AI（キャラクター制御）

ノーマル（最初から）

ステートてっ何？から
簡単なFSMまでの話

ハード（応用）

より実践的な実装へ
HFSMの実装へ

ベリーハード（より発展的）

更に進化する
ビヘイビアツリーを作る

キャラクター制御に関する話の前に（解説）

キャラクター制御でよく聞くステートとは？

オブジェクトの振る舞いに関する実装がされたクラスのこと。

例えば 「ジャンプ状態」「走ってる状態」「攻撃状態」 等。

使うとどんなメリットがあるの？

- ステートという単位でクラスを実装していくため、一つのクラスが小さくなる。
- ゲーム中に必要な状態しか実行されてないので状況を把握しやすい。
- ステートごとでクラスが異なるため、実装者が複数人でも開発しやすい。

つまり最強！

まずは、ステートを使わないコードを見てみる

#仮のコード例

```
void Player::Update()
{
    if (止まってる状態？)
    {
        // 何らかの処理
    }

    if (歩いてる状態？)
    {
        // 何らかの処理
    }

    if (ジャンプ中？)
    {
        // 何らかの処理
    }
}
```

このようにステートを使わないと各状態を管理するために、
沢山のフラグや事象をチェックする必要があり、
処理を記述している部分もドンドン増えていくため、
大変複雑な処理になっていくのは想像がつくと思います。

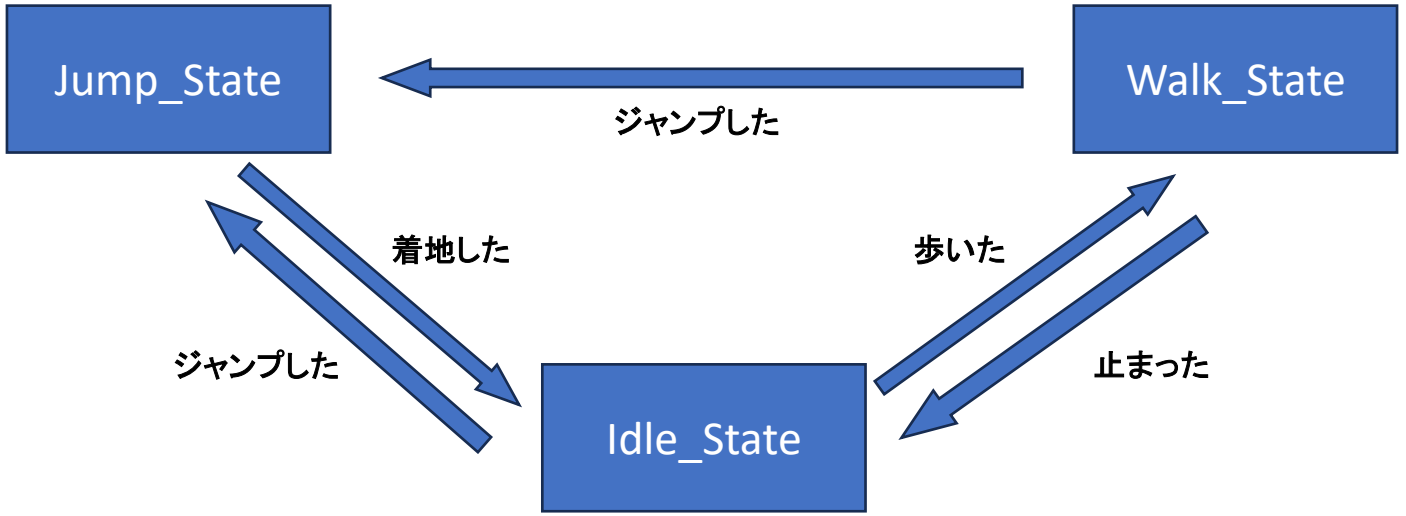
#プラスα

「見にくいのは関数分けしていないからだろ？」って、
鋭い方は思うかと思いますが関数やクラス分けしても、
右のような関係が改善しているわけではないので、
コードの意味としては何も変わらない事が重要な点です。

じゃあ、ステートを使うとどんな感じになるの？（解説）

状態管理に関した処理が毎回実行されたり、フラグの中にフラグがあって非常に見らいです。

また複数の状態が一つのクラスに存在するため、バグが発生しやすいです。



非常にざっくりした内容ですが、ステートを使用すると上記の問題を改善することが可能です。

上記の図では、クラスを3っつの状態に分け、状態遷移で切り変えて使っているイメージです。

↑「ん～？分からん」って方いると思いますが、実装例から順に説明していくので頑張ってください。

じゃあ、ステートを使うとどんな感じになるの？１（実装例）

#完成イメージ

```
7    class Player;
8
9    class IdleState : public State<Player>
10   {
11   public:
12
13       IdleState(Player* context) : State(context) {}
14
15       void OnEnter()  override { printf("OnEnter Idle\n"); }
16       void OnUpdate() override { printf("OnUpdate Idle\n"); }
17       void OnExit()   override { printf("OnExit Idle\n"); }
18   };
19
20   class JumpState : public State<Player>
21   {
22   public:
23
24       JumpState(Player* context) : State(context) {}
25
26       void OnEnter()  override { printf("OnEnter Jump\n"); }
27       void OnUpdate() override { printf("OnUpdate Jump\n"); }
28       void OnExit()   override { printf("OnExit Jump\n"); }
29   };
30
```

今回はプレイヤーの状態を管理するため、

プレイヤーの止まっている状態（IdleStateクラス）と、

プレイヤーがジャンプしている状態（JumpStateクラス）の

クラスを定義しています。

じゃあ、ステートを使うとどんな感じになるの？2（実装例）

#完成イメージ

```
35  class Player
36  {
37  public:
38
39      Player() : m_stateMachine(this)
40      {
41          // 最初のステートをセット
42          m_stateMachine.ChangeState<IdleState>();
43      }
44
45      void Update()
46      {
47          // IdleStateの更新関数が呼ばれる。
48          m_stateMachine.Update();
49
50          // IdleState -> JumpState へ状態遷移させる。
51          m_stateMachine.ChangeState<JumpState>();
52
53          // 状態が変わったため、JumpStateの更新関数が呼ばれる。
54          m_stateMachine.Update();
55      }
56
57  private:
58
59      // ステート管理クラス
60      StateMachine<Player> m_stateMachine;
61  };
62
63  int main()
64  {
65      // テスト用に一回更新関数を呼ぶ
66      Player player;
67      player.Update();
68  }
```

Playerクラスでは、

ステート管理クラスの初期化（IdleStateの状態に）し、

更新関数内でステートをJumpStateへ変更しています。

```
OnEnter Idle
OnUpdate Idle
OnExit Idle
OnEnter Jump
OnUpdate Jump

C:\Users\hinan\OneDrive\デスクト
 0 で終了しました。
デバッグが停止したときに自動的に
```

上記はサンプルの動作結果です。

ちゃんと状態遷移出来ていることが分かると思います。

じゃあ、ステートを使うとどんな感じになるの？３（実装例）

#簡単な実装例

```
#pragma once

template<class T>
class State
{
public:

    State(T* context)
    {
        m_context = context;
    }

    virtual ~State() {}

    // ステートに遷移した時に呼び出される関数
    virtual void OnEnter() = 0;

    // ステートの更新中呼び出される関数
    virtual void OnUpdate() = 0;

    // ステートが終了する時に呼び出される関数
    virtual void OnExit() = 0;

private:

    T* m_context;
};
```

状態を表すStateクラスでは、

ステート遷移時に呼び出されるOnEnter、OnExit関数と、

ステート更新用関数を定義しています。

m_contextは所属するオブジェクト（ Player とか）への

アクセス用に保持しています。

じゃあ、ステートを使うとどんな感じになるの？４（実装例）

#簡単な実装例

```cpp
#pragma once
#include <memory>
#include "State.h"

template<class T>
class StateMachine
{
public:

    StateMachine(T* context)
    {
        m_context = context;
    }

    // 現在のステートを更新する。
    void Update()
    {
        if (m_currentState)
        {
            m_currentState->OnUpdate();
        }
    }

    template<class T>
    void ChangeState()
    {
        // 既にステートが入ってたら終了処理を呼ぶ。
        if (m_currentState)
        {
            m_currentState->OnExit();
        }

        // 新しくステートを生成して初期化している。
        m_currentState.reset(new T(m_context));
        m_currentState->OnEnter();
    }

private:

    // このクラスを保持するオブジェクト（Playerとか）
    T* m_context;

    // 現在セットされたステートオブジェクト
    std::unique_ptr<State<T>> m_currentState;
};

