华南理工大学

South China University of Technology

《High-level Language Programming Project》 Report

Project Name: Ztest: A C++ Unit Testing Framework

School :
Major :
Student Name :
Teacher :
Submission Date:

目录

1 系统需求分析

1.1 系统背景与动机

随着复杂系统架构设计能力的培养需求日益凸显, 掌握面向对象设计原则与模式化工程实践已成为高级软件工程教育的核心目标。这不仅需要理解类与对象间的动态协作关系, 更要具备通过设计模式解决架构难题的抽象思维能力。现有的单元测试框架 (如 Google Test) 具有上手难度较高, 并发支持有限, 报告系统过于简洁, 可拓展性一般等缺点, 我们团队计划开发一个提供一个灵活、高效且易于使用 (带图形用户界面 GUI) 的测试工具, 旨在提供一个直观、易用的环境, 方便开发人员和测试人员编写、运行和管理测试用例。该工具将支持多种测试类型（如单元测试、集成测试等），并提供详细的测试结果报告。

表 1: 主流测试框架对比

框架	GUI 支持	并发测试	报告系统	扩展性	数据驱动
Google Test(C++)	无	有限	基础	中等	不支持
JUnit (Java)	Eclipse 插件	支持	HTML/XML	高	支持
PyTest (Python)	第三方工具	优秀	丰富	优秀	支持
Catch2 (C++)	无	一般	简洁	中等	不支持
Ztest* (C++)	精美	优秀	丰富	高	支持

1.2 系统目标

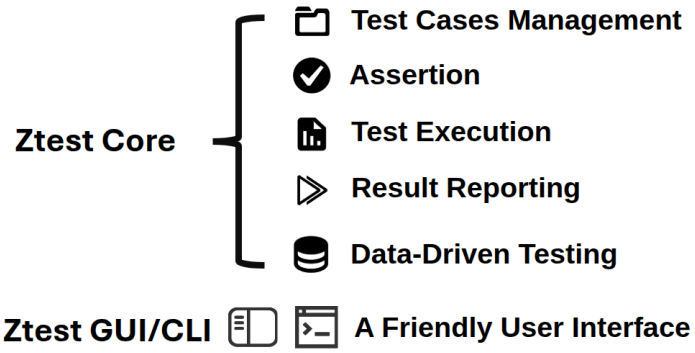


图 1: ztest function

1.2.1 测试管理

相较传统的单元测试框架,Ztest 支持多种测试类型, 如可并行运行且线程安全测试、需串行运行或线程不安全测试、通过迭代评估性能、参数化数据驱动测试。我们提供测试夹具用来管理单个测试。

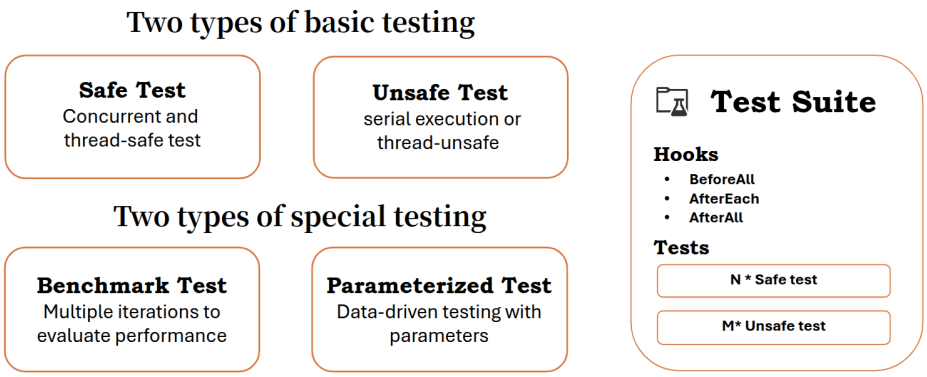


图 2: test type

我们提供类似 Google Test 的宏简化测试定义:

```
ZTEST_F(SuiteName, TestName, safe/unsafe) { ... } //define a test
ZBENCHMARK(SuiteName, TestName, iterations) { ... } // define a benchmark
ZTEST_P(SuiteName, TestName, data) { ... } //define a parameterized test
ZTEST_P_CSV(SuiteName, TestName, "data.csv") { ... } // define a parameterized
    test with csv data
```

同时, 我们能够链式定义测试用例:

```
auto test_case = TestFactory::createTest("Add", ZType::Z_SAFE, "", add, 2, 3)
                .setExpectedOutput(5)
                .beforeAll([]() { logger.info("Init\n"); })
                .afterEach([]() { logger.info("Clean\n"); }).build();
```

1.2.2 断言机制

断言机制用于验证测试用例的预期结果是否正确。框架提供了多种断言宏, 如 **EXPECT_EQ** 和 **ASSERT_TRUE**, 支持在测试中快速验证条件。主要提供以下断言:

- **EXPECT_EQ**: 验证两个值是否相等。
- **ASSERT_TRUE**: 验证条件是否为真。

使用样例如下,

```
// 如果断言失败,抛出异常
EXPECT_EQ(5, add(2, 3));
ASSERT_TRUE(6==add(2, 3));
```

如果断言失败, 抛出 **ZTestFailureException** 异常。同时可以通过继承 **ZTestFailureException** 异常处理函数, 自定义异常处理逻辑。

1.2.3 数据驱动测试

参数化测试是指在测试过程中，将测试用例中某些固定的数据替换为参数，然后通过改变参数的值来生成多组测试数据，从而对软件进行多次测试的方法。它允许测试人员用一组测试逻辑覆盖多种输入情况，而不是为每种情况编写单独的测试代码。数据驱动测试框架（Data-Driven Testing Framework）是一种将测试数据与测试脚本分离的自动化测试框架。它通过外部数据源（如 Excel、CSV 文件、数据库等）来驱动测试用例的执行。

ZTest 实现了参数测试并以从 csv 导入和解析数据的方式实现了数据驱动的测试。为了加速数据的访问，缓存导入的数据。使用懒加载机制，只有在需要时才加载数据到缓存中，减少不必要的资源消耗。同时，使用了 LRU 缓存淘汰策略用来确保缓存中的数据是最有可能被再次访问的，从而最大限度地发挥缓存的作用。

1.2.4 测试执行器

测试执行器负责管理测试用例的运行，支持**多线程**并行执行测试用例，并收集测试结果，实现了自动调度测试和测试结果统计。他的行为如下:

- 并行运行 safe 测试。线程的创建和销毁是一个相对耗时的操作。使用线程池管理线程的生命周期通过复用线程，避免了频繁的线程创建和销毁，从而显著提高了系统的性能。
- 串行运行 unsafe 测试。通过队列来维护测试用例，保证测试用例的顺序执行。
- 串行运行 Benchmark 测试。
- 串行运行 Parameterized 测试。

1.2.5 报告生成

能够生成 HTML,JSON 和 XML 报告，方便查看测试结果和用于 CI/CD。

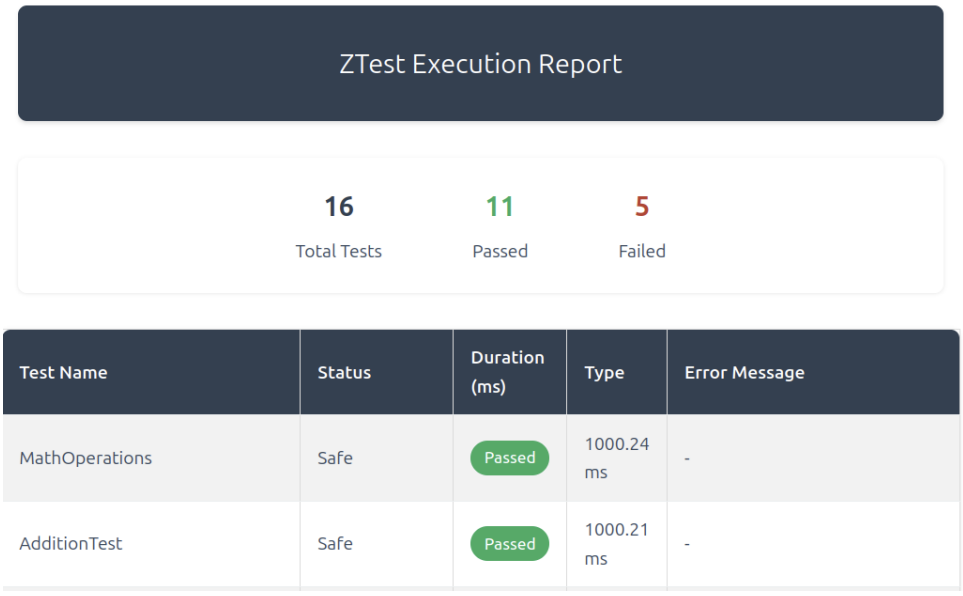


图 3: 测试结果展示界面布局示意图

1.2.6 CLI 接口

```
Usage: executor_name [OPTIONS]
Options:
--help                      Show help
--run-all                   Run all tests
--run-safe                  Run safe tests
--run-unsafe                Run unsafe tests
--run-benchmark             Run benchmark tests
--run-parameterized         Run parameterized tests
--run-test-case TEST_CASE   Run a specific test case
```

