java.util.concurrent介绍

笔记本： 并发控制，锁以及线程池专题研究
创建时间： 2015/4/26 23:16 更新时间： 2015/4/26 23:17
URL： http://www.cnblogs.com/sarafill/archive/2011/05/18/2049461.html

java.util.concurrent 包含许多线程安全、测试良好、高性能的并发构建块。不客气地说，创建 java.util.concurrent 的目的就是要实现 Collection 框架对数据结构所执行的并发操作。通过提供一组可靠的、高性能并发构建块，开发人员可以提高并发类的线程安全、可伸缩性、性能、可读性和可靠性。

如果一些类名看起来相似，可能是因为 java.util.concurrent 中的许多概念源自 Doug Lea 的 util.concurrent 库（请参阅 参考资料）。

JDK 5.0 中的并发改进可以分为三组：

- **JVM 级别更改**。大多数现代处理器对并发对某一硬件级别提供支持，通常以 compare-and-swap （CAS）指令形式。CAS 是一种低级别的、细粒度的技术，它允许多个线程更新一个内存位置，同时能够检测其他线程的冲突并进行恢复。它是许多高性能并发算法的基础。在 JDK 5.0 之前，Java 语言中用于协调线程之间的访问的惟一原语是同步，同步是更重量级和粗粒度的。公开 CAS 可以开发高度可伸缩的并发 Java 类。这些更改主要由 JDK 库类使用，而不是由开发人员使用。

- **低级实用程序类 -- 锁定和原子类**。使用 CAS 作为并发原语，ReentrantLock 类提供与 synchronized 原语相同的锁定和内存语义，然而这样可以更好地控制锁定（如计时的锁定等待、锁定轮询和可中断的锁定等待）和提供更好的可伸缩性（竞争时的高性能）。大多数开发人员将不再直接使用 ReentrantLock 类，而是使用在 ReentrantLock 类上构建的高级类。

- **高级实用程序类**。这些类实现并发构建块，每个计算机科学文本中都会讲述这些类 -- 信号、互斥、闩锁、屏障、交换程序、线程池和线程安全集合类等。大部分开发人员都可以在应用程序中用这些类，来替换许多（如果不是全部）同步、wait() 和 notify() 的使用，从而提高性能、可读性和正确性。

本教程将重点介绍 java.util.concurrent 包提供的高级实用程序类 -- 线程安全集合、线程池和同步实用程序。这些是初学者和专家都可以使用的"现成"类。

在第一小节中，我们将回顾并发的基本知识，尽管它不应取代对线程和线程安全的了解。那些一点都不熟悉线程的读者应该先参考一些关于线程的介绍，如"Introduction to Java Threads"教程（请参阅参考资料）。

接下来的几个小节将研究 java.util.concurrent 中的高级实用程序类 -- 线程安全集合、线程池、信号和同步工具。

最后一小节将介绍 java.util.concurrent 中的低级并发构建块，并提供一些性能测评来显示新 java.util.concurrent 类的可伸缩性的改进。

什么是线程？

所有重要的操作系统都支持进程的概念 -- 独立运行的程序，在某种程度上相互隔离。

线程有时称为 轻量级进程。与进程一样，它们拥有通过程序运行的独立的并发路径，并且每个线程都有自己的程序计数器，称为堆栈和本地变量。然而，线程存在于进程中，它们与同一进程内的其他线程共享内存、文件句柄以及每进程状态。

今天，几乎每个操作系统都支持线程，允许执行多个可独立调度的线程，以便共存于一个进程中。因为一个进程中的线程是在同一个地址空间中执行的，所以多个线程可以同时访问相同对象，并且它们从同一堆栈中分配对象。虽然这使线程更易于与其他线程共享信息，但也意味着您必须确保线程之间不相互干涉。

正确使用线程时，线程能带来诸多好处，其中包括更好的资源利用、简化开发、高吞吐量、更易响应的用户界面以及能执行异步处理。

Java 语言包括用于协调线程行为的原语，从而可以在不违反设计原型或者不破坏数据结构的前提下安全地访问和修改共享变量。

线程有哪些功能？

在 Java 程序中存在很多理由使用线程，并且不管开发人员知道线程与否，几乎每个 Java 应用程序都使用线程。许多 J2SE 和 J2EE 工具可以创建线程，如 RMI、Servlet、Enterprise JavaBeans 组件和 Swing GUI 工具包。

使用线程的理由包括：

- **更易响应的用户界面。** 事件驱动的 GUI 工具包（如 AWT 或 Swing）使用单独的事件线程来处理 GUI 事件。从事件线程中调用通过 GUI 对象注册的事件监听器。然而，如果事件监听器将执行冗长的任务（如文档拼写检查），那么 UI 将出现冻结，因为事件线程直到冗长任务完毕之后才能处理其他事件。通过在单独线程中执行冗长操作，当执行冗长后台任务时，UI 能继续响应。

- **使用多处理器。** 多处理器（MP）系统变得越来越便宜，并且分布越来越广泛。因为调度的基本单位通常是线程，所以不管有多少处理器可用，一个线程的应用程序一次只能在一个处理器上运行。在设计良好的程序中，通过更好地利用可用的计算机资源，多线程能够提高吞吐量和性能。

- **简化建模。** 有效使用线程能够使程序编写变得更简单，并易于维护。通过合理使用线程，个别类可以避免一些调度的详细、交叉存取操作、异步 IO 和资源等待以及其他复杂问题。相反，它们能专注于域的要求，简化开发并改进可靠性。

- **异步或后台处理。** 服务器应用程序可以同时服务于许多远程客户机。如果应用程序从 socket 中读取数据，并且没有数据可以读取，那么对 read() 的调用将被阻塞，直到有数据可读。在单线程应用程序中，这意味着当某一个线程被阻塞时，不仅处理相应请求要延迟，而且处理所有请求也将延迟。然而，如果每个 socket 都有自己的 IO 线程，那么当一个线程被阻塞时，对其他并发请求行为没有影响。

线程安全

如果将这些类用于多线程环境中，虽然确保这些类的线程安全比较困难，但线程安全却是必需的。java.util.concurrent 规范进程的一个目标就是提供一组线程安全的、高性能的并发构建块，从而使开发人员能够减轻一些编写线程安全类的负担。

线程安全类非常难以明确定义，大多数定义似乎都是完全循环的。快速 Google 搜索会显示下列线程安全代码定义的例子，但这些定义（或者更确切地说是描述）通常没什么帮助：

- . . . can be called from multiple programming threads without unwanted interaction between the threads.
- . . . may be called by more than on thread at a time without requiring any other action on the caller's part.

