復旦大學

本科生课程论文

题目：Java、JavaScript 和 C++对异步 IO 的实现方法及其效率

课程名称：系统程序设计　课程代码：SOFT130024

姓　　名：王栋辉　学　　号：18302010012

学　　院：软件学院　专　　业：软件工程

摘要

通过对 IO 的了解，我们知道，CPU 的处理速度远远高于磁盘、网络等 IO。在一个线程中，CPU 执行代码的速度极快，然而一旦遇到 IO 操作，就需要等待 IO 操作完成，才能进行下一步操作，这称为同步 IO。但是，同步 IO 阻塞了当前线程，影响了代码的执行效率。

一种方案是，使用多线程或多进程来并发执行代码。

另一种方案是，使用异步 IO。当前进程（线程）只发出 IO 指令，并不等待 IO 结果，而是继续执行其他代码。等到 IO 结果返回时，再通知 CPU 进行处理。

当然，这两种方案并不独立，有时候，我们也可以从主线程分出子线程来处理异步 IO 操作。

本文首先将阐述 Java、JavaScript、C++三种语言对异步 IO 的解决方案。其次将设计这三种语言的四个具体案例，对异步 IO，尤其是异步读操作的代码和结果进行分析和比较。

最后，本文将得出三种语言对异步 IO 的不同效率。

关键词：异步 IO openMP Java JavaScript C++

目录

第一章 异步 IO 方法简介

1.1 使用 openMP 对 C 语言代码进行改造

代码：

```
1.  #include <omp.h>
2.  #include <stdio.h>
3.
4.   int main()
5.  {
6.      #pragma omp parallel for
7.      for(int i=0; i<10; i++){
8.          printf("%d",i);
9.      }
10.
11.     return 0;
12. }
```

1.2 了解 Java 和 JavaScript 实现异步 IO 的方式

1.2.1 Java 实现异步 IO 的方式

Java AIO，又称为 NIO2.0/NIO.2，是 JDK 1.7 之后引入的包。作为 NIO 的升级版本，它提供了异步非阻塞的 IO 操作方式，所以被称为 AIO(Asynchronous IO)。

当进行读写操作时，只需要直接调用 API 的 read 和 write 方法即可，这两种方法均为异步的。对于读操作，当有流可读取时，操作系统会将可读的流传入 read 方法的缓冲区；对于写操作，当操作系统将 write 方法传递的流写入完毕时，操作系统主动通知应用程序。即可以理解为，read/write 方法都是异步的，完成后会主动调用回调函数。

JDK1.7 中，AIO 作为 NIO.2，主要在 java.nio.channels 包下增加了以下四个异步通道。

```
AsynchronousSocketChannel
AsynchronousServerSocketChannel
AsynchronousFileChannel
AsynchronousDatagramChannel
```

以下是一个 AIO 版 Socket 实现案例：[1]

```
// AIO线程复用版
```

[1] 代码引自 https://www.imooc.com/article/265871

```
Thread sThread = new Thread(new Runnable() {
    @Override
    public void run() {
        AsynchronousChannelGroup group = null;
        try {
            group =
AsynchronousChannelGroup.withThreadPool(Executors.newFixedThreadPool(4)
);
            AsynchronousServerSocketChannel server =
AsynchronousServerSocketChannel.open(group).bind(new
InetSocketAddress(InetAddress.getLocalHost(), port));
            server.accept(null, new
CompletionHandler<AsynchronousSocketChannel,
AsynchronousServerSocketChannel>() {
                @Override
                public void completed(AsynchronousSocketChannel result,
AsynchronousServerSocketChannel attachment) {
                    server.accept(null, this); // 接收下一个请求
                    try {
                        Future<Integer> f =
result.write(Charset.defaultCharset().encode("你好，世界"));
                        f.get();
                        System.out.println("服务端发送时间：" + new
                        result.close();
                    } catch (InterruptedException | ExecutionException
| IOException e) {
                        e.printStackTrace();
                    }
                }

                @Override
                public void failed(Throwable exc,
AsynchronousServerSocketChannel attachment) {
                }
            });
            group.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
    }
});
sThread.start();
```

```
// Socket 客户端
AsynchronousSocketChannel client = AsynchronousSocketChannel.open();
Future<Void> future = client.connect(new
InetSocketAddress(InetAddress.getLocalHost(), port));
future.get();
ByteBuffer buffer = ByteBuffer.allocate(100);
client.read(buffer, null, new CompletionHandler<Integer, Void>() {
    @Override
    public void completed(Integer result, Void attachment) {
        System.out.println("客户端打印: " + new String(buffer.array()));
    }

    @Override
    public void failed(Throwable exc, Void attachment) {
        exc.printStackTrace();
        try {
            client.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
});
Thread.sleep(10 * 1000);
```

1.2.2 JavaScript 实现异步 IO 的方式

JavaScript 异步编程[2]

