XCel 项目总结 – Electron 与 Vue 的性能优化

by Jc (https://github.com/JChehe) on 2016-11-15

XCEL 是由京东用户体验设计部凹凸实验室推出的一个 Excel 数据清洗工具，其通过可视化的方式让用户轻松地对 Excel 数据进行筛选。

XCEL 基于 Electron 和 Vue 2.x，它不仅跨平台（windows 7+、Mac 和 Linux），而且充分利用 Electron 多进程任务处理等功能，使其性能优异。

落地页：https://xcel.aotu.io/ ✨ ✨ ✨
项目地址：https://github.com/o2team/xcel (https://github.com/o2team/xcel) ✨ ✨ ✨

项目背景

用户研究的定量研究和轻量级数据处理中，均需对数据进行清洗处理，以剔除异常数据，保证数据结果的信度和效度。目前因调研数据和轻量级数据的多变性，对轻量级数据清洗往往采取人工清洗，缺少统一、标准的清洗流程，但对于调研和轻量级的数据往往是需要保证数据稳定性的，因此，在对数据进行清洗时最好有标准化的清洗方式。

特性一览

- 基于 Electron 研发并打包成为原生应用，用户体验良好；
- 可视化操作 Excel 数据，支持文件的导入导出；
- 拥有单列运算逻辑、多列运算逻辑和双列范围逻辑三种筛选方式，并且可通过“且”、“或”和“编组”的方式任意组合。

思路与实现

基于用研组的需求，利用 Electron 和 Vue 的特性对该工具进行开发。

技术选型

- Electron：桌面端跨平台框架，为 Web 提供了原生接口的权限。打包后的程序兼容 Windows 7 及以上、Mac、Linux 的 32 / 64 位系统。详情>> (http://electron.atom.io/)
- Vue 全家桶：Vue 拥有数据驱动视图的特性，适合重数据交互的应用。详情>> (http://vuejs.org/)
- js-xlsx：兼容各种电子表格格式的解析器和生成器。纯 JavaScript 实现，适用于 Node.js 和 Web 前端。详情>> (https://github.com/SheetJS/js-xlsx)

实现思路

1. 通过 js-xlsx 将 Excel 文件解析为 JSON 数据
2. 根据筛选条件对 JSON 数据进行筛选过滤
3. 将过滤后的 JSON 数据转换成 js-xlsx 指定的数据结构
4. 利用 js-xlsx 对转换后的数据生成 Excel 文件

纸上得来终觉浅，绝知此事要躬行

相关技术

如果对某项技术比较熟悉，则可略读/跳过。

Electron

Electron 是什么?

Electron 是一个可以用 JavaScript、HTML 和 CSS 构建桌面应用程序的库。这些应用程序能打包到 Mac、Windows 和 Linux 系统上运行，也能上架到 Mac 和 Windows 的 App Store。

- JavaScript、HTML 和 CSS 都是 Web 语言，它们是组成网站的一部分，浏览器（如 Chrome）懂得如何将这些代码转为可视化图像。
- Electron 是一个库：Electron 对底层代码进行抽象和封装，让开发者能在此之上构建项目。

为什么它如此重要?

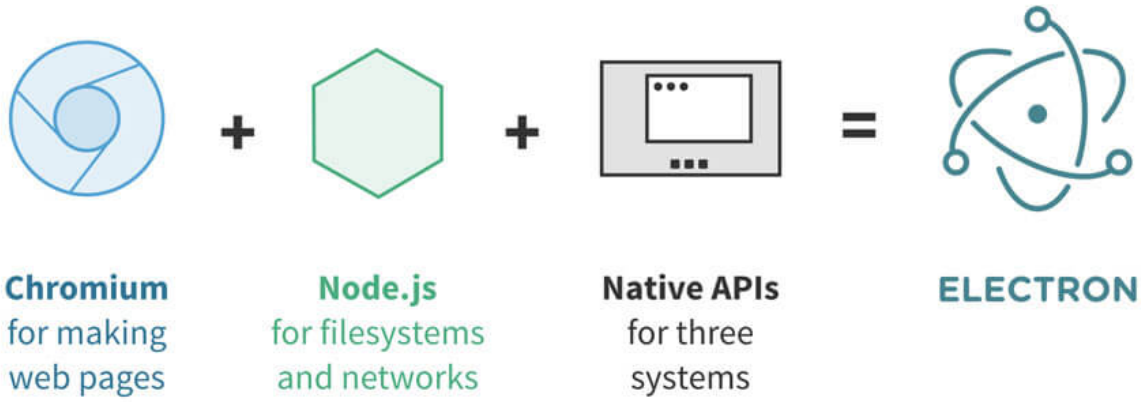
通常来说，每个操作系统的桌面应用都由各自的原生语言进行编写，这意味着需要 3 个团队分别为该应用编写相应版本。而 Electron 则允许你用 Web 语言编写一次即可。

- 原生（操作系统）语言：用于开发主流操作系统应用的原生语言的对应关系（大多数情况下）：Mac 对应 Objective C、Linux 对应 C、Windows 对应 C++。

它由什么组成?

Electron 结合了 Chromium、Node.js 和用于调用操作系统本地功能的 API（如打开文件窗口、通知、图标等）。

- Chromium：Google 创造的一个开源库，并用于 Google 的浏览器 Chrome。
- Node.js（Node）：一个在服务器运行 JavaScript 的运行时（runtime），它拥有访问文件系统和网络权限（你的电脑也可以是一台服务器！）。



开发体验如何?

基于 Electron 的开发就像在开发网页，而且能够无缝地 使用 Node。或者说：在构建一个 Node 应用的同时，通过 HTML 和 CSS 构建界面。另外，你只需为一个浏览器（最新的 Chrome）进行设计（即无需考虑兼容性等）。

- 使用 Node：这还不是全部！除了完整的 Node API，你还可以使用托管在 npm 上超过 350,000 个的模块。
- 一个浏览器：并非所有浏览器都提供一致的样式，Web 设计师和开发者经常因此而不得不花费更多的精力，让网站在不同浏览器上表现一致。
- 最新的 Chrome：可使用超过 90% 的 ES2015 特性和其它很酷的特性（如 CSS 变量）。

两个进程（重点）

Electron 有两种进程：『主进程』和『渲染进程』。部分模块只能在两者之一上运行，而有些则无限制。主进程更多地充当幕后角色，而渲染进程则是应用程序的各个窗口。

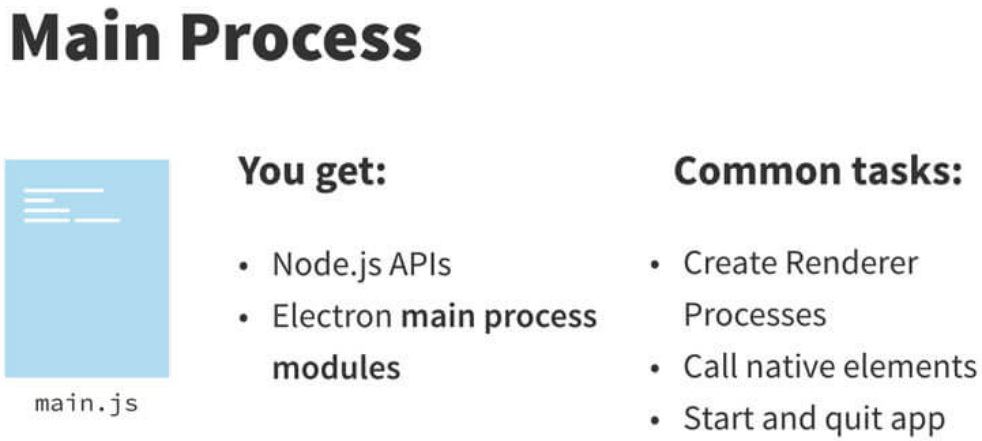
> 注：可通过任务管理器（PC）/活动监视器（Mac）查看进程的相关信息。

- 模块：Electron 的 API 是根据它们的用途进行分组。例如： `dialog` 模块拥有所有原生 dialog 的 API，如打开文件、保存文件和警告等弹窗。

