



下载APP



05 | IO设计：如何设计IO交互来提升系统性能？

2021-05-27 尉刚强

《性能优化高手课》

课程介绍 >



讲述：尉刚强

时长 15:52 大小 14.54M



你好，我是尉刚强。今天这节课，我想从性能的角度，来跟你聊聊 IO 交互设计。

对于一个软件系统来说，影响其性能的因素有很多，与 IO 之间的交互就是其中很关键的一个。不过可能有不少的程序员会觉得，IO 交互是操作系统底层干的事情，好像跟上层的业务关系不太大，所以很少会关注 IO 交互设计。其实，这是一种不太科学的认识。

事实上，在测试软件性能的时候，如果你发现了这样一种很奇怪的现象：虽然 CPU 使用率还没有到 100%，但是系统吞吐量却无法再提升了。那么这个时候，就很有可能是因为 IO 交互设计没有做好，导致软件的很多业务处理线程都被阻塞了，所以性能提不上去。意见，在软件设计当中，良好的 IO 交互设计对于系统性能的提升非常重要。



因此在这节课中，我想先帮你打开一下思路，了解下在软件设计中可能会碰到的各种 IO 场景，从而树立起对 IO 交互设计的正确认知。然后，我会给你介绍下针对不同的 IO 场景，应该怎样进行 IO 交互设计，才能在软件实现复杂度与性能之间实现平衡，从而帮助你提升在 IO 交互设计方面的能力。

那么下面，我们就一起来了解下，在软件设计中都有哪些 IO 场景吧。

突破对 IO 的片面认识


提到 IO，你首先想到的会是什么呢？键盘、鼠标、打印机吗？实际上，现在的软件系统中很少会用到这些东西了。一般来说，大部分程序员所理解的 IO 交互，是文件读取操作、底层网络通信，等等。

那么这里我想问你一个问题：是不是当系统中没有这些操作的时候，就不用进行 IO 交互设计了？

其实并不是的。**对一个软件系统而言，除了 CPU 和内存外，其他资源或者服务的访问也可以认为是 IO 交互。**比如针对数据库的访问、REST 请求，还有消息队列的使用，你都可以认为是 IO 交互问题，因为这些软件服务都在不同的服务器之上，直接信息交互也是通过底层的 IO 设备来实现的。

下面，我就带你来看一段使用 Java 语言访问 MongoDB 的代码实现，你会发现在软件开发中，有很多与 IO 相关的代码实现其实是比较隐蔽的，不太容易被发现。所以，**你应该对这些 IO 相关的问题时刻保持警觉，不要让它们拖垮了软件的业务性能。**

这段代码的业务逻辑是在数据库中查询一条数据并返回，具体代码如下：

 复制代码

```
1 // 从数据库查询一条数据。
2 MongoClient client = new MongoClient("*.*.*.");
3 DBCollection collection = mClient.getDB("testDB").getCollection("firstCollecti
4 BasicDBObject queryObject = new BasicDBObject("name", "999");
5 DBObject obj = collection.findOne(queryObject); // 查询操作
```

其中我们可以发现，代码中的最后一行是采用了同步阻塞的交互方式。也就是说，这段代码在执行过程中，是会把当前线程阻塞起来的，这个过程与读取一个文件的代码原理是一

样的。所以，它也是一种很典型的 IO 业务问题。

可见，我们一定要突破对传统 IO 的那种片面理解和认识，用更加全局性、系统性的视角，来认识系统中的各种 IO 场景，这才是做好基于 IO 交互设计，提升软件性能的先决条件。

那么说到这里，我们具体要如何针对系统中不同的 IO 场景，进行交互设计并提升系统性能呢？接下来，我就给你详细介绍下在软件设计中，IO 交互设计的不同实现模式，进而帮助你理解不同 IO 交互对软件设计与实现以及在性能上的影响。

IO 交互设计与软件设计

我们知道，在 Linux 操作系统内核中，内置了 5 种不同的 IO 交互模式，分别是阻塞 IO、非阻塞 IO、多路复用 IO、信号驱动 IO、异步 IO。但是，不同的编程语言和代码库，都基于底层 IO 接口重新封装了一层接口，而且这些接口在使用上也存在不少的差异。所以，这就导致很多程序员对 IO 交互模型的理解和认识不能统一，进而就对做好 IO 的交互设计与实现造成了比较大的障碍。

