

---

# QT4C Documentation

发布 2.2.0

QTA

2022 年 03 月 25 日



---

## Contents

---

1 使用文档	3
2 索引和搜索	95
Python 模块索引	97
索引	99



QT4C (原名为 TUIA), 即 Quick Test for Client, 是基于 QTA 框架提供的 Windows UI 测试自动化解决方案。



## 1.1 使用前安装

QT4C 依赖 Testbase 模块, 使用前请参考《[Testbase 使用前准备](#)》

除此之外, QT4C 还依赖以下的 Python 库, 可以通过 pip 安装:

```
pip install comtypes PIL pywin32
```

(python 3.x) 中 PIL 需要用 pillow 代替 `pip install comtypes pillow pywin32`

也可以通过以下链接下载:

Python 依赖库下载

之后通过 pip 安装的方式安装 QT4C:

```
pip install qt4c
```

## 1.2 快速入门

### 1.2.1 测试项目

测试用例归属于一个测试项目, 在设计测试用例之前, 如果没有测试项目, 请先参考《[Testbase 创建和修改测试项目](#)》。

## 1.2.2 开发工具

你可以选择你习惯的开发环境，如 Eclipse、命令行执行、PyCharm、VS Code 等，推荐使用 VS Code 开发环境。

## 1.2.3 第一个测试用例

我们先看一个简单的 UI 自动化的用例：

```
import subprocess
import qt4c.wincontrols as wincontrols
from qt4c.testcase import ClientTestCase
from qt4c.qpath import QPath

class QT4CHelloTest(ClientTestCase):
    '''QT4C 示例测试用例'''

    owner = "qta"
    timeout = 1
    priority = ClientTestCase.EnumPriority.Normal
    status = ClientTestCase.EnumStatus.Ready

    def pre_test(self):
        #-----
        self.startStep("关闭当前的所有计算器窗口")
        #-----
        from qt4c.util import Process
        for i in Process.GetProcessesByName('calc.exe'):
            i.terminate()

    def run_test(self):
        #-----
        self.startStep("打开计算器")
        #-----
        subprocess.Popen('calc.exe')
        cw = wincontrols.Window(locator=QPath("/ClassName='CalcFrame' && Text='计算器' &&
↪Visible='True'"))
        btn1 = wincontrols.Control(root=cw, locator=QPath("/ClassName='Button' &&
↪MaxDepth='3' && ControlId='0x83'"))
        result = wincontrols.Control(root=cw, locator=QPath("/ClassName='Static' &&
↪MaxDepth='3' && ControlId='0x96'"))
```

(下页继续)



(续上页)

```
btn1.click()
self.assertEqual("检查按键结果", result.Text, '1')
```

这个测试用例的逻辑很简单：

- 为排除干扰，关闭当前的所有计算器窗口
- 打开一个计算器窗口
- 按下“1”，检查输出结果是否为 1

我们先看主要代码的实现 `run_test`，在第一步通过 `subprocess.Popen` 启动一个计算器进程后，我们实例化一个窗口对象：

```
cw = wincontrols.Window(locator=QPath("/ClassName='CalcFrame' && Text='计算器' && ↵
↵Visible='True'"))
```

可以看到这里使用一个“`qt4c.qpath.QPath`”实例来作为 `locator` 的参数，这个 `locator` 其实是用于唯一定位我们打开的计算器主窗口。

在构造主窗口对象后，我们构造了两个控件实例对象：

```
btn1 = wincontrols.Control(
    root=cw,
    locator=QPath("/ClassName='Button' && MaxDepth='3' && ControlId='0x83'"))
result = wincontrols.Control(
    root=cw,
    locator=QPath("/ClassName='Static' && MaxDepth='3' && ControlId='0x96'"))
```

第一个控件表示的是我们打开的算器的按键“1”，第二个控件表示的是计算器的显示屏，可以看到这里使用两个参数：

- `root`：表示这个控件所属的窗口容器
- `locator`：用于在窗口内唯一定位一个控件

可以看到 `location` 示例是属于我们的 `cw` 窗口，所以其 `root` 为 `cw`。

在构造 `btn1` 和 `result` 对象后，我们点击按键“1”：

```
btn1.click()
```

我们便可以通过 `result` 控件的 `Text` 属性来检查：

```
self.assertEqual("检查按键结果", result.Text, '1')
```

### 1.2.4 定位控件

上面的例子中，我们使用了一个叫 QPath 的 locator 来定位控件。QPath 是怎么唯一定位一个控件的呢？

对于 Windows 平台，其实对于其他的平台也是一样，UI 元素是以树结构组织的。像 Windows 操作系统的文件系统，也是以树结构的形式来管理的，我们使用的文件路径比如：

```
C:\Python27\Lib\ctypes
```

以上的路径就是通过间隔符号 “\” 来分层定位一个文件。所以，可以理解 QPath 其实就是一个 UI 元素的路径定位技术，比如我们实例化计算器窗口的 QPath：

```
/ClassName='CalcFrame' && Text='计算器' && Visible='True'
```

表示的就是从根开始（对于 Windows 来说，桌面窗口就是所有窗口的根），搜索其直接子窗口中符合对应 ClassName, Text 和 Visible 属性的窗口。

而我们构造 location 控件对象时，由于指定了 root 参数，则是以计算器窗口为根节点开始，搜索符合条件的控件元素。从这里可以看出，QT4C 使用两步定位的方式来定位一个控件，先找到这个控件所属的窗口，然后在这个窗口中搜索这个控件。严格来说，Windows 操作系统只有窗口的概念，并没有区分控件和窗口这两个概念，而在 QT4C 中，`qt4c.wincontrols.Window` 的意义是指控件的容器，而 `qt4c.wincontrols.Control` 则表示一个不可以分隔的 UI 元素。因此 QT4C 中是使用 `qt4c.wincontrols.Window` 还是 `qt4c.wincontrols.Control`，关键看使用者怎么理解这个 UI 元素的作用。

更多的控件定位的详细内容，请参考《QPath 语法和使用》

### 1.2.5 理解 UI 结构

上面的 QT4CHelloTest 用例存在两个问题：

- 可读性差，只能依赖构造的实例的名字来猜测控件对应的用途
- 维护成本高，我们在测试用例中硬编码 QPath 的路径，如果被测对象的 UI 结构调整，则需要修改每一个测试用例；而且可以看到在一个用例中，我们也多次引用了同一个 QPath。

为解决这两个问题，我们的测试用例可以这样修改：

```
# -*- coding: utf-8 -*-

from qt4c.qpath import QPath
from qt4c import wincontrols
from qt4c.testcase import ClientTestCase
import subprocess

class CalcWindow(wincontrols.Window):
```

(下页继续)

(续上页)

```

qp = QPath("/ClassName='CalcFrame' && Text='计算器' && Visible='True'")

def __init__(self):
    super(CalcWindow, self).__init__(locator=self.qp)

    locators = {
        '按键 1': {'type': wincontrols.Control, 'root': self, 'locator': QPath("/
↪ClassName='Button' && MaxDepth='3' && ControlId='0x83'")},
        '结果': {'type': wincontrols.Control, 'root': self, 'locator': QPath("/
↪ClassName='Static' && MaxDepth='3' && ControlId='0x96'")}
    }

    self.updateLocator(locators)

    @staticmethod
    def closeAll():
        from qt4c.util import Process
        for i in Process.GetProcessesByName('calc.exe'):
            i.terminate()

class QT4CHelloTest(ClientTestCase):
    '''QT4C 示例测试用例
    '''
    owner = "qta"
    status = ClientTestCase.EnumStatus.Ready
    timeout = 1
    priority = ClientTestCase.EnumPriority.Normal

    def pre_test(self):
        #-----
        self.startStep("关闭当前的所有计算器窗口")
        #-----
        CalcWindow.closeAll()

    def run_test(self):
        #-----
        self.startStep("打开计算器")
        #-----
        subprocess.Popen('calc.exe')
        cw = CalcWindow()

```

(下页继续)

(续上页)

```

cw.Controls['按钮 1'].click()
self.assert_equal("检查按钮结果", cw.Controls['结果'].Text, '1')

```

可以看到我们封装了一个 CalcWindow 类，表示这个计算器的窗口。

先看看 run\_test 用例，和之前有较大的变化，实例化窗口对象不需要提供 locator 参数：

```

cw = CalcWindow()

```

检查其子控件的属性也变得简单易懂：

```

cw.Controls['结果'].Text

```

当然这得益于对 CalcWindow 的封装。CalcWindow 本身是 “*qt4c.wincontrols.Window*” 的子类，表示这个是一个窗口，在其调用基类构造函数的时候传递了 locator 参数，因此使用的时候变得简单，无需任何参数。ExplorerFolder 还在构造函数中调用了 updateLocator 方法，传入一个字典：

```

{
    '按钮 1': {
        'type': wincontrols.Control,
        'root': self,
        'locator': QPath("/ClassName='Button' && MaxDepth='3' && ControlId='0x83'")
    },
    '结果': {
        'type': wincontrols.Control,
        'root': self,
        'locator': QPath("/ClassName='Static' && MaxDepth='3' && ControlId='0x96'")
    }
}

```

这个字典其实就是这个窗口的子控件的布局的描述，每个子控件的描述由四个部分组成：

- 名称：用字符串表示这个子控件的名字，窗口对象可以通过这个名字来引用使用对应的子控件，就像我们在测试用例中使用 “文件地址显示框” 一样。
- type：表示子控件的类型，下面我们会介绍
- root：表示子控件定位时使用的跟节点，一般都是用 self，也就是当前的窗口对象
- locator：表示子控件定位时使用的路径，一般是 QPath 或其他类似的控件定位符号

通过 CalcWindow 类，可以更清晰得看到 QT4C 两层 UI 结构的意义所在。

实际上，除了 Window 和 Control，QT4C 还提供了第三层的 UI 结构，即是 “*qt4c.app.App*”。比如上面的用例，我们可以增加一个 ExplorerApp 类并对应修改 ExplorerFolder 类：

```

class CalcWindow(wincontrols.Window):
    qp_str = "/ClassName='CalcFrame' && Text='计算器' && Visible='True'"

    def __init__(self, app):
        super(CalcWindow, self).__init__(locator=QPath(self.qp_str + " && ProcessId='%s'"
↪ " % app.ProcessId))

        locators = {
            '按键 1': {'type': wincontrols.Control, 'root': self, 'locator': QPath("/
↪ ClassName='Button' && MaxDepth='3' && ControlId='0x83'")},
            '结果': {'type': wincontrols.Control, 'root': self, 'locator': QPath("/
↪ ClassName='Static' && MaxDepth='3' && ControlId='0x96'")}
        }

        self.updateLocator(locators)

class CalcApp(App):
    ''' 计算器应用
    '''

    def __init__(self):
        ''' 构造函数
        '''

        p = subprocess.Popen('calc.exe')
        self._pid = p.pid

    @property
    def ProcessId(self):
        ''' 对应 calc.exe 进程的进程 ID
        '''

        return self._pid

    def quit(self):
        CalcWindow(self).close()

    def kill(self):
        CalcWindow(self).close()

    @staticmethod
    def killAll():
        from qt4c.util import Process

```

(下页继续)

(续上页)

```
for i in Process.GetProcessesByName('calc.exe'):
    i.terminate()
```

对应的，测试用例可以修改为：

```
class QT4CHelloTest(ClientTestCase):
    '''QT4C 示例测试用例'''
    owner = "qta"
    status = ClientTestCase.EnumStatus.Ready
    timeout = 1
    priority = ClientTestCase.EnumPriority.Normal

    def run_test(self):
        #-----
        self.startStep("打开计算器")
        #-----
        calc = CalcApp()
        cw = CalcWindow(calc)
        cw.Controls['按钮 1'].click()
        self.assertEqual("检查按钮结果", cw.Controls['结果'].Text, '1')
```

这个用例和之前的用例有比较大的区别就是少了 `pre_test`，原因是 `CalcApp` 类已经实现了对多个计算器的管理的功能，所以可以不用在测试之前通过将全部窗口都关闭的方式来避免发生干扰。QT4C 的 App 的作用，在 UI 的层面上，就是用于管理多个重复窗口的情况；在操作系统的层面上讲，App 可以理解为用户提供一个特定功能的软件，一般来说可能是对应操作系统的进程、一个线程、或者多个进程集合。

更多的 UI 结构和封装的详细内容，请参考《UI 结构和封装》。

## 1.2.6 控件类型和属性

在指定 UI 布局的时候，我们可以选择对应的控件的类型，在上面的例子里面：

```
{
    '按钮 1': {
        'type': wincontrols.Control,
        'root': self,
        'locator': QPath("/ClassName='Button' && MaxDepth='3' && ControlId='0x83'")
    },
    '结果': {
        'type': wincontrols.Control,
```

(下页继续)

(续上页)

```

        'root': self,
        'locator': QPath("/ClassName='Static' && MaxDepth='3' && ControlId='0x96'")
    }
}

```

type 使用的是 “*qt4c.wincontrols.Control*” 类型，我们需要如何选择控件的类型？有哪些控件的类型呢？同时，在检查控件的属性时，我们使用 Text 属性：

```

cw.Controls['结果'].Text

```

但是我们需要如何知道使用哪个属性？

QT4C 目前支持三种类型的控件，分别是：

- Native 控件，仅支持 Windows 操作系统提供的窗口控件，能力有限，维护和接入成本低，接口请参考《api/qt4c》
- UIA 控件，基于 UI Automation 能力，可以支持各类 UI 控件，能力较强，接入和维护成本低，接口请参考《api/qt4c》
- Web 控件，基于 QT4W 提供的内嵌 Web 页面的控件识别能力

Web 控件和其他的控件的使用略有不同，我们会在《Web 自动化测试支持》中讨论。

Native, UIA 是控件的实现方式的差异导致，在同种实现方式下，也会有很多不同的控件类型，比如对于 Native 控件，我们前面使用到的 “*qt4c.wincontrols.Control*” 表示的就是普通的控件，但如果是可以列表控件，则可以使用 “*qt4c.wincontrols.ListView*”。使用哪种类型的控件，关键是看使用者要如何使用这个控件，QT4C 不会也无法检查选择控件类型和对应的控件是否匹配。例如，对于一个 ComboBox，可以对该控件的 Value 值进行设置，如果要实现这样的操作，则控件的类型必须指定为 “*qt4c.uiacontrols.ComboBox*”：

```

{
    '设置 Value 值': {
        'type': uiacontrols.ComboBox,
        'root': self,
        'locator': QPath('/ControlId = "0x3E9" && MaxDepth="8"')
    },
}

```

指定之后可以这样使用：

```

cw['设置 Value 值'].Value = 'value'

```

## 1.2.7 Demo 工程

你可以从 github 下载《Demo 工程》，其中包括 demo 测试项目源码，Demo 工程中以 Windows 自带的计算器为例，提供了 Windows Native 控件及 UIA 控件的使用参考，同时提供了 Web 自动化的测试用例。

## 1.3 定位 UI 元素

在 QT4C 中，UI 元素的主要类型分为 Window 和 Control 两种类型。对于 UI 元素，不管是 Window 或者是 Control，定位都主要是依靠 QPath 来进行定位的，QT4C 需要用 QPath 来标识每一个元素。这里以系统自带的计算器为例，举一个简单的使用 QPath 封装控件的例子：

```
import qt4c.wincontrols as win
self.updateLocator({
    ' 回退键': {
        'type': win.Control,
        'root': self,
        'locator': QPath("/ClassName='CalcFrame' && Text=' 计算器' && Visible='True' /
↪ClassName='Button' && MaxDepth='3' && Instance='1'")
    }
}),
```

可以看到 QT4C 中定位 UI 元素需要 type, root 和 locator 属性。

- type: 指定控件的类型，和 Windows 中定义的控件类型相对应，如 Control, TextBox, ComboBox 等等。
- root: 指定控件的父节点，指定父节点后，查找控件时，会先找到父节点，然后以父节点为根节点，从根节点开始查找目标控件，这对于你定义的 QPath 找到多个的情况非常有用，如果指定了父节点，就只会返回父节点下的节点，否则找到多个重复节点就会报错。如果指定为 self，则表示会从整颗控件树根节点开始查找目标控件。
- locator: 控件封装的定位器，通常传入字符串类型或 QPath 对象。

QT4C 定位 UI 元素主要依靠的就是 locator 属性，可以传入单独的字符串或者一个 QPath 对象。如果传入的是字符串类型的值，则默认在全局对 Name 属性为指定的字符串的控件进行查找。更多对于 QPath 语法的学习请先参考《QPath 语法和使用》。

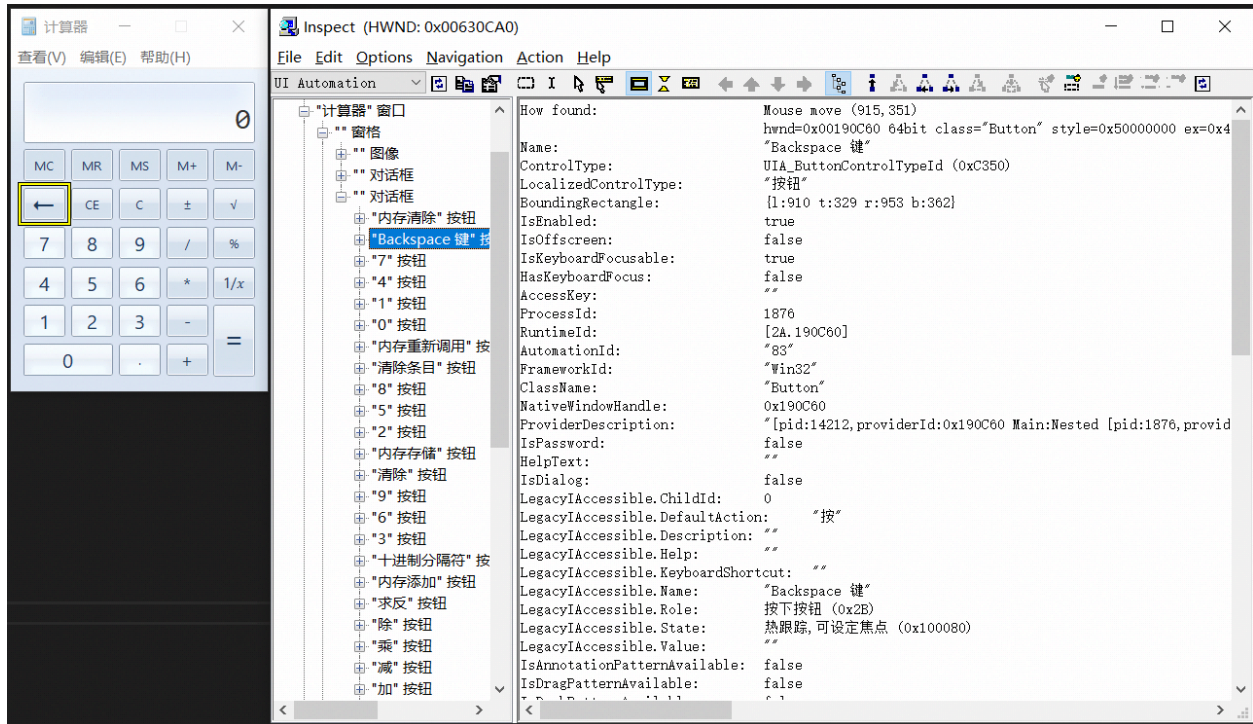
### 1.3.1 使用 Inspect 获取控件

Inspect 是一种 Win32 应用控件抓取工具，你可以使用微软提供的 Inspect.exe 来获取控件，关于微软的 Inspect.exe 的使用，可参考《Inspect.exe 介绍》。



**注解：** UIA 控件使用微软提供的 Inspect.exe 工具中的 UI Automation 模式进行抓取。

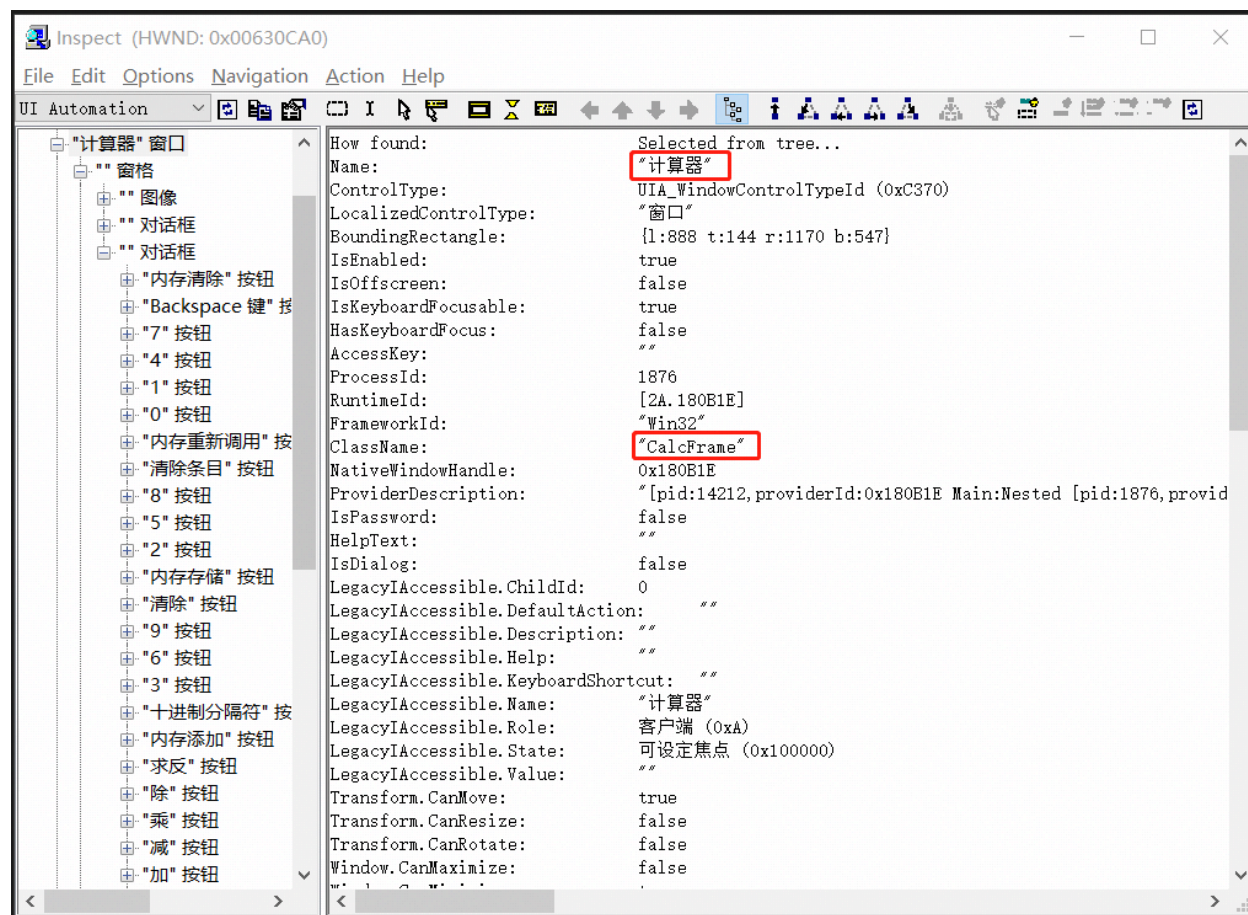
打开系统自带的计算器后，打开 Inspect 控件抓取工具 (这里以 Inspect.exe 为例)，开始探测控件树，得到界面如下：



上面封装的控件就是上图黄框中计算器中的回退键。使用 Inspect 控件抓取工具可以遍历计算器的控件树，同时可以看到某一个 UI 元素的属性值。获取到控件属性之后，你就可以通过这些属性值来设计 QPath 来定位 UI 元素了。

### 1.3.2 使用 QPath 封装控件

下面通过 Inspect 工具看一下计算机主界面的属性，然后对主界面进行封装：



上面标识出来的 ClassName、Text(Name), 以及 ControlId(AutomationId)、Visible 等都是比较常见的可以用于封装 QPath 的属性。不止 Control 类的封装需要使用 QPath, Window 类的封装也是需要 QPath 的。首先封装一个 Window 类:

```
import qt4c.wincontrols as win
from qt4c.qpath import QPath

class MainPanel(win.Window):
    def __init__(self):
        qp = QPath("/ClassName='CalcFrame' && Text='计算器' && Visible='True'")
        super(MainPanel, self).__init__(locator=qp)
```

这里没有传入 root 参数, 那么默认以桌面作为根节点进行查找, 通过 ClassName、Text 和 Visible 属性, 就可以定位到计算器的主页面了, 但是需要注意的是, 同样的窗口的 ControlId(AnimationId) 有时候不一定会是一样的, 所以使用 ControlId 进行控件的封装时需要确认 ControlId 是否不会发生变化。

接下来我们使用 QPath 来定位计算器主界面中的几个按钮:

```
import qt4c.wincontrols as win
```

(下页继续)

(续上页)

```

from qt4c.qpath import QPath

class MainPanel(win.Window):
    def __init__(self):
        qp = QPath("/ClassName='CalcFrame' && Text='计算器' && Visible='True'")
        super(MainPanel, self).__init__(locator=qp)

        locators = {
            '按键 1': {
                'type': win.Control,
                'root': self,
                'locator': QPath("/ClassName='Button' && MaxDepth='3' && ControlId='0x83'
↪")},
            '按键 2': {
                'type': win.Control,
                'root': self,
                'locator': QPath("/ClassName='CalcFrame' /ClassName='#32770' && Instance=
↪'1' /ClassName='Button' && Instance='10'")
            }
        }

        self.updateLocator(locators)

```

以“按键 1”为例，这是 QPath:

