Get started

Samples & tutorials

Development

- User interface
- Data & backend
- Accessibility & internationalization
- Platform integration
- Packages & plugins
- Add Flutter to existing app
- Tools & techniques
  - Android Studio & IntelliJ
  - Visual Studio Code
  - DevTools
    - Overview
    - Install from Android Studio & IntelliJ
    - Install from VS Code
    - Install from command line
    - Flutter inspector
    - Timeline view
    - Memory view
    - Performance view
    - Debugger
    - Logging view
  - Flutter SDK
  - Hot reload
  - Code formatting
- Migration notes

Testing & debugging

Performance & optimization

Deployment

Resources

Reference

- Widget index
- API reference
- Package site

Using the Performance view

Docs > Development > Tools > DevTools > Using the Performance view

Contents

Note: The performance view works with mobile apps only. Use Chrome DevTools to analyze performance of a web app.

What is it?

The performance view allows you to record and profile a session from your Dart application.

Note: If you are running a Flutter application, use a profile build to analyze performance. CPU profiles are not indicative of release performance unless your Flutter application is run in profile mode.

CPU Profiler

Start recording a CPU profile by clicking Record. When you are done recording, click Stop. At this point, CPU profiling data is pulle from the VM and displayed in the profiler views (Call Tree, Bottom Up, and Flame Chart).

Profile granularity

The default rate at which the VM collects CPU samples is 1 sample / 250 μs. This is selected by default on the Performance view "Profile granularity: medium". This rate can be modified via the selector at the top of the page. The sampling rates for low, mediur and high granularity are 1 / 50 μs, 1 / 250 μs, and 1 / 1000 μs, respectively. It is important to know the trade-offs of modifying this setting.

A **higher granularity** profile has a higher sampling rate, and therefore yields a fine-grained CPU profile with more samples. This ma also impact performance of your app since the VM is being interrupted more often to collect samples. This also causes the VM's CPU sample buffer to overflow more quickly. The VM has limited space where it can store CPU sample information. At a higher sampling rate, the space fills up and begins to overflow sooner than it would have if a lower sampling rate was used. This means you may not have access to CPU samples from the beginning of the recorded profile.

A **lower granularity** profile has a lower sampling rate, and therefore yields a coarse-grained CPU profile with fewer samples. Howe this impacts your app's performance less. The VM's sample buffer also fills more slowly, so you can see CPU samples for a longe period of app run time. This means that you have a better chance of viewing CPU samples from the beginning of the recorded pro

Flame chart

This tab of the profiler shows CPU samples for the recorded duration. This chart should be viewed as a top-down stack trace, whe the top-most stack frame calls the one below it. The width of each stack frame represents the amount of time it consumed the C Stack frames that consume a lot of CPU time may be a good place to look for possible performance improvements.

Get started

Samples & tutorials

Development

- User interface
- Data & backend
- Accessibility & internationalization
- Platform integration
- Packages & plugins
- Add Flutter to existing app
- Tools & techniques
  - Android Studio & IntelliJ
  - Visual Studio Code
  - DevTools
    - Overview
    - Install from Android Studio & IntelliJ
    - Install from VS Code
    - Install from command line
    - Flutter inspector
    - Timeline view
    - Memory view
    - Performance view
    - Debugger
    - Logging view
  - Flutter SDK
  - Hot reload
  - Code formatting
- Migration notes