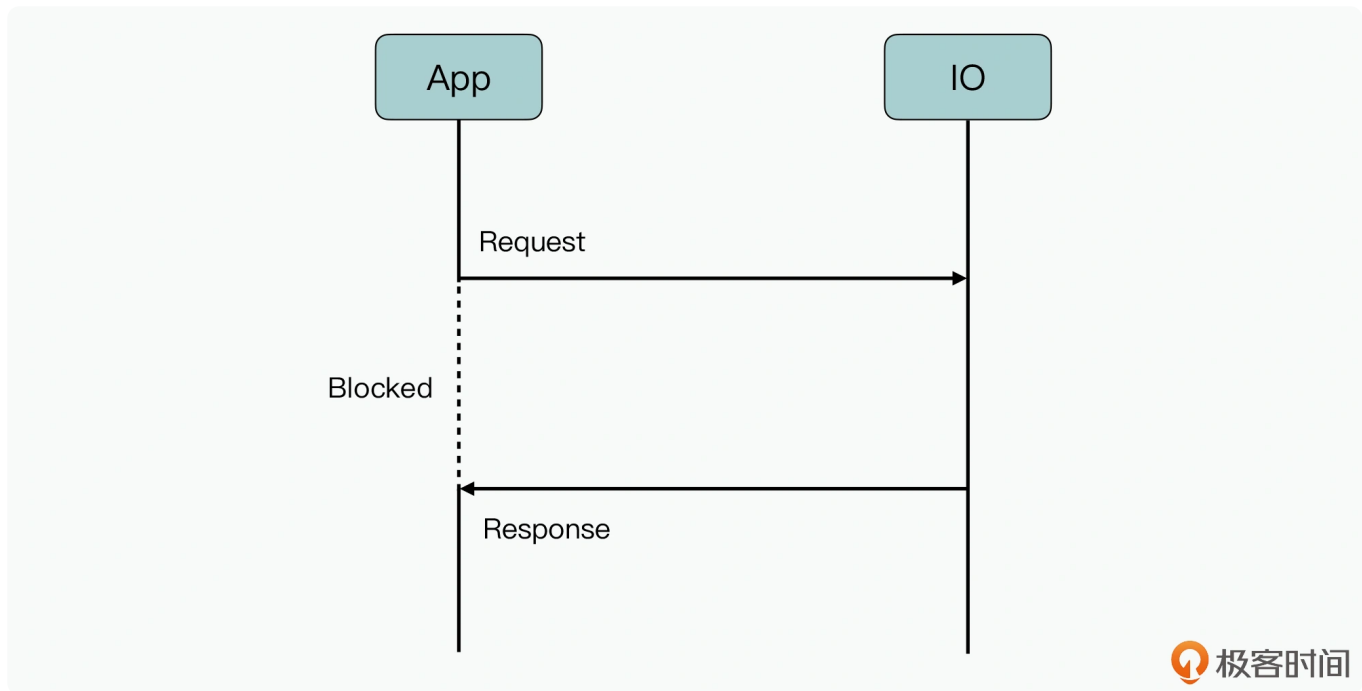
所以接下来，我就会**站在业务使用的视角**，将 IO 交互设计分为三种方式，分别是同步阻塞交互方式、同步非阻塞交互方式和异步回调交互方式，给你一一介绍它们的设计原理。我认为，只要你搞清楚这些 IO 交互设计的原理，以及理解它们在不同的 IO 场景下，如何在软件实现复杂度与性能之间做好权衡，你就离设计出高性能的软件不远了。

另外这里你要知道的是，这三种交互设计方式之间是层层递进的关系，越是靠后的方式，在 IO 交互过程中，CPU 介入开销的可能就会越少。当然 CPU 介入越少，也就意味着在相同 CPU 硬件资源上，潜在可以支撑更多的业务处理流程，因而性能就有可能更高。

同步阻塞交互方式

首先，我们来看看第一种 IO 交互方式：**同步阻塞交互方式**。

什么是同步阻塞交互方式呢？在 Java 语言中，传统的基于流的读写操作方式，其实就是采用的同步阻塞方式，前面我介绍的那个 MongoDB 的查询请求，也是同步阻塞的交互方式。也就是说，虽然从开发人员的视角来看，采用同步阻塞交互方式的程序是同步调用的，但在实际的执行过程中，程序会被操作系统挂起阻塞。我们来看看采用了同步阻塞交互方式的原理示意图：



