格式

在创建测试集、测试用例、关键字库以及编辑测试库的过程中我们能非常方便地使用 HTML 格式。这种格式很像用在 wikis 中的风格，它是为了理角普通文本和 HTML 文本而设计的。

5.5.1 换行

测试集，测试用例和关键字所用文件格式受正则表达式规则约束。也就是说，正常换行不保留，换行或分段时要用转义字符（\n）来表达，如下面的例子所示。

Setting	Value
Documentation	First line.\n \n Second paragraph, this time\n with multiple lines.

对库文档正常换行已经足够，例如以下关键字的文件会创建为上面的测试套件文档相同的最终结果。

```
def example_keyword():
    """First line.

    Second paragraph, this time
    with multiple lines.
    """
    pass
```

5.5.2 粗体和斜体

粗体文本可以通过把选中的文本前或后加一个星号，例如*this is bold*.斜体可以采用同样的方法来生成，但使用的特殊字符是下划线，例如_italic_。它也能用同时具有粗体和斜体的语法_ *bold italic* _.

一个单独星号或下划线或在一个单词的中间不能构成黑体或斜体格式，但在标点符号之前或之后，他们是被允许的。粗体和斜体格式在一行内是有限的，格式跨越多个行，必须明确每一行开始和结束处注明。

黑体、斜体例子:

Unformatted	Formatted
bold	**bold**
italic	*italic*
bold and then _italic_	**bold** and then *italic*
*bold italic*, _italic_, nothing	***bold italic***, *italic*, nothing
This is *bold*\n *on multiple*\n *lines*.	This is **bold** **on multiple** **lines**.

5.5.3 URLs

所有类似 URLsr 的字符串可以自动转换为可点击链接。此外，结尾的扩展名为 jpg、JPEG、png、gif 或 bmp 的 URLs 会自动创建图像。例如像 http://some.url 会自动转换链接，http:///server/image.jpg 和 file：//path/chart.png 会转换为图象。

URLs 自动转换为链接适用于所有的测试数据、日志和报告，但创建图像只在测试集、测试用例和关键字的文档起作用。创建图像是从 Robot Framework2.0.2 开始的。

5.5.4 表格

表是使用管道字符和其周围的空格字符组成单元格边界和换行符作为行分隔符构成的。在库文档正常换行符已经足够，但在测试集、测试用例和关键字文档转义字符（\n）是必要的：

```
| *A*  | *B*   | *C*   |\n
| _1_  | Hello | world |\n
| _2_  | Hi    |       |\n
```

创建的表经常边框较细、文本左对齐。表格单元格中的格式可以用粗体和斜体，因此可以创建表头。例如，上述表格可以创建成这样：

A	B	C
1	Hello	world
2	Hi	

5.5.5 水平标尺

水平尺（即<hr>标签），使得个各个部分之间保留较大的间隔称为可能。他们可以通过在单独一行使用三个连字符创建。测试数据中换行符是必要的：

```
Some text here.\n
\n
---\n
\n
More text...
```

5.6 时间格式

Robot Framework 有自己的时间格式，这些时间格式在很多关键字中被使用（例如 sleep 和 Wait Until Keyword Succeeds），同样测试用例和用户关键字的 timesouts 也使用这种时间格式。这种格式既灵活又易于理解。

5.6.1 数字格式的时间

时间经常用一个简单的数字表示，此时，它被解释为被秒。整型和浮点型数据都有效，也可以用实数和包含数字的字符串表示时间。这种格式很有用，例如，当实际时间被换算成秒时。

5.6.2 文本格式的时间

文本格式的时间是指使用的格式如“2 分 42 秒”，这是比仅仅以秒计算的方法好理解得多。也就是说，例如，“4200 秒”没有“1 小时 10 分钟”好理解。

这种格式的基本思想是用数字和文本内容来表示时间，文本内容说明这个数字代表什么时间。数字可以是整数或浮点数，整个格式是大小写敏感的，下面是可用的文本格式：

- days, day, d
- hours, hour, h
- minutes, minute, mins, min, m
- seconds, second, secs, sec, s
- milliseconds, millisecond, millis, ms

例子：

```
1 min 30 secs
1.5 minutes
90 s
1 day 2 hours 3 minutes 4 seconds 5 milliseconds
1d 2h 3m 4s 5ms
```