システム概要

状態パターンにおける「通常状態」を実装したクラスです。プレイヤーの基本的な移動・ジャンプ・時間操作・ブースト等の入力を処理し、他の特殊状態への遷移も管理します。二段ジャンプ、カメラヘッドボブ、落下距離計測など、プレイヤーの基本動作に関わる全ての機能を担当します。

核心的な設計思想

1. 状態パターンの具現化 - 挙動のカプセル化

DefaultStateは、IPlayerCharacterStateインターフェイスを実装することで、PlayerCharacterから完全に独立した挙動を提供します。

【状態パターンの構造】

IPlayerCharacterState (インターフェイス)

├─ OnEnter() - 状態開始処理

├─ OnUpdate() - 毎フレーム更新

├─ OnExit() - 状態終了処理

├─ 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

UDefaultState (実装)

└→ 各メソッドで通常状態の挙動を定義

この設計により実現したこと:

- PlayerCharacterは現在の状態を意識せず、常に同じ方法で入力を処理
- 状態の追加・変更がPlayerCharacterに影響しない
- 各状態が独立してテスト可能

---

2. 責務の明確な分離 - **DefaultState**の役割

DefaultStateは「通常時の挙動」という単一の責務のみを持ちます。

【DefaultStateが担当する処理】

✓ 基本移動（WASD入力）

✓ 二段ジャンプの管理

✓ 時間記録の開始・停止

✓ ブースト発動

✓ パルクール発動の判定

✓ 壁走りへの遷移判定

✓ 落下距離の計測

✓ カメラヘッドボブの更新

【DefaultStateが担当しない処理】

✗ 実際のカメラ制御 → PlayerCameraControlComponent

✗ 実際の時間操作 → TimeManipulatorComponent

✗ 実際のブースト → BoostComponent

✗ 実際のパルクール → ParkourComponent

✗ 壁走りの検出・実行 → WallRunComponent

DefaultStateは「どの機能を使うか判定」するだけで、実際の処理は各コンポーネントに委譲します。

---

3. 定数管理による保守性向上

マジックナンバーを排除し、名前空間で定数を一元管理しました。

cpp

```
namespace DefaultStateConstants
{
    /** 入力を無視する閾値（スティックの遊び範囲） */
```

```
    static constexpr float INPUT_DEADZONE = 0.2f;

    /** Head Bob の補間速度（停止時の戻り速度） */
    static constexpr float BOB_INTERP_SPEED = 5.0f;

    /** 二段ジャンプの最大回数 */
    static constexpr int MAX_JUMPCOUNT = 2;

    /** 着地硬直が発生し始める最小落下距離 */
    static constexpr float MIN_FALL_DISTANCE_FOR_LAG = 800.0f;

    /** 最大着地硬直時間を発生させる落下距離 */
    static constexpr float MAX_FALL_DISTANCE_FOR_LAG = 2000.0f;

    /** 最小着地硬直時間（秒） */
    static constexpr float MIN_LANDING_LAG = 0.0f;

    /** 最大着地硬直時間（秒） */
    static constexpr float MAX_LANDING_LAG = 1.0f;
}
```

この設計の利点:

- 定数の意味が名前で明確
- 変更時の影響範囲が明確
- コンパイル時に最適化（constexpr）

- チーム内での調整が容易

システム全体構造

```
【DefaultStateの位置付け】

PlayerCharacter
    └→ PlayerStateManager
        ├→ DefaultState (通常状態) ← 本クラス
        ├→ WallRunState (壁走り状態)
        ├→ LandingState (着地硬直状態)
        └→ RewindState (巻き戻し状態)

【DefaultStateの依存関係】

DefaultState
    ├→ IPlayerInfoProvider (インターフェイス経由でPlayerCharacterにアクセス)
    ├→ ITimeControllable (インターフェイス経由で時間操作)
    │
    ├→ WallRunComponent (壁走り判定を委譲)
    ├→ PlayerCameraControlComponent (ヘッドボブ更新を委譲)
    ├→ CharacterMovementComponent (移動処理を委譲)
    │
    └→ PostProcessEffectHandle (視覚エフェクト管理)
```

```

---

## データフロー

### 状態のライフサイクル

