JavaScriptデザインパターン – 第2部：アダプター、デコレーター、ファクトリ

著者 Joseph Zimmerman

http://www.joezimjs.com

Content

アダプター
デコレーター
ファクトリ

作成日

15 May 2012

ページ ツール

Facebookでシェア
Twitterでツイート
LinkedInでシェア
印刷

この記事に設定されたタグ

HTML5 JavaScript mobile

必要条件

この記事に必要な予備知識
Basic JavaScript programming knowledge.

ユーザーレベル
すべて

本稿はJavaScriptデザインパターン特集の第2部です。第1部からしばらく時間が空いてしまったので、シングルトン、コンポジット、ファサードの各パターンについて、記憶を新たにした方がよいかもしれません。また、第3部では、さらにプロキシ、オブザーバー、コマンドという3つのデザインパターンについて説明します。

今回は、アダプター、デコレーター、ファクトリの各パターンについて学習します。

アダプター

アダプターパターンでは、ニーズに合うようにインターフェイスを変換（適応）させます。それには、必要なインターフェイスを備えたオブジェクトを別途作成し、そのオブジェクトを、インターフェイスを変更したいオブジェクトに接続します。

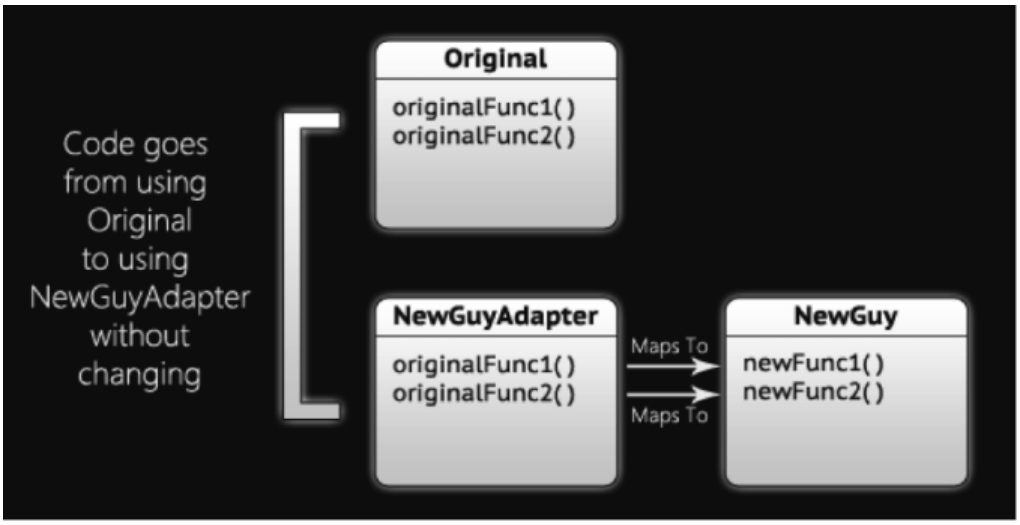

図1.アダプターパターンの構造

アダプターを必要とする理由

非常によくあることですが、アプリケーションの開発や保守を行っていて、アプリケーションのチャンク（例えばログやその種のデータの保管に使用しているライブラリ）を置き換える必要があると判断したとします。新しく置き換えるために用意したライブラリが、古いライブラリとまったく同じインターフェイスを持っている可能性は低いでしょう。この先の作業については、次の2つの選択肢があります。

1. コード全体を確認し、古いライブラリを参照している部分をすべて変更する
2. 新しいライブラリで古いライブラリとまったく同じインターフェイスを使用できるようにアダプターを作成する

アプリケーションが小さかったり、古いライブラリの参照が少ししかない場合は、新しい抽象化層でコードをコンパイルするよりも、コード全体を見直して新しいライブラリに合うようにコードを変更した方がよいと思えることもあるでしょう。とはいえ、多くの場合は、アダプターを作成した方が現実的で時間の節約になります。

アダプターの例

それでは、このコードサンプルに、上述の仮想のロガーシナリオを適用してみましょう。元の自分のコードで、ログの生成用にブラウザーに組み込まれているコンソールを使用しているかもしれませんが、それには若干問題があります。コンソールはすべてのブラウザーに組み込まれているわけではないので（特に古いブラウザーにはないことが多い）、自分のアプリケーションを他のユーザーが使用した場合に生成されるログを見ることができず、自己テストで検出できなかった問題を調べられないのです。そこで、ロガーを導入し、AJAXを使用してそれらのログをサーバーに転送して処理してもらうことにしました。新しい`AjaxLogger`ライブラリのAPIは以下のようになります。

```
AjaxLogger.sendLog(arguments);
AjaxLogger.sendInfo(arguments);
AjaxLogger.sendDebug(arguments);
etc...
```