JavaScript 语言采用了“单线程”模式。

好处：实现起来比较简单，执行环境相对单纯；

坏处：只要有一个任务耗时很长，后面的任务都必须排队等着，会拖延整个程序的执行。常见的表现是浏览器无响应（假死）。

为了解决这个问题，JavaScript 语言将任务的执行模式分成两种：同步（Synchronous）和异步（Asynchronous）。

"同步模式"就是上一段的模式，后一个任务等待前一个任务结束，然后再执行，程序的执行顺序与任务的排列顺序是一致的、同步的；"异步模式"则完全不同，每一个任务有一个或多个回调函数（callback），前一个任务结束后，不是执行后一个任务，而是执行回调函数，后一个任务则是不等前一个任务结束就执行，所以程序的执行顺序与任务的排列顺序是不一致的、异步的。

以下是"异步模式"编程的 4 种方法：

一、回调函数

[2] 引自 https://blog.csdn.net/youhan26/article/details/47057559

假定有两个函数 f1 和 f2，后者等待前者的执行结果。

```
f1();
f2();
```

如果 f1 是一个很耗时的任务，可以考虑改写 f1，把 f2 写成 f1 的回调函数。

```
function f1(callback){
    setTimeout(function () {
        // f1的任务代码
        callback();
    }, 1000);
}
```

执行代码就变成下面这样：

```
f1(f2);
```

采用这种方式，我们把同步操作变成了异步操作，f1 不会堵塞程序运行，相当于先执行程序的主要逻辑，将耗时的操作推迟执行。

回调函数的优点是简单、容易理解和部署，缺点是不利于代码的阅读和维护，各个部分之间高度耦合（Coupling），流程会很混乱，而且每个任务只能指定一个回调函数。

二、事件监听

另一种思路是采用事件驱动模式。任务的执行不取决于代码的顺序，而取决于某个事件是否发生。

还是以 f1 和 f2 为例。首先，为 f1 绑定一个事件（这里采用的 jQuery 的写法）。

```
f1.on('done', f2);
```

上面这行代码的意思是，当 f1 发生 done 事件，就执行 f2。然后，对 f1 进行改写：

```
function f1(){
    setTimeout(function () {
        // f1的任务代码
        f1.trigger('done');
    }, 1000);
}
```

f1.trigger('done')表示，执行完成后，立即触发 done 事件，从而开始执行 f2。

这种方法的优点是比较容易理解，可以绑定多个事件，每个事件可以指定多个回调函数，而且可以"去耦合"（Decoupling），有利于实现模块化。缺点是整个程序都要变成事件驱动型，运行流程会变得很不清晰。

三、发布/订阅

上一节的"事件"，完全可以理解成"信号"。

我们假定，存在一个"信号中心"，某个任务执行完成，就向信号中心"发布"（publish）一个信号，其他任务可以向信号中心"订阅"（subscribe）这个信号，从而知道什么时候自己可以开始执行。这就叫做"发布/订阅模式"（publish-subscribe pattern），又称"观察者模式"（observer pattern）。

这个模式有多种实现，下面采用的是 Ben Alman 的 Tiny Pub/Sub，这是 jQuery 的一个插件。

首先，f2 向"信号中心"jQuery 订阅"done"信号。

```
jQuery.subscribe("done", f2);
```

然后，f1 进行如下改写：

```
function f1(){
    setTimeout(function () {
        // f1的任务代码
        jQuery.publish("done");
    }, 1000);
}
```

jQuery.publish("done")的意思是，f1 执行完成后，向"信号中心"jQuery 发布"done"信号，从而引发 f2 的执行。

此外，f2 完成执行后，也可以取消订阅（unsubscribe）。

```
jQuery.unsubscribe("done", f2);
```

这种方法的性质与"事件监听"类似，但是明显优于后者。因为我们可以通过查看"消息中心"，了解存在多少信号、每个信号有多少订阅者，从而监控程序的运行。

四、Promises 对象

Promises 对象是 CommonJS 工作组提出的一种规范，目的是为异步编程提供统一接口。

简单说，它的思想是，每一个异步任务返回一个 Promise 对象，该对象有一个 then 方法，允许指定回调函数。比如，f1 的回调函数 f2,可以写成：

```
f1().then(f2);
```

f1 要进行如下改写（这里使用的是 jQuery 的实现）：

```
function f1(){
    var dfd = $.Deferred();
    setTimeout(function () {
        // f1的任务代码
        dfd.resolve();
```

```
        }, 500);
        return dfd.promise;
    }
```

