Java 并发核心编程

内容涉及:
1、关于 java 并发
2、概念
3、保护共享数据
4、并发集合类
5 线程
6、线程协作及其他

1、关于 java 并发

自从 java 创建以来就已经支持并发的理念，如线程和锁。这篇指南主要是为帮助 java 多线程开发人员理解并发的核心概念以及如何应用这些理念。本文的主题是关于具有 java 语言风格的 Thread、synchronized、volatile，以及 J2SE5 中新增的概念，如锁(Lock)、原子性(Atomics)、并发集合类、线程协作摘要、Executors。开发者通过这些基础的接口可以构建高并发、线程安全的 java 应用程序。

2、概念

本部分描述的 java 并发概念在这篇 DZone Refard 会被通篇使用。
从 JVM 并发看 CPU 内存指令重排序(Memory Reordering):http://kenwublog.com/illustrate-memory-reordering-in-cpu
java 内存模型详解: http://kenwublog.com/explain-java-memory-model-in-detail

概念	描述
Java Memory Model Java 内存模型	在 JavaSE5（JSR133）中定义的 Java Memory Model（JMM）是为了确保当编写并发代码的时候能够提供 Java 程序员一个可用的 JVM 实现。术语 JMM 的作用类似与一个观察同步读写字段的 monitor。它按照“happens-before order（先行发生排序）”的顺序—可以解释为什么一个线程可以获得其他线程的结果，这组成了一个属性同步的程序，使字段具有不变性，以及其他属性。
monitor Monitor	Java 语言中,每个对象都拥有一个访问代码关键部分并防止其他对象访问这段代码的“monitor”（每个对象都拥有一个对代码关键部分提供访问互斥功能的“monitor”）。这段关键部分是使用 synchronized 对方法或者代码标注实现的。同一时间在同一个 monitor 中，只允许一个线程运行代码的任意关键部分。当一个线

	程试图获取代码的关键部分时，如果这段代码的 monitor 被其他线程拥有，那么这个线程会无限期的等待这个 monitor 直到它被其他线程释放。除了访问互斥之外，monitor 还可以通过 wait 和 notify 来实现协作。
原子字段赋值 Atomic field assignment	除了 doubles 和 langs 之外的类型，给一个这些类型的字段赋值是一个原子操作。在 JVM 中，doubles 和 langs 的更新是被实现为 2 个独立的操作，因此理论上可能会有其他的线程得到一个部分更新的结果。为了保护共享的 doubles 和 langs，可以使用 volatile 标记这个字段或者在 synchronized 修饰的代码块中操作字段。
竞争状态 Race condition	竞争发生在当不少于一个线程对一个共享的资源进行一系列的操作，如果这些线程的操作的顺序不同，会导致多种可能的结果。
数据竞争 Data race	数据竞争主要发生在多个线程访问一个共享的、non-final、non-volatile、没有合适的 synchronization 限制的字段。Java 内存模型不会对这种非同步的数据访问提供任何的保证。在不同的架构和机器中数据竞争会导致不可预测的行为。
安全发布 Safe publications	在一个对象创建完成之前就发布它的引用时非常危险的。避免这种使用这种引用的一种方法就是在创建期间注册一个回调接口。另外一种不安全的情况就是在构造子中启动一个线程。在这 2 种情况中，非完全创建的对象对于其他线程来说都是可见的。
不可变字段 Final Fields	不可变字段在对象创建之后必须明确设定一个值，否则编译器就会报出一个错误。一旦设定值后，不可变字段的值就不可以再次改变。将一个对象的引用设定为不可变字段并不能阻止这个对象的改变。例如，ArrayList 类型的不可变字段不能改变为其他 ArrayList 实例的引用，但是可以在这个 list 实例中添加或者删除对象。 在创建结尾,对象会遇到”final field freeze”：如果对象被安全的发布后，即使在没有 synchronization 关键字修饰的情况下，也能保证所有的线程获取 final 字段在构建过程中设定的值。final field freezer 不仅对 final 字段有用，而且作用于 final 对象中的可访问属性。
不可变对象 Immutable objects	在语法上 final 字段能够创建不需要 synchronization 修饰的、能够被共享读取的线程安全的不可变对象。实现 Immutable Object 需要保证如下条件： • 对象被安全的发布（在创建过程中 this 引用是无法避免的） • 所有字段被声明为 final • 在创建之后，在对象字段能够被访问的范围中是不允许修改这个字段的。 • class 被声明为 final（为了防止 subclass 违反这些规则）

