**POCO C++库学习和分析 --  线程 （一）**

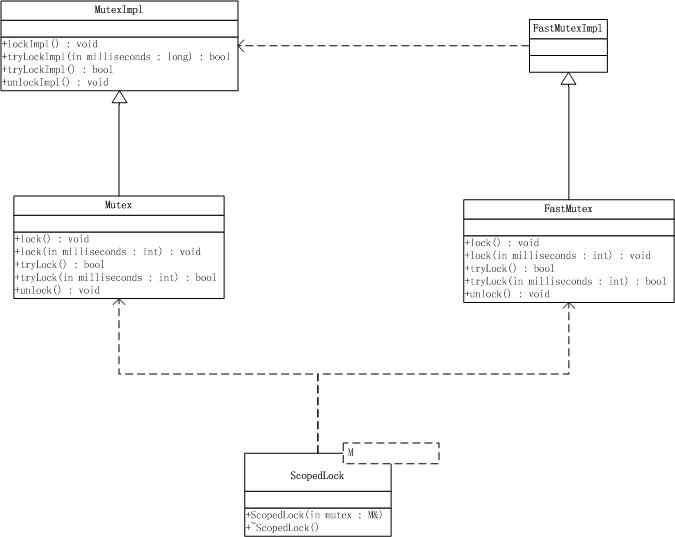
         线程是程序设计中用的非常多的技术，在UI设计，网络通讯设计中广泛使用。在POCO库中，线程模块可以分成6个部分去理解。锁（Lock），线程（Thread），主动对象（ActiveObject），线程池（ThreadPool）, 定时器(Timer)。下面对它们分别介绍。

**1．  数据保护-锁**

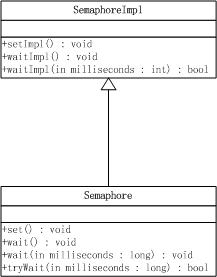
        线程是并行计算中比较复杂的技术之一，使用线程去设计问题时，在获取并行的好处时，也产生了racecondition的问题。锁的存在就是为了解决该问题。

POCO库封装了常见的几种锁，Mutex，Semaphore，Event，Scopelock，ReadWriteLock。类图分别如下：

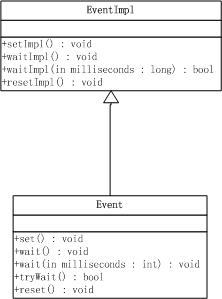
**Mutex**



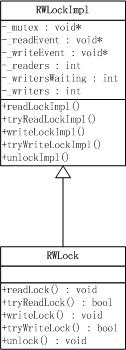
**Semaphore**



**Event**



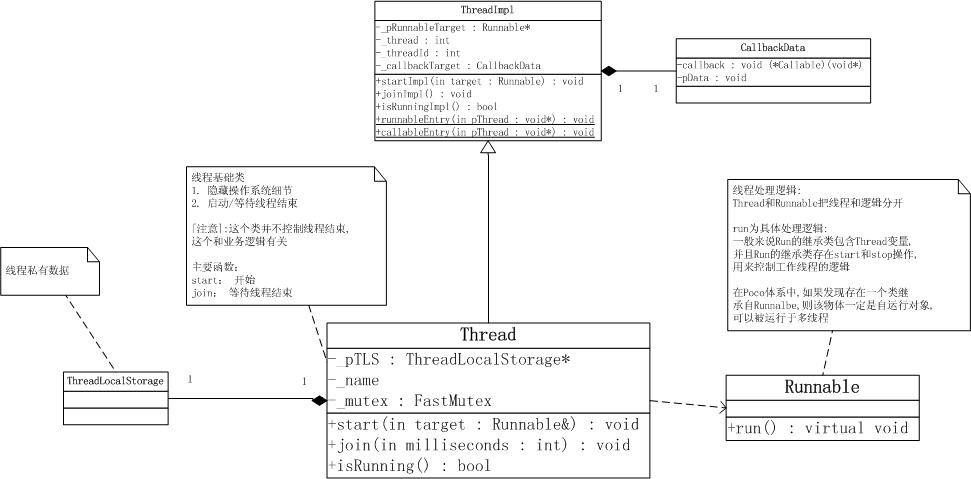
**ReadWriteLock**



         类图非常的简单。就不再多说了，有兴趣的朋友可以自己去看。对于不同平台， POCO基本上选择了比较好的实现方式。比如在Mutex的实现时，Window上用的是criticalsection而非mutex。

