Google Test 小结（基于 gtest version1.7）

一、编译与使用：

1、官方文档的方法:

第一次使用之前使用进入 gtest 目录，使用一下命令编译成链接文件

g++ -I./include -I./ -c ./src/gtest-all.cc

使用一下命令生成 libtest.a 静态库文件

ar -rv libgtest.a gtest-all.o

(ar 命令的 option flag 中 - 可加可不加， 即可以写为 ar rv libgtest.a gtest-all.o， option flag 自行 ar --help)

使用时编译器加 -pthread -I ${GTEST_DIR}/include 和 静态库 ${GTEST_DIR}/libgtest.a

（注 1： ${GTEST_DIR}是 gtest 的路径，也可以选择把/include ,和 libgtest.a 拷贝到代码文件夹里）

（注 2: 编译命令中静态库要在依赖文件之后，编译器是根据以来文件来展开静态库的）

（注 3: gtest 中使用了多线程，所以要使用-pthread，即使你的文件中没有使用多线程，链接的时候也要使用）

2、也可以将源码的编译直接写入 Makefile 文件，链接你的.o 文件和 gtest 的.o 文件

二、检查断言

1. gtest 中产生失败的分类

gtest 中的失败大致可以分为两类：致命的(fatal)和非致命的(nonfatal)

(1). 致命失败(fatal failure)可以有 ASSERT_*系列的断言和 FAIL()产生，致

命性的失败一般用于表示当前检查点失败会导致程序后续运行出现严重异常甚至崩溃的严重错误，gtest 的宏产生一个致命的失败之后会强制推出当前函数。

注意：由于 gtest 官方没有采用抛出异常的方法，所以 fatal failure 只能在返回值为 void 的函数中出现，否则会有编译错误。

(2). 非致命失败(nonfatal failure)可以有 EXPECT_*系列的断言宏以及 ADD_FAILURE()和 ADD_FAILURE_AT("path of your code file", line_number)产生。非致命失败不会结束当前函数，一般用于表示检查点失败的时候仅仅是对程序产生轻度的，不会影响后续进行的错误。nonfatal failure 可以用于其他返回值的函数。

2. 打印错误信息

gtest 中绝大多数情况下检查断言都可以使用重载过的流 S 操作符<<打印相关错误信息，应该注意的是，这个错误信息只会在检查失败的时候被打印出来，如果检查成功，则不打印。

3. 简单逻辑断言

gtest 中提供了最为简单的 bool 值断言宏，分别为

ASSERT_TRUE(condition)	EXPECT_TRUE(condition)	condition is true
ASSERT_FALSE(condition)	EXPECT_FALSE(condition)	condition is false

注意：(ASSERT|EXPECT)_(TRUE|FALSE)()可以接受除原生 bool 值之外的 gtest 的类::testing::AssertionResult。

4. 二元比较断言

(1). 可以比较的类型或者已重载过比较操作符的自定义类型，可以使用以下宏

ASSERT_EQ(val1, val2)	EXPCET_EQ(val1, val2)	val1 == val2
ASSERT_NE(val1, val2)	EXPECT_NE(val1, val2)	val1 != val2
ASSERT_LT(val1, val2)	EXPECT_LT(val1, val2)	val1 < val2
ASSERT_LE(val1, val2)	EXPECT_LE(val1, val2)	val1 <= val2
ASSERT_GT(val1, val2)	EXPECT_GT(val1, val2)	val1 > val2
ASSERT_GE(val1, val2)	EXPECT_GE(val1, val2)	val1 >= val2

（注： EQ ——equal， NE——not equal， LT——less than， LE——less than or equal to， GT——greater than， GE——greater than or equal to）

(2). 字符串的比较

ASSERT_STREQ(str1, str2)	EXPECT_STREQ(str1, str2)	包括宽字符字符串
ASSERT_STRNE(str1, str2)	EXPECT_STRNE(str1, str2)	
ASSERT_STRCASEEQ(str1, str2)	EXPECT_STRCASEEQ(str1, str2)	大小写不敏感，不包括宽字符
ASSERT_STRCASENE(str1, str2)	EXPECT_STRCASENE(str1, str2)	

(3). 浮点数比较

由于浮点数的储存特性，直接使用(ASSERT|EXPECT)_EQ 可能无法达到目的，可以使用以下的宏

ASSERT_FLOAT_EQ(val1, val2)	EXPECT_FLOAT_EQ(val1, val2)
ASSERT_DOUBLE_EQ(val1, val2)	EXPECT_DOUBLE_EQ(val1, val2)

