DIについて考える

ノーマル（最初から）

DIって何？から
簡単な実装までの話

ハード（応用）

更に進化する
DIコンテナ

ベリーハード（より発展的）

間違いやすい
サービスロケータの話

DI（依存性注入）ってなに？

DI（依存性注入）とは？

デザインパターンの一つで、

オブジェクト同士の依存関係をコードに直接書かず、

別の場所からいくつかの方法で与える手法のこと。

使うとどんなメリットがあるの？

- あるクラスAがあるクラスBの実装に依存しなくなる。

  つまり、クラスAが持つクラスBを任意に指定、変更できるようになる。

- クラスAだけを対象とした単体テストが可能になる。

まずは、DIを使わないコードを見てみる

#仮のコード例

```
11  class ClassA
12  {
13  public:
14
15      ClassA()
16      {
17          m_classB = new ClassB();
18      }
19
20  private:
21
22      ClassB* m_classB;
23  };
```

左のように、

クラスAが持つクラスBを直接保持する形にすると、

クラスAがクラスBの実装に依存してしまうため、

Bが修正された際にクラスAにも修正が発生してしまう。

そのためクラスBの柔軟性がなくなってしまう。

#プラスα

もちろんこの他にも、

クラスBがなければクラスAをテスト出来ない等の問題がある

じゃあ、DIを使うとどんな感じになるの？（実装例）

#完成イメージ

```
11  class ClassA
12  {
13  public:
14
15      ClassA(IClassB* classB)
16      {
17          m_classB = classB;
18      }
19
20  private:
21
22      IClassB* m_classB;
23  };
24
25  int main()
26  {
27      auto classA = ClassA(new ClassB);
28  }
```

仮コードでは、

クラスBのインターフェイスであるIClassBを、

クラスAの初期化時に外部から与えるようにしている。

そのため、

クラスAは渡されるIClassBを継承する全てのクラスに

対応することになり、変更が容易になる。

#プラスα

クラスAの使用者側（今回はMain）がカスタマイズ出来るため、

テスト用のクラスBを渡すことで単体テストが可能になる。

DIを使ってみて

先程の仮コードでDIを使った時のイメージはつかめたと思います。

今回使用したのはコンストラクタインジェクションと呼ばれる方法です。
その他にも、関数からクラスBを注入する方法などいくつかあるので、
コードに合った方法を使用してください。

しかし、
今回は簡単にクラスAの引数が一つだけでしたが、
コードによっては3っつ、4っつと呼び出し元の管理が大変になっていきます。
次はそんな場合の解決策を解説していきたいと思います。

DI（応用）

1. DIコンテナについて解説
2. 簡単にDIコンテナを作ってみる？

DIコンテナについて

引数が増えていくことで呼び出し元の管理が困難になっていく問題については、

オブジェクトの生成方法の設定を保持した管理クラスを作成すれば解決できます。

具体的には、

呼び出し元で行う必要があった引数の指定をアプリケーション起動時にコンテナへ登録しておき、

呼び出し元ではクラスAを生成するということだけを指定することで、

コンテナが保持する設定からクラスAのオブジェクトを生成、取得するという方法です。

「何を言っているんだ？」となると思うので早速実装を見てみましょう。

最低限のDIコンテナを作る１（実装例）

#完成イメージ

```
18  class ClassA
19  {
20  public:
21
22      ClassA(IClassB* classB)
23      {
24          m_classB = classB;
25      }
26
27      ~ClassA()
28      {
29          delete m_classB;
30          m_classB = nullptr;
31      }
32
33  private:
34
35      IClassB* m_classB;
36  };
37
38  int main()
39  {
40      auto container = DIContainer();
41
42      // ClassB、ClassA の生成時の設定を登録
43      container.Register<IClassB, ClassB>();
44      container.Register<ClassA>([](auto c)
45      {
46          return new ClassA(c.Resolve<IClassB>());
47      });
48
49      // ClassA を生成（生成時に引数にはClassBが代入される）
50      auto classA = container.Resolve<ClassA>();
51
52      if (classA != nullptr)
53      {
54          delete classA;
55          classA = nullptr;
56      }
57  }
```

