

await、async、事件循环

王红元 coderwhy

目录

content



1 **async、await**

2 **浏览器进程、线程**

3 **宏任务、微任务队列**

4 **Promise面试题解析**

5 **throw、try、catch**

6 **浏览器存储Storage**

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相当于被包裹到Promise.resolve中；
 - 情况二：如果我们的异步函数的返回值是Promise，状态由会由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；

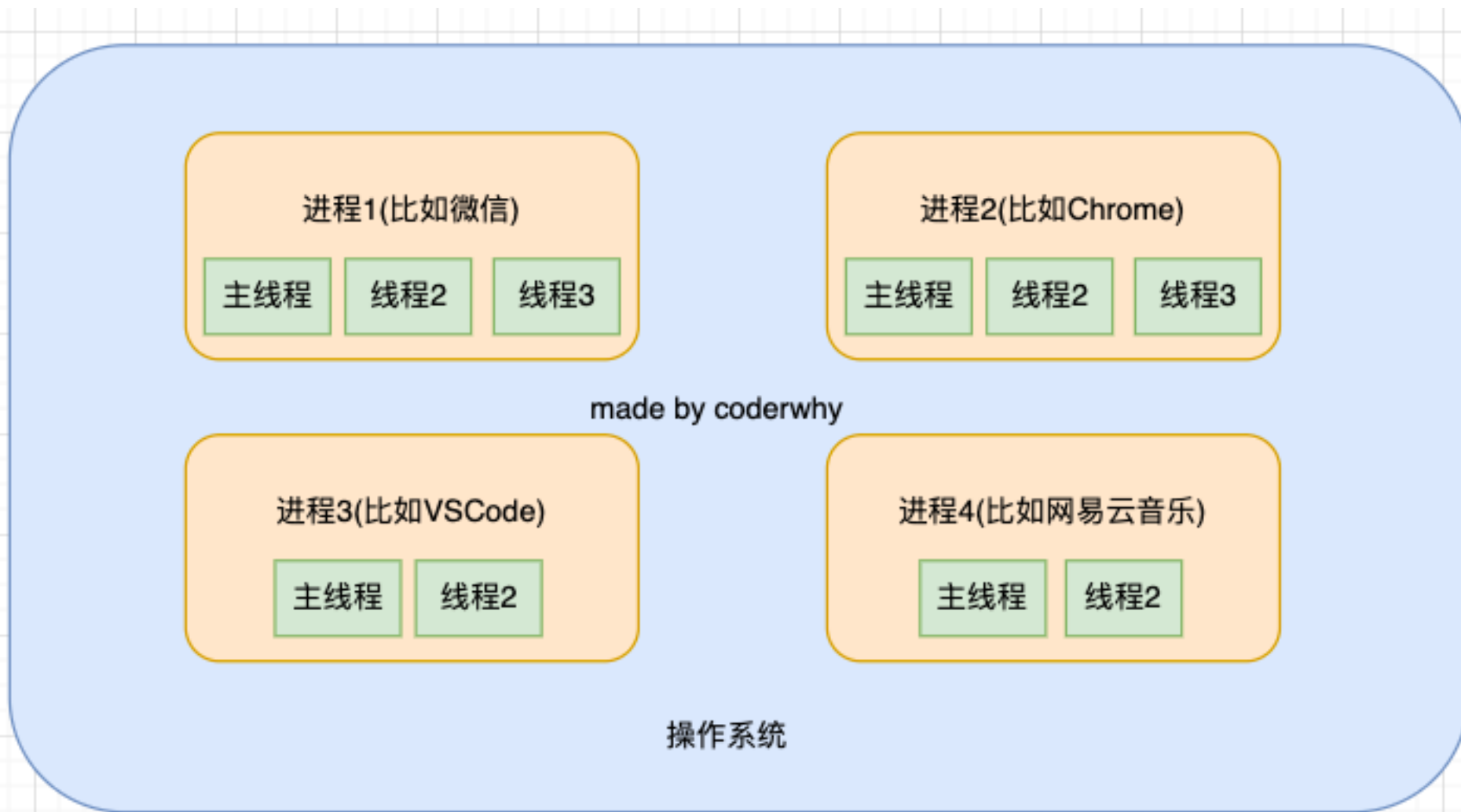
■ 听起来很抽象，这里还是给出我的解释：

- 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



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：

活动监视器 (我的进程)									
CPU 内存 能耗 磁盘 网络									
Q 搜索									
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

