CUnit 中文手册

作者业余时间翻译的资料，请转此文时附上原创地址,仅供大家参考。

有疑问或建议请发邮件至[naibiao.zhou@gmail.com](mailto:naibiao.zhou@gmail.com)

原作者网址：

<http://blog.sina.com.cn/s/blog_735e6dea0100wnxn.html>

目录

[一、CUnit 使用简介 4](#_Toc452549034)

[1.1. 描述 4](#_Toc452549035)

[1.2. 结构 5](#_Toc452549036)

[1.3. 一般用法 6](#_Toc452549037)

[1.4. CUnit API 版本２的变更 6](#_Toc452549038)

[二、 编写 CUnit 测试用例 7](#_Toc452549039)

[2.1. 测试函数 7](#_Toc452549040)

[2.2. CUnit 断言 8](#_Toc452549041)

[三、 测试注册簿 11](#_Toc452549042)

[3.1. 概要 11](#_Toc452549043)

[3.2. 内部结构 11](#_Toc452549044)

[3.3. 初始化 12](#_Toc452549045)

[3.4. 清理 12](#_Toc452549046)

[3.5. 其它测试注册簿操作函数 13](#_Toc452549047)

[四、管理测试用例和测试包 14](#_Toc452549048)

[4.1. 概要 14](#_Toc452549049)

[4.2. 添加测试包到测试注册簿 15](#_Toc452549050)

[4.3. 想测试包中添加测试用例 16](#_Toc452549051)

[4.4. 测试用例管理函数汇总 16](#_Toc452549052)

[五、 运行测试 18](#_Toc452549053)

[5.1. 概要 18](#_Toc452549054)

[5.2. 在CUnit中运行测试 19](#_Toc452549055)

[5.3. 自动模式 20](#_Toc452549056)

[5.4. 基本模式 21](#_Toc452549057)

[5.5. 控制台交互模式 22](#_Toc452549058)

[5.6. Curses交互模式 22](#_Toc452549059)

[5.7. 获得测试结果 23](#_Toc452549060)

[6、错误处理 25](#_Toc452549061)

[6.1. 概要 25](#_Toc452549062)

[6.2. CUnit 错误处理 25](#_Toc452549063)

[6.3. 框架上出错的行为 27](#_Toc452549064)

# 一、CUnit 使用简介

## 1.1. 描述

CUnit是一个使用C语言编写，管理和运行单元测试的系统。CUnit作为一个静态库连接到用户的测试代码当中。

CUnit使用一个简单接框架来构建测试结构，并且为测试公用数据类型提供了一组断言函数。初次之外，还为运行测试和生成的是报告提供了很多不同的接口。包括为代码控制的测试和生成报告提供了自动化测试接口，同时交互式接口是用户能够动态的运行测试和查看结果。

数据类型和函数在如下文件用为标准用户定义：

#include <CUnit/CUnit.h> 在用户的测试中使用的断言宏和包含其他的框架头文件。

#include <CUnit/CUError.h> 错误处理函数和数据类型，CUnit.h会自动包含此文件。

#include <CUnit/TestDB.h> 测试注册簿，测试包和测试的数据类型和函数的定义。

#include <CUnit/TestRun.h>  运行的是和处理结果的数据类型和函数的定义。

#include <CUnit/Automated.h> 自动生成XML文件模式接口定义。

#include <CUnit/Basic.h> 无交互的输出到stdout的基本测试模式接口定义。

#include <CUnit/Console.h> 与控制台交互模式接口定义。

#include <CUnit/CUCurses.h> Curses 图形用户界面交互接口定义。

#include <CUnit/Win.h>　 Windows 接口定义(尚未实现)。

## 1.2. 结构

CUnit是一个的独立于平台的测试框架，拥有丰富的接口。核心框架提供基本的测试注册簿，测试包和测试用例的管理服务。接口促进用户和框架交互以实现运行测试和查看结果的目的。

CUnit的组织结构和传统的单元测试框架相同。

                     　测试注册簿

                            |

             ------------------------------

             |                            |

          测试包 '1'      . . . .       测试包'N'

             |                            |

       ---------------             ---------------

       |             |             |             |

测试用例'11' ... 测试用例'1M'  测试用例'N1' ... 测试用例'NM'

