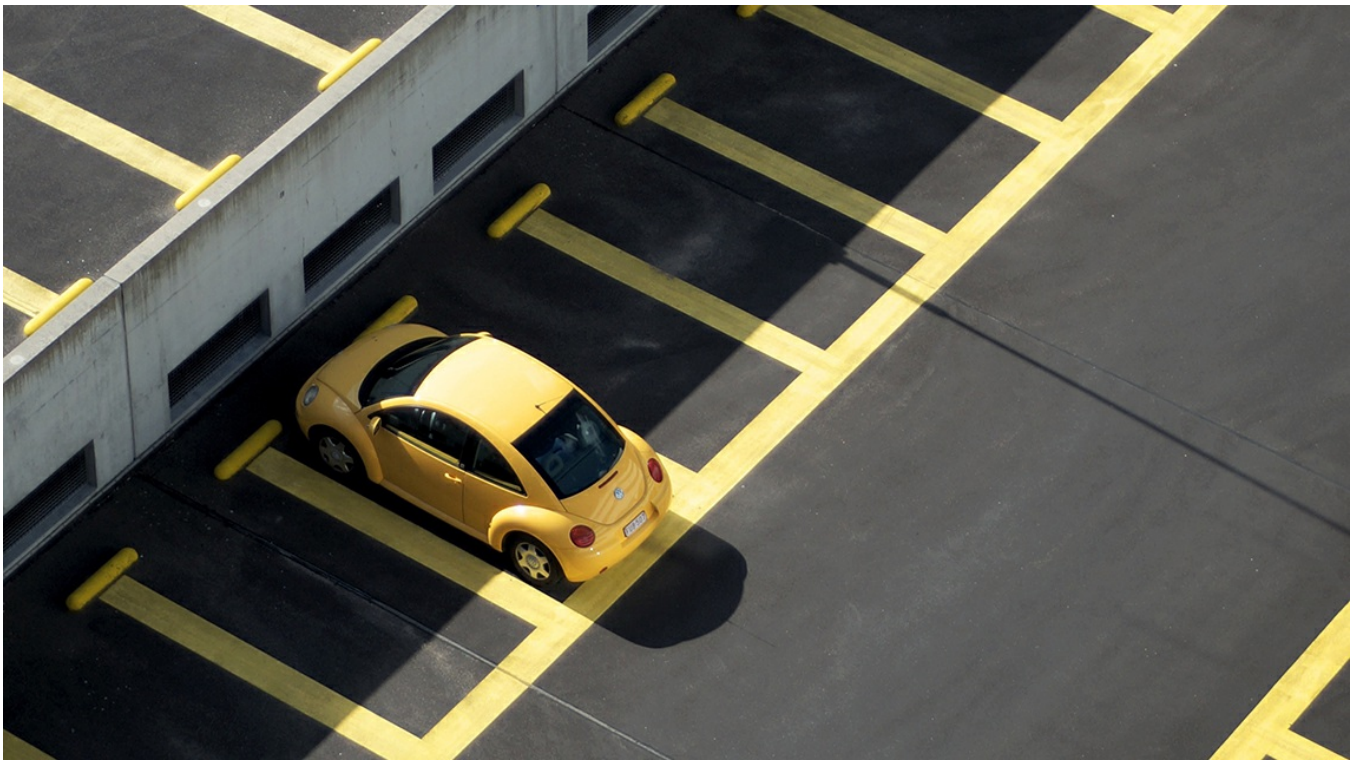
25 | 使用有序的代码，调动异步的事件

2019-03-01 范学雷



同步和异步，是两个差距很大的编程模型。同步，就是很多事情一步一步地做，做完上一件，才能做下一件。异步，就是做事情不需要一步一步的，多件事情，可以独立地做。

比如一个有小鸟的笼子，如果打开笼门，一个一个地放飞小鸟，就是同步。如果拆了整个鸟笼，让小鸟随便飞，爱怎么飞就怎么飞，这就是异步。

为什么需要异步编程？

如果我们观察身边的事物，现实中有很多事情是以异步的方式运营的。我们挤地铁的时候，从来都是好几个人一起挤进去的。当我们正在挤地铁时，外面的风照旧吹，雨照旧下，天坛的大爷大妈们正在秀着各种绝活。没有任何事情会因为我们正在挤地铁就停止活动，等我们挤完地铁再恢复运转。

可是，要是说到其中的任何一个人，就不能同时做两件事情了。在挤地铁的时候，就不能在天坛秀绝活。我们写的程序也是这样，先执行第一行，再执行第二行。哪怕第二行再怎么费周折，第三行代码也要等着。

第二行代码可能需要执行大量的计算，需要很多的CPU；也可能需要大量的传输，占用I/O通道。可是，它不一定会把所有的计算机资源都占用了。

如果第二行代码占用了I/O，我们能不能把多余的CPU用起来？如果第二行代码占用了CPU，我们能不能把空闲的I/O用起来？也就是说，能不能把计算机整体资源更有效地使用起来？