官方给出的文档中说明两个浮点数相对的误差在 4ULPs 以内（Units in the

Last Place，最小精度单位）

另： 如果想要自定义误差范围，可以使用

(ASSERT|EXPECT)_NEAR(val1, val2, err)

其中 err 表示给定的误差范围，由自己给定

5. 表示成功与失败的断言

有时候我们不会实际去使用一个断言来测试一个值或者一个表达式，而仅仅是用来显式的表示检查点的成功或者失败。gtest 中提供了四个宏实现了这种功能，具体如下

(1). 表示成功： SUCCEED()

(2). 失败： FAIL() fatal failure

ADD_FAILURE() nonfatal failure

ADD_FAILURE_AT(“file_path”, line_number) nonfatal failure

（注 1： 官方给出的说明中，SUCCEED()支持流操作符<<的重载，但是由于检查成功所以并不会输出，没有实际的意义。后续的版本中可能会加入输出 SUCCEED()信息的输出。

注 2: ADD_FAILURE_AT()中可以加入当前文件与行号，更有利于错误的发现，建议使用 ADD_FAILURE_AT()而不是 ADD_FAILURE()。）

6. 对(ASSERT|EXPECT)_(TRUE|FALSE)的优化

如果只用(ASSERT|EXPECT)_(TRUE|FALSE)检查一个语句或者函数调用的布尔值，如果检查失败，可能得到的错误信息十分有限，为了能够更好的达到测试效果，可以做一些优化。

(1). 使用已有的 bool 返回值函数和 Predicate Assertions

ASSERT_PRED1(pred1, val1);	EXPECT_PRED1(pred1, val1);	pred1(val1) returns true
ASSERT_PRED2(pred2, val1, val2);	EXPECT_PRED2(pred2, val1, val2);	pred2(val1, val2) returns true

…	…	…

predn 是 n 元的谓词结构，带有 n 个参数，同时返回值为 bool 类型的函数，以后 n 个参数为宏调用函数 predn 时的实际参数。其中可以使用到五元的谓词结构，即 pred 这个函数的参数至多可以为 5。

这样做相对于使用*_(TRUE|FALSE)(predn(val1, …, valN))的好处是可以方便的更为全面的输出 val1, …，valN 的字面量和实际值。

(2). 使用::testing::AssertionResult 类代替 bool 类型以获得更多错误信息

gtest 中构建了一个类::testing::AssertionResult，支持流操作符<<，可以被(ASSERT|EXPECT)_(TRUE|FALSE)()处理。

如果说一个返回值为 bool 的函数其目的是要用于宏(ASSERT|EXPECT)_(TRUE|FALSE)()，那么其返回值可以使用::testing::AssertionResult 代替 bool，以获得更多的错误信息。其中 gtest 有::testing::AssertionSuccess()和::testing::AssertionFailure()的函数分别产生代表失败与成功的::testing::AssertResult 实例。

如：

```
::testing::AssertionResult IsEven(int n) {
  if ((n % 2) == 0)
    return ::testing::AssertionSuccess();
  else
    return ::testing::AssertionFailure() << n << " is odd";
}
```

代替

```
bool IsEven(int n) {
  return (n % 2) == 0;
}
```

那么当使用 EXPECT_TRUE(IsEven(n))检查失败的时候，流入::testing::AssertionFailure()的信息就会被输出。

(3). 如果你还不满意以上两点，你可以把它们结合起来，使用 Predicate-Formatter

将*_PREDn 换成*_PRED_FORMATn，同时，将 predn 改写为返回值为::testing::AssertionResult，前 n 个参数为 const char*类型，后 n 个参数为条件参数。如，

ASSERT_PRED_FORMAT3(pred3, val1, val2, val3)的 pred3 函数应当是这样定义的::testing::AssertResult pred3(const char*, const char*, const char*, Type1, Type2, Type3)

宏调用 predn 时，前 n 个参数是 val1, ..., valN 的字面量， 后 n 个参数是实际量。

7. 检测异常的抛出

用于检测异常抛出的宏：

ASSERT_THROW(statement,exception_type)	EXPECT_THROW(statement,exception_type)	statement throws an exception of the given type
ASSERT_ANY_THROW(statement)	EXPECT_ANY_THROW(statement)	statement throws an exception of any type
ASSERT_NO_THROW(statement)	EXPECT_NO_THROW(statement)	statement doesn't throw any exception

