Extension Framework 简介

钟志威

2018 年 7 月 28 日

1 Extension Framework 使用到的 v8 API

FunctionTemplate 类是 Function 类的模板类，可以理解为设置 Function 的公共特性，通过 FunctionTemplate 类 new 出来的 Function 类就拥有这些特性。相当于页面的 function。

```
/**
@brief new一个FunctionTemplate对象

@param[in] isolate表示一个独立的v8引擎实例，每个实例维护不同的状态
@param[in] callback是回调函数，创建实例或方法被调用时会调用
@param[in] data表示给回调函数传递的额外的参数
@return 返回FunctionTemplate对象
*/
static Local<FunctionTemplate> New(
    Isolate* isolate, FunctionCallback callback = 0,
    Local<Value> data = Local<Value>(),
    Local<Signature> signature = Local<Signature>(), int length = 0,
    ConstructorBehavior behavior = ConstructorBehavior::kAllow);

/**
 * Set the call-handler callback for a FunctionTemplate.  This
 * callback is called whenever the function created from this
 * FunctionTemplate is called.
 */
void SetCallHandler(FunctionCallback callback,
    Local<Value> data = Local<Value>());
```

ObjectTemplate 类是 Object 类的模板类，可以理解为设置 Object 的公共特性，通过 ObjectTemplate 类 new 出来的 Object 类就拥有这些特性。相当于页面的 object。

```
/**
@brief new一个ObjectTemplate对象

@param[in] isolate表示一个独立的v8引擎实例，每个实例维护不同的状态
@param[in] constructor是默认构造函数，只用于创建实例时会调用
@return 返回ObjectTemplate对象
*/
static Local<ObjectTemplate> New(
    Isolate* isolate,
    Local<FunctionTemplate> constructor = Local<FunctionTemplate>())
   ;

/**
@brief new一个Object对象实例

@param[in] context表示JavaScript代码运行环境上下文
@return 返回Object对象
*/
V8_WARN_UNUSED_RESULT MaybeLocal<Object> NewInstance(Local<Context>
   context);

/**
```

```
@brief 能够指定JavaScript访问对象属性时的一个callback

@param[in] getter表示获取属性时会被调用的callback
@param[in] setter表示设置属性时会被调用的callback
*/
void SetNamedPropertyHandler(NamedPropertyGetterCallback getter,
    NamedPropertySetterCallback setter = 0,
    NamedPropertyQueryCallback query = 0,
    NamedPropertyDeleterCallback deleter = 0,
    NamedPropertyEnumeratorCallback enumerator = 0,
    Local<Value> data = Local<Value>());

/**
@brief 通过这个模板生成的Object对象可以设置的内部field的数量

@param[in] value表示设置内部field的数量
*/
void SetInternalFieldCount(int value);
```

Object 类是一个实例对象

```
/**
@brief 设置Object的内部field

@param[in] index表示Object的内部field的索引值，必须小于
   SetInternalFieldCount函数传入的值
@param[in] value表示Object的内部field值
*/
void SetInternalField(int index, Local<Value> value);
```

其他

```
/**
 * A map that uses Global as value and std::map as the backing
 * implementation. Globals are held non-weak.
 *
 * C++11 embedders don't need this class, as they can use
 * Global directly in std containers.
 */
template <typename K, typename V,
          typename Traits = DefaultGlobalMapTraits<K, V> >
class StdGlobalValueMap : public GlobalValueMap<K, V, Traits> {
 public:
  explicit StdGlobalValueMap(Isolate* isolate)
      : GlobalValueMap<K, V, Traits>(isolate) {}
};
```