見たところ、このライブラリの作成者は、開発者がコンソールの置き換えにこのコードを使用することを想定していなかったため、各メソッド名の先頭に「send」を付ける必要があると考えたようです。そこで、「ライ

ブラリを編集して、メソッド名を変更すれば済む話ではないか？」という疑問が出てきます。しかし、次の2つの理由により、その方法はお勧めしません。ライブラリをアップデートする必要が出てきたときに変更が上書きされてしまうので、開発者は再び名前変更の作業をやり直すことになります。また、ライブラリをコンテンツデリバリーネットワークからプルダウンする場合に名前を編集できないからです。そこで、新しいライブラリをコンソールと同じインターフェイスに適応させるオブジェクトを作成することにしましょう。

```
var AjaxLoggerAdapter = {
    log: function() {
        AjaxLogger.sendLog(arguments);
    },
    info: function() {
        AjaxLogger.sendInfo(arguments);
    },
    debug: function() {
        AjaxLogger.sendDebug(arguments);
    },
    ...
};
```

使い方
コンソールを使用する開発者はたいてい、参照によって直接コンソールを呼び出すものです。そこで、`console.xxx`の呼び出しごとに、コンソールではなく、新しいアダプターを参照するにはどうすればよいでしょう。ファクトリのような抽象化を使用していた場合、抽象化層に変更を加えるだけで済みますが、上で述べたとおり、皆が単に`console`を直接参照しているのです。さて、JavaScriptは動的な言語であり、実行時に変更を行うことができます。そこで、新しい`AjaxLoggerAdapter`でコンソールをオーバーライドしてみたらどうでしょうか。

```
window.console = AjaxLoggerAdapter;
```

このやり方は簡単ですが、それだけに注意が必要です。他の人によって使用される前提のコードに対してこのようにすると、コンソールは、それらのユーザーの期待どおりには機能しなくなります。また、このコード例の単純さにだまされないでください。多くの場合、メソッドが簡単に対応し合う（`sendLog`と`log`のように）ことはありません。新しいライブラリと互換性を保つようにインターフェイスを変換するには、実際に自分のロジックを多少実装する必要があるかもしれません。

デコレーター

デコレーターパターンは、他の多くのパターンとはかなり異なります。デコレーターパターンを使用すれば、機能の組み合わせごとにサブクラスを作成することなく、クラスへの機能の追加または変更という問題が解決されます。例えば、デフォルトの機能を備えたcarというクラスがあるとします。carには、車に追加できる多数のオプションの機能があります（自動ロック、パワーウィンドウ、エアコンなど）。サブクラスを使用してこれを処理しようとすると、すべての組み合わせに対応するには合計で8つのクラスが必要になります。

- Car
- CarWithPowerLocks
- CarWithPowerWindows
- CarWithAc
- CarWithPowerLocksAndPowerWindows
- CarWithPowerLocksAndAc
- CarWithPowerWindowsAndAc
- CarWithPowerWindowsAndPowerLocksAndAc

この方法は、オプションを1つ増やすだけで8つのサブクラスが追加されるので、収拾がつかなくなりやすいです。これはデコレーターパターンを使用することで解決できます。この方法では、新しいオプションを追加するたびに、1つのクラスを作成するだけで済み、クラスが倍増することはありません。

デコレーターの構造
デコレーターは、基本オブジェクト（Car）を、基本オブジェクトと同じインターフェイスを持つデコレーターオブジェクトでラップすることで機能します。デコレーターには、メソッドの処理方法に関して次のようなオプションがあります。

1. ラップされているオブジェクトのメソッドを完全にオーバーライドすることができる
2. デコレーターがメソッドの挙動に影響しない場合、ラップされたオブジェクトに単に渡される
3. デコレーターは、ラップされたオブジェクトに呼び出しを渡す前または渡した後に挙動を追加できる

デコレーターは基本オブジェクトをラップできるだけでなく、他のデコレーターをラップすることもできます。これは、すべてのデコレーターが同じインターフェイスを実装しているからです。一般的な構造は、図2のような感じになります。