3、保护共享数据

编写线程安全的 java 程序，当修改共享数据的时候要求开发人员使用合适的锁来保护数据。锁能够建立符合 Java Memory Model 要求的访问顺序，而且确保其他线程知道数据的变化。
注意:

在 Java Memory Model 中,如果没有被 synchronization 修饰,改变数据不需要什么特别的语法表示。JVM 能够自由地重置指令顺序的特性和对可见性的限制方式很容易让开发人员感到奇怪。

3.1、Synchronized

每个对象实例都拥有一个每次只能让一个线程锁住的 monitor。synchronized 能够用在一个方法或者代码块中来锁住这个 monitor。用 synchronized 修饰一个对象，当修改这个对象的一个字段，synchronized 保证其他线程余下的对这个对象的读操作能够获取修改后的值。需要注意的是修改同步块之外的数据或者 synchronized 没有修饰当前被修改的对象,那么不能保证其他线程读到这些最新的数据。synchronized 关键字能够修饰一个对象实例中的函数或者代码块。在一个非静态方法中 this 关键字表示当前的实例对象。在一个 synchronized 修饰的静态的方法中，这个方法所在的类使用 Class 作为实例对象。

3.2、Lock

Java.util.concurrent.locks 包中有个标准 Lock 接口。ReentrantLock 实现了 Lock 接口，它完全拥有 synchronized 的特性，同时还提供了新的功能：获取 Lock 的状态、非阻塞获取锁的方法 tryLock()、可中断 Lock。
下面是使用 ReentrantLock 的详细示例：

```
public class Counter{
   private final Lock lock = new ReentrantLock();
   private int value;
   public int increment() {
      lock.lock();
      try {
         return ++value;
      }finally{
         lock.unlock();
      }
   }
}
```

3.3、ReadWriteLock

Java.util.concurrent.locks 包中还有个 ReadWriteLock 接口（实现类是 ReentrantWriteReadLock），它定义一对锁：读锁和写锁，特征是能够被并发的读取但每次只能有一个写操作。使用 ReentrantReadWriteLock 并发读取特性的详细示例：

```
public class ReadWrite {
   private final ReadWriteLock lock = new ReentrantReadWriteLock();
   private int value;
   public void increment(){
      lock.writeLock().lock();
      try{
         value++;
      }finally{
         lock.writeLock().lock();
      }
   }
   public int current(){
      lock.readLock().lock();
      try{
         return value;
      }finally{
         lock.readLock().unlock();
      }
   }
}
```

3.4、volatile

volatile 原理与技巧: http://kenwublog.com/the-theory-of-volatile

volatile 修饰符用来标注一个字段，表明任何对这个字段的修改都必须能被其他随后访问的线程获取到，这个修饰符和同步无关。因此，volatile 修饰的数据的可见性和 synchronization 类似，但是这个它只作用于对字段的读或写操作。在 JavaSE5 之前，因为 JVM 的架构和实现的原因，不同 JVM 的 volatile 效果是不同的而且也是不可信的。下面是 Java 内存模型明确地定义 volatile 的行为：

```
public class Processor implements Runnable {
   private volatile boolean stop;
   public void stopProcessing(){
      stop = true;
   }
   public void run() {
      while (!stop) {
```

```
            //do processing
        }
    }
}
```

注意：使用 volatile 修饰一个数组并不能让这个数组的每个元素拥有 volatile 特性，这种声明只是让这个数组的 reference 具有 volatile 属性。数组被声明为 AtomicIntegerArray 类型，则能够拥有类似 volatile 的特性。

3.5、原子类

使用 volatile 的一个缺点是它能够保证数据的可见性，却不能在一个原子操作中对 volatile 修饰的字段同时进行校验和更新操作。java.util.concurrent.atomic 包中有一系列支持在单个非锁定(lock)的变量上进行原子操作的类，类似于 volatile。示例：

```
public class Counter{
   private AtomicInteger value = new AtomicInteger();
   private int value;
   public int increment() {
      return value.incrementAndGet();
   }
}
```

incrementAndGet 方法是原子类的复合操作的一个示例。booleans, integers, longs, object references, integers 数组, longs 数组, object references 数组 都有相应的原子类。

