ASP.NET Core性能最佳实践

本文提供ASP.NET Core性能最佳实践的引导。

攻击性缓存

本文档有多个地方讨论缓存，详细信息链接如下[Response caching in ASP.NET Core | Microsoft Learn](https://learn.microsoft.com/en-us/aspnet/core/performance/caching/response?view=aspnetcore-6.0)

掌握热代码路径

本文档中，热代码是指，经常被调用的或能够被很多次执行的代码。热代码通常限制了程序的扩展（缩放）和性能，并且该文档中有多个地方讨论到。

避免死循环调用

ASP.NET Core 应用程序应该被设计为可以同时处理多个请求。异步API应该允许一个小的线程池处理上千个同时的请求，并且不用等待被锁住的请求。而不是等待一个长时间运行的任务去完成，这个线程可以处理其它的请求。

一个常见的性能问题是ASP.NET Core 应用阻断了可以异步处理的请求。更多同步请求阻断问题可以移步 [线程池饥饿](https://learn.microsoft.com/en-us/archive/blogs/vancem/diagnosing-net-core-threadpool-starvation-with-perfview-why-my-service-is-not-saturating-all-cores-or-seems-to-stall)，并且减少响应时间。

不要做：

* 使用Task.wait 或者Task<TResult>.Result阻断异步程序的执行。
* 获取公共代码路径中的锁。ASP.NET Core 应用的架构在parallel上会有更好的性能。
* 调用Task.Run，然后立即调用await，ASP.NET Core 已经运行在普通的线程池内线程内运行该代码，于是调用Task.Run只会导致不必要的额外的线程池的调度。甚至这个调度可能会锁住该线程，Task.Run 不能阻止这样的状态发生

应该这样：

* 保证热代码是异步的
* 调用数据权限，输入/输出，或者长时间运行的API尽量使用异步的API，不要使用Task.Run将一个同步的API转变为异步
* 使Controller/Razor页面的Action异步，这样整个调用堆栈都是异步的，以便从async/await模式中收益。

一些性能分析工具，如PerfView，能够查看频繁的添加到线程池中的线程。Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Start事件标明了一个线程被添加到线程池中。

**从多个小的页面返回一个大的集合**

一个web页面不应该一次性加载大量的数据。当返回一个大集合的数据时，应该要考虑是否会引起性能问题。确定这种设计是否会引起以下问题：

* 内存溢出异常或高内存占用（消费）
* 线程池饥饿（查看下面的IAsyncEnumerable<T>章节）
* 响应时间过长
* 频繁的垃圾收集处理

正确做法是添加分页来减轻上述情况。使用页面大小和页面索引参数，开发人员能够支持返回部分结果的设计。当必须输出一个详细的结果时，分页应该能够异步填充结果集以避免阻塞服务器资源。

更多的分页和限制返回结果集数量的信息，请查看：

* [性能参考](https://learn.microsoft.com/en-us/aspnet/core/data/ef-rp/intro?view=aspnetcore-6.0#performance-considerations)
* [在ASP.NET Core程序内添加分页](https://learn.microsoft.com/en-us/aspnet/core/data/ef-rp/sort-filter-page?view=aspnetcore-6.0#add-paging)

**返回IEnumerable<T>或IAsyncEnumerable<T>**

使用序列化程序，从action的中返回的IEnumerble<T>，这样做的结果会导致调用阻塞和潜在的线程池饥饿。为了避免同步的列举（enumeration），在返回IEnumerable类型前应使用ToListAsync。

从ASP.NET Core 3.0开始，IAsyncEnumerable<T>可以替代IEnumerable<T>实现异步的列举，更多信息，请查看[控制器动作返回类型](https://learn.microsoft.com/en-us/aspnet/core/web-api/action-return-types?view=aspnetcore-6.0#return-ienumerablet-or-iasyncenumerablet)。