主进程

主进程，通常是一个命名为 `main.js` 的文件，该文件是每个 Electron 应用的入口。它控制了应用的生命周期（从打开到关闭）。它既能调用原生元素，也能创建新的（多个）渲染进程。另外，Node API 是内置其中的。

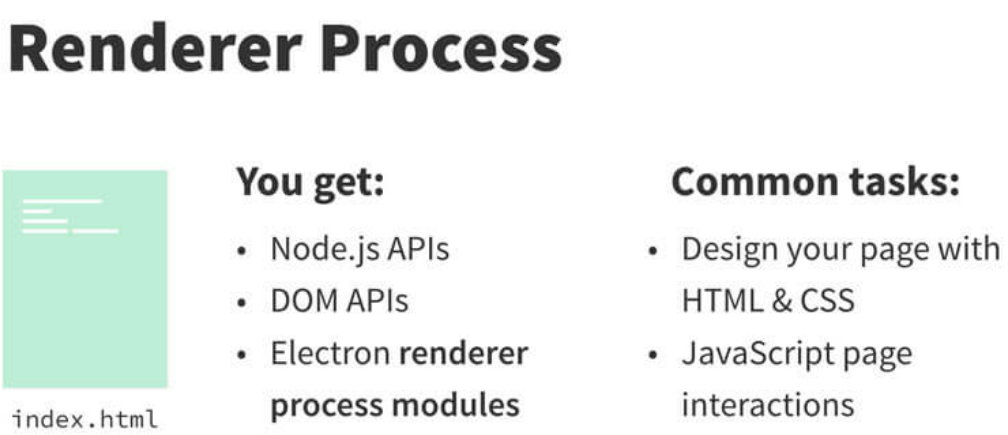
- 调用原生元素：打开 diglog 和其它操作系统的交互均是资源密集型操作（注：出于安全考虑，渲染进程是不能直接访问本地资源的），因此都需要在主进程完成。



渲染进程

渲染进程是应用的一个浏览器窗口。与主进程不同，它能存在多个（注：一个 Electron 应用只能存在一个主进程）并且相互独立（它也能是隐藏的）。主窗口通常被命名为 `index.html`。它们就像典型的 HTML 文件，但 Electron 赋予了它们完整的 Node API。因此，这也是它与浏览器的区别。

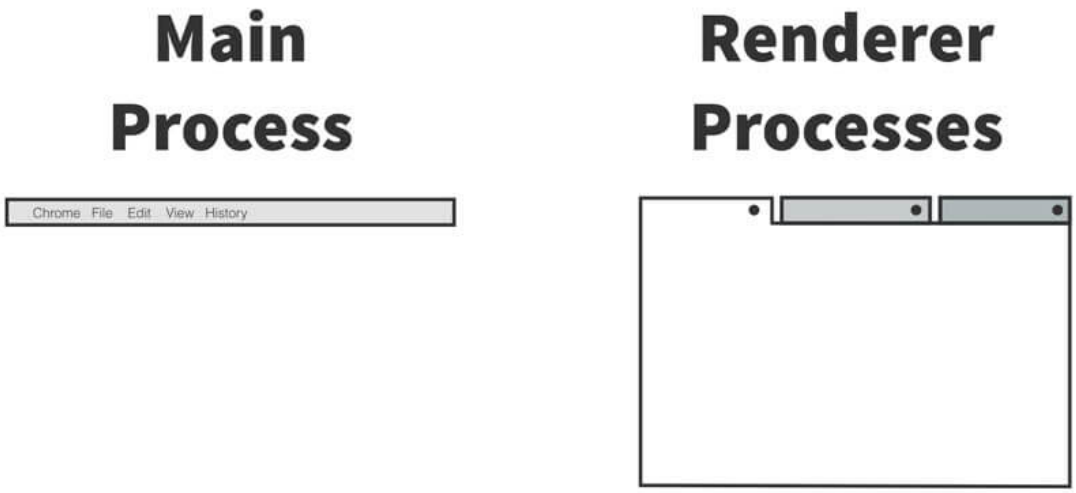
- 相互独立：每个渲染进程都是独立的，这意味着某个渲染进程的崩溃，也不会影响其余渲染进程。
- 隐藏：可隐藏窗口，然后让其在背后运行代码（👍）。



把它们想象成这样

Chrome（或其他浏览器）的每个标签页（tab）及其页面，就好比 Electron 中的一个单独渲染进程。即使关闭所有标签页，Chrome 依然存在。这好比 Electron 的主进程，能打开新的窗口或关闭这个应用。

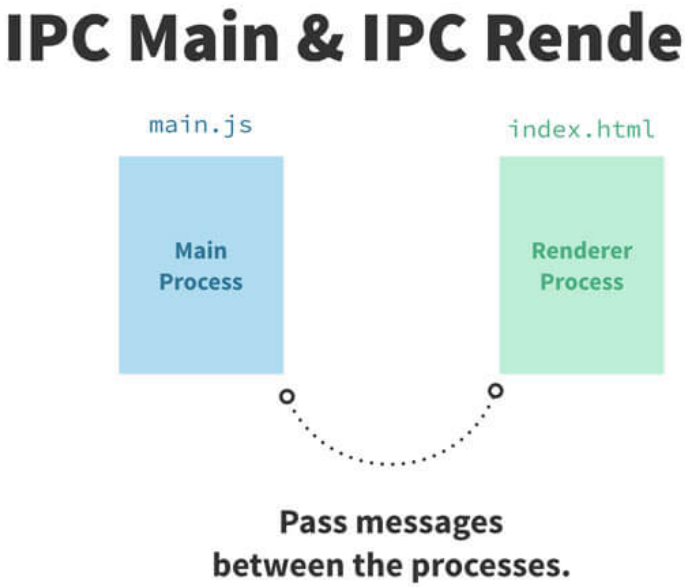
> 注：在 Chrome 浏览器中，一个标签页（tab）中的页面（即除了浏览器本身部分，如搜索框、工具栏等）就是一个渲染进程。



相互通讯

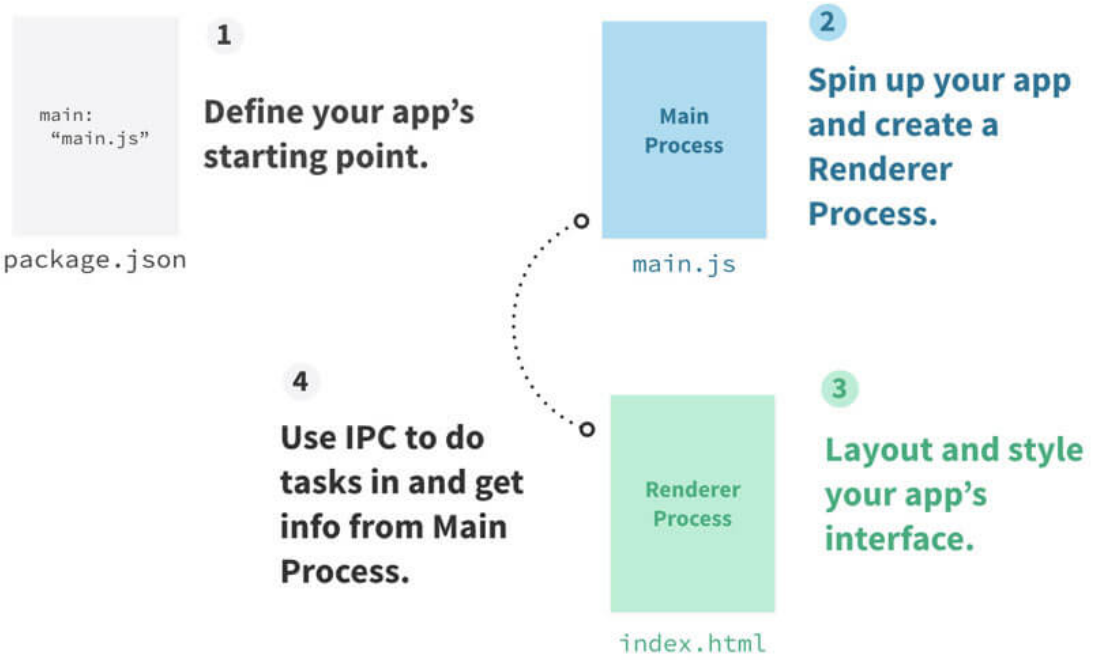
由于主进程和渲染进程各自负责不同的任务，而对于需要协同完成的任务，它们需要相互通讯。IPC就为此而生，它提供了进程间的通讯。但它只能在主进程与渲染进程之间传递信息（即渲染进程之间不能进行直接通讯）。

