相比命令式编程，函数式编程更加强调程序执行的结果而非执行的过程，倡导利用若干简单的执行单元让计算结果不断渐进，逐层推导复杂的运算，而非设计一个复杂的执行过程

举个例子，将数组每个元素进行平方操作，命令式编程与函数式编程如下

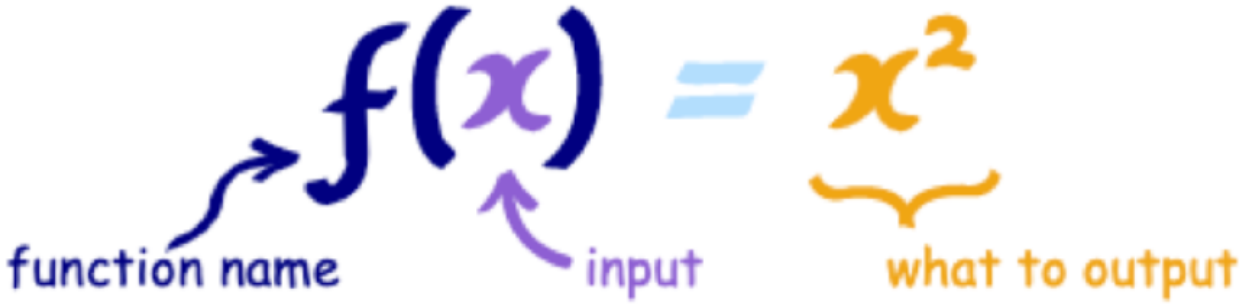
JavaScript | 复制代码

```
// 命令式编程
var array = [0, 1, 2, 3]
for(let i = 0; i < array.length; i++) {
    array[i] = Math.pow(array[i], 2)
}

// 函数式方式
[0, 1, 2, 3].map(num => Math.pow(num, 2))
```

简单来讲，就是要把过程逻辑写成函数，定义好输入参数，只关心它的输出结果

即是一种描述集合和集合之间的转换关系，输入通过函数都会返回有且只有一个输出值



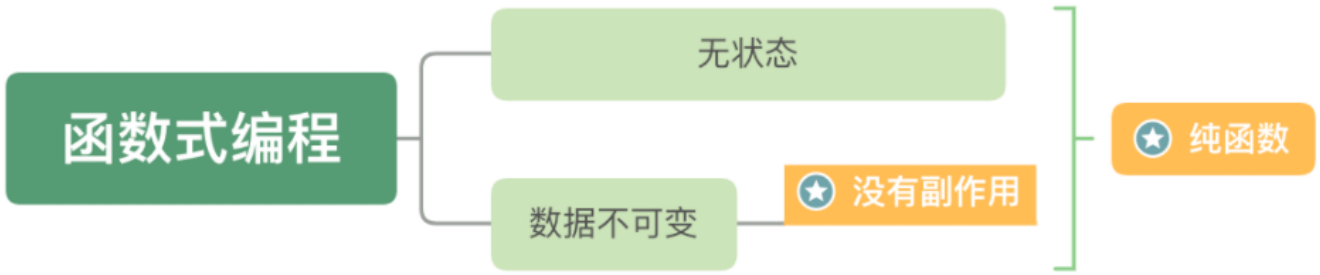
可以看到，函数实际上是一个关系，或者说是一种映射，而这种映射关系是可以组合的，一旦我们知道一个函数的输出类型可以匹配另一个函数的输入，那他们就可以进行组合

