1 H5 缓存机制介绍

H5，即 HTML5，是新一代的 HTML 标准，加入很多新的特性。离线存储（也可称为缓存机制）是其中一个非常重要的特性。H5 引入的离线存储，这意味着 web 应用可进行缓存，并可在没有因特网连接时进行访问。

H5 应用程序缓存为应用带来三个优势：

- 离线浏览 用户可在应用离线时使用它们
- 速度 已缓存资源加载得更快
- 减少服务器负载 浏览器将只从服务器下载更新过或更改过的资源。

根据标准，到目前为止，H5 一共有 6 种缓存机制，有些是之前已有，有些是 H5 才新加入的。

1. **浏览器缓存机制**
2. **Dom Storgage（Web Storage）存储机制**
3. **Web SQL Database 存储机制**
4. **Application Cache（AppCache）机制**
5. **Indexed Database （IndexedDB）**
6. **File System API**

下面我们首先分析各种缓存机制的原理、用法及特点；然后针对 Anroid 移动端 Web 性能加载优化的需求，看如果利用适当缓存机制来提高 Web 的加载性能。

2 H5 缓存机制原理分析

2.1 浏览器缓存机制

浏览器缓存机制是指通过 HTTP 协议头里的 Cache-Control（或 Expires）和 Last-Modified（或 Etag）等字段来控制文件缓存的机制。这应该是 WEB 中最早的缓存机制了，是在 HTTP 协议中实现的，有点不同于 Dom Storage、AppCache 等缓存机制，但本质上是一样的。可以理解为，一个是协议层实现的，一个是应用层实现的。

Cache-Control 用于控制文件在本地缓存有效时长。最常见的，比如服务器回包：Cache-Control:max-age=600 表示文件在本地应该缓存，且有效时长是 600 秒（从发出请求算起）。在接下来 600 秒内，如果有请求这个资源，浏览器不会发出 HTTP 请求，而是直接使用本地缓存的文件。

Last-Modified 是标识文件在服务器上的最新更新时间。下次请求时，如果文件缓存过期，浏览器通过 If-Modified-Since 字段带上这个时间，发送给服务器，由服务器比较时间戳来判断文件是否有修改。如果没有修改，服务器返回 304 告诉浏览器继续使用缓存；如果有修改，则返回 200，同时返回最新的文件。

Cache-Control 通常与 Last-Modified 一起使用。一个用于控制缓存有效时间，一个在缓存失效后，向服务查询是否有更新。

Cache-Control 还有一个同功能的字段：Expires。Expires 的值一个绝对的时间点，如：Expires: Thu, 10 Nov 2015 08:45:11 GMT，表示在这个时间点之前，缓存都是有效的。

Expires 是 HTTP1.0 标准中的字段，Cache-Control 是 HTTP1.1 标准中新加的字段，功能一样，都是控制缓存的有效时间。当这两个字段同时出现时，Cache-Control 是高优化级的。

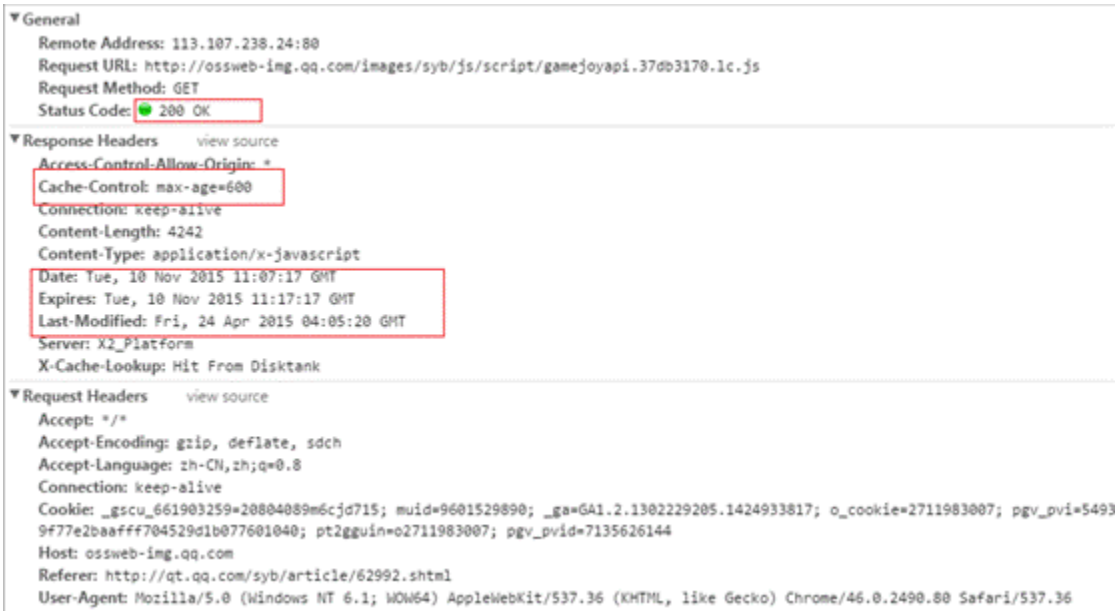
Etag 也是和 Last-Modified 一样，对文件进行标识的字段。不同的是，Etag 的取值是一个对文件进行标识的特征字串。在向服务器查询文件是否有更新时，浏览器通过 If-None-Match 字段把特征字串发送给服务器，由服务器和文件最新特征字串进行匹配，来判断文件是否有更新。没有更新回包 304，有更新回包 200。Etag 和 Last-Modified 可根据需求使用一个或两个同时使用。两个同时使用时，只要满足基中一个条件，就认为文件没有更新。

另外有两种特殊的情况：

- 手动刷新页面（F5），浏览器会直接认为缓存已经过期（可能缓存还没有过期），在请求中加上字段：Cache-Control:max-age=0，发包向服务器查询是否有文件是否有更新。
- 强制刷新页面（Ctrl+F5），浏览器会直接忽略本地的缓存（有缓存也会认为本地没有缓存），在请求中加上字段：Cache-Control:no-cache（或 Pragma:no-cache），发包向服务重新拉取文件。