```

メインとなるステート管理クラスの実装です。

特に重要な所はChangeState関数で、

ここで現在の状態から指定された状態に変化しています。

その他は、コンストラクタで所有者へのポインタを取得。

ステート更新用のUpdate関数が実装されています。

#プラスα

最初の解説の所でいうと、ChangeState関数で

IdleStateからJumpStateへ状態遷移しているという感じです。

↑「ん～あんまりピントこない？」って人は、サンプルを動作させて確認してみてください。

ステートを使うとどんな感じになった？

簡単なサンプルでしたが、状態遷移のイメージはつかめたのではないでしょうか？

またPlayerクラスの実装が各状態クラスに分割され実装内容が減ったことで、

コードが非常に見やすくなるのも理解出来たと思います。

先ほどの実装例では、ChangeState関数を使用すると全ての状態に遷移出来ました。

しかし実際にゲームで使う場合は、

状態遷移図を元に専用のトリガー設定を持たせた方が良いと思います。これがないと

想定外の状態遷移が起こる可能性がありますし、直接型を触るのもあまり良くないです。

次は状態遷移のルールを作り遷移先を指定していきます。

これらは有限ステートマシン（FSM）という名前の実装になります。

遷移条件をつける！1（実装例）

#完成イメージ

```
26  };
27
28  // 状態遷移の発動トリガー
29  enum EventTrigger
30  {
31      ToJump
32  };
33
34  class Player
35  {
36  public:
37
38      Player() : m_stateMachine(this)
39      {
40          // 使用する状態の追加
41          m_stateMachine.AddState<IdleState>();
42          m_stateMachine.AddState<JumpState>();
43
44          // IdleStateからJumpStateに遷移する条件を登録
45          m_stateMachine.AddTransition<IdleState, JumpState>(EventTrigger::ToJump);
46
47          // 最初のステートをセット
48          m_stateMachine.SetStartState<IdleState>();
49      }
50
51      void Update()
52      {
53          // IdleStateの更新関数が呼ばれる。
54          m_stateMachine.Update();
55
56          // IdleState -> JumpState へ状態遷移させるトリガーを送信。
57          m_stateMachine.SendTrigger(EventTrigger::ToJump);
58
59          // 状態が変わったため、JumpStateの更新関数が呼ばれる。
60          m_stateMachine.Update();
61      }
62
63  private:
64
65      // ステート管理クラス
66      StateMachine<Player> m_stateMachine;
67  };
```

仮コードでは先ほどまでのサンプルから、

Playerの初期化時に

使用される全ての状態を登録し、状態遷移の設定も登録

するように変更しています。

その後、ChangeState関数を呼んでいた所を、

設定した専用のeventIdを発行して状態遷移しています。

※実行結果は変わらないので割愛します。

遷移条件をつける！２（実装例）

#簡単な実装例

```
8    State(T* context)
9    {
10       m_context = context;
11   }
12
13   virtual ~State() {}
14
15   // ステートに遷移した時に呼び出される関数
16   virtual void OnEnter() = 0;
17
18   // ステートの更新中呼び出される関数
19   virtual void OnUpdate() = 0;
20
21   // ステートが終了する時に呼び出される関数
22   virtual void OnExit() = 0;
23
24   // 状態遷移の条件を追加する関数
25   // このオブジェクトからの遷移条件だけを追加する。
26   void AddTransition(int eventId, int stateId)
27   {
28       m_transitionMap[eventId] = stateId;
29   }
30
31   // 状態遷移の条件が設定されていれば遷移先のstateIdを返します。
32   bool TryGetTransition(int eventId, int& stateId)
33   {
34       auto hasEvent = m_transitionMap.contains(eventId);
35       if (hasEvent == true)
36       {
37           stateId = m_transitionMap[eventId];
38       }
39       return hasEvent;
40   }
41
42 private:
43
44   T* m_context;
45
46   // 各状態遷移トリガーと遷移先の状態IDへのマップ
47   std::map<int, int> m_transitionMap;
48 };
```

まずは状態を表すStateクラスの変更です。

遷移ルールを保持させる変数（m_transtionMap）と、
設定を登録するAddTransition関数を定義しています。

今回のサンプルでは、
状態のIDとトリガーのIDを使って遷移設定を登録します。

そして、eventIdから状態遷移先のIDを返す
TryGetTransition関数を定義して、
戻り値で所持の有無も返すようにしています。

※戻り値で所持の有無を返すことで、呼び出し側で正しく取得出来たか判断できます。

遷移条件をつける！３（実装例）

#簡単な実装例

```
    }

    // 初期化時の状態遷移をセットする関数
    template<class T>
    void SetStartState()
    {
        auto stateId  = typeid(T).hash_code();
        auto hasState = m_stateMap.contains(stateId);

        if (hasState == false)
        {
            return;
        }

        // 状態が登録されていれば遷移
        ChangeState(stateId);
    }

    // 現在ステートを保持していれば更新する。
    void Update()
    {
        if (m_currentState.expired() == false)
        {
            m_currentState.lock()->OnUpdate();
        }
    }

    // 新しく管理する状態を追加する関数
    template<class T>
    void AddState()
    {
        auto stateId  = typeid(T).hash_code();
        auto hasState = m_stateMap.contains(stateId);

        if (hasState == true)
        {
            return;
        }

        m_stateMap[stateId] = std::make_shared<T>(m_context);
    }
```

そして、メインとなるStateMachineクラスの変更点です。
多いので、分割して説明していきます。

SetStartState関数で最初のステートを設定出来るようにしています。（具体的には型から状態IDに変換している）

更新関数も少し変わっていますが、これはCurrentStateの管理方法をWeekPtrを使用するように変更したからです。

AddState関数で指定された状態IDとそのオブジェクトを、ステートマシンに登録しています。

遷移条件をつける！４（実装例）

#簡単な実装例

```
55
56      // From状態からTo状態へ状態遷移する設定を登録する関数
57      template<class From, class To>
58      void AddTransition(int eventId)
59      {
60          auto fromStateId = typeid(From).hash_code();
61          auto toStateId   = typeid(To).hash_code();
62
63          // 両方の状態が登録されているかを取得。
64          auto hasState = m_stateMap.contains(fromStateId) && m_stateMap.contains(toStateId);
65          if (hasState == false)
66          {
67              return;
68          }
69
70          m_stateMap[fromStateId]->AddTransition(eventId, toStateId);
71      }
72
73      void SendTrigger(int eventId)
74      {
75          // 遷移条件が登録されていればChageState関数を呼ぶ。
76          int stateId = 0;
77          if (m_currentState.lock()->TryGetTransition(eventId, stateId))
78          {
79              ChangeState(stateId);
80          }
81      }
82
83  private:
84
85      void ChangeState(int stateId)
86      {
87          // 既にステートが入ってたら終了処理を呼ぶ。
88          if (m_currentState.expired() == false)
89          {
90              m_currentState.lock()->OnExit();
91          }
92
93          // 新しくステートを生成して初期化している。
94          m_currentState = m_stateMap[stateId];
95          m_currentState.lock()->OnEnter();
96      }
97
```

AddTransition関数ではevetIdが渡された時に、

From状態からTo状態へ遷移する設定を登録しています。

SnedTrigger関数では、eventIdを元に現在の状態から、

遷移出来るかを取得し設定があればstateIdに遷移します。

ChangeState関数では先ほどまでの実装とは違い、

stateIdから既に生成済みの状態へ遷移しています。

#プラスα

「さっきからTypeid(T).hash_code()って何？」と

思うかもしれませんが、

これはクラスに付与された専用のIdを取得しているだけです。

遷移条件をつける！５（実装例）

#簡単な実装例

```
86      {
87          // 既にステートが入ってたら終了処理を呼ぶ。
88          if (m_currentState.expired() == false)
89          {
90              m_currentState.lock()->OnExit();
91          }
92
93          // 新しくステートを生成して初期化している。
94          m_currentState = m_stateMap[stateId];
95          m_currentState.lock()->OnEnter();
96      }
97
98  private:
99
100         // このクラスを保持するオブジェクト（Playerとか）
101         T* m_context;
102
103         // 現在セットされたステートオブジェクト
104         std::weak_ptr<State<T>> m_currentState;
105
106         // 各状態のIDとオブジェクトへのマップ
107         std::map<int, std::shared_ptr<State<T>>> m_stateMap;
108 };
```

