

2.1 实战项目综述



扫码试看/订阅

《Electron 开发实战》视频课程

项目需求-远程控制

背景

提供软件服务时在复杂情况下（如排查故障、演示功能），通过沟通解决非常低效。

目标

希望迅速为用户提供远程协助，能够远程控制用户电脑，提供点击和键入功能，完成排查故障、演示场景。

业务流程

角色

控制端：客服人员/研发人员

傀儡端：用户

流程

1. 傀儡端告知控制端本机控制码
2. 控制端输入控制码连接傀儡端
3. 傀儡端将捕获的画面传至控制端
4. 控制端的鼠标和键盘指令传送至傀儡端
5. 傀儡端响应控制指令

项目演示

2.2 项目设计思路

项目关键点分析

1. 僵尸端告知控制端本机控制码
2. 控制端输入控制码连接僵尸端
3. 僵尸端将捕获的画面传至控制端
4. 控制端的鼠标和键盘指令传送至僵尸端
5. 僵尸端响应控制指令

建立端与控制码的联系 -> 服务端需求

项目关键点分析

1. 僵尸端告知控制端本机控制码
2. 控制端输入控制码**连接**僵尸端
3. 僵尸端将捕获的画面传至控制端
4. 控制端的鼠标和键盘指令传送至僵尸端
5. 僵尸端**响应**控制指令

通过控制码找到用户 -> 服务端需求

建立用户间连接 -> 服务端需求 or 客户端需求

项目关键点分析

1. 僵尸端告知控制端本机控制码
2. 控制端输入控制码连接僵尸端
3. 僵尸端将**捕获的画面传至**控制端
4. 控制端的鼠标和键盘指令传送至僵尸端
5. 僵尸端响应控制指令

捕获画面、播放画面-> 客户端需求

用户间画面传输 -> 服务端需求/客户端需求

项目关键点分析

1. 僵尸端告知控制端本机控制码
2. 控制端输入控制码连接僵尸端
3. 僵尸端将捕获的画面传至控制端
4. 控制端的鼠标和键盘指令传送至僵尸端
5. 僵尸端响应控制指令

