React 之 Scheduler 源码中的三个小知识点，看看你知不知道?

前言

getCurrentTime

Scheduler 中有一个 getCurrentTime 函数，它的具体实现如下：

javascript 复制代码

```
let getCurrentTime;
const hasPerformanceNow =
  typeof performance === 'object' && typeof performance.now === 'function';

if (hasPerformanceNow) {
  const localPerformance = performance;
  getCurrentTime = () => localPerformance.now();
} else {
  const localDate = Date;
  const initialTime = localDate.now();
  getCurrentTime = () => localDate.now() - initialTime;
}
```

虽然函数名为获取当前时间，但根据实现，可以看出，获取的其实是从性能测量时刻开始的毫秒数

如果环境支持 performance.now，那就直接使用 performance.now，我们看下 performance.now

performance.now

根据 MDN 对 performance.now 的介绍：

> 返回值表示为从 time origin 之后到当前调用时经过的时间

time origin 译为“时间源”，MDN 同样有介绍：

时间源是一个可以被认定为当前文档生命周期的开始节点的标准时间，计算方法如下：

- 如果脚本的 global object 是 Window, 则时间源的确定方式如下：
  - 如果当前 Document 是中加载的第一个 Window, 则时间源是创建浏览器上下文的时间。
  - 如果处于卸载窗口中已加载的先前文档的过程中， 一个确认对话框会显示出来，让用户确认是否离开前一页，则时间源是用户确认导航到新页面的这个时间，这一点是被认同的。
  - 如果以上方式都不能确定时间源, 那么时间源是创建窗口中当前 Document 的导航发生的时机。
- 如果脚本中的全局对象是 WorkerGlobalScope (意味着，该脚本以 web worker 的形式运行), 则时间源是这个 worker 被创建的时刻。
- 在所有其他情况下，时间源的值是 undefined。

简单的来说，performance.now 会返回自页面创建过了多久。

Date.now

而 Date.now 返回的是自 1970 年 1 月 1 日 00:00:00 (UTC) 到当前时间的毫秒数。

performance.now VS Date.now

1. performance.now 精度更高，以浮点数表示时间，精度最高可达微秒级（1毫秒=1000微秒），Date.now 则为毫秒级
2. performance.now()以一个恒定的速率慢慢增加的，它不会受到系统时间的影响（系统时钟可能会被手动调整或被 NTP 等软件篡改）
3. performance.now 是浏览器提供的方法，Date.now 是 JavaScript 内置的方法
4. performance.timing.navigationStart + performance.now() 约等于 Date.now()

maxSigned31BitInt

在 `unstable_scheduleCallback` 函数中，当根据优先级计算 timeout 时间时，如果是 `IdlePriority`，对应的 timeout 时间为 `IDLE_PRIORITY_TIMEOUT`，它是一个常量，值为 1073741823，这个数值是怎么算出来的呢?

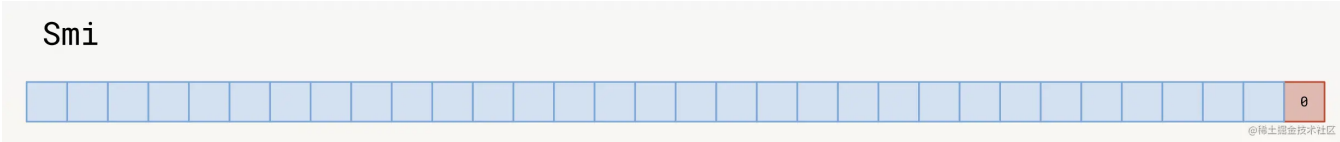
```javascript
// Max 31 bit integer. The max integer size in V8 for 32-bit systems.
// Math.pow(2, 30) - 1
// 0b11111111111111111111111111111111
var maxSigned31BitInt = 1073741823;
// Never times out
var IDLE_PRIORITY_TIMEOUT = maxSigned31BitInt;
```

根据注释可以看出，maxSigned31BitInt 表示 32 位系统下 V8 最大的整数，它是最大的 31 位整数，它的二进制为 `0b11111111111111111111111111111111`，这其中 `0b` 表示这是一个二进制数字，后面跟了一共 30 个 `1`，所以这个值是 2 的 30 次方 - 1，也就是 1073741823，这个时间大概是 12.4 天

那么问题来了，既然是 32 位，一位用于符号位，那还有 31 位呀，为什么是 2 的 30 次方？而不是 2 的 31 次方呢?

