Google C++ Testing Framework 简介

了解易用性和生产级部署方面的关键特性

Google 提供一种用于为 C/C++ 软件开发单元测试的开放源码框架，它很有意思，也很容易使用。本文介绍一些比较有用的 Google C++ Testing Framework 特性。本文基于 version 1.4。

Arpan Sen 是致力于电子设计自动化行业的软件开发首席工程师。他使用各种 UNIX 版本（包括 Solaris、SunOS、HP-UX、IRIX，以及 Linux 和 Microsoft Windows）已经多年。他热衷于各种软件性能优化技术、图论和并行计算。Arpan 获得了软件系统硕士学位。

2010 年 6 月 24 日

为什么要使用 Google C++ Testing Framework?

使用这个框架有许多好理由。本节讨论其中几个。

某些类型的测试有糟糕的内存问题，这些问题只在某几次运行期间出现。Google 的测试框架为处理这种情况提供了出色的支持。可以使用 Google 框架重复运行相同的测试一千次。当出现故障的迹象时，自动地调用调试器。另外，这只需要在命令行上传递两个开关即可实现：`--gtest_repeat=1000 --gtest_break_on_failure`。

与其他许多测试框架相反，可以把 Google 测试框架内置的断言部署在禁用了异常处理（通常由于性能原因）的软件中。因此，也可以在析构函数中安全地使用断言。

运行测试很简单。只需调用预定义的 `RUN_ALL_TESTS` 宏，而不需要通过创建或驱动单独的运行器类来执行测试。这比 CppUnit 等框架方便多了。

只需传递一个开关即可生成 Extensible Markup Language (XML) 报告：`--gtest_output="xml:<file name>"`。在 CppUnit 和 CppTest 等框架中，需要编写很多代码才能生成 XML 输出。

创建基本测试

请考虑 清单 1 所示的平方根函数的原型。

清单 1. 平方根函数的原型

```
double square-root (const double);
```

对于负数，这个例程返回 -1。正数测试和负数测试对它都有意义，所以我们编写两个测试。清单 2 给出测试用例。

清单 2. 平方根函数的单元测试

```
#include "gtest/gtest.h"

TEST (SquareRootTest, PositiveNos) {
    EXPECT_EQ (18.0, square-root (324.0));
    EXPECT_EQ (25.4, square-root (645.16));
    EXPECT_EQ (50.3321, square-root (2533.310224));
}

TEST (SquareRootTest, ZeroAndNegativeNos) {
    ASSERT_EQ (0.0, square-root (0.0));
    ASSERT_EQ (-1, square-root (-22.0));
}
```

清单 2 创建一个名为 `SquareRootTest` 的测试层次结构，然后在其中添加两个单元测试 `PositiveNos` 和 `ZeroAndNegativeNos`。`TEST` 是在 gtest.h（可以在下载的源代码中找到）中定义的预定义宏，它帮助定义这个层次结构。`EXPECT_EQ` 和 `ASSERT_EQ` 也是宏 — 对于前者，即使出现测试失败，测试仍然继续执行；对于后者，测试终止执行。显然，如果 0 的平方根是除 0 之外的任何数字，就没必要再执行测试了。因此，`ZeroAndNegativeNos` 测试只使用 `ASSERT_EQ`，而 `PositiveNos` 测试使用 `EXPECT_EQ`，来报告

存在多少个测试用例是平方根函数失败而不终止测试的情况。

运行第一个测试

既然已经创建了第一个基本测试，现在就该运行它了。清单 3 是运行测试的主例程。

清单 3. 运行平方根测试

```
#include "gtest/gtest.h"

TEST(SquareRootTest, PositiveNos) {
    EXPECT_EQ (18.0, square-root (324.0));
    EXPECT_EQ (25.4, square-root (645.16));
    EXPECT_EQ (50.3321, square-root (2533.310224));
}

TEST (SquareRootTest, ZeroAndNegativeNos) {
    ASSERT_EQ (0.0, square-root (0.0));
    ASSERT_EQ (-1, square-root (-22.0));
}

int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

顾名思义，`::testing::InitGoogleTest` 方法的作用就是对框架进行初始化，必须在调用 `RUN_ALL_TESTS` 之前调用它。在代码中只能调用 `RUN_ALL_TESTS` 一次，因为多次调用会与框架的一些高级特性冲突，不支持这种做法。注意，`RUN_ALL_TESTS` 自动地探测并运行用 `TEST` 宏定义的所有测试。在默认情况下，结果输出到标准输出。清单 4 给出输出。

清单 4. 运行平方根测试的输出

