# 02 | 异步和高并发：为什么 NIO 是异步和高并发编程的基础?

为什么在讲流计算之前，要先讲异步和高并发的问题呢?

- 其一，是因为“流”本质是异步的，可以说“流计算”也是一种形式的异步编程。
- 其二，是因为对于一个流计算系统而言，其起点一定是数据采集，没数据就什么事情都做不了，而数据采集通常就会涉及 IO 问题，如何设计一个高性能的 IO 密集型应用，异步和并发编程既是过不去的坎，也是我们掌握高性能 Java 编程的基础。

所以，在这个课时中，我们就从数据采集模块切入，通过开发一个高性能的数据采集模块，从实战中理解 NIO、异步和高并发的原理。这样，当你以后开发高性能服务时，比如需要支持数万甚至数百万并发连接的 Web 服务时，就知道如何充分发挥出硬件资源的能力，就可以用最低的硬件成本，来达到业务的性能要求。

为了更方便地说明问题，我们今天的讨论，以从互联网上采集数据为例。具体来说，如下图 1 所示，数据通过 REST 接口，从手机或网页端，发送到数据采集服务器。

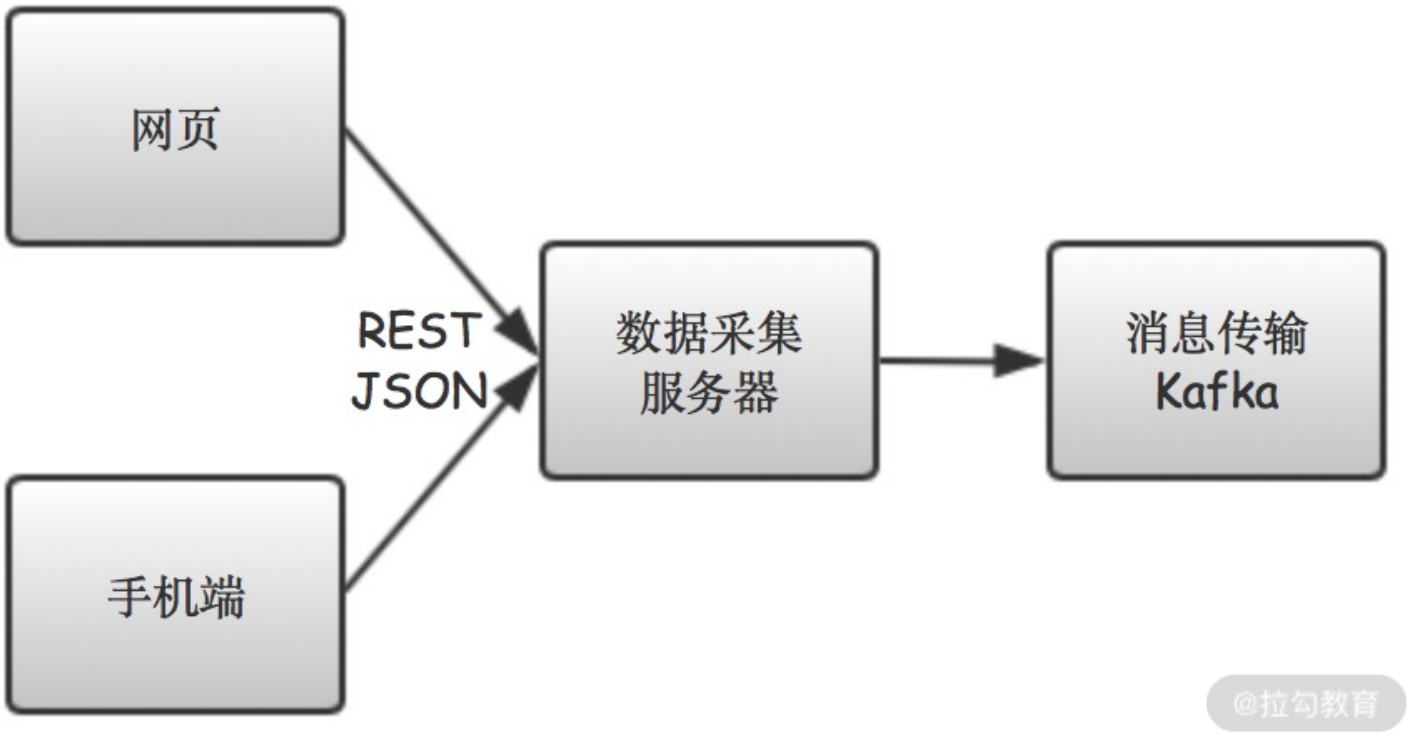