（注：这里的异常为未处理异常，如果在代码块或者语句中抛出了异常而被 Catch 的话， 异常是不会被检测到的）

8. 判断类型是否相等

使用模板函数::testing::StaticAssertTypeEq<T1, T2>()可以用于比较类型 T1 与 T2 是否相等。

9. 对于失败的捕获

类似于 c++中的异常抛出与捕获，gtest 中提供了对于各种断言产生的失败的捕获，include 头文件 gtest/gtest-spi.h 使用一下两个宏可以实现对于产生的失败的捕获：

(1) 捕获致命失败 EXPECT_FATAL_FAILURE(statement, substring);

(2) 捕获非致命失败 EXPECT_NONFATAL_FAILURE(statement, substring);

其中，statement 是会产生失败的语句， substring 是产生的错误信息的子串。

如果无法捕获相应类型的失败或者无法匹配 substring，抛出错误。

注意：

A) 这两个宏的不能通过流及操作符<<输出错误信息。

B) EXPECT_FATAL_FAILURE()中的 Statement 不能引用这个 test 的对象（也就是继承了::testing::Test 类的对象 this）的任何非静态的成员变量和非静态的方法。

C) EXPECT_FATAL_FAILURE()中的 statement 不能有返回值。

(3) 以上两个宏只能支持单线程，如果在 statement 语句中产生了新的线程，那么新的线程不会被追踪，新线程产生的错误将不会被捕获，如果要在新的线程中也追踪和捕获错误，可以使用一下的宏：

EXPECT_FATAL_FAILURE_ON_ALL_THREADS(statement, substring);
EXPECT_NONFATAL_FAILURE_ON_ALL_THREADS(statement, substring);

（即是加上_ON_ALL_THREADS 的后缀）

10. 断言的替换

前面说过，由于 gtest 官方没有采用抛出异常的方法，为了终止函数，使用了 return; 语句，所以 fatal failure 只能在返回值为 void 的函数中出现，否则会有

编译错误。如果一个返回值不为空的函数要出现致命失败，那么这能采取把返回值改写到参数列表的方法，不然就只能用相应的产生非致命失败的宏来做代替。推荐第一种，必要的致命性的失败更有助于保护测试程序的稳定性，不一定能够用非致命失败来代替。

三、测试案例

1. test_case_name 和 test_name

gtest 中的测试可以按照分类分为不同的案例 ,每个案例又有不同的测试实例，所以，对于每一个测试，都有两个标明身份的名词 test_case_name 和 test_name。

由于测试宏的展开中 ,会对 test_case_name 和 test_name 用预处理器连接为 test_case_name_test_name 的形式，所以这两个最好不要使用字符_开头或者结尾。

2. 开始测试

简单的 gtest 的 main 函数可以这样写

```
int main(int argc, char* argv[]) {
    ::testing::InitGoogleTest(argc, argv);
    return RUN_ALL_TEST();
}
```

其中 RUN_ALL_TEST()运行所有符合条件的测试 ,如果测试实例都通过返回 0,其他情况返回 1。

3. 简单测试实例 TEST()

使用 TEST(test_case_name, test_name) {

```
  //The statements that you want to test;
```

} 开始一个简单的测试实例。

4. Test Fixtures

如果同一批测试案例中多个测试共同使用一批数据类型，甚至是使用同一份数据，可以考虑将这些数据封装成一个测试案例类。

该类主要的要求如下：

(1). 类名为 test_case_name

(2). public 或 protected 继承自类 ::testing::Test

该类的可重载方法有：

(1).virtual void SetUp(); 单个案例执行之前，构造之后

(2).virtual void TearDown(); 单个案例执行之后，析构之前

(3).static void SetUpTestCase(); 这批案例执行之前

(4).static void TearDownTestCase(); 这批案例执行之后

如果有这批测试案例共享的成员变量，应当设置为静态变量，同时在SetUpTestCase()中配置资源，在 TearDownTestCase()中释放资源。

进行测试时使用 TEST_F(test_case_name, test_name)开始一个测试。

5. 全局测试的共有资源

在 fixture 类中的静态变量只能给这批测试案例的所有测试共享，如果要使得全局共享同一份数据的话，可以将它封装在一个类里，这个类继承::testing::Environment 类。

除此之外，为了使得所有测试可见，还需要在 main 函数里添加这个全局类，调用 ::testing::AddGlobalTestEnvironment(::testing::Environment *) 函数来实现。

6. 死亡测试

死亡测试是为了测试极端条件中代码崩溃或者异常退出的情况，能够比较安全的测

试这些情况，同时获取相关信息。注意：死亡测试的 test_case_name 后面必须有 DeathTest 的后缀，以便 gtest 优先运行，其主要目的是为了线程安全。