这样写的优点在于，回调函数变成了链式写法，程序的流程可以看得很清楚，而且有一整套的配套方法，可以实现许多强大的功能。

比如，指定多个回调函数：

```
    f1().then(f2).then(f3);
```

再比如，指定发生错误时的回调函数：

```
    f1().then(f2).fail(f3);
```

而且，它还有一个前面三种方法都没有的好处：如果一个任务已经完成，再添加回调函数，该回调函数会立即执行。所以，你不用担心是否错过了某个事件或信号。这种方法的缺点就是编写和理解，都相对比较难。

JavaScript 异步编程补充之 async/await[3]

在 ES7 中，异步解决方案 async/await 应运而生。它是目前社区中公认的优秀异步解决方案。

async/await 实际上是 Generator 函数的语法糖，进一步封装了 ES6 中对异步处理方案：Generator 和 Promise。

await 后面调用的函数需要返回一个 promise，且 await 需要搭配 async 使用。

因为 chrome 和 Node.js 还没有支持 aysnc/await，所以引入 babel 库：

```
$ npm install babel-core --save
$ npm install babel-preset-es2015 --save
$ npm install babel-preset-stage-3 --save
```

以下是一个使用 async/await 的案例：

需要编写两个文件，一个是启动的 js 文件 index.js，另外一个是真正执行程序的 js 文件 async.。

启动文件 index.js

```
require('babel-core/register');
require('./async.js');
```

真正执行程序的 async.js

```
const request = require('request');
```

[3] 引自 https://www.cnblogs.com/cpselvis/p/6344122.html

```
const options = {
  url: 'https://api.github.com/repos/cpselvis/zhihu-crawler',
  headers: {
    'User-Agent': 'request'
  }
};

const getRepoData = () => {
  return new Promise((resolve, reject) => {
    request(options, (err, res, body) => {
      if (err) {
        reject(err);
      }
      resolve(body);
    });
  });
};

async function asyncFun() {
 try {
    const value = await getRepoData();
    //      和上面的koa类似，如果有多个异步流程，可以放在这里，比如
    // const r1 = await getR1();
    // const r2 = await getR2();
    // const r3 = await getR3();
    //
每个await相当于暂停，执行await之后会等待它后面的函数（不是generator）返回值之
    return value;
  } catch (err) {
    console.log(err);
  }
}

asyncFun().then(x => console.log(`x: ${x}`)).catch(err =>
console.error(err));
```

JavaScript 多线程之 Web Worker[4]

尽管 JavaScript 语言采用了单线程模型，但在 HTML5 中，提供了 Web Worker，为 JavaScript 创造了多线程环境（注意，Web Worker 本身是在 HTML 标准中规定的，而非 JavaScript 的标准），允许主线程创建 Worker 线程，将一些任务分配给后者运行。

在主线程运行的同时，Worker 线程在后台运行，两者互不干扰。等到 Worker 线程完成计算任务，再把结果返回给主线程。这样的好处是，一些计算机密集型或高延迟的任务，被

[4] 部分引自 http://www.ruanyifeng.com/blog/2018/07/web-worker.html

Worker 线程负担了，主线程（通常负责 UI 交互）就会很流畅，不会被阻塞或拖慢。

Nodejs 异步 IO 的实现[5]

在浏览器端，由于对资源安全的限制，以及本身适合渲染页面的功能限制，所以即使 JavaScript 对异步和多线程有所支持，也很难对 IO 进行支持。

因此，一般 IO 由服务器端进行负责。Nodejs 就是优秀的服务器端 JavaScript 运行环境之一。它提供了对于异步 IO 的支持和实现。

Nodejs 对在 fs 库总异步读写进行了封装，且其一般 IO 函数都是异步的（要特地调用同步 IO 函数，需调用含 Sync 的函数），如 fs.open()、fs.read()、fs.write()都是异步的。

