ChatGPT

Googleログイン失敗問題の調査・解決提案

根本原因の分析 (Root Cause Analysis)

Chrome DevTools MCP環境から起動した自動化ブラウザでGoogleにログインできないのは、**Google側がブラウザの自動化を検出してログインをブロックしている**ためと考えられます [1] [2]。具体的な検出手法としては以下が推測されます。

- **DevToolsリモートデバッグの検出**: Chromeを `--remote-debugging-port` や `--remote-debugging-pipe` 付きで起動すると、デバッグモードが有効であることをGoogleが察知し、「このブラウザまたはアプリは安全でない可能性があります」エラーを出すようです [1]。実際、Stack Overflowの報告では「Chromeをデバッグモードで起動しているとGoogleログイン時にこのエラーが発生する」とあり、`--remote-debugging-port` オプションを外したところ問題が解消した例もあります [3]。

- **`navigator.webdriver` プロパティ**: Puppeteerなどの自動操作環境で起動したChromeでは、JavaScriptから `navigator.webdriver` を見ると `true` になっています。通常のブラウザでは `false` なので、これが**自動化ブラウザの明確なシグナル**となり得ます [4]。多くのサイトはこの値をチェックしてボットを検出しますが、Googleもログイン時に確認している可能性が高いです。

- **その他のブラウザ指紋**: `navigator.webdriver` 以外にも、ヘッドレス・自動化ブラウザ特有のフィンガープリントがあります。例: 有効なプラグインや言語設定の欠如、一定のBlink機能が無効化されている、画面サイズやユーザエージェントの不自然さ、など [5] [6]。今回のケースではヘッドレスモードではないものの、Chrome起動時に多数の `--disable-***` フラグを付与しているため、通常と異なる挙動（例えばバックグラウンド通信や拡張機能の無効化など）が間接的に検出される可能性もあります。

- **Googleのセキュリティポリシー**: 2019年末頃から、Googleは**自動化ツールによるログイン試行を積極的にブロック**し始めました [1]。これはユーザアカウントを保護するためで、「ブラウザが安全ではない（＝信頼できない制御下にある）」と判断した場合に認証を通さない仕組みになっています [2]。実際、Reddit上でも「Googleは明確に自動化ソリューションを検知してブロックしている」との指摘があります [2]。

以上のことから、本問題の根本原因は**Chromeを自動制御している痕跡（Automation Indicators）がGoogleログインシステムに感知されている**ことにあります。特に `--remote-debugging` によるデバッグモードや `navigator.webdriver` =trueといった明確なフラグが原因である可能性が極めて高いです。

技術的解決策 (Technical Solutions)

上記の原因を踏まえ、Googleによる自動化検出を回避するための技術的な解決策を検討します。

1. **自動化フラグの無効化・隠蔽**:
2. **Chrome起動オプションの見直し**: Puppeteer起動時に付与している `--enable-automation` フラグを除去します。Chromeの自動化用スイッチを無効化することで、自動制御モードであることをブラウザ自身に知らせないようにします [7]。具体的には、Puppeteerの `launch()` オプションで `ignoreDefaultArgs: ["--enable-automation"]` を指定し、この既定スイッチをオフにします [8]。

こうするとChrome内部で読み込まれる自動化用の拡張機能や設定が抑制され、`navigator.webdriver`を含む自動化痕跡が出にくくなります。

3. **Blink AutomationControlledの無効化**: ChromeのBlinkエンジンには自動化検出用の機能フラグがあり、これを無効化できます。`--disable-blink-features=AutomationControlled`という起動オプションを追加すると、**Chromeが自動的に`navigator.webdriver=true`を設定しなくなる**ため、自動化の指紋を隠せます [4]。このフラグはPuppeteerの`args`配列に追加可能で、実際にこのフラグを付けてブラウザを起動すると、`navigator.webdriver`が初めから存在しない（または`undefined`になる）状態でページを読み込めます [9] [10]。

4. **自動化インフォバーの抑制**: 既に設定済みかもしれませんが、`--disable-infobars`はChromeの「自動テストによって制御されています」というインフォバーを非表示にするフラグです。これは直接的な検出要因ではありませんが、目視上の煩わしさを減らすため有効です。