StateMachineクラスの変数の変更と追加は、

トリガーから状態遷移させるため、

状態Idとそのオブジェクトを保持する（m_stateMap）と、

先ほどまでのcurrentStateをweak_ptrに変更しています。

#プラスα

「weak_ptrって何？」と思う人もいるかもしれませんが、

m_stateMapでオブジェクトを保持しているため、

そのオブジェクトへ安全にアクセスするクラスという感じです。

遷移条件をつけてみて

簡単なサンプルでしたが、有限ステートマシンの実装ができたと思います。
便利で使いやすくなり、一つのクラス実装が小さくなってコードもシンプルになりました。

しかも、実行中に状態や遷移条件を追加出来るため使い方によっては、
ゲーム中にアビリティやスキルの付与するといったことも出来ると思います。

しかし、そんな有限ステートマシン（FSM）にはいくつか問題があります。
次の応用ではその問題解決に向けた話をしていきます。

キャラクター制御（応用）

1. 有限ステートマシンの問題とその解決について解説
2. 更なる進化、階層型ステートマシン（HFSM）へ

有限ステートマシンの問題とその解決1（解説）

先ほどまでの実装（有限ステートマシン）では問題があると話しました。

何が問題かというと、

ステート管理される状態が一つのため、同じ処理があったとしても使い回しが出来ない。

状態が変化する度に関連する全ての遷移条件を再評価する必要があり、保守と拡張がしずらい。

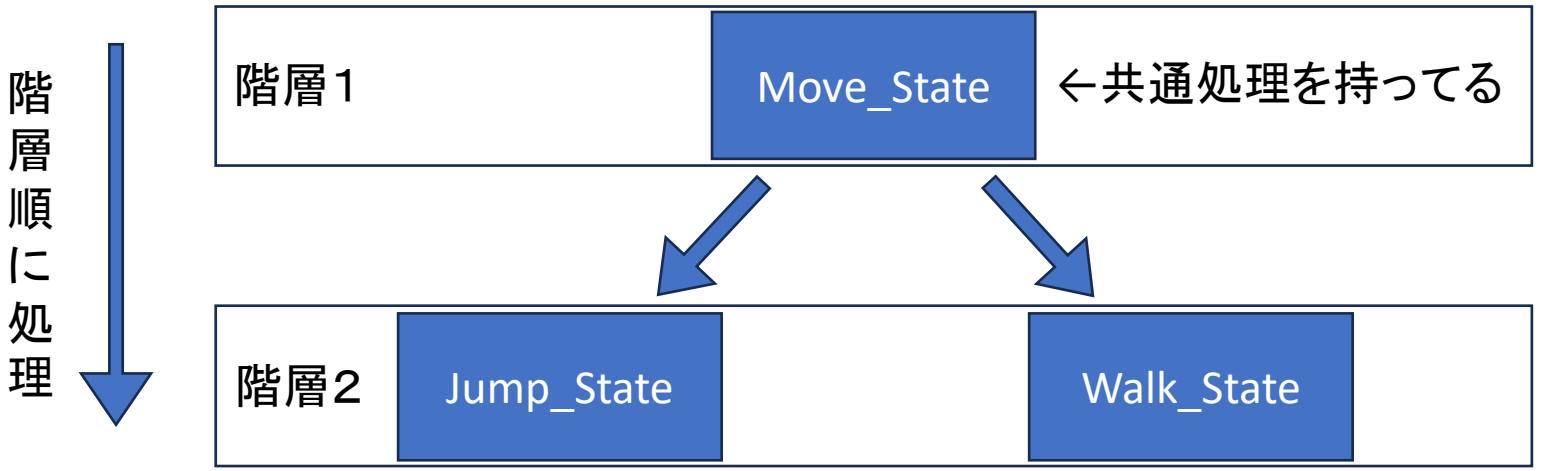
これだけ見ても「ん～？分からん」ってなると思うので順にお話していきます。

先に比較的簡単に、問題解決出来る前者の問題について解説、実装していきます。

有限ステートマシンの問題とその解決2（解説）

先ほどの同じ処理であっても使い回しが出来ず、

コードが重複してしまう問題は**階層型のステート管理にすると解決出来る**と思います。



非常にざっくりした内容ですが、

移動の共通処理を定義したMoveの下階層に、JumpやWalkなどの別ステートが存在する。

というように各ステートの中にも、また別のステートマシンが存在しているというイメージです。

ステートマシンを階層構造にする！１（実装例）

#完成イメージ

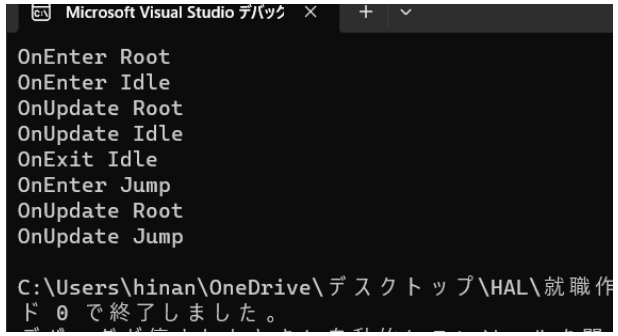
```
33
34  class RootState : public State<Player>
35  {
36  public:
37
38      RootState(Player* context) : State(context)
39      {
40          // 使用する状態の追加
41          AddChildState<IdleState>();
42          AddChildState<JumpState>();
43
44          // IdleStateからJumpStateに遷移する条件を登録
45          AddChildTransition<IdleState, JumpState>(EventTrigger::ToJump);
46
47          // 最初のステートをセット
48          SetCurrentState<IdleState>();
49      }
50
51      void OnEnter()  override { printf("OnEnter Root\n"); }
52      void OnUpdate() override { printf("OnUpdate Root\n"); }
53      void OnExit()   override { printf("OnExit Root\n"); }
54  };
55
56  class Player
57  {
58  public:
59
60      Player() : m_stateMachine(this)
61      {
62          // 使用する状態の追加
63          m_stateMachine.AddState<RootState>();
64
65          // 最初のステートをセット
66          m_stateMachine.SetStartState<RootState>();
67      }
68
69      void Update()
70      {
71          // IdleStateの更新関数が呼ばれる。
72          m_stateMachine.Update();
73
74          // IdleState -> JumpState へ状態遷移させるトリガーを送信。
75          m_stateMachine.SendTrigger(EventTrigger::ToJump);
76
77          // 状態が変わったため、JumpStateの更新関数が呼ばれる。
78          m_stateMachine.Update();
79      }
80
81  private:
82
83      // ステート管理クラス
84      StateMachine<Player> m_stateMachine;
85  };
86
```

仮コードでは先ほどまでのサンプルから、

Playerの初期化時に処理していた、

使用する状態の登録と、遷移条件の設定を、階層の一番上の状態RootStateクラスで行うように変更して、階層の動作チェックを行います。

その他は<u>ステートマシンの初期化で、RootStateを追加、セット</u>しています。



左は実行結果です。
階層順に処理が実行されているのが確認出来ると思います。

ステートマシンを階層構造にする！２（実装例）

#完成イメージ

```cpp
#pragma once
#include <map>
#include <memory>

template<class T>
class State
{
public:

    State(T* context)
    {
        m_context = context;
    }

    virtual ~State() {}

    // ステートに遷移した時に呼び出される関数
    virtual void OnEnter() = 0;

    // ステートの更新中呼び出される関数
    virtual void OnUpdate() = 0;

    // ステートが終了する時に呼び出される関数
    virtual void OnExit() = 0;

    // 現在ステートをセットする初期化用関数
    template<class T>
    void SetCurrentState()
    {
        auto stateId  = typeid(T).hash_code();
        auto hasState = m_childStateMap.contains(stateId);

        if (hasState == false)
        {
            return;
        }

        m_currentState = m_childStateMap[stateId];
    }
```

