



黑客派

爬虫实战 - 使用线程池抓取

作者: [san](#)

原文链接: <https://hacpai.com/article/1533478812733>

来源网站: [黑客派](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

```
<h2 id="前言">前言</h2>
<blockquote>
<p>以前在写爬虫的时候，要么是直接使用 HttpClient 循环去抓取，这样有个弊端，就是人物是阻的，后一个任务必须等待上一个任务结束。后面使用了 Thread 线程去执行任务，有多少个任务就开多少条线程，但是频繁线程的创建与销毁又造成了不必要的资源浪费。于是我们用到了线程池，同时使到 Guava 的并发类 ListenableFuture。</p>
</blockquote>
<h2 id="单线程">单线程</h2>
<p>直接开一条线程去爬虫</p>
<pre><code class="highlight-chroma">new Thread() -&gt; {
    //爬虫任务
}>;</code></pre>
<h2 id="多线程">多线程</h2>
<p>当然想爬多个页面也可以使用这样的方式开多条线程。。。</p>
<pre><code class="highlight-chroma">while (条件) {
    new Thread() -&gt; {
        //爬虫任务
    };
}>;</code></pre>
<p>那你就 out 太多了， new Thread 的弊端如下：</p>
<p>a. 每次 new Thread 新建对象性能差。<br> b. 线程缺乏统一管理，可能无限制新建线程，相互间竞争，及可能占用过多系统资源导致死机或 oom。<br> c. 缺乏更多功能，如定时执行、定期执行线程中断。</p>
<p>相比 new Thread， Java 提供的四种线程池的好处在于：</p>
<p>a. 重用存在的线程，减少对象创建、消亡的开销，性能佳。<br> b. 可有效控制最大并发线程数提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。<br> c. 提供定时执行、定期执行、单程、并发数控制等功能。</p>
<h2 id="线程池">线程池</h2>
<h3 id="无执行结果">无执行结果</h3>
<p>无返回结果是实现了 Runnable 接口，重写的 run 方法返回值为 void，线程执行完毕无返回结果同时线程池 execute 方法也是无返回值的。</p>
<pre><code class="highlight-chroma">//创建条三个线程的线程池
ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);
//提交3个任务
for (int i = 0; i < 3; i++) {
    fixedThreadPool.execute(new Thread() -&gt; {
        //爬虫任务
    });
}
}>;</code></pre>
<h3 id="有执行结果">有执行结果</h3>
<p>线程实现 Callable 接口，重写 call 方法，并且在类上定义返回值泛型。将结果再 call 方法中返回可。同时使用线程池有返回值的 Submit 方法。</p>
<pre><code class="highlight-chroma">public static void main(String[] args) throws ExecutionException, InterruptedException {
    //创建条三个线程的线程池
    ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);
    //提交3个任务
    for (int i = 0; i < 3; i++) {
        //无返回值
        fixedThreadPool.execute(new Thread() -&gt; {
```

```

//爬虫任务
})
);
//有返回值
Future<String> future = fixedThreadPool.submit(new Crawler());
System.out.println(future.get());
}
}

static class Crawler implements Callable<String> {
    public String call() {
        //执行爬虫任务
        return "成功";
    }
}

```

</code></pre>

输出

上面的线程池创建方式是有很多弊端的，一般使用都会根据自己的场景配置线程池。

这里就不细写了，网上很多文章讲都非常不错。下面贴两篇

[Java 并发编程：线程的使用](https://link.hacpai.com/forward?goto=https%3A%2F%2Fwww.cnblogs.com%2Folphi
n0520%2Fp%2F3932921.html)

[Java 线程池](https://link.hacpai.com/forward?goto=https%3A%2F%2Fsegmentfault.com%2F%2F1190000015140610)

Guava线程池实战

[google Guava 包的 ListenableF
uture 解析](https://link.hacpai.com/forward?goto=http%3A%2F%2Fifeve.com%2Fgoogle-g
ava-listenablefuture%2F)

接口

传统 JDK 中的 Future 通过异步的方式计算返回结果：在多线程运算中可能或者可能在没有结束回结果，Future 是运行中的多线程的一个引用句柄，确保在服务执行返回一个 Result。

