

# GoogleTest Primer

C++ 测试驱动开发 (TDD) 工具 GoogleTest 入门指南

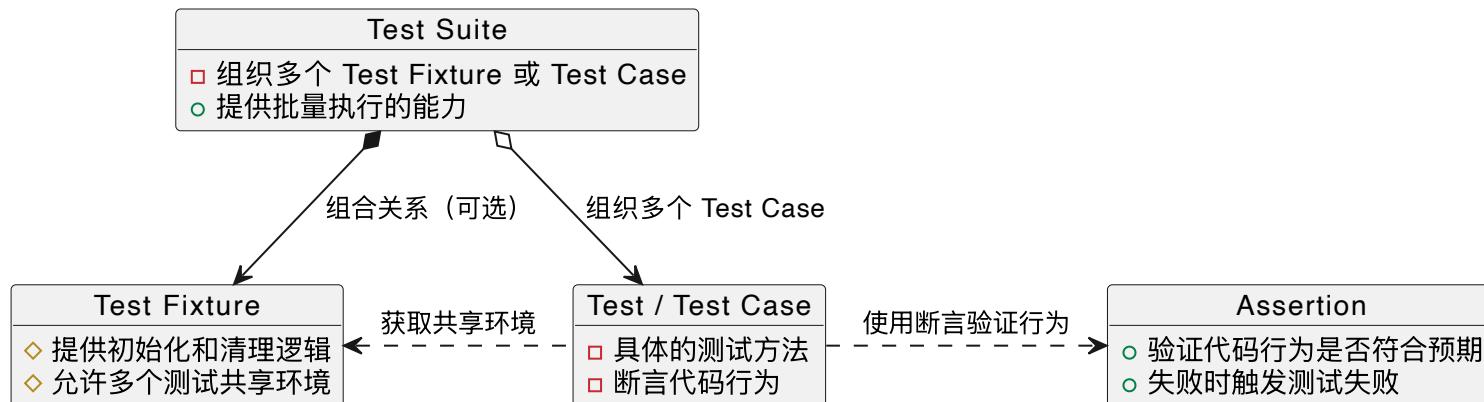
工欲善其事，必先利其器。—《论语·卫灵公》

# 为什么选择 GoogleTest?

- 🧑 帮助编写更好的 C++ 测试
- 😊 基于 xUnit 架构，易于上手
- ✅ 支持多平台 (Linux、Windows、Mac)
- 🛡 支持各种类型的测试，不仅限于单元测试
- ✅ 优秀测试的要点 & GoogleTest 的作用：
  - 独立 & 可重复：通过对象隔离每个测试，并支持单独运行失败测试以便快速调试
  - 组织良好：通过测试套件分组相关测试，便于维护和跨项目协作
  - 可移植 & 可复用：支持多平台、多编译器，适用于不同配置，确保测试代码的跨平台性
  - 提供详细失败信息：失败时提供充分的错误信息，允许非致命失败，使单次测试循环能发现多个问题
  - 简化测试配置：自动管理测试集合，无需手动枚举，让开发者专注于测试内容
  - 执行速度快：支持测试间共享资源，仅进行一次初始化 / 清理，避免相互依赖，提高执行效率

# 关键术语

- **Assertion (断言)** : 检查条件是否为真的语句，结果可以是成功，非致命故障或致命故障
- **Test / Test Case (测试用例)** : 针对特定功能或行为的单个测试，使用断言验证代码是否按预期工作
- **Test Suite (测试套件)** : 一组相关测试的集合，通常用于组织测试用例
- **Test Fixture (测试夹具)** : 为测试套件中的测试提供统一的初始化和清理逻辑，以及共享的环境