3.6、ThreadLocal

通过 ThreadLocal 能数据保存在一个线程中，而且不需要 lock 同步。理论上 ThreadLocal 可以让一个变量在每个线程都有一个副本。ThreadLocal 常用来屏蔽线程的私有变量，例如“并发事务”或者其他的资源。而且，它还被用来维护每个线程的计数器，统计，或者 ID 生成器。

```
public class TransactionManager {
   private static final ThreadLocal<Transaction>
currentTransaction
      = new ThreadLocal<Transaction>() {
      @Override
      protected Transaction initialValue() {
         return new NullTransaction();
      }
   };
   public Transaction currentTransaction() {
      Transaction current = currentTransaction.get();
      if(current.isNull()) {
         current = new TransactionImpl();
```

```
            currentTransaction.put(current);
        }
        return current;
    }
}
```

4、Concurrent Collections（并发集合类）

保护共享数据的一个关键技术是在存储数据的类中封装同步机制。所有对数据的使用都要经过同步机制的确认使这个技术能够避免数据的不当访问。在 java.util.concurrent 包中有很多为并发使用情况下设计的数据结构。通常，使用这些数据结构比使用同步包装器装饰的非同步的集合的效率更高。

4.1、Concurrent lists and sets

在 Table2 中列出了 java.util.concurrent 包中拥有的 3 个并发的 List 和 Set 实现类。

类	描述
CopyOnWriteArraySet	CopyOnWriteArraySet 在语意上提供写时复制(copy-on-werite)的特性，对这个集合的每次修改都需要对当前数据结构新建一个副本，因此写操作发费很大。在迭代器创建的时候，会对当前数据数据结构创建一个快照用于迭代。
CopyOnWriteArrayList	CopyOnWriteArrayList 和 CopyOnWriteArraySet 类似，也是基于 copy-on-write 语义实现了 List 接口
ConcurrentSkipListSet	ConcurrentSkipListSet（在 JavaSE 6 新增的）提供的功能类似于 TreeSet，能够并发的访问有序的 set。因为 ConcurrentSkipListSet 是基于“跳跃列表（skip list）”实现的，只要多个线程没有同时修改集合的同一个部分，那么在正常读、写集合的操作中不会出现竞争现象。

skip list: http://blog.csdn.net/yuanyufei/archive/2007/02/14/1509937.aspx

http://zh.wikipedia.org/zh-cn/%E8%B7%B3%E8%B7%83%E5%88%97%E8%A1%A8

4.2、Concurrent maps

Java.util.concurrent 包中有个继承 Map 接口的 ConcurrentMap 的接口，ConcurrentMap 提供了一些新的方法（表 3）。所有的这些方法在一个原子操作中各自提供了一套操作步骤。如果将每套步骤在放在 map 之外单独实现，在非原子操作的多线程访问的情况下会导致资源竞争。

表 3：ConcurrentMap 的方法：

方法	描述
putIfAbsent(K key, V value) : V	如果 key 在 map 中不存在，则把 key-value 键值对放入 map 中，否则不执行任何操作。返回值为原来的 value，如果 key 不存在 map 中则返回 null
remove（Object key, Object value) : boolean	如果 map 中有这个 key 及相应的 value，那么移除这对数据，否则不执行任何操作
replace (K key, V value) : V	如果 map 中有这个 key，那么用新的 value 替换原来的 value，否则不执行任何操作
replace (K key, V oldValue, V newValue) : boolean	如果 map 中有这对 key-oldValue 数据，那么用 newValue 替换原来的 oldValue，否则不执行任何操作

在表 4 中列出的是 ConcurrentMap 的 2 个实现类

方法	描述
ConcurrentHashMap	ConcurrentHashMap 提供了 2 种级别的内部哈希方法。第一种级别是选择一个内部的 Segment，第二种是在选定的 Segment 中将数据哈希到 buckets 中。第一种方法通过并行地在不同的 Segment 上进行读写操作来实现并发。(ConcurrentHashMap 是引入了 Segment，每个 Segment 又是一个 hash 表，ConcurrentHashMap 相当于是两级 Hash 表，然后锁是在 Segment 一级进行的，提高了并发性。http://mooncui.javaeye.com/blog/380884 http://www.javaeye.com/topic/344876)
ConcurrentSkipListMap	ConcurrentSkipListMap（JavaSE 6 新增的类）功能类似 TreeMap，是能够被并发访问的排序 map。尽管能够被多线程正常的读写---只要这些线程没有同时修改 map 的同一个部分，ConcurrentSkipListMap 的性能指标和 TreeMap 差不多。

