**POCO C++库学习和分析 --  线程 （二）**

**3．**  **线程池**

**3.1线程池的基本概念**

       首先我们来明确线程池的一些概念。

**什么是线程池？线程池的好处？**

       池的英文名：POOL，可以被理解成一个容器。线程池就是放置线程对象的容器。我们知道线程的频繁创建、销毁，是需要耗费一点的系统资源的，如果能够预先创建一系列空线程，在需要使用线程时侯，从线程池里，直接获取IDLE线程，则省去了线程创建的过程，当有频繁的线程出现的时候对性能有比较大的好处，程序执行起来将非常效率。

**什么时候推荐使用线程池？**

       很明显，线程越频繁的被创建和释放，越是能体现出线程池的作用。这时候当然推荐使用线程池。

**什么时候不推荐使用线程池？**

       推荐线程池使用的反面情况喽。

       比如长时间运行的线程（线程运行的时间越长，其创建和销毁的开销在其生命周期中比重越低）。

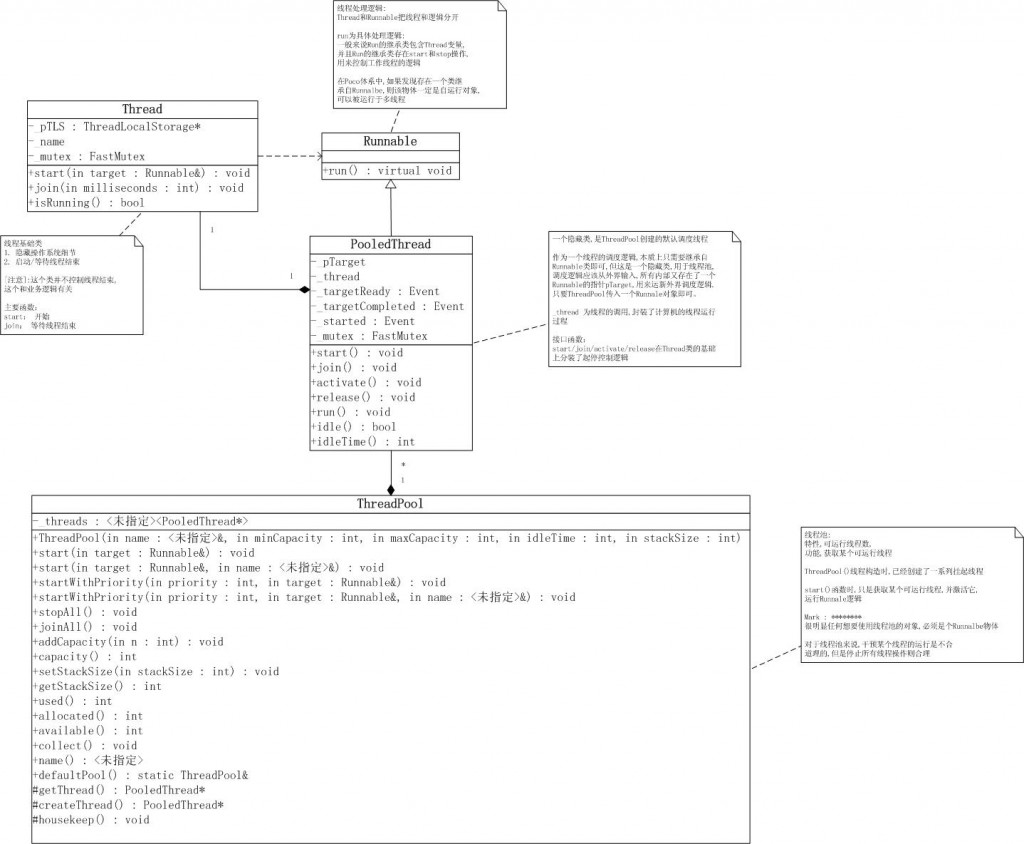
       需要永久标识来标识和控制线程，比如想使用专用线程来终止该线程，将其挂起或按名称发现它。因为线程池中的线程都是平等的。

**线程池需要具备的元素**

* 线程池要有列表，可以用来管理多个线程对象。
* 线程池中的线程，具体执行的内容，可自定义。
* 线程池中的线程，使用完毕后，还能被收回，供下次使用。
* 线程池要提供获取空闲（IDLE）线程方法。当然这个方法可以被封装在线程池中，成为其内部接口。

**3.2 Poco中线程池实现**

       先看一看Poco中内存池的类图吧。



       对于Poco中的线程池来说，设计上分成了两层。第一层为ThreadPool，第二层为PooledThread对象。

       第一层中，ThreadPool负责管理线程池，定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8592579)

