Save Design 使用ガイド - 目次

1. はじめに

1.1 購入のお礼とスクリプトリファレンスの案内

このたびは **Save Design** をご購入いただき、誠にありがとうございます。 本アセットが、あなたのプロジェクトにおけるセーブ機能の実装・保守を少しでも快適にすることを願っています。

このドキュメントでは、**Save Design の基本的な使い方**や導入の流れについて説明します。 各 API の詳細仕様や属性の使い方、インターフェース定義などのより技術的な情報については、 別途ご用意している **「スクリプトリファレンス」** をあわせてご参照ください。

> スクリプトリファレンスは、`Documentation/save-design-script-reference.pdf` に同梱されています。 使用方法とあわせてお読みいただくことで、よりスムーズにご活用いただけます。

1.2 Save Design とは（特徴と仕組みの概要）

Save Design は、Unity 向けに設計されたセーブデータ管理フレームワークです。 最大の特徴は、**キー文字列を一切使わずにセーブデータへアクセスできること**です。

開発者は `[SharedData]` や `[SlotData]` などの属性を使って、セーブ対象のクラスを明示的に定義します。 それに対して Save Design が **静的 API を自動生成**することで、 IDE 補完で型安全にセーブ／ロード／初期化を行えるようになります。

🔧 Save Design の構成要素

要素	説明
属性によるデータ定義	`[SharedData]`, `[SlotData]`, `[TempData]` などでセーブ対象を分類
静的 API の自動生成	`SD.Save.Slot(...)` や `SD.Load.Shared()` などの関数が自動的に生成される
補完・リネームに強い	すべての処理が型情報に基づいており、キーの書き間違いや更新漏れを防止
導入が簡単	属性を付けるだけで導入でき、初期化や設定も最小限
暗号化や非同期処理に対応（オプション）	必要に応じて AES + HMAC 暗号化や `async/await` による非同期処理も導入可能

🎯 Save Design の狙い

- データ構造が複雑になっても **整理されたセーブ設計を維持**したい
- セーブ処理の実装や保守を **型安全に・シンプルに管理**したい

- 複数スロットや一時的な状態など、**現実的なセーブ要件を柔軟に対応**したい

このようなニーズを持つ開発者にとって、Save Design は「**セーブ周りの面倒ごとを消してくれる**」頼れるツールです。

---

2. 最小限の導入ステップ

2.1 必要な準備（Unityバージョン／インストール）

✅ 対応 Unity バージョン

- **Unity 2022.3 LTS 以降** を推奨
- .NET Standard 2.1 相当の API が使用可能なバージョンである必要があります

📦 パッケージのインポート

Save Design は Unity のパッケージとして配布されており、 インポート後は `Assets/Plugins/Save Design/` フォルダ配下に以下の構成が展開されます：

```
Assets/Plugins/Save Design/
├── Runtime/         // Save Design の動作に必要なソースコード
├── Editor/          // エディタ拡張（セットアップ、暗号化設定・コード生成）
├── Samples/         // 使用例とサンプルシーン
└── Documentation/   // スクリプトリファレンスと使用ガイド（このファイル）
```

🔧 Save Design のセットアップをエディタから簡単に行う

Save Design には、初期セットアップを簡略化するための **専用エディタツール**が用意されています。 スクリプトの雛形生成から設定アセットの作成まで、数クリックで完了できます。

🔹 スクリプトの自動生成

1. Unity のメニューから **Tools > Save Design > Setup** を選択し、**Save Design Setup ウィンドウ**を開きます。
2. `Namespace` と `Output Path` を入力したら、**Generate scripts** ボタンをクリックします。
3. 指定したパスに以下のスクリプトが自動生成されます：
    - `SaveDesignRoot` 属性が付与された SD クラス（ルートクラス）
    - `ISaveDesignConfig` を実装した SaveDesignConfig クラス（設定用）

