1. JUC队列

单线程编程时我们会使用到一些集合类，如ArrayList等。ArrayList不是线程安全的，Vector通过加snychronized锁的方式实现线程安全，但是效率很低，同时Collections.synchronizedList(List<T> list)方法也能够将ArrayList转换为线程安全的。这些方法都很低效，都是串行执行的。下面介绍阻塞队列与非阻塞队列。

1. 非阻塞队列ConcurrentLinkedQueue

一个基于链接节点的无边界的线程安全队列，遵循队列的FIFO原则，采用CAS算法实现的，理论上的无边界队列，不过还是要受限与硬件。要尽量避免使用该队列的size()方法，因为它是由链表实现的，会遍历一遍队列，非常慢。当我们要检查队列中是否还有元素时，用poll()弹出队首元素或者用isEmpty()检查。

这个队列只能保证我们在对这个队列进行单个操作时是线程安全的，如果要同时进行多个操作就需要自己来加锁保证线程的安全。

1. 阻塞队列BlockingQueue
2. BlockingQueue

阻塞队列接口，在某些情况下对阻塞队列的访问可能会造成阻塞，被阻塞的情况有以下两种：

队列满了的情况下进行入队操作；

队列空的情况下进行出队操作；

阻塞队列接口对插入，删除，获取元素操作提供了不同的方法，如下所示：

| **操作** | **抛出异常** | **特殊值** | **阻塞** | **超时** |
| --- | --- | --- | --- | --- |
| 插入 | add(e) | offer(e) | put(e) | offer(e, time, unit) |
| 移除 | remove() | poll() | take() | poll(time, unit) |
| 检查 | element() | peek() | 不可用 | 不可用 |

1. ArrayBlockingQueue
2. 概述

这是阻塞队列接口的实现类，具体是由数组实现的有界阻塞队列，遵循FIFO原则。该队列是有界且固定的，因为是数组实现的，在构造时大小已经决定了。该队列对于等待的生产者和消费者线程的排序可以选择公平策略和非公平策略。

1. 实现

ArrayBlockingQueue继承AbstractQueue，实现BlockingQueue接口，内部使用可重入锁ReentrantLock+Condition实现多线程环境的并发操作，在该类中有以下参数：

items: 一个定长数组，维护ArrayBlockingQueue中的元素；

takeIndex: 队首的位置；

putIndex: 队尾的位置；

count: 元素个数

lock: 锁，出队和入队都需要锁，两个步骤公用一个锁；

public class ArrayBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, Serializable {

   private static final long serialVersionUID = -817911632652898426L;

   final Object[] items;

   int takeIndex;

   int putIndex;

   int count;

   // 重入锁

   final ReentrantLock lock;

   // notEmpty condition

   private final Condition notEmpty;

   // notFull condition

   private final Condition notFull;

   transient ArrayBlockingQueue.Itrs itrs;

}

1. 实例

通过经典的生产者和消费者实例来测试数组阻塞队列。

package thread10;  
  
import java.util.Random;  
import java.util.concurrent.ArrayBlockingQueue;  
import java.util.concurrent.TimeUnit;  
  
public class ArrayBlockingQueueDemo {  
 private static ArrayBlockingQueue<Integer> *queue* = new ArrayBlockingQueue<>(5, true);  
  
 public static void main(String[] args) {  
 Thread t1 = new Thread(new ProducerTask());  
 Thread t2 = new Thread(new ConsumerTask());  
 t1.start();  
 t2.start();  
 }  
  
 //生产者线程  
 static class ProducerTask implements Runnable{  
  
 private Random rnd = new Random();  
  
 @Override  
 public void run() {  
 try {  
 while(true){  
 int value = rnd.nextInt(100);  
 *queue*.put(value);  
 System.*out*.println("生产者" +value);  
 TimeUnit.*MILLISECONDS*.sleep(100);  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 //消费者  
 static class ConsumerTask implements Runnable{  
 @Override  
 public void run(){  
 try {  
 while(true){  
 Integer value = *queue*.take();  
 System.*out*.println("消费者"+ value);  
 TimeUnit.*MILLISECONDS*.sleep(10);  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

1. LinkedBlockingQueue

基于链表的有界阻塞队列，队列中的锁是分离的，入队的锁是putLock，出队的锁是takeLock。数组阻塞队列直接将枚举对象插入或者移除，而链表阻塞队列需要将枚举对象转换为Node后再进行插入或移除操作，这样会影响性能。

1. PriorityBlockingQueue
2. 概述

优先级队列，始终保证出队的元素时优先级最高的元素，并且可以定制优先级的规则。内部使用二叉堆，通过二叉树最小堆算法来维护内部的数组，这个数组是可扩容的，若当前的元素个数大于最大容量时会通过算法扩容，扩容时其他的线程是不能进行出队操作的，实际上使用了先释放锁，然后通过CAS保证同时只有一个线程扩容。

