notes-learning-java-concurrency

angelosun

Published
with GitBook

Table of Contents

Java并发编程学习笔记

这是我两年前(2012-05)的学习笔记。点击直接访问web版页面。

-- 本文不会详细介绍java5以前的多线程开发的知识,而是重点介绍java5以来多线程开发的一些新变化。部分文字、代码都是摘抄、提炼过来的，大家有兴趣可查看(8.相关资料)中的提供的原材料。

本文的主要内容如下：

2.集合类。主要介绍多线程开发中如何使用集合(三种方法)。建议大家关注一下java.util. concurrent包中的相关类。

4.线程池。你需要明确的是: 任务只是一组逻辑工作单元，而线程则是任务异步执行的机制。任务与任务的执行是相分离的。

5.任务异步返回结果和取消关闭。以前的线程实现都是没有返回值的，但java5中，有返回值的线程是如何实现的呢？

6.显式锁。协调共享对象访问，在java5以前用synchronized实现,现在可以用Lock显式的lock()和unlock(),并且有定时锁，读写锁等，你用过吗？

7.栅栏CyclicBarrier、计数器CountDownLatch、信号量Semaphore。这几个类确实让人眼前一亮。你知道它们是做什么的吗？

看完本文，你可以放心的遵循《Effective Java中文版(第2版)》的并发实践了：

> 第68条：executor和task优先于线程 第69条：并发工具优先于wait和notify

常用的集合结构

```
Collection
    --List： 将以特定次序存储元素。所以取出来的顺序可能和放入顺序不同。
          --ArrayList / LinkedList / Vector
    --Set ： 不能含有重复的元素
          --HashSet / TreeSet
Map
    --HashMap
    --HashTable
    --TreeMap
```

集合与线程安全

线程安全的集合包的三种实现方式。

```
摘抄：《Java 理论与实践: 并发集合类》

在Java类库中出现的第一个关联的集合类是 Hashtable ，它是JDK 1.0的一部分。 Hashtable 提供了一种易于使用

Hashtable 的后继者 HashMap 是作为JDK1.2中的集合框架的一部分出现的，它通过提供一个不同步的基类和一个同步

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 包装器对象

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
```

(1) 线程安全类 Vector,HashTable

Vector，因为效率较低，现在已经不太建议使用。

(2) 使用ArrayList/HashMap和同步包装器

```
List synchArrayList = Collections.synchronizedList(new ArrayList());
Map synchHashMap = Collections.synchronizedMap(new HashMap())  ;
```

如果要迭代，需要这样

```
synchronized (synchHashMap)  {
   Iterator iter = synchHashMap.keySet().iterator();
   while (iter.hasNext()) . . .;
}
```

(3) java.util.concurrent包

用java5.0新加入的ConcurrentHashMap、ConcurrentLinkedQueue、CopyOnWriteArray- List 和 CopyOnWriteArraySet，对这些集合进行并发修改是安全的。

ConcurrentHashMap是线程安全的HashMap实现。无论元素数量为多少，在线程数为10时，ConcurrentHashMap带来的性能提升并不是很明显。 但在线程数为50和100时，Concurrent- HashMap在添加和删除元素时带来了一倍左右的性能提升，在查找元素上更是带来了10倍左右的性能提升，并且随着线程数增长，ConcurrentHashMap性能并没有出现下降的现象。

CopyOnWriteArrayList是一个线程安全、并且在读操作时无锁的ArrayList。：在读多写少的并发环境中，

一般用 CopyOnWriteArrayList 类替代 ArrayList 。

队列：Queue, BlockingQueue

JDK1.5也增加了两种新的容器类型：Queue和BlockingQueue。

Queue是用来临时保存一组等待处理的元素。Queue上的操作不会阻塞，如果队列为空，那么获取元素的操作将返回空值。

BlockingQueue扩展了Queue，增加了可阻塞的插入和获取等级操作。如果队列为空，那么获取元素的操作将一直阻塞，直到队列中出现一个可用的元素。如果队列已满，那么插入元素的操作将一直阻塞，直到队列中出现可用的空间。在生产者、消费者模式中，阻塞队列是非常有用的。

所有已知实现类： ArrayBlockingQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue

基于阻塞队列BlockingQueue的生产者、消费者模式

```
import java.util.concurrent.BlockingQueue;

public class Producer implements Runnable {
    private BlockingQueue queue;

    public Producer(BlockingQueue queue) {
        this.queue = queue;
    }

    public void run() {
        for (int product = 1; product <= 10; product++) {
            try {
                Thread.sleep((int) Math.random() * 3000);
                queue.put(product);
                System.out.println("Producer 生产： " + product);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

import java.util.concurrent.BlockingQueue;

public class Consumer implements Runnable {
    private BlockingQueue queue;

    public Consumer(BlockingQueue queue) {
        this.queue = queue;
    }

    public void run() {
        for (int i = 1; i <= 10; i++) {
            try {
                Thread.sleep((int) (Math.random() * 3000));
                System.out.println("Consumer 消费： "+ queue.take());
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class BlockingQueueDemo {
    public static void main(String[] args) {
        BlockingQueue queue = new ArrayBlockingQueue(1);
        Thread producerThread = new Thread(
        new Producer(queue));
        Thread consumerThread = new Thread(
        new Consumer(queue));
        producerThread.start();
        consumerThread.start();
    }
}
```

```
运行结果：
Producer 生产： 1
Consumer 消费： 1
Producer 生产： 2
Consumer 消费： 2
Producer 生产： 3
Consumer 消费： 3
Producer 生产： 4
Consumer 消费： 4
Producer 生产： 5
Consumer 消费： 5
Producer 生产： 6
Consumer 消费： 6
Producer 生产： 7
Consumer 消费： 7
Producer 生产： 8
Consumer 消费： 8
Producer 生产： 9
Consumer 消费： 9
Producer 生产： 10
Consumer 消费： 10
```

并发Deque

在JDK6中，还提供了一种双端队列(Double-Ended Queue),简称Deque。Deque允许在队列的头部或尾部进行出队和入队操作。

精巧好用的DelayQueue

原文：http://www.cnblogs.com/jobs/archive/2007/04/27/730255.html

我们谈一下实际的场景吧。我们在开发中，有如下场景

a) 关闭空闲连接。服务器中，有很多客户端的连接，空闲一段时间之后需要关闭之。

b) 缓存。缓存中的对象，超过了空闲时间，需要从缓存中移出。

c) 任务超时处理。在网络协议滑动窗口请求应答式交互时，处理超时未响应的请求。

一种笨笨的办法就是，使用一个后台线程，遍历所有对象，挨个检查。这种笨笨的办法简单好用，但是对象数量过多时，可能存在性能问题，检查间隔时间不好设置，间隔时间过大，影响精确度，多小则存在效率问题。而且做不到按超时的时间顺序处理。

这场景，使用DelayQueue最适合了。

DelayQueue是一个BlockingQueue，其特化的参数是Delayed。（不了解BlockingQueue的同学，先去了解BlockingQueue再看本文） Delayed扩展了Comparable接口，比较的基准为延时的时间值，Delayed接口的实现类getDelay的返回值应为固定值（final）。DelayQueue内部是使用PriorityQueue实现的。

DelayQueue = BlockingQueue + PriorityQueue + Delayed

DelayQueue的关键元素BlockingQueue、PriorityQueue、Delayed。可以这么说，DelayQueue是一个使用优先队列（PriorityQueue）实现的BlockingQueue，优先队列的比较基准值是时间。

他们的基本定义如下

```
public interface Comparable<T> {
    public int compareTo(T o);
}

public interface Delayed extends Comparable<Delayed> {
    long getDelay(TimeUnit unit);
}

public class DelayQueue<E extends Delayed> implements BlockingQueue<E> {
    private final PriorityQueue<E> q = new PriorityQueue<E>();
}
```

DelayQueue内部的实现使用了一个优先队列。当调用DelayQueue的offer方法时，把Delayed对象加入到优先队列q中。如下：

```
public boolean offer(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        E first = q.peek();
```

```
            q.offer(e);
            if (first == null || e.compareTo(first) < 0)
                available.signalAll();
            return true;
        } finally {
            lock.unlock();
        }
    }
```

DelayQueue的take方法，把优先队列q的first拿出来（peek），如果没有达到延时阀值，则进行await处理。如下：

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        for (;;) {
            E first = q.peek();
            if (first == null) {
                available.await();
            } else {
                long delay =  first.getDelay(TimeUnit.NANOSECONDS);
                if (delay > 0) {
                    long tl = available.awaitNanos(delay);
                } else {
                    E x = q.poll();
                    assert x != null;
                    if (q.size() != 0)
                        available.signalAll(); // wake up other takers
                    return x;

                }
            }
        }
    } finally {
        lock.unlock();
    }
}
```

以下是Sample，是一个缓存的简单实现。共包括三个类Pair、DelayItem、Cache。如下：

public class Pair { public K first;

```
public V second;

public Pair() {}

public Pair(K first, V second) {
    this.first = first;
    this.second = second;
}
```

}

以下是Delayed的实现

```
import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;

public class DelayItem<T> implements Delayed {
    /** Base of nanosecond timings, to avoid wrapping */
    private static final long NANO_ORIGIN = System.nanoTime();

    /**
     * Returns nanosecond time offset by origin
     */
    final static long now() {
        return System.nanoTime() - NANO_ORIGIN;
    }

    /**
     * Sequence number to break scheduling ties, and in turn to guarantee FIFO order amon
     * entries.
     */
    private static final AtomicLong sequencer = new AtomicLong(0);

    /** Sequence number to break ties FIFO */
    private final long sequenceNumber;