2 extension framework 架构分析

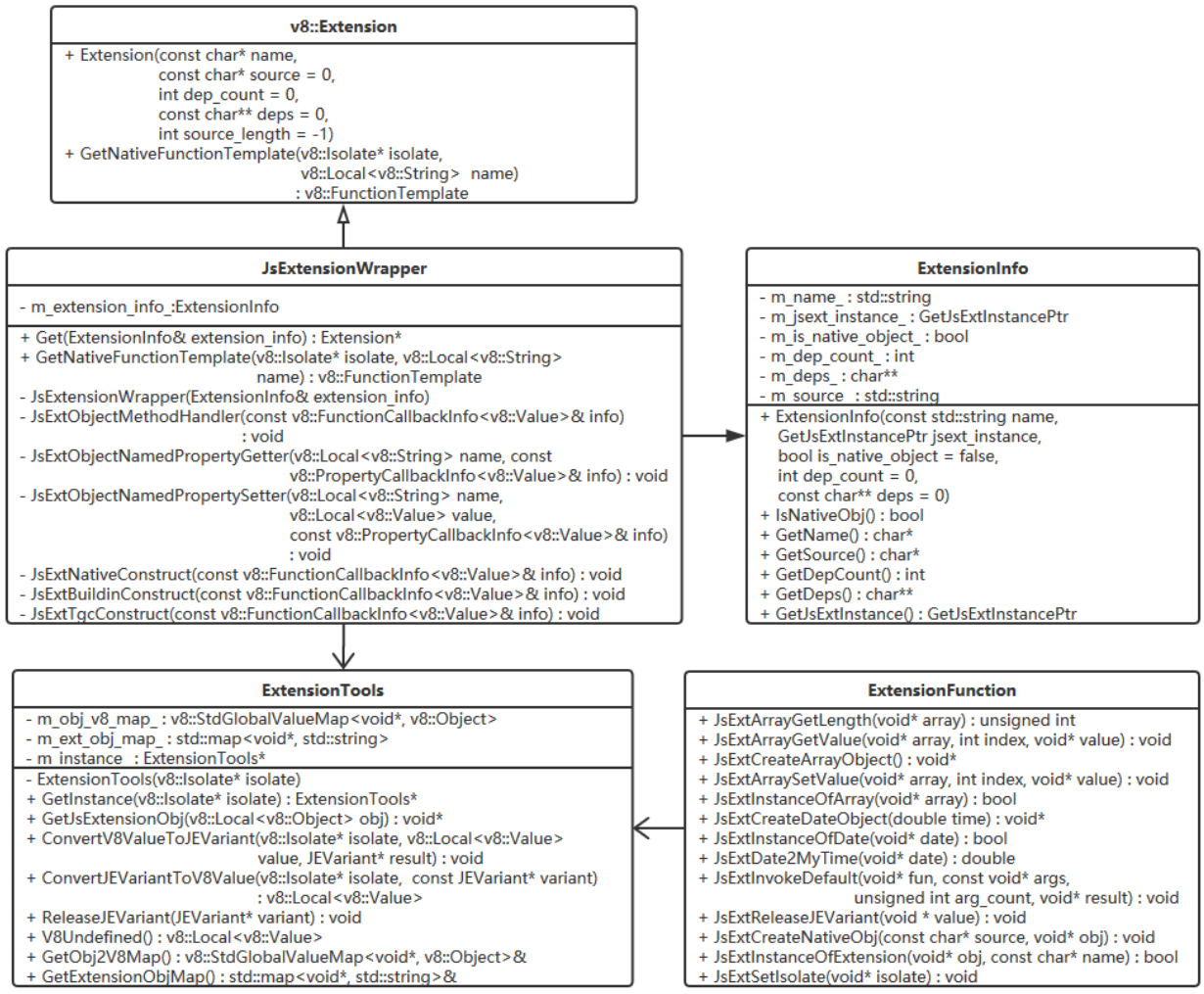

图 2.1: extension framwork 静态类图

如第图 2.1所示: v8 扩展机制主要是继承 v8 内置的 Extension 类，通过重写其 GetNativeFunctionTemplate 方法来获取 FunctionTemplate 对象；JsExtensionWrapper 类主要就是封装创建出来的 FunctionTemplate 对象的一些特性，比如构造函数，属性拦截器，方法执行；ExtensionInfo 类主要记录了创建 v8 扩展所有需要的关键信息；ExtensionTools 类主要是提供简易的函数，比如通过 v8::Object 类获取 ExtensionBase 对象，将 v8 变量转换为扩展变量，将扩展变量转换为 v8 变量，释放扩展变量等；ExtensionFunction 主要提供常用对外接口给 js 扩展调用。