Testing & debugging

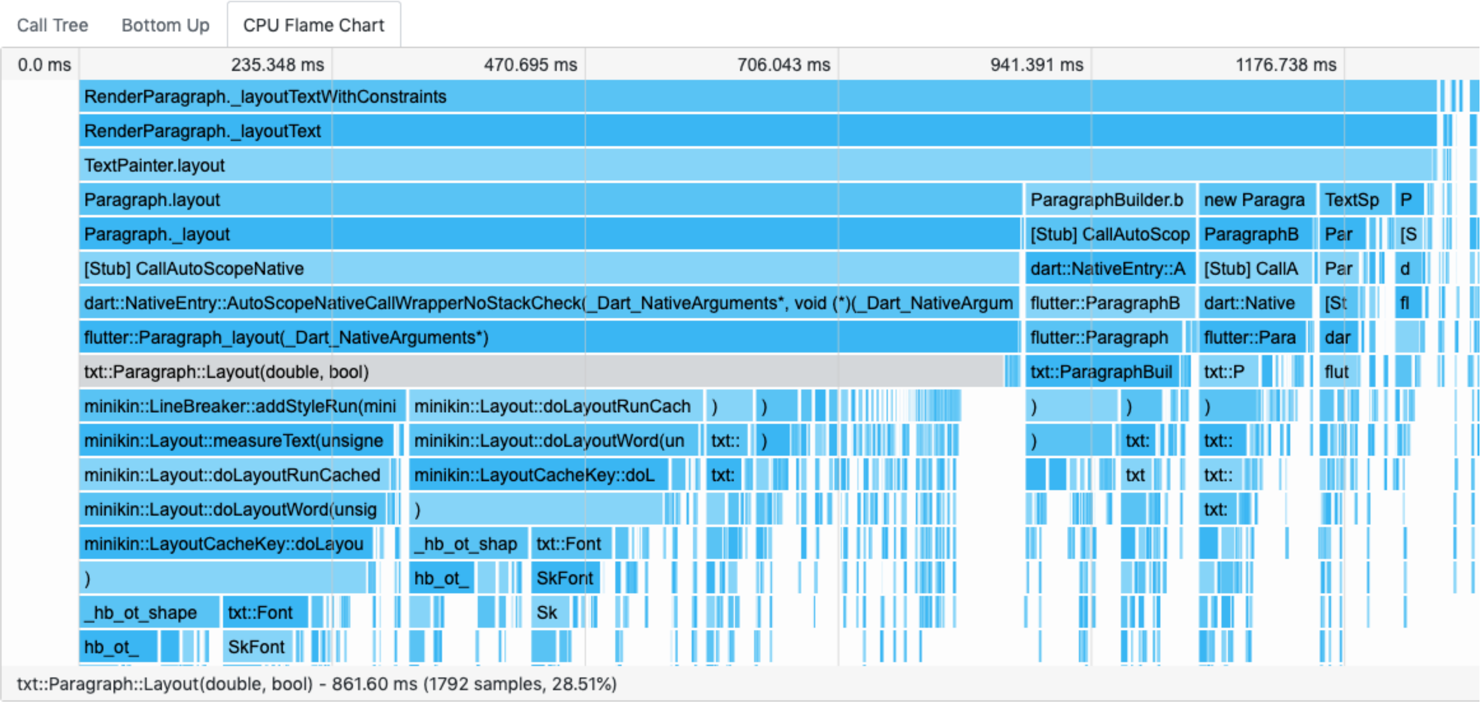
Performance & optimization

Deployment

Resources

Reference

- Widget index
- API reference
- Package site



Call tree

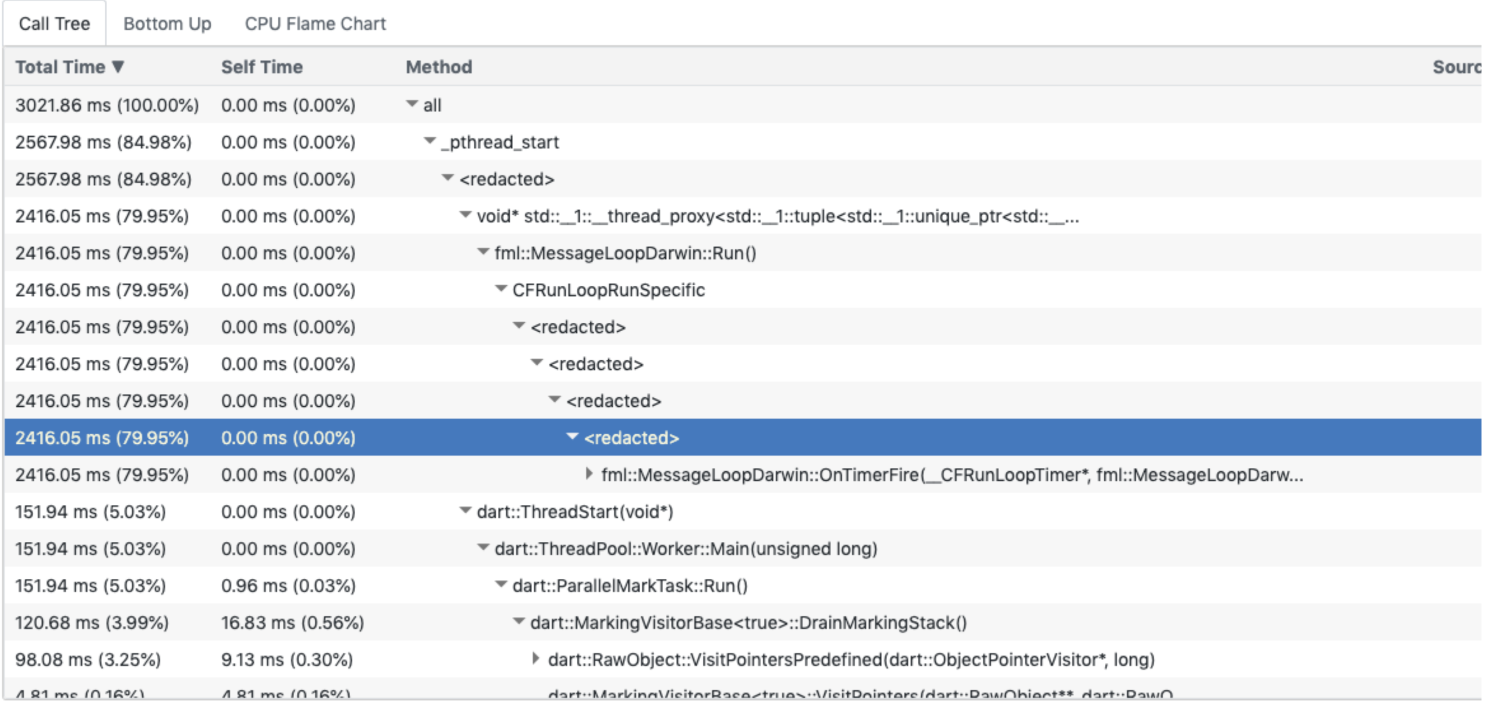
The call tree view shows the method trace for the CPU profile. This table is a top-down representation of the profile, meaning that method can be expanded to show its *callees*.

Total time
Time the method spent executing its own code as well as the code for its callees.

Self time
Time the method spent executing only its own code.

Method
Name of the called method.

Source
File path for the method call site.



Call Tree | Bottom Up | CPU Flame Chart

Total Time ▼	Self Time	Method	Sourc
3021.86 ms (100.00%)	0.00 ms (0.00%)	▼ all	
2567.98 ms (84.98%)	0.00 ms (0.00%)	▼ _pthread_start	
2567.98 ms (84.98%)	0.00 ms (0.00%)	▼ <redacted>	
2416.05 ms (79.95%)	0.00 ms (0.00%)	▼ void* std::__1::__thread_proxy<std::__1::tuple<std::__1::unique_ptr<std::__...	
