

# Uter 用户指南

| 作者          | 日期         | 内容 |
|-------------|------------|----|
| YAOZHONGKAI | 2016.04.24 | 初稿 |
|             |            |    |
|             |            |    |

# 内容目录

- 1 介绍.....4
  - 1.1 关于单元测试.....4
  - 1.2 Uter 单元测试软件.....4
- 2 安装.....5
  - 2.1 上位机安装.....5
  - 2.2 目标机安装.....5
- 3 配置.....6
  - 3.1 编译.....6
  - 3.2 配置.....6
  - 3.3 启动.....9
- 4 操作入门.....10
  - 4.1 Python 控制台.....10
  - 4.2 自动化测试.....11
  - 4.3 覆盖率.....12
- 5 数据操作.....13
  - 5.1 数据类型与数据对象.....13
  - 5.2 基本类型.....13
  - 5.3 枚举类型.....13
  - 5.4 结构体类型.....13
  - 5.5 共用体类型.....13
  - 5.6 数组类型.....13
  - 5.7 指针类型.....13
  - 5.8 VOID 类型.....13
- 6 内存与字符串.....13
  - 6.1 数据类型转换.....13
  - 6.2 内存分配与释放.....13
  - 6.3 字符串操作.....13
- 7 对象导出.....13
  - 7.1 Range 对象.....13
  - 7.2 数据类型导出.....13
  - 7.3 变量导出.....13
  - 7.4 函数导出.....13
- 8 宏操作.....14
- 9 函数操作.....14
  - 9.1 调用.....14
  - 9.2 打桩.....14
- 10 目标操作.....14
  - 10.1 字节序.....14
  - 10.2 桩管理.....14
  - 10.3 Range.....14
- 11 脚本端覆盖率.....14
- 12 自动测试.....14
  - 12.1 编写测试用例.....14

|                       |    |
|-----------------------|----|
| 12.2 批量测试.....        | 14 |
| 13 UI 工具.....         | 14 |
| 13.1 程序浏览器.....       | 14 |
| 13.2 程序分析器.....       | 14 |
| 13.3 UI 端覆盖率.....     | 14 |
| 14 附录：术语.....         | 15 |
| 15 附录：Python 入门.....  | 15 |
| 15.1 Yyyyyyyyyyy..... | 15 |
| 15.2 Yyyyyyyyyyy..... | 15 |

# 1 介绍

## 1.1 关于单元测试

在 C/C++ 软件开发过程中，单元测试是一个重要的质量保障步骤，有效的单元测试可以发现大量的编码 Bug。不过目前的单元测试方案存在以下问题：

- 需要集成测试框架和测试软件：需要将 CUnit 与被测试软件集成，编写测试用例文件并且注册测试用例，在软件启动的时候，对测试框架进行调用，然后将这些代码一同编译构建，运行测试。在单元测试工作完成以后，通常单元测试相关的测试框架和测试例代码都不会被保留而只提交代码，这样在代码维护过程如果需要进行单元测试回归验证，则需要重新集成框架和测试例代码，非常繁琐，耗时耗力，性价比差。在使用旧代码开发新版本的时候，不仅需要重新集成框架和测试代码，同时也会因为测试例代码长期与被测试代码分离而失去维护，导致开发新版本的时候旧版本的测试例代码难以使用。

- 编写测试例代价高：使用传统的单元测试系统，编写测试例需要经过测试例代码编写，编译链接，有的还需要进行部署。由于大型项目编译部署一次耗时非常长，每次编译部署少量的测试例新增和修改性价比会非常差，因此在这种情况下通常会采用一次性写多个测试例然后编译部署，使得平均在每个测试例上的编译和部署时间会变少。不过，这样做的代价是一次性写关于多个测试点的测试用例子，编译部署以后又回过头来检查多个测试点的测试效果，对于单个测试点，未能在写测试用例和观察测试效果之间形成反馈，不符合人的思考习惯，未能有效的利用测试人员的精力。理想的测试方案应该能够思考代码逻辑，参考测试点(可以在制定测试计划的时候已经定义完成)，书写测试代码，观察测试结果，通过测试结果再思考代码逻辑，调整测试点，修改测试代码，再测试。。。直到达到预期的测试效果。不过目前一次性写大量测试用例再一次性测试观察的方法是对测试思路的一种破坏，同时如果在测试过程中，需要修改和新增测试用例，如果修改量小，则性价比低，积攒在一起编译部署不会立即观察到新增和修改测试例的效果。总之目前的方案在时间消耗和测试思路之间存在严重的矛盾。

- 对函数进行打桩困难：为了对软件代码进行充分的测试，如果被测试软件调用了其他函数，为了充分测试其他函数的返回值和出参相关的代码逻辑，就需要对被调用的函数进行打桩，以为如果不对其进行打桩，该函数未必能够如预期的返回和设置出参，而无法验证相关的逻辑(比如总是返回正确，而导致调用后的错误处理逻辑难以被验证)。然而传统的测试框架并没有提供对函数进行打桩的方案，因为这些框架的原理都是管理着对被测试代码的调用。如果需要对函数进行打桩，则需要借助宏进行名称替换，必须对源文件进行修改，如果需要对被调用的函数打不同的桩，则需要写很多代码进行辅助支持，性价比很差。同时，对代码的打桩也存在编译部署时间长的问题。

- 收集覆盖率信息困难：覆盖程度是单元测试的一个重要指标，不过传统的单元测试本身也并不支持覆盖率统计，需要借助 GCOV 来统计覆盖率信息。使用 GCOV 必须要等到软件运行结束才能查看覆盖率信息，测试人员无法通过实时的获得覆盖信息，用于在测试过程中及时的增加测试例。

传统的 C/C++ 单元测试方案包含了太多的工作和耗时过程，而这些消耗并没有有效的作用与单元测试的测试对象--被测试代码和逻辑。从而造成了单元测试工作投入高产出的错误印象，经常出现在软件开发过程中裁剪单元测试环节的状况，导致软件项目质量不高。因此一款开发效率高，易于使用的单元测试工具是很有必要的。

## 1.2 Uter 单元测试软件

Uter 是一款 C 语言单元测试工具，提供了 C 语言单元测试解决方案，采用脚本方式方式对被测试软件

进行测试，开发效率高。Uter 的特性如下：

- 无需集成：Uter 并非是一个测试软件框架，因此不需要集成。Uter 在本质上是一个调试器，因此被测软件只需要被编译成调试版本，Uter 就可以对其进行测试。这给单元测试工作带来了极大的便利，在编码-编译-Reivew 之后编译成调试版本就可以直接开始单元测试。在软件维护的过程中，如果因为代码修订需要重新进行单元测试(回归验证)，只需要将软件编译成调试版本，然后就可以立即开始单元测试。由于单元测试工作启动代价小，因此可以让单元测试工作伴随整个开发过程：编码完成后可立即进行单元测试，维护过程中修改代码可以进行单元测试回归，使用代码开发新版本的时候可以利用原有测试用例。

- 编写测试用例容易：Uter 的测试用例采用 Python 脚本，新增和修改测试用例都是在修改 Python 脚本，不需要重新编译部署被测软件。测试人员可以思考代码逻辑，新增或者修改测试用例，执行测试用例，根据执行效果再思考代码逻辑，调整测试脚本。。。测试人员可以始终专注与代码逻辑。由于 Uter 支持测试，因此可以在测试脚本的驱动下，对被测试软件进行跟踪调试，进一步暴露代码中存在的问题。

- 使用脚本进行打桩：Uter 支持使用脚本对被测试软件调用的函数进行打桩，在测试过程中调用到被打桩函数的时候，会自动调用对应的 Python 桩函数，Python 桩函数可以对出参作出修改和返回任意值，用于充分的测试被测函数。并且桩函数可以被自由设置和清除，在不同的测试例中使用不同的桩函数，可以将被测函数中所有与桩返回值和出参有关的条件。

- 支持实时覆盖率统计：Uter 支持两种覆盖率统计方式，一种为脚本端覆盖率信息，在完成对一个函数的测试以后，通过脚本接口判断该函数是否被完整测试，如果没有被充分测试，则测试不通过，该方式可以在维护修订并自动单元测试回归的过程中及时的发现未被完整覆盖的函数。另外一种为 UI 端的覆盖率信息，测试人员可以执行完测试用例以后，实时的查看代码的覆盖情况，发现未覆盖的代码，并且增加新测试用例以达到完全的覆盖。

采用调试方式和 Python 脚本的 Uter 软件，避免了传统单元测试方案的开发效率问题，可以使测试人员在单元专注与被测试代码的逻辑，同时轻松的书写 Python 测试脚本，在 Python 脚本的启动下跟踪调试被测代码，通过 UI 段覆盖率的显示及时发现未被覆盖的代码，及时补充测试例。

## 2 安装

### 2.1 上位机安装

略

### 2.2 目标机安装

略

## 3 配置

### 3.1 编译

Uter 支持 Linux 操作系统下由 gcc 编译的 C 语言软件，被测试代码需要编译成 Debug 版本，本测试的软件称为目标软件，下文同，在 gcc 命令中使用 -g 参数。如下例：

直接编译:  
gcc -c -g file\_1.c -o file\_1.o  
gcc -c -g file\_2.c -o file\_2.o  
gcc file\_1.o file\_2.o -o target.bin

Makefile 方式:  
CC = gcc  
C\_FLAGS = -g  
TARGET = target.bin  
OBJS = file\_1.o file\_2.o  
\$(OBJS):%.o:%.c  
    \${CC} \${C\_FLAGS} -c \$< -o \$@  
\${TARGET}:\${OBJS}  
    \${CC} \${OBJS} -o \$@  
.PHONY:all  
all:\${TARGET}

如果未能使用 -g 选项编译被测试软件，则无法进行正常的单元测试。编译完成后的测试期间，编译目标软件所使用的源文件不可以被修改，如有修改将会导致调试和代码显示功能异常，如果已经发生了修改，则需要重新编译，重新启动目标程序。

### 3.2 配置

对目标软件进行测试前，需要先在 Uter 进行配置，主要包括如下步骤：

- 创建项目：

在开始测试前，先创建一个测试项目，保存测试相关的参数和文件。

菜单: Project => New project



插图 1: New project

Folder：项目文件的路径

File：项目文件的名字(名字需要以.xml 结尾)

OK：确定。

- 添加测试目标：

测试目标是关于被测试软件的运行位置和启动方式的配置。一个项目中可以配置多个测试目标，比如分别对应着本地和远程目标等。

在 Project 栏中的 Project => Targets 上右键 New Target

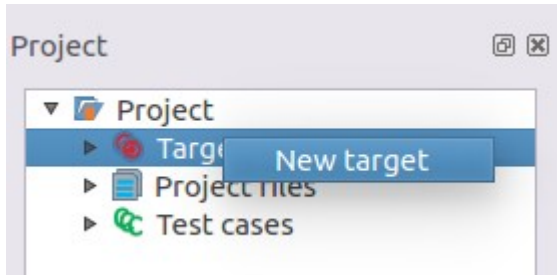


插图 2: New target

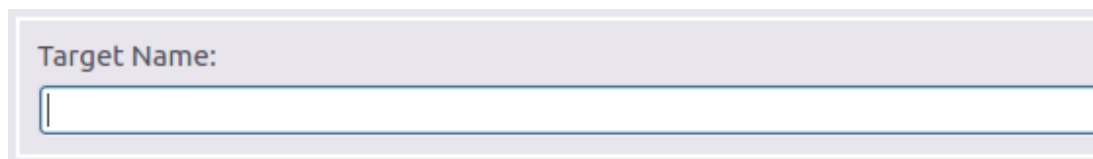
A form with a label 'Target Name:' followed by a single-line text input field.

插图 3: Target name

Target Name：填写测试目标的名称

Connection type：


A form with a label 'Connection type:' followed by a dropdown menu currently showing 'Local'.

插图 4: Local connection

- ◆ Local：调试本地的目标程序。

A form with a label 'Connection type:' followed by a dropdown menu showing 'Tcp'. Below this are two input fields: 'IP:' and 'Port:'.

插图 5: Remote connection

- ◆ Tcp：略

Start type：

- ◆ New：

插图 6: Start new

Start arguments：启动参数（以空格分隔）

Start environment：环境变量（每个变量以 key=value 方式，变量之间以;分隔）

Append native environment：设置的环境变量引用在现有的环境变量集合中

Replace native environment：设置的环境变量替换现有的环境变量集合

#### ◆ Attach：

略

#### ● 添加项目文件：

项目文件是被编译到目标程序中的项目文件，以后将会对这些文件统计 UI 覆盖率和生成测试报告等操作，在开发项目中，需要测试的文件以项目文件的方式加入到项目中。

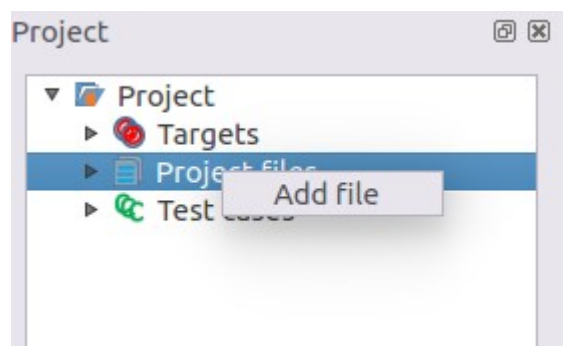


插图 7: Add project file

浏览到项目文件并添加到项目

#### ● 添加测试例目录：



在执行自动化测试过程中，测试例文件需要保持在指定的测试例目录下，这样执行全部测试例的时候才能找到这些测试例文件。

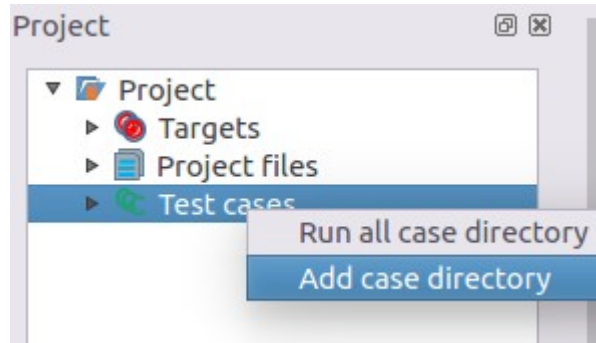


插图 8: Add case directory

### 3.3 启动

在开始测试和调试之前，首先需要选择测试目标，并将测试目标选中。

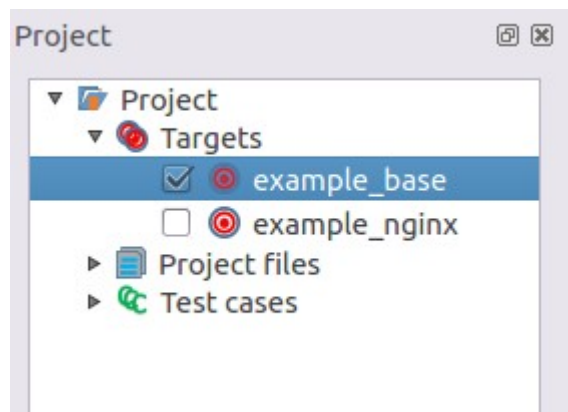


插图 9: Select target



插图 10: Connect

连接成功以后，再使用 Toolbar 中的 Start 按钮启动目标程序。



插图 11: Start

连接和启动过程中，在 Python 控制台中都会有相关的提示信息。

## 4 操作入门

### 4.1 Python 控制台

Uter 采用 Python 作为测试脚本，可以使用保存在测试例目录中的 Python 脚本文件测试，也可以在直接使用 Python 控制台进行直接交互。使用 Python 控制台可以在设计测试例的时候先验证测试方案的有效性，验证以后再编写测试例子文件运行，也可以在诊断测试错误的时候使用。

