线程池

线程池的优点
1.重用线程池中的线程，避免因为线程的创建和销毁所带来的的性能开销
2.能有效控制线程池的最大并发数，避免大量线程之间因互相抢夺系统资源而导致的阻塞现象。
3.能够对线程进行简单的管理，并提供定时执行以及指向间隔循环执行等功能。

线程池的创建

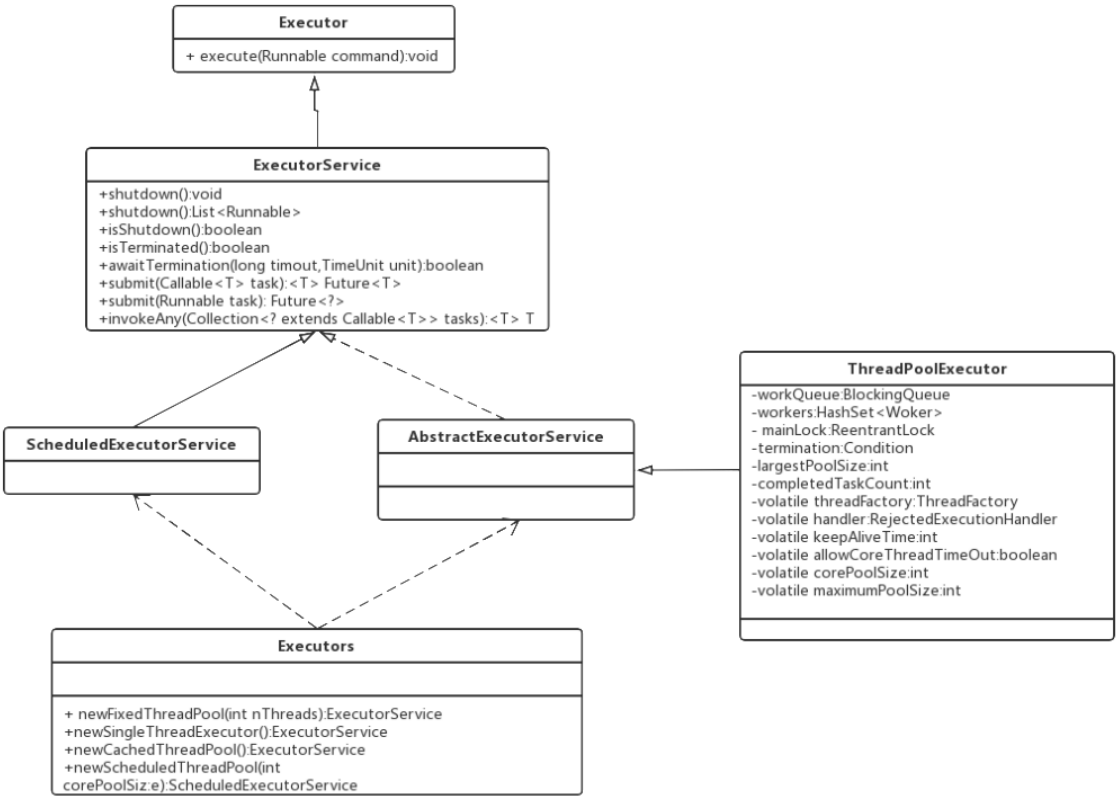
Java SE 的java.util.concurrent包中的执行器(Executor)将为你管理Thread对象，从而简化了并发编程。Executor在客户端和任务执行之间提供了一个间接层；与客户端直接执行任务不同，这个中介对象将执行任务。Executor允许你管理异步任务的执行，而无须显示的管理线程的生命周期。Executor在Java中启动任务的优选方法。

```
package ThreadDemo;

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

public class CachedThreadPool {
    public static void main(String[] args) {
        class MyRunnable implements Runnable{
            private int a = 2;

            public void run() {
                synchronized (this){
                    for (int i = 0; i <5 ; i++) {
                        if(this.a>0){
                            System.out.println(Thread.currentThread().getName()+"a的
值"+this.a--);
                        }
                    }
                }
            }
        }
        ExecutorService service = Executors.newCachedThreadPool();
        for (int i = 0; i <5 ; i++) {
            service.execute(new MyRunnable());
        }
        service.shutdown();

    }
}
```