    /** The time the task is enabled to execute in nanoTime units */
    private final long time;

    private final T item;

    public DelayItem(T submit, long timeout) {
        this.time = now() + timeout;
        this.item = submit;
        this.sequenceNumber = sequencer.getAndIncrement();
    }

    public T getItem() {
        return this.item;
    }

    public long getDelay(TimeUnit unit) {
        long d = unit.convert(time - now(), TimeUnit.NANOSECONDS);
        return d;
    }

    public int compareTo(Delayed other) {
        if (other == this) // compare zero ONLY if same object
            return 0;
        if (other instanceof DelayItem) {
            DelayItem x = (DelayItem) other;
            long diff = time - x.time;
            if (diff < 0)
                return -1;
            else if (diff > 0)
                return 1;
            else if (sequenceNumber < x.sequenceNumber)
                return -1;
```

```
            else
                return 1;
        }
        long d = (getDelay(TimeUnit.NANOSECONDS) - other.getDelay(TimeUnit.NANOSECONDS));
        return (d == 0) ? 0 : ((d < 0) ? -1 : 1);
    }
}
```

以下是Cache的实现，包括了put和get方法，还包括了可执行的main函数。

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.DelayQueue;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Cache<K, V> {
    private static final Logger LOG = Logger.getLogger(Cache.class.getName());

    private ConcurrentMap<K, V> cacheObjMap = new ConcurrentHashMap<K, V>();

    private DelayQueue<DelayItem<Pair<K, V>>> q = new DelayQueue<DelayItem<Pair<K, V>>>()

    private Thread daemonThread;

    public Cache() {

        Runnable daemonTask = new Runnable() {
            public void run() {
                daemonCheck();
            }
        };

        daemonThread = new Thread(daemonTask);
        daemonThread.setDaemon(true);
        daemonThread.setName("Cache Daemon");
        daemonThread.start();
    }

    private void daemonCheck() {

        if (LOG.isLoggable(Level.INFO))
            LOG.info("cache service started.");

        for (;;) {
            try {
                DelayItem<Pair<K, V>> delayItem = q.take();
                if (delayItem != null) {
                    // 超时对象处理
                    Pair<K, V> pair = delayItem.getItem();
                    cacheObjMap.remove(pair.first, pair.second); // compare and remove
                }
            } catch (InterruptedException e) {
                if (LOG.isLoggable(Level.SEVERE))
                    LOG.log(Level.SEVERE, e.getMessage(), e);
```

```
                    break;
                }
            }

            if (LOG.isLoggable(Level.INFO))
                LOG.info("cache service stopped.");
        }

        // 添加缓存对象
        public void put(K key, V value, long time, TimeUnit unit) {
            V oldValue = cacheObjMap.put(key, value);
            if (oldValue != null)
                q.remove(key);

            long nanoTime = TimeUnit.NANOSECONDS.convert(time, unit);
            q.put(new DelayItem<Pair<K, V>>(new Pair<K, V>(key, value), nanoTime));
        }

        public V get(K key) {
            return cacheObjMap.get(key);
        }

        // 测试入口函数
        public static void main(String[] args) throws Exception {
            Cache<Integer, String> cache = new Cache<Integer, String>();
            cache.put(1, "aaaa", 3, TimeUnit.SECONDS);

            Thread.sleep(1000 * 2);
            {
                String str = cache.get(1);
                System.out.println(str);
            }

            Thread.sleep(1000 * 2);
            {
                String str = cache.get(1);
                System.out.println(str);
            }
        }
    }
```

运行Sample，main函数执行的结果是输出两行，第一行为aaa，第二行为null。

探索 ConcurrentHashMap 高并发性的实现机制

http://www.ibm.com/developerworks/cn/java/java-lo-concurrenthashmap/index.html?ca=drs-

Java 理论与实践: 并发集合类

http://www.ibm.com/developerworks/cn/java/j-jtp07233/

Java 理论与实践: 构建一个更好的 HashMap

http://www.ibm.com/developerworks/cn/java/j-jtp08223/

原子操作类Atomic

相关类有：AtomicInteger,AtomicLong,AtomicBoolean,AtomicReference。

这些原子类的方法都是基于CPU的CAS原语来操作的。基于CAS的方式比使用synchronized的方式性能提高2.8倍。因此对于使用JDK5以上版本且必须支持高并发而言，应尽量使用atomic的类。

Atomic典型应用:按顺序获取ID

传统方式必须在每次获取时加锁，以防止出现并发时取到同样id的现象。

用AtomicInteger实现示例如下：

```
private static AtomicInteger counter = new AtomicInteger();
public static int getNextId() {
    return counter.incrementAndGet();
}
```

示例

Java线程：新特征-原子量

http://lavasoft.blog.51cto.com/62575/222541

反面例子（切勿模仿）：

```
/**
 * Java线程：新特征-原子量
*/
public class Test {
    public static void main(String[] args) {
        ExecutorService pool = Executors.newFixedThreadPool(2);
        Runnable t1 = new MyRunnable("张三", 2000);
        Runnable t2 = new MyRunnable("李四", 3600);
        Runnable t3 = new MyRunnable("王五", 2700);
        Runnable t4 = new MyRunnable("老张", 600);
        Runnable t5 = new MyRunnable("老牛", 1300);
        Runnable t6 = new MyRunnable("胖子", 800);
        // 执行各个线程
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        pool.execute(t4);
        pool.execute(t5);
        pool.execute(t6);
        // 关闭线程池
        pool.shutdown();
    }
}

class MyRunnable implements Runnable {
    // 原子量，每个线程都可以自由操作
    private static AtomicLong aLong = new AtomicLong(10000);
    private String name; // 操作人
    private int x; // 操作数额

    MyRunnable(String name, int x) {
        this.name = name;
        this.x = x;
    }

    public void run() {
        System.out.println(name + "执行了" + x + "，当前余额：" + aLong.addAndGet(x));
    }
}
```

```
运行结果：
李四执行了3600，当前余额：13600
王五执行了2700，当前余额：16300
老张执行了600，当前余额：16900
老牛执行了1300，当前余额：18200
```

```
胖子执行了800，当前余额：19000
张三执行了2000，当前余额：21000

张三执行了2000，当前余额：12000
王五执行了2700，当前余额：18300
老张执行了600，当前余额：18900
老牛执行了1300，当前余额：20200
胖子执行了800，当前余额：21000
李四执行了3600，当前余额：15600

张三执行了2000，当前余额：12000
李四执行了3600，当前余额：15600
老张执行了600，当前余额：18900
老牛执行了1300，当前余额：20200
胖子执行了800，当前余额：21000
王五执行了2700，当前余额：18300
```

从运行结果可以看出，虽然使用了原子量，但是程序并发访问还是有问题，那究竟问题出在哪里了？

这里要注意的一点是，原子量虽然可以保证单个变量在某一个操作过程的安全，但无法保证你整个代码块，或者整个程序的安全性。因此，通常还应该使用锁等同步机制来控制整个程序的安全性。

下面是对这个错误修正：

```
/**
 * Java线程：新特征-原子量
 */
public class Test {
    public static void main(String[] args) {
        ExecutorService pool = Executors.newFixedThreadPool(2);
        Lock lock = new ReentrantLock(false);
        Runnable t1 = new MyRunnable("张三", 2000, lock);
        Runnable t2 = new MyRunnable("李四", 3600, lock);
        Runnable t3 = new MyRunnable("王五", 2700, lock);
        Runnable t4 = new MyRunnable("老张", 600, lock);
        Runnable t5 = new MyRunnable("老牛", 1300, lock);
        Runnable t6 = new MyRunnable("胖子", 800, lock);
        // 执行各个线程
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        pool.execute(t4);
        pool.execute(t5);
        pool.execute(t6);
        // 关闭线程池
        pool.shutdown();
    }
}

class MyRunnable implements Runnable {
    // 原子量，每个线程都可以自由操作
    private static AtomicLong aLong = new AtomicLong(10000);
    private String name; // 操作人
    private int x; // 操作数额
    private Lock lock;
```

```
    MyRunnable(String name, int x, Lock lock) {
        this.name = name;
        this.x = x;
        this.lock = lock;
    }

    public void run() {
        lock.lock();
        System.out.println(name + "执行了" + x + ", 当前余额：" + aLong.addAndGet(x));
        lock.unlock();
    }
}
```

```
执行结果：
张三执行了2000, 当前余额：12000
王五执行了2700, 当前余额：14700
老张执行了600, 当前余额：15300
老牛执行了1300, 当前余额：16600
胖子执行了800, 当前余额：17400
李四执行了3600, 当前余额：21000
```

这里使用了一个对象锁，来控制对并发代码的访问。不管运行多少次，执行次序如何，最终余额均为21000，这个结果是正确的。

线程池Executor，ThreadPoolExecutor，Executors

线程池介绍

线程池的基本思想还是一种对象池的思想，开辟一块内存空间，里面存放了众多（未死亡）的线程，池中线程执行调度由池管理器来处理。当有线程任务时，从池中取一个，执行完成后线程对象归池，这样可以避免反复创建线程对象所带来的性能开销，节省了系统的资源。

任务是一组逻辑工作单元，而线程则是任务异步执行的机制。

线程池可以解决两个不同问题：由于减少了每个任务调用的开销，它们通常可以在执行大量异步任务时提供增强的性能，并且还可以提供绑定和管理资源（包括执行集合任务时使用的线程）的方法。每个ThreadPoolExecutor还维护着一些基本的统计数据，如完成的任务数。

线程池简化了线程的管理工作，并且java.util.concurrent提供了一种灵活的线程池实现作为Executor框架的一部分。在java类库中，任务执行的主要抽象不是Thread，而是Executor。

```
public interface Executor {
    void execute(Runnable command);
}
```