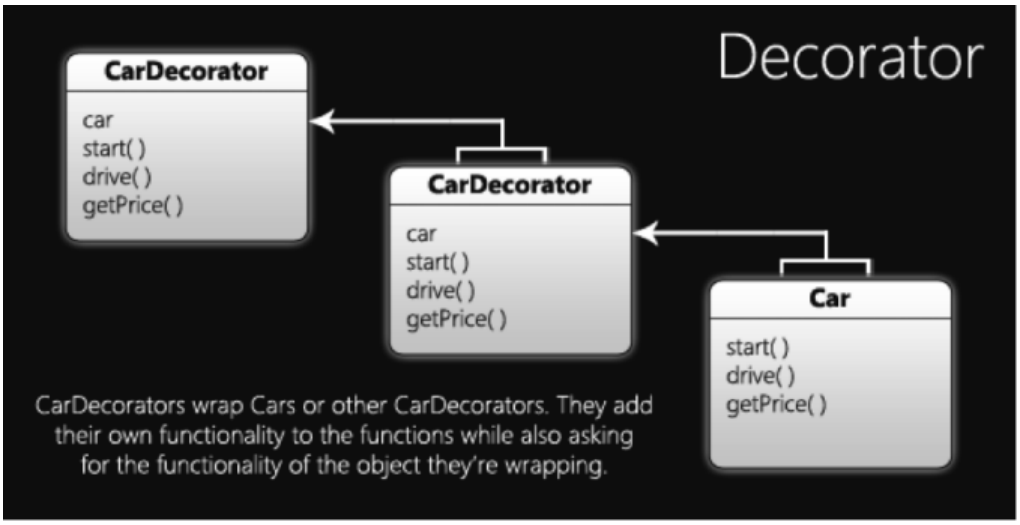

図2.デコレーターパターンの構造

デコレーターパターンの例

上のcarの図をコード化してみましょう。まず、ベースの`Car`クラスを構築します。

```
var Car = function() {
    console.log('Assemble: build frame, add core parts');
};

// The decorators will also need to implement this interface
Car.prototype = {
    start: function() {
        console.log('The engine starts with roar!');
    },
    drive: function() {
        console.log('Away we go!');
    },
    getPrice: function() {
        return 11000.00;
    }
};
```

フレーズをコンソールに単に送信し、何が起こっているかを示すことで、構造をシンプルに保ちます。もちろん、自動車は複雑な機械なので、もっと多くの機能を追加することもできますが、このデモンストレーションではシンプルな構造のまま使用します。

次に、`CarDecorator`クラスを作成します。これは抽象化クラスとなるのでクラス自体はインスタンス化しませんが、本格的なデコレーターを作成するためにサブクラス化する必要があります。

```
var CarDecorator = function(car) {
    this.car = car;
};

// CarDecorator implements the same interface as Car
CarDecorator.prototype = {
    start: function() {
        this.car.start();
    },
    drive: function() {
        this.car.drive();
    },
    getPrice: function() {
        return this.car.getPrice();
    }
};
```

いくつか重要なポイントがあります。まず、`CarDecorator`コンストラクターは、carを取ります。さらに厳密に言うと、`Car`と同じインターフェイスを実装するオブジェクトを取ります。これには、複数の`Car`や、`CarDecorator`サブクラスが含まれます。また、すべてのメソッドは、ラップされたオブジェクトに要求を渡すだけです。これは、すべてのデコレーターが、変更されていないメソッドに対して使用するデフォルトの挙動です。

次に、デコレーターを全部作成します。CarDecoratorから継承しているので、変更する機能をオーバーライドするだけです。他のすべての点で、CarDecoratorは完全に機能します。

```
var PowerLocksDecorator = function(car) {
    // Call Parent Constructor
    CarDecorator.call(this, car);
    console.log('Assemble: add power locks');
};
PowerLocksDecorator.prototype = new CarDecorator();
PowerLocksDecorator.prototype.drive = function() {
    // You can either do this
    this.car.drive();
    // or you can call the parent's drive function:
    // CarDecorator.prototype.drive.call(this);
    console.log('The doors automatically lock');
};
PowerLocksDecorator.prototype.getPrice = function() {
    return this.car.getPrice() + 100;
};

var PowerWindowsDecorator = function(car) {
    CarDecorator.call(this, car);
    console.log('Assemble: add power windows');
};
PowerWindowsDecorator.prototype = new CarDecorator();
PowerWindowsDecorator.prototype.getPrice = function() {
    return this.car.getPrice() + 200;
};

var AcDecorator = function(car) {
    CarDecorator.call(this, car);
    console.log('Assemble: add A/C unit');
};
AcDecorator.prototype = new CarDecorator();
AcDecorator.prototype.start = function() {
    this.car.start();
    console.log('The cool air starts blowing.');
};
AcDecorator.prototype.getPrice = function() {
    return this.car.getPrice() + 600;
};
```

