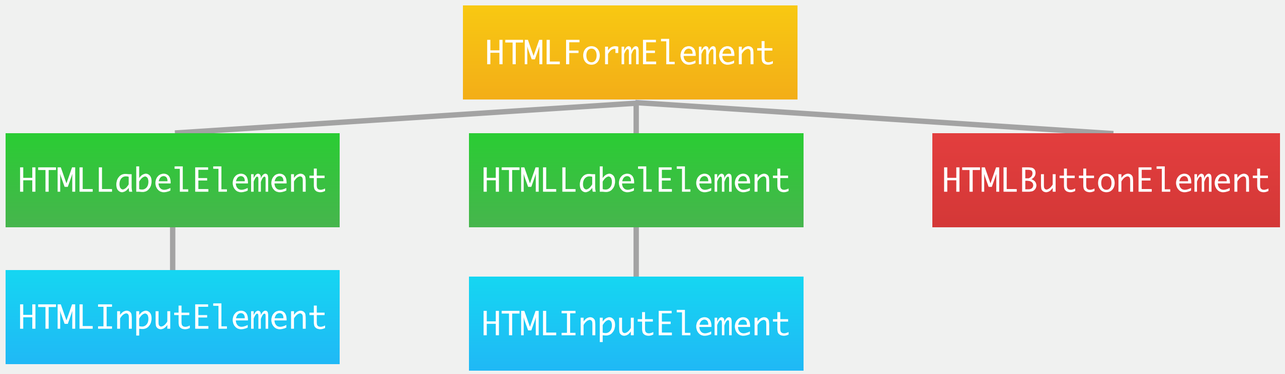
DOM &
WEB API

DOM

```
<form class="login-form">
    <label>用户名: <input type="text" name="username"></label>
    <label>密码: <input type="password" name="password"></label>
    <button class="submit">提交</button>
</form>
```



DOM 全称 Document Object Model，即文档对象模型，是将 HTML 文本映射到代码（通常是 JavaScript）中的模型。

DOM - document

```
document.querySelector('form') // 获取文档中的第一个 form 元素
document.querySelector('.login-form .submit') // 获取第一个符合 selector 的元素

const inputs = document.querySelectorAll('input') // 获取文档中的所有 input 元素
console.log(inputs.length) // 2
console.log(inputs instanceof Array) // false
// to Array:
const inputsArray = [...inputs]

// querySelectorAll 返回的是一个类数组对象，而不是数组
// 如果可能的话，将 query 得到的结果缓存下来，不要多次重复 query，以免带来性能损耗
// 但是缓存结果不会自动响应 DOM 变化，因此 DOM 变化之后可能需要重新 query 来更新缓存
```


document 对象是一个全局变量，它是浏览器所加载的整个文档的映射，代表整个 DOM 树。我们对 DOM 树的遍历通常从 document 对象开始。

DOM - document

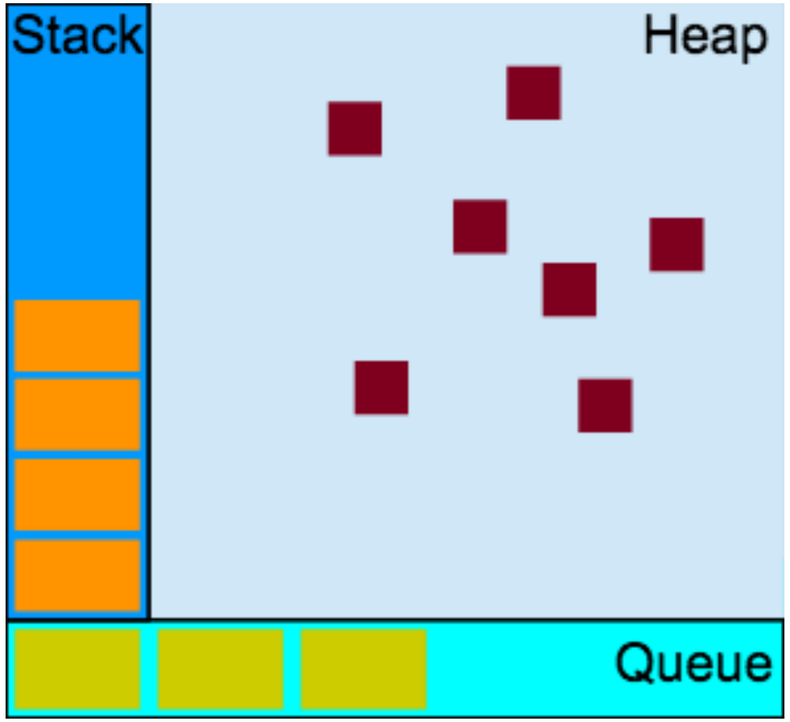
```
const button = document.createElement('button')
button.innerHTML = '清空'
const form = document.querySelector('form')
form.appendChild(button)
```

