PPAPI 插件概要设计书

钟志威

2018 年 8 月 30 日

文档历史发放及记录

序号	变更（+/-说明）	作者	版本号	日期	批准
1	创建	钟志威	V0.1	2018/7/27	
2	修改 (+PPAPI 历史、注册、生命周期)	钟志威	V0.2	2018/8/2	

目 录

1 引言

1.1 插件历史

插件一直是浏览器的重要组成部分，实现一些 HTML+JS 实现不了本地应用（比如音视频、文件操作等）。

早期 IE 发布的插件 ActiveX，它基于 COM 规范，在 IE 占浏览器市场主流份额的时代，银行开发网页安全控件都是使用 ActiveX 开发。虽然 ActiveX 出尽了风头，但它并不是浏览器业内的插件标准，由网景发布的 NPAPI 是除了 IE 之外的其他浏览器共同支持的业内标准。

现在，随着 IE 逐渐退出历史舞台，ActiveX 插件的运行范围越来越窄。而 NPAPI 插件定义的 API 接口太旧且易用性差，崩溃和安全问题也很多，无法适应现代浏览器的发展。虽然很多银行已经支持 NPAPI 版本的安全控件，但 NPAPI 却做为不安全因素被主流浏览器（Chrome、FireFox）抛弃。

Google 从 2012 年，基于 NPAPI 自主研发了一套 API 接口，称为 PPAPI 接口，全称是 Pepper Plugin API。相比于 NPAPI，PPAPI 接口封装得更完善，开发更方便，且运行在 Chrome 的沙箱模型里，安全性得到保证。从 2014 年起，Chrome 逐渐放弃对 NPAPI 插件的支持，因此本文接下来要分析的是 PPAPI 插件。

1.2 开发背景

目前海外产品需要支持 hbbtv 和 netrange 两业务，而支持这两业务必须实现其插件。以前 tbrowser1.0 和 tbrowser2.0 分别使用 NPAPI 去实现插件，而最新的 tbrowser2.61 已经不再支持 NPAPI，使用的则是 PPAPI。本文旨在介绍 PPAPI 的加载流程和使用方法，供开发人员参考。

1.3 软件名称

PPAPI 插件 —— 由 Google 自主研发的浏览器插件系统

1.4 术语和缩略语

缩略语/术语	全　称	说　明
PPAPI	Pepper Plugin Application Programming Interface	

1.5 参考资料

https://blog.csdn.net/luoshengyang/article/details/48417255
https://blog.csdn.net/wbsong1978/article/details/77658912
https://code.google.com/p/ppapi/wiki/InterfacingWithJavaScript

2 总体设计

2.1 需求规定

1. 对 HTML 的 embed 和 object 标签插件的支持
2. 支持创建插件并由页面调用到其自定义的属性和方法

2.2 运行环境

Linux

2.3 上下文定义

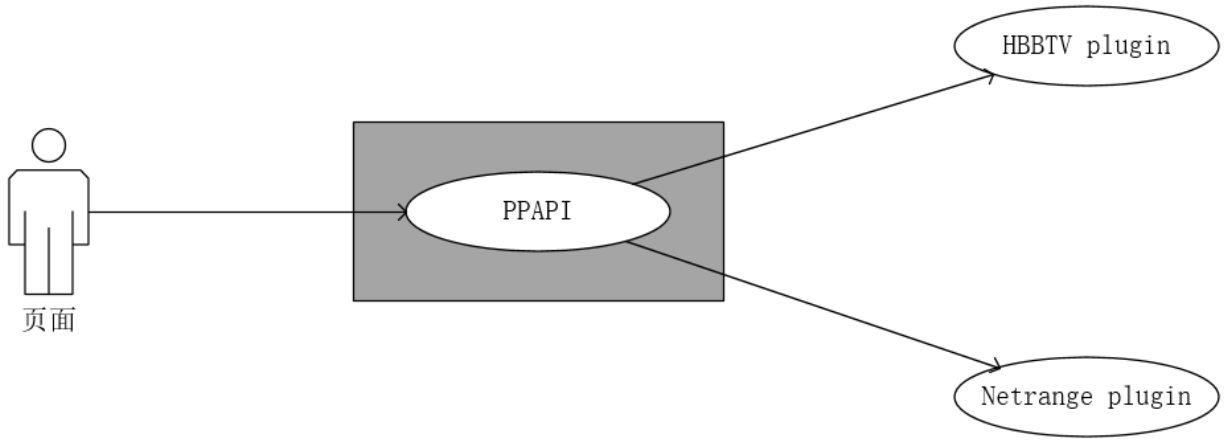

图 2.1: PPAPI 插件上下文

功能模块	说明
页面	使用 object 或者 embed 标签的 HTML 页面
PPAPI	Google 自主研发的插件系统
Netrange Plugin	Netrange 插件实现
HBBTV Plugin	HBBTV 插件实现

