- 然后判断输入参数是不是在 `cache` 的中。如果已经存在，直接返回 `cache` 的内容，如果没有存在，使用函数 `func` 对输入参数求值，然后把结果存储在 `cache` 中

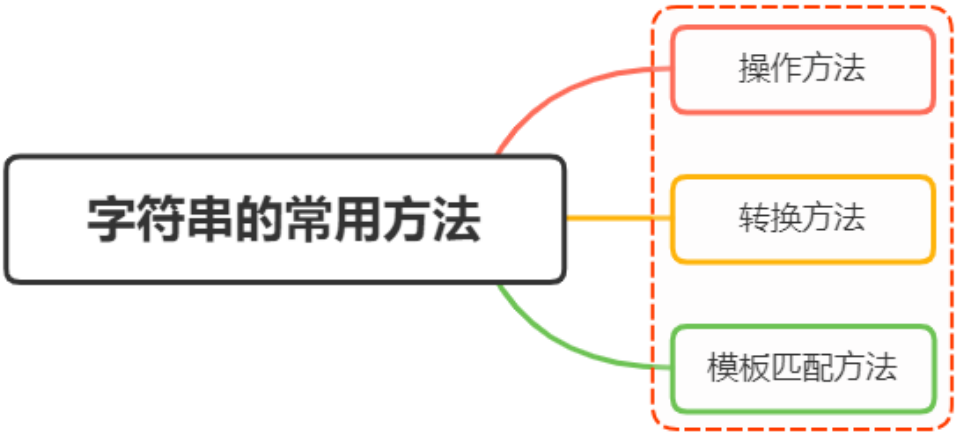
18.3. 应用场景

虽然使用缓存效率是非常高的，但并不是所有场景都适用，因此千万不要极端的将所有函数都添加缓存

以下几种情况下，适合使用缓存：

- 对于昂贵的函数调用，执行复杂计算的函数
- 对于具有有限且高度重复输入范围的函数
- 对于具有重复输入值的递归函数
- 对于纯函数，即每次使用特定输入调用时返回相同输出的函数

19. JavaScript字符串的常用方法有哪些?



19.1. 操作方法

我们也可将字符串常用的操作方法归纳为增、删、改、查，需要知道字符串的特点是一旦创建了，就不可变

19.1.1. 增

这里增的意思并不是说直接增添内容，而是创建字符串的一个副本，再进行操作

除了常用 `+` 以及 `${}` 进行字符串拼接之外，还可通过 `concat`

19.1.1.1. concat

用于将一个或多个字符串拼接成一个新字符串

JavaScript 复制代码

```
let stringValue = "hello ";
let result = stringValue.concat("world");
console.log(result); // "hello world"
console.log(stringValue); // "hello"
```

19.1.2. 删

这里的删的意思并不是说删除原字符串的内容，而是创建字符串的一个副本，再进行操作

常见的有：

- slice()
- substr()
- substring()

这三个方法都返回调用它们的字符串的一个子字符串，而且都接收一或两个参数。

JavaScript 复制代码

```
let stringValue = "hello world";
console.log(stringValue.slice(3)); // "lo world"
console.log(stringValue.substring(3)); // "lo world"
console.log(stringValue.substr(3)); // "lo world"
console.log(stringValue.slice(3, 7)); // "lo w"
console.log(stringValue.substring(3,7)); // "lo w"
console.log(stringValue.substr(3, 7)); // "lo worl"
```

19.1.3. 改

这里改的意思也不是改变原字符串，而是创建字符串的一个副本，再进行操作

常见的有：

- trim()、trimLeft()、trimRight()
- repeat()
- padStart()、padEnd()
- toLowerCase()、 toUpperCase()

19.1.3.1. trim()、trimLeft()、trimRight()

删除前、后或前后所有空格符，再返回新的字符串

JavaScript 复制代码

```
let stringValue = " hello world ";
let trimmedStringValue = stringValue.trim();
console.log(stringValue); // " hello world "
console.log(trimmedStringValue); // "hello world"
```

19.1.3.2. repeat()

接收一个整数参数，表示要将字符串复制多少次，然后返回拼接所有副本后的结果

JavaScript 复制代码

```
let stringValue = "na ";
let copyResult = stringValue.repeat(2) // na na
```

19.1.3.3. padEnd()

