05:52:27.218 [0] HMR.refreshToken token:AAAAAgAAADdZtDxfAAUxNTgxMQAAAAAAAF0EFQxsAAnjQA6vhnRCTIANWHgrteeSo1qyqMYag 05:57:35.392 demo_getToken uid:15811

PSA: How to write reliable layout tests

alexclarke@, rmcilroy@, skyostil@

Last updated: July 14, 2015

During the development of the Blink Scheduler we ran into a number of anti-patterns in layout tests which made the tests fail if the order or timing of task scheduling changed substantially. This document describes some of the most common pitfalls and how to avoid them when writing layout tests.

Unit testing vs. integration testing

It's generally better to write unit tests for C++ classes in C++, although this can be prohibitively difficult for tightly coupled code. Layout tests are great for integration style tests, and ideally your feature will have both.

Most layout tests aspire to be simple unit tests that verify the functionality of a single feature, but actually end up being integration tests by virtue of using high level testing facilities such as pixel screenshots or layer tree text dumps to verify their output. As a result these types of tests can start failing if any part of the the system is modified so that the test output changes, even if that change isn't related to the feature being tested.

Therefore, when designing a layout test, consider if you really need to verify a pixel-accurate result or the exact configuration of LayoutObjects, or whether your test would work better as a series of more focused js-test or testharness assertions.

Bad	Better
`<script>`	`<script src="js-test.js"></script>`

```
var result = 1 + 1;
document.querySelector('#result')
    .innerText = result;
</script>

or

<script>
var result = 1 + 1;
document.querySelector('#result')
    .innerText = result;
window.testRunner.dumpAsText();
</script>
```

```
<script>
description('Tests that basic
    arithmetic works.');

var result = 1 + 1;
shouldBe('result', '2');
</script>

or

<script src="testharness.js">
</script>

<script>
test(function() {
  assert_true(1 + 1 == 2);
}, 'Tests that basic arithmetic
    works.');
</script>
```

Another convenient way to avoid spurious changes in layout test output is to use ref tests. Instead of comparing an HTML file against a pre-generated screenshot, a reference test checks that the output from two separate HTML files is identical.

If you do need to check the positions or sizes of LayoutObjects, either use check-layout.js or a ref test to do this in a more reliable way. Finally, if a pixel test is your only option, refer to a guide on designing good pixels tests.

Resource loading

Always wait for your resources to finish loading before verifying assumptions about them. A common example is manipulating the contents of an **<iframe>**:

Bad

```
<iframe
  src="something.html"></iframe>

<script>
// Frame contents may not be ready.
shouldBe('frame.innerText', 'foo');
</script>
```

Better

```
<iframe onload="runTest()"
    src="something.html"></iframe>

<script>
testRunner.waitUntilDone();
function runTest() {
  // Frame contents are now ready.
```

	`  shouldBe('frame.innerText',`
	`           'foo');`
	`  testRunner.notifyDone();`
	`}`
	`</script>`

Additionally beware of polling the `readyState` flag of an iframe: iframes start out with the about:blank document loaded with `readyState == 'complete'` (example). This means your test might conclude that the iframe has finished loading even though it is still displaying the initial blank page. The fix is to either use a onload listener instead (preferred) or inspect the iframe's URL in conjunction with `readyState`.

Bad	Better

`<iframe`
`  src="target.html"></iframe>`
 	
`<script>`
`if (frame.readyState == 'complete')`
`{`
`  shouldBe('frame.innerText', ...);`
`}`
`</script>`
	`<iframe`
	`  src="target.html"></iframe>`
	
	`<script>`
	`if (frame.readyState == 'complete'`
	`    && frame.contentWindow`
	`    .location.href != 'about:blank')`
	`{`
	`  shouldBe('frame.innerText', ...);`
	`}`
	`</script>`

Custom fonts can also cause test failures if the layout test terminates before all of the fonts have finished loading. Because of this the result can randomly contain fallback fonts instead of the expected ones (example). You can make this deterministic with the `onload` callback:

Bad	Better

`<style>`
`@font-face {`
`  font-family: Foo;`
`  src: url(foo.otf);`
`}`
`</style>`
	`<style>`
	`@font-face {`
	`  font-family: Foo;`
	`  src: url(foo.otf);`
	`}`
	`</style>`
	
	`<script>`

	`// Kick off loading @font-face.`
	`document.body.offsetLeft;`
	`testRunner.waitUntilDone();`
	`onload = function () {`
	`  testRunner.notifyDone();`
	`};`
	`</script>`

Incremental parsing

Blink parses HTML in incremental chunks, which can lead to surprising results. A common example is an inline **<script>** which expects to be executed before any of the HTML above it has been parsed or painted (example). Since the parser can decide to display the HTML before it has even seen the **<script>** tag, this assumption is not valid. To work around this, generate the necessary HTML from the script directly or run the script from a callback that fires when the document is in a known state.