2.4 PPAPI 插件进程与其他进程关系图

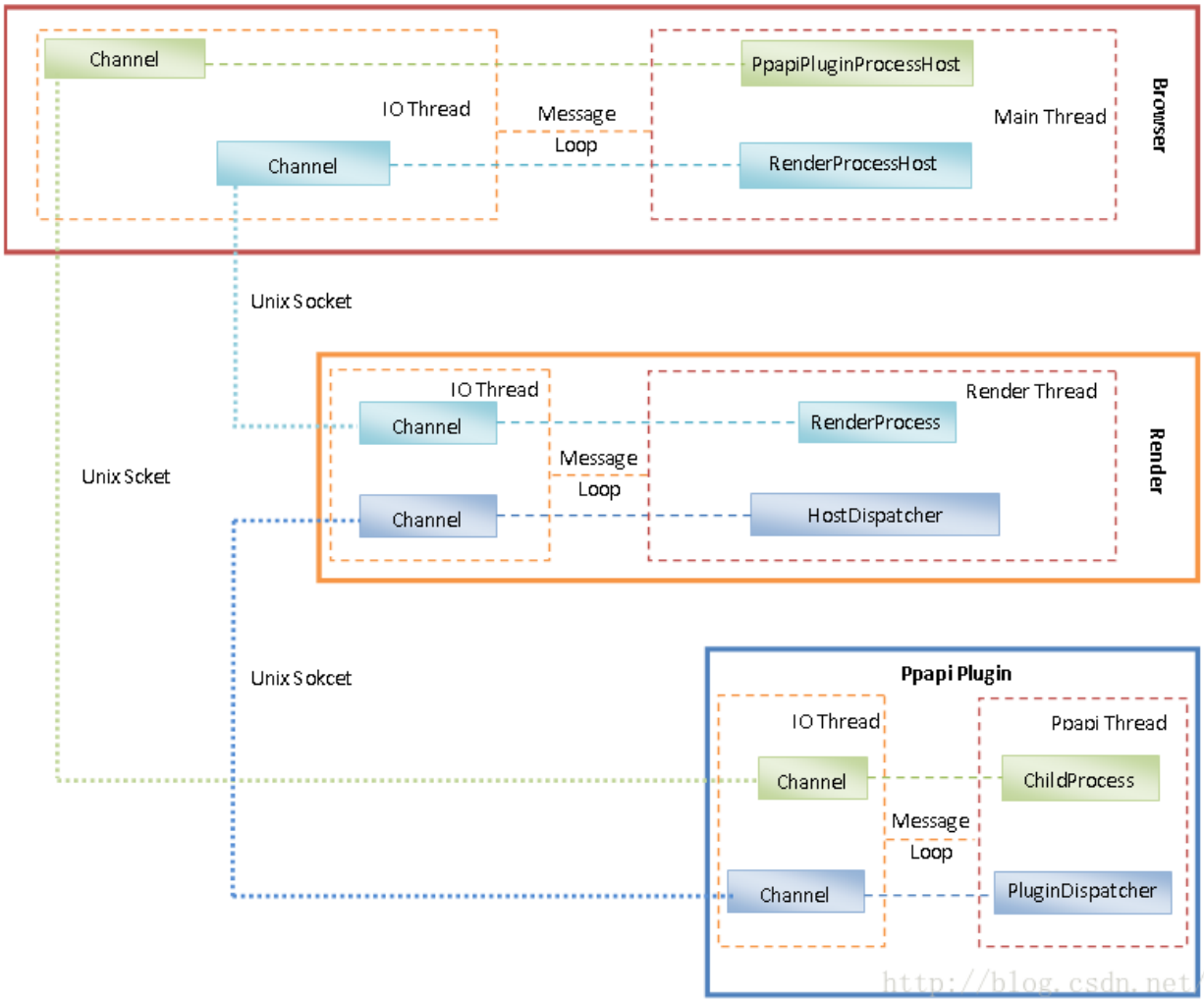

图 2.2: PPAPI 插件进程与 Browser 进程、Render 进程的关系

如图 2.2所示: Render 进程在解析网页时，如果遇到一个 object 标签，那么就会创建一个 HTMLPlugInElement 对象来描述该标签，并且调用该 HTMLPlugInElement 对象的成员函数 loadPlugin 加载对应的 PPAPI 插件。但是 Render 进程没有启动 PPAPI 插件进程的权限，