この例では、オリジナルの機能に追加されるメソッドを追加するたびに、親クラスのメソッドを呼び出すことはせず、単にthis.car.x()を呼び出します。通常は、親のメソッドを呼び出した方が適切なのですが、このコードは非常にシンプルなので、carのプロパティを直接呼び出した方が簡単だと判断しました。このようにすると、もし要求を渡す以外の機能を担わせるためにCarDecoratorに組み込まれたデフォルトの挙動を変更する必要が生じた場合に面倒なことになりますが、これは単なるサンプルなので、その局面は多分ないでしょう。

そういうわけで、必要な要素はすべて揃ったので、実際に作成してみましょう。

```
var car = new Car();                    // log "Assemble: build frame, add core pa

// give the car some power windows
car = new PowerWindowDecorator(car);    // log "Assemble: add power windows"

// now some power locks and A/C
car = new PowerLocksDecorator(car);     // log "Assemble: add power locks"
car = new AcDecorator(car);             // log "Assemble: add A/C unit"

// let's start this bad boy up and take a drive!
car.start(); // log 'The engine starts with roar!' and 'The cool air starts blowin
car.drive(); // log 'Away we go!' and 'The doors automatically lock'
```

必要な機能を備えた自動車を作成するのは簡単です。自動車および必要なデコレーターを作成し、順次、最新の装備セットを備えた自動車をデコレーターに追加していきます。特に、CarWithPowerLocksAndPowerWindowsAndAcのような1つのオブジェクトを単にインスタンス化するのと較べると、作成コードは非常に長くなります。しかし心配は無用です。装飾されたオブジェクトを作成するよりスマートな方法を、ファクトリパターンに関する項でご紹介します。デコレーターパターンについてもう少

し詳しく知りたい場合は、筆者の個人ブログの「JavaScript Design Patterns: Decorator」の記事をお読みください。

ファクトリ

ファクトリという名前は、オブジェクトの作成を単純化するという、その用途に由来しています。シンプルなファクトリは、新しいキーワードである用途すべてを抽象化したもので、クラス名が変更されたり他のクラス名で置き換えられたりしたとしても、1か所で変更を加えるだけで済みます。さらに、多数の異なる種類のオブジェクト、または異なるオプションを持つ1種類のオブジェクトを作成するための、ワンストップショップを設定します。標準的なファクトリについて手短に解説するのはちょっと難しいので、また後で説明します。

シンプルなJavaScriptファクトリ

この例は、サンプルのファクトリで、上記のデコレーターの例で生じた、「各種機能を搭載した自動車を作成するための実際のコードは長すぎる」という難題を使用しています。ファクトリを使用することで、この長いコードを減らして、1つの関数呼び出しにまとめることができます。最初に、1つの関数を含むオブジェクトリテラルを作成します。JavaScriptでは、オブジェクトリテラル/シングルトンは、シンプルなファクトリを構築する方法です。従来型のオブジェクト指向のプログラミング言語では、静的クラスがこれに相当します。

```
var CarFactory = {
    makeCar: function(features) {
        var car = new Car();

        // If they specified some features then add them
        if (features && features.length) {
            var i = 0,
                l = features.length;

            // iterate over all the features and add them
            for (; i < l; i++) {
                var feature = features[i];

                switch(feature) {
                    case 'powerwindows':
                        car = new PowerWindowsDecorator(car);
                        break;
                    case 'powerlocks':
                        car = new PowerLocksDecorator(car);
                        break;
                    case 'ac':
                        car = new ACDecorator(car);
                        break;
                }
            }
        }

        return car;
    }

}
```

このファクトリが持つ1つの関数が`makeCar`であり、多数の煩雑な処理を実行します。まず第一に、関数が受け取る引数は、別のdecoratorクラスに対応付けられる文字列の配列です。`makeCar`はプレーンな`Car`オブジェクトを作成し、機能を反復処理し、自動車を装飾します。次に、これらの機能をすべて備えた自動車を作成しますが、最少でも4行のコードを記述する代わりに、次の1行だけが必要です。

```
Var myCar = CarFactory.makeCar(['powerwindows', 'powerlocks', 'ac']);
```

`makeCar`関数を調整して、任意の種類のデコレーターが1つだけ使用され、指定した順序で（実際に工場で製造されるように）アタッチされるようにすることができます。このコード例や、ファクトリパターンを効果的に使用する方法について詳しくは、筆者の個人ブログで「JavaScript Design Patterns: Factory」の記事をお読みください。

標準のファクトリ

標準のファクトリパターンは、シンプルなファクトリとはかなり異なりますが、もちろん、オブジェクト作成という役割を持つ点は同じです。シングルトンを使用するのではなく、クラスに対して単に抽象化メソッドを使用します。例えば、様々な自動車メーカーがあり、それぞれに専用の販売代理店があるとします。ほとんどの場合、すべての販売代理店は同じ販売方法を採用していますが、メーカーが製造する自動車だけが異なっています。したがって、すべての販売店は同じプロトタイプからメソッドを継承しますが、製造プロセスだけは独自のものを実装します。