30.2. 概念

30.2.1. 纯函数

函数式编程旨在尽可能的提高代码的无状态性和不变性。要做到这一点，就要学会使用无副作用的函数，也就是纯函数

纯函数是对给定的输入返还相同输出的函数，并且要求你所有的数据都是不可变的，即纯函数=无状态+数据不可变



举一个简单的例子

```JavaScript
let double = value=>value*2;
```

特性：

- 函数内部传入指定的值，就会返回确定唯一的值
- 不会造成超出作用域的变化，例如修改全局变量或引用传递的参数

优势：

- 使用纯函数，我们可以产生可测试的代码

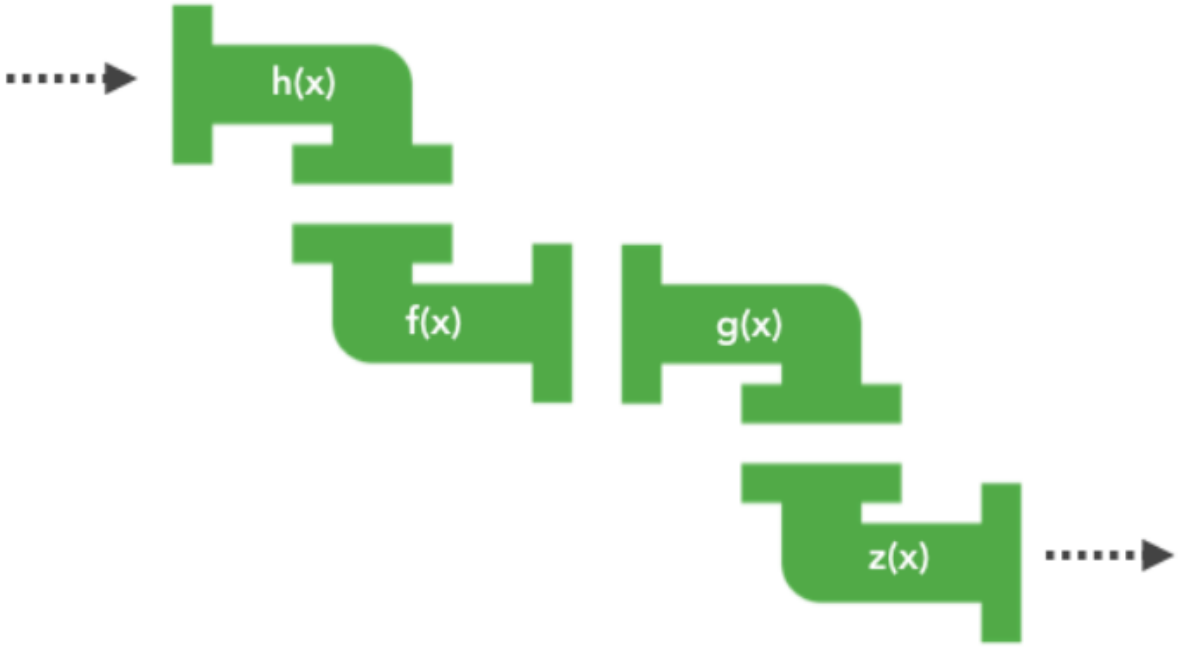
```JavaScript
test('double(2) 等于 4', () => {
  expect(double(2)).toBe(4);
})
```

- 不依赖外部环境计算，不会产生副作用，提高函数的复用性
- 可读性更强 ，函数不管是否是纯函数 都会有一个语义化的名称，更便于阅读
- 可以组装成复杂任务的可能性。符合模块化概念及单一职责原则

30.2.2. 高阶函数

在我们的编程世界中，我们需要处理的其实也只有“数据”和“关系”，而关系就是函数

编程工作也就是在找一种映射关系，一旦关系找到了，问题就解决了，剩下的事情，就是让数据流过这种关系，然后转换成另一个数据，如下图所示



在这里，就是高阶函数的作用。高级函数，就是以函数作为输入或者输出的函数被称为高阶函数

通过高阶函数抽象过程，注重结果，如下面例子

JavaScript | 复制代码

```javascript
const forEach = function(arr,fn){
    for(let i=0;i<arr.length;i++){
        fn(arr[i]);
    }
}
let arr = [1,2,3];
forEach(arr,(item)=>{
    console.log(item);
})
```

上面通过高阶函数 `forEach` 来抽象循环如何做的逻辑，直接关注做了什么

高阶函数存在缓存的特性，主要是利用闭包作用

```javascript
const once = (fn)=>{
    let done = false;
    return function(){
        if(!done){
            fn.apply(this,fn);
        }else{
            console.log("该函数已经执行");
        }
        done = true;
    }
}
```

30.2.3. 柯里化

柯里化是把一个多参数函数转化成一个嵌套的一元函数的过程

一个二元函数如下：

```javascript
let fn = (x,y)=>x+y;
```

转化成柯里化函数如下：

```javascript
const curry = function(fn){
    return function(x){
        return function(y){
            return fn(x,y);
        }
    }
}
let myfn = curry(fn);
console.log( myfn(1)(2) );
```