因此它就会通过之前与 Browser 进程建立的 IPC 通道向 Browser 进程发出一个启动 PPAPI 插件进程的请求。

Browser 进程接收到启动 PPAPI 插件进程的请求之后，就会启动一个 PPAPI 插件进程，并且创建一个 PpapiPluginProcessHost 对象描述该 PPAPI 插件进程。PPAPI 插件进程启动起来之后，会在内部创建一个 ChildProcess 对象，用来与 Browser 进程中的 PpapiPluginProcessHost 对象建立一个 IPC 通道。

有了这个 IPC 通道之后，Browser 进程请求 PPAPI 插件进程创建另外一个 UINX Socket。该 UNIX Socket 的 Server 端文件描述符保留在 PPAPI 进程中，Client 端文件描述符则返回给 Render 进程。基于这个 UNIX Socket 的 Server 端和 Client 端文件描述符，PPAPI 插件进程和 Render 进程将分别创建一个 HostDispatcher 对象和一个 PluginDispatcher 对象，构成一个 Plugin 通道。以后 PPAPI 插件进程和 Render 进程就通过该 Plugin 通道进行通信。

3 运行设计

3.1 注册 PPAPI 插件

chromium 注册插件默认是通过 register-pepper-plugins 启动命令传给浏览器并保存在 PepperPluginInfo 类里，示例如下：

```
--register-pepper-plugins="/data/tbrowser2.61/plugins/
    libdemo_plugin.so;application/demo_plugin;application/
    demo_plugin1,/data/tbrowser2.61/plugins/libutil.so;application/
    util"
```

首先，多个 PPAPI 插件进程以”,” 为间隔，我们称一个 PPAPI 插件进程对应一个 plugin_module。

接着，一个 plugin_module 以”;” 为间隔，第一个参数代表 PPAPI 插件动态库的路径，后面的参数表示 PPAPI 插件的 type 类型，可以有多个。

换而言之，一个 PPAPI 插件动态库对应一个 PPAPI 插件进程，PPAPI 插件动态库与 PPAPI 插件类型可以是一对一，也可以是一对多的关系（详情可见插件之基于 C++ 的实现）。

最后，pepper_plugin_list.cc 文件里描述了 register-pepper-plugins 具体参数格式：

```
command-line = <plugin-entry> + *( LWS + "," + LWS + <plugin-entry>
    )
plugin-entry =
  <file-path> +
  ["#" + <name> + ["#" + <description> + ["#" + <version>]]] +
```

```
  *1( LWS + ";" + LWS + <mime-type-data> )
mime-type-data = <mime-type> + [ LWS + "#" + LWS + <extension> ]
```

3.2 PPAPI 插件生命周期

以单个插件库为例，当页面检测到有 object 标签，且 type 与之前注册的 type 匹配，插件进程就会启动，并且会创建一个 pp::Module 实例。

```
<object id="pluginId" type="application/demo_plugin"></object>
<object id="pluginId1" type="application/demo_plugin1"></object>
```

此时页面检测到两个 object 标签，pp::Module 实例就会创建两个 pp::Instance 的实例，一一对应。当其中一个 object 标签被删除的时候，对应的 pp::Instance 类实例就被析构。当最后一个 object 标签被删除时，对应的 pp::Instance 的实例也被析构。此时，页面已无 object 标签，在 plugin_process_dispatcher.cc 文件里有一个 kPluginReleaseTimeSeconds 整型变量，值为 30，其含义是在 30s 之内，如果页面还是没有创建新的对应该进程插件 type 的 object 标签，pp::Module 类就会被析构，插件进程也会退出。当再次检测到有 object 标签时，重复上过程。

