## 1前言

作为一名QA，如何快速地进行项目组内人员对项目意见的收集并及时进行跟进与反馈是非常重要的一件事情。在作者项目组内，其他成员的反馈意见除了私聊，更多的还是通过popo群的形式，直接把问题的截图以及自己对问题的看法一起发送到群里。然后QA对这些问题进行一个收集与反馈。

但是这种反馈和收集是基于popo消息甚至是口头消息，因此信息的时效性和准确性可能会受到影响。 所以作者通过feedback结合chrome插件，以及易协作的接口对这种情况做了一些改进。

## 2 FeedBack的用法

FeedBack是一个可以根据关键字筛选popo群内信息，并自动在网页上生成一个反馈详情页面的平台。通过把feedback的机器人拉进对应popo群，然后配置好相应的关键字，即可进行群内成员的反馈信息收集了。

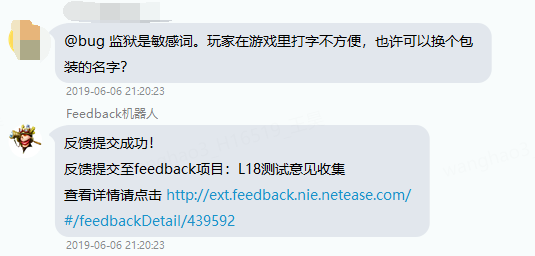


图2-1 反馈群信息收集



图2-2 Feedback平台页面

接下来就是要把FeedBack里面的反馈信息开成bug单。本来是要获取到关键词时调一下开bug单的接口，但是由于记录的信息过于杂乱，并不能简单粗暴的把所有的单子都一股脑的变成bug单。因此作者打算通过chrome插件的形式进行页面信息的获取，对获取的页面信息进行筛选后再进行开单。

## 3 chrome-extension核心介绍

Manifest

Chrome插件最重要也是必不可少的文件，用来配置所有和插件相关的配置，必须放在根目录。其中，manifest\_version、name、version3个是必不可少的，description和icons是推荐的。

Background Script

是一个常驻的页面，它的生命周期是插件中所有类型页面中最长的，它随着浏览器的打开而打开，随着浏览器的关闭而关闭，所以通常把需要一直运行的、启动就运行的、全局的代码放在background里面。

background的权限非常高，几乎可以调用所有的Chrome扩展API（除了devtools），而且它可以无限制跨域，也就是可以跨域访问任何网站而无需要求对方设置CORS。永远看不到它的界面，只能调试它的代码。

Content Script

其实就是Chrome插件中向页面注入脚本的一种形式（虽然名为script，其实还可以包括css的），借助content-scripts我们可以实现通过配置的方式轻松向指定页面注入JS和CSS（如果需要动态注入，可以参考下文），最常见的比如：广告屏蔽、页面CSS定制，等等。在后面作者将用它来进行feedback页面内容的获取与操控。

Popup

Popup页面是点击扩展按钮后弹出的一个web窗口。可以通过default\_popup字段来指定popup页面，也可以调用setPopup()

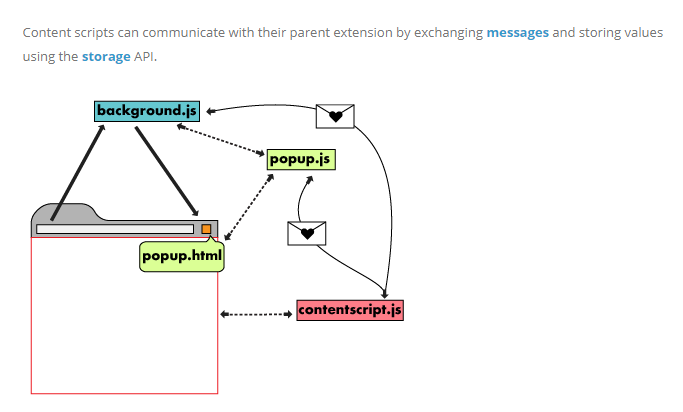


图3-1 popup、backgroun与content之间的通信

## 4 开发过程

大致需要获取的页面信息有如下几种：项目名称、反馈类型、反馈提交人、反馈来源、反馈详情、提交时间以及截图。成功开单后点一下【接受】按钮并且在评论列表附上开单成功以及相应的单号即可。



