await-async-事件循环

王红元 coderwhy

异步函数 async function

- async关键字用于声明一个异步函数：
  - async是asynchronous单词的缩写，异步、非同步；
  - sync是synchronous单词的缩写，同步、同时；
- async异步函数可以有很多中写法：

```
async function foo1() {
}

const foo2 = async function() {
}

const foo3 = async () => {
}

class Person {
  async foo() {
  }
}
```

异步函数的执行流程

- 异步函数的内部代码执行过程和普通的函数是一致的，默认情况下也是会被同步执行。

- **异步函数有返回值时，和普通函数会有区别：**
  - 情况一：异步函数也可以有返回值，但是异步函数的返回值会被包裹到Promise.resolve中；
  - 情况二：如果我们的异步函数的返回值是Promise，Promise.resolve的状态会由Promise决定；
  - 情况三：如果我们的异步函数的返回值是一个对象并且实现了thenable，那么会由对象的then方法来决定；

- 如果我们在async中抛出了异常，那么程序它并不会像普通函数一样报错，而是会作为Promise的reject来传递；

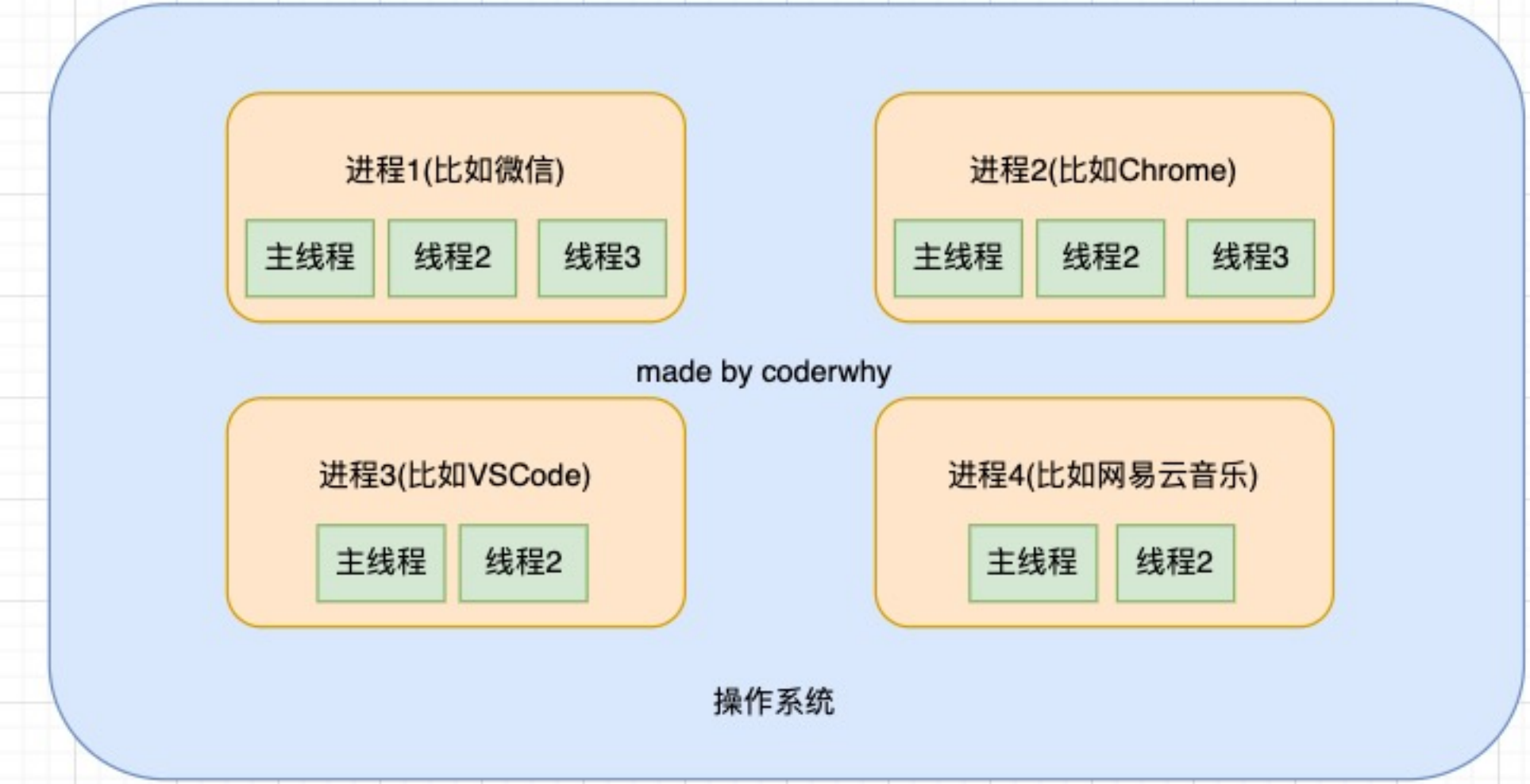
await关键字

- async函数另外一个特殊之处就是可以在它内部使用await关键字，而普通函数中是不可以的。
- await关键字有什么特点呢？
  - 通常使用await是后面会跟上一个表达式，这个表达式会返回一个Promise；
  - 那么await会等到Promise的状态变成fulfilled状态，之后继续执行异步函数；
- 如果await后面是一个普通的值，那么会直接返回这个值；
- 如果await后面是一个thenable的对象，那么会根据对象的then方法调用来决定后续的值；
- 如果await后面的表达式，返回的Promise是reject的状态，那么会将这个reject结果直接作为函数的Promise的reject值；

进程和线程

- **线程和进程是操作系统中的两个概念：**
  - 进程（process）：计算机已经运行的程序，是操作系统管理程序的一种方式；
  - 线程（thread）：操作系统能够运行运算调度的最小单位，通常情况下它被包含在进程中；
- **听起来很抽象，这里还是给出我的解释：**
  - 进程：我们可以认为，启动一个应用程序，就会默认启动一个进程（也可能是多个进程）；
  - 线程：每一个进程中，都会启动至少一个线程用来执行程序中的代码，这个线程被称之为主线程；
  - 所以我们也可以说进程是线程的容器；
- **再用一个形象的例子解释：**
  - 操作系统类似于一个大工厂；
  - 工厂中里有很多车间，这个车间就是进程；
  - 每个车间可能有一个以上的工人在工厂，这个工人就是线程；