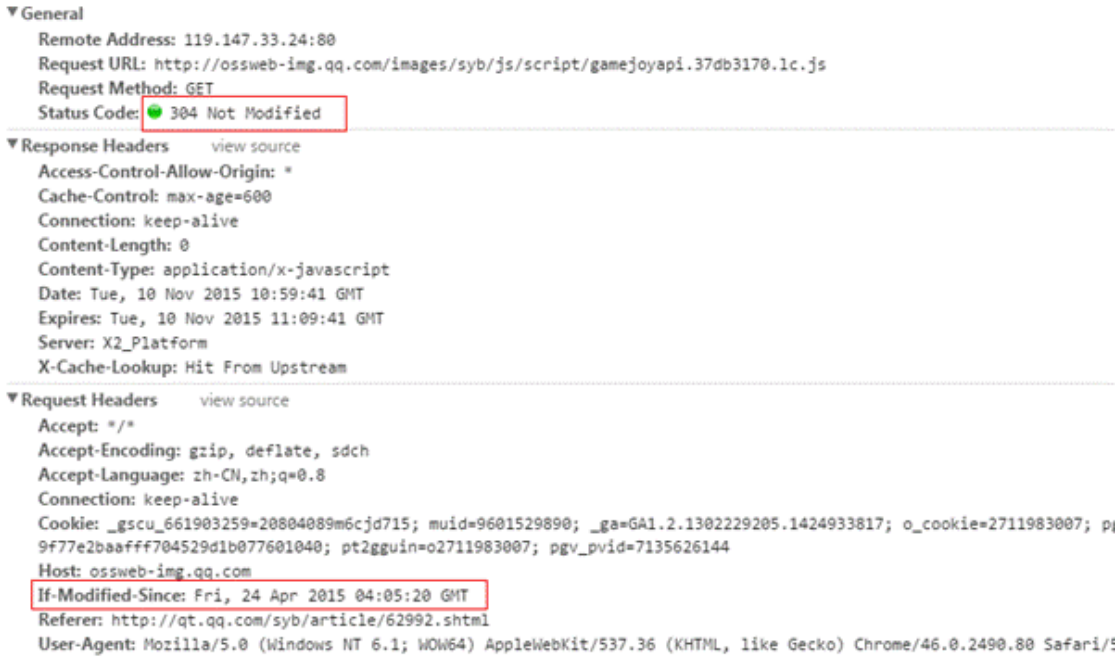
下面是通过 Google Chrome 浏览器（用其他浏览器+抓包工具也可以）自带的开发者工具，对一个资源文件不同情况请求与回包的截图。

首次请求：200



缓存有效期内请求：200(from cache)

缓存过期后请求：304（Not Modified)



一般浏览器会将缓存记录及缓存文件存在本地 Cache 文件夹中。Android 下 App 如果使用 Webview，缓存的文件记录及文件内容会存在当前 app 的 data 目录中。

分析：Cache-Control 和 Last-Modified 一般用在 Web 的静态资源文件上，如 JS、CSS 和一些图像文件。通过设置资源文件缓存属性，对提高资源文件加载速度，节省流量很有意义，特别是移动网络环境。但问题是：缓存有效时长该如何设置？如果设置太短，就起不到缓存的使用；如果设置的太长，在资源文件有更新时，浏览器如果有缓存，则不能及时取到最新的文件。

Last-Modified 需要向服务器发起查询请求，才能知道资源文件有没有更新。虽然服务器可能返回 304 告诉没有更新，但也还有一个请求的过程。对于移动网络，这个请求可能是比较耗时的。有一种说法叫“消灭 304”，指的就是优化掉 304 的请求。

抓包发现，带 if-Modified-Since 字段的请求，如果服务器回包 304，回包带有 Cache-Control:max-age 或 Expires 字段，文件的缓存有效时间会更新，就是文件的缓存会重新有效。304 回包后如果再请求，则又直接使用缓存文件了，不再向服务器查询文件是否更新了，除非新的缓存时间再次过期。

另外，Cache-Control 与 Last-Modified 是浏览器内核的机制，一般都是标准的实现，不能更改或设置。以 QQ 浏览器的 X5 为例，Cache-Control 与 Last-Modified 缓存不能禁用。缓存容量是 12MB，不分 HOST，过期的缓存会最先被清除。如果都没过期，应该优先清最早的缓存或最快到期的或文件大小最大的；过期缓存也有可能还是有效的，清除缓存会导致资源文件的重新拉取。

还有，浏览器，如 X5，在使用缓存文件时，是没有对缓存文件内容进行校验的，这样缓存文件内容被修改的可能。

分析发现，浏览器的缓存机制还不是非常完美的缓存机制。完美的缓存机制应该是这样的：

1. 缓存文件没更新，尽可能使用缓存，不用和服务器交互；
2. 缓存文件有更新时，第一时间能使用到新的文件；
3. 缓存的文件要保持完整性，不使用被修改过的缓存文件；
4. 缓存的容量大小要能设置或控制，缓存文件不能因为存储空间限制或过期被清除。

以 X5 为例，第 1、2 条不能同时满足，第 3、4 条都不能满足。

在实际应用中，为了解决 Cache-Control 缓存时长不好设置的问题，以及为了”消灭 304“，

Web 前端采用的方式是：

1. 在要缓存的资源文件名中加上版本号或文件 MD5 值字串，如 common.d5d02a02.js，common.v1.js，同时设置 Cache-Control:max-age=31536000，也就是一年。在一年时间内，资源文件如果本地有缓存，就会使用缓存；也就不会有 304 的回包。
2. 如果资源文件有修改，则更新文件内容，同时修改资源文件名，如 common.v2.js，html 页面也会引用新的资源文件名。

通过这种方式，实现了：缓存文件没有更新，则使用缓存；缓存文件有更新，则第一时间使用最新文件的目的。即上面说的第 1、2 条。第 3、4 条由于浏览器内部机制，目前还无法满足。

2.2 Dom Storage 存储机制

DOM 存储是一套在 Web Applications 1.0 规范中首次引入的与存储相关的特性的总称，现在已经分离出来，单独发展成为独立的 W3C Web 存储规范。 DOM 存储被设计为用来提供一个更大存储量、更安全、更便捷的存储方法，从而可以代替掉将一些不需要让服务器知道的信息存储到 cookies 里的这种传统方法。

上面一段是对 Dom Storage 存储机制的官方表述。看起来，Dom Storage 机制类似 Cookies，但有一些优势。

Dom Storage 是通过存储字符串的 Key/Value 对来提供的，并提供 5MB （不同浏览器可能不同，分 HOST)的存储空间（Cookies 才 4KB)。另外 Dom Storage 存储的数据在本地，不像 Cookies，每次请求一次页面，Cookies 都会发送给服务器。

DOM Storage 分为 sessionStorage 和 localStorage。localStorage 对象和 sessionStorage 对象使用方法基本相同，它们的区别在于作用的范围不同。sessionStorage 用来存储与页面相关的数据，它在页面关闭后无法使用。而 localStorage 则持久存在，在页面关闭后也可以使用。

