5 分钟撸一个前端性能监控工具

2018/07/18 · JavaScript · 性能监控

原文出处： 奇舞团 - 刘宇晨

编者按：本文作者是来自360奇舞团的前端开发工程师刘宇晨，同时也是W3C 性能工作组成员。跟着他一起学习一下前端性能监控吧～

用（上）户（帝）说，这个页面怎么这么慢，还有没有人管了？！

为什么监控

- 关注性能是工程师的本性 + 本分；
- 页面性能对用户体验而言十分关键。每次重构对页面性能的提升，仅靠工程师开发设备的测试数据是没有说服力的，需要有大量的真实数据用于验证；
- 资源挂了、加载出现异常，不能总靠用户投诉才后知后觉，需要主动报警。

  一次性能重构，在千兆网速和万元设备的条件下，页面加载时间的提升可能只有 0.1%，但是这样的数（土）据（豪）不具备代表性。网络环境、硬件设备千差万别，对于中低端设备而言，性能提升的主观体验更为明显，对应的数据变化更具备代表性。

  不少项目都会把资源上传到 CDN。而 CDN 部分节点出现问题的时候，一般不能精准的告知“某某，你的 xx 资源挂了”，因此需要我们主动监控。

  根据谷歌数据显示，当页面加载超过 10s 时，用户会感到绝望，通常会离开当前页面，并且很可能不再回来。

用什么监控

关于前端性能指标，W3C 定义了强大的 Performance API，其中又包括了 High Resolution Time 、 Frame Timing 、 Navigation Timing 、Performance Timeline 、Resource Timing 、 User Timing 等诸多具体标准。

本文主要涉及 Navigation Timing 以及 Resource Timing。截至到 2018 年中旬，各大主流浏览器均已完成了基础实现。


Navigation Timing API - REC
API for accessing timing information related to navigation and elements.
Global 94.05%
unprefixed: 94.01%
Current aligned
Usage relative
Date relative
Show all
IE
Edge
Firefox
Chrome
Safari
iOS Safari
Opera Mini
Chrome for Android
UC Browser for Android
Samsung Internet
49
60
66
10.3
11.2
4
6.2
11
17
61
67
11.1
11.4
all
67
11.8
7.2
18
62
63
68
69
70
12
TP
12
Notes
Known issues (0)
Resources (7)
Feedback
Removed in iOS 8.1 due to poor performance.

Performance API 功能众多，其中一项，就是将页面自身以及页面中各个资源的性能表现（时间细节）记录了下来。而我们要做的就是查询和使用。

读者可以直接在浏览器控制台中输入 performance ，查看相关 API。

接下来，我们将使用浏览器提供的 window.performance 对象（Performance API 的具体实现），来实现一个简易的前端性能监控工具。

5 分钟撸一个前端性能监控工具

第一行代码

将工具命名为 pMonitor，含义是 performance monitor。

```
JavaScript
```

监控哪些指标

既然是“5 分钟实现一个 xxx”系列，那么就要有取舍。因此，本文只挑选了最为重要的两个指标进行监控：

- 页面加载时间
- 资源请求时间

看了看时间，已经过去了 4 分钟，小编表示情绪稳定，没有一丝波动。

页面加载

有关页面加载的性能指标，可以在 Navigation Timing 中找到。Navigation Timing 包括了从请求页面起，到页面完成加载为止，各个环节的时间明细。

可以通过以下方式获取 Navigation Timing 的具体内容：

```JavaScript
const navTimes = performance.getEntriesByType('navigation')
```