Executor的实现还提供了对生命周期的支持，以及统计信息收集、应用程序管理机制和性能监视等机制。

任务执行的三种模式

(1)串行的执行任务

```
//串行的web服务器
class SingleThreadWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }
}
```

SingleThreadWebServer很简单，且在理论上是正确的，但在实际生产环境中的执行性能却很糟糕，因为它每次只能处理一次请求。在服务器应用程序中，串行处理机制都无法提供高吞吐率或快速响应性。

(2)显式的为任务创建线程

```
//在web服务器中为每个请求启动一个新的线程(不要这么做)
class ThreadPerTaskWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            new Thread(task).start();
        }
    }
}
```

在正常的情况下，”为每个任务分配一个线程”的方法能提升串行执行任务的性能。只要请求的到达速率不超过服务器的请求处理能力，那么这种方法可以同时带来更快的响应性和更高的吞吐率。

但这种方法存在一些缺陷，尤其当需要创建大量线程的时候。

- 线程生命周期的开销非常高。
- 资源消耗。活跃的线程会消耗系统资源，尤其是内存。如果可运行的线程数多于可用处理器的数量，那么有些线程将闲置。大量空闲的线程会占用许多内存，给垃圾回收器带来压力，而且大量线程在竞争CPU资源时还将产生其他的性能开销。
- 稳定性。在可创建线程的数量上存在一个限制。如果超过了这些限制，很可能会抛出outOfMemoryError异常，要想从这种错误中恢复过来是非常危险的，更简单的方法是通过构造程序来

避免超出这些限制。

(3).基于线程池的实现

```
// 基于线程池的web服务器
class TaskExecutionWebServer {
    private static final int NTHREADS = 100;
    private static final Executor exec = Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
}
```

在TaskExecutionWebServer中，通过使用Executor，将请求处理任务的提交与任务的实际执行解耦开来，并且只需采用另一种不同的Executor实现，就可以改变服务器的行为。

ThreadPoolExecutor

1.通用构造函数

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

用给定的初始参数和默认的线程工厂及处理程序创建新的 ThreadPoolExecutor。使用 Executors 工厂方法之一比使用此通用构造方法方便得多。

参数：

corePoolSize - 池中所保存的线程数，包括空闲线程。

maximumPoolSize - 池中允许的最大线程数。

keepAliveTime - 当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间。

unit - keepAliveTime 参数的时间单位。

workQueue - 执行前用于保持任务的队列。此队列仅保持由 execute 方法提交的 Runnable 任务。

threadFactory - 执行程序创建新线程时使用的工厂。

handler - 由于超出线程范围和队列容量而使执行被阻塞时所使用的处理程序。

抛出：

IllegalArgumentException - 如果 corePoolSize 或 keepAliveTime 小于零，或者 maximumPoolSize 小于或等于零，或者 corePoolSize 大于 maximumPoolSize。

NullPointerException - 如果 workQueue、threadFactory 或 handler 为 null。

2. 线程的创建和销毁

线程池的基本大小（corePoolSize）、最大大小（maximumPoolSize）以及存活时间等因素共同负责线程的创建与销毁。

基本大小也是线程池的目标大小，即在没有任务执行时线程池的大小，并且只有在工作队列满了的情况下才会创建超出这个数量的线程。

最大大小表示可同时活动的线程数量的上限。

如果某个线程的空闲时间超过了存活时间，那么将被标记为可回收的，并且当线程池的当前大小超过基本大小时，这个线程将被终止。

3. 管理队列任务

ThreadPoolExecutor允许提供一个BlockingQueue来保存等待执行的任务。基本的任务排队方法有3种：有界队列, 无界队列, 同步移交（Synchronous Handoff）。

4. 有界队列饱和策略

有界队列被填满后，饱和策略开始发挥作用。饱和策略可以通过调用setRejectedExecutionHandler来修改。jdk提供了几种不同的RejectedExecutionHandler实现：AbortPolicy, CallerRunsPolicy, DiscardOldestPolicy, DiscardPolicy 。

AbortPolicy：默认的饱和策略。抛出 RejectedExecutionException。调用者可以捕获这个异常，然后根据需求编写自己的处理代码。

CallerRunsPolicy: 不抛弃任务，不抛出异常，而将任务退回给调用者。

DiscardOldestPolicy：放弃最旧的未处理请求，然后重试 execute；如果执行程序已关闭，则会丢弃该任务。

DiscardPolicy:默认情况下它将放弃被拒绝的任务。

5. 扩展ThreadPoolExecutor

ThreadPoolExecutor是可扩展的，它提供了几个可以在子类化中改写的方法：beforeExecute、afterExcute和terminated。在这些方法中，还可以添加日志、计时、监视或统计信息收集的功能。

无论任务是从run中正常返回，还是会抛出一个异常在而返回，afterExcute都会被调用。（如果任务执行完成后带有Error，那么就不会调用afterExcute）。

如果beforeExecute抛出一个RuntimeException，那么任务将不被执行，并且afterExcute也会被调用。

在线程池完成关闭操作时调用terminated，也就是在所有任务都已经完成并且所有工作者线程也已经关闭后，terminated可以释放Executor在其生命周期里分配的各种资源，此外还可以执行发送通知，记录日志或者收集finalize统计信息等操作。

```
//增加了日志和计时等功能的线程池
public class TimingThreadPool extends ThreadPoolExecutor {

    private final ThreadLocal<Long> startTime = new ThreadLocal<Long>();
    private final Logger log = Logger.getLogger("TimingThreadPool");
    private final AtomicLong numTasks = new AtomicLong();
    private final AtomicLong totalTime = new AtomicLong();
    //构造函数
```

```
    //...

    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        log.fine(String.format("Thread %s: start %s", t, r));
        startTime.set(System.nanoTime());
    }

    protected void afterExecute(Runnable r, Throwable t) {
        try {
            long endTime = System.nanoTime();
            long taskTime = endTime - startTime.get();
            numTasks.incrementAndGet();
            totalTime.addAndGet(taskTime);
            log.fine(String.format("Thread %s: end %s, time=%dns", t, r,
                    taskTime));
        } finally {
            super.afterExecute(r, t);
        }
    }

    protected void terminated() {
        try {
            log.info(String.format("Terminated: avg time=%dns", totalTime.get()
                    / numTasks.get()));
        } finally {
            super.terminated();
        }
    }
}
```

Executors创建线程池

类库提供了一个灵活的线程池以及一些有用的默认配置，可以通过调用Executors中的静态工厂方法之一创建一个线程池。

```
// 创建一个可重用固定线程数的线程池
ExecutorService pool = Executors.newFixedThreadPool(2);
```

(1) newFixedThreadPool

创建一个可重用固定线程集合的线程池，以共享的无界队列方式来运行这些线程。

(2) newCachedThreadPool

创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。

(3) newSingleThreadExecutor

创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。

(4) newScheduledThreadPool

创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

(5) newFixedThreadPool,newCachedThreadPool返回通用的ThreadPoolExecutor实例。

线程池创建示例

1. 固定大小的线程池

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

/**
 * Java线程：线程池-固定线程数的线程池
 */
public class Test {
    public static void main(String[] args) {
        // 创建一个可重用固定线程数的线程池
        ExecutorService pool = Executors.newFixedThreadPool(2);
        // 创建实现了Runnable接口对象，Thread对象当然也实现了Runnable接口
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        Thread t3 = new MyThread();
        Thread t4 = new MyThread();
        Thread t5 = new MyThread();
        // 将线程放入池中进行执行
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        pool.execute(t4);
        pool.execute(t5);
        // 关闭线程池
        pool.shutdown();
    }
}

class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + "正在执行。。。");
    }
}
```

```
pool-1-thread-1正在执行。。。
pool-1-thread-2正在执行。。。
pool-1-thread-1正在执行。。。
pool-1-thread-2正在执行。。。
pool-1-thread-1正在执行。。。
```

2. 单任务线程池

在上例的基础上改一行创建pool对象的代码为：

```
//创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。
```

```
ExecutorService pool = Executors.newSingleThreadExecutor();

输出结果为：
pool-1-thread-1正在执行。。。
pool-1-thread-1正在执行。。。
pool-1-thread-1正在执行。。。
pool-1-thread-1正在执行。。。
pool-1-thread-1正在执行。。。
```

3. 可变尺寸的线程池

与上面的类似，只是改动下pool的创建方式：

```
//创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。
ExecutorService pool = Executors.newCachedThreadPool();

pool-1-thread-2正在执行。。。
pool-1-thread-1正在执行。。。
pool-1-thread-3正在执行。。。
pool-1-thread-4正在执行。。。
pool-1-thread-5正在执行。。。
```

4. 延迟连接池

```
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

/**
 * Java线程：线程池
 */
public class Test {
    public static void main(String[] args) {
        //创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。
        ScheduledExecutorService pool = Executors.newScheduledThreadPool(2);
        //创建实现了Runnable接口对象，Thread对象当然也实现了Runnable接口
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        Thread t3 = new MyThread();
        Thread t4 = new MyThread();
        Thread t5 = new MyThread();
        //将线程放入池中进行执行
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        //使用延迟执行风格的方法
        pool.schedule(t4, 10, TimeUnit.MILLISECONDS);
        pool.schedule(t5, 10, TimeUnit.MILLISECONDS);
        //关闭线程池
        pool.shutdown();
    }
}
```

```
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + "正在执行。。。");
    }
}
```

