qytz-notes Documentation

发布 *0.1*

qytz(lennyh)

2019 年 05 月 16 日

Contents

Contents:

CHAPTER 1

Python笔记

Contents:

1.1 离线环境下Python运行环境配置指南

1.1.1 安装并配置 Python 运行环境

根据需要下载对应的 Miniconda 版本并安装。

假定 Miniconda 已经安装到了 [miniconda] 目录。

创建新的Python运行环境并激活:

```
$ [miniconda]/bin/conda create -n myenv python
$ source [miniconda]/bin/activate myenv
```

在此之后即可使用此 Python 环境进行开发，安装依赖可使用 pip 工具:

```
$ pip install sqlalchemy
```

1.1.2 使用 pip wheel 打包依赖到本地

下载依赖包的 wheel 文件:

```
$ source [miniconda]/bin/activate myenv
$ pip wheel -r requirements.txt -w wheelhouse
```

默认情况下，上述命令会下载 requirements.txt 中每个包的 wheel 包到当前目录的 wheelhouse 文件夹, 包括依赖的依赖。现在你可以把这个 wheelhouse 文件夹打包到你的安装包中。

1.1.3 安装本地依赖包

首先 安装并配置 *Python* 运行环境

在你的安装脚本中执行:

```
$ source [miniconda]/bin/activate myenv
$ pip install --use-wheel --no-index --find-links=wheelhouse -r requirements.txt
```

依赖环境已经安装，现在可以在此环境运行你的程序了。

1.2 Python 3 中的异步

1.2.1 coroutine, Future, Task

Pyhton 3 的 *asyncio* 模块引入了 *coroutine Future Task* 三个新概念，要使用 *asyncio* 模块进行异步程序开发，就必须要先了解这三个概念。

- *coroutine* 译为协程，是 *asyncio* 事件循环的最小调度单位，类比于操作系统中的进程与线程，是 *asyncio* 事件循环中的最小调度单位。
- *Future* 暂时没有明确的翻译，类比于js中的 *promise* ，用于等待一个未来发生的事件，常见于 io 操作，用于封装回调函数中的异步执行部分。
- *Task* 是 *Future* 的子类，用于调度 *coroutine* 的执行。

Future 是异步操作中的核心概念，封装了一个 *coroutine* 中的异步操作，常见于 io 等待。 *coroutine* 是一个静态概念，处在未执行状态，执行状态的 *coroutine* 就是 *Task* 。

在一个事件循环中同时只调度一个 *Task* 在执行，与操作系统中的时间片调度不同， 只有一个 *Task* 需要等待 *Future* 发生时才能调度其他的 *Task* 执行。

因此 *coroutine* 中理应至少有一个需要等待的 *Future* 操作，否则无法实现异步操作，也就没有了意义。不过这并非强制性的，我们完全可以定义一个与一般函数一样的 *coroutine* 。

1.2.2 coroutine的创建

@asyncio.coroutine 或者使用新的关键字 *async def*

1.2.3 coroutine 的执行

像调用其他函数一样调用一个 *coroutine* 并不会执行，只会返回一个 *coroutine* 对象。调度这个 *coroutine* 对象的方法包括:

- 在其他协程中调用: await coroutine / yield from coroutine – await 应该优先使用。
- 使用 *ensure_future()* 或 *AbstractEventLoop.create_task()* 方法来调度协程执行

1.2.4 Future 的创建

不推荐使用 *future = asyncio.Future()* ， 优先使用 *AbstractEventLoop.create_future()* ， 事件循环的不同实现可能会实现不同的 *Future* 类。

1.2.5 concurrent.futures.Future

concurrent.futures.Future 由 *Executor.submit()* 创建。

可以使用 *asyncio.wrap_future(Future)* 将 *concurrent.futures.Future* 包装为 *asyncio.future* 。

1.2.6 异常捕获

应该在 *coroutine* 执行的地方捕获异常，在从 *Future* 获取结果的时候获得异常的详细信息， *Future.exception()* 可以获得设置给 *Future* 的异常。

1.3 Python算法笔记

Contents:

1.3.1 冒泡

冒泡排序中是计算机的一种简单的排序方法，步骤如下：

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。
3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

每一趟遍历找出最大的元素放到最后，就像每次浮出一个最大的气泡一样，故名冒泡排序。

此方法时间复杂度为O(n^2).

```
def bubble(arr):
    for i in range(len(arr)-1,0,-1):
        for j in range(0, i):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

1.3.2 快排

通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

```
#方法1
def quickSort(arr):
    less = []
    pivotList = []
    more = []
    if len(arr) <= 1:
        return arr
    else:
```

(下页继续)

(续上页)

```
        pivot = arr[0]       #将第一个值做为基准
        for i in arr:
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
            else:
                pivotList.append(i)

        less = quickSort(less)       #得到第一轮分组之后，继续将分组进行下去。
        more = quickSort(more)

        return less + pivotList + more

#方法2
# 分为<, >, = 三种情况，如果分为两种情况的话函数调用次数会增加许多，以后几个好像都有相似的问题
# 如果测试1000个100以内的整数，如果分为<, >=两种情况共调用函数1801次，分为<, >, = 三种情况，共调用
函数201次
def qsort(L):
    return (qsort([y for y in L[1:] if y <  L[0]]) + L[:1] + [y for y in L[1:] if y␣
↪== L[0] + qsort([y for y in L[1:] if y > L[0]])) if len(L) > 1 else L

#方法3
#基本思想同上，只是写法上又有所变化
def qsort(list):
    if not list:
        return []
    else:
        pivot = list[0]
        less = [x for x in list      if x <  pivot]
        more = [x for x in list[1:] if x >= pivot]
        return qsort(less) + [pivot] + qsort(more)

#方法4
from random import choice
def qSort(a):
    if len(a) <= 1:
        return a
    else:
        q = choice(a)        #基准的选择不同于前，是从数组中任意选择一个值做为基准
        return qSort([elem for elem in a if elem < q]) + [q] * a.count(q) +␣
↪qSort([elem for elem in a if elem > q])


#方法5
#这个最狠了，一句话搞定快速排序，瞠目结舌吧。
qs = lambda xs : ( (len(xs) <= 1 and [xs]) or [ qs( [x for x in xs[1:] if x < xs[0]]␣
↪) + [xs[0]] + qs( [x for x in xs[1:] if x >= xs[0]] ) ] )[0]
```

1.3.3 二分法

二分搜索每次把搜索区域减少一半，时间复杂度为 O(logn) （n代表集合中元素的个数） 在计算机科学中，二分搜索（英语：binary search），也称折半搜索（英语：half-interval search）、对数搜索（英语：logarithmic search），是一种在有序数组中查找某一特定元素的搜索算法。搜索过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜索过程结束；如果某一特定元素大于或者小于中间元素，

则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组为空，则代表找不到。这种搜索算法每一次比较都使搜索范围缩小一半。

步骤如下：

1. 确定该期间的中间位置K
2. 将查找的值T与array[k]比较。若相等，查找成功返回此位置；否则确定新的查找区域，继续二分查找。

猜数游戏。

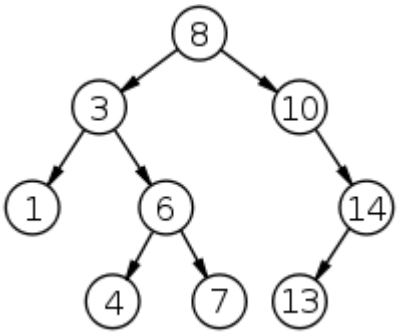
```
def BinarySearch(array,t):
    low = 0
    height = len(array)-1
    while low < height:
        mid = (low+height)/2
        if array[mid] < t:
            low = mid + 1
        elif array[mid] > t:
            height = mid - 1
        else:
            return array[mid]
    return -1
```

1.3.4 二叉树

二叉树的性质：

- 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- 任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- 任意节点的左、右子树也分别为二叉树。
- 没有键值相等的节点（no duplicate nodes）。

二叉查找树相比于其他数据结构的优势在于查找、插入的时间复杂度较低。二叉查找树是基础性数据结构，用于构建更为抽象的数据结构，如集合、multiset、关联数组等。




```
class Node:
    """
    二叉树左右枝
    """
    def __init__(self, data):
        """
        节点结构
```

(下页继续)

(续上页)

```
        """
        self.left = None
        self.right = None
        self.data = data

    def insert(self, data):
        """
        插入节点数据
        """
        if data < self.data:
            if self.left is None:
                self.left = Node(data)
            else:
                self.left.insert(data)
        elif data > self.data:
            if self.right is None:
                self.right = Node(data)
            else:
                self.right.insert(data)

    def lookup(self, data, parent=None):
        """
        二叉树查找
        """
        if data < self.data:
            if self.left is None:
                return None, None
            return self.left.lookup(data, self)
        elif data > self.data:
            if self.right is None:
                return None, None
            return self.right.lookup(data, self)
        else:
            return self, parent

    def children_count(self):
        """
        子节点个数
        """
        cnt = 0
        if self.left:
            cnt += 1
        if self.right:
            cnt += 1
        return cnt

    def delete(self, data):
        """
        删除节点
        """
        node, parent = self.lookup(data)          #已有节点
        if node is not None:
            children_count = node.children_count()       #判断子节点数
            if children_count == 0:
                # 如果该节点下没有子节点，即可删除
                if parent.left is node:
```

(下页继续)

(续上页)

```
                    parent.left = None
                else:
                    parent.right = None
                del node
            elif children_count == 1:
                # 如果有一个子节点，则让子节点上移替换该节点（该节点消失）
                if node.left:
                    n = node.left
                else:
                    n = node.right
                if parent:
                    if parent.left is node:
                        parent.left = n
                    else:
                        parent.right = n
                del node
            else:
                # 如果有两个子节点，则要判断节点下所有叶子
                parent = node
                successor = node.right
                while successor.left:
                    parent = successor
                    successor = successor.left
                node.data = successor.data
                if parent.left == successor:
                    parent.left = successor.right
                else:
                    parent.right = successor.right

    def compare_trees(self, node):
        """
        比较两棵树，比较两个二叉树的方法中，只要有一个节点（叶子）与另外一个树的不同，就返回 False，
也包括缺少对应叶子的情况。
        """
        if node is None:
            return False
        if self.data != node.data:
            return False
        res = True
        if self.left is None:
            if node.left:
                return False
        else:
            res = self.left.compare_trees(node.left)
        if res is False:
            return False
        if self.right is None:
            if node.right:
                return False
        else:
            res = self.right.compare_trees(node.right)
        return res

    #前序（pre-order，NLR）
    def preorder(node):
        if node is not None:
```

(下页继续)

(续上页)

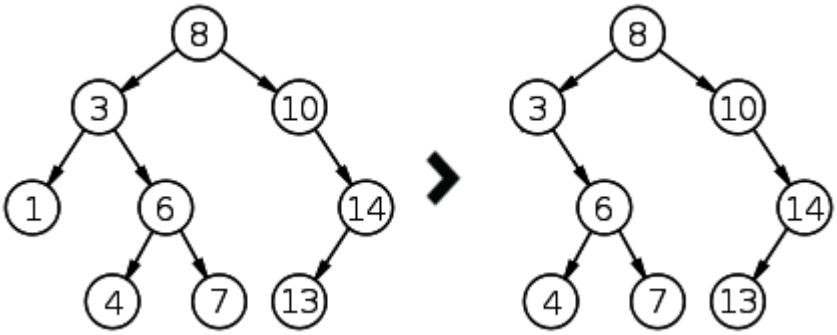
```
            print(node.data)
            preorder(node.left)
            preorder(node.right)

    #中序（in-order，LNR）
    def inorder(node):
        if node is not None:
            inorder(node.left)
            print(node.data)
            inorder(node.right)

    #后序（post-order，LRN）
    def postorder(node):
        if node is not None:
            postorder(node.left)
            postorder(node.right)
            print(node.data)


    #层序（level-order）
    def levelorder(node, more=None):
        if node is not None:
            if more is None:
                more = []
            more += [node.left, node.right]
            print node.data,
        if more:
            levelorder(more[0], more[1:])
```

二叉树删除节点示例

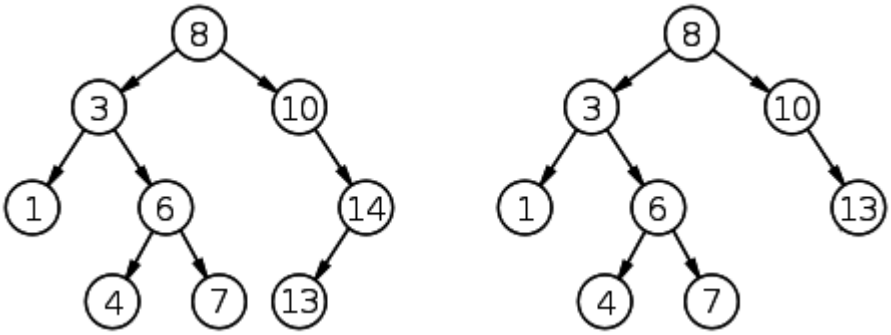
得到要删除节点下的子节点数目后，需要进行三种情况的判断

- 如果没有子节点，直接删除
- 如果有一个子节点，要将下一个子节点上移到当前节点，即替换之
- 如果有两个子节点，要对自己点的数据进行判断，并从新安排节点排序

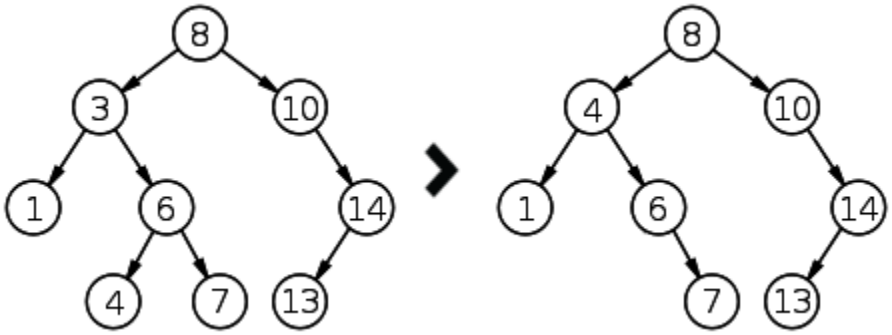
root.delete(1)



root.delete(14)



root.delete(3)



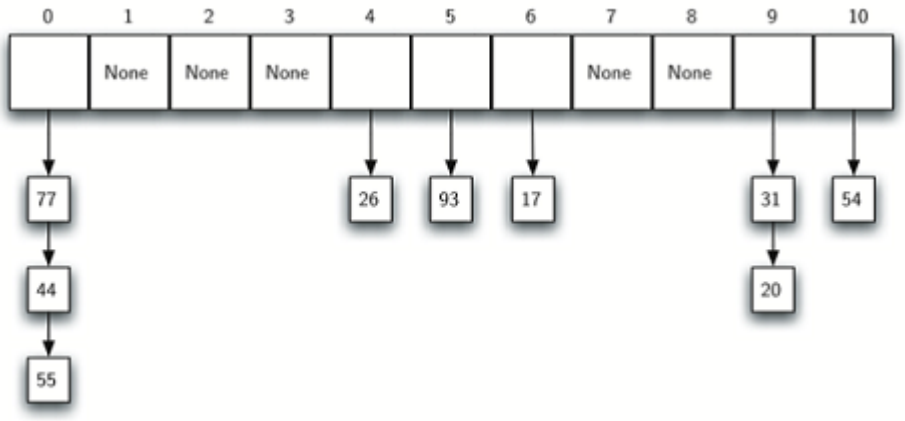
1.3.5 哈希

Hash 的定义

Hash，就是把任意长度的输入，通过哈希算法，变换成固定长度的输出，该输出就是哈希值。不同的输入可能会哈希成相同的输出，所以不可能从哈希值来唯一的确定输入值。

链式存储

原理图如下，其实就是将发生有冲突的元素放到同一位置，然后通过“指针“来串联起来



```
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [list()] * self.size
        self.data = [list()] * self.size

    def hash_function(self, key, size):
        return key % size

    def __getitem__(self, key):
        return self.get(key)

    def __setitem__(self, key, data):
        self.put(key, data)

    def put(self, key, data):
        hash_value = self.hash_function(key,len(self.slots))
        self.slots[hash_value].append(key)
        self.data[hash_value].append(data)

    def get(self, key):
        hash_value = self.hash_function(key, len(self.slots))
        for i, ikey in enumerate(self.slots[hash_value]):
            if ikey == key:
                return self.data[hash_value][i]
```

1.3.6 算法问题

水仙花数

一个N位的十进制正整数，如果它的每个位上的数字的N次方的和等于这个数本身，则称其为水仙花数。

例如：

当N=3时，153就满足条件，因为 1^3 + 5^3 + 3^3 = 153，这样的数字也被称为水仙花数（其中，“^”表示乘方，5^3表示5的3次方，也就是立方）。

当N=4时，1634满足条件，因为 1^4 + 6^4 + 3^4 + 4^4 = 1634。

当N=5时，92727满足条件。

实际上，对N的每个取值，可能有多个数字满足条件。

```
# 直接循环遍历所有N位的数进行判断，效率低
def get_flower(n):
    start, end = pow(10, n-1), pow(10, n) - 1
    for i in range(start, end):
        index1 = i // 100;
        index2 = (i % 100) // 10;
        index3 = i % 10;
        if pow(index1, 3) + pow(index2, 3) + pow(index3, 3) == i:
            print(i)

# 另外一个思路，找出N位数所包含所有数字的组合，遍历这些组合，检查所有数字的N次方之和的数字组成是否匹配这些数组。
# N位数所包含的数字组合是数学组合问题，其组合数目大大小于N位数的数目，因此算法效率高很多
```

(下页继续)

(续上页)

```
# https://oeis.org/A005188
from itertools import combinations_with_replacement
A005188_list = []
for k in range(1, 10):
    a = [i**k for i in range(10)]
    for b in combinations_with_replacement(range(10), k):
        x = sum(map(lambda y:a[y], b))
        if x > 0 and tuple(int(d) for d in sorted(str(x))) == b:
            A005188_list.append(x)
A005188_list = sorted(A005188_list) # Chai Wah Wu, Aug 25 2015

# 优化效率后的方法，少做了很多次循环
# 找出N个数字的的N次方之和落在N位数范围内的所有组合
# 遍历这些组合判断组合组成的数字能否符合水仙花数的规则
# （即不管这个组合中数字是在哪一位，只需判断N次方和数字的组成与该组合的数字组成一样就可以）
def get_flower(n):
    D_pow = [pow(i,n) for i in range(0,10)]
    V_min = pow(10,n-1)
    # V_max=sum((9*pow(10,x) for x in range(0,n)))
    V_max = pow(10, n) - 1
    T_count = 0
    print(D_pow, V_max, V_min)
    nums = [1] + [0] * (n-1)
    print('Start:', nums)
    tests = []

    idx = n - 1
    tmp_l = [0]*10
    while True:
        nums[idx] += 1
        if nums[idx] < 10:
            j = idx+1
            while j < n:
                nums[j] = nums[idx] # reset
                j += 1
            v = sum((D_pow[x] for x in nums))
            if v <= V_max and v >= V_min:
                T_count+=1
                # test if is flower
                # print('do test:', ''.join(map(str,nums)))
                N = n
                tmp_l = [0]*10
                for k in nums:
                    tmp_l[k] += 1
                while N > 0:
                    p = v % 10
                    if tmp_l[p] > 0:
                        tmp_l[p] -= 1
                        N -= 1
                    else:
                        break
                    v //= 10
                if N == 0:
                    print('hit', sum((D_pow[x] for x in nums)))
            idx = n-1
        elif idx == 0:
```

(下页继续)

(续上页)

```
            print('done')
            break
        else:
            idx -= 1
    print('t_count', T_count)
    # print('tests: ', str(tests))
```

终极解决方法是，十进制的水仙花数总共有89个:

```
# https://zh.wikipedia.org/wiki/%E6%B0%B4%E4%BB%99%E8%8A%B1%E6%95%B0
0
1
2
3
4
5
6
7
8
9
153
370
371
407
1634
8208
9474
54748
92727
93084
548834
1741725
4210818
9800817
9926315
24678050
24678051
88593477
146511208
472335975
534494836
912985153
4679307774
32164049650
32164049651
40028394225
42678290603
44708635679
49388550606
82693916578
94204591914
28116440335967
4338281769391370
4338281769391371
21897142587612075
35641594208964132
```

(下页继续)

(续上页)

```
35875699062250035
1517841543307505039
3289582984443187032
4498128791164624869
4929273885928088826
6310542598859969391 6
12846864304373139125 2
44917739914603869730 7
218876968411229162888 58
278796948930540744714 05
279078650099770525678 14
283612813213192294633 98
354525901040316919359 43
174088005938065293023 722
188451485447897896036 875
239313664430041569350 093
155047533421450153908 8894
155324216289377185066 9378
370690799595547598864 4380
370690799595547598864 4381
442209511809589961945 7938
121204998563613372405438066
121270696006801314328439376
128851796696487777842012787
174650464499531377631639254
177265453171792792366489765
146076406129719803726148730 89
190081741362542799950127347 40
190081741362542799950127347 41
238667164355239759803903692 95
114503727576549102592429205 0346
192789045714296069758063623 6639
230909268261619030750969533 8915
173335099977822493087251039 62772
186709961001538790100634132 976990
186709961001538790100634132 976991
112276328532937254159282290 0204593
126393695171037903289478072 01478392
126799377802722785663038855 94196922
121916721962543412156973580 3609966019
128157920783660599550997705 45296129367
115132219018763992565095597 973971522400
115132219018763992565095597 973971522401
```

我们把这些数存到数组，直接取出来就可以了。

回文算法

回文（Palindrome），就是一个序列（如字符串）正着读反着读是一样的。

```
def isPlidromNonRecursive(inputStr):
    strLen = len(inputStr)
    currentStart = 0
    currentEnd = strLen - 1
    while currentStart <= currentEnd:
```

(下页继续)

(续上页)

```
        if inputStr[currentStart] != inputStr[currentEnd]:
            return False
        else:
            currentStart += 1
            currentEnd -= 1
    return True

def isPlidromRecursive(inputStr, start, end):
    if len(inputStr) <= 1:
        return True
    if start >= end:
        return True
    if inputStr[start] != inputStr[end]:
        return False
    else:
        return isPlidromRecursive(inputStr, start+1, end-1)
isPlidromRecursive('abc', 0, len('abc')-1)

# 复杂度O(n)的，不过是python内部用C语言实现的，猜测会比前2个方法快。
def isPalindrome(s):
    return s == s[::-1]
```

1.3.7 正则模块

正则模块的用法

Python 正则模块的用法非常简单，总结起来就是两种用法和几个函数。

可以导入 *re* 模块后直接使用 *re.match* 和 *re.search* 进行匹配

```
In [1]: import re

In [2]: orig_str = '123abcdef'

In [3]: match_str = 'abc'

In [4]: re.match(match_str, orig_str) is None
Out[4]: True

In [5]: re.search(match_str, orig_str) is None
Out[5]: False

In [6]: match = re.search(match_str, orig_str)

In [7]: match.start()
Out[7]: 3

In [8]: match.end()
Out[8]: 6

In [9]: orig_str[match.start():match.end()]
Out[9]: 'abc'

In [10]:
```

(下页继续)

(续上页)

```
In [11]: match_str = '(abc)'

In [12]: match = re.search(match_str, orig_str)

In [13]: match.group()
Out[13]: 'abc'

In [14]: match.groups()
Out[14]: ('abc',)
```

也可可以导入 *re* 模块后使用 *re.compile* 后再进行匹配

```
In [1]: import re

In [2]: orig_str = '123abcdef'

In [3]: match_str = '(abc)'

In [4]: pattern = re.compile(match_str)

In [5]: pattern.search(orig_str)
Out[5]: <_sre.SRE_Match object; span=(3, 6), match='abc'>

In [6]: ret = pattern.search(orig_str)

In [7]: ret.start()
Out[7]: 3

In [8]: ret.end()
Out[8]: 6


In [9]:

In [9]: ret = pattern.match(orig_str)

In [10]: ret is None
Out[10]: True
```

如何写出满足需求的正则表达式

参考 Python正则表达式指南

1.3.8 参考资料

- https://github.com/qiwsir/algorithm
- python查找算法的实现-二分法
- python 下的数据结构与算法—8：哈希一下【dict与set的实现】
- 【水仙花数】Python求解水仙花数
- 水仙花数
- A005188

- python判断字符串是否是回文结构
- python 递归判断回文串
- Python正则表达式指南

CHAPTER 2

开源代码阅读笔记

Contents:

CHAPTER 3

IPython NoteBooks

Contents:

CHAPTER 4

技术笔记

Contents:

4.1 The Python GTK+ 3 Tutorial

Release 0.1

Date 2019 年 05 月 16 日

本教程介绍了如何使用Python来编写Gtk+3程序。

阅读本教程前，你最好已经掌握了Python编程语言。GUI编程相对于与标准输出（控制台/终端）交互引入了新的问题。 阅读本教程前，你需要知道如何创建并运行Python脚本文件，了解基本的解释器错误，并且了解字符串、整数、浮点数和布尔值的相关操作。 对于更多的高级窗口控件，则需要list和tuple的知识。

尽管本教程描述了GTK+3中绝大多数重要的类和方法，但本教程的意图显然不是作为API参考手册的，关于GTK+3的API的详细信息请查阅 GTK+3参考手册 。

Contents:

4.1.1 安装

在开始我们真正的编程前，需要先设置好 PyGObject 的依赖。PyGObject是一个Python模块，可以让开发者使用Python调用基于GObject的庞大类库，比如GTK+。 它只支持GTK+3及以上的版本。如果你想在你要编写的程序中使用GTK+2，请绕行使用 PyGTK 代替。

依赖

- GTK+3

- Python 2 (2.6或更高) 或 Python 3 (3.1或更高)
- gobject-introspection

预编译的二进制包

最近版本的PyGObject及其依赖包已经被几乎所有的Linux发行版打包 。因此，如果你使用Linux的话（这貌似是废话哎），你可以直接从你的发行版的官方仓库安装这些包。

从源代码安装

从源代码安装PyGObject的最简单方法就是使用 JHBuild 。其就是为了简化源码包的编译及检测 哪些依赖包要以怎样的顺序来编译而设计的。要安装JHBuild，请移步至： JHBuild manual 。 一旦你成功的安装了JHBuild，从[1] 下载最新的配置文件，并将其拷贝至JHBuild的模块目录， 重命名以 *.modules* 后缀结尾。然后将示例文件—— *sample-tarball.jhbuildrc* 拷贝至 *~/.jhbuildrc* 。 如果你完成了上述步骤，测试下你的编译环境是否可以正常运行:

```
$ jhbuild sanitycheck
```

如果一切正常，将会打印出现在你的系统中缺失的库和程序。你应该使用你发发行版的软件仓库来安装这些东西。 不同发行版的的 包名称列表 在GNOME wiki上面有维护。 完成之后再次运行该命令以确保需要的包都已经安装。执行下面的命令就可以编译PyGObject及其所有的依赖了:

```
$ jhbuild build pygobject
```

最后，你可能也会想要从源代码安装GTK+: （呃，谁会那么傻呢。。。）:

```
$ jhbuild build gtk+
```

要打开一个与JHBuild相同环境变量的shell，请执行: （真麻烦）:

```
$ jhbuild shell
```

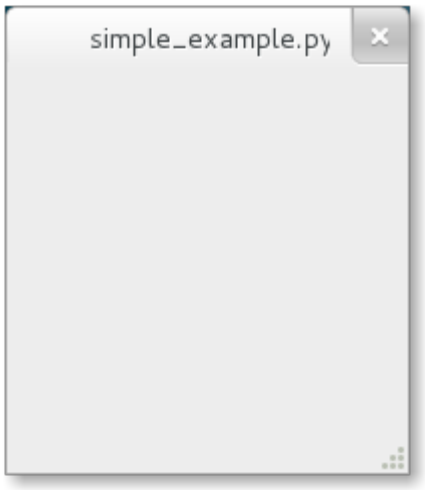
Ps：哎，这年头应该没有人真的从源代码来编译这个玩意吧，故以上代码未验证，翻译页可能不准确，请见谅。

4.1.2 从零开始

最简单的例子

好吧，我们以创建最简单的例子来开始我们的教程。运行这个程序就会创建一个空的200x200的窗口，如图：我反正是真的运行了。

[1] http://download.gnome.org/teams/releng/




```
1 #!/usr/bin/python
2 from gi.repository import Gtk
3
4 win = Gtk.Window()
5 win.connect("delete-event", Gtk.main_quit)
6 win.show_all()
7 Gtk.main()
```

现在我们开始逐行讲解这个例子。（呃，要不要这样啊。。。。）

```
#!/usr/bin/python
```

首行，以 #!开头，后面跟着你想要调用的Python解释器的路径。

```
from gi.repository import Gtk
```

要访问GTK+的类和函数，你必须首先导入Gtk模块。下一行创建了一个空的窗口。

```
win = Gtk.Window()
```

接下来连接窗口的delete-event以保证当我们点击x来关闭窗口时能够关闭这个程序。

```
win.connect("delete-event", Gtk.main_quit)
```

下一步我们显示了这个窗口。

```
win.show_all()
```

最后，我们开始了GTK+的处理循环，这个循环在我们关闭窗口时才会退出（详情见代码第五行的事件连接）。

```
Gtk.main()
```

要运行这个程序，直接在终端:

```
python simple_example.py
```

扩展的例子

让事情变得稍微有那么点儿意义吧, PyGObject版本的 “Hello World” 程序。


```
#!/usr/bin/python
from gi.repository import Gtk

class MyWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Hello World")

        self.button = Gtk.Button(label="Click Here")
        self.button.connect("clicked", self.on_button_clicked)
        self.add(self.button)

    def on_button_clicked(self, widget):
        print "Hello World"

win = MyWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

这个例子与上一个例子的不同在于我们子类化了 `Gtk.Window` 来定义我们自己的 `MyWindow` 类。

```
class MyWindow(Gtk.Window):
```

在我们的类的构造函数中我们必须要调用父类的构造函数。另外，我们告诉它设置 *title* 属性的值为 *Hello World* 。

```
        Gtk.Window.__init__(self, title="Hello World")
```

接下来的三行我们创建了一个按钮控件，连接了其 *clicked* 信号，然后将其添加为顶层窗口的孩子。

```
        self.button = Gtk.Button(label="Click Here")
        self.button.connect("clicked", self.on_button_clicked)
        self.add(self.button)
```

如上，如果你点击了这个按钮，`on_button_clicked()` 方法就会被调用。

```
    def on_button_clicked(self, widget):
        print "Hello World"
```

最后面在类外面这一段，与上面那个例子很类似，但我们没有创建:class:*Gtk.Window* 类的实例， 而是创建了我们的 `MyWindow` 类的实例。

4.1.3 基础知识

在这一章我们来介绍GTK+中最重要的方面。

主循环与信号

跟绝大多数的GUI工具包一样，GTK+采用了一个事件驱动的编程模型。当用户什么也不做时，GTK+就在主循环里等着输入信息，如果用户执行了某个动作，比如点了下鼠标，主循环马上“醒来”并将事件投递给GTK+。

当窗口部件收到一个事件时，他们通常会触发一个或多个信号。信号就会通知你的程序说 “某些有意思的事情发生，你赶紧来看看吧”， 怎么通知呢，当然是调用你你连接到信号上的函数了。这个函数就是通常所说的 回调函数了 。当你的回调函数被调用， 你一般会做些操作，比如当一个“打开”按钮“被点击时你一般会打开一个文件选择对话框。 回调函数执行结束后GTK+就返回到主循环继续等待更多的用户输入了。

通常的例子类似于这个样子：

```
handler_id = widget.connect("event", callback, data)
```

首先， *widget* 是一个我们之前创建的窗口部件的实例。 第二，event就是我们感兴趣的事件，每个窗口部件都有其自己的事件。例如，按钮的 ”clicked“ 事件，意思是按钮被按下时信号就被触发了。 第三，*callback* 参数就是回调函数的名字了，其包含相关信号被触发时要执行的代码。 最后， *data* 参数包含任何你想传递给回调函数的数据。但是，这个参数完全是可选的，如果你不需要你完全可以忽略。

这个函数返回一个信号-回调函数的id，当你想断开信号与回调函数的连接时会用到这个id。断开后将要触发及正在触发的该信号都不会再调用该回调函数了。

```
widget.disconnect(handler_id)
```

几乎所有的应用程序都会将 “delete-event” 信号与顶层窗口连接。如果用户关闭顶层窗口时该信号就会被触发。 默认的处理方法只是销毁窗口，但是并不终止程序。将 “delete-event” 与 `Gtk.main_quit()` 连接会达到想要的要求。

```
window.connect("delete-event", Gtk.main_quit)
```

调用 `Gtk.main_quit()` 使 `Gtk.main()` 中的主循环返回。

属性

属性描述了窗口不见的配置和状态信息。每一个窗口部件都有其不同的属性。 例如，一个按钮有 “label” 属性，该属性包含一个在按钮内部显示的label部件的文本。 当创建该窗口部件的实例时，你可以通过关键字参数指定这些属性的名字和值。 要创建一个标签，25度角并且右对齐的显示“Hello World”：

```
label = Gtk.Label(label="Hello World", angle=25, halign=Gtk.Align.END)
```

当然，这与下面的代码是等价的：

```
label = Gtk.Label()
label.set_label("Hello World")
label.set_angle(25)
label.set_halign(Gtk.Align.END)
```

除了使用这些get_xxx和set_xxx外你也可以使用 `widget.get_property("prop-name")` 和 `widget.set_property("prop-name", value)` 来获取和设置属性。 PS：测试了下，显示真的是25度角倾斜的文本，真的很好玩，哈哈～～

4.1.4 处理字符串相关问题

本章介绍在Python 2.x、Python 3.x及Gtk+中字符串是如何表示的，讨论当使用字符串时通常可能会遇到的问题。

字符串之深度定义

从概念上来讲，字符串是一串如 'A'、'B'、'C'或者'É' 这样的字符的列表。 **字符** 呢，其实只是抽象的代表， 其意义依赖于他们所用于的语言和上下文环境。Unicode标准定义了字符是如何有码位 （code points，觉得译为代码点太别扭了，嗯，码位貌似还行。。。）来表示的。例如上面哪些字符即是由U+0041, U+0042, U+0043, and U+00C9这些码位来表示的。码位是一个从0到0x10FFFF之间的数。

之前有提到（有吗？），把字符串作为系列字符的列表来理解很抽象。为了将这抽象而无语的表示转换成一系列的字节表示， Unicode字符串必须要被 **编码** 。最简单的ASCII编码方法是这样的:

1. 如果码位小于128，每个字节的值即与码位的值相同。
2. 如果码位大于等于128，这个Unicode字符串就不能按照ASCII来编码了。（Python在这种情况下会产生 `UnicodeEncodeError` 异常。）

尽管ASCII编码很简单，但其只能编码128个不同的字符，对我们亚洲那里够呢。 那么其中最常用的解决了这个问题的编码方法就是UTF-8了（可以处理所有Unicode码位）。 UTF代表"Unicode Transformation Format"，而 '8' 则指这种编码方式使用八位来编码。

Python 2

Python 2.x 中的Unicode支持

Python 2 中有两种不同类型的对象可以用来表示字符串： `str` 和 `unicode` 。 后者的实例用来表示Unicode字符串，而 `str` 则是字节表示（即编码后的字符串）。 Python 中表示Unicode字符串为16位或者32位的整数， 这依赖于Python解释器是怎样被编译的。Unicode字符串可以使用 `unicode.encode()` 被编码为八位的字符串:

```
>>> unicode_string = u"Fu\u00dfb\u00e4lle"
>>> print unicode_string
Fußbälle
>>> type(unicode_string)
<type 'unicode'>
>>> unicode_string.encode("utf-8")
'Fu\xc3\x9fb\xc3\xa4lle'
```

Python中的八位字符串有一个 `str.decode()` 方法可以以给定的方法来翻译字符串:

```
>>> utf8_string = unicode_string.encode("utf-8")
>>> type(utf8_string)
<type 'str'>
>>> u2 = utf8_string.decode("utf-8")
>>> unicode_string == u2
True
```

不幸的是，如果八位的字符串只包含七位的（ASCII字符）字节Python 2.x 允许你混淆 `unicode` 和 `str` ，但如果字符串包含非ASCII字符时则会产生 `UnicodeDecodeError`

```
>>> utf8_string = " sind rund"
>>> unicode_string + utf8_string
u'Fu\xdfb\xe4lle sind rund'
>>> utf8_string = " k\xc3\xb6nnten rund sein"
>>> print utf8_string
 könnten rund sein
>>> unicode_string + utf8_string
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 2: ordinal not␣
↪in range(128)
```

GTK+中的Unicode

GTK+中所有的文本都使用UTF-8编码。这意味着如果你调用一个返回字符串的方法你总是会得到一个 `str` 类型的实例。 对于需要一个或多个字符串参数的方法也是一样的，它们（指参数们）必须是UTF-8编码的。 但是为了方便，作为参数时PyGObject总是自动的将 `unicode` 实例转换 `str`:

```
>>> from gi.repository import Gtk
>>> label = Gtk.Label()
>>> unicode_string = u"Fu\u00dfb\u00e4lle"
>>> label.set_text(unicode_string)
>>> txt = label.get_text()
>>> type(txt), txt
(<type 'str'>, 'Fu\xc3\x9fb\xc3\xa4lle')
>>> txt == unicode_string
__main__:1: UnicodeWarning: Unicode equal comparison failed to convert both arguments␣
↪to Unicode - interpreting them as being unequal
False
```

注意最后的警告。尽管我们以一个unicode实例作为参数来调用 `Gtk.Label.set_text()` ， `Gtk.Label.get_text()` 总是返回一个 `str` 实例。因此， `txt` 和 `unicode_string` 并 不 相等。

如果你要使用gettext来国际化你的程序，这尤其重要。对所有语言你需要确保 gettext 会返回UTF-8编码的字符串。 通常建议不要在GTK+程序中使用 `unicode` 对象，而是只使用UTF-8编码的 `str` 对象， 因为GTK+并没有完全的整合 `unicode` 对象。 否则，你每次调用GTK+方法都要解码返回值为Unicode字符串:

```
>>> txt = label.get_text().decode("utf-8")
>>> txt == unicode_string
True
```

Python 3

Python 3.x的Unicode支持

自从Python 3.0开始，所有的字符串都以 `str` 类型的实例来存储为Unicode了。 编码后的字符串则是以二进制形式为 `bytes` 类型的实例。从概念上来讲， `str` 代表 文本 ， 而 `bytes` 则代表 数据 。使用 `str.encode()` 将 `str` 转换为 `bytes` 。 使用:meth:*bytes.decode* 将 `bytes` 转换为 `str` 。 当然，也就不能再混淆Unicode字符串和编码后的字符串了，因为这样会产生 `TypeError`:

```
>>> text = "Fu\u00dfb\u00e4lle"
>>> data = b" sind rund"
>>> text + data
```

(下页继续)

(续上页)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'bytes' object to str implicitly
>>> text + data.decode("utf-8")
'Fußbälle sind rund'
>>> text.encode("utf-8") + data
b'Fu\xc3\x9fb\xc3\xa4lle sind rund'
```

GTK+中的Unicode

结果，在Python 3.x中事情就变的非常简单了，因为如果你传递一个字符串给一个方法或一个方法返回了字符串， PyGObject会自动的编码为/解码为UTF-8。字符串或者 文本 会自动翻译为 `str` 的实例

```
>>> from gi.repository import Gtk
>>> label = Gtk.Label()
>>> text = "Fu\u00dfb\u00e4lle"
>>> label.set_text(text)
>>> txt = label.get_text()
>>> type(txt), txt
(<class 'str'>, 'Fußbälle')
>>> txt == text
True
```

References

What's new in Python 3.0 描述了能清晰的区分文本和数据的新概念。

Unicode HOWTO 讨论了Python 2.x对Unicode的支持，并解释了人们使用Unicode时可能会遇到的问题。

Unicode HOWTO for Python 3.x 讨论了Python 3.x对Unicode的支持。

UTF-8 encoding table and Unicode characters 包含了一个Unicode的码位和其UTF-8编码的对应关系。

4.1.5 布局管理

当很多GUI工具包要求你使用绝对位置来精确地摆放窗括部件时，GTK+使用了一个不同的方式。 不需要指定每一个窗口部件的位置和大小，你可以以行、列或者表格来摆放你的窗口部件。 你的窗口的大小可以自动的决定——基于其包含的窗口部件的尺寸。并且， 部件的尺寸是由其包含的文本数或者你指定的最大最小尺寸或者你指定在部件之间共享的空间大小来决定的。 你可以通过指定每一个部件的填充大空间小来设置你自己的布局。当用户操作窗口时GTK+根据这些信息可以很平滑地重新摆放和改变这些部件的大小。

GTK+使用 容器 来分层次地管理这些窗口部件。这些容器对于终端用户来是是不可见的，容器一般插入到窗口里，或者放入到其他容器里以布局组件。 有两种类型的容器：单孩子容器—— `Gtk.Bin` 的子孙和多孩子容器—— `Gtk.Container` 的子孙。 最常用的容器是水平和垂直的Box盒子(*`Gtk.Box`*)，表格(*`Gtk.Table`*)和网格(*`Gtk.Grid`*)。

Boxes盒子

Box盒子是不可见的容器，我们可以把我们的窗口控件打包近Box盒子。当将窗口控件打包进水平（horizontal）的盒子时， 这些对象水平地插入到盒子中来，根据使用 *`Gtk.Box.pack_start()`* 或者

Gtk.Box.pack_end() 可能是从左到右页可能是从右到左。而对于一个垂直的盒子，对象可能是垂直地从上到下或相反方向插入到盒子里来。 你可以使用任何组合方式的盒子（放在其他盒子中或者挨着其他盒子）来创建任何你想要达到的效果。

Box 对象

class Gtk.**Box**([*homogenous*[, *spacing*]])

Gtk.Box 的矩形区域被组织为或者是单行或者是单列的子窗口部件——这依赖于 “orientation”属性被设置为 Gtk.Orientation.HORIZONTAL 或者 Gtk.Orientation.VERTICAL 。 如果 *homogeneous* 属性为 True ， 盒子中所有的部件大小将相同，它们的大小由最大的那个部件决定。如果不设置，默认为 False 。 *spacing* 指子窗口部件之间默认的像素值。如果忽略，则不填充空白，即 *spacing* 为0。 默认子窗口部件被组织为一行，即“orientation”属性为 Gtk.Orientation.HORIZONTAL 。 *Gtk.Box* 使用了 打包（*packing*） 的概念。打包指将一个窗口部件的一个引用添加到一个 Gtk.Container 容器中。 对于一个 *Gtk.Box* ， 有两个不同的位置：盒子的开始与结尾。如果“orientation”属性设置为 Gtk.Orientation.VERTICAL ， 开始是盒子的顶端结尾是底端。如果“orientation”属性设置为 Gtk.Orientation.HORIZONTAL ， 左端是开始右端是结尾。

pack_start(*child*, *expand*, *fill*, *padding*)

添加 *child* 到盒子中，打包到其他已经打包到盒子开始的 *child* 后面。 *child* 应该是一个 *Gtk.Widget* 类型。 *expand* 参数设置为 True 允许 *child* 占用所有其可以占用的空间。如果设置为 False ， 盒子就会萎缩到与孩子控件一样的大小。 如果 *fill* 参数设置为 True ， *child* 会占用所有可用的空间，与盒子的大小相等。 只有 *expand* 设置为 True 时才有效。一个孩子总是占用垂直盒子的全部高度或者一个水平盒子的所有宽度。这个选项会影响其他的尺寸。 *padding* 是指在孩子和其“邻居”之间的额外的空白像素点，会覆盖全局的 “spacing” 设定。如果一个 *child* 控件在盒子的一个边缘， 那么填充的像素也会存在于孩子与边缘之间。

pack_end(*child*, *expand*, *fill*, *padding*)

添加一个 *child* ， 打包到盒子的末尾。会放在所有已经打包到末尾的孩子之前。

参数与 *pack_start()* 相同。

set_homogeneous(*homogeneous*)

如果 *homogeneous* 设置为 True ， 盒子中所有的空间大小相同，尺寸依赖于最大的孩子控件的尺寸。

例子

轻松一下，将上面那个有例子扩展为包含两个按钮。


```
from gi.repository import Gtk

class MyWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Hello World")

        self.box = Gtk.Box(spacing=6)
        self.add(self.box)
```

(下页继续)

(续上页)

```
10
11          self.button1 = Gtk.Button(label="Hello")
12          self.button1.connect("clicked", self.on_button1_clicked)
13          self.box.pack_start(self.button1, True, True, 0)
14
15          self.button2 = Gtk.Button(label="Goodbye")
16          self.button2.connect("clicked", self.on_button2_clicked)
17          self.box.pack_start(self.button2, True, True, 0)
18
19      def on_button1_clicked(self, widget):
20          print "Hello"
21
22      def on_button2_clicked(self, widget):
23          print "Goodbye"
24
25  win = MyWindow()
26  win.connect("delete-event", Gtk.main_quit)
27  win.show_all()
28  Gtk.main()
```

首先，我们创建了一个水平的盒子容器，并设置孩子之间有六个像素的空白填充。并将这个盒子容器设置为顶层窗口的孩子。

```
        self.box = Gtk.Box(spacing=6)
        self.add(self.box)
```

接下来我们一次向盒子容器中添加了两个不同的按钮。

```
        self.button1 = Gtk.Button(label="Hello")
        self.button1.connect("clicked", self.on_button1_clicked)
        self.box.pack_start(self.button1, True, True, 0)

        self.button2 = Gtk.Button(label="Goodbye")
        self.button2.connect("clicked", self.on_button2_clicked)
        self.box.pack_start(self.button2, True, True, 0)
```

Gtk.Box.pack_start() 添加的孩子从左到右依次放置，而 *Gtk.Box.pack_end()* 添加的孩子从右到左依次放置。

Grid网格

Gtk.Grid 是一个将其孩子控件按照行列来放置的容器，但是在构造函数中你不用指定网格的大小。使用 *Gtk.Grid.attach()* 来添加孩子。他们可以占用多行活在个多列。也可以使用 *Gtk.Grid.attach_next_to()* 来挨着一个孩子添加另一个孩子。

Gtk.Grid 如果使用 *Gtk.Grid.add()* 可以像使用 *Gtk.Box* 那样一个孩子挨着一个孩子放置——以“orientation”属性指定的方向（默认为 `Gtk.Orientation.HORIZONTAL`）。

Grid 对象

class Gtk.**Grid**

创建一个新的网格控件。

attach(*child*, *left*, *top*, *width*, *height*)

向网格中添加一个孩子。

孩子的位置根据单元格左边（*left*）和上边（*top*）单元格的书目来决定，孩子的大小（占用几个单元格）由 *width* 和 *height* 来决定。

attach_next_to(*child*, *sibling*, *side*, *width*, *height*)
挨着其 兄弟（*sibling*） 添加一个 *child*， *side* 决定添加到该兄弟的哪一边，可以是以下其中之一：

- `Gtk.PositionType.LEFT`
- `Gtk.PositionType.RIGHT`
- `Gtk.PositionType.TOP`
- `Gtk.PositionType.BOTTOM`

Width 和 *height* 决定了孩子可以占用几个单元格。

add(*widget*)
根据“orientation”决定的方向添加一个窗口部件。

Example


```
from gi.repository import Gtk

class GridWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Grid Example")

        grid = Gtk.Grid()
        self.add(grid)

        button1 = Gtk.Button(label="Button 1")
        button2 = Gtk.Button(label="Button 2")
        button3 = Gtk.Button(label="Button 3")
        button4 = Gtk.Button(label="Button 4")
        button5 = Gtk.Button(label="Button 5")
        button6 = Gtk.Button(label="Button 6")

        grid.add(button1)
        grid.attach(button2, 1, 0, 2, 1)
        grid.attach_next_to(button3, button1, Gtk.PositionType.BOTTOM, 1, 2)
        grid.attach_next_to(button4, button3, Gtk.PositionType.RIGHT, 2, 1)
        grid.attach(button5, 1, 2, 1, 1)
        grid.attach_next_to(button6, button5, Gtk.PositionType.RIGHT, 1, 1)

win = GridWindow()
win.connect("delete-event", Gtk.main_quit)
```

(下页继续)

(续上页)

```
27 win.show_all()
28 Gtk.main()
```

Table

表格允许我们以类似于 *Gtk.Table* 的方式放置控件。 由于此接口是过时的，最好不再使用，因此不再翻译，请使用 *Gtk.Grid* 来代替。

Gtk.Table.set_row_spacing() and *Gtk.Table.set_col_spacing()* set the spacing between the rows at the specified row or column. Note that for columns, the space goes to the right of the column, and for rows, the space goes below the row.

You can also set a consistent spacing for all rows and/or columns with *Gtk.Table.set_row_spacings()* and *Gtk.Table.set_col_spacings()*. Note that with these calls, the last row and last column do not get any spacing.

Table Objects

3.4 版后已移除: Use *Gtk.Grid* instead.

class Gtk.**Table**(*rows*, *columns*, *homogeneous*)

The first argument is the number of rows to make in the table, while the second, obviously, is the number of columns. If *homogeneous* is `True`, the table cells will all be the same size (the size of the largest widget in the table).

attach(*child*, *left_attach*, *right_attach*, *top_attach*, *bottom_attach*[, *xoptions*[, *yoptions*[, *xpadding*[, *ypadding*]]]])

Adds a widget to a table.

child is the widget that should be added to the table. The number of 'cells' that a widget will occupy is specified by *left_attach*, *right_attach*, *top_attach* and *bottom_attach*. These each represent the leftmost, rightmost, uppermost and lowest column and row numbers of the table. (Columns and rows are indexed from zero).

For example, if you want a button in the lower-right cell of a 2 x 2 table, and want it to occupy that cell only, then the code looks like the following.

```
button = Gtk.Button()
table = Gtk.Table(2, 2, True)
table.attach(button, 1, 2, 1, 2)
```

If, on the other hand, you wanted a widget to take up the entire top row of our 2 x 2 table, you'd use

```
table.attach(button, 0, 2, 0, 1)
```

xoptions and *yoptions* are used to specify packing options and may be bitwise ORed together to allow multiple options. These options are:

- `Gtk.AttachOptions.EXPAND`: The widget should expand to take up any extra space in its container that has been allocated.
- `Gtk.AttachOptions.FILL`: The widget will expand to use all the room available.
- `Gtk.AttachOptions.SHRINK`: Reduce size allocated to the widget to prevent it from moving off screen.

If omitted, *xoptions* and *yoptions* defaults to `Gtk.AttachOptions.EXPAND | Gtk.AttachOptions.FILL`.

Finally, the padding arguments work just as they do for *Gtk.Box.pack_start()*. If omitted, *xpadding* and *ypadding* defaults to `0`.

set_row_spacing(*row*, *spacing*)
Changes the space between a given table row and the subsequent row.

set_col_spacing(*col*, *spacing*)
Alters the amount of space between a given table column and the following column.

set_row_spacings(*spacing*)
Sets the space between every row in this table equal to *spacing*.

set_col_spacings(*spacing*)
Sets the space between every column in this table equal to *spacing*.

Example


```
from gi.repository import Gtk

class TableWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Table Example")

        table = Gtk.Table(3, 3, True)
        self.add(table)

        button1 = Gtk.Button(label="Button 1")
        button2 = Gtk.Button(label="Button 2")
        button3 = Gtk.Button(label="Button 3")
        button4 = Gtk.Button(label="Button 4")
        button5 = Gtk.Button(label="Button 5")
        button6 = Gtk.Button(label="Button 6")

        table.attach(button1, 0, 1, 0, 1)
        table.attach(button2, 1, 3, 0, 1)
        table.attach(button3, 0, 1, 1, 3)
        table.attach(button4, 1, 3, 1, 2)
        table.attach(button5, 1, 2, 2, 3)
        table.attach(button6, 2, 3, 2, 3)

win = TableWindow()
win.connect("delete-event", Gtk.main_quit)
```

(下页继续)

(续上页)

```
27 win.show_all()
28 Gtk.main()
```

4.1.6 标签Label

标签是在窗口中显示不可编辑文本的主要方法，例如要在 *Gtk.Entry* 控件旁边放置一个标题。你可以在构造函数中指定文本的内容，也可以在后面调用 `Gtk.Label.set_text()` 或 `Gtk.Label.set_markup()` 方法设置。

标签的宽度会自动调整。你可以把换行符（“\n”）放在标签内容中来产生一个多行的标签。

标签可以通过 `Gtk.Label.set_selectable()` 设置为可以选择。可选择的标签允许用户拷贝标签的内容到剪贴板。一般只有包含需要拷贝的内容——如错误信息时才设置标签为可选择的。

标签文本可以通过 `Gtk.Label.set_justify()` 方法来设置对齐方式。当然你可以自动换行——只要调用 `Gtk.Label.set_line_wrap()` 。

Gtk.Label 支持一种简单的格式，例如允许你使某些文本显示为粗体、某种颜色或者更大号。你可以通过使用 `Gtk.Label.set_markup()` 设置标签内容来实现这些效果，这个函数使用Pango标记语言的语法[1]。例如，`<b>bold text</b> and <s>strikethrough text</s>`。另外，*Gtk.Label* 也支持可以点开的超链接。超链接的标记借用了HTML的方法，使用带有href和taitle属性的a标记。GTK+呈现的超链接与浏览器中很相似——带有颜色和下划线。Title作为该链接的提示信息。

```
label.set_markup("Go to <a href=\"http://www.gtk.org\" title=\"Our website\">GTK+
↪website</a> for more")
```

标签也可以包含 助记符*(mnemonics)*，助记符是标签中带有下划线的字符，用户键盘导航（呃，其实就是快捷键啦）。助记符通过在助记符字符前添加下划线来实现，例如 “_File”，需要调用 *Gtk.Label.new_with_mnemonic()* 或者 `Gtk.Label.set_text_with_mnemonic()` 。助记符会自动计划包含该标签的控件。例如 *Gtk.Button*； 如果标签不在助记符的目标控件中，你需要通过 `Gtk.Label.set_mnemonic_widget()` 手动设置。

Label Objects

class Gtk.**Label**([*text*])

创建一个内容为 *text* 的标签，如果 *text* 忽略，则会创建一个空标签。

static new_with_mnemonic(*text*)

创建了一个新的标签，内容为 *text* 。

如果 *text* 里的字符前面加一下划线，那这些字符即被强调。如果你需要一个下划线字符就要用‘__’。第一个下划线代助记符的按键可以用来激活另一个窗口控件，这个窗口控件是自动选择的，或者你也可以调用 `Gtk.Label.set_mnemonic_widget()` 来修改。

如果没有调用 `Gtk.Label.set_mnemonic_widget()` ，则 *Gtk.Label* 标签的第一个可被激活的祖先会被选择作为助记符控件。例如，如果标签是在一个按钮活在个一个菜单项里，那么按钮或菜单项就会自动成为助记符控件并且会自动被助记符激活。

set_justify(*justification*)

设置标签内的文本行相对于其他行的对齐方式， *justification* 可以设置为 `Gtk.Justification.LEFT`, `Gtk.Justification.RIGHT`, `Gtk.Justification.CENTER`, `Gtk.Justification.FILL` 之一。对于包含单行文本的标签无效。

[1] Pango Markup Syntax, http://developer.gnome.org/pango/stable/PangoMarkupFormat.html

set_line_wrap(*wrap*)

如果 *wrap* 为 `True`，如果文本超过控件的大小时会自动折行。如果设置为 `False`，若超过控件大小，文本会根据控件的边缘被截断。

set_markup(*markup*)

根据Pango标记语言解析 *markup* 标记后的文本设置标签内容。标记必须有效，例如，<, >, &这些字符必须写作< > 和&amp。

set_mnemonic_widget(*widget*)

如果标签设置了带有助记符的内容，此方法设置与助记符关联的目标控件。

set_selectable(*selectable*)

设置使用允许用户可以选择标签内的文本，用于复制粘贴。

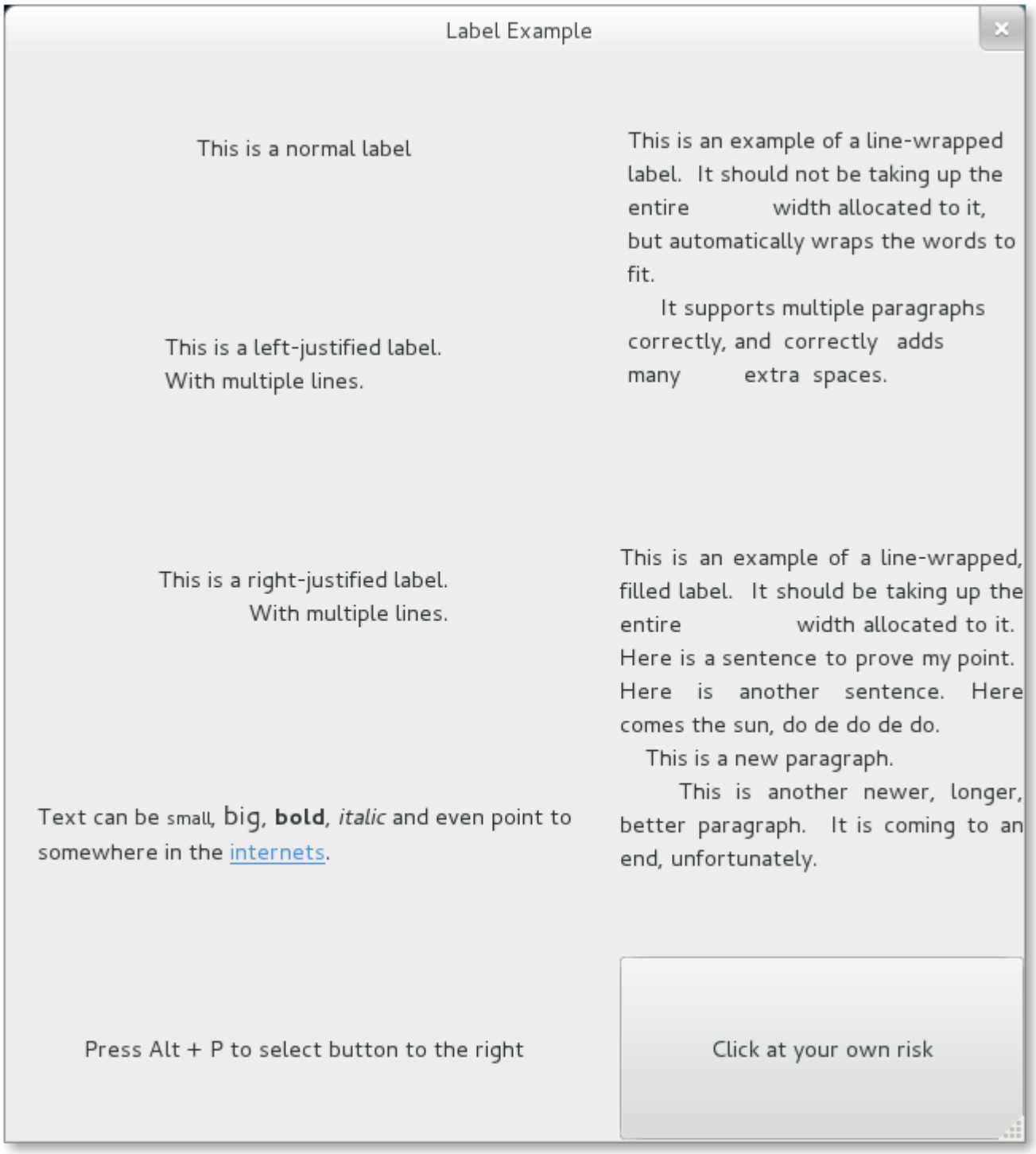
set_text(*text*)

设置控件内的文本，会覆盖之前设置的文本。

set_text_with_mnemonic(*text*)

参照 `new_with_mnemonic()` 。

Example




```
from gi.repository import Gtk

class LabelWindow(Gtk.Window):

```

(下页继续)

(续上页)

```
    def __init__(self):
        Gtk.Window.__init__(self, title="Label Example")

        hbox = Gtk.Box(spacing=10)
        hbox.set_homogeneous(False)
        vbox_left = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=10)
        vbox_left.set_homogeneous(False)
        vbox_right = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=10)
        vbox_right.set_homogeneous(False)

        hbox.pack_start(vbox_left, True, True, 0)
        hbox.pack_start(vbox_right, True, True, 0)

        label = Gtk.Label("This is a normal label")
        vbox_left.pack_start(label, True, True, 0)

        label = Gtk.Label()
        label.set_text("This is a left-justified label.\nWith multiple lines.")
        label.set_justify(Gtk.Justification.LEFT)
        vbox_left.pack_start(label, True, True, 0)

        label = Gtk.Label("This is a right-justified label.\nWith multiple lines.")
        label.set_justify(Gtk.Justification.RIGHT)
        vbox_left.pack_start(label, True, True, 0)

        label = Gtk.Label("This is an example of a line-wrapped label.  It "
                          "should not be taking up the entire              "
                          "width allocated to it, but automatically "
                          "wraps the words to fit.\n"
                          "     It supports multiple paragraphs correctly, "
                          "and  correctly   adds "
                          "many          extra  spaces. ")
        label.set_line_wrap(True)
        vbox_right.pack_start(label, True, True, 0)

        label = Gtk.Label("This is an example of a line-wrapped, filled label.  "
                          "It should be taking "
                          "up the entire              width allocated to it.  "
                          "Here is a sentence to prove "
                          "my point.  Here is another sentence. "
                          "Here comes the sun, do de do de do.\n"
                          "    This is a new paragraph.\n"
                          "    This is another newer, longer, better "
                          "paragraph.  It is coming to an end, "
                          "unfortunately.")
        label.set_line_wrap(True)
        label.set_justify(Gtk.Justification.FILL)
        vbox_right.pack_start(label, True, True, 0)

        label = Gtk.Label()
        label.set_markup("Text can be <small>small</small>, <big>big</big>, "
                         "<b>bold</b>, <i>italic</i> and even point to somewhere "
                         "in the <a href=\"http://www.gtk.org\" "
                         "title=\"Click to find out more\">internets</a>.")
        label.set_line_wrap(True)
        vbox_left.pack_start(label, True, True, 0)
```

(下页继续)

(续上页)