上面的 `curry` 函数只能处理二元情况，下面再来实现一个实现多参数的情况

```javascript
// 多参数柯里化;
const curry = function(fn){
    return function curriedFn(...args){
        if(args.length<fn.length){
            return function(){
                return curriedFn(...args.concat([...arguments]));
            }
        }
        return fn(...args);
    }
}
const fn = (x,y,z,a)=>x+y+z+a;
const myfn = curry(fn);
console.log(myfn(1)(2)(3)(1));
```

关于柯里化函数的意义如下：

- 让纯函数更纯，每次接受一个参数，松散解耦
- 惰性执行

30.2.4. 组合与管道

组合函数，目的是将多个函数组合成一个函数

举个简单的例子：

```javascript
function afn(a){
    return a*2;
}
function bfn(b){
    return b*3;
}
const compose = (a,b)=>c=>a(b(c));
let myfn =  compose(afn,bfn);
console.log( myfn(2));
```

可以看到 `compose` 实现一个简单的功能：形成了一个新的函数，而这个函数就是一条从 `bfn -> afn` 的流水线

下面再来看看如何实现一个多函数组合：

```JavaScript
const compose = (...fns)=>val=>fns.reverse().reduce((acc,fn)=>fn(acc),val);
```

`compose` 执行是从右到左的。而管道函数，执行顺序是从左到右执行的

```JavaScript
const pipe = (...fns)=>val=>fns.reduce((acc,fn)=>fn(acc),val);
```

组合函数与管道函数的意义在于：可以把很多小函数组合起来完成更复杂的逻辑

30.3. 优缺点

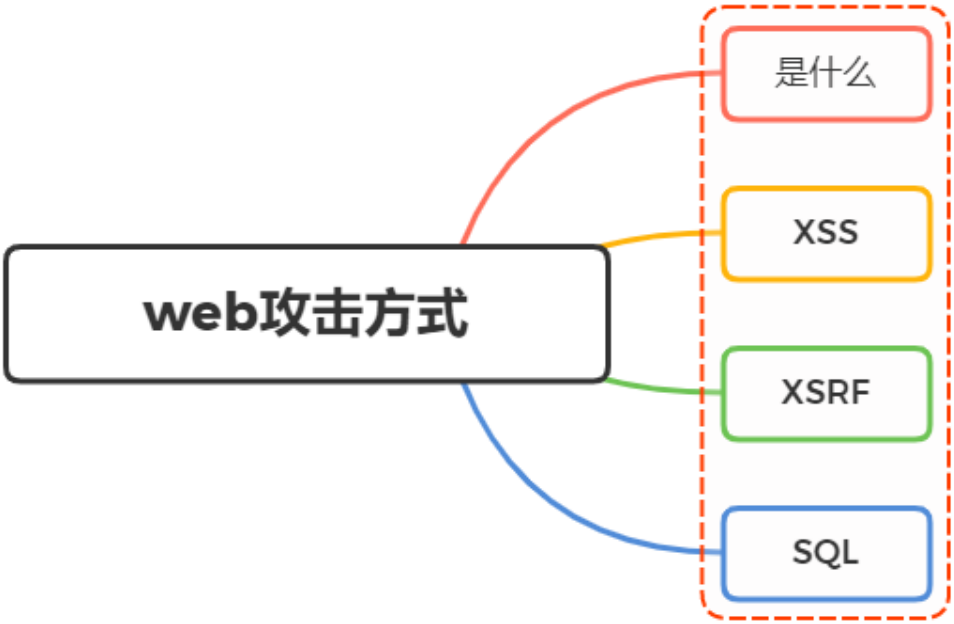
30.3.1. 优点

- 更好的管理状态：因为它的宗旨是无状态，或者说更少的状态，能最大化的减少这些未知、优化代码、减少出错情况
- 更简单的复用：固定输入->固定输出，没有其他外部变量影响，并且无副作用。这样代码复用时，完全不需要考虑它的内部实现和外部影响
- 更优雅的组合：往大的说，网页是由各个组件组成的。往小的说，一个函数也可能是由多个小函数组成的。更强的复用性，带来更强大的组合性
- 隐性好处。减少代码量，提高维护性

