JavaScript design patterns – Part 3: Proxy, observer, and command

著者 Joseph Zimmerman

http://www.joezimjs.com

Content

作成日
23 July 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

CSS3 HTML5 JavaScript

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デザインパターンシリーズの最終編です。第1部ではシングルトン、コンポジット、ファサードの各パターンについて、第2部ではアダプター、デコレーター、ファクトリの各パターンについて説明しました。今回は、プロキシ、オブザーバー、コマンドの各パターンに関する知識を皆さんの頭に詰め込みます。

プロキシ

「プロキシ」の語義は「*代理*」であり、この語義自体がプロキシパターンの役目を十分に説明しています。さて、一体何の代理をするのでしょうか。また、なぜオリジナルを使用しないのでしょうか。

プロキシを必要とする理由

基本的には、他のオブジェクトの代理をするオブジェクトを作成することになります。以下のような理由により、プロキシオブジェクトの作成が必要となります。

1. 本当に必要になるまで、大きなオブジェクトのインスタンス化を遅らせる
2. リモートのオリジナルオブジェクトにアクセスを提供する
3. オリジナルオブジェクトに対するアクセスを制御する

仮想プロキシ

仮想プロキシは、上のリストの最初のケースに相当し、大きなオブジェクトのインスタンス化を遅らせます。仮に、コード内に巨大なオブジェクトがあるとします。ここでの巨大オブジェクトとは、長く複雑なロジックを持つ機能を多数備えたオブジェクトか、あるいは膨大なデータを持つオブジェクトです。仮想プロキシの使用が適しているのは、オブジェクトがインスタンス化されるが使用されないという確かな見通しがある場合、または、負荷の高い作業を直ちに実行すると、高速な処理と応答を行いたいときに処理が遅くなるので、本当に必要になるまでその作業を延期したい場合です。例えば、`VehicleList`というオブジェクトがあるとします。これがオリジナルのオブジェクトです。このオブジェクトは、インスタンス化されたときに、自動車ごとにT型フォード以来の自動車メーカーとモデルの長いリストが含められます。そして、明らかに、作成に高いコストがかかります。このコードは仮想プロキシを作成し、オブジェクトのインスタンス化を遅らせます。

```
var VehicleListProxy = function() {
    // Variable to hold real VehicleList when it's instatiated
    this.vehicleList = null;
};
VehicleListProxy.prototype = {
    // Whenever a method requires that the real VehicleList is created
    // it should call this function to initialize it (if not already
    // initialized)
    _init: function() {
        if ( !this.vehicleList ) {
            this.vehicleList = new VehicleList();
        }
    },
    getCar: function( ... ) {
        this._init();
        return this.vehicleList.getCar( ... );
    },
    ...
}
```

このコードは、すべての要求をオリジナルオブジェクトに渡すオブジェクトを作成するだけですが、要求のいずれかが実行されるまでオリジナルオブジェクトのインスタンス化は行われません。必要なときにオリジナルオブジェクトが確実にインスタンス化されるように、他のすべてのメソッドによって呼び出されるメソッドを作成しました。

リモートプロキシ

リモートプロキシは、仮想プロキシと基本的に同じですが、オリジナルオブジェクトのインスタンス化を遅延させるのではなく、オリジナルオブジェクトはインターネット上のリモートの場所に既に存在しています。オブジェクトプロパティを通じて直接アクセスできるオリジナルオブジェクトに要求を単に渡すのではなく、AJAX要求として渡す必要があります。同じインターフェイスを採用するリモートオブジェクトがサーバーに存在していても問題ありません。オブジェクトが存在しているかどうかも問題になりません。そのURLへのAJAX要求が期待どおりに動作し、期待する値を返す限り、サーバー上に何が存在していてもかまいません。`VehicleListProxy`を使用している例をもう一度見てみましょう。

```
var VehicleListProxy = function() {
    this.url = "http://www.welistallcarmodels.com";
};
VehicleListProxy.prototype = {
    getCar: function( ... ) {
        // Skip the rest of the implementation to just show the
        // important stuff
        ajax( this.url + "/getCar/" + args);
        // Package up the data that you got back and return it
    },
    ...
}
```