今回のメインとなるStateクラスの変更です。

非常に多いので分割して説明していきます。

最初に簡単に説明すると、

先ほどまでのStateMachineクラスの処理をStateクラスが持つようなる感じです。

基本的な要素は先ほどまでのサンプルと同じです。

SetCurrentStateでステートが持つ子ステートをセットし、初期化しています。

ステートマシンを階層構造にする！３（実装例）

#完成イメージ

```
41      // 初期化時はルートから順にOnEnterを
42      void SendEnter()
43      {
44          OnEnter();
45
46          // 現在ステートが設定されていれば通知
47          if (m_currentState.expired() == false)
48          {
49              m_currentState.lock()->SendEnter();
50          }
51      }
52
53      // 更新時はルートから順にOnUpdateを呼んでいく。
54      void SendUpdate()
55      {
56          OnUpdate();
57
58          // 現在ステートが設定されていれば通知
59          if (m_currentState.expired() == false)
60          {
61              m_currentState.lock()->SendUpdate();
62          }
63      }
64
65      // 終了時は末端ステートから順にOnExitを呼んでいく。
66      void SendExit()
67      {
68          // 現在ステートが設定されていれば通知
69          if (m_currentState.expired() == false)
70          {
71              m_currentState.lock()->SendExit();
72          }
73
74          OnExit();
75      }
76
```

そして、**各Send関数で階層順に処理を実行**しています。

SendExit関数だけは末端状態から終了させています。

#プラスα

どちらでもいいのですがStateMachineクラスで、

Whileなどを使いCurrentStateに再帰的にアクセスして、

全てのUpdateなどの各処理を呼ぶ方がいいと思います。

ステートマシンを階層構造にする！ ４（実装例）

#完成イメージ

```
80
81      // 新しく管理する子状態を追加する関数
82      template<class T>
83      void AddChildState()
84      {
85          auto stateId  = typeid(T).hash_code();
86          auto hasState = m_childStateMap.contains(stateId);
87
88          if (hasState == true)
89          {
90              return;
91          }
92
93          m_childStateMap[stateId] = std::make_shared<T>(m_context);
94      }
95
96      // From状態からTo状態へ状態遷移する設定を登録する関数
97      template<class From, class To>
98      void AddChildTransition(int eventId)
99      {
100         auto fromStateId = typeid(From).hash_code();
101         auto toStateId   = typeid(To).hash_code();
102
103         // 両方の状態が登録されているかを取得。
104         auto hasState = m_childStateMap.contains(fromStateId) && m_childStateMap.contains(toStateId);
105         if (hasState == false)
106         {
107             return;
108         }
109
110         m_childStateMap[fromStateId]->AddTransition(eventId, toStateId);
111     }
112
113     // Triggerを通知して設定をチェックし、遷移する関数
114     void SendTrigger(int eventId)
115     {
116         if (m_currentState.expired())
117         {
118             return;
119         }
120
121         // 遷移条件が登録されていればChageState関数を呼ぶ。
122         int stateId = 0;
123         if (m_currentState.lock()->TryGetTransition(eventId, stateId))
124         {
125             ChangeState(stateId);
126         }
127
128         m_currentState.lock()->SendTrigger(eventId);
129     }
130
```

名前にChildがついていますが、

AddChildTransition関数とAddChildState関数のは

先ほどまでのサンプルでStateMachineクラスで実装していたものと同じ処理です。

SnedTrigger関数は、

先ほど同様に階層順に再帰的に呼び出しています。

ステートマシンを階層構造にする！５（実装例）

#完成イメージ

```
128
129     // 状態遷移の条件を追加する関数
130     // このオブジェクトからの遷移条件だけを追加する。
131     void AddTransition(int eventId, int stateId)
132     {
133         m_transitionMap[eventId] = stateId;
134     }
135
136     // 状態遷移の条件が設定されていれば遷移先のstateIdを返します。
137     bool TryGetTransition(int eventId, int& stateId)
138     {
139         auto hasEvent = m_transitionMap.contains(eventId);
140         if (hasEvent == true)
141         {
142             stateId = m_transitionMap[eventId];
143         }
144         return hasEvent;
145     }
146
147 private:
148
149     void ChangeState(int stateId)
150     {
151         // 既にステートが入ってたら終了処理を呼ぶ。
152         if (m_currentState.expired() == false)
153         {
154             m_currentState.lock()->SendExit();
155         }
156
157         // 新しくステートを生成して初期化している。
158         m_currentState = m_childStateMap[stateId];
159         m_currentState.lock()->SendEnter();
160     }
161
```

AddTransition関数とTryGetTransition関数は変わらず、

追加したChangeState関数はStateMachineクラスに

実装していたものと同様ですね。

ステートマシンを階層構造にする！６（実装例）

#完成イメージ

```
146
147            // 新しくステートを生成して初期化している。
148            m_currentState = m_childStateMap[stateId];
149            m_currentState.lock()->SendEnter();
150        }
151
152    private:
153
154        T* m_context;
155
156        // 現在セットされた子供ステートオブジェクト
157        std::weak_ptr<State<T>> m_currentState;
158
159        // 各状態遷移トリガーと遷移先の状態IDへのマップ
160        std::map<int, int> m_transitionMap;
161
162        // 各状態のIDと子供ステートオブジェクトへのマップ
163        std::map<int, std::shared_ptr<State<T>>> m_childStateMap;
164    };
```

こちらも同様で、

StateMachineクラスが持っていた

現在セットされている子供ステート（m_currentState）と、

状態Idとその子供ステートへのマップ（m_childStateMap）

変数が追加されています。

ステートマシンを階層構造にする！ 7（実装例）

#完成イメージ

```
65              return;
66          }
67
68          m_stateMap[fromStateId]->AddTransition(eventId, toStateId);
69      }
70
71      void SendTrigger(int eventId)
72      {
73          if (m_currentState.expired())
74          {
75              return;
76          }
77
78          // 遷移条件が登録されていればChageState関数を呼ぶ。
79          int stateId = 0;
80          if (m_currentState.lock()->TryGetTransition(eventId, stateId))
81          {
82              ChangeState(stateId);
83          }
84
85          m_currentState.lock()->SendTrigger(eventId);
86      }
87
88  private:
89
90      void ChangeState(int stateId)
91      {
92          // 既にステートが入ってたら終了処理を呼ぶ。
93          if (m_currentState.expired() == false)
94          {
95              m_currentState.lock()->SendExit();
96          }
97
98          // 新しくステートを生成して初期化している。
99          m_currentState = m_stateMap[stateId];
100         m_currentState.lock()->SendEnter();
101     }
102
```