通过类似这样的定义，不奇怪我们为什么对线程安全如此迷惑。这些定义几乎就是在说"如果可以从多个线程安全调用类，那么该类就是线程安全的"。这当然是线程安全的解释，但对我们区别线程安全类和不安全类没有什么帮助。我们使用"安全"是为了说明什么？

要成为线程安全的类，首先它必须在单线程环境中正确运行。如果正确实现了类，那么说明它符合规范，对该类的对象的任何顺序的操作（公共字段的读写、公共方法的调用）都不应该使对象处于无效状态；观察将处于无效状态的对象；或违反类的任何变量、前置条件或后置条件。

而且，要成为线程安全的类，在从多个线程访问时，它必须继续正确运行，而不管运行时环境执行那些线程的调度和交叉，且无需对部分调用代码执行任何其他同步。结果是对线程安全对象的操作将用于按固定的整体一致顺序出现所有线程。

如果没有线程之间的某种明确协调，比如锁定，运行时可以随意在需要时在多线程中交叉操作执行。

在 JDK 5.0 之前，确保线程安全的主要机制是 synchronized 原语。访问共享变量（那些可以由多个线程访问的变量）的线程必须使用同步来协调对共享变量的读写访问。java.util.concurrent 包提供了一些备用并发原语，以及一组不需要任何其他同步的线程安全实用程序类。

令人厌烦的并发

即使您的程序从没有明确创建线程，也可能会有许多工具或框架代表您创建了线程，这时要

求从这些线程调用的类是线程安全的。这样会对开发人员带来较大的设计和实现负担，因为开发线程安全类比开发非线程安全类有更多要注意的事项，且需要更多的分析。

AWT 和 Swing

这些 GUI 工具包创建了称为时间线程的后台线程，将从该线程调用通过 GUI 组件注册的监听器。因此，实现这些监听器的类必须是线程安全的。

TimerTask

JDK 1.3 中引入的 TimerTask 工具允许稍后执行任务或计划定期执行任务。在 Timer 线程中执行 TimerTask 事件，这意味着作为 TimerTask 执行的任务必须是线程安全的。

Servlet 和 JavaServer Page 技术

Servlet 容器可以创建多个线程，在多个线程中同时调用给定 servlet，从而进行多个请求。因此 servlet 类必须是线程安全的。

RMI

远程方法调用（remote method invocation，RMI）工具允许调用其他 JVM 中运行的操作。实现远程对象最普遍的方法是扩展 UnicastRemoteObject。例示 UnicastRemoteObject 时，它是通过 RMI 调度器注册的，该调度器可能创建一个或多个线程，将在这些线程中执行远程方法。因此，远程类必须是线程安全的。

正如所看到的，即使应用程序没有明确创建线程，也会发生许多可能会从其他线程调用类的情况。幸运的是，java.util.concurrent 中的类可以大大简化编写线程安全类的任务。

例子 -- 非线程安全 servlet

下列 servlet 看起来像无害的留言板 servlet，它保存每个来访者的姓名。然而，该 servlet 不是线程安全的，而这个 servlet 应该是线程安全的。问题在于它使用 HashSet 存储来访者的姓名，HashSet 不是线程安全的类。

当我们说这个 servlet 不是线程安全的时，是说它所造成的破坏不仅仅是丢失留言板输入。在最坏的情况下，留言板数据结构都可能被破坏并且无法恢复。

```
public class UnsafeGuestbookServlet extends HttpServlet {

    private Set visitorSet = new HashSet();

    protected void doGet(HttpServletRequest httpServletRequest,

            HttpServletResponse httpServletResponse) throws ServletException,
IOException {

        String visitorName = httpServletRequest.getParameter("NAME");
```

```
        if (visitorName != null)
            visitorSet.add(visitorName);
    }
}
```

通过将 visitorSet 的定义更改为下列代码，可以使该类变为线程安全的：

```
private Set visitorSet = Collections.synchronizedSet(new HashSet());
```

如上所示的例子显示线程的内置支持是一把双刃剑 -- 虽然它使构建多线程应用程序变得很容易，但它同时要求开发人员更加注意并发问题，甚至在使用留言板 servlet 这样普通的东西时也是如此。

线程安全集合

JDK 1.2 中引入的 Collection 框架是一种表示对象集合的高度灵活的框架，它使用基本接口 List、Set 和 Map。通过 JDK 提供每个集合的多次实现（HashMap、Hashtable、TreeMap、WeakHashMap、HashSet、TreeSet、Vector、ArrayList、LinkedList 等等）。其中一些集合已经是线程安全的（Hashtable 和 Vector），通过同步的封装工厂（Collections.synchronizedMap()、synchronizedList() 和 synchronizedSet()），其余的集合均可表现为线程安全的。

java.util.concurrent 包添加了多个新的线程安全集合类（ConcurrentHashMap、CopyOnWriteArrayList 和 CopyOnWriteArraySet）。这些类的目的是提供高性能、高度可伸缩性、线程安全的基本集合类型版本。

java.util 中的线程集合仍有一些缺点。例如，在迭代锁定时，通常需要将该锁定保留在集合中，否则，会有抛出 ConcurrentModificationException 的危险。（这个特性有时称为条件线程安全；有关的更多说明，请参阅参考资料。）此外，如果从多个线程频繁地访问集合，则常常不能很好地执行这些类。java.util.concurrent 中的新集合类允许通过在语义中的少量更改来获得更高的并发。

JDK 5.0 还提供了两个新集合接口 -- Queue 和 BlockingQueue。Queue 接口与 List 类似，但它只允许从后面插入，从前面删除。通过消除 List 的随机访问要求，可以创建比现有 ArrayList 和 LinkedList 实现性能更好的 Queue 实现。因为 List 的许多应用程序实际上不需要随机访问，所以Queue 通常可以替代 List，来获得更好的性能。

弱一致的迭代器

java.util 包中的集合类都返回 fail-fast 迭代器，这意味着它们假设线程在集合内容中进行迭代时，集合不会更改它的内容。如果 fail-fast 迭代器检测到在迭代过程中进行了更改操作，那么它会抛出 ConcurrentModificationException，这是不可控异常。

在迭代过程中不更改集合的要求通常会对许多并发应用程序造成不便。相反，比较好的是它允许并发修改并确保迭代器只要进行合理操作，就可以提供集合的一致视图，如 java.util.concurrent 集合类中的迭代器所做的那样。

java.util.concurrent 集合返回的迭代器称为弱一致的（weakly consistent）迭代器。对于这些类，如果元素自从迭代开始已经删除，且尚未由 next() 方法返回，那么它将不返回到调用者。如果元素自迭代开始已经添加，那么它可能返回调用者，也可能不返回。在一次迭代中，无论如何更改底层集合，元素不会被返回两次。

CopyOnWriteArrayList 和 CopyOnWriteArraySet

可以用两种方法创建线程安全支持数据的 List -- Vector 或封装 ArrayList 和 Collections.synchronizedList()。java.util.concurrent 包添加了名称繁琐的 CopyOnWriteArrayList。为什么我们想要新的线程安全的List类？为什么Vector还不够？