3.3 实现 PPAPI Plugin

基于 PPAPI 机制实现的插件有 C 和 C++ 两种实现。

3.3.1 基于 C 的实现

无论是可信的还是不可信的 PPAPI 组件，一般都会提供三个导出函数：

```
// @brief 初始化PPAPI模块
// @param[in] module 传入浏览器生成的模块句柄PP_Module
// @param[in] get_browser_interface 给插件模块提供浏览器宿主常见的功能的函
    数指针接口，比如PPB_Core、
// PPB_Messaging 、PPB_URLLoader、PPB_FileIO等接口。
// @return 返回PP_OK表示成功，其他都为失败
int32_t PPP_InitializeModule(PP_Module module, PPB_GetInterface
    get_browser_interface);
```

```
// @brief 关闭PPAPI模块
void PPP_ShutdownModule(void);
```

```
// @brief 提供给浏览器的插件功能接口
// @param[in] interface_name 提供给浏览器的插件功能接口字符串，比如
    PPP_Instance、PPP_InputEvent、PPP_Messaging
// 等接口
// @return 返回这个接口的指针，返回NULL表示无此接口
const void* PPP_GetInterface(const char* interface_name);
```

注:除了 PPP_ShutdownModule 是可选的,其他两个导出函数必须实现。PPB_GetInterface 和 PPP_GetInterface 是插件（PPAPI）和宿主（浏览器的 Plugin 进程）功能交互的唯一切入口。

命名规范上，宿主 Browser 提供给插件（PPAPI）的接口以 PPB 命名开头，可以从导出函数 PPP_InitializeModule 传入的 PPB_GetInterface 参数中根据接口 ID 查询对应的功能接口，下面列举一些常用的功能接口：

功能说明	接口 ID	接口定义	重要成员方法说明
核心基础接口	PPB_CORE_INTERFACE	PPB_Core	AddRefResource、ReleaseResource：增加/减少资源的引用计数 GetTime、GetTimeTicks：获取当前的时间 CallOnMainThread、IsMainThread：主线程相关的处理
消息接口	PPB_MESSAGING_INTERFACE	PPB_Messaging	PostMessage：抛送消息（Plugin->JS，JS 通过 addEventListener 挂接 message 事件）RegisterMessageHandler、UnregisterMessageHandler：自定义消息处理对象，用于拦截 PPP_Messaging 接口的 HandleMessage 处理（应用场景未明）
HTTP 请求接口	PPB_URLLOADER_INTERFACE	PPB_URLLoader	常用的 HTTP 请求操作:Open、GetResponseInfo、ReadResponseBody、Close 等等
文件 IO 操作	PPB_FILEIO_INTERFACE	PPB_FileIO	File 的基本操作：Create、Open、Read、Write、Close 等
网络操作	PPB_UDPSOCKET_INTERFACE PPB_TCPSOCKET_INTERFACE PPB_HOSTRESOLVER_INTERFACE 等	PPB_UDPSocket PPB_TCPSocket PPB_HostResolver 等	UDP 套接字、TCP 套接字、域名解析服务等常用的操作接口
音视频操作	PPB_AUDIO_INTERFACE PPB_VIDEODECODER_INTERFACE PPB_VIDEOENCODER_INTERFACE 等	PPB_Audio PPB_VideoDecoder PPB_VideoEncoder 等	封装音视频的编解码等操作

此外还有 Image、2D、3D 等接口，基本上满足开发本地应用的所有功能。

插件（PPAPI）提供给宿主 Browser 的接口以 PPP 命名开头:

功能说明	接口 ID	接口定义	重要成员方法说明
插件实例对象	PPP_INSTANCE_INTERFACE	PPP_Instance	DidCreate、DidDestroy:插件创建销毁通知; DidChangeView、DidChangeFocus: 插件尺寸、焦点改变通知; HandleDocumentLoad: Document 加载完成通知;这些调用最终都映射到 Instance 实例的相关成员方法上
输入接口	PPP_INPUT_EVENT_INTERFACE	PPP_InputEvent	HandleInputEvent: 处理输入消息，最终转接到 Instance 的 HandleInputEvent 接口
消息接口	PPP_MESSAGING_INTERFACE	PPP_Messaging	HandleMessage: 处理消息（JS->Plugin，JS 调用插件的 postMessage 方法），最终映射到 Instance 实例的 HandleMessage 方法