1. 特点

既然优先队列中需要排序，那么存储的对象就必须是可比较的对象，必须实现Comparable接口，不允许空值。优先队列的头是排序最小的元素，如果多个对象有同样的排序，则随机选取其中一个，返回队列的头对象。优先队列的大小是不受限制的，但在创建时可以指定初始大小，当向队列中增加元素时，队列大小会自动增加。

1. 实例

创建一个自定义的对象类实现Comparable接口，传入PriorityBlockingQueue中。

package thread11;  
  
import java.util.Random;  
import java.util.concurrent.PriorityBlockingQueue;  
  
public class PriorityBlockingQueueDemo {  
 public static void main(String[] args) throws InterruptedException {  
 PriorityBlockingQueue<User> queue = new PriorityBlockingQueue<>();  
 PriorityDemo demo = new PriorityDemo(queue);  
  
 for(int i = 0; i < 3; ++i){  
 new Thread(demo).start();  
 }  
  
 Thread.*sleep*(100);  
 User u = queue.poll();  
 while(u != null){  
 System.*out*.println("优先级是" + u.getPriority() + "," + u.getUsername());  
 u = queue.poll();  
 }  
 }  
  
 static class PriorityDemo implements Runnable{  
  
 PriorityBlockingQueue queue;  
 Random r = new Random();  
  
 public PriorityDemo(PriorityBlockingQueue queue){  
 this.queue = queue;  
 }  
 @Override  
 public void run() {  
 for(int i = 0; i < 3; ++i){  
 User user = new User();  
 user.setPriority(r.nextInt(100));  
 user.setUsername("张三" + i);  
 queue.add(user);  
 }  
 }  
 }  
  
 static class User implements Comparable<User>{  
  
 private Integer priority;  
 private String username;  
  
 @Override  
 public int compareTo(User user) {  
 return this.priority.compareTo(user.getPriority());  
 }  
  
 public Integer getPriority() {  
 return priority;  
 }  
  
 public void setPriority(Integer priority) {  
 this.priority = priority;  
 }  
  
 public String getUsername() {  
 return username;  
 }  
  
 public void setUsername(String username) {  
 this.username = username;  
 }  
 }  
}

1. synchronousQueue
2. 概述

同步队列，实际上不是一个真正的队列，因为它不会为队列中的元素维护存储空间，它维护的是一组线程，这些线程等待着将元素加入或移除队列。由于该队列没有存储功能，队列中至多有一个元素，因此put入队操作会一直阻塞，直到有take出队操作执行。

同步队列是用于直接交付操作的，所谓的直接交付指的是直接将任务交到消费者的手上，而不是放到队列中等待消费者来消费。因为同步队列中没有存储能力，所以消费者能够直接得到生产者生产的任务，降低了将数据从生产者移动到消费者的延迟。也正是因为没有存储能力，同步队列只适合于存在多个消费者且总是有一个消费者处于准备状态的情况，这样生产者的任务才能够被消费者消费。

1. 特点

阻塞队列，每个put必须等待一个take。

线程安全的，阻塞的。

不允许使用null元素

1. put和offer

put()方法将元素放入队列后，会一直等待直到有其他线程将这个元素取走；

offer()方法将元素放入队列后，直接返回，如果有其他线程取走了元素则返回true，否则返回false;

1. take和poll

take()取出并且remove掉queue中的element，取不到会一直等待；

poll()取出并且remove掉queue中的element，取到了就返回element，否则返回null。

1. JUC线程池
2. 优势
3. 降低资源消耗，通过重用已创建的线程来降低线程创建和销毁的性能损失；
4. 提高响应速度，任务无需创建线程，直接执行；
5. 提高线程的可管理性，统一管理，分配，调优和监控。
6. 线程池状态

AtomicInteger类型的变量ctl中定义了线程池中的任务数量和线程池的状态两个信息，共32位，其中高3位表示线程池状态，低29位表示线程池中的任务数量。线程池有五种状态，如下所示：

1. RUNNING

处于该状态下线程池能够接受新的任务，以及对新添加的任务进行处理。

1. SHUTDOWN

处于该状态的线程池不可以接受新的任务，但是可以对已添加的任务进行处理。通过调用shutdown()函数进入该状态。

1. STOP

处于该状态的线程池不接受新的任务，也不处理已添加的任务，直接中断正在处理的任务。通过调用shutdownNow()函数进入该状态，这个函数尽量不要使用，因为会中断正在执行的线程。

1. TIDYING

当所有的任务终止，ctl记录的任务数量为0时，线程池变为TIDYING状态。之后会执行钩子函数terminated()，该函数在ThreadPoolExecutor类中是空的，若用户想在线程池变为TIDYING时进行相应的处理，可以重载terminated()函数。