上の説明を理解しやすいように、この例をコード化してみましょう。最初に、サブクラス化のみを目的とした`manufactureCar`というメソッドを1つ持っている自動車販売代理店を作成します。これがスタブです。これは、サブクラスによって上書きされない限り、エラーを返します。

```
/* Abstract CarShop "class" */
var CarShop = function(){};
CarShop.prototype = {
    sellCar: function (type, features) {
        var car = this.manufactureCar(type, features);

        getMoney(); // make-believe function

        return car;
    },
    decorateCar: function (car, features) {
        /*
        Decorate the car with features using the same
        technique laid out in the simple factory
        */
    },
    manufactureCar: function (type, features) {
        throw new Error("manufactureCar must be implemented by a subclass");
    }
};
```

`sellCar`が`manufactureCar`を呼び出すことに注目してください。これは、自動車を販売するには、`manufactureCar`がサブクラスによって実装される必要があることを意味しています。そこで、対になる自動車販売店を作成し、どのように実装するのか見てみましょう。

```
/* Subclass CarShop and create factory method */
var JoeCarShop = function() {};
JoeCarShop.prototype = new CarShop();
JoeCarShop.prototype.manufactureCar = function (type, features) {
    var car;

    // Create a different car depending on what type the user specified
    switch(type) {
        case 'sedan':
            car = new JoeSedanCar();
            break;
        case 'hatchback':
            car = new JoeHatchbackCar();
            break;
        case 'coupe':
        default:
            car = new JoeCoupeCar();
    }

    // Decorate the car with the specified features
    return this.decorateCar(car, features);
};


/* Another CarShop and with factory method */
var ZimCarShop = function() {};
ZimCarShop.prototype = new CarShop();
ZimCarShop.prototype.manufactureCar = function (type, features) {
    var car;

    // Create a different car depending on what type the user specified
    // These are all Zim brand
    switch(type) {
        case 'sedan':
            car = new ZimSedanCar();
            break;
        case 'hatchback':
            car = new ZimHatchbackCar();
            break;
        case 'coupe':
        default:
            car = new ZimCoupeCar();
```

```
    }

    // Decorate the car with the specified features
    return this.decorateCar(car, features);
};
```

メソッドの挙動は基本的に同じですが、それぞれが別種の自動車を製造するという点で異なります（ここでは必要ないため、carクラスの実装を省略して簡潔にしています）。ポイントは、`manufactureCar`メソッドがファクトリメソッドであるという点です。ファクトリメソッドは親クラスの抽象化であり、サブクラスによって実装され、オブジェクトの作成を担当します（各ファクトリメソッドは同じインターフェイスを備えたオブジェクトを作成します）。

仕上げ
ここでは、ファクトリパターンの基本について説明しました。ファクトリパターンについてもう少し詳しく知りたい場合は、筆者の個人ブログの「JavaScript Design Patterns: Factory」（シンプルなファクトリ）および「JavaScript Design Patterns: Factory Part 2」（標準的なファクトリ）の記事をお読みください。

次のステップ

以上でJavaScriptデザインパターンシリーズの第2部を終了します。おそらく、JavaScriptにおけるアダプター、デコレーター、ファクトリの各パターンの実装方法に関する基本は理解いただけたでしょう。このシリーズ記事の第1部では、シングルトン、コンポジット、ファサードという別の3つのデザインパターンについて紹介しました。第3部では、さらに、プロキシ、オブザーバー、コマンドという3つのデザインパターンを取り上げます。

JavaScriptデザインパターンに関する第3部にして最終編となる記事は、近日中に掲載されます。もし待ちきれないという場合は、筆者の個人ブログ「JavaScript Design Patterns on Joe Zim's JavaScript Blog」をお読みください。このブログには、JavaScriptのデザインパターンに関する完結済みのシリーズ記事を掲載しており、このシリーズで取り上げないパターンをいくつか紹介しています。

More Like This

Using the Geolocation API
CSSシェーダー：Web上の映画的エフェクト
CSS3の基本
CSS Regions：HTMLとCSS3でリッチなページレイアウトを実現する
JavaScript design patterns – Part 3: Proxy, observer, and command
JavaScriptデザインパターン – 第1部：シングルトン、コンポジット、ファサード
Integrating Rails and jQuery Mobile
CSS Blending 入門
Flame on! A beginner's guide to Ember.js
セマンティックHTMLの使用