捕获指令 -> 客户端需求

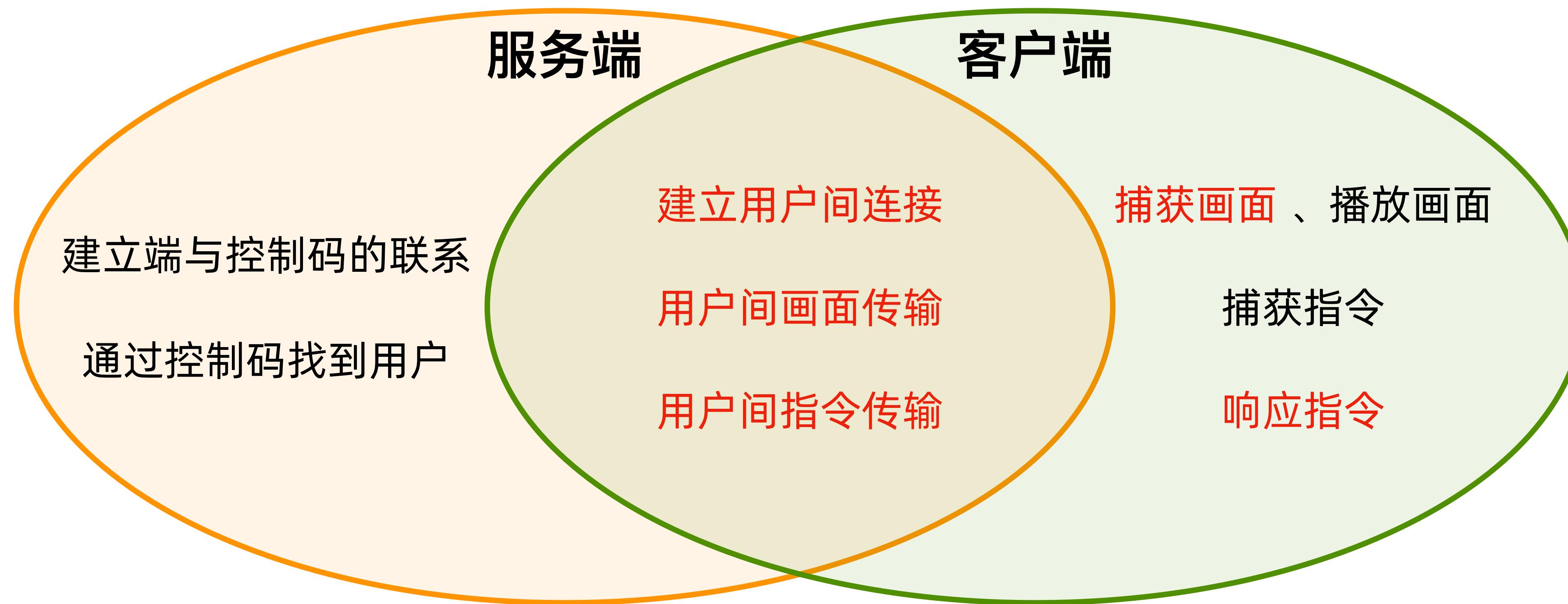
用户间指令传输 -> 服务端需求/客户端需求

项目关键点分析

1. 僵尸端告知控制端本机控制码
2. 控制端输入控制码连接僵尸端
3. 僵尸端将捕获的画面传至控制端
4. 控制端的鼠标和键盘指令传送至僵尸端
5. 僵尸端**响应**控制指令

响应指令 -> 客户端需求

需求汇总



技术关键点

1. 怎么捕获画面？

Electron desktopCapturer

desktopCapturer

通过[`navigator.mediaDevices.getUserMedia`] API，可以访问那些用于从桌面上捕获音频和视频的媒体源信息。

进程: [Renderer](#)

技术关键点

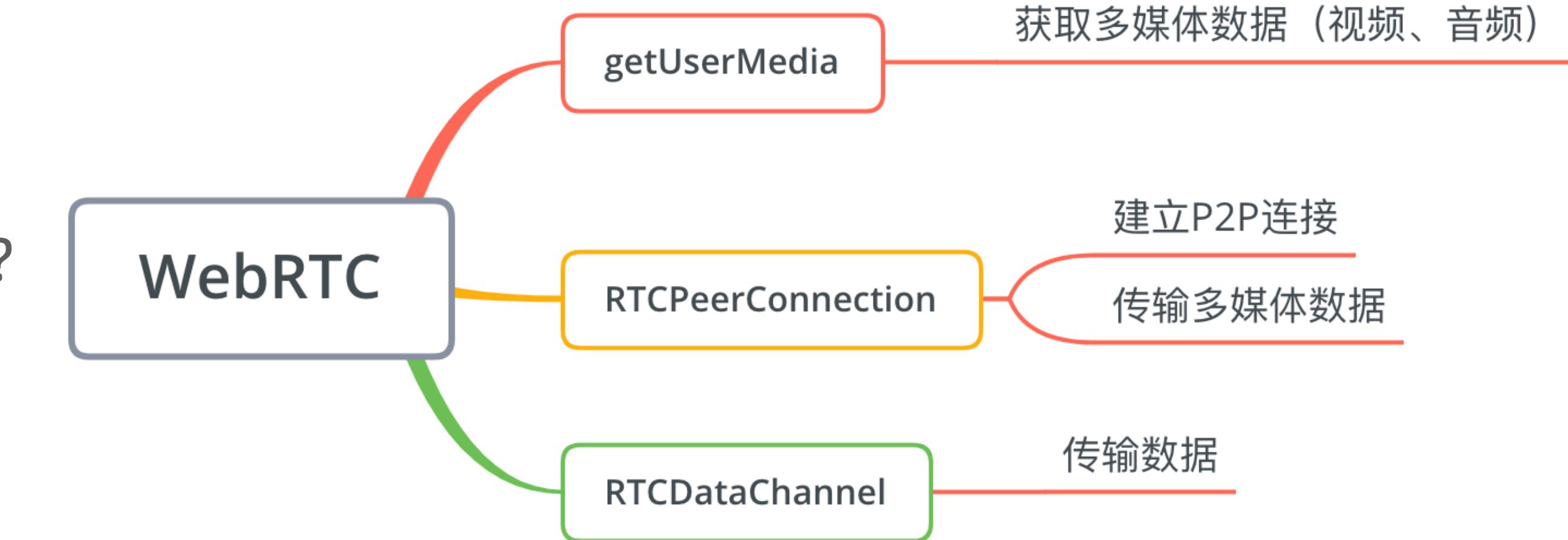
1. 怎么捕获画面?

