CUnit Progammers Guide

1. Introduction to Unit Testing with CUnit

1.1. Description

CUnit is a system for writing, administering, and running unit tests in C. It is built as a static library which is linked with the user's testing code.

CUnit 是用于在 C 中编写，管理和运行单元测试的系统。它被构建为与用户测试代码相关联的静态库。

CUnit uses a simple framework for building test structures, and provides a rich set of assertions for testing common data types. In addition, several different interfaces are provided for running tests and reporting results. These include automated interfaces for code-controlled testing and reporting, as well as interactive interfaces allowing the user to run tests and view results dynamically.

CUnit 使用简单的框架来构建测试结构，并提供了一组丰富的断言来测试常见的数据类型。 此外，还提供了几个不同的界面来运行测试和报告结果。 这些包括用于代码控制测试和报告的自动化界面，以及允许用户动态运行测试和查看结果的交互式界面。

The data types and functions useful to the typical user are declared in the following header files:

对典型用户有用的数据类型和功能在以下头文件中断言：

Header File	Description
#include <CUnit/CUnit.h>	ASSERT macros for use in test cases, and includes other framework headers.
#include <CUnit/CUError.h>	Error handing functions and data types. *Included automatically by CUnit.h.*
#include <CUnit/TestDB.h>	Data type definitions and manipulation functions for the test registry, suites, and tests. *Included automatically by CUnit.h.*
#include <CUnit/TestRun.h>	Data type definitions and functions for running tests and retrieving results. *Included automatically by CUnit.h.*
#include <CUnit/Automated.h>	Automated interface with xml output.
#include <CUnit/Basic.h>	Basic interface with non-interactive output

	to stdout.
#include <CUnit/Console.h>	Interactive console interface.
#include <CUnit/CUCurses.h>	Interactive console interface (*nix).
#include <CUnit/Win.h>	Windows interface (not yet implemented).

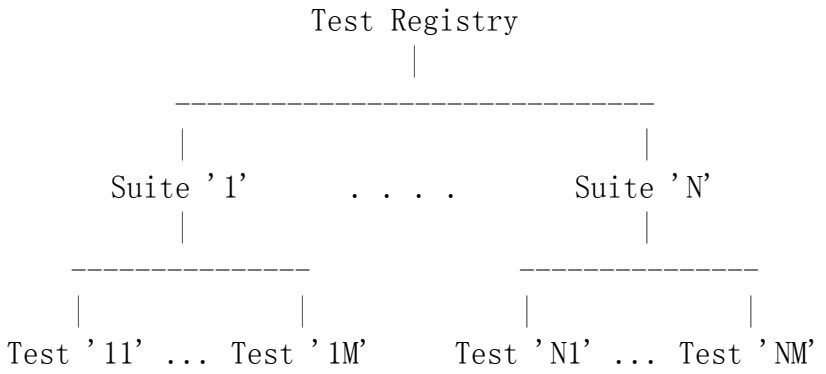
1.2. Structure

CUnit is a combination of a platform-independent framework with various user interfaces. The core framework provides basic support for managing a test registry, suites, and test cases. The user interfaces facilitate interaction with the framework to run tests and view results.

CUnit 是与平台无关的框架与各种用户界面的组合。 核心框架为管理测试注册表，套件和测试用例提供了基本的支持。 用户界面促进与框架的交互以运行测试和查看结果。

CUnit is organized like a conventional unit testing framework:

CUnit 是像传统的单元测试框架一样被组织起来：



Individual test cases are packaged into suites, which are registered with the active test registry. Suites can have setup and teardown functions which are automatically called before and after running the suite's tests. All suites/tests in the registry may be run using a single function call, or selected suites or tests can be run

单个测试用例被打包成套件，它们已经在活动测试注册表中注册。 套件可以有运行套件测试之前和之后自动调用的设置和拆卸功能。 注册表中的所有套件/测试可以使用单个函数调用运行，或者可以运行选定的套件或测试.

1.3. General Usage

A typical sequence of steps for using the CUnit framework is:

1. Write functions for tests (and suite init/cleanup if necessary).
2. Initialize the test registry - CU_initialize_registry()
3. Add suites to the test registry - CU_add_suite()
4. Add tests to the suites - CU_add_test()
5. Run tests using an appropriate interface, e.g. CU_console_run_tests
6. Cleanup the test registry - CU_cleanup_registry

使用 **CUnit** 框架的典型步骤是：

1.编写测试功能（如果需要，可以使用套件初始化/清理）。

2.初始化测试注册表 - CU_initialize_registry（）

3.将套件添加到测试注册表 - CU_add_suite（）

4.向套件添加测试 - CU_add_test（）

5.使用适当的界面运行测试，例如 CU_console_run_tests

6.清理测试注册表 - CU_cleanup_registry

1.4. Changes to the CUnit API in Version 2

All public names in CUnit are now prefixed with 'CU_'. This helps minimize clashes with names in user code. Note that earlier versions of CUnit used different names without this prefix. The older API names are deprecated but still supported. To use the older names, user code must now be compiled with USE_DEPRECATED_CUNIT_NAMES defined.

CUnit 中的所有公共名称现在都以'CU_'作为前缀。 这有助于最大程度地减少与用户代码中的名称的冲突。 请注意，早期版本的 CUnit 使用不同的名称，没有这个前缀。 旧的 API 名称已被弃用，但仍然受支持。 要使用较旧的名称，现在必须使用定义的 USE_DEPRECATED_CUNIT_NAMES 来编译用户代码。

The deprecated API functions are described in the appropriate sections of the documentation.

已弃用的 API 函数在文档的相应部分进行了说明。

2. Writing CUnit Test Cases

2.1. Test Functions

A CUnit "test" is a C function having the signature:

void test_func(void)

There are no restrictions on the content of a test function, except that it should not modify the CUnit framework (e.g. add suites or tests, modify the test registry, or initiate a test run). A test function may call other functions (which also may not modify the framework). Registering a test will cause it's function to be run when the test is run.

