


Iterator ～順々にアクセスする

 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、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

振る舞い+オブジェクト

はじめに

ここではIteratorパターンについて説明します。

「iterate」には「繰り返す」「反復する」といった意味ですので、「iterator」は「反復するもの」となるでしょうか？

名前から想像できるように、Iteratorパターンはオブジェクトに対する反復操作をおこなうための統一APIを提供するパターンです。

たとえば

PHPで複数の値を含むリストを表現する場合、どの様にコードを記述していますか？おそらく、配列を使うことが多いのではないかと思います。

●複数の値を含むリスト

```
<?php
$arr = array('PHP', 'デザインパターン', 'iterator');
$arr[] = 'GoFパターン';
$arr['version'] = 'PHP5';
?>
```

次に、このリストに含まれる値をすべて表示する場合はどうでしょうか？多くの場合、for文やforeach文を使うと思います。特にforeach文を使用すると、添字が文字列である連想配列も扱うことができますね。

●リストに含まれる値をすべて表示するPHPスクリプト

```
<?php
// for文を使用した場合
for ($i = 0; $i < count($arr); $i++) {
    echo $arr[$i] . '<br>';
}
// foreach文を使用した場合
foreach ($arr as $value) {
    echo $value . '<br>';
}
?>
```

では、ディレクトリツリーを表すような不規則な多次元配列の場合はどうでしょうか？ある程度のプログラミング経験があれば、再帰を使えば何とかかなりそうだと気づくかも知れませんが。

このように、基本的にはfor文やforeach文を使用することで、リストの要素を取り出すことができます。

ここまではどの様なリストを処理するのが、あらかじめ分かっている状況でした。しかし、**どの様なリストを処理するのか分からない場合はどうでしょうか？**つまり、リストの内部構造が分からない状況です。ここで「すべての場合を想定してループを用意して・・・」と考えてしまうかも知れませんが、ちょっと現実的ではありませんね。

また、要素の値を判断して取り出す順序を変えたい場合はどうでしょうか？たとえば、氏名と年齢、性別を持つユーザオブジェクトのリストを考えてみましょう。このリストから氏名順にユーザオブジェクトを取り出したい場合もあれば、年齢順の場合もあるでしょう。女性、男性のどちらかだけというのもあり得ますね。また、それらの逆順で取り出したい場合もあるかも知れません。

このような色々な要求に対応する場合、要素を取り出すスクリプトコードを修正することになってしまいがちです。つまり、**コードの再利用はできない**ということです。

さて、こういった場合、どの様にすれば良いのでしょうか？

最初のコード例では、for文もforeach文もリストから1つずつ要素を取り出していました。ここに注目してみましょう。

やりたいことはリストから要素を取り出すことです。しかし、リストはどのような構造なのか分からなかったり、取り出す方法が複数あるかも知れません。

そこで、「**リストから1つずつ要素を取り出す**」役割を担うものを用意するとどうでしょうか？利用側は「次の要素をちょうだい」とお願いするだけで済むようになります。

一方の「リストから1つずつ要素を取り出す」役の方はどのようになりそうでしょうか？たとえば「次の要素をちょうだい」と依頼された場合は、リストの構造によって返す要素を決定し、その要素を返す処理をおこなえば良いことになります。つまり、利用者に、「**どのような構造を持つリストなのか**」を意識させないようにできます。また、ある条件にマッチする値だけ取り出したい場合も、この「リストから1つずつ要素を取り出す」役にうまく納めることができそうです。

