面试官：讲一下你对 WebWorker 的理解

前言

提到 WebWorker，可能有些小伙伴比较陌生，不知道是做什么的，甚至不知道使用场景，今天这篇文章就带大家一起简单了解一下什么是 webworker!

概念

WebWorker 实际上是运行在浏览器后台的一个 `单独的线程` ，因此可以执行一些耗时的操作而不会阻塞主线程。WebWorker 通过与主线程之间传递消息实现通信，这种通信是 `双向的` 。WebWorker 不能直接访问 DOM，也不能使用像 window 对象这样的浏览器接口对象，但可以使用一些WebWorker 标准接口和 Navigator 对象的部分属性和方法。

为什么使用WebWorker?

1. **提高应用响应能力：** 主线程被独占执行一些 `耗时` 的计算或操作时，会导致 UI `无响应` 。WebWorker 可以把这些任务 `转移到后台线程` ，从而保证主线程的运行，提高应用的响应能力。
2. **充分利用多核CPU：** 现代 CPU 都是多核的，WebWorker 可以让 Web 应用利用多核 `CPU 的并行计算能力` ，充分发挥计算机硬件性能。
3. **避免渲染阻塞：** JavaScript 运行在 `主线程` ，如果主线程一直 `占用` ，就无法执行 GUI 渲染任务，导致界面 `渲染受阻` 。WebWorker 把一些费时任务 `分流到后台线程` 就可以避免这一问题。
4. **后台持续运行：** WebWorker 所在后台线程可持续运行，即使页面被挂起或最小化，任务仍在后台执行，非常适合一些需要长时间运行的操作。

使用场景

一般来说,当遇到如下几种情况时可以考虑使用 WebWorker:

1. **大量数据的计算/处理：** 比如图像处理、数据分析等
2. **长时间运行的操作：** 如一些复杂的数学计算
3. **非阻塞式操作：** 希望执行一些耗时操作时不阻塞主线程

示例

假设我们有一个需要计算斐波那契数列的任务，我们可以使用 Web Worker 来进行计算，以避免阻塞主线程。以下是一个简单的示例：

主线程

主线程创建 worker 实例，通过 **postMessage 向worker发送消息**，通过 **onmessage 监听 worker 返回**的数据。

```
1 js
2 复制代码
3       const myWorker = new Worker('./worker.js')      myWorker.onmessage =
  function (e) {        console.log('Fibonacci result:', e.data)      }
  myWorker.postMessage(40) // 请求计算斐波那契数列的第40项
```

worker.js

在同级目录下创建 `worker.js` 文件，通过 `onmessage` 接收主线程发来的数据，计算后通过 `postMessage` 将计算结果返回主线程。

```
1 js
2 复制代码
3     self.onmessage = function (e) {      const n = e.data      let a = 0,
    b = 1,        temp      for (let i = 2; i <= n; i++) {        temp = a
   a = b        b = temp + b      }      self.postMessage(b)    }
```

运行结果

可以看到主线程打印出 worker 计算的运行结果

Vue、React项目使用

接下来为大家演示 vue 以及 react 项目如何使用

Vue使用

vue版本："vue": "^2.6.14"，vue-cli版本：@vue/cli 5.0.8

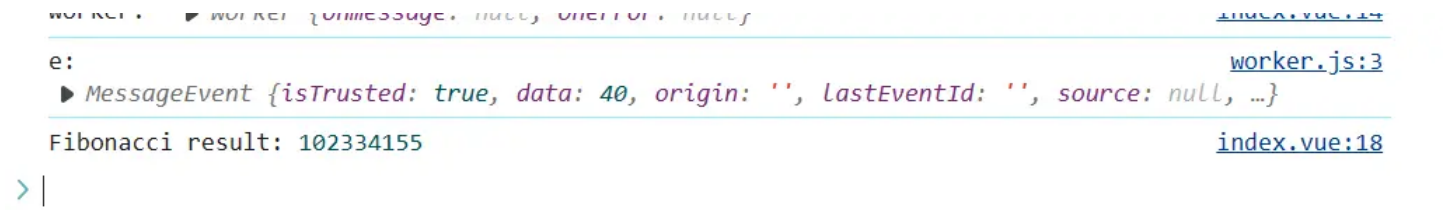
页面使用

```
1 js
2 复制代码
3 <template><div我的页面</div</template><scriptexport default {  name:
  'MyselfView',  data() {    return {      worker: null,    }  },  mounted() {
   // 创建 WebWorker 实例   this.worker = new Worker(new URL('./worker.js',
  import.meta.url))    console.log('worker: ', this.worker)
  this.worker.postMessage(40) // 请求计算斐波那契数列的第40项
  this.worker.addEventListener('message', (event) => {
  console.log('Fibonacci result:', event.data)    })  },  beforeDestroy() {
  // 组件销毁时终止 WebWorker    this.worker.terminate()  },}</script
```

worker.js

```
1 js
2 复制代码
3 // worker.jsself.addEventListener('message', (e) => {  console.log('e: ', e)
  const n = e.data  let a = 0,    b = 1,    temp  for (let i = 2; i <= n; i++) {
     temp = a    a = b    b = temp + b  }  self.postMessage(b)})
```

效果



React使用

react版本： "react": "^18.2.0"

```
1 js
2 复制代码
3 import React, { useEffect } from 'react'const Demo = () => {  useEffect(() =>
  {    const worker = new Worker(new URL('./worker.worker.js', import.meta.url))
     worker.onmessage = function (e) {      console.log('Fibonacci result:',
  e.data)    }    worker.postMessage(40) // 请求计算斐波那契数列的第40项    // 使用
  worker ...    return () => worker.terminate()  }, []) return (    <div
  <pcount的值</p    </div  )
```

效果

```
Fibonacci result: 102334155                                    about.tsx:35
```

注意

由于我们在项目开发时，使用不同的打包工具(`vite/webpack`)。幸运的是，最新版的 `vite/webpack` 都支持 `Web Worker` 了。

我们可以通过： `new URL()` 的方式 -- `vite/webpack` 都支持

```
1 js
2 复制代码
3 new Worker(  new URL(    './worker.js',     import.meta.url  ));
```

总结

WebWorker是一种在 Web 应用中实现 `多线程运行的技术` ，可以将耗费 CPU 的任务交给 `后台线程处理` ， `避免阻塞主线程` ，从而提高 Web 应用的响应性能和用户体验。 总之，WebWorker的引入解决了 Web 应用长期以来在 `单个线程` 中运行所带来的诸多问题，有效提升了Web应用的运行性能和用户体验。