

一、概念

- DOMContentLoaded

当初始的 HTML 文档被完全加载和解析完成之后，DOMContentLoaded 事件被触发，而无需等待样式表、图像和子框架的完成加载。

- load

load 仅用于检测一个完全加载的页面，页面的html、css、js、图片等资源都已经加载完之后才会触发 load 事件。

二、浏览器的一些基本概念

- 下载/加载

浏览器将资源下载到本地的过程。

- 解析

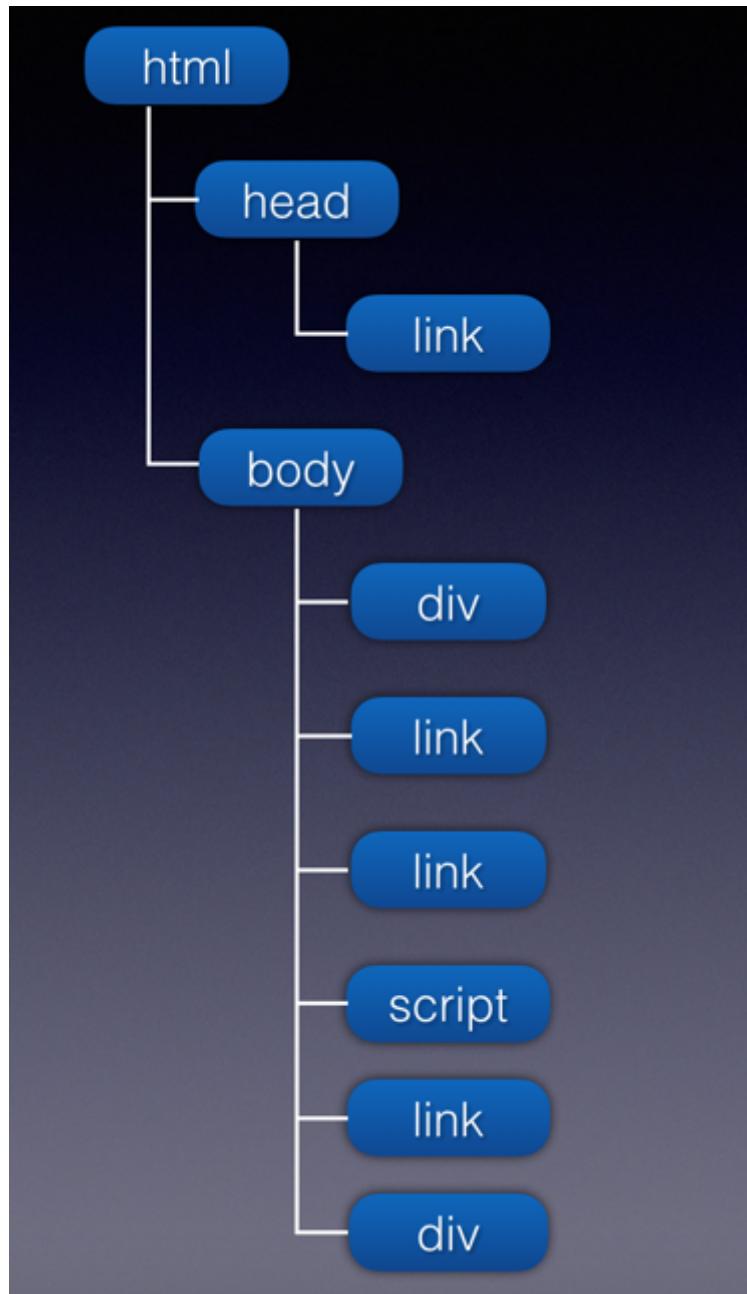
解析的意思是将一个元素通过一定的方式转换成另一种形式。

比如 html 的解析。首先要明确，html 下载到浏览器的表现形式就是包含字符串的文件。浏览器将 html 文件里面的字符串读取到内存中，按照 html 规则，对字符串进行取词编译，将字符串转化成另一种易于表达的数据结构。

比如下面的代码：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>只有css</title>
  <link rel="stylesheet" href="./index.css" />
</head>
<body>
  <div id="div1"></div>
  <link rel="stylesheet" href="./c1.css" />
  <link rel="stylesheet" href="./c3.css" />
  <script src="http://test.com:9000/mine/load/case2/j1.js
"></script>
  <link rel="stylesheet" href="./c4.css" />
  <div id="div2"></div>
</body>
</html>
```

浏览器会对这个 html 文件进行编译，转化成类似下面的结构：



浏览器会对转化后的数据结构自上而下进行分析：首先开启下载线程，对所有的资源进行优先级排序下载（注意，这里仅仅是下载）。同时主线程会对文档进行解析：

- 遇到 script 标签时，首先阻塞后续内容的解析，同时检查该 script 是否已经下载下来，如果已下载，便执行代码。
- 遇到 link 标签时，不会阻塞后续内容的解析（比如 DOM 构建），检查 link 资源是否已下载，如果已下载，则构建 cssom。
- 遇到 DOM 标签时，执行 DOM 构建，将该 DOM 元素添加到文档树中。

⚠在 body 中第一个 script 资源下载完成之前，浏览器会进行首次渲染，将该 script 标签前面的 DOM 树和 CSSOM 合并成一棵 Render 树，渲染到页面中。这是页面从白屏到首次渲染的时间节点。

- DOM 构建

将文档中的所有 DOM 元素构建成一个树型结构，DOM 构建是自上而下进行构建的，会受到 js 执行的干扰。

- CSS 构建

将文档中的所有 CSS 资源合并。

- render 树

将 DOM 树和 CSS 合并成一棵渲染树，render 树在合适的时机会被渲染到页面中。

三、HTML 文档的加载与页面的首次渲染

- 1、浏览器首先下载该地址所对应的 html 页面。
- 2、浏览器解析 html 页面的 DOM 结构。
- 3、开启下载线程对文档中的所有资源按优先级排序下载。
- 4、主线程继续解析文档，到达 head 节点，head 里的外部资源是外链样式表和外链 js。
 - 发现有外链 css 或者外链 js，如果是外链 js，则停止解析后续内容，等待该资源下载，下载完后立刻执行。如果是外链 css，继续解析后续内容。
- 5、解析到 body
 - 只有 DOM 元素
 - 这种情况比较简单了，DOM 树构建完，页面首次渲染。
 - 有 DOM 元素、外链 js
 - 当解析到外链 js 的时候，该 js 尚未下载到本地，则 js 之前的 DOM 会被渲染到页面上，同时 js 会阻止后面 DOM 的构建，即后面的 DOM 节点并不会添加到文档的 DOM 树中。所以，js 执行完之前，我们在页面上看不到该 js 后面的 DOM 元素。
 - 有 DOM 元素、外链 css
 - 外链 css 不会影响 css 后面的 DOM 构建，但是会阻碍渲染。简单点说，外链 css 加载完之前，页面还是白屏。
 - 有 DOM 元素、外链 js、外链 css
 - 外链 js 和外链 css 的顺序会影响页面渲染，这点尤为重要。当 body 中 js 之前的外链 css 未加载完之前，页面是不会被渲染的。
 - 当 body 中 js 之前的外链 css 加载完之后，js 之前的 DOM 树和 css 合并渲染树，页面渲染出该 js 之前的 DOM 结构。

6、文档解析完毕，页面重新渲染。当页面引用的所有 js 同步代码执行完毕，触发 DOMContentLoaded 事件。

7、html 文档中的图片资源，js 代码中有异步加载的 css、js、图片资源都加载完毕之后，load 事件触发。

如下代码所示：

```
<body>
<!-- 白屏 -->
<div id="div1"></div>
<!-- 白屏 -->
<link rel="stylesheet" href="./c1.css" />
<!-- 白屏 -->
<link rel="stylesheet" href="./c3.css" />
<!-- 如果此时 j1.js 尚未下载到本地，则首次渲染，此时的 DOM 树 只有 div1，所以页面上只会显示 div1，样式是 c1.css 和 c3.css 的并集。-->
<!-- 如果此时 j1.js 已经下载到本地，则先执行 j1.js，页面不会渲染，所以此时仍然是白屏。-->
<!-- 下面的 js 阻塞了 DOM 树的构建，所以下面的 div2 没有在文档的 DOM 树中。 -->
<script src="http://test.com:9000/mine/load/case2/j1.js"
```

```

"\></script\>
<!-- j1.js 执行完毕，继续 DOM 解析，div2 被构建在文档 DOM 树中，此时页面上有 div2 元素，样式仍然是 c1.css 和 c3.css 的并集 \-->
<link rel="stylesheet" href"./c4.css" />
<!-- c4.css 加载完毕，重新构建 render 树，样式变成了 c1.css、c3.css 和 c4.css 的并集 \-->
<div id="div2"></div>
<script>
// 利用 performance 统计 load 加载时间。
window.onload=function(){console.log(performance.timing.loadEventStart \-
performance.timing.fetchStart);}
</script>
</body>

```

四、DomContentLoaded 事件的触发

DOMContentLoaded 事件在 html 文档加载完毕，并且 html 所引用的内联 js、以及外链 js 的同步代码都执行完毕后触发。

五、load 事件的触发

当页面 DOM 结构中的 js、css、图片，以及 js 异步加载的 js、css、图片都加载完成之后，才会触发 load 事件。

1 遍历数组通常用for循环

ES5的话也可以使用forEach，ES5具有遍历数组功能的还有map、filter、some、every、reduce、reduceRight等，只不过他们的返回结果不一样。但是使用foreach遍历数组的话，使用break不能中断循环，使用return也不能返回到外层函数。

```

Array.prototype.method=function(){
    console.log(this.length);
}
var myArray=[1,2,4,5,6,7]
myArray.name="数组"
for (var index in myArray) {
    console.log(myArray[index]);
}

```

2 for in遍历数组的毛病

- 1.index索引为字符串型数字，不能直接进行几何运算
- 2.遍历顺序有可能不是按照实际数组的内部顺序
- 3.使用for in会遍历数组所有的可枚举属性，包括原型。例如上栗的原型方法method和name属性所以for in更适合遍历对象，不要使用for in遍历数组。

那么除了使用for循环，如何更简单的正确的遍历数组达到我们的期望呢（即不遍历method和name），ES6中的for of更胜一筹。

```
Array.prototype.method=function(){
    console.log(this.length);
}
var myArray=[1,2,4,5,6,7]
myArray.name="数组";
for (var value of myArray) {
    console.log(value);
}
```

记住，*for in*遍历的是数组的索引（即键名），而*for of*遍历的是数组元素值。

*for of*遍历的只是数组内的元素，而不包括数组的原型属性method和索引name

3 遍历对象

遍历对象通常用*for in*来遍历对象的键名

```
Object.prototype.method=function(){
    console.log(this);
}
var myObject={
    a:1,
    b:2,
    c:3
}
for (var key in myObject) {
    console.log(key);
}
```

*for in*可以遍历到myObject的原型方法method,如果不想遍历原型方法和属性的话，可以在循环内部判断一下,hasOwnProperty方法可以判断某属性是否是该对象的实例属性

```
for (var key in myObject) {
    if (myObject.hasOwnProperty(key)){
        console.log(key);
    }
}
```

同样可以通过ES5的Object.keys(myObject)获取对象的实例属性组成的数组，不包括原型方法和属性

```
Object.prototype.method=function(){
    console.log(this);
}
var myObject={
    a:1,
    b:2,
    c:3
}
```

总结

- for..of适用遍历数/数组对象/字符串/map/set等拥有迭代器对象的集合.但是不能遍历对象,因为没有迭代器对象.与forEach()不同的是,它可以正确响应break、continue和return语句
- for-of循环不支持普通对象,但如果你想迭代一个对象的属性,你可以用for-in循环(这也是它的本职工作)或内建的Object.keys()方法:

```
for (var key of Object.keys(someObject)) {  
    console.log(key + ": " + someObject[key]);  
}
```

- 遍历map对象时适合用解构,例如;

```
for (var [key, value] of phoneBookMap) {  
    console.log(key + "'s phone number is: " + value);  
}
```

- 当你为对象添加myObject.toString()方法后,就可以将对象转化为字符串,同样地,当你向任意对象添加myObject`Symbol.iterator`方法,就可以遍历这个对象了。

举个例子,假设你正在使用jQuery,尽管你非常钟情于里面的.each()方法,但你还是想让jQuery对象也支持for-of循环,你可以这样做:

```
jQuery.prototype[Symbol.iterator] = Array.prototype[Symbol.iterator];
```

所有拥有`Symbol.iterator`的对象被称为可迭代的。在接下来的文章中你会发现,可迭代对象的概念几乎贯穿于整门语言之中,不仅是for-of循环,还有Map和Set构造函数、解构赋值,以及新的展开操作符。

- for...of的步骤
or-of循环首先调用集合的`Symbol.iterator`方法,紧接着返回一个新的迭代器对象。迭代器对象可以是任意具有.next()方法的对象;for-of循环将重复调用这个方法,每次循环调用一次。举个例子,这段代码是我能想出来的最简单的迭代器:

```
var zeroesForeverIterator = {  
    [Symbol.iterator]: function () {  
        return this;  
    },  
    next: function () {  
        return {done: false, value: 0};  
    }  
};
```

本文转自 <https://www.jianshu.com/p/c43f418d6bf0>, 如有侵权,请联系删除。浅谈移动端的click事件被tap事件代替的原因

1. 移动端点击事件click出现延迟

工作中接触了移动端，发现同事们都会用如下代码去写移动端的点击事件，尝试使用，屡试不爽，一旦没有用下边这段代码，点击事件就会出现各种各样的问题，在连续使用了N多次之后（本人并没有爱钻研的精神~~有点儿懒），终于决定自己上网查一些资料，看看到底是什么原因吧。（解释jQuery的方法**data()**: 在匹配元素上存储任意相关数据 或 返回匹配的元素集合中的第一个元素的给定名称的数据存储的值。**trigger()**: 规定被选元素要触发的事件。）

```
//自定义tap
$(document).on("touchstart", function(e) {
    if(!$(e.target).hasClass("disable")) $(e.target).data("isMoved", 0);
});
$(document).on("touchmove", function(e) {
    if(!$(e.target).hasClass("disable")) $(e.target).data("isMoved", 1);
});
$(document).on("touchend", function(e) {
    if(!$(e.target).hasClass("disable") && $(e.target).data("isMoved") == 0)
        $(e.target).trigger("tap");
});
```

2. 问题出在哪了

为什么要用tap事件代替click事件？答案：300毫秒延迟

这要追溯至2007年初。苹果公司在发布首款iPhone前夕，遇到一个问题——当时的网站都是为大屏幕设备所设计的。于是苹果的工程师们做了一些约定，应对iPhone这种小屏幕访问电脑版的网页的问题。这其中最出名的，当属双击缩放(double tap to zoom)。

当用户一次点击屏幕之后，浏览器并不能立刻判断用户是要进行双击缩放，还是想要进行单击操作。因此，iOS Safari就等待300毫秒，以判断用户是否再次点击了屏幕。

于是，300毫秒延迟就这么诞生了。

鉴于iPhone的成功，其他移动浏览器都复制了iPhone Safari浏览器的多数约定，包括双击缩放，几乎现在所有的移动端浏览器都有这个功能（嗯~尝试了一下好像微信浏览器没有再遵守这个双击缩放的约定）。之前人们刚刚接触移动端的页面，在欣喜的时候往往不会care这个300ms的延时问题，可是如今移动端界面如雨后春笋，用户对体验的要求也更高，这300ms带来的卡顿慢慢变得让人难以接受。

3. 实例操作300ms出现的过程

一开始触摸事件touchstart、touchmove和touchend是[iOS](#)版Safari[浏览器](#)为了向开发人员传达一些信息新添加的事件。因为iOS设备既没有鼠标也没有键盘，所以在为移动Safari浏览器开发交互性网页的时候，PC端的鼠标和键盘事件是不够用的。

在iPhone 3Gs发布的时候，其自带的移动Safari浏览器就提供了一些与触摸(touch)操作相关的新事件。随后，[Android](#)上的浏览器也实现了相同的事件（大家统一了，移动端浏览器都有了touch事件啦）。触摸事件(touch)会在用户手指放在屏幕上面的时候、在屏幕上滑动的时候或者是从屏幕上移开的时候触发。下面具体说明：

touchstart事件：当手指触摸屏幕时候触发，即使已经有一个手指放在屏幕上也会触发。

touchmove事件：当手指在屏幕上滑动的时候连续地触发。这个事件发生期间，调用preventDefault()事件可以阻止滚动。

touchend事件：当手指从屏幕上离开的时候触发。

touchcancel事件：当系统停止跟踪触摸的时候触发。关于这个事件的确切时间，文档中并没有具体说明，这里不再详述。

touch事件及click事件同时绑定在一个元素上demo: (一言不合上代码~~e.type指代事件类型)

```
$(document).on('touchstart', '.wantUp', function(e) {  
    $(".upArtical1 h5.up").append(e.type);  
})  
$(document).on('touchmove', '.wantUp', function(e) {  
    $(".upArtical1 h5.up").append(e.type);  
})  
$(document).on('touchend', '.wantUp', function(e) {  
    $(".upArtical1 h5.up").append(e.type);  
})  
$(document).on('click', '.wantUp', function(e) {  
    $(".upArtical1 h5.up").append(e.type);  
});
```

可以看到，我在一个元素上边绑定了 touchstart , touchmove , touchend , click事件，奇妙的事情发生了：

- touch, click事件的执行顺序: touchstart > touchmove > touchend > click。很明显，touch事件执行完毕后才会到click事件，这就是我们所说的300ms的延迟。
- touchmove , click事件互斥,即touchmove触发执行, click事件不再执行，事件的执行顺序就为 touchstart > touchmove (可以多次执行) > touchend 。
- 短暂触摸 (点) 一下屏幕，上述代码的事件的执行顺序: touchstart > touchend > click。

OK，到目前为止我们已经阐述了300ms的产生，而这300ms的产生还带来一个巨大的问题，点透事件。

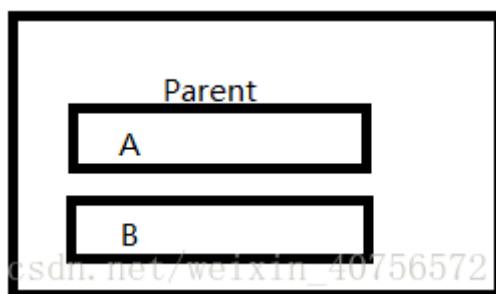
4. 点透事件

点透发生的条件:

- A 和 B不是后代继承关系(如果是后代继承关系的话，就直接是冒泡之类的话题了 (在此不再赘述))
- A发生touch(也可以是click)后立即消失,B事件绑定click
- A z-index大于B，即A显示在B浮层之上

点透发生的原因:

当手指触摸到屏幕的时候，系统生成两个事件，一个是touch 一个是click，touch先执行，touch执行完成后，A从文档树上面消失了，而且由于移动端click还有延迟200-300ms的关系，当系统要触发click的时候，发现在用户点击的位置上面，目前离用户最近的元素是B，所以就直接把click事件作用在B元素上面了



```
A.addEventListener('touch', function(e) {
    A.style.display = 'none';
});
A.onclick = function() {
    console.log('B莫名被点击了');
}
```

解决方案是touch阶段取消掉 click 事件： touch事件内调用： e.preventDefault()

5. 解决方案

以上问题的解决方案就是：既然浏览器统一了touch事件，就用touch事件去模拟click事件。

还记得文章最开始的那段代码吗？

