npyscreen Document zh_CN

发布 *2*

Nicholas Cole [中文 William Mei, Never-way]

2021 年 07 月 04 日

Contents

目录:

CHAPTER 1

npyscreen 简介

‘甩开其他维度的脏活，专心写好界面’

1.1 项目目标

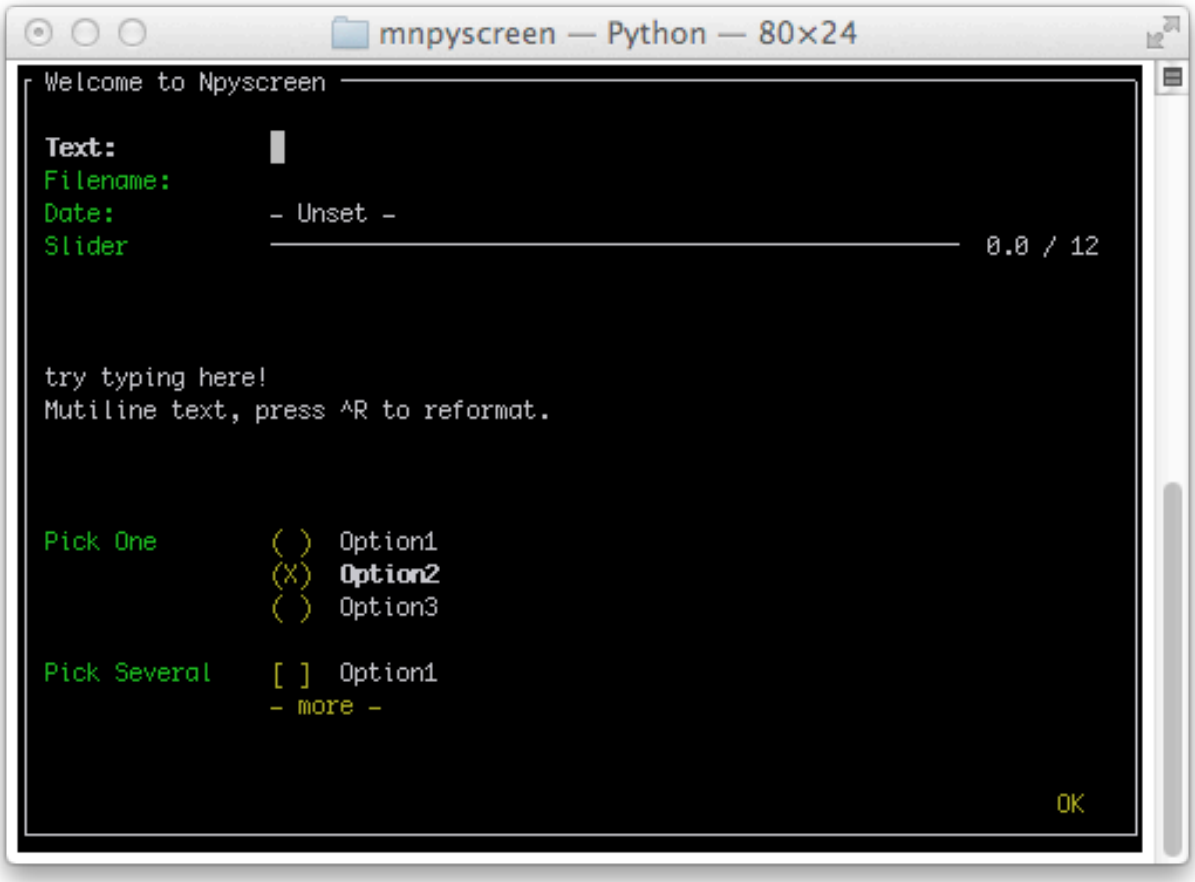
npyscreen 是一个用于编写命令行终端和控制台程序的 python 的部件库和应用程序框架. 它构建于标准库中的 ncurses 库之上.

如果询问用户给出信息可以变得简单一点那该有多好啊？简单的就像这样:

```
MyForm = Form()

usrn_box = MyForm.add_widget(TitleText, name="Your name:")
internet = MyForm.add_widget(TitleText, name="Your favourite internet page:")

MyForm.edit()

# usrn_box.value and internet.value now hold the user's answers.
```

要是你也这么想，这个软件库很适合你.

1.2 代码示例



这里是一个简单的, 单屏幕的应用程序的例子. 更多其他的高级应用将会用到 NPSAppManaged 框架:

```
#!/usr/bin/env python
# encoding: utf-8

import npyscreen
class TestApp(npyscreen.NPSApp):
    def main(self):
        # 这几行代码会创建一个窗口并放上控件.
        # 一个挺复杂的窗口只花了 8 行代码 - 一行一个控件.
        F  = npyscreen.Form(name = "Welcome to Npyscreen",)
        t  = F.add(npyscreen.TitleText, name = "Text:",)
        fn = F.add(npyscreen.TitleFilename, name = "Filename:")
        fn2 = F.add(npyscreen.TitleFilenameCombo, name="Filename2:")
        dt = F.add(npyscreen.TitleDateCombo, name = "Date:")
```

(下页继续)

(续上页)

```
        s  = F.add(npyscreen.TitleSlider, out_of=12, name = "Slider")
        ml = F.add(npyscreen.MultiLineEdit,
               value = """try typing here!\nMutiline text, press ^R to reformat.\n""",
               max_height=5, rely=9)
        ms = F.add(npyscreen.TitleSelectOne, max_height=4, value = [1,], name="Pick One",
                values = ["Option1","Option2","Option3"], scroll_exit=True)
        ms2= F.add(npyscreen.TitleMultiSelect, max_height =-2, value = [1,], name="Pick␣
↪Several",
                values = ["Option1","Option2","Option3"], scroll_exit=True)

        # This lets the user interact with the Form.
        F.edit()

        print(ms.get_selected_objects())

if __name__ == "__main__":
    App = TestApp()
    App.run()
```

1.3 优势

这个框架应该足够强大到能用来创建所有从快捷, 简便的小程序, 到复杂的多屏幕的应用程序. 它被设计成可快速完成简单任务, 并大幅度减轻编写大型程序的负担.

它拥有种类非常丰富的默认控件 - 从简单的文本字段到更复杂的树和网格视图, 应有尽有.

这个库的关注点一直是提供一个快速的开发控制台程序方法. 通常给屏幕增加个控件就只需一行.

1.4 不足

在 2.0pre88 版本中引进了窗体随终端大小自动调整的能力. 而之前的版本都一直假定使用固定大小的终端.

1.5 兼容性

当前版本是在 python3 下完成的, 但是这些代码也兼容较新的 python2. 一些涉及 Unicode 的特性在 python3 下会运行的更好.

它被设计成了仅使用 python 的标准库即可运行, 只需可运行的 python (2.6 或更高) 和 curses 库被安装即可. Npyscreen 也因此可以运行在几乎所有的通用平台, 甚至在 window 中的 Cygwin 环境下. Windows 下的

另一个选择是从 http://www.lfd.uci.edu/~gohlke/pythonlibs/#curses 直接使用 curses 的 Python 库.

1.6 Python 3.4.0

在 python3.4.0 的 curses 模块中有一个灾难性的 bug: http://bugs.python.org/issue21088

该 bug 在 python3.4.1 中被修复, 而直到 3.4.1 发布出来也没有人提醒我这件事, 我不打算在 npyscreen 里发布一个的解决方案, 因为我觉得坚守 python3.4.0 的人数应该很小. 如果这给你带来了问题, 请与我联系.

1.7 Unicode

从 2.0pre47 版本开始所有的文本控件在兼容 utf-8 的终端上应该都支持 utf-8 字符显示和输入了. 这也解决了该库一个长期存在的限制, 并且也让其适合在针对非英语用户的项目中使用了.

自 2.0pre48 版本开始, 该库即计划在所有控件的 Unicode 处理上变得更加健壮. 目前系统的还有一些地方在 utf-8/Unicode 的支持上仍需要进一步的努力. 如果你碰到了的话请向我们发送 bug 报告文档.

1.8 类似的项目

你可能还会喜欢看一下 http://excess.org/urwid/

与 npyscreen 相比, urwid(二胡) 更像一个传统的事件驱动的 GUI 库, 主要针对其他的显示设备与光标.

CHAPTER 2

创建 npyscreen 应用程序

2.1 对象概览

Npyscreen 应用程序是由 3 种主要类型的对象构建出来的.

Form Objects 窗口 窗体对象 (通常是整个终端的大小, 但有时大一些或者 -用于菜单之类的时候会小一些) 可以提供出一个区域容纳其他控件. 它们可能提供一些额外功能, 比如菜单的控制系统, 或是在当用户选择”ok”按钮时启动的协程. 它们可能还会定义一些在用户按键间隙, 或是用户在窗口中移动时要进行的操作.

Widget Objects 控件 这些是窗体上独立的控制部件, -文本框, 标签, 滚动条等等.

Application Objects 应用 这些对象提供一种更方便的管理你的应用程序运行的方式. 虽然有时也能不用应用对象来写一些简单的应用, 但不建议这样. 讲道理, 应用对象在多屏幕应用的管理上更不容易出错 (在那些随时让应用崩溃的 bug 上). 另外, 使用这些对象让你能利用 npyscreen 开发好的高级特性.

2.2 立取可用的程序结构

大多数新应用程序看起来都像下面这样:

```
import npyscreen


# 这里应用充当了 curses 初始化封装器的角色
# 同时也管理着应用的实际形态.
```

(下页继续)

(续上页)

```
class MyTestApp(npyscreen.NPSAppManaged):
    def onStart(self):
        self.registerForm("MAIN", MainForm())

# 这个窗口类定义了展示给用户的显示内容.

class MainForm(npyscreen.Form):
    def create(self):
        self.add(npyscreen.TitleText, name = "Text:", value= "Hellow World!" )

    def afterEditing(self):
        self.parentApp.setNextForm(None)

if __name__ == '__main__':
    TA = MyTestApp()
    TA.run()
```

2.3 应用程序结构细节 (教程)

第一次使用的用户可能会对上面的代码比较困惑. 要是这样, 下面的教程就来详细解释一个 npyscreen 程序的结构. 哪怕你很不了解底层的 curses 系统, 你也应该可以跟上我们的节奏.

2.3.1 窗体, 控件和应用

使用封装器

切换到和切出 curses 环境是个非常枯燥的任务. python 的 curses 模块提供了一个封装器来完成这些. 这在 npyscreen 中以 wrapper_basic 向外暴露. 一个很简单的应用程序的基本框架看起来应该是这样的:

```
import npyscreen

def myFunction(*args):
    pass

if __name__ == '__main__':
    npyscreen.wrapper_basic(myFunction)
    print( "Blink and you missed it!" )
```

没啥复杂的. curses 环境启动并且啥也没干就退出了. 但这只是一个开始.

注意, npyscreen 还提供其他封装器来完成一些稍有不同的东西.

使用窗体

现在我们来放点东西到屏幕上. 为此我们需要一个 *Form* 实例:

```
F = npyscreen.Form(name='My Test Application')
```

这应该就够了, 让我们把他放到封装器里面:

```
import npyscreen

def myFunction(*args):
    F = npyscreen.Form(name='My Test Application')

if __name__ == '__main__':
    npyscreen.wrapper_basic(myFunction)
    print( "Blink and you missed it!" )
```

这看起来好像还是啥也没干 – 因为我们实际还没有把窗体显示出来. *F.display()* 可以把它放到屏幕上, 但是我们实际上是希望用户可以玩一下, 所以让我们用 F.edit() 换掉它:

```
import npyscreen

def myFunction(*args):
    F = npyscreen.Form(name='My Test Application')
    F.edit()

if __name__ == '__main__':
    npyscreen.wrapper_basic(myFunction)
    print( "Blink and you missed it!" )
```

但还是没运行, 因为当你想要尝试编辑窗体的时候 npyscreen 会发现没有控件可编辑. 我们把它改正过来.

添加第一个控件

我们把一个带标题的文本框放到里面. 我们用下面的代码来做到这点:

```
F.add(npyscreen.TitleText, name="First Widget")
```

The full code is:

```
import npyscreen

def myFunction(*args):
    F = npyscreen.Form(name='My Test Application')
    F.add(npyscreen.TitleText, name="First Widget")
    F.edit()

if __name__ == '__main__':
    npyscreen.wrapper_basic(myFunction)
    print( "Blink and you missed it!" )
```

好多了! 这样我们就有了一个有应用样子的东西了. 加上 3 点小调整我们就可以把关闭显示的信息改成用户输的随便什么内容:

```
import npyscreen

def myFunction(*args):
    F = npyscreen.Form(name='My Test Application')
    myFW = F.add(npyscreen.TitleText, name="First Widget")    # <------- Change 1
    F.edit()
    return myFW.value    # <------- Change 2

if __name__ == '__main__':
    print( npyscreen.wrapper_basic(myFunction) ) # <---- and change 3
```

让我们更加面向对象一点

我们现在用的这个方法在简单程序上还可以. 一旦我们开始在窗体上创建大量控件, 还是把那些代码收进对象里面更好. 不再过程化的用基础的 Form() 类, 让我们创建一个自己的窗口类. 我们会重写 Form 类的 *create()* 方法, 只要窗口被创建都会调用它:

```
class myEmployeeForm(npyscreen.Form):
    def create(self):
        super(myEmployeeForm, self).create()  # This line is not strictly necessary: the␣
↪API promises that the create method does nothing by default.
                                              # I've ommitted it from later example code.
        self.myName        = self.add(npyscreen.TitleText, name='Name')
        self.myDepartment  = self.add(npyscreen.TitleText, name='Department')
        self.myDate        = self.add(npyscreen.TitleDateCombo, name='Date Employed')
```

我们可以用前面的封装器的代码来利用这个特性:

```
import npyscreen

class myEmployeeForm(npyscreen.Form):
    def create(self):
        self.myName        = self.add(npyscreen.TitleText, name='Name')
        self.myDepartment  = self.add(npyscreen.TitleText, name='Department')
        self.myDate        = self.add(npyscreen.TitleDateCombo, name='Date Employed')

def myFunction(*args):
    F = myEmployeeForm(name = "New Employee")
    F.edit()
    return "Created record for " + F.myName.value

if __name__ == '__main__':
    print( npyscreen.wrapper_basic(myFunction) )
```

提供选项

实际上, 我们可能就是不太想要任何旧部门名称输进来 - 我们希望给出一列选项. 让我们来用 TitleSelectOne 控件. 这是一个多行控件, 我们需要注意, 让它只占用屏幕的几行就好 (如果让它自己定, 它会占用屏幕上剩余的所有空间):

```
self.myDepartment = self.add(npyscreen.TitleSelectOne, max_height=3,
                             name='Department',
                             values = ['Department 1', 'Department 2', 'Department 3
↪'],
                             scroll_exit = True  # 让用户移通过按下下方向键而不是 tab 来
移出控件.
                                                 # 可以试试看看它们的不同.
                             )
```

Putting that in context:

```
import npyscreen

class myEmployeeForm(npyscreen.Form):
    def create(self):
        self.myName        = self.add(npyscreen.TitleText, name='Name')
        self.myDepartment = self.add(npyscreen.TitleSelectOne, scroll_exit=True, max_
↪height=3, name='Department', values = ['Department 1', 'Department 2', 'Department 3'])
```

(下页继续)

(续上页)

```
        self.myDate        = self.add(npyscreen.TitleDateCombo, name='Date Employed')

def myFunction(*args):
    F = myEmployeeForm(name = "New Employee")
    F.edit()
    return "Created record for " + F.myName.value

if __name__ == '__main__':
    print( npyscreen.wrapper_basic(myFunction) )
```

更彻底面相对象一点

到现在我们都做得还不错, 但还是比较糙. 我们还是手动调用 F.edit() 方法, 这对单窗体的应用还过得去, 但是以后有递归深度的时候, 一不小心就会有问题. 这也让一些这个库本身的更巧妙的特性无法操作了. 更好的办法是使用 *NPSAppManaged* 类来管理你的应用程序.

