- Openloop SDK
    - ドキュメント
- Openloop SDK
    - 仕様書
- アーキテクチャ概要
    - Openloop SDK とは
    - レイヤー構成
    - セキュリティモデル
    - 対応チェーン
    - Transport 接続方式
    - npm パッケージ一覧
    - APDU プロトコル概要
    - 次のステップ
- クイックスタート
    - インストール
    - 基本フロー
    - Ethereum の完全サンプル
    - Bitcoin の完全サンプル
    - Bluetooth 接続
    - LocalWS 接続（全ブラウザ対応）
    - React での利用
    - 自動再接続
    - 次のステップ
- Transport ガイド
    - プラットフォーム対応表
    - 選択フローチャート
    - 各 Transport 詳細
    - OpenloopSDK クラスによる自動検出
    - 再接続パターン
    - 次のステップ
- Bitcoin API
    - インポート
    - 初期化
    - メソッド一覧
    - getAddress
    - getAddressWithPath
    - getAccountXpub
    - signMessage

    - signPsbt
    - signPsbtHex
    - signPsbtWithPaths
    - PSBT プロトコル詳細
    - bitcoinjs-lib 連携
    - 次のステップ
- Ethereum API
    - インポート
    - 初期化
    - メソッド一覧
    - getAddress
    - signPersonalMessage
    - signTransaction
    - signTypedData
    - signAuthorization
    - ethers.js 連携
    - viem 連携
    - 次のステップ
- XRP API
    - Ethereum との主な違い
    - インポート
    - 初期化
    - メソッド一覧
    - getAddress
    - signTransaction
    - xrpl.js 連携
    - 次のステップ
- Solana API
    - インポート
    - 初期化
    - メソッド一覧
    - getAddress
    - signTransaction
    - signOffchainMessage
    - @solana/web3.js 連携
    - 次のステップ
- TRON API
    - Ethereum との主な違い
    - インポート

    - 初期化
    - メソッド一覧
    - getAddress
    - signTransaction
    - signMessage
    - TronWeb 連携
    - 次のステップ
- WalletConnect 連携
    - 2 つの方式
    - WcTransport（APDU 中継方式）
    - AppKit（高レベル WC 方式）
    - WcTransport vs AppKit の使い分け
    - 制限事項
    - 次のステップ
- エラー処理ガイド
    - エラークラス階層
    - ApduError
    - TransportError
    - エラーハンドリングパターン
    - 次のステップ
- API リファレンス
    - 目次
    - インターフェース
    - App クラス
    - エラークラス
    - OpenloopSDK クラス
    - WcTransport クラス
    - 定数
    - ユーティリティ関数

Openloop SDK

ドキュメント

バージョン: 0.1.0

ハードウェアウォレット SDK for Web & Node.js

Ethereum | Bitcoin | Solana | TRON | XRP

Openloop

Openloop SDK
仕様書

バージョン v0.0.1 | 最終更新 2026-03-13
株式会社ハウディ・クリプト　代表取締役社長　浅田 一憲

アーキテクチャ概要

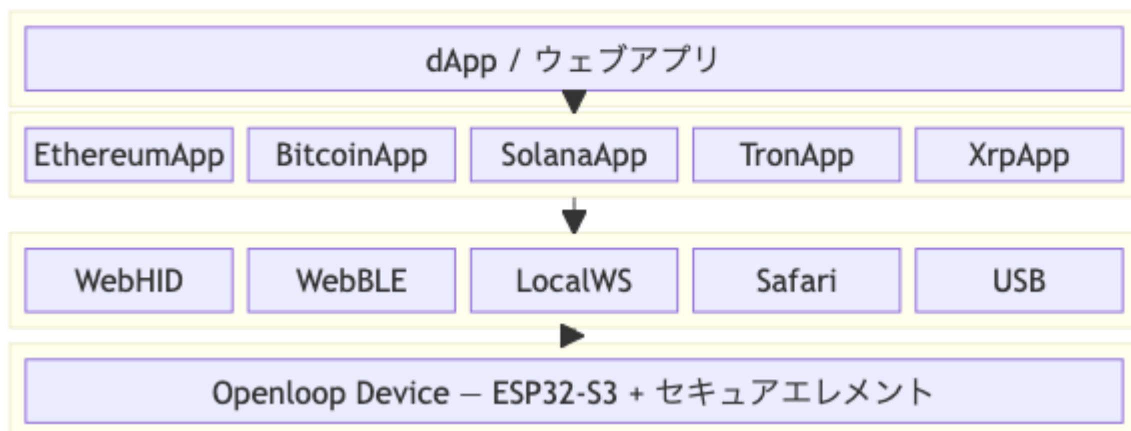
Openloop SDK とは

Openloop SDK は、Openloop ハードウェアウォレットとの通信を行う TypeScript ライブラリです。

> **ライセンスについて**: Openloop SDK の使用には、株式会社ハウディ・クリプトとのライセンス契約が必要です。ご要望に応じて SDK のカスタマイズも可能ですので、お気軽に info@haudi.co.jp までお問い合わせください。

最も重要な原則: 秘密鍵はデバイス内に保持され、外部に出ることはありません。SDK は APDU（Application Protocol Data Unit）コマンドの送受信のみを行います。署名はすべてデバイス内のセキュアエレメントで実行されます。

レイヤー構成



各レイヤーの役割

レイヤー	役割
dApp	ユーザーインターフェース、トランザクション構築
App Class	チェーン固有の APDU コマンド構築・レスポンス解析
ITransport	デバイスとの物理的通信（USB/BLE/WebSocket等）
Device	秘密鍵管理、署名実行、ユーザー承認

セキュリティモデル

SDK が行うこと

- APDU コマンドの構築と送信
- レスポンスの解析（アドレス、署名等）
- Transport の接続・切断管理

SDK が行わないこと

- 秘密鍵の生成・保持・操作
- 署名の実行
- シードフレーズの管理

デバイス側のセキュリティ

- 秘密鍵はデバイス内に生成・保存され、外部に出ることはない
- すべての署名はデバイス内で実行される
- トランザクション内容はデバイスの画面に表示され、ユーザーが物理ボタンで承認する
- ユーザーが拒否した場合、デバイスはステータスワード `0x6985` を返し、署名は実行されない

対応チェーン

チェーン	App クラス	楕円曲線	BIP44 coin_type
Ethereum (EVM)	`EthereumApp`	secp256k1	`60'`
Bitcoin	`BitcoinApp`	secp256k1	`0'` (mainnet), `1'` (testnet)
Solana	`SolanaApp`	Ed25519	`501'`
TRON	`TronApp`	secp256k1	`195'`
XRP Ledger	`XrpApp`	Ed25519 / secp256k1	`144'`

EVM 互換チェーン

`EthereumApp` は Ethereum だけでなく、EVM 互換チェーンで使用できます。`signPersonalMessage` と `signTypedData` の `chainId` パラメータでチェーンを指定します。

Transport 接続方式

SDK は 6 種類の接続方式を提供し、幅広いプラットフォームをカバーします。

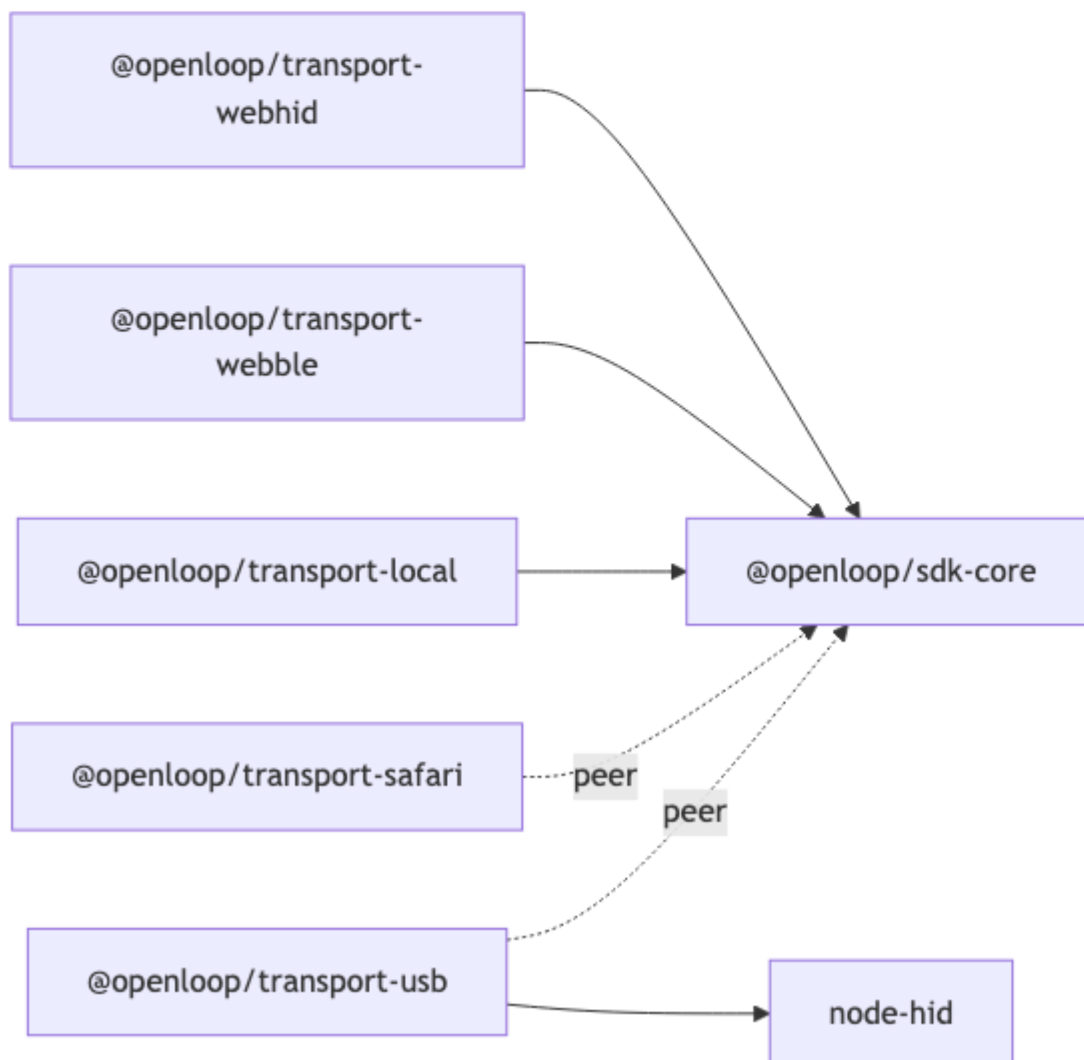
Transport	パッケージ	ユースケース
WebHID (USB)	`@openloop/transport-webhid`	Chrome/Edge デスクトップ
Web Bluetooth	`@openloop/transport-webble`	Chrome/Edge + Android
LocalWS	`@openloop/transport-local`	全ブラウザ（Connect 経由）
Safari Extension	`@openloop/transport-safari`	iOS Safari
USB HID (native)	`@openloop/transport-usb`	Node.js / Electron
WalletConnect	`@openloop/sdk-core` 内蔵	全ブラウザ（リモート署名）

詳細は Transport 選択ガイド を参照してください。

npm パッケージ一覧

パッケージ	説明	依存関係
`@openloop/sdk-core`	コアライブラリ（App, Types, Errors, Constants, WcTransport）	なし
`@openloop/transport-webhid`	WebHID (USB) Transport	sdk-core
`@openloop/transport-webble`	Web Bluetooth Transport	sdk-core
`@openloop/transport-local`	LocalWS Transport	sdk-core
`@openloop/transport-safari`	Safari Extension Transport	sdk-core (peer)
`@openloop/transport-usb`	Native USB HID Transport	sdk-core (peer), node-hid

