Материалы занятия

Курс: Разработка интерфейса на JavaScript

Дисциплина: Основы JavaScript

Тема занятия №38: Создание своих плагинов в jQuery

Приступая к работе

Сперва создаём новое свойство-функцию для объекта jQuery, где именем нового свойства будет имя нашего плагина:

```
jQuery.fn.myPlugin = function() {

  // Тут пишем функционал нашего плагина

};
```

Но постойте, где-же привычный нам значок доллара, который мы все хорошо знаем? Он всё ещё здесь, а чтобы он не конфликтовал с другими библиотеками, которые тоже могут использовать символ доллара, рекомендуется «обернуть» объект jQuery в непосредственно выполняемую функцию-выражение (IIFE, Immediately Invoked Function Expression), которое связывает объект jQuery с символом "$", чтобы он не был переопределён другой библиотекой во время выполнения.

```
(function( $ ) {
  $.fn.myPlugin = function() {

      // Тут пишем функционал нашего плагина

  };
})(jQuery);
```

Так лучше. Теперь внутри этого замыкания (closure), мы можем использовать знак доллара как нам заблагорассудится.

Контекст

Теперь у нас есть оболочка, внутри которой мы можем начать писать код плагина. Но прежде, чем мы начнём, я хотел бы сказать несколько слов о контексте. В непосредственной области видимости функции нашего плагина ключевое слово «this» ссылается на объект jQuery, для которого был вызван этот плагин.

И тут часто ошибаются, полагая, что в других вызовах, где jQuery принимает callback-функцию, «this» указывает на элемент DOM-дерева. Что, в свою очередь, приводит к тому, что разработчики дополнительно оборачивают «this» в функцию jQuery.

```
(function( $ ){

  $.fn.myPlugin = function() {

    // нет необходимости писать $(this), так как "this" - это уже объект jQuery
    // выражение $(this) будет эквивалентно $($('#element'));

    this.fadeIn('normal', function(){

      // тут "this" - это элемент дерева DOM

    });

  };
})( jQuery );
```

```
$('#element').myPlugin();
```

Основы

Теперь, когда мы понимаем, как работать с контекстом, напишем плагин jQuery, который выполняет полезную работу.

```
(function( $ ){

  $.fn.maxHeight = function() {

    var max = 0;

    this.each(function() {
      max = Math.max( max, $(this).height() );
    });

    return max;
  };
})( jQuery );
```

```
var tallest = $('div').maxHeight(); // Возвращает высоту самого высокого div-а
```

Это простой плагин, который, используя .height(), возвращает нам высоту самого высокого div-а на странице.

Поддерживаем возможность цепочек вызовов

Предыдущий пример рассчитывает и возвращает целочисленное значение наиболее высокого div-а на странице. Обычно, плагин модифицирует набор элементов дерева DOM, и передает их дальше, следующему методу в цепочке вызовов. В этом заключается красота jQuery и одна из причин его популярности. Итак, чтобы ваш плагин поддерживал цепочки вызовов, убедитесь в том, что ваш плагин возвращает this.

```
(function( $ ){

  $.fn.lockDimensions = function( type ) {

    return this.each(function() {

      var $this = $(this);

      if ( !type || type == 'width' ) {
        $this.width( $this.width() );
      }

      if ( !type || type == 'height' ) {
        $this.height( $this.height() );
      }

    });

  };
})( jQuery );
```

```
$('div').lockDimensions('width').css('color', 'red');
```

Так как плагин возвращает this в своей непосредственной области видимости, следовательно он поддерживает цепочки вызовов, и коллекция jQuery может продолжать обрабатываться методами jQuery, например, такими как .css. И, если ваш плагин не должен возвращать никакого рассчитанного значения, вы должны всегда возвращать this в непосредственной области видимости функции плагина. Аргументы, которые передаются в плагин при вызове, передаются в непосредственную область видимости функции плагина. Так, в предыдущем примере, строка 'width' является значением параметра «type» для функции плагина.

