イベントシステム（メッセージング）

ノーマル（最初から）

イベントてっ何？から
簡単な実装までの話

ハード（応用）

より実践的な実装へ
もう一つのイベント管理

ベリーハード（より発展的）

更に進化する
タイムアウト処理
別スレッドでの話

イベントシステムって何？

イベントとは？

「敵にぶつかった」「敵をやっつけた」「ゲームクリアした」 等の、

ゲーム中に発生する事象やメッセージのことです。

使うとどんなメリットがあるの？

- メッセージという単位でアクセスするため、依存関係を整理出来る。
- 依存関係を整理出来ているから、変更に強いコードに出来る。
- イベントの送信側と受信側で分かれているため、コードが追いやすい。

つまり最強！

まずは、イベントを使わないコードを見てみる

#仮のコード例

```
void MainLoop()
{
    if (ゲームクリアした？)
    {
        // 何らかの処理
    }

    if (ゲームオーバーになった？)
    {
        if (プレイヤーは落下で死んだ？)
        {
            // 何らかの処理
        }
        // 何らかの処理
    }
}
```

このようにイベントを使わないで普通に処理を書くと、
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。

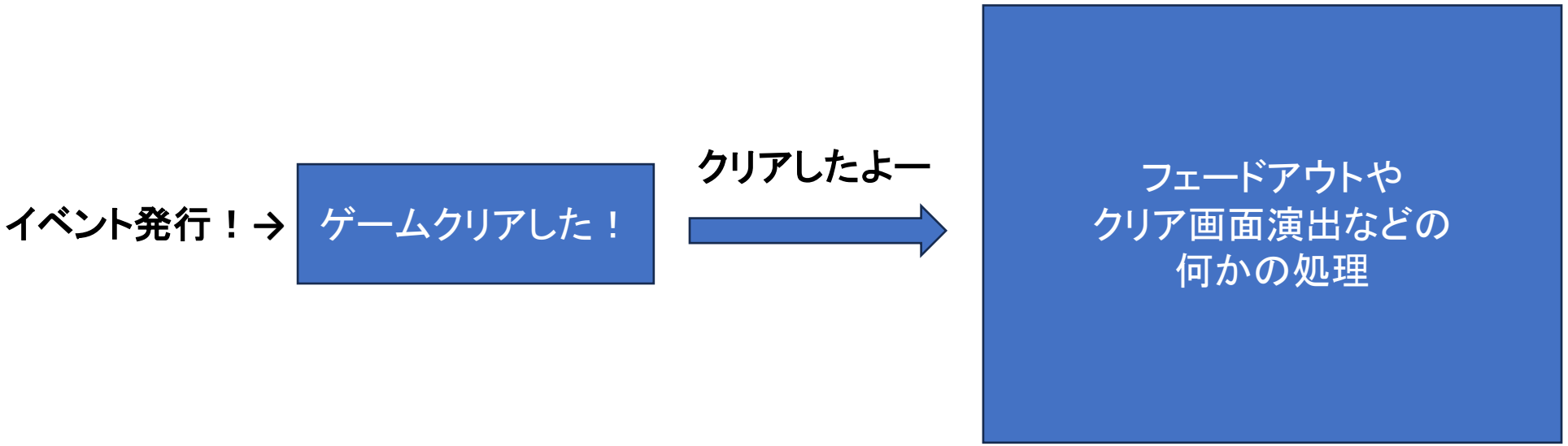
#プラスα

「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、イベント使うとどんな感じになるの？（解説）

毎回ゲームクリア、ゲームオーバーに関した判定を毎回しなくちゃいけないし、

フラグ確認のためだけに相互参照になっていたり、コードは見ずらいしで全然いいことありません。



非常にざっくりした内容ですが、

ゲームクリアした瞬間に登録された各処理に対して、ゲームクリアを通知しています。

「関数と何が違うんだ？」と思うかもしれませんが、通知した先が何かの処理であることが大事です。

じゃあ、イベント使うとどんな感じになるの？（実装例）

#簡単な実装例

```cpp
#include <iostream>
#include <functional>

struct GameClear
{
    std::function<void()> GameClearEvent;
};

void OnGameClear()
{
    printf("ゲームクリア！\n");
}

int main()
{
    GameClear gameClear;

    // ゲームクリア時の処理を追加
    gameClear.GameClearEvent = OnGameClear;

    // ゲームループ
    while (true)
    {
        // if（ゲームクリアした!）
        // {
                gameClear.GameClearEvent();
        // }
    }
}
```