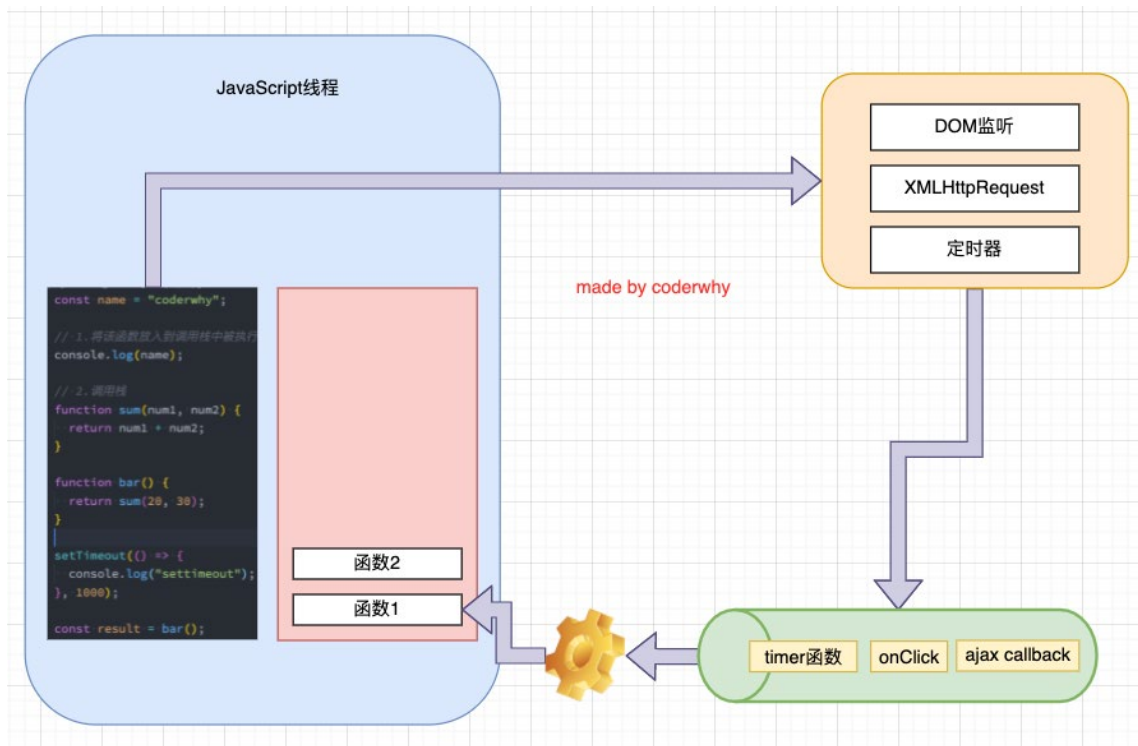
- 我们经常会说JavaScript是单线程（可以开启workers）的，但是JavaScript的线程应该有自己的容器进程：浏览器或者Node。
- 浏览器是一个进程吗，它里面只有一个线程吗？
 - 目前多数的浏览器其实都是多进程的，当我们打开一个tab页面时就会开启一个新的进程，这是为了防止一个页面卡死而造成所有页面无法响应，整个浏览器需要强制退出；
 - 每个进程中又有很多的线程，其中包括执行JavaScript代码的线程；
- JavaScript的代码执行是在一个单独的线程中执行的：
 - 这就意味着JavaScript的代码，在同一个时刻只能做一件事；
 - 如果这件事是非常耗时的，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子的呢？

