1、JS有哪些数据类型？

答：可分为值类型和引用类型两大类。

包括5种原始类型：Number、String、Boolean、Undefined、Null；

引用类型通常叫做类class，即Object。

typeof可以得到对应的数据类型，除了object、function外都是值类型，值类型可直接比较是否相等；引用类型所保存的值只是数据的存储地址，所以判断两个对象是否相等时经常要递归到值类型再比较。

2、Promise 怎么使用？

答：Promise是一个类，在ES6（ECMAScript 2015）中被列为正式规范。一个Promise就是一个代表了异步操作最终完成或者失败的对象。

本质上，一个promise是某个函数返回的对象，你可以把回调函数绑定在这个对象上，而不是把回调函数当做参数传进函数。promise提供了异步函数调用的方案，使代码更加简介清晰。

保证

不像旧式函数那样传递回调函数，promise会带来一些保证：

*在JS事件队列的当前运行完成之前，回调函数永远不会被调用

*通过.then形式添加的回调函数，甚至都在异步操作完成之后才被添加的函数，都会被调用。

*通过多次调用.then，可以添加多个回调函数，它们会按照插入顺序并且独立运行。

但是，promise最直接的好处就是链式调用。

链式调用

一个常见的需求就是连续执行两个或者多个异步操作，这种情况下，每一个后来的操作都在前面的操作执行成功之后，带着上一步操作所返回的结果开始执行。我们可以通过创造一个promise chain来完成这种需求。

```
const promise = doSomething();
const promise2 = promise.then(successCallback, failureCallback);
```

```
const promise2 = doSomething().then(successCallback, failureCallback);
```

第二个primise不仅代表doSomething()函数的完成，也代表了你传入的successCallback或者failureCallback的完成，这也可能是其他异步函数返回的promise。这样的话，任何被添加给promise2的回调函数都会被排在successCallback或failureCallback返回的promise后

面。

基本上，每一个promise代表了链式中另一个异步过程的完成。

在过去，做多重的异步操作，会导致经典的回调地狱：

```
doSomething(function(result){
    doSomethingElse(result, function(newResult){
        doThirdThing(newResult, function(finalResult){
            console.log('Got the final result: ' + finalResult);
        }, failureCallback);
    }, failureCallback);
}, failureCallback);
```

通过现代的函数，我们把回调附加到被返回的promise上代替以往的做法，形成一个promise链：

```
doSomething().then(function(result) {
    return doSomethingElse(result);
}).then(function(newResult){
    return doThirdThing(newResult);
}).then(function(finalResult){
    console.log('Got the final result: ' + finalResult);
}).catch(failureCallback);
```

then里的参数是可选的，catch(failureCallback)是then(null, failureCallback)的缩略形式。如下所示，也可以用arrow functions（箭头函数）来表示。

```
doSomething().then(result=>doSomethingElse(result))
.then(newResult=>doThirdThing(newResult))
.then(finalResult=>{
    console.log(`Got the final result: ${finalResult}`);
})
.catch(failureCallback);
```

注意：如果想要在回调中获取上个promise中的结果，上个promise中必须要返回结果。

catch之后的链式操作

在一个失败操作（即一个catch）之后可以继续使用连式操作，即使链式中的一个动作失败之后还能有助于新的动作继续完成。

```
new Promise((resolve,reject)=>{
```

```
    console.log('Initial');
    resolve();
}).then(()=>{
    throw new Error('Something failed');
    console.log('Do this');
}).catch(()=>{
    console.log('Do that');
}).then(()=>{
    console.log('Do this whatever happened before');
});
```

输出结果如下：

```
Initial
Do that
Do this whatever happened before
```

注意，由于“Something failed”错误导致了拒绝操作，所以“Do this”文本没有被输出。

错误传播

在之前的回调地狱示例中，你可能记得有3次failureCallback的调用，而在promise链中只有底部的一次调用。

```
doSomething()
.then(result=>doSomethingElse(value))
.then(newResult=>doThirdThing(newResult))
.then(finalResult=>console.log(`Got the final result: ${finalResult}`))
.catch(failureCallback);
```