このように、リストの内部構造を隠したまま、それぞれの要素にアクセスさせるためのパターンが**Iteratorパターン**です。

Iteratorパターンとは？

Iterator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。

集約オブジェクトが基にある内部表現を公開せずに、その要素に順にアクセスする方法を提供する。

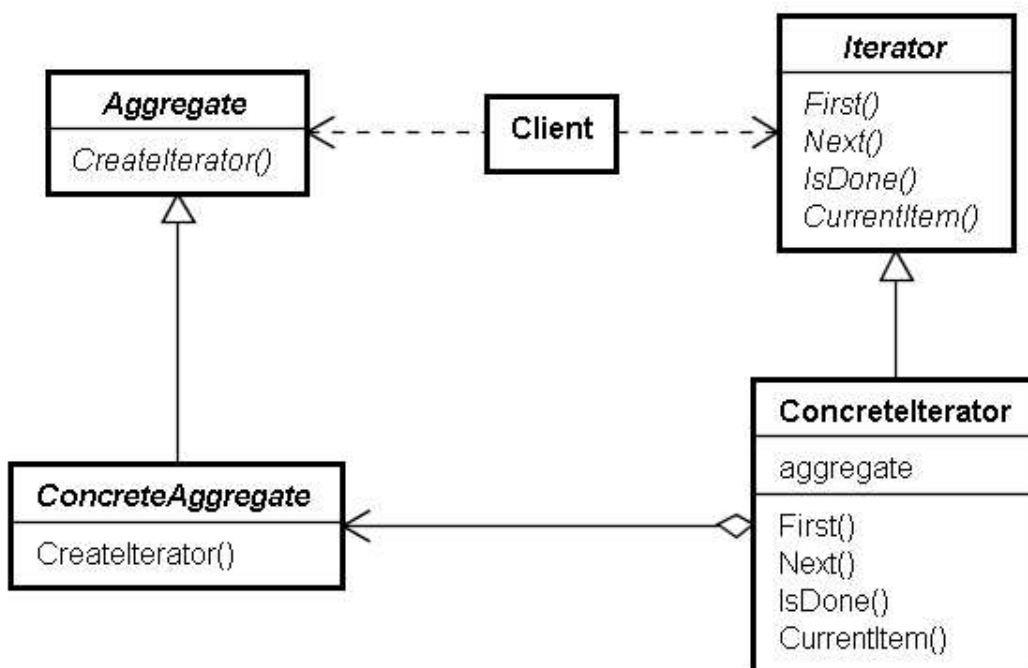
Iteratorパターンは、オブジェクトの振る舞いに注目したパターンです。

Iteratorパターンでは、リストのように**複数のオブジェクトをまとめる集約オブジェクトを走査**するためのAPIを提供します。これにより、利用側は集約オブジェクトの内部を意識することなく、要素にアクセスできます。その結果、**異なる内部構造を持つリストの要素に同じAPIでアクセス**できます。

また、走査処理の具体的な実装を変えることで、逆方向に走査させたり任意の要素に直接アクセスさせることもできます。

Iteratorパターンの構造

Iteratorパターンのクラス図と構成要素は、次のようになります。



Iteratorパターンの構成要素

Iteratorクラス

要素にアクセスするためのAPIを提供します。

ConcreteIteratorクラス

Iteratorクラスのサブクラスで、定義されたAPIを実装するクラスです。ここには、リストの内部構造に依存する走査処理が実装されます。

Aggregateクラス

Iteratorオブジェクトを返すAPIを提供します。

ConcreteAggregateクラス

Aggregateクラスのサブクラスで、リスト固有のIteratorオブジェクトを返します。

Iteratorパターンのメリット

Iterator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。

リストの具体的な内部構造をクライアントから隠蔽する

リストを走査する処理はすべてConcreteIteratorクラス内に閉じこめられます。このため、クライアントはリストの内部構造を意識することなく走査することができます。

リストに対する操作方法を複数用意できる

リストとそれを走査する役割のConcreteIteratorクラスが分かれていますので、異なる実装のConcreteIteratorクラスを用意することで、走査方法を容易に変更できます。

Iteratorパターンの適用例

Iteratorパターンの適用例を見てみましょう。

PHP5より**SPL (Standard PHP Library) 拡張**が追加されました。SPL拡張では標準的な関数やクラス、インターフェース、例外が定義されていますが、様々なイテレータも併せて用意されています。PHP5を使ってIteratorパターンを適用する場合、SPL拡張をそのまま、もしくは拡張して使用する場面が多くなるでしょう。

