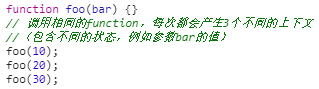
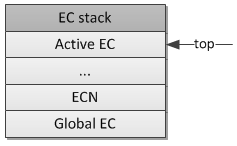
1. 底层
   1. 编译和核心

**1.执行上下文**

ECMAScript引擎将ECMAScript代码编译为可执行代码（executable code），即可在CPU上执行。ES5中，EC有三种类型：global、function和eval。代码开始执行时，即进入global上下文，这也是堆栈中最底部的元素。调用执行function代码会进入函数执行中的上下文。eval函数的执行，也会进入eval执行中的上下文。

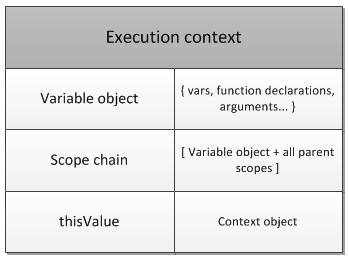


一个执行上下文可以激活另一个上下文，如同一个函数调用另一个函数，或全局上下文中调用了一个全局函数。实现这种方式是栈，称之为上下文堆栈。堆栈底部为全局上下文，顶部就是当前（活动的）执行上下文。堆栈在EC类型进入和退出上下文的时候被修改（推入或弹出）。



激活其它上下文的称为调用者（caller）。被激活的上下文称为被调用者（callee）。当一个caller激活了一个callee，caller就会暂停自身的执行，将控制权交给callee，callee被放入堆栈，称为进行中的上下文。当callee的上下文结束之后，会把控制权再次交给caller，caller会在刚才暂停的地方继续执行。一个callee可以通过返回（return）或者抛出异常（exception）来结束自身的上下文。

一个执行的上下文可以理解为一个object。每一个执行的上下文都有一系列的属性（称为上下文状态），主要有变量对象、this指针、作用域链，还可以有任意额外属性。

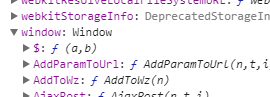


**2.变量对象**

（1）变量对象是与执行上下文相关的数据作用域，用于存储被定义在上下文中的变量、函数声明和函数的形参。

（2）global上下文的变量对象就是全局对象。全局对象在进入任何执行上下文之前创建，且只存在一份，终止于程序退出的那一刻。

全局对象创建阶段就将Math、String、Date、parseInt等作为自身属性。全局对象的window属性可用来引用全局对象自身。



（3）函数执行上下文中的VO不能直接访问，此时由活动对象（activation object、AO）扮演VO的角色。活动对象在进入函数上下文时被创建。

执行上下文的代码被分成两个基本阶段来处理：进入执行上下文和执行代码。

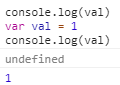
当进入函执行上下文（可以称为词法分析阶段）时，VO包含如下属性：函数的所有形参、所有函数声明、所有变量声明，不包含函数表达式。分为3步：首先分析参数，再分析变量声明，在分析函数声明。具体步骤为：

1）将形参作为活动对象的属性名，将形参值作为对应属性值。

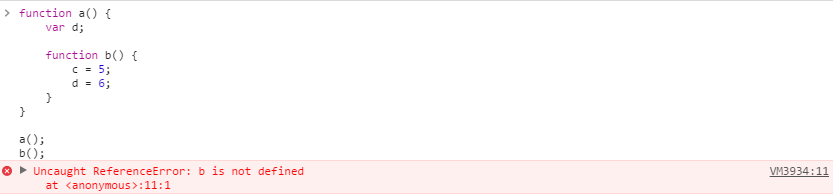
2）分析函数体内变量声明（var），如果AO已有该属性（形参），不做任何处理。如果没有，添加到AO上，但不赋值（undefined）。

3）分析函数声明，将函数名添加到AO上，如果之前已存在，覆盖。

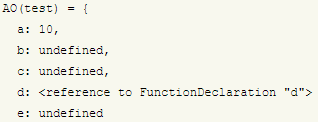
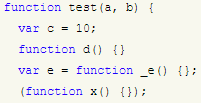
（4）如下，声明阶段并不赋值，在执行的时候才会被赋值。



（5）在函数内部嵌套声明的函数，外部无法访问，不会将声明提升外部，即使外部函数执行也不行。



（7）如下，当执行到test(10)函数代码时（test函数声明的提示是在其调用上下文中），首先生成如下活动对象：

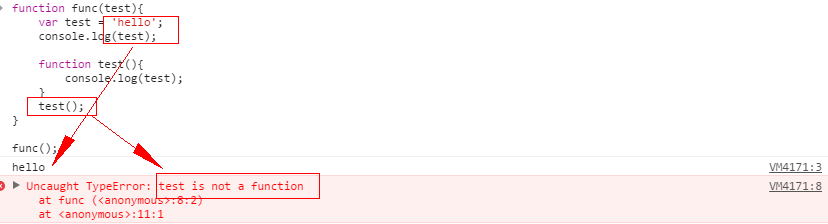


代码解释期间被修改如下：



因为x和\_e都是函数表达式，无论是进入上下文阶段，还是执行代码阶段，x和\_e不会作为AO的属性，无法直接访问。

（8）又如：



具体过程为：

首先创建AO={}。然后添加形参属性AO={test:undefined}。分析变量声明，因为已经有test属性，不影响。分析函数声明，更改为AO={test:function(){}}。

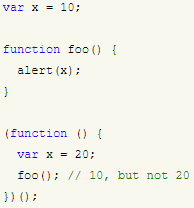
但是，在执行过程中，有：对test赋值，更改AO为{test:hello}。test为字符串，因此报错。

**2.作用域链**

（1）作用域链[[scope]]在函数创建时创建，为immutable，可以理解为1个数组。活动对象是作用域数组的第一个对象，即添加到作用域的前端。



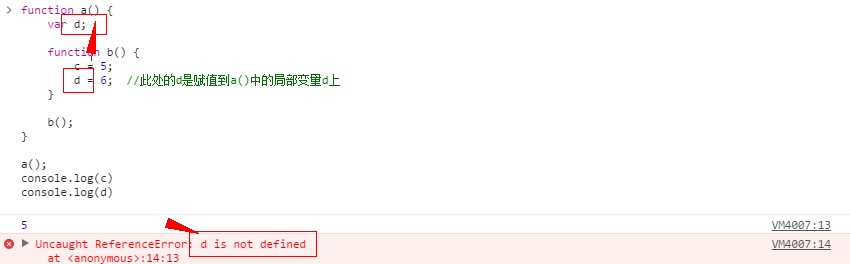
如下，[[Scope]]包括在函数内创建的词法作用域，闭包是函数代码和其[[scope]]的结合。



