[**实战Concurrent**](http://www.iteye.com/topic/363625)

编写多线程的程序一直都是一件比较麻烦的事情，要考虑很多事情，处理不好还会出很多意想不到的麻烦。加上现在很多开发者接触到的项目都是打着企业级旗号的B/S项目，大多数人都很少涉及多线程，这又为本文的主角增加了一份神秘感。

讲到Java多线程，大多数人脑海中跳出来的是Thread、Runnable、synchronized……这些是最基本的东西，虽然已经足够强大，但想要用好还真不容易。从JDK 1.5开始，增加了java.util.concurrent包，它的引入大大简化了多线程程序的开发（要感谢一下大牛Doug Lee）。

java.util.concurrent包分成了三个部分，分别是java.util.concurrent、java.util.concurrent.atomic和java.util.concurrent.lock。内容涵盖了并发集合类、线程池机制、同步互斥机制、线程安全的变量更新工具类、锁等等常用工具。

为了便于理解，本文使用一个例子来做说明，交代一下它的场景：

假设要对一套10个节点组成的环境进行检查，这个环境有两个入口点，通过节点间的依赖关系可以遍历到整个环境。依赖关系可以构成一张有向图，可能存在环。为了提高检查的效率，考虑使用多线程。

**1、Executors**

通过这个类能够获得多种线程池的实例，例如可以调用newSingleThreadExecutor()获得单线程的ExecutorService，调用newFixedThreadPool()获得固定大小线程池的ExecutorService。拿到ExecutorService可以做的事情就比较多了，最简单的是用它来执行Runnable对象，也可以执行一些实现了Callable<T>的对象。用Thread的start()方法没有返回值，如果该线程执行的方法有返回值那用ExecutorService就再好不过了，可以选择submit()、invokeAll()或者invokeAny()，根据具体情况选择合适的方法即可。

**Java代码  [收藏代码](javascript:void())**

1. **package** service;
3. **import** java.util.ArrayList;
4. **import** java.util.List;
5. **import** java.util.concurrent.ExecutionException;
6. **import** java.util.concurrent.ExecutorService;
7. **import** java.util.concurrent.Executors;
8. **import** java.util.concurrent.Future;
9. **import** java.util.concurrent.TimeUnit;
11. /\*\*
12. \* 线程池服务类
13. \*
14. \* @author DigitalSonic
15. \*/
16. **public** **class** ThreadPoolService {
17. /\*\*
18. \* 默认线程池大小
19. \*/
20. **public** **static** **final** **int**  DEFAULT\_POOL\_SIZE    = 5;
22. /\*\*
23. \* 默认一个任务的超时时间，单位为毫秒
24. \*/
25. **public** **static** **final** **long** DEFAULT\_TASK\_TIMEOUT = 1000;
27. **private** **int**              poolSize             = DEFAULT\_POOL\_SIZE;
28. **private** ExecutorService  executorService;
30. /\*\*
31. \* 根据给定大小创建线程池
32. \*/
33. **public** ThreadPoolService(**int** poolSize) {
34. setPoolSize(poolSize);
35. }
37. /\*\*
38. \* 使用线程池中的线程来执行任务
39. \*/
40. **public** **void** execute(Runnable task) {
41. executorService.execute(task);
42. }
44. /\*\*
45. \* 在线程池中执行所有给定的任务并取回运行结果，使用默认超时时间
46. \*
47. \* @see #invokeAll(List, long)
48. \*/
49. **public** List<Node> invokeAll(List<ValidationTask> tasks) {
50. **return** invokeAll(tasks, DEFAULT\_TASK\_TIMEOUT \* tasks.size());
51. }
53. /\*\*
54. \* 在线程池中执行所有给定的任务并取回运行结果
55. \*
56. \* @param timeout 以毫秒为单位的超时时间，小于0表示不设定超时
57. \* @see java.util.concurrent.ExecutorService#invokeAll(java.util.Collection)
58. \*/
59. **public** List<Node> invokeAll(List<ValidationTask> tasks, **long** timeout) {
60. List<Node> nodes = **new** ArrayList<Node>(tasks.size());
61. **try** {
62. List<Future<Node>> futures = **null**;
63. **if** (timeout < 0) {
64. futures = executorService.invokeAll(tasks);
65. } **else** {
66. futures = executorService.invokeAll(tasks, timeout, TimeUnit.MILLISECONDS);
67. }
68. **for** (Future<Node> future : futures) {
69. **try** {
70. nodes.add(future.get());
71. } **catch** (ExecutionException e) {
72. e.printStackTrace();
73. }
74. }
75. } **catch** (InterruptedException e) {
76. e.printStackTrace();
77. }
78. **return** nodes;
79. }
81. /\*\*
82. \* 关闭当前ExecutorService
83. \*
84. \* @param timeout 以毫秒为单位的超时时间
85. \*/
86. **public** **void** destoryExecutorService(**long** timeout) {
87. **if** (executorService != **null** && !executorService.isShutdown()) {
88. **try** {
89. executorService.awaitTermination(timeout, TimeUnit.MILLISECONDS);
90. } **catch** (InterruptedException e) {
91. e.printStackTrace();
92. }
93. executorService.shutdown();
94. }
95. }
97. /\*\*
98. \* 关闭当前ExecutorService，随后根据poolSize创建新的ExecutorService
99. \*/
100. **public** **void** createExecutorService() {
101. destoryExecutorService(1000);
102. executorService = Executors.newFixedThreadPool(poolSize);
103. }
105. /\*\*
106. \* 调整线程池大小
107. \* @see #createExecutorService()
108. \*/
109. **public** **void** setPoolSize(**int** poolSize) {
110. **this**.poolSize = poolSize;
111. createExecutorService();
112. }
113. }