最简单的答案是与迭代和并发修改之间的交互有关。使用 Vector 或使用同步的 List 封装器，返回的迭代器是 fail-fast 的，这意味着如果在迭代过程中任何其他线程修改 List，迭代可能失败。

Vector 的非常普遍的应用程序是存储通过组件注册的监听器的列表。当发生适合的事件时，该组件将在监听器的列表中迭代，调用每个监听器。为了防止 ConcurrentModificationException，迭代线程必须复制列表或锁定列表，以便进行整体迭代，而这两种情况都需要大量的性能成本。

CopyOnWriteArrayList 类通过每次添加或删除元素时创建支持数组的新副本，避免了这个问题，但是进行中的迭代保持对创建迭代器时的当前副本进行操作。虽然复制也会有一些成本，但是在许多情况下，迭代要比修改多得多，在这些情况下，写入时复制要比其他备用方法具有更好的性能和并发性。

如果应用程序需要 Set 语义，而不是 List，那么还有一个 Set 版本 -- CopyOnWriteArraySet。

ConcurrentHashMap

正如已经存在线程安全的 List 的实现，您可以用多种方法创建线程安全的、基于 hash 的 Map -- Hashtable，并使用 Collections.synchronizedMap() 封装 HashMap。JDK 5.0 添加了 ConcurrentHashMap 实现，该实现提供了相同的基本线程安全的 Map 功能，但它大大提高了并发性。

Hashtable 和 synchronizedMap 所采取的获得同步的简单方法（同步 Hashtable 中或者同步的 Map 封装器对象中的每个方法）有两个主要的不足。首先，这种方法对于可伸缩性是一种障碍，因为一次只能有一个线程可以访问 hash 表。同时，这样仍不足以提供真正的线程安全性，许多公用的混合操作仍然需要额外的同步。虽然诸如 get() 和 put() 之类的简单操作

可以在不需要额外同步的情况下安全地完成，但还是有一些公用的操作序列，例如迭代或者 put-if-absent（空则放入），需要外部的同步，以避免数据争用。

Hashtable 和 Collections.synchronizedMap 通过同步每个方法获得线程安全。这意味着当一个线程执行一个 Map 方法时，无论其他线程要对 Map 进行什么样操作，都不能执行，直到第一个线程结束才可以。

对比来说，ConcurrentHashMap 允许多个读取几乎总是并发执行，读和写操作通常并发执行，多个同时写入经常并发执行。结果是当多个线程需要访问同一 Map 时，可以获得更高的并发性。

在大多数情况下，ConcurrentHashMap 是 Hashtable或 Collections.synchronizedMap(new HashMap()) 的简单替换。然而，其中有一个显著不同，即 ConcurrentHashMap 实例中的同步不锁定映射进行独占使用。实际上，没有办法锁定 ConcurrentHashMap 进行独占使用，它被设计用于进行并发访问。为了使集合不被锁定进行独占使用，还提供了公用的混合操作的其他（原子）方法，如 put-if-absent。ConcurrentHashMap 返回的迭代器是弱一致的，意味着它们将不抛出 ConcurrentModificationException ，将进行"合理操作"来反映迭代过程中其他线程对 Map 的修改。

队列

原始集合框架包含三个接口：List、Map 和 Set。List 描述了元素的有序集合，支持完全随即访问 -- 可以在任何位置添加、提取或删除元素。

LinkedList 类经常用于存储工作元素（等待执行的任务）的列表或队列。然而，List 提供的灵活性比该公用应用程序所需要的多得多，这个应用程序通常在后面插入元素，从前面删除元素。但是要支持完整 List 接口则意味着 LinkedList 对于这项任务不像原来那样有效。Queue 接口比 List 简单得多，仅包含 put() 和 take() 方法，并允许比 LinkedList 更有效的实现。

Queue 接口还允许实现来确定存储元素的顺序。ConcurrentLinkedQueue 类实现先进先出（first-in-first-out，FIFO）队列，而 PriorityQueue 类实现优先级队列（也称为堆），它对于构建调度器非常有用，调度器必须按优先级或预期的执行时间执行任务。

```
interface Queue extends Collection {

    boolean offer(E x);

    E poll();

    E remove() throws NoSuchElementException;

    E peek();

    E element() throws NoSuchElementException;
```

}

实现 Queue 的类是：

- LinkedList 已经进行了改进来实现 Queue。
- PriorityQueue 非线程安全的优先级对列（堆）实现，根据自然顺序或比较器返回元素。
- ConcurrentLinkedQueue 快速、线程安全的、无阻塞 FIFO 队列。

任务管理之线程创建

线程最普遍的一个应用程序是创建一个或多个线程，以执行特定类型的任务。Timer 类创建线程来执行 TimerTask 对象，Swing 创建线程来处理 UI 事件。在这两种情况中，在单独线程中执行的任务都假定是短期的，这些线程是为了处理大量短期任务而存在的。

在其中每种情况中，这些线程一般都有非常简单的结构：

```
while (true) {
  if (no tasks)
    wait for a task;
  execute the task;
}
```

通过例示从 Thread 获得的对象并调用 Thread.start() 方法来创建线程。可以用两种方法创建线程：通过扩展 Thread 和覆盖 run() 方法，或者通过实现 Runnable 接口和使用 Thread(Runnable) 构造函数：

```
class WorkerThread extends Thread {
  public void run() { /* do work */ }
}
Thread t = new WorkerThread();
t.start();
```

或者：

```
Thread t = new Thread(new Runnable() {
  public void run() { /* do work */ }
}
```

t.start();

重新使用线程

因为多个原因，类似 Swing GUI 的框架为事件任务创建单一线程，而不是为每项任务创建新的线程。首先是因为创建线程会有间接成本，所以创建线程来执行简单任务将是一种资源浪费。通过重新使用事件线程来处理多个事件，启动和拆卸成本（随平台而变）会分摊在多个事件上。

Swing 为事件使用单一后台线程的另一个原因是确保事件不会互相干涉，因为直到前一事件结束，下一事件才开始处理。该方法简化了事件处理程序的编写。

使用多个线程，将要做更多的工作来确保一次仅一个线程地执行线程相关的代码。

如何不对任务进行管理

大多数服务器应用程序（如 Web 服务器、POP 服务器、数据库服务器或文件服务器）代表远程客户机处理请求，这些客户机通常使用 socket 连接到服务器。对于每个请求，通常要进行少量处理（获得该文件的代码块，并将其发送回 socket），但是可能会有大量（且不受限制）的客户机请求服务。

