JS 预解析机制与定义函数的三种方式

一、定义函数的三种方式

1、 函数声明（用的最多，推荐优先使用）

```
<script>
//函数声明的语法
function fn() {
    console.log("这是函数声明的方式");
}
fn(); //直接调用函数名
</script>
```

2、 函数表达式（也叫匿名函数）

```
<script>
var fn = function([参数列表]) {
    console.log("这是一个匿名函数");
}
/*
顾名思义，因为这种写法是把函数赋给了一个变量，函数并没有真正的名字
因此就叫匿名函数
*/
fn(); //调用匿名函数，使用变量名称加();
</script>
```

3、 自执行函数

（1）第一种写法

```
<script>
/*
顾名思义：自己执行自己，并且在声明的同时就调用自己，只能调用一次
*/
// 有两种写法，这是第一种：
(function() {
    console.log("这是一个自执行函数");
})();
</script>
```

这样写的原理是：他是由函数表达式演变过来的，var fn = function(){} fn(); 把 fn 替换成：function(){}，就成了: function(){}(); 为了保持整体性，在 function 加上一个(),所以就变成了 (function([这里是形参]){})([这里是实参])

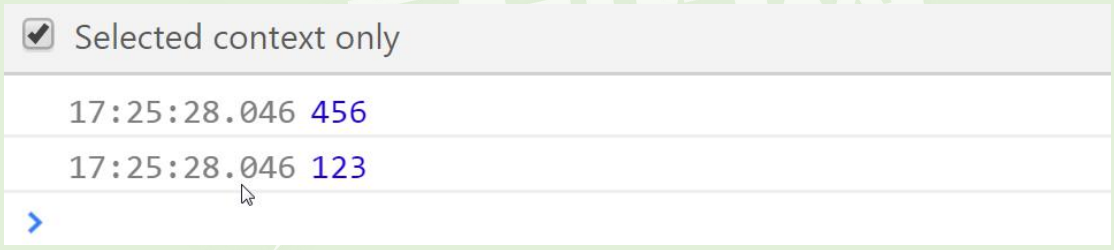
（2）第二种写法

```
<script>
//第二种写法：
(function() {
    console.log("这是一个自执行函数");
}())
</script>
```

值得注意的是，用这种写法会有一个小漏洞，看个例子：

```
<script>
var fn = function() {
        console.log(123);
}
(function() {
        console.log(456);
}())
</script>
```

输出结果是：



这里匿名函数 fn 明明没调用，为什么会输出？看下面简化后的例子：

```
<script>
var fn = function() {
    console.log(123);
}(function() {
    console.log(456);
}())
</script>
```

可以很明显的看到(function(){console.log(456);}())这个自执行函数调用输出了 456，同时，这个自执行函数跟在匿名函数的后面，JS 解析的时候把这个匿名函数也解析成为了自执行函数，所以就输出了 123，解决的办法很简单，在第一个匿名函数的后面加一个分号“;”或者直接采取第一种写法就能避免这种问题

二、JS 预解析机制

1、预解析过程：

（1）把变量的声明提升到当前作用域的最前面，只会提升声明，不会提升赋值。

（2）把函数的声明提升到当前作用域的最前面，只会提升声明，不会提升调用。

（3）先提升 var，在提升 function

因此，我们在定义函数时，绝大多数都使用第一种方式(函数声明的方式)

当匿名函数多用于定时器里面和注册事件的时候，比如

```
<script>
btn.onclick = function() {
    console.log("这是一个匿名函数");
}
</script>
```

2、预解析示例：

```
<script>
var num = 789;
function fn() {
    console.log(num);
    var num = 789;
}
fn(); //调用fn函数
</script>
```

此时控制台打印的是 undefined，原因是：JS 解析代码时，把函数的声明还有变量的声明提升到当前作用域的最前面，所以代码就变成：

```
<script>
var num;
num = 789;
function fn(){
    var num; //从这里可以看出num只是声明而没有定义赋值
    console.log(num); //输出的时候必然是undefined
    num = 789;
}
fn();
</script>
```