5. **`navigator.webdriver`プロパティの偽装**:
   万一Chromeの起動フラグだけでは`navigator.webdriver`が残ってしまう場合、**スクリプトによるプロパティ偽装**を行います。Puppeteerでは新しいページが開く前にスクリプトを注入できるので、以下のように`navigator.webdriver`を上書き/削除します [11]。例えば:

```
await page.evaluateOnNewDocument(() => {
  Object.defineProperty(navigator, 'webdriver', { get: () => undefined });
});
```

これにより、ページ側からは`navigator.webdriver`が`undefined`となり、trueを返さなくなります [11]。別のアプローチとして、Stack　Overflowで提案されている**プロトタイプからプロパティを削除する方法**もあります [12]:

```
// プロトタイプからwebdriverを削除する例
const newProto = navigator.__proto__;
delete newProto.webdriver;
navigator.__proto__ = newProto;
```

どちらの方法でも、サイトのボット検知スクリプトが`navigator.webdriver`をチェックした際に自動化ブラウザだとバレにくくなります。実際、この種の細工はPuppeteer製のステルスプラグインが内部で実行していることでもあります [13]。

1. **ユーザーエージェントと他の指紋対策**:
   現在のユーザーエージェント文字列（例: `Chrome/140.0.0.0`）が正常なChromeと一致しているか確認します。自動化環境ではChromeのバージョン部分が`0.0.0.0`のようになる場合がありますが、これは一部サイトで怪しまれる可能性があります [14]。必要に応じて`page.setUserAgent()`で実際の最新Chromeと同じ値に設定してください。また、`navigator.plugins`や`navigator.languages`などもチェックされる場合があります。Puppeteerステルスプラグインでは、これらを人間のブラウザらしく見せる細工（例えばプラグイン情報を適当に設定する等）も実装されています [15] [16]。今回のGoogleログインにどこまで必要かは不明ですが、**指紋の不自然さを総合的に減らす**ことが理想です。例えば画面サイズや色深度を標準的な値に設定する、`WebGL`や`Canvas`の挙動を本物に近づける等は高度な対策ですが、極力「普通のChrome」と同じ環境になるよう調整します。

2. **Puppeteer Extra Stealthの活用**:
   手動でフラグ設定やスクリプト注入を行う代わりに、コミュニティが提供する**puppeteer-extra-plugin-stealth**を使う方法があります。これはPuppeteerのプラグインで、ブラウザ起動後に自動化

特有の痕跡をまとめて隠蔽するものです。導入もシンプルで、`puppeteer-extra`経由でStealthプラグインを読み込むだけで、`navigator.webdriver`やPlugins，Languages，`chrome.app`など様々なプロパティを偽装してくれます[13]。コード例:

```
const puppeteer = require('puppeteer-extra');
const StealthPlugin = require('puppeteer-extra-plugin-stealth');
puppeteer.use(StealthPlugin());
// 以降は通常のpuppeteerと同様にlaunchやpage.gotoを実行
const browser = await puppeteer.launch({...});
```

このプラグインはデフォルトで複数のエヴェイジョン（回避）テクニックを適用し、一般的なボット検知をかわすよう設計されています[13]。Googleログインのケースでも、多くの開発者がこのプラグインで問題を回避できたと報告しています[17]。ただし、後述のリスク評価にあるように、万能ではない点に注意が必要です。

1. **Chromeユーザープロファイルの活用**:
   既に`userDataDir`を指定してプロファイルを保持しているとのことなので、**そのプロファイルに事前にログイン情報を持たせておく**手もあります。つまり、自動化なしの通常Chrome（または自動化を隠蔽したChrome）で一度Googleアカウントにログインし、認証クッキーやOAuthトークンをプロファイルに保存しておきます。そのプロファイルを使ってPuppeteerを起動すれば、ログイン画面を経ずにサービスにアクセスできる可能性があります。実際、Redditの議論では「**すでにGoogleにログイン済みのプロフィールを使ってSelenium/ブラウザを起動すれば良い**」というアドバイスもあります[18]。ログインフロー自体を自動化しようとするからブロックされるので、「ログイン済みの状態から開始する」ことで回避しようというアプローチです。これが可能なら、**認証済みセッションを再利用**する形で拡張機能のテストを継続できます。

