Flyweight pattern

Author : Rhio.kim
Date : 2008.09.04
Blog : blog.ecmas4.com
Mail : Rhio.kim@gmail.com
Version : 0.9.2 / Update : 2008.09.17

일반적으로 Flyweight pattern 은 효율적으로 유사한 오브젝트들의 공유를 제공하여 메모리 사용을 최소화하고 고성능을 위한 소프트웨어 디자인 패턴입니다.

Ajax/Rich UI 개발의 경우 수 많은 DOM 오브젝트에 비슷한 기능을 부여하거나 수행하는 경우가 많다. 이런 경우 각 DOM 오브젝트에 메소드를 부여하거나 상속에 의한 혹은 prototype chain 에 의한 레퍼런스를 통하여 기능을 수행하도록 처리합니다.

하지만 경우에 따라서는 매우 복잡한 DOM 오브젝트 제어가 필요하게 됩니다. 그럴 경우에 위의 패턴을 적용하면 성능 향상에 뛰어남을 보입니다.

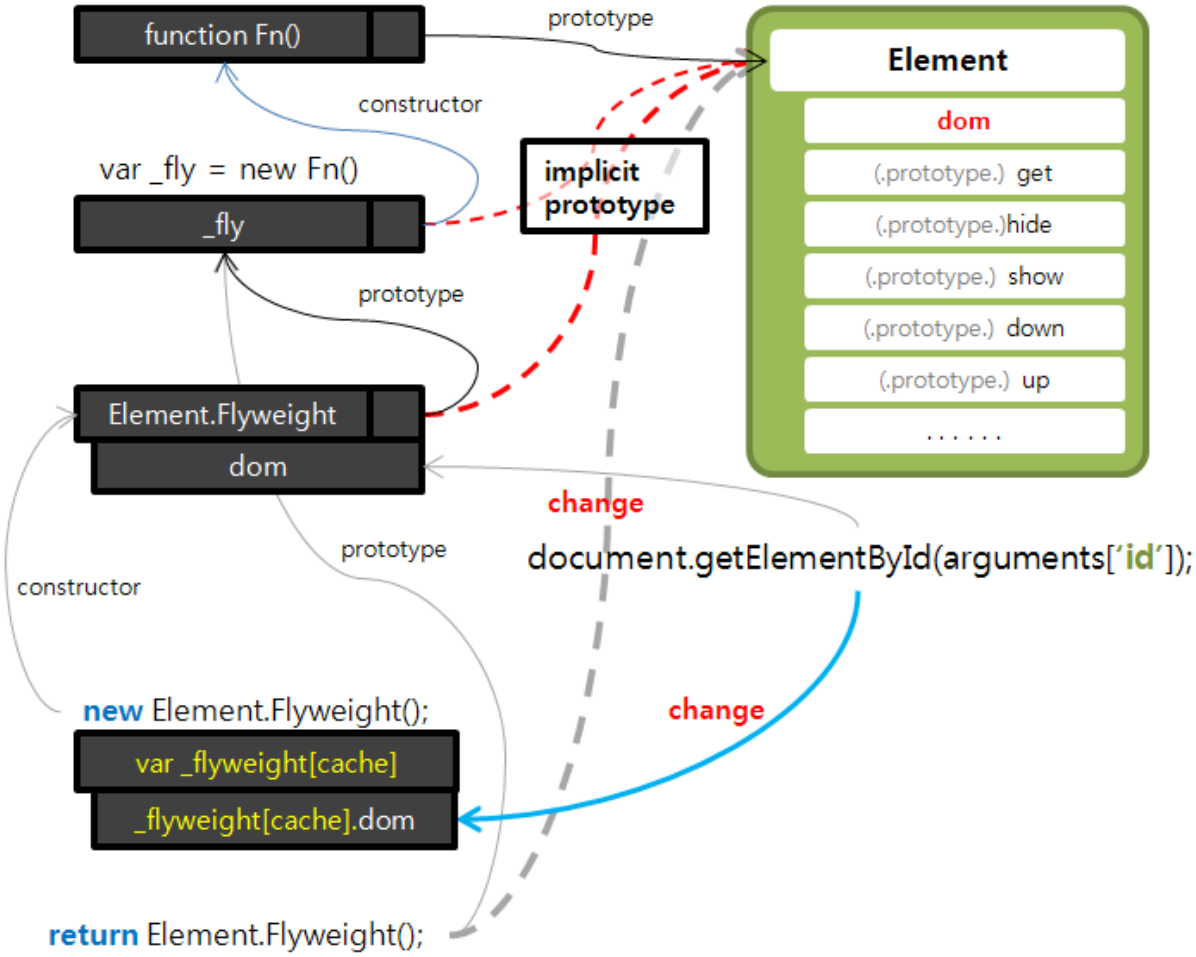
실제로 jQuery 와 ExtJS 에서는 이 패턴을 적용하고 있습니다. 반대로 prototype.js 의 Element 의 경우에는 $함수를 통하여 얻어온 DOM 오브젝트에 Element 가 갖는 메소드 즉 down() up(), next(), previous() 등등 수 많은 메소드를 prototype 에 의해서 DOM 오브젝트가 Element.down(), Element.up() 등의 메소드를 레퍼런스하게 됩니다.

