CodeMirror 6
design doc

This document describes the architecture of CodeMirror 6, and the main changes compared to earlier versions.

Overview

Our architecture separates the view of the editor—the thing users see in their browser and interact with—and the editor state. When you create a view, you give it a state, which holds things like the document and selection. When the user interacts with that view, the view dispatches transactions. Those can be used to create a new state, at which point the view is updated to show that state.

The view displays the document as editable DOM content. Changes to that content are detected and dispatched as state transactions. The DOM selection is kept in sync with the primary selection from the state.

The DOM

The most striking difference between CodeMirror 6 and versions 2 to 5 is that in the new version, the visible representation of the document is editable, using the browser's `contentEditable` feature. Because `contentEditable` is known to be difficult to work with, previous versions drew an uneditable representation of the document and implemented editable behavior almost entirely in the library.

This was a good approach, in 2011, because `contentEditable` was terribly buggy, and browsers awfully incompatible. But it's 2018, Internet Explorer is almost extinct (we do want to support version 11, possibly with some features degraded), and other browsers have come a long way. Our experience with ProseMirror shows that you *can* build something solid on `contentEditable` now.

Letting the browser handle the things that it can do anyway is preferable to reimplementing them in JavaScript. The old CodeMirror had its own implementation of mouse selection, selection drawing, and even the whole bidi algorithm (which computes the order in which mixed-direction text is laid out). That's a lot of code, a lot of space for bugs, and it still in some cases fell short of the native implementation.

Since a fake selection is not going to work well on mobile, we tried to bolt a mode that uses `contentEditable` onto the existing architecture a few years ago. This works, sometimes, on some mobile browsers, but never really got solid, because the codebase wasn't really built for it.

And that's one of the things the rewrite addressed. By using `contentEditable` from the ground up, and stealing some ideas from ProseMirror, we're able to do a lot better than CodeMirror 5.

The editor core is just an editable element that is synced with a model of the document and selection. When the DOM changes, those changes are parsed and applied to the document model. When the document model changes, the DOM is updated to match it.

There's a feature called *decorations* (similar to `markText` in the current version) that makes it possible to add styling and attributes to the rendered document, or even prevent pieces of it from being rendered entirely. This is used to do things like syntax coloring, highlighting warnings, and code folding.

Scrolling and Lazy Rendering

It is not uncommon for people to load enormous files into a text editor. The browser's document representation, especially when set to be editable, is not great at handling that much data—it'll use a lot of memory and get rather slow.

So, an editor like this is expected to be smart about what it renders to the DOM—it must make sure to fill the visible viewport, but the content that the user cannot see doesn't have to be rendered. That complicates everything quite a bit.

Previous versions of the editor went to extraordinary lengths to detect that scrolling happened before the browser updated the screen. This led to a number of really fragile hacks and, with browsers continuing to aggressively prioritize scroll performance over scriptable control, even that didn't fully solve the problem of blank content occasionally being visible when you scroll quickly.

So we're giving up that battle—you get the native scrolling behavior, and a relatively simple, performant DOM structure, but if you scroll very fast, you may occasionally see the code flickering into existence.

Another scroll-related issue is the fixed-position gutter. With most text editors that have a line number gutter at the side, if you scroll horizontally, that gutter stays in place and the code moves behind it. That was almost impossible to do natively in the past, but current browsers support the CSS property `position: sticky`, which does pretty much exactly what we need there.

State and Transactions

In the new architecture, the editor's *state*—those values that would be necessary to reconstruct an editor just like it—are kept strictly separate from the editor *view*—the interface component that provides the editable view to the user.

The state holds the document and selection, and can be extended by plugins to store additional information, such as an undo history or some parsed representation of the code structure.

States are not directly changed. Instead, every modification goes through a *transaction*, which is an object that precisely describes the changes that are being made. From such a transaction, a new state can be created. States are immutable, so at this point the old state is also still intact—this can be very helpful when comparing them, or when integrating the editor in an architecture like Redux, which expects state data to be immutable.

Each change (document modification) is itself an object, which can be read from a transaction. This makes tracking them, reasoning about them, or sharing them to implement collaborative editing, a lot more straightforward. A common operation is *mapping* positions in the document before a change or transaction to their new position in the updated document.

The editor view does not automatically track a given editor state. Instead, transactions are *dispatched* to the view, which updates the view to the new state after that transaction. This dispatch mechanism can be overridden, which is the way you can wire an editor view into your app's data flow mechanism.

Document representation

As mentioned, the editor state is immutable, so the document representation has to be as well.

In contrast to CodeMirror 5, in which the document data structure stores all kinds of data, from highlighting info to the height the line takes up in the editor, the new document structure stores only the document text.

In order to get away from the way the previous architecture does everything at line granularity (making it slow on huge lines), the new document structure isn't structured around lines, and is addressed by a single character offset integer, rather than the `{line, ch}` objects used previously. It does store some data about line breaks, so you *can* still efficiently address it in that way, but the default way to work with text is to see it as a flat space.

Under the covers, the document is a tree data structure, making it cheap to modify (most of the structure can be reused across changes).

Modularity

The new editor will (eventually) be distributed as a number of small modules. As much functionality as possible is kept out of the core, and implemented in plugins that are distributed separately. We hope that this will make the code easier to understand by enforcing strict separation boundaries, and we want to avoid the current situation, where everything is contained in one single monster distribution, putting the burden for the whole thing on the maintainers of that package.

Plugins can define new pieces of editor state, as well as influence the view by handling DOM events or decorating the editor content. This way, it's possible to implement a wide range of functionality outside of the core. The undo history, for example, observes transactions and builds up a representation of the changes that the user made, and dispatches a transaction that reverts such changes when the undo command is activated. A language mode (incrementally) parses the code in the viewport, and adds decorations to the different tokens to provide highlighting.

A "main" editor package that depends on all of the common modules will provide a low-friction way of setting up an editor—you don't have to go hunting for dozens of packages when you are just starting out with the library. A heavily customized or slimmed-down setup might opt to avoid the main module and build an editor from pieces, possibly replacing some of those pieces with custom implementations.

Existing language modes will be slightly modified to work with the legacy mode adapter and put into a package. When we've decided what our new approach to language modes looks like (this is still a research project), we'll port the modes for major languages to that new system, and maintain those from that point on.