```
61
62          label = Gtk.Label.new_with_mnemonic("_Press Alt + P to select button to the␣
    ↪right")
63          vbox_left.pack_start(label, True, True, 0)
64          label.set_selectable(True)
65
66          button = Gtk.Button(label="Click at your own risk")
67          label.set_mnemonic_widget(button)
68          vbox_right.pack_start(button, True, True, 0)
69
70          self.add(hbox)
71
72  window = LabelWindow()
73  window.connect("delete-event", Gtk.main_quit)
74  window.show_all()
75  Gtk.main()
```

4.1.7 Entry输入框

输入框控件允许用户输入文本，你可以使用 *Gtk.Entry.set_text()* 方法来改变输入框的内容。使用 *Gtk.Entry.get_text()* 来获取输入框当前的内容。你也可以使用 *Gtk.Entry.set_max_length()* 限制输入框可以输入的最大文本数。

有时候你想要设置输入框只读，可以通过传递 `False` 给方法 *Gtk.Entry.set_editable()* 。

输入框也可以用来从用户获取密码，通常要隐藏用户的输入以避免输入被地丧恶人看见，调用 *Gtk.Entry.set_visibility()* 传递 `False` 可以使输入的文本被隐藏。

Gtk.Entry 也可以在文本的后面显示进度或者活动的信息，类似于 *Gtk.ProgressBar* 控件，用在浏览器中显示下载的进度。要使输入框显示这样的信息，需要调用 *Gtk.Entry.set_progress_fraction()* , *Gtk.Entry.set_progress_pulse_step()* 或 *Gtk.Entry.progress_pulse()* 。

另外，输入框也可以在前面或者后面显示图标，这些图标可以通过点击激活，可以设置为拖拽源，也可以提示信息。要添加这样的图标，可以调用 *Gtk.Entry.set_icon_from_stock()* 或者任何一个 从图标的名字、pixbuf或icon主题中设置图标的变种函数。要设置图标的提示信息，使用 *Gtk.Entry.set_icon_tooltip_text()* 或者相应的函数。

Entry 输入框对象

class Gtk.**Entry**

get_text()
获取输入框控件的内容。

set_text(*text*)
设置控件的文本，会替换原来的内容。

set_visibility(*visible*)
设置输入框的内容是否可见。当 *visible* 为 `False` 时， 字符显示为不可见的字符——即使是从其他地方复制过来的。

set_max_length(*max*)
设置允许输入文本的最大长度。如果当前的内容比设置的长度常，则文本会被截断。

set_editable(*is_editable*)
设置用户可否比啊及输入框中的内容。如果 *is_editable* 为 `True`，用户可以编辑文本。

set_progress_fraction(*fraction*)
设置进度条指针填充到的进度，设置的值必须介于0.0和1.0之间，包括0和1。

set_progress_pulse_step(*fraction*)
设置每次调用 *progress_pulse()* 时进度反弹块移动的宽度占输入框总宽度的百分比。

progress_pulse()
一些进度向前走了，但是你不知道前进了多少，调用词函数使输入框的进度指示针进入活动状态， 这样会有一个反弹块前后移动。每次调用 *progress_pulse()* 会使反弹块移动一点 （移动的多少由 *set_progress_pulse_step()* 来决定）。

set_icon_from_stock(*icon_pos*, *stock_id*)
设置在输入框的特定位置显示图标，图标 *stock_id* 参考 *stock item*。 如果 *stock_id* 为 `None`， 则不会显示图标。

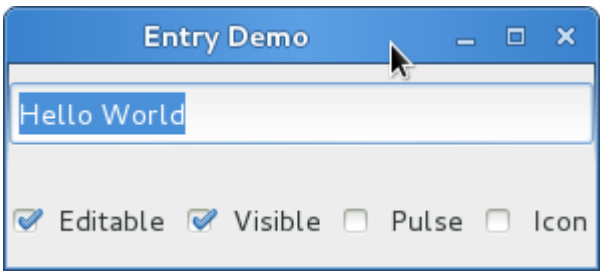
icon_pos 指定了图标放在输入框的哪一侧，可能的值有：

- `Gtk.EntryIconPosition.PRIMARY`: 在输入框的开始（要根据文本的方向）。
- `Gtk.EntryIconPosition.SECONDARY`: 在输入框的结尾（要根据文本的方向）。

set_icon_tooltip_text(*icon_pos*, *tooltip*)
设置 *tooltip* 的内容作为指定位置的图标的提示信息。如果 *tooltip* 为 `None`， 那么之前设置的提示信息被移除。

icon_pos 允许的值参见 *set_icon_from_stock()*。

Example




```
from gi.repository import Gtk, GObject

class EntryWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Entry Demo")
        self.set_size_request(200, 100)

        self.timeout_id = None

        vbox = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=6)
        self.add(vbox)

        self.entry = Gtk.Entry()
        self.entry.set_text("Hello World")
        vbox.pack_start(self.entry, True, True, 0)

```

(下页继续)

(续上页)

```
        hbox = Gtk.Box(spacing=6)
        vbox.pack_start(hbox, True, True, 0)

        self.check_editable = Gtk.CheckButton("Editable")
        self.check_editable.connect("toggled", self.on_editable_toggled)
        self.check_editable.set_active(True)
        hbox.pack_start(self.check_editable, True, True, 0)

        self.check_visible = Gtk.CheckButton("Visible")
        self.check_visible.connect("toggled", self.on_visible_toggled)
        self.check_visible.set_active(True)
        hbox.pack_start(self.check_visible, True, True, 0)

        self.pulse = Gtk.CheckButton("Pulse")
        self.pulse.connect("toggled", self.on_pulse_toggled)
        self.pulse.set_active(False)
        hbox.pack_start(self.pulse, True, True, 0)

        self.icon = Gtk.CheckButton("Icon")
        self.icon.connect("toggled", self.on_icon_toggled)
        self.icon.set_active(False)
        hbox.pack_start(self.icon, True, True, 0)

    def on_editable_toggled(self, button):
        value = button.get_active()
        self.entry.set_editable(value)

    def on_visible_toggled(self, button):
        value = button.get_active()
        self.entry.set_visibility(value)

    def on_pulse_toggled(self, button):
        if button.get_active():
            # self.entry.set_progress_fraction(0.6)
            self.entry.set_progress_pulse_step(0.2)
            # Call self.do_pulse every 100 ms
            self.timeout_id = GObject.timeout_add(100, self.do_pulse, None)
        else:
            # Don't call self.do_pulse anymore
            GObject.source_remove(self.timeout_id)
            self.timeout_id = None
            self.entry.set_progress_pulse_step(0)

    def do_pulse(self, user_data):
        self.entry.progress_pulse()
        return True

    def on_icon_toggled(self, button):
        if button.get_active():
            stock_id = Gtk.STOCK_FIND
        else:
            stock_id = None
        self.entry.set_icon_from_stock(Gtk.EntryIconPosition.PRIMARY,
            stock_id)

win = EntryWindow()
```

(下页继续)

(续上页)

```
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

4.1.8 按钮控件

按钮

按钮控件是另一个经常使用的控件。按钮通常会添加一个当点击按钮时要调用的函数。

Gtk.Button 按钮控件可以包含任何有效的子控件，即可以包含绝大多数的其他标准的 *Gtk.Widget* 控件。最常会添加的子控件是 *Gtk.Label* 。

通常，你想要连接一个按钮的 “clicked” 信号，该信号当你按下并释放鼠标按钮时会触发。

按钮对象

class Gtk.**Button**([*label*[, *stock*[, *use_underline*]]])

如果lable不是 None ， 会创建一个带有 *Gtk.Label* 的 *Gtk.Button* 按钮， 标签会包含给定的文本。

如果 *stock* 不为 None ， 创建的按钮包含 *stock item* 的图像和文本。

如果 *use_underline* 为 True, 则label中的下划线后面的字符为助记符加速键。

set_label(*label*)

设置按钮标签的内容为 *label* 。

set_use_underline(*use_underline*)

如果为 *True* ， 按钮标签文本中的下划线预示着下一个字符用于助记符加速键。

例子


```
#!/usr/bin/eny python
#coding:utf-8

from  gi.repository import Gtk

class ButtonWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='Button Demo')
        self.set_border_width(10)

        hbox = Gtk.Box(spacing=6)
        self.add(hbox)
```

(下页继续)

(续上页)

```
14
15          button = Gtk.Button('Click Me')
16          button.connect('clicked', self.on_click_me_clicked)
17          hbox.pack_start(button, True, True, 0)
18
19          button = Gtk.Button(stock=Gtk.STOCK_OPEN)
20          button.connect('clicked', self.on_open_clicked)
21          hbox.pack_start(button, True, True, 0)
22
23          button = Gtk.Button('_Close', use_underline=True)
24          button.connect('clicked', self.on_close_clicked)
25          hbox.pack_start(button, True, True, 0)
26
27      def on_click_me_clicked(self, button):
28          print '"click me" button was clicked'
29
30      def on_open_clicked(self, button):
31          print '"open" button was clicked'
32
33      def on_close_clicked(self, button):
34          print 'Closing application'
35          Gtk.main_quit()
36
37  wind = ButtonWindow()
38  wind.connect('delete-event', Gtk.main_quit)
39  wind.show_all()
40  Gtk.main()
```

ToggleButton

Gtk.ToggleButton 与 *Gtk.Button* 非常类似，但是当点击后， Toggle按钮保持激活状态，知道再次点击。当按钮的状态改变时， “toggled” 信号会被触发。

要获得 *Gtk.ToggleButton* 的状态，我们可以调用 *Gtk.ToggleButton.get_active()* 方法，如果Toggle按钮处于按下状态，函数返回 `True` 。 当然你也可以设置Toggle按钮的状态——通过 *Gtk.ToggleButton.set_active()* 方法。 如果你这样做了，并且Toggle按钮的状态变了，那么 “toggle” 信号会被触发。

ToggleButton 对象

class Gtk.**ToggleButton**([*label*[, *stock*[, *use_underline*]]])

参数与 *Gtk.Button* 的构造函数一样。

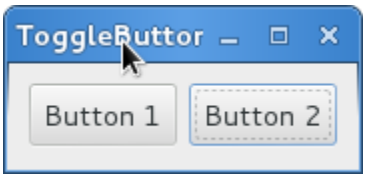
get_active()

返回Toggle按钮当前的状态。如果按钮处于按下状态返回 `True` ， 否则返回 `False` 。

set_active(*is_active*)

设置Toggle按钮的状态，如果想设置按钮为按下状态则传递 `True` ， 否则传递‘‘False‘‘ 。 会导致 “toggle” 信号被触发。

例子




```
#!/usr/bin/eny python
#coding:utf-8

from gi.repository import Gtk

class ToggleButtonWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="ToggleButton Demo")
        self.set_border_width(10)

        hbox = Gtk.Box(spacing=6)
        self.add(hbox)

        button = Gtk.ToggleButton("Button 1")
        button.connect('toggled', self.on_button_toggled, '1')
        hbox.pack_start(button, True, True, 0)

        button = Gtk.ToggleButton("B_utton 2", use_underline=True)
        button.set_active(True)
        button.connect('toggled', self.on_button_toggled, '2')
        hbox.pack_start(button, True, True, 0)

    def on_button_toggled(self, button, name):
        if button.get_active():
            state = 'on'
        else:
            state = 'off'
        print 'Button', name, 'was tunned', state

wind = ToggleButtonWindow()
wind.connect('delete-event', Gtk.main_quit)
wind.show_all()
Gtk.main()
```

CheckButton（复选按钮）

Gtk.CheckButton 继承自 *Gtk.ToggleButton* 。唯一的不同是 *Gtk.CheckButton* 外观。 *Gtk.CheckButton* 会在 *Gtk.ToggleButton* 的旁边放置一个分离的控件——通常是一个 *Gtk.Label* 。“toggled” 信号， *Gtk.ToggleButton.set_active()* 与 *Gtk.ToggleButton.get_active()* 则继承过来了。

CheckButton 对象

class Gtk.**CheckButton**([*label*[, *stock*[, *use_underline*]]])

参数与 *Gtk.Button* 同。

RadioButton（单选按钮）

与复选按钮一样，单选按钮也是继承自 *Gtk.ToggleButton*，但是单选按钮 按照组的方式来工作，并且在组中只有一个 *Gtk.RadioButton* 可以被选中。 因此，*Gtk.RadioButton* 是一种让用户从很多选项中选择一个的方法。

单选按钮Radio buttons可以使用以下任何一个静态函数创建： *Gtk.RadioButton.new_from_widget()* ， *Gtk.RadioButton.new_with_label_from_widget()* 或者 *Gtk.RadioButton.new_with_mnemonic_from_widget()* 。 一个组中第一个radio button创建时 *group* 参数传递 None，在随后的调用中， 你想要将此按钮加入的组应该作为参数传递。

当第一次运行时，组内的第一个radio按钮会是激活状态的。可以通过 *Gtk.ToggleButton.set_active()* 传递 True 来修改。

在创建后改变 *Gtk.RadioButton* 控件的分组信息可以通过调用 *Gtk.RadioButton.join_group()* 来实现。

RadioButton 对象

class Gtk.**RadioButton**

static new_from_widget(*group*)
创建一个新的 *Gtk.RadioButton*，将其添加到与 *group* 控件同一组中。 如果 *group* 为 None，会创建一个新的组。

static new_with_label_from_widget(*group*, *label*)
创建一个 *Gtk.RadioButton* 。标签控件内的文本会被设置为 *lable* 。 *group* 参数与 *new_from_widget()* 相同。

static new_with_mnemonic_from_widget(*group*, *label*)
与 *new_with_label_from_widget()* 相同，但是 *label* 中的下划线会被解析为按钮的助记符。

join_group(*group*)
将radio button加入到另一个 *Gtk.RadioButton* 对象的组中。

Example


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

from gi.repository import Gtk

class RadioButtonWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="RadioButton Demo")
        self.set_border_width(10)
```

(下页继续)

(续上页)

```
11
12          hbox = Gtk.Box(spacing=6)
13          self.add(hbox)
14
15          button1 = Gtk.RadioButton.new_with_label_from_widget(None, "Button 1")
16          button1.connect('toggled', self.on_button_toggled, '1')
17          hbox.pack_start(button1, False, False, 0)
18
19          button2 = Gtk.RadioButton.new_from_widget(button1)
20          button2.set_label('Button 2')
21          button2.connect('toggled', self.on_button_toggled, '2')
22          hbox.pack_start(button2, False, False, 0)
23
24          button3 = Gtk.RadioButton.new_with_mnemonic_from_widget(button1, "B_utton 3")
25          button3.connect('toggled', self.on_button_toggled, '3')
26          hbox.pack_start(button3, False, False, 0)
27
28      def on_button_toggled(self, button, name):
29          if button.get_active():
30              state = 'on'
31          else:
32              state = 'off'
33          print 'Button', name, 'was turned', state
34
35  wind = RadioButtonWindow()
36  wind.connect('delete-event', Gtk.main_quit)
37  wind.show_all()
38  Gtk.main()
```

LinkButton

Gtk.LinkButton 是带有链接的 *Gtk.Button* 。与浏览器中使用的链接类似—— 当点击时会触发一个动作，当要快速的链接到一个资源时很有用。

绑定到 *Gtk.LinkButton* 的URI可以通过 *Gtk.LinkButton.set_uri()* 来设置，可以通过 *Gtk.LinkButton.get_uri()* 来获取绑定的URI。

LinkButton 对象

class Gtk.**LinkButton**(*uri*[, *label*])

uri 是需要加载的网页的地址。label是显示的文本。如果label为 None 或者忽略，则显示网址。

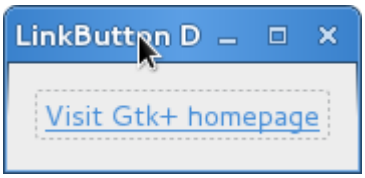
get_uri()

获取 *set_uri()* 设置的URI。

set_uri(*uri*)

设置按钮指向的 *uri* 地址。作为副作用，会取消按钮的 'visited' 状态。

Example




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

from gi.repository import Gtk

class LinkButtonWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='LinkButton Demo')
        self.set_border_width(10)

        button = Gtk.LinkButton('http://www.gtk.org', 'Visit Gtk+ homepage')
        self.add(button)

wind = LinkButtonWindow()
wind.connect('delete-event', Gtk.main_quit)
wind.show_all()
Gtk.main()
```

SpinButton

Gtk.SpinButton 是一种让用户设置某些属性的值的完美方法。*Gtk.SpinButton* 不是让用户直接在 *Gtk.Entry* 中输入一个数字，而是提供两个箭头让用户增加或减小显示的值。值也可以直接输入，可以附加检查以保证 输入的值在要求的范围内。*Gtk.SpinButton* 的主要属性通过 *Gtk.Adjustment* 来设置。

要改变 *Gtk.SpinButton* 显示的值，使用 *Gtk.SpinButton.set_value()* 。通过:meth:*Gtk.SpinButton.get_value* 或者 *Gtk.SpinButton.get_value_as_int()* 获取按钮的值——根据你的要求可以是整数或浮点值。

当spin button显示浮点数时，你可以通过 *Gtk.SpinButton.set_digits()* 调整显示的浮点数的位数。

默认情况下，*Gtk.SpinButton* 接受文本数据。如果你想限制其为数值，可以调用 *Gtk.SpinButton.set_numeric()* ，并传递 True 。

我们也可以设置 *Gtk.SpinButton* 显示的更新策略。有两种可选: 默认是即使输入的数据不合法也会显示； 我们也可以设置为只有输入的值正确时才需要更新——通过调用 *Gtk.SpinButton.set_update_policy()* 。

SpinButton 对象

class Gtk.**SpinButton**

set_adjustment(*adjustment*)
替换与该spin button关联的 *Gtk.Adjustment* 。

set_digits(*digits*)
设置spin button显示的精度——最高可以支持20个数字。

set_increments(*step*, *page*)
设置按钮值增加的 step 和 page 。这会影响当按钮的箭头按下时值的变化速度。 step是按下上下键改变的值大小，page则是指按下page up/down是改变的值大小。

set_value(*value*)
设置按钮的值。

get_value()
返回按钮的值——浮点数类型。

get_value_as_int()
获取按钮的值——整型。

set_numeric(*numeric*)
如果 *numeric* 为 `False` ，非数字的文本可以输入给spin button，否则只允许输入数值。

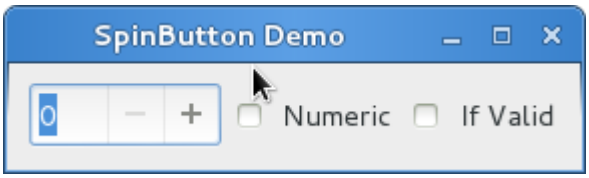
set_update_policy(*policy*)
设置按钮的更新行为。这决定了按钮的值是总会更新还是只有值合法时才会更新。 *policy* 参数的值可以是 `Gtk.SpinButtonUpdatePolicy.ALWAYS` 或者 `Gtk.SpinButtonUpdatePolicy.IF_VALID` 。

Adjustment 对象

class `Gtk.`**Adjustment**(*value*, *lower*, *upper*, *step_increment*, *page_increment*, *page_size*)
`Gtk.Adjustment` 对象代表一个有最大与最小界限的值，也包含每次增加的 step和pagement。这在一些Gtk+窗口控件中使用，包括 *`Gtk.SpinButton`* ， `Gtk.Viewport` 和 `Gtk.Range` 。

value 为初始值， *lower* 为最小值， *upper* 为最大值， *step_increment* 为每次up/down键增加/减小的值，*page_increment* 是 按下page up/down键改变的值大小，而 *page_size* 代表页大小。

Example




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

from gi.repository import Gtk

class SpinButtonWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='SpinButton Demo')
        self.set_border_width(10)

        hbox = Gtk.Box(spacing=6)
        self.add(hbox)

```

(下页继续)

(续上页)

```
        adjustment = Gtk.Adjustment(0, 0, 100, 1, 10, 0)
        self.spinbutton = Gtk.SpinButton()
        self.spinbutton.set_adjustment(adjustment)
        hbox.pack_start(self.spinbutton, False, False, 0)

        check_numeric = Gtk.CheckButton("Numeric")
        check_numeric.connect('toggled', self.on_numeric_toggled)
        hbox.pack_start(check_numeric, False, False, 0)

        check_ifvalid = Gtk.CheckButton('If Valid')
        check_ifvalid.connect('toggled', self.on_ifvalid_toggled)
        hbox.pack_start(check_ifvalid, False, False, 0)

    def on_numeric_toggled(self, button):
        self.spinbutton.set_numeric(button.get_active())

    def on_ifvalid_toggled(self, button):
        if button.get_active():
            policy = Gtk.SpinButtonUpdatePolicy.IF_VALID
        else:
            policy = Gtk.SpinButtonUpdatePolicy.ALWAYS
        self.spinbutton.set_update_policy(policy)

win = SpinButtonWindow()
win.connect('delete-event', Gtk.main_quit)
win.show_all()
Gtk.main()
```

4.1.9 进度条

Gtk.ProgressBar 通常用来显示一个较长时间操作的进度信息，它给操作正在 进行中提供了一个视觉的效果。 *Gtk.ProgressBar* 有两种模式： *percentage mode* 和 *activity mode* 。

当程序知道需要执行的工作的工作量（例如从一个文件读取固定字节数）并且可以监视进度时， 可以使用 *Gtk.ProgressBar* 的 *percentage mode* ， 这样用户可以看到一个正在前进的进度条 ——指示任务完成的百分比，这种模式程序要周期性地调用 *Gtk.ProgressBar.set_fraction()* 来更新进度条，传递一个介于0和1之间的值表示新的百分比。

当一个程序不知道任务的工作量时，可以使用 *activity mode* ， 这种模式会显示一个 前后移动的进度块显示任务正在进行中。这种模式下，应用程序要周期性地调用 *Gtk.ProgressBar.pulse()* 来更新进度条，你也可以通过 *Gtk.ProgressBar.set_pulse_step()* 设置每次进度前进的数量。

默认情况下， *Gtk.ProgressBar* 水平的从左向右显示进度，但你也可以通过调用 *Gtk.ProgressBar.set_orientation()* 来改变进度条显示的方向是水平还是竖直。 进度条前进的方向则可以使用 *Gtk.ProgressBar.set_inverted()* 来改变。 *Gtk.ProgressBar* 也可以通过 *Gtk.ProgressBar.set_text()* 和 *Gtk.ProgressBar.set_show_text()* 来设置显示一些文本信息。

ProgressBar 对象

class Gtk.**ProgressBar**

set_fraction(*fraction*)
使进度条 填充 给定百分比的进度。 *fraction* 应该是在0.0和1.0之间，包含0和1。

set_pulse_step(*fraction*)
设置每次调用 *pulse()* 进度条滑块移动的进度占总进度的百分比。

pulse()
一些进度完成了，但是程序不知道完成了多少时调用此函数使进度条进入 *activity mode*，这时会有一个进度滑块前后地移动。每次调用 *pulse()* 使滑块向前移动一点（每次pulse移动的数量由 *set_pulse_step()* 来设置）。

set_orientation(*orientation*)
设置进度条显示的方向，*orientation* 可以是一下其中之一：

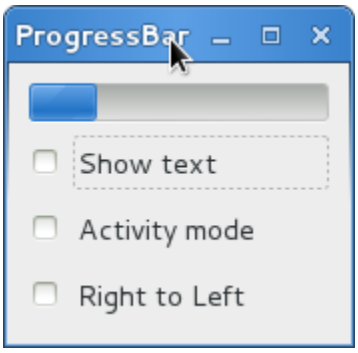
- `Gtk.Orientation.HORIZONTAL`
- `Gtk.Orientation.VERTICAL`

set_show_text(*show_text*)
设置是否在进度条上叠加显示一些文本信息。显示的信息或者是 "text" 属性的值（*set_text()* 设置），或者若其为 `None`，会显示进度的百分比。

set_text(*text*)
使给定的 *text* 叠加显示到进度条上。

set_inverted(*inverted*)
进度条的方向一般是从左向右前进，inverted使进度条前进的方向翻转。

Example




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

from gi.repository import Gtk, GObject

class  ProgressBarWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='ProgressBar Demo')
        self.set_border_width(10)

        vbox = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=6)
        self.add(vbox)

        self.progressbar = Gtk.ProgressBar()
        vbox.pack_start(self.progressbar, True, True, 0)

        button = Gtk.CheckButton('Show text')
```

(下页继续)

(续上页)

```
        button.connect('toggled', self.on_show_text_toggled)
        vbox.pack_start(button, True, True, 0)

        button = Gtk.CheckButton('Activity mode')
        button.connect('toggled', self.on_activity_mode_toggled)
        vbox.pack_start(button, True, True, 0)

        button = Gtk.CheckButton('Right to Left')
        button.connect('toggled', self.on_right_to_left_toggled)
        vbox.pack_start(button, True, True, 0)

        self.timeout_id = GObject.timeout_add(50, self.on_timeout, None)
        self.activity_mode = False

    def on_show_text_toggled(self, button):
        show_text = button.get_active()
        if show_text:
            text = 'some text'
        else:
            text = None
        self.progressbar.set_text(text)
        self.progressbar.set_show_text(show_text)

    def on_activity_mode_toggled(self, button):
        self.activity_mode = button.get_active()
        if self.activity_mode:
            self.progressbar.pulse()
        else:
            self.progressbar.set_fraction(0.0)

    def on_right_to_left_toggled(self, button):
        value = button.get_active()
        self.progressbar.set_inverted(value)

    def on_timeout(self, user_data):
        '''
        Update value on the progress bar
        '''
        if self.activity_mode:
            self.progressbar.pulse()
        else:
            new_value = self.progressbar.get_fraction()+0.01
            if new_value>1:
                new_value = 0.0

            self.progressbar.set_fraction(new_value)

        # time out function, return True so that it will be called continuedly
        return True

win = ProgressBarWindow()
win.connect('delete-event', Gtk.main_quit)
win.show_all()
Gtk.main()
```

4.1.10 Spinner

Gtk.Spinner 显示一个图标大小的旋转动画。通常用来显示一个无尽的活动，代替进度条 GtkProgressBar 。

要开始一个旋转动画，调用 *Gtk.Spinner.start()*，要停止旋转动画，调用 *Gtk.Spinner.stop()* 。

Spinner 对象

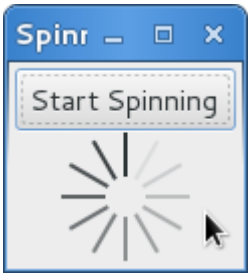
class Gtk.**Spinner**

start()
开始旋转动画。

stop()
停止旋转动画。

Example

注：原例子使用的table布局，但前面翻译的时候分明又说table已经不建议使用了，故自作主张将例子修改为了grid布局的版本了，见谅。




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

from gi.repository import Gtk

class SpinnerAnimation(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='Spinner')
        self.set_border_width(3)
        self.connect('delete-event', Gtk.main_quit)

        self.button = Gtk.ToggleButton('Start Spinning')
        self.button.connect('toggled', self.on_button_toggled)
        self.button.set_active(False)

        self.spinner = Gtk.Spinner()

        self.grid = Gtk.Grid()
        # set all column width same
        self.grid.set_column_homogeneous(True)
        # set all row height same
```

(下页继续)

(续上页)

```
        self.grid.set_row_homogeneous(True)
        self.grid.attach(self.button, 0, 0, 2, 1)
        #self.grid.add(self.button)
        self.grid.attach_next_to(self.spinner, self.button, Gtk.PositionType.BOTTOM,␣
↪2, 2)
        #self.grid.attach(self.spinner, 0, 1, 2, 2)

        self.add(self.grid)
        self.show_all()

    def on_button_toggled(self, button):

        if button.get_active():
            self.spinner.start()
            self.button.set_label('Stop Spinning')
        else:
            self.spinner.stop()
            self.button.set_label('Start Spinning')

myspinner = SpinnerAnimation()
Gtk.main()
```

4.1.11 树与列表控件

Gtk.TreeView 及其相关的控件绝对是用来显示数据的一个及其强大的控件。 使用时通常与 *Gtk.ListStore* 或 *Gtk.TreeStore* 关联，以提供一种 显示数据的方法并提供了很多种维护数据的方法，包括：

- 数据添加、删除或编辑后自动更新；
- 鼠标拖拽支持；
- 数据排序；
- 支持嵌入check box、进度条等控件；
- 可重新排序及调整宽度的列
- 数据过滤

由 *Gtk.TreeView* 的强大与灵活自然也就使其变的很复杂。由于需要的函数很多 对于初级开发者很难掌握其正确的使用。

The Model(模型)

每一个 *Gtk.TreeView* 都有一个与之关联的 *Gtk.TreeModel* ， TreeMode包含要显示给TreeView的数据。而每一个 class:*Gtk.TreeModel* 可以被多个 *Gtk.TreeView* 使用。例如这样可以允许同一份数据同时被以两种不同的方式显示 和编辑，或者是两个TreeView分别显示TreeNode数据不同的列，这类似于使用两条SQL 查询 （或者视图？）就可以从同一个数据库表里取出不同的字段。

尽管理论上你可以实现你自己的模型，但你通常都是使用 *Gtk.ListStore* 或者 *Gtk.TreeStore* 。*Gtk.ListStore* 包含很多行简单的数据，每一行并没有“子行”，而 *Gtk.TreeStore* 则是包含很多行的数据，同时每一行都可以有其孩子行。

当你构造一个模型的时候，你要指定你的每列数据的类型。

```
store = Gtk.ListStore(str, str, float)
```

上面的代码创建了一个列表存储，包含三列，两列类型是字符串，一列是浮点数。

要向这个模型添加数据使用 *Gtk.ListStore.append()* 或者 *Gtk.TreeStore.append()* ——根据你创建的是那种类型的数据模型。

```
treeiter = store.append(["The Art of Computer Programming", "Donald E. Knuth", 25.46])
```

这两个方法都是返回一个 `Gtk.TreeIter` 的实例，并指向最新插入的行。你可以 通过调用 `Gtk.TreeMode.get_iter()` 来取回 `Gtk.TreeIter` 。

一旦数据已经被插入你就可以通过tree的迭代器（tree iter）和列索引取回或者修改数据

```
print store[treeiter][2] # Prints value of third column
store[treeiter][2] = 42.15
```

基于Python内建的 `list` 对象，你可以使用 `len()` 获取行数并使用切片 操作获取或者他们的值。

```
# Print number of rows
print len(store)
# Print all but first column
print store[treeiter][1:]
# Print last column
print store[treeiter][-1]
# Set first two columns
store[treeiter][:2] = ["Donald Ervin Knuth", 41.99]
```

迭代一个tree model所有行的方法也很简单。

```
for row in store:
    # Print values of all columns
    print row[:]
```

记住，如果你使用了 *Gtk.TreeStore* ，上面的代码只会遍历顶层的行， 但是不会遍历节点的孩子。要遍历所有的行和他们的孩子，使用 `print_tree_store` 代替。

```
def print_tree_store(store):
    rootiter = store.get_iter_first()
    print_rows(store, rootiter, "")

def print_rows(store, treeiter, indent):
    while treeiter != None:
        print indent + str(store[treeiter][:])
        if store.iter_has_child(treeiter):
            childiter = store.iter_children(treeiter)
            print_rows(store, childiter, indent + "\t")
        treeiter = store.iter_next(treeiter)
```

除了像上面那样使用list-like方式访问 *Gtk.TreeModel* 中的数据，你也可以使用 `Gtk.TreeIter` 或者 *Gtk.TreePath* 的实例，这两个君代表了一个tree model 的特定的一行数据。 你页可以通过调用 *Gtk.TreeModel.get_iter()* 将path转换为迭代器（iter）。 鉴于 *Gtk.ListStore* 只包含一层，即节点没有子节点，path实际上就是你要访问的 一行数据的index。

```
# Get path pointing to 6th row in list store
path = Gtk.TreePath(5)
treeiter = liststore.get_iter(path)
```

(下页继续)

(续上页)

```
# Get value at 2nd column
value = liststore.get_value(treeiter, 1)
```

对于 *Gtk.TreeStore*，path其实就是索引或者字符串的list。字符串的形式是以冒号分割的数字，每一个数字代表那一层的偏移。例如，path "0" 代表根节点，path "2:4" 代表第三个节点的第五个孩子。

```
# Get path pointing to 5th child of 3rd row in tree store
path = Gtk.TreePath([2, 4])
treeiter = treestore.get_iter(path)
# Get value at 2nd column
value = treestore.get_value(treeiter, 1)
```

Gtk.TreePath 的实例可以想list那样访问，例如 `len(treepath)` 返回 `treepath` 指向的节点的深度，而 `treepath【i】` 则返回第i层的孩子的索引。

TreeModel 对象

class Gtk.**TreeModel**

get_iter(*path*)
返回指向 *path* 的 `Gtk.TreeIter` 的实例。

path 应该是逗号分割的数字组成的字符串或者数组元组。例如，字符串 "10:4:0" 创建的path有三层，指向根节点的第11个孩子，的第五个孩子，的第一个孩子。有点儿绕～～

iter_next(*treeiter*)
返回指向当前level的下一个节点的 `Gtk.TreeIter` 的实例，如果没有下一个则返回 `None`。

iter_previous(*treeiter*)
饭或指向当前level的前一个节点的 `Gtk.TreeIter` 的实例，如果没有前一个节点则返回 `None`。

iter_has_child(*treeiter*)
如果 *treeiter* 有孩子则返回 `True`，否则返回 `False`。

iter_children(*treeiter*)
返回一个指向treeiter的第一个孩子的 `Gtk.TreeIter` 的实例，或者如果没有孩子则返回 `None`。

get_iter_first()
返回指向树的第一个节点的（path "0" 的那个节点） `Gtk.TreeIter` 的实例，或者如果为空树则返回 `None`。

ListStore 对象

class Gtk.**ListStore**(*data_type*[, ...])
创建一个新的 *Gtk.ListStore*，参数指定每一列的数据类型。添加到ListStore的每一行都要在每一列有相应的数据。

支持的数据类型包括标准的Python类型和GTK+的类型：

- str, int, float, long, bool, object
- GObject.GObject

append([*row*])

向ListStore中添加一个新行。

row 是一个包含每列数据的列表，即 `len(store) == len(row)` 。 如果 *row* 忽略或者传递 `None` ，则添加一个空行。

返回指向新添加的行的 `Gtk.TreeIter` 的实例。

TreeStore 对象

class Gtk.**TreeStore**(*data_type*[, ...])
参数与 *Gtk.ListStore* 的构造函数一样。

append(*parent*[, *row*])

向TreeStore中添加一新行数据。 *parent* 必须是一个有效的 `Gtk.TreeIter` 。如果 *parent* 不为 `None` ，会在 *parent* 的最后一个孩子后面添加一个新行，否则会在顶层添加一行。

row 是一个包含每列数据的列表，即 `len(store) == len(row)` 。 如果 *row* 忽略或者传递 `None` ，则添加一个空行。

返回指向新添加的行的 `Gtk.TreeIter` 的实例。

TreePath 对象

class Gtk.**TreePath**(*path*)
构造一个指向由 *path* 指定的节点的 *Gtk.TreePath* 的实例。

如果 *path* 为字符串，则要求是冒号分割的数字列表。例如，字符串 “10:4:0” 创建了一个三层深的path，指向根节点的第11个孩子，的第五个孩子，的第一个孩子。又来了。。。

如果 *path* 为一个列表list或元组tuple，则要包含节点的索引， 参照上面的例子，表达式 `Gtk.TreePath("10:4:0")` 与 `Gtk.TreePath([10, 4, 3])` 等效。

The View（视图）

尽管有很多不同的模型可以选择，但只有一个视图控件。这个视图控件可以与list或者tree store一起工作。 设置好一个 *Gtk.TreeView* 并不是一件困难的事：可以通过构造函数或者调用:meth:*Gtk.TreeView.set_model* 来创建一个:class:*Gtk.TreeModel* 的实例来获取数据。

```
tree = Gtk.TreeView(store)
```

一旦 *Gtk.TreeView* 控件有了一个模型之后，它还需要知道如何显示这个模型，一般是通过列和单元格渲染器（cell renderer）来完成。

Cell renderer 用来以一种方式将数据展现在tree model中。GTK+自带了很多的cell renderer，例如： *Gtk.CellRendererText* ， *Gtk.CellRendererPixbuf* 和 *Gtk.CellRendererToggle*. 另外，相对来说很容易自己定制一个renderer。

Gtk.TreeViewColumn 是 *Gtk.TreeView* 用来在tree iew中组织一列数据的对象。 一般需要列名作为标签显示给用户，要用那种的单元格渲染器，及从模型中获取哪一些数据这些参数。

```
renderer = Gtk.CellRendererText()
column = Gtk.TreeViewColumn("Title", renderer, text=0)
tree.append_column(column)
```

要在一列中渲染多列model中的数据的话，你需要创建一个 *Gtk.TreeViewColumn* 的实例并使用 *Gtk.TreeViewColumn.pack_start()* 来添加model的列。

```
column = Gtk.TreeViewColumn("Title and Author")

title = Gtk.CellRendererText()
author = Gtk.CellRendererText()

column.pack_start(title, True)
column.pack_start(author, True)

column.add_attribute(title, "text", 0)
column.add_attribute(author, "text", 1)

tree.append_column(column)
```

TreeView 对象

class Gtk.**TreeView**([*treemodel*])
创建一个新的 *Gtk.TreeView* 控件，其model初始化为 *treemodel* 。 *treemodel* 必须是一个实现了 *Gtk.TreeModel* 的类，例如 *Gtk.ListStore* 或者 *Gtk.TreeStore* 。如果 *treemodel* 忽略了或者传递 None ， model保持未设置状态并且你后面需要调用 *set_model()* 来设置。

set_model(*model*)
设置tree iew的model。如果之前已经设置了一个model，那么会替换掉原来的，如果 *model* 为 None ，则会清除掉原来旧的model。

get_model()
返回tree view的model。如果之前没有设置model则返回 None 。

append_column(*column*)
添加 *column* 到列列表中，即添加新的一列吧。

get_selection()
返回与tree view关联的 *Gtk.TreeSelection* 。

enable_model_drag_source(*start_button_mask*, *targets*, *actions*)
参数与 *Gtk.Widget.drag_source_set()* 相同。

enable_model_dest_source(*targets*, *actions*)
参数与 *Gtk.Widget.drag_dest_set()* 相同。

TreeViewColumn 对象

class Gtk.**TreeViewColumn**(*label*[, *renderer*[, ***kwargs*]])
创建一个 *Gtk.TreeViewColumn* 。

renderer 为 Gtk.CellRenderer 的实例， *kwargs* 键值对指定 *renderer* 的属性的默认值。这与对每一个键值对调用 *pack_start()* 和 *add_attribute()* 的效果一样。

如果 *renderer* 忽略，你需要手动调用 *pack_start()* 或者 *pack_end()* 。

add_attribute(*renderer*, *attribute*, *value*)
添加一个映射给这一列的属性。

将value设置给 *renderer* 的 *attribute* 属性。例如模型的第二列包含字符串，你可以给 *Gtk.CellRendererText* 设置 “text” 属性来获取第二列的值。

pack_start(*renderer*, *expand*)
打包 *renderer* 到这一列的开始。如果 expand为 `False` ， *renderer* 只会分配需要的空间的大小。如果expand为 `True` ，那未使用的控件会平均的分配给各个单元格。

pack_end(*renderer*, *expand*)
打包 *renderer* 到这一列的最后。如果 expand为 `False` ， *renderer* 只会分配需要的空间的大小。如果expand为 `True` ，那未使用的控件会平均的分配给各个单元格。

set_sort_column_id(*sort_column_id*)
设置模型的哪一列用来给视图的这一列排序，这同时使得本列的列头可以点击。

get_sort_column_id()
返回由 *Gtk.TreeViewColumn.set_sort_column_id()* 设置的id。

set_sort_indicator(*setting*)
设置是否在列头显示一个小箭头。

setting 可以为 `True` （显示提示） 或者 `False` 。

get_sort_indicator()
返回 *Gtk.TreeViewColumn.set_sort_indicator()* 设置的值。

set_sort_order(*order*)
改变本列的排序方式。

order 可以是 `Gtk.SortType.ASCENDING` 或者 `Gtk.SortType.DESCENDING` 。

get_sort_order()
返回 *Gtk.TreeViewColumn.set_sort_order()* 设置的值。

The Selection

绝大多数应用不仅需要处理显示数据的问题，也需要从用户那里接受输入事件。 要接受输入时间，只要创建一个selection对象的引用并且连接到 “changed” 信号。

```
select = tree.get_selection()
select.connect("changed", on_tree_selection_changed)
```

如下代码获取返回选中行的数据：

```
def on_tree_selection_changed(selection):
    model, treeiter = selection.get_selected()
    if treeiter != None:
        print "You selected", model[treeiter][0]
```

你可通过调用 *Gtk.TreeSelection.set_mode()* 来控制哪种选择被允许。 如果你将mode设置为 `Gtk.SelectionMode.MULTIPLE` ，那么 *Gtk.TreeSelection.get_selected()* 就不能工作了，你需要调用 *Gtk.TreeSelection.get_selected_rows()* 。

TreeSelection 对象

class Gtk.**TreeSelection**

set_mode(*type*)
type的值为以下其一：

- `Gtk.SelectionMode.NONE`: 不可选。

- `Gtk.SelectionMode.SINGLE`: 零或一个项目可以选中。
- `Gtk.SelectionMode.BROWSE`: 有且只有一个项目可被选中。 在某些环境，例如在初始化或搜索操作时，是可能没有项目可以选中的。 必须选中是强调用户不能取消当前选中的项目——除非选择了一个新的项目。
- `Gtk.SelectionMode.MULTIPLE`: 任意数量的项目可以被选中。 点击会改变项目的选中状态。Ctrl键可以用来多选，Shift键用来选择一个范围。 一些控件也允许点击然后拖拽来选择一个范围内的项目。

get_selected()
返回 `(model, treeiter)` 的元组， *model* 是当期的模型， *treeiter* 是 `Gtk.TreeIter` 的一个实例并且指向当前选中的行。如果没有行被选中， *treeiter* 为 `None` 。

如果选择的模式为 `Gtk.SelectionMode.MULTIPLE` 此函数将不能工作。

get_selected_rows()
返回所有选中行的 *Gtk.TreePath* 的实例的列表。

排序

排序对于tree view是一个重要的特性，并且标准的实现了 *Gtk.TreeSortable* 接口的tree model（ *Gtk.TreeStore* and *Gtk.ListStore* ）就支持排序。

通过点击列标题排序

Gtk.TreeView 的列通过调用 *Gtk.TreeViewColumn.set_sort_column_id()* 可以很容易的实现排序。 之后这一列就可以通过单击标题来排序了。

首先我们需要一个简单的 *Gtk.TreeView* 和其模型 *Gtk.ListStore* 。

```
model = Gtk.ListStore(str)
model.append(["Benjamin"])
model.append(["Charles"])
model.append(["alfred"])
model.append(["Alfred"])
model.append(["David"])
model.append(["charles"])
model.append(["david"])
model.append(["benjamin"])

treeView = Gtk.TreeView(model)

cellRenderer = Gtk.CellRendererText()
column = Gtk.TreeViewColumn("Title", renderer, text=0)
```

下一步就是是能排序。注意 *column_id* （例子中为 `0` ）指模型中的列而 **不是** 视图中的列。

```
column.set_sort_column_id(0)
```

设置一个定制的排序函数

当然你也可以定制一个比较函数来改变排序的行为。例子中我们创建一个 比较函数来实现大小写敏感的排序功能。 上面的例子排序后的列表像下面这样：：

```
alfred
Alfred
benjamin
Benjamin
charles
Charles
david
David
```

大小写敏感的排序排序后的列表像这样：：

```
Alfred
Benjamin
Charles
David
alfred
benjamin
charles
david
```

首先一个我们需要一个比较函数。这个函数需要两个行，并且如果第一个行应该排在前面返回负数，如果两个行的比较结果相等返回0，如果第二个行应该排在前面则返回一个正数。

```
def compare(model, row1, row2, user_data):
    sort_column, _ = model.get_sort_column_id()
    value1 = model.get_value(row1, sort_column)
    value2 = model.get_value(row2, sort_column)
    if value1 < value2:
        return -1
    elif value1 == value2:
        return 0
    else:
        return 1
```

排序函数需要通过 *Gtk.TreeSortable.set_sort_func()* 来设置。

```
model.set_sort_func(0, compare, None)
```

TreeSortable 对象

class Gtk.**TreeSortable**

set_sort_column_id(*sort_column_id*, *order*)
设置当前的排序列为 *sort_column_id* 。

order 可以为 `Gtk.SortType.ASCENDING` 或者 `Gtk.SortType.DESCENDING` 。

get_sort_column_id()
返回一个包含当前排序列和排序方法的元组。

set_sort_func(*sort_column_id*, *sort_func*, *user_data*)
设置用来通过 *sort_column_id* 来排序时的比较函数。

user_data 会传递给 *sort_func* 。

sort_func 是一个原型为 `sort_func(model, iter1, iter2, user_data)` 的函数。并且在 *iter1* 应该排在 *iter2* 前时返回一个负数，相等时返回0，*iter2* 应该排在 *iter1* 前时返回一个正数。

set_default_sort_func(*sort_func*, *user_data*)
参见 *Gtk.TreeSortable.set_sort_func()* 。用来设置当使用默认的排序列时使用的比较函数。

4.1.12 单元格渲染器（CellRenderers）

Gtk.CellRenderer 控件用来在例如 *Gtk.TreeView* 或 *Gtk.ComboBox* 这样的控件中显示信息。这些单元格渲染器与相关联的控件联系紧密并且非常强大，有大量的配置选项用来以 不同的方式来显示大量的数据。有七种 Gtk.CellRenderer 可以用于不同的目的。:

- *Gtk.CellRendererText*
- *Gtk.CellRendererToggle*
- *Gtk.CellRendererPixbuf*
- *Gtk.CellRendererCombo*
- *Gtk.CellRendererProgress*
- Gtk.CellRendererSpinner
- *Gtk.CellRendererSpin*
- Gtk.CellRendererAccel

CellRendererText

Gtk.CellRendererText 在单元格中渲染给定的文本，并使用其属性提供的字体、颜色与style信息。 如果 “ellipsize” 允许，文本太长时会显示为带省略号的形式。

默认 *Gtk.CellRendererText* 控件中的文本是不可编辑的。可以通过其 “editable” 属性设置为 True 来改变该行为。

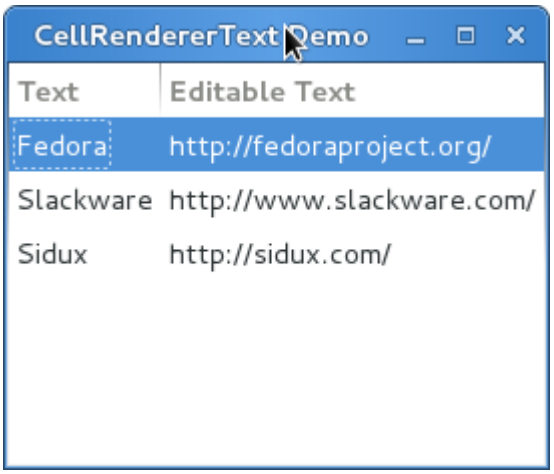
```
cell.set_property("editable", True)
```

可编辑之后你就可以连接 “editable” 信号来更新你的 *Gtk.TreeModel* 了。

CellRendererText 对象

class Gtk.**CellRendererText**
创建一个新的 *Gtk.CellRendererText* 的实例。使用对象属性来调整文本的绘制。 与 *Gtk.TreeViewColumn* 一起，你可以绑定 *Gtk.TreeModel* 的值到一个属性。 例如，你可以绑定 “text” 属性与模型中的一个字符串值，来在 *Gtk.TreeView* 的每一行渲染不同的文本。

例子




```
from gi.repository import Gtk

class CellRendererTextWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='CellRendererText Demo')

        self.set_default_size(200, 200)

        self.liststore = Gtk.ListStore(str, str)
        self.liststore.append(['Fedora', 'http://fedoraproject.org/'])
        self.liststore.append(['Slackware', 'http://www.slackware.com/'])
        self.liststore.append(['Sidux', 'http://sidux.com/'])

        treeview = Gtk.TreeView(model=self.liststore)

        renderer_text = Gtk.CellRendererText()
        col_txt = Gtk.TreeViewColumn('Text', renderer_text, text=0)
        treeview.append_column(col_txt)

        renderer_editabletext = Gtk.CellRendererText()
        renderer_editabletext.set_property('editable', True)

        col_editabletxt = Gtk.TreeViewColumn('Editable Text',
                renderer_editabletext, text=1)
        treeview.append_column(col_editabletxt)
        renderer_editabletext.connect('edited', self.text_edited)

        self.add(treeview)

    def text_edited(self, widget, path, text):
        self.liststore[path][1] = text

win = CellRendererTextWindow()
win.connect('delete-event', Gtk.main_quit)
win.show_all()
Gtk.main()
```

CellRendererToggle

Gtk.CellRendererToggle 在单元格内渲染一个 toggle button 。 按钮被渲染为一个radio 按钮或者checkbutton按钮，根据 “radio” 属性。当激活（active）后会 激发 “toggled” 信号。

由于 *Gtk.CellRendererToggle* 有两个状态，active 和 not active，你要将 “active” 属性与一个布尔值的模型绑定，这样才能使得check button的状态反映出模型的状态。

CellRendererToggle 对象

class Gtk.**CellRendererToggle**
创建一个新的 *Gtk.CellRendererToggle* 实例。

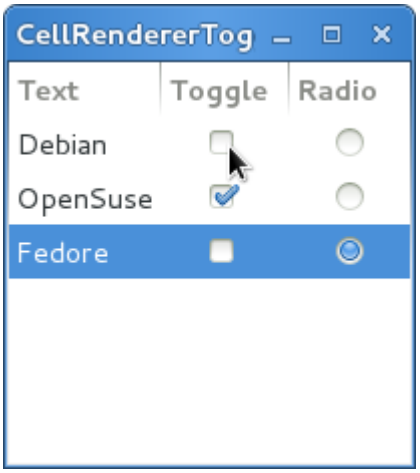
set_active(*setting*)
设置单元格渲染器的是Activates 还是 deactivates 。

get_active()
返回单元格渲染器的active状态。

set_radio(*radio*)
如果 *radio* 为 `True`，单元格渲染器会渲染一个radio样式的toggle按钮（即 一个互斥的按钮组），如果为 `False` ，则渲染为一个一个 check toggle。

get_radio()
返回是否被渲染为一个radio 按钮组。

Example




```
from gi.repository import Gtk

class CellRendererToggleWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='CellRendererToggle Demo')

        self.set_default_size(200, 200)

        self.liststore = Gtk.ListStore(str, bool, bool)
        self.liststore.append(['Debian', False, True])
```

(下页继续)

(续上页)

```
        self.liststore.append(['OpenSuse', True, False])
        self.liststore.append(['Fedore', False, False])

        treeview = Gtk.TreeView(model=self.liststore)

        renderer_text = Gtk.CellRendererText()
        col_txt = Gtk.TreeViewColumn('Text', renderer_text, text=0)
        treeview.append_column(col_txt)

        renderer_toggle = Gtk.CellRendererToggle()
        renderer_toggle.connect('toggled', self.on_cell_toggled)
        col_toggle =Gtk.TreeViewColumn('Toggle', renderer_toggle, active=1)
        treeview.append_column(col_toggle)

        renderer_radio = Gtk.CellRendererToggle()
        renderer_radio.set_radio(True)
        renderer_radio.connect('toggled', self.on_cell_radio_toggled)
        col_radio = Gtk.TreeViewColumn('Radio', renderer_radio, active=2)
        treeview.append_column(col_radio)

        self.add(treeview)

    def on_cell_toggled(self, widget, path):
        self.liststore[path][1] = not self.liststore[path][1]

    def on_cell_radio_toggled(self, widget, path):
        selected_path = Gtk.TreePath(path)
        for row in self.liststore:
            row[2] = (row.path == selected_path)

win = CellRendererToggleWindow()
win.connect('delete-event', Gtk.main_quit)
win.show_all()
Gtk.main()
```

CellRendererPixbuf

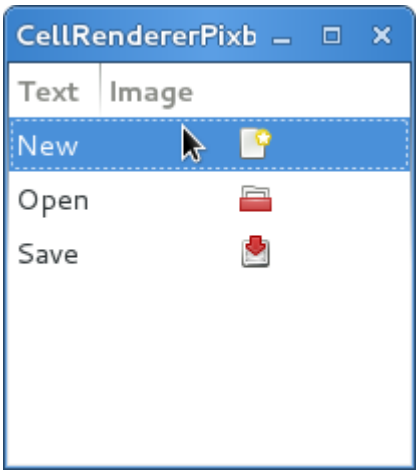
Gtk.CellRendererPixbuf 用于在单元格中渲染一张图片。 允许渲染一个给定的 Gdk.Pixbuf (设置 “pixbuf” 属性) 或者 *stock item* (设置 “stock-id” 属性)。

CellRendererPixbuf 对象

class Gtk.**CellRendererPixbuf**

创建一个新的 *Gtk.CellRendererPixbuf* 。使用对象属性来调整渲染的 参数，例如你可以绑定单元格渲染器的 “pixbuf” 或者 “stock-id” 属性到模型的pixbuf值，这样就可以在 *Gtk.TreeView* 的每一行渲染不同的图片。

Example




```
from gi.repository import Gtk

class CellRendererPixbufWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="CellRendererPixbuf Example")

        self.set_default_size(200, 200)

        self.liststore = Gtk.ListStore(str, str)
        self.liststore.append(["New", Gtk.STOCK_NEW])
        self.liststore.append(["Open", Gtk.STOCK_OPEN])
        self.liststore.append(["Save", Gtk.STOCK_SAVE])

        treeview = Gtk.TreeView(model=self.liststore)

        renderer_text = Gtk.CellRendererText()
        column_text = Gtk.TreeViewColumn("Text", renderer_text, text=0)
        treeview.append_column(column_text)

        renderer_pixbuf = Gtk.CellRendererPixbuf()

        column_pixbuf = Gtk.TreeViewColumn("Image", renderer_pixbuf, stock_id=1)
        treeview.append_column(column_pixbuf)

        self.add(treeview)

win = CellRendererPixbufWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

CellRendererCombo

Gtk.CellRendererCombo 像 *Gtk.CellRendererText* 一样在单元格内渲染文本，但是后者只提供了一个 简单的输入框来编辑文本，而 *Gtk.CellRendererCombo* 提供了一个 *Gtk.ComboBox* 控件来编辑文本。在组合框（combo box）中显示的 值通过 “model” 属性从 *Gtk.TreeModel* 中获取。

组合框单元格渲染器对于添加文本到组合框列表中很严格，通过设置 “text-column” 属性 来设置要显示的列。

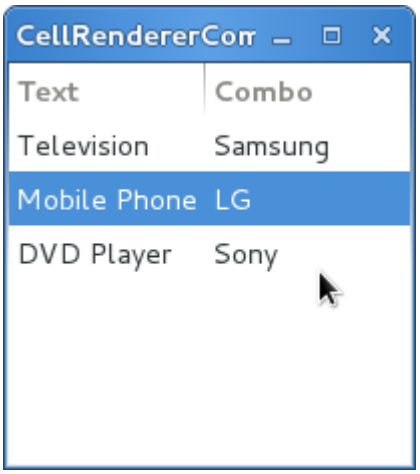
Gtk.CellRendererCombo 有两种操作模式，即有或者没有关联的 *Gtk.Entry* 控件，依赖于 “has-entry” 属性的值。

CellRendererCombo 对象

class Gtk.**CellRendererCombo**

创建一个新的 *Gtk.CellRendererCombo* 。使用对象的属性来调整渲染效果。 例如绑定单元格渲染器的 “text” 属性到模型的string字段来在 *Gtk.TreeView* 的每一行显示不同的文本。

Example




```
from gi.repository import Gtk

class CellRendererComboWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="CellRendererCombo Example")

        self.set_default_size(200, 200)

        liststore_manufacturers = Gtk.ListStore(str)
        manufacturers = ["Sony", "LG", "Panasonic", "Toshiba", "Nokia", "Samsung"]
        for item in manufacturers:
            liststore_manufacturers.append([item])

        self.liststore_hardware = Gtk.ListStore(str, str)
        self.liststore_hardware.append(["Television", "Samsung"])
        self.liststore_hardware.append(["Mobile Phone", "LG"])
        self.liststore_hardware.append(["DVD Player", "Sony"])

        treeview = Gtk.TreeView(model=self.liststore_hardware)

        renderer_text = Gtk.CellRendererText()
        column_text = Gtk.TreeViewColumn("Text", renderer_text, text=0)
        treeview.append_column(column_text)
```

(下页继续)

(续上页)

```
        renderer_combo = Gtk.CellRendererCombo()
        renderer_combo.set_property("editable", True)
        renderer_combo.set_property("model", liststore_manufacturers)
        renderer_combo.set_property("text-column", 0)
        renderer_combo.set_property("has-entry", False)
        renderer_combo.connect("edited", self.on_combo_changed)

        column_combo = Gtk.TreeViewColumn("Combo", renderer_combo, text=1)
        treeview.append_column(column_combo)

        self.add(treeview)

    def on_combo_changed(self, widget, path, text):
        self.liststore_hardware[path][1] = text

win = CellRendererComboWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

CellRendererProgress

Gtk.CellRendererProgress 渲染一个数值为一个显示在单元格中的进度条。 你也可以在进度条上面显示文本。

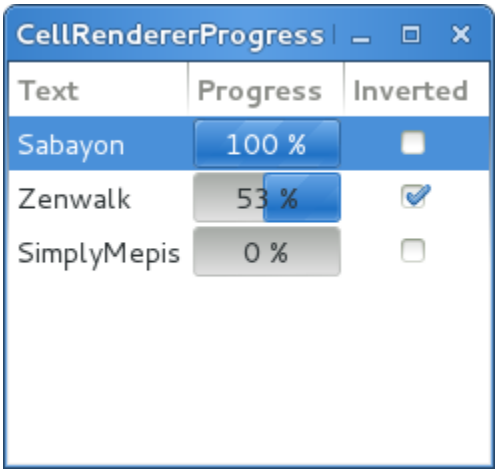
进度条的进度百分比值可以通过改变 “value” 属性来修改。类似于 *Gtk.ProgressBar* ， 你可以通过设置 “pulse” 属性而不是 “value” 属性来使能 *activity mode* 。

CellRendererProgress 对象

class Gtk.**CellRendererProgress**

创建一个新的 *Gtk.CellRendererProgress*.

Example



```
from gi.repository import Gtk, GObject

class CellRendererProgressWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="CellRendererProgress Example")

        self.set_default_size(200, 200)

        self.liststore = Gtk.ListStore(str, int, bool)
        self.current_iter = self.liststore.append(["Sabayon", 0, False])
        self.liststore.append(["Zenwalk", 0, False])
        self.liststore.append(["SimplyMepis", 0, False])

        treeview = Gtk.TreeView(model=self.liststore)

        renderer_text = Gtk.CellRendererText()
        column_text = Gtk.TreeViewColumn("Text", renderer_text, text=0)
        treeview.append_column(column_text)

        renderer_progress = Gtk.CellRendererProgress()
        column_progress = Gtk.TreeViewColumn("Progress", renderer_progress,
            value=1, inverted=2)
        treeview.append_column(column_progress)

        renderer_toggle = Gtk.CellRendererToggle()
        renderer_toggle.connect("toggled", self.on_inverted_toggled)
        column_toggle = Gtk.TreeViewColumn("Inverted", renderer_toggle,
            active=2)
        treeview.append_column(column_toggle)

        self.add(treeview)

        self.timeout_id = GObject.timeout_add(100, self.on_timeout, None)

    def on_inverted_toggled(self, widget, path):
        self.liststore[path][2] = not self.liststore[path][2]

    def on_timeout(self, user_data):
        new_value = self.liststore[self.current_iter][1] + 1
        if new_value > 100:
            self.current_iter = self.liststore.iter_next(self.current_iter)
            if self.current_iter == None:
                self.reset_model()
            new_value = self.liststore[self.current_iter][1] + 1

        self.liststore[self.current_iter][1] = new_value
        return True

    def reset_model(self):
        for row in self.liststore:
            row[1] = 0
        self.current_iter = self.liststore.get_iter_first()

win = CellRendererProgressWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

CellRendererSpin

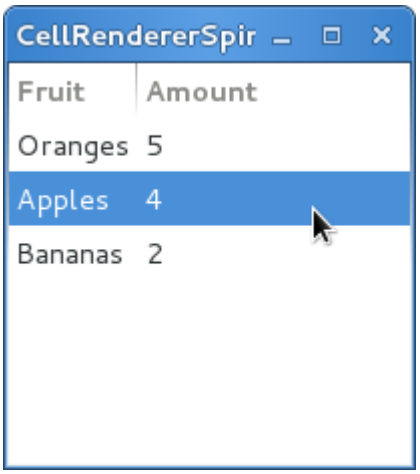
Gtk.CellRendererSpin 与renders text in a cell like *Gtk.CellRendererText* 类似，在单元格中渲染文本，但与后者提供一个简单的 输入框编辑不同， *Gtk.CellRendererSpin* 提供了一个 *Gtk.SpinButton* 控件。当然这意味着文本必须是一个float浮点数。

spinbutton 的范围从 “adjustment” 属性来获取，可以设置映射为模型的一列。 *Gtk.CellRendererSpin* 也可以通过属性来设置步进值及显示的数字的位数。

