

介紹

基本介紹

GoogleTest (全名 Google Testing Framework) 是由 Google 開發的測試框架，同時也是在 GitHub 上星星數超過 30k 的開源專案。早期是 Google 內部使用，直到 2008 年開源給所有人。知名專案如：Chromium 與 OpenCV 皆使用 GoogleTest 來做測試。

GoogleTest 有可攜性，不論在 Windows, Mac 與 Linux 系統上都能測試 C++ 程式，而不是只限於特定平台。此外，功能也不僅僅包含單元測試。

設計理念

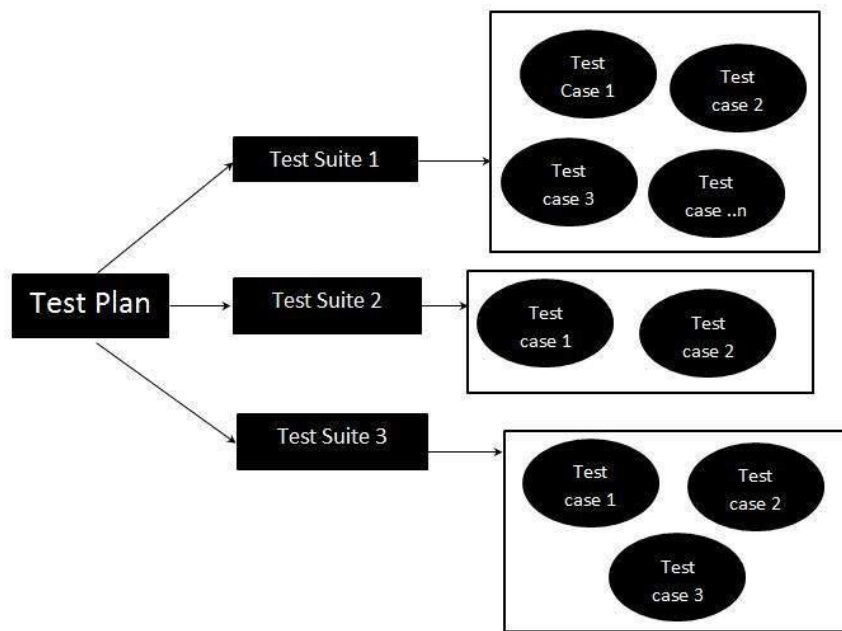
GoogleTest 的設計理念包含以下幾點：

1. 測試應該要是彼此獨立、可重複的
2. 測試是要有良好的組織性，而且能反映出被測試程式的架構
3. 測試應該要是具有可攜性且可以重複使用
4. 當測試失敗時，要能提供該問題足夠的資訊
5. 測試框架應當讓測試人員專心於測試本身
6. 測試要能快速實現

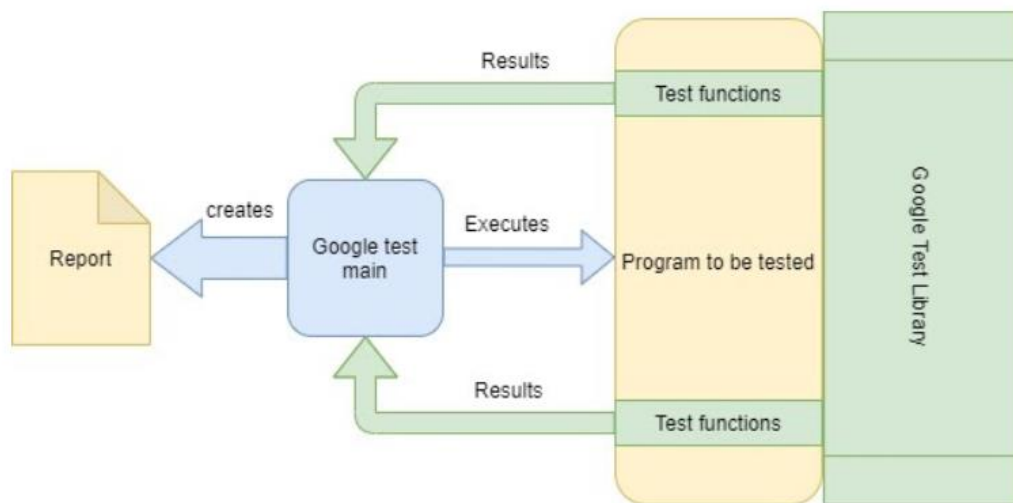
背景知識

在 GoogleTest 中，每個 test 底下會撰寫 assertion 檢查條件或敘述 (statement) 是否符合預期；若不符合，則代表測試失敗。

Test 會透過 assertion 測試程式的行為。一個 test suite 由一個或多個 test 組成，這種形式能反映出測試程式本身的架構。整個測試程式，也就是 test program，會包含一個以上的 test suite。可以參考下面的示意圖：



