```
Chrome Extension 组成:
    1. html 界面
    2. js 后台逻辑
    3. 配置文件
    4. 资源文件
```

Overview

Once you've finished this page and the **Getting Started** tutorial, you'll be all set to start writing extensions.

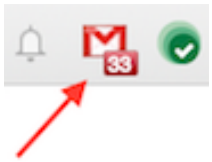
The basics

An extension is a zipped bundle of files—HTML, CSS, JavaScript, images, and anything else you need—that adds functionality to the Google Chrome browser. Extensions are essentially web pages, and they can use all the **APIs that the browser provides to web pages**, from XMLHttpRequest to JSON to HTML5.

Extensions can interact with web pages or servers using **content scripts** or **cross-origin XMLHttpRequests**. Extensions can also interact programmatically with browser features such as **bookmarks** and **tabs**.

Extension UIs

Many extensions—but not Chrome Apps—add UI to Google Chrome in the form of **browser actions** or **page actions**. Each extension can have at most one browser action or page action. Choose a **browser action** when the extension is relevant to most pages. Choose a **page action** when the extension's icon should be active or inactive (and grayed out), depending on the page.

		
This **Google Mail Checker extension** uses a *browser action*.	This **Mappy extension** uses a *page action* and *content script* (code injected into a web page).	This **Set Page Color extension** features a browser action that, when clicked, shows a *popup*.

Extensions (and Chrome Apps) can also present a UI in other ways, such as adding to the Chrome context menu, providing an options page, or using a content script that changes how pages look. See the **Developer's Guide** for a complete list of extension features, with links to implementation details for each one.

Files

Each extension has the following files:

- A **manifest file**
- One or more **HTML files** (unless the extension is a theme)
- *Optional:* One or more **JavaScript files**
- *Optional:* Any other files your extension needs—for example, image files

While you're working on your extension, you put all these files into a single folder. When you distribute your extension, the contents of the folder are packaged into a special ZIP file that has a `.crx` suffix. If you upload your extension using the **Chrome Developer Dashboard**, the `.crx` file is created for you. For details on distributing extensions, see **Hosting**.

Referring to files

You can put any file you like into an extension, but how do you use it? Usually, you can refer to the file using a relative URL, just as you would in an ordinary HTML page. Here's an example of referring to a file named `myimage.png` that's in a subfolder named `images`.

```
<img src="images/myimage.png">
```

As you might notice while you use the Google Chrome debugger, every file in an extension is also accessible by an absolute URL like this:

chrome-extension://*<extensionID>***/***<pathToFile>*

In that URL, the *<extensionID>* is a unique identifier that the extension system generates for each extension. You can see the IDs for all your loaded extensions by going to the URL **chrome://extensions**. The *<pathToFile>* is the location of the file under the extension's top folder; it's the same as the relative URL.

While you're working on an extension (before it's packaged), the extension ID can change. Specifically, the ID of an unpacked extension will change if you load the extension from a different directory; the ID will change again when you package the extension. If your extension's code needs to specify the full path to a file within the extension, you can use the `@@extension_id` **predefined message** to avoid hardcoding the ID during development.

When you package an extension (typically, by uploading it with the dashboard), the extension gets a permanent ID, which remains the same even after you update the extension. Once the extension ID is permanent, you can change all occurrences of `@@extension_id` to use the real ID.

The manifest file

The manifest file, called `manifest.json`, gives information about the extension, such as the most important files and the capabilities that the extension might use. Here's a typical manifest file for a browser action that uses information from google.com:

```
{
  "name": "My Extension",
  "version": "2.1",
  "description": "Gets information from Google.",
  "icons": { "128": "icon_128.png" },
  "background": {
    "persistent": false,
    "scripts": ["bg.js"]
  },
  "permissions": ["http://*.google.com/", "https://*.google.com/"],
  "browser_action": {
    "default_title": "",
    "default_icon": "icon_19.png",
    "default_popup": "popup.html"
  }
}
```

For details, see **Manifest Files**.