图 1 基于 REST 协议的数据采集服务器

## BIO 连接器的问题

由于是面向互联网采集数据，所以我们要实现的数据采集服务器，就是一个常见的 Web 服务。说到 Web 服务开发，作为 Java 开发人员，十有八九会用到 Tomcat。毕竟 Tomcat 一直是 Spring 生态的默认 Web 服务器，使用面是非常广的。

但使用 Tomcat 需要注意一个问题。在 Tomcat 7 及之前的版本中， Tomcat 默认使用的是 BIO 连接器， BIO 连接器的工作原理如下图 2 所示。

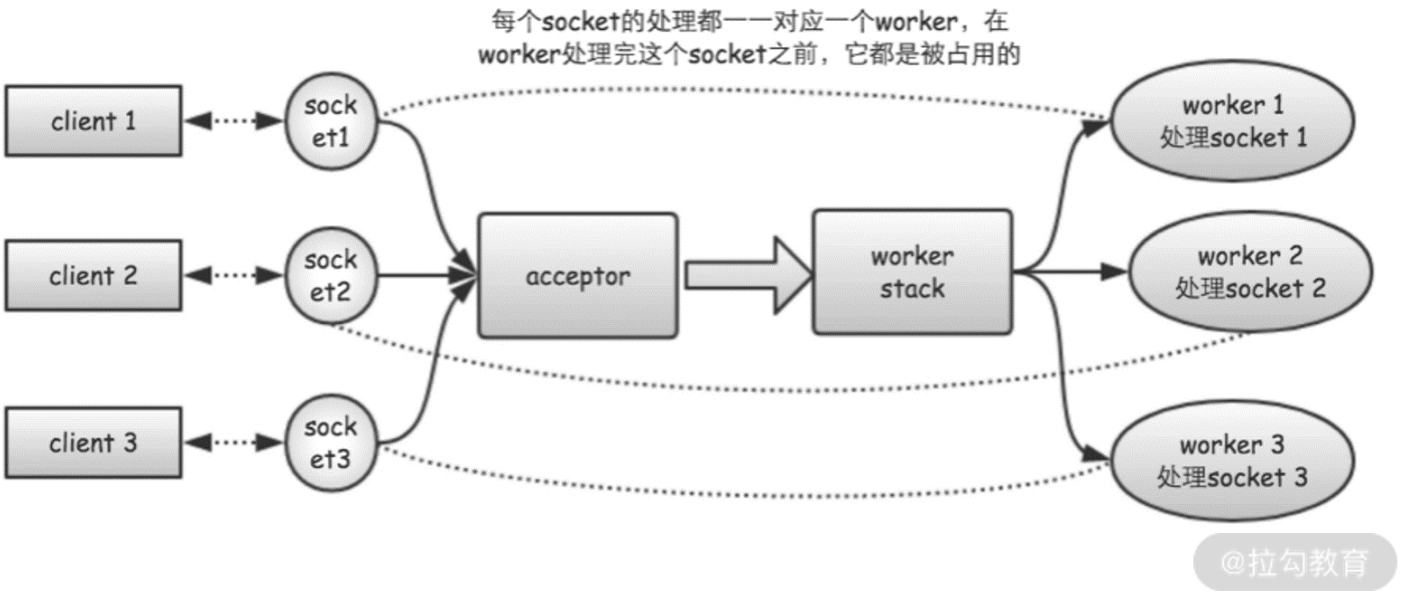



图 2 BIO连接器工作原理

当使用 BIO 连接器时，Tomcat 会为每个客户端请求，分配一个独立的工作线程进行处理。这样，如果有 100 个客户端同时发送请求，就需要同时创建 100 个工作线程。如果有 1 万个客户端同时请求，就需要创建 1 万个工作线程。而如果是 100 万个客户端同时请求呢？是不是需要创建 100 万个工作线程？