30.3.2. 缺点

- 性能：函数式编程相对于指令式编程，性能绝对是一个短板，因为它往往会对一个方法进行过度包装，从而产生上下文切换的性能开销
- 资源占用：在 JS 中为了实现对象状态的不可变，往往会创建新的对象，因此，它对垃圾回收所产生的压力远远超过其他编程方式
- 递归陷阱：在函数式编程中，为了实现迭代，通常会采用递归操作

31. web常见的攻击方式有哪些？如何防御？



31.1. 是什么

Web攻击（WebAttack）是针对用户上网行为或网站服务器等设备进行攻击的行为

如植入恶意代码，修改网站权限，获取网站用户隐私信息等等

Web应用程序的安全性是任何基于Web业务的重要组成部分

确保Web应用程序安全十分重要，即使是代码中很小的 bug 也有可能导致隐私信息被泄露

站点安全就是为保护站点不受未授权的访问、使用、修改和破坏而采取的行为或实践

我们常见的Web攻击方式有

- XSS (Cross Site Scripting) 跨站脚本攻击
- CSRF（Cross-site request forgery）跨站请求伪造
- SQL注入攻击

31.2. XSS

XSS，跨站脚本攻击，允许攻击者将恶意代码植入到提供给其它用户使用的页面中

`XSS` 涉及到三方，即攻击者、客户端与 `Web` 应用

`XSS` 的攻击目标是为了盗取存储在客户端的 `cookie` 或者其他网站用于识别客户端身份的敏感信息。一旦获取到合法用户的信息后，攻击者甚至可以假冒合法用户与网站进行交互

举个例子：

一个搜索页面，根据 `url` 参数决定关键词的内容

```HTML
<input type="text" value="<%= getParameter("keyword") %>">
<button>搜索</button>
<div>
  您搜索的关键词是：<%= getParameter("keyword") %>
</div>
```

这里看似并没有问题，但是如果不按套路出牌呢?

用户输入 `"><script>alert('XSS');</script>` ，拼接到 HTML 中返回给浏览器。形成了如下的 HTML：

```HTML
<input type="text" value=""><script>alert('XSS');</script>">
<button>搜索</button>
<div>
  您搜索的关键词是："><script>alert('XSS');</script>
</div>
```

浏览器无法分辨出 `<script>alert('XSS');</script>` 是恶意代码，因而将其执行，试想一下，如果是获取 `cookie` 发送对黑客服务器呢?

根据攻击的来源， `XSS` 攻击可以分成：

- 存储型
- 反射型
- DOM 型

31.2.1. 存储型

存储型 XSS 的攻击步骤：

1. 攻击者将恶意代码提交到目标网站的数据库中
2. 用户打开目标网站时，网站服务端将恶意代码从数据库取出，拼接在 HTML 中返回给浏览器
3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作

这种攻击常见于带有用户保存数据的网站功能，如论坛发帖、商品评论、用户私信等

31.2.2. 反射型 XSS

反射型 XSS 的攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码
2. 用户打开带有恶意代码的 URL 时，网站服务端将恶意代码从 URL 中取出，拼接在 HTML 中返回给浏览器
3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作

反射型 XSS 跟存储型 XSS 的区别是：存储型 XSS 的恶意代码存在数据库里，反射型 XSS 的恶意代码存在 URL 里。

反射型 XSS 漏洞常见于通过 URL 传递参数的功能，如网站搜索、跳转等。

由于需要用户主动打开恶意的 URL 才能生效，攻击者往往会结合多种手段诱导用户点击。

POST 的内容也可以触发反射型 XSS，只不过其触发条件比较苛刻（需要构造表单提交页面，并引导用户点击），所以非常少见

31.2.3. DOM 型 XSS

DOM 型 XSS 的攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码
2. 用户打开带有恶意代码的 URL
3. 用户浏览器接收到响应后解析执行，前端 JavaScript 取出 URL 中的恶意代码并执行
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作

DOM 型 XSS 跟前两种 XSS 的区别：DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，而其他两种 XSS 都属于服务端的安全漏洞