运行结果：

```
D:\JAVA\jdk1.8.0_171\bin\java.exe ...
pool-1-thread-1a的值2
pool-1-thread-1a的值1
pool-1-thread-5a的值2
pool-1-thread-4a的值2
pool-1-thread-3a的值2
pool-1-thread-3a的值1
pool-1-thread-2a的值2
pool-1-thread-4a的值1
pool-1-thread-5a的值1
pool-1-thread-2a的值1
```

ExcutorService是一个接口，并继承了接口Executor.而Executors是一个工具类，下面来看看它们之间的UML图：



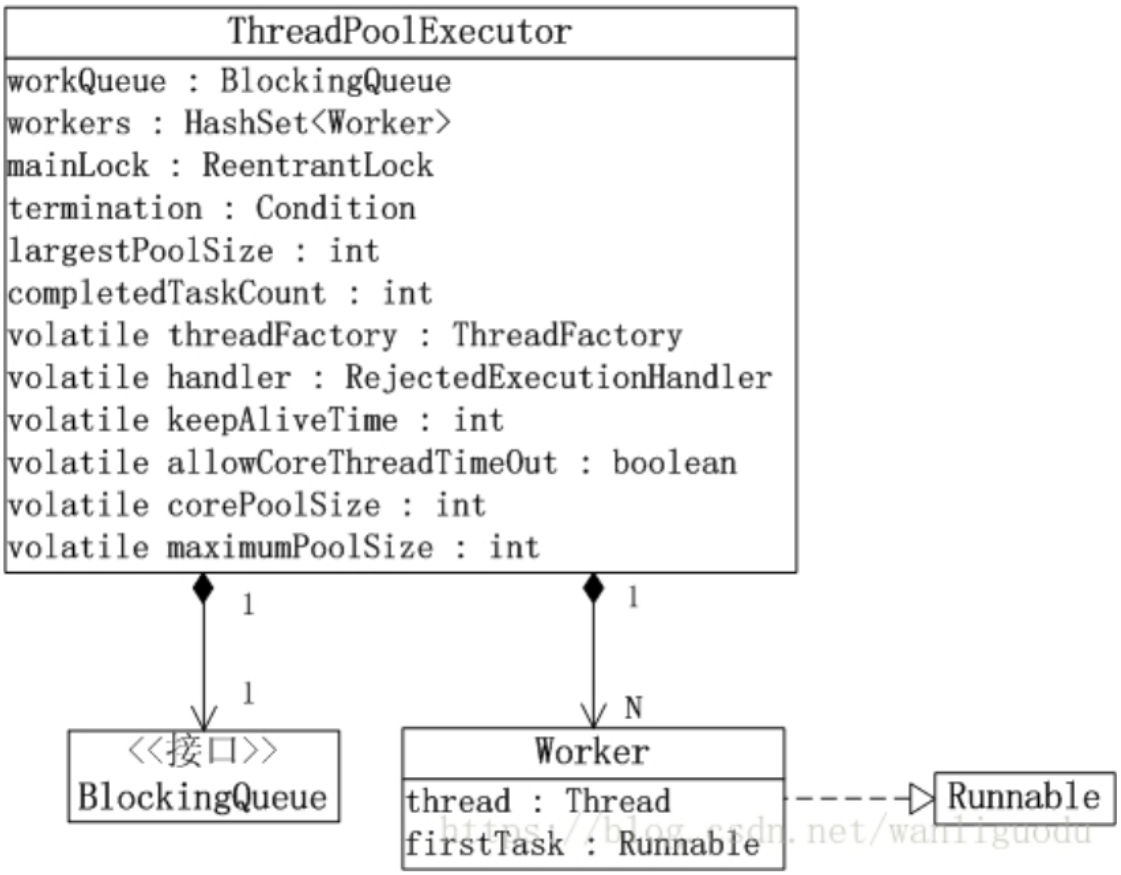
其中最为重要的是ThreadPoolExecutor类和Excutors中的四类方法，下面来分析一下：

