为了确保用户与应用的交互流畅，应用应在每个设备确定的刷新率下渲染每一帧——例如，Pixel 6 允许最大每秒渲染 90 帧，这意味着每帧需要在 11 毫秒内渲染。如果应用遇到 UI 渲染缓慢的问题，Android 框架将被迫跳帧。也称为卡顿。

本文重点介绍由应用程序引起的卡顿，以及 Android Studio 提供的工具来发现和修复这些卡顿，通过检查实时应用程序并打开记录的跟踪来解决应用程序中的性能问题。

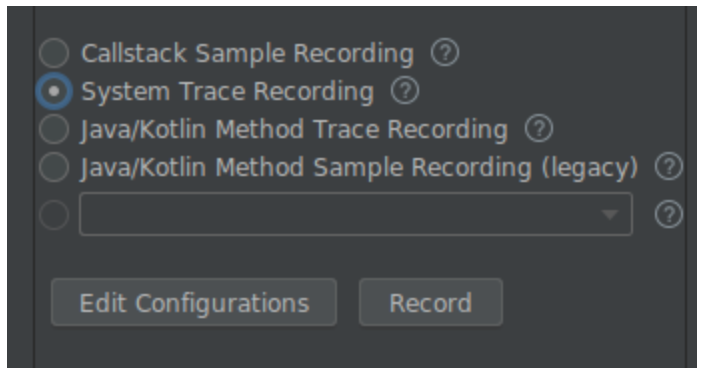
记录实时交互的跟踪：

以下示例使用 GitHub 性能示例存储库中的 JankStatsSample 应用程序来展示如何使用 CPU 分析器发现卡顿的原因。

1. 打开 JankStatsSample 并运行应用程序
2. 打开 Android Studio 底部的 Profiler 选项卡
3. 通过单击 Profiler 左侧窗格中的 + 图标启动新的分析会话，然后选择要运行分析器的设备名称和应用程序进程。