（2）函数表达式和函数声明一样，保存类似的[[scope]]。

（3）代码执行阶段，如果遇到with、catch语句，它们添加到作用域链的最前端。

（4）变量声明时如果没有加var，则仅作为1个赋值操作。赋值时仍然向上寻找变量声明。



执行a函数时，c、d都会向上查看是否有声明，其中d被a拦截，c则一直查找到window，作为window的属性。此时输入window.d只是undefined但不会报错。



**3.this**

和其他编程语言类似，作为构造函数，this完全指向由构造器新创建的对象。但除此之外，还有其他特性。有4种绑定方式：

（1）全局上下文中，this指向全局对象本身（window）。



（2）作为对象的方法调用，此时this指向方法的调用者即obj对象。



又如yy调用test方法是显示为yy。即，不管声明在何处，只管调用时被谁调用。



（3）作为构造函数

根据此时创建对象的过程，this会绑定到新创建的那个对象上。

（4）函数使用apply、call调用

注意：this也不能沿着AO链往上查找。this也不能手动重置。

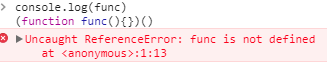
* 1. 函数

**1.概念**

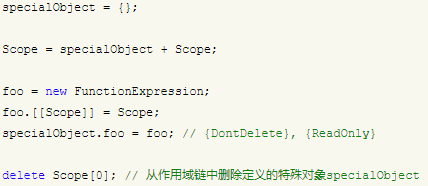
（1）ECMAScript中有三种函数类型：函数声明，函数表达式和函数构造器创建的函数。

（2）不要在if-else等分支语句中声明函数。

（3）闭包中的函数声明可以称为命名函数表达式，函数声明不会提示，不影响上下文的变量对象。



当执行到命名函数表达式时，会先创建了辅助的特定对象，并添加到当前作用域链的最前端。



因此，在函数外部这个名称不可用的（因为它不在父作用域链中），但特定对象已经存储在函数的[[scope]]中，在函数内名称是可用的。

**2.函数式编程**

（1）函数式语言中，函数即数据，可以赋值给变量，可以当参数传递，还可以从函数里返回等。可以以正常数据形式存在的函数都是第一类对象。

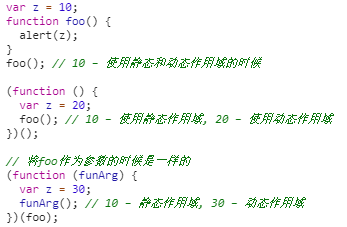
（2）接受自己作为参数的函数，称为自应用函数。以自己为返回值的函数称为自复制函数。

（3）在面向堆栈的编程语言中，函数的局部变量都是保存在栈上的，每当函数激活的时候，这些变量和函数参数都会压入到该堆栈上。当函数返回的时候，这些参数又会从栈中移除。

这种模型对将函数作为函数式值使用的时候有很大的限制（如，作为返回值从父函数中返回），主要问题是当函数有自由变量的时候。自由变量是指在函数中使用的，但既不是函数参数也不是函数的局部变量的变量。

如下，如果使用动态作用域，变量是通过变量动态栈来管理的。此时自由变量是在当前活跃的动态链中查询的，而不是在函数创建的时候保存起来的静态作用域链中查询的。此时就会产生冲突：z仍然存在（与之前从栈中移除变量的例子相反），还是会有这样一个问题：在不同的函数调用中，Z的值到底取哪个呢（从哪个上下文，哪个作用域中查询）？

上述描述的就是两类funarg问题：取决于是否将函数以返回值返回（第一类问题）以及是否将函数当函数参数使用（第二类问题）。

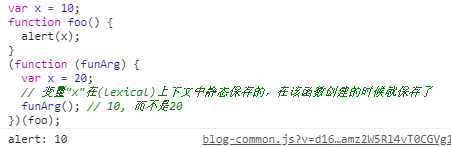


为了解决上述问题，就引入了闭包的概念。

**3.闭包**

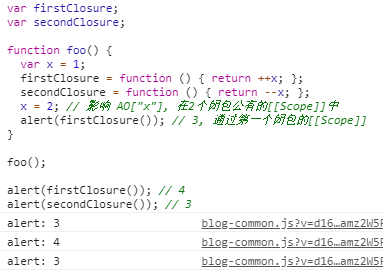
（1）闭包是代码块和创建该代码块的上下文中数据的结合。如，创建函数的上下文已经销毁而函数仍然存在（如从父函数中返回），或在代码中引用了自由变量，此时的函数就是闭包。

（2）ECMAScript使用静态（词法）作用域实现闭包。

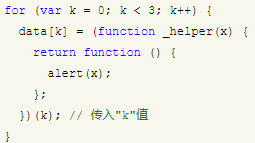
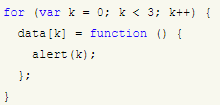


即，创建该函数的父级上下文的数据是保存在函数的内部属性[[Scope]]中。因此，本质上讲，ECMAScript中所有的函数都是闭包，因为它们都是在创建的时候就保存了上层上下文的作用域链。

（3）同一个父上下文中创建的闭包是共用一个[[Scope]]属性的。即，某个闭包对其中[[Scope]]的变量做修改会影响到其他闭包对其变量的读取。



之所以循环语句里创建函数得不到预期结果，和上面一样，都是同一个上下文中创建的闭包是共用一个[[Scope]]属性。如果在创建1个闭包，\_helper创建出来之后，每次\_helper都会创建一个新的变量对象，其中含有参数x，即传递进来的k。