基本上，一个promise链式遇到异常就会停止，查看链式的底端，寻找catch处理程序来代替当前执行。在同步的代码执行之后，这是非常模型化的。

```
try{
    let result = syncDoSomething();
    let newResult = syncDoSomethingElse(result);
    let finalResult = syncDoThirdThing(newResult);
    console.log(`Got the final result: ${finalResult}`);
}catch(error){
    failureCallback(error);
}
```

在ECMAScript2017标准的async/await语法糖中，这种同步形式代码的整齐性得到了极致

的体现：

```
async function foo() {
    try{
        let result = await doSomething();
        let newResult = await doSomethingElse(result);
        let finalResult = await doThirdThing(newResult);
        console.log(`Got the final result: ${finalResult}`);
    }catch(error){
        failureCallback(error);
    }
}
```

这个例子是在promise的基础上构建的，例如，doSomething()与之前的函数是相同的。

通过捕获所有的错误，甚至抛出异常和程序错误，promise解决了回调厄运金字塔的基本缺陷。这是异步操作的基本功能。

在旧式回调API中创建Promise

Promise通过它的构造器从头开始创建。只需要包裹旧的API就行了。

理想状态下，所有的异步函数都已经返回promise了。但有一些API仍然使用旧式的被传入的成功或者失败的回调。典型的例子就是setTimeout()函数：

```
setTimeout(()=>saySomething('10 seconds passed'), 10000);
```

混合旧式回调和promise是会有问题的。如果saySomething函数失败了或者包含了编程错误，那就没有办法捕获它了。

幸运的是我们可以用promise来包裹它。最佳实践是在尽可能底层的地方来包裹有问题的函数，并且永远不要再直接调用它们：

```
const wait = ms => new Promise(resolve => setTimeout(resolve,ms));
wait(10000).then(()=>saySomething('10 seconds')).catch(failureCallback);
```

通常，promise的构造器会有一个可以让我们手动操作resolve和reject的执行函数。既然setTimeout没有真的执行失败，那么我们可以在这种情况下忽略reject。

组成

Promise.resolve()和Promise.reject()是手动创建一个已经resolve或者reject的promise快捷方法。它们有时很有用。

Promise.all()和Promise.race()是并行运行异步操作的两个组合式工具。

时序组合可以使用一些优雅的JS形式：

```
[func1, func2].reduce((p, f) => p.then(f), Promise.resolve());
```

通常，我们递归调用一个由异步函数组成的数组时相当于一个promise链式：

```
Promise.resolve().then(func1).then(func2);
```

我们也可以写成可复用的函数形式，这在函数式编程中极为普遍：

```
let applyAsync = (acc, val) => acc.then(val);
let composeAsync = (...funcs) => x=> funcs.reduce(applyAsync,
Promise.resolve(x));
```

composeAsync函数将会接受任意数量的函数作为其参数，并返回一个新的函数，该函数接受一个通过composition pipeline传入的初始值。这对我们来说非常有益，因为任一函数可以是异步或同步的，它们能被保证按顺序执行：

```
let transformData = composeAsync(func1, asyncFunc1, asyncFunc2, func2);
transformData(data);
```

在ECMAScript 2017标准中，时序组合可以通过使用async/await而变得更简单：

```
for(let f of [func1, func2]){
    await f();
}
```

Timing

为了避免意外，传递给then的函数永远不要再同步调用，即使它是一个已经变成resolve状态的promise：

```
Promise.resolve().then(() => console.log(2));
console.log(1); //1, 2
```

传递到then中的函数被置入了一个微任务队列，而不是立即执行，这意味着它是在JS事件队列的所有运行时结束了，事件队列被清空之后才开始执行。

```
const wait = ms => new Promise(resolve => setTimeout(resolve, ms));

wait().then(() => console.log(4));
Promise.resolve().then(() => console.log(2))
.then(() => console.log(3));
console.log(1); // 1, 2, 3, 4
```