# 编写第一个测试

```
#include <gtest/gtest.h>

#include "me.h"

using namespace testing;      // GoogleTest 原生
using namespace testing::ext; // HUAWEI 扩展

// TestSuiteName / TestName 使用大驼峰，避免下划线
// !!! 测试名称应具有描述性，便于理解测试目的
TEST(MeTest, TestGradeIsCorrect) {
    Me me(90);
    ASSERT_EQ(me.GetGrade(), 90); // 致命断言
    // ...
}

// 同一组测试，TestSuiteName 应相同
HWTEST(MeTest, TestGradeDetermineGoodMood, Function | SmallTest | Level0) {
    Me me1(90);
    EXPECT_TRUE(me1.IsGoodMood()); // 非致命断言

    Me me2(50);
    ASSERT_FALSE(me2.IsGoodMood());
}
```

```
// me.h
class Me {
public:
    Me(int grade) : grade_(grade) {}

    int GetGrade() const {
        return grade_;
    }

    bool IsGoodMood() const {
        return grade_ ≥ 60;
    }

private:
    int grade_;
};
```

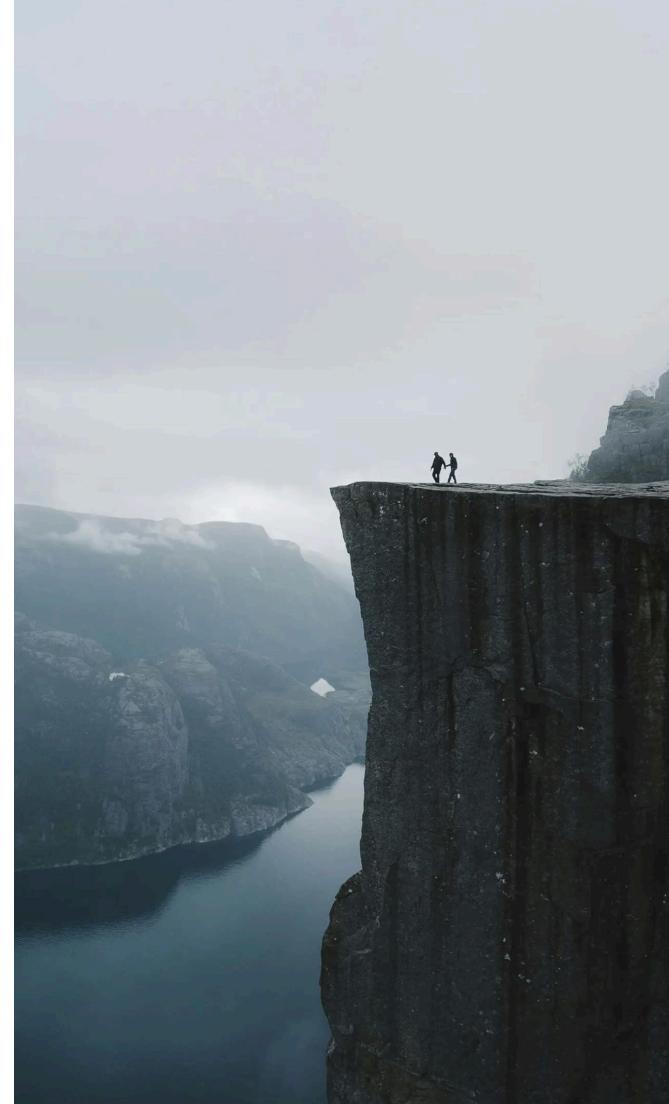
# 断言 (Assertion)

GoogleTest 提供了一系列断言宏 (assertion macros)，用于验证代码的行为。使用这些宏时，需要包含头文件 `#include <gtest/gtest.h>`。

## 两类断言

- `EXPECT_*`：非致命断言，失败时继续执行。通常优先使用 `EXPECT_*` 以报告多个故障，但若失败后继续运行无意义（如空指针），应使用 `ASSERT_*`。
- `ASSERT_*`：致命断言，失败时终止测试。可能跳过清理代码导致资源泄漏，若堆检查器报错，应考虑是否由此引起并决定修复与否。
- 所有断言宏都支持通过 `<<` 操作符追加自定义错误消息，例如：

```
EXPECT_TRUE(my_condition) << "My condition is not true";
```



