04 | 流与异步：为什么说掌握流计算先要理解异步编程?

在前面的课时中，我们详细分析了“异步”的工作原理，并且在解决异步系统的 OOM 问题时，使用了“反向压力”的方法。在讨论过程中，我们已经明确地使用到，诸如上游、下游、streams 这样的概念都暗示着我们，“流”和“异步”之间有着某种关联。

所以今天，我们就借助于目前四种主流的异步编程方案，来详细分析下“流”和“异步”之间这种紧密关系。

异步编程框架

说到“异步编程”或者“高并发编程”，你首先想到的是什么呢?

根据我以往当面试官的经验：

- **青铜级**的求职者，一般会说多线程、synchronized、锁等知识，更有甚者还会扯到 Redis 神马的。很显然，这类求职者对异步和高并发编程，其实是没有什么概念的；
- **白银级**的求职者，则会说线程池、executor、ConcurrentHashMap 等，这类同学对异步和高并发编程，已经有了初步认识，但却还不够深入；
- **王者级别**的求职者，则会对 NIO、Netty、CompletableFuture 等技术如数家珍，甚至还会谈到 Fiber。

其实很多时候，我问求职者的问题，都是在实际开发过程中，需要使用或注意的知识点，要求并不苛刻。毕竟面试的目的，是尽快招到合适的开发人员一起做事，而不是为了刁难人家。但可惜的是，我遇到最多的是青铜，少有白银，王者则更是稀有了。我自己也曾面试过某 BAT 大厂之一，记得当时最后一轮技术面，是三个不同部门的老大同时面我。他们问了我很多问题，其中印象最深的一个，就是关于异步和高并发编程的问题。当时我从“流”的角度，结合 NIO 和 CompletableFuture 等工具，跟他们详细讲解了我在平时开发过程中，总结出的最佳实践方案。最后，我顺利拿到了 offer。

所以，回到问题本身，当我们谈论“异步”和“高并发”编程时，到底是在说什么呢？通过 第 02 课时 的学习，我们已经知道，**“高并发”其实是我们要解决的问题，而“异步”则是为了更有效地利用 CPU 和 IO 资源，来解决“高并发”问题时的编程方式**。在“高并发”场景下，我们通常会使用“异步”的编程方式，来提升 CPU 和 IO 的使用效率，从而提高程序性能。

所以进一步地，我们的问题落在了选择“异步”编程方案上。那究竟怎样实现异步编程呢？其实，异步编程的框架非常多，目前主流的异步编程可以分为四类模式：Promise、Actor、ReactiveX 和纤程（或者说协程）。下面我们逐一讨论。

Promise 模式

Promise 模式是非常基本的异步编程模式，在 JavaScript、Java、Python、C++、C# 等语言中，都有 Promise 模式的实现。**Promise 正如其名，代表了一个异步操作在其完成时会返回并继续执行的承诺**。

Promise 模式在前端 JavaScript 开发中，是非常常见的。这是因为 JavaScript 本身是单线程的，为了解决诸如并发网络请求的问题，JavaScript 使用了大量异步编程的技巧。早期的 JavaScript 还不支持 Promise 模式，为了实现异步编程，采用的都是回调的方式。但是回调会有一个问题，就是所谓的“回调陷阱”。

举个例子，当你需要依次调用 A、B、C、D 四次网络请求时，如果采用回调的编程方式，那么四次网络请求的回调函数，会依次嵌套起来。这样，整个回调函数的实现会非常长，逻辑会异常复杂，不容易理解和维护。

为了解决“回调陷阱”的问题，JavaScript 引入了 Promise 模式。类似于下面这样：

```
let myPromise = new Promise(function(myResolve, myReject) {
  setTimeout(function() { myResolve("Hello World!"); }, 5000);
});
myPromise.then(function(value) {
  document.getElementById("test").innerHTML = value;
})
```

在上面的这段 JavaScript 代码中，实现了一个异步的定时器。定时器定时 5 秒后返回，然后将 id 为"test"的元素设置为“Hello World!”。

可以看出，Promise 模式将嵌套的回调过程，变成了平铺直叙的 Promise 链，这极大地简化了异步编程的复杂程度。

那在 Java 中的 Promise 模式呢？在 Java 8 之前，JDK 是不支持 Promise 模式的。好在 Java 8 为我们带来了 CompletableFuture 类，这就是 Promise 模式的实现。比如在 03 课时 的异步执行代码，正是一个 Promise 链。

```
CompletableFuture
            .supplyAsync(() -> this.decode(ctx, req), this.decoderExecutor)
            .thenApplyAsync(e -> this.doExtractCleanTransform(ctx, req, e), this.ectExecutor)
            .thenApplyAsync(e -> this.send(ctx, req, e), this.senderExecutor);
```