复制字符串，如果小于指定长度，则在相应一边填充字符，直至满足长度条件

JavaScript 复制代码

```
let stringValue = "foo";
console.log(stringValue.padStart(6)); // " foo"
console.log(stringValue.padStart(9, ".")); // "......foo"
```

19.1.4. toLowerCase()、 toUpperCase()

大小写转化

JavaScript 复制代码

```
let stringValue = "hello world";
console.log(stringValue.toUpperCase()); // "HELLO WORLD"
console.log(stringValue.toLowerCase()); // "hello world"
```

19.1.5. 查

除了通过索引的方式获取字符串的值，还可通过：

- chatAt()
- indexOf()
- startWith()
- includes()

19.1.5.1. charAt()

返回给定索引位置的字符，由传给方法的整数参数指定

JavaScript 复制代码

```
let message = "abcde";
console.log(message.charAt(2)); // "c"
```

19.1.5.2. indexOf()

从字符串开头去搜索传入的字符串，并返回位置（如果没找到，则返回 -1 ）

JavaScript 复制代码

```
let stringValue = "hello world";
console.log(stringValue.indexOf("o")); // 4
```

19.1.5.3. startWith()、includes()

从字符串中搜索传入的字符串，并返回一个表示是否包含的布尔值

JavaScript 复制代码

```
let message = "foobarbaz";
console.log(message.startsWith("foo")); // true
console.log(message.startsWith("bar")); // false
console.log(message.includes("bar")); // true
console.log(message.includes("qux")); // false
```

19.2. 转换方法

19.2.1. split

把字符串按照指定的分割符，拆分成数组中的每一项

```JavaScript
let str = "12+23+34"
let arr = str.split("+") // [12,23,34]
```

19.3. 模板匹配方法

针对正则表达式，字符串设计了几个方法：

- match()
- search()
- replace()

19.3.1. match()

接收一个参数，可以是一个正则表达式字符串，也可以是一个 `RegExp` 对象，返回数组

```JavaScript
let text = "cat, bat, sat, fat";
let pattern = /.at/;
let matches = text.match(pattern);
console.log(matches[0]); // "cat"
```

19.3.2. search()

接收一个参数，可以是一个正则表达式字符串，也可以是一个 `RegExp` 对象，找到则返回匹配索引，否则返回 –1

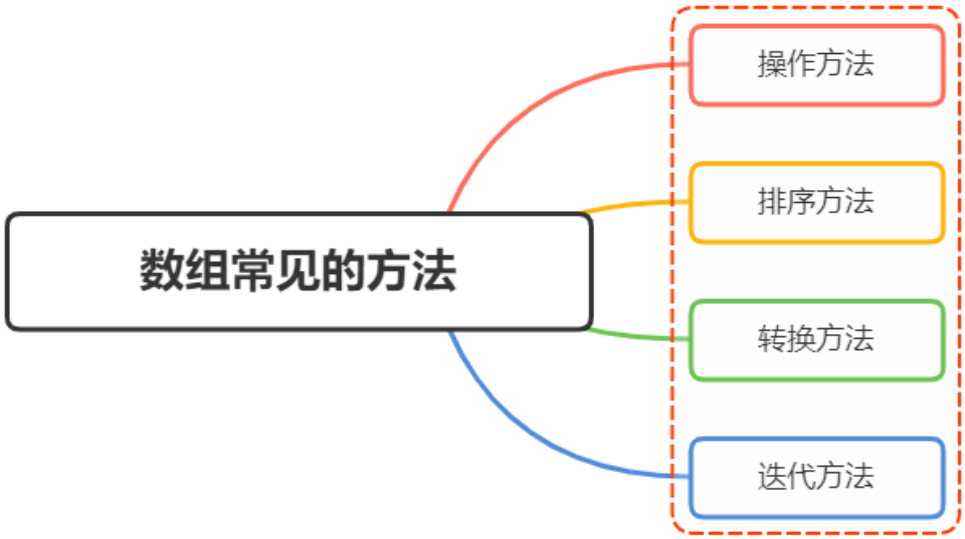
```JavaScript
let text = "cat, bat, sat, fat";
let pos = text.search(/at/);
console.log(pos); // 1
```

19.3.3. replace()

接收两个参数，第一个参数为匹配的内容，第二个参数为替换的元素（可用函数）

```javascript
let text = "cat, bat, sat, fat";
let result = text.replace("at", "ond");
console.log(result); // "cond, bat, sat, fat"
```

20. 数组的常用方法有哪些?



