- 确定 this 的值，也被称为 `This Binding`
- LexicalEnvironment（词法环境） 组件被创建
- VariableEnvironment（变量环境） 组件被创建

伪代码如下：

```JavaScript
ExecutionContext = {
  ThisBinding = <this value>,      // 确定this
  LexicalEnvironment = { ... },    // 词法环境
  VariableEnvironment = { ... },   // 变量环境
}
```

12.2.1.1. This Binding

确定 `this` 的值我们前面讲到， `this` 的值是在执行的时候才能确认，定义的时候不能确认

12.2.1.2. 词法环境

词法环境有两个组成部分：

- 全局环境：是一个没有外部环境的词法环境，其外部环境引用为 `null` ，有一个全局对象， `this` 的值指向这个全局对象
- 函数环境：用户在函数中定义的变量被存储在环境记录中，包含了 `arguments` 对象，外部环境的引用可以是全局环境，也可以是包含内部函数的外部函数环境

伪代码如下：

```javascript
GlobalExectionContext = {  // 全局执行上下文
  LexicalEnvironment: {       // 词法环境
    EnvironmentRecord: {     // 环境记录
      Type: "Object",           // 全局环境
      // 标识符绑定在这里
      outer: <null>           // 对外部环境的引用
  }
}

FunctionExectionContext = { // 函数执行上下文
  LexicalEnvironment: {     // 词法环境
    EnvironmentRecord: {    // 环境记录
      Type: "Declarative",      // 函数环境
      // 标识符绑定在这里      // 对外部环境的引用
      outer: <Global or outer function environment reference>
  }
}
```

12.2.1.3. 变量环境

变量环境也是一个词法环境，因此它具有上面定义的词法环境的所有属性

在 ES6 中，词法环境和变量环境的区别在于前者用于存储函数声明和变量（ `let` 和 `const` ）绑定，而后者仅用于存储变量（ `var` ）绑定

举个例子

```javascript
let a = 20;
const b = 30;
var c;

function multiply(e, f) {
 var g = 20;
 return e * f * g;
}

c = multiply(20, 30);
```

执行上下文如下：

JavaScript | 复制代码

```
GlobalExectionContext = {

  ThisBinding: <Global Object>,

  LexicalEnvironment: {  // 词法环境
    EnvironmentRecord: {
      Type: "Object",
      // 标识符绑定在这里
      a: < uninitialized >,
      b: < uninitialized >,
      multiply: < func >
    }
    outer: <null>
  },

  VariableEnvironment: {  // 变量环境
    EnvironmentRecord: {
      Type: "Object",
      // 标识符绑定在这里
      c: undefined,
    }
    outer: <null>
  }
}

FunctionExectionContext = {

  ThisBinding: <Global Object>,

  LexicalEnvironment: {
    EnvironmentRecord: {
      Type: "Declarative",
      // 标识符绑定在这里
      Arguments: {0: 20, 1: 30, length: 2},
    },
    outer: <GlobalLexicalEnvironment>
  },

  VariableEnvironment: {
    EnvironmentRecord: {
      Type: "Declarative",
      // 标识符绑定在这里
      g: undefined
    },
    outer: <GlobalLexicalEnvironment>
```

```
    }
}
```

留意上面的代码，`let` 和 `const` 定义的变量 `a` 和 `b` 在创建阶段没有被赋值，但 `var` 声明的变量从在创建阶段被赋值为 `undefined`

这是因为，创建阶段，会在代码中扫描变量和函数声明，然后将函数声明存储在环境中

但变量会被初始化为 `undefined`（`var` 声明的情况下)和保持 `uninitialized`（未初始化状态)(使用 `let` 和 `const` 声明的情况下）

