**POCO C++库学习和分析 -- 线程 （四）**

**5. 主动对象**

**5.1 线程回顾**

        在讨论主动对象之前，我想先说一下对于Poco中多线程编程的理解。大家都知道，对于多线程编程而言最基本的元素只有两个数据:锁和线程。线程提高了程序的效率，也带来了数据的竞争，因此为了保证数据的正确性，孪生兄弟"锁"随之产生。

        对于不同的操作系统和编程语言而言，线程和锁通常是以系统API的方式提供的，不同语言和不同操作系统下API并不相同，但线程和锁的特性是一致的，这也是对线程和锁进行封装的基础。比如所有的系统线程API都提供了线程开始函数，其中可以设置线程的入口函数，提供了线程终止等功能。用面对对象的思想对线程和锁进行封装后，线程和锁就可以被看成编程时的一个基本粒子，一堆积木中的一个固定模块，用来搭建更大的组件。

        除了线程和锁这两个基本模块之外，定时器和线程池也比较常用。线程池多用作线程频繁创建的时候。在Poco中，把线程池封装成为一个对象，池中的线程在池存在时始终存活，只不过是线程状态有所不同，不是运行中就是挂起。如果把线程看成一种资源的话，线程资源的申请和释放被放入了线程池的构造和析构函数中，Poco的这种封装也就是C++推荐的方法。  
  
       在Poco的线程池实现中，ThreadPool类还提供了一个线程池的单件接口。这个由静态函数定义：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8618571)

1. **static** ThreadPool& defaultPool();

        通过这个函数，使用者可以很方便的从Poco库中获取一个线程的起点，而无需关心线程维护的细节，这样使用者可以进一步把注意力放在需要实现的业务上。在实现了ThreadPool的这个接口后，Poco类中关于线程的更高级封装即可以实现。如定时器(Timer)，主动对象(Activity Object)，任务(Task)。  
  
        在Poco实现定时器，实现ThreadPool中的PooledThread，以及接下来要讨论的主动对象中的ActiveRunnable，RunnableAdapter，ActiveDispatcher时，可以发现这些类都从Runnable继承。这些类需要实现自己的run函数，只不过在run函数中做的工作不同。

        定时器中的run函数工作就是计时，定期更新实现，至触发时刻运行使用者定义的用户事件。而PooledThread的工作则是，控制线程状态，在挂起和运行间切换，当有用户业务需要运行时，运行用户业务。说穿了这些类都是用户业务的一个代理。只不过代理时，实现的手段不同。

**5.2 主动对象**

        总结了前几章后，让我们继续往下看一下主动对象。首先是什么是主动对象。Poco中对于主动对象有如下描述：  
        主动对象是一个对象，这个对象使用自己线程运行自己的成员函数。  
        1.  在Poco中，主动对象支持两种主动成员函数。  
        2.  Activity类型的主动对象使用在用户业务为不需要返回值和无参数的成员函数时侯。  
        3.  ActiveMethod类型的主动对象使用在用户业务为需要返回值和需要参数的成员函数时侯。  
        4.  所有的主动对象即能够共享一个单线程，也可以拥有其自己的线程。  
  
        事实上在Poco库中实现了3种类型的主动对象，分别是Activity、ActiveMethod、ActiveDispatcher。其中ActiveDispatcher可以看成是ActiveMethod的一个变种。Activity和ActiveMethod的区别在于是否需要关心用户业务的返回值，因为ActiveMethod模板实现时也特化了一个没有用户参数的版本。  
  
        其实我们自己也能很容易的实现一个主动对象。简单的接口如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8618571)

1. **class** MyObject : **public** Runnable
2. {
3. **public**:
4. **void** start();
5. **void** stop();
6. **void** run();

9. **public**:
10. **void** realrun();

13. **public**:
14. Thread \_thread;
15. };