1.2.7 GUI 展示

可以通过 GUI 展示测试结果, 筛选测试结果, 观看系统资源状态, 查看某个测试的运行细节等功能。

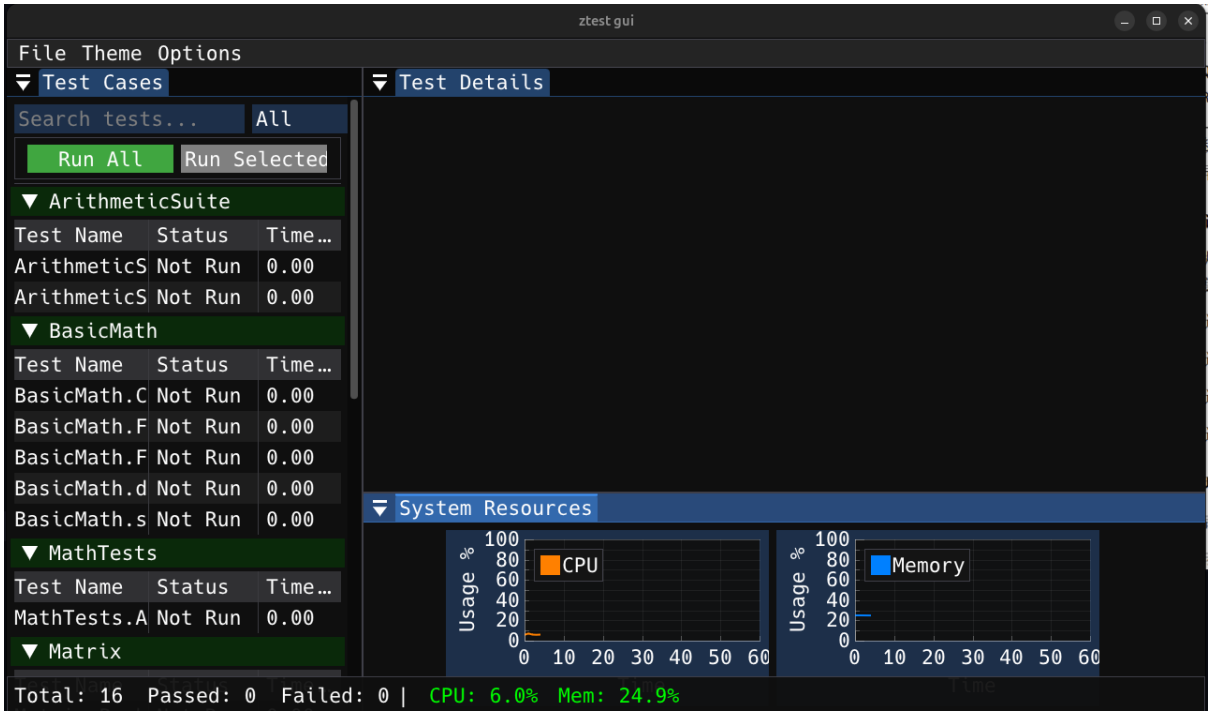


图 4: 测试管理界面布局示意图

1.2.8 AI 智能诊断功能

通过调用 qwen3 的接口，实现智能诊断功能，同时可以在 GUI 界面上获取对于某一个测试用例的具体分析。在报告中可以得到

1. 识别失败根本原因
2. 提供修复建议

3. 指出高风险测试用例

4. 评估整体测试覆盖率

5. 提出系统稳定性改进建议

2 程序分析

2.1 系统关键问题

2.1.1 提升测试运行效率

传统的 C++ 测试框架 (如 gtest,catch2) 多使用顺序运行测试的方法执行所有测试, 导致测试时间较长, 降低了生产效率。同时 vibe coding 的出现导致编写测试用例的时间大大降低, 而运行测试用例的时间仍然没有大的降低。我们将任务分为四个类别：短测试时间且更关注结果正确性的任务, 如加法运算、字符串拼接和用户登录验证；长测试时间且更关注结果正确性的任务, 如合并多个文件、复杂字符串匹配与替换和大量数据排序结果验证；短测试时间且更关注运行过程评估的任务, 如读取或写入大文件、低复杂度算法性能测试和数据库查询；以及长测试时间且更关注运行过程评估的任务, 如多线程处理任务、压力测试和时间复杂度高算法测试。

图 5: task type

对于测试所需时间长, 更见关注结果正确性, 且线程安全的任务,ZTest 在 C++ 测试框架体系中引入了多线程测试执行，直接融入测试框架，而不是 google-parallel 这样的第三方工具，实现了更高细粒度的控制。同时, 引入数据驱动测试并使用缓存加速使得大规模的单元测试的定义和执行速度大大提升, 意义重大。

2.1.2 语法糖的实现

如何让用户愿意使用单元测试框架？其关键在于语法定义的简易性。我们使用一系列复杂的宏定义来实现语法糖, 从而简化用户的定义。

2.1.3 待测函数的自动类型推导

我们设计在使用单个测试样例链式定义的时候, 我们希望用户只需要传入函数名、参数和期望结果, 就可以由为我们的框架接管运行的具体逻辑和测试结果的比较。这要求我们实现自动推导待测函数的参数类型, 以便调用函数用于测试和测试结果验证。我们使用工厂模式来实现返回值类型的指定, 使用建造者模式来实现参数的构造和期望结果的设定。

2.2 职责分配

- 学生 1: [任务 1]
- 学生 2: [任务 2]
- 学生 3: [任务 3]

3 技术路线

3.1 运行环境

表 2: 开发和运行环境

Component	Tool Used for Development
处理器	Intel i9-14900HX (32) @ 5.800GHz
操作系统	Ubuntu 24.04.2 LTS x86_64 (kernel 6.11.0-26-generic)
编译器	gcc 13.3.0 或 clang 18.1.3 (不可以使用 MSVC)
图形 API	glfw 3.4 + glad 4.0.1
GUI 框架	ImGui-1.91.7-docking
数据可视化工具	implot v0.16
构建系统	XMake v2.9.9+HEAD.40815a0
C++ 标准	C++20(必要)

3.2 总体设计

3.2.1 Ztest Core 架构图

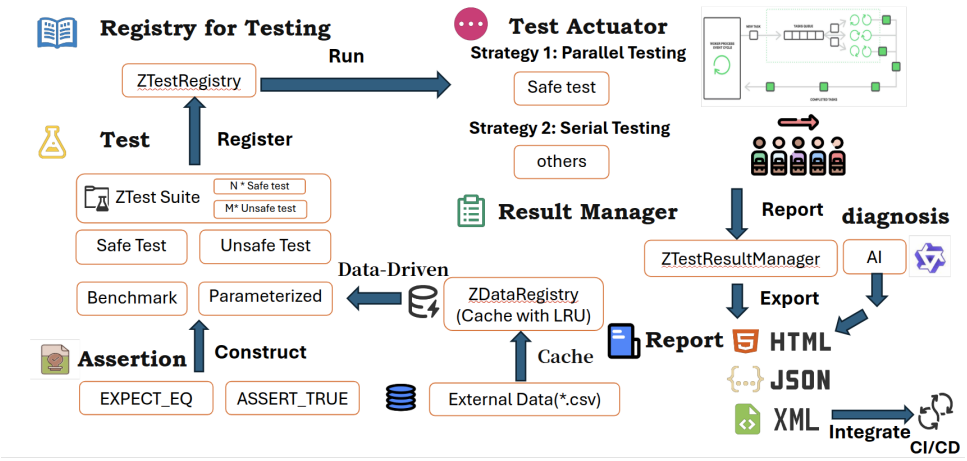


图 6: ztest design

ZTest 的核心架构流程开始于测试定义阶段，测试 (包括 ZTest Suite、安全测试、不安全测试、基准测试和参数化测试) 通过会使用断言定义, 如 EXPECT_EQ 和 ASSERT_TRUE, 来验证测试结果的正确性。接着，测试将被注册到 ZTestRegistry 进行统一的管理。注册完成后, 测试会被发送

