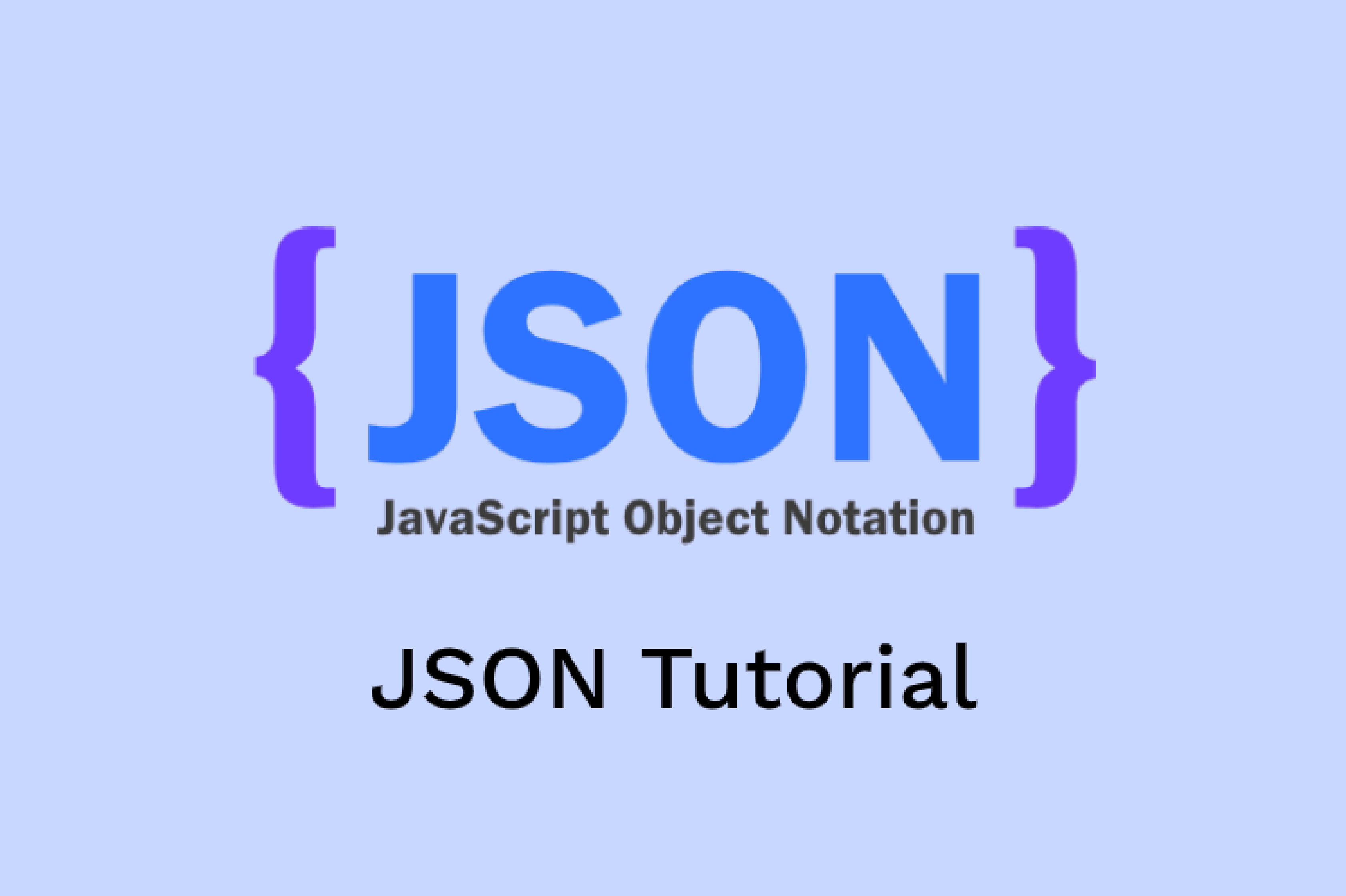
.NET 性能分析：.NET 8 中的 Newtonsoft.Json 与 System.Text.Json性能比较

作者：微信公众号：【架构师老卢】

2-6 15:42 4



概述：作为一个充满激情的 .NET 开发人员，我发现自己一直在寻找机会来扩展我对 .NET 生态系统的理解。在这篇文章中，我开始了探索.NET性能复杂性的旅程，特别关注两个著名的JSON框架：Newtonsoft.Json和Microsoft的System.Text.Json。我的目标是与像我一样热衷于破译我们技术生态系统中细微差别的开发人员分享在这次探索中获得的见解和启示。本文的灵感来自 Tobias Streng 关于 .NET 7 性能的值得注意的文章，我调整了我的调查，以符合 .NET 8 的最新进展。和我一起探索这两个 JSON 强国之间的细微差别，并更深入地了解它们在实际场景中的性能影响