2416.05 ms (79.95%)	0.00 ms (0.00%)	▼ fml::MessageLoopDarwin::Run()	
2416.05 ms (79.95%)	0.00 ms (0.00%)	▼ CFRunLoopRunSpecific	
2416.05 ms (79.95%)	0.00 ms (0.00%)	▼ <redacted>	
2416.05 ms (79.95%)	0.00 ms (0.00%)	▼ <redacted>	
2416.05 ms (79.95%)	0.00 ms (0.00%)	▼ <redacted>	
2416.05 ms (79.95%)	0.00 ms (0.00%)	▼ <redacted>	
2416.05 ms (79.95%)	0.00 ms (0.00%)	▶ fml::MessageLoopDarwin::OnTimerFire(__CFRunLoopTimer*, fml::MessageLoopDarw...	
151.94 ms (5.03%)	0.00 ms (0.00%)	▼ dart::ThreadStart(void*)	
151.94 ms (5.03%)	0.00 ms (0.00%)	▼ dart::ThreadPool::Worker::Main(unsigned long)	
151.94 ms (5.03%)	0.96 ms (0.03%)	▼ dart::ParallelMarkTask::Run()	
120.68 ms (3.99%)	16.83 ms (0.56%)	▼ dart::MarkingVisitorBase<true>::DrainMarkingStack()	
98.08 ms (3.25%)	9.13 ms (0.30%)	▶ dart::RawObject::VisitPointersPredefined(dart::ObjectPointerVisitor*, long)	
4.81 ms (0.16%)	4.81 ms (0.16%)	dart::MarkingVisitorBase<true>::VisitPointers(dart::RawObject**, dart::RawO	

