

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: 2026-02-27 (v2026.02.27)

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。

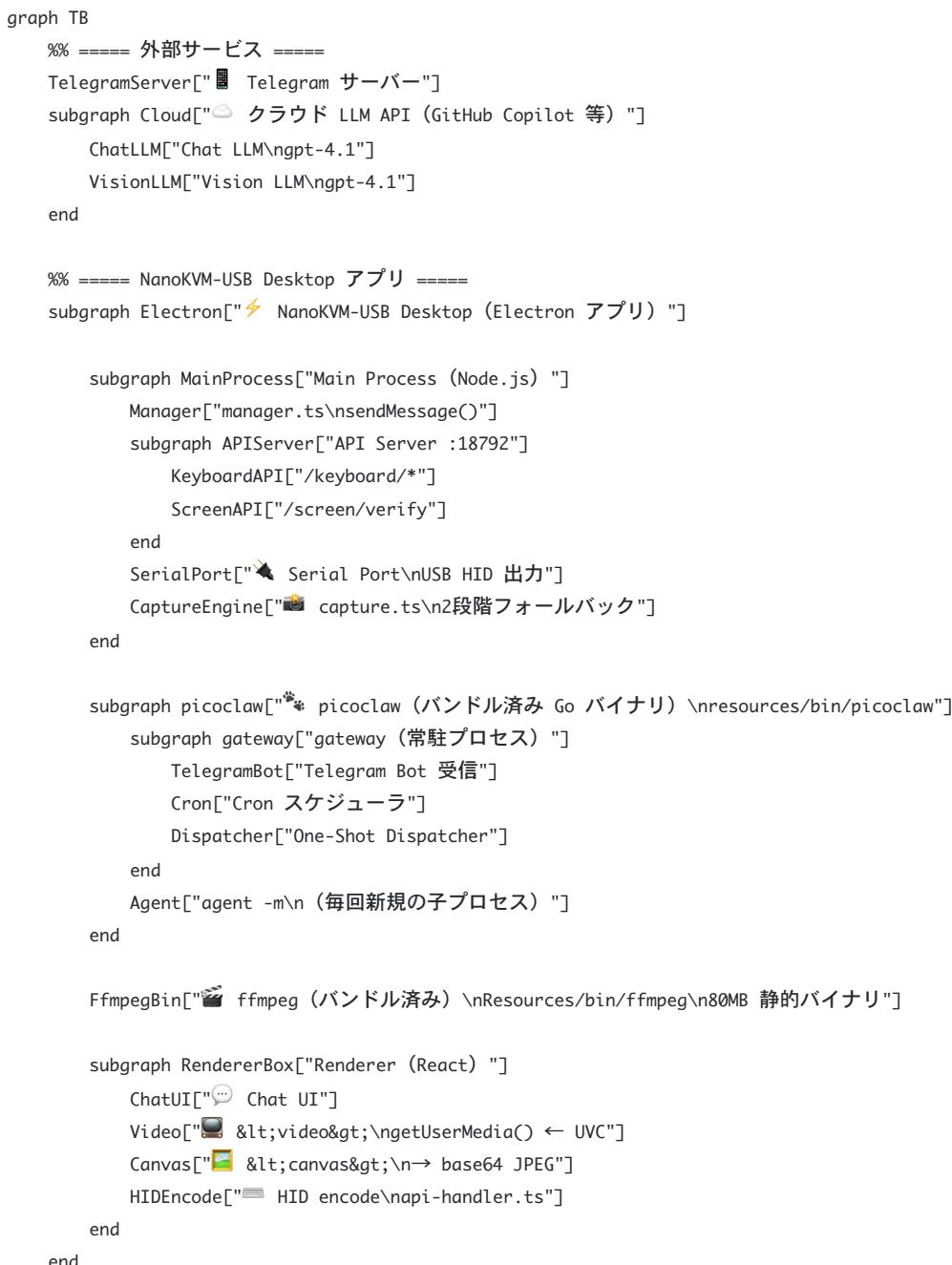
入力経路: チャット UI (アプリ内) または Telegram ボット

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



```

%% ===== ハードウェア =====
NanoKVM["NanoKVM-USB\n (Hardware) "]
WindowsPC["Windows PC\n (リモート対象) "]

%% ===== 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 -- "IPC" --> 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: ① 通常パス)
ScreenAPI --> CaptureEngine
CaptureEngine -- "① IPC\n通常パス" --> Canvas
Video -- "drawImage()" --> Canvas
Canvas -- "IPC\nbase64 JPEG" --> CaptureEngine

%% API Server → ffmpeg (映像キャプチャ系: ② フォールバック)
CaptureEngine -- "② spawn\nロック時フォールバック" --> FfmpegBin
FfmpegBin -- "base64\nJPEG" --> CaptureEngine

CaptureEngine -- "base64 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
NanoKVM -. "USB UVC\n (AVFoundation経由)" .-> FfmpegBin

%% スタイル
style Cloud fill:#e8f4fd,stroke:#4a90d9
style Electron fill:#fff3e0,stroke:#ff9800
style MainProcess fill:#fff8e1,stroke:#ffc107
style picoclaw fill:#f0f9e8,stroke:#7cb342
style gateway fill:#e8f5e9,stroke:#66bb6a
style RendererBox fill:#fce4ec,stroke:#e91e63
style APIServer fill:#e8f4fd,stroke:#4a90d9
style FfmpegBin fill:#e8eaf6,stroke:#5c6bc0
style CaptureEngine fill:#f3e5f5,stroke:#9c27b0
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 人物 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: Vision LLM が
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"

ChatUI ロック操作のシーケンス図

アプリ内蔵の ChatUI から同じロック操作を行う場合のシーケンスです。 Telegram 経由とは異なり、gateway を介さず Renderer → Main Process 間の IPC で直接 picoclaw を呼び出します。

