个人资料

breaksoftware

访问：673860次
积分：8696
等级：BLOG 6
排名：第2191名

原创：201篇
转载：1篇
译文：0篇
评论：441条

Google Test(GTest)使用方法和源码解析——Listener技术分析和应用

2016-04-07 23:55 1016人阅读 评论(0) 收藏 举报

分类: GTest使用方法和源码解析（10）

目录(?) [+]

在《Google Test(GTest)使用方法和源码解析——结果统计机制分析》文中，我么分析了GTes进行统计的。本文我们将解析其结果输出所使用到的Listener机制。(转载请指明出于breaksoftware

解析

源码中，我们经常要和UnitTest类打交道。它提供了一个单例方法返回自己的一个对象，然后各这个单例的方法。所以说它是GTest框架中非常重要的衔接环。而在其内部，实际工作的却是一个UnitTestImpl对象

```cpp
internal::UnitTestImpl* impl_;
```

该指针在UnitTest构造函数中被新建出来

```cpp
UnitTest::UnitTest() {
  impl_ = new internal::UnitTestImpl(this);
}
```

之后，我们调用UnitTest单例的方法，很多都是直接调用该对象的方法。其构造函数是这么写的

```cpp
UnitTestImpl::UnitTestImpl(UnitTest* parent)
    : parent_(parent),
      GTEST_DISABLE_MSC_WARNINGS_PUSH_(4355 /* using this in initializer */)
      default_global_test_part_result_reporter_(this),
      default_per_thread_test_part_result_reporter_(this),
      GTEST_DISABLE_MSC_WARNINGS_POP_()
      global_test_part_result_repoter_(
          &default_global_test_part_result_reporter_),
      per_thread_test_part_result_reporter_(
          &default_per_thread_test_part_result_reporter_),
      .......
      catch_exceptions_(false) {
  listeners()->SetDefaultResultPrinter(new PrettyUnitTestResultPrinter);
}
```

本文要讲解的内容将和上面的代码有很大的关系。

在GTest测试框架中，它提出了一个Listener的概念，以供开发者监听执行过程。GTest框架就是使用Listener机制实现了结果输出。我们看下listener基类的设计

```cpp
class TestEventListener {
 public:
  virtual ~TestEventListener() {}

  // Fired before any test activity starts.
  virtual void OnTestProgramStart(const UnitTest& unit_test) = 0;

  // Fired before each iteration of tests starts.  There may be more than
  // one iteration if GTEST_FLAG(repeat) is set. iteration is the iteration
  // index, starting from 0.
  virtual void OnTestIterationStart(const UnitTest& unit_test,
```

WMI技术介绍和应用 (24)
Apache服务搭建和插件实现 (7)
网络编程模型的分析、实现和对比 (6)
GTest使用方法和源码解析 (11)
PE文件结构和相关应用 (11)
windows安全 (9)
网络通信 (5)
沙箱 (7)
内嵌及定制Lua引擎技术 (3)
IE控件及应用 (7)
反汇编 (15)
开源项目 (16)
C++ (15)
界面库 (3)
python (11)
疑难杂症 (24)
PHP (8)
Redis (8)
IT项目研发过程中的利器 (4)
libev源码解析 (6)

文章存档

2017年08月 (7)
2017年07月 (4)
2017年05月 (9)
2017年02月 (1)
2016年12月 (10)
展开

阅读排行

使用WinHttp接口实现HT (35595)
WMI技术介绍和应用—— (18359)
如何定制一款12306抢票 (13984)
一种准标准CSV格式的介 (12486)
一种精确从文本中提取UI (12203)
实现HTTP协议Get、Post (11999)
分析两种Dump(崩溃日志 (11576)
一种解决运行程序报"应用 (11171)
实现HTTP协议Get、Post (11158)
反汇编算法介绍和应用— (10676)

评论排行

使用WinHttp接口实现HT (33)
使用VC实现一个"智能"自 (27)
WMI技术介绍和应用—— (23)
WMI技术介绍和应用—— (20)
实现HTTP协议Get、Post (20)
如何定制一款12306抢票 (17)
在windows程序中嵌入Lu (15)
一个分析"文件夹"选择框 (13)
反汇编算法介绍和应用— (12)
使用VC内嵌Python实现自 (10)