这里要额外说明一下invokeAll()和invokeAny()方法。前者会执行给定的所有Callable<T>对象，等所有任务完成后返回一个包含了执行结果的List<Future<T>>，每个Future.isDone()都是true，可以用Future.get()拿到结果；后者只要完成了列表中的任意一个任务就立刻返回，返回值就是执行结果。

**还有一个比较诡异的地方**

本代码是在JDK 1.6下编译测试的，如果在JDK 1.5下测试，很可能在invokeAll和invokeAny的地方出错。明明ValidationTask实现了 Callable<Node>，可是它死活不认，类型不匹配，这时可以将参数声明由List<ValidationTask>改为 List<Callable<Node>>。  
造成这个问题的主要原因是两个版本中invokeAll和invokeAny的方法签名不同，1.6里是invokeAll(Collection<? extends Callable<T>> tasks)，而1.5里是invokeAll(Collection<Callable<T>> tasks)。网上也有人遇到类似的问题([invokeAll() is not willing to acept a Collection<Callable<T>>](http://stackoverflow.com/questions/370707/invokeall-is-not-willing-to-acept-a-collectioncallablet))。

和其他资源一样，线程池在使用完毕后也需要释放，用shutdown()方法可以关闭线程池，如果当时池里还有没有被执行的任务，它会等待任务执行完毕，在等待期间试图进入线程池的任务将被拒绝。也可以用shutdownNow()来关闭线程池，它会立刻关闭线程池，没有执行的任务作为返回值返回。

**2、Lock**

多线程编程中常常要锁定某个对象，之前会用synchronized来实现，现在又多了另一种选择，那就是java.util.concurrent.locks。通过Lock能够实现更灵活的锁定机制，它还提供了很多synchronized所没有的功能，例如尝试获得锁(tryLock())。

使用Lock时需要自己获得锁并在使用后手动释放，这一点与synchronized有所不同，所以通常Lock的使用方式是这样的：

**Java代码  [收藏代码](javascript:void())**

1. Lock l = ...;
2. l.lock();
3. **try** {
4. // 执行操作
5. } **finally** {
6. l.unlock();
7. }

java.util.concurrent.locks中提供了几个Lock接口的实现类，比较常用的应该是ReentrantLock。以下范例中使用了ReentrantLock进行节点锁定：