シンプルそのものです。サーバーに処理を任せる場合と処理をローカルで行う場合を判定する明示的な関数を通して、Backbone.js内のモデルとコレクションは、ほとんどプロキシと同様に機能します。本記事で触れた「モデル」、「コレクション」、「Backbone.js」などの用語になじみがない場合は、Adobe Developer Connectionの「Christophe Coenraets' Backbone.js tutorials」をお読みいただくか、筆者が製作した「Backbone.jsビデオチュートリアルシリーズ」をご覧いただくと役に立つかもしれません。または、特に気にせず忘れていただいても大丈夫です。

プロキシ経由のアクセス制御

次は、プロキシのユースケースの最後となる「オリジナルオブジェクトに対するアクセス制御」をご紹介します。JavaScriptにおけるこの種のプロキシについて最初に理解しなければならないのは、実際にオブジェクトへのアクセスを制御するには、そのオブジェクトの周囲にクロージャが必要であるということです。クロージャがないと、グローバルスコープからアクセス可能になってしまいます。次に、必要とするすべての人に対して、確実にプロキシが利用可能になるようにします。この例では、計画されたリリース日までは、だれもアクセスできないようにします。どのようにこれを行ったか、以下のコードを見てみましょう。

```
// Close it off in a function
(function() {
// We already know what the VehicleList looks like, so I
// won't rewrite it here
var VehicleList = ...

var VehicleListProxy = function() {
    // Don't initialize the VehicleList yet.
    this.vehicleList = null;
    this.date = new Date();
};
VehicleListProxy.prototype = {
    // this function is called any time any other
    // function gets called in order to initialize
    // the VehicleList only when needed. The VehicleList will
    // not be initialized if it isn't time to yet.
    _initIfTime: function() {
        if ( this._isTime() ) {
            if ( !this.vehicleList ) {
                this.vehicleList = new VehicleList();
            }
            return true;
        }
        return false;
    },

    // Check to see if we've reached the right date yet
    _isTime() {
        return this.date > plannedReleaseDate;
    },

    getCar: function(...) {
        // if _initIfTime returns a falsey value, getCar will
        // return a falsey value, otherwise it will continue
        // and return the expression on the right side of the
        // && operator
        return this._initIfTime() && this.vehicleList.getCar(...);
    },
    ...
}

// Make the VehicleListProxy publicly available
```

```
window.VehicleListProxy = VehicleListProxy;

// you could also do the below statement so people don't even
// know they're using a proxy:
window.VehicleList = VehicleListProxy;

}());
```

上のコードでは、すべてのパブリックメソッドが、まず、アクセスできるかどうかを確認するために`this.initIfTime()`を呼び出し、次に、アクセスを許可された場合は、必ず`vehicleList`がインスタンス化されるようにします。`initIfTime()`が`false`を返した場合は、`return`ステートメント内の条件式が`false`を返し、その次の式は実行されません。しかし、メソッドが`initIfTime`から`true`を受け取った場合は、その次の式が実行され、値が返されます。

最終的に、最後の数行で、`window`にアタッチすることによって、コードの他の部分に対してプロキシが公開されます。上で示したように、ユーザーがプロキシの存在を意識さえしなくても、プロキシ経由でのアクセス制御を行うことができるのです。

プロキシ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JavaScriptブログで「JavaScript Design Patterns: Proxy」の記事をお読みください。

オブザーバー

さて、今度はオブザーバーについて説明します。オブザーバーパターンは、JavaScriptで最もよく使用されているパターンの1つです（Publish/Subscribeとも呼びます。省略形はPub/Sub）。これは、DOM要素のイベント処理は、（あらゆる規模のJavaScriptアプリケーションで非常によく見られることですが）オブザーバーパターン経由で実行されるからです。