**尽量减少大对象的分配**

在ASP.NET Core中GC自动管理内存的回收和释放。自动的垃圾会搜通常意味着开发人员不需要考虑内存的释放，然而清除不需要的对象是需要消费CPU时间的，所以开发人员应该尽量减少对热路径下对象的分配。垃圾回收对大对象（大于85K bytes）的清理尤其耗时（贵）。大对象应该保存在[大对象堆（large object heap）](https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/large-object-heap)，然后需要一个GC（第二代）去清理。不同于第0代和第1代的垃圾回收系统，第二代垃圾回收系统需要临时暂停程序的执行。对大对象频繁的分配和取消分配操作会导致不好的性能。

建议：

* 对频繁使用的大对象考虑是否可以缓存。缓存后就不会导致对大对象进行频繁的分配动作。
* 使用ArraryPool<T>池缓冲区来保存大的数组。
* 在热代码路径内不要分配多的、生命周期很短的大对象。

内存问题，就像上面提到的这些，可以通过PerfView查看垃圾回收（GC）状态来诊断和详细检查：

* GC暂停时间
* 垃圾回收时间占处理时间的百分比
* 使用了多少GC，并且他们的是第几代的

更多信息，请参阅[垃圾回收与性能](https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/performance)

**优化数据存储和输入输出（I/O）**

在ASP.NET Core程序中，数据存储和与其他远程服务交互通常是最慢的部分。高效的数据读写对于良好的性能至关重要。

建议：

* 异步调用所有的数据读写接口
* 不要获取非必要的数据。编写查询应该只返回对本次HTTP请求必要的数据。
* 对频繁从数据库或远程服务获取的数据，如果这些数据可以接受在短时间内的过期，应该考虑对这些数据进行缓存。取决于具体场景，可以使用[MemoryCache](https://learn.microsoft.com/en-us/aspnet/core/performance/caching/memory?view=aspnetcore-6.0)和[DistributionCache](https://learn.microsoft.com/en-us/aspnet/core/performance/caching/distributed?view=aspnetcore-6.0)。更多信息，请参考[在ASP.NET Core的响应缓存](https://learn.microsoft.com/en-us/aspnet/core/performance/caching/response?view=aspnetcore-6.0)。
* 减少网络往返。尽量在一次请求中获取必要的数据，而不是多次请求。
* 在EFCore中，当查询的数据只是用来读的时候，应当使用[不追踪的查询（no-tracking queries）](https://learn.microsoft.com/en-us/ef/core/querying/tracking#no-tracking-queries)。EF Core能够更有效率的返回不追踪的查询对象。
* 过滤和聚合Linq查询（比如 where,Select,Sum），以便这些过滤由数据库来执行。
* 考虑EFCore中的有些查询是否在客户端（Client）执行，这些查询会拖慢查询性能，更多信息，请参考[客户端评估性能问题](https://learn.microsoft.com/en-us/ef/core/querying/client-eval#client-evaluation-performance-issues)。
* 不要在集合上使用投影查询，这样会导致类似“N+1”的sql查询，更多信息，请参考：[相关子查询的优化](https://learn.microsoft.com/en-us/ef/core/what-is-new/ef-core-2.1#optimization-of-correlated-subqueries)。

参考[EF高性能](https://learn.microsoft.com/en-us/ef/core/what-is-new/ef-core-2.0" \l "explicitly-compiled-queries)，了解更多在大规模应用中提升性能的方法：

* [共用DbContext](https://learn.microsoft.com/en-us/ef/core/what-is-new/ef-core-2.0#dbcontext-pooling)
* [精确编译的查询](https://learn.microsoft.com/en-us/ef/core/what-is-new/ef-core-2.0#explicitly-compiled-queries)

我们建议在提交基础代码前横梁上诉高性能方法的影响。一些复杂的查询并不适合性能的提升。

可以通过应用程序洞察力或分析工具来判断应用程序花费在读取数据上的时间来检测查询问题。大部分数据库也提供了对于频繁执行的查询的统计信息。

**使用HttpClientFactory管理Http连接池**

尽管HttpClient实现了IDisposable接口，在设计时考虑服用。关闭HttpClient实例会让该实例打开的socket端口在TIME\_WAIT状态等待一段时间。如果代码中频繁的创建和销毁HttpClient实例，则有可能会用尽所有可用的socket端口连接。HttpClientFactory在ASP.NET Core 2.1中为了解决这个问题而添加的。它能共用Http连接以优化性能和提高可靠性。

