zhuanlan.zhihu.com

通过Handle理解V8的代码设计（基于V0.1.5）

21-27 minutes

前言：Handle在V8里是一个非常重要的概念，本文从早期的源码分析Handle的原理，在分析的过程中我们还可以看到V8在代码设计上的一些细节。

假设我们有以下代码

```
HandleScope scope;
Local<String> hello = String::New(参数);
```

这个看起来很简单的过程，其实在V8的内部实现起来比较复杂。

HandleScope

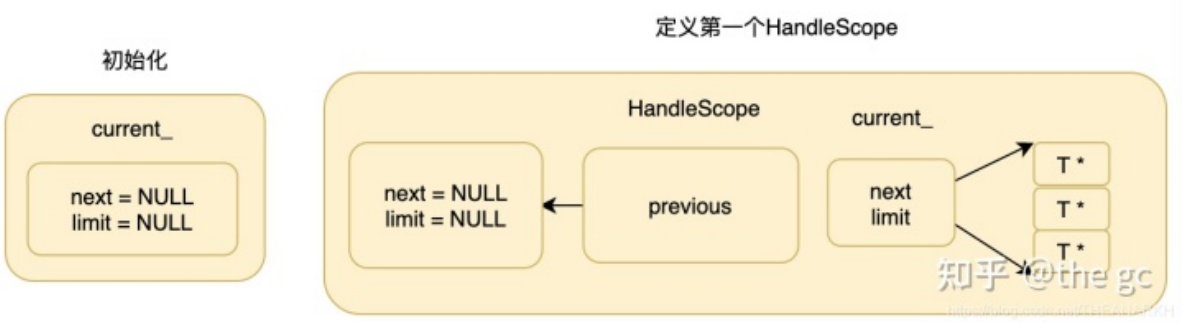
我们从创建一个HandleScope对象开始分析。
HandleScope是负责管理多个Handle的对象，主要是为了方便管理Handle的分配和释放。

```
class HandleScope {
 public:
```

```
  HandleScope() : previous_(current_),
is_closed_(false) {
    current_.extensions = 0;
  }

  static void** CreateHandle(void* value);

 private:

  class Data {
   public:
    // 分配了一块内存后，又额外分配的块数
    int extensions;
    // 下一个可用的位置
    void** next;
    // 达到limit执行的地址后说明当前内存块用完了
    void** limit;
    inline void Initialize() {
      extensions = -1;
      next = limit = NULL;
    }
  };
  // 当前的HandleScope
  static Data current_;
  // 上一个HandleScope
```

```
  const Data previous_;
};


HandleScope::Data HandleScope::current_ = { -1,
NULL, NULL };
```

通过HandleScope的构造函数我们知道每次定义一个HandleScope对象的时候，previous就会指向前一个HandleScope的数据（但是current_除了第一次创建HandleScope的时候更新了（见CreateHandle），后续似乎没有更新？后续详细看一下），从HandleScope的定义中我们知道他的布局如下。



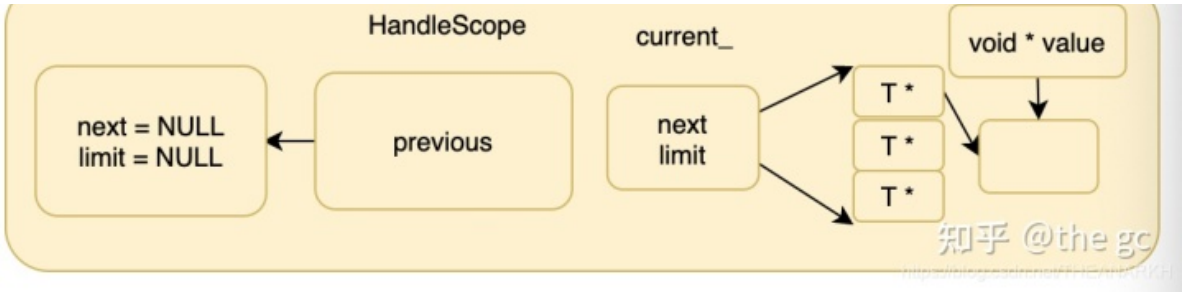
接着我们看HandleScope的CreateHandle方法。

```
void** v8::HandleScope::CreateHandle(void* value)
{
  // 获取下一个可用的地址
  void** result = current_.next;
  // 到达limit的地址了或者为空（初始化的时候）则
获取新的内存
  if (result == current_.limit) {
```

```
    // Block是二维数组，每个元素指向一个可以存储数据的数组。非空说明可能有可用的内存空间
    if (!thread_local.Blocks()->is_empty()) {
      // 拿到list中最后一个元素，得到一个数组首地址，然后再获取他的limit地址，即末地址
      void** limit = &thread_local.Blocks()->last()
[i::kHandleBlockSize];
      if (current_.limit != limit) {
        current_.limit = limit;
        // v8里少了这一句，看起来是需要修改result的值的
        // result = limit - i::kHandleBlockSize;
      }
    }
  }
  // 下一个可用的地址
  current_.next = result + 1;
  *result = value;
  return result;
}
```