作为一个充满激情的 .NET 开发人员，我发现自己一直在寻找机会来扩展我对 .NET 生态系统的理解。在这篇文章中，我开始了探索.NET性能复杂性的旅程，特别关注两个著名的JSON框架：Newtonsoft.Json和Microsoft的System.Text.Json。我的目标是与像我一样热衷于破译我们技术生态系统中细微差别的开发人员分享在这次探索中获得的见解和启示。

本文的灵感来自 Tobias Streng 关于 .NET 7 性能的值得注意的文章，我调整了我的调查，以符合 .NET 8 的最新进展。和我一起探索这两个 JSON 强国之间的细微差别，并更深入地了解它们在实际场景中的性能影响。

原文： .NET Performance #2： Newtonsoft vs. System.Text.Json by Tobias Streng

框架流行度

截至 2024 年 1 月 27 日，Newtonsoft.Json 拥有超过 42 亿次下载的骄人记录，巩固了其作为 NuGet 上下载次数最多的软件包的地位。相比之下，System.Text.Json 以大约 18 亿次的下载量落后。值得注意的是，自 .NET Core 3.1 以来，System.Text.Json 作为默认包含在 .NET SDK 中，这极大地促进了其广泛采用。

将这些数字与原始 .NET 7 文章进行比较，可以发现一个令人信服的叙述。当时，Newtonsoft.Json 的下载量已累积 23 亿次，这意味着在 15 个月的时间里，下载量增长了 82.6%。在同一时间段内，System.Text.Json 经历了 200% 的显着增长，这表明采用速度更快。然而，在检查纯粹的下载数字时，Newtonsoft.Json 在此期间增加了惊人的 19 亿次下载——超过了 System.Text.Json 自 2019 年引入 .NET SDK 以来的总下载量。

基准测试场景

为了重新创建与原始文章相同的场景，我们将重点介绍两个主要用例：

1. 单个大型数据集的序列化和反序列化。
2. 许多小型数据集的序列化和反序列化。

对于测试数据，我们将利用 NuGet 包 Bogus 生成具有自己唯一标识的随机用户。

```
[Params(10000)]
public int Count { get; set; }

private List<User> testUsers =  [ ];

[GlobalSetup]
public void GlobalSetup()
{
    var faker = new Faker<User>().CustomInstantiator(
        f =>
            new User(
            Guid.NewGuid(),
            f.Name.FirstName(),
            f.Name.LastName(),
            f.Name.FullName(),
            f.Internet.UserName(f.Name.FirstName(), f.Name.LastName()),
            f.Internet.Email(f.Name.FirstName(), f.Name.LastName())
            )
    );

    testUsers = faker.Generate(Count);
}

```

基准测试设置

- Newtonsoft.Json 13.0.3 版本
- System.Text.Json 8.0.1
- 虚假的 35.4.0
- 基准点网 0.13.12

序列化基准

序列化大数据对象

在此基准测试中，我们使用数据结构检查单个大型对象的序列化性能。这两个框架都使用默认的 .ListContractResolver

```
[Benchmark]
public void NewtonsoftSerializeBigData() =>
    _ = Newtonsoft.Json.JsonConvert.SerializeObject(testUsers);

[Benchmark]
public void MicrosoftSerializeBigData() =>
    _ = System.Text.Json.JsonSerializer.Serialize(testUsers);
```

结果：

结果反映了 .NET 7 分析中的结果，其中 System.Text.Json 的速度比 Newtonsoft.Json 快两倍以上。Microsoft的软件包还表现出卓越的内存效率，与Newtonsoft.Json相比，使用的内存不到一半。

使用自定义 json 序列化器设置序列化大数据对象

在此方案中，我们重新审视前面的序列化测试，引入一个新元素：将 JSON 属性转换为 snake 大小写。请注意：

多次实例化 ContractResolver 可能会导致性能下降，因此需要仔细考虑。

```
[Benchmark]
public void NewtonsoftSerializeBigDataWithSettings()
{
    var settings = new Newtonsoft.Json.JsonSerializerSettings()
    {
        Formatting = Newtonsoft.Json.Formatting.Indented,
        ContractResolver = new DefaultContractResolver
        {
            NamingStrategy = new SnakeCaseNamingStrategy()
        }
    };

    _ = Newtonsoft.Json.JsonConvert.SerializeObject(testUsers, settings);
}

[Benchmark]
public void MicrosoftSerializeBigDataWithSettings()
{
    var settings = new JsonSerializerOptions()
    {
        WriteIndented = true,
        PropertyNamingPolicy = new SnakeCasePropertyNamingPolicy()
    };

    _ = System.Text.Json.JsonSerializer.Serialize(testUsers, settings);
}
```