- IPC：主进程和渲染进程各自拥有一个 IPC 模块。



汇成一句话

Electron 应用就像 Node 应用，它也依赖一个 `package.json` 文件。该文件定义了哪个文件作为主进程，并因此让 Electron 知道从何启动应用。然后主进程能创建渲染进程，并能使用 IPC 让两者间进行消息传递。



至此，Electron 的基础部分介绍完毕。该部分是基于笔者之前翻译的一篇文章《Essential Electron》(http://jlord.us/essential-electron/)，译文可点击 这里(https://segmentfault.com/a/1190000007503495)。

Vue 全家桶

该工具使用了 Vue、Vuex、Vuex-router。在工具基本定型阶段，由 1.x 升级到了 2.x。

为什么选择 Vue

对于笔者来说：

- 简单易用，一般使用只需看官方文档。
- 数据驱动视图，所以基本不用操作 DOM 了。
- 框架的存在是为了帮助我们应对复杂度。
- 全家桶的好处是：对于一般场景，我们就不需要考虑用哪些个库（插件）。

Vue 1.x -> Vue 2.0 的版本迁移用 vue-migration-helper (https://github.com/vuejs/vue-migration-helper) 即可分析出大部分需要更改的地方。

网上已有很多关于 Vue 的教程，故在此不再赘述。至此，Vue 部分介绍完毕。

---

js-xlsx

该库支持各种电子表格格式的解析与生成。它由 JavaScript 实现，适用于前端和 Node。详情 >> (https://github.com/SheetJS/js-xlsx)

目前支持读入的格式有（不断更新）：

- Excel 2007+ XML Formats (XLSX/XLSM)
- Excel 2007+ Binary Format (XLSB)
- Excel 2003-2004 XML Format (XML "SpreadsheetML")
- Excel 97-2004 (XLS BIFF8)
- Excel 5.0/95 (XLS BIFF5)
- OpenDocument Spreadsheet (ODS)

支持写出的格式有：

- XLSX
- CSV (and general DSV)
- JSON and JS objects (various styles)

目前该库提供的 `sheet_to_json` 方法能将读入的 Excel 数据转为 JSON 格式。而对于导出操作，我们需要为 js-xlsx 提供指定的 JSON 格式。

更多关于 Excel 在 JavaScript 中处理的知识可查看凹凸实验室的《Node读写Excel文件探究实践》。但该文章存在两处问题（均在 js-xlsx 实战的导出表格部分）：

1. 生成头部时，Excel 的列信息简单地通过 `String.fromCharCode(65+j)` 生成。当列大于 26 时会出现问题。这个问题会在后面章节中给出解决方案；
2. 转换成 worksheet 需要的结构处，出现逻辑性错误，并且会导致严重的性能问题。逻辑问题在此不讲述，我们看看性能问题：
   随着 ECMAScript 的不断更新，JavaScript 变得更加强大和易用。尽管如此，我们还是要做到『物尽所用』，而不要『大材小用』，否则可能会得到"反效果"。这里导致性能问题的正是 Object.assign() (https://developer.mozilla.org/zh-

CN/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) 方法，该方法可以把任意多个源对象的可枚举属性拷贝至目标对象，并返回目标对象。由于该方法自身的实现机制，会在此案例中产生大量的冗余操作。在该案例中，单元格信息是唯一的，所以直接通过 forEach 为一个空对象赋值即可。提升 N 倍性能的同时，也把逻辑性错误解决了。

原来的：

```
var result = 某数组.reduce((prev, next) => Object.assign({}, prev, {[next.position]: {v: ne
```

改为：

```
var result = 某数组.forEach((v, i) => data[v.position]= {v: v.v})
```

实践是检验真理的唯一标准

在理解上述知识后，下面就谈谈在该项目实践中总结出来的技巧、难点和重点。

CSS、JavaScript 和 Electron 相关的知识和技巧

高亮 table 的列

Excel 单元格采用 `table` 标签展示。在 Excel 中，被选中的单元格会高亮相应的『行』和『列』，以提醒用户。在该应用中也有做相应的处理，横向高亮采用 `tr:hover` 实现，而纵向呢？这里所采用的一个技巧是：

假设 HTML 结构如下：

```
div.container
  table
    tr
      td
```

CSS 代码如下：

```
.container { overflow:hidden; }
td { position: relative; }
td:hover::after {
  position: absolute;
  left: 0;
```

```
  right: 0;
  top: -1个亿px; // 小目标达成，不过是负的😭
  bottom: -1个亿px;
  z-index: -1; // 避免遮住自身和同列 td 的内容、border 等
}
```

斜分割线

如图： 

分割线可以通过 `::after/::before` 伪类元素实现一条直线，然后通过 `transform:rotate();` 旋转特定角度实现。但这种实现的一个问题是：由于宽度是不定的，因此需要通过 JavaScript 运算才能得到准确的对角分割线。

因此，这里可以通过 CSS 线性渐变 `linear-gradient(to top right, transparent, transparent calc(50% - .5px), #d3d6db calc(50% - .5px), #d3d6db calc(50% + .5px), transparent calc(50% + .5px))` 实现。无论宽高如何变，依然妥妥地自适应。

Excel 的列转换

- Excel 的列需要用『字母』表示，但不能简单地通过 String.fromCharCode() (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/fromCharCode) 实现，因为当超出 26 列时就会产生问题（如：第 27 列，`String.fromCharCode(65+26)` 得到的是 `[`，而不是 AA）。因此，这需要通过『十进制和 26 进制转换』算法来实现。

```
// 将传入的自然数转换为26进制表示。映射关系：[0-25] -> [A-Z]。
function getCharCol (n) {
  let s = ''
  let m = 0
  while (n >= 0) {
    m = (n % 26) + 1
    s = String.fromCharCode(m + 64) + s
    n = (n - m) / 26
  }
  return s
}
```

```
// 将传入的26进制转换为自然数。映射关系：[A-Z] ->[0-25]。
function getNumCol (s) {
  if (!s) return 0
```

```
  let n = 0
  for (let i = s.length - 1, j = 1; i >= 0; i--, j *= 26) {
    const c = s[i].toUpperCase()
    if (c < 'A' || c > 'Z') return 0
    n += (c.charCodeAt() - 64) * j
  }
  return n - 1
}
```

为 DOM 的 File 对象增加了 path 属性

Electron 为 File 对象额外增了 path 属性，该属性可得到文件在文件系统上的真实路径。因此，你可以利用 Node 为所欲为😈 。应用场景有：拖拽文件后，通过 Node 提供的 File API 读取文件等。

支持常见的编辑功能，如粘贴和复制

Electron 应用在 MacOS 中默认不支持『复制』『粘贴』等常见编辑功能，因此需要为 MacOS 显式地设置复制粘贴等编辑功能的菜单栏，并为此设置相应的快捷键。

```
// darwin 就是 MacOS
if (process.platform === 'darwin') {
    var template = [{
      label: 'FromScratch',
      submenu: [{
        label: 'Quit',
        accelerator: 'CmdOrCtrl+Q',
        click: function() { app.quit(); }
      }]
    }, {
      label: 'Edit',
      submenu: [{
        label: 'Undo',
        accelerator: 'CmdOrCtrl+Z',
        selector: 'undo:'
      }, {
        label: 'Redo',
        accelerator: 'Shift+CmdOrCtrl+Z',
        selector: 'redo:'
      }, {
        type: 'separator'
      }, {
        label: 'Cut',
        accelerator: 'CmdOrCtrl+X',
        selector: 'cut:'
      }, {
        label: 'Copy',
        accelerator: 'CmdOrCtrl+C',
        selector: 'copy:'
```

```
      }, {
        label: 'Paste',
        accelerator: 'CmdOrCtrl+V',
        selector: 'paste:'
      }, {
        label: 'Select All',
        accelerator: 'CmdOrCtrl+A',
        selector: 'selectAll:'
      }]
    }];
    var osxMenu = menu.buildFromTemplate(template);
    menu.setApplicationMenu(osxMenu);
}
```