最初にIClassBが指定された時、

DIコンテナがClassBを生成し、返すように設定しています。

そしてその下で、

ClassAを指定された際の設定をしています。

（ここではIClassB（つまりClassB）を引き数にしています）

その後、実際にClassAをコンテナから生成しています。

#プラスα

C++にはリフレクション機能がないためこのような実装ですが、

自前でリフレクションを作れば色んな方法が考えれます。

最低限のDIコンテナを作る２（実装例）

#完成イメージ

```cpp
#pragma once
#include <map>
#include <type_traits>
#include <concepts>
#include <functional>

class DIContainer
{
    using Provider = std::function<void*(DIContainer&)>;
    using TypeId = size_t;
public:

    // Keyの生成設定を登録する関数
    template<class Key, class T>
    void Register()
    {
        Register<Key>([](auto c) { return new T; });
    }

    // Keyの生成設定を登録する関数
    template<class Key>
    void Register(Provider func)
    {
        auto typeId = typeid(Key).hash_code();
        auto hasType = m_providerMap.contains(typeId);

        // 二重登録は不可
        if (hasType == true)
        {
            return;
        }

        m_providerMap[typeId] = func;
    }

    // Keyから実際にオブジェクトを取得する関数
    template<class Key>
    Key* Resolve()
    {
        auto typeId = typeid(Key).hash_code();
        auto hasType = m_providerMap.contains(typeId);

        if (hasType == false)
        {
            return nullptr;
        }

        return static_cast<Key*>(m_providerMap[typeId](*this));
    }

private:

    // 生成クラスIdとオブジェクトの生成関数へのマップ
    std::map<TypeId, Provider> m_providerMap;
};
```

メインとなるDIコンテナクラスです。

Register関数は2つありますが、

どちらもKeyに対応した生成関数の登録を行っています。

Resolve関数は設定された生成関数を使用して、

実際にオブジェクトを生成し、返しています。

#プラスα

今回はサンプルのため、

KeyとTの型比較などのチェックをしてませんが、

もし実践に使うならチェック等、導入をすることをお勧めします。

シンプルなDIコンテナを作ってみて

非常に簡単な実装でしたが、

DIコンテナを作成出来たのではないでしょうか？

今回のサンプルではあまりありがたみが感じませんでしたが、

プロジェクトが大きくなったり、引き数の数が増えていくと大変嬉しいことがあります。

しかしDIコンテナを理解せずに使ってしまうとアンチパターンという、

良くない使い方をしてしまっている可能性があるので次はそちらの解説をします。

#プラスα

途中解説しましたが、

自前でリフレクションを作成することでもっと便利になりますので挑戦する方は頑張ってください。

DI（より発展的）

1. 間違いやすいサービスロケータ
2. なぜ間違うと良くないのか？解説

サービスロケータとは何か？（実装例）

#完成イメージ

```
18  class ClassA
19  {
20  public:
21
22      ClassA(DIContainer* container)
23      {
24          m_classB = container->Resolve<IClassB>();
25      }
26
27      ~ClassA()
28      {
29          delete m_classB;
30          m_classB = nullptr;
31      }
32
33  private:
34
35      IClassB* m_classB;
36  };
```

先ほどの実装からClassAだけ変更しました。

先ほどまではIClassBを引き数としていましたが、

今回はDIコンテナを引き数にしています。

そしてこれがサービスロケータであり、

良くない方法（アンチパターン）と言われる実装です。

サービスロケータは何故良くない？（解説）

察しの良い方ならお気づきかもしれませんが、

分からない方は、DIについてもう一度考えれば分かると思います。

そもそもDIとは、

他のクラスへの依存を減らすため、コードの柔軟性を上げるために使用しました。

そのため先程のように、

DIコンテナを引き数としてしまうことは単純にコンテナへの依存が増えてしまいます。

大事なので伝えておきますが、今回の目的としては使えないだけなので、

「サービスロケータがだめだ」と言っているわけではありません。（人によりますが）

DIのまとめ

今回は簡単な例でしたが、

DI、DIコンテナ、サービスロケータの三つについて解説しました。

DIとDIコンテナが別物ということをまずは理解して頂ければと思います。

もし実践で使う場合は、「何のために？」ということを意識すれば、

サービスロケータと間違うということは起こらないと思いますので、

「ただDIコンテナを作る」という目的なら使わない方が良いでしょう。

少しでも、皆さんの開発のためになれれば幸いです。