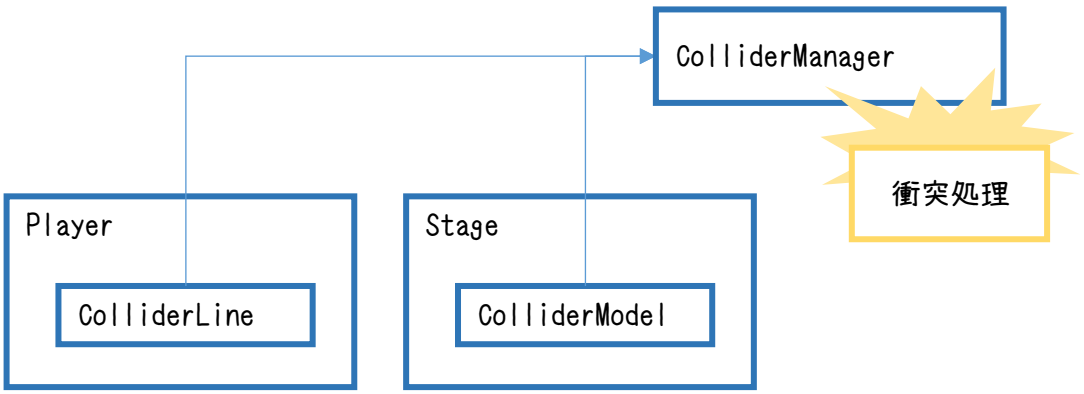
オブジェクト指向的な衝突処理

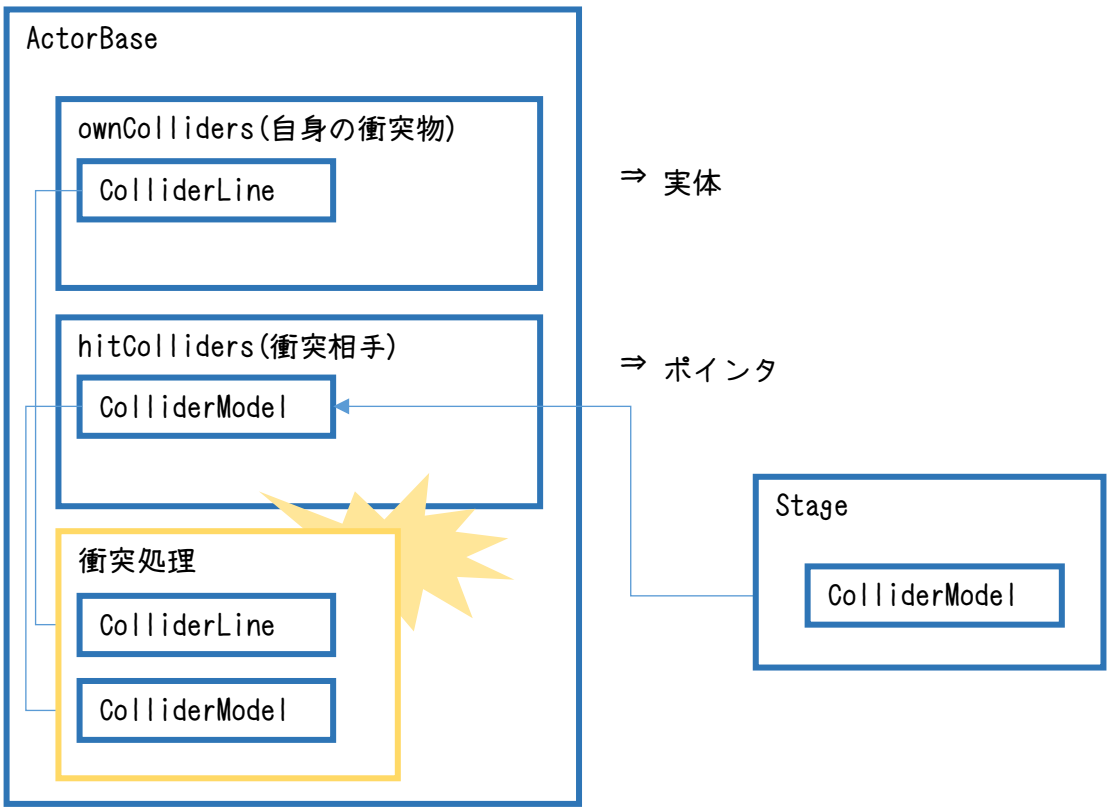
具体的な衝突処理を実装していきます。

すぐに思いつくのが、
ColliderManagerを作って、各ActorのColliderを収集して、
衝突管理を行わせる、オブジェクト指向的な衝突設計④のやり方ですが、

