

chromium 按键事件处理流程研究

王勇望

2016 年 10 月 22 日

1 引言

众所周知，chromium 是多线程框架的，整个 chromium 浏览器包括四类进程：browser 主进程、render 渲染进程、GPU 进程和插件进程。本文所要讲述的 chromium 事件处理流程主要牵涉到其中的两种进程：browser 进程和 render 进程。

事件处理的大体流程是由 browser 进程接收并传递给 render 进程处理。然而我们知道 render 会不止一个，那么 browser 进程如何获取事件？又如何传递给特定的 render 进程？render 进程如何处理事件消息？我们下面就通过分析代码来一一探明。

chromium 系统有多个平台的实现，而每个平台都有自己不同的事件管理方式，chromium 也会有一些平台相关的代码。本文目前主要是研究 Linux 平台相关的实现，其他平台后续再做研究。chromium 中的事件也有许多类型，如按键事件、鼠标事件、滚轮事件等等。本文也只是以按键事件为例研究，在以下内容中，如无特殊说明，事件均指的是按键事件。

2 按键事件获取流程

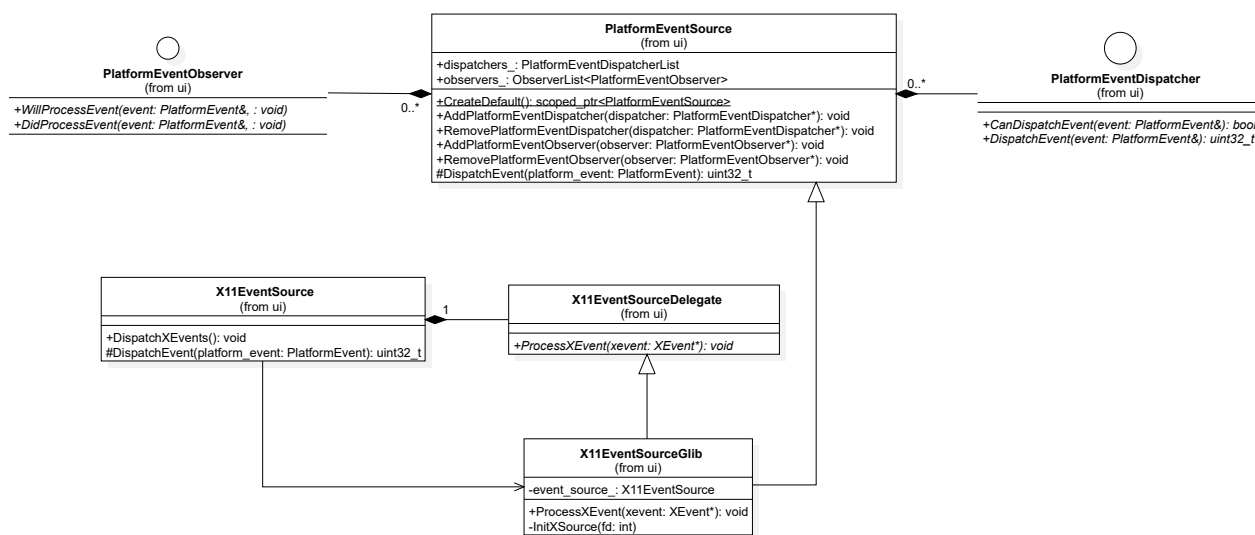


图 1: chromium linux 平台事件源静态类图

如第图 1所示: chromium 对事件源抽象了一个 PlatformEventSource 类表示，不同的平台继承该类实现自己平台的子类。不过，目前发现仅有 Linux 平台实现了其子类 X11EventSource 与 X11EventSourceGlib。Windows 和 Android 都未采用该方法。