Dom Storage 提供了以下的存储接口：

```
interface Storage {
readonly attribute unsigned long length;
[IndexGetter] DOMString key(in unsigned long index);
[NameGetter] DOMString getItem(in DOMString key);
[NameSetter] void setItem(in DOMString key, in DOMString data);
[NameDeleter] void removeItem(in DOMString key);
void clear();
};
```

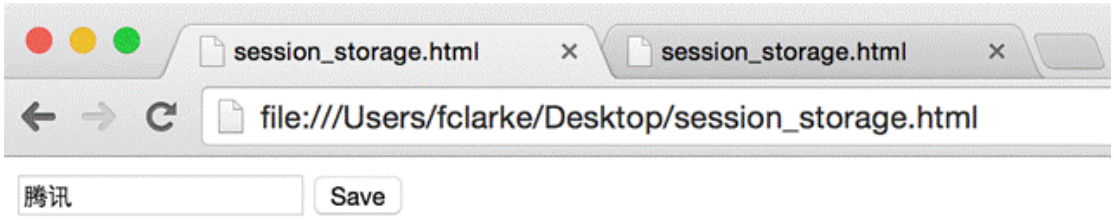
sessionStorage 是个全局对象，它维护着在页面会话(page session)期间有效的存储空间。只要浏览器开着，页面会话周期就会一直持续。当页面重新载入(reload)或者被恢复(restores)时，页面会话也是一直存在的。每在新标签或者新窗口中打开一个新页面，都会初始化一个新的会话。

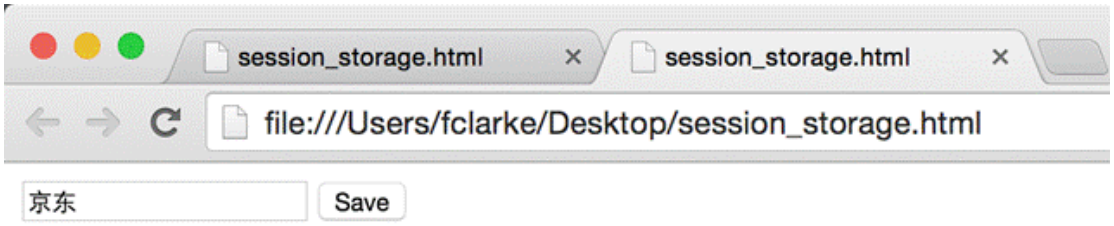
```
// 当页面刷新时，从 sessionStorage 恢复之前输入的内容
window.onload = function(){
  if (window.sessionStorage) {
    var name = window.sessionStorage.getItem("name");
    if (name != "" || name != null){
      document.getElementById("name").value = name;
    }
  }
};
```

```
10      // 将数据保存到 sessionStorage 对象中
11      function saveToStorage() {
12        if (window.sessionStorage) {
13           var name = document.getElementById("name").value;
14           window.sessionStorage.setItem("name", name);
15           window.location.href="session_storage.html";
16         }
17      }
```

当浏览器被意外刷新的时候，一些临时数据应当被保存和恢复。sessionStorage 对象在处理这种情况的时候是最有用的。比如恢复我们在表单中已经填写的数据。

把上面的代码复制到 session_storage.html（也可以从附件中直接下载）页面中，用 Google Chrome 浏览器的不同 PAGE 或 WINDOW 打开，在输入框中分别输入不同的文字，再点击“Save”，然后分别刷新。每个 PAGE 或 WINDOW 显示都是当前 PAGE 输入的内容，互不影响。关闭 PAGE，再重新打开，上一次输入保存的内容已经没有了。

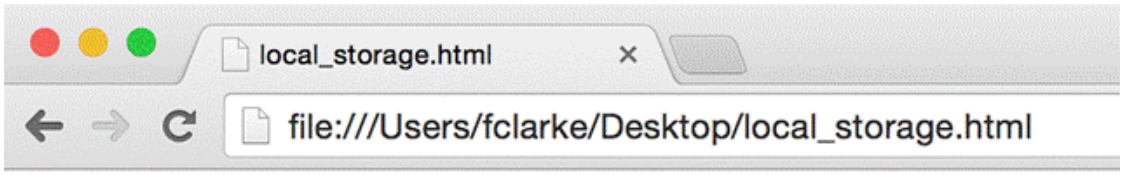


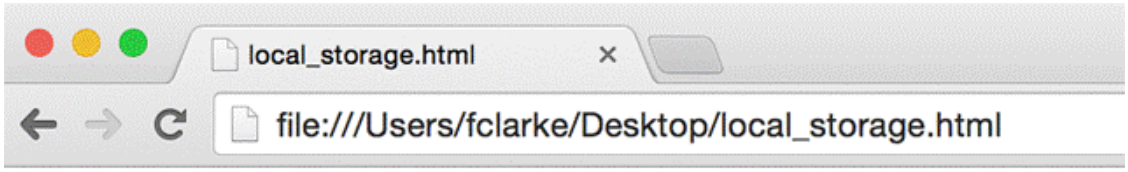


Local Storage 的接口、用法与 Session Storage 一样，唯一不同的是：Local Storage 保存的数据是持久性的。当前 PAGE 关闭（Page Session 结束后），保存的数据依然存在。重新打开 PAGE，上次保存的数据可以获取到。另外，Local Storage 是全局性的，同时打开两个 PAGE 会共享一份存数据，在一个 PAGE 中修改数据，另一个 PAGE 中是可以感知到的。

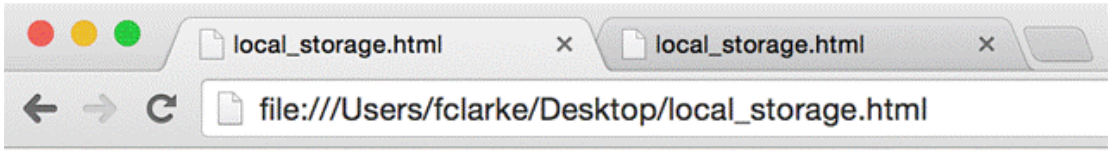
```
1       //通过 localStorage 直接引用 key, 另一种写法，等价于：
2       //localStorage.getItem("pageLoadCount");
3       //localStorage.setItem("pageLoadCount", value);
4       if (!localStorage.pageLoadCount)
5     localStorage.pageLoadCount = 0;
6         localStorage.pageLoadCount = parseInt(localStorage.pageLoadCount) + 1;
7         document.getElementById('count').textContent = localStorage.pageLoadCount;    You have viewed this page
8       an untold number of    time(s).
```

