**POCO C++库学习和分析 -- 异常、错误处理、调试**

**1. 异常处理**

        C++同C语言相比，提供了异常机制。通过使用try,catch关键字可以捕获异常，这种机制使得程序员在程序异常发生时，可以通过判断异常类型，来决定程序是否继续执行，并在程序结束之前优雅的释放各类资源。当然对于C++的异常机制也存在着很多的争议。在这里，并不对此展开讨论，只介绍一下Poco中的异常类。  
  
  
        **Poco中的异常类：**  
        1. 所有的异常类都是Poco::Exception的子类。  
        2. Poco::Exception继承自std::exception类。  
        3. Foundation库中涉及的异常类，包括了下面一些:  
                   a) Poco::LogicException类负责处理程序错误，包括了:  
                           AssertionViolationException  
                          NullPointerException  
                           NullValueException  
                           BugcheckException  
                           InvalidArgumentException  
                           NotImplementedException  
                           RangeException  
                           IllegalStateException  
                           InvalidAccessException  
                           SignalException  
                           UnhandledException  
                   b) Poco::ApplicationException类负责处理应用程序相关的错误，即使用Poco库的用户自定义异常。  
                   c) Poco::RuntimeException类负责处理程序运行时的错误，包括了:  
                           RuntimeException  
                           NotFoundException  
                           ExistsException  
                           TimeoutException  
                           SystemException  
                           RegularExpressionException  
                           LibraryLoadException  
                           LibraryAlreadyLoadedException  
                           NoThreadAvailableException  
                           PropertyNotSupportedException  
                           PoolOverflowException  
                           NoPermissionException  
                           OutOfMemoryException  
                           DataException  
                           DataFormatException  
                           SyntaxException  
                           CircularReferenceException  
                           PathSyntaxException  
                           IOException  
                           ProtocolException  
                           FileException  
                           FileExistsException  
                           FileNotFoundException  
                           PathNotFoundException  
                           FileReadOnlyException  
                           FileAccessDeniedException  
                           CreateFileException  
                           OpenFileException  
                           WriteFileException  
                           ReadFileException  
                           UnknownURISchemeException  
  
  
**成员函数及数据定义:**  
        1. Poco::Exception包括了一个名字，这是一个静态的字符串,用来描述异常本身。比如说LogicException名字为"Logic exception",TimeoutException名字为"Timeout"。  
        2. Poco::Exception还包含了一个字符串消息，这是用来进一步描述异常的。使用的的人可以在运行时定义它。比如都是LogicException异常,函数一处抛出异常时可定义为"Function1"，函数二处抛出时异常时可定义为用"Function2"，它可以用来说明异常发生的具体位置和原因。  
        3. 一个可选的嵌套异常类  
        4. 构造函数：  
                   a) 可以使用0个，1个或2个字符串参数来构造异常。在Poco::Exception内部存储的时候，第二个字符串会使用字符":"和第一个字符串串联。  
                   b) 构造时如果使用了字符串和嵌套异常的方式，嵌套异常会被复制一份。  
        5. Poco::Exception支持拷贝和赋值运算符  
        6. const char\* name()  
                   返回异常的名称  
        7. const std::string& message()  
                   返回在构造时传入的消息字符串  
        8. std::string displayText() const  
                   同时返回异常名字和消息字符串，中间使用": "分隔  
        9. const Exception\* nested() const  
                   如果存在嵌套异常的话，返回之歌指向嵌套异常的指针，否则返回0  
        10. Exception\* clone() const  
                   返回一个异常的拷贝  
        11. void rethrow() const  
                   重新抛出异常  
  
  
**定义自己的异常：**  
        因为从Poco::Exception继承，去定义自己的异常时，工作非常的枯燥且重复(用户需要重载大量的虚函数)，在库中提供了两个宏来完成这个工作:  
                   POCO\_DECLARE\_EXCEPTION：用来申明异常宏  
                   POCO\_IMPLEMENT\_EXCEPTION： 用来定义异常宏的执行体

        两个宏分别定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698353)

1. // MyException.h
2. #include "Poco/Exception.h"
3. POCO\_DECLARE\_EXCEPTION(MyLib\_API, MyException, Poco::Exception)

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698353)

1. // MyException.cpp
2. #include "MyException.h"POCO\_IMPLEMENT\_EXCEPTION(MyException, Poco::Exception,"Something really bad happened...")