オブザーバーパターンの説明

オブザーバーパターンはシンプルで、2つのエンティティで構成されます。監視対象のオブジェクト（「監視対象」と呼ばれる）と、監視対象を監視するオブジェクト（「オブザーバー」と呼ばれる）です。どちらも図1に示されています。DOMイベントの場合は、オブザーバーオブジェクトが特定のインターフェイスを採用することを要求しないため、JavaScriptでよくあるように、オブザーバーは単なるコールバック関数です。大半の言語では関数/メソッドはファーストクラスオブジェクトではなく渡せないため、このような使い方はできませんが、JavaScriptはこの点においては非常に幸運です。

図1.オブザーバーパターンの構造

オブザーバーの種類

オブザーバーパターンの実装方法には、プッシュとプルの2つがあります。プッシュメソッドでは、オブザーバーは監視対象オブジェクトをサブスクライブし、監視対象に注目すべき事象が発生した場合にオブザーバーに連絡して知らせます。これが、DOMイベントの挙動です。プルメソッドでは、監視対象は、オブザーバーの持つサブスクリプションリストに追加されます。定期的に、または指定されたときに、オブザーバーは、監視対象内で変更が発生したかどうかをチェックし、変更があった場合はその変更に反応して何らかの処理を実行します。多くのデスクトップソフトウェアで、アップデートはこのように動作します。アプリケーションは、起動時にアップデートがあるかどうかサーバーをチェックし、アップデートが見つかった場合は、アップデートのインストールを開始します。

プッシュオブザーバー

まず、プッシュメソッドを介したオブザーバーパターンのシンプルな例の構築について、ひと通り見てみましょう。ここではプルメソッドの例は紹介しませんが、それはほとんどのプルメソッドがサーバーとのやり取り（AJAXを使ってサーバーに照会するだけ）に使用されるからです。フロントエンドWeb開発におけるプルメソッドの使用は限定的です。

```
var Observable = function() {
    this.subscribers = [];
}

Observable.prototype = {
    subscribe: function(callback) {
        // Just add the callback to the subscribers list
        this.subscribers.push(callback);
```

```
    },
    unsubscribe: function(callback) {
        var i = 0,
            len = this.subscribers.length;

        // Iterate through the array and if the callback is
        // found, remove it from the list of subscribers.
        for (; i < len; i++) {
            if (this.subscribers[i] === callback) {
                this.subscribers.splice(i, 1);
                // Once we've found it, we don't need to
                // continue, so just return.
                return;
            }
        }
    },
    publish: function(data) {
        var i = 0,
            len = this.subscribers.length;

        // Iterate over the subscribers array and call each of
        // the callback functions.
        for (; i < len; i++) {
            this.subscribers[i](data);
        }
    }
};

// The observer is simply a function
var Observer = function (data) {
    console.log(data);
}

// Here's where it gets used.
observable = new Observable();
observable.subscribe(Observer);
observable.publish('We published!');
// 'We published!' will be logged in the console
observable.unsubscribe(Observer);
observable.publish('Another publish!');
// Nothing happens because there are no longer any subscribed observers
```

では、順を追って少しずつ見ていきましょう。まず、`Observable`コンストラクターを作成します。このコンストラクターは、サブスクライバー/オブザーバーを格納する空の配列を作成するだけです。次に、`subscribe`、`unsubscribe`、`publish`というプロトタイプ関数を作成します。これらの3つの関数は、すべてのタイプのプッシュ監視対象に必要です。もちろん、これらの関数の名前は変更できます。`subscribe`関数は、オブザーバー関数をサブスクライバーの配列に単に追加します。`unsubscribe`関数は、指定されたオブザーバーを検索し、リストから削除します。`publish`は、オブザーバーのリスト全体を反復処理し、実行します。