3、AJAX手写一下

答：

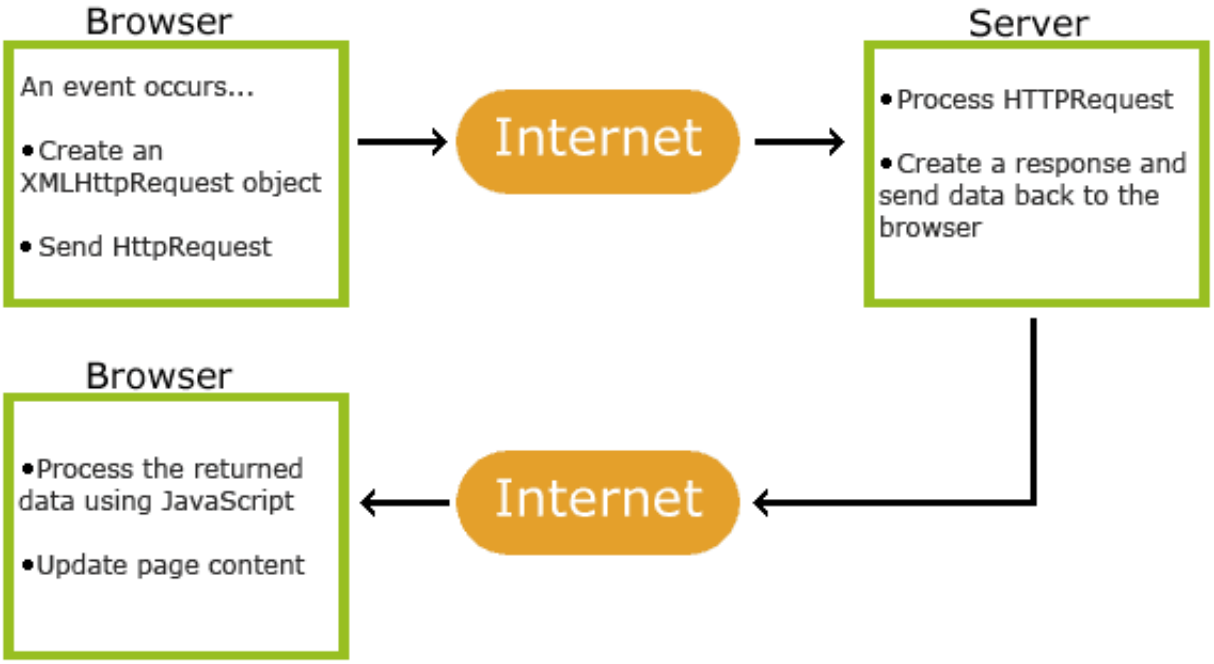
AJAX教程

AJAX（Asynchronous JavaScript and XML）异步的JavaScript和XML。AJAX不是新的编程语言，而是一种使用现有标准的新方法。AJAX最大的优点是在不重新加载整个页面的情况下，可以与服务器交换数据并更新部分网页内容。AJAX不需要任何浏览器插件，但需要用户允许JavaScript在浏览器上执行。

AJAX简介

AJAX是一种在无需重新加载整个网页的情况下，能够更新部分网页的技术。

AJAX=异步JavaScript和XML。AJAX是一种用于创建快速动态网页的技术。通过在后台与服务器进行少量数据交换，AJAX可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。传统的网页（不使用AJAX）如果需要更新内容，必须重载整个网页面。



AJAX是基于现有的Internet标准，并且联合使用它们：XML HTTPRequest对象（异步的与服务器交换数据）；JavaScript/DOM（信息显示/交互）；CSS（给数据定义样式）；XML（作为转换数据的格式）。AJAX应用程序与浏览器和平台无关！

2005年，Google通过其Google Suggest使AJAX变得流行起来。Google Suggest使用AJAX创造出动态性极强的web界面：当您在谷歌的搜索框输入关键字时，JavaScript会把这些字符发送到服务器，然后服务器会返回一个搜索建议的列表。