ここで挙げたサンプルアプリケーションは社員一覧を表示するものですが、次のSPL標準クラス・インターフェースを使用しています。

名称	概要
ArrayObjectクラス	配列をオブジェクトとして扱うためのラッパークラス
ArrayIteratorクラス	配列用のiteratorクラス。ConcreteIteratorクラスに相当する。
FilterIteratorクラス	iterator用のフィルタクラス。また、抽象クラスとして定義されているため、利用するにはクラスの継承をおこないacceptメソッドを実装する必要がある。
IteratorAggregateインターフェース	Aggregateクラスに相当する。iteratorを生成するためのメソッドgetIteratorが宣言されている。

では、早速サンプルアプリケーションのコードを見ていきましょう。

まずは社員を表すEmployeeクラスです。このクラスは、コンストラクタに氏名・年齢・職務名を受け取り、内部に保持するだけのクラスです。特に難しいところはないと思います。

●Employeeクラス (Employee.class.php)

```
<?php
class Employee {
    private $name;
    private $age;
    private $job;
    public function __construct($name, $age, $job) {
        $this->name = $name;
        $this->age = $age;
        $this->job = $job;
    }
    public function getName() {
        return $this->name;
    }
    public function getAge() {
        return $this->age;
    }
    public function getJob() {
        return $this->job;
    }
}
```

次は社員リストを表すEmployeesクラスです。SPL拡張のIteratorAggregateインターフェースを実装し、ConcreteAggregateクラスに相当します。

●Employeesクラス (Employees.class.php)

```

<?php
require_once 'Employee.class.php';
?>
<?php
class Employees implements IteratorAggregate {
    private $employees;
    public function __construct() {
        $this->employees = new ArrayObject();
    }
    public function add(Employee $employee) {
        $this->employees[] = $employee;
    }
    public function getIterator() {
        return $this->employees->getIterator();
    }
}
?>

```

Employeesクラスは内部に複数のEmployeeオブジェクトを保持しますが、配列ではなくArrayObjectオブジェクトで管理します。また、getIteratorメソッドを実装し、ArrayObjectクラスのgetIteratorメソッドを呼び出しています。このメソッドからは、ArrayIteratorオブジェクトが返されます。利用側は、このArrayIteratorオブジェクトを使って社員リストにアクセスします。

続いて、SalesmanIteratorクラスを見てみましょう。このクラスはSPL拡張のFilterIteratorクラスを継承しています。

●SalesmanIteratorクラス (SalesmanIterator.class.php)

```

<?php
require_once 'Employee.class.php';
?>
<?php
class SalesmanIterator extends FilterIterator {
    public function __construct($iterator) {
        parent::__construct($iterator);
    }

    public function accept() {
        $employee = $this->current();
        return ($employee->getJob() === 'SALESMAN');
    }
}

```

FilterIteratorクラスは「オブジェクトを包み込むクラス」で、包み込まれるオブジェクトに付加的な機能を提供します。これは、FilterIteratorクラスの抽象メソッドであるacceptメソッドを実装することで実現します。

SalesmanIteratorクラスでは、包み込むArrayIteratorオブジェクトに「動作を変更する」という機能を提供するために、acceptメソッドに取得する条件を実装しています。その結果、条件にマッチした値のみを取り出すことが可能です。今回は、職務名が「SALESMAN」であるオブジェクトだけを取り出すようになっています。

また、FilterIteratorクラスを使わなくとも、独自のIteratorを実装しても実現可能です。この場合も**走査対象のEmployeesクラスやEmployeeクラスには影響を与えません**。

最後に、説明してきたクラス群を利用するクライアント側のコードです。

まず、社員リストを作成し、イテレータを取り出しています。先ほど説明したとおり、イテレータはArrayIteratorオブジェクトになります。その後、イテレータのメソッドを利用して一覧を表示する場合と、foreach文を使って表示しています。

