

# picoclaw によるリモート Windows ロック・ログイン機能仕様書

最終更新: 2026-02-26

## 概要

NanoKVM-USB デスクトップアプリに組み込まれた AI エージェント「picoclaw」を通じて、自然言語の指示でリモート接続された Windows PC のロック・ログイン操作を行う機能です。

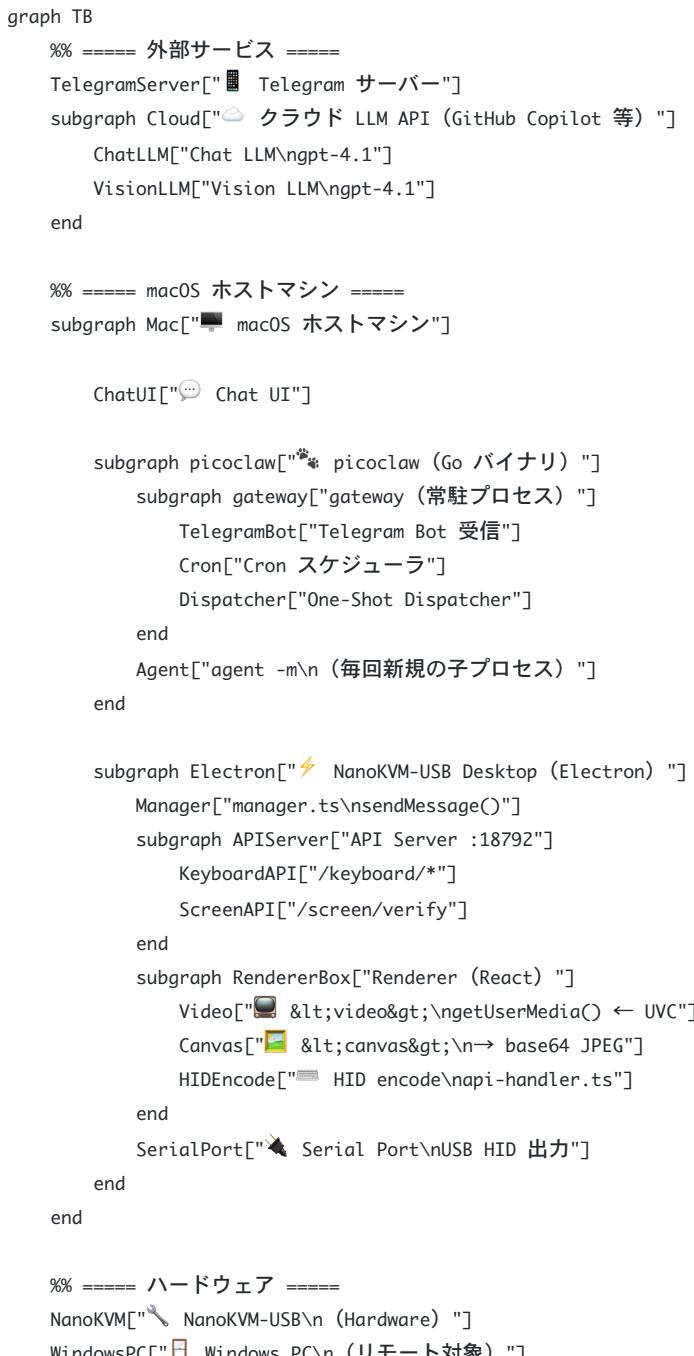
入力経路: チャット UI (アプリ内) または Telegram ボット

出力先: NanoKVM-USB ハードウェア経由で Windows PC に HID キーボード入力を送信

処理方式: 両経路とも `picoclaw agent -m` を毎回サブプロセスとして起動 (One-Shot 方式)

## システム構成ブロック図

### 全体アーキテクチャ



```

%% ===== 接続線 =====
%% Telegram 経路
TelegramServer -- "Bot API" --> TelegramBot
TelegramBot --> Dispatcher
Cron --> Dispatcher
Dispatcher -- "spawn\n (メッセージごと)" --> Agent

%% Chat UI 経路
ChatUI --> Manager
Manager -- "spawn\nagent -m" --> Agent

%% agent → LLM
Agent -- "HTTPS\nChat LLM 呼出" --> ChatLLM

%% agent → API Server (Tool 実行)
Agent -- "HTTP POST\nTool 実行" --> KeyboardAPI
Agent -- "HTTP POST\n画面検証" --> ScreenAPI

%% API Server → Renderer (操作系)
KeyboardAPI -- "IPC" --> HIDEncode
HIDEncode -- "IPC" --> SerialPort

%% API Server → Renderer (映像キャプチャ系)
ScreenAPI -- "IPC\nキャプチャ要求" --> Canvas
Video -- "drawImage()" --> Canvas
Canvas -- "IPC\nbase64 JPEG" --> ScreenAPI
ScreenAPI -- "HTTPS\nbase64 JPEG 送信" --> VisionLLM

%% ハードウェア接続
SerialPort -- "USB" --> NanoKVM
NanoKVM -- "USB HID\n操作転送" --> WindowsPC
WindowsPC -- "HDMI\n映像出力" --> NanoKVM
NanoKVM -- "USB UVC\n映像転送" --> Video

%% スタイル
style Cloud fill:#e8f4fd,stroke:#4a90d9
style picoclaw fill:#f0f9e8,stroke:#7cb342
style Electron fill:#fff3e0,stroke:#ff9800
style gateway fill:#e8f5e9,stroke:#66bb6a
style RendererBox fill:#fce4ec,stroke:#e91e63
style APIServer fill:#fff8e1,stroke:#ffc107
style NanoKVM fill:#f3e5f5,stroke:#9c27b0
style WindowsPC fill:#e3f2fd,stroke:#2196f3