```
pool-1-thread-2正在执行。。。
pool-1-thread-1正在执行。。。
pool-1-thread-2正在执行。。。
pool-1-thread-1正在执行。。。
pool-1-thread-2正在执行。。。
```

5. 单任务延迟连接池

在四代码基础上，做改动

```
//创建一个单线程执行程序，它可安排在给定延迟后运行命令或者定期地执行。
ScheduledExecutorService pool = Executors.newSingleThreadScheduledExecutor();

pool-1-thread-1正在执行。。。
pool-1-thread-1正在执行。。。
pool-1-thread-1正在执行。。。
pool-1-thread-1正在执行。。。
pool-1-thread-1正在执行。。。
```

6. 自定义线程池

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

/**
 * Java线程：线程池-自定义线程池
 */
public class Test {
    public static void main(String[] args) {
        // 创建等待队列
        BlockingQueue<Runnable> bqueue = new ArrayBlockingQueue<Runnable>(20);
        // 创建一个单线程执行程序，它可安排在给定延迟后运行命令或者定期地执行。
        ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 2, TimeUnit.MILLISECONDS,
        // 创建实现了Runnable接口对象，Thread对象当然也实现了Runnable接口
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        Thread t3 = new MyThread();
        Thread t4 = new MyThread();
        Thread t5 = new MyThread();
        Thread t6 = new MyThread();
```

线程池创建示例

```
        Thread t7 = new MyThread();
        // 将线程放入池中进行执行
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        pool.execute(t4);
        pool.execute(t5);
        pool.execute(t6);
        pool.execute(t7);
        // 关闭线程池
        pool.shutdown();
    }
}

class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + "正在执行。。。");
        try {
            Thread.sleep(100L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
pool-1-thread-1正在执行。。。
pool-1-thread-2正在执行。。。
pool-1-thread-1正在执行。。。
pool-1-thread-2正在执行。。。
pool-1-thread-1正在执行。。。
pool-1-thread-2正在执行。。。
pool-1-thread-2正在执行。。。
```

任务异步返回结果和取消关闭

携带结果的任务Callable与Future

Executor框架使用Runnable作为其任务的基本表达形式。Runnable是一个有很大局限的抽象，它不能返回一个值或抛出一个受检查的异常。

Callable是更好的抽象：它认为主进入点(即call)将返回一个值，并可能抛出一个异常。

Future表示一个任务的生命周期，并提供了相应的方法来判断是否已经完成或取消，以及获取任务的结果和取消任务等。在Future规范中包含的隐含意义是，任务的生命周期只能前进，不能后退，当某个任务完成后，它将永远停留在“完成”的状态上。

get方法的行为取决于任务的状态（尚未开始、正在运行、已完成）。

如果任务已经完成，那么get方法会立即返回或都抛出一个Exception。

如果任务没有完成，那么get将阻塞并直到任务完成。

如果任务抛出了异常，那么get将该异常封装为ExecutionException并重新抛出。如果get抛出ExecutionException，那么可以通过getCause获得被封装的初始异常。

如果任务被取消，那么get将抛出CanclelationException。

java code：callable与Future接口

```
public interface Callable<V> {
    V call() throws Exception;
}
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException,
    CancellationException;
    V get(long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException,
    CancellationException, TimeoutException;
}
```

实例

可返回值的任务必须实现Callable接口，类似的，无返回值的任务必须Runnable接口。

执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了。

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
```

```
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

/**
 * Java线程：有返回值的线程
 */
public class Test {
    public static void main(String[] args) throws ExecutionException,
            InterruptedException {
        // 创建一个线程池
        ExecutorService pool = Executors.newFixedThreadPool(2);
        // 创建两个有返回值的任务
        Callable c1 = new MyCallable("A");
        Callable c2 = new MyCallable("B");
        // 执行任务并获取Future对象
        Future f1 = pool.submit(c1);
        Future f2 = pool.submit(c2);
        // 从Future对象上获取任务的返回值，并输出到控制台
        System.out.println(">>>" + f1.get().toString());
        System.out.println(">>>" + f2.get().toString());
        // 关闭线程池
        pool.shutdown();
    }
}

class MyCallable implements Callable {
    private String oid;

    MyCallable(String oid) {
        this.oid = oid;
    }

    public Object call() throws Exception {
        return oid + "任务返回的内容";
    }
}
```

```
>>>A任务返回的内容
>>>B任务返回的内容
```

任务取消(todo 中断策略 ...)

如果外部代码能在某个操作正常完成之前将其置入"完成"状态，那么这个操作就可以称为可取消的(cancellable)。

取消的原因有多种：

(1). 用户请求取消。如通过管理接口来发送取消请求。

(2). 有时间限制的操作。例如某个程序需要在有限时间内搜索问题空间，并在这个时间内选择最佳的解决方案。当计时器超时时，需要取消所有正在搜索的任务。

(3). 应用程序事件。例如某个程序对某个问题空间进行分解并搜索，从而使不同的任务可以搜索问题空间中的不同区域。当其中一个任务找到了解决方案时，所有其它仍在搜索的任务都将被取消。

(4). 错误。网页爬虫程序搜索相关的页面，并将页面或摘要数据保存到硬盘。当一个爬虫任务发生错误时(例如磁盘满)，那么所有的搜索任务都会取消，此时可能会记录它们的当前状态，以便稍后重新启动。

(5). 关闭。当一个程序或服务关闭时，必须对正在处理和等待处理的工作执行来某种操作。在平缓的关闭中，当前正在执行的任务将继续执行直到完成，而在立即关闭过程中，当前的任务可能取消。

1. 使用volatile类型的域来保存取消状态

```
public class PrimeGenerator implements Runnable {
    private final List<BigInteger> primes = new ArrayList<BigInteger>();
    private volatile boolean cancelled;

    public void run() {
        BigInteger p = BigInteger.ONE;
        while (!cancelled) {
            p = p.nextProbablePrime();
            synchronized (this) {
                primes.add(p);
            }
        }
    }

    public void cancel() {
        cancelled = true;
    }

    public synchronized List<BigInteger> get() {
        return new ArrayList<BigInteger>(primes);
    }
}
```

2. 通过中断来取消

Thread中的中断方法

```
public class Thread {
    public void interrupt() { ... }
    public boolean isInterrupted() { ... }
    public static boolean interrupted() { ... }
    ...
}

class PrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;
    PrimeProducer(BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }
    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!Thread.currentThread().isInterrupted())
                queue.put(p = p.nextProbablePrime());
        } catch (InterruptedException consumed) {
            /* Allow thread to exit */
        }
    }
    public void cancel() {
        interrupt();
    }
}
```

3. 通过Future来取消

```
public static void timedRun(Runnable r,
                            long timeout, TimeUnit unit)
throws InterruptedException {
    Future<?> task = taskExec.submit(r);
    try {
        task.get(timeout, unit);
    } catch (TimeoutException e) {
        // 接下来任务将被取消
    } catch (ExecutionException e) {
        // 如果在任务中抛出了异常，那么重新抛出异常
        throw launderThrowable(e.getCause());
    }
    finally {
        // 如果任务已经结束，那么取消也不会带来任何影响
        task.cancel(true); // 如果任务正在运行，那么将被终短
    }
}
```

显式锁 Lock

协调共享对象访问的机制：java5之前是synchronized和volatile()，java5增加了ReentrantLock。

Lock与ReentrantLock

Lock接口中定义了一组抽象的加锁机制。Lock 实现提供了比使用 synchronized 方法和语句可获得的更广泛的锁定操作。

ReentrantLock实现了Lock接口，并提供了与synchronized相同的互斥性和内存可见性。

```
public interface Lock {
void lock();
//如果当前线程未被中断，则获取锁定。
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long timeout, TimeUnit unit)
    throws InterruptedException;
    void unlock();
    Condition newCondition();
}
```

使用ReentrantLock来保护对象状态。

```
Lock lock = new ReentrantLock();
lock.lock();
try {
    //相关操作
} finally {
    lock.unlock();  //一定要释放锁
}
```

轮询锁和定时锁

可定时的与可轮询的锁获取模式是由tryLock方法实现的，与无条件的锁获取模式相比，它具有更完善的错误恢复机制。

1. 轮询锁

```
//示例：通过tryLock来避免锁顺序死锁。
//利用tryLock来获取两个锁，如果不能同时获得，那么回退并重新尝试。如果在指定的时间内不能获得所需要的锁，那
 public boolean transferMoney(Account fromAcct,
                              Account toAcct,
                              DollarAmount amount,
                              long timeout,
                              TimeUnit unit)
throws InsufficientFundsException, InterruptedException {
    long fixedDelay = getFixedDelayComponentNanos(timeout, unit);
    long randMod = getRandomDelayModulusNanos(timeout, unit);
    long stopTime = System.nanoTime() + unit.toNanos(timeout);
    while (true) {
        if (fromAcct.lock.tryLock()) {
            try {
                if (toAcct.lock.tryLock()) {
                    try {
                        if (fromAcct.getBalance().compareTo(amount) < 0)
                            throw new InsufficientFundsException();
                        else {
                            fromAcct.debit(amount);
                            toAcct.credit(amount);
                            return true;
                        }
                    }
                    finally {
                        toAcct.lock.unlock();
                    }
                }
            }
            finally {
                fromAcct.lock.unlock();
            }
        }
        if (System.nanoTime() < stopTime)
            return false;
        NANOSECONDS.sleep(fixedDelay + rnd.nextLong() % randMod);
    }
}
```

2. 定时锁

Java 5提供了更灵活的锁工具，可以显式地索取和释放锁。那么在索取锁的时候可以设定一个超时时间，如果超过这个时间还没索取到锁，则不会继续堵塞而是放弃此次任务，示例代码如下：

