使用C语言实现Lox脚本语言

Table of Contents

1. 字节码块

> 如果你发现自己几乎把所有的时间都花在了理论上，那就开始把注意力转向实践；它会改进你的理论。如果你发现你几乎把所有的时间都花在了实践上，那就开始把注意力转向理论；它会改善你的实践。
>
> — 高德纳

1.1. 开始

让我们先编写一些基本的 代码 。先从 `main` 函数开始。

main.c, create new file

```c
#include "common.h"

int main(int argc, const char* argv[]) { ①
  return 0;
}
```

① `const char*` 表示一个可变指针指向了不可变的字符/字符串。

我们会把常用的一些类型和常量放置在 `common.h` 中。

common.h, create new file

```c
#ifndef clox_common_h ①
#define clox_common_h

#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

#endif
```

① `#ifndef clox_common_h` 表示如果没有定义过 `clox_common_h` ，则定义之。如果定义过，则不执行以上代码片段。

1.2. 指令块

块（chunk）表示字节码序列。

chunk.h, create new file

```
#ifndef clox_chunk_h
#define clox_chunk_h

#include "common.h"

#endif
```

在字节码格式中，每条指令都对应一个单字节的操作码（opcode）。所以才叫字节码。我们先来编写一条最简单的字节码指令 `OP_RETURN` 。这条指令表示“从当前函数返回”。不过现在还不具备这个功能。

chunk.h

```
#include "common.h"

typedef enum {
  OP_RETURN,
} OpCode;

#endif
```

1.2.1. 指令的动态数组

字节码是一系列指令。我们会存储指令和一些其它数据，所以让我们创建一个结构体来保存数据。

chunk.h, add after enum OpCode

```
} OpCode;

typedef struct {
  uint8_t* code; ①
} Chunk;

#endif
```

① `code` 是指向字节数组的开头位置的指针。

由于我们不知道字节数组的具体大小，所以需要使用动态数组。动态数组有以下特点：

- 对缓存友好，因为是紧挨着存储的。
- 通过数组索引查找元素是常数时间复杂度。
- 在数组末尾追加元素是常数时间复杂度。

动态数组其实就是Java中的 `ArrayList` 数据类型。在C语言中需要我们自己来实现。

chunk.h, in struct Chunk

```
typedef struct {
  int count;    ①
  int capacity; ②
  uint8_t* code;
} Chunk;
```

① 数组中已经使用的数量

② 数组的容量（大小）

创建一个实例化 `Chunk` 的接口：

chunk.h, add after struct Chunk

```
} Chunk;

void initChunk(Chunk* chunk);

#endif
```

然后实现接口：

chunk.c, create new file

```
#include <stdlib.h>

#include "chunk.h"

void initChunk(Chunk* chunk) {
  chunk->count = 0;
  chunk->capacity = 0;
  chunk->code = NULL;
}
```

动态数组的初始状态是空数组。我们还没有分配一个数组出来。为了可以将一个字节追加到块的末尾，我们需要一个新的接口。

chunk.h, add after initChunk()

```
void initChunk(Chunk* chunk);
void writeChunk(Chunk* chunk, uint8_t byte);

#endif
```

然后我们实现 `writeChunk` 接口。首先检查数组容量是否够用，如果不够用需要扩展动态数组的大小，然后再将字节码添加到数组末尾。

chunk.c, add after initChunk()

```
void writeChunk(Chunk* chunk, uint8_t byte) {
  if (chunk->capacity < chunk->count + 1) {
    int oldCapacity = chunk->capacity;
    chunk->capacity = GROW_CAPACITY(oldCapacity);
    chunk->code = GROW_ARRAY(uint8_t, chunk->code,
        oldCapacity, chunk->capacity);
  }

  chunk->code[chunk->count] = byte;
  chunk->count++;
}
```

以上代码中的宏定义我们定义在 memory.h 头文件中。我们先来引入这个头文件。

chunk.c

```
#include "chunk.h"
#include "memory.h"

void initChunk(Chunk* chunk) {
```

在头文件中定义所需要的宏。

memory.h, create new file

```
#ifndef clox_memory_h
#define clox_memory_h

#include "common.h"

#define GROW_CAPACITY(capacity) \
    ((capacity) < 8 ? 8 : (capacity) * 2)   ①

#endif
```

