システム概要

PlayerStateManagerは、プレイヤーの行動状態を管理するステートマシンの中核を担うクラスです。DefaultState、WallRunState、LandingState、RewindStateなど、複数の状態を動的に切り替え、各状態に応じた挙動を実現します。

主な特徴:

- 状態パターンの完全実装
- ActorComponentベースの動的ステート生成
- インターフェイス駆動設計による疎結合
- データ駆動型のステート管理
- 型安全なステート切り替え

システム全体構造

【ステートマシンのアーキテクチャ】

PlayerCharacter
└→ PlayerStateManager (状態管理の司令塔)
├→ StateClassMap (状態クラスの辞書)
│ ├─ Default → UDefaultState::StaticClass()
│ ├─ WallRun → UWallRunState::StaticClass()
│ ├─ Landing → ULandingState::StaticClass()
│ └─ Rewinding → URewindState::StaticClass()
│
└→ CurrentState (現在の状態)
└→ TScriptInterface<IPlayerCharacterState>
├→ Object: UDefaultState のインスタンス
└→ Interface: IPlayerCharacterState*

【ステート遷移の流れ】

```
PlayerCharacter::Jump()
    ↓
PlayerStateManager::ChangeState(WallRun)
    ↓
1. CurrentState->OnExit()      // DefaultStateを終了
2. NewObject<UWallRunState>()  // WallRunStateを生成
3. NewState->OnEnter()         // WallRunStateを開始
4. CurrentState = NewState     // 状態を切り替え
```

この設計の狙い:

- **PlayerStateManager**: 状態の切り替えのみを担当（シンプル）
- 各**State**: 自身の挙動のみを実装（責務の明確化）
- **Interface**: 状態間の共通規約を定義（疎結合）

核心的な設計思想

1. ステートパターンの完全実装

本システムは、GoFデザインパターンのステートパターンを忠実に実装しています。これにより、プレイヤーの複雑な挙動を明確に分離管理できます。

【ステートパターンの基本構造】

```
IPlayerCharacterState (インターフェイス)
    ├→ OnEnter(AActor* Owner)    // 状態開始時
    ├→ OnUpdate(float DeltaTime) // 毎フレーム更新
    ├→ OnExit()                  // 状態終了時
    ├→ Movement(...)             // 移動入力
    ├→ Jump(...)                 // ジャンプ入力
    ├→ RePlayAction(...)         // 時間操作入力
```

```
    └→ BoostAction(...)          // ブースト入力

具象State (各状態の実装)
    ├→ UDefaultState      // 通常状態
    ├→ UWallRunState      // 壁走り状態
    ├→ ULandingState      // 着地硬直状態
    └→ URewindState       // 巻き戻し状態
```

インターフェイス定義:

cpp

```
UINTERFACE(MinimalAPI, Blueprintable)
class UPlayerCharacterState : public UInterface
{
    GENERATED_BODY()
};

class IPlayerCharacterState
{
    GENERATED_BODY()

public:
    //ライフサイクル
    virtual bool OnEnter(AActor* Owner) = 0;
    virtual bool OnUpdate(float DeltaTime) = 0;
    virtual void OnExit() = 0;

    // 入力処理
```

```
    virtual bool Movement(const FInputActionValue& Value) = 0;
    virtual bool Jump(const FInputActionValue& Value) = 0;
    virtual bool RePlayAction(const FInputActionValue& Value) = 0;
    virtual bool BoostAction(const FInputActionValue& Value) = 0;
};
```

PlayerCharacterでの使用:

cpp

```
void APlayerCharacter::Jump()
{
    if (StateManager && StateManager->GetCurrentState())
    {
        // 現在の状態に関係なく、同じ方法で呼び出し
        StateManager->GetCurrentState()->Jump(InputValue);
    }
}
```

この設計の利点:

- PlayerCharacterは現在の状態を意識しない
- 新しい状態の追加が容易（既存コードに影響なし）
- 各状態が完全に独立してテスト可能

---

2. データ駆動型のステート管理

ステートクラスをコードに直接書くのではなく、**TMap**で管理することで、エディタから自由に設定できるようにしました。

cpp

```
// ステート種別の列挙型
UENUM(BlueprintType)
enum class EPlayerStateType : uint8
```

```
{
    Default,    //通常状態
    WallRun,    //壁走り状態
    Landing,    //着地硬直状態
    Rewinding   //巻き戻し状態
};

//ステートクラスのマップ
UPROPERTY(EditAnywhere, Category = "State Machine")
TMap<EPlayerStateType, TSubclassOf<UObject>> StateClassMap;
```

Unreal Editorでの設定:

```
PlayerCharacter の詳細パネル:
State Machine
    ├─ Default   → DefaultState (UDefaultState)
    ├─ WallRun   → WallRunState (UWallRunState)
    ├─ Landing   → LandingState (ULandingState)
    └─ Rewinding → RewindState (URewindState)
```

この設計の流れ:

```
【ゲーム起動時】
1. Unreal Editor でステートクラスを登録
```

```
StateClassMap[Default] = UDefaultState::StaticClass()
StateClassMap[WallRun] = UWallRunState::StaticClass()
...

【ゲームプレイ中】
2. ChangeState(WallRun) が呼ばれる
  ↓
3. StateClassMap[WallRun] からクラスを取得
  ↓
4. NewObject<UWallRunState>() で動的生成
  ↓
5. OnEnter() で初期化して実行
```

この設計の利点:

- 新しいステートを追加してもコード変更不要
- Blueprintでステートを拡張可能
- プランナーがエディタで状態を追加可能

---

3. TScriptInterfaceによる安全なインターフェイス管理

Unreal Engineでインターフェイスを扱う場合、**TScriptInterface**を使うことで、BlueprintとC++の両方で安全にアクセスできます。

```
cpp
// 現在のステートを保持
UPROPERTY()
TScriptInterface<IPlayerCharacterState> CurrentState;
```

TScriptInterfaceの内部構造:

```

TScriptInterface<IPlayerCharacterState>

    ├→ ObjectPointer (UObject*)

    │    └→ 実際のインスタンス（例: UDefaultState）

    │

    └→ InterfacePointer (IPlayerCharacterState*)

        └→ インターフェイスへのポインタ
```

使用方法:

cpp

```
// ステート切り替え時

TScriptInterface<IPlayerCharacterState>
UPlayerStateManager::ChangeState(EPlayerStateType NextStateTag)

{

    // 1. UObjectとして生成

    UObject* NewStateObject = NewObject<UObject>(this, StateClass);


    // 2. インターフェイスとしてキャスト

    IPlayerCharacterState* NewStateInterface =
Cast<IPlayerCharacterState>(NewStateObject);


    // 3. TScriptInterfaceに両方を設定

    CurrentState.SetObject(NewStateObject);

    CurrentState.SetInterface(NewStateInterface);


    return CurrentState;

}
```

```
// ステート使用時
void Update(float DeltaTime)
{
    if (CurrentState)  // null チェック
    {
        CurrentState->OnUpdate(DeltaTime);  // インターフェイス経由で呼び出し
    }
}
```

TScriptInterfaceの利点:

- Blueprintとの互換性
- ガベージコレクションによる自動メモリ管理
- 型安全なインターフェイスアクセス
- null チェックが容易

---

4. 動的ステート生成によるメモリ効率化

各ステートを使用時に生成し、切り替え時に破棄することで、メモリを効率的に使用します。

cpp

```
TScriptInterface<IPlayerCharacterState>
UPlayerStateManager::ChangeState(EPlayerStateType NextStateTag)
{
    // 現在のステートを終了
    if (CurrentState)
    {
        CurrentState->OnExit();
        // → ここで旧ステートはガベージコレクション対象になる
    }

    // 新しいステートを動的生成
```

```
    UObject* NewStateObject = NewObject<UObject>(this, StateClass);

    // 初期化
    IPlayerCharacterState* NewStateInterface =
Cast<IPlayerCharacterState>(NewStateObject);
    NewStateInterface->OnEnter(GetOwner());

    // 現在のステートとして設定
    CurrentState.SetObject(NewStateObject);
    CurrentState.SetInterface(NewStateInterface);

    return CurrentState;
}
```