对测试功能的内容没有限制，除了它不应该修改 CUnit 框架（例如添加套件或测试，修改测试注册表或启动测试运行）。 测试功能可以调用其他功能（也可能不会修改框架）。 注册测试将导致运行测试时运行它的功能。

An example test function for a routine that returns the maximum of 2 integers might look like:

用于返回最大 2 个整数的例程的示例测试函数可能如下所示：

```
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 Assertions

CUnit provides a set of assertions for testing logical conditions. The success or failure of these assertions is tracked by the framework, and can be viewed when a test run is complete.

CUnit 提供了一组用于测试逻辑条件的断言。 这些断言的成功或失败是由框架跟踪的，并且可以在测试运行完成时查看。

Each assertion tests a single logical condition, and fails if the condition evaluates to **FALSE**. Upon failure, the test function continues unless the user chooses the '**xxx_FATAL**' version of an assertion. In that case, the test function is aborted and returns immediately.

每个断言测试单个逻辑条件，如果条件求值为 FALSE，则失败。 失败后，测试功能将继续进行，除非用户选择“xxx_FATAL”版本的断言。 在这种情况下，测试功能将中止并立即返回。

FATAL versions of assertions should be used with caution!

FATAL 版本的断言应谨慎使用！

There is no opportunity for the test function to clean up after itself once a FATAL assertion fails. The normal suite cleanup function is not affected, however.

一旦 FATAL 断言失败，测试功能就没有机会自己清理。 然而，正常的套件清理功能不受影响。

There are also special "assertions" for registering a pass or fail with the framework without performing a logical test. These are useful for testing flow of control or other conditions not requiring a logical test:

还有一些特殊的“断言”，用于在不执行逻辑测试的情况下在框架中注册通过或失败。这些可用于测试流程的控制或不需要逻辑测试的其他情形。

```
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.");
}
```

Other functions called by a registered test function may use the CUnit assertions freely. These assertions will be counted for the calling function. They may also use FATAL versions of assertions - failure will abort the original test function and its entire call chain.

注册测试函数调用的其他函数可以自由地使用 CUnit 断言。 这些断言将被计入调用函数。 他们也可能使用 FATAL 版本的断言 - 失败将中止原始测试功能及其整个调用链。

The assertions defined by CUnit are:

CUnit 定义的断言是：

#include <CUnit/CUnit.h>

CU_ASSERT(int expression) **CU_ASSERT_FATAL**(int expression) **CU_TEST**(int expression) **CU_TEST_FATAL**(int expression)	Assert that *expression* is TRUE (non-zero)
CU_ASSERT_TRUE(value) **CU_ASSERT_TRUE_FATAL**(value)	Assert that *value* is TRUE (non-zero)
CU_ASSERT_FALSE(value) **CU_ASSERT_FALSE_FATAL**(value)	Assert that *value* is FALSE (zero)
CU_ASSERT_EQUAL(actual, expected) **CU_ASSERT_EQUAL_FATAL**(actual, expected)	Assert that *actual* == *expected*
CU_ASSERT_NOT_EQUAL(actual, expected)) **CU_ASSERT_NOT_EQUAL_FATAL** (actual, expected)	Assert that *actual* != *expected*
CU_ASSERT_PTR_EQUAL(actual, expected) **CU_ASSERT_PTR_EQUAL_FATAL** (actual, expected)	Assert that pointers *actual* == *expected*
CU_ASSERT_PTR_NOT_EQUAL(actual, expected) **CU_ASSERT_PTR_NOT_EQUAL_FATAL**(actual, expected)	Assert that pointers *actual* != *expected*
CU_ASSERT_PTR_NULL(value) **CU_ASSERT_PTR_NULL_FATAL**(value)	Assert that pointer *value* == NULL
CU_ASSERT_PTR_NOT_NULL(value) **CU_ASSERT_PTR_NOT_NULL_FATAL**(value)	Assert that pointer *value* != NULL
CU_ASSERT_STRING_EQUAL(actual, expected) **CU_ASSERT_STRING_EQUAL_FATAL**(actual, expected)	Assert that strings *actual* and *expected* are equivalent

CU_ASSERT_STRING_NOT_EQUAL(actual, expected) **CU_ASSERT_STRING_NOT_EQUAL_FATAL**(actual, expected)	Assert that strings *actual* and *expected* differ
CU_ASSERT_NSTRING_EQUAL(actual, expected, count) **CU_ASSERT_NSTRING_EQUAL_FATAL**(actual, expected, count)	Assert that 1st count chars of *actual* and *expected* are the same
CU_ASSERT_NSTRING_NOT_EQUAL(actual, expected, count) **CU_ASSERT_NSTRING_NOT_EQUAL_FATAL**(actual, expected, count)	Assert that 1st count chars of *actual* and *expected* differ
CU_ASSERT_DOUBLE_EQUAL(actual, expected, granularity) **CU_ASSERT_DOUBLE_EQUAL_FATAL**(actual, expected, granularity)	Assert that |*actual* - *expected*| <= |*granularity*| *Math library must be linked in for this assertion.*
CU_ASSERT_DOUBLE_NOT_EQUAL(actual, expected, granularity) **CU_ASSERT_DOUBLE_NOT_EQUAL_FATAL**(actual, expected, granularity)	Assert that |*actual* - *expected*| > |*granularity*| *Math library must be linked in for this assertion.*
CU_PASS(message)	Register a passing assertion with the specified message. No logical test is performed.
CU_FAIL(message) **CU_FAIL_FATAL**(message)	Register a failed assertion with the specified message. No logical test is performed.

2.3. Depecated v1 Assertions

The following assertions are deprecated as of version 2. To use these assertions, user code must be compiled with USE_DEPRECATED_CUNIT_NAMES defined. Note that they behave the same as in version 1 (issue a 'return' statement upon failure).

以下变量和函数自版本 2 以来已被弃用。要使用这些不推荐使用的名称，必须使用定义的 USE_DEPRECATED_CUNIT_NAMES 编译用户代码。

#include <CUnit/CUnit.h>

Deprecated Name	Equivalent New Name
ASSERT	CU_ASSERT_FATAL
ASSERT_TRUE	CU_ASSERT_TRUE_FATAL
ASSERT_FALSE	CU_ASSERT_FALSE_FATAL
ASSERT_EQUAL	CU_ASSERT_EQUAL_FATAL
ASSERT_NOT_EQUAL	CU_ASSERT_NOT_EQUAL_FATAL
ASSERT_PTR_EQUAL	CU_ASSERT_PTR_EQUAL_FATAL
ASSERT_PTR_NOT_EQUAL	CU_ASSERT_PTR_NOT_EQUAL_FATAL
ASSERT_PTR_NULL	CU_ASSERT_PTR_NULL_FATAL
ASSERT_PTR_NOT_NULL	CU_ASSERT_PTR_NOT_NULL_FATAL
ASSERT_STRING_EQUAL	CU_ASSERT_STRING_EQUAL_FATAL
ASSERT_STRING_NOT_EQUAL	CU_ASSERT_STRING_NOT_EQUAL_FATAL
ASSERT_NSTRING_EQUAL	CU_ASSERT_NSTRING_EQUAL_FATAL
ASSERT_NSTRING_NOT_EQUAL	CU_ASSERT_NSTRING_NOT_EQUAL_FATAL
ASSERT_DOUBLE_EQUAL	CU_ASSERT_DOUBLE_EQUAL_FATAL
ASSERT_DOUBLE_NOT_EQUAL	CU_ASSERT_DOUBLE_NOT_EQUAL_FATAL

3. The Test Registry

3.1. Synopsis

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

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. Internal Structure

The test registry is the repository for suites and associated tests. CUnit maintains an active test registry which is updated when the user adds a suite or test. The suites in this active registry are the ones run when the user chooses to run all tests.

测试注册表是套件和相关测试的存储库。 CUnit 维护一个活动的测试注册表，当用户添加套件或测试时，它将被更新。 此活动注册表中的套件是用户选择运行所有测试时运行的套件。

The CUnit test registry is a data structure CU_TestRegistry declared in <CUnit/TestDB.h>. It includes fields for the total numbers of suites and tests stored in the registry, as well as a pointer to the head of the linked list of registered suites.

CUnit 测试注册表是在<CUnit / TestDB.h>中断言的一个数据结构 CU_TestRegistry。 它包括存储在注册表中的套件总数和测试的字段，以及指向注册套件链接列表头部的指针。