これにより、Save Design を使うための最低限の構成がすぐに整います。

🔹 設定アセットの作成

次に、Unity メニューから **Tools > Save Design > Create Save Design Config** を選択します。

この操作により、`Assets/Resources` フォルダ内に `SaveDesignConfig` アセットが自動生成されます。 このアセットでは以下のような **ファイルに関する設定**を行うことができます：

- セーブフォルダ名
- 共有データ／スロットデータのファイル名
- 拡張子

設定が完了すれば、あとは `[SharedData]` や `[SlotData]` を付けたデータクラスを定義するだけで、 Save Design の仕組みをすぐに使い始めることができます。

このセットアップ手順はプロジェクトごとに一度だけ実行すれば十分です。 以降は自動生成された API を通じて、IDE補完つきで安全かつ簡潔にセーブ処理を実装できます。

⚠️ 不要なサンプルを削除してください

Save Design には、導入直後の動作確認や使い方の参考として、簡単な **サンプルクラス**や**サンプルシーン**が同梱されています。 しかし、これらは実際のゲームには不要であることが多く、**そのままにしておくと最終ビルドにも含まれてしまう可能性があります **。

開発に必要な最小構成だけを残し、本番環境に不要なデータを含めないようにしましょう。

🧩 任意で導入できるライブラリ

UniTask（非同期処理）

- 使用したい場合は、UniTask を導入してください
- `SAVE_DESIGN_SUPPORT_UNITASK` シンボルを定義することで、UniTask ベースの非同期APIが自動生成されます

MessagePack for C#（高速シリアライザ）

- 使用したい場合は、MessagePack for C# を導入してください
- ルートクラスに `[SaveDesignRoot(SerializerType.MessagePack)]` を付けることで対応可能です

> これらのライブラリは Save Design に**同梱されていません**。 必要に応じて各自導入し、ライセンスに従って使用してください。

2.2 データ定義クラスの作成

Save Design では、セーブ対象のデータをクラスとして定義し、 そこに `[SharedData]` や `[SlotData]` などの属性を付けて分類します。

この「データ定義」によって、Save Design は自動的にセーブ用 API を生成できるようになります。

🔹 データの種類と使い分け

属性名	用途
`[SharedData]`	全セーブスロット共通のプレイヤー設定や進行状況に
`[SlotData]`	スロットごとに異なるセーブ内容（キャラ情報、所持品など）に
`[TempData]`	保存しない一時的なデータ（セッション内フラグなど）に
`[SlotMetaData]`	スロット表示用のメタ情報（保存日時、プレイ時間など）に

> 属性の詳細な使い方や構造は、スクリプトリファレンスの「2. 属性」セクションをご参照ください。

🧩 例：スロットごとのプレイヤーデータ

```
[SlotData]            // セーブスロットごとに分けるデータは SlotData 属性
[System.Serializable] // JsonUtility でシリアライズするために必要
public class Player
{
    public int level;
    public float hp;
    public Vector3 position;
}
```

上記のようなクラスを定義するだけで、自動生成された API 経由で下記のようにこのデータを保存・読み込みできるようになります。

```
SD.Load.Slot(slotIndex);          // 読み込み

var level = SD.Slot.Player.level; // 値の取得

SD.Slot.Player.level++;           // 値の変更

SD.Save.Slot(slotIndex);          // 保存
```

> 💡 **注意：フィールドの保存対象は使用するシリアライザーに依存します**
>
> Save Design では、データクラスの内容を実際に保存・復元する処理は、JsonUtility や MessagePack for C# などの **外部シリアライザーによって行われます**。 そのため、「どのフィールドが保存されるか」は選択したシリアライザーの仕様に従います。
>
> たとえば `UnityEngine.JsonUtility` を使用する場合：
>
> - `public` フィールドは保存されます
> - `private` フィールドでも `[SerializeField]` が付いていれば保存されます
> - プロパティや `[NonSerialized]` 属性が付いたフィールドは保存されません
>
> 一方、`MessagePack for C#` を使用する場合は `[MessagePackObject]` と `[Key(n)]` 指定が必要です。