所以，**BIO 连接器的最大问题是它的工作线程和请求连接是一一对应耦合起来的**。当同时建立的请求连接数比较少时，使用 BIO 连接器是合适的，因为这个时候线程数是够用的。但考虑下，像 BATJ 等大厂的使用场景，哪家不是成万上亿的用户，哪家不是数十万、数百万的并发连接。在这些场景下，使用 BIO 连接器就根本行不通了。

所以，我们需要采取新的方案，这就是 Tomcat NIO 连接器。

## 使用 NIO 支持百万连接

毫无意外的是，从 Tomcat 8 开始，Tomcat 已经将 NIO 设置成了它的默认连接器。所以，如果你此时还在使用 Tomcat 7 或之前的版本的话，需要检查下你的服务器，究竟使用的是哪种连接器。

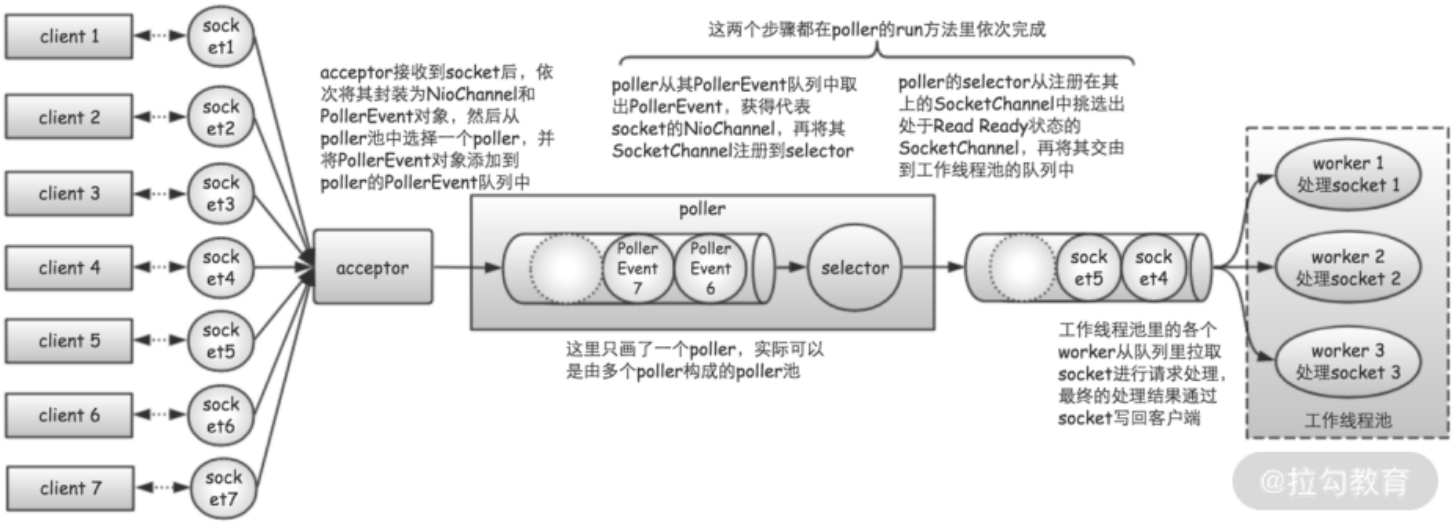



图 3 NIO连接器工作原理

图 3 是 NIO 连接器的工作原理。可以看出，**NIO 连接器相比 BIO 连接器，主要做出了两大改进**。

- **一是，使用“队列”将请求接收器和工作线程隔开；**
- **二是，引入选择器来更加精细地管理连接套接字。**

NIO 连接器的这两点改进，带来了两个非常大的好处。

- 一方面，将请求接收器和工作线程隔离开，可以让接收器和工作线程，各自尽其所能地工作，从而更加充分地使用 IO 和 CPU 资源。
- 另一方面，NIO 连接器能够保持的并发连接数，不再受限于工作线程数量，这样无须分配大量线程，数据接收服务器就能支持大量并发连接了。