DOM - element

appendChild	在非空元素末尾添加子节点
removeChild	移除指定的子节点
insertBefore	将给定节点插入到参考节点之前
replaceChild	将指定的子节点用另一个节点替换

DOM - 事件

```
const submitButton = document.querySelector('.login-form .submit')
submitButton.addEventListener('click', (event) => {
  console.log('submit button clicked', event)
})
```



DOM - 事件

```
const submitButton = document.querySelector('.login-form .submit')

const handler = (event) => {
  console.log('submit button clicked', event)
  submitButton.removeEventListener('click', handler)
}
submitButton.addEventListener('click', handler)
```

DOM - 事件冒泡


```
const form = document.querySelector('.login-form')
const button = form.querySelector('.submit')
form.addEventListner('click', (event) => {
  console.log('form', 'capture', event.target)
}, true)
form.addEventListner('click', (event) => {
  console.log('form', 'bubble', event.target)
})
button.addEventListner('click', (event) => {
  console.log('button', 'capture', event.target)
}, true)
button.addEventListner('click', (event) => {
  console.log('button', 'bubble', event.target)
})
event.preventDefault() // 取消默认操作，例如链接点击时跳转等
event.stopPropagation() // 停止传播（事件不会再传播到路径上的其他元素）
```

```
const submitButton = document.querySelector('.login-form .submit')

const handler = (event) => {
  console.log('submit button clicked', event)
  submitButton.removeEventListener('click', handler)
}
submitButton.addEventListener('click', handler)
```

DOM - 事件

名称	触发时机	冒泡
click/dblclick	当用户点击或者双击时触发	是
focus/blur	当指定对象被聚焦/失焦时触发	否
keydown/ keypress/keyup	键盘按键被按下、按住以及抬起时触发	是
mousedown/ mousemove/ mouseup	鼠标按键按下、鼠标移动、鼠标按键抬起时触发	是
mouseenter/ mouseover/ mouseleave/ mouseout	鼠标指针进入对象所在范围(enter, over)、离开其范围(leave, out)时触发	否/是/否/是

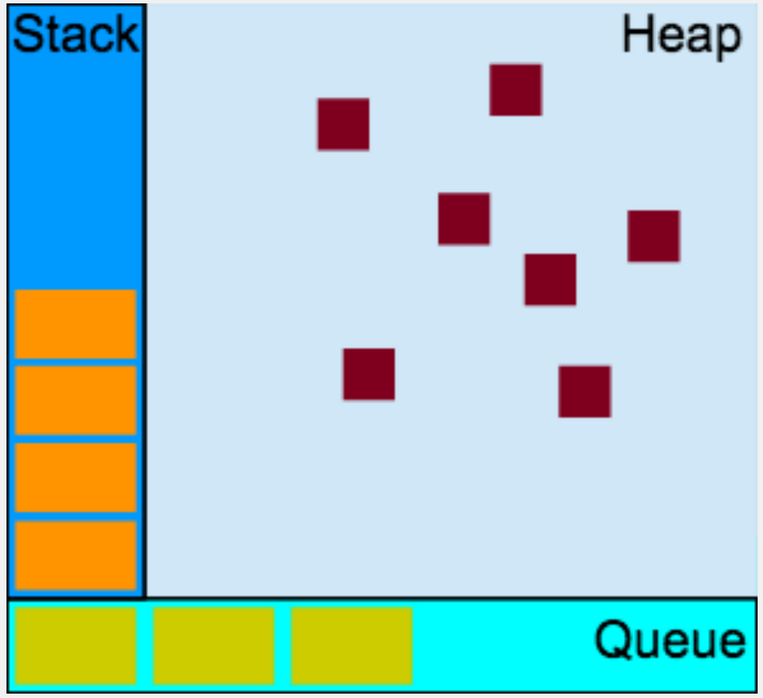
Web API


```
// 1500 ms 后向 event queue 插入一个定时器
// 事件
setTimeout(() => {
  console.log('after 1.5s')
}, 1500)

let i = 0
// 每隔 1500 ms 向 event queue 插入一个定
// 时器事件
setInterval(() => {
  console.log('every 1.5s', i)
  i++
}, 1500)
```

```
let lastTime = Date.now()
const frameHandler = () => {
  let currentTime = Date.now()
  let diffTime = currentTime - lastTime
  console.log(diffTime)
}
// 下次浏览器重绘前调用回调函数
requestAnimationFrame(frameHandler)
```