单个的测试用例包含在测试包当中，测试包在测试注册簿中注册。

测试包包含在运行测试前自动调用的初始化函数和运行测试后自动调用的清理函数。

所有在测试注册簿当中的测试包/测试用例都可以单独调用或者有选择的运行。

## 1.3. 一般用法

CUnit框架的一个典型的使用顺序为：

1. 为测试编写函数(如果需要的话编写测试包的 init/cleanup 函数).
2. 初始化测试注册簿 - CU\_initialize\_registry()
3. 添加测试包到测试注册簿 - CU\_add\_suite()
4. 添加测试用例到的是测试包 - CU\_add\_test()
5. 用适当的接口函数运行测试, 例如. CU\_console\_run\_tests()
6. 清理测试注册簿 - CU\_cleanup\_registry()

## 1.4. CUnit API 版本２的变更

所有的公开的CUnit名字以CU\_ 开头。这可以帮助解决和用户代码中名字冲突的问题。说明以前的版本使用不带前缀的名字。早期API名字不建议使用但是新的版本依然支持老版本的API。要想使用旧版本的名字用户代码必须添加宏USE\_DEPRECATED\_CUNIT\_NAMES后再编译。

弃用的API函数将会在以后的章节中介绍。

# 二、 编写 CUnit 测试用例

## 2.1. 测试函数

一个 CUnit "测试" 是一个C 函数像:void test\_func(void)

除了测试函数不能修改CUnit框架(例如：添加测试或者测试用例，修改测试注册簿或者启动测试)外，对于测试函数的内容没有严格的规定。一个测试函数可以调用其它函数(这个函数也不会修改框架)。当测试运行时注册一个测试用例将会引起这个函数运行。

一个测试一个返回两个整数的最大值的程序的函数的例子可能会是这样的：

int maxi(int i1, int i2)

{

return (i1 > i2) ? i1 : i2;

}

void test\_maxi(void)

{

CU\_ASSERT(maxi(0,2) == 2);

CU\_ASSERT(maxi(0,-2) == 0);

CU\_ASSERT(maxi(2,2) == 2);

}

## 2.2. CUnit 断言

CUnit 提供一组断言来测试逻辑条件.断言的成功与失败会被框架跟踪并且在测试运行完成后可以在报告中查看结果。

每个断言测试一个逻辑条件，如果条件为FALSE断言会失败。测试函数是否继续运行取决于用户是否选定该断言是'xxx\_FATAL'版本。在这种情况下测试函数会中止并且立即返回。FATAL版本的断言在使用时要格外小心。即使正常的测试包清理函数没有受到影响，一旦FATAL断言失败，测试函数将不能清理现场。

还有一些特殊的断言可以在不执行逻辑测试的情况下注册一个通过测试或者失败测试。这些断言在测试流程控制和其它不需要逻辑测试的情况下非常有用。

void test\_longjmp(void)

{

jmp\_buf buf;

int i;

i = setjmp(buf);

if (i == 0) {

run\_other\_func();

CU\_PASS("run\_other\_func() succeeded.");

}

else

CU\_FAIL("run\_other\_func() issued longjmp.");

}

其它被注册的测试函数调用的函数可以自由的使用CUnit断言。这些断言对于调用函数非常有用。函数也可以调用FATAL版本的断言-失败会中止测试函数和它的整个调用链。