Bad	Better

`<div style="overflow: scroll">`
`  ...`
`</div>`
 	
`<script>`
`// Can run before or after initial`
`// paint of the <div>.`
`div.scrollTop = 1000;`
`runRepaintTest();`
`</script>`
	`<div style="overflow: scroll">`
	`  ...`
	`</div>`
	
	`<script>`
	`testRunner.layoutAndPaintAsyncThen(`
	`  function() {`
	`  // Always runs after initial`
	`  // paint.`
	`  div.scrollTop = 1000;`
	`  runRepaintTest();`
	`  });`
	`</script>`

In more extreme cases incremental parsing can lead to differences in the final RenderObject tree structure depending on at exactly which points the browser decided to layout and paint the parsed HTML chunks (example). A good way to avoid these problems is to avoid comparing text or render tree dumps unless absolutely necessary.

Animations to at

The clock which is used to drive animations is not automatically synchronized with the rest of layout test execution. This means the final animation frame which is captured in a pixel dump

depends on how quickly the test happens to run. To avoid spurious differences in the results, ensure that the animation is either hidden or brought to a deterministic state (example).

Bad

```
<marquee>Hi from the 90s</marquee>
```

Better

```
<style>
  marquee { color: white };
</style>

<marquee>Hi from the 90s</marquee>
```

DOM vs. internal API state

The layout test runner exposes various APIs (`window.internals.*`) for inspecting Blink's internal state. This can lead to problems if that state is inspected before changes in the DOM have been applied to it. One example of where this can happen are properties that depend on the compositing layer tree having been updated (example):

Bad

```
<iframe src="scrollable.html"
  onload="test()">

<script>
function test() {
  shouldBe('internals.nonFast
      ScrollableRects', ...);
}
</script>
```

Better

```
<iframe src="scrollable.html"
  onload="test()">

<script>
function test() {
  // Ensure a compositing update.
  requestAnimationFrame(function() {
    shouldBe('internals.nonFast
        ScrollableRects', ...);
  });
}
</script>
```

Magic timeouts

Don't rely on timing constructs such as `setTimeout(..., 0)` because they do not guarantee that tasks such as layout or painting get performed before the timeout triggers. Instead, check for specific events such as

- `window.onload` – called when everything on the page has finished loading (including iframes).

- `requestAnimationFrame()` – invokes animation, style, layout and compositing updates (but not paint), but only after the callback has run.
- `runAfterLayoutAndPaint()` – invokes animation, style, layout, **paint,** and compositing updates and then runs your callback. (Internally uses `testRunner.layoutAndPaintAsyncThen()`).

([example])

Bad	Better

`<script>`
`testRunner.waitUntilDone();`
`div.style.color = 'blue';`
 	
`setTimeout(function() {`
`  // Surely we have painted by now?`
`  shouldBe('internals.`
`  layerTreeAsText()', ...);`
`  testRunner.notifyDone();`
`}, 10);`
`</script>`
	`<script>`
	`testRunner.waitUntilDone();`
	`div.style.color = 'blue';`
	
	`testRunner.layoutAndPaintAsyncThen(`
	`  function() {`
	`  // Guaranteed to have painted.`
	`  shouldBe('internals.`
	`  layerTreeAsText()', ...);L:`
	`  testRunner.notifyDone();`
	`  });`
	`</script>`

Testing for flakiness

Once you've written a new layout test following these guidelines, it's a good idea to verify that the test is actually reliable. An easy way to do this is to repeat the test several times:

```
$ third_party/WebKit/Tools/Scripts/run-webkit-tests shiny-new-test.html
--repeat-each=100
```

A common source of flakiness are race conditions due to asynchronous parsing as described above. These can be tricky to diagnose, but if you add some artificial sleeps to the background parser and the Chromium layout test harness you can often reliably flush out problems:

- Artificial delay patch for Blink
- Artificial delay patch for Chromium

Resources

- Layout test overview
- Layout test style guidelines
- Flakiness dashboard