```
typedef struct CU_TestRegistry
  {
    unsigned int uiNumberOfSuites;
    unsigned int uiNumberOfTests;
    CU_pSuite    pSuite;
  } CU_TestRegistry;

  typedef CU_TestRegistry* CU_pTestRegistry;
```

The user normally only needs to initialize the registry before use and clean up afterwards. However, other functions are provided to manipulate the registry when necessary.

用户通常只需要在使用前初始化注册表，在使用后进行清理。 但是在必要时，其他函数将被提供来操作注册表。

3.3. Initialization

CU_ErrorCode **CU_initialize_registry**(void)

The active CUnit test registry must be initialized before use. The user should call CU_initialize_registry() before calling any other CUnit functions. Failure to do so will likely result in a crash.

活动 CUnit 测试注册表在使用前必须进行初始化。 在调用任何其他 CUnit 函数之前，用户应调用 CU_initialize_registry（）。 否则可能导致崩溃。

An error status code is returned:

CUE_SUCCESS	initialization was successful.
CUE_NOMEMORY	memory allocation failed.

3.4. Cleanup

void **CU_cleanup_registry**(void)

When testing is complete, the user should call this function to clean up and release memory used by the framework. This should be the last CUnit function called (except for restoring the test registry using CU_initialize_registry() or CU_set_registry()).

测试完成后，用户应该调用此功能来清理和释放框架使用的内存。 这应该是调用的最后一个 CUnit 函数（除了使用 CU_initialize_registry（）或 CU_set_registry（）还原测试注册表）。

Failure to call CU_cleanup_registry() will result in memory leaks. It may be called more than once without creating an error condition. *Note that this function will destroy all suites (and associated tests) in the registry.* Pointers to registered suites and tests should not be dereferenced after cleaning up the registry.

未调用 CU_cleanup_registry（）将导致内存泄漏。 它可能会被多次调用，而不会创建错误条件。 请注意，此功能将销毁注册表中的所有套件（和关联的测试）。 清理注册表后，不应取消注册注册套件和测试的指针。

Calling CU_cleanup_registry() will only affect the internal CU_TestRegistry maintained by the CUnit framework. Destruction of any other test registries owned by the user are the responsibility of the user. This can be done explictly by calling CU_destroy_existing_registry(), or implicitly by making the registry active using CU_set_registry() and calling CU_cleanup_registry() again.

调用 CU_cleanup_registry（）将仅影响 CUnit 框架维护的内部的 CU_TestRegistry。 用户拥有的任何其他测试注册表的销毁都是用户的责任。 这可以通过调用 CU_destroy_existing_registry（）来显式地完成；或通过使用 CU_set_registry（）使注册表激活，并再次调用 CU_cleanup_registry（）隐式地完成。

3.5. Other Registry Functions

Other registry functions are provided primarily for internal and testing purposes. However, general users may find use for them and should

be aware of them.

其他注册表函数主要用于内部和测试目的。 然而，一般用户可能会使用它们，应该意了解它们。

These include:

(1) CU_pTestRegistry **CU_get_registry**(void)

Returns a pointer to the active test registry. The registry is a variable of data type CU_TestRegistry. Direct manipulation of the internal test registry is not recommended - API functions should be used instead. The framework maintains ownership of the registry, so the returned pointer will be invalidated by a call to CU_cleanup_registry() or CU_initialize_registry().

返回指向活动测试注册表的指针。注册表是一个 CU_TestRegistry 数据类型的变量。不建议直接操作内部测试注册表， 应该使用 API 函数来替代。 框架维护注册表的所有权，所以通过调用 CU_cleanup_registry（）或 CU_initialize_registry（）返回的指针将将是无效的。

(2) CU_pTestRegistry **CU_set_registry**(CU_pTestRegistry pTestRegistry)

Replaces the active registry with the specified one. A pointer to the previous registry is returned. ***It is the caller's responsibility to destroy the old registry***. This can be done explictly by calling CU_destroy_existing_registry() for the returned pointer. Alternatively, the registry can be made active using CU_set_registry() and destroyed implicitly when CU_cleanup_registry() is called. Care should be taken not to explicitly destroy a registry that is set as the active one. This can result in multiple frees of the same memory and a likely crash.

用指定的注册表替换活动注册表。指向上一个注册表的指针被返回。 调用者有责任注销旧的注册表。这可以通过为得到返回指针而调用 CU_destroy_existing_registry（）时显式地完成。 或者，可以使用 CU_set_registry（）激活注册表，并在调用 Cuc_cleanup_registry（）时隐式地注销。 应注意的是，不要显式地销毁设置为活动注册表的注册表。 这可能导致同一内存的多次释放和可能的崩溃。