依存関係の図



`sdk-core` はブラウザ API やネイティブモジュールに依存しない純粋な TypeScript パッケージです。各 Transport パッケージがプラットフォーム固有の実装を提供します。

APDU プロトコル概要

SDK とデバイス間の通信は APDU（Application Protocol Data Unit）コマンドで行われます。

コマンド構造

```
[CLA] [INS] [P1] [P2] [Lc] [Data...]
```

フィールド	サイズ	説明
CLA	1 byte	クラスバイト（`0xE0`: Ledger 互換, `0xF0`: Openloop 独自）
INS	1 byte	命令コード
P1	1 byte	パラメータ 1（チャンキング制御等）
P2	1 byte	パラメータ 2（曲線選択等）
Lc	1 byte	データ長
Data	Lc bytes	コマンドデータ

レスポンス構造

```
[Data...] [SW1] [SW2]
```

フィールド	サイズ	説明
Data	可変	レスポンスデータ
SW1-SW2	2 bytes	ステータスワード（`0x9000` = 成功）

CLA バイト

CLA	用途
`0xE0`	Ledger 互換コマンド（ETH, BTC getAddress, SOL, TRON, XRP）
`0xF0`	Openloop 独自コマンド（chain_id 対応署名, PSBT, BTC xpub）

詳細な APDU ステータスワードは エラー処理 を参照してください。

次のステップ

- クイックスタート — 開発者向け導入ガイド
- Transport 選択ガイド — プラットフォーム対応表
- API リファレンス — 全型定義・インターフェース一覧

クイックスタート

Openloop SDK を使ってハードウェアウォレットに接続し、アドレス取得やトランザクション署名を行う手順を説明します。

インストール

コアパッケージと、使用する接続方式の Transport パッケージをインストールします。

```
# USB 接続（Chrome/Edge デスクトップ）
npm install @openloop/sdk-core @openloop/transport-webhid

# Bluetooth 接続（Chrome/Edge + Android）
npm install @openloop/sdk-core @openloop/transport-webble

# 全ブラウザ対応（Openloop Connect 経由）
npm install @openloop/sdk-core @openloop/transport-local

# iOS Safari
npm install @openloop/sdk-core @openloop/transport-safari

# Node.js / Electron
npm install @openloop/sdk-core @openloop/transport-usb
```

基本フロー

すべての接続方式で共通の 4 ステップです。

```
1. Transport 接続 → 2. App 作成 → 3. 操作 → 4. 切断
```

```
import { EthereumApp, DEFAULT_ETH_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

// 1. Transport 接続（ユーザージェスチャー内で呼び出す）
const transport = await WebHidTransport.connect()

// 2. App 作成
const eth = new EthereumApp(transport)

// 3. 操作
const { address } = await eth.getAddress(DEFAULT_ETH_PATH)
console.log('Address:', address)

// 4. 切断
await transport.close()
```

> **注意**: `WebHidTransport.connect()` と `WebBleTransport.connect()` はブラウザのセキュリティ制約により、ボタンクリック等のユーザージェスチャー内で呼び出す必要があります。

Ethereum の完全サンプル

```
import {
  EthereumApp,
  DEFAULT_ETH_PATH,
  type SignatureResult,
} from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

async function main() {
  // 接続
  const transport = await WebHidTransport.connect()
  const eth = new EthereumApp(transport)

  try {
    // アドレス取得
    const { address, publicKey } = await eth.getAddress(DEFAULT_ETH_PATH)
    console.log('Address:', address)
    console.log('Public Key:', publicKey)

    // メッセージ署名（EIP-191 personal_sign）
    const msgSig: SignatureResult = await eth.signPersonalMessage(
      DEFAULT_ETH_PATH,
      'Hello, Openloop!',
      1 // chainId: Ethereum Mainnet
    )
    console.log('Message Signature:', { v: msgSig.v, r: msgSig.r, s: msgSig.s })

    // トランザクション署名（RLP エンコード済みの raw tx を渡す）
    const rawTx = 'f86c...' // RLP-encoded transaction hex
    const txSig: SignatureResult = await eth.signTransaction(DEFAULT_ETH_PATH, rawTx)
    console.log('TX Signature:', { v: txSig.v, r: txSig.r, s: txSig.s })

    // EIP-712 Typed Data 署名
    const typedSig: SignatureResult = await eth.signTypedData(
      DEFAULT_ETH_PATH,
      'aabbccdd...', // domainSeparatorHash (32 bytes hex)
      '11223344...', // messageHash (32 bytes hex)
      1
    )
    console.log('TypedData Signature:', typedSig)
  } finally {
    await transport.close()
  }
}
```

Bitcoin の完全サンプル

```
import {
  BitcoinApp,
  BTC_MAINNET_PATH,
  type BtcMessageSignature,
} from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

async function main() {
  const transport = await WebHidTransport.connect()
  const btc = new BitcoinApp(transport)

  try {
    // アドレス取得（SegWit bech32）
    const { address, publicKey, chainCode } = await btc.getAddress(false) // mainnet
    console.log('Address:', address)  // bc1q...

    // メッセージ署名（BIP-137）
    const msgSig: BtcMessageSignature = await btc.signMessage(
      'Hello, Bitcoin!',
      BTC_MAINNET_PATH
    )
    console.log('Signature (base64):', msgSig.signature)

    // PSBT 署名
    const psbtHex = '70736274ff...' // PSBT binary as hex
    const signedPsbtHex = await btc.signPsbtHex(psbtHex)
    console.log('Signed PSBT:', signedPsbtHex)

    // PSBT 署名（入力パス指定）
    const signedWithPaths = await btc.signPsbtWithPaths(psbtHex, [
      { index: 0, path: "84'/0'/0'/0/0" },
      { index: 1, path: "84'/0'/0'/0/1" },
    ])
    console.log('Signed PSBT with paths:', signedWithPaths)
  } finally {
    await transport.close()
  }
}
```

Bluetooth 接続

WebHID を WebBLE に置き換えるだけで、他のコードは同じです。

```
import { WebBleTransport } from '@openloop/transport-webble'

// Bluetooth 接続（ユーザージェスチャー内で呼び出す）
const transport = await WebBleTransport.connect()

// 以降は WebHID と同じ
const eth = new EthereumApp(transport)
const { address } = await eth.getAddress(DEFAULT_ETH_PATH)
```

LocalWS 接続（全ブラウザ対応）

Openloop Connect デスクトップアプリ経由で接続します。Firefox や Safari でも動作します。

```
import { LocalWsTransport } from '@openloop/transport-local'

// Connect アプリが起動しているか確認
const available = await LocalWsTransport.isAvailable()
if (!available) {
  console.log('Openloop Connect を起動してください')
}

// 接続
const transport = new LocalWsTransport()
await transport.open()

// 以降は同じ
const eth = new EthereumApp(transport)
```

React での利用

```tsx
import { useState, useCallback } from 'react'
import { EthereumApp, DEFAULT_ETH_PATH, type ITransport } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

function WalletButton() {
  const [address, setAddress] = useState<string>('')
  const [transport, setTransport] = useState<ITransport | null>(null)

  const connect = useCallback(async () => {
    const t = await WebHidTransport.connect()
    setTransport(t)
    const eth = new EthereumApp(t)
    const { address } = await eth.getAddress(DEFAULT_ETH_PATH)
    setAddress(address)
  }, [])

  const disconnect = useCallback(async () => {
    await transport?.close()
    setTransport(null)
    setAddress('')
  }, [transport])

  return (
    <div>
      {address ? (
        <>
          <p>{address}</p>
          <button onClick={disconnect}>Disconnect</button>
        </>
      ) : (
        <button onClick={connect}>Connect Wallet</button>
      )}
    </div>
  )
}
```

サンプルアプリの `useOpenloop` フック（packages/sample-app/src/hooks/useOpenloop.ts）は、全 Transport 切り替え・WalletConnect 統合・自動再接続を含む実装例です。

自動再接続

WebHID と WebBLE は、以前接続が許可されたデバイスへの自動再接続をサポートしています。

```
// 以前許可されたデバイスに自動再接続（ユーザージェスチャー不要）
const transport = await WebHidTransport.reconnect()
if (transport) {
  // 再接続成功
  const eth = new EthereumApp(transport)
} else {
  // デバイスが見つからない → connect() でペアリングが必要
}
```

次のステップ

- Transport 選択ガイド — 環境に合った接続方式を選ぶ
- Ethereum API — ethers.js / viem 連携
- Bitcoin API — PSBT プロトコル詳細
- エラー処理 — エラーハンドリングのパターン

Transport ガイド

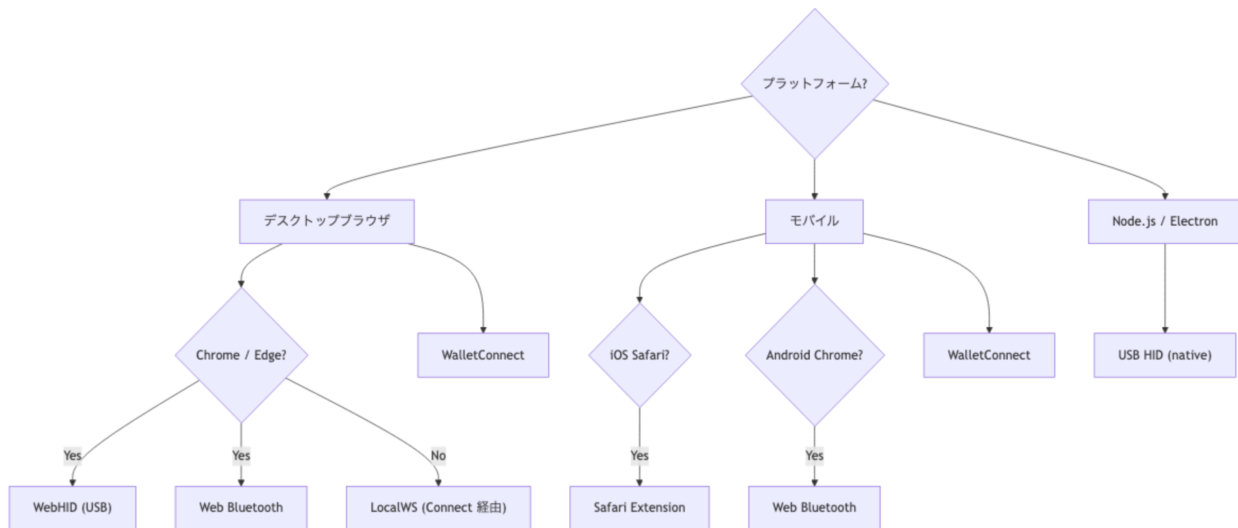
Openloop SDK は複数の接続方式（Transport）をサポートしています。dApp の対象プラットフォームに応じて適切な Transport を選択してください。

プラットフォーム対応表

Transport	Chrome/ Edge	Firefox	Safari (Desktop)	iOS Safari	Android Chrome	Node.js / Electron
WebHID (USB)	✓	✗	✗	✗	✗	✗
Web Bluetooth	✓	✗	✗	✗	✓	✗
LocalWS	✓	✓	✓	✗	✓	✗
Safari Extension	✗	✗	✗	✓	✗	✗
USB HID (native)	✗	✗	✗	✗	✗	✓
WalletConnect	✓	✓	✓	✓	✓	✗

> **WalletConnect** はデバイスと直接通信するのではなく、Openloop Connect アプリ経由のリモート署名です。デバイス操作（アドレス取得等）は Connect アプリ側で行われます。