2. **認証方式の変更**:
   Web経由のログインに固執しない代替として、Google APIを用いた認証やサービスアカウントの利用も検討します。例えば、テスト用のGoogleアカウントに対してあらかじめOAuth2.0でリフレッシュトークンを発行し、テスト時にはそのトークンを使って必要なAPIにアクセスする方法です。Chrome拡張機能がGoogleのOAuthフローを使用する場合、その部分だけは自動でなく**別プロセスでトークンを取得して注入**することも技術的には可能です（ただし拡張機能の動作要件によります）。また、もし拡張機能がGoogleのクライアントID/シークレットを使って認証するのであれば、テスト用に**モックの認証サーバ**を用意するか、Googleのテスト用クレデンシャルを使うことも検討できます。ただこの方法は、ブラウザ内で実際にログイン状態になるわけではないので、拡張機能の挙動（UIでGoogleにアクセスするケースなど）によっては適さない場合があります。

以上が主な技術的解決策の候補です。**推奨されるのは1と2の組み合わせ**（Chrome起動フラグ調整＋`navigator.webdriver`隠蔽）で、できればそれに加えて3の細かい調整も行うことです[6][8]。それでも不安が残る場合に4のステルスプラグイン導入を検討し、どうしてもGoogle公式の制限を回避できない場合は5や6のような迂回策に切り替えるという方針が考えられます。

具体的な実装方法 (Implementation Plan)

上記の解決策を実際に試すため、MCPプロジェクトのコードや設定を以下のように変更します。

1. **Puppeteer起動設定の変更**: `puppeteer.launch()`のオプションを見直します。具体的には:

2. **既定の自動化引数を無視**: `ignoreDefaultArgs: ['--enable-automation']` を追加します。これにより、Puppeteerがデフォルトで渡す `--enable-automation` スイッチを無効化します [8]。例えば現在の起動コードを以下のように変更します。

```
const browser = await puppeteer.launch({
  executablePath: resolvedExecutablePath,
  userDataDir: '/Users/usedhonda/chrome-mcp-profile',
  pipe: true,
  headless: false,
  ignoreDefaultArgs: ['--enable-automation'],
  args: [
   // 既存の引数から--enable-automationを除き、代わりに--disable-blink-featuresを追加
   '--disable-background-networking',
   '--disable-background-timer-throttling',
   // ...（中略）...
   '--disable-features=Translate,AcceptCHFrame,MediaRouter,...',
   '--enable-features=PdfOopif',
   '--user-data-dir=/Users/usedhonda/chrome-mcp-profile',
   '--remote-debugging-pipe',
   '--disable-blink-features=AutomationControlled'  // ★追加★
  ]
});
```

上記では、元の引数リストから `--enable-automation` を削除し、新たに `--disable-blink-features=AutomationControlled` を追加しています。 [4] で説明されているように、このフラグにより `navigator.webdriver` が自動設定されなくなります。

3. **`navigator.webdriver` 隠蔽スクリプトの注入**:
Chrome起動オプションの調整だけで問題が解決しない場合（あるいは念のため）、ページロード前に `navigator.webdriver` を偽装するスクリプトを仕込みます。Puppeteerでは `page.evaluateOnNewDocument()` を使って、新しいドキュメント（ナビゲーションごとに実行されるスクリプト）を登録できます。MCPのコード中でブラウザを開いた後、ログインページに遷移する前に例えば以下を実行します。

```
const page = await browser.newPage();
await page.evaluateOnNewDocument(() => {
  Object.defineProperty(window.navigator, 'webdriver', { get: () => undefined });
});
await page.goto('https://accounts.google.com/signin');
```

これで、そのページ内では `navigator.webdriver` は `undefined` として振る舞い、Googleのスクリプトがチェックしても自動化ブラウザだとわからないようになります [11]。この手法はSelenium+Chromeでも一般的で、Seleniumの場合は `excludeSwitches: ["enable-automation"]` とJavaScript実行で同様の隠蔽を行う例が公開されています [19] [20]。

1. **ユーザーエージェント・他の設定**:
必要に応じて、Puppeteerの `page.setUserAgent()` で標準Chromeと同じUAをセットします。また `page.setViewport()` でウィンドウサイズを一般的な大きさ（例: 1366x768）に設定し、