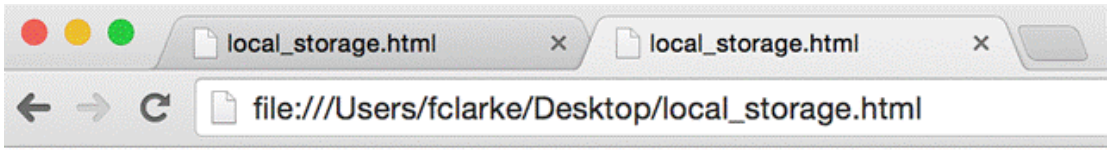
将上面代码复制到 local_storage.html 的页面中，用浏览器打开，pageLoadCount 的值是 1；关闭 PAGE 重新打开，pageLoadCount 的值是 2。这是因为第一次的值已经保存了。





用两个 PAGE 同时打开 local_storage.html，并分别交替刷新，发现两个 PAGE 是共享一个 pageLoadCount 的。





分析：Dom Storage 给 Web 提供了一种更录活的数据存储方式，存储空间更大（相对 Cookies)，用法也比较简单，方便存储服务器或本地的一些临时数据。

从 DomStorage 提供的接口来看，DomStorage 适合存储比较简单的数据，如果要存储结构化的数据，可能要借助 JASON 了，将要存储的对象转为 JASON 字串。不太适合存储比较复杂或存储空间要求比较大的数据，也不适合存储静态的文件等。

在 Android 内嵌 Webview 中，需要通过 Webview 设置接口启用 Dom Storage。

```
1    WebView myWebView = (WebView) findViewById(R.id.webview);
2    WebSettings webSettings = myWebView.getSettings();
3    webSettings.setDomStorageEnabled(true);
```

拿 Android 类比的话，Web 的 Dom Storage 机制类似于 Android 的 SharedPreference 机制。

2.3 Web SQL Database 存储机制

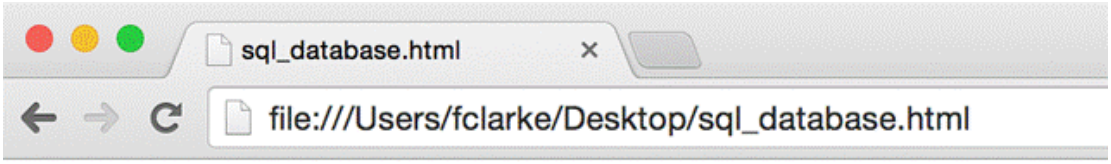
H5 也提供基于 SQL 的数据库存储机制，用于存储适合数据库的结构化数据。根据官方的标准文档，Web SQL Database 存储机制不再推荐使用，将来也不再维护，而是推荐使用 AppCache 和 IndexedDB。

现在主流的浏览器(点击查看浏览器支持情况)都还是支持 Web SQL Database 存储机制的。Web SQL Database 存储机制提供了一组 API 供 Web App 创建、存储、查询数据库。

下面通过简单的例子，演示下 Web SQL Database 的使用。

```
1         if(window.openDatabase){
2          //打开数据库，如果没有则创建
3          var db = openDatabase('mydb', '1.0', 'Test DB', 2 * 1024);
4          //通过事务，创建一个表，并添加两条记录
5          db.transaction(function (tx) {
6              tx.executeSql('CREATE TABLE IF NOT EXISTS LOGS (id unique, log)');
7              tx.executeSql('INSERT INTO LOGS (id, log) VALUES (1, "foobar")');
8              tx.executeSql('INSERT INTO LOGS (id, log) VALUES (2, "logmsg")');
9          });
10         //查询表中所有记录，并展示出来
11         db.transaction(function (tx) {
12           tx.executeSql('SELECT * FROM LOGS', [], function (tx, results) {
13             var len = results.rows.length, i;
14             msg = "Found rows: " + len + "";
15             for(i=0; i" + results.rows.item(i).log + "";
16             }
17             document.querySelector('#status').innerHTML =  msg;
18             }, null);
19         });
20     }Status Message
```

将上面代码复制到 sql_database.html 中，用浏览器打开，可看到下面的内容。



Found rows: 2

foobar

logmsg

官方建议浏览器在实现时，对每个 HOST 的数据库存储空间作一定限制，建议默认是 5MB（分 HOST）的配额；达到上限后，可以申请更多存储空间。另外，现在主流浏览器 SQL Database 的实现都是基于 SQLite。

分析：SQL Database 的主要优势在于能够存储结构复杂的数据，能充分利用数据库的优势，可方便对数据进行增加、删除、修改、查询。由于 SQL 语法的复杂性，使用起来麻烦一些。SQL Database 也不太适合做静态文件的缓存。

在 Android 内嵌 Webview 中，需要通过 Webview 设置接口启用 SQL Database，同时还要设置数据库文件的存储路径。

```
1    WebView myWebView = (WebView) findViewById(R.id.webview);
2    WebSettings webSettings = myWebView.getSettings();
3    webSettings.setDatabaseEnabled(true);
4    final String dbPath = getApplicationContext().getDir("db", Context.MODE_PRIVATE).getPath();
5    webSettings.setDatabasePath(dbPath);
```

Android 系统也使用了大量的数据库用来存储数据，比如联系人、短消息等；数据库的格式也 SQLite。Android 也提供了 API 来操作 SQLite。Web SQL Database 存储机制就是通过提供一组 API，借助浏览器的实现，将这种 Native 的功能提供给了 Web App。

