# 一 非utf-8编码文本处理

## 1 字符串基本原则

如果你需要对非 UTF8 编码的文本进行处理，如 GBK 编码的，那么我们只有遵循以下原则处理就ok：

fs.readFile(filepath, cb)

=> GBK encoding Buffer

=> iconv decode gbk

=> utf8 js String

=> 处理String

=> iconv encode gbk

=> GBK encoding Buffer

=> send to client

如果你只需要将 GBK 编码的文本直接输出，那么就不需要转换，直接将 GBK 编码的 Buffer 输出即可.

res.send(gbkBuffer);

## 2 示例

将 GBK 模板文件进行简单处理，能按 UTF8 和 GBK 两种编码选择输出.

var connect = require('connect');var fs = require('fs');var path = require('path');var iconv = require('iconv-lite');

var app = connect();

var filepath = path.join(\_\_dirname, 'index.html');var content = fs.readFileSync(filepath);

app.use(function (req, res) {

var s = iconv.decode(content, 'gbk');

if (req.url === '/utf8') {

res.setHeader('Content-Type', 'text/html; charset=utf8');

s = s.replace('<title>淘宝网 - 淘！我喜欢</title>', '<title>淘宝网 - 淘！我喜欢 UTF8</title>')

.replace('<meta charset="gbk">', '<meta charset="utf8">')

.replace('你好啊', '你好啊 UTF8');

// show utf8 page

return res.end(s);

}

// show gbk page

res.setHeader('Content-Type', 'text/html; charset=gbk');

s = s.replace('<title>淘宝网 - 淘！我喜欢</title>', '<title>淘宝网 - 淘！我喜欢 GBK</title>')

.replace('你好啊', '你好啊 GBK');

res.end(iconv.encode(s, 'gbk'));});

app.listen(1984);

# 二 事件队列



# 三 Node异常逻辑处理

Node是单线程的，为了保证服务器不崩溃，必须保证项目的高容错性。

require一个文件时，如果文件不存在，那么会出现一些可能的突发情况：

require('http').createServer(function (req,res) {  
 const test = require('test');  
 res.end('hello world');  
}).listen(80);

这个代码中，启动服务不会有任何异常，但是一旦访问地址，就会报错终止进程。

我们可以使用try catch来避免这种异常导致的进程终止。

require('http').createServer(function (req,res) {  
 try{  
 const test = require('test');  
 } catch (e){  
 console.log(e);  
 }  
 res.end('hello world');  
}).listen(80);

# 四 Node缓存

const http = require('http');  
const fs = require('fs');  
const url = require('url');  
  
http.createServer((req, res) => {  
 let {pathname} = url.parse(req.url);  
  
 *//获取文件日期* fs.stat(`www${pathname}`, (err, stat) => {  
 if (err) {  
 res.writeHeader(404);  
 res.write('Not Found');  
 res.end();  
 } else {  
 if (req.headers['if-modified-since']) {  
 let oDate = new Date(req.headers['if-modified-since']);  
 let time\_client = Math.floor(oDate.getTime() / 1000);  
  
 let time\_server = Math.floor(stat.mtime.getTime() / 1000);  
  
 if (time\_server > time\_client) { *//服务器的文件时间>客户端手里的版本* sendFileToClient();  
 } else {  
 res.writeHeader(304);  
 res.write('Not Modified');  
 res.end();  
 }  
 } else {  
 sendFileToClient();  
 }  
  
 function sendFileToClient() {  
 *//发送* let rs = fs.createReadStream(`www${pathname}`);  
 res.setHeader('Last-Modified', stat.mtime.toGMTString());  
  
 *//输出* rs.pipe(res);  
  
 rs.on('error', err => {  
 res.writeHeader(404);  
 res.write('Not Found');  
 res.end();  
 });  
 }  
 }  
 });  
}).listen(8080);



# 五 事件循环与监听

首先，Node不为每个用户开辟一个线程，所以非常极端的选择了单线程。单线程，要照顾所有的用户，那么就必须有非阻塞I/O，否则一个人的I/O就把别人、自己都阻塞了。一旦有非阻塞I/O，一个人如果I/O去了，就会放弃CPU的使用权，换成另一个人使用CPU（或者执行此人后面的语句）。所以CPU的利用率100%。第一个人I/O结束了，就要用事件来通知线程，执行回调函数。此时必须有事件环，就有一个排队调度机制。Node中有超过半数的C++代码，在搭建事件环。