建议：

* 不要直接创建和销毁HttpClient实例
* 使用HttpClientFactory获取HttpClient实例，更多信息，请参考 使用[HttpClientFacotry实现可服用的Http请求](https://learn.microsoft.com/en-us/dotnet/standard/microservices-architecture/implement-resilient-applications/use-httpclientfactory-to-implement-resilient-http-requests)

**保持公共代码的快速的执行**

你希望你所有的代码都快速执行。频繁调用的代码是最需要优化的，这些包括：

* 请求处理管道里的中间件，尤其是处于管道前面部分的。这些组件对性能有很大的影响。
* 每次请求都会运行一次或多次的代码。比如：定制化的日志、认证处理、或初始化一些临时的服务。

建议：

* 不要使用自定义长时间运行的中间件组件。
* 使用性能检测工具，比如：Visual Studio Diagnostic Tools, PerfView 来检查常用代码性能。

**在Http请求之外完成长时间运行的任务**

大部分到ASP.NET Core程序的请求会被一个Controller或页面模型调用必要的服务执行，并返回Http响应。有一些请求可能会包含长时间运行的任务，最好是把这种请求做一个整体的请求-响应异步进程。

建议：

* 不要在普通的Http请求中等待长时间运行的任务完成
* 考虑将长时间运行的请求使用后台服务或进程外的Azure Function实现。在进程外完成CPU密集型的任务特别有益。
* 使用实时交互，如SignalR，与客户端进行异步通讯。

**缩小客户端资源**

ASP.NET Core程序使用复杂的前端技术的会频繁调用很多js，css或图片文件。提升初始化加载的性能可以考虑：

* Bunding，将多个文件合并成一个。
* Minifying（缩小），去掉文件内的空格和注释，减少文件的大小

建议：

* 参考打包和压缩引导，介绍了兼容的工具，并且展示了如何利用ASP.NET Core环境标签来处理开发和产品环境信息。
* 考虑第三方的一些工具，如Webpack，管理客户端的资源。

**压缩Http响应**

减少响应的大小会非常显著的提升程序的响应性。一个方法就是减少响应负荷大小的方法就是压缩响应，更多详细信息，请参考[压缩响应](https://learn.microsoft.com/en-us/aspnet/core/performance/response-compression?view=aspnetcore-6.0)。

最小化异常

异常应该是罕见的。与其它代码流模式相比，抛出和捕捉异常是一个相对较慢的模式。基于此，异常不应该被用来控制正常的程序逻辑。

建议：

* 不要使用抛出和捕获异常来作为保证程序正常流程的一种手段，尤其是在常用代码中。
* 在程序中应包含 对可能触发的异常的检测和处理。
* 为意外或异常情况抛出或捕获异常。

程序诊断工具，如Application Insights，可以帮助检查程序可能影响性能的常见异常。

**性能和可靠性**

以下部分提供了性能提示和已知的可靠性问题及解决方案

**在HttpRequest和HttpRespose body中避免同步读写**

在ASP.NET Core 所有的I/O操作都是异步的。服务实现了Stream接口，该接口有同步和异步方法的的重载。使用异步方法可以避免阻塞线程池线程。线程的阻塞可能会导致线程池饥饿。

不要这样做：下面的示例使用了ReadToEnd犯法，阻塞了当前线程等待结果，这是一个同步异步的例子：

public class BadStreamReadController:Controller

{

[HttpGet(“/contoso”)]

public ActionResult<ContosoData> Get()

{

var json=new StreamReader(Request.Body).ReadToEnd();

return JsonSerializer.Deserialize<ContosoData>(json);

}

}