選択フローチャート



各 Transport 詳細

WebHID (USB)

パッケージ: `@openloop/transport-webhid` **対応環境**: Chrome 89+ / Edge 89+（デスクトップのみ）

USB ケーブルでデバイスを直接接続します。最も高速で安定した接続方式です。

Static メソッド

メソッド	説明
`isSupported(): boolean`	WebHID API が利用可能か
`connect(): Promise<WebHidTransport>`	デバイスピッカーを表示して接続（ユーザージェスチャー必要）
`reconnect(): Promise<WebHidTransport | null>`	以前許可されたデバイスに自動再接続

コード例

```
import { WebHidTransport } from '@openloop/transport-webhid'

// サポート確認
if (!WebHidTransport.isSupported()) {
  console.log('WebHID is not supported in this browser')
}

// 初回接続（ボタンクリック内で）
const transport = await WebHidTransport.connect()

// 自動再接続（ページ読み込み時）
const transport = await WebHidTransport.reconnect()

// 切断検知
transport.onDisconnect(() => {
  console.log('Device disconnected')
})
```

Web Bluetooth

パッケージ: `@openloop/transport-webble` **対応環境**: Chrome 56+ / Edge 79+（デスクトップ）、Android Chrome

Bluetooth Low Energy (BLE) でワイヤレス接続します。

Static メソッド

メソッド	説明
`isSupported(): boolean`	Web Bluetooth API が利用可能か
`connect(): Promise<WebBleTransport>`	デバイスピッカーを表示して接続（ユーザージェスチャー必要）
`reconnect(): Promise<WebBleTransport | null>`	以前ペアリングしたデバイスに再接続（Chrome 100+）

コード例

```
import { WebBleTransport } from '@openloop/transport-webble'

if (!WebBleTransport.isSupported()) {
  console.log('Web Bluetooth is not supported')
}

// 初回接続
const transport = await WebBleTransport.connect()

// 自動再接続
const transport = await WebBleTransport.reconnect()
```

注意事項

- BLE はアイドル時に自動切断されることがあります。Transport は次の `exchange()` 呼び出し時に自動再接続を試みます。
- MTU（Maximum Transmission Unit）はデバイスとネゴシエーションされ、フレームサイズが自動調整されます。
- デフォルト MTU: 155 バイト

LocalWS（全ブラウザ対応）

パッケージ: `@openloop/transport-local` **対応環境**: WebSocket をサポートする全ブラウザ **前提条件**: Openloop Connect デスクトップアプリが起動していること

Openloop Connect デスクトップアプリが提供する WebSocket サーバー（`ws://127.0.0.1:21320`）経由で通信します。Firefox、Safari 等の WebHID/WebBLE 非対応ブラウザでも動作します。

Static メソッド

メソッド	説明
`isAvailable(port?, host?): Promise<boolean>`	Connect サーバーが稼働しているか

コード例

```
import { LocalWsTransport } from '@openloop/transport-local'

// Connect アプリの起動確認
const available = await LocalWsTransport.isAvailable()

// 接続
const transport = new LocalWsTransport({
  port: 21320,     // デフォルト
  host: '127.0.0.1', // デフォルト
})
await transport.open()

// デバイス接続状態の変更通知
transport.onDeviceChange((connected: boolean) => {
  console.log('Device:', connected ? 'connected' : 'disconnected')
})
```

プロトコル

JSON-RPC 2.0 over WebSocket:

メソッド	説明
`openloop_exchange`	APDU コマンド送受信
`openloop_lock`	デバイスの排他ロック取得
`openloop_unlock`	デバイスの排他ロック解放
`openloop_status`	デバイス接続状態の取得

Safari Extension（iOS）

パッケージ: `@openloop/transport-safari` **対応環境**: iOS Safari（Openloop Safari Extension インストール済み）

Safari Web Extension を経由して iOS の CoreBluetooth にアクセスし、BLE でデバイスと通信します。

Static メソッド

メソッド	説明
`isAvailable(timeout?): Promise<boolean>`	Safari Extension がインストールされているか（デフォルト 3 秒タイムアウト）
`scan(duration?): Promise<DeviceInfo[]>`	BLE デバイスをスキャン
`connect(deviceId?): Promise<SafariTransport>`	デバイスに接続

コード例

```
import { SafariTransport } from '@openloop/transport-safari'

// Extension の確認
const available = await SafariTransport.isAvailable()

// デバイスのスキャン
const devices = await SafariTransport.scan(5000) // 5秒スキャン
console.log('Found devices:', devices)

// 接続
const transport = await SafariTransport.connect(devices[0]?.deviceId)
```

USB HID (Native)

パッケージ: `@openloop/transport-usb` **対応環境**: Node.js / Electron **依存**: `node-hid`

ブラウザを使わずに直接 USB デバイスと通信します。サーバーサイドやデスクトップアプリに適しています。

Static メソッド

メソッド	説明
`discover(): UsbDeviceInfo[]`	接続されている Openloop/Ledger デバイスを列挙

コード例

```
import { UsbHidTransport } from '@openloop/transport-usb'

// デバイス検出
const devices = UsbHidTransport.discover()
if (devices.length === 0) {
  console.log('No device found')
}

// 接続
const transport = new UsbHidTransport({ path: devices[0].path })
await transport.open()

// 自動リトライ（デフォルト最大3回）
const transport = new UsbHidTransport({
  path: devices[0].path,
  maxRetries: 3,
  exchangeTimeout: 120000, // 2分
})
```

WalletConnect

パッケージ: `@openloop/sdk-core` （WcTransport は sdk-core に含まれる） **対応環境**: WebSocket をサポートする全ブラウザ

WalletConnect v2 リレー経由で、リモートの Openloop Connect アプリと通信します。カスタムメソッド `openloop_exchange` を使って APDU コマンドを中継します。

詳細は WalletConnect 連携 を参照してください。

OpenloopSDK クラスによる自動検出

`OpenloopSDK` クラスを使うと、利用可能な Transport を自動検出して接続できます。

```
import { OpenloopSDK, EthereumApp, DEFAULT_ETH_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'
import { WebBleTransport } from '@openloop/transport-webble'
import { LocalWsTransport } from '@openloop/transport-local'

// Transport を登録
OpenloopSDK.registerTransport('webhid', {
  factory: () => WebHidTransport.connect(),
  name: 'WebHID (USB)',
  isAvailable: () => WebHidTransport.isSupported(),
})

OpenloopSDK.registerTransport('webble', {
  factory: () => WebBleTransport.connect(),
  name: 'Web Bluetooth',
  isAvailable: () => WebBleTransport.isSupported(),
})

OpenloopSDK.registerTransport('local', {
  factory: async () => {
    const t = new LocalWsTransport()
    await t.open()
    return t
  },
  name: 'LocalWS (Connect)',
  isAvailable: () => LocalWsTransport.isAvailable(),
})

// 利用可能な Transport を確認
const transports = await OpenloopSDK.discover()
console.log(transports)
// [
//   { type: 'webhid', name: 'WebHID (USB)', available: true },
//   { type: 'webble', name: 'Web Bluetooth', available: true },
//   { type: 'local', name: 'LocalWS (Connect)', available: false },
// ]

// 指定した Transport で接続
const transport = await OpenloopSDK.connect({ transport: 'webhid' })

// または最初に利用可能な Transport で接続
const transport = await OpenloopSDK.connect()
```

再接続パターン

```
// ページ読み込み時に以前のデバイスへ自動再接続
async function autoReconnect(): Promise<ITransport | null> {
  // WebHID: 以前許可されたデバイスを検出
  if (WebHidTransport.isSupported()) {
    const transport = await WebHidTransport.reconnect()
    if (transport) return transport
  }

  // WebBLE: 以前ペアリングしたデバイスを検出（Chrome 100+）
  if (WebBleTransport.isSupported()) {
    const transport = await WebBleTransport.reconnect()
    if (transport) return transport
  }

  // LocalWS: Connect アプリが起動していれば接続
  if (await LocalWsTransport.isAvailable()) {
    const transport = new LocalWsTransport()
    await transport.open()
    return transport
  }

  return null
}
```

次のステップ

- Ethereum API — EthereumApp の全メソッド
- Bitcoin API — PSBT 署名の詳細
- WalletConnect 連携 — WcTransport / AppKit

Bitcoin API

`BitcoinApp` クラスは、Bitcoin のアドレス取得、メッセージ署名（BIP-137）、PSBT（Partially Signed Bitcoin Transaction）署名を提供します。

インポート

```
import {
  BitcoinApp,
  BTC_MAINNET_PATH,
  BTC_TESTNET_PATH,
  type BtcMessageSignature,
  type InputSigningPath,
} from '@openloop/sdk-core'
```

初期化

```
const btc = new BitcoinApp(transport)
```

メソッド一覧

メソッド	説明
`getAddress(testnet?)`	SegWit アドレス取得（mainnet/testnet 自動切替）
`getAddressWithPath(path)`	任意の BIP32 パスでアドレス取得
`getAccountXpub(coinType)`	アカウントレベル拡張公開鍵の取得
`signMessage(message, path)`	BIP-137 メッセージ署名
`signPsbt(psbt)`	PSBT 署名（バイナリ入出力）
`signPsbtHex(psbt)`	PSBT 署名（hex 入出力）
`signPsbtWithPaths(psbt, inputPaths)`	入力パス指定付き PSBT 署名

getAddress

SegWit（bech32）アドレスを取得します。

```
async getAddress(testnet?: boolean): Promise<{
  publicKey: string   // 非圧縮公開鍵 (hex)
  address: string     // bech32 アドレス ("bc1..." or "tb1...")
  chainCode: string   // チェーンコード (hex, 32 bytes)
}>
```

パラメータ

名前	型	デフォルト	説明
`testnet`	`boolean`	`false`	`true`： testnet パス（`84'/1'/0'/0/0`）

デフォルトパス

ネットワーク	パス	定数
Mainnet	`84'/0'/0'/0/0`	`BTC_MAINNET_PATH`
Testnet	`84'/1'/0'/0/0`	`BTC_TESTNET_PATH`

使用例

```
// Mainnet
const { address } = await btc.getAddress()
// address: "bc1q..."

// Testnet
const { address } = await btc.getAddress(true)
// address: "tb1q..."
```

---

getAddressWithPath

任意の BIP32 パスでアドレスを取得します。マルチアカウントやお釣り用アドレスの取得に使用します。

```
async getAddressWithPath(path: string): Promise<{
  publicKey: string
  address: string
  chainCode: string
}>
```

使用例

```
// 2番目のアカウントの最初のアドレス
const { address } = await btc.getAddressWithPath("84'/0'/1'/0/0")

// お釣り用アドレス
const { address } = await btc.getAddressWithPath("84'/0'/0'/1/0")

// "m/" プレフィックスも可
const { address } = await btc.getAddressWithPath("m/84'/0'/0'/0/5")
```

---

getAccountXpub

アカウントレベルの拡張公開鍵（`m/84'/coin'/0'`）を取得します。ソフトウェア側で子鍵導出を行い、残高確認やアドレス生成に使用します。

```
async getAccountXpub(coinType: number): Promise<{
  publicKey: string   // 圧縮公開鍵 (hex, 33 bytes)
  chainCode: string   // チェーンコード (hex, 32 bytes)
}>
```

パラメータ

名前	型	説明
`coinType`	`number`	`0` : mainnet, `1` : testnet

使用例

```
const { publicKey, chainCode } = await btc.getAccountXpub(0)  // mainnet
```