到测试执行器, 该执行器根据测试的类型采取不同的测试策略, 对于安全测试采用并行测试策略, 而对于其他类型的测试则采用串行测试策略。在对数据驱动的测试的执行过程中, 数据驱动模块通过 ZDataRegistry（带有 LRU 机制的缓存）来管理外部数据, 如 CSV 文件, 以支持测试的进行。测试完成后, 结果管理器 ZTestResultManager 负责收集和处理测试结果, 并将这些结果导出为 HTML、JSON 或 XML 格式, 以便于报告和进一步分析。在导出为 HTML 格式时由 AI(qwen turbo) 进行测试诊断，并融入到 HTML 测试报告中。此外, 整个测试流程还支持与持续集成/持续部署（CI/CD）系统的集成, 使得测试结果可以自动地反馈到开发流程中, 从而提高软件开发的效率和质量。

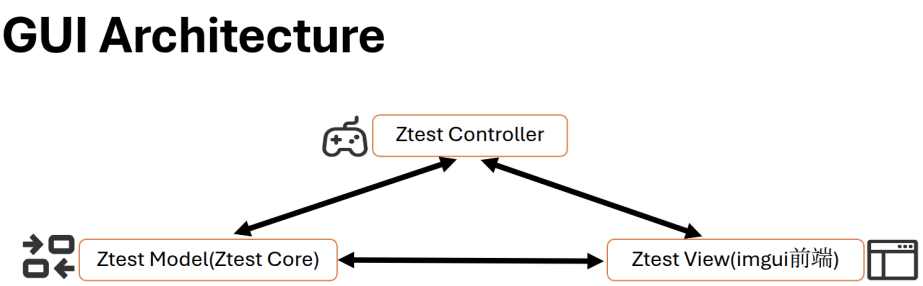


图 7: ztest gui architecture

3.2.2 GUI 框架

GUI 框架使用 MCV 架构开发，其中 Model 层负责数据处理，其中包含了对于 Ztest core 的进一步封装，View 层负责界面绘制，主要是使用 ImGui 框架绘制，Controller 层负责用户交互，将用户在 UI 界面上的点击和筛选等操作转换成对 Model 层和 View 层的调用，从而实现用户界面的更新和数据的更新。

图 8: ztest class

3.3 详细设计

3.3.1 测试定义相关类

- `ZTestInterface`: 测试接口，定义测试执行框架。
- `ZTestBase` (ztest_base.hpp): 抽象测试基类，封装通用属性与生命周期钩子。采用模板方法模式定义测试执行框架，通过虚函数表实现策略模式，利用钩子函数实现装饰器模式扩展。
- `ZTestSingleCase` (ztest_singlecase.hpp): 单例测试实现，使用模板方法模式执行测试逻辑，配合 TestFactory 体现工厂模式，支持链式配置符合建造者模式特征。
- `ZTestSuite`(ztest_suite.hpp): 测试套件类，继承自 ZTestBase，用于组织测试用例，实现模板方法模式，支持钩子函数扩展。
- `ZBenchMark` (ztest_benchmark.hpp): 基准测试类，继承 ZTestBase 并重写 run() 方法，通过迭代执行测试函数实现基准测试。
- `ZTestParameterized` (ztest_parameterized.hpp): 参数化测试基类，采用组合模式封装测试数据，通过 run() 方法实现数据驱动测试框架。
- 宏定义系统 (ztest_macros.hpp): 通过宏定义实现语法糖，采用工厂模式自动生成测试类，利用命名连接技术实现自动化注册。

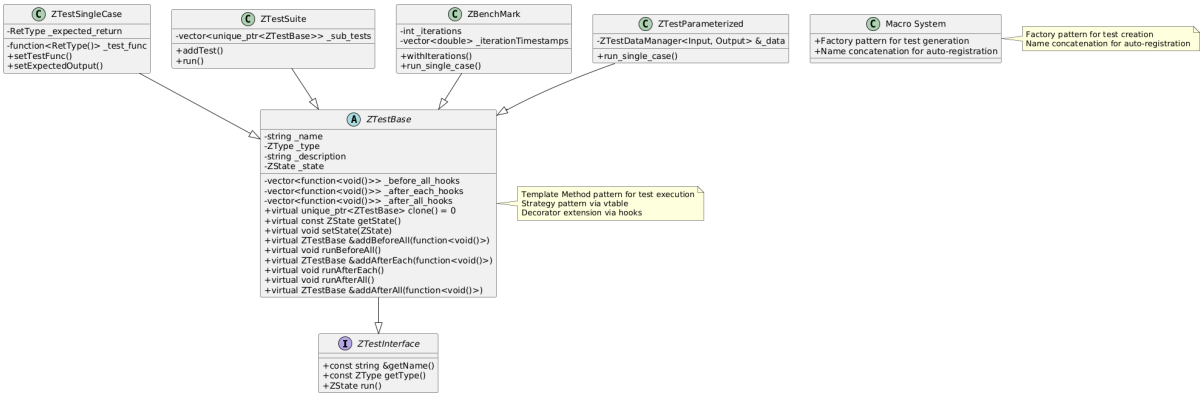


图 9: ztest class

3.3.2 测试注册相关类

- `ZTestRegistry` (ztest_registry.hpp): 单例模式实现的全局注册中心，管理测试用例集合并保障线程安全的注册/获取操作。

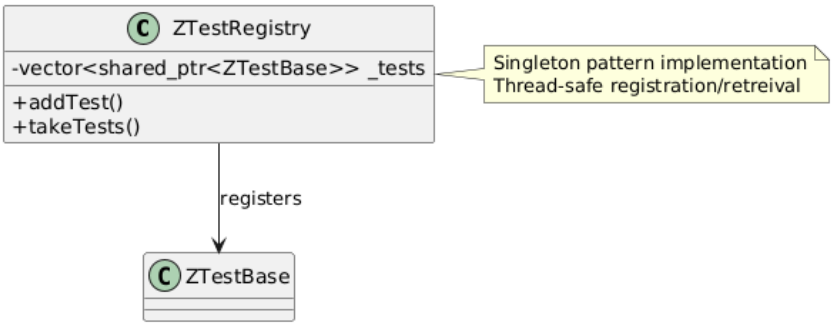


图 10: ztest class

3.3.3 测试执行相关类

- `ZTestContext` (ztest_context.hpp): 测试执行上下文，采用策略模式根据测试类型选择执行策略，通过线程池模式实现并行测试执行。
- `ZThreadPool` (ztest_thread.hpp): 线程池实现对象池模式，使用生产者-消费者模式管理任务队列，通过 future/promise 机制实现异步执行监控。

