

异步编程示例

*此示例与 Microsoft 游戏开发工具包(2020 年 6 月)兼容*

# 说明

此示例展示了如何使用 **XAsync**、**XTaskQueue** 和 **XAsyncProvider** 以各种方式实现异步编程和任务处理。虽然需要将 XAsync 与 GDK 异步函数结合使用，但库本身功能强大，且在 Windows 10、Xbox One 和 Xbox Series X|S 之间完全可移植。

示例 UI 提供了用于运行示例测试并显示输出的界面。请参阅以下**实现说明**，从而深入了解每个测试的说明以及如何遵循这些用例的代码。

# 生成示例

如果使用 Xbox One 开发工具包，请将活动解决方案平台设置为 Gaming.Xbox.XboxOne.x64。

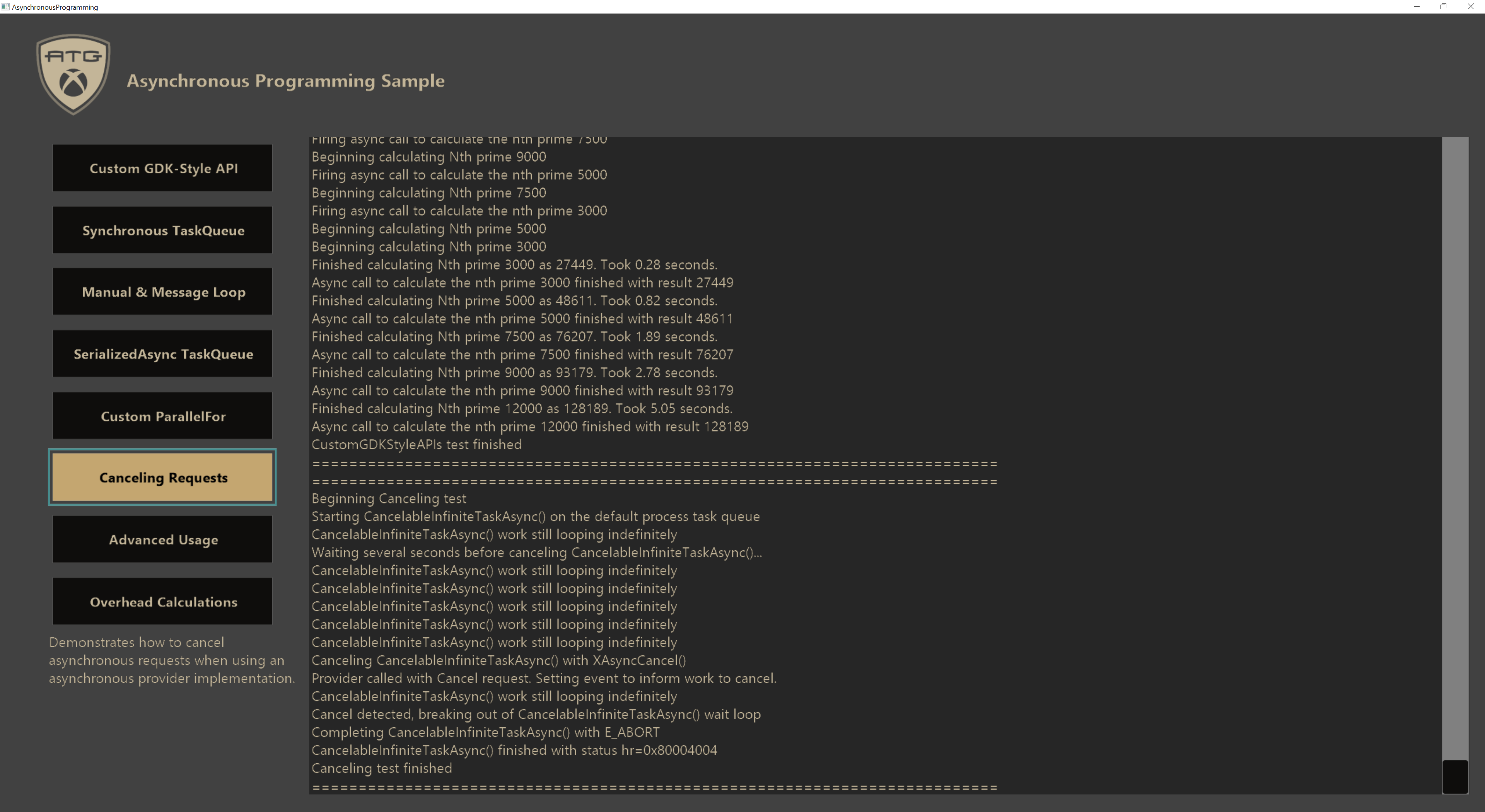
如果使用 Xbox Series X|S 开发工具包，请将活动解决方案平台设置为 Gaming.Xbox.Scarlett.x64。