1.1.ThreadPoolExecutor

1).ThreadPoolExecutor简介

ThreadPoolExecutor是线程池类。对于线程池，可以通俗的将它理解为“存放一定数量的一个线程集合”，线程池允许若干个线程同时运行，同时运行的线程数量就是线程池的容量。当添加到线程池

中的线程超过它的容量时，会有一部分线程阻塞等待，线程池会通知相应的调度策略和拒绝策略，对添加到线程池中的线程进行管理。



下面是ThreadPoolExecutor类中比较典型的部分代码：

```
public class ThreadPoolExecutor extends AbstractExecutorService {
    // 阻塞队列。
    private final BlockingQueue<Runnable> workQueue;
    // 互斥锁
    private final ReentrantLock mainLock = new ReentrantLock();
    // 线程集合。一个Worker对应一个线程。
    private final HashSet<Worker> workers = new HashSet<Worker>();
    //  “终止条件”，与“mainLock”绑定。
    private final Condition termination = mainLock.newCondition();
    // 线程池中线程数量曾经达到过的最大值。
    private int largestPoolSize;
    // 已完成任务数量
    private long completedTaskCount;
    // ThreadFactory对象，用于创建线程。
    private volatile ThreadFactory threadFactory;
    // 拒绝策略的处理句柄。
    private volatile RejectedExecutionHandler handler;
    // 保持线程存活时间。
    private volatile long keepAliveTime;

    private volatile boolean allowCoreThreadTimeOut;
    // 核心池大小
```

```
    private volatile int corePoolSize;
    // 最大池大小
    private volatile int maximumPoolSize;

    //构造方法
    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,
                              ThreadFactory threadFactory,
                              RejectedExecutionHandler handler) {
        if (corePoolSize < 0 ||
            maximumPoolSize <= 0 ||
            maximumPoolSize < corePoolSize ||
            keepAliveTime < 0)
            throw new IllegalArgumentException();
        if (workQueue == null || threadFactory == null || handler == null)
            throw new NullPointerException();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
}
```

对一些关键的变量进行介绍：

workers
workers是HashSet类型，它是一个Worker集合，而一个Worker对应一个线程，也就是说线程池通过workers包含了“一个线程集合”。当Worker对应的线程池启动时，它会执行线程池中的任务；当执行完一个任务后它会从线程池的阻塞队列中取出一个阻塞的任务来继续运行。workers的作用是：线程池通过它来实现了“允许多个线程同时运行”。

workQueue
workQueue是BlockingQueue类型，它是一个阻塞队列，当线程池中的线程超过它的容量的时候，线程会进入阻塞队列进行阻塞等待。workQueue的作用是：让线程池实现了阻塞功能。
mainLock
mainLock是互斥锁，通过mainLock实现了对线程池的互斥访问

corePoolSize和maximumPoolSize
corePoolSize是“核心池大小”，maximumPoolSize是“最大池大小”。它们的作用是:调整“线程池中实际运行的线程的数量”。

例如：当新任务提交给线程池时(通过execute方法)。
1.如果线程池中运行的线程数量小于corePoolSize;则仅当阻塞队列满时才创建新线程。
2.如果线程池中运行的线程数量大于corePoolSIze,但小于maximumPoolSize,则仅当阻塞队列满时才创建新线程。
3.如果corePoolSize和maximumPoolSize相同，则创建了固定大小的线程池，如果maximumPoolSize设置为基本的无界值(如，Integer.MAX_VALUE),则允许线程池适应任意数量的并