```
public boolean trySendOnSharedLine(String message,
                                   long timeout, TimeUnit unit)
throws InterruptedException {
long nanosToLock = unit.toNanos(timeout)
- estimatedNanosToSend(message);
    if (!lock.tryLock(nanosToLock, NANOSECONDS))
        return false;
    try {
        return sendOnSharedLine(message);
    } finally {
        lock.unlock();
    }
}
```

可中断的锁

```
public boolean sendOnSharedLine(Stringmessage)
      throws InterruptedException{
    lock.lockInterruptibly(); //如果当前线程未被中断，则获取锁定。
    try{
        return cancellableSendOnSharedLine(message);
    }finally{
        lock.unlock();
    }
}
private boolean cancellableSendOnSharedLine(String message)
    throwsInterruptedException{...}
}
```

读-写锁

ReadWriteLock 维护了一对相关的锁定，一个用于只读操作，另一个用于写入操作。

```
//ReadWriteLock 接口
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

示例：用读-写锁来包装Map

```
public class ReadWriteMap<K, V> {
    private final Map<K, V> map;
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock r = lock.readLock();
    private final Lock w = lock.writeLock();
    public ReadWriteMap(Map<K, V> map) {
        this.map = map;
    }
    public V put(K key, V value) {
        w.lock();
        try {
            return map.put(key, value);
        }
        finally {
            w.unlock();
        }
    }
    // 对 remove(), putAll(), clear()等方法执行相同的操作

    public V get(Object key) {
        r.lock();
        try {
            return map.get(key);
        }
        finally {
            r.unlock();
        }
    }
    // 对其它只读的方法执行相同的操作
}
```

同步

栅栏CyclicBarrier与计数器CountDownLatch

1. 栅栏 CyclicBarrier

栅栏类似于闭锁，它能阻塞一组线程直到某个事件发生。所有线程必须同时到达栅栏位置，才能继续执行。栅栏用于实现一些协议，例如几个家庭成员决定在某个地方集合：”所有人6:00在麦当劳集合，到了以后要等其他人，之后再讨论下一步要做的事”。

CyclicBarrier可以使一定数量的参与方反复地在栅栏位置汇集，它在并行迭代算法中非常有用。

在模拟程序中通常需要栅栏，例如某个步骤中的计算可以并行执行，但必须等到该步骤中的所有计算都执行完毕才能进入下一个步骤。

```
ExecutorService service = Executors.newCachedThreadPool();
// 构造方法里的数字标识有几个线程到达集合地点开始进行下一步工作
final CyclicBarrier cb = new CyclicBarrier(3);
for (int i = 0; i < 3; i++) {
    Runnable runnable = new Runnable() {
        public void run() {
            try {
                Thread.sleep((long) (Math.random() * 10000));
                System.out.println("线程"
                        + Thread.currentThread().getName()
                        + "即将到达集合地点1，当前已有" + cb.getNumberWaiting()
                        + "个已经到达，正在等候");
                cb.await();

                Thread.sleep((long) (Math.random() * 10000));
                System.out.println("线程"
                        + Thread.currentThread().getName()
                        + "即将到达集合地点2，当前已有" + cb.getNumberWaiting()
                        + "个已经到达，正在等候");
                cb.await();
                Thread.sleep((long) (Math.random() * 10000));
                System.out.println("线程"
                        + Thread.currentThread().getName()
                        + "即将到达集合地点3，当前已有" + cb.getNumberWaiting()
                        + "个已经到达，正在等候");
                cb.await();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };
    service.execute(runnable);

}
service.shutdown();
```

```
执行结果：
线程pool-1-thread-2即将到达集合地点1，当前已有0个已经到达，正在等候
```

```
线程pool-1-thread-1即将到达集合地点1，当前已有1个已经到达，正在等候
线程pool-1-thread-3即将到达集合地点1，当前已有2个已经到达，正在等候
线程pool-1-thread-1即将到达集合地点2，当前已有0个已经到达，正在等候
线程pool-1-thread-3即将到达集合地点2，当前已有1个已经到达，正在等候
线程pool-1-thread-2即将到达集合地点2，当前已有2个已经到达，正在等候
线程pool-1-thread-1即将到达集合地点3，当前已有0个已经到达，正在等候
线程pool-1-thread-2即将到达集合地点3，当前已有1个已经到达，正在等候
线程pool-1-thread-3即将到达集合地点3，当前已有2个已经到达，正在等候
```

2. 计数器 CountDownLatch

```
ExecutorService service = Executors.newCachedThreadPool();
final CountDownLatch cdOrder = new CountDownLatch(1);
final CountDownLatch cdAnswer = new CountDownLatch(3);
for (int i = 0; i < 3; i++) {
    Runnable runnable = new Runnable() {
        public void run() {
            try {
                System.out.println("线程"
                        + Thread.currentThread().getName() + "正准备接受命令");
                cdOrder.await();
                System.out.println("线程"
                        + Thread.currentThread().getName() + "已接受命令");
                Thread.sleep((long) (Math.random() * 10000));
                System.out
                        .println("线程"
                                + Thread.currentThread().getName()
                                + "回应命令处理结果");
                cdAnswer.countDown();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };
    service.execute(runnable);
}
try {
    Thread.sleep((long) (Math.random() * 10000));

    System.out.println("线程" + Thread.currentThread().getName()
            + "即将发布命令");
    cdOrder.countDown();
    System.out.println("线程" + Thread.currentThread().getName()
            + "已发送命令，正在等待结果");
    cdAnswer.await();
    System.out.println("线程" + Thread.currentThread().getName()
            + "已收到所有响应结果");
} catch (Exception e) {
    e.printStackTrace();
}
service.shutdown();
```

```
结果：
线程pool-1-thread-1正准备接受命令
```

```
线程pool-1-thread-2正准备接受命令
线程pool-1-thread-3正准备接受命令
线程main即将发布命令
线程main已发送命令，正在等待结果
线程pool-1-thread-1已接受命令
线程pool-1-thread-2已接受命令
线程pool-1-thread-3已接受命令
线程pool-1-thread-1回应命令处理结果
线程pool-1-thread-2回应命令处理结果
线程pool-1-thread-3回应命令处理结果
线程main已收到所有响应结果
```

信号量Semaphore

计数信号量（Counting Semaphore）用来控制同时访问某个特定资源的操作数量，或者同时执行某个指定操作的数量。计数信号量还可以用来实现某种资源池，或者对容器施加边界。

Semaphore中管理着一组虚拟的许可（permit），许可的初始数量可通过构造函数来指定。在执行操作时可以首先获得许可（只要还有剩余的许可），并在使用以后释放许可。如果没有许可，那么acquire将阻塞直到许可（或者被中断或者超时）。release方法将返回一个许可信号量。

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

public class SemaphoreTest {
    public static void main(String[] args) {
        MyPool myPool = new MyPool(20);
        // 创建线程池
        ExecutorService threadPool = Executors.newFixedThreadPool(2);
        MyThread t1 = new MyThread("任务A", myPool, 3);
        MyThread t2 = new MyThread("任务B", myPool, 12);
        MyThread t3 = new MyThread("任务C", myPool, 7);
        // 在线程池中执行任务
        threadPool.execute(t1);
        threadPool.execute(t2);
        threadPool.execute(t3);
        // 关闭池
        threadPool.shutdown();
    }
}

/**
 * 一个池
 */
class MyPool {
    private Semaphore sp; // 池相关的信号量

    /**
     * 池的大小，这个大小会传递给信号量
     *
     * @param size
     *             池的大小
     */
    MyPool(int size) {
        this.sp = new Semaphore(size);
    }

    public Semaphore getSp() {
        return sp;
    }

    public void setSp(Semaphore sp) {
        this.sp = sp;
    }
}
```

```
class MyThread extends Thread {
    private String threadname; // 线程的名称
    private MyPool pool; // 自定义池
    private int x; // 申请信号量的大小

    MyThread(String threadname, MyPool pool, int x) {
        this.threadname = threadname;
        this.pool = pool;
        this.x = x;
    }

    public void run() {
        try {
            // 从此信号量获取给定数目的许可
            pool.getSp().acquire(x);
            // todo：也许这里可以做更复杂的业务
            System.out.println(threadname + "成功获取了" + x + "个许可！");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            // 释放给定数目的许可，将其返回到信号量。
            pool.getSp().release(x);
            System.out.println(threadname + "释放了" + x + "个许可！");
        }
    }
}
```