CUnit的断言定义如下：

#include <[CUnit/CUnit.h](http://cunit.sourceforge.net/doc/headers/CUnit.h)>

|  |  |
| --- | --- |
| **CU\_ASSERT**(int expression) **CU\_ASSERT\_FATAL**(int expression) **CU\_TEST**(int expression) **CU\_TEST\_FATAL**(int expression) | 断言 *expression* 为 TRUE (非零) |
| **CU\_ASSERT\_TRUE**(value) **CU\_ASSERT\_TRUE\_FATAL**(value) | 断言 *value* 为 TRUE (非零) |
| **CU\_ASSERT\_FALSE**(value) **CU\_ASSERT\_FALSE\_FATAL**(value) | 断言 *value* 为 FALSE (零) |
| **CU\_ASSERT\_EQUAL**(actual, expected) **CU\_ASSERT\_EQUAL\_FATAL**(actual, expected) | 断言*actual* = = *expected* |
| **CU\_ASSERT\_NOT\_EQUAL**(actual, expected)) **CU\_ASSERT\_NOT\_EQUAL\_FATAL**(actual, expected) | 断言*actual* != *expected* |
| **CU\_ASSERT\_PTR\_EQUAL**(actual, expected) **CU\_ASSERT\_PTR\_EQUAL\_FATAL**(actual, expected) | 断言pointers *actual* = = *expected* |
| **CU\_ASSERT\_PTR\_NOT\_EQUAL**(actual, expected) **CU\_ASSERT\_PTR\_NOT\_EQUAL\_FATAL**(actual, expected) | 断言pointers *actual* != *expected* |
| **CU\_ASSERT\_PTR\_NULL**(value) **CU\_ASSERT\_PTR\_NULL\_FATAL**(value) | 断言pointer *value* == NULL |
| **CU\_ASSERT\_PTR\_NOT\_NULL**(value) **CU\_ASSERT\_PTR\_NOT\_NULL\_FATAL**(value) | 断言pointer *value* != NULL |
| **CU\_ASSERT\_STRING\_EQUAL**(actual, expected) **CU\_ASSERT\_STRING\_EQUAL\_FATAL**(actual, expected) | 断言strings *actual* and *expected*are equivalent |
| **CU\_ASSERT\_STRING\_NOT\_EQUAL**(actual, expected) **CU\_ASSERT\_STRING\_NOT\_EQUAL\_FATAL**(actual, expected) | 断言strings *actual* and *expected*differ |
| **CU\_ASSERT\_NSTRING\_EQUAL**(actual, expected, count) **CU\_ASSERT\_NSTRING\_EQUAL\_FATAL**(actual, expected, count) | 断言*actual* 的前面*count*个字符 与*expected*相同 |
| **CU\_ASSERT\_NSTRING\_NOT\_EQUAL**(actual, expected, count) **CU\_ASSERT\_NSTRING\_NOT\_EQUAL\_FATAL**(actual, expected, count) | 断言*actual 的前面count个字符*与*expected* 不同 |
| **CU\_ASSERT\_DOUBLE\_EQUAL**(actual, expected, granularity) **CU\_ASSERT\_DOUBLE\_EQUAL\_FATAL**(actual, expected, granularity) | 断言 |*actual* - *expected*| <= |*granularity*| *数学库必须连接到程序当中* |
| **CU\_ASSERT\_DOUBLE\_NOT\_EQUAL**(actual, expected, granularity) **CU\_ASSERT\_DOUBLE\_NOT\_EQUAL\_FATAL**(actual, expected, granularity) | 断言 |*actual* - *expected*| > |*granularity*| *数学库必须连接到程序当中* |
| **CU\_PASS**(message) | 注册一个带有消息的通过断言. 在没有逻辑测试运行时. |
| **CU\_FAIL**(message) **CU\_FAIL\_FATAL**(message) | 注册一个带有消息的失败断言. 在没有逻辑测试运行时. |

# 三、 测试注册簿

## 3.1. 概要

#include <CUnit/TestDB.h> (自动被 <CUnit/CUnit.h>包含)

typedef struct CU\_TestRegistry

typedef CU\_TestRegistry\* CU\_pTestRegistry

CU\_ErrorCode CU\_initialize\_registry(void)

void CU\_cleanup\_registry(void)

CU\_pTestRegistry CU\_get\_registry(void)

CU\_pTestRegistry CU\_set\_registry(CU\_pTestRegistry pTestRegistry)

CU\_pTestRegistry CU\_create\_new\_registry(void)

void CU\_destroy\_existing\_registry(CU\_pTestRegistry\* ppRegistry)

## 3.2. 内部结构

测试注册簿是一个测试包和相关测试用例的仓库。CUnit维护一个活动测试注册簿，当用户添加一个测试包或测试用例时测试簿一同更新。测试包是用户选择运行所有活动注册簿中作测试中的一个。

CUnit 测试注册簿是在<CUnit/TestDB.h>中定义的结构体CU\_TestRegistry。CU\_TestRegistry包含测试注册簿当中测试包和测试用例的数量，和指向注册的测试包链表的头指针。

typedef struct CU\_TestRegistry

{

unsigned int uiNumberOfSuites;

unsigned int uiNumberOfTests;

CU\_pSuite pSuite;

} CU\_TestRegistry;

typedef CU\_TestRegistry\* CU\_pTestRegistry;

正常情况下用户只需要初始化测试注册簿，并且在使用之后清理测试注册簿。但是在需要时也可使用其它函数来操作测试注册簿。

## 3.3. 初始化

CU\_ErrorCode CU\_initialize\_registry(void)

活动CUnit测试注册簿必须再使用前进行初始化。用户应该在使用任何CUnit函数之前调用CU\_initialize\_registry()对测试注册簿进行初始化，否则将会产是程序崩溃。

返回值：

CUE\_SUCCESS

初始化成功.