20.1. 操作方法

数组基本操作可以归纳为 增、删、改、查，需要留意的是哪些方法会对原数组产生影响，哪些方法不会

下面对数组常用的操作方法做一个归纳

20.1.1. 增

下面前三种是对原数组产生影响的增添方法，第四种则不会对原数组产生影响

- push()
- unshift()
- splice()
- concat()

20.1.1.1. push()

`push()` 方法接收任意数量的参数，并将它们添加到数组末尾，返回数组的最新长度

JavaScript 复制代码

```
let colors = []; // 创建一个数组
let count = colors.push("red", "green"); // 推入两项
console.log(count) // 2
```

20.1.1.2. unshift()

unshift()在数组开头添加任意多个值，然后返回新的数组长度

JavaScript 复制代码

```
let colors = new Array(); // 创建一个数组
let count = colors.unshift("red", "green"); // 从数组开头推入两项
alert(count); // 2
```

20.1.1.3. splice()

传入三个参数，分别是开始位置、0（要删除的元素数量）、插入的元素，返回空数组

JavaScript 复制代码

```
let colors = ["red", "green", "blue"];
let removed = colors.splice(1, 0, "yellow", "orange")
console.log(colors) // red,yellow,orange,green,blue
console.log(removed) // []
```

20.1.1.4. concat()

首先会创建一个当前数组的副本，然后再把它的参数添加到副本末尾，最后返回这个新构建的数组，不会影响原始数组

JavaScript 复制代码

```
let colors = ["red", "green", "blue"];
let colors2 = colors.concat("yellow", ["black", "brown"]);
console.log(colors); // ["red", "green","blue"]
console.log(colors2); // ["red", "green", "blue", "yellow", "black", "brow
n"]
```

20.1.2. 删

下面三种都会影响原数组，最后一项不影响原数组：

- pop()
- shift()
- splice()
- slice()

20.1.2.1. pop()

`pop()` 方法用于删除数组的最后一项，同时减少数组的 `length` 值，返回被删除的项

JavaScript | 复制代码

```
let colors = ["red", "green"]
let item = colors.pop(); // 取得最后一项
console.log(item) // green
console.log(colors.length) // 1
```

20.1.2.2. shift()

`shift()` 方法用于删除数组的第一项，同时减少数组的 `length` 值，返回被删除的项

JavaScript | 复制代码

```
let colors = ["red", "green"]
let item = colors.shift(); // 取得第一项
console.log(item) // red
console.log(colors.length) // 1
```

20.1.2.3. splice()

传入两个参数，分别是开始位置，删除元素的数量，返回包含删除元素的数组

JavaScript | 复制代码

```
let colors = ["red", "green", "blue"];
let removed = colors.splice(0,1); // 删除第一项
console.log(colors); // green,blue
console.log(removed); // red，只有一个元素的数组
```

20.1.3. slice()

slice() 用于创建一个包含原有数组中一个或多个元素的新数组，不会影响原始数组

```JavaScript
let colors = ["red", "green", "blue", "yellow", "purple"];
let colors2 = colors.slice(1);
let colors3 = colors.slice(1, 4);
console.log(colors)   // red,green,blue,yellow,purple
concole.log(colors2); // green,blue,yellow,purple
concole.log(colors3); // green,blue,yellow
```

20.1.3.1. 改

即修改原来数组的内容，常用 `splice`

20.1.3.2. splice()

传入三个参数，分别是开始位置，要删除元素的数量，要插入的任意多个元素，返回删除元素的数组，对原数组产生影响

```JavaScript
let colors = ["red", "green", "blue"];
let removed = colors.splice(1, 1, "red", "purple"); // 插入两个值，删除一个元素
console.log(colors); // red,red,purple,blue
console.log(removed); // green，只有一个元素的数组
```

20.1.3.3. 查

即查找元素，返回元素坐标或者元素值

- indexOf()
- includes()
- find()

20.1.3.4. indexOf()

返回要查找的元素在数组中的位置，如果没找到则返回 –1

```JavaScript
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
numbers.indexOf(4) // 3
```