nodejs 的核心之一就是非阻塞的异步 IO。用了一段 js 代码 test-fs-read.js 做测试，代码如下：

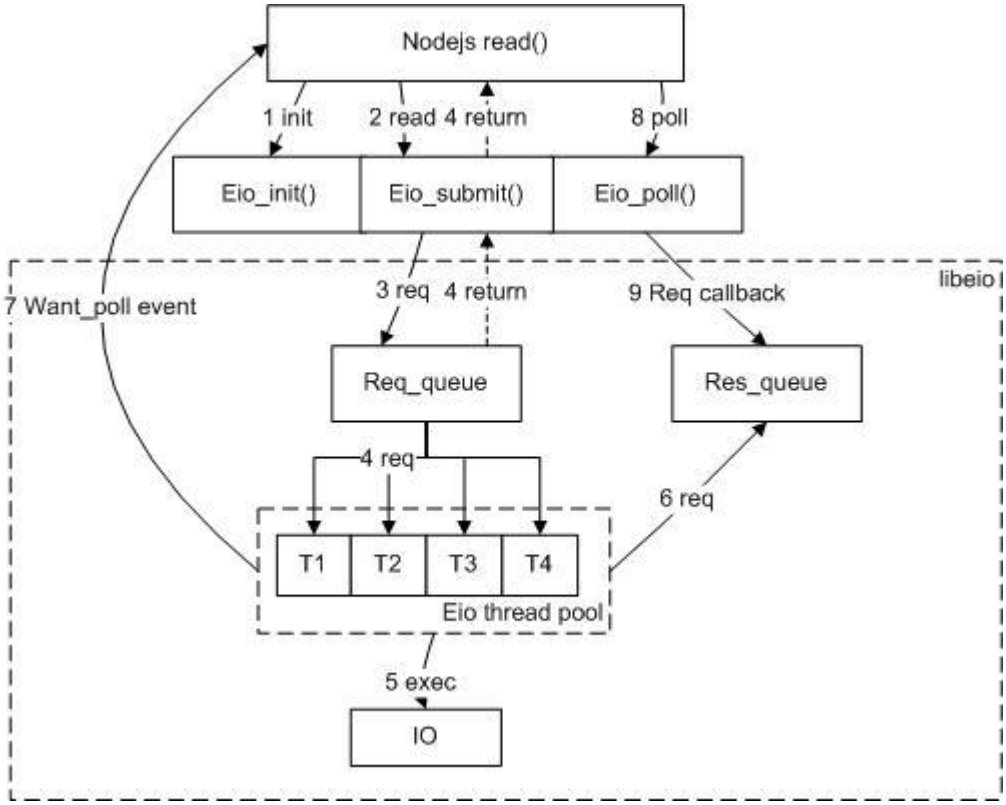
```
var path = require('path'),
fs = require('fs'),
filepath = path.join(__dirname, 'experiment.log'),
fd = fs.openSync(filepath, 'r');

fs.read(fd, 12*1024*1024, 0, 'utf-8', function(err, str, bytesRead) {
    console.log('[main thread] execute read callback');
});
console.log('[main thread] execute operation after read');
```

异步的结果是：先打印出[main thread] execute operation after read，在打印出[main thread] execute read callback。

Nodejs 的 IO 执行过程如下图：

[5] 部分引自 https://blog.csdn.net/yczz/article/details/7015463



1.3 基于 C++实现异步 IO

使用了 openMP。

main 函数：调用了 omp_set_num_threads 函数，设置 1 个子线程。

接着，用 pragma omp parallel 包裹住整个代码块，表示用设置的这 1 个子线程执行异步读函数 asyncRead。

最后，用 pragma omp master 包裹住计算函数 compute，表示该函数由主进程执行。

```
int main(int argc, char* argv[])
{
        //设置一个子线程，异步执行读操作
        omp_set_num_threads(1);
        #pragma omp parallel
    {

        asyncRead(argv[1]);

        #pragma omp master
    {
        //主线程计算斐波那契数列
        compute();
```

```
    }

    }

    return 0;

}
```

异步 IO（读）函数：创建读操作文件流 ifstream 类的实例 in，调用 in.open 方法打开文件，调用 in.eof 方法判断是否读到文件末尾，调用 in.read 方法每次读一个字符，并将变量 count 自增 1，从而用 count 统计总字符数。

```
void asyncRead(char* path){
    clock_t start,end;
        ifstream in;
        int count = 0;

        cout << "Start reading" << endl;
        start = clock();
        in.open(path);
        if(!in){
            cerr << "打开文件出错" << endl;
            exit(1);
        }

        char ch;
        while(!in.eof()){
            in.read(&ch,1);
            //cout << ch;
            count++;
        }
        end = clock();
        cout << "End reading, words read: " << count << " and it takes
" << (double)(end-start) << " ms!" << endl;
}
```

第二章 异步 IO 效率的实验及分析

2.1 运行并行模式下的 C 语言代码

1. 当编译时不开启与 OpenMP 一起运行的选项时（即不开启-fopenmp），代码只在主线程中运行，所以结果将以顺序输出。

运行命令：

```
gcc test.c -o test
test.exe  //windows
```

运行结果：

```
C:\Users\lenovo\Desktop\系统程序论文\1.1 使用openMP对C语言代码进行改造>gcc test.c -o test

C:\Users\lenovo\Desktop\系统程序论文\1.1 使用openMP对C语言代码进行改造>test.exe
0123456789
C:\Users\lenovo\Desktop\系统程序论文\1.1 使用openMP对C语言代码进行改造>
```

2. 当编译时开启与 OpenMP 一起运行的选项时，代码开启多线程调用 for 循环，但线程数随机。所以结果以随机顺序输出。

运行命令：

```
gcc test.c -fopenmp -o test
test.exe  //windows
```

运行结果：

```
C:\Users\lenovo\Desktop\系统程序论文\1.1 使用openMP对C语言代码进行改造>gcc test.c -fopenmp -o test

C:\Users\lenovo\Desktop\系统程序论文\1.1 使用openMP对C语言代码进行改造>test.exe
5723016984
C:\Users\lenovo\Desktop\系统程序论文\1.1 使用openMP对C语言代码进行改造>test.exe
8237501469
C:\Users\lenovo\Desktop\系统程序论文\1.1 使用openMP对C语言代码进行改造>
```