Architecture

Many extensions have a *background page*, an invisible page that holds the main logic of the extension. An extension can also contain other pages that present the extension's UI. If an extension needs to interact with web pages that the user loads (as opposed to pages that are included in the extension), then the extension must use a content script.

The background page

Background pages defined by `background.html` can include JavaScript code that controls the behavior of the extension. There are two types of background pages: **persistent background pages**, and **event pages**. Persistent background pages, as the name suggests, are always open. Event pages are opened and closed as needed. Unless you absolutely need your background page to run all the time, prefer to use an event page.

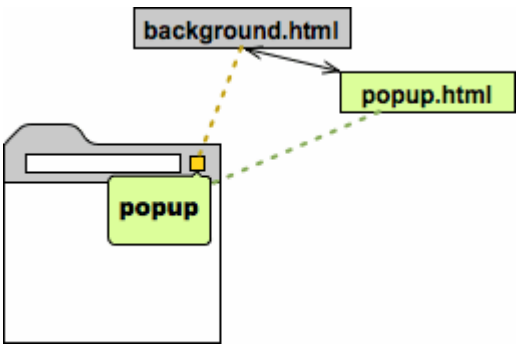
See **Event Pages** and **Background Pages** for more details.

UI pages

Extensions can contain ordinary HTML pages that display the extension's UI. For example, a browser action can have a popup, which is implemented by an HTML file. Any extension can have an options page, which lets users customize how the extension works. Another type of special page is the override page. And finally, you can use **tabs.create** or `window.open()` to display any other HTML files that are in the extension.

The HTML pages inside an extension have complete access to each other's DOMs, and they can invoke functions on each other.

The following figure shows the architecture of a browser action's popup. The popup's contents are a web page defined by an HTML file (`popup.html`). This extension also happens to have a background page (`background.html`). The popup doesn't need to duplicate code that's in the background page because the popup can invoke functions on the background page.



background 相当于基类

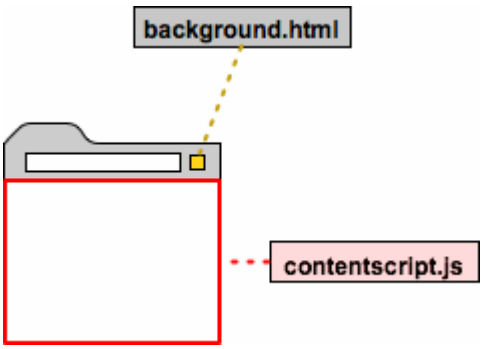
See **Browser Actions**, **Options**, **Override Pages**, and the **Communication between pages** section for more details.

Content scripts

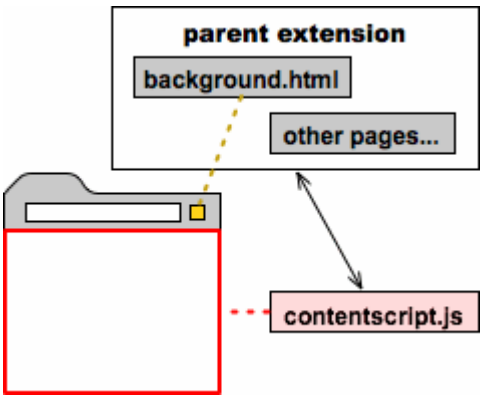
If your extension needs to interact with web pages, then it needs a *content script*. A content script is some JavaScript that executes in the context of a page that's been loaded into the browser. Think of a content script as part of that loaded page, not as part of the extension it was packaged with (its *parent extension*).

从这里读取页面的信息

Content scripts can read details of the web pages the browser visits, and they can make changes to the pages. In the following figure, the content script can read and modify the DOM for the displayed web page. It cannot, however, modify the DOM of its parent extension's background page.



Content scripts aren't completely cut off from their parent extensions. A content script can exchange messages with its parent extension, as the arrows in the following figure show. For example, a content script might send a message whenever it finds an RSS feed in a browser page. Or a background page might send a message asking a content script to change the appearance of its browser page.