20.1.3.5. includes()

返回要查找的元素在数组中的位置，找到返回 `true`，否则 `false`

```JavaScript
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
numbers.includes(4) // true
```

20.1.3.6. find()

返回第一个匹配的元素

```JavaScript
const people = [
    {
        name: "Matt",
        age: 27
    },
    {
        name: "Nicholas",
        age: 29
    }
];
people.find((element, index, array) => element.age < 28) // // {name: "Matt", age: 27}
```

20.2. 排序方法

数组有两个方法可以用来对元素重新排序：

- reverse()
- sort()

20.2.1. reverse()

顾名思义，将数组元素方向反转

```JavaScript
let values = [1, 2, 3, 4, 5];
values.reverse();
alert(values); // 5,4,3,2,1
```

20.2.2. sort()

sort()方法接受一个比较函数，用于判断哪个值应该排在前面

```JavaScript
function compare(value1, value2) {
    if (value1 < value2) {
        return -1;
    } else if (value1 > value2) {
        return 1;
    } else {
        return 0;
    }
}
let values = [0, 1, 5, 10, 15];
values.sort(compare);
alert(values); // 0,1,5,10,15
```

20.3. 转换方法

常见的转换方法有：

20.3.1. join()

join() 方法接收一个参数，即字符串分隔符，返回包含所有项的字符串

```JavaScript
let colors = ["red", "green", "blue"];
alert(colors.join(",")); // red,green,blue
alert(colors.join("||")); // red||green||blue
```

20.4. 迭代方法

常用来迭代数组的方法（都不改变原数组）有如下：

- some()
- every()
- forEach()
- filter()
- map()

20.4.1. some()

对数组每一项都运行传入的测试函数，如果至少有1个元素返回 true ，则这个方法返回 true

JavaScript 复制代码

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
let someResult = numbers.some((item, index, array) => item > 2);
console.log(someResult) // true
```

20.4.2. every()

对数组每一项都运行传入的测试函数，如果所有元素都返回 true ，则这个方法返回 true

JavaScript 复制代码

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
let everyResult = numbers.every((item, index, array) => item > 2);
console.log(everyResult) // false
```

20.4.3. forEach()

对数组每一项都运行传入的函数，没有返回值

JavaScript 复制代码

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
numbers.forEach((item, index, array) => {
    // 执行某些操作
});
```

20.4.4. filter()

对数组每一项都运行传入的函数，函数返回 `true` 的项会组成数组之后返回

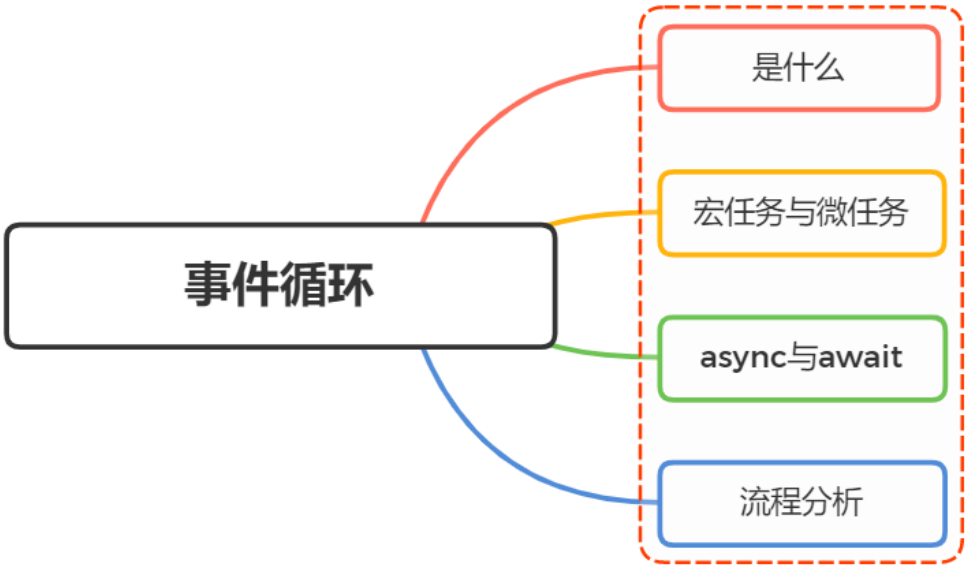
```javascript
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
let filterResult = numbers.filter((item, index, array) => item > 2);
console.log(filterResult); // 3,4,5,4,3
```

20.4.5. map()

对数组每一项都运行传入的函数，返回由每次函数调用的结果构成的数组