更贴近原生应用

Electron 的一个缺点是：即使你的应用是一个简单的时钟，但它也不得不包含完整的基础设施（如 Chromium、Node 等）。因此，一般情况下，打包后的程序至少会达到几十兆（根据系统类型进行浮动）。当你的应用越复杂，就越可以忽略文件体积问题。

众所周知，页面的渲染难免会导致『白屏』，而且这里采用了 Vue 这类框架，情况就更加糟糕了。另外，Electron 应用也避免不了『先打开浏览器，再渲染页面』的步骤。下面提供几种方法来减轻这种情况，以让程序更贴近原生应用。

1. 指定 BrowserWindow 的背景颜色；
2. 先隐藏窗口，直到页面加载后再显示；
3. 保存窗口的尺寸和位置，以让程序下次被打开时，依然保留的同样大小和出现在同样的位置上。

对于第一点，若应用的背景不是纯白（`#fff`）的，那么可指定窗口的背景颜色与其一致，以避免渲染后的突变。

```
mainWindow = new BrowserWindow({
    title: 'XCel',
    backgroundColor: '#f5f5f5',
};
```

对于第二点，由于 Electron 本质是一个浏览器，需要加载非网页部分的资源。因此，我们可以先隐藏窗口。

```
var mainWindow = new BrowserWindow({
    title: 'ElectronApp',
    show: false,
};
```

等到渲染进程开始渲染页面的那一刻，在 `ready-to-show` 的回调函数中显示窗口。

```
mainWindow.on('ready-to-show', function() {
    mainWindow.show();
    mainWindow.focus();
});
```

对于第三点，笔者并没有实现，原因如下：

1. 用户一般是根据当时的情况对程序的尺寸和位置进行调整，即视情况而定。
2. 以上是我个人臆测，主要是我懒🐶 。

其实现方式，可参考《4 must-know tips for building cross platform Electron apps》(https://blog.avocode.com/blog/4-must-know-tips-for-building-cross-platform-electron-apps) 。

如何在渲染进程调用原生弹框?

在渲染进程中调用原本专属于主进程中的 API （如弹框）的方式有两种：

1. IPC 通讯模块：先在主进程通过 `ipcMain` 进行监听，然后在渲染进程通过 `ipcRenderer` 进行触发；
2. remote 模块：该模块为渲染进程和主进程之间提供了快捷的通讯方式。

对于第二种方式，在渲染进程中，运行以下代码即可：

```
const remote = require('electron').remote

remote.dialog.showMessageBox({
  type: 'question',
  buttons: ['不告诉你', '没有梦想'],
  defaultId: 0,
  title: 'XCel',
  message: '你的梦想是什么？'
}
```

自动更新

如果 Electron 应用没有提供自动更新功能，那么就意味着用户想体验新开发的功能或用上修复 Bug 后的新版本，只能靠用户自己主动地去官网下载，这无疑是糟糕的体验。Electron 提供的 autoUpdater (http://electron.atom.io/docs/api/auto-updater/) 模块可实现自动更新功能，该模块提供了第三方框架 Squirrel (https://github.com/Squirrel) 的接口，但 Electron 目前只内置了

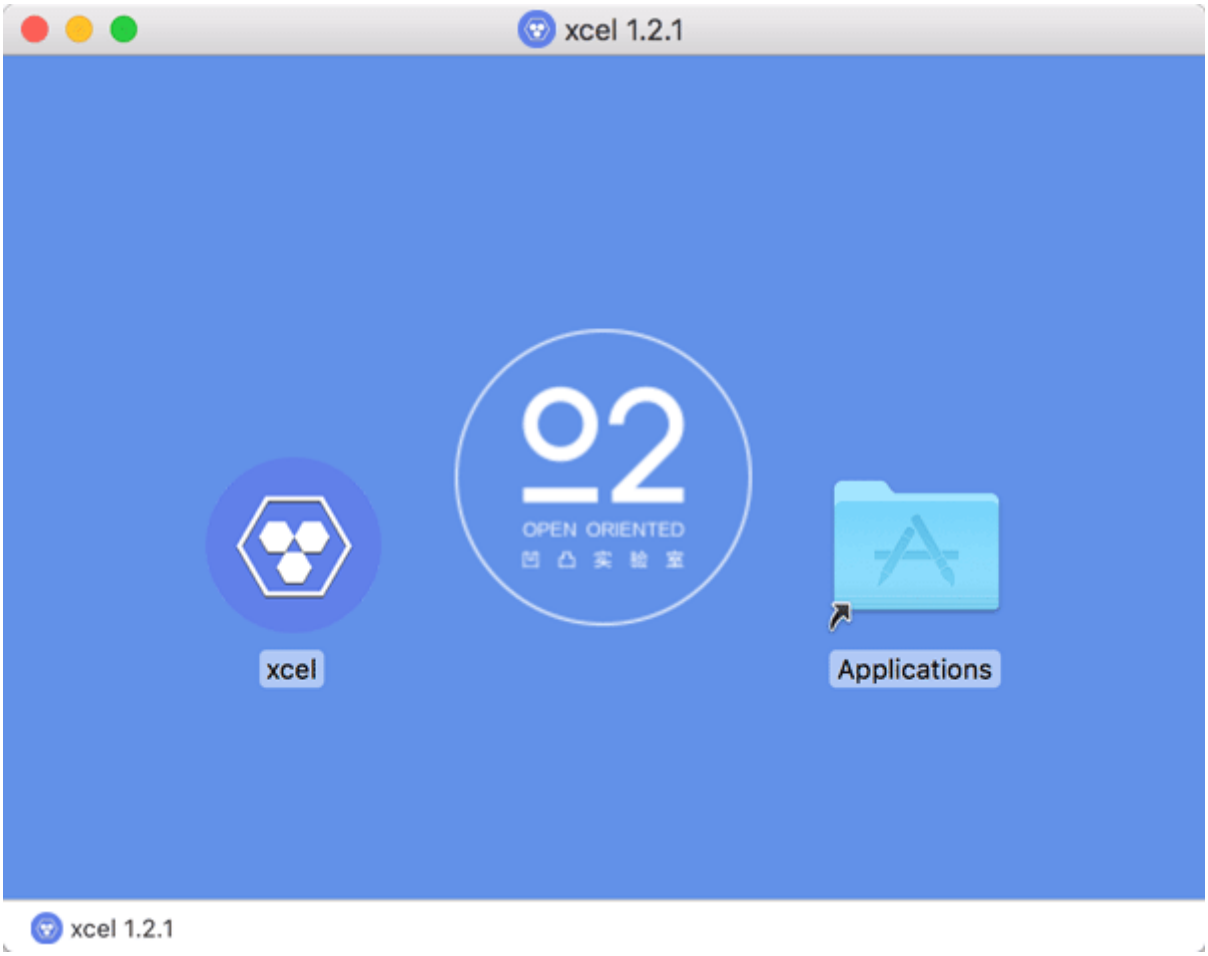
Squirrel.Mac (https://github.com/Squirrel/Squirrel.Mac)，且它与 Squirrel.Windows (https://github.com/Squirrel/Squirrel.Windows)（需要额外引入）的处理方式也不一致（在客户端与服务器端两方面）。因此如果对该模块不熟悉，处理起来会相对比较繁琐。具体可以参考笔者的另一篇译文《Electron 自动更新的完整教程（Windows 和 OSX）》(https://segmentfault.com/a/1190000007616641)。

> 目前 Electron 的 autoUpdater 模块不支持 Linux 系统。

另外，XCel 目前并没有采用 autoUpdater 模块实现自动更新功能，而是利用 Electron 的 DownloadItem (http://electron.atom.io/docs/api/download-item/) 模块实现，而服务器端则采用了 Nuts (https://nuts.gitbook.com/)。