1. **class** ThreadPool
2. {
3. **public**:
4. ThreadPool(**int** minCapacity = 2,
5. **int** maxCapacity = 16,
6. **int** idleTime = 60,
7. **int** stackSize = POCO\_THREAD\_STACK\_SIZE);
8. ThreadPool(**const** std::string& name,
9. **int** minCapacity = 2,
10. **int** maxCapacity = 16,
11. **int** idleTime = 60,
12. **int** stackSize = POCO\_THREAD\_STACK\_SIZE);
13. ~ThreadPool();
14. **void** addCapacity(**int** n);
15. **int** capacity() **const**;
16. **void** setStackSize(**int** stackSize);
17. **int** getStackSize() **const**;
18. **int** used() **const**;
19. **int** allocated() **const**;
20. **int** available() **const**;
21. **void** start(Runnable& target);
22. **void** start(Runnable& target, **const** std::string& name);
23. **void** startWithPriority(Thread::Priority priority, Runnable& target);
24. **void** startWithPriority(Thread::Priority priority, Runnable& target, **const** std::string& name);
25. **void** stopAll();
26. **void** joinAll();
27. **void** collect();
28. **const** std::string& name() **const**;
29. **static** ThreadPool& defaultPool();
31. **protected**:
32. PooledThread\* getThread();
33. PooledThread\* createThread();
35. **void** housekeep();
37. **private**:
38. ThreadPool(**const** ThreadPool& pool);
39. ThreadPool& operator = (**const** ThreadPool& pool);
41. **typedef** std::vector<PooledThread\*> ThreadVec;
43. std::string \_name;
44. **int** \_minCapacity;
45. **int** \_maxCapacity;
46. **int** \_idleTime;
47. **int** \_serial;
48. **int** \_age;
49. **int** \_stackSize;
50. ThreadVec \_threads;
51. **mutable** FastMutex \_mutex;
52. };

       从ThreadPool的定义看，它是一个PooledThread对象的容器。职责分成两部分：

       第一，维护和管理池属性，如增加线程池线程数目，返回空闲线程数目，结束所有线程

       第二，把需要运行的业务委托给PooledThread对象，通过接口start(Runnable& target)

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8592579)

1. **void** ThreadPool::start(Runnable& target)
2. {
3. getThread()->start(Thread::PRIO\_NORMAL, target);
4. }

       函数getThread()为ThreadPool的私有函数，作用是获取一个空闲的PooledThread线程对象，实现如下

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8592579)

1. PooledThread\* ThreadPool::getThread()
2. {
3. FastMutex::ScopedLock lock(\_mutex);
5. **if** (++\_age == 32)
6. housekeep();
8. PooledThread\* pThread = 0;
9. **for** (ThreadVec::iterator it = \_threads.begin(); !pThread && it != \_threads.end(); ++it)
10. {
11. **if** ((\*it)->idle()) pThread = \*it;
12. }
13. **if** (!pThread)
14. {
15. **if** (\_threads.size() < \_maxCapacity)
16. {
17. pThread = createThread();
18. **try**
19. {
20. pThread->start();
21. \_threads.push\_back(pThread);
22. }
23. **catch** (...)
24. {
25. **delete** pThread;
26. **throw**;
27. }
28. }
29. **else** **throw** NoThreadAvailableException();
30. }
31. pThread->activate();
32. **return** pThread;
33. }

         第二层中PooledThread对象为一个在线程池中线程。作为线程池中的线程，其创建于线程池的创建时，销毁于线程池的销毁，生命周期同线程池。在其存活的周期中，状态可分为running task和idle。running状态为正在运行业务任务，idle为线程为闲置状态。Poco中PooledThread继承自Runnable，并且包含一个Thread对象。

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8592579)

1. **class** PooledThread: **public** Runnable
2. {
3. **public**:
4. PooledThread(**const** std::string& name, **int** stackSize = POCO\_THREAD\_STACK\_SIZE);
5. ~PooledThread();
7. **void** start();
8. **void** start(Thread::Priority priority, Runnable& target);
9. **void** start(Thread::Priority priority, Runnable& target, **const** std::string& name);
10. **bool** idle();
11. **int** idleTime();
12. **void** join();
13. **void** activate();
14. **void** release();
15. **void** run();
17. **private**:
18. **volatile** **bool**        \_idle;
19. **volatile** std::**time\_t** \_idleTime;
20. Runnable\*            \_pTarget;
21. std::string          \_name;
22. Thread               \_thread;
23. Event                \_targetReady;
24. Event                \_targetCompleted;
25. Event                \_started;
26. FastMutex            \_mutex;
27. };

         对于PooledThread来说，其线程业务就是不断的检测是否有新的外界业务\_pTarget，如果有就运行，没有的话，把自己状态标志位限制，供线程池回收。

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8592579)

1. **void** PooledThread::run()
2. {
3. \_started.set();
4. **for** (;;)
5. {
6. \_targetReady.wait();
7. \_mutex.lock();
8. **if** (\_pTarget) // a NULL target means kill yourself
9. {
10. \_mutex.unlock();
11. **try**
12. {
13. \_pTarget->run();
14. }
15. **catch** (Exception& exc)
16. {
17. ErrorHandler::handle(exc);
18. }
19. **catch** (std::exception& exc)
20. {
21. ErrorHandler::handle(exc);
22. }
23. **catch** (...)
24. {
25. ErrorHandler::handle();
26. }
27. FastMutex::ScopedLock lock(\_mutex);
28. \_pTarget  = 0;
29. #if defined(\_WIN32\_WCE)
30. \_idleTime = wceex\_time(NULL);
31. #else
32. \_idleTime = time(NULL);
33. #endif
34. \_idle     = **true**;
35. \_targetCompleted.set();
36. ThreadLocalStorage::clear();
37. \_thread.setName(\_name);
38. \_thread.setPriority(Thread::PRIO\_NORMAL);
39. }
40. **else**
41. {
42. \_mutex.unlock();
43. **break**;
44. }
45. }
46. }

         Poco中线程池的实现，耦合性其实是很低的，这不得不归功于其在线程池上两个层次的封装和抽象，类的内聚性非常强的，每个类各干各的事。

**3.3 其他**

         除了上面线程池的主要属性和接口外，Poco中线程池还实现了一些其他特性。如设置线程运行的优先级，实现了一个默认线程的单件等。

（版权所有，转载时请注明作者和出处 <http://blog.csdn.net/arau_sh/article/details/8592579>）