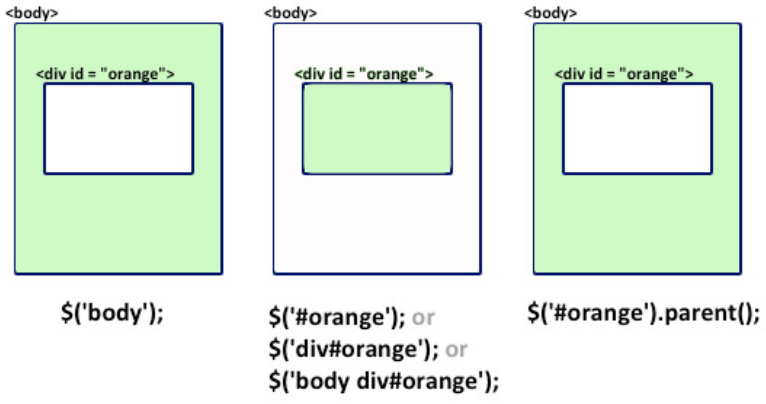
Basic steps for creating plugins with jQuery

In the article below, we will introduce and guide you a few basic steps to create a plugin using jQuery. Whether you are a beginner to learn about web development or have been exposed to Javascript for a long time, jQuery is a great framework and not to be missed ...

TipsMake.com - In the article below, we will introduce and guide you a few basic steps to create plugins with jQuery . Whether you are a beginner to learn about web development or have been exposed to **JavaScript** for a long time, jQuery is a great and not to be framework framework, especially for those who come to jQuery from available Javascript platform.

We can hardly understand jQuery without understanding CSS Selectors:

In jQuery, the **CSS selector** code is started by the main **jQuery** object, this is just the name of the **function** , and it comes with the parameter - it's our **CSS Selector** . And essentially, the **Selector** works similar to the **CSS selectors** commonly found in **style.css** files. Many people only know **ID selector (#)** and **class selector (** but they are only certain objects belonging to the **selectors)** :



If you can understand the details in the chart above, then the rest of **jQuery** is not difficult to grasp. And once you know the most basic principles of the jQuery plugin process, almost no one wants to return to **Javascript** . Here are some common questions when we embark on the process of writing jQuery plugins:

1. What are **CSS selectors** ?
2. What is the difference when using **$ .myfunction** and **$ .fn.myfunction** ?
3. What does the symbol **$** mean?

4. Main function and usage of function **jQuery.extend** ?
5. How to initialize **jQuery** plugin and move to function parameter?
6. How to pass the **default** value and initialize the **override** parameter?

Objects in JavaScript:

As we all know, JavaScript is an object-oriented programming language, and it is not suitable to describe or demonstrate jQuery's features to people who have never studied the concept of objects. in JavaScript. The basic feature of object design in JavaScript is that when a user declares or initializes a function, that function will be a class in the default mode, and then will be used as an object.

Besides, we can use the new operator to create a copy of the same class. And it only happens when the user replaces with the familiar class keyword, JavaScript will prefer the function keyword. Note that the class in JavaScript can be used as a class that has been inherited from another class (by implementing **prototype inheritance**) or built exactly as a function from within the source code. In short, in JavaScript, all are objects.

On the other hand, **function closures are** often used with events in **jQuery** . A **function closure is** typically a type of function without any customization or any name, but simply an object of that function.

Main objects in jQuery:

In essence, jQuery's main objects are all defined with the function keyword as an object with an identifier named $, and they are equivalent to a **global function** , such as window. $. For example, in **JavaScript** when we extend a certain window object (must be available in the browser from the design stage), or in other words, assign an additional **member function** to (.) To the **window** object, and also means that we can call that object with **window.myobj ();** or simply **myobj ();** . Because every function attached to the **object window** can be called from the global scope in the entire script. From within, jQuery objects will be created as follows:

```
var jQuery = window.jQuery = window. $ = function (selector, context)
{
// .
// other internal initialization code goes here
};
```

This exact declaration algorithm allows users to easily jump to main jQuery objects via jQuery names or $ symbols. You need to know that **jQuery** , **window.jQuery** , **window. $** Or simply $ can all be used interchangeably, because the declaration process is at the beginning of the code set, and it is related to the whole object.

One more point to note is that the parameters of selector and context of jQuery function as a normal object.**The selector** is usually a **string** that selects a certain number of elements from the **DOM - Document Object Model** . Or it can be **this** (**self reference**) object. The **jQuery selector** parameter can accept many of the same values ??in case the user wants to use it during **CSS** initialization. For example, please refer to the process of calling objects in **jQuery** as follows:

```
// Select all elements of class: "someClass" and
// apply a red border to all of them
jQuery(".someClass").css("border", "1px solid red");
```

// Ch?n m?t ph?n t? v?i id: m?tId và chèn ???c ??ng html vào nó
jQuery ("# ??someId"). html (" **So Bold!** ");

```
// Exactly the same as above, but using $
$("#someId").html(" So Bold! ");
```

Very short and simple, you no longer have to write the document.getElementById **function** which is very confusing. With just one command line, jQuery will select all the required components with a full **DOM** review and apply 'effect'. Another very interesting point of jQuery is the compatibility mode for all browsers.