```
//自定义tap
$(document).on("touchstart", function(e) {
    if(!$(e.target).hasClass("disable")) $(e.target).data("isMoved", 0);
});
$(document).on("touchmove", function(e) {
    if(!$(e.target).hasClass("disable")) $(e.target).data("isMoved", 1);
});
$(document).on("touchend", function(e) {
    if(!$(e.target).hasClass("disable") && $(e.target).data("isMoved") == 0)
        $(e.target).trigger("tap");
});
```

解释一下吧：现在原理就很明显了~

基于touchstart、touchmove、touchend这三个事件，通过事件委托的方式来实现tap事件。

e.target是事件源的触发节点，\$(e.target)是该节点的jQuery封装对象，isMoved这个就相当于一个开关，移动了置为1，没移动置为0。

第一步：监听touchstart事件，事件触发后通过jQuery的数据方法设置该对象的isMoved状态为0。

第二步：监听touchmove事件，事件触发后通过jQuery的数据方法设置该对象的isMoved状态为1。

第三步：监听touchend事件，事件触发后判断该对象是否touchMove过，没有则触发tap事件。

如何算tap事件：手指点上去 不移动 快速松开。

6. 如何使用这段代码？

- 复制上述代码到页面js处
- 对该使用click事件的元素统一换成tap事件

本文转自 https://blog.csdn.net/weixin_40756572/article/details/81776615，如有侵权，请联系删除。OK，排序这一个篇章也快要结束了。

这一篇主要说的是快速排序，说的方式主要还是先说原理，然后再用代码来进行实现。



所谓快速排序，就是分为三步走：

第一步：选择第一个数字分离出来为基数

第二步：然后将序列中大于基数的放在基数右边，小于基数的放在基数的左边

第三步：然后对基数的左边和右边两个序列重复第二步和第三步

这样就能形成一个有序的序列，那么重点是我们如何来实现第二步这个过程呢？

我们用图解来说明一下。



倘若我们有这样一个序列

4	5	8	9	1	0	3	2	7	6
---	---	---	---	---	---	---	---	---	---

https://blog.csdn.net/weixin_46726346

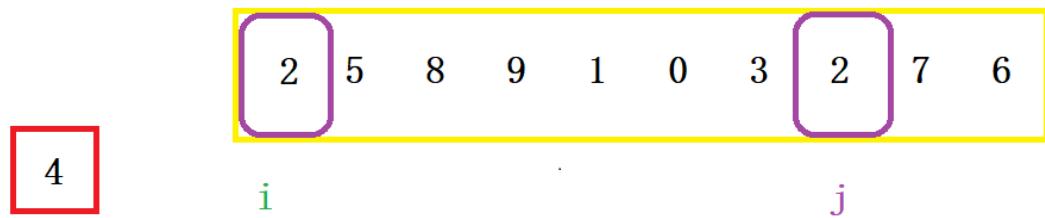
现在我们按步骤来，先把第一个元素4储存起来。

然后定义两个指针分别指向第一个元素和最后一个元素。

4	5	8	9	1	0	3	2	7	6
4	i							j	

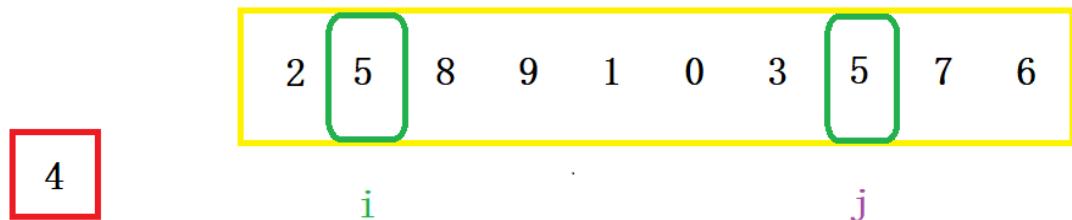
https://blog.csdn.net/weixin_46726346

然后我们让j先移动（记住一定是j先移动），一旦碰到比基数小的我们就停止移动，并且把j位置的值赋给i位置上的值。



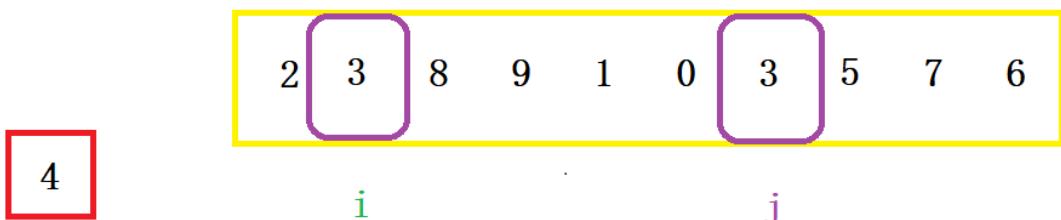
https://blog.csdn.net/weixin_46726346

好现在j停下来，我们让i移动，一旦碰到比基数大的值，就把i位置上的值赋给j位置上的值。



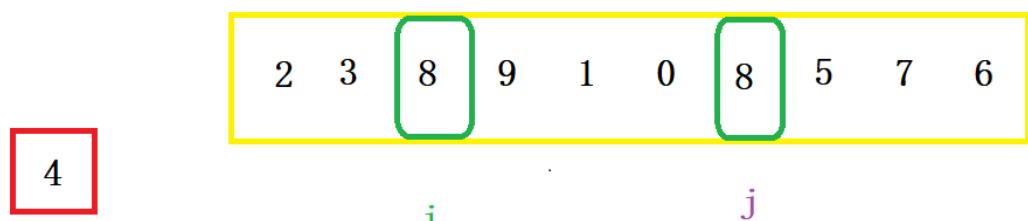
https://blog.csdn.net/weixin_46726346

OK，现在i停下来，j移动。



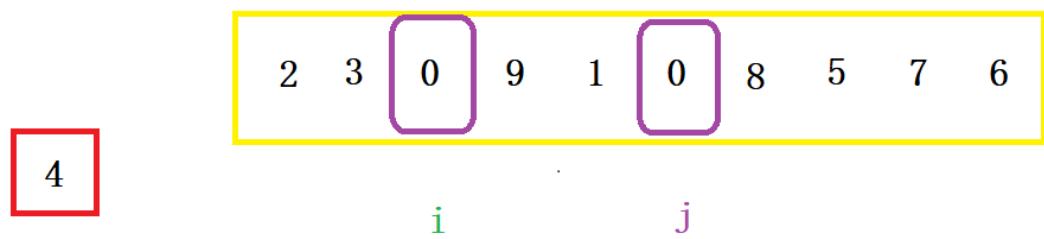
https://blog.csdn.net/weixin_46726346

j停下来，i移动



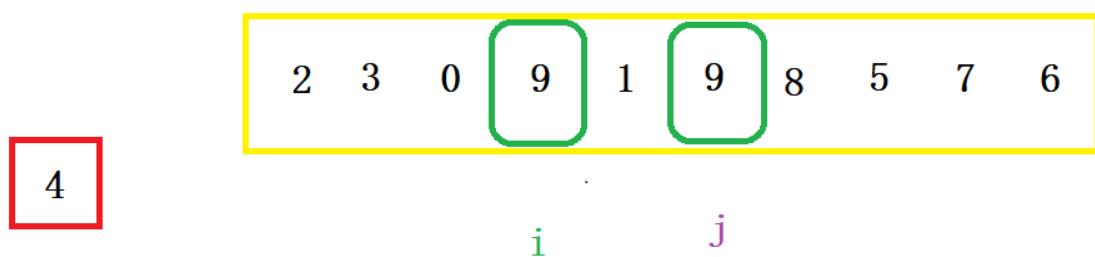
https://blog.csdn.net/weixin_46726346

i停下来，j移动



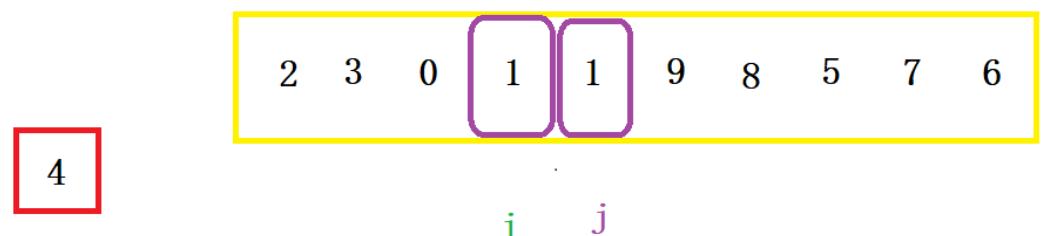
j停下来，*i*移动

https://blog.csdn.net/weixin_46726346



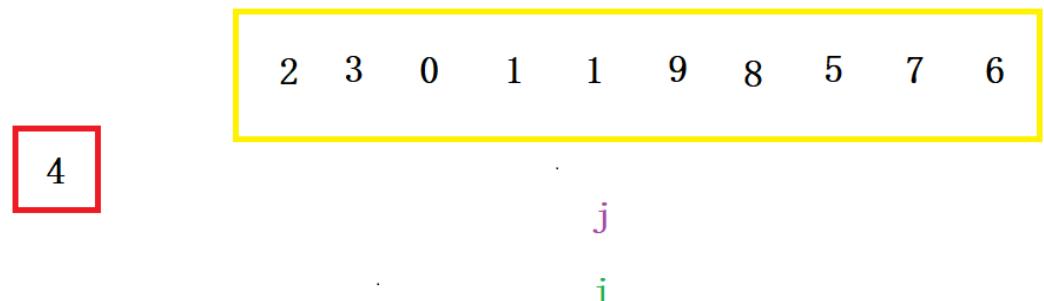
*i*停下来，*j*移动

https://blog.csdn.net/weixin_46726346



然后最后*i*再移动一下

https://blog.csdn.net/weixin_46726346



此时*i*和*j*所处的位置是一样的

这个时候我们把基数放在*i*和*j*共同的位置上就可以了。

https://blog.csdn.net/weixin_46726346

2 3 0 1 4 9 8 5 7 6

j

i

https://blog.csdn.net/weixin_46726346

这样就能做到基数4左边都是比自己小的，右边都是比自己大的。

这样这个基数的位置就找到了。

现在我们再对基数左边的序列和右边的序列重复上面的步骤，就可以得到一个有序的序列啦。



现在我们用代码来实现一下

```
<script>
    function sort(arr,begin,end){
        if(begin < end){
            let i = begin;
            let j = end;
            let empty = arr[begin];
            while(i < j){
                while(arr[j] > empty && i < j){
                    j--;
                }
                arr[i] = arr[j];
                while(arr[i] < empty && i < j){
                    i++;
                }
                arr[j] = arr[i];
            }
            arr[i] = empty;
            sort(arr,begin,i-1);
            sort(arr,i+1,end);
        }else{
            return;
        }
    }

    let arr = [2,3,1,4,8,7,9,6];
    sort(arr,0,7);
    console.log(arr);
</script>
```

OK，关于快速排序也就说完了。

本文转自 https://blog.csdn.net/weixin_46726346/article/details/115839126, 如有侵权, 请联系删除。简单区分

总的来说 `null` 和 `undefined` 都代表空, 主要区别在于 `undefined` 表示尚未初始化的变量的值, 而 `null` 表示该变量有意缺少对象指向。

- `undefined`
- 这个变量从根本上就没有定义
- 隐藏式空值
- `null`
- 这个值虽然定义了, 但它并未指向任何内存中的对象
- 声明式空值

MDN 中给出的定义

`null`

值 `null` 是一个字面量, 不像 `undefined`, 它不是全局对象的一个属性。`null` 是表示缺少的标识, 指示变量未指向任何对象。把 `null` 作为尚未创建的对象, 也许更好理解。在 API 中, `null` 常在返回类型应是一个对象, 但没有关联的值的地方使用。

`undefined`

`undefined` 是 全局对象 的一个属性。也就是说, 它是全局作用域的一个变量。`undefined` 的最初值就是原始数据类型 `undefined`。

一张神奇的图片

接下来我们看一张比较经典的图片, 该图来自 stackoverflow 的回答, 本人没有找到准确的出处。

Non-zero value



`null`

0



`undefined`



知乎 @Thomas Lin

表现形式

在更深入理解 `null` 和 `undefined` 的区别前，我们首先要知道 `null` 和 `undefined` 在 JS 中有什么不同的表现形式，用以方便我们更好的理解 `null` 和 `undefined` 的区别。

typeof

```
typeof null // 'object'  
typeof undefined // 'undefined'  
复制代码
```

Object.prototype.toString.call

```
typeof null // '[object Null]'  
typeof undefined // '[object Undefined]'  
复制代码
```

\== 与 ===

```
null == undefined // true  
null === undefined // false  
!!null === !!undefined // true  
复制代码
```

Object.getPrototypeOf(Object.prototype)

JavaScript 中第一个对象的原型指向 `null`。

```
Object.getPrototypeOf(Object.prototype) // null  
复制代码
```

+ 运算与 Number()

```
let a = undefined + 1 // NaN  
let b = null + 1 // 1  
Number(undefined) // NaN  
Number(null) // 0  
复制代码
```

JSON

```
JSON.stringify({a: undefined}) // '{}'  
JSON.stringify({b: null}) // '{b: null}'  
JSON.stringify({a: undefined, b: null}) // '{b: null}'  
复制代码
```

```
let undefiend = 'test'
```

```
function test(n) {  
    let undefined = 'test'  
    return n === undefined  
}  
  
test() // false  
test(undefined) // false  
test('test') // ture  
  
let undefined = 'test' // Uncaught SyntaxError: Identifier 'undefined' has  
already been declared  
复制代码
```

深入探索

为什么 `typeof null` 是 `object`?

`typeof null` 输出为 `'object'` 其实是一个底层的错误，但直到现阶段都无法被修复。

原因是，在 `JavaScript` 初始版本中，值以 `32位` 存储。前 `3位` 表示数据类型的标记，其余位则是值。

对于所有的对象，它的前 `3位` 都以 `000` 作为类型标记位。在 `JavaScript` 早期版本中，`null` 被认为是一个特殊的值，用来对应 `C` 中的 `空指针`。但 `JavaScript` 中没有 `C` 中的指针，所以 `null` 意味着什么都没有或者 `void` 并以 `全0(32个)` 表示。

因此每当 `JavaScript` 读取 `null` 时，它前端的 `3位` 将它视为 `对象类型`，这也是为什么 `typeof null` 返回 `'object'` 的原因。

为什么 `Object.prototype.toString.call(null)` 输出 `'[object Null]'`

`toString()` 是 `Object` 的原型方法，调用该方法，默认返回当前对象的 `[[Class]]`。这是一个内部属性，其格式为 `[object xxx]`，其中 `xxx` 就是对象的类型。

JavaScript 万物皆对象，为什么 `xxx.toString()` 不能返回变量类型？

这是因为各个类中重写了 `toString` 的方法，因此需要调用 `Object` 中的 `toString` 方法，必须使用 `toString.call()` 的方式调用。

对于 `Object` 对象，直接调用 `toString()` 就能返回 `'[object Object]'`。而对于其他对象，则需要通过 `call / apply` 来调用才能返回正确的类型信息。

为什么 == 和 === 对比会出现 true 和 false ?

很多文章说: `undefined` 的布尔值是 `false` , `null` 的布尔值也是 `false` , 所以它们在比较时都转化为了 `false` , 所以 `undefined == null` 。

实际上并不是这样的。

ECMA 在 11.9.3 章节中明确告诉我们:

1. If `x` is `null` and `y` is `undefined`, return `true`.
2. If `x` is `undefined` and `y` is `null`, return `true`.

这是 `JavaScript` 底层的内容了, 至于更深入的内容, 如果有兴趣可以扒一扒 `JavaScript` 的源码。

为什么 `null + 1` 和 `undefined + 1` 表现不同?

这涉及到 `JavaScript` 中的隐式类型转换, 在执行 `加法运算` 前, 隐式类型转换会尝试将表达式中的变量转换为 `number` 类型。如: `'1' + 1` 会得到结果 `11`。

- `null` 转化为 `number` 时, 会转换成 `0`
- `undefined` 转化为 `number` 时, 会转换为 `Nan`

至于为什么执行如此的转换方式, 我猜测是 `JavaScript` 早期的一个糟糕设计。

从语言学的角度来看:

`null` 意味着一个明确的没有指向的空值, 而 `undefined` 则意味着一个未知的值。

在某种程度上, `0` 意味着数字空值。

这虽然看起来有些牵强, 但是我在这一阶段能所最能想到的可能了。

为什么 `JSON.stringify` 会将值为 `undefined` 的内容删除?

其实这条没有很好的解释方式, `JSON` 会将 `undefined` 对应的 key 删除, 这是 `JSON` 自身的转换原则。

在 `undefined` 的情况下, 有无该条数据是没有区别的, 因为他们在表现形式上并无不同:

```
let obj1 = { a: undefined }
let obj2 = {}

console.log(obj1.a) // undefined
console.log(obj2.a) // undefined
```

复制代码

但需要注意的是, 你可能在调用接口时, 需要对 `JSON` 格式的数据中的 `undefined` 进行特殊处理。

为什么 `let undefined = 'test'` 可以覆盖掉 `JavaScript` 自身的 `undefined`?

`JavaScript` 对于 `undefined` 的限制方式为全局创建了一个只读的 `undefined` , 但是并没有彻底禁止局部 `undefined` 变量的定义。

据说在 `JavaScript` 高版本禁止了该操作, 但我没有准确的依据。

请在任何时候, 都不要进行 `undefined` 变量的覆盖, 就算是你的 `JSON` 转换将 `undefined` 转换为 `''` 。也不要通过该操作进行, 这将是及其危险的行为。

总结

关于使用 undefined 还是 null

这是一条公说公有理婆说婆有理的争议内容。

本人更倾向于使用 `null`，因为这是显示定义空值的方式。我并不能给出准确的理由。

但关于使用 `undefined` 我有一条建议：

如果你需要使用 `undefined` 定义空值，请不要采取以下两种方式：

- `let a;`
- `let a = undefined;`

进而采取下面这种方式显式声明 `undefined`：

- `let a = void 0;`

转载自：<https://juejin.cn/post/7051144396615450655>

service worker 是什么

一句话概括

一个服务器与浏览器之间的中间人角色，如果网站中注册了service worker那么它可以拦截当前网站所有的请求，进行判断（需要编写相应的判断程序），如果需要向服务器发起请求的就转给服务器，如果可以直接使用缓存的就直接返回缓存不再转给服务器。从而大大提高浏览体验。

以下是一些细碎的描述

- 基于web worker（一个独立于JavaScript主线程的独立线程，在里面执行需要消耗大量资源的操作不会堵塞主线程）
- 在web worker的基础上增加了离线缓存的能力
- 本质上充当Web应用程序（服务器）与浏览器之间的代理服务器（可以拦截全站的请求，并作出相应的动作->由开发者指定的动作）
- 创建有效的离线体验（将一些不常更新的内容缓存在浏览器，提高访问体验）
- 由事件驱动的，具有生命周期
- 可以访问cache和indexDB
- 支持推送
- 并且可以让开发者自己控制管理缓存的内容以及版本

如何使用

1. 注册Service worker 在你的index.html加入以下内容

```
/* 判断当前浏览器是否支持serviceworker */
if ('serviceworker' in navigator) {
    /* 当页面加载完成就创建一个serviceworker */
    window.addEventListener('load', function () {
        /* 创建并指定对应的执行内容 */
        /* scope 参数是可选的，可以用来指定你想让 service worker 控制的内容的子目录。
        在这个例子里，我们指定了 '/', 表示 根网域下的所有内容。这也是默认值。 */
        navigator.serviceWorker.register('./serviceworker.js', {scope: './'})
```

```

        .then(function (registration) {
            console.log('ServiceWorker registration successful with scope: ', registration.scope);
        })
        .catch(function (err) {
            console.log('ServiceWorker registration failed: ', err);
        });
    });
}

```

1. 安装worker: 在我们指定的处理程序 `serviceworker.js` 中书写对应的安装及拦截逻辑

```

/* 监听安装事件，install 事件一般是被用来设置你的浏览器的离线缓存逻辑 */
this.addEventListener('install', function (event) {
    /* 通过这个方法可以防止缓存未完成，就关闭serviceworker */
    event.waitUntil(
        /* 创建一个名叫v1的缓存版本 */
        caches.open('v1').then(function (cache) {
            /* 指定要缓存的内容，地址为相对于跟域名的访问路径 */
            return cache.addAll([
                './index.html'
            ]);
        })
    );
});

/* 注册fetch事件，拦截全站的请求 */
this.addEventListener('fetch', function(event) {
    event.respondWith(
        // magic goes here