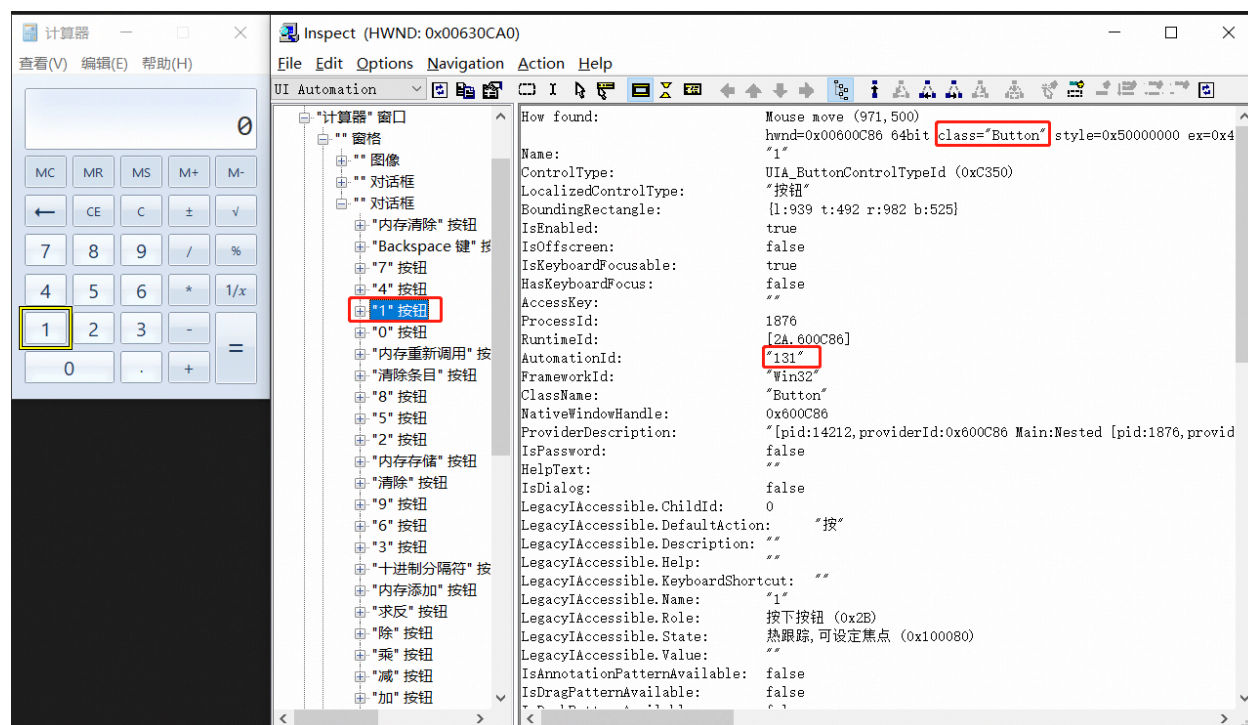
```

'按键 1': {
    'type': win.Control,
    'root': self,
    'locator': QPath("/ClassName='Button' && MaxDepth='3' && ControlId='0x83'")
}

```

这里以主界面作为根节点进行查找，在 Inspect 中可以看到，该控件是在以主界面为根节点深度为 3 的位置，所以可以直接设定 MaxDepth=3，但是发现与该控件同一级的出现了多个 Button，只从 ClassName 属性是无法准确定位到“按键 1”的。不过通过打开多个计算器窗口进行控件抓取发现这些按钮控件的 ControlId 是不会发生变化的，因此可以利用 ControlId 来进行唯一定位。

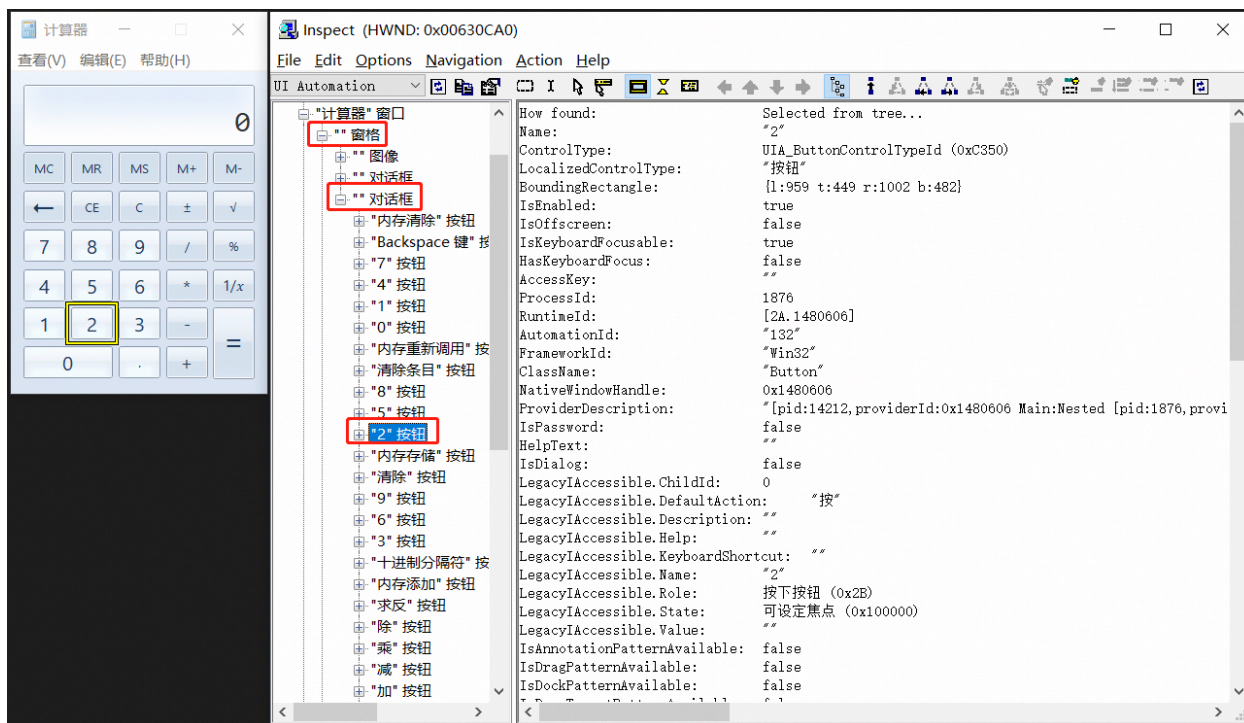




那如果恰好你需要定位的控件的 ControlId 并非唯一确定的呢？这个时候你就需要使用“按键 2”的定位方法：

```
'按键 2': {
    'type': win.Control,
    'root': self,
    'locator': QPath("/ClassName='CalcFrame' /ClassName='#32770' && Instance='1' /
    ↪ClassName='Button' && Instance='10'")
}
```

首先以主窗口为根节点，编写多个 QPath 依次定位到其子孙节点，其中当定位到第二层的控件的时候，因为同一层出现了多个 ClassName 为“#32770”的控件，可以通过图中红框所示来确定控件在该层的位置，然后使用 Instance 来确定需要的控件，最后定位到“按键 2”也是同理，这样子就可以避免了 ControlId 不唯一导致无法准确定位控件的情况了。



通过类似的方法，你就可以一一封装计算器主界面中的所有按钮了。

### 1.3.3 QT4C 常用 QPath 属性

#### 控件的常用属性

##### ClassName 属性

ClassName 是 QT4C 中定位控件比较常用的一个属性，通过 Inspect 工具就可以获取到控件的 ClassName 属性，但是该属性一般不唯一，通常需要搭配其他属性进行使用。

此外，该属性大多数情况下还可以作为 type 属性选择的一个重要参考，除了最基础的 Control 类型，QT4C 还提供了 TextBox、ListView、ComboBox 等控件类型，具体请参考接口文档进行使用。

##### ControlId 属性

该属性是控件的唯一标识，一般优先考虑使用 ControlId 进行定位，但并不是所有控件都存在 ControlId 属性，当控件不存在 ControlId 属性时，则需要使用其他属性进行定位。

**警告：** 在不同 Inspect 工具中，ControlId 的命名各不相同，请根据实际情况在 Inspect 中获取对应的 ControlId。

## Text 属性

Text 属性在 Text 类控件出现比较多，当不存在 ControlId 且 Text 属性不为空时，可以使用 Text 属性进行定位。

## Visible 属性

每个控件都有 Visible 属性，该属性常用于同时存在多个控件时，定位当前可见的控件。

## QPath 的特殊属性

### UIType 属性

在 QT4C 中出现有多种控件类型，包括 Win32 控件、UIA 控件和 html 控件，当 UIType 没有指定时默认取值为父节点的值。有时候使用 UIType 也可以高效辅助进行控件的定位。

### MaxDepth 属性

有时你需要定义跨层的控件，例如 M 层定义后，接下来要定义第 N 层 ( $M-N>1$ ) 的控件，但是如果一层一层定义下来比较费事，就可以借助 MaxDepth，就会搜索 M 层下的 N 层内的所有控件。使用可参考上面控件“按键 1”的封装和《QPath 语法和使用》。

### Instance 属性

有时候我们需要定位的控件没有可以用来直接定位的属性，那么你可以使用 Instance 属性，该属性可以在当前 QPath 定位到的多个控件中定位到你需要的第 n 个控件。如果没有明确指定时默认取值为 '1'，即直接父子关系。具体使用可参考上面控件“按键 2”的封装和《QPath 语法和使用》。

## 正则匹配

在上面的例子中，我们都是用全匹配的方式，即在 QPath 中都是用“=”，如果要定义的内容有部分是要变化的，可以考虑用正则匹配的方式，例如上面提到的“按键 2”控件，如果前后有可能有别的变化的内容，可如下定义：

```
'按键 2': {
    'type': win.Control,
    'root': self,
    'locator': QPath("/ClassName~='Calc.*' /ClassName='#32770' && Instance='1' /
↪ClassName='Button' && Instance='10'")
}
```

---

**注解：**从上面可以看出，同一个控件是可以有多种写法的，所以定义时应该选择最简洁的写法，不要不必要地复杂化。当需要多个字段辅助定义才能定位一个控件时，才进行结合使用。

---

## 1.4 UI 结构和封装

Window 窗口中的各个控件元素需要用 QPath 进行定位，同时控件类型分为 win32 控件、uia 控件两种控件类型来分别进行封装使用。

### 1.4.1 封装测试基类

#### 测试基类概述

QTAF 中实现的测试基类《TestCase》提供了很多功能接口，如环境准备和清理、断言、日志相关等功能，详细见测试基类的相关说明。QT4C 的测试基类 ClientTestCase 重载了 QTAF 中的测试基类 TestCase，在 TestCase 的基础上复用了其功能并重写了部分方法。

#### 测试基类封装

QT4C 的测试基类 `qt4c.testcase.ClientTestCase` 实现了部分 Windows 端运行测试用例所需功能。你可以在 `demolib/testcase.py` 封装你的测试基类 DemoTestCase，并且根据测试项目的实际需要重载各个接口，用法参考如下：

```
# -*- coding: utf-8 -*-
''' 示例测试用例
'''

from testbase import testcase
from qt4c.testcase import ClientTestCase

class DemoTestCase(ClientTestCase):
    '''demo 测试用例基类
    '''
    pass
```

#### 测试基类使用

封装的测试基类 DemoTestCase 可以直接作为测试用例的基类使用，例如在 `demotest/hello.py` 使用如下：

```
class HelloTest(DemoTestCase):
```

## 1.4.2 封装 App

### App 类概述

App 类是 QT4C 中的应用程序基类模块，它可以理解为提供一个特定功能的软件，一般来说可能是对应操作系统的一个进程、一个线程、或者多个进程集合。App 类提供了 Windows 上应用程序的部分相关功能，包括退出程序、退出所有应用程序等等。

### App 类封装

App 基类中维护了一个记录所有打开的应用程序的列表，并且提供了一些静态方法来对当前记录的所有应用程序进行操作，如 App.quitAll() 退出所有应用程序，更多使用方法可查看 [qt4c.app.App](#) 进行查看。

我们以 Windows 系统自带的计算器为例，你可以在 demolib/calcap.py 封装你的测试基类 CalcApp，实现 App 类所需的基本功能，用法参考如下：

```
# -*- coding: utf-8 -*-

from qt4c.app import App
import subprocess, time

class CalcApp(App):
    ''' 计算器 App
    '''
    def __init__(self):
        App.__init__(self)
        self._process = subprocess.Popen('calc.exe')

    def quit(self):
        self._process.kill()
        App.quit(self)
```

上述代码实现了最基本的功能，你可以根据需要去定义更多的接口。\_\_init\_\_ 函数中需要实现应用程序的启动，这里我们通过 subprocess 启动一个 calc.exe 的子进程并获取其 pid。而 quit 函数中需要实现程序退出。

**警告：** 重载 quit 函数时，必须显示调用基类的函数，以通知基类该程序退出。



## App 类自定义接口

可能上面 demo 实现的基本功能无法满足你的需求，你可以自定义一些操作，然后自行实现。例如我们希望在实例化 App 之后直接调用 App 进行计算，那么可以修改如下：

```
# -*- coding: utf-8 -*-
from qt4c.app import App
import subprocess, time

class CalcApp(App):
    ''' 计算器 App
    '''

    def __init__(self, cmd, params=[]):
        App.__init__(self)
        params.insert(0, cmd)
        self._process = subprocess.Popen(params)

    @property
    def ProcessId(self):
        return self._process.pid

    def quit(self):
        self._process.kill()
        App.quit(self)

    def calculate(self, expression, expect_value):
        ''' 计算表达式并比较运算结果
        '''
        pass
```

之后我们可以通过调用 calculate 方法来直接进行计算：

```
calcapp = CalcApp('calc.exe')
calcapp.calculate('3*3', 9)
```

## App 类使用

在测试用例中，实例化一个被测应用程序，参考如下：

```
from demolib.calcapp import CalcApp
calc = CalcApp()
```

实例化之后，我们就可以看到一个计算器进程被启动。

### 1.4.3 封装 Window

#### Window 概述

每个应用程序都包含多个窗口 (Window)，每个窗口可以封装成一个窗口类，在 lib 库中实现，建议 App 中一个产品功能模块封装在一个 py 文件中，例如 app 的主面板相关作为一个 py 文件 mainpanel.py，app 的登录相关作为一个文件 login.py，在文件中再具体实现多个相关类的 UI 封装。

#### Window 封装

封装 Window 类需要继承 QT4C 的 Window 基类，根据当前应用程序的不同，可以选择不同的 Window 基类。当前 QT4C 提供两种 Window 基类：

- Win32 窗口： `qt4c.wincontrols.Window`
- UIA 窗口： `qt4c.uiacontrols.UIAWindows`

其中 Win32 窗口是 Windows 原生窗口，UIA 窗口是基于 UIA 实现的，它是 Microsoft Windows 的一个辅助功能框架，使用时需要针对不同场景选择不同的 Window 基类。

在《[定位 UI 元素](#)》中已经简单介绍了 QPath 的设计，你可以利用 QPath 来封装一个 Window 类：

```
import qt4c.wincontrols as win
from qt4c.qpath import QPath

class MainPanel(win.Window):
    def __init__(self):
        qp = QPath("/ClassName='CalcFrame' && Text='计算器' && Visible='True'")
        super(MainPanel, self).__init__(locator=qp)

        locators = {
            '按键 1': {'type': win.Control, 'root': self, 'locator': QPath("/ClassName=
↪ 'Button' && MaxDepth='3' && ControlId='0x83'")},
            '按键 2': {'type': win.Control, 'root': self, 'locator': QPath("/ClassName=
↪ 'Button' && MaxDepth='3' && ControlId='0x84'")},
            '加号': {'type': win.Control, 'root': self, 'locator': QPath("/ClassName=
↪ 'Button' && MaxDepth='3' && ControlId='0x5D'")},
            '等号': {'type': win.Control, 'root': self, 'locator': QPath("/ClassName=
↪ 'Button' && MaxDepth='3' && ControlId='0x79'")},
```

(下页继续)

(续上页)

```

        '结果': {'type': win.Control, 'root': self, 'locator': QPath("/ClassName=
↪'Static' && MaxDepth='3' && ControlId='0x96'")}]

    }

    self.updateLocator(locators)

```

这是一个简单的计算器的主界面，因为在一个应用程序中，一个窗口的 UI 元素的属性基本上不会发生太大变化，所以你可以在封装一个 Window 类时就定义好 QPath，这样子就不需要每一次在实例化窗口的时候都要编写一遍 QPath 了。你可以在你的 Window 类实现你需要的功能，例如封装一个 addNum 方法来实现两个整数的相加（假设以“按键 x”封装了计算器中的数字和小数点按键）：

```

def addNums(self, num1, num2):
    self.wait_for_exist(5, 0.2)
    for i in str(num1):
        self.Controls[('按键%s' % i)].click()
    self.Controls['加号'].click()
    for i in str(num2):
        self.Controls[('按键%s' % i)].click()
    self.Controls['等号'].click()

```

对于 UIAWindows，请参照接口文档进行使用。

## Window 类使用

定义好窗口类之后，在用例中我们可以实例化窗口类，并调用对应的功能接口：

```

from demolib.mainpanel import MainPanel
main_panel = MainPanel()
main_panel.bringForeground()    # 将窗口设为最前端窗口 (QT4C 提供)
main_panel.addNum(1, 2)         # 自定义方法

```

### 1.4.4 封装 Control

QT4C 中提供了丰富的控件类型供使用，如 ListView, TrayNotifyBar, ComboBox 等，对于 Win32 控件和 UIA 控件，QT4C 封装了不同的控件类型可供使用，QT4C 在 Python 层也对这些控件的自动化做了封装，以供使用。

QT4C 已支持的一些常用控件类型如下（对于未封装的控件类型应使用 Control 基类进行封装）：

Win32 控件：

控件类型	使用场景	对应接口
输入框	常用于一般的输入框	<code>qt4c.wincontrols.TextBox</code>
下拉框	可获取/选择下拉选项中某项	<code>qt4c.wincontrols.ComboBox</code>
列表 (项)	常用于一般的列表	<code>qt4c.wincontrols.ListView</code>
菜单 (项)	可获取菜单项	<code>qt4c.wincontrols.Menu</code>
树列表 (项)	树形控件，如文件列表树	<code>qt4c.wincontrols.TreeView</code>

UIA 控件：

控件类型	使用场景	对应接口
输入框	可进行输入操作	<code>qt4c.uiacontrols.Edit</code>
下拉框	可获取/选择下拉选项中某项	<code>qt4c.uiacontrols.ComboBox</code>
单选按钮	可获取按钮状态	<code>qt4c.uiacontrols.RadioButton</code>

更多控件类型请参考《qt4c\_package》进行使用。

参照《定位 UI 元素》，一个简单的控件定义如下，这里封装了计算器主界面的按键 1：

```
class MainPanel(win.Window):
    def __init__(self, qpath=None):
        qp = QPath("/ClassName='CalcFrame' && Text='计算器' && Visible='True'")
        super(MainPanel, self).__init__(locator=qp)

        self.updateLocator({
            '按键 1': {'type': win.Control, 'root': self, 'locator': QPath("/ClassName=
→ 'Button' && MaxDepth='3' && ControlId='0x83'")},
        })
```

你也可以使用封装好的控件类型作为基类进行封装（假设按键 1 是一个 UIA 按钮控件）：

```
import qt4c.uiacontrols as uia
self.updateLocator({
    '按键 1': {'type': uia.Button, 'root': self, 'locator': QPath("/ClassName='Button' &&
→ MaxDepth='3' && ControlId='0x83'")},
})
```

定义完成之后，在 MainPanel 的接口下调用方式如下：

```
self.Controls['按键 1']
```

我们还可以获取其属性，例如 ControlId 属性：

```
print self.Controls['按键 1'].ControlId
```

或者点击:

```
self.Controls['按键 1'].click()
```

## 1.5 输入操作

在 Win32 应用自动化过程中, 避免不了的是对设备的各种操作, 例如鼠标点击、拖拽、键盘输入等等。QT4C 通过 Mouse 类 `qt4c.mouse.Mouse` 和 Keyboard 类 `qt4c.keyboard.Keyboard`, 提供了常用的鼠标操作和键盘输入操作。

用户可以调用控件内已封装的接口进行设备操作, 也可以调用 Mouse 类和 Keyboard 类的方法进行设备操作。

**警告:** 请尽量使用控件内已封装的方法进行设备操作, 不推荐直接调用 Mouse 类和 Keyboard 类的方法进行设备操作。

### 1.5.1 控件的点击及输入

QT4C 在 Control 类和 Window 类中都封装有用于鼠标点击的接口, 你可以直接调用这些接口进行设备操作:

```
from demolib.mainpanel import MainPanel
main_panel = MainPanel()
main_panel.click(xOffset=100, yOffset=100)
main_panel.Controls['按键 1'].click()
```

对于部分 Edit 类控件, 你可以直接修改其 Text 属性来达到键盘输入的效果, 假设 `main_panel.Controls['密码框']` 是一个密码输入框:

```
main_panel.Controls['密码框'].Text = 'XXXXXXXX'
```

### 1.5.2 鼠标输入

Mouse 类中提供了鼠标移动、点击、拖拽等等操作, 这里对一些比较常用的操作进行说明。除了以下操作之外, 其他操作请具体参照接口文档 `qt4c.mouse.Mouse` 进行使用。

## 鼠标移动

在 `Mouse` 类封装了 `move`、`sendMove`、`PostMove` 等方法，可以对鼠标进行移动操作。使用参考如下：

```
from qt4c.mouse import Mouse
Mouse.move(650, 200)           # 移动鼠标到 (650, 200) 位置
Mouse.sendMove(0x7001B8, 650, 200) # 通过发消息的方式产生鼠标移动的操作
```

`sendMove` 和 `PostMove` 是在目标窗口通过发消息的方式产生鼠标移动的操作，所以需要额外传入目标窗口句柄作为参数，具体使用请参照接口文档。

## 鼠标点击

在 `Mouse` 类封装了 `click`、`sendClick`、`PostClick` 等方法，可以对鼠标进行点击操作。使用参考如下：

```
from qt4c.mouse import Mouse, MouseFlag, MouseButton
Mouse.click(650, 200, MouseFlag.LeftButton, MouseButton.SingleClick) # 在 (650, 200)
位置点击鼠标左键
```

其中 `MouseFlag.LeftButton` 和 `MouseButton.SingleClick` 分别用于控制点击的鼠标键类型和点击方式，更多类型请参考接口文档进行使用。

而 `sendClick` 和 `PostClick` 同样是在目标窗口通过发消息的方式产生鼠标点击的操作，所以需要额外传入目标窗口句柄作为参数，使用方式参照鼠标移动，除此之外，`Mouse` 类还提供了缓慢点击等其他点击功能，具体使用请参照接口文档。

## 鼠标拖拽

在 `Mouse` 类封装了 `drag` 方法用于鼠标拖拽。使用参考如下：

```
from qt4c.mouse import Mouse
Mouse.drag(0, 0, 650, 200)    # 鼠标从 (0, 0) 拖拽到 (650, 200)
```

## 鼠标滚动

在 `Mouse` 类封装了 `scroll` 方法用于鼠标滚动。使用参考如下：

```
from qt4c.mouse import Mouse
Mouse.scroll(False)          # 鼠标向后滚动
```

### 1.5.3 键盘输入

Keyboard 类中提供了两种键盘输入方式，一种使用模拟键盘输入的方式，另外一种则是使用 Windows 消息的机制将字符串直接发送到窗口。

Keyboard 类支持以下字符的输入：

1. 特殊字符：^, +, %, {, }

- ‘^’ 表示 Control 键，同 ‘{CTRL}’。’+’ 表示 Shift 键，同 ‘{SHIFT}’。’%’ 表示 Alt 键，同 ‘{ALT}’。
- ‘^’，‘+’，‘%’ 可以单独或同时使用，如 ‘^a’ 表示 CTRL+a，’^%a’ 表示 CTRL+ALT+a。
- {}：大括号用来输入特殊字符本身和虚键，如 ‘{+}’ 输入加号，’{F1}’ 输入 F1 虚键，’{}}’ 表示输入 ‘}’ 字符。

2、ASCII 字符：除了特殊字符需要 {} 来转义，其他 ASCII 码字符直接输入，

3、Unicode 字符：直接输入，如”测试”。

4、虚键：

- {F1}, {F2}, ..., {F12}
- {Tab}, {CAPS}, {ESC}, {BKSP}, {HOME}, {INSERT}, {DEL}, {END}, {ENTER}
- {PGUP}, {PGDN}, {LEFT}, {RIGHT}, {UP}, {DOWN}, {CTRL}, {SHIFT}, {ALT}, {APPS}..