（4）闭包使用到的参数和变量不会被垃圾回收机制回收，会增大内存使用量。

（5）闭包常见形式：

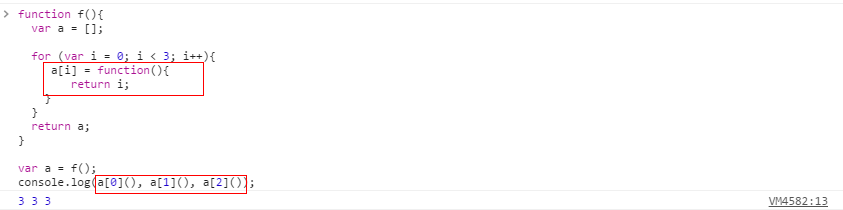
直接返回1个函数。



将函数赋给1个变量，然后返回这个变量。



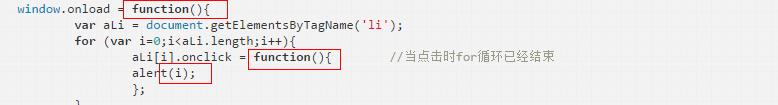
将多个函数保存到1个数组中，返回返回这个数组。



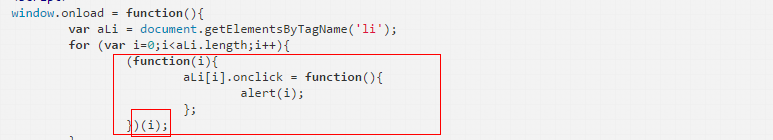
（6）使用闭包模拟私有变量



（7）绑定dom



上面只会绑定aLi.length-1这个节点事件。



（8）将闭包设为null。

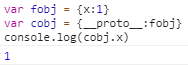
* 1. 对象

**1.概念**

（1）js中的对象是一个属性的集合。每个对象都拥有一个原型对象，该原型对象可以是一个对象或者null值。对象内部[[prototype]]属性指向这个原型对象。通常，对象内部属性用\_\_<内部属性名>\_\_的下划线来代替双括号。即，对象的\_\_proto\_\_指向其原型对象。

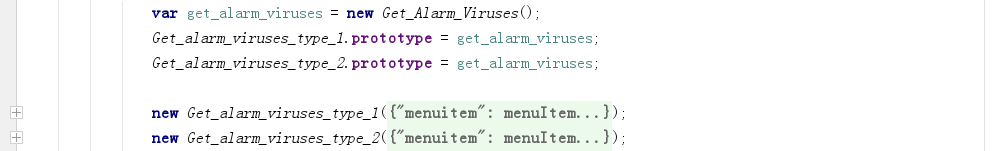
（2）原型对象也可能是普通的对象，并且也有可能有自己的原型。如果一个原型对象的原型不为null的话，就称之为原型链。

ECMAScript中没有类的概念，但通过原型链，仍然可以实现重用。这种基于原型链的继承，称为原型链继承。



（3）如果一个对象的prototype没有显示的声明过，那么\_\_prototype\_\_的默认值就是object.prototype。而object.prototype也会有一个\_\_prototype\_\_，它就是原型链的终点，被设置为null。

（4）构造函数可用于创建对象，同时还自动为创建的新对象设置了原型对象。所有函数都有prototype属性，构造函数创建的对象的原型也就指向构造函数的这个属性。



**2.创建对象模式**

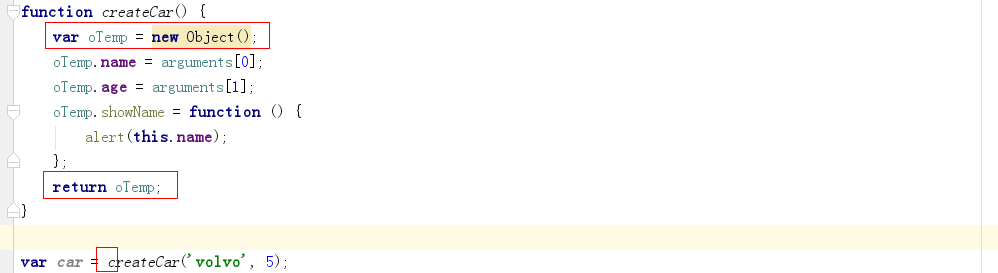
（1）字面量/Object构造函数





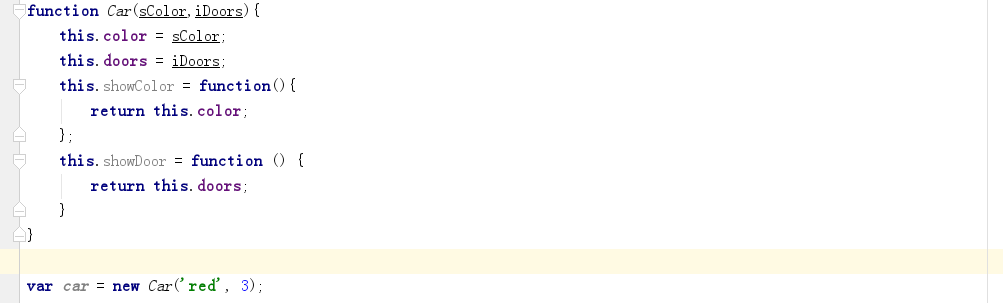
Object后面的()可以省略。因每次创建对象，都要写属性名/值，不同对象不能复用，导致重复代码太多。

（2）工厂模式



将创建对象的过程包装在1个函数中，通过入参接口返回不同的对象。优点：减少代码量。缺点：1）无法识别对象类别（创建对象不用new）。2）每个方法都是不同实例。

（3）构造函数模式



创建过程：1）new出1个新的空对象。2）将构造函数的作用域指向这个对象。3）执行构造函数。4）返回新对象。

优点：通过new构造函数创建，因此能够识别对象类型。

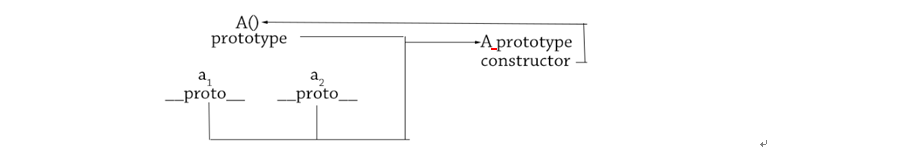


缺点：不同实例的方法都不同。解决方法：将方法从构造函数中移到外面，但这样1是破坏语义，2是方法成了全局的了。

（4）原型模式

1）原理

* 每新建1个函数，就会自动为该函数创建1个prototype属性，该属性指向该函数的原型对象。
* 初始情况下，该原型对象除继承Object属性外，还会为该原型对象生成constuctor（构造函数）属性，指向新建的函数。
* 用new这个函数创建出来的实例，都会有\_\_proto\_\_属性，指向原型对象（不是指向构造函数，实例和构造函数没有直接联系）。



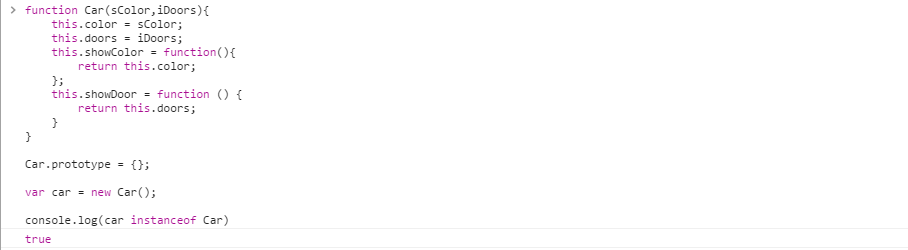
* 对象是new创建的，并非函数创建的。实例通过\_\_proto\_\_指向原型建立间接关系，因此，给函数添加静态方法实例无法访问。
* 可以通过A.prototype.func=?创建方法，实例能够访问。
* 可以通过hasOwnProperty()检测是自身还是原型对象属性。