这就是变量提升的实际原因

12.2.2. 执行阶段

在这阶段，执行变量赋值、代码执行

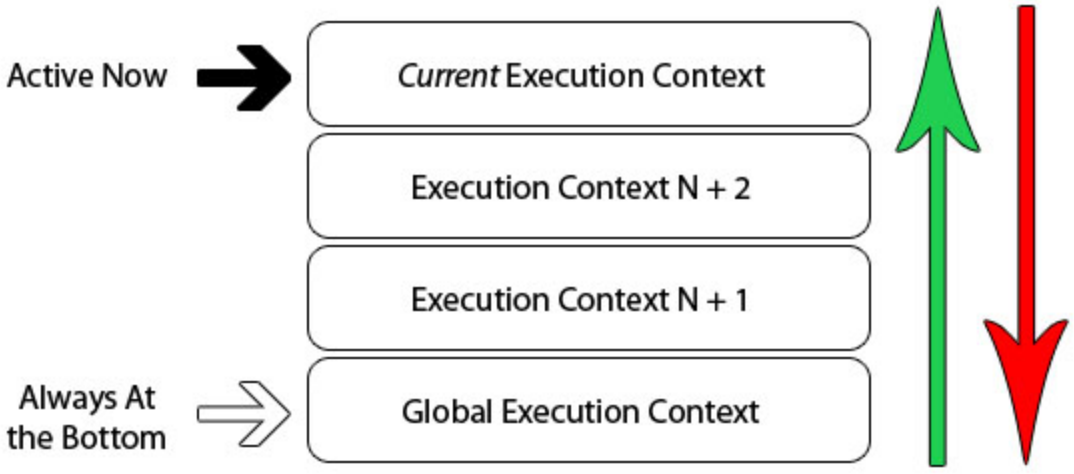
如果 `Javascript` 引擎在源代码中声明的实际位置找不到变量的值，那么将为其分配 `undefined` 值

12.2.3. 回收阶段

执行上下文出栈等待虚拟机回收执行上下文

12.3. 执行栈

执行栈，也叫调用栈，具有 LIFO（后进先出）结构，用于存储在代码执行期间创建的所有执行上下文



当 `Javascript` 引擎开始执行你第一行脚本代码的时候，它就会创建一个全局执行上下文然后将它压到执行栈中

每当引擎碰到一个函数的时候，它就会创建一个函数执行上下文，然后将这个执行上下文压到执行栈中

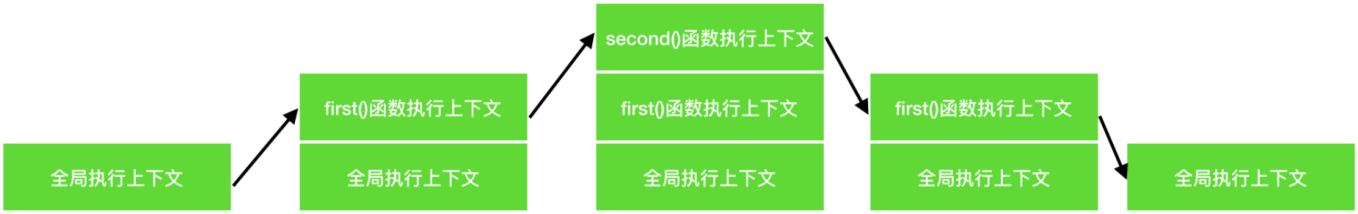
引擎会执行位于执行栈栈顶的执行上下文(一般是函数执行上下文)，当该函数执行结束后，对应的执行上下文就会被弹出，然后控制流程到达执行栈的下一个执行上下文

举个例子：

JavaScript | 复制代码

```
let a = 'Hello World!';
function first() {
  console.log('Inside first function');
  second();
  console.log('Again inside first function');
}
function second() {
  console.log('Inside second function');
}
first();
console.log('Inside Global Execution Context');
```