AJAX实例

XHR创建对象

XMLHTTPRequest是AJAX的基础。所有现代浏览器均支持XMLHTTPRequest对象（IE5和IE6使用ActiveXObject）。XMLHTTPRequest用于在后台与服务器交换数据。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。

```
var xmlhttp;
if(window.XMLHttpRequest){
    // IE7+、Firefox、Chrome、Opera、Safari浏览器执行代码
    xmlhttp = new XMLHttpRequest();
}
else{
    // IE5\IE6浏览器执行代码
    xmlhttp = new ActiveXObject('Microsoft.XMLHTTP');
}
```

如需将请求发送到服务器，我们使用XMLHttpRequest对象的open()和send()方法：

```
xmlhttp.open('GET','ajax_info.txt',true);
xmlhttp.send();
```

方法	描述
open(*method*,*url*,*async*)	规定请求的类型、URL 以及是否异步处理请求。 *method*：请求的类型；GET 或 POST *url*：文件在服务器上的位置 *async*：true（异步）或 false（同步）
send(*string*)	将请求发送到服务器。 *string*：仅用于 POST 请求

与POST相比，GET更简单也更快，并且在大部分情况下都能用。然而，在以下情况中，请使用POST请求：

*无法使用缓存文件（更新服务器上的文件或数据库）

*向服务器发送大量数据（POST没有数据量限制）
*发送包含未知字符的用户输入时，POST比GET更稳定也更可靠

属性	描述
onreadystatechange	存储函数（或函数名），每当 readyState 属性改变时，就会调用该函数。
readyState	存有 XMLHttpRequest 的状态。从 0 到 4 发生变化。 0: 请求未初始化 1: 服务器连接已建立 2: 请求已接收 3: 请求处理中 4: 请求已完成，且响应已就绪
status	200: "OK" 404: 未找到页面

手写原生ajax

```
//创建XHR对象
function createXMLHTTPRequest() {
    //1.创建XMLHttpRequest对象
    //这是XMLHttpRequest对象外部使用中最复杂的一步
    //需要针对IE和其他类型的浏览器建立这个对象的不同方式写不同的代码
    var xmlHttpRequest;
    if(window.XMLHttpRequest){
        //针对Firefox、Mozillar、Opera、Safari、IE7
        xmlHttpRequest = new XMLHttpRequest();
        //针对某些特定版本的mozillar浏览器的BUG进行修正
        if(xmlHttpRequest.overrideMimeType){
            xmlHttpReuqest.overrideMimeType('text/xml');
        }
    }
    else if(window.ActiveXObject){
        //针对IE6、IE5.5、IE5
        //两个可以用于创建XMLHTTPRequest对象的控件名称，保存在一个js的数组中
        //排在前面的版本较新
        var activexName = ['MSXML2.XMLHTTP', 'Microsoft.XMLHTTP'];
        for(var i = 0; i < activexName.length; i++){
            try{
                //取出一个控件名进行创建，如果创建成功就终止循环
                //如果创建失败，回抛出异常，然后可以继续循环，继续尝试创建
                xmlHttpRequest = new ActiveXObject(activexName[i]);
```

```
                if(xmlHttpRequest){
                    break;
                }
            }
            catch(e){}
        }
    }
    return xmlHttpRequest;
}

/*readyState状态
    0：请求为初始化
    1：服务器连接已建立
    2：请求已接收
    3：请求处理中
    4：请求已完成，且响应已就绪
*/
/*status状态
    200：请求成功
    404：未找到
    500：服务器内部错误
*/

//GET请求
function get() {
    var req = createXMLHTTPRequest();
    if(req){
        req.open('GET' 'http://test.com/?keywords=手机', true);
        req.onreadystatechange = function() {
            if(req.readyState == 4){
                if(req.status == 200){
                    alert('success');
                }
                else{
                    alert('error');
                }
            }
        }
        req.send(null);
    }
}

//POST请求
function post(){
    var req = createXMLHTTPRequest();
    if(req){
```

```
        req.open('POST', 'http://test.com/', true);
        req.setRequestHeader('Content-Type', 'application/x-www-form-
urlencoded; charset=gbk;');
        req.send('keywords=手机');
        req.onreadystatechange = function() {
            if(req.readyState == 4){
                if(req.status == 200){
                    alert('success');
                }
                else{
                    alert('error');
                }
            }
        }
    }
}
```