我们看到CreateHandle会首先获取一片内存，然后把入参value的值保存到该内存中。

定义第一个HandleScope



String::New

了解了HandleScope后，我们继续分析String::New。

```
Local<String> v8::String::New(const char* data,
int length) {
  i::Handle<i::String> result =

i::Factory::NewStringFromUtf8(i::Vector<const
char>(data, length));
  return Utils::ToLocal(result);
}
```

我们接着看NewStringFromUtf8。

```
Handle<String>
Factory::NewStringFromUtf8(Vector<const char>
string,PretenureFlag pretenure) {

CALL_HEAP_FUNCTION(Heap::AllocateStringFromUtf8(s
                   pretenure),
                   String);
}
```

我们先看一下AllocateStringFromUtf8的实现，然后再看CALL_HEAP_FUNCTION。

```
Object* Heap::AllocateStringFromUtf8(Vector<const
char> string,PretenureFlag pretenure) {
  return AllocateStringFromAscii(string,
pretenure);
}

Object*
Heap::AllocateStringFromAscii(Vector<const char>
string,PretenureFlag pretenure) {
  // 从堆中分配一块内存
  Object* result =
AllocateRawAsciiString(string.length(),
pretenure);
  // 设置堆对象的内容
  AsciiString* string_result =
AsciiString::cast(result);
  for (int i = 0; i < string.length(); i++) {
    string_result->AsciiStringSet(i, string[i]);
  }
  return result;
}
```

我们看到AllocateStringFromUtf8最后返回了一个堆内存

地址。接着我们看下CALL_HEAP_FUNCTION这个宏。

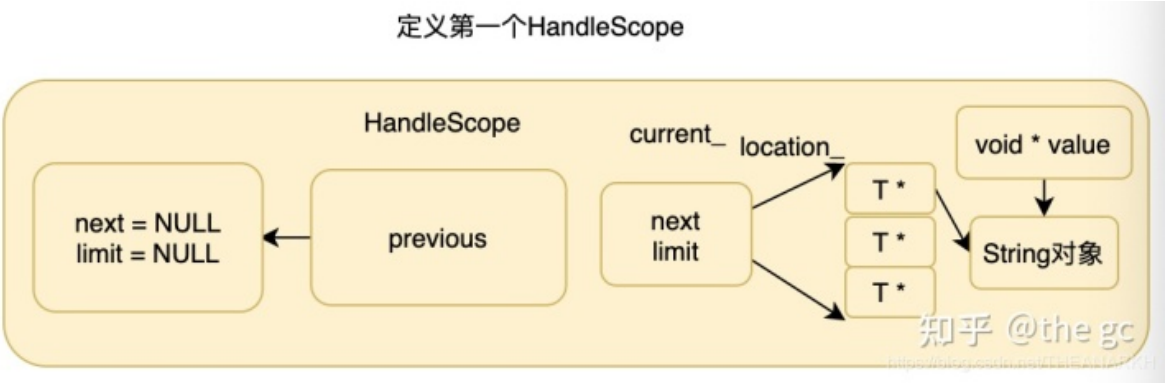
```
#define CALL_HEAP_FUNCTION(FUNCTION_CALL, TYPE)
  do {
    Object* __object__ = FUNCTION_CALL;
    return Handle<TYPE>(TYPE::cast(__object__));
  } while (false)
```

CALL_HEAP_FUNCTION的作用是把函数FUNCTION_CALL执行的结果转成Handle对象。我们知道FUNCTION_CALL函数返回的结果是一个堆内存指针。接下来我们看看是如何转成Handle的。这个Handle不是我们在代码里使用的Handle。而是V8内部使用的Handle（代码在handles.h），我们看看实现。

```
template<class T>
class Handle {
 public:
  explicit Handle(T* obj);
 private:
  T** location_;
};


template<class T>
Handle<T>::Handle(T* obj) {
  location_ = reinterpret_cast<T**>
(HandleScope::CreateHandle(obj));
```

```
}
```

我们看到Handle内部使用的是T**二级指针，而我们刚才拿到堆内存地址是一级指针，自然不能直接赋值，而是通过CreateHandle又处理了一下。
HandleScope::CreateHandle我们刚才已经分析过了。
执行CreateHandle后布局如下。



所以NewStringFromUtf8最后返回了一个Handle对象（里面维护了一个二级指针location_），接着V8调用Utils::ToLocal把他转成外部使用的Handle。接着赋值给Handle hello。这里的Handle是外部使用的Handle。

```
Local<v8::String>
Utils::ToLocal(v8::internal::Handle<v8::internal:
obj) {
    return Local<String>
(reinterpret_cast<String*>(obj.location()));
    }
```

