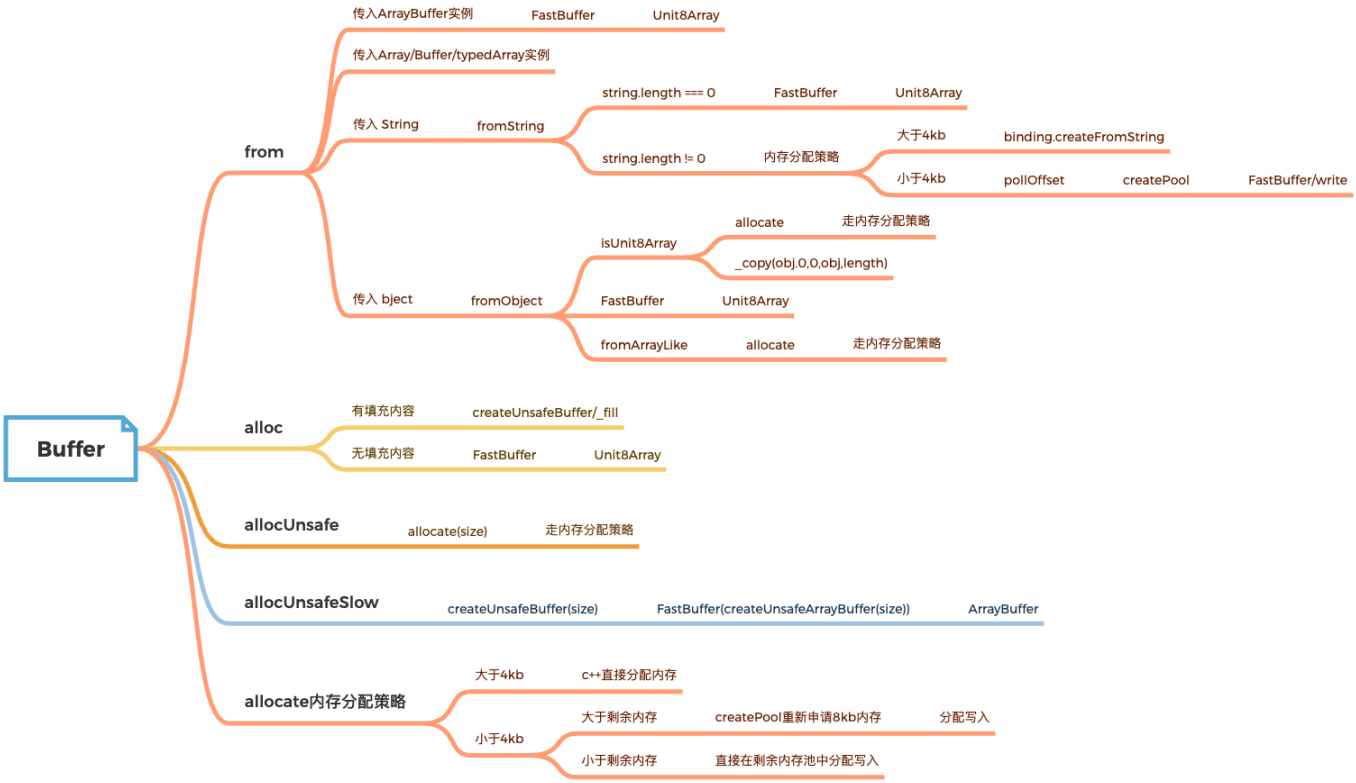
Node 的编码与缓冲 - Buffer

本节目标：实现一个图片拷贝小工具。



二进制数据和编码

计算机的世界只有0和1，文字、图片、视频、应用程序都不例外。我们在网上互相发送信息，本质上都是在交换数据，而这些数据又是由0和1组成，无论消息是中文、英文还是数字、标点符号，都能对应到一段0和1组成的一组二进制数据。

这样的二进制，需要有一种规则来把它对应到正确的字符，这个对应的规则或标准理解回来就是编码，我把字符串用 utf8 编码，那么你收到的 0 和 1 再用同一种编码解码开，就不会乱码了，在 Node 里面，理解字符编码对于我们理解 buffer 和 stream 很有必要，简单看下：

- latin1 (别名 binary)，也就是二进制，0 和 1 组成，很好理解。
- hex 也就是 16 进制，0 - 9 表示 0 到 9，A - F 表示 10 到 15，一个 8 位二进制 0100 0001，前
- 面的四位二进制，也就是 2 的 2 次方是 4，后面的四位二进制，2 的 0 次方是 1，所以可以用 41 表
- 示，41 也就是 ASCII 里面的英文字母 A.