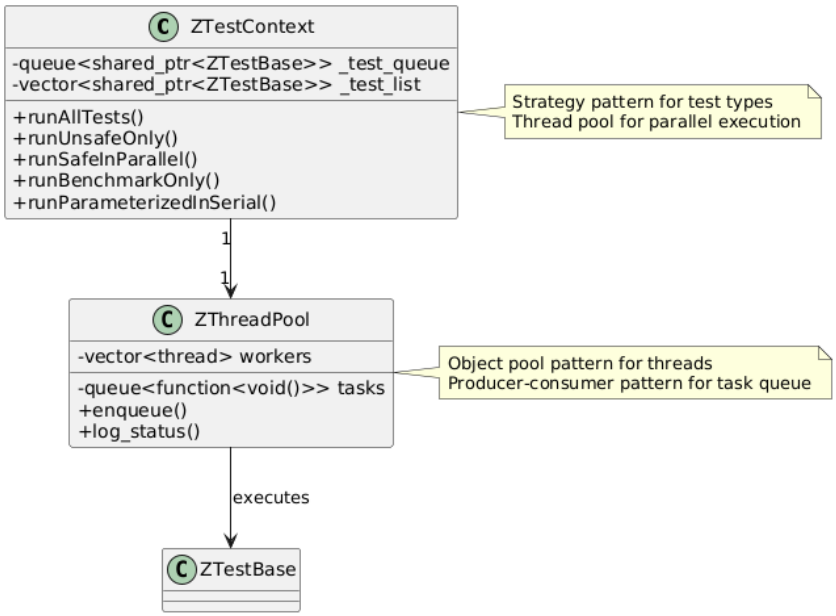


图 11: ztest class

3.3.4 数据管理相关类

- `ZDataRegistry` (ztest_dataregistry.hpp): 数据缓存管理器，采用单例模式全局访问，通过 LRU 策略实现缓存淘汰。
- `ZDataManager`(ztest_parameterized.hpp): 是数据管理类的抽象基类，提供数据管理接口，实现类似 python 中的迭代器的功能。
- `ZTestDataManager` (ztest_parameterized.hpp) 泛型类，实现了由用户在代码中指定的数据类型的测试数据管理功能。

- `ZTestCSVDataManager`(ztest_parameterized.hpp) 双继承于 `ZTestDataManager` 和 `ZDataManager`，实现了从 csv 文件中读取测试数据功能并处理成便于参数化测试的形式。

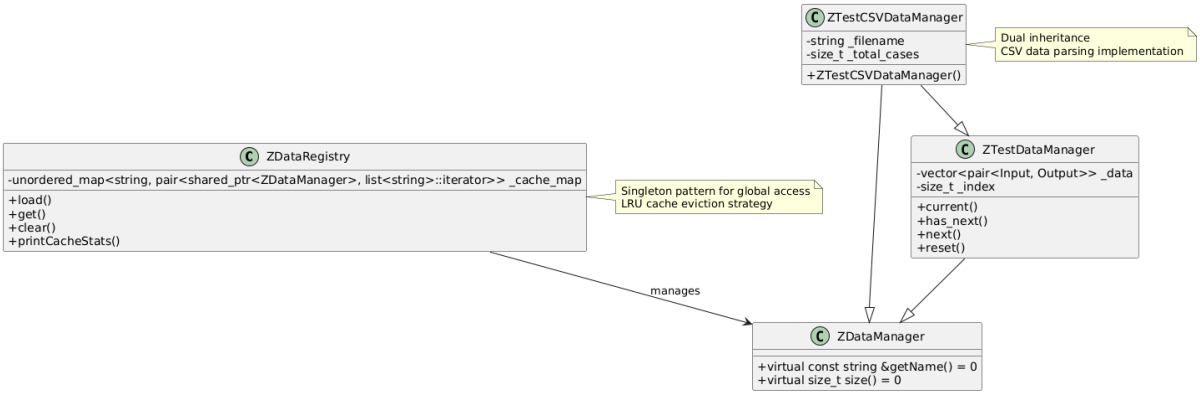


图 12: ztest class

3.3.5 结果管理相关类

- `ZTestResult` (ztest_result.hpp): 值对象模式的测试结果类，封装测试状态、耗时等不可变数据。
- `ZTestResultManager` (ztest_result.hpp): 单例模式实现的结果管理器，采用责任链模式处理结果存储与查询。

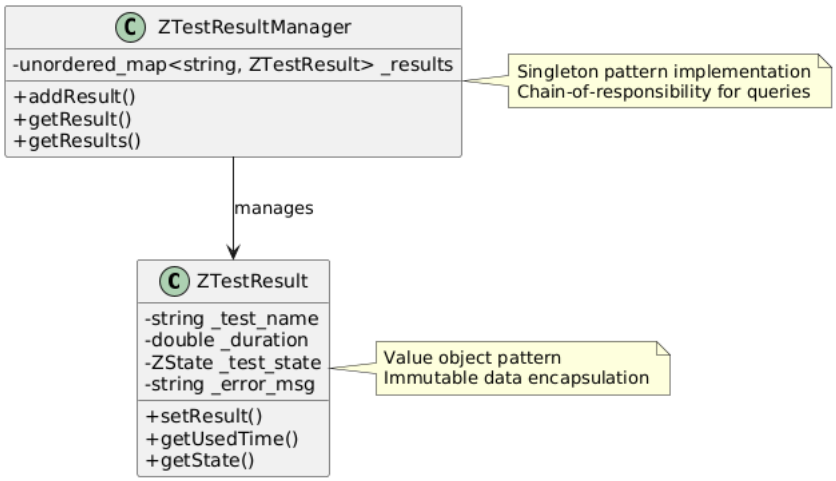


图 13: ztest class

3.3.6 报告生成相关类

- `ZLogger` (ztest_logger.hpp): 多格式报告生成器，应用模板方法模式定义报告生成流程，支持 HTML/JSON/JUnit 报告格式。

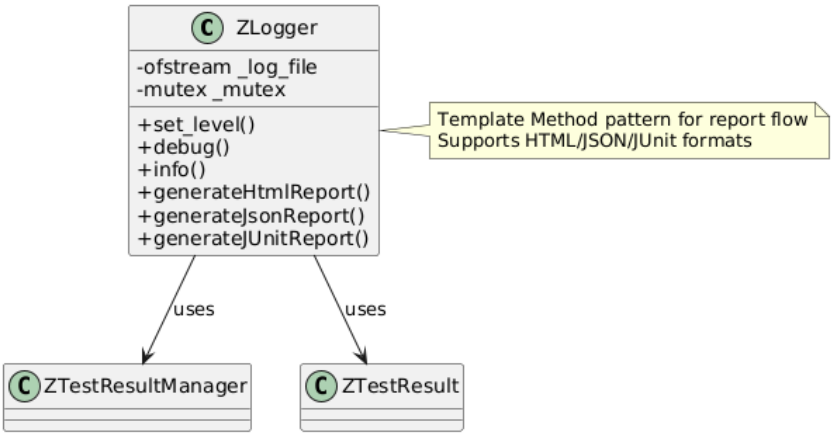


图 14: ztest class

3.3.7 工具模块相关类

- `ZTimer` (ztest_timer.hpp): RAII 模式实现的计时器，封装时间测量功能。
- `CSVStream`(ztest_utils.hpp)：实现了类似标准输入输出库的 CSV 流操作，支持读，写，打印基本信息。

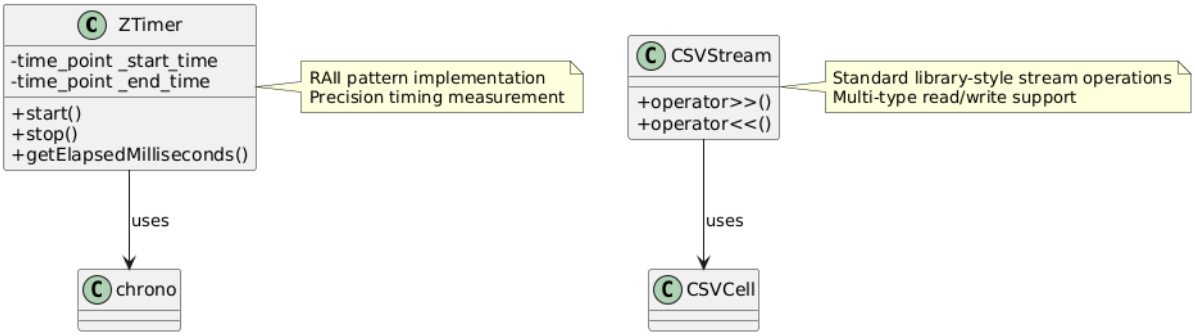


图 15: ztest class

3.3.8 GUI 模块相关类

- `ZTestModel` (gui.hpp): MVC 架构中的 Model 层，采用观察者模式监听测试状态变化。
- `ZTestController` (gui.hpp): MVC 架构中的 Controller 层，实现命令模式封装测试执行操作。
- `ZTestView` (gui.hpp): MVC 架构中的 View 层，使用桥接模式分离界面元素与实现。