注意：虽然可以分析可调试应用程序，但建议分析可分析应用程序，因为分析可调试应用程序会增加显着的性能成本作为副作用。

1. 单击 CPU 行
2. 选择系统跟踪记录并单击记录



1. 与应用程序交互以收集一些数据，然后单击停止按钮。

然后 Android Studio 将显示显示部分，其中的 janky 帧可见

或者通过选中 All Frames 复选框，您可以切换跟踪是否也显示非 janky 帧。



可以通过将鼠标指针悬停在每个帧上或单击它来查看详细的帧信息。如果您选中所有帧复选框，则会出现三种类型的帧。

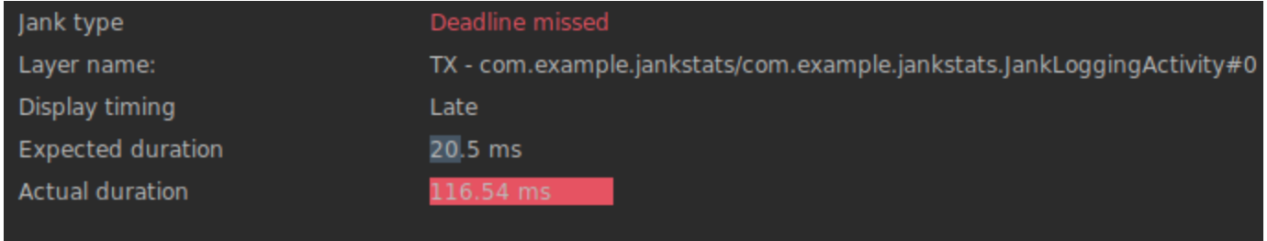
绿框：

不被视为卡顿的正常帧



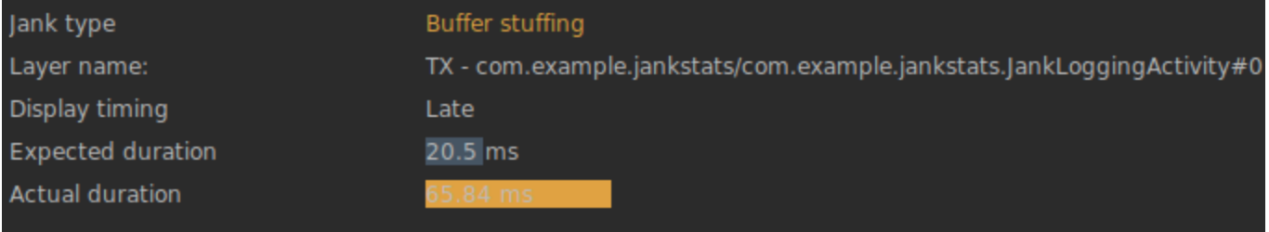
红框

由于应用程序进程运行时间超过预期并且错过了帧预期持续时间的最后期限，因此被视为卡顿的帧。从应用程序开发人员的角度来看，可操作框架通常是红色框架。