发任务，在大多数情况下，核心池大小和最大池大小的值在创建线程池时设置的，但是，也可以使用setCorePoolSIze(int)和setMaximumPoolSIze(int)进行动态更改。

poolSize
poolSIze是当前线程池的实际大小，即线程池中任务的数量

allowCoreThreadTimeOut和keepAliveTime
allowCoreThreadTimeOut表示是否允许“线程在空闲状态下，仍然能够存活”
keepAliveTime表示线程池处于空闲状态的时候，超过keepAliveTime时间之后，空闲的线程会被终止。

threadFactory
threadFactory是ThreadFactory对象，它是一个线程工厂类，即“线程池通过ThreadFActory创建线程”

handler
handler是RejectedExecutionHandler类型，它是“线程池拒绝策略”的句柄，也就是说“当某任务添加到线程池中，而线程池拒绝任务时，线程池会通过handler进行相应的处理”

线程池的分类

ExecutorService是Executor直接的扩展接口，也是最常用的线程池接口，我们通常用到的线程池定时任务线程池都是它的实现类。
Executors有4个静态方法可以调用，每个方法都有不同特性，它们都是直接或间接的通过配置ThreadPoolExecutor来实现自己的功能特性，这四类线程池分别是EixedThreadPool,CacheThreadPool,ScheduleThreadPool以及SingleTheradExecutor.

1).FixedThreadPool
通过Executor的newFixedThreadPool方法来创建，它是一种线程数固定的线程池，当线程池处于空闲状态时，它们并不会被回收，除非线程池被关闭了，当所有的线程都处于活动状态时，新任务都会处于等待状态，直到有线程空闲出来，由于FixedThreadPool只有核心线程，并且这些核心线程没有超时机制，另外任务队列也是没有大小限制的。

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

2).SingleThreadExecutor

通过Executor的newSingleThreadExecutor方法来创建，这类线程池内部只有一个核心线程，它确保所有的任务都在同一个线程中按顺序执行，SingleThreadExecutor的意义在于统一所有的外界任务到一个线程中，这使得这些任务之间不需要处理线程同步的问题。

```
 public static ExecutorService newSingleThreadExecutor() {
        return new FinalizableDelegatedExecutorService
            (new ThreadPoolExecutor(1, 1,
```

```
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>()));
    }
```

3).ScheduledThreadPool
通过Executors的newScheduledPool方法来创建，它的核心线程数量是固定的，而非核心线程数量是没有限制的，并且当非核心线程闲置是会被立即回收。ScheduledThreadPool这类线程主要用于执行定时任务和具有固定周期的重复任务。

```
    public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
            return new ScheduledThreadPoolExecutor(corePoolSize);
        }

     public ScheduledThreadPoolExecutor(int corePoolSize) {
        super(corePoolSize, Integer.MAX_VALUE, 0, TimeUnit.NANOSECONDS,
              new DelayedWorkQueue());
    }
```

4).CachedThreadPool

通过Executors的newCacheTHreadPool方法来创建，它是一种线程数量不定的线程池，它只有非核心线程，并且其最大线程数为Integer.MAX_VALUE,由于Integer.MAX_VALUE是一个很大的数，实际上就相当于最大线程数可以任意大。当线程池中的线程都是处于活动状态时，线程池会创建新的线程来处理新任务，否则就会利用空闲的线程来处理新任务。线程池中的空闲线程都有超时机制，这个超时长为60秒，超过60秒闲置线程就会被回收。

和FixThreadPool不同的是，CacheTHreadPool的任务队列其实相当于一个空集合，这将导致任何任务都会立即被执行，因为在这种场景下SynchronousQUeue是无法插入任务的，SynchronousQueue是一个非常特殊的队列，在很多情况下可以把它简单理解为一个无法存储元素的队列，由于它在实际中较少使用，这里就不赘述了。