        /* 在缓存中匹配对应请求资源直接返回 */
        caches.match(event.request)
    );
});

```

以上为一个最简单的使用例子，更多内部api请查看[mdn service worker](#)

注意事项

Service worker运行在worker上下文 --> 不能访问DOM

它设计为完全异步，同步API（如XHR和localStorage）不能在service worker中使用

出于安全考量，Service workers只能由HTTPS承载

在Firefox浏览器的用户隐私模式，Service Worker不可用

其生命周期与页面无关（关联页面未关闭时，它也可以退出，没有关联页面时，它也可以启动）

[兼容情况](#)

有趣的事

在MDN的兼容情况中可以看到Safari对于Service workers的全线不支持，这是因为通过Service workers可以在浏览器上实现一种类似小程序的功能（PWA）。这将绕过苹果的app store导致苹果不能再和开发者37开分成，所以苹果不喜欢这项技术。ps:不过还是在18年开始支持了

本文转自 <https://zhuanlan.zhihu.com/p/115243059>，如有侵权，请联系删除。

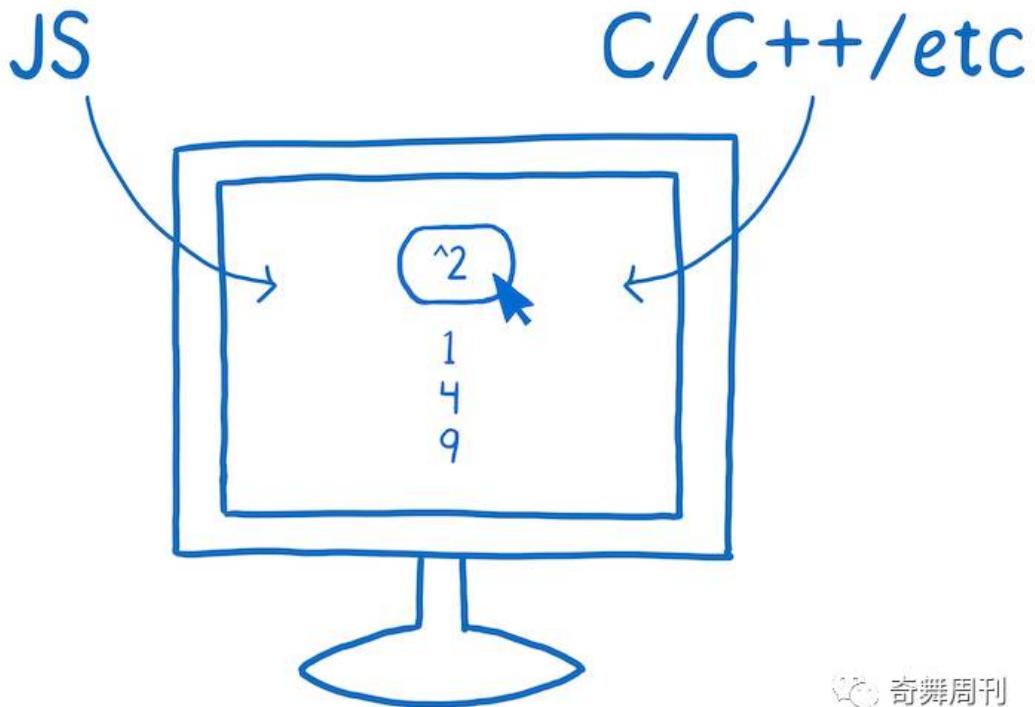


(图片来源: giphy.com)

编者按：本文由明非在众成翻译平台上翻译。

最近，WebAssembly 在 JavaScript 圈非常的火！人们都在谈论它多么多么快，怎样怎样改变 Web 开发领域。但是没有人讲他到底为什么那么快。在这篇文章里，我将会帮你了解 WebAssembly 到底为什么那么快。

第一，我们需要知道它到底是什么！WebAssembly 是一种可以使用非 JavaScript 编程语言编写代码并且能在浏览器上运行的技术方案。



 奇舞周刊

当大家谈论起 WebAssembly 时，首先想到的就是 JavaScript。现在，我没有必须在 WebAssembly 和 JavaScript 中选一个的意思。实际上，我们期待开发者在一个项目中把 WebAssembly 和 JavaScript 结合使用。但是，比较这两者是有用的，这对你了解 WebAssembly 有一定帮助。

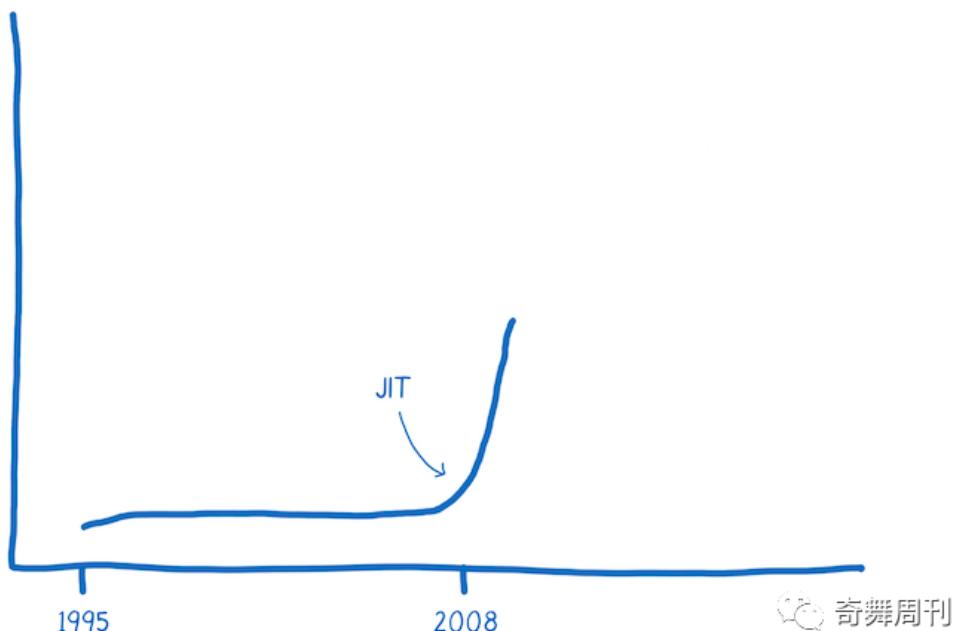
1. 一点点性能历史

1995 年 JavaScript 诞生。它的设计时间非常短，前十年发展迅速。

紧接着浏览器厂商们就开始了更多的竞争。

2008年，人们称之为浏览器性能大战的时期开始了。很多浏览器加入了即时编译器，又称之为JITs。在这种模式下，JavaScript在运行的时候，JIT 选择模式然后基于这些模式使代码运行更快。

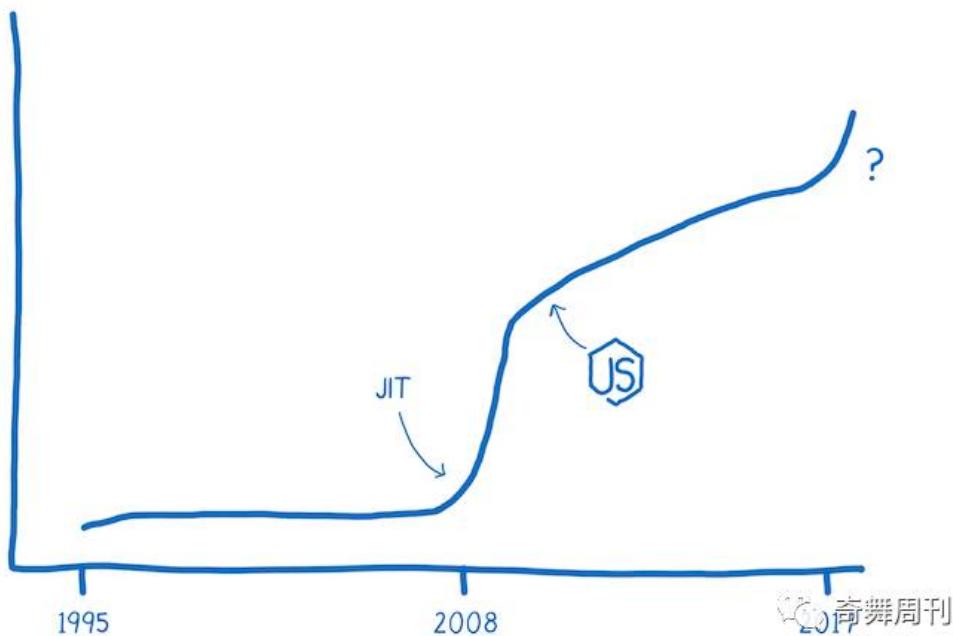
这些 JITs 的引入是浏览器运行代码机制的一个转折点。所有的突然之间，JavaScript 的运行速度快了10倍。



 奇舞周刊

随着这种改进的性能，JavaScript 开始被用于意想不到的事情，比如使用Node.js和Electron构建应用程序。

现在 WebAssembly 可能是的另一个转折点。



在我们没有搞清楚 JavaScript 和 WebAssembly 之间的性能差前，我们需要理解 JS 引擎所做的工作。

2. JavaScript 是如何在浏览器中运行的呢？

作为一个开发人员，您将JavaScript添加到页面时，您有一个目标并遇到一个问题。

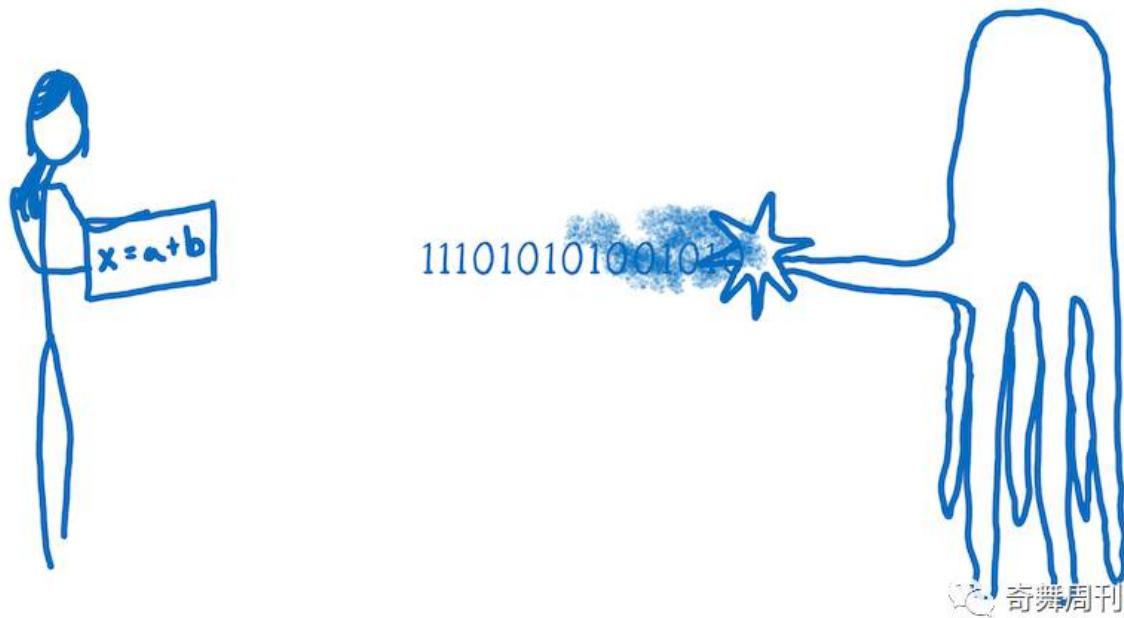
目标：你想要告诉计算机做什么

问题：你和计算机使用不通的语言。

您说的是人类的语言，计算机说的是机器语言。尽管你不认为 JavaScript 或者其他高级语言是人类语言，但事实就是这样的。它们的设计是为了让人们认知，不是为机器设计的。

所以JavaScript引擎的工作就是把你的人类语言转化成机器所理解的语言。

我想到电影《Arrival》，这就像人类和外星人进行交谈。

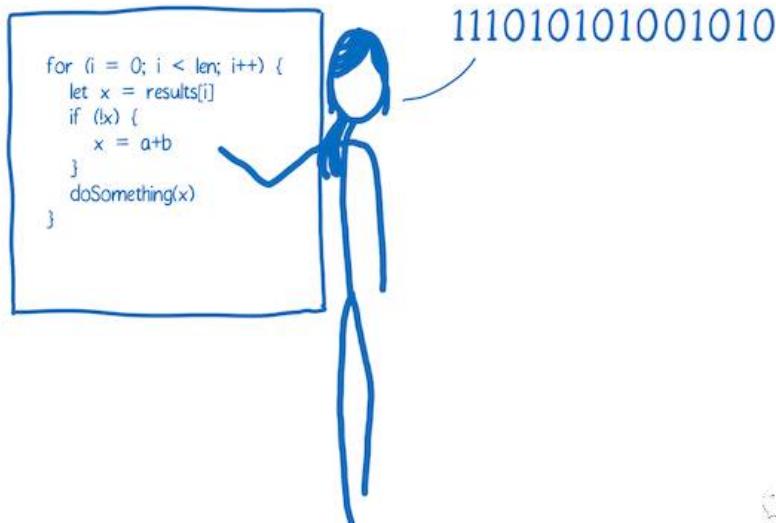


在这部电影中，人类语言不能从逐字翻译成外星语言。他们的语言反映出两种对世界不同的认知。人类和机器也是这样。

所以，怎么进行翻译呢？

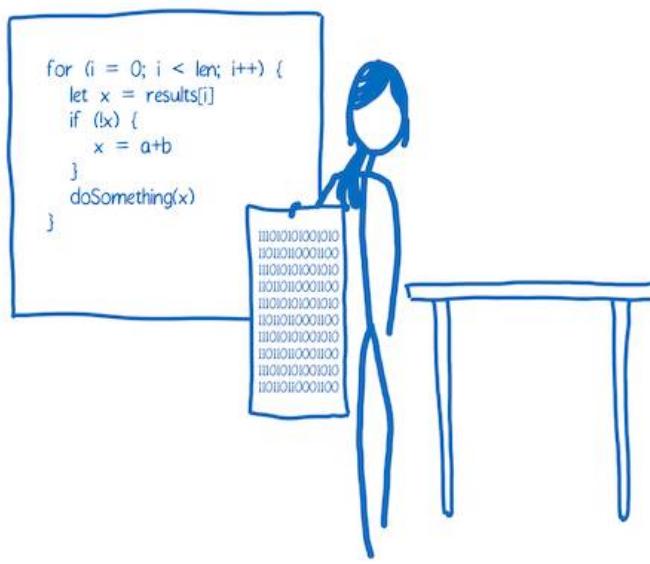
在编程中，通常有两种翻译方法将代码翻译成机器语言。你可以使用解释器或者编译器。

使用解释器，翻译的过程基本上是一行一行及时生效的。



奇舞周刊

编译器是另外一种工作方式，它在执行前翻译。



奇舞周刊

每种翻译方法都有利弊。

2.1 解释器的利弊

解释器很快的获取代码并且执行。您不需要在您可以执行代码的时候知道全部的编译步骤。因此，解释器感觉与 JavaScript 有着自然的契合。web 开发者能够立即得到反馈很重要。

这也是浏览器最开始使用 JavaScript 解释器的原因之一。

但是实用解释器的弊端是当你运行相同的代码的时候。比如，你执行了一个循环。然后你就会一遍又一遍的做同样的事情。

2.2 编译器的利弊

编译器则有相反的效果。在程序开始的时候，它可能需要稍微多一点的时间来了解整个编译的步骤。但是当运行一个循环的时候他会更快，因为他不需要重复的去翻译每一次循环里的代码。

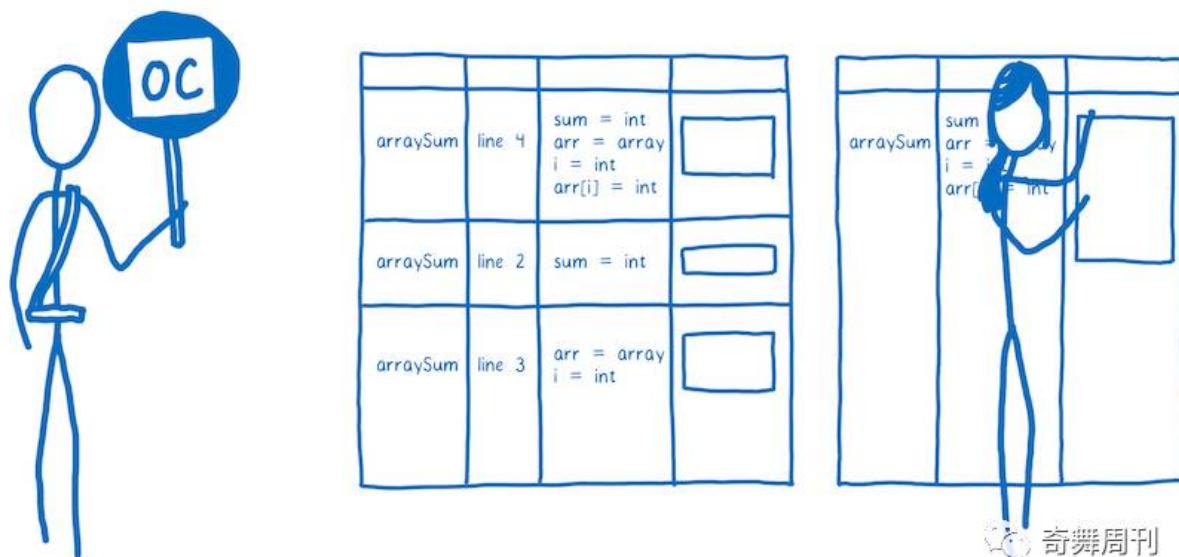
因为解释器必须在每次循环访问时不断重新转换代码，作为一个可以摆脱解释器低效率的方法，浏览器开始将编译器引入。

不同的浏览器实现起来稍有不同，但是基本目的是相同的。他们给 JavaScript 引擎添加了一个新的部分，称为监视器（也称为分析器）。该监视器在 JavaScript 运行时监控代码，并记录代码片段运行的次数以及使用了那些数据类型。

如果相同的代码行运行了几次，这段代码被标记为“warm”。如果运行次数比较多，就被标记为“hot”。

被标记为“warm”的代码被扔给基础编译器，只能提升一点点的速度。被标记为“hot”的代码被扔给优化编译器，速度提升的更多。

