一：箭头函数的基本概念和使用

1.1

使用ES6箭头函数语法定义函数，将原函数的“function”关键字和函数名都删掉，并使用“=>”连接参数列表和函数体。

通常函数的定义方法

```
var fn1 = function(a, b) {
    return a + b
}

function fn2(a, b) {
    return a + b
}
```

使用ES6箭头函数语法定义函数，将原函数的“function”关键字和函数名都删掉，并使用“=>”连接参数列表和函数体。

```
var fn1 = (a, b) => {
    return a + b
}

(a, b) => {
    return a + b
}
```

1.2当函数参数只有一个，括号可以省略；但是没有参数时，括号不可以省略。

```
// 无参
```

```
var fn1 = function() {}
var fn1 = () => {}

// 单个参数
var fn2 = function(a) {}
var fn2 = a => {}

// 多个参数
var fn3 = function(a, b) {}
var fn3 = (a, b) => {}

// 可变参数
var fn4 = function(a, b, ...args) {}
var fn4 = (a, b, ...args) => {}
```

1.3箭头函数相当于匿名函数，并且简化了函数定义。箭头函数有两种格式，一种只包含一个表达式，省略掉了{ ... }和return。还有一种可以包含多条语句，这时候就不能省略{ ... }和return

```
() => return 'hello'
(a, b) => a + b

(a) => {
  a = a + 1
  return a
}
```

1.4如果返回一个对象，需要特别注意，如果是单表达式要返回自定义对象，不写括号会报错，因为和函数体的{ ... }有语法冲突。

注意，用小括号包含大括号则是对象的定义，而非函数主体

```
x => {key: x} // 报错
x => ({key: x}) // 正确
```

1.5箭头函数看上去是匿名函数的一种简写，但实际上，箭头函数和匿名函数有个明显的区别：箭头函数内部的this是词法作用域，由上下文确定。（词法作用域就是定义在词法阶段的作用域。换句话说，词法作用域是由你在写代码时将变量和块作用域写在哪里来决定的，因此当词法分析器处理代码时会保持作用域不变 。）

```
> var Person = {
      firstName: 'hello',
      lastName: 'world',
      getFullName: function() {
          console.log(this)
          var first = this.firstName // hello
          var fn = function() {
              console.log(this)
              return this.firstName + this.lastName
          }
          return fn()
      }
  }
  Person.getFullName()
  ▶ {firstName: "hello", lastName: "world", getFullName: ƒ}
  ▶ Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, frames: Window, …}
< NaN
```


现在，箭头函数完全修复了this的指向，this总是指向词法作用域，也就是外层调用者Person

由于this在箭头函数中已经按照词法作用域绑定了，所以，用call()或者apply()调用箭头函数时，无法对this进行绑定，即传入的第一个参数被忽略

JavaScript中的每一个Function对象都有一个apply()方法和一个call()方法

apply调用一个对象的一个方法，用另一个对象替换当前对象。例如：
B.apply(A, arguments)；即A对象调用B对象的方法。func.apply(thisArg, [argsArray])
call调用一个对象的一个方法，用另一个对象替换当前对象。例如：
B.call(A, args1,args2)；即A对象调用B对象的方法。func.call(thisArg, arg1, arg2, ...)

```
> var Person = {
      firstName: 'hello',
      lastName: 'world',
      getFullName: function(firstName) {
          console.log(this)
          var first = this.firstName // hello
          var fn = function(f) {
              console.log(this)
              return f + this.lastName
          }
          return fn.call({firstName: 'hh'}, firstName)
      }
  }
  Person.getFullName('hi')
  ▶ {firstName: "hello", lastName: "world", getFullName: ƒ}
  ▶ {firstName: "hh"}
< "hiundefined"
```


非箭头函数，调用call()时打印的数据

```
> var Person = {
      firstName: 'hello',
      lastName: 'world',
      getFullName: function(firstName) {
          console.log(this)
          var first = this.firstName // hello
          var fn = (f) => {
              console.log(this)
              return f + this.lastName
          }
          return fn.call({firstName: 'hh'}, firstName)
      }
  }
  Person.getFullName('hi')
  ▶ {firstName: "hello", lastName: "world", getFullName: ƒ}
  ▶ {firstName: "hello", lastName: "world", getFullName: ƒ}
< "hiworld"
```


总结

类似于匿名函数，在某些情况下使用，可减少代码量

代码简洁，this提前定义

代码太过简洁，导致不好阅读

this提前定义，导致无法使用js进行一些在ES5里面看起来非常正常的操作（若使用箭头函数，在监听点击事件的回调函数中，就无法获取到当前点击的元素咯，详见《正确使用箭头函数——什么时候不该用ES6箭头函数》）