**警告：** 当使用联合键时，请注意此类的问题：对于 inputKeys( ‘^W’ ) 和 inputKeys( ‘%w’ )，字母 ‘w’ 的大小写产生的效果可能不一样

### 键盘输入

Keyboard 类封装了 inputKeys 和 postKeys 方法用于键盘输入，使用方式如下：

```
from qt4c.keyboard import Keyboard
Keyboard.inputKeys("QT4C")           # 模拟键盘输入 "QT4C"
Keyboard.postKeys(0x7001B8, "QT4C") # 将 "QT4C" 字符串以窗口消息的方式发送到指定 win32 窗口
```

## 1.6 Web 自动化测试支持

QT4C 的 Web 自动化测试依赖 QT4W，关于 Web 自动化测试的相关内容，使用前请先参考《QT4W 使用文档》熟悉 QT4W 的基础知识。

在使用 QT4C 进行 Web 自动化测试，你需要先安装 QT4W：

```
pip install qt4w
```

在 QT4C 中提供了对 Web 自动化测试的支持，并且封装了 cef3webview、chromewebview、iewebview 和 tbswebview 四种 webview：

webview 类型	支持 Web 页面类型	实现方式
iewebview	IE 浏览器页面以及 IE 内嵌页面	通过 JS 注入
chromewebview	Chrome 浏览器页面	远程调试
cef3webview	cef3 内嵌页面	通过 JS 注入
tbswebview	tbs 内嵌页面	远程调试

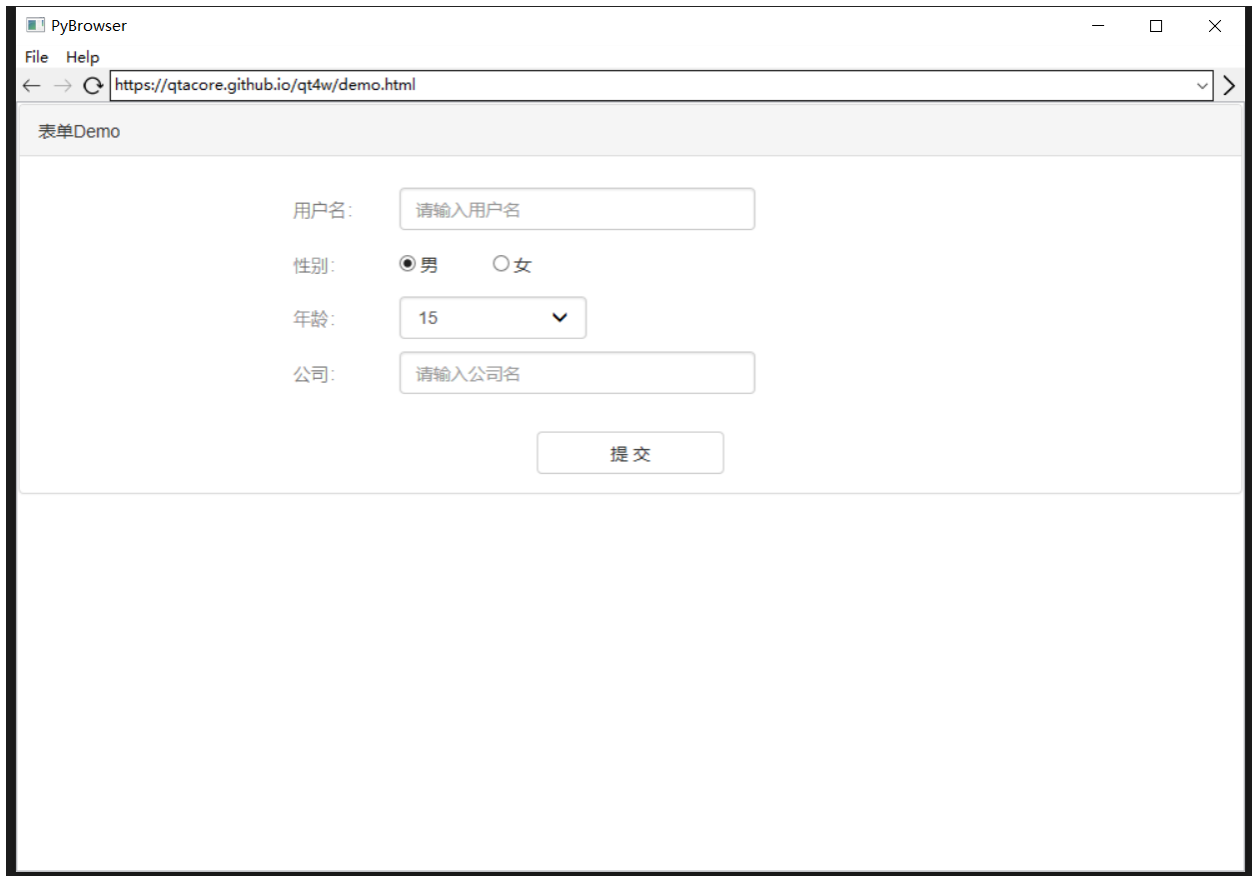
在 Web 自动化测试中，你需要对 WebPage 和 WebElement 进行封装。在 QT4C 中提供了两种封装方式：

- 使用 webview 基类进行 Webview 的封装，封装内嵌页面标识
- 使用浏览器 Browser 类来打开浏览器窗口，封装基础页面标识

### 1.6.1 Web 内嵌页面的自动化测试

QT4C 还提供了对 Web 内嵌页面的支持，这里以一个简单的 Demo 应用程序 PyBrowser 为例，这是一个简单的输入 url 访问对应网站的应用程序，其中包含了一个 IE 内嵌页面：





首先参考《封装 App》对 Demo 应用程序封装一个简单的 App 类:

```
# -*- coding: utf-8 -*-
from qt4c.app import App
import subprocess, time

class PyBrowserApp(App):
    '''DemoWeb App
    '''

    def __init__(self):
        App.__init__(self)
        self._process = subprocess.Popen('C:\\\\Users\\qta\\Desktop\\pyBrowser.exe')

    @property
    def ProcessId(self):
        return self._process.pid

    def quit(self):
```

(下页继续)

(续上页)

```

self._process.kill()
from qt4c.util import Process
for i in Process.GetProcessesByName('pyBrowser.exe'):
    i.terminate()
App.quit(self)

```

接下来以 QT4W 示例页面为例，使用 Inspect 获取该窗口的控件树，对 Web 页面进行封装。

### 封装 webview

webview 相当于一个内嵌页面的容器控件，如果你想要对内嵌 Web 页面进行封装的话，你必须先对容器进行封装。

根据不同内嵌页面，你可以选择不同的 webview 进行封装。因为 Demo 应用程序的内嵌页面属于 IE 内嵌页面，因此可以使用 iwebview 来进行封装，参考如下：

```

from qt4c.webview.iwebview import IEWebView
from qt4c import wincontrols

class Webkit(IEWebView):
    ''' 用于展示内嵌 Web 页面的容器控件
    '''

    def __init__(self, locator):
        self._win = wincontrols.Control(locator=locator)
        super(WebKit, self).__init__(self._win)

```

### WebPage 的封装和使用

WebPage 一般对应一个 Web 页面，它里面包含多个 Web 控件元素。QT4C 继承了 QT4W 的 WebPage 的实现，封装了 PC 端自动化所需的页面相关逻辑。在封装某个页面的 WebPage 时，一般会直接继承 QT4C 的 WebPage 的实现。

首先以一个实例介绍如何封装一个 Web 控件，参考如下：

```

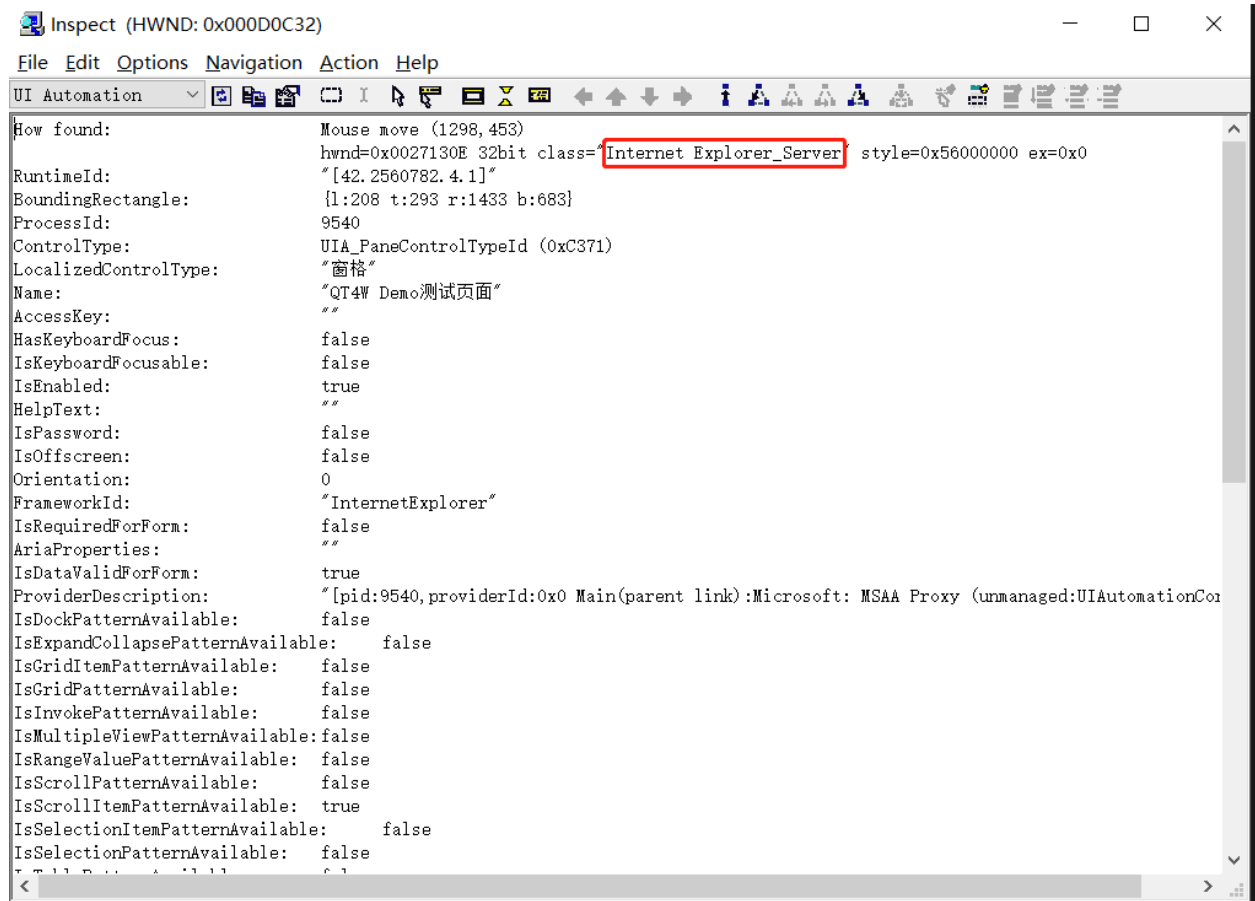
'controlname':{
    'type': WebElement,
    'locator': XPath('//div[@id="controlid"]'),
}

```

封装一个 Web 控件需要控件名，而 type 用来指示控件的类型，locator 传入的是 XPath 对象，用于定位控件。关于 XPath 的详细内容可参考《QT4W 使用文档》。

至于更多的控件类型如容器类控件标识、容器类控件标识等请参考《Web 控件标识》进行使用。

当 webview 封装完成之后，就可以使用封装好的 webview 来对需要进行测试的 Web 页面进行封装，这里需要通过 Inspect 工具获取内嵌页面的容器的控件属性：



获取到控件属性之后，编写对应的 QPath，实例化一个 Control 对象作为我们所封装的 WebPage 的容器控件：

```
from qt4c.webcontrols import WebPage, WebElement, XPath
from qt4c.qpath import QPath

class BrowserPage(WebPage):
    '''Demo 页面'''

    def __init__(self, locator):
        qp = QPath("/ClassName='wxWindowNR' && Text='PyBrowser' && Visible='True' /
↳ClassName='Internet Explorer_Server' && MaxDepth='4'")
        self._win = Webkit(locator=qp)
        WebPage.__init__(self, self._win)
        ui_map = {
```

(下页继续)

```
        'title': XPath('//div[@class="panel-heading"]'),
        'name': {'type': InputElement, 'locator': XPath('//input[@id="name"]')},
        'female': XPath('//input[@value="female"]'),
        'male': XPath('//input[@value="male"]'),
        'age': {'type': SelectElement, 'locator': XPath('//select[@id="age"]')},
        'company': {'type': InputElement, 'locator': XPath('//input[@id="company"]')}
    },

    'submit': XPath('//button[@id="submit"]'),
}

self.update_ui_map(ui_map)

def set_name(self, name):
    ''' 设置姓名
    '''
    self.control('name').value = name

def set_female(self):
    ''' 设置性别为女性
    '''
    self.control('female').click()

def set_male(self):
    ''' 设置性别为男性
    '''
    self.control('male').click()

def set_age(self, age):
    ''' 设置年龄
    '''
    self.control('age').selection = age

def set_company(self, company):
    ''' 设置公司名
    '''
    self.control('company').value = company

def submit(self):
    ''' 提交
    '''
```

(续上页)

```
self.control("submit").click()
```

初始化 `BrowserPage` 对象之后，就可以参考上面 `Browser` 类的介绍使用封装好的 `WebPage` 对 Web 页面进行操作了：

```
pybrowserApp = PyBrowserApp()
pybrowserEmbedPage = BrowserPage(pybrowserApp)
pybrowserEmbedPage.set_name('qta')
```

这里进行的操作是，在应用程序的内嵌页面中设置名称为“qta”。

## 1.6.2 使用 Browser 类封装基础页面标识

在 QT4C 中封装了对 Chrome、IE 浏览器的支持，你可以通过调用 qt4w 的 `Browser` 类来进行使用。要想使用 `Browser` 类来封装 web 基础页面标识，你同样需要对 `WebPage` 和 `Webelement` 进行封装。

### WebPage 的封装和使用

根据《Web 控件的标识与使用》中的 demo 范例，我们封装一个 `WebPage`：

```
from qt4c.webcontrols import XPath
from qt4c.webcontrols import WebPage, WebElement

class DemoPage(WebPage):
    ''' 登录页面
    '''

    ui_map = {
        'title': XPath('//div[@class="panel-heading"]'),
        'name': {'type': InputElement, 'locator': XPath('//input[@id="name"]')},
        'female': XPath('//input[@value="female"]'),
        'male': XPath('//input[@value="male"]'),
        'age': {'type': SelectElement, 'locator': XPath('//select[@id="age"]')},
        'company': {'type': InputElement, 'locator': XPath('//input[@id="company"]')},
        'submit': XPath('//button[@id="submit"]'),
    }

    def set_name(self, name):
        ''' 设置姓名
        '''
        self.control('name').value = name
```

(下页继续)

```
def set_female(self):
    ''' 设置性别为女性
    '''
    self.control('female').click()

def set_male(self):
    ''' 设置性别为男性
    '''
    self.control('male').click()

def set_age(self, age):
    ''' 设置年龄
    '''
    self.control('age').selection = age

def set_company(self, company):
    ''' 设置公司名
    '''
    self.control('company').value = company

def submit(self):
    ''' 提交
    '''
    self.control("submit").click()
```

接下来, 就可以调用 Browser 类来打开一个 Web 页面, 直接获取一个指定的 WebPage 对象:

```
from demolib.demopage import DemoPage
from qt4w.browser import Browser

Browser.register_browser('Chrome', 'browser.chrome.ChromeBrowser')
browser = Browser("Chrome")          # 指定浏览器
page = browser.open_url('https://qtacore.github.io/qt4w/demo.html', DemoPage)  # 打开网页, 返回指定的 WebPage 页
page.set_name('qta')
```

这里需要指出的是, 在使用 Browser(“browsername”) 获取浏览器对象时, 需要先使用 register\_browser() 注册一下, 才能使用, 此处注册一次即可, 具体用法参考《跨端跨平台测试》。

除了自定义的方法, QT4W 还提供了对以下操作的支持:

- 1、基本属性的定义及操作，包括页面的 URL、Title、ReadyState、cookie 等
- 2、页面滑动操作
- 3、查找元素：find\_element 和 find\_elements
- 4、其他操作：执行 JS 接口 (eval\_script)、激活窗口 activate() 以及 upload\_file() 等

此外，关于 Browser 类的更多接口，请参考 browser\_package 对应的浏览器类型进行使用。

## 1.7 API Reference

This page contains auto-generated API reference documentation<sup>1</sup>.

### 1.7.1 qt4c

QT4C(Client Driver for QTA)

#### Subpackages

`qt4c.webview`

IWebView 接口定义及 Windows 平台上的实现

#### Subpackages

`qt4c.webview.chromewebview`

Chrome WebView 实现

#### Submodules

`qt4c.webview.chromewebview.chromedriver`

Chrome 浏览器驱动

#### Module Contents

#### Classes

---

<sup>1</sup> Created with sphinx-autoapi

<i>ChromeDriver</i>	Chrome 驱动
<i>WebkitDebugger</i>	Webkit 调试器

**exception** qt4c.webview.chromewebview.chromedriver.ChromeDriverError(*code*, *msg*)

Bases: RuntimeError

Chrome 驱动错误

**code**

**message**

**\_\_str\_\_**(*self*)

Return str(self).

**class** qt4c.webview.chromewebview.chromedriver.ChromeDriver(*port*)

Bases: object

Chrome 驱动

**inst\_dict**

**get\_page\_list**(*self*)

获取打开的页面列表

**get\_debugger**(*self*, *url=None*, *title=None*)

获取 Web 调试器

**class** qt4c.webview.chromewebview.chromedriver.WebkitDebugger(*ws\_addr*)

Bases: object

Webkit 调试器

**\_\_del\_\_**(*self*)

**on\_open**(*self*, *ws*)

**on\_message**(*self*, *ws*, *message*)

收到消息

**on\_error**(*self*, *ws*, *error*)

**on\_close**(*self*, *ws*)

**\_wait\_for\_ready**(*self*, *timeout=10*, *interval=0.1*)

等待 WebSocket 连接

**\_init**(*self*)

初始化

**\_get\_context\_id**(*self*, *frame\_id*)

获取 contextId



```
on_recv_notify_msg(self, method, params)
    接收到通知消息

work_thread(self)
    工作线程

_wait_for_response(self, seq, timeout=600, interval=0.1)
    等待返回数据

send_request(self, method, **kws)
    发送请求

    参数 method (string) - 命令字

enable_runtime(self)

get_frame_tree(self)
    获取 frame 树

eval_script(self, frame_id, script)
    执行 JavaScript

screenshot(self)
    通过 Chrome 开发者协议获取 page 页面截图,
```

Package Contents

Classes

<i>WebViewBase</i>	PC 端 WebView 基类
<i>ChromeDriver</i>	Chrome 驱动
<i>ChromewebView</i>	Chrome WebView 实现

Functions

<i>_get_pid_by_port</i> (port)	利用端口，获取对应端口的进程 id
<i>get_pid_by_port</i> (port)	增加延时和重试机制，防止网络初始化太慢导致的查找失败

```
class qt4c.webview.chromewebview.WebViewBase(window, webdriver, offscreen_win=None)
    Bases: qt4w.webview.webview.IWebView

    PC 端 WebView 基类

    browser_type
```

**rect**

当前可见窗口的坐标信息

**\_\_getattr\_\_(self, attr)**

转发给 WebDriver 实现

**\_handle\_result(self, result, frame\_xpaths)**

处理执行 JavaScript 的结果

**参数**

- **result** (*string*) – 要处理的数据
- **frame\_xpaths** (*list*) – 执行 js 所在 frame 的 xpath

**\_handle\_offset(self, x\_offset, y\_offset)**

win10 上如果设置了 DPI 需要进行坐标修正

**\_inner\_click(self, flag, click\_type, x\_offset, y\_offset)**

**\_inner\_long\_click(self, flag, x\_offset, y\_offset, duration)**

**click(self, x\_offset, y\_offset)**

**double\_click(self, x\_offset, y\_offset)**

**right\_click(self, x\_offset, y\_offset)**

**long\_click(self, x\_offset, y\_offset, duration=1)**

**hover(self, x\_offset, y\_offset)**

**scroll(self, backward=True)**

**send\_keys(self, keys)**

**activate(self, is\_true=True)**

激活当前窗口

**参数 is\_true** (*bool*) – 是否激活，默认为 True

**screenshot(self)**

当前 WebView 的截图: return: PIL.Image

**upload\_file(self, file\_path)**

**class qt4c.webview.chromewebview.ChromeDriver(port)**

Bases: object

Chrome 驱动

**inst\_dict**

**get\_page\_list(self)**

获取打开的页面列表

`get_debugger(self, url=None, title=None)`

获取 Web 调试器

`class qt4c.webview.chromewebview.ChromeWebView(page_wnd, url, pid, port=9200)`

Bases: `qt4c.webview.base.WebViewBase`

Chrome WebView 实现

`_get_frame(self, parent, name, url)`

根据 frame 的 name 和 url 获取 frameTree 节点

:param parent 要查找的 frameTree 节点:type parent dict :param name: frame 的 id 或 name 属性:type name: string :param url: frame 的 url :type url: string

`_get_frame_id_by_xpath(self, frame_xpaths)`

根据 XPath 对象查找 frame id

**参数** `frame_xpaths (list)` – frame 的 xpath 数组

`eval_script(self, frame_xpaths, script)`

在指定 frame 中执行 JavaScript, 并返回执行结果

**参数**

- **frame\_xpaths (list or string)** – frame 元素的 XPATH 路径, 如果是顶层页面, 则传入 “[]” 或者是 frame id
- **script (string)** – 要执行的 JavaScript 语句

`click(self, x_offset, y_offset)`

Chrome 中按住 shift 键点击, 以便在新窗口中打开页面

`qt4c.webview.chromewebview._get_pid_by_port(port)`

利用端口, 获取对应端口的进程 id

`qt4c.webview.chromewebview.get_pid_by_port(port)`

增加延时和重试机制, 防止网络初始化太慢导致的查找失败

`qt4c.webview.iewebview`

IE WebView 实现

## Submodules

`qt4c.webview.iewebview.iedriver`

IE 驱动模块

IE 中的坑: 1、为避免用户传入的 js 存在语法错误, 使用 eval 方式执行; 这种方式可以获得最后一句话的返回值 2、eval 中使用 var xxx=123; 不能定义变量, 需要使用 window[ 'xxx' ] = 123; 改成使用 window.eval 可以解决, ie8 还是不行 3、eval 中使用 function xx(){ } 不能定义函数, 需要加上 window[ 'xx' ] = xx;

## Module Contents

### Classes

---

<i>IEDriver</i>	window[ 'qt4w_driver_lib' ]
-----------------	-----------------------------

---

qt4c.webview.iwebview.iedriver.SID\_SWebBrowserApp

exception qt4c.webview.iwebview.iedriver.IEDriverError

Bases: RuntimeError

class qt4c.webview.iwebview.iedriver.IEDriver(*ie\_server\_hwnd*)

Bases: object

window[ 'qt4w\_driver\_lib' ]

*\_init\_com\_obj(self)*

初始化 com 对象

*\_retry\_for\_access\_denied(self, func)*

IE 中经常出现可重试解决的 80070005 错误

*\_check\_valid(self)*

检查 com 对象的有效性

*\_get\_document(self, frame)*

获取 frame 的 IHTMLDocument2 指针, 此方法可以跨域

*get\_frames(self, doc)*

*get\_frame\_window(self, win, frame\_id, url)*

获取 doc 中 id 或 name 为 frame\_id, 或者 url 匹配的 frame 的 IHTMLWindow 对象

*handle\_error\_page(self, doc)*

处理错误页面

*eval\_script(self, frame\_win, script, use\_eval=True)*

IE10 以上异常对象才有 stack 属性

## Package Contents

## Classes

<i>IEDriver</i>	window[ 'qt4w_driver_lib' ]
<i>WebViewBase</i>	PC 端 WebView 基类
<i>Mouse</i>	鼠标操作静态类
<i>MouseFlag</i>	鼠标按钮枚举类
<i>MouseClickType</i>	鼠标点击枚举类
<i>IEWebView</i>	IE WebView 实现

```
class qt4c.webview.iewebview.IEDriver(ie_server_hwnd)
    Bases: object
    window[ 'qt4w_driver_lib' ]

    _init_com_obj(self)
        初始化 com 对象

    _retry_for_access_denied(self, func)
        IE 中经常出现可重试解决的 80070005 错误

    _check_valid(self)
        检查 com 对象的有效性

    _get_document(self, frame)
        获取 frame 的 IHTMLDocument2 指针，此方法可以跨域

    get_frames(self, doc)

    get_frame_window(self, win, frame_id, url)
        获取 doc 中 id 或 name 为 frame_id，或者 url 匹配的 frame 的 IHTMLWindow 对象

    handle_error_page(self, doc)
        处理错误页面

    eval_script(self, frame_win, script, use_eval=True)
        IE10 以上异常对象才有 stack 属性

class qt4c.webview.iewebview.WebViewBase(window, webdriver, offscreen_win=None)
    Bases: qt4w.webview.webview.IWebView

    PC 端 WebView 基类

    browser_type

    rect
        当前可见窗口的坐标信息

    __getattr__(self, attr)
        转发给 WebDriver 实现
```

`_handle_result(self, result, frame_xpaths)`

处理执行 JavaScript 的结果

**参数**

- **result** (*string*) – 要处理的数据
- **frame\_xpaths** (*list*) – 执行 js 所在 frame 的 xpath

`_handle_offset(self, x_offset, y_offset)`

win10 上如果设置了 DPI 需要进行坐标修正

`_inner_click(self, flag, click_type, x_offset, y_offset)`

`_inner_long_click(self, flag, x_offset, y_offset, duration)`

`click(self, x_offset, y_offset)`

`double_click(self, x_offset, y_offset)`

`right_click(self, x_offset, y_offset)`

`long_click(self, x_offset, y_offset, duration=1)`

`hover(self, x_offset, y_offset)`

`scroll(self, backward=True)`

`send_keys(self, keys)`

`activate(self, is_true=True)`

激活当前窗口

**参数** **is\_true** (*bool*) – 是否激活，默认为 True

`screenshot(self)`

当前 WebView 的截图: return: PIL.Image

`upload_file(self, file_path)`

`class qt4c.webview.iewebview.Mouse`

Bases: object

鼠标操作静态类

`_last_click_time`

`static handle_position(x, y)`

坐标转换

`static click(x, y, flag=MouseFlag.LeftButton, clicktype=MouseClickType.SingleClick)`