`navigator.plugins`や`languages`もステルスプラグインなしで対応するならカスタムスクリプトで偽装できます。ただしこれらを一つ一つ実装するのは大変なので、**puppeteer-extra-plugin-stealth**を導入した方が手軽です。導入後は先述のコードのように`puppeteer.use(StealthPlugin())`するだけで、主要な偽装は自動で行われます[13]。ステルスプラグインを使う場合でも、上記のChrome起動引数の調整（特に`--disable-blink-features=AutomationControlled`）は併用するのが望ましいでしょう。

2. **テスト手順**:
   実装変更後、以下の段階的なテストを行います。

3. **ステルス機能の検証**: まず`about://version`や開発者コンソールで、`navigator.webdriver`が確かに`false`または`undefined`になっていることを確認します。`console.log(navigator.webdriver)`をページ内で実行して目視確認することも可能です。
4. **Googleログイン試行**: 修正後の環境で`accounts.google.com`にアクセスし、実際にアカウントメールアドレス・パスワードを入力してみます。期待される挙動は、「このブラウザは安全ではない」エラーが表示されず、通常通りパスワード入力→2段階認証（設定されている場合）→ログイン成功まで進むことです。
5. **他サービスでの確認**: Googleに正常ログインできたら、GmailやYouTubeなど他のサービスにアクセスし、セッションが有効か（ログイン状態が保持されているか）を確認します。問題なく閲覧できれば、CookieやOAuthトークンの保持も想定通り機能しています。
6. **自動操作との連携**: その後、目的のChrome拡張機能を読み込んで、自動操作シナリオを実行し、拡張機能がGoogleサービスにアクセスできることを確認します。例えばGmail APIを拡張機能が使うならメールの一覧取得ができるか、YouTube連携なら正しくデータを取得・表示できるかなどをテストします。ログインが前提の機能が全てエラーなく動作すれば解決成功です。

もし上記テスト中に再びエラーが出たり、新たなブロック（CAPTCHA要求など）が発生した場合は、追加対策が必要です。その際は**puppeteer-extra-plugin-stealthで足りていない指紋を個別に潰す**（例えばCanvas偽装やWebGL情報偽装の実装を検討）か、あるいは**別経路での認証取得**（後述の代替アプローチ）を改めて検討してください。

1. **ログインセッションの永続化**:
   ログインが一度成功したら、`userDataDir`に保持されたクッキーやローカルストレージを**消さずに再利用**することが重要です。MCPの実行ごとにプロファイルをリセットしないようにし、同じプロファイルディレクトリを使い回すことで、毎回ログイン操作を行わずに済むようになります。テストアカウントに長期間ログインした状態が続くとセキュリティ上問題になる可能性もありますが、少なくとも開発中は利便性が向上します（定期的にパスワード再入力や2段階認証が求められることもあり得ますので注意してください）。

以上が実装と検証の手順です。要約すると、**Chrome起動設定の微調整＋自動化痕跡の隠蔽**により、Googleログインのブロックを回避する実装を行います。このアプローチは他の開発者コミュニティでも広く共有されており、Selenium/WebDriverでも同様の対策でGoogleへのログイン問題を乗り越えている事例があります[19] [20]。

代替アプローチ (Alternative Approaches)

上記はあくまで「ブラウザ側で偽装してログインする」方法でしたが、根本的に発想を変えた代替アプローチもいくつか考えられます。

- **既存セッションの利用**: 手動または別プロセスで取得したGoogleログインセッションを使い回す方法です。前述したように、一度人間の操作でログインしたChromeプロファイルをそのままPuppeteerで使用すれば、以後ログイン画面をスキップできます[18]。この方法ではGoogleのブロックを根本回避

できます。ただし、長期的にはセッションが切れたり、端末・アプリの認証が無効化される可能性もあり得ます。また、プロファイルに保存された認証情報（CookieやOAuthトークン）を適切に保護する必要があります。

