# 正则表达式回溯

# 案例

前几天有小伙伴来求救说页面上有一个 input 框，随着用户不断输入内容，页面响应会越来越慢直到完全失去响应。

简单沟通过后得知具体场景是这样的：

* input 框中允许用户输入一连串逗号分隔的商品id
* 在用户输入的过程中实时检测用户输入的内容是否符合规则，若不符合则给出提示信息：

123456789,123456789,123456789,123456789,123456789,123456789,123456789,123456789,123456789,123456789

经过反复验证得到如下规律：

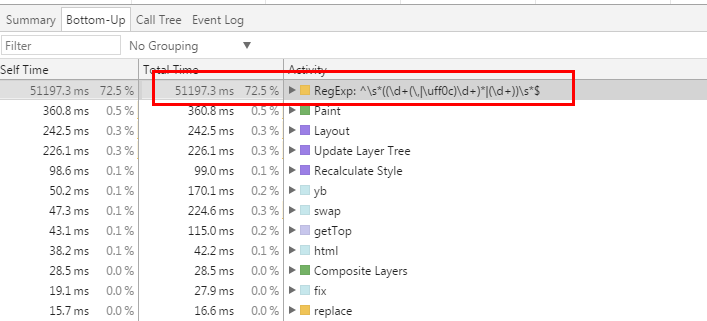
* 用户在输入商品 id 的过程中（连续输入多个数字）不会卡顿
* 当用户输入逗号时，出现卡顿。随着输入商品 id 的数量增加，卡顿越来越明显，直至浏览器失去响应。

于是打开 Chrome 开发者工具，选择 Performance (原 Timeline) 标签页。将整个过程记录下来，得到如下时间线：



其中黄色宽条表示 JavaScript 主线程的执行情况。连续的黄条越长，表示单次 JavaScript 运行的时间越长。也就意味着 UI 失去响应的时间越长。这一点从截图中的蓝色框中也可以得到印证。蓝色框中的红色长条表示浏览器一帧（一次渲染）所需要的时间。

那么到底是 JavaScript 中的哪些代码占中了这么长 CPU 时间呢？我们在底部的选项卡中选中 Bottom-Up ，按 Total Time 降序排列。得到如下结果：



可以看出，72.% 的 CPU 时间用在了一条正则表达式上。你肯定想到了，这就是小伙伴用来检查用户输入是否合法的正则表达式。

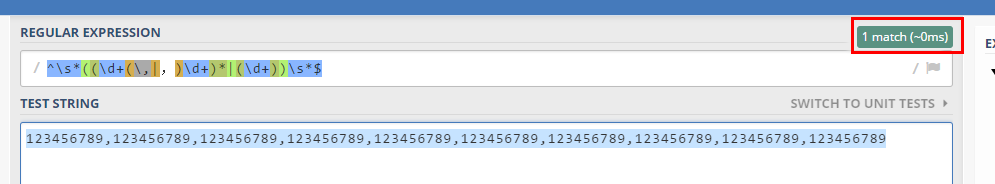
完整的正则表达式是这样的：

/^\s\*((\d+(\,|，)\d+)\*|(\d+))\s\*$/

接着去 [regex101](https://link.juejin.im/?target=https%3A%2F%2Fregex101.com%2F) 上测试一下，测试数据如下，由 10 个商品 ID 组成的字符串：

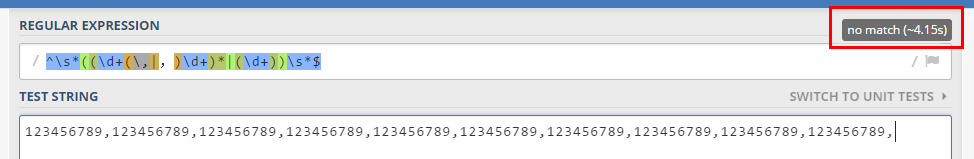
123456789,123456789,123456789,123456789,123456789,123456789,123456789,123456789,123456789,123456789