> 意図したデータが正しく保存・復元されるように、各シリアライザーの仕様を確認のうえ設計してください。

---

3. 実用的な使い方例

3.1 共有データの読み書きについて

`[SharedData]` を付与したデータは、すべてのセーブスロットに共通してアクセスされる「共有データ」として扱われます。 ユーザー設定やゲーム全体の進行状況など、スロットに依存しない情報の保存に適しています。

✅ 読み込みの推奨タイミングと方法

共有データは、ゲーム起動時に一度だけ読み込んでおき、以降は常駐データとして扱うのが一般的です。 この読み込み処理は、ルートクラスに以下のような静的初期化関数を実装する形で行うことを推奨します：

```
[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
static void InitSaveDesignConfig()
{
    // ファイル関連の設定を事前に行う
    config = Resources.Load<SaveDesignConfig>("SaveDesignConfig");

    // 共有データの読み込みを行い、データが無ければ初期化する
    if (!Load.Shared()) Initialize.Shared();
}
```

この処理により、**どのシーンよりも早い段階で共有データが準備される**ため、 後続の処理が安心して `SD.Shared.xxx` にアクセスできる状態になります。

💾 保存のタイミングと設計方針

共有データの保存タイミングについては、**変更が発生した直後に保存する方式**と、**ゲーム終了時にまとめて保存する方式** の2種類があります。 どちらも使用可能ですが、信頼性と意図の明確さの観点から、**基本的には前者（即時保存）を推奨**します。

✅ 方式①：値が変更された直後に保存する（推奨）

```
SD.Shared.settings.volume = newVolume;
SD.Save.Shared();
```

この方法は最も安全で、アプリケーションが予期せず終了した場合にも、**直前の変更が確実に保存されている状態を保てます**。 とくに設定や進行状況など、後で復元が必要になる情報にはこの方式を使うことを推奨します。

⚠️ **方式②：アプリケーション終了時にまとめて保存する（補助的）**

```
[RuntimeInitializeOnLoadMethod]
private static void RegisterSaveOnExit()
{
    Application.quitting += () => SD.Save.Shared();
}
```

この方法は記述が簡単で漏れがなく、**軽量なプロジェクトや頻繁に変更されない設定項目**に向いています。 ただし、 `Application.quitting` はあくまで Unity が「正常終了」と判断した場合にしか呼び出されないため、 **タスクマネージャーでの強制終了や、モバイルOSによるプロセス終了などでは実行されない可能性があります**。

3.2 スロットセーブとオートセーブ

Save Design では、典型的な「スロット1〜3」のように番号で分けられた**スロットごとのセーブデータ**を柔軟に扱うことができます。

そして、番号だけではなく**識別子（文字列）による保存**にも対応しています。

たとえば、タイトル画面でプレイヤーがスロットを選択した場合には番号を使ってデータを読み込み、 ゲームプレイ中のチェックポイントなど自動保存用途では `"autosave"` や `"checkpoint-3"` のような文字列を使うと便利です。

ただし、Save Design は自動的にデータを保存するわけではありません。 これは意図的な設計であり、開発者が意図しないタイミングでデータが書き換わるといった 予期せぬトラブルやデバッグ困難な状態を防ぐためです。

Save Design は、「保存したいときに、明示的に保存する」シンプルなルールに従って動作します。 裏で勝手に動作したり、開発者が気づかないタイミングで I/O が発生することはありません。 そのため、**Save Design の仕組みを意識せずにプロジェクト全体を安全に保つことができます**。

この設計方針により、開発者は「今、どのデータが保存されているのか」「いつ保存されたのか」を常に把握でき、 セーブ処理をコントロールしやすく、バグの発生原因も明確になります。

3.3 SlotMetaData の活用と注意点

`SlotMetaData` は、セーブスロットの一覧画面や選択画面などに表示するための、**スロット情報のメタデータ**を保持するための仕組みです。 プレイヤー名やプレイ時間、保存日時など、ゲーム内容とは直接関係しないが UI 表示に必要な情報を格納するのに適しています。

