pace.js解析

源码地址：[CodeByZach/pace: Automatically add a progress bar to your site. (github.com)](https://github.com/CodeByZach/pace)

pace.js是一个自动页面进度条加载的库，提供了丰富的主题库供选择。不依赖其他库，并且大小只有4kb。

pace的使用，引入之后，可以直接使用，如果有特殊需求，pace也暴露出一些配置来处理

Fus webGUI运行时需要加载资源，我们使用pace.js，配置了`elements: { selectors: ['.app-page'] }`。等待app-page渲染完之后关闭进度条

```
// Pace是全自动的，无需配置即可开始。
// 如果您想进行一些调整，请按以下步骤操作：
// 您可以在引入文件之前进行设置：window.paceOptions
window.paceOptions = {
    // Disable the 'elements' source
    elements: false,
    // Only show the progress on regular and ajax-y page navigation, not every
request
    restartOnRequestAfter: false
}
```

```
<!-- 还可以在脚本标签上放置选项 -->
<script data-pace-options='{ "ajax": false }' src="pace.min.js"></script>
```

监听配置项

Pace包括4个默认的收集器

- ajax：监视页面上的所有 ajax 请求
- document：检查document readyState
- eventLag：检查javascript正在执行的事件循环滞后信号
- elements：检查页面上是否存在特定元素

```
// pace.js解析
paceOptions = {
    ajax: false, // disabled
    document: false, // disabled
    eventLag: false, // disabled
    elements: {
        selectors: ['.my-page']
    }
    restartOnRequestAfter: false,
};
```

AjaxMonitor

ajax监听器可以监听两种类型的，一种是xhr，另外一种是websocket。

通过给原生的请求加代理，从而监听请求

```
monitorXHR = function(req) {
    var _open;
    _open = req.open;
    return req.open = function(type, url, async) {
        if (shouldTrack(type)) {
            _this.trigger('request', {
                type: type,
                url: url,
                request: req
            });
        }
        return _open.apply(req, arguments);
    };
};
window.XMLHttpRequest = function(flags) {
    var req;
    req = new _XMLHttpRequest(flags);
    monitorXHR(req);
    return req;
};
```

XHRRequestTracker：通过progress事件来计算请求进度。

SocketRequestTracker：只监听`error`和`open`事件。websocket链接建立完成，进度就是100

ElementMonitor

为每一个需要监听的节点创建一个ElementTracker对象

ElementTracker：通过定时器监听节点是否存在，如果节点存在则，监听结束

DocumentMonitor

通过onreadystatechange事件，判断document是否加载完成

EventLagMonitor

事件监听器会创建一个定时器，计算定时器每次执行的时间间隔，如果时间间隔大于某个数(默认为3ms)，表示还有js事件在执行,否则表示事件结束。

```
        this.progress = 0;
        avg = 0;
        samples = [];
        points = 0;
        last = now();
        interval = setInterval(function() {
            var diff;
            diff = now() - last - 50;
            last = now();
            samples.push(diff);
            if (samples.length > options.eventLag.sampleCount) {
                samples.shift();
            }
            avg = avgAmplitude(samples);
            if (++points >= options.eventLag.minSamples && avg < options.eventLag.lagThreshold) {
                _this.progress = 100;
                return clearInterval(interval);
            } else {
                return _this.progress = 100 * (3 / (avg + 3));
            }
        }, 50);
```

关于为什么能用setInterval去判断：参考链接：setInterval

在浏览器中，定时器执行之前，如果其他事件没有执行完,定时器会等待事件执行完之后执行

```
function sleep(time) {
  let startTime = window.performance.now();
  while (window.performance.now() - startTime < time) {}
}

let count = 1;
let getTime = window.performance;
let startTime = getTime.now();

console.log(startTime);
// 源码的例子好懂，这里模仿我们的进度条
setInterval(function () {
  console.log(`第${count}次开始 ${getTime.now() - startTime}`); // 显示开始时间
  count += 1;
}, 300); // 300ms间隔

setTimeout(() =>sleep(500), 100)
setTimeout(() =>sleep(1000), 1000)
```

第1次执行	100.79999995231628
第2次执行	401
第3次执行	508.39999997615814
第4次执行	602.6000000238419
第5次执行	711.2999999523163
第6次执行	804.5
第7次执行	913.3999999761581
第8次执行	1006.3999999761581
第9次执行	1507.7000000476837
第10次执行	1602.5
第11次执行	1713.5