推荐文章

* CSDN日报20170817——《如果不从事编程，我可以做什么？》

```
                                    int iteration) = 0;

  // Fired before environment set-up for each iteration of tests starts.
  virtual void OnEnvironmentsSetUpStart(const UnitTest& unit_test) = 0;

  // Fired after environment set-up for each iteration of tests ends.
  virtual void OnEnvironmentsSetUpEnd(const UnitTest& unit_test) = 0;

  // Fired before the test case starts.
  virtual void OnTestCaseStart(const TestCase& test_case) = 0;

  // Fired before the test starts.
  virtual void OnTestStart(const TestInfo& test_info) = 0;

  // Fired after a failed assertion or a SUCCEED() invocation.
  virtual void OnTestPartResult(const TestPartResult& test_part_result) = 0;

  // Fired after the test ends.
  virtual void OnTestEnd(const TestInfo& test_info) = 0;

  // Fired after the test case ends.
  virtual void OnTestCaseEnd(const TestCase& test_case) = 0;

  // Fired before environment tear-down for each iteration of tests starts.
  virtual void OnEnvironmentsTearDownStart(const UnitTest& unit_test) = 0;

  // Fired after environment tear-down for each iteration of tests ends.
  virtual void OnEnvironmentsTearDownEnd(const UnitTest& unit_test) = 0;

  // Fired after each iteration of tests finishes.
  virtual void OnTestIterationEnd(const UnitTest& unit_test,
                                  int iteration) = 0;

  // Fired after all test activities have ended.
  virtual void OnTestProgramEnd(const UnitTest& unit_test) = 0;
};
```

它暴露了很多接口，每个都对应于执行过程的一个状态，比如OnTestCaseStart，字面意思就是测试用例执行开始处（要执行自定义逻辑），此处比较适合输出测试用例的基本信息；再比如OnTestCaseEnd，是测试用例执行结束处（要执行自定义逻辑），此处比较适合输出测试用例的执行结果。

在UnitTestImpl构造函数中有个listeners()函数，其返回了UnitTestImpl类的TestEventListeners成员变量指针。从名字上看可以看出它是一个Listener的集合，因为用户可以新增自定义的Listener，所以要将其设计为一个集合。但是实际上它只是集合的封装

```
[cpp]

class GTEST_API_ TestEventListeners {
 private:
 .........
  // The actual list of listeners.
  internal::TestEventRepeater* repeater_;
  // Listener responsible for the standard result output.
  TestEventListener* default_result_printer_;
  // Listener responsible for the creation of the XML output file.
  TestEventListener* default_xml_generator_;
}
```

TestEventRepeater才是Listener的集合，同时它也是继承于TestEventListener接口类。

```
[cpp]

class TestEventRepeater : public TestEventListener {
 public:
  TestEventRepeater() : forwarding_enabled_(true) {}
  virtual ~TestEventRepeater();
  void Append(TestEventListener *listener);
  TestEventListener* Release(TestEventListener* listener);

  // Controls whether events will be forwarded to listeners_. Set to false
  // in death test child processes.
  bool forwarding_enabled() const { return forwarding_enabled_; }
  void set_forwarding_enabled(bool enable) { forwarding_enabled_ = enable; }

  virtual void OnTestProgramStart(const UnitTest& unit_test);
  virtual void OnTestIterationStart(const UnitTest& unit_test, int iteration);
  virtual void OnEnvironmentsSetUpStart(const UnitTest& unit_test);
  virtual void OnEnvironmentsSetUpEnd(const UnitTest& unit_test);
```

* Android自定义EditText：你需要一款简单实用的SuperEditText(一键删除&自定义样式)

* 从JDK源码角度看Integer

* 微信小程序——智能小秘"通知之"源码分享（语义理解基于olami）

* 多线程中断机制

* 做自由职业者是怎样的体验

最新**评论**

使用WinHttp接口实现HTTP协议(
breaksoftware: @qq_34534425:你过谦了。多总结、多练习、多借鉴就好了。

使用WinHttp接口实现HTTP协议(
qq_34534425: 代码真心nb，感觉自己写的就是渣渣

朴素、Select、Poll和Epoll网络编程
zhangcunli8499: @Breaksoftware:多谢

朴素、Select、Poll和Epoll网络编程
breaksoftware: @zhangcunli8499:这篇 http://blog.csdn.net/breaksoftwa...

朴素、Select、Poll和Epoll网络编程
zhangcunli8499: 哥们，能传一下完整的代码吗？

C++拾趣——类构造函数的隐式转
breaksoftware: @wuchalilun:多谢鼓励，其实我就想写出点不一样的地方，哈哈。

C++拾趣——类构造函数的隐式转
Ray_Chang_988: 其他相关的explicit的介绍文章也看了，基本上explicit的作用也都解释清楚了，但是它们都没...

