

方亮的专栏

目录视图

摘要视图

RSS 订阅

个人资料



breaksoftware

访问: 673874次

积分: 8696

等级:

BLDG

5

排名: 第2191名

原创: 201篇

转载: 1篇

译文: 0篇

评论: 441条

文章搜索

Q

博客专栏

利器

IT项目研发过程中的利器

文章:4篇

阅读:443

Libev

libev源码解析

文章:6篇

阅读:676

PE

PE文件和COFF文件格式分析

文章:11篇

阅读:28979

动态链接库

DLLMain中不当操作导致死锁问题的分析

文章:9篇

阅读:36501

WMI

WMI技术介绍和应用

文章:24篇

阅读:110417

Google Test

GTest源码解析

文章:11篇

阅读:25240

文章分类

DLLMain中的做与不做 (9)

赠书 | 异步2周年,技术图书免费送 每周荐书:渗透测试,K8s、架构(评论送书) 项目管理+代码托管+文档协作,开发更流畅

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

2016-04-08 00:04

2186人阅读

评论(0)

收藏

举报

分类: GTest使用方法和源码解析 (10)

版权声明:本文为博主原创文章,未经博主允许不得转载。

目录(?)

死亡测试是为了判断一段逻辑是否会导致进程退出而设计的。这种场景并不常见,但是GTest提供了这个功能。我们先看下其应用实例。[\(转载请指明出于breaksoftware的csdn博客\)](#)

死亡测试技术应用

我们可以使用TEST声明并注册一个简单的测试特例。其实现内部才是死亡测试相关代码运行。我们提供了如下的宏用于组织测试逻辑

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_DEATH(statement, regex);	EXPECT_DEATH(statement, regex);	statement crashes with the given error code or predicate
ASSERT_DEATH_IF_SUPPORTED(statement, regex);	EXPECT_DEATH_IF_SUPPORTED(statement, regex);	if death tests are supported, verifies that statement crashes with the given error code or predicate; otherwise verifies nothing
ASSERT_EXIT(statement, predicate, regex);	EXPECT_EXIT(statement, predicate, regex);	statement exits with the given error code and the exit code matches the given predicate

宏中的statement是测试逻辑的表达式,它可以是个函数,可以是对象的方法,也可以是几个表达式的组合,比如

```
[cpp]
01. EXPECT_DEATH({ int n = 4; n = 5; }, "");
```

regex是一个正则表达式,它用于匹配stderr输出的内容。如果匹配上了,则测试成功,否则测试失败。比如

```
[cpp]
01. void Foo() {
02.     std::cerr<<"Failed Foo";
03.     _exit(0);
04. }
05. EXPECT_DEATH(Foo(), ". *Foo");
06. EXPECT_DEATH(Foo(), ". *FAAA");
```

第5行的局部测试匹配上了测试预期,而第6行没有。

注意下正则表达式这个功能只支持Linux系统, windows上不支持,所以windows上我们对此参数传空串。我们看个完整的例子

```
[cpp]
01. void Foo() {
02.     std::cerr<<"Fail Foo";
03.     _exit(0);
04. }
```

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“应月 (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实现的 (10)

推荐文章

* CSDN日报20170817——《如果不从事编程，我可以做什么？》

```
05.
06.     TEST(MyDeathTest, Foo) {
07.         EXPECT_EXIT(Foo(), ::testing::ExitedWithCode(0), ".*Foo");
08.     }
```

注意下我们测试用例名——MyDeathTest, GTest强烈建议测试用例名以DeathTest结尾。这是为了让死亡测试在所有其他测试之前运行。

死亡测试技术分析

死亡测试非常依赖于系统的实现。本文并不打算把每个系统都覆盖到,我将以windows系统上的实现详细讲解其过程。在Linux上实现的思路基本和windows上相同,只是在一些系统实现上存在差异导致GTest具有不同的属性。

先概括的讲一下windows上实现的过程

1. 测试实体中准备启动新的进程,进程路径就是本进程可执行文件路径
2. 子进程传入了标准输入输出句柄
3. 启动子进程时传入类型筛选,即指定执行该测试用例
4. 监听子进程的输出
5. 判断子进程退出模式

子进程的执行过程是:

1. 执行父进程指定的测试特例
2. 运行死亡测试宏中的表达式
3. 如果没有crash,则根据情况选择退出模式

我们来看下EXPECT_DEATH的实现,其最终将调用到GTEST_DEATH_TEST_宏中

```
[cpp]
01. # define GTEST_DEATH_TEST_(statement, predicate, regex, fail) \
02.     GTEST_AMBIGUOUS_ELSE_BLOCKER_ \
03.     if (::testing::internal::AlwaysTrue()) { \
04.         const ::testing::internal::RE& gtest_regex = (regex); \
05.         ::testing::internal::DeathTest* gtest_dt; \
06.         if (!::testing::internal::DeathTest::Create(#statement, >est_regex, \
07.             __FILE__, __LINE__, >est_dt)) { \
08.             goto GTEST_CONCAT_TOKEN_(gtest_label_, __LINE__); \
09.         } \
10.         if (gtest_dt != NULL) { \
11.             ::testing::internal::scoped_ptr< ::testing::internal::DeathTest> \
12.                 gtest_dt_ptr(gtest_dt); \
13.             switch (gtest_dt->AssumeRole()) { \
14.                 case ::testing::internal::DeathTest::OVERSEE_TEST: \
15.                     if (!gtest_dt->Passed(predicate(gtest_dt->Wait())))) { \
16.                         goto GTEST_CONCAT_TOKEN_(gtest_label_, __LINE__); \
17.                     } \
18.                     break; \
19.                 case ::testing::internal::DeathTest::EXECUTE_TEST: { \
20.                     ::testing::internal::DeathTest::ReturnSentinel \
21.                         gtest_sentinel(gtest_dt); \
22.                     GTEST_EXECUTE_DEATH_TEST_STATEMENT_(statement, gtest_dt); \
23.                     gtest_dt->Abort(::testing::internal::DeathTest::TEST_DID_NOT_DIE); \
24.                     break; \
25.                 } \
26.                 default: \
27.                     break; \
28.             } \
29.         } \
30.     } else \
31.         GTEST_CONCAT_TOKEN_(gtest_label_, __LINE__): \
32.         fail(::testing::internal::DeathTest::LastMessage())
```

第5行我们声明了一个DeathTest*指针,这个类暴露了一个静态方法用于创建对象。可以说它是一个接口类,我们看下它重要的部分定义

```
[cpp]
01. enum TestRole { OVERSEE_TEST, EXECUTE_TEST };
02.
03. // An enumeration of the three reasons that a test might be aborted.
04. enum AbortReason {
05.     TEST_ENCOUNTERED_RETURN_STATEMENT,
06.     TEST_THREW_EXCEPTION,
07.     TEST_DID_NOT_DIE
08. }
```

* 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\$ 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\$ 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 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...

```
08. };
09.
10. // Assumes one of the above roles.
11. virtual TestRole AssumeRole() = 0;
12.
13. // Waits for the death test to finish and returns its status.
14. virtual int Wait() = 0;
15.
16. // Returns true if the death test passed; that is, the test process
17. // exited during the test, its exit status matches a user-supplied
18. // predicate, and its stderr output matches a user-supplied regular
19. // expression.
20. // The user-supplied predicate may be a macro expression rather
21. // than a function pointer or functor, or else Wait and Passed could
22. // be combined.
23. virtual bool Passed(bool exit_status_ok) = 0;
24.
25. // Signals that the death test did not die as expected.
26. virtual void Abort(AbortReason reason) = 0;
```

TestRole就是角色, 我们父进程角色是OVERSEE_TEST, 子进程的角色是EXECUTE_TEST。因为父子进程都将进入这个测试特例逻辑, 所以要通过角色标记来区分执行逻辑。AbortReason枚举中类型表达了测

试。AssumeRole是主要是父进程后动子进程的逻辑。Wait是父进程等待子进程执行完毕, 并尝试调出。

DeathTest::Create方法最终会进入DefaultDeathTestFactory::Create方法

```
[cpp]
01. bool DefaultDeathTestFactory::Create(const char* statement, const RE* regex,
02.                                     const char* file, int line,
03.                                     DeathTest** test) {
04.     UnitTestImpl* const impl = GetUnitTestImpl();
05.     const InternalRunDeathTestFlag* const flag =
06.         impl->internal_run_death_test_flag();
07.     const int death_test_index = impl->current_test_info()
08.         ->increment_death_test_count();
09.
10.     if (flag != NULL) {
11.         if (death_test_index > flag->index()) {
12.             DeathTest::set_last_death_test_message(
13.                 "Death test count (" + StreamableToString(death_test_index)
14.                 + ") somehow exceeded expected maximum ("
15.                 + StreamableToString(flag->index()) + ")");
16.             return false;
17.         }
18.
19.         if (!(flag->file() == file && flag->line() == line &&
20.             flag->index() == death_test_index)) {
21.             *test = NULL;
22.             return true;
23.         }
24.     }
```

