为了更好地解释实现效果，采用如下方式调用_.debounce方法：

```
let debounceInputEl = document.getElementById('debounce')
    debounceInputEl.addEventListener('keyup', (e) => {
     // 调用防抖方法
     // 设置immediate为true
     _.debounce(ajax, 1000, true)(e.target.value, '额外的参数')
     // 设置immediate默认为false
     _.debounce(ajax, 1000)(e.target.value, '额外的参数')
})
function ajax(...params) {
    console.log('实际执行传入的函数func----', `参数：${params[0]}、${params[1]}`, format(+new Date()))
}
```

设置immediate为true，执行结果如下：

```
timeout undefined
input触发:  ▶(2) ["q", "额外的参数"] 23:16:03
实际执行传入的函数func---- 参数：q、额外的参数 23:16:03
timeout 14
input触发:  ▶(2) ["qwe", "额外的参数"] 23:16:03
timeout 15
input触发:  ▶(2) ["qwe", "额外的参数"] 23:16:03
timeout 16
input触发:  ▶(2) ["qweqw", "额外的参数"] 23:16:04
timeout 17
input触发:  ▶(2) ["qweqw", "额外的参数"] 23:16:04
timeout 18
input触发:  ▶(2) ["qweqwe", "额外的参数"] 23:16:05
timeout null
input触发:  ▶(2) ["qweqwer", "额外的参数"] 23:16:06
实际执行传入的函数func---- 参数：qweqwer、额外的参数 23:16:06
```

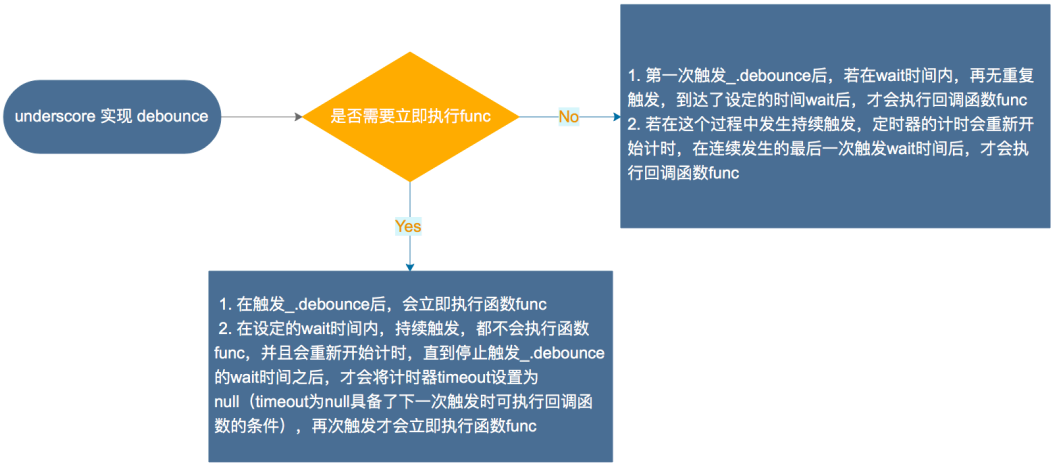
在例子中，所设定的wait时间间隔是1s。

在3s时，第一次触发，看到传入_.debounce的ajax函数被立即调用；在3s-4s的时间内，又重新触发，计时器会重新开始计算，原本在4s可以清空计时器，现在需要在第5s才会清空计时器，以此类推，在设定的wait时间内，只要持续触发，函数ajax都再不会执行。5s时，结束了上一次的连续触发，在wait时间后，即6s时，timeout已经被清空为null，此时，触发_.debounce又会立即调用ajax。

设置immediate为false，执行结果如下：

```
timeout undefined
input触发:  ▶ (2) ["q", "额外的参数"] 23:42:31
timeout 14
input触发:  ▶ (2) ["qw", "额外的参数"] 23:42:31
timeout 15
input触发:  ▶ (2) ["qwe", "额外的参数"] 23:42:32
实际执行传入的函数func---- 参数：qwe、额外的参数 23:42:33
timeout null
input触发:  ▶ (2) ["qweqw", "额外的参数"] 23:42:34
timeout 17
input触发:  ▶ (2) ["qweqwe", "额外的参数"] 23:42:34
timeout 18
input触发:  ▶ (2) ["qweqwe", "额外的参数"] 23:42:34
实际执行传入的函数func---- 参数：qweqwe、额外的参数 23:42:35
```