Bottom up

The bottom up view shows the method trace for the CPU profile but, as the name suggests, it’s a bottom-up representation of the profile. This means that each top-level method in the table is actually the last method in the call stack for a given CPU sample (in other words, it’s the leaf node for the sample).

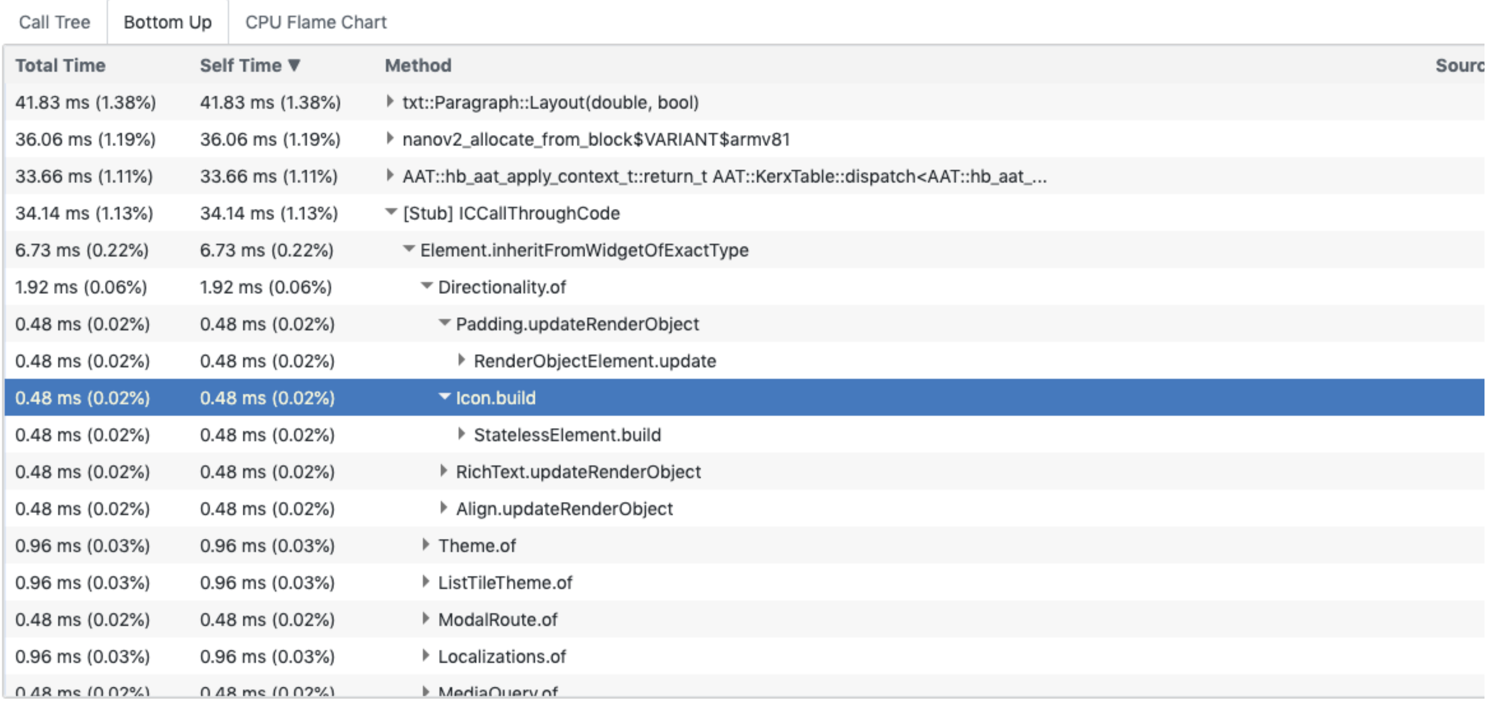
In this table, a method can be expanded to show its *callers*.