For more information, see **Content Scripts**.

Using the chrome.* APIs

In addition to having access to all the APIs that web pages and apps can use, extensions can also use Chrome-only APIs (often called *chrome.* APIs*) that allow tight integration with the browser. For example, any extension or web app can use the standard `window.open()` method to open a URL. But if you want to specify which window that URL should be displayed in, your extension can use the Chrome-only **tabs.create** method instead.

跳转到一个网页

Asynchronous vs. synchronous methods

大部分的Chrome API都是异步加载的，不需要等待，只需要在callback函数中声明操作即可

Most methods in the chrome.* APIs are **asynchronous**: they return immediately, without waiting for the operation to finish. If you need to know the outcome of that operation, then you pass a callback function

into the method. That callback is executed later (potentially *much* later), sometime after the method returns. Here's an example of the signature for an asynchronous method:

```
chrome.tabs.create(object createProperties, function callback)
```

Other chrome.* methods are **synchronous**. Synchronous methods never have a callback because they don't return until they've completed all their work. Often, synchronous methods have a return type. Consider the **runtime.getURL** method:

```
string chrome.runtime.getURL()
```

This method has no callback and a return type of `string` because it synchronously returns the URL and performs no other, asynchronous work.

Example: Using a callback

Say you want to navigate the user's currently selected tab to a new URL. To do this, you need to get the current tab's ID (using **tabs.query**) and then make that tab go to the new URL (using **tabs.update**).

If `query()` were synchronous, you might write code like this:

```
//THIS CODE DOESN'T WORK
var tab = chrome.tabs.query({'active': true}); //WRONG!!!
chrome.tabs.update(tab.id, {url:newUrl});
someOtherFunction();
```

That approach fails because `query()` is asynchronous. It returns without waiting for its work to complete, and it doesn't even return a value (although some asynchronous methods do). You can tell that `query()` is asynchronous by the *callback* parameter in its signature:

```
chrome.tabs.query(object queryInfo, function callback)
```

To fix the preceding code, you must use that callback parameter. The following code shows how to define a callback function that gets the results from `query()` (as a parameter named `tab`) and calls `update()`.

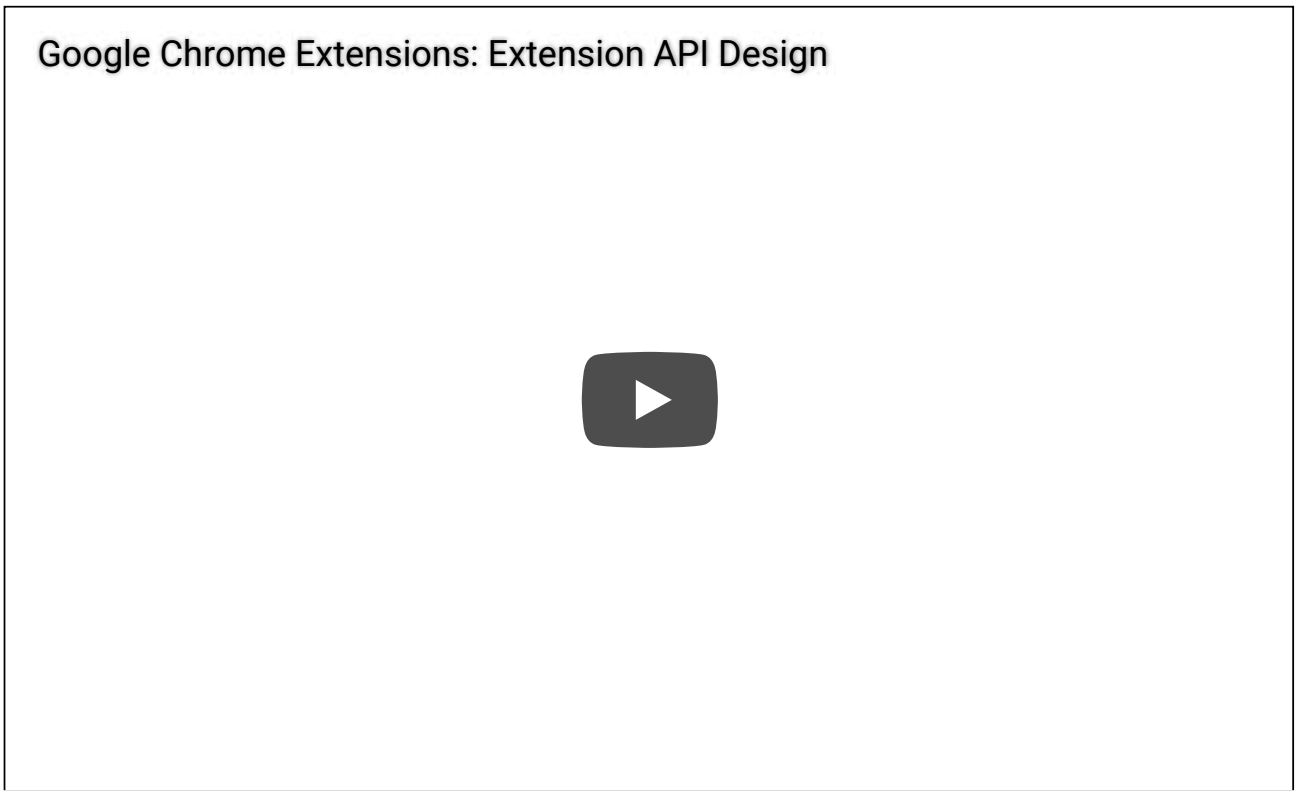
```
//THIS CODE WORKS     update 一个网页，指向另一个url
chrome.tabs.query({'active': true}, function(tabs) {
  chrome.tabs.update(tabs[0].id, {url: newUrl});
});
someOtherFunction();
```