        宏展开分别为：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698353)

1. // MyException.h
2. #include "Poco/Exception.h"
3. POCO\_DECLARE\_EXCEPTION(MyLib\_API, MyException, Poco::Exception)
4. **class** MyLib\_API MyException: **public** Poco::Exception
5. {
6. **public**:
7. MyException();
8. MyException(**const** std::string& msg);
9. MyException(**const** std::string& msg, **const** std::string& arg);
10. MyException(**const** std::string& msg, **const** Poco::Exception& nested);
11. MyException(**const** MyException& exc);
12. ~MyException();
13. MyException& operator = (**const** MyException& exc);
14. **const** **char**\* name() **const**;
15. ...
16. };

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698353)

1. // MyException.cpp
2. #include "MyException.h"
3. POCO\_IMPLEMENT\_EXCEPTION(MyException, Poco::Exception,
4. "Something really bad happened...")
5. ...
6. **const** **char**\* MyException::name() **const** **throw**()
7. {
8. **return** "Something really bad happened...";
9. }
10. ...

        下面是一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698353)

1. #include "Poco/Exception.h"
2. #include <iostream>
3. **int** main(**int** argc, **char**\*\* argv)
4. {
5. Poco::Exception\* pExc = 0;
6. **try**
7. {
8. **throw** Poco::ApplicationException("just testing");
9. }
10. **catch** (Poco::Exception& exc)
11. {
12. pExc = exc.clone();
13. }
14. **try**
15. {
16. pExc->rethrow();
17. }
18. **catch** (Poco::Exception& exc)
19. {
20. std::cerr << exc.displayText() << std::endl;
21. }
22. **delete** pExc;
23. **return** 0;
24. }

**2. 断言**

        POCO库中提供了一些断言的宏来进行运行时检查，这些断言能够提供出错代码的行号和文件信息。  
        1. Debugger::\_assert(cond)  
         如果cond ≠ true时，抛出一个AssertionViolationException异常。  
        2. poco\_assert\_dbg(cond)  
           同poco\_assert类似，但是只在debug模式下起作用  
        3. poco\_check\_ptr(ptr)  
           如果ptr为空，则抛出NullPointerException异常  
        4. poco\_bugcheck(), poco\_bugcheck\_msg(string)  
           抛出BugcheckException异常  
  
        POCO的断言类在debug调试模式下(比如在Visual C++)中时，会触发一个breakpoint。比如：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698353)

1. **void** foo(Bar\* pBar)
2. {
3. poco\_check\_ptr (pBar);
4. ...
5. }
6. **void** baz(**int** i)
7. {
8. poco\_assert (i >= 1 && i < 3);
9. **switch** (i)
10. {
11. **case** 1:
12. ...
13. **break**;
14. **case** 2:
15. ...
16. **break**;
17. **default**:
18. poco\_bugcheck\_msg("i has invalid value");
19. }
20. }

        这主要是因为Poco中的断言类是通过Poco::Debugger去实现的，在Poco::Debugger底层调用了不同操作系统的API，去判断程序是否处于调试状态。如VC下,调用了

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698353)

1. **BOOL** WINAPI IsDebuggerPresent(**VOID**);
2. **VOID** WINAPI DebugBreak(**VOID**);

**3. NDC(Nested Diagnostic Context)**

**3.1  概述**

        NestedDiagnosticContext是为了多线程诊断而设计的。我们在写程序时，一般都需要同时处理多个线程。为了更加便捷的处理多线程情况，为每个线程产生各自的日志。Neil Harrison 在他的书中"[Patterns for Logging Diagnostic Messages,](http://c2.com/cgi/wiki?PatternsForLoggingDiagnosticMessages)" in Pattern Languages of Program Design 3, edited by R. Martin, D. Riehle, and F. Buschmann (Addison-Wesley, 1997) 中提出了一个方法。独特地标记每个日志请求，用户把上下文信息送入NDC，NDC是 Nested Diagnostic Context的缩写。在这本书里提到了3种日志方法，分别是：  
        1. DiagnosticLogger  
         分离日志和程序其他模块  
        2. TransactionalBuckets  
        事务桶，为事务单独建立日志  
        3. TypedDiagnostics  
        类型化诊断，为所有的诊断信息提供统一的展现  
  