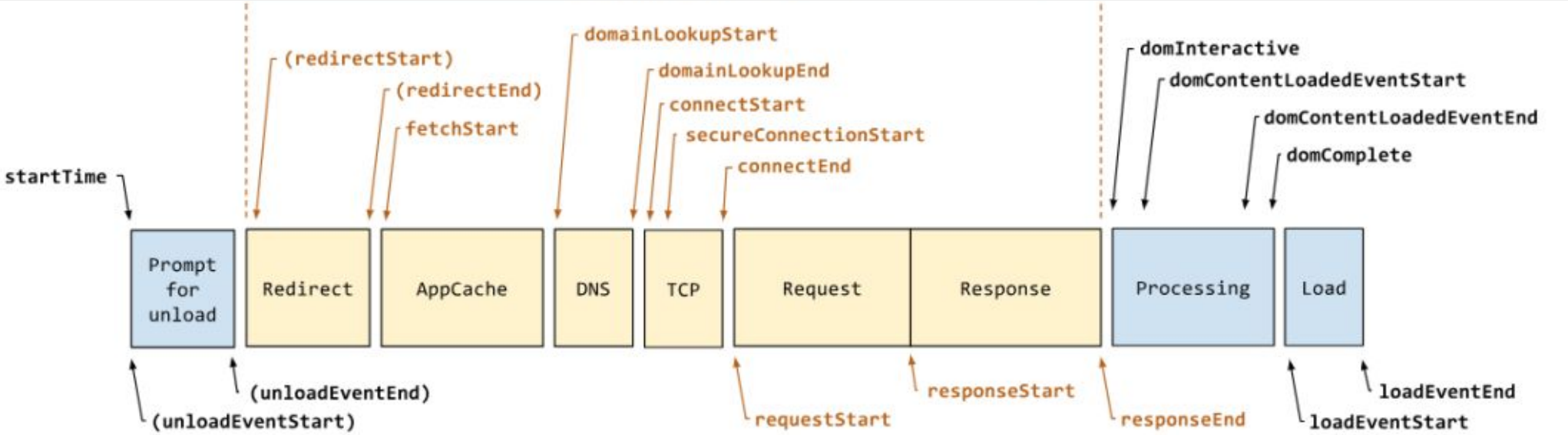
> getEntriesByType 是我们获取性能数据的一种方式。performance 还提供了 getEntries 以及 getEntriesByName 等其他方式，由于“时间限制”，具体区别不在此赘述，各位看官可以移步到此：https://www.w3.org/TR/performance-timeline-2/#dom-performance。

返回结果是一个数组，其中的元素结构如下所示：

```JavaScript
{
"connectEnd": 64.15495765894057,
"connectStart": 64.15495765894057,
```

```
"domComplete": 2002.5385066728431,
"domContentLoadedEventEnd": 2001.7384263440083,
"domContentLoadedEventStart": 2001.2386167400286,
"domInteractive": 1988.638474368076,
"domLoading": 271.75174283737226,
"duration": 2002.9385468372606,
"entryType": "navigation",
"fetchStart": 64.15495765894057,
"loadEventEnd": 2002.9385468372606,
"loadEventStart": 2002.7383663540235,
"name": "document",
"navigationStart": 0,
"redirectCount": 0,
"redirectEnd": 0,
"redirectStart": 0,
"requestStart": 65.28225608537441,
"responseEnd": 1988.283025689508,
"responseStart": 271.75174283737226,
"startTime": 0,
"type": "navigate",
"unloadEventEnd": 0,
"unloadEventStart": 0,
"workerStart": 0.9636893776343863
}
```


关于各个字段的时间含义，Navigation Timing Level 2 给出了详细说明：



不难看出，细节满满。因此，能够计算的内容十分丰富，例如 DNS 查询时间，TLS 握手时间等等。可以说，只有想不到，没有做不到～

既然我们关注的是页面加载，那自然要读取 domComplete:

```JavaScript
const [{ domComplete }] = performance.getEntriesByType('navigation')
```

定义个方法，获取 domComplete：

```JavaScript
pMonitor.getLoadTime = () => {
    const [{ domComplete }] = performance.getEntriesByType('navigation')
    return domComplete
}
```

到此，我们获得了准确的页面加载时间。

资源加载

答案是肯定的，其名为 Resource Timing。它包含了页面中各个资源从发送请求起，到完成加载为止，各个环节的时间细节，和 Navigation Timing 十分类似。

获取资源加载时间的关键字为 ‘resource’，具体方式如下：

```JavaScript
performance.getEntriesByType('resource')
```

不难联想，返回结果通常是一个很长的数组，因为包含了页面上所有资源的加载信息。

每条信息的具体结构为：

```JavaScript
{

"connectEnd": 462.95008929525244,

"connectStart": 462.95008929525244,

"domainLookupEnd": 462.95008929525244,

"domainLookupStart": 462.95008929525244,

"duration": 0.9620853673520173,

"entryType": "resource",

"fetchStart": 462.95008929525244,

"initiatorType": "img",

"name": "https://cn.bing.com/sa/simg/SharedSpriteDesktopRewards_022118.png",

"nextHopProtocol": "",

"redirectEnd": 0,

"redirectStart": 0,

"requestStart": 463.91217466260445,

"responseEnd": 463.91217466260445,
```