```

## Telegram ロック操作のシーケンス図

```

sequenceDiagram
    actor User as ユーザー
    participant TG as Telegram
    participant GW as gateway<br>(常駐)
    participant Agent as agent -m<br>(子プロセス)
    participant ChatLLM as Chat LLM<br>(Groq)
    participant API as API Server<br>:18792
    participant Render as Renderer
    participant KVM as NanoKVM
    participant WinPC as Windows PC

```

User->>TG: 「ロックして」

TG->>GW: Bot API 通知

Note over GW: One-Shot Dispatcher

GW->>Agent: spawn (新規プロセス)

```
rect rgb(232, 244, 253)
```

Note over Agent,ChatLLM: Chat LLM にユーザー意図を問い合わせ

Agent->>ChatLLM: HTTPS (メッセージ送信)

ChatLLM-->Agent: Tool Call: nanokvm\_lock

end

```
rect rgb(255, 243, 224)
```

Note over Agent,WinPC: Win+L キー送信

Agent->>API: POST /keyboard/type

API->>Render: IPC (HID encode)

Render->>KVM: Serial Port (USB)

KVM->>WinPC: USB HID (Win+L)

Note over WinPC: ロック画面に遷移

end

Note over Agent: ⏳ 3秒待機

```
rect rgb(252, 228, 236)
```

Note over Agent,ChatLLM: 画面検証 (Vision LLM)

Agent->>API: POST /screen/verify

API->>Render: IPC (キャプチャ要求)

Note over Render: drawImage() → base64 JPEG

WinPC-->KVM: HDMI 映像

KVM-->Render: USB UVC 映像

Render-->>API: IPC (base64 JPEG)

API->>ChatLLM: HTTPS (Vision LLM)

Note right of ChatLLM: Llama 4 Scout<br>画面を解析

ChatLLM-->>API: "LOCK\_SCREEN"

API-->>Agent: 検証結果

end

Agent-->>GW: stdout: "✅ ロック完了"

Note over Agent: プロセス終了

GW->>TG: 返信送信

TG->>User: "✅ ロック完了"

#### データフローの要点:

方向	フロー	説明
➡ 操作	Chat/Telegram → picoclaw agent -m → Chat LLM → Tool Call → API Server → Renderer → Serial Port → NanoKVM → Windows PC	キー・マウス操作の送信
⬅ 映像	Windows PC → HDMI → NanoKVM → USB (UVC) → Renderer <video> (getUserMedia)	HDMI 映像のリアルタイム表示
➡ 検証	API Server → Renderer canvas キャプチャ → base64 JPEG → Vision LLM (Groq) → 判定結果	画面キャプチャ + Vision 解析

#### 対応コマンド

## 1. ロック（画面ロック）

項目	内容
操作内容	Windows のショートカット Win + L を送信
対応フレーズ例	「Windowsをロックして」「PCをロックしてください」「ロック」「lock the screen」
内部動作	Win キー押下 → L キー押下 → 100ms 保持 → 逆順にリリース
事後検証	3秒後に画面キャプチャ → Vision LLM で LOCK_SCREEN を確認

## 2. ログイン（PIN / パスワード入力）

項目	内容
操作内容	ロック画面を解除し、PIN コードまたはパスワードを入力してログイン
対応フレーズ例	「PINコード mypin でログインして」「パスワードで入って」「login with PIN」
事後検証	15秒後に画面キャプチャ → Vision LLM で結果を判定

### ログインシーケンス（PINのみ）

ステップ	操作	目的	待機時間
0	Escape キー押下	前回のPINエラーダイアログが残っている場合に閉じる	300ms
1	Space キー押下	ロック画面からサインイン画面を呼び起す	500ms
1b	Space キー再押下	バックアップの起動操作	-
2	待機	Windows がPIN入力欄を描画・フォーカスするのを待つ	3,000ms
3	Backspace × 20回	入力欄の既存文字をクリア (Ctrl+A はPIN欄で無効のため)	各30ms間隔
4	PIN を1文字ずつ入力	HID キーボードレポートで各文字を送信	各150ms間隔
5	Enter キー押下	PIN を送信してログイン	-

### ログインシーケンス（ユーザー名 + パスワード）

ステップ	操作	目的	待機時間
0	Escape キー押下	エラーダイアログを閉じる（前回失敗時の残り）	300ms
1～3	上記と同じ	画面起動・フィールドクリア	同上
4	ユーザー名を入力	ユーザー名欄にテキスト入力	300ms
5	Tab キー押下	パスワード欄に移動	300ms
6	パスワードを入力	パスワード欄にテキスト入力	300ms
7	Enter キー押下	ログイン実行	-