鼠标点击 (x,y) 点

**参数**

- **x** (*int*) – 屏幕 x 坐标

- `y (int)` – 屏幕 `y` 坐标
- `mouseFlag (qt4c.mouse.MouseFlag)` – 鼠标按钮
- `clickType (qt4c.mouse.MouseClickType)` – 鼠标动作, 如双击还是单击

`static _clickSlowly(x, y, flag=MouseFlag.LeftButton, interval=0.1)`

模拟鼠标缓慢点击, 在鼠标键按下和释放之间存在一个 `interval` 的时间间隔

`static sendClick(hwnd, x, y, flag=MouseFlag.LeftButton, click-  
type=MouseClickType.SingleClick)`

在目标窗口通过 `SendMessage` 方式产生鼠标点击的动作

#### 参数

- `hwnd` (整数) – 目标窗口句柄
- `x` (整数) – 屏幕 `x` 坐标
- `y` (整数) – 屏幕 `y` 坐标
- `flag` (枚举类型, `MouseFlag.LeftButton/MouseFlag.MiddleButton/MouseFlag.RightButton`) – 鼠标键类型
- `clicktype` (枚举类型, `MouseClickType.SingleClick / MouseClickType.DoubleClick`) – 鼠标键点击方式

`static postClick(hwnd, x, y, flag=MouseFlag.LeftButton, click-  
type=MouseClickType.SingleClick)`

在目标窗口通过 `PostMessage` 的方式产生鼠标点击的动作

#### 参数

- `hwnd` (整数) – 目标窗口句柄
- `x` (整数) – 屏幕 `x` 坐标
- `y` (整数) – 屏幕 `y` 坐标
- `flag` (枚举类型, `MouseFlag.LeftButton/MouseFlag.MiddleButton/MouseFlag.RightButton`) – 鼠标键类型
- `clicktype` (枚举类型, `MouseClickType.SingleClick / MouseClickType.DoubleClick`) – 鼠标键点击方式

`static sendNCClick(hwnd, x, y, flag=MouseFlag.LeftButton, click-  
type=MouseClickType.SingleClick)`

在目标窗口的 `Non-Client` 区域通过发消息的方式产生鼠标点击的动作

#### 参数

- `hwnd` (整数) – 目标窗口句柄
- `x` (整数) – 屏幕 `x` 坐标
- `y` (整数) – 屏幕 `y` 坐标

- **flag** (枚举类型, *MouseFlag.LeftButton*/*MouseFlag.MiddleButton*/*MouseFlag.RightButton*) – 鼠标键类型
- **clicktype** (枚举类型, *MouseClickType.SingleClick* / *MouseClickType.DoubleClick*) – 鼠标键点击方式

**static drag**(*fromX*, *fromY*, *toX*, *toY*, *flag*=*MouseFlag.LeftButton*, *intervaltime*=1)

鼠标从 (*fromX*, *fromY*) 拖拽到 (*toX*, *toY*)

#### 参数

- **fromX** (整数) – 屏幕 x 坐标
- **fromY** (整数) – 屏幕 y 坐标
- **toX** (整数) – 屏幕 x 坐标
- **toY** (整数) – 屏幕 y 坐标
- **flag** (枚举类型, *MouseFlag.LeftButton*/*MouseFlag.MiddleButton*/*MouseFlag.RightButton*) – 鼠标键类型

**static press**(*x*, *y*, *flag*=*MouseFlag.LeftButton*)

在某个位置按下鼠标键

#### 参数

- **x** (整数) – 屏幕 x 坐标
- **y** (整数) – 屏幕 y 坐标
- **flag** (枚举类型, *MouseFlag.LeftButton*/*MouseFlag.MiddleButton*/*MouseFlag.RightButton*) – 鼠标键类型

**static release**(*x*, *y*, *flag*=*MouseFlag.LeftButton*)

在某个位置释放鼠标键, 与 **press** 配对使用

#### 参数

- **x** (整数) – 屏幕 x 坐标
- **y** (整数) – 屏幕 y 坐标
- **flag** (枚举类型, *MouseFlag.LeftButton*/*MouseFlag.MiddleButton*/*MouseFlag.RightButton*) – 鼠标键类型

**static postMove**(*hwnd*, *toX*, *toY*)

**static sendMove**(*hwnd*, *toX*, *toY*)

**static move**(*toX*, *toY*)

鼠标移动到 (*toX*, *toY*)

#### 参数

- **x** (*int*) – 屏幕 x 坐标



- `y (int)` – 屏幕 `y` 坐标

`static getPosition()`

当前 Mouse 的位置

`static getCursorType()`

返回当前鼠标图标类型

返回类型 *MouseCursorType*

`static scroll(bForward=False)`

鼠标滚动 `bForward`: `True` 则向前滚动, `False` 则向后滚动。默认是 `False`。

`class qt4c.webview.iwebview.MouseFlag`

Bases: `object`

鼠标按键枚举类

`class qt4c.webview.iwebview.MouseClickType`

Bases: `object`

鼠标点击枚举类

`class qt4c.webview.iwebview.IEWebView(ie_window_or_hwnd)`

Bases: *qt4c.webview.base.WebViewBase*

IE WebView 实现

`_get_frame_window_by_xpath(self, frame_xpaths)`

根据 `xpath` 查找对应的 `frameHTMLWindow` 对象

`eval_script(self, frame_xpaths, script, use_eval=True)`

在指定 `frame` 中执行 JavaScript, 并返回执行结果

参数

- `frame_xpaths (list)` – `frame` 元素的 XPATH 路径, 如果是顶层页面, 怎传入 “[]”
- `script (string)` – 要执行的 JavaScript 语句

`highlight(self, elem_xpaths)`

使元素高亮

参数 `elem_xpaths (list)` – 元素的 XPATH 路径

## Submodules

*qt4c.webview.base*

PC 端 WebView 基类

## Module Contents

### Classes

<i>WebViewBase</i>	PC 端 WebView 基类
<i>UploadFileDialog</i>	上传文件对话框，目前 ie、chrome 封装是一样的，故放在这里，后面如果有不同，

```
class qt4c.webview.base.WebViewBase(window, webdriver, offscreen_win=None)
```

Bases: qt4w.webview.webview.IWebView

PC 端 WebView 基类

**browser\_type**

**rect**

当前可见窗口的坐标信息

**\_\_getattr\_\_(self, attr)**

转发给 WebDriver 实现

**\_handle\_result(self, result, frame\_xpaths)**

处理执行 JavaScript 的结果

**参数**

- **result** (*string*) – 要处理的数据
- **frame\_xpaths** (*list*) – 执行 js 所在 frame 的 xpath

**\_handle\_offset(self, x\_offset, y\_offset)**

win10 上如果设置了 DPI 需要进行坐标修正

**\_inner\_click(self, flag, click\_type, x\_offset, y\_offset)**

**\_inner\_long\_click(self, flag, x\_offset, y\_offset, duration)**

**click(self, x\_offset, y\_offset)**

**double\_click(self, x\_offset, y\_offset)**

**right\_click(self, x\_offset, y\_offset)**

**long\_click(self, x\_offset, y\_offset, duration=1)**

**hover(self, x\_offset, y\_offset)**

**scroll(self, backward=True)**

**send\_keys(self, keys)**

`activate(self, is_true=True)`

激活当前窗口

参数 `is_true (bool)` – 是否激活, 默认为 `True`

`screenshot(self)`

当前 WebView 的截图: return: `PIL.Image`

`upload_file(self, file_path)`

`class qt4c.webview.base.UploadFileDialog(process_id)`

Bases: `qt4c.filedialog.FileDialog`

上传文件对话框, 目前 ie、chrome 封装是一样的, 故放在这里, 后面如果有不同,

`upload_file(self, file_path)`

上传文件

## Package Contents

`qt4c.webview.fmt`

`qt4c.webview.handler`

## Submodules

`qt4c.accessible`

用于访问支持 `IAccessible` 接口的控件

## Module Contents

### Classes

<code>EnumAccessibleObjectRole</code>	Accessible Object 的角色
<code>EnumAccessibleObjectState</code>	Accessible Object 的状态
<code>_AccessibleObjectWrapper_win32com</code>	使用 win32com 模块实现 <code>IAccessible</code> 接口的包裹类
<code>_AccessibleObjectWrapper_comtypes</code>	使用 comtypes 模块实现 <code>IAccessible</code> 接口的包裹类
<code>AccessibleObject</code>	支持 <code>IAccessible</code> 接口的对象 (控件) 的包裹类

`class qt4c.accessible.EnumAccessibleObjectRole`

Bases: `object`

Accessible Object 的角色

```

ROLE_SYSTEM_TITLEBAR = 1
ROLE_SYSTEM_MENUBAR = 2
ROLE_SYSTEM_SCROLLBAR = 3
ROLE_SYSTEM_GRIP = 4
ROLE_SYSTEM_SOUND = 5
ROLE_SYSTEM_CURSOR = 6
ROLE_SYSTEM_CARET = 7
ROLE_SYSTEM_ALERT = 8
ROLE_SYSTEM_WINDOW = 9
ROLE_SYSTEM_CLIENT = 10
ROLE_SYSTEM_MENUPOPUP = 11
ROLE_SYSTEM_MENUITEM = 12
ROLE_SYSTEM_TOOLTIP = 13
ROLE_SYSTEM_APPLICATION = 14
ROLE_SYSTEM_DOCUMENT = 15
ROLE_SYSTEM_PANE = 16
ROLE_SYSTEM_CHART = 17
ROLE_SYSTEM_DIALOG = 18
ROLE_SYSTEM_BORDER = 19
ROLE_SYSTEM_GROUPING = 20
ROLE_SYSTEM_SEPARATOR = 21
ROLE_SYSTEM_TOOLBAR = 22
ROLE_SYSTEM_STATUSBAR = 23
ROLE_SYSTEM_TABLE = 24
ROLE_SYSTEM_COLUMNHEADER = 25
ROLE_SYSTEM_ROWHEADER = 26
ROLE_SYSTEM_COLUMN = 27
ROLE_SYSTEM_ROW = 28
ROLE_SYSTEM_CELL = 29
ROLE_SYSTEM_LINK = 30

```

ROLE\_SYSTEM\_HELPBALLOON = 31

ROLE\_SYSTEM\_CHARACTER = 32

ROLE\_SYSTEM\_LIST = 33

ROLE\_SYSTEM\_LISTITEM = 34

ROLE\_SYSTEM\_OUTLINE = 35

ROLE\_SYSTEM\_OUTLINEITEM = 36

ROLE\_SYSTEM\_PAGETAB = 37

ROLE\_SYSTEM\_PROPERTYPAGE = 38

ROLE\_SYSTEM\_INDICATOR = 39

ROLE\_SYSTEM\_GRAPHIC = 40

ROLE\_SYSTEM\_STATICTEXT = 41

ROLE\_SYSTEM\_TEXT = 42

ROLE\_SYSTEM\_PUSHBUTTON = 43

ROLE\_SYSTEM\_CHECKBUTTON = 44

ROLE\_SYSTEM\_RADIOBUTTON = 45

ROLE\_SYSTEM\_COMBOBOX = 46

ROLE\_SYSTEM\_DROPLIST = 47

ROLE\_SYSTEM\_PROGRESSBAR = 48

ROLE\_SYSTEM\_DIAL = 49

ROLE\_SYSTEM\_HOTKEYFIELD = 50

ROLE\_SYSTEM\_SLIDER = 51

ROLE\_SYSTEM\_SPINBUTTON = 52

ROLE\_SYSTEM\_DIAGRAM = 53

ROLE\_SYSTEM\_ANIMATION = 54

ROLE\_SYSTEM\_EQUATION = 55

ROLE\_SYSTEM\_BUTTONDROPDOWN = 56

ROLE\_SYSTEM\_BUTTONMENU = 57

ROLE\_SYSTEM\_BUTTONDROPDOWNGRID = 58

ROLE\_SYSTEM\_WHITESPACE = 59

ROLE\_SYSTEM\_PAGETABLIST = 60

```
ROLE_SYSTEM_CLOCK = 61
```

```
ROLE_SYSTEM_SPLITBUTTON = 62
```

```
ROLE_SYSTEM_IPADDRESS = 63
```

```
ROLE_SYSTEM_OUTLINEBUTTON = 64
```

```
class qt4c.accessible.EnumAccessibleObjectState
```

```
    Bases: object
```

```
    Accessible Object 的状态
```

```
    STATE_SYSTEM_UNAVAILABLE = 1
```

```
    STATE_SYSTEM_SELECTED = 2
```

```
    STATE_SYSTEM_FOCUSED = 4
```

```
    STATE_SYSTEM_PRESSED = 8
```

```
    STATE_SYSTEM_CHECKED = 16
```

```
    STATE_SYSTEM_MIXED = 32
```

```
    STATE_SYSTEM_INDETERMINATE
```

```
    STATE_SYSTEM_READONLY = 64
```

```
    STATE_SYSTEM_HOTTRACKED = 128
```

```
    STATE_SYSTEM_DEFAULT = 256
```

```
    STATE_SYSTEM_EXPANDED = 512
```

```
    STATE_SYSTEM_COLLAPSED = 1024
```

```
    STATE_SYSTEM_BUSY = 2048
```

```
    STATE_SYSTEM_FLOATING = 4096
```

```
    STATE_SYSTEM_MARQUEED = 8192
```

```
    STATE_SYSTEM_ANIMATED = 16384
```

```
    STATE_SYSTEM_INVISIBLE = 32768
```

```
    STATE_SYSTEM_OFFSCREEN = 65536
```

```
    STATE_SYSTEM_SIZEABLE = 131072
```

```
    STATE_SYSTEM_MOVEABLE = 262144
```

```
    STATE_SYSTEM_SELFVOICING = 524288
```

```
    STATE_SYSTEM_FOCUSABLE = 1048576
```

```
    STATE_SYSTEM_SELECTABLE = 2097152
```

```

STATE_SYSTEM_LINKED = 4194304
STATE_SYSTEM_TRAVERSED = 8388608
STATE_SYSTEM_MULTISELECTABLE = 16777216
STATE_SYSTEM_EXTSELECTABLE = 33554432
STATE_SYSTEM_ALERT_LOW = 67108864
STATE_SYSTEM_HASSUBMENU
STATE_SYSTEM_ALERT_MEDIUM = 134217728
STATE_SYSTEM_ALERT_HIGH = 268435456
STATE_SYSTEM_PROTECTED = 536870912
STATE_SYSTEM_VALID = 1073741823
STATE_SYSTEM_HASPOPUP = 1073741824

```

```
class qt4c.accessible._AccessibleObjectWrapper_win32com(acc__disp)
```

Bases: object

使用 win32com 模块实现 IAccessible 接口的包裹类

`accChildCount`

`accFocus`

`accName`

`accRole`

`accDescription`

`accState`

`accValue`

`accParent`

`get_accName(self, childID)`

```
class qt4c.accessible._AccessibleObjectWrapper_comtypes(acc__disp)
```

Bases: object

使用 comtypes 模块实现 IAccessible 接口的包裹类

`accChildCount`

`accFocus`

`accName`

`accRole`

`accDescription`

`accState`

`accValue`

`accParent`

`_accessible_object_from_window(self, hwnd)`

返回句柄指定的 AccessibleObject

**参数** `hwnd (int)` – 句柄

**Raises** ValueError

**返回类型** `comtypes.gen.Accessibility.IAccessible`

`_accessible_object_from_point(self, pt)`

返回坐标对应的 AccessibleObject

**参数** `pt (tuple)` – (x,y), 相对于桌面的坐标

**Raises** ValueError

**返回类型** `comtypes.gen.Accessibility.IAccessible`

`get_accName(self, childID)`

`class qt4c.accessible.AccessibleObject(acc_disp)`

Bases: object

支持 IAccessible 接口的对象（控件）的包裹类

`accFocus`

获取具有焦点的控件

**返回类型** `int` or *AccessibleObject* or None

**返回** 如果返回为 0 代表具有焦点的控件是其本身，返回类型为整数，则代表其获得焦点的子控件的控件 ID；返回类型为 AccessibleObject，则代表其获得焦点的子控件实例；返回为 None，代表未实现此接口。

`accName`

获取名称

**返回类型** `string`

`accRole`

获取角色

**返回类型** *Enum.AccessibleObjectRole*

`accDescription`

获取描述

**返回类型** `string`



`accState`  
获取状态值  
  
返回类型 *EnumAccessibleObjectState*

`accValue`  
获取值  
  
返回类型 `string`

`accParent`  
获取父控件  
  
返回类型 *AccessibleObject*

`accChildCount`

`get_accName(self, childID=None)`

`qt4c.app`

应用程序基类模块

Module Contents

Classes

---

<i>App</i>	应用程序基类
------------	--------

---

```
class qt4c.app.App
    Bases: object

    应用程序基类

    _totalapps = []

    quit(self)
        请在子类中实现，并调用此方法通知程序退出

    static quitAll()
        退出所有应用程序

    static clearAll()
        清除所有程序记录

    static killAll()
        结束所有记录的进程
```

qt4c.control

控件基类模块

Module Contents

Classes

<i>Control</i>	控件基类
<i>ControlContainer</i>	控件集合接口

class qt4c.control.Control

Bases: object

控件基类

`_timeout`

`Children`

返回此控件的子控件。需要在子类中实现。

`BoundingRect`

返回窗口大小。未实现！

`_click(self, mouseFlag, clickType, xOffset, yOffset)`

点击控件

参数

- `mouseFlag` (`qt4c.mouse.MouseFlag`) – 鼠标按钮
- `clickType` (`qt4c.mouse.MouseClickType`) – 鼠标动作
- `xOffset` (`int`) – 距离控件区域左上角的偏移。默认值为 `None`，代表控件区域 `x` 轴上的中点；如果为负值，代表距离控件区域右边的绝对值偏移；
- `yOffset` (`int`) –  
距离控件区域左上角的偏移。默认值为 `None`，代表控件区域 `y` 轴上的中点；  
如果为负值，代表距离控件区域上边的绝对值偏移；

`click(self, mouseFlag=MouseFlag.LeftButton, clickType=MouseClickType.SingleClick, xOffset=None, yOffset=None)`

点击控件

参数

- `mouseFlag` (`qt4c.mouse.MouseFlag`) – 鼠标按钮

- **clickType** (`qt4c.mouse.MouseClickType`) – 鼠标动作
- **xOffset** (*int*) – 距离控件区域左上角的偏移。默认值为 `None`，代表控件区域 x 轴上的中点。如果为负值，代表距离控件区域右上角的 x 轴上的绝对值偏移。
- **yOffset** (*int*) – 距离控件区域左上角的偏移。默认值为 `None`，代表控件区域 y 轴上的中点。如果为负值，代表距离控件区域右上角的 y 轴上的绝对值偏移。

**\_getClickXY**(*self*, *xOffset*, *yOffset*)

通过指定的偏移值确定具体要点击的 x,y 坐标

**doubleClick**(*self*, *xOffset=None*, *yOffset=None*)

左键双击，参数参考 click 函数

**hover**(*self*)

鼠标移动到该控件上

**rightClick**(*self*, *xOffset=None*, *yOffset=None*)

右键双击，参数参考 click 函数

**drag**(*self*, *toX*, *toY*)

拖拽控件到指定位置

**scroll**(*self*, *backward=True*)

发送鼠标滚动命令

**sendKeys**(*self*, *keys*)

发送按键命令

**setFocus**(*self*)

设控件为焦点

**waitForValue**(*self*, *prop\_name*, *prop\_value*, *timeout=10*, *interval=0.5*, *regularMatch=False*)

等待控件属性值出现，如果属性为字符串类型，则使用正则匹配

#### 参数

- **prop\_name** – 属性名字
- **prop\_value** – 等待出现的属性值
- **timeout** – 超时秒数，默认为 10
- **interval** – 等待间隔，默认为 0.5
- **regularMatch** – 参数 `property_name` 和 `waited_value` 是否采用正则表达式的比较。默认为不采用 (`False`) 正则，而是采用恒等比较。

**equal**(*self*, *other*)

判断两个对象是否相同。未实现!

参数 **other** (`Control`) – 本对象实例

`__eq__(self, other)`  
重载对象恒等操作符 (==)

`__ne__(self, other)`  
重载对象不等操作符 (!=)

`get_metis_view(self)`  
返回 MetisView

`class qt4c.control.ControlContainer`

Bases: object

控件集合接口

当一个类继承本接口，并设置 Locator 属性后，该类可以使用 Controls 属性获取控件。如

`>>>class SysSettingWin(uia.UIAWindows, ControlContainer)`

`def __init__(self):`

`locators={ '常规页' : { 'type' :uia.Control, 'root' :self, 'locator' =' PageBasicGeneral' },`  
`'退出程序单选框' : { 'type' :uia.RadioButton, 'root' : ' @ 常规页', 'locator' =QPath(`  
`"/name=' ExitProgramme_RI' && maxdepth=' 10'" )}}`

`self.updateLocator(locators)`

则 SysSettingWin().Controls[ '常规页' ] 返回设置窗口上常规页的 uia.Control 实例，而 SysSettingWin().Controls[ '退出程序单选框' ]，返回设置窗口的常规页下的退出程序单选框实例。其中 'root' = ' @ 常规页' 中的 ' @ 常规页' 表示参数 'root' 的值不是这个字符串，而是 key '常规页' 指定的控件。

**Controls**

返回控件集合。使用如 foo.Controls[ '最小化按钮' ] 的形式获取控件

`__findctrl_recur(self, ctrlkey)`

`__getitem__(self, index)`

获取 index 指定控件

**参数** index (string) – 控件索引，如 '查找按钮'

`clearLocator(self)`

清空控件定位参数

`hasControlKey(self, control_key)`

是否包含控件 control\_key

**返回类型** boolean

`updateLocator(self, locators)`

更新控件定位参数

**参数** `locators` (*dict*) – 定位参数, 格式是 { ‘控件名’ : { ‘type’ : 控件类, 控件类的参数 dict 列表}, ... }

`isChildCtrlExist(self, childctrlname)`

判断指定名字的子控件是否存在

**参数** `childctrlname` (*str*) – 指定的子控件名称

**返回类型** `boolean`

qt4c.exceptions

异常模块定义

qt4c.filedialog

文件窗口模块

Module Contents

Classes

<i>FileDialog</i>	文件窗口基类
<i>OpenFileDialog</i>	打开文件窗口
<i>SelectFileDialog</i>	选择文件/文件夹窗口, 适用于 1.90 之后发送文件 打开的窗口
<i>SaveAsDialog</i>	另存为窗口
<i>BrowseDialog</i>	浏览文件夹窗口
<i>BrowseFolderDialog</i>	浏览文件夹窗口
<i>FileFolder</i>	打开文件窗口
<i>ExploreFileFolder</i>	XP 左边有文件树型结构的文件窗口
<i>Win7ExploreFileFolder</i>	win7 下在 aio 中打开收到文件的文件夹窗口

`class qt4c.filedialog.FileDialog(qpath=None)`

Bases: `qt4c.wincontrols.Window`, `qt4c.control.ControlContainer`

文件窗口基类

**FilePath**

返回当前文件路径。如果没有选择文件, 返回的是当前文件夹路径

`class qt4c.filedialog.OpenFileDialog`

Bases: `qt4c.filedialog.FileDialog`

打开文件窗口

`open(self, filename)`

`class qt4c.filedialog.SelectFileDialog`

Bases: `qt4c.filedialog.FileDialog`

选择文件/文件夹窗口, 适用于 1.90 之后发送文件打开的窗口

`open(self, filename)`

`class qt4c.filedialog.SaveAsDialog`

Bases: `qt4c.filedialog.FileDialog`

另存为窗口

`save(self, filepath, style=None)`

保存至路径

#### 参数

- `filepath (string)` – 要保存至的全路径
- `style (string)` – 保存类型

`class qt4c.filedialog.BrowseDialog`

Bases: `qt4c.filedialog.FileDialog`

浏览文件夹窗口

`class qt4c.filedialog.BrowseFolderDialog`

Bases: `qt4c.wincontrols.Window`

浏览文件夹窗口

`class qt4c.filedialog.FileFolder`

Bases: `qt4c.filedialog.FileDialog`

打开文件窗口

`class qt4c.filedialog.ExploreFileFolder`

Bases: `qt4c.filedialog.FileDialog`

XP 左边有文件树型结构的文件窗口

`class qt4c.filedialog.Win7ExploreFileFolder`

Bases: `qt4c.filedialog.FileDialog`

win7 下在 aio 中打开收到文件的文件夹窗口

`qt4c.keyboard`

键盘输入模块

Module Contents

Classes

<i>_KeyboardEvent</i>	
<i>Key</i>	一个按键
<i>Keyboard</i>	键盘输入类，实现了两种键盘输入方式。

Functions

<i>_scan2vkey(scan)</i>
-------------------------

```
qt4c.keyboard._SHIFT
qt4c.keyboard._MODIFIERS
qt4c.keyboard._MODIFIER_KEY_MAP
qt4c.keyboard._CODES
qt4c.keyboard._scan2vkey(scan)
class qt4c.keyboard._KeyboardEvent
    Bases: object
        KEYEVENTF_EXTENDEDKEY = 1
        KEYEVENTF_KEYUP = 2
        KEYEVENTF_UNICODE = 4
        KEYEVENTF_SCANCODE = 8
qt4c.keyboard.is_64bits
qt4c.keyboard.MAPVK_VK_TO_VSC = 0
qt4c.keyboard.ULONG_PTR
exception qt4c.keyboard.KeyInputError
    Bases: Exception
        键盘输入错误
class qt4c.keyboard.Key(key)
    Bases: object
        一个按键
```

`appendModifierKey(self, key)`

Modifier Key comes with the key

参数 **key** (*Key*) – Ctrl, Shift or Atl Key

`_isExtendedKey(self, vkey)`

`_inputKey(self, up)`

`inputKey(self)`

键盘模拟输入按键

`_postKey(self, hwnd, up)`

给某个窗口发送按钮

`postKey(self, hwnd)`

将按键消息发到 hwnd

`_isPressed(self)`

该键是否被按下

`_isToggled(self)`

该键是否被开启，如 Caps Lock 或 Num Lock 等

**class** qt4c.keyboard.Keyboard

Bases: object

键盘输入类，实现了两种键盘输入方式。

一类方法使用模拟键盘输入的方式。另一类方法使用 Windows 消息的机制将字符串直接发送的窗口。

键盘输入类支持以下字符的输入。1、特殊字符：^, +, %, {, }

‘^’ 表示 Control 键，同 ‘{CTRL}’。‘+’ 表示 Shift 键，同 ‘{SHIFT}’。‘%’ 表示 Alt 键，同 ‘{ALT}’。‘^’，‘+’，‘%’ 可以单独或同时使用，如 ‘^a’ 表示 CTRL+a，‘^%a’ 表示 CTRL+ALT+a。{}：大括号用来输入特殊字符本身和虚键，如 ‘{+}’ 输入加号，‘{F1}’ 输入 F1 虚键，‘{}}’ 表示输入 ‘}’ 字符。

2、ASCII 字符：除了特殊字符需要 {} 来转义，其他 ASCII 码字符直接输入，3、Unicode 字符：直接输入，如”测试”。4、虚键：

{F1}, {F2}, ..., {F12} {Tab}, {CAPS}, {ESC}, {BKSP}, {HOME}, {INSERT}, {DEL}, {END}, {ENTER} {PGUP}, {PGDN}, {LEFT}, {RIGHT}, {UP}, {DOWN}, {CTRL}, {SHIFT}, {ALT}, {APPS}..

