Animations overview

Contents

The animation system in Flutter is based on typed `Animation` objects. Widgets can either incorporate these animations in their bu functions directly by reading their current value and listening to their state changes or they can use the animations as the basis o more elaborate animations that they pass along to other widgets.

Animation

The primary building block of the animation system is the `Animation` class. An animation represents a value of a specific type tha can change over the lifetime of the animation. Most widgets that perform an animation receive an `Animation` object as a parame from which they read the current value of the animation and to which they listen for changes to that value.

addListener

Whenever the animation's value changes, the animation notifies all the listeners added with `addListener`. Typically, a `State` objec that listens to an animation will call `setState` on itself in its listener callback to notify the widget system that it needs to rebuild w the new value of the animation.

This pattern is so common that there are two widgets that help widgets rebuild when animations change value: `AnimatedWidget` `AnimatedBuilder`. The first, `AnimatedWidget`, is most useful for stateless animated widgets. To use `AnimatedWidget`, simply subclass it and implement the `build` function. The second, `AnimatedBuilder`, is useful for more complex widgets that wish to include an animation as part of a larger build function. To use `AnimatedBuilder`, simply construct the widget and pass it a `build` function.

addStatusListener

Animations also provide an `AnimationStatus`, which indicates how the animation will evolve over time. Whenever the animation's status changes, the animation notifies all the listeners added with `addStatusListener`. Typically, animations start out in the `dismissed` status, which means they're at the beginning of their range. For example, animations that progress from 0.0 to 1.0 will `dismissed` when their value is 0.0. An animation might then run `forward` (e.g., from 0.0 to 1.0) or perhaps in `reverse` (e.g., from 1 to 0.0). Eventually, if the animation reaches the end of its range (e.g., 1.0), the animation reaches the `completed` status.

AnimationController

To create an animation, first create an `AnimationController`. As well as being an animation itself, an `AnimationController` lets control the animation. For example, you can tell the controller to play the animation `forward` or `stop` the animation. You can also `fling` animations, which uses a physical simulation, such as a spring, to drive the animation.

Once you've created an animation controller, you can start building other animations based on it. For example, you can create a `ReverseAnimation` that mirrors the original animation but runs in the opposite direction (e.g., from 1.0 to 0.0). Similarly, you can create a `CurvedAnimation` whose value is adjusted by a curve.

Tweens

To animate beyond the 0.0 to 1.0 interval, you can use a `Tween<T>`, which interpolates between its `begin` and `end` values. Many ty have specific `Tween` subclasses that provide type-specific interpolation. For example, `ColorTween` interpolates between colors an `RectTween` interpolates between rects. You can define your own interpolations by creating your own subclass of `Tween` and overri its `lerp` function.

By itself, a tween just defines how to interpolate between two values. To get a concrete value for the current frame of an animatic you also need an animation to determine the current state. There are two ways to combine a tween with an animation to get a concrete value:

1. You can `evaluate` the tween at the current value of an animation. This approach is most useful for widgets that are already listening to the animation and hence rebuilding whenever the animation changes value.

2. You can `animate` the tween based on the animation. Rather than returning a single value, the animate function returns a nev `Animation` that incorporates the tween. This approach is most useful when you want to give the newly created animation tc another widget, which can then read the current value that incorporates the tween as well as listen for changes to the value

Architecture

Animations are actually built from a number of core building blocks.

Scheduler

The `SchedulerBinding` is a singleton class that exposes the Flutter scheduling primitives.

For this discussion, the key primitive is the frame callbacks. Each time a frame needs to be shown on the screen, Flutter's engine triggers a "begin frame" callback which the scheduler multiplexes to all the listeners registered using `scheduleFrameCallback()`. these callbacks are given the official time stamp of the frame, in the form of a `Duration` from some arbitrary epoch. Since all the callbacks have the same time, any animations triggered from these callbacks will appear to be exactly synchronised even if they t a few milliseconds to be executed.

Tickers

The `Ticker` class hooks into the scheduler's `scheduleFrameCallback()` mechanism to invoke a callback every tick.

A `Ticker` can be started and stopped. When started, it returns a `Future` that will resolve when it is stopped.

Each tick, the `Ticker` provides the callback with the duration since the first tick after it was started.