用于构建服务器应用程序的简单化模型会为每个请求创建新的线程。下列代码段实现简单的 Web 服务器，它接受端口 80 的 socket 连接，并创建新的线程来处理请求。不幸的是，该代码不是实现 Web 服务器的好方法，因为在重负载条件下它将失败，停止整台服务器。

```
class UnreliableWebServer {
  public static void main(String[] args) {
    ServerSocket socket = new ServerSocket(80);
      while (true) {
      final Socket connection = socket.accept();
      Runnable r = new Runnable() {
        public void run() {
          handleRequest(connection);
        }
      };
      // Don't do this!
      new Thread(r).start();
```

```
    }
  }
}
```

当服务器被请求吞没时，UnreliableWebServer 类不能很好地处理这种情况。每次有请求时，就会创建新的类。根据操作系统和可用内存，可以创建的线程数是有限的。

不幸的是，您通常不知道限制是多少 -- 只有当应用程序因为 OutOfMemoryError 而崩溃时才发现。

如果足够快地在这台服务器上抛出请求的话，最终其中一个线程创建将失败，生成的 Error 会关闭整个应用程序。当一次仅能有效支持很少线程时，没有必要创建上千个

线程，无论如何，这样使用资源可能会损害性能。创建线程会使用相当一部分内存，其中包括有两个堆栈（Java 和 C），以及每线程数据结构。如果创建过多线程，其中

每个线程都将占用一些 CPU 时间，结果将使用许多内存来支持大量线程，每个线程都运行得很慢。这样就无法很好地使用计算资源。

使用线程池解决问题

为任务创建新的线程并不一定不好，但是如果创建任务的频率高，而平均任务持续时间低，我们可以看到每项任务创建一个新的线程将产生性能（如果负载不可预知，还有稳定性）问题。

如果不是每项任务创建一个新的线程，则服务器应用程序必须采取一些方法来限制一次可以处理的请求数。这意味着每次需要启动新的任务时，它不能仅调用下列代码。

```
new Thread(runnable).start()
```

管理一大组小任务的标准机制是组合工作队列和线程池。工作队列就是要处理的任务的队列，前面描述的 Queue 类完全适合。线程池是线程的集合，每个线程都提取公用工作队列。当一个工作线程完成任务处理后，它会返回队列，查看是否有其他任务需要处理。如果有，它会转移到下一个任务，并开始处理。

线程池为线程生命周期间接成本问题和资源崩溃问题提供了解决方案。通过对多个任务重新使用线程，创建线程的间接成本将分布到多个任务中。作为一种额外好处，因为请求到达时，线程已经存在，从而可以消除由创建线程引起的延迟。因此，可以立即处理请求，使应用程序更易响应。而且，通过正确调整线程池中的线程数，可以强制超出特定限制的任何请求等待，直到有线程可以处理它，它们等待时所消耗的资源要少于使用额外线程所消耗的资源，这样可以防止资源崩溃。

Executor 框架

java.util.concurrent 包中包含灵活的线程池实现，但是更重要的是，它包含用于管理实现

Runnable 的任务的执行的整个框架。该框架称为 Executor 框架。

Executor 接口相当简单。它描述将运行 Runnable 的对象：

```
public interface Executor {

  void execute(Runnable command);

}
```

任务运行于哪个线程不是由该接口指定的，这取决于使用的 Executor 的实现。它可以运行于后台线程，如 Swing 事件线程，或者运行于线程池，或者调用线程，或者新的线程，它甚至可以运行于其他 JVM！通过同步的 Executor 接口提交任务，从任务执行策略中删除任务提交。Executor 接口独自关注任务提交 -- 这是Executor 实现的选择，确定执行策略。这使在部署时调整执行策略（队列限制、池大小、优先级排列等等）更加容易，更改的代码最少。

java.util.concurrent 中的大多数 Executor 实现还实现 ExecutorService 接口，这是对 Executor 的扩展，它还管理执行服务的生命周期。这使它们更易于管理，并向生命可能比单独 Executor 的生命更长的应用程序提供服务。

```
public interface ExecutorService extends Executor {

  void shutdown();

  List shutdownNow();

  boolean isShutdown();

  boolean isTerminated();

  boolean awaitTermination(long timeout,

                           TimeUnit unit);

  // other convenience methods for submitting tasks

}
```

Executor

java.util.concurrent 包包含多个 Executor 实现，每个实现都实现不同的执行策略。什么是执行策略？执行策略定义何时在哪个线程中运行任务，执行任务可能消耗的资源级别（线程、内存等等），以及如果执行程序超载该怎么办。

执行程序通常通过工厂方法例示，而不是通过构造函数。Executors 类包含用于构造许多不同类型的 Executor 实现的静态工厂方法：

- Executors.newCachedThreadPool() 创建不限制大小的线程池，但是当以前创建的线

程可以使用时将重新使用那些线程。如果没有现有线程可用，

- 将创建新的线程并将其添加到池中。使用不到 60 秒的线程将终止并从缓存中删除。

- Executors.newFixedThreadPool(int n) 创建线程池，其重新使用在不受限制的队列之外运行的固定线程组。在关闭前，所有线程都会因为执行

- 过程中的失败而终止，如果需要执行后续任务，将会有新的线程来代替这些线程。

- Executors.newSingleThreadExecutor() 创建 Executor，其使用在不受限制的队列之外运行的单一工作线程，与 Swing 事件线程非常相似。

- 保证顺序执行任务，在任何给定时间，不会有多个任务处于活动状态。

更可靠的 Web 服务器 -- 使用 Executor

前面 如何不对任务进行管理 中的代码显示了如何不用编写可靠服务器应用程序。幸运的是，修复这个示例非常简单，只需将 Thread.start() 调用替换为向 Executor 提交任务即可：

```
class ReliableWebServer {
  Executor pool =
    Executors.newFixedThreadPool(7);
    public static void main(String[] args) {
    ServerSocket socket = new ServerSocket(80);
      while (true) {
      final Socket connection = socket.accept();
      Runnable r = new Runnable() {
        public void run() {
          handleRequest(connection);
        }
      };
      pool.execute(r);
    }
  }
```

}

注意，本例与前例之间的区别仅在于 Executor 的创建以及如何提交执行的任务。

定制 ThreadPoolExecutor

Executors 中的 newFixedThreadPool 和 newCachedThreadPool 工厂方法返回的 Executor 是类 ThreadPoolExecutor 的实例，是高度可定制的。

通过使用包含 ThreadFactory 变量的工厂方法或构造函数的版本，可以定义池线程的创建。ThreadFactory 是工厂对象，其构造执行程序要使用的新线程。

使用定制的线程工厂，创建的线程可以包含有用的线程名称，并且这些线程是守护线程，属于特定线程组或具有特定优先级。

下面是线程工厂的例子，它创建守护线程，而不是创建用户线程：

```
public class DaemonThreadFactory implements ThreadFactory {
    public Thread newThread(Runnable r) {
        Thread thread = new Thread(r);
        thread.setDaemon(true);
        return thread;
    }
}
```