Python 语言是一门简洁易用的语言，不过与 C/C++ 语言有很大差异，为了让 C/C++ 开发人员快速掌握 Python 语言，本手册附录中提供了 Python 入门指引。

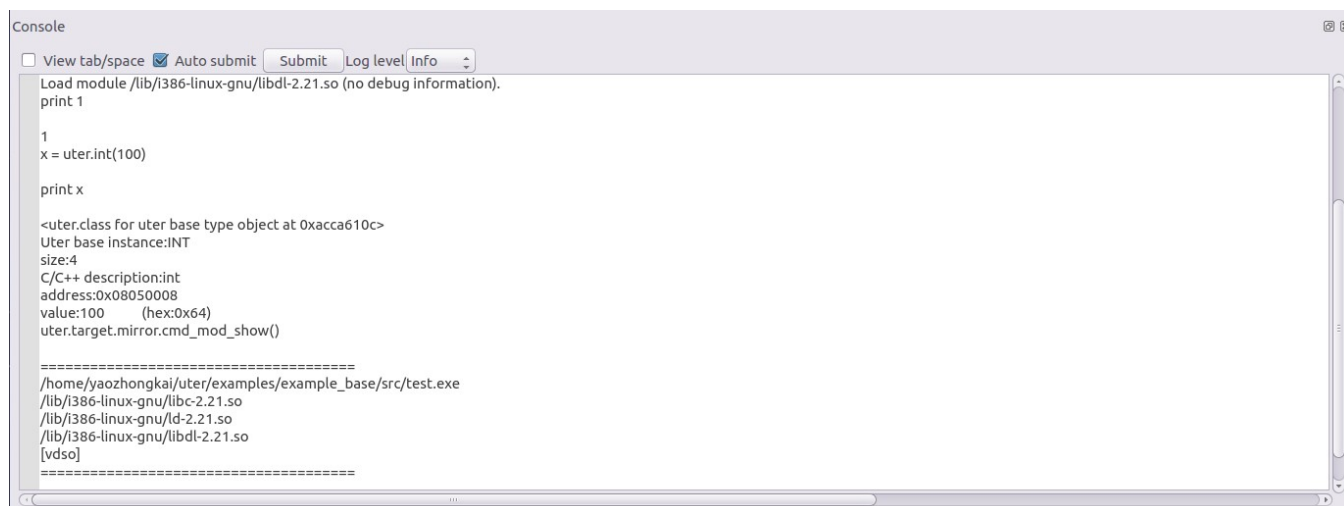


插图 12: Python 控制台

- Uter 中的 Python 版本为 Python-2.7.9。
- Python 语言条件语句或者函数等使用 TAB 键或者空格键排版，并且作为语法的一部分。Python 控制台提供了 View tab/space 选项（默认不被选中），选项被选中以后，在每代码行前方起语法作用的的 TAB 键和空格键会被用颜色标识，以方便编码使用。
- 在 Python 控制台中输入有效的 Python 语句并回车以后，默认情况下会被自动提交执行，比如：

以下 Python 语句会在输入回车后自动提交：

```
print 100
```

一下语句会在末尾的空行输入回车后自动提交，否则就无法完整的输入整个函数了：

```
def test_fun(x,y):  
    print x  
    print y
```

Python 控制台提供的 Auto submit 选项（默认被选中），选项被取消选中以后，输入的 Python 语句将不会被自动提交，始终处于被编辑状态，直到使用 Submit 按钮提交。

- Python 控制台同时也是 Uter 软件的 Log 和调试信息的输出控制台，Log level 选项提供了输出 Log 的级别(通常不需要对 Log level 进行设置)。

## 4.2 自动化测试

### 4.2.1 编写测试用例

使用 Python 控制台所执行的测试并不能被永久保存，需要将测试过程永久保存需要将测试脚本写入到测试例文件中，测试例文件需要保存在测试例目录中(测试例目录需要被添加到项目中)。向测试例目录中增加新测试例文件：

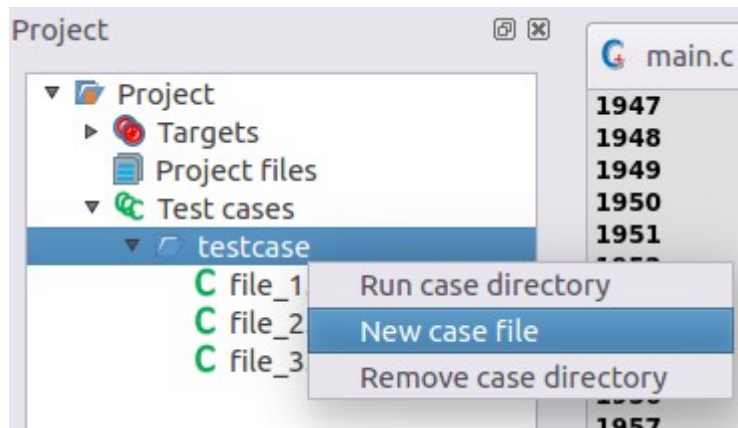


插图 13: New case file

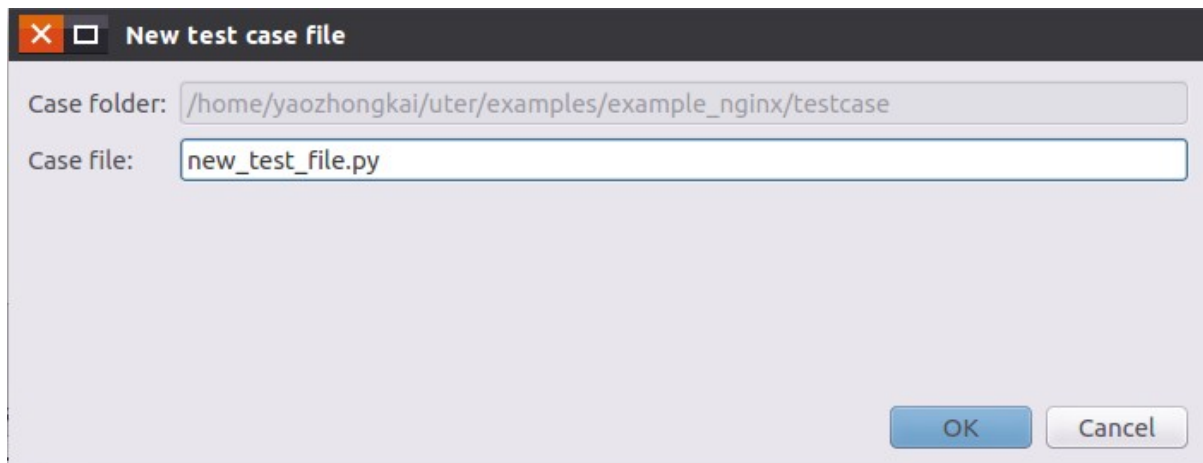


插图 14: New case file

Uter 的测试例需要按照 PyUnit 规范编写，如下例：

```
#coding=utf-8

import uter
import unittest

class test_case_class(unittest.TestCase):
    @classmethod
```

```

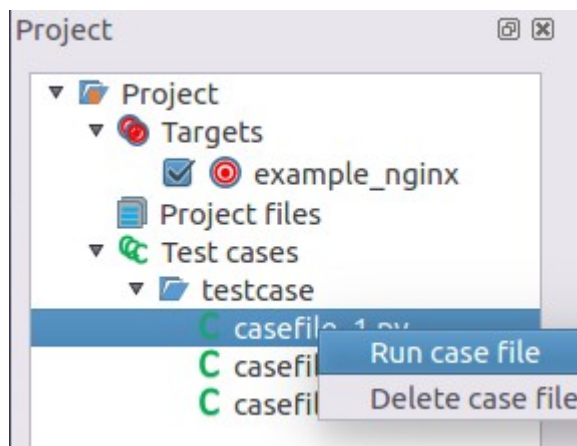
def setUpClass(cls):
    fun_obj = uter.export.function("fun_coverage_base")
    cls.cov = uter.coverage(fun_obj)
    cls.cov.on()
    @classmethod
def tearDownClass(cls):
    covs,sum = cls.cov.count()
    cls.cov.off()
    if covs != sum:
        raise "Coverage not enough"
def test_base(self):
    fun_obj = uter.export.function("fun_coverage_base")
    ret_obj = fun_obj._Call()
    self.assertEqual(True, ret_obj._GetValue() == 0, "")
    self.assertEqual(True, ret_obj._GetSize() == 4, "")

```

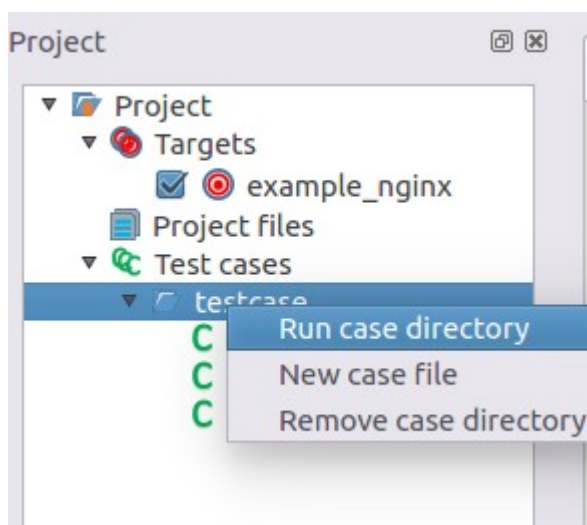
- #coding=utf-8：用于声明测试例文本文件所使用的编码（使用 Python 注释的形式，这个注释不是普通的注释，有提示文件编码的作用），推荐使用 utf-8。
- import uter：使用 Uter 提供的 uter 模块中的功能进行测试，因此需要引入 uter 模块。
- import unittest：使用了 PyUnit 提供的 unittest 模块进行测试，因此需要引入 unittest 模块。
- class test\_case\_class(unittest.TestCase)：定义了测试类，测试类需要继承自 unittest.TestCase，类中函数需要按照 PyUnit 规范命名。
- def setUpClass(cls)：该函数前边必须带@classmethod 注解，以表明该函数是一个类方法，不针对类各个实例。该方法只能包含一个参数（按照 PyUnit 规范只能有一个参数，如果是普通的类函数，可以包含至少一个参数，第一个参数表示类本身），一般使用 cls，cls 代表这个类，通过 cls 可以给类设置/清除/修改/查询属性，对类属性的修改将影响到类的所有实例。setUpClass 会在该类中的各个的是方法开始测试之前被自动调用，一般用于测试准备工作，比如打开对特定函数的覆盖统计，或者设定一个 Range 对象供所有测试方法使用等等。
- def tearDownClass(cls)：该函数同 setUpClass 一样必须带@classmethod 注解，是一个类方法，只能包含一个一般叫 cls 的参数，该函数在该类中各个测试方法测试完只有被调用，一般用于做清理工作。
- def test\_base(self)：测试类以 test 开头的函数类被认为是测试函数，用于完成一个测试例，一个测试点可以有多个以 test 开头的测试函数来测试不同的条件或者场景。测试函数普通的类方法，只能包含一个参数（按照 PyUnit 规范只能有一个参数，如果是普通的方法，可以包含至少一个参数，第一个参数表示类对象），一般使用 self，self 代表类对象。
- self.assertEqual(True, ret\_obj.\_GetValue() == 0, "")：该调用用于在测试中检查条件，地一个参数为预期值，第二个参数为实际值，最后一个参数为检查失败时候的提示信息。在 PyUnit 中可以一个测试函数可以不调用或者多次调用 self.assertEqual，如果没有调用过 self.assertEqual 且测试函数运行未发现异常，则认为测试通过，如果有多个 self.assertEqual，则所有被调用过的 self.assertEqual 都检查通过该测试函数才算检查通过。
- 测试例文件可以直接在 Uter 提供的编辑窗口中编辑，如果用户习惯与使用其他 IDE，则可以将安装目录下的 python/uter.py 导入到 IDE 中以生成输入提示。

