Manifest V3 下 `chrome.scripting.executeScript({ world: "MAIN" })` 的边界与限制深度研究报告

- 生成时间：2026-02-13
- 研究目标：明确 `world: "MAIN"` 在 MV3 下到底能做什么/不能做什么，特别关注：参数与返回值的“复杂对象”限制、异步返回值、CSP 影响、已知限制与替代方案。

Executive Summary

在 Chrome MV3 中，`chrome.scripting.executeScript()` 支持通过 `world` 选择脚本执行世界：默认 `ISOLATED`（扩展私有世界），也可以选择 `MAIN`（与页面脚本共享的主世界）。选择 `MAIN` 的核心收益是：注入代码可直接访问页面主世界暴露在 `window` / `globalThis` 上的变量与函数，从而绕过 content script 默认隔离导致的“能操作 DOM 但访问不到页面 JS 上下文”的问题。[1][2]

但 `MAIN` world 也带来关键限制：注入脚本会受到**页面 CSP** 约束（官方明确指出“注入到 main world 时，页面的 CSP 生效”），并且会暴露在页面脚本可干预的执行环境中；同时，`executeScript` 的参数与返回值在 Chrome 中存在“更偏 JSON-serializable”的传输限制，复杂对象（DOM 节点、函数、循环引用、BigInt 等）无法直接跨边界返回，通常需要手动序列化为 JSON 或通过 message channel 传递。[1][3]

Key Findings

- **异步返回值是支持的**：Chrome 文档明确：若注入脚本“求值结果是一个 Promise”，浏览器会等待其 settle，并把最终值作为 `InjectionResult.result` 返回。[3]
- **参数/返回值的可传输类型在 Chrome 更严格**：MDN 指出 Chrome 的注入结果要求“JSON-serializable value”，而不是更通用的 structured clone（Firefox）。这会直接影响你能否“返回复杂对象”。[4]
- **页面 CSP 对 MAIN world 生效**：Chrome content scripts 文档明确：当 content script 注入到 main world 时，页面的 CSP 生效；这意味着严格 CSP 的页面可能会禁止 `eval/new Function`、限制 `script-src` 等，从而影响你在 MAIN world 里想做的某些动态执行/加载行为。[2]
- **`MAIN` world 的安全边界更弱**：manifest 的 `content_scripts[].world` 文档给出警告：选择 `MAIN` 会与页面共享执行环境，页面可访问并干扰注入脚本，因此不应把敏感逻辑或秘密放在 MAIN world。[1]
- **有替代世界可绕过页面 CSP（但不是 MAIN）**：`chrome.userScripts` 提供 `USER_SCRIPT` world，文档明确该 world “exempt from the page's CSP”，用于用户脚本场景；但它不是页面主世界，因此通常不能直接读取页面 JS 全局变量——更适合“需要绕过 CSP 但不需要共享主世界”的脚本执行。[5]

Detailed Analysis

1) `world` 的语义：为什么 `MAIN` 能拿到页面 `window` 变量

Chrome `chrome.scripting` 把执行世界分为：

- `ISOLATED`：扩展独享；页面与扩展变量/函数互不可见；
- `MAIN`：DOM 的主世界，与页面 JavaScript 共享执行环境。[3]

因此当你用：

```
chrome.scripting.executeScript({
  target: { tabId },
  world: 'MAIN',
  func: () => window.somePageFunction('hello')
});
```

只要页面确实把 `somePageFunction` 暴露在 `window` 上，就能直接调用。

但注意：**MAIN world 也拿不到“非全局”变量**。页面里用模块作用域、闭包私有变量、或 `let/const` 顶层但不成为 `window` 属性的变量，你仍然无法从外部直接访问（这不是扩展限制，而是 JS 作用域规则）。这一点在调试时很容易误判为“MAIN 也不行”。

2) 复杂对象能不能传？——分三类看

2.1 注入参数：`args`

Chrome 文档给出 `args: any[]`，但没有在同一页面显式说明可传输类型边界。[3]
实践与跨浏览器文档（MDN）提示：为了稳定，`args` 应当限制在 JSON 可序列化集合（字符串/数字/布尔/null、数组、纯对象，无循环引用）。[4]

建议工程准则（稳定优先）：

- ✅ 传：`string | number | boolean | null | object | array`（无循环引用）
- ❌ 不传：`function`、DOM Node、`Window`、`Map/Set`（除非先转数组/对象）、含循环引用对象、`BigInt`（先转 string）

2.2 注入返回值：`InjectionResult.result`

MDN 明确指出：Chrome 的注入结果需要 JSON-serializable value。[4]
这解释了很多“返回对象卡住/回调不触发/结果为空”的现象：当返回值不可序列化时，Chrome 侧可能直接让注入失败或 promise reject（不同版本/场景表现可能不一致）。

最佳实践：

- 优先返回 **JSON 字符串**（`return JSON.stringify(obj)`），扩展侧再 parse。
- 或者只返回 primitive，复杂数据用 message 发送（见下节）。

2.3 错误返回：`InjectionResult.error` 的缺口

MDN 提到 Chrome 尚未支持 `InjectionResult.error`（存在相关 issue），因此不要依赖 `result/error` 二选一来判断成功失败；更稳妥的做法是注入代码自己 try/catch 并返回 `{ ok:false, message }` 这种结构化结果。[4]