プロトタイプを介してではなくオブジェクトリテラルを使用して`Observable`オブジェクトを作成する場合、mixinsを使用して任意のオブジェクトを監視対象にできます。オブザーバーパターンとプルメソッドを一緒に使用する方法や、オブザーバーパターンに関する一般的な情報をもう少し詳しく知りたい場合には、筆者のJavaScriptブログで「JavaScript Design Patterns: Observer」の投稿をお読みください。

コマンド

本稿およびこのシリーズの最後を飾るパターンはコマンドパターンです。従来のオブジェクト指向プログラミングのコンテキストの中では、このパターンは、標準から大きく外れています。通常は、オブジェクトは何らかの名詞を表すものですが（よって「オブジェクト」と呼ばれる）、このコマンドパターンでは、オブジェクトは動詞を表します。オブジェクトの役目は、メソッドの呼び出しをカプセル化することです。コマンドパターンとは、メソッドとそのメソッドを呼び出したいオブジェクトを実装するオブジェクト間の抽象層にすぎません。これは、オブジェクト指向言語が従来型であるほど便利であり、ユーザーインターフェイスに最もよく使用されます。JavaScriptでは、コマンドオブジェクトは単なる関数であることもあり、この関数はコードの単純化に大きく貢献します。

アラーム時計のコマンドパターン化

ここでは、従来のコマンドパターンの使い方をわかりやすく説明します。関係のないコードと混同せずにコマンドパターンの動作を示すために、この例では非常にシンプルなコードを使用します。この例はアラーム時計です。

```
var EnableAlarm = function(alarm) {
    this.alarm = alarm;
}
EnableAlarm.prototype.execute = function () {
    this.alarm.enable();
}

var DisableAlarm = function(alarm) {
    this.alarm = alarm;
}
DisableAlarm.prototype.execute = function () {
    this.alarm.disable();
}

var SetAlarm = function(alarm) {
    this.alarm = alarm;
}
SetAlarm.prototype.execute = function () {
    this.alarm.set();
}
```

このコードでは、少なくとも、enable、disable、setという3つのメソッドを持つアラームオブジェクトが既に作成済みであると仮定しています。これらの3つのメソッドをカプセル化するために3つの異なるクラスを作成しました。各クラスは、この場合はexecuteメソッドのみを持つ特定のインターフェイスを採用しています。

次のコードにはButtonクラスがあり、ボタンの文字列ラベルとコマンドオブジェクトという2つの引数を受け取ります。このコマンドオブジェクトのexecuteメソッドは、ボタンが押されたときに呼び出されます。本質的には、単に、UI要素へのバインディングイベントに対して従来のオブジェクト指向のアプローチを採用しているだけです。

```
var alarms = [/* array of alarms */],
    i = 0,
    len = alarms.length;

for (; i < len; i++) {
    var enable_alarm = new EnableAlarm(alarms[i]),
        disable_alarm = new DisableAlarm(alarms[i]),
        set_alarm = new SetAlarm(alarms[i]);

    new Button('Enable', enable_alarm);
    new Button('Disable', disable_alarm);
    new Button('Set', set_alarm);
}
```

これはアラームのリストで、各アラームは制御用のボタンを必要としています。そこで、各アラーム用に3つのコマンドオブジェクトを作成し、各アラームの3つのボタンにパラメーターとして送信します。この例では、JavaScriptの柔軟性により、すべてのコマンドオブジェクトを数行のコードに短くまとめることができます。

```
var makeCommand = function( object, methodName ) {
    return {
        execute: function() {
            object[methodName]();
        }
    }
}

// This is how it's used now:
var alarms = [/* array of alarms */],
    i = 0, len = alarms.length;

for (; i < len; i++) {
    var enable_alarm = makeCommand(alarms[i], 'enable'),
        disable_alarm = makeCommand(alarms[i], 'disable'),
        set_alarm = makeCommand(alarms[i], 'set');

    new Button('enable', enable_alarm);
    new Button('disable', disable_alarm);
```

```
    new Button('set', set_alarm);
}
```

