


Proxy ～具体的な実装を隠す身代わり ☐ ☐

 絶版に伴い、校正前の原稿テキストを公開したものです。基本的に原稿そのままをHTML形式に変換したものですので、誤字/脱字、説明不足の箇所もあるかも知れませんがご了承ください。初出:「PHPによるデザインパターン入門」(下岡秀幸/畑勝也/道端良 著, 秀和システム, ISBN4-7980-1516-4, 2006年11月23日発売)

GoF本における分類

構造+オブジェクト

はじめに

ここではProxyパターンについて説明します。

Windowsには「ショートカット」という機能があります。たとえばブラウザのショートカットをデスクトップ上などに登録しておき、実行したい時にそのショートカットをダブルクリックすることで、ショートカットの実体であるブラウザが実行されます。他のアプリケーションやテキストファイルでも同様の動作をおこないます。このように「実際は本物の身代わりだが、本物にアクセスしているかのように振る舞うもの」、それがプロキシです。

Proxyパターンもこれと同様で、身代わりとなるオブジェクトを通じて間接的に目的のオブジェクトにアクセスさせるためのパターンです。

たとえば

データベースを使ったシステムを考えてみましょう。

データベースを使ったシステムには、当然ですが何らのデータベースが必要になりますね。しかし、データベースを使ったシステムを開発する場合、肝心のデータベースがないとプログラムを作成できない、もしくはテストできないといった状況になりがちです。

なぜなら、システムを成すプログラムコードにデータベースに関するコードが記述されている、つまりビジネスロジックを含むプログラムとデータベースが強く結びついてしまっているからです。「データベースを使ったシステム」なので当然といえば当然ですが、ビジネスロジックは本来データベースとは関係がありません。しかし、ビジネスロジックとデータベースに関連する処理をまとめて書いてしまうことで、データベースと強く結びつきすぎてしまうことが問題なのです。

Webシステムのような変化の激しいアプリケーションでは、変化への素早い対応が求められることが多く、ビジネス上の重要なポイントになっています。これを踏まえると、システムの設計・開発段階から何らか考慮しておくことがシステム構築の鍵を握っているとも言えます。

また、データベースに関連する処理は比較的成本のかかる処理となります。このため、更新されないデータをキャッシュしたり、データベースとの接続やデータ取得のタイミングの調整が必要になる場合があります。

これらの問題を回避するには、どうしたら良いでしょうか？

ここで、データベースに関連する部分と関連しない部分を分けて(本来ビジネスロジックに関係ないデータベースに関連する処理を切り離すパターンは**DAOパターン**とも呼ばれます)、その間に**クッション役のクラス**を用意してやります。そうすると、データベースに関連する部分と関連しない部分のやりとりが間接的になりますね。間接的になるということは、具体的なクラスに依存しないコードになるということです。

では、このクッションに色々な役割を持たせてやると、どうなるでしょうか・・・何となく想像できますか？

たとえば、このクッションがデータベースにアクセスしているかのような振る舞いをするとしたら？データベースに関連しない部分のコードはデータベースがなくても開発できるようになりますね(このようなオブジェクトを**モック(mock)オブジェクト**といいます)。