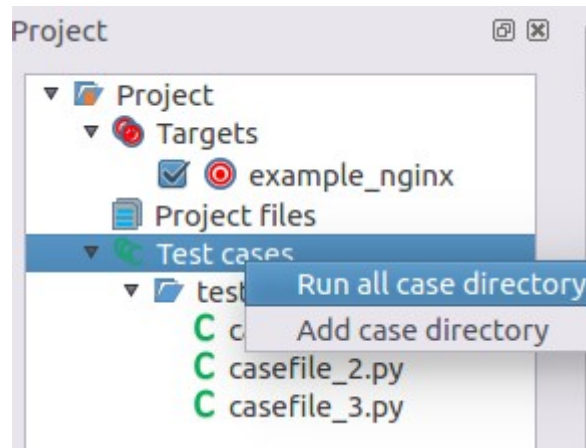
### 4.2.2 执行测试用例

编写完并保存的测试例，可以在测试例文件可以通过右键菜单中的 Run case file 项来运行测试例。如下图所示：



对于整个测试例目录中的测试例文件，可以通过测试例目录邮件菜单中的 Run case directory 项来运行整个目录的测试例，如下图所示：

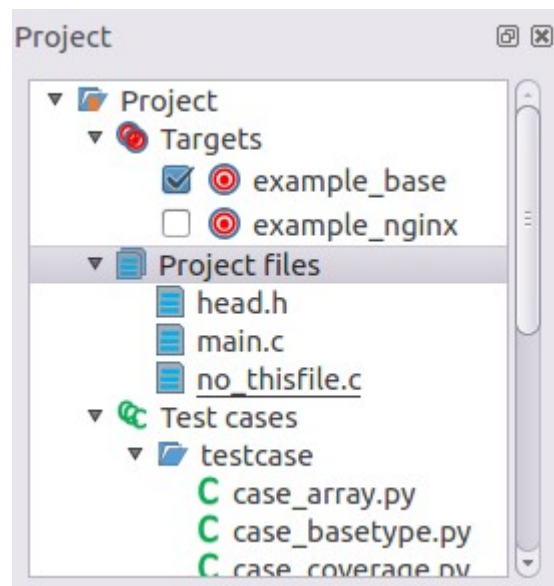


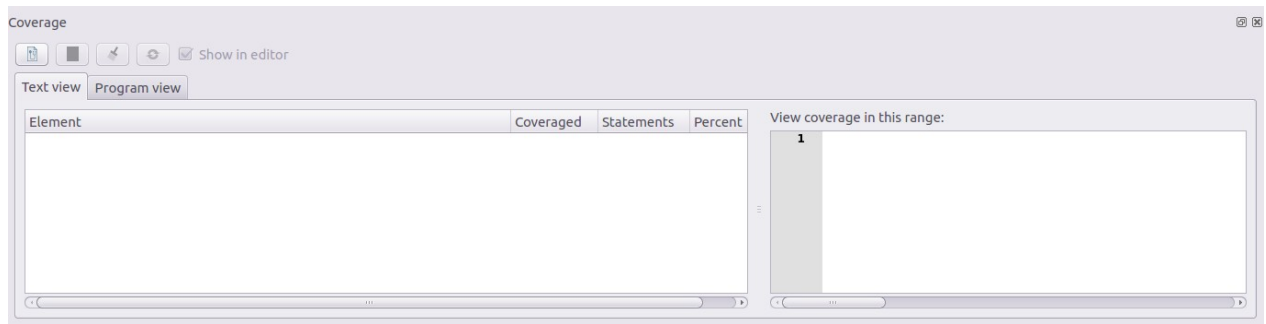


执行测试例的结果可以通过 Python 控制台观察。

### 4.3 覆盖率

Uter 软件支持实时的代码覆盖率统计，在显示文件的覆盖信息之前，需要先将源文件添加为项目文件。如下图（如果被添加的项目文件不存在，或者被移动/删除，文件名下方会出现下划线提示）：



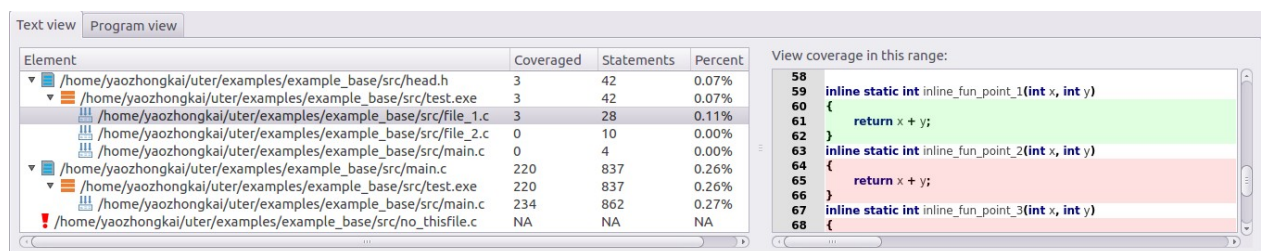


在覆盖率窗口中，主要的控制功能如下：

- ◆ Start coverage：开启覆盖统计功能
- ◆ Stop coverage：关闭覆盖统计功能
- ◆ Clear coverage：清楚已经收集到的统计数据
- ◆ Show coverage：将覆盖统计信息显示在覆盖率窗口，当有覆盖信息被更新以后，需要通过此按钮更新覆盖率窗口和编辑窗体中的覆盖率显示。
- ◆ Show in editor：将覆盖统计信息显示在覆盖率窗口的同时显示在代码编辑窗体（默认选中）。

在覆盖率窗口中，支持两种显示视图，分别能显示不同角度的覆盖数据。

- ◆ Text view：以源文件为统计对象，如果源文件被编译使用多次，则进行多次统计。比如同一个源文件在主程序和动态库中分别被编译，则在主程序和动态库中的覆盖率统计数据将被合并。还比如头文件被多次包含编译（头文件中的数据结构定义不涉及到覆盖率，主要是头文件中的静态函数等可执行成分），则在不同编译单元（在一次编译过程中，被编译的主文件，直接和间接包含的其他文件，编译参数等信息的集合称为编译单元，一般将被编译的主文件路径作为编译单元的标识）中的覆盖率统计数据将被合并。



在 Text view 中会显示各个被有效统计了覆盖率的源文件的信息，如果源文件没有被编译，则统计信息均为 NA。有效统计了覆盖率的文件：

- ◆ 会显示该源文件文件的语句数量和被覆盖的语句数量以及语句覆盖率（如果该文件被多次编译，则显示的数量是合并的结果）。选中该源文件以后，右侧源代码窗体显示合并以后的覆盖信息。
- ◆ 会在该源文件下显示编译了这个源文件的模块（即主程序或者是动态库），并且显示该源文件在这个模块中的语句数量和被覆盖的语句数量以及语句覆盖率（如果该文件在此模块中被多次编译，



则显示的数量是合并的结果)。选中该模块以后，右侧源代码窗体显示合并以后的覆盖信息。

- ◆ 会在模块下显示包含了这个源文件的编译单元，并且显示该源文件在这个编译单元中的语句数量和被覆盖的语句数量以及语句覆盖率。选中该编译单元以后，右侧源代码窗体显示在这个编译单元中的覆盖信息。
- ◆ 统计的覆盖率为语句覆盖率，并非行覆盖率，虽然在普通的代码中两者是一致的，比如如下代码：

```
int iloop;
for (iloop = 0; iloop < 100; iloop++) {
    /*do something*/
}
```

在这段代码中，for 语句所在的行一共有三个语句 `iloop = 0` 和 `iloop < 100` 还有 `iloop++`。在程序执行过程中只要执行到该 for 语句，前两个语句就会被执行，但是如果在第一次循环的时候就执行 break 的话，则最后一个语句 `iloop++` 是没有被运行的。

- ◆ 在代码中显示覆盖信息时，使用绿色显示已经完全语句覆盖的代码行，红色显示没有语句覆盖的代码行，黄色显示部分语句覆盖的代码行，而没有颜色表示该代码行没有被编译，比如注释，数据结构定义，`#if 0` 区间代码等。
- ◆ Program view：以程序元素为统计对象，显示各个模块中，各个编译单元中，各个函数的覆盖情况。被显示的编译单元需要与项目文件匹配，其他编译单元不会被显示，不包含这个编译单元模块也不会被显示。

Text view

Program view

Element

/home/yaozhongkai/uter/examples/example\_base/src/test.exe

/home/yaozhongkai/uter/examples/example\_base/src/main.c

| File  | Function       | Coveraged | Statements | Percent | Line No. |
|---|----------------|-----------|------------|---------|----------|
| /home/yaozhongkai/uter/examples/example_base/src/test.exe | stack_test_5   | 4         | 4          | 1.00%   | 1545     |
|   | stack_so_deep  | 4         | 4          | 1.00%   | 1550     |
|   | debug_loop     | 0         | 5          | 0.00%   | 1556     |
|   | debug_step_01  | 8         | 8          | 1.00%   | 1569     |
|   | debug_step_02  | 8         | 8          | 1.00%   | 1579     |
|   | debug_step_03  | 8         | 8          | 1.00%   | 1589     |
|   | debug_step_04  | 8         | 8          | 1.00%   | 1599     |
|   | debug_step     | 45        | 59         | 0.76%   | 1609     |
|   | debug_datatype | 75        | 75         | 1.00%   | 1716     |
|   | dump_args      | 10        | 10         | 1.00%   | 1890     |

- ◆ 按照程序结构，使用树形结构依次显示 模块，编译单元。
- ◆ 编译单元被选中以后，在右侧显示编译单元内函数的语句数和语句覆盖数和覆盖率。
- ◆ 双击函数的行以后，会自动跳转到 Text view，显示这个函数的覆盖情况（只显示源文件在当前模块中当前编译单元的覆盖信息，不存在覆盖率合并）。

## 5 数据操作

### 5.1 类型与对象

#### 5.1.1 类型与对象

在 Uter 测试中，有两个与数据操作相关的概念，数据类型和数据对象：

- 数据类型：数据类型是一组用于代表 C 语言中数据类型的 Python 类，代表的类型包括基本类型，



结构体，共用体，指针等类型。数据类型的 Python 类用于描述所代表的数据类型，可以使用数据类型创建数据对象（创建对象），也可以使用这个 Python 类来描述其他数据类型，比如用 int，char 的数据类型描述一个结构体类型（定义类型）。

在 Python 语言中 Python 类的类方法区别与普通方法，可以独立与类对象存在，直接通过类名就可以调用（当然在类创建的对象中也可以通过类对象调用这些类方法），比如以下代码：

```
var = uter.int()

print uter.int._GetSize()
4
print var._GetSize()
4

print uter.int._GetAlign()
4
print var._GetAlign()
4

print uter.int
<class 'uter.class for uter base type'>
Uter base type:int
size:4
C/C++ description:int
```

以上的方法是所有数据类型都支持的，另外还有很多具体类型才具有的类方法，比如结构体和共用体类型支持成员操作，指针和数组类型支持所用类型操作，具体可查阅相关的数据类型。

- 数据对象：数据对象是由数据类型对象创建的 Python 对象，每个数据对象在目标程序中都有一个对应的数据内存，比如 int 型数据对象在内存中就会对应一个 4 字节的内存。通过这个 Python 数据对象，可以操作目标程序中对应的内存，比如从 int 数据对象的内存中读取和写入值。数据对象作为 Python 对象，在失去引用以后将会被系统回收，此时数据对象在目标程序中对应内存也将被释放。

Python 对象支持作用与对象的普通方法（当然所属类的类方法也可以在对象中调用），比如以下代码：

```
var = uter.int()

print hex(var._GetAddress())
0x8050008L

print var
<uter.class for uter base type object at 0xacca610c>
Uter base instance:int
size:4
C/C++ description:int
address:0x08050008
value:0 (hex:0x0)
```

不同类型的对象也支持不同的方法，比如结构体和共用体对象支持成员操作，数组类型对象支持下标操作。

## 5.1.2 编程手册

| 函数/成员/方法          | 参数 | 返回值            | 描述        |
|-------------------|----|----------------|-----------|
| cls._GetSize()    | NA | Python int     | 数据类型的长度   |
| cls._GetAlign()   | NA | Python int     | 数据类型的对齐值  |
| cls.__str__()     | NA | Python str     | 数据类型的描述信息 |
| obj._GetAddress() | NA | Python long    | 数据对象的地址   |
| obj._GetType()    | NA | Uter data type | 数据对象的数据类型 |
| obj.__str__()     | NA | Python str     | 数据对象的描述信息 |

## 5.2 基本类型

### 5.2.1 基本类型

Uter 对 C 语言基本数据类型的数据类型类已经提前定义好，用户可以直接使用，具体如下：

| 数据类型               | 变量名            |
|--------------------|----------------|
| char               | uter.char      |
| unsigned char      | uter.uchar     |
| short              | uter.short     |
| unsigned short     | uter.ushort    |
| int                | uter.int       |
| unsigned int       | uter.uint      |
| long               | uter.long      |
| unsigned long      | uter.ulong     |
| long long          | uter.longlong  |
| unsigned long long | uter.ulonglong |
| float              | uter.float     |
| double             | uter.double    |
| void               | uter.void      |

使用上表中的 Python 类来创建基本类型的数据对象，在创建数据对象的时候可以指定初始值，如果过没有指定初始值，则数据对象被设置为 0。比如以下代码：

```
var_1 = uter.char()
var_2 = uter.char(10)

var_3 = uter.uint()
var_4 = uter.uint(100)
```

创建的基本类型数据对象支持设置值和获取值，比如以下代码：

```
var = uter.uint()
```

```

print var._GetValue()
0

var = uter.uint(100)
print var._GetValue()
100

var._SetValue(200)
print var_1._GetValue()
200

```

Uter 支持的各个数据对象的操作方法都是以下单个划线开头单词首字母大写的方式命名，比如\_GetValue 和\_SetValue，这样有别与 Python 中以双下划线开头的私有变量名，也有别与双下划线开头和结尾的系统变量名，比如 \_\_init\_\_ 和 \_\_new\_\_。

基本类型的数据对象支持左移和右移操作，比如以下代码：

```

var = uter.int()
var._SetValue(0x01)
print hex(var._GetValue())
0x1

var._LeftShift(2)
print hex(var._GetValue())
0x4

var._RightShift(1)
print hex(var._GetValue())
0x1

```

## 5.2.2 编程手册

| 函数/成员/方法                 | 参数   | 返回值                        | 描述         |
|--------------------------|--|----------------------------|------------|
| cls.__str__()            | NA   | Python str                 | 数据类型的描述信息  |
| cls.__init__(init_value) | ( 可选 ) init_value : Python int/Python long 对象初始值 | NA                         | 基本数据类型创建对象 |
| obj._GetValue()          | NA   | Python int/<br>Python long | 返回数据对象的值   |
| obj._SetValue(value)     | value : Python int/<br>Python long 需要设定的值        | NA                         | 设置数据对象的值   |
| obj._LeftShift(bits)     | bits : Python int 位移的位数                          | NA                         | 对数据对象的值左移  |
| obj._RightShift(bits)    | bits : Python int 位移的位数                          | NA                         | 对数据对象的值右移  |
| obj.__str__()            | NA   | Python str                 | 数据对象的描述信息  |