(3) CU_pTestRegistry **CU_create_new_registry**(void)

Creates a new registry and returns a pointer to it. The new registry will not contain any suites or tests. It is the caller's responsibility to destroy the new registry by one of the mechanisms described previously.

创建一个新的注册表并返回一个指针。 新的注册表不会包含任何套件或测试。 调用者有责任通过以前描述机制之一（显式方式或隐式方式）来销毁新的注册表。

(4) void **CU_destroy_existing_registry**(CU_pTestRegistry* ppRegistry)

Destroys and frees all memory for the specified test registry, including any registered suites and tests. This function should not be called for a registry which is set as the active test registry (**e.g.** a CU_pTestRegistry pointer retrieved using CU_get_registry()). This will result in a multiple free of the same memory when CU_cleanup_registry() is called.

销毁并释放指定测试注册表的所有内存，包括任何已注册的套件和测试。 此函数不应该被设置为活动测试注册表的注册表（例如使用 CU_get_registry（）检索到的 CU_pTestRegistry 指针）所调用。 这将导致当调用 CU_cleanup_registry（）时同一内存的多次释放。

Calling this function with NULL has no effect.

3.6. Deprecated v1 Data Types & Functions

The following data types and functions are deprecated as of version 2. To use these deprecated names, user code must be compiled with USE_DEPRECATED_CUNIT_NAMESdefined.

以下变量和函数自版本 2 以来已被弃用。要使用这些不推荐使用的名称，必须使用定义的 USE_DEPRECATED_CUNIT_NAMES 编译用户代码。

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

Deprecated Name	Equivalent New Name
_TestRegistry	CU_TestRegistry
_TestRegistry.uiNumberOfGroups PTestRegistry->uiNumberOfGroups	CU_TestRegistry.uiNumberOfSuites CU_pTestRegistry->uiNumberOfSuites
_TestRegistry.pGroup PTestRegistry->pGroup	CU_TestRegistry.pSuite CU_pTestRegistry->pSuite
PTestRegistry	CU_pTestRegistry
initialize_registry()	CU_initialize_registry()
cleanup_registry()	CU_cleanup_registry()
get_registry()	CU_get_registry()
set_registry()	CU_set_registry()

4. Managing Tests & Suites

In order for a test to be run by CUnit, it must be added to a test

collection (suite) which is registered with the test registry.

为了测试能在 CUnit 中运行，它必须被添加到测试集合（suite）中。测试 suite 通过测试注册表中注册。

4.1. Synopsis

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

```
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. Adding Suites to the Registry

```
CU_pSuite   CU_add_suite( const char*     strName,
                          CU_InitializeFunc    pInit,
                          CU_CleanupFunc    pClean )
```

Creates a new test collection (suite) having the specified name, initialization function, and cleanup function. The new suite is registered with (and owned by) the test registry, so the registry must be initialized before adding any suites. The current implementation does not support the creation of suites independent of the test registry.

创建具有指定名称、初始化函数和清除功能的新测试集合（suite）。 新套件已注册（并拥有）测试注册表，因此注册表必须在添加任何套件之前初始化。 当前的实现不支持独立于测试注册表的套件的创建。

The suite's name must be unique among all suites in the registry. The initialization and cleanup functions are optional, and are passed as pointers to functions to be called before and after running the tests contained in the suite. This allows the suite to set up and tear down temporary fixtures to support running the tests. These functions take no arguments and should return zero if they are completed successfully (non-zero otherwise). If a suite does not require one or both of these functions, pass **NULL** to CU_add_suite().

套件的名称在注册表中的所有套件中必须是唯一的。 初始化和清理函数是可选的，并且作为指针传递至包含在套件中的在测试之前和测试之后所被调用的函数之中。 这允许套件启动和注销临时的测试夹具（test fixture）以支持运行测试。 这些函数没有参数，如果成功完成，则返回零（否则为非零）。 如果一个套件不需要这些函数中的一个或两个，则将 NULL 传递给 CU_add_suite（）。

A pointer to the new suite is returned, which is needed for adding tests to the suite. If an error occurs, NULL is returned and the framework error code is set to one of the following:

返回指向新套件的指针，这是将套件添加到套件中所需要的。 如果发生错误，则返回 NULL，并将框架错误代码设置为以下之一：

CUE_SUCCESS	suite creation was successful.
CUE_NOREGISTRY	the registry has not been initialized.
CUE_NO_SUITENAME	strName was NULL.
CUE_DUP_SUITE	the suite's name was not unique.
CUE_NOMEMORY	memory allocation failed.