执行结果如下：



可以看到执行速度非常快，只用了不到 1ms。

接下来在测试数据结尾加一个逗号，以模拟不符合规则的情况：



正则表达式执行的时间暴增到 4.15s。

经过多次测试发现：每次正常匹配执行的时间都很短。每次不匹配时，执行的时间都很长，且随着字符串长度的增加，时间成倍的增长。

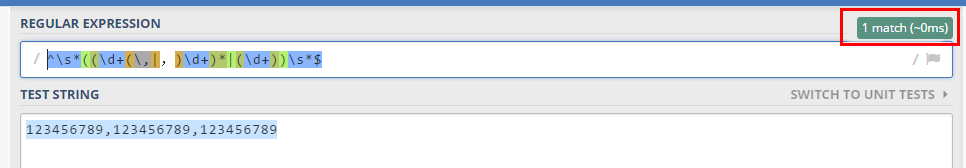
接下来让我们认真的观察一下这个正则表达式：

/^\s\*((\d+(\,|，)\d+)\*|(\d+))\s\*$/

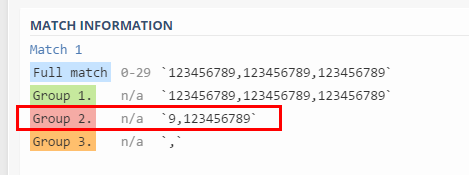
去掉匹配首尾的空白字符，其核心结构只有两部分 ((\d+(\,|，)\d+)\* 与 (\d+)。前者用于匹配多个商品 ID 的情况，后者匹配只有一个商品 ID 的情况。

前者的基本模式是这样的 商品ID,商品ID，然后把该模式重复多次。仔细观察后很快我就发现了第一个问题，假设用户输入的内容是 商品ID,商品ID,商品ID 。你会发现它符合输入规则，但是不与该正则表达式匹配。因为第一次匹配后剩余的字符串部分 ,商品ID 无法与基本模式形成匹配。