为 Electron 应用生成 Windows 安装包

通过 electron-builder (https://www.electron.build/) 可直接生成常见的 MacOS 安装包，但它生成的 Windows 的安装包却略显简洁（默认选项时）。



Mac 常见的安装模式，将“左侧的应用图标”拖拽到“右侧的 Applications”即可

通过 electron-builder 生成的 Windows 安装包与我们在 Windows 上常见的软件安装界面不太一样，它没有安装向导和点击“下一步”的按钮，只有一个安装时的 gif 动画（默认的 gif 动画如下图，当然你也可以指定特定的 gif 动画），因此也就关闭了用户选择安装路径等权利。

Windows 安装时 默认显示的 gif 动画 (https://github.com/electron/windows-installer/blob/master/resources/install-spinner.gif)

如果你想为打包后的 Electron 应用（即通过 electron-packager/electron-builder 生成的，可直接运行的程序目录）生成拥有点击“下一步”按钮和可让用户指定安装路径的常见安装包，可以尝试 NSIS 程序，具体可看这篇教程 《[教學]只要10分鐘學會使用 NSIS 包裝您的桌面軟體—安裝程式打包。完全免費。》 (http://seesawworld.blogspot.com/2016/02/1-nsis.html) 。

注：electron-builder 也提供了生成安装包的配置项，具体查看>> (https://www.electron.build/configuration/nsis) 。

> NSIS（Nullsoft Scriptable Install System）是一个开源的 Windows 系统下安装程序制作程序。它提供了安装、卸载、系统设置、文件解压缩等功能。正如其名字所描述的那样，NSIS 是通过它的脚本语言来描述安装程序的行为和逻辑的。NSIS 的脚本语言和常见的编程语言有类似的结构和语法，但它是为安装程序这类应用所设计的。

至此，CSS、JavaScript 和 Electron 相关的知识和技巧部分阐述完毕。

性能优化

下面谈谈『性能优化』，这部分涉及到运行效率和内存占用量。
注：以下内容均基于 Excel 样例文件（数据量为：1913 行 x 180 列）得出的结论。

执行效率和渲染的优化

Vue 性能真的好?

Vue 一直标榜着自己性能优异，但当数据量上升到一定量级时（如 1913 x 180 ≈ 34 万个数据单元），会出现严重的性能问题（未做相应优化的前提下）。

如直接通过列表渲染 `v-for` 渲染数据时，会导致程序卡死。
答：通过查阅相关资料可得， `v-for` 在初次渲染时，需要对每个子项进行初始化（如数据绑定等操作，以便拥有更快的更新速度），这对于数据量较大时，无疑会造成严重的性能问题。

当时，我想到了两种解决思路：

1. Vue 是数据驱动视图的，对数据分段 push，即将一个庞大的任务分割为 N 份。
2. 自己拼接 HTML 字符串，再通过 innerHTML 一次性插入。

最终，我选择了第二条，理由是：

1. 性能最佳，因为每次执行数据过滤时，Vue 都要进行 diff，性能不佳。
2. 更符合当前应用的需求：纯展示且无需动画过渡等。
3. 实现更简单

将原本繁重的 DOM 操作（Vue）转换为 JavaScript 的拼接字符串后，性能得到了很大提升（不会导致程序卡死而渲染不出视图）。这种优化方式难道不就是 Vue、React 等框架解决的问题之一吗？只不过框架考虑的场景更广，有些地方需要我们自己根据实际情况进行优化而已。

> 在浏览器当中，JavaScript 的运算在现代的引擎中非常快，但 DOM 本身是非常缓慢的东西。当你调用原生 DOM API 的时候，浏览器需要在 JavaScript 引擎的语境下去接触原生的 DOM 的实现，这个过程有相当的性能损耗。所以，本质的考量是，要把耗费时间的操作尽量放在纯粹的计算中去做，保证最后计算出来的需要实际接触真实 DOM 的操作是最少的。 —— 《Vue 2.0——渐进式前端解决方案》(http://www.infoq.com/cn/articles/vue-2-progressive-front-end-solution)

当然，由于 JavaScript 天生单线程，即使执行数速度再快，也难免会导致页面有短暂的时间拒绝用户的输入。此时可通过 Web Worker 或其它方式解决，这也将是我们后续讲到的问题。

也有网友提供了优化大量列表的方法：https://clusterize.js.org/ (https://clusterize.js.org/) 。但在此案例中笔者并没有采用此方式。

强大的 GPU 加速

将拼接的字符串插入 DOM 后，出现了另外一个问题：滚动会很卡。猜想这是渲染问题，毕竟 34 万个单元格同时存在于界面中。

添加 `transform: translate3d(0, 0, 0) / translateZ(0)` 属性启动 GPU 渲染，即可解决这个渲染性能问题。再次感叹该属性的强大。🐂

后来，考虑到用户并不需要查看全部数据，只需展示部分数据让用户进行参考即可。我们对此只渲染前 30/50 行数据。这样即可提升用户体验，也能进一步优化性能。

记得关闭 Vuex 的严格模式

另外，由于自己学艺不精和粗心大意，忘记在生产环境关闭 Vuex 的『严格模式』。

Vuex 的严格模式要在生产环境中关闭，否则会对 state 树进行一个深观察 (deep watch)，产生不必要的性能损耗。也许在数据量少时，不会注意到这个问题。

还原当时的场景：导入 Excel 数据后，再进行交互（涉及 Vuex 的读写操作），需要等几秒才会响应，而直接通过纯 DOM 监听的事件则无此问题。由此，判断出是 Vuex 问题。

```
const store = new Vuex.Store({
  // ...
  strict: process.env.NODE_ENV !== 'production'
})
```

多进程！！！

前面说道，JavaScript 天生单线程，即使再快，对于数据量较大时，也会出现拒绝响应的问题。因此需要 Web Worker 或类似的方案去解决。

在这里我不选择 Web worker 的原因有如下几点：

1. 有其它更好的替代方案：一个主进程能创建多个渲染进程，通过 IPC 即可进行数据交互；
2. Electron 不支持 Web Worker！（当然，可能会在新版本支持，最新信息请关注官方）

Electron 作者在 2014.11.7 在《state of web worker support?》 issue 中回复了以下这一段：

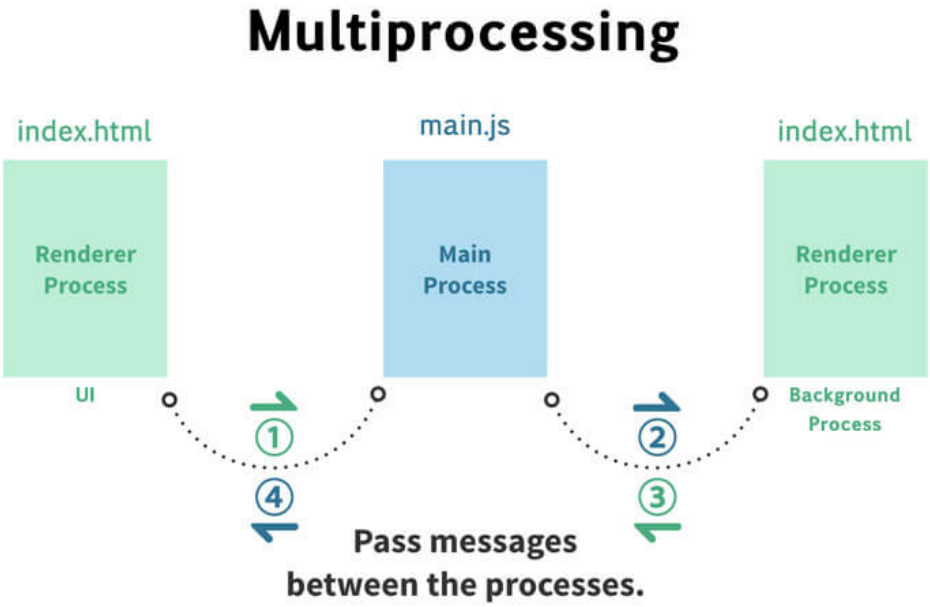
> Node integration doesn't work in web workers, and there is no plan to do. Workers in Chromium are implemented by starting a new thread, and Node is not thread safe. Back in past we had tried to add node integration to web workers in Atom, but it crashed too easily so we gave up on it.

