Mejores prácticas de rendimiento

Generalmente, las aplicaciones Flutter son eficientes por defecto, por lo que sólo necesita evitar los errores comunes para obtener un rendimiento excelente en lugar de tener que microoptimizar con herramientas de perfilado complicadas. Estas mejores recomendaciones te ayudarán a escribir la aplicación Flutter con el mayor rendimiento posible.

Mejores prácticas

¿Cómo diseñas una aplicación Flutter para renderizar tus escenas de la forma más eficiente? En particular, ¿cómo te aseguras de que el código de pintado generado por el framework es lo más eficiente posible? He aquí algunas cosas a considerar al diseñar tu aplicación:

Controla el coste del método build()

- Evita el trabajo repetitivo y costoso en los métodos `build()` ya que `build()` puede ser invocado frecuentemente cuando los Widgets ancestros se reconstruyen.
- Evita los Widgets únicos con una función `build()` larga. Divídelos en diferentes Widgets basados en la encapsulación pero también en como cambian:
    - Cuando `setState()` es llamado en un State, todo los widgets descendientes se reconstruirán. Por lo tanto, coloca la llamada a `setState()` en la parte del subárbol cuya UI realmente necesita cambiar. Evita llamar setState() arriba en el árbol si el cambio está contenido en una parte pequeña del árbol.
    - El recorrido para reconstruir todos los descendientes se detiene cuando se vuelve a encontrar la misma instancia del hijo que en el frame anterior. Esta técnica es muy utilizada en el framework para optimizar animaciones cuando la animación no afecta al subárbol de hijos. Mira el patrón TransitionBuilder y el SlideTransition los cuales utilizan estos principios para evitar reconstruir sus descendientes cuando se anima.

Ver también:

- Consideraciones de rendimiento, parte del documento de la API StatefulWidget

Aplicar los efectos sólo cuando sea necesario

Utiliza los efectos con cuidado, ya que pueden ser caros. Algunos de ellos invocan `saveLayer()` entre bastidores, lo que puede ser una operación costosa.

Nota: ¿Por qué es costoso saveLayer?
¿Por qué es costoso saveLayer? Llamando a `saveLayer()` se asigna un buffer offscreen. La incorporación de contenido en el búfer offscreen puede desencadenar cambios de destino que son especialmente lentos en las GPU más antiguas.
Algunas reglas generales al aplicar efectos específicos:

- Usa el widget Opacity sólo cuando sea necesario. Consulta Transparent image en la página de la API Opacity para ver un ejemplo de aplicación de opacidad directamente a una imagen, lo cual es más rápido que usar el widget de opacidad.
- Clipping no llama a `saveLayer()` (a menos que se solicite explícitamente con `Clip.antiAliasWithSaveLayer`), por lo que estas operaciones no son tan costosas como la Opacidad, pero clipping sigue siendo costoso, por lo que debe usarse con precaución. De forma predeterminada, el clipping está desactivado (`Clip.none`), por lo que deberás habilitarlo explícitamente cuando sea necesario.

Otros widgets que pueden activar `saveLayer()` y son potencialmente costosos:

- ShaderMask
- ColorFilter
- Chip—puede causar llamada a `saveLayer()` si `disabledColorAlpha != 0xff`
- Text—puede causar llamada a `saveLayer()` si hay un `overflowShader`

Formas de evitar llamadas a `saveLayer()`:

- Para implementar el desvanecimiento en una imagen, considera la posibilidad de utilizar el widget FadeInImage, que aplica una opacidad gradual utilizando el sombreador de fragmentos de la GPU. Para más información, véase Opacity.
- Para crear un rectángulo con esquinas redondeadas, en lugar de aplicar un clipping al rectángulo, considera usar la propiedad borderRadius que ofrecen muchas de las clases de widgets.

Renderizar grids y lists lentamente

Utiliza los métodos lazy, con callbacks, cuando construyas grillas o listas grandes. De esta forma, sólo se crea la parte visible de la pantalla en el momento del inicio.

Ver también:

- Trabajar con listas largas en el Cookbook
- Creación de un ListView que carga una página a la vez por AbdulRahman AlHamali
- Listview.builder API

Construir y mostrar frames en 16ms

Puesto que hay dos threads separados para la compilación y el renderizado, ustedes tienen 16ms para compilar y 16ms para renderizar en una pantalla de 60Hz. Si la latencia es un problema, construye y muestra un frame en 16ms *o menos*. Ten en cuenta que significa compilado en 8ms o menos, y renderizados en 8ms o menos, para un total de 16ms o menos. Si la falta de frames (jankyness) es una preocupación, entonces 16ms para cada una de las etapas de compilación y renderizado está bien.