注意：当使用联合键时，注意此类的问题，inputKeys( ‘^W’ ) 和 inputKeys( ‘%w’ )，字母 ‘w’ 的大小写产生的效果可能不一样

`_keyclass`

`_pressedkey`

**static** selectKeyClass(newkeyclass)

**static** \_parse\_keys(keystring)



`static inputKeys(keys, interval=0.01)`  
模拟键盘输入字符串

参数

- `keys` (*utf-8 str or unicode*) – 键盘输入字符串，可输入组合键，如”{CTRL}{MENU}a”
- `interval` (*number*) – 输入的字符和字符之间的暂停间隔。

`static postKeys(hwnd, keys, interval=0.01)`  
将字符串以窗口消息的方式发送到指定 win32 窗口。

参数

- `hwnd` (*number*) – windows 窗口句柄
- `keys` (*utf8 str 或者 unicode*) – 键盘输入字符串
- `interval` (*number*) – 输入的字符和字符之间的暂停间隔。

`static pressKey(key)`  
按下某个键

`static releaseKey(key=None)`  
释放上一个被按下的键

`static isPressed(key)`  
是否被按下

`static clear()`  
释放被按下的按键

`static isTroggled(key)`  
是否开启，如 Caps Lock 或 Num Lock 等

qt4c.mouse

鼠标操作模块

Module Contents

Classes

<i>MouseFlag</i>	鼠标按键枚举类
<i>MouseClickedType</i>	鼠标点击枚举类
<i>MouseCursorType</i>	鼠标图标枚举类型

下页继续

表 13 – 续上页

<i>Mouse</i>	鼠标操作静态类
<pre>class qt4c.mouse.MouseFlag</pre> <p>Bases: object</p> <p>鼠标按键枚举类</p>	
<pre>class qt4c.mouse.MouseClickType</pre> <p>Bases: object</p> <p>鼠标点击枚举类</p>	
<pre>class qt4c.mouse.MouseCursorType</pre> <p>Bases: object</p> <p>鼠标图标枚举类型</p>	
<pre>qt4c.mouse._cursor_flags</pre>	
<pre>qt4c.mouse._mouse_msg</pre>	
<pre>qt4c.mouse._mouse_msg_param</pre>	
<pre>qt4c.mouse._mouse_ncmsg_param</pre>	
<pre>class qt4c.mouse.Mouse</pre> <p>Bases: object</p> <p>鼠标操作静态类</p> <p><code>_last_click_time</code></p> <p><code>static handle_position(x, y)</code></p> <p>坐标转换</p> <p><code>static click(x, y, flag=MouseFlag.LeftButton, clicktype=MouseClickType.SingleClick)</code></p> <p>鼠标点击 (x,y) 点</p> <p>参数</p> <ul style="list-style-type: none"> <li>• <code>x (int)</code> – 屏幕 x 坐标</li> <li>• <code>y (int)</code> – 屏幕 y 坐标</li> <li>• <code>mouseFlag (qt4c.mouse.MouseFlag)</code> – 鼠标按钮</li> <li>• <code>clickType (qt4c.mouse.MouseClickType)</code> – 鼠标动作, 如双击还是单击</li> </ul> <p><code>static _clickSlowly(x, y, flag=MouseFlag.LeftButton, interval=0.1)</code></p> <p>模拟鼠标缓慢点击, 在鼠标键按下和释放之间存在一个 interval 的时间间隔</p> <p><code>static sendClick(hwnd, x, y, flag=MouseFlag.LeftButton, click-</code></p> <p><code>type=MouseClickType.SingleClick)</code></p> <p>在目标窗口通过 SendMessage 方式产生鼠标点击的动作</p>	

**参数**

- **hwnd** (整数) – 目标窗口句柄
- **x** (整数) – 屏幕 x 坐标
- **y** (整数) – 屏幕 y 坐标
- **flag** (枚举类型, *MouseFlag.LeftButton*/*MouseFlag.MiddleButton*/*MouseFlag.RightButton*) – 鼠标键类型
- **clicktype** (枚举类型, *MouseClickType.SingleClick* / *MouseClickType.DoubleClick*) – 鼠标键点击方式

**static postClick**(hwnd, x, y, flag=*MouseFlag.LeftButton*, click-  
type=*MouseClickType.SingleClick*)

在目标窗口通过 PostMessage 的方式产生鼠标点击的动作

**参数**

- **hwnd** (整数) – 目标窗口句柄
- **x** (整数) – 屏幕 x 坐标
- **y** (整数) – 屏幕 y 坐标
- **flag** (枚举类型, *MouseFlag.LeftButton*/*MouseFlag.MiddleButton*/*MouseFlag.RightButton*) – 鼠标键类型
- **clicktype** (枚举类型, *MouseClickType.SingleClick* / *MouseClickType.DoubleClick*) – 鼠标键点击方式

**static sendNCClick**(hwnd, x, y, flag=*MouseFlag.LeftButton*, click-  
type=*MouseClickType.SingleClick*)

在目标窗口的 Non-Client 区域通过发消息的方式产生鼠标点击的动作

**参数**

- **hwnd** (整数) – 目标窗口句柄
- **x** (整数) – 屏幕 x 坐标
- **y** (整数) – 屏幕 y 坐标
- **flag** (枚举类型, *MouseFlag.LeftButton*/*MouseFlag.MiddleButton*/*MouseFlag.RightButton*) – 鼠标键类型
- **clicktype** (枚举类型, *MouseClickType.SingleClick* / *MouseClickType.DoubleClick*) – 鼠标键点击方式

**static drag**(fromX, fromY, toX, toY, flag=*MouseFlag.LeftButton*, intervaltime=1)

鼠标从 (fromX, fromY) 拖拽到 (toX, toY)

**参数**

- **fromX** (整数) – 屏幕 x 坐标

- **fromY** (整数) – 屏幕 y 坐标
- **toX** (整数) – 屏幕 x 坐标
- **toY** (整数) – 屏幕 y 坐标
- **flag** (枚举类型, *MouseFlag.LeftButton*/*MouseFlag.MiddleButton*/*MouseFlag.RightButton*) – 鼠标键类型

**static press**(*x, y, flag=MouseFlag.LeftButton*)

在某个位置按下鼠标键

#### 参数

- **x** (整数) – 屏幕 x 坐标
- **y** (整数) – 屏幕 y 坐标
- **flag** (枚举类型, *MouseFlag.LeftButton*/*MouseFlag.MiddleButton*/*MouseFlag.RightButton*) – 鼠标键类型

**static release**(*x, y, flag=MouseFlag.LeftButton*)

在某个位置释放鼠标键, 与 **press** 配对使用

#### 参数

- **x** (整数) – 屏幕 x 坐标
- **y** (整数) – 屏幕 y 坐标
- **flag** (枚举类型, *MouseFlag.LeftButton*/*MouseFlag.MiddleButton*/*MouseFlag.RightButton*) – 鼠标键类型

**static postMove**(*hwnd, toX, toY*)

**static sendMove**(*hwnd, toX, toY*)

**static move**(*toX, toY*)

鼠标移动到 (*tox,toy*)

#### 参数

- **x** (*int*) – 屏幕 x 坐标
- **y** (*int*) – 屏幕 y 坐标

**static getPosition**()

当前 Mouse 的位置

**static getCursorType**()

返回当前鼠标图标类型

返回类型 *MouseCursorType*

**static scroll**(*bForward=False*)

鼠标滚动 *bForward*: True 则向前滚动, False 则向后滚动。默认是 False。

qt4c.qpath

qpath 模块

详见 QPath 类说明

## Module Contents

### Classes

<i>EnumQPathKey</i>	
<i>EnumUIType</i>	
<i>QPath</i>	Query Path 类，使用 QPath 字符串定位 UI 控件

### Functions

<i>_find_by_name</i> (root, name)	
-----------------------------------	--

```
class qt4c.qpath.EnumQPathKey
```

Bases: object

MAX\_DEPTH = MAXDEPTH

INSTANCE = INSTANCE

UI\_TYPE = UITYPE

```
class qt4c.qpath.EnumUIType
```

Bases: object

WIN = Win

UIA = UIA

```
exception qt4c.qpath.QPathError
```

Bases: Exception

QPath 异常类定义

```
class qt4c.qpath.QPath(qpath_string)
```

Bases: object

Query Path 类，使用 QPath 字符串定位 UI 控件

QPath 的定义: Qpath ::= Seperator UIObjectLocator Qpath Seperator ::= 路径分隔符，任意的单个字符  
 UIObjectLocator ::= UIObjectProperty && UIObjectLocator UIObjectProperty ::= UIProperty  
 | RelationProperty | IndexProperty | UITypeProperty UIProperty ::= [Window/UIA/Html]Property

Operator “Value” UITypeProperty ::= Win | UIA | Html RelationProperty ::= MaxDepth = “Number” (最大搜索子孙深度) IndexProperty ::= Instance=” Integer” (Integer: 找到的多个控件中的第几个 (负数表示从后往前数) )

Operator ::= ‘=’ | ‘~=’ ( ‘=’ 表示精确匹配; ‘~=’ 表示用正则表达式匹配)

UI 控件基本上都是由树形结构组织起来的。为了方便定位树形结构的节点，QPath 采用了路径结构的字符串形式。QPath 以第一个字符为路径分隔符，如“/Node1/Node2/Node3”和“|Node1|Node2|Node3”是一样的路径，都表示先找到 Node1，再在 Node1 的子孙节点里找 Node2，然后在 Node2 的子孙节点里找 Node3。而定位每个 Node 需要改节点的多个属性以” &&” 符号连接起来，形成 “/Property1=’ value1’ && property2~=’ value2’ && …” 的形式，其中” ~=” 表示正则匹配。QPath 支持的属性包括 wincontrols.Window 和 htmlcontrols.HtmlElement 的类属性。QPath 中的每个节点都有两个默认属性’ UIType’ 和’ MaxDepth’。“UIType=’ Win32|UIA|Html’”，三个取值分别对应了三种 QPath 支持的 UI 类型。当 UIType 没有指定时默认取值为父节点的值。而” MaxDepth” 表示该节点离祖先节点的最大深度，如果没有明确指定时默认取值为’ 1’，即直接父子关系。QPath 还支持” Instance” 属性，用于当找到多个节点时指定选择第几个节点。

例子：Qpath =” / ClassName=’ TxGuiFoundation’ && Caption~=’ QQd+’ && Instance=’ -1’  
/ UIType=’ UIA’ && name=’ mainpanel’ && MaxDepth=’ 10”

PROPERTY\_SEP = &&

OPERATORS = [‘=’, ‘~=’]

MATCH\_FUNCS

CONTROL\_TYPES

update\_control\_type(self)

\_find\_controls\_recur(self, root, qpath)

递归查找控件

#### 参数

- **root** – 根控件
- **qpath** – 解析后的 qpath 结构

**返回** 返回 (found\_controls, remain\_qpath)，其中 found\_controls 是找到的控件，remain\_qpath

是未能找到控件时剩下的未能匹配的 qpath。

\_match\_control(self, control, props)

控件是否匹配给定的属性

#### 参数

- **control** – 控件
- **props** – 要匹配的控件属性字典，如 { ‘classname’ : [ ‘=’, ‘window’ ] }

`_parse_property(self, prop_str)`

解析 property 字符串，返回解析后结构

例如将 “ClassName=’ Dialog’ ” 解析返回 {ClassName: [ ‘=’ , ‘Dialog’ ]}

`_parse(self, qpath_string)`

解析 qpath，并返回 QPath 的路径分隔符和解析后的结构

将例如” | ClassName=’ Dialog’ && Caption~=’ SaveAs’ | UIType=’ UIA’ && ControlID=’ 123’ && Instanc=’ -1’ ” 的 QPath 解析为下面结构: [{ ‘ClassName’ : [ ‘=’ , ‘Dialog’ ], ‘Caption’ : [ ‘~=’ , ‘SaveAs’ ]}, { ‘UIType’ : [ ‘=’ , ‘UIA’ ], ‘ControlID’ : [ ‘=’ , ‘123’ ], ‘Instance’ : [ ‘=’ , ‘-1’ ]}]

**参数** `qpath_string` – qpath 字符串

**返回** (separator, parsed\_qpath)

`__str__(self)`

返回格式化后的 QPath 字符串

`getErrorPath(self)`

返回最后一次 QPath.search 搜索未能匹配的路径

**返回类型** string

`search(self, root=None)`

根据 qpath 和 root 查找控件

**参数** `root` (实例类型) – 查找开始的控件

**返回** 返回找到的控件列表

`qt4c.qpath._find_by_name(root, name)`

`qt4c.testcase`

TUIA 测试用例基类

## Module Contents

### Classes

---

*ClientTestCase*

QT4C 测试用例基类

---

### Functions

<code>_screenshot(path)</code>	save screenshot
--------------------------------	-----------------

```
qt4c.testcase._screenshot(path)
    save screenshot

class qt4c.testcase.ClientTestCase
    Bases: testbase.testcase.TestCase

    QT4C 测试用例基类

    init_test

    clean_test

    initTest(self, testresult)
        测试用例初始化。慎用此函数，尽量将初始化放到 preTest 里。

    cleanTest(self)
        测试用例反初始化

    get_extra_fail_record(self)
```

qt4c.uiacontrols

使用 UIA 方式去访问控件

Module Contents

Classes

<i>Control</i>	UIA 方式访问控件基类
<i>UIAWindows</i>	UIA 控件窗体定义
<i>Edit</i>	UIA 方式访问控件基类
<i>ComboBox</i>	UIA 方式访问控件基类
<i>RadioButton</i>	UIA 方式访问控件基类

Functions

<code>find_UIAElm(Condition, timeout=10)</code>
---

```
qt4c.uiacontrols.IUIAutomation
qt4c.uiacontrols.UIAutomationClient
```



```

qt4c.uiaccontrols.RawWalker

qt4c.uiaccontrols.ControlWalker

qt4c.uiaccontrols.UIAControlType

qt4c.uiaccontrols.find_UIAElm(Condition, timeout=10)

class qt4c.uiaccontrols.Control(root=None, locator=None)
    Bases: qt4c.control.Control

    UIA 方式访问控件基类

    Valid
        是否是有效控件

    Width
        控件宽度

    Height
        控件高度

    BoundingRect
        返回控件

    ProcessId

    ControlType
        返回 UIA 控件的类型

    Enabled
        此控件是否可用

    Children
        返回子控件列表:rtype ListType

    Parent

    Name
        返回 control 的 name 属性

    Type
        返回控件类型

    Value
        返回控件 value 属性（通常是文本信息）

    Hwnd
        返回控件句柄

    HWnd
        兼容其他类型控件

    hwnd

```

**HasKeyboardFocus**

返回是否被选为键盘输入

**ClassName**

返回 ClassName

**\_init\_uiaobj(self)**

初始化 uia 对象

**empty\_invoke(self)**

无意义调用，用于主动初始化对象

**SetFocus(self)**

**\_getrect(self)**

**click(self, mouseFlag=MouseFlag.LeftButton, clickType=MouseClickType.SingleClick, xOffset=**  
**set=None, yOffset=None)**

点击控件:type mouseFlag:qt4c.mouse.MouseFlag :param mouseFlag: 鼠标按钮类型:type click-  
Type:qt4c.mouse.MouseClickType :param clickType: 点击动作类型:type xOffset:int :param 横向  
偏移量:type yOffset:int :param 纵向偏移量

**equal(self, other)**

判断两个对象是否相同。未实现!

**参数 other (Control)** – 本对象实例

**exist(self)**

判断控件是否存在

**wait\_for\_exist(self, timeout, interval)**

等待控件存在

**class qt4c.uiacontrols.UIAWindows(root=None, locator=None)**

Bases: *qt4c.uiacontrols.Control, qt4c.control.ControlContainer*

UIA 控件窗体定义

**Window**

返回 wincontrols.Window

**Visible**

窗口是否可见

**Minimized**

该窗口是否最小化

**class qt4c.uiacontrols.Edit(root=None, locator=None)**

Bases: *qt4c.uiacontrols.Control*

UIA 方式访问控件基类

`input(self, data)`  
对 Edit 类型控件进行输入:type keys: utf-8 str or unicode :param keys: 键盘输入字符串, 可输入组合键, 如” {CTRL}{MENU}a”

`class qt4c.uiacontrols.ComboBox(root=None, locator=None)`  
Bases: `qt4c.uiacontrols.Control`

UIA 方式访问控件基类

**Value**  
返回控件 value 属性 (通常是文本信息)

`input(self, data)`  
对 Edit 类型控件进行输入:type keys: utf-8 str or unicode :param keys: 键盘输入字符串, 可输入组合键, 如” {CTRL}{MENU}a”

`expand(self)`

`collapse(self)`

`class qt4c.uiacontrols.RadioButton(root=None, locator=None)`  
Bases: `qt4c.uiacontrols.Control`

UIA 方式访问控件基类

`is_select(self)`

`qt4c.util`

其他共用类模块

Module Contents

Classes

<i>Point</i>	坐标点类
<i>Rectangle</i>	矩形区域类
<i>ProcessMem</i>	跨进程数据读写
<i>MsgSyncer</i>	Thread Post Message 同步
<i>Process</i>	提供系统进程的操作类
<i>MetisView</i>	各端实现的 MetisView

Functions

<i>getEncoding</i> (s)	获取字符串的编码
<i>myEncode</i> (s, ec=' unicode' , sc=None)	将字符串转换为指定的编码的字符串
<i>getToolTips</i> (className=' tooltips_class32' , retry=10)	获取系统的浮动 tips
<i>remote_inject_dll</i> (process_id, dll_path)	在 process_id 进程中远程注入 dll_path 的 DLL
<i>is_system_locked</i> ()	检测系统是否处于锁屏界面
<i>getDpi</i> (hwnd=None)	

qt4c.util.unicode

qt4c.util.\_DEFAULT\_BUFFER\_SIZE = 255

qt4c.util.SIZEOF

class qt4c.util.Point(x\_y)

Bases: object

坐标点类

X

Y

All

\_\_eq\_\_(self, pt)

Return self==value.

class qt4c.util.Rectangle(left\_top\_right\_bottom)

Bases: object

矩形区域类

All

Bottom

Center

Left

Right

Top

Width

Height

\_\_str\_\_(self)

Return str(self).

**isInRect**(*self*, *rc*)

判断此区域是否在参数 *rc* 的范围内

**参数** *rc* (*Rectangle*) – 包含的区域

**highlight**(*self*)

高亮此区域

**\_\_eq\_\_**(*self*, *rc*)

Return self==value.

**\_\_ne\_\_**(*self*, *rc*)

Return self!=value.

**class** qt4c.util.ProcessMem(*processId*, *buffer\_size*=None, *remote\_buffer*=None)

Bases: object

跨进程数据读写

**Buffer**

返回远程进程中内存 *buffer* 地址

**\_\_del\_\_**(*self*)

**write**(*self*, *local\_buffer*, *buffer\_size*)

将 *local\_buffer* 中的数据写入远程内存中。

**参数**

- **local\_buffer** (*buffer*) – A pointer to the buffer that contains data to be written in the address space of the specified process
- **buffer\_size** (*unsigned long*) – The number of bytes to be written to the specified process

**read**(*self*, *local\_buffer*, *buffer\_size*)

将远程数据读到本地 *buffer*

**参数**

- **local\_buffer** (*buffer*) – A pointer to a buffer that receives the contents from the address space of the specified process
- **buffer\_size** (*unsigned long*) – The number of bytes to be read from the specified process

qt4c.util.getEncoding(*s*)

获取字符串的编码

**返回类型** string

**返回** ‘GBK’ , ‘UNICODE’ , ‘UTF-8’ , ‘UNKNOWN’

`qt4c.util.myEncode(s, ec='unicode', sc=None)`

将字符串转换为指定的编码的字符串

#### 参数

- `s (string)` – 待转换的字符串
- `ec (string)` – 待转换的编码. [ 'GBK' , ' UNICODE' , ' UTF-8' ]
- `sc (string)` – 待转换的字符串的编码

**Attention** `sc` 默认值为 `None`, 此时函数会自动判断 `s` 的编码 (有一定概率会判断错误)

**返回** 转换后的字符串

`qt4c.util.getToolTips(className='tooltips_class32', retry=10)`

获取系统的浮动 tips

#### 参数

- `className` (字符串) – 类名, 默认值为” `tooltips_class32`”
- `retry` (整数) – 尝试次数, 每个 0.5 秒尝试一次

`class qt4c.util.MsgSyncer(hwnd)`

Bases: object

Thread Post Message 同步

`pid_event_map`

`wait(self, timeout=60)`

等待消息同步

`qt4c.util.remote_inject_dll(process_id, dll_path)`

在 `process_id` 进程中远程注入 `dll_path` 的 DLL

`class qt4c.util.Process(pid)`

Bases: object

提供系统进程的操作类使用例子: `for proc in Process.GetProcessesByName( 'iexplore.exe' ):`

`print proc.ProcessId`

**ProcessName**

返回进程名字。失败返回 `None`

**Live**

**ProcessId**

**ProcessPath**

获取进程可执行文件的全路径

**static GetProcessesByName(process\_name)**

返回进程名为 `process_name` 的 `Process` 类实例列表

`waitForQuit(self, timeout=10, interval=0.5)`

在指定的时间内等待退出

**返回** 如果在指定的时间内退出，返回 True；否则返回 False

`_adjustProcessPrivileges(self)`

提升权限

`terminate(self)`

终止进程

`qt4c.util.is_system_locked()`

检测系统是否处于锁屏界面

@return: bool 如果处于锁屏，返回 True

`class qt4c.util.MetisView(control)`

Bases: object

各端实现的 MetisView

**rect**

元素相对坐标 (x, y, w, h)

**os\_type**

系统类型，例如" android", " ios", " pc"

`screenshot(self)`

当前容器的区域截图

`click(self, offset_x=None, offset_y=None)`

点击

### 参数

- **offset\_x** (*float/None*) – 相对于该控件的坐标 offset\_x, 百分比 ( 0 -> 1 ), 不传入则默认该控件的中央
- **offset\_y** (*float/None*) – 相对于该控件的坐标 offset\_y, 百分比 ( 0 -> 1 ), 不传入则默认该控件的中央

`send_keys(self, text)`

`double_click(self, offset_x=None, offset_y=None)`

`long_click(self, offset_x=None, offset_y=None)`

`qt4c.util.getDpi(hwnd=None)`

`qt4c.version`

Module Contents

qt4c.version.version = 2.2.0

qt4c.webcontrols

Web 自动化公共接口

Module Contents

Classes

<i>WebPage</i>	封装 PC 端自动化所需的页面相关的逻辑
----------------	----------------------

```
class qt4c.webcontrols.WebPage(webview)
    Bases: qt4w.webcontrols.WebPage
    封装 PC 端自动化所需的页面相关的逻辑
    close(self)
    activate(self)
```

qt4c.wincontrols

Window 的控件模块

Module Contents

Classes

<i>_CWindow</i>	Win32 Window 类 (bridge)
<i>Control</i>	Win32 Window 类, 实现 Win32 窗口常用属性。
<i>ListViewItem</i>	sysListView32 的每一项
<i>ListView</i>	sysLisView32 控件类型
<i>TextBox</i>	编辑控件
<i>Window</i>	Win32 Window 类, 实现 Win32 窗口常用属性。
<i>TrayNotifyBar</i>	系统的通知区域
<i>TrayTaskBar</i>	系统的任务栏区域 (win7 以上不可用)