**Java代码  [收藏代码](javascript:void())**

1. **package** service;
3. **import** java.util.concurrent.locks.Lock;
4. **import** java.util.concurrent.locks.ReentrantLock;
6. /\*\*
7. \* 节点类
8. \*
9. \* @author DigitalSonic
10. \*/
11. **public** **class** Node {
12. **private** String name;
13. **private** String wsdl;
14. **private** String result = "PASS";
15. **private** String[] dependencies = **new** String[] {};
16. **private** Lock lock = **new** ReentrantLock();
17. /\*\*
18. \* 默认构造方法
19. \*/
20. **public** Node() {
21. }
23. /\*\*
24. \* 构造节点对象，设置名称及WSDL
25. \*/
26. **public** Node(String name, String wsdl) {
27. **this**.name = name;
28. **this**.wsdl = wsdl;
29. }
31. /\*\*
32. \* 返回包含节点名称、WSDL以及验证结果的字符串
33. \*/
34. @Override
35. **public** String toString() {
36. String toString = "Node: " + name + " WSDL: " + wsdl + " Result: " + result;
37. **return** toString;
38. }
40. // Getter & Setter
41. **public** String getName() {
42. **return** name;
43. }
45. **public** **void** setName(String name) {
46. **this**.name = name;
47. }
49. **public** String getWsdl() {
50. **return** wsdl;
51. }
53. **public** **void** setWsdl(String wsdl) {
54. **this**.wsdl = wsdl;
55. }
57. **public** String getResult() {
58. **return** result;
59. }
61. **public** **void** setResult(String result) {
62. **this**.result = result;
63. }
65. **public** String[] getDependencies() {
66. **return** dependencies;
67. }
69. **public** **void** setDependencies(String[] dependencies) {
70. **this**.dependencies = dependencies;
71. }
73. **public** Lock getLock() {
74. **return** lock;
75. }
77. }

**Java代码  [收藏代码](javascript:void())**

1. **package** service;
3. **import** java.util.concurrent.Callable;
4. **import** java.util.concurrent.locks.Lock;
5. **import** java.util.logging.Logger;
7. **import** service.mock.MockNodeValidator;
9. /\*\*
10. \* 执行验证的任务类
11. \*
12. \* @author DigitalSonic
13. \*/
14. **public** **class** ValidationTask **implements** Callable<Node> {
15. **private** **static** Logger logger = Logger.getLogger("ValidationTask");
17. **private** String        wsdl;
19. /\*\*
20. \* 构造方法，传入节点的WSDL
21. \*/
22. **public** ValidationTask(String wsdl) {
23. **this**.wsdl = wsdl;
24. }
26. /\*\*
27. \* 执行针对某个节点的验证<br/>
28. \* 如果正有别的线程在执行同一节点的验证则等待其结果，不重复执行验证
29. \*/
30. @Override
31. **public** Node call() **throws** Exception {
32. Node node = ValidationService.NODE\_MAP.get(wsdl);
33. Lock lock = **null**;
34. logger.info("开始验证节点：" + wsdl);
35. **if** (node != **null**) {
36. lock = node.getLock();
37. **if** (lock.tryLock()) {
38. // 当前没有其他线程验证该节点
39. logger.info("当前没有其他线程验证节点" + node.getName() + "[" + wsdl + "]");
40. **try** {
41. Node result = MockNodeValidator.validateNode(wsdl);
42. mergeNode(result, node);
43. } **finally** {
44. lock.unlock();
45. }
46. } **else** {
47. // 当前有别的线程正在验证该节点，等待结果
48. logger.info("当前有别的线程正在验证节点" + node.getName() + "[" + wsdl + "]，等待结果");
49. lock.lock();
50. lock.unlock();
51. }
52. } **else** {
53. // 从未进行过验证，这种情况应该只出现在系统启动初期
54. // 这时是在做初始化，不应该有冲突发生
55. logger.info("首次验证节点：" + wsdl);
56. node = MockNodeValidator.validateNode(wsdl);
57. ValidationService.NODE\_MAP.put(wsdl, node);
58. }
59. logger.info("节点" + node.getName() + "[" + wsdl + "]验证结束，验证结果：" + node.getResult());
60. **return** node;
61. }
63. /\*\*
64. \* 将src的内容合并进dest节点中，不进行深度拷贝
65. \*/
66. **private** Node mergeNode(Node src, Node dest) {
67. dest.setName(src.getName());
68. dest.setWsdl(src.getWsdl());
69. dest.setDependencies(src.getDependencies());
70. dest.setResult(src.getResult());
71. **return** dest;
72. }
73. }