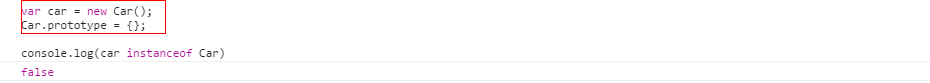
* 可以通过in/for-in判断/获取对象可访问属性（无论是自身还是原型中）。



* 如果重写整个原型，将导致原型的constructor不再指向构造函数，而是指向Object的构造函数。



* 如果实例化后再重写原型，将会导致实例无法访问原型属性/方法。



上面2种情况原型无法通过其constructor属性获取构造函数，但前者instanceof没问题，说明不是用的constructor来判断。

2）创建对象

将所有属性定义在原型对象上，导致所有实例共用同一个属性。

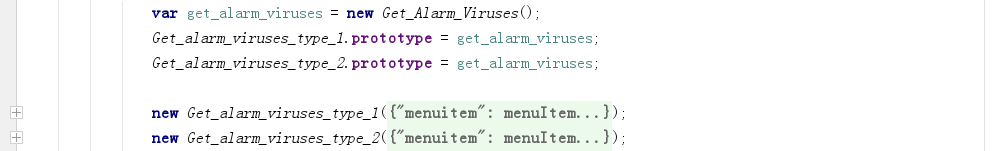
3）构造函数+原型对象

属性定义在构造函数中，方法定义在原型中。

**3.继承**

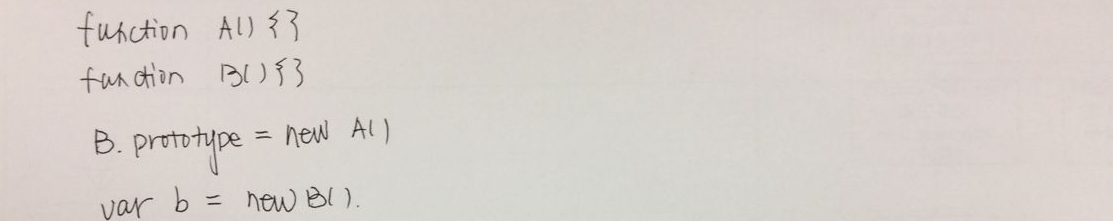
（1）原型链继承

重写1个构造函数的原型对象，使其成为另1个构造函数的实例，使得实例的\_\_proto\_\_指向另1个构造函数实例，进而能够访问这个实例的属性、方法。

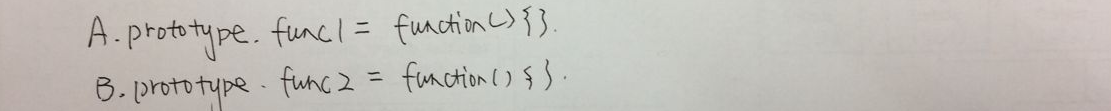




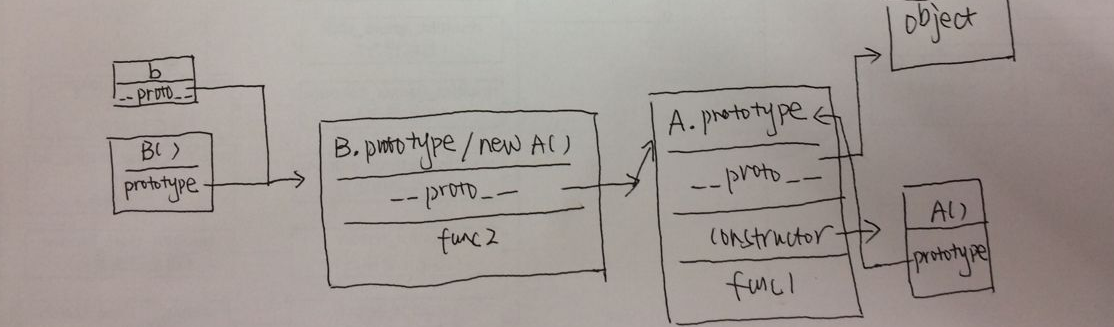
（2）完整原型链



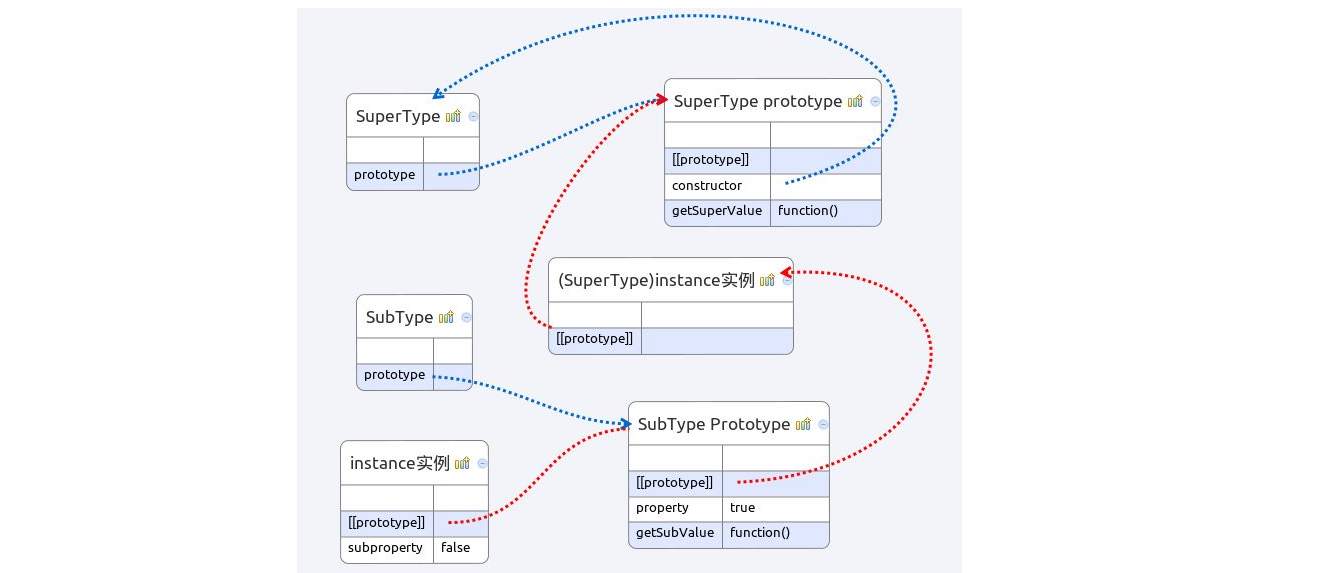
B默认的原型为object，现在重写为A的实例。



给B的原型即A的实例添加方法，是在对象上的方法。



* 实例b的\_\_proto\_\_和构造方法B的prototype都为A的1个实例。
* 因为重写B的prototype，因此它没有constructor。
* A的prototype（是1个对象）的\_\_proto\_\_指向object。