如果使用 Windows 10，请将活动解决方案平台设置为 Gaming.Desktop.x64。

*有关详细信息，请参阅 GDK 文档中的“*运行示例”*。*

# 使用示例

只需使用主屏幕上的按钮即可启动不同的测试场景，从而实现 **XAsync**/**XTaskQueue**/**XAsyncProvider** 库的不同用法。



# 实现说明

示例提供了 8 种不同的测试场景，用于涵盖 **XAsync**、 **XTaskQueue** 和 **XAsyncProvider** 的不同功能。

*有关这些库的详细信息，请参阅 GDK 文档。简单来说，****XAsync*** *提供了运行并管理异步任务的方法，* ***XTaskQueue*** *提供了管理异步任务的执行上下文/行为的方法，****XAsyncProvider*** *提供了对整体异步任务进程管道的高级管理。*

所有用法和测试都在类 **XAsyncExamples** 中实现，除了集成到 Windows 消息循环中以进行其中一个测试的情况。每个测试都位于名为 "**StartTest\_[TestName]**" 的函数中，因此这是查看实现的良好起点。

任务队列在 **XAsyncExamples::CreateTaskQueues()** 中创建，线程在 **XAsyncExamples::CreateThreads()**; 中生成。任务队列的关闭逻辑依赖于其端口模式，因此请参阅 **XAsyncExamples::ShutdownTaskQueues()** 以了解实现的详细信息。当任务队列具有手动端口时，必须继续调度该端口，直到它返回，这是因为 **XTaskQueueTerminate()** 会将事件添加到每个端口。

**RunTask()** 是一种常见的帮助程序方法，可实现标准的 **XAsyncBlock** 创建和设置，然后使用 **XAsyncRun()** 启动异步调用。它只需指定任务队列、工作回调和(可选)完成回调，大大简化了任务的执行。

下面迭代了每个测试的详细信息:

**自定义 GDK 样式 API 测试**

此测试演示了如何以 GDK API 函数的样式设置异步方法。调用和结果进程与异步 GDK API 方法的工作方式相同。

任务队列:

* m\_taskQueue\_CustomGDKStyleAPIs
  + 工作端口:ThreadPool
  + 完成端口:手动

**NthPrimeAsync()** 和 **NthPrimeAsyncResult()** 是以 GDK 异步样式实现的两种方法。这两个函数都需要设置 **XAsyncBlock** 并在启动异步任务进程时传递它。为管道使用 **NthPrimeAsync()** 实现异步提供程序，操作结果从 **NthPrimeAsyncResult()** 返回。

测试函数会设置对 **NthPrimeAsync()** 的 5 次异步调用，并在完成回调中报告结果。由于使用的任务队列具有 ThreadPool 工作端口，因此这 5 个请求在系统线程池上并行运行。对于“手动”完成端口，通过调用 **XAsyncExamples::Update()** 中的 **XTaskQueueDispatch()** 来调度回调。完成端口上的回调按其排队顺序执行。

**同步 TaskQueue 测试**

此测试演示了如何仅基于 **XAsyncBlock** 结构中的指定任务队列同步调用异步任务。此外，此测试还展示了如何更改进程默认任务队列。

任务队列:

* m\_taskQueue\_SynchronousTaskQueue
  + 工作端口:即时
  + 完成端口:即时
* 进程默认任务队列
  + 工作端口:ThreadPool
  + 完成端口:ThreadPool

测试函数首先将进程默认任务队列替换为自定义任务队列，其中两个端口都设置为“即时”。即时端口会导致在尝试将回调排入队列时立即执行回调，而不是将回调排入队列以供稍后调度。测试还使用名为 **ParallelExecute()** 的帮助程序函数，该函数在指定的任务队列上启动指定次数的回调，然后等待所有这些任务完成。

整个测试首先使用系统默认任务队列的缓存句柄，通过 **RunTask()** 移动到系统线程池。这样做的目的是允许测试在后台运行，而不会冻结应用程序。然后，20 个任务与使用 ThreadPool 端口的缓存进程默认任务队列句柄并行执行。最后，相同的 20 个任务以展示的相同方式执行，将任务队列切换到端口设置为“即时”的测试队列。你将在输出中注意到，由于并行化，异步工作会快速完成，但同步版本必须一次执行一个任务。

