Mejores prácticas de rendimiento

Contenidos

Generalmente, las aplicaciones Flutter son eficientes por defecto, por lo que sólo necesita evitar los errores comunes para obten 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 d que el código de pintado generado por el framework es lo más eficiente posible? He aquí algunas cosas a considerar al diseñar t 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 lo 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 e 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 que en el frame anterior. Esta técnica es muy utilizada en el framework para optimizar animaciones cuando la animac no afecta al subárbol de hijos. Mira el patrón TransitionBuilder y el SlideTransition los cuales utilizan estos principios 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 s 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 explícitamer 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 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 propied 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 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 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 **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 tot 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 Flutte 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 la animación. Este subárbol se reconstruye para cada tick de la animación. En su lugar, construye esa parte del subárbol un 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 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