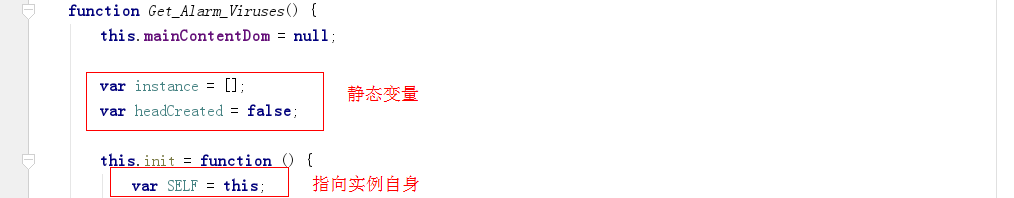
（3）借用构造函数继承（伪对象继承）



因为方法都是在构造函数中定义，因此仍然无法复用。

**4.总结**

（1）在原型内的this为实例自身。但变量可以视为静态变量。



（2）原型中的属性为所有子类所共享，但是，通过1个方法返回这些属性值，可以实现每个子类共用一段代码但同时属性值又不相同的效果。



1. 一些概念厘清
   1. JSON

JSON只是一个数据交换语言，并非JavaScript对象字面量，只有我们将之用在string上下文的时候它才叫JSON。

JSON是设计成描述数据交换格式的，他也有自己的语法，这个语法是JavaScript的一个子集。

对象字面量不是JSON对象，但是有真正的JSON对象。但是两者完全不一样概念，在新版的浏览器里JSON对象已经被原生的内置对象。

* 1. 事件循环

**1.浏览器环境**

（1）单线程：在执行代码的任何时候，都只有一个主线程来处理所有的任务。js使用单线程的主要原因在于避免多个线程对同一dom操作。

（2）非阻塞：当代码执行一项异步任务时，主线程会挂起这个任务，等待异步任务返回结果时，再根据一定规则去执行相应的回调。

（3）js执行时，将不同的变量存于内存不同位置：堆（heap）和栈（stack）。其中，堆存放着一些对象。栈存放着一些基础类型变量以及对象的指针。

（4）调用1个方法时，js会生成一个与这个方法对应的执行环境，即执行上下文。执行环境中保存了方法的私有作用域，上层作用域的指向，方法的参数，作用域中定义的变量，以及这个作用域的this对象。

当一系列方法被依次调用的时候，因js是单线程的，同一时间只能执行一个方法，因此这些方法被排队在一个单独的地方，即执行栈。执行完1个方法并返回结果后，js会退出这个执行环境并把这个执行环境销毁，回到上一个方法的执行环境。

上述过程即代码的同步执行过程。

（5）js引擎遇到一个异步事件后并不会一直等待其返回结果，而是会将这个事件挂起，继续执行执行栈中的其他任务。

当一个异步事件返回结果后，js会将这个事件加入与当前执行栈不同的另一个队列，即事件队列。事件队列对应的回调不会立刻执行，而是等待当前执行栈中的所有任务都执行完，主线程处于闲置状态后，主线程才会去查找事件队列是否有任务。如果有则取出事件对应的回调放入执行栈中，然后执行其中的同步代码。如此反复形成了一个无限的循环，即事件循环。

（6）异步任务也有优先级：被区分为微任务（micro task）和宏任务（macro task）。前者有Promise、MutaionObserver等，后者有setInterval、setTimeout等。

当前执行栈执行完毕时，主线程会立刻先处理所有微任务队列中的事件，然后再去宏任务队列中取出一个事件。即，同一次事件循环中，微任务永远在宏任务之前执行。

**2.node环境**

浏览器环境下，microtask的任务队列是每个macrotask执行完之后执行。而在Node.js中，microtask会在事件循环的各个阶段之间执行，也就是一个阶段执行完毕，就会去执行microtask队列的任务。

* 1. 文件上传及处理
     1. 古老浏览器

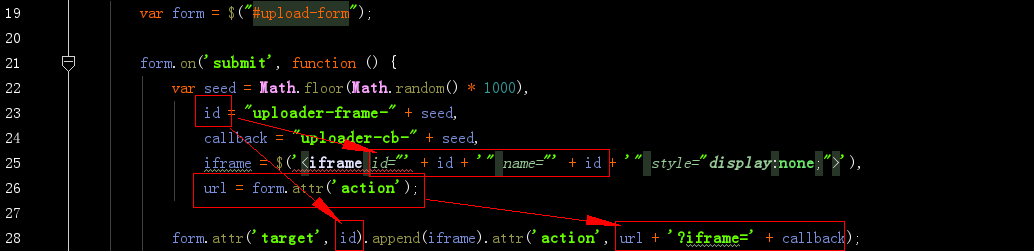
**1.原始上传**

即，直接使用表单元素input[type=file]。但上传文件不同于一般数据的POST，不能通过ajax方式，必须通过表单提交完成。

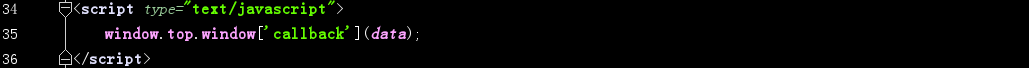
**2.利用iframe上传**

在HTML5没有出现之前，只能使用iframe模拟异步上传（即，不用像原始上传那样还需要页面跳转）。需要做到：

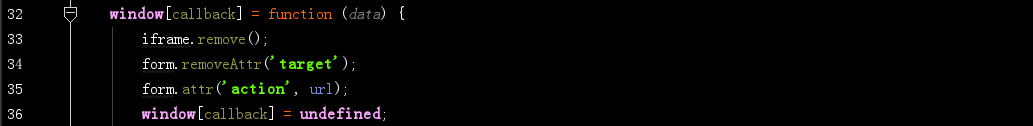
（1）用户点击submit时，动态插入一个iframe元素。该iframe元素的name=form的target。通常动态设置form的target，并与新建的iframe一致。