从CachedThreadPool的特性来看，这类线程池比较适合执行大量的耗时较少的任务。当整个线程池都处于闲置状态时，线程池中的线程都会被终止，这个时候CacheThreadPool之中实际上是没有任何线程，它几乎是不占用任何系统资源的。

```
   public static ExecutorService newCachedThreadPool() {
        return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                      60L, TimeUnit.SECONDS,
                                      new SynchronousQueue<Runnable>());
    }
```

线程池中任务的添加

1.execute()

```
public void execute(Runnable command) {
    // 如果任务为null，则抛出异常。
    if (command == null)
```

```
        throw new NullPointerException();
    // 获取ctl对应的int值。该int值保存了"线程池中任务的数量"和"线程池状态"信息
    int c = ctl.get();
    // 当线程池中的任务数量 < "核心池大小"时，即线程池中少于corePoolSize个任务。
    // 则通过addWorker(command, true)新建一个线程，并将任务(command)添加到该线程中；然后，启动该线程从而执行任务。
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    // 当线程池中的任务数量 >= "核心池大小"时，
    // 而且，"线程池处于允许状态"时，则尝试将任务添加到阻塞队列中。
    if (isRunning(c) && workQueue.offer(command)) {
        // 再次确认“线程池状态”，若线程池异常终止了，则删除任务；然后通过reject()执行相应的拒绝策略的内容。
        int recheck = ctl.get();
        if (! isRunning(recheck) && remove(command))
            reject(command);
        // 否则，如果"线程池中任务数量"为0，则通过addWorker(null, false)尝试新建一个线程，新建线程对应的任务为null。
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    // 通过addWorker(command, false)新建一个线程，并将任务(command)添加到该线程中；然后，启动该线程从而执行任务。
    // 如果addWorker(command, false)执行失败，则通过reject()执行相应的拒绝策略的内容。
    else if (!addWorker(command, false))
        reject(command);
}
```

注：execute()的作用是将任务添加到线程池中执行。它分为三种情况：

1).如果“线程池中任务数量” < “核心池大小” 时，即线程池中少于corePoolSize个任务；此时就新建一个线程，并将该任务添加到线程中进行执行。

2).如果“线程池中任务数量” >= “核心池大小” ，并且“线程池是允许状态”；此时，则将任务添加到阻塞队列中阻塞等待。在该情况下，会再次确认“线程状态”，如果“第2次读到的线程池状态”和“第1次读到的线程次状态”不同，则从阻塞队列中删除该任务。

3).如果非上述的两种情况，就会尝试新建一个线程，并将该任务添加到线程中进行执行。如果执行失败，则通过reject()拒绝该任务。

2.addWorker()