CellRendererSpin 对象

class Gtk.**CellRendererSpin**
创建一个新的 *Gtk.CellRendererSpin* 。

Example




```
from gi.repository import Gtk

class CellRendererSpinWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="CellRendererSpin Example")

        self.set_default_size(200, 200)

        self.liststore = Gtk.ListStore(str, int)
        self.liststore.append(["Oranges", 5])
        self.liststore.append(["Apples", 4])
        self.liststore.append(["Bananas", 2])

        treeview = Gtk.TreeView(model=self.liststore)

        renderer_text = Gtk.CellRendererText()
        column_text = Gtk.TreeViewColumn("Fruit", renderer_text, text=0)
        treeview.append_column(column_text)

        renderer_spin = Gtk.CellRendererSpin()
```

(下页继续)

(续上页)

```
        renderer_spin.connect("edited", self.on_amount_edited)
        renderer_spin.set_property("editable", True)

        adjustment = Gtk.Adjustment(0, 0, 100, 1, 10, 0)
        renderer_spin.set_property("adjustment", adjustment)

        column_spin = Gtk.TreeViewColumn("Amount", renderer_spin, text=1)
        treeview.append_column(column_spin)

        self.add(treeview)

    def on_amount_edited(self, widget, path, value):
        self.liststore[path][1] = int(value)

win = CellRendererSpinWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

4.1.13 组合框（ComboBox）

Gtk.ComboBox 允许你从一个下拉列表中选择项目。 由于其占用更少的空间，应该是在屏幕上显示很多radio按钮的优选方案。如果恰当，他可以 显示很多条目额外的信息，如文本，图片，checkbox或者一个进度条。

Gtk.ComboBox 与 *Gtk.TreeView* 非常类似，他们都使用模型-视图模式； 有效选项的列表在tree model中指定，而选项的显示可以通过 *cell renderers* 适配到模型中。如果组合框包含很多的项目， 那么将他们显示在网格中比显示在列表中更好，调用 *Gtk.ComboBox.set_wrap_width()* 可以实现。

Gtk.ComboBox 控件通常会限制用户可能的选择，但是你可以设置一个 *Gtk.Entry* ， 允许用户输入任意的文本如果列表中的项目均不合适。 要设置 *Gtk.Entry* ， 使用静态方法 *Gtk.ComboBox.new_with_entry()* 或者 *Gtk.ComboBox.new_with_model_and_entry()* 来创建 *Gtk.ComboBox* 的实例。

对于简单的文本选择下拉列表， *Gtk.ComboBox* 的 模型-视图API可能有点大材小用（太复杂），因此 *Gtk.ComboBoxText* 提供了一种更简单的选择。 *Gtk.ComboBox* 和 *Gtk.ComboBoxText* 均可以包含一个输入框。

ComboBox objects

class Gtk.**ComboBox**

static new_with_entry()
创建一个带有输入框的空的 *Gtk.ComboBox* 。

static new_with_model(*model*)
创建一个新的 *Gtk.ComboBox* ,模型被初始化为 *model* 。

static new_with_model_and_entry(*model*)
创建一个新的带有输入框的 *Gtk.ComboBox* ， 并且模型被初始化为 *model* 。

get_active_iter()
返回一个 Gtk.TreeIter 的实例，并且指向当前激活的项目。 如果没有激活的项目，则返回 None 。

set_model(*model*)
设置组合框使用的 *model* 。这会替换之前设置过的 model（如果有）。 如果model为 None ， 则会取消之前的设置。注意此方法不会清除单元格渲染器的内容。

get_model()
返回作为组合框数据源的 *Gtk.TreeModel* 。

set_entry_text_column(*text_column*)
设置组合框要从模型的哪一列 *text_column* 来获取文笔内容，模型中的 *text_column* 必须为 str 类型。

只用组合框的 “has-entry” 属性为 True 时才可以用。

set_wrap_width(*width*)
设置组合框的wrap width为 *width* 。wrap width指当你想弹出一个网格显示的 列表时的首选列数。

ComboBoxText objects

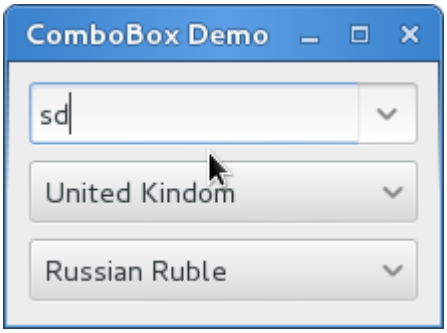
class Gtk.**ComboBoxText**

static new_with_entry()
创建一个空的带有输入框的 *Gtk.ComboBoxText* 。

append_text(*text*)
添加 *text* 到组合框的字符串列表中。

get_active_text()
返回组合框当前激活的文本字符串，如果没有被选中的，则返回 None 。 本函数会返回字符串的内容（不一定是列表中的项目）。

Example




```
from gi.repository import Gtk

class ComboBoxWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='ComboBox Demo')

        self.set_border_width(10)

        name_store = Gtk.ListStore(int, str)
        name_store.append([1, 'Billy Bob'])
        name_store.append([11, 'Billy Bob Junior'])
```

(下页继续)

(续上页)

```
        name_store.append([12, 'Sue Bob'])
        name_store.append([2, 'Joey Jijo'])
        name_store.append([3, 'Rob McRoberts'])
        name_store.append([31, 'Xavier McRoberts'])

        vbox = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=6)

        name_combo = Gtk.ComboBox.new_with_model_and_entry(name_store)
        name_combo.connect('changed', self.on_name_combo_changed)
        name_combo.set_entry_text_column(1)
        vbox.pack_start(name_combo, False, False, 0)

        country_store = Gtk.ListStore(str)
        countries = ['Austria', 'Brazil', 'Belgium', 'France', 'Germany',
                'Switzerland', 'United Kindom', 'United States of America', 'Uruguay']
        for country in countries:
            country_store.append([country])

        country_combo = Gtk.ComboBox.new_with_model(country_store)
        country_combo.connect('changed', self.on_country_combo_changed)
        renderer_text = Gtk.CellRendererText()
        country_combo.pack_start(renderer_text, True)
        country_combo.add_attribute(renderer_text, 'text', 0)
        vbox.pack_start(country_combo, False, False, True)

        currencies = ['Euro', 'US Dollars', 'British Pound', 'Japanese Yen',
                'Russian Ruble', 'Mexican peso', 'Swiss franc']
        currency_combo = Gtk.ComboBoxText()
        currency_combo.set_entry_text_column(0)
        currency_combo.connect('changed', self.on_currency_combo_changed)
        for currency in currencies:
            currency_combo.append_text(currency)

        vbox.pack_start(currency_combo, False, False, 0)

        self.add(vbox)

    def on_name_combo_changed(self, combo):
        tree_iter = combo.get_active_iter()
        if tree_iter != None:
            model = combo.get_model()
            row_id, name = model[tree_iter][:2]
            print 'Selected: ID=%d, name=%s' %(row_id, name)
        else:
            entry = combo.get_child()
            print 'Entered: %s' %entry.get_text()

    def on_country_combo_changed(self, combo):
        tree_iter = combo.get_active_iter()
        if tree_iter != None:
            model = combo.get_model()
            country = model[tree_iter][0]
            print 'Select: country=%s' %country

    def on_currency_combo_changed(self, combo):
        text = combo.get_active_text()
```

(下页继续)

(续上页)

```
69          if text != None:
70              print 'Selected: currency:%s' %text
71
72  win = ComboBoxWindow()
73  win.connect('delete-event', Gtk.main_quit)
74  win.show_all()
75  Gtk.main()
```

4.1.14 IconView

Gtk.IconView 是一个在网格视图中显示很多图标的控件。支持诸如拖拽、排序等特性。

与 *Gtk.TreeView* 类似， *Gtk.IconView* 使用 Gtk.location 作为其数据模型，但其并不使用 *cell renderers* ，而是要求 与其关联的 *Gtk.ListStore* 要有一列包含 GdkPixbuf.Pixbuf 对象。

Gtk.IconView 支持很多中选择模式以允许可以一次多选很多个图标，或者只能 选择一个，或者直接完全不可选择。要指定选择模式，使用 *Gtk.IconView.set_selection_mode()* 方法并传递 Gtk.SelectionMode 的实例来指定选择模式。

IconView 对象

class Gtk.**IconView**

static new_with_area(*area*)
创建一个新的 *Gtk.IconView* 控件并使用指定的 *area* 来布局 图标。

static new_with_model(*model*)
创建一个新的 *Gtk.IconView* 控件并使用指定的 *model* 。

set_model(*model*)
设置 *Gtk.IconView* 要使用的数据模型。 如果 *Gtk.IconView* 已经设置了一个模型，将会被替换。 如果 *modle* 为 None ，则会取消原来设置的模型。

get_model()
返回 *Gtk.IconView* 基于的模型，如果未曾设置则返回 None 。

set_text_column(*column*)
设置文本列为 *column* ，文本列的类型必须是 str 。

get_text_column()
返回文本列，如未设置则返回-1。

set_markup_column(*column*)
设置 *Gtk.IconView* 的带有标记信息的列的内容为 *column* 。 带有标记信息的列必须是 str 类型的。 如果设置了带有标记信息的列，则会覆盖 *set_text_column()* 设置的文本列。

get_markup_column()
返回带有标记信息的列，如果未曾设置则返回-1。

set_pixbuf_column(*column*)
设置图标列的图标为 *column* 。 图标列的类型必须是 GkdPixbuf.pixbufs

get_pixbuf_column()
返回设置的图标列，若未曾设置则返回-1。

get_item_at_pos(*x*, *y*)
查找在 (*x*, *y*) 点处的path，点的坐标为bin_window的相对坐标。 与 `get_path_at_pos()` 不同，本方法也或获取指定坐标的单元格。 关于将控件的坐标转换为bin_window坐标，请参考 `convert_widget_to_bin_window_coords()` 。

convert_widget_to_bin_coords(*x*, *y*)
将控件的坐标转换为bin_window的坐标，以供类似 `get_path_at_pos()` 这样的方法调用。

set_cursor(*path*, *cell*, *start_editing*)
设置当前的键盘焦点为 *path* 并选中。当你想把用户的注意力集中到某个项目时这 非常有用。如果 *cell* 不为 `None` ，则会聚焦到 *cell* 上。 另外，如果 *start_editing* 为 True, 则会在指定的cell上立即开始编辑状态。

本函数经常跟在 `grab_focus()` 后不调用以获取键盘的焦点。注意编辑状态 只有在实现了这种功能的控件上可用。

get_cursor()
返回当前的光标（cursor）路径和单元格。如果cursor未设置，path返回 `None`。 如果没有聚焦于单元格，cell也返回 `None` 。

selected_foreach(*func*, *data*)
对于每一个选中的图标调用函数func，注意在调用此方法时模型和选择的项目不能修改。 Note that the model or selection cannot be modified from within this method.

set_selection_mode(*mode*)
设置 *Gtk.IconView* 的选择模式 `Gtk.SelectionMode` 。

get_selection_mode()
返回 `Gtk.IconView`的选择模式 `:class:`Gtk.SelectionMode` 。

set_item_orientation(*orientation*)
设置项目的 “item-orientation” 属性，这决定了图标的标签是在图标的旁边还是下面。

get_item_orientation()
返回决定图标的标签是在图标旁边还是在图标下面的 “item-orientation” 属性的值。

set_columns(*columns*)
设置决定了图标排布列数的 “columns” 属性的值。 如果 *column* 的值为-1，则列数会自动选择来填充可用的区域。

get_columns()
返回 “cloumns” 属性的值。

set_item_width(*item_width*)
设置指定每一个项目所占宽度的 “item-width” 属性的值。 如果设置为-1，则会自动决定一个合适的尺寸。

get_item_width()
返回 “item-width” 属性的值。

set_spacing(*spacing*)
设置 “spacing” 属性的值，该属性指定了项目的单元格（即图标和标签）间的空白。

set_row_spacing(*row_spacing*)
设置 “row-spacing” 属性的值，该属性指定了每行之间空白的大小。

get_row_spacing()
返回 “row-spacing” 属性的值。

set_column_spacing(*column_spacing*)
设置 “column-spacing” 属性的值，该属性指定了列之间的空白的大小。

get_column_spacing()
返回 “column-spacing” 属性的值。

set_margin(*margin*)
设置 “margin” 属性的值，该属性指定了在顶部、底部及左右边的空白大小。

get_margin()
返回 “margin” 属性的值。

set_item_padding(*item_padding*)
设置 “item-padding” 属性的值，该属性指定了图标周围的填充的大小。

get_item_padding()
返回 “item-padding” 属性的值。

select_path(*path*)
选择 *path* 指向的行。

unselect_path(*path*)
反选 *path* 指向的行。

path_is_selected(*path*)
如果 *path* 指向的图标被选中返回 `True` ，否则返回 `False` 。

get_selected_items()
创建指向所有选中项目的的path的列表。另外，如果你打算调用完本函数后修改模型里 的值，你可能需要将返回的列表list转换为 `Gtk.TreeRowReference` 的列表。

select_all()
选中所有的图标。要求 *Gtk.IconView* 的选择模式必须被设置为 `Gtk.SelectionMode.MULTIPLE` 。

unselect_all()
反选所有的图标。

scroll_to_path(*path*, *use_align*, *row_align*, *col_align*)
将 *Gtk.IconView* 的基线（alignments）移动到 *path* 所指向的位置（行）。 *row_align* 决定了该项目的行应该显示在什么位置， *col_align* 决定了该项目的列应该显示在什么位置， 这两个参数均要求是介于0.0和1.0之间的值。0.0代表左/上的位置，1.0代表右/下的位置， 0.5代表中间。

如果 *use_align* 为 `False` ，alignment参数会被忽略，会做最少的工作来 将该项目滚动到屏幕中。这意味着项目会被滚动到离现在的位置最近的边缘。如果项目 当前在屏幕上可见，则什么也不做。

只有设置过模型本函数才可以工作，并且 *path* 是model中的有效行。如果模型在 *Gtk.IconView* 实现之前改变了，则path也会改变以反应这个变化。

get_visible_range()
返回地一个和最后一个可见的项目的 *Gtk.TreePath* 。 注意中间可能有不可见的path。

set_tooltip_item(*tooltip*, *path*)
设置 *tooltip* 的显示区域为 *path* 指定的区域 更简单的可选方法请参考 *set_tooltip_column()* 。 请参考 `Gtk.Tooltip.set_tip_area()` 。

set_tooltip_cell(*tooltip*, *path*, *cell*)
设置 *tooltip* 的显示区域为 *path* 指向的项目中 *cell* 所占据的区域。 更简单的可选方法请参考 *set_tooltip_column()* 。 请参考 `Gtk.Tooltip.set_tip_area()` 。

get_tooltip_context(*x*, *y*, *keyboard_tip*)
本函数被设计用于 *Gtk.IconView* 的 “query-tooltip” 信号处理函数。 *x*, *y* 和 *keyboard_tip* 的值即为信号处理函数收到的参数的值，应该不修改 就传递给本函数。

返回值指示给定坐标处是(True)否(False)有一个图标视图的项目的鼠标提示。 对于键盘的提示则是指光标处的项目。 当返回 `True` 时，所有返回的项目会设置为指向该行及对应的模型。 当 *keyboard_tooltip* 为 `False` 时， *x* 和 *y* 总是转换为想对于 *Gtk.IconView* 的bin_window的相对坐标。

set_tooltip_column(*column*)
如果你只是打算给所有的项目提供一个简单的（只显示文本）tooltips，你可以使用 此函数让 *Gtk.IconView* 自动处理这些情况。 *column* 为包含tooltips的 *Gtk.IconView* 的模型中的列，若传递-1则 禁用此项特性。

使能后，”has-tooltip” 属性会被置为 `True` 并且 *Gtk.IconView* 会连接一个 “query-tooltip” 信号处理函数。

注意信号处理函数使用 `Gtk.Tooltips.set_markup()` 来设置文本，因此 &, <等都要被转义。

get_tooltip_column()
返回用于在 *Gtk.IconView* 的行上显示提示信息的 *Gtk.IconView* 的模型的列，如果禁用了提示信息，则返回-1 。

get_item_row(*path*)
获取 *path* 当前显示的项目的行，行号从0开始。

get_item_column(*path*)
获取 *path* 当前显示的项目的列，列号从0开始。

enable_model_drag_source(*start_button_mask*, *targets*, *n_targets*, *actions*)
设置 *Gtk.IconView* 为自动的拖拽中拖拽的源。 调用此函数会设置 “reorderable” 为 `False` 。

enable_model_drag_dest(*targets*, *n_targets*, *actions*)
设置 *Gtk.IconView* 为自动的拖拽中拖拽的目的。 调用此函数会设置 “reorderable” 为 `False` 。

unset_model_drag_source()
取消 *enable_model_drag_source()* 的效果。 调用此函数会设置 “reorderable” 为 `False` 。

unset_model_drag_dest()
取消 *enable_model_drag_dest()* 的效果。 调用此函数会设置 “reorderable” 为 `False` 。

set_reorderable(*reorderable*)
本函数为允许你为支持 `Gtk.TreeDragSource` 和 `Gtk.TreeDragDest` 接口模型的排序提供了方便。 *Gtk.TreeStore* 和 *Gtk.ListStore* 均支持。 如果 *reorderable* 为 `True` ， 用户可以通过拖拽对应的行来排序。 开发者可以通过连接模型的 “row_inserted” 和 “row_deleted” 信号来获知行顺序的变化。 排序功能的实现是通过设置图标视图为拖拽的源和目的来实现的。因此此时拖拽 不能用于其他任何的目的。

对于顺序本函数不会给你任何程度的控制——任何的排序均是支持的。如果你需要更多 的控制，你应该手动来处理拖拽。

get_reorderable()
获取用户是否可以通过拖拽来重新排序。 参考 *set_reorderable()*.

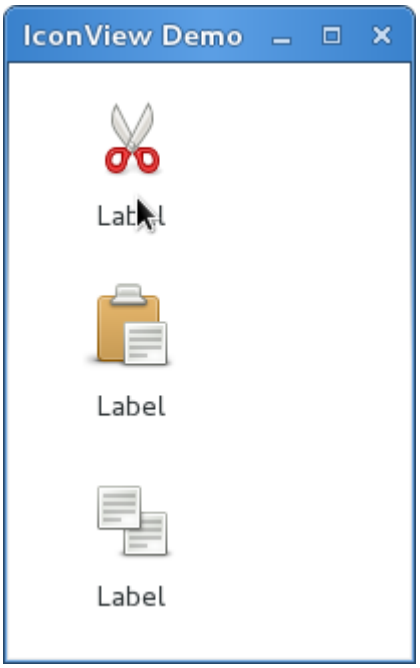
set_drag_dest_item(*path*, *pos*)
设置项目高亮以提供回馈效果。

get_drag_dest_item()
返回高亮回馈的项目的信息。

get_dest_item_at_pos(*drag_x*, *drag_y*)
返回给定位置的目的项目。

create_drag_icon(*path*)
创建一个 Cairo.Surface 来代表 *path* 处的项目。 本图片用于一个拖拽的图标。

Example




```
from gi.repository import Gtk
from gi.repository.GdkPixbuf import Pixbuf

icons = ['gtk-cut', 'gtk-paste', 'gtk-copy']

class IconViewWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='IconView Demo')
        self.set_default_size(200, 200)

        liststore = Gtk.ListStore(Pixbuf, str, str)
        iconview = Gtk.IconView.new()
        iconview.set_model(liststore)
        iconview.set_pixbuf_column(0)
        iconview.set_text_column(1)
        iconview.set_tooltip_column(2)

        for icon in icons:
            pixbuf = Gtk.IconTheme.get_default().load_icon(icon, 64, 0)
            liststore.append([pixbuf, 'Label', icon])

        self.add(iconview)

win = IconViewWindow()
win.connect('delete-event', Gtk.main_quit)
win.show_all()
Gtk.main()
```

4.1.15 Multiline Text Editor

Gtk.TextView 控件可以用来显示和编辑大量的格式化的文本。与 *Gtk.TreeView* 类似，其也有一个模型/视图的设计， *Gtk.TextBuffer* 就是代表正在被编辑的文本 的模型。模型也允许两个或更多个的 *Gtk.TextView* 共享同一个 *Gtk.TextBuffer* ， 并允许这些文本有不同的显示。或者你也可以在同一个 *Gtk.TextView* 的不同时间显示 不同 text buffer 的文本。

视图（The View）

Gtk.TextView 就是用户可以添加、编辑或者删除文本的前端，其通常用来编辑 多行的文本。当创建一个 *Gtk.TextView* 时其自动包含一个默认的 *Gtk.TextBuffer* ， 你可以通过 *Gtk.TextView.get_buffer()* 来获取。

默认情况下，文本可以在 *Gtk.TextView* 中被添加、编辑与删除，你可以通过调用 *Gtk.TextView.set_editable()* 来禁止编辑。如果文本不可被编辑，你也通常想要使用 *Gtk.TextView.set_cursor_visible()* 来隐藏文本光标。在有些时候可以通过 *Gtk.TextView.set_justification()* 来设置文本的对齐方式很有用，文本可以显示为 左对齐(Gtk.Justification.LEFT)，右对齐(attr:*Gtk.Justification.RIGHT*)，或者中间对齐(attr:*Gtk.Justification.CENTER*)，或者占满整行(attr:*Gtk.Justification.FILL*)。

Gtk.TextView 的另一项默认设置是长行会一直水平延伸知道遇到一个断行。调用 *Gtk.TextView.set_wrap_mode()* 可以使文本自动换行一避免文本跑出屏幕边缘。

TextView 对象

class Gtk.**TextView**

创建一个新的 *Gtk.TextView* 并关联一个空的 *Gtk.TextBuffer* 。

get_buffer()

返回 text view 正在显示的 *Gtk.TextBuffer* 。

set_editable(*editable*)

设置 *Gtk.TextView* 的文本是否可编辑。

set_cursor_visible(*visible*)

设置现在光标的位置是否显示。不可编辑文本不应该有一个可见的光标，因此你 可能会关闭光标的显示。

set_justification(*justification*)

设置文本的额默认对齐方式。

justification 的值可以是下列之一：

- Gtk.Justification.LEFT: 文本左对齐。
- Gtk.Justification.RIGHT: 文本右对齐。
- Gtk.Justification.CENTER: 文本居中对齐。
- Gtk.Justification.FILL: 文本占满整行。

set_wrap_mode(*wrap_mode*)

设置view是否自动断行。

wrap_mode 的值可以是下列之一：

- Gtk.WrapMode.NONE: 不自动断行，文本域会一直随着文本变宽。
- Gtk.WrapMode.CHAR: 自动断行，可以光标可以出现的地方断行（通常字符之间）。

- `Gtk.WrapMode.WORD`: 自动断行，可以在单词之间断行。
- `Gtk.WrapMode.WORD_CHAR`: 自动断行，可以在单词之间断行，但如果 空间不够时Wrap text, 也可以在 graphemes 出断行。

模型（The Model）

Gtk.TextBuffer 是 *Gtk.TextView* 控件的核心，用来保存 *Gtk.TextView* 显示的文本。设置和获取文本内容可以他用过 *Gtk.TextBuffer.set_text()* 和 *Gtk.TextBuffer.get_text* 。 但是绝大多数的文本操作是通过 *iterators* 来完成的，即 *Gtk.TextIter* 。 iterator迭代器代表了文本buffer中两个字符之间的一个位置。迭代器并不是一直有效的， 一旦文本内容被修改并影响了buffer的内容，所有的迭代去就都无效了。

正因为此，迭代器不能用来在buffer修改前后保留位置。要保存一个位置，使用 *Gtk.TextMark* 。 一个text buffer包含两个内建的标记，”insert” 标记光标的位置，”selection_bound” 标记， 可以通过 *Gtk.TextBuffer.get_insert()* 和 *Gtk.TextBuffer.get_selection_bound()* 来获得他们。默认 *Gtk.TextMark* 的位置是不显示的，可以通过 *Gtk.TextMark.set_visible()* 设置。

有很多方法可以获取 *Gtk.TextIter* 。例如， *Gtk.TextBuffer.get_start_iter()* 返回只想text buffer地一个位置的迭代器，而 *Gtk.TextBuffer.get_end_iter()* 则返回 最后一个有效字符处的迭代器。获取选中文本的边界可以通过 *Gtk.TextBuffer.get_selection_bounds()* 。

要在一个指定位置插入文本请使用 *Gtk.TextBuffer.insert()* 。另一个非常有用的方法 *Gtk.TextBuffer.insert_at_cursor()* 可以在光标指向的位置插入文本。要删除一部分 文本请使用 *Gtk.TextBuffer.delete()* 。

另外， *Gtk.TextIter* 可以通过 *Gtk.TextIter.forward_search()* 和 *Gtk.TextIter.backward_search()* 用来搜索文本。根据需求可以使用buffer开始和结束 的iters来进行向前/后的搜索。

TextBuffer 对象

class Gtk.**TextBuffer**

set_text(*text*[, *length*])
删除buffer当前的内容，并插入 *length* 个 *text* 中的字符。 如果 *length* 为-1或忽略， *text* 全部被插入。

get_text(*start_iter*, *end_iter*, *include_hidden_chars*)
返回从 *start_iter* (包含)和 *end_iter* (不含)之间的文本，如果 *include_hidden_chars* 为 `False` ，则不包含未显示的文本。

get_insert()
返回代表当前光标位置(插入点)的 *Gtk.TextMark* 。

get_selection_bound()
返回代表选中区域边界的 *Gtk.TextMark* 。

create_mark(*mark_name*, *where*[, *left_gravity*])
在 *Gtk.TextIter* *where* 处创建一个 *Gtk.TextMark* 。 如果 *mark_name* 为 `None` ，则mark是匿名的。如果一个标记有left_gravity， 文本被插入到当前位置后，标记会移动到新插入文本的左边。如果标记为right_gravity (*left_gravity* 为 `False`)，标记会移动到新插入文本的右边。因此标准的 从左到右的光标为right gravity的标记（当你输入的时候，光标会出现在你输入文本的右边）。

如果 *left_gravity* 被忽略，默认为 `False` 。

get_mark(*mark_name*)
返回buffer中名字为mark_name的 :class‘Gtk.TextMark‘ ，如果不存在则返回 `None` 。

get_start_iter()
返回指向buffer地一个位置的 *Gtk.TextIter* 。

get_end_iter()
返回只想buffer最后一个有效字符的 *Gtk.TextIter* 。

get_selection_bounds()
返回包含两个 *Gtk.TextIter* tuple对象，分别指向选中的第一个字符和 后不第一个未选中的字符。如果没有文本被选中则返回空的tuple。

insert(*text_iter*, *text*[, *length*])
在 *text_iter* 处插入 *text* 的 *length* 个字符。如果 *length* 为-1或忽略， 全部的 *text* 会被插入。

insert_at_cursor(*text*[, *length*])
insert() 的简单调用，使用光标位置作为插入点。

delete(*start_iter*, *end_iter*)
删除 *start_iter* 与 *end_iter* 之间的文本。

create_tag(*tag_name*, ***kwargs*)
创建一个tag并添加到buffer的tag表中。

如果 *tag_name* 为 `None` ，则tag是匿名的，否则tag_name不能与tag表中 已经存在的tag重名。

kwargs 为任意数量的键值对，代表了tag的属性列表，可以通过 `tag.set_property` 来设置。

apply_tag(*tag*, *start_iter*, *end_iter*)
应用 *tag* 到给定范围的文本。

remove_tag(*tag*, *start_iter*, *end_iter*)
删除给定范围内所有的 *tag* 。

remove_all_tags(*start_iter*, *end_iter*)
删除给定范围内所有的tag。

class Gtk.**TextIter**

forward_search(*needle*, *flags*, *limit*)
向前搜索 *needle* 。搜索在达到limit后不会再继续。

flags 可以为下列之一，或者他们的组合（通过或操作符 `|` ）。

- 0: 精确匹配
- `Gtk.TextSearchFlags.VISIBLE_ONLY`: 匹配可能在 *needle* 中间穿插 有不可见字符，即 *needle* 为匹配到的字符串的可能不连续的子序列。
- `Gtk.TextSearchFlags.TEXT_ONLY`: 匹配可能包含图像pixbuf或者子空间 混合在匹配到的范围内。
- `Gtk.TextSearchFlags.CASE_INSENSITIVE`: 匹配忽略大小写。

返回包含指向开始与匹配到的文本后边的 *Gtk.TextIter* 的元组。如果 没有找到，则返回 `None` 。

backward_search(*needle*, *flags*, *limit*)
与 *forward_search()* 相同，但是向后搜索。

class Gtk.**TextMark**

set_visible(*visible*)
设置标记的可见性。插入点通常是可见的，即你可以看到一个竖直的光标条；而且， text控件也

会使用一个可见的标记来指示拖拽操作的释放点。绝大多数其他的比较 是不可见的。标记默认是不可见的。

Tags

buffer内的文本可以通过tag来标记。tag就是一个可以应用到一个文本范围的属性。例如， “bold” tag使应用到文本加粗。然而tag的概念比其更多，tag不一定会影响外观，也可能会 影响鼠标和按键的行为，”lock” 可以锁定一段文本使用户不能编辑，或者数不尽的其他的事情。 tag由 `Gtk.TextTag` 对象代表。一个 `Gtk.TextTag` 可以应用到任何数量 的文本范围，任何数量的buffer。

所有的tag都存储在 `Gtk.TextTagTable` 中。一个tab表定义了一系列的tag并可以一起使用。 每一个buffer都有一个与之关联的tag表，只有表中的tag才可以应用到buffer。但一个tag表可以 在多个buffer间共享。

要指定buffer内的一部分为本应该有特定的格式，你必须先定义该格式的tag，然后使用 *Gtk.TextBuffer.create_tag()* 和 *Gtk.TextBuffer.apply_tag()* 来将tag应用到文本区域。

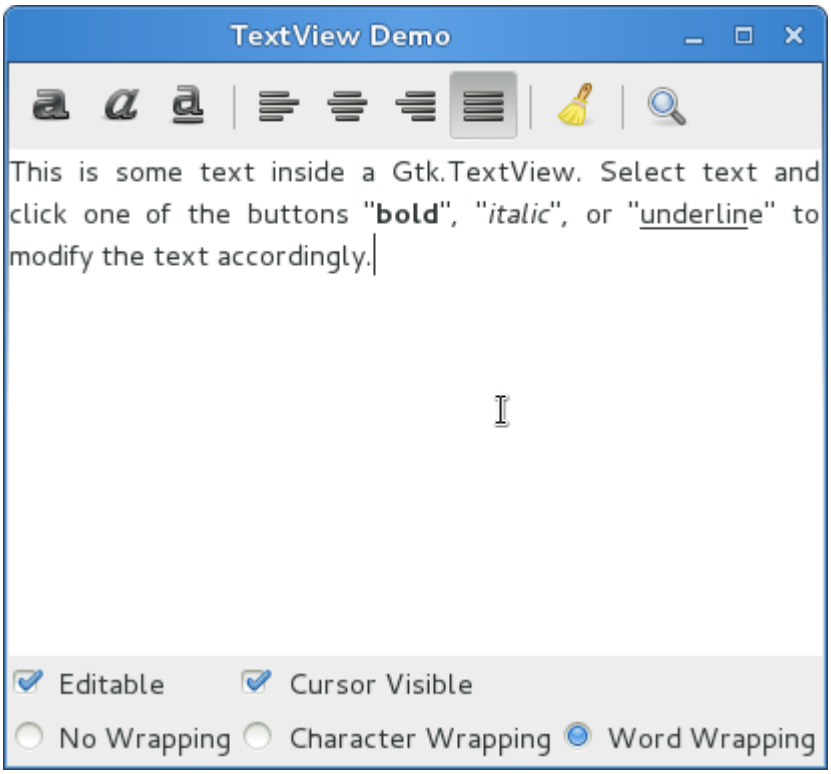
```
tag = textbuffer.create_tag("orange_bg", background="orange")
textbuffer.apply_tag(tag, start_iter, end_iter)
```

以下列出一些应用到文本的通常使用的风格:

- 背景颜色(“foreground” property)
- 前景颜色(“background” property)
- 下划线(“underline” property)
- 粗体(“weight” property)
- 斜提(“style” property)
- 删除线(“strikethrough” property)
- 对齐(“justification” property)
- 大小(“size” and “size-points” properties)
- 自动换行(“wrap-mode” property)

你也可以使用 *Gtk.TextBuffer.remove_tag()* 删除某个特定的tag，或者使用 *Gtk.TextBuffer.remove_all_tags()* 删除给的区域所有的tag。

Example




```
from gi.repository import Gtk, Pango

class SearchDialog(Gtk.Dialog):

    def __init__(self, parent):
        Gtk.Dialog.__init__(self, 'Search', parent,
                Gtk.DialogFlags.MODAL, buttons=(
                    Gtk.STOCK_FIND, Gtk.ResponseType.OK,
                    Gtk.STOCK_CANCEL, Gtk.ResponseType.CANCEL))

        box = self.get_content_area()

        label = Gtk.Label('Insert text you want to search for:')
        box.add(label)

        self.entry = Gtk.Entry()
        box.add(self.entry)

        self.show_all()

class TextViewWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='TextView Demo')

        self.set_default_size(-1, 350)

        self.grid = Gtk.Grid()
```

(下页继续)

(续上页)

```
        self.add(self.grid)

        self.create_textview()
        self.create_toolbar()
        self.create_buttons()

    def create_toolbar(self):
        toolbar = Gtk.Toolbar()
        self.grid.attach(toolbar, 0, 0, 3, 1)

        button_bold = Gtk.ToolButton.new_from_stock(Gtk.STOCK_BOLD)
        toolbar.insert(button_bold, 0)

        button_italic = Gtk.ToolButton.new_from_stock(Gtk.STOCK_ITALIC)
        toolbar.insert(button_italic, 1)

        button_underline = Gtk.ToolButton.new_from_stock(Gtk.STOCK_UNDERLINE)
        toolbar.insert(button_underline, 2)

        button_bold.connect('clicked', self.on_button_clicked, self.tag_bold)
        button_italic.connect('clicked', self.on_button_clicked, self.tag_italic)
        button_underline.connect('clicked', self.on_button_clicked, self.tag_
↪underline)

        toolbar.insert(Gtk.SeparatorToolItem(), 3)

        radio_justifyleft = Gtk.RadioToolButton()
        radio_justifyleft.set_stock_id(Gtk.STOCK_JUSTIFY_LEFT)
        toolbar.insert(radio_justifyleft, 4)

        radio_justifycenter = Gtk.RadioToolButton.new_with_stock_from_widget(
                radio_justifyleft, Gtk.STOCK_JUSTIFY_CENTER)
        toolbar.insert(radio_justifycenter, 5)

        radio_justifyright = Gtk.RadioToolButton.new_with_stock_from_widget(
                radio_justifyleft, Gtk.STOCK_JUSTIFY_RIGHT)
        toolbar.insert(radio_justifyright, 6)

        radio_justifyfill = Gtk.RadioToolButton.new_with_stock_from_widget(
                radio_justifyleft, Gtk.STOCK_JUSTIFY_FILL)
        toolbar.insert(radio_justifyfill, 7)

        radio_justifyleft.connect('toggled', self.on_justify_toggled,
                Gtk.Justification.LEFT)
        radio_justifycenter.connect('toggled', self.on_justify_toggled,
                Gtk.Justification.CENTER)
        radio_justifyright.connect('toggled', self.on_justify_toggled,
                Gtk.Justification.RIGHT)
        radio_justifyfill.connect('toggled', self.on_justify_toggled,
                Gtk.Justification.FILL)

        toolbar.insert(Gtk.SeparatorToolItem(), 8)

        button_clear = Gtk.ToolButton.new_from_stock(Gtk.STOCK_CLEAR)
        button_clear.connect('clicked', self.on_clear_clicked)
        toolbar.insert(button_clear, 9)
```

(下页继续)

(续上页)

```
        toolbar.insert(Gtk.SeparatorToolItem(), 10)

        button_search = Gtk.ToolButton.new_from_stock(Gtk.STOCK_FIND)
        button_search.connect('clicked', self.on_search_clicked)
        toolbar.insert(button_search, 11)

    def create_textview(self):
        scrolledwindow = Gtk.ScrolledWindow()
        scrolledwindow.set_hexpand(True)
        scrolledwindow.set_vexpand(True)
        self.grid.attach(scrolledwindow, 0, 1, 3, 1)

        self.textview = Gtk.TextView()
        self.textbuffer = self.textview.get_buffer()
        self.textbuffer.set_text('This is some text inside a Gtk.TextView. '
                + 'Select text and click one of the buttons "bold", "italic", '
                + 'or "underline" to modify the text accordingly.')
        scrolledwindow.add(self.textview)

        self.tag_bold = self.textbuffer.create_tag('bold',
                weight=Pango.Weight.BOLD)
        self.tag_italic = self.textbuffer.create_tag('italic',
                style=Pango.Style.ITALIC)
        self.tag_underline = self.textbuffer.create_tag('underline',
                underline=Pango.Underline.SINGLE)
        self.tag_found = self.textbuffer.create_tag('found',
                background='yellow')

    def create_buttons(self):
        check_editable = Gtk.CheckButton('Editable')
        check_editable.set_active(True)
        check_editable.connect('toggled', self.on_editable_toggled)
        self.grid.attach(check_editable, 0, 2, 1, 1)

        check_cursor = Gtk.CheckButton('Cursor Visible')
        check_cursor.set_active(True)
        check_cursor.connect('toggled', self.on_cursor_toggled)
        self.grid.attach_next_to(check_cursor, check_editable,
                Gtk.PositionType.RIGHT, 1, 1)

        radio_wrapnone = Gtk.RadioButton.new_with_label_from_widget(None,
                'No Wrapping')
        self.grid.attach(radio_wrapnone, 0, 3, 1, 1)

        radio_wrapchar = Gtk.RadioButton.new_with_label_from_widget(
                radio_wrapnone,'Character Wrapping')
        self.grid.attach_next_to(radio_wrapchar, radio_wrapnone,
                Gtk.PositionType.RIGHT, 1, 1)

        radio_wrapword = Gtk.RadioButton.new_with_label_from_widget(
                radio_wrapnone, 'Word Wrapping')
        self.grid.attach_next_to(radio_wrapword, radio_wrapchar,
                Gtk.PositionType.RIGHT, 1, 1)

        radio_wrapnone.connect('toggled', self.on_wrap_toggled, Gtk.WrapMode.NONE)
```

(下页继续)

(续上页)

```
        radio_wrapchar.connect('toggled', self.on_wrap_toggled, Gtk.WrapMode.CHAR)
        radio_wrapword.connect('toggled', self.on_wrap_toggled, Gtk.WrapMode.WORD)

    def on_button_clicked(self, widget, tag):
        bounds = self.textbuffer.get_selection_bounds()
        if len(bounds) != 0:
            start, end = bounds
            self.textbuffer.apply_tag(tag, start, end)

    def on_clear_clicked(self, widget):
        start = self.textbuffer.get_start_iter()
        end = self.textbuffer.get_end_iter()
        self.textbuffer.remove_all_tags(start, end)

    def on_editable_toggled(self, widget):
        self.textview.set_editable(widget.get_active())

    def on_cursor_toggled(self, widget):
        self.textview.set_cursor_visible(widget.get_active())

    def on_wrap_toggled(self, widget, mode):
        self.textview.set_wrap_mode(mode)

    def on_justify_toggled(self, widget, justification):
        self.textview.set_justification(justification)

    def on_search_clicked(self, widget):
        dialog = SearchDialog(self)
        response = dialog.run()
        if response == Gtk.ResponseType.OK:
            cursor_mark = self.textbuffer.get_insert()
            start = self.textbuffer.get_iter_at_mark(cursor_mark)
            if start.get_offset() == self.textbuffer.get_char_count():
                start = self.textbuffer.get_start_iter()

            self.search_and_mark(dialog.entry.get_text(), start)

        dialog.destroy()

    def search_and_mark(self, text, start):
        end = self.textbuffer.get_end_iter()
        match = start.forward_search(text, 0, end)

        if match != None:
            match_start, match_end = match
            self.textbuffer.apply_tag(self.tag_found, match_start, match_end)
            self.search_and_mark(text, match_end)

win = TextViewWindow()
win.connect('delete-event', Gtk.main_quit)
win.show_all()
Gtk.main()
```

4.1.16 菜单

GTK+有两种不同的菜单， `Gtk.MenuBar` 和 `Gtk.Toolbar` 。 `Gtk.MenuBar` 是标准的菜单条，包含一个或多个 `Gtk.MenuItem` 或其子类的实例。 `Gtk.Toolbar` 控件用于可以快速地访问应用程序经常使用的功能，其包含一个多个 `Gtk.ToolItem` 或其子类的实例。

Actions

尽管有相应的API来创建菜单和工具条，你应该使用 *Gtk.UIManager* 并创建 *Gtk.Action* 的实例。action被组织为组， *Gtk.ActionGroup* 实质上就是 从名字到 *Gtk.Action* 对象的映射。所有要用在某个特定上下文的action都应该 放在一个组中。多个action group可以用于特殊的用户界面，通常非一般的程序会使用多个组。 例如，在一个可以编辑多个文档的应用程序中，一组处理全局的action（如退出、关于、新建）， 另一组每文档相关的则处理该文档的action（如保存、剪切/复制/粘贴）。没一个窗口的action 都应该由这两个action group组合而成。

不同的类代表了action不同的种类:

- *Gtk.Action*: 一个可以通过菜单或工具条项目触发的action。
- *Gtk.ToggleAction*: 一个可以通过在两种状态间切换触发的action。
- *Gtk.RadioAction*: 在一组中只有一个可以激活的action。
- `Gtk.RecentAction`: 代表一个最近使用文件的列表的action。

action代表用户可以执行的操作及其如何呈现的一些信息。这些信息包括name（不是用来显示），label（用于显示），加速键，label是否引用 *stock item* ， 提到信息及 action被激活时的回调函数。

要创建action，你可以直接调用构造函数，或者通过调用 *Gtk.ActionGroup.add_action()* 或 `Gtk.ActionGrou.add_action_with_accel()` 或以下便捷函数之一将action加入到action group中:

- *Gtk.ActionGroup.add_actions()*,
- *Gtk.ActionGroup.add_toggle_actions()*
- *Gtk.ActionGroup.add_radio_actions()* 。

注意你必须指定菜单项和子菜单的action。

Action 对象

class `Gtk.`**Action**(*name*, *label*, *tooltip*, *stock_id*)
name 必须唯一。

如果 *label* 不为 `None` ， 将会在菜单项和按钮中显示。

如果 *tooltip* 不为 `None` ， 将会用于action的提示信息。

如果 *stock_id* 不为 `None` ， 将会用于查找 *stock item* 代表action在控件中显示。

class `Gtk.`**ToggleAction**(*name*, *label*, *tooltip*, *stock_id*)
参数与 *Gtk.Action* 的构造函数相同。

class `Gtk.`**RadioAction**(*name*, *label*, *tooltip*, *stock_id*, *value*)
前四个参数与 *Gtk.Action* 的构造函数相同。

value 代表当action被选中时 *get_current_value()* 返回的值。

get_current_value()
获取action所属组中当前激活的项目的 “value” 属性的值。

join_group(*group_source*)
radio action对象加入 *group_source* radio action对象所属的组。

group_source 必须我们要加入的组的一个 radio action 对象。 或者传递 None 来将其移除组外。

class Gtk.**ActionGroup**(*name*)
创建一个新的 *Gtk.ActionGroup* 实例。action group 的名字用于与action关联健绑定。

add_action(*action*)
添加一个 *Gtk.Action* 对象到组中。

注意本方法不会设置action的加速键，如需要请使用 *add_action_with_accel()* 代替。

add_action_with_accel(*action*, *accelerator*)
添加一个 *Gtk.Action* 对象到action group中并设置加速键。

accelerator 必须是 *Gtk.accelerator_parse()* 可以解析的格式，或者 "" 不设置加速键，或者 None 使用stock的加速键。

add_actions(*entries*[, *user_data*])
这是一个便捷函数用来创建多个 *Gtk.Action* 对象并将他们加入到action group中。

entries 是一个包含一到六个以下元素的元组的列表:

- action的name(必须的)
- action 的 *stock item* (默认为 None)
- action 的lable (默认为 None)
- action 的加速键，格式为 *Gtk.accelerator_parse()* 可以识别的格式(默认为 None)
- action 的提示信息。 (默认为 None)
- action激活时调用的回调函数。(默认为 None)

aciton 的 “activate” 信号会与回调函数连接。

如果 *user_data* 不为 None ，将会传递给毁掉函数（如果回调函数指定）。

add_toggle_actions(*entries*[, *user_data*])
创建多个 *Gtk.ToggleAction* 对象被添加到本组中的便捷函数。

entries 是一个包含一到七个以下元素的元组的列表:

- action的name(必须的)
- action 的 *stock item* (默认为 None)
- action 的lable (默认为 None)
- action 的加速键，格式为 *Gtk.accelerator_parse()* 可以识别的格式(默认为 None)
- action 的提示信息。 (默认为 None)
- action激活时调用的回调函数。(默认为 None)
- 一个表示本toggle action是否激活的布尔值。(默认为 False)

aciton 的 “activate” 信号会与回调函数连接。

如果 *user_data* 不为 None ，将会传递给毁掉函数（如果回调函数指定）。

add_radio_actions(*entries*[, *value*[, *on_change*[, *user_data*]]])
创建多个 *Gtk.RadioAction* 并添加到本组中的便捷函数。

entries 是一个包含一到六个以下元素的元组的列表:

- action的name(必须的)

- action 的 *stock item* (默认为 None)
- action 的lable (默认为 None)
- action 的加速键，格式为 *Gtk.accelerator_parse()* 可以识别的格式(默认为 None)
- action 的提示信息。 (默认为 None)
- radio action 的值 (默认为 0)

value 指定应该被激活的radio action。

如何 *on_change* 指定，其会被连接到第一个radio action的 “changed” 信号。

如果 *user_data* 不为 None ，将会传递给毁掉函数（如果回调函数指定）。

Gtk.**accelerator_parse**(*accelerator*)

解析代表加速键的字符串。格式类似于 “<Control>a” 或者 “<Shift><Alt>F1” 或者 “<Release>z” (最后一个为按钮释放)。解析器是相当自由的，允许大小写及 “<Ctl>” and “<Ctrl>” 这样的字符。对于character key，名字不是其符号，而是小写的英文，例如： 应该使用 “<Ctrl>minus” 而不是 “<Ctrl>-” 。

返回 (accelerator_key, accelerator_mods) 元组，其中后者代表 accelerator modifier mask 地一个代表加速键的值。如果解析失败两者均返回0 。

UI Manager

Gtk.UIManager 提供了一种简单的方式以使用一种 类似与XML的描述 来创建菜单和工具条。

首先，你先要使用 *Gtk.UIManager.insert_action_group()* 添加 *Gtk.ActionGroup* 到 UI Manage。此时最好调用 *Gtk.UIManager.get_accel_group()* 和 Gtk.window.add_accel_group() 来通知父窗口响应指定的快捷键。

然后你就可以定义菜单和工具条的可见的布局信息并添加了。”ui string” 使用UML格式，其中你应该指定你已经创建的action的name。记住这些名字只是标识符而已，他们并不是 用户在菜单和工具条中看到的文本。我们创建action时一般会使用能表达其意思的name。

最后，你通过 *Gtk.UIManager.get_widget()* 获取root widget并将其添加到 *Gtk.Box* 之类的容器中。

UIManager Objects

class Gtk.**UIManager**

insert_action_group(*action_group*[, *pos*])

将action_group插入与mananger关联的action group组的列表中。之前组中的action会 隐藏后面组中相同name的action。

pos 为组插入的位置，如果忽略则会被追加到最后面。

get_accel_group()

返回与此manage相关联的全局加速键。

get_widget(*path*)

根据path查找控件，path包含UI的XML描述字符串中指定的name。 XML中不包含name或者action属性的元素(例如 <popup>)可以通过XML的以 ‘/’ 分隔的标记名 (如 <popup>)来标识。根标记(“/ui”)可以被忽略。

返回根据 *path* 找到的控件，如果没找到则返回 None 。

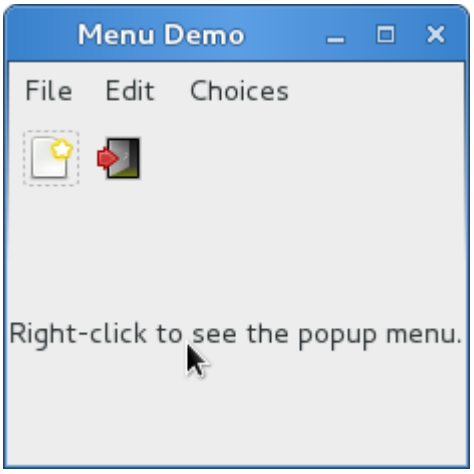
add_ui_from_string(*text*)

解析 *text* 包含的 UI 定义 并与当前manager的内容合并。如果没有闭合的<ui>元素，则会自动添加一个。

返回合并后的UI的id。

如果发生错误则抛出异常。

Example




```
from gi.repository import Gtk,Gdk

UI_INFO = '''
<ui>
  <menubar name='MenuBar'>
    <menu action='FileMenu'>
      <menu action='FileNew'>
        <menuitem action='FileNewStandard' />
        <menuitem action='FileNewFoo' />
        <menuitem action='FileNewGoo' />
      </menu>
      <separator />
      <menuitem action='FileQuit' />
    </menu>

    <menu action='EditMenu'>
      <menuitem action='EditCut' />
      <menuitem action='EditCopy' />
      <menuitem action='EditPaste' />
      <menuitem action='EditSomething' />
    </menu>

    <menu action='ChoicesMenu'>
      <menuitem action='ChoiceOne' />
      <menuitem action='ChoiceTwo' />
      <separator />
      <menuitem action='ChoiceThree' />
    </menu>
  </menubar>

```

(下页继续)

(续上页)

```
  <toolbar name='ToolBar'>
    <toolitem action='FileNewStandard' />
    <toolitem action='FileQuit' />
  </toolbar>

  <popup name='PopupMenu'>
    <menuitem action='EditCut' />
    <menuitem action='EditCopy' />
    <menuitem action='EditPaste' />
  </popup>
</ui>
'''

class MenuExampleWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='Menu Demo')

        self.set_default_size(200, 200)

        action_group = Gtk.ActionGroup('my_actions')

        self.add_file_menu_actions(action_group)
        self.add_edit_menu_actions(action_group)
        self.add_choices_menu_actions(action_group)

        uimanager = self.create_ui_manager()
        uimanager.insert_action_group(action_group)

        menubar = uimanager.get_widget('/MenuBar')

        box = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
        box.pack_start(menubar, False, False, 0)

        toolbar = uimanager.get_widget('/ToolBar')
        box.pack_start(toolbar, False, False, 0)

        eventbox = Gtk.EventBox()
        eventbox.connect('button-press-event', self.on_button_press_event)
        box.pack_start(eventbox, True, True, 0)

        label = Gtk.Label('Right-click to see the popup menu.')
        eventbox.add(label)

        self.popup = uimanager.get_widget('/PopupMenu')

        self.add(box)

    def add_file_menu_actions(self, action_group):
        action_filemenu = Gtk.Action('FileMenu', 'File', None, None)
        action_group.add_action(action_filemenu)

        action_filenewmenu = Gtk.Action('FileNew', None, None, Gtk.STOCK_NEW)
        action_group.add_action(action_filenewmenu)

        action_new = Gtk.Action('FileNewStandard', '_New',
```

(下页继续)

(续上页)

```
                'Create a new file', Gtk.STOCK_NEW)
        action_new.connect('activate', self.on_menu_file_new_generic)
        action_group.add_action_with_accel(action_new, None)

        action_group.add_actions([
            ('FileNewFoo', None, 'New Foo', None, 'Create new foo',
                self.on_menu_file_new_generic),
            ('FileNewGoo', None, '_New Goo', None, 'Create new goo',
                self.on_menu_file_new_generic),
            ])

        action_filequit = Gtk.Action('FileQuit', None, None, Gtk.STOCK_QUIT)
        action_filequit.connect('activate', self.on_menu_file_quit)
        action_group.add_action(action_filequit)

    def add_edit_menu_actions(self, action_group):
        action_group.add_actions([
            ('EditMenu', None, 'Edit'),
            ('EditCut', Gtk.STOCK_CUT, None, None, None, self.on_menu_others),
            ('EditCopy', Gtk.STOCK_COPY, None, None, None, self.on_menu_others),
            ('EditPaste', Gtk.STOCK_PASTE, None, None, None, self.on_menu_others),
            ('EditSomething', None, 'Something', '<control><alt>S', None, self.on_
↪menu_others),
            ])

    def add_choices_menu_actions(self, action_group):
        action_group.add_action(Gtk.Action('ChoicesMenu', 'Choices', None, None))

        action_group.add_radio_actions([
            ('ChoiceOne', None, 'One', None, None, 1),
            ('ChoiceTwo', None, 'Two', None, None, 2)
            ], 1, self.on_menu_choices_changed)

        three = Gtk.ToggleAction('ChoiceThree', 'Three', None, None)
        three.connect('toggled', self.on_menu_choices_toggled)
        action_group.add_action(three)

    def create_ui_manager(self):
        uimanager = Gtk.UIManager()

        # throw exception if Something went wrong
        uimanager.add_ui_from_string(UI_INFO)

        # add the accelerator group to the toplevel window
        accelgroup = uimanager.get_accel_group()
        self.add_accel_group(accelgroup)
        return uimanager

    def on_menu_file_new_generic(self, widget):
        print 'A File|New menu iter was selected.'

    def on_menu_file_quit(self, widget):
        Gtk.main_quit()

    def on_menu_others(self, widget):
        print 'Menu item' + widget.get_name() + 'was selected'
```

(下页继续)

(续上页)

```
142
143     def on_menu_choices_changed(self, widget, current):
144         print current.get_name() + 'was selected.'
145
146     def on_menu_choices_toggled(self, widget):
147         if widget.get_active():
148             print widget.get_name() + 'activated'
149         else:
150             print widget.get_name() + 'deactivated'
151
152     def on_button_press_event(self, widget, event):
153         # check if Right mouse button was pressed
154         if event.type == Gdk.EventType.BUTTON_PRESS and event.button == 3:
155             self.popup.popup(None, None, None, None, event.button, event.time)
156             return True # evnet has been handled
157
158 win = MenuExampleWindow()
159 win.connect('delete-event', Gtk.main_quit)
160 win.show_all()
161 Gtk.main()
```

4.1.17 对话框（Dialog）

对话框窗口与标准的窗口非常相似，用来向用户显示信息或者从用户那里获取信息，例如 通常用于提供一个首选项窗口。对话框主要的不同是一些预打包好的控件自动布局在窗口中。 我们可以简单地添加标签label、按钮、复选按钮等等，另一个很大的不同是响应的处理， 控制应用在用户与对话框的交互之后如何处理。

有很多你可能会觉得很有用的派生的对话框。 *Gtk.MessageDialog* 用于绝大多数 的提示信息。但是其他的时候你肯呢个需要派生你自己的对话框类来提供更加复杂的功能。

自定义对话框

要打包控件到一个自定义的对话框，你应该把他们打包到一个 *Gtk.Box* 中，可以 通过 *Gtk.Dialog.get_content_area()* 获取box。要简单的添加一个按钮到对话框的 底部，你可以使用 *Gtk.Dialog.add_button()* 。

可以通过 *Gtk.Dialog.set_modal* 或者在 *Gtk.Dialog* 的构造函数的 `flag` 参数上包含 `Gtk.DialogFlags.MODAL` 创建 '模态' 对话框 （即会冻结对应用其他部分的反应）。

点击按钮会触发一个 "response" 信号，如果你想要阻塞等待对话框返回后再执行你的 控制流程，你可以调用 *Gtk.Dialog.run()* ，该方法返回一个整型，可能是 `Gtk.ResponseType` ，也可能是你在 *Gtk.Dialog* 的构造函数或 *Gtk.Dialog.add_button()* 方法中自定义的返回值。

最后，有两种方法可以删除对话框， `Gtk.Widget.hide()` 从视图中移除对话框，但是 该对话框仍然在内存中，这通常在后边仍然需要显示的对话框很有用，可以阻止对此构造。 `Gtk.Widget.destroy()` 方法则会在你不需要对话框时从内存中删除它，注意如果在 销毁后如果需要再次访问该对话框你必须重新构造，否则对话框窗口是空的。

Dialog 对象

class `Gtk.`**Dialog**(*[title[, parent[, flags[, buttons]]]]*)

创建一个新的 *Gtk.Dialog* ，标题设置为 *title* 并将临时的父窗口设置为 *parent* 。 *flags* 参数可

以用来设置对话框为模态(Gtk.DialogFlags.MODAL) 和/或 设置为与其父窗口一起销毁(Gtk.DialogFlags.DESTROY_WITH_PARENT)。

buttons 为可以提供一个按钮和响应的不同按钮的元组。详情参考 *add_button()* 。

所有参数都是可选的，也可以用作key-word关键字参数。

get_content_area()
返回对话框的内容区域。

add_button(*button_text*, *response_id*)
根据给定的文本添加一个按钮(或者是stock button，如果 *button_text* 为 *stock item*)，并设置当点击按钮时会触发 "响应" 信号 并传递 *response_id* ，按钮会被添加都对话框活动区域的最后。

response_id 可以为任何正整数或者以下 Gtk.ResponseType 预定义的值：

- Gtk.ResponseType.NONE
- Gtk.ResponseType.REJECT
- Gtk.ResponseType.ACCEPT
- Gtk.ResponseType.DELETE_EVENT
- Gtk.ResponseType.OK
- Gtk.ResponseType.CANCEL
- Gtk.ResponseType.CLOSE
- Gtk.ResponseType.YES
- Gtk.ResponseType.NO
- Gtk.ResponseType.APPLY
- Gtk.ResponseType.HELP

创建的按钮控件会被返回，但通常用不到。

add_buttons(*button_text*, *response_id*[, ...])
添加多个按钮到对话框，使用传递的按钮数据参数。本方法与重复调用 *add_button()* 方法一样。按钮数据对——按钮文本(或者 *stock item*) 和响应id是分别传递的，例如:

```
dialog.add_buttons(Gtk.STOCK_OPEN, 42, "Close", Gtk.ResponseType.CLOSE)
```

set_modal(*is_modal*)
设置对话框为模态或非模态。模态对话框用户与应用程序的其他窗口交互。

run()
阻塞于一个递归的循环直到对话框或者触发 "response" 信号或者被销毁。如果 对话框在调用 *run()* 期间被销毁，则会返回 Gtk.ResponseType.NONE 。 否则会返回对信号的回应的响应id。

Example