（2）类似于jsonp，动态将1个回调附在form原来的action后面，服务器返回的是1个script，并需要将数据传给这个回调。



由此，服务器返回的这个脚本会自动写入iframe并执行。因此，可以在原来的代码中定义执行成功回调。



* + 1. 现代浏览器

**1.<form>标签的enctype属性**

enctype属性规定在发送到服务器之前应该如何对表单数据进行编码。而跟url编码、Content-Type等没关系。只有3个值，如下：

（1）application/x-www-form-urlencoded和Content-Type

两者貌似没有明显区别，使用get方式提交时，把表单数据以键值对形式追加到url后。使用post方式提交时，把表单数据以键值对放在请求体中传输。

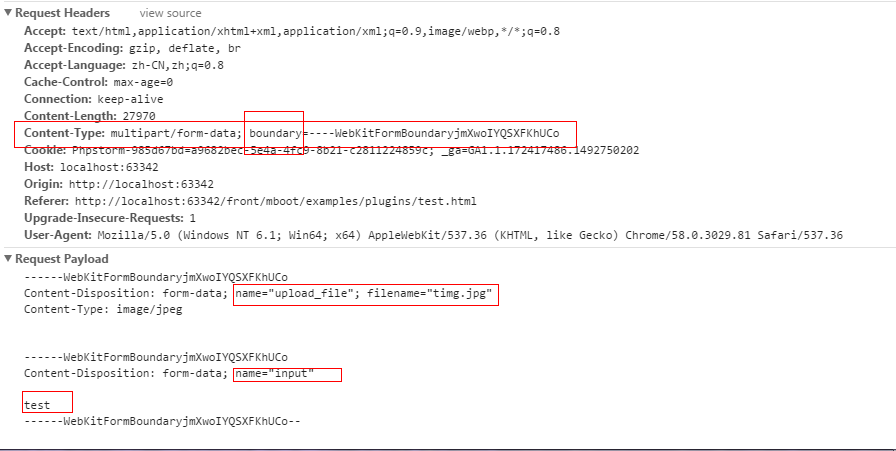




（2）multipart/form-data

表单数据被编码为一条消息，页上的每个<input>对应消息中的一个部分，用boundary（值由浏览器生成）分割各个部分。它不会对字符进行编码，一般用于传输二进制文件（图片、视频等）。



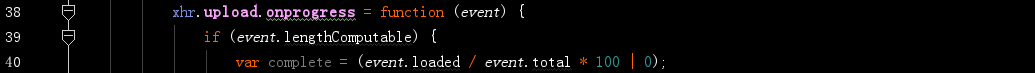


**2.ajax上传**

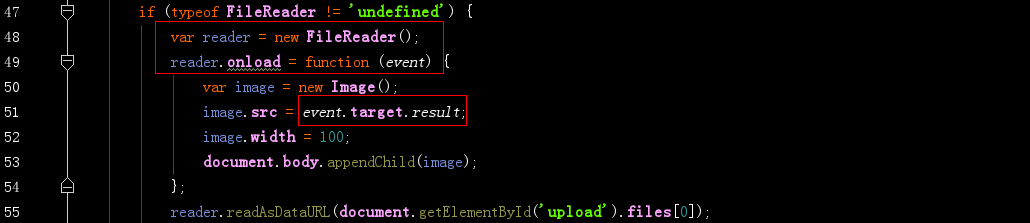
HTML5提出了第二版XMLHttpRequest对象（ie10支持），可以通过ajax上传文件。这是真正的异步上传。iframe上传可以用作老式浏览器的替代方案。



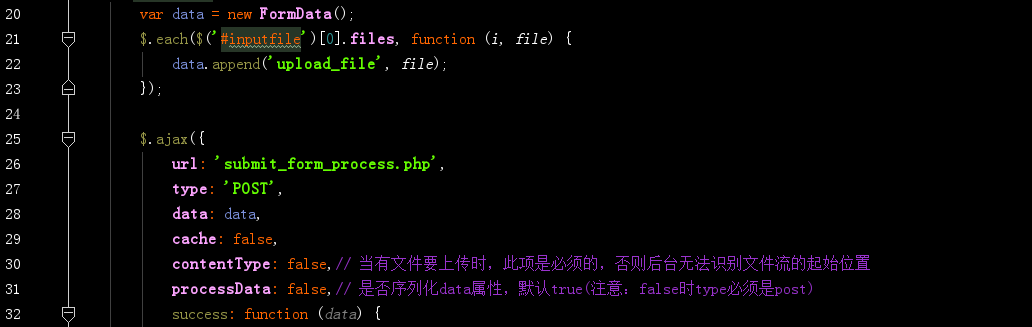
此外，HTML5还定义了一个progress事件，可以用来制作进度条。



如果上传的是图片文件，利用FileReader对象（ie10兼容）FileAPI实现图片预览。



结合jQuery，使用FormData。



* + 1. FormData

**1.概要**

（1）XHR2新增的1个接口，最大特点在于可以异步上传1个二进制文件。

（2）构造函数可以有1个表单入参。



**2.方法**

（1）set、append

添加/更新1个值，其中set为如果key不存在则新增，存在则替换。而append为如果key不存在则什么都不做，存在在append到值的最后。



其中，value可以为blob对象。

（2）get、getAll、keys、values、entries等

获取值。

（3）delete

删除值。

**3.事件**

（1）loadstart、loadend

上传开始、结束时调用。

（2）progress、abort

进度信息、中止。

* + 1. Blob

（1）Blob对象表示一个不可变、原始数据的类文件对象。Blob表示的不一定是JavaScript原生格式的数据。

（2）要从其他非blob对象和数据构造一个Blob，请使用Blob()构造函数。要创建包含另一个blob数据的子集blob，请使用slice()方法。

（3）构造函数：



其中，array是一个由ArrayBuffer、ArrayBufferView、Blob、DOMString（可以看做是string）等对象构成的Array。Options的值可以为type：表示数组内容的MIME类型；endings：表示\n如何写入，可以为transparent或native，默认值为transparent，表示保持blob中保存的结束符不变。