The bottom line of jQuery plugins:

Here, we'll start with a simple example of a fairly popular **jQuery plugin** . Specifically, when we first learn about a certain programming language or **framework** , we all have to know the key point of the problem. In the time when jQuery has not yet appeared, **JavaScript** programmers often have the habit of executing code, important **functions** in **window.onload function** as illustrated below:

```
// Override the onload event
window.onload = function()
{
// the page finished loading, do something here.
}
```

In fact, the above code only performs the task of overwriting the tag's **onload event** in **HTML** , or understanding that our code is only executed when the website finishes loading. In many cases, this is very difficult because there are sites that need more time to download content, or the download process is interrupted due to different data browsing structures.

Here, we don't want to compile and execute any **JavaScript** code on the page being loaded. JQuery's internal architecture makes the most of the advantage of event **window.onload** , but it also does a full check of whether the DOM has been loaded, because the process is very important.

However, the above factors are not enough for jQuery to 'know' whether the website has been fully loaded, but we must make sure that the **DOM** has been 'built' fully - done by 'listening'. **DOMContentLoaded** signal on almost all browsers. But it is fortunate because we do not need to perform the test as above, jQuery has taken care of all this.

To support users, jQuery has provided a new method called **ready** that we can call from jQuery's main object. Specifically, when writing the jQuery plugin, you use this **function ready** to check if the system is ready to execute that code. Note that this is not the main plugin code but the key point of the plugin. Or understand that this is the jQuery version of the **window.onload function:**

For example, here we will write a plugin called **Shuffle** , and this plugin has 2 separate functions to initialize and execute the basic code, which looks like this:

As usual, any code can be customized based on user needs. The syntax used below is considered popular, because it is applied by most **jQuery** programmers, but it is easy to confuse especially for beginners to learn about both **JavaScript** and **jQuery** :

This is a very common case in intuition, a dot is often replaced by the code to be executed. Specifically here, you need to refer to the definition of anonymous function (also called **closure function**) with the parameter is $. And this function is enclosed in parentheses, why? Function in this example is essentially an anonymous function that does not refer to itself, but because of the 'scope' of parentheses, JavaScript will allow the user to refer to any **anonymous function** created.

Besides, we can add a dot after the initialization of the function is enclosed in parentheses, and call a **member member** supported by this function. Next, add a dot at the end of that clause and call another one **member function** that can be executed based on the format of the object returned by the previous function. This function of chaining function of **JavaScript is** quite common and is applied in many other programming languages, such as **Perl** . It is a kind of feature of scripting language because it makes the part of the user code more concise, manageable and more intuitive.

Therefore, by placing the **anonymous** function in parentheses, we can easily refer to the memory part of that function without having to pass the name - which is actually impossible because the function has no name.

On the other hand, you can call that **function** using the clause used to create the function. And that is also our way here. The functions without names are defined, enclosed in parentheses, and will be called immediately. This is just a simple example of how to use the closure function that we can easily recognize in many parts of advanced jQuery and **JavaScript** source code.

What are the reasons to do so? The first is to minimize the length of the code, followed by hiding all of the parameters that are started with $ from the **global global scope** model. Besides, for those cases where we use many different frameworks, using the $ symbol for the function object in the model of the whole program is very confusing.

But in reality that is entirely dependent on the circumstances and the main choice of the developer. Specifically, in the next part of the tutorial, we do not use any external framework or additional code, the general syntax can be a little bit confusing and difficult to understand, but conflict rates and errors will arise. guaranteed at the lowest level. The general basic syntax is as follows:

With **doSomething** and **doSomethingElse** simply are function objects that were previously defined and initialized, but if you do so, we also lose the ability to combine, string jQuery plugins with existing **API** functions. available. Finally, if you want the plugin to be chaining capable, you must use the **sell-calling** function as explained above. If you combine the plugin's chaining function with the code of many other projects, you only need to pair the object and make the necessary variable to the **$** object. or **$ .fn** . Note that in jQuery, some characters like $ sign or some special cases such as **window. $** Can be used interchangeably, and are also

used to refer to exactly any object. that's saved in memory. In the specific documentation of jQuery, it is shown that we should use jQuery object instead of $ sign. Since the $ symbol must be hidden, it should only be used within the entire jQuery structure, and absolutely not to 'expose' any $ parameter to the **plugin implementators** .