## 5.3 枚举类型

### 5.3.1 枚举类型

枚举定义：

枚举类型是一种自定义类型，不同与基本数据类型已经被系统预先定义好，枚举数据类型需要用户自行定义（也可以从目标程序中导出，具体参考后文），定义枚举数据类型采用 `uter.enum` 接口。比如以下代码：

```
datatype = uter.enum(("M1",1), ("M2",2), ("M3",), ("M4",), ("M5",100))

print datatype
<class 'uter.class for uter enum type'>
Uter enum type:(no name)
size:4
C/C++ description:
enum (no name) {
M1 = 1,
M2 = 2,
M3 = 3,
M4 = 4,
M5 = 100,
};
```

`uter.enum` 接口采用可变参数，支持一个或者多个参数，每个参数都是需要是一个 Python tuple（元组），元组的第一个元素是一个 Python str，代表枚举成员的名字，元组的第二个元素是一个 Python int，代表枚举成员的值。元组的第二个元素是可选的，如果省略了第二个元素，则对应的枚举元素的值为默认(从 0 开始或者前一个成员的值增加 1)。

在 Python 中如果元组只有一个元素，则需要这样定义：(元素 1,)，而不是(元素 1)，因为后者会被认为是普通的括号操作，等同于：元素 1。对于需要省略值的枚举成员，Uter 提供了兼容定义方式，比如以下代码：

```
datatype = uter.enum(("M1",1), "M2", "M3", "M4", ("M5",100))

print datatype
<class 'uter.class for uter enum type'>
Uter enum type:(no name)
size:4
C/C++ description:
enum (no name) {
M1 = 1,
M2 = 2,
M3 = 3,
M4 = 4,
M5 = 100,
};
```

枚举数据类型还具有以枚举成员名命名的成员，成员的类型为 Python int。比如以下代码：

```
datatype = uter.enum(("M1",1), ("M2",2), ("M3",), ("M4",), ("M5",100))
```

```
print datatype.M1
1

print datatype.M2
2

print datatype.M3
3

print datatype.M4
4

print datatype.M5
100

print type(datatype.M1)
<type 'int'>
```

使用成员名读取枚举值的方式存在一种特殊情况，就是枚举成员以下划线开头的，由于以下划线开头的成员名在 Python 中属于特殊成员，因此访问以下划线开头的枚举成员值，需要使用 `_Member` 接口，比如以下代码：

```
datatype = uter.enum(("M1",1), ("_M2",2), ("__M3",), ("___M4",), ("M5",100))

print datatype.M1
1

print datatype._M2
Traceback (most recent call last):
  File "<string>", line 1, in <module>
AttributeError: type object 'class for uter enum type' has no attribute '_M2'

print datatype.__M3
Traceback (most recent call last):
  File "<string>", line 1, in <module>
AttributeError: type object 'class for uter enum type' has no attribute '__M3'

print datatype.___M4
Traceback (most recent call last):
  File "<string>", line 1, in <module>
AttributeError: type object 'class for uter enum type' has no attribute '___M4'

print datatype.M5
100

print datatype._Member("M1")
1

print datatype._Member("_M2")
2

print datatype._Member("__M3")
```

```
3
```

```
print datatype._Member("__M4")
```

```
4
```

```
print datatype._Member("M5")
```

```
100
```

### 枚举初始化：

使用枚举数据类型可以创建枚举数据对象，其定义方法与基本数据类型相同，可以省略初始值，也可以指定初始值，初始值可以是任意 Python int 也可以是枚举数据类型的成员，比如以下代码：

```
datatype = uter.enum(("M1",1), ("M2",2), ("M3",), ("M4",), ("M5",100))
```

```
var = datatype()
```

```
print var
```

```
<uter.class for uter enum type object at 0xacc9a90c>
```

```
Uter enum instance:(no name)
```

```
size:4
```

```
C/C++ description:
```

```
enum (no name) {
```

```
M1 = 1,
```

```
M2 = 2,
```

```
M3 = 3,
```

```
M4 = 4,
```

```
M5 = 100,
```

```
};
```

```
address:0x08050028
```

```
value:0 (hex:0x0)
```

```
value math enumerator:
```

```
var = datatype(100)
```

```
<uter.class for uter enum type object at 0xacc9abcc>
```

```
Uter enum instance:(no name)
```

```
size:4
```

```
C/C++ description:
```

```
enum (no name) {
```

```
M1 = 1,
```

```
M2 = 2,
```

```
M3 = 3,
```

```
M4 = 4,
```

```
M5 = 100,
```

```
};
```

```
address:0x08050038
```

```
value:100 (hex:0x64)
```

```
value math enumerator:M5,
```

```
var = datatype(datatype.M2)
```

```
<uter.class for uter enum type object at 0xacc9a98c>
```

```
Uter enum instance:(no name)
```

```
size:4
C/C++ description:
enum (no name) {
M1 = 1,
M2 = 2,
M3 = 3,
M4 = 4,
M5 = 100,
};
address:0x08050048
value:2      (hex:0x2)
value math enumerator:M2,
```

对于枚举数据对象，也支持基本类型数据对象的操作，比如\_SetValue，\_GetValue等，具体参考基本数据类型。

### 5.3.2 编程手册

| 函数/成员/方法                 | 参数  | 返回值                        | 描述                   |
|--------------------------|---|----------------------------|----------------------|
| uter.enum(*args)         | 1 个到多个 Python tuple，tuple 的第一个元素为 Python str 代表成员名字;元组的第二个元素（可选）是一个 Python int，代表成员的值 | Uter data type             | 返回定义的 Python 数据类型    |
| cls._Member(name)        | name：枚举成员名  | Python int                 | 返回枚举成员的值             |
| cls.枚举成员的名               | NA  | Python int                 | 返回枚举成员的值，枚举名不能以下划线开头 |
| cls.__str__()            | NA  | Python str                 | 数据类型的描述信息            |
| cls.__init__(init_value) | （可选）init_value：Python int/<br>Python long 对象初始值                                       | NA                         | 枚举类型创建对象             |
| obj._GetValue()          | NA  | Python int/<br>Python long | 返回数据对象的值             |
| obj._SetValue(value)     | value：Python int/<br>Python long 需要设定的值   | NA                         | 设置数据对象的值             |
| obj.__str__()            | NA  | Python str                 | 数据对象的描述信息            |

## 5.4 结构体类型

### 5.4.1 结构体类型

结构体定义：

结构体类型同样需要被定义使用（也支持导出，具体参考后文）。定义结构体类型使用 uter.struct 接

□，比如以下代码：

```
datatype = uter.struct(("m1", uter.int), ("m2", uter.char), ("m3", uter.char), ("m4", uter.char))

print datatype
<class 'uter.class for uter struct type'>
Uter struct type:(no name)
size:8
C/C++ description:
struct (no name) {
/*00-03*/ int m1;
/*04-04*/ char m2;
/*05-05*/ char m3;
/*06-06*/ char m4; /*07-07*/
};
```

uter.struct 接口使用 0 个或者多个 Python tuple 作为参数，元组用于描述结构体的各个成员，元组的第一个元素须是一个 Python str，代表结构成员的名字，元组的第二个元素是一个 Uter 数据类型，代表结构成员的数据类型。结构成员的数据类型可以是基本类型，或者是结构体共用体等其他合法 Uter 数据类型。比如一下代码：

```
datatypein = uter.struct(("inm1", uter.char), ("inm2", uter.char))
datatype = uter.struct(("m1", uter.int), ("m2", datatypein))

print datatypein
<class 'uter.class for uter struct type'>
Uter struct type:(no name)
size:2
C/C++ description:
struct (no name) {
/*00-00*/ char inm1;
/*01-01*/ char inm2;
};

print datatype
<class 'uter.class for uter struct type'>
Uter struct type:(no name)
size:8
C/C++ description:
struct (no name) {
/*00-03*/ int m1;
/*04-05*/ struct (no name) m2; /*06-07*/
};
```

结构体类型所具有的成员同样支持 \_Member 接口和通过成员名称（不以下划线开头的成员名）方式获得。在结构体类型中获得的成员是结构体成员的类型，比如以下代码：

```
datatypein = uter.struct(("inm1", uter.char), ("inm2", uter.char))
datatype = uter.struct(("m1", uter.int), ("m2", datatypein))

print datatype.m1
<class 'uter.class for uter base type'>
Uter base type:int
```



```
size:4
C/C++ description:int

print datatype.m2
<class 'uter.class for uter struct type'>
Uter struct type:(no name)
size:2
C/C++ description:
struct (no name) {
/*00-00*/ char inm1;
/*01-01*/ char inm2;
};
```

### 结构体初始化：

使用结构体数据类型创建结构体数据对象的时候，可以不指定初始参数，也可以制定初始参数，但如果指定了初始参数，就需要将各个成员的初始参数保存在各自元组中，对号作为结构体数据对象的初始参数。比如以下代码：

```
datatypein = uter.struct(("inm1", uter.char), ("inm2", uter.char))
datatype = uter.struct(("m1", uter.int), ("m2", datatypein))

varin = datatypein()
varin = datatypein(), ()
varin = datatypein((10,), ())
varin = datatypein((10,), (20,))

var = datatype()
var = datatype(), ()
var = datatype((30,), ())
var = datatype((30,), ((10,), (20,)))
```

由于 Python tuple 在只有一个元素的时候，必须在元素的后方加逗号，导致代码可读性下降。因此 Uter 提供了使用 Python list 的代替方案（仅用于对象初始化，Uter 内部会将 list 转换为 tuple），比如以下代码：

```
datatypein = uter.struct(("inm1", uter.char), ("inm2", uter.char))
datatype = uter.struct(("m1", uter.int), ("m2", datatypein))

varin = datatypein()
varin = datatypein([], [])
varin = datatypein([10], [])
varin = datatypein([10], [20])

var = datatype()
var = datatype([], [])
var = datatype([30], [])
var = datatype([30], [[10], [20]])

print var
<uter.class for uter struct type object at 0xa8428eac>
```

```

Uter struct instance:(no name)
size:8
C/C++ description:
struct (no name) {
/*00-03*/ int m1;
/*04-05*/ struct (no name) m2; /*06-07*/
};
address:0x08050148
value:
struct (no name) {
/*00-03*/ int m1; -->30 (hex:0x1e)
/*04-05*/ struct (no name) m2; /*06-07*/
};

```

使用 Python list 替代 Python tuple 以后，代码可读性会大幅度提高。对于一个初始化结构体对象，可以省略所有成员的初始化参数，也可以给每个结构体的成员指定一个列表。如果哪个指定的列表为空，则对应的成员对象采用默认初始化。

结构体对象同样支持对象操作，使用 `_Member` 接口或者不以下划线开始的结构成员名。通过接口或者成员名返回的是结构体成员的数据对象（结构体类型的成员是成员的数据类型，结构体对象的成员是成员的数据对象）。比如以下代码：

```

datatypein = uter.struct(("inm1", uter.char), ("inm2", uter.char))
datatype = uter.struct(("m1", uter.int), ("m2", datatypein))

var = datatype([30], [[10], [20]])

print var.m1
<uter.class for uter base type object at 0xa842888c>
Uter base instance:int
size:4
C/C++ description:int
address:0x08050158
value:30      (hex:0x1e)

print var.m2
<uter.class for uter struct type object at 0xa842852c>
Uter struct instance:(no name)
size:2
C/C++ description:
struct (no name) {
/*00-00*/ char inm1;
/*01-01*/ char inm2;
};
address:0x0805015c
value:
struct (no name) {
/*00-00*/ char inm1; -->10 (hex:0xa)
/*01-01*/ char inm2; -->20 (hex:0x14)
};

```

操作结构体对象的数据，都是对其成员的对象进行操作，比如设置和读取值，如果其成员也是结构体或者共用体，则操作成员的成员，以此类推，知道操作到基本类型成员。

## 5.4.2 编程手册

| 函数/成员/方法            | 参数   | 返回值              | 描述                      |
|---------------------|--|------------------|-------------------------|
| uter.struct(*args)  | 0 个到多个 Python tuple，元组的第一个元素为 Python str 代表成员名字;元组的第二个元素是一个 Uter data type，代表成员的类型。    | Uter data type   | 返回定义的结构体数据类型            |
| cls._Member(name)   | name：Python str 结构成员名  | Uter data type   | 返回结构成员的数据类型             |
| cls.结构成员的名          | NA   | Uter data type   | 返回结构成员的数据类型，成员名不能以下划线开头 |
| cls.__str__()       | NA   | Python str       | 数据类型的描述信息               |
| cls.__init__(*args) | 0 个或者与结构成员数量相等的 Python tuple（也可以是 Python list），tuple 中包含各个成员的参数，tuple 可以为空，则成员会被默认初始化。 | NA               | 结构类型创建对象                |
| obj._Member(name)   | name：结构成员名   | Uter data object | 返回结构成员的数据对象             |
| obj.结构成员的名          | NA   | Uter data object | 返回结构成员的数据对象，成员名不能以下划线开头 |
| obj.__str__()       | NA   | Python str       | 数据对象的描述信息               |

## 5.5 共用体类型

### 5.5.1 共用体类型

共用体定义：

共用体数据类型的定义使用 uter.union 接口，其调用参数与定义结构体数据类型相同，比如以下代码：

```
datatype = uter.union(("m1", uter.int), ("m2", uter.char), ("m3", uter.char))

print datatype
<class 'uter.class for uter union type'>
Uter union type: (no name)
size:4
C/C++ description:
union (no name) {
/*00-03*/ int m1;
/*00-00*/ char m2;
/*00-00*/ char m3;
};
```

定义共用体时也支持使用自定义类型，比如以下代码：

```
datatypein = uter.struct(("inm1", uter.char), ("inm2", uter.char))
```

```
datatype = uter.union(("m1", uter.int), ("m2", datatypein))
```

```
print datatypein
<class 'uter.class for uter struct type'>
Uter struct type:(no name)
size:2
C/C++ description:
struct (no name) {
/*00-00*/ char inm1;
/*01-01*/ char inm2;
};

print datatype
<class 'uter.class for uter union type'>
Uter union type: (no name)
size:4
C/C++ description:
union (no name) {
/*00-03*/ int m1;
/*00-01*/ struct (no name) m2;
};
```