```
结果：
任务A成功获取了3个许可！
任务B成功获取了12个许可！
任务A释放了3个许可！
任务B释放了12个许可！
任务C成功获取了7个许可！
任务C释放了7个许可！
```

ThreadLocal

1.介绍

ThreadLocal是什么呢？其实ThreadLocal并非是一个线程的本地实现版本，它并不是一个Thread，而是threadlocalvariable(线程局部变量)。也许把它命名为ThreadLocalVar更加合适。ThreadLocal功能非常简单，就是为每一个使用该变量的线程都提供一个变量值的副本，是Java中一种较为特殊的线程绑定机制，是每一个线程都可以独立地改变自己的副本，而不会和其它线程的副本冲突。

从线程的角度看，每个线程都保持一个对其线程局部变量副本的隐式引用，只要线程是活动的并且ThreadLocal实例是可访问的；在线程消失之后，其线程局部实例的所有副本都会被垃圾回收（除非存在对这些副本的其他引用）。

通过ThreadLocal存取的数据，总是与当前线程相关，也就是说，JVM 为每个运行的线程，绑定了私有的本地实例存取空间，从而为多线程环境常出现的并发访问问题提供了一种隔离机制。

ThreadLocal的应用场合，我觉得最适合的是按线程多实例（每个线程对应一个实例）的对象的访问，并且这个对象很多地方都要用到。

ThreadLocal的接口方法： T get() 返回此线程局部变量的当前线程副本中的值。

protected T initialValue() 返回此线程局部变量的当前线程的“初始值”。

void remove() 移除此线程局部变量当前线程的值。

void set(T value) 将此线程局部变量的当前线程副本中的值设置为指定值。

2.ThreadLocal的几种误区

最近由于需要用到ThreadLocal，在网上搜索了一些相关资料，发现对ThreadLocal经常会有下面几种误解：

(1) ThreadLocal是java线程的一个实现 ThreadLocal的确是和java线程有关，不过它并不是java线程的一个实现，它只是用来维护本地变量。针对每个线程，提供自己的变量版本，主要是为了避免线程冲突，每个线程维护自己的版本。彼此独立，修改不会影响到对方。

(2) ThreadLocal是相对于每个session的

ThreadLocal顾名思义，是针对线程。在java web编程上，每个用户从开始到会话结束，都有自己的一个session标识。但是ThreadLocal并不是在会话层上。其实，Threadlocal是独立于用户session的。它是一种服务器端行为，当服务器每生成一个新的线程时，就会维护自己的ThreadLocal。对于这个误解，个人认为应该是开发人员在本地基于一些应用服务器测试的结果。众所周知，一般的应用服务器都会维护一套线程池，也就是说，对于每次访问，并不一定就新生成一个线程。而是自己有一个线程缓存池。对于访问，先从缓存池里面找到已有的线程，如果已经用光，才去新生成新的线程。所以，由于开发人员自己在测试时，一般只有他自己在测，这样服务器的负担很小，这样导致每次访问可能是共用同样一个线程，导致会有这样的误解：每个session有一个ThreadLocal

(3) ThreadLocal是相对于每个线程的，用户每次访问会有新的ThreadLocal

理论上来说，ThreadLocal是的确是相对于每个线程，每个线程会有自己的ThreadLocal。但是上面已经讲到，一般的应用服务器都会维护一套线程池。因此，不同用户访问，可能会接受到同样的线程。因此，在做基于TheadLocal时，需要谨慎，避免出现ThreadLocal变量的缓存，导致其他线程访问到本线程变量

(4) 对每个用户访问，ThreadLocal可以多用

可以说，ThreadLocal是一把双刃剑，用得来的话可以起到非常好的效果。但是，ThreadLocal如果用得不好，就会跟全局变量一样。代码不能重用，不能独立测试。因为，一些本来可以重用的类，现在依赖于ThreadLocal变量。如果在其他没有ThreadLocal场合，这些类就变得不可用了。

个人觉得ThreadLocal用得很好的几个应用场合，值得参考

- 存放当前session用户：quake want的jert
- 存放一些context变量，比如webwork的ActionContext
- 存放session，比如Spring hibernate orm的session

下面来看看ThreadLocal的实现原理（jdk1.5源码）

```
public class ThreadLocal<T> {
    /**
     * ThreadLocals rely on per-thread hash maps attached to each thread
     * (Thread.threadLocals and inheritableThreadLocals).  The ThreadLocal
     * objects act as keys, searched via threadLocalHashCode.  This is a
     * custom hash code (useful only within ThreadLocalMaps) that eliminates
     * collisions in the common case where consecutively constructed
     * ThreadLocals are used by the same threads, while remaining well-behaved
     * in less common cases.
     */
    private final int threadLocalHashCode = nextHashCode();

    /**
     * The next hash code to be given out. Accessed only by like-named method.
     */
    private static int nextHashCode = 0;

    /**
     * The difference between successively generated hash codes - turns
     * implicit sequential thread-local IDs into near-optimally spread
     * multiplicative hash values for power-of-two-sized tables.
     */
    private static final int HASH_INCREMENT = 0x61c88647;

    /**
     * Compute the next hash code. The static synchronization used here
     * should not be a performance bottleneck. When ThreadLocals are
     * generated in different threads at a fast enough rate to regularly
     * contend on this lock, memory contention is by far a more serious
     * problem than lock contention.
     */
    private static synchronized int nextHashCode() {
        int h = nextHashCode;
        nextHashCode = h + HASH_INCREMENT;
        return h;
    }

    /**
     * Creates a thread local variable.
     */
    public ThreadLocal() {
    }

    /**
     * Returns the value in the current thread's copy of this thread-local
     * variable.  Creates and initializes the copy if this is the first time
     * the thread has called this method.
     *
     * @return the current thread's value of this thread-local
     */
    public T get() {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
```

```
            return (T)map.get(this);

        // Maps are constructed lazily.  if the map for this thread
        // doesn't exist, create it, with this ThreadLocal and its
        // initial value as its only entry.
        T value = initialValue();
        createMap(t, value);
        return value;
    }

    /**
     * Sets the current thread's copy of this thread-local variable
     * to the specified value.  Many applications will have no need for
     * this functionality, relying solely on the {@link #initialValue}
     * method to set the values of thread-locals.
     *
     * @param value the value to be stored in the current threads' copy of
     *        this thread-local.
     */
    public void set(T value) {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
    }

    /**
     * Get the map associated with a ThreadLocal. Overridden in
     * InheritableThreadLocal.
     *
     * @param  t the current thread
     * @return the map
     */
    ThreadLocalMap getMap(Thread t) {
        return t.threadLocals;
    }

    /**
     * Create the map associated with a ThreadLocal. Overridden in
     * InheritableThreadLocal.
     *
     * @param t the current thread
     * @param firstValue value for the initial entry of the map
     * @param map the map to store.
     */
    void createMap(Thread t, T firstValue) {
        t.threadLocals = new ThreadLocalMap(this, firstValue);
    }

    .......

    /**
     * ThreadLocalMap is a customized hash map suitable only for
     * maintaining thread local values. No operations are exported
     * outside of the ThreadLocal class. The class is package private to
     * allow declaration of fields in class Thread.  To help deal with
     * very large and long-lived usages, the hash table entries use
```

```
     * WeakReferences for keys. However, since reference queues are not
     * used, stale entries are guaranteed to be removed only when
     * the table starts running out of space.
     */
    static class ThreadLocalMap {

    ........

    }

}
```

可以看到ThreadLocal类中的变量只有这3个int型：

```
private final int threadLocalHashCode = nextHashCode();
private static int nextHashCode = 0;
private static final int HASH_INCREMENT = 0x61c88647;
```

而作为ThreadLocal实例的变量只有 threadLocalHashCode 这一个，nextHashCode 和 HASH_INCREMENT 是ThreadLocal类的静态变量，实际上HASH_INCREMENT是一个常量，表示了连续分配的两个ThreadLocal实例的threadLocalHashCode值的增量，而nextHashCode 的表示了即将分配的下一个ThreadLocal实例的threadLocalHashCode 的值。

可以来看一下创建一个ThreadLocal实例即new ThreadLocal()时做了哪些操作，从上面看到构造函数ThreadLocal()里什么操作都没有，唯一的操作是这句：

```
private final int threadLocalHashCode = nextHashCode();
```

那么nextHashCode()做了什么呢：

```
private static synchronized int nextHashCode() {
    int h = nextHashCode;
       nextHashCode = h + HASH_INCREMENT;
    return h;
}
```

就是将ThreadLocal类的下一个hashCode值即nextHashCode的值赋给实例的threadLocalHashCode，然后nextHashCode的值增加HASH_INCREMENT这个值。

因此ThreadLocal实例的变量只有这个threadLocalHashCode，而且是final的，用来区分不同的ThreadLocal实例，ThreadLocal类主要是作为工具类来使用，那么ThreadLocal.set()进去的对象是放在哪儿的呢？

看一下上面的set()方法，两句合并一下成为

```
ThreadLocalMap map = Thread.currentThread().threadLocals;
```

这个ThreadLocalMap 类是ThreadLocal中定义的内部类，但是它的实例却用在Thread类中：

```
1.    public class Thread implements Runnable {
2.        ......
3.
4.        /* ThreadLocal values pertaining to this thread. This map is maintained
5.         * by the ThreadLocal class. */
6.        ThreadLocal.ThreadLocalMap threadLocals = null;
7.        ......
8.    }
```

再看这句：

```
if (map != null)
    map.set(this, value);
```

也就是将该ThreadLocal实例作为key，要保持的对象作为值，设置到当前线程的ThreadLocalMap 中，get()方法同样大家看了代码也就明白了，ThreadLocalMap 类的代码太多了，我就不帖了，自己去看源码吧。

例子

```
public class SeqGen {
    private static ThreadLocal<Integer> seqNum = new ThreadLocal<Integer>() {
        @Override
        public Integer initialValue() {
            return 0;
        }
    };

    public int getNextNum() {
        seqNum.set(seqNum.get() + 1);
        return seqNum.get();
    }
}

public class TestThread implements Runnable {
    private SeqGen sn;

    public TestThread(SeqGen sn) {
        this.sn = sn;
    }

    public void run() {
        for (int i = 0; i < 3; i++) {
            String str = Thread.currentThread().getName() + "-->"
                    + sn.getNextNum();
            System.out.println(str);
        }
    }
}

public class Test {
    public static void main(String[] args) {
        SeqGen sn = new SeqGen();
        TestThread tt1 = new TestThread(sn);
        TestThread tt2 = new TestThread(sn);
        TestThread tt3 = new TestThread(sn);
        Thread t1 = new Thread(tt1);
        Thread t2 = new Thread(tt2);
        Thread t3 = new Thread(tt3);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

```
输出结果：
Thread-2-->1
Thread-2-->2
Thread-2-->3
Thread-1-->1
Thread-0-->1
Thread-0-->2
```

```
Thread-0-->3
Thread-1-->2
Thread-1-->3
```