除了这三个基础的接口，Module 对象（后面再细讲）的 AddPluginInterface 提供了扩展插件接口的功能，比如：

封装 PDF 插件功能的 PPP_Pdf 接口；

封装视频采集功能的 PPP_VideoCapture_Dev 接口；

封装视频硬件解码的 PPP_VideoDecoder_Dev 接口；

封装 3D 图形相关的通知事件的 PPP_Graphics3D 接口；

此外还有 PPP_MouseLock、PPP_Find_Private、PPP_Instance_Private 等私有接口。

3.3.2 基于 C++ 的实现

首先,使用 C++ 版本,开发者就无需关心上述三个导出函数的具体实现,框架在 ppp_entrypoints.cc 文件里对三个导出函数做了实现，创建 Module 对象标识整个插件模块：

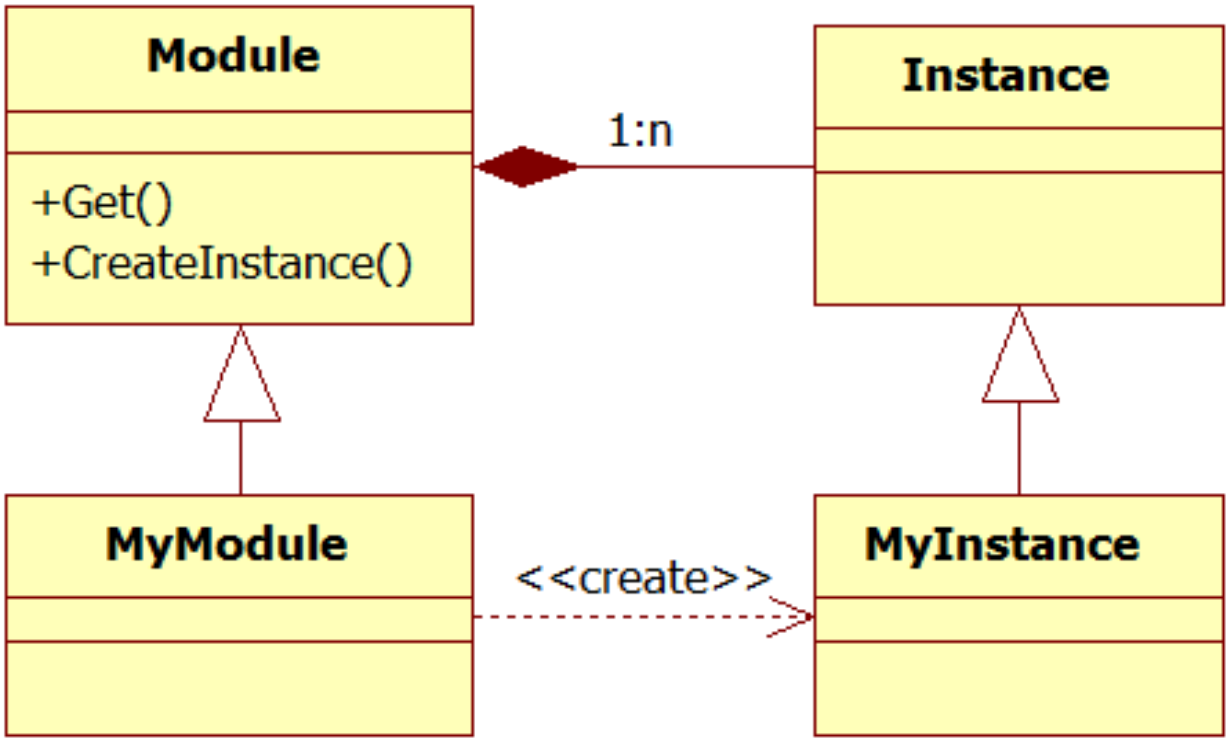

图 3.1: ppapi plugin 静态类图

开发者只需要提供自己的 Module 实现 MyModule，定制自己的插件实例 MyInstance 即可，示例代码：

```
class MyInstance : public pp::Instance {
 public:
  explicit MyInstance(PP_Instance instance) : pp::Instance(instance)
     {}
  virtual ~MyInstance() {}

  virtual bool Init(uint32_t argc, const char* argn[], const char*
   argv[]) {
    return true;
  }
};

// This object is the global object representing this plugin library
    as long
// as it is loaded.
class MyModule : public pp::Module {
 public:
  MyModule() : pp::Module() {}
```

```
  virtual ~MyModule() {}

  // Override CreateInstance to create your customized Instance
    object.
  virtual pp::Instance* CreateInstance(PP_Instance instance) {
    return new MyInstance(instance);
  }
};

namespace pp {
// Factory function for your specialization of the Module object.
Module* CreateModule() {
  return new MyModule();
}
}  // namespace pp
```