图4-1 Feedback反馈详细页面

准备好background.js、content\_script.js等一些关键性的文件。

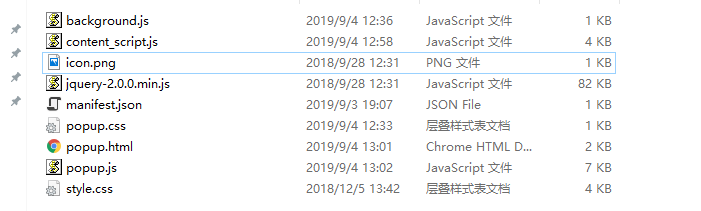


图4-2 chrome-extension文件夹

4.1 Manifest文件配置

Background用来配置相应的后台脚本，需要用到的是jquery和background.js

page\_action用来配置浏览器上面对应的按钮以及点击按钮后弹出的页面

permissions用来申请一些必要的权限

content\_scripts 用来配置注入脚本

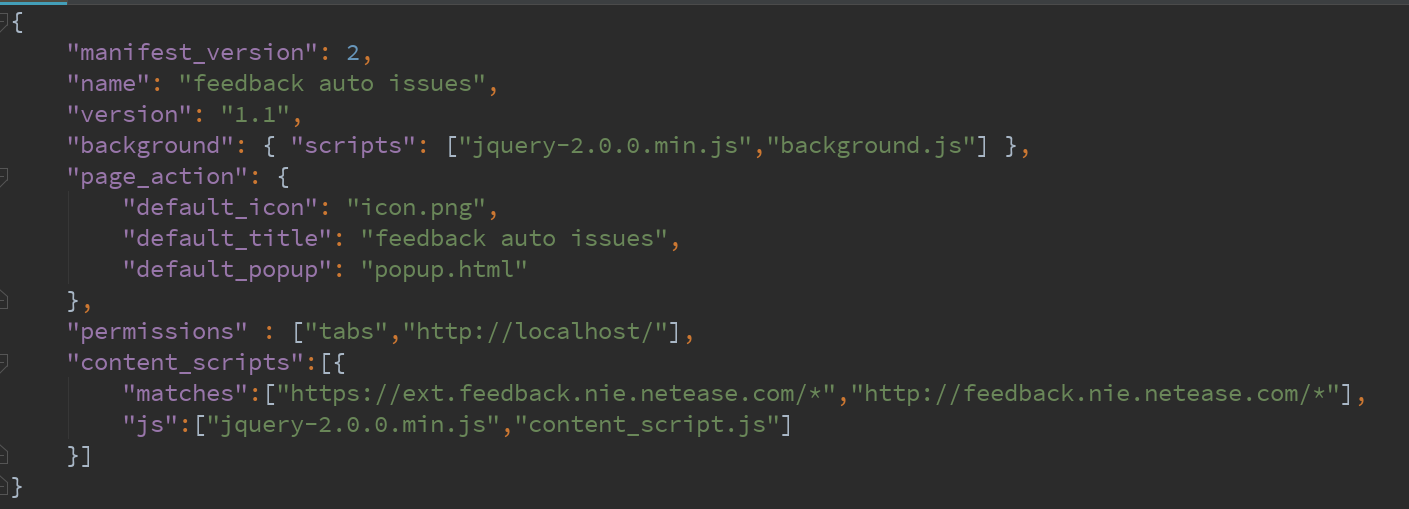


图4-3 Manifest文件配置

4.2 Background.js文件

background的权限非常高，几乎可以调用所有的Chrome扩展API（除了devtools），而且它可以无限制跨域，也就是可以跨域访问任何网站而无需要求对方设置。由于其会一致常驻后台，作者把它用来检测当前的url是否满足chrome插件允许的条件。