① 宏定义用来扩展数组的容量，如果数组容量小于8，那么扩展为8个元素的容量。如果大于等于8，则扩展为原来容量的2倍。

memory.h

```
#define GROW_CAPACITY(capacity) \
    ((capacity) < 8 ? 8 : (capacity) * 2)

#define GROW_ARRAY(type, pointer, oldCount, newCount) \
    (type*)reallocate(pointer, sizeof(type) * (oldCount), \
        sizeof(type) * (newCount))

void* reallocate(void* pointer, size_t oldSize, size_t newSize);
```

```
#endif
```

memory.c, create new file

```
#include <stdlib.h>

#include "memory.h"

void* reallocate(void* pointer, size_t oldSize, size_t newSize) { ①
  if (newSize == 0) { ②
    free(pointer);
    return NULL;
  }

  void* result = realloc(pointer, newSize); ③
  return result;
}
```

① void* 表示可以指向任意类型的指针，类似Java中的Object。

② 如果newSize为0，则释放pointer指向的内存块。

③ realloc会扩展之前pointer指向的内存块，扩展后的大小为newSize，并且之前内存中的内容都还在。

memory.c, in reallocate()

```
  void* result = realloc(pointer, newSize);
  if (result == NULL) exit(1); ①
  return result;
```

① 如果扩展数组失败，则报错退出。

chunk.h, add after initChunk()

```
void initChunk(Chunk* chunk);
void freeChunk(Chunk* chunk); ①
void writeChunk(Chunk* chunk, uint8_t byte);
```

① 释放块数组的接口。

chunk.c, add after initChunk()

```
void freeChunk(Chunk* chunk) {
  FREE_ARRAY(uint8_t, chunk->code, chunk->capacity); ①
  initChunk(chunk); ②
}
```

① 释放块数组

② 重新初始化一个空的块

memory.h

```
#define GROW_ARRAY(type, pointer, oldCount, newCount) \
    (type*)reallocate(pointer, sizeof(type) * (oldCount), \
        sizeof(type) * (newCount))

#define FREE_ARRAY(type, pointer, oldCount) \
    reallocate(pointer, sizeof(type) * (oldCount), 0) ①

void* reallocate(void* pointer, size_t oldSize, size_t newSize);
```

① 传入参数0，释放pointer指向的内存块。

1.3. 对字节码块进行反汇编

main.c, in main()

```
int main(int argc, const char* argv[]) {
  Chunk chunk;
  initChunk(&chunk);              ①
  writeChunk(&chunk, OP_RETURN);  ②
  freeChunk(&chunk);              ③
  return 0;
```

① 初始化空块

② 在块中追加一条指令 OP_RETURN

③ 释放块并重新初始化一个空块

引入必要的头文件。

main.c

```
#include "common.h"
#include "chunk.h"

int main(int argc, const char* argv[]) {
```

我们在块中添加了一条 OP_RETURN 指令以后，将块传递给反汇编函数。

main.c, in main()

```
  initChunk(&chunk);
  writeChunk(&chunk, OP_RETURN);

  disassembleChunk(&chunk, "test chunk"); ①
  freeChunk(&chunk);
```

① 对块 chunk 反汇编

由于反汇编的功能主要用于debug，也就是说如果没有反汇编功能，也不影响虚拟机的执行。但对虚拟机代码

的编写至关重要，因为方便我们的调试。所以我们把反汇编的功能都放在debug模块中。

main.c

```
#include "chunk.h"
#include "debug.h" ①

int main(int argc, const char* argv[]) {
```

① 引入头文件

在下面的代码中定义反汇编的接口。

debug.h, create new file

```
#ifndef clox_debug_h
#define clox_debug_h

#include "chunk.h"

void disassembleChunk(Chunk* chunk, const char* name);
int disassembleInstruction(Chunk* chunk, int offset);

#endif
```

然后实现接口。

debug.c, create new file

```
#include <stdio.h>

#include "debug.h"

void disassembleChunk(Chunk* chunk, const char* name) {
  printf("== %s ==\n", name);

  for (int offset = 0; offset < chunk->count;) {
    offset = disassembleInstruction(chunk, offset);
  }
}
```

debug.c, add after disassembleChunk()

```
int disassembleInstruction(Chunk* chunk, int offset) {
  printf("%04d ", offset);

  uint8_t instruction = chunk->code[offset];
  switch (instruction) {
    case OP_RETURN:
      return simpleInstruction("OP_RETURN", offset);
    default:
```

```
      printf("Unknown opcode %d\n", instruction);
      return offset + 1;
  }
}
```

debug.c, add after disassembleChunk()

```
static int simpleInstruction(const char* name, int offset) {
  printf("%s\n", name);
  return offset + 1;
}
```

```
== test chunk ==
0000 OP_RETURN
```