**手动与消息循环测试**

此测试演示了如何使用手动任务队列以及如何将该任务队列与 Windows 消息循环集成以进行完成回调。多个线程用于并行调度“手动”工作端口，任务队列监视器回调用于向 Windows 消息循环发送通知以处理“手动”完成回调。

任务队列:

* m\_taskQueue\_ManualAndMessageLoop
  + 工作端口:手动
  + 完成端口:手动

此测试(代码形式)有几个位置让人感兴趣，这些位置不仅仅位于 **StartTest\_ManualAndMessageLoop() 中**。注意，请参阅 **WndProc()**, **Notify\_TaskQueueMonitor()**, **ThreadProc\_ManualAndMessageLoop()**, **Monitor\_ManualAndMessageLoop()**, **CreateTaskQueues() 和** **ShutdownTaskQueues()**。

在 **CreateTaskQueues()** 中创建此测试的任务队列时，请注意，也会为任务队列创建监视器。每当向任务队列上的端口新增回调时，就会执行此监视器。每当执行该监视器时，每当检测到完成端口的回调时，它都将使用具有 *WM\_TASKQUEUEMONITOR* 的 **PostMessage()**。**WndProc()** 设置为检查 *WM\_TASKQUEUEMONITOR* 事件以及对 **Notify\_TaskQueueMonitor()** 的调用，后者在 Windows 消息循环的上下文中从完成端口调度回调。

最后，测试函数是一款简单的驱动程序，它会在测试任务队列上启动 20 个异步请求，其中指定了工作和完成回调。

**SerializedAsync TaskQueue 测试**

此测试演示了如何使用 SerializedThreadPool 端口设置并使用任务队列，以及这些端口如何工作。

任务队列:

* m\_taskQueue\_SerializedAsync
  + 工作端口:SerializedThreadPool
  + 完成端口:SerializedThreadPool

设置为 SerializedThreadPool 的端口会自动调度系统线程池上的回调，但一次只调度一个。回调按其排队顺序调度。此行为与使用“手动”端口并一次只在一个线程上调度一个回调相同。但是，系统线程池不需要每次都使用相同的线程。

由于 SerializedThreadPool 端口一次只运行一个回调，因此可以在异步工作的不同回调之间具有依赖关系。测试函数在任务队列上将 5 个不同的任务排入队列以利用这一点，其中每个任务都使用上一个任务的结果，并会更改下一个任务的数据。

**自定义 ParallelFor 测试**

此测试演示了一种可实现 ParallelFor 函数的方法。手动任务队列用于执行工作，执行了前缀和计算以测试性能。

任务队列:

* m\_taskQueue\_ParallelFor
  + 工作端口:手动
  + 完成端口:即时

为了尝试获取最大并行性能，为每个逻辑处理器生成了线程，并将每个线程初始化为具有处理器关联。**ThreadProc\_SingleCore()** 可以同时调用 **XTaskQueueDispatch()** 以实现并行化，且每个线程都将获得不同的任务。

前缀总和以两种方式进行计算，用于测试同步和异步行为。 为了提供最佳比较，为同步用例实现了紧环，并将不同的并行优化算法用于异步用例。日志中报告了用例计时。

**取消请求测试**

此测试演示了如何取消异步任务。由于取消行为必须在异步任务的提供程序内手动执行，因此并非所有异步任务都可以取消。因此，无法取消通过 **XAsyncRun()** (由 **RunTask()** 使用)运行的简单任务。为了演示可取消的任务，使用了自定义的异步提供程序函数(很像 GDK 样式测试)并实现了取消路径。

任务队列:

* 默认进程任务队列
  + 工作端口:ThreadPool
  + 完成端口:ThreadPool

请参阅 **CancelableInfiniteTaskAsync()**，从而了解有关使用自定义提供程序实现异步方法的信息。此异步方法不会返回数据，但会无限运行，除非被取消。实现了 **XAsyncOp::Cancel** 用例以设置无限运行时可以检测到的事件。设置该事件后，提供程序就会知道需使用 *E\_ABORT* 完成 **XAsyncComplete()**。

测试用例函数会启动对 **CancelableInfiniteTaskAsync()** 的调用，并在一段时间后启动另一个任务以调用 **XAsyncCancel()**。请观看输出日志以查看运行中的测试。