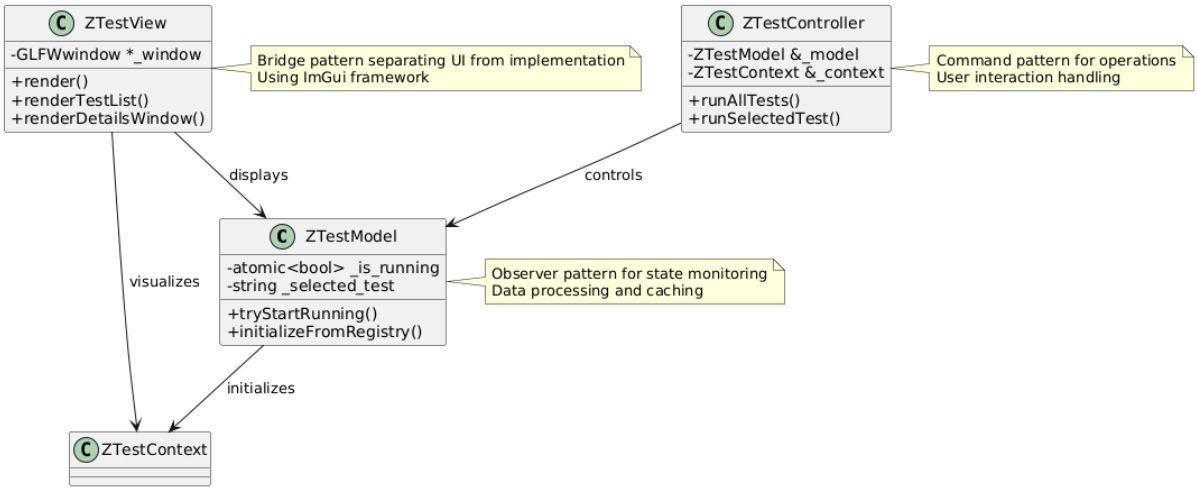


图 16: ztest class

4 编程进度

任务阶段	**计划**
确定主题 2024.04.26-2024.05.03	调查主题, 提交提案。
实现核心代码 2024.05.03-2024.05.05	实现了 GUI 和 Safe test 和 Unsafe test 的执行逻辑
重大性能优化 2024.05.05-2024.05.08	线程池优化
功能优化 2024.05.08-2024.05.11	完善 JSON，HTML 输出，JUnit 格式输出,CI 集成
功能优化 2024.05.11-2024.05.13	GUI 引入了 imgui Docking 功能
功能优化 2024.05.13-2024.05.15	增加 CLI
重大功能优化 2024.05.15-2024.05.24	增加 BENCHMARK 测试
功能优化 2024.05.24-2024.05.25	增加了设备状态监测和可视化
重大功能优化 2024.05.25-2024.05.28	增加了参数化测试并增加了数据驱动
重大性能优化 2024.05.28-2024.06.02	增加了对数据的缓存和 LRU 缓存淘汰技术
重大功能优化 2024.06.02-2024.06.07	添加了 AI 诊断
总结工作 2024.6.02-2024.06.08	跨平台移植 & 报告撰写

表 3: 编程进度

5 测试报告

5.1 功能测试 & 系统测试

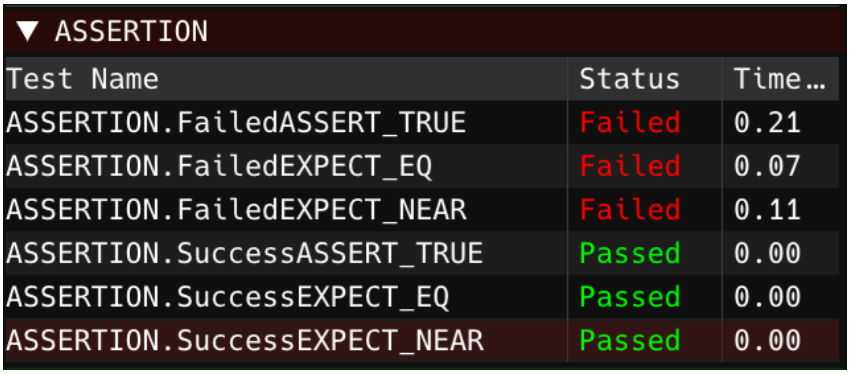
5.1.1 断言功能测试

该测试旨在验证 EXPECT_EQ、EXPECT_NEAR 和 ASSERT_TRUE 等断言宏在成功和失败情况下的行为。通过设计不同的测试用例，可以检查这些断言是否能够正确地识别预期结果与实际结果之间的匹配情况，从而确保测试框架的断言功能能够可靠地工作。

```
int add(int a, int b) { return a + b; }
double subtract(double a, double b) { return a - b; }
ZTEST_F(ASSERTION, FailedEXPECT_EQ) {
  EXPECT_EQ(6, add(2, 3));
  return ZState::z_success;
}
ZTEST_F(ASSERTION, SuccessEXPECT_EQ) {
  EXPECT_EQ(5, add(2, 3));
  return ZState::z_success;
}
ZTEST_F(ASSERTION, SuccessEXPECT_NEAR) {
  EXPECT_NEAR(2, subtract(5.0, 3.0), 0.001);
  return ZState::z_success;
}

ZTEST_F(ASSERTION, FailedEXPECT_NEAR) {
  EXPECT_NEAR(2, subtract(5.1, 3.0), 0.001);
  return ZState::z_success;
}

ZTEST_F(ASSERTION, FailedASSERT_TRUE) {
  ASSERT_TRUE(false);
  return ZState::z_success;
}
ZTEST_F(ASSERTION, SuccessASSERT_TRUE) {
  ASSERT_TRUE(true);
  return ZState::z_success;
}
```



▼ ASSERTION

Test Name	Status	Time…
ASSERTION.FailedASSERT_TRUE	Failed	0.21
ASSERTION.FailedEXPECT_EQ	Failed	0.07
ASSERTION.FailedEXPECT_NEAR	Failed	0.11
ASSERTION.SuccessASSERT_TRUE	Passed	0.00
ASSERTION.SuccessEXPECT_EQ	Passed	0.00
ASSERTION.SuccessEXPECT_NEAR	Passed	0.00

图 17: assertion function test

5.1.2 测试管理测试

本测试模块旨在验证测试管理系统的功能完整性与可靠性。通过定义多种测试用例，包括单个安全测试、单个不安全测试以及包含多个断言的测试套件，系统能够全面覆盖不同的测试场景。此外，利用动态测试用例构建功能，可以灵活地创建和注册新的测试用例，进一步增强了测试框架的可扩展性。测试过程中，通过内存分配、函数调用验证以及断言检查，确保了测试用例的正确执行与预期结果的一致性。同时，测试框架还提供了前置和后置钩子功能，用于在测试前后进行必要的设置和清理操作，保障了测试环境的稳定性和测试结果的准确性。

```
ZTEST_F(TESTMANAGE, safe_test_single_case, safe) {
  ASSERT_TRUE(true);
  return ZState::z_success;
}
ZTEST_F(TESTMANAGE, unsafe_test_single_case, unsafe) {
  ASSERT_TRUE(true);
  return ZState::z_success;
}
ZTEST_F(TESTMANAGE, test_suite) {
  const size_t MB100 = 100 * 1024 * 1024;
  auto ptr = std::make_unique<char[]>(MB100);
  ASSERT_TRUE(ptr != nullptr);
  EXPECT_EQ(3, subtract(5, 3));
  EXPECT_EQ(6, add(2, 3));
  return ZState::z_success;
}
void createSingleTestCase() {
  // Use TestBuilder to construct test
  auto test =
      TestFactory::createTest("AdditionTest",         // Test name
                              ZType::z_safe,          // Execution
                              "Test addition functionality", // Description
```

```
                        add, 2, 3 // Function and arguments
                        )
        .setExpectedOutput(5) // Set expected result
        .beforeAll([]() {  // Setup hook
          std::cout << "Setting up single test..." << std::endl;
        })
        .afterEach([]() { // Teardown hook
          std::cout << "Cleaning up after test..." << std::endl;
        })
        .withDescription("Verify basic addition")
        .registerTest()
        .build(); // Register with test
}
// in main()
createSingleTestCase();
```