由运行结果可以看到，连续触发，计时器会重新开始计算，总是在连续触发的最后一次触发的1s之后，才会延迟调用传入_.debounce的ajax函数。



throttle

如果觉得源码在没有上下文的情况下晦涩难懂，可以结合以下例子理解。

采用如下方式调用_.throttle方法：

```
let throttleInput = document.getElementById('throttle')
let options = {
  // leading: false
```

```
  // trailing: false
}
throttleInput.addEventListener('keyup', (e) => {
  // 调用节流方法
  _.throttle(ajax, 4000, options)(e.target.value)
})
function ajax(...params) {
  console.log('实际执行传入的函数func----', `参数：${params[0]}`,
format(+new Date()))
}
```

1. 未设置leading和trailing

效果：第一次触发，立即执行。此后，若触发时间与上一次触发时间的差值大于wait，则立即执行，否则，延后执行，执行时间是：上一次执行func的时间+wait。

```
input触发:  ▸Arguments ["q", callee: (...), Symbol(Symbol.iterator): f] 00:23:48
实际执行传入的函数func---- 参数: q 00:23:48
input触发:  ▸Arguments ["qw", callee: (...), Symbol(Symbol.iterator): f] 00:23:53
实际执行传入的函数func---- 参数: qw 00:23:53
input触发:  ▸Arguments ["qwe", callee: (...), Symbol(Symbol.iterator): f] 00:23:55
实际执行传入的函数func---- 参数: qwe 00:23:57
```

2. trailing:false

效果：第一次触发，立即执行。此后，若触发时间与上一次触发时间的差值大于wait，则立即执行，否则，不执行。

```
input触发:  ▸Arguments ["q", callee: (...), Symbol(Symbol.iterator): f] 00:08:06
实际执行传入的函数func---- 参数: q 00:08:06
input触发:  ▸Arguments ["qw", callee: (...), Symbol(Symbol.iterator): f] 00:08:07
input触发:  ▸Arguments ["qwe", callee: (...), Symbol(Symbol.iterator): f] 00:08:09
input触发:  ▸Arguments ["qwer", callee: (...), Symbol(Symbol.iterator): f] 00:08:14
实际执行传入的函数func---- 参数: qwer 00:08:14
```

3. 两个为false(关闭立即和延时执行)

效果：第一次触发，不立即执行，并将该次触发时间赋值给previous，标记为已执行。此后每次的触发时间与previous的差值，若大于wait，则执行func，否则，不执行。当执行func时，会更新previous，再次通过以上规则进行比较，推测何时可执行func，以此类推。

```
input触发:  undefined 02:36:03
input触发:  ▸Arguments ["q", callee: (...), Symbol(Symbol.iterator): f] 02:36:03
input触发:  ▸Arguments ["qw", callee: (...), Symbol(Symbol.iterator): f] 02:36:04
input触发:  ▸Arguments ["qwe", callee: (...), Symbol(Symbol.iterator): f] 02:36:06
input触发:  ▸Arguments ["qweq", callee: (...), Symbol(Symbol.iterator): f] 02:36:08
实际执行传入的函数func---- 参数: qweqw 02:36:08
```

4. leading为false

效果：第一次触发，不立即执行。此后触发，延迟执行func，执行时间是：上一次执行结束后的第一轮触发时间+wait。

```
input触发:  ▸Arguments ["q", callee: (...), Symbol(Symbol.iterator): f] 00:18:05
实际执行传入的函数func---- 参数: q 00:18:09
input触发:  ▸Arguments ["qe", callee: (...), Symbol(Symbol.iterator): f] 00:18:19
实际执行传入的函数func---- 参数: qe 00:18:23
input触发:  ▸Arguments ["qer", callee: (...), Symbol(Symbol.iterator): f] 00:18:31
实际执行传入的函数func---- 参数: qer 00:18:35
```

综上，默认选项是时间戳和延时两种方式的结合实现，时间戳的实现本质是为了在

理解关键点：每次执行成功后，都会给previous赋值。