有时，Executor 不能执行任务，因为它已经关闭或者因为 Executor 使用受限制队列存储等待任务，而该队列已满。在这种情况下，需要咨询执行程序的 RejectedExecutionHandler 来确定如何处理任务 -- 抛出异常（默认情况），放弃任务，在调用者的线程中执行任务，或放弃队列中最早的任务以为新任务腾出空间。
ThreadPoolExecutor.setRejectedExecutionHandler 可以设置拒绝的执行处理程序。

还可以扩展 ThreadPoolExecutor，并覆盖方法 beforeExecute 和 afterExecute，以添加装置，添加记录，添加计时，重新初始化线程本地变量，或进行其他执行定制。

需要特别考虑的问题

使用 Executor 框架会从执行策略中删除任务提交，一般情况下，人们希望这样，那是因为它允许我们灵活地调整执行策略，不必更改许多位置的代码。然而，当提交代码暗含假设特定执行策略时，存在多种情况，在这些情况下，重要的是选择的 Executor 实现一致的执行策略。

这类情况中的其中的一种就是一些任务同时等待其他任务完成。在这种情况下，当线程池没有足够的线程时，如果所有当前执行的任务都在等待另一项任务，而该任务因为线程池已满不能

执行，那么线程池可能会死锁。

另一种相似的情况是一组线程必须作为共同操作组一起工作。在这种情况下，需要确保线程池能够容纳所有线程。

如果应用程序对特定执行程序进行了特定假设，那么应该在 Executor 定义和初始化的附近对这些进行说明，从而使善意的更改不会破坏应用程序的正确功能。

调整线程池

创建 Executor 时，人们普遍会问的一个问题是"线程池应该有多大？"。当然，答案取决于硬件和将执行的任务类型（它们是受计算限制或是受 IO 的限制？）。

如果线程池太小，资源可能不能被充分利用，在一些任务还在工作队列中等待执行时，可能会有处理器处于闲置状态。

另一方面，如果线程池太大，则将有许多有效线程，因为大量线程或有效任务使用内存，或者因为每项任务要比使用少量线程有更多上下文切换，性能可能会受损。

所以假设为了使处理器得到充分使用，线程池应该有多大？如果知道系统有多少处理器和任务的计算时间和等待时间的近似比率，Amdahl 法则提供很好的近似公式。

用 WT 表示每项任务的平均等待时间，ST 表示每项任务的平均服务时间（计算时间）。则 WT/ST 是每项任务等待所用时间的百分比。对于 N 处理器系统，池中可以近似有 N*(1+WT/ST) 个线程。

好的消息是您不必精确估计 WT/ST。"合适的"池大小的范围相当大；只需要避免"过大"和"过小"的极端情况即可。

Future 接口

Future 接口允许表示已经完成的任务、正在执行过程中的任务或者尚未开始执行的任务。通过 Future 接口，可以尝试取消尚未完成的任务，查询任务已经完成还是取消了，以及提取（或等待）任务的结果值。

FutureTask 类实现了 Future，并包含一些构造函数，允许将 Runnable 或 Callable（会产生结果的 Runnable）和 Future 接口封装。因为 FutureTask 也实现 Runnable，所以可以只将 FutureTask 提供给 Executor。一些提交方法（如 ExecutorService.submit()）除了提交任务之外，还将返回 Future 接口。

Future.get() 方法检索任务计算的结果（或如果任务完成，但有异常，则抛出 ExecutionException）。如果任务尚未完成，那么 Future.get() 将被阻塞，直到任务完成；如果任务已经完成，那么它将立即返回结果。

使用 Future 构建缓存

该示例代码与 java.util.concurrent 中的多个类关联，突出显示了 Future 的功能。它实现缓存，使用 Future 描述缓存值，该值可能已经计算，或者可能在其他线程中"正在构造"。

它利用 ConcurrentHashMap 中的原子 putIfAbsent() 方法，确保仅有一个线程试图计算给定关键字的值。如果其他线程随后请求同一关键字的值，它仅能等待（通过 Future.get() 的帮助）第一个线程完成。因此两个线程不会计算相同的值。

```
public class Cache {
   ConcurrentMap> map = new ConcurrentHashMap();
   Executor executor = Executors.newFixedThreadPool(8);
   public V get(final K key) {
      FutureTask f = map.get(key);
      if (f == null) {
         Callable c = new Callable() {
            public V call() {
               // return value associated with key
            }
         };
         f = new FutureTask(c);
         FutureTask old = map.putIfAbsent(key, f);
         if (old == null)
            executor.execute(f);
         else
            f = old;
      }
      return f.get();
   }
}
```

CompletionService

CompletionService 将执行服务与类似 Queue 的接口组合，从任务执行中删除任务结果的处理。CompletionService 接口包含用来提交将要执行的任务的 submit() 方法和用来询问下一完成任务的 take()/poll() 方法。

CompletionService 允许应用程序结构化，使用 Producer/Consumer 模式，其中生产者创建任务并提交，消费者请求完成任务的结果并处理这些结果。CompletionService 接口由 ExecutorCompletionService 类实现，该类使用 Executor 处理任务并从 CompletionService 导出 submit/poll/take 方法。

下列代码使用 Executor 和 CompletionService 来启动许多"solver"任务，并使用第一个生成非空结果的任务的结果，然后取消其余任务：

```
void solve(Executor e, Collection> solvers)
    throws InterruptedException {
      CompletionService ecs = new ExecutorCompletionService(e);
      int n = solvers.size();
      List> futures = new ArrayList>(n);
      Result result = null;
      try {
          for (Callable s : solvers)
              futures.add(ecs.submit(s));
          for (int i = 0; i < n; ++i) {
              try {
                  Result r = ecs.take().get();
                  if (r != null) {
                      result = r;
                      break;
                  }
              } catch(ExecutionException ignore) {}
          }
      }
```

```
        finally {
            for (Future f : futures)
                f.cancel(true);
        }
        if (result != null)
            use(result);
    }
```

java.util.concurrent 中其他类别的有用的类也是同步工具。这组类相互协作，控制一个或多个线程的执行流。

Semaphore、CyclicBarrier、CountdownLatch 和 Exchanger 类都是同步工具的例子。每个类都有线程可以调用的方法，方法是否被阻塞取决于正在使用的特定同步工具的状态和规则。

Semaphore

Semaphore 类实现标准 Dijkstra 计数信号。计数信号可以认为具有一定数量的许可权，该许可权可以获得或释放。如果有剩余的许可权，acquire() 方法将成功，否则该方法将被阻塞，直到有可用的许可权（通过其他线程释放许可权）。线程一次可以获得多个许可权。