(1)(ASSERT|EXPECT)_DAETH(statement, regex)

regex 是用于匹配错误信息的正则表达式

statement 代码异常退出或者被中断，与此同时 stderr 中的错误信息与 regex 匹配否则打印错误信息。

(2)(ASSERT|EXPECT)_DEATH_IF_SUPPORTED(statement, regex)

如果系统支持死亡测试则进行死亡测试，否则是空宏

(3)(ASSERT|EXPECT)_EXIT(statement, predicate, regex)

Predicate 是一个返回值为 bool，且参数为一个 int 类型的值，当返回值为 true 的时候，death test 才有可能成功。

Gtest 提供了如下两个常用函数：

a) ::testing::ExitedWithCode(exit_code) 当 statement 以 exit_code 异常退出时返回值为 true

b) ::testing::KilledBySignal(signal_code) 当 statement 以 signal_code 被 killed 的时候返回 true

(4). Regex 正则表达式 gtest 可以使用两种风格，可以使用 GTEST_USES_SIMPLE_RE=1 和 GTEST_USES_POSIX_RE=1 来设置，默认为 POSIX 风格

(5). 死亡测试的运行方式

默认为 testing::FLAGS_gtest_death_test_style = "fast";也可以设置为线程安全 testing::FLAGS_gtest_death_test_style = "threadsafe";

fast 模式没有考虑到线程安全问题，运行速度可能略优于 threadsafe 模式。同时死亡测试中如果有多线程，运行测试程序的过程中输出可能会给予相

关警告。同时，即使使用了线程安全模式，也不能保证线程绝对安全，有可能出现死锁等安全问题。

这个运行方式可以在 main 函数中设置，也可以在某次测试中单独设置，还可以通过命令行参数的方式设置。

7. Value Parameterized Tests（多值测试）

如果相对同一个测试做同一对象不同值的测试，可以使用多值测试来避免代码的重复。

如果想对同一个测试进行多值测试，可以使用一个类继承::testing::TEST 还有::testing::TestWithParam<T>，类型 T 是测试值的类型。

然后使用 TEST_P(case_name, test_name)开始一个测试样例，在 TEST_P 中 使用 GetParam()获得当前值。注意 case_name 与类名一致。

用 INSTANTIATE_TEST_CASE_P(instantiation_name, case_name, param_list)获得参数列表，gtest 提供了以下方式获得初始参数列表 param_list：

Range(begin, end[, step])	从 begin 开始直至(end - 1)，range 默认值是 1，该值具有递增的操作
Values(v1, v2, ..., vN)	由 v1, v2, ...，vN 构成的集合
ValuesIn(container) ValuesIn(begin, end)	容器或者迭代器范围内的数据集合
Bool()	集合{true, false}
Combine(g1, g2, ..., gN)	集合（g1, g2, ...，gN)与自身的笛卡尔积（要有

Range(begin, end[, step])	从 begin 开始直至(end - 1) ,range 默认值是 1， 该值具有递增的操作
	头文件<tr1/tuple>）

注意：INSTANTIATE_TEST_CASE_P()会实例化所有该案例测试，不论在给宏的前面还是后面。

8. 多类型测试(Typed Tests)

为了测试模板类和模板函数，我们可以使用 Typed Tests 来简化代码。使用一个模板类继承::testing::Test 类，模板类中包含测试的模板对象。

使用一下语句初始化类型列表：

typedef ::testing::Types<**Your type list**> MyTypes;
TYPED_TEST_CASE(FooTest, MyTypes);

应当注意的是 ,为了能使宏 TYPED_TEST_CASE 正确的展开 ,这里的 typedef 是必要的。

最后使用 TYPED_TEST(test_case_name, test_name)开始一个测试样例 ,这个样例会逐一使用类型列表里的所有类型实例化。

9. Type-Parameterized Tests

与 Typed Tests 类似 ,首先定义一个类 ,用 TYPED_CASE_NAME_P(class_name) 声明进行 Type-Parameterized Tests,每个测试使用 TYPED_TEST_P(class_name, test_name)，接着使用 REGISTER_TYPED_TEST_CASE_P(class_name, ...)（后面为所有的测试名）注册测试，最后使用如下语句初始化类型列表：

typedef ::testing::Types<type list> MyTypes;
INSTANTIATE_TYPED_TEST_CASE_P(typed_param_test_name, class_name, MyTypes);