此处通过获取flag变量, 得知当前运行的是子进程还是父进程。如果flag不是NULL, 则是子进程, 它主要做些输出的工作; 如果是父进程, 则进入下面代码

```
[cpp]
01. # if GTEST_OS_WINDOWS
02.
03.     if (GTEST_FLAG(death_test_style) == "threadsafe" ||
04.         GTEST_FLAG(death_test_style) == "fast") {
05.         *test = new WindowsDeathTest(statement, regex, file, line);
06.     }
07.
08. # else
09.
10.     if (GTEST_FLAG(death_test_style) == "threadsafe") {
11.         *test = new ExecDeathTest(statement, regex, file, line);
12.     } else if (GTEST_FLAG(death_test_style) == "fast") {
13.         *test = new NoExecDeathTest(statement, regex);
14.     }
15.
16. # endif // GTEST_OS_WINDOWS
```

可见Windows上死亡测试最终将由WindowsDeathTest代理, 而linux系统根据传入参数不同而选择不同的类。

它们都是DeathTest的派生类。为什么linux系统上支持参数选择,这要从系统暴露出来的接口和系统实现来说。windows系统上进程创建只要调用CreateProcess之类的函数就可以了,这个函数调用后,子进程就创建出来了。而linux系统上则要调用fork或者clone之类,这两中函数执行机制也不太相同。fork是标准的子进程和父进程分离执行,所以threadsafe对应的ExecDeathTest类在底层调用的是fork,从而可以保证是安全的。但是clone用于创建轻量级进程,即创建的子进程与父进程共用线性地址空间,只是它们的堆栈不同,这样不用执行父子进程分离,执行当然会快些,所以这种方式对应的是fast——NoExecDeathTest。

我们看下WindowsDeathTest::AssumeRole()的实现

```
[cpp]
01. // The AssumeRole process for a Windows death test. It creates a child
02. // process with the same executable as the current process to run the
03. // death test. The child process is given the --gtest_filter and
04. // --gtest_internal_run_death_test flags such that it knows to run the
05. // current death test only.
06. DeathTest::TestRole WindowsDeathTest::AssumeRole() {
07.     const UnitTestImpl* const impl = GetUnitTestImpl();
08.     const InternalRunDeathTestFlag* const flag =
09.         impl->internal_run_death_test_flag();
10.     const TestInfo* const info = impl->current_test_info();
11.     const int death_test_index = info->result()->death_test_count();
12.
13.     if (flag != NULL) {
14.         // ParseInternalRunDeathTestFlag() has performed all the necessary
15.         // processing.
16.         set_write_fd(flag->write_fd());
17.         return EXECUTE_TEST;
18.     }
}
```