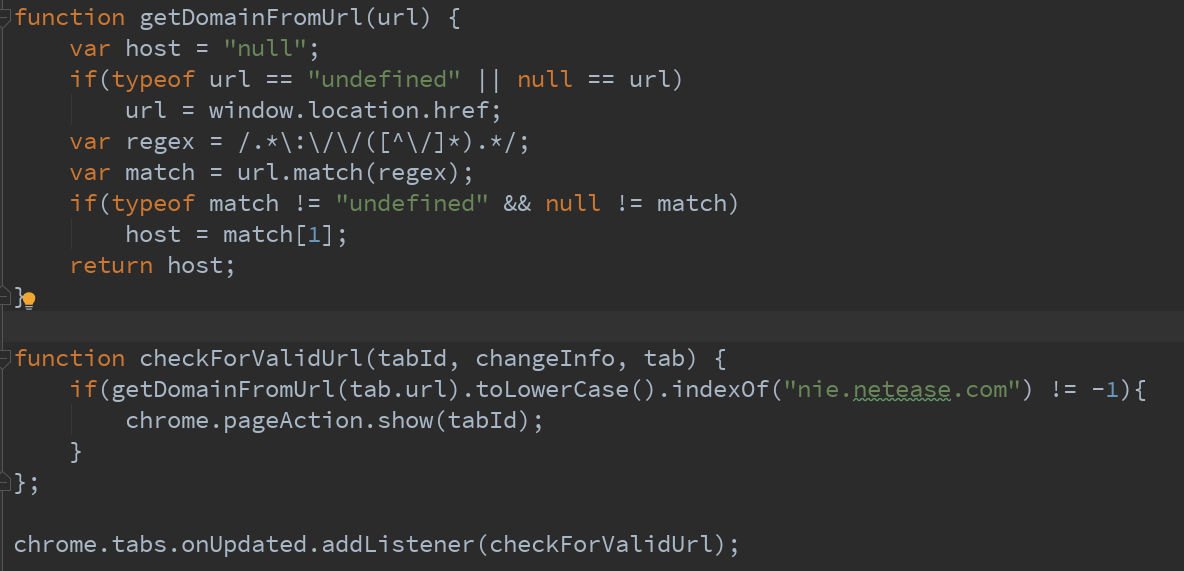


图4-4 Background文件配置

4.3 Popup文件和content\_script之间的通信

当按下按钮时，popup会通知content\_script进行页面信息的获取，因此需要在这里建立popup和content\_script之间的长通信。建立通道后，双方可以互相发送和监听消息。

如下图，pupup在等待DOMContentLoaded完成后去请求content\_script获取页面的信息。在这里必须要注意的就是进行页面信息获取时是通过获取页面的dom树来进行的，如果在dom还没渲染完成时就去请求，那么得到的数值是不正确的。比如我在content\_script运行时就直接var postInfo = document.getElementsByClassName('is-subject');来进行获取dom。

当我打印出这个postInfo后，发现postinfo是一个HTMLCollection对象。但是当我试图打印其长度，或者遍历这个对象的子集时，却发现这个对象又为空了。在网上看了很多评论，有许多人碰到了这个问题。原因是因为这里获取到的是个伪数组,渲染之前打印的这个数组，看到的是渲染后的数据，通俗的讲，就是这个dom结构还没加载完，js就执行了，解决办法就等dom结构加载完成后再去获取这个数组。