我们还是放弃这个支持我们这么久的旧框架, 然后以另一个基础开始我们的应用程序吧:

```
import npyscreen

class MyApplication(npyscreen.NPSAppManaged):
    pass

if __name__ == '__main__':
    TestApp = MyApplication().run()
    print( "All objects, baby." )
```

这样其实会异常退出, 因为你没有一个‘MAIN’窗口, 这是 NPSAppManaged 程序的起始点.

我们把它改过来. 我们会用一下前面的窗口类:

```
import npyscreen

class myEmployeeForm(npyscreen.Form):
    def create(self):
        self.myName        = self.add(npyscreen.TitleText, name='Name')
        self.myDepartment = self.add(npyscreen.TitleSelectOne, scroll_exit=True, max_
↪height=3, name='Department', values = ['Department 1', 'Department 2', 'Department 3'])
        self.myDate        = self.add(npyscreen.TitleDateCombo, name='Date Employed')

class MyApplication(npyscreen.NPSAppManaged):
```

(下页继续)

(续上页)

```
    def onStart(self):
        self.addForm('MAIN', myEmployeeForm, name='New Employee')

if __name__ == '__main__':
    TestApp = MyApplication().run()
    print( "All objects, baby." )
```

如果你运行上面的代码, 你可能觉得自己有点挫败, 因为这个程序会一直显示那个要你编辑的窗口, 然后你不得不按下 “^C” (Control C) 来退出.

这是因为 NPSAppManaged 类一直会显示任何以它的 NEXT_ACTIVE_FORM 属性 (这个例子里, 默认是 – ‘MAIN’) 命名的窗口. 旧版的教程会建议直接去设定它, 但是你得用 setNextForm(formid) 方法.

让我们改一下 myEmployeeForm 来告诉它, 在 NPSAppManaged 上下文中运行之后, 它应该通知它的父对象 NPSAppManaged 停止显示窗口. 我们通过创建一个叫做 *afterEditing* 的特殊方法来实现:

```
class myEmployeeForm(npyscreen.Form):
    def afterEditing(self):
        self.parentApp.setNextForm(None)

    def create(self):
        self.myName        = self.add(npyscreen.TitleText, name='Name')
        self.myDepartment  = self.add(npyscreen.TitleSelectOne, scroll_exit=True, max_
↪height=3, name='Department', values = ['Department 1', 'Department 2', 'Department 3'])
        self.myDate        = self.add(npyscreen.TitleDateCombo, name='Date Employed')
```

如果喜欢的话, 我们还可以通过在 MyApplication 类中定义一个特殊的 *onInMainLoop* 方法来实现同样的结果 – 这个方法会在每个窗口被编辑完之后被调用.

我们的代码现在看起来是这样的:

```
import npyscreen

class myEmployeeForm(npyscreen.Form):
    def afterEditing(self):
        self.parentApp.setNextForm(None)

    def create(self):
        self.myName        = self.add(npyscreen.TitleText, name='Name')
        self.myDepartment = self.add(npyscreen.TitleSelectOne, scroll_exit=True, max_
↪height=3, name='Department', values = ['Department 1', 'Department 2', 'Department 3'])
        self.myDate        = self.add(npyscreen.TitleDateCombo, name='Date Employed')
```

(下页继续)

(续上页)

```
class MyApplication(npyscreen.NPSAppManaged):
    def onStart(self):
        self.addForm('MAIN', myEmployeeForm, name='New Employee')
        # A real application might define more forms here.......

if __name__ == '__main__':
    TestApp = MyApplication().run()
```

处理方式的选择

上面最后一个例子, 对于只是写个很简单的小应用可能有点杀鸡焉用牛刀了. 但是这样能提供一个健壮的多的可以构建更大型的应用程序框架, 比我们在教程一开始用的那个也就只多写了几行而已. 如果你要显示不止一个屏幕, 或者一直运行一个应用的话, 这就是你该用的处理方法.

CHAPTER 3

应用对象

NPSAppManaged 提供一个框架开始和结束应用程序, 以一种不会有递归深度问题的方式完成对你创建的各种窗口的显示管理.

除非你有什么特别好的理由要用其他方式, 否则 *NPSAppManaged* 应该就是管理你的应用的最好方法.

跟简朴的 NPSApp 类不同, 你不用自己写主循环 - *NPSAppManaged* 会管理好你的应用的每个窗口的显示. 设定好你的窗口对象, 然后就只要调用你的 NPSAppManaged 实例的 *.run()* 方法就可以了.

3.1 让 NPSAppManaged 管理你的窗口

有 3 种方法来用 NPSAppManaged 实例注册一个窗口对象:

`NPSAppManaged.addForm(`**id**, **FormClass**, *...*`)`

在种版本的写法会先创建一个新的窗口, 然后用 NPSAppManaged 实例来注册. 它从窗口对象返回一个 weakref.proxy [弱引用的对象代理]. 其 *.id* 需要是一个可以唯一标识该窗口的字符串. *FormClass* 应该是要创建的窗口的类. 任何额外的参数都会别传递到该窗口的构造函数中 [用来指定窗口的创建规格]. 如果你没有单独存放窗口的引用的地方, 就用这种写法.

`NPSAppManaged.addFormClass(`**id**, **FormClass* ...*`)`

这种写法注册一个窗口类而不是一个实例. 每次被编辑完都会有一个新实例被创建. 额外的参数在每次创建的时被传递到窗口的构造函数中.

`NPSAppManaged.registerForm(`*id*, *fm*`)`

id 需要是一个可以唯一标识该窗口的字符串. *fm* 需要是一个窗口对象. 要注意, 跟.addForm 的写法比起来 - 这种写法只在 NPSAppManaged 中存放一个 weakref.proxy .

所有用 NPSAppManaged 实例注册的窗口都可以用 *self.parentApp* 来访问到对应用对象的控制.

若因某些原因需要删除一个窗口, 你可以用 *.removeForm(*id*)* 方法来去做.

3.2 运行 NPSApplicationManaged 应用

run()
开始一个 NPSAppManaged 应用的主循环. 该方法会激活默认的窗口, 它应该被指定用 “MAIN” 来作为 id .

NPSAppManaged.**STARTING_FORM**
如果你出于什么原因要改默认窗口的名称, 你可以在这改.

一旦程序开始运行, 下面的方法会控制哪个窗口展示给用户.

NPSAppManaged.**setNextForm**(*formid*)
设定在当前窗口退出后, 要显示的窗口.

NPSAppManaged.**setNextFormPrevious**()
设定在当前窗口退出到历史中的上一个窗口时, 要显示的窗口.

NPSAppManaged.**switchForm**(*formid*)
立即切换到指定的窗口, 绕过一切当前窗口的退出逻辑.

NPSAppManaged.**switchFormPrevious**()
立即切换到历史记录中之前工作的窗口.

3.2.1 细节说明

一旦所有的窗口都准备好并注册到 NPSAppManaged 实例, 你就该调用.run() 方法了.

这个方法会激活默认的窗口, 它应该以 “MAIN” 为 id. 你可以通过修改类或者实例的 *.STARTING_FORM* 变量来改变这个默认 id.

这之后, 下一个被显示的窗口就是被实例的 *NEXT_ACTIVE_FORM* 变量所指定的. 不管什么时候当一个窗口的编辑循环退出了, 这个变量所指定的新窗口都会被激活. 如果此时 *NEXT_ACTIVE_FORM* 是 None, 主循环会退出. *NEXT_ACTIVE_FORM* 应该通过调用应用的 *setNextForm(formid)* 方法来设定. 这份文档在过去曾建议直接设置该属性. 虽然当前还没有立即弃用该属性的计划, 不过还是要避免直接去设定它.

有 3 种建议使用的在窗口中控制 *NEXT_ACTIVE_FORM* 的机制.

1. 所有用 NPSAppManaged 注册的窗口里面那些 *没有* *.activate()* 特殊方法的, 其 *.afterEditing* 方法会被调用, 如果后者能有的话. 判定哪个应该是 *NEXT_ACTIVE_FORM* 逻辑应该放到这里. *NEXT_ACTIVE_FORM* 应该是通过调用应用的 *setNextForm(formid)* 方法来配置的. 如果你是希望用户来选择 ok/确认或者 cancel/取消按钮的话, 这是你切换屏幕的首选方法.

2. 应用的 *switchForm(formid)* 方法会造成应用立即停止编辑当前窗口, 并切换到指定的窗口. 依窗口的类型而定, 它们相关联的退出逻辑可能也会被绕过.

3. 用 NPSAppManaged 注册的窗口可能被赋予一个 *.activate()* 方法, NPSAppManaged 会调用它来代替较常用的 *.edit()* 方法. 这可以包含进一步的逻辑. 这 **不是**最优方法, 但是却能允许有更大的灵活性. 要注意这种情况下, 除非你指明要调用, 否则通常的.edit() 方法不会被调用. 举个栗子, 一个.activate() 方法看着可能是这样的:

```
def activate(self):
    self.edit()
    self.parentApp.setNextForm(None)
```

这就会导致主循环在窗口结束后会退出.

3.3 NPSAppManaged 提供的附加服务

下面这些方法可以在 NPSAppManaged 子类里有效重写. 默认他们什么也不做.

NPSAppManaged.onInMainLoop()
在程序运行时, 在每个屏幕 [切换] 之间被调用. 但不在第一个屏幕之前被调用.

NPSAppManaged.onStart()
重写该方法可执行一切初始化任务. 如果你愿意, 你可以在这里布置应用的窗口.

NPSAppManaged.onCleanExit()
重写该方法以便在程序无错退出时, 执行一切清理任务.

NPSAppManaged.keypress_timeout_default
如果配置了该项, 新窗口在创建时会将 keypress_timeout 配置成它, 只要它们知道自己所属的应用程序 - 也就是说, 它们在创建时被传递了 *parentApp=* 属性. 如果你用了 NPSAppManaged, 这些都是自动的.

NPSAppManaged.while_waiting()
应用可能都还有一个 *while_waiting* 方法. 你可以自由地定义和重写这部分, 它会在应用程序等待用户输入的时候被调用 (参考其他窗口里的 while_waiting 方法).

NPSAppManaged._internal_while_waiting()
该方法用于 npyscreen 内部使用.

NPSAppManaged.switchForm(*formid***)**
立即停止编辑当前窗口并切换到指定的窗口.

NPSAppManaged.switchFormPrevious()
立即切换到历史中的前一个窗口.

NPSAppManaged.resetHistory()
遗忘之前访问过的窗口 [重置历史记录].

NPSAppManaged.getHistory()
获取之前访问过的窗口的列表.

3.4 这个类管理的窗口的方法和属性

为 NPSAppManaged 所调用的窗口可提供的方法

`Form.beforeEditing()`
在窗口的编辑循环被调用之前调用.

`Form.afterEditing()`
窗口退出时被调用.

`Form.activate()`
该方法的存在会完全覆盖已有的.beforeEditing .edit 和 afterEditing 方法.

CHAPTER 4

其他应用类

`class NPSApp`

要用 NPSApp 要将它子类化, 然后给出你自己的 *.main()* 定义. 当你准备好调用 *.run()* 运行应用程序的时候, 你的主循环就会被执行.

虽然它提供了尽可能大的灵活性, 但是 NPSApp 几乎在其他各方面都次于 NPSAppManaged. 别在新项目中使用它, 要把它当成只在内部使用的基类.

CHAPTER 5

窗口对象

窗口对象就是屏幕上一个容纳控件的区域. 窗口对象可以控制用户当前编辑哪个控件, 并且还可能提供附加功能, 比如弹出菜单和特定按键被按下应该产生的动作.

5.1 创建窗口

class Form(*name=None*, *lines=0*, *columns=0*, *minimum_lines=24*, *minimum_columns=80*)

窗口具有以下类属性:

```
DEFAULT_LINES      = 0
DEFAULT_COLUMNS    = 0
SHOW_ATX           = 0
SHOW_ATY           = 0
```

使用这些默认值会创建一个显示在左上角并且填满整个屏幕的窗口. 控制窗口大小的细节参见传递进构造函数的参数.