# 显式成功和失败断言

这些断言用于直接生成成功或失败，而不是验证特定值或表达式。适用于流程控制决定测试结果的情况，例如：

```
switch(expression) {
    case 1:
        ... // 一些检查
    case 2:
        ... // 另一些检查
    default:
        FAIL() << "We shouldn't get here."; // 不应到达此处
}
```

- `SUCCEED()`：生成成功（仅用于文档记录，当前不会影响测试结果）。
- `FAIL()`：生成致命失败，立即返回当前函数（仅适用于返回 `void` 的函数）。
- `ADD_FAILURE()`：生成非致命失败，允许函数继续执行。
- `ADD_FAILURE_AT(file_path, line_number)`：在指定文件和行号处生成非致命失败。

可以使用 `GTEST_SKIP()` 宏在运行时阻止进一步执行测试。

```
GTEST_SKIP() << "Skipping this test due to some condition.;"
```

# 广义断言 (Generalized Assertion)

`EXPECT_THAT(value, matcher)` / `ASSERT_THAT(value, matcher)` 使用匹配器 (`matcher`) 对值进行验证，示例：

```
#include <gmock/gmock.h> // ! 广义断言宏在 gmock 中

using namespace testing;

EXPECT_THAT(value1, StartsWith("Hello"));           // 验证字符串以 "Hello" 开头
EXPECT_THAT(value2, MatchesRegex("Line \\\d+"));     // 验证匹配正则表达式
ASSERT_THAT(value3, AllOf(Gt(5), Lt(10)));         // 验证 value3 在 (5,10) 之间
ASSERT_THAT(value4, Each(Ne(0)));                   // 验证 value4 中每个元素都不等于 0
```

如果 `EXPECT_THAT(value1, StartsWith("Hello"))` 失败，错误信息将类似：

```
Value of: value1
Actual: "Hi, world!"
Expected: starts with "Hello"
```

# 基本断言

- `EXPECT_TRUE(condition) / ASSERT_TRUE(condition)`
- `EXPECT_FALSE(condition) / ASSERT_FALSE(condition)`

# 二元比较

💡 以下断言也适用于窄字符串对象和宽字符串对象（`string` 和 `wstring`）

- `EXPECT_EQ(val1, val2) / ASSERT_EQ(val1, val2)`
- `EXPECT_NE(val1, val2) / ASSERT_NE(val1, val2)`
- `EXPECT_LT(val1, val2) / ASSERT_LT(val1, val2)`
- `EXPECT_LE(val1, val2) / ASSERT_LE(val1, val2)`
- `EXPECT_GT(val1, val2) / ASSERT_GT(val1, val2)`
- `EXPECT_GE(val1, val2) / ASSERT_GE(val1, val2)`

## C 字符串比较

- EXPECT\_STREQ(str1, str2) / ASSERT\_STREQ(str1, str2)
- EXPECT\_STRNE(str1, str2) / ASSERT\_STRNE(str1, str2)
- EXPECT\_STRCASEEQ(str1, str2) / ASSERT\_STRCASEEQ(str1, str2)
- EXPECT\_STRCASENE(str1, str2) / ASSERT\_STRCASENE(str1, str2)

## 浮点比较

- EXPECT\_FLOAT\_EQ(val1, val2) / ASSERT\_FLOAT\_EQ(val1, val2)
- EXPECT\_DOUBLE\_EQ(val1, val2) / ASSERT\_DOUBLE\_EQ(val1, val2)
- EXPECT\_NEAR(val1, val2, abs\_error) / ASSERT\_NEAR(val1, val2, abs\_error)