In this example, the lines are executed in the following order: 1, 4, 2. The callback function specified to `query()` is called (and line 2 executed) only after information about the currently selected tab is available,

which is sometime after `query()` returns. Although `update()` is asynchronous, this example doesn't use its callback parameter, since we don't do anything about the results of the update.

More details

For more information, see the **chrome.* API docs** and watch this video:



Communication between pages

The HTML pages within an extension often need to communicate. Because all of an extension's pages execute in same process on the same thread, the pages can make direct function calls to each other.

To find pages in the extension, use `chrome.extension` methods such as `getViews()` and `getBackgroundPage()`. Once a page has a reference to other pages within the extension, the first page can invoke functions on the other pages, and it can manipulate their DOMs.

Saving data and incognito mode

Extensions can save data using the **storage** API, the HTML5 **web storage API** (such as `localStorage`) or by making server requests that result in saving data. Whenever you want to save something, first consider whether it's from a window that's in incognito mode. By default, extensions don't run in incognito windows. You need to consider what a user expects from your extension when the browser is incognito.

隐身模式

Incognito mode promises that the window will leave no tracks. When dealing with data from incognito windows, do your best to honor this promise. For example, if your extension normally saves browsing

history to the cloud, don't save history from incognito windows. On the other hand, you can store your extension's settings from any window, incognito or not.

> **Rule of thumb:** If a piece of data might show where a user has been on the web or what the user has done, don't store it if it's from an incognito window.

To detect whether a window is in incognito mode, check the `incognito` property of the relevant **tabs.Tab** or **windows.Window** object. For example:

```
function saveTabData(tab, data) {  // 隐身模式下存储原则
  if (tab.incognito) {
    chrome.runtime.getBackgroundPage(function(bgPage) {
      bgPage[tab.url] = data;       // Persist data ONLY in memory
    });
  } else {
    localStorage[tab.url] = data;  // OK to store data
  }
}
```

Now what?

Now that you've been introduced to extensions, you should be ready to write your own. Here are some ideas for where to go next:

- **Tutorial: Getting Started**
- **Tutorial: Debugging**
- **Developer's Guide**
- **Samples**
- **Videos**, such as **Extension Message Passing**