Electron desktopCapture

2. 怎么完成用户间连接、画面+指令传输?

WebRTC

Web Real-Time Communications



技术关键点

1. 怎么捕获画面?

Electron desktopCapture

2. 怎么完成用户间连接、画面+指令传输?

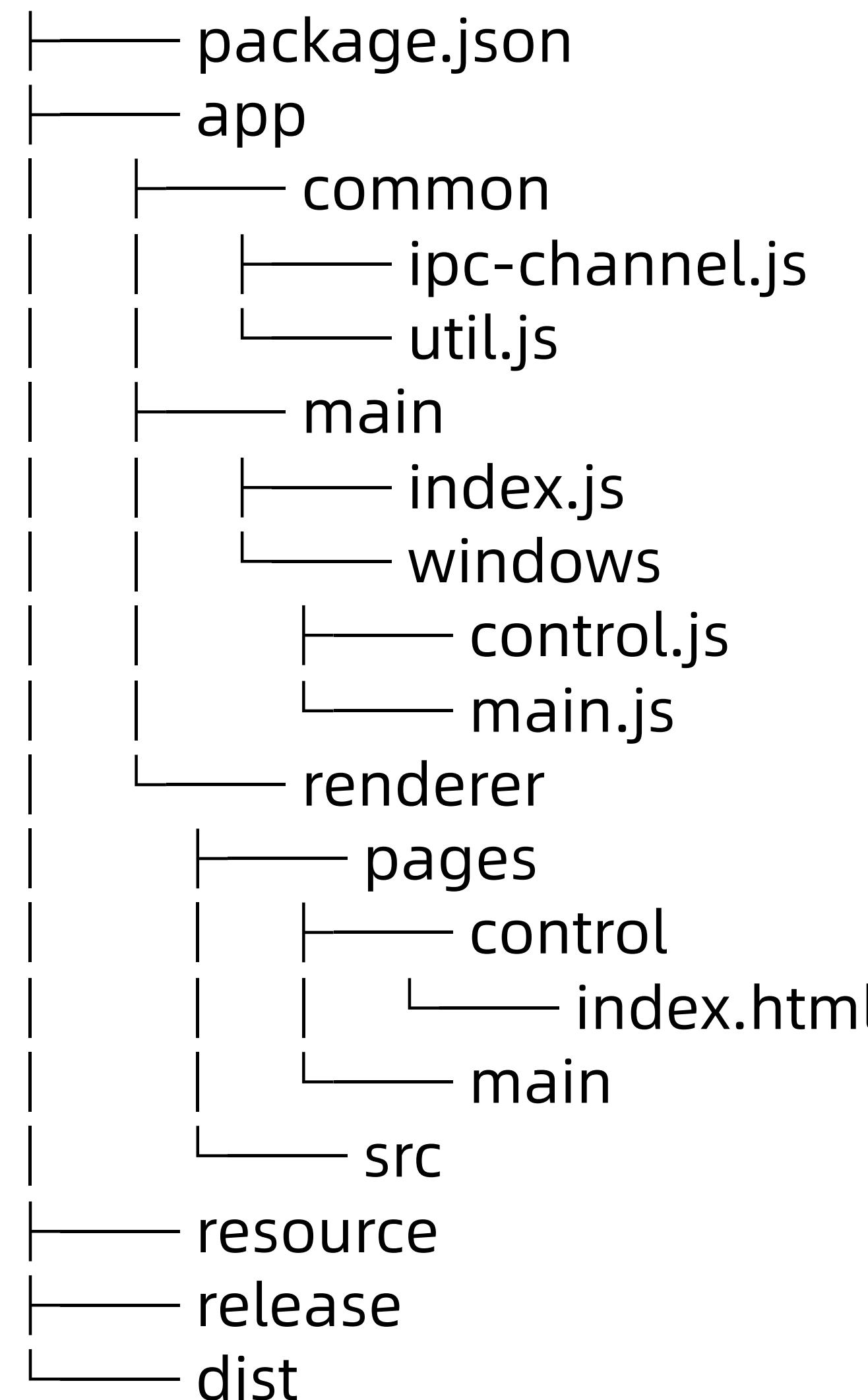
WebRTC

3. 怎么响应控制指令?

robotjs (Node.js)

2.3 项目架构：Electron 与 React 框架结合

目录架构



- common 存放渲染进程、主进程可复用代码
- 前端框架在 render/src/ 页面，构建产物在 pages/ 页面
- 纯 JS 直接在 Pages 页面下

与 React 框架结合

跟 Electron 在一起工作要做些什么呢？

- 书写 React，并且编译它。CRA 其实一个好的选择。
- 处理引入 electron/node 模块：
 - Webpack 配置：<https://webpack.js.org/configuration/target/>
 - window.require
- Windows 根据环境信息加载本地或者 devServer url
 - electron-is-dev
- 启动命令适配。等到编译成功再启动 Electron
 - concurrently
 - wait on

代码演示

其他模板

- electron-react-boilerplate
- electron-vue
- svelte-template-electron
- angular-electron