18. **void** MyObject::start()
19. {
20. \_thread.start();
21. }

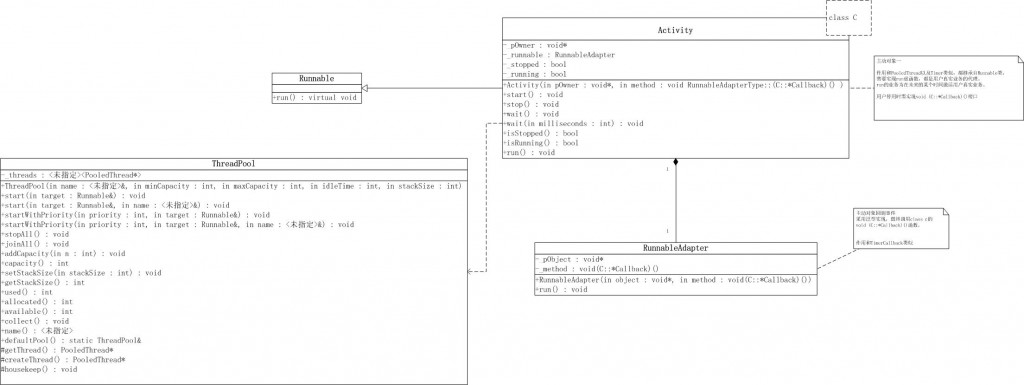
24. **void** MyObject::stop()
25. {
26. \_thread.join();
27. }

30. **void** MyObject::run()
31. {
32. **this**->realrun();
33. }

        从上面分析一下，实现一个主动对象需要些什么：  
        1.  一个线程驱动。在上例中主动对象包含了一个Thread对象  
        2.  用户真实业务(在上例中由函数MyObject::realrun提供)，在继承自Runnabled的封装类的run函数中被调用。

        Poco中主动对象要做的事情是很类似的，但是Poco提供的是一个框架，供开发者使用的框架。这个框架即需要满足用户需求，坚固，还要求便于使用。为了实现这个框架需要一些编程上的技巧。对于Poco的开发者而言，使用Poco库的人就是他们的用户。使用者必须很容易的通过Poco库把自己的类变成一个主动对象，而Poco的开发者很明显并不知道用户会如何定义一个的类。所以实现这样一种可变的结构，C++语言最适合方法的无疑是泛型编程了。下面我们来具体说一下Poco中的主动对象。

**5.3 Activities**

        首先来看一下Activities的特性：  
        1.  Activities能够在对象构造时自动启动，也能够稍后手动启动  
        2.  Activities能够在任何时候被停止。为了完成这个工作，isStopped()成员函数必须周期性的被调用。  
        3. Activities主动对象运行的成员函数不能够携带参数和返回值  
        4. Activities的线程驱动来自于默认的线程库  
  
        来看一下Activities的类图：  
  
  
        从类图中我们可以看到Activity的线程驱动来自于默认的线程库。在Activity中为了调用到用户的真实业务函数，需要把对象实例和类的函数入口传进Activity中。这在Activity的构造函数中实现。

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8618571)

1. Activity(C\* pOwner, Callback method):
2. \_pOwner(pOwner),
3. \_runnable(\*pOwner, method),
4. \_stopped(**true**),
5. \_running(**false**),
6. \_done(**false**)
7. /// Creates the activity. Call start() to
8. /// start it.
9. {
10. poco\_check\_ptr (pOwner);
11. }

        通过泛型，pOwner可以指向任何外界定义的实例。Activity由于包含了线程驱动，在start()中调用了ThreadPool::defaultPool().start(\*this)，所以对于调用者而言，可以被看成一个线程驱动。  
        RunnableAdapter看名字就是一个适配类，用于存储调用对象的指针和调用类的入口函数。

        下面是Activity的使用例子。

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8618571)

1. #include "Poco/Activity.h"
2. #include "Poco/Thread.h"
3. #include <iostream>
4. **using** Poco::Thread;
5. **class** ActivityExample
6. {
7. **public**:
8. ActivityExample(): \_activity(**this**,
9. &ActivityExample::runActivity)
10. {}
11. **void** start()
12. {
13. \_activity.start();
14. }
15. **void** stop()
16. {
17. \_activity.stop(); // request stop
18. \_activity.wait(); // wait until activity actually stops
19. }
20. **protected**:
21. **void** runActivity()
22. {
23. **while** (!\_activity.isStopped())
24. {
25. std::cout << "Activity running." << std::endl;
26. Thread::sleep(200);
27. }
28. }
29. **private**:
30. Poco::Activity<ActivityExample> \_activity;
31. };
33. **int** main(**int** argc, **char**\*\* argv)
34. {
35. ActivityExample example;
36. example.start();
37. Thread::sleep(2000);
38. example.stop();
39. **return** 0;
40. }

