ASAP

Promise and asynchronous observer libraries, as well as hand-rolled callback programs and libraries, often need a mechanism to postpone the execution of a callback until the next available event. (See Designing API's for Asynchrony.) The `asap` function executes a task **as soon as possible** but not before it returns, waiting only for the completion of the current event and previously scheduled tasks.

```
asap(function () {
    // ...
});
```

This CommonJS package provides an `asap` module that exports a function that executes a task function *as soon as possible*.

ASAP strives to schedule events to occur before yielding for IO, reflow, or redrawing. Each event receives an independent stack, with only platform code in parent frames and the events run in the order they are scheduled.

ASAP provides a fast event queue that will execute tasks until it is empty before yielding to the JavaScript engine's underlying event-loop. When a task gets added to a previously empty event queue, ASAP schedules a flush event, preferring for that event to occur before the JavaScript engine has an opportunity to perform

IO tasks or rendering, thus making the first task and subsequent tasks semantically indistinguishable. ASAP uses a variety of techniques to preserve this invariant on different versions of browsers and Node.js.

By design, ASAP prevents input events from being handled until the task queue is empty. If the process is busy enough, this may cause incoming connection requests to be dropped, and may cause existing connections to inform the sender to reduce the transmission rate or stall. ASAP allows this on the theory that, if there is enough work to do, there is no sense in looking for trouble. As a consequence, ASAP can interfere with smooth animation. If your task should be tied to the rendering loop, consider using `requestAnimationFrame` instead. A long sequence of tasks can also effect the long running script dialog. If this is a problem, you may be able to use ASAP's cousin `setImmediate` to break long processes into shorter intervals and periodically allow the browser to breathe. `setImmediate` will yield for IO, reflow, and repaint events. It also returns a handler and can be canceled. For a `setImmediate` shim, consider YuzuJS setImmediate.

Take care. ASAP can sustain infinite recursive calls without warning. It will not halt from a stack overflow, and it will not consume unbounded memory. This is behaviorally equivalent to an infinite loop. Just as with infinite loops, you can monitor a Node.js process for this behavior with a heart-beat signal. As with infinite loops, a very small amount of caution goes a long way to avoiding problems.

```
function loop() {
    asap(loop);
}
loop();
```

In browsers, if a task throws an exception, it will not interrupt the flushing of high-priority tasks. The exception will be postponed to a later, low-priority event to avoid slow-downs. In Node.js, if a task throws an exception, ASAP will resume flushing only if—and only after—the error is handled by `domain.on("error")` or `process.on("uncaughtException")`.

Raw ASAP

Checking for exceptions comes at a cost. The package also provides an `asap/raw` module that exports the underlying implementation which is faster but stalls if a task throws an exception. This internal version of the ASAP function does not check for errors. If a task does throw an error, it will stall the event queue unless you manually call `rawAsap.requestFlush()` before throwing the error, or any time after.

In Node.js, `asap/raw` also runs all tasks outside any domain. If you need a task to be bound to your domain, you will have to do it manually.

```
if (process.domain) {
    task = process.domain.bind(task);
```

```
}
rawAsap(task);
```

Tasks

A task may be any object that implements `call()`. A function will suffice, but closures tend not to be reusable and can cause garbage collector churn. Both `asap` and `rawAsap` accept task objects to give you the option of recycling task objects or using higher callable object abstractions. See the `asap` source for an illustration.

Compatibility

ASAP is tested on Node.js v0.10 and in a broad spectrum of web browsers. The following charts capture the browser test results for the most recent release. The first chart shows test results for ASAP running in the main window context. The second chart shows test results for ASAP running in a web worker context. Test results are inconclusive (grey) on browsers that do not support web workers. These data are captured automatically by Continuous Integration.

interaction problems with Mobile Internet Explorer in favor of an implementation backed on the newer and more reliable DOM `MutationObserver` interface. These changes were back-ported into this library.

In addition, ASAP factored into `asap` and `asap/raw`, such that `asap` remained exception-safe, but `asap/raw` provided a tight kernel that could be used for tasks that guaranteed that they would not throw exceptions. This core is useful for promise implementations that capture thrown errors in rejected promises and do not need a second safety net. At the same time, the exception handling in `asap` was factored into separate implementations for Node.js and browsers, using the the Browserify `browser` property in `package.json` to instruct browser module loaders and bundlers, including Browserify, Mr, and Mop, to use the browser-only implementation.

License

unisolated circumstances. Timers generally delay the flushing of ASAP's task queue for four milliseconds.

