Chrome 扩展的 `world` / `ExecutionWorld` 机制深度研究报告（ISOLATED vs MAIN）

- 生成时间：2026-02-13
- 研究范围：Chrome Extensions（MV3）中的“脚本执行世界（world）”概念；重点覆盖 Content Script 默认 `ISOLATED` world、以及通过 `chrome.scripting.executeScript({ world: "MAIN" })` 进入页面主 JS 上下文的能力。

Executive Summary

Chrome 扩展把“页面脚本”和“扩展脚本（内容脚本/注入脚本）”划分到不同的 JavaScript 执行世界（world）中：默认 Content Script 运行在 `ISOLATED` world，可以读写 DOM 但无法直接访问页面脚本创建的 JS 变量/函数，从而避免命名冲突并降低被页面脚本干扰的风险。[1][2]

当你确实需要访问页面主世界的 `window` 变量、调用页面暴露的函数（例如你举例的 `window.somePageFunction()`），可以在注入时显式选择 `world: "MAIN"`，让注入代码与页面脚本共享同一个执行环境；但代价是页面脚本也能“看到并影响”这段代码，安全边界更弱，需要更谨慎的隔离与通信设计。[1][3]

Key Findings

- **默认隔离是设计目标**：Chrome 明确指出 Content Scripts 与页面分别处于不同隔离世界，彼此不能访问对方创建的变量/函数；它们共享 DOM，因此能操作页面元素，但不共享 JS 全局环境。[2]
- **`world` 在多个入口存在**：
  - `manifest.json` 的 `content_scripts[].world` 支持 `ISOLATED` / `MAIN`，默认 `ISOLATED`；选择 `MAIN` 会与页面脚本共享执行环境，并有被页面干扰的风险。[1]
  - `chrome.scripting.executeScript()` 的 `ScriptInjection.world` 同样支持 `ISOLATED` / `MAIN`，默认 `ISOLATED`。[3]
- **`MAIN` world ≠ 扩展权限**：即使把脚本放到 `MAIN` world，它在语义上仍是“页面脚本环境”，并不会自动获得扩展上下文能力（例如在该注入函数里直接调用 `chrome.runtime.*` 并不等价于内容脚本/扩展页环境）。因此常见做法是：`MAIN` world 仅做“读页面状态/调用页面函数/打点”，再把结果回传到扩展侧处理。[2][3]
- **注入函数会被序列化**：`executeScript({ func })` 的函数会被序列化/反序列化，绑定参数与闭包上下文会丢失；需要通过 `args` 传入参数，并确保参数可 JSON 序列化。[3]

- **“搞定”的前提是目标在主世界可见**：`world: "MAIN"` 能让代码处在页面主执行环境，但你能否直接拿到目标对象取决于页面是否把它暴露为可访问的全局（例如挂到 `window` / `globalThis`），很多现代站点的顶层 `let/const`、模块作用域、闭包内部变量并不会成为 `window` 属性——这类情况即使 `MAIN` world 也拿不到。[2]

Detailed Analysis

1) 什么是 `world`：`ISOLATED` vs `MAIN`

Chrome 把“内容脚本”和“页面脚本”的 JavaScript 环境隔离开：

- `ISOLATED` world：扩展独享的 JS 执行环境；页面脚本无法直接访问内容脚本创建的变量/函数，反之亦然。[2][3]
- `MAIN` world：页面主世界；脚本与页面 JS 共享执行环境，因此可以直接访问页面脚本暴露的对象、变量与函数（例如 `window.somePageFunction`）。[1][3]

关键点是：**隔离的是“JS 全局与作用域”，不是 DOM**。因此在 `ISOLATED` world 中依然可以通过 DOM API 读写页面元素、监听事件、修改样式等。[2]

2) `world` 在哪里配置：静态、动态、一次性注入

2.1 静态（manifest）Content Scripts：`content_scripts[].world`

在 `manifest.json` 的 `content_scripts` 配置里可以指定 `world`，取值 `ISOLATED` 或 `MAIN`，默认 `ISOLATED`。[1]

当你把静态 content script 配到 `MAIN` 时，它将“长期”处于页面主世界，适合需要持续与页面 JS 深度交互的场景，但安全风险也随之上升（页面可以更容易地干扰你的代码）。[1]