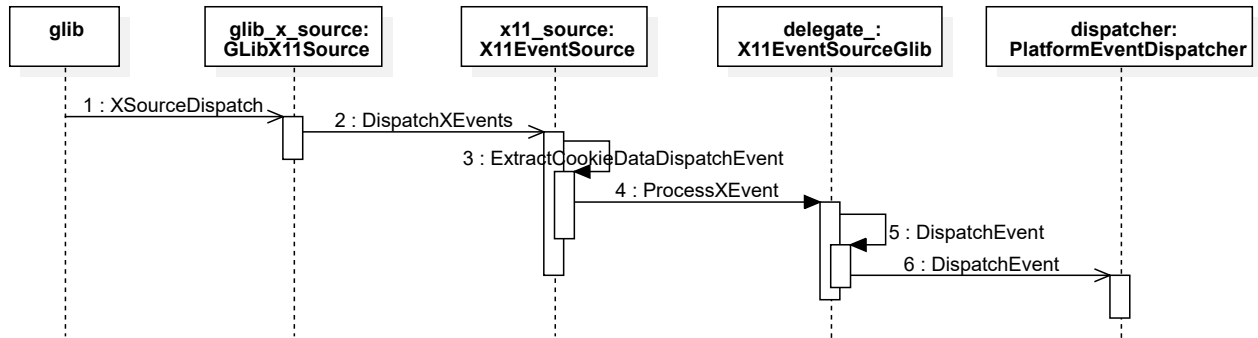


图 2: chromium linux 平台事件获取时序图

browser 进程获取按键事件的调用时序图如图 2所示:

- 在 browser 进程启动时, 会创建一个线程循环调用 `g_main_context_iteration` 方法。其作用是驱动 glib 检查事件, 如果有事件发生, 则 glib 会调用初始化时注册的 `XSourceDispatch` 回调来将事件传递给 `PlatformEventSource` 的子类。
- 第 5 步调用时是调用的父类 `PlatformEventSource` 方法, 将事件传递给 `PlatformEventDispatcher`。

```

1  for (;;) {
2      // Don't block if we think we have more work to do.
3      bool block = !more_work_is_plausible;
4
5      more_work_is_plausible = g_main_context_iteration(context_,
6      block);
7      if (state_>should_quit)
8          break;
9
10     more_work_is_plausible |= state_>delegate->DoWork();
11     if (state_>should_quit)
12         break;
13
14     more_work_is_plausible |=
15         state_>delegate->DoDelayedWork(&delayed_work_time_);
16     if (state_>should_quit)
17         break;
18
19     if (more_work_is_plausible)
20         continue;
21
22     more_work_is_plausible = state_>delegate->DoIdleWork();
23     if (state_>should_quit)
24         break;
25 }
  
```

3 按键事件传递流程

这里的按键传递流程是指 browser 进程从事件源处取得事件之后将其传递给特定的 render 进程的过程。这里研究的是使用 chromium 的 aura 窗口系统的流程，平台是 Linux 平台，然而据初步研究 Windows 和 Android 也是使用同样的流程，只是传入事件的位置不同罢了。