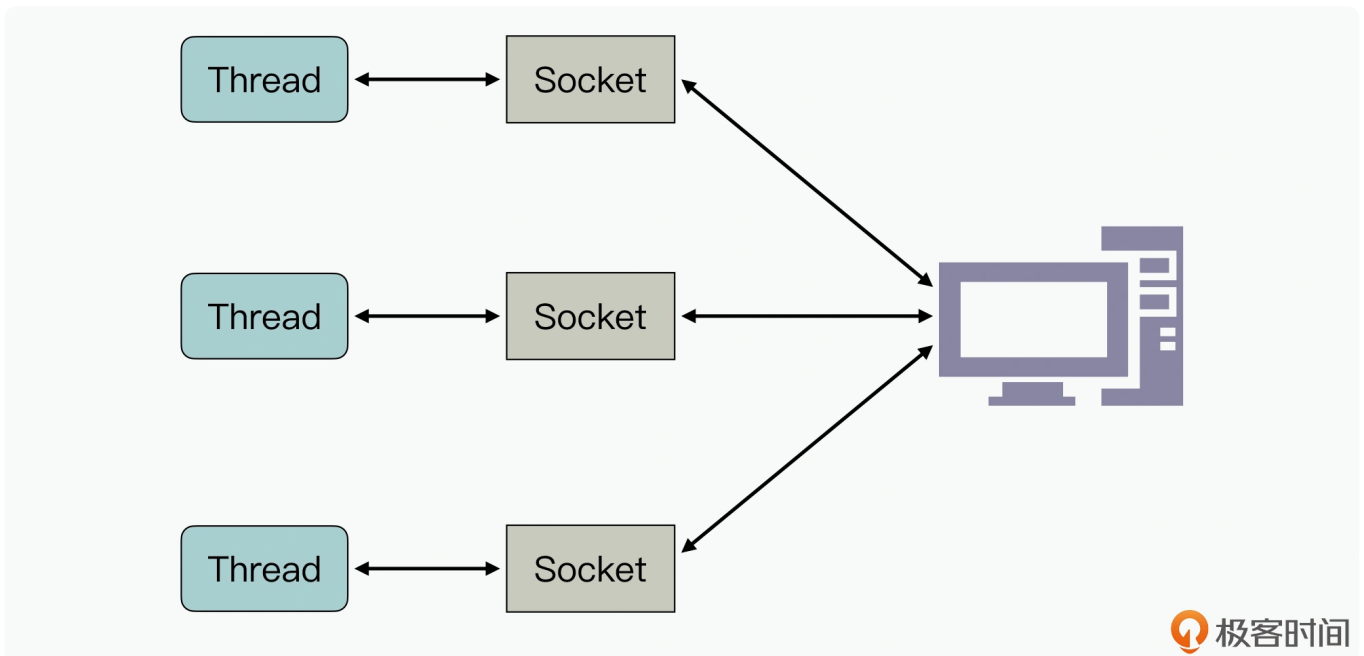
从图上你可以看到，业务代码中发送了读写请求之后，当前的线程或进程会被阻塞，只有等 IO 处理结束之后才会被唤醒。

所以这里你可能会产生一个疑问：**是不是使用同步阻塞交互方式，性能就一定会非常差呢？**实际上，并没有那么绝对，因为并不是所有的 IO 访问场景都是性能关键的场景。

我给你举个例子，针对在程序启动过程中加载配置文件的场景，因为软件在运行过程中只会加载配置文件一次，所以这次的读取操作并不会对软件的业务性能产生影响，这样我们就应该选择最简单的实现方式，也就是同步阻塞交互方式。

既然如此，你可能又要问了：**如果系统中有很多这样的 IO 请求操作时，那么软件系统架构会是怎样的呢？**

实际上，早期的 Java 服务器端经常使用 Socket 通信，也是采用的同步阻塞交互方式，它对应的架构图是这样的：



可以看到，每个 Socket 会单独使用一个线程，当使用 Socket 接口写入或读取数据的时候，这个对应的线程就会被阻塞。那么对于这样的架构来说，如果系统中的连接数比较少，即使某一个线程发生了阻塞，也还有其他的业务线程可以正常处理请求，所以它的系统性能实际上并不会非常差。