```
Running main() from user_main.cpp
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from SquareRootTest
[ RUN      ] SquareRootTest.PositiveNos
..\user_sqrt.cpp(6862): error: Value of: sqrt (2533.310224)
  Actual: 50.332
Expected: 50.3321
[  FAILED  ] SquareRootTest.PositiveNos (9 ms)
[ RUN      ] SquareRootTest.ZeroAndNegativeNos
[       OK ] SquareRootTest.ZeroAndNegativeNos (0 ms)
[----------] 2 tests from SquareRootTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (10 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] SquareRootTest.PositiveNos

 1 FAILED TEST
```

Google C++ Testing Framework 的选项

在 清单 3 中，`InitGoogleTest` 函数接受测试基础设施的参数。本节讨论可以通过测试框架的参数实现的一些功能。

通过在命令行上传递 `--gtest_output="xml:report.xml"`，可以把输出转储为 XML 格式。当然，可以把 `report.xml` 替换为您喜欢的任何文件名。

某些测试有时候会失败，但是在大多数时候会顺利通过。这是与内存损坏相关的问题的典型特点。如果多次运行测试，就能够提高发现失败的可能性。如果在命令行上传递 `--gtest_repeat=2 --gtest_break_on_failure`，就重复运行相同的测试两次。如果测试失败，会自动调用调试器。

并不需要每次都运行所有测试，尤其是在修改的代码只影响某几个模块的情况下。为了支持运行一部分测试，Google 提供 `--gtest_filter=<test string>`。test string 的格式是由冒号 (:) 分隔的一系列通配符模式。例如，`--gtest_filter=*` 运行所有测试，而 `--gtest_filter=SquareRoot*` 只运行 `SquareRootTest` 测试。如果希望只运行 `SquareRootTest` 中的正数单元测试，应该使用 `--gtest_filter=SquareRootTest.*-SquareRootTest.Zero*`。注意，`SquareRootTest.*` 表示属于 `SquareRootTest` 的所有测试，而 `-SquareRootTest.Zero*` 表示不运行名称以 Zero 开头的测试。

清单 5 中的示例用 `gtest_output`、`gtest_repeat` 和 `gtest_filter` 选项运行 `SquareRootTest`。

清单 5. 用 `gtest_output`、`gtest_repeat` 和 `gtest_filter` 选项运行 `SquareRootTest`

```
[arpan@tintin] ./test_executable --gtest_output="xml:report.xml" --gtest_repeat=2 --
gtest_filter=SquareRootTest.*-SquareRootTest.Zero*
```

```
Repeating all tests (iteration 1) . . .

Note: Google Test filter = SquareRootTest.*-SquareRootTest.Z*
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from SquareRootTest
[ RUN      ] SquareRootTest.PositiveNos
..\user_sqrt.cpp (6854): error: Value of: sqrt (2533.310224)
  Actual: 50.332
Expected: 50.3321
[  FAILED  ] SquareRootTest.PositiveNos (2 ms)
[----------] 1 test from SquareRootTest (2 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (20 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] SquareRootTest.PositiveNos
 1 FAILED TEST

Repeating all tests (iteration 2) . . .

Note: Google Test filter = SquareRootTest.*-SquareRootTest.Z*
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from SquareRootTest
[ RUN      ] SquareRootTest.PositiveNos
..\user_sqrt.cpp (6854): error: Value of: sqrt (2533.310224)
  Actual: 50.332
Expected: 50.3321
[  FAILED  ] SquareRootTest.PositiveNos (2 ms)
[----------] 1 test from SquareRootTest (2 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (20 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] SquareRootTest.PositiveNos
 1 FAILED TEST
```

临时禁用测试

可以临时禁用测试吗？可以，只需在逻辑测试名或单元测试名前面加上 `DISABLE_` 前缀，它就不会执行了。清单 6 演示如何禁用 清单 2 中的 `PositiveNos` 测试。

清单 6. 临时禁用测试

```
#include "gtest/gtest.h"

TEST (DISABLE_SquareRootTest, PositiveNos) {
    EXPECT_EQ (18.0, square-root (324.0));
    EXPECT_EQ (25.4, square-root (645.16));
    EXPECT_EQ (50.3321, square-root (2533.310224));
}