```
from gi.repository import Gtk

class DialogExample(Gtk.Dialog):

    def __init__(self, parent):
        Gtk.Dialog.__init__(self, 'My Dialog', parent, 0,
                (Gtk.STOCK_CANCEL, Gtk.ResponseType.CANCEL,
                 Gtk.STOCK_OK, Gtk.ResponseType.OK))

        self.set_default_size(150, 100)

        label = Gtk.Label('This is a dialog to display addtiional infomation')

        box = self.get_content_area()
        box.add(label)
        self.show_all()

class DialogWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='Dialog Example')

        self.set_border_width(6)

        button = Gtk.Button('Open dialog')
        button.connect('clicked', self.on_button_clicked)

        self.add(button)

    def on_button_clicked(self, widget):
        dialog = DialogExample(self)
        response = dialog.run()

        if response == Gtk.ResponseType.OK:
            print 'The OK button was clicked'
        elif response == Gtk.ResponseType.CANCEL:
            print 'The Cancel button was clicked'

        dialog.destroy()

win = DialogWindow()
win.connect('delete-event', Gtk.main_quit)
win.show_all()
Gtk.main()
```

消息对话框

Gtk.MessageDialog 是一个便利类，用于创建简单的，标准的消息对话框，带有 一个消息，一个图标，及用于用户响应的按钮。你可以在 *Gtk.MessageDialog* 的 构造函数中指定消息和文本的类型，也可以指定标准的按钮。

有一些对话框需要更多的描述到底发生了什么，此时可以添加次要的文本信息。这时创建 消息对话框时指定的主要的文本信息会设置为更大并为粗体。次要的消息可以通过 *Gtk.MessageDialog.format_secondary_text()* 指定。

MessageDialog 对象

class Gtk.**MessageDialog**(*[parent[, flags[, message_type[, buttons, [message_format]]]]]*)

创建一个新的 *Gtk.MessageDialog* ， 将临时的父窗口设置为 *parent* 。 *flags* 参数可以用来设置对话框为模态(`Gtk.DialogFlags.MODAL`) 和/或 设置为与其父窗口一起销毁(`Gtk.DialogFlags.DESTROY_WITH_PARENT`) 。

message_type 可以设置为以下之一:

- `Gtk.MessageType.INFO`: 提示消息
- `Gtk.MessageType.WARNING`: 非致命警告信息
- `Gtk.MessageType.QUESTION`: 需要用户选择的问题消息
- `Gtk.MessageType.ERROR`: 致命的错误
- `Gtk.MessageType.OTHER`: 非以上的，不会获取图标

也可以给消息对话框设置多种的按钮，来从用户获取不同的响应，可以使用以下值之一:

- `Gtk.ButtonsType.NONE`: 没有按钮
- `Gtk.ButtonsType.OK`: 确定按钮
- `Gtk.ButtonsType.CLOSE`: 关闭按钮
- `Gtk.ButtonsType.CANCEL`: 取消按钮
- `Gtk.ButtonsType.YES_NO`: 是和否按钮
- `Gtk.ButtonsType.OK_CANCEL`: 确定和取消按钮

最后， *message_format* 是用户要看到的文本信息。

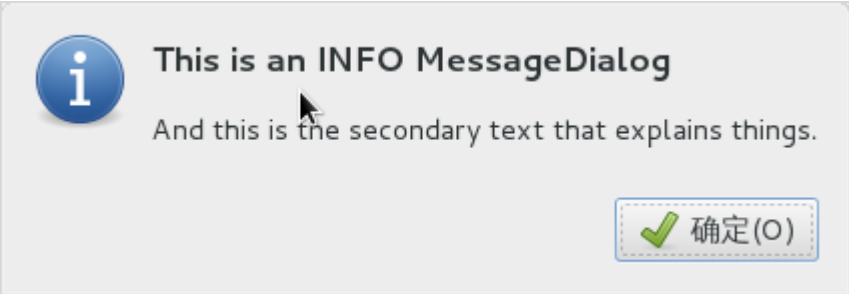
所有参数都是可选的，也可以用作key-word关键字参数。

format_secondary_text(*message_format*)

设置消息对话框的次要文本信息为 *message_format* 。

注意设置次要文本会使主要的文本(*Gtk.MessageDialog* 构造函数的 *message_format* 参数)变为粗体，除非你提供了明确的标记。

Example




```
from gi.repository import Gtk

class MessageDialogWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='MessageDialog Example')

        box = Gtk.Box(spacing=6)
        self.add(box)

        button1 = Gtk.Button('Infomation')
        button1.connect('clicked', self.on_info_clicked)
        box.add(button1)

        button2 = Gtk.Button('Error')
        button2.connect('clicked', self.on_error_clicked)
        box.add(button2)

        button3 = Gtk.Button('Warning')
        button3.connect('clicked', self.on_warn_clicked)
        box.add(button3)

        button4 = Gtk.Button('Question')
        button4.connect('clicked', self.on_question_clicked)
        box.add(button4)

    def on_info_clicked(self, widget):
        dialog = Gtk.MessageDialog(self, 0, Gtk.MessageType.INFO,
                Gtk.ButtonsType.OK, 'This is an INFO MessageDialog')
        dialog.format_secondary_text('And this is the secondary text that explains␣
↪things.')
        dialog.run()
        print 'INFO dialog closed'

        dialog.destroy()

    def on_error_clicked(self, widget):
        dialog = Gtk.MessageDialog(self, 0, Gtk.MessageType.ERROR,
                Gtk.ButtonsType.OK, 'This is an ERROR MessageDialog')
        dialog.format_secondary_text('And this is the secondary text that explains␣
↪things.')
        dialog.run()
        print 'ERROR dialog closed'

```

(下页继续)

(续上页)

```
        dialog.destroy()

    def on_warn_clicked(self, widget):
        dialog = Gtk.MessageDialog(self, 0, Gtk.MessageType.WARNING,
                Gtk.ButtonsType.OK_CANCEL, 'This is an WARNING MessageDialog')
        dialog.format_secondary_text('And this is the secondary text that explains␣
↪things.')
        response = dialog.run()
        if response == Gtk.ResponseType.OK:
            print 'WARN dialog closed by clicking OK button'
        elif response == Gtk.ResponseType.CANCEL:
            print 'WARN dialog closed by clicking CANCEL button'

        dialog.destroy()

    def on_question_clicked(self, widget):
        dialog = Gtk.MessageDialog(self, 0, Gtk.MessageType.QUESTION,
                Gtk.ButtonsType.YES_NO, 'This is an QUESTION MessageDialog')
        dialog.format_secondary_text('And this is the secondary text that explains␣
↪things.')
        response = dialog.run()
        if response == Gtk.ResponseType.YES:
            print 'QUESTION dialog closed by clicking YES button'
        elif response == Gtk.ResponseType.NO:
            print 'QUESTION dialog closed by clicking NO button'

        dialog.destroy()

win = MessageDialogWindow()
win.connect('delete-event', Gtk.main_quit)
win.show_all()
Gtk.main()
```

文件选择对话框

Gtk.FileChooserDialog 用于 “文件/打开” 或者 “文件/保存” 菜单项很合适。 对于文件选择对话框你可以使用所有 *Gtk.FileChooser* 和 *Gtk.Dialog* 的方法。

当创建 *Gtk.FileChooserDialog* 时你需要定义对话框的目的:

- 要选择一个用于打开的问，用于文件/打开命令，使用 `Gtk.FileChooserAction.OPEN`
- 要第一次保存一个文件，用于文件/保存命令，使用 `Gtk.FileChooserAction.SAVE`， 并使用 *Gtk.FileChooser.set_current_name()* 指定一个建议的名字例如 “Untitled”
- 要保存一个文件为不同的名字，用于文件/另存为命令，使用 `Gtk.FileChooserAction.SAVE`，并且通过 *Gtk.FileChooser.set_filename()* 设置已存在的文件的名字。
- 要选择一个文件夹而不是文件，使用 `Gtk.FileChooserAction.SELECT_FOLDER`.

Gtk.FileChooserDialog 继承自 *Gtk.Dialog*， 因此按钮也有响应id如 `Gtk.ResponseType.ACCEPT` 和 `Gtk.ResponseType.CANCEL`， 这些也都可以 在 *Gtk.FileChooserDialog* 的构造函数中指定。与 *Gtk.Dialog* 类似， 你可以使用自定义的响应id，文件选择对话框应该至少包含以下id的按钮之一:

- `Gtk.ResponseType.ACCEPT`
- `Gtk.ResponseType.OK`

- Gtk.ResponseType.YES
- Gtk.ResponseType.APPLY

当用户完成文件的选择，你的程序可以获取选择的文件的文件名(*Gtk.FileChooser.get_filename()*) 或者URI(*Gtk.FileChooser.get_uri()*)。

默认 *Gtk.FileChooser* 一次只允许选择一个文件，要使用多选可以调用 *Gtk.FileChooser.set_select_multiple()*。获取选择的文件的列表可以使用 *Gtk.FileChooser.get_filenames()* 或者 *Gtk.FileChooser.get_uris()*。

Gtk.FileChooser 也支持各种选项使得文件和目录更加的可配置和访问。

- *Gtk.FileChooser.set_local_only()*: 只有本地文件可以选择。
- Gtk.FileChooser.show_hidden(): 显示隐藏文件。
- *Gtk.FileChooser.set_do_overwrite_confirmation()*: 如果文件选择对话框被配置为 Gtk.FileChooserAction.SAVE 模式，当用户输入的文件名已存在时，会显示一个确认对话框。

另外，你可以通过创建 *Gtk.FileFilter* 对象并调用 *Gtk.FileChooser.add_filter()* 来指定显示那些类型的文件。用户可以在文件选择对话框的底部的组合框选择添加的过滤器。 of the file chooser.

FileChooser 对象

class Gtk.**FileChooserDialog**(*[title[, parent[, action[, buttons]]]]*)

创建一个新的 *Gtk.FileChooserDialog* 并设置标题为 *title*，临时的父窗口为 *parent*。

action 可以为一下之一:

- Gtk.FileChooserAction.OPEN: 只允许用回选择一个已经存在的文件。
- Gtk.FileChooserAction.SAVE: 允许用户使用一个已经存在的文件的名字或者输入一个新的文件名。
- Gtk.FileChooserAction.SELECT_FOLDER: 允许用户选择一个已经存在的目录。
- Gtk.FileChooserAction.CREATE_FOLDER: 允许用户命名一个新的或已经存在的目录。

buttons 参数与 *Gtk.Dialog* 的格式一样。

class Gtk.**FileChooser**

set_current_name(*name*)

设置文件选择框当前文件的名字，就像用户输入的一样。

set_filename(*filename*)

设置 *filename* 为文件选择框当前的文件名，改变到文件的父目录并选择列表中的 文件，其他的文件都不会被选择。 如果选择器为 Gtk.FileChooserAction.SAVE 模式，文件的基本名页会显示 在对话框文件名输入框。

注意文件必须存在，否则除了改变目录其他什么也不做。

set_select_multiple(*select_multiple*)

设置可以选择多个文件，只在 Gtk.FileChooserAction.OPEN 模式或 Gtk.FileChooserAction.SELECT_FOLDER 有效。

set_local_only(*local_only*)

设置是否只可以选择本地文件。

set_show_hidden(*show_hidden*)
设置是否显示隐藏的文件和目录。

set_do_overwrite_confirmation(*do_overwrite_confirmation*)
设置在保存模式是否提示覆盖。

get_filename()
返回文件选择框当前选中的文件的名字。如果选中了多个文件，请使用 *get_filenames()* 代替。

get_filenames()
返回在当前文件夹选中的文件和子目录的列表。返回的名字是绝对路径，如果当前 目录的文件不是本地文件则会被忽略，要获取请使用 *get_uris()* 来代替。

get_uri()
返回文件选择框当前选中的文件的URI。如果多个文件被选中，请使用 *get_uris()* 代替。

get_uris()
返回当前目录选中的所有文件和子目录的列表，返回的名字是完整的URI。

add_filter(*filter*)
添加 *Gtk.FileFilter* 的实例 *filter* 到用户可以选择的过滤器列表。 当过滤器被选中时只有通过过滤器的文件才会被显示。

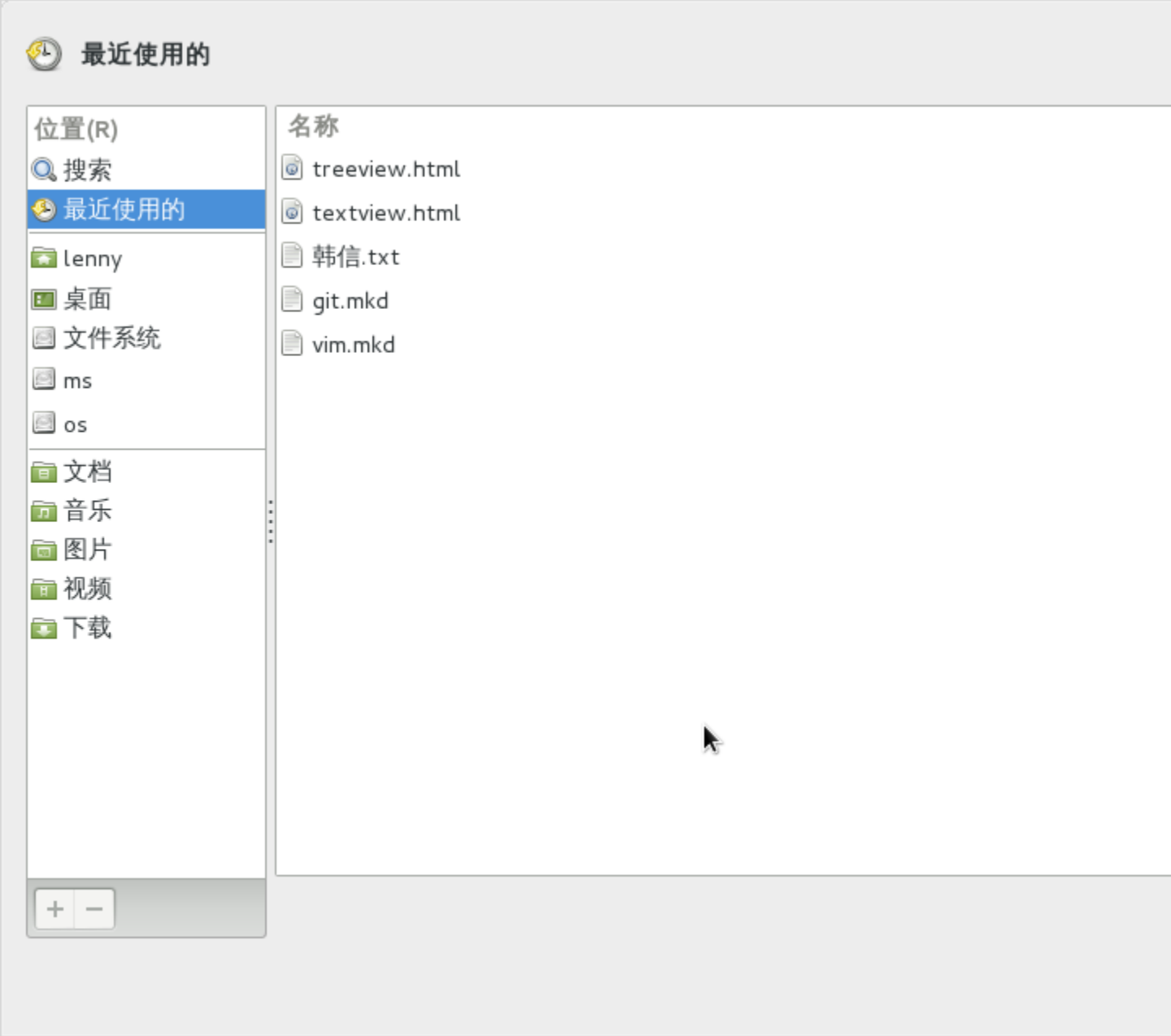
class Gtk.**FileFilter**

set_name(*name*)
设置过滤器的易于阅读的名字。这也是将会在文件选择框中显示的字符串。

add_mime_type(*mime_type*)
添加一个允许给定的MIME类型的规则到过滤器。

add_pattern(*pattern*)
添加一个允许shell风格的全局的规则到过滤器。

Example




```
from gi.repository import Gtk

class FileChooserWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="FileChooser Example")

        box = Gtk.Box(spacing=6)
        self.add(box)

```

(下页继续)

(续上页)

```
        button1 = Gtk.Button("Choose File")
        button1.connect("clicked", self.on_file_clicked)
        box.add(button1)

        button2 = Gtk.Button("Choose Folder")
        button2.connect("clicked", self.on_folder_clicked)
        box.add(button2)

    def on_file_clicked(self, widget):
        dialog = Gtk.FileChooserDialog("Please choose a file", self,
            Gtk.FileChooserAction.OPEN,
            (Gtk.STOCK_CANCEL, Gtk.ResponseType.CANCEL,
             Gtk.STOCK_OPEN, Gtk.ResponseType.OK))

        self.add_filters(dialog)

        response = dialog.run()
        if response == Gtk.ResponseType.OK:
            print "Open clicked"
            print "File selected: " + dialog.get_filename()
        elif response == Gtk.ResponseType.CANCEL:
            print "Cancel clicked"

        dialog.destroy()

    def add_filters(self, dialog):
        filter_text = Gtk.FileFilter()
        filter_text.set_name("Text files")
        filter_text.add_mime_type("text/plain")
        dialog.add_filter(filter_text)

        filter_py = Gtk.FileFilter()
        filter_py.set_name("Python files")
        filter_py.add_mime_type("text/x-python")
        dialog.add_filter(filter_py)

        filter_any = Gtk.FileFilter()
        filter_any.set_name("Any files")
        filter_any.add_pattern("*")
        dialog.add_filter(filter_any)

    def on_folder_clicked(self, widget):
        dialog = Gtk.FileChooserDialog("Please choose a folder", self,
            Gtk.FileChooserAction.SELECT_FOLDER,
            (Gtk.STOCK_CANCEL, Gtk.ResponseType.CANCEL,
             "Select", Gtk.ResponseType.OK))
        dialog.set_default_size(800, 400)

        response = dialog.run()
        if response == Gtk.ResponseType.OK:
            print "Select clicked"
            print "Folder selected: " + dialog.get_filename()
        elif response == Gtk.ResponseType.CANCEL:
            print "Cancel clicked"

        dialog.destroy()
```

(下页继续)

(续上页)

```
67
68 win = FileChooserWindow()
69 win.connect("delete-event", Gtk.main_quit)
70 win.show_all()
71 Gtk.main()
```

4.1.18 剪贴板

Gtk.Clipboard 为各种数据提供了一个存储区域，包括文本和图像。使用剪贴板 允许这些数据在不同的程序间通过复制、剪切、粘贴等动作共享。这些动作通常通过三种 方式完成：使用键盘快捷键、使用 Gtk.MenuItem，将这些动作的函数与 *Gtk.Button* 控件连接。

对于不同的目的有多种的剪贴板，在觉到多数环境下， CLIPBOARD 用于日常的复制和 粘贴，PRIMARY 则存储光标选中的文本。

Clipboard 对象

class Gtk.**Clipboard**

get(*selection*)
根据 selection 获得相应的 *Gtk.Clipboard* 。

selection 为描述使用哪一个剪贴板的 Gdk.Atom 的实例。 预定义的值：

- Gdk.SELECTION_CLIPBOARD
- Gdk.SELECTION_PRIMARY

set_text(*text*, *length*)
设置剪贴板的内容为给的的文本。

text 是要放进剪贴板的字符串。

length 是要存放的字符数。如果存储整个字符串则可以忽略。

set_image(*image*)
设置剪贴板的内容为给定的图像。

image 必须为 Gdk.Pixbuf 的实例。 要从 Gdk.Image 获取，使用 image.get_pixbuf() 。

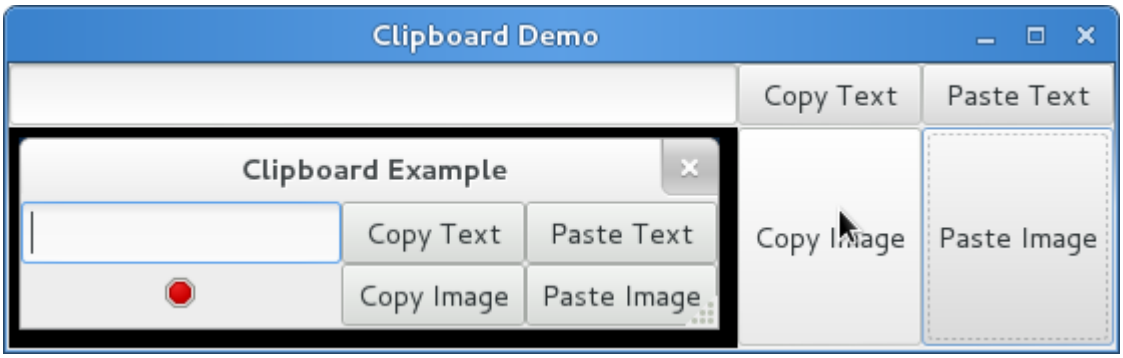
wait_for_text()
以字符串返回剪贴板的内容，如果剪贴板为空或者当前没有文本则返回 None 。

wait_for_image()
以 Gtk.Pixbuf 返回接铁板的内容。 如果剪贴板内没有图像或为空则返回 None 。

store()
在本程序之外保存剪贴板的内容，否则拷贝到剪贴板中的数据可能会在程序退出时会消失。

clear()
清除剪贴板的内容。 使用请注意，可能会清除其他程序设置的内容。

Example




```
from gi.repository import Gtk, Gdk

class ClipboardWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='Clipboard Demo')

        grid = Gtk.Grid()

        self.clipboard = Gtk.Clipboard.get(Gdk.SELECTION_CLIPBOARD)

        self.entry = Gtk.Entry()
        self.image = Gtk.Image.new_from_stock(Gtk.STOCK_STOP, Gtk.IconSize.MENU)

        button_copy_text = Gtk.Button('Copy Text')
        button_paste_text = Gtk.Button('Paste Text')
        button_copy_image = Gtk.Button('Copy Image')
        button_paste_image = Gtk.Button('Paste Image')

        grid.attach(self.entry, 0, 0, 1, 1)
        grid.attach_next_to(self.image, self.entry, Gtk.PositionType.BOTTOM, 1, 1)
        grid.attach_next_to(button_copy_text, self.entry, Gtk.PositionType.RIGHT, 1,␣
↪1)
        grid.attach_next_to(button_paste_text, button_copy_text, Gtk.PositionType.
↪RIGHT, 1, 1)
        grid.attach_next_to(button_copy_image, button_copy_text, Gtk.PositionType.
↪BOTTOM, 1, 1)
        grid.attach_next_to(button_paste_image, button_paste_text, Gtk.PositionType.
↪BOTTOM, 1, 1)

        button_copy_text.connect('clicked', self.copy_text)
        button_paste_text.connect('clicked', self.paste_text)
        button_copy_image.connect('clicked', self.copy_image)
        button_paste_image.connect('clicked', self.paste_image)

        self.add(grid)

    def copy_text(self, widget):
        self.clipboard.set_text(self.entry.get_text(), -1)

    def paste_text(self, widget):
        text = self.clipboard.wait_for_text()
```

(下页继续)

(续上页)

```
        if text != None:
            self.entry.set_text(text)
        else:
            print 'No Text on the clipboard'

    def copy_image(self, widget):
        if self.image.get_storage_type() == Gtk.ImageType.PIXBUF:
            self.clipboard.set_image(self.image.get_pixbuf())
        else:
            print 'No image has been pasted yet'

    def paste_image(self, widget):
        image = self.clipboard.wait_for_image()
        if image != None:
            self.image.set_from_pixbuf(image)
        else:
            print 'No Image on the clipboard'

win = ClipboardWindow()
win.connect('delete-event', Gtk.main_quit)
win.show_all()
Gtk.main()
```

4.1.19 拖拽支持

注解: 低于3.0.3版本的PyGObject有一个bug使得拖拽功能不正确，因此下面的例子要求 PyGObject 的版本大于等于3.0.3。

要在控件间设置拖拽功能，首先要使用 *Gtk.Widget.drag_source_set()* 选择一个 拖拽的源控件，然后通过 *Gtk.Widget.drag_dest_set()* 设置拖拽的目的控件，然后 在两个控件上处理相关的信号。

有一些专门的控件并不使用 *Gtk.Widget.drag_source_set()* 和 *Gtk.Widget.drag_dest_set()*，而是使用一些特殊的函数 (例如 *Gtk.TreeView* 和 *Gtk.IconView*)。

一个基本的拖拽支持只要求源控件连接 “drag-data-get” 信号，目地控件连接 “drag-data-received” 信号。更多复杂的功能如指定释放的区域和定制拖拽的图片则要求 连接 额外的信号 并和其提供的 `Gdk.DragContext` 对象交互。

要在源和目的控件间传递数据，你必须使用 *Gtk.SelectionData* 的get与set函数 与 *“drag-data-get”* 和 *“drag-data-received”* 信号提供的 *Gtk.SelectionData* 变量交互。

Target Entries

要知道拖拽的源和目的发送和接收的数据，需要一个 *Gtk.TargetEntry's* 的列表。 *Gtk.TargetEntry* 描述了被拖拽源发送或被拖拽目的接收的一片数据。

有两种方式添加 *Gtk.TargetEntry's* 源或目的。如果是简单的 拖拽支持并且源目的节点都是不同的类型，可以使用函数组 `mentioned here` 。如果需要拖拽多种 类型的数据并对数据做更加复杂的操作，则需要使用 *Gtk.TargetEntry.new()* 创建 `Gtk.TargetEntry`的 。

拖拽方法与对象

class Gtk.**Widget**

drag_source_set(*start_button_mask*, *targets*, *actions*)

设置控件为拖拽源。

start_button_mask 是多个:attr:*Gdk.ModifierType* 的组合设置了要使拖拽支持 发生要按下的按钮。 *targets* 是一个 *Gtk.TargetEntry* 的列表描述了要 在源和目的之间传递的数据。 *actions* 是 Gdk.DragAction 的组合标记了 可能的拖拽动作。

drag_dest_set(*flags*, *targets*, *actions*)

设置控件为拖拽的目的。

flags 为 Gtk.DestDefaults 的组合配置了拖拽发生时的动作。 *targets* 是 *Gtk.TargetEntry*的 的列表描述了拖拽 源与目的之间的数据。 *actions* 是 Gdk.DragAction 的组合描述了可能的拖拽动作。

drag_source_add_text_targets()
drag_dest_add_text_targets()

添加 *Gtk.TargetEntry* 为拖拽源或目的，其包含了一段文本。

drag_source_add_image_targets()
drag_dest_add_image_targets()

添加 *Gtk.TargetEntry* 为拖拽源或目的，其包含了:class:*GdkPixbuf.Pixbuf* 。

drag_source_add_uri_targets()
drag_dest_add_uri_targets()

添加 *Gtk.TargetEntry* 为拖拽源或目的，其包含了一个 URI 列表。

class Gtk.**TargetEntry**

static new(*target*, *flags*, *info*)

创建一个新的目的节点。

target 为一个字符串描述了目的节点描述的数据的类型。

flags 控制数据在拖拽源/目的之间传递的条件，为 Gtk.TargetFlags 的组合:

- Gtk.TargetFlags.SAME_APP - 只在相同程序间传递。
- Gtk.TargetFlags.SAME_WIDGET - 只在相同控件间传递。
- Gtk.TargetFlags.OTHER_APP - 只在不同程序间传递。
- Gtk.TargetFlags.OTHER_WIDGET - 只在不同控件间传递。

info 为应用程序ID，可以用来决定在一次拖拽操作中不同的数据片。

class Gtk.**SelectionData**

get_text()

返回selection data中包含的文本数据。

set_text(*text*)

设置selection data包含的文本为 *text* 。

get_pixbuf()

返回selection data包含的pixbuf图像。

set_pixbuf(*pixbuf*)

设置selection data包含的pixbuf为 *pixbuf* 。

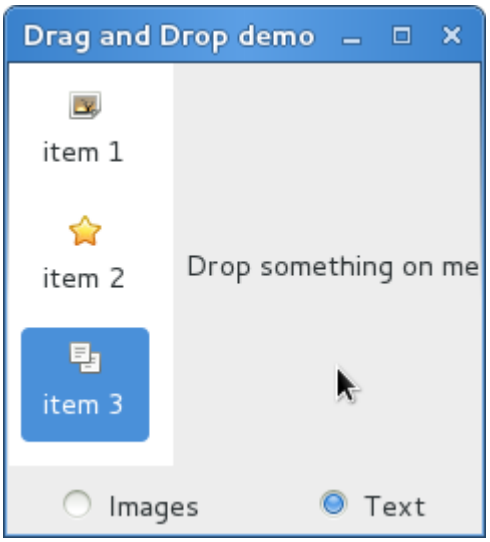
拖拽源信号

名字	触发时机	通常目的
drag-data-get	拖拽目地请求数据时	在拖拽源与目的间传递数据
drag-data-delete	action为 Gdk.DragAction.MOVE 的拖拽操作完成	完成 'move' 操作时删除源数据
drag-data-end	拖拽完成	撤销任何拖拽开始后的动作

Drag Destination Signals

名字	触发时机 | 通常目的	
drag-motion	拖拽图标移动到目地区域	只允许拖拽到指定的区域
drag-drop	在目地区域释放了图标	只允许在指定的区域释放数据
drag-data-received	拖拽目地收到了数据	在源与目地之间传递数据

Example




```
from gi.repository import Gtk, Gdk, GdkPixbuf

(TARGET_ENTRY_TEXT, TARGET_ENTRY_PIXBUF) = range(2)
(COLUMN_TEXT, COLUMN_PIXBUF) = range(2)
DRAG_ACTION = Gdk.DragAction.COPY

class DragDropWindow(Gtk.Window):

    def __init__(self):
```

(下页继续)

(续上页)

```
        Gtk.Window.__init__(self, title='Drag and Drop demo')

        vbox = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=6)
        self.add(vbox)

        hbox = Gtk.Box(spacing=6)
        vbox.pack_start(hbox, True, True, 0)

        self.icon_view = DragSourceIconView()
        self.drop_area = DropArea()

        hbox.pack_start(self.icon_view, True, True, 0)
        hbox.pack_start(self.drop_area, True, True, 0)

        button_box = Gtk.Box(spacing=6)
        vbox.pack_start(button_box, True, False, 0)

        image_button = Gtk.RadioButton.new_with_label_from_widget(None, 'Images')
        image_button.connect('toggled', self.add_image_targets)
        button_box.pack_start(image_button, True, False, 0)

        text_button = Gtk.RadioButton.new_with_label_from_widget(image_button, 'Text')
        text_button.connect('toggled', self.add_text_targets)
        button_box.pack_start(text_button, True, False, 0)

        self.add_image_targets()

    def add_image_targets(self, button=None):
        targets = Gtk.TargetList.new([])
        targets.add_image_targets(TARGET_ENTRY_PIXBUF, True)

        self.drop_area.drag_dest_set_target_list(targets)
        self.icon_view.drag_source_set_target_list(targets)

    def add_text_targets(self, button=None):
        self.drop_area.drag_dest_set_target_list(None)
        self.icon_view.drag_source_set_target_list(None)

        self.drop_area.drag_dest_add_text_targets()
        self.icon_view.drag_source_add_text_targets()

class DragSourceIconView(Gtk.IconView):

    def __init__(self):
        Gtk.IconView.__init__(self)
        self.set_text_column(COLUMN_TEXT)
        self.set_pixbuf_column(COLUMN_PIXBUF)

        model = Gtk.ListStore(str, GdkPixbuf.Pixbuf)
        self.set_model(model)
        self.add_item('item 1', 'image')
        self.add_item('item 2', 'gtk-about')
        self.add_item('item 3', 'edit-copy')

        self.enable_model_drag_source(Gdk.ModifierType.BUTTON1_MASK, [], DRAG_ACTION)
        self.connect('drag-data-get', self.on_drag_data_get)
```

(下页继续)

(续上页)

```
66
67      def on_drag_data_get(self, widget, drag_context, data, info, time):
68          selected_path = self.get_selected_items()[0]
69          selected_iter = self.get_model().get_iter(selected_path)
70
71          if info == TARGET_ENTRY_TEXT:
72              text = self.get_model().get_value(selected_iter, COLUMN_TEXT)
73              data.set_text(text, -1)
74          elif info == TARGET_ENTRY_PIXBUF:
75              pixbuf = self.get_model().get_value(selected_iter, COLUMN_PIXBUF)
76              data.set_pixbuf(pixbuf)
77
78      def add_item(self, text, icon_name):
79          pixbuf = Gtk.IconTheme.get_default().load_icon(icon_name, 16, 0)
80          self.get_model().append([text, pixbuf])
81
82  class DropArea(Gtk.Label):
83
84      def __init__(self):
85          Gtk.Label.__init__(self, 'Drop something on me')
86          self.drag_dest_set(Gtk.DestDefaults.ALL, [], DRAG_ACTION)
87
88          self.connect('drag-data-received', self.on_drag_data_received)
89
90      def on_drag_data_received(self, widget, drag_context, x, y, data, info, time):
91          if info == TARGET_ENTRY_TEXT:
92              text = data.get_text()
93              print 'Received text:%s' %text
94          elif info == TARGET_ENTRY_PIXBUF:
95              pixbuf = data.get_pixbuf()
96              width = pixbuf.get_width()
97              height = pixbuf.get_height()
98
99              print 'Received pixbuf with width %spx and height %spx' %(width, height)
100
101 win = DragDropWindow()
102 win.connect('delete-event', Gtk.main_quit)
103 win.show_all()
104 Gtk.main()
```

4.1.20 Glade and Gtk.Builder

Gtk.Builder 可以让你不用写一行的代码就可以设计界面。这是通过使用一个 XML文件来描述界面然后在运行时通过Builder类加载此XML描述文件并自动创建对象来实现的。 Glade 应用可以使你不需要手动编写XML文件而以一种 WYSIWYG(所见即所得)的方式来设计界面。

这种方式有很多优点:

- 写更少的代码。
- UI 更加容易改变，因此可以改进UI。
- 没有编程经验的设计师可以创建并编辑UI。
- UI的描述与正在使用的编程语言相互独立。

仍然需要用户对界面修改的代码，但是 *Gtk.Builder* 允许你集中注意力实现程序的功能。

创建并加载 .glade 文件

首先你需要下载并安装Glade。有 很多关于Glade的教程， 因此这里不再详细介绍。让我们开始创建一个带有一个按钮的窗口并保存其为 *example.glade* 。 最终的XML文件看起来像下面这样:

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <!-- interface-requires gtk+ 3.0 -->
  <object class="GtkWindow" id="window1">
    <property name="can_focus">False</property>
    <child>
      <object class="GtkButton" id="button1">
        <property name="label" translatable="yes">button</property>
        <property name="use_action_appearance">False</property>
        <property name="visible">True</property>
        <property name="can_focus">True</property>
        <property name="receives_default">True</property>
        <property name="use_action_appearance">False</property>
      </object>
    </child>
  </object>
</interface>
```

要在Python中加载这个文件我们需要一个 *Gtk.Builder* 对象。

```
builder = Gtk.Builder()
builder.add_from_file("example.glade")
```

第二行加载所有 *example.glade* 定义的对象到Buildeer对象中。

当然也可以只加载一部分对象。下载的代码只加载了tuple中指定的对象(及其孩子对象)。

```
# we don't really have two buttons here, this is just an example
builder.add_objects_from_file("example.glade", ("button1", "button2"))
```

这两种方法也可以在字符串中加载对象。他们对应的函数为 *Gtk.Builder.add_from_string()* 和 *Gtk.Builder.add_objects_from_string()* ， 他们从参数中取得XML字符串而不是文件名。

访问控件

现在我们想要显示的窗口和按钮都加载进来了。因此要在该窗口上调用 `Gtk.Window.show_all()` 方法，但是我们如何访问我们的对象呢?

```
window = builder.get_object("window1")
window.show_all()
```

每一个控件都可以通过 *Gtk.Builder.get_object()* 方法和控件的 *id* 来获取。哈哈，真简单。

当然也可以加载所有对象到一个列表中

```
builder.get_objects()
```

连接信号

Glade 也使得定义你想要连接到你代码里的函数的信号成为可能，并却这不需要从builder 加载每一个对象并且手动去连接信号。要做的第一件事就是在Glade中声明信号的名字。 例子中我们在按钮按下时关闭窗

口，所以我们给窗口的 “delete-event” 信号设置处理函数 “onDeleteWindow”， “pressed” 信号设置处理函数 “onButtonPressed”。 现在XML文件看起来像下面这个样子：

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <!-- interface-requires gtk+ 3.0 -->
  <object class="GtkWindow" id="window1">
    <property name="can_focus">False</property>
    <signal name="delete-event" handler="onDeleteWindow" swapped="no"/>
    <child>
      <object class="GtkButton" id="button1">
        <property name="label" translatable="yes">button</property>
        <property name="use_action_appearance">False</property>
        <property name="visible">True</property>
        <property name="can_focus">True</property>
        <property name="receives_default">True</property>
        <property name="use_action_appearance">False</property>
        <signal name="pressed" handler="onButtonPressed" swapped="no"/>
      </object>
    </child>
  </object>
</interface>
```

现在我们要在代码中定义处理函数。 *onDeleteWindow* 应该简单地调用 `Gtk.main_quit()` 。 当按钮按下时我们可能想要打印字符串 “Hello World!”，因此我们定义的函数看起来像：

```
def hello(button):
    print "Hello World!"
```

接下来，我们要连接信号和处理函数。最简单的方法是定义一个带有名字到处理函数的映射 *dict* 并传递给 *`Gtk.Builder.connect_signals()`* 方法。

```
handlers = {
    "onDeleteWindow": Gtk.main_quit,
    "onButtonPressed": hello
}
builder.connect_signals(handlers)
```

一种可选的方法是创建一个包含要调用的函数的类，在我们的例子中最终的代码片段如下：

```
1  class Handler:
2      def onDeleteWindow(self, *args):
3          Gtk.main_quit(*args)
4
5      def onButtonPressed(self, button):
6          print 'Hello World!'
7
8  builder = Gtk.Builder()
9  builder.add_from_file('builder_example.glade')
10 builder.connect_signals(Handler())
```

Builder 对象

class Gtk.**Builder**

add_from_file(*filename*)
加载并解析给定的文件并合并到builder当前的内容。

add_from_string(*string*)
解析给定的字符串并合并到builder当前的内容。

add_objects_from_file(*filename*, *object_ids*)
与 *Gtk.Builder.add_from_file()* 一样，但是只加载 *object_ids* 列表给定的对象。

add_objects_from_string(*filename*, *object_ids*)
与 *Gtk.Builder.add_from_string()* 一样，但是只加载 *object_ids* 列表指定的对象。

get_object(*object_id*)
从由builder加载的对象中获取 *object_id* 指定的控件。

get_objects()
返回所有加载的对象。

connect_signals(*handler_object*)
连接信号到 *handler_object* 指定的方法。 *handler_object* 可以为任何包含 界面描述文件中指定的信号处理函数名字键或属性的对象，例如一个类或者字典。

Example

The final code of the example

```
from gi.repository import Gtk

class Handler:
    def onDeleteWindow(self, *args):
        Gtk.main_quit(*args)

    def onButtonPressed(self, button):
        print 'Hello World!'

builder = Gtk.Builder()
builder.add_from_file('builder_example.glade')
builder.connect_signals(Handler())

win = builder.get_object('window1')
win.show_all()
Gtk.main()
```

4.1.21 对象系统

GObject 是基础类型提供了Gtk+对象系统需要的所有属性和方法。 *GObject.GObject* 提供了构造和析构对象的方法，属性访问方法和信号支持。

本节将要介绍Python实现的GObject一些重要的方面。

从GObject.GObject继承

一个原生的GObject可以通过 *GObject.GObject* 访问，但通常很少直接实例化，而是 使用继承后的类。 *Gtk.Widget* 就是一个继承自 *GObject.GObject* 的类。 创建一个继承类来创建一个新的如设置对话框的新控件通常很有趣。

要从 *GObject.GObject* 继承，你必须在你的构造函数中调用 GObject.GObject.__init__()，例如如果你的类继承自 *Gtk.Button*，则必须 调用 Gtk.Button.__init__()， 就像下面这样。

```
from gi.repository import GObject

class MyObject(GObject.GObject):

    def __init__(self):
        GObject.GObject.__init__(self)
```

信号

信号可以随意的连接到程序相关的事件，可以有任意多个收听者。例如， 在GTK+里面，每一个 用户事件(按键或鼠标移动) 从X server接收后产生一个GTK+事件在给的的对象实例上触发信号。

每一个信号都与其可以触发的类型一起在类型系统里面注册：当类型的使用者注册信号触发 时的回调函数时，要链接信号到给定类型的实例。

接收信号

参考 主循环与信号

创建新的信号

可以通过添加信号到 *GObject.GObject.__gsignals__* 字典中创建新的信号。

当创建新的信号时也可以定义一个处理方法，该方法会在每次信号触发时调用，叫做 do_signal_name 。

```
class MyObject(GObject.GObject):
    __gsignals__ = {
        'my_signal': (GObject.SIGNAL_RUN_FIRST, None,
                      (int,))
    }

    def do_my_signal(self, arg):
        print "class method for `my_signal' called with argument", arg
```

GObject.SIGNAL_RUN_FIRST 指示在信号触发的第一阶段调用对象方法 (此处:meth:*do_my_signal*)。也可以设置为 *GObject.SIGNAL_RUN_LAST* (方法在信号触发的第三阶段调用)和 *GObject.SIGNAL_RUN_CLEANUP* (在信号触发 的最后一个阶段调用)。

第二个参数， None 指示信号的返回类型，通常为 None 。

(int,) 指示信号的参数，此处信号只接收一个参数，类型为 int 。参数类型列表必须 以逗号结束。

信号可以使用 *GObject.GObject.emit()* 触发。

```
my_obj.emit("my_signal", 42) # emit the signal "my_signal", with the
                             # argument 42
```

属性

GObject一个很好的特性就是其对于对象属性的get/set方法。每一个继承自 *GObject.GObject* 的类都可以定义新的属性，每一个属性作为一个类型永远不会改变(例如 str, float, int等)。 例如 *Gtk.Button* 的 “label” 属性包含了按钮的文本。

使用已有的属性

GObject.GObject 提供了多个很有用的函数来管理已有的属性， *GObject.GObject.get_property()* 和 *GObject.GObject.set_property()* 。

一些属性也有他们相应的函数，叫做getter和setter。对于按钮的 “label” 属性，有两个函数 来获取和设置该属性， `Gtk.Button.get_label()` 和 *Gtk.Button.set_label()* 。

创建新的属性

属性通过名字和类型定义，即使python是动态类型的，一旦定义你也不能改变属性的类型。 属性可以通过 `GObject.property()` 创建。

```
from gi.repository import GObject

class MyObject(GObject.GObject):

    foo = GObject.property(type=str, default='bar')
    property_float = GObject.property(type=float)
    def __init__(self):
        GObject.GObject.__init__(self)
```

如果你想让某些属性只读不可写，属性也可以为只读的。要这样做，你可以给属性定义添加 一些标志flags，来控制读写权限。标志有 *GObject.PARAM_READABLE* (只能通过 外部代码读取)， *GObject.PARAM_WRITABLE* (只可写)， *GObject.PARAM_READWRITE* (public):

```
foo = GObject.property(type=str, flags = GObject.PARAM_READABLE) # won't be writable
bar = GObject.property(type=str, flags = GObject.PARAM_WRITABLE) # won't be readable
```

你也可以通过 `GObject.property()` 与函数修饰符创建新的函数来定义只读属性。

```
from gi.repository import GObject

class MyObject(GObject.GObject):

    def __init__(self):
        GObject.GObject.__init__(self)

    @GObject.property
    def readonly(self):
        return 'This is read-only.'
```

你可以使用以下代码获取该属性:

```
my_object = MyObject()
print my_object.readonly
print my_object.get_property("readonly")
```

也有定义数值类型属性的最大值和最小值的方法，需要使用更加复杂的形式:

```
from gi.repository import GObject

class MyObject(GObject.GObject):

    __gproperties__ = {
        "int-prop": (int, # type
                     "integer prop", # nick
                     "A property that contains an integer", # blurb
                     1, # min
                     5, # max
                     2, # default
                     GObject.PARAM_READWRITE # flags
                    ),
    }

    def __init__(self):
        GObject.GObject.__init__(self)
        self.int_prop = 2

    def do_get_property(self, prop):
        if prop.name == 'int-prop':
            return self.int_prop
        else:
            raise AttributeError, 'unknown property %s' % prop.name

    def do_set_property(self, prop, value):
        if prop.name == 'int-prop':
            self.int_prop = value
        else:
            raise AttributeError, 'unknown property %s' % prop.name
```

属性必须通过 *GObject.GObject.__gproperties__* 字典定义，并通过 do_get_property 和 do_set_property 来处理。

监视属性

当属性被修改时，会触发一个信号，“notify::property_name”：

```
my_object = MyObject()

def on_notify_foo(obj, gparamstring):
    print "foo changed"

my_object.connect("notify::foo", on_notify_foo)

my_object.set_property("foo", "bar") # on_notify_foo will be called
```

API

class GObject.**GObject**

get_property(*property_name*)
获取属性的值。

set_property(*property_name*, *value*)
设置属性 *property_name* 的值为 *value* 。

emit(*signal_name*, ...)
触发信号 *signal_name* 。信号的参数必须在后面传递，例如，如果你的信号类型 为 `(int,)` ，则要像下面这样触发:

```
self.emit(signal_name, 42)
```

freeze_notify()
本函数会冻结所有 “notify::” 信号(这些信号会在属性改变时触发) 指导 *thaw_notify()* 被调用。

建议调用 *freeze_notify()* 时使用 *with* 语句，这样可以确保 *thaw_notify()* 在语句块的最后被调用:

```
with an_object.freeze_notify():
    # Do your work here
    ...
```

thaw_notify()
解冻所有的被 *freeze_notify()* 冻结的 “notify::” 信号。

建议不要明确地调用 *thaw_notify()* 而是 *with* 语句与 *freeze_notify()* 一起使用。

handler_block(*handler_id*)
阻塞实例的处理函数 *handler_id* 因此在任何信号触发之时都不会被调用，直到 *handler_unblock()* 被调用。因此 “阻塞” 一个信号处理函数意味着临时 关闭它，信号处理函数必须与之前阻塞次数相同的取消阻塞才能被再次激活。

建议 *handler_block()* 与 *with* 语句一起使用，这样会在语句块的最后 隐式调用 *handler_unblock()*

```
with an_object.handler_block(handler_id):
    # Do your work here
    ...
```

handler_unblock(*handler_id*)
取消 *handler_block()* 的效果。阻塞后的处理函数会在信号被触发时略过， 并且直到取消阻塞相同次数之前不会被调用。

建议不要直接调用 *handler_unblock()* 而是与 *with* 语句一起使用 *handler_block()* 。

__gsignals__
一个继承类可以定义新信号的字典。

字典中的每一个元素都是一个新的信号。key为信号的名字，值为一个元组，如下:

```
(GObject.SIGNAL_RUN_FIRST, None, (int,))
```

GObject.SIGNAL_RUN_FIRST 可以被替换为 *GObject.SIGNAL_RUN_LAST* 或者 *GObject.SIGNAL_RUN_CLEANUP* 。 `None` 为信号的返回类型。 `(int,)` 为参数的列表，必须以逗号结尾。

__gproperties__
__gproperties__ 字典是一个可以定义你的对象属性的属性。这并不是建议 的方式定义新属性，上面提到的方法更加的简洁。本方法的优点是可以对属性作更多 的设置，像数值类型的最大值与最小值之类。

key 为属性的名字。

value 为描述属性的元组。 元组中元素的数目依赖于第一个元素，但一般至少都会包含下面的元素。

第一个元素为属性的类型(例如 `int`, `float` 等)。

第二个元素是属性的小名(昵称)，即对属性简短描述的字符串。这通常用于有 很强内省能力的程序，像GUI builder Glade 。

第三个是属性的描述或导语，即另一个更长的，描述属性的字符串。也是给 Glade 及类似程序使用的。

最后一个为属性的标志flags: :*GObject.PARAM_READABLE* ，::const::*GObject.PARAM_WRITABLE*， *GObject.PARAM_READWRITE* 。稍后我们会看到，这并不见得是第四个参数。

元组的长度依赖于属性的类型(元组的地一个元素)。具体包括下面的情形:

如果类型为 `bool` 或者 `str` ，第四个元素为属性的默认值。

如果类型为 `int` 或者 `float` ，第四个元素是可接收的最小值，第五个 为可接收的最大值，第六个为其默认值。

如果类型不是上面这些，则没有额外的元素。

`GObject.`**`SIGNAL_RUN_FIRST`**
在信号触发第一阶段调用处理方法。

`GObject.`**`SIGNAL_RUN_LAST`**
在信号触发第三阶段调用处理方法。

`GObject.`**`SIGNAL_RUN_CLEANUP`**
在信号触发最后一个阶段调用处理方法。

`GObject.`**`PARAM_READABLE`**
属性只读。

`GObject.`**`PARAM_WRITABLE`**
属性只写。

`GObject.`**`PARAM_READWRITE`**
属性可读可写。

4.1.22 Stock Items

Stock items represent commonly-used menu or toolbar items such as “Open” or “Exit”. Each stock item is identified by a stock ID; stock IDs are just strings, but constants such as *Gtk.STOCK_OPEN* are provided to avoid typing mistakes in the strings.

`Gtk.`**`STOCK_ABOUT`**


`Gtk.`**`STOCK_ADD`**


`Gtk.`**`STOCK_APPLY`**


`Gtk.`**`STOCK_BOLD`**


Gtk.**STOCK_CANCEL**

Gtk.**STOCK_CAPS_LOCK_WARNING**

Gtk.**STOCK_CDROM**

Gtk.**STOCK_CLEAR**

Gtk.**STOCK_CLOSE**

Gtk.**STOCK_COLOR_PICKER**

Gtk.**STOCK_CONNECT**

Gtk.**STOCK_CONVERT**

Gtk.**STOCK_COPY**

Gtk.**STOCK_CUT**

Gtk.**STOCK_DELETE**

Gtk.**STOCK_DIALOG_AUTHENTICATION**

Gtk.**STOCK_DIALOG_INFO**

Gtk.**STOCK_DIALOG_WARNING**

Gtk.**STOCK_DIALOG_ERROR**

Gtk.**STOCK_DIALOG_QUESTION**

Gtk.**STOCK_DISCARD**

Gtk.**STOCK_DISCONNECT**

Gtk.**STOCK_DND**

Gtk.**STOCK_DND_MULTIPLE**

Gtk.**STOCK_EDIT**

Gtk.**STOCK_EXECUTE**

Gtk.**STOCK_FILE**

Gtk.**STOCK_FIND**

Gtk.**STOCK_FIND_AND_REPLACE**

Gtk.**STOCK_FLOPPY**

Gtk.**STOCK_FULLSCREEN**

Gtk.**STOCK_GOTO_BOTTOM**

Gtk.**STOCK_GOTO_FIRST**

LTR variant:

RTL variant:

Gtk.**STOCK_GOTO_LAST**

LTR variant:

RTL variant:

Gtk.**STOCK_GOTO_TOP**

Gtk.**STOCK_GO_BACK**

LTR variant:

RTL variant:

Gtk.**STOCK_GO_DOWN**

Gtk.**STOCK_GO_FORWARD**

LTR variant:

RTL variant:

Gtk.**STOCK_GO_UP**

Gtk.**STOCK_HARDDISK**

Gtk.**STOCK_HELP**

Gtk.**STOCK_HOME**

Gtk.**STOCK_INDEX**

Gtk.**STOCK_INDENT**

LTR variant:

RTL variant:

Gtk.**STOCK_INFO**

Gtk.**STOCK_ITALIC**

Gtk.**STOCK_JUMP_TO**

LTR variant:

RTL variant:

Gtk.**STOCK_JUSTIFY_CENTER**

Gtk.**STOCK_JUSTIFY_FILL**

Gtk.**STOCK_JUSTIFY_LEFT**

Gtk.**STOCK_JUSTIFY_RIGHT**

Gtk.**STOCK_LEAVE_FULLSCREEN**

Gtk.**STOCK_MISSING_IMAGE**

Gtk.**STOCK_MEDIA_FORWARD**

LTR variant:

RTL variant:

Gtk.**STOCK_MEDIA_NEXT**

LTR variant:

RTL variant:

Gtk.**STOCK_MEDIA_PAUSE**

Gtk.**STOCK_MEDIA_PLAY**

LTR variant:

RTL variant:

Gtk.**STOCK_MEDIA_PREVIOUS**

LTR variant:

RTL variant:

Gtk.**STOCK_MEDIA_RECORD**

Gtk.**STOCK_MEDIA_REWIND**

LTR variant:

RTL variant:

Gtk.**STOCK_MEDIA_STOP**

Gtk.**STOCK_NETWORK**

Gtk.**STOCK_NEW**

Gtk.**STOCK_NO**

Gtk.**STOCK_OK**

Gtk.**STOCK_OPEN**

Gtk.**STOCK_ORIENTATION_PORTRAIT**

Gtk.**STOCK_ORIENTATION_LANDSCAPE**

Gtk.**STOCK_ORIENTATION_REVERSE_LANDSCAPE**

Gtk.**STOCK_ORIENTATION_REVERSE_PORTRAIT**

Gtk.**STOCK_PAGE_SETUP**

Gtk.**STOCK_PASTE**

Gtk.**STOCK_PREFERENCES**

Gtk.**STOCK_PRINT**

Gtk.**STOCK_PRINT_ERROR**

Gtk.**STOCK_PRINT_PAUSED**

Gtk.**STOCK_PRINT_PREVIEW**

Gtk.**STOCK_PRINT_REPORT**

Gtk.**STOCK_PRINT_WARNING**

Gtk.**STOCK_PROPERTIES**

Gtk.**STOCK_QUIT**

Gtk.**STOCK_REDO**

LTR variant:

RTL variant:

Gtk.**STOCK_REFRESH**

Gtk.**STOCK_REMOVE**

Gtk.**STOCK_REVERT_TO_SAVED**

LTR variant:

RTL variant:

Gtk.**STOCK_SAVE**

Gtk.**STOCK_SAVE_AS**

Gtk.**STOCK_SELECT_ALL**

Gtk.**STOCK_SELECT_COLOR**

Gtk.**STOCK_SELECT_FONT**

Gtk.**STOCK_SORT_ASCENDING**

Gtk.**STOCK_SORT_DESCENDING**

Gtk.**STOCK_SPELL_CHECK**

Gtk.**STOCK_STOP**

Gtk.**STOCK_STRIKETHROUGH**

Gtk.**STOCK_UNDELETE**

LTR variant:

RTL variant:

Gtk.**STOCK_UNDERLINE**

Gtk.**STOCK_UNDO**

LTR variant:

RTL variant:

Gtk.**STOCK_UNINDENT**

LTR variant:

RTL variant:

Gtk.**STOCK_YES**

Gtk.**STOCK_ZOOM_100**

Gtk.**STOCK_ZOOM_FIT**

Gtk.**STOCK_ZOOM_IN**

4.2 GStreamer 开发经验

4.2.1 GStreamer系列之几个入门概念

Overview

GStreamer是一个多媒体框架，它可以允许你轻易地创建、编辑与播放多媒体文件，这是 通过创建带有很多特殊的多媒体元素的管道来完成的。

管道-pipeline

GStreamer的工作方式非常简单，你只需创建一个包含很多元素的管道，这与Linux命令行 的管道非常类似，例如，一般命令行的管道是这样的:

```
foo@bar:~$ ps ax | grep "apache" |wc -l
```

这个命令首先捕获一个进程列表然后返回名字包含 “apache” 的进程并传递给 wc 命令并 统计出行数。我们可以看出每一个使用 | 连接，并且 | 左边的命令的输出传递 给其右边的命令作为输入。

GStreamer的工作方式与此类似，GStreamer中你将很多元素串联起来，每一个元素都完成 某些特定的事。我们来演示一下:

```
gst-launch-1.0 filesrc location=越单纯越幸福.mp3 ! decodebin ! audioconvert ! alsasink
```

运行这条命令你就可以听到动听的音乐了，当然前提是你的当前目录有这个音乐文件。

`gst-launch-1.0` 可以用来运行 GStreamer 管道，你只需要将需要使用的元素一个一个 传递给它就可以了，每一个命令使用 ! 来连接。此处你可以把 ! 当作命令行里 的 | 。上面那条命令包含了几个元素，我们简单解释一下： a. filesrc——这个元素从本地磁盘加载了一个文件，使用该元素时你设置了 *location* 属性指向了音乐文件，关于属性我们后边聊。 b. decodebin——我们需要从 *filesrc* 解码，因此我们使用了这个元素。这个元素是一个 聪明的小家伙，它会自动检测文件的类型并在后台构造一些GStreamer元素来解码。因此， 此处对于 mp3 你可以使用 mad 代替之试一下。 c. audioconvert——一个声音文件中有各种各样的信息，每种信息传递到喇叭都是不同的， 因此要使用此元素来做一下转换。 d. alsasink——这个元素做的事很简单，就是把你的音频使用ALSA传递给你的声卡。

文章写到这里，我们就可以使用管道来做各种试验了，但首先我们要知道有那些元素可以 使用啊:

```
foo@bar:~$ gst-inspect-1.0
```

这个命令列出了可用的元素，你也可以使用该命令查看某一元素的详细信息，例如 filesrc 元素:

```
foo@bar:~$ gst-inspect-1.0 filesrc
```

下面介绍一些GStreamer的相关术语，一些人可能很快就会对 *pad*, *cap* 这些术语搞晕， 就不要说 *bin* 和 *ghost pad* 了。其实这些术语都相当的简单。。。

元素element

其实我们已经讨论了管道，而元素就在管道上。每一个元素都有很多属性用来设置该元素。 例如，*volume* 元素（设置管道的音量）有一个熟悉 **volume** 可以设置音量或者静音。 当你创建自己的管道时就要给很多的元素设置属性。

pad

每一个元素都有虚拟的插头供数据流入和流出，即pad。如果你把元素看作一个对输入的 数据做一些处理的黑盒。在盒子的左右两边就是插孔了，你可以插入电缆向盒子传入信息， 这就是pad要做的事。绝大多数元素有一个输入pad（叫做sink）和一个输出pad(叫做src)。 因此，我们上面的管道看起来是这样的:

```
[src] ! [sink src] ! [sink src] ! [sink]
```

最左边的元素只有一个src pad用来提供信息（如filesrc）。接下来的几个元素接收信息并 做一些处理，因此他们有sink和src pad（例如decodebin和audiocovert），最后一个元素 只接收信息（例如alsasink）。当你使用 `gst-inspect-1.0` 命令查看一个元素的详细 信息时，就会列出该元素的pad信息。

注意可能与平时大家认为的概念有些不同的是，src pad是用来发送数据的端点，即数据的 输出端；而sink pad是用来接收数据的端点，即数据的输入端。

而且，一般来说，src pad只能连接到sink pad。当然，没有例外的规则是不存在的， ghost pad两端就要连接相同类型的pad，具体请参考后面的例子吧。

cap

我们已经了解了pad和从管道第一个元素到最后一个元素的数据流是怎么回事了，那么我们 来讨论下 *cap* 。每一个元素都有特定的cap，cap是指该元素可以接收什么样的信息（ 例如是接收音频还是视频）。你

可以把cap看成是电源插座上其可以接受什么范围电压的规则。

bin

很多人不理解bin，其实它很简单。bin就是一种便捷的方式，可以把很多个元素放进一个 容器中。例如你有很多个元素用来解码视频并对其使用一些效果。要使事情变得简单一些， 你可以把这些元素放进一个bin（就像一个容器）中，以后你就可以使用bin来引用这些元素了。 这样其实bin变成了一个元素，例如你的管道是 `a ! b ! c ! d`，你可以把他们放进 mybin，这样当你使用mybin时其实是引用了 `a ! b ! c ! d`。

ghost pad

当你创建了一个bin并在里面放置了很多元素时，该bin变成了你自定义的元素，该元素按 顺序调用里面的元素。要做到这样，你的bin很自然地需要它自己的pad，它自己的pad会挂接 到bin里面元素的pad上，这就是 *ghost pad* 了。当你创建一个bin时，你创建了ghost pad 并告诉他们要去挂接里面哪一个元素。

Example

话不多说，上例子。

```
import gi
gi.require_version('Gst', '1.0')
from gi.repository import Gst, GObject, GLib
GObject.threads_init()
Gst.init(None)

class Play:
    def __init__(self):
        self.pipeline = Gst.Pipeline()

        self.audiotestsrc = Gst.ElementFactory.make('audiotestsrc', 'audio')
        # set property of element
        # self.audiotestsrc.set_property('freq', 300)
        print('freq:%d' %self.audiotestsrc.get_property('freq'))
        self.pipeline.add(self.audiotestsrc)

        self.sink = Gst.ElementFactory.make('alsasink', 'sink')
        self.pipeline.add(self.sink)

        self.audiotestsrc.link(self.sink)

        self.pipeline.set_state(Gst.State.PLAYING)

start = Play()
loop = GLib.MainLoop()
loop.run()
```

上面这个例子很简单，使用命令行的例子来写:

```
gst-launch-1.0 audiotestsrc ! alsasink
gst-launch-1.0 audiotestsrc freq=300 ! alsasink
```

我们把上面的例子加上GUI，让它看起来是一个真正的桌面应用。

使用Glade构建的应用界面:

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <!-- interface-requires gtk+ 3.0 -->
  <object class="GtkWindow" id="window1">
    <property name="can_focus">False</property>
    <child>
      <object class="GtkBox" id="box1">
        <property name="visible">True</property>
        <property name="can_focus">False</property>
        <child>
          <object class="GtkEntry" id="entry1">
            <property name="visible">True</property>
            <property name="can_focus">True</property>
            <property name="invisible_char">•</property>
            <property name="invisible_char_set">True</property>
            <property name="input_purpose">alpha</property>
            <signal name="activate" handler="on_freq_changed" swapped="no"/>
          </object>
          <packing>
            <property name="expand">False</property>
            <property name="fill">True</property>
            <property name="position">0</property>
          </packing>
        </child>
        <child>
          <object class="GtkButton" id="button1">
            <property name="label" translatable="yes">Play</property>
            <property name="visible">True</property>
            <property name="can_focus">True</property>
            <property name="receives_default">True</property>
            <property name="image_position">bottom</property>
            <signal name="clicked" handler="on_play_clicked" swapped="no"/>
          </object>
          <packing>
            <property name="expand">False</property>
            <property name="fill">True</property>
            <property name="position">1</property>
          </packing>
        </child>
        <child>
          <object class="GtkButton" id="button2">
            <property name="label" translatable="yes">Stop</property>
            <property name="visible">True</property>
            <property name="can_focus">True</property>
            <property name="receives_default">True</property>
            <signal name="clicked" handler="on_stop_clicked" swapped="no"/>
          </object>
          <packing>
            <property name="expand">False</property>
            <property name="fill">True</property>
            <property name="position">2</property>
          </packing>
        </child>
        <child>
          <object class="GtkButton" id="button3">
            <property name="label" translatable="yes">Quit</property>
            <property name="visible">True</property>
```