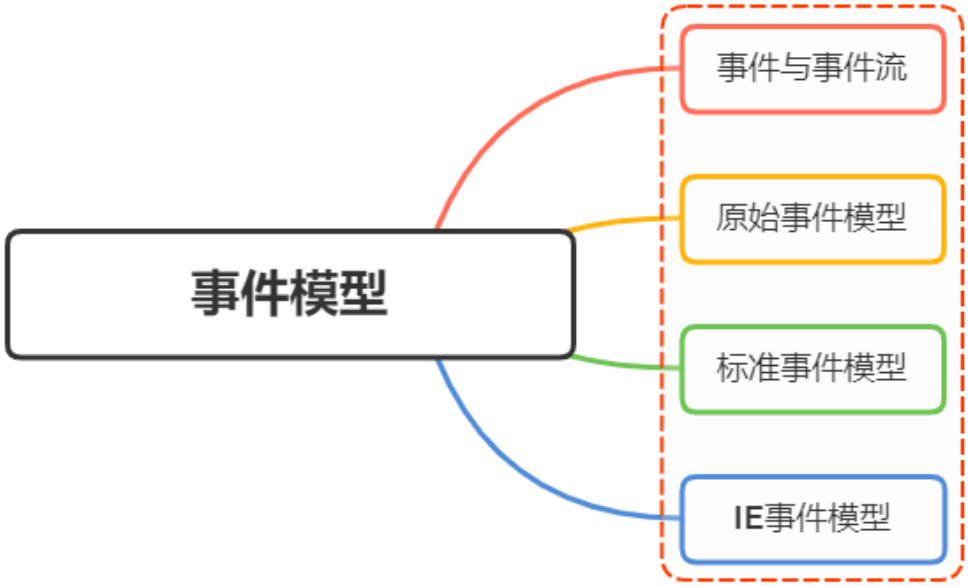
转化成图的形式



简单分析一下流程：

- 创建全局上下文请压入执行栈
- `first` 函数被调用，创建函数执行上下文并压入栈
- 执行 `first` 函数过程遇到 `second` 函数，再创建一个函数执行上下文并压入栈
- `second` 函数执行完毕，对应的函数执行上下文被推出执行栈，执行下一个执行上下文 `first` 函数
- `first` 函数执行完毕，对应的函数执行上下文也被推出栈中，然后执行全局上下文
- 所有代码执行完毕，全局上下文也会被推出栈中，程序结束

13. 说说JavaScript中的事件模型



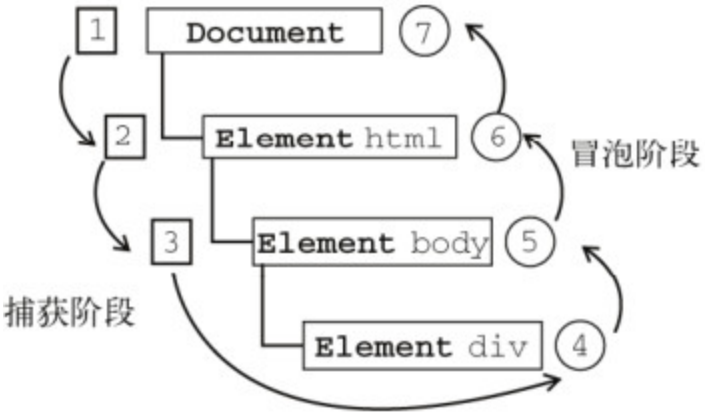
13.1. 事件与事件流

`javascript` 中的事件，可以理解就是在 `HTML` 文档或者浏览器中发生的一种交互操作，使得网页具备互动性， 常见的有加载事件、鼠标事件、自定义事件等

由于 `DOM` 是一个树结构，如果在父子节点绑定事件时候，当触发子节点的时候，就存在一个顺序问题，这就涉及到了事件流的概念

事件流都会经历三个阶段：

- 事件捕获阶段(capture phase)
- 处于目标阶段(target phase)
- 事件冒泡阶段(bubbling phase)



事件冒泡是一种从下往上的传播方式，由最具体的元素（触发节点）然后逐渐向上传播到最不具体的那个节点，也就是 `DOM` 中最高层的父节点

```HTML
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Event Bubbling</title>
    </head>
    <body>
        <button id="clickMe">Click Me</button>
    </body>
</html>
```

然后，我们给 `button` 和它的父元素，加入点击事件

```JavaScript
var button = document.getElementById('clickMe');

button.onclick = function() {
  console.log('1.Button');
};
document.body.onclick = function() {
  console.log('2.body');
};
document.onclick = function() {
  console.log('3.document');
};
window.onclick = function() {
  console.log('4.window');
};
```

点击按钮，输出如下

```JavaScript
1.button
2.body
3.document
4.window
```

点击事件首先在 `button` 元素上发生，然后逐级向上传播

事件捕获与事件冒泡相反，事件最开始由不太具体的节点最早接受事件，而最具体的节点（触发节点）最后接受事件

13.2. 事件模型

事件模型可以分为三种：

- 原始事件模型（DOM0级）
- 标准事件模型（DOM2级）
- IE事件模型（基本不用）

13.2.1. 原始事件模型

事件绑定监听函数比较简单，有两种方式：

- HTML代码中直接绑定

```
<input type="button" onclick="fun()">
```

- 通过 `JS` 代码绑定

```
var btn = document.getElementById('.btn');
btn.onclick = fun;
```

13.2.1.1. 特性

- 绑定速度快

`DOM0` 级事件具有很好的跨浏览器优势，会以最快的速度绑定，但由于绑定速度太快，可能页面还未完全加载出来，以至于事件可能无法正常运行

- 只支持冒泡，不支持捕获
- 同一个类型的事件只能绑定一次

```
<input type="button" id="btn" onclick="fun1()">

var btn = document.getElementById('.btn');
btn.onclick = fun2;
```