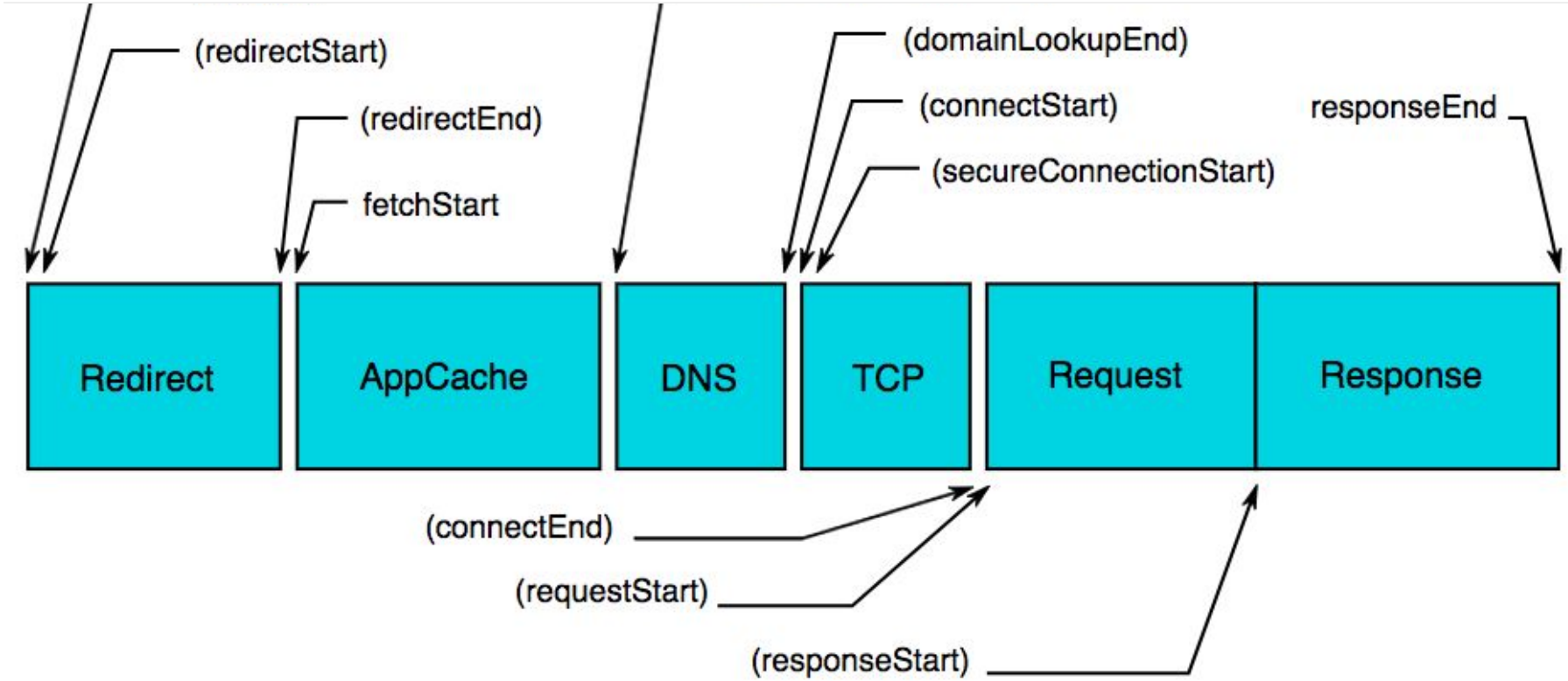
```
32
33 "startTime": 462.95008929525244,
34
35 "workerStart": 0
36
37 }
```

以上为 2018 年 7 月 7 日，在 https://cn.bing.com 下搜索 test 时，performance.getEntriesByType（“resource”）返回的第二条结果。

我们关注的是资源加载的耗时情况，可以通过如下形式获得：

```JavaScript
1 const [{ startTime, responseEnd }] = performance.getEntriesByType('resource');
2 const loadTime = responseEnd - startTime;
```

同 Navigation Timing 相似，关于 startTime 、 fetchStart、connectStart 和 requestStart 的区别， Resource Timing Level 2 给出了详细说明：



并非所有的资源加载时间都需要关注，重点还是加载过慢的部分。

出于简化考虑，定义 10s 为超时界限，那么获取超时资源的方法如下：

JavaScript

```
const SEC = 1000
const TIMEOUT = 10 * SEC
const setTime = (limit = TIMEOUT) => time => time >= limit
const isTimeout = setTime()
const getLoadTime = ({ startTime, responseEnd }) => responseEnd - startTime
const getName = ({ name }) => name
const resourceTimes = performance.getEntriesByType('resource')
const getTimeoutRes = resourceTimes
  .filter(item => isTimeout(getLoadTime(item)))
  .map(getName)
```