```
【状態遷移の全体フロー】

1. OnEnter() - 状態開始
   ├→ OwnerActorを保存
   ├→ IPlayerInfoProviderインターフェイスを取得
   ├→ 移動速度を初期化
   ├→ 記録フラグをリセット
   └→ 現在の地面高度を記録

2. OnUpdate() - 毎フレーム更新
   ├→ 接地状態をチェック
   ├→ 接地中ならジャンプカウントをリセット
   ├→ 接地中なら現在高度を記録
   └→ 落下中は高度を記録しない（落下開始時の高度を保持）

3. OnExit() - 状態終了
   └→（現在は特別な処理なし）
```

**OnUpdate()の詳細:**
```

cpp

```cpp
bool UDefaultState::OnUpdate(float DeltaTime)
{
    ACharacter* Character = Cast<ACharacter>(OwnerActor);
    if (Character)
    {
        UCharacterMovementComponent* MoveComp = Character->GetCharacterMovement();

        // 地面に接地している場合
        if (MoveComp->IsMovingOnGround())
        {
            // ジャンプ回数をリセット（二段ジャンプ可能に）
            JumpCount = DefaultStateConstants::MAX_JUMPCOUNT;

            // ★重要: 接地中は常に現在高度を記録
            // これにより、次に落下したときの落下距離を正確に計算できる
            LastGroundHeight = OwnerActor->GetActorLocation().Z;
        }
        // 落下中は高度を更新しない（落下開始時の高度を保持）
    }
    return true;
}
```

この実装の工夫:

- 接地中は常に高度を更新することで、階段や坂道でも正確な落下距離を計測

- 落下中は更新しないことで、落下開始時の高度を保持
- ジャンプカウントのリセットも同じタイミングで実行

移動入力の処理フロー

```
【Movement() の処理フロー】

1. 入力受信
   OnMove(Value) がPlayerCharacterから呼ばれる
   ↓
2. 壁走り中かチェック
   if (WallRunComponent->IsWallRunning())
       return true;  // 壁走り中は通常移動しない
   ↓
3. 入力値を取得
   FVector2D MoveInput = Value.Get<FVector2D>();

4. 移動方向を計算
   FVector MoveDir = CalculateMoveDirection(Character, MoveInput);
   ↓ カメラの向きと入力を組み合わせて
   ↓ ワールド空間の移動方向に変換
   ↓
5. CharacterMovementComponentに移動を指示
   Character->AddMovementInput(MoveDir, MoveSpeed);
```

↓

6. カメラヘッドボブを更新

CameraControl->UpdateHeadBob(MoveInput, IsMoving, IsFalling);

CalculateMoveDirection() の詳細:

cpp

```
FVector UDefaultState::CalculateMoveDirection(
    ACharacter* Character,
    const FVector2D& MoveInput) const
{
    // 1. カメラの回転を取得
    FRotator CamRot = Character->GetControlRotation();

    // 2. カメラの前方・右方向ベクトルを取得
    FVector CamForward = FRotationMatrix(CamRot).GetUnitAxis(EAxis::X);
    FVector CamRight = FRotationMatrix(CamRot).GetUnitAxis(EAxis::Y);

    // 3. ★重要: 逆さ状態（天井に張り付いている等）なら左右反転
    if (FVector::DotProduct(OwnerActor->GetActorUpVector(), FVector::UpVector) < 0.f)
        CamRight *= -1.f;

    // 4. ワールド空間の移動方向を算出
    FVector WorldMoveDir = (CamRight * MoveInput.X + CamForward *
MoveInput.Y).GetSafeNormal();

    // 5. ローカル空間に変換
    FVector LocalMoveDir =
OwnerActor->GetActorTransform().InverseTransformVectorNoScale(WorldMoveDir);
```

```
    // 6. キャラクターの向きに合わせた最終的な移動方向
    return OwnerActor->GetActorRightVector() * LocalMoveDir.Y +
           OwnerActor->GetActorForwardVector() * LocalMoveDir.X;
}
```