异步与 Event Loop



异步是指在主程序流程之外的独立发生的事件。例如用户的点击、定时器到时、网络请求完成等，都是在 JavaScript 代码流程之外发生的，都是常见的异步行为。

浏览器中的 setTimeout/setInterval/requestAnimationFrame，用于网络请求的 XHR，fetch 等 API，用于窗口间、WebWorker 间通讯的 postMessage 等方法都是异步的。而 Node 中几乎所有的 IO 操作都是异步的。

JavaScript 引擎会在执行完当前所有的代码，将栈清空后去检查 event queue 中是否仍有事件未处理，如果有的话，将会执行对应的事件绑定的函数。这次执行又会产生新的调用栈，同时可能向 event queue 中添加新的事件。引擎会执行到调用栈再次清空，然后去检查 event queue。

Ajax

AJAX 是 JavaScript 使用代码控制动态网络请求的方法，通常我们会使用 XMLHttpRequest(XHR) 的 API，但是这套 API 已经较为老旧，并且使用起来较为复杂，jQuery 等库封装出了 $.ajax, $.post 等方法来屏蔽掉原生 XHR 的复杂性。最近浏览器开始支持 Fetch API，使用 fetch 函数来简化网络请求发送。

Fetch API

```
// Promise 的链式 then 调用
// fetch 默认发送 GET 请求
fetch('/comments')
.then((res) => res.json()) // 当请求完成后将 res 转换为 json 格式
.then((comments) => {      // 转换的结果会传到下一个 then 回调里
  console.log(comments)
})

// POST 请求，发送 JSON 格式数据
fetch('/comment', {
  method: 'POST',
  headers: { 'Content-type': 'application/json; charset=UTF-8' },
  body: { content, username }
})
```

fetch 是一个全局函数，该函数会发送异步请求，并返回一个 Promise 类型的值。

可以先把 Promise 看成是一种事件注册机制，例如 promise.then(() => {...}) 会在 promise 对应的异步事件完成后调用 then 的回调函数。

Promise

```
// 使用 Promise 封装定时器函数
const wait = (ms) => new Promise((resolve) => {
  // resolve 函数调用之后，这个 Promise 就处于完成状态了
  // 其对应的 then 回调会在当前代码栈执行完之后开始执行
  setTimeout(resolve, ms)
})

wait(1000).then(() => console.log('waited for 1s'))
console.log('waiting')
```

Promise 是 ECMAScript 2015 增加的改进 JavaScript 异步编程的 API。使用 Promise 可以将原来深度嵌套的回调链拉平，同时改善 JavaScript 的异步错误处理。

Promise

```
// 使用 Promise.prototype.catch 进行错误处理
const waitRandom = (ms, threshold) => new Promise((resolve, reject) => {
  setTimeout(() => {
    const random = Math.random()
    if (random > threshold) {
      return reject('random too large')
    }
    resolve(random)
  }, ms)
})

waitRandom(1000, 0.5)
.then((random) => console.log(random))
.catch((reason) => console.log(reason))
```

异步 - async/await

```
// 仅使用 promise 实现等一段时间再进行后续操作的逻辑
const wait = (ms) => new Promise((resolve) => {
  setTimeout(resolve, ms)
})

wait(1000)
.then(console.log('waited for 1s'))

// 使用 async/await 语法实现
async function doSomething() {
  await wait(1000)
  console.log('waited for 1s')
}
doSomething()
```

async/await 是 ECMAScript 2015 新增的支持异步编程的语法，使用 async/await 语法可以使用表观上接近于同步顺序编程的代码风格完成异步编程，减轻程序员的心智负担。

异步 - async/await

```
const waitRandom = (ms, threshold) => new Promise((resolve, reject) => {
  setTimeout(() => {
    const random = Math.random()
    if (random > threshold) {
      return reject('random too large')
    }
    resolve(random)
  }, ms)
})
```

```
async function doSomething() {
  try {
    let random = await waitRandom(1000, 0.5)
    console.log(random)
  } catch (error) {
    console.log(error)
  }
}
doSomething()
```

JavaScript 练习

以下部分标注了 ID，请严格遵守，判题程序将按照给定的 ID 获取元素后，进行事件的触发，并检测 ID 对应元素内的 innerText 值是否符合预期