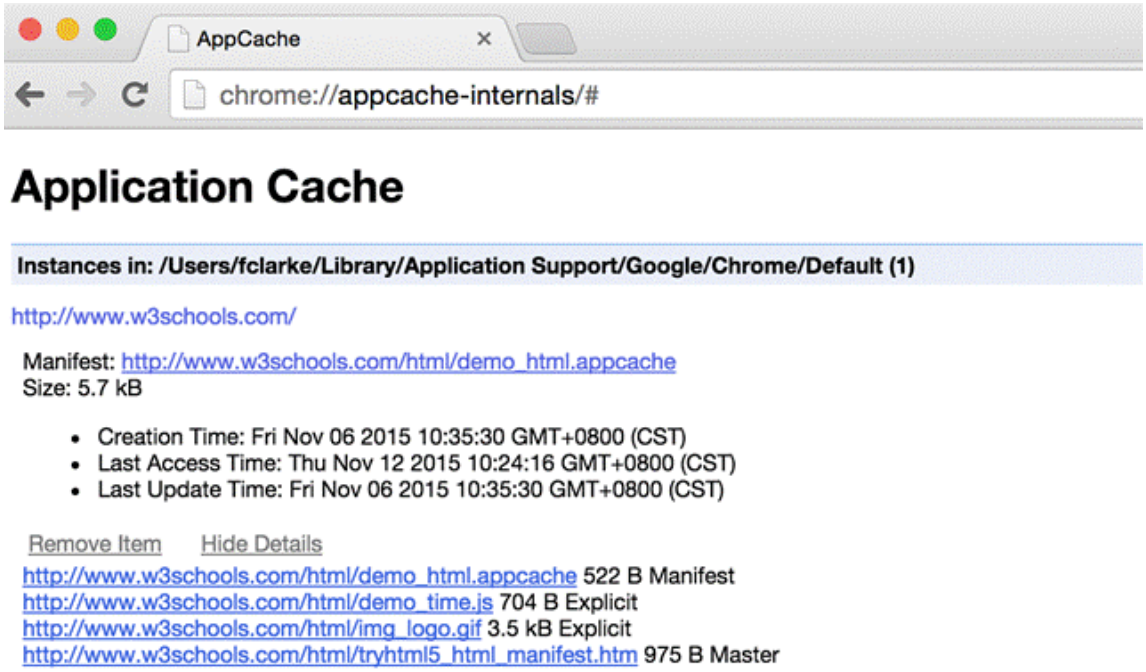
2.4 Application Cache 机制

Application Cache（简称 AppCache)似乎是为支持 Web App 离线使用而开发的缓存机制。它的缓存机制类似于浏览器的缓存（Cache-Control 和 Last-Modified）机制，都是以文件为单位进行缓存，且文件有一定更新机制。但 AppCache 是对浏览器缓存机制的补充，不是替代。先拿 W3C 官方的一个例子，说下 AppCache 机制的用法与功能。

上面 HTML 文档，引用外部一个 JS 文件和一个 GIF 图片文件，在其 HTML 头中通过 manifest 属性引用了一个 appcache 结尾的文件。

我们在 Google Chrome 浏览器中打开这个 HTML 链接，JS 功能正常，图片也显示正常。禁用网络，关闭浏览器重新打开这个链接，发现 JS 工作正常，图片也显示正常。当然也有可能是浏览缓存起的作用，我们可以在文件的浏览器缓存过期后，禁用网络再试，发现 HTML 页面也是正常的。

通过 Google Chrome 浏览器自带的工具，我们可以查看已经缓存的 AppCache（分 HOST)。

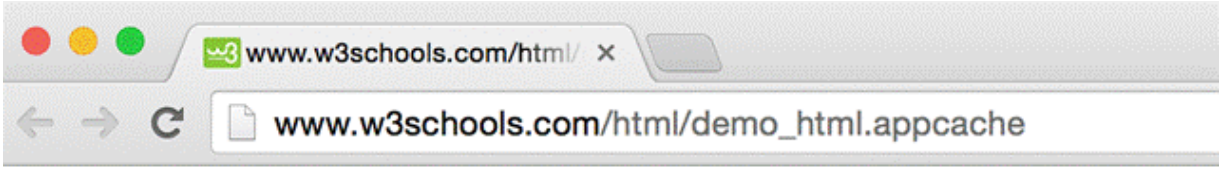


上面截图中的缓存，就是我们刚才打开 HTML 的页面 AppCache。从截图中看，HTML 页面及 HTML 引用的 JS、GIF 图像文件都被缓存了；另外 HTML 头中 manifest 属性引用的 appcache 文件也缓存了。

AppCache 的原理有两个关键点：manifest 属性和 manifest 文件。

HTML 在头中通过 manifest 属性引用 manifest 文件。manifest 文件，就是上面以 appcache

结尾的文件，是一个普通文件文件，列出了需要缓存的文件。




```
CACHE MANIFEST
demo_time.js
img_logo.gif
```

上面截图中的 manifest 文件，就 HTML 代码引用的 manifest 文件。文件比较简单，第一行是关键字，第二、三行就是要缓存的文件路径（相对路径）。这只是最简单的 manifest 文件，完整的还包括其他关键字与内容。引用 manifest 文件的 HTML 和 manifest 文件中列出的要缓存的文件最终都会被浏览器缓存。

完整的 manifest 文件，包括三个 Section，类型 Windows 中 ini 配置文件的 Section，不过不要中括号。

1. **CACHE MANIFEST** - Files listed under this header will be cached after they are downloaded for the first time
2. **NETWORK** - Files listed under this header require a connection to the server, and will never be cached
3. **FALLBACK** - Files listed under this header specifies fallback pages if a page is inaccessible

完整的 manifest 文件，如：

```
1    CACHE MANIFEST
2    # 2012-02-21 v1.0.0
3    /theme.css
4    /logo.gif
5    /main.js
6    NETWORK:
7    login.asp
8    FALLBACK:
9    /html/ /offline.html
```

总的来说，浏览器在首次加载 HTML 文件时，会解析 manifest 属性，并读取 manifest 文件，获取 Section：CACHE MANIFEST 下要缓存的文件列表，再对文件缓存。

AppCache 的缓存文件，与浏览器的缓存文件分开存储的，还是一份？应该是分开的。因为 AppCache 在本地也有 5MB（分 HOST）的空间限制。

AppCache 在首次加载生成后，也有更新机制。被缓存的文件如果要更新，需要更新 manifest 文件。因为浏览器在下次加载时，除了会默认使用缓存外，还会在后台检查 manifest 文件有没有修改（byte by byte)。发现有修改，就会重新获取 manifest 文件，对 Section：CACHE MANIFEST 下文件列表检查更新。manifest 文件与缓存文件的检查更新也遵守浏览器缓存机制。

如用用户手动清了 AppCache 缓存，下次加载时，浏览器会重新生成缓存，也可算是一种缓存的更新。另外， Web App 也可用代码实现缓存更新。

分析：AppCache 看起来是一种比较好的缓存方法，除了缓存静态资源文件外，也适合构建 Web 离线 App。在实际使用中有些需要注意的地方，有一些可以说是”坑“。

1. **要更新缓存的文件**，需要更新包含它的 manifest 文件，那怕只加一个空格。常用的方法，是**修改 manifest 文件注释中的版本号**。如：# 2012-02-21 v1.0.0
2. 被缓存的文件，浏览器是先使用，再通过检查 manifest 文件是否有更新来更新缓存文件。这样缓存文件可能用的不是最新的版本。
3. 在更新缓存过程中，如果有一个文件更新失败，则整个更新会失败。
4. manifest 和引用它的 HTML 要在相同 HOST。
5. manifest 文件中的文件列表，如果是相对路径，则是相对 manifest 文件的相对路径。
6. manifest 也有可能更新出错，导致缓存文件更新失败。
7. 没有缓存的资源在已经缓存的 HTML 中不能加载，即使有网络。例如：http://appcache-demo.s3-website-us-east-1.amazonaws.com/without-network/
8. manifest 文件本身不能被缓存，且 manifest 文件的更新使用的是浏览器缓存机制。所以 manifest 文件的 Cache-Control 缓存时间不能设置太长。