- **公式APIの活用**: 拡張機能が必要とするGoogleサービスのデータを、ウェブUI経由ではなくGoogle提供のAPI経由で取得・操作することも検討します。例えばGmailのデータ取得であればGmail APIを直接呼び出す、Google Drive操作であればDrive APIを使う、といった手法です。この場合、ブラウザ内でログインセッションを確立する必要がなく、テストスクリプト側でOAuth2.0認証を別途行い、得られたアクセストークンをHTTPリクエストに付与することで用が足ります。拡張機能側も、もし外部からトークンを受け取れるよう実装を変更できるのであれば、UIログインなしでのテストが可能になるでしょう。しかし**拡張機能の動作検証**という目的上、実ユーザと同じフロー（ブラウザ上でのOAuth認証→Cookieセット）を通すことに意義があるなら、この方法は本質的解決にはなりません。

- **専用テストモードの実装**: プロジェクト側で、テスト時にはGoogle認証をスキップまたはモックできるような**裏口機能**を設けるのも一案です。例えば、拡張機能にテスト用の設定を設けて、固定のテスト用クッキー値やトークンを内部的に使用するようにする、といったものです。これにより、Googleにアクセスせずとも拡張機能が「ログイン済み」と見なして動作確認できます。ただしGoogleサービスとの正確な連携をテストできなくなるため、本来の目的から逸れてしまう恐れがあります。

- **他のブラウザの利用**: 極端な策ですが、Chromeではなく**Firefox**など他のブラウザでログインしてみる方法もあります。Selenium経由でFirefoxを操作するとGoogleログインが比較的通りやすいという報告例も稀にあります（Firefoxは`navigator.webdriver`を最初から持たないため検出が難しいとも言われます）。もっとも、今回MCPはChrome DevToolsベースのプロジェクトなので現実的ではないですが、どうしてもChrome側で解決不能な場合の最後の手段として言及します。

代替アプローチはいずれも一長一短で、根本的には**「Googleの意図する正規の手段」で認証する**ことを検討するものです。しかし、MCPの目的（人間がChrome上で拡張機能を操作するのと同じ状況を自動テストしたい）を考えると、可能な限り**通常のログイン手順を実際に踏んだ状態**でテストを行えることが望ましいでしょう。そのため、まずは前述の技術的解決策でGoogle側のブロックをかわしつつ、どうしても無理な部分のみ代替案を適用する方針が現実的です。

リスク評価と長期的考慮事項 (Risk Assessment & Long-term Considerations)

提案した解決策には、技術的・倫理的なリスクや留意すべき点も存在します。

- **セキュリティと倫理面**: Googleが自動化ブラウザをブロックしているのは、悪意あるボットからユーザアカウントを守るための措置です。これを回避するということは、ある意味**Googleのセキュリティ機構を意図的にすり抜ける**行為です。自社内のテスト目的でありユーザ本人のアカウントを扱う場合でも、Googleの利用規約に抵触する可能性があります（たとえば「自動手段によるアカウントアクセスの禁止」といった条項が該当し得ます）。従って、本手法は社内テストや開発用途に限定し、決して第三者のアカウントや許可無きデータ取得に使わないことが重要です。倫理的にも、ユーザに無断で自動ログインを実行するような機能は提供すべきではありません。

- **安定性**: **今回実装する回避策が今後も通用する保証はありません。** Google側の検出ロジックは継続的に進化しており、単一のシグナルを消しただけでは根本的な解決にならない場合があります[21]。例えば、`navigator.webdriver`を隠す策を講じても、他の手掛かり（例えばTiming APIを使った処理のタイミングの不自然さや、Canvas描画結果の差異など）で将来的に検出される可能性があります[22]。実際に「BlinkのAutomationControlledフラグを無効化しただけでは不十分で、他の多くの

ボットシグナルも潰す必要がある」と指摘する資料もあります [21] 。長期的に見れば、Googleが検出手法を強化する度にこちらも対策をアップデートする**イタチごっこ**になりかねません。

- **サードパーティ製ツールへの依存**: Stealthプラグインなどオープンソースの回避ソリューションは便利ですが、これらも**常に最新の検出技術に追随できるとは限らない**という点に注意が必要です [23] 。オープンソースのステルスプラグインはボランティアベースで更新されていますが、Googleなど大企業の対策が高度化すると、プラグイン側の追従にラグが出ることがあります [23] 。場合によっては、ステルスプラグイン自体が検出対象になる可能性すらあります（特定バージョンのプラグインが作る独特の挙動パターンを機械学習で検知する、といった例も理論上はあり得ます）。そのため、プラグインに頼りきりにせず、自チームでも定期的に検出回避策の見直しを行うべきです。