架構與流程



GoogleTest 的架構如上圖所示，會由 GoogleTest main function 執行被測試的程式，透過撰寫的 test function 把測試的結果回傳，最後輸出整體結果的報告。

實作

環境建置、安裝與第一支測試程式

環境：

Windows + WSL Ubuntu 22.04.2 LTS

安裝：

安裝 Cmake

```
pip install cmake
```

第一支測試程式：

GoogleTest 中每一個測試程式除了要測試的程式檔案外，還包含 CMakeLists.txt 這個描述 dependency 的檔案。

假設要測試 main.cpp 中的函式 hello()，內容如下：

```
#include<iostream>
using namespace std;

string hello()
{
    return "hello world";
}
```

為此我們需要撰寫一個測試程式 test.cpp，內容如下：

```
#include <gtest/gtest.h>
#include "main.cpp"

TEST(HelloTest, hello){
    EXPECT_EQ("hello world", hello());
}
```

第一行與第二行分別引入 GoogleTest 跟欲測試的程式。接著透過 TEST()來檢驗函式 hello()的結果是否如預期。關於 TEST()的語法會在後面做介紹，此處先行省略。

接下來是把 GoogleTest 加入 Cmake 中，內容如下：

```

cmake_minimum_required(VERSION 3.14)
project(hello)

# GoogleTest requires at least C++14
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

include(FetchContent)
FetchContent_Declare(
    googletest
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG release-1.11.0
)

FetchContent_MakeAvailable(googletest)

enable_testing()

add_executable(
    hello_world
    test.cpp
)
target_link_libraries(
    hello_world
    GTest::gtest_main
)

include(GoogleTest)
gtest_discover_tests(hello_world)

```

完成 Cmake 跟測試程式撰寫後，可以來進行測試。

指令如下：

```
mkdir build && cd build
```

```
cmake ..; make
```

```
./hello_world
```

Terminal 上的執行解果顯示測試通過：

```

(base) cc@LAPTOP-9F6MDV08:~/software_testing/midterm/hello_world/build$ ./hello_world
Running main() from /home/cc/software_testing/midterm/hello_world/build/_deps/googletest-src/googletest/src/gtest_main.c
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from HelloTest
[ RUN     ] HelloTest.hello
[ OK      ] HelloTest.hello (0 ms)
[-----] 1 test from HelloTest (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (0 ms total)
[ PASSED ] 1 test.

```

也可以透過指令 `ctest` 來執行測試。

之後若有修改被測程式或測試程式，只要在 `build` 資料夾底下再次 `make` 跟跑 `./hello_world` 即可。

GoogleTest 語法

Assertions

Assertion 的語法如下：

```
EXPECT_TRUE(condition) << "The arrestion is not correct!!";
```

代表若 `condition` 不成立，則會印出 "The arrestion is not correct!!" 的字樣。後面輸出字串的部分可以選擇省略。

常用的 `assertion` 有以下幾種：

Boolean

`EXPECT_TRUE(condition)`

`EXPECT_FALSE(condition)`

二元判斷

`EXPECT_EQ(val1, val2) // val1 == val2`

`EXPECT_NE(val1, val2) // val1 != val2`

`EXPECT_LE(val1, val2) // val1 <= val2`

...

值得注意的是，`EXPECT_EQ` 看的是記憶體的位置。也就是說若 `val1` 與 `val2` 為字串，那麼測試的依據是檢驗它們的記憶體位置是否相同，而不是它們有沒有相同的數值。因此，判斷兩字串的值要用 `EXPECT_STREQ`；同理，檢驗指標是不是空值，要用 `EXPECT_NE(ptr, nullptr)`。

EXPECT_THROW

語法為 `EXPECT_THROW(statement, exception_type)`。

假設測試函式

```
void sayHi(string name)
{
    if(!('A' <= name[0] && name[0] <= 'Z'))
        throw std::runtime_error("Your name should begin with capital letter");
    cout << "Hi, " << name << endl;
}
```