ListenableFuture 可以允许你注册回调方法(callbacks)，在运算（多线程执行）完成的时候进行用，或者在运算（多线程执行）完成后立即执行。这样简单的改进，使得可以明显的支持更多的操作这样的功能在 JDK concurrent 中的 Future 是不支持的。

ListenableFuture 中的基础方法是 [addListener\(Runnable, Executor\)](https://link.hacpai.com/forward?goto=http%3
2F%2Fdocs.guava-libraries.googlecode.com%2Fgit-history%2Frelease%2Fjavadoc%2Fcom
2Fgoogle%2Fcommon%2Futil%2Fconcurrent%2FListenableFuture.html%23addListener%28ja
a.lang.Runnable%2C%2520java.util.concurrent.Executor%29)，该方法会在多线程运算完的时候，指定的 Runnable 参传入的对象会被指定的 Executor 执行。

添加回调-Callbacks-

toc_h2_10

多数用户喜欢使用 [execute\(Runnable\)](https://link.hacpai.com/forward?goto=http%3A%2F%2Fdocs.
com%2Fgoogle%2Fcommon%2Futil%2Fconcurrent%2FExecutor.html%23execute%28java
.lang.Runnable%29)，该方法会在多线程运算完的时候，指定的 Runnable 参传入的对象会被指定的 Executor 执行。

