عنوان: **متدهای async تقلبی**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۱۰ ۱۸:۱۳
آدرس: www.dotnettips.info
گروه‌ها: C#, Asynchronous Programming

تا اینجا مشاهده کردیم که اگر یک چنین متد زمانبری را داشته باشیم که در آن عملیاتی طولانی انجام می‌شود،

```
class MyService
{
  public int CalculateXYZ()
  {
    // Tons of work to do in here!
    for (int i = 0; i != 10000000; ++i)
      ;
    return 42;
  }
}
```

برای نوشتن معادل async آن فقط کافی است که امضای متد را به async Task تغییر دهیم و سپس داخل آن از Task.Run استفاده کنیم:

```
class MyService
{
  public async Task<int> CalculateXYZAsync()
  {
    return await Task.Run(() =>
    {
      // Tons of work to do in here!
      for (int i = 0; i != 10000000; ++i)
        ;
      return 42;
    });
  }
}
```

و ... اگر از آن در یک کد UI استفاده کنیم، ترد آن‌را قفل نکرده و برنامه، پاسخگوی سایر درخواست‌های رسیده خواهد بود. اما ... به این روش اصطلاحا Fake Async گفته می‌شود؛ یا Async تقلبی!

کاری که در اینجا انجام شده، استفاده‌ی ناصحیح از Task.Run در حین طراحی یک متد و یک API است. عملیات انجام شده در آن واقعا غیرهمزمان نیست و در زمان انجام آن، باز هم ترد جدید اختصاص داده شده را تا پایان عملیات قفل می‌کند. اینجا است که باید بین CPU-bound operations و IO-bound operations تفاوت قائل شد. اگر Entity Framework 6 و یا کلاس WebClient و امثال آن، متدهایی Async را نیز ارائه داده‌اند، این‌ها به معنای واقعی کلمه، غیرهمزمان هستند و در آن‌ها کوچکترین CPU-bound operation ایی انجام نمی‌شود.

در حلقه‌ای که در مثال فوق در حال پردازش است و یا تمام اعمال انجام شده توسط CPU، از مرزهای سیستم عبور نمی‌کنیم. نه قرار است فایلی را ذخیره کنیم، نه با اینترنت سر و کار داشته باشیم و یا مثلا اطلاعاتی را از وب سرویسی دریافت کنیم و نه هیچگونه IO-bound operation خاصی قرار است صورت گیرد.

زمانیکه برنامه نویسی قرار است با API شما کار کند و به امضای async Task می‌رسد، فرضش بر این است که در این متد واقعا یک کار غیرهمزمان در حال انجام است. بنابراین جهت بالابردن کارآیی برنامه، این نسخه را نسبت به نمونه‌ی غیرهمزمان انتخاب می‌کند.

حال تصور کنید که استفاده کننده از این API یک برنامه‌ی دسکتاپ نیست، بلکه یک برنامه‌ی ASP.NET است. در اینجا Task.Run فراخوانی شده صرفا سبب خواهد شد عملیات مدنظر، بر روی یک ترد دیگر، نسبت به ترد اصلی اختصاص داده شده توسط ASP.NET برای فراخوانی و پردازش CalculateXYZAsync، صورت گیرد. این عملیات بهینه نیست. تمام پردازش‌های درخواست‌های ASP.NET در تردهای خاص خود انجام می‌شوند. وجود ترد دوم ایجاد شده توسط Task.Run در اینجا چه حاصلی را بجز سوئیچ بی‌جهت بین تردها و همچنین بالا بردن میزان کار Garbage collector دارد؟ در این حالت نه تنها سبب بالا بردن مقیاس پذیری سیستم نشده‌ایم، بلکه میزان کار Garbage collector و همچنین سوئیچ بین تردهای مختلف را در Thread pool برنامه به شدت افزایش داده‌ایم. همچنین یک چنین سیستمی برای تدارک تردهای بیشتر و مدیریت آن‌ها، مصرف حافظه‌ی بیشتری نیز خواهد داشت.

یک اصل مهم در طراحی کدهای Async

استفاده از Task.Run در پیاده سازی بدنه متدهای غیرهمزمان، یک code smell محسوب می‌شود.

چکار باید کرد؟

اگر در کدهای خود اعمال Async واقعی دارید که IO-bound هستند، از معادل‌های Async طراحی شده برای کار با آن‌ها، مانند متد SaveChangesAsync در EF، متد DownloadStringTaskAsync کلاس WebClient و یا متدهای جدید Async کلاس Stream برای خواندن و نوشتن اطلاعات استفاده کنید. در یک چنین حالتی ارائه متدهای async Task بسیار مفید بوده و در جهت بالابردن مقیاس پذیری سیستم بسیار مؤثر واقع خواهند شد.

اما اگر کدهای شما صرفا قرار است بر روی CPU اجرا شوند و تنها محاسباتی هستند، اجازه دهید مصرف کننده تصمیم بگیرد که آیا لازم است از Task.Run برای فراخوانی متد ارائه شده در کدهای خود استفاده کند یا خیر. اگر برنامه‌ی دسکتاپ است، این فراخوانی مفید بوده و سبب آزاد شدن ترد UI می‌شود. اگر برنامه‌ی وب است، به هیچ عنوان نیازی به Task.Run نبوده و فراخوانی متداول آن با توجه به اینکه درخواست‌های برنامه‌های ASP.NET در تردهای مجزایی اجرا می‌شوند، کفایت می‌کند.

به صورت خلاصه

از Task.Run در پیاده سازی بدنه متدهای API خود استفاده نکنید.

از Task.Run در صورت نیاز (مثلا در برنامه‌های دسکتاپ) در حین فراخوانی و استفاده از متدهای API ارائه شده استفاده نمائید:

```
private async void MyButton_Click(object sender, EventArgs e)
{
  await Task.Run(() => myService.CalculateXYZ());
}
```

در این مثال از همان نسخه‌ی غیرهمزمان متد محاسباتی استفاده شده‌است و اینبار مصرف کننده است که تصمیم گرفته در حین فراخوانی و استفاده نهایی، برای آزاد سازی ترد UI از await Task.Run استفاده کند (یا خیر).

بنابراین نوشتن یک چنین کدهایی در پیاده سازی یک API غیرهمزمان

```
await Task.Run(() =>
{
   for (int i = 0; i != 10000000; ++i)
     ;
});
```

صرفا خود را گول زدن است. کل این عملیات بر روی CPU انجام شده و هیچگاه از مرزهای IO سیستم عبور نمی‌کند.

برای مطالعه بیشتر

Should I expose asynchronous wrappers for synchronous methods