4.3、Queues

Queues 类似于沟通“生产者”和“消费者”的管道。组件从管道的一端放入，然后从另一端取出：“先进先出”(FIFO)的顺序。Queue 接口在 JavaSE5 新添加到 java.util 中的，能够被用于单线程访问的场景中，主要适用于多个生产者、一个或多个消费者的情景，所有的读写操作都是基于同一个队列。

java.util.concurrent 包中的 BlockingQueue 接口是 Queue 的子接口，而且还添加了新的特性处理如下场景：队列满（此时刚好有一个生产者要加入一个新的组件）、队列空（此时刚好有一个消费者读取或者删除一个组件）。BlockingQueue 提供如下方案解决这些情况：一直阻塞等待直到其他线程修改队列的数据状态;阻塞一段时间之后返回，如果在这段时间内有其他线程修改队列数据，那么也会返回。

表 5：Queue 和 BlockingQueue 的方法：

方法	策略	插入	移除	核查
Queue	抛出异常	add	remove	element

	返回特定的值	offer	poll	peek
Blocking Queue	一直阻塞	put	take	n/a
	超时阻塞	offer	poll	n/a

在 JDK 中提供了一些 Queue 的实现，在表 6 中是这些实现类的关系列表。

方法	描述
PriorityQueue	PriorityQueue 是唯一一个非线程安全的队列实现类，用于单线程存放数据并且将数据排序。
CurrentLinkedQueue	一个无界的、基于链接列表的、唯一一个线程安全的队列实现类，不支持 BlockingQueue。
ArrayBlockingQueue	一个有界的、基于数组的阻塞队列。
LinkedBlockingQueue	一个有界的、基于链接列表的阻塞队列。有可能是最常用的队列实现。
PriorityBlockingQueue	一个无界的、基于堆的阻塞队列。队列根据设置的 Comparator（比较器）来确定组件读取、移除的顺序（不是队列默认的 FIFO 顺序）
DelayQueue	一个无界的、延迟元素（每个延迟元素都会有相应的延迟时间值）的阻塞队列实现。只有在延时期过了之后，元素才能被移除，而且最先被移除的是延时最先到期的元素。
SynchronousQueue	一种 0 容量的队列实现，生产者添加元素之后必须等待消费者移除后才可以返回，反之依然。如果生产者和消费者 2 个线程同时访问，那么参数直接从生产者传递到消费者。经常用于线程之间的数据传输。

4.4、Deque

在 JavaSE6 中新增加了两端都可以添加和删除的队列-Deque (发音"deck",not "dick"). Deques 不仅可以从一端添加元素，从另一端移除，而且两端都可以添加和删除元素。如同 BlockingQueue，BlockingDeque 接口也为阻塞等待和超时等待的特殊情况提供了解决方法。因为 Deque 继承 Queue、BlockingDeque 继承 BlockingQueue，下表中的方法都是可以使用的：

接口	头或尾	策略	插入	移除	核查
Queue	Head	抛出异常	addFirst	removeFirst	getFirst
		返回特定的值	offerFirst	pollFirst	peekFirst
	Tail	抛出异常	addLast	removeLast	getLast
		返回特定的值	offerLast	pollLast	peekLast
BlockingQueue	Head	一直阻塞	putFirst	takeFirst	n/a
		超时阻塞	offerFirst	pollFirst	n/a
	Tail	一直阻塞	putLast	takeLast	n/a
		超时阻塞	offerLast	pollLast	n/a

Deque 的一个特殊应用场景是只在一个端口进行添加、删除、检查操作--堆栈（first-in-last-out 顺序)。Deque 接口提供了 stack 相同的方法：push(), pop()和 peek()，这方法和 addFirst(), removeFirst(), peekFirst()一一对应，可以把 Deque 的任何一个实现类当做堆栈使用。表 6 中

是 JDK 中 Deque 和 BlockingDeque 的实现。注意 Deque 继承 Queue，BlockingDeque 继承自 BlockingQueue。