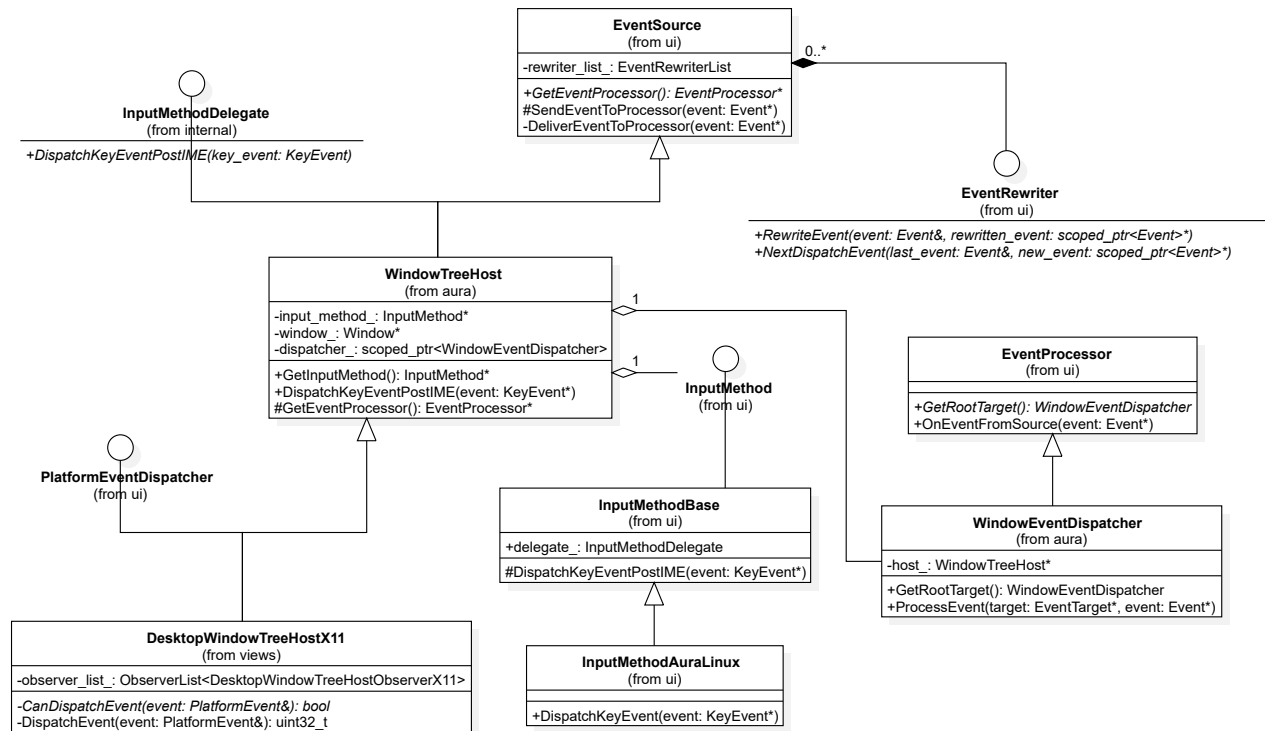


图 3: chromium browser 进程事件传递静态类图-1

图 3 是按键事件从 PlatformEventSource 到按键分发到特定 window 之前, 所涉及的主要类的静态类图。

- DesktopWindowTreeHostX11 类实现 PlatformEventDispatcher 接口, 同时又继承了 WindowTreeHost 和 EventSource。这样 DesktopWindowTreeHostX11 类在 browser 进程中就扮演了重要的角色, 可谓身兼数职。在事件处理流程上看它既是 PlatformEventSource 的 dispatcher, 又是 UI 系统的 EventSource。
- EventSource 中包含 EventRewriter 对象, 可以在事件处理前改写事件。
- InputMethodBase 内存在 WindowTreeHost 的引用。