不过，现在很多基于 Java 开发的后端服务，在访问数据库的时候其实也是使用同步阻塞的方式，所以就只能采用很多个线程，来分别处理不同的数据库操作请求。而如果**针对系统中线程数很多的场景，每次访问数据库时都会引起阻塞**，那么就很容易导致系统的性能受限。

由此，我们就需要考虑采用其他类型的 IO 交互方式，避免因频繁地进行线程间切换而造成 CPU 资源浪费，以此进一步提升软件的性能。所以，同步非阻塞交互模式就被提出来，目的就是为了解决这个问题，下面我们具体来看看它的设计原理。

同步非阻塞交互方式

这里，我们先来了解下同步非阻塞交互方式的设计特点：在请求 IO 交互的过程中，如果 IO 交互没有结束的话，当前线程或者进程并不会被阻塞，而是会去执行其他的业务代码，然后等过段时间再来查询 IO 交互是否完成。Java 语言在 1.4 版本之后引入的 NIO 交互模式，其实就属于同步非阻塞的模式。

注意：实际上，Java NIO 使用的并不是完全的同步非阻塞交互方式，比如 FileChannel 就不支持非阻塞模式。另外，Java NIO 具备高性能的其中一个重要原因，是因为它增加

了缓冲机制，通过引入 Buffer 来支持数据的批量处理。

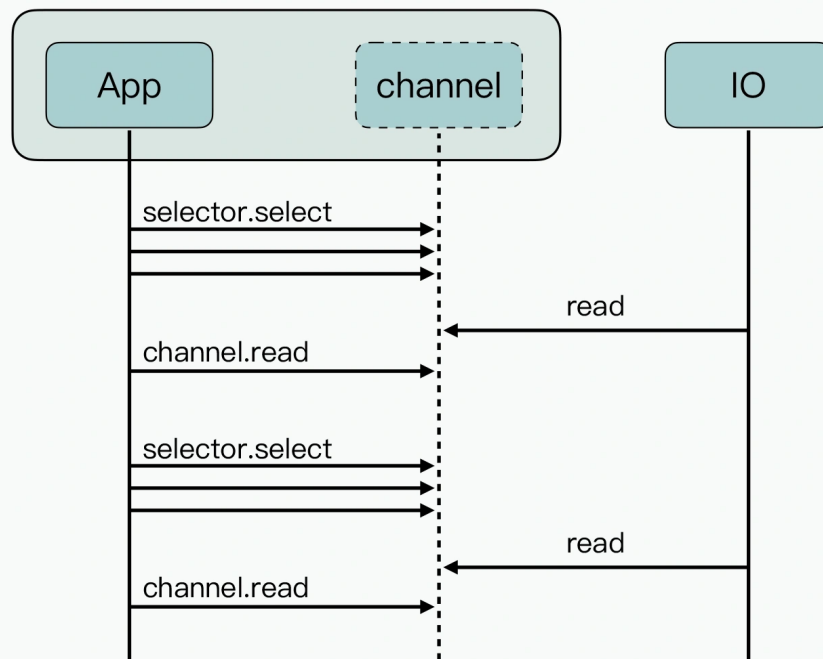
那么接下来，我们就通过一个 SocketChannel 在非阻塞模式中读取数据的代码片段，来具体看看同步非阻塞交互方式的工作原理：

[复制代码](#)

```
1 while(selector.select()>0){ //不断循环选择可操作的通道。
2     for(SelectionKey sk:selector.selectedKeys()){
3         selector.selectedKeys().remove(sk);
4         if(sk.isReadable()){ //是一个可读的通道
5             SocketChannel sc=(SocketChannel)sk.channel();
6             String content="";
7             ByteBuffer buff=ByteBuffer.allocate(1024);
8             while(sc.read(buff)>0){
9                 sc.read(buff);
10                buff.flip();
11                content+=charset.decode(buff);
12            }
13            System.out.println(content);
14            sk.interestOps(SelectionKey.OP_READ);
15        }
16    }
17 }
```

你能看到，业务代码中会不断地循环执行 `selector.select()` 操作，选择出可读就绪的 `SocketChannel`，然后再调用 `channel.read`，把通道数据读取到 Buffer 中。