下页继续



表 23 – 续上页

<i>_TrayIcon</i>	通知栏或任务栏的项
<i>ComboBox</i>	ComboBox 控件类型
<i>TreeView</i>	Sys TreeView 32 控件类型
<i>TreeViewItem</i>	TreeView Item 类，不要直接实例化这个类，而通过 TreeView.Items 来获取。
<i>_ITEMLIST</i>	Built-in mutable sequence.
<i>MenuItem</i>	菜单项控件。不要直接实例化这个类，而通过 Menu.MenuItems 来获取。
<i>Menu</i>	菜单控件

```
class qt4c.wincontrols._CWindow(hwnd)
    Bases: object
    Win32 Window 类 (bridge)

    HWnd
        窗口句柄

class qt4c.wincontrols.Control(root=None, locator=None)
    Bases: qt4c.control.Control
    Win32 Window 类，实现 Win32 窗口常用属性。

    BoundingRect
        返回窗口大小
            返回类型 util.Rectangle
            返回 util.Rectangle 实例

    Caption
        返回窗口标题
            返回类型 StringType
            返回 窗口标题

    Children
        返回子控件列表
            返回类型 ListType

    ClassName
        返回窗口类名

    ControlId
        返回控件 ID

    Enabled
        此控件是否可用
```

### ExStyle

此控件的扩展样式

### HWnd

### hwnd

### Parent

返回父窗口

**返回类型** *Window*

**返回** 获取父窗口

**Attention** 如果是 desktop 窗口, 则返回 None; 如果是顶级窗口, 则返回 desktop 窗口; 否则返回父窗口。

### ProcessId

### Style

此控件的样式

### Text

### ThreadId

窗口线程 ID

### TopLevelWindow

返回控件的最上层窗口

**返回类型** *Window*

### Valid

窗口是否存在

### Visible

此控件是否可见

### AccessibleObject

返回 AccessibleObject

**返回类型** *qt4c.accessible.AccessibleObject*

### Width

宽度

### Height

高度

### \_init\_wndobj(*self*)

初始化 Win32 窗口对象

### static \_\_enum\_childwin\_callback(*hwnd*, *hwnds*)

`__validCtrlNum(self, ctrls)`

`equal(self, other)`

判断两个对象是否相同。

**参数** `other` (`Control`) – 本对象实例

`exist(self)`

判断控件是否存在

`wait_for_exist(self, timeout, interval)`

等待控件存在

`waitForExist(self, timeout, interval)`

等待控件存在

`wait_for_invalid(self, timeout=10.0, interval=0.5)`

等待控件失效

`waitForInvalid(self, timeout=10.0, interval=0.5)`

等待控件失效

`click(self, mouseFlag=MouseFlag.LeftButton, clickType=MouseClickType.SingleClick, xOffset=None, yOffset=None)`

点击控件

**参数**

- `mouseFlag` (`qt4c.mouse.MouseFlag`) – 鼠标按钮
- `clickType` (`qt4c.mouse.MouseClickType`) – 鼠标动作
- `xOffset` (`int`) – 距离控件区域左上角的偏移。默认值为 `None`, 代表控件区域 x 轴上的中点; 如果为负值, 代表距离控件区域右上角的 x 轴上的绝对值偏移;
- `yOffset` (`int`) –  
距离控件区域左上角的偏移。默认值为 `None`, 代表控件区域 y 轴上的中点;  
如果为负值, 代表距离控件区域右上角的 y 轴上的绝对值偏移;

`setFocus(self)`

将此控件设为焦点

`class qt4c.wincontrols.ListViewItem(parent, item_index)`

Bases: `qt4c.control.Control`

sysListView32 的每一项

**SubItems**

**BoundingRect**

获取 ListView 的某项 Item 的文本

### Text

获取 ListView 的某项 Item 的文本

```
class qt4c.wincontrols.ListView(root=None, locator=None)
```

Bases: [qt4c.wincontrols.Control](#)

sysListView32 控件类型

### ItemCount

The number of items in the ListView

### Items

```
__iter__(self)
```

```
__getitem__(self, key)
```

```
class qt4c.wincontrols.TextBox(root=None, locator=None)
```

Bases: [qt4c.wincontrols.Control](#)

编辑控件

```
class qt4c.wincontrols.Window(root=None, locator=None)
```

Bases: [qt4c.wincontrols.Control](#), [qt4c.control.ControlContainer](#)

Win32 Window 类，实现 Win32 窗口常用属性。

### Maximized

该窗口是否最大化

### Minimized

该窗口是否最小化

### OwnerWindow

此窗口的所有者窗口

返回类型 [Window](#)

返回 获取 Owner Window

### PopupWindow

此窗口的弹出窗口

返回类型 [Window](#)

返回 获取 EnabledPopup

### TopMost

是否具有总在最前端

```
bringForeground(self)
```

将窗口设为最前端窗口

```
_wait_for_disabled_or_invisible(self, timeout=60, interval=0.5)
```

`close(self)`  
 关闭窗口  
 返回类型 `bool`  
 返回 窗口销毁返回 `True`, 否则返回 `False`

`hide(self)`  
 隐藏

`maximize(self)`  
 最大化窗口

`minimize(self)`  
 最小化窗口

`resize(self, width, height)`  
 设置窗口大小

`restore(self)`  
 恢复窗口

`show(self)`  
 显示窗口

`move(self, x, y)`  
 移动窗口

`waitForInvalid(self, timeout=10.0, interval=0.5)`  
 等待窗口失效

参数 `timeout (float)` – 超时秒数

`waitForInvisible(self, timeout=10.0, interval=0.5)`  
 等待窗口消失

参数 `timeout (float)` – 超时秒数

`class qt4c.wincontrols.TrayNotifyBar`

Bases: `qt4c.wincontrols.Control`

系统的通知区域

**Items**

返回 `TrayNotifyBar` 的全部 `TrayNotifyIcon`

`refresh(self)`  
 刷新通知区域

`__getitem__(self, key)`

`class qt4c.wincontrols.TrayTaskBar`

Bases: `qt4c.wincontrols.Control`

系统的任务栏区域 (win7 以上不可用)

#### Items

返回 TrayTaskBar 的全部 TrayNotifyIcon

`__getitem__(self, key)`

`class qt4c.wincontrols._TrayIcon(tbButton, trayData, notifyBar)`

Bases: `qt4c.control.Control`

通知栏或任务栏的项

#### BoundingRect

托盘图标的位置

返回类型 `util.Rectangle`

返回 `util.Rectangle`

#### ProcessId

图标所代表的进程 Id

#### State

图标状态, 隐藏 or 显示

#### Style

图标风格, 分隔符 or 自定义图片之类

#### Tips

图标提示

#### Visible

是否可见

`click(self, mouseFlag=MouseFlag.LeftButton, clickType=MouseClickedType.SingleClick, xOffset=None, yOffset=None)`  
 点击控件 (同步操作)

#### 参数

- `mouseFlag` (`qt4c.mouse.MouseFlag`) – 鼠标按钮
- `clickType` (`qt4c.mouse.MouseClickType`) – 鼠标动作
- `xOffset` (`int`) – 距离控件区域左上角的偏移。默认值为 `None`, 代表控件区域 x 轴上的中点; 如果为负值, 代表距离控件区域右上角的 x 轴上的绝对值偏移;
- `yOffset` (`int`) –  
 距离控件区域左上角的偏移。默认值为 `None`, 代表控件区域 y 轴上的中点;  
 如果为负值, 代表距离控件区域右上角的 y 轴上的绝对值偏移;

`destroy(self)`

销毁该图标使之不再显示

`equal(self, other)`

判断两个对象是否相同。未实现!

**参数** `other` (`Control`) – 本对象实例

`class qt4c.wincontrols.ComboBox(root=None, locator=None)`

Bases: `qt4c.wincontrols.Control`

ComboBox 控件类型

**Count**

返回 ComboBox 的项目数

**SelectedIndex**

返回当选选中的索引值

`_getTextByIndex(self, idx=-1)`

`getFullPath(self)`

`class qt4c.wincontrols.TreeView(root=None, locator=None)`

Bases: `qt4c.wincontrols.Control`

Sys TreeView 32 控件类型

**Items**

返回 TreeView 的首层节点列表

**返回类型** list

`count(self)`

返回 TreeView 的 item 数

`class qt4c.wincontrols.TreeViewItem(hwnd, item)`

Bases: `qt4c.control.Control`

TreeView Item 类，不要直接实例化这个类，而通过 `TreeView.Items` 来获取。

**Hwnd**

窗口句柄

**hwnd**

窗口句柄

**BoundingRect**

返回窗口大小。未实现!

**Items**

**Selected**

**Text**

**NextSibling**

**ensureVisible(self)**

先保证 item 完全可见

**select(self)**

选中自身结点

**class qt4c.wincontrols.\_ITEMLIST**

Bases: list

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

**\_\_getitem\_\_(self, key)**

x.\_\_getitem\_\_(y) <==> x[y]

**class qt4c.wincontrols.MenuItem(menu, index)**

Bases: *qt4c.wincontrols.Control*

菜单项控件。不要直接实例化这个类，而通过 Menu.MenuItems 来获取。

**class EnumMenuItemState**

Bases: object

**DISABLED = Disabled**

**GRAYED = Grayed**

**NORMAL = Normal**

**UNKNOWN = Unknown**

**IsSeperator**

返回该子项是否是分割线

**State**

返回菜单项的状态

**返回类型** *EnumMenuItemState*

**Text**

可见文本

**SubMenu**

鼠标移动到该子项上，产生子菜单，并返回该子菜单

**返回类型** *Menu*

**BoundingRect**

返回 rect

**static \_\_enum\_childwin\_callback(hwnd, hwnds)**



`__getSysMenuWindow(self)`

`click(self, mouseFlag=MouseFlag.LeftButton, clickType=MouseClickType.SingleClick, xOffset=None, yOffset=None)`  
 点击控件

#### 参数

- `mouseFlag` (`qt4c.mouse.MouseFlag`) – 鼠标按钮
- `clickType` (`qt4c.mouse.MouseClickType`) – 鼠标动作
- `xOffset` (`int`) – 距离控件区域左上角的偏移。默认值为 `None`, 代表控件区域 `x` 轴上的中点; 如果为负值, 代表距离控件区域右上角的 `x` 轴上的绝对值偏移;
- `yOffset` (`int`) –  
 距离控件区域左上角的偏移。默认值为 `None`, 代表控件区域 `y` 轴上的中点;  
 如果为负值, 代表距离控件区域右上角的 `y` 轴上的绝对值偏移;

`class qt4c.wincontrols.Menu(root=None, locator=None)`

Bases: `qt4c.wincontrols.Window`

菜单控件

#### MenuItems

获取 MenuItem。通过 `MenuItems[菜单项索引]` 或 `MenuItems[菜单项文字]` 返回 MenuItem 实例。

`__getSubMenuItemsCount(self)`

`__findSysMenuWindow(self)`

`static closeAllSysMenuWindow()`

关闭所有的系统 menu 窗口

`static __enum_childwin_callback(hwnd, hwnds)`

`__iter__(self)`

`__getitem__(self, key)`

根据 key 返回 MenuItem key: 菜单索引或菜单文字

`qt4c.wintypes`

## Module Contents

### Classes

<i>RECT</i>	The RECT structure defines the coordinates of the upper-left and lower-right corners of a rectangle
<i>TBBUTTON</i>	Contains information about a button in a system's traynotifybar.
<i>TRAYDATA</i>	for description
<i>PROCESSENTRY32</i>	<b>desc</b> Describes an entry from a list that enumerates the processes residing in the system address space when a snapshot was taken.
<i>APPBARDATA</i>	App Bar Data Structure
<i>LVITEM</i>	<b>desc</b> Specifies or receives the attributes of a list-view item. This structure has been updated
<i>LVITEM64</i>	<b>desc</b> Specifies or receives the attributes of a list-view item. This structure has been updated
<i>TVITEM</i>	
<i>BITMAPINFOHEADER</i>	BITMAP Info Header
<i>RGBTRIPLE</i>	RGB Define
<i>BITMAPINFO</i>	BITMAP Info
<i>BITMAP</i>	BITMAP
<i>DIBSECTION</i>	DIB Section
<i>BITMAPFILEHEADER</i>	BITMAP File Header
<i>MODULEENTRY32</i>	This structure describes an entry from a list that enumerates
<i>THREADENTRY32</i>	This structure describes an entry from a list that enumerates the threads executing in the system when a snapshot was taken.

qt4c.wintypes.ULONG\_PTR

class qt4c.wintypes.RECT

Bases: ctypes.Structure

The RECT structure defines the coordinates of the upper-left and lower-right corners of a rectangle

```

    _fields_ = [None, None, None, None]

class qt4c.wintypes.TBBUTTON
    Bases: ctypes.Structure

    Contains information about a button in a system' s traynotifybar.

    _fields_ = [None, None, None, None, None, None, None]

class qt4c.wintypes.TRAYDATA
    Bases: ctypes.Structure

    for description

    _fields_ = [None, None, None, None, None]

class qt4c.wintypes.PROCESSENTRY32
    Bases: ctypes.Structure

    Desc Describes an entry from a list that enumerates the processes residing in the system
        address space when a snapshot was taken.

    _fields_ = [None, None, None, None, None, None, None, None, None, None, None, None]

class qt4c.wintypes.APPBARDATA
    Bases: ctypes.Structure

    App Bar Data Structure

    _fields_ = [None, None, None, None, None, None]

class qt4c.wintypes.LVITEM
    Bases: ctypes.Structure

    Desc Specifies or receives the attributes of a list-view item. This structure has been updated
        to support a new mask value (LVIF_INDENT) that enables item indenting. This structure supersedes
        the LV_ITEM structure

    _fields_ = [None, None, None, None, None, None, None, None, None, None, None, None, None]

class qt4c.wintypes.LVITEM64
    Bases: ctypes.Structure

    Desc Specifies or receives the attributes of a list-view item. This structure has been updated
        to support a new mask value (LVIF_INDENT) that enables item indenting. This structure supersedes
        the LV_ITEM structure

    _fields_ = [None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None]

class qt4c.wintypes.TVITEM
    Bases: ctypes.Structure

    _fields_ = [None, None, None, None, None, None, None, None, None, None]

```

```
class qt4c.wintypes.BITMAPINFOHEADER
    Bases: ctypes.Structure

    BITMAP Info Header

    _fields_ = [None, None, None, None, None, None, None, None, None, None]

class qt4c.wintypes.RGBTRIPLE
    Bases: ctypes.Structure

    RGB Define

    _fields_ = [None, None, None, None]

class qt4c.wintypes.BITMAPINFO
    Bases: ctypes.Structure

    BITMAP Info

    _fields_ = [None, None]

class qt4c.wintypes.BITMAP
    Bases: ctypes.Structure

    BITMAP

    _fields_ = [None, None, None, None, None, None, None]

class qt4c.wintypes.DIBSECTION
    Bases: ctypes.Structure

    DIB Section

    _fields_ = [None, None, None, None, None]

class qt4c.wintypes.BITMAPFILEHEADER
    Bases: ctypes.Structure

    BITMAP File Header

    _fields_ = [None, None, None, None, None]

class qt4c.wintypes.MODULEENTRY32
    Bases: ctypes.Structure

    This structure describes an entry from a list that enumerates the modules used by a specified process.

    _fields_ = [None, None, None, None, None, None, None, None, None, None]

class qt4c.wintypes.THREADENTRY32
    Bases: ctypes.Structure

    This structure describes an entry from a list that enumerates the threads executing in the system when
    a snapshot was taken.
```

```
_fields_ = [None, None, None, None, None, None, None]
```

1.7.2 browser

浏览器类库

Submodules

browser.chrome

ChromeBrowser 的接口实现

Module Contents

Classes

<i>ChromeBrowser</i>	Chrome 浏览器
----------------------	------------

Functions

<i>is_port_occupied</i> (port)	端口是否被占用
<i>get_next_avail_port</i> (port)	

```
class browser.chrome.ChromeBrowser(port=9200)
    Bases: qt4w.browser.browser.IBrowser
    Chrome 浏览器

    temp_path
    Url

    _handle_title(self, title)
        处理标题

    get_chrome_window_list(self, pid)
        通过 pid 查找对应的 chrome 窗口列表

    open_url(self, url, page_cls=None)

    find_by_url(self, url, page_cls=None, timeout=10)
        在当前打开的页面中查找指定 url, 返回 WebPage 实例, 如果未找到, 则抛出异常

    get_page_cls(self, webview, page_cls=None)
```

得到具体页面类

`static get_browser_path()`

获取 chrome.exe 的路径

`search_chrome_webview(self, url)`

根据 url 查找 chrome 对应的 webview 类

returns ChromeWebView: ChromeWebView 类

`clear_cache(self)`

`close(self)`

`static killall()`

杀掉所有 chrome 进程

`browser.chrome.is_port_occupied(port)`

端口是否被占用

`browser.chrome.get_next_avail_port(port)`

`browser.firefox`

firefox 浏览器模块

## Module Contents

### Classes

---

*FireFoxApp*

FireFox 浏览器 App

---

`class browser.firefox.FireFoxApp(locator)`

Bases: *qt4c.app.App*

FireFox 浏览器 App

`browser.ie`

IE 模块

## Module Contents

Classes

<a href="#"><i>IEWindow</i></a>	IE 窗口 qt4w 使用
<a href="#"><i>IEBrowser</i></a>	IE 浏览器
<a href="#"><i>IEApp</i></a>	to be deleted

```
class browser.ie.IEWindow(process_id)
    Bases: qt4c.wincontrols.Window

    IE 窗口 qt4w 使用

    _timeout

    pid

    ie_window
        获取 Internet Explorer_Server 对应的 ie 窗口

    webview
        返回 WebView

        返回类型 IEWebView

        返回 IEWebView, 用于实例化对应的 WebPage

    Url
        返回当前的 URL 地址
```

```
class browser.ie.IEBrowser
    Bases: qt4w.browser.IBrowser

    IE 浏览器

    static get_version()
        获取注册表中的 IE 版本

    static get_path()
        获取注册表中 IE 安装位置

    static searh_ie_window(url)
        查找 ie 进程查找到就退出, 现在无法解决 url 对应的标签不在 IE 最前面的问题

    static killall()
        kill 掉所有 IE 进程

    close(self)
        kill 掉所有 IE 进程

    open_url(self, url, page_cls=None)
        打开一个 url, 返回对应的 webpage 实例类
```

Params url url

Params page\_cls page 实例类

`find_by_url(self, url, page_cls=None, timeout=10)`

通过 url 查找页面，支持正则匹配

`_get_page_cls(self, process_id_or_window, page_cls=None)`

获取具体的 webpage 实例类

`class browser.ie.IEApp`

Bases: object

to be deleted

`static killAll()`

## Package Contents

### Classes

---

*EnumBrowserType*

---

### Functions

---

*\_get\_default\_browser1()*

获取 xp 下的默认浏览器

---

*\_get\_default\_browser2()*

获取 vista 或以上的默认浏览器

---

*get\_default\_browser()*

获取默认浏览器类型

---

`class browser.EnumBrowserType`

Bases: object

`IE = iexplorer`

`TT = ttraverler`

`FireFox = firefox`

`Chrome = chrome`

`QQBrowser = qqbrowser`

`browser.browser_list = []`

`browser._get_default_browser1()`

获取 xp 下的默认浏览器

`browser._get_default_browser2()`



获取 vista 或以上的默认浏览器

`browser.get_default_browser()`

获取默认浏览器类型

返回类型 *EnumBrowserType*



## CHAPTER 2

---

### 索引和搜索

---

- `genindex`
- `modindex`
- `search`



## b

`browser`, 89  
`browser.chrome`, 89  
`browser.firefox`, 90  
`browser.ie`, 90

## q

`qt4c`, 35  
`qt4c.accessible`, 47  
`qt4c.app`, 53  
`qt4c.control`, 54  
`qt4c.exceptions`, 57  
`qt4c.filedialog`, 57  
`qt4c.keyboard`, 58  
`qt4c.mouse`, 61  
`qt4c.qpath`, 65  
`qt4c.testcase`, 67  
`qt4c.uiacontrols`, 68  
`qt4c.util`, 71  
`qt4c.version`, 75  
`qt4c.webcontrols`, 76  
`qt4c.webview`, 35  
`qt4c.webview.base`, 45  
`qt4c.webview.chromewebview`, 35  
`qt4c.webview.chromewebview.chromedriver`, 35  
`qt4c.webview.iewebview`, 39  
`qt4c.webview.iewebview.iedriver`, 39  
`qt4c.wincontrols`, 76  
`qt4c.wintypes`, 85



## 符号

- `__AccessibleObjectWrapper_comtypes` (qt4c.accessible 中的类), 51
- `__AccessibleObjectWrapper_win32com` (qt4c.accessible 中的类), 51
- `__CODES()` (在 qt4c.keyboard 模块中), 59
- `__CWindow` (qt4c.wincontrols 中的类), 77
- `__DEFAULT_BUFFER_SIZE()` (在 qt4c.util 模块中), 72
- `__ITEMLIST` (qt4c.wincontrols 中的类), 84
- `__KeyboardEvent` (qt4c.keyboard 中的类), 59
- `__MODIFIERS()` (在 qt4c.keyboard 模块中), 59
- `__MODIFIER_KEY_MAP()` (在 qt4c.keyboard 模块中), 59
- `__SHIFT()` (在 qt4c.keyboard 模块中), 59
- `__TrayIcon` (qt4c.wincontrols 中的类), 82
- `__del__()` (qt4c.util.ProcessMem 方法), 73
- `__del__()` (qt4c.webview.chromewebview.chromedriver.WebKitDebugger 方法), 36
- `__enum_childwin_callback()` (qt4c.wincontrols.Control 静态方法), 78
- `__enum_childwin_callback()` (qt4c.wincontrols.Menu 静态方法), 85
- `__enum_childwin_callback()` (qt4c.wincontrols.MenuItem 静态方法), 84
- `__eq__()` (qt4c.control.Control 方法), 55
- `__eq__()` (qt4c.util.Point 方法), 72
- `__eq__()` (qt4c.util.Rectangle 方法), 73
- `__findSysMenuWindow()` (qt4c.wincontrols.Menu 方法), 85
- `__findctrl_recur()` (qt4c.control.ControlContainer 方法), 56
- `__getSysMenuWindow()` (qt4c.wincontrols.MenuItem 方法), 84
- `__getattr__()` (qt4c.webview.base.WebViewBase 方法), 46
- `__getattr__()` (qt4c.webview.chromewebview.WebViewBase 方法), 38
- `__getattr__()` (qt4c.webview.iewebview.WebViewBase 方法), 41
- `__getitem__()` (qt4c.control.ControlContainer 方法), 56
- `__getitem__()` (qt4c.wincontrols.ListView 方法), 80
- `__getitem__()` (qt4c.wincontrols.Menu 方法), 85
- `__getitem__()` (qt4c.wincontrols.TrayNotifyBar 方法), 81
- `__getitem__()` (qt4c.wincontrols.TrayTaskBar 方法), 82
- `__getitem__()` (qt4c.wincontrols.\_ITEMLIST 方法), 84
- `__iter__()` (qt4c.wincontrols.ListView 方法), 80
- `__iter__()` (qt4c.wincontrols.Menu 方法), 85
- `__ne__()` (qt4c.control.Control 方法), 56
- `__ne__()` (qt4c.util.Rectangle 方法), 73
- `__str__()` (qt4c.qpath.QPath 方法), 67
- `__str__()` (qt4c.util.Rectangle 方法), 72
- `__str__()` (qt4c.webview.chromewebview.chromedriver.ChromeDriver 方法), 36

- 方法), 36
- \_\_validCtrlNum() (qt4c.wincontrols.Control 方法), 78
- \_\_accessible\_object\_from\_point() (qt4c.accessible.\_\_AccessibleObjectWrapper\_comtypes方法), 52
- \_\_accessible\_object\_from\_window() (qt4c.accessible.\_\_AccessibleObjectWrapper\_comtypes方法), 52
- \_\_adjustProcessPrivileges() (qt4c.util.Process 方法), 75
- \_\_check\_valid() (qt4c.webview.iwebview.IEDriver 方法), 41
- \_\_check\_valid() (qt4c.webview.iwebview.iedriver.IEDriver 方法), 40
- \_\_click() (qt4c.control.Control 方法), 54
- \_\_clickSlowly() (qt4c.mouse.Mouse 静态方法), 62
- \_\_clickSlowly() (qt4c.webview.iwebview.Mouse 静态方法), 43
- \_\_cursor\_flags() (在 qt4c.mouse 模块中), 62
- \_\_fields\_\_ (qt4c.wintypes.APPBAR\_DATA 属性), 87
- \_\_fields\_\_ (qt4c.wintypes.BITMAP 属性), 88
- \_\_fields\_\_ (qt4c.wintypes.BITMAPFILEHEADER 属性), 88
- \_\_fields\_\_ (qt4c.wintypes.BITMAPINFO 属性), 88
- \_\_fields\_\_ (qt4c.wintypes.BITMAPINFOHEADER 属性), 88
- \_\_fields\_\_ (qt4c.wintypes.DIBSECTION 属性), 88
- \_\_fields\_\_ (qt4c.wintypes.LVITEM 属性), 87
- \_\_fields\_\_ (qt4c.wintypes.LVITEM64 属性), 87
- \_\_fields\_\_ (qt4c.wintypes.MODULEENTRY32 属性), 88
- \_\_fields\_\_ (qt4c.wintypes.PROCESSENTRY32 属性), 87
- \_\_fields\_\_ (qt4c.wintypes.RECT 属性), 86
- \_\_fields\_\_ (qt4c.wintypes.RGBTRIPLE 属性), 88
- \_\_fields\_\_ (qt4c.wintypes.TBBUTTON 属性), 87
- \_\_fields\_\_ (qt4c.wintypes.THREADENTRY32 属性), 88
- \_\_fields\_\_ (qt4c.wintypes.TRAYDATA 属性), 87
- \_\_fields\_\_ (qt4c.wintypes.TVITEM 属性), 87
- \_\_find\_by\_name() (在 qt4c.qpath 模块中), 67
- \_\_find\_controls\_recur() (qt4c.qpath.QPath 方法), 66
- \_\_getClickXY() (qt4c.control.Control 方法), 55
- \_\_getSubMenuItemCount() (qt4c.wincontrols.Menu 方法), 85
- \_\_getTestByIndex() (qt4c.wincontrols.ComboBox 方法), 83
- \_\_get\_context\_id() (qt4c.webview.chromewebview.chromedriver.WebView 方法), 36
- \_\_get\_default\_browser1() (在 browser 模块中), 92
- \_\_get\_default\_browser2() (在 browser 模块中), 92
- \_\_get\_document() (qt4c.webview.iwebview.IEDriver 方法), 41
- \_\_get\_document() (qt4c.webview.iwebview.iedriver.IEDriver 方法), 40
- \_\_get\_frame() (qt4c.webview.chromewebview.ChromeWebView 方法), 39
- \_\_get\_frame\_id\_by\_xpath() (qt4c.webview.chromewebview.ChromeWebView 方法), 39
- \_\_get\_frame\_window\_by\_xpath() (qt4c.webview.iwebview.IEWebView 方法), 45
- \_\_get\_page\_cls() (browser.ie.IEBrowser 方法), 92
- \_\_get\_pid\_by\_port() (在 qt4c.webview.chromewebview 模块中), 39
- \_\_getrect() (qt4c.uiaccontrols.Control 方法), 70
- \_\_handle\_offset() (qt4c.webview.base.WebViewBase 方法), 46
- \_\_handle\_offset() (qt4c.webview.chromewebview.WebViewBase 方法), 38
- \_\_handle\_offset() (qt4c.webview.iwebview.WebViewBase 方法), 42
- \_\_handle\_result() (qt4c.webview.base.WebViewBase 方法), 46
- \_\_handle\_result() (qt4c.webview.chromewebview.WebViewBase 方法), 38
- \_\_handle\_result() (qt4c.webview.iwebview.WebViewBase 方法), 41
- \_\_handle\_title() (browser.chrome.ChromeBrowser 方法), 89
- \_\_init() (qt4c.webview.chromewebview.chromedriver.WebkitDebugger