如上，当希望为同一个元素绑定多个同类型事件的时候（上面的这个 `btn` 元素绑定2个点击事件），是不被允许的，后绑定的事件会覆盖之前的事件

删除 `DOM0` 级事件处理程序只要将对应事件属性置为 `null` 即可

```JavaScript
btn.onclick = null;
```

13.2.2. 标准事件模型

在该事件模型中，一次事件共有三个过程:

- 事件捕获阶段：事件从 `document` 一直向下传播到目标元素, 依次检查经过的节点是否绑定了事件监听函数，如果有则执行
- 事件处理阶段：事件到达目标元素, 触发目标元素的监听函数
- 事件冒泡阶段：事件从目标元素冒泡到 `document` , 依次检查经过的节点是否绑定了事件监听函数，如果有则执行

事件绑定监听函数的方式如下:

```Plain Text
addEventListener(eventType, handler, useCapture)
```

事件移除监听函数的方式如下:

```Plain Text
removeEventListener(eventType, handler, useCapture)
```

参数如下：

- `eventType` 指定事件类型(不要加on)
- `handler` 是事件处理函数
- `useCapture` 是一个 `boolean` 用于指定是否在捕获阶段进行处理，一般设置为 `false` 与IE浏览器保持一致

举个例子：

```JavaScript
var btn = document.getElementById('.btn');
btn.addEventListener(‘click’, showMessage, false);
btn.removeEventListener(‘click’, showMessage, false);
```

13.2.2.1. 特性

- 可以在一个 `DOM` 元素上绑定多个事件处理器，各自并不会冲突

```JavaScript
btn.addEventListener(‘click’, showMessage1, false);
btn.addEventListener(‘click’, showMessage2, false);
btn.addEventListener(‘click’, showMessage3, false);
```

- 执行时机

当第三个参数(`useCapture`)设置为 `true` 就在捕获过程中执行，反之在冒泡过程中执行处理函数

下面举个例子：

```JavaScript
<div id='div'>
    <p id='p'>
        <span id='span'>Click Me!</span>
    </p >
</div>
```

设置点击事件

```JavaScript
var div = document.getElementById('div');
var p = document.getElementById('p');

function onClickFn (event) {
    var tagName = event.currentTarget.tagName;
    var phase = event.eventPhase;
    console.log(tagName, phase);
}

div.addEventListener('click', onClickFn, false);
p.addEventListener('click', onClickFn, false);
```

上述使用了 `eventPhase` ，返回一个代表当前执行阶段的整数值。1为捕获阶段、2为事件对象触发阶段、3为冒泡阶段

点击 `Click Me!` ，输出如下

```
P 3
DIV 3
```

可以看到， `p` 和 `div` 都是在冒泡阶段响应了事件，由于冒泡的特性，裹在里层的 `p` 率先做出响应

如果把第三个参数都改为 `true`

```
div.addEventListener('click', onClickFn, true);
p.addEventListener('click', onClickFn, true);
```

输出如下

```
DIV 1
P 1
```

两者都是在捕获阶段响应事件，所以 `div` 比 `p` 标签先做出响应

13.2.3. IE事件模型

IE事件模型共有两个过程:

- 事件处理阶段：事件到达目标元素, 触发目标元素的监听函数。
- 事件冒泡阶段：事件从目标元素冒泡到 `document` , 依次检查经过的节点是否绑定了事件监听函数，如果有则执行

事件绑定监听函数的方式如下:

```
attachEvent(eventType, handler)
```

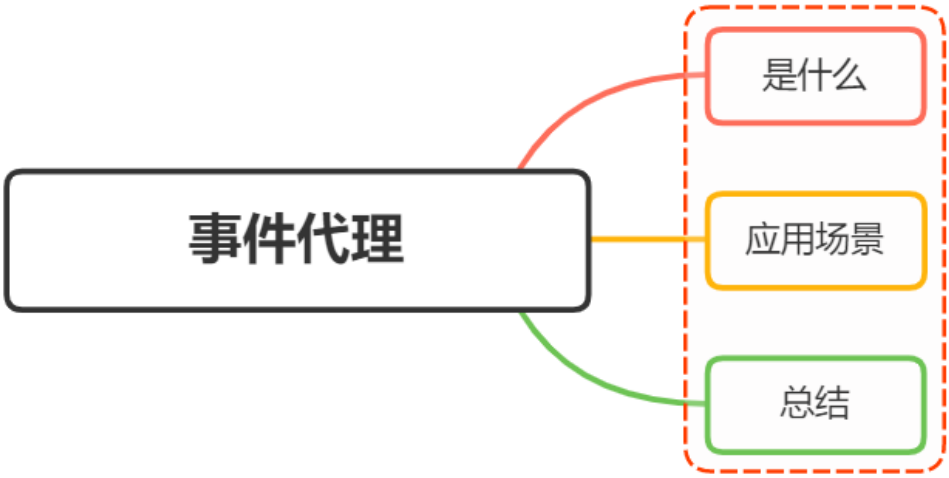
事件移除监听函数的方式如下:

```
detachEvent(eventType, handler)
```