操作系统 – 进程 – 线程


进程1(比如微信)
主线程
线程2
线程3
进程2(比如Chrome)
主线程
线程2
线程3
made by coderwhy
进程3(比如VSCode)
主线程
线程2
进程4(比如网易云音乐)
主线程
线程2
操作系统

操作系统的工作方式

- **操作系统是如何做到同时让多个进程（边听歌、边写代码、边查阅资料）同时工作**呢？
  - 这是因为CPU的运算速度非常快，它可以快速的在多个进程之间迅速的切换；
  - 当我们进程中的线程获取到时间片时，就可以快速执行我们编写的代码；
  - 对于用户来说是感受不到这种快速的切换的；
- 你可以在Mac的活动监视器或者Windows的资源管理器中查看到很多进程：

进程名称	% CPU	CPU时间	线程	闲置唤醒	% GPU	GPU时间	PID	用户
预览	0.0	1:06.84	5	0	0.0	1.07	22238	coderwhy
通知中心	0.0	13.80	4	0	0.0	0.11	1157	coderwhy
访达	0.2	12:38.73	11	1	0.0	0.30	1114	coderwhy
聚焦	0.1	1:34.28	6	0	0.0	0.02	1154	coderwhy
网易云音乐	0.0	14:02.49	19	1	0.0	0.01	51482	coderwhy
程序坞	0.5	7:52.11	6	12	0.0	0.03	1112	coderwhy
活动监视器	16.5	4.77	10	2	0.0	0.00	85180	coderwhy
搜狗输入法	0.1	19:59.60	5	0	0.0	0.00	1179	coderwhy
微信	37.5	2:50:05.87	60	65	0.2	5:51.12	27788	coderwhy
小程序	0.0	22.70	14	1	0.0	0.00	27808	coderwhy
备忘录	0.0	3:12.88	4	0	0.0	1.17	74893	coderwhy

浏览器中的JavaScript线程

- 我们经常会说**JavaScript是单线程**的，但是**JavaScript的线程应该有自己的容器进程**：浏览器或者Node。
- **浏览器是一个进程吗，它里面只有一个线程吗？**
  - 目前多数的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，这是为了防止一个页面卡死而造成所有页面无法响应，整个浏览器需要强制退出；
  - 每个进程中又有很多的线程，其中包括执行JavaScript代码的线程；
- **JavaScript的代码执行是在一个单独的线程中执行的：**
  - 这就意味着JavaScript的代码，在同一个时刻只能做一件事；
  - 如果这件事是非常耗时的，就意味着当前的线程就会被阻塞；
- **所以真正耗时的操作，实际上并不是由JavaScript线程在执行的：**
  - 浏览器的每个进程是多线程的，那么其他线程可以来完成这个耗时的操作；
  - 比如网络请求、定时器，我们只需要在特性的时候执行应该有的回调即可；