メモリ使用量の比較:

```
【全ステート常駐の場合】
DefaultState:  5KB
WallRunState:  4KB
LandingState:  3KB
RewindState:   6KB
合計: 18KB（常に確保）

【動的生成の場合】
アクティブなステートのみ: 3〜6KB
非アクティブ時: 0KB
```

平均: 約5KB（約70%削減）

```

**動的生成の流れ:**

```
【通常状態 → 壁走り状態】

1. DefaultState がメモリ上に存在
   ├→ OnUpdate() が毎フレーム呼ばれる
   └→ メモリ使用: 5KB

2. ChangeState(WallRun) 呼び出し
   ├→ DefaultState->OnExit()
   ├→ DefaultState は参照を失う
   └→ ガベージコレクション対象に

3. NewObject<UWallRunState>()
   ├→ WallRunState がメモリ上に生成
   ├→ OnEnter() で初期化
   └→ メモリ使用: 4KB

4. ガベージコレクション実行（適宜）
   └→ DefaultState がメモリから削除
```

この設計の利点:

- 使用していないステートのメモリ解放
- 状態数が増えてもメモリ増加が最小限
- 初期化コストが分散（起動時の負荷軽減）
```

---

5. 型安全なステート切り替え

列挙型を使うことで、コンパイル時に不正なステート指定を検出できます。

cpp

```
// ✗ 文字列ベース（タイプミスに弱い）
void BadChangeState(FString StateName)
{
    if (StateName == "WalRun")  // ← タイプミス、実行時まで気づかない
    {
        // ...
    }
}

// ✓ 列挙型ベース（タイプセーフ）
void UPlayerStateManager::ChangeState(EPlayerStateType NextStateTag)
{
    // NextStateTag は EPlayerStateType のみ受け付ける
    //コンパイル時に型チェック
}

// 使用例
ChangeState(EPlayerStateType::WallRun);  // OK
ChangeState(EPlayerStateType::WalRun);   //コンパイルエラー！
```

データフロー

初期化の流れ

```
【ゲーム開始時の初期化】

1. PlayerCharacter::BeginPlay()
   ↓
2. PlayerStateManager = NewObject<UPlayerStateManager>()
   ├→ コンポーネントとして登録
   └→ StateClassMap はエディタで設定済み
   ↓
3. PlayerStateManager->Init()
   ↓
4. ChangeState(EPlayerStateType::Default)
   ├→ StateClassMap[Default] を取得
   │    └→ UDefaultState::StaticClass()
   │
   ├→ NewObject<UDefaultState>()
   │    └→ DefaultState のインスタンス生成
   │
   ├→ DefaultState->OnEnter(this)
   │    ├→ OwnerActor を保存
   │    ├→ IPlayerInfoProvider 取得
   │    ├→ 移動速度を初期化
```

```
   │   └→ ジャンプカウントをリセット
   │
   └→ CurrentState = DefaultState
   ↓
5. ゲームプレイ開始
   └→ DefaultState->OnUpdate() が毎フレーム呼ばれる
```

ステート切り替えの詳細フロー

```
【壁に接触してWallRunStateに遷移】

1. DefaultState::Movement() 実行中
   ├→ WallRunComponent->CheckWallContact()
   └→ 壁を検出
   ↓
2. OwnerInterface->ChangeState(EPlayerStateType::WallRun)
   ↓
3. PlayerStateManager::ChangeState(WallRun)

   【現在のステート終了】
   ├→ CurrentState->OnExit()
   │   └→ DefaultState::OnExit()
   │       └→（特に処理なし）
```

```
    │
  【新しいステート生成】
    ├→ StateClassMap[WallRun] 取得
    │    └→ UWallRunState::StaticClass()
    │
    ├→ NewObject<UWallRunState>(this, StateClass)
    │    └→ WallRunState インスタンス生成
    │
    ├→ Cast<IPlayerCharacterState>(NewStateObject)
    │    └→ インターフェイスポインタ取得
    │
  【新しいステート開始】
    ├→ WallRunState->OnEnter(PlayerCharacter)
    │    ├→ OwnerActor を保存
    │    ├→ WallRunComponent 取得
    │    ├→ 壁走り開始処理
    │    └→ カメラロール適用
    │
  【ステート切り替え】
    ├→ CurrentState.SetObject(WallRunState)
    ├→ CurrentState.SetInterface(WallRunStateInterface)
    └→ return CurrentState
    ↓
4. 次フレームから
    └→ WallRunState->OnUpdate() が呼ばれる
```