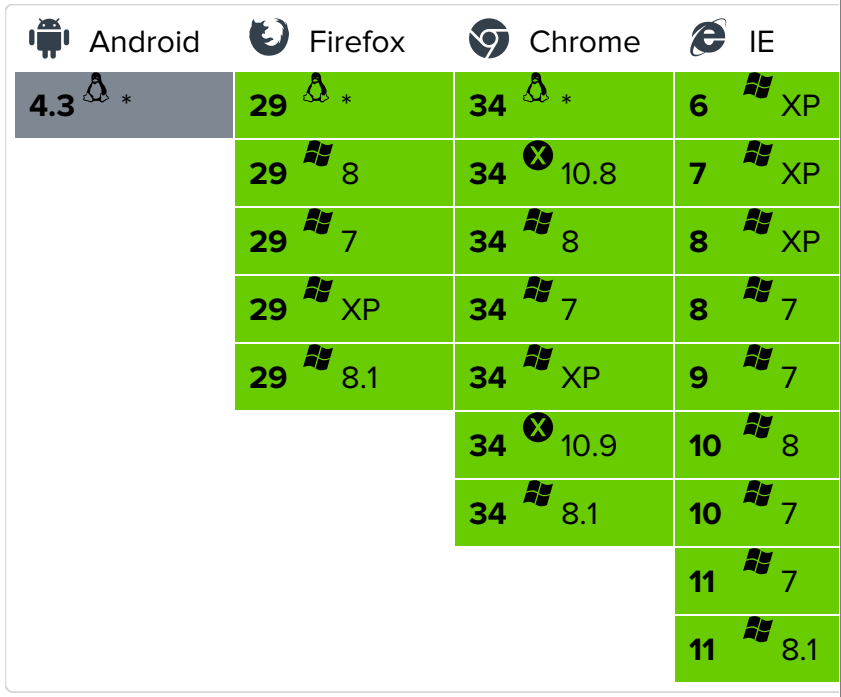
- Firefox 3–13
- Internet Explorer 6–10
- iPad Safari 4.3
- Lynx 2.8.7

Heritage

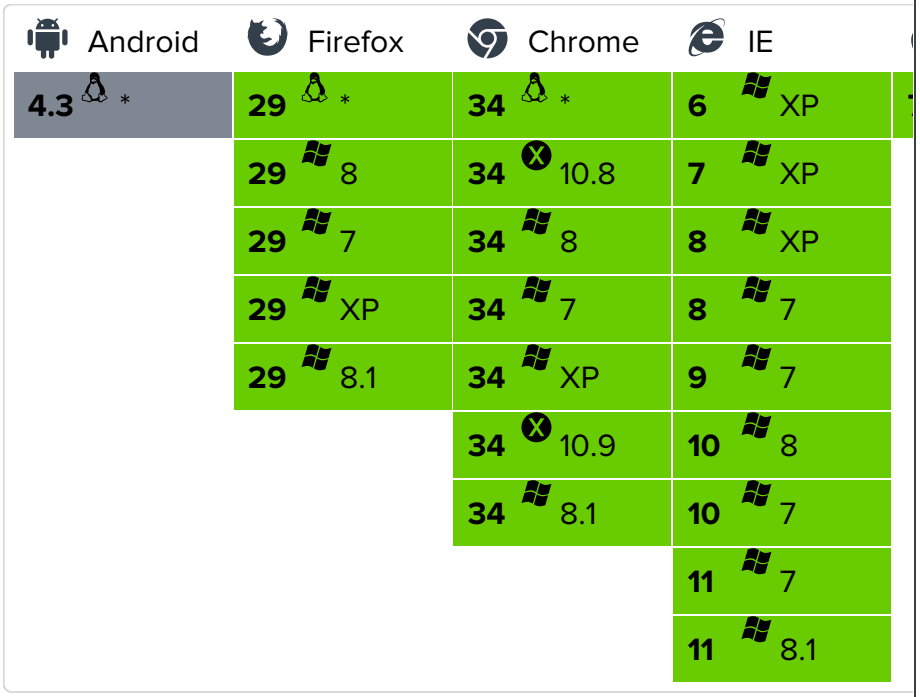
ASAP has been factored out of the Q asynchronous promise library. It originally had a naïve implementation in terms of `setTimeout`, but Malte Ubl provided an insight that `postMessage` might be useful for creating a high-priority, no-delay event dispatch hack. Since then, Internet Explorer proposed and implemented `setImmediate`. Robert Katić began contributing to Q by measuring the performance of the internal implementation of `asap`, paying particular attention to error recovery. Domenic, Robert, and Kris Kowal collectively settled on the current strategy of unrolling the high-priority event queue internally regardless of what strategy we used to dispatch the potentially lower-priority flush event. Domenic went on to make ASAP cooperate with Node.js domains.

For further reading, Nicholas Zakas provided a thorough article on The Case for setImmediate.

Ember's RSVP promise implementation later adopted the name ASAP but further developed the implentation. Particularly, The `MessagePort` implementation was abandoned due to



Browser Compatibility



Compatibility in Web Workers

Caveats

When a task is added to an empty event queue, it is not always possible to guarantee that the task queue will begin flushing immediately after the current event. However, once the task queue begins flushing, it will not yield until the queue is empty, even if the queue grows while executing tasks.

The following browsers allow the use of DOM mutation observers to access the HTML microtask queue, and thus begin flushing ASAP's task queue immediately at the end of the current event loop turn, before any rendering or IO:

- Android 4–4.3
- Chrome 26–34
- Firefox 14–29
- Internet Explorer 11
- iPad Safari 6–7.1
- iPhone Safari 7–7.1
- Safari 6–7

In the absense of mutation observers, there are a few browsers, and situations like web workers in some of the above browsers, where message channels would be a useful way to avoid falling back to timers. Message channels give direct access to the HTML task queue, so the ASAP task queue would flush after any already queued rendering and IO tasks, but without having the minimum delay imposed by timers. However, among these browsers, Internet Explorer 10 and Safari do not reliably dispatch messages, so they are not worth the trouble to implement.

- Internet Explorer 10
- Safair 5.0-1
- Opera 11-12

In the absense of mutation observers, these browsers and the following browsers all fall back to using `setTimeout` and `setInterval` to ensure that a `flush` occurs. The implementation uses both and cancels whatever handler loses the race, since `setTimeout` tends to occasionally skip tasks in