        在上例中，可以看到使用类ActivityExample包容了一个\_activity对象。为了能够在任何时刻停止，在ActivityExample的真实业务runActivity()函数中，定期调用了\_activity.isStopped()函数。

**5.4 Active Methods**

        来看一下Poco中Active Methods的特性：  
        Active Methods主动对象拥有一个能在自身线程中运行的包含参数和返回值成员函数方法，其线程驱动也来自于默认线程池。  
        1. 主动对象能够共享一个线程。当一个主动对象运行时，其他的对象等待。  
        2. 运行业务的成员函数可以拥有一个参数并能返回一个值。  
        3. 如果函数需要传递更多的参数，可以使用结构体、std::pair、或者Poco::Tuple.  
        4. Active Methods主动对象的结果由Poco::ActiveResult 提供。  
        5. 通常主动对象的函数的返回值不会在调用函数后立刻返回，所以在设计时设计了Poco::ActiveResult类。  
        6. ActiveResult是一个模板类，在函数返回结果时被创建。  
        7. ActiveResult的返回结果可能是一个需要的结果，也可能是一个异常。  
        8. 使用者通过ActiveResult::wait()函数等待到结果，通过ActiveResult::data()获取真实返回值。

        抛开上面文档中提到的Active Methods的特性不提。Active Methods和Activities的区别在于Active Methods调用的自身函数拥有返回值和参数。也就是说，在Active Methods中类对象、类对象的入口函数、返回值和传入参数都是未定的。所以用泛型实现时，ActiveMethod定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8618571)

1. **template** <**class** ResultType, **class** ArgType, **class** OwnerType, **class** StarterType = ActiveStarter<OwnerType> >
2. **class** ActiveMethod
3. {
4. // ...
5. }

        其中ResultType代表了返回值的共性，ArgType代表了输入参数的共性，OwnerType代表了业务调用拥有者的共性，StarterType代表了线程驱动的共性。类对象的入口函数则被包装进入另一个类ActiveRunnable中，其定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8618571)

1. **template** <**class** ResultType, **class** ArgType, **class** OwnerType>
2. **class** ActiveRunnable: **public** ActiveRunnableBase
3. {
4. // ...
5. }

        让我们来看一个Active Methods的例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8618571)

1. #include "Poco/ActiveMethod.h"
2. #include "Poco/ActiveResult.h"
3. #include <utility>
4. **using** Poco::ActiveMethod;
5. **using** Poco::ActiveResult;
6. **class** ActiveAdder
7. {
8. **public**:
9. ActiveAdder():
10. add(**this**, &ActiveAdder::addImpl)
11. {}
12. ActiveMethod<**int**, std::pair<**int**, **int**>, ActiveAdder> add;
13. **private**:
14. **int** addImpl(**const** std::pair<**int**, **int**>& args)
15. {
16. **return** args.first + args.second;
17. }
18. };
20. **int** main(**int** argc, **char**\*\* argv)
21. {
22. ActiveAdder adder;
23. ActiveResult<**int**> sum = adder.add(std::make\_pair(1, 2));
24. sum.wait();
25. std::cout << sum.data() << std::endl;
26. **return** 0;
27. }

        在上面这个例子中ActiveMethod初始化方式也和Activity类似，ActiveAdder拥有一个ActiveMethod对象，在构造时对add进行实例化，传入了业务调用对象实例的指针，调用对象函数的入口地址。

        不同的地方是：

        1. 让我们来仔细观察下面这行代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8618571)

1. ActiveResult<**int**> sum = adder.add(std::make\_pair(1, 2));

        这行代码是立刻返回的，sum此时并没有得到真实的运行结果。只有等到sum.wait()返回，真实的运算结果才被放置于sum中。  
        正是由于ActiveResult<int> sum = adder.add(std::make\_pair(1, 2))立刻返回，才让出了执行的主线程，让多线程的威力显现出来。这和把等待任务函数放在Activities中不同，ActiveMethod由于有结果交换的过程，其等待函数放于结果返回值类ActiveResult<T>中。

        2. 为了传入参数和传出结果，通过泛型规范了其定义

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8618571)

1. ActiveMethod<**int**, std::pair<**int**, **int**>, ActiveAdder>

        3. 深度封装了线程，因为从调用方来看根本无法看到线程的影子。没有start,stop等函数.   
  