2.2 运行第一章中三种异步 IO，进行对照试验

三种异步 I0 代码的基本逻辑是：主线程先调用异步 I0 代码，再调用一个递归函数运算斐波那契数列的第 44 个数。观察异步 I0 代码和递归函数的运行时间和输出提示语句的先后顺序。

2.2.1 Java 异步 IO 代码与运行结果

代码：详见文件夹 1.2.1 Java 实现异步 IO

代码概述：

main 方法：先调用异步读函数 asyncRead，再调用计算斐波那契数列函数 fn(44)，分别计时打印提示信息。

```
public static void main(String[] args) throws Exception{
        asyncRead("src/passage.txt");
        System.out.println("Start computing");
        long start1 = System.currentTimeMillis();
        int result1 = fn(44);
        long end1 = System.currentTimeMillis();
        System.out.println("Finish computing, the result is " + result1
+ " and it takes " + (end1 - start1) + "ms");
}
```

异步 IO（读）函数：asyncRead，创建了 AsychronousFileChannel 类的实例 channel 打开文件，调用 channel.read 方法，并传入 CompletionHandler 类的实例来重写 completed 和 failed 方法。其中 completed 方法在文件成功读完后会调用。这样写相当于回调式的异步读。

```
public static void asyncRead(String path) throws Exception{
       System.out.println("Start reading");
       long start = System.currentTimeMillis();

       Path file = Paths.get(path);
       AsynchronousFileChannel channel =
AsynchronousFileChannel.open(file);

       ByteBuffer buffer = ByteBuffer.allocate(100000);

       // 回调式
       channel.read(buffer, 0, buffer,new CompletionHandler<Integer,
ByteBuffer>() {
        @Override
        public void completed(Integer result, ByteBuffer attachment) {
            long end = System.currentTimeMillis();

            //反转buffer
            buffer.flip();
           //System.out.println(Charset.forName("UTF-
8").decode(attchment).toString());
            System.out.println("Finish reading, words read " + result +
" and it takes " + (end - start) + "ms");
        }

        @Override
        public void failed(Throwable exc, ByteBuffer attachment) {
            System.out.println("read error");
```

```
        }
      });
}
```

结果：

```
asyncIO
"C:\Program Files\Java\jdk-11.0.11\bin\java.exe" "-javaagent:C:\Progr
Start reading
Start computing
Finish reading, words read 4631 and it takes 21ms
Finish computing, the result is 701408733 and it takes 3393ms

Process finished with exit code 0
```

分析：main 主线程先从异步读开始。而由于异步非阻塞，所以在读取文件的过程中，主线程继续调用计算函数。在计算函数还在运行时，异步读的子线程读取文件完毕，并调用 completed 回调函数，打印文件信息。最后主线程计算函数运行完毕，打印结果信息。

整个程序分成了两个部分：处理异步 IO 的子线程，和运行计算函数并接收子线程结果的主进程。

2.2.2 JavaScript 异步 IO 代码与结果

一、Nodejs 解决方案

代码：详见文件夹 1. 2. 2JavaScript 实现异步 IO/nodejs-solution

代码概述：

主线程：这次异步读代码块并没有封装成函数。

```
//异步IO代码块（详见下面）
……


//计算斐波那契数列
console.log("Start computing");
console.time("Time for computing");
let result = fn(44);
console.timeEnd("Time for computing");
console.log(`Finish computing, the result is ${result}`);
```

异步 IO（读）代码块：使用了 nodejs 的 fs 库的 API —— fs.open、fs.readFile。在 fs.readFile 的回调函数中结束异步读操作的计时以及打印结果信息。

```
fs.open(filePath,'r',(err,fd)=>{
```

```
    if(err){
        if(err.code === 'ENOENT'){
            console.log(`File:${filePath} doesn't exist!`);
        }
        else{
            console.log("error:",err);
        }
        return false;
    }
});

let buf;

//异步读
console.log("Start reading");
console.time("Time for read");
fs.readFile(filePath,(err,data)=>{
    if(err){
        console.log("error:",err);
        return false;
    }
    else{
        buf = data;
        console.timeEnd('Time for read');
        console.log(`End reading, words read: ${buf.length}`);
    }
});
```

结果:

```
C:\Users\lenovo\Desktop\系统程序论文\1.2.2JavaScript实现异步IO\nodejs-soluction>npm start

> JavaScript-async-io@1.0.0 start C:\Users\lenovo\Desktop\系统程序论文\1.2.2JavaScript实现异步IO\nodejs-soluction
> node asyncIO.js

Start reading
Start computing
Time for computing: 9470.028ms
Finish computing, the result is 701408733
Time for read: 9522.105ms
End reading, words read: 4631
```