这段代码的注释写的很清楚,父进程将向子进程传递什么样的参数。

和之前一样,需要获取flag,如果不是NULL,则是子进程,设置写入句柄,并返回自己角色。如果父进程,则返回EXECUTE_TEST。下面逻辑

```
[cpp]
01. // WindowsDeathTest uses an anonymous pipe to communicate results of
02. // a death test.
03. SECURITY_ATTRIBUTES handles_are_inheritable = {
04.     sizeof(SECURITY_ATTRIBUTES), NULL, TRUE };
05. HANDLE read_handle, write_handle;
06. GTEST_DEATH_TEST_CHECK(
07.     ::CreatePipe(&read_handle, &write_handle, &handles_are_inheritable,
08.         0) // Default buffer size.
09.     != FALSE);
10. set_read_fd(::open_osfhandle(reinterpret_cast<intptr_t>(read_handle),
11.     0_RDONLY));
12. write_handle_.Reset(write_handle);
13. event_handle_.Reset(::CreateEvent(
14.     &handles_are_inheritable,
15.     TRUE, // The event will automatically reset to non-signaled state.
16.     FALSE, // The initial state is non-signalled.
17.     NULL)); // The even is unnamed.
18. GTEST_DEATH_TEST_CHECK(event_handle_.Get() != NULL);
19. const std::string filter_flag =
20.     std::string("--") + GTEST_FLAG_PREFIX_ + kFilterFlag + "=" +
21.     info->test_case_name() + "." + info->name();
22. const std::string internal_flag =
23.     std::string("--") + GTEST_FLAG_PREFIX_ + kInternalRunDeathTestFlag +
24.     "=" + file_ + "|" + StreamableToString(line_) + "|" +
25.     StreamableToString(death_test_index) + "|" +
26.     StreamableToString(static_cast<unsigned int>(::GetCurrentProcessId())) +
27.     // size_t has the same width as pointers on both 32-bit and 64-bit
28.     // Windows platforms.
29.     // See http://msdn.microsoft.com/en-us/library/tcxfdw6.aspx.
30.     "|" + StreamableToString(reinterpret_cast<size_t>(write_handle)) +
31.     "|" + StreamableToString(reinterpret_cast<size_t>(event_handle_.Get()));
32.
33. char executable_path[_MAX_PATH + 1]; // NOLINT
34. GTEST_DEATH_TEST_CHECK(
35.     _MAX_PATH + 1 != ::GetModuleFileNameA(NULL,
36.         executable_path,
37.         _MAX_PATH));
38.
39. std::string command_line =
40.     std::string(::GetCommandLineA()) + " " + filter_flag + " \"" +
41.     internal_flag + "\"";
```

```

42.
43.   DeathTest::set_last_death_test_message("");
44.
45.   CaptureStderr();
46.   // Flush the log buffers since the log streams are shared with the child.
47.   FlushInfoLog();
48.
49.   // The child process will share the standard handles with the parent.
50.   STARTUPINFOA startup_info;
51.   memset(&startup_info, 0, sizeof(STARTUPINFO));
52.   startup_info.dwFlags = STARTF_USESTDHANDLES;
53.   startup_info.hStdInput = ::GetStdHandle(STD_INPUT_HANDLE);
54.   startup_info.hStdOutput = ::GetStdHandle(STD_OUTPUT_HANDLE);
55.   startup_info.hStdError = ::GetStdHandle(STD_ERROR_HANDLE);
56.
57.   PROCESS_INFORMATION process_info;
58.   GTEST_DEATH_TEST_CHECK(::CreateProcessA(
59.       executable_path,
60.       const_cast<char*>(command_line.c_str()),
61.       NULL, // Retuned process handle is not inheritable.
62.       NULL, // Retuned thread handle is not inheritable.
63.       TRUE, // Child inherits all inheritable handles (for write_handle_).
64.       0x0, // Default creation flags.
65.       NULL, // Inherit the parent's environment.
66.       UnitTest::GetInstance()->original_working_dir(),
67.       &startup_info,
68.       &process_info) != FALSE);
69.   child_handle_.Reset(process_info.hProcess);
70.   ::CloseHandle(process_info.hThread);
71.   set_spawned(true);
72.   return OVERSEE_TEST;

```

这段逻辑创建了父进程和子进程通信的匿名管道和事件句柄,这些都通过命令行参数传递给子进程。

我们再看下父进程等待的过程

```

[cpp]
01. int WindowsDeathTest::Wait() {
02.     if (!spawned())
03.         return 0;
04.
05.     // Wait until the child either signals that it has acquired the write end
06.     // of the pipe or it dies.
07.     const HANDLE wait_handles[2] = { child_handle_.Get(), event_handle_.Get() };
08.     switch (::WaitForMultipleObjects(2,
09.                                     wait_handles,
10.                                     FALSE, // Waits for any of the handles.
11.                                     INFINITE)) {
12.     case WAIT_OBJECT_0:
13.     case WAIT_OBJECT_0 + 1:
14.         break;
15.     default:
16.         GTEST_DEATH_TEST_CHECK(false); // Should not get here.
17.     }
18.
19.     // The child has acquired the write end of the pipe or exited.
20.     // We release the handle on our side and continue.
21.     write_handle_.Reset();
22.     event_handle_.Reset();
23.
24.     ReadAndInterpretStatusByte();