---

jQuery Plugin Design Pattern `A`

Next, we will refer to the internal code section of the plugin optionally with the **anonymous self-referencing** function and the function to retrieve jQuery objects as parameters. But here, we will be confused with this keyword referenced in this pattern and other locations in the entire code. Specifically, we created some local variables with the name vari - stored in the main jQuery object ($ .vari) and the **jQuery fn** object **($ .fn.vari)** . They can also be **function functions** , but for simplicity and comprehension, we only use the basic variable part. You will easily see these values ??displayed along with the **alert - alert function** in the code below.

First is the un-shortened pattern:

```
 (function($)
 {
 $.vari = "$.vari";
 $.fn.vari = "$.fn.vari";

// $ .fn is the object we add our custom functions vào
$ .fn.DoSomethingLocal = function ()
{
return this.each (function ()
{
alert (this.vari); // would output `undefined`
alert ($ (this) .vari); // would output `$ .fn.vari`
});
};
}) (jQuery);

// $ is the main jQuery object, we can attach a global function to it
$ .DoSomethingGlobal = function ()
{
alert ("Do Something Globally, where` this.vari` = "+ this.vari);
};

$(document).ready(function()
{
$("div").DoSomethingLocal();
$.DoSomethingGlobal();
});
```

And specific details:

```
// plugin-name.js - define your plugin implementation pattern
(function($) // The $ here signifies a parameter name
// As you can see from below, (jQuery) is
// immediately passed as the $ param
{
$.vari = "$.vari";
$.fn.vari = "$.fn.vari";

// 1.) Add a custom interface `DoSomethingLocal`
// Which will modify all elements selected!
// If b?n có m?t máy ?nh máy ?nh, think v? này
// a member function of the main jQuery class
$ .fn.DoSomethingLocal = function ()
{
// return the object back to the chained call flow
return this.each (function () // This is the main processor
// function that executes on
// each element selected
// (eg: jQuery ("div"))
{
// this ~ points to a DOM element
// $ (this) ~ link to a jQuery object

// Here, the `this` keyword is a self-refence to the
// selected object `này.vari` là` undefined` do
// it links to DOM elements selected. So, we can do
// something like: var borderStyle = this.style.border;
// While $ (this) .vari, or jQuery (this) .vari address
// to `$ .fn.vari`

// You would use the object to perform ((this))
// nào ???c ch?n thay ??i ?? các ph?n ???c ch?n
// $ (this) is simply a reference to the jQuery object
// of the selected elements
alert (this.vari); // would output `undefined`
alert ($ (this) .vari); // would output `$ .fn.vari`
});
};
}) (jQuery); // pass the jQuery object to this function

// 2.) Or we can add a interface custom to the jQuery global
// object. In case, it makes no sense to enumerate
// qua các ??i t??ng v?i `m?i ký t? c?a vì này
// will theoretically work in `global` scope. If you are
// a professional software engineer, think about this
// as a [static function]
$ .DoSomethingGlobal = function ()
{
```

```
// this will output this.vari = $ .vari
alert ("Do Something Globally, where` this.vari` = "+ this.vari);
};

// index.html - test the plugin
$(document).ready(function()
{
$("div").DoSomethingLocal();
$.DoSomethingGlobal();
});
```

In fact, there are two different types of interfaces that we can assign to jQuery objects that have been initialized by the framework. Please refer to the example above, where we have added 2 **functions** , that is **DoSomethingLocal** and **DoSomethingGlobal** .

1. **DoSomethingLocal** is defined as **$ .fn.DoSomethingLocal** . Users can assign custom functions to the jQuery.fn object (or $ .fn) required by the plugin implementation process. Any function assigned to the $ .fn object works on the custom part of a selected object or component. And this is also the meaning of the $ .fn syntax.
2. **DoSomethingGlobal** is applied directly to the global jQuery object in the form of **$ .DoSomethingGlobal** . A function assigned to the jQuery framework in such a way will not work on the selected component, but can function in global scope and contain any implementation capabilities.

jQuery Plugin Design Pattern `B`

In fact, many people like to put the plugin code in **anonymous function** , and call it by appending parentheses at the end of the clause, then passing that jQuery object like **jQuery Plugin Design Pattern `A ` ** on, but do we really need to do so? Is not. Refer to another way to create **jQuery plugins** with a simpler and easier structure.

The pattern below must be used in case we do not perform the **chainability** function, or in other words the **function** of the **plugin** will not return the jQuery object.

```
 // Plugin base object
 $.gShuffle = function()
 {

}

// Initializes plugin
$ .gShuffle.initialize = function ()
{

}

// Runs the plugin
$ .gShuffle.run = function ()
{

};
```

Hopefully the above information will help you better understand the basic steps of creating **jQuery** plugins. Good luck!

You finished reading the article "**Basic steps for creating plugins with jQuery**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.
