Dealing with box constraints

Contents

Note: You might be directed to this page if the framework detects a problem involving box constraints.

In Flutter, widgets are rendered by their underlying `RenderBox` objects. Render boxes are given constraints by their parent, and siz themselves within those constraints. Constraints consist of minimum and maximum widths and heights; sizes consist of a speci width and height.

Generally, there are three kinds of boxes, in terms of how they handle their constraints:

- Those that try to be as big as possible. For example, the boxes used by `Center` and `ListView`.
- Those that try to be the same size as their children. For example, the boxes used by `Transform` and `Opacity`.
- Those that try to be a particular size. For example, the boxes used by `Image` and `Text`.

Some widgets, for example `Container`, vary from type to type based on their constructor arguments. In the case of `Container`, it defaults to trying to be as big as possible, but if you give it a `width`, for instance, it tries to honor that and be that particular size.

Others, for example `Row` and `Column` (flex boxes) vary based on the constraints they are given, as described below in the “Flex” section.

The constraints are sometimes “tight”, meaning that they leave no room for the render box to decide on a size (for example, if the minimum and maximum width are the same, it is said to have a tight width). An example of this is the `App` widget, which is contai by the `RenderView` class: the box used by the child returned by the application’s `build` function is given a constraint that forces it exactly fill the application’s content area (typically, the entire screen). Many of the boxes in Flutter, especially those that just take a single child, pass their constraint on to their children. This means that if you nest a bunch of boxes inside each other at the root o your application’s render tree, they’ll all exactly fit in each other, forced by these tight constraints.

Some boxes *loosen* the constraints, meaning the maximum is maintained but the minimum is removed. For example, `Center`.

Unbounded constraints

In certain situations, the constraint that is given to a box is *unbounded*, or infinite. This means that either the maximum width or t maximum height is set to `double.INFINITY`.

A box that tries to be as big as possible won’t function usefully when given an unbounded constraint and, in debug mode, such a combination throws an exception that points to this file.

The most common cases where a render box finds itself with unbounded constraints are within flex boxes (`Row` and `Column`), and **within scrollable regions** (`ListView` and other `ScrollView` subclasses).

In particular, `ListView` tries to expand to fit the space available in its cross-direction (for example, if it’s a vertically-scrolling block tries to be as wide as its parent). If you nest a vertically scrolling `ListView` inside a horizontally scrolling `ListView`, the inner one to be as wide as possible, which is infinitely wide, since the outer one is scrollable in that direction.

Flex

Flex boxes themselves (`Row` and `Column`) behave differently based on whether they are in bounded constraints or unbounded constraints in their given direction.

In bounded constraints, they try to be as big as possible in that direction.

In unbounded constraints, they try to fit their children in that direction. In this case, you cannot set `flex` on the children to anythin other than 0 (the default). In the widget library, this means that you cannot use `Expanded` when the flex box is inside another flex b or inside a scrollable. If you do, you’ll get an exception message pointing you at this document.

In the *cross* direction, for example, in the width for `Column` (vertical flex) or in the height for `Row` (horizontal flex), they must never b unbounded, otherwise they would not be able to reasonably align their children.