---

signMessage

BIP-137 形式でメッセージに署名します。

```
async signMessage(
  message: string,
  path: string
): Promise<BtcMessageSignature>
```

パラメータ

名前	型	説明
`message`	`string`	UTF-8 メッセージ
`path`	`string`	BIP32 パス（例: `"84'/0'/0'/0/0"`）

返り値: `BtcMessageSignature`

フィールド	型	説明
`v`	`number`	リカバリー ID (35-38: P2WPKH native SegWit)
`r`	`string`	R 値 (32 bytes hex)
`s`	`string`	S 値 (32 bytes hex)
`signature`	`string`	完全な署名 (65 bytes: V || R || S) の Base64

使用例

```
const sig = await btc.signMessage('Hello, Bitcoin!', BTC_MAINNET_PATH)
console.log(sig.signature) // Base64 encoded signature
```

---

signPsbt

PSBT（Partially Signed Bitcoin Transaction）に署名します。PSBT 内の `BIP32_DERIVATION` フィールドからパスを自動検出します。

```
async signPsbt(psbt: Uint8Array | string): Promise<Uint8Array>
```

パラメータ

名前	型	説明
`psbt`	`Uint8Array | string`	PSBT バイナリデータまたは hex 文字列

返り値

署名済み PSBT のバイナリデータ（`Uint8Array`）

---

signPsbtHex

hex 文字列での PSBT 署名。入出力ともに hex 文字列です。

```
async signPsbtHex(psbt: string): Promise<string>
```

使用例

```
const signedHex = await btc.signPsbtHex('70736274ff...')
```

---

signPsbtWithPaths

各入力の BIP32 パスを明示的に指定して PSBT に署名します。PSBT に `BIP32_DERIVATION` フィールドがない場合や、特定の入力のみ署名したい場合に使用します。

```
async signPsbtWithPaths(
  psbt: string,
  inputPaths: InputSigningPath[]
): Promise<string>
```

パラメータ

名前	型	説明
`psbt`	`string`	PSBT hex 文字列
`inputPaths`	`InputSigningPath[]`	入力ごとのパス指定

`InputSigningPath`

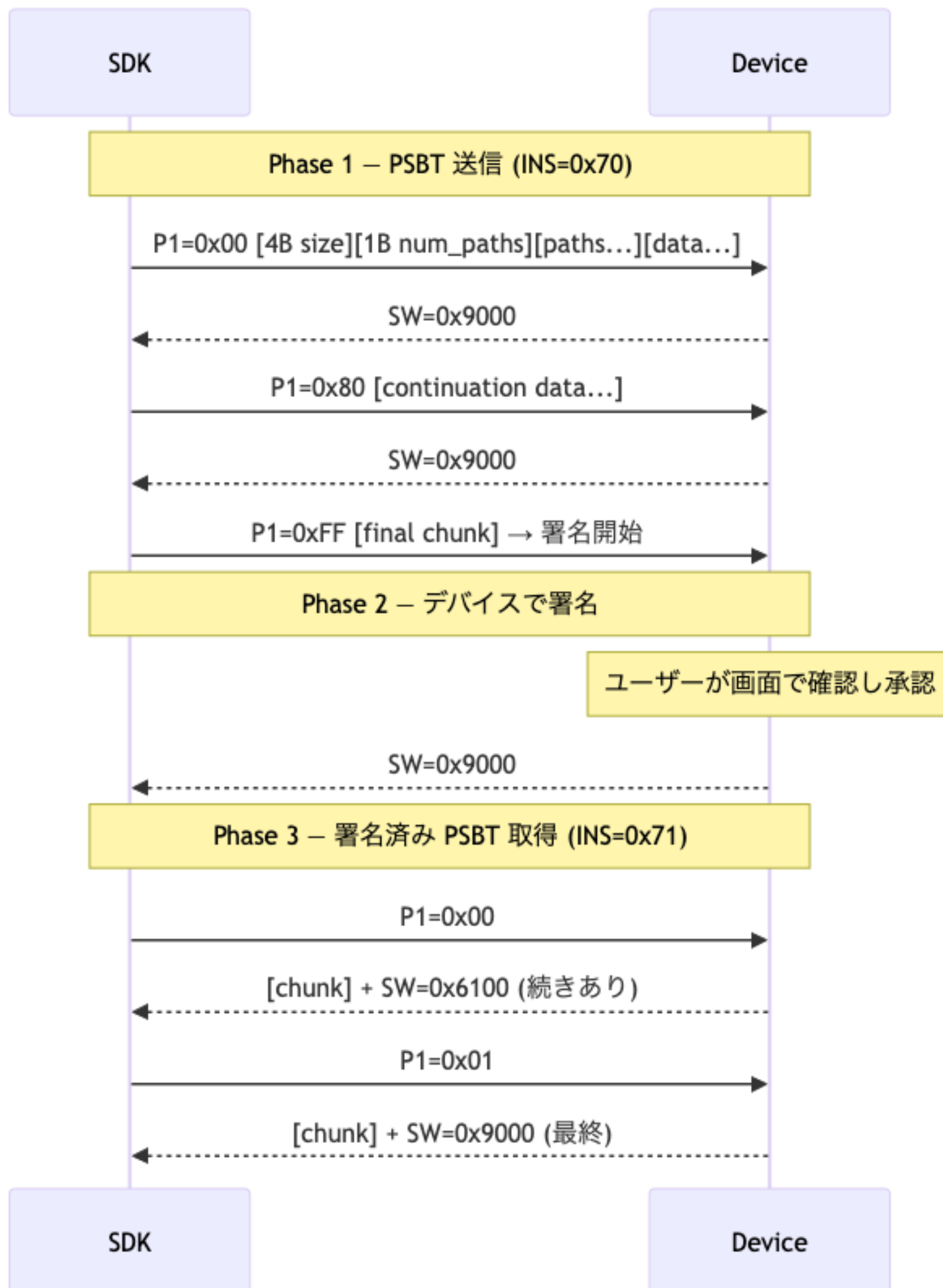
フィールド	型	説明
`index`	`number`	PSBT の入力インデックス
`path`	`string`	BIP32 パス（例: `"84'/0'/0'/0/5"`）

使用例

```
const signedHex = await btc.signPsbtWithPaths('70736274ff...', [
  { index: 0, path: "84'/0'/0'/0/0" },
  { index: 1, path: "84'/0'/0'/0/3" },
  { index: 2, path: "84'/0'/0'/1/0" }, // お釣り
])
```

PSBT プロトコル詳細

PSBT 署名はデバイスとの 3 フェーズ通信で行われます。


SDK
Device
Phase 1 — PSBT 送信 (INS=0x70)
P1=0x00 [4B size][1B num_paths][paths...][data...]
SW=0x9000
P1=0x80 [continuation data...]
SW=0x9000
P1=0xFF [final chunk] → 署名開始
Phase 2 — デバイスで署名
ユーザーが画面で確認し承認
SW=0x9000
Phase 3 — 署名済み PSBT 取得 (INS=0x71)
P1=0x00
[chunk] + SW=0x6100 (続きあり)
P1=0x01
[chunk] + SW=0x9000 (最終)
SDK
Device

Phase 1: PSBT 送信

チャンク	P1	データ形式
最初	`0x00`	`[4B total_size][1B num_paths][paths...][psbt_data...]`
継続	`0x80`	`[psbt_data...]`
最終	`0xFF`	`[remaining_data]` → デバイスで署名開始

- `num_paths=0` : PSBT 内の `BIP32_DERIVATION` から自動検出
- `num_paths>0` : 各入力のパスを明示指定（ `[1B input_index][1B path_len][4B * path_len elements]` ）

Phase 2: デバイスで署名

ユーザーがデバイス画面でトランザクション内容を確認し、承認ボタンを押します。

Phase 3: 署名済み PSBT 取得

レスポンス	ステータスワード	意味
`[signed_psbt_chunk]`	`0x61XX`	続きデータあり（次のチャンクを要求）
`[signed_psbt_chunk]`	`0x9000`	最終チャンク（取得完了）

チャンクインデックスを 0 から順にインクリメントし、 `0x9000` が返るまで繰り返します。

---

bitcoinjs-lib 連携

```
import * as bitcoin from 'bitcoinjs-lib'
import { BitcoinApp, BTC_MAINNET_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

const transport = await WebHidTransport.connect()
const btc = new BitcoinApp(transport)

// アドレス取得
const { address, publicKey } = await btc.getAddress()

// PSBT 作成
const psbt = new bitcoin.Psbt({ network: bitcoin.networks.bitcoin })
psbt.addInput({
  hash: 'txid...',
  index: 0,
  witnessUtxo: {
    script: Buffer.from('0014...', 'hex'),
    value: 50000,
  },
})
psbt.addOutput({
  address: 'bc1q...',
  value: 40000,
})

// PSBT をデバイスで署名
const psbtHex = psbt.toHex()
const signedHex = await btc.signPsbtWithPaths(psbtHex, [
  { index: 0, path: BTC_MAINNET_PATH },
])

// 署名済み PSBT を読み込み
const signedPsbt = bitcoin.Psbt.fromHex(signedHex)
signedPsbt.finalizeAllInputs()

// ブロードキャスト用 raw transaction
const rawTx = signedPsbt.extractTransaction().toHex()
```

次のステップ

- Solana API
- エラー処理 — ユーザー拒否時のハンドリング

Ethereum API

`EthereumApp` クラスは、Ethereum のアドレス取得、メッセージ署名、トランザクション署名、EIP-712 Typed Data 署名、EIP-7702 Authorization 署名を提供します。

インポート

```
import {
  EthereumApp,
  DEFAULT_ETH_PATH,
  type SignatureResult,
} from '@openloop/sdk-core'
```

初期化

```
const eth = new EthereumApp(transport)
```

`transport` は `ITransport` インターフェースを実装する任意の Transport インスタンスです。

メソッド一覧

メソッド	説明
`getAddress(path)`	アドレスと公開鍵を取得
`signPersonalMessage(path, message, chainId?)`	EIP-191 personal_sign
`signTransaction(path, rawTx)`	RLP エンコード済みトランザクションに署名
`signTypedData(path, domainHash, msgHash, chainId?)`	EIP-712 Typed Data に署名
`signAuthorization(path, authorizationRlp)`	EIP-7702 Authorization に署名

getAddress

デバイスから Ethereum アドレスと公開鍵を取得します。

```
async getAddress(path: string): Promise<{
  publicKey: string  // 非圧縮公開鍵 (hex, 65 bytes)
  address: string    // チェックサム付きアドレス ("0x...")
}>
```

パラメータ

名前	型	説明
`path`	`string`	BIP44 パス（例: `"44'/60'/0'/0/0"`）

使用例

```
import { DEFAULT_ETH_PATH } from '@openloop/sdk-core'

const { address, publicKey } = await eth.getAddress(DEFAULT_ETH_PATH)
// address: "0x742d35Cc6634C0532925a3b844Bc9e7595f2bD18"
// publicKey: "04a1b2c3d4..."
```

デフォルトパス

`DEFAULT_ETH_PATH = "44'/60'/0'/0/0"`

---

signPersonalMessage

EIP-191 personal_sign でメッセージに署名します。チェーン ID を認識する Openloop 独自の拡張プロトコルを使用します。

```
async signPersonalMessage(
  path: string,
  message: string,
  chainId?: number
): Promise<SignatureResult>
```

パラメータ

名前	型	デフォルト	説明
`path`	`string`	—	BIP44 パス
`message`	`string`	—	メッセージ（UTF-8 文字列または `0x` プレフィックス付き hex）
`chainId`	`number`	`1`	EVM チェーン ID