以下参数能被传进窗口的构造函数里:

name= 命名该窗口. 对于一些控件, 这样会显示一个标题.

lines=0, columns=0, minimum_lines=24, minimum_columns=80

> 要调整窗口的大小, 要么给出其尺寸绝对值 (使用 *lines=* 和 *columns=*), 要么给出其最小尺寸值 (*minimum_lines=* 与 *minimum_columns=*). 默认的最小值 (24x80) 给出了在终端上的标准大小. 如果你设计好让窗口适配到这个大小以内, 那么他们应该在所有的系统的界面上不滚动窗口

的情况下也全可见. 注意, 你可以在一个方向使用绝对值大小, 另一个方向上使用最小值大小, 要是你想要的话.

标准的构造函数都会调用 *.create()* 方法, 你应该对它重写来给窗口创建控件. 请看下面.

5.2 把控件放到窗口

要把控件放到窗口上, 要用这些方法:

Form.add(*WidgetClass*, *...*)

WidgetClass 必须得是一个类, 所有的附加参数都会被传递到控件自己的构造函数里. 控件的引用会被返回.

控件的位置和大小由控件的构造函数控制. 不过, 有一些指示是窗口类给出的. 如果你不重写控件的位置, 它会根据窗口实例的 *.nextrelx* 和 *nextrely* 属性值来归置. 这里的 *.nextrely* 属性是随每次的控件放置而自增长的. 你也可以自行把它加大, 就像下面这样:

```
self.nextrely += 1
```

这会在前一个被被放置的控件跟下一个之间留下一个间隔.

Form.nextrely

下一控件创建时, 在 y 轴的位置. 每当控件被加进窗口, 标准的窗口都会把它设成上一被创建控件的下一行.

nextrelx

下一控件被创时, 在 x 轴的位置.

5.3 标准窗口的其他特性

Form.create()

该方法由窗口的构造函数调用. 默认它什么也不做 - 它是准备让你在子类中重写的, 不过它是对窗口上所有的控件进行配置的最佳位置. 所以, 等着这个方法装满 *self.add(…)* 方法的调用吧!

Form.while_editing()

这个方法在用户于控件之间移动时被调用. 它也是打算让你在子类中重写的, 做一些像是让一个控件按另一个的值改变之类的事情.

Form.adjust_widgets()

用这个方法的时候要很小心. 窗口被编辑时, 每次按键时它都被调用, 并且不保证它不会被调用的更频繁. 默认它没有动作, 并且是打算让你来重写的. 鉴于它被如此频繁地调用, 这里大意的话就要拖慢你的整个应用了.

比如, 对重绘整个窗口 (这是个缓慢的操作) 就要尽量保守一点 - 要确保在代码中实际验证是否每个重绘都必要, 试着只重绘那些真的需要被调整的控件, 而不是重绘整个屏幕.

如果窗口的 parentApp (父级的应用对象) 也有叫 *adjust_widgets* 的方法, 它也会被调用.

Form.while_waiting()

如果你想在等待用户按下某个键时执行动作, 你可以定义一个 *while_waiting* 方法. 同时你还要设置 *keypress_timeout* 属性, 这是个以毫秒 (ms) 为单位的值. 每当开始等待输入, 如果时间超过了 *keypress_timeout* 的给定时间, *while_waiting* 就会被调用. 注意, npyscreen 没有采取任何流程来保证 *while_waiting()* 精确按照规律的时间间隔被调用, 实际要是用户持续按着键, 它可能永远也不会被调用到.

如果窗口的 parentApp 有叫 *while_waiting* 的方法, 它也会被调用.

一个 *keypress_timeout* 为 10 的值, 就意味着 *while_waiting* 方法每秒都会被调用到, 假设用户没有其它动作.

功能完整的例子参见代码包含的 Example-waiting.py 样例文件.

Form.keypress_timeout

参考上面的 *while_waiting* 方法.

Form.set_value(*value*)

将 *value* 存到 *Form* 对象的 *.value* 属性中, 然后调用所有控件的 *when_parent_changes_value* 方法, 如果它们有的话.

Form.value

所有的窗口类可能都有 *set_value(value)* 方法. 这会设定 *value* 属性的值并调用内部窗口所含的每一控件的 *when_parent_changes_value* 方法.

5.4 显示并编辑窗口

Form.display()

重绘窗口及其每个控件.

Form.DISPLAY()

重绘窗口, 并额外确认显示器被重置. 这是一个缓慢的操作, 要尽量避免调用它. 你有些时候可能需要使用到它, 比如外部进程破坏了终端.

Form.edit()

允许用户交互式地编辑每个控件的值. 如果正确使用 *NPSAppManaged* 类, 你应该不需要用到这个方法. 你应该尽可能避免调用该方法, 不过有时候可能用到, 要是写简单应用用不上 NPSAppManaged 类的话. 直接调用该方法更类似于在 GUI 应用中创建一个模式对话框 [模态框, 覆盖在父窗体上的子窗体]. 尽可能地去把这个方法当成一个内部的 API 调用.

5.4.1 窗口退出的情况

窗口可能因为很多原因需要退出其编辑模式. 在 NPSAppManaged 应用中, 控制它的应用也可能导致窗口退出.

自行把 *.editing* 属性设定为 False, 其实也会导致窗口退出.

5.5 标准窗口类

class Form

基本的窗口类. 在编辑窗口的时候, 用户可以通过选择右下角的 OK 按钮来退出.

默认情况下, 窗口会填满终端. 弹出式窗口只是具有更小的默认大小的窗口.

class Popup

弹出式窗口只是个默认大小更小的窗口.

class ActionForm

动作窗口 ActionForm 会创建 OK/确认和 cancel/取消按键. 选择任一个都会退出窗口. 当窗口退出时 (假设是用户选择了这之中的一个按钮) *on_ok* 或 *on_cancel* 方法将被调用. 因此, 在子类中可以有效重写其中一个或者两个, 默认它们没有动作.

on_ok()

按下 ok 按钮的时被调用. 设置该方法的 *.editing* 属性为 True 会中止窗口的编辑.

on_cancel()

当按下 cancel 按钮的时候被调用. 设置该方法的 *.editing* 属性为 True 会中止窗口的编辑.

class ActionFormV2

在 4.3.0 版本中被新加入. 这个版本的动作窗口 ActionForm 的动作跟前面的 ActiveForm 类似, 但是代码更清爽. 它应该更容易子类化. 最终这个版本应该会完全取代 ActionForm.

class ActionFormMinimal

于 4.4.0 版本被新加入. 这个版本的 ActionFormV2 只有一个 OK 按钮. 按用户的要求被添加用于特殊情况.

class ActionPopup

是 ActionForm 小点的版本.

class SplitForm

SplitForm 中间有一个水平线. 其 *get_half_way()* 方法会告诉你它被绘制在哪.

draw_line_at

改属性定义了横穿屏幕的横线所绘制的位置. 它可以通过传递 *draw_line_at=* 到构造函数来设置, 或者根据 *get_half_way* 方法的返回值自动设定.

get_half_way()

返回穿过窗口中间的横线的 y 轴坐标. 实际上在此窗口的子类中, 也没有什么特别原因, 要让 y 轴坐标在实际上位于窗口向下刚好一半的位置, 其实子类可会返回任何方便的值.

MOVE_LINE_ON_RESIZE

这个类属性指定了当窗口调整大小时, 横线的位置是否应该要被移动. 因为横线下面的

所有的控件也都需要被移动 (设想到该窗口的子类中 *resize* 方被重写了的情况, 该值默认被设为 False).

class FormWithMenus

类似于 Form 类, 但是提供了弹出菜单的附加功能.

要添加新菜单到窗口, 请使用 *new_menu(name='')* 方法. 这样会创建菜单并返回其代理. 更多细节参见下面菜单部分.

class ActionFormWithMenus

类似于 ActionForm 类, 但是提供了弹出菜单的附加功能.

要添加新菜单到窗口, 请使用 *new_menu(name='')* 方法. 这样会创建菜单并返回其代理对象. 更多细节参见下面菜单部分.

class ActionFormV2WithMenus

在 4.3.0 版本被新加入. 这个版本的 ActionFormWithMenus 表现的跟上面的 ActionForm 类似, 只是代码更加清爽. 子类操作上应该更容易. 最终, 这个版本应该会完全取代 ActionFormWithMenus.

class FormBaseNew

这种窗口默认没有 *ok* 或 *cancel* 按钮. 附加方法 *pre_edit_loop* 和 *post_edit_loop* 会在该窗口被编辑之前与之后被调用. 默认版本中没有动作. 该类准备用作更复杂用户界面的基础.

pre_edit_loop()

窗口开始被编辑之前被调用.

post_edit_loop()

在编辑循环退出后被调用.

class FormBaseNewWithMenus

开启菜单的 FormBaseNew.

5.6 类-Mutt 窗口

class FormMutt

受类似 *mutt* [一个字符界面邮件客户端] 或 *irssi* [一个字符界面 IRC 程序] 用户界面的启发, 这种窗口定义了 4 种默认控件:

wStatus1 它位于屏幕的顶部. 你可以通过调整窗口的 *STATUS_WIDGET_CLASS* 类属性改变要使用的控件类型 (注意, 它在两个状态行里面都要用到).

wStatus2 它占据了屏幕的倒数第二行. 你可以通过调整窗口的 *STATUS_WIDGET_CLASS* 类属性改变要使用的控件类型 (注意, 它在两个状态行里面都要用到).

wMain 它占据 wStatus1 和 wStatus2 之间的区域, 而且是一个多行控件. 你可以改变出现在这里的控件的类型, 先子类化 *FormMutt* , 然后改变 *MAIN_WIDGET_CLASS* 类属性即可.

wCommand 这个区域占据屏幕的最后一行. 你可以通过改变 *COMMAND_WIDGET_CLASS* 类属性来改变要用的控件类型.

默认, wStatus1 和 wStatus2 都把 *editable* 属性设为了 False.

FormMuttActive, FormMuttActiveWithMenus, FormMuttActiveTraditional, FormMuttActiveTraditionalW

这些类都是用来简化创建更复杂应用的. 每个类都用了额外的 *NPSFilteredDataBase*, *ActionControllerSimple*, *TextCommandBox*, *TextCommandBoxTraditional* 类.

常见的 *nix 风格终端应用 (像 mutt 和 irssi 等用到的) 都有个带中央显示的时间列表或网格, 一个底部的命令行, 还有一些状态信息行.

这些类让配置一个类似的窗口变得容易. *FormMuttActive* 和 *FormMuttActiveTraditional* 类的不同点是, 在后者中, 用户最终实质编辑的唯一控件, 是屏幕底部的命令行控件. 不过, 如果这些控件没有在编辑命令行, 按键动作会被传递到在显示中央的多行控件里, 以允许用户来回滚动并选择屏幕上的条目.

实际上什么要被显示到屏幕上, 是由 *ActionControllerSimple* 类控制的, 以它为基础, 数据不是被任意独立控件, 而是由 *NPSFilteredDatabase* 类来存储的.

更多信息参见该文档后续的编写类-Mutt 应用程序部分.

5.7 多页面窗口

class FormMultiPage(*new in version 2.0pre63*)

这个 实验性的类添加了多页面窗口的支持. 默认, 在一个页面上向下滚动超出上最后一个控件, 就会移动到下一个页面, 而从第一个控件继续向上移动就会回到上一页面.

默认的该类会你把所在的页面显示在屏幕的右下角, 如果 *display_pages* 属性为 True 且页面多于一个的话. 你也可以把 *display_pages=False* 传递到构造函数. 用来进行显示的颜色存在 *pages_label_color* 属性中. 默认它的值是 'NORMAL' . 其他好用的值有 'STANDOUT' , 'CONTROL' 或 'LABEL' . 同样, 你也能把他们传进构造函数.

要注意这个类是實驗性的. 其 API 还在审查中, 并且可能会在以后的版本中有所调整. 它计划用于那些可能不得不动态地去创建窗口的应用程序上, 它们可能需要创建比屏幕还要大的单个窗口 (比如一个要显示服务器所指定的 xmpp 表的 Jabber 客户端). 它 不是用来显示任意大的项目列表的. 要打算那样的话, multiline 类的控件可能会高效得多.

有 3 个新的方法被加到该窗口对象中:

FormMultiPage.**add_page**()

用于窗口的创建时期. 这会添加一个新的页面, 并且重置新控件添加点的位置. 新添页面的索引页数将被返回.

FormMultiPage.**switch_page**(**index**)

该方法将活跃页面改为由 *index* 指定的页.

FormMultiPage.**add_widget_intelligent**(**args*, ***keywords*)

该方法会添加一个控件到窗口. 如果当前页面没有足够的空间, 它会尝试创建一个新页面然后再把控件加到那儿. 要注意, 如果用户指定了哪怕是在新页面是也会防止控件被显示的选项, 这个方法可能依然会抛出异常.

class FormMultPageAction(*new in version 2.0pre64*)

这是个 实验版的 FormMultiPage 类, 添加了 ActionForm 的 on_ok 和 on_cancel 方法, 并且窗口的最后一页页自动创建 cancel 和 ok 按钮.

class FormMultiPageWithMenus

开启菜单版的 MultiPage.

class FormMultiPageActionWithMenus

开启菜单版的 MultiPageAction.

5.8 菜单

一些窗口支持使用弹出窗口. 理论上菜单也可以作为独立的控件来使用. 我们选择使用弹出菜单而不是下拉菜单 (实际上受 RiscOS 的菜单系统启发), 因其更适合键盘环境使用, 并有效利用可用的屏幕空间, 也更容易部署到各种大小的终端上.

默认, 支持的窗口都会显示一个带菜单系统的广告页给用户, 及一个菜单列表的快捷键. 如果窗口有多个菜单, 那么一个将其全部列出的‘根’菜单会被显示出来.