```
sequenceDiagram
    actor User as 人物 ユーザー
    participant ChatUI as 話す Chat UI<br>(Renderer)
    participant Main as パソコン Main Process
    participant Agent as 犬 agent -m<br>(子プロセス)
    participant ChatLLM as 云う Chat LLM
    participant API as 雷 API Server<br>:18792
    participant Render as ディスプレイ Renderer
    participant KVM as キーボード NanoKVM
    participant WinPC as ノート Windows PC

    User->>ChatUI: 「ロックして」と入力
    ChatUI->>Main: IPC: PICOCLAW_SEND_MESSAGE<br>(message, language, sessionId)
    Main->>Agent: spawn picoclaw agent -m<br>"ロックして" -l ja -s chatId

    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: Vision LLM が<br>画面を解析
        ChatLLM-->API: "LOCK_SCREEN"
        API-->>Agent: 検証結果
    end

    Agent-->>Main: stdout: "✅ ロック完了"
    Note over Agent: プロセス終了
    Main->>Main: stripActionTags(output)
    Main-->>ChatUI: IPC 応答: { success, response }
    ChatUI->>User: 話す "✅ ロック完了"
```

Telegram と ChatUI の比較:

項目	Telegram	ChatUI
入口	Telegram Bot API → gateway (常駐)	Renderer IPC → Main Process
プロセス管理	gateway が spawn・監視	manager.ts が spawn・監視
セッション	Telegram Chat ID	chat-{timestamp}
レスポンス	Bot API で返信	IPC で ChatUI に直接返却
言語設定	picoclaw config	i18n.language を引数で渡す
Tool Call 後処理	gateway が interceptToolCallText	manager.ts が interceptToolCallText
エラー表示	Telegram メッセージ	ChatUI 赤バブル + レート制限ポップアップ

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 (通常)	API Server → capture.ts → ① Renderer IPC canvas キャプチャ → base64 JPEG → Vision LLM → 判定結果	高速パス (~50ms)
⌚ 検証 (ロック時)	API Server → capture.ts → ② ffmpeg spawn → AVFoundation → base64 JPEG → Vision LLM → 判定結果	フォールバック (~500ms)

バンドル済みコンポーネント（アプリ内蔵）：

コンポーネント	配置パス	サイズ	役割
picoclaw	resources/bin/picoclaw	~18MB	AI エージェント (Go 静的バイナリ)
ffmpeg	Resources/bin/ffmpeg	~80MB	ロック時キャプチャ (ffmpeg-static)
Electron/Chromium	Frameworks/	~254MB	アプリ基盤・Renderer・WebRTC

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	200ms
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	1,500ms
3	Backspace × 10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80ms間隔
5	Enter キー押下	PIN を送信してログイン	-

skipWake 最適化: picoclaw 経由のログインでは、画面確認 (screen_check) で既に PC が起動済みと確認されているため、S3 スリープ復帰シーケンス (5秒のマウス+キーボードウェイク) をスキップします。これにより、ログイン所要時間が約14秒から約5秒に短縮されました。

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 側のインターフェース (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

セキュリティ：Credential Vault（認証情報保管庫）

課題：現行方式のセキュリティリスク

現行のログイン操作では、ユーザーがチャットメッセージに PIN コードやパスワードを平文で記載します。これにより以下のリスクが存在します：

リスク	現行の状態	影響
チャット履歴への露出	Telegram 履歴・Chat UI に PIN/パスワードが平文で残る	第三者の覗き見、端末紛失時の漏洩
LLM プロバイダへの送信	メッセージ全体が Chat LLM に送信される	クラウドサーバーに認証情報が到達
ログファイルへの記録	インターフェース・API Server のログに記録される可能性	ディスク上に平文で残存

現行の緩和策: LLM がパスワードを *** にマスクした場合は実行を拒否する「REDACTED 拒否」機能は存在しますが、これは LLM が自発的にマスクしたケースのみを防ぐもので、LLM への送信自体は防止できません。

解決策: Credential Vault

暗号化された認証情報保管庫（Credential Vault）を導入し、Windows の PIN/パスワードがチャット経路を一切通らない設計にします。

設計原則

1. Windows 認証情報は Vault 内にのみ保存 — チャットメッセージに記載しない
2. マスターpasswordも LLM に送信しない — Vault 解錠時は LLM をバイパス
3. 毎回マスターpasswordを要求 — セッション保持なし (One-Shot と一致)
4. Telegram ではマスターpasswordメッセージを即時削除 — チャット履歴に残さない

ログイン操作フロー (Vault 有効時)