返り値: `SignatureResult`

フィールド	型	説明
`v`	`number`	リカバリー ID (27 or 28)
`r`	`string`	R 値 (32 bytes hex, `0x` なし)
`s`	`string`	S 値 (32 bytes hex, `0x` なし)

使用例

```
const sig = await eth.signPersonalMessage(
  DEFAULT_ETH_PATH,
  'Hello, Openloop!',
  1
)
// sig.v = 27
// sig.r = "a1b2c3..."
// sig.s = "d4e5f6..."
```

> **注意**: 長いメッセージは自動的にチャンク分割されてデバイスに送信されます（最初のチャンク 150 バイト、以降 255 バイトずつ）。

---

signTransaction

RLP エンコード済みの Ethereum トランザクションに署名します。

```
async signTransaction(
  path: string,
  rawTx: string
): Promise<SignatureResult>
```

パラメータ

名前	型	説明
`path`	`string`	BIP44 パス
`rawTx`	`string`	RLP エンコード済みトランザクション (hex, `0x` プレフィックスあり/なし)

使用例

```
const sig = await eth.signTransaction(
  DEFAULT_ETH_PATH,
  'f86c0a8502540be400825208...'
)
```

---

signTypedData

EIP-712 Typed Data に署名します。事前にハッシュ化された domainSeparator と message を渡します。

```
async signTypedData(
  path: string,
  domainSeparatorHash: string,
  messageHash: string,
  chainId?: number
): Promise<SignatureResult>
```

パラメータ

名前	型	デフォルト	説明
`path`	`string`	—	BIP44 パス
`domainSeparatorHash`	`string`	—	EIP-712 domain separator ハッシュ (32 bytes hex)
`messageHash`	`string`	—	EIP-712 message ハッシュ (32 bytes hex)
`chainId`	`number`	`1`	EVM チェーン ID

使用例

```
const sig = await eth.signTypedData(
  DEFAULT_ETH_PATH,
  'aabbccddee...', // domainSeparatorHash
  '1122334455...', // messageHash
  1
)
```

---

signAuthorization

EIP-7702 Authorization（EOA に対する Set Code）に署名します。

```
async signAuthorization(
  path: string,
  authorizationRlp: string
): Promise<SignatureResult>
```

パラメータ

名前	型	説明
`path`	`string`	BIP44 パス
`authorizationRlp`	`string`	`RLP([chain_id, address, nonce])` の hex

使用例

```
const sig = await eth.signAuthorization(
  DEFAULT_ETH_PATH,
  'c3010a...' // RLP([1, "0x...", 0])
)
```

ethers.js 連携

```
import { ethers } from 'ethers'
import { EthereumApp, DEFAULT_ETH_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

const transport = await WebHidTransport.connect()
const eth = new EthereumApp(transport)

// アドレス取得
const { address } = await eth.getAddress(DEFAULT_ETH_PATH)

// トランザクション作成
const tx = {
  to: '0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045',
  value: ethers.parseEther('0.01'),
  gasLimit: 21000n,
  gasPrice: (await provider.getFeeData()).gasPrice,
  nonce: await provider.getTransactionCount(address),
  chainId: 1n,
}

// RLP エンコードして署名
const unsignedTx = ethers.Transaction.from(tx).unsignedSerialized
const sig = await eth.signTransaction(DEFAULT_ETH_PATH, unsignedTx)

// 署名済みトランザクションを構築
const signedTx = ethers.Transaction.from({
  ...tx,
  signature: {
    v: sig.v,
    r: '0x' + sig.r,
    s: '0x' + sig.s,
  },
})

// ブロードキャスト
const txResponse = await provider.broadcastTransaction(signedTx.serialized)
```

viem 連携

```
import { createPublicClient, http, parseEther } from 'viem'
import { mainnet } from 'viem/chains'
import { EthereumApp, DEFAULT_ETH_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

const transport = await WebHidTransport.connect()
const eth = new EthereumApp(transport)

const { address } = await eth.getAddress(DEFAULT_ETH_PATH)

// personal_sign
const sig = await eth.signPersonalMessage(
  DEFAULT_ETH_PATH,
  'Hello from viem!',
  1
)

// viem の署名フォーマットに変換
const signature = `0x${sig.r}${sig.s}${sig.v.toString(16)}` as `0x${string}`
```

次のステップ

- Bitcoin API — PSBT 署名
- エラー処理 — ユーザー拒否等のハンドリング

XRP API

`XrpApp` クラスは、XRP Ledger のアドレス取得とトランザクション署名を提供します。Ed25519（デフォルト）と secp256k1 の両方の楕円曲線をサポートしています。

Ethereum との主な違い

項目	Ethereum	XRP
アドレス形式	`0x...` (hex)	`r...` (Base58Check)
デフォルト曲線	secp256k1	Ed25519
Ed25519 公開鍵	—	`0xED` プレフィックス付き
署名フォーマット	v/r/s	raw bytes (Ed25519: 64B, secp256k1: DER)
BIP44 パス	`44'/60'/...`	`44'/144'/...` (Ed25519: 全ハードニング)

インポート

```
import {
  XrpApp,
  DEFAULT_XRP_PATH,
  type XrpSignatureResult,
  type XrpCurve,
} from '@openloop/sdk-core'
```

初期化

```
const xrp = new XrpApp(transport)
```

メソッド一覧

メソッド	説明
`getAddress(path, curve?)`	XRP アドレスと公開鍵を取得
`signTransaction(path, txBlob)`	シリアライズ済みトランザクションに署名

---

getAddress

デバイスから XRP Ledger アドレスを取得します。

```
async getAddress(
  path: string,
  curve?: XrpCurve
): Promise<{
  publicKey: string  // 公開鍵 (hex, Ed25519: 33B with 0xED prefix, secp256k1: 33B compressed)
  address: string    // XRP アドレス ("r...")
}>
```

パラメータ

名前	型	デフォルト	説明
`path`	`string`	—	BIP44 パス
`curve`	`XrpCurve`	`'ed25519'`	`'ed25519'` or `'secp256k1'`

`XrpCurve` 型

```
type XrpCurve = 'ed25519' | 'secp256k1'
```

デフォルトパス

`DEFAULT_XRP_PATH = "44'/144'/0'/0'/0'"` （Ed25519: 全要素ハードニング）

使用例

```
// Ed25519（デフォルト）
const { address, publicKey } = await xrp.getAddress(DEFAULT_XRP_PATH)
// address: "rHb9CJAWyB4rj91VRWn96DkukG4bwdtyTh"
// publicKey: "ed01a2b3c4..." (0xED prefix)

// secp256k1
const { address, publicKey } = await xrp.getAddress(
  "44'/144'/0'/0/0",  // secp256k1 は非ハードニングパスも可
  'secp256k1'
)
```

曲線選択と P2 値

曲線	P2 値	説明
`'ed25519'`	`0x80`	Ed25519（デフォルト、推奨）
`'secp256k1'`	`0x40`	secp256k1（Ethereum 互換）

---

signTransaction

シリアライズ済みの XRP トランザクションに署名します。

```
async signTransaction(
  path: string,
  txBlob: string
): Promise<XrpSignatureResult>
```

パラメータ

名前	型	説明
`path`	`string`	BIP44 パス
`txBlob`	`string`	シリアライズ済みトランザクション (hex, `0x` あり/なし)

返り値: `XrpSignatureResult`

フィールド	型	説明
`signature`	`string`	署名の hex（Ed25519: 64 bytes, secp256k1: DER エンコード可変長）

使用例

```
const { signature } = await xrp.signTransaction(
  DEFAULT_XRP_PATH,
  '120000228...' // serialized transaction blob
)
```

xrpl.js 連携

```
import { Client, Wallet, encode, decode } from 'xrpl'
import { XrpApp, DEFAULT_XRP_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

const transport = await WebHidTransport.connect()
const xrp = new XrpApp(transport)

// アドレス取得
const { address, publicKey } = await xrp.getAddress(DEFAULT_XRP_PATH)

// XRP Ledger に接続
const client = new Client('wss://xrplcluster.com/')
await client.connect()

// トランザクション作成
const prepared = await client.autofill({
  TransactionType: 'Payment',
  Account: address,
  Amount: '1000000', // 1 XRP = 1,000,000 drops
  Destination: 'rPT1Sjq2YGrBMTttX4GZHjKu9dyfzbpAYe',
})

// シリアライズしてデバイスで署名
const txBlob = encode(prepared)
const { signature } = await xrp.signTransaction(DEFAULT_XRP_PATH, txBlob)

// 署名をトランザクションに追加
const signedTx = {
  ...prepared,
  TxnSignature: signature.toUpperCase(),
  SigningPubKey: publicKey.toUpperCase(),
}

// ブロードキャスト
const result = await client.submitAndWait(encode(signedTx))
console.log('TX Result:', result.result.meta.TransactionResult)

await client.disconnect()
```

次のステップ

- WalletConnect 連携
- エラー処理

Solana API

`SolanaApp` クラスは、Solana のアドレス取得、トランザクション署名、オフチェーンメッセージ署名を提供します。Ledger Solana アプリ互換の APDU プロトコルを使用します。

インポート

```
import {
  SolanaApp,
  DEFAULT_SOL_PATH,
} from '@openloop/sdk-core'
```

初期化

```
const sol = new SolanaApp(transport)
```

メソッド一覧

メソッド	説明
`getAddress(path)`	Ed25519 公開鍵と Base58 アドレスを取得
`signTransaction(path, txBytes)`	トランザクションに署名
`signOffchainMessage(path, messageBytes)`	オフチェーンメッセージに署名

---

getAddress

デバイスから Solana アドレス（Ed25519 公開鍵の Base58 エンコード）を取得します。

```
async getAddress(path: string): Promise<{
  publicKey: string  // Ed25519 公開鍵 (hex, 32 bytes)
  address: string    // Base58 アドレス
}>
```

パラメータ

名前	型	説明
`path`	`string`	BIP44 パス（例: `"44'/501'/0'/0'"`）

デフォルトパス

`DEFAULT_SOL_PATH = "44'/501'/0'/0'"`

> **注意**: Solana は Ed25519 を使用するため、パスの全要素がハードニングされています。

使用例

```
const { address, publicKey } = await sol.getAddress(DEFAULT_SOL_PATH)
// address: "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU"
```

---

signTransaction

Solana トランザクションのバイナリデータに Ed25519 署名を行います。

```
async signTransaction(
  path: string,
  txBytes: Uint8Array
): Promise<string>  // 64-byte Ed25519 署名 (hex)
```

パラメータ

名前	型	説明
`path`	`string`	BIP44 パス
`txBytes`	`Uint8Array`	シリアライズ済みトランザクション（raw binary）

返り値

64 バイトの Ed25519 署名（hex 文字列）

使用例

```
const signatureHex = await sol.signTransaction(
  DEFAULT_SOL_PATH,
  transactionBytes
)
```

> **注意**: 大きなトランザクションは自動的に Ledger P2 チャンキングプロトコルで分割送信されます。

---

signOffchainMessage

オフチェーンメッセージ（Sign-In With Solana 等）に署名します。

```
async signOffchainMessage(
  path: string,
  messageBytes: Uint8Array
): Promise<string>  // 64-byte Ed25519 署名 (hex)
```

パラメータ

名前	型	説明
`path`	`string`	BIP44 パス
`messageBytes`	`Uint8Array`	メッセージの raw bytes

使用例

```
const message = new TextEncoder().encode('Hello, Solana!')
const signatureHex = await sol.signOffchainMessage(DEFAULT_SOL_PATH, message)
```