(下页继续)

(续上页)

```
            <property name="can_focus">True</property>
            <property name="receives_default">True</property>
            <property name="halign">end</property>
            <property name="use_underline">True</property>
            <signal name="clicked" handler="on_quit_clicked" swapped="no"/>
          </object>
          <packing>
            <property name="expand">False</property>
            <property name="fill">True</property>
            <property name="position">3</property>
          </packing>
        </child>
      </object>
    </child>
  </object>
</interface>
```

程序的处理代码：

```
import gi
gi.require_version('Gst', '1.0')
from gi.repository import Gst, GObject, Gtk
GObject.threads_init()
Gst.init(None)

class Play:
    def __init__(self):
        handlers = {
                'on_play_clicked' : self.on_play,
                'on_stop_clicked' : self.on_stop,
                'on_quit_clicked' : self.on_quit,
                'on_freq_changed' : self.on_freq_change,
        }

        self.builder = Gtk.Builder()
        self.builder.add_from_file('audiotest_gui.glade')
        self.builder.connect_signals(handlers)


        # Gstreamer gays
        self.pipeline = Gst.Pipeline()

        self.audiotestsrc = Gst.ElementFactory.make('audiotestsrc', 'audio')
        freq = self.audiotestsrc.get_property('freq')
        self.pipeline.add(self.audiotestsrc)

        self.sink = Gst.ElementFactory.make('alsasink', 'sink')
        self.pipeline.add(self.sink)

        self.audiotestsrc.link(self.sink)

        entry = self.builder.get_object('entry1')
        entry.set_text(str(freq))
        win = self.builder.get_object('window1')
        win.connect('delete-event', Gtk.main_quit)
        win.show_all()
```

(下页继续)

(续上页)

```
38
39      def on_freq_change(self, widget, *args):
40          print('freq changde...')
41          entry = self.builder.get_object('entry1')
42          freq = entry.get_text()
43          self.audiotestsrc.set_property('freq', int(freq))
44
45      def on_play(self, *args):
46          print('palying...')
47          self.pipeline.set_state(Gst.State.PLAYING)
48
49      def on_stop(self, *args):
50          print('stoped.')
51          self.pipeline.set_state(Gst.State.NULL)
52          #self.pipeline.set_state(Gst.State.PAUSED)
53          #self.pipeline.set_state(Gst.State.READY)
54
55      def on_quit(self, *args):
56          Gtk.main_quit()
57
58  start = Play()
59  Gtk.main()
```

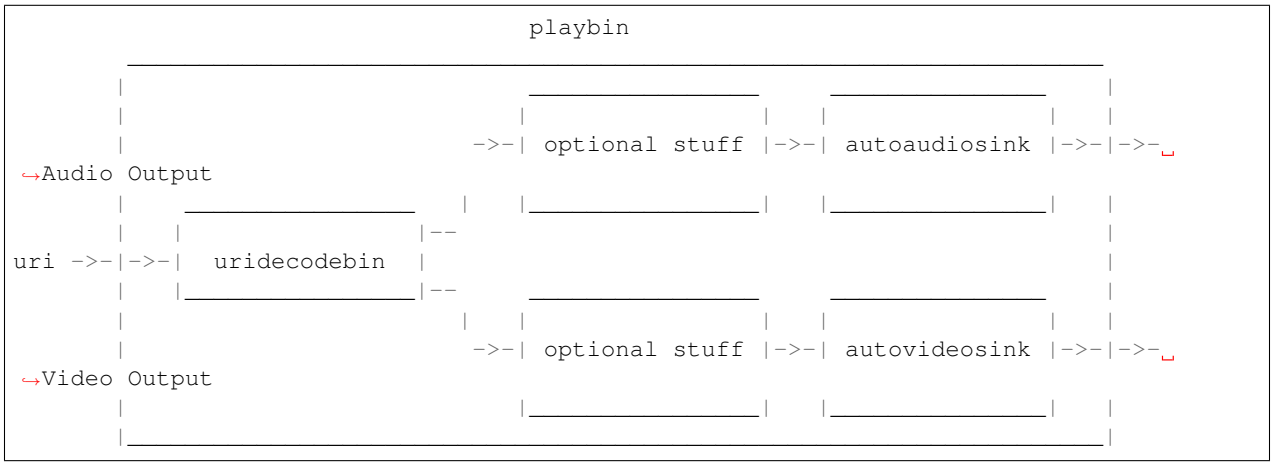
参考文献

Getting started with GStreamer with Python

4.2.2 GStreamer系列之概览

playbin是一个高级别的，自动化的音视频播放器，一般来说，它会播放发送给他的任何 支持的多媒体数据。

playbin的内部看起来是这个样子的:



“uri” 属性可以使用任何GStreamer插件支持的协议。playbin支持你将sink换成其他的，就 像我们在下面的例子中做的。一般来说，playbin总是会自动设置好你所要的一切，因此你 不要指定那些playbin没有实现的特性，开箱即用就不错的。

Example 1：播放起来

```
import sys, os
import gi
gi.require_version('Gst', '1.0')
from gi.repository import Gst, Gtk, GObject
GObject.threads_init()
Gst.init(None)

class PlayBin(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='Audio Player')
        self.set_default_size(300, -1)
        self.connect('delete-event', Gtk.main_quit)

        vbox = Gtk.VBox()
        self.add(vbox)

        self.entry = Gtk.Entry()
        vbox.pack_start(self.entry, False, True, 0)
        self.button = Gtk.ToggleButton('Start')
        self.button.connect('clicked', self.start_stop)
        vbox.add(self.button)
        self.show_all()

        # build a palyer
        self.player = Gst.ElementFactory.make('playbin', 'player')
        # we don't need video output
        #fakesink = Gst.ElementFactory.make('fakesink', 'fakesink')
        #self.player.set_property('video-sink', fakesink)
        bus = self.player.get_bus()
        bus.add_signal_watch()
        bus.connect('message', self.on_message)

    def start_stop(self, widget):
        if self.button.get_active():
            filepath = self.entry.get_text()
            if os.path.isfile(filepath):
                self.button.set_label('Stop')
                self.player.set_property('uri', 'file://'+filepath)
                self.player.set_state(Gst.State.PLAYING)
        else:
            self.player.set_state(Gst.State.NULL)
            self.button.set_label('Start')

    def on_message(self, bus, message):
        t = message.type
        if t == Gst.MessageType.EOS:
            self.player.set_state(Gst.State.NULL)
            self.button.set_label('Start')
        elif t == Gst.MessageType.ERROR:
            self.player.set_state(Gst.State.NULL)
            err, debug = message.parse_error()
            print('Error: %s' %err, debug)
            self.button.set_label('Start')

```

(下页继续)

(续上页)

```
win = PlayBin()
Gtk.main()
```

由于playbin插件总是会自动播放音视频流，因此我们将视频重定向至 `fakesink` ，这相当于Gstreamer中的 `/dev/null` 。你如果想要播放视频流，只需要注释掉这两行代码

```
        #fakesink = Gst.ElementFactory.make('fakesink', 'fakesink')
        #self.player.set_property('video-sink', fakesink)
```

Example 2：控制视频的显示

当然上面的例子在播放视频时总是会打开一个新的窗口，如果你想要在指定的窗口播放则 需要使用 `enable_sync_message_emission()` 方法。

```
import sys, os
import gi
gi.require_version('Gst', '1.0')
from gi.repository import Gst, Gtk, GObject

# important!!! needed by window.get_xid(), xvimagesink.set_window_handle.
from gi.repository import GdkX11, GstVideo

GObject.threads_init()
Gst.init(None)

class PlayBin(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='Audio Player')
        self.set_default_size(500, 400)
        self.connect('delete-event', Gtk.main_quit)

        vbox = Gtk.VBox()
        self.add(vbox)

        hbox = Gtk.HBox()
        vbox.pack_start(hbox, False, False, 0)
        self.entry = Gtk.Entry()
        hbox.pack_start(self.entry, True, True, 0)
        self.button = Gtk.ToggleButton('Start')
        self.button.connect('clicked', self.start_stop)
        hbox.add(self.button)
        self.movie = Gtk.DrawingArea()
        vbox.add(self.movie)
        # build a palyer
        self.player = Gst.ElementFactory.make('playbin', 'player2')
        bus = self.player.get_bus()
        bus.add_signal_watch()
        bus.connect('message', self.on_message)
        bus.enable_sync_message_emission()
        bus.connect('sync-message::element', self.on_sync_message)

        self.show_all()
        self.xid = self.movie.get_property('window').get_xid()

```

(下页继续)

(续上页)

```
    def start_stop(self, widget):
        if self.button.get_active():
            filepath = self.entry.get_text()
            if os.path.isfile(filepath):
                self.button.set_label('Stop')
                self.player.set_property('uri', 'file://'+filepath)
                self.player.set_state(Gst.State.PLAYING)
        else:
            self.player.set_state(Gst.State.NULL)
            self.button.set_label('Start')

    def on_message(self, bus, message):
        t = message.type
        if t == Gst.MessageType.EOS:
            self.player.set_state(Gst.State.NULL)
            self.button.set_label('Start')
        elif t == Gst.MessageType.ERROR:
            self.player.set_state(Gst.State.NULL)
            err, debug = message.parse_error()
            print('Error: %s' %err, debug)
            self.button.set_label('Start')

    def on_sync_message(self, bus, message):
        if message.get_structure().get_name() == 'prepare-window-handle':
            message.src.set_window_handle(self.xid)

win = PlayBin()
Gtk.main()
```

Example 3：添加时间显示

我们可以让事情变的更有趣，我们将playbin的 *videosink* 切换为我们自己的GHostPad，并在上面增加了一个时间显示的小元素。

```
import sys, os
import gi
gi.require_version('Gst', '1.0')
from gi.repository import Gst, Gtk, GObject

# important!!! needed by window.get_xid(), xvimagesink.set_window_handle.
from gi.repository import GdkX11, GstVideo

GObject.threads_init()
Gst.init(None)

class PlayBin(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title='Audio Player')
        self.set_default_size(500, 400)
        self.connect('delete-event', Gtk.main_quit)

        vbox = Gtk.VBox()
        self.add(vbox)

```

(下页继续)

(续上页)

```
        hbox = Gtk.HBox()
        vbox.pack_start(hbox, False, False, 0)
        self.entry = Gtk.Entry()
        hbox.pack_start(self.entry, True, True, 0)
        self.button = Gtk.ToggleButton('Start')
        self.button.connect('clicked', self.start_stop)
        hbox.add(self.button)
        self.movie = Gtk.DrawingArea()
        vbox.add(self.movie)
        # build a palyer
        self.player = Gst.ElementFactory.make('playbin', 'player2')

        mybin = Gst.Bin()
        timeoverlay = Gst.ElementFactory.make('timeoverlay', 'timeoverlay')
        mybin.add(timeoverlay)
        pad = timeoverlay.get_static_pad('video_sink')
        ghostpad = Gst.GhostPad.new('sink', pad)
        mybin.add_pad(ghostpad)
        videosink = Gst.ElementFactory.make('autovideosink', 'autovideosin')
        mybin.add(videosink)
        timeoverlay.link(videosink)
        self.player.set_property('video-sink', mybin)

        bus = self.player.get_bus()
        bus.add_signal_watch()
        bus.connect('message', self.on_message)
        bus.enable_sync_message_emission()
        bus.connect('sync-message::element', self.on_sync_message)

        self.show_all()
        self.xid = self.movie.get_property('window').get_xid()

    def start_stop(self, widget):
        if self.button.get_active():
            filepath = self.entry.get_text()
            if os.path.isfile(filepath):
                self.button.set_label('Stop')
                self.player.set_property('uri', 'file://'+filepath)
                self.player.set_state(Gst.State.PLAYING)
        else:
            self.player.set_state(Gst.State.NULL)
            self.button.set_label('Start')

    def on_message(self, bus, message):
        t = message.type
        if t == Gst.MessageType.EOS:
            self.player.set_state(Gst.State.NULL)
            self.button.set_label('Start')
        elif t == Gst.MessageType.ERROR:
            self.player.set_state(Gst.State.NULL)
            err, debug = message.parse_error()
            print('Error: %s' %err, debug)
            self.button.set_label('Start')

    def on_sync_message(self, bus, message):
        if message.get_structure().get_name() == 'prepare-window-handle':
```

(下页继续)

(续上页)

```
            message.src.set_window_handle(self.xid)

win = PlayBin()
Gtk.main()
```

4.2.3 GStreamer系列之管道

管道是带有其自己的bus和clock的高级bin，如果你的程序只包含bin一样的对象，这就是 你想要的。你可以如下创建一个管道对象:

```
mypipeline = Gst.Pipeline()
```

其实管道就是一个容器，你可以把其他对象放进去，然后指定要播放的文件并设置管道 的状态为 `Gst.State.Playing`，这样就会有多媒体流在管道里流到了。

例如，如下的一个音乐播放器的例子，使用了我们自己的mp3解码器代替了上一章中的playbin。 你可以直接使用 `gst-lanuch-1.0` 来测试管道:

```
$ gst-launch-1.0 filesrc location=ydcyxf.mp3 ! mad ! audioconvert ! alsasink
```

使用ASCII的图表可以形象地表示上面这行代码:

```
                                    Gst.Pipeline
             ____________________________________________________________________
            |                                                                    |
            |    __________       ________       _______________       __________    |
            |   |          |     |        |     |               |     |          |   |
file.mp3 ->-|---| filesrc |-->--|  mad   |-->--| audioconvert |-->--| alsasink |---|->-
↪ Audio Output
            |   |__________|     |________|     |_______________|     |__________|   |
            |                                                                    |
            |____________________________________________________________________|
```

以下是python代码的例子:

```
import sys, os

import gi
gi.require_version('Gst', '1.0')
from gi.repository import Gst, GObject, Gtk

GObject.threads_init()
Gst.init(None)

class PipeMain(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self, title='MP3-Player')
        self.resize(400, 200)
        self.connect('delete-event', Gtk.main_quit)
        vbox = Gtk.VBox()
        self.add(vbox)
        self.entry = Gtk.Entry()
        vbox.pack_start(self.entry, False, True, 1)
        self.button = Gtk.Button('Start')
```

(下页继续)

(续上页)

```
        self.button.connect('clicked', self.start_stop)
        vbox.add(self.button)
        self.show_all()

        self.player = Gst.Pipeline()
        source = Gst.ElementFactory.make('filesrc', 'file-source')
        decoder = Gst.ElementFactory.make('mad', 'mp3-decoder')
        conv = Gst.ElementFactory.make('audioconvert', 'converter')
        sink = Gst.ElementFactory.make('alsasink', 'alsa-output')

        self.player.add(source)
        self.player.add(decoder)
        self.player.add(conv)
        self.player.add(sink)

        source.link(decoder)
        decoder.link(conv)
        conv.link(sink)

        bus = self.player.get_bus()
        bus.add_signal_watch()
        bus.connect('message', self.on_message)

    def start_stop(self, w):
        if self.button.get_label() == 'Start':
            filepath = self.entry.get_text()
            print(filepath)
            if os.path.isfile(filepath):
                self.button.set_label('Stop')
                self.player.get_by_name('file-source').set_property('location',␣
↪filepath)
                self.player.set_state(Gst.State.PLAYING)
        else:
            self.player.set_state(Gst.State.NULL)
            self.button.set_label('Start')

    def on_message(self, bus, msg):
        t = msg.type
        if t == Gst.MessageType.EOS:
            self.player.set_state(Gst.State.NULL)
            self.button.set_label('Start')
        elif t == Gst.MessageType.ERROR:
            self.player.set_state(Gst.State.NULL)
            self.button.set_label('Start')
            err, debug = msg.parse_error()
            print('Error:%s' % err)

if __name__ == '__main__':
    PipeMain()
    Gtk.main()
```

4.3 Pycairo Documentation

4.3.1 概览

Pycairo 是cairo图形哭的Python语言绑定。

Pycairo 绑定被设计为与cairo的C语言API尽可能的接近，只在某些明显可以以更加 'Pythonic' 化方式实现的地方稍有改变。

Pycairo 绑定的特性：

- 提供一个面向对象的接口。
- 调用 Pycairo_Check_Status() 函数来检查cairo操作的状态，在适当的时候发送异常。
- 提供C API以供其他Python extension使用。

Pycairo 绑定并没有提供cairo_reference(), cairo_destroy(), cairo_surface_reference(), cairo_surface_destroy() (以及用于Surface和pattern的等价的函数)cairo 的这些C函数，因为对象的构造和销毁由Pycairo来处理。

要使用 pycairo库，请导入:

```
import cairo
```

参考 *Reference* 了解更详细的信息。

pycairo 的例子请参考pycairo发行代码中的 'example' 目录。

4.3.2 Reference

模块函数与常量

模块函数

cairo.**cairo_version**()

返回 编码后的版本号

返回类型 int

返回编码为一个整数的底层cairo C库的版本号。

cairo.**cairo_version_string**()

返回 编码后的版本号

返回类型 str

以便于人类阅读的"X.Y.Z"字符串形式返回底层cairo C库的版本号。

模块常量

cairo.**version**
pycairo的版本，字符串类型。

cairo.**version_info**
pycairo的版本号，元组类型。

cairo.HAS

cairo.**HAS_ATSUI_FONT**
cairo.**HAS_FT_FONT**
cairo.**HAS_GLITZ_SURFACE**
cairo.**HAS_IMAGE_SURFACE**
cairo.**HAS_PDF_SURFACE**
cairo.**HAS_PNG_FUNCTIONS**
cairo.**HAS_PS_SURFACE**
cairo.**HAS_SVG_SURFACE**
cairo.**HAS_USER_FONT**
cairo.**HAS_QUARTZ_SURFACE**
cairo.**HAS_WIN32_FONT**
cairo.**HAS_WIN32_SURFACE**
cairo.**HAS_XCB_SURFACE**
cairo.**HAS_XLIB_SURFACE**
1 代表底层cairo C库支持该特性，0 代表不支持。

cairo.ANTIALIAS

ANTIALIAS 指定了渲染文本或形状时的抗锯齿类型。

cairo.**ANTIALIAS_DEFAULT**
针对子系统和目标设备使用默认的抗锯齿。

cairo.**ANTIALIAS_NONE**
使用双级Alpha遮罩(bilevel alpha mask)。

cairo.**ANTIALIAS_GRAY**
使用单色抗锯齿（例如使用白色背景上黑色文本的灰度）。

cairo.**ANTIALIAS_SUBPIXEL**
通过利用LCD面板等设备的亚像素渲染特性实现抗锯齿效果。（译注：相关内容可以参考 亚像素显示）

cairo.CONTENT

这些常量用于描述 *Surface* 包含的内容，即颜色信息、alpha信息（半透明vs不透明度），或者两者都有。

cairo.**CONTENT_COLOR**
surface 只包含颜色信息。

cairo.**CONTENT_ALPHA**
surface 只包含alpha通道信息。

cairo.**CONTENT_COLOR_ALPHA**
surface 包含颜色和alpha信息。

cairo.EXTEND

这些常量用来描述 *Pattern* 对于超出pattern正常区域的颜色/alpha如何渲染的问题，（例如：超出surface边界或者超出渐变区域边界）。

对于 *SurfacePattern* 默认的模式是 *EXTEND_NONE* ， 对于 *Gradient* 的pattern， 默认的模式是 *EXTEND_PAD* 。

cairo.**EXTEND_NONE**
超过source pattern边界的像素完全透明。

cairo.**EXTEND_REPEAT**
（超出部分）pattern被重复的平铺。

cairo.**EXTEND_REFLECT**
pattern通过反射边缘平铺（>=1.6版本实现）。 the pattern is tiled by reflecting at the edges (Implemented for surface patterns since 1.6)

cairo.**EXTEND_PAD**
超出pattern边界部分的像素直接拷贝与其相邻的边界部分的像素（>=1.2版本，只有>=1.6版本的surface pattern实现）。

未来的版本可能会增加新的模式。

cairo.FILL_RULE

这些常量用于选择path填充的模式。对于所有这两种模式，一个点是否被填充取决于： 从该点向无穷远画一条射线，该射线与path的交叉点。射线的方向任意，只要交叉点不通过path某一段的终点， 或者与path相切（事实上填充并不是按照这种方式实现的，这只是填充规则的描述）。

默认的填充规则是 *FILL_RULE_WINDING* 。

cairo.**FILL_RULE_WINDING**
如果路径从左向右穿过射线，count+1，如果路径从右向左穿过射线，count-1。（左和右由射线的起点看起），如果count最终非0，该点被填充。

If the path crosses the ray from left-to-right, counts +1. If the path crosses the ray from right to left, counts -1. (Left and right are determined from the perspective of looking along the ray from the starting point.) If the total count is non-zero, the point will be filled.

cairo.**FILL_RULE_EVEN_ODD**
不考虑方向，统计交叉点的总数，如果总数是奇数，则该点被填充。

未来的版本可能会增加新的模式。

cairo.FILTER

这些常量用于描述渲染pattern的像素点时使用的filter。 函数 *SurfacePattern.set_filter()* 可以设置pattern要使用的filter。

cairo.**FILTER_FAST**
一个高性能的filter，质量与 *FILTER_NEAREST* 差不多。

cairo.**FILTER_GOOD**
性能一般的filter，质量与 *FILTER_BILINEAR* 差不多。

cairo.**FILTER_BEST**
最高质量的可用的filter，性能可能并不适合交互界面使用。

cairo.**FILTER_NEAREST**
近邻过滤（Nearest-neighbor filtering）。

cairo.**FILTER_BILINEAR**
二维线性差值（Linear interpolation in two dimensions）。

cairo.**FILTER_GAUSSIAN**
该值当前未实现，当前代码不应该使用。

cairo.FONT_SLANT

这些常量用于描述 *FontFace* 的倾斜度。

cairo.**FONT_SLANT_NORMAL**
正常的字体风格。

cairo.**FONT_SLANT_ITALIC**
斜体风格（Italic font style）。

cairo.**FONT_SLANT_OBLIQUE**
伪斜体风格（Oblique font style）。

译注：参考 伪斜体

cairo.FONT_WEIGHT

这些常量描述了 *FontFace* 的字体的粗细。

cairo.**FONT_WEIGHT_NORMAL**
正常的自体粗细形式。

cairo.**FONT_WEIGHT_BOLD**
粗体形式。

cairo.FORMAT

这些常量描述:class:*ImageSurface* 数据在内存中的格式。

未来的版本可能会增加新的格式。

cairo.**FORMAT_ARGB32**
每个像素32位，高8位描述alpha通道，然后是红、绿、蓝三原色。该32位的值以本地字节序存储。 最终的渲染会使用预乘过的alpha值（即红色50%透明的值是0x80800000，而不是0x80ff0000）。

cairo.**FORMAT_RGB24**
每个像素点是一个32位的值，但高8位没有使用，红、绿、蓝三原色依序存储在余下的24位中。

cairo.**FORMAT_A8**
每个像素点8位——存储alpha通道的值。

cairo.**FORMAT_A1**
每个像素使用1位，用于存储alpha值。很多像素点一起打包成32位的数，位序按照平台序来定。在大端机器上， 第一个像素在最高位，在小端机器上第一个像素在最低位。

cairo.**FORMAT_RGB16_565**
每个像素16位，其中依序红色在高5位，绿色在中间6位，蓝色在低5位。

cairo.HINT_METRICS

这些常量描述是否微调字体规格（hint font metrics），意即量化字体规格，字体在显示设备空间的表示为整数。这样做可以 提高字母和线段间距的连续性，但是也意味着文字在不同的缩放程度时的布局可能会不同。

cairo.**HINT_METRICS_DEFAULT**
以字体后端和目标设备默认的方式微调字体。

cairo.**HINT_METRICS_OFF**
不微调字体规格。

cairo.**HINT_METRICS_ON**
微调字体规格。

cairo.HINT_STYLE

这些常量描述对字体轮廓微调的类型。微调的主要工作是将字体的轮廓过滤映射到像素栅格以提升外观。 由于微调后的轮廓可能会与原来的稍有不同，因此渲染出来的字体可能并不完全忠实于原来的外形轮廓。 并不是所有的微调类型被所有字体后端支持。

cairo.**HINT_STYLE_DEFAULT**
使用字体后端及目标设备的默认类型。

cairo.**HINT_STYLE_NONE**
不微调字体廓量。

cairo.**HINT_STYLE_SLIGHT**
轻微的调整来提高对比度，尽量忠实与原始形状。

cairo.**HINT_STYLE_MEDIUM**
中度的调整在忠实于原始形状与对比度之间尽量折衷。

cairo.**HINT_STYLE_FULL**
调整轮廓以获取尽可能大的对比度。

为了版本可能会增加新的类型。

cairo.LINE_CAP

这些常量描述在一次绘画(stroke)时如何处理路径的端点。

默认的形式是 *LINE_CAP_BUTT*

cairo.**LINE_CAP_BUTT**
在开始（结束）点停止线段。

cairo.**LINE_CAP_ROUND**
在端点增加圆角，圆心就是线段的端点。

cairo.**LINE_CAP_SQUARE**
在端点增加一个正方形，正方形的中心在线段的端点。

cairo.LINE_JOIN

这些常量描述当完成一次绘画时如何渲染两条线段的交叉点。

默认的样式是 *LINE_JOIN_MITER*

cairo.**LINE_JOIN_MITER**
圆角的交叉点，参考： *Context.set_miter_limit()*

cairo.**LINE_JOIN_ROUND**
圆角的交叉点，圆心正是交叉点。

cairo.**LINE_JOIN_BEVEL**
斜角的交叉点，在一半线宽的位置切掉交叉点的尖角。

cairo.OPERATOR

这些常量用于设置cairo绘画操作的合成操作。

默认的操作符是 *OPERATOR_OVER*.

标记为 *unbounded* 的操作符即使超出了屏蔽层（mask layer）也会修改，即屏蔽层并不能限制这些效果的范围。 但是通过clip这样的方式仍然能限制有效的范围。

为了使事情变的简单，此处只记录了这些操作符在源和目的或者都透明或者都不透明时的行为， 其实实现对半透明效果也支持。

要获取每个操作符更加详细的信息，包括其数学原理，请参考： http://cairographics.org/operators.

cairo.**OPERATOR_CLEAR**
完全清除目的层 (bounded)

cairo.**OPERATOR_SOURCE**
完全替换目的层(bounded)

cairo.**OPERATOR_OVER**
在目的层的上面绘制源的内容(bounded)

cairo.**OPERATOR_IN**
在目的层有内容的地方绘制源。(unbounded)

cairo.**OPERATOR_OUT**
在目的层没有内容的地方绘制源。(unbounded)

cairo.**OPERATOR_ATOP**
只在目的层内容的上面绘制源。

cairo.**OPERATOR_DEST**
忽略源。

cairo.**OPERATOR_DEST_OVER**
在源上面绘制目的层。

cairo.**OPERATOR_DEST_IN**
只在源有内容的地方保留目的层。(unbounded)

cairo.**OPERATOR_DEST_OUT**
只在源没有内容的地方保留目的层。

cairo.**OPERATOR_DEST_ATOP**
只保留源有内容的地方的目的层内容。(unbounded)

cairo.**OPERATOR_XOR**
源和目的执行异或操作（只有源或者目的时绘制）。

cairo.**OPERATOR_ADD**
源和目的层累积。

cairo.**OPERATOR_SATURATE**
与OPERATOR_OVER类似，但是假定源和目的几何上分离（but assuming source and dest are disjoint geometries）。

cairo.PATH

这些常量描述 *Path* 的类型。

These constants are used to describe the type of one portion of a path when represented as a *Path*.

cairo.**PATH_MOVE_TO**
move-to 操作

cairo.**PATH_LINE_TO**
line-to 操作

cairo.**PATH_CURVE_TO**
curve-to 操作

cairo.**PATH_CLOSE_PATH**
close-path 操作

cairo.PS_LEVEL

这些常量描述生成的PostScript文件的版本。注意：只有cairo编译时开启了PS支持时才会定义这些常量。

cairo.**PS_LEVEL_2**
PostScript level 2

cairo.**PS_LEVEL_3**
PostScript level 3

cairo.SUBPIXEL_ORDER

亚像素顺序（The subpixel order）描述了在显示设备上开启抗锯齿模式 *ANTIALIAS_SUBPIXEL* 时 每个像素点色彩元素的顺序。

cairo.**SUBPIXEL_ORDER_DEFAULT**
使用目的显示设备默认的亚像素顺序。

cairo.**SUBPIXEL_ORDER_RGB**
亚像素按照红色在做的顺序水平排列。

cairo.**SUBPIXEL_ORDER_BGR**
亚像素按照蓝像素在做的顺序水平排列。

cairo.**SUBPIXEL_ORDER_VRGB**
亚像素按照红色在顶的顺序竖直排列。

cairo.**SUBPIXEL_ORDER_VBGR**
亚像素按照蓝色在顶的顺序竖直排列。

Cairo Context

class Context()

Context 是你使用cairo绘制时主要用到的对象。当你要使用cairo绘制时，你首先创建一个 *Context* 上下文，设置目标surface及上下文的选项，调用 *Context.move_to()* 等方法 创建shape形状，然后调用 *Context.stroke()* 或者 *Context.fill()* 将形状绘制到surface。

Contexts 可以通过 *Context.save()* 保存到栈上，然后你就可以安全的修改上下文而不用担心丢失任何状态了。 完成之后调用 *Context.restore()* 来恢复之前保存的状态。

class cairo.**Context**(*target*)

参数 **target** – 上下文的目标 *Surface*

返回 一个新分配的上下文

Raises 没有内存时产生 *MemoryError* 异常

创建一个新的 *Context*，所有的状态参数设置为默认值，并以 *target* 作为目地surface，目的surface应该使用 后端特定的函数来构建，比如： *ImageSurface* （或者其他的cairo后端surface的构造函数）

append_path(*path*)

参数 **path** – *Path* to be appended

将 *path* 添加到当前的路径path。 *path* 可以是由 *Context.copy_path()* 或者 *Context.copy_path_flat()* 返回，也可能是手动构建的（使用C语言）。

arc(*xc*, *yc*, *radius*, *angle1*, *angle2*)

参数

- **xc** (*float*) – 圆心的X坐标
- **yc** (*float*) – 圆心的Y坐标
- **radius** (*float*) – 圆的半径
- **angle1** (*float*) – 起始角度，以弧度表示
- **angle2** (*float*) – 结束角度，以弧度表示

以给定的半径 *radius* 在当前的path路径上添加一个圆弧。圆弧以(*xc, yc*)为圆心，以 *angle1* 角度为起始点， 按照角度增加的方向直到 *angle2* 结束，如果 *angle2* 小于 *angle1* ，则会绘制整个的圆（角度增长 2*PI 直到大于 *angle1* ）。

如果有一个当前的点，那么一条从当前点到圆弧起点的线段会添加到路径。 如果你不想绘制这条线段，可以在调用 *Context.arc()* 之前先调用 *Context.new_sub_path()* 。

角度以弧度为单位。角度0是X轴的正方向（user space），角度PI/2.0（90度）是Y轴的正方向（user space），角度从 X轴方向到Y轴正方向的方向增长，因此在默认的转换矩阵（default transformation matrix）下角度以顺时针方向增长。

要从角度转换到弧度，使用 `degrees * (math.pi / 180)` 。

本函数以角度正向增长的方向绘制圆弧，如果需要向相反的方向绘制请参考： *Context.arc_negative()* 。

圆弧的绘制在user space是圆形的。要想绘制椭圆，你可以在X和Y方向以不同的数量缩放（scale）当前的转换矩阵（current transformation matrix）。 例如要绘制 *x, y, width, height 大小的椭圆，可以使用如下的代码:

```
ctx.save()
ctx.translate(x + width / 2., y + height / 2.)
ctx.scale(width / 2., height / 2.)
ctx.arc(0., 0., 1., 0., 2 * math.pi)
ctx.restore()
```

arc_negative(*xc*, *yc*, *radius*, *angle1*, *angle2*)

参数

- **xc** (*float*) – 圆弧圆心X坐标

- **yc** (*float*) – 圆弧圆心Y坐标
- **radius** (*float*) – 圆弧的半径
- **angle1** (*float*) – 圆弧的起始角度
- **angle2** (*float*) – 圆弧的结束角度

以给定的半径 *radius* 在当前的path路径上添加一个圆弧。圆弧以(*xc, yc*)为圆心，以 *angle1* 角度为起始点， 按照角度减小的方向直到 *angle2* 结束，如果 *angle2* 大于 *angle1* ，则会绘制整个的圆（角度减小 2*PI 直到小于 *angle1* ）。

详细信息参考 *Context.arc()* 。这两个函数唯一的不同在于两个角度间圆弧的方向。

clip()

使用当前的path路径建立一个新的裁切区域，旧的裁切区域会根据当前的 *FILL RULE* （参考 *Context.set_fill_rule()* ） 调用 *Context.fill()* 填充。

调用 *clip()* 后当前的路径path会从 *Context* 清除。

裁切区域会影响所有的绘制操作——掩盖掉所有在当前绘制区域之外的改变。

调用 *clip()* 只能创建更小的裁切区域，无法创建更大大的裁切区域。 但是当前裁切区域也是绘制状态（graphics state）的一部分，因此通过 *clip()* 创建临时的裁切区域前后，你可能需要调用 *Context.save()*/*Context.restore()* 。 其他的增加裁切区域的方法只有 *Context.reset_clip()* 。

clip_extents()

返回 (x1, y1, x2, y2)

返回类型 (float, float, float, float)

- *x1*: 返回范围的左边界
- *y1*: 返回范围的顶部边界
- *x2*: 返回范围的右边界
- *y2*: 返回范围的底部边界

计算覆盖当前裁切区域的一个边界范围的坐标。

1.4 新版功能.

clip_preserve()

使用当前的path路径建立一个新的裁切区域，旧的裁切区域会根据当前的 *FILL RULE* （参考 *Context.set_fill_rule()* ） 调用 *Context.fill()* 填充。

与 *Context.clip()* 不同， *clip_preserve()* 会保存 *Context* 的当前路径path。

裁切区域会影响所有的绘制操作——掩盖掉所有在当前绘制区域之外的改变。

调用 *clip_preserve()* 只能创建更小的裁切区域，无法创建更大大的裁切区域。 但是当前裁切区域也是绘制状态（graphics state）的一部分，因此通过 *clip()* 创建临时的裁切区域前后，你可能需要调用 *Context.save()*/*Context.restore()* 。 其他的增加裁切区域的方法只有 *Context.reset_clip()* 。

close_path()

从当前点（最近调用 *Context.move_to()* 传递的点）到当前子路径的起点添加一条线段， 闭合当前的子路径。调用完成之后，连接到的子路径的断点成为新的当前点。

在一次绘制stroke时， *close_path()* 的行为与以等价的终点坐标调用 *Context.line_to()* 并不相同。 闭合的子路径在一次绘制stroke时， 在子路径的终点并没有'盖'（cap）（译注：见

cairo.LINE_CAP ）， 而是使用线段连接line join（译注：见 *cairo.LINE_CAP* ）将子路径 的起点和终点连接起来。

如果在调用 `close_path()` 时没有当前绘制点（current point），调用将没有任何效果。

Note: 1.2.4版本的cairo在任何调用 `close_path()` 时在路径闭合后都会执行 MOVE_TO， （例子参考 `Context.copy_path()` ）。这在某些情况下可以简化路径的处理——因为这样在处理时 就不用保存最后移动到的点了，因为 MOVE_TO 会提供这个点。

copy_clip_rectangle_list()

返回 当前裁切区域的矩形坐标列表

返回类型 四个浮点数的元组的列表

(列表中的 status 可能是 %CAIRO_STATUS_CLIP_NOT_REPRESENTABLE，意指裁切区域不能用一个用户态矩形（user-space rectangles）代表。 status 也可能是指代其他错误的值。——pycairo中未实现)

1.4 新版功能.

copy_page()

对于支持多页操作的后端，发射（emits，疑应翻译为显示？）当前页，但是内容并不会被清除，因此当前页的内容会保留给下一个页。 如果你想在发射后得到一个空的页，需要调用 `Context.show_page()` 。

本函数是一个便捷函数，只是调用了 *Context* 的目标设备的 `Surface.copy_page()` 函数。

copy_path()

返回 `Path`

Raises 没有内存时触发 *MemoryError* 异常

创建当前路径的一个拷贝并且以 `Path` 返回给用户。

copy_path_flat()

返回 `Path`

Raises 没有内存时触发 *MemoryError* 异常

获取当前路径的一个 flattened 的拷贝并且以 `Path` 返回给用户。

本函数与 `Context.copy_path()` 类似，但是路径中所有的曲线都以分段线性的方式被线性化（使用当前的容错值，current tolerance value）。即结果肯定不含有CAIRO_PATH_CURVE_TO类型的元素， 所有这种类型的元素都被替换为一系列的CAIRO_PATH_LINE_TO元素了。

curve_to(*x1*, *y1*, *x2*, *y2*, *x3*, *y3*)

:param x1:第一个控制点的X坐标 :type x1: float :param y1: 第一个控制点的Y坐标 :type y1: float :param x2: 第二个控制点的X坐标 :type x2: float :param y2: 第二个控制点的Y坐标 :type y2: float :param x3: 第三个控制点的X坐标 :type x3: float :param y3: 第三个控制点的Y坐标 :type y3: float

从当前点到 *(x3, y3)* 点添加一条贝塞尔曲线（cubic Bézier spline），使用 *(x1, y1)* 和 *(x2, y2)* 作为控制点。坐标均未用户态的值，调用完成之后当前点移动到 *(x3, y3)* 。

如果在调用 `curve_to()` 当前点没有被设置，函数的行为类似于调用 `ctx.move_to(x1, y1)` 函数。

device_to_user(*x*, *y*)

参数

- **x** (*float*) – 坐标的X值
- **y** (*float*) – 坐标的Y值

返回 (x, y)

返回类型 (float, float)

通过将给定点乘以前转换矩阵（current transformation matrix，CTM）的逆矩阵将设备空间的坐标转换为用户空间的坐标。

device_to_user_distance(*dx*, *dy*)

参数

- **dx** (*float*) – 距离向量的X值
- **dy** (*float*) – 距离向量的Y值

返回 (dx, dy)

返回类型 (float, float)

将设备空间的距离向量转换到用户空间。 本函数与 *Context.device_to_user()* 类似，但是CTM的逆矩阵的转换相关部分（translation components）会被忽略。

fill()

根据当前的填充规则（*FILL RULE*）填充当前路径的一个绘制操作，在绘制前，所有的子路径都被隐式的闭合了。 在调用 *fill()* 之后，当前的路径会从 class:*Context* 清除。请参考 *Context.set_fill_rule()* 和 *Context.fill_preserve()* 。

fill_extents()

返回 (x1, y1, x2, y2)

返回类型 (float, float, float, float)

- *x1*: 返回填充延展区域的左边界
- *y1*: 返回填充延展区域的上边界
- *x2*: 返回填充延展区域的右边界
- *y2*: 返回填充延展区域的下边界

计算当前路径的 *Context.fill()* 操作会影响（即填充）的区域的边界盒子的用户空间坐标。如果当前路径为空，则返回一个空的矩形(0,0,0,0)，surface区域和裁切并未考虑在内（Surface dimensions and）。

与 *Context.path_extents()* 的区别是 *Context.path_extents()* 对于没有填充区域的一些路径（例如一条简单的线段）返回 非零的扩展区域。

请注意 *fill_extents()* 函数需要做更多的操作才能计算出精确的填充操作被填充的的区域的大小， 因此对于性能要求比较高的地方 *Context.path_extents()* 可能更适合。

参考 *Context.fill()* ， *Context.set_fill_rule()* 和 *Context.fill_preserve()*.

fill_preserve()

根据当前的填充规则（*FILL RULE*）填充当前路径的一个绘制操作，在绘制前，所有的子路径都被隐式的闭合了。 但是不像 *Context.fill()* ， *fill_preserve()* 会保留 *Context* 的路径。

参考 *Context.set_fill_rule()* and *Context.fill()*.

font_extents()

返回 (ascent, descent, height, max_x_advance, max_y_advance)

返回类型 (float, float, float, float, float)

获取当前选择字体的延展区域（extents）。

get_antialias()

返回 当前的 *ANTIALIAS* 模式（由 *Context.set_antialias()* 设置）。

get_current_point()

返回 (x, y)

返回类型 (float, float)

- *x*: 当前点的X坐标
- *y*: 当前点的Y坐标

获取当前路径的当前点，理论上来讲就是路径目前的终点。

返回当前点的用户态坐标。如果没有当前点，或者 *Context* 处在一个错误的状态， *x* 和 *y* 都返回0.0。 可以通过 *Context.has_current_point()* 来检查当前点。

绝大多数的路径构建函数都会修改当前点的位置，参考以下函数以了解这些操作对当前点的详细影响： *Context.new_path()*, *Context.new_sub_path()*, *Context.append_path()*, *Context.close_path()*, *Context.move_to()*, *Context.line_to()*, *Context.curve_to()*, *Context.rel_move_to()*, *Context.rel_line_to()*, *Context.rel_curve_to()*, *Context.arc()*, *Context.arc_negative()*, *Context.rectangle()*, *Context.text_path()*, *Context.glyph_path()*, Context.stroke_to_path().

以下函数会修改当前点但是并不改变当前的路径： *Context.show_text()*.

以下函数会删除（unset）当前的路径，因此也会删除单签点： *Context.fill()*, *Context.stroke()*.

get_dash()

返回 (dashes, offset)

返回类型 (tuple, float)

- *dashes*: dash数组
- *offset*: 当前dash的偏移值。

获取当前的dash数组。

1.4 新版功能.

get_dash_count()

返回 dash数组的长度，如果dash数组没有被设置则返回0。

返回类型 int

参考 *Context.set_dash()* 和 *Context.get_dash()*.

1.4 新版功能.

get_fill_rule()

返回 当前的 *FILL RULE* （由 *Context.set_fill_rule()* 函数设置）。

get_font_face()

返回 *Context* 当前的 *FontFace* 。

get_font_matrix()

返回 *Context* 当前的 *Matrix* 。

参考 *Context.set_font_matrix()*.

get_font_options()

返回 *Context* 当前的 *FontOptions* 。

获取 *Context.set_font_options()* 设置的字体渲染选项。 注意返回的选项并不包含从底层surface继承的选项；从字面意思来看返回的就是 传递给 *Context.set_font_options()* 的选项。

get_group_target()

返回 the target *Surface*.

返回 *Context* 的当前目的 *Surface* ，或者是传递给 *Context* 的 原来的目的，或者是最近调用 *Context.push_group()* 或者 *Context.push_group_with_content()* 设置的当前组的目的surface。

1.2 新版功能.

get_line_cap()

返回 当前的 *LINE_CAP* 风格，由 *Context.set_line_cap()* 设置。

get_line_join()

返回 当前的 *LINE_JOIN* 风格，由 *Context.set_line_join()* 设置。

get_line_width()

返回 当前的线宽

返回类型 float

本函数返回由 *Context.set_line_width()* 设置的当前的线宽。 注意即使CTM在调用 *Context.set_line_width()* 之后已经被改变，返回的值也不变。

get_matrix()

返回 当前转换矩阵 *Matrix* (CTM)

get_miter_limit()

返回 当前的斜切限制，由 *Context.set_miter_limit()* 设置。

返回类型 float

get_operator()

返回 *Context* 的当前合成操作 *OPERATOR* 。

get_scaled_font()

返回 *Context* 当前的 *ScaledFont* 。

1.4 新版功能.

get_source()

返回 *Context* 的当前pattern源 *Pattern* 。

get_target()

返回 *Context* 的目的surface *Surface* 。

get_tolerance()

返回 当前的容错值（the current tolerance value），由 *Context.set_tolerance()* 设置。

返回类型 float

glyph_extents(*glyphs*[, *num_glyphs*])

参数

- **glyphs** (*a sequence of (int, float, float)*) – glyphs
- **num_glyphs** (*int*) – number of glyphs to measure, defaults to using all

返回 x_bearing, y_bearing, width, height, x_advance, y_advance

返回类型 6-tuple of float

Gets the extents for an array of glyphs. The extents describe a user-space rectangle that encloses the "inked" portion of the glyphs, (as they would be drawn by *Context.show_glyphs()*). Additionally, the x_advance and y_advance values indicate the amount by which the current point would be advanced by *Context.show_glyphs()*.

Note that whitespace glyphs do not contribute to the size of the rectangle (extents.width and extents.height).

glyph_path(*glyphs*[, *num_glyphs*])

参数

- **glyphs** (*a sequence of (int, float, float)*) – glyphs to show
- **num_glyphs** (*int*) – number of glyphs to show, defaults to showing all

Adds closed paths for the glyphs to the current path. The generated path if filled, achieves an effect similar to that of *Context.show_glyphs()*.

has_current_point()

returns: True iff a current point is defined on the current path. See *Context.get_current_point()* for details on the current point.

1.6 新版功能.

identity_matrix()

Resets the current transformation *Matrix* (CTM) by setting it equal to the identity matrix. That is, the user-space and device-space axes will be aligned and one user-space unit will transform to one device-space unit.

in_fill(*x*, *y*)

参数

- **x** (*float*) – 测试点的X坐标
- **y** (*float*) – 测试点的Y坐标

返回 如果该点在当前路径的 *Context.fill()* 操作的影响区域内返回True， surface的尺寸和裁切并没有考虑在内。

参考 *Context.fill()*， *Context.set_fill_rule()* 和 *Context.fill_preserve()*。

in_stroke(*x*, *y*)

参数

- **x** (*float*) – X coordinate of the point to test
- **y** (*float*) – Y coordinate of the point to test

返回 True iff the point is inside the area that would be affected by a *Context.stroke()* operation given the current path and stroking parameters. Surface dimensions and clipping are not taken into account.

See *Context.stroke()*, *Context.set_line_width()*, *Context.set_line_join()*, *Context.set_line_cap()*, *Context.set_dash()*, and *Context.stroke_preserve()*.

line_to(*x*, *y*)

参数

- **x** (*float*) – the X coordinate of the end of the new line
- **y** (*float*) – the Y coordinate of the end of the new line

Adds a line to the path from the current point to position *(x, y)* in user-space coordinates. After this call the current point will be *(x, y)*.

If there is no current point before the call to *line_to()* this function will behave as `ctx.move_to(x, y)`.

mask(*pattern*)

参数 **pattern** – a *Pattern*

A drawing operator that paints the current source using the alpha channel of *pattern* as a mask. (Opaque areas of *pattern* are painted with the source, transparent areas are not painted.)

mask_surface(*surface*, *x=0.0*, *y=0.0*)

参数

- **surface** – a *Surface*
- **x** (*float*) – X coordinate at which to place the origin of *surface*
- **y** (*float*) – Y coordinate at which to place the origin of *surface*

A drawing operator that paints the current source using the alpha channel of *surface* as a mask. (Opaque areas of *surface* are painted with the source, transparent areas are not painted.)

move_to(*x*, *y*)

参数

- **x** (*float*) – the X coordinate of the new position
- **y** (*float*) – the Y coordinate of the new position

Begin a new sub-path. After this call the current point will be *(x, y)*.

new_path()
Clears the current path. After this call there will be no path and no current point.

new_sub_path()
Begin a new sub-path. Note that the existing path is not affected. After this call there will be no current point.

In many cases, this call is not needed since new sub-paths are frequently started with *Context.move_to()*.

A call to *new_sub_path()* is particularly useful when beginning a new sub-path with one of the *Context.arc()* calls. This makes things easier as it is no longer necessary to manually compute the arc's initial coordinates for a call to *Context.move_to()*.

1.6 新版功能.

paint()
A drawing operator that paints the current source everywhere within the current clip region.

paint_with_alpha(*alpha*)

参数 **alpha** (*float*) – alpha value, between 0 (transparent) and 1 (opaque)

A drawing operator that paints the current source everywhere within the current clip region using a mask of constant alpha value *alpha*. The effect is similar to *Context.paint()*, but the drawing is faded out using the alpha value.

path_extents()

返回 (x1, y1, x2, y2)

返回类型 (float, float, float, float)

- *x1*: left of the resulting extents
- *y1*: top of the resulting extents
- *x2*: right of the resulting extents
- *y2*: bottom of the resulting extents

Computes a bounding box in user-space coordinates covering the points on the current path. If the current path is empty, returns an empty rectangle (0, 0, 0, 0). Stroke parameters, fill rule, surface dimensions and clipping are not taken into account.

Contrast with *Context.fill_extents()* and *Context.stroke_extents()* which return the extents of only the area that would be "inked" by the corresponding drawing operations.

The result of *path_extents()* is defined as equivalent to the limit of *Context.stroke_extents()* with cairo.LINE_CAP_ROUND as the line width approaches 0.0, (but never reaching the empty-rectangle returned by *Context.stroke_extents()* for a line width of 0.0).

Specifically, this means that zero-area sub-paths such as *Context.move_to()*; *Context.line_to()* segments, (even degenerate cases where the coordinates to both calls are identical), will be considered as contributing to the extents. However, a lone *Context.move_to()* will not contribute to the results of *Context.path_extents()*.

1.6 新版功能.

pop_group()

返回 a newly created *SurfacePattern* containing the results of all drawing operations performed to the group.

Terminates the redirection begun by a call to *Context.push_group()* or *Context.push_group_with_content()* and returns a new pattern containing the results of all drawing operations performed to the group.

The *pop_group()* function calls *Context.restore()*, (balancing a call to *Context.save()* by the *Context.push_group()* function), so that any changes to the graphics state will not be visible outside the group.

1.2 新版功能.

pop_group_to_source()
Terminates the redirection begun by a call to *Context.push_group()* or *Context.push_group_with_content()* and installs the resulting pattern as the source *Pattern* in the given *Context*.

The behavior of this function is equivalent to the sequence of operations:

```
group = cairo_pop_group()
ctx.set_source(group)
```

but is more convenient as their is no need for a variable to store the short-lived pointer to the pattern.

The *Context.pop_group()* function calls *Context.restore()*, (balancing a call to *Context.save()* by the *Context.push_group()* function), so that any changes to the graphics state will not be visible outside the group.

1.2 新版功能.

push_group()

Temporarily redirects drawing to an intermediate surface known as a group. The redirection lasts until the group is completed by a call to *Context.pop_group()* or *Context.pop_group_to_source()*. These calls provide the result of any drawing to the group as a pattern, (either as an explicit object, or set as the source pattern).

This group functionality can be convenient for performing intermediate compositing. One common use of a group is to render objects as opaque within the group, (so that they occlude each other), and then blend the result with translucence onto the destination.

Groups can be nested arbitrarily deep by making balanced calls to *Context.push_group()*/*Context.pop_group()*. Each call pushes/pops the new target group onto/from a stack.

The *push_group()* function calls *Context.save()* so that any changes to the graphics state will not be visible outside the group, (the pop_group functions call *Context.restore()*).

By default the intermediate group will have a *CONTENT* type of cairo.CONTENT_COLOR_ALPHA. Other content types can be chosen for the group by using *Context.push_group_with_content()* instead.

As an example, here is how one might fill and stroke a path with translucence, but without any portion of the fill being visible under the stroke:

```
ctx.push_group()
ctx.set_source(fill_pattern)
ctx.fill_preserve()
ctx.set_source(stroke_pattern)
ctx.stroke()
ctx.pop_group_to_source()
ctx.paint_with_alpha(alpha)
```

1.2 新版功能.

push_group_with_content(*content*)

参数 **content** – a *CONTENT* indicating the type of group that will be created

Temporarily redirects drawing to an intermediate surface known as a group. The redirection lasts until the group is completed by a call to *Context.pop_group()* or *Context.pop_group_to_source()*. These calls provide the result of any drawing to the group as a pattern, (either as an explicit object, or set as the source pattern).

The group will have a content type of *content*. The ability to control this content type is the only distinction between this function and *Context.push_group()* which you should see for a more detailed description of group rendering.

1.2 新版功能.

rectangle(*x*, *y*, *width*, *height*)

参数

- **x** (*float*) – the X coordinate of the top left corner of the rectangle
- **y** (*float*) – the Y coordinate to the top left corner of the rectangle
- **width** (*float*) – the width of the rectangle
- **height** (*float*) – the height of the rectangle

Adds a closed sub-path rectangle of the given size to the current path at position *(x, y)* in user-space coordinates.

This function is logically equivalent to:

```
ctx.move_to(x, y)
ctx.rel_line_to(width, 0)
ctx.rel_line_to(0, height)
ctx.rel_line_to(-width, 0)
ctx.close_path()
```

rel_curve_to(*dx1*, *dy1*, *dx2*, *dy2*, *dx3*, *dy4*)

参数

- **dx1** (*float*) – the X offset to the first control point
- **dy1** (*float*) – the Y offset to the first control point
- **dx2** (*float*) – the X offset to the second control point
- **dy2** (*float*) – the Y offset to the second control point
- **dx3** (*float*) – the X offset to the end of the curve
- **dy3** (*float*) – the Y offset to the end of the curve

Raises *cairo.Error* if called with no current point.

Relative-coordinate version of *Context.curve_to()*. All offsets are relative to the current point. Adds a cubic Bézier spline to the path from the current point to a point offset from the current point by *(dx3, dy3)*, using points offset by *(dx1, dy1)* and *(dx2, dy2)* as the control points. After this call the current point will be offset by *(dx3, dy3)*.

Given a current point of (x, y), `ctx.rel_curve_to(dx1, dy1, dx2, dy2, dx3, dy3)` is logically equivalent to `ctx.curve_to(x+dx1, y+dy1, x+dx2, y+dy2, x+dx3, y+dy3)`.

rel_line_to(*dx*, *dy*)

参数

- **dx** (*float*) – the X offset to the end of the new line
- **dy** (*float*) – the Y offset to the end of the new line

Raises *cairo.Error* if called with no current point.

Relative-coordinate version of *Context.line_to()*. Adds a line to the path from the current point to a point that is offset from the current point by *(dx, dy)* in user space. After this call the current point will be offset by *(dx, dy)*.

Given a current point of (x, y), `ctx.rel_line_to(dx, dy)` is logically equivalent to `ctx.line_to(x + dx, y + dy)`.

rel_move_to(*dx*, *dy*)

参数

- **dx** (*float*) – the X offset
- **dy** (*float*) – the Y offset

Raises *cairo.Error* if called with no current point.

Begin a new sub-path. After this call the current point will offset by *(dx, dy)*.

Given a current point of (x, y), `ctx.rel_move_to(dx, dy)` is logically equivalent to `ctx.(x + dx, y + dy)`.

reset_clip()

Reset the current clip region to its original, unrestricted state. That is, set the clip region to an infinitely large shape containing the target surface. Equivalently, if infinity is too hard to grasp, one can imagine the clip region being reset to the exact bounds of the target surface.

Note that code meant to be reusable should not call *reset_clip()* as it will cause results unexpected by higher-level code which calls *clip()*. Consider using *save()* and *restore()* around *clip()* as a more robust means of temporarily restricting the clip region.

restore()

Restores *Context* to the state saved by a preceding call to *save()* and removes that state from the stack of saved states.

rotate(*angle*)

参数 **angle** (*float*) – angle (in radians) by which the user-space axes will be rotated

Modifies the current transformation matrix (CTM) by rotating the user-space axes by *angle* radians. The rotation of the axes takes places after any existing transformation of user space. The rotation direction for positive angles is from the positive X axis toward the positive Y axis.

save()

Makes a copy of the current state of *Context* and saves it on an internal stack of saved states. When *restore()* is called, *Context* will be restored to the saved state. Multiple calls to *save()* and *restore()* can be nested; each call to *restore()* restores the state from the matching paired *save()*.

scale(*sx*, *sy*)

参数

- **sx** (*float*) – scale factor for the X dimension
- **sy** (*float*) – scale factor for the Y dimension

Modifies the current transformation matrix (CTM) by scaling the X and Y user-space axes by *sx* and *sy* respectively. The scaling of the axes takes place after any existing transformation of user space.

select_font_face(*family*[, *slant*[, *weight*]])

参数

- **family** (*str*) – a font family name
- **slant** – the *FONT_SLANT* of the font, defaults to *cairo.FONT_SLANT_NORMAL*.
- **weight** – the *FONT_WEIGHT* of the font, defaults to *cairo.FONT_WEIGHT_NORMAL*.

Note: The *select_font_face()* function call is part of what the cairo designers call the "toy" text API. It is convenient for short demos and simple programs, but it is not expected to be adequate for serious text-using applications.

Selects a family and style of font from a simplified description as a family name, slant and weight. Cairo provides no operation to list available family names on the system (this is a "toy", remember), but the standard CSS2 generic family names, ("serif", "sans-serif", "cursive", "fantasy", "monospace"), are likely to work as expected.

For "real" font selection, see the font-backend-specific font_face_create functions for the font backend you are using. (For example, if you are using the freetype-based cairo-ft font backend, see cairo_ft_font_face_create_for_ft_face() or cairo_ft_font_face_create_for_pattern().) The resulting font face could then be used with cairo_scaled_font_create() and cairo_set_scaled_font().

Similarly, when using the "real" font support, you can call directly into the underlying font system, (such as fontconfig or freetype), for operations such as listing available fonts, etc.

It is expected that most applications will need to use a more comprehensive font handling and text layout library, (for example, pango), in conjunction with cairo.

If text is drawn without a call to *select_font_face()*, (nor *set_font_face()* nor *set_scaled_font()*), the default family is platform-specific, but is essentially "sans-serif". Default slant is cairo.FONT_SLANT_NORMAL, and default weight is cairo.FONT_WEIGHT_NORMAL.

This function is equivalent to a call to *ToyFontFace* followed by *set_font_face()*.

set_antialias (*antialias*)

参数 **antialias** – the new *ANTIALIAS* mode

Set the antialiasing mode of the rasterizer used for drawing shapes. This value is a hint, and a particular backend may or may not support a particular value. At the current time, no backend supports *cairo.ANTIALIAS_SUBPIXEL* when drawing shapes.

Note that this option does not affect text rendering, instead see *FontOptions.set_antialias()*.

set_dash (*dashes*[, *offset=0*])

参数

- **dashes** (*sequence of float*) – a sequence specifying alternate lengths of on and off stroke portions.
- **offset** (*int*) – an offset into the dash pattern at which the stroke should start, defaults to 0.

Raises *cairo.Error* if any value in *dashes* is negative, or if all values are 0.

Sets the dash pattern to be used by *stroke()*. A dash pattern is specified by *dashes* - a sequence of positive values. Each value provides the length of alternate "on" and "off" portions of the stroke. The *offset* specifies an offset into the pattern at which the stroke begins.

Each "on" segment will have caps applied as if the segment were a separate sub-path. In particular, it is valid to use an "on" length of 0.0 with *cairo.LINE_CAP_ROUND* or *cairo.LINE_CAP_SQUARE* in order to distributed dots or squares along a path.

Note: The length values are in user-space units as evaluated at the time of stroking. This is not necessarily the same as the user space at the time of *set_dash()*.

If the number of dashes is 0 dashing is disabled.

If the number of dashes is 1 a symmetric pattern is assumed with alternating on and off portions of the size specified by the single value in *dashes*.

set_fill_rule (*fill_rule*)

参数 **fill_rule** – 要设置的 *FILL RULE* 。填充规则用于 决定一个区域是属于还是不属于一个复杂的路径（潜在的自相交等）。当前的填充规则会影响 *fill()* 和 *clip()* 。

默认的填充规则是 *cairo.FILL_RULE_WINDING* 。

set_font_face(*font_face*)

参数 **font_face** – a *FontFace*, or None to restore to the default *FontFace*

Replaces the current *FontFace* object in the *Context* with *font_face*.

set_font_matrix(*matrix*)

参数 **matrix** – a *Matrix* describing a transform to be applied to the current font.

Sets the current font matrix to *matrix*. The font matrix gives a transformation from the design space of the font (in this space, the em-square is 1 unit by 1 unit) to user space. Normally, a simple scale is used (see *set_font_size()*), but a more complex font matrix can be used to shear the font or stretch it unequally along the two axes

set_font_options(*options*)

参数 **options** – *FontOptions* to use

Sets a set of custom font rendering options for the *Context*. Rendering options are derived by merging these options with the options derived from underlying surface; if the value in *options* has a default value (like *cairo.ANTIALIAS_DEFAULT*), then the value from the surface is used.

set_font_size(*size*)

参数 **size** (*float*) – the new font size, in user space units

Sets the current font matrix to a scale by a factor of *size*, replacing any font matrix previously set with *set_font_size()* or *set_font_matrix()*. This results in a font size of *size* user space units. (More precisely, this matrix will result in the font's em-square being a *size* by *size* square in user space.)

If text is drawn without a call to *set_font_size()*, (nor *set_font_matrix()* nor *set_scaled_font()*), the default font size is 10.0.

set_line_cap(*line_cap*)

参数 **line_cap** – a *LINE_CAP* style

Sets the current line cap style within the *Context*.