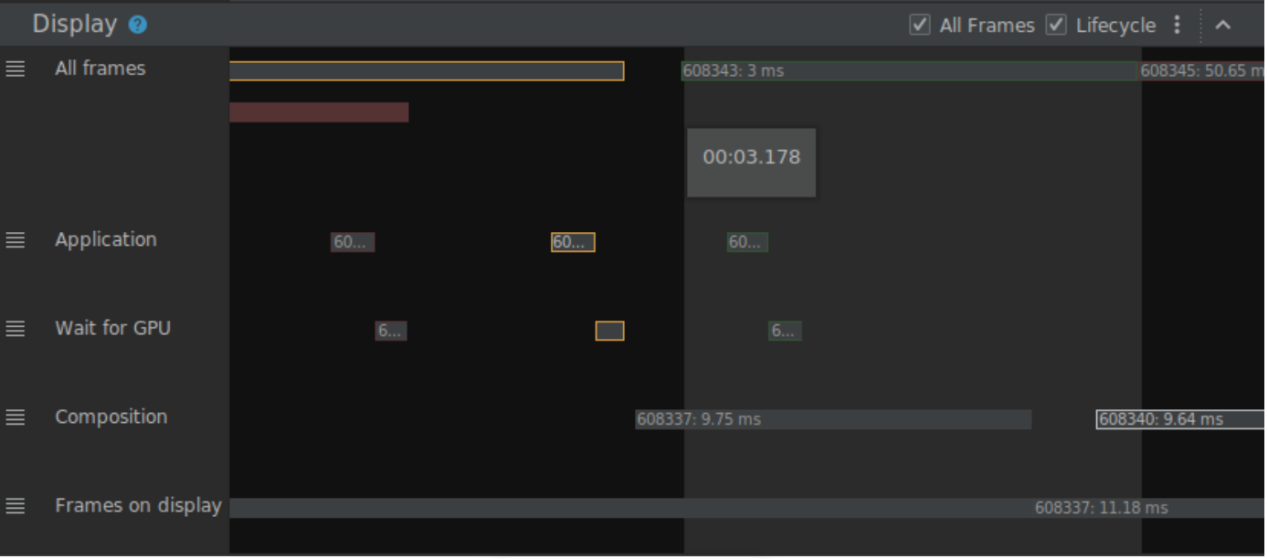


黄框

被视为缓冲区填充的帧，这意味着应用程序会在呈现前一帧之前不断将帧发送到系统。这通常是由于前一帧卡顿造成的，并且应用程序开发人员对黄色帧无能为力。



通过选中 Lifecycle 复选框，您可以切换显示/隐藏四个附加轨道。



这四个轨道：

Application

Wait for GPU

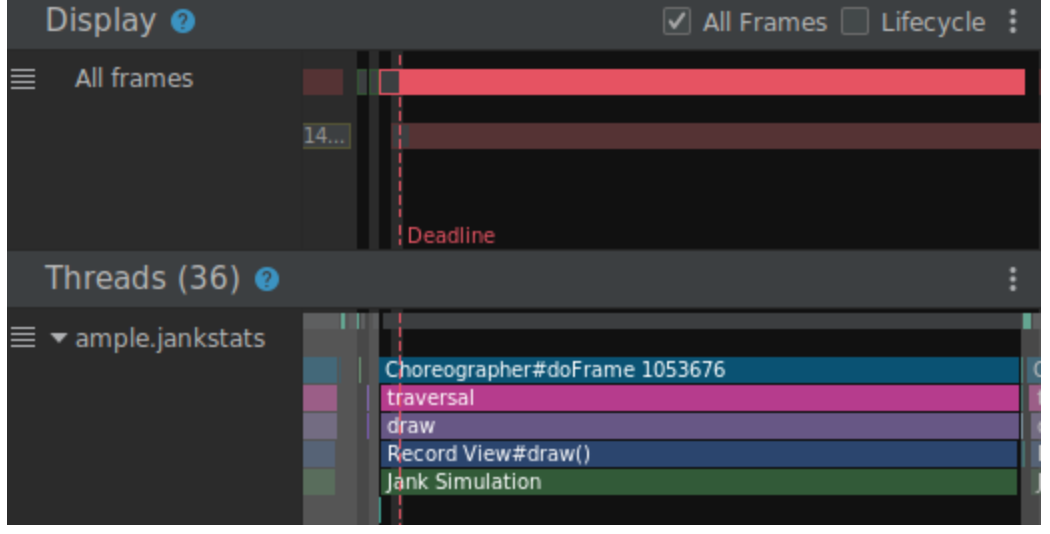
Composition

Frames on display

检查 janky 帧

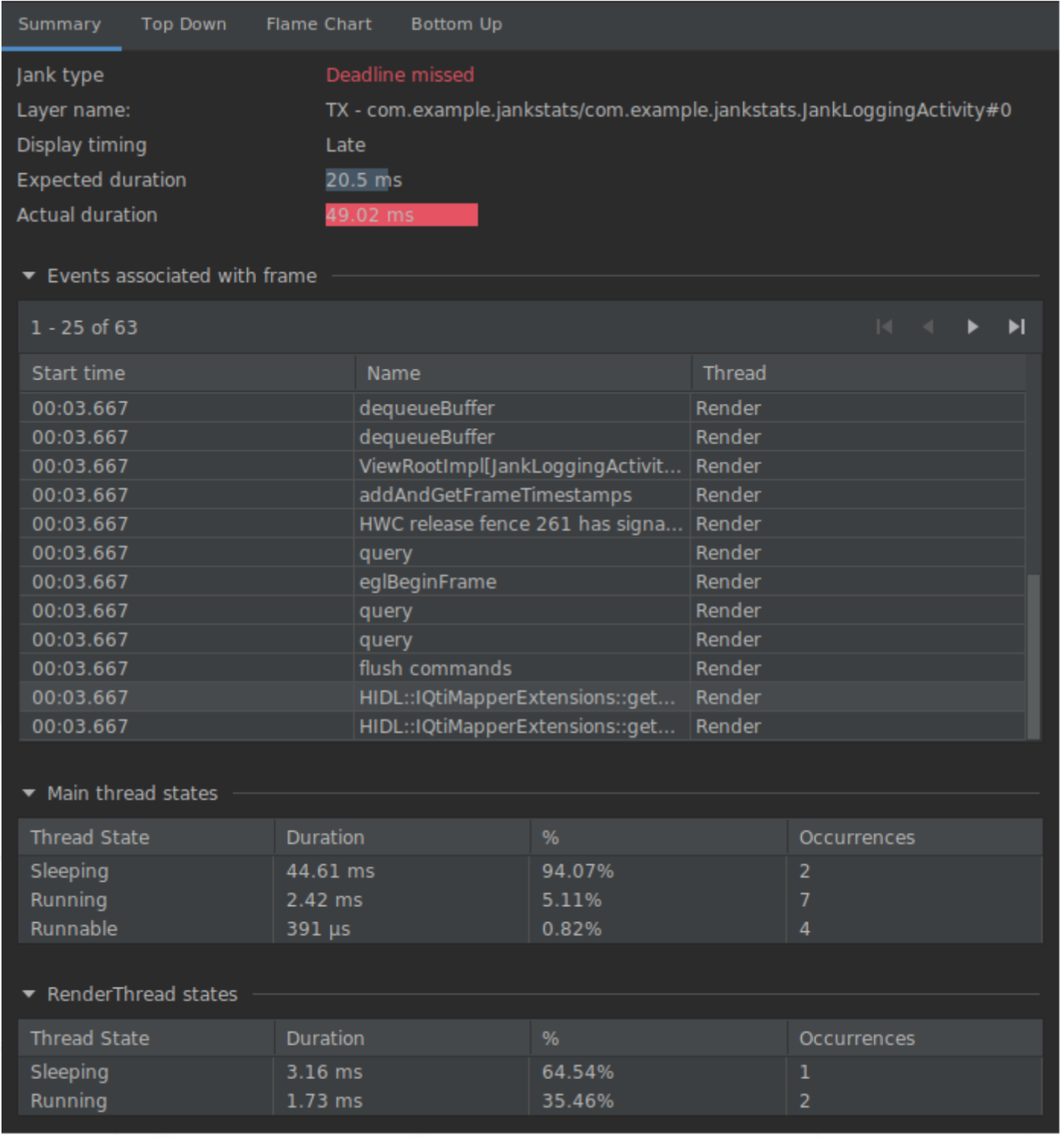
现在，让我们看看如何诊断导致应用中出现 janky 帧的原因。

在 Janky frames track 中选择一个 janky frame，然后在 Display 部分突出显示相应的生命周期数据，并在 Threads 部分突出显示相应的线程数据。