```
sequenceDiagram
    actor User as ユーザー
    participant Chat as Chat UI /<br>Telegram
    participant GW as gateway /<br>manager.ts
    participant Agent as agent -m
    participant LLM as Chat LLM
    participant Vault as Vault API<br>(Main Process)
    participant API as API Server
    participant KVM as NanoKVM

    User->>Chat: 「ログインして」
    Chat->>GW: メッセージ転送

    GW->>Agent: spawn agent -m
    Agent->>LLM: HTTPS (メッセージ送信)
    LLM->>Agent: Tool Call: nanokvm_login<br>(password 引数なし)

    Agent->>API: POST /api/keyboard/login<br>{"password": "@vault"}
    API->>Vault: Vault 解錠済み ?

    alt Vault 未解錠
        Vault-->>API: 未解錠
        API-->>Agent: {"error": "vault_locked",<br>"message": "マスターpasswordが必要です"}
        Agent-->>GW: "🔒 マスターpasswordを入力してください"
        GW-->>Chat: 表示
        GW->>GW: vault_pending_unlock = true

        User->>Chat: 「mymaster123」

        Note over GW: vault_pending_unlock = true<br>→ LLM に送信せず Vault へ直接渡す
        GW-->>Vault: POST /api/vault/unlock<br>{"master_password": "mymaster123"}

    alt Telegram の場合
        GW-->>Chat: deleteMessage (passwordメッセージを即時削除)
        Chat-->>User: (メッセージが消える)
    end

    alt Chat UI の場合
        Chat-->>Chat: 表示を 「🔒 ****」 にマスク
    end

    Vault-->>Vault: PBKDF2 で鍵導出<br>AES-256-GCM で復号
    Vault-->>GW: {"success": true}

    GW-->>GW: vault_pending_unlock = false
```

```

GW->>Agent: spawn agent -m (ログイン再実行)
Agent->>LLM: HTTPS
LLM-->>Agent: Tool Call: nanokvm_login
Agent->>API: POST /api/keyboard/login<br>{"password": "@vault"}
API->>Vault: 認証情報を取得
Vault-->>API: {"pin": "1234"}
end

API->>KVM: HID キー入力 (PIN)
Note over KVM: Windows にログイン
API-->>Agent: {"success": true}
Agent-->>GW: "✓ ログイン完了"
GW-->>Chat: 表示
Chat-->>User: "✓ ログイン完了"

```

LLM バイパスの仕組み

マスターpasswordの入力時にLLMを経由させない仕組みは以下のとおりです。

段階	メッセージ	LLM に送信	チャット表示	ログ記録
① ログイン要求	「ログインして」	✓	そのまま表示	✓
② マスターPW要求	(システム応答)	—	「🔒 マスターpasswordを入力してください」	—
③ マスターPW入力	「mymaster123」	✗ 送信しない	Chat UI: 「🔒 ****」 / Telegram: 即時削除	✗
④ ログイン実行	(自動)	✗	「✓ ログイン完了」	✓ (PINなし)

状態管理: gateway (Telegram) / manager.ts (Chat UI) に `vault_pending_unlock` フラグを追加。このフラグが `true` の間は、次のユーザー入力を agent spawn せず、直接 Vault API に渡します。

Telegram メッセージ即時削除

Telegram Bot API の `deleteMessage` を使用して、マスターpasswordを含むメッセージを受信直後に削除します。

```

POST https://api.telegram.org/bot<token>/deleteMessage
{
  "chat_id": <chat_id>,
  "message_id": <password_message_id>
}

```

- 削除はサーバー側で行われるため、全端末のチャット履歴から消える
- 削除後、ボットから「🔒 認証を受け付けました」と返信
- 一瞬表示されるが、履歴には残らない

ストレージ設計