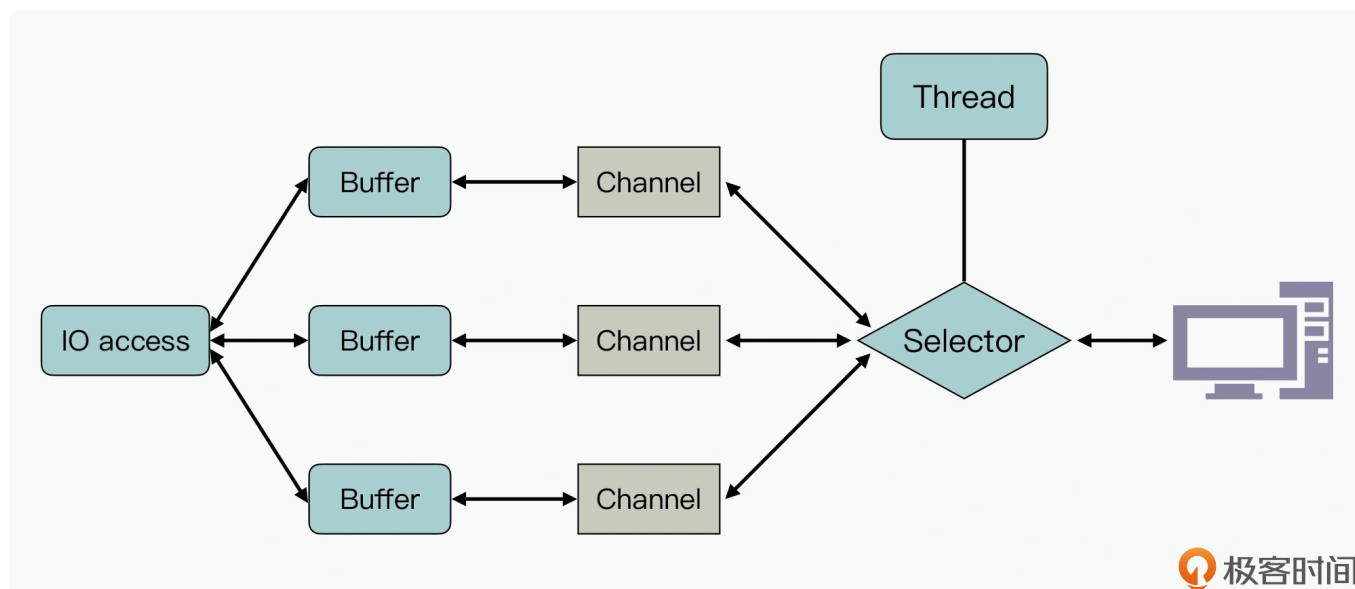
也就是说，在这个代码执行过程中，`SocketChannel` 从网口设备接收数据期间，并不会长时间地阻塞当前业务线程的执行，所以就可以进一步提升性能。这个 IO 交互方式对应的原理图如下：



从图中你能看到，当前的业务线程虽然避免了长时间被阻塞挂起，但是在业务线程中，会频繁地调用 `selector.select` 接口来查询状态。这也就是说，在单 IO 通道的场景下，使用这种同步非阻塞交互方式，性能提升其实是非常有限的。

不过，与同步阻塞交互方式刚好相反，当业务系统中同时存在很多的 IO 交互通道时，使用同步非阻塞交互方式，我们就可以复用一条线程，来查询可读就绪的通道，这样就可以大大减少 IO 交互引起的频繁切换线程的开销。

因此，在软件设计的过程中，如果你发现核心业务逻辑也是多 IO 交互的问题，你就可以基于这种 IO 同步非阻塞交互方式，来支撑产品的软件架构设计。在采用这种 IO 交互设计方式实现多个 IO 交互时，它的软件架构如下图所示：



如果你详细阅读了前面 `SocketChannel` 在非阻塞模式中读取数据的代码片段，你就会发现这个图中包含了三个很熟悉的概念，分别是 `Buffer`、`Channel`、`Selector`，它们正是 Java NIO 的核心。这里我也给你简单介绍下：`Buffer` 是一个缓冲区，用来缓存读取和写入的数据；`Channel` 是一个通道，负责后台对接 IO 数据；而 `Selector` 实现的主要功能，就是主动查询哪些通道是处于就绪状态。

所以，Java NIO 正是基于这个 IO 交互模型，来支撑业务代码实现针对 IO 进行同步非阻塞的设计，从而降低了原来传统的同步阻塞 IO 交互过程中，线程被频繁阻塞和切换的开销。