- ASCII 是基于拉丁字母的编码，用 8 个 0/1 二进制表示，前面一个 bit 为 0，后面 7 位定义了 128个字符，其中 95 个是可显示字符，比如英文字母 A，它二进制表示是 0100 0001，十进制是 65，十六进制是 41，进制不同表示不同。
- Unicode 万国码/国际码，是计算机字符的业界标准，为了兼容全世界的语言文字创造了这些字符，每一个字符都能找到对应的一个 Unicode 编码，无论是中文还是英文，最新版本是 2022年9月公布的15.0.0，已经收录超过14万个字符。
- UTF-8 是 Unicode 标准的一种实现而已（还有 UTF-16 等），它可以表示 Unicode 标准的任何字符，第一个字节与 ASCII 兼容，是邮箱网页等应用优先采用的编码，也是 Node Buffer 默认的编码，范围从 U+0000 到 U+10FFFF。
- utf16le (别名 UCS-2)，使用两个字节编码，范围从 0000 - FFFF，当然还有 UCS-4，使用 4 个字节编码，它们都有大端序还有小端序，也就是文件前加上 FF FE 就是小端序，反过来 FE FF 就是大端序，只是编码字符的控制信息而已。
- Base64 基于 64 个可打印字符来表示二进制数据的一种编码格式，比如 Hello 的 base64 是 SGVsbG8=，具体转换过程大家可以自行 Google。

而 binary/hex/ascii/utf8/utf16le/base64，也是 Node 所支持的编码格式，其中 utf8 是默认的编码，而类似 GBK/GB2312 等编码是 Node 无法解析的，可能需要 iconv 这样的三方库来支持，无非是字符和编码集的映射关系而已，既然提到了字符，也提到了 bit，我们再稍微温习下字节和字符。

字节和字符

Byte，也即字节，是一种对数据大小衡量的单位，所以 1GB = 1024MB = 1048576KB =

1073741824B，也就是 10 亿字节，而 1 字节 = 8比特(bit)，也就是 8 位，8 位就是一个长度为 8 的二进制，比如 10011001，它能表示的范围就是从 0-255，也就是 2 的 8 次方，所以 ASCII 实际上可以容纳 256 个字符。

我们知道一个英文字母，占据 1 个字节（8 位二进制）的空间，而一个汉字占据 2 个字节的空间，而字母或汉字，其实就是字符了，所以 1GB 可以容纳 5 亿个汉字。

无论是操作字节还是操作字符，当以流的方式进行，它们本质上就是连续的二进制数据，而字节又是最小单位，所以本质上我们操作的就是字节，也即字节流，无论哪一种，在我们概念里面只要清楚整个计算机数据底层是 0/1 的一坨坨数据就可以了，那流是什么呢，我们往下看。

二进制数据的搬运问题

了解进制、编码、字节和字符后，我们知道数据无非是一坨 0 和 1，配合编码就能传输和解析我们的文字啊，文件啊等等，我们来看第二个问题，也就是数据搬运，你打开一个网页很快，而播放一部电影就可能卡顿，本质是二进制数据搬运传输效率（以及编码解码等）的问题，数据要么在服务器上待着，要么就在去往目的地的路上，我们讨论下搬运的载体和搬运的机制或手段。

想象下，你有一大堆的沙子（假如有 100 GB）需要运走，但是想一次性运走，先不说电脑带宽，光服务器内存可能都吃不消，所以得有一种机制，能不断的把沙子拎出来放到管道里面，像水一样不间断

的流走，直到这一堆沙子完全到达目的地，那么这样的一个机制，在计算机的世界里，就是通过 Stream（流） 来实现数据的动态读写，再借助管道 pipe，流可以被生产也可以被消耗，说白了就是流数据可读可写。

无论多大的数据，无非就是一段段的 0 和 1，可以把它丢到一个管道里，不断的涌向另一个终点，就像水管里的水一样，既然能流动，那每次搬运数据、接收数据的时候，数据都存在哪里呢，为了保证速度应该是放到内存中的吧，那它长什么样子呢，我们接下来就来了解下Buffer，也就是缓冲。

