`navigator.modelContext` polyfill 的可行性研究报告（注入方式、可写性、与未来原生实现的冲突风险）

- 生成时间：2026-02-13
- 研究目标：评估“在不支持 WebMCP 的页面上注入一个假的 `navigator.modelContext`”的技术可行性与风险：
  - `navigator` / `Navigator` 的可写性与 `defineProperty` 可行路径
  - 必须在哪个 world 注入才对页面 JS 生效
  - 与未来 Chrome 原生实现（规范属性）是否会冲突

Executive Summary

从规范层面看，WebMCP（Draft CG Report, 2026-02-12）明确把 `modelContext` 定义为 `Navigator` 上的只读属性，并要求 `[SecureContext, SameObject]` 语义。[1] 这意味着 polyfill 的目标形态很清晰：在页面运行时提供一个稳定的 `ModelContext` 对象，并在安全上下文中暴露为 `navigator.modelContext`。

从工程实践层面看，已经存在公开的 polyfill 实现（例如 `@mcp-b/global`）通过 `Object.defineProperty(window.navigator, 'modelContext', { value, writable:false, configurable:false })` 来安装 `navigator.modelContext`，并且会在检测到原生实现时跳过安装；这说明“在现代浏览器里把属性加到 `navigator` 上”总体是可行的。[2]

关键限制在于执行世界：内容脚本默认运行在 `ISOLATED` world，与页面 JS 上下文隔离；如果你在 `ISOLATED` world 里做 `navigator.modelContext = ...`，页面脚本通常看不到。要让页面 JS 直接使用 polyfill，必须把安装逻辑注入到 `MAIN` world（例如 `chrome.scripting.executeScript({ world: 'MAIN' })` 或 `content_scripts[].world = 'MAIN'`）。[3][4]

Key Findings

- **标准目标形态**：规范要求 `Navigator.modelContext` 是 `readonly attribute ModelContext modelContext` 且 `[SecureContext, SameObject]`；polyfill 需要尽量模拟这些语义（至少：稳定引用 + 仅在安全上下文启用）。[1]
- **实现可行性已有佐证**：`@mcp-b/global` polyfill 直接对 `window.navigator` 做 `Object.defineProperty(..., 'modelContext', ...)`，并在检测到原生 API 时不覆盖；这证明“给 navigator 增加属性”在实际 Chrome/现代浏览器上可用。[2]
- **world 决定可见性**：Chrome 官方说明 content scripts 与页面脚本处于不同 isolated worlds，变量/函数不可互访；要让页面脚本访问到你注入的 `navigator.modelContext`，需要在 `MAIN` world 安装 polyfill。[3][4]
- **未来原生实现的冲突可控**：只要遵循 feature detection（`if ('modelContext' in navigator) return;`）并避免覆盖已有实现，polyfill 与未来原生实现通常不会在“下一次加载”发生冲突；冲突风险主要来自“在同一页面生命周期中动态切换/启用原生实现”这类非常规场景。[1][2]

Detailed Analysis

1) 规范视角：polyfill 应该模拟什么

WebMCP 规范把 `Navigator` 扩展为：

- `[SecureContext, SameObject] readonly attribute ModelContext modelContext;`。[1]

对应到 polyfill 的“最低模拟目标”：

1. **SecureContext**：在 `!isSecureContext` 时不暴露/直接报错/返回 undefined（取决于你们产品策略）；
2. **SameObject**：`navigator.modelContext` 每次读到同一个对象引用（不要 getter 每次 new）；

3. **最小方法集合**：`provideContext/clearContext/registerTool/unregisterTool`（如果你要与规范对齐）；
4. **工具回调签名**：`execute(input, client)` 返回 Promise，`client.requestUserInteraction()` 作为用户交互钩子（如果你实现 client）。[1]

2) 页面上“能不能改 navigator”：两条常见技术路径

路径 A：在 `window.navigator` 实例上直接 defineProperty（最直接）

已有 polyfill（`@mcp-b/global`）采用：

- `Object.defineProperty(window.navigator, 'modelContext', { value: bridge.modelContext, writable: false, configurable: false })`。[2]

优点：

- 实现简单，读属性就是固定对象（天然 SameObject）。

风险与注意：

- 若某些环境把 `navigator` 设为不可扩展（extensible=false），直接 defineProperty 可能抛错；需要 try/catch 兜底。
- `configurable:false` 会让你在当前页面生命周期内无法卸载/重定义该属性；如果你们有“热切换原生/ polyfill”的需求，建议设为 configurable:true 并把卸载逻辑纳入生命周期管理（这属于工程策略，不是标准要求）。

路径 B：在 `Navigator.prototype` 上挂 getter（更“像规范”）

规范本身是 Web IDL attribute，往往体现在原型链上的访问器属性；理论上 polyfill 也可以选择把 getter 安装在 `Navigator.prototype`。