首先通过obj.location()拿到一个二级指针。然后转成一个String *指针。接着构造一个Local对象。ToLocal是V8

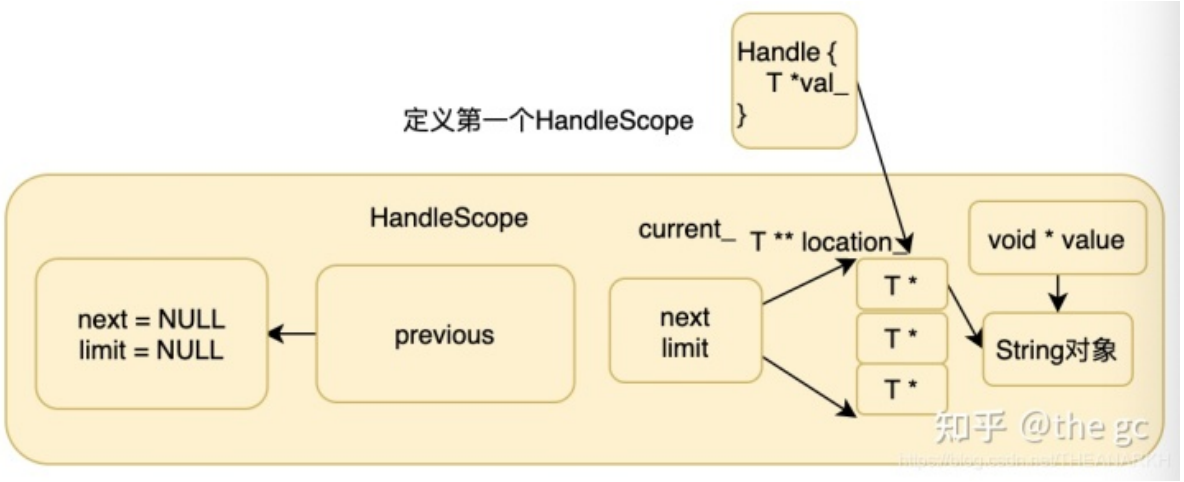
代码的分水岭，我们看看Local的定义。

```
template <class T> class Local : public Handle<T>
{
 public:
  template <class S> inline Local(S* that) :
Handle<T>(that) { }
};
```

直接调用Handle类的函数

```
template <class T> class Handle {
  explicit Handle(T* val) : val_(val) { }
  private:
    T* val_;
}
```

这时候的结构图如下



所以最后通过ToLocal返回一个外部Handle对象给用户。当执行

```
Local <String> xxx = Local对象
```

时就会调用Local的拷贝函数。

```
template <class S>
      inline Local(Local<S> that)
      // *that即取得他底层对象的地址
      : Handle<T>(reinterpret_cast<T*>(*that)) {}
```

我们首先看一下*that。Handle类重载了*运算符。

```
template <class T>
T* Handle<T>::operator*() {
  return val_;
}
```

所以reinterpret_cast(*that)拿到了Handle底层指针的值并转成String 类型。接着执行

```
explicit Handle(T* val) : val_(val) { }
```

整个过程下来，其实就是把被复制对象的底层指针复制过来。=

当我们使用Handle hello这个对象的方法时是怎样的，比如hello->Length()。Handle重载了->运算符。

```
template <class T>
T* Handle<T>::operator->() {
  return val_;
}
```

我们看到执行hello->Length()的时候首先会拿到一个String *。然后调用Length方法。其实就是调用String对象（在v8.h中定义）的Length方法。我们看看Length方法的实现。

```
int String::Length() {
  return Utils::OpenHandle(this)->length();
}
```

首先通过传入this调用OpenHandle拿到内部Handle。从前面的架构图中我们知道this（即val_和location_指向的值）本质上是一个String **，即二级指针。

```
v8: :internal: :Handle < v8: :internal: :String >
Utils: :OpenHandle(v8: :String * that) {
    return v8: :internal: :Handle < v8:
:internal: :String > (reinterpret_cast < v8:
:internal: :String * *>(that));
}
```

OpenHandle就是首先把外部的表示转成一个二级指针。然后再构造一个内部Handle。在内部Handle里保存了这个二级指针。接着访问这个Handle对象的length方法。而Handle重载了->运算符。

```
INLINE(T* operator ->() const)  { return
operator*(); }
```

```
template <class T>
inline T* Handle<T>::operator*() const {
  return *location_;
}
```

我们看到->的操作最终会被解引用一次变成String *，然后访问函数length，也就是访问String对象的length函数。

后记：从上面的分析中我们不仅看到了Handle的实现原理，也看到了V8代码的一些设计细节，V8在内部实现了一类对象，然后把内部对象转成外部使用的类型后返回给用户，当用户使用该返回的对象时，V8又会转成内部的对象再操作这个对象。核心的数据结构是两个Handle族的类。因为他们是维护了真实对象的句柄。其他的一些类，比如String，同样分为外部和内部类，内部类是实现了String的细节，而外部类只是一个壳子，他负责给用户暴露API，而不负责实现细节，但用户操作这些类时，V8会会转成内部类再进行操作。外部类的定义在v8.h中，这是我们使用V8时需要了解的最好文档。内部类的实现根据版本不同而不同，比如早期版本都是在object.h里实现的，而实现内外部对象转换的方法在api.c中定义。