寫 TEST 時要給 exception_type 相對應的型態：

```
TEST(HiTest, sayhi)
{
    EXPECT_THROW(sayHi("rick"), std::runtime_error);
}
```

得到執行結果：

```
Running main() from /home/cc/software_testing/midterm/hello_world/build/_deps/googletest-src/googletest/src/gtest_main.c
c
[=====] Running 1 test from 1 test suite.
[=====] Global test environment set-up.
[=====] 1 test from HiTest
[ RUN      ] HiTest.sayhi
[ OK       ] HiTest.sayhi (7 ms)
[=====] 1 test from HiTest (7 ms total)

[=====] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (7 ms total)
[ PASSED  ] 1 test.
```

TEST

GoogleTest 撰寫 testcase 的語法如下：

```
TEST(TestSuiteName, TestName) {
    ... statements ...
}
```

第一個參數 TestSuiteName 代表屬於哪個 test suite，TestName 如同字面上的意思表示該 test 的名稱。

TEST_F

除了 TEST 外，GoogleTest 有個更進階、名為 TEST_F (test fixture)的功能。

假設想要撰寫一支測試程式，會多次在某些物件、函式做類似的行為，此時就可以使用 TEST_F 來減少重複的程式碼，做初始化(initialize)或者清除(cleanup)，需要在測試程式實作相對應的 SetUp()與 TearDown()函式。

使用 TEST_F 時需要先定義 fixture class，並依需求實現 SetUp()跟 TearDown() 函式。以測試 Stack 物件的程式為例：

```

class StackTest : public testing::Test {
protected:
    void SetUp() override {
        st1.push(0);
        st1.push(1);
        st1.push(2);
    }

    // void TearDown() override {
    // }

    Stack st1, st2;
};

```

接下來如同 TEST，撰寫測試：

```

TEST_F(StackTest, isEmptyAndFull) {
    EXPECT_FALSE(st1.is_full());
    EXPECT_FALSE(st1.is_empty());
    EXPECT_TRUE(st2.is_empty());
    EXPECT_FALSE(st2.is_full());
}

```

這裡的 st1 一開始就已經 push 過三個 element 進去，st2 則沒有。在 isEmptyAndFull 這個 test 中會分別檢驗 is_full() 與 is_empty() 運作是否如預期，執行結果如下：

```

(base) cc@LAPTOP-9F6MDV0B:~/software_testing/midterm/hello_world/build$ ./hello_world
Running main() from /home/cc/software_testing/midterm/hello_world/build/_deps/googletest-src/googletest/src/gtest_main.c
c
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from StackTest
[ RUN     ] StackTest.isEmptyAndFull
[ OK      ] StackTest.isEmptyAndFull (0 ms)
[-----] 1 test from StackTest (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (0 ms total)
[ PASSED ] 1 test.

```

總結來說，測試程式運行的流程為：

1. 依據 StackTest 的 SetUp() 進行初始化
2. 執行測試 isEmptyAndFull
3. 執行 StackTest 的 TearDown() (本例子沒有使用)
4. 繼續跑下一個 TEST_F

TEST_P

TEST_P 是一個參數化的測試(Value-Parameterized Test)，提供高效率的參數測

試方式。舉例來說，若要測試 `isOdd(n)` 函式，用 `TEST` 來寫的話可能長這樣：

```
TEST(isOddTest, handleTrueReturn)
{
    EXPECT_TRUE(isOdd(1));
    EXPECT_TRUE(isOdd(3));
    EXPECT_TRUE(isOdd(5));
    EXPECT_TRUE(isOdd(7));
    EXPECT_TRUE(isOdd(9));
}
```

但如果要測試成千上萬的參數，難道還要一行一行的寫嗎？`TEST_P` 就是為了應對這種情況而設計的。

與 `TEST_F` 類似，要先定義一個 `class`：

```
class isOddParamTest : public::testing::TestWithParam<int>
{
};
```