  
我们还是回到Poco中的NDC上。在Poco中和NDC相关的内容包括了，NestedDiagnosticContext类，NDCScope类，宏poco\_ndc和poco\_ndc\_dbg。其中NestedDiagnosticContext类维护一个NDC对象，其中包括了上下文的栈信息，有函数方法名，源文件代码文件名，行号。宏poco\_ndc(func) or poco\_ndc\_dbg(func)申明了一个NDCScope对象。而NDCScope对象则完成了上下文的入栈工作。下面是一个例子：

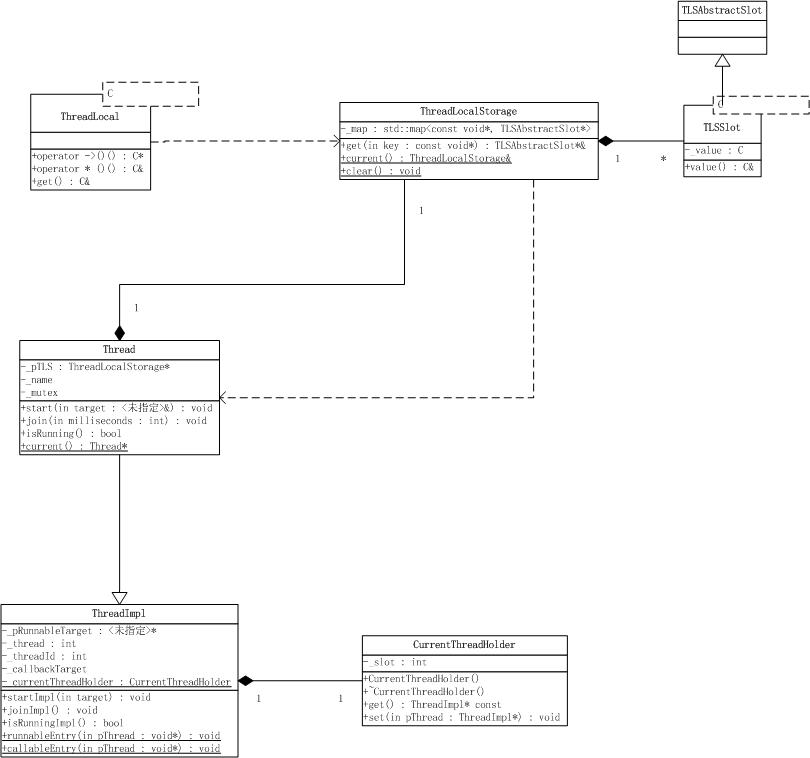
**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698353)

1. #include "Poco/NestedDiagnosticContext.h"
2. #include <iostream>
3. **void** f1()
4. {
5. poco\_ndc(f1);
6. Poco::NDC::current().dump(std::cout);
7. }
8. **void** f2()
9. {
10. poco\_ndc(f2);
11. f1();
12. }
13. **int** main(**int** argc, **char**\*\* argv)
14. {
15. f2();
16. **return** 0;
17. }

**3.2 实现**

**3.2.1 线程本地存储**

        在Poco中实现时，用了一些小技巧，即线程本地存储。我们来看Poco中TLS的类图：



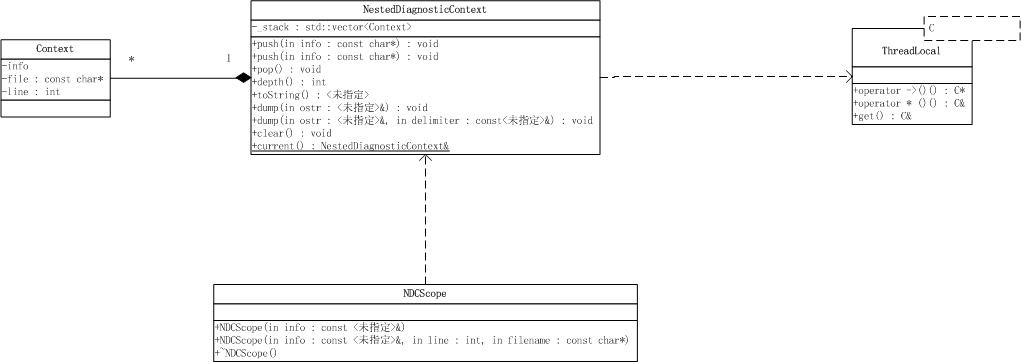
        CurrentThreadHolder类是TLS实现的具体类，在每个Thread对象中包含了一个CurrentThreadHolder对象。Thread创建的时候，CurrentThreadHolder会调用不同操作系统的API函数，获取并保存一个固定槽位，用于保存Thread对象的指针。  
        每个Thread对象中还包含了一个ThreadLocalStorage对象。ThreadLocalStorage类用于保存具体的线程信息数据，它是一个TLSSlot对象的集合。通过泛型实现TLSSlot后，ThreadLocalStorage可用于保存任何数据的。