**Note:** ユーザーが明示的に指定しない限り、ユーザー名は送信されません。LLM がOS 名やボット名をユーザー名として使用することを防ぐバリデーションが入っています。

## picoclaw ツール呼び出しの流れ

picoclaw は Go 側でネイティブなツール実行（function calling）を行います。また、LLM がテキスト中にツール呼び出しを埋め込むケースに備え、Electron 側のインターフェンス（manager.ts）がフォールバック検出を行います。

## Go 側ネイティブツール（主要パス）

picoclaw agent はツール定義を LLM に渡し、構造化された function call で実行します:

ツール名	説明	パラメータ
nanokvm_lock	Win+L でロック	なし
nanokvm_login	PIN/パスワード入力でログイン	password, username?
nanokvm_shortcut	キーボードショートカット送信	keys[]
nanokvm_type	テキスト入力	text
nanokvm_mouse_click	マウスクリック	button

## Electron 側インターフェンス（フォールバック）

LLM がテキスト出力にツール呼び出しを含めた場合の検出パターン（3形式対応）：

優先度	フォーマット	例
1	アクションタグ	<<nanokvm:login:mypin>>
2	JSON オブジェクト	{"type": "function", "name": "nanokvm_login", "parameters": {"password": "..."}}
3	Python 関数呼び出し	nanokvm_login(password='...')

## 重複実行防止

- 同一エンドポイント + 同一パラメータのリクエストは **15秒間の重複排除** (debounce)
- Go 側がネイティブ実行 → HTTP API 呼び出し → インターフェンスも同じ API を呼ぼうとする → debounce で弾かれる

## レスポンスマッセージ

ユーザーに表示するメッセージは、LLM 出力からツール呼び出し構文を除去（stripActionTags）して生成されます。

### 除去対象

- <<nanokvm:...>> アクションタグ
- nanokvm\_ を含む JSON オブジェクト
- nanokvm\_xxx(...) 関数呼び出し構文
- LLM プリアンブル文（例: "The function call that best answers the prompt is:"）

### フォールバック

除去後にテキストが空になった場合、「コマンドを実行しました」がデフォルトメッセージとして表示されます。

## 対応キーワード

ショートカットで使用できるキーワードは LLM の出力する一般的な名前から自動変換されます。

入力名	変換先 (HID コード)
Win, Windows, Meta, Cmd	MetaLeft
Ctrl, Control	ControlLeft
Alt, Option	AltLeft
Shift	ShiftLeft

Del	Delete
Esc	Escape
Return	Enter
A～Z（单一文字）	KeyA～KeyZ
0～9（单一数字）	Digit0～Digit9
F1～F24	F1～F24

## API エンドポイント一覧

HTTP API サーバー（127.0.0.1:18792）が提供するエンドポイント：

メソッド	パス	パラメータ	説明
POST	/api/keyboard/shortcut	{"keys": ["Win", "L"]}	キーボードショートカット送信
POST	/api/keyboard/login	{"password": "...", "username?": "user"}	Windows ログイン実行
POST	/api/keyboard/type	{"text": "Hello"}	テキスト入力
POST	/api/mouse/click	{"button": "left"}	マウスクリック
GET	/api/screen/capture	なし	現在の画面をキャプチャ（base64 JPEG）
POST	/api/screen/verify	{"action": "lock" "login" "status"}	画面状態を Vision LLM で検証・確認

## 入力経路別の挙動

項目	チャット UI	Telegram
picoclaw 常駐プロセス	なし（都度起動）	picoclaw gateway（常駐）
メッセージ処理	manager.ts → spawn agent -m	One-Shot Dispatcher → spawn agent -m
LLM呼び出し	サブプロセス内	サブプロセス内（同一）
ツール実行	Go 側ネイティブ	Go 側ネイティブ（同一）
レスポンス表示	チャット吹き出し	Telegram メッセージ
セッション蓄積	なし（毎回フレッシュ）	なし（毎回フレッシュ）

## 制約事項

- 同時押しキー数：HID 仕様により最大6キー（修飾キーは別枠）
- 文字入力：ASCII 英数字・基本記号のみ対応（日本語入力は非対応）
- マウス移動：座標指定は未実装（クリックのみ対応）
- ログイン待機：PIN 入力後 15 秒の固定待機（デスクトップ描画完了まで）
- NanoKVM 操作回数：1メッセージあたり最大 4 回（不要な反復を防止）
- ユーザー名バリデーション："windows", "linux", "ubuntu" 等の OS 名は自動除外

## 画面状態確認機能（Screen Check）

「画面状態を確認して」などの指示で、リモート PC の現在の画面をキャプチャし、Vision LLM で分析して結果を返却します。

## 対応キーワード

- 「画面状態を確認して」
- 「今何が映ってる？」
- 「画面を見て」
- 「スクリーンショット」
- "screen check"

## 動作フロー

- picoclaw エージェントが `nanokvm_screen_check` ツールを呼び出し
- Go 側が `POST /api/screen/verify` に `{"action": "status"}` を送信
- Electron API Server が HDMI キャプチャを実行
- 黒画面 (**brightness < 3**) の場合: マウスジグル (1px右+1px左) を HID 送信してスリープ復帰を試行 → 2秒待機 → 再キャプチャ
- 映像取得成功なら Vision LLM に汎用プロンプトで画面内容を記述させる
- ステータス分類 ( `DESKTOP` / `LOCK_SCREEN` / `LOGIN_SCREEN` / `DESCRIBED` ) と詳細説明を返却
- リトライ後も黒画面の場合は `BLACK_SCREEN` ステータスを返却 ('マウスジグルでの復帰を試みましたが画面が変わりませんでした')
- 早期終了で即座にユーザーに結果を表示