もう1つ。このクッションがデータベースから取得したデータをキャッシュする機能を持っていたら・・・もうお分かりですね。

このように、やりとりをするオブジェクトの間にクッション役を用意し、このクッションに色々な役割を持たせるパターン・・・これが**Proxyパターン**です。

Proxyパターンとは？

Proxyパターンの目的は、GoF本では次のように定義されています。

あるオブジェクトへのアクセスを制御するために、そのオブジェクトの代理、または入れ物を提供する。

Proxyパターンではオブジェクトの構造に注目しています。Proxyパターンでは、オブジェクトとその利用側との間に緩衝剤を用意

することで、お互いを分離したり付加的な機能を追加することを目的としています。

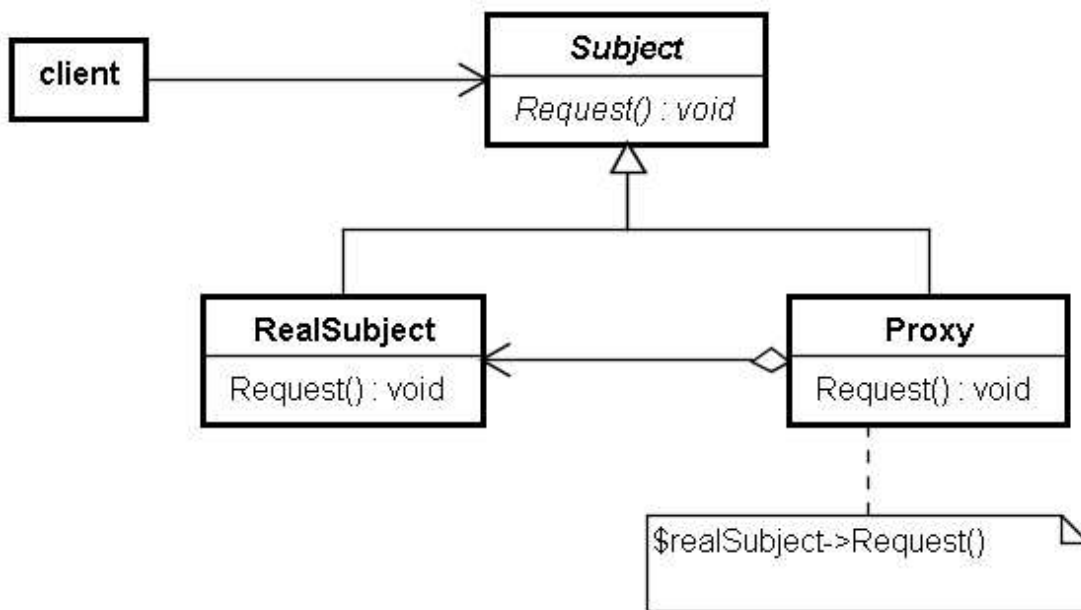
みなさんは「**プロキシサーバ**」という言葉聞いたことがあるかも知れません。プロキシサーバ(キャッシュサーバとも呼ばれます)は、外部ネットワークとのアクセス制御やコンテンツのキャッシング、さらにはインターネット上での匿名性を高めるために利用されるサーバです。

代理オブジェクトの利用側もプロキシサーバの場合と同様、**代理オブジェクトにアクセスしているのか目的のオブジェクトにアクセスしているのかを意識させないような構造**をしています。具体的には、代理オブジェクトと目的のオブジェクトが共通のインターフェースを実装する事で実現されています。

また、代理オブジェクトは目的のオブジェクトを内部に保持して、**具体的な処理を目的のオブジェクトに転送**します。

Proxyパターンの構造

Proxyパターンのクラス図と構成要素は、次のようになります。



Subjectクラス

RealSubjectクラスと**Proxy**クラスが提供する共通のAPIを定義します。

RealSubjectクラス

Subjectクラスのサブクラスで、**Subject**クラスで宣言されたメソッドを実装します。このクラスが実際の処理を提供します。

Proxyクラス

RealSubjectと同様、**Subject**クラスのサブクラスで、**Subject**クラスで宣言されたメソッドを実装します。ただし、具体的な処理は、内部に保持した**RealSubject**クラスのインスタンスに転送します。

また、このクラス内で、**RealSubject**クラスに対するアクセス制御や**RealSubject**クラスのインスタンスの生成タイミングなどを調整します。

Proxyパターンのメリット

Proxyパターンのメリットとしては、以下のものが挙げられます。

オブジェクトへのアクセスが間接的になる

Proxyパターンを適用すると、クライアントは目的のオブジェクトに**間接的にアクセス**するようになります。間接的になることで**Proxy**クラスにさまざまな機能を持たせることができ、**目的のオブジェクトへのアクセス手段を変化させる**ことができます。

Proxyパターンを適用する

Proxyパターンの適用例を見てみましょう。

ここでは、商品情報をデータベースから取得する簡単なアプリケーションを用意しました。Proxyパターンを適用し、同じAPIで実データを扱うクラスとダミーデータを扱うクラスを切り替えられるようにしています。

まずは、商品クラスから説明します。

Itemクラスは、純粋に商品の情報や注文の情報を内部に保持するだけの役割を担っており、コンストラクタに商品情報(商品IDと商品名)を受け取って、その値にアクセスするためのメソッドが用意されているだけです。特に難しいところはありませんね。

- Itemクラス (Item.class.php)

```
<?php
class Item {
    private $id;
    private $name;
    public function __construct($id, $name) {
        $this->id = $id;
        $this->name = $name;
    }
    public function getId() {
        return $this->id;
    }
    public function getName() {
        return $this->name;
    }
}
?>
```

次はSubjectクラスに相当するItemDaoインターフェースです。サブクラスの共通APIとなるfindByIdメソッドを宣言しています。このメソッドは商品IDを引数として受け取り、該当するItemオブジェクトを返すためのものです。

- ItemDaoインターフェース (ItemDao.class.php)

```
<?php
interface ItemDao {
    public function findById($item_id);
}
?>
```