在上面的代码中，我们使用的 executor 都是带队列的线程池，也就是类似于下面这样。

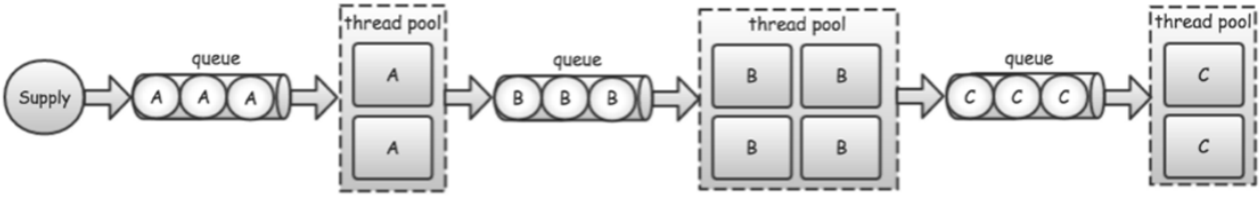

图1 带队列的线程池

从上面的图 1 可以看出，这个过程有生产者，有队列，有消费者，是不是已经非常像“流”?

当然，CompletableFuture 类也可以使用其他类型的 executor，比如，使用栈管理线程的 executor。在这种 executor 的实现中，每次调用 execute() 方法时，都是从栈中取出一个线程来执行任务，像这种不带任务队列的执行器，就和“流”相差甚远了。

Actor 模式

Actor 模式是另外一种非常著名的异步编程模式。在这种模式中，用 Actor 来表示一个个的活动实体，这些活动实体之间以消息的方式，进行通信和交互。

Actor 模式非常适用的一种场景是游戏开发。比如 DotA 游戏里的小兵，就可以用一个个 Actor表示。如果要小兵去攻击防御塔，就给这个小兵 Actor 发一条消息，让它移动到塔下，再发一条消息，让它攻击塔。

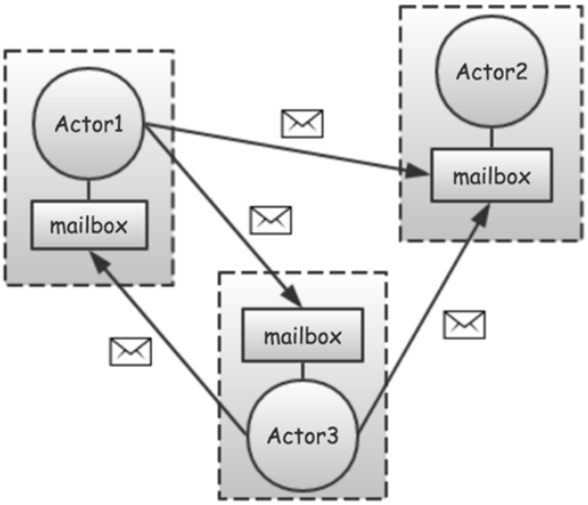

图2 Actor系统

必须强调的是，Actor 模式最好是构建在纤程上，这样 Actor 才能随心所欲地想干吗就干吗，你写代码时就不会有过多的约束。

如果 Actor 是基于线程构建，那么当存在较多 Actor 时，Actor 的代码就不宜做过多 IO 或 sleep 操作。但大多数情况下，IO 操作都是难以避免的，所以为了减少 IO 和 sleep 操作对其他 Actor 的影响，应将涉及 IO 操作的 Actor 与其他非 IO 操作的 Actor 隔离开。给涉及 IO 操作的 Actor 分配专门的线程，不让这些 Actor 和其他非 IO 操作的 Actor 分配到相同的线程。这样可以保证 CPU 和 IO 资源，都能充分利用，提高了程序的性能。

在 JVM 平台上，比较有名的 Actor 模式是 Akka。但是 Akka 是构建在线程而非纤程上，所以使用起来就存在上面说的这些问题。

如果你要用 Akka 的话，需要注意给以 IO 操作为主的 Actor ，分配专门的线程池。另外，Akka 自身不具备反向压力功能，所以使用起来时，还需要自己进行流量控制才行。