**高级使用测试**

此测试演示了一些不常见的高级功能，包括复合任务队列、复制的任务队列句柄以及使用等待程序和延迟事件调度。在此测试中，在测试函数中创建并销毁了所有任务队列和其他测试数据，从而展示整个过程。

该测试首先运行了复合队列测试。复合队列是一个任务队列，它的端口由来自其他任务队列的端口组成。首先创建了常规任务队列，其中 ThreadPool 作为工作端口，“手动”设置为完成端口。然后，创建了复合任务队列，该队列将第一个任务队列的工作端口用作其自己的两个端口。这样就可以使工作和完成回调在原始队列的同一端口上运行。

接下来，该测试演示了如何复制队列句柄。每当复制任务队列句柄时，必须多次将其关闭才可完成清理。重复句柄的运行方式与原始句柄相同，只是增加了对资源的所有权。

最后，测试了等待程序和延迟调度。等待程序是多次提交回调以执行的方法，但仅在每次触发特定事件时才进行提交。这有助于自动执行事件。延迟调度是将回调直接提交到任务队列(具有可选延迟)的其中一个端口。常规异步提交需要 **XAsyncBlock**，但 **XTaskQueueSubmitCallback**/**XTaskQueueSubmitDelayedCallback** 改为将回调直接提交到任务队列。它们通常由异步提供程序在内部使用。

**开销计算测试**

此测试计算了利用 **XAsync**/**XTaskQueue**/**XAsyncProvider** 时各种开销的计时。了解利用异步方法的不同开销会很有用。

这些开销针对三种不同的场景计算，其中每个场景都有多个用例。请参阅以下图表，从而了解有关每个计时的说明。

|  |  |
| --- | --- |
| 计时名称 | 说明 |
| XAsyncRun\_InvokeToWork (进程默认任务队列) | 时间: 调用 **XAsyncRun()** 与和在进程默认任务队列上启动工作回调之间所用的时间。 |
| XAsyncRun\_InvokeToWork (手动任务队列) | 时间: 调用 **XAsyncRun()** 与工作回调开始将手动队列与已在等待的线程结合使用之间所用的时间。 |
| XAsyncRun\_WorkToCompletion (进程默认任务队列) | 时间: 从 **XAsyncRun()** 退出工作回调计划与使用进程默认任务队列启动完成回调之间所用的时间。 |
| XAsyncRun\_WorkToCompletion (手动任务队列) | 时间: 从 **XAsyncRun()** 退出工作回调计划与完成回调开始将手动队列与已在等待的线程结合使用之间所用的时间。 |
| ParallelFor\_InvokeToWork | 时间: 此示例中实现的自定义 **ParallelFor()** 方法从调用到启动其中一个工作回调所用的时间。 |
| ParallelFor\_InvokeToReturn | 时间: 在此示例中实现的自定义 **ParallelFor()** 方法完成方法的整个调用(无需大量工作)所用的时间。 |
| GDKAsyncStyle\_TimeInProviderAverage (进程默认任务队列) | 时间: 使用进程默认任务队列时自定义提供程序方法所用的平均时间。 |
| GDKAsyncStyle\_TimeInProviderAverage (手动任务队列) | 时间: 将手动队列与已在等待的线程结合使用时，自定义提供程序方法所用的平均时间。 |
| GDKAsyncStyle\_TimeInProviderOverall (进程默认任务队列) | 时间: 使用进程默认任务队列时，自定义提供程序方法所用的总时间。 |
| GDKAsyncStyle\_TimeInProviderOverall (手动任务队列) | 时间: 将手动队列与已在等待的线程结合使用时，自定义提供程序方法所用的总时间。 |
| GDKAsyncStyle\_InvokeToWork (进程默认任务队列) | 时间: 使用具有系统默认任务队列的自定义提供程序时，从调用到启动工作回调所用的时间。 |
| GDKAsyncStyle\_InvokeToWork (手动任务队列) | 时间: 使用具有已在等待线程的手动队列的自定义提供程序时，从调用到启动工作回调所用的时间。 |
| GDKAsyncStyle\_WorkToCompletion (进程默认任务队列) | 时间: 使用进程默认任务队列时，自定义提供程序方法的工作回调和完成回调之间所用的时间。 |
| GDKAsyncStyle\_WorkToCompletion (手动任务队列) | 时间: 将手动任务队列与已在等待的线程结合使用时，自定义提供程序方法的工作回调和完成回调之间所用的时间。 |