补充：但 Java 的 NIO 接口设计得并不是非常友好，代码中需要关注 `Channel` 的选择细节，而且还需要不断关注 `Buffer` 的状态切换过程。因此，基于这套接口的代码实现起来会比较复杂。

那么有没有什么办法可以帮助降低代码实现的复杂度呢？我们可以基于 Java NIO 设计的 **Netty 框架**来帮助屏蔽这些细节问题。`Netty` 框架是一个开源异步事件编程框架，它的系统性能非常高，同时在接口使用上也非常友好，所以目前使用也很广泛。如果你在开发网络通信的高性能服务器产品，那么你也可以考虑使用这种框架（`Elasticsearch` 底层实际上就是采用的这种机制）。

不过，基于同步非阻塞方式的 IO 交互设计，如果在并发设计中，没有平衡好 IO 状态查询与业务处理 CPU 执行开销管理，就很容易导致软件执行期间存在大量的 IO 状态的冗余查询，从而造成对 CPU 资源的浪费。

因此，我们还需要从业务角度的 IO 交互设计出发，来进一步减少 IO 对 CPU 带来的额外开销，而这就是我接下来要给你介绍的异步回调交互方式的重要优势。

异步回调交互方式

所谓异步回调的意思就是，当业务代码触发 IO 接口调用之后，当前的线程会接着执行后续处理流程，然后等 IO 处理结束之后，再通过回调函数来执行 IO 结束后的代码逻辑。

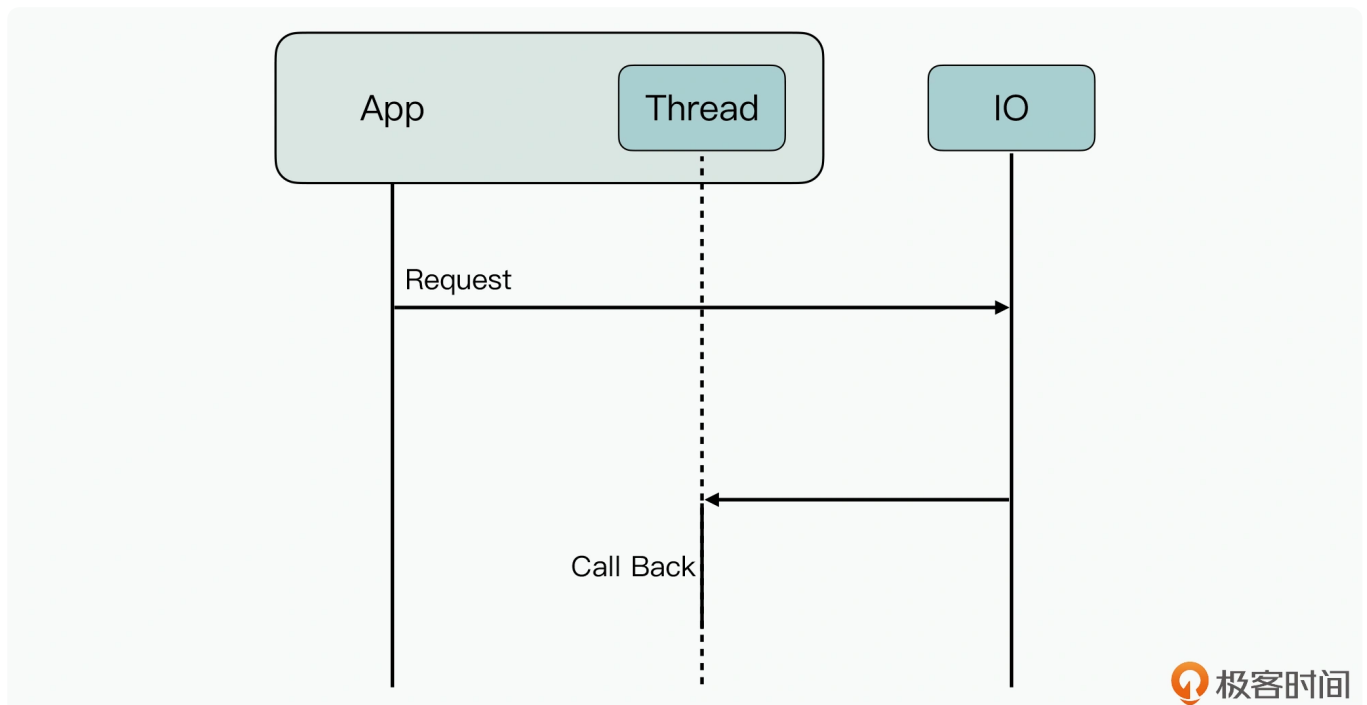
这里我们同样来看一段代码示例，这是 Java 语言针对 `MongoDB` 的插入操作，它采用的就是异步回调的实现方式：


```
1 Document doc = new Document("name", "Geek")
2     .append("info", new Document("age", 203).append("sex", "male"))
3
4 collection.insertOne(doc, new SingleResultCallback<Void>() {
5     @Override
6     public void onResult(final Void result, final Throwable t) {
7         System.out.println("Inserted success");
8     }
9 });
```