菜单通常是调用 (支持的) 窗口的 *new_menu* 方法创建的. 2.0pre82 版本添加了 *shortcut=None* 参数到该方法. 在窗口显示的菜单列表中, 这个快捷键也会被显示. 在一个菜单被创建之后, 该对象的以下方法会很有用:

NewMenu.**addItem**(*text=''*, *onSelect=function*, *shortcut=None*, *arguments=None*, *keywords=None*)

text 应该是菜单上显示的字符串. *onSelect* 应该是菜单项被用户选中后需要调用的函数. 这是仅有的几个 npyscreen 中容易创建循环引用的地方之一 - 你可能希望只传递一个代理进来. 我一直在尽力保护你远离循环引用, 而这也只其中一次, 很多时候我也无法猜测你的应用程序结构. 2.0pre82 版本增加了添加快捷键的功能.

从 3.6 版本以后, 菜单项可以使用 参数列表加上或者只用一个关键字字典来指定.

NewMenu.**addItemsFromList**(*item_list*)

该函数的参数应该是一个列表或者元组. 其中的每个元素都应该是创建一个菜单的项参数的元组. 该方法已经 * 废弃 *, 并且可能在以后的版本中被移除或者修改.

NewMenu.**addNewSubmenu**(*name=None*, *shortcut=None*, *preDisplayFunction=None*, *pdfuncArguments=None*, *pdfuncKeywords=None*)

创建一个子菜单 (返回其代理). 这是创建子菜单的最佳方式. 2.0pre82 版本增加了添加键盘快捷键的功能.

从 3.7 版本之后, 你可以在这个菜单显示之前定义一个被调用的函数及参数. 这可能意味着你可以在菜单显示的时刻调整其内容. 应用户要求添加.

NewMenu.**addSubmenu**(*submenu*)

将一个已经存在的菜单添加到菜单中作为子菜单. 综合考虑, addNewSubmenu 通常都是更好的选择.

(在内部, 这个菜单系统被叫做“新”菜单系统 - 它替代了我一直不怎么喜欢的下拉菜单系统.)

5.9 窗口重调大小 (2.0pre88 版本新增)

当窗口的大小被重新调整, 会有一个信号发往屏幕上的当前窗口. 该窗口是否处理它取决于 3 件事.

如果你设置 *npyscreen.DISABLE_RESIZE_SYSTEM* 变量为 True. 窗口将完全不会调整大小.

类属性 *ALLOW_RESIZE* (默认 =True). 如果它被设置为 False, 窗口不会调整自身大小.

类属性 *FIX_MINIMUM_SIZE_WHEN_CREATED* 控制着窗口是否可以变得比创建时更小. 默认它被设为 *False*. 这是因为十多年来,npyscreen 都假设窗口永远不会改变大小, 并且很多程序可能都依赖于窗口大小永远不被调整这一现实. 如果你是在从头开始写新代码, 你可以把这个值设成 True, 只是要确保你测试过结果, 以确定调整窗口大小不会让你的应用程序崩溃.

当窗口被重新调整大小, *resize* 方法会在新的窗口大小被确定后被调用. 窗口都可以重写此方法, 来将控件移动到新的位置, 或修改任何相关让窗口布局更合适的东西.

当你使用 *NPSAppManaged* 系统时, 窗口会在显示之前被自动调整大小.

CHAPTER 6

控件：基本特点

6.1 创建控件

控件是通过表单类中 *add(…)* 方法的第一个参数来创建的。其余参数将传递给控件自身的构造函数。这些控制诸如大小、位置、名称和初始值等

6.2 构造函数参数

name= 你可能需要给每个控件命名（字符串）。在适当的时候，它将被用于该控件的标签

relx=, rely= 控件在表格上的位置是被整型参数 relx 和 rely 控制的。你不需要指定他们，那样的话表单将尽最大可能决定控件放在何处。你可以指定他们中的其中一个如果你这样选择（eg. 你通常不需要指定 relx)。*New in Version 4.3.0*: 如果你给 rely 或 relx 一个负值，控件将被定位在相对于窗口底部或右侧的位置，如果窗体调整了大小，npyscreen 将尽最大可能保持控件的位置

width=, height=, max_width=, max_height= 默认情况下，控件会展开以填充右侧和下方的可用空间，除非这样做没有意义-例如单行文本不需要一行，因此无需另外声明。因此，为了改变控件大小，指定一个不同的 *max_width* 或 *max_height*。最好使用 max_version - 如果空间不足或指定的空间太多，这些将不会报错，但会试图将控件压缩到剩余空间中。

value= 控件的值是用户可以更改的——字符串，日期，选择项，文件名。此处可初始化设置 *.value* 属性

values= 当控件向用户提供列表值中的一个选项，这些在此处指定：这是 *value* 属性的初始化设置。

editable=True 用户是否能够编辑控件（*.editable* 属性的初始化设置）

hidden=False 控件是否可见（*.hidden* 属性的初始化设置）

color='DEFAULT', labelColor='LABEL' 为颜色管理系统提供控件应如何显示的线索

更多详细信息参考 'setting up colors <color_reference>'。

scroll_exit=False, slow_scroll=False, exit_left, exit_right 这些影响了用户和多行控件的交互模式。*scroll_exit* 决定了用户是否能将第一个或最后一个项目移动到前一个或后一个控件。*slow_scroll* 意味着滚动控件每次只能滚动一行，而不是满屏滚动。选项 *exit_left/right* 命令用户是否能够使用向左和向右箭头键退出控件。

6.3 使用和显示控件

所有控件包含以下方法：

display() 重绘控件并且告知 curses 更新屏幕

update(clear=True) 重绘控件，但不告知 curses 更新屏幕（更新所有控件，然后让控件所在的表单一次性告诉 curses 重新绘制屏幕。

大多数控件接收可选参数 *clear=False/True*，该参数会影响在重新绘制之前是否先清空它所占用的区域。

when_parent_changes_value() 在调用父表单的 *set_value(value)* 方法时调用。

when_value_edited() 在编辑控件过程中，当它的值改变时调用，之后按下键盘。你可以通过设置属性 *check_value_change* 为 False 来禁用它。

你可以根据自己的使用需要覆盖它。

when_cursor_moved() 在编辑控件的过程中，当它的光标已经被移动时调用。你可以通过设置属性 *check_cursor_move* 为 False 来禁用这个检查。

你可以根据自己的使用需要覆盖它。

edit() 允许用户与控件的交互作用。当用户离开控件则该方法返回。在多数情况下，你将永远不需要自己调用这个方法，并且在很大程度上，这应该被视为 npyscreen 内部 API 的一部分

set_relyx() 设置控件在表单中的位置。如果 y 或 x 是负值，npyscreen 将尝试相对于表单底部或右侧边缘进行定位。注意，这会忽略表单可能定义的任何边距。(新版本 4.3.0)

safe_to_exit() 这个方法是在用户试图从控件退出之前，通过默认处理器被调用得。如果这被允许，它则返回 True；如果不允许，则返回 False。在允许用户退出之前，你可以覆盖这个方法来执行对字段内容的任何验证。(新版本 4.3.0)

6.4 给控件加标题

许多控件存在两种表单，一种有标签，一种没有。例如，Textbox 和 TitleText。如果该标签特别长（在构建时），该标签可能在自己的线上。附加构造器参数：

use_two_lines= 如果为真或为假，则覆盖控件原本选择的内容

field_width=（对于文本字段）-控件的输入部分应当有多宽?

begin_entry_at=16 控件输入部分应当从哪一行开始?

内部命名的控件实际上是一个文本框 (用于标签) 和任何其他类型的控件是必需的。你可以通过 *label_widget* 和 *entry_widget* 属性访问单独的控件 (如果你需要—实际你不应该)。但是，你可能永远都不需要这样做，因为组合控件的 *value* 和 *values* 属性会按照预期工作。

6.5 创建自己的控件

所有控件应继承自 'Widget'类。

calculate_area_neeeded 这个函数被调用是为了询问控件需要多少行和列（用于最小显示）。你应该返回一个正好包含两个数字的元组。两者中的一个参数返回 0 说明，如果控件可用，应为它提供在显示上的所有剩余空间

如果你正在屏幕中编辑文本，你应当避免直接使用光标，而是使用函数替代

add_line(realy, realx, unicode_string, attributes_list, max_columns, force_ascii=False) 这个函数添加了一行文本显示。'realy'和 'realx'是在表格中的绝对定位。'attributes_list'是一个应被应用于每个字符的属性列表。如果所有这些要求同样的属性，使用 'make_attributes_list'方法来创建一个正确长度的列表。

make_attributes_list(unicode_string, attribute) 一个方便的函数。重新设置提供的 unicode_string 长度的列表，列表中的每个项都包含一个属性副本。

resize() 当控件被调整大小，你可以覆盖这个方法来执行任何必要的操作。（新版本 4.3.0）

CHAPTER 7

控件: 显示文本

Textfield, TitleText 一个单行的文本, 不过长度是任意的 - 是最基本的输入控件. 注意, 一些在更正中的文档版本指的会是‘textbox’[文本框] 控件.

FixedText, TitleFixedText 单行的文本, 但其文本框的编辑功能被移除了.

PasswordEntry, TitlePassword 一个文本框, 但被修改了, 以便刚好 *.value* 的字母不显示出来.

Autocomplete 这是一个带有附加功能的文本框 - 我们的想法是如果用户按下了 TAB 键控件会尝试‘补全’用户正在输入的内容, 适当给出可选择的选项. 其调用的方法是 *auto_complete(inputch)*.

当然, 上下文就是一切. *Autocomplete* 也因此没那么有用, 但其用意是做为你可以子类化的东西. 参见 Filename 和 TitleFilename 这两个类作为例子.

Filename, TitleFilename 一个会尝试‘补全’用户输入的文件名或者路径的文本框.

这是个 *Autocomplete* 控件的示例.

FilenameCombo, TitleFilenameCombo 这是更高级的选择文件的方法. 2.0pre82 版本新添.

MultiLineEdit 该控件可允许用户编辑若干行的文本.

Pager, TitlePager 该 [页面] 控件会显示多行文本, 并允许用户来回滚动, 但不能编辑. 要显示的文本存放在 *.values* 属性中.

7.1 细节

`class Textbox`

`display_value`(*vl*)

控制 *.value* 属性的值如何显示. 由于各自版本的文本控件都要用在其他混合控件里 (比如大多数的多行控件类), 该方法通常会被重写.

`show_brief_message`()

发出蜂鸣并显示一个简短的信息给用户. 通常来说会有更好的方式来做这个, 但这个有时候也会有用, 比如在 Autocomplete 类显示错误的时候.

CHAPTER 8

控件: 选取选项

MultiLine 向用户给出一个选项列表.(该控件可能应该可以有一个更好的名字, 但是我们现在先这样吧)

选项应该以列表形式被存在 *values* 属性中. *value* 属性保存了用户所选项目的索引. 如果你想返回实际选择的值而不是某个索引, 请使用 *get_selected_objects()* 方法.

MultiLine 及其衍生控件类的最重要的特性之一就是, 它们可以很容易的被适配用来让用户选择不同类型的对象. 要这么做, 请重写 *display_value(self, vl)* 方法. 其中 *vl* 参数会是被显示的对象, 函数应该会返回一个能被显示到屏幕上的字符串.

换句话说, 你可以传入一个任意类型对象的列表. 默认, 他们会被用 *str()* 来显示, 但是通过重写 *display_value* 你能用任何你认为合适的方式展示它们.

MultiLine 也会允许用户‘过滤’条目. (绑定的按键 I, L, n, p 默认是过滤器, 清除过虑器, 下一个和上一个). 当前的实现会高亮显示屏幕上匹配的行. 未来的版本可能会隐藏其他的行或者会给出一个选项. 你可以通过重写 filter_value 方法来控制过滤器如何进行操作. 它要接收一个索引作为参数 (用来在.values 列表中找到某一行) 并且匹配则返回 True, 否则返回 False. 从 2.0pre74 版本开始, 整个过滤系统可以通过设置 *.allow_filtering* 为 False 来禁用. 这也可以作为参数传递到构造函数中.

MultiLine 控件是一个容器控件, 然后容纳一系列的处理各个显示部分的其他控件. 所有的多行控件类都会有一个 *_contained_widget* 类属性. 它控制控件如何被构建. 其 *_contained_widget_height* 类属性指定了给每个控件多少屏幕里的行.

TitleMultiLine 一个带标题版的 MultiLine.

如果要创建自己的 MultiLine 的子类, 你可以子类化该对象然后修改类的 *_entry_type* 变量就能创出一个带标题的版本了.

MultiSelect, TitleMultiSelect, 向用户给出一个选项的列表, 允许他/她选择其中的多个.

其 *value* 属性是一个用户所选项的索引的列表. 与 MultiLine 控件一样, 选项的列表存放在 *values* 属性中.

SelectOne, TitleSelectOne 功能上, 这些类跟 MultiLine 版本的差不多, 但是显示的跟 MultiSelect 控件更相似.

MultiSelectFixed, TitleMultiSelectFixed 这些 MultiSelect 特殊版本其实是用来显示数据的, 但是像 Textfixed 一样, 不允许用户去实际编辑它们.

MultiLineAction 这种控件的一个常见的使用场景就是, 在用户按下回车, 空格等键时, 对当前高亮的条目执行一个动作. 重写该类的 *actionHighlighted(self, act_on_this, key_press)* 方法即可做出这种控件. 此方法会在用户‘选中’一个条目时被调用 (虽然这种情况下.value 还没被实际设定), 并且被传进来高亮条目以及用户实际按下的键.

MultiSelectAction 这个跟上面的 MultiLineAction 控件类似, 不过它还提供 *actionSelected(self, act_on_these, keypress)* 方法. 这个可以被重写, 且如果用户按下’;’(分号) 键它会被调用. 这个函数会收到被选择的对象的列表和按下的键. 你或许会想调整它默认的按键绑定让它更好用.

BufferPager, TitleBufferPager ***2.0pre90* 版本新增** *BufferPager* 类似是 *Pager* 类的一个子类. 它被设计用来把文本以非常类似于在 *nix 环境下 tail -f 的方式显示给用户. 默认.value 属性被设为一个 *collections.deque* [双向队列] 类的实例. 你可以传递 *maxlen=* 到其构造函数. 否则, deque 对象的 maxlen[最大长度] 会从 *DEFAULT_MAXLEN* 类属性中获取, 而它默认是 None.