Total time
Time the method spent executing its own code as well as the code for its callee.

Self time
For top-level methods in the bottom-up tree (leaf stack frames in the profile), this is the time the method spent executing only its code. For sub nodes (the callers in the CPU profile), this is the self time of the callee when being called by the caller. In the followi example, the self time of the caller `Element.updateSlotForChild.visit()` is equal to the self time of the callee `[Stub] OneArgCheckInLineCache` when being called by the caller.

Method
Name of the called method.

Source
File path for the method call site.



Total Time	Self Time ▼	Method	Sourc
41.83 ms (1.38%)	41.83 ms (1.38%)	▸ txt::Paragraph::Layout(double, bool)	
36.06 ms (1.19%)	36.06 ms (1.19%)	▸ nanov2_allocate_from_block$VARIANT$armv81	
33.66 ms (1.11%)	33.66 ms (1.11%)	▸ AAT::hb_aat_apply_context_t::return_t AAT::KerxTable::dispatch<AAT::hb_aat_...	
34.14 ms (1.13%)	34.14 ms (1.13%)	▾ [Stub] ICCallThroughCode	
6.73 ms (0.22%)	6.73 ms (0.22%)	▾ Element.inheritFromWidgetOfExactType	
1.92 ms (0.06%)	1.92 ms (0.06%)	▾ Directionality.of	
0.48 ms (0.02%)	0.48 ms (0.02%)	▾ Padding.updateRenderObject	
0.48 ms (0.02%)	0.48 ms (0.02%)	▸ RenderObjectElement.update	
0.48 ms (0.02%)	0.48 ms (0.02%)	▾ Icon.build	
0.48 ms (0.02%)	0.48 ms (0.02%)	▸ StatelessElement.build	
0.48 ms (0.02%)	0.48 ms (0.02%)	▸ RichText.updateRenderObject	
0.48 ms (0.02%)	0.48 ms (0.02%)	▸ Align.updateRenderObject	
0.96 ms (0.03%)	0.96 ms (0.03%)	▸ Theme.of	
0.96 ms (0.03%)	0.96 ms (0.03%)	▸ ListTileTheme.of	
0.48 ms (0.02%)	0.48 ms (0.02%)	▸ ModalRoute.of	
0.96 ms (0.03%)	0.96 ms (0.03%)	▸ Localizations.of	
0.48 ms (0.02%)	0.48 ms (0.02%)	▸ MediaQuery.of	

Get started

Samples & tutorials

Development

- User interface
- Data & backend
- Accessibility & internationalization
- Platform integration
- Packages & plugins
- Add Flutter to existing app

Except as otherwise noted, this work is licensed under a Creative Commons Attribution 4.0 International License, and code samples are licensed under the BSD License.

- Install from Android Studio & IntelliJ
- Install from VS Code
- Install from command line
- Flutter inspector
- Timeline view
- Memory view
- Performance view
- Debugger
- Logging view
- Flutter SDK
- Hot reload
- Code formatting
- Migration notes

Testing & debugging

Performance & optimization

Deployment

Resources

Reference

- Widget index
- API reference
- Package site