因此，我们最终采用了创建一个新的渲染进程 `background process` 进行处理数据。由 Electron 章节可知，每个 Electron 渲染进程是独立的，因此它们不会互相影响。但这也带来了一个问题：它们不能相互通讯?

错！下面有 3 种方式进行通讯：

1. Storage API (https://developer.mozilla.org/en-US/docs/Web/API/Storage)：对某个标签页的 localStorage/sessionStorage 对象进行增删改时，其他标签页能通过 window.storage 事件监听到。
2. IndexedDB (https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API)：IndexedDB 是一个为了能够在客户端存储可观数量的结构化数据，并且在这些数据上使用索引进行高性能检索的 API。
3. 通过主进程作为中转站：设主界面的渲染进程是 A，`background process` 是 B，那么 A 先将 Excel 数据传递到主进程，然后主进程再转发到 B。B 处理完后再原路返回，具体如下图。当然，也可以将数据存储在主进程中，然后在多个渲染进程中使用 remote 模块来访问它。

该工具采用了第三种方式的第一种情况：



1、主页面渲染进程 A 的代码如下：

```
1  //①
2  ipcRenderer.send('filter-start', {
3      filterTagList: this.filterTagList,
4          filterWay: this.filterWay,
```

```
        curActiveSheetName: this.activeSheet.name
})

// ⑥ 在某处接收 filter-response 事件
ipcRenderer.on("filter-response", (arg) => {
    // 得到处理数据
})
```

2、作为中转站的主进程的代码如下：

```
//②
ipcMain.on("filter-start", (event, arg) => {
    // webContents 用于渲染和控制 web page
    backgroundWindow.webContents.send("filter-start", arg)
})

// ⑤ 用于接收返回事件
ipcMain.on("filter-response", (event, arg) => {
    mainWindow.webContents.send("filter-response", arg)
})
```

3、处理繁重数据的 background process 渲染进程 B 的代码如下：

```
// ③
ipcRenderer.on('filter-start', (event, arg) => {
    // 进行运算
    ...

    // ④ 运算完毕后，再通过 IPC 原路返回。主进程和渲染进程 A 也要建立相应的监听事件
    ipcRenderer.send('filter-response', {
        filRow: tempFilRow
    })
})
```

至此，我们将『读取文件』、『过滤数据』和『导出文件』三大耗时的数据操作均转移到了 background process 中处理。

这里，我们只创建了一个 background process，如果想要做得更极致，我们可以新建『CPU 线程数 − 1』个的 background process 同时对数据进行处理，然后在主进程对处理后数据进行拼接，最后再将拼接后的数据返回到主页面的渲染进程。这样就可以充分榨干 CPU 了。当然，在此笔者不会进行这个优化。

> 不要为了优化而优化，否则得不偿失。 —— 某网友

内存占有量过大

解决了执行效率和渲染问题后，发现也存在内存占用量过大的问题。当时猜测是以下几个原因：

1. 三大耗时操作均放置在 `background process` 处理。在通讯传递数据的过程中，由于不是共享内存（因为 IPC 是基于 Socket 的），导致出现多份数据副本（在写这篇文章时才有了这相对确切的答案）。
2. Vuex 是以一个全局单例的模式进行管理，但它会是不是对数据做了某些封装，而导致性能的损耗呢?
3. 由于 JavaScript 目前不具有主动回收资源的能力，所以只能主动对闲置对象设置为 `null`，然后等待 GC 回收。

> 由于 Chromium 采用多进程架构，因此会涉及到进程间通信问题。Browser 进程在启动 Render 进程的过程中会建立一个以 UNIX Socket 为基础的 IPC 通道。有了 IPC 通道之后，接下来 Browser 进程与 Render 进程就以消息的形式进行通信。我们将这种消息称为 IPC 消息，以区别于线程消息循环中的消息。
> ——《Chromium的IPC消息发送、接收和分发机制分析》(http://blog.csdn.net/luoshengyang/article/details/47822689)

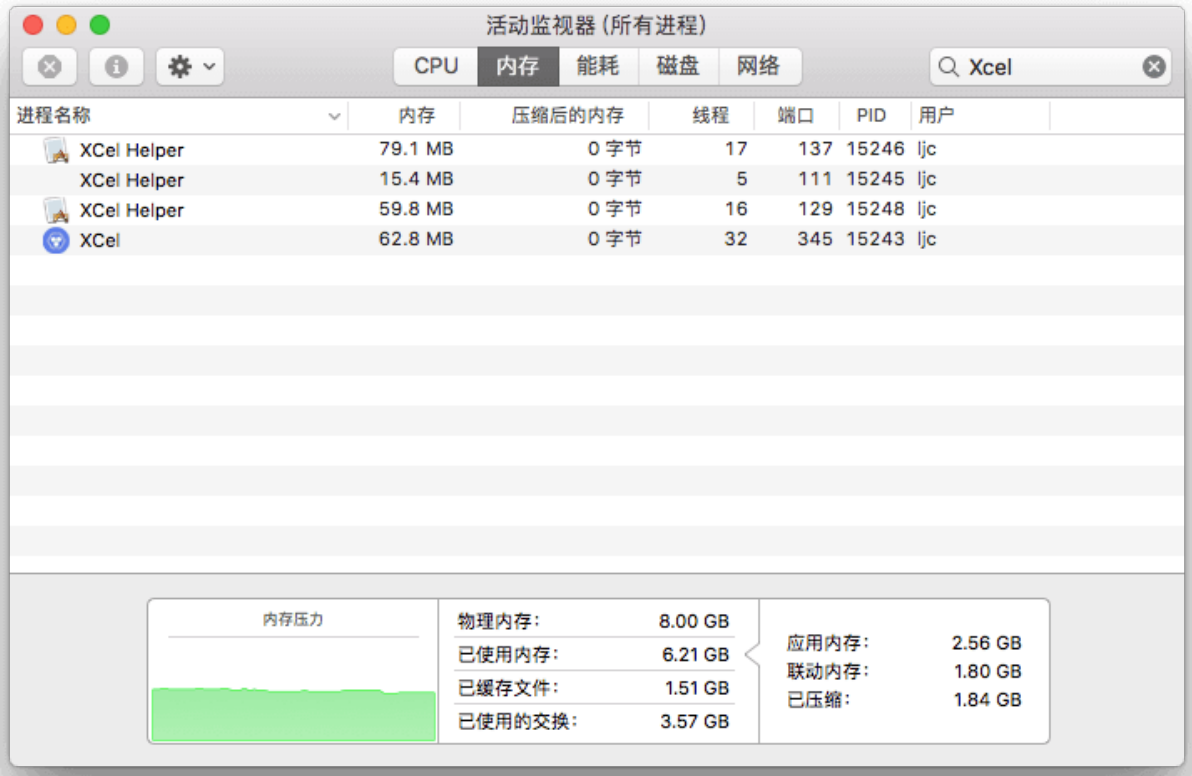
定义：为了易于理解，以下『Excel 数据』均指 Excel 的全部有效单元格转为 JSON 格式后的数据。