□ 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");  
});
```

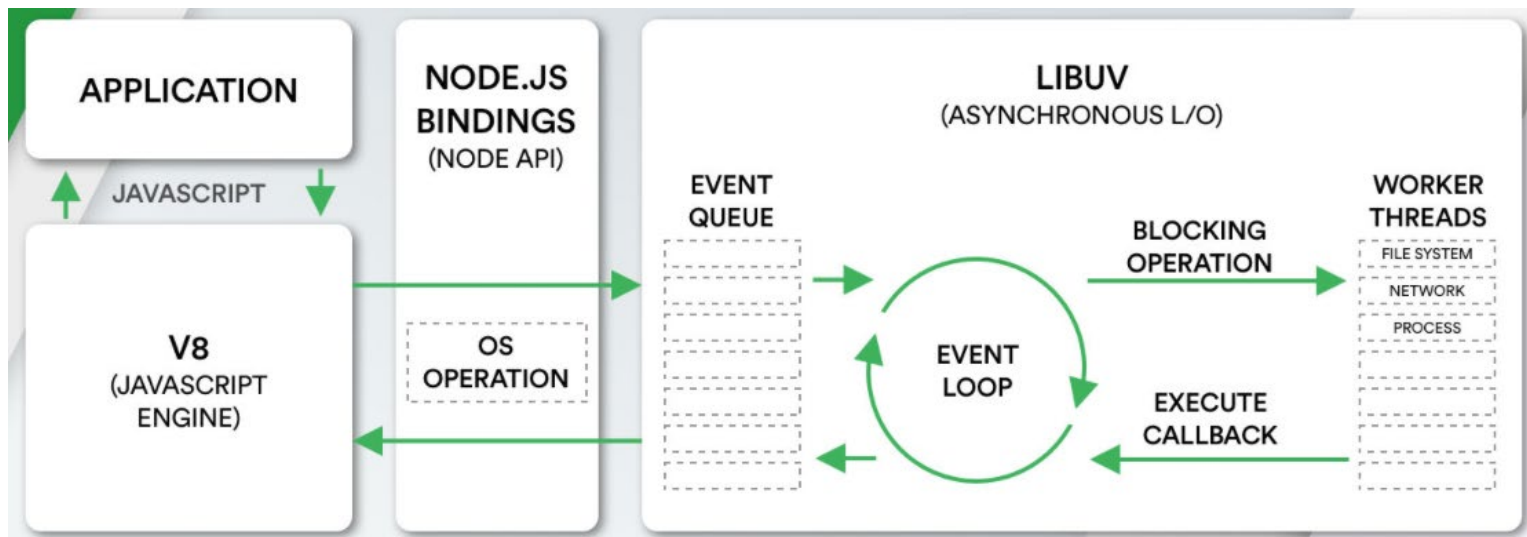
promise async await 面试题

```
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')
```

Node的事件循环

- 浏览器中的EventLoop是根据HTML5定义规范来实现的，不同的浏览器可能会有不同的实现，而Node中是由libuv实现的。
- 这里我们来给出一个Node的架构图：
 - 我们会发现libuv中主要维护了一个EventLoop和worker threads（线程池）；
 - EventLoop负责调用系统的一些其他操作：文件的IO、Network、child-processes等
- libuv是一个多平台的专注于异步IO的库，它最初是为Node开发的，但是现在也被使用到Luvit、Julia、pyuv等其他地方；



Node事件循环的阶段

- 我们最前面就强调过，**事件循环**像是一个桥梁，是连接着应用程序的JavaScript和系统调用之间的通道：
 - 无论是我们的文件IO、数据库、网络IO、定时器、子进程，在完成对应的操作后，都会将**对应的结果和回调函数**放到事件循环（任务队列）中；
 - 事件循环会不断的从**任务队列中取出对应的事件（回调函数）**来执行；
- 但是一次完整的事件循环Tick分成很多个阶段：
 - **定时器（Timers）**：本阶段执行已经被 setTimeout() 和 setInterval() 的调度回调函数。
 - **待定回调（Pending Callback）**：对某些系统操作（如TCP错误类型）执行回调，比如TCP连接时接收到ECONNREFUSED。
 - **idle, prepare**：仅系统内部使用。
 - **轮询（Poll）**：检索新的 I/O 事件；执行与 I/O 相关的回调；
 - **检测（check）**：setImmediate() 回调函数在这里执行。
 - **关闭的回调函数**：一些关闭的回调函数，如：socket.on('close', ...).

Node事件循环的阶段图解





Node的宏任务和微任务

- 我们会发现从一次事件循环的Tick来说，Node的事件循环更复杂，它也分为微任务和宏任务：
 - 宏任务 (macrotask) : setTimeout、setInterval、IO事件、setImmediate、close事件;
 - 微任务 (microtask) : Promise的then回调、process.nextTick、queueMicrotask;
- 但是，Node中的事件循环不只是 微任务队列和 宏任务队列：
 - 微任务队列：
 - ✓ next tick queue: process.nextTick;
 - ✓ other queue: Promise的then回调、queueMicrotask;
 - 宏任务队列：
 - ✓ timer queue: setTimeout、setInterval;
 - ✓ poll queue: IO事件;
 - ✓ check queue: setImmediate;
 - ✓ close queue: close事件;

Node事件循环的顺序

■ 所以，在每一次事件循环的tick中，会按照如下顺序来执行代码：

- next tick microtask queue;
- other microtask queue;
- timer queue;
- poll queue;
- check queue;
- close queue;

Node执行面试题

