任务执行

6.1 在线程中执行任务

- 清晰的任务边界。在理想情况下，各个任务之间是相互独立的：任务并不依赖于其他任务的状态，结果或边界效应。
  独立性有助于实现并发。
- 大多数服务器应用程序都提供了一种自然的任务边界选择方式，以独立的客户请求为边界。
- 串行地执行任务效率太低
- 显示的为任务创建线程。线程开销高，系统资源有限，系统性能不会随着线程数量增长而无线增长。

6.2 Executor框架

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。

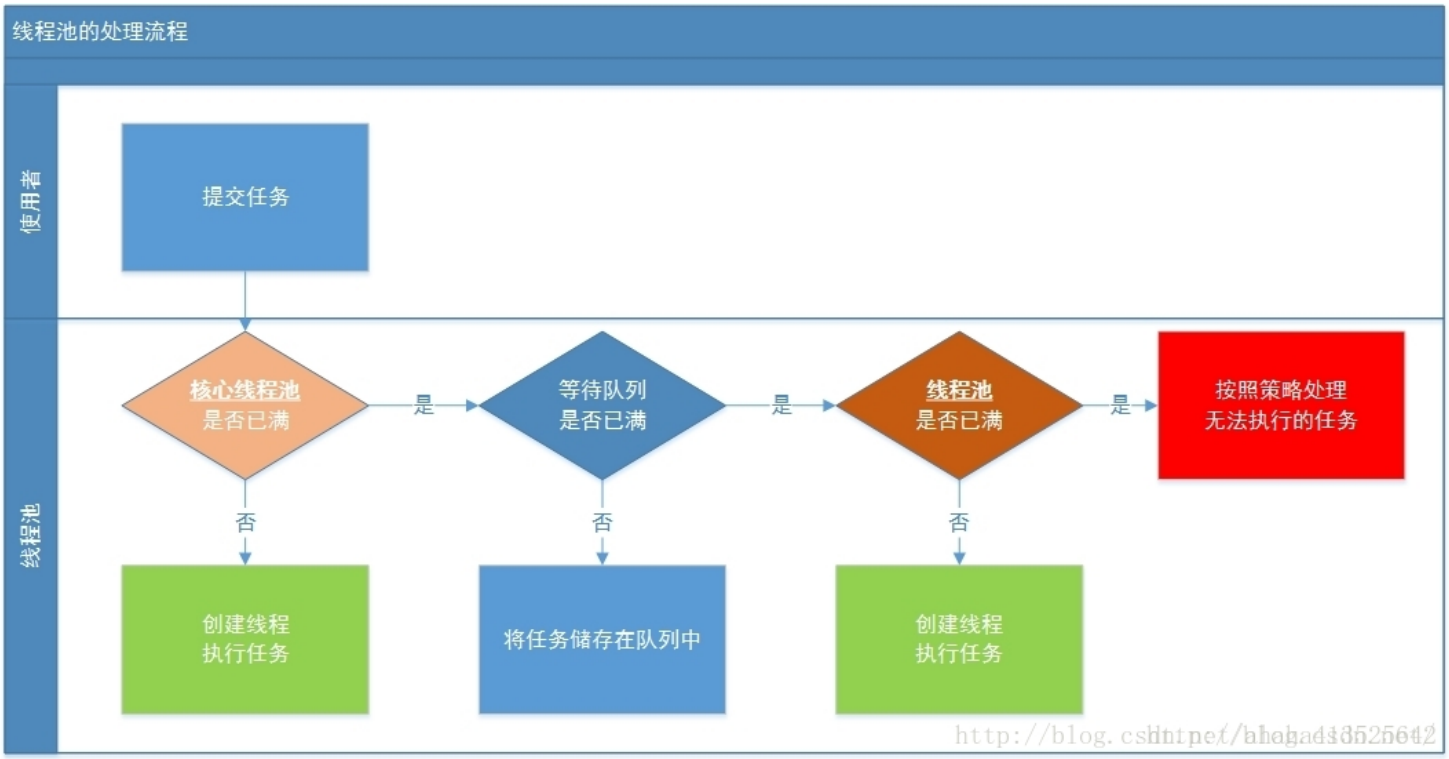
任务是一组逻辑工作单元，而线程则是使任务异步执行的机制。

执行策略

- 在什么线程中执行任务
- 任务按照什么顺序执行（FIFO,LIFO,优先级）
- 有多少个任务可并发执行
- 在队列中有多少个任务在等待执行
- 如果系统由于过程而需要拒绝一个任务，应该选择哪一个任务？另外，如何通知应用程序有任务被拒绝?
- 在执行一个任务之前或之后，应该进行哪些动作（可重载的方法beforeExecute(wt, task), afterExecute(task, thrown)）