我自己曾经实现过，感觉还是有点小麻烦的。主要的问题在于 Actor 系统对邮箱的定位，已经要求邮箱，也就是 Actor 用于接收消息的队列，最好不要阻塞。所以如果是做流量控制的话，就不能直接将邮箱，设置为容量有限的阻塞队列，这样在 Actor 系统中，非常容易造成死锁。

ReactiveX 模式

ReactiveX 模式又称之为响应式扩展，它是一种观察者模式。在Java 中，ReactiveX 模式的实现是 RxJava。ReactiveX 模式的核心是观察者（Observer）和被观察者（Observable）。被观察者（Observable）产生一系列的事件，比如网络请求、数据库操作、文件读取等，然后观察者会观察到这些事件，之后就触发一系列后续动作。

下面就是使用 RxJava 编写的一段异步执行代码。

```
//被观察者
Observable observable = Observable.create(new Observable.OnSubscribe<String>() {
    @Override
    public void call(Subscriber<? super String> subscriber) {
        subscriber.onNext("Hello");
        subscriber.onNext("World");
        subscriber.onNext("!!!");
        subscriber.onCompleted();
    }
});

//观察者
Observer<String> observer = new Observer<String>() {
    @Override
    public void onNext(String s) {
        Log.d(tag, "onNext: " + s);
    }

    @Override
    public void onCompleted() {
        Log.d(tag, "onCompleted called");
    }

    @Override
    public void onError(Throwable e) {
        Log.d(tag, "onError called");
    }
};

// 订阅
observable.subscribe(observer);
```

在上面的代码中，被观察者依次发出“Hello”“World”“!!!”三个事件，然后观察者观察到这三个事件后，就将每个事件打印出来。

非常有趣的是，ReactiveX 将其自身定义为一个异步编程库，却明确地将被观察者的事件序列，按照“无限流”（infinite streams）的方式来进行处理，还实现了 Reactive-Streams 标准，支持反向压力功能。

你说，是不是他们也发现了，流和异步之间有着相通之处呢?

不过，相比 Java 8 的 CompletableFuture，我觉得这个 RxJava 还是显得有些复杂，理解和使用起来都更加麻烦，但明显的优势又没有，所以我不太推荐使用这种异步编程模式。

我在之前的工作中，也有见过其他同事在 Android 开发时使用这种模式。所以，如果你感兴趣的话，也可以了解一下。如果是我，我就直接使用 CompletableFuture 了。

纤程/协程模式

最后是纤程（fiber）模式，也称之为协程（coroutine）模式。应该说纤程是最理想的异步编程方案了，没有之一！它是用“同步方式写异步代码”的最高级别形态。

下面的图 3 是纤程的工作原理，纤程是一种用户态的线程，其调度逻辑在用户态实现，从而避免过多地进出内核态，进行调度和上下文切换。

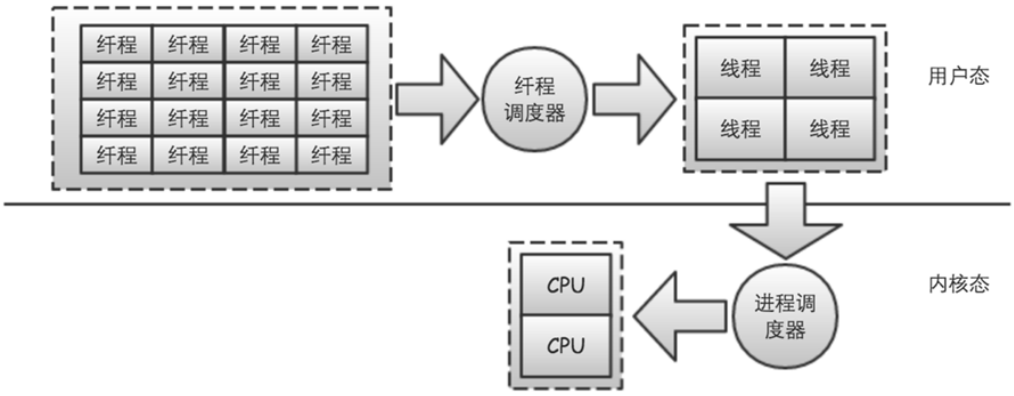

图3 纤程的工作原理

实现纤程的关键，是要在执行过程中，能够在恰当的时刻和地方中断，并将 CPU 让给其他纤程使用。具体实现起来就是，将 IO 操作委托给少量固定线程，再使用另外少量线程负责 IO 状态检查和纤程调度，再使用另外一批线程执行纤程。

