Content Scripts 用来获取xml文件

Content scripts are files that run in the context of web pages. By using the standard **Document Object Model** (DOM), they are able to read details of the web pages the browser visits, make changes to them and pass information to their parent extension.

Understand Content Script Capabilities

Content scripts can access Chrome APIs used by their parent extension by exchanging **messages** and access information by making **cross-site XMLHttpRequests** to parent sites. They can also access the URL of an extension's file with `chrome.runtime.getURL()` and use the result the same as other URLs.

```
//Code for displaying <extensionDir>/images/myimage.png:
var imgURL = chrome.runtime.getURL("images/myimage.png");
document.getElementById("someImage").src = imgURL;
```

Additionally, content script can access the following chrome APIs directly:

- **i18n**
- **storage**
- **runtime**:
  - **connect**
  - **getManifest**
  - **getURL**
  - **id**
  - **onConnect**
  - **onMessage**
  - **sendMessage**

Content scripts are unable to access other APIs directly.

Work in Isolated Worlds

Content scripts live in an isolated world, allowing a content script to makes changes to its JavaScript environment without conflicting with the page or additional content scripts.

An extension may run in a web page with code similar to the example below.

Content script 的两种注入方式

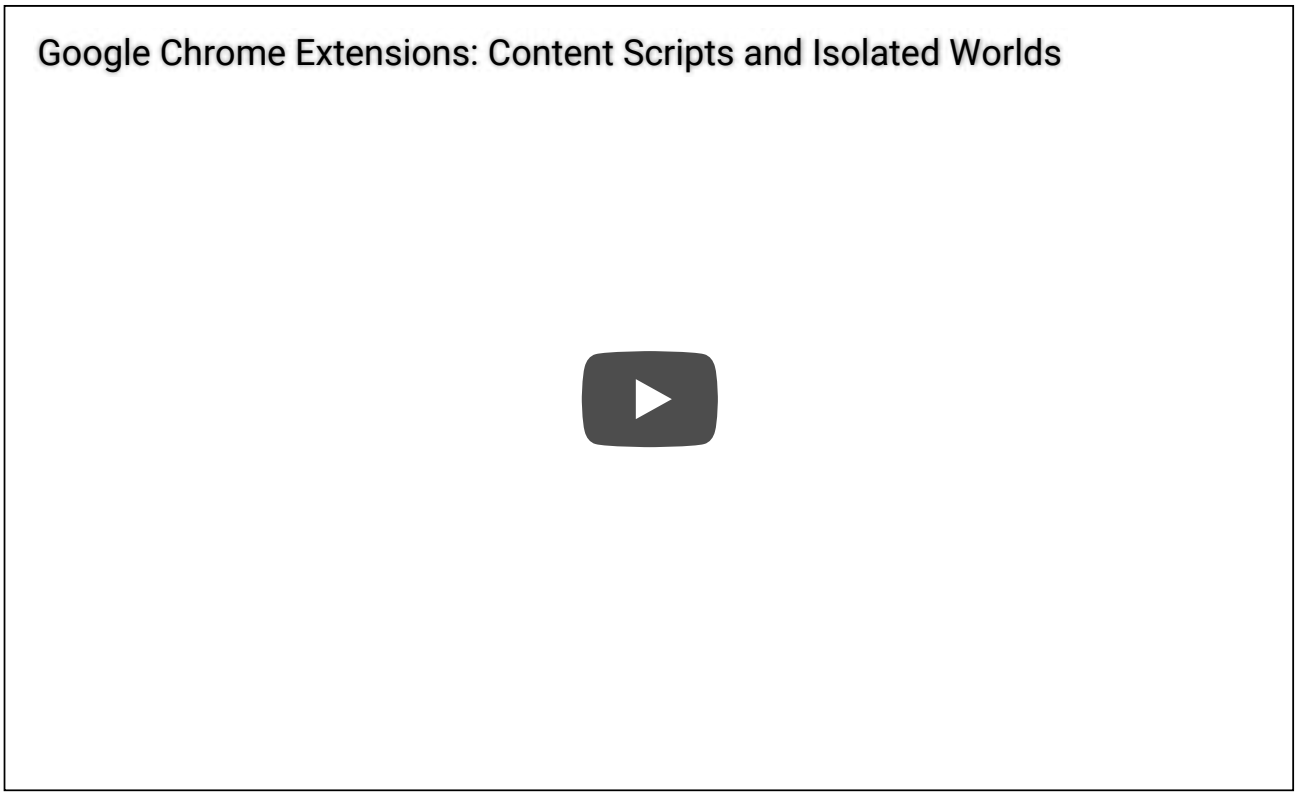
```
<html>
  <button id="mybutton">click me</button>
  <script>
    var greeting = "hello, ";
    var button = document.getElementById("mybutton");
    button.person_name = "Bob";
    button.addEventListener("click", function() {
      alert(greeting + button.person_name + ".");
    }, false);
  </script>
</html>
```

That extension could inject the following content script.

```
var greeting = "hola, ";
var button = document.getElementById("mybutton");
button.person_name = "Roberto";
button.addEventListener("click", function() {
  alert(greeting + button.person_name + ".");
}, false);
```

Both alerts would appear if the button was pressed.