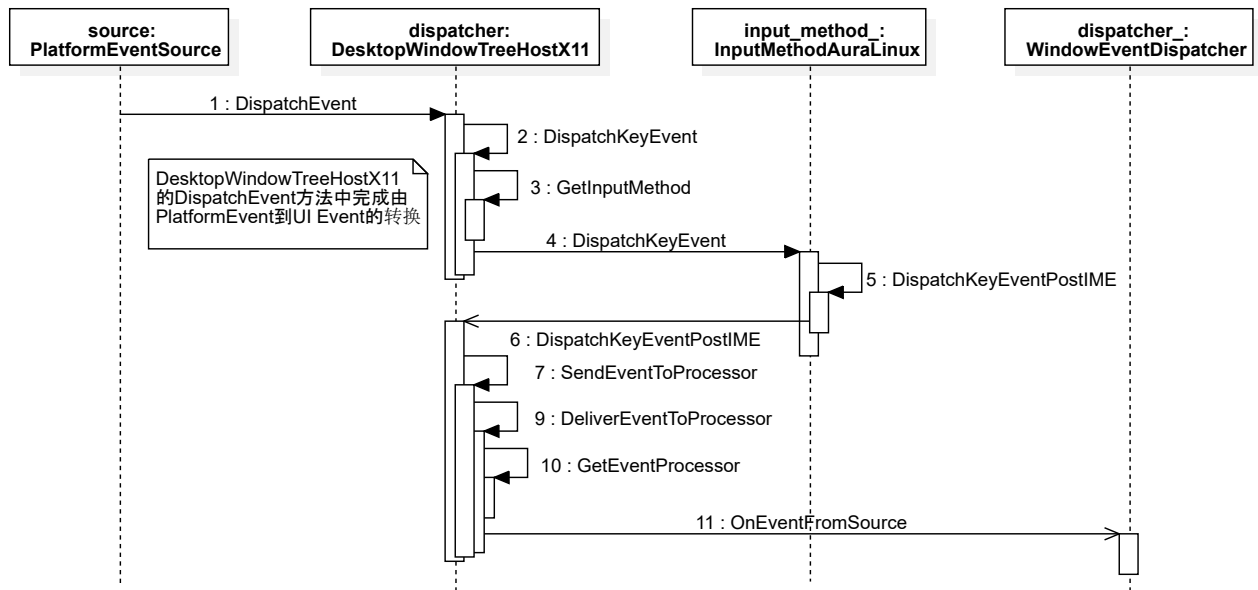


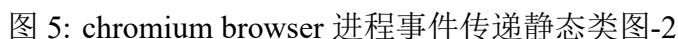
图 4: chromium browser 进程事件传递时序图-1

图 4 是按键事件从 PlatformEventSource 到按键分发到特定 window 之前的调用时序图。

- 在第 1 步调用中（DispatchEvent）就完成了由 PlatformEvent 到 ui::Event 的转换。也就屏蔽了平台层的事件，转换为 chromium 内部的事件类型。
- DesktopWindowTreeHostX11 中的大部分调用都在其父类和祖父类中实现的。
- OnEventFromSource 函数在 WindowEventDispatcher 的父类中实现。

```

1  case KeyPress: {
2      ui::KeyEvent keydown_event(xev);
3      DispatchKeyEvent(&keydown_event);
4      break;
5  }
6  case KeyRelease: {
7      // There is no way to deactivate a window in X11 so ignore
input if
8      // window is supposed to be 'inactive'. See comments in
9      // X11DesktopHandler::DeactivateWindow() for more details.
10     if (!IsActive() && !HasCapture())
11         break;
12
13     ui::KeyEvent key_event(xev);
14     DispatchKeyEvent(&key_event);
15     break;
16 }
  
```



- WindowTreeHost 继承了 EventSource，通过实例化其子类 DesktopWindowTreeHostX11 来充当 UI 抽象的事件源。
- EventDispatcher 和 EventDispatcherDelegate 是事件分发工作的主要执行者，EventDispatcherDelegate 类实例化的是其子类 WindowEventDispatcher。
- window 继承自 EventTarget 可以作为事件的目标，同时又包含 EventTargeter 对象，用来筛选合适的 window 处理事件。
- EventTarget 中实际处理事件的对象是 EventHandler。EventTarget 中包含 3 类 handler，其中主处理 handler 仅有一个是 target_handler。在主 handler 处理前会有 pre_target_list 中的 handler 尝试处理事件，主 handler 处理后由 post_target_list_ 尝试处理。这里的处理大部分是指传递给 Render 进程。
- RenderWidgetHostViewAura 类继承 EventHandler，具有处理事件的能力。通过调试代码发现并不是所有的 window 内的 target_handler 都是 RenderWidgetHostViewAura。目前的结论是一个 Render 进程中的所有 window 只有一个 target_handler 是 RenderWidgetHostViewAura。
- RenderWidgetHostImpl 与 Render 进程中的 RenderWidget 对应。