4、闭包是什么

答：

MDN解释：

闭包是函数和声明该函数的词法环境的组合。

词法作用域：

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

init()创建了一个局部变量name和一个名为displayName()的函数。displayName()是定义在init()里的内部函数，仅在该函数体内可用。displayName()内没有自己的局部变量，然而它可以访问到外部函数的变量，所以displayName()可以使用父函数init()中声明的变量name。但是，如果有同名变量name在displayName()中被定义，则会使用displayName()中定义的name。

运行代码可以发现displayName()内的alert()语句成功的显示了在其父函数中声明的name变量的值。这个词法作用域的例子介绍了引擎是如何解析函数嵌套中的变量的。词法

作用域中使用的域，是变量在代码中声明的位置所决定的。嵌套的函数可以访问在其外部声明的变量。

闭包：

```
function makeFunc() {
    var name = "Mozilla";
    function displayName() {
        alert(name);
    }
    return displayName;
}

var myFunc = makeFunc();
myFunc();
```

运行这段代码和之前的 `init()` 示例的效果完全一样。其中的不同 — 也是有意思的地方 — 在于内部函数 `displayName()` 在执行前，被外部函数返回。
第一眼看上去，也许不能直观的看出这段代码能够正常运行。在一些编程语言中，函数中的局部变量仅在函数的执行期间可用。一旦 `makeFunc()` 执行完毕，我们会认为 `name` 变量将不能被访问。然而，因为代码运行得没问题，所以很显然在 JavaScript 中并不是这样的。

这个谜题的答案是，JavaScript中的函数会形成闭包。 闭包是由函数以及创建该函数的词法环境组合而成。这个环境包含了这个闭包创建时所能访问的所有局部变量。在我们的例子中，`myFunc` 是执行 `makeFunc` 时创建的 `displayName` 函数实例的引用，
而 `displayName` 实例仍可访问其词法作用域中的变量，即可以访问到 `name` 。由此，当 `myFunc` 被调用时，`name` 仍可被访问，其值 `Mozilla` 就被传递到alert中。

下面是一个更有意思的示例 — `makeAdder` 函数：

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

从本质上讲，`makeAdder` 是一个函数工厂 — 他创建了将指定的值和它的参数相加求和的函数。在上面的示例中，我们使用函数工厂创建了两个新函数 — 一个将其参数和 5 求和，另一个和 10 求和。

在这个示例中，我们定义了 `makeAdder(x)` 函数，它接受一个参数 `x` ，并返回一个新的函数。返回的函数接受一个参数 `y` ，并返回`x+y`的值。

`add5` 和 `add10` 都是闭包。它们共享相同的函数定义，但是保存了不同的词法环境。在 `add5`的环境中，`x` 为 5。而在 `add10` 中，`x` 则为 10。

解释一：

理解闭包，要先理解JS特殊的变量作用域。变量分两种：全局变量和局部变量。JS语言的特殊之处，就在于函数内部可以直接读取全局变量。

类是有行为的数据，闭包是有数据的行为。

解释二：

闭包并不是什么新奇的概念，它早在高级语言开始发展的年代就产生了。闭包（Closure）是词法闭包（Lexical Closure）的简称。对闭包的具体定义有很多种说法，这些说法大体可以分为两类：

*一种说法认为闭包是符合一定条件的函数，比如参考资源中这样定义闭包：闭包是在其词法上下文中引用了自由变量（除局部变量以外的变量）的函数。

*另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。比如参考资源中就有这样的定义：在实现深约束（绑定）时，需要创建一个能显示表示引用环境的东西，并将它与相关的子程序捆绑在一起，这样捆绑起来的整体被成为闭包。

这两种定义在某种意义上是对立的，一个认为闭包是函数，另一个认为闭包是函数和引用环境组成的整体。虽然有些咬文嚼字，但可以肯定第二种说法更确切。闭包只是在形式和表现上像函数，但实际上不是函数。函数是一些可执行的代码，这些代码在函数被定义后就确定了，不会在执行时发生变化，所以一个函数只有一个实例。闭包在运行时可以有多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。所谓引用环境是指在程序执行中的某个点所有处于活跃状态的约束所组成的集合。其中的约束是指一个变量的名字和其所代表的对象之间的联系。那么为什么要把引用环境与函数组合起来呢？这主要是因为在支持嵌套作用域的语言中，有时不能简单直接地确定函数的引用环境。这样的语言一般具有这样的特性：