Redis源码解析——字典结构
breaksoftware: @u011548018:多谢鼓励

Redis源码解析——字典结构
生无可恋只能打怪升级: 就冲这图也得点1024个赞

WMI技术介绍和应用——查询系统
breaksoftware: @hobbyonline:我认为这种属性的信息不准确是很正常的，因为它的正确与否不会影响到系统在不同...

```
17.   virtual void OnTestCaseStart(const TestCase& test_case);
18.   virtual void OnTestStart(const TestInfo& test_info);
19.   virtual void OnTestPartResult(const TestPartResult& result);
20.   virtual void OnTestEnd(const TestInfo& test_info);
21.   virtual void OnTestCaseEnd(const TestCase& test_case);
22.   virtual void OnEnvironmentsTearDownStart(const UnitTest& unit_test);
23.   virtual void OnEnvironmentsTearDownEnd(const UnitTest& unit_test);
24.   virtual void OnTestIterationEnd(const UnitTest& unit_test, int iteration);
25.   virtual void OnTestProgramEnd(const UnitTest& unit_test);
26.
27.  private:
28.   // Controls whether events will be forwarded to listeners_. Set to false
29.   // in death test child processes.
30.   bool forwarding_enabled_;
31.   // The list of listeners that receive events.
32.   std::vector<TestEventListener*> listeners_;
33.
34.   GTEST_DISALLOW_COPY_AND_ASSIGN_(TestEventRepeater);
35. };
```

这个类的设计也非常有意思，它既在成员变量listeners_中保存了一系列用户自定义的监听者（TestEventListener*对象指针），又让自身继承于TestEventListener，那就是说它自己也是一个Liste大内总管，自己是个太监，手下也是太监。小太监要干的活，大内总管也要能去做（继承于TestEventl内总管不用自己亲手去做，而是调度小太监去做。这样的功能传递是通过类似下面代码的调用去实现

[cpp]

```
01. // Since most methods are very similar, use macros to reduce boilerplate.
02. // This defines a member that forwards the call to all listeners.
03. #define GTEST_REPEATER_METHOD_(Name, Type) \
04. void TestEventRepeater::Name(const Type& parameter) { \
05.   if (forwarding_enabled_) { \
06.     for (size_t i = 0; i < listeners_.size(); i++) { \
07.       listeners_[i]->Name(parameter); \
08.     } \
09.   } \
10. }
11. // This defines a member that forwards the call to all listeners in reverse
12. // order.
13. #define GTEST_REVERSE_REPEATER_METHOD_(Name, Type) \
14. void TestEventRepeater::Name(const Type& parameter) { \
15.   if (forwarding_enabled_) { \
16.     for (int i = static_cast<int>(listeners_.size()) - 1; i >= 0; i--) { \
17.       listeners_[i]->Name(parameter); \
18.     } \
19.   } \
20. }
```

TestEventRepeater从TestEventListener继承来的方法便如此定义

[cpp]

```
01. GTEST_REPEATER_METHOD_(OnTestProgramStart, UnitTest)
02. GTEST_REPEATER_METHOD_(OnEnvironmentsSetUpStart, UnitTest)
03. GTEST_REPEATER_METHOD_(OnTestCaseStart, TestCase)
04. GTEST_REPEATER_METHOD_(OnTestStart, TestInfo)
05. GTEST_REPEATER_METHOD_(OnTestPartResult, TestPartResult)
06. GTEST_REPEATER_METHOD_(OnEnvironmentsTearDownStart, UnitTest)
07. GTEST_REVERSE_REPEATER_METHOD_(OnEnvironmentsSetUpEnd, UnitTest)
08. GTEST_REVERSE_REPEATER_METHOD_(OnEnvironmentsTearDownEnd, UnitTest)
09. GTEST_REVERSE_REPEATER_METHOD_(OnTestEnd, TestInfo)
10. GTEST_REVERSE_REPEATER_METHOD_(OnTestCaseEnd, TestCase)
11. GTEST_REVERSE_REPEATER_METHOD_(OnTestProgramEnd, UnitTest)
```

可以见得这个大内总管，对于上面的指令只是做了一个简单的判断就抛给下面每个小太监去做了。

说个题外话，个人觉得TestEventRepeater中repeater的翻译，不要叫做"重复者"，叫"中继者"比较好。

我们再回顾下监听者的传递过程

[cpp]

```
01. listeners()->SetDefaultResultPrinter(new PrettyUnitTestResultPrinter);
```

[cpp]

```
01. void TestEventListeners::SetDefaultResultPrinter(TestEventListener* listener) {
02.   if (default_result_printer_ != listener) {
03.     // It is an error to pass this method a listener that is already in the
```

```
    // list.
    delete Release(default_result_printer_);
    default_result_printer_ = listener;
    if (listener != NULL)
      Append(listener);
  }
}
```