2.2 动态注册：`chrome.scripting.registerContentScripts()` 的 `world`

通过 `chrome.scripting` 动态注册内容脚本时，同样可以指定 `world`，默认 `ISOLATED`；该字段在 `RegisteredContentScript` 类型上明确标注为 `Chrome 102+`。[3]

这让你可以“按需在运行时”决定某些站点/场景用 `MAIN`，其它默认继续 `ISOLATED`，把风险面压到最小。

2.3 一次性注入：`chrome.scripting.executeScript({ world })`

`chrome.scripting.executeScript()` 的注入参数 `ScriptInjection` 支持 `world`（`Chrome 95+`），默认 `ISOLATED`。[3]

这正对应你给的思路：

- 绝大多数逻辑仍在扩展侧（SW、扩展页、ISOLATED content script）完成；
- 仅在需要“触达页面 JS 全局”的那一瞬间，用 `executeScript({ world: "MAIN" })` 注入最小的桥接函数读取/调用，然后把结果带回扩展侧。

3) 为什么 `ISOLATED` world 访问不到页面的 `window` 变量/函数

官方对隔离世界的定义强调：页面、内容脚本、以及其它扩展之间的 JS 上下文是相互隔离的；彼此创建的变量/函数不可见。[2]

这解释了你描述的现象：

- `ISOLATED` content script 可以 `document.querySelector(...)` 操作 DOM；
- 但如果页面脚本有 `window.somePageFunction = ...` 或 `var x = ...`（主世界变量），内容脚本不能直接读写/调用它们。

4) 使用 `world: "MAIN"` 的真实能力边界（你这段代码“为什么能行”）

在 `chrome.scripting` 的类型定义中，`ExecutionWorld.MAIN` 被定义为“与 host page JavaScript 共享的主世界”。[3]

因此当你这样注入：

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

注入函数是在页面主世界执行的——如果页面确实把 `somePageFunction` 暴露在 `window` 上，你就能直接调用并把返回值带回扩展侧（以每个 frame 一个 `InjectionResult` 的形式返回）。[3]

但要注意两个“经常误判的前提条件”：

1. **页面目标必须在主世界可见**：很多页面并不会把关键函数放到 `window`，而是包在模块或闭包里；`MAIN` world 也无法越过语言作用域规则去拿到这些隐藏状态。[2]
2. **返回值/参数需要可序列化**：注入函数的 `args` 必须 JSON 可序列化；注入结果会被带回扩展侧，实务上应返回简单可序列化的值（字符串/数字/对象/数组），避免返回 DOM、函数等复杂对象。[3]

[4]

5) `MAIN` world 的主要风险与最佳实践

manifest 文档对 `MAIN` world 给出了明确警告：使用 `MAIN` 时，host page 可以访问并干扰注入脚本。[1]

从工程角度，常见风险包括：

- **被页面脚本篡改运行时环境**：页面可能修改全局变量、原型链（例如 `Array.prototype`）、或关键 API 的行为，导致你的逻辑被劫持或出现兼容性问题。
- **权限边界误用**：把敏感逻辑放在与页面共享的世界，会增加数据泄露/被操纵的可能性。

官方安全建议中也强调：内容脚本与页面同进程运行，可能遭遇 DOM 操纵、侧信道等风险；敏感操作应在扩展的 service worker 等更受控的环境执行，并对来自 content script 的输入保持不信任（校验/净化）。[5]

实战建议（在不牺牲功能的前提下“最小化 MAIN”）：

- **默认用 `ISOLATED`**；只有“必须读页面 JS 状态/调用页面函数”那一小段，用 `executeScript({ world: 'MAIN' })` 注入最小桥接函数。[1][3]
- **桥接函数只做读取/调用，不做敏感处理**：拿到结果后立刻回传到扩展侧（ISOLATED 或 SW）做业务处理。
- **跨世界通信要有协议与校验**：若需要持续通信，优先用 `window.postMessage()` 这种基于 DOM 的通信方式，并严格校验 `event.source` / `event.origin` / `type` 字段，避免被页面伪造消息。[2]