该怎么办呢？想想家里的一把手做事的风格吧。

“你去小区菜店买瓶酱油，买回来我们就做饭。”第一道指令发布完毕。

“你把垃圾扔出去吧，都有馊味了。”第二道指令发布完毕。

“我赶快收拾下屋子，有两天没打扫了。”第三道指令发布完毕。

尽管每一道指令都很简短，但是每件事情都交代得很清楚。然后，每个人都忙碌了起来，各忙各的事情。效率也就比一件事情做完再做下一件高出很多。

如果我们把三行代码换成三道指令。第三行代码虽然依然要等待，但只需等待第二道指令发布完成，而不是第二道指令背后的事情完成。等待的时间变短，效率也就提升了。

我想，这就是异步编程的背后的驱动力量，以及基本的处理逻辑。为了更有效地利用计算资源，我们使用有序的代码，调动起独立的事件。

从过程到事件

异步编程和我们熟悉的同步编程最大的区别，就是它要我们从事件的角度来编写和理解代码。就像我举的生活中的一些例子，说的做的多是“事情”。由于我们一般先学习的是对象、方法和过程这些模型，已经建立了一定的思考模式，对于事件驱动的编程模型可能会有点不习惯。事实上，熟悉了异步编程的思路，你会发现异步编程很贴近我们的生活模式。

在下面的例子，我使用了JDK 11新添加的HttpClient接口。最后一个语句，就是一个异步模式。这个语句的意思，就是交代一件事情：“访问www.example.com，并且把响应数据打印到标准输出上。”需要注意的是，这个语句就是发布了这条指令。指令发布完，这个语句的任务就完成了，就可以执行下一个语句了，不需要等待指令交代的任务完成。

```
// Create an HTTP client that prefers HTTP/2.
HttpClient httpClient = HttpClient.newBuilder()
            .version(Version.HTTP_2)
            .build();

// Create a HTTP request.
HttpRequest httpRequest = HttpRequest.newBuilder()
            .uri(URI.create("https://www.example.com/"))
            .build();

// Send the request and set the HTTP response handler
httpClient.sendAsync(httpRequest, BodyHandlers.ofString())
            .thenApply(HttpResponse::body)
            .thenAccept(System.out::println);

// next action
```

我们可以对比一下传统的代码。下面的代码使用了JDK 10以前的HttpURLConnection接口。完成的是同样的任务。不同的是，下一件事情的代码需要等待上一件事情的完成，才能执行。也就是说，建立网络连接之后，才能执行读取响应数据的代码。

```
// Open the connection
URL url = new URL("https://www.example.com/");
HttpsURLConnection urlc = (HttpsURLConnection)url.openConnection();

// Read the response
try (InputStream is = urlc.getInputStream()) {
    while (is.read() != -1) {    // read to EOF
        // dump the response
        // snipped
    }
}

// next action
```

使用HttpURLConnection接口的代码，无论是连接过程，还是响应数据的读取过程，都依赖于网

络环境，而不仅仅是计算机的环境。如果网络环境的响应时间是三秒，那么上面的代码就要阻塞三秒，无法执行下一步操作。

而HttpClient接口的代码，指令发布完，就可以执行下一步操作了。这个指令的执行时间，一般是毫秒以下的数量级别。

如果我们不考虑其他因素的影响，那么上面的两个例子中，异步模式在网络阻塞期间，能够更好地利用其他的计算资源，从而提高整体的效率。

异步是怎么实现的？

你是不是有个疑问，指令交代的任务是怎么完成的？异步的实现，依赖于底层的硬件和操作系统；如果操作系统不支持，异步也可以通过线程来模拟。

即便是只能通过线程来模拟，异步编程也简化了线程管理的难度。甚至能够把线程管理变透明，隐藏起来。比如我们上面使用的HttpClient接口的代码，就没有线程的影子，看起来像一个单线程程序。

异步编程对性能的爆炸性的提升来自于硬件和操作系统对异步的支持。

比如说，早期传统的套接字编程，应用程序需要等待下一个连接的到来，然后等待连接数据的传输....这些等待，都需要耗费很多资源。这些被占用的资源，在连接和数据到来之前，都是没有被充分利用的资源。

如果操作系统能够主动告诉应用程序，什么时候有一个连接请求，这个连接里什么时候有数据。应用程序就可以在连接和数据到来之后，再分配资源进行处理。操作系统在合适的时间，遇到触发事件，主动调用设置的应用程序，执行相关的操作。这就是操作系统对异步I/O的支持。