每次运行时都会重新计算计时，且会将计时输出到日志中。

# 有关 SMT 和时间敏感线程的说明

**SMT (同时多线程)**

Xbox Series X|S 设备支持 SMT，从而允许逻辑核心计数翻倍。但是，功耗增加会导致启用 SMT 时核心频率下降。这意味着游戏可以选择具有 7 个快速核心或 14 个慢速核心(需考虑一些共享注意事项)。

有关详细信息，请参阅标题为“*Xbox One 与Project Scarlett CPU 和内存”*节中的文档。

要在此示例中启用 SMT，请打开 MicrosoftGame.config 文件，并在其中取消注释 <VirtualMachine> 节。

为了突出显示 CPU 行为的差异，以下是在开、关 SMT 的情况下测试几个不同测试时的一些计时:

|  |  |  |
| --- | --- | --- |
|  | ParallelFor PrefixSum 异步时间 | 5000th 次主要计算同步/并行 |
| SMT 开 | 范围从 100ms 到 125ms | 一致的 0.47s 同步，3.88~9.20s 并行，总计 ~9.3s |
| SMT 关 | 范围从 140ms 到 175ms | 一致的 0.44s 同步，0.44~1.14s 并行，总计 ~4.5s |

观察:

* 打开 SMT 后，旨在并行化的算法的性能更好，但仅用密集型计算填充每个核心可能会损害性能。
* 关闭 SMT 后，并行算法在核心较少的情况下不会获得尽可能多的性能优势，但单线程类型的工作速度更快，且核心更能抵御性能损失，且密集型工作会得到充分利用。

最终由开发人员确定 SMT 是否适合游戏。

**时间敏感线程**

并非每个 GDK API 函数都能够安全地调用应在不阻止的情况下快速运行的线程。这些线程称为“时间敏感”线程。为了支持确保 GDK API 函数不会阻止这些线程，添加了时间敏感 API 以允许标记时间敏感线程。标记后，调用可能缓慢的 GDK API 函数将触发断言，以便开发人员了解相关情况。这些断言可以通过恢复执行继续向前。

有关详细信息，请参阅标题为*“时间敏感线程”*节中的文档。

要在此示例中进行测试，请取消注释对 **Sample::Initialize()** 中**XThreadSetTimeSensitive()** 的调用。这将为此线程启用时间敏感检查。启用后，将在两个不同的测试中触发这些断言，如下所述:

* 同步任务队列测试
  + **XTaskQueueSetCurrentProcessTaskQueue()**
    - 此函数用于演示如何更改进程的默认任务队列。通常，应在加载时执行此操作以避免潜在减速，在此测试中执行此操作是为了展示功能。
* 高级使用测试
  + **XTaskQueueCreate()**
    - 在示例中，大多数任务队列在初始化早期创建，然后将线程标记为时间敏感。对于此测试，任务队列直接在测试中创建。此函数非时间敏感安全函数，因为它可能需要一小段时间才可创建。
  + **XTaskQueueGetPort()**
    - 获取任务队列的端口非快速操作，实际上被认为只在初始化过程中执行，例如调用 **XTaskQueueCreate()** 时。
  + **XTaskQueueCreateComposite()**
    - 这与使用所有相同的注意事项调用 **XTaskQueueCreate()** 时相同。唯一不同的是新任务队列正在使用的端口。
  + **XTaskQueueRegisterWaiter()**
    - 注册等待程序是另一项缓慢操作，应为加载时间保留或完成时间敏感线程以避免减速或搭便车。

由于任务队列通常会停留很长时间，因此通常应保留上述触发时间敏感断言的函数以在加载时进行调用。执行此操作时，线程可以在加载序列期间通过对 **XThreadSetTimeSensitive(false)** 的另一个调用暂时删除其时间敏感状态。如果需要在游戏期间或其他活动时间执行此操作，则可以将调用移动到非时间敏感线程，从而避免对应用程序性能产生明显影响。

# 更新历史记录

**初始版本:**Microsoft 游戏开发工具包(2020 年 5 月)

2020 年 6 月:修复了导致测试无法完成的 lambda 捕获 bug。修复了渲染大小和丢失设备的已知问题并删除了相关节。向示例添加了自述文件和时间敏感线程调用的 SMT 和时间敏感线程信息。

# 隐私声明

在编译和运行示例时，将向 Microsoft 发送示例可执行文件的文件名以帮助跟踪示例使用情况。若要选择退出此数据收集，你可以删除 Main.cpp 中标记为“示例使用遥测”的代码块。

有关 Microsoft 的一般隐私策略的详细信息，请参阅 [Microsoft 隐私声明](https://privacy.microsoft.com/en-us/privacystatement/)。