结果：

结果表明，在应用自定义命名策略时，Newtonsoft.Json 和 Microsoft 的 System.Text.Json 都会遇到性能下降和内存使用量增加的问题。但是，与System.Text.Json（1.493毫秒）相比，Newtonsoft.Json的平均执行时间（4.678毫秒）增加了更多，这表明在应用相同的命名策略时，Newtonsoft.Json的执行时间比Microsoft的包多出大约213%。

序列化许多小数据对象

此方案代表了 JSON 序列化的实际用例，密切模拟 REST API。基准测试涉及循环遍历并单独序列化每个用户。List<User>

```
[Benchmark]
public void NewtonsoftSerializeIndividualData()
{
    foreach (var user in testUsers)
    {
        _ = Newtonsoft.Json.JsonConvert.SerializeObject(user);
    }
}

[Benchmark]
public void MicrosoftSerializeIndividualData()
{
    foreach (var user in testUsers)
    {
        _ = System.Text.Json.JsonSerializer.Serialize(user);
    }
}
```

结果：

如前所述，与Newtonsoft.Json相比，Microsoft的System.Text.Json再次展示了更快的平均执行时间。此外，请务必注意两个软件包之间内存分配的显著差异。Tobias Streng 强调了节省堆内存的重要性，并考虑了堆内存对整体应用程序性能的影响。

“节省堆内存比速度更重要，你在这里看到。堆内存最终将不得不被垃圾回收，这将阻止您的整个应用程序执行”

反序列化基准

反序列化大数据对象

现在，我们将把重点转移到反序列化上，从将一个大型JSON字符串反序列化为相应的.NET对象的基准开始。List

```
[Benchmark]
public void NewtonsoftDeserializeBigData() =>
    _ = Newtonsoft.Json.JsonConvert.DeserializeObject<List<User>>(serializedTestUsers);

[Benchmark]
public void MicrosoftDeserializeBigData() =>
    _ = System.Text.Json.JsonSerializer.Deserialize<List<User>>(serializedTestUsers);
```

结果：

值得注意的是，将Newtonsoft与Microsoft的反序列化进行比较，过去一年没有实质性的变化。虽然Microsoft似乎在内存分配方面进行了小幅优化，但总体趋势表明Microsoft比Newtonsoft快得多。

反序列化许多小数据对象

最后，我们将对来自 .List<string>

```
[Benchmark]
public void NewtonsoftDeserializeIndividualData()
{
    foreach (var user in serializedTestUsersList)
    {
        _ = Newtonsoft.Json.JsonConvert.DeserializeObject<User>(user);
    }
}

[Benchmark]
public void MicrosoftDeserializeIndividualData()
{
    foreach (var user in serializedTestUsersList)
    {
        _ = System.Text.Json.JsonSerializer.Deserialize<User>(user);
    }
}
```

结果：

Microsoft再次展示了近两倍的速度，并且令人惊讶的是，反序列化所需的空间比Newtonsoft少30 MB以上。这与 .NET 7 基准测试的结果相呼应，表明 Microsoft 具有一致的性能优势。此外，Microsoft似乎进行了进一步的优化，使用的内存比去年略少。

结论

在 .NET 8 环境中的 JSON 序列化和反序列化领域，我们的基准测试提出了一个令人信服的案例。尽管Newtonsoft.Json声称其高性能，但结果明确表明，Microsoft的System.Text.Json始终优于其同类产品。无论是处理大型还是小型数据集，System.Text.Json 都具有卓越的速度和内存效率。

关键要点：

- **序列化和反序列化性能：**System.Text.Json 在序列化和反序列化任务方面始终表现出色。
- **内存效率：**SDK 原生包 System.Text.Json 不仅在速度上超过了 Newtonsoft.Json，而且在内存分配方面也表现出了显著的效率。

这些发现特定于 .NET 8，请务必认识到性能特征可能因版本而异。基于从 .NET 7 获得的见解，可以合理地断言 System.Text.Json 是 .NET 7 和 8 的所有测试方案中的更快选择。虽然关于未来版本的假设仍然不确定，但Newtonsoft.Json的创建者最近与Microsoft的一致性表明了一个有利于System.Text.Json的潜在轨迹。