可以看出，当有代码在执行时，定时器会等待事件执行完毕再去执行。所以获取到定时器执行的时间间隔，多次取平均值，判断是否有代码执行。

Scaler(缩放器，计算进度条的百分比)

pace 会创建一个总的scaler，然后会给每一个开放的监听器创建一scaler，将所有监听器的进度平局值当做总进度条的值

runAnimation循环计算进度，只到进度到100

bar: 进度条UI对象

uniScaler: 总的进度对象

```
return animation = runAnimation(function(frameTime, enqueueNextFrame) {
    var avg, count, done, element, elements, i, j, remaining, scaler, scalerList, sum, _j, _k, _len1, _len2, _ref2;
    remaining = 100 - bar.progress;
    count = sum = 0;
    done = true;
    for (i = _j = 0, _len1 = sources.length; _j < _len1; i = ++_j) {
        source = sources[i];
        scalerList = scalers[i] != null ? scalers[i] : scalers[i] = [];
        elements = (_ref2 = source.elements) != null ? _ref2 : [source];
        for (j = _k = 0, _len2 = elements.length; _k < _len2; j = ++_k) {
            element = elements[j];
            scaler = scalerList[j] != null ? scalerList[j] : scalerList[j] = new Scaler(element);
            done &= scaler.done;
            if (scaler.done) {
                continue;
            }
            count++;    监听器数量
            sum += scaler.tick(frameTime);  监听器总进度
        }
    }
    avg = sum / count;
    bar.update(uniScaler.tick(frameTime, avg));
```

其他配置项

restartOnPushState：默认为true，任何的`history.pushState``history.replaceState`都会渲染进度条

```
    handlePushState = function() {
        if (options.restartOnPushState) {
            return Pace.restart();
        }
    };

    if (window.history.pushState != null) {
        _pushState = window.history.pushState;
        window.history.pushState = function() {
            handlePushState();
            return _pushState.apply(window.history, arguments);
        };
    }

    if (window.history.replaceState != null) {
        _replaceState = window.history.replaceState;
        window.history.replaceState = function() {
            handlePushState();
            return _replaceState.apply(window.history, arguments);
        };
    }
```

事件

Pace 触发以下事件：

- `start`：最初开始配速时，或作为重新启动的一部分时
- `stop`：手动停止配速时，或作为重新启动的一部分时
- `restart`：重新启动 pace 时（手动或通过新的 AJAX 请求）
- `done`：当配速完成时
- `hide`：当速度被隐藏时（可以晚于 、基于和 `done``ghostTime``minTime`）

可以使用 和 方法绑定到事件：`on``off``once`

- `Pace.on(event, handler, [context])`：触发时调用（可选带上下文）`handler``event`
- `Pace.off(event, [handler])`：解绑提供的和组合。`event``handler`
- `Pace.once(event, handler, [context])`：绑定到下一个（且仅下一个）发生率 `handler``event`

`Pase`继承了自定义的`Evented`对象，`Evented`对象中定义了`on, once, off, trigger`等方法，此处简单介绍on方法。

```
/**
    on方法会将监听的事件放入bindings数组中
    每一个数组对象包括cb，上下文，是否只监听一次（由此可知once方法就是直接调用on方法），
    在触发事件时，根据类型获取bindings数组的事件，执行cb
*/
Evented.prototype.on = function(event, handler, ctx, once) {
    var _base;
    if (once == null) {
```

```
        once = false;
    }
    if (this.bindings == null) {
        this.bindings = {};
    }
    if ((_base = this.bindings)[event] == null) {
        _base[event] = [];
    }
    return this.bindings[event].push({
        handler: handler,
        ctx: ctx,
        once: once
    });
};
Evented.prototype.trigger = function() {
    var args, ctx, event, handler, i, once, _ref, _ref1, _results;
    event = arguments[0], args = 2 <= arguments.length ? __slice.call(arguments,
1) : [];
    if ((_ref = this.bindings) != null ? _ref[event] : void 0) {
        i = 0;
        _results = [];
        while (i < this.bindings[event].length) {
            _ref1 = this.bindings[event][i], handler = _ref1.handler;
            ctx = _ref1.ctx, once = _ref1.once;
            handler.apply(ctx != null ? ctx : this, args);
            if (once) {
                _results.push(this.bindings[event].splice(i, 1));
            } else {
                _results.push(i++);
            }
        }
        return _results;
    }
};
```