最容易处理的无疑是第三点，手动将不再需要的变量及时设置为 `null`，但效果并不明显。

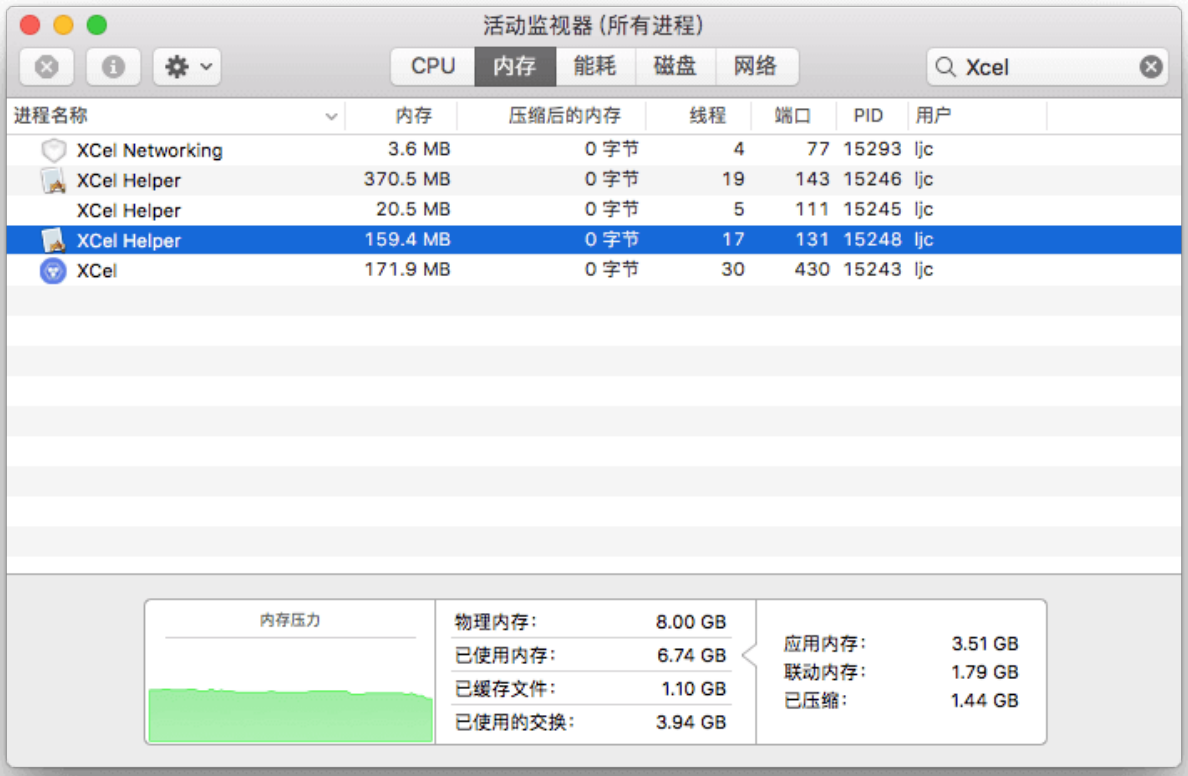
后来，通过操作系统的『活动监视器』（Windows 上是任务管理器）对该工具的每阶段（打开时、导入文件时、筛选时和导出时）进行粗略的内存分析，得到以下报告：

————————– S：报告分割线 ————————–
经观察，主要耗内存的是页面渲染进程。下面通过截图说明：
`PID 15243` 是主进程
`PID 15246` 是页面渲染进程
`PID 15248` 是 background 渲染进程

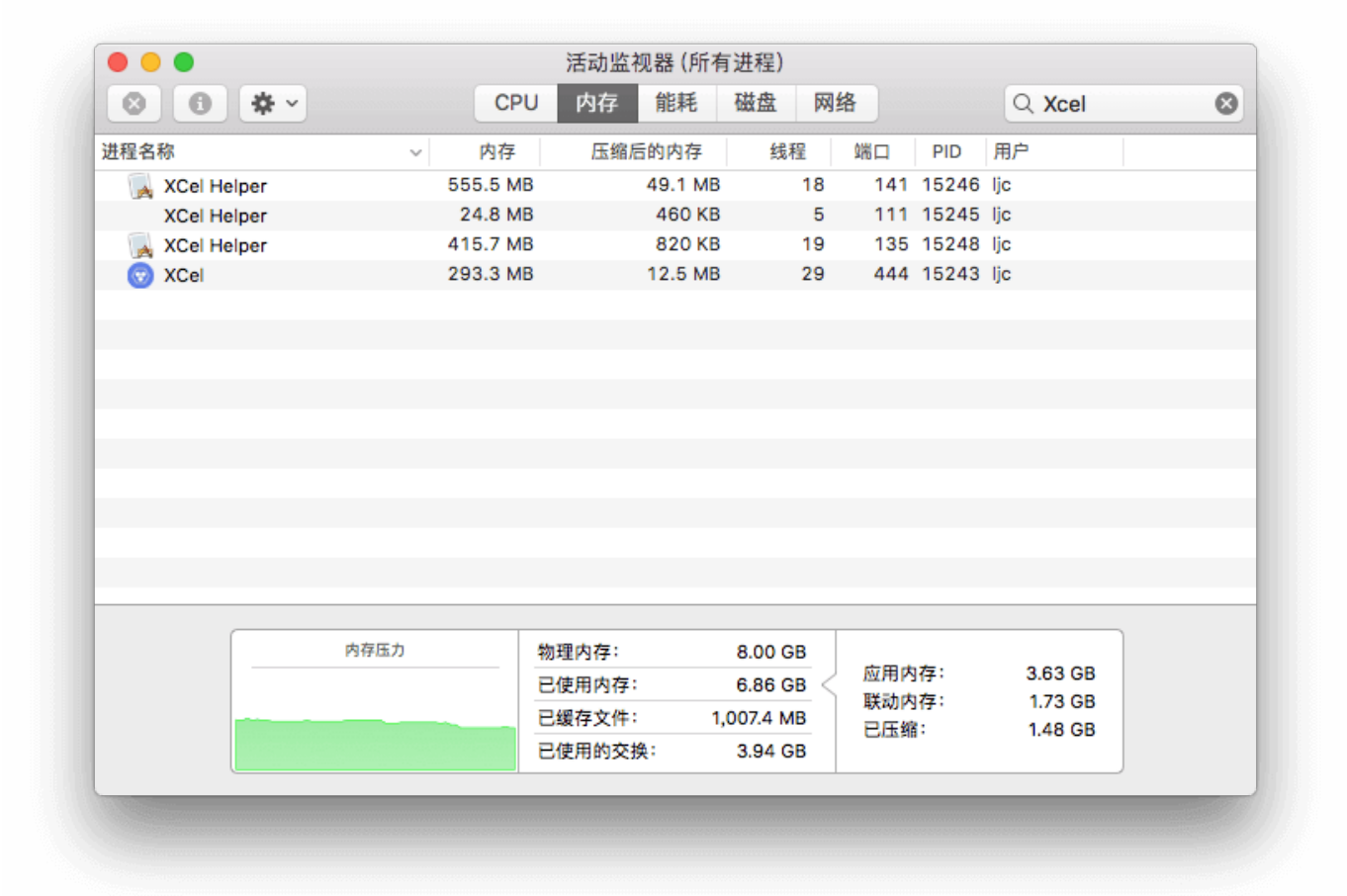
a、首次启动程序时（第 4 行是主进程；第 1 行是页面渲染进程；第 3 行是 background 渲染进程 ）



