Design Doc for CSS ::part and ::theme

Status: under construction (2018-02-06)
Author: fergal@chromium.org
Reviewers: hayato@chromium.org, kochi@chromium.org, futhark@chromium.org

Background

Docs

Part and Theme spec

- This design doc assumes a **theme=** attribute for use with **::theme()** (spec is not yet updated)
- This design doc ignores the bulk part mapping syntaxes as they are in flux

Explainer
Selectors 3 Spec
Selectors 4 Spec
CSS Scoping spec

Terminology

A reminder of keys terms

- Rule, selector, declaration: - a Rule is a ‘,’-separated list of Selectors followed by a {} containing a ‘;’-separated list of Declarations

::part cases

The spec provides for a number of ways of giving part-names to elements and this is still in flux but fundamentally it reduces to a base case and a recursive case

- An element directly inside a shadow tree attached at **host** is given a part name by it’s **part=”part-name”** attribute and is selected by **host::part(part-name)**.
- An element contained inside a shadow tree attached at **host2** which is itself inside a shadow tree attached at **host1** is given a name inside **host2** by a **part=”part-name”** attribute on the element. It is then given a name inside **host1** by a **part=”external-part-name => part-name”** attribute on **host1**. It is not necessary to change the name in this mapping and syntax for bulk-mapping names and prefixing etc will exist.

```
<body>
  <host1>
```

```
        # shadow tree
        <host2 part="part-name => external-part-name">
          # shadow tree
          <element part="part-name">
        </host2>
      </host1>
    </body>
```

Determining whether a selector matches an element

Points of note

The part-name of an element depends on scope of the selector being considered

Although each selectable element has a **part="part-name"** attribute, the name used in the **::part(selected-part-name)** may be different from the matched element's part name due to the mappings applied at shadow-hosts. E.g. in the second case above, the element would be matched by the selector **host2::part(part-name)** if the selector was used in a rule inside **host1** but would be matched by **host1::part(external-part-name)** if the selector was used in a rule in the light tree containing **host1**. E.g.

```
<body>
  <style>
    # this selector matches span below
    host1::part(external-part-name) { color: red; }
    # this selector does not match span below
    ::part(part-name) { font-weight: bold; }
  </style>
  <host1>
    # shadow tree
    # part map: external-part-name => [<span…>]
    <style>
      # this selector matches span below
      host2::part(part-name) { font-size: 300%; }
      # this selector does not match span below
      ::part(external-part-name) { font-family: verdana; }
    </style>
    <host2 part="part-name => external-part-name">
      # shadow tree
      <span part="part-name" >red, 300%, not bold, not verdana</span>
    </host2>
  </host1>
```

```
</body>
```

We cannot allow the selector ::part(part-name) in **body**'s scope to find the span inside **host2.**

At each host, there is a multi-map of part-names to part-names

For the mapped case above, it is possible to map the same name to multiple parts. E.g. **part="part-name1 => external-part-name, part-name2 => external-part-name"**. It is also possible to just use **part="external-part-name"** directly on an element inside the shadow tree.

Favoured Proposal - ascend tree to find the right name

Store part data in element

Change ElementRareData so that it stores a list of part names similar to the current list of class names.
https://chromium-review.googlesource.com/c/chromium/src/+/958311

Change ShadowRoot (or ElementRareData?) to store a shadow part name map, a map from string to list of strings mapping internal part names to external part names.

Update resolver to find part rules

Add a new call to MatchScopedRules to match part rules.
https://chromium-review.googlesource.com/c/chromium/src/+/972766
https://chromium-review.googlesource.com/c/chromium/src/+/972788

Make CheckOne match parts

Add new case to CheckPseudoElement to match **::part()** against the element

- Reject if the element has no **part=**
- Ascend through treescopes until we reach the scope containing the selector being matched
    - At each level apply mappings to the part names of this element to find out what it's name is in this scope (it seems possible for an element to be given multiple names by a host, so we need to keep a *set* of names. the use case for giving a part 2 different names in the mapping is e.g. migrating to a new scheme or providing compatibility with alternative component).
    - Reject if it has no name in this scope
- Having reached the scope containing the selector, succeed if any of the names of the element in this scope match the name in the **::part()** component

https://chromium-review.googlesource.com/c/chromium/src/+/972772

Add implicit relation between host and part

Add a new combinator/RelationType for **kShadowPart** this combinator is inserted automatically

before **::part()** it does not have any syntax. It's behaviour is to ascend the tree from the element matched by **::part()** to find the shadow host element that is within the same scope as the rule being matched. It does not consider any renaming etc as this has already been considered by the code that matched **::part()**.
https://chromium-review.googlesource.com/c/chromium/src/+/970925

Discussion

E.g. for **host1::part(external-part-name)** in **body** we would descend into **host1** and then **host2**, finding a span with **part="part-name"**. We would then ascend the tree, applying the mapping and see that the name is now **external-part-name**. We would ascend the tree once more, with no mapping applied, keeping the name the same and we have reached the selectors tree, so we match.