```
function arraySum(arr) {  
    var sum = 0;  
    for (var i = 0; i < arr.length; i++) {  
        sum += arr[i];  
    }  
}
```

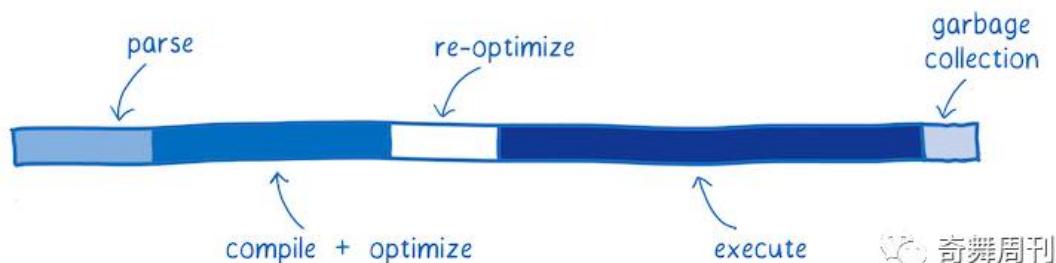


了解更多，可以读 <https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/>

3. 耗时比较：JavaScript Vs. WebAssembly

这张图大致给出了现在一个程序的启动性能，目前 JIT 编译器在浏览器中很常见。

该图显示了 JS 引擎运行程序花费的时间。显示的时间并不是平均的。这个图片表明，JS 引擎做的这些任务花费的时间取决于页面中 JavaScript 做了什么事情。但是我们可以用这个图来构建一个心理模型。



每栏显示花费在特定任务上的时间。

Parsing - 讲源码转换成解释器可以运行的东西所用的事情。

Compiling + optimizing - 花费在基础编译和优化编译上的时间。有一些优化编译的工作不在主线程，所以这里并不包括这些时间。

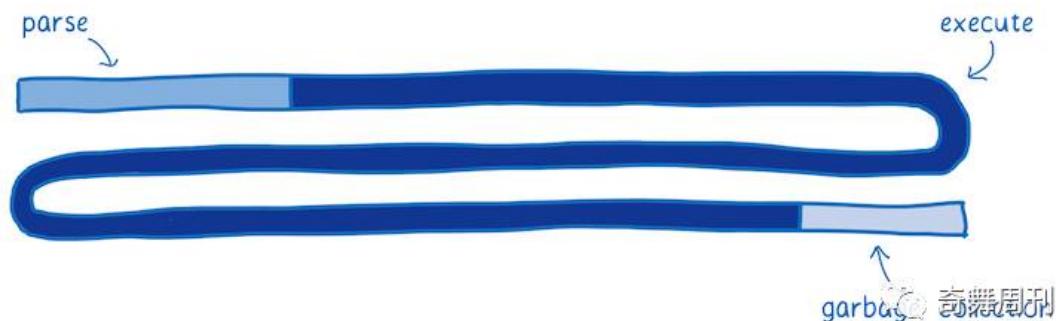
Re-optimizing - 当预先编译优化的代码不能被优化的情况下，JIT 将这些代码重新优化，如果不能重新优化那么久丢给基础编译去做。这个过程叫做重新优化。

Execution - 执行代码的过程

Garbage collection - 清理内存的时间

一个重要的事情要注意：这些任务不会发生在离散块或特定的序列中。相反，它们将被交叉执行。比如正在做一些代码解析时，还执行者一些其他的逻辑，有些代码编译完成后，引擎又做了一些解析，然后又执行了一些逻辑，等等。

这种交叉执行对早期 JavaScript 的性能有很大的帮助，早期的 JavaScript 的执行就像下图一样：

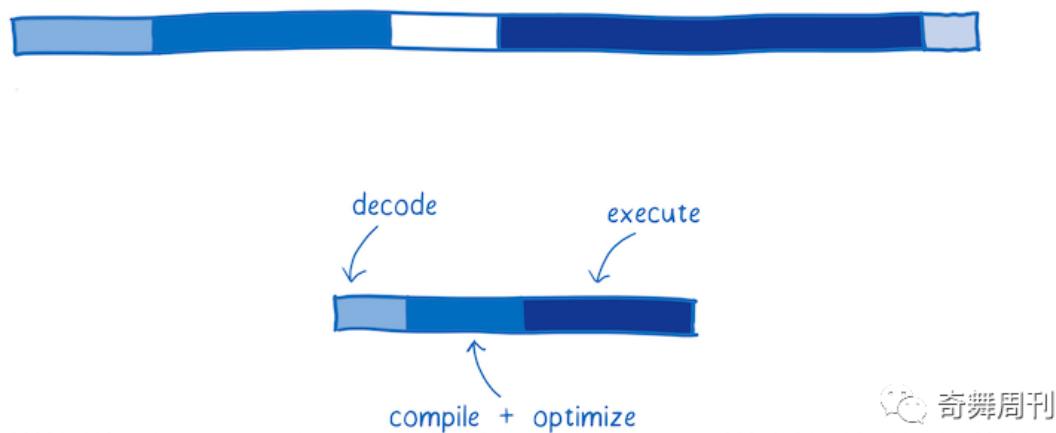


一开始，当只有一个解释器运行 JavaScript 时，执行速度相当缓慢。JITs 的引入，大大提升了执行效率。

监视和编译代码的开销是需要权衡的事情。如果 JavaScript 开发人员按照相同的方式编写 JavaScript，解析和编译时间将会很小。但是，性能的提升使开发人员能够创建更大的JavaScript应用程序。

这意味着还有改进的余地。

下面是 WebAssembly 如何比较典型 web 应用。



浏览器的 JS 引擎有轻微的不同。我是基于 SpiderMonkey 来讲。

3.1 请求

这没有展示在图上，但是从服务器获取文件是会消耗时间的

下载执行与 JavaScript 等效的 WebAssembly 文件需要更少的时间，因为它的体积更小。
WebAssembly 设计的体积更小，可以以二进制形式表示。

即使使用 gzip 压缩的 JavaScript 文件很小，但 WebAssembly 中的等效代码可能更小。

所以说，下载资源的时间会更少。在网速慢的情况下更能显示出效果来。

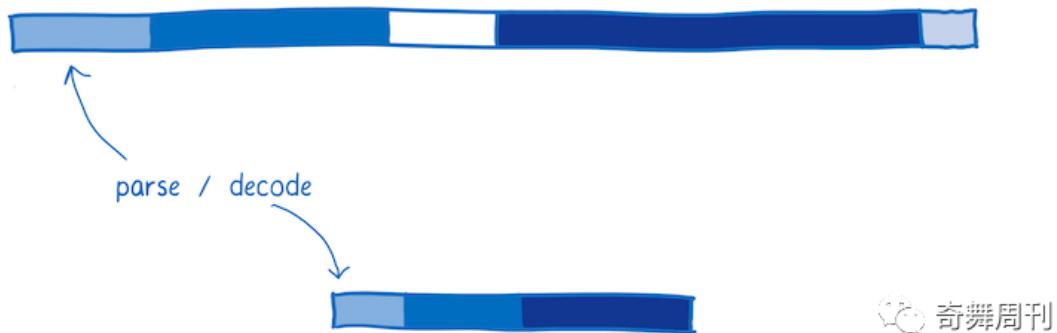
3.2 解析

JavaScript 源码一旦被下载到浏览器，源将被解析为抽象语法树（AST）。

通常浏览器解析源码是懒惰的，浏览器首先会解析他们真正需要的东西，没有及时被调用的函数只会被创建成存根。

在这个过程中，AST 被转换为该 JS 引擎的中间表示（称为字节码）。

相反，WebAssembly 不需要被转换，因为它已经是字节码了。它仅仅需要被解码并确定没有任何错误。



奇舞周刊

3.3 编译 + 优化

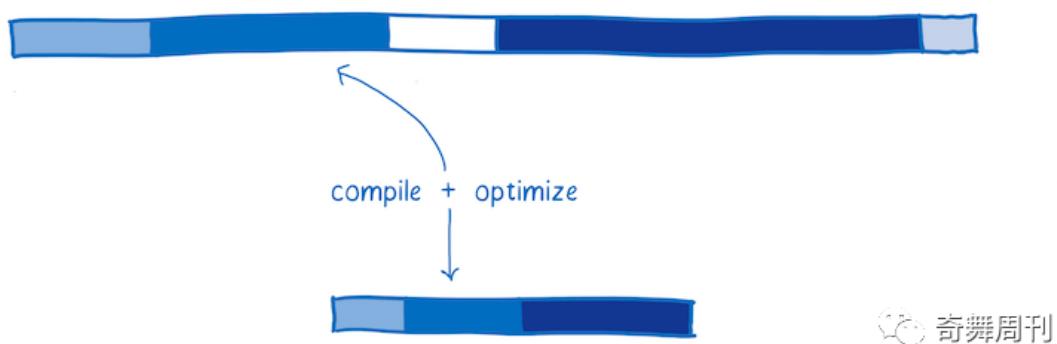
如前所述，JavaScript 是在执行代码期间编译的。因为 JavaScript 是动态类型语言，相同的代码在多次执行中都有可能都因为代码里含有不同的类型数据被重新编译。这样会消耗时间。

相反，WebAssembly 与机器代码更接近。例如，类型是程序的一部分。这是速度更快的一个原因：

编译器不需要在运行代码时花费时间去观察代码中的数据类型，在开始编译时做优化。

编译器不需要去每次执行相同代码中数据类型是否一样。

更多的优化在 LLVM 最前面就已经完成了。所以编译和优化的工作很少。



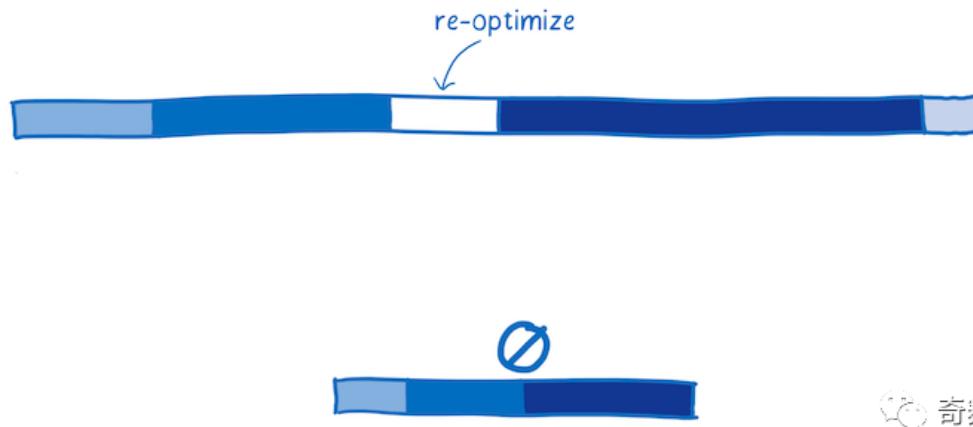
奇舞周刊

3.4 重新优化

有时 JIT 抛出一个优化版本的代码，然后重新优化。

JIT 基于运行代码的假设不正确时，会发生这种情况。例如，当进入循环的变量与先前的迭代不同时，或者在原型链中插入新函数时，会发生重新优化。

在 WebAssembly 中，类型是明确的，因此 JIT 不需要根据运行时收集的数据对类型进行假设。这意味着它不必经过重新优化的周期。



奇舞周刊

3.5 执行

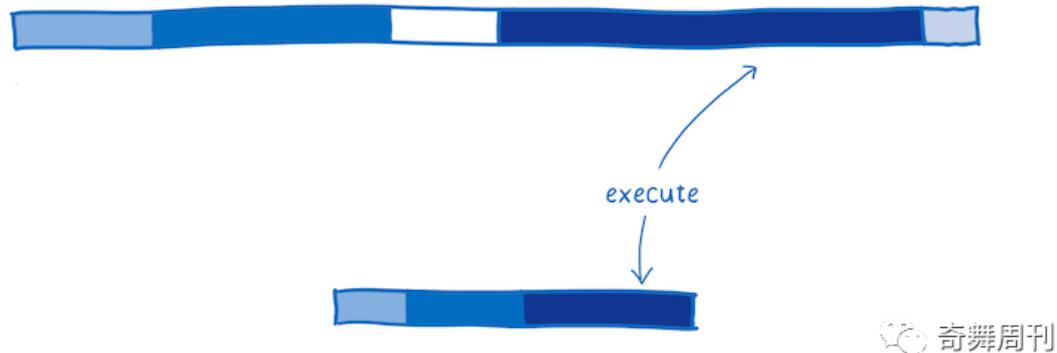
尽可能编写执行性能好的 JavaScript。所以，你可能需要知道 JIT 是如何做优化的。

然而，大多数开发者并不知道 JIT 的内部原理。即使是那些了解 JIT 内部原理的开发人员，也很难实现最佳的方案。有很多时候，人们为了使他们的代码更易于阅读（例如：将常见任务抽象为跨类型工作的函数）会阻碍编译器优化代码。

正因如此，执行 WebAssembly 代码通常更快。有些必须对 JavaScript 做的优化不需要用在 WebAssembly 上

另外，WebAssembly 是为编译器设计的。意思是，它是专门给编译器来阅读，并不是当做编程语言让程序员去写的。

由于程序员不需要直接编程，WebAssembly 提供了一组更适合机器的指令。根据您的代码所做的工作，这些指令的运行速度可以在10%到800%之间。



奇舞周刊

3.6 垃圾回收

在 JavaScript 中，开发者不需要担心内存中无用变量的回收。JS 引擎使用一个叫垃圾回收器的东西来自动进行垃圾回收处理。

这对于控制性能可能并不是一件好事。你并不能控制垃圾回收时机，所以它可能在非常重要的时间去工作，从而影响性能。

现在，WebAssembly 根本不支持垃圾回收。内存是手动管理的（就像 C/C++）。虽然这些可能让开发者编程更困难，但它的确提升了性能。



奇舞周刊

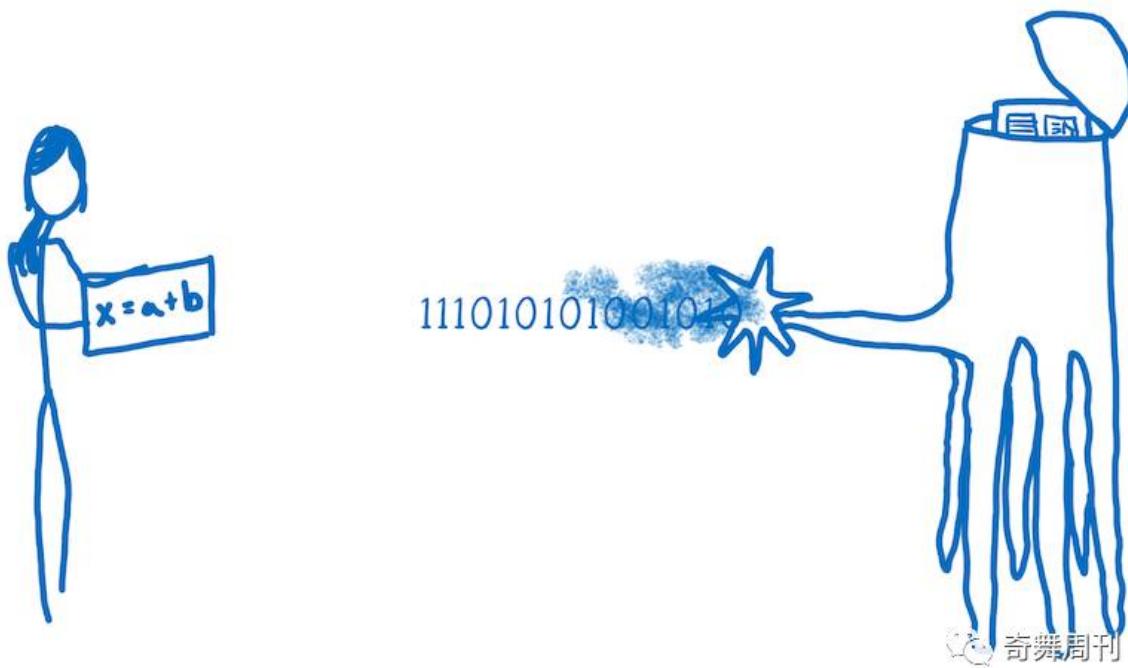
总而言之，这些都是在许多情况下，在执行相同任务时WebAssembly 将胜过 JavaScript 的原因。

在某些情况下，WebAssembly 不能像预期的那样执行，还有一些更改使其更快。我在另一篇文章中更深入地介绍了这些未来功能。

4. WebAssembly 是如何工作的？

现在，您了解开发人员为什么对 WebAssembly 感到兴奋，让我们来看看它是如何工作的。

当我谈到上面的 JIT 时，我谈到了与机器的沟通像与外星人沟通。



我现在想看看这个外星人的大脑如何工作 - 机器的大脑如何解析和理解交流内容。

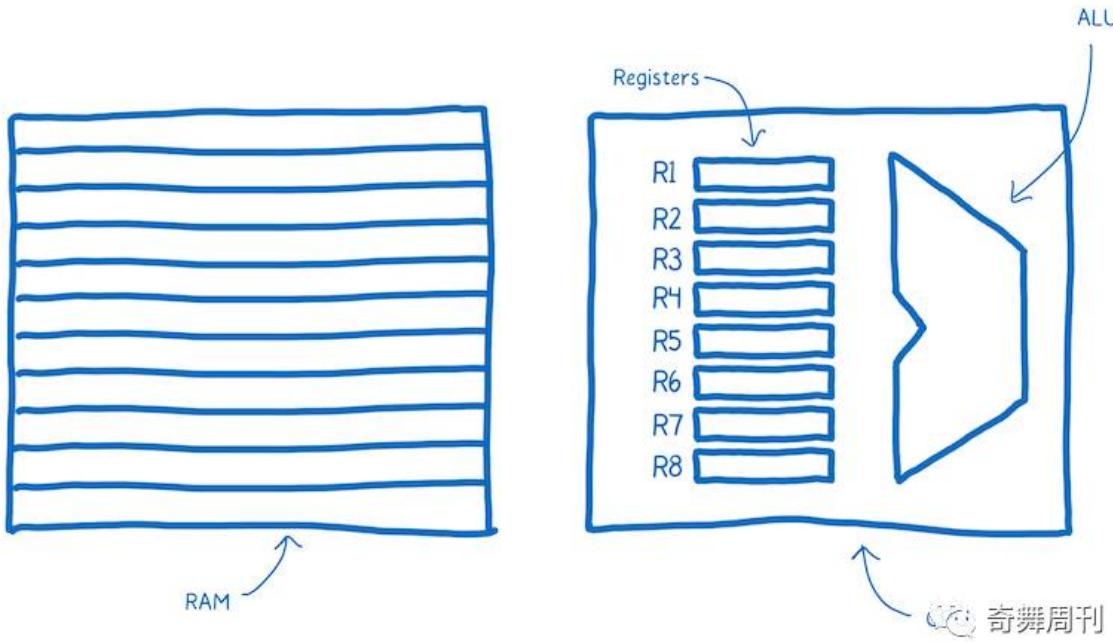
这个大脑的一部分是专注于思考，例如算术和逻辑。有一部分脑部提供短期记忆，另一部分提供长期记忆。

这些不同的部分都有名字。

负责思考的部分是算术逻辑单元 (ALU) 。

短期储存由寄存器 (Registers) 提供。

随机存储器 (或RAM) 来提供长期储存能力。



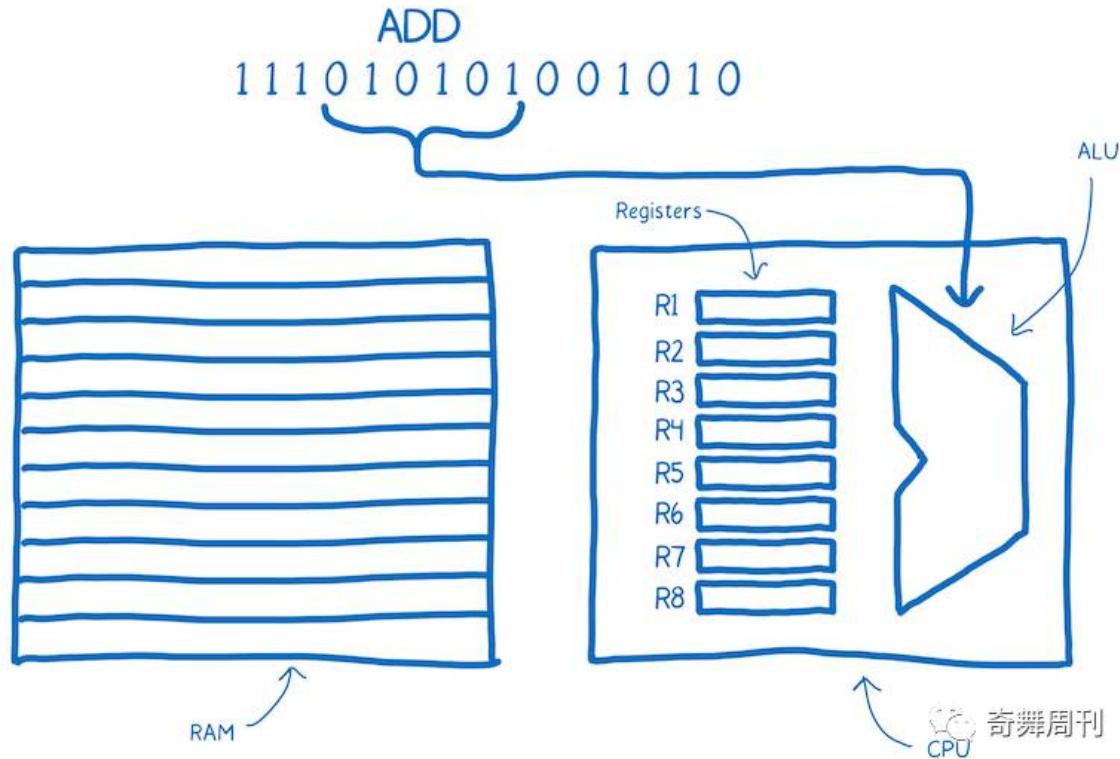
机器码中的语句被称为指令。

当一条指令进入大脑时会发生什么？它被拆分成了多个的部分并有特殊的含义。

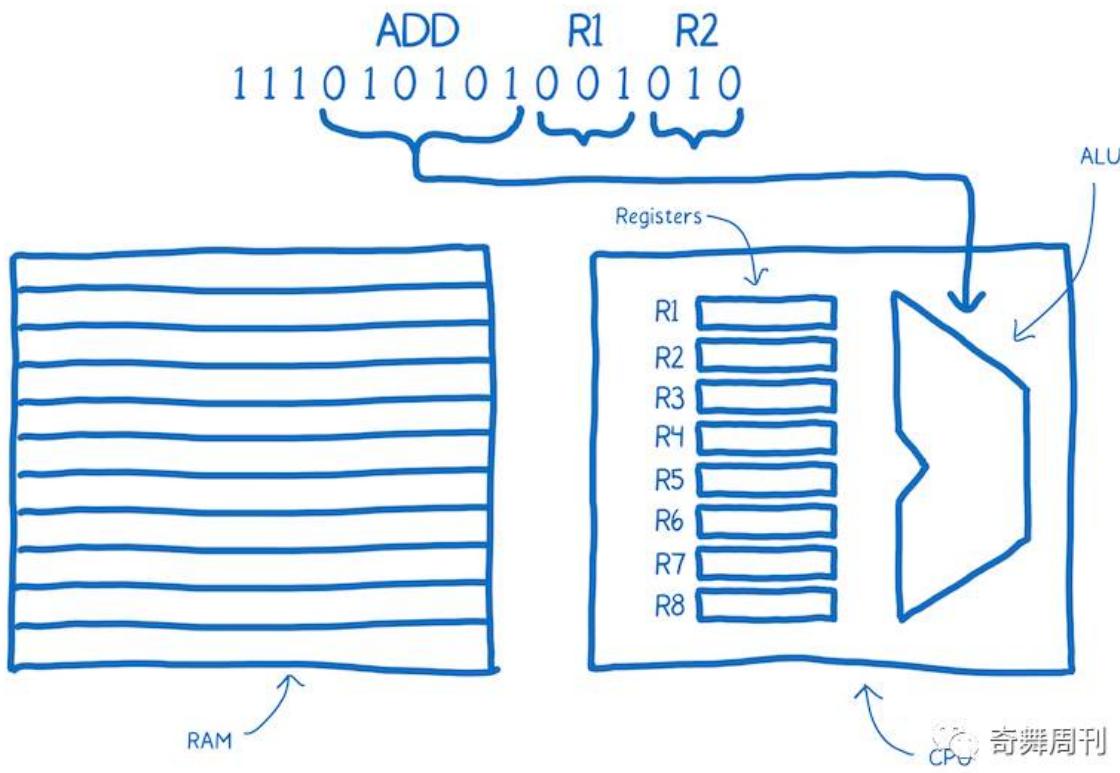
被拆分成的多个部分分别进入不同的大脑单元进行处理，这也是拆分指令所依赖的方式。

例如，这个大脑从机器码中取出4-10位，并将它们发送到 ALU。ALU进行计算，它根据 0 和 1 的位置来确定是否需要将两个数相加。

这个块被称为“操作码”，因为它告诉 ALU 执行什么操作。



那么这个大脑会拿后面的两个块来确定他们所要操作的数。这两个块对应的是寄存器的地址。



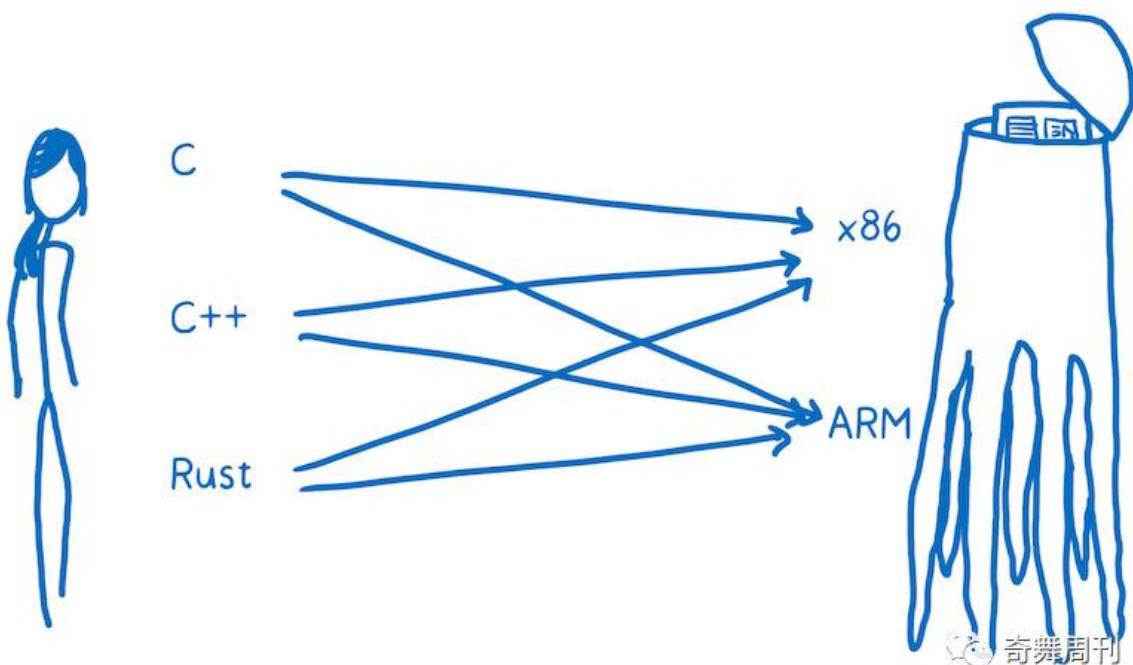
请注意添加在机器码上面的标注 (ADD R1 R2)，这使我们更容易了解发生了什么。这就是汇编。它被称为符号机器码。这样人类也能看懂机器码的含义。

您可以看到，这个机器的汇编和机器码之间有非常直接的关系。每种机器内部有不同的结构，所以每种机器都有自己独有的汇编语言。

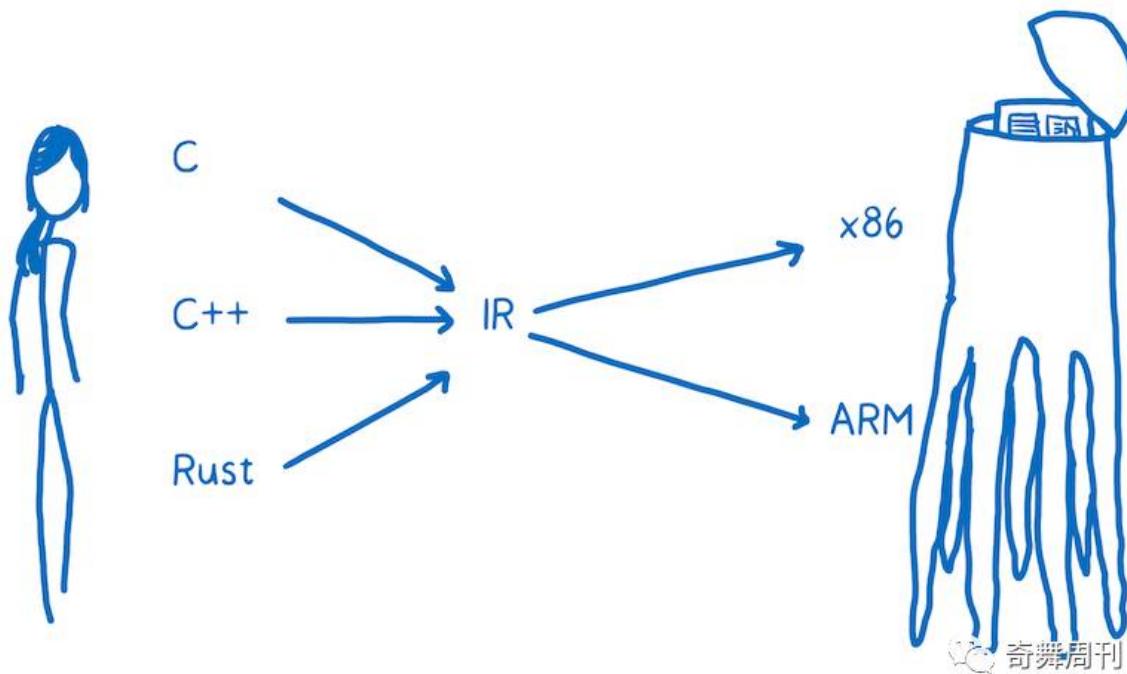
所以我们并不只有一个翻译的目标。

相反，我们的目标是不同类型的机器码。就像人类说不同的语言一样，机器也有不同的语言。

您希望能够将这些任何一种高级编程语言转换为任何一种汇编语言。这样做的一个方法是创建一大堆不同的翻译器，可以从任意一种语言转换成任意一种汇编语言。

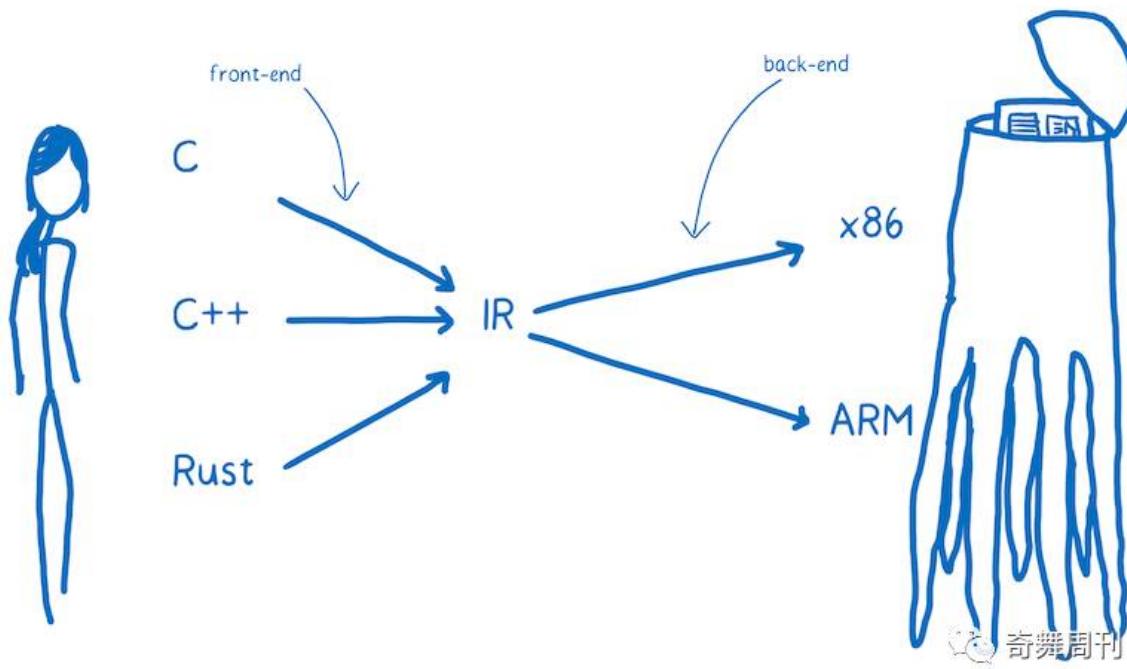


这样做的效率非常低。为了解决这个问题，大多数编译器会在高级语言和汇编语言之间多加一层。编译器将把高级语言翻译成一种更低级的语言，但比机器码的等级高。这就是中间代码（IR）。



意思就是编译器可以将任何一种高级语言转换成一种中间语言。然后，编译器的另外的部分将中间语言编译成目标机器的汇编代码。

编译器的“前端”将高级编程语言转换为IR。编译器的“后端”将 IR 转换成目标机器的汇编代码。



4.1 WebAssembly 适合在哪里使用？

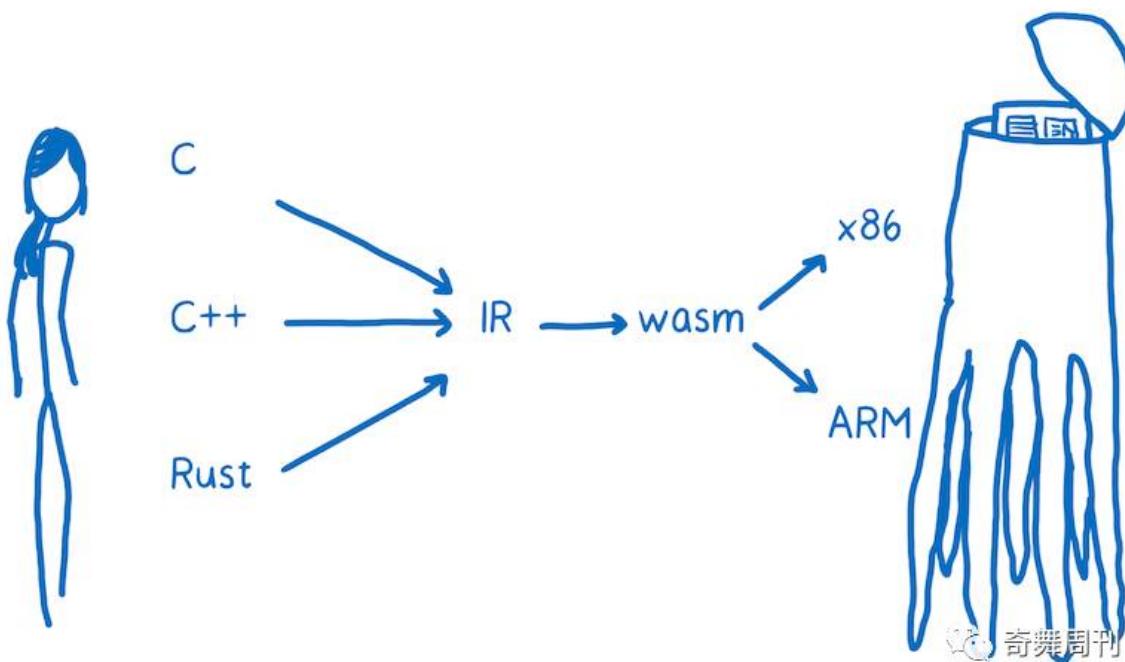
您可能会将 WebAssembly 当做是另外一种目标汇编语言。这是真的，这些机器语言（x86, ARM等）中的每一种都对应于特定的机器架构。

当你的代码运行在用户的机器的 web 平台上的时候，你不知道你的代码将会运行在那种机器结构上。

所以 WebAssembly 和别的汇编语言是有一些不同的。所以他是在一个概念机上的机器语言，不是在一个真正存在的物理机上运行的机器语言。

正因如此，WebAssembly 指令有时候被称为虚拟指令。它比 JavaScript 代码更快更直接的转换成机器代码，但它们不直接和特定硬件的特定机器代码对应。

在浏览器下载 WebAssembly 后，使 WebAssembly 的迅速转换成目标机器的汇编代码。



如果想在您的页面里上添加 WebAssembly，您需要将您的代码编译成 .wasm 文件。

5. 编译到 .wasm 文件

当前对 WebAssembly 支持最多的编译器工具链称是 LLVM。有许多不同的“前端”和“后端”可以插入到 LLVM 中。

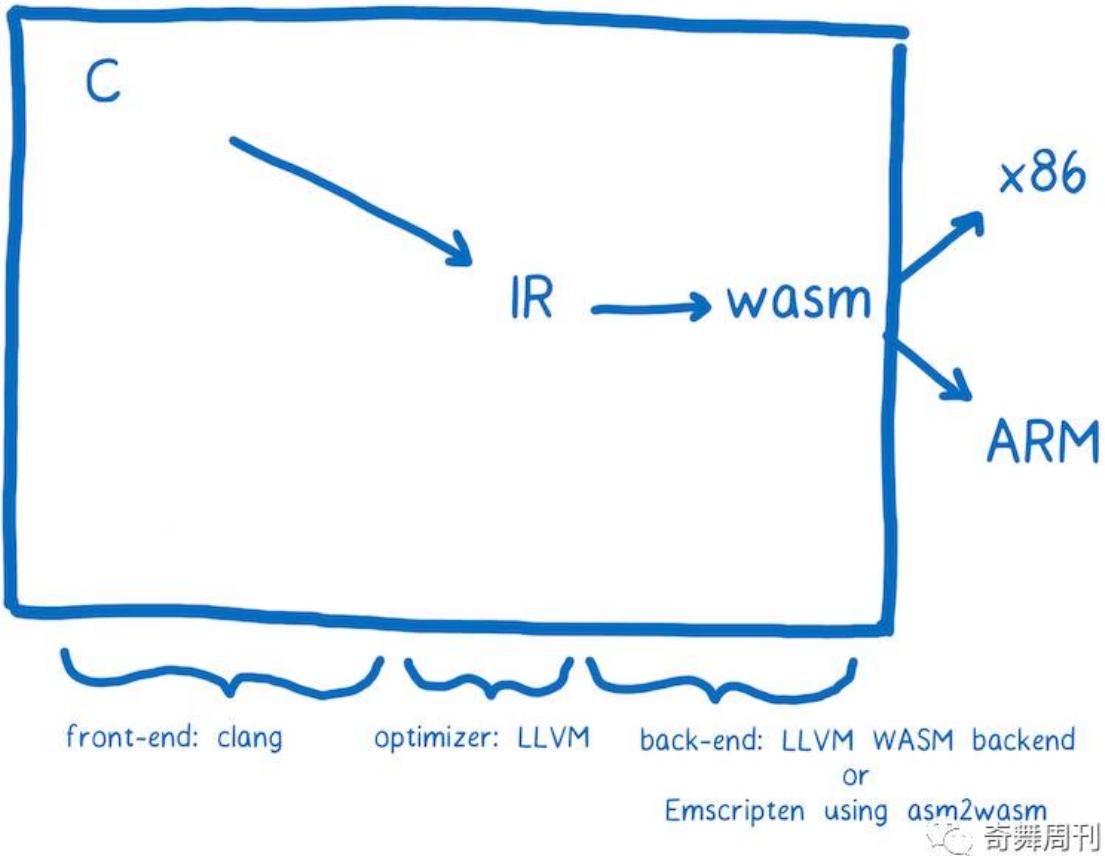
注意：大多数 WebAssembly 模块开发者使用 C 和 Rust 编写代码，然后编译成 WebAssembly，但是这里有其他创建 WebAssembly 模块的途径。比如，这里有一个实验性工具，他可以帮你使用 Type 创建一个 WebAssembly 模块，你可以在这里直接编辑 WebAssembly。

假设我们想通过 C 来创建 WebAssembly。我们可以使用 clang “前端”从 C 编译成 LLVM 中间代码。当它变成 LLVM 的中间代码 (IR) 以后，LLVM 可以理解他，所以 LLVM 可以对代码做一些优化。

如果想让 LLVM 的 IR 变成 WebAssembly，我们需要一个“后端”。目前 LLVM 项目中有一个正在开发中的。这个“后端”对做这件事情很重要，应该很快就会完成。可惜，它现在还不能用。

另外有一个工具叫做 Emen，它用起来比较简单。它还可以有比较有用的可以选择，比如说由 IndexDB 支持的文件系统。

Compiler toolchain



不管你使用的什么工具链，最终的结果都应该是以 .wasm 结尾的文件。来让我们看一下如何将它用在你的 web 页面。

6. 在 JavaScript 中加载一个 .wasm 组件

.wasm 文件是 WebAssembly 组件，它可以被 JavaScript 加载。到目前为止，加载过程有点复杂。

