事件循环丨存储

进程\线程

浏览器其实都是多进程的，当我们打开一个tab页面时就会开启一个新的进程

多线程操作是很容易产生不安全的数据的，我们访问数据的时候通常都是需要上锁的(多线程的情况)，而上锁再解锁的操作是比较耗时的

js 单线程

网络请求、定时器等耗时操作都由浏览器其余线程完成，例如定时器设置2s后执行，当运行到该行代码时，会传递给浏览器，由浏览器的其余线程来"计时"这2s，当2s结束后，浏览器会通知JS线程执行定时器内的代码

同步任务

JS线程怎么知道到时间后要执行的是哪一行代码?

事件循环中，有一个事件队列，浏览器到时间准备让JS线程执行代码时，会将要执行的代码放入这个事件队列中，然后JS线程可以从这个队列中挨个取出需要执行的部分，该队列遵循先进先出原则

异步任务

进程的切换耗时占比太高，所以诞生线程来进行快速切换提升并发。而事件队列由于一些事件耗时太高，所以区分出微任务来执行更关键以及耗时更少的部分，以便提高代码执行效率防止阻塞以及用户体验

微任务

Promise的then回调、 Mutation Observer API、queueMicrotask()、-process.nextTick

宏任务

ajax、setTimeout、setInterval、DOM监听、UI Rendering(UI渲染)，-setImmediate
- requestAnimationFrame，IO

在执行任何宏任务之前，都需要确保微任务队列已经被清空

面试题

```
Promise.resolve()
  .then(() => {
    console.log(0);
    // 0 1 4 2 3 5 6
    //   return 4

    // return的thenable会被视为一个then方法，因此被视为一个微任务，不会立刻执行传递给第二个then方法，而是先push到微任务队列
    // 0 1 2 4 3 5 6
    // return {
    //   then: function (resolve) {
    //     // 1.5
    //     // 大量的计算
    //     resolve(4);
    //   }
    // };

    // Promise.resolve()静态方法通过扩展，能够清楚这是立即执行的同步代码没错，但返回的值是需要then进行接收的
    // 相当于两个 then
    // 第 1个微任务：等待 Promise.resolve(4) 被解析
    // 第2个微任务：将解析后的值 4 传递给下一个 .then()
    // 0 1 2 3 4 5 6
    return Promise.resolve(4);
    /**
     * return new Promise(resolve => {
        resolve(4);
      }).then(value => {
        return value;
      });
     */
  })
  .then((res) => {
    console.log(res);
  });

// 第二个Promise链
Promise.resolve()
  .then(() => {
    console.log(1);
  })
  .then(() => {
    console.log(2);
  })
  .then(() => {
    console.log(3);
  })
  .then(() => {
    console.log(5);
  })
  .then(() => {
    console.log(6);
  });
// Promise.resolve() 本质上是 new Promise((resolve) => resolve())
因此是同步代码
```

```
async function async1() {
  console.log('async1 start')
  await async2();
  console.log('async1 end')
}
async function async2() { console.log('async2') }
console.log('script start')
setTimeout(function () { console.log('setTimeout') }, 0)
async1();
new Promise(function (resolve) {
  console.log('promise1')
  resolve();
}).then(function () {
  console.log('promise2')
})
console.log('script end')
// await在async异步函数中，会造成函数后续执行的暂停，直到await 的 Promise 解决为止，因此后续代码会直接归属到微任务队列中
// script start、async1 start、async2、promise1、script end、async1 end、promise2、setTimeout
```

```
setTimeout(function () {
  console.log('setTimeout1');
  new Promise(function (resolve) {
    resolve();
  }).then(function () {
    new Promise(function (resolve) {
      resolve();
    }).then(function () {
      console.log('then4');
    });
    console.log('then2');
  });
});
new Promise(function (resolve) {
  console.log('promise1'); resolve();
}).then(function () {
  console.log('then1')
});
setTimeout(function () {
console.log('setTimeout2') });
console.log(2);
queueMicrotask(() => {
console.log('queueMicrotask1') });
new Promise(function (resolve) { resolve()
}).then(function () {
  console.log('then3');
});
//  promise1 2 then1 queueMicrotask1 then3
setTimeout1 then2 then4 setTimeout2
```

```
const $inner =
document.querySelector('#inner')
const $outer =
document.querySelector('#outer')

function handler () {
  console.log('click') // 直接输出

  Promise.resolve().then(_ =>
console.log('promise')) // 注册微任务

  setTimeout(() =>
console.log('timeout')) // 注册宏任务

  requestAnimationFrame(_ =>
console.log('animationFrame')) // 注册宏任务

  $outer.setAttribute('data-random',
Math.random()) // DOM属性修改，触发微任务
}

new MutationObserver(_ => {
  console.log('observer')
}).observe($outer, {
  attributes: true
})

$inner.addEventListener('click',
handler)
$outer.addEventListener('click',
handler)
```