请注意ValidationTask的call()方法，这里会先检查节点是否被锁定，如果被锁定则表示当前有另一个线程正在验证该节点，那就不用重复进行验证。第50行和第51行，那到锁后立即释放，这里只是为了等待验证结束。

讲到Lock，就不能不讲Conditon，前者代替了synchronized，而后者则代替了Object对象上的wait()、notify()和notifyAll()方法(Condition中提供了await()、signal()和signalAll()方法)，当满足运行条件前挂起线程。Condition是与Lock结合使用的，通过Lock.newCondition()方法能够创建与Lock绑定的Condition实例。JDK的JavaDoc中有一个例子能够很好地说明Condition的用途及用法：

**Java代码  [收藏代码](javascript:void())**

1. **class** BoundedBuffer {
2. **final** Lock lock = **new** ReentrantLock();
3. **final** Condition notFull  = lock.newCondition();
4. **final** Condition notEmpty = lock.newCondition();
6. **final** Object[] items = **new** Object[100];
7. **int** putptr, takeptr, count;
9. **public** **void** put(Object x) **throws** InterruptedException {
10. lock.lock();
11. **try** {
12. **while** (count == items.length)
13. notFull.await();
14. items[putptr] = x;
15. **if** (++putptr == items.length) putptr = 0;
16. ++count;
17. notEmpty.signal();
18. } **finally** {
19. lock.unlock();
20. }
21. }
23. **public** Object take() **throws** InterruptedException {
24. lock.lock();
25. **try** {
26. **while** (count == 0)
27. notEmpty.await();
28. Object x = items[takeptr];
29. **if** (++takeptr == items.length) takeptr = 0;
30. --count;
31. notFull.signal();
32. **return** x;
33. } **finally** {
34. lock.unlock();
35. }
36. }
37. }

说到这里，让我解释一下之前的例子里为什么没有选择Condition来等待验证结束。await()方法在调用时当前线程先要获得对应的锁，既然我都拿到锁了，那也就是说验证已经结束了。。。

**3、并发集合类**

集合类是大家编程时经常要使用的东西，ArrayList、HashMap什么的，java.util包中的集合类有的是线程安全的，有的则不是，在编写多线程的程序时使用线程安全的类能省去很多麻烦，但这些类的性能如何呢？java.util.concurrent包中提供了几个并发结合类，例如ConcurrentHashMap、ConcurrentLinkedQueue和CopyOnWriteArrayList等等，根据不同的使用场景，开发者可以用它们替换java.util包中的相应集合类。

CopyOnWriteArrayList是ArrayList的一个变体，比较适合用在读取比较频繁、修改较少的情况下，因为每次修改都要复制整个底层数组。ConcurrentHashMap中为Map接口增加了一些方法(例如putIfAbsenct())，同时做了些优化，总之灰常之好用，下面的代码中使用ConcurrentHashMap来作为全局节点表，完全无需考虑并发问题。ValidationService中只是声明(第17行)，具体的使用是在上面的ValidationTask中。