资料来源：

(1) 理解ThreadLocal http://blog.csdn.net/qjyong/article/details/2158097

(2) ThreadLocal的几种误区 http://www.blogjava.net/jspark/archive/2006/08/01/61165.html

(3) 正确理解ThreadLocal http://lujh99.iteye.com/blog/103804

并行程序设计模式

Future模式

某一段程序提交了一个请求，期望得到一个答复。在传统单线程环境下，必须等服务程序返回结果后，才能进行其它处理。而在Future模式中，调用方式改为异步，在主调用函数中，原来等待返回结果的时间段，可以处理其它事务。

Future模式的主要参与者

参与者	作用
Main	系统调用，调用Client发出请求
Client	返回Data对象，立即返回FutureData,并开启ClientThread线程装配RealData
Data	返回数据的接口
FutureDate	Future数据，构造很快，但是是一个虚拟的数据，需要装配RealData
RealData	真实数据，但是构造较慢

Future模式代码实现

(1) Data.java

```
public interface Data {
    public String getResult();
}
```

(2) RealData.java

```
public class RealData implements Data {
    protected final String result;
    public RealData(String para) {
        //RealData的构造可能很慢，需要用户等待很久
        StringBuffer sb=new StringBuffer();
        for (int i = 0; i < 10; i++) {
            sb.append(para);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        result=sb.toString();
    }
    public String getResult() {
        return result;
    }
}
```

(3) FutureData.java

```
public class FutureData implements Data {
    protected RealData realdata = null;
    protected boolean isReady = false;
    public synchronized void setRealData(RealData realdata) {
        if (isReady) {
            return;
        }
        this.realdata = realdata;
        isReady = true;
        notifyAll();
    }
    public synchronized String getResult() {
        while (!isReady) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        return realdata.result;
    }
}
```

(4) Client.java

```
public class Client {
    public Data request(final String queryStr) {
        final FutureData future = new FutureData();
        // RealData的构建很慢
        new Thread() {
            public void run() {
                RealData realdata = new RealData(queryStr);
                future.setRealData(realdata);
            }
        }.start();
        return future;
    }
}
```

(5) Main.java

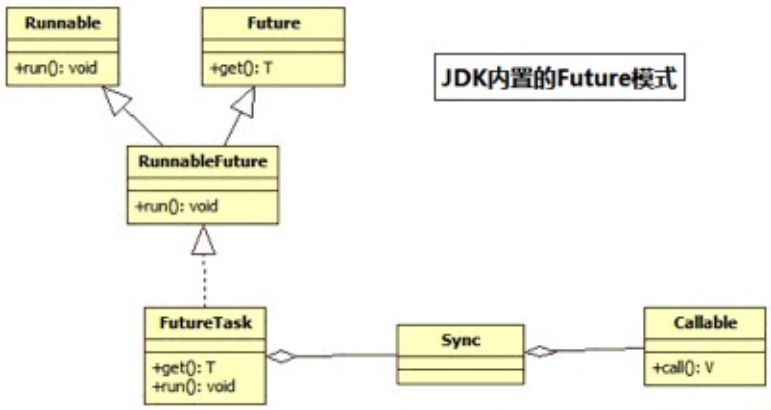
```
public class Main {
    public static void main(String[] args) {
        Client client = new Client();

        Data data = client.request("a");
        System.out.println("请求完毕");
        try {
            //这里可以用一个sleep代替了对其它业务逻辑的处理
            Thread.sleep(2000);
        } catch (InterruptedException e) {
```

```
        }
            //使用真实的数据
        System.out.println("数据 = " + data.getResult());
    }
}
```

```
输出结果：
请求完毕
数据 = aaaaaaaaaa
```

JDK实现



(1) RealData.java

```
import java.util.concurrent.Callable;

public class RealData implements Callable<String> {
    private String para;
    public RealData(String para){
        this.para=para;
    }
    @Override
    public String call() throws Exception {

        StringBuffer sb=new StringBuffer();
        for (int i = 0; i < 10; i++) {
            sb.append(para);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        return sb.toString();
    }
}
```

(2) Main.java

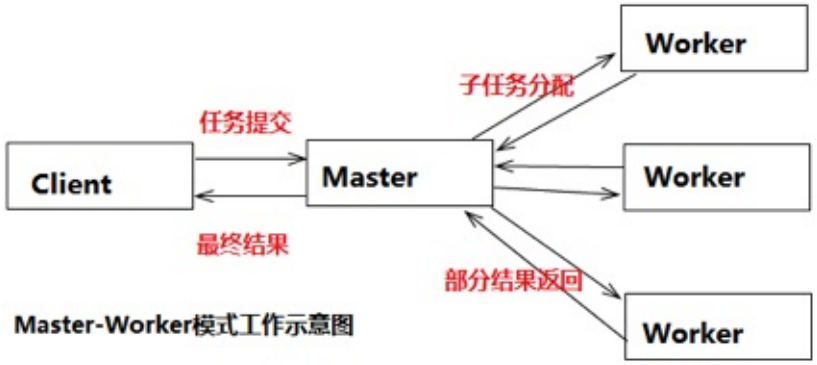
```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.FutureTask;

public class Main {
    public static void main(String[] args) throws InterruptedException, ExecutionExceptio
        //构造FutureTask
        FutureTask<String> future = new FutureTask<String>(new RealData("a"));
        ExecutorService executor = Executors.newFixedThreadPool(1);
        //执行FutureTask，相当于上例中的 client.request("a") 发送请求
        //在这里开启线程进行RealData的call()执行
        executor.submit(future);
        System.out.println("请求完毕");
        try {
        //这里依然可以做额外的数据操作，这里使用sleep代替其他业务逻辑的处理
            Thread.sleep(2000);
        } catch (InterruptedException e) {
        }
        //相当于上例中得data.getContent()，取得call()方法的返回值
        //如果此时call()方法没有执行完成，则依然会等待
        System.out.println("数据 = " + future.get());
    }
}
```

```
输出结果：
请求完毕
数据 = aaaaaaaaaa
```

Master-Worker模式

Master-Worker模式是常用的并行模式。它的核心思想是，系统由两类进程协作工作：Master进程，Worker进程。Master进程负责接收和分配任务，worker进程负责处理子任务。当各个Worker进程将子任务处理完成后，将结果返回给Master进程，由Master进程做归纳和汇总，从而得到系统的最终结果。



代码实现

应用Master-Worker框架，实现计算立方和的应用，并计算1~100的立方和。

计算任务被分解为100个任务，每个任务仅用于计算单独的立方和。Master产生固定个数的worker来处理所有这些子任务。Worker不断地从任务集体集合中取得这些计算立方和的子任务，并将计算结果返回给Master.Master负责将所有worker累加，从而产生最终的结果。在计算过程中，Master和Worker的运行也是完全异步的,Master不必等所有的Worker都执行完成后，就可以进行求和操作。Master在获得部分子任务结果集时，就可以开始对最终结果进行计算，从而进一步提高系统的并行度和吞吐量。

(1) Master.java

```
import java.util.HashMap;
import java.util.Map;
import java.util.Queue;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentLinkedQueue;

public class Master {
    // 任务队列
    protected Queue<Object> workQueue = new ConcurrentLinkedQueue<Object>();
    // Worker线程队列
    protected Map<String, Thread> threadMap = new HashMap<String, Thread>();
    // 子任务处理结果集
    protected Map<String, Object> resultMap = new ConcurrentHashMap<String, Object>();

    // 是否所有的子任务都结束了
    public boolean isComplete() {
        for (Map.Entry<String, Thread> entry : threadMap.entrySet()) {
            if (entry.getValue().getState() != Thread.State.TERMINATED) {
                return false;
```

```
            }
        }
        return true;
    }

    // Master的构造，需要一个Worker进程逻辑，和需要的Worker进程数量
    public Master(Worker worker, int countWorker) {
        worker.setWorkQueue(workQueue);
        worker.setResultMap(resultMap);
        for (int i = 0; i < countWorker; i++)
            threadMap.put(Integer.toString(i),
                    new Thread(worker, Integer.toString(i)));
    }

    // 提交一个任务
    public void submit(Object job) {
        workQueue.add(job);
    }

    // 返回子任务结果集
    public Map<String, Object> getResultMap() {
        return resultMap;
    }

    // 开始运行所有的Worker进程，进行处理
    public void execute() {
        for (Map.Entry<String, Thread> entry : threadMap.entrySet()) {
            entry.getValue().start();
        }
    }
}
```

(2) Worker.java

```
import java.util.Map;
import java.util.Queue;

public class Worker implements Runnable {
    // 任务队列，用于取得子任务
    protected Queue<Object> workQueue;
    // 子任务处理结果集
    protected Map<String, Object> resultMap;

    public void setWorkQueue(Queue<Object> workQueue) {
        this.workQueue = workQueue;
    }

    public void setResultMap(Map<String, Object> resultMap) {
        this.resultMap = resultMap;
    }

    // 子任务处理的逻辑，在子类中实现具体逻辑
    public Object handle(Object input) {
        return input;
    }

    @Override
```

```
    public void run() {
        while (true) {
            // 获取子任务
            Object input = workQueue.poll();
            if (input == null)
                break;
            // 处理子任务
            Object re = handle(input);
            // 将处理结果写入结果集
            resultMap.put(Integer.toString(input.hashCode()), re);
        }
    }
}
```