这的是这样吗？



测试发现，依然可以匹配。但匹配的内容和我们预期的并不一致。



最后一次匹配的内容是，9,123456789。不难想象第一次的匹配结果就是 123456789,12345678。

这里可以看出小伙伴编写的正则有两个问题：

* 逻辑错误。通过测试结果可以看出无法匹配出正确的商品 ID。如果商品 ID 运行只有 1 位数字，则匹配失败。
* 性能差。

# 解决办法

在了解需求后，我给小伙伴提供了一种正则写法：

^\s\*(\d+(,|，))\*\d+\s\*$

经过测试，这种写法在保证逻辑无误的前提下还保证了执行效率(在有数百个商品 ID 的情况下依然可以在几毫秒内执行完毕)。

讲到这里，你可能会有两个问题：

1. 为何第一种写法的正则表达式匹配结果和我们预想的不一致。
2. 为何两种写法的性能差别如此之大。

要回答这个问题，还要从正则表达式中 \* 符号的执行逻辑说起。

# 回溯

大家都知道 \* 表示匹配前面的子表达式 0 次或多次（且尽可能多的匹配）。但这个逻辑具体是如何执行的呢？让我们通过几个小例子来看一下。

### Round 1

假设有正则表达式 /^(a\*)b$/ 和字符串 aaaaab。如果用该正则匹配这个字符串会得到什么呢？

答案很简单。两者匹配，且捕获组捕获到字符串 aaaaa。

### Round 2

这次让我们把正则改写成 /^(a\*)ab$/。再次和字符串 aaaaab 匹配。结果如何呢？

两者依然匹配，但捕获组捕获到字符串 aaaa。因为捕获组后续的表达式占用了 1 个 a 字符。但是你有没有考虑过这个看似简单结果是经过何种过程得到的呢？

让我们一步一步来看：

1. 匹配开始 (a\*) 捕获尽可能多的字符 a。
2. (a\*) 一直捕获，直到遇到字符 b。这时 (a\*) 已经捕获了 aaaaa。
3. 正则表达式继续执行 (a\*) 之后的 ab 匹配。但此时由于字符串仅剩一个 b 字符。导致无法完成匹配。
4. (a\*) 从已捕获的字符串中“吐”出一个字符 a。这时捕获结果为 aaaa，剩余字符串为 ab。
5. 重新执行正则中 ab的匹配。发现正好与剩余字符串匹配。整个匹配过程结束。返回捕获结果 aaaa。

从第3，4步可以看到，暂时的无法匹配并不会立即导致整体匹配失败。而是会从捕获组中“吐出”字符以尝试。这个“吐出”的过程就叫**回溯**。

回溯并不仅执行一次，而是会一直回溯到另一个极端。对于 \* 符号而言，就是匹配 0 次的情况。

### Round 3

这次我们把正则改为 /^(a\*)aaaab$/。字符串依然为 aaaaab。根据前边的介绍很容易直到。此次要回溯 4 次才可以完成匹配。具体执行过程不再赘述。

# 悲观回溯

了解了回溯的工作原理，再来看**悲观回溯**就很容易理解了。

### Round 4

这次我们的正则改为 /^(a\*)b$/。但是把要匹配的字符串改为 aaaaa。去掉了结尾的字符 b。

让我们看看此时的执行流程：

1. (a\*) 首先匹配了所有 aaaaa。
2. 尝试匹配 b。但是匹配失败。
3. 回溯 1 个字符。此时剩余字符串为 a。依然无法匹配字符 b。
4. 回溯一直进行。直到匹配 0 次的情况。此时剩余字符串为 aaaaa。依然无法匹配 b。
5. 所有的可能性均已尝试过，依然无法匹配。最终导致整体匹配失败。

可以看到，虽然我们可以一眼看出二者无法匹配。但正则表达式在执行时还要“傻傻的”逐一回溯所有可能性，才能确定最终结果。这个“傻傻的”回溯过程就叫**悲观回溯**。

虽然这个过程看起来有点傻，但是不是感觉也没什么大问题？为何会有性能问题呢？让我们回到最初的那个正则表达式。

### Round 5

这次正则表达式回到 ^\s\*((\d+(\,|，)\d+)\*|(\d+))\s\*$。字符串为123456789,123456789,123456789, 执行的结果依然为不匹配。这点毫无疑问。但问题是，执行的过程中，进行了多少次回溯呢？

让我们统计一下：

1. 首轮执行过后的捕获结果是 123456789,12345678，9,123456789。但这时剩余字符串仅剩 ,一个字符。于是开始悲观回溯。
2. 首先看第一个匹配不变的情况下，第二个匹配组回溯的情况。

a. 回退 1 个字符。剩余字符串为 9,。不匹配。共回溯 1 次。  
b. 回退 2 个字符。剩余字符串为 89,。不匹配。但是 89 又进行一次回溯。共回溯 2 次。  
c. 以此类推。最多回退 8 个字符。此时剩余字符串为 23456789,。共可以回溯 8 次。  
d. 累计回溯 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36 次。

1. 接着，第一个捕获组回溯 1 个字符。捕获结果变为 123456789,1234567，89,123456789。此时又将循环一遍 2 中的所有逻辑。累计回溯 36 + 1次。
2. 以此类推，全部回溯完成，需要回溯 324 次。

假设我们增加一个商品 ID，字符串变为 123456789,123456789,123456789,123456789,。此时的回溯次数增加到 2628 次。

以此类推可得。

| **商品 ID 个数** | **回溯次数** |
| --- | --- |
| 3 | 324 |
| 4 | 2628 |
| 5 | 21060 |
| 6 | 168516 |
| 7 | 1348164 |
| 8 | 10785348 |
| 9 | 86282820 |
| 10 | 690262596 |