分析：可以看到，虽然计算函数在异步读代码块后被调用，但异步读的结果一直阻塞到主线程结束后才被执行和打印。最大的证据是：Time for read 和 Time for computing 是接近的。原因在于：nodejs 运行环境仍然是单线程的，异步函数的返回结果实际上会暂存在 JavaScript 的内置消息队列中。只有当主线程的代码执行完毕后，才会去查看并执行消息队列中的返回结果。

这里所说的“阻塞”实际上并不准确。异步读的代码块在读取完文件后，就被释放了，但是其执行结果仍被暂存在消息队列。我们这里记录的读取事件 Time for read 实际也并不准确，它大概只是因为 console 函数被单线程调用，所以异步读的计时结束迟迟没有被调用，直到主线程代码结束，即计算函数计时结束。计时结束函数才得以被调用。

二、Web Worker 解决方案

代码：详见文件夹 1.2.2JavaScript 实现异步 IO/webworker-solution

代码概述：

主线程：在 Index.html 中，创建了一个 Worker 类实例 worker，分出一个子线程执行 worker.js 中的异步读代码。接着调用计算函数。

注意：所有脚本代码都包含在了文件控件 fileInput 的 onchange 事件的回调函数中。因为浏览器不能直接读取本地文件，而是通过文件控件手动添加文件，所以这段代码要在用户添加完想要读取的文件后执行。

```
let fileInput = document.getElementById('file');
fileInput.onchange = function(){

    //创建worker
    let worker = new Worker('worker.js');
    console.log('Start reading');
    worker.postMessage(fileInput.files);
    worker.onmessage = function(event){
        console.log(event.data);
    }


    //执行计算函数
    console.log('Start computing');
    console.time('Time for computing');
    let result = fn(44);
    console.timeEnd('Time for computing');
    console.log(`End computing, the result is ${result}`);
    function fn(n){
        if(n===0) return 0;
        if(n===1) return 1;

        return fn(n-1)+fn(n-2);
    }
}
```

异步 IO（读）代码：当主线程调用 worker.postMessage 方法，就会触发 worker 的 message 事件。所以在 worker.js 中，子线程要监听这一事件。在事件的回调函数中，拿到主线程塞入的 event 对象（此处声明为 e）中的 file 对象，即添加的文件。创建 FileReader 类实例 fr，调用 fr.readAsText 方法读取文件。

当文件读取完毕，触发 fs 的 onload 事件，在其回调函数中结束计时并打印文件信息。

```
self.addEventListener('message',function(e){
    if(self.FileReader){
        var fr = new FileReader();
        console.time('Time for read');
```

```
        fr.readAsText(e.data[0],'utf-8');
        fr.onload = function(){
            let res = this.result;
            console.timeEnd('Time for read');
            self.postMessage(`End reading, words read ${res.length}`)
        }
    }
})
```

结果：

```
Start reading                                                    index.html:15
Start computing                                                  index.html:21
Time for computing: 9901.638916015625 ms                         index.html:24
End computing, the result is 701408733                           index.html:25
Time for read: 2.47412109375 ms                                    worker.js:8
End reading, words read 4501                                     index.html:18
>
```

分析：与 nodejs 不同，web worker 因为实现了真正的多线程，所以 console 函数真正地由两个线程调用，所以异步读的计时准确表示了读取文件的时间。

然而，因为 JavaScript 对异步函数的处理没有改变，所以异步读的结果还是等到主线程代码执行完毕后才打印，所以会看到计算函数的结果先打印出来，异步读的结果再打印出来。

2.2.3 C++异步 IO 代码与结果

代码：详见文件夹 1. 3 基于 c++实现异步 IO
代码概述：见本文 1. 3 基于 C++实现异步 IO
结果：

```
C:\Users\lenovo\Desktop\系统程序论文\1.3基于c++实现异步IO>g++ asyncIO.cpp -fopenmp -o asyncIO

C:\Users\lenovo\Desktop\系统程序论文\1.3基于c++实现异步IO>asyncIO.exe
Start reading
End reading, words read: 4596 and it takes 1 ms!
Start computing
End computing, the result is 701408733 and it takes 4481 ms!

C:\Users\lenovo\Desktop\系统程序论文\1.3基于c++实现异步IO>g++ asyncIO.cpp -fopenmp -o asyncIO

C:\Users\lenovo\Desktop\系统程序论文\1.3基于c++实现异步IO>asyncIO.exe
Start readingStart reading

End reading, words read: 4596 and it takes 0 ms!
Start computing
End reading, words read: 4596 and it takes 1 ms!
End computing, the result is 701408733 and it takes 4435 ms!

C:\Users\lenovo\Desktop\系统程序论文\1.3基于c++实现异步IO>_
```

