Performance best practices

Contents

Generally, Flutter applications are performant by default, so you only need to avoid common pitfalls to get excellent performance instead of needing to micro-optimize with complicated profiling tools. These best recommendations will help you write the most performant Flutter app possible.

Best practices

How do you design a Flutter app to most efficiently render your scenes? In particular, how do you ensure that the painting code generated by the framework is as efficient as possible? Here are a few things to consider when designing your app:

Controlling build() cost

- Avoid repetitive and costly work in `build()` methods since `build()` can be invoked frequently when ancestor Widgets rebui
- Avoid overly large single Widgets with a large `build()` function. Split them into different Widgets based on encapsulation b also on how they change:
  - When `setState()` is called on a State, all descendent widgets will rebuild. Therefore, localize the `setState()` call to th part of the subtree whose UI actually needs to change. Avoid calling setState() high up in the tree if the change is contained to a small part of the tree.
  - The traversal to rebuild all descendents stops when the same instance of the child widget as the previous frame is re-encountered. This technique is heavily used inside the framework for optimizing animations where the animation doe affect the child subtree. See the `TransitionBuilder` pattern and the source code for `SlideTransition`, which uses th principle to avoid rebuilding its descendents when animating.

Also see:

- Performance considerations, part of the `StatefulWidget` API doc

Apply effects only when needed

Use effects carefully, as they can be expensive. Some of them invoke `saveLayer()` behind the scenes, which can be an expensive operation.

Why is savelayer expensive?

Calling `saveLayer()` allocates an offscreen buffer. Drawing content into the offscreen buffer might trigger render target switches that are particularly slow in older GPUs.

Some general rules when applying specific effects:

- Use the `Opacity` widget only when necessary. See the Transparent image section in the Opacity API page for an example of applying opacity directly to an image, which is faster than using the Opacity widget.
- **Clipping** doesn't call `saveLayer()` (unless explicitly requested with `Clip.antiAliasWithSaveLayer`) so these operations ar as expensive as Opacity, but clipping is still costly, so use with caution. By default, clipping is disabled (`Clip.none`), so you must explicitly enable it when needed.

Other widgets that might trigger `saveLayer()` and are potentially costly:

- `ShaderMask`
- `ColorFilter`
- `Chip`—might cause call to `saveLayer()` if `disabledColorAlpha != 0xff`
- `Text`—might cause call to `saveLayer()` if there's an `overflowShader`

Ways to avoid calls to `saveLayer()`:

- To implement fading in an image, consider using the FadeInImage widget, which applies a gradual opacity using the GPU's fragment shader. For more information, see the `Opacity` docs.

- To create a rectangle with rounded corners, instead of applying a clipping rectangle, consider using the `borderRadius` prop offered by many of the widget classes.

Render grids and lists lazily

Use the lazy methods, with callbacks, when building large grids or lists. That way only the visible portion of the screen is built at startup time.

Also see:

- Working with long lists in the Cookbook
- Creating a ListView that loads one page at a time a community article by AbdulRahman AlHamali
- `Listview.builder` API

Build and display frames in 16ms

Since there are two separate threads for building and rendering, you have 16ms for building, and 16ms for rendering on a 60Hz display. If latency is a concern, build and display a frame in 16ms *or less*. Note that means built in 8ms or less, and rendered in 8r or less, for a total of 16ms or less. If missing frames (jankyness) is a concern, then 16ms for each of the build and render stages OK.

If your frames are rendering in well under 16ms total in profile mode, you likely don't have to worry about performance even if son performance pitfalls apply, but you should still aim to build and render a frame as fast as possible. Why?

- Lowering the frame render time below 16ms might not make a visual difference, but it **improves battery life** and thermal iss
- It might run fine on your device, but consider performance for the lowest device you are targeting.
- When 120fps devices become widely available, you'll want to render frames in under 8ms (total) in order to provide the smoothest experience.

If you are wondering why 60fps leads to a smooth visual experience, see the video Why 60fps?

Pitfalls

If you need to tune your app's performance, or perhaps the UI isn't as smooth as you expect, the Flutter plugin for your IDE can he In the Flutter Performance window, enable the **Show widget rebuild information** check box. This feature helps you detect when frames are being rendered and displayed in more than 16ms. Where possible, the plugin provides a link to a relevant tip.

The following behaviors might negatively impact your app's performance.

- Avoid using the `Opacity` widget, and particularly avoid it in an animation. Use `AnimatedOpacity` or `FadeInImage` instead. Fo more information, see Performance considerations for opacity animation.
- When using an AnimatedBuilder, avoid putting a subtree in the builder function that builds widgets that don't depend on the animation. This subtree is rebuilt for every tick of the animation. Instead, build that part of the subtree once and pass it as a child to the AnimatedBuilder. For more information, see Performance optimizations.
- Avoid clipping in an animation. If possible, pre-clip the image before animating it.
- Avoid using constructors with a concrete List of children (such as `Column()` or `ListView()`) if most of the children are not visible on screen to avoid the build cost.

Resources

For more performance info, see the following resources:

- Performance optimizations in the AnimatedBuilder API page
- Performance considerations for opacity animation in the Opacity API page
- Child elements' lifecycle and how to load them efficiently, in the ListView API page
- Performance considerations of a `StatefulWidget`