代码演示

2.4 主页面基础业务：Real World IPC

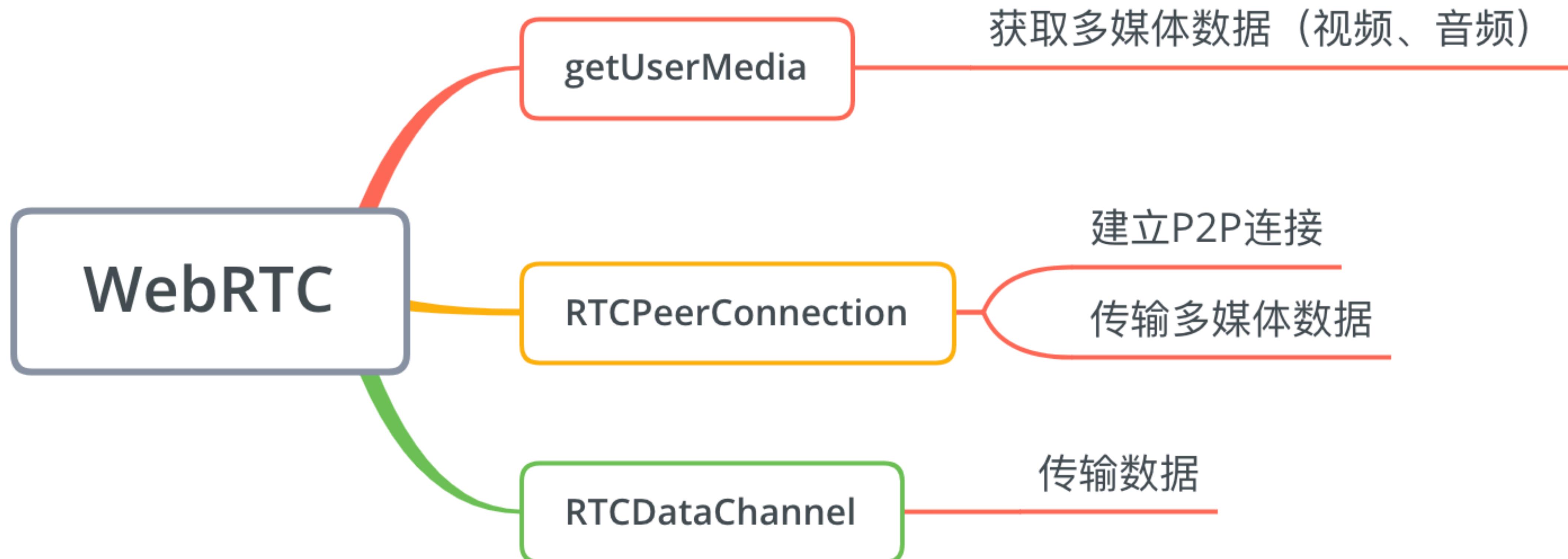
基础业务页面

- 获取自身控制码
- 发起控制：文本框 + 确认按钮
- 连接状态：未连接、正在控制屏幕、屏幕被控制中
- 确认按钮点击后创建控制屏幕窗口

IPC 回顾