N

ode的事件监听

const EventEmitter = require('events').EventEmitter;  
  
let channel = new EventEmitter();  
  
channel.on('join',function () {  
 console.log('test');  
});  
*//发射事件:只有发射了事件才能执行join对应的函数，事件只是个键，可以是任何字符串*channel.emit('join');  
  
*//错误处理*channel.on('error',function (err) {  
 console.log(err.message);  
});  
channel.emit('error',new Error('Something is wrong!'));

# 六 V8

## 1 V8隐藏类

V8设计之初，是为了加快Chrome浏览器执行网页脚本的效率，当网页加载完成，V8一步到位，编译成机器码，CPU就开始执行了。V8省去了生成中间码解释执行的过程，程序更早运行，直接执行编译好的机器指令，不足的是，代码的优化变得非常困难。

在静态类型语言中，每一个变量，都有唯一确定的类型，对象的成员信息在编译阶段就可确定，执行时CPU只需要用对象首地址---在C++中是this指针，加上成员在对象内部的偏移量即可访问内部成员。但是JS中，变量在运行时可以随时由不同类型的对象赋值，并且对象本身可以随时添加成员。访问对象属性需要的信息完完完全由运行时决定。为了实现按照索引方式访问成员，V8内部给运行中的对象分了类，产生了V8内部的数据结构，即隐藏类，隐藏类本身是一个对象。

当定义一个构造函数，使用这个函数生成第一个对象时，V8会为它初始化一个隐藏类。以后使用这个构造函数生成的对象指向同一个隐藏类。但假如程序中对某个对象添加或者删除了某个属性，V8立即创建一个新的隐藏类，改变之后的对象指向新的隐藏类。

所以，隐藏类起到了给对象分组的作用。同一组对象，具有相同的成员名称，隐藏类记录了成员名称和偏移量，根据这些信息，V8能够按照对象首地址+偏移量访问成员变量。在程序中，访问对象成员非常频繁，相比于把属性名作为键值，使用字典查找的方式存取成员，使用索引的方式对性能的改进更明显。

借助隐藏类，可以使用数组索引的方式存取对象成员。但成员的索引值是以哈希表的方式存储在隐藏类中。如果每次访问属性都搜寻隐藏类的哈希表，那么这种用偏移量的方式不会带来任何好处。内联缓存是基于程序运行的局部性原理，动态生成使用索引查找的代码。下一次存取成员不必再去搜寻哈希表。

V8还使用了Crankshaft编译器生成更高效的机器码。当V8发现某函数执行频繁，就会将其标记为热点函数，V8会认为该函数比较稳定，类型已经确定，会调用Crankshaft编译器，生成更高效的机器码，只有遇到类型变化，才会回退到优化前的情况。

示例：

*//代码1*let obj1 = {};  
obj1.name = 'lisi';  
*//代码2*let obj2 = {  
 name:'lisi'  
};

代码2的效率要比代码1更高，因为代码1是在动态的添加属性。

当然我们也可以借助typescript来进行类型检测。

## 2 垃圾回收与内存控制

JS对象在V8的堆中创建，V8会自动回收不被引用的对象，这样虽然降低了内存管理的负担，但是也引起了一些不便，如：V8堆内存的内存代销限制。在32位系统上限制为0.7GB，64位限制为1.4GB，之所以存在这种限制，根源在于垃圾回收算法的限制。V8在执行垃圾回收的时候会阻塞JS代码的运行，堆内存过大导致回收算法执行时间过长。所以有垃圾回收的地方，就会存在堆内存限制。

V8的堆分为三部分：年轻分代、年老分代、大对象空间，三者保存不同对象。

### 2.1 年轻分代

年轻分代的堆空间一分为二，只有一半处于使用中，另外一半用于清理垃圾。年轻分代主要用来保存声明周期短暂的对象，例如函数中的局部变量，当函数返回，调用栈中的局部变量就会被析构掉。当V8发现内存空间不够时，才会进行回收。

回收步骤是：

将还被引用的对象复制到另一半区域；

释放当前一半空间，

把当前被释放的空间留作备用，两者角色互换。

年轻分代类似线程的栈空间，本身不太大，占用它空间的对象类似C++中的局部对象，生命周期非常短，因此大部分都是需要被清理掉的，需要赋值对象极少，虽然牺牲了部分内存，但是速度极快。

### 2.2 年老分代

年老分代类似C++中使用new操作符在堆中分配的对象。因为这类对象一般不会因为函数退出而销毁，因此生命周期较长。年老分代的大小远大于年轻分代（32位为700M，64位为1.4GB），包含：