```javascript
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
let mapResult = numbers.map((item, index, array) => item * 2);
console.log(mapResult) // 2,4,6,8,10,8,6,4,2
```

21. 说说你对事件循环的理解



21.1. 是什么

首先，`JavaScript`是一门单线程的语言，意味着同一时间内只能做一件事，但是这并不意味着单线程就是阻塞，而实现单线程非阻塞的方法就是事件循环

在`JavaScript`中，所有的任务都可以分为

- 同步任务：立即执行的任务，同步任务一般会直接进入到主线程中执行
- 异步任务：异步执行的任务，比如`ajax`网络请求，`setTimeout`定时函数等

同步任务与异步任务的运行流程图如下：

从上面我们可以看到，同步任务进入主线程，即主执行栈，异步任务进入任务队列，主线程内的任务执行完毕为空，会去任务队列读取对应的任务，推入主线程执行。上述过程的不断重复就事件循环

21.2. 宏任务与微任务

如果将任务划分为同步任务和异步任务并不是那么的准确，举个例子：

JavaScript | 复制代码

```
console.log(1)

setTimeout(()=>{
    console.log(2)
}, 0)

new Promise((resolve, reject)=>{
    console.log('new Promise')
    resolve()
}).then(()=>{
    console.log('then')
})

console.log(3)
```

如果按照上面流程图来分析代码，我们会得到下面的执行步骤：

- `console.log(1)`，同步任务，主线程中执行
- `setTimeout()`，异步任务，放到 `Event Table`，0 毫秒后 `console.log(2)` 回调推入 `Event Queue` 中
- `new Promise`，同步任务，主线程直接执行
- `.then`，异步任务，放到 `Event Table`
- `console.log(3)`，同步任务，主线程执行

所以按照分析，它的结果应该是 `1` => `'new Promise'` => `3` => `2` => `'then'`

但是实际结果是：`1` => `'new Promise'` => `3` => `'then'` => `2`

出现分歧的原因在于异步任务执行顺序，事件队列其实是一个“先进先出”的数据结构，排在前面的事件会优先被主线程读取

例子中 `setTimeout` 回调事件是先进入队列中的，按理说应该先于 `.then` 中的执行，但是结果却偏偏相反