3) 异步返回：能不能 `await`？

可以。
Chrome 文档明确：如果脚本求值结果是 Promise，浏览器会等待 promise settle 并返回结果。[3]

这意味着你可以：

- 注入 `async () => { ...; return value }`
- 或者返回 `fetch(...).then(r => r.json())` 的 promise

需要记住：最终返回值仍需满足上一节的“可序列化”约束。[4]

4) CSP：严格页面会不会阻断注入？

这里要区分两件事：

1. **注入是否发生**（`executeScript` 能否跑起来）
2. **注入代码在 MAIN world 里能否做你想做的事**（是否触发 CSP 违规）

Chrome 官方在 content scripts 文档中明确：当脚本注入到 main world 时，**页面 CSP 生效**。[2]

因此：

- `executeScript` 把函数注入到 MAIN world 这件事通常能发生（前提是你有 host 权限且目标页面允许注入），但
- 你在 MAIN world 里如果尝试 `eval()`、动态构造脚本、加载 CSP 不允许的 `script-src` 资源等，会被页面 CSP 拦住。

这也是为什么“用 MAIN world 直接塞一段大 polyfill/运行时”在某些严格 CSP 站点上会更脆弱：不是因为 MAIN world 不能跑，而是因为你做的动作触发了页面 CSP。

5) 权限与可注入页面范围（别把 world 当万能钥匙）

`chrome.scripting` 依赖：

- `"permissions": ["scripting"]`，以及目标 URL 的 host 权限（`host_permissions` 或 `activeTab`）。[3]

此外，match patterns 对 scheme 有硬限制（常规只覆盖 `http/https/file/*` 这类），所以像 `chrome://`、`about:` 等 scheme 并不是你能用 `<all_urls>` 覆盖的范围。[6]

6) 替代方案：`chrome.userScripts` 的 `USER_SCRIPT` world（绕 CSP）

`chrome.userScripts` 文档定义了 `ExecutionWorld`：

- `MAIN`：与页面共享主世界

- `USER_SCRIPT`：用户脚本世界，并且“exempt from the page's CSP”。[5]

如果你的核心问题是“页面 CSP 太严导致某些动态执行/加载不行”，而不是“必须访问页面 `window` 变量”，那么 `USER_SCRIPT` world 更可能是正确工具。

但若你的核心问题是“必须访问页面主世界变量/函数”，那 `MAIN` world 仍是正解，只是要把注入代码压到最小、避免触发 CSP。

Areas of Consensus

- `world: "MAIN"` 的价值就是进入页面主 JS 上下文；代价是失去隔离并受页面 CSP 影响。[1][2][3]
- `executeScript` 支持 promise/async 返回（等待 settle），但返回值在 Chrome 侧更偏向 JSON-serializable 类型。[3][4]

Areas of Debate

- **Chrome 对不可序列化返回值的失败表现**：MDN 指出 Chrome 的限制，但具体“失败是 reject 还是 silent failure/console error”可能随版本变化；建议用一组用例在你们目标 Chrome 版本上做基准测试并固化为适配层规范。[4]

Sources

[1] Chrome for Developers — *Manifest: content scripts*（`world: ISOLATED|MAIN` + MAIN 风险警告）
https://developer.chrome.com/docs/extensions/reference/manifest/content-scripts （权威：官方）

[2] Chrome for Developers — *Content scripts*（isolated worlds + “注入到 main world 时页面 CSP 生效”）
https://developer.chrome.com/docs/extensions/develop/concepts/content-scripts （权威：官方）

[3] Chrome for Developers — *chrome.scripting API reference*（`ExecutionWorld`、`executeScript()` promise settle 语义、方法签名） https://developer.chrome.com/docs/extensions/reference/api/scripting （权威：官方）

[4] MDN — *scripting.executeScript()*（Chrome 返回值 JSON-serializable 限制、Chrome 不支持 InjectionResult.error 的兼容性说明） https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/scripting/executeScript （较权威：MDN，跨浏览器对照，需结合官方/实测）

[5] Chrome for Developers — *chrome.userScripts API reference*（`USER_SCRIPT` world “exempt from the page's CSP” + world 语义） https://developer.chrome.com/docs/extensions/reference/api/userScripts （权威：官方）

[6] Chrome for Developers — *Match patterns*（scheme 限制；`<all_urls>` 仅覆盖允许 scheme）
https://developer.chrome.com/docs/extensions/develop/concepts/match-patterns （权威：官方）

Gaps and Further Research

- **“复杂对象”边界的实测矩阵**：建议按你们目标 Chrome 版本跑一套注入用例：返回 `Date/RegExp/Map/Set/TypedArray/BigInt/Error/Promise/DOM Node/循环引用对象`，记录每类的行为（resolve/reject/空值/console 报错），并把结论固化到 TS SDK 的序列化层。[3][4]
- **CSP 影响的实测**：选取几种典型 CSP（无 `unsafe-eval`、无外链、Trusted Types 强制等），验证 MAIN world 注入时可行的“桥接最小代码”模板，作为后续 WebMCP polyfill 注入策略依据。[2]