(3) TestMasterWorker.java

```
import java.util.Map;
import java.util.Set;

import org.junit.Test;

public class TestMasterWorker {

    public class PlusWorker extends Worker {
        public Object handle(Object input) {
            Integer i = (Integer) input;
            return i * i * i;
        }
    }

    @Test
    public void testMasterWorker() {
        Master m = new Master(new PlusWorker(), 5);
        for (int i = 0; i < 100; i++) {
            m.submit(i);
        }
        m.execute();
        int re = 0;
        Map<String, Object> resultMap = m.getResultMap();
        while (resultMap.size() > 0 || !m.isComplete()) {
            Set<String> keys = resultMap.keySet();
            String key = null;
            for (String k : keys) {
                key = k;
                break;
            }
            Integer i = null;
            if (key != null) {
                i = (Integer) resultMap.get(key);
            }
            if (i != null) {
                re += i;
            }
            if (key != null) {
                resultMap.remove(key);
            }
        }
```

```
        System.out.println("testMasterWorker:" + re);
    }

    @Test
    public void testPlus() {
        int re = 0;
        for (int i = 0; i < 100; i++) {
            re += i * i * i;
        }
        System.out.println("testPlus:" + re);
    }

}
```

```
执行输出结果：
testMasterWorker:24502500
testPlus:24502500
```

Guarded-Suspension模式

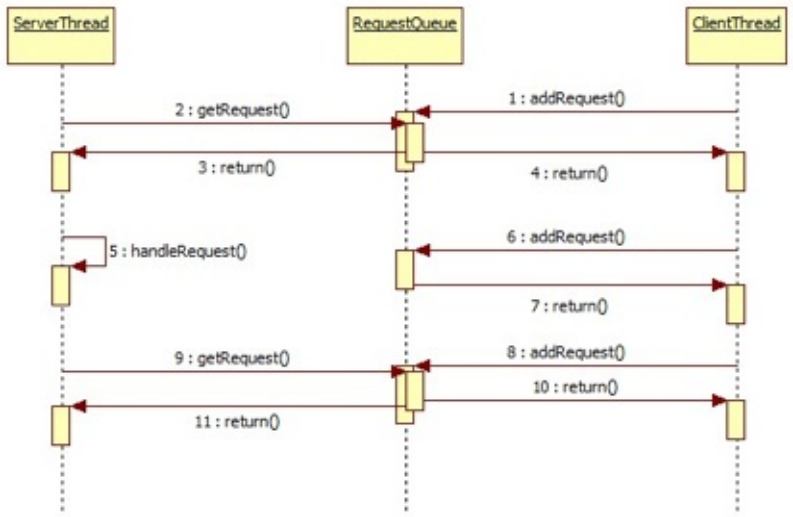
Guarded-Suspension意为保护暂停，其核心思想是仅当服务进程准备好，才提供服务。设 想一种场景，服务器可能会在很短时间内承受大量的客户端请求，客户端请求的数量可能超过服务器本身的即时处理能力，而服务器程序又不能丢弃任何一个客户请求。此时，最佳的处理方案莫过于让客户端请求进行排队，由服务端程序一个接一个处理。这样既保征了所有的客户端请求均不丢失，同时也避免了服务器由于同时处理太多的请求而崩溃。

(1) Guarded-Suspension结构

Guarded-Suspension模式的主要角色

角色	作用
Request	表示客户端请求
RequestQueue	用于保存客户端请求队列
ClientThread	客户端进程
ServerThread	服务端进程

ClientThread负责不断的发起请求，并将请求对象放入请求队列。ServerThread则根据其自身的状态，在有能力处理请求时，从RequestQueue中提取请求对象加以处理，系统的工作流程如图：



代码实现

(1) Request.java

```
public class Request {
    private String name;
    public Request(String name) {
        this.name = name;
    }
```

```
    public String getName() {
        return name;
    }
    public String toString() {
        return "[ Request " + name + " ]";
    }
}
```

(2) RequestQueue.java

```
import java.util.LinkedList;

public class RequestQueue {
    private LinkedList<Request> queue = new LinkedList<Request>();

    public synchronized Request getRequest() {
        while (queue.size() == 0) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        return (Request) queue.remove();
    }

    public synchronized void addRequest(Request request) {
        queue.add(request);
        notifyAll(); // 通知getRequest()方法
    }
}
```

(3) ClientThread.java

```
public class ClientThread extends Thread {
    private RequestQueue requestQueue;
    public ClientThread(RequestQueue requestQueue, String name) {
        super(name);
        this.requestQueue = requestQueue;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            Request request = new Request("RequestID:" + i+" Thread_Name:"+Thread.current
            System.out.println(Thread.currentThread().getName() + " requests " + request)
            requestQueue.addRequest(request);
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
            }
            System.out.println("ClientThread Name is:"+Thread.currentThread().getName());
        }
        System.out.println(Thread.currentThread().getName()+" request end");
    }
}
```

(4) ServerThread.java

```
public class ServerThread extends Thread {
    private RequestQueue requestQueue;
    public ServerThread(RequestQueue requestQueue, String name) {
        super(name);
        this.requestQueue = requestQueue;
    }
    public void run() {
        while (true) {
            final Request request = requestQueue.getRequest();
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + " handles  " + request)

        }
    }
}
```

(5) Main.java

```
public class Main {
    public static void main(String[] args) {
        RequestQueue requestQueue = new RequestQueue();
        for (int i = 0; i < 10; i++) {
            new ServerThread(requestQueue, "ServerThread" + i).start();
        }
        for (int i = 0; i < 10; i++) {
            new ClientThread(requestQueue, "ClientThread" + i).start();
        }
    }
}
```

```
执行结果：
ClientThread4 requests [ Request RequestID:0 Thread_Name:ClientThread4 ]
…省略…
ClientThread Name is:ClientThread1
ClientThread Name is:ClientThread4
ClientThread Name is:ClientThread2
ClientThread4 requests [ Request RequestID:1 Thread_Name:ClientThread4 ]
ClientThread1 requests [ Request RequestID:1 Thread_Name:ClientThread1 ]
ClientThread2 requests [ Request RequestID:1 Thread_Name:ClientThread2 ]
…省略…
ServerThread1 handles  [ Request RequestID:0 Thread_Name:ClientThread8 ]
ClientThread Name is:ClientThread3
ClientThread3 request end
ServerThread3 handles  [ Request RequestID:0 Thread_Name:ClientThread3 ]
ServerThread7 handles  [ Request RequestID:0 Thread_Name:ClientThread2 ]
```

```
ClientThread Name is:ClientThread5
ClientThread Name is:ClientThread1
ClientThread1 request end
ClientThread Name is:ClientThread6
ServerThread0 handles  [ Request RequestID:0 Thread_Name:ClientThread5 ]
ClientThread6 request end
ClientThread5 request end
ServerThread8 handles  [ Request RequestID:1 Thread_Name:ClientThread1 ]
ServerThread9 handles  [ Request RequestID:1 Thread_Name:ClientThread4 ]
…省略…
ServerThread6 handles  [ Request RequestID:9 Thread_Name:ClientThread3 ]
```

不变模式immutable

当对象产生后，不发生任何改变。

适合场景：

(1) 当对象被创建后，其内部状态和数据不再发生变化。

(2) 对象需要共享、被多线程频繁访问。

```
public final class Product {
    private final String  no;
    private final String name;
    private final double price;

    public Product(String no, String name, double price) {
        super();
        this.no = no;
        this.name = name;
        this.price = price;
    }

    public String getNo() {
        return no;
    }
    public String getName() {
        return name;
    }
    public double getPrice() {
        return price;
    }
}
```

请参考：基于阻塞队列BlockingQueue的生产者、消费者模式

并发控制方法

并发框架Amino

参考资料

(1) 《Java并发编程实战》 （美）Brian Goetz等 著 童云兰等 译

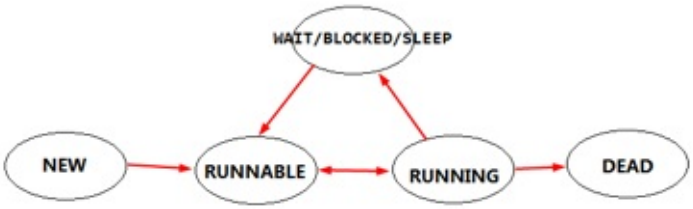
(2) java5线程 Callable与Future的应用 http://blog.csdn.net/itm_hadf/article/details/7479274

(3) Java线程：大总结 http://lavasoft.blog.51cto.com/62575/222742

(4) CyclicBarrier与CountDownLatch、栅栏与计数器 http://www.iteye.com/topic/713053

(5) 《Java程序性能优化》 葛一鸣

1. 线程状态



(1) NEW：线程对象已经创建，还没有在其上调用start()方法。

(2) RUNNABLE：当线程有资格运行，但调度程序还没有把它选定为运行线程时线程所处的状态。当start()方法调用时，线程首先进入可运行状态。在线程运行之后或者从阻塞、等待或睡眠状态回来后，也返回到可运行状态。

(3) RUNNING：线程调度程序从可运行池中选择一个线程作为当前线程时线程所处的状态。这也是线程进入运行状态的唯一一种方式。

(4) 等待/阻塞/睡眠状态(WAIT/ BLOCKED/SLEEP)：线程有资格运行时它所处的状态。实际上这个三状态组合为一种，其共同点是：线程仍旧是活的，但是当前没有条件运行。换句话说，它是可运行的，但是如果某件事件出现，他可能返回到RUNNABLE状态。

(5) DEAD(TERMINATED ?)：当线程的run()方法完成时就认为它死去。这个线程对象也许是活的，但是，它已经不是一个单独执行的线程。线程一旦死亡，就不能复生。 如果在一个死去的线程上调用start()方法，会抛出java.lang.IllegalThreadStateException异常。