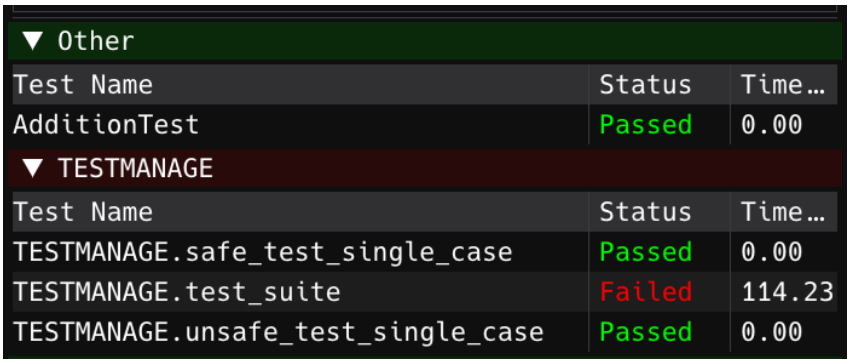


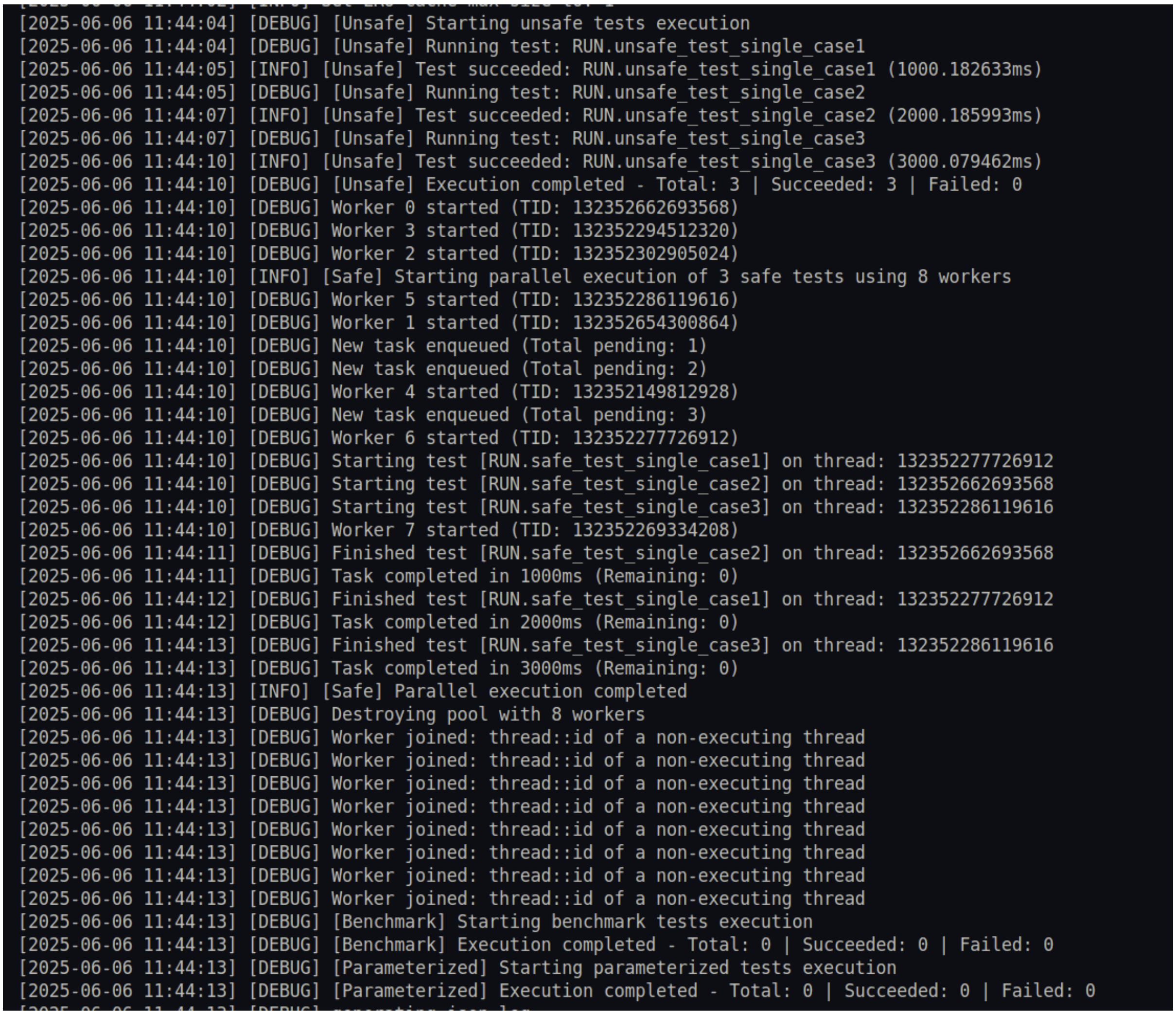
图 18: test management function test

5.1.3 测试运行管理

在本测试模块中，主要验证了测试运行管理机制的效率与准确性。通过定义多个安全和不安全的测试用例，分别模拟了不同执行时间、不同任务类型的测试场景。每个测试用例通过 sleep 函数模拟了不同的执行时间，以测试并行执行和顺序执行的性能差异。测试结果解释如下

- 使用一个具有八个线程的线程池，并行执行了三个测试用例
- 顺序执行了三个测试用例

```
ZTEST_F(RUN, safe_test_single_case1, safe) {
  sleep(2);
  ASSERT_TRUE(true);
  return ZState::z_success;
}
ZTEST_F(RUN, safe_test_single_case2, safe) {
  sleep(1);
  ASSERT_TRUE(true);
  return ZState::z_success;
}
ZTEST_F(RUN, safe_test_single_case3, safe) {
  sleep(3);
  ASSERT_TRUE(true);
  return ZState::z_success;
}
ZTEST_F(RUN, unsafe_test_single_case1, unsafe) {
  sleep(1);
  EXPECT_EQ(false, false);
  return ZState::z_success;
}
ZTEST_F(RUN, unsafe_test_single_case2, unsafe) {
  sleep(2);
  EXPECT_EQ(false, false);
  return ZState::z_success;
}
ZTEST_F(RUN, unsafe_test_single_case3, unsafe) {
  sleep(3);
  EXPECT_EQ(false, false);
  return ZState::z_success;
}
```



```
[2025-06-06 11:44:04] [DEBUG] [Unsafe] Starting unsafe tests execution
[2025-06-06 11:44:04] [DEBUG] [Unsafe] Running test: RUN.unsafe_test_single_case1
[2025-06-06 11:44:05] [INFO] [Unsafe] Test succeeded: RUN.unsafe_test_single_case1 (1000.182633ms)
[2025-06-06 11:44:05] [DEBUG] [Unsafe] Running test: RUN.unsafe_test_single_case2
[2025-06-06 11:44:07] [INFO] [Unsafe] Test succeeded: RUN.unsafe_test_single_case2 (2000.185993ms)
[2025-06-06 11:44:07] [DEBUG] [Unsafe] Running test: RUN.unsafe_test_single_case3
[2025-06-06 11:44:10] [INFO] [Unsafe] Test succeeded: RUN.unsafe_test_single_case3 (3000.079462ms)
[2025-06-06 11:44:10] [DEBUG] [Unsafe] Execution completed - Total: 3 | Succeeded: 3 | Failed: 0
[2025-06-06 11:44:10] [DEBUG] Worker 0 started (TID: 132352662693568)
[2025-06-06 11:44:10] [DEBUG] Worker 3 started (TID: 132352294512320)
[2025-06-06 11:44:10] [DEBUG] Worker 2 started (TID: 132352302905024)
[2025-06-06 11:44:10] [INFO] [Safe] Starting parallel execution of 3 safe tests using 8 workers
[2025-06-06 11:44:10] [DEBUG] Worker 5 started (TID: 132352286119616)
[2025-06-06 11:44:10] [DEBUG] Worker 1 started (TID: 132352654300864)
[2025-06-06 11:44:10] [DEBUG] New task enqueued (Total pending: 1)
[2025-06-06 11:44:10] [DEBUG] New task enqueued (Total pending: 2)
[2025-06-06 11:44:10] [DEBUG] Worker 4 started (TID: 132352149812928)
[2025-06-06 11:44:10] [DEBUG] New task enqueued (Total pending: 3)
[2025-06-06 11:44:10] [DEBUG] Worker 6 started (TID: 132352277726912)
[2025-06-06 11:44:10] [DEBUG] Starting test [RUN.safe_test_single_case1] on thread: 132352277726912
[2025-06-06 11:44:10] [DEBUG] Starting test [RUN.safe_test_single_case2] on thread: 132352662693568
[2025-06-06 11:44:10] [DEBUG] Starting test [RUN.safe_test_single_case3] on thread: 132352286119616
[2025-06-06 11:44:10] [DEBUG] Worker 7 started (TID: 132352269334208)
[2025-06-06 11:44:11] [DEBUG] Finished test [RUN.safe_test_single_case2] on thread: 132352662693568
[2025-06-06 11:44:11] [DEBUG] Task completed in 1000ms (Remaining: 0)
[2025-06-06 11:44:12] [DEBUG] Finished test [RUN.safe_test_single_case1] on thread: 132352277726912
[2025-06-06 11:44:12] [DEBUG] Task completed in 2000ms (Remaining: 0)
[2025-06-06 11:44:13] [DEBUG] Finished test [RUN.safe_test_single_case3] on thread: 132352286119616
[2025-06-06 11:44:13] [DEBUG] Task completed in 3000ms (Remaining: 0)
[2025-06-06 11:44:13] [INFO] [Safe] Parallel execution completed
[2025-06-06 11:44:13] [DEBUG] Destroying pool with 8 workers
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] [Benchmark] Starting benchmark tests execution
[2025-06-06 11:44:13] [DEBUG] [Benchmark] Execution completed - Total: 0 | Succeeded: 0 | Failed: 0
[2025-06-06 11:44:13] [DEBUG] [Parameterized] Starting parameterized tests execution
[2025-06-06 11:44:13] [DEBUG] [Parameterized] Execution completed - Total: 0 | Succeeded: 0 | Failed: 0
```