@solana/web3.js 連携

```
import {
  Connection,
  PublicKey,
  SystemProgram,
  Transaction,
} from '@solana/web3.js'
import { SolanaApp, DEFAULT_SOL_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

const transport = await WebHidTransport.connect()
const sol = new SolanaApp(transport)

// アドレス取得
const { address } = await sol.getAddress(DEFAULT_SOL_PATH)
const publicKey = new PublicKey(address)

// トランザクション作成
const connection = new Connection('https://api.mainnet-beta.solana.com')
const recentBlockhash = (await connection.getLatestBlockhash()).blockhash

const tx = new Transaction()
tx.add(
  SystemProgram.transfer({
    fromPubkey: publicKey,
    toPubkey: new PublicKey('...'),
    lamports: 1000000, // 0.001 SOL
  })
)
tx.recentBlockhash = recentBlockhash
tx.feePayer = publicKey

// デバイスで署名
const txBytes = tx.serializeMessage()
const signatureHex = await sol.signTransaction(DEFAULT_SOL_PATH, txBytes)

// 署名をトランザクションに追加
const signatureBytes = Uint8Array.from(
  signatureHex.match(/.{2}/g)!.map(b => parseInt(b, 16))
)
tx.addSignature(publicKey, Buffer.from(signatureBytes))

// ブロードキャスト
const txId = await connection.sendRawTransaction(tx.serialize())
console.log('TX:', txId)
```

次のステップ

- TRON API
- XRP API

TRON API

`TronApp` クラスは、TRON のアドレス取得、トランザクション署名、メッセージ署名を提供します。Ethereum と同じ INS コードを使用しますが、BIP44 パスの `coin_type=195'` によりファームウェアが TRON モードで動作します。

Ethereum との主な違い

項目	Ethereum	TRON
アドレス形式	`0x...` (hex)	`T...` (Base58Check)
TX ハッシュ	Keccak-256	SHA-256
署名レスポンス	V(1) + R(32) + S(32)	R(32) + S(32) + V(1)
V の値	27 or 28	0 or 1 (recovery ID)

インポート

```
import {
  TronApp,
  DEFAULT_TRON_PATH,
  type SignatureResult,
} from '@openloop/sdk-core'
```

初期化

```
const tron = new TronApp(transport)
```

メソッド一覧

メソッド	説明
`getAddress(path)`	TRON アドレス（`T...`）と公開鍵を取得
`signTransaction(path, rawDataHex)`	トランザクション raw_data に署名
`signMessage(path, message)`	メッセージに署名

getAddress

デバイスから TRON アドレス（Base58Check 形式 `T...`）を取得します。

```
async getAddress(path: string): Promise<{
  publicKey: string  // 非圧縮公開鍵 (hex, 65 bytes)
  address: string    // Base58Check アドレス ("T...")
}>
```

パラメータ

名前	型	説明
`path`	`string`	BIP44 パス（例: `"44'/195'/0'/0/0"`）

デフォルトパス

`DEFAULT_TRON_PATH = "44'/195'/0'/0/0"`

使用例

```
const { address } = await tron.getAddress(DEFAULT_TRON_PATH)
// address: "TLsV62...txnV1Fm"
```

signTransaction

TRON トランザクションの `raw_data` に署名します。

```
async signTransaction(
  path: string,
  rawDataHex: string
): Promise<SignatureResult>
```

パラメータ

名前	型	説明
`path`	`string`	BIP44 パス
`rawDataHex`	`string`	protobuf エンコード済み `raw_data` の hex（`0x` あり/なし）

返り値: `SignatureResult`

フィールド	型	説明
`v`	`number`	リカバリー ID (0 or 1)
`r`	`string`	R 値 (32 bytes hex)
`s`	`string`	S 値 (32 bytes hex)

使用例

```
const sig = await tron.signTransaction(
  DEFAULT_TRON_PATH,
  'a1b2c3d4...' // raw_data hex
)
```

---

signMessage

TRON メッセージに署名します。ファームウェアが BIP44 パスの `coin_type=195'` を見て TRON 用のメッセージフォーマットを適用します。

```
async signMessage(
  path: string,
  message: string
): Promise<SignatureResult>
```

パラメータ

名前	型	説明
`path`	`string`	BIP44 パス
`message`	`string`	UTF-8 メッセージ文字列

使用例

```
const sig = await tron.signMessage(DEFAULT_TRON_PATH, 'Hello, TRON!')
```

TronWeb 連携

```
import TronWeb from 'tronweb'
import { TronApp, DEFAULT_TRON_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

const transport = await WebHidTransport.connect()
const tron = new TronApp(transport)

// アドレス取得
const { address } = await tron.getAddress(DEFAULT_TRON_PATH)

// TronWeb でトランザクション作成
const tronWeb = new TronWeb({ fullHost: 'https://api.trongrid.io' })

const transaction = await tronWeb.transactionBuilder.sendTrx(
  'TLsV62...', // 送信先
  1000000,     // 1 TRX = 1,000,000 SUN
  address
)

// raw_data を hex に変換してデバイスで署名
const rawDataHex = Buffer.from(
  JSON.stringify(transaction.raw_data)
).toString('hex')

const sig = await tron.signTransaction(DEFAULT_TRON_PATH, rawDataHex)

// 署名をトランザクションに追加
const signedTx = {
  ...transaction,
  signature: [`${sig.r}${sig.s}0${sig.v}`],
}

// ブロードキャスト
const result = await tronWeb.trx.sendRawTransaction(signedTx)
```

次のステップ

- XRP API
- エラー処理

WalletConnect 連携

Openloop SDK は WalletConnect v2 を通じたリモートデバイスアクセスを 2 つの方式でサポートしています。

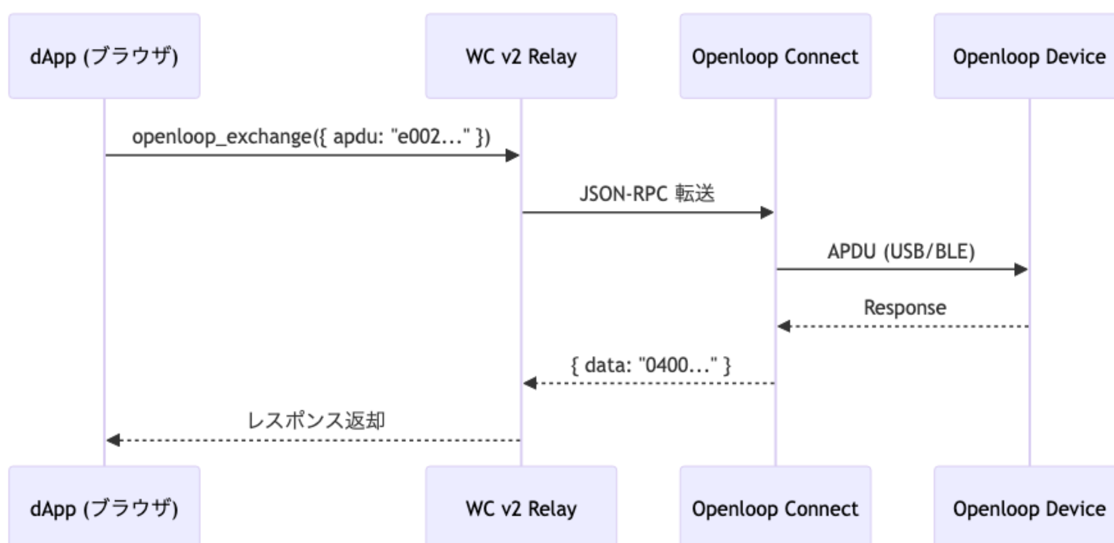
2 つの方式

方式	レベル	説明
WcTransport	低レベル	APDU コマンドを WalletConnect リレー経由で中継
AppKit	高レベル	Reown AppKit でマルチチェーン対応の標準 WC 接続

WcTransport（APDU 中継方式）

`WcTransport` は `ITransport` を実装し、WalletConnect v2 リレーを通じて APDU コマンドをリモートの Openloop Connect アプリに送信します。

仕組み



カスタム JSON-RPC メソッド `openloop_exchange` で hex エンコードされた APDU を送受信します。

セットアップ

```
import { WcTransport, EthereumApp, DEFAULT_ETH_PATH } from '@openloop/sdk-core'
import SignClient from '@walletconnect/sign-client'

// WalletConnect SignClient を初期化
const client = await SignClient.init({
  projectId: 'your-project-id',
})

// ペアリングとセッション確立
const { uri, approval } = await client.connect({
  requiredNamespaces: {
    eip155: {
      methods: ['openloop_exchange', 'openloop_lock', 'openloop_unlock'],
      chains: ['eip155:1'],
      events: [],
    },
  },
})

// URI を QR コードで表示 → ユーザーが Openloop Connect でスキャン
console.log('Pairing URI:', uri)

// セッション承認を待つ
const session = await approval()

// WcTransport 作成
const transport = new WcTransport({
  client,
  session: { topic: session.topic },
  chainId: 'eip155:1', // オプション（デフォルト: 'eip155:1'）
})

await transport.open()

// 通常の App クラスがそのまま使える
const eth = new EthereumApp(transport)
const { address } = await eth.getAddress(DEFAULT_ETH_PATH)
```

WcTransport の型定義

```
interface WcTransportOptions {
  client: IWcSignClient   // WalletConnect SignClient インスタンス
  session: WcSession       // アクティブセッション { topic: string }
  chainId?: string         // チェーン ID（デフォルト: "eip155:1"）
}

interface IWcSignClient {
  request<T>(params: {
    topic: string
    chainId: string
    request: { method: string; params: unknown }
  }): Promise<T>
}

interface WcSession {
  topic: string
}
```

対応メソッド

メソッド	説明
`openloop_exchange`	APDU コマンド送受信
`openloop_lock`	デバイスの排他ロック取得
`openloop_unlock`	デバイスの排他ロック解放

特徴

- デバイスに直接アクセスできない環境でも署名可能
- `ITransport` インターフェースを実装しているため、全 App クラスがそのまま使える
- デバイスロック（`lock` / `unlock`）もリレー経由で動作

AppKit（高レベル WC 方式）

Reown AppKit を使うと、WalletConnect の標準メソッド（`eth_signTransaction`, `personal_sign` 等）で署名できます。この方式では APDU は直接扱わず、Openloop Connect が内部でデバイス操作を行います。

対応チェーンとメソッド

EVM (eip155)

チェーン: - `eip155:1` — Ethereum Mainnet - `eip155:11155111` — Sepolia Testnet

メソッド: | メソッド | 説明 | |———|——| | `eth_sendTransaction` | トランザクション送信 | | `eth_signTransaction` | トランザクション署名 | | `eth_sign` | データ署名 | | `personal_sign` | EIP-191 メッセージ署名 | | `eth_signTypedData` | EIP-712 Typed Data 署名 | | `eth_signTypedData_v3` | EIP-712 v3 | | `eth_signTypedData_v4` | EIP-712 v4 | | `wallet_getCapabilities` | ウォレット機能取得 |

Bitcoin (bip122)

チェーン: - `bip122:000000000019d6689c085ae165831e93` — Bitcoin Mainnet - `bip122:000000000933ea01ad0ee984209779ba` — Bitcoin Testnet3

メソッド: | メソッド | 説明 | |———|——| | `signPsbt` | PSBT 署名 | | `getAccountAddresses` | アカウントアドレス取得 | | `signMessage` | メッセージ署名 | | `sendTransfer` | 送金 |

Solana

チェーン: - `solana:5eykt4UsFv8P8NJdTREpY1vzqKqZKvdp` — Mainnet Beta - `solana:EtWTRABZaYq6iMfeYKouRu166VU2xqa1` — Devnet