（注意： 在以上两个方法中在 test 中模板类型为 TypeParam。同时，如果你的类型列表里只有一种类型（虽然这样使用这种测试案例意义不大），那么你可以直接用 INSTANTIATE_TYPED_TEST_CASE_P(typed_param_test_name, class_name, type_name);代替）

10. 私有成员的测试

(1)

(2). 将 fixture 类定义为友元类。

(3). 在类中添加 FRIEND_TEST（test_case_name,test_name)宏，这样做会把同样 test_case_name 并且同样 test_name 的测试加入测试类的友元。

11. 重写事件监听器

事件监听器的类继承 gtest 给的监听器类 ::testing::TestEventListener 或者::testing::EmptyTestEventListener，在 gtest.h 里声明，在 main 函数里使用单例类::testing::UnitTest 的来获取监听器控制类 listeners 的实例，具体为 ::testing::UnitTest::GetInstance()->listeners().Append(*testing::TestEventListener);添加监听器。

（注 1、 虽然::testing::EmptyTestEventListener 是实现了抽象基类的方法，但是都是空函数；

注 2、使用自定义的 listener 并不会删除原来的监听器，要关闭原来默认的监听器可以使用 delete listeners.Release(listeners.default_result_printer())。但是不建议这样做，gtest 默认的监听器打印的信息已经足够多了，可以做一些补充，但是删除重写没有必要）

::testing::EmptyTestEventListener 及其基类抽象类::testing::TestEventLinstener 主要有一下几个方法：

(1) void OnTestProgramStart(::testing::UnitTest&)

测试程序开始时执行

(2) void OnTestIterationStart(::testing::UnitTest&, int /*iteration*/)

如果命令行参数--gtest_repeat=被设置，每一个重复测试开始之前执行，iteration 是当前测试的序号，从零开始

(3) void OnEnvironmentsSetUpStart(const ::testing::UnitTest&)

已添加的全局变量（继承自::testing::Environment)的 SetUp()方法调用之前调用。

(4) void OnEnvironmentsSetUpEnd(const ::testing::UnitTest&)

已添加的所有全局变量的 Setup()方法调用完毕之后调用。

(5) void OnTestCaseStart(const ::testing::TestCase&)

同一批 Case 执行之前，如果有 Fixture 类的话，在 Fixture 类的静态方法 SetUpTestCase()调用之前调用。

(6) void OnTestStart(const ::testing::TestInfo&)

每一个 test 开始时调用，可以使用 TestInfo 类获得当前测试的相关信息。

(7) void OnTestEnd(const ::testing::TestInfo&)

每一个测试结束之后调用。

(8) void OnTestPartResult(const ::testing::TestPartResult&)

test 中产生错误的时候输出（不包含被 EXPECT_*_FAILURE*()宏处理过的错误）错误信息之后，OnTestEnd()之前调用。 ::testing::TestPartResult 中包含这一次测试的错误相关信息。

(9) void OnTestCaseEnd(const ::testing::TestCase&)

一批 test 的所有测试完成之后调用。

(10) void OnEnvironmentsTearDownStart(const ::testing::UnitTest&)

全局量的 TearDown()被调用之前。

(11) void OnEnvironmentsTearDownEnd(const ::testing::UnitTest&)

所有全局量的 TearDown()执行完之后。

(12) void OnTestIterationEnd(const ::teseting::UnitTest&, int /*iteration*/)

(13) void OnTestProgramEnd(const ::testing::UnitTest&)

（关于 TestInfo，UnitTert，TestCase，TestPartResult 的方法均在头文件 gtest/gtest.h 中定义）

四、命令行参数

1. 进行部分测试 --gtest_filter=pattern

   * 通配字符串

   ？ 通配单字符

   - 除去某个 pattern

   ： 格式列表的间隔

   如--gtest_fliter=caseA.*:testB.*-caseA.？

   运行 test_case_name 为 caseA 中 test_name 不为单字符的测试和 caseB 中所有测试

2. 遇到失败时终止 --gtest_break_on_failure

3. 重复测试 --gtest_repeat=n 重复测试 n 次，当 n 为负数时无限重复

4. --gtest_also_run_disabled_tests 运行 disabled 的测试 （disabled 的测试需要在 test_case_name 上加前缀 DISABLED_）

5. 使用 --gtest_output=xml[:path] 生成 xml 格式的输出

6. --gtest_list_tests 不执行测试，列出所有测试（包括 Disabled）

7. 使用-？、/?、--help 获得命令行参数帮助