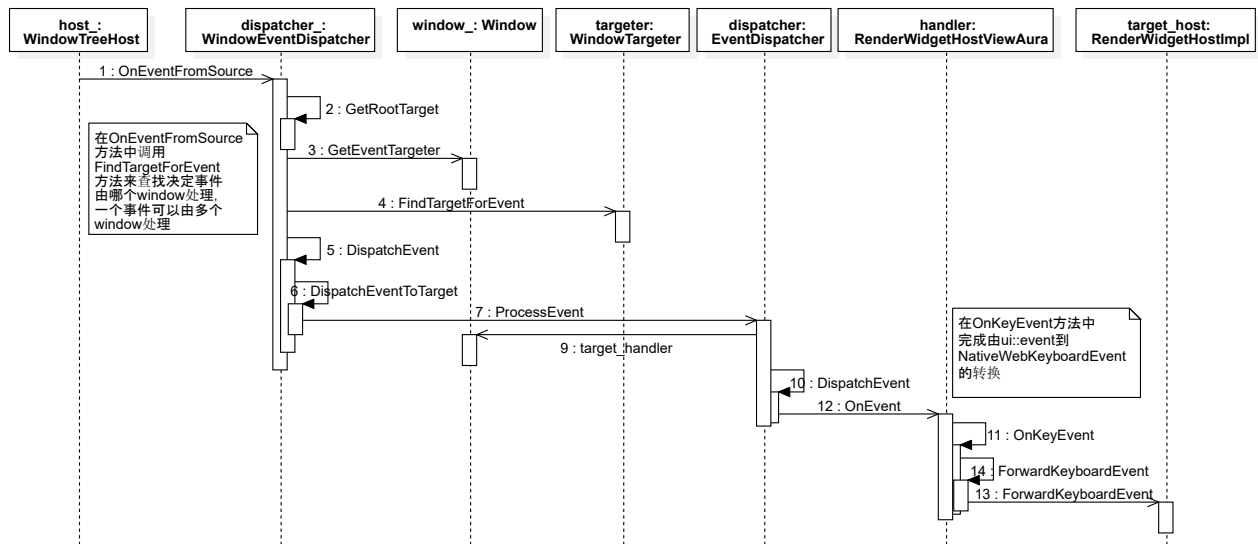


图 6: chromium browser 进程事件传递时序图-2

图 6 是按键事件从 PlatformEventSource 到按键分发到特定 window 之前的调用时序图。

- dispatcher_ 对象的调用一些方法实现在其父类和祖父类中。其中调用 GetRootTarget 方法返回的对象就是 root window。
- WindowTargeter 类的 FindTargetForEvent 方法返回合适的 window 对象作为事件的 target。由于 window 是树状结构，处理是会循环调用 FindNextBestTarget 方法。
- RenderWidgetHostViewAura 类中的 OnKeyEvent 方法中完成 ui::KeyEvent 到 NativeWebKeyboardEvent 转换, NativeWebKeyboardEvent 又是 blink::WebKeyboardEvent 的子类。所以按键事件是通过 WebKeyboardEvent 的形式传递给 Render 进程的。

```

1 if (!event_to_dispatch->handled())
2 target = targeter->FindTargetForEvent(root, event_to_dispatch);
3
4 EventDispatchDetails details;
5 while (target) {
6 details = DispatchEvent(target, event_to_dispatch);
7
8 ...
9
10 target = targeter->FindNextBestTarget(target, event_to_dispatch);
11 }

```

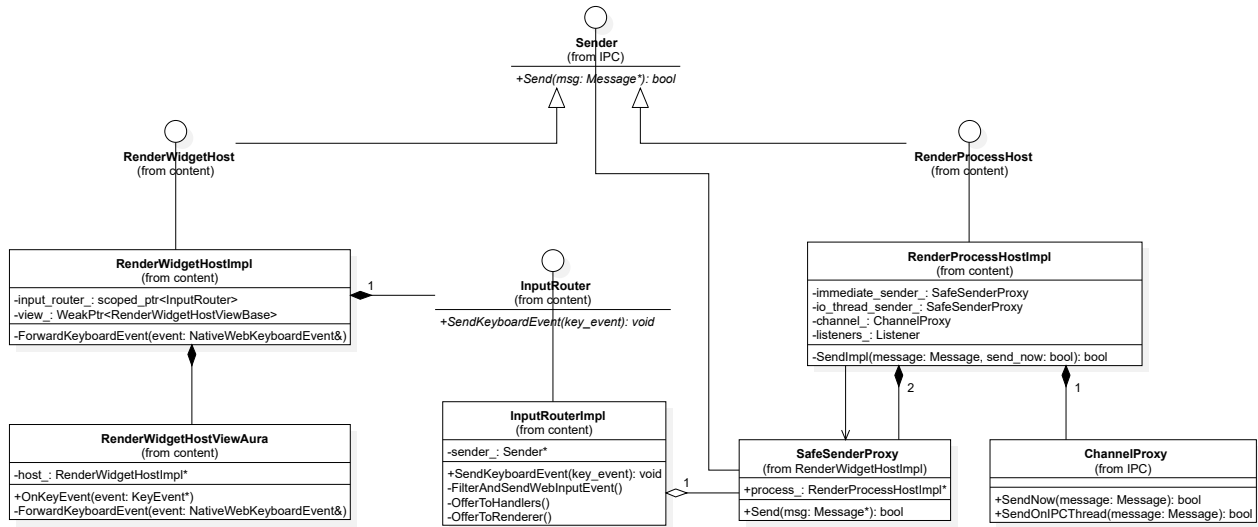


图 7: chromium browser 进程事件传递静态类图-3

如图 7 所示：是 browser 进程发送按键消息到 render 进程过程中涉及的类的静态类图，其中：

- RenderWidgetHostImpl、InputRouter、RenderProcessHost 等类同时也继承了 Listener 接口，用于接收 render 进程的消息。
- RenderWidgetHost 与 Render 进程中的 RenderWidget 对应，在 chromium 给出的多进程框架示意图中可以看出一个 render 进程可以存在多个 RenderWidget，但只有一个 RenderProcess 负责与 browser 进程通信。同样，每个 render 进程在 browser 进程中都有一个 RenderProcessHost。
- 发送消息的具体实施者是 ChannelProxy 类，在 browser 进程中实例化的 ChannelProxy 对象只能发送异步消息，后来 chromium 有在对象内部添加了发送同步消息的函数，但是对外的接口仍默认是异步消息。
- 发送消息的底层实现并没有在这里体现，留待后续研究。