メソッド: | メソッド | 説明 | |———|——| | `solana_signTransaction` | トランザクション署名 | | `solana_signAllTransactions` | 複数トランザクション署名 | | `solana_signAndSendTransaction` | 署名＋送信 | | `solana_signMessage` | メッセージ署名 |

TRON

チェーン: - `tron:0x2b6653dc` — TRON Mainnet - `tron:0x94a9059e` — TRON Shasta Testnet

メソッド: | メソッド | 説明 | |———|——| | `tron_signTransaction` | トランザクション署名 | | `tron_signMessage` | メッセージ署名 |

AppKit の設定例

```
import { createAppKit } from '@reown/appkit/react'
import { WagmiAdapter } from '@reown/appkit-adapter-wagmi'
import { SolanaAdapter } from '@reown/appkit-adapter-solana'
import { BitcoinAdapter } from '@reown/appkit-adapter-bitcoin'
import { TronAdapter } from '@reown/appkit-adapter-tron'
import { mainnet, sepolia } from '@reown/appkit/networks'

const projectId = 'your-walletconnect-project-id'

// アダプター作成
const wagmiAdapter = new WagmiAdapter({ projectId, networks: [mainnet, sepolia] })
const solanaAdapter = new SolanaAdapter()
const bitcoinAdapter = new BitcoinAdapter()
const tronAdapter = new TronAdapter()

// AppKit 作成
const modal = createAppKit({
  projectId,
  adapters: [wagmiAdapter, solanaAdapter, bitcoinAdapter, tronAdapter],
  networks: [mainnet, sepolia],
  metadata: {
    name: 'My dApp',
    description: 'My dApp with Openloop',
    url: 'https://my-dapp.com',
    icons: ['https://my-dapp.com/icon.png'],
  },
  customWallets: [
    {
      id: 'openloop-connect',
      name: 'Openloop Connect',
      homepage: 'https://crypto.haudi.jp/openloop/',
      mobile_link: 'openloop://',
      desktop_link: 'openloop://',
      image_url: 'https://crypto.haudi.jp/openloop/images/logo/openloop-icon.png',
    },
  ],
})
```

WcTransport vs AppKit の使い分け

比較項目	WcTransport	AppKit
APDU レベル制御	✓ 完全制御	✗ 抽象化済み
導入の容易さ	やや複雑	簡単
UI コンポーネント	なし（自前で実装）	モーダル付き
マルチチェーン	手動切替	自動切替
EIP-7702	✓ 対応	✗ 非対応
対応ウォレット	Openloop Connect のみ	全 WC 対応ウォレット

推奨: - Openloop デバイスとの統合が主目的 → **WcTransport** - 複数ウォレットをサポートしたい → **AppKit**

制限事項

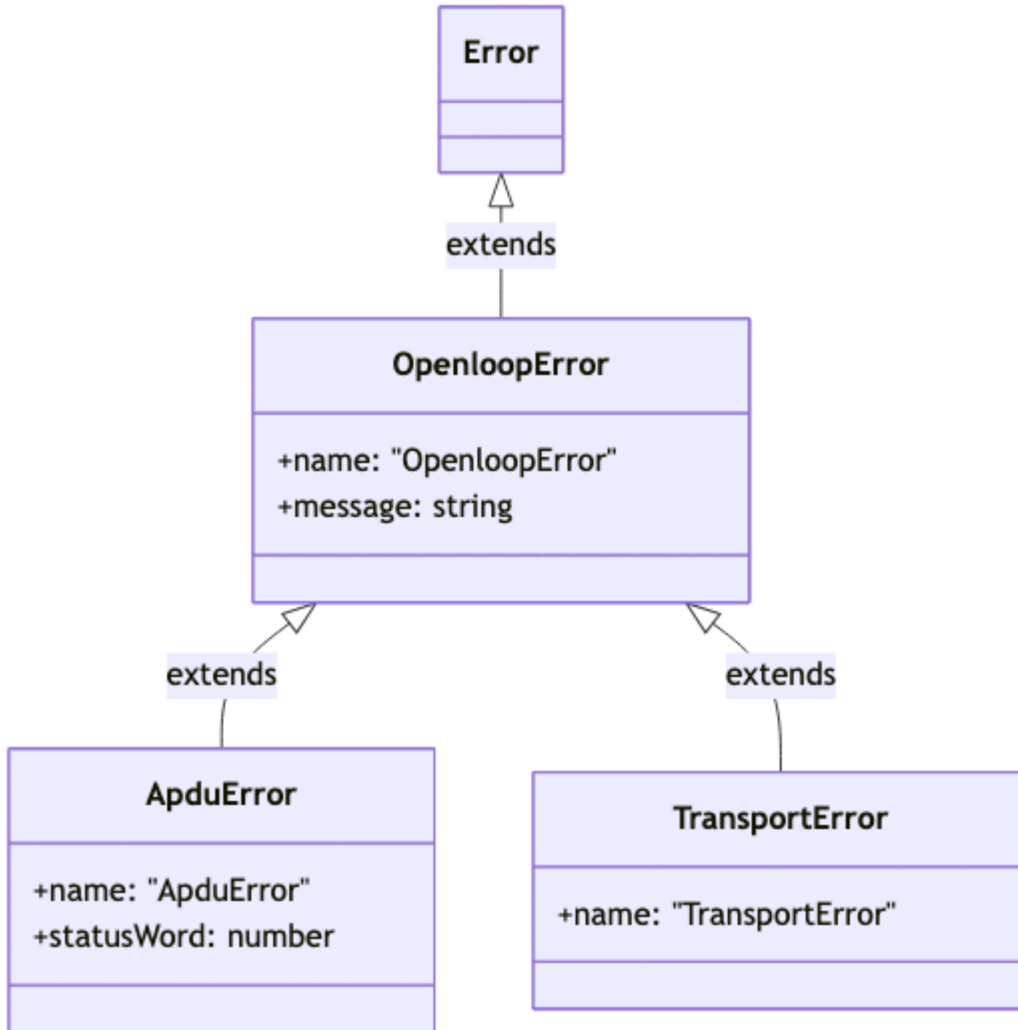
- WalletConnect は常にリモートの Openloop Connect アプリを経由するため、デバイスが Connect アプリに接続されている必要がある
- EIP-7702 Authorization は WcTransport 経由でのみ対応（AppKit では非対応）
- XRP は現在 WalletConnect では非対応
- WcTransport の `exchange` タイムアウトは WalletConnect リレーの制約に依存

次のステップ

- エラー処理
- API リファレンス

エラー処理ガイド

エラークラス階層



インポート

```
import { OpenloopError, ApduError, TransportError } from '@openloop/sdk-core'
```

ApduError

デバイスがエラーのステータスワード（`0x9000` 以外）を返した場合にスローされます。

```
class ApduError extends OpenloopError {
  readonly statusWord: number  // 例: 0x6985
}
```

APDU ステータスワード表

ステータスワード	定数名	説明	よくある原因
`0x9000`	—	成功	—
`0x6985`	User rejected	ユーザーがデバイスで拒否	ユーザーが拒否ボタンを押した
`0x6a80`	Invalid data	無効なデータ	不正な BIP44 パスやトランザクション
`0x6a82`	App not found	アプリが見つからない	デバイスで対応アプリが開いていない
`0x6d00`	Instruction not supported	命令未サポート	デバイスのファームウェアバージョンが古い
`0x6e00`	CLA not supported	CLA 未サポート	不正なクラスバイト
`0x6f00`	Internal error	内部エラー	デバイス内部の予期しないエラー
`0x61XX`	More data	続きデータあり	PSBT 取得時の中間レスポンス（エラーではない）

使用例

```
import { ApduError } from '@openloop/sdk-core'

try {
  const sig = await eth.signPersonalMessage(DEFAULT_ETH_PATH, 'Hello')
} catch (err) {
  if (err instanceof ApduError) {
    if (err.statusWord === 0x6985) {
      console.log('ユーザーがデバイスで署名を拒否しました')
    } else {
      console.log(`APDU エラー: 0x${err.statusWord.toString(16)}`)
    }
  }
}
```

TransportError

Transport レイヤー（USB/BLE/WebSocket 等）で通信エラーが発生した場合にスローされます。

```
class TransportError extends OpenloopError {
  // message にエラー詳細が含まれる
}
```

よくある TransportError

メッセージ	原因	対処法
`WalletConnect transport not connected`	WcTransport が未接続	`open()` を呼んでから使用する
`Transport type "..." not registered`	OpenloopSDK に未登録の Transport	`registerTransport()` で登録する
`No available transport found`	利用可能な Transport がない	Transport の登録と環境を確認
`Device disconnected`	デバイスが切断された	再接続を試みる
`Exchange timeout`	APDU 応答タイムアウト	デバイスの状態を確認（承認待ちかもしれない）

エラーハンドリングパターン

基本パターン

```
import { OpenloopError, ApduError, TransportError } from '@openloop/sdk-core'

try {
  const { address } = await eth.getAddress(DEFAULT_ETH_PATH)
} catch (err) {
  if (err instanceof ApduError) {
    // デバイスからのエラー応答
    switch (err.statusWord) {
      case 0x6985:
        showMessage('デバイスで拒否されました')
        break
      case 0x6a80:
        showMessage('無効なデータです')
        break
      default:
        showMessage(`デバイスエラー: ${err.message}`)
    }
  } else if (err instanceof TransportError) {
    // 通信エラー
    showMessage('デバイスとの接続が切れました。再接続してください。')
  } else if (err instanceof OpenloopError) {
    // その他の SDK エラー
    showMessage(`エラー: ${err.message}`)
  } else {
    // 予期しないエラー
    throw err
  }
}
```

再接続付きリトライパターン

```
async function withRetry<T>(
  fn: () => Promise<T>,
  reconnect: () => Promise<void>,
  maxRetries: number = 1
): Promise<T> {
  for (let i = 0; i <= maxRetries; i++) {
    try {
      return await fn()
    } catch (err) {
      if (err instanceof TransportError && i < maxRetries) {
        await reconnect()
        continue
      }
      throw err
    }
  }
  throw new Error('Unreachable')
}

// 使用例
const { address } = await withRetry(
  () => eth.getAddress(DEFAULT_ETH_PATH),
  async () => {
    transport = await WebHidTransport.reconnect()
    eth = new EthereumApp(transport!)
  }
)
```

ユーザー拒否の判定

```
function isUserRejected(err: unknown): boolean {
  return err instanceof ApduError && err.statusWord === 0x6985
}

try {
  const sig = await eth.signTransaction(DEFAULT_ETH_PATH, rawTx)
} catch (err) {
  if (isUserRejected(err)) {
    // ユーザーの意図的な操作 — エラー表示不要
    return
  }
  // その他のエラーは表示
  showError(err)
}
```

次のステップ

- API リファレンス — 全型定義一覧

API リファレンス

全型定義・インターフェース・クラス・定数の一覧です。すべて `@openloop/sdk-core` からエクスポートされています。

目次

- インターフェース
- App クラス
- エラークラス
- OpenloopSDK クラス
- WcTransport クラス
- 定数
- ユーティリティ関数

インターフェース

ITransport

デバイスとの通信を抽象化するインターフェース。全 Transport が実装します。

```
interface ITransport {
  open(): Promise<void>
  close(): Promise<void>
  isConnected(): boolean
  exchange(apdu: Uint8Array): Promise<Uint8Array>
  lock(): Promise<void>
  unlock(): Promise<void>
}
```