●クライアント側コード (iterator_client.php)

```

<?php
require_once 'Employee.class.php';
require_once 'Employees.class.php';
require_once 'SalesmanIterator.class.php';
?>
<?php
function dumpWithForeach($iterator) {
    echo '<u|>';
    foreach ($iterator as $employee) {
        printf('<li>%s (%d, %s)</li>',
            $employee->getName(),
            $employee->getAge(),

```

```

        $employee->getJob());
    }
    echo '</ul>';
    echo '<hr>';
}
?>
<?php
$employees = new Employees();
$employees->add(new Employee('SMITH', 32, 'CLERK'));
$employees->add(new Employee('ALLEN', 26, 'SALESMAN'));
$employees->add(new Employee('MARTIN', 50, 'SALESMAN'));
$employees->add(new Employee('CLARK', 45, 'MANAGER'));
$employees->add(new Employee('KING', 58, 'PRESIDENT'));

$iterator = $employees->getIterator();

/**
 * iteratorのメソッドを利用する
 */
echo '<ul>';
while ($iterator->valid()) {
    $employee = $iterator->current();
    printf('<li>%s (%d, %s)</li>',
        $employee->getName(),
        $employee->getAge(),
        $employee->getJob());

    $iterator->next();
}
echo '</ul>';
echo '<hr>';

/**
 * foreach文を利用する
 */
dumpWithForeach($iterator);

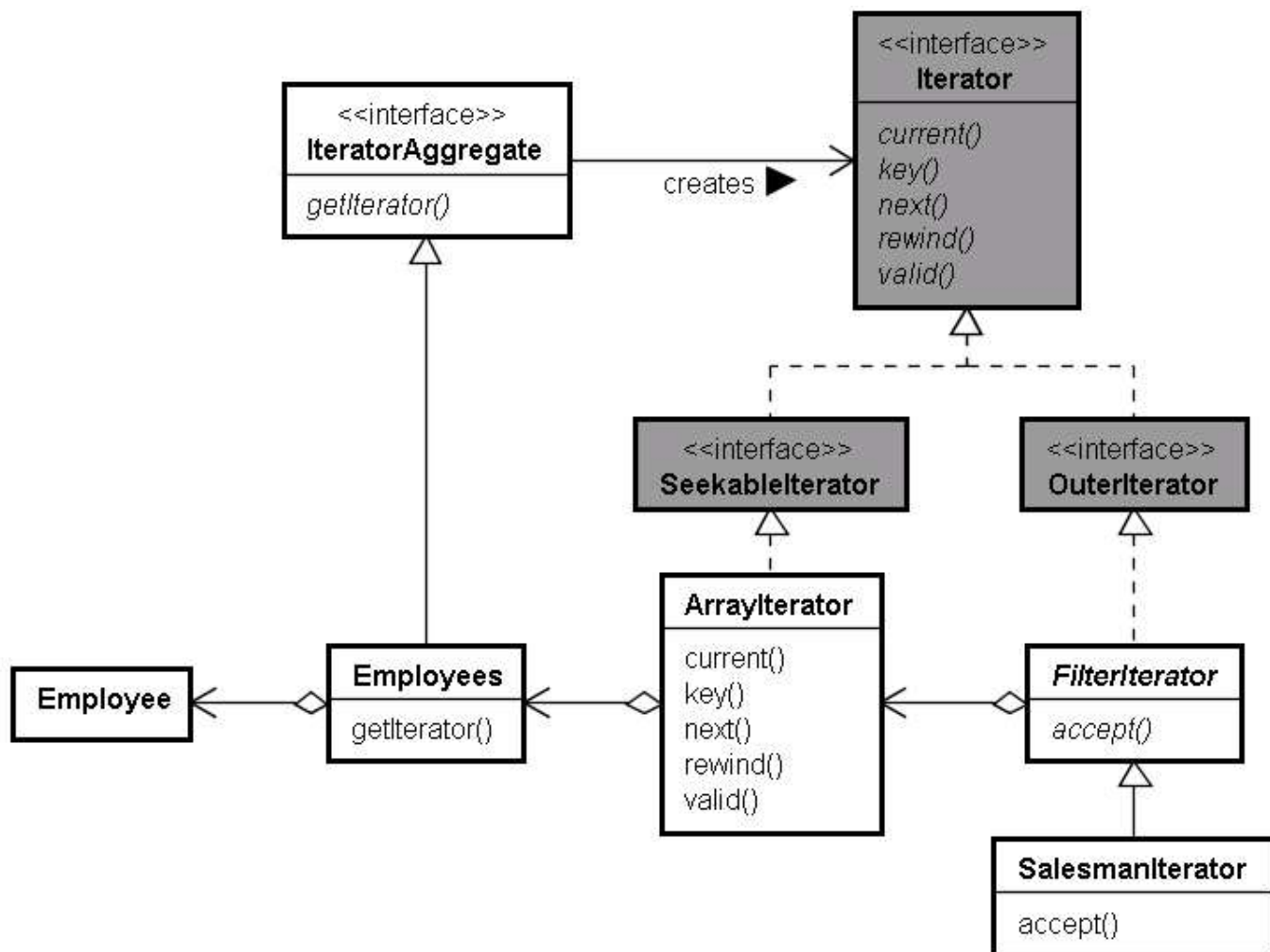
/**
 * 異なるiteratorで要素を取得する
 */
dumpWithForeach(new SalesmanIterator($iterator));
?>