As with the other stroke parameters, the current line cap style is examined by *stroke()*, *stroke_extents()*, and `stroke_to_path()`, but does not have any effect during path construction.

The default line cap style is *cairo.LINE_CAP_BUTT*.

set_line_join(*line_join*)

参数 **line_join** – a *LINE_JOIN* style

Sets the current line join style within the *Context*.

As with the other stroke parameters, the current line join style is examined by *stroke()*, *stroke_extents()*, and `stroke_to_path()`, but does not have any effect during path construction.

The default line join style is *cairo.LINE_JOIN_MITER*.

set_line_width(*width*)

参数 **width** (*float*) – a line width

Sets the current line width within the *Context*. The line width value specifies the diameter of a pen that is circular in user space, (though device-space pen may be an ellipse in general due to scaling/shear/rotation of the CTM).

Note: When the description above refers to user space and CTM it refers to the user space and CTM in effect at the time of the stroking operation, not the user space and CTM in effect at the time of the call to *set_line_width()*. The simplest usage makes both of these spaces identical. That is, if there is no change to the CTM between a call to *set_line_width()* and the stroking operation, then one can just pass user-space values to *set_line_width()* and ignore this note.

As with the other stroke parameters, the current line width is examined by *stroke()*, *stroke_extents()*, and stroke_to_path(), but does not have any effect during path construction.

The default line width value is 2.0.

set_matrix(*matrix*)

参数 **matrix** – a transformation *Matrix* from user space to device space.

Modifies the current transformation matrix (CTM) by setting it equal to *matrix*.

set_miter_limit(*limit*)

参数 **limit** – miter limit to set

Sets the current miter limit within the *Context*.

If the current line join style is set to *cairo.LINE_JOIN_MITER* (see *set_line_join()*), the miter limit is used to determine whether the lines should be joined with a bevel instead of a miter. Cairo divides the length of the miter by the line width. If the result is greater than the miter limit, the style is converted to a bevel.

As with the other stroke parameters, the current line miter limit is examined by *stroke()*, *stroke_extents()*, and stroke_to_path(), but does not have any effect during path construction.

The default miter limit value is 10.0, which will convert joins with interior angles less than 11 degrees to bevels instead of miters. For reference, a miter limit of 2.0 makes the miter cutoff at 60 degrees, and a miter limit of 1.414 makes the cutoff at 90 degrees.

A miter limit for a desired angle can be computed as:

```
miter limit = 1/math.sin(angle/2)
```

set_operator(*op*)

参数 **op** – the compositing *OPERATOR* to set for use in all drawing operations.

The default operator is *cairo.OPERATOR_OVER*.

set_scaled_font(*scaled_font*)

参数 **scaled_font** – a *ScaledFont*

Replaces the current font face, font matrix, and font options in the *Context* with those of the *ScaledFont*. Except for some translation, the current CTM of the *Context* should be the same as that of the *ScaledFont*, which can be accessed using *ScaledFont.get_ctm()*.

1.2 新版功能.

set_source(*source*)

参数 **source** – a *Pattern* to be used as the source for subsequent drawing operations.

Sets the source pattern within *Context* to *source*. This pattern will then be used for any subsequent drawing operation until a new source pattern is set.

Note: The pattern's transformation matrix will be locked to the user space in effect at the time of *set_source()*. This means that further modifications of the current transformation matrix will not affect the source pattern. See *Pattern.set_matrix()*.

The default source pattern is a solid pattern that is opaque black, (that is, it is equivalent to `set_source_rgb(0.0, 0.0, 0.0)`.

set_source_rgb(*red*, *green*, *blue*)

参数

- **red** (*float*) – red component of color
- **green** (*float*) – green component of color
- **blue** (*float*) – blue component of color

Sets the source pattern within *Context* to an opaque color. This opaque color will then be used for any subsequent drawing operation until a new source pattern is set.

The color components are floating point numbers in the range 0 to 1. If the values passed in are outside that range, they will be clamped.

The default source pattern is opaque black, (that is, it is equivalent to `set_source_rgb(0.0, 0.0, 0.0)`.

set_source_rgba(*red*, *green*, *blue*[, *alpha=1.0*])

参数

- **red** (*float*) – red component of color
- **green** (*float*) – green component of color
- **blue** (*float*) – blue component of color
- **alpha** (*float*) – alpha component of color

Sets the source pattern within *Context* to a translucent color. This color will then be used for any subsequent drawing operation until a new source pattern is set.

The color and alpha components are floating point numbers in the range 0 to 1. If the values passed in are outside that range, they will be clamped.

The default source pattern is opaque black, (that is, it is equivalent to `set_source_rgba(0.0, 0.0, 0.0, 1.0)`.

set_source_surface(*surface*[, *x=0.0*[, *y=0.0*]])

参数

- **surface** – a *Surface* to be used to set the source pattern
- **x** (*float*) – User-space X coordinate for surface origin
- **y** (*float*) – User-space Y coordinate for surface origin

This is a convenience function for creating a pattern from a *Surface* and setting it as the source in *Context* with *set_source()*.

The *x* and *y* parameters give the user-space coordinate at which the surface origin should appear. (The surface origin is its upper-left corner before any transformation has been applied.) The *x* and *y* patterns are negated and then set as translation values in the pattern matrix.

Other than the initial translation pattern matrix, as described above, all other pattern attributes, (such as its extend mode), are set to the default values as in *SurfacePattern*. The resulting pattern can be queried with *get_source()* so that these attributes can be modified if desired, (eg. to create a repeating pattern with *Pattern.set_extend()*).

set_tolerance(*tolerance*)

参数 **tolerance** (*float*) – the tolerance, in device units (typically pixels)

Sets the tolerance used when converting paths into trapezoids. Curved segments of the path will be subdivided until the maximum deviation between the original path and the polygonal approximation is less than *tolerance*. The default value is 0.1. A larger value will give better performance, a smaller value, better appearance. (Reducing the value from the default value of 0.1 is unlikely to improve appearance significantly.) The accuracy of paths within Cairo is limited by the precision of its internal arithmetic, and the prescribed *tolerance* is restricted to the smallest representable internal value.

show_glyphs(*glyphs*[, *num_glyphs*])

参数

- **glyphs** (*a sequence of (int, float, float)*) – glyphs to show
- **num_glyphs** (*int*) – number of glyphs to show, defaults to showing all glyphs

A drawing operator that generates the shape from an array of glyphs, rendered according to the current font face, font size (font matrix), and font options.

show_page()

Emits and clears the current page for backends that support multiple pages. Use *copy_page()* if you don't want to clear the page.

This is a convenience function that simply calls `ctx.get_target() . show_page()`

show_text(*text*)

参数 **text** (*str*) – text

A drawing operator that generates the shape from a string of text, rendered according to the current font_face, font_size (font_matrix), and font_options.

This function first computes a set of glyphs for the string of text. The first glyph is placed so that its origin is at the current point. The origin of each subsequent glyph is offset from that of the previous glyph by the advance values of the previous glyph.

After this call the current point is moved to the origin of where the next glyph would be placed in this same progression. That is, the current point will be at the origin of the final glyph offset by its advance values. This allows for easy display of a single logical string with multiple calls to *show_text()*.

Note: The *show_text()* function call is part of what the cairo designers call the "toy" text API. It is convenient for short demos and simple programs, but it is not expected to be adequate for serious text-using applications. See *show_glyphs()* for the "real" text display API in cairo.

stroke()

A drawing operator that strokes the current path according to the current line width, line join, line cap, and dash settings. After *stroke()*, the current path will be cleared from the cairo context. See *set_line_width()*, *set_line_join()*, *set_line_cap()*, *set_dash()*, and *stroke_preserve()*.

Note: Degenerate segments and sub-paths are treated specially and provide a useful result. These can result in two different situations:

1. Zero-length "on" segments set in *set_dash()*. If the cap style is *cairo.LINE_CAP_ROUND* or *cairo.LINE_CAP_SQUARE* then these segments will be drawn as circular dots or squares respectively.

In the case of *cairo.LINE_CAP_SQUARE*, the orientation of the squares is determined by the direction of the underlying path.

2. A sub-path created by *move_to()* followed by either a *close_path()* or one or more calls to *line_to()* to the same coordinate as the *move_to()*. If the cap style is *cairo.LINE_CAP_ROUND* then these sub-paths will be drawn as circular dots. Note that in the case of *cairo.LINE_CAP_SQUARE* a degenerate sub-path will not be drawn at all, (since the correct orientation is indeterminate).

In no case will a cap style of *cairo.LINE_CAP_BUTT* cause anything to be drawn in the case of either degenerate segments or sub-paths.

stroke_extents()

返回 (x1, y1, x2, y2)

返回类型 (float, float, float, float)

- *x1*: left of the resulting extents
- *y1*: top of the resulting extents
- *x2*: right of the resulting extents
- *y2*: bottom of the resulting extents

Computes a bounding box in user coordinates covering the area that would be affected, (the "inked" area), by a *stroke()* operation given the current path and stroke parameters. If the current path is empty, returns an empty rectangle (0, 0, 0, 0). Surface dimensions and clipping are not taken into account.

Note that if the line width is set to exactly zero, then *stroke_extents()* will return an empty rectangle. Contrast with *path_extents()* which can be used to compute the non-empty bounds as the line width approaches zero.

Note that *stroke_extents()* must necessarily do more work to compute the precise inked areas in light of the stroke parameters, so *path_extents()* may be more desirable for sake of performance if non-inked path extents are desired.

See *stroke()*, *set_line_width()*, *set_line_join()*, *set_line_cap()*, *set_dash()*, and *stroke_preserve()*.

stroke_preserve()

A drawing operator that strokes the current path according to the current line width, line join, line cap, and dash settings. Unlike *stroke()*, *stroke_preserve()* preserves the path within the cairo context.

See *set_line_width()*, *set_line_join()*, *set_line_cap()*, *set_dash()*, and *stroke_preserve()*.

text_extents(*text*)

参数 **text** (*str*) – text to get extents for

返回 x_bearing, y_bearing, width, height, x_advance, y_advance

返回类型 6-tuple of float

Gets the extents for a string of text. The extents describe a user-space rectangle that encloses the "inked" portion of the text, (as it would be drawn by *Context.show_text()*). Additionally, the x_advance and y_advance values indicate the amount by which the current point would be advanced by *Context.show_text()*.

Note that whitespace characters do not directly contribute to the size of the rectangle (extents.width and extents.height). They do contribute indirectly by changing the position of non-whitespace characters. In

particular, trailing whitespace characters are likely to not affect the size of the rectangle, though they will affect the x_advance and y_advance values.

text_path(*text*)

参数 **text** (*str*) – text

Adds closed paths for text to the current path. The generated path if filled, achieves an effect similar to that of *Context.show_text()*.

Text conversion and positioning is done similar to *Context.show_text()*.

Like *Context.show_text()*, After this call the current point is moved to the origin of where the next glyph would be placed in this same progression. That is, the current point will be at the origin of the final glyph offset by its advance values. This allows for chaining multiple calls to to *Context.text_path()* without having to set current point in between.

Note: The *text_path()* function call is part of what the cairo designers call the "toy" text API. It is convenient for short demos and simple programs, but it is not expected to be adequate for serious text-using applications. See *Context.glyph_path()* for the "real" text path API in cairo.

transform(*matrix*)

参数 **matrix** – a transformation *Matrix* to be applied to the user-space axes

Modifies the current transformation matrix (CTM) by applying *matrix* as an additional transformation. The new transformation of user space takes place after any existing transformation.

translate(*tx*, *ty*)

参数

- **tx** (*float*) – amount to translate in the X direction
- **ty** (*float*) – amount to translate in the Y direction

Modifies the current transformation matrix (CTM) by translating the user-space origin by *(tx, ty)*. This offset is interpreted as a user-space coordinate according to the CTM in place before the new call to *translate()*. In other words, the translation of the user-space origin takes place after any existing transformation.

user_to_device(*x*, *y*)

参数

- **x** (*float*) – X value of coordinate
- **y** (*float*) – Y value of coordinate

返回 (x, y)

返回类型 (float, float)

- *x*: X value of coordinate
- *y*: Y value of coordinate

Transform a coordinate from user space to device space by multiplying the given point by the current transformation matrix (CTM).

user_to_device_distance(*dx*, *dy*)

参数

- **dx** (*float*) – X value of a distance vector

- **dy** (*float*) – Y value of a distance vector

返回 (dx, dy)

返回类型 (float, float)

- *dx*: X value of a distance vector
- *dy*: Y value of a distance vector

Transform a distance vector from user space to device space. This function is similar to *Context.user_to_device()* except that the translation components of the CTM will be ignored when transforming *(dx,dy)*.

Exceptions

当一个cairo函数或方法调用失败时会触发异常。I/O错误会触发IOError，内存错误会触发MemoryError，其他的错误会触发cairo.Error。

cairo.Error()

exception cairo.**Error**

当一个cairo对象返回一个错误的状态值时触发本异常。

Matrix

class Matrix()

Matrix 在cairo中被广泛的用于不同坐标系统间的转换。一个 *Matrix* 代表一个仿射变换，例如 缩放、旋转、剪辑（shear）或以上几种的组合。点 (x, y) 的转换通过如下方式给出:

```
x_new = xx * x + xy * y + x0
y_new = yx * x + yy * y + y0
```

Context 的当前转换矩阵CTX即 *Matrix* ，定义了从用户空间坐标到设备空间坐标的转换。

一些标准的Python操作可以被用于矩阵 matrix:

读取矩阵 *Matrix* 的值:

```
xx, yx, xy, yy, x0, y0 = matrix
```

两个矩阵相乘:

```
matrix3 = matrix1.multiply(matrix2)
# or equivalently
matrix3 = matrix1 * matrix2
```

比较两个矩阵:

```
matrix1 == matrix2
matrix1 != matrix2
```

更多矩阵转换相关的内容请参考 http://www.cairographics.org/matrix_transform （译注，矩阵相关的文章 http://blog.csdn.net/xiaojidan2011/article/details/8213873 ）

class cairo.**Matrix**(*xx = 1.0*, *yx = 0.0*, *xy = 0.0*, *yy = 1.0*, *x0 = 0.0*, *y0 = 0.0*)

参数

- **xx** (*float*) – 仿射变换的xx组件
- **yx** (*float*) – 仿射变换的yx组件
- **xy** (*float*) – 仿射变换的xy组件
- **yy** (*float*) – 仿射变换的yy组件
- **x0** (*float*) – 仿射变换的X组件
- **y0** (*float*) – 仿射变换的Y组件

使用 *xx, yx, xy, yy, x0, y0* 定义的仿射变换创建一个矩阵 *Matrix* 。仿射变换的公式如下:

```
x_new = xx * x + xy * y + x0
y_new = yx * x + yy * y + y0
```

创建一个新的独立的矩阵:

```
matrix = cairo.Matrix()
```

要创建一个在X和Y轴变换(translates by)tx和ty的矩阵，可以向下面这样:

```
matrix = cairo.Matrix(x0=tx, y0=ty)
```

要创建一个在X和Y轴缩放(scale by)tx和ty的矩阵，可以向下面这样:

```
matrix = cairo.Matrix(xx=sy, yy=sy)
```

classmethod init_rotate(*radians*)

参数 radians (*float*) – 旋转的角度，单位为弧度。旋转的方向从正X轴到正Y轴为正向的角度。 再考虑到cairo中轴默认的方向，正角度为时钟旋转的方向。

返回 设置旋转角度为 *radians* 的新的 *Matrix* 。

invert()

返回 如果 *Matrix* 有逆矩阵（inverse），修改 *Matrix* 为其逆矩阵并返回None。

Raises 如果 *Matrix* 没有逆矩阵触发 *cairo.Error* 异常。

将矩阵 *Matrix* 转换为其逆矩阵，并不是所有的转换矩阵都有逆矩阵，如果 那么本函数会失败。（if the matrix collapses points together (it is *degenerate*), then it has no inverse and this function will fail.）

multiply(*matrix2*)

参数 matrix2 (cairo.Matrix) – 另一个矩阵

返回 一个新的 *Matrix*

两个仿射矩阵 *Matrix* 和 *matrix2* 相乘。新矩阵产生的效果为先将第一个矩阵变换 *Matrix* 作用于坐标再 将第二个矩阵 *matrix2* 作用于坐标。

结果与 *Matrix* 或 *matrix2* 不相同也是可以接受的。

It is allowable for result to be identical to either *Matrix* or *matrix2*.

rotate(*radians*)

参数 **radians** (*float*) – 旋转的角度，单位为弧度。旋转的方向从正X轴到正Y轴为正向的角度。 再考虑到cairo中轴默认的方向，正角度为时钟旋转的方向。

初始化 *Matrix* 为一个旋转 *radians* 弧度的一个转换。

scale(*sx*, *sy*)

参数

- **sx** (*float*) – X方向的缩放因子
- **sy** (*float*) – Y方向的缩放因子

对 *Matrix* 中的转换应用 *sx, sy* 缩放。新的转换的效果为首先以 *sx* 和 *sy* 缩放坐标， 然后对坐标应用原来的转换。

transform_distance(*dx*, *dy*)

参数

- **dx** (*float*) – 空间向量的X组件
- **dy** (*float*) – 空间向量的Y组件

返回 转换后的空间向量 (dx,dy)

返回类型 (float, float)

使用矩阵 *Matrix* 转换空间向量 *(dx,dy)* 。其类似于 *transform_point()* ，但是转换的转换组件被忽略。 （except that the translation components of the transformation are ignored.） 返回的向量是如此计算得出的:

```
dx2 = dx1 * a + dy1 * c
dy2 = dx1 * b + dy1 * d
```

仿射变换是位置不变的，因此相同的向量总是转换为相同的向量。如果 *(x1,y1)* 转换为 *(x2,y2)* 那么对于所有的 *x1* 和 *x2 (x1+dx1,y1+dy1)* 会转换为 *(x1+dx2,y1+dy2)* 。

Affine transformations are position invariant, so the same vector always transforms to the same vector. If *(x1,y1)* transforms to *(x2,y2)* then *(x1+dx1,y1+dy1)* will transform to *(x1+dx2,y1+dy2)* for all values of *x1* and *x2*.

transform_point(*x*, *y*)

参数

- **x** (*float*) – 点的X坐标
- **y** (*float*) – 点的Y坐标

返回 转换后的点 (x,y)

返回类型 (float, float)

使用 *Matrix* 转换点 *(x, y)* 。

translate(*tx*, *ty*)

参数

- **tx** (*float*) – X方向转换的数量
- **ty** (*float*) – Y方向转换的数量

对 *Matrix* 的转换应用 *tx, ty* 转换。新的转换的效果为先以 *tx* 和 *ty* 转换坐标，再 应用原来的变换。

Paths

class Path()

class cairo.**Path**

Path 不能被直接实例化，只能通过调用 *Context.copy_path()* 和 *Context.copy_path_flat()* 创建。

str(path) 会列出路径的所有元素。

参考 *PATH attributes*

Path 是一个迭代器。

例子用法参考 examples/warpedtext.py 。

Pattern

Pattern是cairo绘图的颜料，其主要应用时作为所有cairo绘制操作的源，当然也可以用于mask，即画笔。

一个cairo *Pattern* 由以下列出的任何一个 *PatternType* 的构造函数来创建，或者可以隐式的通过 *Context.set_source_<type>()* 方法来创建。

class Pattern()

Pattern 是所有其他pattern类继承的抽象基类，不能直接被实例化。

class cairo.**Pattern**

get_extend()

返回 绘制 *Pattern* 时当前的扩展模式（ *EXTEND* 属性 ）。

返回类型 int

获取 *Pattern* 当前的扩展模式。每种扩展模式的详细信息参考 *EXTEND* 属性 。

get_matrix()

返回 存储了 *Pattern* 的仿射变换矩阵信息的类 *Matrix* 。

set_extend(*extend*)

参数 **extend** – 描述 *Pattern* 外围区域如何绘制的 *EXTEND* 。

设置绘制 *Pattern* 外围区域时使用的模式。

对于 *SurfacePattern* 和 *cairo.EXTEND_PAD* 默认的模式是 *cairo.EXTEND_NONE* 。 对于 *Gradient* Pattern 默认的模式时 *cairo.EXTEND_PAD* 。

set_matrix(*matrix*)

参数 **matrix** – *Matrix* 的实例

设置 *Pattern* 的变换矩阵为 *matrix* 。这个矩阵用于从用户空间转换到pattern空间。

当 *Patter* 第一次被创建时总是会创建一个唯一的变换矩阵用于其仿射变换，因此 pattern空间和用户空间从一开始就是不同的。

重要: 请注意这个变化矩阵的方向是从用户空间到pattern空间，这意味着如果你想要从 pattern空间到用户空间（即设备空间），那么其坐标变换使用的是 *Pattern* 矩阵的反转矩阵。

例如，如果你想要创建一个比之前大一倍的 *Pattern* ， 正确的代码应该是:

```
matrix = cairo.Matrix(xx=0.5,yy=0.5)
pattern.set_matrix(matrix)
```

如果在上面的代码中使用值2.0代替0.5，会使得 *Patter* 是原来大小的一半。

另外，需要注意的时用户空间的讨论仅限于 *Context.set_source* 语义范围内。

class SolidPattern(Pattern)

class cairo.**SolidPattern**(*red*, *green*, *blue*, *alpha=1.0*)

参数

- **red** (*float*) – 颜色的红色组件的值
- **green** (*float*) – 颜色的绿色组件的值
- **blue** (*float*) – 颜色的蓝色组件的值
- **alpha** (*float*) – 颜色的alpha组件的值

返回 一个新的 *SolidPattern* 的实例

Raises 没有内存时触发 *MemoryError* 异常

使用参数中的半透明颜色创建一个新的 *SolidPattern* 。颜色组件为0~1的浮点数。 如果传递的参数超过这个范围，则使用最接近该值的范围内的值（clamped）。

get_rgba()

返回 (red, green, blue, alpha) 浮点数元组

获取 *SolidPattern* 的颜色。

1.4 新版功能.

class SurfacePattern(Pattern)

class cairo.**SurfacePattern**(*surface*)

参数 **surface** – cairo *Surface*

返回 使用参数surface新创建的 *SurfacePattern*

Raises 没有内存时触发 *MemoryError* 异常

get_filter()

返回 当前用于调整 *SurfacePattern* 的 *FILTER* 。

get_surface()

返回 *SurfacePattern* 的 *Surface*

1.4 新版功能.

set_filter(*filter*)

参数 **filter** – 用于描述如何调整 *Pattern* 的 *FILTER* 。

注意：即使你并没有使用 *Pattern* 时（例如使用 *Context.set_source_surface()* 时）也想要控制调整的模式。 这种情况下使用 *Context.get_source()* 来访问 cairo 隐式创建的 pattern 更方便。 例如:

```
context.set_source_surface(image, x, y)
surfacepattern.set_filter(context.get_source(), cairo.FILTER_NEAREST)
```

class Gradient(Pattern)

Pattern 是其他pattern类继承的抽象基类，不能直接被实例化。

class cairo.**Gradient**

add_color_stop_rgb(*offset*, *red*, *green*, *blue*)

参数

- **offset** (*float*) – [0.0 .. 1.0] 范围内的偏移值
- **red** (*float*) – 颜色的红色组件的值
- **green** (*float*) – 颜色的绿色组件的值
- **blue** (*float*) – 颜色的蓝色组件的值

向 *Gradient* 添加一个不透明的颜色。 offset 为沿着渐变的控制向量的位置点。例如，线性渐变 *LinearGradient's* 的控制向量 从 (x0,y0) 到 (x1,y1)，而径向渐变 *RadialGradient's* 的控制向量从圆的起点到圆的终点。

颜色设置方法与 *Context.set_source_rgb()* 相同。

如果设置了两个（或更多个）点，则所有的停止点会根据添加顺序排序（添加早的点在前）。 这在想要创建尖锐的颜色渐变（sharp color transition）而非混合（blend）渐变时很有用。

add_color_stop_rgba(*offset*, *red*, *green*, *blue*, *alpha*)

参数

- **offset** (*float*) – [0.0 .. 1.0] 范围内的偏移值
- **red** (*float*) – 颜色的红色组件的值
- **green** (*float*) – 颜色的绿色组件的值
- **blue** (*float*) – 颜色的蓝色组件的值
- **alpha** – 颜色的alpha组件的值

向 *Gradient* 添加一个不透明的颜色。 （译注：原文如此，疑有误） offset 为沿着渐变的控制向量的位置点。例如，线性渐变 *LinearGradient's* 的控制向量 从 (x0,y0) 到 (x1,y1)，而径向渐变 *RadialGradient's* 的控制向量从圆的起点到圆的终点。

颜色设置方法与 *Context.set_source_rgb()* 相同。

如果设置了两个（或更多个）点，则所有的停止点会根据添加顺序排序（添加早的点在前）。 这在想要创建尖锐的颜色渐变（sharp color transition）而非混合（blend）渐变时很有用。

class LinearGradient(Gradient)

class cairo.**LinearGradient**(*x0*, *y0*, *x1*, *y1*)

参数

- **x0** (*float*) – 起始点的x坐标
- **y0** (*float*) – 起始点的y坐标
- **x1** (*float*) – 结束点的x坐标
- **y1** (*float*) – 结束点的y坐标

返回 *LinearGradient*

Raises 没有内存时触发 *MemoryError* 异常

根据 (x0, y0) 和 (x1,y1) 确定的直线创建一个新的 *LinearGradient* 。 在使用 *Gradient* pattern之前，需要使用 *Gradient.add_color_stop_rgb()* 或者 *Gradient.add_color_stop_rgba()* 定义一系列的停止点。

注意: 此处的坐标为pattern空间。对于一个新的 *Pattern* ， pattern空间与用户空间时不同的， 但是他们之间的关系可以通过 *Pattern.set_matrix()* 改变。

get_linear_points()

返回

(x0, y0, x1, y1) - 浮点数元组

- x0: 第一个点的x坐标
- y0: 第一个点的y坐标
- x1: 第二个点的x坐标
- y1: 第二个点的y坐标

获取 *LinearGradient* 渐变的端点坐标。

1.4 新版功能.

class RadialGradient(Gradient)

class cairo.**RadialGradient**(*cx0*, *cy0*, *radius0*, *cx1*, *cy1*, *radius1*)

参数

- **cx0** (*float*) – 起始圆心的x坐标
- **cy0** (*float*) – 起始圆心的y坐标
- **radius0** (*float*) – 起始圆的半径
- **cx1** (*float*) – 终点圆心的x坐标
- **cy1** (*float*) – 终点圆心的y坐标
- **radius1** (*float*) – 终点圆的半径

返回 新创建的 *RadialGradient*

Raises 没有内存时触发 *MemoryError* 异常

在(cx0, cy0, radius0) 和 (cx1, cy1, radius1) 两个圆之间创建一个新的 *RadialGradient* 径向渐变pattern。在使用 *Gradient* pattern之前， 需要使用 *Gradient.add_color_stop_rgb()* 或者 *Gradient.add_color_stop_rgba()* 定义一系列的停止点。

注意: 此处的坐标为pattern空间。对于一个新的 *Pattern* ， pattern空间与用户空间时不同的， 但是他们之间的关系可以通过 *Pattern.set_matrix()* 改变。

get_radial_circles()

返回

(x0, y0, r0, x1, y1, r1) - 一个浮点数元组

- x0: 起始圆心的x坐标
- y0: 起始圆心的y坐标
- r0: 起始圆的半径
- x1: 终点圆心的x坐标
- y1: 终点圆心的y坐标
- r1: 终点圆的半径

获取 *RadialGradient* 的端点的圆，每个圆以其圆心坐标和半径表示。

1.4 新版功能.

Surfaces

cairo.Surface is the abstract type representing all different drawing targets that cairo can render to. The actual drawings are performed using a *Context*.

A cairo.Surface is created by using backend-specific constructors of the form cairo.<XXX>Surface().

class Surface()

class cairo.**Surface**

Surface is the abstract base class from which all the other surface classes derive. It cannot be instantiated directly.

copy_page()

Emits the current page for backends that support multiple pages, but doesn't clear it, so that the contents of the current page will be retained for the next page. Use *show_page()* if you want to get an empty page after the emission.

Context.copy_page() is a convenience function for this.

1.6 新版功能.

create_similar(*content*, *width*, *height*)

参数

- **content** – the *CONTENT* for the new surface
- **width** (*int*) – width of the new surface, (in device-space units)
- **height** – height of the new surface (in device-space units)

返回 a newly allocated *Surface*.

Create a *Surface* that is as compatible as possible with the existing surface. For example the new surface will have the same fallback resolution and *FontOptions*. Generally, the new surface will also use the same backend, unless that is not possible for some reason.

Initially the surface contents are all 0 (transparent if contents have transparency, black otherwise.)

finish()
This method finishes the *Surface* and drops all references to external resources. For example, for the Xlib backend it means that cairo will no longer access the drawable, which can be freed. After calling finish() the only valid operations on a *Surface* are flushing and finishing it. Further drawing to the surface will not affect the surface but will instead trigger a *cairo.Error* exception.

flush()
Do any pending drawing for the *Surface* and also restore any temporary modification's cairo has made to the *Surface's* state. This method must be called before switching from drawing on the *Surface* with cairo to drawing on it directly with native APIs. If the *Surface* doesn't support direct access, then this function does nothing.

get_content()

返回 The *CONTENT* type of *Surface*, which indicates whether the *Surface* contains color and/or alpha information.

1.2 新版功能.

get_device_offset()

返回

(x_offset, y_offset) a tuple of float

- x_offset: the offset in the X direction, in device units
- y_offset: the offset in the Y direction, in device units

This method returns the previous device offset set by *set_device_offset()*.

1.2 新版功能.

get_fallback_resolution()

返回

(x_pixels_per_inch, y_pixels_per_inch) a tuple of float

- x_pixels_per_inch: horizontal pixels per inch
- y_pixels_per_inch: vertical pixels per inch

This method returns the previous fallback resolution set by *set_fallback_resolution()*, or default fallback resolution if never set.

1.8 新版功能.

get_font_options()

返回 a *FontOptions*

Retrieves the default font rendering options for the *Surface*. This allows display surfaces to report the correct subpixel order for rendering on them, print surfaces to disable hinting of metrics and so forth. The result can then be used with *ScaledFont*.

mark_dirty()
Tells cairo that drawing has been done to *Surface* using means other than cairo, and that cairo should reread any cached areas. Note that you must call *flush()* before doing such drawing.

mark_dirty_rectangle(*x*, *y*, *width*, *height*)

参数

- **x** (*int*) – X coordinate of dirty rectangle
- **y** (*int*) – Y coordinate of dirty rectangle

- **width** (*int*) – width of dirty rectangle
- **height** (*int*) – height of dirty rectangle

Like *mark_dirty()*, but drawing has been done only to the specified rectangle, so that cairo can retain cached contents for other parts of the surface.

Any cached clip set on the *Surface* will be reset by this function, to make sure that future cairo calls have the clip set that they expect.

set_device_offset(*x_offset*, *y_offset*)

参数

- **x_offset** (*float*) – the offset in the X direction, in device units
- **y_offset** (*float*) – the offset in the Y direction, in device units

Sets an offset that is added to the device coordinates determined by the CTM when drawing to *Surface*. One use case for this function is when we want to create a *Surface* that redirects drawing for a portion of an onscreen surface to an offscreen surface in a way that is completely invisible to the user of the cairo API. Setting a transformation via *Context.translate()* isn't sufficient to do this, since functions like *Context.device_to_user()* will expose the hidden offset.

Note that the offset affects drawing to the surface as well as using the surface in a source pattern.

set_fallback_resolution(*x_pixels_per_inch*, *y_pixels_per_inch*)

参数

- **x_pixels_per_inch** (*float*) – horizontal setting for pixels per inch
- **y_pixels_per_inch** (*float*) – vertical setting for pixels per inch

Set the horizontal and vertical resolution for image fallbacks.

When certain operations aren't supported natively by a backend, cairo will fallback by rendering operations to an image and then overlaying that image onto the output. For backends that are natively vector-oriented, this function can be used to set the resolution used for these image fallbacks, (larger values will result in more detailed images, but also larger file sizes).

Some examples of natively vector-oriented backends are the ps, pdf, and svg backends.

For backends that are natively raster-oriented, image fallbacks are still possible, but they are always performed at the native device resolution. So this function has no effect on those backends.

Note: The fallback resolution only takes effect at the time of completing a page (with *Context.show_page()* or *Context.copy_page()*) so there is currently no way to have more than one fallback resolution in effect on a single page.

The default fallback resoultion is 300 pixels per inch in both dimensions.

1.2 新版功能.

show_page()

Emits and clears the current page for backends that support multiple pages. Use *copy_page()* if you don't want to clear the page.

There is a convenience function for this that takes a *Context.show_page()*.

1.6 新版功能.

write_to_png(*fobj*)

参数 **fobj** (*str, file or file-like object*) – the file to write to

Raises *MemoryError* if memory could not be allocated for the operation

IOError if an I/O error occurs while attempting to write the file

Writes the contents of *Surface* to *fobj* as a PNG image.

class ImageSurface(Surface)

A *cairo.ImageSurface* provides the ability to render to memory buffers either allocated by cairo or by the calling code. The supported image formats are those defined in *FORMAT attributes*.

class cairo.**ImageSurface**(*format*, *width*, *height*)

参数

- **format** – *FORMAT* of pixels in the surface to create
- **width** – width of the surface, in pixels
- **height** – height of the surface, in pixels

返回 a new *ImageSurface*

Raises *MemoryError* in case of no memory

Creates an *ImageSurface* of the specified format and dimensions. Initially the surface contents are all 0. (Specifically, within each pixel, each color or alpha channel belonging to format will be 0. The contents of bits within a pixel, but not belonging to the given format are undefined).

classmethod create_for_data(*data*, *format*, *width*, *height*[, *stride*])

Not yet available in Python 3

classmethod create_from_png(*fobj*)

参数 **fobj** – a filename, file, or file-like object of the PNG to load.

返回 a new *ImageSurface* initialized the contents to the given PNG file.

static format_stride_for_width(*format*, *width*)

参数

- **format** – a cairo *FORMAT* value
- **width** – the desired width of an *ImageSurface* to be created.

返回 the appropriate stride to use given the desired format and width, or -1 if either the format is invalid or the width too large.

返回类型 int

This method provides a stride value that will respect all alignment requirements of the accelerated image-rendering code within cairo. Typical usage will be of the form:

```
stride = cairo.ImageSurface.format_stride_for_width (format, width)
surface = cairo.ImageSurface.create_for_data (data, format, width, height,␣
↪stride)
```

1.6 新版功能.

get_data()

Not yet available in Python 3

get_format()

返回 the *FORMAT* of the *ImageSurface*.

1.2 新版功能.

get_height()

返回 the height of the *ImageSurface* in pixels.

get_stride()

返回 the stride of the *ImageSurface* in bytes. The stride is the distance in bytes from the beginning of one row of the image data to the beginning of the next row.

get_width()

返回 the width of the *ImageSurface* in pixels.

class PDFSurface(Surface)

The PDFSurface is used to render cairo graphics to Adobe PDF files and is a multi-page vector surface backend.

class cairo.**PDFSurface**(*fobj*, *width_in_points*, *height_in_points*)

参数

- **fobj** (*None, str, file or file-like object*) – a filename or writable file object. None may be used to specify no output. This will generate a *PDFSurface* that may be queried and used as a source, without generating a temporary file.
- **width_in_points** (*float*) – width of the surface, in points (1 point == 1/72.0 inch)
- **height_in_points** (*float*) – height of the surface, in points (1 point == 1/72.0 inch)

返回 a new *PDFSurface* of the specified size in points to be written to *fobj*.

Raises *MemoryError* in case of no memory

1.2 新版功能.

set_size()

参数

- **width_in_points** (*float*) – new surface width, in points (1 point == 1/72.0 inch)
- **height_in_points** (*float*) – new surface height, in points (1 point == 1/72.0 inch)

Changes the size of a *PDFSurface* for the current (and subsequent) pages.

This function should only be called before any drawing operations have been performed on the current page. The simplest way to do this is to call this function immediately after creating the surface or immediately after completing a page with either *Context.show_page()* or *Context.copy_page()*.

1.2 新版功能.

class PSSurface(Surface)

The *PSSurface* is used to render cairo graphics to Adobe PostScript files and is a multi-page vector surface backend.

class cairo.**PSSurface**(*fobj*, *width_in_points*, *height_in_points*)

参数

- **fobj** (*None, str, file or file-like object*) – a filename or writable file object. None may be used to specify no output. This will generate a *PSSurface* that may be queried and used as a source, without generating a temporary file.
- **width_in_points** (*float*) – width of the surface, in points (1 point == 1/72.0 inch)
- **height_in_points** (*float*) – height of the surface, in points (1 point == 1/72.0 inch)

返回 a new *PDFSurface* of the specified size in points to be written to *fobj*.

Raises *MemoryError* in case of no memory

Note that the size of individual pages of the PostScript output can vary. See *set_size()*.

dsc_begin_page_setup()

This method indicates that subsequent calls to *dsc_comment()* should direct comments to the PageSetup section of the PostScript output.

This method call is only needed for the first page of a surface. It should be called after any call to *dsc_begin_setup()* and before any drawing is performed to the surface.

See *dsc_comment()* for more details.

1.2 新版功能.

dsc_begin_setup()

This function indicates that subsequent calls to *dsc_comment()* should direct comments to the Setup section of the PostScript output.

This function should be called at most once per surface, and must be called before any call to *dsc_begin_page_setup()* and before any drawing is performed to the surface.

See *dsc_comment()* for more details.

1.2 新版功能.

dsc_comment(*comment*)

参数 **comment** (*str*) – a comment string to be emitted into the PostScript output

Emit a comment into the PostScript output for the given surface.

The comment is expected to conform to the PostScript Language Document Structuring Conventions (DSC). Please see that manual for details on the available comments and their meanings. In particular, the %%IncludeFeature comment allows a device-independent means of controlling printer device features. So the PostScript Printer Description Files Specification will also be a useful reference.

The comment string must begin with a percent character (%) and the total length of the string (including any initial percent characters) must not exceed 255 characters. Violating either of these conditions will place *PSSurface* into an error state. But beyond these two conditions, this function will not enforce conformance of the comment with any particular specification.

The comment string should not have a trailing newline.

The DSC specifies different sections in which particular comments can appear. This function provides for comments to be emitted within three sections: the header, the Setup section, and the PageSetup section. Comments appearing in the first two sections apply to the entire document while comments in the BeginPageSetup section apply only to a single page.

For comments to appear in the header section, this function should be called after the surface is created, but before a call to *dsc_begin_setup()*.

For comments to appear in the Setup section, this function should be called after a call to *dsc_begin_setup()* but before a call to *dsc_begin_page_setup()*.

For comments to appear in the PageSetup section, this function should be called after a call to *dsc_begin_page_setup()*.

Note that it is only necessary to call *dsc_begin_page_setup()* for the first page of any surface. After a call to *Context.show_page()* or *Context.copy_page()* comments are unambiguously directed to the PageSetup section of the current page. But it doesn't hurt to call this function at the beginning of every page as that consistency may make the calling code simpler.

As a final note, cairo automatically generates several comments on its own. As such, applications must not manually generate any of the following comments:

Header section: %!PS-Adobe-3.0, %Creator, %CreationDate, %Pages, %BoundingBox, %DocumentData, %LanguageLevel, %EndComments.

Setup section: %BeginSetup, %EndSetup

PageSetup section: %BeginPageSetup, %PageBoundingBox, %EndPageSetup.

Other sections: %BeginProlog, %EndProlog, %Page, %Trailer, %EOF

Here is an example sequence showing how this function might be used:

```
surface = PSSurface (filename, width, height)
...
surface.dsc_comment (surface, "%%Title: My excellent document")
surface.dsc_comment (surface, "%%Copyright: Copyright (C) 2006 Cairo Lover")
...
surface.dsc_begin_setup (surface)
surface.dsc_comment (surface, "%%IncludeFeature: *MediaColor White")
...
surface.dsc_begin_page_setup (surface)
surface.dsc_comment (surface, "%%IncludeFeature: *PageSize A3")
surface.dsc_comment (surface, "%%IncludeFeature: *InputSlot LargeCapacity")
surface.dsc_comment (surface, "%%IncludeFeature: *MediaType Glossy")
surface.dsc_comment (surface, "%%IncludeFeature: *MediaColor Blue")
... draw to first page here ..
ctx.show_page (cr)
...
surface.dsc_comment (surface, "%%IncludeFeature:  PageSize A5");
...
```

1.2 新版功能.

get_eps()

返回 True iff the *PSSurface* will output Encapsulated PostScript.

1.6 新版功能.

static ps_level_to_string(*level*)

参数 **level** – a *PS_LEVEL*

返回 the string associated to given level.

返回类型 str

Raises *cairo.Error* if *level* isn't valid.

Get the string representation of the given *level*. See `ps_get_levels()` for a way to get the list of valid level ids.

1.6 新版功能.

restrict_to_level(*level*)

参数 **level** – a *PS_LEVEL*

Restricts the generated PostSript file to *level*. See `ps_get_levels()` for a list of available level values that can be used here.

This function should only be called before any drawing operations have been performed on the given surface. The simplest way to do this is to call this function immediately after creating the surface.

1.6 新版功能.

set_eps(*eps*)

参数 **eps** (*bool*) – True to output EPS format PostScript

If *eps* is True, the PostScript surface will output Encapsulated PostScript.

This function should only be called before any drawing operations have been performed on the current page. The simplest way to do this is to call this function immediately after creating the surface. An Encapsulated PostScript file should never contain more than one page.

1.6 新版功能.

set_size(*width_in_points*, *height_in_points*)

参数

- **width_in_points** (*float*) – new surface width, in points (1 point == 1/72.0 inch)
- **height_in_points** (*float*) – new surface height, in points (1 point == 1/72.0 inch)

Changes the size of a PostScript surface for the current (and subsequent) pages.

This function should only be called before any drawing operations have been performed on the current page. The simplest way to do this is to call this function immediately after creating the surface or immediately after completing a page with either *Context.show_page()* or *Context.copy_page()*.

1.2 新版功能.

class SVGSurface(Surface)

The *SVGSurface* is used to render cairo graphics to SVG files and is a multi-page vector surface backend

class cairo.**SVGSurface**(*fobj*, *width_in_points*, *height_in_points*)

参数

- **fobj** (*None, str, file or file-like object*) – a filename or writable file object. None may be used to specify no output. This will generate a *SVGSurface* that may be queried and used as a source, without generating a temporary file.
- **width_in_points** (*float*) – width of the surface, in points (1 point == 1/72.0 inch)
- **height_in_points** (*float*) – height of the surface, in points (1 point == 1/72.0 inch)

返回 a new *SVGSurface* of the specified size in points to be written to *fobj*.

Raises *MemoryError* in case of no memory

get_versions()
Not implemented in pycairo (yet)

restrict_to_version()
Not implemented in pycairo (yet)

version_to_string()
Not implemented in pycairo (yet)

class Win32Surface(Surface)

The Microsoft Windows surface is used to render cairo graphics to Microsoft Windows windows, bitmaps, and printing device contexts.

class cairo.**Win32Surface**(*hdc*)

参数 **hdc** (*int*) – the DC to create a surface for

返回 the newly created surface

Creates a cairo surface that targets the given DC. The DC will be queried for its initial clip extents, and this will be used as the size of the cairo surface. The resulting surface will always be of format cairo.FORMAT_RGB24, see *FORMAT attributes*.

class Win32PrintingSurface(Surface)

The Win32PrintingSurface is a multi-page vector surface type.

class cairo.**Win32PrintingSurface**(*hdc*)

参数 **hdc** (*int*) – the DC to create a surface for

返回 the newly created surface

Creates a cairo surface that targets the given DC. The DC will be queried for its initial clip extents, and this will be used as the size of the cairo surface. The DC should be a printing DC; antialiasing will be ignored, and GDI will be used as much as possible to draw to the surface.

The returned surface will be wrapped using the paginated surface to provide correct complex rendering behaviour; *cairo.Surface.show_page()* and associated methods must be used for correct output.

class XCBSurface(Surface)

The XCB surface is used to render cairo graphics to X Window System windows and pixmaps using the XCB library.

Note that the XCB surface automatically takes advantage of the X render extension if it is available.

class cairo.**XCBSurface**

参数

- **connection** – an XCB connection
- **drawable** – a X drawable
- **visualtype** – a X visualtype
- **width** – The surface width
- **height** – The surface height

Creates a cairo surface that targets the given drawable (pixmap or window).

注解: This methods works using xpyb.

set_size(*width*, *height*)

参数

- **width** – The width of the surface
- **height** – The height of the surface

Informs cairo of the new size of the X Drawable underlying the surface. For a surface created for a Window (rather than a Pixmap), this function must be called each time the size of the window changes. (For a subwindow, you are normally resizing the window yourself, but for a toplevel window, it is necessary to listen for ConfigureNotify events.)

A Pixmap can never change size, so it is never necessary to call this function on a surface created for a Pixmap.

class XlibSurface(Surface)

The XLib surface is used to render cairo graphics to X Window System windows and pixmaps using the XLib library.

Note that the XLib surface automatically takes advantage of X render extension if it is available.

class cairo.**XlibSurface**

注解: *XlibSurface* cannot be instantiated directly because Python interaction with Xlib would require open source Python bindings to Xlib which provided a C API. However, an *XlibSurface* instance can be returned from a function call when using pygtk http://www.pygtk.org/.

---

get_depth()

返回 the number of bits used to represent each pixel value.

1.2 新版功能.

get_height()

返回 the height of the X Drawable underlying the surface in pixels.

1.2 新版功能.

get_width()

返回 the width of the X Drawable underlying the surface in pixels.

1.2 新版功能.

Text

cairo有两种方式可以解析字体:

- 一种是cairo的'玩具'text API。这个玩具API使用UTF-8编码的文本并且其功能仅限制为 解析从左到右的文本，没有高级特性。这意味着像Hebrew，Arabic和Indic字体 这些最复杂的字体（script）并不在讨论范围内，也没有字符间距调整或字体位置相关的标记。 字体的选择也很受限制，同时也不能处理选择的字体并不包含要显示的文本的问题。 这个API起始真的就是一个玩具，用于测试和演示的目的，任何认真考虑的程序都应该避免使用该API。

- 另一种使用cairo底层text API的方式可以访问字形的名字。底层的API需要用户将文本转换为字形 索引和位置。这其实是一件很麻烦的事最好使用外部库来处理，比如pangocairo，Pango文本布局和解析库的一部分。 Pango的网站 http://www.pango.org/

class FontFace()

cairo.FontFace 定义了字体除字体大小和字体矩阵 （字体矩阵用于改变字体，在不同方向不同的切向或缩放字体）外的所有方面。 *Context* 可以通过:meth:*Context.set_font_face* 设置 *FontFace* ， 字体 大小和矩阵可以通过 *Context.set_font_size()* 和 *Context.set_font_matrix()* 来设置。

根据使用的字体后端，有多种 *FontFace* 。

class cairo.**FontFace**

> **注解:** 这个类不能被直接实例化，是由 *Context.get_font_face()* 返回的。

class FreeTypeFontFace(FontFace)

FreeType Fonts - 支持FreeType的字体

FreeType字体后端主要用于GNU/Linux 上的字体渲染，也可以用于其他平台。

> **注 解:** FreeType 字 体 在pycairo中 没 有 实 现 ， 因 为 目 前 并 没 有 提 供C API的FreeType（和fontconfig）的开源python绑定，如果有人有兴趣为pycairo添加FreeType支持也可以。

class ToyFontFace(FontFace)

cairo.ToyFontFace 相比与 *Context.select_font_face()* 可以创建一个独立于context的toy font 。 class can be used instead of *Context.select_font_face()* to create a toy font independently of a context.

class cairo.**ToyFontFace**(*family*[, *slant*[, *weight*]])

参数

- **family** (*str*) – font family名
- **slant** – 字体的 *FONT_SLANT* 属性， defaults to *cairo.FONT_SLANT_NORMAL*.
- **weight** – 字 体 的 *FONT_WEIGHT* 属 性 ， defaults to *cairo.FONT_WEIGHT_NORMAL*.

返回 一个新的 *ToyFontFace*

根据family、slant、weight三元组创建一个 *ToyFontFace* 。这些font face用于实现'toy' API。

如果family 为0长字符串 “”，则使用平台特定的默认family。默认的family可以使用 *get_family()* 来查询。

Context.select_font_face() 使用此接口来创建font face，参考该函数以了解toy font face的限制。

1.8.4 新版功能.

get_family()

返回 toy font的family

返回类型 str

1.8.4 新版功能.

get_slant()

返回 *FONT_SLANT* 的值

1.8.4 新版功能.

get_weight()

返回 *FONT_WEIGHT* 的值

1.8.4 新版功能.

class UserFontFace(FontFace)

user-font 特性允许cairo用户提供字体字形的绘制。这在实现非标准格式的字体时很有用，比如 SVG字体和Flash字体，但是也可以用于游戏和其他程序来绘制“有意思”的字体。

注解: UserFontFace 支持还没有添加到pcairo中。如果你需要在pycairo中使用这个特性，请向cairo的邮件列表发送消息或者向pycairo报告bug。

class ScaledFont()

ScaledFont 是一个缩放到特定尺寸和设备解析度的字体，其在使用底层字体很有用，例如 库或应用想要缓存缩放字体的引用来加速计算。

根据使用的字体后端，有多种缩放字体。

class cairo.**ScaledFont**(*font_face*, *font_matrix*, *ctm*, *options*)

参数

- **font_face** – *FontFace* 的实例
- **font_matrix** – 字体的字体空间到用户空间转换的矩阵 *Matrix* 。对于最简单的情况，一个N个点的字体，矩阵 只是缩放N，但是矩阵也可以用于在不同的轴有不同的缩放以拉伸或改变字体的形状，参考 *Context.set_font_matrix()* 。
- **ctm** – 字体使用的用户空间到设备空间转换的矩阵 *Matrix* 。
- **options** – *FontOptions* 实例，用于获取或解析字体。

根据 *FontFace* 和描述字体尺寸及使用环境的矩阵创建一个 *ScaledFont* 对象。

extents()

返回 (ascent, descent, height, max_x_advance, max_y_advance)，一个浮点数元组

获取 *ScaledFont* 的metric信息。

get_ctm()

pycairo中暂未实现。

get_font_face()

返回 使用 *ScaledFont* 的 *FontFace* 。

1.2 新版功能.

get_font_matrix()
pycairo中暂未实现。

get_font_options()
pycairo中暂未实现。

get_scale_matrix()

返回 缩放矩阵 *Matrix*

缩放矩阵是字体矩阵和与字体关联的当前转换矩阵ctm的产物，即是从字体空间到设备空间的映射。

1.8 新版功能.

glyph_extents()
pycairo中暂未实现。

text_extents(*text*)

参数 **text** (*str*) – text

返回 (x_bearing, y_bearing, width, height, x_advance, y_advance)

返回类型 六个浮点数的元组

获取文本字符串的范围（extent），该范围使用一个包含文本绘制的范围的用户空间的矩形描述。该范围从原点（0,0）开始，因为如果cairo的状态被设置为相同的font face，font matrix，ctm和字体选项 *ScaledFont* ，其将要用 *Context.show_text()* 来绘制。另外，x_advance和y_advance的值表示当前点会被 *Context.show_text()* 推进的值。

as it would be drawn by *Context.show_text()* if the cairo graphics state were set to the same font_face, font_matrix, ctm, and font_options as *ScaledFont* . Additionally, the x_advance and y_advance values indicate the amount by which the current point would be advanced by *Context.show_text()*.

注意空白字符并未直接体现在矩形的尺寸（宽和高）上，这些字符间接的改变了非空白字符的位置。尤其是尾部的空白字符很可能不会影响该矩形的尺寸，尽管他们会印象 x_advance 和 y_advance 的值。

1.2 新版功能.

text_to_glyphs()
pycairo中暂未实现。

class FontOptions()

一个“不透明”的结构，存储了渲染字体时用到的所有选项。

FontOptions 的每个特性可以通过函数 *FontOptions.set_<feature_name>* 或 *FontOptions.get_<feature_name>* 被设置或访问，例如 *FontOptions.set_antialias()* 和 *FontOptions.get_antialias()* 。

未来可能会为 *FontOptions* 添加新的特性，因此应该使用 `FontOptions.copy()` 、 `FontOptions.equal()` 、 `FontOptions.merge()` 和 `FontOptions.hash()` 执行拷贝、检查相等、合并或计算FontOptions hash值的操作。

class cairo.**FontOptions**

返回 一个新分配的 *FontOptions*.

分配一个新的 *FontOptions* 对象，所有的选项被初始化为默认值。

get_antialias()

返回 *FontOptions* 对象的 *ANTIALIAS* 模式。

get_hint_metrics()

返回 *FontOptions* 对象的 *HINT METRICS* 模式。

get_hint_style()

返回 *FontOptions* 对象的 *HINT STYLE* 。

get_subpixel_order()

返回 *FontOptions* 对象的 *SUBPIXEL_ORDER* 。

set_antialias(*antialias*)

参数 **antialias** – *ANTIALIAS* 模式。

这个选项设置了渲染字体时抗锯齿的类型。

set_hint_metrics(*hint_metrics*)

参数 **hint_metrics** – *HINT METRICS* 模式。

这个选项控制 metrics 在设备单元是否被量化为整数。

set_hint_style(*hint_style*)

参数 **hint_style** – *HINT STYLE*

这个选项控制是否使字体轮廓适配像素网格，如果是，时优化字体显示更忠实于原字体或者使用更高的对比度。

set_subpixel_order(*subpixel_order*)

参数 **subpixel_order** – *SUBPIXEL_ORDER*

亚像素顺序描述使用亚像素抗锯齿模式 *cairo.ANTIALIAS_SUBPIXEL* 渲染时显示设备中每个像素中原色颜色的顺序。

4.3.3 FAQ

Pycairo FAQ - Frequently Asked Questions

Q: 我可以子类化 Pycairo 的类吗?

A: cairo 的C库，并不是一个面向对象的库，所有 Python绑定永远都不可能提供真正的面向对象的接口。 如果采用一个很长的模块函数列表的方式来实现绑定，可能最准确的代表了C库的形式。 但是Pycairo（以及绝大多数cairo其他语言的绑定？）选择了实现Context、Surface、Pattern等这几个类的 方式来实现。这样做的优点是按照cairo库中类似的函数按照不同的类分成为了几组，缺点是产生了cairo是一个面向对象的库的错觉，然后开发者可能就会去尝试创建子类来覆盖criro的一些方法。 而事实上没有方法可以被覆盖，只有不能被覆盖的cairo方法。

cario 文档的附录A “给cairo创建一个语言绑定”的 “内存管理” 章节描述了为什么从Surface派生会产生问题， 因此最好避免这样使用。

cairo.Context 可以被子类化。 所有其他的 Pycairo 子类都不能被子类化。

Q: pycairo如何与numpy一起使用?

A: 参见 test/isurface_create_for_data2.py

Q: pycairo如何与pygame一起使用?