Isolated worlds do not allow for content scripts, the extension, and the web page to access any variables or functions created by the others. This also gives content scripts the ability to enable functionality that should not be accessible to the web page.



Inject Scripts

Content Scripts can be **programmatically** or **declaratively** injected.

Inject Programmatically

Use programmatic injection for content scripts that need to run on specific occasions.

To inject a programmatic content script, provide the **activeTab** permission in the manifest. This grants secure access to the active site's host and temporary access to the **tabs** permission, enabling the content script to run on the current active tab without specifying **cross-origin permissions**.

```
{
  "name": "My extension",
  ...
  "permissions": [
    "activeTab"
  ],
  ...
}
```

Content scripts can be injected as code.

```
chrome.runtime.onMessage.addListener(
  function(message, callback) {
    if (message == “changeColor”){
      chrome.tabs.executeScript({
        code: 'document.body.style.backgroundColor="orange"'

      });
    }
 });
```

Or an entire file can be injected.

```
chrome.runtime.onMessage.addListener(
  function(message, callback) {
    if (message == “runContentScript”){
      chrome.tabs.executeScript({
        file: 'contentScript.js'
      });
    }
 });
```

Inject Declaratively

Use declarative injection for content scripts that should be automatically run on specified pages.

Declaratively injected scripts are registered in the manifest under the `"content_scripts"` field. They can include JavaScript files, CSS files or both. All auto-run content scripts must specify **match patterns**.

```
{
 "name": "My extension",
```

```
  ...
  "content_scripts": [
    {
      "matches": ["http://*.nytimes.com/*"],
      "css": ["myStyles.css"],
      "js": ["contentScript.js"]
    }
  ],
  ...
}
```

Name	Type	Description
`matches`	array of strings	*Required.* Specifies which pages this content script will be injected into. See **Match Patterns** for more details on the syntax of these strings and **Match patterns and globs** for information on how to exclude URLs.
`css`	array of strings	*Optional.* The list of CSS files to be injected into matching pages. These are injected in the order they appear in this array, before any DOM is constructed or displayed for the page.
`js`	array of strings	*Optional.* The list of JavaScript files to be injected into matching pages. These are injected in the order they appear in this array.

Exclude Matches and Globs

Specified page matching is customizable by including the following fields in the manifest registration.

Name	Type	Description
`exclude_matches` 不匹配什么	array of strings	*Optional.* Excludes pages that this content script would otherwise be injected into. See **Match Patterns** for more details on the syntax of these strings.
`include_globs`	array of strings	*Optional.* Applied after `matches` to include only those URLs that also match this glob. Intended to emulate the `@include` Greasemonkey keyword.
`exclude_globs`	array of string	*Optional.* Applied after `matches` to exclude URLs that match this glob. Intended to emulate the `@exclude` Greasemonkey keyword.

The content script will be injected into a page if its URL matches any `matches` pattern and any `include_globs` pattern, as long as the URL doesn't also match an `exclude_matches` or `exclude_globs` pattern.

Because the `matches` property is required, `exclude_matches`, `include_globs`, and `exclude_globs` can only be used to limit which pages will be affected.

The following extension would injected the content script into **http://www.nytimes.com/ health** but not into **http://www.nytimes.com/ business** .

```
{
  "name": "My extension",
  ...
  "content_scripts": [
    {
      "matches": ["http://*.nytimes.com/*"],
      "exclude_matches": ["*://*/*business*"],
      "js": ["contentScript.js"]
    }
  ],
  ...
}
```

Glob properties follow a different, more flexible syntax than **match patterns**. Acceptable glob strings are URLs that may contain "wildcard" asterisks and question marks. The asterisk ***** matches any string of any length, including the empty string, while the question mark **?** matches any single character.

For example, the glob **http:// ??? .example.com/foo/ *** matches any of the following:

- **http:// www .example.com/foo /bar**
- **http:// the .example.com/foo /**

However, it does *not* match the following:

- **http:// my .example.com/foo/bar**
- **http:// example .com/foo/**
- **http://www.example.com/foo**

This extension would inject the content script into **http:/www.nytimes.com/ arts /index.html** and **http://www.nytimes.com/ jobs /index.html** but not into **http://www.nytimes.com/ sports /index.html**.

```
{
  "name": "My extension",
```

```
  ...
  "content_scripts": [
    {
      "matches": ["http://*.nytimes.com/*"],
      "include_globs": ["*nytimes.com/???s/*"],
      "js": ["contentScript.js"]
    }
  ],
  ...
}
```