Buffer 基础知识

既然数据是许多个 8 比特数据单位组成，每个 8 比特都有它包含的信息，那么我们就是在不停的操作这些二进制的 8 比特单位数据，当一坨数据被搬运往另外一个地点时，假设 A 是起点（数据文件本身），B是目的地（待写入的数据文件），从 A 往 B 搬运时，一头是 A 不断的提供数据，一头是 B 处理存储数据，当 B 处理数据比较慢，或者从 A 拿出数据比较慢的时候，就会出现两头速度不一致，要么 A 等待B，要么 B 等待 A，这个时候，一定有一部分数据在路上无处安放，那么就需要有一个地方，来暂时存储

这个数据，缓冲一下运输速度，这个地方一般选择在内存里面，这个专门开辟的区域就是缓冲，即 Buffer，在 Node 里面，Buffer 是一个专门提供的 API 接口，挂载到了全局对象上面，不需要用 require 关键字来加载它，我们来到命令行输入 node，进到 repl 模式输入 Buffer：

```
[Function: Buffer] {
  poolSize: 8192, // 分配缓冲区内存的容量
  from: [Function: from], // 根据传入的数据内容创建 buffer
  copyBytesFrom: [Function: copyBytesFrom],
  of: [Function: of],
  alloc: [Function: alloc], // 正常创建 buffer
  allocUnsafe: [Function: allocUnsafe], // 不安全的创建 buffer 方法
  allocUnsafeSlow: [Function: allocUnsafeSlow],// 不安全的创建 buffer 方法
  isBuffer: [Function: isBuffer],// 判断是否是 Buffer 实例对象
  compare: [Function: compare],// 比较两个 Buffer 对象的相对位置
  isEncoding: [Function: isEncoding],// 判断 Nodejs 是否支持某种编码
  concat: [Function: concat],// 拼接几个 Buffer 对象，创建出一个新 Buffer 对象
  byteLength: [Function: byteLength],// 跟进特定编码统计 buffer 字节数
  [Symbol(kIsEncodingSymbol)]: [Function: isEncoding]
}
```

可以发现，Buffer 是一个对象，同时也是一个构造函数，具有自己的属性和静态方法，API 也就这么多了，alloc 顾名思义，就是分配内存，我们来试下创建 Buffer，输入如下代码：

```
const buffer = Buffer.from('Hello, World!');
console.log(buffer)
```

```
3 <Buffer 48 65 6c 6c 6f 2c 20 57 6f 72 6c 64 21>
```

可以看到我们成功创建了一个Buffer对象，在打印的结果中

- <Buffer ...>代表这是一个Buffer对象
- 每个两位数的十六进制数字表示一个字节的数据，它们对应了ASCII字符串的字符编码。例如大写字母H的 ASCII 码是 72（十进制），转换为十六进制是 48，所以在 Buffer 中的第一个字节是 48。

缓冲内存的大小

虽然 Node 的代码运行底层是 V8，实际分配给缓冲的内存是在 C++ 层申请，也就是在 v8 之外的堆内存，所以 Buffer 类更像是一个混合体，底层细节都在 C++ 里面，调度策略这些在 JS 的接口里面，其中一个原因就是 v8 引擎一开始也并不是为服务端设计的，所以它内部的最大可用堆内存只有 1.4G，可以通过传入 --max-old-space-size 新老生代参数来解除限制，但面对大内存管理有时候还是不够用，所以有时候我们运行代码抛出 RangeError 错误的时候，有可能就是内存实例爆池了。

而 Node 里面，要想突破 v8 的内存限制，还有一种手段就是通过 Buffer，Buffer 是通过 C++ 层面申请的，它不走 V8 的内存机制，单个 Buffer 实例在 64 位系统上是 2GB，这个限制在 C++ 层面就已经约束了。

内存分配策略 - 8KB

原本 JS 是没有一种机制来读取或者操作二进制数据流的，但 ES6 里面新增了 TypedArray，而 Node 也

很快跟进，底层则采用 Unit8Array 来为 Buffer 提供数据结构支持，这意味着从 JS 层面有了二进制数据流操作分配和管理的能力，先来了解几个基础概念：