总的来说，箭头函数只是一种函数的简写，有其利弊，可用可不用，看大家心情，当然也得用的正确

二：正确使用箭头函数

1.在对象上定义函数

先来看下面这段代码

```
var obj = {
    array: [1, 2, 3],
    sum: () => {
        console.log(this === window); // => true
        return this.array.reduce((result, item) => result + item);
    }
};

// Throws "TypeError: Cannot read property 'reduce' of undefined"
obj.sum();
```

*sum*方法定义在*obj*对象上，当调用的时候我们发现抛出了一个*TypeError*，因为函数中的*this*是*window*对象，所以*this.array*也就是*undefined*。原因也很简单，相信只要了解过es6 箭头函数的都知道

> 箭头函数没有它自己的this值，箭头函数内的this值继承自外围作用域

箭头函数没有自己的this值，箭头函数内的this值继承自外围作用域

解决方法也很简单，就是不用呗。这里可以用es6里函数表达式的简洁语法，在这种情况下，this值就取决于函数的调用方式了。

```
var obj = {
    array: [1, 2, 3],
    sum() {
        console.log(this === obj); // => true
        return this.array.reduce((result, item) => result + item);
    }
};

obj.sum(); // => 6
```

> 通过object.method()语法调用的方法使用非箭头函数定义，这些函数需要从调用者的作用域中获取一个有意义的this值。

2.在原型上定义函数

在对象原型上定义函数也是遵循着一样的规则

```
function Person (pName) {
    this.pName = pName;
}

Person.prototype.sayName = () => {
    console.log(this === window); // => true
    return this.pName;
}

var person = new Person('wdg');

person.sayName(); // => undefined
```

使用function函数表达式

```
function Person (pName) {
    this.pName = pName;
}

Person.prototype.sayName = function () {
    console.log(this === person); // => true
    return this.pName;
}

var person = new Person('wdg');

person.sayName(); // => wdg
```

所以给对象原型挂载方法时，使用function函数表达式

3.动态上下文中的回调函数

*this*是js中非常强大的特点，他让函数可以根据其调用方式动态的改变上下文，然后箭头函数直接在声明时就绑定了this对象，所以不再是动态的。
在客户端，在dom元素上绑定事件监听函数是非常普遍的行为，在dom事件被触发时，回调函数中的this指向该dom,可当我们使用箭头函数时:

```
var button = document.getElementById('myButton');
button.addEventListener('click', () => {
  console.log(this === window); // => true
  this.innerHTML = 'Clicked button';
});
```

因为这个回调的箭头函数是在全局上下文中被定义的，所以他的this是window。所以当this是由目标对象决定时，我们应该使用函数表达式:

```
var button = document.getElementById('myButton');
button.addEventListener('click', function() {
  console.log(this === button); // => true
  this.innerHTML = 'Clicked button';
});
```

4.构造函数中

在构造函数中，this指向新创建的对象实例

```
this instanceOf MyFunction === true
```

需要注意的是，构造函数不能使用箭头函数，如果这样做会抛出异常

```
var Person = (name) => {
    this.name = name;
}

// Uncaught TypeError: Person is not a constructor
var person = new Person('wdg');
```

理论上来说也是不能这么做的，因为箭头函数在创建时this对象就绑定了，更不会指向对象实例。

5.太简短的（难以理解）函数

箭头函数可以让语句写的非常的简洁，但是一个真实的项目，一般由多个开发者共同协作完成，就算由单人完成，后期也并不一定是同一个人维护，箭头函数有时候并不会让人很好的理解，比如

```
let multiply = (a, b) => b === undefined ? b => a * b : a * b;

let double = multiply(2);

double(3); // => 6

multiply(2, 3); // =>6
```

这个函数的作用就是当只有一个参数*a*时，返回接受一个参数*b*返回*a***b*的函数，接收两个参数时直接返回乘积，这个函数可以很好的工作并且看起很简洁，但是从第一眼看去并不是很好理解。
为了让这个函数更好的让人理解，我们可以为这个箭头函数加一对花括号，并加上*return*语句，或者直接使用函数表达式:

```
function multiply(a, b) {
    if (b === undefined) {
        return function (b) {
            return a * b;
        }
    }
    return a * b;
}

let double = multiply(2);

double(3); // => 6
multiply(2, 3);  // => 6
```