原因在于异步任务还可以细分为微任务与宏任务

21.2.1. 微任务

一个需要异步执行的函数，执行时机是在主函数执行结束之后、当前宏任务结束之前

常见的微任务有：

- Promise.then
- MutaionObserver
- Object.observe（已废弃；Proxy 对象替代）
- process.nextTick（Node.js）

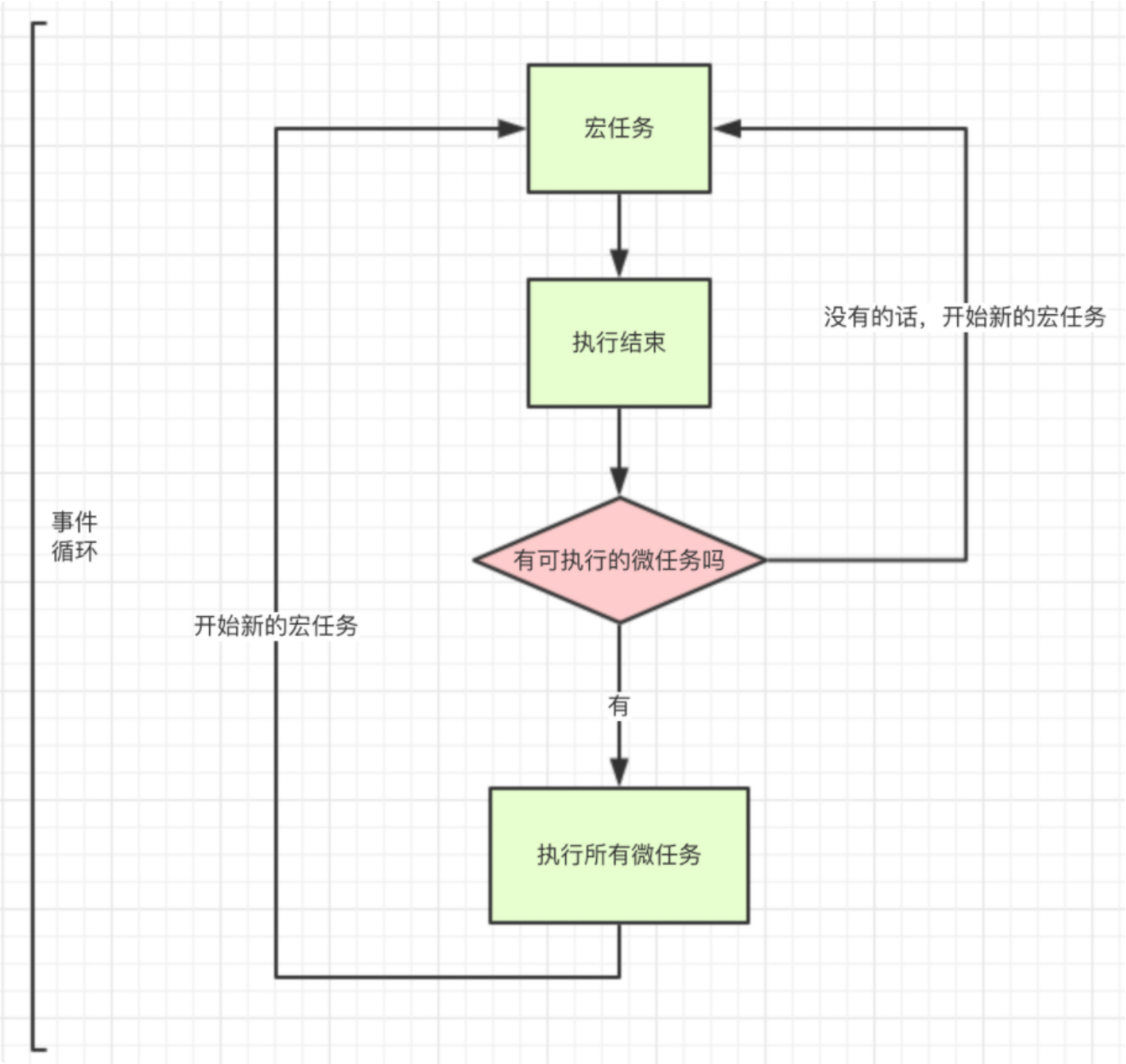
21.2.2. 宏任务

宏任务的时间粒度比较大，执行的时间间隔是不能精确控制的，对一些高实时性的需求就不太符合

常见的宏任务有：

- script (可以理解为外层同步代码)
- setTimeout/setInterval
- UI rendering/UI事件
- postMessage、MessageChannel
- setImmediate、I/O（Node.js）

这时候，事件循环，宏任务，微任务的关系如图所示



按照这个流程，它的执行机制是：

- 执行一个宏任务，如果遇到微任务就将它放到微任务的事件队列中
- 当前宏任务执行完成后，会查看微任务的事件队列，然后将里面的所有微任务依次执行完

回到上面的题目

```JavaScript
console.log(1)
setTimeout(()=>{
    console.log(2)
}, 0)
new Promise((resolve, reject)=>{
    console.log('new Promise')
    resolve()
}).then(()=>{
    console.log('then')
})
console.log(3)
```

流程如下

```JavaScript
// 遇到 console.log(1) ，直接打印 1
// 遇到定时器，属于新的宏任务，留着后面执行
// 遇到 new Promise，这个是直接执行的，打印 'new Promise'
// .then 属于微任务，放入微任务队列，后面再执行
// 遇到 console.log(3) 直接打印 3
// 好了本轮宏任务执行完毕，现在去微任务列表查看是否有微任务，发现 .then 的回调，执行它，打印 'then'
// 当一次宏任务执行完，再去执行新的宏任务，这里就剩一个定时器的宏任务了，执行它，打印 2
```

21.3. async与await

`async` 是异步的意思， `await` 则可以理解为 `async wait` 。所以可以理解 `async` 就是用来声明一个异步方法，而 `await` 是用来等待异步方法执行

21.3.1. async

`async` 函数返回一个 `promise` 对象，下面两种方法是等效的