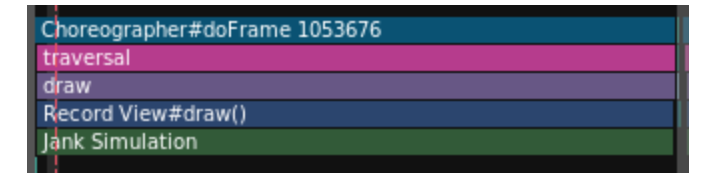


虚线的Deadline代表截止日期。当帧的持续时间超过该期限时，该帧被视为 janky。

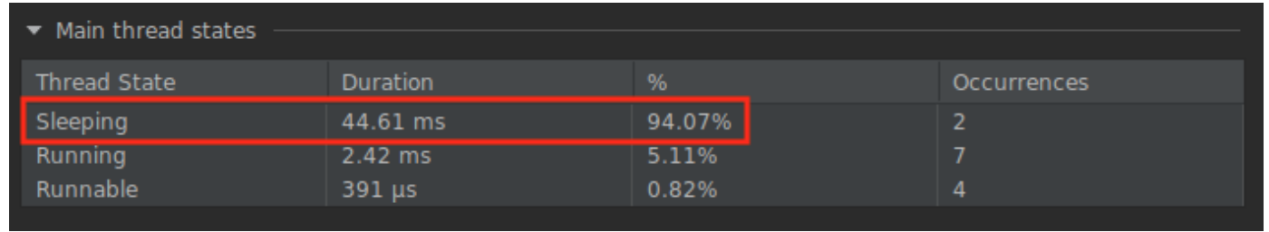
还可以在 Android Studio 的右侧窗格中查看详细分析。



如果查看应用程序主线程中的相应跟踪部分，您可以看到大量时间都花在了 `View#draw` 中。



此外，如果在详细分析窗格中查看主线程状态，可以看到大量线程状态正在休眠。



让我们看一下调用 View#draw 的代码。

可以发现 View#onDraw 在 JankyView 类中被覆盖。

override fun onDraw(canvas: Canvas) {

simulateJank()

super.onDraw(canvas)