**Java代码  [收藏代码](javascript:void())**

1. **package** service;
3. **import** java.util.ArrayList;
4. **import** java.util.List;
5. **import** java.util.Map;
6. **import** java.util.concurrent.ConcurrentHashMap;
8. /\*\*
9. \* 执行验证的服务类
10. \*
11. \* @author DigitalSonic
12. \*/
13. **public** **class** ValidationService {
14. /\*\*
15. \* 全局节点表
16. \*/
17. **public** **static** **final** Map<String, Node> NODE\_MAP = **new** ConcurrentHashMap<String, Node>();
19. **private** ThreadPoolService threadPoolService;
21. **public** ValidationService(ThreadPoolService threadPoolService) {
22. **this**.threadPoolService = threadPoolService;
23. }
25. /\*\*
26. \* 给出一个入口节点的WSDL，通过广度遍历的方式验证与其相关的各个节点
27. \*
28. \* @param wsdl 入口节点WSDL
29. \*/
30. **public** **void** validate(List<String> wsdl) {
31. List<String> visitedNodes = **new** ArrayList<String>();
32. List<String> nextRoundNodes = **new** ArrayList<String>();
34. nextRoundNodes.addAll(wsdl);
35. **while** (nextRoundNodes.size() > 0) {
36. List<ValidationTask> tasks = getTasks(nextRoundNodes);
37. List<Node> nodes = threadPoolService.invokeAll(tasks);
39. visitedNodes.addAll(nextRoundNodes);
40. nextRoundNodes.clear();
41. getNextRoundNodes(nodes, visitedNodes, nextRoundNodes);
42. }
43. }
45. **private** List<String> getNextRoundNodes(List<Node> nodes,
46. List<String> visitedNodes, List<String> nextRoundNodes) {
47. **for** (Node node : nodes) {
48. **for** (String wsdl : node.getDependencies()) {
49. **if** (!visitedNodes.contains(wsdl)) {
50. nextRoundNodes.add(wsdl);
51. }
52. }
53. }
54. **return** nextRoundNodes;
55. }
57. **private** List<ValidationTask> getTasks(List<String> nodes) {
58. List<ValidationTask> tasks = **new** ArrayList<ValidationTask>(nodes.size());
59. **for** (String wsdl : nodes) {
60. tasks.add(**new** ValidationTask(wsdl));
61. }
62. **return** tasks;
63. }
64. }

**4、AtomicInteger**

对变量的读写操作都是原子操作（除了long或者double的变量），但像数值类型的++ --操作不是原子操作，像i++中包含了获得i的原始值、加1、写回i、返回原始值，在进行类似i++这样的操作时如果不进行同步问题就大了。好在java.util.concurrent.atomic为我们提供了很多工具类，可以以原子方式更新变量。

以AtomicInteger为例，提供了代替++ --的getAndIncrement()、incrementAndGet()、getAndDecrement()和decrementAndGet()方法，还有加减给定值的方法、当前值等于预期值时更新的compareAndSet()方法。

下面的例子中用AtomicInteger保存全局验证次数(第69行做了自增的操作)，因为validateNode()方法会同时被多个线程调用，所以直接用int不同步是不行的，但用AtomicInteger在这种场合下就很合适。