```
function fetchAndInstantiate(url, importObject){  
    return fetch(url).then(response =>  
        response.arrayBuffer()  
    ).then(bytes =>  
        WebAssembly.instantiate(bytes, importObject)  
    ).then(results =>  
        results.instance  
    );  
}
```

您可以在文档中更深入地了解这些。

我们正在努力使这个过程更容易。我们期望对工具链进行改进，并与现有的模块管理工具（如 Webpack）或加载器（如 SystemJS）相结合。我相信，加载 WebAssembly 模块越来越简单，就像加载 JavaScript 一样。

但是，WebAssembly模块和JS模块之间存在重大差异。目前，WebAssembly 中的函数只能使用 WebAssembly 类型（整数或浮点数）作为参数或返回值。

```
function js_func() {           → 1   int c_func(int x) {  
    let result = c_func(1)      ← 2   }   return x+1;  
}  
奇舞周刊
```

对于任何更复杂的数据类型（如字符串），必须使用 WebAssembly 模块的内存。

如果你之前主要使用 JavaScript，可能对于直接访问内存是不熟悉的。C, C++ 和 Rust 等性能更高的语言往往具有手动内存管理功能。WebAssembly 模块的内存模拟这些语言中的堆。

为此，它使用 JavaScript 中称为 ArrayBuffer。ArrayBuffer 是一个字节数组。因此，数组的索引作为内存地址。

如果要在 JavaScript 和 WebAssembly 之间传递一个字符串，需要将字符转换为等效的字符码。然后你需要将它写入内存数组。由于索引是整数，所以可以将索引传递给 WebAssembly 函数。因此，字符串的第一个字符的索引可以当作指针。

```
function js_func() {           → 0   void c_func(char** ptr) {  
    // create pointer  
    c_func(ptr)  
    // get value from pointer  
}  
                                *ptr = "heap data";  
                                }  
  
[  
 01101000,  
 01100101,  
 01100001,  
 01110000,  
 00100000,  
 01100100,  
 01100001,  
 01110100,  
 01100001,  
 00000000  
]  
奇舞周刊
```

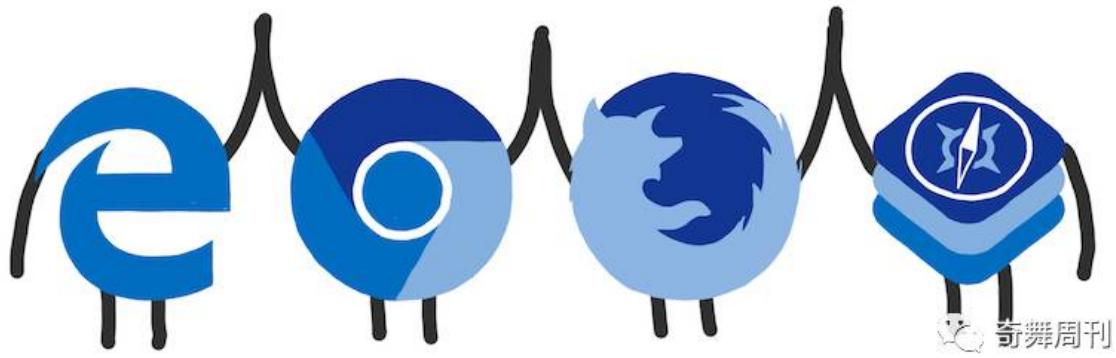
任何人开发的 WebAssembly 模块很可能被 Web 开发人员使用并为该模块创建一个的装饰器。这样，您当做用户来使用这个模块就不需要考虑内存管理的事情了。

我已经在另一篇文章中解释了更多关于使用 WebAssembly 模块的内容。

7. WebAssembly 现在是什么状态？

二月二十八日，四大浏览器宣布达成共识，即 WebAssembly 的 MVP（最小化可行产品）已经完成。大约一周后，Firefox 会默认打开 WebAssembly 支持，而 Chrome 则在第二周开始。它也可用于预览版本的 Edge 和 Safari。

这提供了一个稳定的初始版本，浏览器开始支持。



该核心不包含社区组织计划的所有功能。即使在初始版本中，WebAssembly 也会很快。但是，通过修复和新功能的组合，将来应该能够更快。我在另一篇文章中详细介绍了这些功能。

8. 总结

使用WebAssembly，可以更快地在 web 应用上运行代码。这里有 几个 WebAssembly 代码运行速度比 JavaScript 高效的原因。

文件加载 - WebAssembly 文件体积更小，所以下载速度更快。

解析 - 解码 WebAssembly 比解析 JavaScript 要快

编译和优化 - 编译和优化所需的时间较少，因为在将文件推送到服务器之前已经进行了更多优化，JavaScript 需要为动态类型多次编译代码

重新优化 - WebAssembly 代码不需要重新优化，因为编译器有足够的信息可以在第一次运行时获得正确的代码

执行 - 执行可以更快，WebAssembly 指令更接近机器码

垃圾回收 - 目前 WebAssembly 不直接支持垃圾回收，垃圾回收都是手动控制的，所以比自动垃圾回收效率更高。

目前浏览器中的 MVP（最小化可行产品）已经很快了。在接下来的几年里，随着浏览器的发展和新功能的增加，它将在未来几年内变得更快。没有人可以肯定地说，这些性能改进可以实现什么样的应用。但是，如果过去有任何迹象，我们可以期待惊奇。

本文链接：<https://www.jianshu.com/p/bff8aa23fe4d>

这是英文原文 <https://www.smashingmagazine.com/2017/05/abridged-cartoon-introduction-to-webassembly/>

这是中文原文 <https://www.zcfy.cc/article/an-abridged-cartoon-introduction-to-webassembly-nash-smashing-magazine>

本文转自 <https://www.jianshu.com/p/bff8aa23fe4d>，如有侵权，请联系删除。 在上世纪90年代，几乎所有的网站都由HTML页面实现，服务器处理每一个用户请求都需要重新加载网页。用户体验极差！由于每次应用的沟通都需要向服务器发送请求，应用的回应时间依赖于服务器的回应时间。这导致了用户界面的回应比本机应用慢得多。

在 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简介

Ajax 并不算是一种新的技术，全称是 asynchronous javascript and xml，可以说是已有技术的组合，主要用来实现客户端与服务器端的异步通信效果，实现页面的局部刷新，早期的浏览器并不能原生支持 ajax，可以使用隐藏帧（iframe）方式变相实现异步效果，后来的浏览器提供了对 ajax 的原生支持使用 ajax 原生方式发送请求主要通过 XMLHttpRequest（标准浏览器）、ActiveXObject（IE 浏览器）对象实现异步通信效果

ajax是基于现有的Internet标准，并且联合使用它们：

- XMLHttpRequest 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优缺点

【1】优点

- 页面局部刷新，用户体验好。
- 异步通信，更加快的响应能力。
- 减少冗余请求，减轻了服务器负担；按需获取数据，节约带宽资源。
- 基于标准化的并被广泛支持的技术，不需要下载插件或者小程序

【2】缺点

- ajax 干掉了 back 按钮和加入收藏书签功能，即对浏览器后退机制的破坏。
- 存在一定的安全问题，AJAX 暴露了与服务器交互的细节。
- 对搜索引擎的支持比较弱。
- 破坏了程序的异常机制。
- 无法用 URL 直接访问

创建Ajax的步骤

Ajax 的原理简单来说通过 XMLHttpRequest 对象来向服务器发异步请求，从服务器获得数据，然后用 javascript 来操作 DOM 而更新页面。这其中最关键的一步就是从服务器获得请求数据

ajax过程：

1. 创建 XMLHttpRequest 对象，也就是创建一个异步调用对象
2. 创建一个新的 HTTP 请求，并指定该 HTTP 请求的方法、URL 及验证信息
3. 设置响应 HTTP 请求状态变化的函数
4. 发送 HTTP 请求
5. 获取异步调用返回的数据
6. 使用 JavaScript 和 DOM 实现局部刷新

【1】创建XMLHttpRequest

XMLHttpRequest 是 AJAX 的基础。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。所有现代浏览器均支持 XMLHttpRequest 对象（IE5 和 IE6 使用 ActiveXObject）。