*函数是一阶值（First-class value），即函数可以作为另一个函数的返回值或参数，还可以作为一个变量的值。

*函数可以嵌套定义，即在一个函数内部可以定义另一个函数。

虽然建立在相似的思想之上，各种语言所实现的闭包却有着不同的表现形式。JavaScript中的闭包

```
function addx(x){
    return function(y){return x+y;}
}

add8 = addx(8);
add9 = addx(9);

alert(add8(100));
alert(add9(100));
```

闭包的应用：

加强模块化：闭包有益于模块化编程，它能以简单的方式开发较小的模块，从而提高开发速度和程序的可复用性。和没有使用必报的程序相比，使用闭包可将模块划分得更小。

抽象：闭包是数据和行为的组合，这使得闭包具有较好抽象能力。

简化代码：

总结：闭包能优雅地解决很多问题，很多主流语言也顺应潮流，已经或将要引入闭包支持。相信闭包会成为更多人爱不释手的工具。闭包起源于函数语言，也许掌握一门函数语言是理解闭包的最佳途径，而且通过学习函数语言可以了解不同的编程思想，有益于写出更好的程序。

5、这段代码里的this是什么？

6、什么是立即执行函数？使用立即执行函数的目的是什么？

答：IIFE（立即调用函数表达式）是一个在定义时就会立即执行的JS函数。这是一个被成为自执行匿名函数的设计模式，主要包含两部分。 第一部分是包围在圆括号运算符()里的一个匿名函数，这个匿名函数拥有独立的词法作用域。这不仅避免了外界访问此IIFE中的变量，而且又不会污染全局作用域。 第二部分再一次使用()创建了一个立即执行函数表达式，JS引擎到此将直接执行函数。

示例：当函数变成立即执行的函数表达式时，表达式中的变量不能从外部访问。

```
(function () {
    var name = "Barry";
})();
// 外部不能访问变量 name
name // undefined
```

将 IIFE 分配给一个变量，不是存储 IIFE 本身，而是存储 IIFE 执行后返回的结果。

```
var result = (function () {
    var name = "Barry";
    return name;
```

```
})();
// IIFE 执行后返回的结果:
result; // "Barry"
```

7、 async/await 语法了解吗？目的是什么？

答：async/await的用途是简化使用 promises 异步调用的操作，并对一组 Promises执行某些操作。正如Promises类似于结构化回调， async/await类似于组合生成器和promises。

async function 声明将定义一个返回 AsyncFunction 对象的异步函数。

返回值：返回的Promise对象会以async function的返回值进行resolve，或者以该函数抛出的异常进行reject。

描述：当调用一个 async 函数时，会返回一个 Promise 对象。当这个 async 函数返回一个值时，Promise 的 resolve 方法会负责传递这个值；当 async 函数抛出异常时，Promise 的 reject 方法也会传递这个异常值。 async 函数中可能会有 await 表达式，这会使 async 函数暂停执行，等待表达式中的 Promise 解析完成后继续执行 async 函数并返回解决结果。 注意， await 关键字仅仅在 async function中有效。如果在 async function函数体外使用 await ，你只会得到一个语法错误（SyntaxError）。

不要将await和Promise.all混淆：在函数add1中，程序为第一个await停留了2秒， 然后为第二个await又停留了2秒。第一个计时器结束后，第二个计时器才被创建。在函数add2中，两个计时器均被创建，然后一起被await。 这导致程序运行出结果需要2秒而非4秒，因为这两个计时器是同时运行的。但是这两个await调用仍然是串行而非并行的： Promise.all并没有自动做这种操作。如果你想要同时await两个或者更多Promise对象，必须使用Promise.all。