Si tus frames se están renderizando en un total de menos de 16ms en una construcción de perfil, es probable que no tengas que preocuparte por el rendimiento incluso si se aplican algunas dificultades de rendimiento, pero aún así deberías intentar construir y renderizar un frame lo más rápido posible. Por qué?

- Reducir el tiempo de renderizado del frame por debajo de 16ms puede que no suponga una diferencia visual, pero mejorará la duración de la batería y los problemas térmicos.
- Puede funcionar bien en tu dispositivo, pero ten en cuenta el rendimiento para el dispositivo más bajo que estés buscando.
- Cuando los dispositivos de 120 fps estén ampliamente disponibles, querrás renderizar los frames en menos de 8ms (en total) para proporcionar la experiencia más suave.

Si te preguntas por qué 60fps conduce a una experiencia visual fluida, consulta el vídeo ¿Por qué 60fps?

Obstáculos

Si necesitas ajustar el rendimiento de tu aplicación, o quizás la interfaz de usuario no es tan fluida como esperas, el plugin Flutter para tu IDE puede ayudarte. En la ventana Rendimiento de Flutter, activa la casilla Mostrar información de reconstrucción de widgets. Esta función te ayuda a detectar cuándo se renderizan

los frames y se muestran en más de 16 ms. Siempre que sea posible, el plugin proporciona un enlace a un consejo relevante.

Los siguientes comportamientos podrían afectar negativamente el rendimiento de tu aplicación.

- Evita usar el widget `Opacity`, y en particular evita utilizarlo en una animación. En su lugar, utiliza `AnimatedOpacity` o `FadeInImage`. Para obtener más información, consulta Consideraciones de rendimiento para la animación de la opacidad.
- Cuando utilices un AnimatedBuilder, evita poner un subárbol en la función builder que construye widgets que no dependan de la animación. Este subárbol se reconstruye para cada tick de la animación. En su lugar, construye esa parte del subárbol una vez y pásala como un child al AnimatedBuilder. Para obtener más información, consulta Performance optimizations.
- Evita el clipping en una animación. Si es posible realiza un pre-clip a la imagen antes de animarla.
- Evita usar constructores con una lista concreta de children (como `Column()` o `ListView()`) si la mayoría de los children no son visibles en la pantalla, y así evitar el costo de la construcción.

Recursos

Para obtener más información sobre el rendimiento, consulta los siguientes recursos:

- Optimizaciones de rendimiento en la página de la API AnimatedBuilder
- Consideraciones de rendimiento para la animación de opacidad en la pagina de la API Opacity
- Ciclo de vida de los elementos y cómo cargarlos eficientemente, en la página de la API ListView
- Consideraciones de rendimiento de un StatefulWidget

Referencia:
https://esflutter.dev/docs/testing/best-practices

● **Android Studio**: Entorno de desarrollo integrado (IDE) oficial para el desarrollo de apps para Android y está basado en IntelliJ IDEA. Además del potente editor de códigos y las herramientas para desarrolladores de IntelliJ, Android Studio ofrece incluso más funciones que aumentan tu productividad cuando desarrollas apps para Android, como las siguientes:

- Un sistema de compilación flexible basado en Gradle
- Un emulador rápido y cargado de funciones
- Un entorno unificado donde puedes desarrollar para todos los dispositivos Android
- Aplicación de cambios para insertar cambios de código y recursos a la app en ejecución sin reiniciarla
- Integración con GitHub y plantillas de código para ayudar a compilar funciones de apps comunes y también importar código de muestra
- Variedad de marcos de trabajo y herramientas de prueba
- Herramientas de Lint para identificar problemas de rendimiento, usabilidad y compatibilidad de versiones, entre otros
- Compatibilidad con C++ y NDK
- Compatibilidad integrada con Google Cloud Platform, que facilita la integración con Google Cloud Messaging y App Engine

● **Android API 30 platform**: Paquete de la Plataforma de SDK de Android, que se necesita a fin de compilar la app para esa versión.

● **Flutter**: Framework de Dart para crear aplicaciones multiplataforma con un único código. A diferencia de otros frameworks multiplataforma como por ejemplo Ionic, el código de una aplicación de Flutter se compila a código nativo, por lo que el rendimiento alcanzado es superior a aplicaciones basadas en web-views.

● **Dart** :Lenguaje de programación orientado a objetos y creado por Google. A diferencia de muchos lenguajes, Dart se diseñó con el objetivo de hacer el proceso de desarrollo lo más cómodo y rápido posible para los desarrolladores. Por eso, viene con un conjunto bastante extenso de herramientas integrado, como su propio gestor de paquetes, varios compiladores/transpiladores, un analizador y formateador. Además, la máquina virtual de Dart y la compilación *Just-in-Time* hacen que los cambios realizados en el código se puedan ejecutar inmediatamente.