仮コードではゲームクリア時のイベントの例です。

ゲームクリア時のイベントに処理を追加しています。

簡単な例ではありますが、

ゲームクリア時の処理が固定されていない

（セットしている関数を変更出来る）

状態であることが後々非常に便利になります。

#プラスα

今回は関数での実装ですが、クラス間で使用することで、

参照関係が整理でき、コードが追いやすくなります。

UEなどでも似たようなでイベント処理が実装されています。

イベントをクラスとして作ってみる（実装例）

#簡単な実装例

```cpp
#include <iostream>
#include <functional>

class GameClearEvent
{
public:

    void AddFunc(std::function<void()> func)
    {
        Events.push_back(func);
    }

    void Invoke()
    {
        for (auto Event : Events)
        {
            Event();
        }
    }

private:

    std::vector<std::function<void()>> Events;
};

void OnGameClear()
{
    printf("GameClear");
}

int main()
{
    GameClearEvent gameClearEvent;

    // ゲームクリア時の処理追加
    gameClearEvent.AddFunc(OnGameClear);

    // ゲームループ
    while (true)
    {
        // ゲームクリア時に呼ぶ
        gameClearEvent.Invoke();
    }
}
```

先ほどの実装例だと関数を一つしか
登録できないため、非常に使いにくいです。
そこでいくつか関数を登録出来る様にしました。

実行中にイベントの登録解除出来る関数用意するなど
様々な拡張をしていくとより使いやすくなっていきます。

#プラスα

ゲームでは様々なイベントが必要なため、
「引き数の数を持たせたい！」「継承とかもめんどくさい！」
と色んな要望が出てくると思います。
今回の場合は、テンプレートクラスが有効かもしれません。

↑後で解説します。

イベントシステム（応用）

1. 先ほどまでの実装例をもっと使いやすくしていく。
2. イベントシステムのもう一つの実装例を紹介します。

最初の実装例をもっと使いやすくしてみる！1（実装例）

#テストで作ったコード

```cpp
#include <iostream>
#include <functional>

template<class T>
class Event
{
public:

    void AddFunc(std::function<void(T)> func)
    {
        Events.push_back(func);
    }

    void Invoke(T data)
    {
        for (auto Event : Events)
        {
            Event(data);
        }
    }

private:

    std::vector<std::function<void(T)>> Events;
};

template<>
class Event<void>
{
public:

    void AddFunc(std::function<void()> func)
    {
        Events.push_back(func);
    }

    void Invoke()
    {
        for (auto Event : Events)
        {
            Event();
        }
    }

private:

    std::vector<std::function<void()>> Events;
};
```

テンプレートクラスとして定義することで、

使用者側で引き数を追加できるようにしています。

テンプレートTの指定型がvoidの場合は、

Invoke関数の引き数がいらないのでコード下部分で、

特殊化して引き数なしバージョンを作成しています。

#プラスα

最初の実装例を元に、継承を使った場合でも、

テストコード同様に引き数を増やすことは可能です。

そちらの方法が良い場合もあるので、

使用用途に合わせて実装を考えてみてください。

最初の実装例をもっと使いやすくしてみる！2（実装例）

#テストで作ったコード

```
227
228 int main()
229 {
230     // 引き数なし
231     Event<void> event1;
232     event1.AddFunc([]() { printf("event1"); });
233     event1.Invoke();
234
235     // 引き数あり
236     Event<int> event2;
237     event2.AddFunc([](int id) { printf("id = %d", id); });
238     event2.Invoke(10);
239 }
```

#テストの実行結果

```
event1
id = 10

C:\Users\hinan\OneDrive\デスクトップ
ました。
デバッグが停止したときに自動的にコン
```

実際に使ってみたサンプルです。

便利ではあるが、引き数1個じゃ不便ですよね？

次は引き数の数を増やしていきます。

可変長引き数にして引き数の数を増やす！（実装例）

#テストで作ったコード

```cpp
#include <iostream>
#include <functional>

template<class... Args>
class Action
{
    using Func = std::function<void(Args...)>;
public:

    void AddFunc(Func&& func)
    {
        functions.push_back(func);
    }

    void Invoke(Args... args)
    {
        for (auto func : functions)
        {
            func(args...);
        }
    }

private:

    std::vector<Func> functions;
};

template<>
class Action<void>
{
    using Func = std::function<void()>;
public:

    void AddFunc(Func&& func)
    {
        functions.push_back(func);
    }

    void Invoke()
    {
        for (auto func : functions)
        {
            func();
        }
    }

private:

    std::vector<Func> functions;
};
```