セーブスロットを使わないようなゲームでは `SlotMetaData` をあえて定義しないことも可能です。

✅ SlotMetaData の用途例

- セーブスロット選択画面での「プレイヤー名」「現在の章」「プレイ時間」の表示
- 「前回保存日時」の記録

- スロットが空かどうかを判断するためのフラグ

`SlotMetaData` のデータは `SD.Save.Slot(...)` でスロットデータを保存するときに一緒に保存されます。
`SD.Load.SlotMeta(...)` で読み込んで表示や分岐に使用します。

⚠️ 注意：SlotMetaData は常に新規インスタンスで保存される

SlotMetaData は他のデータ（SharedData や SlotData）とは異なり、**既存のデータに変更を加えて保存するという使い方ができません。 毎回新しいインスタンスを生成し、すべてのフィールドを書き込み前に明示的にセットする必要があります。**

たとえば、前回読み込んだ SlotMetaData を取得して一部のフィールドだけ変更して保存しようとしても反映されません。

```
if (SD.Load.SlotMeta(slotIndex, out var meta))
{
    meta.value += 100; // ❌ メタ情報の値を書き換えても保存されない
}
```

この設計は、**メタ情報は他のセーブデータの値から自動的に生成されるべき**という考えに基づいています。

SlotMetaData は `IBeforeSaveCallback` を使って下記のように実装することを推奨します。

```
[SlotMetaData, System.Serializable]
public class SlotMetaData : IBeforeSaveCallback
{
    public string playerName;
    public int level;
    public int money;

    void IBeforeSaveCallback.OnBeforeSave()
    {
        // 他のデータの値を使ってフィールドを初期化する
        var player = SD.Slot.Player;
        playerName = player.name;
        level = player.level;
        money = player.money;
    }
}
```

3.4 TempData を使った一時的な状態管理

`TempData` は、セーブファイルとして保存されない **一時的な状態**を管理するための仕組みです。 一度ゲームを終了したり、スロットを切り替えたりすると自動的に破棄されるため、「このセッション中だけ覚えておきたい値」「ロードし直したらリセットしたいフラグ」のような場面に最適です。

たとえば、以下のような用途に `TempData` は活用できます：

- 起動後のチュートリアル表示済みフラグ
- 一時的なアイテム取得状態やダイアログ進行状況
- バトル中の一時的なリザルト、スコア履歴
- ロード直後にだけ発火する一度きりの処理フラグ

`TempData` を定義するには、通常のデータクラスに `[TempData]` 属性を付けるだけです。 さらに `TempDataResetTiming` を指定することで、**どのタイミングでリセットされるか**を細かく制御できます。

たとえば、`OnGameStart` を指定すればゲームの起動時に、 `OnSlotDataLoad` を指定すればスロットを切り替えたときにリセットされます。

この仕組みにより、「初期化し忘れて前回の状態が残ってしまう」といった**ありがちなバグを防止**でき、 一時的な状態管理をより安全かつ簡潔に実装できます。

なお、デフォルトのリセットタイミングは `TempDataResetTiming.OnSharedDataLoad` です。

3.5 初期化処理の活用方法

Save Design では、新規ゲーム開始時やセーブデータが存在しない状態での開始処理として、 `Initialize` を使った **明示的な初期化**が可能です。

初期化には `SD.Initialize.Shared()` と `SD.Initialize.Slot()` の2種類があり、 それぞれ共有データ／スロットデータに対して、空のインスタンスを生成します。

主に以下のようなタイミングで使用します：

- **「はじめから」や「ニューゲーム」** など、完全に新しいセーブデータを作成する場面
- **タイトル画面から新規スロットを作る**とき
- **テスト時にデータを一度リセットしたい**とき

この処理を呼ぶと、既存のセーブデータ（該当スロットや共有データ）は上書きされます。 そのため、**まだデータが保存されていない状態では実行しない**よう注意してください。

また、初期化時に特定の処理を走らせたい場合は、対象のデータクラスに `IAfterInitializeCallback` インターフェースを実装することで、 **インスタンス生成直後に一度だけ** `OnAfterInitialize()` が呼ばれます。