```

它等待子进程句柄或者完成事件。一旦等到,则在ReadAndInterpretStatusByte中读取子进程的输出

```

[cpp]
01. void DeathTestImpl::ReadAndInterpretStatusByte() {
02.     char flag;
03.     int bytes_read;
04.
05.     // The read() here blocks until data is available (signifying the
06.     // failure of the death test) or until the pipe is closed (signifying
07.     // its success), so it's okay to call this in the parent before
08.     // the child process has exited.
09.     do {
10.         bytes_read = posix::Read(read_fd(), &flag, 1);
11.     } while (bytes_read == -1 && errno == EINTR);
12.
13.     if (bytes_read == 0) {

```

```
14.         set_outcome(DIED);
15.     } else if (bytes_read == 1) {
16.         switch (flag) {
17.             case kDeathTestReturned:
18.                 set_outcome(RETURNED);
19.                 break;
20.             case kDeathTestThrew:
21.                 set_outcome(THREW);
22.                 break;
23.             case kDeathTestLived:
24.                 set_outcome(LIVED);
25.                 break;
26.             case kDeathTestInternalError:
27.                 FailFromInternalError(read_fd()); // Does not return.
28.                 break;
29.             default:
30.                 GTEST_LOG_(FATAL) << "Death test child process reported "
31.                                     << "unexpected status byte ("
32.                                     << static_cast<unsigned int>(flag) << ")";
33.         }
34.     } else {
35.         GTEST_LOG_(FATAL) << "Read from death test child process failed: "
36.                             << GetLastErrorDescription();
37.     }
38.     GTEST_DEATH_TEST_CHECK_SYSCALL_(posix::Close(read_fd()));
39.     set_read_fd(-1);
40. }
```

这段代码可以用于区分子进程的退出状态。如果子进程crash了,则读取不到数据,进入第14行。

子进程则是执行完表达式后调用Abort返回相应错误。GTEST_DEATH_TEST_剩下的实现,把这个过程表达的很清楚

```
[cpp]
01. if (gtest_dt != NULL) { \
02.     ::testing::internal::scoped_ptr< ::testing::internal::DeathTest> \
03.         gtest_dt_ptr(gtest_dt); \
04.     switch (gtest_dt->AssumeRole()) { \
05.         case ::testing::internal::DeathTest::OVERSEE_TEST: \
06.             if (!gtest_dt->Passed(predicate(gtest_dt->Wait()))) { \
07.                 goto GTEST_CONCAT_TOKEN_(gtest_label_, __LINE__); \
08.             } \
09.             break; \
10.         case ::testing::internal::DeathTest::EXECUTE_TEST: { \
11.             ::testing::internal::DeathTest::ReturnSentinel \
12.                 gtest_sentinel(gtest_dt); \
13.             GTEST_EXECUTE_DEATH_TEST_STATEMENT_(statement, gtest_dt); \
14.             gtest_dt->Abort(::testing::internal::DeathTest::TEST_DID_NOT_DIE); \
15.             break; \
16.         } \
17.         default: \
18.             break; \
19.     } \
20. }
```

顶 踩
1 0

上一篇 Google Test(GTest)使用方法和源码解析——模板类测试技术分析和应用

下一篇 GoogleLog(GLog)源码分析

相关文章推荐

- 玩转Google开源C++单元测试框架Google Test系...
- Google Test(GTest)使用方法和源码解析——结果...
- 【直播】机器学习之凸优化--马博士
- 【套餐】2017软考系统集成项目管理工程师顺利通...
- Google开源C++单元测试框架gTest 5:死亡测试
- Google Test(GTest)使用方法和源码解析——自定...
- 【直播】计算机视觉原理及实战--屈教授
- 【课程】深入探究Linux/VxWorks的设备树--宋宝华
- Google Test(GTest)使用方法和源码解析——断言...
- Google Test(GTest)使用方法和源码解析——Liste...
- 机器学习&数据挖掘7周实训--韦玮
- 分析两种Dump(崩溃日志)文件生成的方法及比较
- Google Test(GTest)使用方法和源码解析——预处...
- minidumpwritetodump在release版本调用失败问题
- 机器学习之数学基础系列--AI100
- 了解Gtest中的事件机制

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

[公司简介](#) | [招贤纳士](#) | [广告服务](#) | [联系方式](#) | [版权声明](#) | [法律顾问](#) | [问题报告](#) | [合作伙伴](#) | [论坛反馈](#)[网站客服](#) [杂志客服](#) [微博客服](#) webmaster@csdn.net 400-660-0108 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2017, CSDN.NET, All Rights Reserved 