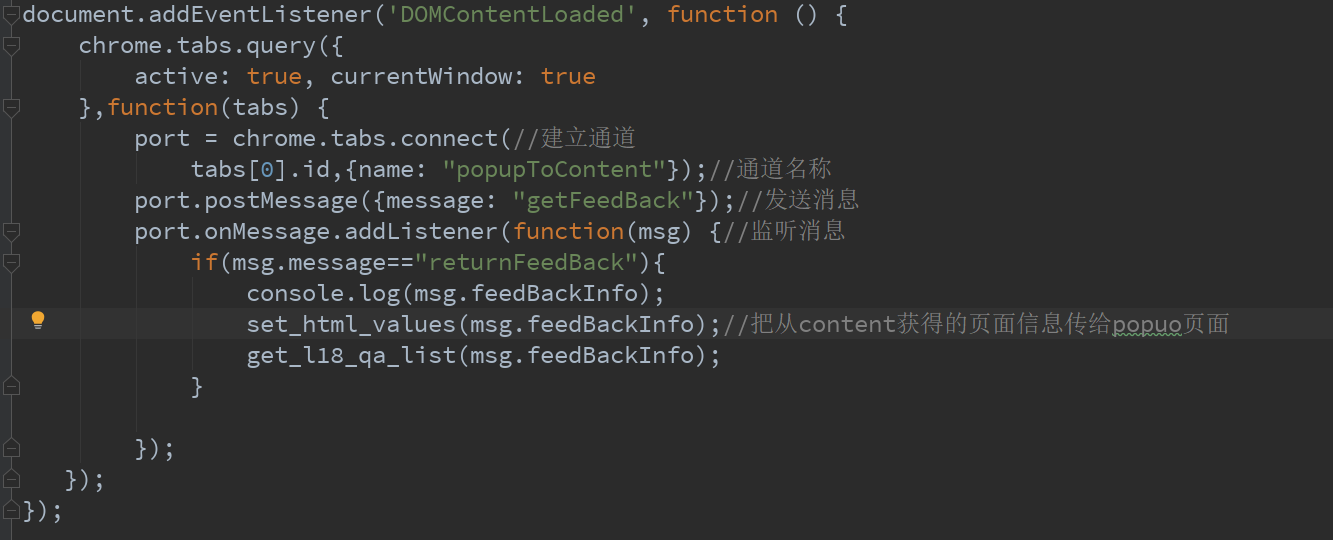


图4-5 popup.js建立通道

Content\_script文件

我们可以通过Content\_script轻松向指定页面注入JS和CSS。因此获取页面信息、操作页面的主力就是它了。因此content\_script需要监听从popup发来的端口建立信息。端口建立后开始监听popup发来的请求。