特定のタイプのオブジェクトではなく、executeメソッドを備えたオブジェクトリテラルを返します。重要なのは、どちらの場合も、同じインターフェイスが使用されるということです。それでもまだ、成果の割に作業が多すぎるようです。シンプルなコールバック関数を使用して書き換えると、次のようになります。

```
var alarms = [/* array of alarms */],
    i = 0, len = alarms.length;

for (; i < len; i++) {
    new Button('enable', alarm.enable);
    new Button('disable', alarm.disable);
    new Button('set', alarm.set);
}
```

ただし、問題が1つあります。これらのコールバック関数を実行したとき、コンテキストが失われ、thisがアラームを参照しなくなり、大きな障害となります。そこで、コンテキスト変数を取るButtonコンストラクターを作成するか、アラームメソッドのプロキシを作成するか（ほとんどの場合コマンドパターンではこの方法を取る）、メソッド内でのアラームを参照するselfやthatなどを使用できるようにクロージャを持つAlarmコードを書き直す必要があります。お気づきのように、既存のクラスに変更を加えずに済む唯一の方法は、プロキシまたはアラームとボタン間に別種の抽象層を使用するやり方だけです。コード行は増えますが、この方法は非常によい選択肢です。

問題の単純さ
この例における最も重要なポイントは、前にも述べましたが、これがコマンドパターンの仕組みを示すことを目的とした比較的シンプルな例であり、本来の性能や利便性を十分に表せてはいないということです。コマンドパターンは、もっとずっと便利に使用できます。上の例では、setメソッドは、アラームが設定する内容を把握できるよう、パラメーターをいくつか受け取ります。この場合、コマンドオブジェクトはパラメーターの特定に責任を負い、パラメーターをalarm.setに渡します。

他にコマンドパターンの使用が役立つ状況としては、alarmまたはその他のオブジェクトに対して複数のメソッドの呼び出しを行う場合などがあります。基本的に、execute関数が少しでも複雑になると、コマンドオブジェクトがより有用になります。

最後に、コマンドパターンが最も役に立つ状況は、executeやundoなどの関数が2つ以上含まれている場合です。この場合は、execute関数が実行した任意のアクションを取り消すためのundo関数を作成できます。この関数を活用するには、ボタンがコマンドオブジェクトに対してexecuteメソッドを呼び出したときに、それ以前にexecuteメソッドを呼び出していた他のコマンドオブジェクトのスタックに、そのコマンドオブジェクトを積み重ねます。ユーザーがCtrl＋Zキーを押して直前のアクションを取り消そうとした場合、スタックからコマンドオブジェクトが引き出され、そのundoメソッドが呼び出されます。このように使用すると、コマンドパターンは上の例よりもずっと便利でパワフルです。

コマンドの使用
コマンドパターンがこの記事の締めくくりです。コマンドパターンについてもう少し詳しく知りたい場合、またはJavaScriptにおけるコマンドオブジェクトの作成の別の方法について具体的に知りたい場合は、筆者のJavaScriptブログの「JavaScript Design Patterns: Command」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を終了します。このシリーズのその他のデザインパターンについては、第1部のシングルトン、コンポジット、ファサードの各パターンおよび第2部のアダプター、デコレーター、ファクトリの各パターンに関する記事をお読みください。

このシリーズで触れなかったパターンもいくつかあり、例えば責務（responsibility）パターンのブリッジやチェインなどですが、それらに関しては「JavaScript Design Patterns on Joe Zim's JavaScript Blog」に記事があります。Ross Harmes氏およびDustin Diaz氏による「Pro JavaScript Design Patterns」（筆者がこのデザインパターンに関するシリーズ記事を執筆するきっかけとなった書籍）など、パターンに関するかなり詳しい書籍もいくつか出版されています。皆さんがこのシリーズに費やした時間よりも大きな成果を得られていますように。それでは皆さん、ハッピーコーディング！

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
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
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