# 《C#并发编程经典实例》



1.前言

最近趁着项目的一段平稳期研读了不少书籍，其中《C#并发编程经典实例》给我的印象还是比较深刻的。当然，这可能是由于近段日子看的书大多嘴炮大于实际，如《Head First 设计模式》《Cracking the coding interview》等，所以陡然见到一本打着“实例”旗号的书籍，还是挺让我觉得耳目一新。本着分享和加深理解的目的，我特地整理了一些笔记（主要是 Web 开发中容易涉及的内容，所以部分章节如数据流，RX 等我看了看就直接跳过了），以供审阅学习。语言和技术的魅力，真是不可捉摸

2.开宗明义

一直以来都有一种观点是实现底层架构，编写驱动和引擎，或者是框架和工具开发的才是高级开发人员，做上层应用的人仅仅是“码农”，其实能够利用好平台提供的相关类库，而不是全部采用底层技术自己实现，开发出高质量，稳定的应用程序，对技术能力的考验并不低于开发底层库，如 TPL，async，await 等。

3.开发原则和要点

## （1）并发编程概述

并发：同时做多件事情
多线程：并发的一种形式，它采用多个线程来执行程序
并行处理：把正在执行的大量的任务分割成小块，分配给多个同时运行的线程
并行处理是多线程的一种，而多线程是并发的一种处理形式
异步编程：并发的一种形式，它采用future模式或者callback机制，以避免产生不必要的线程
异步编程的核心理念是异步操作：启动了的操作会在一段时间后完成。这个操作正在执行时，不会阻塞原来的线程。启动了这个操作的线程，可以继续执行其他任务。当操作完成后，会通知它的future，或者调用回调函数，以便让程序知道操作已经结束
await关键字的作用：启动一个将会被执行的Task（该Task将在新线程中运行），并立即返回，所以await所在的函数不会被阻塞。当Task完成后，继续执行await后面的代码
响应式编程：并发的一种基于声明的编程方式，程序在该模式中对事件作出反应
不要用 void 作为 async 方法的返回类型！ async 方法可以返回 void，但是这仅限于编写事件处理程序。一个普通的 async 方法如果没有返回值，要返回 Task，而不是 void
async 方法在开始时以同步方式执行。在 async 方法内部，await 关键字对它的参数执行一个异步等待。它首先检查操作是否已经完成，如果完成了，就继续运行 （同步方式）。否则，它会暂停 async 方法，并返回，留下一个未完成的 task。一段时间后， 操作完成，async
方法就恢复运行。
await代码中抛出异常后，异常会沿着Task方向前进到引用处
你一旦在代码中使用了异步，最好一直使用。调用 异步方法时，应该（在调用结束时）用 await 等待它返回的 task 对象。一定要避免使用 Task.Wait 或 Task .Result 方法，因为它们会导致死锁
线程是一个独立的运行单元，每个进程内部有多个线程，每个线程可以各自同时执行指令。 每个线程有自己独立的栈，但是与进程内的其他线程共享内存
每个.NET应用程序都维护着一个线程池，这种情况下，应用程序几乎不需要自行创建新的线程。你若要为 COM interop 程序创建 SAT 线程，就得 创建线程，这是唯一需要线程的情况
线程是低级别的抽象，线程池是稍微高级一点的抽象
并发编程用到的集合有两类：并发变成+不可变集合
大多数并发编程技术都有一个类似点：它们本质上都是函数式的。这里的函数式是作为一种基于函数组合的编程模式。函数式的一个编程原则是简洁（避免副作用），另一个是不变性（指一段数据不能被修改）
.NET 4.0 引入了并行任务库（TPL），完全支持数据并行和任务并行。但是一些资源较少的 平台（例如手机），通常不支持 TPL。TPL 是 .NET 框架自带的

# （2）异步编程基础

指数退避是一种重试策略，重试的延迟时间会逐 次增加。在访问 Web 服务时，最好的方式就是采用指数退避，它可以防止服务器被太多的重试阻塞

```
static async Task<string> DownloadStringWithRetries(string uri)
{
    using (var client = new HttpClient())
    {
        // 第 1 次重试前等 1 秒，第 2 次等 2 秒，第 3 次等 4 秒。
        var nextDelay = TimeSpan.FromSeconds(1);
        for (int i = 0; i != 3; ++i)
        {
            try
            {
                return await client.GetStringAsync(uri);
            }
            catch
            { }

            await Task.Delay(nextDelay);
            nextDelay = nextDelay + nextDelay;
        }

        // 最后重试一次，以便让调用者知道出错信息。
        return await client.GetStringAsync(uri);
    }
}
```

Task.Delay 适合用于对异步代码进行单元测试或者实现重试逻辑。要实现超时功能的话， 最好使用 CancellationToken

如何实现一个具有异步签名的同步方法。如果从异步接口或基类继承代码，但希望用同步的方法来实现它，就会出现这种情况。解决办法是可以使用 Task.FromResult 方法创建并返回一个新的 Task 对象，这个 Task 对象是已经 完成的，并有指定的值

使用 IProgress 和 Progress 类型。编写的 async 方法需要有 IProgress 参数，其 中 T 是需要报告的进度类型，可以展示操作的进度

Task.WhenALl可以等待所有任务完成，而当每个Task抛出异常时，可以选择性捕获异常

Task.WhenAny可以等待任一任务完成，使用它虽然可以完成超时任务（其中一个Task设为Task.Delay），但是显然用专门的带有取消标志的超时函数处理比较好

第一章提到async和上下文的问题：在默认情况下，一个 async 方法在被 await 调用后恢复运行时，会在原来的上下文中运行。而加上扩展方法ConfigureAwait(false)后，则会在await之后丢弃上下文