`BufferPager.clearBuffer()`
清空缓存.

`BufferPager.buffer(`*lines*, *scroll_end=True*, *scroll_if_editing=False*`)`
将 *lines* 添加到所包含的双向队列对象中. 如果 *scroll_end* 为 True, 则滚动到缓冲区的结尾. 如果 *scroll_if_editing* 为 True, 那么即使用户当前正在编辑页面控件也会滚动到末尾. 如果包含的双向队列对象在创建时被指定了最大长度, 那么新数据可能会导致较旧数据被遗忘.

MultiLineEditable 一个用户可以编辑的选项列表, 基于多行控件类. 3.9 版本新增.

`get_new_value()`
这个方法会返回一个’空白’对象, 它可以用来初始化列表中的新选项. 默认它返回一个空字符串.

`check_line_value(`*vl*`)`
这个方法会说明 vl 是否是一个可以被添加到列表的有效对象, 返回 True 或 False. 默认该方法拒绝空字符串.

MultiLineEditableTitle 标题版的 MultiLineEditable. 其 *_entry_type* 类属性控制着容纳的控件的类型.

MultiLineEditableBoxed 带框版的 MultiLineEditable. 其 *_entry_type* 类属性控制着容纳的控件的类型.

8.1 自定义多选控件

多行控件是一个容器控件, 它容纳一系列用于处理各显示部分的其他控件. 所有的多行控件类都有一个 *_contained_widget* 类属性. 这个控制控件如何被构造. *_contained_widget_height* 指定了应该给各个控件多少屏幕上的行.

从 3.4 版本以后, 内含的带有 *.selected* 属性的控件的处理有些不同了: 若行被选中, 则把它们的 *.selected* 属性设置为 *True* , 否则为 *False*. 控件的 *.important* 属性可能也会被设置为 True 或 False, 取决于他们是否被当前的 filter 所包含 (参见上面).

没有 *selected* 属性的控件, 每一行的值都会被放到 *name* 属性中, 且不论行是否被选中都放到它们的 *value* 属性中. 这是个遗留问题, 因为实际上标准的多选控件是用多个选框 [checkboxes] 来显示每一行的.

从 4.8.7 版本以后, 多行控件都开始使用 *set_is_line_important*, *set_is_line_bold* 和 *set_is_line_cursor* 方法来控制每行的显示. 这些方法会被传递进待选择的控件对象和一个布尔值. 它们都是用来被重写的.

CHAPTER 9

控件：树和显示树

9.1 表示树的数据

TreeData 类 TreeData 是被用于表示树对象。每一个树节点，包括根节点，是一个 NPSTreeData 实例。每个节点可能有它自己的内容，父节点或子节点。

每个节点的内容不是在创建的时候设置，就是使用 *.set_content* 方法设置。

get_content() 返回该内容。

get_content_for_display() 是控件用于显示树，这些控件期望返回一个能够给用户显示内容的字符串。

new_child(content=⋯) 创建一个新的子节点

selectable 如果这个属性为真，则用户可以标记一个值为’selected’。这被 MLTreeMultiSelect 控件使用，并且默认值为 True。

ignore_root 这个属性控制着是否向用户展示树的根节点。

expanded 这个属性控制着是否扩展树的分支，假设该节点有任何子节点。

sort 这个属性控制是否对树进行排序。

sort_function 如果树已排序，当树显示时，将使用此属性中命名的函数作为对树排序的键。

walk_tree(only_expanded=True, ignore_root=True, sort=None, sort_function=None) 遍历树。你可以覆盖标准 sort 和 sort 函数，并决定是否只遍历那些被标记为展开的树节点。

9.2 Trees

MLTree, MLTreeAction 这个类的 *values* 属性必须存储一个 NPSTree 实例。尽管如此，如果你希望覆盖类的 *convertToTree* 方法。这个方法应返回一个 NPSTree 实例。当 *values* 被赋值时，该函数自动被调用。

默认情况下，这个类使用 *TreeLine* 控件来显示树的每一行树。在派生类中，你可以通过改变类属性 *__contained_widgets* 来更改它。类属性 *__contained_widget_height* 定义了每个给定的控件显示多少行。

MLTreeAnnotated, MLTreeAnnotatedAction 在默认情况下，这个类使用 *TreeLineAnnotated* 控件来显示树的每一行。在派生类中，你可以通过改变类属性 *__contained_widgets* 来更改它。

MLTreeMultiSelect *New in version 2.0pre70*

这个类允许你选择树的多个项。你可以通过设置 NPSTreeData 节点的 *selectable* 属性来选择用户能够选择的 NPSTreeData 节点。

方法 *get_selected_objects(self, return_node=True,)* 返回一个列出被选中节点的生成器对象。如果 return_node 时 True，则产生实际节点本身，否则会产生 *node.getContent()* 的值。

New in Version 2.0pre71 如果属性 *select_cascades* 为 True(可通过创建时传递参数 *select_cascades* 或者后续直接设置属性来设置)，则选择节点将自动选择已选中节点下任何可选节点。默认设置为 True。

所选节点的属性 *selected* 也被设置为 True，因此，你可以遍历树来找到他们，如果你愿意的话。

用于显示每一行的控件是 *TreeLineSelectable* 。

MLTreeMultiSelectAnnotated *New in version 2.0pre71*

MLTreeMultiSelect 类的一个版本，是使用 *TreeLineSelectableAnnotated* 作为显示控件。

9.3 弃用的 Tree 类

NPSTreeData 不赞同使用的 TreeData 类。NPSTreedata 类是用于表示 tree 对象。每一个树节点，包括根节点，是一个 NPRSTreeData 实例。每个节点会有自己的内容，父节点或子节点。

每个节点的内容不是在创建的时候设置，就是使用 *.set_Content* 方法设置。

.getContent 返回内容。

.getContentForDisplay 是控件用于显示树，这些控件期望返回一个能够给用户显示内容的字符串。你可能想重载该方法。

newChild(content=…) 创建一个子节点。

selectable (new in version 2.0pre70) 如果该属性为真用户可标价一个值为’selected’。这是被 MLTreeMultiSelect 控件使用，且默认值为 True。

MultiLineTree, SelectOneTree, and MultiLineTree 这些控件以一种和无树版本非常相似的方式来工作，除非他们希望在.values 属性中包含一个 NPSTree。其他主要不同点是他们的.value 属性不包括所选值的索引，不过所选值索引本身。然而，这些类是弃用的，支持使用更多改进的 *MLTree* 和 *MLTreeAction* 类。

MultiLineTreeNew, MultiLineTreeNewAction 提供这些类仅仅是为了与旧版本兼容。新类应该使用 *MLTree* 和相关类

这个类的 *values* 属性必须存储一个 NPSTree 实例。不管怎样，如果你想，你可以重载这个类的 *convertToTree* 方法。这个方法将返回一个 NPSTree 实例。当 *values* 被赋值时，该函数自动被调用。

默认情况下，这个类使用 *TreeLineAnnotated* 控件来显示树的每一行树。在派生类中，你可以通过改变类属性 *__contained_widgets* 来更改它。

MutlilineTreeNewAnnotated, MultilineTreeNewAnnotatedAction 提供这些类仅仅是为了与旧版本兼容。新类应该使用 *MLTree* 和相关类

默认情况下，这个类使用 *TreeLineAnnotated* 控件来显示树的每一行树。在派生类中，你可以通过改变类属性 *__contained_widgets* 来更改它。

CHAPTER 10

控件：日期，滑块和组合控件

DateCombo, TitleDateCombo 这些控件允许用户选择一个日期。日期的实际选择是由类 MonthBox 完成的，它在一个临时窗口中显示。构造函数可以被传递通过以下参数 - *allowPastDate=False* 和 *allowTodaysDate=False* - 这两者将影响用户被允许选择什么。构造函数也可以接受参数 'allowClear=True'。

ComboBox, TitleCombo 这个框看起来像一个文本框，但用户仅仅能够从选项列表中选择。如果用户想更改值，就在临时窗口中显示。就像 MultiLine 控件一样，属性 *value* 是列表 *values* 中一个选择的索引。通过重载 *display_value(self, vl)* 方法，ComboBox 控件也可以被定制。

FilenameCombo, TitleFilenameCombo 这展示了一个选择文件名的控件。以下参数可传递给构造函数:

```
select_dir=False
must_exist=False
confirm_if_exists=False
sort_by_extension=True
```

这些也与控件中所示可以设置的属性相对应。

该控件在版本 2.0pre81 中被添加。

Slider, TitleSlider 滑块表示一个水平滑动块。以下附加参数对构造函数是很有用的:

out_of=100 滑块的最大值。

step=1 用户增加或减少值得增量。

lowest=0 用户能选择的最小值。注意，滑块设计不允许用户选择负值。*lowest* 应当 >= 0

label=True 是否在滑块旁打印文本标签。如果这样，请参考 *translate_value* 方法。

block_color = None 显示滑块的水平块的颜色。默认情况下（None），与滑块本身颜色相同。

所有这些选项设置了相同名称的属性，它们在控件中存在时可能随时被更改。

在控件 (如果 *label=True*) 旁边显示的文本是由 *translate_value* 方法生成。它没有选项并返回一个字符串。把 Slider 对象划入子类并重载该方法是有意义的。确保生成固定长度的字符串可能是有意义的。因此默认代码如下:

```
stri = "%s / %s" %(self.value, self.out_of)
l = (len(str(self.out_of)))*2+4
stri = stri.rjust(l)
return stri
```

SliderNoLabel, TitleSliderNoLabel 这些版本的滑块不显示标签。(类似于使用通常带 *label=False* 的滑块)。新版本 4.3.5

SliderPercent, TitleSliderPercent 这些版本的滑块在标签中显示百分比。通过设置属性 *accuracy* 或向构造函数传递关键字 *accuracy*，可以设置小数位数。默认是 2。新版本 4.3.5。

CHAPTER 11

控件：网格

SimpleGrid 这提供了一种类似电子表格的显示方式。默认值仅仅用于显示信息（在文本字段的网格中）。但是，它是为了灵活且易于自定义显示各种不同数据而设计的。未来的版本可能包括新的网格类型。注意，你可以通过在创建控件时指定 *columns* 或 *column_width* 来控制网格的外观。将来可能从这个类派生出其他多行类。

光标位置是由属性 *.edit_cell* 指定。注意，这遵循“先指定行，再指定列”的（奇数）curses 约定。

values 应被指定为一个二维数组。

方便的函数 *set_grid_values_from_flat_list(new_values, max_cols=None, reset_cursor=True)* 采取一个简单列表并显示在网格中。

以下参数可被传递给构造函数:

```
columns = None
column_width = None,
col_margin=1,
row_height = 1,
values = None
always_show_cursor = False
select_whole_line = False (new in version 4.2)
```

从 SimpleGrid 派生的类可能希望修改以下类属性:

```
_contained_widgets    = textbox.Textfield
default_column_number = 4
additional_y_offset   = 0  # 在网格前留在控件内部的额外偏移量
additional_x_offset   = 0  # 在网格前留在控件内部的额外偏移量
select_whole_line   # 高亮显示游标所在的整行
```

GridColTitles 像简单网格，但使用网格前两行来显示列标题。这些可在构造时作为 *col_titles* 参数提供，或者通过任意时间设置 *col_titles* 属性。在这两种情况下，提供一个字符串列表。

11.1 自定义单个网格单元的外观

新版本 4.8.2

对某些应用程序，最为理想的是自定义属性，那些包含依赖于他们内容的网格控件。网格控件在设置单元格的值之后，在单元格内容被绘制在屏幕上之前，调用一个名为 *custom_print_cell(actual_cell, display_value)* 的方法。参数 *actual_cell* 是用于显示底部控件对象，而 *display_value* 是被设置为单元格内容的对象（它是 *display_value* 方法的输出）。

以下代码演示如何使用这个工具来调整网格中显示的文本颜色。特别感谢 Johan Lundstrom 提出的这个特点:

```
class MyGrid(npyscreen.GridColTitles):
    # 你需要覆盖 custom_print_cell 来操作如果打印单元格。在本例中，我们根据单元格的字符串值更改文本颜色
    def custom_print_cell(self, actual_cell, cell_display_value):
        if cell_display_value == 'FAIL':
           actual_cell.color = 'DANGER'
        elif cell_display_value == 'PASS':
           actual_cell.color = 'GOOD'
        else:
           actual_cell.color = 'DEFAULT'

def myFunction(*args):
    # 制作一个实例表单
    F = npyscreen.Form(name='Example viewer')
    myFW = F.add(npyscreen.TitleText)
    gd = F.add(MyGrid)

    # 给网格添加值，这段代码只是通过任意 PASS/FAIL 字符串填充一个 2 x 4 网格
    gd.values = []
    for x in range(2):
```

(下页继续)

(续上页)

```
        row = []
        for y in range(4):
            if bool(random.getrandbits(1)):
                row.append("PASS")
            else:
                row.append("FAIL")
        gd.values.append(row)
    F.edit()

if __name__ == '__main__':
    npyscreen.wrapper_basic(myFunction)
```

CHAPTER 12

控件：其他控件

Checkbox, RoundCheckBox 这些提供一个单独选项 - 该标签是由属性 *name* 生成，作为标题控件。属性 *value* 为真或为假。

当用户切换到复选框时，调用函数 whenToggled(self)。你可以重载它。

CheckboxBare 这没有标签，并且只有在特殊情况下才有用。它是根据用户请求添加得。

CheckBoxMultiline, RoundCheckBoxMultiline 这控件允许复选框的标签超过一行。控件的名称应指定为字符串的列表或元组。

把这些控件作为多行控件的一部分来使用，执行以下代码:

```
class MultiSelectWidgetOfSomeKind(npyscreen.MultiSelect):
    _contained_widgets = npyscreen.CheckBoxMultiline
    _contained_widget_height = 2

    def display_value(self, vl):
        # 这个函数应返回字符串列表
```

