设计文档

1.确定所有的输入和输出

2.定义好所有输入和输出的边界和限制条件。

输入：

1.函数名：

限制：

1.1 函数名长度最大64字节

2.函数参数

限制：

2.1 所有函数参数总长度限制在1024字节

使用sizeof

输出：

返回结果

限制：

1.1返回部分总长度限制在4096

需要辅助类

array类：

1.m\*n的数组

需要定义太多辅助变量

elenum，eleSize（分init和max）elements，

2.总长度为q，m\*n《q，m和n任意组合。

struct：

len：当前数组的长度

element\_pointer\_len: 数组存储元素指针的长度

elements，

useSize：m\*指针大小+每个元素的大小之和（n1+n2+...ni)

maxSize: 设定的数组最大空间，useSize必须要小于maxSize

k： 初始化时先初始化k个元素，如果add超过k个，并且useSize小于maxSize，则再申请k个元素。每个指针64位机器位8字节，5个k元素就需要40个字节。

add（）: 要做两样事情，插入元素和重新分配元素指针空间（如果需要）

动态按需malloc单个元素，但每次需要relloc 元素个数elements 按照固定步增的方式 n\*k.

先判断useSize+插入的元素的长度是否超过maxSize，如果有则报错。

再判断就要重新realloc 元素指针空间，把分配的元素增加k个，

UseSize 增加

例子：len=4，k=5，当想插入第5个元素时，

如果len不等于0，并且（len+1）%k 等于0的时候，

要先判断 useSize+插入元素的长度+k\*指针大小 是否超过maxSize，如果有则报错，没有就插入元素，重新分配元素指针空间。

如果len%k 不等于0的时候，比如插入第6个元素时，这时只需判断

useSize+插入元素的长度是否超过maxSize，如果有则报错，没有就插入元素。

delete（）

例子：len=6，k=5，当想删除第6个元素时，

如果len等于0，报错。

如果（len-1）%k 等于0 并且 element\_pointer\_len-（len-1）=2k的时候，就删除元素，并且重新分配元素指针空间（缩减），把空间分配成element\_pointer\_len-k。

否则只是删除元素。

empty（）

init（）:

先malloc k个元素，指针本身也占空间啊，每个元素等具体add时再malloc。

destroy （）free ele[i]和ele

util类：

checkmemory（）

overflow（）

safeadd（）

常量定义：

输入：

1.函数名大小 64

2.参数部分大小，使用总大小为r的数组。

1024

输出：

1.返回参数部分大小，使用总大小为q的数组

4096

main类：

add（）

concat（）

因为把函数插入StringArray已经检查了是否有超出最大大小，因此分配的空间可以用

useSize- sizeof(char\*)\* element\_pointer\_len 计算的来。

先遍历计算出所有参数拼接起来需要的总空间，

如果不超过返回结果数组设置的最大大小，则给concat\_str分配内存空间，把参数真正拼接起来，返回字符串。

funcMap（）

print-result（）

编码过程：

1. 先编写成员属性
2. 再编写成员方法

先写下主要逻辑。

1. 然后定义全局常量和变量

对emcc自动产生的Test.js 关键修改点：

1642行

1576行

最核心的两个更改地方：

1645行：

return WebAssembly.instantiate(binary, info);

1656行：

WebAssembly.instantiateStreaming(fetch(wasmBinaryFile, { credentials: 'same-origin' }), info)

导入的外部对象

var info = {

'global': null,

'env': null,

'asm2wasm': asm2wasmImports,

'parent': Module // Module inside wasm-js.cpp refers to wasm-js.cpp; this allows access to the outside program.

};

添加：

//Add wasm meters by lmh --20181016

                'metering': {

                    'usegas': (gas) => {

                     gasUsed += gas

                     if (gasUsed > limit) {

                        throw new Error('out of gas!')

                     }

                    }

                }

真正的main方法执行在doRun方法里的 Module[‘callMain’](args)：

function doRun() {

if (Module['calledRun']) return; // run may have just been called while the async setStatus time below was happening

Module['calledRun'] = true;

if (ABORT) return;

ensureInitRuntime();

preMain();

if (Module['onRuntimeInitialized']) Module['onRuntimeInitialized']();

if (Module['\_main'] && shouldRunNow) Module['callMain'](args);

postRun();

}

Module[‘callMain’](args)方法会调用Module[‘\_main’](argc,argv,0)，实现对程序main方法的调用。

Module['callMain'] = function callMain(args) {

assert(runDependencies == 0, 'cannot call main when async dependencies remain! (listen on \_\_ATMAIN\_\_)');

assert(\_\_ATPRERUN\_\_.length == 0, 'cannot call main when preRun functions remain to be called');

args = args || [];

ensureInitRuntime();

var argc = args.length+1;

var argv = stackAlloc((argc + 1) \* 4);

HEAP32[argv >> 2] = allocateUTF8OnStack(Module['thisProgram']);

for (var i = 1; i < argc; i++) {

HEAP32[(argv >> 2) + i] = allocateUTF8OnStack(args[i - 1]);

}

HEAP32[(argv >> 2) + argc] = 0;

try {

var ret = Module['\_main'](argc, argv, 0);

// if we're not running an evented main loop, it's time to exit

exit(ret, /\* implicit = \*/ true);

}

catch(e) {

if (e instanceof ExitStatus) {

// exit() throws this once it's done to make sure execution

// has been stopped completely

return;

} else if (e == 'SimulateInfiniteLoop') {

// running an evented main loop, don't immediately exit

Module['noExitRuntime'] = true;

return;

} else {

var toLog = e;

if (e && typeof e === 'object' && e.stack) {

toLog = [e, e.stack];

}

err('exception thrown: ' + toLog);

Module['quit'](1, e);

}

} finally {

calledMain = true;

}

}