}

从onDraw调用的simulateJank方法定义如下，

fun simulateJank(

jankProbability: Double = 0.3,

extremeJankProbability: Double = 0.02

) {

val probability = nextFloat()

if (probability > 1 - jankProbability) {

val delay = if (probability > 1 - extremeJankProbability) {

nextLong(500, 700)

} else {

nextLong(32, 82)

}

try {

// Make jank easier to spot in the profiler through tracing.

trace("Jank Simulation") {

Thread.sleep(delay)

}

} catch (e: Exception) {

}

}

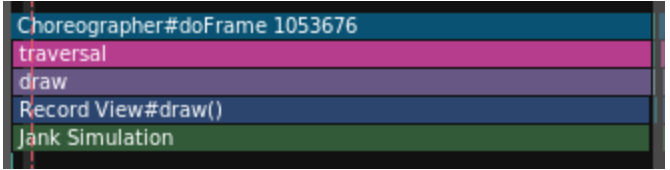
}

然后你可以发现 Thread.sleep 在simulateJank 方法中被调用。这可能很明显，因为创建 JankStatsSample 应用程序是为了故意模拟 jank，但重要的是，您可以通过从 janky 帧的概述到更详细的分析来发现实际代码。

注意：虽然在这个例子中调用 Thread.sleep 很明显是有问题的，但实际上你在优化你的真实应用程序代码时需要做出更困难的决定。 Microbenchmark 库可以帮助您衡量您所做的更改是否具有预期的效果。

注意：系统跟踪显示了由平台代码和应用程​​序库捕获的各个部分。通常它没有足够的信息。为了改善这一点，添加自定义跟踪，添加自定义跟踪的方法之一是使用 trace(“MySection”) { /\* 这将在跟踪 \*/ } 中的 AndroidX 跟踪库。

例如，此示例中的 trace(“Jank Simulation”) { ... } 显示在相应线程的跟踪部分中。



让我们将代码更改为不调用 onDraw 中的模拟 Jank 方法，然后查看是否仍然观察到 janky 帧。

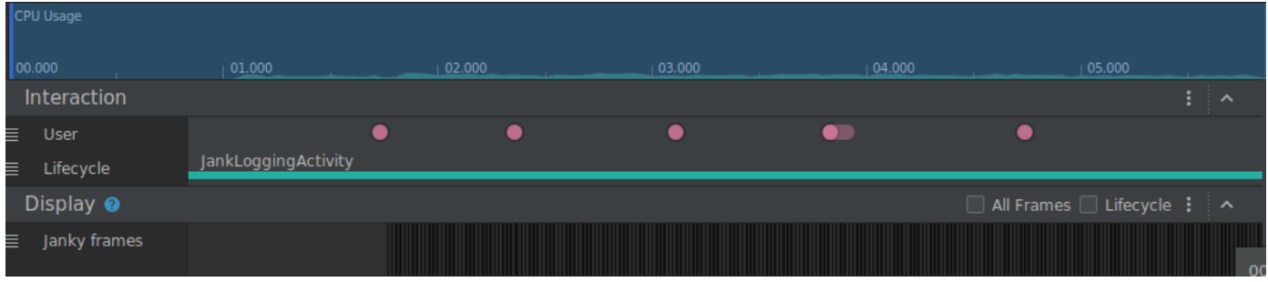
override fun onDraw(canvas: Canvas) {

// simulateJank()

super.onDraw(canvas)

}

这一次，当您重新运行系统跟踪记录时，您将在与应用程序交互后看到没有卡顿！



加载保存的轨迹

或者，您也可以通过以下步骤保存跟踪并稍后加载。保存和加载跟踪很有用，因为您可以比较不同版本的跟踪，与其他人共享。

注意：您也可以通过使用 Macrobenchmark 库来获取系统跟踪。

按照与记录实时交互的跟踪相同的步骤 1 到 6。

通过单击保存图标导出record

JankStats 库可帮助您跟踪和分析应用的性能问题。卡顿是指呈现应用帧用时过长的情况，JankStats 库提供关于应用卡顿统计信息的报告。

JankStats 基于现有的 Android 平台功能构建，包括 Android 7（API 级别 24）及更高版本中的 FrameMetrics API，该 API 用于跟踪帧需要多长时间才能完成。JanksStats 库提供了两项额外的功能，使其更加动态且更易于使用，即：卡顿启发法和界面状态。