结构体类型所具有的成员同样支持\_Member 接口和通过成员名称（不以下划线开头的成员名）方式获得。在结构体类型中获得的成员是结构体成员的类型，比如以下代码：

```
datatypein = uter.struct(("inm1", uter.char), ("inm2", uter.char))
datatype = uter.union(("m1", uter.int), ("m2", datatypein))

print datatype.m1
<class 'uter.class for uter base type'>
Uter base type:int
size:4
C/C++ description:int

print datatype.m2
<class 'uter.class for uter struct type'>
Uter struct type:(no name)
size:2
C/C++ description:
struct (no name) {
/*00-00*/ char inm1;
/*01-01*/ char inm2;
};
```

### 共用体初始化：

共用体类型创建共用体对象与结构体类型有很大区别，由于共用体的各个成员使用同一块内存，所以在结构体对象初始化的时候，需要指定要初始化成员的名字（Python str 类型），和初始化成员所使用的参数，参数可选，如果忽略成员初始化参数，则共用体对象做默认初始化，如果制定了成员初始化参数，则参数需要 Python tuple 或者 Python list 类型，可以为空 tuple 或者空 list，则使用默认参数初始化成员。比如以下代码：

```
datatypein = uter.struct(("inm1", uter.char), ("inm2", uter.char))
datatype = uter.union(("m1", uter.int), ("m2", datatypein))
```

```
var = datatype()
```

```
var = datatype("m1")
```

```
var = datatype("m1", ())
var = datatype("m1", (100,))
```

```
var = datatype("m1", [])
var = datatype("m1", [100])
```

```
var = datatype("m2")
var = datatype("m2", [])
var = datatype("m2", [[1], [2]])
```

```
print var
<uter.class for uter union type object at 0xa8391aec>
Uter union instance: (no name)
size:4
C/C++ description:
union (no name) {
/*00-03*/ int m1;
/*00-01*/ struct (no name) m2;
};
address:0x080530e8
value:
union (no name){
/*00-03*/ int m1; -->513 (hex:0x201)
/*00-01*/ struct (no name) m2;
};
```

创建好的共用体对象，使用 `_Member` 接口或者不以下划线开始的结构成员名，通过接口或者成员名返回的是共用体对象的成员对象（共用体类型的成员是成员的数据类型，共用体对象的成员是成员的数据对象）。比如以下代码：

```
datatypein = uter.struct(("inm1", uter.char), ("inm2", uter.char))
datatype = uter.union(("m1", uter.int), ("m2", datatypein))
```

```
var = datatype("m2", [[0x1], [0x2]])
```

```
print var.m1
<uter.class for uter base type object at 0xacc933ec>
Uter base instance:int
size:4
C/C++ description:int
address:0x08050008
value:513 (hex:0x201)
```

```
print var.m2
<uter.class for uter struct type object at 0xacc9b92c>
```

```

Uter struct instance:(no name)
size:2
C/C++ description:
struct (no name) {
/*00-00*/ char inm1;
/*01-01*/ char inm2;
};
address:0x08050008
value:
struct (no name) {
/*00-00*/ char inm1; -->1 (hex:0x1)
/*01-01*/ char inm2; -->2 (hex:0x2)
};

```

操作共用体对象的数据，都是对其成员的对象进行操作，比如设置和读取值，如果其成员也是结构体或者共用体，则操作成员的成员，以此类推，知道操作到基本类型成员。

## 5.5.2 编程手册

| 函数/成员/方法                        | 参数  | 返回值              | 描述                                     |
|---------------------------------|---|------------------|--|
| uter.union(*args)               | 0 个到多个 Python tuple，元组的第一个元素为 Python str 代表成员名字；元组的第二个元素是一个 Uter data type，代表成员的类型。                     | Uter data type   | 返回定义的共用体数据类型                           |
| cls._Member(name)               | name：共用成员名  | Uter data type   | 返回共用成员的数据类型                            |
| cls.共用成员的名                      | NA  | Uter data type   | 返回共用成员的数据类型，成员名不能以下划线开头                |
| cls.__str__()                   | NA  | Python str       | 数据类型的描述信息                              |
| cls.__init__(mem_name, mem_arg) | (可选) mem_name：Python str 被初始化的成员名<br>(可选) mem_arg：Python tuple (也可以是 Python list)，用于初始化 mem_name 指定的成员。 | NA               | 共用体类型创建对象，如果指定了 mem_arg，则必须指定 mem_name |
| obj._Member(name)               | name：共用成员名  | Uter data object | 返回共用成员的数据对象                            |
| obj.共用成员的名                      | NA  | Uter data object | 返回共用成员的数据对象，成员名不能以下划线开头                |
| obj.__str__()                   | NA  | Python str       | 数据对象的描述信息                              |

## 5.6 数组类型

### 5.6.1 数组类型

数组数据类型同样需要数组使用的类型定义，并需要指定数组的下标，定义的接口为 `uter.point`，该接口有两个参数，第一个参数为数组的类型，第二个参数是一个元组，元组由一个到多个 Python `int` 元素组成，每个元素对应一级数组下标。为了方便输入，该元组可以使用链表代替，如果数组只有一级，则可以直接使用 Python `int`，比如以下代码：

```
datatype = uter.array(uter.int, (2,3))
datatype = uter.array(uter.int, (2,))

datatype = uter.array(uter.int, [2,3])
datatype = uter.array(uter.int, [2])

datatype = uter.array(uter.int, 2)

typeto = uter.struct(("m1", uter.int), ("m2", uter.char), ("m3", uter.char), ("m4", uter.char))
datatype = uter.array(typeto, [2,3])

print datatype
<class 'uter.class for uter array type'>
Uter array type: struct (no name)
size:48
C/C++ description:struct (no name) [2][3]
```

数组数据类型通过 `_GetFinalType` 方法可以获得到定义数组的原始类型，比如以下代码：

```
typeto = uter.struct(("m1", uter.int), ("m2", uter.char), ("m3", uter.char), ("m4", uter.char))
datatype = uter.array(typeto, [2,3])

print datatype
<class 'uter.class for uter array type'>
Uter array type: struct (no name)
size:48
C/C++ description:struct (no name) [2][3]

print datatype._GetFinalType()
<class 'uter.class for uter struct type'>
Uter struct type:(no name)
size:8
C/C++ description:
struct (no name) {
/*00-03*/ int m1;
/*04-04*/ char m2;
/*05-05*/ char m3;
/*06-06*/ char m4; /*07-07*/
};
```

数组数据类型的数组下标可以通过 `_GetBound` 接口获得数组下标，数组下标以 Python `tuple` 方式保存，比如以下代码：

```
typeto = uter.struct(("m1", uter.int), ("m2", uter.char), ("m3", uter.char), ("m4", uter.char))
```

```
datatype = uter.array(typeto, [2,3])

print datatype._GetBound()
(2, 3)
```

### 数组初始化：

数组类型创建数组对象的时候不需要初始化参数，比如以下代码：

```
typeto = uter.struct(("m1", uter.int), ("m2", uter.char), ("m3", uter.char), ("m4", uter.char))
datatype = uter.array(typeto, [2,3])

var = datatype()

print var
<uter.class for uter array type object at 0xacc44bcc>
Uter array instance: struct (no name)
size:48
C/C++ description:struct (no name) [2][3]
address:0x08050008
value:.....
```

数组对象支持下标操作，下标操作返回小一个维度的数组，如果是一维数组取下标，则返回原始数据类型对象。比如以下代码：

```
typeto = uter.struct(("m1", uter.int), ("m2", uter.char), ("m3", uter.char), ("m4", uter.char))
datatype = uter.array(typeto, [2,3])

var = datatype()

print var[1]
<uter.class for uter array type object at 0xa94f288c>
Uter array instance: struct (no name)
size:24
C/C++ description:struct (no name) [3]
address:0x08050068
value:.....

print var[1][1]
<uter.class for uter struct type object at 0xa834cdcc>
Uter struct instance:(no name)
size:8
C/C++ description:
struct (no name) {
/*00-03*/ int m1;
/*04-04*/ char m2;
/*05-05*/ char m3;
/*06-06*/ char m4; /*07-07*/
};
address:0x08050070
value:
struct (no name) {
```



```

/*00-03*/ int m1; -->0 (hex:0x0)
/*04-04*/ char m2; -->0 (hex:0x0)
/*05-05*/ char m3; -->0 (hex:0x0)
/*06-06*/ char m4; -->0 (hex:0x0) /*07-07*/
};

```

上例中的 `var[1][1]`，实际上是 `var[1]` 返回了子数组（1 维数组）对象，然后再使用 `[1]` 取得到结构体类型的对象。

## 5.6.2 编程手册

| 函数/成员/方法                                      | 参数  | 返回值              | 描述  |
|---|---|------------------|---|
| <code>uter.array(datatype, rangetuple)</code> | <code>datatype</code> : Uter data type<br><code>rangetuple</code> : Python tuple , 所有的元素都需要是 Python int , tuple 只要有一个元素 | Uter data type   | 返回定义的数组数据类型                                     |
| <code>cls._GetFinalType()</code>              | NA  | Uter data type   | 返回定义数组所使用的数据类型                                  |
| <code>cls._GetBound()</code>                  | NA  | Python tuple     | 返回定义数组所使用的下标集合                                  |
| <code>cls.__str__()</code>                    | NA  | Python str       | 数据类型的描述信息                                       |
| <code>cls.__init__()</code>                   | NA  | NA               | 数组类型创建对象  |
| <code>obj.__getitem__(key)</code>             | <code>key</code> : Python int   | Uter data object | 如果数组大于 1 维，则返回子数组对象，如果数组等于 1 维，则返回定义数组使用的类型的对象。 |
| <code>obj.__str__()</code>                    | NA  | Python str       | 数据对象的描述信息                                       |

## 5.7 指针类型

### 5.7.1 指针类型

指针定义：

指针数据类型的定义比较简单，通过 `uter.point` 接口以及一个数据类型作为参数，比如以下代码：

```

datatype = uter.point(uter.int)

typeto = uter.struct(("m1", uter.int), ("m2", uter.char), ("m3", uter.char), ("m4", uter.char))
datatype = uter.point(typeto)

print datatype
<class 'uter.class for uter point type'>
Uter point type to struct (no name) :
size:4
C/C++ description: struct (no name) *

```

和数组类型一样，指针类型支持通过 `_GetFinalType` 获取定义指针所使用的数据类型，比如以下代码：

```

datatype = uter.point(uter.int)

```

```

print datatype._GetFinalType()
<class 'uter.class for uter base type'>
Uter base type:int
size:4
C/C++ description:int

typeto = uter.struct(("m1", uter.int), ("m2", uter.char), ("m3", uter.char), ("m4", uter.char))
datatype = uter.point(typeto)
print datatype._GetFinalType()
<class 'uter.class for uter struct type'>
Uter struct type:(no name)
size:8
C/C++ description:
struct (no name) {
/*00-03*/ int m1;
/*04-04*/ char m2;
/*05-05*/ char m3;
/*06-06*/ char m4; /*07-07*/
};

```

### 指针始化：

指针类型数据对象类似于基本数据类型 unsigned long，指针的值（即指针的指向，一个地址值）可以通过\_SetValue 和\_GetValue 来设置指针的值。

直接对指针进行初始化，如果通过参数指定初始值，如果省略参数则指针对象的初始值为 0。比如以下代码：

```

typeto = uter.struct(("m1", uter.int), ("m2", uter.char), ("m3", uter.char), ("m4", uter.char))
datatype = uter.point(typeto)

var = datatype()
print var
<uter.class for uter point type object at 0xacc8ff2c>
Uter point instance to struct (no name) :
size:4
C/C++ description: struct (no name) *
address(point variable on):0x08050008
value(point to):0 (hex:0x0)

var = datatype(0x12345678)
print var
<uter.class for uter point type object at 0xacc963ec>
Uter point instance to struct (no name) :
size:4
C/C++ description: struct (no name) *
address(point variable on):0x08050018
value(point to):305419896 (hex:0x12345678)

var._SetValue(0x88888888)
print hex(var._GetValue())

```

```
0x88888888L
```

如果想要将一个指针指向某对象，先取得对象的地址，然后设置给指针。比如以下代码：

```
typeto = uter.struct(("m1", uter.int), ("m2", uter.char), ("m3", uter.char), ("m4", uter.char))
datatype = uter.point(typeto)

varto = typeto()
print hex(varto._GetAddress())
0x8050028

var = datatype()
print hex(var._GetValue())
0x0

var._SetValue(varto._GetAddress())
print hex(var._GetValue())
0x8050028

print var
<uter.class for uter point type object at 0xacc8ff2c>
Uter point instance to struct (no name) :
size:4
C/C++ description: struct (no name) *
address(point variable on):0x08050038
value(point to):134545448      (hex:0x8050028)
```

Uter 提供了为对象创建一个指向自己的指针型对象的接口 `uter.dopoint`，和从指针型对象获取指针指向对象的接口 `uter.depoint`。比如以下代码：

```
typeto = uter.struct(("m1", uter.int), ("m2", uter.char), ("m3", uter.char), ("m4", uter.char))
datatype = uter.point(typeto)

#Data object to point
varto = typeto()
print hex(varto._GetAddress())
0x8050008L

var = uter.dopoint(varto)
print var
<uter.class for uter point type object at 0xacc9aaac>
Uter point instance to struct (no name) :
size:4
C/C++ description: struct (no name) *
address(point variable on):0x08050018
value(point to):134545416      (hex:0x8050008)

#Point to data object
to_obj = uter.depoint(var)
print hex(to_obj.GetAddress())
0x8050008

print to_obj
```

```

<uter.class for uter struct type object at 0xacc9a94c>
Uter struct instance:(no name)
size:8
C/C++ description:
struct (no name) {
/*00-03*/ int m1;
/*04-04*/ char m2;
/*05-05*/ char m3;
/*06-06*/ char m4; /*07-07*/
};
address:0x08050008
value:
struct (no name) {
/*00-03*/ int m1; -->0 (hex:0x0)
/*04-04*/ char m2; -->0 (hex:0x0)
/*05-05*/ char m3; -->0 (hex:0x0)
/*06-06*/ char m4; -->0 (hex:0x0) /*07-07*/
};