可変長テンプレートクラスとして定義することで、

使用者側で引き数を任意の数追加できます。

そのため、先ほどのものより更に使用しやすいです。

※これが最初から普通に組める人なら、このイベントシステムの資料は参考にならないかも

使用しているコード

```cpp
int main()
{
    // test 1
    {
        Action<void> action;
        action.AddFunc([]() { std::cout << "call method\n"; });
        action.Invoke();
    }

    // test 2
    {
        Action<int> action;
        action.AddFunc([](int id) { printf("hp = %d\n", id); });
        action.Invoke(10);
    }

    // test 3
    {
        Action<int, int> action;
        action.AddFunc([](int id, int hp) { printf("hp = %d hp = %d\n", id, hp); });
        action.Invoke(10, 1000);
    }

    return 0;
}
```

イベントシステムには二つの実装方法が考えられる

○イベント専用のクラスを作成する方法（先ほどの実装ようなもの）

メリット

- 依存関係が整理されているためコードが追いやすい。
- 呼び出した瞬間にイベントが実行されることが保証されている。

デメリット

- 遅延実行や、処理落ち時のタイムアウト処理などの実装が難しい。

○イベントを一元管理するマネージャークラスを作成する方法

メリット

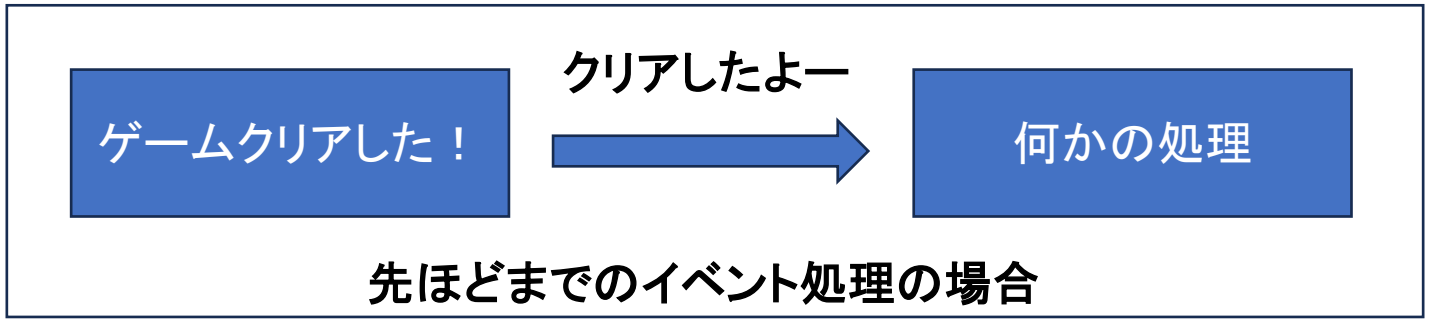
- 遅延実行やタイムアウト処理などの拡張機能を作成しやすい。
- シーンを超えたイベント処理などに有効（特にUnity等）

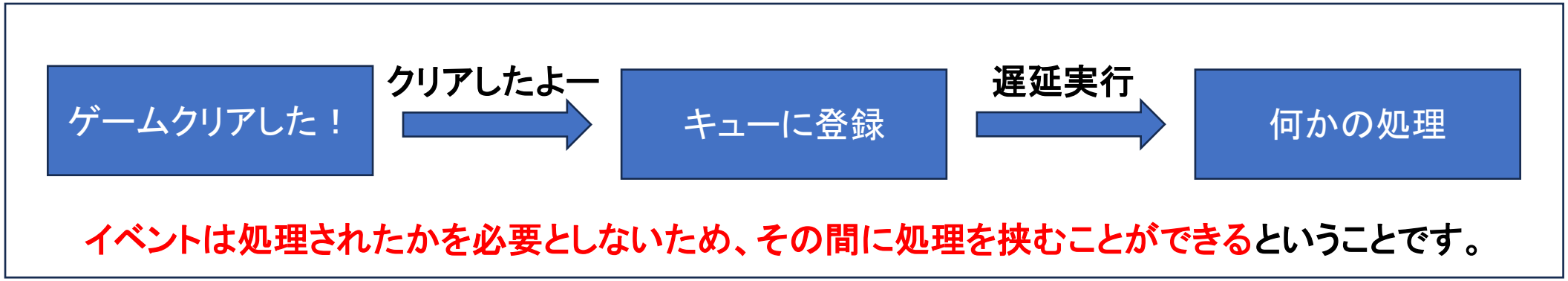
デメリット

- 使い方によってマネージャークラスへの依存が増えるだけになる。

マネージャークラスを使用する方法を作る（解説）

イベントを管理するマネージャークラスを介してイベントの発行を行うことで、
様々なメリットがあるとざっくりですが、先ほどページでは書いてました。
ここでは、ざっくり具体的に解説します。