方法), 36  
 \_\_init\_com\_obj() (qt4c.webview.iewebview.IEDriver 方法), 41  
 \_\_init\_com\_obj() (qt4c.webview.iewebview.iedriver.IEDriver 方法), 40  
 \_\_init\_uiaobj() (qt4c.uiacontrols.Control 方法), 70  
 \_\_init\_wndobj() (qt4c.wincontrols.Control 方法), 78  
 \_\_inner\_click() (qt4c.webview.base.WebViewBase 方法), 46  
 \_\_inner\_click() (qt4c.webview.chromewebview.WebViewBase 方法), 38  
 \_\_inner\_click() (qt4c.webview.iewebview.WebViewBase 方法), 42  
 \_\_inner\_long\_click() (qt4c.webview.base.WebViewBase 方法), 46  
 \_\_inner\_long\_click() (qt4c.webview.chromewebview.WebViewBase 方法), 38  
 \_\_inner\_long\_click() (qt4c.webview.iewebview.WebViewBase 方法), 42  
 \_\_inputKey() (qt4c.keyboard.Key 方法), 60  
 \_\_isExtendedKey() (qt4c.keyboard.Key 方法), 60  
 \_\_isPressed() (qt4c.keyboard.Key 方法), 60  
 \_\_isToggled() (qt4c.keyboard.Key 方法), 60  
 \_\_keyclass (qt4c.keyboard.Keyboard 属性), 60  
 \_\_last\_click\_time (qt4c.mouse.Mouse 属性), 62  
 \_\_last\_click\_time (qt4c.webview.iewebview.Mouse 属性), 42  
 \_\_match\_control() (qt4c.qpath.QPath 方法), 66  
 \_\_mouse\_msg() (在 qt4c.mouse 模块中), 62  
 \_\_mouse\_msg\_param() (在 qt4c.mouse 模块中), 62  
 \_\_mouse\_ncmsg\_param() (在 qt4c.mouse 模块中), 62  
 \_\_parse() (qt4c.qpath.QPath 方法), 67  
 \_\_parse\_keys() (qt4c.keyboard.Keyboard 静态方法), 60  
 \_\_parse\_property() (qt4c.qpath.QPath 方法), 66  
 \_\_postKey() (qt4c.keyboard.Key 方法), 60  
 \_\_pressedkey (qt4c.keyboard.Keyboard 属性), 60  
 \_\_retry\_for\_access\_denied() (qt4c.webview.iewebview.IEDriver 方法), 41  
 \_\_retry\_for\_access\_denied() (qt4c.webview.iewebview.iedriver.IEDriver 方法), 40  
 \_\_scan2vkey() (在 qt4c.keyboard 模块中), 59  
 \_\_screenshot() (在 qt4c.testcase 模块中), 68  
 \_\_timeout (browser.ie.IEWindow 属性), 91  
 \_\_timeout (qt4c.control.Control 属性), 54  
 \_\_totalapps (qt4c.app.App 属性), 53  
 \_\_wait\_for\_disabled\_or\_invisible() (qt4c.wincontrols.Window 方法), 80  
 \_\_wait\_for\_ready() (qt4c.webview.chromewebview.chromedriver.WebViewBase 方法), 36  
 \_\_wait\_for\_response() (qt4c.webview.chromewebview.chromedriver.WebViewBase 方法), 37

## A

accChildCount (qt4c.accessible.\_\_AccessibleObjectWrapper\_comtypes 属性), 51  
 accChildCount (qt4c.accessible.\_\_AccessibleObjectWrapper\_win32comtypes 属性), 51  
 accChildCount (qt4c.accessible.AccessibleObject 属性), 53  
 accDescription (qt4c.accessible.\_\_AccessibleObjectWrapper\_comtypes 属性), 51  
 accDescription (qt4c.accessible.\_\_AccessibleObjectWrapper\_win32comtypes 属性), 51  
 accDescription (qt4c.accessible.AccessibleObject 属性), 52  
 AccessibleObject (qt4c.accessible 中的类), 52  
 AccessibleObject (qt4c.wincontrols.Control 属性), 78  
 accFocus (qt4c.accessible.\_\_AccessibleObjectWrapper\_comtypes 属性), 51  
 accFocus (qt4c.accessible.\_\_AccessibleObjectWrapper\_win32comtypes 属性), 51  
 accFocus (qt4c.accessible.AccessibleObject 属性), 52  
 accName (qt4c.accessible.\_\_AccessibleObjectWrapper\_comtypes 属性), 51  
 accName (qt4c.accessible.\_\_AccessibleObjectWrapper\_win32comtypes 属性), 51  
 accName (qt4c.accessible.AccessibleObject 属性), 52  
 accParent (qt4c.accessible.\_\_AccessibleObjectWrapper\_comtypes 属性), 52  
 accParent (qt4c.accessible.\_\_AccessibleObjectWrapper\_win32comtypes 属性), 52

- 属性), 51
  - accParent (qt4c.accessible.AccessibleObject 属性), 53
  - accRole (qt4c.accessible.\_AccessibleObjectWrapper\_compatible 属性), 51
  - accRole (qt4c.accessible.\_AccessibleObjectWrapper\_win32com 属性), 51
  - accRole (qt4c.accessible.AccessibleObject 属性), 52
  - accState (qt4c.accessible.\_AccessibleObjectWrapper\_compatible 属性), 51
  - accState (qt4c.accessible.\_AccessibleObjectWrapper\_win32com 属性), 51
  - accState (qt4c.accessible.AccessibleObject 属性), 52
  - accValue (qt4c.accessible.\_AccessibleObjectWrapper\_compatible 属性), 52
  - accValue (qt4c.accessible.\_AccessibleObjectWrapper\_win32com 属性), 51
  - accValue (qt4c.accessible.AccessibleObject 属性), 53
  - activate() (qt4c.webcontrols.WebPage 方法), 76
  - activate() (qt4c.webview.base.WebViewBase 方法), 46
  - activate() (qt4c.webview.chromewebview.WebViewBaseBuffer (qt4c.util.ProcessMem 属性), 73 方法), 38
  - activate() (qt4c.webview.iewebview.WebViewBase 方法), 42
  - All (qt4c.util.Point 属性), 72
  - All (qt4c.util.Rectangle 属性), 72
  - App (qt4c.app 中的类), 53
  - APPBARDATA (qt4c.wintypes 中的类), 87
  - appendModifierKey() (qt4c.keyboard.Key 方法), 59
- ## B
- BITMAP (qt4c.wintypes 中的类), 88
  - BITMAPFILEHEADER (qt4c.wintypes 中的类), 88
  - BITMAPINFO (qt4c.wintypes 中的类), 88
  - BITMAPINFOHEADER (qt4c.wintypes 中的类), 87
  - Bottom (qt4c.util.Rectangle 属性), 72
  - BoundingRect (qt4c.control.Control 属性), 54
  - BoundingRect (qt4c.uiacontrols.Control 属性), 69
  - BoundingRect (qt4c.wincontrols.\_TrayIcon 属性), 82
  - BoundingRect (qt4c.wincontrols.Control 属性), 77
  - BoundingRect (qt4c.wincontrols.ListViewItem 属性), 79
  - BoundingRect (qt4c.wincontrols.MenuItem 属性), 84
  - BoundingRect (qt4c.wincontrols.TreeViewItem 属性), 84
  - bringForeground() (qt4c.wincontrols.Window 方法), 80
  - BrowseDialog (qt4c.filedialog 中的类), 58
  - BrowseFolderDialog (qt4c.filedialog 中的类), 58
  - browser (模块), 89
  - browser.chrome (模块), 89
  - browser.firefox (模块), 90
  - browser.ie (模块), 90
  - browser\_list() (在 browser 模块中), 92
  - browser\_type (qt4c.webview.base.WebViewBase 属性), 46
  - browser\_type (qt4c.webview.chromewebview.WebViewBase 属性), 37
  - browser\_type (qt4c.webview.iewebview.WebViewBase 属性), 41
- ## C
- Caption (qt4c.wincontrols.Control 属性), 77
  - Center (qt4c.util.Rectangle 属性), 72
  - Children (qt4c.control.Control 属性), 54
  - Children (qt4c.uiacontrols.Control 属性), 69
  - Children (qt4c.wincontrols.Control 属性), 77
  - Chrome (browser.EnumBrowserType 属性), 92
  - ChromeBrowser (browser.chrome 中的类), 89
  - ChromeDriver (qt4c.webview.chromewebview 中的类), 38
  - ChromeDriver (qt4c.webview.chromewebview.chromedriver 中的类), 36
  - ChromeDriverError, 36
  - ChromeWebView (qt4c.webview.chromewebview 中的类), 39
  - ClassName (qt4c.uiacontrols.Control 属性), 70
  - ClassName (qt4c.wincontrols.Control 属性), 77
  - clean\_test (qt4c.testcase.ClientTestCase 属性), 68
  - cleanTest() (qt4c.testcase.ClientTestCase 方法), 68
  - clear() (qt4c.keyboard.Keyboard 静态方法), 61

clear\_cache() (browser.chrome.ChromeBrowser 方法), 90

clearAll() (qt4c.app.App 静态方法), 53

clearLocator() (qt4c.control.ControlContainer 方法), 56

click() (qt4c.control.Control 方法), 54

click() (qt4c.mouse.Mouse 静态方法), 62

click() (qt4c.uiaccontrols.Control 方法), 70

click() (qt4c.util.MetisView 方法), 75

click() (qt4c.webview.base.WebViewBase 方法), 46

click() (qt4c.webview.chromewebview.ChromeWebView 方法), 39

click() (qt4c.webview.chromewebview.WebViewBase 方法), 38

click() (qt4c.webview.iewebview.Mouse 静态方法), 42

click() (qt4c.webview.iewebview.WebViewBase 方法), 42

click() (qt4c.wincontrols.\_TrayIcon 方法), 82

click() (qt4c.wincontrols.Control 方法), 79

click() (qt4c.wincontrols.MenuItem 方法), 85

ClientTestCase (qt4c.testcase 中的类), 68

close() (browser.chrome.ChromeBrowser 方法), 90

close() (browser.ie.IEBrowser 方法), 91

close() (qt4c.webcontrols.WebPage 方法), 76

close() (qt4c.wincontrols.Window 方法), 80

closeAllSysMenuWindow() (qt4c.wincontrols.Menu 静态方法), 85

code (qt4c.webview.chromewebview.chromedriver.ChromeDriverError 属性), 36

collapse() (qt4c.uiaccontrols.ComboBox 方法), 71

ComboBox (qt4c.uiaccontrols 中的类), 71

ComboBox (qt4c.wincontrols 中的类), 83

Control (qt4c.control 中的类), 54

Control (qt4c.uiaccontrols 中的类), 69

Control (qt4c.wincontrols 中的类), 77

CONTROL\_TYPES (qt4c.qpath.QPath 属性), 66

ControlContainer (qt4c.control 中的类), 56

ControlId (qt4c.wincontrols.Control 属性), 77

Controls (qt4c.control.ControlContainer 属性), 56

ControlType (qt4c.uiaccontrols.Control 属性), 69

ControlWalker() (在 qt4c.uiaccontrols 模块中), 69

Count (qt4c.wincontrols.ComboBox 属性), 83

count() (qt4c.wincontrols.TreeView 方法), 83

## D

destroy() (qt4c.wincontrols.\_TrayIcon 方法), 82

DIBSECTION (qt4c.wintypes 中的类), 88

DISABLED (qt4c.wincontrols.MenuItem.EnumMenuItemState 属性), 84

double\_click() (qt4c.util.MetisView 方法), 75

double\_click() (qt4c.webview.base.WebViewBase 方法), 46

double\_click() (qt4c.webview.chromewebview.WebViewBase 方法), 38

double\_click() (qt4c.webview.iewebview.WebViewBase 方法), 42

doubleClick() (qt4c.control.Control 方法), 55

drag() (qt4c.control.Control 方法), 55

drag() (qt4c.mouse.Mouse 静态方法), 63

drag() (qt4c.webview.iewebview.Mouse 静态方法), 44

## E

Edit (qt4c.uiaccontrols 中的类), 70

empty\_invoke() (qt4c.uiaccontrols.Control 方法), 70

enable\_runtime() (qt4c.webview.chromewebview.chromedriver.Webkit 方法), 37

Enabled (qt4c.uiaccontrols.Control 属性), 69

Enabled (qt4c.wincontrols.Control 属性), 77

ensureVisible() (qt4c.wincontrols.TreeViewItem 方法), 83

EnumAccessibleObjectRole (qt4c.accessible 中的类), 47

EnumAccessibleObjectState (qt4c.accessible 中的类), 50

EnumBrowserType (browser 中的类), 92

EnumQPathKey (qt4c.qpath 中的类), 65

EnumUIType (qt4c.qpath 中的类), 65

equal() (qt4c.control.Control 方法), 55

equal() (qt4c.uiaccontrols.Control 方法), 70

equal() (qt4c.wincontrols.\_TrayIcon 方法), 82

equal() (qt4c.wincontrols.Control 方法), 79

eval\_script() (qt4c.webview.chromewebview.chromedriver.WebkitDev 方法), 37

eval\_script() (qt4c.webview.chromewebview.ChromeWebView.extra\_fail\_record() 方法), 39

eval\_script() (qt4c.webview.iewebview.IEDriver 方法), 41

eval\_script() (qt4c.webview.iewebview.iedriver.IEDriver.get\_frame\_window() 方法), 40

eval\_script() (qt4c.webview.iewebview.IEWebView.get\_frame\_window() 方法), 45

exist() (qt4c.uiacontrols.Control 方法), 70

exist() (qt4c.wincontrols.Control 方法), 79

expand() (qt4c.uiacontrols.ComboBox 方法), 71

ExploreFileFolder (qt4c.filedialog 中的类), 58

ExStyle (qt4c.wincontrols.Control 属性), 78

## F

FileDialog (qt4c.filedialog 中的类), 57

FileFolder (qt4c.filedialog 中的类), 58

FilePath (qt4c.filedialog.FileDialog 属性), 57

find\_by\_url() (browser.chrome.ChromeBrowser 方法), 89

find\_by\_url() (browser.ie.IEBrowser 方法), 92

find\_UIAElm() (在 qt4c.uiacontrols 模块中), 69

Firefox (browser.EnumBrowserType 属性), 92

FirefoxApp (browser.firefox 中的类), 90

fnt() (在 qt4c.webview 模块中), 47

## G

get\_accName() (qt4c.accessible.\_\_AccessibleObjectWrapper, comtypes 方法), 52

get\_accName() (qt4c.accessible.\_\_AccessibleObjectWrapper, win32com 方法), 51

get\_accName() (qt4c.accessible.AccessibleObject 方法), 53

get\_browser\_path() (browser.chrome.ChromeBrowser 静态方法), 90

get\_chrome\_window\_list() (browser.chrome.ChromeBrowser 方法), 89

get\_debugger() (qt4c.webview.chromewebview.ChromeDriver 方法), 38

get\_debugger() (qt4c.webview.chromewebview.chromedriver.ChromeDriver 方法), 36

get\_default\_browser() (在 browser 模块中), 93

get\_extra\_fail\_record() (qt4c.testcase.ClientTestCase 方法), 68

get\_frame\_tree() (qt4c.webview.chromewebview.chromedriver.Webki 方法), 37

get\_frame\_window() (qt4c.webview.iewebview.IEDriver 方法), 41

get\_frame\_window() (qt4c.webview.iewebview.iedriver.IEDriver 方法), 40

get\_frames() (qt4c.webview.iewebview.IEDriver 方法), 41

get\_frames() (qt4c.webview.iewebview.iedriver.IEDriver 方法), 40

get\_metis\_view() (qt4c.control.Control 方法), 56

get\_next\_avail\_port() (在 browser.chrome 模块中), 90

get\_page\_cls() (browser.chrome.ChromeBrowser 方法), 89

get\_page\_list() (qt4c.webview.chromewebview.ChromeDriver 方法), 38

get\_page\_list() (qt4c.webview.chromewebview.chromedriver.Chrome 方法), 36

get\_path() (browser.ie.IEBrowser 静态方法), 91

get\_pid\_by\_port() (在 qt4c.webview.chromewebview 模块中), 39

get\_version() (browser.ie.IEBrowser 静态方法), 91

getCursorType() (qt4c.mouse.Mouse 静态方法), 64

getCursorType() (qt4c.webview.iewebview.Mouse 静态方法), 45

getDpi() (在 qt4c.util 模块中), 75

getEncoding() (在 qt4c.util 模块中), 73

getErrorPath() (qt4c.qpath.QPath 方法), 67

getFullPath() (qt4c.wincontrols.ComboBox 方法), 83

getPosition() (qt4c.mouse.Mouse 静态方法), 64

getPosition() (qt4c.webview.iewebview.Mouse 静态方法), 45

GetProcessesByName() (qt4c.util.Process 静态方法), 74

getToolTips() (在 qt4c.util 模块中), 74

GRAYED (qt4c.wincontrols.MenuItem.EnumMenuItemState 属性), 84

## H

- handle\_\_error\_\_page() (qt4c.webview.iewebview.IEDriver 方法), 41
  - handle\_\_error\_\_page() (qt4c.webview.iewebview.iedriver.IEDriver 方法), 40
  - handle\_\_position() (qt4c.mouse.Mouse 静态方法), 62
  - handle\_\_position() (qt4c.webview.iewebview.Mouse 静态方法), 42
  - handler() (在 qt4c.webview 模块中), 47
  - hasControlKey() (qt4c.control.ControlContainer 方法), 56
  - HasKeyboardFocus (qt4c.uiacontrols.Control 属性), 69
  - Height (qt4c.uiacontrols.Control 属性), 69
  - Height (qt4c.util.Rectangle 属性), 72
  - Height (qt4c.wincontrols.Control 属性), 78
  - hide() (qt4c.wincontrols.Window 方法), 81
  - highLight() (qt4c.util.Rectangle 方法), 73
  - highlight() (qt4c.webview.iewebview.IEWebView 方法), 45
  - hover() (qt4c.control.Control 方法), 55
  - hover() (qt4c.webview.base.WebViewBase 方法), 46
  - hover() (qt4c.webview.chromewebview.WebViewBase 方法), 38
  - hover() (qt4c.webview.iewebview.WebViewBase 方法), 42
  - HWnd (qt4c.uiacontrols.Control 属性), 69
  - Hwnd (qt4c.uiacontrols.Control 属性), 69
  - hwnd (qt4c.uiacontrols.Control 属性), 69
  - HWnd (qt4c.wincontrols.\_CWindow 属性), 77
  - HWnd (qt4c.wincontrols.Control 属性), 78
  - hwnd (qt4c.wincontrols.Control 属性), 78
  - HWnd (qt4c.wincontrols.TreeViewItem 属性), 83
  - hwnd (qt4c.wincontrols.TreeViewItem 属性), 83
- I
- IE (browser.EnumBrowserType 属性), 92
  - ie\_\_window (browser.ie.IEWindow 属性), 91
  - IEApp (browser.ie 中的类), 92
  - IEBrowser (browser.ie 中的类), 91
  - IEDriver (qt4c.webview.iewebview 中的类), 41
  - IEDriver (qt4c.webview.iewebview.iedriver 中的类), 40
  - IEDriverError, 40
  - IEWebView (qt4c.webview.iewebview 中的类), 45
  - IEWindow (browser.ie 中的类), 91
  - init\_\_test (qt4c.testcase.ClientTestCase 属性), 68
  - initTest() (qt4c.testcase.ClientTestCase 方法), 68
  - input() (qt4c.uiacontrols.ComboBox 方法), 71
  - input() (qt4c.uiacontrols.Edit 方法), 70
  - inputKey() (qt4c.keyboard.Key 方法), 60
  - inputKeys() (qt4c.keyboard.Keyboard 静态方法), 60
  - inst\_\_dict (qt4c.webview.chromewebview.ChromeDriver 属性), 38
  - inst\_\_dict (qt4c.webview.chromewebview.chromedriver.ChromeDriver 属性), 36
  - INSTANCE (qt4c.qpath.EnumQPathKey 属性), 65
  - is\_\_64bits() (在 qt4c.keyboard 模块中), 59
  - is\_\_port\_\_occupied() (在 browser.chrome 模块中), 90
  - is\_\_select() (qt4c.uiacontrols.RadioButton 方法), 71
  - is\_\_system\_\_locked() (在 qt4c.util 模块中), 75
  - isChildCtrlExist() (qt4c.control.ControlContainer 方法), 57
  - isInRect() (qt4c.util.Rectangle 方法), 72
  - isPressed() (qt4c.keyboard.Keyboard 静态方法), 61
  - IsSeperator (qt4c.wincontrols.MenuItem 属性), 84
  - isTroggled() (qt4c.keyboard.Keyboard 静态方法), 61
  - ItemCount (qt4c.wincontrols.ListView 属性), 80
  - Items (qt4c.wincontrols.ListView 属性), 80
  - Items (qt4c.wincontrols.TrayNotifyBar 属性), 81
  - Items (qt4c.wincontrols.TrayTaskBar 属性), 82
  - Items (qt4c.wincontrols.TreeView 属性), 83
  - Items (qt4c.wincontrols.TreeViewItem 属性), 83
  - IUIAutomation() (在 qt4c.uiacontrols 模块中), 68
- K
- Key (qt4c.keyboard 中的类), 59
  - Keyboard (qt4c.keyboard 中的类), 60
  - KEYEVENTF\_\_EXTENDEDKEY (qt4c.keyboard.\_\_KeyboardEvent 属性), 59
  - KEYEVENTF\_\_KEYUP (qt4c.keyboard.\_\_KeyboardEvent 属性), 59