**Java代码  [收藏代码](javascript:void())**

1. **package** service.mock;
3. **import** java.util.ArrayList;
4. **import** java.util.HashMap;
5. **import** java.util.List;
6. **import** java.util.Map;
7. **import** java.util.concurrent.atomic.AtomicInteger;
8. **import** java.util.logging.Logger;
10. **import** service.Node;
12. /\*\*
13. \* 模拟执行节点验证的Mock类
14. \*
15. \* @author DigitalSonic
16. \*/
17. **public** **class** MockNodeValidator {
18. **public** **static** **final** List<Node>         ENTRIES  = **new** ArrayList<Node>();
19. **private** **static** **final** Map<String, Node> NODE\_MAP = **new** HashMap<String, Node>();
21. **private** **static** AtomicInteger           count    = **new** AtomicInteger(0);
22. **private** **static** Logger                  logger   = Logger.getLogger("MockNodeValidator");
24. /\*
25. \* 构造模拟数据
26. \*/
27. **static** {
28. Node node0 = **new** Node("NODE0", "http://node0/check?wsdl"); //入口0
29. Node node1 = **new** Node("NODE1", "http://node1/check?wsdl");
30. Node node2 = **new** Node("NODE2", "http://node2/check?wsdl");
31. Node node3 = **new** Node("NODE3", "http://node3/check?wsdl");
32. Node node4 = **new** Node("NODE4", "http://node4/check?wsdl");
33. Node node5 = **new** Node("NODE5", "http://node5/check?wsdl");
34. Node node6 = **new** Node("NODE6", "http://node6/check?wsdl"); //入口1
35. Node node7 = **new** Node("NODE7", "http://node7/check?wsdl");
36. Node node8 = **new** Node("NODE8", "http://node8/check?wsdl");
37. Node node9 = **new** Node("NODE9", "http://node9/check?wsdl");
39. node0.setDependencies(**new** String[] { node1.getWsdl(), node2.getWsdl() });
40. node1.setDependencies(**new** String[] { node3.getWsdl(), node4.getWsdl() });
41. node2.setDependencies(**new** String[] { node5.getWsdl() });
42. node6.setDependencies(**new** String[] { node7.getWsdl(), node8.getWsdl() });
43. node7.setDependencies(**new** String[] { node5.getWsdl(), node9.getWsdl() });
44. node8.setDependencies(**new** String[] { node3.getWsdl(), node4.getWsdl() });
46. node2.setResult("FAILED");
48. NODE\_MAP.put(node0.getWsdl(), node0);
49. NODE\_MAP.put(node1.getWsdl(), node1);
50. NODE\_MAP.put(node2.getWsdl(), node2);
51. NODE\_MAP.put(node3.getWsdl(), node3);
52. NODE\_MAP.put(node4.getWsdl(), node4);
53. NODE\_MAP.put(node5.getWsdl(), node5);
54. NODE\_MAP.put(node6.getWsdl(), node6);
55. NODE\_MAP.put(node7.getWsdl(), node7);
56. NODE\_MAP.put(node8.getWsdl(), node8);
57. NODE\_MAP.put(node9.getWsdl(), node9);
59. ENTRIES.add(node0);
60. ENTRIES.add(node6);
61. }
63. /\*\*
64. \* 模拟执行远程验证返回节点，每次调用等待500ms
65. \*/
66. **public** **static** Node validateNode(String wsdl) {
67. Node node = cloneNode(NODE\_MAP.get(wsdl));
68. logger.info("验证节点" + node.getName() + "[" + node.getWsdl() + "]");
69. count.getAndIncrement();
70. **try** {
71. Thread.sleep(500);
72. } **catch** (InterruptedException e) {
73. e.printStackTrace();
74. }
75. **return** node;
76. }
78. /\*\*
79. \* 获得计数器的值
80. \*/
81. **public** **static** **int** getCount() {
82. **return** count.intValue();
83. }
85. /\*\*
86. \* 克隆一个新的Node对象（未执行深度克隆）
87. \*/
88. **public** **static** Node cloneNode(Node originalNode) {
89. Node newNode = **new** Node();
91. newNode.setName(originalNode.getName());
92. newNode.setWsdl(originalNode.getWsdl());
93. newNode.setResult(originalNode.getResult());
94. newNode.setDependencies(originalNode.getDependencies());
96. **return** newNode;
97. }
98. }

上述代码还有另一个功能，就是构造测试用的节点数据，一共10个节点，有2个入口点，通过这两个点能够遍历整个系统。每次调用会模拟远程访问，等待500ms。环境间节点依赖如下：

**环境依赖**

Node0 [Node1, Node2]  
Node1 [Node3, Node4]  
Node2 [Node5]  
Node6 [Node7, Node8]  
Node7 [Node5, Node9]  
Node8 [Node3, Node4]