新版本 2.0pre83。

Button 功能上与复选框控件相似，但看起来不相同。按钮通常用在表单和相似内容上的 OK 和 Cancel 按钮，尽管它们可能应被 ButtonPress 类型替代。按钮被选中时的颜色要么与按钮的颜色相反，要么通过属性 *cursor_color* 来选择。这个值也可传递给构造函数。如果这个值是 None, 将使用按钮颜色的相反值。

ButtonPress 不是一个触发器，仅仅一个控件。这个控件由方法 *whenPressed(self)*，该方法你应该重载来做你自己的事情。

从版本 4.3.0 开始，该构造函数接受一个参数 *when_pressed_function=None*。如果以这个方式指定一个 callable，它将被调用，而不是方法 *whenPressed*. NB. when_pressed_function 功能上有潜在危险。它能设置一个循环引用，该垃圾收集器将永远不会释放。如果这对你的程序存在风险，最好把这个对象设置为子类，并且重载*when_pressed_function* 方法来替代。

FormControlCheckbox 复选框的一般用法是为用户提供输入额外数据的选项。例如 “输入有效期”。在这种情况下，表单在某些情况下需要显示额外字段，但在其他情况下则不需要。FormControlCheckbox 使这变得很简单。

定义两种方法:

addVisibleWhenSelected(*wg*) *wg* 应是一个控件

这个方法不创建一个控件，而是将现有控件置于 FormControlCheckbox 的控制之下。如果已选择 FormControlCheckbox，该控件就是可见的。

通过这种方式按照你的意愿添加许多控件。

addInvisibleWhenSelected(*wg*) 只有在 FormControlCheckbox 没被选择时，以这种方式注册的控件才是可见的。

AnnotateTextboxBase, TreeLineAnnotated, TreeLineSelectableAnnotated *AnnotateTextboxBase* 类主要是为了给多行列表控件使用，用于显示的每一项都需要在选项本身左侧提供注释的情况。这些类的 API 有些难看，因为这些类最初只用于内部管理。在以后的本本中可能会提供更友好的版本。派生自 *AnnotateTextboxBase* 的类应定义以下内容:

ANNOTATE_WIDTH 这个类属性定义了在文本输入控件本身之前按需要留出多少边距。在 TreeLineAnnotated 类中，边距需要动态计算，且不需要 ANNOTATE_WIDTH。

getAnnotationAndColor 这个函数应返回一个元组，该元组由作为注释显示的字符串和显示时使用颜色名字组成。在 B/W 显示器上，颜色会被忽略，但在所有情况下都应提供，并且字符串长度应不超过 *ANNOTATE_WIDTH*，尽管默认情况下该类不检查这个。

annotationColor*, *annotationNoColor 这些方法在屏幕上绘制注释。如果仅使用字符串，这些应不需要覆盖。如果其中一个被更改，另一个也应改变，因为 npyscreen 将使用一个的显示被配置为彩色，且另一个为黑白色。

CHAPTER 13

控件：带标题的控件

大多数标准控件版本有两种形式 - 一个基础版本和一个相应版本，后者也打印一个带控件名称的标签。例如，Textfield 和 TitleText。

标题版本实际上是一个包含控件的装饰器，而不是成为一个对它们本身合适的控件，并且会在修改它们行为时造成混乱。

In general, to create your own version of these widgets, you should first create the contained widget, and then create a titled version. 通常情况下，为了创建自己的这些控件版本，你首先应该创建包含控件，然后创建一个标题版本。

例如:

```
class NewTextWidget(textbox.Textfield):
        # 这个类的所有自定义代码
        # 应在这里


class TitleProductSearch(TitleText):
        _entry_type = NewTextWidget
```

你可以通过将参数 *begin_entry_at* 传递给构造函数来调整子控件在屏幕上的位置。默认值是 16。你也可以通过在创建控件时传递参数 *use_two_lines=True/False* 来覆盖是否使用单独行做标题的控件。默认的 *use_two_lines=None* 将保持标题和包含控件在同一行，除非标签太长。

你可以在创建时使用参数 *labelColor='LABEL'* 来改变标签颜色。你可以从你正在使用的主题中指定任何颜色名称。

创建之后，由 TitleWidget 管理的两个控件可以通过对象的 *label_widget* 和 *entry_widget* 属性来访问。

13.1 带标题的多行控件

如果你正在创建带标题的多行控件版本，你将发现最好的方法是从类 *TitleMultiLine* 继承，它封装了更多的多行功能

CHAPTER 14

控件：框控件

这些控件以一种与带标题的控件版本类似的方式工作。box 控件包括另一个类的控件。

BoxBasic BoxBasic 在屏幕上打印一个带可选名称和页脚的框。它是为了作为一个深一层控件的基类，不是直接使用。

BoxTitle

BoxTitle 是 Title 控件和多行控件的混合物。此外，它的主要目的是作为更复杂布局的基类。这个类有一个 *_contained_widget* 属性，它在创建时将控件放入框中。在 Boxtitle 类，这是一个多行控件。控件的标题可以传递给参数 *name=⋯*. 的 *___init___*。另一个周长 *footer=⋯*给页脚框的文本。这些对应的属性命名为 *name* 和 *footer*，可以随时更改。

属性 *entry_widget* 提供对包含控件的直接访问。

属性 *editable*、*values* 和 *value* 提供对 *entry_widget* 属性的直接访问。

这个控件的构造函数可以传递给参数 *contained_widget_arguments*。这个应是一个参数字典，在 entry_widget 创建时将被传递。注意，此时没有对该字典进行完整性检查。(4.8.0 版本新增)

你可以用与新的标题控件相同的方式来创建这些控件自己的版本。首先创建包含控件，然后创建 box 类封装类:

```
class NewMultiLineClass
        # 在这里做巧妙的事情!
        # ....
```

(下页继续)

(续上页)

```
class BoxTitle(BoxBasic):
    _contained_widget = NewMultiLineClass
```

CHAPTER 15

关于键绑定

15.1 这是怎么回事

许多对象可以基于用户按键来操作。所有这样的对象继承自内部类 InputHandler。该类定义了一个名为 *handlers* 的字典和一个名为 *complex_handlers* 的列表。它们两者都是通过在构造函数时调用一个名为 *set_up_handlers* 的方法来设置。

handlers 可能看起来像这样:

```
{curses.ascii.NL:   self.h_exit_down,
 curses.ascii.CR:   self.h_exit_down,
 curses.ascii.TAB:  self.h_exit_down,
 curses.KEY_DOWN:   self.h_exit_down,
 curses.KEY_UP:     self.h_exit_up,
 curses.KEY_LEFT:   self.h_exit_left,
 curses.KEY_RIGHT:  self.h_exit_right,
 "^P":              self.h_exit_up,
 "^N":              self.h_exit_down,
 curses.ascii.ESC:  self.h_exit_escape,
 curses.KEY_MOUSE:  self.h_exit_mouse,
 }
```

如果按下一个作为在字典中以键值对存在的键（注意，支持标记像 “^N” 代表 “Control-N” 和 ”!a” 代表 “Alt N”），与它关联的函数将被调用。没有采取进一步的动作。按照惯例，处理用户输入的函数以 h_ 为前缀。

complex_handlers 这个列表应包含列表或元组对类似这样 (test_func, dispatch_func)。

如果该键未在字典 *handlers* 中命名，则运行每个 test_func。如果它返回 True，则运行 dispatch_func 并停止搜索。

例如，使用复杂的处理程序是为了保证只将可打印字符输入到文本框中。因为它们将被频繁运行，应尽可能少并且尽可能快地运行它们。

当一个用户正在编辑控件和按下按键，使用 *handlers* 和 *complex_handlers* 来尝试找到一个执行的函数。如果控件没有定义要执行的操作，则检查父表单的 *handlers* 和 *complex_handlers*。因此，如果你想覆盖已绑定键的处理程序，请记住你在控件上操作而不是在它们的所属表单上，因为控件处理程序优先。

15.2 添加自己的处理程序

可以处理用户输入的对象定义了以下方法来帮助添加自己的键绑定:

```
*add_handlers(new_handlers)*
```

new_handlers 应是一个字典。

add_complex_handlers(new_handlers) *new_handlers* 应是列表的列表。每个子列表必须是一对 *(test_function, callback)*

CHAPTER 16

增强鼠标支持

想要更细致的处理鼠标事件的控件需要重写 *.handle_mouse_event(self, mouse_event)* 方法. 注意 *mouse_event* 是一个元组:

```
def handle_mouse_event(self, mouse_event):
    mouse_id, x, y, z, bstate = mouse_event # 参阅下面的提示.
    # 这里处理后续任务 ....
```

这个通常很有用, 但是 x 和 y 都是绝对位置, 而不是相对位置. 因为这个原因, 你应该利用已经提供好的便利功能来把这些值转换成与相对于控件的坐标. 这样的话, 大多的鼠标处理函数都会看起来像这样:

```
def handle_mouse_event(self, mouse_event):
    mouse_id, rel_x, rel_y, z, bstate = self.interpret_mouse_event(mouse_event)
    # Do things here.....
```

鼠标控制器只会在控件是“可编辑”的条件下被调用. 只有很少见的情况下, 你可能会想用一个不可编辑的控件来响应鼠标事件. 那种情况下, 你可以把控件的 *.self.interested_in_mouse_even_when_not_editable* 属性设为 True.

关于鼠标事件的更多细节参见 python 标准库的 curses 模块文档.

CHAPTER 17

对颜色的支持

17.1 设置颜色

所有的标准控件完全可以在黑白终端上显示。然而，目前这是一个多彩的世界，npyscreen 让你显示你的控件，当然，如果不是 Technicolor(TM)，并且尽可能允许接近 curses。

颜色由 ThemeManager 类来处理。一般来说，你的应用程序应坚持使用 ThemeManager，你应使用 *setTheme(ThemeManager)* 函数来设置它。所以举个例子:

```
npyscreen.setTheme(npyscreen.Themes.ColorfulTheme)
```

任何由 npyscreen 定义的默认主题将通过 npyscreen.Themes 访问。

一个基础主题看起来像这样:

```
class DefaultTheme(npyscreen.ThemeManager):
    default_colors = {
    'DEFAULT'     : 'WHITE_BLACK',
    'FORMDEFAULT' : 'WHITE_BLACK',
    'NO_EDIT'     : 'BLUE_BLACK',
    'STANDOUT'    : 'CYAN_BLACK',
    'CURSOR'      : 'WHITE_BLACK',
    'CURSOR_INVERSE': 'BLACK_WHITE',
    'LABEL'       : 'GREEN_BLACK',
    'LABELBOLD'   : 'WHITE_BLACK',
```

(下页继续)

(续上页)

```
    'CONTROL'     : 'YELLOW_BLACK',
    'IMPORTANT'   : 'GREEN_BLACK',
    'SAFE'        : 'GREEN_BLACK',
    'WARNING'     : 'YELLOW_BLACK',
    'DANGER'      : 'RED_BLACK',
    'CRITICAL'    : 'BLACK_RED',
    'GOOD'        : 'GREEN_BLACK',
    'GOODHL'      : 'GREEN_BLACK',
    'VERYGOOD'    : 'BLACK_GREEN',
    'CAUTION'     : 'YELLOW_BLACK',
    'CAUTIONHL'   : 'BLACK_YELLOW',
    }
```

颜色 - 例如 WHITE_BLACK(“黑底白字”) - 是在 ThemeManager 类的 *initialize_pairs* 方法中定义的:

```
('BLACK_WHITE',      curses.COLOR_BLACK,      curses.COLOR_WHITE),
 ('BLUE_BLACK',       curses.COLOR_BLUE,       curses.COLOR_BLACK),
 ('CYAN_BLACK',       curses.COLOR_CYAN,       curses.COLOR_BLACK),
 ('GREEN_BLACK',      curses.COLOR_GREEN,      curses.COLOR_BLACK),
 ('MAGENTA_BLACK',    curses.COLOR_MAGENTA,    curses.COLOR_BLACK),
 ('RED_BLACK',        curses.COLOR_RED,        curses.COLOR_BLACK),
 ('YELLOW_BLACK',     curses.COLOR_YELLOW,     curses.COLOR_BLACK),
)
```

(‘WHITE_BLACK’ 总是明确的。)

如果你发现你需要更多类，把 ThemeManager 归为子类并且改变类属性 *__colours_to_define*。你可以使用除了标准 curses 颜色的其他颜色，但由于不是所有终端都支持这样做，npyscreen 不默认这样。

如果你想禁用应用程序中的所有颜色，npyscreen 定义了两个方便的函数: *disableColor()* 和 *enableColor()*。

17.2 控件如何使用颜色

当一个控件被绘制时，它要求有效的 ThemeManager 去告诉它适当的颜色。例如，’ LABEL’，是一个给予颜色的标签，将用于控件的标签。主题管理器在它的 *default_colors* 字典中查找相关名称，并且返回合适的 colour-pair 作为 curses 属性，该属性随后用于在屏幕上绘制控件。

单个控件往往有自己的 *color* 属性（可能由构造函数设置）。这是通常设置为 ‘DEFAULT’，但可以更改为任何其他定义的名称。这种机制很典型地仅仅允许单个控件更改其颜色方案的某个特定部分。

标题…控件版本也定义了属性 *labelColor*，它可以用于改变他们标签颜色的风格

17.3 自定义颜色（强烈反对）

在某些终端上，可以自定义颜色值。txvt/urxvt 是这样一种终端。从版本 4.8.4 开始，theme manager 类内置了对此的支持。

类变量 color_values 将在类被初始化来重新自定义颜色值时被使用:

```
_color_values = (
                # 重新定义一种标准颜色
        (curses.COLOR_GREEN, (150,250,100)),
                # 定义另一种颜色
                (70, (150,250,100)),
    )
```