OR

TEST (SquareRootTest, DISABLE_PositiveNos) {
    EXPECT_EQ (18.0, square-root (324.0));
    EXPECT_EQ (25.4, square-root (645.16));
    EXPECT_EQ (50.3321, square-root (2533.310224));
}
```

注意，如果禁用了任何测试，Google 框架会在测试执行结束时输出警告消息，见 清单 7。

清单 7. Google 警告用户在框架中有禁用的测试

```
1 FAILED TEST
  YOU HAVE 1 DISABLED TEST
```

如果希望继续运行禁用的测试，那么在命令行上传递 `-gtest_also_run_disabled_tests` 选项。清单 8 给出运行 `DISABLE_PositiveNos` 测试时的输出。

清单 8. Google 允许运行已经禁用的测试

```
[----------] 1 test from DISABLED_SquareRootTest
[ RUN      ] DISABLED_SquareRootTest.PositiveNos
..\user_sqrt.cpp(6854): error: Value of: square-root (2533.310224)
  Actual: 50.332
Expected: 50.3321
[  FAILED  ] DISABLED_SquareRootTest.PositiveNos (2 ms)
[----------] 1 test from DISABLED_SquareRootTest (2 ms total)

[  FAILED  ] 1 tests, listed below:
[  FAILED  ] SquareRootTest. PositiveNos
```

断言

Google 测试框架提供一整套预定义的断言。断言分为两类 — 名称以 `ASSERT_` 开头的断言和名称以 `EXPECT_` 开头的断言。如果一个断言失败，`ASSERT_*` 断言会终止程序执行，而 `EXPECT_*` 断言继续运行。当断言失败时，这两者都输出文件名、行号和可定制的消息。比较简单的断言包括 `ASSERT_TRUE (condition)` 和 `ASSERT_NE (val1, val2)`。前者期望条件总是成立，而后者期望两个值不匹配。这些

断言也适用于用户定义的类型，但是必须覆盖相应的比较操作符（==、!=、<= 等等）。

浮点数比较

Google 提供 清单 9 所示的宏以支持浮点数比较。

清单 9. 用于浮点数比较的宏

```
ASSERT_FLOAT_EQ (expected, actual)
ASSERT_DOUBLE_EQ (expected, actual)
ASSERT_NEAR (expected, actual, absolute_range)

EXPECT_FLOAT_EQ (expected, actual)
EXPECT_DOUBLE_EQ (expected, actual)
EXPECT_NEAR (expected, actual, absolute_range)
```

为什么需要用单独的宏进行浮点数比较？使用 ASSERT_EQ 不行吗？使用 ASSERT_EQ 和相关的宏可能可以，也可能不行，但是使用专门用于浮点数比较的宏更好。通常，不同的中央处理单元 (CPU) 和操作环境以不同的方式存储浮点数，简单地比较期望值和实际值是无效的。例如，ASSERT_FLOAT_EQ (2.00001, 2.000011) 会顺利通过 — 如果直到小数点后四位都匹配，Google 就不会抛出错误。如果需要更精确的比较，应该使用 ASSERT_NEAR (2.00001, 2.000011, 0.0000001)，就会得到 清单 10 所示的错误。

清单 10. ASSERT_NEAR 产生的错误消息

```
Math.cc(68): error: The difference between 2.00001 and 2.000011 is 1e-006, which exceeds
0.0000001, where
2.00001 evaluates to 2.00001,
2.000011 evaluates to 2.00001, and
0.0000001 evaluates to 1e-007.
```

死亡测试

Google C++ Testing Framework 有一类有趣的断言（ASSERT_DEATH、ASSERT_EXIT 等），它们称为死亡断言。使用这种断言检查在例程的输入错误的情况下是否发出正确的错误消息，或者例程是否退出并返回正确的退出码。例如，在 清单 3 中，当执行 square-root (-22.0) 时应该产生错误消息，程序退出，返回状态应该是 -1 而不是 -1.0。清单 11 使用 ASSERT_EXIT 检查是否是这样。

清单 11. 使用 Google 框架运行死亡测试

```
#include "gtest/gtest.h"

double square-root (double num) {
    if (num < 0.0) {
        std::cerr << "Error: Negative Input\n";
        exit(-1);
    }
    // Code for 0 and +ve numbers follow

}

TEST (SquareRootTest, ZeroAndNegativeNos) {
    ASSERT_EQ (0.0, square-root (0.0));
    ASSERT_EXIT (square-root (-22.0), ::testing::ExitedWithCode(-1), "Error:
Negative Input");
}

int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

ASSERT_EXIT 检查函数是否退出并返回正确的退出码（即 exit 或 _exit 例程的参数），对引号中的字符串和函数发送到标准错误的字符串进行比较。注意，错误消息必须发送到 std::cerr 而不是 std::cout。清单 12 给出 ASSERT_DEATH 和 ASSERT_EXIT 的原型。