分析：奇怪的是，在只指定一个子进程时，主进程在开始调用异步 IO 后，并没有接着开始调用计算函数，而是先返回了异步读操作的结果。

于是，我将子进程数量设定为 2，结果开始发生变化。第二个线程的读取结果在主进程调用计算函数后显示。

这证明了代码确实实现了并行。但为什么子进程只有一个时，看似并没有实现异步呢？

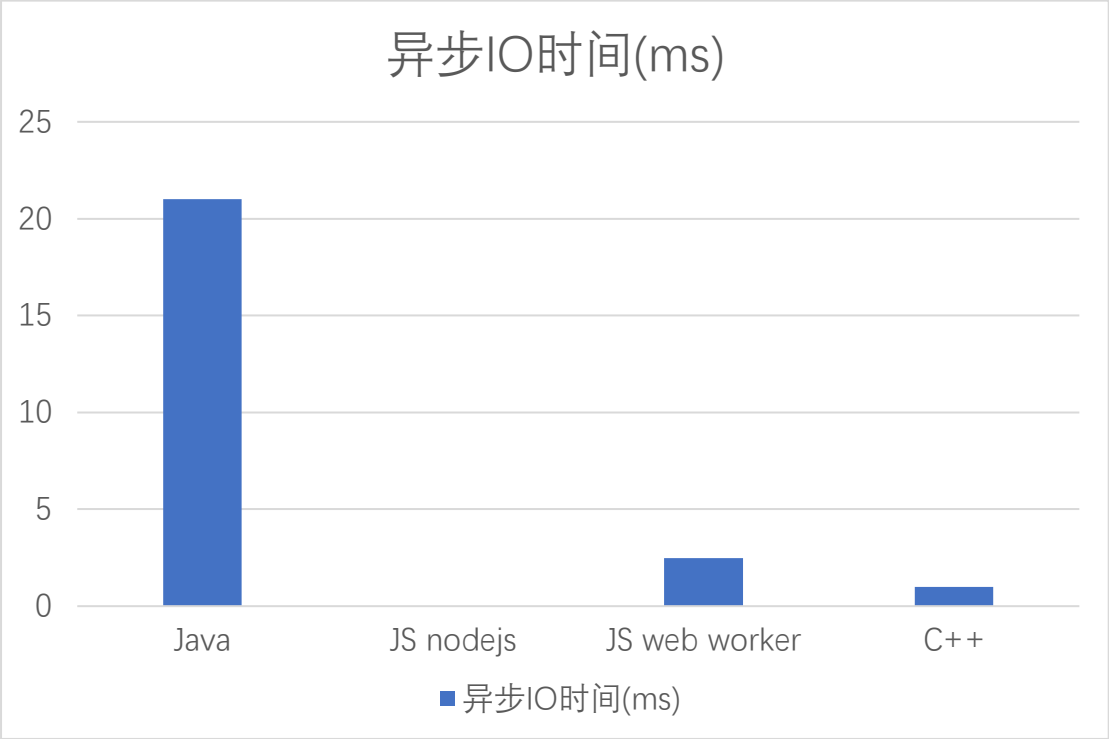
原因可能是读取文件的速度太快，只有 1ms，导致主进程还没有调用计算函数，子线程就已经返回文件信息了。（此点存疑）

2.2.4 三种语言实验结果的对照实验

一、横向对比

准备了 4631 字节的 passage.txt，对三种语言的四种方案进行计时，截图已在上面 2.2.1/2.2.2/2.2.3 的结果中列出。

语言	Java	JS nodejs	JS web worker	C++
异步 IO 时间(ms)	21	/	2.4741	1



注：由于 node.js 方案并不能准确测出异步 IO 时间，所以其数据不列在其上。

二、纵向对比

准备 5 个大小不同的文件，它们分别是：

文件名	passage.txt	passage1.txt	passage2.txt	passage3.txt	passage4.txt
大小(B)	4631	9262	13893	18524	23155

实际是把 passage.txt 的内容复制了 N+1 次，形成 passageN.txt

截图详见文件夹图片/${语言} 横向对比

Java

文件	运行时间（三次）ms	运行时间（平均值）ms
passage.txt	21、17、16	18
passage1.txt	46、30、27	34.333
passage2.txt	72、50、23	49.333
passage3.txt	68、20、27	38.333
passage4.txt	20、30、77	42.333

分析：可以看出，Java 解决方案在文件大小成倍增长时，除了两倍时运行时间增加 1 倍，其他运行时间基本持平。

JS web worker

文件	运行时间（三次）ms	运行时间（平均值）ms
passage.txt	2.4741、4.2277、5.7849	4.162
passage1.txt	2.7580、1.7929、1.7558	2.102
passage2.txt	1.6750、2.2978、2.7580	2.244
passage3.txt	2.4341、2.0649、2.6750	2.391
passage4.txt	1.7490、1.6191、1.8379	1.735