卡顿启发法

虽然您可以使用 FrameMetrics 跟踪帧时长，但 FrameMetrics 无法帮助确定实际的卡顿情况。但是，JankStats 具有可配置的内部机制，可确定发生卡顿的时间，从而使报告更加即时实用。

界面状态

了解应用中出现性能问题的情境通常很必要。例如，如果您开发了一个使用 FrameMetrics 的复杂跨屏应用，并且发现您的应用经常出现存在严重卡顿的帧，那么您就会想了解出现问题的位置、当时用户在执行的操作以及如何进行重现，以便在具体情境下分析相关信息。

JankStats 通过引入一个 state API 解决了这个问题。借助该 API，您可以与该库进行通信，提供应用 activity 的相关信息。JankStats 记录有关卡顿帧的信息时，会将应用的当前状态添加到卡顿报告中。

用法

如需开始使用 JankStats，请针对每个 Window 实例化并启用该库。每个 JankStats 对象只跟踪一个 Window 中的数据。将该库实例化需要一个 Window 实例以及一个执行器和一个 OnFrameListener 监听器，这两者均用于向执行器线程上的客户端发送指标。系统会通过 FrameData 对每一帧调用监听器；监听器会提供以下详细信息：

帧开始时间

时长值

帧是否应被视为卡顿

一组字符串对，其中包含呈现帧期间应用状态的相关信息

为了让 JankStats 更加实用，应用应使用相关的界面状态信息填充该库，以便在 FrameData 中报告。您可以通过 PerformanceMetricsState API（而非直接通过 JankStats）实现此操作，其中包含所有状态管理逻辑和 API。

初始化

如需开始使用 JankStats 库，请先将 JankStats 依赖项添加到您的 Gradle 文件中：

implementation "androidx.metrics:metrics-performance:1.0.0-alpha01"

接下来，针对每个 Window 初始化并启用 JankStats。当 activity 进入后台时，您也应该暂停 JankStats 跟踪。在您的 activity 替换中创建并启用 JankStats 对象：

class JankLoggingActivity : AppCompatActivity() {

private lateinit var jankStats: JankStats

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

// ...

// metrics state holder can be retrieved regardless of JankStats initialization

val metricsStateHolder = PerformanceMetricsState.getForHierarchy(binding.root)

// initialize JankStats for current window

jankStats = JankStats.createAndTrack(

window,

Dispatchers.Default.asExecutor(),

jankFrameListener,

)

// add activity name as state

metricsStateHolder.state?.addState("Activity", javaClass.simpleName)

// ...

}

上面的示例会在构建 JankStats 对象后注入有关当前 activity 的状态信息。现在，日后为此 JankStats 对象创建的所有 FrameData 报告也会包含 activity 信息。

JankStats.createAndTrack 方法接受对 Window 对象的引用，该对象是该 Window 内的视图层次结构的代理，也是 Window 本身的代理。在启用 JankStats 对象的情况下，每个帧的 Executor 确定的线程上都会调用 jankFrameListener。

注意：窗口必须处于活跃状态并具有非 null DecorView，否则会返回 IllegalStateException。

如要对任何 JankStats 对象启用跟踪和报告功能，请调用 isTrackingEnabled = true。虽然跟踪功能默认处于启用状态，但暂停某项 activity 会停用该功能。在这种情况下，请务必先重新启用跟踪功能，然后再继续操作。如需停止跟踪，请调用 isTrackingEnabled = false。

override fun onResume() {

super.onResume()

jankStats.isTrackingEnabled = true

}

override fun onPause() {

super.onPause()

jankStats.isTrackingEnabled = false

}

报告

JankStats 库会将每一帧的所有跟踪数据报告给已启用的 JankStats 对象的 OnFrameListener。应用可以存储和聚合这些数据，以便日后上传。

如需创建 jankFrameListener 对象以使用 OnFrameListener 初始化 JankStats，请执行以下操作：

private val jankFrameListener = JankStats.OnFrameListener { frameData ->

// A real app could do something more interesting, like writing the info to local storage and later on report it.