**ウェイクジグルの安全性:** ネットの移動量は  $\pm 0px$  (1px右移動+1px左移動)。ユーザーがPCを操作中でも知覚不能。クリックやキー入力と異なり、アプリケーション操作に一切影響しない。

## レスポンス例

ステータス	アイコン	例
<code>DESKTOP</code>		デスクトップ画面です。複数のアプリケーションウィンドウが見えます。
<code>LOCK_SCREEN</code>		ロック画面です。時計とユーザーアバターが表示されています。
<code>LOGIN_SCREEN</code>		サインイン画面です。PIN 入力フィールドが表示されています。
<code>DESCRIBED</code>		画面状態: (Vision LLM の記述)
<code>NO_VIDEO</code>		映像がありません。PCが接続されていてストリーミングが開始されていることを確認してください。
<code>BLACK_SCREEN</code>		画面が真っ黒です。マウスジグルでの復帰を試みましたが画面が変わりませんでした。PCの電源が入っていることを確認してください。
<code>NO_SIGNAL</code>		信号がありません。黒い画面またはブランク画面が検出されました。

## エラーハンドリング

状況	原因	Go側の判定	ユーザーへのメッセージ
<code>映像なし (screen_check)</code>	PC 未接続 / ストリーミング未開始	<code>success=false, status="NO_VIDEO"</code>	映像がありません。PCが NanoKVM-USBに接続されていて...
<code>映像なし (lock)</code>	同上	<code>v.Status == "NO_VIDEO"</code>	Win+Lを送信しましたが、映像がないため結果を確認できません...
<code>映像なし (login)</code>	同上	<code>v.Status == "NO_VIDEO"</code>	ログイン操作を送信しましたが、映像がないため結果を確認できません...
<code>黒画面 (screen_check)</code>	PC スリープ / HDMI 信号未安定	<code>success=false, status="BLACK_SCREEN"</code>	画面が真っ黒です。PCがスリープ中か、HDMI信号が安定していない...
<code>黒画面 (lock)</code>	同上	<code>v.Status == "BLACK_SCREEN"</code>	Win+Lを送信しましたが、画面が真っ黒です。PCがスリープ中か...

黒画面 (login)	同上	v.Status == "BLACK_SCREEN"	⚠️ ログイン操作を送信しましたが、画面が真っ黒です...
アプリ未起動	API Server に接続不可	result == nil	NanoKVM-USB デスクトップアプリに接続できませんでした...
Vision 未設定	Vision LLM プロバイダ/モデル未設定	visionConfigured=false	Vision LLM が設定されていません...
セッション履歴破損	並列 tool_calls のレスポンス欠落	sanitizeToolCallHistory() で自動トランケート	(自動修復：ログに WARNING 出力、ユーザーへのメッセージなし)

## チャット用 LLM (Chat LLM)

picoclaw のチャット機能（自然言語によるコマンド解釈・応答生成）に使用する LLM プロバイダとモデルの一覧です。自動更新機能によりモデルリストは定期的に更新されますが、以下が現在のデフォルト構成です。

### チャット LLM プロバイダ・モデル一覧

プロバイダ	デフォルトモデル	認証方式	料金	備考
Groq	llama-3.3-70b-versatile	API Key	無料枠あり	推奨: 高速・クレカ不要
OpenAI	gpt-5.2	API Key	有料	高品質・安定
Anthropic	claude-sonnet-4.6	API Key	有料	高品質
DeepSeek	deepseek-chat	API Key	安価	コスト効率
Google Gemini	gemini-2.0-flash-exp	API Key	無料枠あり	高速
GitHub Copilot	gpt-4.1	OAuth (gh CLI)	無料	gh auth login で認証・推奨
OpenRouter	auto	API Key	従量制	多プロバイダ統合
Mistral AI	mistral-small-latest	API Key	有料	欧州拠点
Ollama	llama3	不要	無料	ローカル実行
VLLM	custom-model	不要	無料	ローカル実行
NVIDIA	nemotron-4-340b-instruct	API Key	要確認	GPU 推論
Cerebras	llama-3.3-70b	API Key	要確認	高速推論
Qwen	qwen-plus	API Key	安価	中国拠点
Zhipu AI	glm-4.7	API Key	安価	中国拠点
Moonshot	moonshot-v1-8k	API Key	安価	中国拠点
Volcengine	doubao-pro-32k	API Key	安価	ByteDance
ShengsuanYun	deepseek-v3	API Key	安価	中国拠点

### GitHub Copilot / GitHub Models 対応モデル

GitHub Copilot プロバイダは [GitHub Models API](#) を使用します。 gh auth login で取得した OAuth トークン ( gho\_\* ) で認証し、API キーの手動入力は不要です。