NB. 当前的 npyscreen 版本在应用程序退出时没有尝试重置这些值

使用这些设备是被阻止的，因为准确地判断一个终端是否确实支持自定义颜色是不可能的。该功能是根据用户请求添加的，为了支持定制的应用程序。

CHAPTER 18

显示简明的消息和选择

下列函数允许你向用户显示一个简短的消息或选择。

通知相关的方法是在 npyscreen/utilNotify.py 中实现的。

本页的例子从这个基本程序上建立的:

```
import npyscreen


class NotifyBaseExample(npyscreen.Form):
    def create(self):
        key_of_choice = 'p'
        what_to_display = 'Press {} for popup \n Press escape key to quit'.format(key_of_
↪choice)

        self.how_exited_handers[npyscreen.wgwidget.EXITED_ESCAPE] = self.exit_application
        self.add(npyscreen.FixedText, value=what_to_display)

    def exit_application(self):
        self.parentApp.setNextForm(None)
        self.editing = False


class MyApplication(npyscreen.NPSAppManaged):
```

(下页继续)

(续上页)

```
    def onStart(self):
        self.addForm('MAIN', NotifyBaseExample, name='To be improved upon')


if __name__ == '__main__':
    TestApp = MyApplication().run()
```

notify(*message*, *title="Message"*, *form_color='STANDOUT'*, *wrap=True*, *wide=False*)

这个函数在屏幕上显示一条消息。它不阻塞，且用户不会与它交互 - 在其他事情正在发生时，使用它来显示类似 “Please wait” 的消息。

列表 1: ../examples/notify/notify.py snippet

```
import npyscreen
import time


class NotifyExample(npyscreen.Form):
    def create(self):
        key_of_choice = 'p'
        what_to_display = 'Press {} for popup \n Press escape key to quit'.
↪format(key_of_choice)

        self.how_exited_handers[npyscreen.wgwidget.EXITED_ESCAPE] = self.exit_
↪application
        self.add_handlers({key_of_choice: self.spawn_notify_popup})
        self.add(npyscreen.FixedText, value=what_to_display)

    def spawn_notify_popup(self, code_of_key_pressed):
        message_to_display = 'I popped up \n passed: {}'.format(code_of_key_pressed)
        npyscreen.notify(message_to_display, title='Popup Title')
        time.sleep(1) # needed to have it show up for a visible amount of time
```

notify_wait(*message*, *title="Message"*, *form_color='STANDOUT'*, *wrap=True*, *wide=False*)

这个函数在屏幕上显示一条消息，且阻塞很短的一段时间。用户不会与它进行交互。

列表 2: ../examples/notify/notify_wait.py snippet

```
class NotifyWaitExample(npyscreen.Form):
    def create(self):
        key_of_choice = 'p'
```

(下页继续)

(续上页)

```
7           what_to_display = 'Press {} for popup \n Press escape key to quit'.
    ↪format(key_of_choice)
8
9           self.how_exited_handers[npyscreen.wgwidget.EXITED_ESCAPE] = self.exit_
    ↪application
10          self.add_handlers({key_of_choice: self.spawn_notify_popup})
11          self.add(npyscreen.FixedText, value=what_to_display)
12
13      def spawn_notify_popup(self, code_of_key_pressed):
14          message_to_display = 'I popped up \n passed: {}'.format(code_of_key_pressed)
15          npyscreen.notify_wait(message_to_display, title='Popup Title')
```

notify_confirm(*message*, *title="Message"*, *form_color='STANDOUT'*, *wrap=True*, *wide=False*, *editw=0*)

显示一条消息和 OK 按钮。如果需要，用户可以滚动消息。editw 控制对话框首次显示时选择哪一个控件；设置为 1 则立刻激活 OK 按钮。

列表 3: ../examples/notify/notify_confirm.py snippet

```
4   class NotifyConfirmExample(npyscreen.Form):
5       def create(self):
6           key_of_choice = 'p'
7           what_to_display = 'Press {} for popup \n Press escape key to quit'.
    ↪format(key_of_choice)
8
9           self.add_handlers({key_of_choice: self.spawn_notify_popup})
10          self.how_exited_handers[npyscreen.wgwidget.EXITED_ESCAPE] = self.exit_
    ↪application
11          self.add(npyscreen.FixedText, value=what_to_display)
12
13      def spawn_notify_popup(self, code_of_key_pressed):
14          message_to_display = 'You need to confirm me, so hit TAB, then ENTER'
15          npyscreen.notify_confirm(message_to_display, title= 'popup')
```

notify_ok_cancel(*message*, *title="Message"*, *form_color='STANDOUT'*, *wrap=True*, *editw = 0*)

显示一条消息, 如果用户选择‘OK’, 则返回 True；如果用户选择‘Cancel’, 则返回 False。

列表 4: ../examples/notify/notify_ok_cancel.py snippet

```
4   class NotifyOkCancelExample(npyscreen.Form):
5       def create(self):
6           key_of_choice = 'p'
```

(下页继续)

(续上页)

```
        what_to_display = 'Press {} for popup \n Press escape key to quit'.
↪format(key_of_choice)

        self.add_handlers({key_of_choice: self.spawn_notify_popup})
        self.how_exited_handers[npyscreen.wgwidget.EXITED_ESCAPE] = self.exit_
↪application
        self.add(npyscreen.FixedText, value=what_to_display)

    def spawn_notify_popup(self, code_of_key_pressed):
        message_to_display = 'You have a choice, to Cancel and return false, or Ok␣
↪and return true.'
        notify_result = npyscreen.notify_ok_cancel(message_to_display, title= 'popup
↪')
        npyscreen.notify_wait('That returned: {}'.format(notify_result), title=
↪'results')
```

notify_yes_no(*message*, *title="Message"*, *form_color='STANDOUT'*, *wrap=True*, *editw = 0*)

与 *notify_ok_cancel* 相似，除了按钮名称是 'Yes' 和 'No'，返回 True 或 False。

selectFile(*select_dir=False*, *must_exist=False*, *confirm_if_exists=True*, *sort_by_extension=True*)

显示一个提示用户选择文件名的对话框。使用来自目录的调用作为初始化文件夹。返回值是所选文件的名称。

Warning: 这个形式当前还是试验阶段。

列表 5: ../examples/notify/select_file.py snippet

```
class SelectFileExample(npyscreen.Form):
    def create(self):
        key_of_choice = 'p'
        what_to_display = 'Press {} for popup \n Press escape key to quit'.
↪format(key_of_choice)

        self.how_exited_handers[npyscreen.wgwidget.EXITED_ESCAPE] = self.exit_
↪application
        self.add_handlers({key_of_choice: self.spawn_file_dialog})
        self.add(npyscreen.FixedText, value=what_to_display)

    def spawn_file_dialog(self, code_of_key_pressed):
        the_selected_file = npyscreen.selectFile()
        npyscreen.notify_wait('That returned: {}'.format(the_selected_file), title=
↪'results')
```

(下页继续)

(续上页)

CHAPTER 19

空白的屏幕

`blank_terminal()`

这个函数使终端消失。如果显示的表单没有占满整个屏幕，它有时可能被需要。

列表 1: ../examples/notify/blank_terminal.py snippet

```
import npyscreen
import time


class BlankTerminalExample(npyscreen.Form):
    def create(self):
        key_of_choice = 'b'
        what_to_display = 'Press {} to blank screen \n Press escape key to quit'.
↪format(key_of_choice)

        self.how_exited_handers[npyscreen.wgwidget.EXITED_ESCAPE] = self.exit_
↪application
        self.add_handlers({key_of_choice: self.initiate_blanking_sequence})
        self.add(npyscreen.FixedText, value=what_to_display)

    def initiate_blanking_sequence(self, code_of_key_pressed):
        npyscreen.blank_terminal()
        time.sleep(1.5)
```

(下页继续)

(续上页)

```
        npyscreen.notify('..and we\'re back', title='Phew')
        time.sleep(0.75)

    def exit_application(self):
        self.parentApp.setNextForm(None)
        self.editing = False


class MyApplication(npyscreen.NPSAppManaged):
    def onStart(self):
        self.addForm('MAIN', BlankTerminalExample, name='To show off blank_screen')


if __name__ == '__main__':
    TestApp = MyApplication().run()
```

CHAPTER 20

应用支持

20.1 选项和选项列表

一个常见问题就是显示一个选项列表给用户. 在简单应用中, 为了这个目的可能会用一个自定义设计的窗口, 但是很多任务中自动生成的窗口会更合适. 一个支持该特性的 *实验性的*系统在 2.0pre84 版本被引入.

这个系统的核心是 *Option* 对象的概念. 这些对象根据它们的类型, 要么存放单一值, 要么存放值列表, 同时与待定选项相关的所有文档应该也会展示给用户. 选项对象是用下面的参数创建的: *OptionType(name, value=None, documentation=None, short_explanation=None, option_widget_keywords = None, default = None)*. *short_explanation* 参数当前还没有在默认控件中使用, 但是会在未来的版本中被使用. 选项对象被设计成可让用户从也可能是由 *choices* 参数创建的有限的选项中进行选择.

所有的选项类也都有 DEFAULT 和 WIDGET_TO_USE 类属性. 如果默认值还没被定义的话, 前者会定义默认值. 第二个定义了让用户来调节待定选项的控件的类.

下面是当前版本定义的选项类: *OptionFreeText*, *OptionSingleChoice*, *OptionMultiChoice*, *OptionMultiFreeList*, *OptionBoolean*, *OptionFilename*, *OptionDate*, *OptionMultiFreeText*. 存到 option 对象的值都应该用 *set(value)* 方法设定, 用 *get()* 方法获取. 所有的选项类也都定义了一个可被重写的 *when_set* 方法, 而它会在值被修改之后调用. 允许用户从一系列的有限项作选择的选项类还有 *setChoices(choices)* 和 *getChoices* 方法.

选项列表可以使用 *OptionListDisplay* 控件来显示. 它接收选项列表当作其 *OptionListDisplay* 属性的值. 如果某个选项被选中, 一个给用户显示文档 (如果有的话) 的窗口会展示给用户并让用户改变它的值.

选项集合可以用一个 *OptionList* 对象合在一起. *OptionList* 类有一个 *options* 属性. 它只是一个列表, 选项对象可能被被加进来. 未来的版本可能会定义一个不一样的 API. *OptionList* 对象的目的帮助保存和恢复选项集合到文件中的. 这些文件的格式是一个自定义的文本格式, 类似于标准的 Unix 文件, 但是可以存储和恢

复字符串列表 (使用 tab[制表符] 作为分隔符). 这个格式还在演进, 并可能未来的版本中被改变. 只有和默认值不同的值会被存下来.

OptionList 对象的创建可以带 *filename* 参数, 它有 *write_to_file(fn=None)* 和 *reload_from_file(fn=None)* 方法.

SimpleOptionForm 类是一个设计用来展示这些元素如何运作的窗口. *OptionListDisplay* 控件作为一个名为 *wOptionList* 的属性被创建.

20.2 示例代码

下面简短的 demo 程序会在调用中存储选定的选项到文件 '/tmp/test'

```
#!/usr/bin/env python
# encoding: utf-8

import npyscreen
class TestApp(npyscreen.NPSApp):
    def main(self):
        Options = npyscreen.OptionList()

        # 为了方便让我们不用不停的写 Options.options
        options = Options.options

        options.append(npyscreen.OptionFreeText('FreeText', value='', documentation=
↪"This is some documentation."))
        options.append(npyscreen.OptionMultiChoice('Multichoice', choices=['Choice 1',
↪'Choice 2', 'Choice 3']))
        options.append(npyscreen.OptionFilename('Filename', ))
        options.append(npyscreen.OptionDate('Date', ))
        options.append(npyscreen.OptionMultiFreeText('Multiline Text', value=''))
        options.append(npyscreen.OptionMultiFreeList('Multiline List'))

                try:
                Options.reload_from_file('/tmp/test')
        except FileNotFoundError:
                        pass

        F  = npyscreen.Form(name = "Welcome to Npyscreen",)

        ms = F.add(npyscreen.OptionListDisplay, name="Option List",
                values = options,
```

(下页继续)

(续上页)

```
                scroll_exit=True,
                max_height=None)

        F.edit()

        Options.write_to_file('/tmp/test')

if __name__ == "__main__":
    App = TestApp()
    App.run()
```

CHAPTER 21

编写更复杂的表单

终端应用程序的一个非常典型的编程风格是有一个带命令行的屏幕，通常显示在屏幕底部，然后一些列表控件或其他显示占用了大部分屏幕顶部的标题栏和状态栏上方的命令行。在这个模式上的变化在像 Mutt、less、Vim、irssi 等应用程序中可以找到。

为了让编写这些类型的表单更容易，npyscreen 提供一系列能够协同工作的类。

FormMuttActive, FormMuttActiveWithMenus, FormMuttActiveTraditional, FormMuttActiveTraditionalW

这些类定义了基本表单。以下 *class attribute* 明确指定了表单如何工作:

```
MAIN_WIDGET_CLASS   = wgmultiline.MultiLine
MAIN_WIDGET_CLASS_START_LINE = 1
STATUS_WIDGET_CLASS = wgtextbox.Textfield
STATUS_WIDGET_X_OFFSET = 0
COMMAND_WIDGET_CLASS= wgtextbox.Textfield
COMMAND_WIDGET_NAME = None
COMMAND_WIDGET_BEGIN_ENTRY_AT = None
COMMAND_ALLOW_OVERRIDE_BEGIN_ENTRY_AT = True

DATA_CONTROLER    = npysNPSFilteredData.NPSFilteredDataList

ACTION_CONTROLLER  = ActionControllerSimple
```

默认定义使以下实例属性在初始化后可用:

```
# Widgets -
self.wStatus1 # by default a title bar
self.wStatus2 # just above the command line
self.wMain    # the main area of the form - by default a MultiLine object
self.wCommand # the command widget

self.action_controller # not a widget. See below.
```

表单的 *.value* 属性设置为一个指定对象实例 DATA_CONTROLLER

通常，应用程序希望定义属于自己的 DATA_CONTROLLER 和 ACTION_CONTROLLER。

传统和非传统的表单之间的区别是，传统表单的重点总是停留在命令行控件，尽管一些按键将传递到 MAIN_WIDGET_CLASS - 所以，从用户角度看，看起来像他们同时在交互。

