Flyweight ～同じものは一度しか作らない

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＋クラス、オブジェクト

はじめに

ここではFlyweightパターンについて説明します。

flyweightと言えば、ボクシングの「フライ級」を連想する人も多いと思いますが、その通りです。Flyweightパターンは、フライ級のオブジェクトを効率よく共有するためのパターンなのです。オブジェクトを共有、つまり使いまわすことでメモリの節約を図ります。

たとえば

商品情報を格納するクラスを考えてみましょう。

商品情報には、商品番号や商品名などの「状況によって変わらない情報」と在庫数などの「状況によって変わる情報」があります。

ある商品を表すオブジェクトを生成するには、この商品クラスをインスタンス化しますが、「状況によって変わらない情報」を持つオブジェクトを毎回インスタンス化するのはもったいないですね。なぜなら、インスタンス化された商品オブジェクトは、それも全く同じ情報を持つものだからです。

一方で在庫数をこの商品クラスに持たせるのは得策ではありません。なぜなら、インスタンス化したタイミングで在庫数が変わっている可能性があるためです。さもないと、在庫が余りすぎたり足りなかったりしてしまうでしょう。

また、多くのシステムでは商品情報をデータベースに格納しています。つまり、商品クラスをインスタンス化する際にデータベースから商品情報を取得していることが多くなります。このため、毎回インスタンス化するコストが無視できない場合も出てきます。

こういった場合、どのような対処をおこなえばよいでしょうか？

一度インスタンス化した後にそのインスタンスを使いまわした方が良さそうです。そうすれば、インスタンス化のコストを大きく抑えることができます。また、インスタンス化に伴うメモリの消費量も抑えられると期待できます。

ここで説明するFlyweightパターンは、一度インスタンス化したオブジェクトを使い回し、生成されるオブジェクトの数やリソースの消費を抑えます。

Flyweightパターンとは？

Flyweight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。

> 多数の細かいオブジェクトを効率よくサポートするために共有を利用する。

Flyweightパターンは、オブジェクトの構造に注目したパターンです。Flyweightパターンでは、生成されるクラスとそのインスタンスを生成・管理するファクトリに分かれています。

クライアント側でそのクラスのインスタンスが必要になった場合、ファクトリに生成を依頼し、インスタンスを入手します。一方のファクトリ側は、生成したインスタンスを内部に保持します。再び生成の依頼を受けた場合、その保持したインスタンスを返します。つまり、一度生成したインスタンスをそのまま使いまわすわけです。

こうすることで、必要となったインスタンスだけ作成することになり、使用するメモリや生成時のコストを抑えることができます。

何か良いことずくめのようですが、Flyweightパターンを適用する場合に注意することがあります。それは、どのオブジェクトを共有して良いか、という点です。

「情報」には、環境や状況によって変化しないものと、そうでないものがあります。前者は「intrinsic」(「本質的な」の意)、後者は「extrinsic」(「非本質的な」の意)と呼ばれます。

環境や状況によって変化してしまう情報を、あちらこちらで共有してしまうと、他のすべての共有場所にその影響が出てしまうことになります。たとえば、誕生日は環境や状況によって変化しませんので、intrinsicな情報ということになります。しかし、年齢は時が経てば変わってしまいます。つまり、extrinsicな情報ということになります。

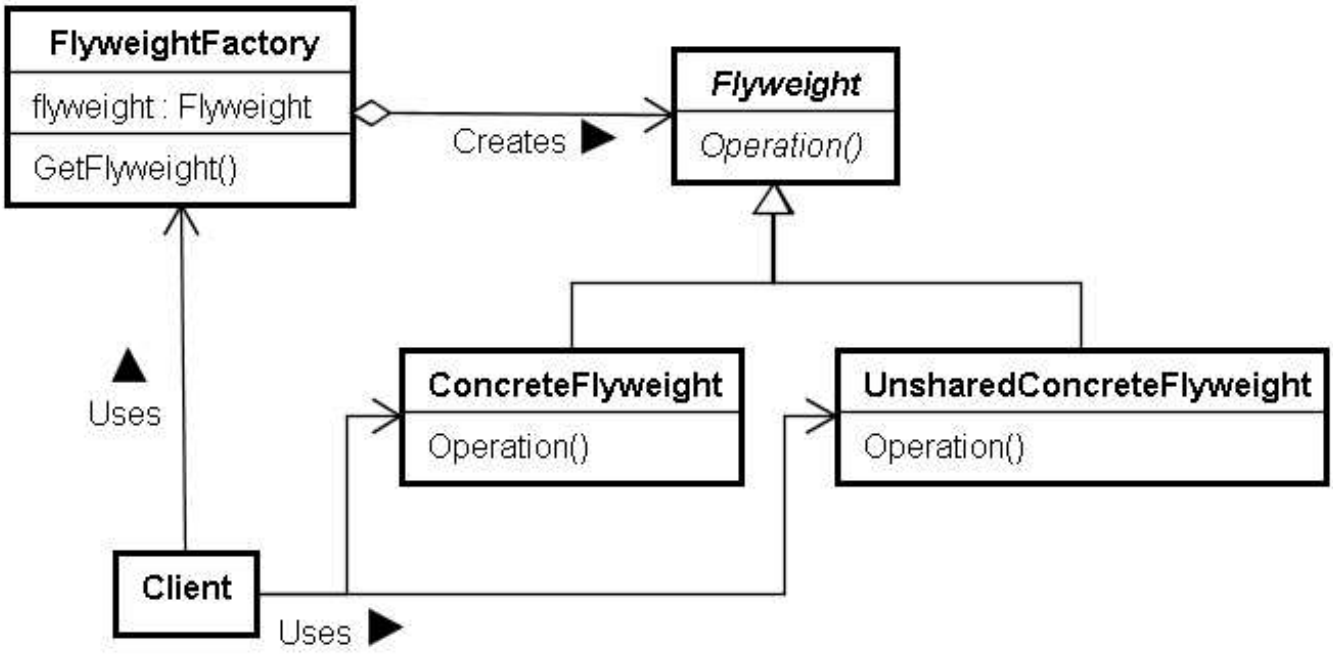
情報の種類	意味	共有
intrinsic	環境や状況によって変化しない	できる
extrinsic	環境や状況によって変化する	できない