A: 参见 **test/pygame-test1.py** test/pygame-test2.py

4.3.4 Pycairo 的 C API

这篇手册描述了给那些想使用pycairo来写扩展模块的C和C++程序员使用的API。

访问Pycairo 的 C API

下面的例子展示了如何导入 pycairo的API:

```
#include "py3cairo.h"

PyMODINIT_FUNC
PyInit_client(void)
{
  PyObject *m;

  m = PyModule_Create(&clientmodule);
  if (m == NULL)
      return NULL;
  if (import_cairo() < 0)
      return NULL;
  /* additional initialization can happen here */
  return m;
}
```

Pycairo 对象

Objects:

```
PycairoContext
PycairoFontFace
PycairoToyFontFace
PycairoFontOptions
PycairoMatrix
PycairoPath
PycairoPattern
PycairoSolidPattern
PycairoSurfacePattern
PycairoGradient
PycairoLinearGradient
PycairoRadialGradient
PycairoScaledFont
PycairoSurface
PycairoImageSurface
PycairoPDFSurface
PycairoPSSurface
PycairoSVGSurface
PycairoWin32Surface
PycairoXCBSurface
PycairoXlibSurface
```

Pycairo Types

Types:

```
PyTypeObject *Context_Type;
PyTypeObject *FontFace_Type;
PyTypeObject *ToyFontFace_Type;
PyTypeObject *FontOptions_Type;
PyTypeObject *Matrix_Type;
PyTypeObject *Path_Type;
PyTypeObject *Pattern_Type;
PyTypeObject *SolidPattern_Type;
PyTypeObject *SurfacePattern_Type;
PyTypeObject *Gradient_Type;
PyTypeObject *LinearGradient_Type;
PyTypeObject *RadialGradient_Type;
PyTypeObject *ScaledFont_Type;
PyTypeObject *Surface_Type;
PyTypeObject *ImageSurface_Type;
PyTypeObject *PDFSurface_Type;
PyTypeObject *PSSurface_Type;
PyTypeObject *SVGSurface_Type;
PyTypeObject *Win32Surface_Type;
PyTypeObject *XCBSurface_Type;
PyTypeObject *XlibSurface_Type;
```

Functions

cairo_t * **PycairoContext_GET**(obj)
从PycairoContext获取C语言的cairo_t 对象

PyObject * **PycairoContext_FromContext**(cairo_t *ctx, PyTypeObject *type, PyObject *base)

PyObject * **PycairoFontFace_FromFontFace**(cairo_font_face_t *font_face)

PyObject * **PycairoFontOptions_FromFontOptions**(cairo_font_options_t *font_options)

PyObject * **PycairoMatrix_FromMatrix**(const cairo_matrix_t *matrix)

PyObject * **PycairoPath_FromPath**(cairo_path_t *path)

PyObject * **PycairoPattern_FromPattern**(cairo_pattern_t *pattern, PyObject *base)

PyObject * **PycairoScaledFont_FromScaledFont**(cairo_scaled_font_t *scaled_font)

PyObject * **PycairoSurface_FromSurface**(cairo_surface_t *surface, PyObject *base)

int **PycairoCheck_Status**(cairo_status_t status)

4.4 D-Bus 使用

4.4.1 D-Bus 系列之入门

注解: 本文整理自 D-Bus 学习系列文章

背景知识

有很多 IPC（interprocess communication ），用于不同的解决方案：CORBA 是用于面向对象编程中复杂的 IPC 的一个强大的解决方案。 DCOP 是一个较轻量级的 IPC 框架，功能较少，但是可以很好地集成到 K 桌面环境中。SOAP 和 XML-RPC 设计用于 Web 服务，因而使用 HTTP 作为其传输协议。 D-BUS 设计用于桌面应用程序和 OS 通信。D-Bus（其中 D 原先是代表桌面"Desktop" 的意思），即：用于桌面操作系统的通信总线。 现在逐渐被引入到嵌入式系统中，不过名字还是保留原先的叫法而已。

典型的桌面都会有多个应用程序在运行，而且，它们经常需要彼此进行通信。DCOP 是一个用于 KDE 的解决方案，但是它依赖于 Qt，所以不能用于其他桌面环境之中。 类似的，Bonobo 是一个用于 GNOME 的解决方案，但是非常笨重，因为它是基于 CORBA 的。它还依赖于 GObject，所以也不能用于 GNOME 之外。 D-BUS 的目标是将 DCOP 和 Bonobo 替换为简单的 IPC，并集成这两种桌面环境。 由于尽可能地减少了 D-BUS 所需的依赖，所以其他可能会使用 D-BUS 的应用程序不用担心引入过多依赖。 相对于其它的 IPC, D-Bus 丢掉了一些不必要的、复杂的东西，也正是因为这个原因，D-Bus 比较快、简单。 D-Bus 不和低层的 IPC 直接竞争，比如 sockets, shared memory or message queues. 这些低层点的 IPC 有它们自己的特点，和 D-Bus 并不冲突。

什么是 D-Bus

D-Bus 是一个为应用程序间通信的消息总线系统，用于进程之间的通信。它是个 3 层架构的 IPC 系统，包括:

a. 函数库 libdbus ，用于两个应用程序互相联系和交互消息。

b. 一个基于 libdbus 构造的消息总线守护进程，可同时与多个应用程序相连，并能把来自一个应用程序的消息路由到 0 或者多个其他程序。

c. 基于特定应用程序框架的封装库或捆绑（wrapper libraries or bindings ）。例如，libdbus-glib 和 libdbus-qt，还有绑定在其他语言，例如 Python 的。大多数开发者都是使用这些封装库的 API，因为它们简化了 D-Bus 编程细节。libdbus 被有意设计成为更高层次绑定的底层后端（low-level backend ）。大部分 libdbus 的 API 仅仅是为了用来实现绑定。

在 D-Bus 中，"bus"是核心的概念，它是一个通道：不同的程序可以通过这个通道做些操作，比如方法调用、发送信号和监听特定的信号。

D-Bus 是低延迟而且低开销的，设计得小而高效，以便最小化传送的往返时间。另外，协议是二进制的，而不是文本的，这样就排除了费时的序列化过程。 从开发者的角度来看，D-BUS 是易于使用的。有线协议容易理解，客户机程序库以直观的方式对其进行包装。D-Bus 的主要目的是提供如下的一些更高层的功能:

a. 结构化的名字空间

b. 独立于架构的数据格式

c. 支持消息中的大部分通用数据元素

d. 带有异常处理的通用远程调用接口

e. 支持广播类型的通信

Bus daemon 是一个特殊的进程：这个进程可以从一个进程传递消息给另外一个进程。 当然了，如果有很多 applications 链接到这个通道上，这个 daemon 进程就会把消息转发给这些链接的所有程序。 在最底层，D-Bus 只支持点对点的通信，一般使用本地套接字（AF_UNIX）在应用和 bus daemon 之间通信。D-Bus 的点对点是经过 bus daemon 抽象过的，由 bus daemon 来完成寻址和发送消息，因此每个应用不必要关心要把消息发给哪个进程。 D-Bus 发送消息通常包含如下步骤（正常情况下）:

a. 创建和发送消息 给后台 bus daemon 进程，这个过程中会有两个上下文的切换

b. 后台 bus daemon 进程会处理该消息，并转发给目标进程，这也会引起上下文的切换

c. 目标程序接收到消息，然后根据消息的种类，做不同的响应：要么给个确认、要么应答、还有就是忽略它。最后一种情况对于"通知"类型的消息而言，前两种都会引起进一步的上 下文切换。

在一台机器上总线守护有多个实例 (instance)。这些总线之间都是相互独立的。

一个持久的系统总线（*system bus*） 它在引导时就会启动。这个总线由操作系统和后台进程使用，安全性非常好，以使得任意的应用程序不能欺骗系统事件。 它是桌面会话和操作系统的通信，这里操作系统一般而言包括内核和系统守护进程。 这种通道的最常用的方面就是发送系统消息，比如：插入一个新的存储设备；有新的网络连接；等等。

还将有很多会话总线（*session buses*） 这些总线当用户登录后启动，属于那个用户私有。它是用户的应用程序用来通信的一个会话总线。 同一个桌面会话中两个桌面应用程序的通信，可使得桌面会话作为整体集成在一起以解决进程生命周期的相关问题。 这在 GNOME 和 KDE 桌面中大量使用。

综上原因，如果你准备在不同的进程之间传递大量的 数据，D-Bus 可能不是最有效的方法，最有效的方法是使用共享内存，但是对共享内存的管理也是相当复杂的。

D-Bus 编程经常涉及的一些概念

地址 连接建立有 server 和 client，对于 bus daemon，应用就是 client，daemon 是 server。 一个 D-Bus 的地址是指 server 用于监听，client 用于连接的地方，例如 unix:path=/tmp/abcedf 标识 server 将在路径 /tmp/abcedf 的 UNIX domain socket 监听。 地址可以是指定的 TCP/IP socket 或者其他在或者将在 D-Bus 协议中定义的传输方式。 如果使用 bus daemon，libdbus 将通过读取环境变量自动获取 session bus damon 的地址，通过检查一个指定的 UNIX domain socket 路径获取 system bus 的地址。如果使用 D-bus，但不是 daemon，需要定义那个应用是 server，那个是 client，并定义一套机制是他们认可 server 的地址，这不是通常的做法。

***Bus Names* 总线名字** 当一个应用连接到 bus daemon，daemon 立即会分配一个名字给这个连接，称为 unique connection name， 这个唯一标识的名字以冒号：开头，例如：34-907，这个名字在 daemon 的整个生命周期是唯一的。 但是这种名字总是临时分配，无法确定的，也难以记忆，因此应用可以要求有另外一个名字 well-known name 来对应这个唯一标识，就像我们使用域名来对应 IP 地址一样。 例如可以使用 com.mycompany 来映射：34-907。应用程序可能会要求拥有额外的周知名字（well-known name）。 例如，你可以写一个规范来定义一个名字叫做 com.mycompany.TextEditor。 应用程序就可以发送消息到这个总线名字，对象，和接口以执行方法调用。 当一个应用结束或者崩溃是，OS kernel 会关闭它的总线连接。总线发送 notification 消息告诉其他应用，这个应用的名字已经失去他的 owner。 当检测到这类 notification 时，应用可以知道其他应用的生命周期。这种方式也可用于只有一个实例的应用，即不开启同样的两个应用的情况。

原生对象和对象路径 所有使用 D-BUS 的应用程序都包含一些对象，当经由一个 D-BUS 连接受到一条消息时，该消息是被发往一个对象而不是整个应用程序。 在开发中程序框架定义着这样的对象，例如 JAVA，GObject，QObject 等等，在 D-Bus 中成为 native object。 对于底层的 D-Bus 协议，即 libdbus API，并不理会这些 native object，它们使用的是一个叫做 object path 的概念。 通过 object path，高层编程可以为对象实例进行命名，并允许远程应用引用它们。 这些名字看起来像是文件系统路径，易读的路径名是受鼓励的做法，但也允许使用诸如"/com/mycompany/c5yo817y0c1y1c5b"等，只要它可以为你的应用程序所用。 Namespacing 的对象路径以开发者所有的域名开始（如 /org/kde）以避免系统不同代码模块互相干扰。 简单地说：一个应用创建对象实例进行 D-Bus 的通信，这些对象实例都有一个名字，命名方式类似于路径， 例如 /com/mycompany，这个名字在全局（session 或者 system）是唯一的，用于消息的路由。

***Proxies* 代理** 代理对象用来表示其他远程的 remote object。 当我们触发了 proxy 对象的 method 时，将会在 D-Bus 上发送一个 method_call 的消息，并等待答复，根据答复返回。使用非常方便，就像调用一个本地的对象。

接口 *Interface* 每一个对象支持一个或者多个接口，接口是一组方法和信号，接口定义一个对象实体的类型。 D-Bus 对接口的命名方式，类似 org.freedesktop.Introspectable。开发人员通常将使用编程语言类的的名字作为接口名字。

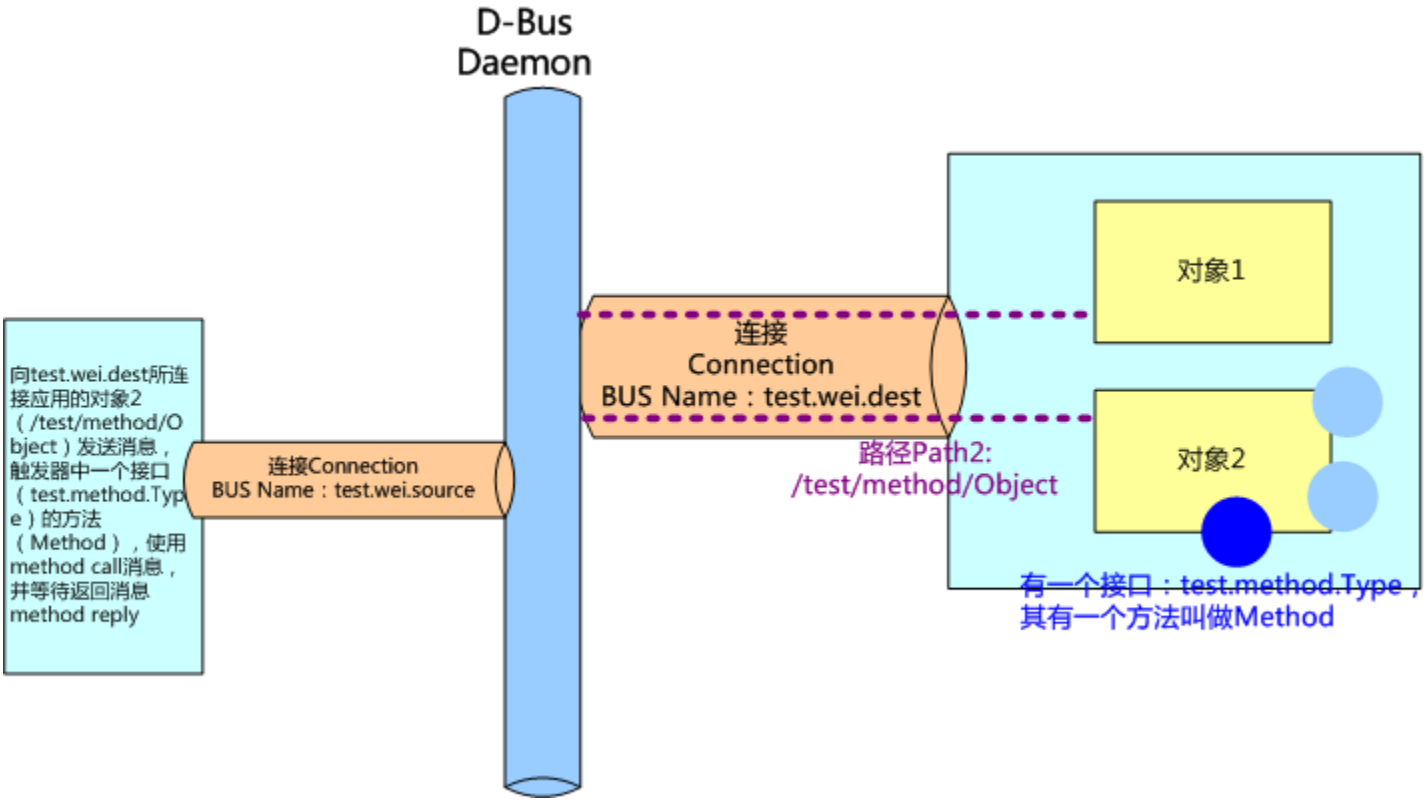
方法和信号 *Methods and Signals* 每一个对象有两类成员：方法和信号。方法就是 JAVA 中同样概念，方法是一段函数代码，带有输入和输出。信号是广播给所有兴趣的其他实体，信号可以带有数据 payload。 在 D-BUS 中有四种类型的消息：方法调用（method calls）、方法返回（method returns）、信号（signals）和错误（errors）。 要执行 D-BUS 对象的方法，您需要向对象发送一个方法调用消息。 它将完成一些处理（就是执行了对象中的 Method，Method 是可以带有输入参数的。）并返回，返回消息或者错误消息。 信号的不同之处在于它们不返回任何内容：既没有“信号返回”消息，也没有任何类型的错误消息。

通过上面的描述，我们可以获得下面的视图：：

Address –> [Bus Name] –> Path –> Interface –> Method

bus name 不是必要的，它只在 daemon 的情况下用于路由，点对点的直接连接是不需要的。 简单地说：Address 是 D-Bus 中 server 用来监听 client 的地址，当一个 client 连接上 D-Bus，通常是 Daemo 的方式，这个 client 就有了一个 Bus Name。 其他应用可以根据消息中所带的 Bus Name，来判断和哪个应用相关。 消息在总线中传递的时候，传递到应用中，再根据 object path，送至应用中具体的对象实例中，也就是是应用中根据 Interface 创建的对象。 这些 Interface 有 method 和 singal 两种，用来发送、接收、响应消息。

这些概念对初学者可能会有些混淆，但是通过后面学习一个小程序，就很清楚，这是后面一个例子的示意图，回过头来看看之前写的这篇文章，这个示意图或许会更清楚。



D-Bus 消息类型

消息通过 D-Bus 在进程间传递。有四类消息： a. Method call 消息：将触发对象的一个 method #. Method return 消息：触发的方法返回的结果 #. Error 消息：触发的方法返回一个异常 #. Signal 消息：通知，可以看作为事件消息。

一个消息有消息头 header，里面有 field，有一个消息体 body，里面有参数 arguments。消息头包含消息体的路由信息，消息体就是净荷 payload。头字段可能包括发送者的 bus 名，目的地的 bus 名，方法或者

signal 名等等，其中一个头字段是用于描述 body 中的参数的类型，例如“i”标识 32 位整数，”ii”表示净荷为 2 个 32 为整数。

发送 Method call 消息的场景

一个 method call 消息从进程 A 到进程 B，B 将应答一个 method return 消息或者 error 消息。 在每个 call 消息带有一个序列号，应答消息也包含同样的号码，使之可以对应起来。 他们的处理过程如下：

a. 如果提供 proxy，通过触发本地一个对象的方法从而触发另一个进程的远端对象的方法。应用调用 proxy 的一个方法，proxy 构造一个 method call 消息发送到远端进程。

b. 对于底层的 API，不使用 proxy，应用需要自己构造 method call 消息。

c. 一个 method call 消息包含：远端进程的 bus name，方法名字，方法的参数，远端进程中 object path，可选的接口名字。

d. method call 消息发送到 bus daemon

e. bus daemon 查看目的地的 bus name。如果一个进程对应这个名字，bus daemon 将 method call 消息发送到该进程中。如果没有发现匹配，bus daemon 创建一个 error 消息作为应答返回。

f. 进程接收后将 method call 消息分拆。对于简单的底层 API 情况，将立即执行方法，并发送一个 method reply 消息给 bus daemon。对于高层的 API，将检查对象 path，interface 和 method，触发一个 native object 的方法，并将返回值封装在一个 method reply 消息中。

g. bus daemon 收到 method reply 消息，将其转发到原来的进程中

h. 进程查看 method reply 消息，获取返回值。这个响应也可以标识一个 error 的残生。当使用高级的捆绑，method reply 消息将转换为 proxy 方法的返回值或者一个 exception。

Bus daemon 保证 message 的顺序，不会乱序。例如我们发送两个 method call 消息到同一个接受方，他们将按顺序接受。接收方并不要求一定按顺序回复。消息有一个序列号了匹配收发消息。

发送 Signal 的场景

signal 是个广播的消息，不需要响应，接收方向 daemon 注册匹配的条件，包括发送方和信号名，bus 守护只将信号发送给希望接受的进程。

处理流程如下：

a. 一个 signal 消息发送到 bus daemon。

b. signal 消息包含发布该信号的 interface 名字，signal 的名字，进程的 bus 名字，以及参数。

c. 任何进程都可以注册的匹配条件（match rules) 表明它所感兴趣的 signal。总线有个注册 match rules 列表。

d. bus daemon 检查那些进程对该信号有兴趣，将信号消息发送到这些进程中。

e. 收到信号的进程决定如何处理。如果使用高层的捆绑，一个 porxy 对象将会释放一个 native 的信号。如果使用底层的 API，进程需要检查信号的发送方和信号的名字决定如何进行处理。

Introspection D-Bus 对象可能支持一个接口 org.freedesktop.DBus.Introspectable。该接口有一个方法 Introspect，不带参数，将返回一个 XML string。这个 XML 字符串描述接口，方法，信号。

简单的例子

D-Bus 类型和 GType 的映射

在 D-Bus 编程中，基础类型和 GType 的映射表格如下。在后面的程序小例子中我们会看到具体如何对应。

D-Bus basic type	GType	Free function	Notes
BYTE	G_TYPE_BOOLEAN		
INT16	G_TYPE_INT		Will be changed to a G_TYPE_INT16 once GLib has it
UINT16	G_TYPE_UINT		Will be changed to a G_TYPE_UINT16 once GLib has it
INT32	G_TYPE_INT		Will be changed to a G_TYPE_INT32 once GLib has it
UINT32	G_TYPE_UINT		Will be changed to a G_TYPE_UINT32 once GLib has it
INT64			G_TYPE_GINT64
UINT64	G_TYPE_GUINT64		
DOUBLE	G_TYPE_DOUBLE		
STRING	G_TYPE_STRING	g_free	
OBJECT_PATH	DBUS_TYPE_G_PROXY	g_object_unref	The returned proxy does not have an interface set; use dbus_g_p

在 D-Bus 编程中，container 类型和 GType 的映射表格如下：

D-Bus type signature	Description	GType	C typedef	Free function	Notes
as	Array of strings	G_TYPE_STRV	char **	g_strfreev	
v	Generic value container	G_TYPE_VALUE	GValue *	g_value_unset	The calling conven callers have allocat
a{ss}	Dictionary mapping strings to strings	DBUS_TYPE_G_STRING_STRING_HASHTABLE	GHashTable *	g_hash_table_destroy	

在下面的例子中，使用了 dbus-1 dbus-glib-1 glib-2.0。Makefile 的如下：

```
CC=gcc

CFLAGS += `pkg-config --cflags dbus-glib-1 glib-2.0`
LIBS += `pkg-config --libs dbus-glib-1 glib-2.0`

all: sync_sample async_sample signal_send signal_recv method_service method_call

async_sample: async_sample.c
	$(CC) -Wall -g $(CFLAGS) $< $(LIBS) -o $@

sync_sample: sync_sample.c
	$(CC) -Wall -g $(CFLAGS) $< $(LIBS) -o $@

```

(下页继续)

(续上页)

```
signal_send: signal_send.c
	$(CC) -Wall -g $(CFLAGS) $< $(LIBS) -o $@

signal_recv: signal_recv.c
	$(CC) -Wall -g $(CFLAGS) $< $(LIBS) -o $@

method_service: method_service.c
	$(CC) -Wall -g $(CFLAGS) $< $(LIBS) -o $@

method_call:method_call.c
	$(CC) -Wall -g $(CFLAGS) $< $(LIBS) -o $@

clean:
	rm -f sync_sample async_sample signal_send signal_recv method_service method_
↪call
```

同步的例子

同步即程序发出 method call 消息，等待 method_return 消息。下面是一个小例子，如果我们用 dbus-send 命令，可以使用：

```
dbus-send --session --print-reply --type=method_call --dest=org.freedesktop.
↪Notifications / org.freedesktop.DBus.Introspectable.Introspect
```

```
#include <stdio.h>
#include <stdlib.h>
#include <dbus/dbus-glib.h>

int main(int argc, char **argv)
{
    GError *error;
    DBusGConnection *conn;
    DBusGProxy *proxy;
    char *str;

    /* gtype init */
    error = NULL;
    /* connect session bus with dbus_g_get, or DBUS_BUS_SYSTEM connect system bus */
    conn = dbus_g_bus_get(DBUS_BUS_SESSION, &error);
    if (conn == NULL) {
        g_printerr("Failed to open connection to bus:%s\n", error->message);
        g_error_free(error);
        exit(1);
    }

    /* create a proxy object for  */
    proxy = dbus_g_proxy_new_for_name(conn,
            "org.freedesktop.Notifications", /* service */
            "/", /* path */
            "org.freedesktop.DBus.Introspectable"/* interface */
            );
    error = NULL;
    if (!dbus_g_proxy_call(proxy, "Introspect", &error, G_TYPE_INVALID, G_TYPE_STRING,
↪ &str, G_TYPE_INVALID)) {
```

(下页继续)

(续上页)

```
        if (error->domain == DBUS_GERROR && error->code == DBUS_GERROR_REMOTE_
↪EXCEPTION) {
            g_printerr("Caught remote method exception:%s-%s\n", dbus_g_error_get_
↪name(error), error->message);
        } else {
            g_printerr("Error:%s\n", error->message);
        }
        g_error_free(error);
        exit(1);
    }

    g_print("Message Method return from bus:\n%s\n", str);
    g_free(str);
    g_object_unref(proxy);
    exit(1);

    return 0;
}
```

异步的例子

异步中，程序将不等返回消息，继续执行，等有返回消息的时候，触发一个回调函数 。下面是同样的操作，但是用异步的方式来实现:

```
#include <stdio.h>
#include <stdlib.h>
#include <dbus/dbus-glib.h>

static void callback_func(DBusGProxy *proxy, DBusGProxyCall *call_id, void *user_data)
{
    GError *err = NULL;
    gchar *str = NULL;
    GMainLoop *main_loop = user_data;

    /* 结束一条消息的收发，处理收到的消息 */
    dbus_g_proxy_end_call(proxy, call_id, &err, G_TYPE_STRING, &str, G_TYPE_INVALID);
    if (err != NULL) {
        g_print("Error in method call:%s\n", err->message);
        g_error_free(err);
    } else {
        g_print("Success, message:\n%s\n", str);
    }

    g_main_loop_quit(main_loop);
}

int main(int argc, char **argv)
{
    GError *error;
    DBusGConnection *conn;
    DBusGProxy *proxy;
    GMainLoop *main_loop;

    error = NULL;
```

(下页继续)

(续上页)

```
    main_loop = g_main_loop_new(NULL, FALSE);
    /* connect session bus with dbus_g_get, or DBUS_BUS_SYSTEM connect system bus */
    conn = dbus_g_bus_get(DBUS_BUS_SESSION, &error);
    if (conn == NULL) {
        g_printerr("Failed to open connection to bus:%s\n", error->message);
        g_error_free(error);
        exit(1);
    }

    /* create a proxy object for  */
    proxy = dbus_g_proxy_new_for_name(conn,
            "org.freedesktop.Notifications", /* service */
            "/", /* path */
            "org.freedesktop.DBus.Introspectable"/* interface */
            );
    error = NULL;
    dbus_g_proxy_begin_call(proxy, "Introspect", callback_func, main_loop, NULL, G_
↪TYPE_INVALID);
    g_main_loop_run(main_loop);

    return 0;
}
```

Signal 的收发的例子

现在从底层，即 libdbus 学习如何发送 signal，以及如何监听 signal。signal 在 D-Bus 的 Daemon 中广播，为了提高效率，只发送给向 daemon 注册要求该 singal 的对象。 程序，第一步需要将应用和 D-Bus 后台建立连接，也就是和 System D-Bus daemon 或者 Session D-Bus daemon 建立连接。 一旦建立，daemon 会给这条连接分配一个名字，这个名字在 system 或者 session 的生命周期是唯一的， 即 unique connection name，为了方便记忆，可以为这条连接分配一个便于记忆的 well-known name。 对于信号方式，分配这个名字不是必须的（在 method_call 中是需要的，我们在下一次学习中谈到），因为在信号的监听中秩序给出 Interface 的名字和信号名称。 在下面的例子中，可以将相关的代码屏蔽掉，不影响运行，但是通常我们都这样处理，尤其在复杂的程序中。

在我们的例子中，定义这个 BUS name 为 test.singal.source。当然一个好的名字，为了避免于其他应用重复，应当使用 com.mycompany.myfunction 之类的名字。 而 interface 的名字，一般前面和 connection 的 BUS name 一致。

发送方的程序

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dbus/dbus-glib.h>
#include <dbus/dbus.h>
#include <unistd.h>

int send_a_signal( char * sigvalue)
{
    DBusError err;
    DBusConnection * connection;
    DBusMessage * msg;
    DBusMessageIter arg;
```

(下页继续)

(续上页)

```
    dbus_uint32_t  serial = 0;
    int ret;

    //步骤1:建立与D-Bus后台的连接
    /* initialise the erroes */
    dbus_error_init(&err);
    /* Connect to Bus*/
    connection = dbus_bus_get(DBUS_BUS_SESSION , &err );
    if(dbus_error_is_set(&err)) {
        fprintf(stderr,"Connection Err : %s\n",err.message);
        dbus_error_free(&err);
    }
    if(connection == NULL) {
        return -1;
    }

    //步骤2:给连接名分配一个well-known的名字作为Bus name，这个步骤不是必须的，可以用if 0来注释着一段代码，我们可以用这个名字来检查，是否已经开启了这个应用的另外的进程。
#if 1
    ret = dbus_bus_request_name(connection,"test.singal.source",DBUS_NAME_FLAG_REPLACE_EXISTING,&err);
    if(dbus_error_is_set(&err)) {
        fprintf(stderr,"Name Err : %s\n",err.message);
        dbus_error_free(&err);
    }
    if(ret != DBUS_REQUEST_NAME_REPLY_PRIMARY_OWNER)
        return -1;
#endif

    //步骤3:发送一个信号
    //根据图，我们给出这个信号的路径（即可以指向对象），接口，以及信号名，创建一个Message
    if((msg = dbus_message_new_signal ("/test/signal/Object","test.signal.Type","Test")) == NULL) {
        fprintf(stderr,"Message NULL\n");
        return -1;
    }
    //给这个信号（messge）具体的内容
    dbus_message_iter_init_append (msg,&arg);
    if(!dbus_message_iter_append_basic (&arg,DBUS_TYPE_STRING,&sigvalue)) {
        fprintf(stderr,"Out Of Memory!\n");
        return -1;
    }

    //步骤4: 将信号从连接中发送
    if( !dbus_connection_send(connection,msg,&serial)) {
        fprintf(stderr,"Out of Memory!\n");
        return -1;
    }
    dbus_connection_flush (connection);
    printf("Signal Send\n");

    //步骤5: 释放相关的分配的内存。
    dbus_message_unref(msg);
    return 0;
}

```

(下页继续)

(续上页)

```
67
68 int main( int argc , char ** argv)
69 {
70     send_a_signal("Hello,world!");
71     return 0;
72 }
```

接收该信号的的例子

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <dbus/dbus-glib.h>
5  #include <dbus/dbus.h>
6  #include <unistd.h>
7
8  void listen_signal()
9  {
10     DBusMessage * msg;
11     DBusMessageIter arg;
12     DBusConnection * connection;
13     DBusError err;
14     int ret;
15     char * sigvalue;
16
17     //步骤1:建立与D-Bus后台的连接
18     dbus_error_init(&err);
19     connection = dbus_bus_get(DBUS_BUS_SESSION, &err);
20     if(dbus_error_is_set(&err)) {
21         fprintf(stderr,"Connection Error %s\n",err.message);
22         dbus_error_free(&err);
23     }
24     if(connection == NULL) {
25         return;
26     }
27
28     //步骤2:给连接名分配一个可记忆名字test.singal.dest作为Bus name，这个步骤不是必须的,但推荐这样
   处理
29     ret = dbus_bus_request_name(connection,"test.singal.dest",DBUS_NAME_FLAG_REPLACE_
   ↪EXISTING,&err);
30     if(dbus_error_is_set(&err)) {
31         fprintf(stderr,"Name Error %s\n",err.message);
32         dbus_error_free(&err);
33     }
34     if(ret != DBUS_REQUEST_NAME_REPLY_PRIMARY_OWNER) {
35         return;
36     }
37
38     //步骤3:通知D-Bus daemon，希望监听来行接口test.signal.Type的信号
39     dbus_bus_add_match(connection,"type='signal',interface='test.signal.Type'",&err);
40     //实际需要发送东西给daemon来通知希望监听的内容，所以需要flush
41     dbus_connection_flush(connection);
42     if(dbus_error_is_set(&err)) {
43         fprintf(stderr,"Match Error %s\n",err.message);
```

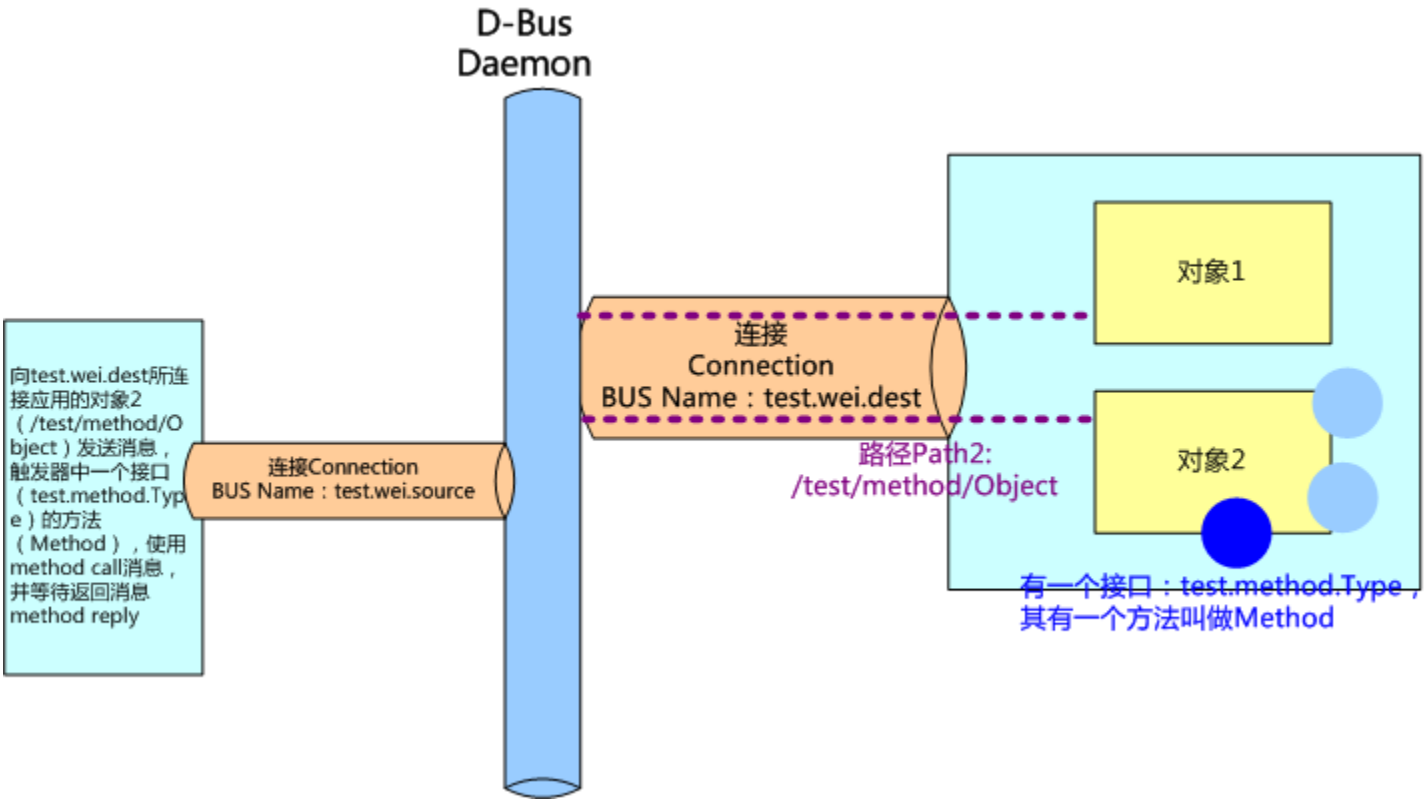
(下页继续)

(续上页)

```
        dbus_error_free(&err);
    }

    //步骤4:在循环中监听，每隔开1秒，就去试图自己的连接中获取这个信号。这里给出的是从连接中获取任何消息的方式，所以获取后去检查一下这个消息是否我们期望的信号，并获取内容。我们也可以通过这个方式来获取method call消息。
    while(1) {
        dbus_connection_read_write(connection,0);
        msg = dbus_connection_pop_message (connection);
        if(msg == NULL){
            sleep(1);
            continue;
        }

        if(dbus_message_is_signal(msg,"test.signal.Type","Test") ){
            if(!dbus_message_iter_init(msg,&arg)) {
                fprintf(stderr,"Message Has no Param");
            } else if(dbus_message_iter_get_arg_type(&arg) != DBUS_TYPE_STRING) {
                g_printerr("Param is not string");
            } else {
                dbus_message_iter_get_basic(&arg,&sigvalue);
                printf("Got Singal with value : %s\n",sigvalue);
            }
        }
        dbus_message_unref(msg);
    }//End of while

    return ;
}

int main( int argc , char ** argv){
    listen_signal();
    return 0;
}
```

Method 的收发小例子

现在我们从底层，即 libdbus 学习如何发送 Method 以及如何等待应答，在之前的代码中，我们给出了同步方式的代码，这是更为高层的处理方式，建议使用。监听 method 和监听 signal 的方式非常相似。在给出例子之前，我希望和上次学习一样给出一个示意图，更好地了解 D-Bus 的各个概念。



在下面的例子中，我们将学习如何在消息中加入多个参数的情况。

方法监听的例子

Method 的监听和 signal 的监听的处理时一样，但是信号是不需要答复，而 Method 需要。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dbus/dbus-glib.h>
#include <dbus/dbus.h>
#include <unistd.h>

/*读取消息的参数，并且返回两个参数，一个是bool值stat，一个是整数level*/
void reply_to_method_call(DBusMessage * msg, DBusConnection * conn)
{
    DBusMessage * reply;
    DBusMessageIter arg;
    char * param = NULL;
    dbus_bool_t stat = TRUE;
    dbus_uint32_t level = 2010;
    dbus_uint32_t serial = 0;

    //从msg中读取参数，这个在上一次学习中学过
    if(!dbus_message_iter_init(msg,&arg)) {
        printf("Message has no args/n");
    } else if(dbus_message_iter_get_arg_type(&arg) != DBUS_TYPE_STRING) {
        printf("Arg is not string!/n");
```

(下页继续)

(续上页)

```
    } else {
        dbus_message_iter_get_basic(&arg,& param);
    }
    if(param == NULL) return;

    //创建返回消息reply
    reply = dbus_message_new_method_return(msg);
    //在返回消息中填入两个参数，和信号加入参数的方式是一样的。这次我们将加入两个参数。
    dbus_message_iter_init_append(reply,&arg);
    if(!dbus_message_iter_append_basic(&arg,DBUS_TYPE_BOOLEAN,&stat)) {
        printf("Out of Memory!/n");
        exit(1);
    }
    if(!dbus_message_iter_append_basic(&arg,DBUS_TYPE_UINT32,&level)) {
        printf("Out of Memory!/n");
        exit(1);
    }
    //发送返回消息
    if( !dbus_connection_send(conn, reply, &serial)){
        printf("Out of Memory/n");
        exit(1);
    }
    dbus_connection_flush (conn);
    dbus_message_unref (reply);
}

/* 监听D-Bus消息，我们在上次的例子中进行修改 */
void listen_dbus()
{
    DBusMessage * msg;
    DBusMessageIter arg;
    DBusConnection * connection;
    DBusError err;
    int ret;
    char * sigvalue;

    dbus_error_init(&err);
    //创建于session D-Bus的连接
    connection = dbus_bus_get(DBUS_BUS_SESSION, &err);
    if(dbus_error_is_set(&err)){
        fprintf(stderr,"Connection Error %s/n",err.message);
        dbus_error_free(&err);
    }
    if(connection == NULL) {
        return;
    }
    //设置一个BUS name: test.wei.dest
    ret = dbus_bus_request_name(connection,"test.wei.dest",DBUS_NAME_FLAG_REPLACE_
↪EXISTING,&err);
    if(dbus_error_is_set(&err)) {
        fprintf(stderr,"Name Error %s/n",err.message);
        dbus_error_free(&err);
    }
    if(ret != DBUS_REQUEST_NAME_REPLY_PRIMARY_OWNER) {
        return;
    }
```

(下页继续)

(续上页)

```

    //要求监听某个singal: 来自接口test.signal.Type的信号
    dbus_bus_add_match(connection,"type='signal',interface='test.signal.Type'",&err);
    dbus_connection_flush(connection);
    if(dbus_error_is_set(&err)){
        fprintf(stderr,"Match Error %s/n",err.message);
        dbus_error_free(&err);
    }

    while(TRUE){
        dbus_connection_read_write (connection,0);
        msg = dbus_connection_pop_message (connection);

        if(msg == NULL){
            sleep(1);
            continue;
        }

        if(dbus_message_is_signal(msg,"test.signal.Type","Test")){
            if(!dbus_message_iter_init(msg,&arg)) {
                fprintf(stderr,"Message Has no Param");
            } else if(dbus_message_iter_get_arg_type(&arg) != DBUS_TYPE_STRING) {
                g_printerr("Param is not string");
            } else {
                dbus_message_iter_get_basic(&arg,&sigvalue);
                printf("Got Singal with value : %s\n",sigvalue);
            }
        }else if(dbus_message_is_method_call(msg,"test.method.Type","Method")){
            //我们这里面先比较了接口名字和方法名字，实际上应当现比较路径
            if(strcmp(dbus_message_get_path (msg),"/test/method/Object") == 0) {
                reply_to_method_call(msg, connection);
            }
        }
        dbus_message_unref(msg);
    }
}

int main( int argc , char ** argv)
{
    listen_dbus();
    return 0;
}
```

方法调用的例子

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dbus/dbus-glib.h>
#include <dbus/dbus.h>
#include <unistd.h>

//建立与session D-Bus daemo的连接，并设定连接的名字，相关的代码已经多次使用过了
DBusConnection *  connect_dbus()
```

(下页继续)

(续上页)

```
{
    DBusError err;
    DBusConnection * connection;
    int ret;

    //Step 1: connecting session bus
    /* initialise the erroes */
    dbus_error_init(&err);
    /* Connect to Bus*/
    connection = dbus_bus_get(DBUS_BUS_SESSION, &err);
    if(dbus_error_is_set(&err)){
        fprintf(stderr,"Connection Err : %s\n",err.message);
        dbus_error_free(&err);
    }
    if(connection == NULL) {
        return NULL;
    }

    //step 2: 设置BUS name，也即连接的名字。
    ret = dbus_bus_request_name(connection,"test.wei.source",DBUS_NAME_FLAG_REPLACE_
↪EXISTING,&err);
    if(dbus_error_is_set(&err)) {
        fprintf(stderr,"Name Err : %s\n",err.message);
        dbus_error_free(&err);
    }

    if(ret != DBUS_REQUEST_NAME_REPLY_PRIMARY_OWNER) {
        return NULL;
    }

    return connection;
}

void send_a_method_call(DBusConnection * connection,char * param)
{
    DBusError err;
    DBusMessage * msg;
    DBusMessageIter    arg;
    DBusPendingCall * pending;
    dbus_bool_t * stat;
    dbus_uint32_t * level;

    dbus_error_init(&err);

    //针对目的地地址，请参考图，创建一个method call消息。 Constructs a new message to invoke␣
↪a method on a remote object.
    msg = dbus_message_new_method_call ("test.wei.dest","/test/method/Object","test.
↪method.Type","Method");
    if(msg == NULL) {
        g_printerr("Message NULL");
        return;
    }

    //为消息添加参数。Append arguments
    dbus_message_iter_init_append(msg, &arg);
    if(!dbus_message_iter_append_basic (&arg, DBUS_TYPE_STRING,&param)) {
```

(下页继续)

(续上页)

```
        g_printerr("Out of Memory!");
        exit(1);
    }

    //发送消息并获得reply的handle 。Queues a message to send, as with dbus_connection_
↪send() , but also returns a DBusPendingCall used to receive a reply to the message.
    if(!dbus_connection_send_with_reply(connection, msg,&pending, -1)){
        g_printerr("Out of Memory!");
        exit(1);
    }

    if(pending == NULL) {
        g_printerr("Pending Call NULL: connection is disconnected ");
        dbus_message_unref(msg);
        return;
    }

    dbus_connection_flush(connection);
    dbus_message_unref(msg);

    //waiting a reply，在发送的时候，已经获取了method reply的handle，类型为DBusPendingCall。
    // block until we recieve a reply,  Block until the pending call is completed.
    dbus_pending_call_block (pending);
    // get the reply message，Gets the reply, or returns NULL if none has been␣
↪received yet.
    msg = dbus_pending_call_steal_reply (pending);
    if (msg == NULL) {
        fprintf(stderr, "Reply Null\n");
         exit(1);
    }
     // free the pending message handle
     dbus_pending_call_unref(pending);
    // read the parameters
    if (!dbus_message_iter_init(msg, &arg)) {
        fprintf(stderr, "Message has no arguments!\n");
    } else if ( dbus_message_iter_get_arg_type (&arg) != DBUS_TYPE_BOOLEAN) {
        fprintf(stderr, "Argument is not boolean!\n");
    } else {
        dbus_message_iter_get_basic (&arg, &stat);
    }

    if (!dbus_message_iter_next(&arg)) {
        fprintf(stderr, "Message has too few arguments!\n");
    } else if ( dbus_message_iter_get_arg_type (&arg) != DBUS_TYPE_UINT32 ) {
        fprintf(stderr, "Argument is not int!\n");
    } else {
        dbus_message_iter_get_basic (&arg, &level);
    }

    printf("Got Reply: %d, %d\n", (int)stat, (int)level);
    dbus_message_unref(msg);
}

int main( int argc , char ** argv)
{
    DBusConnection * connection;
```

(下页继续)

(续上页)

```
    connection = connect_dbus();
    if(connection == NULL) {
        return -1;
    }

    send_a_method_call(connection,"Hello, D-Bus");
    return 0;
}
```

使用 xml 定义 D-Bus 接口

在 D-Bus 中，可以将 D-Bus 接口定义用 XML 格式表述处理，并利用工具，自动生成头文件，给出工整的调用方式。

下面是一个 XML 的例子。

客户端

```
<?xml version="1.0" encoding="UTF-8" ?>

<node name="/com/wei/MyObject">
    <interface name="com.wei.MyObject.Sample">
        <method name="Test">
            <annotation name="org.freedesktop.DBus.GLib.CSymbol" value="wei_response"/>
            <arg type="u" name="x" direction="in" />
            <arg type="d" name="d_ret" direction="out" />
        </method>
    </interface>
</node>

```

```
dbus-binding-tool --mode=glib-client --prefix=com_wei wei.xml > wei_client.h
```

服务端

```
<?xml version="1.0" encoding="UTF-8" ?>
<node name="/com/wei/MyObject">
    <interface name="com.wei.MyObject.Sample">
        <method name="Test">
            <arg name="x" type="u" direction="in" />
            <arg name="d_ret" type="d" direction="out" />
        </method>
    </interface>
</node>
```

```
dbus-binding-tool --mode=glib-server --prefix=com_wei wei.xml > wei_server.h
```

其他参考资料

原始链接

1. Nokia D-Bus page
2. KDE Base D-Bus example
3. telepathy D-Bus page

4.4.2 D-Bus系列之权限配置文件

D-Bus配置文件 D-Bus消息守护进程的配置文件配置了总线的类型，资源限制，安全参数等。 配置文件的格式并不是标准的一部分，也不保证向后兼容。。。

标准的系统总线和每会话总线在"/etc/dbus-1/system.conf" and "/etc/dbus-1/session.conf"，这两个文件 会包含system-local.conf或session-local.conf，因此你自己的配置应该放在单独的local文件中。

该配置文件的格式就是一个xml文档，且必须有如下类型声明：

<!DOCTYPE busconfig PUBLIC "-//freedesktop//DTD D-Bus Bus Configuration 1.0//EN" "http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">

<busconfig> 根元素

<type> 一般为system或session。

<include> 在当前位置包含该文件，如果是相对目录，则应为相对于当前配置文件所在目录，<include>有一个选项"ignore_missing=(yes|no)"，默认为no

<includedir> 在当前位置包含该目录所有的配置文件，目录中文件的包含顺序不定，且只有以".conf"结尾的文件才会被包含。

<user> daemon运行的用户，可以为用户名或uid，如果守护进程无法切换到该用户就会自动退出，如果有多个<user>配置项，会采用最后一个。

<fork> 进程成为一个真正的守护进程。

<keep_umask> 若存在，守护进程在fork时会保留原来的umask。

<listen> 总线监听的地址，地址为包含传输地址加参数/选项的标准D-Bus格式，例如:

```
<listen>unix:path=/tmp/foo</listen>
<listen>tcp:host=localhost,port=1234</listen>
```

<auth> 指定授权机制。如果不存在，所有已知的机制都被允许。如果有多项配置，则所有列出的机制都被允许。

<servicedir> 添加扫描.service文件的目录，Service用于告诉总线如何自动启动一个程序，主要用于每用户的session bus。

<standard_session_servicedirs/> 等效于设定一系列的<servicedir/>元素， "XDG Base Directory Specification"

<standard_system_servicedirs/> 设定标准的系统级service搜索目录，默认为/usr/share/dbus-1/system-services， 只用于/etc/dbus-1/system.conf.定义的系统级总线，放在其他配置文件中无效。

<servicehelper/> 设定setuid helper，使用设置的用户启动系统服务的守护进程，一般来说应该是dbus-daemon-launch-helper。 该选项仅用于系统总线。

<limit> 资源限制一般用于系统总线。 设置资源限制，例如:

```
<limit name="max_message_size">64</limit>
<limit name="max_completed_connections">512</limit>
```

可用的限制名有:

```
"max_incoming_bytes"          : total size in bytes of messages
                                incoming from a single connection
"max_incoming_unix_fds"       : total number of unix fds of messages
                                incoming from a single connection
"max_outgoing_bytes"          : total size in bytes of messages
                                queued up for a single connection
"max_outgoing_unix_fds"       : total number of unix fds of messages
                                queued up for a single connection
"max_message_size"            : max size of a single message in
                                bytes
"max_message_unix_fds"        : max unix fds of a single message
"service_start_timeout"       : milliseconds (thousandths) until
                                a started service has to connect
"auth_timeout"                : milliseconds (thousandths) a
                                connection is given to
                                authenticate
"max_completed_connections"   : max number of authenticated connections
"max_incomplete_connections" : max number of unauthenticated
                                connections
"max_connections_per_user"    : max number of completed connections from
                                the same user
"max_pending_service_starts" : max number of service launches in
                                progress at the same time
"max_names_per_connection"    : max number of names a single
                                connection can own
"max_match_rules_per_connection": max number of match rules for a single
                                   connection
"max_replies_per_connection" : max number of pending method
                                replies per connection
                                (number of calls-in-progress)
"reply_timeout"               : milliseconds (thousandths)
                                until a method call times out
```

<policy> 定义用于一组特定连接的安全策略，策略由<allow>和<deny>元素组成。 策略一般用于系统总线，模拟防火墙的功能来只允许期望的连接。 当前系统总线的默认策略会阻止发送方法调用和获取总线名字，其他的如消息回复、信号等是默认允许的。

通常来说，最好是保证系统服务尽可能的小，目标程序在自己的进程中运行并且提供一个总线名字来提供服务。 <allow>规则使得程序可以设置总线名字，<send_destination>允许一些或所有的uid访问我们的服务。

<policy>元素可以有下面四个属性中的一个： context="(default|mandatory)" at_console="(true|false)" user="username or userid" group="group name or gid"

策略以下面的规则应用到连接： - 所有 context="default" 的策略被应用 - 所有 group="connection's user's group" 的策略以不定的顺序被应用 - 所有 user="connection's auth user" 的策略以不定顺序被应用 - 所有 at_console="true" 的策略被应用 - 所有 at_console="false" 的策略被应用 - 所有 context="mandatory" 的策略被应用 后应用的策略会覆盖前面的策略。

<allow>和<deny>出现在<policy>元素下面，<deny>禁止一些动作，而<allow>则创建上面<deny>元素的一些例外。 这两个元素可用的属性包括:

send_interface="interface_name" send_member="method_or_signal_name" send_error="error_name" send_destination="name" send_type="method_call" | "method_return" |

“signal” | “error” send_path=”/path/name”

receive_interface=”interface_name” receive_member=”method_or_signal_name” receive_error=”error_name” receive_sender=”name” receive_type=”method_call” | “method_return” | “signal” | “error” receive_path=”/path/name”

send_requested_reply=”true” | “false” receive_requested_reply=”true” | “false”

eavesdrop=”true” | “false”

own=”name” own_prefix=”name” user=”username” group=”groupname”

send_destination跟receive_sender是指发送到目的为或者收到来自于该名字的owner，而不是该名字。因此 如果一个连接有三个服务A、B、C，如果拒绝发送到A，那么发送到B和C也不行。 其他的send_*和receive_*则匹配消息头的字段。

4.4.3 D-Bus系列之获取发送者UID及PID的方法

获取PID及UID的原理

```
org.freedesktop.DBus提供了一系列的消息，其中就有根据服务名获取进程PID及UID的接口
"org.freedesktop.DBus", --服务
"/org/freedesktop/DBus", --对象
"org.freedesktop.DBus", --接口
"GetConnectionUnixProcessID", --方法
 UINT32 GetConnectionUnixProcessID (in STRING bus_name);
"GetConnectionUnixUser", --方法
UINT32 GetConnectionUnixUser (in STRING bus_name);
```

QT DBUS获取的方法及示例

QT DBUS提供了相关调用的封装:

1. const QDBusMessage & QDBusContext::message () const Returns the message that generated this call.
2. QString QDBusMessage::service () const Returns the name of the service or the bus address of the remote method call.
3. QDBusReply QDBusConnectionInterface::servicePid (const QString & serviceName) const Returns the Unix Process ID (PID) for the process currently holding the bus service serviceName.

例如:

```
bool SomeMethod( const QString &name )
{
     qDebug() << "PID is: " << connection().interface()->servicePid( message().
↪service() );
}
```

D-Bus glib绑定及GDBus

D-Bus Glib 的绑定提供了获取发送者名字的方法:

```
const char *dbus_message_get_sender(DBusMessage *message);
```

但是没有提供获取进程PID及UID的方法，需自己编写代码调用GetConnectionUnixProcessID和GetConnectionUnixUser方法。好像GDbus也没有提供，只找到了g_dbus_message_get_sender()方法。

```
/* proxy for getting PID info */
g_dbus_proxy_new_for_bus(G_BUS_TYPE_SYSTEM,
    G_DBUS_PROXY_FLAGS_DO_NOT_LOAD_PROPERTIES,
    NULL,
    "org.freedesktop.DBus",
    "/org/freedesktop/DBus",
    "org.freedesktop.DBus",
    NULL,
    (GAsyncReadyCallback)dbus_proxy_connect_cb,
    NULL);
void
dbus_proxy_connect_cb(GObject *source_object,
               GAsyncResult *res,
               gpointer user_data)
{
    GError *error = NULL;

    dbus_proxy = g_dbus_proxy_new_finish (res, &error);
    if (error) {
        g_warning("dbus_proxy_connect_cb failed: %s", error->message);
        g_error_free(error);
        dbus_proxy = NULL;
    }
    else {
        g_debug("dbus_proxy_connect_cb succeeded");
    }
}
gboolean
handle_request_sys_state (PowerdSource *obj, GDBusMethodInvocation     *invocation,␣
↪int state)
{
    // get the name of the dbus object that called us
    owner = g_dbus_method_invocation_get_sender(invocation);
    if (dbus_proxy) {
        result = g_dbus_proxy_call_sync(dbus_proxy,
                "GetConnectionUnixProcessID",
                g_variant_new("(s)", owner),
                G_DBUS_CALL_FLAGS_NONE,
                -1,
                NULL,
                &error);
        if (error) {
            g_error("Unable to get PID for %s: %s", owner, error->message);
            g_error_free(error);
            error = NULL;
        }
        else {
            g_variant_get(result, "(u)", &owner_pid);
            g_info("request is from pid %d\n", owner_pid);
        }
    }
    ...
}
```

参考资料

1. D-Bus Specification
2. Getting the PID and Process Name From a dbus Caller in C
3. Get sender PID from DBUS
4. Qt doc及DBus-Glib、GDBus文档

4.5 shell 脚本常用操作入门

4.5.1 shell 字符串操作

判断读取字符串值

表达式	含义
${var}	变量 var 的值，与 $var 相同
${var-DEFAULT}	如果 var 没有被声明，那么就以 $DEFAULT 作为其值 *
${var:-DEFAULT}	如果 var 没有被声明，或者其值为空，那么就以 $DEFAULT 作为其值 *
${var=DEFAULT}	如果 var 没有被声明，那么就以 $DEFAULT 作为其值 *
${var:=DEFAULT}	如果 var 没有被声明，或者其值为空，那么就以 $DEFAULT 作为其值 *
${var+OTHER}	如果 var 声明了，那么其值就是 $OTHER, 否则就为 null 字符串
${var:+OTHER}	如果 var 被设置了，那么其值就是 $OTHER, 否则就为 null 字符串
${var?ERR_MSG}	如果 var 没被声明，那么就打印 $ERR_MSG *
${var:?ERR_MSG}	如果 var 没被设置，那么就打印 $ERR_MSG *
${!varprefix*}	匹配之前所有以 varprefix 开头进行声明的变量
${!varprefix@}	匹配之前所有以 varprefix 开头进行声明的变量

字符串操作（长度，读取，替换）

表达式	含义
${#string}	$string 的长度
${string:position}	在 $string 中，从位置 $position 开始提取子串
${string:position:length}	在 $string 中，从位置 $position 开始提取长度为 $length 的子串
${string#substring}	从变量 $string 的开头，删除最短匹配 $substring 的子串
${string##substring}	从变量 $string 的开头，删除最长匹配 $substring 的子串
${string%substring}	从变量 $string 的结尾，删除最短匹配 $substring 的子串
${string%%substring}	从变量 $string 的结尾，删除最长匹配 $substring 的子串
${string/substring/replacement}	使用 $replacement, 来代替第一个匹配的 $substring
${string//substring/replacement}	使用 $replacement, 代替所有匹配的 $substring
${string/#substring/replacement}	如果 $string 的前缀匹配 $substring, 那么就用 $replacement 来代替匹配到的 $substring
${string/%substring/replacement}	如果 $string 的后缀匹配 $substring,

举例

```
/// 取得字符串长度
string=abc12342341          // 等号二边不要有空格
echo ${#string}             // 结果 11
```

(下页继续)

(续上页)

```
expr length $string          // 结果 11
expr "$string" : ".*"        // 结果 11 冒号二边要有空格，这里的：根 match 的用法差不多

/// 字符串所在位置
expr index $string '123'     // 结果 4 字符串对应的下标是从 1 开始的
str="abc"
expr index $str "b"  # 2
expr index $str "x"  # 0
expr index $str ""   # 0

/// 字符串截取
echo ${string:4}       //2342341  从第 4 位开始截取后面所有字符串
echo ${string:3:6}     //123423   从第 3 位开始截取后面 6 位
echo ${string: -4}     //2341  : 右边有空格    截取后 4 位
echo ${string:(-4)}    //2341  同上
expr substr $string 3 3   //123  从第 3 位开始截取后面 3 位
str="abcdef"
expr substr "$str" 4 5  # 从第四个位置开始取 5 个字符， def
echo ${str:(-6):5}        # 从倒数第二个位置向左提取字符串，abcde
echo ${str:(-4):3}      # 从倒数第二个位置向左提取 6 个字符，cde

/// 匹配显示内容
expr match $string '\([a-c]*[0-9]*\)'  //abc12342341
expr $string : '\([a-c]*[0-9]\)'        //abc1
expr $string : '.*[([0-9][0-9][0-9]\)' //341 显示括号中匹配的内容

/// 截取不匹配的内容
echo ${string#a*3}     //42341  从 $string 左边开始，去掉最短匹配子串
echo ${string#c*3}     //abc12342341  这样什么也没有匹配到
echo ${string#*c1*3}   //42341  从 $string 左边开始，去掉最短匹配子串
echo ${string##a*3}    //41     从 $string 左边开始，去掉最长匹配子串
echo ${string%3*1}     //abc12342  从 $string 右边开始，去掉最短匹配子串
echo ${string%%3*1}    //abc12     从 $string 右边开始，去掉最长匹配子串
str="abbc,def,ghi,abcjkl"
echo ${str#a*c}      # 输出，def,ghi,abcjkl  一个井号 (#) 表示从左边截取掉最短的匹配 （这里把
↪abbc 字串去掉）
echo ${str##a*c}     # 输出 jkl,                两个井号 (##) 表示从左边截取掉最长的匹配 （这里
把 abbc,def,ghi,abc 字串去掉）
echo ${str#"a*c"}    # 输出 abbc,def,ghi,abcjkl 因为 str 中没有"a*c"子串
echo ${str##"a*c"}   # 输出 abbc,def,ghi,abcjkl 同理
echo ${str#*a*c*}    # 空
echo ${str##*a*c*}   # 空
echo ${str#d*f)      # 输出 abbc,def,ghi,abcjkl,
echo ${str#*d*f}     # 输出，ghi,abcjkl
echo ${str%a*l}      # abbc,def,ghi  一个百分号 (%) 表示从右边截取最短的匹配
echo ${str%%b*l}     # a             两个百分号表示 (%%) 表示从右边截取最长的匹配
echo ${str%a*c}      # abbc,def,ghi,abcjkl

/// 匹配并且替换
echo ${string/23/bb}    //abc1bb42341  替换一次
echo ${string//23/bb}   //abc1bb4bb41  双斜杠替换所有匹配
echo ${string/#abc/bb} //bb12342341   #以什么开头来匹配，根 php 中的 ^ 有点像
echo ${string/%41/bb}   //abc123423bb  % 以什么结尾来匹配，根 php 中的 $ 有点像
str="apple, tree, apple tree"
echo ${str/apple/APPLE}    # 替换第一次出现的 apple
echo ${str//apple/APPLE}   # 替换所有 apple
```

(下页继续)

(续上页)

```
echo ${str/#apple/APPLE}  # 如果字符串 str 以 apple 开头，则用 APPLE 替换它
echo ${str/%apple/APPLE}  # 如果字符串 str 以 apple 结尾，则用 APPLE 替换它

/// 比较
[[ "a.txt" == a* ]]         # 逻辑真 (pattern matching)
[[ "a.txt" =~ .*\.txt ]]    # 逻辑真 (regex matching)
[[ "abc" == "abc" ]]        # 逻辑真 (string comparision)
[[ "11" < "2" ]]            # 逻辑真 (string comparision), 按 ascii 值比较

/// 字符串删除
$ test='c:/windows/boot.ini'
$ echo ${test#/}
c:/windows/boot.ini
$ echo ${test#*/}
windows/boot.ini
$ echo ${test##*/}
boot.ini
$ echo ${test%/*}
c:/windows
$ echo ${test%%/*}
#${变量名#substring 正则表达式}从字符串开头开始配备 substring, 删除匹配上的表达式。
#${变量名 %substring 正则表达式}从字符串结尾开始配备 substring, 删除匹配上的表达式。
#注意: ${test##*/},${test%/*} 分别是得到文件名，或者目录地址最简单方法。
```

4.5.2 数组操作

声明一个数组

```
declare -a array
```

数组赋值

```
A. array=(var1 var2 var3 ... varN)
B. array=([0]=var1 [1]=var2 [2]=var3 ... [n]=varN)
C. array[0]=var1
   arrya[1]=var2
   ...
   array[n]=varN
D. ARRAY=()
   ARRAY+=('foo')
   ARRAY+=('bar')
```

计算数组元素个数

```
${#array[@]}  或者  ${#array[*]}
```

引用数组

```
echo ${array[n]}
```

遍历数组

```
filename=(`ls`)
for var in ${filename[@]};do
```

(下页继续)

(续上页)

```
    echo $var
done
```

4.5.3 参考资料

1. linux shell 字符串操作详解 （长度，读取，替换，截取，连接，对比，删除，位置 ）
2. BASH 数组用法小结

4.6 JavaScript 的移位运算与 IP 地址处理

4.6.1 JS 左移运算符号位的问题

最近在做项目时有一个需求，将用户输入的 地址 / 掩码 对解析出来，并将数字掩码转换成点分的格式。

想到在 C 代码里面应该还算容易实现，通过最大 32 位整数位移就可完成，但是在 JavaScript 中如何实现还是很不清楚的。

因为考虑到 JavaScript 是弱类型的语言，首先就会遇到数字和字符串的转换等问题，接着还有按位操作的问题也不知道在 JavaScript 中如何实现。 然后开始查询资料尝试解决这些问题。数字和字符串类型的转换是不需要的，写代码尝试了下应该 JavaScript 自动做了类型转换。

首先想到的思路是根据最大 32 位整数位移获取掩码对应的整数值，然后将结果转换成点分格式的字符串。在网上找到了如下的代码。

```
function ip2long(ip) {
    var ipl=0;
    ip.split('.').forEach(function( octet ) {
        ipl<<=8;
        ipl+=parseInt(octet);
    });
    return(ipl >>>0);
}

function long2ip (ipl) {
    return ( (ipl>>>24) +'.' +
        (ipl>>16 & 255) +'.' +
        (ipl>>8 & 255) +'.' +
        (ipl & 255) );
}
```

有了这两个方法就可以将计算得到的掩码整数值转换成点分格式了。 但是在做位运算的时候发现了一个坑，在一边查资料一边尝试实现的第一个版本中，掩码为 0 和掩码为 32 产生的结果一样，都是 255.255.255.255。

看来此方案不可行，接着查资料发现了一个新的思路，根据掩码生成四个小于等于 255 的值，将这四个值拼成点分格式。这样实现避免了对符号位的操作，甚好。

```
function createNetmaskAddr(bitCount) {
  var mask=[];
  for(i=0;i<4;i++) {
    var n = Math.min(bitCount, 8);
```

(下页继续)

(续上页)

```
    mask.push(256 - Math.pow(2, 8-n));
    bitCount -= n;
  }
  return mask.join('.');
}
```

4.6.2 参考资料

- Unsigned Integer in Javascript
- ECMAScript 位运算符
- CIDR to netmask converion in javascript

4.7 automake 和 autoconf 使用简明教程

4.7.1 步骤概览

1. create project
2. touch NEWS README ChangeLog AUTHORS
3. autoscan
4. configure.scan ==> configure.in/configure.ac
5. aclocal
6. autoheader（可选，生成 config.h.in)
7. Makefile.am（根据源码目录可能需要多个）
8. libtoolize –automake –copy –force（如果 configure.ac 中使用了 libtool)
9. automake –add-missing
10. autoconf
11. ./configure && make && make install

4.7.2 修改 configure.scan ==> configure.ac

1. 在 AC_INIT 宏下一行添加 AM_INIT_AUTOMAKE([foreign -Wall -Werror])，选项可根据自己需要修改，选项参见 automake 手册。
2. 若需要使用 config.h 的宏，添加 AC_CONFIG_HEADERS([config.h]) 宏。
3. 配置编译环境。AC_PROG_CXX->g++, AC_PROG_CC->gcc AC_PROG_RANLIB->libs，LT_INIT->使用 libtool，库可以指定 PKG_CHECK_MODULES()。
4. 添加自己的检测处理（可选）。
5. 在 AC_OUTPUT 上一行添加 AC_CONFIG_FILES 宏，指定输出的文件。如：AC_CONFIG_FILES([Makefile tools/Makefile])

pkg-config 的使用

为使工程支持 pkg-config，提供了以下的几个 autoconf 宏

PKG_PROG_PKG_CONFIG([MIN-VERSION]): 判断系统中的 pkg-config 及其版本符合兼容性要求。 PKG_CHECK_EXISTS(MODULES, [ACTION-IF-FOUND], [ACTION-IF-NOT-FOUND]): 检查系统中是否有某些模块。 PKG_CHECK_MODULES(VARIABLE-PREFIX, MODULES, [ACTION-IF-FOUND], [ACTION-IF-NOT-FOUND]): 检查系统中是否有某些模块，如果是则根据 pkg-config –cflags 和 pkg-config –libs 的输出设置<VARIABLE-PREFIX>_CFLAGS and <VARIABLE-PREFIX>_LIBS 变量。