```

ご覧の通り、イテレータのメソッドでもforeach文でも同様の結果を得られます。つまり、foreach文が暗黙的なイテレータとなり得ます。このため、Iteratorパターンを適用した場合、イテレータのメソッドではなくforeach文が使われることが多いでしょう。ただし、取り出す方法はConcreteIteratorクラスに依存しています。

そこで最後に、異なるイテレータを使って営業職の社員だけを一覧表示しています。このとき、Employeesオブジェクトや表示するためのコードを修正する必要はありませんし、新しいFilterIteratorクラスのサブクラスを作成するだけで新しい取得方法を追加できるということを確認してください。

最後にIteratorパターンを適用したアプリケーションのクラス図は次のようになっています。なお、色付きのインターフェースは、先の説明では出てきていないものになります。



Iteratorパターンのオブジェクト指向的要素

Iteratorパターンは「非カプセル化」をおこなうパターンです。

これまで見てきたように、Iteratorパターンでは集約オブジェクトを走査する「操作」を他のクラスに任せています。

本来、オブジェクト指向では、関連するデータと操作を一つの塊(オブジェクト)として扱うことでカプセル化を行いますので、集約オブジェクト自身に操作するためのコードを用意するのが本来の姿かも知れません。しかし、無理にカプセル化することでオブジェクトが肥大化したり、複雑になり過ぎる場合があります。たとえば、様々な走査をサポートしたい場合を考えてみてください。

また、集約オブジェクトを走査する場面はかなり多いものと思います。当然、多くのクラスに似たようなコードを持たせることになります。しかし、**それぞれのクラスに走査のためのコードを記述するよりも、別のクラスに記述して再利用した方が有利になると考えられます**。また、集約オブジェクトに影響を与えないで具体的な実装を変えることもできます。

Iteratorパターンでは、**集約オブジェクトに対する操作を別クラスとして切り出して**、オブジェクトが不要に肥大化したり複雑になることを避けています。確かに、集約オブジェクトを操作する場合は切り出されたクラスと連携する必要がありますが、切り出されたクラスを切り替えることで様々な操作をサポートできるようになっています。

関連するパターン

Compositeパターン

Compositeパターンは再帰的な構造を作るパターンです。このような構造に対してIteratorパターンが適用されることがあります。

Factory Methodパターン

適切なConcreteIteratorクラスをインスタンス化するために、Factory Methodパターンが適用される場合があります。

まとめ

ここでは集約オブジェクトから要素を取り出すためのAPIを提供するIteratorパターンについて見てきました。集約オブジェクトとは別のクラスに具体的に切り出す処理を任せることで、様々な操作をサポートできるようになります。