计数信号可以用于限制有权对资源进行并发访问的线程数。该方法对于实现资源池或限制 Web 爬虫（Web crawler）中的输出 socket 连接非常有用。

注意信号不跟踪哪个线程拥有多少许可权；这由应用程序来决定，以确保何时线程释放许可权，该信号表示其他线程拥有许可权或者正在释放许可权，以及其他线程知道它的许可权已释放。

互斥

计数信号的一种特殊情况是互斥，或者互斥信号。互斥就是具有单一许可权的计数信号，意味着在给定时间仅一个线程可以具有许可权（也称为二进制信号）。互斥可以用于管理对共享资源的独占访问。

虽然互斥许多地方与锁定一样，但互斥还有一个锁定通常没有的其他功能，就是互斥可以由具有许可权的线程之外的其他线程来释放。这在死锁恢复时会非常有用。

CyclicBarrier 类可以帮助同步，它允许一组线程等待整个线程组到达公共屏障点。CyclicBarrier 是使用整型变量构造的，其确定组中的线程数。当一个线程到达屏障时（通过调用 CyclicBarrier.await()），它会被阻塞，直到所有线程都到达屏障，然后在该点允许所有线

程继续执行。该操作与许多家庭逛商业街相似 -- 每个家庭成员都自己走，并商定 1:00 在电影院集合。当您到电影院但不是所有人都到了时，您会坐下来等其他人到达。然后所有人一起离开。

认为屏障是循环的是因为它可以重新使用；一旦所有线程都已经在屏障处集合并释放，则可以将该屏障重新初始化到它的初始状态。 还可以指定在屏障处等待时的超时；如果在该时间内其余线程还没有到达屏障，则认为屏障被打破，所有正在等待的线程会收到 BrokenBarrierException。

下列代码将创建 CyclicBarrier 并启动一组线程，每个线程将计算问题的一部分，等待所有其他线程结束之后，再检查解决方案是否达成一致。如果不一致，那么每个工作线程将开始另一个迭代。该例将使用 CyclicBarrier 变量，它允许注册 Runnable，在所有线程到达屏障但还没有释放任何线程时执行 Runnable。

```
class Solver { // Code sketch
  void solve(final Problem p, int nThreads) {
  final CyclicBarrier barrier =
    new CyclicBarrier(nThreads,
      new Runnable() {
        public void run() { p.checkConvergence(); }}
    );
    for (int i = 0; i < nThreads; ++i) {
      final int id = i;
      Runnable worker = new Runnable() {
        final Segment segment = p.createSegment(id);
        public void run() {
          try {
            while (!p.converged()) {
              segment.update();
              barrier.await();
            }
          }
```

```
        catch(Exception e) { return; }
      }
    };
    new Thread(worker).start();
  }
}
```

CountdownLatch

CountdownLatch 类与 CyclicBarrier 相似，因为它的角色是对已经在它们中间分摊了问题的一组线程进行协调。它也是使用整型变量构造的，指明计数的初始值，但是与 CyclicBarrier 不同的是，CountdownLatch 不能重新使用。

其中，CyclicBarrier 是到达屏障的所有线程的大门，只有当所有线程都已经到达屏障或屏障被打破时，才允许这些线程通过，CountdownLatch 将到达和等待功能分离。任何线程都可以通过调用 countDown() 减少当前计数，这种不会阻塞线程，而只是减少计数。await() 方法的行为与 CyclicBarrier.await() 稍微有所不同，调用 await() 任何线程都会被阻塞，直到闩锁计数减少为零，在该点等待的所有线程才被释放，对 await() 的后续调用将立即返回。

当问题已经分解为许多部分，每个线程都被分配一部分计算时，CountdownLatch 非常有用。在工作线程结束时，它们将减少计数，协调线程可以在闩锁处等待当前这一批计算结束，然后继续移至下一批计算。

相反地，具有计数 1 的 CountdownLatch 类可以用作"启动大门"，来立即启动一组线程；工作线程可以在闩锁处等待，协调线程减少计数，从而立即释放所有工作线程。下例使用两个 CountdownLatche。一个作为启动大门，一个在所有工作线程结束时释放线程：

```
class Driver { // ...
  void main() throws InterruptedException {
    CountDownLatch startSignal = new CountDownLatch(1);
    CountDownLatch doneSignal = new CountDownLatch(N);
    for (int i = 0; i < N; ++i) // create and start threads
      new Thread(new Worker(startSignal, doneSignal)).start();
    doSomethingElse();            // don't let them run yet
    startSignal.countDown();      // let all threads proceed
```

```
    doSomethingElse();
    doneSignal.await();           // wait for all to finish
  }
}
class Worker implements Runnable {
  private final CountDownLatch startSignal;
  private final CountDownLatch doneSignal;
  Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
    this.startSignal = startSignal;
    this.doneSignal = doneSignal;
  }
  public void run() {
    try {
      startSignal.await();
      doWork();
      doneSignal.countDown();
    } catch (InterruptedException ex) {} // return;
  }
}
```

Exchanger 类方便了两个共同操作线程之间的双向交换；这样，就像具有计数为 2 的 CyclicBarrier，并且两个线程在都到达屏障时可以"交换"一些状态。（Exchanger 模式有时也称为聚集。）

Exchanger 通常用于一个线程填充缓冲（通过读取 socket），而另一个线程清空缓冲（通过处理从 socket 收到的命令）的情况。当两个线程在屏障处集合时，它们交换缓冲。下列代码说明了这项技术：

```
class FillAndEmpty {
  Exchanger exchanger = new Exchanger();
```

```
DataBuffer initialEmptyBuffer = new DataBuffer();
DataBuffer initialFullBuffer = new DataBuffer();
class FillingLoop implements Runnable {
  public void run() {
    DataBuffer currentBuffer = initialEmptyBuffer;
    try {
      while (currentBuffer != null) {
        addToBuffer(currentBuffer);
        if (currentBuffer.full())
          currentBuffer = exchanger.exchange(currentBuffer);
      }
    } catch (InterruptedException ex) { ... handle ... }
  }
}
class EmptyingLoop implements Runnable {
  public void run() {
    DataBuffer currentBuffer = initialFullBuffer;
    try {
      while (currentBuffer != null) {
        takeFromBuffer(currentBuffer);
        if (currentBuffer.empty())
          currentBuffer = exchanger.exchange(currentBuffer);
      }
    } catch (InterruptedException ex) { ... handle ...}
  }
}
```

```
  void start() {
    new Thread(new FillingLoop()).start();
    new Thread(new EmptyingLoop()).start();
  }
}
```