        第二点很有意思，能够实现这一点，在于在ActiveMethod中重载了操作符().下面是其定义：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8618571)

1. ActiveResultType operator () (**const** ArgType& arg)
2. /// Invokes the ActiveMethod.
3. {
4. ActiveResultType result(**new** ActiveResultHolder<ResultType>());
5. ActiveRunnableBase::Ptr pRunnable(**new** ActiveRunnableType(\_pOwner, \_method, arg, result));
6. StarterType::start(\_pOwner, pRunnable);
7. **return** result;
8. }

       上面这个操作符就是线程的起点。

        StarterType::start(\_pOwner, pRunnable)定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8618571)

1. **static** **void** ActiveStarter::start(OwnerType\* pOwner, ActiveRunnableBase::Ptr pRunnable)
2. {
3. ThreadPool::defaultPool().start(\*pRunnable);
4. pRunnable->duplicate(); // The runnable will release itself.
5. }

        在调用StarterType::start(\_pOwner, pRunnable)后，接下来会进入ActiveRunnableType对象的run函数。ActiveRunnable::run()函数定义为：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8618571)

1. **void** run()
2. {
3. ActiveRunnableBase::Ptr guard(**this**, **false**); // ensure automatic release when done
4. **try**
5. {
6. \_result.data(**new** ResultType((\_pOwner->\*\_method)(\_arg)));
7. }
8. **catch** (Exception& e)
9. {
10. \_result.error(e);
11. }
12. **catch** (std::exception& e)
13. {
14. \_result.error(e.what());
15. }
16. **catch** (...)
17. {
18. \_result.error("unknown exception");
19. }
20. \_result.notify();
21. }

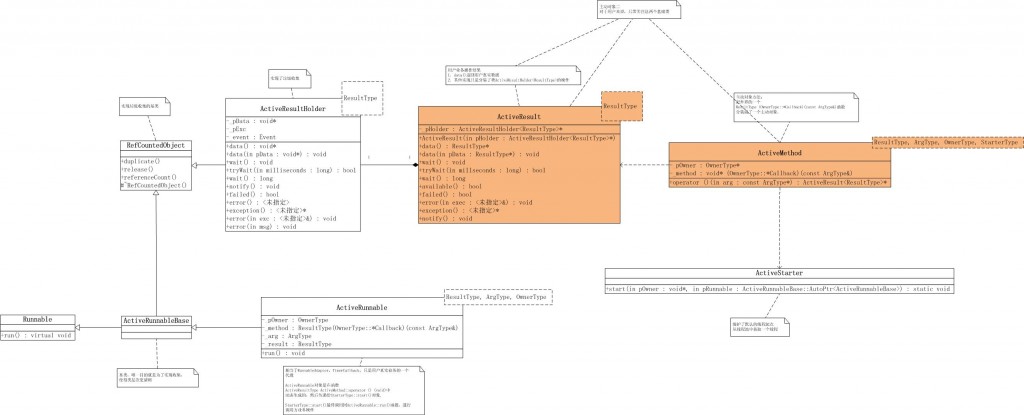
        语句\_result.data(new ResultType((\_pOwner->\*\_method)(\_arg)))完成了用户函数的真实调用。  
  
        那线程的停止呢。这个被封装到ActiveResult中,ActiveResult::wait()定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8618571)

