绘制周期(Draw cycle)

网络应用的运行效率应该是非常流畅的，用户从不喜欢等待；“每分每秒的等待都很难受！”对于简单的应用，提高它的运行效率不会很困难，通常只要遵守提高应用效率的最佳实践，并写出高质量的Javascript就可以办到。但是对于复杂的应用，上述的优化方式在DOM操作和样式渲染时并没有太大作用，这时，我们就需要MontageJS的介入。

为了最大限度的优化应用的运行效率，MontageJS的组件会在一个被有效控制的绘制周期里运行，这个绘制周期存在的目的在于减少浏览器重绘页面时带来的昂贵开销。DOM的读/写操作被分离、分批的有计划通过定时循环的方式由浏览器的 `requestAnimationFrame()` API执行着。

影响效率的因素

绘制周期是一个让组件操作关联的DOM的定时循环系统。技术上来说，组件可以在任何时候操作DOM；但从性能优化的最佳实践考虑，我们应该将DOM的读/写操作分批进行处理。但受限于浏览器对DOM的控制方式，我们想要控制这个流程是很困难的。总的来说，浏览器还是非常聪明的，它会将DOM的操作排列组合起来并在适当的时候执行，以此来最大化页面的运行效率。但是当你的脚本读取DOM样式时（比如 `element.offsetLeft` ）,会强迫浏览器给你最新的值，这回导致页面的重绘，这个动作有时会消耗很大。

> 了解浏览器将页面源代码载入之后会做些什么，请阅读 Stoyan Stefanov的文章Rendering: repaint, reflow/relayout, restyle.

解决交错读/写DOM问题在组件层面是比较容易的；你只需要小心的处理DOM操作即可。但是在复杂的网络应用程序中。你需要面对不可预计的组件，有一些可能还不是你自己写的。

我们举一个有两个组件的应用为例，这个应用包括一个漂亮的滑块和一个文本框，它们的值相互绑定。当用户拖拽滑块时，文本框里的数字会和滑块当前位置的值一样；同样的，当用户在文本框里输入数字时，滑块也会移动到对应位置。

所以拖拽滑块会迫使滑块组件更新DOM，同时文本框组件也必须更新DOM来显示滑块所处位置的值。如果每个组件的绘制都是独立的，我们很大可能会遇到DOM交错读/写的问题。想像一下将这个场景扩大10倍，20倍。这就是我们需要Montage绘制管理的原因：它确保了DOM的读/写操作不会交错进行，并且组件还是保持着相互独立。

绘制周期简介

当一个组件需要更新自己时，它的 `needsDraw` 属性会被设置为 `true` （ `this.needsDraw = true` ）。通过这个设置页面模版会知道有一个子组件需要绘制，一个绘制周期会产生。大体来说，绘制周期有这样两个阶段

1. 要求绘制阶段, 这个阶段会根据哪些组件的 `needsDraw` 被设置为 `true` 产生一个需要被绘制的组件列表。
2. 绘制阶段，这个阶段组件列表里组件的一系列的回调函数会被有序的执行，三个主要的回调函数是 `willDraw()` ， `draw()` 和 `didDraw()`

绘制周期会自动执行浏览器 `requestAnimationFrame()` 方法，如果不支持该方法， `setTimeout()` 会代替它。

创建绘制列表

在每个绘制周期，绘制控制器会遍历整个应用的组件树并将 `needsDraw` 为 `true` 的组件加入绘制列表。如果一个组件是第一次被加入绘制列表， `prepareForDraw()` 回调函数会被最早执行。(这个调用保证了元素在被设置事件监听前肯定存在于HTML文档中，进一步来看，如果这个组件包含HTML模版，这个时刻就是组件的 `.element` 属性真正关联到模版元素的时刻)

在绘制列表初始化之后，绘制管理器会调用每个组件的 `willDraw()` 方法。这时很用可能更多的组件（子组件）的 `needsDraw` 会被设置为 `true` 。为了将这些新增组件加入绘制周期，绘制控制器会再次遍历组件树将新增的组件加入绘制列表。接着新组件的 `willDraw()` 方法也会被调用，这个过程会一直重复直到没有新组件需要加入列表。（或直到浏览器开始它的绘制周期）。在开始绘制后再要求绘制的组件会被加入到下一次的绘制周期。

执行绘制列表

在绘制列表被创建完成以后，绘制控制器会调用列表中每个组件的 `draw()` 方法。这个方法中组件需要操作DOM或者CSS样式表；我们不允许在 `draw()` 方法以外的地方对DOM进行操作。

最后，如果组件需要在绘制周期结束后做些什么，在所有的 `draw()` 方法被执行之后，绘制列表中每个组件的 `didDraw()` 方法会被调用。

回调函数

下面我们列出了在绘制周期中组件会被调用的几个回调函数。这些方法绝不应该被直接调用；MontageJS框架会自动调用他们，任何需要操作DOM的组件都需要实现 `draw()` 方法。

prepareForDraw()

- **执行时刻**：组件第一次被加入到绘制列表时。

- **描述**：运行组件为第一次绘制做准备，对于有模版的组件，这个方法会在模版被加载进入DOM后被执行。

willDraw()

- **执行时刻**：在 `draw()` 执行之前，绘制列表生成以后被执行。
- **描述**：这个方法中DOM可以允许被读取。如果在这个方法中将其他某个组件的 `needsDraw` 设置为 `true`，这些组件会被加入到此次绘制周期中。

draw()

- **执行时刻**： `willDraw()` 方法执行以后被调用
- **描述**：在这里对DOM进行操作。

didDraw()

- **执行时刻**： `draw()` 方法执行以后被调用（如果组件需要在绘制周期最终阶段做点什么）
- **描述**：在绘制周期结束之前给组件一个机会读取DOM。

如何使用绘制周期

绘制周期是MontageJS内部的执行机制。只要你使用MontageJS的内置组件，MontageJS框架就已经为你实现了绘制周期。当你创建自己的组件时，务必遵守以下规则：

1. 组件绝不能在 `draw()` 方法以操作DOM，操作包括对CSS样式表的操作以及添加、删除页面元素。
2. 任何对DOM尺寸的读取（比如元素的 `offsetWidth` 属性）只允许在 `willDraw()` 和 `didDraw()` 方法中进行，绝不要在 `draw()` 方法中操作。

在你的组件中遵循这些规则可以最大限度提高你整个应用的运行效率。

查看MFiddle学习绘制周期的例子。