この実装の特徴:

- カメラの向きに基づいた直感的な移動
- 逆さ状態でも正しい方向に移動（重力反転ゲーム等に対応）
- ローカル座標系への変換により、キャラクターの向きに依存しない移動

ジャンプ処理の優先順位制御

```
【Jump() の処理フロー - 優先順位順】

1. 最優先: パルクール判定
   if (OwnerInterface->PlayParkour())
       return true;
   → 壁の前なら、ジャンプではなくパルクールを実行

2. 第2優先: 壁走りジャンプ判定
   if (WallRunComponent->HandleJumpPressed())
   {
```

```
        JumpCount = MAX_JUMPCOUNT;  // 壁ジャンプ後は二段ジャンプ可能に
        return true;
    }
    → 壁走り中なら壁ジャンプを実行

3. 第3優先: 通常ジャンプ
    if (JumpCount <= 0)
        return false;  // ジャンプ回数使い切り

    moveComp->AddImpulse(FVector(0, 0, 1200.0f), true);
    --JumpCount;
    return true;
```

この優先順位の理由:

1. パルクールが最優先 - 壁の前でジャンプ入力 = パルクールしたい意図
2. 壁ジャンプが次 - 壁走り中のジャンプ = 壁から飛び離れたい意図
3. 通常ジャンプが最後 - 上記以外の場合のみ

壁ジャンプ後のジャンプカウントリセットの意図:

```
cpp
if (WallRunComponent->HandleJumpPressed())
{
    JumpCount = MAX_JUMPCOUNT;  // ★ここでリセット
    return true;
}
```

- 壁ジャンプ後、再度二段ジャンプが可能に
- 壁→壁→壁と連続で移動できるパルクールアクションを実現

時間操作（RePlayAction）の処理フロー

```

【RePlayAction() の処理フロー】

【状態A: 記録していない時】
プレイヤーがEキーを押す
  ↓
1. TimeControllable->StartTimeRecording()
   └→ TimeManipulatorComponentが位置・回転の記録開始

2. USoundHandle::PlaySE("StartRecording")
   └→ 記録開始の効果音

3. IsRecording = true
   └→ 内部フラグを立てる

【状態B: 記録中】
プレイヤーが再度Eキーを押す
  ↓
1. TimeControllable->StopTimeRecording()
   └→ 記録を停止

2. USoundHandle::PlaySE("Replay", true)
   └→ 巻き戻し開始の効果音（ループ）
```

3. OwnerInterface->ChangeState(EPlayerStateType::Rewinding)

   └→ RewindStateに遷移

      └→ RewindStateで実際の巻き戻し処理が実行される

この設計の特徴:

- DefaultStateは「記録の開始/停止」と「状態遷移の判断」のみ担当
- 実際の巻き戻し処理はRewindStateが担当（責務の分離）
- IsRecordingフラグで現在の状態を管理

ブースト処理

cpp

```
bool UDefaultState::BoostAction(const FInputActionValue& ActionValue)
{
    if(OwnerInterface == nullptr)
        return false;

    // PlayerCharacterのPlayBoost()を呼び出す
    OwnerInterface->PlayBoost();
    return true;
}
```

シンプルな実装の理由:

- DefaultStateは「ブースト可能かどうか」の判断のみ
- 実際のブースト処理はBoostComponentが担当
- インターフェイス経由で疎結合を維持

主要な実装上の工夫

工夫1: 落下距離に応じた動的硬直時間計算

実装の詳細:

cpp

```cpp
float UDefaultState::CalculateLandingLagDuration(float FallDistance) const
{
    // 800cm未満の落下 → 硬直なし
    if (FallDistance < MIN_FALL_DISTANCE_FOR_LAG)  // 800.0f
    {
        return 0.0f;
    }

    // 2000cm以上の落下 → 最大硬直（1.0秒）
    if (FallDistance >= MAX_FALL_DISTANCE_FOR_LAG)  // 2000.0f
    {
        return MAX_LANDING_LAG;  // 1.0f
    }

    // 800cm～2000cmの間は線形補間
    // 例: 1400cm → 0.5秒硬直
    float NormalizedDistance = (FallDistance - MIN_FALL_DISTANCE_FOR_LAG)
        / (MAX_FALL_DISTANCE_FOR_LAG - MIN_FALL_DISTANCE_FOR_LAG);

    return FMath::Lerp(MIN_LANDING_LAG, MAX_LANDING_LAG, NormalizedDistance);
}
```