浏览器的事件循环

- 如果在执行JavaScript代码的过程中，有异步操作呢？
  - 中间我们插入了一个setTimeout的函数调用；
  - 这个函数被放到入调用栈中，执行会立即结束，并不会阻塞后续代码的执行；

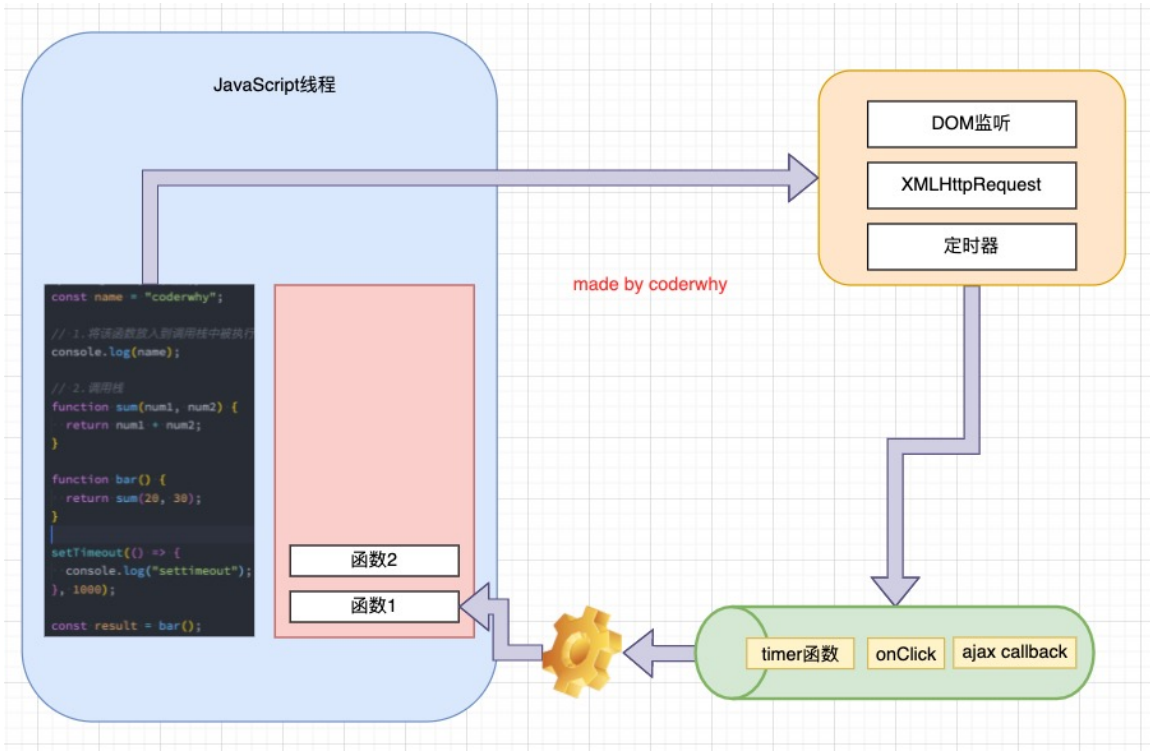
```
function sum(num1, num2) {
  return num1 + num2;
}

function bar() {
  return sum(20, 30);
}

setTimeout(() => {
  console.log("settimeout");
}, 1000);

const result = bar();

console.log(result);
```



宏任务和微任务

- **但是事件循环中并非只维护着一个队列，事实上是有两个队列：**
  - 宏任务队列（macrotask queue）：ajax、setTimeout、setInterval、DOM监听、UI Rendering等
  - 微任务队列（microtask queue）：Promise的then回调、 Mutation Observer API、queueMicrotask()等
- **那么事件循环对于两个队列的优先级是怎么样的呢？**
  - 1.main script中的代码优先执行（编写的顶层script代码）；
  - 2.在执行任何一个宏任务之前（不是队列，是一个宏任务），都会先查看微任务队列中是否有任务需要执行
    - ✓ 也就是宏任务执行之前，必须保证微任务队列是空的；
    - ✓ 如果不为空，那么就优先执行微任务队列中的任务（回调）；
- 下面我们通过几到面试题来练习一下。

Promise面试题

```
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");
});
```

promise async await 面试题

```
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')
```

Promise较难面试题

```
Promise.resolve().then(() => {
  console.log(0);
  return Promise.resolve(4)
}).then((res) => {
  console.log(res)
})


Promise.resolve().then(() => {
  console.log(1);
}).then(() => {
  console.log(2);
}).then(() => {
  console.log(3);
}).then(() => {
  console.log(5);
}).then(() =>{
  console.log(6);
})
```