举个例子：

```javascript
var btn = document.getElementById('.btn');
btn.attachEvent('onclick', showMessage);
btn.detachEvent('onclick', showMessage);
```

14. 解释下什么是事件代理？应用场景？



14.1. 是什么

事件代理，俗地来讲，就是把一个元素响应事件（`click`、`keydown`……）的函数委托到另一个元素

前面讲到，事件流的都会经过三个阶段： 捕获阶段 -> 目标阶段 -> 冒泡阶段，而事件委托就是在冒泡阶段完成

事件委托，会把一个或者一组元素的事件委托到它的父层或者更外层元素上，真正绑定事件的是外层元素，而不是目标元素

当事件响应到目标元素上时，会通过事件冒泡机制从而触发它的外层元素的绑定事件上，然后在外层元素上去执行函数

下面举个例子：

比如一个宿舍的同学同时快递到了，一种笨方法就是他们一个个去领取

较优方法就是把这件事情委托给宿舍长，让一个人出去拿好所有快递，然后再根据收件人一一分发给每个同学

在这里，取快递就是一个事件，每个同学指的是需要响应事件的 `DOM` 元素，而出去统一领取快递的宿舍长就是代理的元素

所以真正绑定事件的是这个元素，按照收件人分发快递的过程就是在事件执行中，需要判断当前响应的事件应该匹配到被代理元素中的哪一个或者哪几个

14.2. 应用场景

如果我们有一个列表，列表之中有大量的列表项，我们需要在点击列表项的时候响应一个事件

JavaScript | 复制代码

```
<ul id="list">
  <li>item 1</li>
  <li>item 2</li>
  <li>item 3</li>
  ......
  <li>item n</li>
</ul>
```

如果给每个列表项一一都绑定一个函数，那对于内存消耗是非常大的

JavaScript | 复制代码

```
// 获取目标元素
const lis = document.getElementsByTagName("li")
// 循环遍历绑定事件
for (let i = 0; i < lis.length; i++) {
    lis[i].onclick = function(e){
        console.log(e.target.innerHTML)
    }
}
```

这时候就可以事件委托，把点击事件绑定在父级元素 `ul` 上面，然后执行事件的时候再去匹配目标元素

```JavaScript
// 给父层元素绑定事件
document.getElementById('list').addEventListener('click', function (e) {
    // 兼容性处理
    var event = e || window.event;
    var target = event.target || event.srcElement;
    // 判断是否匹配目标元素
    if (target.nodeName.toLocaleLowerCase === 'li') {
        console.log('the content is: ', target.innerHTML);
    }
});
```

还有一种场景是上述列表项并不多，我们给每个列表项都绑定了事件

但是如果用户能够随时动态的增加或者去除列表项元素，那么在每一次改变的时候都需要重新给新增的元素绑定事件，给即将删去的元素解绑事件

如果用了事件委托就没有这种麻烦了，因为事件是绑定在父层的，和目标元素的增减是没有关系的，执行到目标元素是在真正响应执行事件函数的过程中去匹配的

举个例子：

下面 `html` 结构中，点击 `input` 可以动态添加元素

```HTML
<input type="button" name="" id="btn" value="添加" />
<ul id="ul1">
    <li>item 1</li>
    <li>item 2</li>
    <li>item 3</li>
    <li>item 4</li>
</ul>
```

使用事件委托

```javascript
const oBtn = document.getElementById("btn");
const oUl = document.getElementById("ul1");
const num = 4;

//事件委托，添加的子元素也有事件
oUl.onclick = function (ev) {
    ev = ev || window.event;
    const target = ev.target || ev.srcElement;
    if (target.nodeName.toLowerCase() == 'li') {
        console.log('the content is: ', target.innerHTML);
    }

};

//添加新节点
oBtn.onclick = function () {
    num++;
    const oLi = document.createElement('li');
    oLi.innerHTML = `item ${num}`;
    oUl.appendChild(oLi);
};
```