CUE\_NOMEMORY

内存申请失败

## 3.4. 清理

void CU\_cleanup\_registry(void)

当测试完成时用户应该调用此函数进行清理和内存释放。并且这个函数是此次测试调用的最后一个CUnit函数(除非使用CU\_initialize\_registry() 或者CU\_set\_registry()恢复测试注册簿)。

不调用CU\_cleanup\_registry()函数将导致内存泄漏。此函数可以被多次调用而不会产生错误。此函数一旦被调用将会破坏所有在此测试注册簿当中的测试包和测试用例。指向测试包和测试用例的指针在调用此函数后不要再引用。

调用CU\_cleanup\_registry()函数将会影响到CUnit框架中维护的CU\_TestRegistry。析构用户使用的测试注册簿是用户的责任。通过调用CU\_destroy\_existing\_registry()可以实现此功能。或者先调用CU\_set\_registry()激活测试注册簿然后调用CU\_cleanup\_registry()进行清理 。

## 3.5. 其它测试注册簿操作函数

其它测试注册簿操作函数主要是为内部测试目的。一般用户可以找到并使用它们但是应该小心使用。

函数:

CU\_pTestRegistry CU\_get\_registry(void)

返回指向活动测试注册簿的指针。测试注册簿是一个CU\_TestRegistry类型的变量。不建议直接操作内部测试注册簿-应该使用API函数代替直接操作。CUnit框架拥有和维护测试注册簿，所以直接返回测试注册簿的指针可能导致CU\_cleanup\_registry() 或者 CU\_initialize\_registry()失效。

CU\_pTestRegistry CU\_set\_registry(CU\_pTestRegistry pTestRegistry)

用另外一个测试注册簿代替活动测试注册簿。返回上一个测试注册簿的指针。调用者负责清理此测试注册簿的工作。可以通过调用CU\_destroy\_existing\_registry()函数来完成。此测试注册簿可以通过调用CU\_set\_registry()激活后调用CU\_cleanup\_registry()清理。不要显示的清理已经被激活的测试注册簿。这样将会同一块内存多次释放和程序崩溃。

CU\_pTestRegistry CU\_create\_new\_registry(void)

创建一个新的测试注册簿并且返回指向这个测试注册簿的指针。新的测试注册簿不包含任何测试包和测试用例。调用者有责任用上面讲到的机制去清理此测试注册簿。

void CU\_destroy\_existing\_registry(CU\_pTestRegistry\* ppRegistry)

清理和释放特定测试注册簿的所有内存，包括任何注册的测试包和测试用例。对于已经注册并且激活的测试注册簿请勿调用此函数（例如：CU\_get\_registry()函数返回的CU\_pTestRegistry指针）。在调用CU\_cleanup\_registry()函数时将产生内存重复释放。此函数传入NULL指针将没有任何反应。

# 四、管理测试用例和测试包

为了测试用例可以被CUnit运行，必须添加一个测试包（suite），这个包将在测试注册簿中注册。

## 4.1. 概要

#include <CUnit/TestDB.h> (将会被<CUnit/CUnit.h>自动包含)

typedef struct CU\_Suite

typedef CU\_Suite\* CU\_pSuite

typedef struct CU\_Test

typedef CU\_Test\* CU\_pTest

typedef void (\*CU\_TestFunc)(void)

typedef int (\*CU\_InitializeFunc)(void)

typedef int (\*CU\_CleanupFunc)(void)

CU\_pSuite CU\_add\_suite(const char\* strName,

CU\_InitializeFunc pInit,

CU\_CleanupFunc pClean);

CU\_pTest CU\_add\_test(CU\_pSuite pSuite,

const char\* strName,

CU\_TestFunc pTestFunc);

typedef struct CU\_TestInfo

typedef struct CU\_SuiteInfo

CU\_ErrorCode CU\_register\_suites(CU\_SuiteInfo suite\_info[]);

CU\_ErrorCode CU\_register\_nsuites(int suite\_count, ...);

## 4.2. 添加测试包到测试注册簿

CU\_pSuite CU\_add\_suite(const char\* strName, CU\_InitializeFunc pInit, CU\_CleanupFunc pClean)

创建一个新的有指定名字，初始化函数和清理函数的测试包。新的测试包将在测试注册簿中注册。所以测试注册簿必须在添加任何一个测试包之前被初始化。现在的实现不支持创建独立于测试注册簿的测试包。

测试包的名字在测试注册簿的所有测试包中必须是唯一的。初始化和清理函数是指针函数在运行测试包中的所有测试用例前后调用，这两个函数时可选的，在不需要这两个或者其中一个的情况下在调用CU\_add\_suite()函数时可以设置为NULL。这两个函数使得测试包可以建立和毁掉测试用例使用的临时变量。这两个函数没有参数并且如果成功的话返回0。

返回指向新的测试包的指针，这个指针可以在向测试包中添加测试用例是使用。如果有错误则返回NULL并且框架将产生一个错误代码如下：

CUE\_SUCCESS

测试包创建成功.