（4）属性有size：字节大小；type：MIME类型。

（5）方法只有1个，为slice：



Start表示新拷贝的起始字节位置，end结束位置，contentType为设置1个新的type类型。

（6）事件基本同FormData。

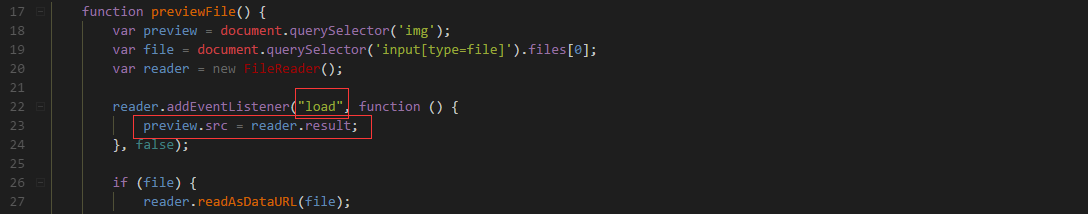
* + 1. fileReader

（1）兼容ie10。

（2）重要属性result，可以从中获取文件内容，前提是文件已上传完成。

（3）重要方法readAsArrayBuffer、readAsDataURL，读取指定的Blob或File对象。读取完成后，触发loadend事件，同时result属性将包含一个data:URL格式的字符串（base64编码）以表示所读取文件的内容。





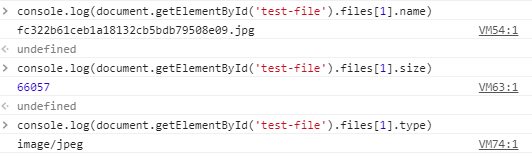
* + 1. 应用

**1.多文件上传**

（1）一直都可以多文件上传，只是h5新增了input[type="file"]的multiple属性，可以在1个input框中选多个文件。



（2）对于ie10以上浏览器，获取的input有files属性，可以获取文件的name、type等。



**2.图片预览**

见fileReader。

**3.文件拖拽上传**

（1）元素拖放

1）拖放是h5新增，兼容ie9，任何元素都支持拖放。

2）设置元素为可拖放：将draggable 属性设置为 true即可。



3）

* 1. 文件上传

**1.参考**

http://www.ruanyifeng.com/blog/2012/08/file\_upload.html

http://www.cnblogs.com/Zjmainstay/archive/2012/08/09/jQuery\_upload\_image.html

**2.<form>标签的enctype属性**

enctype属性规定在发送到服务器之前应该如何对表单数据进行编码。而跟url编码、Content-Type等没关系。只有3个值，如下：

（1）application/x-www-form-urlencoded和Content-Type

两者貌似没有明显区别，使用get方式提交时，把表单数据以键值对形式追加到url后。使用post方式提交时，把表单数据以键值对放在请求体中传输。

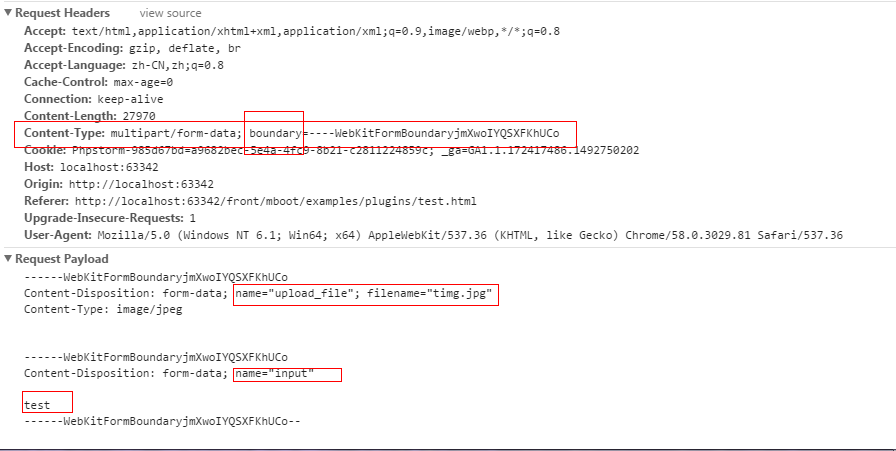




（2）multipart/form-data

表单数据被编码为一条消息，页上的每个<input>对应消息中的一个部分，用boundary（值由浏览器生成）分割各个部分。它不会对字符进行编码，一般用于传输二进制文件（图片、视频等）。





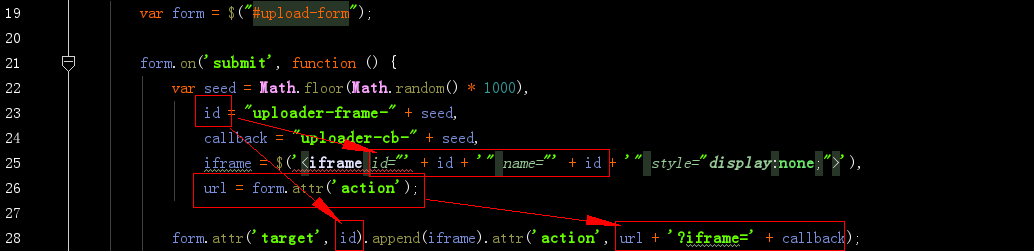
**3.原始文件上传**

即，直接使用表单元素input[type=file]。但上传文件不同于一般数据的POST，不能通过ajax方式，必须通过表单提交完成。

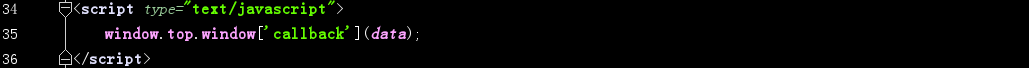
**4.利用iframe上传**

在HTML5没有出现之前，只能使用iframe模拟异步上传。需要做到：

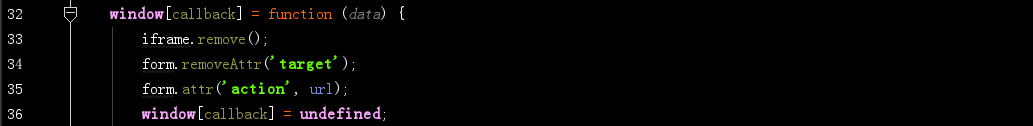
（1）用户点击submit时，动态插入一个iframe元素。该iframe元素的name=form的target。为了动态添加，可以在先不设置form的target，而是在代码中动态添加iframe和form一样的值。