比如说，如果一个简单的服务就返回一个"Hello, World!"，它能够同时接受多少用户访问呢？

如果使用传统的一个线程一个用户的模式，这个用户数量完全取决于线程的效率和容量。随着用户数的增加，线程数量也线性增加，线程管理也越来越复杂，线程的效率也加速下降，线程处理能力决定了系统最大可承载的用户数。

如果使用异步I/O，每一个CPU分派一个线程就足以应付所有的连接。这时候，连接的效率就主要取决于硬件和操作系统的能力了。

根据常见的数据，这种效率的提升通常可以达到几百倍。

下面的例子，就是一个简单异步服务的框架。你可以比较一下，它和传统服务器代码的差异。

```
final AsynchronousServerSocketChannel listener =
        AsynchronousServerSocketChannel
                .open()
                .bind(new InetSocketAddress("localhost", 6789));

listener.accept(null, new CompletionHandler<AsynchronousSocketChannel,Void>() {
    @Override
    public void completed(AsynchronousSocketChannel ch, Void att) {
        // accept the next connection, non-blocking
        listener.accept(null, this);

        // handle this connection
        handle(ch);
    }

    @Override
    public void failed(Throwable exc, Void att) {
        // snipped
    }
});
```

零拷贝，进一步的性能提升

异步编程的性能并没有止步于异步I/O，它还有提升的空间。

前面，我们讨论了减少内存使用的两个大方向，减少实例数量和减少实例的尺寸。使用共享内存，减少内存拷贝，甚至是零拷贝，可以减少CPU消耗，也是减少实例数量和减少实例尺寸的一个办法。

下面的例子中，我们使用了ByteBuffer.allocateDirect()方法分配了一块内存空间。这个方法的实现，会尽最大的努力，减少中间环节的内存拷贝，把套接字的缓存数据，直接拷贝到应用程序操作的内存空间里。这样，就减少了内存的占用、分配、拷贝和废弃，提高了内存使用的效率。

```
listener.accept(null, new CompletionHandler<AsynchronousSocketChannel,Void>() {
    @Override
    public void completed(AsynchronousSocketChannel ch, Void att) {
        // accept the next connection, non-blocking
        listener.accept(null, this);

        // handle this connection
        ByteBuffer bbr = ByteBuffer.allocateDirect(1024);
        ch.read(bbr, null, new CompletionHandler<Integer, Object>() {
    @Override
    public void completed(Integer result, Object attachment) {
                    // snipped
                }

    @Override
    public void failed(Throwable exc, Object attachment) {
                    // snipped
    }
   });
    }

    @Override
    public void failed(Throwable exc, Void att) {
        // snipped
    }
});
```

需要注意的是，这种方式分配的内存，分配和废弃的效率一般比常规的Java内存分配差一些。所以，只建议用在数据量比较大，存活时间比较长的情况下，比如网络连接的I/O。而且，一个连接最多只用一个读、一个写两块空间。这样，才能把它的效率充分发挥出来。

小结

今天，我们主要讨论了异步的一些基本概念，以及异步对于效率提升的作用。异步编程，常见的模型是事件驱动的。我们通过使用有序的代码，调动独立的事件，来更有效地利用计算资源。

一起来动手

这一次的几个例子，大致提供了异步连接编程的一个基本框架。你可以试着把这些代码丰富起

来，组成一个可以运行的客户端和服务端。客户端使用HttpClient接口发起HTTP连接；服务端使用异步的模式，把客户端的HTTP请求数据原封不动发回去。

下一篇文章，我会介绍一个简单的测试代码性能的工具。如果有兴趣，你可以继续测试下你编写的代码的性能，是不是比同步的编程模式有所提高。

欢迎你把你的代码公布在讨论区，我们一起来学习，一起来进步。如果你想和朋友或者同事比试一下，不妨把这篇文章分享给他们，互相切磋。

精选留言

黄智勇 1

这就体现了我用nodejs来做开发的优势了，async/await 的方式，还可以开发异步程序像开发同步程序，开发效率一样高

2019-03-02

正在减肥的胖籽。 1

范老师您好，在Java项目中，tomcat的线程一般开多少个线程比较好？这块有好的心德吗？在项目开发过程中一直对线程池的个数拿捏不准。

2019-03-01

作者回复

一般的，线程个数和很多因素相关，比如软件架构、用户数等。线程数和CPU数匹配是一个常

见的设置，但是也仅适用于少数场景，比如每个计算任务都很快。性能参数选择，一般可以做成可配置的，然后反复测试，找到合适的组合。

很抱歉，我不了解tomcat的细节，没有办法给你建议。留言区的小伙伴们帮帮忙！

2019-03-01

轻歌赋

0

其实我更想知道异步编程的一些编码规范，来帮助我少些一些多线程异常代码

2019-03-06

作者回复

异步编程的编码模式吗？ 编码规范，如果你指的是代码规范，和普通的代码规范区别不大。

2019-03-07