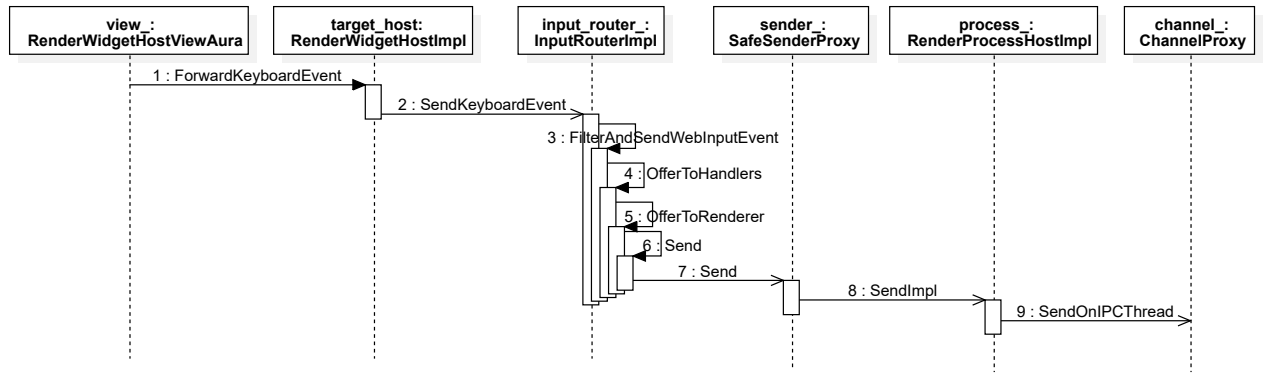


图 8: chromium browser 进程事件传递时序图-3

图 8 是 browser 进程发送按键消息到 render 进程过程中的调用时序图，其中：

- SendImpl 调用 SendOnIPCThread 是发送异步消息，同步消息是调用 SendNow。

```

1  bool RenderProcessHostImpl::SendImpl(std::unique_ptr<IPC::Message>
    msg,
2                                     bool send_now) {
3  #if !defined(OS_ANDROID)
4      DCHECK(!msg->is_sync());
5  #endif
6      if (!channel_) {
7  #if defined(OS_ANDROID)
8          if (msg->is_sync())
9              return false;
10 #endif
11         if (!is_initialized_) {
12             queued_messages_.emplace(std::move(msg));
13             return true;
14         } else {
15             return false;
16         }
17     }
18
19     if (child_process_launcher_.get() && child_process_launcher_->
        IsStarting()) {
20  #if defined(OS_ANDROID)
21         if (msg->is_sync())
22             return false;
23  #endif
24         queued_messages_.emplace(std::move(msg));
25         return true;
26     }
27
28     if (send_now)
29         return channel_->SendNow(std::move(msg));
30
31     return channel_->SendOnIPCThread(std::move(msg));
32 }
  
```

4 按键事件处理流程

这里的按键事件处理流程指的是 render 进程接收到按键事件之后的处理流程。大致过程是从 browser 进程获取事件；查找事件处理的页面、节点；调用节点注册的监听函数处理事件。