```
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('setTimeout0')  
}, 0)  
  
setTimeout(function () {  
  console.log('setTimeout2')  
}, 300)
```

```
setImmediate(() => console.log('setImmediate'));  
  
process.nextTick(() => console.log('nextTick1'));  
  
async1();  
  
process.nextTick(() => console.log('nextTick2'));  
  
new Promise(function (resolve) {  
  console.log('promise1')  
  resolve();  
  console.log('promise2')  
}).then(function () {  
  console.log('promise3')  
})  
  
console.log('script end')
```

错误处理方案

■ 开发中我们会封装一些工具函数，封装之后给别人使用：

- 在其他人使用的过程中，可能会传递一些参数；
- 对于函数来说，需要对这些参数进行验证，否则可能得到的是我们不想要的结果；

■ 很多时候我们可能验证到不是希望得到的参数时，就会直接return：

- 但是return存在很大的弊端：调用者不知道是因为函数内部没有正常执行，还是执行结果就是一个undefined；
- 事实上，正确的做法应该是如果没有通过某些验证，那么应该让外界知道函数内部报错了；

■ 如何可以让一个函数告知外界自己内部出现了错误呢？

- 通过throw关键字，抛出一个异常；

■ throw语句：

- throw语句用于抛出一个用户自定义的异常；
- 当遇到throw语句时，当前的函数执行会被停止（throw后面的语句不会执行）；

■ 如果我们执行代码，就会报错，拿到错误信息的时候我们可以及时的去修正代码。

throw关键字

- throw表达式就是在throw后面可以跟上一个表达式来表示具体的异常信息：

```
throw expression;
```

- throw关键字可以跟上哪些类型呢？

- **基本数据类型**：比如number、string、Boolean

- **对象类型**：对象类型可以包含更多的信息

- 但是每次写这么长的对象又有点麻烦，所以我们可以创建一个类：

```
class HYError {  
  constructor(errCode, errMsg) {  
    this.errCode = errCode  
    this.errMessage = errMsg  
  }  
}
```

- 事实上，JavaScript已经给我们提供了一个Error类，我们可以直接创建这个类的对象：

```
function foo() {  
  throw new Error("error message", "123")  
}
```

- Error包含三个属性：

- **message**：创建Error对象时传入的message；
- **name**：Error的名称，通常和类的名称一致；
- **stack**：整个Error的错误信息，包括函数的调用栈，当我们直接打印Error对象时，打印的就是stack；

- Error有一些自己的子类：

- RangeError：下标值越界时使用的错误类型；
- SyntaxError：解析语法错误时使用的错误类型；
- TypeError：出现类型错误时，使用的错误类型；

异常的处理

■ 我们会发现在之前的代码中，一个函数抛出了异常，调用它的时候程序会被强制终止：

- 这是因为如果我们在调用一个函数时，这个函数抛出了异常，但是我们并没有对这个异常进行处理，那么这个异常会继续传递到上一个函数调用中；
- 而如果到了最顶层（全局）的代码中依然没有对这个异常的处理代码，这个时候就会报错并且终止程序的运行；

■ 我们先来看一下这段代码的异常传递过程：

- foo函数在被执行时会抛出异常，也就是我们的bar函数会拿到这个异常；
- 但是bar函数并没有对这个异常进行处理，那么这个异常就会被继续传递到调用bar函数的函数，也就是test函数；
- 但是test函数依然没有处理，就会继续传递到我们的全局代码逻辑中；
- 依然没有被处理，这个时候程序会终止执行，后续代码都不会再执行了；

```
function foo() {  
  throw "coderwhy error message"  
}  
  
function bar() {  
  foo()  
}
```

```
function test() {  
  bar()  
}  
  
test()  
console.log("test后续代码~")
```

异常的捕获

■ 但是很多情况下当出现异常时，我们并不希望程序直接推出，而是希望可以正确的处理异常：

□ 这个时候我们就可以使用try catch

```
try {  
  try_statements  
}  
[catch (exception_var_1) {  
  catch_statements_1  
}]  
[finally {  
  finally_statements  
}]
```