```

~/.picoclaw/vault.enc ← AES-256-GCM 暗号化済みクレデンシャル
~/.picoclaw/vault.salt ← PBKDF2 用ソルト (32バイト)

```

項目	方式	理由
暗号化	AES-256-GCM	Node.js crypto 標準、認証付き暗号化（改竄検知付き）
鍵導出	PBKDF2-SHA256 (100,000 イテレーション)	マスターpasswordから 256bit 暗号化キーを導出
ソルト	crypto.randomBytes(32)	初回セットアップ時に生成、以後固定

マスターPW検証	GCM 認証タグ	復号失敗 = パスワード不一致（専用ハッシュ不要）
----------	----------	---------------------------

保存データ構造（暗号化前の平文 JSON）：

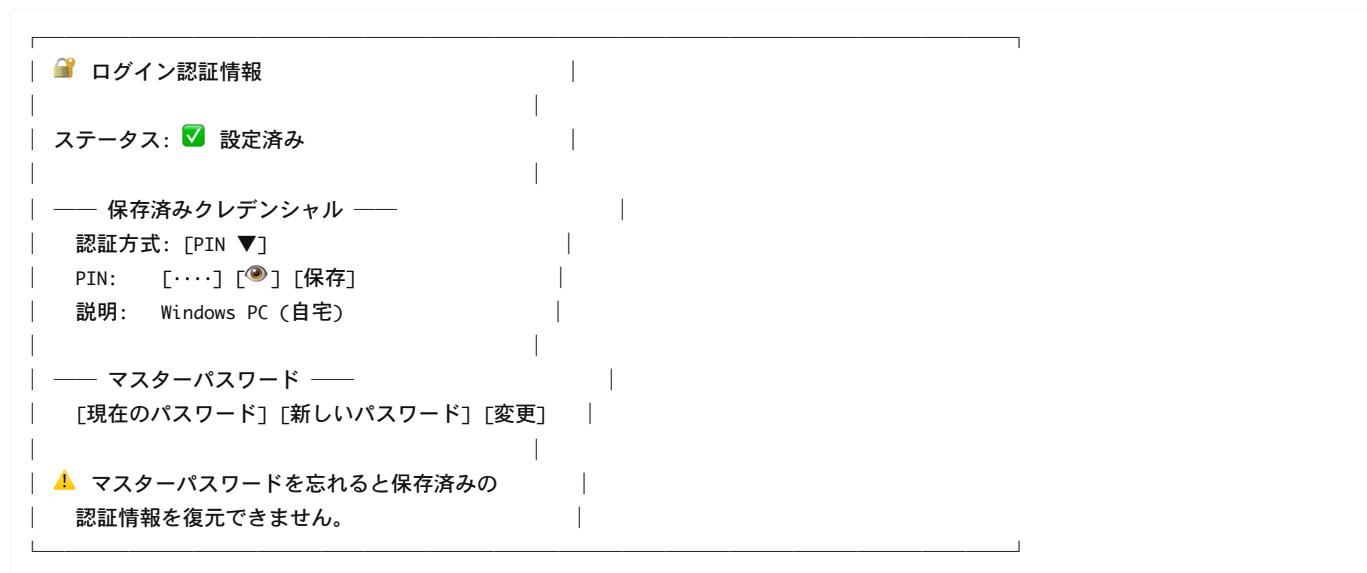
```
{
  "pin": "1234",
  "username": "",
  "description": "Windows PC"
}
```

Note: プロファイルは1つのみ。複数PCを操作する場合はSettings UIから切り替えます。

コンポーネント変更一覧

コンポーネント	ファイル	変更内容
Vault Manager	src/main/vault/manager.ts (新規)	AES-256-GCM 暗号化ストレージ管理
IPC Events	src/common/ipc-events.ts	VAULT_SETUP, VAULT_UNLOCK, VAULT_SAVE, VAULT_STATUS 追加
API Server	src/main/api/server.ts	POST /api/vault/unlock, GET /api/vault/credential 追加
API Server (login)	src/main/api/server.ts	password: "@vault" の場合に Vault から自動取得
Settings UI	src/renderer/src/components/menu/settings/picoclaw.tsx	「 ログイン認証情報」セクション追加
manager.ts	src/main/picoclaw/manager.ts	vault_pending_unlock 状態管理、LLM バイパス
picoclaw (Go)	gateway 側	vault_pending_unlock 状態管理、deleteMessage呼び出し

Settings UI（設定画面）



セキュリティ比較

脅威	現行	Vault 導入後
チャット履歴に Windows PIN が露出	✗ 丸見え	✓ 一切表示されない
LLM プロバイダに PIN が送信される	✗ 送信される	✓ 送信されない

LLM プロバイダにマスター PW が送信される	—	<input checked="" type="checkbox"/> 送信されない (LLM バイパス)
ログファイルに PIN が記録される	<input checked="" type="checkbox"/> 記録される可能性	<input checked="" type="checkbox"/> @vault のみ記録
Telegram 履歴にマスター PW が残る	—	<input checked="" type="checkbox"/> 即時削除 (deleteMessage)
NanoKVM アプリへの不正アクセス	<input checked="" type="checkbox"/> PIN なしでログイン可能	<input checked="" type="checkbox"/> マスター PW が必要
ディスク上の保存データ	—	<input checked="" type="checkbox"/> AES-256-GCM で暗号化

後方互換性

Vault 未設定時は従来どおり 「PIN xxxx でログインして」 でのログインも引き続き動作します。 Vault が設定されている場合のみ、PIN/パスワードなしの 「ログインして」 で Vault 経由のフローが有効になります。

状態	「ログインして」	「PIN 1234 でログインして」
Vault 未設定	LLM が PIN を質問	そのまま実行 (従来動作)
Vault 設定済み・未解錠	マスター PW を要求	Vault を優先 (メッセージ中の PIN は無視)
Vault 設定済み・解錠済み	Vault から PIN 取得して実行	Vault を優先

ffmpeg バンドルとクロスプラットフォーム対応

画面キャプチャ機能 (macOS ロック中のフォールバック) で使用する ffmpeg は、`ffmpeg-static` npm パッケージを通じてアプリにバンドルされます。

バンドル方式

項目	内容
パッケージ	<code>ffmpeg-static v5.3.0</code> (静的リンク済みバイナリ)
配置場所	<code>electron-builder.yml</code> の <code>extraResources</code> で <code><app>/Contents/Resources/bin/ffmpeg</code> にコピー
バイナリサイズ	約 70~80MB (プラットフォームにより異なる)
検索優先順位	① バンドル済み → ② <code>/usr/local/bin/ffmpeg</code> → ③ <code>/opt/homebrew/bin/ffmpeg</code> → ④ <code>/usr/bin/ffmpeg</code>

プラットフォーム対応

プラットフォーム	キャプチャ方式	デバイス検出	状態
macOS (Intel)	AVFoundation	[index] Device Name	<input checked="" type="checkbox"/> 動作確認済み
macOS (Apple Silicon)	AVFoundation	同上	<input checked="" type="checkbox"/> 対応 (ffmpeg-static が arm64 バイナリを提供)
Windows	DirectShow	"Device Name"	実装済み (未テスト)
Linux	—	—	<input checked="" type="checkbox"/> 未対応

Windows 対応の実装詳細

Windows では ffmpeg の DirectShow 入力を使用して USB キャプチャカードから映像を取得します:

```
ffmpeg -f dshow -video_size 1920x1080 -i "video=USB3 Video" -frames:v 1 -f image2pipe -vcodec mjpeg -q:v 5 -
```

- デバイス検出: `ffmpeg -f dshow -list_devices true -i dummy` でデバイス名を取得
- 入力指定: `video=<デバイス名>` 形式 (AVFoundation のインデックス指定とは異なる)
- バイナリ名: `ffmpeg.exe` (`ffmpeg-static` がビルドプラットフォームに応じて自動選択)

Note: Windows では macOS ロック時のような Renderer スロットリング問題は通常発生しませんが、リモートデスクトップ切断時等に同様の状況が起こる可能性があるため、ffmpeg フォールバックを共通基盤として実装しています。

ffmpeg キャプチャ実装アーキテクチャ

2段階フォールバック設計

画面キャプチャは **Renderer IPC (高速パス)** と **ffmpeg ネイティブキャプチャ (フォールバック)** の2段階で動作します。

```
flowchart TD
    Request["💻 API Server\nrequestScreenCapture()"] --> Step1["① Renderer IPC\ngetUserMedia() → canvas → base64"]
    Step1 --> Check1{"成功?"}
    Check1 -- "✓ dataUrl あり" --> Return1["📸 キャプチャ結果を返却"]
    Check1 -- "✗ null / timeout" --> CheckFfmpeg{"ffmpeg\n利用可能?"}

    CheckFfmpeg -- "✓ isFfmpegCaptureAvailable()" --> Step2["② ffmpeg ネイティブ\ncaptureFrameNative()"]
    CheckFfmpeg -- "✗ 未インストール" --> ReturnFail["✗ キャプチャ失敗"]

    Step2 --> Check2{"成功?"}
    Check2 -- "✓ dataUrl あり" --> Return2["📸 キャプチャ結果を返却"]
    Check2 -- "✗ エラー" --> ReturnBoth["✗ 両方失敗\nrenderer: 理由; ffmpeg: 理由"]

    style Request fill:#fff3e0,stroke:#ff9800
    style Step1 fill:#e8f4fd,stroke:#4a90d9
    style Step2 fill:#f0f9e8,stroke:#7cb342
    style Return1 fill:#e8f5e9,stroke:#4caf50
    style Return2 fill:#e8f5e9,stroke:#4caf50
    style ReturnFail fill:#ffeb3b,stroke:#f44336
    style ReturnBoth fill:#ffeb3b,stroke:#f44336
```

通常利用時（ロック解除状態）とロック時の動作

状態	映像表示	キャプチャ方式	ffmpeg の関与
通常利用（ロック解除）	getUserMedia() WebRTC	Renderer IPC (canvas → base64 JPEG)	呼ばれない
macOS ロック中	getUserMedia() はスロットリングで凍結	Renderer IPC タイムアウト → ffmpeg フォールバック	単一フレーム取得
リモートデスクトップ切断	同上の可能性	同上	単一フレーム取得

重要: ffmpeg は連続ストリーミングには使用しません。 -frames:v 1 オプションにより1フレームだけ取得して即座にプロセス終了します。通常の映像表示（<video> タグ + WebRTC）には一切影響しません。

同時アクセスの安全性

懸念	実態	理由
ffmpeg が映像品質を劣化させる？	影響なし	通常時は ffmpeg が呼ばれないため
ロック中に同時アクセスで問題？	影響なし	macOS AVFoundation は複数プロセスからの同時読み取りを許容
ffmpeg プロセスがリソースを占有？	極めて軽微	1フレーム < 1秒で完了、5秒のセーフティタイムアウトあり

Windows で getUserMedia と ffmpeg が競合？	可能性あり	DirectShow は排他的ロックの場合あり。ただし「Resource busy」エラーで graceful に処理
---	-------	---

ffmpeg キャプチャパイプライン

capture.ts の実装詳細:

```

captureFrameNative(width, height, quality)
|
|--- findFfmpeg()
|   |--- getBundledFfmpegPath() ... Packaged: Resources/bin/ffmpeg, Dev: require('ffmpeg-static')
|
|--- detectCaptureDevice()
|   |--- [macOS] detectCaptureDeviceMac() ... AVFoundation デバイス一覧 → インデックス返却
|   |--- [Windows] detectCaptureDeviceWin() ... DirectShow デバイス一覧 → デバイス名返却
|
|--- ffmpeg spawn ... プラットフォーム別の引数でプロセス起動
|   |--- [macOS] -f avfoundation -framerate 30 -video_size WxH -i <index>
|   |--- [Windows] -f dshow -video_size WxH -i "video=<name>" ...
|   |--- 共通: -frames:v 1 -f image2pipe -vcodec mjpeg -q:v <quality> -
|
|--- stdout → Buffer[] → Buffer.concat ... JPEG バイナリを収集
|
|--- base64 変換 → data:image/jpeg;base64,... ... data URL 生成
|
|--- 黒画面検出 ... JPEG < 2KB の場合に警告ログ (呼び出し元に判断委任)

```

デバイス検出パターン

USB キャプチャカードの自動検出に使用するデバイス名パターン:

パターン	対象デバイス	除外条件
USB3 Video	NanoKVM-USB 標準	—
USB2.0 HD UVC	汎用UVCキャプチャカード	—
nanokvm (大文字小文字不問)	NanoKVM ブランド全般	—
capture (大文字小文字不問)	汎用キャプチャデバイス	screen を含む場合は除外

キャッシング機構

キャッシング対象	変数名	初期値	クリア条件
ffmpeg バイナリパス	cachedFfmpegPath	null	アプリ再起動時のみ
AVFoundation デバイスインデックス	cachedDeviceIndex	null	resetCaptureCache() またはデバイス未検出エラー時
DirectShow デバイス名	cachedDshowDeviceName	null	resetCaptureCache() またはデバイス未検出エラー時

エラーハンドリング

エラー	発生条件	対応
ffmpeg not available	ffmpeg バイナリが見つからない	フォールバック不可、Renderer IPC のみに依存
no USB capture device detected	USB キャプチャカード未接続	null 返却、呼び出し元で NO_VIDEO 処理
capture device not found	検出済みデバイスが切断された	キャッシングクリア → 次回再検出

capture device busy	他プロセスが排他的に使用中	null返却（Windows DirectShowで発生可能性）
ffmpeg capture timed out (5s)	ffmpegが5秒以内に完了しない	SIGKILLで強制終了
ffmpeg output too small	出力が100バイト未満	不正な出力として拒否

実装ファイル

ファイル	役割
src/main/device/capture.ts	ffmpegネイティブキャプチャの全実装（findFfmpeg, detectCaptureDevice, captureFrameNative, resetCaptureCache）
src/main/api/server.ts → requestScreenCapture()	2段階フォールバック制御（Renderer IPC → ffmpeg）
src/main/api/server.ts → requestScreenCaptureViaIpc()	Renderer IPC キャプチャ（5秒タイムアウト）
src/renderer/src/libs/media/camera.ts	Renderer側getUserMedia() → <video> → <canvas> → base64

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: マウス左クリック+キーボードSpaceキーをHID送信してスリープ復帰を試行→4秒待機→再キャプチャ（最大2回リトライ）
- 映像取得成功ならVisionLLM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（「マウスクリックとキーボード入力でスリープ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

ウェイク操作の設計: マウス左クリック（press+release）とキーボードSpaceキー（press+release）の両入力を送信。S3スリープからの復帰はキーボード入力のみに応答するPCもあるため、両方を併用。4秒の待機時間はS3レジューム+HDMI信号安定に必要な時間を考慮。最大2回までリトライ。

レスポンス例

ステータス	アイコン	例
DESKTOP	💻	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。
LOCK_SCREEN	🔒	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。
LOGIN_SCREEN	🔑	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。

DESCRIBED		画面状態: (Vision LLM の記述)
NO_VIDEO		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。
BLACK_SCREEN		画面が真っ黒です。マウスクリックとキーボード入力でスリープ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。
NO_SIGNAL		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
映像なし (screen_check)	PC 未接続 / ストリーミング未開始	success=false, status="NO_VIDEO"	映像がありません。PCが NanoKVM-USB に接続されています...
映像なし (lock)	同上	v.Status == "NO_VIDEO"	⚠️ 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...
映像なし (login)	同上	v.Status == "NO_VIDEO"	⚠️ 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...
黒画面 (screen_check)	PC スリープ / HDMI 信号未安定	success=false, status="BLACK_SCREEN"	画面が真っ黒です。マウスクリック+キーボード入力で復帰試行済み...
黒画面 (lock)	同上	v.Status == "BLACK_SCREEN"	⚠️ 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)
並列 tool レスポンス欠落	sanitizeHistoryForProvider() が連続 tool メッセージを削除	predecessor チェックを role=="tool" にも拡張	(透過的修正: ユーザー影響なし)

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
manager.ts → cancelGitHubAuth()	認証プロセスを kill
picoclaw.ts (events)	IPC ハンドラ (INITIATE_GITHUB_AUTH, CANCEL_GITHUB_AUTH)
picoclaw.tsx (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
gpt-4o-mini	Chat	✓	高速・軽量
gpt-4o	Chat	✓	高品質
gpt-4.1	Chat	✓	推奨: 最新世代・高品質
gpt-4.1-mini	Chat	✓	最新世代・軽量
gpt-4.1-nano	Chat	-	超軽量
o1	Reasoning	-	推論特化
o3	Reasoning	-	推論特化
o3-mini	Reasoning	-	推論特化・軽量
o4-mini	Reasoning	-	最新推論モデル

Meta-Llama-3.1-405B-Instruct	Chat	-	大型オープン
Meta-Llama-3.1-8B-Instruct	Chat	-	軽量オープン
Llama-3.2-11B-Vision-Instruct	Chat	✓	Vision対応: 画面検証に使用可能
Llama-3.2-90B-Vision-Instruct	Chat	✓	Vision対応: 高精度
Phi-4	Chat	-	Microsoft 軽量
Phi-4-multimodal-instruct	Chat	✓	Vision対応: Microsoft
DeepSeek-R1	Reasoning	-	推論特化
MAI-DS-R1	Reasoning	-	Microsoft + DeepSeek
Mistral-large-2407	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。

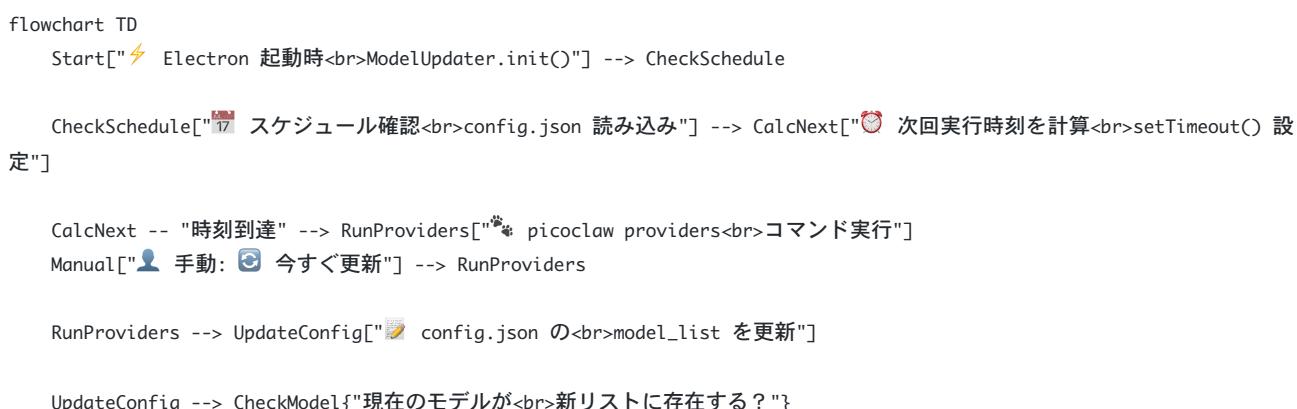
概要

- 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	0:00
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	有効

更新フロー