```

## 5.7.2 编程手册

| 函数/成员/方法                 | 参数   | 返回值                    | 描述                   |
|--------------------------|--|------------------------|----------------------|
| uter.dopoint(from_obj)   | from_obj : Uter data object 数据对象                 | Uter data object 指针对象  | 返回一个指针类型对象，并指向参数对象。  |
| uter.depoin(with_point)  | from_obj : Uter data object 指针对象                 | Uter data object 数据对象  | 返回一个数据对象，该对象被指针对象所指。 |
| uter.point(datatype)     | datatype : Uter data type                        | Uter data type         | 返回定义的指针数据类型          |
| cls._GetFinalType()      | NA   | Uter data type         | 返回定义指针所使用的数据类型       |
| cls.__str__()            | NA   | Python str             | 数据类型的描述信息            |
| cls.__init__(init_value) | ( 可选 ) init_value : Python int/Python long 对象初始值 | NA                     | 指针数据类型创建对象           |
| obj._GetValue()          | NA   | Python int/Python long | 返回指针对象的值（类似与基本数据类型）  |
| obj._SetValue(value)     | value : Python int/Python long 需要设定指针值           | NA                     | 设置指针对象的值（类似与基本数据类型）  |
| obj.__str__()            | NA   | Python str             | 数据对象的描述信息            |

## 5.8 VOID 类型

Uter 中提供了预定义类型 `uter.void`，与 C 语言中的 `void` 类型一样，`uter.void` 类型无法生成 `void` 型数据对象，不过可以用于定义 `void` 类型的指针，并且 `uter` 中预定义了 `uter.voidpoint` 数据类型，用于实现 C 语言中的 `void *`。比如以下代码：

```

var = uter.voidpoint()

print var
<uter.class for uter point type object at 0xa83e2fec>
Uter point instance to void :
size:4
C/C++ description: void *
address(point variable on):0x08050028
value(point to):0      (hex:0x0)

var = uter.voidpoint(0x12345678)
print var
<uter.class for uter point type object at 0xacc9630c>
Uter point instance to void :
size:4
C/C++ description: void *
address(point variable on):0x08050038
value(point to):305419896      (hex:0x12345678)

```

## 5.9 Typedef 类型

### 5.9.1 Typedef 类型

在 C/C++ 语言中，typedef 并没有产生新的数据类型，只是给原数据类型设置的别名。在 Uter 中也是一样，通过 uter.typedef 接口定义 typedef 类型，typedef 只是给数据类型设置了别名，被设置过 typedef 的数据类型与原数据类型是等效的，typedef 相当与只是给原数据类型增加了注释，二者除了在显示描述信息时被 typedef 过的类型有显示被 typedef 过的名字意外，没有其他任何区别。比如以下代码：

```

datatype = uter.struct(("m1", uter.int), ("m2", uter.char))
deftype = uter.typedef(datatype, "testdef")

print datatype
<class 'uter.class for uter struct type'>
Uter struct type:(no name)
size:8
C/C++ description:
struct (no name) {
/*00-03*/ int m1;
/*04-04*/ char m2; /*05-07*/
};

print deftype
<class 'uter.class for uter struct type'>
Typedef as testdef
Uter struct type:(no name)
size:8
C/C++ description:
struct (no name) {
/*00-03*/ int m1;
/*04-04*/ char m2; /*05-07*/
};

```

```
};
```

typedef 的类型与原类型在创建对象上也是一致的，比如以下代码：

```
datatype = uter.struct(("m1", uter.int), ("m2", uter.char))
deftype = uter.typedef(datatype, "testdef")

data_var = datatype()
def_var = deftype()

print data_var
<uter.class for uter struct type object at 0xa80c088c>
Uter struct instance:(no name)
size:8
C/C++ description:
struct (no name) {
/*00-03*/ int m1;
/*04-04*/ char m2; /*05-07*/
};
address:0x08052f48
value:
struct (no name) {
/*00-03*/ int m1; -->0 (hex:0x0)
/*04-04*/ char m2; -->0 (hex:0x0) /*05-07*/
};

print def_var
<uter.class for uter struct type object at 0xa80e744c>
Typedef as testdef
Uter struct instance:(no name)
size:8
C/C++ description:
struct (no name) {
/*00-03*/ int m1;
/*04-04*/ char m2; /*05-07*/
};
address:0x08052f58
value:
struct (no name) {
/*00-03*/ int m1; -->0 (hex:0x0)
/*04-04*/ char m2; -->0 (hex:0x0) /*05-07*/
};
```

所以，在 Uter 中，typedef 不没有起到什么作用，支持 typedef 的主要作用是支持 typedef 数据类型导出，具体参考后文数据导出。

## 5.9.2 编程手册

| 函数/成员/方法               | 参数                          | 返回值            | 描述                  |
|------------------------|-----------------------------|----------------|---------------------|
| uter.typedef(datatype, | datatype : Uter data type 被 | Uter data type | 返回设置过 typedef 的数据类型 |

|       |   |  |  |
|-------|---|--|--|
| name) | typedef 的数据类型。<br>name : Python string<br>typedef 的名字 |  |  |
|-------|---|--|--|

## 6 类型转换/内存/字符串

### 6.1 数据类型转换

#### 6.1.1 数据类型转换

Uter 中支持数据类型的转换，使用数据对象的 `_AsType()` 接口，将一个数据类型的对象转换成另外一个类型的对象，转换的参数是新对象的类型，转换以后原对象还继续存在，只是转换前后的两个对象在目标程序中的地址是相同的，比如以下代码：

```
var_before = uter.int(0x12345678)

var_after = var_before._AsType(uter.char)

print var_after
<uter.class for uter base type object at 0xa83ea58c>
Uter base instance:char
size:1
C/C++ description:char
address:0x08050048
value:120      (hex:0x78)

print var_before
<uter.class for uter base type object at 0xa83ea7ec>
Uter base instance:int
size:4
C/C++ description:int
address:0x08050048
value:305419896      (hex:0x12345678)
```

在转换过程中，新旧两个对象使用相同的内存地址，但是将短长度数据转换成长长度数据，会存在过界，这属于正常现象。

在实际应用中，将一维字符型数组元素转换成指定的数据类型，广泛的用于内存分配使用和消息处理，比如以下代码：

```
array_type = uter.array(uter.char, 100)
array_data = array_type()

st_type = uter.struct(("m1", uter.int), ("m2", uter.char), ("m3", uter.char), ("m4", uter.char), ("m5",
uter.char))

st_data_1 = array_data._AsType(st_type)

print array_data
<uter.class for uter array type object at 0xa83eaa6c>
```

```

Uter array instance: char
size:100
C/C++ description:char [100]
address:0x08050058
value:.....

print st_data_1
<uter.class for uter struct type object at 0xa83ea34c>
Uter struct instance:(no name)
size:8
C/C++ description:
struct (no name) {
/*00-03*/ int m1;
/*04-04*/ char m2;
/*05-05*/ char m3;
/*06-06*/ char m4;
/*07-07*/ char m5;
};
address:0x08050058
value:
struct (no name) {
/*00-03*/ int m1; -->0 (hex:0x0)
/*04-04*/ char m2; -->0 (hex:0x0)
/*05-05*/ char m3; -->0 (hex:0x0)
/*06-06*/ char m4; -->0 (hex:0x0)
/*07-07*/ char m5; -->0 (hex:0x0)
};

st_data_2 = array_data[st_type._GetSize()]._AsType(st_type)

print st_data_2
<uter.class for uter struct type object at 0xa83ea66c>
Uter struct instance:(no name)
size:8
C/C++ description:
struct (no name) {
/*00-03*/ int m1;
/*04-04*/ char m2;
/*05-05*/ char m3;
/*06-06*/ char m4;
/*07-07*/ char m5;
};
address:0x08050060
value:
struct (no name) {
/*00-03*/ int m1; -->0 (hex:0x0)
/*04-04*/ char m2; -->0 (hex:0x0)
/*05-05*/ char m3; -->0 (hex:0x0)
/*06-06*/ char m4; -->0 (hex:0x0)
/*07-07*/ char m5; -->0 (hex:0x0)
};

```

以上代码中，将数组本身或者数组中指定的元素转换成其他复杂类型，数组或者数组元素主要的一个



作用就是提供地址。

### 6.1.2 编程手册

| 函数/成员/方法             | 参数                                      | 返回值              | 描述       |
|----------------------|---|------------------|----------|
| obj._AsType(as_type) | as_type : Uter data type ,<br>要转换成的数据类型 | Uter data object | 返回转换成的对象 |

## 6.2 内存分配与释放

### 6.2.1 内存分配与释放

Uter 提供了分配内存接口 `uter.memory.malloc()` 和释放内存接口 `uter.memory.free()`，其参数分别为长度和内存地址（也可以使用数据对象，取对象的内存地址用于释放）。通过 `uter.memory.malloc` 分配的内存可以被程序中的 `free` 释放，程序中 `malloc` 分配的内存可以通过 `uter.memory.free` 所释放。因此，使用这两个接口存在内存泄漏和重复释放问题，需要注意使用。`uter.memory.malloc` 接口返回 `char` 类型的数组，数组的长度为分配内存的长度（与返回 `void *` 类型相比，除了都支持的获取内存地址，还保存着内存块的长度，还方便进行位移操作）。比如以下代码：

```
var_array = uter.memory.malloc(100)

print var_array
<uter.class for uter array type object at 0xacc983cc>
Uter array instance: char
size:100
C/C++ description:char [100]
address:0x08050008
value:.....

st_type = uter.struct(("m1", uter.int), ("m2", uter.char), ("m3", uter.char), ("m4", uter.char), ("m5",
uter.char))
var_st = var_array._AsType(st_type)

print var_st
<uter.class for uter struct type object at 0xacc9ca2c>
Uter struct instance:(no name)
size:8
C/C++ description:
struct (no name) {
/*00-03*/ int m1;
/*04-04*/ char m2;
/*05-05*/ char m3;
/*06-06*/ char m4;
/*07-07*/ char m5;
};
address:0x08050008
value:
struct (no name) {
/*00-03*/ int m1; -->0 (hex:0x0)
```

```
/*04-04*/ char m2; -->0 (hex:0x0)
/*05-05*/ char m3; -->0 (hex:0x0)
/*06-06*/ char m4; -->0 (hex:0x0)
/*07-07*/ char m5; -->0 (hex:0x0)
};

uter.memory.free(var_array)
```

6.2.2 编程手册

| 函数/成员/方法                      | 参数   | 返回值                          | 描述                                  |
|-------------------------------|--|------------------------------|-------------------------------------|
| uter.memory.malloc(len)       | len : Python int , 分配数据的长度                               | Uter data object , char 数组类型 | 返回一个 char 数组类型对象，数组的长度为分配的长度。       |
| uter.memory.free(addr_or_obj) | addr_or_obj : Python int/Python long 或者 Uter data object | NA                           | 对指定的地址做释放，如果是数据对象做参数，则按照数据对象的地址做释放。 |

6.3 字符串操作

6.3.1 字符串操作

Uter 还提供了字符串操作接口，对于 ASCII 的字符串，使用 uter.string.ascii.set 和 uter.string.ascii.get 接口进行操作，比如以下代码：

```
uchar_array_type = uter.array(uter.char, 100)
uchar_array_data = uchar_array_type()

uter.string.ascii.set(uchar_array_data._GetAddress(), "Hello")

print chr(uchar_array_data[0]._GetValue())
H

print chr(uchar_array_data[1]._GetValue())
e

print chr(uchar_array_data[2]._GetValue())
l

print chr(uchar_array_data[3]._GetValue())
l

print chr(uchar_array_data[4]._GetValue())
o

print chr(uchar_array_data[5]._GetValue())
#No out put
```

```
print uchar_array_data[5]._GetValue()
0

uchar_array_data[5]._SetValue(ord('W'))
uchar_array_data[6]._SetValue(ord('o'))
uchar_array_data[7]._SetValue(ord('r'))
uchar_array_data[8]._SetValue(ord('l'))
uchar_array_data[9]._SetValue(ord('d'))

print uter.string.ascii.get(uchar_array_data._GetAddress())
HelloWorld
```

6.3.2 编程手册

| 函数/成员/方法                                       | 参数  | 返回值                 | 描述        |
|--|---|---------------------|-----------|
| uter.string.ascii.set(setaddr, setstr, setlen) | setaddr : Python int/Python long , 内存地址<br>setstr : Python str 需要设置的字符串<br>( 可选 ) setlen : 设置字符串的最大长度 | NA                  | 向内存设置字符串  |
| uter.string.ascii.get(getaddr, getlen)         | getaddr : Python int/Python long , 内存地址<br>( 可选 ) getlen : 获取字符串的最大长度                                 | Python str : 获得的字符串 | 从内存中读取字符串 |

7 对象导出

7.1 range 对象

7.1.1 range 对象

在 Uter 中自定义型数据类型，比如结构体类型等，不仅能够通过接口定义得到数据类型类，还可以从目标程序中导出自定义数据类型。由于在同一个目标程序中，多个数据类型或者多个变量使用同一个名字，因此在导出的时候，通常需要指定导出范围。

在一个目标程序中首先由不同的模块组成，模块包括主程序和动态库（在 Linux 中通常以 so 作为扩展名），模块是由一个或者多个 obj 文件链接而成的产物，模块的路径作为模块标识。比如以下的程序/usr/bin/wget，在启动以后，主程序/usr/bin/wget 和列出的其他动态库都是该进程 的模块。

```
ldd /usr/bin/wget
linux-gate.so.1 => (0xb7791000)
libpcre.so.3 => /lib/i386-linux-gnu/libpcre.so.3 (0xb7684000)
libuuid.so.1 => /lib/i386-linux-gnu/libuuid.so.1 (0xb767e000)
libssl.so.1.0.0 => /lib/i386-linux-gnu/libssl.so.1.0.0 (0xb7613000)
```

```
libcrypto.so.1.0.0 => /lib/i386-linux-gnu/libcrypto.so.1.0.0 (0xb7424000)
libz.so.1 => /lib/i386-linux-gnu/libz.so.1 (0xb7409000)
libidn.so.11 => /usr/lib/i386-linux-gnu/libidn.so.11 (0xb73d6000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb721b000)
libpthread.so.0 => /lib/i386-linux-gnu/libpthread.so.0 (0xb71fd000)
/lib/ld-linux.so.2 (0x800d9000)
libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0xb71f8000)
```

一个模块由不同的编译单元组成，编译单元对应着一次编译（GCC 中使用 -c 进行的编译，只编译不链接），编译单元由编译的主文件以及其包含的文件一同编译并产生 obj 文件，编译单元的主文件路径作为编译单元的标识。比如以下的例子中，/home/usr/project/file\_1.c 和 /home/usr/project/file\_2.c 都将是 /home/user/project/main\_module 模块的编译单元。

```
gcc -c -g /home/usr/project/file_1.c -o /home/usr/project/file_1.o
gcc -c -g /home/usr/project/file_2.c -o /home/usr/project/file_2.o

gcc /home/usr/project/file_1.o /home/usr/project/file_2.o -o /home/user/project/main_module
```