CUE\_NOREGISTRY

测试注册簿尚未初始化.

CUE\_NO\_SUITENAME

strName 是NULL.

CUE\_DUP\_SUITE

测试包的名字已经存在

CUE\_NOMEMORY

内存申请失败.

## 4.3. 想测试包中添加测试用例

CU\_pTest CU\_add\_test(CU\_pSuite pSuite, const char\* strName, CU\_TestFunc pTestFunc)

创建一个有指定名字的测试用例和测试函数，并且注册到一个指定的测试包。测试包必须已经调用CU\_add\_suite()函数创建。现在的实现不支持创建独立于已经注册的测试包的测试用例。

测试用例的名字在一个注册包的所有测试用例当中是唯一的。测试函数不能是NULL，并且指向测试运行时被调用的函数。测试函数没有返回至和参数。

返回一个指向新策是用例的指针。如果在创建测试用例过程中有错误产生则返回NULL，并且框架将设置一个出错代码如下：

CUE\_SUCCESS

测试用例创建成功.

CUE\_NOSUITE

指定的测试包为NULL或者无效

CUE\_NO\_TESTNAME

strName 为 NULL.

CUE\_NO\_TEST

pTestFunc 为 NULL 或者无效.

CUE\_DUP\_TEST

测试用例名字已经存在.

CUE\_NOMEMORY

内存申请失败.

## 4.4. 测试用例管理函数汇总

#define CU\_ADD\_TEST(suite, test) (CU\_add\_test(suite, #test, (CU\_TestFunc)test))

此宏自动基于测试函数的名字生成一个唯一名字的测试用例，同时添加测试用例到指定的测试包，用户可以通过检查返回值确定是否成功。

CU\_ErrorCode CU\_register\_suites(CU\_SuiteInfo suite\_info[])

CU\_ErrorCode CU\_register\_nsuites(int suite\_count, ...)

对于拥有大量测试用例和测试包的测试结构来说，管理测试用例/测试包注册是一件乏味而且容易出错的工作，所以CUnit提供一套特殊的注册系统来帮助管理测试包和测试用例。主要的好处在于集中注册测试包和相关的测试用例，可以使检查用户编写代码错误的工作最小化。

测试用例首先是用CU\_TestInfo实体组织在一起的数组。(在 <CUnit/TestDB.h>中定义):

CU\_TestInfo test\_array1[] = {

{ "testname1", test\_func1 },

{ "testname2", test\_func2 },

{ "testname3", test\_func3 },

CU\_TEST\_INFO\_NULL,

};

数组中的每个元素包含一个单个测试用例，它有唯一的名字和测试函数。数组的最后一个元素的所有字段必须是NULL，CU\_TEST\_INFO\_NULL宏此时可以使用。测试用例数组中的测试用例将会被注册到同一个测试包当中。

测试包被定义为一个包含一个或多个CU\_SuiteInfo实体的数组(在 <CUnit/TestDB.h>中定义):

CU\_SuiteInfo suites[] = {

{ "suitename1", suite1\_init-func, suite1\_cleanup\_func, test\_array1 },

{ "suitename2", suite2\_init-func, suite2\_cleanup\_func, test\_array2 },

CU\_SUITE\_INFO\_NULL,

};

数组的每个元素包含一个测试包，它有唯一的名字，测试包初始化函数，测试包清理函数，和一个测试包的CU\_TestInfo数组。如果测试包不需要初始化或者清理 函数则可以将其置为NULL.数组必须以全为NULL的元素结尾，CU\_SUITE\_INFO\_NULL可以使用作为结尾。

CU\_SuiteInfo数组中的所有测试包将用一条语句注册：

CU\_ErrorCode error = CU\_register\_suites(suites);

如果在注册当中有错误将返回错误代码。错误代码与测试包注册，测试用例添加操作的错误代码相同。函数CU\_register\_nsuites()是为了满足有的用户希望用一条语句注册多个CU\_SuiteInfo数组而设计的：

CU\_ErrorCode error = CU\_register\_nsuites(2, suites1, suites2);

此函数接受多个CU\_SuiteInfo数组，第一个参数表明有多少个数组被传入。

# 五、 运行测试

## 5.1. 概要

#include <CUnit/Automated.h>

void CU\_automated\_run\_tests(void)