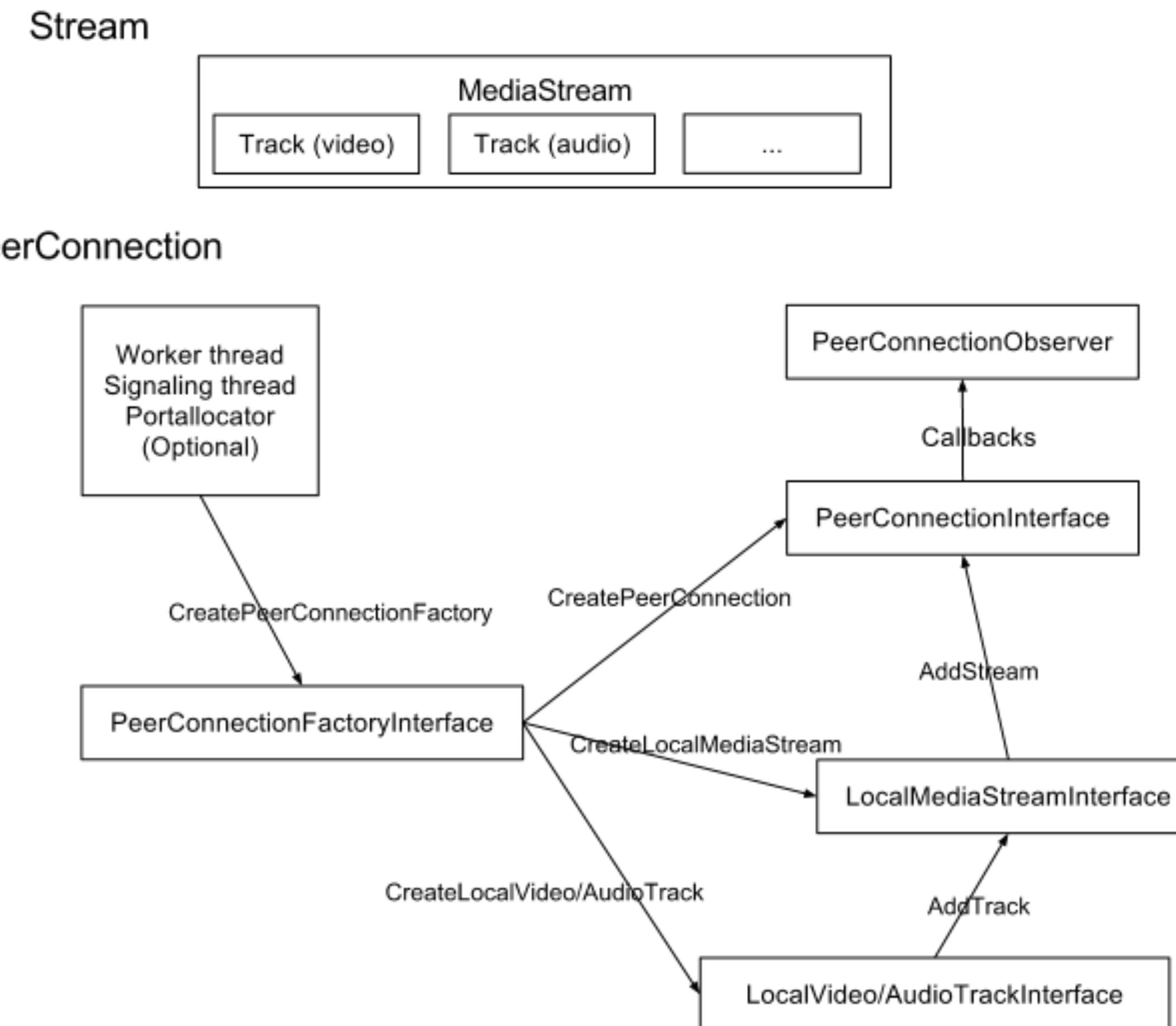
- 渲染进程请求+主进程响应（获取自己的控制码） ipcRenderer.invoke + ipcMain.handle
- 主进程推送（告知状态）， webContents.send, ipcRenderer.on
- 渲染进程发起请求（申请控制）， ipcRenderer.send, ipcMain.on