表 8：Deques

类	描述
LinkedList	这个经常被用到的类在 JavaSE6 中有了新的改进-实现了 Deque 接口。在 LinkedList 中，可以使用标准的 Deque 方法来添加或者删除 list 两端的元素。LinkedList 还可以被当做一个非同步的堆栈，用来替代同步的 Stack 类
ArrayDeque	一个非同步的、支持无限队列长度（根据需要动态扩展队列的长度）的 Deque 实现类
LinkedBlockingDeque	LinkeBlockingDeque 是 Deque 实现中唯一支持并发的、基于链接列表、队列长度可选的类。

5、线程

在 Java 中，java.lang.Thread 类是用来代表一个应用或者 JVM 线程。代码是在某个线程类的上下文环境中执行的（使用 Thread.currentThread()来获取当前运行的线程）。

5.1、线程通讯

线程之间最简单的通讯方式是一个线程直接调用另一个线程对象的方法。表 9 中列出的是线程之间可以直接交互的方法。

表 9：线程协作方法

线程方法	描述
start	启动一个线程实例，并且执行它的 run() 方法。
join	一直阻塞直到其他线程退出
interrupt	中断其他线程。线程如果在一个方法中被阻塞，会对 interrupt 操作做出回应，并在这个方法执行的线程中抛出 InterruptedException 异常；否则线程的中断状态被设定。
stop, suspend, resume, destroy	这些方法都被废弃，不应该再使用了。因为线程处理过程中状态问题会导致危险的操作。相反，应该使用 interrupt() 或者 volatile 标示来告诉一个线程应该做什么。

5.2、"未捕获异常"处理器

线程能够指定一个 UncaughtExceptionHandler 来接收任何一个导致线程非正常突然终止的未捕获异常的通知。

```
    Thread t = new Thread(runnable);
    t.setUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {
        public void uncaughtException(Thread t, Throwable
e) {
            // TODO get Logger and log uncaught exception
        }
    });
    t.start();
```

5.3、死锁

当存在多个线程（最少 2 个）等待对方占有的资源，就会形成资源循环依赖和线程等待，产生死锁。最常见的导致死锁的资源是对象 monitor，同时其他阻塞操作（例如 wait/notify）也能导致死锁。
很多新的 JVM 能够检测 Monitor 死锁，并且可以将线程 dump 中由信号（中断信号）、jstack 或者其他线程 dump 工具生成的死锁原因显示打印出来。
除了死锁，线程之间还会出现饥饿（starvation）和活锁(livelock). Starvation 是因为一个线程长时间占有一个锁导致其他的线程一直处于等待状态无法进行下一步操作。Livelock 是因为线程发费大量的时间来协调资源的访问或者检测避免死锁导致没有一个线程真正的干活。

6、线程协作

6.1、wait/notify

wait/notify 关键字适用于一个线程通知另一个线程所需的条件状态已就绪，最常用于线程在循环中休眠直到获取特定条件的场景. 例如，一个线程一直等待直到队列中有一个组件能够处理；当组件添加到队列时，另一个线程能够通知这个等待的线程。
wait 和 notify 的经典用法是：

```
public class Latch {
   private final Object lock = new Object();
   private volatile boolean flag = false;
   public void waitTillChange(){
      synchronized (lock) {
         while(!flag){
            try {
               lock.wait();
```

```
            } catch (InterruptedException e) {
            }
        }
    }
    public void change(){
        synchronized (lock) {
            flag = true;
            lock.notifyAll();
        }
    }
}
```

在代码中需要注意的重要地方是：

- wait、notify、notifyAll 必须在 synchronized 修饰的代码块中执行，否则会在运行的时候抛出 IllegalMonitorStateException 异常
- 在循环语句 wait 的时候一定要设定循环的条件--这样能够避免 wait 开始之前，线程所需的条件已经被其他线程提供了却依然开始此线程 wait 导致的时间消耗。同时，这种办法还能够保证你的代码不被虚假的信息唤醒。
- 总是要保证在调用 notify 和 notifyAll 之前，能够提供符合线程退出等待的条件。否则会出现即使线程接收到通知信息，却不能退出循环等待的情况。