这样，少量线程就可以支撑大量纤程的执行，从而保证了 CPU 和 IO 资源的使用效率，提升了程序的性能。

使用纤程还可以极大地降低异步和并发编程的难度。但可惜的是，当前 Java 还不支持纤程。Java 对纤程的支持还在路上，你可以查阅一个被称之为 Loom 的项目来跟踪进度。所以这里，我就先借助当前另外一款火爆的编程语言 Golang，来对纤程做一番演示。

下面就是一个 Golang 协程的示例代码。

```
package main
import (
   "fmt"
   "time"
)
func produce(queue chan int) {
   for i := 0; true; i++ {
      time.Sleep(1 * time.Second)
      queue <- i
      fmt.Printf("produce item[%d]\n", i)
   }
}
func consume(queue chan int) {
   for {
      e := <-queue
      fmt.Printf("consume item[%d]\n", e)
   }
}
func waitForever() {
   for {
      time.Sleep(1 * time.Second)
   }
}
func main() {
   // 创建传递消息的队列，这是一个容量为10的阻塞队列
   queue := make(chan int, 10)
   // 启动生产者协程
   go produce(queue)
   // 启动消费者协程
   go consume(queue)
   // 防止程序退出
   waitForever()
}
```

在上面的代码中，我们使用了 Golang 最核心的两个概念，即 goroutine 和 channel。Golang 最推崇的并发编程思路就是，通过通信来共享内存，而不是通过共享内存来进行通信。所以，我们可以非常直观地看到，这里的 channel 就是一个容量有限的阻塞队列，天然就具备了反向压力的能力。

所以，Golang 这种生产者、队列、消费者的模式，不就是“流”的一种雏形吗?

异步和流之间的关系

至此，我们已经讨论了四种不同的异步编程模式。除了像 async/await 这样的异步编程语法糖外，上面讨论的四种模式，基本覆盖了当前所有主流的异步编程模式。这里稍微提一下，async/await 这个异步编程语法糖，还是非常有趣的，Python 和 JavaScript 都支持，建议你了解一下。

我们再回过头来看下，这四种异步编程模式，它们都已经暗含了“流”的影子。

首先是 Promise 模式，当 CompletableFuture 使用的执行器，是带队列的线程池时，Promise 异步调用链的过程，在底层就是事件在队列中“流”转的过程。

然后是 Actor 模式，每个 Actor 的邮箱就是一个非阻塞的队列，Actor 之间的通信过程，就是消息在这些非阻塞队列之间“流”转的过程。

接下来就是 ReactiveX 模式，将自己定义为异步编程库的 ReactiveX，明确地将事件按照“无限流”的方式来处理，还实现了 Reactive-Streams 标准，支持反向压力功能。

最后是纤程和协程，Golang 语言明确将“队列”作为了异步和并发编程时最主要的通信模式，甚至将“通过通信来共享内存，而不是通过共享内存来进行通信”，作为一种编程哲学思想来进行推崇。

所以，在**四种异步编程模式中，我们都能够发现“队列”的身影**，而“队列”正是“流”计算系统最重要的组成结构。我们可以利用这种结构，来实现“流”计算的过程。

有“队列”的系统，注定了它会是一个异步的执行过程，这也意味着“流”这种计算模式注定了是“异步” 的。异步系统中存在的 OOM 问题，在“流”计算系统中也存在，而且同样是使用“反向压力”的方式来解决。

小结

今天，我们分析了四种主流的异步编程模式，并讨论了“流”和“异步”之间的关系。总的来说，**“流”和“异步”是相通的。其中，“异步”是“流”的本质，而“流”是“异步”的一种表现形式**！

至此，我们已经讨论完了所有有关异步编程的内容。可以看到，我在模块一中，花了整整三个课时，来讲解有关 NIO、异步编程和流计算的工作原理，以及它们之间的关联关系。这是因为这些知识，不仅仅是后面流计算内容的基础，它们也可以被用于其他任何高并发编程场景。而且根据我多年的工作经验，我坚定地认为，理解 NIO 和异步编程，是程序员成为大神的必要条件。所以，我希望你能够彻底地掌握和理解模块一的内容，它们一定会对于你以后的工作，有所帮助！

下一讲，我们将正式进入流计算系统的模块。那对于“流”和“异步”之间的关系，你还有什么问题或想法呢？有的话可以在留言区写下来，我看到后会进行分析和解答！

最后，用一个脑图来概括下本课时讲解的内容，以便于你理解。

点击此链接查看本课程所有课时的源码

PB 级企业大数据项目实战 + 拉勾硬核内推，5 个月全面掌握大数据核心技能。点击链接，全面赋能！