图 19: test 执行器 function test

5.1.4 数据驱动测试

展示了两种不同类型的数据驱动测试用例，分别使用了内存中的数据集以及外部 CSV 文件作为测试数据源。

```
ZTestDataManager<vector<int>, int> sum_test_data = {
    {{1, 2}, 3}, {{-1, 1}, 0}, {{10, 20}, 30}};
ZTEST_P(ArithmeticSuite, SumTest, sum_test_data) {
  auto &&[inputs, expected] = _data.current();
  int actual = inputs[0] + inputs[1];
  EXPECT_EQ_FOREACH(expected, actual);
  return ZState::z_success;
}
ZTestDataManager<tuple<float, int>, float> sum_test_data2 = {
    {{1.2, 2}, 3.2}, {{-1.0, 1}, 0.0}, {{10.1, 20}, 30.2}};
ZTEST_P(ArithmeticSuite, SumTestfordiff, sum_test_data2) {
```

```
  auto &&[inputs, expected] = _data.current();
  float actual = std::get<0>(inputs) + std::get<1>(inputs);
  EXPECT_EQ_FOREACH(expected, actual);
  return ZState::z_success;
}
ZTEST_P_CSV(MathTests, AdditionTests, "data.csv") {
  auto inputs = getInput();
  auto expected = getOutput();
  double actual = std::get<double>(inputs[0]) + std::get<double>(inputs[1]);
  EXPECT_EQ(actual, std::get<double>(expected));
  return ZState::z_success;
}
```

```
[2025-06-06 11:54:40] [DEBUG] Checking cache for: data.csv
[2025-06-06 11:54:40] [INFO] Loading new file: data.csv
[2025-06-06 11:54:40] [DEBUG] Initializing CSV data manager for: data.csv
[2025-06-06 11:54:40] [INFO] Loaded 4 rows from CSV file
[2025-06-06 11:54:40] [DEBUG] Processed 4 test cases
[2025-06-06 11:54:40] [DEBUG] Cached file: data.csv | Size: 4
[2025-06-06 11:54:40] [INFO] Set LRU cache max size to: 1
[2025-06-06 11:54:42] [INFO] [Safe] Starting parallel execution of 0 safe tests using 8 workers
[2025-06-06 11:54:42] [INFO] [Safe] Parallel execution completed
[2025-06-06 11:54:42] [INFO] [Parameterized] Test succeeded: ArithmeticSuite.SumTest (0.006080ms)
[2025-06-06 11:54:42] [INFO] [Parameterized] Test succeeded: ArithmeticSuite.SumTestfordiff (0.001709ms)
[2025-06-06 11:54:42] [INFO] [Parameterized] Test succeeded: MathTests.AdditionTests (0.018732ms)
```

图 20: 数据驱动 function test

5.1.5 benchmark 测试

测试 benchmark 类型测试的定义，以及测试时间分布的可视化，以及在 GUI 上对于 CPU 和内存使用率的监视。

```
ZBENCHMARK(Vector, PushBack) {
  std::vector<int> v;
  for (int i = 0; i < 10000; ++i)
    v.push_back(i);
  return ZState::z_success;
}
ZBENCHMARK(Matrix, PushBack, 20000) {
  std::vector<int> v;
  for (int i = 0; i < 1000; ++i)
    v.push_back(random());
  return ZState::z_success;
}
```