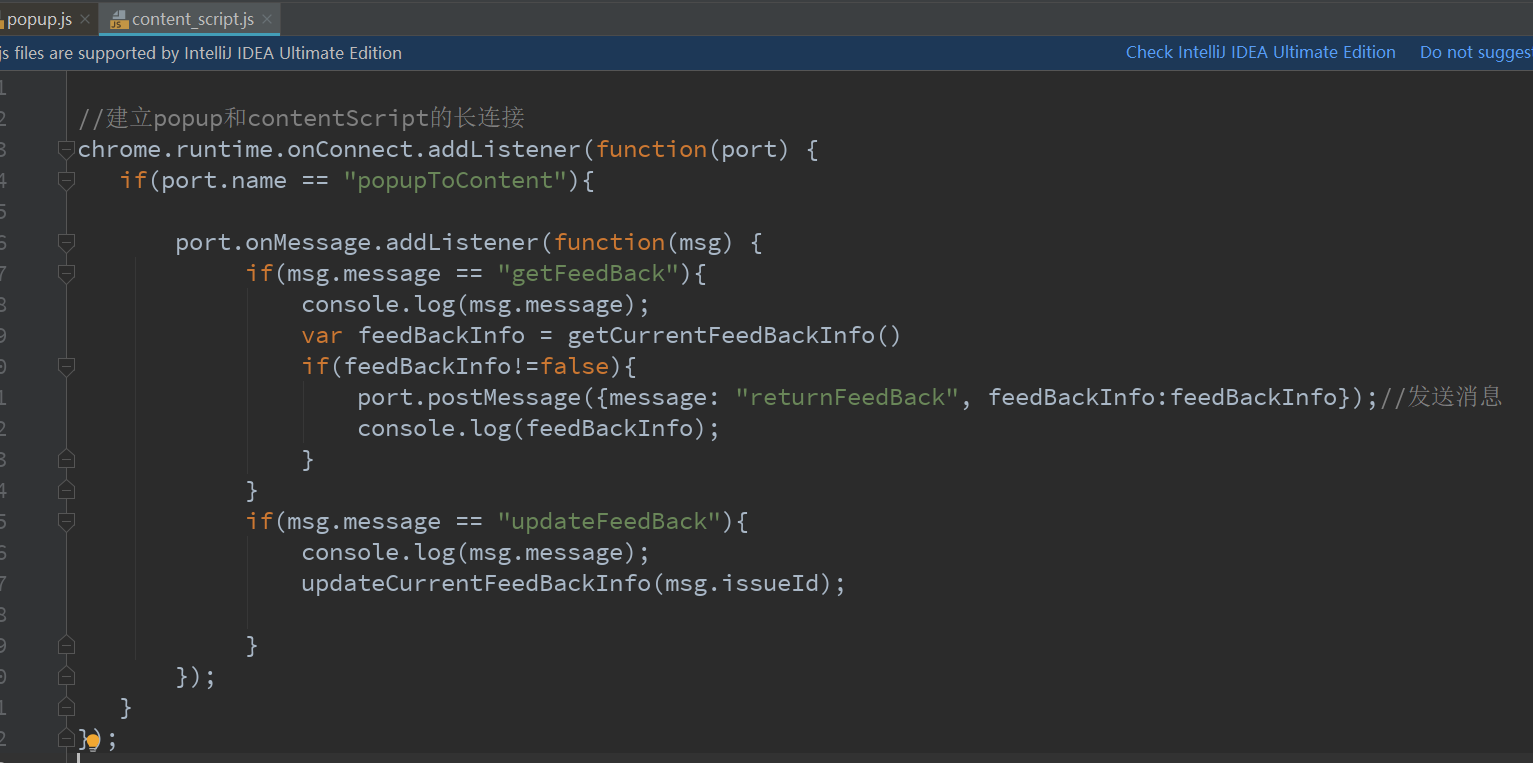


图4-6 content.js建立通道并监听

当popup发出getFeedBack的请求之后，content\_script开始进行dom的获取。



图4-7 content.js获取后台dom

获取完成后，content\_script发送给popup returnFeedBack 的消息。Popup接受后把这些信息通过JS赋值给自定义的popup.html，进行一个页面的展示。