- **デバッグ容易性への影響**: `--remote-debugging-pipe` を使わずにChromeを起動することはPuppeteer制御上できませんので、デバッグ接続自体は維持します。しかし、仮に `--remote-debugging-port` を指定しない（あるいはpipeのみ利用する）運用にすると、**外部からの手動デバッグアタッチがやりにくくなる**可能性があります。MCP開発時に手動でChromeに接続して調査したいケースでは、デバッグフラグを一時的に戻す必要があるでしょう。つまり**テスト用設定**と**デバッグ用設定**を用途に応じて切り替える、といった運用上の配慮が求められます。

- **ユーザーデータの保護**: プロファイルに本物のGoogleアカウントのクッキーやOAuthトークンを保存して運用する場合、そのデータの取り扱いには最新の注意が必要です。特に社内で複数人がそのプロファイルディレクトリにアクセスできてしまうと、最悪場合によっては不正利用のリスクがあります。また、テストとはいえGoogleアカウントに大量の自動操作を行うと、Google側でアカウントに一時的なロックが掛かったりCAPTCHAが発動する可能性もあります。**アクセス頻度や操作内容が不自然になり過ぎないよう制御**し、人間の利用パターンに近い振る舞いを心掛けることも大切です [5] [24] 。

- **法的遵守**: もしこのプロジェクトが社外提供されるツールに組み込まれるなら、Googleの利用規約や各種法規に抵触しないか法務確認が必要です。一般ユーザー向け製品であれば、自動ログインを実現するよりもGoogle公式のOAuthフローに従うべきですし、企業内利用ツールであっても社内ポリシー上問題ないかチェックが要ります。

総合すると、今回提案したのは**あくまで開発・テスト目的のワークアラウンド**です。根本的にはGoogleが公式に提供する認証手段（OAuthやサードパーティ認証フロー）を使うのが筋ですが、開発要件上ブラウザ上でのログインセッションが必要ということで、止むを得ず自動化検出を避ける策を講じています [25] 。実運用では、こうした「隠れる」方法が将来的に破綻するリスクを常に念頭に置き、代替プランを用意しておくことが重要です。

最後に、他プロジェクトでの成功事例としては、**puppeteer-extra-stealthを導入してGoogleログインを突破したケース**や、**ChromeのAutomationControlledフラグ無効化だけでログインできたケース**などが報告されています [17] [3] 。一方で、「ステルスプラグインでは不十分で追加対策が必要だった」という声 [22] もあり、状況次第です。したがって、本提案の手法を実装した後も、ログインが確実に成功するかどうかを監視し、もし再度ブロックされるようであれば迅速に対処法をアップデートできるよう、**継続的な検証**体制を整えておくことを推奨します。

以上の対策と考慮事項を踏まえ、MCPプロジェクトでのGoogleログイン問題に対処してください。これらの変更により、**Chrome拡張機能の自動テストがGoogleサービスでも滞りなく行えるようになること**を期待しています。 [1] [17]

---

1 3 "Browser or app may not be secure. Try using a different browser." error with Flutter Firebase Google Login - Stack Overflow
https://stackoverflow.com/questions/59480956/browser-or-app-may-not-be-secure-try-using-a-different-browser-error-with-fl

2 13 17 18 22 25 Browser Automation to Log into Google with Headless Server : r/webscraping
https://www.reddit.com/r/webscraping/comments/11rzcz9/browser_automation_to_log_into_google_with/

4 9 10 21 23 Using --disable-blink-features=AutomationControlled to Reduce Detection in Headless Chrome - ZenRows
https://www.zenrows.com/blog/disable-blink-features-automationcontrolled

5 6 7 8 11 14 15 16 19 20 24 How do I prevent detection of Headless Chromium by websites? | WebScraping.AI
https://webscraping.ai/faq/headless-chromium/how-do-i-prevent-detection-of-headless-chromium-by-websites

12 selenium - Puppeteer Delete Navigator.Webdriver - Stack Overflow
https://stackoverflow.com/questions/53934397/puppeteer-delete-navigator-webdriver