StateMachineクラスはほとんど変更ありませんが、

OnEnterやOnUpdate、OnExitを呼んでいた所を、

各Send関数に変更しているだけです。

※左のサンプルにはありませんが、Updateも同様です。

#プラスα

「StateMachineクラスとStateクラスで共通処理があって汚い」

と思う人は色々やり方はあると思いますが、

StateMachineクラスも一つのStateクラスである、

という考えにすれば良いのではないでしょうか？

※今回はサンプルなので、説明を少なくするために色々やってないことがあります。

ステートマシンを階層構造にしてみて

簡単なサンプルでしたが、階層型ステートマシン（HFSM）の実装ができたと思います。

第一階層だけを使えば前回の有限ステートマシンと同様にですし、階層型になったことで更にステートを細分化が出来、処理の使い回しがしやすくなりました。

また処理を使い回しをしなくても、
階層型になったことでステートの管理がめちゃくちゃ楽になります。
特にゲームのPlayer制御なんかに最適かもしれません。

しかし、階層型にしただけなのでまだもう一つ問題が残っています。
次の発展ではその問題解決に向けた話をしていきます。（多分激ムズです）

キャラクター制御（より発展的）

1. 有限、階層型ステートマシンの問題解決について解説
2. 簡単にビヘイビアツリーを実装する

※非常にシンプルな実装にしているので実践で使う場合は改良が必要です。あとで解説します。

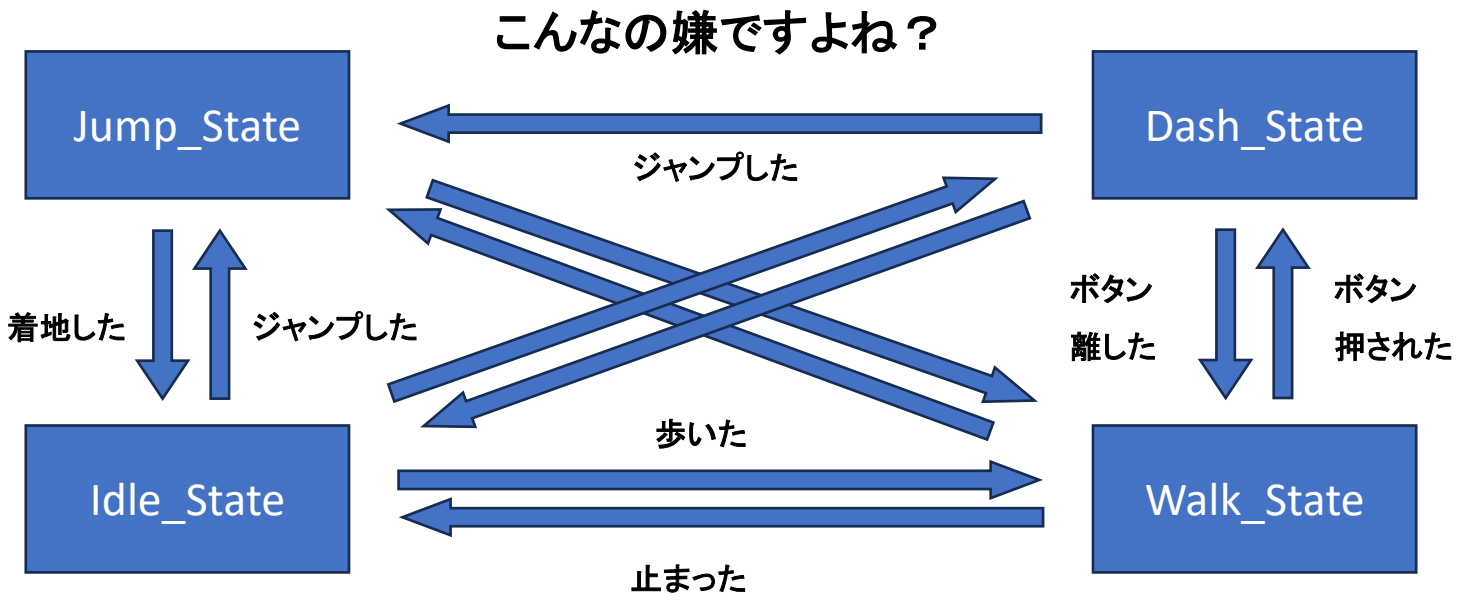
ステートマシンのもう一つの問題とその解決1（解説）

先ほどまでの実装（階層型ステートマシン）ではまだ問題が残っていると話しました。

残っている問題はというと、

状態が変化する度に関連する全ての遷移条件を再評価する必要があり、保守と拡張がしずらい。

つまり、**状態の遷移条件が複雑化し、管理しずらいということ**です。



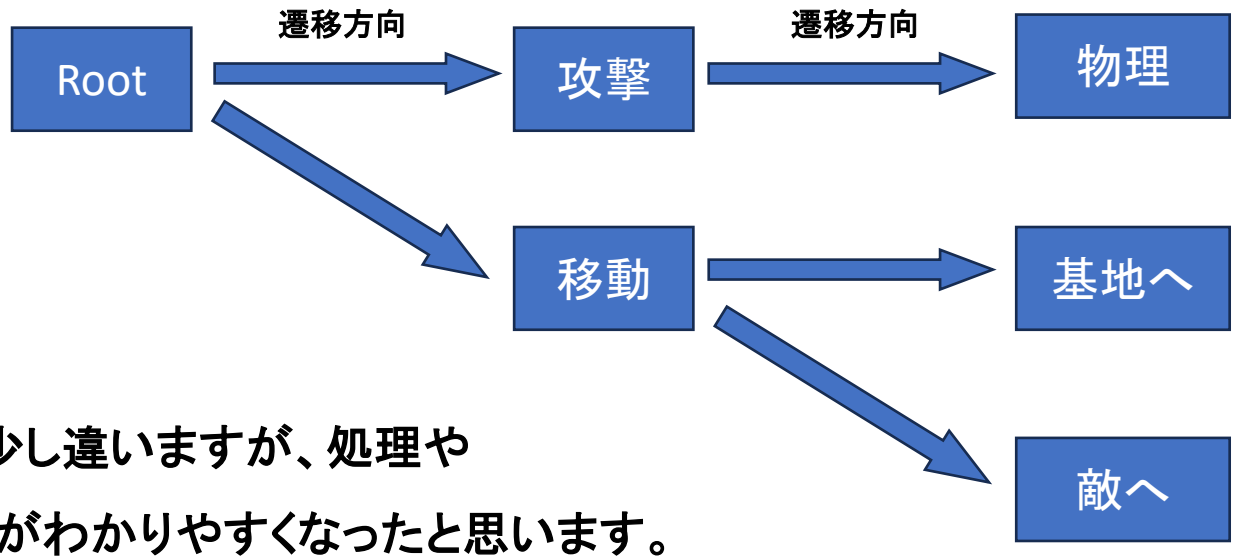
ステートマシンのもう一つの問題とその解決2（解説）

先ほどの解説より処理や条件が複雑化してしまうのは想像つくと思います。

「では、どうするのか？」というと、

先ほどまでのステートマシンでは、各ステートが循環して遷移してしまうため複雑になっています。

なので、一方向のみの状態遷移すれば処理の順番や制御がしやすくなると思いませんか？



先ほどの例とは少し違いますが、処理や

状態遷移の流れがわかりやすくなったと思います。

ステートマシンのもう一つの問題とその解決3（解説）

「どこかで見たことあるなぁ？」と思った人もいると思いますが、

そう！これが！あの！**ビヘイビアツリー**です！

さっそく実装と言いたいですが、知らない人のためにビヘイビアツリーのお話をします。

もし知っているという方はこちらから実装まで飛ばしてください。

ビヘイビアツリーってなに？（解説）

ビヘイビアツリーとは？

キャラクターの振る舞い（思考や行動）をツリー構造で構築し、

行動に至るまでの処理の流れなどを視覚的に分かりやすくしたものです。

ビヘイビアツリーを構築するのは「ノード」という単位で、

いくつかの基本的なノードを組み合わせながらツリーを構築していきます。

そして各状況の判断でたどり着いた末端ノードで行動を行うという感じです。

次はその基本的なノードについて解説します。

Actionノード（解説）

実際に「攻撃する」「移動する」「逃げる」といった処理を実行するノードです。

末端専用のノードであるため、子ノードを持ちません。

「成功」「失敗」「実行中」といった実行結果をそのまま戻り値として返します。

先ほどまでのステートでいう振る舞いを実装する部分です。

Conditionノード（解説）