Because tickers always give their elapsed time relative to the first tick after they were started, tickers are all synchronised. If you s three ticks at different times between two frames, they will all nonetheless be synchronised with the same starting time, and will subsequently tick in lockstep.

Simulations

The `Simulation` abstract class maps a relative time value (an elapsed time) to a double value, and has a notion of completion.

In principle simulations are stateless but in practice some simulations (for example, `BouncingScrollSimulation` and `ClampingScrollSimulation`) change state irreversibly when queried.

There are various concrete implementations of the `Simulation` class for different effects.

Animatables

The `Animatable` abstract class maps a double to a value of a particular type.

`Animatable` classes are stateless and immutable.

Tweens

The `Tween` abstract class maps a double value nominally in the range 0.0-1.0 to a typed value (e.g. a `Color`, or another double). It an `Animatable`.

It has a notion of an output type (`T`), a `begin` value and an `end` value of that type, and a way to interpolate (`lerp`) between the begi and end values for a given input value (the double nominally in the range 0.0-1.0).

`Tween` classes are stateless and immutable.

Composing animatables

Passing an `Animatable<double>` (the parent) to an `Animatable`'s `chain()` method creates a new `Animatable` subclass that appli the parent's mapping then the child's mapping.

Curves

The `Curve` abstract class maps doubles nominally in the range 0.0-1.0 to doubles nominally in the range 0.0-1.0.

`Curve` classes are stateless and immutable.

Animations

The `Animation` abstract class provides a value of a given type, a concept of animation direction and animation status, and a liste interface to register callbacks that get invoked when the value or status change.

Some subclasses of `Animation` have values that never change (`kAlwaysCompleteAnimation`, `kAlwaysDismissedAnimation`, `AlwaysStoppedAnimation`); registering callbacks on these has no effect as the callbacks are never called.

The `Animation<double>` variant is special because it can be used to represent a double nominally in the range 0.0-1.0, which is th input expected by `Curve` and `Tween` classes, as well as some further subclasses of `Animation`.

Some `Animation` subclasses are stateless, merely forwarding listeners to their parents. Some are very stateful.

Composable animations

Most `Animation` subclasses take an explicit "parent" `Animation<double>`. They are driven by that parent.

The `CurvedAnimation` subclass takes an `Animation<double>` class (the parent) and a couple of `Curve` classes (the forward and reverse curves) as input, and uses the value of the parent as input to the curves to determine its output. `CurvedAnimation` is immutable and stateless.

The `ReverseAnimation` subclass takes an `Animation<double>` class as its parent and reverses all the values of the animation. It assumes the parent is using a value nominally in the range 0.0-1.0 and returns a value in the range 1.0-0.0. The status and directi of the parent animation are also reversed. `ReverseAnimation` is immutable and stateless.

The `ProxyAnimation` subclass takes an `Animation<double>` class as its parent and merely forwards the current state of that par However, the parent is mutable.

The `TrainHoppingAnimation` subclass takes two parents, and switches between them when their values cross.

Animation Controllers

The `AnimationController` is a stateful `Animation<double>` that uses a `Ticker` to give itself life. It can be started and stopped. E tick, it takes the time elapsed since it was started and passes it to a `Simulation` to obtain a value. That is then the value it report the `Simulation` reports that at that time it has ended, then the controller stops itself.

The animation controller can be given a lower and upper bound to animate between, and a duration.

In the simple case (using `forward()`, `reverse()`, `play()`, or `resume()`), the animation controller simply does a linear interpolation from the lower bound to the upper bound (or vice versa, for the reverse direction) over the given duration.

When using `repeat()`, the animation controller uses a linear interpolation between the given bounds over the given duration, but not stop.

When using `animateTo()`, the animation controller does a linear interpolation over the given duration from the current value to the given target. If no duration is given to the method, the default duration of the controller and the range described by the controller's lower bound and upper bound is used to determine the velocity of the animation.

When using `fling()`, a `Force` is used to create a specific simulation which is then used to drive the controller.

When using `animateWith()`, the given simulation is used to drive the controller.

These methods all return the future that the `Ticker` provides and which will resolve when the controller next stops or changes simulation.

Attaching animatables to animations

Passing an `Animation<double>` (the new parent) to an `Animatable`'s `animate()` method creates a new `Animation` subclass that like the `Animatable` but is driven from the given parent.