```

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>Ollama: 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:#ffebee,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）

多言語レスポンス対応: Vision LLM のプロンプトには、picoclaw 設定の language フィールドに応じた言語指示が自動付加されます。例: language: "ja" の場合「日本語で回答してください」が追加され、画面状態の説明が日本語で返されます。対応言語: ja, zh, ko, de ほか。

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
Groq	llama-3.2-90b-vision-preview	無料	低速	Llama 3.2 90B Vision · 高精度
GitHub Copilot	gpt-4.1	無料	高速	推奨: gh 認証のみ · 最新世代
GitHub Copilot	gpt-4o	無料	高速	高品質
GitHub Copilot	gpt-4.1-mini	無料	高速	最新世代 · 軽量
GitHub Copilot	Llama-3.2-11B-Vision-Instruct	無料	中速	オープンモデル
GitHub Copilot	Llama-3.2-90B-Vision-Instruct	無料	低速	高精度
GitHub Copilot	Phi-4-multimodal-instruct	無料	中速	Microsoft モデル

Ollama	moondream2:latest	無料	~60秒	1.7B・ローカル・CPU向き
Ollama	moondream:latest	無料	~60秒	Moondream v1
Ollama	llava:latest	無料	~3分	LLaVA 7B・高精度
Ollama	llava:7b	無料	~3分	LLaVA 7B 明示指定
Ollama	llava:13b	無料	~5分	LLaVA 13B・高精度・要GPU
Ollama	llava-llama3:latest	無料	~3分	LLaVA-Llama3・新世代
Ollama	bakllava:latest	無料	~3分	BakLLaVA・Mistral ベース
OpenRouter	google/gemini-2.0-flash-001	安価	高速	Gemini 2.0 Flash
OpenRouter	google/gemini-pro-1.5	安価	中速	Gemini Pro 1.5
OpenRouter	anthropic/clause-3.5-sonnet	高額	中速	Claude 3.5 Sonnet
OpenRouter	anthropic/clause-3-5-sonnet	高額	中速	Claude 3.5 Sonnet (旧ID)
OpenRouter	anthropic/clause-3-haiku	安価	高速	Claude 3 Haiku
OpenRouter	openai/gpt-4o	高額	中速	GPT-4o (OpenRouter 経由)
OpenRouter	openai/gpt-4o-mini	安価	高速	GPT-4o Mini (OpenRouter 経由)
OpenAI	gpt-4o-mini	安価	高速	推奨・低コスト
OpenAI	gpt-4o	高額	中速	高精度
OpenAI	gpt-4-turbo	高額	低速	GPT-4 Turbo
Anthropic	claude-3-5-haiku-20241022	安価	高速	コスト効率
Anthropic	claude-3-5-sonnet-20241022	高額	中速	高精度
Anthropic	claude-3-opus-20240229	最高額	低速	最高精度
Anthropic	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 → トークン消費を最小化
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。Get modelName() 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	BLACK_SCREEN 検出時にマウスクリック+キーボード Space キーを HID 送信してスリープ復帰を試行 → 4秒待機 → 再キャプチャ (最大2回リトライ)。S3 スリープからの復帰にはキーボード入力が必要な PC にも対応
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
並列 tool_calls プロバイダセイタイザ修正	sanitizeHistoryForProvider() が連続する tool レスポンス (並列 tool_calls 由来) の 2 件目以降を誤って削除していた問題を修正。predecessor チェックを role=="assistant" のみから role=="tool" も許容するよう拡張。10 件のユニットテスト追加