线程池

线程池，是指管理一组同构工作线程的资源池。线程池是与工作队列（Work Queue）密切相关的，其中在工作队列中保存了所有等待执行的任务。工作者线程（Worker Thread）的任务很简单，从工作队列中获取一个任务，执行任务，然后返回线程池并等待下一个任务。



如果在执行过程中发生异常等导致线程销毁，线程池也会重新创建一个线程来执行后续的任务

newFixedThreadPool

- 构造函数

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

- 核心线程数和最大线程数一致，阻塞队列用的是有界的（int的最大值）链表结构的linkedBlockingQueue
- 任务数量超过核心线程数后会堆积在linkedBlockingQueue中等待核心线程执行

SingleThreadExecutor

```
public static ExecutorService newSingleThreadExecutor() {
       return new FinalizableDelegatedExecutorService
           (new ThreadPoolExecutor(1, 1,
                                   0L, TimeUnit.MILLISECONDS,
                                   new LinkedBlockingQueue<Runnable>()));
   }

   public static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory) {
       return new FinalizableDelegatedExecutorService
           (new ThreadPoolExecutor(1, 1,
                                   0L, TimeUnit.MILLISECONDS,
                                   new LinkedBlockingQueue<Runnable>(),
                                   threadFactory));
   }
```

- 基本约等于newFixedThreadPool(1)
- FinalizableDelegatedExecutorService对ExecutorService进行了一个包装，减少暴露一些方法并加了一个finalize方法调用shundown()

CachedThreadPool

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

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

- 核心线程为0，最大线程数为Integer.MAX_VALUE，过期时间为60s，意味着线程池中不存在核心线程，全是60s过期的临时线程
- 阻塞队列是SynchronousQueue，不会存储任务，所以提交任务时线程池会启动一个临时线程执行

ScheduledThreadPool

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

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

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

   public ScheduledThreadPoolExecutor(int corePoolSize,
                                      ThreadFactory threadFactory,
                                      RejectedExecutionHandler handler) {
       super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
             new DelayedWorkQueue(), threadFactory, handler);
   }
```

- 队列是DelayedWorkQueue, 所以支持定时任务和周期任务
- Timer因为是单线程，所以会出现问题。
- Timer 在执行所有定时任务时只会创建一个线程。如果某个任务的执行时间过长，那么将破坏其他 TimerTask 的定时准确性。
- 如果 TimerTask 抛出了一个未检查的异常，那么 Timer 将表现出糟糕的行为。
  Timer 线程并不捕获异常，因此当 TimerTask 抛出未检查的异常时将终止定时线程。
  这种情况下，Timer 也不会恢复线程的执行，而是错误地认为整个 Timer 都被取消了。

ForkJoinPool

它的主要特点是可以充分利用多核CPU , 可以把一个任务拆分为多个子任务，这些子任务放在不同的处理器上并行执行，当这些子任务执行结束后再把这些结果合并起来，这是一种分治思想。
双端队列+工作密取

CompletionService: Executor 与 BlockingQueue

ExecutorCompletionService 的实现很简单。在构造函数中创建一个 BlockingQueue 来保存计算完成的结果。当计算完成时，调用 Future-Task 中的 done 方法。当提交给某个任务时，该任务将首先包装为一个 QueueingFuture，这是 FutureTask 的一个子类，然后再改写子类的 done 方法，并将结果放入 BlockingQueue 中。

```
//      6-14  由ExecutorCompletionService使用的QueueingFuture类
private class QueueingFuture<V> extends FutureTask<V> {
QueueingFuture(Callable<V> c) { super(c); }
QueueingFuture(Runnable t, V r) { super(t, r); }

protected void done() {
    completionQueue.add(this);
}
}
```

- FutureTask 完成之后会调用done()方法
- invokeAny() 方法中用到了CompletionService类

invokeAll() 和invokeAny()方法