初回認証フロー

GitHub Copilot を使用するには GitHub CLI ( gh ) のインストールと認証が必要です。アプリ内の「🔑 GitHub 認証を開始」ボタンから以下のフローで認証できます:

```
sequenceDiagram
    actor User as ユーザー
    participant UI as ⚡ NanoKVM-USB<br>設定画面
    participant Main as Main Process
    participant GH as 🐳 gh CLI
    participant Browser as 🌐 ブラウザ<br>github.com/login/device

    Note over UI: 🛡️ GitHub Copilot 接続設定<br>⚠️ GitHub 認証が必要です

    User->>UI: 「🔑 GitHub 認証を開始」クリック
    UI->>Main: IPC: INITIATE_GITHUB_AUTH

    rect rgb(255, 243, 224)
        Note over Main,GH: gh CLI で Device Flow 開始
        Main->>GH: spawn: gh auth login<br>-h github.com -p https -w
        GH-->Main: stderr: one-time code<br>「XXXX-XXXX」
    end

    Main-->>UI: { code: "XXXX-XXXX", url }

    rect rgb(232, 244, 253)
        Note over UI,Browser: デバイスコード表示 + ブラウザ起動
        UI-->UI: 画面にコード表示<br>「XXXX-XXXX」
        UI-->Browser: shell.openExternal()<br>github.com/login/device
        User-->Browser: コードを入力
        User-->Browser: 認証を許可
    end

    rect rgb(232, 245, 233)
        Note over UI,GH: ポーリングで認証完了を検出
        loop 3秒間隔ポーリング (最大5分)
            UI-->Main: IPC: DETECT_GITHUB_AUTH
            Main-->GH: gh auth token
            GH-->Main: gho_xxxxxx (OAuth トークン)
            Main-->>UI: { found: true, token, user }
        end
    end

    UI-->>UI: ✅ GitHub 認証済み (user: xxx)
    UI-->>UI: トークンを config に自動保存
    UI-->>User: 🎉 認証完了通知
```

#### 前提条件:

- [GitHub CLI \( gh \)](#) がインストール済み
- GitHub アカウントを持っている（無料アカウントでOK）
- GitHub Copilot の登録は**不要**（GitHub Models API は全ユーザーに無料提供）

#### 実装ファイル:

ファイル	役割
manager.ts → initiateGitHubAuth()	gh auth login --web を spawn、デバイスコード取得
manager.ts → detectGitHubToken()	gh auth token でトークン検出

<code>manager.ts</code> → <code>cancelGitHubAuth()</code>	認証プロセスを kill
<code>picoclaw.ts</code> (events)	IPC ハンドラ ( <code>INITIATE_GITHUB_AUTH</code> , <code>CANCEL_GITHUB_AUTH</code> )
<code>picoclaw.tsx</code> (renderer)	UI: デバイスコード表示、ポーリング、完了通知

モデル名	種別	Vision	備考
<b>gpt-4o-mini</b>	Chat	✓	高速・軽量
<b>gpt-4o</b>	Chat	✓	高品質
<b>gpt-4.1</b>	Chat	✓	推奨: 最新世代・高品質
<b>gpt-4.1-mini</b>	Chat	✓	最新世代・軽量
<b>gpt-4.1-nano</b>	Chat	-	超軽量
<b>o1</b>	Reasoning	-	推論特化
<b>o3</b>	Reasoning	-	推論特化
<b>o3-mini</b>	Reasoning	-	推論特化・軽量
<b>o4-mini</b>	Reasoning	-	最新推論モデル
<b>Meta-Llama-3.1-405B-Instruct</b>	Chat	-	大型オープン
<b>Meta-Llama-3.1-8B-Instruct</b>	Chat	-	軽量オープン
<b>Llama-3.2-11B-Vision-Instruct</b>	Chat	✓	<b>Vision対応:</b> 画面検証に使用可能
<b>Llama-3.2-90B-Vision-Instruct</b>	Chat	✓	<b>Vision対応:</b> 高精度
<b>Phi-4</b>	Chat	-	Microsoft 軽量
<b>Phi-4-multimodal-instruct</b>	Chat	✓	<b>Vision対応:</b> Microsoft
<b>DeepSeek-R1</b>	Reasoning	-	推論特化
<b>MAI-DS-R1</b>	Reasoning	-	Microsoft + DeepSeek
<b>Mistral-large-2407</b>	Chat	-	Mistral 大型

**Note:** GitHub Copilot の Web/VS Code 版では Claude 等の Anthropic モデルも選択可能ですが、 GitHub Models API (`models.inference.ai.azure.com`) ではサポートされていません。 picoclaw は GitHub Models API を経由するため、上記のモデルのみ利用可能です。

## モデルリスト自動更新機能

picoclaw のモデルリストは、プロバイダが新モデルを追加した際に自動更新されます。

### 概要

- 更新対象:** 各プロバイダが提供するモデルの一覧 (`~/.picoclaw/config.json` の `model_list`)
- 更新元:** `picoclaw providers` コマンドで取得するデフォルトモデル + UI 上の追加定義
- 表示:** 設定画面のモデル選択ドロップダウンに反映

### スケジュール設定

項目	オプション	デフォルト
頻度	日次 / 週次 / 月次	月次

実行時刻	0:00 ~ 23:00	<b>0:00</b>
曜日 (週次)	月~日	月曜日
日付 (月次)	1~28日	1日
有効/無効	トグル	<b>有効</b>