31.2.4. XSS的预防

通过前面介绍，看到 `XSS` 攻击的两大要素：

- 攻击者提交而恶意代码
- 浏览器执行恶意代码

针对第一个要素，我们在用户输入的过程中，过滤掉用户输入的恶劣代码，然后提交给后端，但是如果攻击者绕开前端请求，直接构造请求就不能预防了

而如果在后端写入数据库前，对输入进行过滤，然后把内容给前端，但是这个内容在不同地方就会有不同显示

例如：

一个正常的用户输入了 `5 < 7` 这个内容，在写入数据库前，被转义，变成了 `5 < 7`

在客户端中，一旦经过了 `escapeHTML()`，客户端显示的内容就变成了乱码(`5 < 7`)

在前端中，不同的位置所需的编码也不同。

- 当 `5 < 7` 作为 HTML 拼接页面时，可以正常显示：

```HTML
<div title="comment">5 &lt; 7</div>
```

- 当 `5 < 7` 通过 Ajax 返回，然后赋值给 JavaScript 的变量时，前端得到的字符串就是转义后的字符。这个内容不能直接用于 Vue 等模板的展示，也不能直接用于内容长度计算。不能用于标题、alert 等

可以看到，过滤并非可靠的，下面就要通过防止浏览器执行恶意代码：

在使用 `.innerHTML`、`.outerHTML`、`document.write()` 时要特别小心，不要把不可信的数据作为 HTML 插到页面上，而应尽量使用 `.textContent`、`.setAttribute()` 等

如果用 `Vue/React` 技术栈，并且不使用 `v-html` / `dangerouslySetInnerHTML` 功能，就在前端 `render` 阶段避免 `innerHTML`、`outerHTML` 的 XSS 隐患

DOM 中的内联事件监听器，如 `location`、`onclick`、`onerror`、`onload`、`onmouseover` 等，`<a>` 标签的 `href` 属性，JavaScript 的 `eval()`、`setTimeout()`、`setInterval()` 等，都能把字符串作为代码运行。如果不可信的数据拼接到字符串中传递给这些 API，很容易产生安全隐患，请务必避免

JavaScript | 复制代码

```
<!-- 链接内包含恶意代码 -->
< a href=" ">1</ a>

<script>
// setTimeout()/setInterval() 中调用恶意代码
setTimeout("UNTRUSTED")
setInterval("UNTRUSTED")

// location 调用恶意代码
location.href = 'UNTRUSTED'

// eval() 中调用恶意代码
eval("UNTRUSTED")
```

31.3. CSRF

CSRF（Cross-site request forgery）跨站请求伪造：攻击者诱导受害者进入第三方网站，在第三方网站中，向被攻击网站发送跨站请求

利用受害者在被攻击网站已经获取的注册凭证，绕过后台的用户验证，达到冒充用户对被攻击的网站执行某项操作的目

一个典型的CSRF攻击有着如下的流程：

- 受害者登录a.com，并保留了登录凭证（Cookie）
- 攻击者引诱受害者访问了b.com
- b.com 向 a.com 发送了一个请求：a.com/act=xx。浏览器会默认携带a.com的Cookie
- a.com接收到请求后，对请求进行验证，并确认是受害者的凭证，误以为是受害者自己发送的请求
- a.com以受害者的名义执行了act=xx
- 攻击完成，攻击者在受害者不知情的情况下，冒充受害者，让a.com执行了自己定义的操作

`csrf` 可以通过 `get` 请求，即通过访问 `img` 的页面后，浏览器自动访问目标地址，发送请求

同样，也可以设置一个自动提交的表单发送 `post` 请求，如下：

JavaScript 复制代码

```
<form action="http://bank.example/withdraw" method=POST>
    <input type="hidden" name="account" value="xiaoming" />
    <input type="hidden" name="amount" value="10000" />
    <input type="hidden" name="for" value="hacker" />
</form>
<script> document.forms[0].submit(); </script>
```

访问该页面后，表单会自动提交，相当于模拟用户完成了一次 `POST` 操作

还有一种为使用 `a` 标签的，需要用户点击链接才会触发

访问该页面后，表单会自动提交，相当于模拟用户完成了一次POST操作

HTML 复制代码

```
< a href="http://test.com/csrf/withdraw.php?amount=1000&for=hacker" taget="_blank">
    重磅消息！！
<a/>
```