从年轻分代中移动过来的对象；

JIT后产生的代码；

全局对象。  
 如果年老分代采用年轻分代一样的清理算法，浪费一般空间不说，复制大块对象在时间上让人无法忍受，因此必须采用新的方式。V8采用标记清除和标记整理的算法即将垃圾回收分为2个过程：

标记清除阶段遍历堆中的所有对象，把有效的对象标记出来，之后清除垃圾对象。因为年老分代中需要回收的对象比例极小，所以效率极高。

当执行完一次标记清除后，堆内存变得不连续，内存碎片的存在使得不能有效使用内存。在后续的执行中，当遇到没有一块碎片内存能够满足申请对象需要的内存空间时，将处罚V8执行标记整理算法：标记整理移动对象，紧缩V8堆空间，将碎片的内存整理为 大块内存。

由于垃圾回收会阻塞JS代码的运行，上述两种算法是在JS运行时，会采取交替运行的方式，有效的减少了垃圾回收给程序造成的最大停顿时间。

### 2.3 大对象空间

大对象空间主要存储数据和JIT代码，垃圾回收不会移动大对象，这部分内存使用的特点是：整块分配，一次性整块回收。

### 2.4 使用Buffer

Buffer使用堆外内存，操作文件或者发起网络请求时，应该直接使用Buffer操作数据，而不是将其转成字符串，这样可以显著提升效率。

Buffer在堆外申请的空间释放时间是在Buffer对象被垃圾回收时，我们不能决定V8什么时候进行垃圾回收，因此在高并发使用Buffer时，有可能造成Buffer维护的堆外内存迟迟无法释放。这时，可以考虑引入第三方模块，用来手动释放Buffer空间。

Node目前使用的Buffer是基于V8的Unit8Array类，这个类提供了将堆内存的控制权交出的函数，可以很容易实现手工释放内存。

#include <stdlib.h>  
#include <Node.h>  
#include <v8.h>  
#include <node\_buffer.h>  
using v8::ArrayBuffer;  
using v8::HandleScope;  
using v8::Isolate;  
using v8::Local;  
using v8::Object;  
using v8::Value;  
using v8::Unit8Array;  
  
inline bool HasInstance(Local<Object> obj){  
 return obj->IsUnit8Array;  
}  
  
void Method(const v8::FunctionCallbackInfo::Value>& args){  
 Isolate\* isolate = args.GetIsolate();  
 HandleScope scope(isolate);  
 Local<Object> buf = args[0].As<Object>();  
 if(!HasInstance(buf)){  
 return;  
 }  
 Local<Unit8Array> array = buf.As<Unit8Array>();  
 if(array->Buffer()->GetContents().ByteLength() <= 8 \* 1024 || array->Buffer()->IsExternal)  
 return;  
 int64\_t change\_in\_bytes = -static\_cast<int64\_t>(array->Buffer()->GetContents().ByteLength());  
 ArrayBuffer::Contents array\_c = array-Buffer()->Externalize();  
 free(array\_c.Data());  
 isolate->AdjustAmountOfExternalAllocateMemory(change\_in\_bytes);  
}  
  
void init(v8::Local<v8::Object> exports,v8::Local<v8::Object> module){  
 NODE\_SET\_METHOD(module,"exports",Method);  
}  
  
NODE\_SET\_METHOD(binding,init);

上述代码直接导出了一个函数，这个函数接收一个Buffer对象，对于小于8kb的Buffer，它的内存来自Unit8Array的一个片段，因此不能简单释放。如果这个对象维护的堆内存大于8KB，就可以将内存释放掉，而这行代码：

isolate->AdjustAmountOfExternalAllocateMemory(change\_in\_bytes);  
}

用来告知V8堆外内存已经改变了。传入的参数为负数，代表堆外内存减少了相应值。这个函数内部判断了一下堆外内存是否超过一个固定值：

*//I::kExternalAllocationLimit is const as (192 \* 1024 \* 1024)*if(change\_inbytes > 0 && amount - \*amount\_of\_external\_allocated\_memory\_at\_last\_global\_gc > I::kExternalAllocationLimit){  
 ReportExternalAllocationLimitReached();  
}

可见，如果参数为正数，且堆外内存超过这个固定值，就会调用V8的内置函数Rep...