「HPが足りてるか？」「視界に入っているか？」といった条件を判定するノードです。

末端専用のノードであるため、子ノードを持ちません。

条件結果で「成功」「失敗」といった結果を戻り値として返します。

Decoratorノード（解説）

こちらも条件判定をするノードです。

違うのは、Decoratorノードは一つだけ子ノードを持ちます。

条件に成功すれば、子ノードを実行しその結果を戻り値として返します。

また条件に失敗すれば、子ノードを実行せずに終了します。

Sequencerノード（解説）

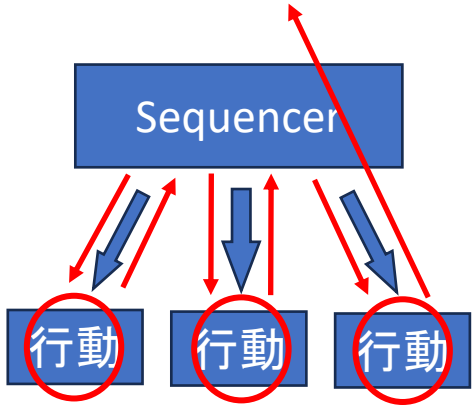
複数の子ノードを登録された順番に実行していくノードです。

子ノードが一つでも「失敗」すればノードを終了します。

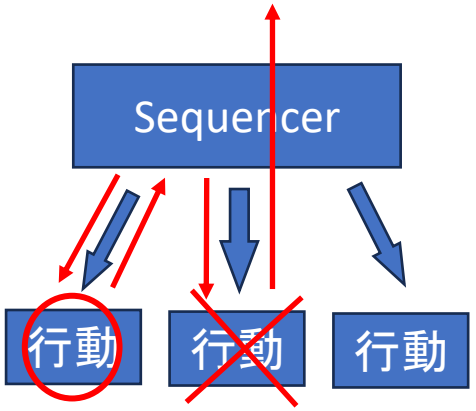
全ての子ノードが「成功」すれば「成功」としてノードを終了します。

使い方としては先ほどのActionノードやConditionノードを持たせることで、

「HPが〇〇以下になれば、特殊行動する」みたいな使い方ができます。



成功すれば次へ



失敗すれば終了

Selectorノード（解説）

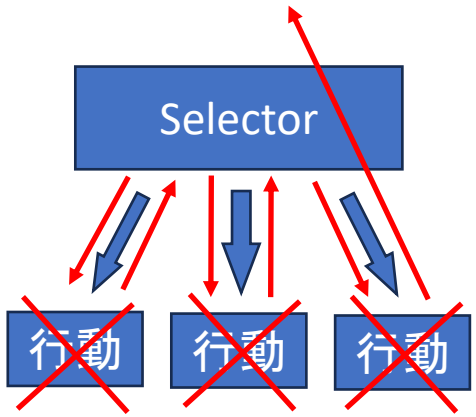
こちらも**複数の子ノードを登録された順番に実行していくノード**です。

子ノードが一つでも「成功」を返せばノードを終了します。

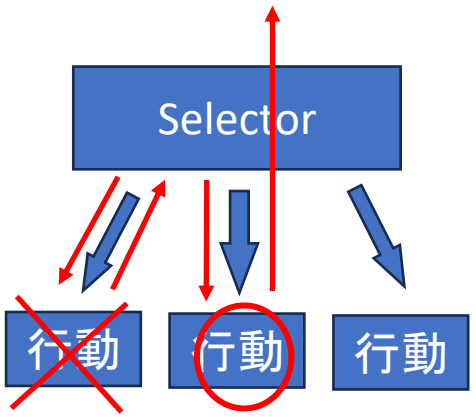
全ての子ノードが「失敗」すれば「失敗」としてノードを終了します。

使い方としては先ほどのActionノードやConditionノードを持たせることで、

「HPが〇〇以上なら通常攻撃、以下なら特殊攻撃する」みたいな使い方をします。



失敗すれば次へ



成功すれば終了

Repeaterノード（解説）

指定された回数分、子ノードを実行するノードです。

指定回数子ノードが実行されると「成功」を返します。

まだ子ノードを実行中の場合、「実行中」を親に返します。

Parallelノード（解説）

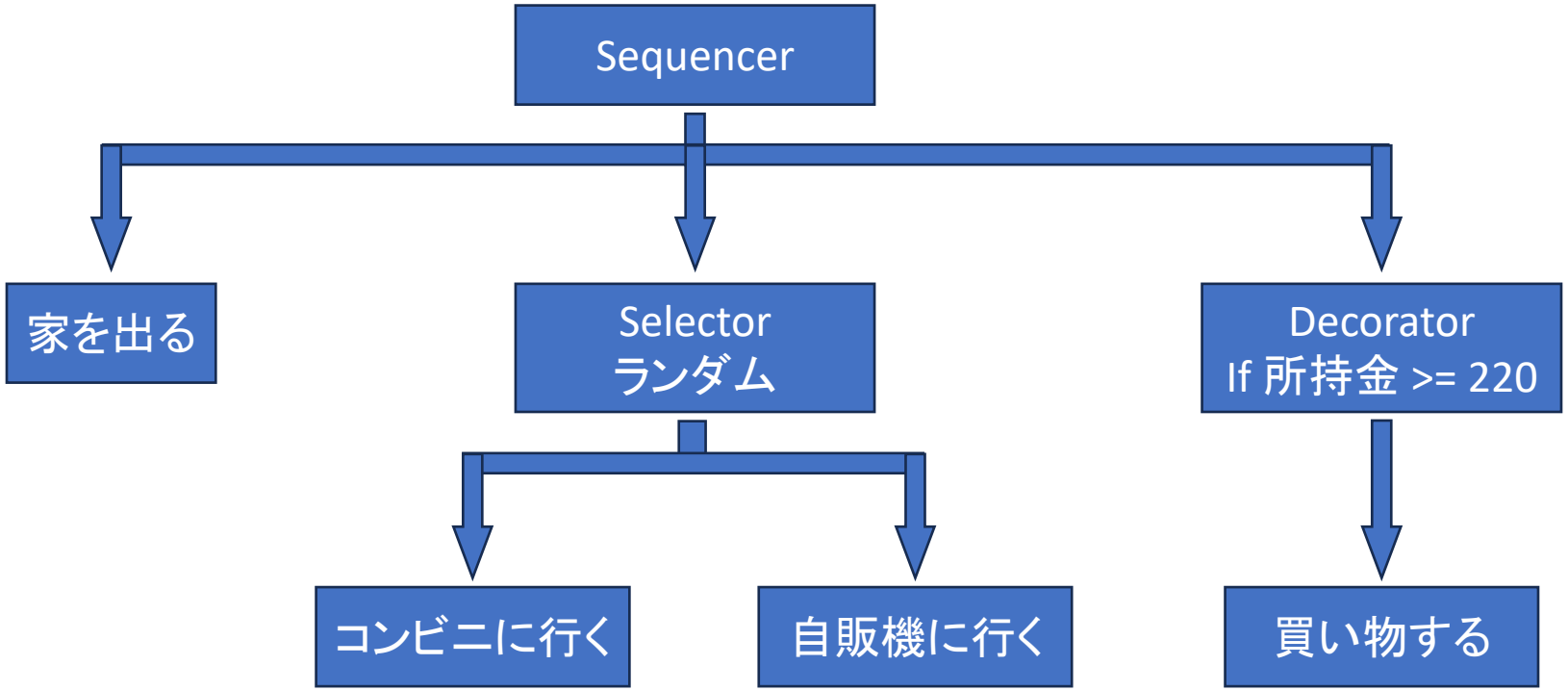
登録された全ての子ノードを同時に実行するノードです。

子ノードが一つでも「失敗」を返せばノードを終了します。
子ノードがまだ終了していない場合は「実行中」を返します。
そして、全ての子ノードが「成功」すれば、「成功」として親に返します。

使い方としては先ほどのActionノードと組み合わせて、
「移動しながら攻撃する」「逃げながら煙幕たく」みたいな使い方をします。

簡単なビヘイビアツリーを実装する！（成果物イメージ）

今回のサンプルはコードだけでは分かりにくいので、サンプルの流れを図で説明しています。



左から家を出て、行き先を決めて向かい、そしてお金があれば買い物するという流れで処理されます。

シンプルなものにしたので説明した全てのノードは使用していませんが、使い方のイメージはつかめると思います。

簡単なビヘイビアツリーを実装する！ 1（実装例）

#完成イメージ

```cpp
#include <iostream>
#include "BehaviorTree.h"
#include "SequencerNode.h"
#include "SelectorNode.h"
#include "DecoratorNode.h"
#include "ActionNode.h"

int main()
{
    // ランダム値を使用するのでシード値を初期化
    std::srand(std::time(nullptr));

    // 今回は買い物にいくビヘイビアツリーを実装します。
    int money = 220;
    BehaviorTree behaviorTree;