优点：

- 更接近“浏览器原生 attribute”的形态；
- 如果 navigator 实例不可扩展，仍可能通过原型链提供属性。

风险：

- 需要确保不会影响到其它 realm/iframe（以及与原生实现的覆盖关系），实现复杂度更高。

> 结论：如果你只需要“在现代 Chrome 的普通网页里让页面 JS 能访问到 modelContext”，路径 A 足够；如果要覆盖更多极端环境/更强兼容性，再考虑路径 B。

3) 最关键的工程前提：必须在 `MAIN` world 安装

Chrome 官方解释：content scripts 默认在 isolated world，页面脚本与内容脚本之间变量/函数互不可见。[3]
同时 `chrome.scripting` 定义了 `ExecutionWorld`：

- `ISOLATED`：扩展私有
- `MAIN`：与页面共享主世界。[4]

因此：

- 在 `ISOLATED` world 里“安装 polyfill”通常只对扩展侧可见，对页面脚本不可见；
- 要让网站自己的 JS 直接调用 `navigator.modelContext.registerTool(...)`，必须在 `MAIN` world 执行安装逻辑。

4) 与未来原生实现的冲突：怎么判断“会不会冲突”

4.1 常规场景（页面刷新/重启浏览器）

原生实现若未来出现在某个 Chrome 版本里，通常会在 JS 执行前就把 `navigator.modelContext` 挂好。
此时 polyfill 只要先做：

- `if (window.navigator.modelContext) return;`
  就不会覆盖原生属性，从而避免冲突。[2]

4.2 非常规场景（同一页面生命周期内“动态启用原生”）

如果存在类似“用户在 DevTools/flags 里切换实验特性而不刷新页面就生效”的机制（通常不成立），而你又把 polyfill 属性设为 `configurable:false`，那确实可能阻止后续重定义。

这类风险可以通过两种策略降低：

- polyfill 属性设为 configurable:true（可卸载/替换）；
- 或者明确声明“需要刷新页面才能切换原生/ polyfill”。

结合 `@mcp-b/global` 的实现，它选择了 `configurable:false` 并通过“检测到原生则不安装”来规避常规冲突；这对大多数场景足够，但对“热切换”不友好。[2]

5) 与 MV3 注入能力的耦合：CSP/序列化/返回值

如果你的 polyfill 是通过扩展注入：

- 使用 `chrome.scripting.executeScript({ world: 'MAIN' })` 是必要条件（上一节）。[4]
- 同时要意识到：当脚本注入到 main world 时，页面 CSP 生效，尽量避免 polyfill 在安装过程里使用 `eval`、动态外链脚本等容易触发 CSP 的行为。[3]

Areas of Consensus

- 规范给出了明确的目标属性与最小方法集合；polyfill 只要遵循 feature detection 并尽量模拟 SameObject/SecureContext，就可以做到“低冲突、可渐进迁移”。[1]
- 要让页面脚本使用 polyfill，必须在 `MAIN` world 注入安装逻辑，这是由 Chrome 的 isolated worlds 模型决定的。[3][4]

Areas of Debate

- **到底该把属性挂在 `navigator` 实例还是 `Navigator.prototype`**：两者都可能工作；实例挂载更简单、原型挂载更像原生但复杂度更高。不同站点/反爬/安全产品对 navigator 污染的敏感度也不同，需要结合你们目标站点做风险评估与 A/B 测试。

Sources

[1] Web Machine Learning Community Group — *WebMCP Draft Community Group Report*（`Navigator.modelContext` 的 SecureContext/SameObject/readonly 语义与 ModelContext API surface） https://webmachinelearning.github.io/webmcp/ （权威：规范草案）

[2] WebMCP-org/npm-packages — `@mcp-b/global` polyfill 源码（通过 `Object.defineProperty(window.navigator, 'modelContext', ...)` 安装，并检测原生 API） https://raw.githubusercontent.com/WebMCP-org/npm-packages/main/packages/global/src/global.ts （中可信：实现参考/事实佐证）

[3] Chrome for Developers — *Content scripts*（isolated worlds；以及“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）

[4] Chrome for Developers — *chrome.scripting API reference*（`ExecutionWorld` 的 `ISOLATED/MAIN` 语义）
https://developer.chrome.com/docs/extensions/reference/api/scripting （权威：官方）

Gaps and Further Research

- **在你们目标 Chrome 版本上的实测**：建议在 MAIN world 的真实页面里验证：
  - `Object.isExtensible(window.navigator)`
  - `Object.defineProperty(window.navigator, 'modelContext', ...)` 是否总成功
  - 属性描述符（writable/configurable/enumerable）在不同站点是否被安全脚本监控/拦截
    并记录在“注入兼容性矩阵”里，为 Phase 0 的适配策略兜底。
- **与页面自带 WebMCP 冲突处理**：如果页面未来自己实现 `navigator.modelContext`（不管是原生还是站点自带 polyfill），扩展侧注入必须严格“只在缺失时安装”，并提供可观测日志/诊断页，避免多方重复安装导致工具注册错乱。