↑「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。

マネージャークラスを使用する方法を作る1（実装例）

#テストで作ったコード

```cpp
#include <iostream>
#include <functional>
#include <any>
#include <set>
#include <map>

class EventListener
{
public:

    void Action(std::any data)
    {
        func(data);
    }

    void SetFunction(int InEventId, std::function<void(std::any)> InFunc)
    {
        eventId = InEventId;
        func = InFunc;
    }

    void RegisterToEventManager()
    {
        EventManager::Get().AddEventListener(this, eventId);
    }

    void UnRegisterFromEventManager()
    {
        EventManager::Get().RemoveEventListener(this, eventId);
    }

private:

    int eventId;

    std::function<void(std::any)> func;
};
```

簡単なものですがこのクラスはAction関数を介して、
登録したイベント処理を実行することができます。

今回は何かしらの引き数が一つ欲しかったので、
anyクラスを引き数として取得できるようにしています。

マネージャークラスを使用する方法を作る2（実装例）

#テストで作ったコード

```
69  class EventManager
70  {
71  public:
72
73      static EventManager& Get() noexcept
74      {
75          static EventManager instance;
76          return instance;
77      }
78
79      void Invoke(int id, std::any data)
80      {
81          auto hasEvent = EventListenerMap.contains(id);
82          if (hasEvent == false)
83          {
84              return;
85          }
86
87          for (auto eventListener : EventListenerMap[id])
88          {
89              eventListener->Action(data);
90          }
91      }
92
93      // 追加、解除処理を記述しているものとしています。
94      void AddEventListener(EventListener* InEventListener, int InEventId){}
95      void RemoveEventListener(EventListener* InEventListener, int InEventId){}
96
97  private:
98
99      std::map<int, std::set<EventListener*>> EventListenerMap;
100 };
```

こちらも内容は簡単なものですが、このクラスは

イベントの登録、解除や、実行を行います。

特に全てのイベント実行処理を行っている点がミソです。

#プラスα

何故かというと、

今回はInvoke関数内でイベント処理まで行っていますが、

イベントを処理せずにキュー等に貯めておくことで、

一括でイベント処理を行うことができるようになるので、

タイムアウト処理追加や、別スレッドでの実行にするなど、

最初の実装例では難しかった機能拡張などが

簡単に実装出来るようになります。

↑後で解説します。

イベントシステム（より発展的）

1. 先ほど話に出ていたタイムアウト処理の追加。
2. 別スレッドを使ったイベント処理を解説します。

タイムアウト処理を追加する前の準備（実装例）

#テストで作ったコード

```
        }

        for (auto eventListener : EventListenerMa
        {
            eventListener->Action(data);
        }
    }

    // 追加、解除処理を記述しています。
    void AddEventListener(EventListener* InEventListener, int InEventId){}
    void RemoveEventListener(EventListener* InEventListener, int InEventId){}

private:

    struct EventData
    {
        int id;
        std::any data;
    };

    // タイムアウトの時間計測用
    Stopwatch Stopwatch;

    // * 現在使用中キュー番号
    uint32_t NumActiveQueue = 0;

    // * 今は重要ではないですが、次の布石のため２つキューを用意
    std::array<std::list<EventData>, 2> EventQueues;

    std::map<int, std::set<EventListener*>> EventListenerMap;
};
```

タイムアウト処理を作っていく前に、

必要な変数を容易しておきます。

EventDataではInvoke関数で

引き数として渡されたデータを保持するためのものです。

EventQueueを二つ用意している理由は、

イベントがイベントを呼び続けその結果、イベントが

帰ってこなくなる問題があるためその対処として必要です。

そのため、使用中のキュー番号も保持しています。

タイムアウト処理を追加する前の準備（実装例）

#テストで作ったコード

```cpp
enum TimePrecision
{
    Second,
    Milli,
};

class Stopwatch
{
public:

    void Start()
    {
        m_startTime = std::chrono::high_resolution_clock::now();
    }

    void Stop()
    {
        m_stopTime = std::chrono::high_resolution_clock::now();
    }

    double GetRap(TimePrecision precision = TimePrecision::Second)
    {
        Stop();

        // rap 計測
        std::chrono::duration<double, std::milli> ms = m_stopTime - m_startTime;

        switch (precision)
        {
        case Second: return (ms.count() / 1000.0);
        case Milli:  return ms.count();
        default: break;
        }
    }

private:

    // * 最新フレーム更新時間
    std::chrono::high_resolution_clock::time_point m_startTime;
    std::chrono::high_resolution_clock::time_point m_stopTime;
};
```