6) 推荐的“桥接”模式：MAIN 世界读取 + ISOLATED 世界处理

官方“隔离世界通信”章节给出了一个常见实现：通过 `window.postMessage()` 在页面与内容脚本之间传递结构化消息。[2]

结合 `world`，你可以把体系拆成三段：

1. `MAIN` 注入脚本：读取 `window` 上的对象/调用页面函数，把结果 `postMessage` 出来。
2. `ISOLATED` content script：监听 message，做校验，然后 `chrome.runtime.sendMessage()` 把结果交给 SW。
3. Service Worker：执行需要权限/存储/网络的逻辑。

这样既能“触达页面 JS”，又能把扩展的权限边界尽量留在扩展侧。

Areas of Consensus

- 默认 Content Script 运行在 `ISOLATED` world，与页面 JS 上下文隔离，但共享 DOM；要与页面 JS 通信，需通过共享 DOM（如 `postMessage`）或切换到 `MAIN` world 执行。[2]
- `world` 作为执行世界选择器，在 `content_scripts`（静态注入）与 `chrome.scripting.executeScript/registerContentScripts`（动态/一次性注入）中都存在，且默认值为 `ISOLATED`。[1][3]
- `MAIN` world 能解决“访问页面 `window` 变量/函数”的核心诉求，但同时引入页面可干扰注入脚本的风险，需要最小化使用并遵循安全建议。[1][5]

Areas of Debate

- **是否应该把长期 content script 直接放到 `MAIN`**：对需要深度集成的扩展来说它很诱人，但安全与兼容性成本更高；更稳妥的折中是“绝大部分逻辑留在 ISOLATED，只把必要桥接放在 MAIN”。[1][5]
- **返回值序列化规则的边界**：Chrome 官方 `chrome.scripting` 文档明确 `args` 需 JSON 可序列化，但对 `InjectionResult.result` 的可传输类型限制描述相对简略；跨浏览器文档指出 Chrome 对结果类型的限制可能更严格（更偏向 JSON 可序列化），实务上建议只返回简单值，并在复杂场景用消息通道传输。[3][4]

Sources

[1] Chrome for Developers — “Manifest - content scripts”（官方文档，描述 `content_scripts[].world` 的 `ISOLATED/MAIN`、默认值与风险警告，Last updated 2023-08-10 UTC）https://developer.chrome.com/docs/extensions/reference/manifest/content-scripts （权威：官方）

[2] Chrome for Developers — “Content scripts（Work in isolated worlds / Communication with the embedding page）”（官方概念文档，解释隔离世界、变量不可见、DOM 共享与 `window.postMessage` 通信示例） https://developer.chrome.com/docs/extensions/develop/concepts/content-scripts （权威：官方）

[3] Chrome for Developers — “chrome.scripting API reference”（官方 API 文档，定义 `ExecutionWorld`、`ScriptInjection.world`、`RegisteredContentScript.world`，以及 `func` 序列化、`args` JSON-serializable、注入结果返回方式等）https://developer.chrome.com/docs/extensions/reference/api/scripting （权威：官方）

[4] MDN — “scripting.executeScript()”（跨浏览器参考，补充 `world` 参数与注入结果类型/兼容性差异说明；用于理解 Chrome 可能更偏向 JSON 可序列化的返回值约束） https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/scripting/executeScript （较权威：MDN，需结合官方/实测）

[5] Chrome for Developers — “Stay secure（Use content scripts carefully）”（官方安全建议，强调内容脚本与页面交互的风险、敏感操作应放在更受控上下文、对消息输入做校验） https://developer.chrome.com/docs/extensions/develop/security-privacy/stay-secure （权威：官方）

Gaps and Further Research

- **MAIN world 的工程化“防干扰”实践**：例如如何在页面可能污染原型链的情况下实现更强健的桥接（冻结关键引用、最小依赖全局、严控输入输出）。需要结合具体目标站点做对抗性测试与基准用例沉淀。[1][5]
- **返回值类型的严格边界与报错形态**：建议用一组系统化用例（返回 DOM/BigInt/循环引用/TypedArray/Promise 等）在目标 Chrome 版本上验证 `InjectionResult.result` 的可传输类型与失败表现，以形成团队规范。[3][4]