Uter 中使用 `uter.target.mirror.range` 类专门用于描述元素范围，其简化变量名为 `uter.range`。通过 `uter.range` 类初始化对象的时候可以省略或者制定初始化参数，其中 `mod_path` 和 `mod_regex` 至多指定其中一个，`cu_path` 和 `cu_regex` 至多指定其中一个，制定的模块参数或者编译单元参数用于选择匹配的模块和其中的编译单元，如果没有制定匹配参数，则会匹配所有模块和编译单元。`mod_path` 和 `cu_path` 参数如果是 / 为开头的绝对路径，则进行绝对路径匹配，如果不是以 / 开头的相对路径，则进行相对路径匹配，比如以下代码：

```
myrange = uter.range()
print myrange
<uter.range object at 0xa80e63ec>
Mod: /home/yaozhongkai/uter/examples/example_base/src/test.exe
  Cu:/home/yaozhongkai/uter/examples/example_base/src/file_1.c
  Cu:/home/yaozhongkai/uter/examples/example_base/src/main.c
  Cu:/home/yaozhongkai/uter/examples/example_base/src/file_2.c
  Cu:/home/yaozhongkai/uter/examples/example_base/src/useful.c
Mod: /lib/i386-linux-gnu/libc-2.21.so
Mod: /lib/i386-linux-gnu/ld-2.21.so
Mod: /lib/i386-linux-gnu/libdl-2.21.so
Mod: [vdso]

myrange = uter.range(mod_path = "src/test.exe")
print myrange
<uter.range object at 0xa80e618c>
Mod: /home/yaozhongkai/uter/examples/example_base/src/test.exe
  Cu:/home/yaozhongkai/uter/examples/example_base/src/file_1.c
  Cu:/home/yaozhongkai/uter/examples/example_base/src/main.c
  Cu:/home/yaozhongkai/uter/examples/example_base/src/file_2.c
  Cu:/home/yaozhongkai/uter/examples/example_base/src/useful.c

myrange = uter.range(mod_regex = ".*test.exe")
print myrange
<uter.range object at 0xa80e63ec>
Mod: /home/yaozhongkai/uter/examples/example_base/src/test.exe
  Cu:/home/yaozhongkai/uter/examples/example_base/src/file_1.c
```

```
Cu:/home/yaozhongkai/uter/examples/example_base/src/main.c
Cu:/home/yaozhongkai/uter/examples/example_base/src/file_2.c
Cu:/home/yaozhongkai/uter/examples/example_base/src/useful.c
```

```
myrange = uter.range(mod_path = "src/test.exe", cu_path = "src/file_1.c")
```

```
print myrange
```

```
<uter.range object at 0xacc9326c>
```

```
Mod: /home/yaozhongkai/uter/examples/example_base/src/test.exe
```

```
Cu:/home/yaozhongkai/uter/examples/example_base/src/file_1.c
```

```
myrange = uter.range(mod_regex = ".*test.exe", cu_regex = ".*src/file_1.c")
```

```
print myrange
```

```
<uter.range object at 0xa80e63ec>
```

```
Mod: /home/yaozhongkai/uter/examples/example_base/src/test.exe
```

```
Cu:/home/yaozhongkai/uter/examples/example_base/src/file_1.c
```

## 7.1.2 编程手册

| 函数/成员/方法   | 参数   | 返回值        | 描述  |
|--|--|------------|---|
| uter.target.mirror.range(mod_path, mod_regex, cu_path, cu_regex) | mod_path : Python str 模块路径<br>mod_regex : Python str 模块正则表达式<br>cu_path : Python str 编译单元路径<br>cu_regex : Python str 编译单元正则表达式 | Range 对象   | mod_path 和 mod_regex 不能同时使用, cu_path 和 cu_regex 不能同时使用。 |
| uter.range(mod_path, mod_regex, cu_path, cu_regex)               | mod_path : Python str 模块路径<br>mod_regex : Python str 模块正则表达式<br>cu_path : Python str 编译单元路径<br>cu_regex : Python str 编译单元正则表达式 | Range 对象   | uter.target.mirror.range 的短名称                           |
| obj.__str__  | NA   | Python str | 返回 Range 对象的描述信息  |

## 7.2 数据类型导出

从 Uter 中直接导出自定义数据类型可以免去自定义的麻烦, 而且和还可以避免发生错误。对于容易发生变动的结构, 导出数据自定义数据类型更能最大的避免因为结构修改而频繁变更测试用例。导出数据类型的接口有多个, 分别为:

- uter.export.struct : 导出结构体类型
- uter.export.union : 导出共用体类型
- uter.export.enum : 导出枚举类型
- uter.export.typedef 导出 typedef 类型
- uter.export.anytype : 导出以上支持的数据类型。

所有的导出接口都支持必须的数据类型名参数和可选的 range 参数，比如以下 C 代码：

```
struct st_base {
    char m_1;
    int m_2;
};
struct st_base st_base_var;

/*gcc -g main.c -o test.bin*/
```

使用如下代码导出结构体类型：

```
datatype = uter.export.struct("st_base")
print datatype
<class 'uter.class for uter struct type'>
Uter struct type:st_base
size:8
C/C++ description:
struct st_base {
/*00-00*/ char m_1; /*01-03*/
/*04-07*/ int m_2;
};
```

也可以使用 range 参数，限定导出的范围，避免冲突，比如以下代码：

```
myrange = uter.range(mod_path = "test.bin", cu_path = "main.c")

datatype = uter.export.struct("st_base", myrange)
print datatype
<class 'uter.class for uter struct type'>
Uter struct type:st_base
size:8
C/C++ description:
struct st_base {
/*00-00*/ char m_1; /*01-03*/
/*04-07*/ int m_2;
};
```

如果想要导出的数据类型不存在，则会发生异常，比如以下代码：

```
datatype = uter.export.struct("st_base_no_exist")

Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "/...../python/uter.py", line 2561, in struct
    return internal.export.type("struct", name, range)
  File "/...../python/uter.py", line 1913, in type
    raise error.NotExist("No such type.")
uter.NotExist: No such type.
```

如果想要导出的数据类型发生重复，则会发生异常，比如以下代码：

```
datatype = uter.export.struct("st_base_type_dup")

Traceback (most recent call last):
  File "<string>", line 1, in <module>
```

```
File "/...../python/uter.py", line 2561, in struct
    return internal.export.type("struct", name, range)
File "/...../python/uter.py", line 1922, in type
    raise error.Ambiguous(str)
uter.Ambiguous: Type duplicate:
Module:/...../test.bin    Compile unit:/...../src/file_1.c
Module:/...../test.bin    Compile unit:/...../src/main.c
```

## 7.3 变量导出

Uter 除了支持导出数据类型为数据类型，还至此导出变量为数据对象，使用 `uter.export.variable` 接口。比如以下 C 代码：

```
struct st_base {
    char m_1;
    int m_2;
};
struct st_base st_base_var;

/*gcc -g main.c -o test.bin*/
```

使用以下代码将全局变量导出：

```
var = uter.export.variable("st_base_var")

print var
<uter.class for uter struct type object at 0xa839266c>
Uter struct instance:st_base
size:8
C/C++ description:
struct st_base {
/*00-00*/ char m_1; /*01-03*/
/*04-07*/ int m_2;
};
address:0x0804f5dc
value:
struct st_base {
/*00-00*/ char m_1; -->0 (hex:0x0) /*01-03*/
/*04-07*/ int m_2; -->0 (hex:0x0)
};
```

被到处数据对象与通过数据类型创建的数据对象一样可以被设置和获取值（基本数据类型）和获取成员操作（结构体/共用体类型），只是这些数据对象的内存地址就是全局变量的地址。

变量导出接口支持一个 `static` 参数，默认为 `False`，如果设置 `True`，则导出变量的时候会对 `static` 变量做导出，否则会忽略 `static` 变量。

变量导出接口也支持 `range` 参数，并却会在到处变量不存在或者导出变量冲突的时候发生异常。

## 7.4 函数导出

Uter 还支持导出函数，像导出变量的数据对象一样导出函数的函数对象，函数对象也是一种内存中的对象，只是不能被设置和获取值，但是可以被执行和打桩。执行和打桩会在后文中介绍，本节先只介绍函数的导出。导出函数使用 `uter.export.function` 接口，导出参数与导出变量是一样的。比如以下 C 代码：

```
int fun_base(int a, int b)
{
    return a + b;
}

/*gcc -g main.c -o test.bin*/
```

导出函数使用如下代码：

```
fun = uter.export.function("fun_base")

print fun
<uter.class for uter function type object at 0xacc8ff0c>
Uter fuction instance: (no name)
Type:
C/C++ description:int (no name) (int , int )
name:fun_base
address(function at):0x080487fc
value:NA
length:10
```

## 7.5 编程手册

| 函数/成员/方法   | 参数   | 返回值              | 描述              |
|--|--|------------------|-----------------|
| <code>uter.export.struct(name, range)</code>           | name : Python str 结构体的名字<br>(可选) range : range 对象。       | Uter data type   | 返回结构体数据类型       |
| <code>uter.export.union(name, range)</code>            | name : Python str 共用体的名字<br>(可选) range : range 对象。       | Uter data type   | 返回共用体数据类型       |
| <code>uter.export.enum(name, range)</code>             | name : Python str 枚举的名字<br>(可选) range : range 对象。        | Uter data type   | 返回枚举数据类型        |
| <code>uter.export.typedef(name, range)</code>          | name : Python str typedef 的名字<br>(可选) range : range 对象。  | Uter data type   | 返回 typedef 数据类型 |
| <code>uter.export.anytype(name, range)</code>          | name : Python str 数据类型的名字<br>(可选) range : range 对象。      | Uter data type   | 返回匹配的数据类型       |
| <code>uter.export.variable(name, range, static)</code> | name : Python str 变量的名字<br>(可选) static : Python bool, 是否 | Uter data object | 返回变量的数据对象       |



|   |  |                      |           |
|---|--|----------------------|-----------|
| e, static, range)                         | 对 static 变量导出<br>( 可选 ) range : range 对象。  |                      |           |
| uter.export.function(name, static, range) | name : Python str 函数的名字<br>( 可选 ) static : Python bool , 是否对 static 函数导出<br>( 可选 ) range : range 对象。 | Uter function object | 返回函数的函数对象 |

## 8 宏操作

略

## 9 函数操作

在 Uter 中函数也是一种对象，在目标程序中有对应的内存，虽然无法像数据对象一样执行\_SetValue和\_GetValue，但是可以对数据对象进行调用和打桩。函数有对象同时也有函数类型，函数类型与数据类型类似，描述着函数的结构。函数的结构包括返回值类型，参数的数量和各个参数的类型。在 Uter 中通过接口uter.function 创建函数类型，比如以下代码：

```
funtype = uter.function(uter.int, (uter.int, uter.int))

print funtype
<class 'uter.class for uter function type'>
Uter fuction type: (no name)
Type:
C/C++ description:int (no name) (int , int )
```

函数类型支持\_RetType 和 \_ArgType 获取函数的返回类型和参数类型，比如以下代码：

```
funtype = uter.function(uter.int, (uter.int, uter.int))

print funtype._RetType()
<class 'uter.class for uter base type'>
Uter base type:int
size:4
C/C++ description:int

print funtype._ArgType()
(<class 'uter.class for uter base type'>, <class 'uter.class for uter base type'>)

print funtype._ArgType()[0]
<class 'uter.class for uter base type'>
Uter base type:int
size:4
C/C++ description:int

print funtype._ArgType()[1]
<class 'uter.class for uter base type'>
Uter base type:int
size:4
C/C++ description:int
```

可以通过函数类型来创建函数对象，被创建以后的函数对象会具有内存地址。被设置的地址必须是有效的函数地址，并且所指向的函数需要与函数类型有一致的返回值类型和参数类型。如果数据类型不一致，则可能会造成程序异常。比如以下代码：

```
funtype = uter.function(uter.int, (uter.int, uter.int))

funvar = funtype(address = 0x12345678)
print funvar
<uter.class for uter function type object at 0xacc9828c>
Uter fuction instance: (no name)
Type:
C/C++ description:int (no name) (int , int )
name:(no name)
address(function at):0x12345678
value:NA
length:0
```

以上代码中仅仅是例子，实际使用中 0x12345678 因该是内存中函数的地址。函数对象也可以通过 \_GetAddress 接口获取其内存地址。

## 9.1 调用

对函数进行调用时需要调用函数对象的 \_Call 接口，在 \_Call 接口中传入指定的参数对象，参数的数量需要与函数的参数数量相同，而却还需要是参数类型的数据对象。并且 \_Call 接口会按照函数的返回值类型返回数据对象。比如以下 C 代码：

```
int fun_base(int a, int b)
{
    return a + b;
}

/*gcc -g main.c -o test.bin*/
```

使用如下代码调用函数对象：

```
fun = uter.export.function("fun_base")

arg_1 = uter.int(5)
arg_2 = uter.int(6)

ret = fun._Call(arg_1, arg_2)
print ret
<uter.class for uter base type object at 0xa81698ac>
Uter base instance:int
size:4
C/C++ description:int
address:0x08052fa8
value:11      (hex:0xb)
```

上边例子中的函数有两个参数，调用函数前首先需要准备好调用所使用的参数。然后使用 \_Call 接口调用目标程序中的函数。函数调用结束以后，返回了数据对象，通过 \_GetValue 获得返回对象的值。在调用 \_Call 的时，一些特殊情况需说明：

- 没有参数的函数在\_Call 的时候，不需要传入任何数据对象。
- 返回值为 void 的函数，\_Call 调用返回 uter.void

## 9.2 打桩

在对函数的调用过程中，如果被调用的函数调用了其他函数，则这个其他函数的返回值和出参将会影响到被测试函数执行。为了保证被测试函数被测试充分，这个被间接调用的其他函数需要返回各种正确错误和设置各种出参数以满足被测试函数需要，然而在现实情况中，被间接调用的函数往往总是只给返回错误，或者返回一部分异常情况，往往无法满足被测试函数对返回结果和出参数的条件组合需要。Uter 提供了对函数打桩功能，能够接管间接调用函数的调用，通过脚本控制调用过程。