4.3. Adding Tests to Suites

```
CU_pTest    CU_add_test( CU_pSuite      pSuite,
                         const char*    strName,
                         CU_TestFunc    pTestFunc )
```

Creates a new test having the specified name and test function, and registers it with the specified suite. The suite must already have been created using CU_add_suite(). The current implementation does not support the creation of tests independent of a registered suite.

创建具有指定名称和测试功能的新测试，并将其注册到指定的套件中。 该套件必须已经使用 CU_add_suite（）创建。 当前的实现不支持独立于注册套件的测试的创建。

The test's name must be unique among all tests added to a single suite. The test function cannot be NULL, and points to a function to be called when the test is run. Test functions have neither arguments nor return values.

测试名称在添加到单个套件的所有测试中必须是唯一的。 测试函数不能为 NULL，并指向运行测试时要调用的函数。 测试函数既没有参数也没有返回值。

A pointer to the new test is returned. If an error occurs during creation of the test, NULL is returned and the framework error code is set to one of the following:

返回指向新测试的指针。 如果在创建测试期间发生错误，则返回 NULL，并将框架错误代码设置为以下之一：

CUE_SUCCESS	suite creation was successful.
CUE_NOSUITE	the specified suite was NULL or invalid.
CUE_NO_TESTNAME	strName was NULL.
CUE_NO_TEST	pTestFunc was NULL or invalid.
CUE_DUP_TEST	the test's name was not unique.
CUE_NOMEMORY	memory allocation failed.

4.4. Shortcut Methods for Managing Tests

```
#define  CU_ADD_TEST( suite, test )   ( CU_add_test ( suite, #test,
                                        ( CU_TestFunc ) test))
```

This macro automatically generates a unique test name based on the test function name, and adds it to the specified suite. The return value should be checked by the user to verify success.

该宏根据测试功能名称自动生成唯一的测试名称，并将其添加到指定的套件中。 返回值应由用户检查，以验证成功。

```
CU_ErrorCode  CU_register_suites( CU_SuiteInfo  suite_info[ ] )
CU_ErrorCode  CU_register_nsuites( int  suite_count, ... )
```

For large test structures with many tests and suites, managing test/suite associations and registration is tedious and error-prone. CUnit provides a special registration system to help manage suites and tests. It's primary benefit is to centralize the registration of suites

and associated tests, and to minimize the amount of error checking code the user needs to write.

对于具有许多测试和套件的大型测试结构，管理测试/套件的关联和注册是乏味且容易出错的。 CUnit 提供了一个特殊的注册系统来帮助管理套件和测试。 主要的优点是集中注册套件和相关测试，并最大限度地减少用户需要编写的错误检查代码的数量。

Test cases are first grouped into arrays of CU_TestInfo instances (defined in <CUnit/TestDB.h>):

```
CU_TestInfo   test_array1[ ] = {
      { "testname1", test_func1 },
      { "testname2", test_func2 },
      { "testname3", test_func3 },
      CU_TEST_INFO_NULL,
};
```

Each array element contains the (unique) name and test function for a single test case. The array must end with an element holding NULL values, which the macro CU_TEST_INFO_NULL conveniently defines. The test cases included in a single CU_TestInfo array form the set of tests that will be registered with a single test suite.

每个数组元素包含单个测试用例的（唯一）名称和测试函数。 数组必须以包含 NULL 值的元素结束，宏 CU_TEST_INFO_NULL 方便地定义。 包含在单个 CU_TestInfo 数组中的测试用例形成了将在单个测试套件中注册的测试集。

Suite information is then defined in one or more arrays of CU_SuiteInfo instances (defined in <CUnit/TestDB.h>):

```
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,
};
```

Each of these array elements contain the (unique) name, suite initialization function, suite cleanup function, and CU_TestInfo array for a single suite. As usual, NULL may be used for the initialization or cleanup function if the given suite does not need it. The array must end with an all-NULL element, for which the macro CU_SUITE_INFO_NULL may be used.

每一个数组元素都包含单个套件的（唯一）名称，套件初始化函数，套件清理函数和 CU_TestInfo 数组。 像往常一样，如果给定的套件不需要，则可以将 NULL 用于初始化或清理函数。数组必须以一个全为 NULL 的元素结束， 为此可以使用宏 CU_SUITE_INFO_NULL。

All suites defined in a CU_SuiteInfo array can then be registered in a single statement:

CU_SuiteInfo 数组中定义的所有套件都可以在单个语句中注册：

```
CU_ErrorCode  error = CU_register_suites(suites);
```

If an error occurs during the registration of any suite or test, an error code is returned. The error codes are the same as those returned by normal suite registration and test addition operations. The function CU_register_nsuites() is provided for situations in which the user wishes to register multiple CU_SuiteInfo arrays in a single statement:

如果在任何套件或测试的注册过程中发生错误，则返回错误代码。 错误代码与通常的套件注册和测试添加操作返回的错误代码相同。 函数 CU_register_nsuites（）用于用户希望在单个语句中注册多个 CU_SuiteInfo 数组的情况：

```
CU_ErrorCode  error = CU_register_nsuites(2, suites1, suites2);
```

This function accepts a variable number of CU_SuiteInfo arrays. The first argument indicates the actual number of arrays being passed.