Log.v("JankStatsSample", frameData.toString())

}

您需要为应用创建并提供 OnFrameListener，以接收每帧报告；

fun interface OnFrameListener {

fun onFrame(frameData: FrameData)

}

监听器中使用的 FrameData 结构如下：

class FrameData(

/\*\*

\* The time at which this frame began (in nanoseconds)

\*/

val frameStartNanos: Long,

/\*\*

\* The duration of this frame (in nanoseconds)

\*/

val frameDurationNanos: Long,

/\*\*

\* Whether this frame was determined to be janky, meaning that its

\* duration exceeds the duration determined by the system to indicate jank (@see

\* [JankStats.jankHeuristicMultiplier])

\*/

val isJank: Boolean,

/\*\*

\* The UI/app state during this frame. This is the information set by the app, or by

\* other library code, that can be used later, during analysis, to determine what

\* UI state was current when jank occurred.

\*

\* @see PerformanceMetricsState.addState

\*/

val states: List<StateInfo>

)

在 FrameMetrics 机制中，Android 12（API 级别 31）及更高版本会提供更多关于帧时长的数据。JankStats 会相应地提供关于这些版本的更多信息：

class FrameDataApi24 : FrameData {

/\*\*

\* The time spent in the non-GPU portions of this frame (in nanoseconds).

\*

\* This includes the time spent on the UI thread [frameDurationUiNanos] plus time

\* spent on the RenderThread.

\*/

val frameDurationCpuNanos: Long

}

class FrameDataApi31 : FrameDataApi24 {

/\*\*

\* The amount of time past the frame deadline that this frame took to complete.

\*

\* A positive value indicates some jank, a negative value indicates that the frame was

\* complete within the given deadline

\*/

val frameOverrunNanos: Long,

}

监听器中的 StateInfo 如下所示：

class StateInfo(

val stateName: String,

val state: String

)

聚合

您可能希望应用代码聚合每帧数据，从而让您可以自行决定保存和上传信息。虽然 Alpha 版 JankStats API 无法提供关于保存和上传操作的详细信息，但您可以利用 GitHub 代码库中提供的 JankAggregatorActivity 查看将每帧数据聚合到大型集合的初步 activity。

JankAggregatorActivity 使用 JankStatsAggregator 类在 JankStats OnFrameListener 机制之上叠加自己的报告机制，以便在仅报告多个帧的相关信息集合时提供更高级别的抽象。

JankAggregatorActivity 不会直接创建 JankStats 对象，而是创建一个 JankStatsAggregator 对象，该对象会在内部创建自己的 JankStats 对象：

class JankAggregatorActivity : AppCompatActivity() {

private lateinit var jankStatsAggregator: JankStatsAggregator

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

// ...

// Metrics state holder can be retrieved regardless of JankStats initialization.

val metricsStateHolder = PerformanceMetricsState.getForHierarchy(binding.root)

// Initialize JankStats with an aggregator for the current window.

jankStatsAggregator = JankStatsAggregator(

window,

Dispatchers.Default.asExecutor(),

jankReportListener

)

// Add the Activity name as state.

metricsStateHolder.state?.addState("Activity", javaClass.simpleName)

}

JankAggregatorActivity 中使用了一种类似的机制来暂停和恢复跟踪，还添加了 pause() 事件作为信号，以通过调用 issueJankReport() 来发出报告；这是由于生命周期变更似乎是捕获应用卡顿状态的合适时机：

override fun onResume() {

super.onResume()

jankStatsAggregator.jankStats.isTrackingEnabled = true

}

override fun onPause() {

super.onPause()