TextCommandBox TextCommandBox 是像一个普通的文本框，只是将用户输入传递给了 action_controller。另外，它可以保持命令输入历史记录。请参阅 ActionControllerSimple 文档，获取更多细节。

TextCommandBoxTraditional 这个与 TextCommandBox 相同，不同点在于它将特定的按键传递给由 *self.linked_widget* 指定的控件。在默认情况下，任何与 TextCommanBoxTraditional 中处理程序不匹配的按键都将被传递给链接控件。除此之外，列表 *self.always_pass_to_linked_widget* 列出的任何按键将被链接控件处理。但是，如果当前命令行是以类属性 *BEGINNING_OF_COMMAND_LINE_CHARS* 列表中的任意字母开头，用户输入将由该类处理，而不是链接控件。

这相当复杂，但举个例子可能会更清晰。默认的 BEGINNING_OF_COMMAND_LINE_CHARS 明确’:’或’/’标志着命令行的开始。在那之后，按键由这个控件处理，而不是链接控件，因此向上和向下的箭头开始导航命令历史记录。但是，如果当前命令行是空的，这些键将导航到链接控件。

与 TextCommandBox 控件一样，命令行的值传递给父窗体的 action_controller 对象。

ActionControllerSimple 这个对象接受命令行并且执行 call-back 函数。

它可识别两种类型的命令行 - 一个”live”命令行，指命令行每次更新都会执行一个动作，并且在按下返回键时执行一条命令。

使用 *add_action(ident, function, live)* 方法添加回调。’ident’是一个正则表达式，它将与命令行相匹配，*function* 是回调本身，*live* 为真或为假，用于指定是否应该在每次按键时执行回调（假设”ident”匹配）。

匹配正则表达式’ident’的命令行导致回调函数去调用以下参数：*call_back(command_line, control_widget_proxy, live=True)*。这里 *command_line* 是命令行的字符串，*control_widget_proxy* 是对命令行控件的弱引用，live 指定这个函数将调用’live’还是作为一个返回结果。

方法 *create()* 可被覆盖。当创建对象是调用它。默认不做任何事。你可能想使用它来调用 *self.add_action*。

NPSFilteredDataBase 默认 *NPSFilteredDataBase* 类建议如何管理代码的显示可能被分隔成一个单独的对象。精确的方法将非常依赖于应用程序。这不是此类型应用程序最重要的部分，但这是把数据库访

问逻辑从用户接口逻分离出来最好的实践。

21.1 Example Code 示例代码

以下示例展示该模块如果工作。应用程序创建一个由搜索功能的 ActionController。该功能调用了用户自定义函数 set_search，它与表单的父类进行通信。值（实际是 NPSFilteredDataBase 类）然后它使用这个类去设置 wMain.values 并且调用 wMain.display() 来更新显示。

FmSearchActive 是一个简单的 FormMuttActiveTraditional 类，它有一个类属性，指定表单应使用我们的功能控制器:

```
class ActionControllerSearch(npyscreen.ActionControllerSimple):
    def create(self):
        self.add_action('^/.*', self.set_search, True)

    def set_search(self, command_line, widget_proxy, live):
        self.parent.value.set_filter(command_line[1:])
        self.parent.wMain.values = self.parent.value.get()
        self.parent.wMain.display()


class FmSearchActive(npyscreen.FormMuttActiveTraditional):
    ACTION_CONTROLLER = ActionControllerSearch

class TestApp(npyscreen.NPSApp):
    def main(self):
        F = FmSearchActive()
        F.wStatus1.value = "Status Line "
        F.wStatus2.value = "Second Status Line "
        F.value.set_values([str(x) for x in range(500)])
        F.wMain.values = F.value.get()

        F.edit()


if __name__ == "__main__":
    App = TestApp()
    App.run()
```

CHAPTER 22

编写测试

(4.7.0 版本新增)

脚本化 npyscreen 应用的键盘输入用于测试目的是可行的.

npyscreen 模块导出下面包含了相关设置的字典:

```
TEST_SETTINGS = {
    'TEST_INPUT': None,
    'TEST_INPUT_LOG': [],
    'CONTINUE_AFTER_TEST_INPUT': False,
    'INPUT_GENERATOR': False
    }
```

如果‘TEST_INPUT’是 None, 应用照常进行. 如果它是一个数组, 按键动作会从数组的左边开始被加载, 然后送到应用程序获取键盘输入的地方. 注意, 像 *curses.KEYDOWN* 这样的特殊字符也可以被处理, 而控制字符可以用像 “^X” 这样的字符串来表示.

这种方式发送到应用程序的按键动作会自动的添加到 ‘*TEST_INPUT_LOG*’ , 所以就能看到在处理输入的时候在哪里出现了错误.

如果‘CONTINUE_AFTER_TEST_INPUT’为 True, 那么在指定自动输入之后, ‘*TEST_INPUT*’就被设成 *None* 并且应用程序会正常继续. 如果它为 False, 那就会抛出 *ExhaustedTestInput* 异常. 这可以让单元测试来接着检测应用的状态.

‘INPUT_GENERATOR’可以设置为一个可迭代对象. 调用 *next(INPUT_GENERATOR)* 每一个按键都会被读取. 假如可迭代对象是线程安全的, 这就让用一个线程来提供测试用的输入变得容易了. 相对

TEST_INPUT 可以多用一下这个. 在 4.9 版本中按用户要求加入.

22.1 便捷函数 (4.8.5 版本新增)

npyscreen.add_test_input_from_iterable(*iterable*)

将每个 *iterable* 项加入 *TEST_SETTINGS[‘TEST_INPUT’]*.

npyscreen.add_test_input_ch(*ch*)

添加 *ch* 到 *TEST_SETTINGS[‘TEST_INPUT’]*.

22.2 防止编写单元测试产生分叉

为了避免底层 curses 模块的内存溢出, npyscreen 库有时会悬着在 fork 出的进程中运行应用程序代码. 处于一定的测试目的这可能不是我们想要的, 或许会想在你的应用程序中传递 *fork=False* 到 *run()* 方法来测试.

22.3 例子

下面是个小例子:

```
#!/usr/bin/python
import curses
import npyscreen

npyscreen.TEST_SETTINGS['TEST_INPUT'] = [ch for ch in 'This is a test']
npyscreen.TEST_SETTINGS['TEST_INPUT'].append(curses.KEY_DOWN)
npyscreen.TEST_SETTINGS['CONTINUE_AFTER_TEST_INPUT'] = True

class TestForm(npyscreen.Form):
    def create(self):
        self.myTitleText = self.add(npyscreen.TitleText, name="Events (Form Controlled):
↪", editable=True)

class TestApp(npyscreen.StandardApp):
    def onStart(self):
        self.addForm("MAIN", TestForm)


if __name__ == '__main__':
    A = TestApp()
```

(下页继续)

(续上页)

```
    A.run(fork=False)
    # 'This is a test' will appear in the first widget, as if typed by the user.
```

CHAPTER 23

示例程序: 一个简单的通讯录

一个通讯录程序需要一个数据库. 为了方便, 我们会用 sqlite.

```
class AddressDatabase(object):
    def __init__(self, filename="example-addressbook.db"):
        self.dbfilename = filename
        db = sqlite3.connect(self.dbfilename)
        c = db.cursor()
        c.execute(
        "CREATE TABLE IF NOT EXISTS records\
            ( record_internal_id INTEGER PRIMARY KEY, \
              last_name     TEXT, \
              other_names   TEXT, \
              email_address TEXT \
              )" \
            )
        db.commit()
        c.close()

    def add_record(self, last_name = '', other_names='', email_address=''):
        db = sqlite3.connect(self.dbfilename)
        c = db.cursor()
        c.execute('INSERT INTO records(last_name, other_names, email_address) \
                    VALUES(?,?,?)', (last_name, other_names, email_address))
```

(下页继续)

(续上页)

```
        db.commit()
        c.close()

    def update_record(self, record_id, last_name = '', other_names='', email_address=''):
        db = sqlite3.connect(self.dbfilename)
        c = db.cursor()
        c.execute('UPDATE records set last_name=?, other_names=?, email_address=? \
                    WHERE record_internal_id=?', (last_name, other_names, email_address,␣
↪\
                                                  record_id))
        db.commit()
        c.close()

    def delete_record(self, record_id):
        db = sqlite3.connect(self.dbfilename)
        c = db.cursor()
        c.execute('DELETE FROM records where record_internal_id=?', (record_id,))
        db.commit()
        c.close()

    def list_all_records(self, ):
        db = sqlite3.connect(self.dbfilename)
        c = db.cursor()
        c.execute('SELECT * from records')
        records = c.fetchall()
        c.close()
        return records

    def get_record(self, record_id):
        db = sqlite3.connect(self.dbfilename)
        c = db.cursor()
        c.execute('SELECT * from records WHERE record_internal_id=?', (record_id,))
        records = c.fetchall()
        c.close()
        return records[0]
```

这个应用程序的主屏幕会是一个名字的列表. 当用户选择一个名字, 我们会想要编辑它. 我们会子类化 MultiLineAction, 然后重写 *display value* 方法来改变每个记录是怎么展示的. 我们还会重写 *actionHighlighted* 方法以需要的时候转换到编辑窗. 最后, 我们会添加两个新的按键 - 一个用来添加一个用来删除记录. 在切换到 EDITRECORDFM 之前, 我们要么会创建一个新窗口设置它的值为 None, 要么把它的值设置成我们希望编辑的记录的值.

```
class RecordList(npyscreen.MultiLineAction):
    def __init__(self, *args, **keywords):
        super(RecordList, self).__init__(*args, **keywords)
        self.add_handlers({
            "^A": self.when_add_record,
            "^D": self.when_delete_record
        })

    def display_value(self, vl):
        return "%s, %s" % (vl[1], vl[2])

    def actionHighlighted(self, act_on_this, keypress):
        self.parent.parentApp.getForm('EDITRECORDFM').value =act_on_this[0]
        self.parent.parentApp.switchForm('EDITRECORDFM')

    def when_add_record(self, *args, **keywords):
        self.parent.parentApp.getForm('EDITRECORDFM').value = None
        self.parent.parentApp.switchForm('EDITRECORDFM')

    def when_delete_record(self, *args, **keywords):
        self.parent.parentApp.myDatabase.delete_record(self.values[self.cursor_line][0])
        self.parent.update_list()
```

实际上用来显示记录列表的窗口会是一个 FormMutt 子类. 我们会把 *MAIN_WIDGET_CLASS* 类变量改成我们自己的 RecordList 控件, 然后确保每一次窗口展示给用户的时候记录列表都是最新的.

```
class RecordListDisplay(npyscreen.FormMutt):
    MAIN_WIDGET_CLASS = RecordList
    def beforeEditing(self):
        self.update_list()

    def update_list(self):
        self.wMain.values = self.parentApp.myDatabase.list_all_records()
        self.wMain.display()
```

用来编辑每个记录的窗口会是 ActionForm 的一个示例. 记录只会在用户选择了‘OK’键之后才被修改. 在窗口展示给用户之前, 每一个独立控件的值都会更新以匹配数据库的记录, 或者被清空要是创建一个新的记录的话.

```
class EditRecord(npyscreen.ActionForm):
    def create(self):
```

(下页继续)

(续上页)

```
        self.value = None
        self.wgLastName   = self.add(npyscreen.TitleText, name = "Last Name:",)
        self.wgOtherNames = self.add(npyscreen.TitleText, name = "Other Names:")
        self.wgEmail      = self.add(npyscreen.TitleText, name = "Email:")

    def beforeEditing(self):
        if self.value:
            record = self.parentApp.myDatabase.get_record(self.value)
            self.name = "Record id : %s" % record[0]
            self.record_id          = record[0]
            self.wgLastName.value   = record[1]
            self.wgOtherNames.value = record[2]
            self.wgEmail.value      = record[3]
        else:
            self.name = "New Record"
            self.record_id          = ''
            self.wgLastName.value   = ''
            self.wgOtherNames.value = ''
            self.wgEmail.value      = ''

    def on_ok(self):
        if self.record_id: # We are editing an existing record
            self.parentApp.myDatabase.update_record(self.record_id,
                                        last_name=self.wgLastName.value,
                                        other_names = self.wgOtherNames.value,
                                        email_address = self.wgEmail.value,
                                        )
        else: # We are adding a new record.
            self.parentApp.myDatabase.add_record(last_name=self.wgLastName.value,
            other_names = self.wgOtherNames.value,
            email_address = self.wgEmail.value,
            )
        self.parentApp.switchFormPrevious()

    def on_cancel(self):
        self.parentApp.switchFormPrevious()
```

最后, 我们需要一个能管理两个窗口和数据库的应用对象:

```
class AddressBookApplication(npyscreen.NPSAppManaged):
```

(下页继续)

(续上页)

```
    def onStart(self):
        self.myDatabase = AddressDatabase()
        self.addForm("MAIN", RecordListDisplay)
        self.addForm("EDITRECORDFM", EditRecord)

if __name__ == '__main__':
    myApp = AddressBookApplication()
    myApp.run()
```

CHAPTER 24

索引与一览表

- genindex
- modindex
- search

24.1 其他信息

readthedocs 原版: https://npyscreen.readthedocs.io/index.html

https://github.com/npcole/npyscreen (git 仓库)

readthedoc 中文版 https://npyscreen-doc-zh.readthedocs.io/zh_CN/latest/ 建设中…

译注使用中括号 [] 标注

npyscreen 中文仓库 https://github.com/will-mei/npyscreen_doc_zh

目前只更新文档, 代码没有更新.

说明: 之前的页面比较简陋, 校对比较随意太粗糙, 所以重新装修的新版面构建, 克隆原项目仓库后由 readthedocs 自动构建更新.

比起之前新页面算的上是 用料原装, 构建工具进口, 翻修人工校对, 另外配置了全面中文检索支持, 哈哈.

祝疫情过去, 一切安好.

索引

符号

A

B

C

D

E

F

G

K