另外，根据官方文档，AppCache 已经不推荐使用了，标准也不会再支持。现在主流的浏览器都是还支持 AppCache 的，以后就不太确定了。

在 Android 内嵌 Webview 中，需要通过 Webview 设置接口启用 AppCache，同时还要设置缓存文件的存储路径，另外还可以设置缓存的空间大小。

```
1    WebView myWebView = (WebView) findViewById(R.id.webview);
2    WebSettings webSettings = myWebView.getSettings();
3    webSettings.setAppCacheEnabled(true);
4    final String cachePath = getApplicationContext().getDir("cache", Context.MODE_PRIVATE).getPath();
5    webSettings.setAppCachePath(cachePath);
6    webSettings.setAppCacheMaxSize(5*1024*1024);
```

2.5 Indexed Database

IndexedDB 也是一种数据库的存储机制，但不同于已经不再支持的 Web SQL Database。IndexedDB 不是传统的关系数据库，可归为 NoSQL 数据库。IndexedDB 又类似于 Dom Storage 的 key-value 的存储方式，但功能更强大，且存储空间更大。

IndexedDB 存储数据是 key-value 的形式。Key 是必需，且要唯一；Key 可以自己定义，也可由系统自动生成。Value 也是必需的，但 Value 非常灵活，可以是任何类型的对象。一般 Value 都是通过 Key 来存取的。

IndexedDB 提供了一组 API，可以进行数据存、取以及遍历。这些 API 都是异步的，操作的结果都是在回调中返回。

下面代码演示了 IndexedDB 中 DB 的打开（创建）、存储对象(可理解成有关系数据的"表")的创建及数据存取、遍历基本功能。

```
1        var db;
2        window.indexedDB = window.indexedDB || window.mozIndexedDB || window.webkitIndexedDB || window.m
3        //浏览器是否支持 IndexedDB
4        if (window.indexedDB) {
5          //打开数据库，如果没有，则创建
6          var openRequest = window.indexedDB.open("people_db", 1);
7          //DB 版本设置或升级时回调
8          openRequest.onupgradeneeded = function(e) {
9            console.log("Upgrading...");
10           var thisDB = e.target.result;
11           if(!thisDB.objectStoreNames.contains("people")) {
```

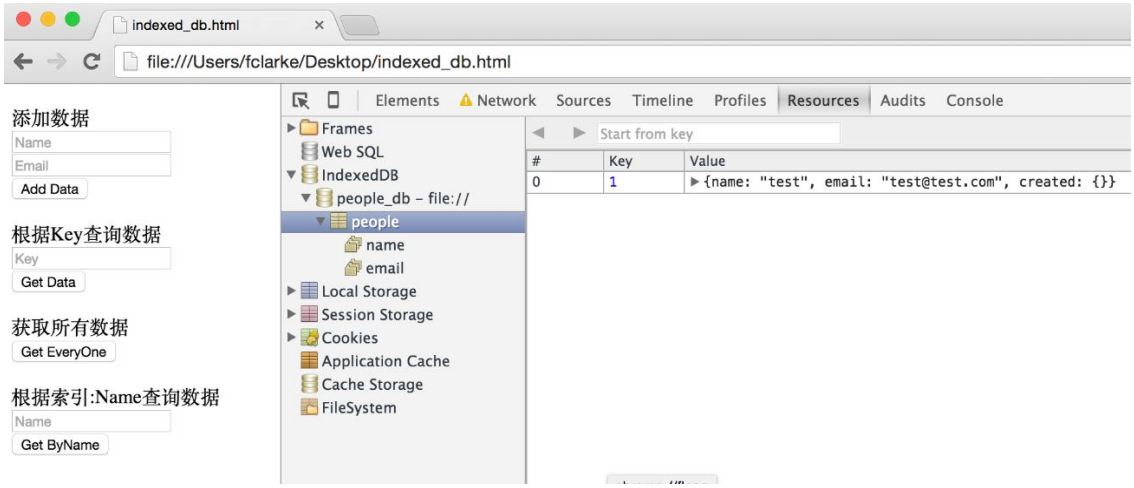
```
12              console.log("Create Object Store: people.");
13              //创建存储对象，类似于关系数据库的表
14              thisDB.createObjectStore("people", { autoIncrement:true });
15             //创建存储对象， 还创建索引
16              //var objectStore = thisDB.createObjectStore("people",{ autoIncrement:true });
17             // //first arg is name of index, second is the path (col);
18            //objectStore.createIndex("name","name", {unique:false});
19           //objectStore.createIndex("email","email", {unique:true});
20          }
21      }
22      //DB 成功打开回调
23      openRequest.onsuccess = function(e) {
24         console.log("Success!");
25         //保存全局的数据库对象，后面会用到
26         db = e.target.result;
27        //绑定按钮点击事件
28          document.querySelector("#addButton").addEventListener("click", addPerson, false);
29         document.querySelector("#getButton").addEventListener("click", getPerson, false);
30         document.querySelector("#getAllButton").addEventListener("click", getPeople, false);
31         document.querySelector("#getByName").addEventListener("click", getPeopleByNameIndex1, false);
32      }
33       //DB 打开失败回调
34       openRequest.onerror = function(e) {
35          console.log("Error");
36          console.dir(e);
37        }
38      }else{
39         alert('Sorry! Your browser doesn\'t support the IndexedDB.');
40      }
41      //添加一条记录
42      function addPerson(e) {
43         var name = document.querySelector("#name").value;
44         var email = document.querySelector("#email").value;
45         console.log("About to add "+name+"/"+email);
46         var transaction = db.transaction(["people"],"readwrite");
47      var store = transaction.objectStore("people");
48        //Define a person
49        var person = {
50           name:name,
51           email:email,
52           created:new Date()
53        }
54        //Perform the add
55        var request = store.add(person);
```

```
56          //var request = store.put(person, 2);
57          request.onerror = function(e) {
58             console.log("Error",e.target.error.name);
59             //some type of error handler
60          }
61          request.onsuccess = function(e) {
62            console.log("Woot! Did it.");
63          }
64        }
65        //通过 KEY 查询记录
66        function getPerson(e) {
67          var key = document.querySelector("#key").value;
68          if(key === "" || isNaN(key)) return;
69          var transaction = db.transaction(["people"],"readonly");
70          var store = transaction.objectStore("people");
71          var request = store.get(Number(key));
72          request.onsuccess = function(e) {
73             var result = e.target.result;
74             console.dir(result);
75             if(result) {
76               var s = "Key "+key+"";
77               for(var field in result) {
78                  s+= field+"="+result[field]+"";
79               }
80               document.querySelector("#status").innerHTML = s;
81             } else {
82                document.querySelector("#status").innerHTML = "No match!";
83             }
84           }
85        }
86        //获取所有记录
87        function getPeople(e) {
88          var s = "";
89           db.transaction(["people"], "readonly").objectStore("people").openCursor().onsuccess = function(e) {
90             var cursor = e.target.result;
91             if(cursor) {
92                s += "Key "+cursor.key+"";
93                for(var field in cursor.value) {
94                   s+= field+"="+cursor.value[field]+"";
95                }
96                s+="";
97                cursor.continue();
98             }
99              document.querySelector("#status2").innerHTML = s;
```

```
100         }
101     }
102     //通过索引查询记录
103     function getPeopleByNameIndex(e)
104     {
105       var name = document.querySelector("#name1").value;
106       var transaction = db.transaction(["people"],"readonly");
107       var store = transaction.objectStore("people");
108       var index = store.index("name");
109       //name is some value
110       var request = index.get(name);
111       request.onsuccess = function(e) {
112         var result = e.target.result;
113         if(result) {
114           var s = "Name "+name+"";
115           for(var field in result) {
116             s+= field+"="+result[field]+"";
117           }
118           s+="";
119       } else {
120           document.querySelector("#status3").innerHTML = "No match!";
121        }
122       }
123     }
124     //通过索引查询记录
125     function getPeopleByNameIndex1(e)
126     {
127       var s = "";
128       var name = document.querySelector("#name1").value;
129       var transaction = db.transaction(["people"],"readonly");
130       var store = transaction.objectStore("people");
131       var index = store.index("name");
132       //name is some value
133       index.openCursor().onsuccess = function(e) {
134         var cursor = e.target.result;
135         if(cursor) {
136           s += "Key "+cursor.key+"";
137           for(var field in cursor.value) {
138             s+= field+"="+cursor.value[field]+"";
139           }
140           s+="";
141           cursor.continue();
142         }
143          document.querySelector("#status3").innerHTML = s;
```

```
144         }
145     }添加数据 Add Data 根据 Key 查询数据 Get Data 获取所有数据 Get EveryOne 根据索引:Name 查询数据
```