Flyweightパターンで共有するオブジェクトは、intrinsicな情報を持つオブジェクトになります。

また、GoFパターンではありませんが、ObjectPoolパターンと呼ばれるパターンがあります。Flyweightパターンでオブジェクトの生成を担当するファクトリは、ObjectPoolパターンと良く似ています。ObjectPoolパターンも、オブジェクトを再利用するためのパターンです。

Flyweightパターンの構造

Flyweightパターンのクラス図と構成要素は、次のようになります。



Flyweightクラス

共有するオブジェクトの共通APIを定義します。

ConcreteFlyweightクラス

Flyweightクラスのサブクラスで、Flyweightクラスで定義されたAPIを実装するクラスです。このクラスのインスタンスが共有されますので、intrinsicな情報のみ保持するようにします。

UnsharedConcreteFlyweightクラス

Flyweightクラスのサブクラスで、Flyweightクラスで定義されたAPIを実装するクラスです。このクラスのインスタンスは共有されません。従って、extrinsicな情報を保持しても構いません。

FlyweightFactoryクラス

Flyweight型のインスタンスを生成・保持します。

Clientクラス

Flyweight型のオブジェクトへの参照を保持します。

Flyweightパターンのメリット

Flyweight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。

生成されるオブジェクトの数を抑える

Flyweightパターンでは、オブジェクトを共有するパターンです。一度生成したインスタンスを内部に保存しておき、再び生成する必要が生じた場合、その保持したインスタンスを返すようにしています。

このため、毎回newしてオブジェクトを生成するよりも、実際に生成されるオブジェクトの数を圧倒的に抑えることができます。

リソースの消費を抑える

冒頭では、使用メモリの節約について説明しました。Flyweightパターンでは、メモリ以外のリソースも節約することができます。たとえば、インスタンスを生成する時間がそうです。インスタンスを生成することは、非常に時間がかかる処理の一つです。Flyweightパターンでは一度生成したオブジェクトを使いまわすことで、使用メモリを節約すると共に、生成にかかる時間も節約します。これによって、プログラムのパフォーマンスを上げることができます。

Flyweightパターンの適用例

では、Flyweightパターンの適用例を見てみましょう。

ここでは、ファイルに格納された商品情報から商品オブジェクトを生成し、一覧表示するアプリケーションです。また、生成した同じ商品オブジェクトを比較し、同一のオブジェクトかどうかを確認しています。

まずは、商品情報を格納するItemクラスから見てみます。

このクラスはコンストラクタに渡された商品情報を内部に保持し、その情報にアクセスするためのメソッドを用意しただけのシンプルなものです。このクラスには特に難しいところはないでしょう。

このオブジェクトにはintrinsicな情報、つまり、環境や状況によって変化しない情報だけが格納されます。

●Itemクラス(Item.class.php)

```php
<?php
/**
 * FlyweightとConcreteFlyweightに相当する
 */
class Item {

    private $code;
    private $name;
    private $price;

    public function __construct($code, $name, $price) {
        $this->code = $code;
        $this->name = $name;
        $this->price = $price;
    }

    public function getCode() {
        return $this->code;
    }

    public function getName() {
        return $this->name;
    }

    public function getPrice() {
        return $this->price;
    }

}
?>
```