CU\_ErrorCode CU\_list\_tests\_to\_file(void)

void CU\_set\_output\_filename(const char\* szFilenameRoot)

#include <CUnit/Basic.h>

typedef enum CU\_BasicRunMode

CU\_ErrorCode CU\_basic\_run\_tests(void)

CU\_ErrorCode CU\_basic\_run\_suite(CU\_pSuite pSuite)

CU\_ErrorCode CU\_basic\_run\_test(CU\_pSuite pSuite, CU\_pTest pTest)

void CU\_basic\_set\_mode(CU\_BasicRunMode mode)

CU\_BasicRunMode CU\_basic\_get\_mode(void)

Void CU\_basic\_show\_failures(CU\_pFailureRecord pFailure)

#include <CUnit/Console.h>

void CU\_console\_run\_tests(void)

#include <CUnit/CUCurses.h>

void CU\_curses\_run\_tests(void)

#include <CUnit/TestRun.h> (included automatically by <CUnit/CUnit.h>)

unsigned int CU\_get\_number\_of\_suites\_run(void)

unsigned int CU\_get\_number\_of\_suites\_failed(void)

unsigned int CU\_get\_number\_of\_tests\_run(void)

unsigned int CU\_get\_number\_of\_tests\_failed(void)

unsigned int CU\_get\_number\_of\_asserts(void)

unsigned int CU\_get\_number\_of\_successes(void)

unsigned int CU\_get\_number\_of\_failures(void)

typedef struct CU\_RunSummary

typedef CU\_Runsummary\* CU\_pRunSummary

const CU\_pRunSummary CU\_get\_run\_summary(void)

typedef struct CU\_FailureRecord

typedef CU\_FailureRecord\* CU\_pFailureRecord

const CU\_pFailureRecord CU\_get\_failure\_list(void)

unsigned int CU\_get\_number\_of\_failure\_records(void)

## 5.2. 在CUnit中运行测试

CUnit 支持运行所有已注册测试包中的所有测试用例，单独的测试包和测试用例也可以运行，框架会跟踪测试包，测试用例和断言通过和失败的次数。测试结果即使有失败发生也会在每次新的测试运行时将被清除。

CUnit 提供测试包和测试用例运行控制的原始函数，但是大部分用户想使用简化的接口。这些接口处理用户与框架交互的细节，并且提供像测试输出和结果给客户。

在CUnit 库中有如下接口:

|  |  |  |
| --- | --- | --- |
| **Interface** | **Platform** | **Description** |
| [Automated](http://cunit.sourceforge.net/doc/running_tests.html#automated) | all | 非交互式输出 xml 文件 |
| [Basic](http://cunit.sourceforge.net/doc/running_tests.html#basic) | all | 非交互式可选输出到stdout |
| [Console](http://cunit.sourceforge.net/doc/running_tests.html#console) | all | 控制台交互方式 在用户的控制下运行 |
| [Curses](http://cunit.sourceforge.net/doc/running_tests.html#curses) | Linux/Unix | Curses交互在用户控制下运行 |

如果这些接口不足以满足用户的需求，用户也可以使用在<CUnit/TestRun.h>中定义的原始框架API。可以从原代码中看到各种各样的接口函数。

## 5.3. 自动模式

自动模式接口是非交互式的接口，客户端启动一次测试，测试结果将会输出的到一个XML文件当中。对于一组已注册的测试包和测试用例也可以将报告输出到一个XML文件当中。

自动模式接口 API有下面几个函数构成:

void CU\_automated\_run\_tests(void)

运行所有注册测试包的所有测试用例。测试结果将输出到一个名为ROOT-Results.xml的文件。文件名中的ROOT可以通过CU\_set\_output\_filename()函数设置，默认的名字为CUnitAutomated-Results.xml。如果每次测试没有设置ROOT文件名，新的测试结果将会覆盖上一次测试结果。

测试结果文件也可以是dtd文件（CUnit-Run.dtd）和XSL文件（CUnit-Run.xsl）。在代码的子目录和安装树中可以找到。

CU\_ErrorCode CU\_list\_tests\_to\_file(void)

把注册的测试包和测试用例列表输出到文件。输出文件的名字为ROT-Listing.xml。文件名中的ROOT可以通过CU\_set\_output\_filename()函数设置否则为CUnitAutomated。如果每次测试没有设置ROOT文件名，新的列表将会覆盖上一次的列表。