```

仮コードの先頭でサンプルで使用するため、

ランダムのシード値を初期化、

買い物用の所持金をセット、

そしてビヘイビアツリーオブジェクトの作成を行っています。

簡単なビヘイビアツリーを実装する！２（実装例）

#完成イメージ

```
17  // ルートノードを初期化
18  auto rootNode = std::make_shared<SequencerNode>();
19  behaviorTree.SetRootNode(rootNode);
20
21  // 家を出てコンビニを目指す。
22  auto action1 = std::make_shared<ActionNode>([]()
23  {
24      printf("家を出る\n");
25      return NodeState::Success;
26  });
27
28  rootNode->AddChild(action1);
29
30  // 近くのコンビニか自販機へ向かいます。
31  auto selector = std::make_shared<SelectorNode>();
32  rootNode->AddChild(selector);
33
34  // 選択にはランダム値を使用します。
35  auto decorator1 = std::make_shared<DecoratorNode>([&]()
36  {
37      return std::rand() % 2;
38  });
39
40  selector->AddChild(decorator1);
41
42  // コンビニに向かった。
43  auto action2 = std::make_shared<ActionNode>([&]()
44  {
45      printf("今日の気分でコンビニに行った。\n");
46      return NodeState::Success;
47  });
48
49  decorator1->AddChild(action2);
50
51  // 自販機に向かった。
52  auto action3 = std::make_shared<ActionNode>([&]()
53  {
54      printf("今日の気分で自販機に行った。\n");
55      return NodeState::Success;
56  });
57
58  selector->AddChild(action3);
```

コードベースのため非常に見にくいですが、

成果物イメージと同様のツリーを構築しています。

１：

ルートノードであるSequencerノードを生成及び登録。

２：

最初の家を出るアクションをルートへ追加。

３：

Selectorノードを使用してランダムでコンビニ、自販機へ向かうアクションを二つ作成しルートへ登録。

簡単なビヘイビアツリーを実装する！３（実装例）

#完成イメージ

```
60      // 買い物ができるかを調べる。
61      constexpr int PRICE = 220;
62      auto decorator2 = std::make_shared<DecoratorNode>([&]()
63      {
64          return money >= PRICE;
65      });
66
67      rootNode->AddChild(decorator2);
68
69      // エナジードリンクを購入する。
70      auto action4 = std::make_shared<ActionNode>([&]()
71      {
72          printf("エナジードリンクを購入。\n");
73          money -= PRICE;
74          printf("残り金額は%dです。\n", money);
75          return NodeState::Success;
76      });
77
78      decorator2->AddChild(action4);
79
80      // メインループ
81      while (true)
82      {
83          // ツリーの更新
84          behaviorTree.Update();
85
86          // 更新が終了したら終了する。
87          if (behaviorTree.IsFinished())
88          {
89              break;
90          }
91      }
92  }
```

4：

Decoratorノードを使って220円以上保持していれば、

買い物するアクションを作成及びルートへ登録。

5：

最後に構築したツリーを動作させ、

買い物が成功、もしくは失敗すれば終了します。

#プラスα

今回はコンソールベースのため、

ツリーの構築が非常に分かりにくいですが、

自前のEditor等作成すれば非常に使いやすくなると思います。

簡単なビヘイビアツリーを実装する！ ４（実装例）

#完成イメージ

```cpp
#pragma once
#include <memory>
#include "BaseNode.h"

class BehaviorTree
{
public:

    void SetRootNode(std::shared_ptr<Node> root)
    {
        m_root = root;
    }

    void Start()
    {
        m_state = NodeState::Running;
    }

    void Update()
    {
        // 既に全ての処理が終了している。
        if (IsFinished())
        {
            return;
        }

        if (m_root)
        {
            m_state = m_root->Execute();
        }
    }

    bool IsSuccess() { return m_state == NodeState::Success; }

    bool IsFailure() { return m_state == NodeState::Failure; }

    bool IsFinished() { return IsSuccess() || IsFailure(); }

private:

    // 木構造のルートノードを保持します。
    std::shared_ptr<Node> m_root;

    // このツリーの状態を保持します。
    NodeState m_state;
};
```

先にメインのビヘイビアツリーから説明します。

ビヘイビアツリーの処理は非常にシンプルで、

ルートノードのセット関数と、

ツリーの初期化、及び更新関数と、

現在のツリーの状態を取得する関数のみです。

メンバ変数はルートノードと、現在状態だけです。

簡単なビヘイビアツリーを実装する！５（実装例）

#完成イメージ

```cpp
#pragma once
#include <vector>
#include <memory>

enum class NodeState
{
    Waiting, // 待機中
    Running, // 処理中
    Success, // 成功
    Failure, // 失敗
};
```

各ノード、ツリーの状態を表すNodeState列挙で、

Waitingは実行待ち状態で、直ぐに実行出来る状態。

Runningは現在何かしらの処理を実行している状態。

Successはノードが成功して終了している状態。

Failureはノードが失敗して終了している状態。

を表しています。

簡単なビヘイビアツリーを実装する！６（実装例）

#完成イメージ

```cpp
class Node
{
public:

    virtual ~Node() {};

    // ノード開始前に呼び出される関数
    virtual void OnEnter() = 0;

    // ノード実行中に呼び出される関数
    virtual NodeState OnUpdate() = 0;

    // ノード終了時に呼び出される関数
    virtual void OnExit() = 0;

    // ノードの実行用関数
    NodeState Execute()
    {
        if (m_state != NodeState::Running)
        {
            OnEnter();
        }

        m_state = OnUpdate();

        if (m_state != NodeState::Running)
        {
            OnExit();
        }

        return m_state;
    }

    // ノードの再初期化のための関数
    void Reset()
    {
        m_state = NodeState::Waiting;
    }

    bool IsSuccess() { return m_state == NodeState::Success; }

    bool IsFailure() { return m_state == NodeState::Failure; }

    bool IsFinished() { return IsSuccess() || IsFailure(); }

protected:

    // 現在のノード状態を保持します。
    NodeState m_state;
};
```

各ノードクラスの基底クラスであるNodeクラスです。

各ノードの独自実装用に、

初期化、更新、終了関数を仮想関数として定義。

コードで実際に使用するのは、Execute関数で

Execute関数は現在状態にあった処理を実行します。

ノードの結果をリセットするReset関数を用意。

他は、ノードの状態を取得するための関数のみです。

簡単なビヘイビアツリーを実装する！ 7（実装例）

#完成イメージ

```
76
77  class InternalNode : public Node
78  {
79  public:
80
81      virtual ~InternalNode() {};
82
83      // 子ノードを追加する関数
84      void AddChild(std::shared_ptr<Node> node)
85      {
86          m_children.push_back(node);
87      }
88
89  protected:
90
91      // オブジェクトの子ノード
92      std::vector<std::shared_ptr<Node>> m_children;
93  };
94
95  class LeafNode : public Node
96  {
97  public:
98
99      virtual ~LeafNode() {};
100 };
```

先ほどのNodeクラスを親クラスとして、

子ノードを保持するInternalNodeクラス

子ノードを持たないLeafNodeクラスを定義しています。

#プラスα

ビヘイビアツリーを更に実践的にしていく場合に、

LeafNodeはともかく、InternalNodeは必須になると思います。

※後で説明します。

簡単なビヘイビアツリーを実装する！８（実装例）

#完成イメージ

```cpp
#pragma once
#include <functional>
#include "BaseNode.h"

class ActionNode : public LeafNode
{
public:

    ActionNode(std::function<NodeState()> action)
    {
        m_action = action;
    }

    void OnEnter() override
    {
        m_state = NodeState::Running;
    }

    NodeState OnUpdate() override
    {
        // 設定された処理の戻り値をそのまま返します。
        return m_action();
    }

    void OnExit() override
    {

    }

private:

    // 実際の振る舞い処理を保持する。
    std::function<NodeState()> m_action;
};
```

サンプルで使用している、Actionノードです。

コンストラクタで独自実装の関数を登録し、

更新時にそのまま結果を返すだけです。

簡単なビヘイビアツリーを実装する！９（実装例）

#完成イメージ

```cpp
#pragma once
#include <functional>
#include "BaseNode.h"

class ConditionNode : public LeafNode
{
public:

    ConditionNode(std::function<bool()> action)
    {
        m_action = action;
    }

    void OnEnter() override
    {
        m_state = NodeState::Running;
    }

    NodeState OnUpdate() override
    {
        // 条件判定が成功か失敗かを、列挙型に変換して返します。
        return m_action() ? NodeState::Success : NodeState::Failure;
    }

    void OnExit() override
    {

    }

private:

    // 条件判定の処理を保持する。
    std::function<bool()> m_action;
};
```