```

var xhr;
if (window.XMLHttpRequest){
    // IE7+, Firefox, Chrome, Opera, Safari 浏览器执行代码
    xhr = new XMLHttpRequest();
} else {
    // IE6, IE5 浏览器执行代码
    xhr = new ActiveXObject("Microsoft.XMLHTTP");
}

```

【2】向服务器发送请求

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

方法	描述
open(<i>method,url,async</i>)	规定请求的类型、URL 以及是否异步处理请求。 1、 <i>method</i> : 请求的类型; GET 或 POST 2、 <i>url</i> : 文件在服务器上的位置 3、 <i>async</i> : true (异步) 或 false (同步)
send(<i>string</i>)	将请求发送到服务器。 1、 <i>string</i> : 仅用于 POST 请求

```

// 一个简单的get请求
xhr.open("GET","ajax_info.txt",true);
xhr.send();

// 一个简单的post请求
xhr.open("POST","/try/ajax/demo_post.php",true);
xhr.send();

```

【3】处理服务器响应

当请求被发送到服务器时，我们需要执行一些基于响应的任务。每当 readyState 改变时，就会触发 onreadystatechange 事件。readyState 属性存有 XMLHttpRequest 的状态信息。

方法	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	<p>存有 XMLHttpRequest 的状态。从 0 到 4 发生变化。</p> <ul style="list-style-type: none"> • 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	<p>200: "OK"</p> <p>404: 未找到页面</p>

```

xhr.onreadystatechange = function(){
    if(xhr.readyState == 4 && xhr.status == 200){
        console.log(xhr.responseText); // responseText获取字符串形式的响应数据
    }
}

```

Ajax实例

【1】原生Ajax请求

```

<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
</head>

<body>
    <div id="myDiv">
        <h2>使用 AJAX 修改该文本内容</h2>
    </div>
    <button type="button" onclick="loadXMLDoc()">修改内容</button>

    <script>
        function loadXMLDoc() {
            var xhr;
            if (window.XMLHttpRequest) {
                // IE7+, Firefox, Chrome, Opera, Safari 浏览器执行代码
                xhr = new XMLHttpRequest();
            }
            else {
                // IE6, IE5 浏览器执行代码
                xhr = new ActiveXObject("Microsoft.XMLHTTP");
            }
            xhr.onreadystatechange = function () {
                if (xhr.readyState == 4 && xhr.status == 200) {

```

```

        document.getElementById("myDiv").innerHTML = xhr.responseText;
    }
}
xhr.open("GET", "/try/ajax/ajax_info.txt", true);
xhr.send();
}
</script>

</body>

</html>

```

【2】jQuery封装Ajax请求

```

<!DOCTYPE html>
<html>

<head>
<meta charset="utf-8">
<title></title>
<script src="https://cdn.staticfile.org/jquery/1.10.2/jquery.min.js">
</script>
</head>

<body>

<div id="myDiv">
    <h2>使用 jQuery AJAX 修改文本内容</h2>
</div>
<button>修改内容</button>

<script>
$(document).ready(function () {
    $("button").click(function () {
        htmlObj = $.ajax({
            type: "GET",
            url: "/jquery/test1.txt",
            data: {},
            dataType: "json",
            async: false,
            success: function (data) {
                $("#myDiv").html(data);
            }
        });
    });
});
</script>

</body>

</html>

```



微信搜一搜



前端大集锦

前端下载文件的几种方式

前言

实习一个人负责一个管理系统的前端部分。其中，就有前端下载文件的需要。最终采用的是使用 `axios` 发送 `get` 请求的方式，因为需要携带 `token`。但是，不应该只注重结果，也应该注重过程，不然可能一直都是拧螺丝。另外提一嘴，找工作最好还是找能去的最大的公司，虽然小公司也能学到东西，但是因为制度不完善的缘故，可能会被不好的小公司坑，问就是一行泪。(去的小公司开始对实习生下手了，甚至有请假回去答辩时被背刺的)

a链接

极简版本

实际上，如果a链接的 `href` 就是指向文件的地址的话，是可以直接下载的。这种方式下载的文件名就是原本的文件名。

```
<a href="https://www.czczh.top/medias/test.xlsx">下载文件</a>
```

[下载文件](#)

test.xlsx

自定义文件名

通过 `download` 属性，可以实现对下载的文件进行重命名。

```
<a href="https://www.czczh.top/medias/test.xlsx" download="cz.xlsx">下载文件</a>
```

上面不能实现重命名？

这是因为通过 `download` 属性来实现对下载的文件进行重命名需要是同源路径下。

换成同源路径后，就能实现自定义文件名了

```
<a href="/test.xlsx" download="clz.xlsx">下载文件</a>
```



动态添加a标签

在上面的例子中，我们是通过点击a标签来实现下载文件的。但是，实际上，我们也可以通过动态添加a标签的形式来实现。

```
<button onclick="download()">下载文件</button>
•
<script>
  function download() {
    const a = document.createElement('a')
    a.href = '/test.xlsx'
    a.download = 'clz.xlsx'
    a.click()
  }
</script>
```

结果和上面一样

指定location的href

通过指定 `location` 对象的 `href` 属性，就可以在当前页面打开URL页面，其实就和上面a链接的极简版本一样效果。

```
<button onclick="download()">下载文件</button>
•
<script>
  function download() {
    window.location.href = '/test.xlsx'
  }
</script>
```

无法重命名

window.open

打开指定的页面的URL

```
<button onclick="download()">下载文件</button>
•
<script>
  function download() {
    window.open('/test.xlsx')
  }
</script>
```

无法重命名

axios

因为下载文件其实就相当于发起get请求，所以可以使用axios来发起请求，这样子就能够很方便地携带token等信息。

因为是下载文件，所以响应类型应该是 `blob`，用来存取二进制数据

```
<button onclick="download()">下载文件</button>
•
<script>
  function download() {
    axios({
      url: "/test.xlsx",
      method: 'GET',
      responseType: 'blob', // 用来存取二进制数据
      headers: {
        token: 'testtest' // 可以携带token
      }
    }).then(res => {
      console.log(res)
    })
  }
</script>
```

Name	Headers	Preview	Response	Initiator	Timing
General					
Request URL: http://127.0.0.1:5500/test.xlsx					
Request Method: GET					
Status Code: 204 OK					
Remote Address: 127.0.0.1:5500					
Referrer Policy: strict-origin-when-cross-origin					
Response Headers					
Accept-Ranges: bytes					
Access-Control-Allow-Credentials: true					
Cache-Control: public, max-age=0					
Connection: keep-alive					
Date: Sat, 30 Apr 2022 13:26:38 GMT					
ETag: W/"2517-18079f80d5d"					
Keep-Alive: timeout=5					
Last-Modified: Sat, 30 Apr 2022 10:15:41 GMT					
Vary: Origin					
Request Headers					
Accept: application/json, text/plain, */*					
Accept-Encoding: gzip, deflate, br					
Accept-Language: zh-CN,zh;q=0.9					
Connection: keep-alive					
Host: 127.0.0.1:5500					
If-Modified-Since: Sat, 30 Apr 2022 10:15:41 GMT					
If-None-Match: W/"2517-18079f80d5d"					
Referer: http://127.0.0.1:5500/icon.html					
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="100", "Google Chrome";v="100"					
sec-ch-ua-mobile: ?0					
sec-ch-ua-platform: "Windows"					
Sec-Fetch-Dest: empty					
Sec-Fetch-Mode: cors					
Sec-Fetch-Site: same-origin					
token: testtest					
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/100.0.4896.127 Safari/537.36					

不过，此时并没有下载文件，因为此时是二进制数据。

Name	Headers	Preview	Response	Initiator	Timing
General					
Request URL: http://127.0.0.1:5500/test.xlsx					
Request Method: GET					
Status Code: 204 OK					
Remote Address: 127.0.0.1:5500					
Referrer Policy: strict-origin-when-cross-origin					
Response Headers					
Accept-Ranges: bytes					
Access-Control-Allow-Credentials: true					
Cache-Control: public, max-age=0					
Connection: keep-alive					
Date: Sat, 30 Apr 2022 13:26:38 GMT					
ETag: W/"2517-18079f80d5d"					
Keep-Alive: timeout=5					
Last-Modified: Sat, 30 Apr 2022 10:15:41 GMT					
Vary: Origin					
Request Headers					
Accept: application/json, text/plain, */*					
Accept-Encoding: gzip, deflate, br					
Accept-Language: zh-CN,zh;q=0.9					
Connection: keep-alive					
Host: 127.0.0.1:5500					
If-Modified-Since: Sat, 30 Apr 2022 10:15:41 GMT					
If-None-Match: W/"2517-18079f80d5d"					
Referer: http://127.0.0.1:5500/icon.html					
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="100", "Google Chrome";v="100"					
sec-ch-ua-mobile: ?0					
sec-ch-ua-platform: "Windows"					
Sec-Fetch-Dest: empty					
Sec-Fetch-Mode: cors					
Sec-Fetch-Site: same-origin					
token: testtest					
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/100.0.4896.127 Safari/537.36					

所以，我们还需要将二进制数据变成文件下载。

那么，怎么将二进制数据变成文件呢？

这里在网上找到一个方法，就是通过 `URL.createObjectURL` 方法，生成对应二进制数据 `blob` 对象的 URL，然后通过动态添加a标签的方法，来实现生成文件。

```
<button onclick="download()">下载文件</button>

<script>
```

```
function download() {
  axios({
    url: "/test.xlsx",
    method: 'GET',
    responseType: 'blob',
    headers: {
      token: 'testtest' // 可以携带token
    }
  }).then(res => {
  }

  const href = URL.createObjectURL(res.data)
  // console.log(href)

  const a = document.createElement('a')
  a.download = 'clz.xlsx'
  a.href = href

  a.click()
}

</script>
```

将二进制数据变成文件也可以通过 `FileReader` 的 `readAsDataURL` 来实现，该方法读取 `blob` 对象或 `file` 对象。读取成功后，能够通过 `onload` 回调函数中通过实例对象的 `target` 属性下的 `result` 属性中获取base64编码的URL。

```
const reader = new FileReader()

reader.readAsDataURL(res.data)

reader.onload = (e) => {
  console.log(e)

  const a = document.createElement('a')
  a.download = 'clz.xlsx'
  a.href = e.target.result

  a.click()
}
```

```
▼ ProgressEvent {isTrusted: true, LengthComputable: true, Loaded: 9495, total: 9495, type: 'Load', ...} ⓘ
  isTrusted: true
  bubbles: false
  cancelBubble: false
  cancelable: false
  composed: false
  ► currentTarget: FileReader {readyState: 2, result: 'data:application/vnd.openxmlformats-officedocument...vY1'
    defaultPrevented: false
    eventPhase: 0
    lengthComputable: true
    loaded: 9495
    path: []
    returnValue: true
  ► srcElement: FileReader {readyState: 2, result: 'data:application/vnd.openxmlformats-officedocument...vY1Byb'}
  ▼ target: FileReader
    error: null
    onabort: null
    onerror: null
    ► onload: (e) => {...}
    onloadend: null
    onloadstart: null
    onprogress: null
    readyState: 2
    result: "data:application/vnd.openxmlformats-officedocumen" Show more (12.7 kB) Copy
  ► [[Prototype]]: FileReader
  timeStamp: 23845
  total: 9495
  type: "load"
  ► [[Prototype]]: ProgressEvent
  ► XHR finished loading: GET "http://127.0.0.1:5500/test.xlsx".
```

[



CSDN 社区图书馆，开张营业! >

深读计划，写书评领图书福利~

层叠上下文

我们假定用户正面向（浏览器）视窗或网页，而 HTML 元素沿着其相对于用户的一条虚构的 z 轴排开，**层叠上下文**就是对这些 HTML 元素的一个三维构想。众 HTML 元素基于其元素属性按照优先级顺序占据这个空间。

层叠上下文

在本篇之前的部分——[运用 z-index](#)，（我们认识到）某些元素的渲染顺序是由其 `z-index` 的值影响的。这是因为这些元素具有能够使他们形成一个**层叠上下文**的特殊属性。

文档中的层叠上下文由满足以下任意一个条件的元素形成：

- 文档根元素 (`<html>`)；
- `position` 值为 `absolute` (绝对定位) 或 `relative` (相对定位) 且 `z-index` 值不为 `auto` 的元素；
- `position` 值为 `fixed` (固定定位) 或 `sticky` (粘滞定位) 的元素 (沾滞定位适配所有移动设备上的浏览器，但老的桌面浏览器不支持)；
- `flex` ([flex](#)) 容器的子元素，且 `z-index` 值不为 `auto`；
- `grid` ([grid](#)) 容器的子元素，且 `z-index` 值不为 `auto`；
- `opacity` 属性值小于 `1` 的元素 (参见 [the specification for opacity](#))；
- `mix-blend-mode` 属性值不为 `normal` 的元素；
- 以下任意属性值不为 `none` 的元素：
 - [transform](#)
 - [filter](#)
 - [backdrop-filter](#)
 - [perspective](#)
 - [clip-path](#)
 - [mask](#) / [mask-image](#) / [mask-border](#)
- `isolation` 属性值为 `isolate` 的元素；
- `will-change` 值设定了任一属性而该属性在 non-initial 值时会创建层叠上下文的元素 (参考[这篇文章](#))；
- `contain` 属性值为 `layout`、`paint` 或包含它们其中之一的合成值 (比如 `contain: strict`、`contain: content`) 的元素。

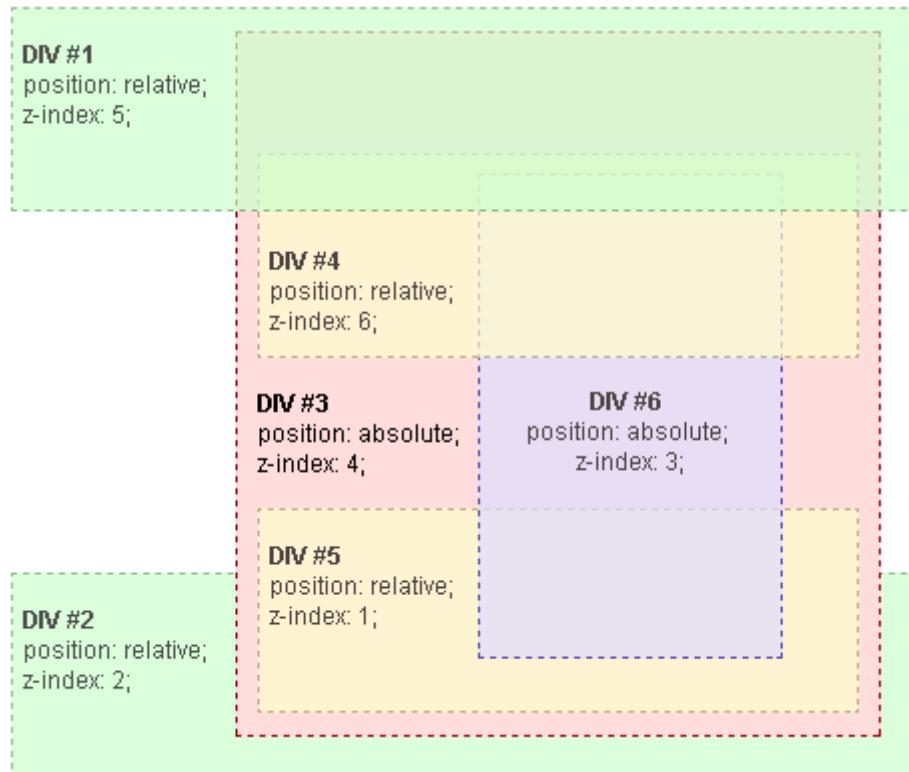
在层叠上下文中，子元素同样也按照上面解释的规则进行层叠。重要的是，其子级层叠上下文的 `z-index` 值只在父级中才有意义。子级层叠上下文被自动视为父级层叠上下文的一个独立单元。

总结：

- 层叠上下文可以包含在其他层叠上下文中，并且一起创建一个层叠上下文的层级。
- 每个层叠上下文都完全独立于它的兄弟元素：当处理层叠时只考虑子元素。
- 每个层叠上下文都是自包含的：当一个元素的内容发生层叠后，该元素将被作为整体在父级层叠上下文中按顺序进行层叠。

Note: 层叠上下文的层级是 HTML 元素层级的一个子级，因为只有某些元素才会创建层叠上下文。可以说这样说，没有创建自己的层叠上下文的元素会被父层叠上下文同化。

示例



在这个例子中，每个被定位的元素都创建了独自的层叠上下文，因为他们被指定了定位属性和 `z-index` 值。我们把层叠上下文的层级列在下面：

- Root
 - DIV #1
 - DIV #2
 - DIV #3
 - DIV #4
 - DIV #5
 - DIV #6

请一定要注意 DIV #4, DIV #5 和 DIV #6 是 DIV #3 的子元素，所以它们的层叠完全在 DIV #3 中被处理。一旦 DIV #3 中的层叠和渲染处理完成，DIV #3 元素就被作为一个整体传递与兄弟元素的 DIV 在 root (根) 元素进行层叠。

注意：

- DIV #4 被渲染在 DIV #1 之下，因为 DIV #1 的 z-index (5) 在 root 元素的层叠上下文中生效，而 DIV #4 的 z-index (6) 在 DIV #3 的层叠上下文中生效。因此，DIV #4 在 DIV #1 之下，因为 DIV #4 归属于 z-index 值较低的 DIV #3 元素。
- 由此可得 DIV #2 (z-index 2) 被渲染在 DIV #5 (z-index 1) 之下，因为 DIV #5 归属于 z-index 较高的 DIV #3 元素。
- DIV #3 的 z-index 值是 4，但是这个值独立于 DIV #4, DIV #5 和 DIV #6 的 z-index 值，因为他们从属于不同的层叠上下文。

- 分辨出层叠的元素在 Z 轴上的渲染顺序的一个简单方法是将它们想象成一系列的版本号，子元素是其父元素版本号之下的次要版本。通过这个方法我们可以轻松地看出为什么一个 z-index 为 1 的元素 (DIV #5) 层叠于一个 z-index 为 2 的元素 (DIV #2) 之上，而一个 z-index 为 6 的元素 (DIV #4) 层叠于 z-index 为 5 的元素 (DIV #1) 之下。在我们的例子中 (依照最终渲染次序排列)：
 - Root
 - DIV #2 - z-index 为 2
 - DIV #3 - z-index 为 4
 - DIV #5 - z-index 为 1，在一个 z-index 为 4 的元素内层叠，所以渲染次序为 4.1
 - DIV #6 - z-index 为 3，在一个 z-index 为 4 的元素内层叠，所以渲染次序为 4.3
 - DIV #4 - z-index 为 6，在一个 z-index 为 4 的元素内层叠，所以渲染次序为 4.6
 - DIV #1 - z-index 为 5

示例源码

HTML

```
<div id="div1">
  <h1>Division Element #1</h1>
  <code>position: relative;<br/>
  z-index: 5;</code>
</div>

<div id="div2">
  <h1>Division Element #2</h1>
  <code>position: relative;<br/>
  z-index: 2;</code>
</div>

<div id="div3">
  <div id="div4">
    <h1>Division Element #4</h1>
    <code>position: relative;<br/>
    z-index: 6;</code>
  </div>

  <h1>Division Element #3</h1>
  <code>position: absolute;<br/>
  z-index: 4;</code>

  <div id="div5">
    <h1>Division Element #5</h1>
    <code>position: relative;<br/>
    z-index: 1;</code>
  </div>

  <div id="div6">
    <h1>Division Element #6</h1>
    <code>position: absolute;<br/>
    z-index: 3;</code>
  </div>
</div>
```

CSS

```
* {
    margin: 0;
}
html {
    padding: 20px;
    font: 12px/20px Arial, sans-serif;
}
div {
    opacity: 0.7;
    position: relative;
}
h1 {
    font: inherit;
    font-weight: bold;
}
#div1,
#div2 {
    border: 1px dashed #696;
    padding: 10px;
    background-color: #cfc;
}
#div1 {
    z-index: 5;
    margin-bottom: 190px;
}
#div2 {
    z-index: 2;
}
#div3 {
    z-index: 4;
    opacity: 1;
    position: absolute;
    top: 40px;
    left: 180px;
    width: 330px;
    border: 1px dashed #900;
    background-color: #fdd;
    padding: 40px 20px 20px;
}
#div4,
#div5 {
    border: 1px dashed #996;
    background-color: #ffc;
}
#div4 {
    z-index: 6;
    margin-bottom: 15px;
    padding: 25px 10px 5px;
}
#div5 {
    z-index: 1;
    margin-top: 15px;
    padding: 5px 10px;
```