所以，使用 NIO 连接器，我们解决了百万并发连接的问题。但想要实现一个高性能的数据采集服务器，光使用 NIO 连接器还不够。因为当系统支持百万并发连接时，也就意味着我们的系统是一个吞吐量非常高的系统。这就要求我们在实现业务逻辑时，需要更加精细地使用 CPU 和 IO 资源。否则，千辛万苦改成 NIO 的努力，就都白白浪费了。

## 如何优化 IO 和 CPU 都密集的任务

考虑实际的应用场景，当数据采集服务器在接收到数据后，往往还需要做三件事情：

- 一是，对数据进行解码；
- 二是，对数据进行规整化，包括字段提取、类型统一、过滤无效数据等；
- 三是，将规整化的数据发送到下游，比如消息中间件 Kafka。

在这三个步骤中，1 和 2 主要是纯粹的 CPU 计算，占用的是 CPU 资源，而 3 则是 IO 输出，占用的是 IO 资源。每接收到一条数据，我们都会执行以上三个步骤，所以也就构成了类似于图 4 所示的这种循环。

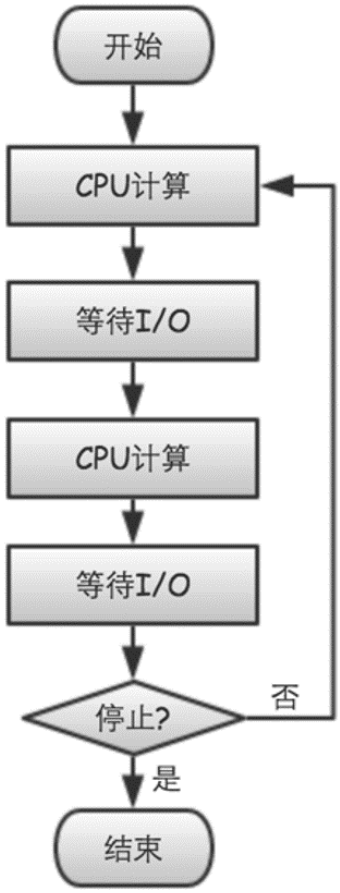




图 4 CPU和IO都密集型任务

从图 4 可以看出，数据采集服务器是一个对 CPU 和 IO 资源的使用都比较密集的场景。为什么我们会强调这种**CPU 和 IO 的使用都比较密集的情况**呢？因为这是**破解“NIO 和异步”为什么比“BIO 和同步”程序，性能更优的关键**所在！下面我们就来详细分析下。

如果想提高 IO 利用率，一种简单且行之有效的方式，是使用更多的线程。这是因为当线程执行到涉及 IO 操作或 sleep 之类的函数时，会触发系统调用。线程执行系统调用，会从用户态进入内核态，之后在其准备从内核态返回用户态时，操作系统将触发一次线程调度的机会。对于正在执行 IO 操作的线程，操作系统很有可能将其调度出去。这是因为触发 IO 请求的线程，通常需要等待 IO 操作

完成，操作系统就会暂时让其在一旁等着，先调度其他线程执行。当 IO 请求的数据准备好之后，线程才再次获得被调度的机会，然后继续之前的执行流程。

但是，是不是能够一直将线程的数量增加下去呢？不是的！如果线程过多，操作系统就会频繁地进行线程调度和上下文切换，这样 CPU 会浪费很多的时间在线程调度和上下文切换上，使得用于有效计算的时间变少，从而造成另一种形式的 CPU 资源浪费。

所以，**针对 IO 和 CPU 都密集的任务，其优化思路是，尽可能让 CPU 不把时间浪费在等待 IO 完成上，同时尽可能降低操作系统消耗在线程调度上的时间**。

那具体如何做到这两点呢？这就是接下来要讲的，“NIO”结合“异步”方法了。

## NIO 结合异步编程

既然要说异步，那什么是异步？举个生活中的例子。当我们做饭时，在把米和水放到电饭锅，并按下电源开关后，不会干巴巴站在一旁等米饭煮熟，而是会利用这段时间去炒菜。当电饭锅的米饭煮熟之后，它会发出嘟嘟的声音，通知我们米饭已经煮好。同时，这个时候我们的菜肴，也差不多做好了。

在这个例子中，我们没有等待电饭锅煮饭，而是让其在饭熟后提醒我们，这种做事方式就是“异步”的。反过来，如果我们一直等到米饭煮熟之后再做菜，这就是“同步”的做事方式。