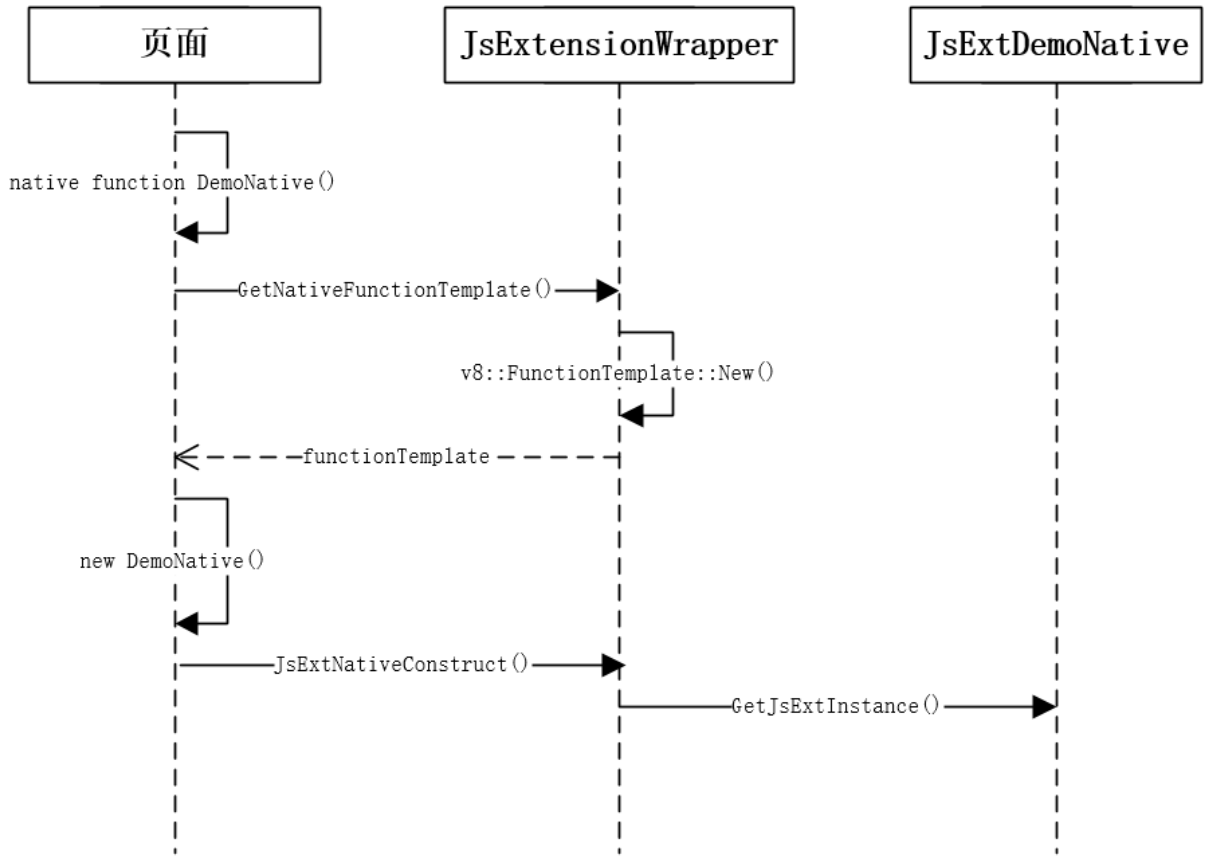

图 2.2: js 扩展对象创建时序图

js 扩展对象创建时序图如图 2.2所示：

- 在页面加载时，v8::extension 会注入页面代码 native function DemoNative()，此时 v8 就会去创建 DemoNative 对应 functionTemplate 模板类。
- 当 web 开发人员在页面使用 new DemoNative() 时，functionTemplate 模板类就会返回一个 v8::Object 实例对象给页面。
- 同理，针对内置对象，v8::extension 会注入页面代码 native function DemoBuildin() 和 var DemoBuildin = DemoBuildin() 两段代码，此时由于 DemoBuildin 变量覆盖了 DemoBuildin 方法名，所以只能通过 DemoBuildin 执行对应方法和函数，不能再被 new。