锁定和原子之**Lock**

Java 语言内置了锁定工具 -- synchronized 关键字。当线程获得监视器时（内置锁定），其他线程如果试图获得相同锁定，那么它们将被阻塞，直到第一个线程释放该锁定。同步还确保随后获得相同锁定的线程可以看到之前的线程在具有该锁定时所修改的变量的值，从而确保如果类正确地同步了共享状态的访问权，那么线程将不会看到变量的"失效"值，这是缓存或编译器优化的结果。

虽然同步没有什么问题，但它有一些限制，在一些高级应用程序中会造成不便。Lock 接口将内置监视器锁定的锁定行为普遍化，允许多个锁定实现，同时提供一些内置锁定缺少的功能，如计时的等待、可中断的等待、锁定轮询、每个锁定有多个条件等待集合以及无阻塞结构的锁定。

```
interface Lock {
    void lock();
    void lockInterruptibly() throws IE;
    boolean tryLock();
    boolean tryLock(long time,
                    TimeUnit unit) throws IE;
    void unlock();
    Condition newCondition() throws
                    UnsupportedOperationException;
  }
```

ReentrantLock

ReentrantLock 是具有与隐式监视器锁定（使用 synchronized 方法和语句访问）相同的基本行为和语义的 Lock 的实现，但它具有扩展的能力。

作为额外收获，在竞争条件下，ReentrantLock 的实现要比现在的 synchronized 实现更

具有可伸缩性。（有可能在 JVM 的将来版本中改进 synchronized 的竞争性能。）

这意味着当许多线程都竞争相同锁定时，使用 ReentrantLock 的吞吐量通常要比 synchronized 好。换句话说，当许多线程试图访问 ReentrantLock 保护的共享资源时，JVM 将花费较少的时间来调度线程，而用更多个时间执行线程。

虽然 ReentrantLock 类有许多优点，但是与同步相比，它有一个主要缺点 -- 它可能忘记释放锁定。建议当获得和释放 ReentrantLock 时使用下列结构：

```
Lock lock = new ReentrantLock();

...

lock.lock();

try {

  // perform operations protected by lock

}

catch(Exception ex) {

 // restore invariants

}

finally {

  lock.unlock();

}
```

因为锁定失误（忘记释放锁定）的风险，所以对于基本锁定，强烈建议您继续使用 synchronized，除非真的需要 ReentrantLock 额外的灵活性和可伸缩性。

ReentrantLock 是用于高级应用程序的高级工具 -- 有时需要，但有时用原来的方法就很好。

Condition

就像 Lock 接口是同步的具体化，Condition 接口是 Object 中 wait() 和 notify() 方法的具体化。Lock 中的一个方法是 newCondition()，它要求锁定向该锁定返回新的 Condition 对象限制。await()、signal() 和 signalAll() 方法类似于 wait()、notify() 和 notifyAll()，但增加了灵活性，每个 Lock 都可以创建多个条件变量。这简化了一些并发算法的实现。

ReadWriteLock

ReentrantLock 实现的锁定规则非常简单 -- 每当一个线程具有锁定时，其他线程必须等

待，直到该锁定可用。有时，当对数据结构的读取通常多于修改时，可以使用更复杂的称为读写锁定的锁定结构，它允许有多个并发读者，同时还允许一个写入者独占锁定。该方法在一般情况下（只读）提供了更大的并发性，同时在必要时仍提供独占访问的安全性。ReadWriteLock 接口和 ReentrantReadWriteLock 类提供这种功能 -- 多读者、单写入者锁定规则，可以用这种功能来保护共享的易变资源。

原子变量

即使大多数用户将很少直接使用它们，原子变量类（AtomicInteger、AtomicLong、AtomicReference 等等）也有充分理由是最显著的新并发类。这些类公开对 JVM 的低级别改进，允许进行具有高度可伸缩性的原子读-修改-写操作。大多数现代 CPU 都有原子读-修改-写的原语，比如比较并交换（CAS）或加载链接/条件存储（LL/SC）。原子变量类使用硬件提供的最快的并发结构来实现。

许多并发算法都是根据对计数器或数据结构的比较并交换操作来定义的。通过暴露高性能的、高度可伸缩的 CAS 操作（以原子变量的形式），用 Java 语言实现高性能、无等待、无锁定的并发算法已经变得可行。

几乎 java.util.concurrent 中的所有类都是在 ReentrantLock 之上构建的，ReentrantLock 则是在原子变量类的基础上构建的。所以，虽然仅少数并发专家使用原子变量类，但 java.util.concurrent 类的很多可伸缩性改进都是由它们提供的。

原子变量主要用于为原子地更新"热"字段提供有效的、细粒度的方式，"热"字段是指由多个线程频繁访问和更新的字段。另外，原子变量还是计数器或生成序号的自然机制。

性能与可伸缩性

虽然 java.util.concurrent 努力的首要目标是使编写正确、线程安全的类更加容易，但它还有一个次要目标，就是提供可伸缩性。可伸缩性与性能完全不同，实际上，可伸缩性有时要以性能为代价来获得。

性能是"可以快速执行此任务的程度"的评测。可伸缩性描述应用程序的吞吐量如何表现为它的工作量和可用计算资源增加。可伸缩的程序可以按比例使用更多的处理器、内存或 I/O 带宽来处理更多个工作量。当我们在并发环境中谈论可伸缩性时，我们是在问当许多线程同时访问给定类时，这个类的执行情况。

java.util.concurrent 中的低级别类 ReentrantLock 和原子变量类的可伸缩性要比内置监视器（同步）锁定高得多。因此，使用 ReentrantLock 或原子变量类来协调共享访问的类也可能更具有可伸缩性。

Hashtable 与 ConcurrentHashMap

作为可伸缩性的例子，ConcurrentHashMap 实现设计的可伸缩性要比其线程安全的上一代 Hashtable 的可伸缩性强得多。Hashtable 一次只允许一个线程访问 Map；

ConcurrentHashMap 允许多个读者并发执行，读者与写入者并发执行，以及一些写入者并发执行。因此，如果许多线程频繁访问共享映射，使用 ConcurrentHashMap 的总的吞吐量要比使用 Hashtable 的好。

下表大致说明了 Hashtable 和 ConcurrentHashMap 之间的可伸缩性差别。在每次运行时，N 个线程并发执行紧密循环，它们从 Hashtable 或 ConcurrentHashMap 中检索随即关键字，60% 的失败检索将执行 put() 操作，2% 的成功检索执行 remove() 操作。测试在运行 Linux 的双处理器 Xeon 系统中执行。数据显示 10,000,000 个迭代的运行时间，对于 ConcurrentHashMap，标准化为一个线程的情况。可以看到直到许多线程，ConcurrentHashMap 的性能仍保持可伸缩性，而 Hashtable 的性能在出现锁定竞争时几乎立即下降。

与通常的服务器应用程序相比，这个测试中的线程数看起来很少。然而，因为每个线程未进行其他操作，仅是重复地选择使用该表，所以这样可以模拟在执行一些实际工作的情况下使用该表的大量线程的竞争。