- ArrayBuffer: 内存中一段原始二进制数据，可以通过视图来解读它，视图的意思是，这段数据可以用不同的方式表示，每一种方式就是一个视图
- TypedArray : 描述二进制数据缓存区的视图，类似于数组 Array 的形式，比如有 Int8Array/Int16Array/Uint8Array 等
- Uint8Array : 是数组类型，表示一个 8 位无符号整型数组，长度为一个字节，类似这样：[0, 0, 0,0, 0, 0, 0, 0]，是 TypedArray 的一种实现，那同样还有 Uint16Array 和浮点数组等。

我们看到这些名词，在脑海中把它翻译成一种数据结构就行了，可以存储或者表示二进制数据，基于这些数据结构以及 Node C++ 层面的设计，Node 在创建一个 Buffer 实例的时候，默认会有一个原始大小，刚才在打印 Buffer 时，有一个 poolSize: 8192，这个就是 Buffer 的 8kb 内存池的尺度，当申请的空间小于 4kb 或者大于 4kb，会走不同的分配策略，要了解这些策略，我们就需要前往 Buffer 的 API 用法去了解了。

Buffer 的 API 用法

Buffer.from 创建

Buffer 是一个二进制数据容器，通过 Buffer.from 这种方式是可以直接创建 Buffer 对象，而 from 里面的入参用法有这么几种：

- Buffer.from(array) 传入 8 字节数组，返回八位字节副本的新缓冲区（也就是 8 字节的 buffer）
- Buffer.from(arrayBuffer [，byteOffset [，length]]) 传入数组 Buffer，返回与传入 Buffer 共享相同分配内存的新缓冲区
- Buffer.from(buffer) 传入 Buffer，返回包含该 Buffer 内容副本的新缓冲区
- Buffer.from(string [，encoding]) 传入字符串，返回包含该字符串副本的新缓冲区

光看用法印象不深，我们来看下源码，在 from 方法里面，基于入参做了一些判断，分别调用了 fromString/fromArrayBuffer/fromObject 来走不同的创建策略：

```
// from 里面根据入参分类，还未涉及到 8kb 内存管理的细节
Buffer.from = function from(value, encodingOrOffset, length) {
  // 1. 基于 string 创建
  if (typeof value === 'string') return fromString(value, encodingOrOffse
  t)
  // 2. 基于 ArrayBuffer 创建
  if (isAnyArrayBuffer(value)) return fromArrayBuffer(value, encodingOrOff
  set, length)
  // 3. 函数返回值不等于自身的，通过 from 转成 Buffer
  const valueOf = value.valueOf && value.valueOf()
  if (valueOf !== null && valueOf !== undefined && valueOf !== value)
  return Buffer.from(valueOf, encodingOrOffset, length)
  // 4. 基于 Object 创建
  var b = fromObject(value)
  if (b) return b
  // 5. 如果 value 支持 Symbol.toPrimitive，依然是调用 from 转成 Buffer
  if (typeof value[Symbol.toPrimitive] === 'function')
  return Buffer.from(value
[Symbol.toPrimitive]('string')
, encodingOrOffs
  et, length)
}
```

只看到 fromString/fromArrayBuffer/fromObject 这一层还比较浅，继续向下挖一下看实际上 Buffer 创建细节：

```
// FastBuffer 实际上就是继承了 Unit8Array 的类
class FastBuffer extends Uint8Array {}
```

```
// 而 fromArrayBuffer 是 FastBuffer 的实例，源头依然是 Uint8Array
function fromArrayBuffer(obj, byteOffset, length) {
  return new FastBuffer(obj, byteOffset, length);
}
function fromObject(obj) {
  // 如果是 8 位数组，直接按照数组长度分配内存
  if (isUint8Array(obj)) {
    const b = allocate(obj.length);
    if (b.length === 0) return b;
    _copy(obj, b, 0, 0, obj.length);
    return b;
  }
  // 如果是类数组或者 Buffer，则通过 fromArrayLike 包装返回，否则调用 FastBuffer
  if (obj.length !== undefined || isAnyArrayBuffer(obj.buffer)) {
    if (typeof obj.length !== "number") return new FastBuffer();
    return fromArrayLike(obj);
  }
  if (obj.type === "Buffer" && Array.isArray(obj.data)) {
    return fromArrayLike(obj.data);
  }
}
// fromString，默认就使用 utf8 的编码，除非特别设置
function fromString(string, encoding = "utf8") {
  // 1. 如果传一个空字符串，直接通过 FastBuffer 创建内存
  if (string.length === 0) return new FastBuffer();
  // 2. 基于编码（默认 utf8）计算 string 的长度
  var length = byteLengthUtf8(string);
  if (encoding !== "utf8") length = byteLength(string, encoding, true);
  // 3. 字符串字节数大于 4KB，通过内置原生的 createFromString 分配内存
  if (length >= Buffer.poolSize >>> 1)
    return binding.createFromString(string, encoding);
  // 4. 所需的字节长度大于剩余空间，重新申请 8K 内存
  if (length > poolSize - poolOffset) createPool();
  // 创建 FastBuffer 对象 写入数据
  var b = new FastBuffer(allocPool, poolOffset, length);
  const actual = b.write(string, encoding);
  if (actual !== length) {
    b = new FastBuffer(allocPool, poolOffset, actual);
  }
  // 修正 pool 偏移量，调用 alignPool 进行校准
  poolOffset += actual;
  alignPool();
  return b;
}
```