**5、CountDownLatch**

CountDownLatch是一个一次性的同步辅助工具，允许一个或多个线程一直等待，直到计数器值变为0。它有一个构造方法，设定计数器初始值，即在await()结束等待前需要调用多少次countDown()方法。CountDownLatch的计数器不能重置，所以说它是“一次性”的，如果需要重置计数器，可以使用CyclicBarrier。在运行环境检查的主类中，使用了CountDownLatch来等待所有验证结束，在各个并发验证的线程完成任务结束前都会调用countDown()，因为有3个并发的验证，所以将计数器设置为3。

最后将所有这些类整合起来，运行环境检查的主类如下。它会创建线程池服务和验证服务，先做一次验证(相当于是对系统做次初始化)，随后并发3个验证请求。系统运行完毕会显示实际执行的节点验证次数和执行时间。如果是顺序执行，验证次数应该是13\*4=52，但实际的验证次数会少于这个数字(我这里最近一次执行了33次验证)，因为如果同时有两个线程要验证同一节点时只会做一次验证。关于时间，如果是顺序执行，52次验证每次等待500ms，那么验证所耗费的时间应该是26000ms，使用了多线程后的实际耗时远小于该数字(最近一次执行耗时4031ms)。

**Java代码  [收藏代码](javascript:void())**

1. **package** service.mock;
3. **import** java.util.ArrayList;
4. **import** java.util.List;
5. **import** java.util.concurrent.CountDownLatch;
7. **import** service.Node;
8. **import** service.ThreadPoolService;
9. **import** service.ValidationService;
11. /\*\*
12. \* 模拟执行这个环境的验证
13. \*
14. \* @author DigitalSonic
15. \*/
16. **public** **class** ValidationStarter **implements** Runnable {
17. **private** List<String>      entries;
18. **private** ValidationService validationService;
19. **private** CountDownLatch    signal;
21. **public** ValidationStarter(List<String> entries, ValidationService validationService,
22. CountDownLatch signal) {
23. **this**.entries = entries;
24. **this**.validationService = validationService;
25. **this**.signal = signal;
26. }
28. /\*\*
29. \* 线程池大小为10，初始化执行一次，随后并发三个验证
30. \*/
31. **public** **static** **void** main(String[] args) {
32. ThreadPoolService threadPoolService = **new** ThreadPoolService(10);
33. ValidationService validationService = **new** ValidationService(threadPoolService);
34. List<String> entries = **new** ArrayList<String>();
35. CountDownLatch signal = **new** CountDownLatch(3);
36. **long** start;
37. **long** stop;
39. **for** (Node node : MockNodeValidator.ENTRIES) {
40. entries.add(node.getWsdl());
41. }
43. start = System.currentTimeMillis();
45. validationService.validate(entries);
46. threadPoolService.execute(**new** ValidationStarter(entries, validationService, signal));
47. threadPoolService.execute(**new** ValidationStarter(entries, validationService, signal));
48. threadPoolService.execute(**new** ValidationStarter(entries, validationService, signal));
50. **try** {
51. signal.await();
52. } **catch** (InterruptedException e) {
53. e.printStackTrace();
54. }
56. stop = System.currentTimeMillis();
57. threadPoolService.destoryExecutorService(1000);
58. System.out.println("实际执行验证次数: " + MockNodeValidator.getCount());
59. System.out.println("实际执行时间: " + (stop - start) + "ms");
60. }
62. @Override
63. **public** **void** run() {
64. validationService.validate(entries);
65. signal.countDown();
66. }
68. }

=================================我是分割线==============================

本文没有覆盖java.util.concurrent中的所有内容，只是挑选一些比较常用的东西，想要获得更多详细信息请阅读JavaDoc。自打有了“轮子”理论，重复造大轮子的情况的确少了，但还是有人会做些小轮子，例如编写多线程程序时用到的小工具(线程池、锁等等)，如果可以，请让自己再“懒惰”一点吧~http://www.iteye.com/images/smiles/icon_smile.gif

* [ConcurrentExample.zip](http://dl.iteye.com/topics/download/ec09f2c6-147b-3cd1-9e55-2ab4ced63122) (7.5 KB)
* 下载次数: 1435