次は、商品オブジェクトを生成・管理するItemFactoryクラスです。このクラスは、Singletonパターンも適用されています。

●ItemFactoryクラス(ItemFactory.class.php)

```php
<?php
```

```
require_once 'Item.class.php';
?>
<?php
/**
 * FlyweightFactoryに相当する
 * また、Singletonパターンにもなっている
 *
 * なお、このサンプルではUnsharedConcreteFlyweightオブジェクトを
 * 返すメソッドは用意されていない
 */
class ItemFactory {
    private $pool;
    private static $instance = null;

    /**
     * コンストラクタ
     * このサンプルでは、インスタンス生成時に保持するオブジェクトを
     * すべて生成している
     */
    private function __construct($filename) {
        $this->buildPool($filename);
    }

    /**
     * Factoryのインスタンスを返す
     */
    public static function getInstance($filename) {
        if (is_null(self::$instance)) {
            self::$instance = new ItemFactory($filename);
        }
        return self::$instance;
    }

    /**
     * ConcreteFlyweightを返す
     */
    public function getItem($code) {
        if (array_key_exists($code, $this->pool)) {
            return $this->pool[$code];
        } else {
            return null;
        }
    }

    /**
     * データを読み込み、プールを初期化する
     */
    private function buildPool($filename) {
        $this->pool = array();

        $fp = fopen($filename, 'r');
        while ($buffer = fgets($fp, 4096)) {
            list($item_code, $item_name, $price) = split("¥t", $buffer);
            $this->pool[$item_code] = new Item($item_code, $item_name, $price);
        }
        fclose($fp);
    }

}
?>
```

このクラスで注目するのは、buildPoolメソッドとgetItemメソッドです。

buildPoolメソッドはプライベートメソッドとなっており、コンストラクタから呼び出されています。このメソッドでは、ファイルに保存された商品情報を読み込んで商品インスタンスを生成し、そのインスタンスを内部に保持しています。

一方のgetItemメソッドでは、指定された商品番号のインスタンスをそのまま返しています。

これにより、一度生成された商品インスタンスが使いまわされていることが分かると思います。

なお、商品データファイルは次のとおりで、タブ区切りテキストとなっています。

●商品データファイル(data.txt)

```
ABC0001 限定Tシャツ　3800
ABC0002 ぬいぐるみ　1500
ABC0003 クッキーセット　800
```

そして、クライアント側のコードです。

まずはItemFactoryクラスのインスタンスを取得し、商品番号を指定して商品オブジェクトを取得しています。

また、商品番号「ABC0001」については、再度インスタンスを取得し、1度目に取得したインスタンスと2度目に取得したインスタンスを比較しています。

●クライアント側コード(flyweight_client.class.php)

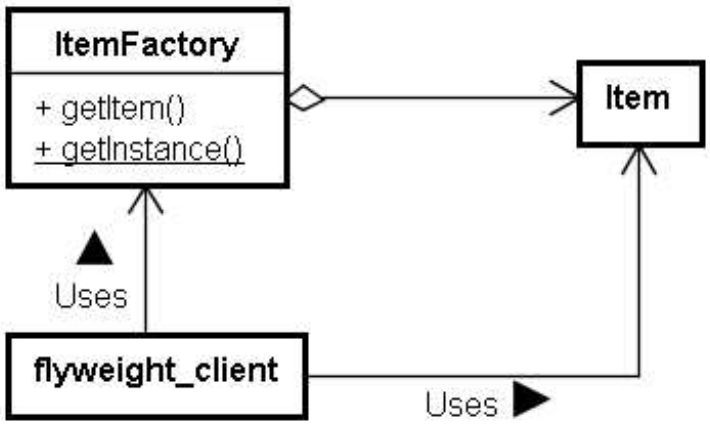
```php
<?php
require_once 'ItemFactory.class.php';
?>
<?php
function dumpData($data) {
    echo '<dl>';
    foreach ($data as $object) {
        echo '<dt>' . htmlspecialchars($object->getName(), ENT_QUOTES) . '</dt>';
        echo '<dd>商品番号：' . $object->getCode() . '</dd>';
        echo '<dd>¥¥' . number_format($object->getPrice()) . '-</dd>';
    }
    echo '</dl>';
}
?>
<?php
    $factory = ItemFactory::getInstance('data.txt');

    /**
     * データを取得する
     */
    $items = array();
    $items[] = $factory->getItem('ABC0001');
    $items[] = $factory->getItem('ABC0002');
    $items[] = $factory->getItem('ABC0003');

    dumpData($items);

    if ($items[0] === $factory->getItem('ABC0001')) {
        echo '同一のオブジェクトです';
    } else {
        echo '同一のオブジェクトではありません';
    }
?>
```

最後に、Flyweightパターンを適用したサンプルのクラス図を確認しておきましょう。



Flyweightパターンのオブジェクト指向的要素

「オブジェクト指向的な要素」という視点から見ると、Flyweightパターンはちょっと特殊な形をしています。あえて言えば、処理の「カプセル化」を利用しているパターンです。

Flyweightパターンでは、Flyweight型のオブジェクトを内部に保持して管理しています。同時に、他のクラスからFlyweight型のオブジェクトを取得する方法も提供しています。ただし、一度保持したオブジェクトが再度必要になったとき、新たに生成するのではなく、保持したオブジェクトを返します。この結果、オブジェクトを生成する処理をクライアント側から隠蔽することができます。

関連するパターン

Compositeパターン

Compositeパターンは、Flyweightパターンと併用されることが多いパターンです。

Stateパターン、Strategyパターン

StateパターンやStrategyパターンは、Flyweightパターンと併用すると効率的な実装ができる場面が多いパターンです。

まとめ

ここではたくさんの小さなオブジェクトを共有するFlyweightパターンについて説明しました。