6.2、Condition

在 JavaSE5 中新添加了 java.util.concurrent.locks.Condition 接口。Condition 不仅在 API 中实现了 wait/notify 语义，而且提供了几个新的特性，例如：为每个 Lock 创建多重 Condition、可中断的等待、访问统计信息等。Condition 是通过 Lock 示例产生的，示例：

```
public class LatchCondition {
   private final Lock lock = new ReentrantLock();
   private final Condition condition =
lock.newCondition();
   private volatile boolean flag = false;

   public void waitTillChange(){
      lock.lock();
      try{
         while(!flag){
            try {
               condition.await();
            } catch (InterruptedException e) {
            }
         }
      }finally{
         lock.unlock();
      }
   }
   public void change(){
      lock.lock();
      try{
         flag = true;
         condition.notifyAll();
      }finally{
         lock.unlock();
      }
   }
}
```

6.3、Coordination classes

java.util.concurrent 包中有几个类适用于常见的多线程通讯。这几个协作类适用范围几乎涵盖了使用 wait/notify 和 Condition 最常见的场景，而且更安全、更易于使用。

CyclicBarrier

在 CyclicBarrier 初始化的时候指定参与者的数量。参与者调用 awart()方法进入阻塞状态直到参与者的个数达到指定数量，此时最后一个到达的线程执行预定的屏障任务，然后释放所有的线程。屏障可以被重复的重置状态。常用于协调分组的线程的启动和停止。

CountDownLatch

需要指定一个计数才能初始化 CountDownLatch。线程调用 await()方法进入等待状态知道计

数变为 0。其他的线程（或者同一个线程）调用 countDown()来减少计数。如果计数变为 0 后是无法被重置的。常用于当确定数目的操作完成后，触发数量不定的线程。

Semaphore

Semaphore 维护一个“许可”集，能够使用 acquire()方法检测这个“许可”集，在“许可”可用之前 Semaphore 会阻塞每个 acquire 访问。线程能够调用 release()来返回一个许可。当 Semaphore 只有一个“许可”的时候，可当做一个互斥锁来使用。

Exchanger

线程在 Exchanger 的 exchange()方法上进行交互、原子操作的方式交换数据。功能类似于数据可以双向传递的 SynchronousQueue 加强版。

7、任务执行

很多 java 并发程序需要一个线程池来执行队列中的任务。在 java.util.concurrent 包中为这种类型的任务管理提供了一种可靠的基本方法。

7.1、ExecutorService

Executor 和易扩展的 ExecutorService 接口规定了用于执行任务的组件的标准。这些接口的使用者可以通过一个标准的接口使用各种具有不同行为的实现类。

最通用的 Executor 接口只能访问这种类型的可执行（Runnable）任务 ：
void execute(Runnable command)
Executor 子接口 ExecutorService 新加了方法，能够执行：Runnable 任务、Callable 任务以及任务集合。
Future<?> submit(Runnable task)
Future<T> submit(Callable<T> task)
Future<T> submit(Runnable task, T result)
List<Future<T>> invokeAll (Collection<? extends Callable<T>> tasks)
List<Future<T>> invokeAll (Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)
T invokeAny(Collection<? extends Callable<T>> tasks)
T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)

7.2、Callable and Future

Callable 类似于 Runnable，而且能够返回值、抛出异常：

- V call() throws Exception;

在一个任务执行框架中提交一个 Callable 任务，然后返回一个 Future 结果是很常见的。Future 表示在将来的某个时刻能够获取到结果。Future 提供能够获取结果或者阻塞直到结果可用的

方法。任务运行之前或正在运行的时候，可以通过 Future 中的方法取消。
如果只是需要一个 Runnable 特性的 Future（例如在 Executor 执行），可用使用 FutureTask。FutureTask 实现了 Future 和 Runnable 接口，可用提交一个 Runnable 类型任务，然后在调用部分使用这个 Future 类型的任务。

7.3、实现 ExecutorService

ExecutorService 最主要的实现类是 ThreadPoolExecutor。这个实现类提供了大量的可配置特性：

- 线程池--设定常用线程数量(启动前可选参数)和最大可用线程数量。
- 线程工厂--通过自定义的线程工厂生成线程，例如生成自定义线程名的线程。
- 工作队列--指定队列的实现类，实现类必须是阻塞的、可以是无界的或有界的。
- 被拒绝的任务--当队列已经满了或者是执行者不可用，需要为这些情况指定解决策略。
- 生命周期中的钩子--重写扩展在任务运行之前或之后的生命周期中的关键点
- 关闭--停止已接受的任务，等待正在运行的任务完成后，关闭 ThreadPoolExecutor。