2.5 傀儡端实现：捕获桌面视频流



MediaStream API

- 媒体内容的流
- 一个流对象可以包含多轨道，包括音频和视频轨道等
- 能通过 WebRTC 传输
- 通过 <Video> 标签可以播放



图片来自 WebRTC 官网：<https://webrtc.org/native-code/native-apis/>

如何捕获媒体流？

navigator.mediaDevices.getUserMedia (MediaStreamConstraints)

返回：Promise，成功后 resolve 回调一个 MediaStream 实例对象

参数：MediaStreamConstraints

- **audio: Boolean | MediaTrackConstraints**
- **video: Boolean | MediaTrackConstraints**
 - **width: 分辨率**
 - **height: 分辨率**
 - **frameRate: 帧率，比如 { ideal: 10, max: 15 }**
 - **...**

例子：捕获音视频媒体流

```
navigator.mediaDevices.getUserMedia({  
    audio: true,  
    video: {  
        width: { min: 1024, ideal: 1280, max: 1920 },  
        height: { min: 576, ideal: 720, max: 1080 },  
        frameRate: { max: 30 }  
    }  
})
```

如何播放媒体流对象

```
var video = document.querySelector('video')
video.srcObject = stream
video.onloadedmetadata = function(e) {
    video.play();
}
```

如何捕获桌面/窗口流

1. `desktopCapturer.getSources({ types: ['window', 'screen'] })` 提取 `chromeMediaSourceId`
 - Electron <5.0 是 callback 调用
 - 5.0后是 promise, 返回的是 `chromeMediaSources` 列表, 包含`id`,`name`,`thumbnail`,`display_id`
2. 通过 `navigator.webkitGetUserMedia({
 audio: false,
 video: {
 mandatory: {
 chromeMediaSource: 'desktop',
 chromeMediaSourceId: chromeMediaSourceId
 width,
 height
 }
}
})`

代码演示

2.6 傀儡端实现：如何响应指令

从一个数学爱情“故事”说起
心形线： $r=a(1-\sin\theta)$

robotjs 介绍

- 用于控制鼠标、键盘
- Node.js、C++、add-on库
- 支持Mac、Windows、Linux

安装和基本使用

- 安装：npm install robotjs
- 鼠标移动：robot.moveMouse(x, y)
- 鼠标点击：mouseClick([button], [double])
- 按键：robot.keyTap(key, [modifier])
- 详细文档：<https://robotjs.io/docs/syntax>

