【前端面试】JS如何实现一个sleep函数?

前言

我们都是 JavaScript 是一个单线程语言，单线程有它的好处也有它的坏处。在我们熟知的如 Java、C++等语言中，都提供了一个叫做 Sleep 的内置函数。这个函数的作用就和它的名字一样：**睡眠**。

假设我们有这样一个场景：我们需要在项目运行起来后的十分钟之后去执行一段代码，这段代码可以是符合你业务场景的任何代码，比如查看内存占用多少等等。

在 Java 这类语言中，可以直接使用 Sleep 这个内置函数实现这个需求，Sleep 函数会让出或者停下当前线程，让其它程序先执行，到底指定时间后在继续执行。

然而我们的 JavaScript 没有提供 sleep 内置函数，大致就是因为单线程的原因把！所以说我们可以尝试着自己封装一个！

1.目标分析

既然我们要去实现一个 sleep 函数，那么我们肯定要先有一个比较实际的场景，这样才好开展工作。

假设我们有如下一段代码：

```
<script>
  function fnA() {
    console.log('A');
  }
  function fnB() {
    console.log('B');
  }
  function fnC() {
    console.log('C');
  }

  // 实现目标
  fnA(); // 1 秒后打印
  fnB(); // 2 秒后打印
  fnC(); // 3 秒后打印
</script>
```

上段代码非常简单，我们的目的就是为了让它们几个分别间隔 1 秒打印，需求非常简单，实现起来也很容易，可能有些小伙伴直接想到了 setTimeout。

确实，setTimeout 可以实现我们的需求，比如下面的代码：

```
setTimeout(fnA, 1000);
setTimeout(fnB, 2000);
setTimeout(fnC, 3000);
```

定时器确实可以满足我们的需求，但是如果项目中到处些定时器的可能会让人很疑惑，所以我们有必要进行封装，写一个自己的 sleep 函数，大家多看几种实现方式应该就会豁然开朗了。

2.setTimeout 封装

这是大家最容易想到也是最暴力的一种方式，毕竟一提到延时执行大家都会想到定时器，所我们直接利用 setTimeout 的这个特性来实现我们的 sleep 函数。

代码如下：

```
<script>
  function fnA() {
    console.log('A');
  }
  function fnB() {
    console.log('B');
  }
  function fnC() {
    console.log('C');
  }
  // sleep 函数
  function sleep(fun, time) {
    setTimeout(() => {
      fun();
    }, time);
  }
  sleep(fnA, 1000); // 1 秒后输出 A
  sleep(fnB, 2000); // 2 秒后输出 B
  sleep(fnC, 3000); // 3 秒后输出 C
</script>
```

这是最原始的一种方式，其实本质就是定时器，只不过我们封装成一个函数罢了。

这种实现方式有如下优缺点：

优点：

简单易实现，兼容性好，毕竟只是用了 setTimeout，而且非常好理解。

缺点：

我们需要传入回调函数的方式进去，如果函数里面有多回调函数可能不太好理解。另外一点就是它不会阻塞同步任务，比如下面代码的输出结果：

```
sleep(fnA, 1000);
console.log('E');
sleep(fnB, 2000);
console.log('G');
sleep(fnC, 3000);
```

输出结果：

```
E
G
A
B
C
>
```

在有些场景下我们可能需要 sleep 函数会阻塞代码，依次执行，这个时候这种封装就满足不了了。

3.Promise 封装

promise 是 ES6 提出的一种异步解决方案，它和 setTimeout 一样，都可以实现异步，区别在于 promise 解决了回调函数的问题，它可以实现链式调用，我们可以接触 promise 来实现 sleep 函数。

代码如下：

```
<script>
  function fnA() {
    console.log('A');
  }
  function fnB() {
    console.log('B');
  }
  function fnC() {
    console.log('C');
  }

  // sleep 函数--Promise 版本
  function sleep(time) {
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve();
      }, time);
    });
  }
  sleep(1000).then(fnA); // 1 秒后输出 A
  sleep(2000).then(fnB); // 2 秒后输出 B
  sleep(3000).then(fnC); // 3 秒后输出 C
```

```
</script>
```

上段代码中利用 promise 实现了 sleep 函数，其实是 promise 和 setTimeout 的结合，不过上段代码中我们可以进行链式调用了，不必再往 sleep 函数中传入回调函数了。

优点：

不用再传入回调函数，采用链式调用。

缺点：

仍未解决阻塞问题，依然会先执行同步任务，**代码如下：**

```
sleep(1000).then(fnA); // 1 秒后输出 A
console.log('E');
sleep(2000).then(fnB); // 2 秒后输出 B
console.log('G');
sleep(3000).then(fnC); // 3 秒后输出 C
```

输出结果：

```
E
G
A
B
C
>
```

4.async/await

前面两个封装中我们一直提及阻塞问题，那么既然我们使用了 promise，我们就很有必要将 async/await 拿出来使用，它们可以完美的阻塞我们的代码，然我们的代码依次执行。

代码如下：

```
<script>
  function fnA() {
    console.log('A');
  }
  function fnB() {
    console.log('B');
  }
  function fnC() {
    console.log('C');
  }
  // sleep 函数--Promise 版本
  function sleep(time) {
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve();
```

```
      }, time);
    });
  }
  async function sleepTest() {
    fnA();                // 输出 A
    await sleep(1000);    // 睡眠 1 秒
    console.log('E');     // 输出 E
    fnB();                // 输出 B
    await sleep(1000);    // 睡眠 1 秒
    fnC();                // 输出 C
    await sleep(1000);    // 睡眠 1 秒
    console.log('G');     // 输出 G
  }
  sleepTest();
</script>
```

输出结果：

```
A
E
B
C
G
>
```

上段代码中我们封装的 sleep 函数并没有改变，只是我们调用 sleep 函数的使用采用了 async/await 的方式调用，这就很好的解决了我们程序没有阻塞的额问题了。

总结

实现 sleep 函数其实非常简单，主要是理解 JavaScript 中异步执行情况。虽然上面的代码中使用 await sleep()的方式看起来最像一个真正的 sleep 函数，但是凡事都有两面性，比如我们有些场景只是需要一定时间后执行某个函数，不想阻塞代码的执行，这个时候 setTimeout 和 promise 都是非常好的选择，但有时候我们就是需要阻塞代码的执行，比如后面的代码用到了前面这个函数的返回结果，这个时候 async/await 就是一个很好的选择了。