```
}

#div6 {
    z-index: 3;
    position: absolute;
    top: 20px;
    left: 180px;
    width: 150px;
    height: 125px;
    border: 1px dashed #009;
    padding-top: 125px;
    background-color: #ddf;
    text-align: center;
}
```

Result

参考

- [Stacking without the z-index property](#): The stacking rules that apply when `z-index` is not used.
- [Stacking with floated blocks](#): How floating elements are handled with stacking.
- [Using z-index](#): How to use `z-index` to change default stacking.
- [Stacking context example 1](#) : 2-level HTML hierarchy, z-index on the last level
- [Stacking context example 2](#) : 2-level HTML hierarchy, z-index on all levels
- [Stacking context example 3](#) : 3-level HTML hierarchy, z-index on the second level

原始文档信息

- 作者: Paolo Lombardi
- 此文是我用意大利语写给 [YappY](#) 的英文版本。我授权以 [Creative Commons: Attribution-Sharealike license](#) 将所有内容分享。
- 上次更新时间: July 9th, 2005

Found a problem with this page?

- [Edit on GitHub](#)
- [Source on GitHub](#)
- [Report a problem with this content on GitHub](#)
- Want to fix the problem yourself? See [our Contribution guide](#).

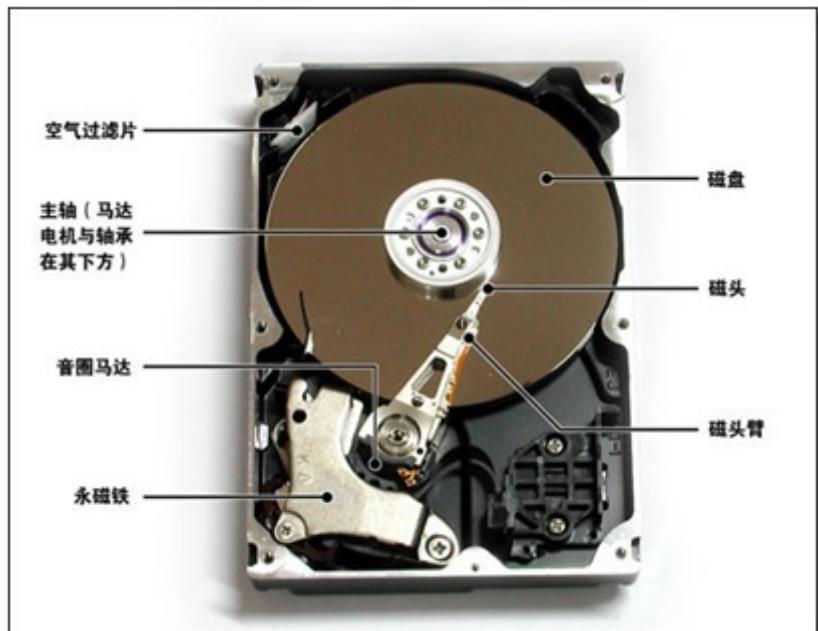
Last modified: 2022年6月1日, [by MDN contributors](#)

本文转自 https://developer.mozilla.org/zh-CN/docs/Web/CSS/CSS_Positioning/Understanding_z_index/The_stacking_context, 如有侵权, 请联系删除。 硬盘的种类主要是SCSI、IDE、以及现在流行的SATA等; 任何一种硬盘的生产都要一定的标准; 随着相应的标准的升级, 硬盘生产技术也在升级; 比如 SCSI标准已经经历了SCSI-1、SCSI-2、SCSI-3; 其中目前咱们经常在服务器网站看到的 Ultra-160就是基于SCSI-3标准的; IDE 遵循的是ATA标准, 而目前流行的SATA, 是ATA标准的升级版本; IDE是并口设备, 而SATA是串口, SATA的发展目的是替换IDE;

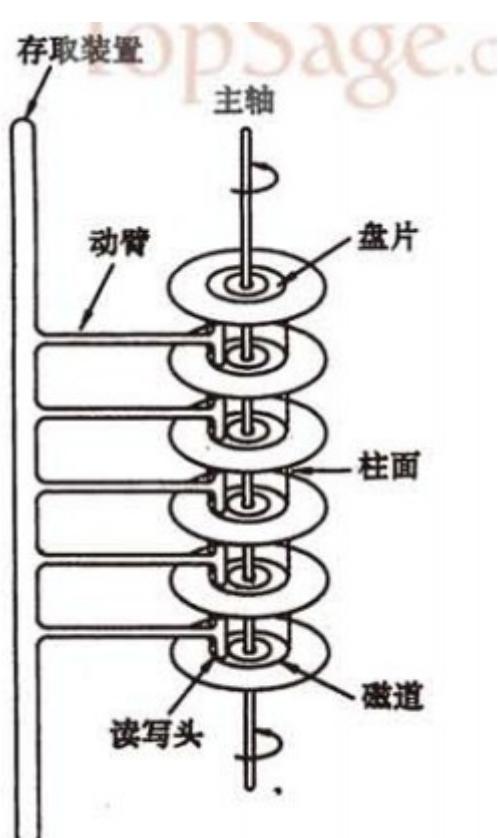
我们知道信息存储在硬盘里，把它拆开也看不见里面有任何东西，只有些盘片。假设，你用显微镜把盘片放大，会看见盘片表面凹凸不平，凸起的地方被磁化，凹的地方是没有被磁化；凸起的地方代表数字1（磁化为1），凹的地方代表数字0。因此硬盘可以以二进制来存储表示文字、图片等信息。

1、硬盘的组成

硬盘大家一定不会陌生，我们可以把它比喻成是我们电脑储存数据和信息的大仓库。一般说来，无论哪种硬盘，都是由盘片、磁头、盘片主轴、控制电机、磁头控制器、数据转换器、接口、缓存等几个部份组成。



平面图：



立体图

所有的盘片都固定在一个旋转轴上，这个轴即盘片主轴。而所有盘片之间是绝对平行的，在每个盘片的存储面上都有一个磁头，磁头与盘片之间的距离比头发丝的直径还小。所有的磁头连在一个磁头控制器上，由磁头控制器负责各个磁头的运动。磁头可沿盘片的半径方向动作，（实际是斜切向运动），每个磁头同一时刻也必须是同轴的，即从正上方向下看，所有磁头任何时候都是重叠的（不过目前已经有多磁头独立技术，可不受此限制）。而盘片以每分钟数千转到上万转的速度在高速旋转，这样磁头就能对盘片上的指定位置进行数据的读写操作。

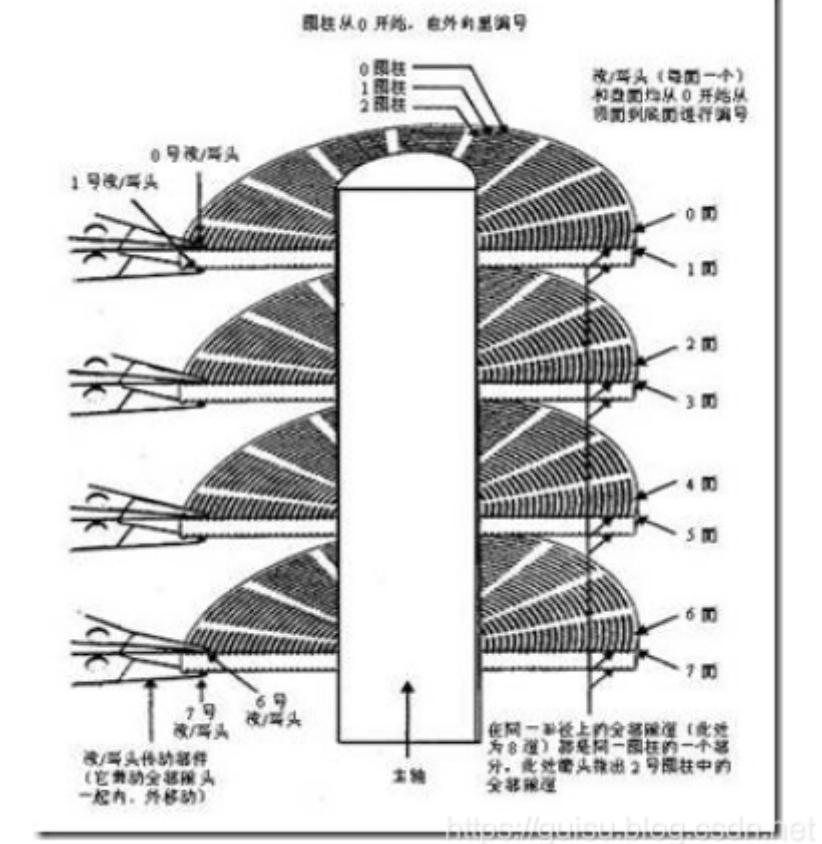


<https://guisu.blog.csdn.net>

由于硬盘是高精密设备，尘埃是其大敌，所以必须完全密封。

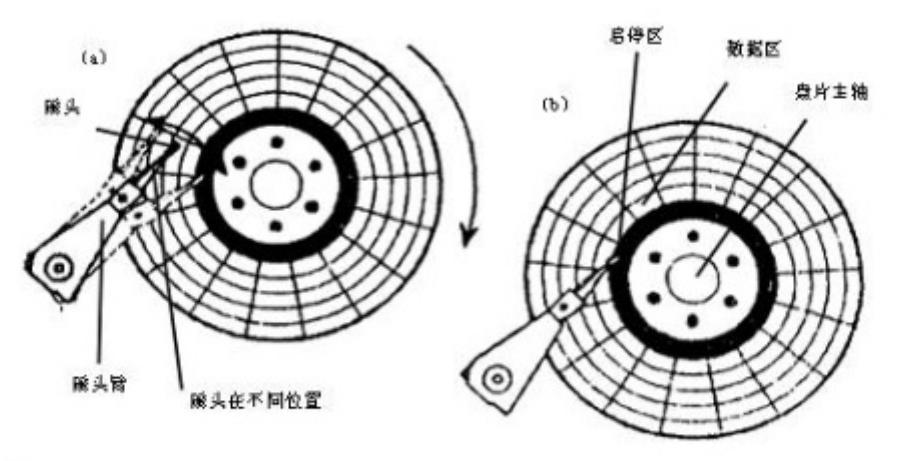
2、硬盘的工作原理

硬盘在逻辑上被划分为磁道、柱面以及扇区。

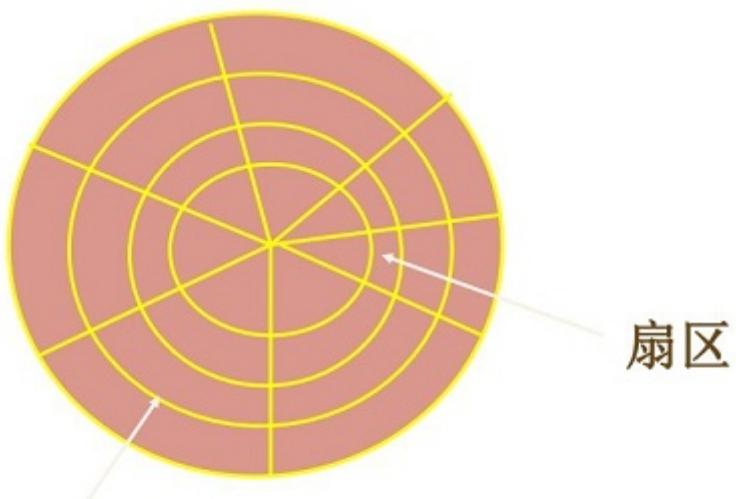
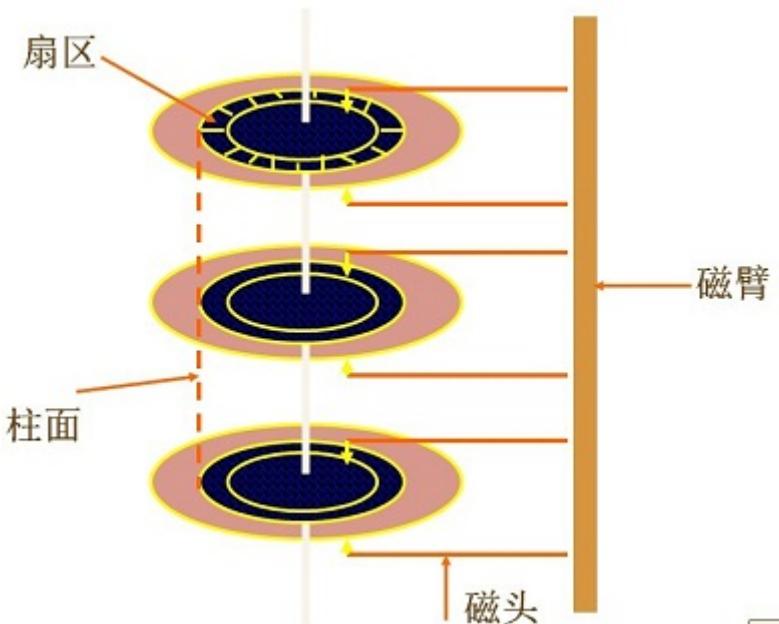


<https://guisu.blog.csdn.net>

硬盘的每个盘片的每个面都有一个读写磁头，磁盘盘面区域的划分如图所示。



<https://guisu.blog.csdn.net>



磁道

磁头靠近主轴接触的表面，即线速度最小的地方，是一个特殊的区域，它不存放任何数据，称为启停区或着陆区（LandingZone），启停区外就是数据区。在最外圈，离主轴最远的地方是“0”磁道，硬盘数据的存放就是从最外圈开始的。那么，磁头是如何找到“0”磁道的位置的呢？在硬盘中还有一个叫“0”磁道检测器的构件，它是用来完成硬盘的初始定位。“0”磁道是如此的重要，以致很多硬盘仅仅因为“0”磁道损坏就报废，这是非常可惜的。

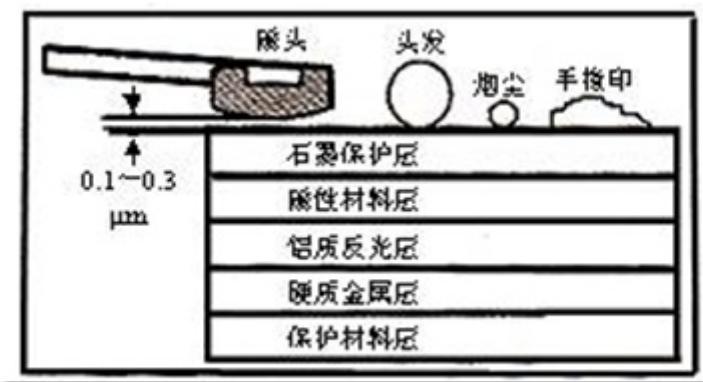
早期的硬盘在每次关机之前需要运行一个被称为Parking的程序，其作用是让磁头回到启停区。现代硬盘在设计上已摒弃了这个虽不复杂却很让人不愉快的小缺陷。硬盘不工作时，磁头停留在启停区，当需要从硬盘读写数据时，磁盘开始旋转。旋转速度达到额定的高速时，磁头就会因盘片旋转产生的气流而抬起，这时磁头才向盘片存放数据的区域移动。

盘片旋转产生的气流相当强，足以使磁头托起，并与盘面保持一个微小的距离。这个距离越小，磁头读写数据的灵敏度就越高，当然对硬盘各部件的要求也越高。早期设计的磁盘驱动器使磁头保持在盘面上方几微米处飞行。稍后一些设计使磁头在盘面上的飞行高度降到约 $0.1\mu\text{m} \sim 0.5\mu\text{m}$ ，现在的水平已经达到 $0.005\mu\text{m} \sim 0.01\mu\text{m}$ ，这只是人类头发直径的千分之一。

气流既能使磁头脱离开盘面，又能使它保持在离盘面足够近的地方，非常紧密地跟随着磁盘表面呈起伏运动，使磁头飞行处于严格受控状态。磁头必须飞行在盘面上方，而不是接触盘面，这种位置可避免擦伤磁性涂层，而更重要的是不让磁性涂层损伤磁头。

但是，磁头也不能离盘面太远，否则，就不能使盘面达到足够强的磁化，难以读出盘上的磁化翻转（磁

极转换形式，是磁盘上实际记录数据的方式）。



<https://guisu.blog.csdn.net>

硬盘驱动器磁头的飞行悬浮高度低、速度快，一旦有小的尘埃进入硬盘密封腔内，或者一旦磁头与盘体发生碰撞，就可能造成数据丢失，形成坏块，甚至造成磁头和盘体的损坏。所以，硬盘系统的密封一定要可靠，在非专业条件下绝对不能开启硬盘密封腔，否则，灰尘进入后会加速硬盘的损坏。另外，硬盘驱动器磁头的寻道伺服电机多采用音圈式旋转或直线运动步进电机，在伺服跟踪的调节下精确地跟踪盘片的磁道，所以，硬盘工作时不要有冲击碰撞，搬动时要小心轻放。

这种硬盘就是采用温彻斯特 (Winchester) 技术制造的硬盘，所以也被称为温盘，目前绝大多数硬盘都采用此技术。

3、盘面、磁道、柱面和扇区

硬盘的读写是和扇区有着紧密关系的。在说扇区和读写原理之前先说一下和扇区相关的“盘面”、“磁道”、和“柱面”。

1. 盘面

硬盘的盘片一般用铝合金材料做基片，高速硬盘也可能用玻璃做基片。硬盘的每一个盘片都有两个盘面 (Side)，即上、下盘面，一般每个盘面都会利用，都可以存储数据，成为有效盘片，也有极个别的硬盘盘面数为单数。每一个这样的有效盘面都有一个盘面号，按顺序从上至下从“0”开始依次编号。在硬盘系统中，盘面号又叫磁头号，因为每一个有效盘面都有一个对应的读写磁头。硬盘的盘片组在2~14片不等，通常有2~3个盘片，故盘面号 (磁头号) 为0~3或0~5。

2. 磁道

磁盘在格式化时被划分成许多同心圆，这些同心圆轨迹叫做磁道 (Track)。磁道从外向内从0开始顺序编号。硬盘的每一个盘面有300~1024个磁道，新式大容量硬盘每面的磁道数更多。信息以脉冲串的形式记录在这些轨迹中，这些同心圆不是连续记录数据，而是被划分成一段段的圆弧，这些圆弧的角速度一样。由于径向长度不一样，所以，线速度也不一样，外圈的线速度较内圈的线速度大，即同样的转速下，外圈在同样时间段里，划过的圆弧长度要比内圈划过的圆弧长度大。每段圆弧叫做一个扇区，扇区从“1”开始编号，每个扇区中的数据作为一个单元同时读出或写入。一个标准的3.5寸硬盘盘面通常有几百到几千条磁道。磁道是“看”不见的，只是盘面上以特殊形式磁化了的一些磁化区，在磁盘格式化时就已规划完毕。

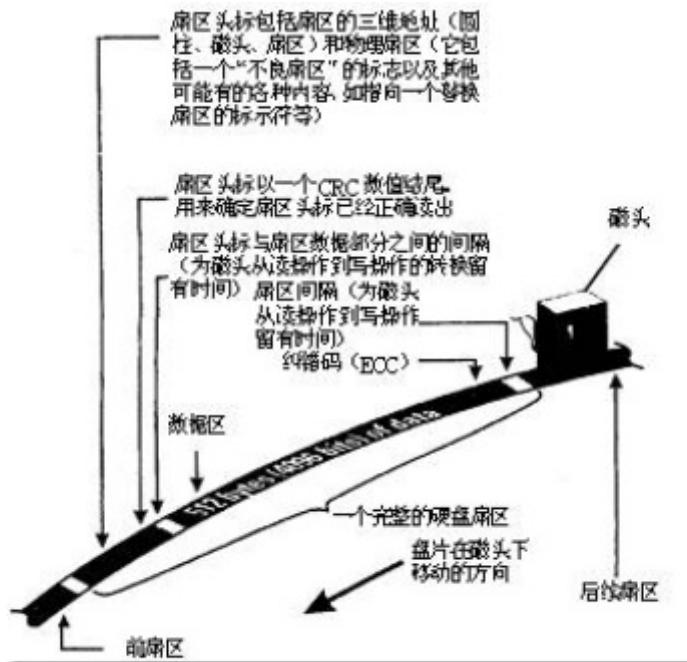
3. 柱面

所有盘面上的同一磁道构成一个圆柱，通常称做柱面 (Cylinder)，每个圆柱上的磁头由上而下从“0”开始编号。数据的读/写按柱面进行，即磁头读/写数据时首先在同一柱面内从“0”磁头开始进行操作，依次向下在同一柱面的不同盘面上进行操作，只在同一柱面所有的磁头全部读/写完毕后磁头才转移到下一柱面 (同心圆的再往里的柱面)，因为选取磁头只需通过电子切换即可，而选取柱面则必须通过机械切换。电子切换相当快，比在机械上磁头向邻近磁道移动快得多，所以，数据的读/写按柱面进行，而不按盘面进行。也就是说，一个磁道写满数据后，就在同一柱面的下一个盘面来写，一个柱面写满后，才移到下一个扇区开始写数据。读数据也按照这种方式进行，这样就提高了硬盘的读/写效率。

一块硬盘驱动器的圆柱数（或每个盘面的磁道数）既取决于每条磁道的宽窄（同样，也与磁头的大小有关），也取决于定位机构所决定的磁道同步距的大小。

4. 扇区

操作系统以扇区（Sector）形式将信息存储在硬盘上，每个扇区包括512个字节的数据和一些其他信息。一个扇区有两个主要部分：存储数据地点的标识符和存储数据的数据段。



<https://guisu.blog.csdn.net>

扇区的第一个主要部分是标识符。标识符，就是扇区头标，包括组成扇区三维地址的三个数字：

盘面号：扇区所在的磁头（或盘面）

柱面号：磁道，确定磁头的径向方向。

扇区号：在磁道上的位置。**也叫块号**。确定了数据在盘片圆圈上的位置。

头标中还包括一个字段，其中有显示扇区是否能可靠存储数据，或者是否已发现某个故障因而不宜使用的标记。有些硬盘控制器在扇区头标中还记录有指示字，可在原扇区出错时指引磁盘转到替换扇区或磁道。最后，扇区头标以循环冗余校验（CRC）值作为结束，以供控制器检验扇区头标的读出情况，确保准确无误。

扇区的第二个主要部分是存储数据的数据段，可分为数据和保护数据的纠错码（ECC）。在初始准备期间，计算机用512个虚拟信息字节（实际数据的存放地）和与这些虚拟信息字节相应的ECC数字填入这个部分。

5. 访盘请求完成过程：

确定磁盘地址（柱面号，磁头号，扇区号），内存地址（源/目）：

当需要从磁盘读取数据时，系统会将数据逻辑地址传给磁盘，磁盘的控制电路按照寻址逻辑将逻辑地址翻译成物理地址，即确定要读的数据在哪个磁道，哪个扇区。

为了读取这个扇区的数据，需要将磁头放到这个扇区上方，为了实现这一点：

- 1) 首先必须找到柱面，即磁头需要移动对准相应磁道，**这个过程叫做寻道**，所耗费时间叫做寻道时间，
- 2) 然后目标扇区旋转到磁头下，即磁盘旋转将目标扇区旋转到磁头下。这个过程耗费的时间叫做旋转时间。

即一次访盘请求（读/写）完成过程由三个动作组成：

- 1) 寻道（时间）：磁头移动定位到指定磁道
- 2) 旋转延迟（时间）：等待指定扇区从磁头下旋转经过
- 3) 数据传输（时间）：数据在磁盘与内存之间的实际传输

因此在磁盘上读取扇区数据（一块数据）所需时间：

$$Ti/o = tseek + tla + n * twm$$

其中：

tseek 为寻道时间

tla 为旋转时间

twm 为传输时间

4、磁盘的读写原理

系统将文件存储到磁盘上时，按柱面、磁头、扇区的方式进行，即最先是第1磁道的第一磁头下（也就是第1盘面的第一磁道）的所有扇区，然后，是同一柱面的下一磁头，……，一个柱面存储满后就推进到下一个柱面，直到把文件内容全部写入磁盘。

（文件的记录在同一盘组上存放是，应先集中放在一个柱面上，然后再顺序存放在相邻的柱面上，对应同一柱面，则应该按盘面的次序顺序存放。）

（从上到下，然后从外到内。数据的读/写按柱面进行，而不按盘面进行，先）

系统也以相同的顺序读出数据。读出数据时通过告诉磁盘控制器要读出扇区所在的柱面号、磁头号和扇区号（物理地址的三个组成部分）进行。磁盘控制器则直接使磁头部件步进到相应的柱面，选通相应的磁头，等待要求的扇区移动到磁头下。在扇区到来时，磁盘控制器读出每个扇区的头标，把这些头标中的地址信息与期待检出的磁头和柱面号做比较（即寻道），然后，寻找要求的扇区号。待磁盘控制器找到该扇区头标时，根据其任务是写扇区还是读扇区，来决定是转换写电路，还是读出数据和尾部记录。找到扇区后，磁盘控制器必须在继续寻找下一个扇区之前对该扇区的信息进行后处理。如果是读数据，控制器计算此数据的ECC码，然后，把ECC码与已记录的ECC码相比较。如果是写数据，控制器计算出此数据的ECC码，与数据一起存储。在控制器对此扇区中的数据进行必要处理期间，磁盘继续旋转。

5、局部性原理与磁盘预读

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的局部性原理：

当一个数据被用到时，其附近的数据也通常会马上被使用。

程序运行期间所需的数据通常比较集中。

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。

预读的长度一般为页 (page) 的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

6、磁盘碎片的产生

俗话说一图胜千言，先用一张ASCII码图来解释为什么会产生磁盘碎片。

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
c	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
e	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
j	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
k	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
l	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
m	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
n	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
o	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
p	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
q	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
r	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
u	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
v	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
w	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

<https://guisu.blog.csdn.net>

上面的ASCII图表示磁盘文件系统，由于目前上面没有任何数据文件，所以我把他表示成0。

在图的最上侧和左侧各有a-z 26个字母，这是用来定位每个数据字节的具体位置，如第1行1列是aa,26行26列是zz。

我们创建一个新文件，理所当然的，我们的文件系统就产生了变化，现在是

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	T	O	C	h	e	l	l	o	.	t	x	t	a	e	l	e	0	0	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
c	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
e	H	e	l	l	o	,		w	o	r	l	d	0	0	0	0	0	0	0	0	0	0	0	0	0
f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

如图所示：“内容表”(TOC)占据了前四行，在TOC里贮存着每件文件在系统里所在的位置。

在上图，TOC包括了一个名字叫hello.txt的文件，其具体内容是“Hello, world”，在系统里的位置是ae到le。

接下来再新建一个文件