```
function foo() {  
  throw "coderwhy error message"  
}  
  
function bar() {  
  try {  
    foo()  
    console.log("foo后续的代码~")  
  } catch (error) {  
    console.log(error)  
  }  
}
```

■ 在ES10 (ES2019) 中，catch后面绑定的error可以省略。

■ 当然，如果有一些必须要执行的代码，我们可以使用finally来执行：

□ finally表示最终一定会被执行的代码结构；

□ 注意：如果try和finally中都有返回值，那么会使用finally当中的返回值；

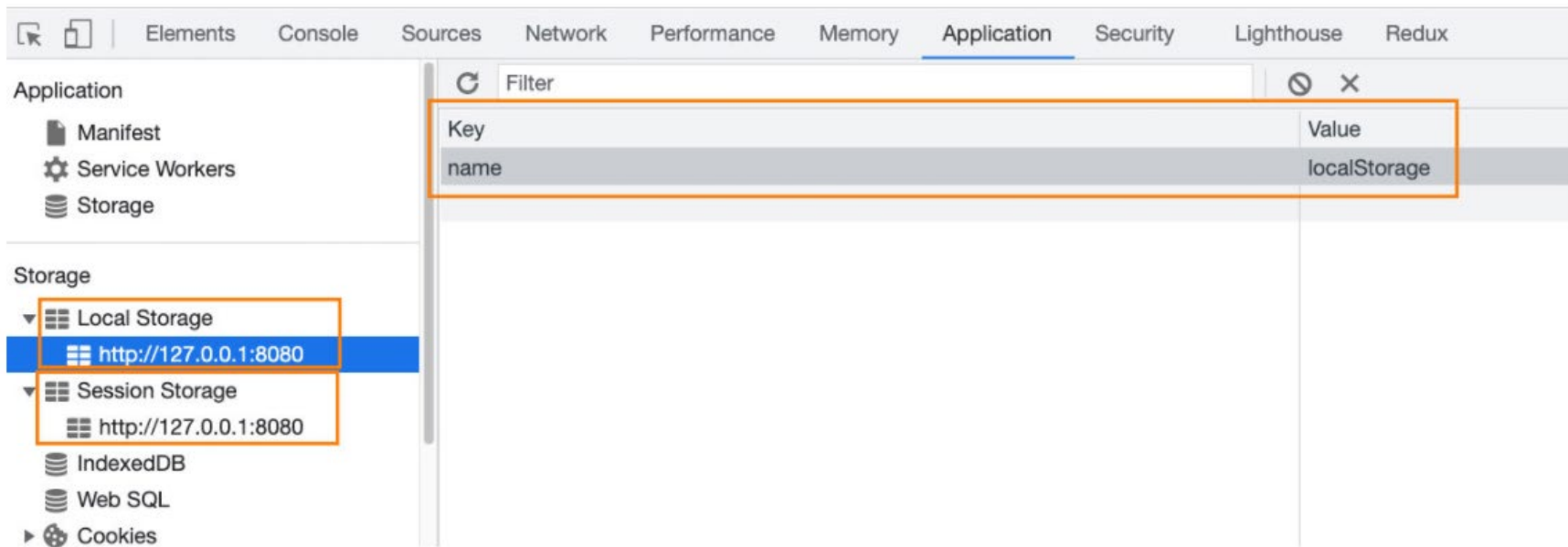
认识Storage

■ WebStorage主要提供了一种机制，可以让浏览器提供一种比cookie更直观的key、value存储方式：

□ localStorage：本地存储，提供的是一种永久性的存储方法，在关闭掉网页重新打开时，存储的内容依然保留；

□ sessionStorage：会话存储，提供的是本次会话的存储，在关闭掉会话时，存储的内容会被清除；

```
localStorage.setItem("name", "localStorage")
sessionStorage.setItem("name", "sessionStorage")
```



localStorage和sessionStorage的区别

- 我们会发现localStorage和sessionStorage看起来非常的相似。
- 那么它们有什么区别呢？
 - 验证一：关闭网页后重新打开，localStorage会保留，而sessionStorage会被删除；
 - 验证二：在页面内实现跳转，localStorage会保留，sessionStorage也会保留；
 - 验证三：在页面外实现跳转（打开新的网页），localStorage会保留，sessionStorage不会被保留；



Storage常见的方法和属性

■ Storage有如下的属性和方法:

■ 属性:

□ **Storage.length**: 只读属性

✓ 返回一个整数, 表示存储在Storage对象中的数据项数量;

■ 方法:

□ **Storage.key(index)**: 该方法接受一个数值n作为参数, 返回存储中的第n个key名称;

□ **Storage.getItem()**: 该方法接受一个key作为参数, 并且返回key对应的value;

□ **Storage.setItem()**: 该方法接受一个key和value, 并且将会把key和value添加到存储中。

✓ 如果key存储, 则更新其对应的值;

□ **Storage.removeItem()**: 该方法接受一个key作为参数, 并把该key从存储中删除;

□ **Storage.clear()**: 该方法的作用是清空存储中的所有key;