addWorker()源码如下：

```
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    // 更新"线程池状态和计数"标记，即更新ctl。
    for (;;) {
        // 获取ctl对应的int值。该int值保存了"线程池中任务的数量"和"线程池状态"信息
        int c = ctl.get();
```

```
        // 获取线程池状态。
        int rs = runStateOf(c);

        // 有效性检查
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
               firstTask == null &&
               ! workQueue.isEmpty()))
            return false;

        for (;;) {
            // 获取线程池中任务的数量。
            int wc = workerCountOf(c);
            // 如果"线程池中任务的数量"超过限制，则返回false。
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            // 通过CAS函数将c的值+1。操作失败的话，则退出循环。
            if (compareAndIncrementWorkerCount(c))
                break retry;
            c = ctl.get();  // Re-read ctl
            // 检查"线程池状态"，如果与之前的状态不同，则从retry重新开始。
            if (runStateOf(c) != rs)
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
        }
    }

    boolean workerStarted = false;
    boolean workerAdded = false;
    Worker w = null;
    // 添加任务到线程池，并启动任务所在的线程。
    try {
        final ReentrantLock mainLock = this.mainLock;
        // 新建Worker，并且指定firstTask为Worker的第一个任务。
        w = new Worker(firstTask);
        // 获取Worker对应的线程。
        final Thread t = w.thread;
        if (t != null) {
            // 获取锁
            mainLock.lock();
            try {
                int c = ctl.get();
                int rs = runStateOf(c);

                // 再次确认"线程池状态"
                if (rs < SHUTDOWN ||
                    (rs == SHUTDOWN && firstTask == null)) {
                    if (t.isAlive()) // precheck that t is startable
                        throw new IllegalThreadStateException();
                    // 将Worker对象(w)添加到"线程池的Worker集合(workers)"中
                    workers.add(w);
                    // 更新largestPoolSize
                    int s = workers.size();
                    if (s > largestPoolSize)
                        largestPoolSize = s;
                    workerAdded = true;
```

```
                }
            } finally {
                // 释放锁
                mainLock.unlock();
            }
            // 如果"成功将任务添加到线程池"中，则启动任务所在的线程。
            if (workerAdded) {
                t.start();
                workerStarted = true;
            }
        }
    } finally {
        if (! workerStarted)
            addWorkerFailed(w);
    }
    // 返回任务是否启动。
    return workerStarted;
}
```

addWorker()的作用是将firstTask添加到线程池中，并启动该任务。当core为true是，则以corePoolSize为界限，若“线程池中已有任务数量” >= corePoolSize ，那么返回false；当core为false时，则以maximumPoolSize为界限，若“线程池中已有任务数量” >= maximumPoolSize，则返回false。addWorker()方法会先通过for循环不断尝试更新 ctl状态，ctl 记录了“线程池中任务数量和线程池状态”。更新成功后，在通过try模块来将任务添加到线程池中，并启动任务所在的线程。

从addWorker()方法中，我们可以发现：线程池在添加任务时，会创建任务对应的Worker对象，而一个Worker对象包含了一个Thread对象。通过将Worker对象添加到“线程的workers集合中”，从而实现将任务添加到线程池中。通过启动Worker对应的Thread线程，则执行该任务。

3.submit()

submit()实际上也是通过调用execute()实现的，源码如下：

```
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<Void> ftask = newTaskFor(task, null);
    execute(ftask);
    return ftask;
}
```

线程池的关闭

在ThreadPoolExecutor类中的shutdown()方法源码为：

```
public void shutdown() {
    final ReentrantLock mainLock = this.mainLock;
    // 获取锁
    mainLock.lock();
    try {
        // 检查终止线程池的“线程”是否有权限。
```

```
            checkShutdownAccess();
            // 设置线程池的状态为关闭状态。
            advanceRunState(SHUTDOWN);
            // 中断线程池中空闲的线程。
            interruptIdleWorkers();
            // 钩子函数，在ThreadPoolExecutor中没有任何动作。
            onShutdown(); // hook for ScheduledThreadPoolExecutor
        } finally {
            // 释放锁
            mainLock.unlock();
        }
        // 尝试终止线程池
        tryTerminate();
    }
```

