提供系统性能的方法就是不要让线程阻塞（空闲）

方法一：

线程告诉系统，将文件读取完成后告诉我（异步），然后系统在读取文件的时候，线程在玩手机，线程玩完手机后，系统已经把文件读取完了，然后线程拿着文件走了

方法二：

线程告诉系统，将文件读取完成后告诉我（异步），这时候线程为了不空闲下来，它必须要玩手机，但它实在不想玩手机了，于是线程选择了自杀，当系统读取完文件后，创建了一个新线程，将旧线程的所有东西都给了新线程（包括它的手机），然后新线程拿着文件走了

**c#使用异步提供Web性能原理**

如下：

static void Main(string[] args)

{

// Web 为每个请求开启一个线程

Task.Run(()=> {

// 调用请求委托

CallControllerAction();

});

Console.WriteLine("Main");

Console.ReadKey();

}

// 请求委托的目的是调用控制器，并将控制器的处理结果发送给客户端

static void CallControllerAction()

{

// 调用控制器方法

ControllerAction1()

.ContinueWith(val =>

{

// 将结果发送给客户端

Console.WriteLine(val.Result);

});

}

// 控制器方法

static Task<string> ControllerAction1()

{

return File.ReadAllTextAsync(".\\File1.txt")

.ContinueWith<string>(file => {

var text = file.Result;

return text;

});

}

请求委托的执行顺序：

1. 读取文件

File.ReadAllTextAsync(".\\File1.txt")

1. 将结果返回请求委托

.ContinueWith<string>(file => {

var text = file.Result;

return text;

});

1. 将结果返回客户端

.ContinueWith(val =>

{

// 将结果发送给客户端

Console.WriteLine(val.Result);

});

由于线程在调用File.ReadAllTextAsync(".\\File1.txt")后直接return，所有线程在步骤1的时候就已经退出，当文件读取完成后，才创建一个新线程继续执行步骤2，然后创建一个新线程执行步骤3

性能分析：

步骤1的线程在读取文件前已退出，步骤2，步骤3的线程没有阻塞，所以性能达到最大

相反，如果使用同步方法File.ReadAllText读取文件

步骤1在读取文件时阻塞，直到文件读取完成

注：

方法ControllerAction1()使用await语法糖效果如下

static async Task<string> ControllerAction()

{

var text = await File.ReadAllTextAsync(".\\File1.txt");

return text;

}

**在没有外部读写的地方，不要使用异步**

如：

static Task<string> ControllerAction()

{

return Task.Run<string>(() =>

{

Thread.Sleep(3000);

return "Hello World";

});

}

创建一个线程调用ControllerAction()方法，由于直接return，所以该线程会直接退出，但Task.Run<string>创建了新线程，该线程还在运行，性能因创建线程会略有损耗

**await关键字与Task.Wait()**

需然看似相同，但原理不同

await关键字会是当前方法立刻返回（返回Task对象），如果当前方法结束线程也结束了，那么线程不会阻塞

Task.Wait()会使线程阻塞，直到所有子线程退出