1. TERMINATED

线程池彻底终止。

1. 构造方法

Java的线程池支持主要是通过ThreadPoolExecutor类实现的，它的构造方法有七个参数，如下所示：

1. corePoolSize

线程池中核心线程的数量（线程池的基本大小），当提交一个任务时，线程池会新建一个线程来执行任务，直到当前线程数等于corePoolSize。如果调用了prestartAllCoreThreads()方法，线程池会提前创建并启动所有基本线程。

1. maximumPoolSize

线程池中允许的最大线程数，若线程池的阻塞队列满了后，还有任务提交，只要当前的线程数小于maximumPoolSize，就会创建线程来执行任务。

1. keepAliveTime

线程的存活时间，由于线程的创建和销毁是需要代价的，在执行完任务后不会立即销毁，而是存活一段时间，默认情况下，该参数只有在线程数大于corePoolSize时才会生效，即只有大于核心线程数时才会销毁多余的线程。

1. unit

keepAliveTime的单位，TimeUnit。

1. workQueue

当任务超过了maximumPoolSize时，就会将等待的任务保存在 workQueue中，等待的任务必须实现Runnable接口，队列有以下几种：

ArrayBlockingQueue: 基于数组的有界阻塞队列，FIFO；

LinkedBlockingQueue: 基于链表的有界阻塞队列，FIFO；

PriorityBlockingQueue: 具有优先级别的阻塞队列，

SynchronousQueue: 不存储元素的阻塞队列；

1. threadFactory

用于设置创建线程的工厂，作用是提供创建线程的功能的线程工厂，通过newThread()方法提供创建线程的功能，该方法创建的线程都是非守护线程，而且线程优先级都是默认优先级。

1. handler

RejectedExecutionHandler，线程池的拒绝策略，所谓拒绝策略是指将任务添加到线程池中时，若次线程池已经饱和，且阻塞队列也已经满了，那么线程池就会选择一种拒绝策略来拒绝该任务。

线程池提供了四种拒绝策略：

AbortPolicy: 直接抛出异常，默认策略；

CallerRunsPolicy: 用调用者所在的线程来执行任务；

DiscardOlestPolicy: 丢弃阻塞队列中最靠前的任务，并执行当前任务；

DiscardPolicy: 直接丢弃任务；

当然，我们也可以实现自己的拒绝策略，如日志记录等，只要实现RejectedExecutionHandler接口即可。

1. 四种线程池

除了使用ThreadPoolExecutor根据实际情况创建线程池以外，Executor框架也提供了三种线程池 ，都可以通过工具类Executors来创建。还有一种线程池ScheduledThreadPoolExecutor，相当于提供了“延迟”和“周期 执行”功能的ThreadPoolExecutor。

1. FixedThreadPool
2. 特点

FixedThreadPool是复用固定数量的线程来处理一个共享的无边界队列，在创建该线程池时，指定了线程数量，corePoolSize和maximumPoolSize都设置为了该数字，则当线程池中的线程数量等于corePoolSize时，如果继续提交任务，就会被添加到阻塞队列workQueue中，而workQueue使用的是LinkedBlockingQueue，但是没有设置范围，则是Integer.MAX\_VALUE范围，相当于无界队列。

1. 定义

该线程池的定义如下：

public static ExecutorService newFixedThreadPool(int nThreads) {

   return new ThreadPoolExecutor(nThreads, nThreads,

                                 0L, TimeUnit.MILLISECONDS,

                                 new LinkedBlockingQueue<Runnable>());

}

从定义中可以看出，线程的存活时间为0，因为corePoolSize和maximumPoolSize是相等的，因此不会有线程数大于corePoolSize的情况，因此也不会销毁线程。当该线程池中的线程数小于corePoolSize时，会创建新的线程来执行其他的任务；当线程数量到达了corePoolSize时，不同的任务会复用该线程池中的固定线程，如果有等待的任务则会进入阻塞队列中等待。

1. 实例

创建固定3个线程的线程池，通过5个线程提交任务，可以看到线程池中的线程是复用的。

package thread12;  
  
import thread.Run;  
  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
public class ThreadPool {  
 public static void main(String[] args) throws InterruptedException {  
 ExecutorService exec = Executors.*newFixedThreadPool*(3);  
 for(int i = 0 ; i < 5; ++i){  
 exec.execute(new Demo());  
 Thread.*sleep*(10);  
 }  
 exec.shutdown();  
  
 }  
  
 static class Demo implements Runnable{  
  
 @Override  
 public void run() {  
 String name = Thread.*currentThread*().getName();  
 for(int i = 0 ; i < 2; ++i){  
 System.*out*.println(name + ":" + i);  
 }  
 }  
 }  
}