Настройки и умолчания

Для более сложных и настраиваемых плагинов, предоставляющих большое количество возможностей настройки лучше иметь настройки по-умолчанию, которые расширяются (с помощью $.extend) во время вызова плагина.

Так вместо вызова плагина с большим количеством параметром, вы можете вызвать его с одним параметром, являющимся объектным литералом настроек, которые вы хотите расширить. Например, вы можете сделать так:

```
(function( $ ){

  $.fn.tooltip = function( options ) {

    // Создаём настройки по-умолчанию, расширяя их с помощью параметров
    var settings = $.extend( {
      'location'         : 'top',
      'background-color' : 'blue'
    }, options);

    return this.each(function() {

      // Тут пишем код плагина tooltip

    });

  };
})( jQuery );
```

```
$('div').tooltip({
  'location' : 'left'
});
```

В этом примере после вызова плагина tooltip с указанными параметрами, значение параметра местоположения ('location') переопределяется значением 'left', в то время, когда значение параметра 'background-color' остаётся равным 'blue'. И в итоге объект settings содержит следующие значения:

```
{
  'location'         : 'left',
  'background-color' : 'blue'
}
```

Это хороший способ создавать гибко-настраиваемые плагины без необходимости определять каждый из доступных параметров настройки.

Определение пространства имён

Корректное определение пространства имён для плагина очень важно и обеспечивает достаточно низкую вероятность переопределения другим плагином или кодом, выполняющимся на той-же странице. Вдобавок определение пространства имён упрощает разработку, так как упрощается отслеживание нужных методов, событий и данных.

Методы плагина

При любых обстоятельствах один плагин должен определять не более одного пространства имён для объекта jQuery.fn.

```
(function( $ ){

  $.fn.tooltip = function( options ) {
    // НЕ НАДО
  };
  $.fn.tooltipShow = function( ) {
    // ТАК
  };
  $.fn.tooltipHide = function( ) {
    // ДЕЛАТЬ
  };
  $.fn.tooltipUpdate = function( content ) {
    // !!!
  };

})( jQuery );
```

Подобная практика не приветствуется, так как она загрязняет пространство имён $.fn

Чтобы избежать этого, объедините все методы вашего плагина в один объектный литерал и вызывайте их, передавая имя метода в виде строки.

```
(function( $ ){

  var methods = {
    init : function( options ) {
      // А ВОТ ЭТОТ
    },
    show : function( ) {
      // ПОДХОД
    },
    hide : function( ) {
      // ПРАВИЛЬНЫЙ
    },
    update : function( content ) {
      // !!!
    }
  };

  $.fn.tooltip = function( method ) {

    // логика вызова метода
    if ( methods[method] ) {
      return methods[ method ].apply( this, Array.prototype.slice.call( arguments, 1 ));
    } else if ( typeof method === 'object' || ! method ) {
      return methods.init.apply( this, arguments );
    } else {
      $.error( 'Метод с именем ' +  method + ' не существует для jQuery.tooltip' );
    }
  };

})( jQuery );

// вызывает метод init
$('div').tooltip();

// вызывает метод init
$('div').tooltip({
  foo : 'bar'
});
```

```
// вызывает метод hide
$('div').tooltip('hide');
```

```
// вызывает метод update
$('div').tooltip('update', 'Теперь тут новое содержимое');
```

Этот тип архитектуры плагинов позволяет вам инкапсулировать все ваши методы в родительском по отношению к плагину замыкании (closure), и

вызывать их, сперва передавая имя метода как строку, а затем передавая любые дополнительные параметры для этого метода. Этот подход к инкапсуляции методов является стандартом в сообществе разработчиков jQuery-плагинов и применяется в бесчисленном множестве плагинов и виджетов в jQueryUI.