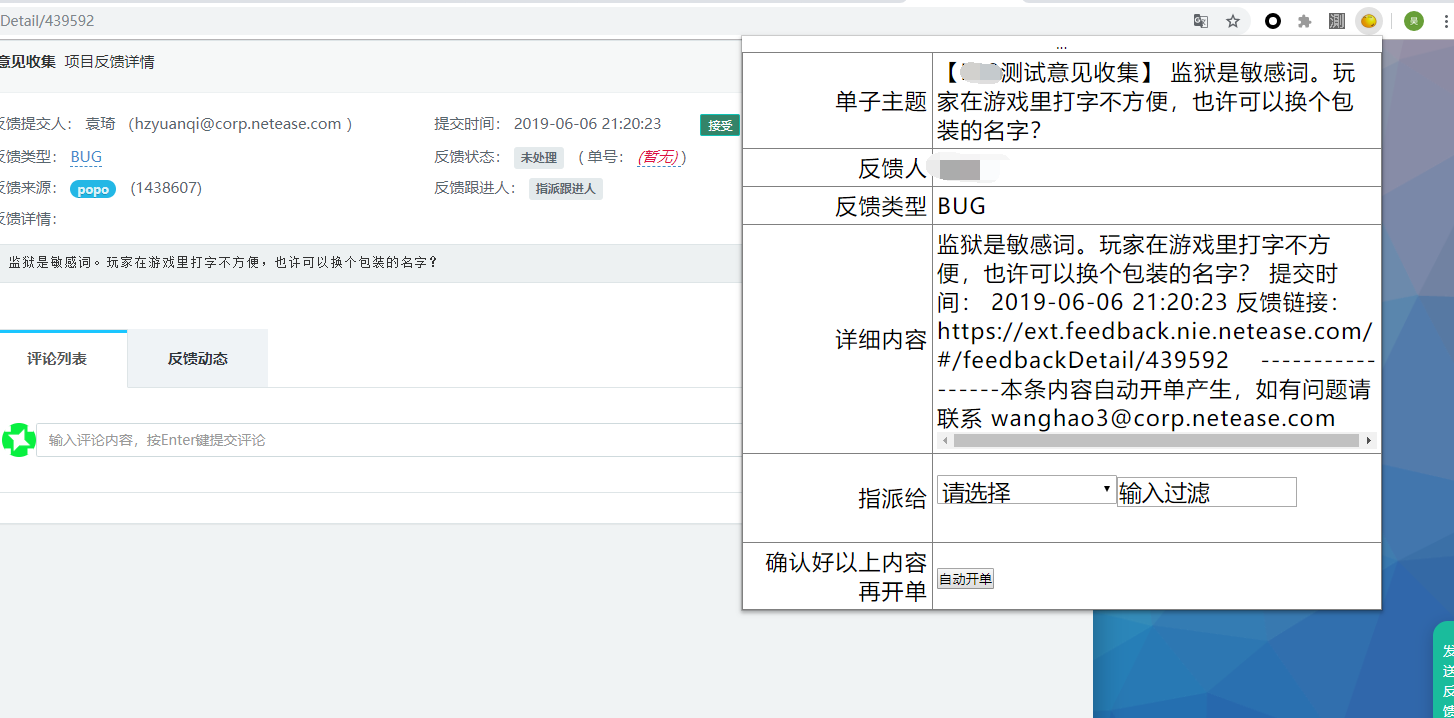


图4-8 feedback的chrome-extension页面

接下来就是等待button的触发，按下这个button之后，popup.js会调用之前写好的开单接口，开单成功后返回对应的单号。接下来要做的就是拿着这个单号去通知content\_script进行feedback页面信息的更新。

