# Catch简介

Catch 是一个流行的 C++ 单元测试框架，它以其简单性、易用性和无需编写测试运行器而受到开发者的青睐。Catch 允许你使用 BDD（行为驱动开发）风格的语法来编写测试，同时也支持传统的 TDD（测试驱动开发）风格

**Catch特点:**

1. **单头文件**：Catch 只包含一个头文件 catch.hpp，无需额外的库或运行时组件。
2. **自动注册**：测试用例和测试套件自动注册，无需显式注册。
3. **丰富的断言**：提供了一系列丰富的断言宏，用于检查预期条件。
4. **测试套件**：允许将相关的测试组织到套件中。
5. **测试过滤器**：可以按名称过滤测试。
6. **测试报告**：生成易于理解的测试报告。
7. **跨平台**：支持多种操作系统和编译器。

# Catch基本用法

## 包含Catch头文件

#include “catch.hpp”

## 编写测试用例

### TEST\_CASE

使用TEST\_CASE 宏定义测试用例 ：

**TEST\_CASE**( "An addition that should fail" ) {

REQUIRE( 1 + 1 == 2 );

}

### TEST\_CASE\_METHOD

使用TEST\_CASE\_METHOD(TEST\_CASE变体)宏定义测试用例，专门用于在类上下文中定义测试方法。

使用 TEST\_CASE\_METHOD，你可以将测试逻辑组织在类中，这样可以利用类成员函数和成员变量来共享测试设置（fixture）和状态。这对于创建需要共享初始化代码或者需要在多个测试之间共享资源的测试非常有用。

在这个例子中，我们定义了一个 MyTestClass 类，并在类中使用了 TEST\_CASE\_METHOD 宏来定义两个测试方法。这些测试方法可以访问类的私有成员和保护成员，因为它们是类的成员。

#include <catch2/catch.hpp>

class MyTestClass {

private:

int sharedValue;

public:

MyTestClass() : sharedValue(42) {}

// 使用 TEST\_CASE\_METHOD 定义一个测试方法

TEST\_CASE\_METHOD(MyTestClass, "Test method 1") {

// 这里可以访问类的成员变量和函数

REQUIRE(sharedValue == 42);

}

// 定义另一个测试方法

TEST\_CASE\_METHOD(MyTestClass, "Test method 2") {

sharedValue = 100;

REQUIRE(sharedValue == 100);

}

};

int main() {

Catch::Session(); // 启动 Catch2 测试会话

return Catch::Session().run(); // 运行测试

}

## 编译和运行测试

使用支持 C++11 的编译器编译你的测试文件。Catch 测试可以在命令行中直接运行

# Catch进阶用法

## 测试套件

将相关的测试用例组织到一个测试套件中：

**TEST\_CASE**( "some test", "[mytag]" ) {

REQUIRE( 2 + 2 == 4 );

}

## 断言宏

Catch 提供了多种断言宏，如 `REQUIRE`, `CHECK`, `ASSERT` 等，用于不同的测试场景。

REQUIRE :

REQUIRE\_FALSE( someCondition );

### 基本断言

* **REQUIRE ：**检查条件为真；如果条件为假，测试将失败，并且测试用例停止执行。

REQUIRE(1 == 2); *// 测试将失败*

* **CHECK**: 同样检查条件是否为真，但即使条件为假，测试用例也会继续执行。

CHECK(1 == 2); *// 测试会继续执行*

* **ASSERT:** 基础断言，如果条件为假，会抛出异常。通常用于调试。

ASSERT(1 == 2); *// 如果条件为假，抛出异常*

### 比较断言

* **REQUIRE\_EQ**和**CHECK\_EQ**: 检查两个值或表达式是否相等。

REQUIRE\_EQ(1, 1); *// 测试通过*

CHECK\_EQ(1, 2); *// 测试失败，但继续执行*

* **REQUIRE\_NE**和**CHECK\_NE**: 检查两个值或表达式是否不相等。

REQUIRE\_NE(1, 2); *// 测试通过*

CHECK\_NE(1, 1); *// 测试失败，但继续执行*

### 布尔断言

* **REQUIRE\_TRUE** 和 **CHECK\_TRUE:** 检查表达式是否为真。

REQUIRE\_TRUE(true); *// 测试通过*

CHECK\_TRUE(someCondition); *// 测试失败，但继续执行*

* **REQUIRE\_FALSE** 和 **CHECK\_FALSE:** 检查表达式是否为假。

REQUIRE\_FALSE(false); *// 测试通过*

CHECK\_FALSE(someCondition); *// 测试失败，但继续执行*

### 异常断言

* **REQUIRE\_THROWS** 和 **CHECK\_THROWS:** 检查代码块是否抛出异常。

REQUIRE\_THROWS(throw std::runtime\_error("error")); *// 测试通过*

CHECK\_THROWS(someFunctionThatMightThrow()); *// 测试失败，但继续执行*

CHECK\_THROWS\_AS( expression, ExceptionType ) ; *//验证给定的表达式是否抛出特定类型的异常。这个宏允许你测试异常处理代码，确保当某个操作失败时，它会抛出预期的异常类型。*

* **REQUIRE\_NOTHROW** 和 **CHECK\_NOTHROW:** 检查代码块是否不抛出异常。

REQUIRE\_NOTHROW(someFunctionThatShouldNotThrow()); *// 测试通过*

CHECK\_NOTHROW(throw std::runtime\_error("error")); *// 测试失败，但继续执行*

### 容器和字符串断言

* **REQUIRE\_THAT**和**CHECK\_THAT**: 使用 Catch2 的匹配器来检查容器、字符串或其他值是否符合预期的模式。

REQUIRE\_THAT("Hello World", Catch::Contains("World")); *// 测试通过*

CHECK\_THAT(std::vector<int>{1, 2, 3}, Catch::Equals(std::vector<int>{1, 2, 3})); *// 测试失败，但继续执行*

### 近似值比较

* **REQUIRE\_CLOSE** 和 **CHECK\_CLOSE**: 检查两个浮点数是否足够接近。

REQUIRE\_CLOSE(1.0, 1.0001, 0.0001); *// 测试通过*

CHECK\_CLOSE(1.0, 1.0001, 0.0001); *// 测试失败，但继续执行*

* **REQUIRE\_CLOSE\_FRACTION** 和 **CHECK\_CLOSE\_FRACTION:** 检查两个浮点数的相对差异是否在允许的范围内。

REQUIRE\_CLOSE\_FRACTION(1.0, 1.01, 0.01); *// 测试通过*

CHECK\_CLOSE\_FRACTION(1.0, 1.01, 0.01); *// 测试失败，但继续执行*

## 参数化测试

Catch 支持参数化测试，允许你为不同的输入运行相同的测试用例。

GIVEN( "a vector of integers" ) {

WHEN( "the vector is empty" ) {

THEN( "it should have size 0" ) {

REQUIRE( vec.size() == 0 );

}

}

}

## 测试标签

给测试用例添加标签，可以用于过滤或排除特定的测试

TEST\_CASE( "a tagged test", "[tags]" ) {

REQUIRE( true );

}