1. SingleThreadExecutor
2. 特点

只是用单个工作线程来执行一个无边界的队列。

1. 定义

该线程池的定义如下：

public static ExecutorService newSingleThreadExecutor() {

   return new FinalizableDelegatedExecutorService

      (new ThreadPoolExecutor(1, 1,

                               0L, TimeUnit.MILLISECONDS,

                               new LinkedBlockingQueue<Runnable>()));

}

可以看到，corePoolSize和maximumPoolSize都设置为1，因此只有单个线程执行任务。同时线程存活时间设置为0，因为不需要没有空闲线程需要销毁。使用的阻塞队列时LinkedBlockingQueue无界的链表阻塞队列。

如果单个线程在执行任务过程中终止了，则新的线程会代替它来执行后续的线程。

1. CachedThreadPool
2. 特点

该线程池会根据任务的需要，在线程可用时，重用之前构造好的池中线程，否则创建新的线程。

1. 定义

该线程池的定义如下所示：

public static ExecutorService newCachedThreadPool() {

   return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,

                                 60L, TimeUnit.SECONDS,

                                 new SynchronousQueue<Runnable>());

}

可以看到，corePoolSize设置为0，maximumPoolSize设置为Integer.MAX\_VALUE，表示能够创建无穷多的线程，只要任务提交就会加入到阻塞队列中，只要线程池中没有线程，就会创建线程来执行任务。

线程的存活时间设置60秒，即当空闲的线程等待新任务的时间超过60秒时会被终止，阻塞队列采用 SynchronousQueue，这是一个不能存储元素的阻塞队列。

因此，该线程池适合执行大量短生命周期的异步任务，不同的任务会重用之前构造的线程，同时线程池即使在长时间空闲也不会消耗任何资源，因为不论是线程还是阻塞队列中都没有存储资源。

不过，这种线程池也存在问题，如果提交任务的速度远远大于线程池的处理速度，那么线程池就会不断地创建新的线程来执行任务，这样会消耗大量的CPU和内存资源，因此使用该线程池需要控制并发的任务数。

1. ScheduledThreadPool
2. Timer和TimerTask的问题

在实现周期线程和延迟调度时可以使用Timer和TimerTask，但是它们存在一些问题：

Timer在执行定时任务时只会创建一个线程，如果存在多个任务，且任务时间过长，超过两个任务的间隔时间会发生问题；

如果TimerTask抛出RuntimeException，Timer会停止所有任务的运行；

Timer执行周期任务时依赖于系统时间，如果系统时间出现问题则执行上也会有问题。

1. 实现

ScheduledThreadPoolExecutor，继承了ThreadPoolExecutor，且实现了ScheduledExecutorService接口，相当于提供了延迟和周期执行功能的ThreadPoolExecutor。它可以安排在给定的延迟时间后执行命令，或者定期执行命令，灵活性优于Timer。

   public ScheduledThreadPoolExecutor(int corePoolSize,

                                      ThreadFactory threadFactory,

                                      RejectedExecutionHandler handler) {

       super(corePoolSize, Integer.MAX\_VALUE, 0, NANOSECONDS,

               new DelayedWorkQueue(), threadFactory, handler);

  }

可以看到，该线程池内部通过继承类ThreadLocalExecutor的构造方法实现的，指定了corePoolSize，maximumPoolSize设置为最大值。使用的阻塞队列变为了DelayedWorkQueue，这个队列类似于延时队列和优先级队列。在执行定时任务时，每个任务的执行时间都不同，所以该队列的任务就是按照执行时间的升序排列，执行时间距离当前时间越近的任务排在靠前的位置，保证按照执行时间执行。

1. 实例

创建三个延迟任务，检验延迟效果。

package thread13;  
  
import thread.Run;  
  
import java.util.Date;  
import java.util.concurrent.Executors;  
import java.util.concurrent.ScheduledExecutorService;  
import java.util.concurrent.TimeUnit;  
  
public class ThreadPool2 {  
 public static void main(String[] args) throws InterruptedException {  
 ScheduledExecutorService scheduledExecutorService = Executors.*newScheduledThreadPool*(2);  
 System.*out*.println("程序开始 "+ new Date());  
 scheduledExecutorService.schedule(new Task(), 0, TimeUnit.*SECONDS*);  
 scheduledExecutorService.schedule(new Task(), 1, TimeUnit.*SECONDS*);  
 scheduledExecutorService.schedule(new Task(), 5, TimeUnit.*SECONDS*);  
  
 Thread.*sleep*(5000);  
 }  
  
 static class Task implements Runnable{  
  
 @Override  
 public void run() {  
 try {  
 String name = Thread.*currentThread*().getName();  
 System.*out*.println(name + " 开始 " + new Date());  
 Thread.*sleep*(1000);  
 System.*out*.println(name+ "结束 " + new Date());  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}