picoclaw (upstream 44コミットマージ v2026.02.26)

変更	内容
正規表現プリコンパイル	ツール出力パーサ等でのランタイム regexp.MustCompile をパッケージレベル変数に移行。hot path でのアロケーション削減 (mattn)
システムプロンプトキャッシュ	BuildSystemPromptWithCache() 追加。ワークスペースファイル変更時の再構築 (mtime チェック)。issue #607 修正
動的コンテキスト分離	時刻・Runtime・ツールリスト・言語設定を buildDynamicContext() に移動。キャッシュ可能な静的部分 (Identity・Skills・Memory) と分離
per-model request_timeout	プロバイダ設定に request_timeout フィールド追加。モデルごとの API タイムアウト設定が可能に
dm_scope デフォルト変更	global → per-channel-peer に変更。DM スコープがチャネル+ピアごとに分離
Cobra CLI リファクタ	upstream が spf13/cobra ベースに移行。フォーク側は既存 switch/case CLI を維持 (cmd_*.go 復元)

後方探索サニタイザー	sanitizeHistoryForProvider() の tool レஸポンス検証を後方探索に変更。並列 tool_calls の複数レスポンスをより堅牢に処理
golangci-lint ルール追加	errorlint, gocritic, revive 等 10+ ルール追加。コード品質向上
冗長ツール説明削除	システムプロンプトからツール定義の重複記述を除去。トークン消費削減
HTTP リトライユーティリティ	pkg/utils/http_retry.go 追加。HTTP リクエストの自動リトライ（指數バックオフ）
connect_mode 設定	プロバイダ設定に connect_mode フィールド追加
ウェイク方式改善	黒画面検出時のウェイク操作をマウスジグル（±1px）からマウスクリック+キーボード Space に変更。S3 スリープ対応、待機時間 2秒→4秒、最大2回リトライ

NanoKVM-USB Desktop (v2026.02.27)

変更	内容
ログイン速度最適化 (skipWake)	picoclaw 経由のログインで S3 ウェイクシーケンスをスキップ。PIN 待機 3000→1500ms、Backspace 20→10回、文字入力遅延 150→80ms に短縮。結果: 14秒 → 5秒
Vision 言語対応	~/.picoclaw/config.json の language フィールドを読み取り、Vision LLM プロンプトに言語指示を付加。日本語・中国語・韓国語・ドイツ語等に対応
ffmpeg バンドル	ffmpeg-static npm パッケージを導入し、アプリ内にffmpegバイナリをバンドル。別のマシンへのコピーでもffmpegなしでロック中キャプチャが動作
Windows ffmpeg 対応	capture.ts を macOS AVFoundation + Windows DirectShow のクロスプラットフォーム設計にリファクタ。デバイス検出・キャプチャの両方を抽象化
ffmpeg キャプチャ仕様書追加	2段階フォールバック設計（Renderer IPC → ffmpeg）、キャプチャパイプライン、デバイス検出パターン、キャッシュ機構、エラーハンドリング、同時アクセス安全性の全仕様を文書化
全体アーキテクチャ図更新	picoclaw・ffmpeg をアプリ内蔵バンドルとして Electron サブグラフ内に移動。Main Process / CaptureEngine ブロック追加。バンドル済みコンポーネント表とサイズ情報を追加
Telegram シーケンス図修正	Vision LLM の記述を特定モデル名（Llama 4 Scout）から汎用的な「Vision LLM」に変更
ChatUI シーケンス図追加	アプリ内蔵 ChatUI からのロック操作シーケンス図を追加。Telegramとの比較表付き
Mermaid 日本語フォント修正	SVG/PDF の Mermaid 図で日本語が文字化けする問題を修正。mermaid.css + mermaid-config.json で Hiragino Sans 等の日本語フォントを指定
Credential Vault 設計書追加	セキュリティセクション新設。AES-256-GCM 暗号化 Vault による PIN/パスワード保管、マスターパスワードの LLM バイパス、Telegram メッセージ即時削除、Chat UI マスク表示の設計を文書化