上述实现方式继承了默认的 Instance 类，该类供页面操作的函数只有 HandleMessage 和 PostMessage 方法，不能使用自定义的方法和属性。如果想使用自定义的属性和方法，还需以下实现方式。

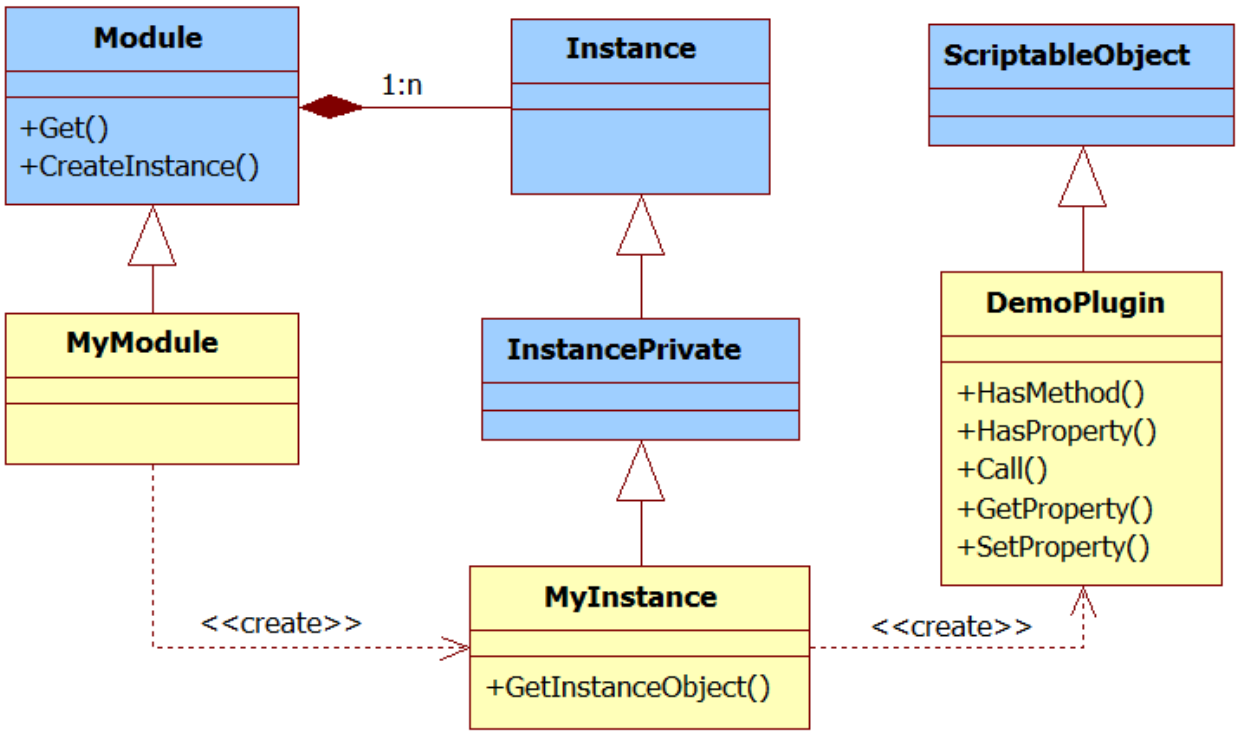

图 3.2: ppapi plugin 可自定义方法静态类图

代码实现：

```
class DemoPlugin : public pp::deprecated::ScriptableObject {
 public:
  explicit DemoPlugin(pp::InstancePrivate* instance)
      : instance_(instance) {}
  ~DemoPlugin() override {}

  // pp::deprecated::ScriptableObject overrides:
  bool HasMethod(const pp::Var& name, pp::Var* exception) override {
    ...
  }

  bool HasProperty(const pp::Var& name, pp::Var* exception) override
    {
    ...
  }

  pp::Var Call(const pp::Var& method_name,
               const std::vector<pp::Var>& args,
               pp::Var* exception) override {
    ...
  }

  pp::Var GetProperty(const pp::Var& name, pp::Var* exception)
   override {
    ...
  }

  void SetProperty(const pp::Var& name,
                   const pp::Var& value,
                   pp::Var* exception) override {
    ...
  }
};


class MyInstance : public pp::InstancePrivate {
 public:
  explicit MyInstance(PP_Instance instance)
      : pp::InstancePrivate(instance) {}
  ~MyInstance() override {}

  // pp::Instance overrides
  bool Init(uint32_t argc, const char* argn[], const char* argv[])
   override {
    ...
  }

  virtual bool HandleInputEvent(const pp::InputEvent& event)
   override {
    ...
  }