## 更新フロー

```

flowchart TD
    Start["⚡️ Electron 起動時<br>ModelUpdater.init()"] --> CheckSchedule
    CheckSchedule["📅 スケジュール確認<br>config.json 読み込み"] --> CalcNext["⏰ 次回実行時刻を計算<br>setTimeout() 設定"]
    CalcNext -- "時刻到達" --> RunProviders["⚙️ picoclaw providers<br>コマンド実行"]
    Manual["👤 手動: ⏱️ 今すぐ更新"] --> RunProviders
    RunProviders --> UpdateConfig["📝 config.json の<br>model_list を更新"]
    UpdateConfig --> CheckModel{"現在のモデルが<br>新リストに存在する??"}
    CheckModel -- "✅ 存在する" --> NoChange["変更なし"]
    CheckModel -- "❌ 存在しない" --> AutoSwitch["⚠️ デフォルトモデルに<br>自動切替 + 警告表示"]
    NoChange --> CalcNext
    AutoSwitch --> CalcNext

    style Start fill:#fff3e0,stroke:#ff9800
    style RunProviders fill:#f0f9e8,stroke:#7cb342
    style CheckModel fill:#e8f4fd,stroke:#4a90d9
    style AutoSwitch fill:#fce4ec,stroke:#e91e63
    style Manual fill:#e8f5e9,stroke:#66bb6a

```

## 手動更新

設定画面の「🕒 今すぐ更新」ボタンで即時実行できます。

## Config 構造 (自動更新関連)

```
{
  "model_update": {
    "enabled": true,
    "frequency": "monthly",
    "hour": 0,
    "dayOfMonth": 1
  }
}
```

ステータスは `~/.picoclaw/model-update-status.json` に保存:

```
{
  "lastChecked": "2026-02-25T00:00:00Z",
  "nextCheck": "2026-03-01T00:00:00Z",
  "lastUpdatedModels": ["groq: 4 models", "openai: 1 model"],
```

```
"autoSwitched": false  
}
```

## 画面検証機能 (Vision LLM)

ロック・ログインコマンド実行後、NanoKVM-USB の HDMI キャプチャ映像をスクリーンキャプチャし、専用の Vision LLM で画面内容を解析して結果を自動判定します。

### 設計思想: チャット用 LLM と Vision LLM の分離

```
block-beta  
    columns 1  
    block:settings["⚙️ 設定画面 (picoclaw.tsx)"]  
        columns 2  
        block:chat["💬 チャット LLM"]:2  
            columns 2  
            chatProvider["Provider: GitHub Copilot"]  
            chatModel["Model: gpt-4.1"]  
            chatKey["API Key: gsk_..."]:2  
        end  
        block:vision["👁️ 画面検証 Vision LLM"]:2  
            columns 2  
            visionProvider["Provider: GitHub Copilot"]  
            visionModel["Model: gpt-4.1 🕵️ "]  
            visionKey["API Key: gsk_... (共有可)"]:2  
        end  
    end  
  
    style settings fill:#fff3e0,stroke:#ff9800  
    style chat fill:#e8f4fd,stroke:#4a90d9  
    style vision fill:#fce4ec,stroke:#e91e63
```

理由: チャットには安価なテキスト LLM、画面検証には Vision 対応 LLM という使い分け。例: チャット = gpt-4.1 (GitHub Copilot 無料)  
+ Vision = gpt-4.1 (GitHub Copilot 無料)

推奨構成: GitHub Copilot の `gpt-4.1` を `Chat`・`Vision` 両方に設定。無料で高品質かつプロンプト調整が最小限で済みます。

### 検証フロー

```
flowchart TD  
    Command["⚙️ ロック or ログイン<br>コマンド実行"] --> HID["💻 HID キー入力送信<br>Win+L or PIN入力"]  
  
    HID --> Wait["⌚ 待機"]  
    Wait -. "Lock: 3秒<br>Login: 15秒<br>0llama: 各 +2~3秒" .-> WaitNote[" "]  
  
    Wait --> Verify["🎥 callScreenVerify()<br>HTTP POST /api/screen/verify"]  
  
    Verify --> Capture["💻 API Server<br>スクリーンキャプチャ"]  
    Capture --> JPEG["📺 HDMI → canvas<br>→ base64 JPEG"]  
    JPEG --> VisionLLM["☁️ Vision LLM<br>画面を解析"]  
  
    VisionLLM --> Result{"判定結果"}  
  
    Result -- "LOCK_SCREEN" --> LockResult{"操作種別"}  
    Result -- "LOGIN_SUCCESS" --> LoginOK["✅ ログイン成功"]  
    Result -- "LOGIN_FAILED" --> LoginFail["✖️ ログイン失敗<br>+ Enter でエラー解除"]
```

```

Result -- "DESKTOP" --> DesktopResult{"操作種別"}

LockResult -- "ロック操作" --> LockOK["✅ ロック成功"]
LockResult -- "ログイン操作" --> LockWarn["⚠️ まだロック画面"]

DesktopResult -- "ロック操作" --> DesktopWarn["⚠️ ロック失敗<br>デスクトップのまま"]
DesktopResult -- "ログイン操作" --> DesktopOK["✅ ログイン成功"]

style Command fill:#f0f9e8,stroke:#7cb342
style HID fill:#fff3e0,stroke:#ff9800
style VisionLLM fill:#e8f4fd,stroke:#4a90d9
style Result fill:#fff8e1,stroke:#ffc107
style LockOK fill:#e8f5e9,stroke:#4caf50
style LoginOK fill:#e8f5e9,stroke:#4caf50
style DesktopOK fill:#e8f5e9,stroke:#4caf50
style LoginFail fill:#ffeb3b,stroke:#f44336
style LockWarn fill:#fff8e1,stroke:#ff9800
style DesktopWarn fill:#fff8e1,stroke:#ff9800
style WaitNote fill:none,stroke:none