Buffer.alloc 创建

除了 from，alloc 也可以创建缓冲内存，它有三种用法，一种是 safe 模式，两种是 unsafe 模式：

- Buffer.alloc(size [，fill [，encoding]]) 传入数值，返回指定大小的 Buffer 实例，比 Unsafe 的创建方法慢，但不包含旧的或者潜在的敏感数据，更安全
- Buffer.allocUnsafe(size) Buffer.allocUnsafeSlow(size) 传入数值，都返回一个指定大小的新的缓冲区，必须通过 buf.fill(0) 初始化，或者完全写入内容以覆盖旧数据

alloc 就是传递一个 size， 以字节为单位, 传参给 alloc 生成一段内存区间，比如：

```
// 初始化一个八位字节长度的 buffer
const bufFromAlloc = Buffer.alloc(8)
console.log(bufFromAlloc)
// <Buffer 00 00 00 00 00 00 00 00>
// 这个实例化后的 buf，有一个 length 的属性，来表示缓冲区的大小
console.log(buf.length)
// 8
```

通过 alloc 分配的内存区间是有固定长度的，如果写入超过长度，那么超出部分是不会被缓冲的：

```
const bufFromAll = Buffer.alloc(8)
bufFromAll.write('123456789')
console.log(bufFromAll)
// <Buffer 31 32 33 34 35 36 37 38>
console.log(bufFromAll.toString())
// 12345678
```

来总结下 from 和 alloc，他们里面关于 8kb 的部分，简言之就是 Node 会准备好了一个内存缓冲区，每次创建 Buffer 的时候，会尽量使用缓冲池里面已有的空闲内存，来节省申请内存本身的开销，如果大于4k 直接申请新内存，如果小于 4kb 而空余的内存够用就直接用，不够用依然重新申请，整理如下：

- from 传入 ArrayBuffer，通过 FastBuffer（继承 Uint8Array） 来创建内存缓冲区
- from 传入 String，如果小于 4k 使用 8k 池创建（剩余空间不够用再去申请），大于 4k 调用 binding.createFromString() 创建
- from 传入 Object，小于 4k 使用 8k 池创建（剩余空间不够用再去申请），大于 4k 调用 createUnsafeBuffer()，这个 object 不是普通的 obj，需要支持 Symbol.toPrimitive or valueOf() 才可以，见这里。
- Buffer.alloc()，用给定字符填充一定长度的内存缓冲，或者用 0 填充
- Buffer.allocUnsafe()，小于 4k 使用 8k pook，大于 4k 调用 createUnsafeBuffer()
- Buffer.allocUnsafeSlow()，调用 createUnsafeBuffer()

缓冲写入 Buffer write

如果要将字符串当做二进制数据来使用，只需将该字符串作为 Buffer from 的参数传入即可，但是有时候，我们需要向已经创建的 Buffer 对象中写入新的字符串，这时就可以使用 write 方法来完成，在write方法中，可以使用四个参数：

buf.write(string[, offset[, length]][, encoding])