```

```
  // pp::InstancePrivate overrides:
  pp::Var GetInstanceObject() override {
    if (instance_var_.is_undefined()) {
      instance_ = new DemoPlugin(this);
      instance_var_ = pp::VarPrivate(this, instance_);
    }
    return instance_var_;
  }

 private:
  pp::VarPrivate instance_var_;
};

class MyModule : public pp::Module {
 public:
  MyModule() {}
  ~MyModule() override {}

  virtual pp::Instance* CreateInstance(PP_Instance instance) {
    return new MyInstance(instance);
  }
};

namespace pp {

Module* CreateModule() {
  return new MyModule();
}

}  // namespace pp
```

问：如何一个插件动态库可以对应多个插件类型呢？

页面：

```
<object id="pluginId" type="application/demo_plugin"></object>
<object id="pluginId1" type="application/demo_plugin1"></object>
```

由上面页面，假设插件 type 都匹配上了，那么 MyInstance 类会创建两个实例，分别执行重写的 Init 方法。Init 方法三个参数含义分别是参数个数、参数 key 值、参数 value 值，其实就是对应上面页面的”id” 和”type”。具体实现如下：

```
class MyInstance : public pp::InstancePrivate {
 public:
  explicit MyInstance(PP_Instance instance)
      : pp::InstancePrivate(instance) {}
  ~MyInstance() override {}
```

```
6
7   // pp::Instance overrides
8   bool Init(uint32_t argc, const char* argn[], const char* argv[])
      override {
9     if (argc < 1)
10      return false;
11
12    for (uint32_t i = 0; i < argc; i++) {
13      if(std::string(argn[i]) == "type")
14        mime_type_ = argv[i];
15    }
16    return true;
17  }
18
19  // pp::InstancePrivate overrides:
20  pp::Var GetInstanceObject() override {
21    if (instance_var_.is_undefined()) {
22      if (mime_type_ == "application/demo_plugin")
23        instance_ = new DemoPlugin(this);
24      else if (mime_type_ == "application/demo_plugin1")
25        instance_ = new DemoPlugin1(this);
26      instance_var_ = pp::VarPrivate(this, instance_);
27    }
28    return instance_var_;
29  }
30
31 private:
32  pp::VarPrivate instance_var_;
33  std::string mime_type_;
34 };
```

我们只需要在 Init 函数的时候保存插件的 mimetype 值，然后在 GetInstanceObject 函数的时候，根据 mimetype 值 new 相应的插件实现类即可。

4 其他

4.1 调用性能对比

在 js 中调用插件方法，传递 1024 字节的字符串到 c++ 代码，连续调用 1000 次，测试 20 组数据。使用 NPAPI 机制平均耗时 1658ms；而使用 PPAPI 机制耗时平均耗时 8136ms，经过代码优化后，目前平均耗时 2208ms。