```
a b c d e f g h i j k l m n o p q r s t u v w x y z  
a T O C h e l l o . t x t a e l e b y e . t x t m e z  
b e 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
c 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
d 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
e H e l l o , _ w o r l d G o o d b y e , _ w o r l d  
f 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

如图，我们新建的文件bye.txt紧贴着第一个文件hello.txt。

其实这是最理想的系统结构，如果你将你的文件都按照上图所表示的那样一个挨着一个，紧紧的贴放在一起的话，那么读取他们将会非常的容易和迅速，这是因为在硬盘里动得最慢的(相对来说)就是传动手臂，少位移一些，读取文件数据的时间就会快一些。

然而恰恰这就是问题的所在。现在我想在"Hello, World"后加上些感叹号来表达我强烈的感情，现在的问题是：在这样的系统上，文件所在的行就没有地方让我放这些感叹号了，因为bye.txt占据了剩下的位置。

现在有俩个方法可以选择，但是没有一个是完美的

1.我们从原位置删除文件，重新建个文件重新写上"Hello, World!!". -这就无意中延长了文件系统的读和写的时间。

2.打碎文件，就是在别的空的地方写上感叹号，也就是"身首异处"-这个点子不错，速度很快，而且方便，但是，这就同时意味着大大的减慢了读取下一个新文件的时间。

如果你对上面的文字没概念,上图

方法一：

```
a b c d e f g h i j k l m n o p q r s t u v w x y z  
a T O C h e l l o . t x t a f n f b y e . t x t m e z  
b e 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
c 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
d 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
e 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
f H e l l o , _ w o r l d ! ! 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

方法二：

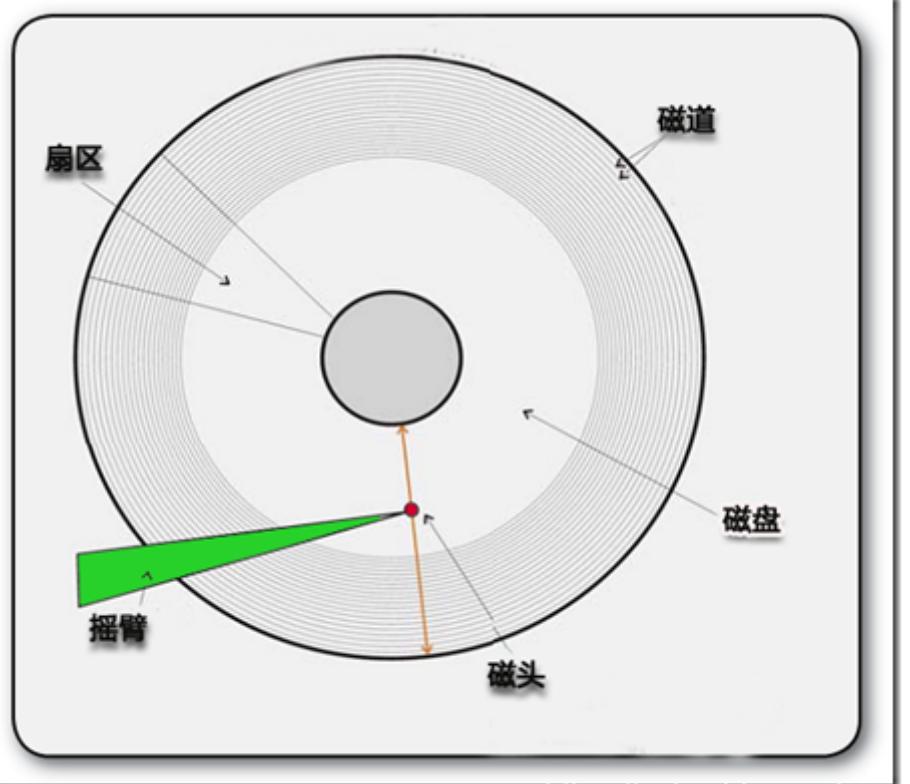
```
a b c d e f g h i j k l m n o p q r s t u v w x y z  
a T O C h e l l o . t x t a e l e a f b f b y e . t x  
b t m e z e 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
c 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
d 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
e H e l l o , _ w o r l d G o o d b y e , _ w o r l d  
f ! ! 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

<https://guisu.blog.csdn.net>

这里所说的方法二就像是我们的windows系统的存储方式，每个文件都是紧挨着的，但如果其中某个文件要更改的话，那么就意味着接下来的数据将会被放在磁盘其他的空余的地方。

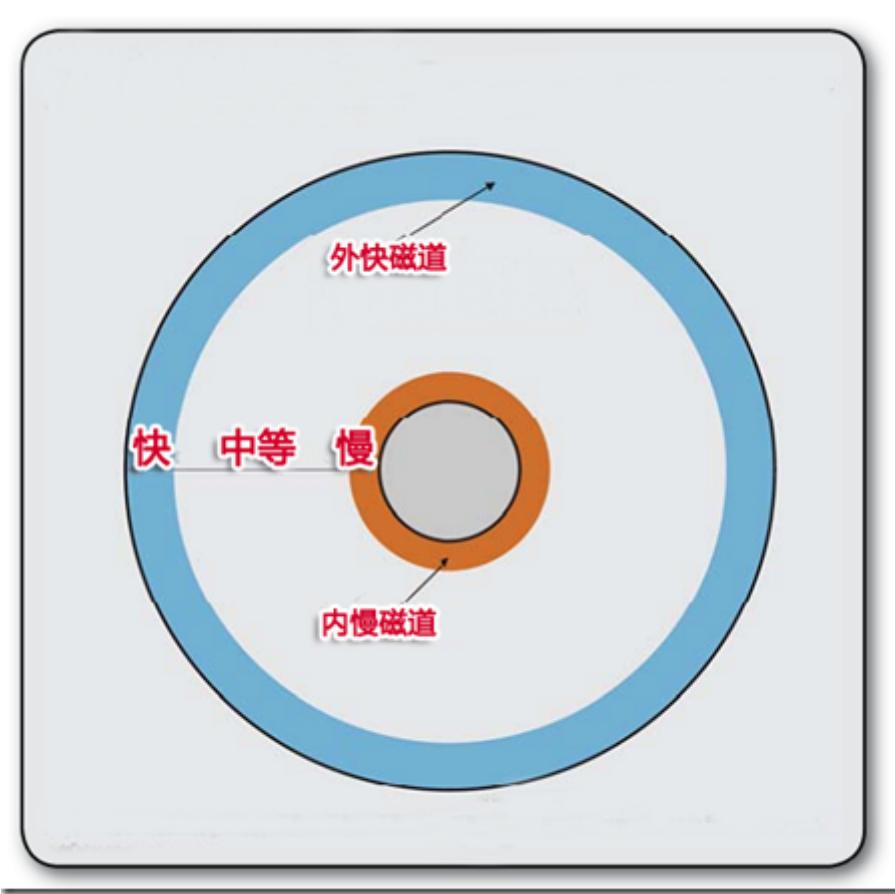
如果这个文件被删除了，那么就会在系统中留下空格，久而久之，我们的文件系统就会变得支离破碎，碎片就是这么产生的。

试着简单点，讲给mm听的硬盘读写原理简化版



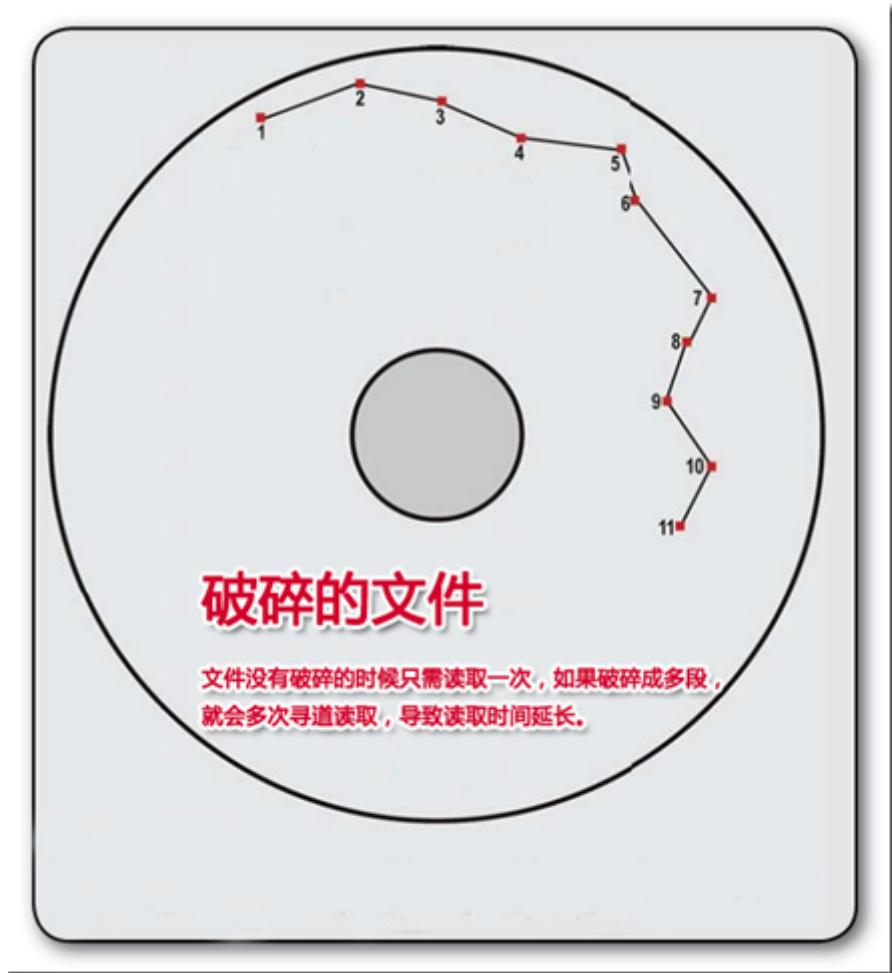
<https://guisu.blog.csdn.net>

硬盘的结构就不多说了,我们平常电脑的数据都是存在磁道上的,大致上和光盘差不多.读取都是靠磁头来进行.



<https://guisu.blog.csdn.net>

我们都知道,我们的数据资料都是以信息的方式存储在盘面的扇区的磁道上,硬盘读取是由摇臂控制磁头从盘面的外侧向内侧进行读写的.所以外侧的数据读取速度会比内侧的数据快很多.



<https://guisu.blog.csdn.net>

其实我们的文件大多数的时候都是破碎的，在文件没有破碎的时候，摇臂只需要寻找1次磁道并由磁头进行读取，只需要1次就可以成功读取；但是如果文件破碎成11处，那么摇臂要来回寻找11次磁道，磁头进行11次读取才能完整的读取这个文件，读取时间相对没有破碎的时候就变得冗长。

因此，磁盘碎片往往也是拖慢系统的重要因素之一，Vista之家团队也计划在Vista优化大师后续版本内加入磁盘碎片整理功能，敬请期待。

7、硬盘容量及分区大小的计算

在linux系统，要计算硬盘容量及分区大小，我们先通过fdisk -l查看硬盘信息：

```
Disk /dev/hda: 80.0 GB, 80026361856 bytes
255 heads, 63 sectors/track, 9729 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Device Boot Start End Blocks Id System
/dev/hda1 * 1 765 6144831 7 HPFS/NTFS
/dev/hda2 766 2805 16386300 c W95 FAT32 (LBA)
/dev/hda3 2806 9729 55617030 5 Extended
/dev/hda5 2806 3825 8193118+ 83 linux
/dev/hda6 3826 5100 10241406 83 linux
/dev/hda7 5101 5198 787153+ 82 linux swap / Solaris
/dev/hda8 5199 6657 11719386 83 linux
/dev/hda9 6658 7751 8787523+ 83 linux
/dev/hda10 7752 9729 15888253+ 83 linux
```

其中

heads 是磁盘面；

sectors 是扇区；

cylinders 是柱面；

每个扇区大小是 512byte，也就是0.5K；

通过上面的例子，我们发现此硬盘有 255个磁盘面，有63个扇区，有9729个柱面；所以整个硬盘体积换算公式应该是：

磁面个数 * 扇区个数 * 每个扇区的大小512 * 柱面个数 = 硬盘体积 (单位bytes)

所以在本例中磁盘的大小应该计算如下：

255 x 63 x 512 x 9729 = 80023749120 bytes

提示：由于硬盘生产商和操作系统换算不太一样，硬盘厂家以10进位的办法来换算，而操作系统是以2进位制来换算，所以在换算成M或者G时，不同的算法结果却不一样；所以我们的硬盘有时标出的是80G，在操作系统下看却少几M；

上面例子中，硬盘厂家算法 和 操作系统算数比较：

硬盘厂家： 80023749120 bytes = 80023749.120 K = 80023.749120 M (向大单位换算，每次除以1000)

操作系统： 80023749120 bytes = 78148192.5 K = 76316.594238281 M (向大单位换算，每次除以1024)

我们在查看分区大小的时候，可以用生产厂家提供的算法来简单推算分区的大小；把小数点向前移动六位就是以G表示的大小；比如 hda1 的大小约为 6.144831G；

磁盘阵列

磁盘阵列是由很多块独立的磁盘，组合成一个容量巨大的磁盘组，利用个别磁盘提供数据所产生的加成效果提升整个磁盘系统效能。利用这项技术，将数据切割成许多区段，分别存放在各个硬盘上。

独立磁盘冗余阵列 (RAID, redundant array of independent disks) 是把相同的数据存储在多个硬盘的不同的地方（因此，冗余地）的方法。通过把数据放在多个硬盘上，输入输出操作能以平衡的方式交叠，改良性能。因为多个硬盘增加了平均故障间隔时间 (MTBF)，储存冗余数据也增加了容错。

RAID技术主要有以下三个基本功能：

- (1)通过对磁盘上的数据进行条带化，实现对数据成块存取，减少磁盘的机械寻道时间，提高了数据存取速度。
- (2)通过对一个阵列中的几块磁盘同时读取，减少了磁盘的机械寻道时间，提高数据存取速度。
- (3)通过镜像或者存储奇偶校验信息的方式，实现了对数据的冗余保护。

优点

- 1) 提高传输速率。** RAID通过在多个磁盘上同时存储和读取数据来大幅提高存储系统的数据吞吐量 (Throughput)。在RAID中，可以让很多磁盘驱动器同时传输数据，而这些磁盘驱动器在逻辑上又是一个磁盘驱动器，所以使用RAID可以达到单个磁盘驱动器几倍、几十倍甚至上百倍的速率。这也是RAID最初想要解决的问题。因为当时CPU的速度增长很快，而磁盘驱动器的数据传输速率无法大幅提高，所以需要有一种方案解决二者之间的矛盾。RAID最后成功了。
- 2) 通过数据校验提供容错功能。** 普通磁盘驱动器无法提供容错功能，如果不包括写在磁盘上的CRC (循环冗余校验) 码的话。RAID容错是建立在每个磁盘驱动器的硬件容错功能之上的，所以它提供更高的安全性。在很多RAID模式中都有较为完备的相互校验/恢复的措施，甚至是直接相互的镜像备份，从而大大提高了RAID系统的容错度，提高了系统的稳定冗余性。

缺点

RAID0没有冗余功能，如果一个磁盘（物理）损坏，则所有的数据都无法使用。

RAID1磁盘的利用率最高只能达到50%(使用两块盘的情况下)，是所有RAID级别中最低的。

RAID0+1以理解为是RAID 0和RAID 1的折中方案。RAID 0+1可以为系统提供数据安全保障，但保障程度要比 Mirror低而磁盘空间利用率要比Mirror高。

本文转自 <https://blog.csdn.net/hguisu/article/details/7408047>，如有侵权，请联系删除。