We would also have to notice that when the part name is not exposed (with either a mapping or under the same name), we may not ascend to the containing tree but no further.

Pros

- Fits to existing code

Cons

- Selectors using ::part must be considered against every element descended from their scope. Even components that expose no parts will have their entire tree considered for rules that can never match (see optimizations).

Optimizations

- For a given element it may be worth caching the part names for each parent scope as these will be needed for all ::part-using rules
- Similarly, share a cache among elements inside the same shadow root that records for each scope whether *any* part names are visible from the current root. So that e.g. if there is a rule at the document scope and a component which exposes no parts but contains components that have parts, we can easily tell for the elements of these inner parts that document-scope rules never match.
- Following on from that add a part_ruleset to RuleSet so that we can skip consider part rules entirely when we know the rules cannot match any elements in this root. This saves us iterating one-by-over over each rule. Maybe not enough of a win.

Alternative Proposal - expanding mapped names

When we hit **host1** we would search directly inside its light-tree for elements with **part="external-part-name"**. We would also search for hosts and when descending into them,

transform the selector from **::part(external-part-name)** to **::part(part-name)** and continue the descent from there. One problem is that Multiple part-names may have been mapped to the same name, so we actually have to search for **::part(part-name1|part-name2|...)** (or for each in turn). As we descend through multiple hosts this list will morph, names that are irrelevant to the host will be dropped, others will be further expanded.

We would also need to flag somehow that the selector is special and if a match is made we should not start ascending from the matched element but from the outermost host element

Pros

- Lazy
- Simple
- Could be improved with caching at the cost of some complexity

Cons

- The list of names may grow large and be inefficient to match against

Alternative Proposal - maintaining maps of elements

Following the spec literally, each host would maintain a map of part names to the elements which have those part names, all the way down through all shadow-trees.

Pros

- Matches spec literally

Cons

- Keeping this mapping up to date with every change of the shadow-tree(s) seems hard. Do we have the infrastructure to do it incrementally?

Conclusion

The first 2 proposals above seem like they are equivalent but just extremes of laziness/eagerness. The proposal to “ascend tree to find the right name” seems to fit well with the current implementation

Style Invalidation - WIP

As documented here, Blink has a mechanism for reducing the amount of style recalculation performed when an element’s properties change. The part= attribute must be accounted for in that.

Changes which need invalidation

Changing part=

An element which changes the value of its part= attribute should be invalidated.

Changing partmap=

An element which changes the value of its partmap= attribute should have it's descendants considered for invalidation. We could

- invalidate all shadow-including-descendants with a part= attribute
- traverse the tree paying attention to partmap= mappings so that we can prune shadow trees that did not map the old or new values of the part name

Other changes

If there is a rule like **.c c-e::part(partp)** and an element changes to/from the class **c** then we need to consider invalidating all of the shadow-including-descendents of all of the shadow hosts which have a part= attribute. Again we could use partmap to prune the tree.

Integration with existing invalidation sets

A rule such as

```
.a .b .c::part(partname) { … }
```

Should result in invalidation sets as follows

```
.a { part(partname) }
.b { part(partname) }
.c { part(partname) }
```

Where the meaning of `.a { part(partname) }` is that when an element has the class "a" added or removed, we find all descendants that could be matched by ::part(partname) and invalidate them. This could either be

1. All elements with a part attribute (in which case, we do not store partname with the invalidation set, we just store a flag that indicates the need for part invalidation)
2. All elements with a part attribute but using absence of partmap to prune the descent
3. All elements with a part attribute =partname, using partmap to determine the mapped names in every tree

Handling ::theme

Examples

Some samples of what would be rendered differently if this was implemented.

Simple ::part

JSFiddle

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      p { color: blue; }
      element-details::part(partp) { color: green; }
    </style>
  </head>
  <body>
    <template id="element-details-template">
      <style>
        p { color: red; }
      </style>
      <p part="partp">
        If this is green, it's working.
      </p>
    </template>
    <p class="f">some text</p>
    <element-details />
  </body>
  <script>
    customElements.define('element-details',
    class extends HTMLElement {
      constructor() {
        super();
        var template = document
          .getElementById('element-details-template')
          .content;
        const shadowRoot = this.attachShadow({mode: 'open'})
```

```
          .appendChild(template.cloneNode(true));
      }
    })
  </script>
</html>
```

Nested ::part

TODO

Simple ::theme

[JSFiddle]

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      p { color: blue; }
      body::theme(partp) { color: green; }
    </style>
  </head>
  <body>
    <template id="element-details-template">
      <style>
        p { color: red; }
      </style>
      <p theme="partp">
        If this is green, it's working.
      </p>
    </template>
    <p class="f">some text</p>
    <element-details />
  </body>
  <script>
    customElements.define('element-details',
    class extends HTMLElement {
      constructor() {
        super();
        var template = document
          .getElementById('element-details-template')
          .content;
        const shadowRoot = this.attachShadow({mode: 'open'})
```

```
            .appendChild(template.cloneNode(true));
        }
      })
    </script>
</html>
```