第一个参数：必须，用于指定需要写入的字符串

第二个参数 offset 指定字符串转换为字节数据后的写入位置

第三个参数 length 指定字符串写入长度

第四个参数用于指定写入字符串时，使用的编码格式，默认为 utf8 格式

```
let bufForWrite = Buffer.alloc(32)
bufForWrite.write('hello xiaojuSurvey', 0, 10)
console.log(bufForWrite.toString())
// hello xiao
```

数组截取 Buffer slice

buf.slice([start[, end]])

Buffer 的截取跟数组类似：

```
let bufFromArr1 = Buffer.from([1, 2, 3, 4, 5])
// <Buffer 01 02 03 04 05>
let bufFromArr2 = bufFromArr1.slice(2, 4)
// <Buffer 03 04>
```

两个参数都是可选项，start 和 end 也可以是负值，为负值时，会首先把这个负值和 Buffer 的长度相加，然后变为正值之后，再做处理。

与 JS 不同的是，如果你修改了 slice 返回的 Buffer 对象中的属性值，那么原来的 Buffer 实例中对应的值，也会被修改，因为 Buffer 中保存的是一个类似指针的东西，指向同一段存储空间，不管以哪一个变量或者指针，都可以修改这段存储空间的值，再通过其他变量或者指针访问该属性时，获取到的也是修改后的值。

数组拷贝 Buffer copy

buf.copy(target[, targetStart[, sourceStart[, sourceEnd]]])

copy 支持四个参数：

第一个参数指定复制的目标 Buffer。

第二个参数指定目标 Buffer 从第几个字节开始写入数据，默认为 0（从开始出写入数据）

第三个参数指定从复制源 Buffer 中获取数据时的开始位置，默认值为0，即从第一个数据开始获取数据。

第四个参数指定从复制源 Buffer 中获取数据的结束位置，默认值为复制源 Buffer 的长度，即 Buffer 的结尾。

```
let bufCopy1 = Buffer.from('Hello')
let bufCopy2 = Buffer.alloc(4)
console.log(bufCopy1)
// <Buffer 48 65 6c 6c 6f>
bufCopy1.copy(bufCopy2, 0, 1, 5)
console.log(bufCopy2)
<Buffer 65 6c 6c 6f>
console.log(bufCopy2.toString())
// ello
```

缓冲填充 Buffer fill

buf.fill(value[, offset[, end]][, encoding])

fill支持三个参数：

第一个参数指定被写入的数值

第二个参数指定从第几个字节开始写入，默认值为 0，也就是从缓存区起始位置写入

第三个参数指定将数值一直写入到第几个字节结束，默认是 Buffer 的 length，也就是写到缓存区尾部

最后一个参数是指定编码

```
const bufForFill = Buffer.alloc(12).fill('11-11 ')
// <Buffer 31 31 2d 31 31 20 31 31 2d 31 31 20>
console.log(bufForFill.toString())
// 11-11 11-11
```

Buffer 还有更多的方法，我们不再一一举例，在认知层面，我们知道它可以在内存里面，申请和存储一段数据就可以了，它的好处是就是帮我们把数据先缓冲起来，用的时候开箱即用，减少 IO 等层面的开销，最重要的是，通过它的缓冲积压，来为流的读写提供一个中间地带，以达到缓冲缓速的作用。

编程练习 - 拷贝图片的小工具

```
// 使用fs的promise api
const fs = require("fs/promises");
!(async function () {
  try {
    // 通过 fs.readFile 读取图片时候，拿到的是缓冲的 Buffer 数据
    const buffer = await fs.readFile("assets/logo.jpg");
    // 把读取到的 Buffer 数据，通过 fs writeFile 写入到一个新图片文件中
    await fs.writeFile("assets/logo-2.jpg", buffer);
    // 再基于原始的 Buffer 创建一个新的 Buffer，通过 toString base64 解码为字符串打印出来;
    const base64Image = Buffer.from(buffer).toString("base64");
    console.log(base64Image);
    // base64Image 是 base64 后的字符串，传参给 from，同时指定编码生成一个新的 Buffer 实例
    const decodedImage = Buffer.from(base64Image, "base64");
    // 比较两个 Buffer 实例的数据
    console.log(Buffer.compare(buffer, decodedImage));
    // 写入到一个新的图片中
    await fs.writeFile("assets/logo_decoded.jpg", decodedImage);
  } catch (error) {
    console.log('复制文件失败', error);
  }
})();
```