 Learn more

---

# 测试夹具 (Test Fixture)

为多个测试使用相同的数据配置

- 定义一个继承自 `testing::Test` 的测试夹具类，在类中定义需要共享的对象和方法
- 使用 `TEST_F()` / `HWTEST_F()` 宏定义测试，用于访问测试夹具

```
class FooTest : public testing::Test {  
protected:  
    // 😊 如果想临时跳过某个测试套件，可以在 SetUpTestSuite() 中调用 GTEST_SKIP()  
    static void SetUpTestSuite() { /* 初始化代码，第一个测试前运行 */ }  
    static void TearDownTestSuite() { /* 清理代码，最后一个测试后运行 */ }  
    // 😊 若成员变量需要是 const 类型，也可以在构造函数中进行初始化  
    void SetUp() override { /* 初始化代码，每个测试前运行 */ }  
    void TearDown() override { /* 清理代码，每个测试后运行 */ }  
};  
  
// 使用 TEST_F() / HWTEST_F() 宏从 Test Fixture 获取数据配置  
TEST_F(FooTest, Test1) { /* ... */ }  
HWTEST_F(FooTest, Test2, Function | SmallTest | Level0) { /* ... */ }
```

## ⚠ 注意事项：

- 如果测试修改了夹具中的共享资源（`static` 成员），应在测试结束前恢复原始状态，避免影响后续测试。
- 我应该使用测试夹具的构造函数 / 析构函数还是 `SetUp()` / `TearDown()`？

# 值参数化测试 (Value-Parameterized Tests)

值参数化测试 (Value-Parameterized Tests) 允许你为一个测试逻辑提供多组不同的输入数据，而无需为每个输入单独写一个 TEST\_F 测试函数，这对于测试相同逻辑但不同输入的情况非常有用。

```
#include <gtest/gtest.h>
#include <string>

// 需要测试的函数
bool IsPalindrome(const std::string& str) { /* ... */ }

// 继承 TestWithParam<T>
class PalindromeTest : public ::testing::TestWithParam<std::string> {};

// 使用 TEST_P() 定义测试逻辑
TEST_P(PalindromeTest, CheckIfPalindrome) {
    std::string input = GetParam(); // 获取当前测试参数
    EXPECT_TRUE(IsPalindrome(input)) << "Failed for input: " << input;
}

// INSTANTIATE_TEST_SUITE_P() 为每个参数单独生成测试用例，测试报告更清晰，单个参数对应用例失败时不会影响其他参数测试
INSTANTIATE_TEST_SUITE_P(
    ValidPalindromes, // 测试组名称
    PalindromeTest, // 绑定的测试夹具类
    ::testing::Values( "madam", "racecar", "level", "deified", "a", "" ) // 测试参数
);
```

# 类型参数化测试 (Typed Tests)

类型参数化测试 (Typed Tests) 的作用类似于模板测试，可以针对不同的类型运行相同的测试代码，而不必为每种类型单独编写测试用例。

```
#include <gtest/gtest.h>
#include <vector>
#include <list>
#include <deque>

// 定义测试夹具（模板）
template <typename T>
class ContainerTest : public ::testing::Test {
public:
    using ContainerType = T;
};

// 关联测试夹具和类型列表
using ContainerTypes = ::testing::Types<std::vector<int>, std::list<int>, std::deque<int>>;
TYPED_TEST_SUITE(ContainerTest, ContainerTypes);
// 使用 TYPED_TEST() 定义测试逻辑
TYPED_TEST(ContainerTest, CanInsertElements) {
    typename TestFixture::ContainerType container;
    container.push_back(42);
    EXPECT_EQ(container.size(), 1);
    EXPECT_EQ(container.front(), 42);
}
```

# 结论

- GoogleTest 是 C++ 测试框架的首选
- 提供强大的断言、测试夹具、参数化测试等功能
- 支持 TDD，提高代码质量与可维护性

## 参考:

- [GoogleTest User's Guide](#)
- [【C++】研发基本功 - GTest / GMock 单元测试实践手册](#)

## 学习资源:

- [IBM 测试和行为驱动开发入门 | Coursera](#)
- [徐昊 · TDD 项目实战 70 讲](#)

# Q&A

有什么问题吗? 😊