我们可以发现，在这段代码中，调用 `collection.insertOne` 在插入数据时，同时还传入了回调函数。

实际上，这个 MongoDB 访问接口在底层使用是 Netty 框架，只是重新封装了接口使用方法而已。

由此，我们就可以最大化地减少 IO 交互过程中 CPU 参与的开销。这种 IO 交互方式的原理图如下所示：



从这个图中可以看到，在使用异步回调这种处理方式时，回调函数经常会被挂载到另外一个线程中去执行。所以使用这种方式会有有一个好处，就是**业务逻辑不需要频繁地查询数据**，但同时，它也会**引入一个新问题**，那就是回调处理函数与正常的业务代码被割裂开了，这会给代码实现增加不少的复杂度。

我给你举个例子，如果代码中的回调函数在处理过程中，还需要进一步执行其他 IO 请求时，如果再使用回调机制，那么就会出现万恶的回调嵌套问题，也就是回调函数中再嵌套一个回调函数，这样一直嵌套下去，代码就会很难阅读和维护。

所以后来，在 Node.js 中就引入了 async 和 await 机制（在 C++、Rust 中，也都引入了类似的机制），比较好地解决了这个问题。我们使用这个机制，可以将背后的回调函数机制封装到语言内部底层实现中，这样我们就依旧可以使用串行思维模式来处理 IO 交互。

而且，当有了这种机制之后，IO 交互方式对软件设计架构的影响就比较少了，所以像 Node.js 这样的单进程模型也可以处理非常多的 IO 请求。

另外，**使用异步回调交互方式还有一个好处**，因为现在的互联网场景中，对数据库、消息队列、REST 请求都是非常频繁的，所以如果你采用异步回调方式，比较有可能将 IO 阻塞引起的线程切换开销，还有频繁查询 IO 状态的时间开销，都降低到比较低的状态。

最后我还想告诉你的是，实际上，IO 交互设计不仅与语言系统的并发设计相关性很大，而且与缓冲区（Buffer）的设计和实现关系也很紧密，我们在进行 IO 交互设计时，其实需要权衡很多因素，这是一个挺复杂的工作，我们一定不能小看它。

小结

今天这节课，我通过一些常见的业务代码逻辑，带你突破了之前对 IO 的片面认识，帮助你更加清楚地识别出系统中的各种 IO 交互场景。另外，我也给你重点介绍了在应用软件设计过程中，三种常用的 IO 同步交互设计，帮助你理解不同 IO 交互对软件设计与实现，以及在性能上的影响。你可以基于这些理解和认识，来指导软件架构设计，避免因为 IO 交互问题而引起比较严重的性能问题。

其实，还有一种在软件设计中使用的比较少的 IO 交互同步方式，我并没有给你介绍，这种 IO 交互方式叫做**零协调交互方式**，它在高性能嵌入式系统设备或高性能服务器中使用会比较多。比如，你可以使用 DPDK 技术，来减少网络数据接收期间操作系统内核的参与，或者使用链式 DMA 拷贝，来减少内存拷贝中的 CPU 介入等。这里你简单了解下即可，如果还想要深入学习的话，你可以参考 [这个文档](#)。

思考题

今天课程中介绍的异步回调交互方式，操作系统底层是不是采用的 Linux 内核的异步 IO 交互方式呢？

欢迎在留言区分享你的答案和思考。如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

分享给需要的人，Ta订阅后你可得 **20元** 现金奖励

👍 赞 1

💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 04 | 缓存设计：做好缓存设计的关键是什么？

下一篇 06 | 通信设计：请不要让消息通信拖垮了系统的整体性能

更多学习推荐

Java 面试必考 300 题 最新汇总

限时免费领取



精选留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

💬 写留言