このコールバックは、たとえば以下のような用途に向いています：

- フィールドの初期値を設定する（セーブ対象外の値も含めて）
- 他のデータ構造を参照して初期状態を構築する
- チュートリアルや開始ステージなどを動的に設定する

これにより、**新規開始時だけに発生する特殊な初期処理**を安全に分離でき、 再ロード時には実行されないという保証が得られます。

ゲームの状態を「明示的に初期化」できるということは、 セーブフローにおけるバグを大幅に減らし、テストもしやすくなるという大きな利点につながります。

> `IAfterInitializeCallback` は `[SharedData]` と `[SlotData]` のどちらかを付与したクラスでのみ処理されます。

> 詳細な仕様は、スクリプトリファレンスの「1.2 IAfterInitializeCallback」をご参照ください。

3.6 読み書き時のコールバックの活用方法

Save Design では、セーブやロードの直前／直後に処理を挟みたい場合に、 `IBeforeSaveCallback` および `IAfterLoadCallback` という2つのインターフェースが利用できます。 これにより、**セーブ対象のデータクラスごとに、任意の前後処理を明確に分離**して記述できます。

✅ `IAfterLoadCallback` の活用例

このインターフェースを実装したデータクラスでは、 セーブデータをロードした直後に `OnAfterLoad()` が自動的に呼び出されます。

この仕組みは以下のような処理に便利です：

- ロード後に初期化が必要なランタイム専用フィールドの再構築
- キャッシュや辞書の再構成（例：ID → オブジェクト辞書）
- サウンド／UIの反映など、見た目に関わるデータの更新トリガー

✅ `IBeforeSaveCallback` の活用例

セーブ直前に `OnBeforeSave()` が呼ばれることで、**保存内容を意図的に整える処理**が記述できます。

たとえば：

- 使用中のオブジェクトの状態を保存用構造に変換
- 逆参照や不要なデータを除外
- 最新のタイムスタンプやプレイ時間を更新

このように、データ整形やログ記録などを事前に実行することで、**より正確で安全なセーブ内容を保証**できます。

💡 なぜコールバックが便利なのか？

コールバックを使えば、「セーブ処理を呼ぶ場所」や「ロード結果を渡す処理」に直接処理を書かなくて済みます。 その結果、**セーブ処理の呼び出し元は常にシンプルな1行のままに保たれ、** 細かな処理ロジックはそれぞれのデータクラスに閉じ込めておけるというメリットがあります。

🔔 他のデータに依存する場合の注意

Save Design では、複数のデータ（ `SharedData` , `SlotData` , `TempData` など）を別々のクラスとして定義できますが、 **データの初期化や読み書きの順序は、特別な指定をしない限り保証されません。**

そのため、コールバック（`IAfterLoadCallback`, `IBeforeSaveCallback`, `IAfterInitializeCallback` など）の中で**他のデータの値を参照したい場合には注意が必要です**。

❌ 順序が保証されていない場合に起きる問題の例

たとえば、`GameSettings`（SharedData）内で、`Profile`（SharedData）の情報を参照して初期設定を行いたいとします：

```
[SharedData, System.Serializable]
public class GameSettings : IAfterLoadCallback
{
    public int volume;

    void IAfterLoadCallback.OnAfterLoad()
    {
        volume = SD.Shared.Profile.defaultVolume; // ← ここでエラーになる可能性
    }
}
```

このとき、`Shared.Profile` がまだ読み込まれていなければ `NullReferenceException` が発生するか、 意図しない初期値が使用されてしまう恐れがあります。

✅ Save Design の解決策：依存関係の明示

Save Design では、データ定義属性に `Type` を渡すことで **依存関係を明示的に宣言**できます。

```
[SharedData(typeof(Profile))]
public class GameSettings : IAfterLoadCallback
{
    ...
}
```

このように記述すると、Save Design は `GameSettings` の読み込みや初期化を 必ず `Profile` よりも後に実行するよう処理順を調整します。

> ✅ 依存先のクラスも同じデータ種別である必要があります（例：SharedData → SharedData）
> ✅ `[TempData]` の場合は `ResetTiming` も一致させる必要があります