此函数接受可变数量的 CU_SuiteInfo 数组。第一个参数指示要传递的数组的实际个数。

4.5. Deprecated v1 Data Types & Functions

The following data types and functions are deprecated as of version 2. To use these deprecated names, user code must be compiled with USE_DEPRECATED_CUNIT_NAMES defined.

以下变量和函数自版本 2 以来已被弃用。要使用这些不推荐使用的名称，必须使用定义的 USE_DEPRECATED_CUNIT_NAMES 编译用户代码。

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

Deprecated Name	Equivalent New Name
TestFunc	CU_TestFunc
InitializeFunc	CU_InitializeFunc
CleanupFunc	CU_CleanupFunc
_TestCase	CU_Test

PTestCase	CU_pTest
_TestGroup	CU_Suite
PTestGroup	CU_pSuite
add_test_group()	CU_add_suite()
add_test_case()	CU_add_test()
ADD_TEST_TO_GROUP()	CU_ADD_TEST()
test_case_t	CU_TestInfo
test_group_t	CU_SuiteInfo
test_suite_t	no equivalent - use CU_SuiteInfo
TEST_CASE_NULL	CU_TEST_INFO_NULL
TEST_GROUP_NULL	CU_SUITE_INFO_NULL
test_group_register	CU_register_suites()
test_suite_register	no equivalent - use CU_register_suites()

5. Running Tests

5.1. Synopsis

#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. Running Tests in CUnit

CUnit supports running all tests in all registered suites, but individual tests or suites can also be run. During each run, the framework keeps track of the number of suites, tests, and assertions run, passed, and failed. Note that the results are cleared each time a test run is initiated (even if it fails).

CUnit 支持在所有注册套件中运行所有测试，但也可以运行单独的测试或套件。 在每次运行期间，框架跟踪运行、传递和失败的套件数、测试次数和断言数。 请注意，每次启动测试运行时（即使失败），结果也会被清除。

While CUnit provides primitive functions for running suites and tests, most users will want to use one of the simplified user interfaces. These interfaces handle the details of interaction with the framework and provide output of test details and results for the user.

虽然 CUnit 提供运行套件和测试的原始功能，但大多数用户都希望使用简化的用户界面之一。 这些接口处理与框架交互的细节，并为用户提供测试详细信息和结果的输出。

The following interfaces are included in the CUnit library:

Interface	Platform	Description
Automated	all	non-interactive with output to xml files
Basic	all	non-interactive with optional output to stdout
Console	all	interactive console mode under user control
Curses	Linux/Unix	interactive curses mode under user control

If these interfaces are not sufficient, clients can also use the primitive framework API defined in <CUnit/TestRun.h>. See the source code for the various interfaces for examples of how to interact with the primitive API directly.

如果这些接口不够，客户端也可以使用<CUnit / TestRun.h>中定义的原语框架 API。 请参阅各种接口的源代码，以了解如何直接与原语 API 交互的示例。

5.3. Automated Mode

The automated interface is non-interactive. Clients initiate a test run, and the results are output to an XML file. A listing of the registered tests and suites can also be reported to an XML file.

自动化界面是非交互式的。 客户端启动测试运行，并将结果输出到 XML 文件。 注册的测试和套件的列表也可以报告给 XML 文件。

The following functions comprise the automated interface API:

以下函数包括自动化界面 API：

void **CU_automated_run_tests**(void)

Runs all tests in all registered suites. Test results are output to a file named *ROOT-Results.xml*. The filename *ROOT* can be set using CU_set_output_filename(), or else the default *CUnitAutomated-Results.xml* is used. Note that if a distinct filename *ROOT* is not set before each run, the results file will be overwritten.

在所有注册套件中运行所有测试。测试结果输出到一个名为 ROOT-Results.xml 的文件。可以使用 CU_set_output_filename（）设置文件名 ROOT，否则使用默认的 CUnitAutomated-Results.xml。 请注意，如果在每个运行之前没有设置一个不同的文件名 ROOT，结果文件将被覆盖。

The results file is supported by both a document type definition file (*CUnit-Run.dtd*) and XSL stylesheet (*CUnit-Run.xsl*). These are provided in the *Share* subdirectory of the source and installation trees.

结果文件由文档类型定义文件（CUnit-Run.dtd）和 XSL 样式表（CUnit-Run.xsl）支持。这些在源和安装树的共享子目录中提供。

CU_ErrorCode **CU_list_tests_to_file**(void)

Lists the registered suites and associated tests to file. The listing file is named *ROOT-Listing.xml*. The filename *ROOT* can be set using CU_set_output_filename(), or else the default *CUnitAutomated* is used. Note that if a distinct filename *ROOT* is not set before each run, the listing file will be overwritten.

列出注册的套件和相关的测试文件。 列表文件命名为 ROOT-Listing.xml。 可以使用 CU_set_output_filename（）设置文件名 ROOT，否则使用默认 CUnitAutomated。 请注意，如果在每次运行之前没有设置一个不同文件名 ROOT，则列表文件将被覆盖。

The listing file is supported by both a document type definition file (*CUnit-List.dtd*) and XSL stylesheet (*CUnit-List.xsl*). These are provided in the *Share* subdirectory of the source and installation trees.

列表文件由文档类型定义文件（CUnit-List.dtd）和 XSL 样式表（CUnit-List.xsl）支持。 这些在源和安装树的共享子目录中提供。

Note also that a listing file is not generated automatically by CU_automated_run_tests(). Client code must explicitly request a listing to when one is desired.

还要注意，列表文件不是由 CU_automated_run_tests（）自动生成的。 当需要时，客户端代码必须显式地请求列表。

void **CU_set_output_filename**(const char* szFilenameRoot)

Sets the output filenames for the results and listing files. *zFilenameRoot* is used to construct the filenames by appending *-Results.xml* and *-Listing.xml*, respectively.