サンプルでは使用していませんが、Conditionノードです。

コンストラクタで独自実装の関数を登録し、

更新時に結果から「成功」「失敗」を返すだけです。

簡単なビヘイビアツリーを実装する！ 10（実装例）

#完成イメージ

```cpp
#pragma once
#include <functional>
#include "BaseNode.h"

class DecoratorNode : public InternalNode
{
public:

    DecoratorNode(std::function<bool()> action)
    {
        m_action = action;
    }

    void OnEnter() override
    {
        m_state = NodeState::Running;
    }

    NodeState OnUpdate() override
    {
        if (m_action())
        {
            // 条件判定が成功していれば、子ノードを処理しそのまま結果を返します。
            return m_children[0]->Execute();
        }
        else
        {
            // 条件判定が失敗したら終了。
            return NodeState::Failure;
        }
    }

    void OnExit() override
    {

    }

private:

    // 条件判定の処理を保持する。
    std::function<bool()> m_action;
};
```

サンプルでは使用している、Decoratorノードです。

コンストラクタで独自実装の関数を登録し、
更新時に結果から子ノードを実行及び、
「失敗」を返しています。

簡単なビヘイビアツリーを実装する！ 11（実装例）

#完成イメージ

```cpp
#pragma once
#include "BaseNode.h"

class SequencerNode : public InternalNode
{
public:

    void OnEnter() override
    {
        m_state = NodeState::Running;
        m_childIndex = 0;
    }

    NodeState OnUpdate() override
    {
        // 子ノードを順に更新していきます。
        // 実行中か失敗が返されたら終了します。
        for (auto& i = m_childIndex; i < m_children.size(); m_childIndex++)
        {
            auto child = m_children[i];
            auto state = child->Execute();

            // 成功したら次の子ノードへ
            if (state != NodeState::Success)
            {
                return state;
            }
        }

        // 全ての子ノードが成功すれば終了。
        return NodeState::Success;
    }

    void OnExit() override
    {

    }

private:

    // 現在処理されている子ノードインデックス。
    int m_childIndex;
};
```

サンプルでは使用している、Sequencerノードです。

更新関数で登録された順番に子ノードを実行していき、

処理が「実行中」「失敗」すれば終了、

処理が「成功」すれば次の子ノードを実行します。

全ての子ノードが「成功」すれば「成功」として終了します。

簡単なビヘイビアツリーを実装する！ 12（実装例）

#完成イメージ

```cpp
#pragma once
#include "BaseNode.h"

class SelectorNode : public InternalNode
{
public:

    void OnEnter() override
    {
        m_state = NodeState::Running;
        m_childIndex = 0;
    }

    NodeState OnUpdate() override
    {
        // 子ノードを順に更新していきます。
        // 実行中か成功が返されたら終了します。
        for (auto& i = m_childIndex; i < m_children.size(); m_childIndex++)
        {
            auto child = m_children[i];
            auto state = child->Execute();

            // 失敗したら次の子ノードへ
            if (state != NodeState::Failure)
            {
                return state;
            }
        }

        // 全ての子ノードが失敗すれば終了。
        return NodeState::Failure;
    }

    void OnExit() override
    {

    }

private:

    // 現在処理されている子ノードインデックス。
    int m_childIndex;
};
```

サンプルでは使用している、Sequencerノードです。

更新関数で登録された順番に子ノードを実行していき、
処理が「実行中」「成功」すれば終了、
処理が「失敗」すれば次の子ノードを実行します。

全ての子ノードが「失敗」すれば「失敗」として終了します。

簡単なビヘイビアツリーを実装する！ 13（実装例）

#完成イメージ

```cpp
#pragma once
#include <functional>
#include "BaseNode.h"

class RepeaterNode : public InternalNode
{
public:

    RepeaterNode(int repeatCount)
    {
        m_repeatCount = 0;
        m_maxRepeatCount = repeatCount;
    }

    void OnEnter() override
    {
        m_state = NodeState::Running;
        m_repeatCount = 0;
    }

    NodeState OnUpdate() override
    {
        auto state = m_children[0]->Execute();

        // 実行中ならそのまま返す。
        if (state == NodeState::Running)
        {
            return state;
        }

        // 処理が終了した場合だと、リピート回数を超えたら終了する。
        ++m_repeatCount;
        if (m_repeatCount == m_maxRepeatCount)
        {
            return NodeState::Success;
        }

        // リピート中なら子ノードをリセットして更新準備をする。
        m_children[0]->Reset();

        return NodeState::Running;
    }

    void OnExit() override
    {

    }

private:

    // 現在のリピート回数を保持する。
    int m_repeatCount;

    // 最大リピート回数を保持する。
    int m_maxRepeatCount;
};
```

サンプルでは使用していませんが、Repeaterノードです。

子ノードを指定回数実行します。

指定回数分繰り返すと「成功」として終了します。

指定回数未満だと、

子ノードをリセットし「実行中」を返します。

簡単なビヘイビアツリーを実装する！ 14（実装例）

#完成イメージ

```cpp
#pragma once
#include <functional>
#include "BaseNode.h"

class ParallelNode : public InternalNode
{
public:

    void OnEnter() override
    {
        m_state = NodeState::Running;
    }

    NodeState OnUpdate() override
    {
        bool allSucceeded = true;
        for (auto child : m_children)
        {
            // 既に終了しているノードはスキップ
            if (child->IsFinished())
            {
                continue;
            }

            auto state = child->Execute();

            // 子ノードが一つでも失敗すれば終了。
            if (state == NodeState::Failure)
            {
                return NodeState::Failure;
            }

            // まだ実行中のものがあれば終了しない。
            if (state == NodeState::Running)
            {
                allSucceeded = false;
            }
        }

        // 全ての子ノードが成功すれば終了。
        if (allSucceeded)
        {
            return NodeState::Success;
        }

        return NodeState::Running;
    }

    void OnExit()
    {

    }
};
```

サンプルでは使用していませんが、Parallelノードです。

全ての子ノードを同時に実行し、

子ノードが一つでも「失敗」すれば「失敗」として終了する。

全ての子ノードが「成功」した時に「成功」として終了する。

それ以外の場合は「実行中」となります。

簡単なビヘイビアツリーを実装してみて１

簡単なサンプルでしたが、ビヘイビアツリーの実装ができたと思います。

前回までのステートマシンと比べると、
今回はコンソールベースのためデメリットの方が大きいかと思います。

しかし専用Editorを作成したり、Excelで作ったデータからツリー構築する等していけば、
非常に使いやすくなり当初の問題も解決出来ると思います。

もちろん、それでもステートマシンの方が有利な場面もあるので、
いいとこどり出来たらもっと良いと思います。

次のページへ

簡単なビヘイビアツリーを実装してみて２

このサンプルを更に実践的にしていく場合は、

一度終了したConditionノードやDecoratorノードを
再評価する仕組みが必要になってくると思います。

再評価が出来ないと、
「近くにいたら攻撃する」という行動に「失敗」した後、
時間が経ち敵が近づいた時に「失敗」として終了しているので攻撃してくれない。
といった問題が出てくると思います。

他にもいくつかありますがまずは、再評価システムは導入したほうが良いと思います。

AI（キャラクター制御）のまとめ

今回は簡単な例でしたが、

FSM、HFSMのステートマシン、そしてビヘイビアツリーを実装してみました。

結構難しい内容になったかもしれませんが、

AI制御のための管理方法の解説のみだったので、実際に環境マップやデータテーブルから

行動していくような振る舞いに関する内容はもっと深いです。（本当に難しいです）

ですが、管理方法さえしっかりしていればバグも少なく出来るので色んな挑戦が出来ると思います。

また今回のサンプルを、より実践的で凝ったものにしようと思うとまだまだ機能が足りません。

特に非同期処理への対応などが考えられます。（同時に更新が呼ばれた際への対処）

自分なりに実装、拡張していってみてください。