具体的な硬直時間の例:

落下距離	硬直時間	状況

500cm	0.0秒	低い段差 - 硬直なし
800cm	0.0秒	硬直が発生し始める境界
1000cm	0.17秒	少し硬直
1400cm	0.5秒	中程度の硬直
1800cm	0.83秒	かなり硬直
2000cm以上	1.0秒	最大硬直

この設計の利点:

- リアルな着地感を演出
- 定数変更のみで調整可能
- 線形補間により滑らかな変化

工夫2: カメラヘッドボブの状態連動

cpp

```
bool UDefaultState::Movement(const FInputActionValue& Value)
{
    // ... 移動処理 ...

    if (UPlayerCameraControlComponent* CameraControl = ...)
    {
        bool IsMoving = MoveInput.Size() >= INPUT_DEADZONE;  // 0.2f
        bool IsFalling = MoveComp->IsFalling();
```

```
        // カメラコンポーネントにヘッドボブ更新を委譲
        CameraControl->UpdateHeadBob(MoveInput, IsMoving, IsFalling);
    }
}
```

IsMovingの判定にデッドゾーンを使う理由:

- コントローラーのスティックは完全にゼロにならない
- 0.2未満の微小な入力は「停止」として扱う
- ヘッドボブが不要に揺れるのを防止

IsFallingの情報も渡す理由:

- 空中と地上でヘッドボブの挙動を変える
- 空中ではヘッドボブを停止または弱める
- よりリアルなカメラ表現

工夫3: インターフェイス経由の疎結合設計

OnEnter()での初期化:

cpp

```
bool UDefaultState::OnEnter(AActor* owner)
{
    OwnerActor = owner;

    // ★インターフェイス経由でアクセス
    if (OwnerActor->GetClass()->ImplementsInterface(UPlayerInfoProvider::StaticClass()))
    {
        OwnerInterface.SetInterface(Cast<IPlayerInfoProvider>(OwnerActor));
        OwnerInterface.SetObject(OwnerActor);
    }
    else
    {
```

```
        return false;  // インターフェイス未実装なら初期化失敗
    }

    // 初期化処理...
    return true;
}
```

TScriptInterfaceを使う理由:

cpp

```
UPROPERTY()
TScriptInterface<IPlayerInfoProvider> OwnerInterface;
```

- Blueprintとの互換性を保持
- インターフェイスとオブジェクトの両方を安全に保持
- null チェックが容易

使用例:

cpp

```
// インターフェイス経由でメソッド呼び出し
OwnerInterface->PlayBoost();
OwnerInterface->ChangeState(EPlayerStateType::Rewinding);
```

この設計の効果:

- DefaultStateはPlayerCharacterの詳細を知らない
- 他のキャラクタータイプでも同じStateを使用可能
- テスト時にモックインターフェイスを注入可能

工夫4: スキルキャンセル時のクリーンアップ

cpp

```
void UDefaultState::SkillActionStop()
{
```

```
    IsRecording = false;
    UPostProcessEffectHandle::DeactivateEffect(
        this,
        EPostProcessEffectTag::Recording,
        true
    );
}
```

この関数が呼ばれるシーン:

- 時間記録中に巻き戻しが中断された
- 時間記録中に他の状態に強制遷移した
- プレイヤーが死亡した

クリーンアップの内容:

- 記録フラグをリセット
- 記録中の視覚エフェクトを削除

なぜ**OnExit()**ではなく別メソッドか:

- OnExit()は正常な状態遷移時のみ
- SkillActionStop()は異常終了時も含む
- より明示的なエラーハンドリング

パフォーマンス最適化

1. constexpr による定数の最適化

cpp

```
namespace DefaultStateConstants
{
    static constexpr float INPUT_DEADZONE = 0.2f;
    static constexpr int MAX_JUMPCOUNT = 2;
    // ... 他の定数
}
```

効果:

- コンパイル時に値が確定
- 実行時のメモリアクセスなし
- 最適化されたマシンコードが生成される

---

2. 早期リターンによる不要な処理のスキップ

cpp

```cpp
bool UDefaultState::Movement(const FInputActionValue& Value)
{
    // null チェックで早期リターン
    if (!OwnerInterface || !GetWorld())
        return false;

    // 壁走り中なら移動処理をスキップ
    if (UWallRunComponent* runCom = ...)
    {
        if (runCom->IsWallRunning())
            return true;
    }

    // 以降、通常の移動処理...
}
```

この実装の利点:

- 不要な計算を回避
- ネストが深くならない
- コードの可読性向上

---

3. コンポーネントの動的取得を最小化

cpp

```
// ✗ 毎フレーム取得（非効率）
void BadExample()
{
    UWallRunComponent* runCom =
OwnerActor->GetComponentByClass<UWallRunComponent>();
    if (runCom->IsWallRunning())
        // ...
}


// ✓ 必要な時のみ取得（効率的）
bool UDefaultState::Movement(const FInputActionValue& Value)
{
    // 壁走り判定が必要な時のみコンポーネント取得
    if (UWallRunComponent* runCom =
OwnerActor->GetComponentByClass<UWallRunComponent>())
    {
        if (runCom->IsWallRunning())
            return true;
    }
    // ...
}
```

最適化の考え方:

- OnEnter()でキャッシュすることも可能
- しかし壁走りは頻繁に使わないので、都度取得でも許容範囲
- 必要性と複雑性のバランスを考慮

エラーハンドリングとロバスト性

null チェックの徹底

cpp

```
bool UDefaultState::Jump(const FInputActionValue& Value)
{
    // Owner の存在確認
    if (OwnerActor == nullptr)
    {
        UE_LOG(LogTemp, Error, TEXT("DefaultState: Owner is Nullptr"));
        return false;
    }

    // MovementComponent の存在確認
    UCharacterMovementComponent* moveComp =
        OwnerActor->GetComponentByClass<UCharacterMovementComponent>();
    if (moveComp == nullptr)
    {
        UE_LOG(LogTemp, Error, TEXT("DefaultState: UCharacterMovementComponent is Nullptr"));
        return false;
    }

    // 処理実行...
}
```

詳細なログ出力の意図:

- どの段階で失敗したか明確化
- デバッグ時の問題特定が容易

- チーム開発でのトラブルシューティング支援

得られた成果

✅ 状態パターンの完全な実装

- インターフェイス経由の疎結合設計
- 責務の明確な分離
- 他の状態との一貫したインターフェイス

✅ リアルなゲームプレイ体験

- 落下距離に応じた動的硬直
- 二段ジャンプと壁ジャンプの組み合わせ
- 直感的な移動とカメラ制御

✅ 高い保守性

- 定数の一元管理
- 詳細なコメントとログ
- 明確な処理フロー

✅ 優れた拡張性

- 新しいアクションの追加が容易
- Blueprint拡張可能な設計
- 他のキャラクタータイプへの再利用

まとめ

UDefaultStateは、状態パターンにおける「通常状態」として、プレイヤーの基本的な挙動を全て管理しています。

設計の核心:

- インターフェイス駆動 - IPlayerCharacterStateとIPlayerInfoProviderによる疎結合
- 責務の明確化 - 判断のみを行い、実処理は各コンポーネントに委譲
- 定数管理 - namespace による保守性の高い定数管理

実装を通じて、状態パターンの強力さを実感しました。各状態が独立しているため、DefaultStateの実装変更が他の状態に影響せず、新しい状態の追加も容易です。また、落下距離計測や二段ジャンプなど、細かいゲームプレイ要素もこの1つのクラスで完結しており、保守性の高い設計となっています。