对应到程序中，我们的角色就相当于 CPU ，电饭锅煮饭的过程，就相当于一次耗时的 IO 操作，而炒菜的过程，就相当于在执行一段算法。很显然，异步的方式能更加有效地使用 CPU 资源。

那在 Java 中，应该怎样完美地将 NIO 和异步编程结合起来呢？这里我采用了 Netty 框架，和 CompletableFuture 异步编程工具类。具体可以看看这段代码（完整代码）：

```
Executor decoderExecutor = ExecutorHelper.createExecutor(2, "decoder");
Executor ectExecutor = ExecutorHelper.createExecutor(8, "ect");
Executor senderExecutor = ExecutorHelper.createExecutor(2, "sender");
@Override
protected void channelRead0(ChannelHandlerContext ctx, HttpRequest req) throws Exception {
    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，由于 Netty 框架本身已经处理好 NIO 的问题，所以我们的工作重点放在实现“异步”处理上。Netty 框架里的 channelRead0 函数，是实现业务逻辑的地方，于是我在这个函数中，将请求处理逻辑细分为，解码（decode）、规整化（doExtractCleanTransform）、发送（send）三个步骤，然后使用 CompletableFuture 类的方法，将这三个步骤串联起来，构成了最终的异步调用链。

至此，我们终于将数据采集服务的整个请求处理过程，都彻彻底底地异步化。所有 CPU 密集型任务和 IO 密集性任务都被隔离开，在各自分配的线程里独立运行，彼此互不影响。这样， CPU 和 IO 资源，都能够得到充分利用，程序的性能也能够彻底释放出来。

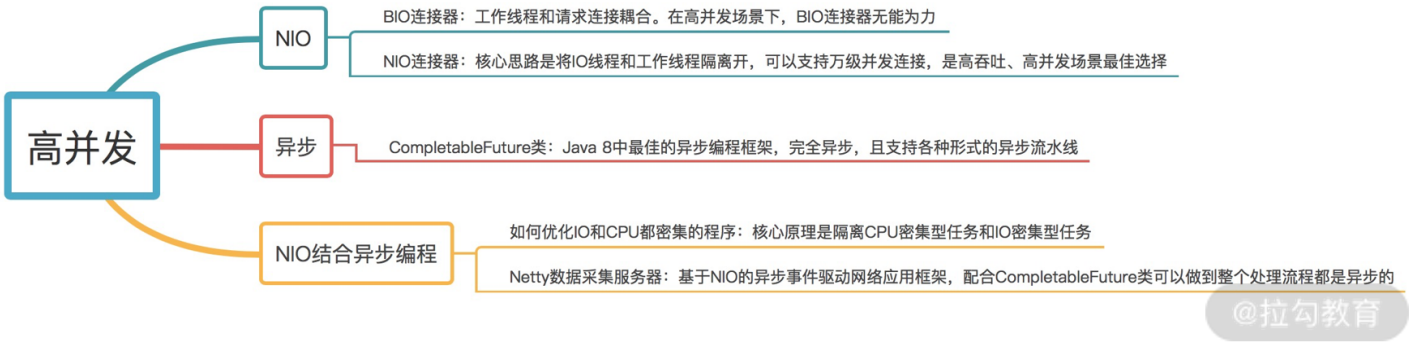
## 小结

今天，我们为了实现了高性能的数据采集服务器，详细分析了 NIO 和异步编程的工作原理，其中，还涉及了一些有关操作系统进行线程调度的知识。我们实现的基于 Netty 的，数据采集服务器，将 NIO 和异步编程技术结合起来，整个请求处理过程都是异步的，最大限度地发挥出， CPU 和 IO 资源的使用效率。

但是，有关异步的内容，还没完全讨论完。在接下来的课程中，我们将着重讨论异步系统的一些问题。我们后面会发现，异步系统的这些问题，也会出现在流计算系统中。

相信通过今天的学习，你对高并发的基础，也就是 NIO 和异步编程，已经有一定理解。那你知道如何在 Spring 框架下，实现 NIO 和异步编程吗？在留言区写出你的想法吧。

本课时精华：




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！