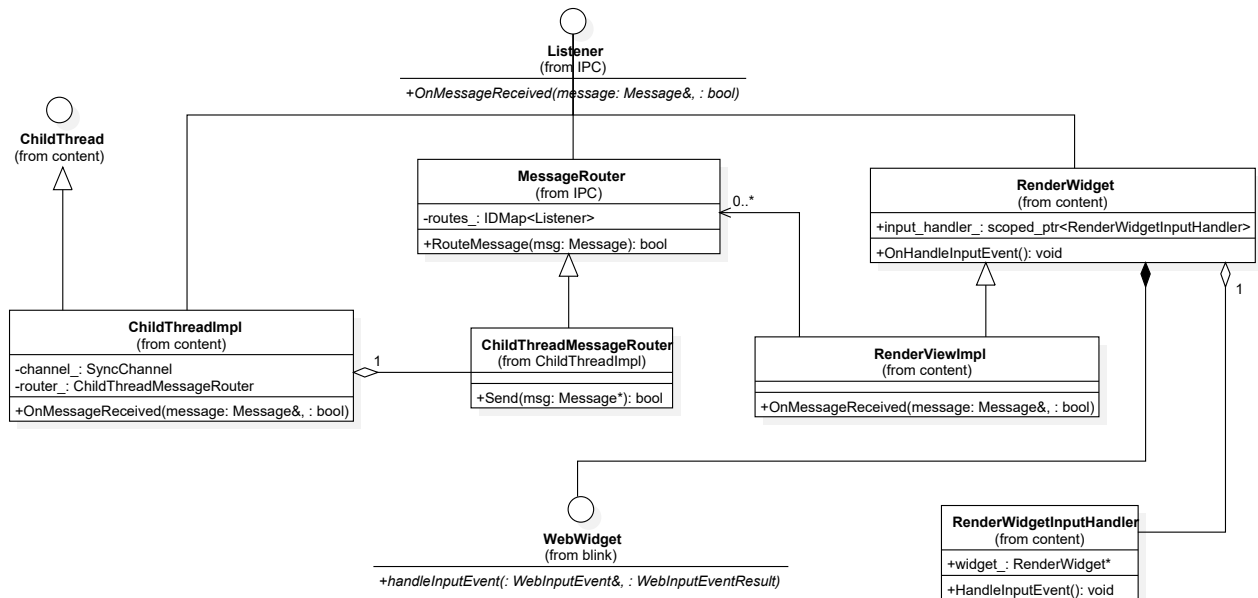


图 9: chromium render 进程处理事件类图-1

如图 9 所示：是 browser 进程的按键消息到达 render 进程后的，到传递给 render 进程中的特定 WebView 处理之前所涉及的类的静态类图。其中：

- ChildThread、MessageRouter、RenderWidget 等类同时也继承了 Sender 接口，用于发送消息到其他进程。
- 消息最先到达 ChildThread 的 OnMessageReceived 函数，在这里会处理一些消息，剩余消息交于 ChildThreadMessageRouter 处理。MessageRouter 将事件传递给 RenderViewImpl 处理，RenderViewImpl 也会处理一些消息，其余交于其父类 RenderWidget 处理，按键事件最终会传递给 RenderWidget。
- ChildThreadImpl 中的 channel 实例是 SyncChannel 对象，可以发送同步消息。
- 接收消息的底层实现并没有在这里体现，留待后续研究。

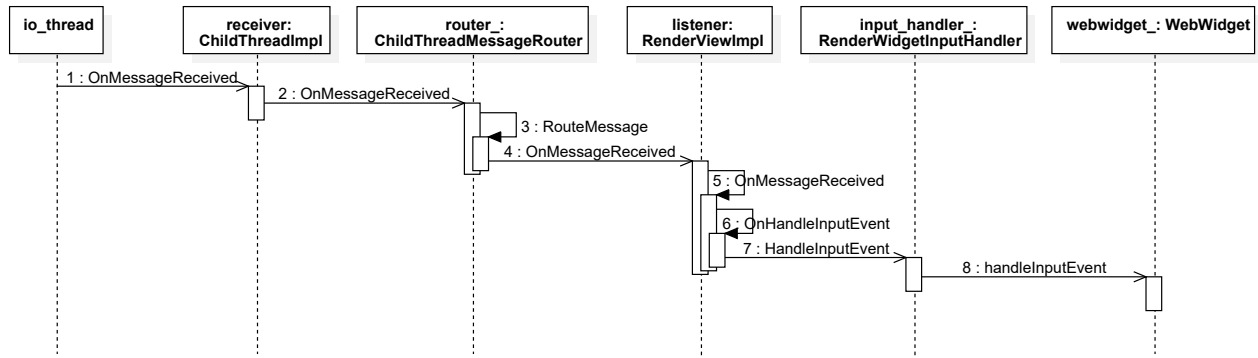


图 10: chromium render 进程处理事件时序图-1

图 10 是 browser 进程的按键消息到达 render 进程后的，到传递给 render 进程中的特定 WebView 处理之前的调用时序图，其中：

- 在 MessageRouter 中的 RouteMessage 中选择哪个 RenderView 处理事件。
- RenderWidgetInputHandler 调用 RenderWidget 的接口获取 RenderWidget 的 webwidget_ 处理事件。

```

1 bool MessageRouter::RouteMessage(const IPC::Message& msg) {
2     IPC::Listener* listener = routes_.Lookup(msg.routing_id());
3     if (!listener)
4         return false;
5
6     return listener->OnMessageReceived(msg);
7 }
8
9 void RenderWidgetInputHandler::HandleInputEvent(
10     const WebInputEvent& input_event,
11     const ui::LatencyInfo& latency_info,
12     InputEventDispatchType dispatch_type) {
13     ...
14
15     WebInputEventResult processed = prevent_default
16                                     ? WebInputEventResult::
17                                     HandledSuppressed
18                                     : WebInputEventResult::
19                                     NotHandled;
20     if (input_event.type != WebInputEvent::Char || !
21         suppress_next_char_events_) {
22         suppress_next_char_events_ = false;
23         if (processed == WebInputEventResult::NotHandled && widget_->
24             webwidget())
25             processed = widget_->webwidget()->handleInputEvent(input_event
26             );
27     }
28     ...
29 }