设置结果和列表文件的输出文件名。zFilenameRoot 用于分别追加-Results.xml 和-Listing.xml 来构造文件名。

5.4. Basic Mode

The basic interface is also non-interactive, with results output to stdout. This interface supports running individual suites or tests, and allows client code to control the type of output displayed during each

run. This interface provides the most flexibility to clients desiring simplified access to the CUnit API.

基本界面也是非交互式的，结果输出到 stdout。 此接口支持运行单独的套件或测试，并允许客户端代码控制每次运行期间显示的输出类型。 该界面为希望简化访问 CUnit API 的客户提供最大的灵活性。

The following public functions are provided:

CU_ErrorCode **CU_basic_run_tests**(void)

Runs all tests in all registered suites. Returns the 1st error code occurring during the test run. The type of output is controlled by the current run mode, which can be set usingCU_basic_set_mode().

在所有注册套件中运行所有测试。 返回在测试运行期间发生的第一个错误代码。 输出类型由当前运行模式控制，该运行模式可以使用 CU_basic_set_mode（）设置。

CU_ErrorCode **CU_basic_run_suite**(CU_pSuite pSuite)

Runs all tests in single specified suite. Returns the 1st error code occurring during the test run. The type of output is controlled by the current run mode, which can be set usingCU_basic_set_mode().

在单个指定套件中运行所有测试。 返回在测试运行期间发生的第一个错误代码。 输出类型由当前运行模式控制，该运行模式可以使用 CU_basic_set_mode（）设置。

CU_ErrorCode **CU_basic_run_test**(CU_pSuite pSuite, CU_pTest pTest)

Runs a single test in a specified suite. Returns the 1st error code occurring during the test run. The type of output is controlled by the current run mode, which can be set usingCU_basic_set_mode().

在指定的套件中运行单个测试。 返回在测试运行期间发生的第一个错误代码。 输出类型由当前运行模式控制，该运行模式可以使用 CU_basic_set_mode（）设置。

void **CU_basic_set_mode**(CU_BasicRunMode mode)

Sets the basic run mode, which controls the output during test runs. Choices are:

设置基本运行模式，它控制测试运行期间的输出。 选择是：

CU_BRM_NORMAL	Failures and run summary are printed.
CU_BRM_SILENT	No output is printed except error messages.
CU_BRM_VERBOSE	Maximum output of run details.

CU_BasicRunMode **CU_basic_get_mode**(void)

Retrieves the current basic run mode code.

检索当前的基本运行模式代码。

void **CU_basic_show_failures**(CU_pFailureRecord pFailure)

Prints a summary of all failures to stdout. Does not depend on the run mode.

打印所有失败的总结到 stdout。 不依赖于运行模式。

5.5. Interactive Console Mode

The console interface is interactive. All the client needs to do is initiate the console session, and the user controls the test run interactively. This include selection & running of registered suites and tests, and viewing test results. To start a console session, use

控制台界面是交互式的。 所有客户端需要做的是启动控制台会话，用户以交互方式控制测试运行。 这包括注册套件和测试的选择和运行，以及查看测试结果。 要启动控制台会话，请使用:

void **CU_console_run_tests**(void)

5.6. Interactive Curses Mode

The curses interface is interactive. All the client needs to do is initiate the curses session, and the user controls the test run interactively. This include selection & running of registered suites and tests, and viewing test results. Use of this interface requires linking the ncurses library into the application. To start a curses session, use

curses 界面是互动的。 所有客户端需要做的是启动 curses 会话，用户以交互方式控制测试运行。 这包括注册套件和测试的选择和运行，以及查看测试结果。 使用此接口需要将 curses 库链接到应用程序中。 要启动诅咒会话，请使用

void **CU_curses_run_tests**(void)

注：curses 是一个在 Linux/Unix 下广泛应用的图形函数库.，作用是可以绘制在 DOS 下的用户界面和漂亮的图形。

5.7. Getting Test Results

The interfaces present results of test runs, but client code may sometimes need to access the results directly. These results include various run counts, as well as a linked list of failure records holding the failure details. Note that test results are overwritten each time a new test run is started, or when the registry is initialized or cleaned up.

接口显示测试运行的结果，但客户端代码有时可能需要直接访问结果。 这些结果包括各种运行计数，以及保存故障细节的故障记录链表。 请注意，每次启动新的测试运行时，或者初始化或清除注册表时，测试结果都将被覆盖。

Functions for accessing the test results are:

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)

These functions report the number of suites, tests, and assertions that ran or failed during the last run. A suite is considered failed if it's initialization or cleanup function returned non-NULL. A test fails if any of its assertions fail. The last 3 functions all refer to individual assertions.

这些功能会报告上次运行中运行或失败的套件数，测试次数和断言数。 如果初始化函数或清理函数返回非空，则认为该套件失败。 如果其任何断言失败，则测试失败。 最后 3 个功函数涉及各自的断言。

To retrieve the total number of registered suites and tests, use CU_get_registry()->uiNumberOfSuites and CU_get_registry()->uiNumberOfTests, respectively.