b、导入文件（第 5 行是主进程；第 2 行是页面渲染进程；第 4 行是 background 渲染进程 ）



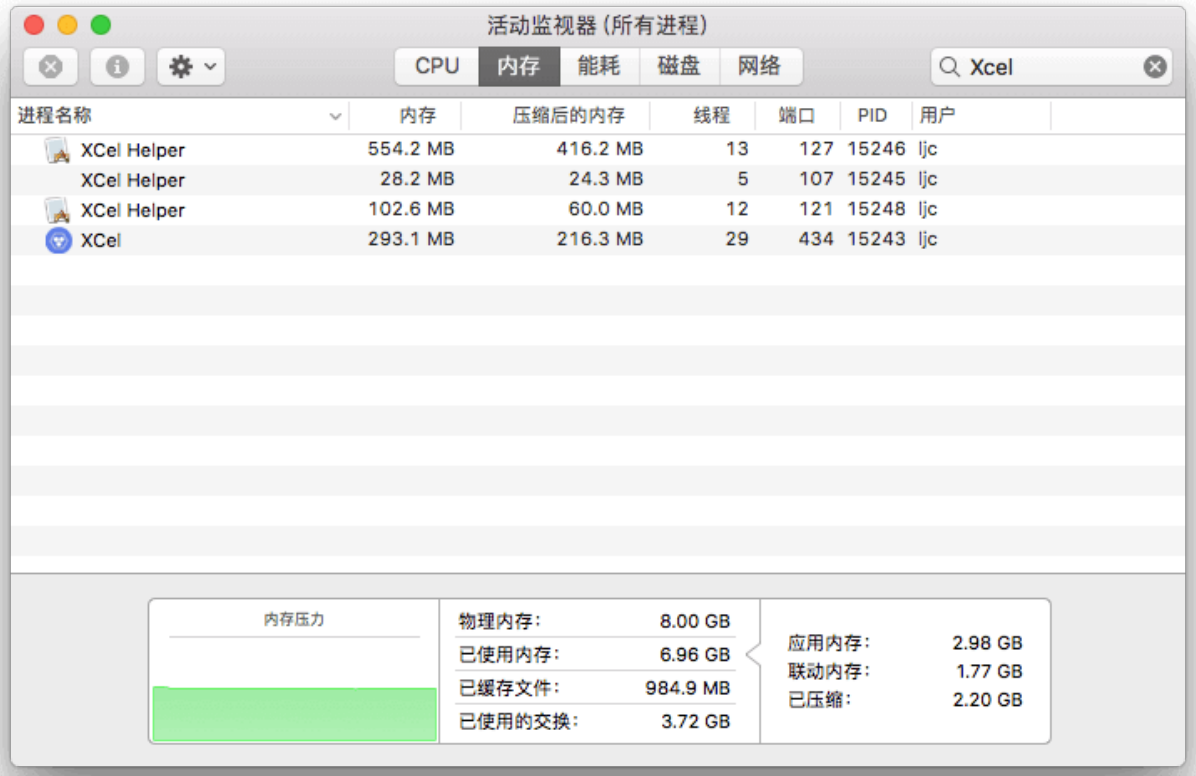
c、筛选数据（第 4 行是主进程；第 1 行是页面渲染进程；第 3 行是 background 渲染进程 ）



由于 JavaScript 目前不具有主动回收资源的功能，所以只能主动将对象设置为 `null` ，然后等待 GC 回收。

因此，经过一段时间等待后，内存占用如下：

d、一段时间后（第 4 行是主进程；第 1 行是页面渲染进程；第 3 行是 background 渲染进程 ）



由上述可得，页面渲染进程由于页面元素和 Vue 等 UI 相关资源是固定的，占用内存较大且不能回收。主进程占用资源也不能得到很好释放，暂时不知道原因，而 background 渲染进程则较好地释放资源。

————————– E：报告分割线 ————————–

根据报告，初步得出的结论是 Vue 和通讯时占用资源较大。

根据该工具的实际应用场景：Excel 数据只在『导入』和『过滤后』两个阶段需要展示，而且展示的是通过 JavaScript 拼接的 HTML 字符串所构成的 DOM 而已。因此将表格数据放置在 Vuex 中，有点滥用资源的嫌疑。

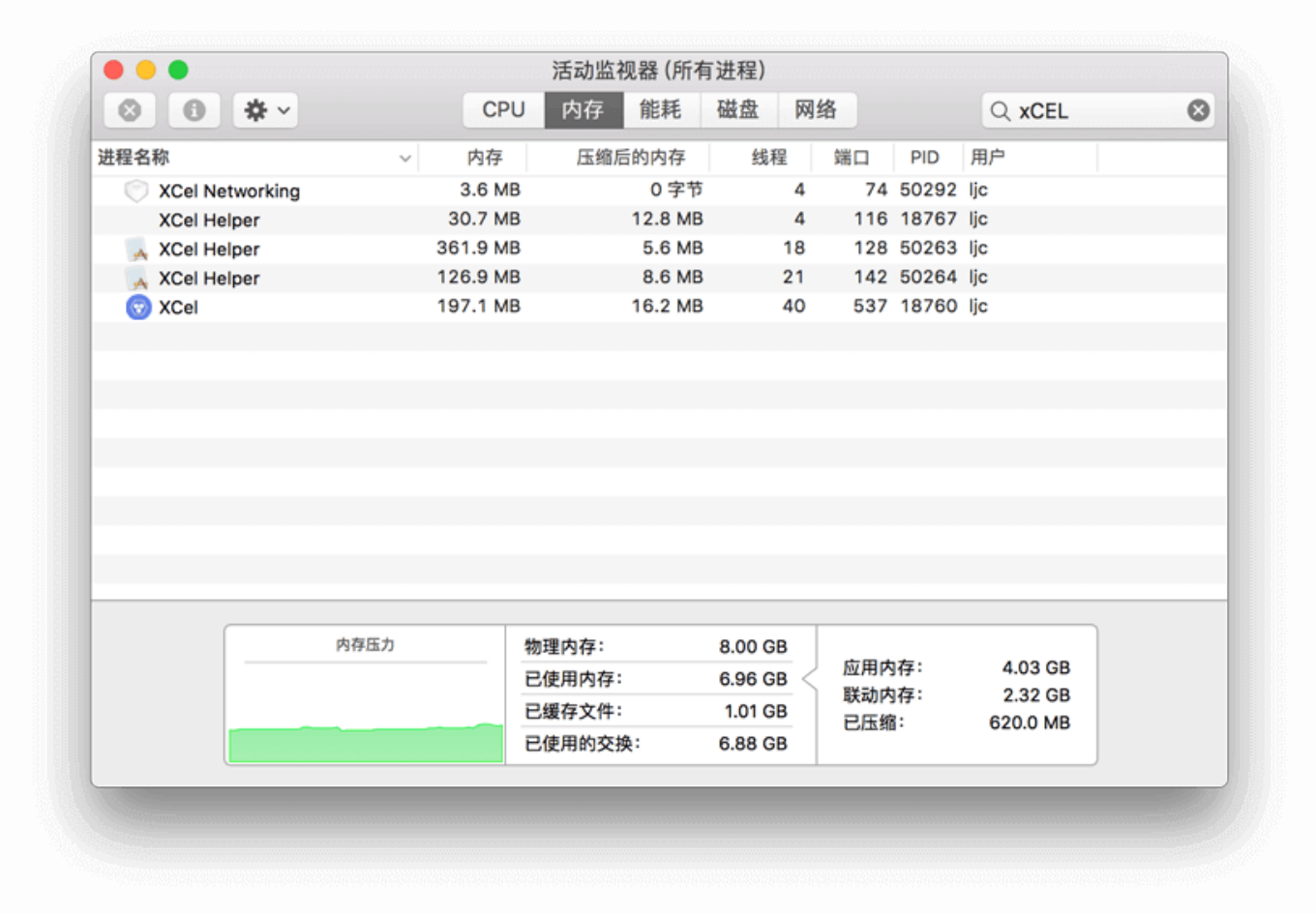
另外，在 `background process` 中也有存有一份 Excel 数据副本。因此，索性只在 `background process` 存储一份 Excel 数据，然后每当数据变化时，通过 IPC 让 `background process` 返回拼接好的 HTML 字符串即可。这样一来，内存占有量立刻下降许多。另外，这也是一个一举多得的优化：

1. 字符串拼接操作也转移到了 `background process`，页面渲染进程进一步减少耗时的操作；
2. 内存占有量大大减小，响应速度也得到了提升。

其实，这也有点像 Vuex 的『全局单例模式管理』，一份数据就好。

当然，对于 Excel 的基本信息，如行列数、SheetName、标题组等均依然保存在 Vuex。

优化后的内存占有量如下图。与上述报告的第三张图相比（同一阶段），内存占有量下降了 44.419%：



另外，对于不需要响应的数据，可通过 `Object.freeze()` 冻结起来。这也是一种优化手段。但该工具目前并没有应用到。

至此，优化部分也阐述完毕了！

该工具目前是开源的，欢迎大家使用或推荐给用研组等有需要的人。

你们的反馈（可提交 issues (https://github.com/o2team/xcel/issues) / pull request (https://github.com/o2team/xcel/pulls)）能让这个工具在使用和功能上不断完善。

最后，感谢 LV (https://github.com/mamboer) 在产品规划、界面设计和优化上的强力支持。全文完！

感谢您的阅读，本文由 Jc (https://github.com/JChehe) 原创提供。如若转载，请注明出处：凹凸实验室（https://aotu.io/notes/2016/11/15/xcel/）

上次更新：2020-05-12 16:38:09