KEYEVENTF\_SCANCODE

(qt4c.keyboard.\_KeyboardEvent 属性), 59

KEYEVENTF\_UNICODE

(qt4c.keyboard.\_KeyboardEvent 属性), 59

KeyInputError, 59

killall() (browser.chrome.ChromeBrowser 静态方法), 90

killAll() (browser.ie.IEApp 静态方法), 92

killall() (browser.ie.IEBrowser 静态方法), 91

killAll() (qt4c.app.App 静态方法), 53

## L

Left (qt4c.util.Rectangle 属性), 72

List View (qt4c.wincontrols 中的类), 80

ListViewItem (qt4c.wincontrols 中的类), 79

Live (qt4c.util.Process 属性), 74

long\_click() (qt4c.util.MetisView 方法), 75

long\_click() (qt4c.webview.base.WebViewBase 方法), 46

long\_click() (qt4c.webview.chromewebview.WebViewBase 方法), 38

long\_click() (qt4c.webview.iewebview.WebViewBase 方法), 42

LVITEM (qt4c.wintypes 中的类), 87

LVITEM64 (qt4c.wintypes 中的类), 87

## M

MAPVK\_VK\_TO\_VSC() (在 qt4c.keyboard 模块中), 59

MATCH\_FUNCS (qt4c.qpath.QPath 属性), 66

MAX\_DEPTH (qt4c.qpath.EnumQPathKey 属性), 65

maximize() (qt4c.wincontrols.Window 方法), 81

Maximized (qt4c.wincontrols.Window 属性), 80

Menu (qt4c.wincontrols 中的类), 85

MenuItem (qt4c.wincontrols 中的类), 84

MenuItem.EnumMenuItemState (qt4c.wincontrols 中的类), 84

MenuItems (qt4c.wincontrols.Menu 属性), 85

message (qt4c.webview.chromewebview.chromedriver.ChromeDriver 属性), 36

MetisView (qt4c.util 中的类), 75

minimize() (qt4c.wincontrols.Window 方法), 81

Minimized (qt4c.uiacontrols.UIAWindows 属性), 70

Minimized (qt4c.wincontrols.Window 属性), 80

MODULEENTRY32 (qt4c.wintypes 中的类), 88

Mouse (qt4c.mouse 中的类), 62

Mouse (qt4c.webview.iewebview 中的类), 42

MouseClickedType (qt4c.mouse 中的类), 62

MouseClickedType (qt4c.webview.iewebview 中的类), 45

MouseCursorType (qt4c.mouse 中的类), 62

MouseFlag (qt4c.mouse 中的类), 62

MouseFlag (qt4c.webview.iewebview 中的类), 45

move() (qt4c.mouse.Mouse 静态方法), 64

move() (qt4c.webview.iewebview.Mouse 静态方法), 44

move() (qt4c.wincontrols.Window 方法), 81

MsgSyncer (qt4c.util 中的类), 74

myEncode() (在 qt4c.util 模块中), 73

## N

Name (qt4c.uiacontrols.Control 属性), 69

NextSibling (qt4c.wincontrols.TreeViewItem 属性), 83

NORMAL (qt4c.wincontrols.MenuItem.EnumMenuItemState 属性), 84

## O

on\_close() (qt4c.webview.chromewebview.chromedriver.WebkitDebugger 方法), 36

on\_error() (qt4c.webview.chromewebview.chromedriver.WebkitDebugger 方法), 36

on\_message() (qt4c.webview.chromewebview.chromedriver.WebkitDebugger 方法), 36

on\_open() (qt4c.webview.chromewebview.chromedriver.WebkitDebugger 方法), 36

on\_recv\_notify\_msg() (qt4c.webview.chromewebview.chromedriver.WebkitDebugger 方法), 36

open() (qt4c.filedialog.OpenFileDialog 方法), 58

open() (qt4c.filedialog.SelectFileDialog 方法), 58

open\_url() (browser.chrome.ChromeBrowser 方法), 89

open\_url() (browser.ie.IEBrowser 方法), 91  
 OpenFileDialog (qt4c.filedialog 中的类), 57  
 OPERATORS (qt4c.qpath.QPath 属性), 66  
 os\_type (qt4c.util.MetisView 属性), 75  
 OwnerWindow (qt4c.wincontrols.Window 属性), 80

## P

Parent (qt4c.uiacontrols.Control 属性), 69  
 Parent (qt4c.wincontrols.Control 属性), 78  
 pid (browser.ie.IEWindow 属性), 91  
 pid\_event\_map (qt4c.util.MsgSyncer 属性), 74  
 Point (qt4c.util 中的类), 72  
 PopupWindow (qt4c.wincontrols.Window 属性), 80  
 postClick() (qt4c.mouse.Mouse 静态方法), 63  
 postClick() (qt4c.webview.iewebview.Mouse 静态方法), 43  
 postKey() (qt4c.keyboard.Key 方法), 60  
 postKeys() (qt4c.keyboard.Keyboard 静态方法), 61  
 postMove() (qt4c.mouse.Mouse 静态方法), 64  
 postMove() (qt4c.webview.iewebview.Mouse 静态方法), 44  
 press() (qt4c.mouse.Mouse 静态方法), 64  
 press() (qt4c.webview.iewebview.Mouse 静态方法), 44  
 pressKey() (qt4c.keyboard.Keyboard 静态方法), 61  
 Process (qt4c.util 中的类), 74  
 PROCESSENTRY32 (qt4c.wintypes 中的类), 87  
 ProcessId (qt4c.uiacontrols.Control 属性), 69  
 ProcessId (qt4c.util.Process 属性), 74  
 ProcessId (qt4c.wincontrols.\_TrayIcon 属性), 82  
 ProcessId (qt4c.wincontrols.Control 属性), 78  
 ProcessMem (qt4c.util 中的类), 73  
 ProcessName (qt4c.util.Process 属性), 74  
 ProcessPath (qt4c.util.Process 属性), 74  
 PROPERTY\_SEP (qt4c.qpath.QPath 属性), 66

## Q

QPath (qt4c.qpath 中的类), 65  
 QPathError, 65  
 QQBrowser (browser.EnumBrowserType 属性), 92  
 qt4c (模块), 35  
 qt4c.accessible (模块), 47

qt4c.app (模块), 53  
 qt4c.control (模块), 54  
 qt4c.exceptions (模块), 57  
 qt4c.filedialog (模块), 57  
 qt4c.keyboard (模块), 58  
 qt4c.mouse (模块), 61  
 qt4c.qpath (模块), 65  
 qt4c.testcase (模块), 67  
 qt4c.uiacontrols (模块), 68  
 qt4c.util (模块), 71  
 qt4c.version (模块), 75  
 qt4c.webcontrols (模块), 76  
 qt4c.webview (模块), 35  
 qt4c.webview.base (模块), 45  
 qt4c.webview.chromewebview (模块), 35  
 qt4c.webview.chromewebview.chromedriver (模块), 35  
 qt4c.webview.iewebview (模块), 39  
 qt4c.webview.iewebview.iedriver (模块), 39  
 qt4c.wincontrols (模块), 76  
 qt4c.wintypes (模块), 85  
 quit() (qt4c.app.App 方法), 53  
 quitAll() (qt4c.app.App 静态方法), 53

## R

RadioButton (qt4c.uiacontrols 中的类), 71  
 RawWalker() (在 qt4c.uiacontrols 模块中), 68  
 read() (qt4c.util.ProcessMem 方法), 73  
 rect (qt4c.util.MetisView 属性), 75  
 rect (qt4c.webview.base.WebViewBase 属性), 46  
 rect (qt4c.webview.chromewebview.WebViewBase 属性), 37  
 rect (qt4c.webview.iewebview.WebViewBase 属性), 41  
 RECT (qt4c.wintypes 中的类), 86  
 Rectangle (qt4c.util 中的类), 72  
 refresh() (qt4c.wincontrols.TrayNotifyBar 方法), 81  
 release() (qt4c.mouse.Mouse 静态方法), 64  
 release() (qt4c.webview.iewebview.Mouse 静态方法), 44  
 releaseKey() (qt4c.keyboard.Keyboard 静态方法), 61  
 remote\_inject\_dll() (在 qt4c.util 模块中), 74

resize() (qt4c.wincontrols.Window 方法), 81	(qt4c.accessible.EnumAccessibleObjectRole 属性), 48
restore() (qt4c.wincontrols.Window 方法), 81	
RGBTRIPLE (qt4c.wintypes 中的类), 88	ROLE_SYSTEM_CHECKBUTTON
Right (qt4c.util.Rectangle 属性), 72	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
right_click() (qt4c.webview.base.WebViewBase 方法), 46	ROLE_SYSTEM_CLIENT
right_click() (qt4c.webview.chromewebview.WebViewBase 方法), 38	(qt4c.accessible.EnumAccessibleObjectRole 属性), 48
right_click() (qt4c.webview.iewebview.WebViewBase 方法), 42	ROLE_SYSTEM_CLOCK
rightClick() (qt4c.control.Control 方法), 55	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
ROLE_SYSTEM_ALERT	ROLE_SYSTEM_COLUMN
(qt4c.accessible.EnumAccessibleObjectRole 属性), 48	(qt4c.accessible.EnumAccessibleObjectRole 属性), 48
ROLE_SYSTEM_ANIMATION	ROLE_SYSTEM_COLUMNHEADER
(qt4c.accessible.EnumAccessibleObjectRole 属性), 49	(qt4c.accessible.EnumAccessibleObjectRole 属性), 48
ROLE_SYSTEM_APPLICATION	ROLE_SYSTEM_COMBOBOX
(qt4c.accessible.EnumAccessibleObjectRole 属性), 48	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
ROLE_SYSTEM_BORDER	ROLE_SYSTEM_CURSOR
(qt4c.accessible.EnumAccessibleObjectRole 属性), 48	(qt4c.accessible.EnumAccessibleObjectRole 属性), 48
ROLE_SYSTEM_BUTTONDROPDOWN	ROLE_SYSTEM_DIAGRAM
(qt4c.accessible.EnumAccessibleObjectRole 属性), 49	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
ROLE_SYSTEM_BUTTONDROPDOWNGRID	ROLE_SYSTEM_DIAL
(qt4c.accessible.EnumAccessibleObjectRole 属性), 49	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
ROLE_SYSTEM_BUTTONMENU	ROLE_SYSTEM_DIALOG
(qt4c.accessible.EnumAccessibleObjectRole 属性), 49	(qt4c.accessible.EnumAccessibleObjectRole 属性), 48
ROLE_SYSTEM_CARET	ROLE_SYSTEM_DOCUMENT
(qt4c.accessible.EnumAccessibleObjectRole 属性), 48	(qt4c.accessible.EnumAccessibleObjectRole 属性), 48
ROLE_SYSTEM_CELL	ROLE_SYSTEM_DROPLIST
(qt4c.accessible.EnumAccessibleObjectRole 属性), 48	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
ROLE_SYSTEM_CHARACTER	ROLE_SYSTEM_EQUATION
(qt4c.accessible.EnumAccessibleObjectRole 属性), 49	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
ROLE_SYSTEM_CHART	ROLE_SYSTEM_GRAPHIC



(qt4c.accessible.EnumAccessibleObjectRole 属性), 49	(qt4c.accessible.EnumAccessibleObjectRole 属性), 50
ROLE_SYSTEM_GRIP	ROLE_SYSTEM_OUTLINEITEM
(qt4c.accessible.EnumAccessibleObjectRole 属性), 48	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
ROLE_SYSTEM_GROUPING	ROLE_SYSTEM_PAGETAB
(qt4c.accessible.EnumAccessibleObjectRole 属性), 48	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
ROLE_SYSTEM_HELPBALLOON	ROLE_SYSTEM_PAGETABLIST
(qt4c.accessible.EnumAccessibleObjectRole 属性), 48	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
ROLE_SYSTEM_HOTKEYFIELD	ROLE_SYSTEM_PANE
(qt4c.accessible.EnumAccessibleObjectRole 属性), 49	(qt4c.accessible.EnumAccessibleObjectRole 属性), 48
ROLE_SYSTEM_INDICATOR	ROLE_SYSTEM_PROGRESSBAR
(qt4c.accessible.EnumAccessibleObjectRole 属性), 49	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
ROLE_SYSTEM_IPADDRESS	ROLE_SYSTEM_PROPERTYPAGE
(qt4c.accessible.EnumAccessibleObjectRole 属性), 50	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
ROLE_SYSTEM_LINK	ROLE_SYSTEM_PUSHBUTTON
(qt4c.accessible.EnumAccessibleObjectRole 属性), 48	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
ROLE_SYSTEM_LIST	ROLE_SYSTEM_RADIOBUTTON
(qt4c.accessible.EnumAccessibleObjectRole 属性), 49	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
ROLE_SYSTEM_LISTITEM	ROLE_SYSTEM_ROW
(qt4c.accessible.EnumAccessibleObjectRole 属性), 49	(qt4c.accessible.EnumAccessibleObjectRole 属性), 48
ROLE_SYSTEM_MENUBAR	ROLE_SYSTEM_ROWHEADER
(qt4c.accessible.EnumAccessibleObjectRole 属性), 48	(qt4c.accessible.EnumAccessibleObjectRole 属性), 48
ROLE_SYSTEM_MENUITEM	ROLE_SYSTEM_SCROLLBAR
(qt4c.accessible.EnumAccessibleObjectRole 属性), 48	(qt4c.accessible.EnumAccessibleObjectRole 属性), 48
ROLE_SYSTEM_MENUPOPUP	ROLE_SYSTEM_SEPARATOR
(qt4c.accessible.EnumAccessibleObjectRole 属性), 48	(qt4c.accessible.EnumAccessibleObjectRole 属性), 48
ROLE_SYSTEM_OUTLINE	ROLE_SYSTEM_SLIDER
(qt4c.accessible.EnumAccessibleObjectRole 属性), 49	(qt4c.accessible.EnumAccessibleObjectRole 属性), 49
ROLE_SYSTEM_OUTLINEBUTTON	ROLE_SYSTEM_SOUND

(qt4c.accessible.EnumAccessibleObjectRole 属性), 48

ROLE\_SYSTEM\_SPINBUTTON (qt4c.accessible.EnumAccessibleObjectRole 属性), 49

ROLE\_SYSTEM\_SPLITBUTTON (qt4c.accessible.EnumAccessibleObjectRole 属性), 50

ROLE\_SYSTEM\_STATICTEXT (qt4c.accessible.EnumAccessibleObjectRole 属性), 49

ROLE\_SYSTEM\_STATUSBAR (qt4c.accessible.EnumAccessibleObjectRole 属性), 48

ROLE\_SYSTEM\_TABLE (qt4c.accessible.EnumAccessibleObjectRole 属性), 48

ROLE\_SYSTEM\_TEXT (qt4c.accessible.EnumAccessibleObjectRole 属性), 49

ROLE\_SYSTEM\_TITLEBAR (qt4c.accessible.EnumAccessibleObjectRole 属性), 47

ROLE\_SYSTEM\_TOOLBAR (qt4c.accessible.EnumAccessibleObjectRole 属性), 48

ROLE\_SYSTEM\_TOOLTIP (qt4c.accessible.EnumAccessibleObjectRole 属性), 48

ROLE\_SYSTEM\_WHITESPACE (qt4c.accessible.EnumAccessibleObjectRole 属性), 49

ROLE\_SYSTEM\_WINDOW (qt4c.accessible.EnumAccessibleObjectRole 属性), 48

**S**

save() (qt4c.filedialog.SaveAsDialog 方法), 58

SaveAsDialog (qt4c.filedialog 中的类), 58

screenshot() (qt4c.util.MetisView 方法), 75

screenshot() (qt4c.webview.base.WebViewBase 方法), 47

screenshot() (qt4c.webview.chromewebview.chromedriver.WebkitDebu 方法), 37

screenshot() (qt4c.webview.chromewebview.WebViewBase 方法), 38

screenshot() (qt4c.webview.iewebview.WebViewBase 方法), 42

scroll() (qt4c.control.Control 方法), 55

scroll() (qt4c.mouse.Mouse 静态方法), 64

scroll() (qt4c.webview.base.WebViewBase 方法), 46

scroll() (qt4c.webview.chromewebview.WebViewBase 方法), 38

scroll() (qt4c.webview.iewebview.Mouse 静态方法), 45

scroll() (qt4c.webview.iewebview.WebViewBase 方法), 42

search() (qt4c.qpath.QPath 方法), 67

search\_chrome\_webview() (browser.chrome.ChromeBrowser 方法), 90

searh\_ie\_window() (browser.ie.IEBrowser 静态方法), 91

select() (qt4c.wincontrols.TreeViewItem 方法), 84

Selected (qt4c.wincontrols.TreeViewItem 属性), 83

SelectedIndex (qt4c.wincontrols.ComboBox 属性), 83

SelectFileDialog (qt4c.filedialog 中的类), 58

selectKeyClass() (qt4c.keyboard.Keyboard 静态方法), 60

send\_keys() (qt4c.util.MetisView 方法), 75

send\_keys() (qt4c.webview.base.WebViewBase 方法), 46

send\_keys() (qt4c.webview.chromewebview.WebViewBase 方法), 38

send\_keys() (qt4c.webview.iewebview.WebViewBase 方法), 42

send\_request() (qt4c.webview.chromewebview.chromedriver.WebkitD 方法), 37

sendClick() (qt4c.mouse.Mouse 静态方法), 62

sendClick() (qt4c.webview.iewebview.Mouse 静态方法), 43

sendKeys() (qt4c.control.Control 方法), 55

sendMove() (qt4c.mouse.Mouse 静态方法), 64

sendMove() (qt4c.webview.iewebview.Mouse 静态方法), 44

sendNCClick() (qt4c.mouse.Mouse 静态方法), 63	属性), 51
sendNCClick() (qt4c.webview.iewebview.Mouse 静态方法), 43	STATE_SYSTEM_FLOATING (qt4c.accessible.EnumAccessibleObjectState 属性), 50
setFocus() (qt4c.control.Control 方法), 55	STATE_SYSTEM_FOCUSABLE (qt4c.accessible.EnumAccessibleObjectState 属性), 50
SetFocus() (qt4c.uicontrols.Control 方法), 70	STATE_SYSTEM_FOCUSED (qt4c.accessible.EnumAccessibleObjectState 属性), 50
setFocus() (qt4c.wincontrols.Control 方法), 79	STATE_SYSTEM_HASPOPOP (qt4c.accessible.EnumAccessibleObjectState 属性), 51
show() (qt4c.wincontrols.Window 方法), 81	STATE_SYSTEM_HASSUBMENU (qt4c.accessible.EnumAccessibleObjectState 属性), 51
SID_SWebBrowserApp() (在 qt4c.webview.iewebview.iedriver 模块中), 40	STATE_SYSTEM_HOTTRACKED (qt4c.accessible.EnumAccessibleObjectState 属性), 50
SIZEOF() (在 qt4c.util 模块中), 72	STATE_SYSTEM_INDETERMINATE (qt4c.accessible.EnumAccessibleObjectState 属性), 50
State (qt4c.wincontrols._TrayIcon 属性), 82	STATE_SYSTEM_INVISIBLE (qt4c.accessible.EnumAccessibleObjectState 属性), 50
State (qt4c.wincontrols.MenuItem 属性), 84	STATE_SYSTEM_LINKED (qt4c.accessible.EnumAccessibleObjectState 属性), 50
STATE_SYSTEM_ALERT_HIGH (qt4c.accessible.EnumAccessibleObjectState 属性), 51	STATE_SYSTEM_MARQUEED (qt4c.accessible.EnumAccessibleObjectState 属性), 50
STATE_SYSTEM_ALERT_LOW (qt4c.accessible.EnumAccessibleObjectState 属性), 51	STATE_SYSTEM_MIXED (qt4c.accessible.EnumAccessibleObjectState 属性), 50
STATE_SYSTEM_ALERT_MEDIUM (qt4c.accessible.EnumAccessibleObjectState 属性), 51	STATE_SYSTEM_MOVEABLE (qt4c.accessible.EnumAccessibleObjectState 属性), 50
STATE_SYSTEM_ANIMATED (qt4c.accessible.EnumAccessibleObjectState 属性), 50	STATE_SYSTEM_MULTISELECTABLE (qt4c.accessible.EnumAccessibleObjectState 属性), 51
STATE_SYSTEM_BUSY (qt4c.accessible.EnumAccessibleObjectState 属性), 50	STATE_SYSTEM_OFFSCREEN (qt4c.accessible.EnumAccessibleObjectState
STATE_SYSTEM_CHECKED (qt4c.accessible.EnumAccessibleObjectState 属性), 50	
STATE_SYSTEM_COLLAPSED (qt4c.accessible.EnumAccessibleObjectState 属性), 50	
STATE_SYSTEM_DEFAULT (qt4c.accessible.EnumAccessibleObjectState 属性), 50	
STATE_SYSTEM_EXPANDED (qt4c.accessible.EnumAccessibleObjectState 属性), 50	
STATE_SYSTEM_EXTSELECTABLE (qt4c.accessible.EnumAccessibleObjectState	

- 属性), 50
- STATE\_SYSTEM\_PRESSED  
(qt4c.accessible.EnumAccessibleObjectState 属性), 50
- STATE\_SYSTEM\_PROTECTED  
(qt4c.accessible.EnumAccessibleObjectState 属性), 51
- STATE\_SYSTEM\_READONLY  
(qt4c.accessible.EnumAccessibleObjectState 属性), 50
- STATE\_SYSTEM\_SELECTABLE  
(qt4c.accessible.EnumAccessibleObjectState 属性), 50
- STATE\_SYSTEM\_SELECTED  
(qt4c.accessible.EnumAccessibleObjectState 属性), 50
- STATE\_SYSTEM\_SELFVOICING  
(qt4c.accessible.EnumAccessibleObjectState 属性), 50
- STATE\_SYSTEM\_SIZEABLE  
(qt4c.accessible.EnumAccessibleObjectState 属性), 50
- STATE\_SYSTEM\_TRAVERSED  
(qt4c.accessible.EnumAccessibleObjectState 属性), 51
- STATE\_SYSTEM\_UNAVAILABLE  
(qt4c.accessible.EnumAccessibleObjectState 属性), 50
- STATE\_SYSTEM\_VALID  
(qt4c.accessible.EnumAccessibleObjectState 属性), 51
- Style (qt4c.wincontrols.\_TrayIcon 属性), 82
- Style (qt4c.wincontrols.Control 属性), 78
- SubItems (qt4c.wincontrols.ListViewItem 属性), 79
- SubMenu (qt4c.wincontrols.MenuItem 属性), 84
- T**
- TBBUTTON (qt4c.wintypes 中的类), 87
- temp\_path (browser.chrome.ChromeBrowser 属性), 89
- terminate() (qt4c.util.Process 方法), 75
- Text (qt4c.wincontrols.Control 属性), 78
- Text (qt4c.wincontrols.ListViewItem 属性), 79
- Text (qt4c.wincontrols.MenuItem 属性), 84
- Text (qt4c.wincontrols.TreeViewItem 属性), 83
- TextBox (qt4c.wincontrols 中的类), 80
- THREADENTRY32 (qt4c.wintypes 中的类), 88
- ThreadId (qt4c.wincontrols.Control 属性), 78
- Tips (qt4c.wincontrols.\_TrayIcon 属性), 82
- Top (qt4c.util.Rectangle 属性), 72
- TopLevelWindow (qt4c.wincontrols.Control 属性), 78
- TopMost (qt4c.wincontrols.Window 属性), 80
- TRAYDATA (qt4c.wintypes 中的类), 87
- TrayNotifyBar (qt4c.wincontrols 中的类), 81
- TrayTaskBar (qt4c.wincontrols 中的类), 81
- TreeView (qt4c.wincontrols 中的类), 83
- TreeViewItem (qt4c.wincontrols 中的类), 83
- TT (browser.EnumBrowserType 属性), 92
- TVITEM (qt4c.wintypes 中的类), 87
- Type (qt4c.uiacontrols.Control 属性), 69
- U**
- UI\_TYPE (qt4c.qpath.EnumQPathKey 属性), 65
- UIA (qt4c.qpath.EnumUIType 属性), 65
- UIAControlType() (在 qt4c.uiacontrols 模块中), 69
- UIAutomationClient() (在 qt4c.uiacontrols 模块中), 68
- UIAWindows (qt4c.uiacontrols 中的类), 70
- ULONG\_PTR() (在 qt4c.keyboard 模块中), 59
- ULONG\_PTR() (在 qt4c.wintypes 模块中), 86
- unicode() (在 qt4c.util 模块中), 72
- UNKNOWN (qt4c.wincontrols.MenuItem.EnumMenuItemState 属性), 84
- update\_control\_type() (qt4c.qpath.QPath 方法), 66
- updateLocator() (qt4c.control.ControlContainer 方法), 56
- upload\_file() (qt4c.webview.base.UploadFileDialog 方法), 47
- upload\_file() (qt4c.webview.base.WebViewBase 方法), 47
- upload\_file() (qt4c.webview.chromewebview.WebViewBase 方法), 38

upload\_file() (qt4c.webview.iewebview.WebViewBase 方法), 42  
 UploadFileDialog (qt4c.webview.base 中的类), 47  
 Url (browser.chrome.ChromeBrowser 属性), 89  
 Url (browser.ie.IEWindow 属性), 91

## V

Valid (qt4c.uiacontrols.Control 属性), 69  
 Valid (qt4c.wincontrols.Control 属性), 78  
 Value (qt4c.uiacontrols.ComboBox 属性), 71  
 Value (qt4c.uiacontrols.Control 属性), 69  
 version() (在 qt4c.version 模块中), 76  
 Visible (qt4c.uiacontrols.UIAWindows 属性), 70  
 Visible (qt4c.wincontrols.\_TrayIcon 属性), 82  
 Visible (qt4c.wincontrols.Control 属性), 78

## W

wait() (qt4c.util.MsgSyncer 方法), 74  
 wait\_for\_exist() (qt4c.uiacontrols.Control 方法), 70  
 wait\_for\_exist() (qt4c.wincontrols.Control 方法), 79  
 wait\_for\_invalid() (qt4c.wincontrols.Control 方法), 79  
 waitForExist() (qt4c.wincontrols.Control 方法), 79  
 waitForInvalid() (qt4c.wincontrols.Control 方法), 79  
 waitForInvalid() (qt4c.wincontrols.Window 方法), 81  
 waitForInvisible() (qt4c.wincontrols.Window 方法), 81  
 waitForQuit() (qt4c.util.Process 方法), 74  
 waitForValue() (qt4c.control.Control 方法), 55  
 WebkitDebugger (qt4c.webview.chromewebview.chromedriver 中的类), 36  
 WebPage (qt4c.webcontrols 中的类), 76  
 webview (browser.ie.IEWindow 属性), 91  
 WebViewBase (qt4c.webview.base 中的类), 46  
 WebViewBase (qt4c.webview.chromewebview 中的类), 37  
 WebViewBase (qt4c.webview.iewebview 中的类), 41  
 Width (qt4c.uiacontrols.Control 属性), 69  
 Width (qt4c.util.Rectangle 属性), 72  
 Width (qt4c.wincontrols.Control 属性), 78  
 WIN (qt4c.qpath.EnumUIType 属性), 65  
 Win7ExploreFileFolder (qt4c.filedialog 中的类), 58

Window (qt4c.uiacontrols.UIAWindows 属性), 70  
 Window (qt4c.wincontrols 中的类), 80  
 work\_thread() (qt4c.webview.chromewebview.chromedriver.WebkitD 方法), 37  
 write() (qt4c.util.ProcessMem 方法), 73

## X

X (qt4c.util.Point 属性), 72

## Y

Y (qt4c.util.Point 属性), 72