将上面的代码复制到 indexed_db.html 中，用 Google Chrome 浏览器打开，就可以添加、查询数据。在 Chrome 的开发者工具中，能查看创建的 DB 、存储对象(可理解成表）以及表中添加的数据。



IndexedDB 有个非常强大的功能，就是 index（索引）。它可对 Value 对象中任何属性生成索引，然后可以基于索引进行 Value 对象的快速查询。

要生成索引或支持索引查询数据，需求在首次生成存储对象时，调用接口生成属性的索引。可以同时对对象的多个不同属性创建索引。如下面代码就对 name 和 email 两个属性都生成了索引。

```
1     var objectStore = thisDB.createObjectStore("people",{ autoIncrement:true });
2     //first arg is name of index, second is the path (col);
3     objectStore.createIndex("name","name", {unique:false});
4     objectStore.createIndex("email","email", {unique:true});
```

生成索引后，就可以基于索引进行数据的查询。

```
1       function getPeopleByNameIndex(e)
2       {
3       var name = document.querySelector("#name1").value;
4       var transaction = db.transaction(["people"],"readonly");
5       var store = transaction.objectStore("people");
6       var index = store.index("name");
7       //name is some value
8       var request = index.get(name);
9       request.onsuccess = function(e) {
10        var result = e.target.result;
11        if(result) {
12           var s = "";
13        } else {
14           document.querySelector("#status3").innerHTML = "";
15        }
16      }
17    }
```

分析：IndexedDB 是一种灵活且功能强大的数据存储机制，它集合了 Dom Storage 和 Web SQL Database 的优点，用于存储大块或复杂结构的数据，提供更大的存储空间，使用起来也比较简单。可以作为 Web SQL Database 的替代。不太适合静态文件的缓存。

1. 以 key-value 的方式存取对象，可以是任何类型值或对象，包括二进制。
2. 可以对对象任何属性生成索引，方便查询。
3. 较大的存储空间，默认推荐 250MB(分 HOST)，比 Dom Storage 的 5MB 要大的多。
4. 通过数据库的事务（tranction）机制进行数据操作，保证数据一致性。
5. 异步的 API 调用，避免造成等待而影响体验。

Android 在 4.4 开始加入对 IndexedDB 的支持，只需打开允许 JS 执行的开关就好了。

```
1    WebView myWebView = (WebView) findViewById(R.id.webview);
2    WebSettings webSettings = myWebView.getSettings();
3    webSettings.setJavaScriptEnabled(true);
```

2.6 File System API

File System API 是 H5 新加入的存储机制。它为 Web App 提供了一个虚拟的文件系统，就像 Native App 访问本地文件系统一样。由于安全性的考虑，这个虚拟文件系统有一定的限制。Web App 在虚拟的文件系统中，可以进行文件（夹）的创建、读、写、删除、遍历等操作。

File System API 也是一种可选的缓存机制，和前面的 SQLDatabase、IndexedDB 和 AppCache 等一样。File System API 有自己的一些特定的优势：

1. 可以满足大块的二进制数据（ large binary blobs）存储需求。
2. 可以通过预加载资源文件来提高性能。
3. 可以直接编辑文件。

浏览器给虚拟文件系统提供了两种类型的存储空间：临时的和持久性的。临时的存储空间是由浏览器自动分配的，但可能被浏览器回收；持久性的存储空间需要显示的申请，申请时浏览器会给用户一提示，需要用户进行确认。持久性的存储空间是 WebApp 自己管理，浏览器不会回收，也不会清除内容。持久性的存储空间大小是通过配额来管理的，首次申请时会一个初始的配额，配额用完需要再次申请。