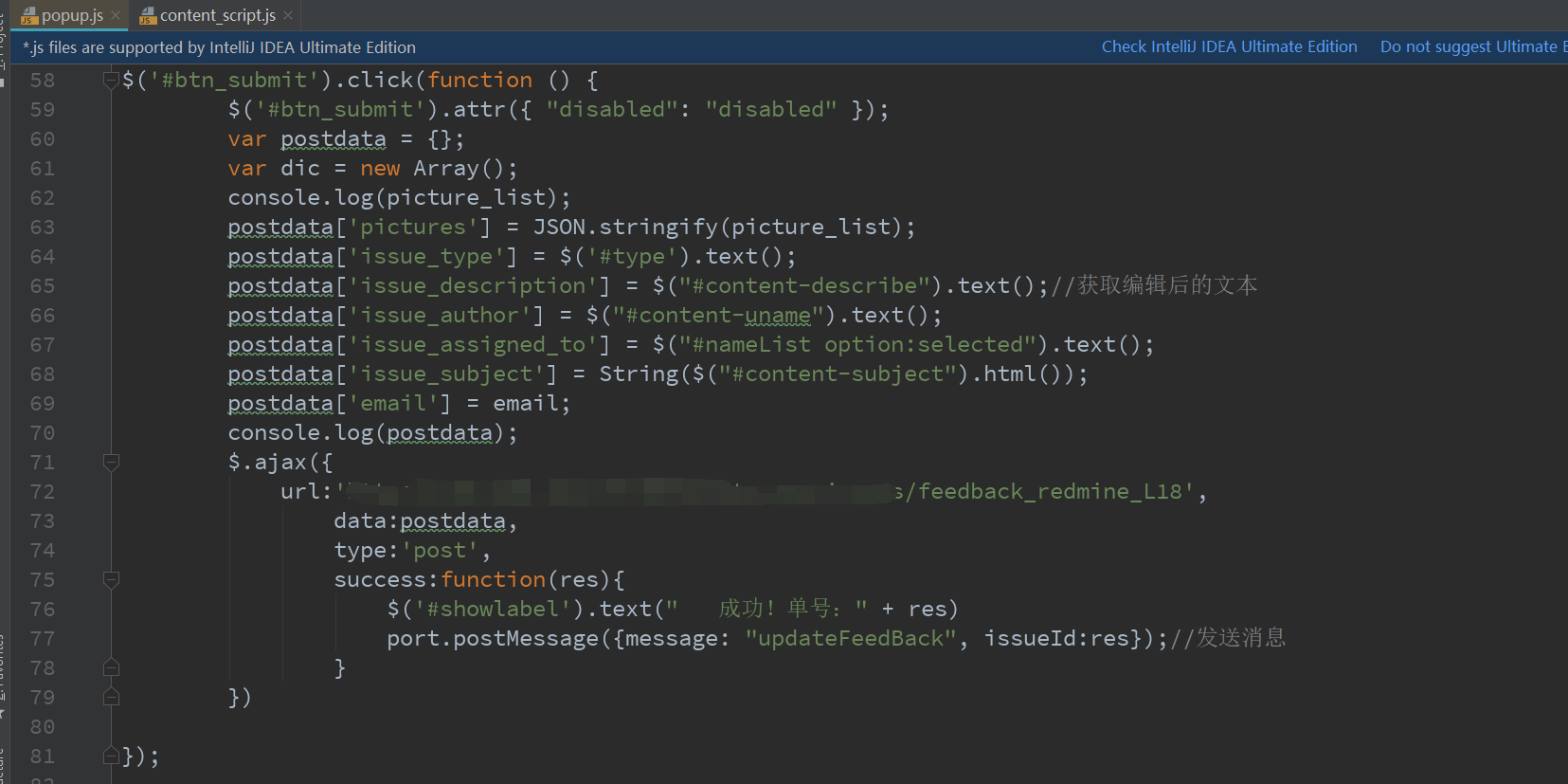


图4-9 popup.js开单并通知content更新

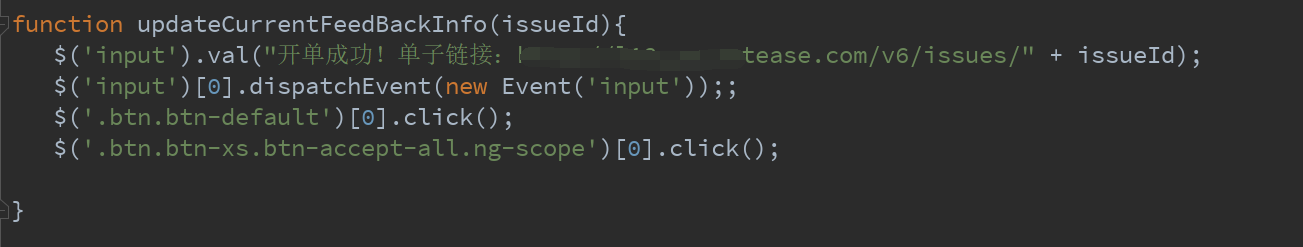


图4-10 content.js注入脚本更新feedback页面

这里作者遇到一个坑，就是feedback是使用了regularJs框架来写的。因此input控件的value是通过model进行和后台的双向绑定的。如果只是改变控件的value属性，会发现model还是为空。经过一番查找资料后，发现regularJs是只会静态加载一次的，当改变了控件的值（模拟用户输入评论）后，需要触发一下“input”事件，告诉后台值被修改了。

当做完这个步骤之后，基本的流程就做完了。