测试结果文件也可以是dtd文件（CUnit-Run.dtd）和XSL文件（CUnit-Run.xsl）。在代码的子目录和安装树中可以找到。

列表文件不会再调用CU\_automated\_run\_tests()后自动生成，客户端代码必须在需要时显式的申请。

void CU\_set\_output\_filename(const char\* szFilenameRoot)

设置测试结果和文件列表输出文件的名字。szFilenameRoot 再加上-Results.xml 或-Listing.xml,构成输出文件的名字。

## 5.4. 基本模式

基本模式接口也是非交互式的接口，结果将会输出到stdout。这套接口支持运行单个测试包和测试用例，并且允许客户代码控制在每次运行时输出的类型。这套接口提供了最为灵活的满足用户需求的简化了的CUnit API。

下面是公开的函数：

CU\_ErrorCode CU\_basic\_run\_tests(void)

运行所有的已注册的测试包和测试用例。返回在运行当中第一次发生错误时的错误代码。输出模式被当前的模式控制，可以通过调用CU\_basic\_set\_mode()函数改变当前输出模式。

CU\_ErrorCode CU\_basic\_run\_suite(CU\_pSuite pSuite)

运行一个指定测试包中的所有测试用例。返回在运行当中第一次发生错误时的错误代码。输出模式被当前的模式控制，可以通过调用CU\_basic\_set\_mode()函数改变当前输出模式。

CU\_ErrorCode CU\_basic\_run\_test(CU\_pSuite pSuite, CU\_pTest pTest)

运行一个指定测试包中的一个测试用例。返回在运行当中第一次发生错误时的错误代码。输出模式被当前的模式控制，可以通过调用CU\_basic\_set\_mode()函数改变当前输出模式。

void CU\_basic\_set\_mode(CU\_BasicRunMode mode)

设置基本模式的输出类型。

可选的类型有：

CU\_BRM\_NORMAL

失败和测试结果概要被打印到stdout

CU\_BRM\_SILENT

除了出错消息外不输出任何消息

CU\_BRM\_VERBOSE

输出全部的详细的测试结果

CU\_BasicRunMode CU\_basic\_get\_mode(void)

返回当前基本模式的输出类型。

void CU\_basic\_show\_failures(CU\_pFailureRecord pFailure)

打印所有失败的概到stdout。不依赖于输出模式。

## 5.5. 控制台交互模式

控制台接口是交互式接口。所有的客户需要启动一个控制台会话，并且用户交互的控制测试的运行。包括选择&运行已注册的测试包和测试用例并且查看测试结果。可以使用下面函数启动一个控制台会话。

void CU\_console\_run\_tests(void)

## 5.6. Curses交互模式

curses 接口是交互式接口. 所有的客户需要启动一个curses会话，并且用户交互的控制测试的运行。包括选择&运行注册的测试包和测试用例并且查看测试结果。使用这个接口需要将ncurses 库连接到应用程序当中。可以使用下面函数启动一个curses会话。

void CU\_curses\_run\_tests(void)

## 5.7. 获得测试结果

这些接口呈现测试结果，但是客户代码有时想要直接访问测试结果，包括运行测试的个数，和记录失败细节的失败报告列表。测试结果在每次新的测试运行起来时被覆盖或者测试注册簿初始化和清理时被覆盖。

访问测试结果的函数有:

unsigned int CU\_get\_number\_of\_suites\_run(void)

unsigned int CU\_get\_number\_of\_suites\_failed(void)

unsigned int CU\_get\_number\_of\_tests\_run(void)

unsigned int CU\_get\_number\_of\_tests\_failed(void)

unsigned int CU\_get\_number\_of\_asserts(void)

unsigned int CU\_get\_number\_of\_successes(void)

unsigned int CU\_get\_number\_of\_failures(void)

这些函数报告在最近一次测试中运行或者失败的测试包，测试用例和断言的计数。一个测试包在他的初始化和清理函数返回非NULL时被认为是失败。一个测试用例在任何一个断言失败时被认为失败。最后三个函数都涉及到断言自身。

可以使用下面的方法返回注册的测试包和测试用例的个数：

CU\_get\_registry()−>uiNumberOfSuites

CU\_get\_registry()−>uiNumberOfTests

const CU\_pRunSummary CU\_get\_run\_summary(void)

一次性返回所有测试结果的计数。返回值指向一个包含所有计数的结构体。这个结构体在<CUnit/TestRun.h>定义(被<CUnit/CUnit.h>自动包含)。

typedef struct CU\_RunSummary

{

unsigned int nSuitesRun;

unsigned int nSuitesFailed;

unsigned int nTestsRun;

unsigned int nTestsFailed;

unsigned int nAsserts;

unsigned int nAssertsFailed;

unsigned int nFailureRecords;