```



- `WebViewImpl` 实现 `WebWidget` 接口和 `WebView` 接口功能，是 `RenderWidget` 的成员变量。传递给它的事件将传递给其成员变量 `m_page`。
- 在一个 `Page` 中可能包含多个 `Frame`，`Page` 将选择合适的 `Frame` 处理传入的事件。事件将交于 `Frame` 中的 `m_eventHandler` 变量处理。
- 在 `EventHandler` 中，将遍历 DOM 树，查找合适的 `Node` 处理事件。最终通过 `Node` 的父类 `EventTarget` 调用所以注册的监听函数。
- 此过程使用的类都属于 `blink` 命名空间，是原来的 `WebKit` 内核代码。

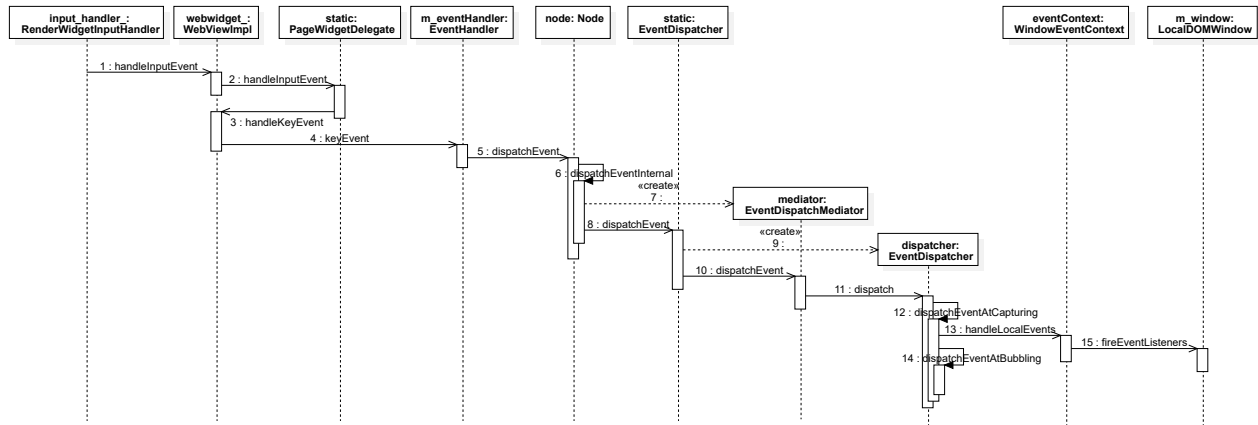


图 12: chromium render 进程处理事件时序图-2

图 12 是 render 进程的渲染引擎 blink 处理按键事件流程调用时序图，其中：

- 第 3 步 `handleKeyEvent` 在调用 `keyEvent` 方法处理后，若无处理将调用 `keyEventDefault` 进行事件的默认处理。
- 第 4 步 `keyEvent` 方法中将调用 `eventTargetNodeForDocument` 函数，用来查找合适的 Node 来处理事件。
- 第 11 步 `EventDispatcher` 的 `dispatch` 方法中首先调用 `dispatchEventAtCapturing` 进行事件捕获，然后调用 `dispatchEventAtBubbling` 进行事件冒泡，这些过程中都会检查事件是否要继续处理。

```

1 DispatchEventResult EventDispatcher::dispatch()
2 {
3     ...
4
5     EventDispatchHandlingState* preDispatchEventHandlerResult =
6     nullptr;
7     if (dispatchEventPreProcess(preDispatchEventHandlerResult) ==
8     ContinueDispatching) {
9         if (dispatchEventAtCapturing() == ContinueDispatching) {
10             if (dispatchEventAtTarget() == ContinueDispatching)
11                 dispatchEventAtBubbling();
12         }
13     }
14     ...
15     return EventTarget::dispatchEventResult(*m_event);
16 }
  
```