这就要说到 `Smis`，在 ECMAScript 标准中，数字会被当成 64 位双精度浮点数处理，但是一直使用 64 位存储数字是十分低效的，就比如数组索引，在规范中，它的最大值是 $2^{32}-2$，没有必要非要使用 64 位储存，所以像 V8 就建立了一个名为 `Smis` 的特殊整数表示方式。

它其实是 `Small Integer` 的缩写，翻译过来就是小整数，它的特殊地方在于它会使用最低有效位（least significant bits）标记这个值是否是 `Smis`：



所以在 32 位平台上，就只有 31 位用来储存 `Smi` 值。

isInputPending

在 shouldYieldToHost 的源码中，其实还涉及了一个 Scheduling API —— isInputPending，它是 facebook 的开发者提出的，规范地址是 wicg.github.io/is-input-pe...。

它是一个函数，作用是检测是否有等待中的用户输入事件，使用语法为：

```javascript
if (navigator.scheduling.isInputPending()){
  // ...
}
```

当这个函数执行的时候，如果浏览器有需要处理的输入事件，isInputPending() 就会返回 true。

React 辛辛苦苦实现 Fiber 结构，其目的就在于能够及时让出线程，让浏览器可以处理用户输入或者动画等，借助 isInputPending() 这个 API，我们可以快速判断出执行期间是否有用户输入，从而做出处理，保证及时响应用户输入。

通过合理使用isInputPending方法，我们可以在页面渲染时及时响应用户输入，并且，当有长耗时的JS任务要执行时，可以通过isInputPending来中断JS的执行，将控制权交还给浏览器来执行用户响应。

一个简单的使用示例如下：

javascript 复制代码

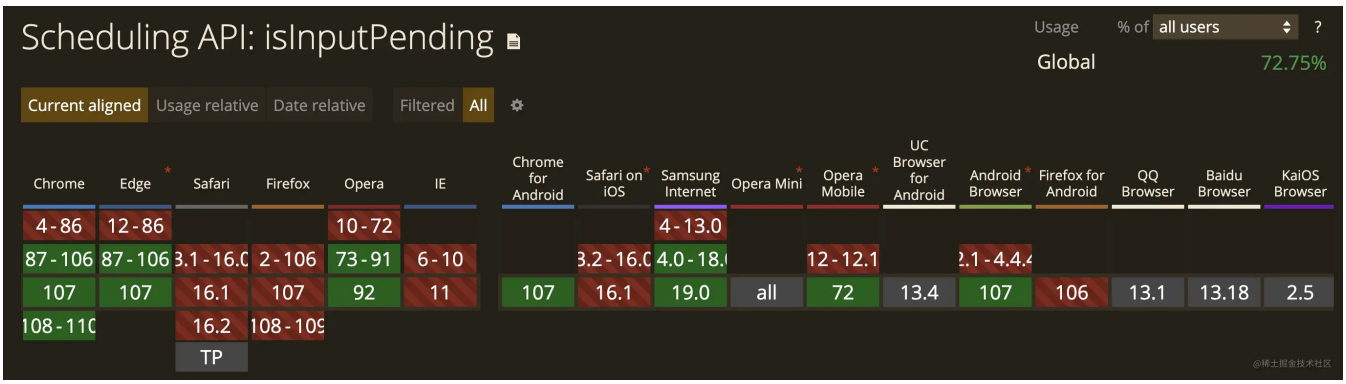
```
let taskQueue = [task1, task2, ...];
const options = {includeContinuous: true};

function doWork() {
  while (let task = taskQueue.pop()) {
    task.execute();
    if (navigator.scheduling.isInputPending(options)) {
      setTimeout(doWork, 0);
      break;
    }
  }
}

doWork();
```

在这个例子中，我们传入了一个 `{includeContinuous: true}` 对象，这是因为在默认情况下，isInputPending 并不会检测连续事件，就比如 `mousemove` 、 `pointermove` 等，设置为 true 后，也会开启检测这些事件。

但目前它的兼容性还不好，Chrome 也只有 87 版本以上才支持：



所以 React 源码中也没有正式开启使用这个 API：

javascript 复制代码

```javascript
const continuousYieldMs = 50;
const continuousInputInterval = continuousYieldMs;

const maxYieldMs = 300;
const maxInterval = maxYieldMs;

const isInputPending =
  typeof navigator !== 'undefined' &&
  navigator.scheduling !== undefined &&
  navigator.scheduling.isInputPending !== undefined
    ? navigator.scheduling.isInputPending.bind(navigator.scheduling)
    : null;

function shouldYieldToHost() {
  const timeElapsed = getCurrentTime() - startTime;
  if (timeElapsed < frameInterval) {
    return false;
  }

  // enableIsInputPending 默认 false
  if (enableIsInputPending) {
    // ...
    if (timeElapsed < continuousInputInterval) {
      if (isInputPending !== null) {
        return isInputPending();
      }
    } else if (timeElapsed < maxInterval) {
      if (isInputPending !== null) {
        return isInputPending(continuousOptions);
      }
    } else {
      return true;
    }
  }
  return true;
}
```