4.7.3 编写自定义的 Autoconf 宏（一般不需要）

Autoconf 的自定义宏在 configure.ac 中调用。 #. 新建 m4 目录，在该目录中编写 xxx.m4 宏，运行 aclocal -I m4 生成 aclocal.m4: #. 在 Makefile 中添加 ACLOCAL_AMFLAGS = -I m4

4.7.4 编写 Makefile.am

1. 每个目录一个 Makefile.am 文件；同时在 configure.ac 的 AC_CONFIG_FILES 宏中指定输出所有的 Makefile 文件。
2. 父目录 Makefile.am 包含子目录：SUBDIRS=test tools
3. Makefile.am 中指明当前目录如何编译。

编译方式

类型	说明	使用方式
PROGRAMS	可执行程序	bin_PROGRAMS
LIBRARIES	库文件	lib_LIBRARIES
LTLIBRARIES	libtool 库文件	lib_LTLIBRARIES
HEADERS	头文件	include_HEADERS
SCRIPTS	脚本文件，有可执行权限	test_SCRIPTS（需要自定义 test 目录）
DATA	数据文件，无可执行权限	conf_DATA（需要自定义 conf 目录）

编译目标

编译目标其实就是编译类型对应的具体文件，其中需要 make 生成的文件主要有如下几个：可执行程序_PROGRAMS，普通库文件_LIBRARIES，libtool 库文件_LTLIBRARIES, 其它类型对应的编译目标不需要编译，源文件就是目标文件。 例如，对于 bin_PROGRAMS = target

```
target_SOURCES: 对应 gcc 命令中的源代码文件
target_LIBADD: 编译链接库时需要链接的其它库，对应 gcc 命令中的*.a, *.so 等文件
target_LDADD: 编译链接程序时需要链接的其他库，对应 gcc 命令中的*.a, *.so 等文件
target_LDFLAGS: 链接选项，对应 gcc 命令中的 -L, -l, -shared, -fpic 等选项
target_LIBTOOLFLAGS: libtool 编译时的选项
target**FLAGS（例如_CFLAGS/_CXXFLAGS): 编译选项，对应 gcc 命令中的 -O2, -g, -I 等选项
```

编译可执行程序

将非 main 函数所在目录的文件编译成静态链接库，然后采用链接静态库的方式编译可执行程序。 main 函数所在目录的文件加到 XXX_SOURCES 变量中。

编译库

Automake 天然支持编译静态库，只需要将编译类型指定为_LIBRARIES 即可。

动态库

需要注意的是：_LIBRARIES 只支持静态库（即*.a 文件）, 而不支持编译动态库 (*.so) 文件，要编译动态链接库，需要使用_PROGRAMS。

除此之外，还需要采用自定义目录的方式避开 Automake 的两个隐含的限制：

1. 如果使用 bin_PROGRAMS, 则库文件会安装到 bin 目录下，这个不符合我们对动态库的要求；
2. automake 不允许用 lib_PROGRAMS

也可以使用 libtool 来编译动态库。

libtool 库

libtoolize 提供了一种标准的方式来将 libtool 支持加入一个软件包， 而 GNU libtool 是一个通用库支持脚本，将使用动态库的复杂性隐藏在统一、可移植的接口中。

对于跨平台可移植的库来说，推荐使用 libtool 编译，而且 Automake 内置了 libtool 的支持，只需要将编译类型修改为_LTLIBRARIES 即可，如下为一个 Makefile.am 的例子：

```
lib_LTLIBRARIES=libtest.la
libtest_la_SOURCES=test.c
```

这里 lib_LTLIBRARIES 的意思是生成的动态库，然后指定动态库依赖的源文件 test.c，若有多个源文件用空格隔开。

需要注意的是：如果要使用 libtool 编译，需要在 configure.ac 中添加 LT_INIT 宏，同时注释掉 AC_PROG_RANLIB，因为使用了 LT_INIT 后，AC_PROG_RANLIB 就没有作用了。 如果 autoreconf 无法识别 LT_INIT 宏，你需要更新 libtool 或者在 configure.ac 里加上宏 AC_PROG_LIBTOOL ，表示利用 libtool 来自动生成动态库。 新工程中应该使用 LT_INIT 而不是使用 AC_PROG_LIBTOOL 。

格式模板

文件类型	书写格式
可执行文件	bin_PROGRAMS = foo foo_SOURCES =xxxx.c foo_LDADD = foo_LDFLAGS = foo_DEPENDENCIES =
静态库	lib_LIBRARIES = libfoo.a foo_a_SOURCES = foo_a_LDADD = foo_a_LIBADD = foo_a_LDFLAGS =
头文件	include_HEADERS = foo.h
数据文件	data_DATA = data1 data2

对于可执行文件和静态库类型，如果只想编译，不想安装到系统中，可以用 noinst_PROGRAMS 代替 bin_PROGRAMS，noinst_LIBRARIES 代替 lib_LIBRARIES。

Makefile.am 还提供了一些全局变量供所有的目标体使用：

变量	含义
INCLUDES	比如链接时所需要的头文件
LDADD	比如链接时所需要的库文件
LDFLAGS	比如链接时所需要的库文件选项标志
EXTRA_DIST	源程序和一些默认的文件将自动打入.tar.gz 包，其它文件若要进入.tar.gz 包可以用这种办法，比如配置文件，数据文件等等。
SUBDIRS	在处理本目录之前要递归处理哪些子目录

在 Makefile.am 中尽量使用相对路径，系统预定义了两个基本路径：

路径变量	含义
$(top_srcdir)	工程最顶层目录，用于引用源程序
$(top_builddir)	定义了生成目标文件上最上层目录，用于引用.o 等编译出来的目标文件。

automake 设置了默认的安装路径：

1. 标准安装路径 默认安装路径为：$(prefix) = /usr/local，可以通过./configure –prefix= 的方法来覆盖。 其它的预定义目录还包括：bindir = $(prefix)/bin, libdir = $(prefix)/lib, datadir = $(prefix)/share, sysconfdir = $(prefix)/etc 等等。
2. 定义一个新的安装路径 比如 test, 可定义 testdir = $(prefix)/test, 然后 test_DATA =test1 test2，则 test1，test2 会作为数据文件安装到 $(prefix)/ /test 目录下。

4.7.5 打包

Automake 缺省情况下会自动打包，自动打包包含如下内容：

1. 所有源文件
2. 所有 Makefile.am/Makefile.in 文件
3. configure 读取的文件
4. Makefile.am's (using include) 和 configure.ac' (using m4_include) 包含的文件
5. 缺省的文件，例如 README, ChangeLog, NEWS, AUTHORS

如果除了这些缺省的文件外，你还想将其它文件打包，有如下两种方法：

1. 粗粒度方式：通过 EXTRA_DIST 来指定。 例如： EXTRA_DIST=conf/config.ini test/test.php tools/initialize.sh
2. 细粒度方式：在"安装目录_编译类型 = 编译目标"前添加 dist（表示需要打包）, 或者 nodist（不需要打包）。 例如： dist_data_DATA = distribute-this；nodist_foo_SOURCES = do-not-distribute.c

4.7.6 一个例子

例子为一个简单但完整的项目，该项目只有 test.c 源代码文件，但使用了系统中使用 pkg-config 配置的 dbus 库。

代码目录如下

```
ChangeLog   configure.ac  Makefile.am  README
AUTHORS     COPYING       INSTALL  NEWS
src:
    Makefile.am test.c
```

生成后并修改过的 configure.ac 文件

```
dnl Process this file with autoconf to produce a configure script.

AC_PREREQ(2.59)
AC_INIT(dbus-tutorial, 1.0)
m4_ifdef([AM_SILENT_RULES], [AM_SILENT_RULES([yes])])

AC_CANONICAL_SYSTEM
AM_INIT_AUTOMAKE()

AC_PROG_CC

PKG_CHECK_MODULES(DBUS, dbus-1)
PKG_CHECK_MODULES(GLIB, glib-2.0)
PKG_CHECK_MODULES(DBUS_GLIB, dbus-glib-1)
```

(下页继续)

(续上页)

```
CFLAGS="$CFLAGS -O0  $DBUS_CFLAGS $GLIB_CFLAGS -g -Wall -Werror"
LDFLAGS="$LDFLAGS $DBUS_LIBS $GLIB_LIBS"

AC_CONFIG_FILES(Makefile src/Makefile)
AC_OUTPUT
```

主目录 Makefile.am

```
SUBDIRS=src
```

代码目录 Makefile.am，configure.ac 中使用 PKG_CHECK_MODULES 定义了 DBUS 和 GLIB 及 DBUS_GLIB 前缀。

```
bin_PROGRAMS= test
test_SOURCES=test.c
test_CFLAGS = @DBUS_CFLAGS@ @GLIB_CFLAGS@
test_LDADD =  \
              @GLIB_LIBS@ \
              @DBUS_LIBS@ \
              @DBUS_GLIB_LIBS@
```

4.7.7 参考链接

1. 大型项目使用 Automake/Autoconf 完成编译配置
2. 大型项目使用 Automake/Autoconf 完成编译配置 (2)
3. Makefile.am 详解
4. 使用 Automake 生成 Makefile 及动态库和静态库的创建
5. Autotools Mythbuster
6. The PKG_CHECK_MODULES Macro

4.8 Python 3 Bytes.decode 遇到的问题

使用 *Python 3* 的 *Subprocess* 模块执行 shell 命令，读取到的结果的类型是 *bytes*，如果是文本需要转成 *str* 类型。

一般来说，Linux 的中文环境大都使用 *utf-8* 编码，我执行操作的系统也确实使用的 *utf-8* 编码，然后还是遇到了问题……

即使是使用了 *utf-8* 编码，我们的文件名仍然可能会产生乱码，比如我们执行 *ls* 可能会看到这样的文件名：：

OITS??–????.mp4

而如果使用 *Subprocess* 模块执行 *ls* 命令，则 *result.decode("utf-8")* 会报 *UnicodeDecodeError* 异常，初见这个问题我非常惊讶，应该 *utf-8* 可以编码所有的字符了吧，为啥我用 *utf-8* decode 还会出现这样的问题。

我觉得出现这个问题的原因很可能是，*Linux* 文件系统使用的 *utf-8* 编码保存文件名，但是 **该文件是从 Windows 的文件系统拷贝过来**，而 **Windows 文件系统的默认编码则是不是 utf-8**，这样我们在 *shell* 执行 *ls* 命令时显示的就是乱码字符了，因为我们的文件系统是存的 *utf-8* 编码的文件名，自然该文件名也是按照 *utf-8* 来解码输出。

要解决这个问题也不是很麻烦，参考资料 1 的答案很清楚，要么使用兼容的 *cp437/latin-1* 解码，要么使用 *utf-8* 解码时进行容错处理。

```
>>> result = b'\xc5\xe0\xd1\xb5--\xd1\xee\xc0\xa5.mp4'
>>> result.decode("utf-8", errors="surrogateescape")
'\udcc5\udce0--\udcd1\udcee\udcc0\udca5.mp4'
>>> result.decode("cp437")
'┼α--ε└Ñ.mp4'
>>> result.decode("latin-1")
'ÅàÑµ--ÑîÀ¥.mp4'
```

4.8.1 参考资料

1. Convert bytes to a Python string
2. PEP 383 – Non-decodable Bytes in System Character Interfaces

4.9 Linux粘滞位说明

sticky bit: 该位可以理解为防删除位. 一个文件是否可以被某用户删除, 主要取决于该用户是否对该文件所属的目录具有写权限. 如果没有写权限, 则这个目录下的所有文件都不能被删除, 同时也不能添加新的文件. 如果希望用户能够添加文件但同时不能删除文件, 则可以对文件使用sticky bit位. 设置该位后, 就算用户对目录具有写权限, 也不能删除该文件.

要删除一个文件，你不一定要有这个文件的写权限，但你一定要有这个文件的上级目录的写权限。也就是说，你即使没有一个文件的写权限，但你有这个文件的上级目录的写权限，你也可以把这个文件给删除，而如果没有一个目录的写权限，也就不能在这个目录下创建文件。

如何才能使一个目录既可以让任何用户写入文件，又不让用户删除这个目录下他人的文件，sticky就是能起到这个作用。stciky一般只用在目录上，用在文件上起不到什么作用。

在一个目录上设了sticky位后，所有的用户都可以在这个目录下创建文件，但只能删除自己创建的文件(root除外)，这就对所有用户能写的目录下的用户文件启到了保护的作用。

可以通过chmod o+t tmp 来设置tmp目录的sticky bit，而且/tmp目录默认是设置了这个位的。

4.10 mysql绿色启动方法

下载 MySQL Linux - Generic 版本 并解压缩。

根据 `support-files/my-default.cnf` 编辑 `my.conf`

初始化并启动MySQL:

```
./bin/mysqld --initialize  --datadir=/datadir
./bin/mysqld --defaults-file=my.conf
```

4.11 LDAP

4.11.1 LDAP 定义

本文不打算重复 LDAP 的标准定义，仅谈一下自己的理解。

LDAP 译为轻量级目录访问协议，一般网上经常拿其与关系型数据库做类比并比较他们的不同。

事实上，LDAP 既不是数据库也不是存储数据的方法，而是用来访问数据的方法。LDAP 可以访问的 是存储在目录信息树（Directory Information Tree (DIT)）中的数据。

目录信息树存储数据的方法与关系型数据库非常的不同。关系型数据库的数据是存储在某一数据库 的某一数据表中的某以记录内的，因此数据库中定位一条记录数据需要三个要素：数据库、 数据表、记录号；目录信息树存储数据的方式是树状结构，定位一条数据的方法是从树根到叶子节点 的唯一路径，数据就存储在叶子节点中。

由此可见，LDAP 与关系型数据库类比其实是两种数据存储与访问方式的类比。

LDAP 是访问目录信息树中数据的协议

事实上，LDAP 仅定义了访问协议，数据的真实存储方式并不在 LDAP 的考量范围内，即只要提供了目录访问接口的数据都可以通过 LDAP 协议来访问，如果关系型数据库实现了该接口也是可以通过 LDAP 协议来反问的。从这个角度来讲，确实不应该将 LDAP 与关系型数据库做类比。

4.11.2 LDAP 特性

且 LDAP 对数据的读取和查询做了优化，并不适用于经常变动的数据。

LDAP 不定义客户端和服务端的工作方式，但会定义客户端和服务端的通信方式， 另外，LDAP 还会定义 LDAP 数据库的访问权限及服务端数据的格式和属性。

LDAP 有三种基本的通信机制：没有处理的匿名访问；基本的用户名、密码形式的认证；使用 SASL、SSL 的安全认证方式。LDAP 和很多其他协议一样，基于 tcp/ip 协议通信，注重服务的可用性、信息的保密性等等。部署了 LDAP 的应用不会直接访问目录中的内容，一般通过函数调用或者 API，应用可以通过定义的 C、Java 的 API 进行访问，Java 应用的访问方式为 JNDI(Java Naming and Directory Interface)。

4.11.3 LDAP 结构

目录信息树

目录信息树以目录条目（entry）来存储和组织数据，每一个目录条目通常描述一个对象（例如：一个人）， 目录条目有一个唯一名（DN，distinguished name）进行标识。DN 由一系列的相对唯一名（RDN，relative distinguished name）来标识。每一个目录条目由一或多个属性来描述该条目，例如描述人的条目有一个 电话号码的属性。

引用一张 Understanding LDAP Design and Implementation 的图示，

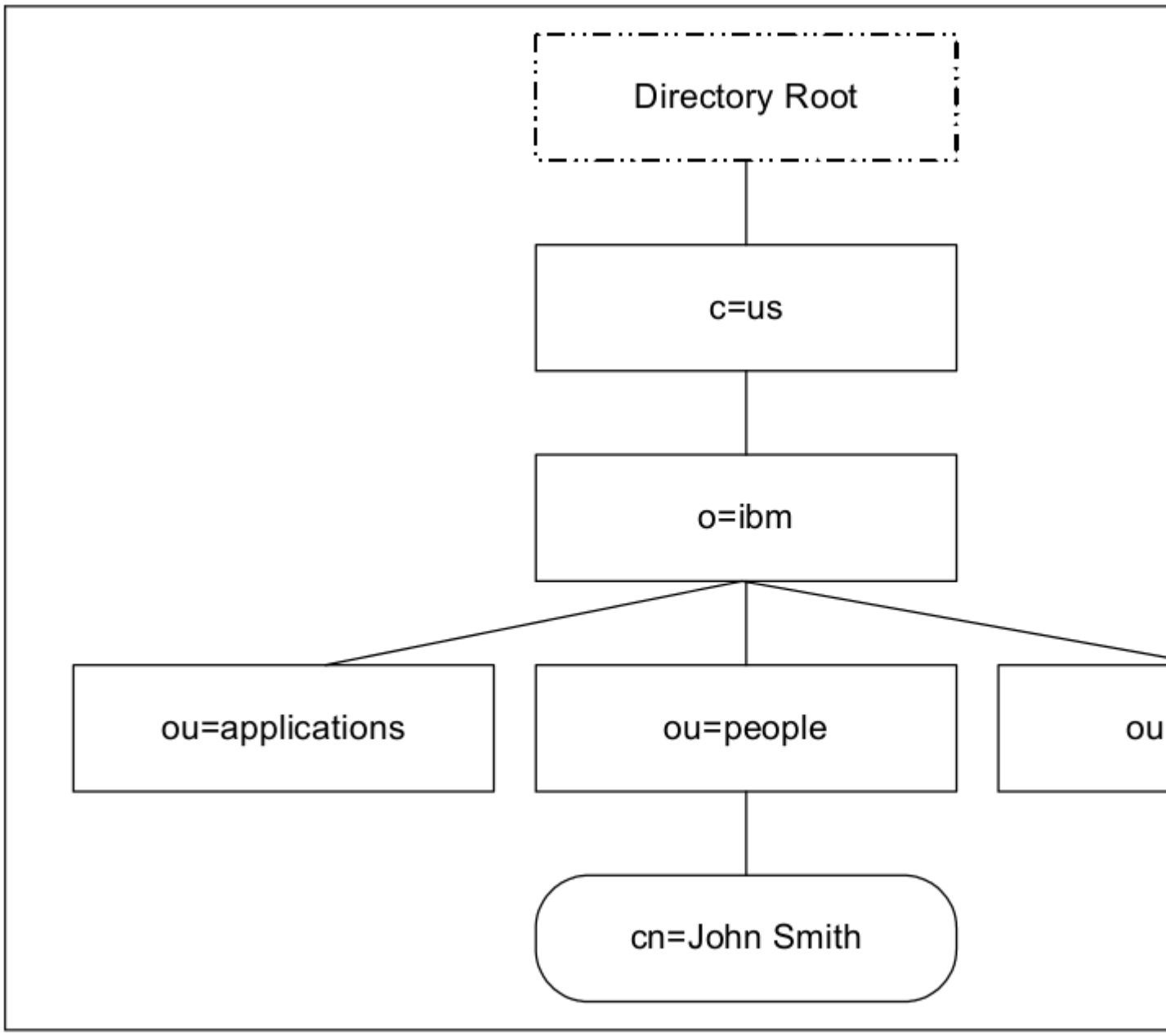


Figure 2-2 Example of a Directory Information Tree (DIT)

目录条目根据他们在目录信息树中的位置命名，上图底部的目录条目的 DN 唯一名是：：

cn=John Smith,ou=people,o=ibm,c=us. The organizational group people has the DN of ou=people,o=ibm,c=us.

LDAP 的查询等操作语句在这里就不介绍了，有需求的可以在参考资料中找。

4.11.4 LDAP 安全

LDAP 安全相关的资料请参考 LDAP 注入与防御剖析

4.11.5 LDAP 服务器配置

请参考 OpenLDAP 管理员指南

4.11.6 参考资料

- LDAP 注入与防御剖析
- Understanding LDAP Design and Implementation
- OpenLDAP 管理员指南

4.12 系统安全的思考

世上并没有绝对的安全，只有安全成本的问题。

真正的安全不过是一种意识。

4.12.1 过往

网络安全

工作相关，做的一直是跟系统安全相关的东西。

前几年的时候在做的是网络安全，所谓的网络安全。

说是网络安全，其实跟安全的关系不是很大，主要还是做网络设备，支持各种协议及功能， 还要尽可能地提高性能。

跟安全的那点儿关系就是，在网络设备里面嵌入了部分跟安全相关的功能。

想来，像访问包过滤（很多厂商也叫 ACL，访问控制列表），NAT 转换这些跟安全的关联性也不大吧，他们是更倾向于控制和管理的功能；DPI 协议识别呢？算是个基础功能吧，接下来可以根据 识别结果做控制与分析，也可以继续深入的做攻击防御。

那么真正跟安全密切相关的功能有哪些呢？ MAC/IP 绑定，攻击防护，网页防篡改等，这在网络安全设备像硬件防火墙这些，的功能中， 着实的算不算主要的功能吧，也不常用。只是说多一个功能就多一个卖点。

其实网路安全设备很少关注自身的安全，挺多厂商把侧重点放在了性能上面。

真的有可以提供整体安全方案的厂商吗？我是不信的，至少国内的厂商是这样的吧。 并不是说这些设备没有必要，确实很多功能还是很好的，但是跟系统安全相差的还是很远的吧。

4.12.2 现今

真正的系统安全

换过工作之后负责系统的一个安全模块，对安全有了稍微深入一点的理解。

安全与易用，在某种程度上会有一些冲突，但这个程度肯定不会很大。

易用也是一种安全，想用户所想。 系统遭遇到安全威胁时，用户能想到的，系统帮他去做了，用户没有想到的， 系统也帮他做好了，系统也帮他做了，这是易用，也是安全。

因此，系统的正确行为定义也是安全的基础，在威胁出现时不能做出正确响应，就没有安全可言了。

安全是不能有短板的，不仅要保证数据的安全，还要保证用户输入的安全，传输的安全等等， 缺了任何一个环节，就像系统有了漏洞一样，攻击者顺着这个漏洞可以获取系统中的很多东西。

安全的核心工作应该是在后台默默完成的，很少的提示用户，更很少让用户来选择， 就像武侠小说中的扫地僧，绝大多数时候用户都意识不到他的存在，他只是在后台默默的做着 自己的工作，他的存在只是为了那突然的攻击出现的时候。

稳定也是一种安全，我们无法保证每一次的修改都可以做到全系统的测试，而每次修改对系统 哪些模块有影响又是不完全预知的，尤其是系统的基础架构，略一更改影响面都很大。系统不 稳定就很容易出问题。

其实，安全是一种意识，这种意识应该无处不在。 开发中的安全意识保证我们的系统漏洞更少，系统更稳健。 如果我们在开发中测试代码随便写，变量不初始化，溢出不检查，系统就很容易被攻击。

使用中的安全意识保证了我们自己这个系统更加不容易被骗。 我们自己如果更加的注意自己的隐私不随意泄露，遇到电话 / 网络诈骗时可以多核实，社会 工程学也不那么容易就获取到我们的电话、邮箱、密码等信息了。

只要在系统的所有环节做到一般性的防护，就已经比在系统的关键环节做很强的防护但在 其他环节不防护安全很多了。

4.12.3 将来

系统的安全机制

系统还是要提供必要的基础的安全机制供其他的系统组件使用的。

如果需要系统提供的一些基础功能需要每个系统组件自己实现并维护，轮子重复造不说， 安全隐患的增加也是很严重的，因为保证这些基础功能的安全变的更加的麻烦。修复起来也很麻烦。

系统的安全机制做的好，省了很多的麻烦。

我们需要更加深入的研究与实现系统的安全机制。

4.13 VPN 高级选项那些事

4.13.1 一、VPN 高级选项

VPN 高级选项有哪些，都是什么意思

- **DNS 搜索域** These are for the mechanism for going from a machine name to a Fully Qualified Domain Name.

  DNS searches can only look at a Fully Qualified Domain Name, such as mymachine.example.com. But, it's a pain to type out mymachine.example.com, you want to be able to just type mymachine.

  Using Search Domains is the mechanism to do this. If you type a name that does not end with a period, it knows it needs to add the search domains for the lookup. So, lets say your Search Domains list was: example.org, example.com

  mymachine

  would try first mymachine.example.org, not find it, then try mymachine.example.com, found it, now done.

  mymachine.example.com

would try mymachine.example.com.example.org (remember, it doesn't end with a period, still adds domains), fail, then mymachine.example.com.example.com, not find it, fall back to mymachine.example.com, found it, now done

mymachine.example.com. Ends with a period, no searching, just do mymachine.example.com

Soooo. . . .

If you have your own DNS domain such as example.com, put it there. If not, ignore it. It really is more corporate than a home setting.

参考 What is the "Search Domains" field for in the tcp/ip DNS settings control panel/preference pane for?>

- **DNS 服务器** 域名系统（英文：Domain Name System，：DNS）是因特网的一项服务。它作为将域名和 IP 地址相互映射的一个分布式数据库，能够使人更方便的访问互联网。DNS 使用 TCP 和 UDP 端口 53。当前，对于每一级域名长度的限制是 63 个字符，域名总长度则不能超过 253 个字符。

  参考 域名系统

- **转发路线** 即转发路由，因 DNS 服务器不提供任何服务，因此所有的请求都需要经过转发才能到达可以响应请求的服务器，转发路线即配置发往哪些地址请求的需经过 VPN 转发。

为什么需要这些选项

这些选项其实只是一些基础的网络参数，因此所有的网络连接（包括 VPN）都需要这些选项。但是并不是所有的网络连接都需要手动配置这些参数。那么为什么 VPN 更加的需要配置这些参数呢?

VPN 是一种常用于连接中、大型企业或团体与团体间的私人网络的通讯方法。虚拟私人网络的讯息透过公用的网络架构（例如：互联网）来传送内联网的网络讯息。这种技术可以用不安全的网络（例如：互联网）送可靠、安全的息。『摘自 维基百科]

DNS 请求呢? 在未配置 VPN 之前，我们使用的是不安全网络上的 DNS 服务器，如果连接到 VPN 之后我们仍然连接不安全网络上的 DNS 服务器，如何保证我们的数据安全? DNS 搜索域是一个方便使用的选项。

路由则指定了哪些数据是需要 VPN 网络来保护的，如果不指定，或者系统中所有的流量都经过 VPN，但是 VPN 连接到的私有网络并不能提供不安全网络中所有的网络请求；或者 系统中所有的流量都不经过 VPN 服务器，连接 VPN 干嘛，当摆设吗?

由此可见，这些网络高级选项在 VPN 的配置中还是很有必要的。

4.13.2 二、关于 connman

使用 connman 管理系统网络连接的例子并不多，网络上相关的资料也很少。 万幸的是，connman 自带的文档 (doc 目录）大概可以把 connman 的设计原则和使用方法解释清除了。 以下内容及为参考该文档及源代码以及本人的推测得来的，不一定准确。

connman 是如何管理所有连接的

```
src/provider.c  -- 管理 connman 中每一个（不是每一种）连接，保存连接。
src/service.c   -- 管理连接服务。
src/task.c  -- connman 中对连接的代码，负责创建与维护真正的连接进程。
```

这几个代码文件大概实现了 connman 连接管理的框架，但是新建连接后还需要设置很多的网络参数，等等，正是这一部分使得 connman 显得更加的复杂。

connman 是如何管理网络参数的（路由、DNS 等）

connman 中对网络参数的管理是基于连接的，即每个连接都有不同的网络参数配置，该连接生效时 connman 会根据 连接属性更新系统的网络参数。

connman 封装了很多对系统网络参数修改的 API，如下列举部分：：

src/inet.c – 实现对系统路由的配置 src/ipconfig.c – 实现对系统地址的配置 src/resolver.c – 实现对系统 DNS 的配置，connman 有选项支持 dns 代理

注解： connman-vpn 与上述描述并不完全一致。 当然，vpn 连接也是由 connman_task 创建具体的任务来连接的，但是。 vpn/vpn-manager.c 提供新建 / 删除 VPN 连接的功能 (create/remove/get_connections...)。 vpn/vpn-provider.c 提供了 vpn 连接 / 断开功能 (do_connect/do_disconnect...)。 vpn 连接建立 / 删除时会发送 ConnectionAdded/ConnectionRemoved 信号， vpn 连接时会发送 PropertyChanged 信号。 connman 的 vpn 插件会监听这些信号，在新建 / 删除 vpn 连接时会在 connman 进程中建立该连接的 provider 及 service。 connman 监听到 PropertyChanged 信号时会根据属性设置系统当前的网络参数（dns 等）。

推荐文档阅读顺序： vpn-overview.txt -> vpn-manager-api.txt -> vpn-connection-api.txt

4.13.3 四、Qt 、QML and D-Bus

Connman 是以 daemon 进程在系统后台运行的，要访问 Connman 提供的服务，只能通过进程间通信类似的机制。 事实上 Connman 提供的服务都是以 D-Bus 方法即信号作为 API 接口的。

例如，新建 / 删除 VPN、连接 VPN 的接口如下：

```
static DBusMessage *create(DBusConnection *conn, DBusMessage *msg, void *data);
static DBusMessage *remove(DBusConnection *conn, DBusMessage *msg, void *data);
static DBusMessage *do_connect(DBusConnection *conn, DBusMessage *msg, void *data);
```

Qt 对 D-Bus 的支持

Qt 对 D-Bus 的支持算是基本完善，该有的都可以有，不该有的可能会可以有。（：D）

可以通过 Qt 中 D-Bus 相关的库函数创建 D-Bus 服务，或者使用别人提供的服务。对发送接收数据类型的支持也比较完整，不仅能够 收发基本的整数、字符串等，复杂的字典、数组等自然也不在话下。

但是 Qt 对 DBUS_TYPE_STRUCT 的支持需要稍多做一些工作，下面的章节会有介绍。

QML 对 D-Bus 的支持

很遗憾，QML 原生并不支持 D-Bus，但是可以通过两种变通的途径使用。 第一是，在 C++ 代码中封装调用 D-Bus 的接口，并注册到 QML 中。 第二种，是采用非 QT 官方的插件，实现，例如： Nemo Mobile D-Bus QML Plugin

好吧，其实是一种，第二种其实同样是 C++ 代码中封装了调用 D-Bus 的接口，但是除此之外，还有什么办法可以扩展 QML 不支持的功能吗?

Qt 对 D-Bus 中 DBUS_TYPE_STRUCT 的支持

Qt 有自己的类型系统，不知是该庆幸还是该懊恼。 Qt 的类型系统极大的丰富了我们的精神文化生活，噢不，是极大的方便了我们的开发，QVariant，信号 / 槽（QObject）等等。 但是这样一来我们自己定义的类型却无法使用这些方便的特性，而且 Qt D-Bus 也不支持自定义类型的发送与接收。

幸运的是，上帝在关上这扇门的时候悄悄给我们开了一扇窗，我们可以将自己定义的类型注册到 Qt 的元类型系统中去， 这样我们自己定义的类型也可以使用 Qt 提供的很多方便的特性了，最重要的是我们自定义的结构也可以通过 Qt 的 D-Bus 接口发送与接收了。

创建方法在此不表，无非是在适当的地方增加几次调用：：

Q_DECLARE_METATYPE(Type)； int qRegisterMetaType(const char * typeName)； int qDBusRegisterMetaType()；

详情参看如下链接：

- 创建自定义 Qt 类型
- Problems with marshalling a struct to Qt/DBus

4.13.4 五、Linux 连接管理

都有哪些连接管理实现

- Android：ConnectivityManager
- **NetworkManager** NetworkManager is a set of co-operative tools that make networking simple and straightforward. Whether Wi-Fi, wired, bond, bridge, 3G, or Bluetooth, NetworkManager allows you to quickly move from one network to another: once a network has been configured and joined, it can be detected and re-joined automatically the next time its available.
- **ConnMan** ConnMan is a daemon for managing Internet connections within embedded device and integrates a vast range of communication features usually split between many daemons such as DHCP, DNS and NTP. The result of this consolidation is low memory consumption with a fast, coherent, synchronized reaction to changing network conditions.

为什么需要连接管理

几乎所有的现代操作系统都有统一的连接管理，这是为什么呢？ 其实这个问题我也不知道。所以，下面的内容纯属揣测，如有不对恳请指正。

大概是有两个原因吧，我想。 一是便于用户的配置，试想，用户连接上一个新的网络（有线、无线、VPN 等）后，要手动的去修改 DNS、路由、地址等信息， 肯定是不可原谅的，或者进一步，需要在不同的位置分别通过不用的程序去配置不同的网络参数，少改了一项网络可就不正常了哦。

二是便于网络的管理，如果每种连接自己管自己的网络配置，可是这些配置的生效可是在一个系统上的，于是每个程序都去修改 DNS 配置，路由，地址等信息，你确保不会改乱?

其实反观其他子系统，声音肯定要在所有要播放 / 录制声音的程序后面有一个 daemon 来负责系统的混音及播放工作，不可能每个程序各播各的，你肯定不原因听到那种声音的。 显示子系统不可能是每个想要在屏幕上显示东西的程序自己向屏幕上写吧，这样我显示了一个窗口，你显示了一个通知，我有显示了一个文档，你确定用户能够看得请？所以还是需要有显示管理器在后面跑的。

同理，系统的网络配置大家一起改，你确定不会改乱？ 这大概是一个趋势吧，只有一种或者两种网络连接的时候，我可以随便改，要是系统有很多种连接类型，可就不能胡来了。

4.14 sphinx 生成本地化的文档

4.14.1 快速指南

1. 使用 *pip install sphinx-intl* 或 *easy_install sphinx-intl* 安装 sphinx-intl。

2. 添加如下配置到你的 Sphinx 文档配置文件 conf.py:

```
locale_dirs = ['locale/']   # path is example but recommended.gett
ext_compact = False     # optional.
```

3. 根据原文档生成可供翻译的 pot 文件：：

   $ make gettext

   执行此命令会在 *_build/locale* 目录生成很多 pot 文件。

4. 生成 / 更新 locale_dir

```
$ sphinx-intl update -p _build/locale -l zh_CN -l ja
```

   执行此命令会在 *locale_dir* 生成如下包含 po 文件的目录

   - ./locale/zh_CN/LC_MESSAGES/
   - ./locale/ja/LC_MESSAGES/

   支持的语言代码参考 http://www.sphinx-doc.org/en/stable/config.html#confval-language

   - bn – Bengali
   - ca – Catalan
   - cs – Czech
   - da – Danish
   - de – German
   - en – English
   - es – Spanish
   - et – Estonian
   - eu – Basque
   - fa – Iranian
   - fi – Finnish
   - fr – French
   - he – Hebrew
   - hr – Croatian
   - hu – Hungarian
   - id – Indonesian
   - it – Italian
   - ja – Japanese
   - ko – Korean
   - lt – Lithuanian
   - lv – Latvian
   - mk – Macedonian
   - nb_NO – Norwegian Bokmal
   - ne – Nepali

- nl – Dutch
- pl – Polish
- pt_BR – Brazilian Portuguese
- pt_PT – European Portuguese
- ru – Russian
- si – Sinhala
- sk – Slovak
- sl – Slovenian
- sv – Swedish
- tr – Turkish
- uk_UA – Ukrainian
- vi – Vietnamese
- zh_CN – Simplified Chinese
- zh_TW – Traditional Chinese

5. 翻译 *./locale/<lang>/LC_MESSAGES/* 目录下的 po 文件
6. 生成本地化的文档，你需要修改 conf.py 文档的 *language* 选项，或者在命令指定此选项：：

   $ make -e SPHINXOPTS="-D language='de'" html

如此，本地化的文档就在 *_build/html* 生成了。

4.14.2 参考资料

1. Sphinx Internationalization

4.15 简易 git 服务器

4.15.1 简易 gitolite 服务器搭建

服务器搭建

步骤如下

1. 创建 git 用户

```
sudo adduser git
su - git
```

2. 添加 ssh 公钥

```
mkdir .ssh
cat /tmp/id_rsa.lenny.pub >> ~/.ssh/authorized_keys
```

3. 在任何 git 用户可读写的位置新建 git 裸仓库

```
mkdir project.git
cd project.git
git --bare init
```

服务器使用

远程直接克隆和推送内容

```
git clone git@gitserver:/home/git/project.git
cd project
echo "sample git project" README
git commit -am 'fix for the README file'
git push origin master
```

4.15.2 使用 gitolite 搭建 git 服务器

服务器搭建

```
apt-get install gitolite3 git-core
```

安装 gitolite3 会要求输入管理员的 ssh 公钥。

服务器管理

下载管理仓库

Gitolite 使用 gitolite-admin.git 仓库来管理，所以需要抓下来修改、设置（未来所有管理也是如此）。

```
git clone gitolite@gitserver:gitolite-admin
cd gitolite-admin
ls
conf/gitolite.conf # 配置项，配置谁可以读写哪个仓库 Repository。
keydir # 目录，存放每个帐号的 public key。 放置的文件命名：user1.pub, user2.pub (user1,␣
↪user2.. 为帐号名称（文件名 = 帐号名），建议使用"帐号.pub" 文件名）
```

新增帐号

```
cd gitolite-admin
cp /tmp/user1.pub keydir/user1.pub # 依照实际帐号命名，不要取 user1, user2
cp /tmp/user1.pub keydir/user1@machine.pub # 若相同帐号，使用 user@machine.pub
cp /tmp/user2.pub keydir/user2.pub
git add keydir/user1.pub keydir/user1@machine.pub keydir/user2.pub
git commit -m 'add user1, user1@machine, user2 public key'
git push
```

gitolite.conf 配置

```
# 取自 2.3.1 授文件基本法
@admin = jiangxin wangsheng

repo gitolite-admin
RW+     = jiangxin

repo ossxp/.+
C       = @admin
RW      = @all

repo testing
RW+                     =   @admin
RW      master          =   junio
RW+     pu              =   junio
RW      cogito$          =   pasky
RW      bw/             =   linus
-                          =   somebody
RW      tmp/             =   @all
RW      refs/tags/v[0-9] =   junio

# 取自 2.3.3 ACL
repo testing
RW+   = jiangxin @admin
RW    = @dev @test
R      = @all
```

```
repo 语法

    repo 法: 《限》 『零或多正表示式批配的引用』 = <user> [<user> ...]
    每指令必指定一限，限可以用下面任何一限的字: C, R, RW, RW+, RWC, RW+C, RWD, RW+D, RWCD,␣
↪RW+CD
        C : 建立
        R : 取
        RW : 取 + 入
        RW+ : 取 + 入 + rewind 的 commit 做制 Push
        RWC : 授指令定 regex (regex 定的 branch、tag 等), 才可以使用此授指令。
        RW+C : 同上，C 是允建立 和 regex 配的引用 (branch、tag 等)
        RWD : 授指令中定 regex (regex 定的 branch、tag 等), 才可以使用此授指令。
        RW+D : 同上，D 是允除 和 regex 配的引用 (branch、tag 等)
        RWCD : 授指令中定 regex (regex 定的 branch、tag 等), 才可以使用此授指令。
        RW+CD : C 是允建立 和 regex 配的引用 (branch、tag 等), D 是允除 和 regex 配的引用␣
↪(branch、tag 等)
        - : 此定不能入，但是可以取
        : 若 regex 不是以 refs/ , 自於前面加上 refs/heads/

群组

    @all 代表所有人的意思
    @myteam user1 user2 : user1, user2 都是於 myteam 群
```

注解: gitolite 配置的应通过 gitolite-admin 仓库修改并提交到服务器，如若手动更改了服务器端的配置，如更改仓库的存储位置（仓库位置为 gitolite 用户的家目录）等， 需运行 gitolite setup/gl-setup 修复。

4.15.3 gitweb 配置

```
sudo apt-get install gitweb
sudo vim /etc/gitweb.conf
    $projectroot = "/home/git/repositories/";
    $projects_list = "/home/git/projects.list";
sudo mv /etc/apache2/conf.d/gitweb /etc/apache2/conf-available/gitweb.conf
    Options +FollowSymLinks +ExecCGI
sudo a2enconf gitweb
sudo a2enmod cgi
sudo apache2ctl restart
```

需主意 gitweb 对 git 的 projectroot 和 projects_list 要有权限。 可以设置 gitolite 的掩码：

```
vim /var/lib/gitolite3/.gitolite.rc #$REPO_UMASK = 0027;
sudo usermod -G gitolite3 www-data
sudo chmod 640 /val/lib/gitolite2/projects.list
```

4.15.4 svn 仓库迁移到 git

参考 迁移 SVN 到 Git Server 及 迁移 SVN 到 Git Server (git-scm)

4.15.5 引用

1. Linux 使用 Gitolite 架 Git Server
2. 服务器上的 Git - Gitolite
3. How to install gitweb in Ubuntu

CHAPTER 5

量化投资

Contents:

5.1 量化投资简介

5.1.1 选股

量化选股就是用量化的方法选择确定的投资组合，期望这样的投资组合可以获得超越大盘的投资收益。常用的选股方法有多因子选股、行业轮动选股、趋势跟踪选股等。

1. 多因子选股

多因子选股是最经典的选股方法，该方法采用一系列的因子（比如市盈率、市净率、市销率等）作为选股标准，满足这些因子的股票被买入，不满足的被卖出。比如巴菲特这样的价值投资者就会买入低PE的股票，在PE回归时卖出股票。

1. 风格轮动选股

风格轮动选股是利用市场风格特征进行投资，市场在某个时刻偏好大盘股，某个时刻偏好小盘股，如果发现市场切换偏好的规律，并在风格转换的初期介入，就可能获得较大的收益。

1. 行业轮动选股

行业轮动选股是由于经济周期的的原因，有些行业启动后会有其他行业跟随启动，通过发现这些跟随规律，我们可以在前者启动后买入后者获得更高的收益，不同的宏观经济阶段和货币政策下，都可能产生不同特征的行业轮动特点。

1. 资金流选股

资金流选股是利用资金的流向来判断股票走势。巴菲特说过，股市短期是投票机，长期看一定是称重机。短期投资者的交易，就是一种投票行为，而所谓的票，就是资金。如果资金流入，股票应该会上涨，如果资金流出，股票应该下跌。所以根据资金流向就可以构建相应的投资策略。

1. 动量反转选股

动量反转选股方法是利用投资者投资行为特点而构建的投资组合。索罗斯所谓的反身性理论强调了价格上涨的正反馈作用会导致投资者继续买入，这就是动量选股的基本根据。动量效应就是前一段强势的股票在未来一段时间继续保持强势。在正反馈到达无法持续的阶段，价格就会崩溃回归，在这样的环境下就会出现反转特征，就是前一段时间弱势的股票，未来一段时间会变强。

1. 趋势跟踪策略

当股价在出现上涨趋势的时候进行买入，而在出现下降趋势的时候进行卖出，本质上是一种追涨杀跌的策略，很多市场由于羊群效用存在较多的趋势，如果可以控制好亏损时的额度，坚持住对趋势的捕捉，长期下来是可以获得额外收益的。

5.1.2 择时

量化择时是指采用量化的方式判断买入卖出点。如果判断是上涨，则买入持有；如果判断是下跌，则卖出清仓；如果判断是震荡，则进行高抛低吸。

常用的择时方法有：趋势量化择时、市场情绪量化择时、有效资金量化择时、SVM量化择时等。

5.1.3 仓位管理

仓位管理就是在你决定投资某个股票组合时，决定如何分批入场，又如何止盈止损离场的技术。 常用的仓位管理方法有：漏斗型仓位管理法、矩形仓位管理法、金字塔形仓位管理法等

5.1.4 止盈止损

止盈，顾名思义，在获得收益的时候及时卖出，获得盈利；止损，在股票亏损的时候及时卖出股票，避免更大的损失。 及时的止盈止损是获取稳定收益的有效方式。

5.1.5 策略的生命周期

一个策略往往会经历产生想法、实现策略、检验策略、运行策略、策略失效几个阶段。

- 产生想法

  任何人任何时间都可能产生一个策略想法，可以根据自己的投资经验，也可以根据他人的成功经验。

- 实现策略

  产生想法到实现策略是最大的跨越，实现策略可以参照上文提到的“一个完整的量化策略包含哪些内容？”

- 检验策略

  策略实现之后，需要通过历史数据的回测和模拟交易的检验，这也是实盘前的关键环节，筛选优质的策略，淘汰劣质的策略。

- 实盘交易

  投入资金，通过市场检验策略的有效性，承担风险，赚取收益。

- 策略失效

  市场是千变万化的，需要实时监控策略的有效性，一旦策略失效，需要及时停止策略或进一步优化策略。 策略没有达到预期收益，就是失效了，可以调整β值建立失效预警函数插入调用。

5.2 量化回测引擎浅谈

最近阅读了几个量化回测引擎的源码，对量化回测引擎的原理有了一些理解。

米筐开源的 RQAlpha 引擎工程化非常好，代码整洁，文档全，推荐阅读学习。 PyAlgoTrade 的代码也很易读，两个结合阅读效果不错。

这两个量化引擎都是基于事件的，即引擎基于时间及交易数据产生事件，用户的策略订阅事件，在事件发生时用户注册的函数会被调用，根据行情信息决定买入还是卖出。账户及券商模块比较特殊，他们既是事件的订阅者也是事件的产生者（订阅行情事件用于产生交易，生成交易事件通知用户）。

常见的事件有：

- on_init
- on_day_start
- on_bar
- on_day_end

当然真实的交易引擎中还有很多其他事件，但最重要的就是这几类事件。

从原理上来讲，量化引擎也是一个决策系统，输入一般是行情数据，用户的策略在引擎内运行，根据行情产生决策，而券商 / 账户板块则负责决策的执行与统计。

RQAlpha 是单事件源，行情以外的信息需要用户调用 api 或自己去取。

PyAlgoTrade 为多事件源，所有的输入都可以是事件源，可以有多种不同的事件源，例如根据关键字过滤 twitter 上的内容等。

当然在 RQAlpha 中也可以通过在 AbstractEventSource 的 event 函数（事件生成器）产生多种类型数据的事件，但实现起来略显麻烦。 而且官方实现中的两个例子（ SimulationEventSource/RealtimeEventSource ）也是只有行情数据的事件源。

其他模块的实现类似，在此就不多写了，还是多看代码吧，看代码的过程中画了两个简单的结构图，供参考。

5.3 凯利公式

5.4 移动平均线

- 算术平均(MA)
- 加权平均(WMA)

5.4.1 ma,dma,ema,sma四函数用法辩析

先看MA和EMA,首先，它们都是求平均值，这应该没疑问吧;

MA是简单算术平均，MA(C,2)=(C1+C2)/2; MA(C,3)=(C1+C2+C3)/3;不分轻重，平均算;

EMA是指数平滑平均，它真正的指标表达是:

```
当日指数平均值 = 平滑系数 *（当日指数值-昨日指数平均值）+ 昨日指数平均值;
平滑系数=2/（周期单位+1）;
```

由以上指标推导开，得到:

```
EMA(C,N)=2*C/(N+1)+(N-1)/(N+1)*昨天的指数收盘平均值;
仔细看: X=EMA(C,2)=2/3*C+1/3*REF(C,1); EMA(C,3)=2/4*C+2/4*X;
```

所以，它在计算平均值时，考虑了前一日的平均值，平滑系数是定的，它是利用今日的值与前一日的平均值的差，再考虑平滑系数，计算出来的平均值，所以也有叫异同平均的。

因此，这两个平均算法是不同的，主要是对数组中的数据的权重侧重不同。

理解了MA，EMA的含义后，就可以理解其用途了，简单的说，当要比较数值与均价的关系时，用MA就可以了，而要比较均价的趋势快慢时，用EMA更稳定；有时，在均价值不重要时，也用EMA来平滑和美观曲线。

理解了MA和EMA的含义和用途后，后面几个函数就好理解了；

因为EMA的平滑系数是定的，=2/（周期+1）；如果要改变平滑系数咋办？这就用到了SMA；

SMA(C,N,M)与EMA的区别就是增加了权重参数M，也就是用M代替EMA平滑系数中的2,这样我们可以根据需要调整当日数值在均价中的权重=M/N。(要求N>M）；

大家注意，权重系数在EMA与SMA中都是用数值与周期计算出来的小数，假设有一个小数可以直接代表权重，如何办？这就有了DMA；

DMA(C,A) 中A为权重值，指标如下：X=DMA(C,A)=A*X+(1-A)*X(A小于1），可以发现，DMA与SMA原理是一至的，只是用一个小数直接代替了M/N;

而在实用中，这个小数最有价值的就是换手率=V/CAPITAL;DMA(C,V/CAPITAL)的直接含义是用换手率作为权重系数，利用当日收盘价在均价中的比重计算均价；

直观理解就是换手率越大，当日收盘价在均价中的作用越大！

这样理解应该知道各函数的作用和用途了！

5.4.2 双均线金叉和死叉

由时间短的均线（如上图蓝色的线）在下方向上穿越时间长一点的均线（如上图黄色的线），为“金叉”，反之为“死叉”。

MACD称为指数平滑移动平均线，是从双指数移动平均线发展而来的，由快的指数移动平均线（EMA12）减去慢的指数移动平均线（EMA26）得到快线DIF，再用2×（快线DIF-DIF的9日加权移动均线DEA）得到MACD柱。MACD的意义和双移动平均线基本相同，即由快、慢均线的离散、聚合表征当前的多空状态和股价可能的发展变化趋势，但阅读起来更方便。当MACD从负数转向正数，是买的信号。当MACD从正数转向负数，是卖的信号。当MACD以大角度变化，表示快的移动平均线和慢的移动平均线的差距非常迅速的拉开，代表了一个市场大趋势的转变。

MACD在应用上应先行计算出快速（一般选12日）移动平均值与慢速（一般选26日）移动平均值。以这两个数值作为测量两者（快速与慢速线）间的“差离值”依据。所谓“差离值”（DIF），即12日EMA数值减去26日EMA数值。因此，在持续的涨势中，12日EMA在26日EMA之上。其间的正差离值（+DIF）会愈来愈大。反之在跌势中，差离值可能变负（-DIF），也愈来愈大。至于行情开始回转，正或负差离值要缩小到一定的程度，才真正是行情反转的信号。MACD的反转信号界定为“差离值”的9日移动平均值（9日EMA）。在MACD的指数平滑移动平均线计算公式中，都分别加T+1交易日的份量权值，以现在流行的参数12和26为例，其公式如下

- 12日EMA的计算:

```
EMA（12） = 前一日EMA（12） X 11/13 + 今日收盘价 X 2/13
26日EMA的计算:
EMA（26） = 前一日EMA（26） X 25/27 + 今日收盘价 X 2/27
```

- 差离值（DIF）的计算:

```
DIF = EMA（12） - EMA（26） 。
根据差离值计算其9日的EMA，即离差平均值，是所求的DEA值。为了不与指标原名相混淆，此值又
名DEA或DEM。今日DEA = （前一日DEA X 8/10 + 今日DIF X 2/10）
```

- 用（DIF-DEA）*2即为MACD柱状图。

故MACD指标是由两线一柱组合起来形成，快速线为DIF，慢速线为DEA，柱状图为MACD。在各类投资中，有以下方法

1. 当DIF和MACD均大于0(即在图形上表示为它们处于零线以上)并向上移动时，一般表示为行情处于多头行情中，可以买入开仓或多头持仓;
2. 当DIF和MACD均小于0(即在图形上表示为它们处于零线以下)并向下移动时，一般表示为行情处于空头行情中，可以卖出开仓或观望。
3. 当DIF和MACD均大于0(即在图形上表示为它们处于零线以上)但都向下移动时，一般表示为行情处于下跌阶段，可以卖出开仓和观望;
4. 当DIF和MACD均小于0时(即在图形上表示为它们处于零线以下)但向上移动时，一般表示为行情即将上涨，股票将上涨，可以买入开仓或多头持仓。

5.5 多因子

5.5.1 因子

因子是什么？通俗来讲。选股择时，我们得有一个标准对不对？这些标准就叫做因子。比如，我认为营收增长率高的公司就是好公司！那我就把营收增长率大于30%的股票拉出来纳入石榴裙下好了。这个营收增长率大于30%就是因子，完毕。

因子有选股的因子（股票好不好），有择时的因子（好股票什么时候买）。由于择时往往跟技术指标关系紧密，本篇中就介绍基本面类的因子吧，偏财务向。

5.5.2 选取因子

最简单的方法，先物色一些自己喜欢的因子，比如增长率啦，市值啦，ROE啦，等等。然后一个个往里面加，看看效果如何，效果好了留下，效果差了删除，反复重复这个过程就能找到心仪的因子啦。

5.6 海龟交易法则

天然的海龟是一个比较成熟而完整的交易系统。构建交易系统的目的就是避免交易员自己做出主观的决策。这样才能真正的让概率发挥作用。海龟的主要捕捉的是趋势。其采用突破法来确定趋势，当价格突破时认为有买入的信号，而随着价格离当初突破的价格越来越远，我们认为趋势成立的概率就越来越高，加仓！那么，这个突破怎么确定呢？我们需要用到唐奇安通道的方法进行处理。 唐奇安通道

在海龟的系统的止盈止损中，实际上就是借助了唐奇安通道。那么，这唐奇安通道究竟是个什么东西呢?

首先引入上线中线下线的概念。上线=Max（前N个交易日的最高价），下线=Min（前N个交易日的最低价），中线=（上线+下线）/2，每个交易日结束之后更新当天的数据。这里N一般默认取20。那么唐奇安通道就是这个上线和下线所形成的走势区间。所谓的突破，也就是指今日盘中股价高过了上线。

止损

有了唐奇安通道，我们有了买入和卖出的依据了，那么止损是干什么的呢？其实止损提出的初衷是，如果某笔交易是亏损的交易，造成的损失不要超过总仓位的k%。在完整的海龟系统里面，海龟用了一系列的公式进行仓位比例的计算

5.7 APT

CHAPTER 6

瞎想

Contents:

6.1 德州扑克

玩了好久的德州扑克，但是水平还是一般，输输赢赢的。说是好久其实就是半年的样子，但是其中的心路历程与这一年三月份开始的入市炒股票的经历相似，且最近思考得出的结论与人生道理暗自契合，因此暂做总结。

开始玩德州扑克是朋友的推荐，是抱着与斗地主稍微有一点点儿不同的心态去玩的，对规则略熟悉之后基于新奇和其与股票有类似的地发，有一段时间玩的比较多。其实当时也没具体去想具体的关系，现在想来，该是德州的几次压注与股票的仓位控制的关联。当时想着玩德州的一些经验能够作用于股票操作上，且有股社区的推荐文章，渐渐熟悉了一下规则，比如开始两张牌比较大时应该积极加注减少牌局的参加人数以增加赢的几率。到这一阶段德州里的金币确实有所增长，但是依然波动较大。

渐渐一段时间之后（其实经历了几次的大起大落之后），渐渐对此游戏的兴趣有所退化，玩的越来越少，感觉有些浪费时间。

终于有一次下定决心，输光里边所有的金币然后卸载软件。于是每次都加注，气势压倒别的玩家。但是这样误打误撞发现这样是增加了波动的范围，却并不是会很快输掉所有金币。当然完全不看牌局直接 ALLIN 还是会输掉所有金币。

于是就去尝试适当加大注的玩法，渐渐摸索出一些门道，并得出当前的结论：**我们不可能也没必要把握住每次的机会，但是可以把握住几次大的机会，只要把握住这几此机会也就把握住了大的方向**。

6.2 思维方式的改变是一件细思恐级的事

思维方式的改变是潜移默化的发生的，在你意识到改变前已经发生了。之所以恐怖是因为，你无法确切知道它是往何方向改变，有时候甚至会与你以前的思维方式相冲突。 而这种改变是说不上好坏的。当然没有无缘无故的改变，有果必有因，这些改变肯定与你的经历有关，是你的经历循序渐进的改变了你。

韩寒的电影《后会无期》中那句"小孩才分对错，大人只看利弊"，谁能说的清是好是坏呢。也许只是往有利的方向发展了，但是何尝不是带着一种悲凉呢。 孩童时期的那份单纯天真也许是无法适应这个复杂的社会的，所以才会随着孩子的长大慢慢变弱直至消失吧。

经历与思维方式改变之间的关系就像先有鸡还是先有蛋的问题一样难解，两者互相关联互为因果，说不清是谁改变了谁，也许在这两者之外还有第三种力量存在吧，它把握了我们的方向。

举目望日，不见长安。

6.3 春节红包总结

今年春节因了支付宝的敬业福以及微信的影响，到处一片抢红包的繁荣气象。除夕晚本是万家团员吃饭聊天的好时候， 现实中的场景却是未见聊天低头，都在低着头刷微信红包，玩支付宝的咻一咻。对于我来说，今年也没有看春晚的心情了， 对春晚少了很多期待把。

除夕晚上，突发奇想，做一个收发红包的小统计调查吧。方法如下：

- 时间以除夕到初一为界；
- 选择疏落的微信好友，首先向该好友发送一条聊天消息 / 表情，作为开始；
- 若收到回复，向该好友发送金额不等的拜年红包；
- 若好友互动则继续聊家常；

作出以下统计：

- 发出初始互动消息数量：60
- 发出红包的份数：37
- 收到红包的份数：17
- 发出红包金额大于收到红包金额的份数：8
- 收到红包金额大于发出红包金额的份数：7

分析如下：

- 未回复消息数量：23，比预想中的数量略大，猜测原因为部分人过年忙着玩 / 与家人聊家常 / 还不习惯微信红包拜年这种方式；
- 收到红包数量：17，红包收发比为 17/37，比预想中低，原因不明，可能为部分好友平时联系较多 / 习惯了收红包，但并不习惯红包拜年这种方式；
- 收到红包金额大小比：8/7，与预期吻合，微信拜年红包设计就是金额随机。

比较有意思的一些事：

- 腾讯的服务器可能出现了问题，部分在聊天中已经领取过的红包，在统计时显示为该红包超时未领取，不知道真实情况到底领没领到……
- 极少数红包为在未收到回复时发送，极极少数的情况时红包超过领取时间后收到了回复……

第一次做这样的统计分析，想想还真的有点儿小激动，以上统计与分析纯属娱乐，如能帮到你纯属巧合。

最后感谢所有那些回复 / 聊天及发红包给我的朋友们：)

6.4 天津见闻奇异录

今天下午请假帮朋友去天津办事，心里想着反正请了半天假，索性事情完了以后在天津转转，缓解一下最近疲惫的心神。 事情办的倒是很顺利，但是办完还是已经下午四点半了，即使马上回来也不算太早，爱玩的我还是禁不住诱惑转到晚上八点多才返回。 虽然时间不长，还是有几件有趣的事值得记录下来。

遗憾的是手机几乎没电，基本上没拍照片。

- 目测天津街上蹬自行车（非电动车）的多不少，而且自行车比电动车的比例也不低。 北京的自行车也很少（除了那些玩骑行的~），而在我们老家农村，自行车基本上被电动车淘汰..... 而天津人蹬自行车，往往越是路口蹬的越快，和汽车抢道抢速度，反而是普通路面蹬的随意了。
- 去逛南开，意外的发现了几个练武术的人（应该是老师和学生），而且还是在练形意拳。 比较有趣的是有个老师在教兵器（应该是剑），用的却是很普通普遍的木棍，着实有影视作品中侠之大者的风范。 算上上次去南京在玄武湖旁边遇到练形意拳的姑娘，最近半年去过的两个地方都遇到了练形意拳的。 由此推断，目前武术没有我们平时说的那么不堪，练习的人还是挺多的；形意拳却是算得上比较大的拳种了。
- 之前一直知道北师大和北邮之间只隔了一条很窄的马路，西门对东门，是很近的了。逛了南开跟天大才发现，两所学校仅一楼之隔。 而且隔开两所学校的楼应该是两所学校共用的，这边写着南开，那边写着天大。 但是感受到的两所学校的气质却完全不一样，天大相对新气，南开则更富底蕴，距离之近差别之大十分惊奇，个人感受而已。
- 吃到了正宗的清真菜，并不是说味道正宗，而是对清真的严格执行。 逛完南开天大又转了一圈水上公园，走的实在累了，想找个小馆子吃个盖饭来瓶啤酒，无奈公园边转了半天找不到，走了大老远才找到一家清真拉面馆。 心想拉面馆应该会有盖饭跟啤酒啊（在北京习惯了~），但是点完盖饭之后却没找打啤酒，点完山海关（类似北冰洋的饮料）才发现后厨的门口写着清真菜馆严禁外带食品，严禁喝酒。 在北京好像还没吃到过禁酒的菜馆，当真是馆子虽小，但人家开的认真啊。
- 天津地铁单次卡是小圆盘，背面写的是“国有资产，用后回收”，国有资产.....

在水上公园转的时候应该是看到天津之眼了，不过没开灯~

京津虽近，差别却很大，很有趣。

CHAPTER 7

关于我

大家好，欢迎访问我的个人网站。

CHAPTER 8

联系

GitHub 主页

微信：lennyh

Email：hhhhhf@foxmail.com

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python 模块索引

C

索引

符号

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X