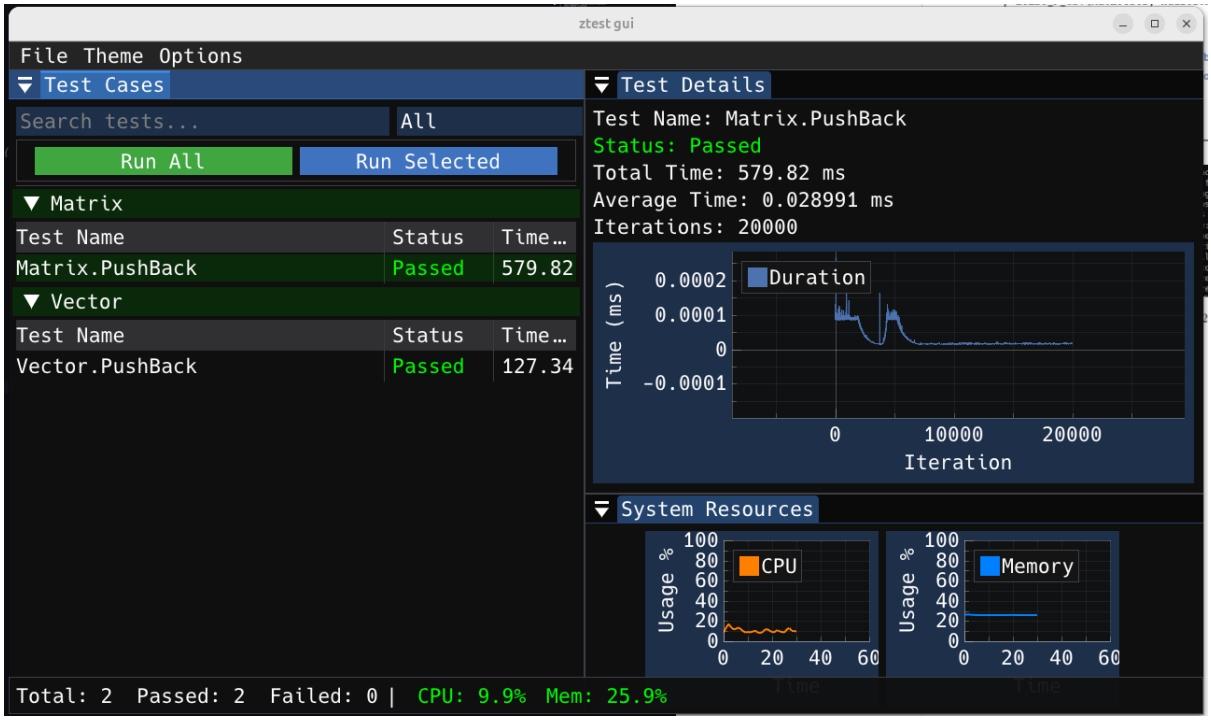


图 21: 数据驱动 function test

-

5.1.6 报告生成测试

测试了 html,json,xml 格式报告的生成。

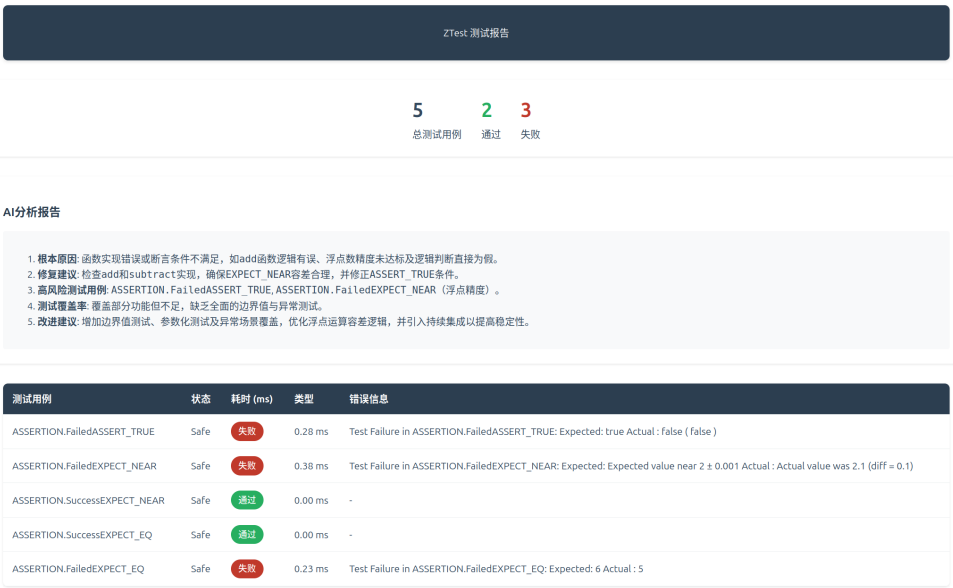


图 22: HTML 测试报告

```
{
  "summary": {
    "total": 15,
    "passed": 12,
    "failed": 3
  },
  "tests": [
    {
      "name": "MathTests.AdditionTests",
      "status": "Passed",
      "duration": 0.02,
      "error": ""
    },
```


图 23: 部分 json 报告


```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites>
  <testsuite name="ASSERTION" tests="5" failures="3" errors="0" time="0.000887069">
    <testcase name="ASSERTION.FailedASSERT_TRUE" classname="ASSERTION" time="0.000">
      <failure message="Test Failure in ASSERTION.FailedASSERT_TRUE:
Expected: true
Actual  : false ( false )"/></testcase>
    <testcase name="ASSERTION.FailedEXPECT_NEAR" classname="ASSERTION" time="0.000">
      <failure message="Test Failure in ASSERTION.FailedEXPECT_NEAR:
Expected: Expected value near 2 ± 0.001
Actual  : Actual value was 2.1 (diff = 0.1)"/></testcase>
    <testcase name="ASSERTION.SuccessEXPECT_NEAR" classname="ASSERTION" time="0.000"></testcase>
    <testcase name="ASSERTION.SuccessEXPECT_EQ" classname="ASSERTION" time="0.000"></testcase>
    <testcase name="ASSERTION.FailedEXPECT_EQ" classname="ASSERTION" time="0.000">
      <failure message="Test Failure in ASSERTION.FailedEXPECT_EQ:
Expected: 6
Actual  : 5"/></testcase>
    </testsuite>
  </testsuites>
```


图 24: xml(JUnit 格式) 报告

5.1.7 GUI 展示

测试了 GUI 的展示，主题的切换，AI 助手展示等功能。

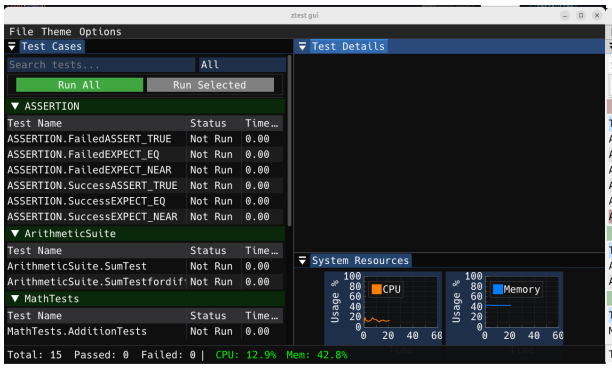


图 25: Dark theme GUI 展示

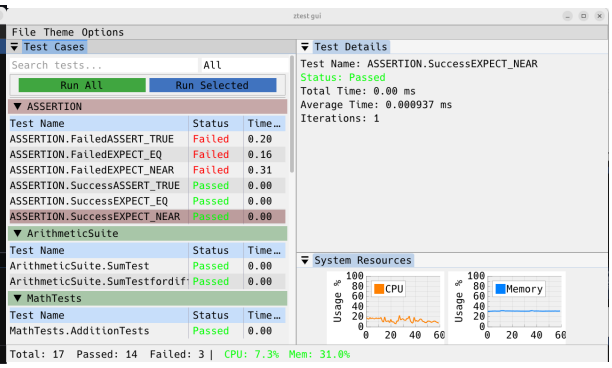


图 26: Light Theme GUI 展示

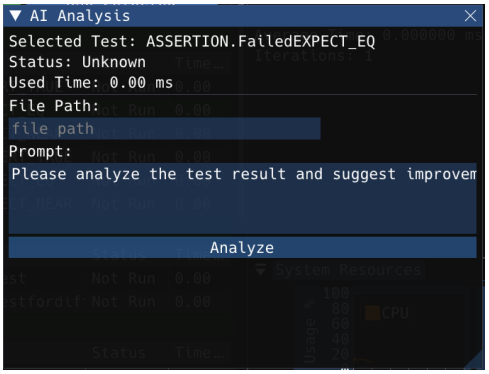
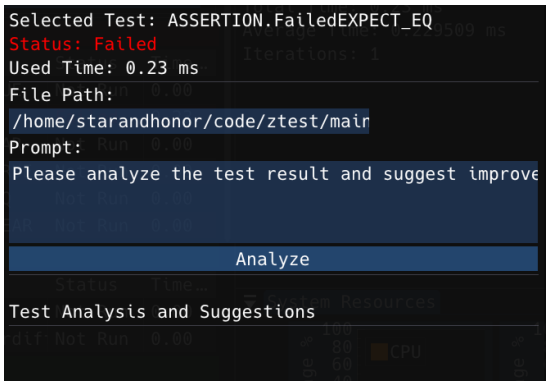


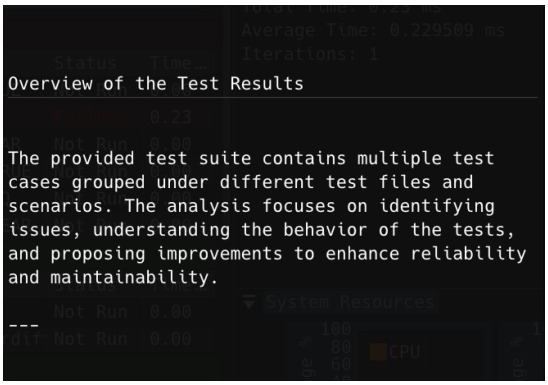
图 27: AI 诊断助手展示

5.1.8 AI 诊断测试

生成报告的时候，调用 AI(qwen turbo) 进行诊断，并写入到 HTML 报告中。

在 GUI 界面，调用 AI(qwen turbo) 进行诊断对于单个测试用例进行诊断并显示，允许用户输入自定义的提示词和测试所在的文件路径。





6 个人总结

7 参考文献

Reference

[1] Google Test. *Google Test Documentation*. [Online]. Available: https://github.com/google/googletest

[2] JUnit. *JUnit Documentation*. [Online]. Available: https://junit.org/junit5/

[3] Catch2. *Catch2 Documentation*. [Online]. Available: https://github.com/catchorg/Catch2

[4] Google Test Parallel. *Google Test Parallel Documentation*. [Online]. Available: https://github.com/google/gtest-parallel

[5] Pytest. *Pytest Documentation*. [Online]. Available: https://docs.pytest.org/en/latest/

[6] MiniUnit. *MiniUnit Documentation*. [Online]. Available: https://github.com/urin/miniunit