要检索注册套件和测试的总数，请分别使用 CU_get_registry（） - > uiNumberOfSuites 和 CU_get_registry（） - > uiNumberOfTests。

const CU_pRunSummary **CU_get_run_summary**(void)

Retrieves all test result counts at once. The return value is a pointer to a saved structure containing the counts. This data type is defined in <CUnit/TestRun.h> (included automatically by <CUnit/CUnit.h>):

立即检索所有测试结果计数。 返回值是指向包含计数的保存结构的指针。 此数据类型在<CUnit / TestRun.h>中定义（由<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;
```

The structure variable associated with the returned pointer is owned by the framework, so the user should not free or otherwise change it. *Note that the pointer may be invalidated once another test run is initiated.*

与返回的指针相关联的结构变量由框架拥有，因此用户不应该释放或以其他方式更改它。 请注意，一旦启动另一个测试运行，指针可能会失效。

const CU_pFailureRecord **CU_get_failure_list**(void)

Retrieves a linked list recording any failures occurring during the last test run (NULL for no failures). The data type of the return value is declared in <CUnit/TestRun.h> (included automatically by <CUnit/CUnit.h>). Each failure record contains information about the location and nature of the failure:

检索链接列表，记录上一次测试运行期间发生的任何故障（NULL，无故障）。 返回值的数据类型在<CUnit / TestRun.h>中声明（由<CUnit / CUnit.h>自动包含）。 每个故障记录都包含有关故障位置和性质的信息：

```
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;
```

The structure variable associated with the returned pointer is owned by the framework, so the user should not free or otherwise change it. *Note that the pointer may be invalidated once another test run is initiated.*

与返回的指针相关联的结构变量由框架拥有，因此用户不应该释放或以其他方式更改它。 请注意，一旦启动另一个测试运行，指针可能会失效。

unsigned int **CU_get_number_of_failure_records**(void)

Retrieves the number of CU_FailureRecords in the linked list of failures returned by CU_get_failure_list(). Note that this can be more than the number of failed assertions, since suite initialization and cleanup failures are included.

检索由 CU_get_failure_list（）返回的链接列表中的 CU_FailureRecords 的数量。 请注意，这可能超过失败的断言数，因为包含初始化和清除故障。

5.8. Deprecated v1 Data Types & Functions

The following data types and functions are deprecated as of version 2. To use these deprecated names, user code must be compiled with USE_DEPRECATED_CUNIT_NAMESdefined.

以下变量和函数自版本 2 以来已被弃用。要使用这些不推荐使用的名称，必须使用定义的 USE_DEPRECATED_CUNIT_NAMES 编译用户代码。

Deprecated Name	Equivalent New Name
automated_run_tests()	CU_automated_run_tests() plus CU_list_tests_to_file()
set_output_filename()	CU_set_output_filename()
console_run_tests()	CU_console_run_tests()
curses_run_tests()	CU_curses_run_tests()

6. Error Handling

6.1. Synopsis

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

```
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 Error Handling

Most CUnit functions set an error code indicating the framework error

status. Some functions return the code, while others just set the code and return some other value. Two functions are provided for examining the framework error status:

大多数 CUnit 函数设置一个指示框架错误状态的错误代码。 一些函数返回代码，而其他函数只是设置代码并返回一些其他值。 提供了两个函数来检查框架错误状态：

CU_ErrorCode **CU_get_error**(void)
const char* **CU_get_error_msg**(void)

The first returns the error code itself, while the second returns a message describing the error status. The error code is an enum of type CU_ErrorCode defined in <CUnit/CUError.h>. The following error code values are defined:

第一个返回错误代码本身，而第二个返回描述错误状态的消息。 错误代码是在<CUnit / CUError.h>中定义的类型为 CU_ErrorCode 的枚举。 定义了以下错误代码值：

Error Value	Description
CUE_SUCCESS	No error condition.
CUE_NOMEMORY	Memory allocation failed.
CUE_NOREGISTRY	Test registry not initialized.
CUE_REGISTRY_EXISTS	Attempt to CU_set_registry() without CU_cleanup_registry().
CUE_NOSUITE	A required CU_pSuite pointer was NULL.
CUE_NO_SUITENAME	Required CU_Suite name not provided.
CUE_SINIT_FAILED	Suite initialization failed.
CUE_SCLEAN_FAILED	Suite cleanup failed.
CUE_DUP_SUITE	Duplicate suite name not allowed.
CUE_NOTEST	A required CU_pTest pointer was NULL.
CUE_NO_TESTNAME	Required CU_Test name not provided.
CUE_DUP_TEST	Duplicate test case name not allowed.
CUE_TEST_NOT_IN_SUITE	Test is not registered in the specified suite.
CUE_FOPEN_FAILED	An error occurred opening a file.
CUE_FCLOSE_FAILED	An error occurred closing a file.
CUE_BAD_FILENAME	A bad filename was requested (NULL, empty, nonexistent, etc.).
CUE_WRITE_ERROR	An error occurred during a write to a file.

6.3. Behavior Upon Framework Errors

The default behavior when an error condition is encountered is for the error code to be set and execution continued. There may be times when clients prefer for a test run to stop on a framework error, or even for the test application to exit. This behavior can be set by the user, for which the following functions are provided:

遇到错误条件时的默认行为是错误代码被设置并继续执行。 客户可能会喜欢测试运行停止框架错误，甚至测试应用程序退出。 此行为可由用户设置，为此提供以下功能：

void **CU_set_error_action**(CU_ErrorAction action)
CU_ErrorAction **CU_get_error_action**(void)

The error action code is an enum of type CU_ErrorAction defined in <CUnit/CUError.h>. The following error action codes are defined:

错误操作代码是在<CUnit / CUError.h>中定义的类型为 CU_ErrorAction 的枚举。 定义了以下错误操作代码：

Error Value	Description
CUEA_IGNORE	Runs should be continued when an error condition occurs (default)
CUEA_FAIL	Runs should be stopped when an error condition occurs
CUEA_ABORT	The application should exit() when an error conditions occurs

6.4. Deprecated v1 Variables & Functions

The following variables and functions are deprecated as of version 2. To use these deprecated names, user code must be compiled with USE_DEPRECATED_CUNIT_NAMES defined.

以下变量和函数自版本 2 以来已被弃用。要使用这些不推荐使用的名称，必须使用定义的 USE_DEPRECATED_CUNIT_NAMES 编译用户代码。

Deprecated Name	Equivalent New Name
get_error()	CU_get_error_msg()
error_code	None. Use CU_get_error()