**2．  线程**

        POCO对不同操作系统的线程进行了分装，使其变成了一个对象。下面是其的类图：



         熟悉JAVA的朋友一定会很开心，这不就是JAVA中使用线程的两种形式之一吗。所有的业务逻辑全部在Runnable中。Thread类只负责开始(Start)和停止(Join)两个动作。

         来看一下Thread的实现，在C++中底层API (windows下 \_beginthreadex, linux下pthread\_create)创建线程时必须要求入口函数是个全局或者静态函数，这要求业务具有唯一性。而事实上不同的线程就是为了完成不同业务的，不同对象对应不同线程。那变化时如何被封装至Thread类中呢。答案在下面。

void ThreadImpl::createImpl(Entry ent, void\* pData)

{

#if defined(\_DLL)

\_thread = CreateThread(NULL, \_stackSize, ent, pData, 0, &\_threadId);

#else

unsigned threadId;

\_thread = (HANDLE) \_beginthreadex(NULL, \_stackSize, ent, this, 0, &threadId);

\_threadId = static\_cast<DWORD>(threadId);

#endif

if (!\_thread)

throw SystemException("cannot create thread");

if (\_prio != PRIO\_NORMAL\_IMPL && !SetThreadPriority(\_thread, \_prio))

throw SystemException("cannot set thread priority");

}

#if defined(\_DLL)

DWORD WINAPI ThreadImpl::callableEntry(LPVOID pThread)

#else

unsigned \_\_stdcall ThreadImpl::callableEntry(void\* pThread)

#endif

{

\_currentThreadHolder.set(reinterpret\_cast<ThreadImpl\*>(pThread));

#if defined(\_DEBUG) && defined(POCO\_WIN32\_DEBUGGER\_THREAD\_NAMES)

setThreadName(-1, reinterpret\_cast<Thread\*>(pThread)->getName().c\_str());

#endif

try

{

ThreadImpl\* pTI = reinterpret\_cast<ThreadImpl\*>(pThread);

pTI->\_callbackTarget.callback(pTI->\_callbackTarget.pData);

}

catch (Exception& exc)

{

ErrorHandler::handle(exc);

}

catch (std::exception& exc)

{

ErrorHandler::handle(exc);

}

catch (...)

{

ErrorHandler::handle();

}

return 0;

}

在ThreadImpl::createImpl(Entry ent, void\* pData)函数中创建线程时beginthreadex带入了this指针，也就是线程对象本身。

\_thread = (HANDLE) \_beginthreadex(NULL, \_stackSize, ent, this, 0, &threadId);

线程对象本身存在一个结构体CallbackData，其中callback指向了真实的业务路口。不同线程对象在初始化时，会被赋值不同的业务入口函数。

而在静态函数callableEntry中，通过调用this指针可以运行真正的业务函数。

ThreadImpl\* pTI = reinterpret\_cast<ThreadImpl\*>(pThread);

pTI->\_callbackTarget.callback(pTI->\_callbackTarget.pData);

最后用一段代码实例来结束吧

#include "Poco/Thread.h"

#include "Poco/Runnable.h"

#include <iostream>

class HelloRunnable: public Poco::Runnable

{

virtual void run()

{

std::cout << "Hello, world!" << std::endl;

}

};

int main(int argc, char\*\* argv)

{

HelloRunnable runnable;

Poco::Thread thread;

thread.start(runnable);

thread.join();

return 0;

}