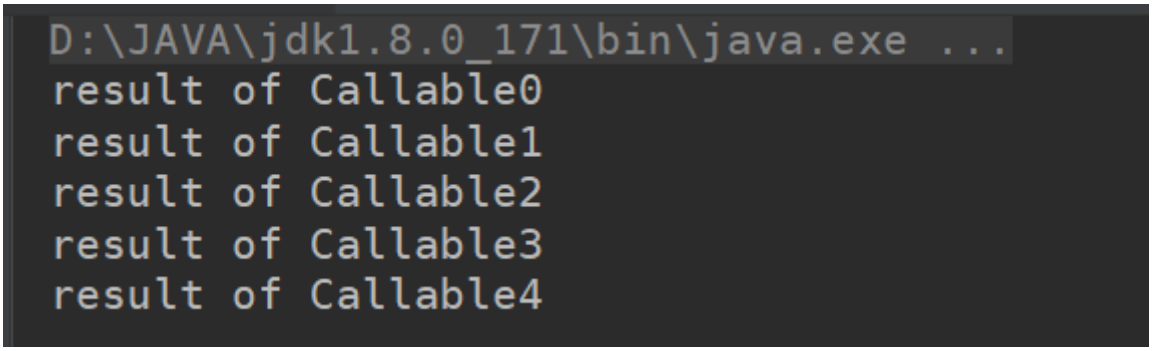
使用Callable

Runnable是执行工作的独立任务，但是它不返回任何值。如果你希望任务在完成时能够返回一个值，那么可以实现Callable接口而不是Runnable接口。在Java SE 5 中引入的Callable是一种具有类型参数的泛型，它的类型参数表示的是从方法call()中返回的值，并且必须使用ExecutorService.submit()方法调用它，下面是简单示例：

```
package ThreadDemo;

import java.util.ArrayList;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CallableDemo {
    public static void main(String[] args) {
        class TaskWithResult implements Callable<String>{

            private int id;
            public TaskWithResult(int id){
                this.id = id;
            }

            public String call() throws Exception {
                return "result of Callable"+id;
            }
        }

        ExecutorService exec = Executors.newCachedThreadPool();
        ArrayList<Future<String>> results = new ArrayList<Future<String>>();
        for (int i = 0; i <5 ; i++) {
            results.add(exec.submit(new TaskWithResult(i)));
        }
        for (Future<String> result : results) {
            try{
                System.out.println(result.get());
            }catch (Exception e){
                e.printStackTrace();
```

```
            }finally {
                exec.shutdown();
            }
        }
    }
}
```

运行结果：



submit()方法会产生Future对象，它用Callable返回结果的特定类型进行了参数化。

解决共享资源竞争

在Java SE5 的java.util.concurrent类库中还包含有定义在java.util.concurrent.locks中的显式的互斥机制。Lcok对象必须被显示的创建、锁定、和释放。因此，它与内间的锁形式相比，代码缺乏优雅性。但是，对于解决某些类型的问题，它更加灵活。下面是用Lock写以解决共享资源的示例：

```
public class LockAndUnLock {

    static Lock lock = new ReentrantLock();//新建锁

    public static void main(String[] args) {

        new Thread("A"){
            public void run() {
                Thread.yield();//当前线程的让步，加快线程切换
                numPrint();

            };
        }.start();

        new Thread("B"){
            public void run() {
                Thread.yield();//当前线程的让步，加快线程切换
                numPrint();
            };
        }.start();

    }

    private static void numPrint(){
        lock.lock();
        try{
```

```
                for(int i=0;i<10;i++){
                    Thread.sleep(100);
                    System.out.println("当前线程"+Thread.currentThread().getName()+"："+i);
                }
            }catch(Exception e){

            }finally{
                lock.unlock();
            }
        }

    }
```

可以看出一个被互斥调用的锁，并使用lock()和unlock()方法在numPrint()内创建了临界资源。当你在使用Lock对象时，将这里的所示的惯用法内部化是很重要的：紧接着的对lock()的调用，你必须放再finally子句中带有unlock()的try-finally语句中。尽管try-finally所需的代码比synchronized关键字要多，但是这也代表了显示的Lock对象的优点之一。如果在使用synchronized关键字，某些事务失败了，那么就会抛出一个异常。但是你没有机会去做任何清理工作，以维护系统使其处于良好状态。有了显示的Lock对象，你就可以使用finally子句将系统维护在正确的状态了。

大体上，当你使用synchronized关键字时，需要写的代码量更少，并且用户错误出现的可能性也会降低，因此通常只有在解决特殊问题时，才使用显示的Lock对象。

以上，先对线程池做了大体的介绍，然后我会逐步介绍JUC中的原子类、线程安全的集合、锁以及深层次剖析线程池原理。