События

Малоизвестная особенность метода bind заключается в том, что он позволяет определять пространства имён для связанных событий. Если ваш плагин связывает некую функциональность с каким-нибудь событием, то хорошим тоном будет задать пространство имён для этого события. И если позднее вам потребуется отвязать эту функциональность от события, то вы сможете это сделать, не затрагивая функциональность, которая может быть прикреплена к этому-же типу события.

Вы можете определить пространство имён для ваших событий, просто добавив точку и название пространства имён к названию типа события, с которым вы связываетесь.

```
(function( $ ){

  var methods = {
     init : function( options ) {

       return this.each(function(){
         $(window).bind('resize.tooltip', methods.reposition);
       });

     },
     destroy : function( ) {

       return this.each(function(){
         $(window).unbind('.tooltip');
       })

     },
     reposition : function( ) {
       // ...
     },
     show : function( ) {
       // ...
     },
     hide : function( ) {
       // ...
     },
     update : function( content ) {
       // ...
     }
  };

  $.fn.tooltip = function( method ) {

    if ( methods[method] ) {
      return methods[method].apply( this, Array.prototype.slice.call( arguments, 1 ));
    } else if ( typeof method === 'object' || ! method ) {
      return methods.init.apply( this, arguments );
    } else {
      $.error( 'Метод с именем ' +  method + ' не существует для jQuery.tooltip' );
    }

  };

})( jQuery );
```

```
$('#fun').tooltip();
// Некоторое время спустя...
$('#fun').tooltip('destroy');
```

В этом примере, когда плагин tooltip проинициализировался с помощью метода init, он связывает метод reposition с событием resize (изменение размеров) окна, с указанием пространства имён 'tooltip'. Позднее, когда разработчик намерен разрушить объект tooltip, он может отвязать все прикреплённые к плагину обработчики путём указания соответствующего пространства имён. В данном случае — 'tooltip' для метода unbind. Это позволяет безопасно отвязать обработчики от событий без риска случайно отвязать событие, связанное с обработчиком вне данного плагина.

Данные

Зачастую во время разработки плагинов, вы можете столкнуться с необходимостью сохранения состояний или проверки, был-ли плагин уже проинициализирован для указанного элемента. Использование метода data из jQuery — это хороший способ отслеживать состояние переменных для каждого элемента. Однако вместо того, чтобы отслеживать множество отдельных вызовов data с разными именами, рекомендуется использовать один объектный литерал, который будет объединять все ваши переменные под одной крышей, и вы будете обращаться к этому объекту через одно пространство имён.

Использование data позволяет отслеживать состояние переменных между вызовами вашего плагина. Определение пространства имён для data в одном объектном литерале, обеспечивает, как простой централизованный доступ к свойствам плагина, так и сокращает пространство имён data, что позволяет просто удалять ненужные данные по мере необходимости.

Заключение и полезные советы

Создание плагинов для jQuery позволяет извлечь максимальную пользу из этой библиотеки и абстрагировать свои наиболее удачные решения и часто используемые функции в повторно используемый код, который может сохранить вам время и сделает процесс разработки более эффективным. Ниже представлена краткая выдержка того, что следует помнить во время разработки своего jQuery плагина:

- Всегда оборачивайте свой плагин в конструкцию:

```
(function( $ ){
          /* тут пишем код плагина */
 })( jQuery );
```

- Примечание переводчика: в оригинальной статье эта синтаксическая конструкция названа замыканием (closure), но это не замыкание, а непосредственно вызываемая функция (IIFE).
- В непосредственной области выполнения функции вашего плагина не оборачивайте this в ненужные синтаксические конструкции.
- Если только вы не возвращаете из функции плагина какое-то определенное значение, всегда возвращайте ссылку на this для поддержки цепочек вызовов.

- При необходимости передачи длинного списка параметров, передайте настройки вашего плагина в виде объектного литерала, значения которого будут распространятся на значения по умолчанию для параметров вашего плагина.
- Для одного плагина определяйте не более одного пространства имён jQuery.fn.
- Всегда определяйте пространство имён для ваших методов, событий и данных.