简单封装一下：

JavaScript

```
const SEC = 1000
const TIMEOUT = 10 * SEC
const setTime = (limit = TIMEOUT) => time => time >= limit
const getLoadTime = ({ requestStart, responseEnd }) => responseEnd - requestStart
const getName = ({ name }) => name
pMonitor.getTimeoutRes = (limit = TIMEOUT) => {
  const isTimeout = setTime(limit)
  const resourceTimes = performance.getEntriesByType('resource')
  return resourceTimes.filter(item => isTimeout(getLoadTime(item))).map(getName)
}
```

上报数据

获取数据之后，需要向服务端上报：

JavaScript

```
// 生成表单数据
const convert2FormData = (data = {}) =>
    Object.entries(data).reduce((last, [key, value]) => {
        if (Array.isArray(value)) {
            return value.reduce((lastResult, item) => {
                lastResult.append(`${key}[]`, item)
                return lastResult
            }, last)
        }
        last.append(key, value)
        return last
    }, new FormData())

// 拼接 GET 时的url
const makeItStr = (data = {}) =>
    Object.entries(data)
        .map(([k, v]) => `${k}=${v}`)
        .join('&')

// 上报数据
pMonitor.log = (url, data = {}, type = 'POST') => {
    const method = type.toLowerCase()
    const urlToUse = method === 'get' ? `${url}?${makeItStr(data)}` : url
```

```
    const option = {
        method,
        ...body
    }
    fetch(urlToUse, option).catch(e => console.log(e))
}
```

回过头来初始化

数据上传的 url、超时时间等细节，因项目而异，所以需要提供一个初始化的方法：

JavaScript

```
// 缓存配置
let config = {}
/**
 * @param {object} option
 * @param {string} option.url 页面加载数据的上报地址
 * @param {string} option.timeoutUrl 页面资源超时的上报地址
 * @param {string=} [option.method='POST'] 请求方式
 * @param {number=} [option.timeout=10000] 超时时间
 */
pMonitor.init = option => {
const { url, timeoutUrl, method = 'POST', timeout = 10000 } = option
  config = {
    url,
    timeoutUrl,
    method,
    timeout
  }
  // 绑定事件 用于触发上报数据

  pMonitor.bindEvent()
}
```

何时触发

性能监控只是辅助功能，不应阻塞页面加载，因此只有当页面完成加载后，我们才进行数据获取和上报（实际上，页面加载完成前也获取不到必要信息）：

JavaScript

```
// 封装一个上报两项核心数据的方法
```

```
    const domComplete = pMonitor.getLoadTime()
    const timeoutRes = pMonitor.getTimeoutRes(config.timeout)
    // 上报页面加载时间
    pMonitor.log(url, {domeComplete}, method)
    if (timeoutRes.length) {
        pMonitor.log(
            timeoutUrl,
            {timeRes},
            method
        )
    }
}
// 事件绑定
pMonitor.bindEvent = () => {
    const oldOnload = window.onload
    window.onload = e => {
        if (oldOnload && typeof oldOnload === 'function') {
            oldOnload(e)
        }
        // 尽量不影响页面主线程
        if (window.requestIdleCallback) {
            window.requestIdleCallback(pMonitor.logPackage)
        } else {
            setTimeout(pMonitor.logPackage)
        }
    }
}
```