对函数进行打桩需要调用函数对象的 \_Stub 接口，Stub 接口的参数为一个 Python 全局函数或者带 @staticmethod 的类函数。被作为参数的 Python 函数（后文称 Stub 函数），参数和返回值都需要满足要求，这些要求包括：

- Stub 函数的参数的数量需要与被打桩函数的数量相同，并且按照参数类型的数据对象来操作函数参数：当 Stub 函数因为间接函数被调用的时候，会按照参数顺序向 Stub 函数传入数据对象进行调用，通过 Stub 函数参数的对象获得调用得到调用时的传值。如果被打桩函数没有参数，则 Stub 函数也不能有参数。
- Stub 函数的返回值需要是被打桩函数返回值类型的对象：当 Stub 函数因为间接函数被调用的时候，原本需要被调用的被打桩函数不会被调用，而是通过调用 Stub 函数取而代之。自然也会通过 Stub 函数得到返回值，因此 Stub 函数需要按照函数的类型返回数据对象。如果被打桩函数返回 void，则 Stub 函数返回 uter.void。
- 如果对已经打桩的函数清除桩，对 \_Stub 接口设置 None 参数。
- 一个函数至多设置一个 Stub 函数，如果在未清除桩的情况下再次设置 Stub 函数，则原有的 Stub 函数会被先清除。

比如以下 C 代码：

```
int fun_stub(int a, int b)
{
    int ret_val = 0;
    /*do something*/
    return ret_val;
}

int fun_call(int a, int b)
{
    int stub_ret = fun_stub(a + 1, b + 1);
    if (stub_ret < 0) {
        return -1;
    } else if (stub_ret == 0) {
        return 0;
    } else {
        return 1;
    }
}
```

```
    }  
    return -1;  
}  
  
/*gcc -g main.c -o test.bin*/
```

使用如下代码打桩：

```
fun_stub = uter.export.function("fun_stub")  
  
fun_call = uter.export.function("fun_call")  
  
def stub_less(a, b):  
    print a._GetValue()  
    print b._GetValue()  
    return uter.int(-10)  
  
def stub_equal(a, b):  
    print a._GetValue()  
    print b._GetValue()  
    return uter.int(0)  
  
def stub_more(a, b):  
    print a._GetValue()  
    print b._GetValue()  
    return uter.int(10)  
  
fun_stub._Stub(stub_less)  
ret_obj = fun_call._Call(uter.int(5), uter.int(6))  
6  
7  
print ret_obj._GetValue()  
-1  
fun_stub._Stub(None)  
  
fun_stub._Stub(stub_equal)  
ret_obj = fun_call._Call(uter.int(5), uter.int(6))  
6  
7  
print ret_obj._GetValue()  
0  
fun_stub._Stub(None)  
  
fun_stub._Stub(stub_more)  
ret_obj = fun_call._Call(uter.int(5), uter.int(6))  
6  
7  
print ret_obj._GetValue()  
1  
fun_stub._Stub(None)
```

9.3 编程手册

| 函数/成员/方法                                 | 参数  | 返回值                           | 描述                                    |
|--|---|-------------------------------|---------------------------------------|
| uter.function(ret_type, args_type_tuple) | ret_type : Uter data type<br>函数返回类型<br>args_type_tuple : Python tuple, 元组的元素为函数参数类型   | Python function type          | 创建函数类型                                |
| cls._RetType()                           | NA  | Python data type              | 函数的返回值类型                              |
| cls._ArgType()                           | NA  | Python tuple                  | 返回元组, 元组内元素依次函数参数的数据类型                |
| cls.__str__()                            | NA  | Python str                    | 函数类型的描述信息                             |
| cls.__init__(address, length)            | (可选)address : Python int/Python long 函数地址<br>(可选)length : Python int/Python long 函数长度 | Python function object        | 函数创建对象                                |
| obj._GetAddress()                        | NA  | Python long                   | 返回函数地址                                |
| obj._GetType()                           | NA  | Python function type, 函数的类型   | 返回函数的函数类型                             |
| obj._Call(*args)                         | 0-n 个 Uter data object, 其数量与类型需要与函数匹配。  | Python data object, 类型为函数返回类型 | 调用函数                                  |
| obj._Stub(stub)                          | stub : Python function/Python static method, 或者是 None。                                | NA                            | 如果参数是函数, 则对函数对象进行打桩。如果参数是 None, 则清除桩。 |
| obj.__str__()                            | NA  | Python str                    | 函数对象的描述信息                             |

10 目标操作

Uter 提供了一系列接口, 用于操作目标程序, 用于获取目标程序信息和对目标程序进行设置。

10.1 字节序

Uter 提供了获取目标程序字节序的接口 `uter.target.endian.isbig()`和 `uter.target.endian.islittle()`, 用于字节序相关的测试例, 同时也提供了 `uter.target.endian.cmd_show()`做查询命令。比如以下代码:

```
uter.target.endian.cmd_show()
=====
Target is little-endian
=====

print uter.target.endian.isbig()
False

print uter.target.endian.islittle()
```

True

## 10.2 桩管理

在 Uter 中可以通过脚本函数对目标程序中的函数打桩，所以专门提供了一系列接口用于管理维护桩信息，包括信息获取接口 `uter.target.stub.list()` 和 清除接口 `uter.target.stub.clear()`。

```
fun_stub = uter.export.function("fun_stub")
fun_call = uter.export.function("fun_call")

def stub_more(a, b):
    print a._GetValue()
    print b._GetValue()
    return uter.int(10)

fun_stub._Stub(stub_more)

uter.target.stub.cmd_show()
=====
Function(C/C++) fun_stub address:0x08049a52:
  Mod:/home/yaozhongkai/uter/examples/example_base/src/test.exe,
  Cu:/home/yaozhongkai/uter/examples/example_base/src/main.c
  Script stub(Python):<function stub_more at 0xacc7fc34>
=====

print uter.target.stub.list()
[{'arg': {'name': 'fun_stub', 'args': ({'align': 4, 'type': <class 'uter.class for uter base type'>, 'len': 4,
'desc': 'int'}, {'align': 4, 'type': <class 'uter.class for uter base type'>, 'len': 4, 'desc': 'int'}), 'driver':
<function FunctionStubDriver at 0xa838bca4>, 'ret': {'align': 4, 'type': <class 'uter.class for uter base
type'>, 'len': 4, 'desc': 'int'}, 'stub': <function stub_more at 0xacc7fc34>, 'cu':
'/home/yaozhongkai/uter/examples/example_base/src/main.c', 'mod':
'/home/yaozhongkai/uter/examples/example_base/src/test.exe'}, 'address': 134519378L}]

print hex(uter.target.stub.list()[0]["address"])
0x8049a52L

print uter.target.stub.list()[0]["arg"]["mod"]
/home/yaozhongkai/uter/examples/example_base/src/test.exe

print uter.target.stub.list()[0]["arg"]["cu"]
/home/yaozhongkai/uter/examples/example_base/src/main.c

print uter.target.stub.list()[0]["arg"]["name"]
fun_stub

print uter.target.stub.list()[0]["arg"]["stub"]
<function stub_more at 0xacc7fc34>

uter.target.stub.clear(0x8049a52L)

uter.target.stub.cmd_show()
```

```
=====
No any stub in target.
=====
```

## 10.3 镜像

在进行数据类型，数据对象，函数对象导出的时候，需要指定查找范围。uter.target.mirror.range 作为导出范围的描述类，而 uter.range 是该类的简便写法。

同时 Uter 还提供了一些关于目标程序的一些操作命令，具体参考以下代码：

```
uter.target.mirror.cmd_mod_show()
=====
/home/yaozhongkai/uter/examples/example_base/src/test.exe
/lib/i386-linux-gnu/libc-2.21.so
/lib/i386-linux-gnu/ld-2.21.so
/lib/i386-linux-gnu/libdl-2.21.so
[vdso]
=====

uter.target.mirror.cmd_cu_show()
=====
Mod:/home/yaozhongkai/uter/examples/example_base/src/test.exe
  Cu:/home/yaozhongkai/uter/examples/example_base/src/file_1.c
  Cu:/home/yaozhongkai/uter/examples/example_base/src/main.c
  Cu:/home/yaozhongkai/uter/examples/example_base/src/file_2.c
  Cu:/home/yaozhongkai/uter/examples/example_base/src/useful.c
Mod:/lib/i386-linux-gnu/libc-2.21.so
Mod:/lib/i386-linux-gnu/ld-2.21.so
Mod:/lib/i386-linux-gnu/libdl-2.21.so
Mod:[vdso]
=====

uter.target.mirror.cmd_file_show()
=====
Mod:/home/yaozhongkai/uter/examples/example_base/src/test.exe
  Cu:/home/yaozhongkai/uter/examples/example_base/src/file_1.c
    File 1:/home/yaozhongkai/uter/examples/example_base/src/head.h
    File 2:/home/yaozhongkai/uter/examples/example_base/src/file_1.c
  Cu:/home/yaozhongkai/uter/examples/example_base/src/main.c
    File 1:/home/yaozhongkai/uter/examples/example_base/src/head.h
    File 2:/home/yaozhongkai/uter/examples/example_base/src/main.c
    File 3:/usr/lib/gcc/i686-linux-gnu/5/include/stddef.h
    File 4:/usr/include/i386-linux-gnu/bits/types.h
    File 5:/usr/include/libio.h
    File 6:/usr/include/i386-linux-gnu/bits/pthreadtypes.h
    File 7:/usr/include/stdio.h
  Cu:/home/yaozhongkai/uter/examples/example_base/src/file_2.c
    File 1:/home/yaozhongkai/uter/examples/example_base/src/head.h
    File 2:/home/yaozhongkai/uter/examples/example_base/src/file_2.c
  Cu:/home/yaozhongkai/uter/examples/example_base/src/useful.c
```

```
File 1:/home/yaozhongkai/uter/examples/example_base/src/useful.c
Mod:/lib/i386-linux-gnu/libc-2.21.so
Mod:/lib/i386-linux-gnu/ld-2.21.so
Mod:/lib/i386-linux-gnu/libdl-2.21.so
Mod:[vdso]
=====
```

10.4 编程手册

| 函数/成员/方法                           | 参数                                  | 返回值                       | 描述          |
|------------------------------------|-------------------------------------|---------------------------|-------------|
| uter.target.endian.isbig()         | NA                                  | Python bool               | 目标程序为大头序否   |
| uter.target.endian.islittle()      | NA                                  | Python bool               | 目标程序为小头序否   |
| uter.target.endian.cmd_show()      | NA                                  | NA                        | 命令：输出字节序信息  |
| uter.target.stub.list()            | NA                                  | Python list ,<br>元素为桩信息字典 |             |
| uter.target.stub.clear(addr)       | addr : Python int/Python long , 桩地址 | NA                        | 清除桩         |
| uter.target.stub.cmd_show()        | NA                                  | NA                        | 命令：输出桩信息    |
| uter.target.mirror.cmd_mod_show()  | NA                                  | NA                        | 命令：输出模块信息   |
| uter.target.mirror.cmd_cu_show()   | NA                                  | NA                        | 命令：输出编译单元信息 |
| uter.target.mirror.cmd_file_show() | NA                                  | NA                        | 命令：输出源文件信息  |

11 脚本端覆盖率

11.1 覆盖率

Uter 支持通过 Python 脚本开启/关闭覆盖率统计 和 获取/清除覆盖率信息。脚本端的覆盖率信息主要针对函数的覆盖情况，因此操作对象也都是函数。因此函数对象有一个成员\_Coverage，该成员提供了此函数覆盖率的各种操作接口，包括：开启覆盖统计 \_Coverage\_On()，关闭覆盖统计 \_Coverage\_Off()，清空统计数据 \_Coverage\_Reset()，获取统计数据 \_Coverage\_Count()，获取详情信息 \_Coverage\_Detail()。比如以下 C 代码：

```
int fun_base(int x)
{
    if (x == 1) {
        return 8;
    } else if (x == 2) {
        return 9;
    } else if (x == 3) {
        return 10;
    } else if (x == 4) {
        return 11;
    }
}
```



```

        } else {
            return 12;
        }
    }

/*gcc -g main.c -o test.bin*/

```

使用如下代码操作：

```

fun = uter.export.function("fun_base")

fun._Coverage._On()

print fun._Coverage._Count()
{'sum': 11, 'covs': 0}

ret = fun._Call(uter.int(1))
print ret._GetValue()
8

print fun._Coverage._Count()
{'sum': 11, 'covs': 4}

ret = fun._Call(uter.int(2))
print ret._GetValue()
9

print fun._Coverage._Count()
{'sum': 11, 'covs': 6}

print fun._Coverage._Detail()
1313:-/-:      return 0;
1314:-/-:}
1315:-/-:
1316:-/-:int fun_base(int x)
1317:1/1:{
1318:1/1:    if (x == 1) {
1319:1/1:        return 8;
1320:1/1:    } else if (x == 2) {
1321:1/1:        return 9;
1322:0/1:    } else if (x == 3) {
1323:0/1:        return 10;
1324:0/1:    } else if (x == 4) {
1325:0/1:        return 11;
1326:-/-:    } else {
1327:0/1:        return 12;
1328:-/-:    }
1329:1/1:}
1330:-/-:
=====Coverage information=====
Source file:/home/...../example_base/src/main.c
Coverage 6/11  54%

```

```
fun._Coverage._Reset()

print fun._Coverage._Count()
{'sum': 11, 'covs': 0}

fun._Coverage._Off()
```

## 11.2 编程手册

| 函数/成员/方法               | 参数 | 返回值  | 描述         |
|------------------------|----|--|------------|
| obj._Coverage._On()    | NA | NA   | 开启函数覆盖统计   |
| obj._Coverage._Off()   | NA | NA   | 关闭函数覆盖统计   |
| obj._Coverage._Reset() | NA | NA   | 清除函数覆盖统计信息 |
| obj._Coverage._Count() | NA | Python dict , key 'sum' 里保存着函数总函数 , key 'covs'里保存着被覆盖的行数 | 获取函数覆盖统计值  |
| obj._Coverage._Detal() | NA | Python str 覆盖详细信息  | 获取函数覆盖详情   |

## 12 UI 工具

### 12.1 程序浏览器

略

### 12.2 程序分析器

略

### 12.3 UI 端覆盖率

略

## 13 附录：术语

略

## 14 附录：Python 入门

略