この仕組みにより、**コールバック内で他のデータに安全にアクセスできる**ようになり、 複雑な依存関係を持つプロジェクトでも安心してデータ設計が行えます。

3.7 非同期処理の導入（UniTask/await 対応）

Save Design では、データの読み書きを **非同期で行う API** も自動生成されます。 セーブやロードが長時間かかる可能性がある場合（モバイル端末・大量データ・バックグラウンド保存など）には、非同期APIの活用によって **フレーム落ちや入力遅延のない快適な動作**を実現できます。

✅ Async 名前空間を使った非同期読み書き

非同期APIは `SD.Load.Async` や `SD.Save.Async` といったサブ名前空間に自動生成されます。 これらは `await` に対応しており、以下のように使用します：

```
if (await SD.Load.Async.Shared())
{
    // 正常に読み込めた
}
```

データの取得や保存が完了するまで待機しつつ、Unity のメインスレッドをブロックしないため、UI更新やアニメーションにも影響しません。

⚠️ 注意：非同期処理中は Unity API を使えません

非同期読み書き中に生成されるデータクラスは、**別スレッド上でインスタンス化されます**。 そのため、`MonoBehaviour.Find()` や `GameObject.Instantiate()` など **Unity の API をコンストラクタやフィールド初期化子、読み書き時のコールバックでは使用できません**。

どうしても Unity API を使いたい場合は、非同期処理を使用しないか、非同期読み書き処理とは分けて記述してください。

---

4. よくあるユースケース

4.1 データ構造の変更にどう対応するか

ゲームの開発が進むにつれて、セーブ対象となるデータクラスに **フィールドを追加したり削除したり** することは珍しくありません。
リリース前であればどのような変更でも問題は起きませんが、すでにリリースしているゲームの場合は注意が必要です。

`JsonUtility` や `MessagePack` を使う場合、**読み込まれるセーブデータに存在しないフィールドはデフォルト値で補完**されます。 一方で、過去のバージョンのセーブデータに不要なフィールドが残っていたとしても、それを無視して読み込むことができます。

より複雑な変更（フィールド名の変更、型の変更など）が発生する場合は、以下のような対応が推奨されます：

- データに `Version` フィールドを追加し、読み込み後にバージョンによって処理を分岐する
- `IAfterLoadCallback` を使って、古い構造を新しい構造に変換する
- 古いデータを一度破棄し、`Initialize` で新しい形式のデータを再生成する

こうしたカスタム処理は Save Design が提供する自由な設計の中で実装可能です。 プロジェクトの成長にあわせて、**セーブ構造も安全に進化させられる仕組み**を整えておくことが重要です。

以下は、一部のフィールドの型を変更する例です。

```
[SlotData, Serializable]
public class ExampleData : IAfterLoadCallback
{
    public int version;    // セーブデータのバージョン
    public int oldField;   // 古いフィールドを削除してしまうとデータが読み込めないため残したままにする
    public float newField; // 型を変更した新しいフィールド

    void IAfterLoadCallback.OnAfterLoad()
    {
        // 古いデータを読み込んだときのマイグレーション処理
        if (version == 0)
        {
            newField = (float)oldField;
            version = 1;
        }

        // さらにバージョンが変わった場合
        if (version == 1)
        {
            ...
        }
    }
}
```

4.2 読み書き時に発生したエラーの確認方法

Save Design は原則として **例外を投げず、失敗した場合は false を返す**設計になっています。 これは、開発中に想定外のデータ破損や I/O エラーが発生しても、ゲーム全体の動作が止まらないようにするためです。

初期化や読み書き時に発生したエラーを確認したい場合は、ルートクラスに生成される部分メソッド `OnGameDataError(System.Exception e)` を実装してください。

```
[SaveDesignRoot]
internal partial class SD
{
    static partial void OnGameDataError(Exception e)
    {
        Debug.LogException(e);
    }
}
```