解説は最小減にしますが、タイムアウトで使用するため、

経過時間をラップとして取得出来るようになっています。

実際にマネージャーにタイムアウト処理を追加1（実装例）

#テストで作ったコード

```
71          return instance;
72      }
73
74      void Invoke(int id, std::any data)
75      {
76          auto hasEvent = EventListenerMap.contains(id);
77          if (hasEvent == false)
78          {
79              return;
80          }
81
82          EventQueues[NumActiveQueue].push_back(EventData(id, data));
83      }
```

Invoke関数内を変更して、

先ほどまでイベントを実行していたものに代わり、

イベントキューに発行されたデータを登録しています。

そして次に、

イベントの実行を行うための更新関数を用意します。

実際にマネージャーにタイムアウト処理を追加2（実装例）

#テストで作ったコード

```
71          return instance;
72      }
73
74      void Invoke(int id, std::any data)
75      {
76          auto hasEvent = EventListenerMap.contains(id);
77          if (hasEvent == false)
78          {
79              return;
80          }
81
82          EventQueues[NumActiveQueue].push_back(EventData(id, data));
83      }
84
85      void Tick()
86      {
87          // キューの入れ替え
88          const auto numQueue = NumActiveQueue;
89          (++NumActiveQueue) %= EventQueues.max_size();
90
91          while (EventQueues[numQueue].size() != 0)
92          {
93              // 先頭から登録されたイベントデータを取得する。
94              auto event = EventQueues[numQueue].front();
95              EventQueues[numQueue].pop_front();
96
97              // 指定されたイベントを実行する。
98              const auto& listeners = EventListenerMap[event.id];
99              for (auto listener : listeners)
100             {
101                 listener->Action(event.data);
102             }
103
104             // タイムアウトしているかをチェック
105             // 今回は16ミリ（60Fps）経つまではイベントを処理する。
106             constexpr double maxTime = 16.0;
107             if (maxTime <= Stopwatch.GetRap(Milli))
108             {
109                 break;
110             }
111         }
112
113         // タイムアウト時に残っているタスクを次のキューに追加しておく。
114         EventQueues[NumActiveQueue].merge(EventQueues[numQueue]);
115     }
116
117     // 追加、解除処理を記述しているものとしています。
118     void AddEventListener(EventListener* InEventListener, int InEventId){}
119     void RemoveEventListener(EventListener* InEventListener, int InEventId){}
```

イベントの実行を行うための更新関数として、

Tick関数を作成しています。

具体的に更新関数では、

イベントキューを入れ替えしてイベント実行中でも、

新しくイベントの追加を行えるようにしています。

その後は、

タイムアウトするか、全てのイベントを処理するまで、

キューに登録された順序でイベントを処理していきます。

※ストップウォッチのStart関数はメインループの先頭で呼び出されているものとします。

別スレッドを使ったイベントループの話（解説）

#テストで作ったコード

```
156  void EventMain()
157  {
158      // イベントループ
159      while (true)
160      {
161          // イベントが発行されていれば処理する
162          if (EventManager::Get().HasEventQueue())
163          {
164              // 発行されたイベントを処理する。
165              EventManager::Get().Tick();
166          }
167      }
168  }
```

イベントループとは、

キューにイベントが発行されれば更新を行うものです。

（普通、、、だよね、、、）

使い道としてはネットワークの対戦ゲームなどで、

待機はしたくないけど、相手からイベントはほしい！とか。

#プラスα

自作のエンジンを開発している人も、EditorとRuntimeを

TCP通信を使って分けて作成している場合などに有効です。

イベントシステムのまとめ

今回は簡単な例でしたが、
C#にある、Actionクラス風のイベントクラスと、
マネージャークラスを使用したイベントクラスを解説しました。

メリットデメリット両方あるので、
どっちか片方だけじゃなくて両方使うという選択肢もあり、
というか大体の場合はそうなると思います。

これが使えるようになると格段に開発効率が上がるので、ぜひ実践してみてください。