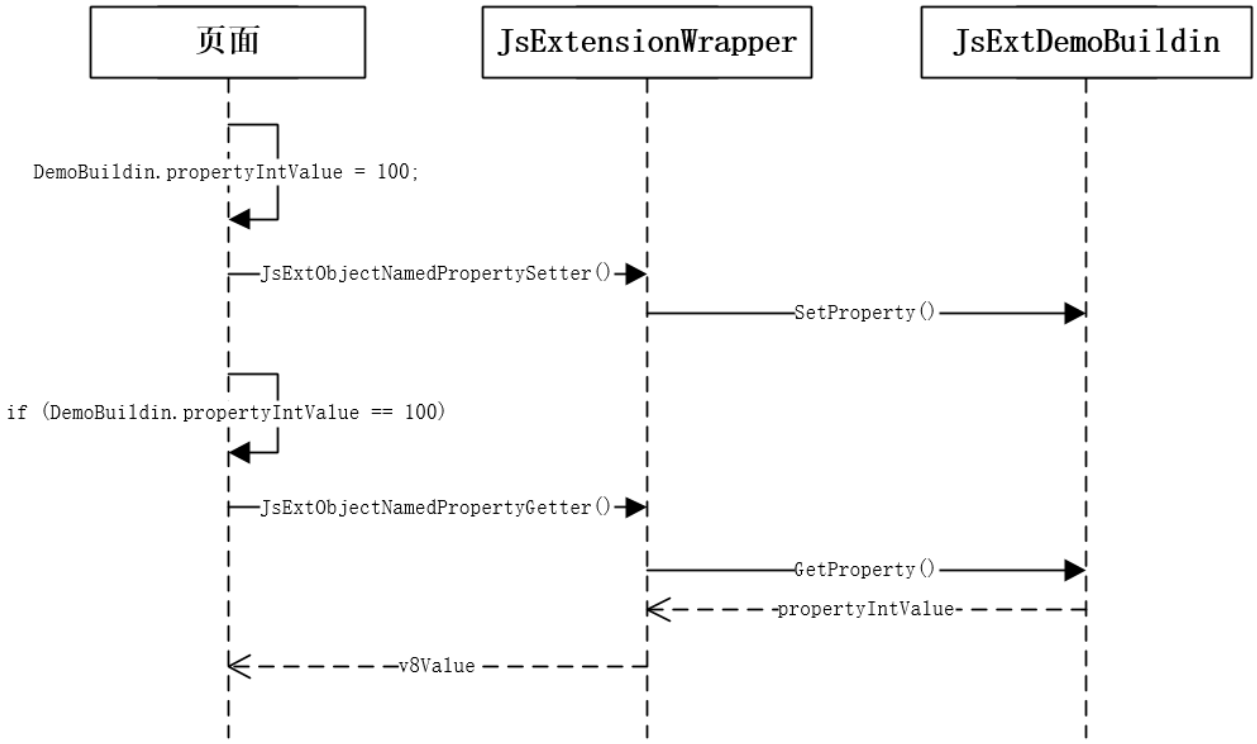

图 2.3: js 扩展对象属性访问时序图

js 扩展对象属性访问时序图如图 2.3所示：

- 由于在创建 DemoBuildin 对象时，对其 ObjectTemplate 模板类设置了拦截器特性，所以任何属性访问都会调用到拦截器的回调方法里。

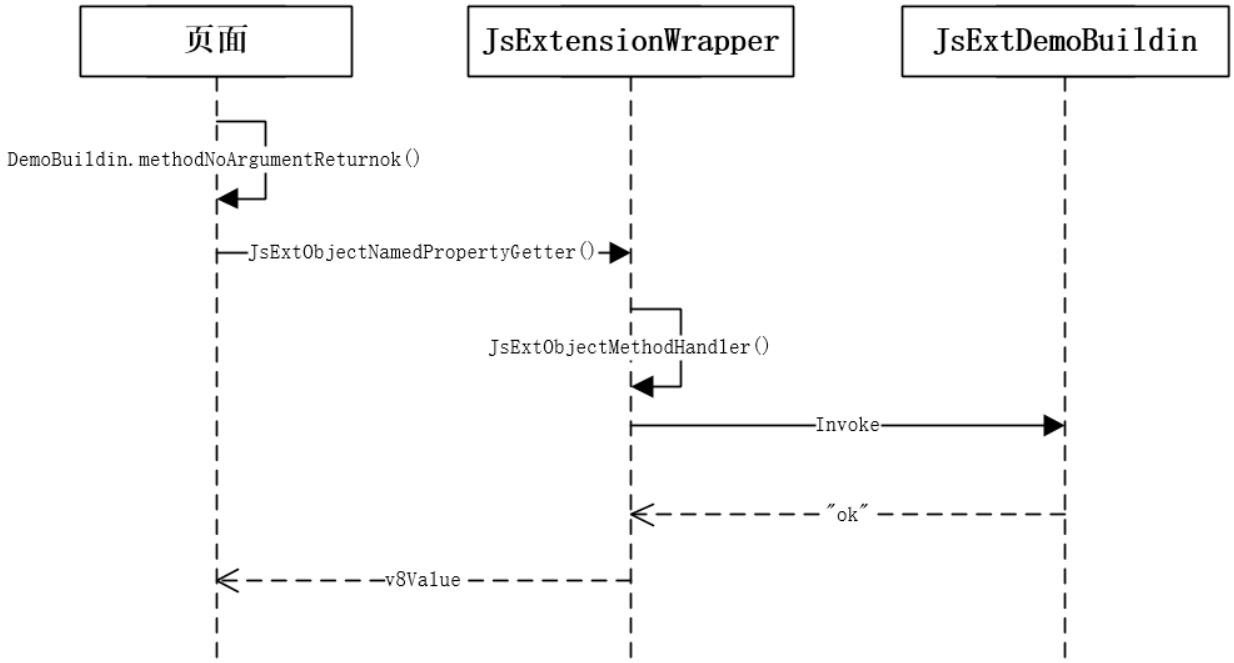

图 2.4: js 扩展对象方法访问时序图

js 扩展对象方法访问时序图如图 2.4所示：

- 同获取属性流程一致，当在拦截器回调函数里查询，如果属性没找到调用名，就去方法里找。

3 Extension Framework 垃圾回收机制

对于内置对象，每次跳转页面，都会去调用其构造函数，此时回收机制的做法是先 delete 上个页面 new 出来的 js 扩展实例对象，再重新 new 一个 js 扩展实例对象，然后绑定到 v8 Object 对象里。

```
void* delete_obj = ExtensionTools::GetJsExtensionObj(v8_obj);
  if (delete_obj) {
    ExtensionBase* ext_base = static_cast<ExtensionBase*>(delete_obj
    );
    delete ext_base;
    ext_base = NULL;
  }

  void* new_obj = NULL;
  extension_info->GetJsExtInstance()(NULL, 0, &new_obj);
  v8_obj->SetInternalField(ExtensionTools::JsExtBaseInternalField,
   v8::External::New(isolate, new_obj));
```

对于本地对象，每次创建对象时，我们会通过一个 std::map 保存 js 扩展实例对象指针以及 name，当跳转页面，都会去调用 tgc 内置对象其构造函数，将这个 map 里 js 扩展实例对象指针全部 delete 掉。

```
//在构造函数中将js扩展实例对象指针以及name保存到一个map中
void JsExtensionWrapper::JsExtNativeConstruct(const
    FunctionCallbackInfo<v8::Value>& info) {
  ...
  ExtensionTools* extension_tools = ExtensionTools::GetInstance(
    isolate);
  extension_tools->GetExtensionObjMap().insert(std::pair<void*, std
    ::string>(new_obj, extension_info->GetName()));
  ...
}

//在tgc内置对象构造函数中将map中的js扩展实例对象指针全部delete掉
void JsExtensionWrapper::JsExtTgcConstruct(const
    FunctionCallbackInfo<v8::Value>& info) {
  ...
  ExtensionTools* extension_tools = ExtensionTools::GetInstance(
    isolate);
  std::map<void*, std::string>::iterator it;
  for (it = extension_tools->GetExtensionObjMap().begin(); it !=
    extension_tools->GetExtensionObjMap().end(); it++) {
     ExtensionBase* ext_base = static_cast<ExtensionBase*>(it->first
    );
     if (ext_base) {
       delete ext_base;
       ext_base = NULL;
     }
  }
  ...
}
```