ScheduledThreadPoolExecutor 是 ThreadPoolExecutor 的一个子类，能够按照定时的方式完成任务（而不是 FIFO 方式）。在 java.util.Timer 不是足够完善的情况下，ScheduleThreadPoolExecutor 具有强大的可适用性。

Executors 类有很多静态方法（表 10）用于创建适用于各种常见情况的预先包装的 ExecutorService 和 ScheduleExecutorService 实例

表 10

方法	描述
newSingleThreadExecutor	创建只有一个线程的 ExecutorService
newFixedThreadPool	返回拥有固定数量线程的 ExecutorService
newCachedThreadPool	返回一个线程数量可变的 ExecutorService
newSingleThreadScheduledExecutor	返回只有一个线程的 ScheduledExecutorService
newScheduledThreadPool	创建拥有一组核心线程的 ScheduledExecutorService

下面的例子是创建一个固定线程池，然后提交一个长期运行的任务：

```
      int processors =
Runtime.getRuntime().availableProcessors();
      ExecutorService executor =
Executors.newFixedThreadPool(processors);
      Future<Integer> futureResult =
executor.submit(new Callable<Integer>() {
         public Integer call() {
            //long time computation that returns an
Integer
            return null;
         }
      });
      Integer result = futureResult.get();
```

在这个示例中提交任务到 executor 之后，代码没有阻塞而是立即返回。在代码的最后一行调用 get()方法会阻塞直到有结果返回。

ExecutorService 几乎涵盖了所有应该创建线程对象或线程池的情景。在代码中需要直接创建一个线程的时候，可以考虑通过 Executor 工厂创建的 ExecutorService 能否实现相同的目标；这样做经常更简单、更灵活。

7.4、CompletionService

除了常见的线程池和输入队列模式，还有一种常见的情况：为后面的处理，每个任务生成的结果必须积累下来。CompletionService 接口允许提交 Callable 和 Runnable 任务，而且还可以从任务队列中获取这些结果：（绿色部分和英文版不一样，已和作者确认，英文版将 take()和 poll()方法混淆了）

- Future<V> take () -- 如果结果存在则获取，否则直接返回
- Future<V> poll () -- 阻塞直到结果可用
- Future<V> poll (long timeout, TimeUnit unit) -- 阻塞直到 timeout 时间结束

ExecutorCompletionService 是 CompletionService 的标准实现类。在 ExecutorCompletionService 的构成函数中需要一个 Executor，ExecutorCompletionService 提供输入队列和线程池。

8、Hot Tip

热门信息：当设置线程池大小的时候，最好是基于当前应用所运行的机器拥有的逻辑处理器的数量。在 java 中，可用使用 Runtime.getRuntime().availableProcessors()获取这个值。在 JVM 的生命周期中，可用处理器的数目是可变的。

9、关于作者

Alex Miller 是 Terracotta Inc 公司 Java 集群开源产品的技术负责人，曾在 BEA System 和 MetaMatrix 工作，是 MetaMatrix 的首席架构师。他对 Java、并发、分布式系统、查询语言和软件设计感兴趣。他的 tweeter：@puredanger，blog：http://tect.puredanger.com，很喜欢在用户组会议中发言。在 St. Louis，Alex 是 Lambda Lounge 小组的创建人，Lambda Lounge 用户组是为了学习、动态语言、Strange Loop 开发会议而创建的。

10、翻译后记

开始阅读英文版的时候，并没有觉得文章中有什么晦涩的地方。但是在翻译之后，才发现将文中的意思清楚地表达出来也是个脑力活，有时一句话能够懂得意思，却是很难用汉语表达出来：“只可意会，不可言传”--这也能解释我当年高中作文为啥每次只能拿 40 分（总分 60）。在禅宗，师傅教弟子佛理，多靠弟子自身的明悟，故有当头棒喝、醍醐灌顶之说。做翻译却不能这样，总不能让读者对着满篇的鸟文去琢磨明悟吧，须得直译、意译并用，梳理文字。

翻译也是一个学习的过程。阅读本文的时候会无意忽略自己以为不重要的词句，待到真正翻译的时候，才发现自己一知半解、一窍不通，就只好 Google 之，翻译完成后，也学了些知识，可谓是一箭双雕。

个人精力所限，翻译中难免有不对的地方，望大家予以指正。

联系方式：MSN/E-Mail:nathanlhb@hotmail.com
Blog:http://blog.csdn.net/liu251