可见问题在于，随着商品 ID 个数的增长，回溯次数会成指数级增长。最终导致 JavaScript 主进程忙于进行计算，使页面失去响应。

但是我当时给出的解决方案：

^\s\*(\d+(,|，))\*\d+\s\*$

也使用了 \* 符号，按说也会进行悲观回溯。为何没有性能问题呢？

答案在于，**对于同一字符串是否有多种可行的匹配模式**。也就是说对于某个固定的字符串，你的正则表达式是否有“唯一解”。

举例对于我给出的正则，对于字符串 123456789,123456789,123456789 只可能有 1 种匹配结果。那就是 123456789,，123456789, 和 123456789。因此，在回溯时只需进行一次线性的回溯即可（24 次）。而不会像前面分析的第一种正则一样，有多种“可能”的匹配方式。

# 解决回溯的思路

在了解了悲观回溯为何会导致性能问题后，就可以考虑如何解决这个问题。要解决这个问题，大概有以下几个思路：

### 思路一： 禁止回溯

这个思路很直接，既然回溯可能有性能问题，那我们是否可以禁止正则表达式进行回溯呢。

答案是：可以。

有两种语法可以防止回溯：

* 有限量词（Possessive Quantifiers）
* 原子分组（Atomic Grouping）

关于这两种语法，感兴趣的同学可以自行 Google。在此不详细解释。因为**这两种语法在 JavaScript 中均不被支持。**

### 思路二：避免导致性能问题的回溯

这个思路也比较容易想到。其实经过思考不难想到。两种模式的正则表达式很可能会导致有性能问题的回溯。

* 前后重复的模式。 例如 /x\*x\*/。虽然这个例子看起来很“弱智”，但是当规则变复杂时，每一个 x 又可能是由多个子表达式组成的。当这些子表达式存在逻辑上的交集时，就可能会出现性能问题。
* 嵌套的量词。例如 /(x\*)\*/。包括文中提到的第一个正则也属于这种模式。

当我们在编写正则表达式时写出了这种模式的时候，大家就要谨慎起来。考虑一下是否有潜在的性能问题，是否有更好的写法了。

### 思路三：不使用正则表达式

其实像文中举的这个例子，甚至都没必要使用正则表达式。直接写一个 JavaScript 函数，按逗号切分字符串，逐个字符判断即可。而且可以保证代码的性能是线性的。

### 思路四：避免性能问题影响页面响应

在必须使用正则表达式，且的确有潜在的性能问题的情况下。要避免正则表达式的性能影响到页面响应。一种可能的方式是像 [regex101](https://link.juejin.im/?target=https%3A%2F%2Fregex101.com%2F) 一样把正则表达式的匹配操作放到 [Service Worker](https://link.juejin.im/?target=https%3A%2F%2Fdeveloper.mozilla.org%2Fen-US%2Fdocs%2FWeb%2FAPI%2FService_Worker_API)中进行。

# 延伸阅读

如果你还想了解更多相关信息，可以阅读以下链接。

* [Regex Tutorial - Repetition with Star and Plus](https://link.juejin.im/?target=http%3A%2F%2Fwww.regular-expressions.info%2Frepeat.html)
* [Regex Tutorial - Possessive Quantifiers](https://link.juejin.im/?target=http%3A%2F%2Fwww.regular-expressions.info%2Fpossessive.html)
* [Regex Tutorial - Atomic Grouping](https://link.juejin.im/?target=http%3A%2F%2Fwww.regular-expressions.info%2Fatomic.html)
* [Runaway Regular Expressions: Catastrophic Backtracking](https://link.juejin.im/?target=http%3A%2F%2Fwww.regular-expressions.info%2Fcatastrophic.html)
* [Catastrophic Backtracking ‒ When Regular Expressions Explode](https://link.juejin.im/?target=https%3A%2F%2Fvimeo.com%2F112065252)