IE, Safari 의 경우에는 __proto__ chain 메카니즘이 존재하지 않기 때문에 해당 엘리먼트에 Element 객체가 갖는 모든 메소드를 부여하는 과정을 한번 더 거치게 됩니다. <관련기사:옷장수님 블로그#Element-메소드-상속>

이 처럼 브라우저에 따라서는 수많은 오브젝트에 확장 메소드를 부여하여 수행하려고 한다면 메모리 사용과 성능면에서 뒤 떨어질 수밖에 없습니다.

Flyweight 패턴의 경우에는 성능 향상과 메모리 사용면에서 탁월하다는 것을 느끼실 수 있을 것입니다.

간단한 그림을 통해서 Pattern 을 이해해보도록 하겠습니다.



좌측은 코드의 주요 수행 과정이고 우측의 Element 는 DOM 오브젝트의 기본적인 제어를 위한 공유 오브젝트입니다. dom 프로퍼티는 Element 객체 내에서 내부적으로 제어를 하게 됩니다. 예를 들어 hide 프로퍼티는 dom.style.display = 'none'; 가 수행되는 기능을 가지고 있게 됩니다.

좀더 섬세한 예제로 Prototype.js 의 $ 함수를 예로 들어보겠습니다.
$('rhio') 라는 명령문을 수행하였다면 document.getElementById('rhio')를 내부적으로 수행하여 순수한 DOM 오브젝트가 반환되면 내부적으로 Element.extend 가 수행되어 DOM 오브젝트에 $('rhio').update('본 내용이 innerHTML 됩니다.'); 라는 명령을 수행할 수 있게 됩니다.

즉 $ 함수를 수행할 때마다 얻어온 DOM 오브젝트에 extend 하는 과정이 수행됩니다. (Firefox 의 경우 __proto__ chain 에 의해서 extend 과정없이 레퍼런스가 가능하기는 합니다.)

[부연설명] "prototype 1.6.0.2 기준으로 2543line 에 보면 $함수를 호출할 때 자동적으로 수행되는 Element.extend 수행 시

```
if (Prototype.BrowserFeatures.SpecificElementExtensions)
    return Prototype.K;
```

명령 구문의 Prototype.BrowserFeatures.SpecificElementExtensions 에 의해서 __proto__ 메카니즘을 제공하는 브라우저인지를 체크되며 Mozilla 브라우저의 경우에는 존재하기 때문에 Prototype.K 를 반환됩니다. " 각설하고...

반면 위의 Flyweight Pattern 의 경우에 순수한 DOM 은 Element.dom 프로퍼티로 레퍼런스되고 그 레퍼런스는 Element 의 공유 메소드들을 통해서 제어되게 됩니다.

```
var Fn = function() { };
Fn.prototype = Element.prototype;
var _fly = new Fn();

Element.Flyweight = function(dom) {
    this.dom = dom;
};
Element.Flyweight.prototype = _fly;
Element.Flyeight.prototype.isFlyweight = true;
var _flyweights = { };

Element.fly = function(el) {
  el = document.getElementById(el);

  if(!el) {
      return null;
  }

  if(! _flyweights['cache']) {
       _flyweights['cache'] = new Element.Flyweight();
  }
  _flyweights['cache'].dom = el;
  return _flyweights['cache'];
};
```

위의 그림과 소스를 번갈아 가면서 보시면 좀더 이해가 빠를 수 있습니다. 설명은 중요한 부분만 정리해 설명드리고 소스는 ExtJS 의 Element.fly 부분을 인용하여 변경한 소스입니다.
c.f) ExtJS > Element.js 2986 line ~ 3019 line

Element.Flyweight.prototype 는 implicit prototype 이 Element.prototype 을 향하고 Element.prototype 이 갖는 메소드들(get, hide, show 등)은 Element 에 의해서 공유되게 됩니다.
Element.Flyweight function 은 new 키워드를 통해 인스턴스화 될 때 Element 의 공유 프로퍼티에 접근할 수 있게 됩니다. 여기까지가 Flyweight Pattern 의 기본 구조라 할 수 있고 ExtJS 의 경우 경우에 따라 좀더 개선된 방식까지 제공하고 있습니다.

ExtJS 의 Framework 이 로드되면서 생성된 변수 **_flyweights** 에 Element.prototype 을 참조하는 function 오브젝트 즉 Element.fly( )를 최초 호출되면 Element.Flyweight( )는 인스턴스화 되어 이를 저장해두고 다음 호출부터는 _flyweights 의 저장된 객체에 dom 메소드만 변경하여 반환하게 됩니다.

일반적으로 DOM 오브젝트를 JavaScript 로 제어하려고 할 때 일관된 메소드를 제공하는게 프레임웍의 기본 방향인 것 같습니다. 하지만 경우에 따라서는(거의 없겠지만) Image, Script 의 DOM 오브젝트에는 일관된 메소드보다 확장된 메소드를 제공하고 싶은 경우거나 반대로 불필요한 메소드를 제공하지 하지 않아야 하는 경우라면 위와 같은 패턴은 비효율적일 수 있겠습니다.

참고 :
Pro JavaScript Design Patterns [Ross Harmes and Dustin Diaz] 179page ~ 195page
http://en.wikipedia.org/wiki/Flyweight_pattern
http://extjs.com/forum/showthread.php?p=140648