毎フレームの更新フロー

```
【PlayerCharacter::Tick()】

1. PlayerCharacter::Tick(DeltaTime)
   ↓
2. PlayerStateManager->Update(DeltaTime)
   ↓
3. if (CurrentState)
       CurrentState->OnUpdate(DeltaTime)
   ↓
4. 現在のステートに応じて処理

   【DefaultStateの場合】
   ├→ 接地チェック
   ├→ ジャンプカウントリセット
   ├→ 落下距離更新
   └→ return true

   【WallRunStateの場合】
   ├→ 壁接触チェック
   ├→ 壁走り継続判定
   ├→ カメラロール更新
```

└→ 壁から離れたら ChangeState(Default)

【LandingStateの場合】

├→ 硬直タイマー更新

├→ 硬直終了判定

└→ 終了なら ChangeState(Default)

【RewindStateの場合】

├→ 巻き戻し進行チェック

├→ TimeManipulator の状態監視

└→ 終了なら ChangeState(Default)

主要な実装上の工夫

工夫1: ステート切り替えの戻り値設計

ChangeState()は切り替え後のステートを返すことで、即座に操作できるようにしています。

cpp

```
TScriptInterface<IPlayerCharacterState>
UPlayerStateManager::ChangeState(EPlayerStateType NextStateTag)

{

    // ... ステート切り替え処理 ...


    return CurrentState;  // 切り替え後のステートを返す

}


// 使用例

void APlayerCharacter::EnterWallRun()
```

```
{
    // ステート切り替えと同時に取得
    if (TScriptInterface<IPlayerCharacterState> NewState =
StateManager->ChangeState(EPlayerStateType::WallRun))
    {
        // 即座に追加の初期化処理が可能
        // （通常は不要だが、特殊な場合に便利）
    }
}
```

工夫2: IsStateMatch()による状態確認

現在のステートが特定の状態かどうかをクラス比較で判定します。

cpp

```
bool UPlayerStateManager::IsStateMatch(EPlayerStateType StateTag)
{
    // マップに存在しない or ステートが null
    if (!StateClassMap.Contains(StateTag) || !CurrentState.GetObject())
    {
        return false;
    }

    // 目的のステートクラスを取得
    const TSubclassOf<UObject> TargetStateClass = StateClassMap[StateTag];

    // 現在のステートのクラスと比較
    return CurrentState.GetObject()->GetClass() == TargetStateClass;
```

```
}
```

使用例:

cpp

```
void APlayerCharacter::TryBoost()
{
    // 通常状態の時だけブースト可能
    if (StateManager->IsStateMatch(EPlayerStateType::Default))
    {
        PlayBoost();
    }
    else
    {
        UE_LOG(LogTemp, Warning, TEXT("Cannot boost in current state"));
    }
}
```

なぜクラス比較なのか:

- 列挙型フラグを持つより正確
- ステートの実体と完全に一致
- Blueprintで拡張されても動作

工夫3: 徹底したnullチェックとログ出力

ステート切り替えは失敗する可能性があるため、各段階でチェックします。

cpp

```
TScriptInterface<IPlayerCharacterState>
UPlayerStateManager::ChangeState(EPlayerStateType NextStateTag)
{
    // 第1段階: ステートタグの有効性
```

```
    if (!StateClassMap.Contains(NextStateTag))
    {
        UE_LOG(LogTemp, Warning, TEXT("UPlayerStateManager::ChangeState - Invalid state tag"));
        return nullptr;
    }

    // 第2段階: ステートクラスの取得
    const TSubclassOf<UObject> StateClass = StateClassMap[NextStateTag];
    if (!StateClass)
    {
        UE_LOG(LogTemp, Error, TEXT("UPlayerStateManager::ChangeState - StateClass is null"));
        return nullptr;
    }

    // 第3段階: インスタンス生成
    UObject* NewStateObject = NewObject<UObject>(this, StateClass);
    if (!NewStateObject)
    {
        UE_LOG(LogTemp, Error, TEXT("UPlayerStateManager::ChangeState - Failed to create state instance"));
        return nullptr;
    }

    // 第4段階: インターフェイス実装確認
    IPlayerCharacterState* NewStateInterface = Cast<IPlayerCharacterState>(NewStateObject);
```

```
    if (!NewStateInterface)
    {
        UE_LOG(LogTemp, Error, TEXT("UPlayerStateManager::ChangeState - State does not implement IPlayerCharacterState"));
        return nullptr;
    }

    // すべてのチェックをパスしたので実行
    // ...
}
```

