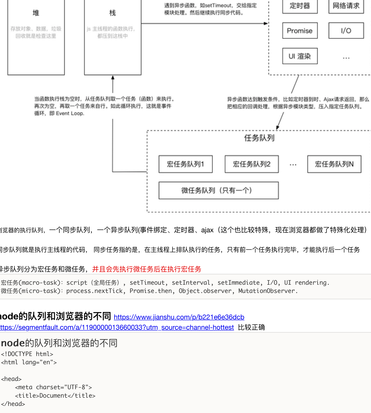
JavaScript 异步、事件循环、任务队列



浏览器的执行队列，一个同步队列，一个异步队列(事件绑定、定时器、ajax（这个也比较特殊，现在浏览器都做了特殊化处理）

同步队列就是执行主线程的代码，同步任务指的是，在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务

异步队列分为宏任务和微任务两，并且会先执行微任务后在执行宏任务

```
宏任务(macro-task): script（全局任务），setTimeout, setInterval, setImmediate, I/O, UI rendering.
微任务(micro-task): process.nextTick, Promise.then, Object.observer, MutationObserver.
```

node的队列和浏览器的不同 https://www.jianshu.com/p/b221e6e36dcb https://segmentfault.com/a/1190000013660337?utm_source=channel-hottest 比较正确

node的队列和浏览器的不同

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>

<body>
    <div>
        在Nodejs事件循环机制中，有任务两个队列：Macrotask队列和Microtask队列，在一个事件循环里，这两个队列会分两步执行，第一步会固定地执行一个（且仅一个）Macrotask任务，第二步会执行整个Microtask队列中的所有任务，并且，在执行Microtask队列任务的时候，也允许加入新的Microtask任务，直到所有Microtask任务全部执行完毕，才会结束循环。 Macrotasks—般包括：setTimeout, setInterval, setImmediate, I/O, UI rendering; Microtasks—般包括：process.nextTick, Promises, Object.observe, MutationObserver.
    </div>
    <script type="text/javascript">
    setTimeout(function() {
        console.log(5);
    }, 0);
    setImmediate(function() {
        console.log(6);
    });
    new Promise(function(resolve) {
        console.log(1);
        resolve();
        console.log(2);
    }).then(function() {
        console.log(4);
    });
    console.log('打酱油');
    // process.nextTick(function() {
    //     console.log(3);
    //     });
    </script>
</body>

</html>
```

MutationObserver的使用

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>

<body>
<div id="some-id" data-test="123">
    为了让 flush 动作能在当前 Task 结束后尽可能早的开始，Vue 会优先尝试将任务 micro-task 队列，具体来说，在浏览器环境中 Vue 会优
先尝试使用 MutationObserver API 或 Promise，如果两者都不可用，则 fallback 到 setTimeout。
<hr/>

</div>
    <script type="text/javascript">
    // Firefox和Chrome早期版本中带有前缀
    var MutationObserver = window.MutationObserver || window.WebKitMutationObserver ||
window.MozMutationObserver
        // 选择目标节点
    var target = document.querySelector('#some-id');
    console.log(MutationObserver);
    // 创建观察者对象
    var observer = new MutationObserver(function(mutations) {
        mutations.forEach(function(mutation) {
            console.log(mutation.type);
        });
    });
    // 配置观察选项:
    var config = {
            attributes: true,
            childList: true,
            characterData: true
        }
        // 传入目标节点和观察选项
    observer.observe(target, config);
    // 随后,你还可以停止观察
    document.querySelector('#some-id').setAttribute('data-test',"xxx");
    // observer.disconnect();
    </script>
</body>

</html>
```

来2个例子试试

```
setTimeout(function(){
    console.log(5);
},0)

console.log(4);

new Promise(function(){
    console.log(1);
    resolve();
    console.log(2);
}).then(function(){
    console.log(3)
})

//4 1 2 3 5
```

```
setTimeout(function(){
    console.log(5);
},0)

new Promise(function(){
    console.log(1);
    resolve();
    console.log(2);
}).then(function(){
    console.log(3);
})

console.log(4);

//1 2 4 3 5
```

一.单线程

我们常说"JavaScript是单线程的"，
所谓单线程，是指在JS引擎中负责解释和执行JavaScript代码的线程只有一个，不妨叫它主线程。
但是实际上还存在其他的线程，例如：处理AJAX请求的线程，处理DOM事件的线程，定时器线程，读写文件的线程(例如在Node.js中)等等。这些线程可能存在于JS引擎之内，也可能存在于JS引擎之外，在此我们不做区分。不妨叫它们工作线程。

二.栈

```
function bar() {
    console.log(1);
}

function foo() {
    console.log(2);
    bar();
}

setTimeout(() => {
    console.log(3)
});

foo();
```

它在栈中的入栈、出栈过程，如图

二.事件循环

Event Loop，不管是前端，还是移动端（IOS和Android）等开发，都离不开事件循环机制，他会循环监听任务，并在适当的时机取出，执行和释放任务，更新UI等操作，更新UI渲染界面比较耗时，不同的渲染引擎有自己的一套渲染时机逻辑，决定要不要马上执行更新，毕竟更新UI成本大。

异步过程中，工作线程在异步操作完成后需要通知主线程。那么这个通知机制是怎样实现的呢？答案是利用消息队列和事件循环。

工作线程将消息放到消息队列，主线程通过事件循环过程去取消息。

- **消息队列**：消息队列是一个先进先出的队列，它里面存放着各种消息。
- **事件循环**：事件循环是指主线程重复从消息队列中取消息、执行的过程。

三.异步与事件绑定

消息队列中的每条消息实际上都对应着一个事件也是异步的。

上文中一直没有提到一类很重要的异步过程：**DOM事件**。
举例来说：

```
var button = document.getElement('#btn');
button.addEventListener('click', function(e) {
    console.log();
});
```

从事件的角度来看，上述代码表示：在按钮上添加了一个鼠标单击事件的事件监听器：当用户点击按钮时，鼠标单击事件触发，事件监听器函数被调用。

从异步过程的角度看，addEventListener函数就是异步过程的发起函数，事件监听器函数就是异步过程的回调函数。事件触发时，表示异步任务完成，会将事件监听器函数封装成一条消息放到消息队列中，等待主线程执行。

另一方面，所有的异步过程也都可以用事件来描述。例如：setTimeout可以看成对应一个时间到了！的事件。前文的setTimeout(fn, 1000)也可以看成：

```
timer.addEventListener('timeout', 1000, fn);
```

主要资源：

JavaScript：彻底理解同步、异步和事件循环(Event Loop) -> https://segmentfault.com/a/1190000004322358
JavaScript 异步、栈、事件循环、任务队列 -> https://segmentfault.com/a/1190000011198232
Javascript-宏队列与微队列 -> https://www.jianshu.com/p/ada516ceb1da