23. async、await 的底层运行机制

1. 核心概念 (Core Concept)

`async` 和 `await` 是 JavaScript 中用于更简洁地处理异步操作的语法糖 (syntactic sugar)，它们建立在 Promises 和生成器 (Generators) 的基础之上，使得异步代码的书写和阅读更接近同步代码的风格。其核心作用是暂停 `async` 函数的执行，直到一个 Promise 解析（resolved）并返回其值，或拒绝（rejected）并抛出错误。

2. 为什么需要它？ (The "Why")

1. **代码可读性与可维护性:** 相较于传统的 Promise 的 `.then()` 方法链或回调函数，`async` / `await` 结构的代码流程更加线性，更容易理解异步操作的顺序和依赖关系，降低了“回调地狱”的复杂性。
2. **错误处理:** `async` / `await` 可以配合标准的 `try...catch` 语句来处理异步操作中的错误，这比 Promise 的 `.catch()` 或 `.finally()` 更加直观，与同步代码的错误处理方式保持一致。
3. **更好的调试体验:** 在许多现代调试工具中，使用 `async` / `await` 编写的异步代码的调用栈更清晰，更容易追踪问题。

3. API 与用法 (API & Usage)

`async` 关键字用于声明一个函数是异步函数。异步函数总是返回一个 Promise。如果函数返回一个非 Promise 的值，这个值会被自动封装在一个 resolved 的 Promise 中。

`await` 关键字只能在 `async` 函数内部使用。它用于等待一个 Promise 的解析。当遇到 `await` 表达式时：

- 如果 `await` 的操作数是一个 Promise， `async` 函数会暂停执行，并将控制权交回给调用栈。当 Promise 解析时，暂停的 `async` 函数会从暂停的地方恢复执行，并 `await` 表达式的值就是 Promise resolve 的值。
- 如果 `await` 的操作数不是一个 Promise，它会被立即解析为它本身的值。

```
// 官方文档示例风格或经典用法

async function fetchData(url) {
  try {
    // await 等待 fetch API 返回的 Promise 解析
    const response = await fetch(url);

    // await 等待 response.json() 返回的 Promise 解析
    const data = await response.json();
```

```
    console.log(data); // 当所有 await 完成后，才会执行到这里
    return data; // async 函数返回一个 Promise，其 resolve 值为 data

  } catch (error) {
    // 使用 try...catch 捕获 await Promise 失败时的错误
    console.error('Fetch failed:', error);
    // async 函数返回的 Promise 会被拒绝
    throw error;
  }
}

// 调用 async 函数
fetchData('https://api.example.com/data')
  .then(result => {
    console.log('Successfully fetched:', result);
  })
  .catch(err => {
    console.error('Error during fetch process:', err);
  });
```

4. 关键注意事项 (Key Considerations)

1. **`await` 必须在 `async` 函数中使用:** `await` 关键字只能出现在 `async` 函数的函数体内部。在顶层作用域（例如模块的根部）直接使用 `await` 是语法错误（除非环境支持顶层 `await`，如现代浏览器模块或 Node.js）。
2. **并非阻塞主线程:** `await` 关键字只是 *暂停* `async` 函数自身的执行，并将控制权交回给事件循环，等待 Promise 决议。它并不会阻塞 JavaScript 所在的整个线程或主线程的事件循环。
3. **顺序执行 vs 并发:** 默认情况下，多个 `await` 操作是顺序执行的。如果要执行并发的异步操作，应该先创建所有的 Promise，然后使用 `Promise.all()` 或 `Promise.allSettled()` 等方法等待它们并行完成后再 `await` 结果数组。

```
// 顺序执行（等待第一个完成后才开始第二个）
async function sequentialFetches() {
  const result1 = await fetchData('/api/data1');
  const result2 = await fetchData('/api/data2'); // 等待 result1 完成后开始
  console.log(result1, result2);
}

// 并发执行（两个 fetch 同时开始）
async function concurrentFetches() {
  const promise1 = fetchData('/api/data1');
  const promise2 = fetchData('/api/data2');

  const [result1, result2] = await Promise.all([promise1, promise2]);
// 等待所有 Promise 完成
```

```
    console.log(result1, result2);
}
```

4. **Error Handling:** 未使用 `try...catch` 包裹的 `await` Promise 拒绝错误（rejected Promise）会在 `async` 函数外部表现为未捕获的 Promise 拒绝（unhandled promise rejection）。总是建议在 `async` 函数内部使用 `try...catch` 或在调用 `async` 函数后使用 `.catch()` 来处理可能的错误。

5. 参考资料 (References)

- **MDN Web Docs:** `async function`
- **MDN Web Docs:** `await`
- **JavaScript.info:** Async/await (提供不错的原理讲解)
- **Google Developers Blog:** Async functions - making asynchronous programming easier (Chrome 工程师关于 `async` / `await` 的介绍)