        使用了TLS技术后，调用Thread的静态函数current可以获取到每个线程对象Thread的指针，然后再通过这个Thread对象的指针，可以获取到ThreadLocalStorage对象，并最终获取或保存数据于TLSSlot中。

        通过类的静态函数获取类实例的指针，在C++中是不存在的，这需要操作系统支持，只有Thread对象才能做到这一点。

**3.2.2 NDC**

        在来看一张Poco中NDC类的类图：



        使用者通过调用宏poco\_ndc和poco\_ndc\_dbg，来构建一个NDCScope对象。宏定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698353)

1. #define poco\_ndc(func) \
2. Poco::NDCScope \_theNdcScope(#func, \_\_LINE\_\_, \_\_FILE\_\_)
4. #if defined(\_DEBUG)
5. #define poco\_ndc\_dbg(func) \
6. Poco::NDCScope \_theNdcScope(#func, \_\_LINE\_\_, \_\_FILE\_\_)
7. #else
8. #define poco\_ndc\_dbg(func)
9. #endif

        NDCScope实现了诊断信息上下文的入栈出栈工作，它通过调用NestedDiagnosticContext类的静态函数current实现了此功能。其定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698353)

1. **inline** NDCScope::NDCScope(**const** std::string& info)
2. {
3. NestedDiagnosticContext::current().push(info);
4. }
6. **inline** NDCScope::NDCScope(**const** std::string& info, **int** line, **const** **char**\* filename)
7. {
8. NestedDiagnosticContext::current().push(info, line, filename);
9. }
11. **inline** NDCScope::~NDCScope()
12. {
13. NestedDiagnosticContext::current().pop();
14. }

        NestedDiagnosticContext类的current()是个静态函数，其定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698353)

1. **namespace**
2. {
3. **static** ThreadLocal<NestedDiagnosticContext> ndc;
4. }
6. NestedDiagnosticContext& NestedDiagnosticContext::current()
7. {
8. **return** ndc.get();
9. }

        而ThreadLocal是一个辅助类，用于获取线程对象的本地存储信息或者是主线程的本地存储信息。

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698353)

1. **template** <**class** C>
2. **class** ThreadLocal
3. /// This template is used to declare type safe thread
4. /// local variables. It can basically be used like
5. /// a smart pointer class with the special feature
6. /// that it references a different object
7. /// in every thread. The underlying object will
8. /// be created when it is referenced for the first
9. /// time.
10. /// See the NestedDiagnosticContext class for an
11. /// example how to use this template.
12. /// Every thread only has access to its own
13. /// thread local data. There is no way for a thread
14. /// to access another thread's local data.
15. {
16. **typedef** TLSSlot<C> Slot;
18. **public**:
19. ThreadLocal()
20. {
21. }
23. ~ThreadLocal()
24. {
25. }
27. C\* operator -> ()
28. {
29. **return** &get();
30. }
32. C& operator \* ()
33. /// "Dereferences" the smart pointer and returns a reference
34. /// to the underlying data object. The reference can be used
35. /// to modify the object.
36. {
37. **return** get();
38. }
40. C& get()
41. /// Returns a reference to the underlying data object.
42. /// The reference can be used to modify the object.
43. {
44. TLSAbstractSlot\*& p = ThreadLocalStorage::current().get(**this**);
45. **if** (!p) p = **new** Slot;
46. **return** **static\_cast**<Slot\*>(p)->value();
47. }
49. **private**:
50. ThreadLocal(**const** ThreadLocal&);
51. ThreadLocal& operator = (**const** ThreadLocal&);
52. };

        到这里Poco中所有的NDC流程都被打通了，用户终于可以实现按线程打印日志信息了。

（版权所有，转载时请注明作者和出处  <http://blog.csdn.net/arau_sh/article/details/8698353>）