执行熟顺序：click -> promise -> observer -> click -> promise -> observer -> animationFrame -> animationFrame -> timeout -> timeout

疑问：为什么不是 click click，同步放到微任务以后为什么

cookie

会话 Cookie：默认情况下，Cookie 是会话级别的，浏览器关闭后，Cookie 会自动失效

持久 Cookie：如果设置了 expires 或 max-age 属性，则 Cookie 可以被持久保存，即使关闭浏览器，直到到达指定的过期时间后才失效
- expires：设置的是Date.toUTCString()，设置格式是;expires=date-in-GMTString-format
- max-age：设置过期的秒钟，max-age=max-age-in-seconds (例如一年为60*60*24*365)

Cookie 与特定的域名和路径相关联，因此它们只能被同一域名下的页面访问，确保数据的安全性

可以主动设置哪些主机可以接受cookie
- Domain：指定哪些主机可以接受cookie
  如果不指定，那么默认是 origin，不包括子域名
  如果指定Domain，则包含子域名。例如，如果设置 Domain=mozilla.org，则 Cookie 也包含在子域名中（如developer.mozilla.org），developer是子域名
- Path：指定主机下哪些路径可以接受cookie
  例如，设置 Path=/docs，则以下地址都会匹配：/docs、/docs/Web/、/docs/Web/https
  通过浏览器左上角的i符号或者DevTools中Application的Cookies选项(位于Storage列表中)，能够查看对应的cookie

删除：document.cookie="name='coderwhy';max-age=0"，首先选择我们所想要删除的内容，然后将其过期时间设置为0，相当于马上过期，则值就会默认转为undefined，实现"删除"效果

缺点
- 大小只能为 4kb
- 会附加到每一次 http 请求中
- 明文传输不安全
- 多平台无法设置 cookie 等

localStorage（存储上限为5-10MB）

1、数据被写入到浏览器缓存数据库（通常是基于文件的存储），并且持久保存在设备的硬盘中
2 、浏览器会将这些数据存储在其特定的存储目录中，因此即使用户关闭浏览器，数据也不会丢失

以通过浏览器的内置存储管理来实现同一个源下的所有页面中保持一致的数据，所以是跨会话、跨标签页的

数据会持久化到磁盘中，读写时相对较慢，尤其是当存储的数据量较大时，磁盘 I/O 的开销会影响性能

多个标签页属于同一个域名，它们可以读取并共享相同的 localStorage 数据

sessionStorage

1 、sessionStorage 中的数据是与浏览器标签页或者窗口实例相关联的。
2 、浏览器为每个打开的标签页（或窗口）分配一个独立的 session context，并在该上下文中存储数据。
3 、这些数据不会被持久化到硬盘，而是保存在内存中或类似于内存的临时存储中

数据的作用域仅限于当前标签页或窗口，同一网站在不同的标签页中拥有各自独立的 sessionStorage 实例，彼此之间的数据无法共享

indexDB（上百MB存储上限）

创建并且打开 indexedDB.open

每一次操作，都归纳为一个事务对象，作为数据库的一次操作，保证数据库的一致性，而db.transaction() 方法用于在数据库中创建一个事务

```
第一次打开/或者版本发生升级
dbRequest.onupgradeneeded = function (event) {
  const db = event.target.result

  console.log(db)

  // 创建一些存储对象
  db.createObjectStore("users", { keyPath: "id" })
}
```

```
// 创建事物 奥要操作 users
const transaction = db.transaction("users", "readwrite")
 // readwrite表示可以都写还有个 readonly
console.log(transaction)
// 过事务获取对象存储，即对数据库中的 "users" 对象存储进行操作，返回的 store 是对象存储的引用，之后可以通过这个引用来进行数据的增、删、改、查等操作
const store = transaction.objectStore("users")
```

store通过add方法进行增添数据

```
for (const user of users) {
  const request = store.add(user)
  request.onsuccess = function() {
    console.log(`${user.name}插入成功`)
  }
}
transaction.oncomplete = function() {
  console.log("添加操作全部完成")
}
```

查询 更新删除等

通过主键

```
const request = store.openCursor()//打开一个游标
request.onsuccess = function(event) {
  const cursor = event.target.result
  if (cursor) {
    if (cursor.key === 101) {
      console.log(cursor.key, cursor.value)
    } else {
      cursor.continue()
    }
  } else {
    console.log("查询完成")
  }
}
```

通过 store.openCursor()打开一个游标