虚拟的文件系统是运行在沙盒中。不同 WebApp 的虚拟文件系统是互相隔离的，虚拟文件系统与本地文件系统也是互相隔离的。

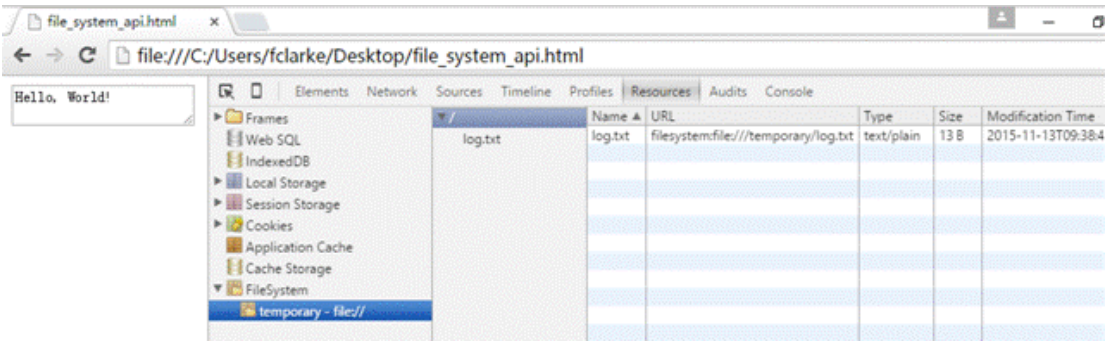
File System API 提供了一组文件与文件夹的操作接口，有同步和异步两个版本，可满足不同的使用场景。下面通过一个文件创建、读、写的例子，演示下简单的功能与用法。

```
1    window.requestFileSystem = window.requestFileSystem || window.webkitRequestFileSystem;
2    //请求临时文件的存储空间
3    if (window.requestFileSystem) {
4        window.requestFileSystem(window.TEMPORARY, 5*1024*1024, initFS, errorHandler);
5    }else{
6      alert('Sorry! Your browser doesn\'t support the FileSystem API');
7    }
8    //请求成功回调
9    function initFS(fs){
10     //在根目录下打开 log.txt 文件，如果不存在就创建
11     //fs 就是成功返回的文件系统对象，fs.root 代表根目录
12     fs.root.getFile('log.txt', {create: true}, function(fileEntry) {
13     //fileEntry 是返回的一个文件对象，代表打开的文件
```

```
14      //向文件写入指定内容
15      writeFile(fileEntry);
16      //将写入的内容又读出来，显示在页面上
17      readFile(fileEntry);
18      }, errorHandler);
19    }
20    //读取文件内容
21    function readFile(fileEntry)
22    {
23       console.log('readFile');
24      // Get a File object representing the file,
25      // then use FileReader to read its contents.
26      fileEntry.file(function(file) {
27        console.log('createReader');
28        var reader = new FileReader();
29        reader.onloadend = function(e) {
30          console.log('onloadend');
31          var txtArea = document.createElement('textarea');
32          txtArea.value = this.result;
33          document.body.appendChild(txtArea);
34        };
35        reader.readAsText(file);
36      }, errorHandler);
37    }
38    //向文件写入指定内容
39    function writeFile(fileEntry)
40    {
41       console.log('writeFile');
42       // Create a FileWriter object for our FileEntry (log.txt).
43       fileEntry.createWriter(function(fileWriter) {
44         console.log('createWriter');
45         fileWriter.onwriteend = function(e) {
46           console.log('Write completed');
47         };
48           fileWriter.onerror = function(e) {
49             console.log('Write failed: ' + e.toString());
50           };
51           // Create a new Blob and write it to log.txt.
52           var blob = new Blob(['Hello, World!'], {type: 'text/plain'});
53           fileWriter.write(blob);
54        }, errorHandler);
55    }
56    function errorHandler(err){
57     var msg = 'An error occured: ' + err;
```

```
58      console.log(msg);
59      };
```

将上面代码复制到 file_system_api.html 文件中，用 Google Chrome 浏览器打开（现在 File System API 只有 Chrome 43+、Opera 32+ 以及 Chrome for Android 46+ 这三个浏览器支持）。由于 Google Chrome 禁用了本地 HTML 文件中的 File System API 功能，在启动 Chrome 时，要加上”—allow-file-access-from-files“命令行参数。



上面截图，左边是 HTML 运行的结果，右边是 Chrome 开发者工具中看到的 Web 的文件系统。基本上 H5 的几种缓存机制的数据都能在这个开发者工具看到，非常方便。

分析：File System API 给 Web App 带来了文件系统的功能，Native 文件系统的功能在 Web App 中都有相应的实现。任何需要通过文件来管理数据，或通过文件系统进行数据管理的场景都比较适合。

到目前，Android 系统的 Webview 还不支持 File System API。

3 移动端 Web 加载性能（缓存）优化

分析完 H5 提供的各种缓存机制，回到移动端（针对 Android，可能也适用于 iOS）的场景。现在 Android App（包括手 Q 和 WX）大多嵌入了 Webview 的组件（系统 Webview 或 QQ 游览器的 X5 组件），通过内嵌 Webview 来加载一些 H5 的运营活动页面或资讯页。这样可充分发挥 Web 前端的优势：快速开发、发布，灵活上下线。但 Webview 也有一些不可忽视的问题，比较突出的就是加载相对较慢，会相对消耗较多流量。

通过对一些 H5 页面进行调试及抓包发现，每次加载一个 H5 页面，都会有较多的请求。除了 HTML 主 URL 自身的请求外，HTML 外部引用的 JS、CSS、字体文件、图片都是一个独立的 HTTP 请求，每一个请求都串行的（可能有连接复用）。这么多请求串起来，再加上浏览器解析、渲染的时间，Web 整体的加载时间变得较长；请求文件越多，消耗的流量也会越多。我们可综合使用上面说到几种缓存机制，来帮助我们优化 Web 的加载性能。

缓存机制	优势	适用场景
浏览器缓存机制	HTTP协议层支持	静态文件的缓存
Dom Storage	较大存储空间，使用简单	临时、简单数据的缓存， Cookies的扩展
Web SQL Database	存储、管理复杂结构数据	用IndexedDB替代， 不推荐使用
AppCache	方便构建离线App	离线App、静态文件缓存， 不推荐使用
IndexedDB	存储任何类型数据、使用简单， 支持索引	结构、关系复杂的数据存储 Web SQL Database的替代
File System API	支持文件系统的操作	数据适合以文件进行管理的场景， Android系统还不支持

结论：综合各种缓存机制比较，对于静态文件，如 JS、CSS、字体、图片等，适合通过浏览器缓存机制来进行缓存，通过缓存文件可大幅提升 Web 的加载速度，且节省流量。但也有一些不足：缓存文件需要首次加载后才会产生；浏览器缓存的存储空间有限，缓存有被清除的可能；缓存的文件没有校验。要解决这些不足，可以参考手 Q 的离线包，它有效的解决了这些不足。

转至：http://www.cocoachina.com/webapp/20151217/14718.html