この多段防御の効果:

- どの段階で失敗したか明確
- デバッグ時の問題特定が容易
- クラッシュを完全に防止

工夫4: GetOwner()を使ったOwner参照

PlayerStateManagerは**ActorComponent**のサブクラスなので、GetOwner()でPlayerCharacterを取得できます。

cpp

```
// PlayerCharacterでの初期化
void APlayerCharacter::BeginPlay()
{
    StateManager = NewObject<UPlayerStateManager>(this);  // this が Owner
    StateManager->Init();
}

// StateManagerでの使用
void UPlayerStateManager::Init()
```

```
{
    ChangeState(EPlayerStateType::Default);
}

TScriptInterface<IPlayerCharacterState> UPlayerStateManager::ChangeState(...)
{
    // ...

    // GetOwner() で PlayerCharacter を取得
    NewStateInterface->OnEnter(GetOwner());  // AActor* を渡す

    // ...
}
```

この設計の利点:

- PlayerCharacterへの参照を明示的に保持不要
- UObjectの仕組みに従った自然な設計
- ガベージコレクションが正しく動作

工夫5: OnExit()の呼び出しタイミング

現在のステートを終了してから新しいステートを開始することで、状態の一貫性を保ちます。

cpp

```
TScriptInterface<IPlayerCharacterState> UPlayerStateManager::ChangeState(...)
{
    // ★重要: 新しいステートを生成する前に、現在のステートを終了
    if (CurrentState)
    {
        CurrentState->OnExit();
```

```
    }

    // この時点で CurrentState は無効な状態
    // （古いステートは終了済み、新しいステートはまだ開始前）

    // 新しいステート生成
    UObject* NewStateObject = NewObject<UObject>(this, StateClass);
    IPlayerCharacterState* NewStateInterface =
Cast<IPlayerCharacterState>(NewStateObject);

    // 新しいステート開始
    NewStateInterface->OnEnter(GetOwner());

    // この時点で CurrentState は新しいステートを指す
    CurrentState.SetObject(NewStateObject);
    CurrentState.SetInterface(NewStateInterface);

    return CurrentState;
}
```

なぜこの順序なのか:

- OnExit()で古いステートのリソースを解放
- OnEnter()で新しいステートのリソースを確保
- 同時に2つのステートがアクティブにならない

得られた成果

✅ 明確な責務分離

- PlayerStateManager: ステート切り替えのみ

- 各State: 自身の挙動のみ
- PlayerCharacter: 入力の受付と委譲のみ

✅ 高い拡張性

- 新しいステートの追加が容易
- Blueprintでステートを拡張可能
- 既存コードへの影響なし

✅ 型安全性

- 列挙型による不正値の排除
- インターフェイスによる契約の保証
- コンパイル時のエラー検出

✅ メモリ効率

- 動的生成による約70%のメモリ削減
- ガベージコレクションによる自動管理
- リソースリークの防止

まとめ

PlayerStateManagerは、ステートパターンとデータ駆動設計を組み合わせた、シンプルで強力な状態管理システムです。

技術的なハイライト:

- ステートパターン: GoFデザインパターンの忠実な実装
- 動的生成: 使用時に生成、未使用時は解放
- **TScriptInterface**: BlueprintとC++の両対応
- データ駆動: エディタから自由に設定可能

この実装により、複雑なプレイヤーの挙動を明確に分離管理し、保守性と拡張性の高いゲームシステムを実現しました。各ステートが独立しているため、新機能の追加やバグ修正が容易で、チーム開発においても大きなメリットをもたらします。