线程 ConcurrentHashMap Hashtable

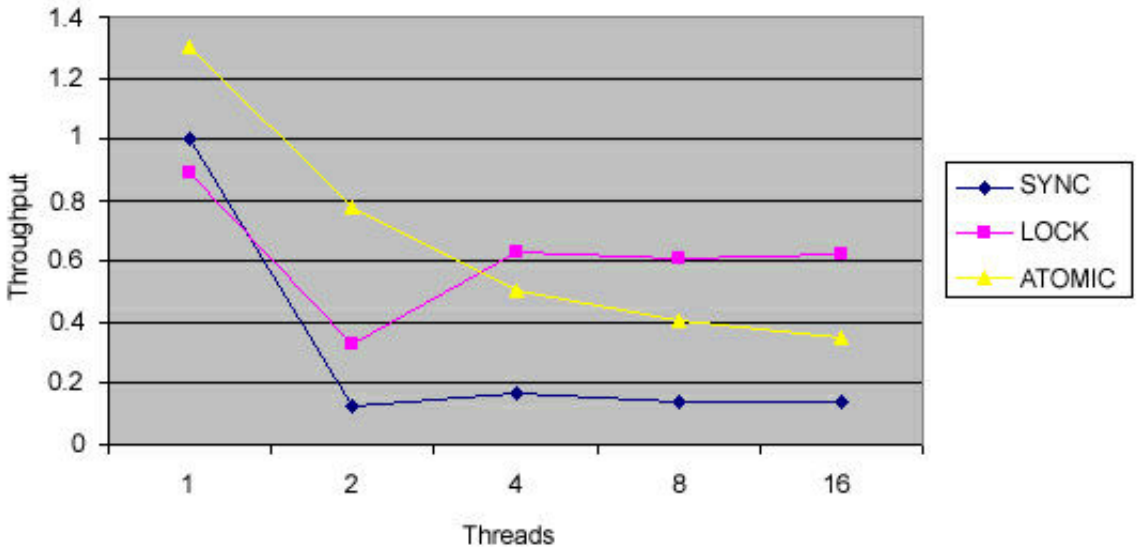
线程	**ConcurrentHashMap**	**Hashtable**
1	1.0	1.51
2	1.44	17.09
4	1.83	29.9
8	4.06	54.06
16	7.5	119.44
32	15.32	237.2

Lock 与 synchronized 与原子

下列基准说明了使用 java.util.concurrent 可能改进可伸缩性的例子。该基准将模拟旋转骰子，使用线性同余随机数生成器。有三个可用的随机数生成器的实现：一个使用同步来管理生成器的状态（单一变量），一个使用 ReentrantLock，另一个则使用 AtomicLong。下图显示

了在 8-way Ultrasparc3 系统上，逐渐增加线程数量时这三个版本的相对吞吐量。（该图对原子变量方法的可伸缩性描述比较保守。）

图 1. 使用同步、Lock 和 AtomicLong 的相对吞吐量



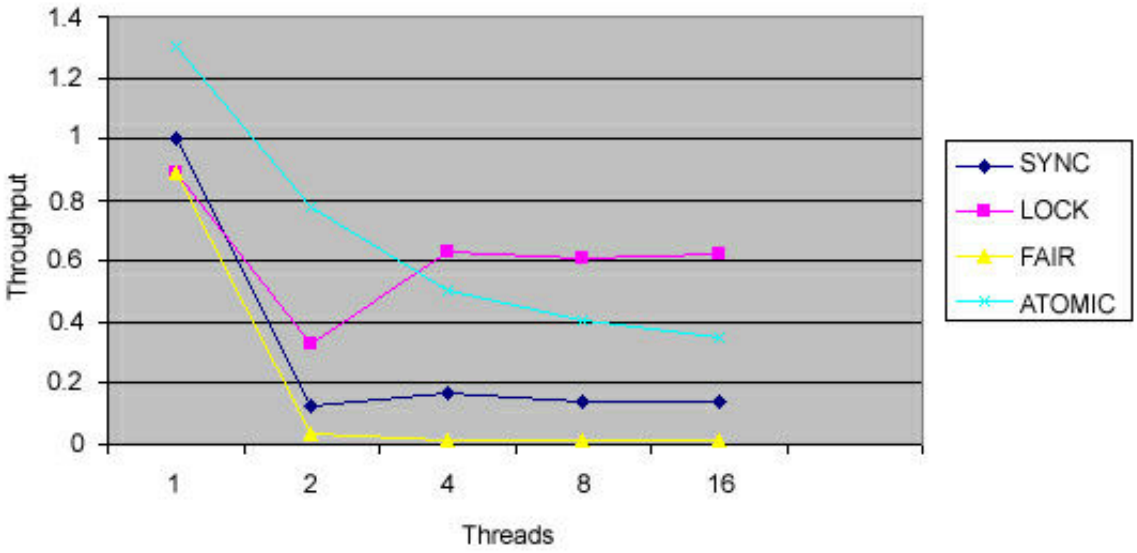
公平与不公平

java.util.concurrent 中许多类中的另外一个定制元素是"公平"的问题。公平锁定或公平信号是指在其中根据先进先出（FIFO）的原则给与线程锁定或信号。ReentrantLock、Semaphore 和 ReentrantReadWriteLock 的构造函数都可以使用变量确定锁定是否公平，或者是否允许闯入（线程获得锁定，即使它们等待的时间不是最长）。

虽然闯入锁定的想法可能有些可笑，但实际上不公平、闯入的锁定非常普遍，且通常很受欢迎。使用同步访问的内置锁定不是公平锁定（且没有办法使它们公平）。相反，它们提供较弱的生病保证，要求所有线程最终都将获得锁定。

大多数应用程序选择（且应该选择）闯入锁定而不是公平锁定的原因是性能。在大多数情况下，完全的公平不是程序正确性的要求，真正公平的成本相当高。下表向前面的面板中的表中添加了第四个数据集，并由一个公平锁定管理对 PRNG 状态的访问。注意闯入锁定与公平锁定之间吞吐量的巨大差别。

图 2. 使用同步、Lock、公平锁定和 AtomicLong 的相对吞吐量



结束语

java.util.concurrent 包中包含大量有用的构建快，可以用它们来改进并发类的性能、可伸缩性、线程安全和可维护性。通过这些构建快，应该可以不再需要在您的代码中大量使用同步、wait/notify 和 Thread.start()，而用更高级别、标准化的、高性能并发实用程序来替换它们。

Exchanger

CyclicBarrier

Synchronizer

本文由 blog博主Caoer（草儿）原创，此处为转载。

由于原文中文末两张图片无法显示，转载时本博(http://www.cnblogs.com/sarafill/)重新引用了图片，并调整了版面。