4.3 一部のデータだけを初期化したい

`Initialize` は指定したデータの種類であればすべて初期化してしまいます。

少し変則的ではありますが、一部のデータだけを初期化したい場合は次のような方法があります。

```
[SharedData("Settings"), Serializable]
public class Audio : IAfterInitializeCallback
{
    public float volume;

    public OnAfterInitialize()
    {
        volume = 1f;
    }
}

// 直接初期化後のコールバックを呼び出す
SD.Shared.Settings.Audio.OnAfterInitialize();
```

---

5. セキュリティと暗号化（オプション）

5.1 暗号化を有効にする方法（エディタツール）

Save Design には、Unity エディタ上で簡単に暗号化設定を行える専用ツールが用意されています。 次の手順で暗号化を有効にすることができます：

1. Unity のメニューバーから **「Tools > Save Design > Encrypt Settings」** を開きます。
2. 表示されたウィンドウに、AES鍵（32文字）と HMAC鍵（32文字）を入力します。 ※ どちらもセキュアなランダム文字列を推奨します。
3. [**Generate Encryptor.cs**] ボタンを押すと、`Encryptor.cs` という暗号化スクリプトが自動生成されます。

このスクリプトには Save Design のランタイム処理にフックされる暗号化ロジックが含まれており、 その後に行われるすべてのセーブは自動的に暗号化され、ロード時には HMAC による**改ざん検知**が行われるようになります。

5.2 独自暗号化処理の組み込み方法

より細かい制御や特殊な暗号方式を使いたい場合、Save Design では**暗号化ロジックを自前で差し替えることも可能**です。 `Encryptor` 属性を付与したクラスに `Encrypt` 関数と `Decrypt` 関数を実装するだけで独自処理を組み込めます。

```
using SaveDesign.Runtime;

[Encryptor]
public static class CustomEncryptor
{
    public static void Encrypt(ref byte[] data)
    {
        ...
    }

    public static void Decrypt(ref byte[] data)
    {
        ...
    }
}
```

これらのメソッドは、それぞれ **保存前**／**読み込み後**に呼び出されるフックポイントです。 ここで `data` に対して任意の暗号化・復号処理を施すことで、独自のセキュリティポリシーに対応できます。

たとえば：

- 業務用タイトルで内部形式に基づいた特殊な暗号ロジックを導入する
- セーブごとにランダムなIVやSaltを用いた複雑な暗号スキームを構築する

- 複数ファイルをまたいだ HMAC チェックを導入する

といったことも可能です。

このように、Save Design の暗号化機能は **簡単に導入できる標準ツール**と、**高度な組み込みが可能な拡張性**の両方を兼ね備えています。 セーブデータの信頼性を高めたい場合は、ぜひ導入を検討してください。

⚠️ 注意：暗号化処理を組み込む場合はゲームのリリース前に済ませてください

平文のデータは暗号化処理を組み込んだ状態では読み込めません。同じく暗号化されたデータは暗号化処理が組み込まれていない場合は読み込めません。

ゲームをリリースした後に暗号化処理を組み込んだり、逆に暗号化処理を削除してしまうとデータが読み込めなくなります。

ライセンスとサードパーティ表記

Save Design は、下記のサードパーティ製ライブラリを利用可能な構成になっています。 これらはすべて MIT ライセンスの下で配布されています。

■ サードパーティライブラリ（オプション）

ライブラリ名	ライセンス	用途
UniTask	MIT	非同期処理のサポート（`Async` API）
MessagePack for C#	MIT	高速バイナリシリアライザー（任意使用）

これらのライブラリは Save Design に同梱されておらず、**オプション機能として利用者が導入・管理する必要があります。** そのため、**ビルドに含めるかどうかは開発者の判断に委ねられます。**

また、これらをプロジェクトに組み込む場合は、ライセンス条件に従い、必要に応じてエンドユーザー向けにライセンスの表記を行ってください。

■ Save Design のライセンスについて

本アセット「Save Design」自体のコード・構成物はすべて商用利用可能な **Unity Asset Store EULA** に基づいて提供されています。 詳細は Unity Asset Store エンドユーザーライセンス をご参照ください。