また、クライアントはこのfindByIdメソッドを通じて、次に挙げるDbItemDaoクラスとMockItemDaoクラスにアクセスします。

DbItemDaoクラスはRealSubjectクラスに相当するクラスで、ItemDaoインターフェースを実装しています。このクラスは、実際のデータベースにアクセスして商品情報を取得します。

- DbItemDaoクラス (DbItemDao.class.php)

```
<?php
require_once 'ItemDao.class.php';
require_once 'Item.class.php';
?>
<?php
class DbItemDao implements ItemDao {
    public function findById($item_id) {
        $fp = fopen('item_data.txt', 'r');

        /**
         * ヘッダ行を抜く
         */
        $dummy = fgets($fp, 4096);

        $item = null;
        while ($buffer = fgets($fp, 4096)) {
            $id = trim(substr($buffer, 0, 10));
            $name = trim(substr($buffer, 10));

            if ($item_id === (int)$id) {
                $item = new Item($id, $name);
                break;
            }
        }
    }
}
```

```

        fclose($fp);
        return $item;
    }
}
?>

```

このアプリケーションでは、ファイルに格納された商品情報を扱うようになっています。データは固定長データとして用意しました。データフォーマットの詳細は次の表を参照してください。なお、項目名はデータファイル内の先頭行にもあります。

●商品情報(item_data.txt)

商品ID	商品名
1	限定Tシャツ
2	ぬいぐるみ
3	クッキーセット

項目	開始位置	終了位置
商品ID	1	10
商品名	11	-

一方のMockItemDaoクラスもItemDaoインターフェースを実装したRealSubjectクラスに相当するクラスです。しかし、findByIdメソッドで受け取った商品IDを使ってそのままItemオブジェクトを生成し、戻り値としています。つまり、どのような商品IDを渡しても、商品名が「ダミー商品」となるItemオブジェクトが生成されます。

●MockItemDaoクラス(MockItemDao.class.php)

```

<?php
require_once 'ItemDao.class.php';
require_once 'Item.class.php';
?>
<?php
class MockItemDao implements ItemDao {
    public function findById($item_id) {
        $item = new Item($item_id, 'ダミー商品');
        return $item;
    }
}
?>

```

続いて、ItemDaoProxyクラスを見てみましょう。ItemDaoProxyクラスもItemDaoインターフェースを実装したクラスですが、Proxyクラスに相当します。見てみると、実際の処理内容は、コンストラクタで受け取ったItemDao型のオブジェクトに委譲していることが分かります。

●ItemDaoProxyクラス(ItemDaoProxy.class.php)

```

<?php
class ItemDaoProxy {
    private $dao;
    private $cache;
    public function __construct(ItemDao $dao) {
        $this->dao = $dao;
        $this->cache = array();
    }
    public function findById($item_id) {
        if (array_key_exists($item_id, $this->cache)) {
            echo '<font color="#dd0000">Proxyで保持しているキャッシュからデータを返します</font><br>';
            return $this->cache[$item_id];
        }

        $this->cache[$item_id] = $this->dao->findById($item_id);
        return $this->cache[$item_id];
    }
}

```

?>

ItemDaoProxyクラスには、簡易的ですが、findByIdメソッドでは商品情報のキャッシュ機能が実装されています。また、キャッシュからデータが取得された場合、メッセージを表示するようになっています。これは実際に動作を確認してみてくださいね。

その他、Proxyクラスに相当するクラスには、データベースとの接続・初期化処理やデータの事前処理、後処理などを記述することができます。

最後にクライアント側コードの説明です。

このコードでは、受け取ったPOSTパラメータ「dao」の値によって、DbItemDaoクラスもしくはMockItemDaoクラスをインスタンス化し分けています。また、POSTパラメータ「proxy」の値によってItemDaoProxyクラスを使用するかどうかを切り替えます。使用する場合、生成したItemDaoオブジェクトをItemDaoProxyクラスのコンストラクタに渡しています。