31.3.1. CSRF的特点

- 攻击一般发起在第三方网站，而不是被攻击的网站。被攻击的网站无法防止攻击发生
- 攻击利用受害者在被攻击网站的登录凭证，冒充受害者提交操作；而不是直接窃取数据
- 整个过程攻击者并不能获取到受害者的登录凭证，仅仅是“冒用”
- 跨站请求可以用各种方式：图片URL、超链接、CORS、Form提交等等。部分请求方式可以直接嵌入在第三方论坛、文章中，难以进行追踪

31.3.2. CSRF的预防

CSRF通常从第三方网站发起，被攻击的网站无法防止攻击发生，只能通过增强自己网站针对CSRF的防护能力来提升安全性

防止 `csrf` 常用方案如下：

- 阻止不明外域的访问
    - 同源检测
    - Samesite Cookie
- 提交时要求附加本域才能获取的信息

- CSRF Token
- 双重Cookie验证

这里主要讲讲 `token` 这种形式，流程如下：

- 用户打开页面的时候，服务器需要给这个用户生成一个Token
- 对于GET请求，Token将附在请求地址之后。对于 POST 请求来说，要在 form 的最后加上

```
HTML | 复制代码
1  <input type="hidden" name="csrftoken" value="tokenvalue"/>
```

- 当用户从客户端得到了Token，再次提交给服务器的时候，服务器需要判断Token的有效性

31.4. SQL注入

Sql 注入攻击，是通过将恶意的 `Sql` 查询或添加语句插入到应用的输入参数中，再在后台 `Sql` 服务器上解析执行进行的攻击

流程如下所示：

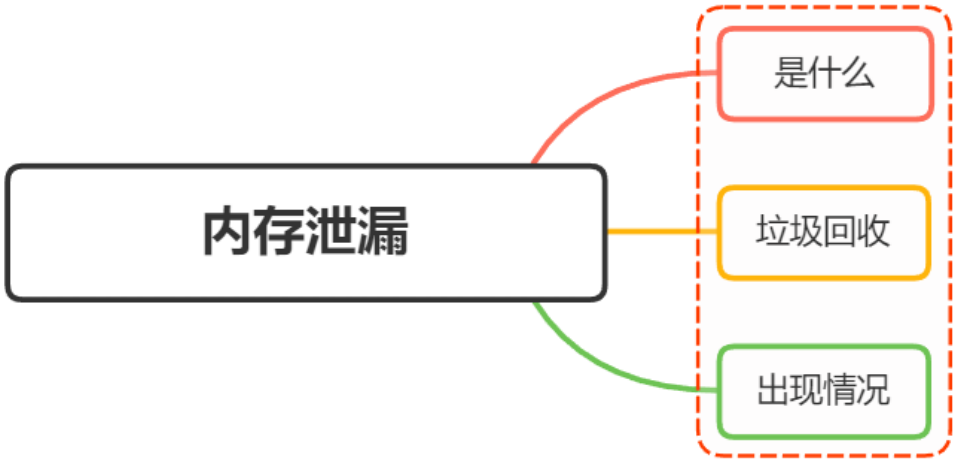
- 找出SQL漏洞的注入点
- 判断数据库的类型以及版本
- 猜解用户名和密码
- 利用工具查找Web后台管理入口
- 入侵和破坏

预防方式如下：

- 严格检查输入变量的类型和格式
- 过滤和转义特殊字符
- 对访问数据库的Web应用程序采用Web应用防火墙

上述只是列举了常见的 `web` 攻击方式，实际开发过程中还会遇到很多安全问题，对于这些问题， 切记不可忽视

32. 说说 JavaScript 中内存泄漏的几种情况?