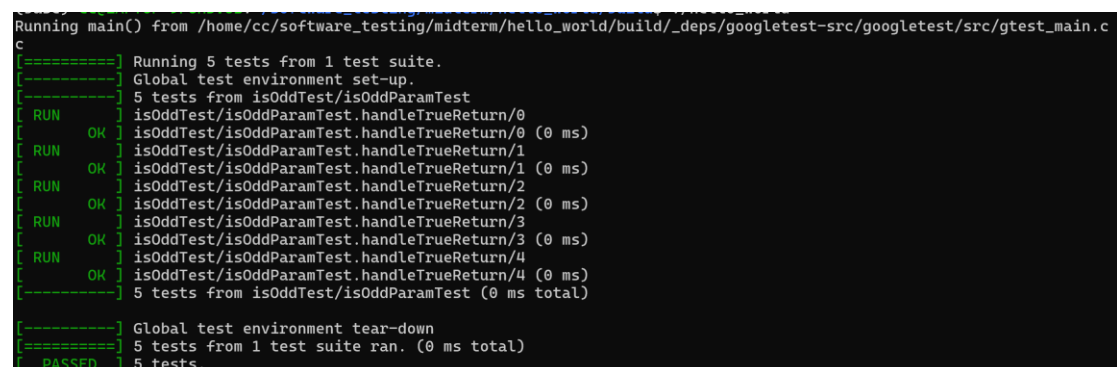
接著寫 `TEST_P`：

```
TEST_P(isOddParamTest, handleTrueReturn)
{
    int n = GetParam();
    EXPECT_TRUE(isOdd(n));
}
```

最後，呼叫 `INSTITUTE_TEST_CASE_P` 並填入要測式的參數即可：

```
INSTITUTE_TEST_CASE_P(isOddTest, isOddParamTest, testing::Values(3, 5, 11, 23, 17));
```

測試程式的執行結果如圖：



```
Running main() from /home/cc/software_testing/midterm/hello_world/build/_deps/googletest-src/googletest/src/gtest_main.c
[=====] Running 5 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 5 tests from isOddTest/isOddParamTest
[ RUN      ] isOddTest/isOddParamTest.handleTrueReturn/0
[ OK       ] isOddTest/isOddParamTest.handleTrueReturn/0 (0 ms)
[ RUN      ] isOddTest/isOddParamTest.handleTrueReturn/1
[ OK       ] isOddTest/isOddParamTest.handleTrueReturn/1 (0 ms)
[ RUN      ] isOddTest/isOddParamTest.handleTrueReturn/2
[ OK       ] isOddTest/isOddParamTest.handleTrueReturn/2 (0 ms)
[ RUN      ] isOddTest/isOddParamTest.handleTrueReturn/3
[ OK       ] isOddTest/isOddParamTest.handleTrueReturn/3 (0 ms)
[ RUN      ] isOddTest/isOddParamTest.handleTrueReturn/4
[ OK       ] isOddTest/isOddParamTest.handleTrueReturn/4 (0 ms)
[-----] 5 tests from isOddTest/isOddParamTest (0 ms total)

[-----] Global test environment tear-down
[=====] 5 tests from 1 test suite ran. (0 ms total)
[ PASSED ] 5 tests.
```


心得

這門課是我第一次接觸到軟體測試相關的工具，因此若要進行比較，僅能與 Nodejs 的測試框架來做分析。

GoogleTest 跟 Nodejs 的 test runner 有個很不一樣的地方在於分成三種類型的 test：TEST, TEST_F 跟 TEST_P，各有各自適合的使用場景。其中 TEST_F 跟 TEST_P 我覺得很實用，以 TEST_F 來說，如果要測試「運行到某個階段」的程式的各種操作，總不會要每個 test 都各自寫一次同樣的初始化流程吧？但在 nodejs 至少以當前幾個 lab 都沒碰到相關的操作。

不過相較 nodejs，GoogleTest 因為是透過 Cmake 編譯，所以還需要額外學 Cmake 的語法，比起來 nodejs 測試的入門會比較容易。

另外，GoogleTest 本身有提供教學跟相對應的 code，還把每個 sample code 的內容簡述。網站設計簡潔又直觀，對比 nodejs 的官方文件，我認為前者對於初學者更友善、對於學習的幫助很大。

若未來有開發 C++ 程式，我會考慮用 GoogleTest 進行測試，比起直接 cout 出來或者設中斷點，GoogleTest 提供了另一種測試程式的選擇。舉例來說，大一時想測試程式設計課的 Final Project，只知道手動輸入各種數值以及在某行加上 cout 來判斷運作是否符合預期，但這樣一來不美觀、程式與測試夾雜，二來沒辦法有效做測試。如果大學時有機會接觸到測試框架，應該能節省不少時間。

參考資料

[GoogleTest GitHub](#)

[GoogleTest User's Guide](#)

[如何使用 GoogleTest 寫 C++ 單元測試](#)

[GTest Framework](#)

[Test the Architecture with Google Test](#)

[Announcing: New Google C++ Testing Framework](#)