●クライアント側コード(proxy_client.php)

```
<?php
if (isset($_POST['dao']) && isset($_POST['proxy'])) {
    $dao = $_POST['dao'];
    switch ($dao) {
        case 1:
            include_once 'MockItemDao.class.php';
            $dao = new MockItemDao();
            break;
        default:
            include_once 'DbItemDao.class.php';
            $dao = new DbItemDao();
            break;
    }

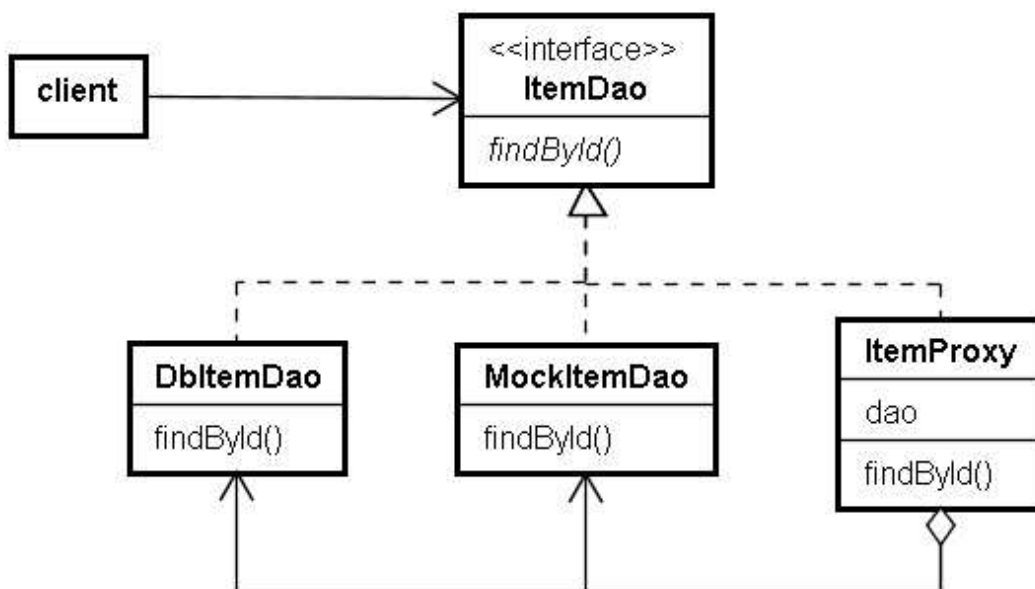
    $proxy = $_POST['proxy'];
    switch ($proxy) {
        case 1:
            include_once 'ItemDaoProxy.class.php';
            $dao = new ItemDaoProxy($dao);
            break;
    }

    for ($item_id = 1; $item_id <= 3; $item_id++) {
        $item = $dao->findById($item_id);
        echo 'ID=' . $item_id . 'の商品は「' . $item->getName() . '」です<br>';
    }

    /**
     * 再度データを取得
     */
    $item = $dao->findById(2);
    echo 'ID=' . $item_id . 'の商品は「' . $item->getName() . '」です<br>';
}
?>
<hr>
<form action="" method="post">
    <div>
        Daoの種類 :
        <input type="radio" name="dao" value="0" checked>DbItemDao
        <input type="radio" name="dao" value="1">MockItemDao
    </div>
    <div>
        Proxyの利用 :
        <input type="radio" name="proxy" value="0" checked>しない
        <input type="radio" name="proxy" value="1">する
    </div>
    <div>
        <input type="submit">
    </div>
</form>
```

それぞれのインスタンスを生成した後に商品情報を取得していますが、ここは共通のコードとなっていることと、生成されるインスタンスによって動作を切り替えることができることを確認してください。

また、Proxyパターンの適用したアプリケーションのクラス図は、次のようになります。



Proxyパターンのオブジェクト指向的要素

Proxyパターンは「**委譲**」を多用したパターンです。

ここまで見てきたように、Proxyパターンでは目的のオブジェクトを包み込むクラス(Proxyクラス)を用意し、クライアントはこのクラスで用意されたAPIを通じて目的のオブジェクトにアクセスします。また、Proxyクラスの内部では具体的な処理はおこなわず、包み込んだ目的のオブジェクトに最終的な処理を依頼しています。

また、包み込む側のクラスは**目的のオブジェクトと共通のAPI**を持ちます。つまり、クライアントはこのAPIを使ってコードを書いている限り、Proxyオブジェクトでも目的のオブジェクトでも気にせず利用できる、ということを意味します。このことを「**透過的である**」といいます。

関連するパターン

Adapterパターン

AdapterパターンはProxyパターンと同様、目的のオブジェクトを包み込むパターンですが、Adapterパターンは目的のオブジェクトが提供するAPIと異なるAPIを提供するためのパターンです。

Decoratorパターン

Decoratorパターンも共通インターフェースを持つクラスが内部に保持したインスタンスに具体的な処理を転送するパターンで、Proxyパターンとよく似ています。しかし、適用する目的が異なります。

Decoratorパターンは、あるオブジェクトに対して新しい機能を追加することを目的としています。Proxyパターンは、あるオブジェクトに対するアクセスを変更可能にするパターンです。

まとめ

ここでは目的のオブジェクトの代理オブジェクトを用意するProxyパターンについて見てきました。