```

## Vision プロンプト

Vision LLM には画面内容に応じた専用プロンプトを使用:

- ロック検証: "Is this a Windows lock screen or a desktop?" → LOCK\_SCREEN / DESKTOP
- ログイン検証: "What is the Windows login status?" → LOGIN\_SUCCESS / LOGIN\_FAILED / LOCK\_SCREEN
  - タスクバーの有無を重視（タスクバーが見える = デスクトップ = LOGIN\_SUCCESS）

## NanoKVM 操作の早期終了条件

Vision 検証の結果が以下のいずれかを含むとき、picoclaw は追加の LLM 呼び出しを行わず即座にユーザーに結果を返します:

結果	条件	動作
✅ 成功	応答に ✅ を含む	早期終了・成功として返却
✗ 失敗	応答に ✗ / LOGIN_FAILED を含む	早期終了・失敗として返却
⚠️ 未完了	応答に ⚠️ / LOCK_SCREEN を含む	早期終了・警告として返却

これにより、不要な反復 LLM 呼び出し（→ 429 エラー）を防止します。

## Vision LLM 未設定時の振る舞い

Vision LLM が設定されていない場合:

- 事後検証: Go 側で callScreenVerify() が呼ばれるが、API Server が「未設定」を返却
- ツール結果は「✅ 操作完了」で返される（検証なし）
- 初回操作時に設定案内メッセージが表示される

## Vision LLM 対応モデル一覧

プロバイダ	モデル	料金	速度	備考
Groq	meta-llama/llama-4-scout-17b-16e-instruct	無料	高速 (~500ms)	推奨: Llama 4 Scout ・ クレカ不要
Groq	meta-llama/llama-4-maverick-17b-128e-instruct	無料	高速	Llama 4 Maverick ・ 高精度
Groq	llama-3.2-11b-vision-preview	無料	高速	Llama 3.2 11B Vision

<b>Groq</b>	llama-3.2-90b-vision-preview	無料	低速	Llama 3.2 90B Vision・高精度
<b>GitHub Copilot</b>	gpt-4.1	無料	高速	推奨: gh 認証のみ・最新世代
<b>GitHub Copilot</b>	gpt-4o	無料	高速	高品質
<b>GitHub Copilot</b>	gpt-4.1-mini	無料	高速	最新世代・軽量
<b>GitHub Copilot</b>	Llama-3.2-11B-Vision-Instruct	無料	中速	オープンモデル
<b>GitHub Copilot</b>	Llama-3.2-90B-Vision-Instruct	無料	低速	高精度
<b>GitHub Copilot</b>	Phi-4-multimodal-instruct	無料	中速	Microsoft モデル
<b>Ollama</b>	moondream2:latest	無料	~60秒	1.7B・ローカル・CPU向き
<b>Ollama</b>	moondream:latest	無料	~60秒	Moondream v1
<b>Ollama</b>	llava:latest	無料	~3分	LLaVA 7B・高精度
<b>Ollama</b>	llava:7b	無料	~3分	LLaVA 7B 明示指定
<b>Ollama</b>	llava:13b	無料	~5分	LLaVA 13B・高精度・要GPU
<b>Ollama</b>	llava-llama3:latest	無料	~3分	LLaVA-Llama3・新世代
<b>Ollama</b>	bakllava:latest	無料	~3分	BakLLaVA・Mistral ベース
<b>OpenRouter</b>	google/gemini-2.0-flash-001	安価	高速	Gemini 2.0 Flash
<b>OpenRouter</b>	google/gemini-pro-1.5	安価	中速	Gemini Pro 1.5
<b>OpenRouter</b>	anthropic/clause-3.5-sonnet	高額	中速	Claude 3.5 Sonnet
<b>OpenRouter</b>	anthropic/clause-3-5-sonnet	高額	中速	Claude 3.5 Sonnet (旧ID)
<b>OpenRouter</b>	anthropic/clause-3-haiku	安価	高速	Claude 3 Haiku
<b>OpenRouter</b>	openai/gpt-4o	高額	中速	GPT-4o (OpenRouter 経由)
<b>OpenRouter</b>	openai/gpt-4o-mini	安価	高速	GPT-4o Mini (OpenRouter 経由)
<b>OpenAI</b>	gpt-4o-mini	安価	高速	推奨・低成本
<b>OpenAI</b>	gpt-4o	高額	中速	高精度
<b>OpenAI</b>	gpt-4-turbo	高額	低速	GPT-4 Turbo
<b>Anthropic</b>	claude-3-5-haiku-20241022	安価	高速	コスト効率
<b>Anthropic</b>	claude-3-5-sonnet-20241022	高額	中速	高精度
<b>Anthropic</b>	claude-3-opus-20240229	最高額	低速	最高精度
<b>Anthropic</b>	claude-3-sonnet-20240229	高額	中速	バランス