4.2 时序图

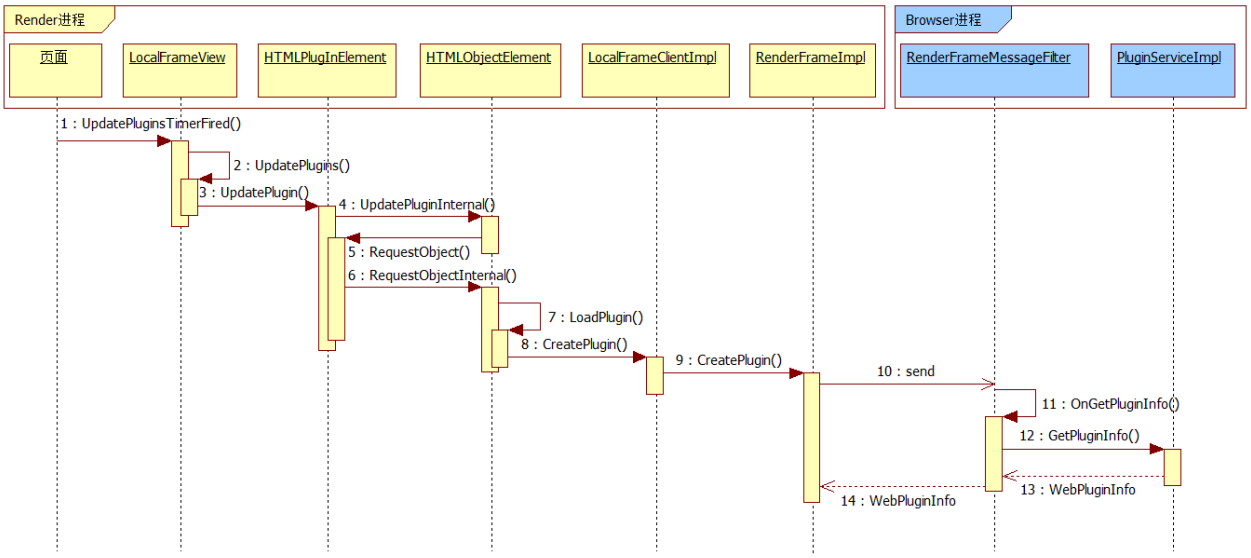

图 4.1: 判断 object 标签是否存在相应 PPAPI 插件 type 时序图

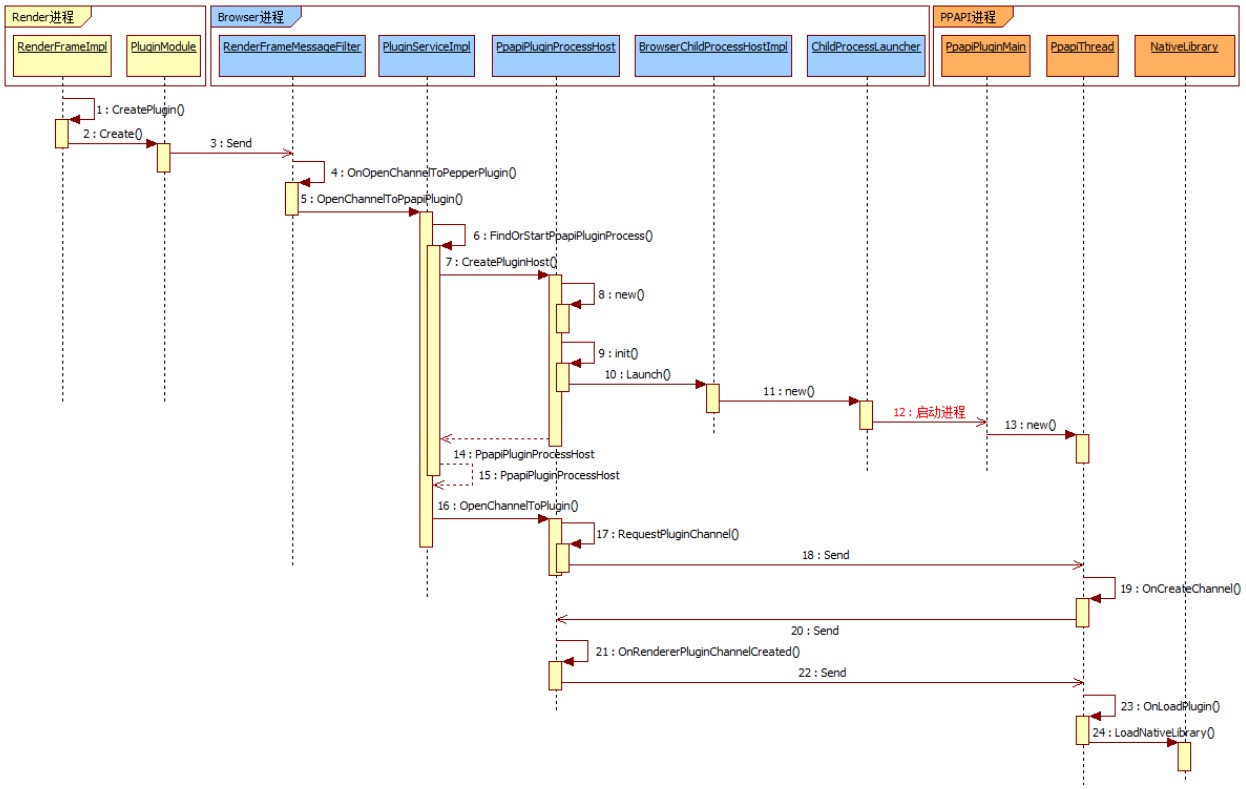

图 4.2: PPAPI 插件进程启动以及加载插件库时序图