该函数的作用就是为增量标记算法的运行提供时机。

### 2.5 避免内存泄露

const MAXLEN = 2000;  
class Queue{  
 constructor(){  
 this.filelist = [];  
 this.top = 0;  
 }  
 Push(path){  
 this.filelist.push(path);  
 }  
 Pop(){  
 if(this.top < this.filelist.length){  
 if(this.top){  
 this.filelist = this.filelist.splice(this.top,this.filelist.length - this.top);  
 this.top = 0;  
 }  
 this.top += 1;  
 return this.filelist[this.top - 1];  
 } else {  
 return null  
 }  
 }  
 Lentgh(){  
 return (this.filelist.length - this.top);  
 }  
}  
  
var queue = new Queue();

假如上述代码调用Pop频率更高，则不会出现问题，如果Push频率高于Pop，那么队列就会不断膨胀，因此上述队列是不安全的，可以给类添加一个成员函数：

Shuff(){  
 if((this.filelist.length) > MAXLEN ){  
 this.filelist = this.filelist.splice(this.top,MAXLEN - 700);  
 this.top = 0;  
 }  
}

在调用Pop方法后，调用一次Shuff方法，如果发现队列超过一定大小，将一部分数据删除，除此之外，应该考虑借助Redis实现生产者消费者队列。

下面是利用Redis做队列的一个例子

let Redis = require('ioredis');  
let redis = new Redis({  
 port:6379,  
 host:'127.0.0.1'  
});  
const QUEUENAME = 'data\_mq';  
redis.push(QUEUENAME,'Electric cars will be popular');  
redis.lpop(QUEUENAME,function (err,data) {  
 console.log(data);  
});

ioredis支持集群模式，使用起来和单机模式没有太大区别，以下是一个连接集群的例子：

let Redis = require('ioredis');  
let redis\_cluster = new Redis.Cluster([{  
 port:6379,  
 host:'ip1'  
},{  
 port:6380,  
 host:'ip2'  
}],{  
 redisOptions: {  
 dropBufferSupport:true,  
 parser:'hiredis'  
 }  
});  
redis\_cluster.multi().set('foo','xbar').get('foo').exec(function (err,results) {  
 console.log(results);  
});

这里连接了一个Redis集群，并指定使用Hiredis（需要安装这个模块，Hriedis是一个用C语言实现的Redis协议解析器）。对于set，get这种简单的操作，使用ioredis这种JS解析器足够了，但是对于Irange或者ZRANGE这类可能返回巨量数据的操作，使用Hiredis效果更为显著。

ioredis为没一个命令提供了一个二进制版本，用以操作二进制数据，例如lop的二进制版本是lpopBuffer：

redis.lpopBuffer(QUEUENAME,function (err,data) {  
 console.log(data instanceof Buffer);  
});

dropBufferSupport选项设置为true，意味着ioredis将强制解析器返回字符串而不是Buffer对象。这个选项默认为false，在使用Hiredis时，应设置为true，以避免不必要的内存赋值，否则会影响GC的性能，如果要使用二进制版本的命令，可以再创建一个使用默认协议解析器的连接实例。

上述实例以事务的方式调用了set和get，在集群模式下，事务内部的操作只能在相同的key上进行。

### 2.6 闭包的内存泄露

JS的闭包机制使得被异步调用打断的逻辑，在等待异步完成的过程中，上下文环境仍然能够保留。异步调用完成之后，回调函数可以在它需要的上下文环境中继续执行。闭包的这个特点，使得它可以引用它之外的自由变量。一个函数执行完毕，其内部变量应该可以被回收，但是闭包的引用，使这个问题变得稍微复杂一些。如果闭包被引用，而这个闭包又在有效期内，则这些变量不会被回收。

function CreatePerson(name) {  
 let o = {  
 sayName:function () {  
 console.log(\_name);  
 }  
 };  
 let \_name = name;  
 return o;  
}  
let p = CreatePerson('zs');  
p.sayName();

构造函数创建的p对象是一个闭包，这个闭包引用了构造函数中的\_name变量，这个变量不会被释放，除非将p赋值为null。

let fn = function () {  
 let largeArr = new Array(1000);  
 return function () {  
 console.log('run once');  
 return largeArr;  
 };  
}();  
setTimeout(fn,2000);  
fn = null;

虽然fn被设置为null，但是对象不会被释放，回调的是fn原来的闭包。

Node运行中，只要满足以下条件中的任意一个，对象均不会被回收。

1 全局变量或者由全局变量触发，可以访问到的对象；

2 正执行函数中的局部对象，包括这些局部对象可以访问到的对象；

3 一个非全局对象，如果被一个闭包引用，则这个对象将和引用它的闭包异同存在，即使离开了创建它的环境。这个对象称为自由变量，它为未来闭包执行的时候保留上下文。