設計④



まだまだ下準備が足りませんので、予定通り、設計③で実装していきます。



衝突結果を自分自身にすぐに伝達できるので、制御しやすい。

①衝突相手の登録

ActorBase.h

```
class ActorBase
{

public:

    ～ 省略 ～

    // 衝突対象となるコライダを登録
    void AddHitCollider(const ColliderBase* hitCollider);

    // 衝突対象となるコライダをクリア
    void ClearHitCollider(void);

protected:

    ～ 省略 ～

    // 衝突相手の情報
    std::vector<const ColliderBase*> hitColliders_;
```

ActorBase.cpp

```
const ColliderBase* ActorBase::GetOwnCollider(int key) const
{
    if (ownColliders_.count(key) == 0)
    {
        return nullptr;
    }
    return ownColliders_.at(key);
}

void ActorBase::AddHitCollider(const ColliderBase* hitCollider)
{
    for (const auto& c : hitColliders_)
    {
```

```
            if (c == hitCollider)
            {
                return;
            }
        }
        hitColliders_.emplace_back(hitCollider);
    }

    void ActorBase::ClearHitCollider(void)
    {
        hitColliders_.clear();
    }
```

GameScene.cpp

```
    void GameScene::Init(void)
    {

        ～ 省略 ～

        // ステージモデルのコライダーをプレイヤーに登録
        const ColliderBase* stageCollider =
            stage_->GetOwnCollider(static_cast<int>(Stage::COLLIDER_TYPE::MODEL));
        player_->AddHitCollider(stageCollider);

        ～ 省略 ～

    }
```

②衝突処理の実装と、処理フローの整理

Player.cpp

```
    void Player::UpdateProcess(void)
    {

        // 移動操作
        ProcessMove();

        // 移動処理
```

```
    transform_.pos = VAdd(transform_.pos, movePow_);

    ⇒ 削除
        移動前座標の確保であったり、処理フローを固めるために、
        衝突処理の直前で移動を行いたいため、
        後述のCharactorBaseに移動させる。

}
```

CharactorBase.h

```
protected:

    ～ 省略 ～

    // 移動前の座標
    VECTOR prevPos_;

    // ジャンプ量
    VECTOR jumpPow_;

    ～ 省略 ～

    // 衝突判定
    void Collision(void);
    void CollisionGravity(void);

};
```

CharactorBase.cpp

```
void CharactorBase::Update(void)
{

    // 移動前座標を更新
    prevPos_ = transform_.pos;

    // 各キャラクターごとの更新処理
    UpdateProcess();

    // 移動方向に応じた遅延回転
```

```
    DelayRotate();

    // 重力による移動量
    CalcGravityPow();

    // 衝突判定
    Collision();

    // モデル制御更新
    transform_.Update();

    // アニメーション再生
    animationController_->Update();

    // 各キャラクターごとの更新後処理
    UpdateProcessPost();

}

void CharactorBase::CalcGravityPow(void)
{

    ～ 省略 ～

    // ジャンプ量を加算
    transform_.pos = VAdd(transform_.pos, jumpPow_);

    ⇒ 削除
        移動前座標の確保であったり、処理フローを固めるために、
        衝突処理の直前で移動を行いたいため、
        後述のCharactorBaseに移動させる。

}

void CharactorBase::Collision(void)
{

    // 移動処理
    transform_.pos = VAdd(transform_.pos, movePow_);

    // ジャンプ量を加算
```

```
    transform_.pos = VAdd(transform_.pos, jumpPow_);

    // 衝突(重力)
    CollisionGravity();

}

void CharactorBase::CollisionGravity(void)
{

    // 線分コライダ
    int lineType = static_cast<int>(COLLIDER_TYPE::LINE);

    // 線分コライダが無ければ処理を抜ける
    if (ownColliders_.count(lineType) == 0) return;

    // 線分コライダ情報
    ColliderLine* colliderLine_ =

    if (colliderLine_ == nullptr) return;
           dynamic_cast<ColliderLine*>(ownColliders_.at(lineType));
    // 線分の始点と終点を取得
    VECTOR s = colliderLine_->GetPosStart();
    VECTOR e = colliderLine_->GetPosEnd();

    // 登録されている衝突物を全てチェック
    for (const auto& hitCol : hitColliders_)
    {

        // ステージ以外は処理を飛ばす
        if (hitCol->GetTag() != ColliderBase::TAG::STAGE) continue;

        // 派生クラスへキャスト
        const ColliderModel* colliderModel =
               dynamic_cast<const ColliderModel*>(hitCol);

        if (colliderModel == nullptr) continue;

        // ステージモデル(地面)との衝突
        auto hit = MV1CollCheck_Line(
               colliderModel->GetFollow()->modelId, -1, s, e);
```

```
        if (hit.HitFlag > 0)
        {

            // 衝突地点から、少し上に移動
            transform_.pos =
                    VAdd(hit.HitPosition, VScale(AsoUtility::DIR_U, 2.0f));

            // ジャンプリセット
            jumpPow_ = AsoUtility::VECTOR_ZERO;

        }

    }

}
```

重力も復活させたので、地形に合わせて移動ができるようになっている。