可以看到，使用事件委托，在动态绑定事件的情况下是可以减少很多重复工作的

14.3. 总结

适合事件委托的事件有：`click`，`mousedown`，`mouseup`，`keydown`，`keyup`，`keypress`

从上面应用场景中，我们就可以看到使用事件委托存在两大优点：

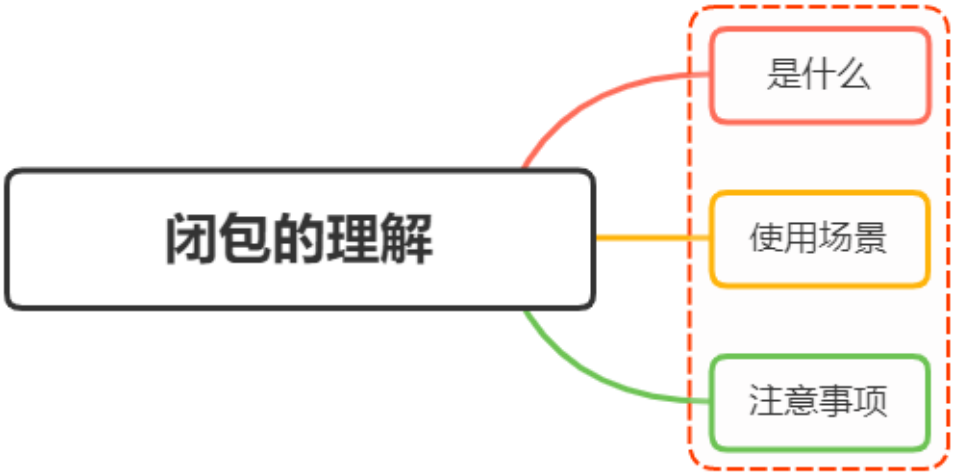
- 减少整个页面所需的内存，提升整体性能
- 动态绑定，减少重复工作

但是使用事件委托也是存在局限性：

- `focus`、`blur` 这些事件没有事件冒泡机制，所以无法进行委托绑定事件
- `mousemove`、`mouseout` 这样的事件，虽然有事件冒泡，但是只能不断通过位置去计算定位，对性能消耗高，因此也是不适合于事件委托的

如果把所有事件都用事件代理，可能会出现事件误判，即本不该被触发的事件被绑定上了事件

15. 说说你对闭包的理解？闭包使用场景



15.1. 是什么

一个函数和对其周围状态（lexical environment，词法环境）的引用捆绑在一起（或者说函数被引用包围），这样的组合就是闭包（closure）

也就是说，闭包让你可以在一个内层函数中访问到其外层函数的作用域

在 `JavaScript` 中，每当创建一个函数，闭包就会在函数创建的同时被创建出来，作为函数内部与外部连接起来的一座桥梁

下面给出一个简单的例子

JavaScript | 复制代码

```
function init() {
    var name = "Mozilla"; // name 是一个被 init 创建的局部变量
    function displayName() { // displayName() 是内部函数，一个闭包
        alert(name); // 使用了父函数中声明的变量
    }
    displayName();
}
init();
```

`displayName()` 没有自己的局部变量。然而，由于闭包的特性，它可以访问到外部函数的变量

15.2. 使用场景

任何闭包的使用场景都离不开这两点：

- 创建私有变量
- 延长变量的生命周期

> 一般函数的词法环境在函数返回后就被销毁，但是闭包会保存对创建时所在词法环境的引用，即便创建时所在的执行上下文被销毁，但创建时所在词法环境依然存在，以达到延长变量的生命周期的目的

下面举个例子：

在页面上添加一些可以调整字号的按钮

JavaScript | 复制代码

```
function makeSizer(size) {
  return function() {
    document.body.style.fontSize = size + 'px';
  };
}

var size12 = makeSizer(12);
var size14 = makeSizer(14);
var size16 = makeSizer(16);

document.getElementById('size-12').onclick = size12;
document.getElementById('size-14').onclick = size14;
document.getElementById('size-16').onclick = size16;
```

15.2.1. 柯里化函数

柯里化的目的在于避免频繁调用具有相同参数函数的同时，又能够轻松的重用