// Before disabling tracking, issue the report with (optionally) specified reason.

jankStatsAggregator.issueJankReport("Activity paused")

jankStatsAggregator.jankStats.isTrackingEnabled = false

}

上面的示例代码展示了应用启用 JankStats 并接收帧数据所需的一切。

管理状态

您可能需要调用其他 API 来自定义 JankStats，例如，注入应用状态信息，使帧数据更有用。

此静态方法可检索给定视图层次结构的当前 [PerformanceMetricsState] 对象。

PerformanceMetricsState.getForHierarchy(view: View): MetricsStateHolder

活跃层次结构中的任何视图都可以使用。在内部，该操作将检查是否有现有的 MetricsStateHolder 对象与该视图层次结构相关联。此信息会缓存到该层次结构顶部的视图中。如果不存在此类对象，getForHierarchy() 会创建一个。

借助静态 getForHierarchy() 方法，您无需将实例缓存到某处以备日后检索，并且还可以更轻松地从代码（甚至库代码；库代码不会以其他方式访问原始实例）中的任何位置检索现有的状态对象。

请注意，返回值是一个容器对象，而不是状态对象本身。容器内的状态对象的值仅由 JankStats 设置。也就是说，如果应用创建了 JankStats 对象，则相应状态对象随后便会创建并设置。否则，在 JankStats 不跟踪信息的情况下，就不需要状态对象，并且应用或库代码也无需注入状态。

采用这种方法可检索 JankStats 可以随后填充的容器。外部代码可以随时请求容器。然后，调用方可以缓存此轻量级对象，并随时使用它来设置状态，具体取决于其内部状态属性的值，如以下示例代码所示，其中状态仅在容器的内部状态属性为非 null 时设置：

val metricsStateHolder = PerformanceMetricsState.getForHierarchy(binding.root)

// ...

metricsStateHolder.state?.addState("Activity", javaClass.simpleName)

如需控制界面/应用状态，应用可以使用 addState 和 removeState 方法注入（或移除）状态。JankStats 会记录任何添加或移除操作的时间戳。如果有帧与状态的开始时间和结束时间重叠，JankStats 会报告该信息以及该帧的时间数据。

对于任何状态，请添加两条信息：stateName（一种状态类别，例如“RecyclerView”）和 state（当时情况的相关信息，例如“滚动”）。

使用 removeState() 方法移除不再有效的状态，以确保报告帧数据时不会报告错误或误导性信息。

addSingleFrameState() 版状态 API 添加了一个状态，该状态在下一个报告的帧中只记录一次。之后，系统会自动将其移除，以确保您不会意外地在代码中看到已过时的状态。请注意，没有等同于 removeState() 的 singleFrame，因为 JankStats 会自动在下一帧中移除这些状态。

private val scrollListener = object : RecyclerView.OnScrollListener() {

override fun onScrollStateChanged(recyclerView: RecyclerView, newState: Int) {

// check if JankStats is initialized and skip adding state if not

val metricsState = metricsStateHolder?.state ?: return

when (newState) {

RecyclerView.SCROLL\_STATE\_DRAGGING -> {

metricsState.addState("RecyclerView", "Dragging")

}

RecyclerView.SCROLL\_STATE\_SETTLING -> {

metricsState.addState("RecyclerView", "Settling")

}

else -> {

metricsState.removeState("RecyclerView")

}

}

}

}

卡顿启发法

如需调整用于确定何为卡顿的内部算法，请使用 jankHeuristicMultiplier 属性。

默认情况下，系统会将卡顿定义为呈现帧的用时为当前刷新率两倍的情况。系统不会将超过刷新率的任何情况都视为卡顿，因为应用呈现时间的相关信息不完全明确。因此，建议您添加缓冲区，并仅在导致明显性能问题时报告问题。

这两个值都可以通过上述方法更改，以便更贴合应用的情况；也可以在测试中更改，以便根据测试需要强制发生或不发生卡顿。