uava-libraries.googlecode.com%2Fgit-history%2Frelease%2Fjavadoc%2Fcom%2Fgoogle%2Fcommon%2Futil%2Fconcurrent%2FFutures.html%23addCallback%28com.google.common.util.concurrent.ListenableFuture%2C%2520com.google.common.util.concurrent.FutureCallback%2C%2520java.util.concurrent.Executor%29" target="_blank" rel="nofollow ugc">>Futures.addCallback(ListenableFuture, FutureCallback, Executor)的方式，或者另外一个版本 >version (译者注: addCallback(ListenableFuture future,FutureCallback callback))，默认是采用 MoreExecutors sameThreadExecutor()线程池，为了简化使用，Callback 采用轻量级的设计。FutureCallback 中实现了两个方法: </p>

```
<ul>
<li><a href="https://link.hacpai.com/forward?goto=http%3A%2F%2Fdocs.guava-libraries.googlecode.com%2Fgit-history%2Frelease%2Fjavadoc%2Fcom%2Fgoogle%2Fcommon%2Futil%2Fconcurrent%2FFutureCallback.html%23onSuccess%28V%29" target="_blank" rel="nofollow ugc">onSuccess(V)</a>, 在 Future 成功的时候执行, 根据 Future 结果来判断。</li>
<li><a href="https://link.hacpai.com/forward?goto=http%3A%2F%2Fdocs.guava-libraries.googlecode.com%2Fgit-history%2Frelease%2Fjavadoc%2Fcom%2Fgoogle%2Fcommon%2Futil%2Fconcurrent%2FFutureCallback.html%23 onFailure%28java.lang.Throwable%29" target="_blank" rel="nofollow ugc">onFailure(Throwable)</a>, 在 Future 失败的时候执行, 根据 Future 结果来判断。</li>
</ul>
<h3 id="爬虫妹子图全站图片">爬虫妹子图全站图片</h3>
<pre><code class="highlight-chroma">public class meizitu {

//保存路径
static String path = "E:\\\\meizitu\\\\";
//图片地址
static String page = "http://meizitu.com/a/more_%d.html";
//起始页
static int pageCount = 1;
//当前可用CPU数
static final int PROCESSORS = Runtime.getRuntime().availableProcessors();
//线程名
```

```
static ThreadFactory threadName = new ThreadFactoryBuilder().setNameFormat("当前线程-%").build();
/***
• 合理配置线程池核心数:
• 如果是CPU密集型任务，就需要尽量压榨CPU，参考值可以设为 NCPU+1
• 如果是IO密集型任务，参考值可以设置为2~NCPU || 3NCPU
•
• 最大线程池为CPU*5
•
• 最大线程池-核心线程的线程，空闲200毫秒，没有任务则收回。
•
• 工作队列最大1000任务
•
• 超过则采用丢弃策略：任务并抛出RejectedExecutionException异常。
•
• 线程池基本知识：https://www.cnblogs.com/dolphin0520/p/3932921.html
*/
static ThreadPoolExecutor executor = new ThreadPoolExecutor(PROCESSORS * 3, PROCESSORS * 5, 200L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());
//guava并发
static ListeningExecutorService guavaExecutor = MoreExecutors.listeningDecorator(executor);

public static void main(String[] args) throws IOException {
    while (true) {
        //初始化httpClient
        CloseableHttpClient httpClient = HttpClients.createDefault();
        //格式化请求的图片页面地址
        String url = String.format(page, pageCount);
        //请求地址
        CloseableHttpResponse httpResponse = httpClient.execute(new HttpGet(url));
        //如果该页不是404则表示页面有数据
        if (404 == httpResponse.getStatusLine().getStatusCode()) {
            break;
        }
        String html = EntityUtils.toString(httpResponse.getEntity());
        //解析每个图片的URL
    }
}
```

```
Elements item = Jsoup.parse(html).getElementsByClass("wp-item");
for (Element element : item) {
String href = element.getElementsByTag("a").attr("href");
//任务结果回调 把每个图片的URL提交到线程池
Futures.addCallback(guavaExecutor.submit(new Crawler(href, pageCouont)), new FutureCallback<String>() {
    @Override
    public void onSuccess(@Nullable String result) {
        //任务执行成功
        System.out.println(result);
    }
    @Override
    public void onFailure(Throwable t) {
        //执行失败
        System.out.println(t.getMessage());
    }
}, executor);
}
pageCouont++;
}
System.out.println("任务页数: " + pageCouont);
}

static class Crawler implements Callable<String> {
private String url;
private int pageCount;

public Crawler(String url, int pageCount) {
this.url = url;
this.pageCount = pageCount;
}

@Override
public String call() {
FileOutputStream output = null;
```

```
ArrayList<String> list = Lists.newArrayList();
try {
    /**
     * 以下是分析图片地址并下载代码
     */
    String body = HttpRequest.get(this.url).body();
    Document parse = Jsoup.parse(body);
    String title = parse.title();
    Element picture = parse.getElementById("picture");
    Elements imgs = picture.getElementsByTag("img");
    for (Element e : imgs) {
        String src = e.attr("src");
        String alt = e.attr("alt");
        byte[] bytes = HttpRequest.get(src).bytes();
        //本来是打算获取到文章标题和图片的介绍保存为图片名字的，获取到的页面是乱码，暂时未解决。
        //String imgPath = path.concat("第" + this.pageCount + "页").concat("\\").concat(title).concat("\\").concat(alt);
        String fileDirs = path.concat("第" + this.pageCount + "页");
        String imgPath = fileDirs.concat("\\").concat(System.currentTimeMillis() + ".jpg");
        File file = new File(imgPath);
        if (!new File(fileDirs).exists()) {
            file.mkdirs();
        }
        File storeFile = new File(imgPath);
        output = new FileOutputStream(storeFile);
        //得到网络资源的字节数组，并写入文件
        output.write(bytes);
        list.add(imgPath);
    }
    String s = "CPU数：" + PROCESSORS + "，当前线程：" + Thread.currentThread().getName() + "\n" +
               "线程池中线程数目：" + executor.getPoolSize() + "，队列中等待执行的任务数目：" + "\n" +
               "executor.getQueue().size()" + "，已执行任务数目：" + executor.getCompletedTaskCount() + "\n" +
               "请求页面：" + url + "，地址：" + imgPath;
    System.out.println(s);
}
return "任务执行成功：" + url + ":" + JSON.toJSONString(list);
```

```
        } catch (IOException e) {
            throw new RuntimeException("任务异常: " + this.url);
        } finally {
            if (Objects.equals(output, null)) {
                try {
                    output.close();
                } catch (IOException e) {
                    throw new RuntimeException("任务异常: " + this.url);
                }
            }
        }
    }
}

</code></pre>
```

<h3 id="结果">结果</h3>
<p>这里设置的工作队列有点小了，差点满了。最大 36 线程在执行，满了会抛异常，一定要合理设该大小！！！不然任务会丢弃</p>
<p></p>
<p></p>
<p></p>
<p></p>