清单 12. 死亡断言的原型

```
ASSERT_DEATH(statement, expected_message)
ASSERT_EXIT(statement, predicate, expected_message)
```

Google 提供预定义的谓词 ::testing::ExitedWithCode(exit_code)。只有在程序退出且退出码与谓词中的 exit_code 相同的情况下，这个谓词的结果才是 true。ASSERT_DEATH 比 ASSERT_EXIT 简单；它只检查标准错误中的错误消息是否是用户期望的消息。

理解测试装备

在执行单元测试之前，通常要执行一些定制的初始化。例如，如果希望度量测试的时间/内存占用量，就需

要放置一些测试专用代码以度量这些值。这就是装备的用途 — 它们帮助完成这种定制的测试初始化。清单 13 给出一个装备类的示例。

清单 13. 测试装备类

```
class myTestFixture1: public ::testing::test {
public:
   myTestFixture1( ) {
       // initialization code here
   }

   void SetUp( ) {
       // code here will execute just before the test ensues
   }

   void TearDown( ) {
       // code here will be called just after the test completes
       // ok to through exceptions from here if need be
   }

   ~myTestFixture1( )  {
       // cleanup any pending stuff, but no exceptions allowed
   }

   // put in any custom data members that you need
};
```

这个装备类派生自 `gtest.h` 中声明的 `::testing::test` 类。清单 14 是使用这个装备类的示例。注意，它使用 `TEST_F` 宏而不是 `TEST`。

清单 14. 装备的使用示例

```
TEST_F (myTestFixture1, UnitTest1) {

.
}

TEST_F (myTestFixture1, UnitTest2) {

.
}
```

在使用装备时，要注意以下几点：

可以在构造函数或 `SetUp` 方法中执行初始化或分配资源。由用户选择具体方式。

可以在 `TearDown` 或析构函数例程中释放资源。但是，如果需要异常处理，那么只能在 `TearDown` 代码中进行，因为从析构函数中抛出异常会导致不确定的结果。

在以后的版本中，Google 断言宏可能会在平台上抛出异常。因此，为了便于维护，最好在 `TearDown` 代码中使用断言宏。

并不跨多个测试使用同一个测试装备。对于每个新的单元测试，框架创建一个新的测试装备。在 清单 14 中，由于要创建两个 `myFixture1` 对象，所以两次调用 `SetUp` 例程（请注意使用正确的拼写）。

结束语

本文仅仅触及了 Google C++ Testing Framework 的皮毛。可以从 Google 网站获得关于这个框架的详细信息。对于高级开发人员，我建议阅读 Boost 单元测试框架和 CppUnit 等开放回归框架方面的其他文章。更多信息参见 参考资料。

参考资料

学习

阅读 Google TestPrimer，初步了解 Google C++ Testing Framework。

关于高级 Google C++ Testing Framework 主题的讨论，参见 Google TestAdvancedGuide。

在 Google TestFAQ 查找关于 Google C++ Testing Framework 的提示和常见问题。

阅读 “开放源码 C/C++ 单元测试工具，第 1 部分: 了解 Boost 单元测试框架 ”。

阅读 “开放源码 C/C++ 单元测试工具，第 2 部分: 了解 CppUnit”。

关于浮点数比较的更多信息，请阅读 David Goldberg 撰写的 What Every

Computer Scientist Should Know About Floating-Point Arithmetic 和 Bruce Dawson 撰写的 Comparing floating point numbers。

在 技术书店 浏览关于这些主题和其他技术主题的图书。

获得产品和技术

下载 Google C++ Testing Framework 软件。

下载 IBM 产品评估版 或者在 IBM SOA Sandbox 中在线试用来自 DB2®、Lotus®、Rational®、Tivoli® 和 WebSphere® 的应用程序开发工具和中间件产品。

讨论

阅读 developerWorks 博客 并加入 developerWorks 社区。

加入 My developerWorks 社区。

参与 AIX 和 UNIX 论坛：

- AIX 论坛
- AIX for developers 论坛
- 集群系统管理
- IBM Support Assistant 论坛
- 性能工具论坛
- 虚拟化论坛
- 更多 AIX 和 UNIX 论坛