[cpp]

```
void TestEventListeners::Append(TestEventListener* listener) {
  repeater_->Append(listener);
}
```

SetDefaultResultPrinter方法看着是传递一个“结果打印者”，但是它实际要接受一个Listener。这个命名虽然很直观，但是也让阅读代码的人一下子转不过弯来：Listener和Printer是一回事啊！

[cpp]

```
class PrettyUnitTestResultPrinter : public TestEventListener {
```

PrettyUnitTestResultPrinter各个事件处理函数我就不罗列了，它们就是一些结果输出。有兴趣查看。

然后我们再来看框架中是如何“触发”这些事件的。

首先是UnitTestImpl::RunAllTests()函数，它处理了几个比较大级别的事件，比如程序启动和结

[cpp]

```
bool UnitTestImpl::RunAllTests() {
  ......
  TestEventListener* repeater = listeners()->repeater();
  ......
  repeater->OnTestProgramStart(*parent_);
  ......
  for (int i = 0; forever || i != repeat; i++) {
    ......
    repeater->OnTestIterationStart(*parent_, i);
    ......
    if (has_tests_to_run) {
      // Sets up all environments beforehand.
      repeater->OnEnvironmentsSetUpStart(*parent_);
      ForEach(environments_, SetUpEnvironment);
      repeater->OnEnvironmentsSetUpEnd(*parent_);

      // Runs the tests only if there was no fatal failure during global
      // set-up.
      if (!Test::HasFatalFailure()) {
        for (int test_index = 0; test_index < total_test_case_count();
             test_index++) {
          GetMutableTestCase(test_index)->Run();
        }
      }

      // Tears down all environments in reverse order afterwards.
      repeater->OnEnvironmentsTearDownStart(*parent_);
      std::for_each(environments_.rbegin(), environments_.rend(),
                    TearDownEnvironment);
      repeater->OnEnvironmentsTearDownEnd(*parent_);
    }
    ......
    repeater->OnTestIterationEnd(*parent_, i);
    ......
  }
  ......
  repeater->OnTestProgramEnd(*parent_);
  ......
}
```

然后GetMutableTestCase(test_index)->Run();进入每个测试用例的运行，它只处理了OnTestCaseStart和OnTestCaseEnd两个事件

[cpp]

```
void TestCase::Run() {
  ......
  internal::UnitTestImpl* const impl = internal::GetUnitTestImpl();
  ......
```

```
  TestEventListener* repeater = UnitTest::GetInstance()->listeners().repeater();
  ......
  repeater->OnTestCaseStart(*this);
  ......

  const internal::TimeInMillis start = internal::GetTimeInMillis();
  for (int i = 0; i < total_test_count(); i++) {
    GetMutableTestInfo(i)->Run();
  }
  ......
  repeater->OnTestCaseEnd(*this);
  ......
}
```

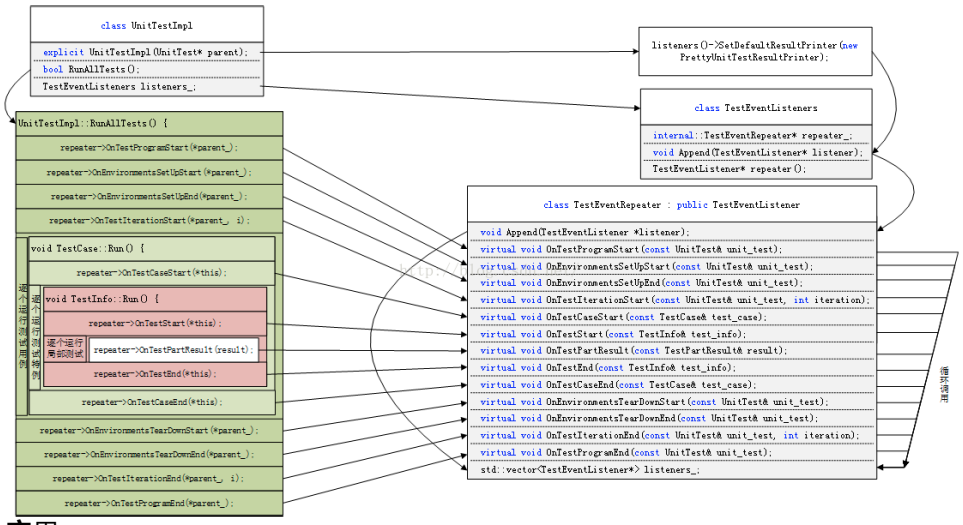
GetMutableTestInfo(i)->Run();方法进入测试特例运行，它只处理了OnTestStart和OnTestEnd

```
[cpp]

void TestInfo::Run() {
  ......
  // Tells UnitTest where to store test result.
  internal::UnitTestImpl* const impl = internal::GetUnitTestImpl();
  ......
  TestEventListener* repeater = UnitTest::GetInstance()->listeners().repeater
  // Notifies the unit test event listeners that a test is about to start.
  repeater->OnTestStart(*this);
  ......
  // Runs the test only if the test object was created and its
  // constructor didn't generate a fatal failure.
  if ((test != NULL) && !Test::HasFatalFailure()) {
    // This doesn't throw as all user code that can throw are wrapped into
    // exception handling code.
    test->Run();
  }
  ......
  // Notifies the unit test event listener that a test has just finished.
  repeater->OnTestEnd(*this);
  ......
}
```