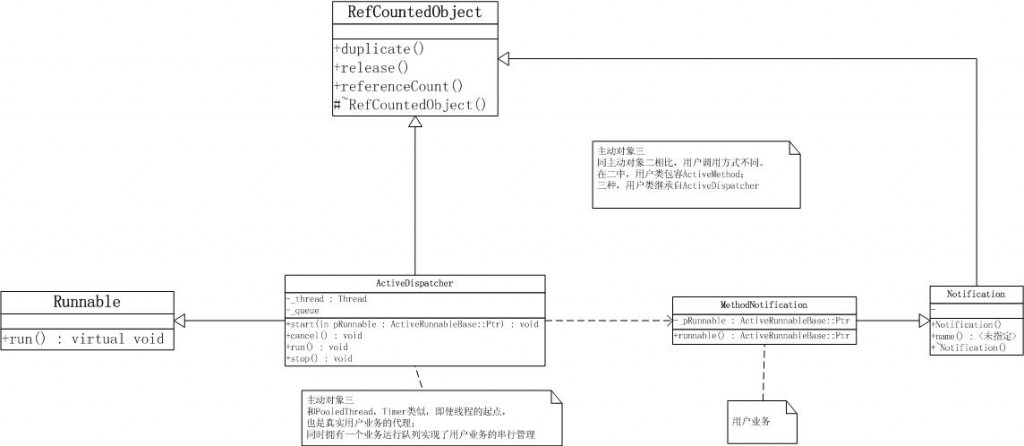
1. **void** ActiveResult::wait()
2. /// Pauses the caller until the result becomes available.
3. {
4. \_pHolder->wait();
5. }

       于是整个流程呼之欲出。

  Active Methods的类图如下：



**5.5 ActiveDispatcher**

        ActiveDispatcher的特性如下：  
        1. ActiveMethod的默认行为并不符合经典的主动对象概念，经典的主动对象定义要求主动对象支持多个方法，并且各方法能够在单线程中被顺序执行。  
        2. 为了实现经典主动对象的行为，ActiveDispatcher被设计成主动对象的基类。  
        3. 通过使用ActiveDispatcher可以顺序执行用户业务函数。  
  
        让我们来看一下ActiveDispatcher的类图：  
  
  
        在类图中用户业务被打包成为MethodNotification对象放入了ActiveDispatcher的queue容器中，然后被顺序执行。其中Notification和NotificationQueue我们在以后介绍。  
        下面是其一个实现的例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8618571)

1. #include "Poco/ActiveMethod.h"
2. #include "Poco/ActiveResult.h"
3. #include "Poco/ActiveDispatcher.h"
4. #include <utility>
5. **using** Poco::ActiveMethod;
6. **using** Poco::ActiveResult;
7. **class** ActiveAdder: **public** Poco::ActiveDispatcher
8. {
9. **public**:
10. ActiveObject(): add(**this**, &ActiveAdder::addImpl)
11. {}
12. ActiveMethod<**int**, std::pair<**int**, **int**>, ActiveAdder,
13. Poco::ActiveStarter<Poco::ActiveDispatcher> > add;
14. **private**:
15. **int** addImpl(**const** std::pair<**int**, **int**>& args)
16. {
17. **return** args.first + args.second;
18. }
19. };
21. **int** main(**int** argc, **char**\*\* argv)
22. {
23. ActiveAdder adder;
24. ActiveResult<**int**> sum = adder.add(std::make\_pair(1, 2));
25. sum.wait();
26. std::cout << sum.data() << std::endl;
27. **return** 0;
28. }

**5.6 其他**

        在讨论完Poco主动对象的实现侯，回过来我们再讨论一下为什么要使用主动对象，使用Poco中主动对象的好处。

        使用主动对象的好处很明显，能够使业务的划分更加清晰。但是我并不推荐在真实的项目中使用Poco封装的主动对象，除非是一些非常简单的场景。原因如下：

        1. 在真实的业务应用中，我们能够很容易的在Thread和runnable基础上自己封装一个主动对象。  
        2. 真实业务不需要抽象，业务都是具体的。不使用主动对象的可读性会更加好。  
  
        虽然我不推荐在真实项目中使用Poco的主动对象，但是我觉得学习Poco中主动对象的实现仍然很有意义，特别是泛型和其他一些编程技巧。  
  
  
  
（版权所有，转载时请注明作者和出处  <http://blog.csdn.net/arau_sh/article/details/8618571>）