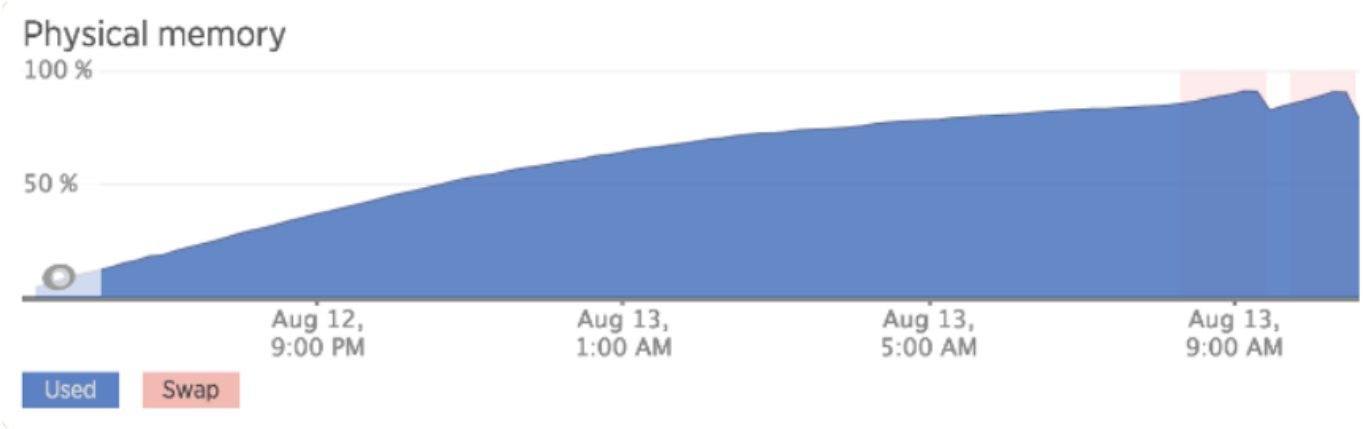
32.1. 是什么

内存泄漏（Memory leak）是在计算机科学中，由于疏忽或错误造成程序未能释放已经不再使用的内存

并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费

程序的运行需要内存。只要程序提出要求，操作系统或者运行时就必须供给内存

对于持续运行的服务进程，必须及时释放不再用到的内存。否则，内存占用越来越高，轻则影响系统性能，重则导致进程崩溃



在 `C` 语言中，因为是手动管理内存，内存泄露是经常出现的事情。

Plain Text | 复制代码

```
char * buffer;
buffer = (char*) malloc(42);

// Do something with buffer

free(buffer);
```

上面是 C 语言代码， `malloc` 方法用来申请内存，使用完毕之后，必须自己用 `free` 方法释放内存。

这很麻烦，所以大多数语言提供自动内存管理，减轻程序员的负担，这被称为"垃圾回收机制"

32.2. 垃圾回收机制

Javascript 具有自动垃圾回收机制（GC：Garbage Collecation），也就是说，执行环境会负责管理代码执行过程中使用的内存

原理：垃圾收集器会定期（周期性）找出那些不在继续使用的变量，然后释放其内存

通常情况下有两种实现方式：

- 标记清除
- 引用计数

32.2.1. 标记清除

`JavaScript` 最常用的垃圾收回机制

当变量进入执行环境是，就标记这个变量为“进入环境“。进入环境的变量所占用的内存就不能释放，当变量离开环境时，则将其标记为“离开环境“

垃圾回收程序运行的时候，会标记内存中存储的所有变量。然后，它会将所有在上下文中的变量，以及被在上下文中的变量引用的变量的标记去掉

在此之后再被加上标记的变量就是待删除的了，原因是任何在上下文中的变量都访问不到它们了

随后垃圾回收程序做一次内存清理，销毁带标记的所有值并收回它们的内存

举个例子：

JavaScript 复制代码

```
var m = 0,n = 19 // 把 m,n,add() 标记为进入环境。
add(m, n) // 把 a, b, c标记为进入环境。
console.log(n) // a,b,c标记为离开环境，等待垃圾回收。
function add(a, b) {
  a++
  var c = a + b
  return c
}
```

32.2.2. 引用计数

语言引擎有一张"引用表"，保存了内存里面所有的资源（通常是各种值）的引用次数。如果一个值的引用次数是 `0` ，就表示这个值不再用到了，因此可以将这块内存释放

如果一个值不再需要了，引用数却不为 `0` ，垃圾回收机制无法释放这块内存，从而导致内存泄漏

JavaScript 复制代码

```
const arr = [1, 2, 3, 4];
console.log('hello world');
```

上面代码中，数组 `[1, 2, 3, 4]` 是一个值，会占用内存。变量 `arr` 是仅有的对这个值的引用，因此引用次数为 `1` 。尽管后面的代码没有用到 `arr` ，它还是会持续占用内存

如果需要这块内存被垃圾回收机制释放，只需要设置如下：

JavaScript 复制代码

```
arr = null
```

通过设置 `arr` 为 `null` ，就解除了对数组 `[1,2,3,4]` 的引用，引用次数变为 0，就被垃圾回收了