This extension would inject the content script into **http:// history .nytimes.com** and **http://.nytimes.com/ history** but not into **http:// science .nytimes.com** or **http://www.nytimes.com/ science** .

```
{
  "name": "My extension",
  ...
  "content_scripts": [
    {
      "matches": ["http://*.nytimes.com/*"],
      "exclude_globs": ["*science*"],
      "js": ["contentScript.js"]
    }
  ],
  ...
}
```

One, all, or some of these can be included to achieve the correct scope.

```
{
  "name": "My extension",
  ...
  "content_scripts": [
    {
      "matches": ["http://*.nytimes.com/*"],
      "exclude_matches": ["*://*/*business*"],
      "include_globs": ["*nytimes.com/???s/*"],
      "exclude_globs": ["*science*"],
      "js": ["contentScript.js"]
    }
  ],
  ...
}
```

Run Time 在什么时候运行

When JavaScript files are injected into the web page is controlled by the `run_at` field. The preffered and default field is `"document_idle"`, but can also be specified as `"document_start"` or `"document_end"` if needed.

```
{
  "name": "My extension",
  ...
  "content_scripts": [
    {
      "matches": ["http://*.nytimes.com/*"],
      "run_at": "document_idle",
      "js": ["contentScript.js"]
    }
  ],
  ...
}
```

Name	Type	Description
`document_idle`	string	*Prefered.* Use `"document_idle"` whenever possible. The browser chooses a time to inject scripts between `"document_end"` and immediately after the **`window.onload`** event fires. The exact moment of injection depends on how complex the document is and how long it is taking to load, and is optimized for page load speed. Content scripts running at `"document_idle"` do not need to listen for the `window.onload` event, they are guaranteed to run after the DOM is complete. If a script definitely needs to run after `window.onload`, the extension can check if `onload` has already fired by using the **`document.readyState`** property.
`document_start`	string	Scripts are injected after any files from `css`, but before any other DOM is constructed or any other script is run.
`document_end`	string	Scripts are injected immediately after the DOM is complete, but before subresources like images and frames have loaded.

Specify Frames

The `"all_frames"` field allows the extension to specify if JavaScript and CSS files should be injected into all frames matching the specified URL requirements or only into the topmost frame in a tab.

```
{
  "name": "My extension",
  ...
  "content_scripts": [
    {
      "matches": ["http://*.nytimes.com/*"],
      "all_frames": true,
      "js": ["contentScript.js"]
    }
  ],
  ...
}
```

Name	Type	Description
`all_frames`	boolean	*Optional.* Defaults to `false`, meaning that only the top frame is matched. If specified `true`, it will inject into all frames, even if the frame is not the topmost frame in the tab. Each frame is checked independently for URL requirements, it will not inject into child frames if the URL requirements are not met.

Communication with the embedding page

Although the execution environments of content scripts and the pages that host them are isolated from each other, they share access to the page's DOM. If the page wishes to communicate with the content script, or with the extension via the content script, it must do so through the shared DOM.

An example can be accomplished using **`window.postMessage`**:

```
var port = chrome.runtime.connect();

window.addEventListener("message", function(event) {

  // We only accept messages from ourselves
  if (event.source != window)
    return;

  if (event.data.type && (event.data.type == "FROM_PAGE")) {
    console.log("Content script received: " + event.data.text);
```

```
    port.postMessage(event.data.text);
  }
}, false);
```

```
document.getElementById("theButton").addEventListener("click",
    function() {
  window.postMessage({ type: "FROM_PAGE", text: "Hello from the webpage!" }, "*");
}, false);
```

The non-extension page, example.html, posts messages to itself. This message is intercepted and inspected by the content script and then posted to the extension process. In this way, the page establishes a line of communication to the extension process. The reverse is possible through similar means.

Stay Secure

While isolated worlds provide a layer of protection, using content scripts can create vulnerabilities in an extension and the web page. If the content script receives content from a separate website, such as making an **XMLHttpRequest**, be careful to filter content **cross-site scripting** attacks before injecting it. Only communicate over HTTPS in order to avoid **"man-in-the-middle"** attacks.

Be sure to filter for malicious web pages. For example, the following patterns are dangerous:

```
var data = document.getElementById("json-data")
// WARNING! Might be evaluating an evil script!
var parsed = eval("(" + data + ")")
```

```
var elmt_id = ...
// WARNING! elmt_id might be "); ... evil script ... //"!
window.setTimeout("animate(" + elmt_id + ")", 200);
```

Instead, prefer safer APIs that do not run scripts:

```
var data = document.getElementById("json-data")
// JSON.parse does not evaluate the attacker's scripts.
var parsed = JSON.parse(data);
```

```
var elmt_id = ...
// The closure form of setTimeout does not evaluate scripts.
window.setTimeout(function() {
```

```
  animate(elmt_id);
}, 200);
```

*Content available under the **CC-By 3.0 license***

Google **Terms of Service** **Privacy Policy** **Report a content bug** **Add us on**