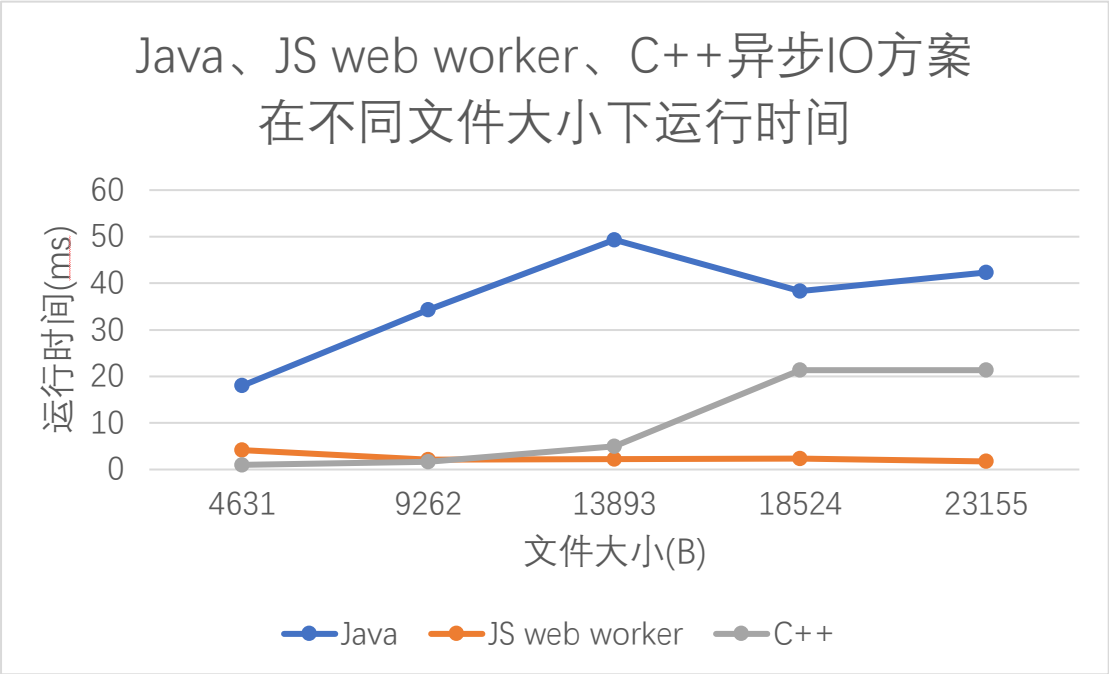
分析：JS 这组数据可能和网络延迟有关，但大体趋势是，在文件大小成倍增长时，运行时间基本持平。

C++

文件	运行时间（三次）ms	运行时间（平均值）ms
passage.txt	1、1、1	1
passage1.txt	2、2、1	1.667
passage2.txt	9、2、4	5
passage3.txt	53、7、4	21.333
passage4.txt	61、3、3	22.333

分析：在文件大小成倍增长时，运行时间呈阶梯式增长。其中较大文件的首次运行时间长，但随后两次运行时间立即缩短。这可能是文件系统的缓存引起的。

综合



结论：JS web worker 方案对于不同大小的文件运行时间最短，且增长趋势平缓。C++方案运行时间次之，且随文件大小的成倍增长，运行时间呈阶梯式增长。Java 方案运行时间最长，但在文件大小较大时增长趋势不明显。

第三章 总结

本文阐述了 Java、JavaScript 和 C++三种编程语言各自对异步 IO，尤其是异步读操作的实现方案，并用四个案例具体分析了这些方案对相同大小文件和不同大小文件的运行时间，展示了这些方案各自的优劣。

在实际工作中，尽管程序员们往往因为工作环境的关系，可能只使用 1 到 2 种编程语言，从而对于不同编程语言对异步 IO 的效率并不太关心。但本文对于四种具体方案的代码和结果的展示，既为实际生产中异步 IO 的代码提供了简单的原型，更窥一斑而知全豹地体现出各语言本身对于多线程的支持程度，不支持的情况下的独特处理以及随之带来的奇特结果，为实际生产中可能发生的问题提供了解释。

相比已有研究成果，本文只覆盖到了三种编程语言最常用的异步 IO 的解决方案。优势在于给出了具体的案例并对不同的结果进行了解释，而劣势在于案例本身只覆盖到了异步读操作，对其他 IO 操作并未涉及。

最后，本文尚且不足的一点在于，在多处只是调用了各语言对 IO 给定的 API，并用一个子线程进行执行，而未给出底层的解决方案。原因在于本文作者的知识尚且不足。通过对各语言不断地深入学习，作者希望在未来可以给出更底层、具体的实现方案。

参考文献