} CU\_RunSummary;

typedef CU\_Runsummary\* CU\_pRunSummary;

返回值指向的数据结构的变量属于框架，所有用户不应该回收或者修改它。在另外一个测试启动时这个指针式无效的。

const CU\_pFailureRecord CU\_get\_failure\_list(void)

返回一个记录所有在最近一次测试运行当中所有失败的链表（NULL或者失败）。返回值的数据类型在<CUnit/TestRun.h>定义(被<CUnit/CUnit.h>自动包含)。每个失败记录包括定位失败和失败性质的信息：

typedef struct CU\_FailureRecord

{

unsigned int uiLineNumber;

char\* strFileName;

char\* strCondition;

CU\_pTest pTest;

CU\_pSuite pSuite;

struct CU\_FailureRecord\* pNext;

struct CU\_FailureRecord\* pPrev;

} CU\_FailureRecord;

typedef CU\_FailureRecord\* CU\_pFailureRecord;

返回值指向的数据结构的变量属于框架，所有用户不应该回收或者修改它。在另外一个测试启动时这个指针式无效的。

unsigned int CU\_get\_number\_of\_failure\_records(void)

返回由CU\_get\_failure\_list()返回的链表中的CU\_FailureRecords的个数。这个数可能比失败的断言多，因为测试包的初始化和清理失败也包含其中。

# 6、错误处理

## 6.1. 概要

#include <CUnit/CUError.h> (自动被<CUnit/CUnit.h>包含)

typedef enum CU\_ErrorCode

CU\_ErrorCode CU\_get\_error(void);

const char\* CU\_get\_error\_msg(void);

typedef enum CU\_ErrorAction

void CU\_set\_error\_action(CU\_ErrorAction action);

CU\_ErrorAction CU\_get\_error\_action(void);

## 6.2. CUnit 错误处理

大多数CUnit函数设置一个错误代码用来指示框架的错误状态。有些函数返回错误代码，其他的则设置错误代码返回其他值。框架提供两个函数用来检查框架的错误状态：

CU\_ErrorCode CU\_get\_error(void)

const char\* CU\_get\_error\_msg(void)

第一个函数返回错误代码本身，第二个函数返回一个描述错误状态的消息。错误代码是一个在<CUnit/CUError.h>中定义的枚举类型CU\_ErrorCode。下面是错误代码值的定义：

错误代码的值

描述

CUE\_SUCCESS

没有错误.

CUE\_NOMEMORY

内存分配失败

CUE\_NOREGISTRY

测试注册簿未初始化

CUE\_REGISTRY\_EXISTS

在没有调用CU\_cleanup\_registry()函数的情况下试图调用CU\_set\_registry()函数

CUE\_NOSUITE

CU\_pSuite 指针是NULL.

CUE\_NO\_SUITENAME

CU\_Suite名字未指定

CUE\_SINIT\_FAILED

测试包初始化失败

CUE\_SCLEAN\_FAILED

测试包清理错误

CUE\_DUP\_SUITE

测试包重名

CUE\_NOTEST

CU\_pTest 指针是NULL.

CUE\_NO\_TESTNAME

CU\_Test 名字未指定

CUE\_DUP\_TEST

测试用例重名

CUE\_TEST\_NOT\_IN\_SUITE

测试用例没有注册到指定的测试包中

CUE\_FOPEN\_FAILED

文件打开失败

CUE\_FCLOSE\_FAILED

文件关闭失败

CUE\_BAD\_FILENAME

错误的文件名 (NULL,空,不存在, 等等.).

CUE\_WRITE\_ERROR

写文件错误

## 6.3. 框架上出错的行为

默认的行为是当错误发生时错误代码将会被设置并且继续执行下去。有些时候客户更喜欢在出错是停下来或者让测试应用程序退出。这样的行为用户可以设置。下面的函数就是为此设计的：

void CU\_set\_error\_action(CU\_ErrorAction action)

CU\_ErrorAction CU\_get\_error\_action(void)

错误动作代码是在<CUnit/CUError.h>中定义的枚举类型CU\_ErrorAction。下面是错误动作代码的定义：

错误动作代码值

描述

CUEA\_IGNORE

当错误产生式继续运行。

CUEA\_FAIL

错误产生时停止运行。

CUEA\_ABORT

错误产生时终止应用程序。