（2）动态添加1个回调，附在form原来的action后面，类似于jsonp，需要服务器将数据传给这个回调，且返回的是1个script。



这个脚本会自动写入iframe并执行。因此，可以在原来的代码中定义执行成功回调。

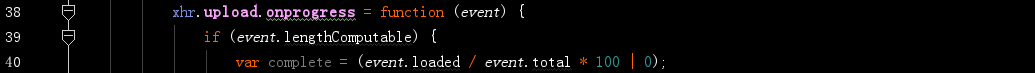


**5.ajax上传**

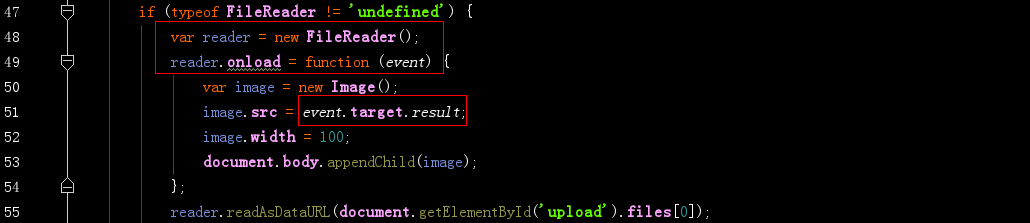
HTML5提出了第二版XMLHttpRequest对象（ie10支持），可以通过ajax上传文件。这是真正的异步上传。iframe上传可以用作老式浏览器的替代方案。



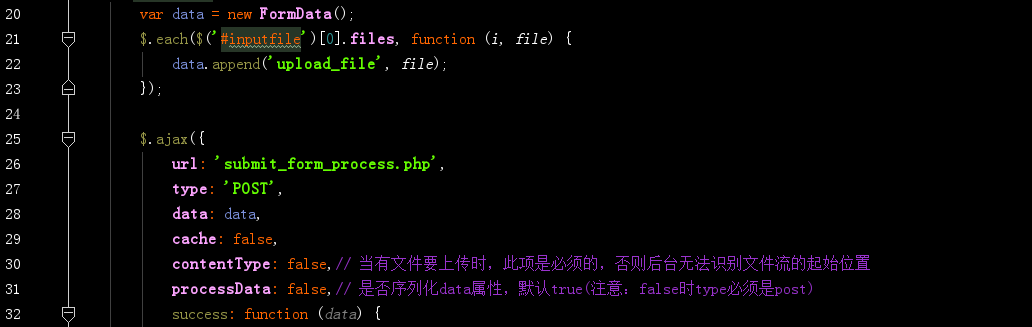
此外，HTML5还定义了一个progress事件，可以用来制作进度条。



如果上传的是图片文件，利用FileReader对象（ie10兼容）FileAPI实现图片预览。



结合jQuery，使用FormData。



* 1. worker

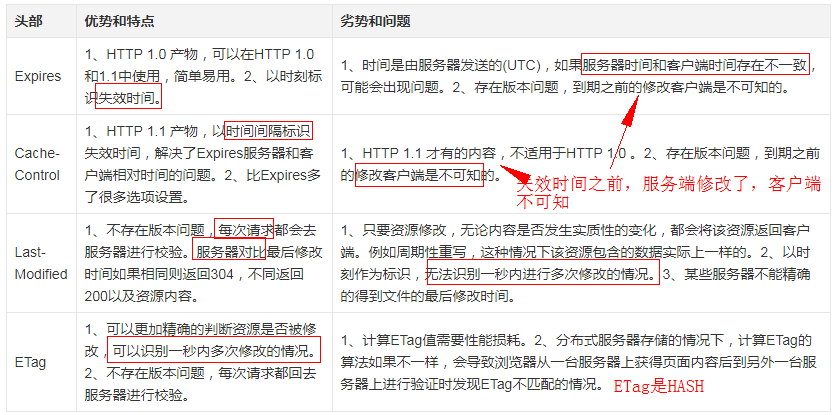
专用workers是指标准worker仅在单一脚本中被使用。共享worker的上下文是SharedWorkerGlobalScope对象。一个专用worker仅仅能被首次生成它的脚本使用，而共享worker可以同时被多个脚本使用。在专用workers的情况下，DedicatedWorkerGlobalScope对象代表了worker的上下文。

workers和主线程间的数据传递通过这样的消息机制进行——双方都使用postMessage()方法发送各自的消息，使用onmessage事件处理函数来响应消息（消息被包含在Message事件的data属性中）。这个过程中数据并不是被共享而是被复制。

可以调用worker的terminate方法立刻终止一个运行中的worker。

* 1. 缓存

服务器响应提供HTTP头指令，以指导浏览器何时可以缓存响应以及可以缓存多久。HTTP报文头部中与缓存相关的字段为：



**1.Pragma**

（1）1.0时代产物，已不再使用，除非向下兼容，优先级最高。

**2.Expires**

（1）1.0时代产物，对应一个GMT（格林尼治时间），用于控制客户端缓存过期时间。

（2）如果时间没有过期，则从缓存中读取，不会向服务端发送请求。

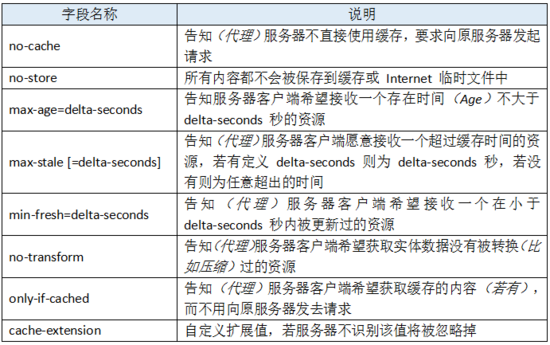
（3）存在客户端上的时间跟服务器上的时间不一致情况（客户端修改系统时间）。

**3.Cache-Control**

（1）针对客户端上的时间跟服务器上的时间不一致情况。

（2）优先级从高到低分别是Pragma->Cache-Control->Expires。

（3）是一个通用首部字段，能分别在请求报文和响应报文中使用。其中作为请求首部时，值可以为：



作为响应首部时，值可以为：



（4）上面的值还可以自由组合，如：



意为缓存（新鲜度）的有效时间为一小时，在后续一小时内，用户重新访问该资源则无须发送请求。

**4.缓存校验字段**

上面的配置都存在1个问题：如果时间没过期，则不会向服务端请求，即使服务端已经发生改变；如果时间过期，则服务端一定会返回内容，即使没有改变。为此，有如下两个字段：

（1）Last-Modified

即资源的最后修改时间。若传递的时间值与服务器上该资源最终修改时间是一致的，则说明该资源没有被修改过，直接返回304状态码，内容为空，这样就节省了传输数据量。

（2）ETag

Last-Modified可能存在的不准确的问题（1秒内多次修改）。ETag实体首部字段。服务器会通过某种算法，给资源计算得出一个唯一标志符（比如md5标志），在把资源响应给客户端的时候，会在实体首部加上“ETag:唯一标识符”一起返回给客户端。