Anthropic	claude-3-haiku-20240307	安価	高速	旧世代・高速
-----------	-------------------------	----	----	--------

## タイムアウト設定

プロバイダ	検証遅延(ロック)	検証遅延(ログイン)	API タイムアウト
クラウド (Groq等)	3秒	15秒	30秒
Ollama (ローカル)	5秒	15秒	120秒

## Config 構造

```
{  
  "agents": {  
    "defaults": {  
      "provider": "github-copilot",  
      "model_name": "github-copilot/gpt-4.1",  
      "vision_provider": "github-copilot",  
      "vision_model": "gpt-4.1"  
    }  
  },  
  "providers": {  
    "github-copilot": { "api_key": "(gh auth token で自動取得)" }  
  },  
  "model_list": [  
    { "model_name": "github-copilot/gpt-4.1", "model": "gpt-4.1" }  
  ]  
}
```

**Note:** 設定フィールド `model` は `model_name` にリネームされました (v2026.02.25~)。旧 `model` フィールドは後方互換で引き続き読み込まれますが、新規設定では `model_name` を推奨します。

## レートリミット対策

Groq 無料枠 (TPM 6000) での運用を前提とした対策:

対策	説明
One-Shot 方式	セッション蓄積なし → トーカン消費を最小化
早期終了	✓ / ✗ / ⚠ で即座に返却 → 追加 LLM呼び出し不要
NanoKVM 回数制限	1メッセージあたり最大 4 回
REDACTED 拒否	LLM がパスワードを *** にマスクした場合は実行拒否
429 ポップアップ	レートリミット時にリトライ待ち時間を UI に表示

## 最新マージ変更履歴 (v2026.02.25)

### NanoKVM-USB (upstream 2コミット)

変更	内容
Right Shift キー修正	normalizeKeyCode() 関数追加。event.code が空の場合に event.key + event.location でフォールバック (browser版)

セキュリティ依存更新	minimatch, tar, ajv 等の脆弱性対応
------------	-----------------------------

## picoclaw (upstream 14コミット)

変更	内容
model → model_name リネーム	設定フィールド名を変更。GetmodelName() ヘルパーで旧フィールドも後方互換で読み込み
reasoning_content 対応	DeepSeek-R1, Moonshot kimi-k2.5 等の推論モデルが返す思考過程フィールドを保持。ツール呼び出しの往復で400エラーが発生する問題を修正
spawn ツール空タスク拒否	空文字列・空白のみのタスクを事前バリデーションで拒否。サブエージェントの無意味な起動を防止
DefaultConfig テンプレート漏れ防止	JSON Unmarshal 時にデフォルト値がユーザー設定に混入する問題を修正
Web プロキシ対応	tools.web.proxy 設定で HTTP プロキシ経由の Web 検索が可能に
GitHub Copilot セッション管理改善	SDK版: mutex 追加・Close() メソッド実装 (HTTP API版には影響なし)
デッドコード削除	Antigravity プロバイダ、WeChat 企業アプリ等の未使用コードを除去
nanokvm_screen_check 追加	画面状態をキャプチャ・Vision LLM で分析して返却する新ツール。早期終了対応
/api/screen/verify 拡張	action: "status" を追加。汎用プロンプトで画面全体を記述
NO_VIDEO 構造化レスポンス	映像なし時に HTTP 500 → 200 + status: "NO_VIDEO" に変更。「アプリ未起動」と「映像なし」を区別可能に
lock/login NO_VIDEO 警告	ロック・ログインの post-verify で映像がない場合、嘘の成功メッセージではなく ! 警告を返すように修正
CGO_ENABLED=0 静的ビルド	クロスプラットフォーム GLIBC エラーを防止
BLACK_SCREEN ステータス追加	NO_VIDEO と BLACK_SCREEN を区別。キャプチャカードが黒フレームを配信する場合 (スリープ/HDMI再取得中) に専用メッセージを表示
CaptureResult IPC 拡張	renderer → main の IPC に rejectReason を追加。キャプチャ失敗の理由をメインプロセスのログに記録
ロック画面誤認修正	Vision LLM が「no taskbar」と否定文で記述した場合のキーワード誤検出を修正。hasPositive() ヘルパーによる否定表現フィルタと LOCK_SCREEN 優先順位変更
track.muted 除去	HDMI 信号再取得中に一時的に muted=true になる問題で NO_VIDEO 誤判定が発生していたため除去
自動ウェイクジグル	BLACK_SCREEN 検出時にマウスジグル (1px右+1px左) を HID 送信してスリープ復帰を試行 → 2 秒待機 → 再キャプチャ。ネット移動量±0pxで操作中でも安全
セッション履歴自動修復	GetHistory() で sanitizeToolCallHistory() を呼び出し、assistant の tool_calls に対応する tool レスポンスが欠落している場合、その手前で履歴をトランケート。並列ツール呼び出し後のセッション破損による LLM API 400 エラーを自動防止