汇总

到此为止，一个完整的前端性能监控工具就完成了～全部代码如下：

JavaScript

```
const base = {
    log() {
    },
    logPackage() {
    },
    getLoadTime() {
    },
    getTimeoutRes() {
    },
    bindEvent() {
    },
```

```
}
const pm = (function () {
    // 向前兼容
    if (!window.performance) return base
    const pMonitor = {...base}
    let config = {}
    const SEC = 1000
    const TIMEOUT = 10 * SEC
    const setTime = (limit = TIMEOUT) => time => time >= limit
    const getLoadTime = ({startTime, responseEnd}) => responseEnd - startTime
    const getName = ({name}) => name
    // 生成表单数据
    const convert2FormData = (data = {}) =>
        Object.entries(data).reduce((last, [key, value]) => {
            if (Array.isArray(value)) {
                return value.reduce((lastResult, item) => {
                    lastResult.append(`${key}[]`, item)
                    return lastResult
                }, last)
            }
            last.append(key, value)
            return last
        }, new FormData())
    // 拼接 GET 时的url
    const makeItStr = (data = {}) =>
        Object.entries(data)
            .map(([k, v]) => `${k}=${v}`)
            .join('&')
    pMonitor.getLoadTime = () => {
        const [{domComplete}] = performance.getEntriesByType('navigation')
        return domComplete
    }
    pMonitor.getTimeoutRes = (limit = TIMEOUT) => {
        const isTimeout = setTime(limit)
        const resourceTimes = performance.getEntriesByType('resource')
        return resourceTimes
            .filter(item => isTimeout(getLoadTime(item)))
            .map(getName)
    }
    // 上报数据
    pMonitor.log = (url, data = {}, type = 'POST') => {
        const method = type.toLowerCase()
        const urlToUse = method === 'get' ? `${url}?${makeItStr(data)}` : url
        const body = method === 'get' ? {} : {body: convert2FormData(data)}
        const init = {
            method,
            ...body
```

```
    }
    // 封装一个上报两项核心数据的方法
    pMonitor.logPackage = () => {
        const {url, timeoutUrl, method} = config
        const domComplete = pMonitor.getLoadTime()
        const timeoutRes = pMonitor.getTimeoutRes(config.timeout)
        // 上报页面加载时间
        pMonitor.log(url, {domeComplete}, method)
        if (timeoutRes.length) {
            pMonitor.log(
                timeoutUrl,
                {timeoutRes},
                method
            )
        }
    }
    // 事件绑定
    pMonitor.bindEvent = () => {
        const oldOnload = window.onload
        window.onload = e => {
            if (oldOnload && typeof oldOnload === 'function') {
                oldOnload(e)
            }
            // 尽量不影响页面主线程
            if (window.requestIdleCallback) {
                window.requestIdleCallback(pMonitor.logPackage)
            } else {
                setTimeout(pMonitor.logPackage)
            }
        }
    }
    /**
     * @param {object} option
     * @param {string} option.url 页面加载数据的上报地址
     * @param {string} option.timeoutUrl 页面资源超时的上报地址
     * @param {string=} [option.method='POST'] 请求方式
     * @param {number=} [option.timeout=10000]
     */
    pMonitor.init = option => {
        const {url, timeoutUrl, method = 'POST', timeout = 10000} = option
        config = {
            url,
            timeoutUrl,
            method,
            timeout
        }
        // 绑定事件 用于触发上报数据
```

```
    return pMonitor
})()
export default pm
```

如何？是不是不复杂？甚至有点简单～

调用

如果想追（吹）求（毛）极（求）致（疵）的话，在页面加载时，监测工具不应该占用主线程的 JavaScript 解析时间。因此，最好在页面触发 onload 事件后，采用异步加载的方式：

JavaScript

```
// 在项目的入口文件的底部
const log = async () => {
    const pMonitor = await import('/path/to/pMonitor.js')
    pMonitor.init({ url: 'xxx', timeoutUrl: 'xxxx' })
    pMonitor.logPackage()
    // 可以进一步将 bindEvent 方法从源码中删除
}
const oldOnload = window.onload
window.onload = e => {
    if (oldOnload && typeof oldOnload === 'string') {
        oldOnload(e)
    }
// 尽量不影响页面主线程
    if (window.requestIdleCallback) {
        window.requestIdleCallback(log)
    } else {
        setTimeout(log)
    }
}
```

跨域等请求问题

工具在数据上报时，没有考虑跨域问题，也没有处理 GET 和 POST 同时存在的情况。

> 5 分钟还要什么自行车！

如有需求，可以自行覆盖 pMonitor.logPackage 方法，改为动态创建 <form/> 和 <iframe/> ，或者使用更为常见的图片打点方式～

这个还是需要服务端配合的嘛[认真脸.jpg]。

既可以是每个项目对应不同的上报 url，也可以是统一的一套 url，项目分配唯一 id 作为区分。

当超时次数在规定时间内超过约定的阈值时，邮件/短信通知开发人员。

细粒度

现在仅仅针对超时资源进行了简单统计，但是没有上报具体的超时原因（DNS？TCP？request？ response？），这就留给读者去优化了，动手试试吧 ~

下一步

本文介绍了关于页面加载方面的性能监控， 此外，JavaScript 代码的解析 + 执行，也是制约页面首屏渲染快慢的重要因素（特别是单页面应用）。下一话，小编将带领大家 进一步探索 Performance Timeline Level 2， 实现更多对于 JavaScript 运行时的性能监控，敬请期待～

参考资料

- https://w3c.github.io/navigation-timing
- https://www.w3.org/TR/resource-timing-2
- https://www.w3.org/TR/performance-timeline-2
- https://developers.google.com/web/fundamentals/performance/rail