test->Run();进入了我们自定义的测试实体，其内部通过层层传导UnitTestImpl的global_test_part_result_repoter_的函数中

```
[cpp]

void DefaultGlobalTestPartResultReporter::ReportTestPartResult(
    const TestPartResult& result) {
  unit_test_->current_test_result()->AddTestPartResult(result);
  unit_test_->listeners()->repeater()->OnTestPartResult(result);
}
```

如此，我们便将listener的运行机制给讲完了。顺便我们也分析了GTest默认结果输出的实现。



应用

要使用Listener技术，我们需要实现一个继承于 testing::TestEventListener 或testing::EmptyTestEventListener的类。如果继承于testing::TestEventListener，则需要我们实现所有纯虚方法；而如果继承于

testing::EmptyTestEventListener，则我们只要关注于部分我们关心的函数实现即可——因为它已经把所有纯虚方法实现为空方法了。

```cpp
class MinimalistPrinter : public ::testing::EmptyTestEventListener {
    // Called before a test starts.
    virtual void OnTestStart(const ::testing::TestInfo& test_info) {
        printf("*** Test %s.%s starting.\n",
            test_info.test_case_name(), test_info.name());
    }

    // Called after a failed assertion or a SUCCEED() invocation.
    virtual void OnTestPartResult(
        const ::testing::TestPartResult& test_part_result) {
            printf("%s in %s:%d\n%s\n",
                test_part_result.failed() ? "*** Failure" : "Success",
                test_part_result.file_name(),
                test_part_result.line_number(),
                test_part_result.summary());
    }

    // Called after a test ends.
    virtual void OnTestEnd(const ::testing::TestInfo& test_info) {
        printf("*** Test %s.%s ending.\n",
            test_info.test_case_name(), test_info.name());
    }
};
```

然后我们就需要在main函数中将该Listener的对象加入到框架中。此处有个地方需要注意下，由于Listener是个列表，那就意味着一堆Listener将会被执行，其中包括GTest默认的Listener——之前结果输出的实现[illegible]！想让我们自定义的Listener执行，则要先将默认Listener去掉（下面代码第3行）。

```cpp
::testing::TestEventListeners& listeners =
    ::testing::UnitTest::GetInstance()->listeners();
delete listeners.Release(listeners.default_result_printer());
listeners.Append(new MinimalistPrinter);
```

这儿有个一个要注意的是：除了OnTestPartResult()之外的函数，都可以使用GTest判断类宏进行数据判断。唯独OnTestPartResult()里不可以，否则会造成OnTestPartResult()被递归调用。

上一篇 Google Test(GTest)使用方法和源码解析——结果统计机制分析
下一篇 Google Test(GTest)使用方法和源码解析——断言的使用方法和解析

相关文章推荐

- Google Test(GTest)使用方法和源码解析——断言…
- 【直播】机器学习之凸优化--马博士
- Google Test(GTest)使用方法和源码解析——死亡…
- 【直播】计算机视觉原理及实战--屈教授
- Google Test(GTest)使用方法和源码解析——预处…
- 机器学习&数据挖掘7周实训--韦玮
- Google Test(GTest)使用方法和源码解析——结果…
- 机器学习之数学基础系列--AI100
- Google Test(GTest)使用方法和源码解析——自定…
- 【套餐】2017软考系统集成项目管理工程师顺利通…
- Google Test(GTest)使用方法和源码解析——私有…
- 【课程】深入探究Linux/VxWorks的设备树--宋宝华
- Google Mock(Gmock)简单使用和源码分析——源…
- 用google mock模拟C++对象
- python3编写简易统计服务器
- google test简介

查看评论

暂无评论

发表评论

用 户 名： GreatProgramer

评论内容：

提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场