通过async方法重写promise链：返回Promise的API将会被用于promise链，它会将函数分成若干部分。例如下面代码：

```
function getProcessedData(url) {
  return downloadData(url) // returns a promise
    .catch(e => {
      return downloadFallbackData(url)  // returns a promise
        .then(v => {
          return processDataInWorker(v); // returns a promise
        });
    })
    .then(v => {
      return processDataInWorker(v); // returns a promise
    });
}
```

可以通过如下所示的一个async函数重写：

```
async function getProcessedData(url){
    let v;
    try {
        v = await downloadData(url);
    }catch(e){
        v = await downloadFallbackData(url);
    }
    return processDatainWorker(v);
}
```

注意：在上述示例中，return语句中没有await操作符，因为async function的返回值将隐式传递给Promise.resolve。

8、如何实现深拷贝？

答：首先js的数据类型可以分为两大类：值类型、引用类型。其中值类型存储的是对应的数值，引用类型存储的只是一个地址。直接赋值的话，值类型没有问题，但是引用类型将会把地址复制过去，这时候对新的变量进行的操作，也会体现到原数据上。可能会引发的问题就是子模块有可能在未知情况下改变了某个全局变量，这就是很大条的事情了。而深拷贝是一层层递归下去直到值类型，然后用取到的数据创建一个新的变量，存放在新的地址，这样对新数据的操作就不会影响到原数据。

实现深拷贝主要是两点：一是判断数据类型是否是引用类型（Array、Object、Function），二是递归复制。

序列化&反序列化可完成Array、Object、String等能够被json表示的数据结构的深拷贝。但是函数这种不能被json表示的类型将不能被正确处理。

结构化克隆算法

结构化克隆算法是由HTML5规范定义的用于序列化复杂JS对象的一个新算法。它比JSON更有能力，因为它支持包含循环图的对象的序列化——对象可以引用在同一个图中引用其他对象的对象。此外，外某些情况下，结构化克隆算法可能比JSON更高效。 算法本质上是将原始对象的所有字段的值复制到新对象里。如果一个字段是对象，这些字段会被递归复制，知道所有的字段和子字段都被复制进新的对象里。

优于JSON的地方：

*结构化克隆可以复制RegExp对象；

*结构化克隆可以复制Blob、File以及FileList对象

*结构化克隆可以复制ImageData对象。CanvasPixelArray的克隆粒度将会跟原始对象相同，并且复制出来相同的像素数据。

*结构化克隆可以正确的复制有循环引用的对象。

结构化克隆所不能做到的：

*Error以及Funciton对象是不能被结构化克隆算法复制的；如果你尝试这样子去做，这会

导致抛出DATA_CLONE_ERR的异常。

*企图去克隆DOM节点同样会抛出DATA_CLONE_ERROR异常；

*对象的某些特定参数也不会被保留

。RegExp对象的lastIndex字段不会被保留

。属性描述符，setters以及getters（以及其他类似元数据的功能）同样不会被复制

。原型链上的属性也不会被追踪以及复制。

另一种方法：深复制

如果你想深复制一个对象（那就是沿着原型链，对所有属性进行递归复制），你必须要用另外一种方法。以下是一个可行的例子。

```
function clone(objectToBeCloned){
    //Basis.
    //基础数据类型
    if(!(objectToBeCloned instanceof Object)){
        return objectToBeCloned;
    }

    var objectClone;

    //Filter out special objects.
    //筛选出特殊对象
    var Constructor = objectToBeCloned.constructor;
    switch(Constructor){
        //Implement other special objects here.
        //在这里执行其他特殊对象
        case RegExp:
            objectClone = new Constructor(objectToBeCloned);
            break;
        case Date:
            objectClone = new Constructor(objectToBeCloned.getTime());
            break;
        default:
            objectClone = new Constructor();
            break;
    }

    //Clone each property.
    //拷贝每个属性（object、array）
    for(var prop in objectToBeCloned){
        objectClone[prop] = clone(objectToBeCloned[prop]);
    }

    return objectClone;
}
```

注意：此算法实际只实现了RegExp、Array和Date特殊对象。根据你的需要，你可以实现其他特殊情况。