メソッド	説明
`open()`	デバイスへの接続を開く
`close()`	接続を閉じる
`isConnected()`	接続中かどうか
`exchange(apdu)`	APDU コマンドを送信してレスポンスを受信
`lock()`	デバイスの排他アクセスを取得（マルチステップ操作用）
`unlock()`	排他アクセスを解放

TransportStatus

```
interface TransportStatus {
  connected: boolean
  deviceName?: string
  deviceId?: string
}
```

TransportEvent

```
type TransportEventType = 'connected' | 'disconnected' | 'error'

interface TransportEvent {
  type: TransportEventType
  error?: Error
}

type TransportEventHandler = (event: TransportEvent) => void
```

TransportInfo

`OpenloopSDK.discover()` が返す Transport 情報。

```
interface TransportInfo {
  type: string          // 登録名（例: 'webhid'）
  name: string          // 表示名（例: 'WebHID (USB)'）
  available: boolean    // 利用可能か
}
```

TransportFactory

```
type TransportFactory = () => Promise<ITransport>
```

SignatureResult

Ethereum / TRON の ECDSA 署名結果。

```
interface SignatureResult {
  v: number    // リカバリー ID
  r: string    // R 値 (32 bytes hex, 0x プレフィックスなし)
  s: string    // S 値 (32 bytes hex, 0x プレフィックスなし)
}
```

BtcMessageSignature

Bitcoin BIP-137 メッセージ署名結果。

```
interface BtcMessageSignature {
  v: number       // リカバリー ID (35-38: P2WPKH native SegWit)
  r: string       // R 値 (32 bytes hex)
  s: string       // S 値 (32 bytes hex)
  signature: string // 完全な署名 (V || R || S, 65 bytes) の Base64
}
```

InputSigningPath

Bitcoin PSBT の入力ごとの BIP32 パス指定。

```
interface InputSigningPath {
  index: number   // PSBT の入力インデックス
  path: string    // BIP32 パス（例: "84'/1'/0'/0/5"）
}
```

DeviceInfo

```
interface DeviceInfo {
  connected: boolean
  path?: string
  vendorId?: number
  productId?: number
}
```

XrpSignatureResult

XRP 署名結果。

```
interface XrpSignatureResult {
  signature: string  // 署名 hex (Ed25519: 64B, secp256k1: DER 可変長)
}
```

XrpCurve

```
type XrpCurve = 'ed25519' | 'secp256k1'
```

WcSession

```
interface WcSession {
  topic: string
}
```

IWcSignClient

```
interface IWcSignClient {
  request<T>(params: {
    topic: string
    chainId: string
    request: { method: string; params: unknown }
  }): Promise<T>
}
```

WcTransportOptions

```
interface WcTransportOptions {
  client: IWcSignClient
  session: WcSession
  chainId?: string  // デフォルト: "eip155:1"
}
```

App クラス

EthereumApp

```
class EthereumApp {
  constructor(transport: ITransport)

  getAddress(path: string): Promise<{ publicKey: string; address: string }>

  signPersonalMessage(path: string, message: string, chainId?: number): Promise<SignatureResult>

  signTransaction(path: string, rawTx: string): Promise<SignatureResult>

  signTypedData(
    path: string, domainSeparatorHash: string, messageHash: string, chainId?: number
  ): Promise<SignatureResult>

  signAuthorization(path: string, authorizationRlp: string): Promise<SignatureResult>
}
```

BitcoinApp

```
class BitcoinApp {
  constructor(transport: ITransport)

  getAddress(testnet?: boolean): Promise<{ publicKey: string; address: string; chainCode: string

  getAddressWithPath(path: string): Promise<{ publicKey: string; address: string; chainCode: str

  getAccountXpub(coinType: number): Promise<{ publicKey: string; chainCode: string }>

  signMessage(message: string, path: string): Promise<BtcMessageSignature>

  signPsbt(psbt: Uint8Array | string): Promise<Uint8Array>

  signPsbtHex(psbt: string): Promise<string>

  signPsbtWithPaths(psbt: string, inputPaths: InputSigningPath[]): Promise<string>
}
```

SolanaApp

```
class SolanaApp {
  constructor(transport: ITransport)

  getAddress(path: string): Promise<{ publicKey: string; address: string }>

  signTransaction(path: string, txBytes: Uint8Array): Promise<string>

  signOffchainMessage(path: string, messageBytes: Uint8Array): Promise<string>
}
```

TronApp

```
class TronApp {
  constructor(transport: ITransport)

  getAddress(path: string): Promise<{ publicKey: string; address: string }>

  signTransaction(path: string, rawDataHex: string): Promise<SignatureResult>

  signMessage(path: string, message: string): Promise<SignatureResult>
}
```

XrpApp

```
class XrpApp {
  constructor(transport: ITransport)

  getAddress(path: string, curve?: XrpCurve): Promise<{ publicKey: string; address: string }>

  signTransaction(path: string, txBlob: string): Promise<XrpSignatureResult>
}
```

エラークラス

```
class OpenloopError extends Error {
  name: 'OpenloopError'
}

class ApduError extends OpenloopError {
  name: 'ApduError'
  readonly statusWord: number
  constructor(statusWord: number)
}

class TransportError extends OpenloopError {
  name: 'TransportError'
  constructor(message: string)
}
```

OpenloopSDK クラス

Transport の登録・検出・接続を管理するクラス。全メソッドは static です。

```
class OpenloopSDK {
  static registerTransport(type: string, config: {
    factory: TransportFactory
    name: string
    isAvailable: () => boolean | Promise<boolean>
  }): void

  static discover(): Promise<TransportInfo[]>

  static connect(opts?: { transport?: string }): Promise<ITransport>

  static getRegisteredTransports(): string[]

  static clearTransports(): void
}
```

WcTransport クラス

```
class WcTransport implements ITransport {
  constructor(options: WcTransportOptions)
  open(): Promise<void>
  close(): Promise<void>
  isConnected(): boolean
  lock(): Promise<void>
  unlock(): Promise<void>
  exchange(apdu: Uint8Array): Promise<Uint8Array>
}
```

定数

USB 識別子

```
const OPENLOOP_VENDOR_ID  = 0x303a   // Espressif VID
const OPENLOOP_PRODUCT_ID = 0x8341   // Openloop PID
const LEDGER_VENDOR_ID    = 0x2c97   // Ledger SAS VID
const LEDGER_PRODUCT_ID   = 0x1011   // Nano S 互換
```

BIP44 デフォルトパス

```
const DEFAULT_ETH_PATH  = "44'/60'/0'/0/0"
const BTC_MAINNET_PATH  = "84'/0'/0'/0/0"
const BTC_TESTNET_PATH  = "84'/1'/0'/0/0"
const DEFAULT_SOL_PATH  = "44'/501'/0'/0'"
const DEFAULT_TRON_PATH = "44'/195'/0'/0/0"
const DEFAULT_XRP_PATH  = "44'/144'/0'/0'/0'"
```

BLE 定数

```
const BLE_SERVICE_UUID      = '13d63400-2c97-0004-0000-4c6564676572'
const BLE_NOTIFY_UUID       = '13d63400-2c97-0004-0001-4c6564676572'
const BLE_WRITE_UUID        = '13d63400-2c97-0004-0002-4c6564676572'
const BLE_DEVICE_NAME_PREFIX = 'Openloop'
const BLE_DEFAULT_MTU       = 155
```

WalletConnect 定数

```
const WALLETCONNECT_PROJECT_ID = 'f43a098a34fa6d741ffa88b5ce738113'
```

WalletConnect 対応チェーン (CAIP-2)

```
const SUPPORTED_EVM_CHAINS = [
  'eip155:1',         // Ethereum Mainnet
  'eip155:11155111',  // Sepolia Testnet
] as const

const SUPPORTED_BIP122_CHAINS = [
  'bip122:000000000019d6689c085ae165831e93', // Bitcoin Mainnet
  'bip122:000000000933ea01ad0ee984209779ba', // Bitcoin Testnet3
] as const

const SUPPORTED_SOLANA_CHAINS = [
  'solana:5eykt4UsFv8P8NJdTREpY1vzqKqZKvdp', // Mainnet Beta
  'solana:EtWTRABZaYq6iMfeYKouRu166VU2xqa1',  // Devnet
] as const

const SUPPORTED_TRON_CHAINS = [
  'tron:0x2b6653dc',  // TRON Mainnet
  'tron:0x94a9059e',  // TRON Shasta Testnet
] as const
```

WalletConnect 対応メソッド

```
const SUPPORTED_EVM_METHODS = [
  'eth_sendTransaction', 'eth_signTransaction', 'eth_sign',
  'personal_sign', 'eth_signTypedData', 'eth_signTypedData_v3',
  'eth_signTypedData_v4', 'wallet_getCapabilities',
] as const

const SUPPORTED_BIP122_METHODS = [
  'signPsbt', 'getAccountAddresses', 'signMessage', 'sendTransfer',
] as const

const SUPPORTED_SOLANA_METHODS = [
  'solana_signTransaction', 'solana_signAllTransactions',
  'solana_signAndSendTransaction', 'solana_signMessage',
] as const

const SUPPORTED_TRON_METHODS = [
  'tron_signTransaction', 'tron_signMessage',
] as const

const SUPPORTED_EVENTS = ['chainChanged', 'accountsChanged'] as const
```

APDU 命令セット

```
const APDU = {
  // クラスバイト
  CLA_ETHEREUM: 0xe0,          // Ledger 互換
  CLA_OPENLOOP: 0xf0,          // Openloop 独自

  // 共通 INS（coin_type で自動ルーティング）
  INS_GET_PUBLIC_KEY: 0x02,     // アドレス/公開鍵取得
  INS_SIGN_TX: 0x04,            // トランザクション署名
  INS_SIGN_MESSAGE: 0x08,      // メッセージ署名
  INS_SIGN_EIP712: 0x0c,       // EIP-712 Typed Data
  INS_SIGN_AUTHORIZATION: 0x34, // EIP-7702

  // Bitcoin PSBT（Openloop 独自）
  INS_BTC_SIGN_PSBT: 0x70,           // PSBT 送信
  INS_BTC_GET_SIGNED_PSBT: 0x71,     // 署名済み PSBT 取得
  INS_BTC_GET_WALLET_PUBLIC_KEY: 0x40, // BTC 公開鍵/アドレス取得
  INS_BTC_GET_ACCOUNT_XPUB: 0x72,    // アカウント xpub 取得
  INS_BTC_SIGN_MESSAGE: 0x73,        // BTC メッセージ署名

  // Solana（Ledger Solana 互換）
  INS_SOL_GET_PUBKEY: 0x05,          // Ed25519 公開鍵取得
  INS_SOL_SIGN_MESSAGE: 0x06,        // トランザクション/メッセージ署名
  INS_SOL_SIGN_OFFCHAIN_MSG: 0x07,  // オフチェーンメッセージ署名
} as const
```

ユーティリティ関数

```
// APDU コマンド構築
function buildApdu(cla: number, ins: number, p1: number, p2: number, data?: Uint8Array): Uint8A

// BIP32 パスをバッファに変換
function pathToBuffer(path: string): Uint8Array

// BIP32 パスを要素配列に変換
function pathToElements(path: string): number[]

// Uint8Array の結合
function concat(...arrays: Uint8Array[]): Uint8Array

// hex 文字列を Uint8Array に変換（0x プレフィックスあり/なし対応）
function hexToBytes(hex: string): Uint8Array

// Uint8Array を hex 文字列に変換（0x プレフィックスなし）
function bytesToHex(bytes: Uint8Array): string

// Base58 エンコード/デコード
function base58Encode(bytes: Uint8Array): string
function base58Decode(str: string): Uint8Array
```