编译原生模块 (robotjs)

- 手动编译
 - npm rebuild —runtime=electron —disturl=https://atom.io/download/atom-shell \
—target=<electron版本> —abi=<对应版本abi>
 - process.versions.electron，可以看到electron版本
 - process.versions.node 可以看到 node 版本，之后再 abi_crosswalk 查找 abi
- electron-rebuild
 - npm install electron-rebuild —save-dev
 - npx electron-rebuild

代码演示

监听键盘+鼠标事件

- window.onkeydown
- window.onmouseup

响应键盘事件

- modifier (修饰键) 处理: shift、ctrl、alt、meta(win/command)
- 按键转换 (vkey)

响应点击事件

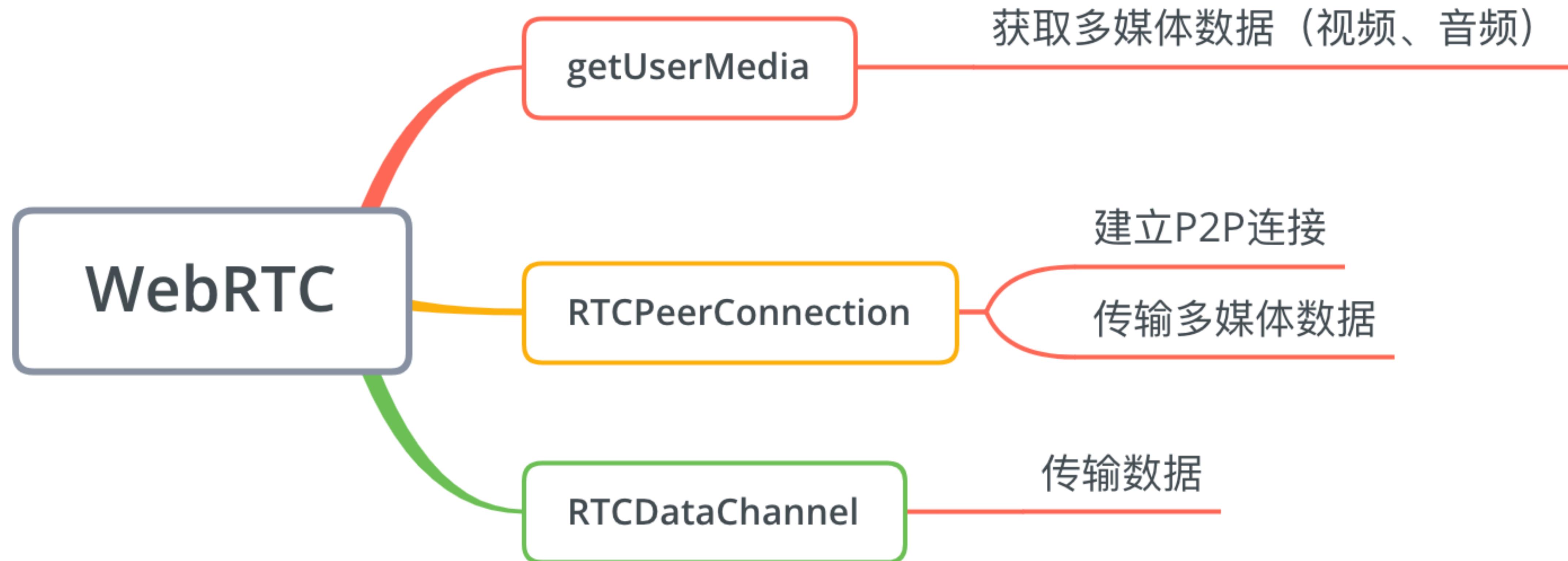
- 鼠标位置缩放（按比例）
 - $x' = x * \text{videoWidth} / \text{screenWidth}$
 - $y' = y * \text{videoHeight} / \text{screenHeight}$

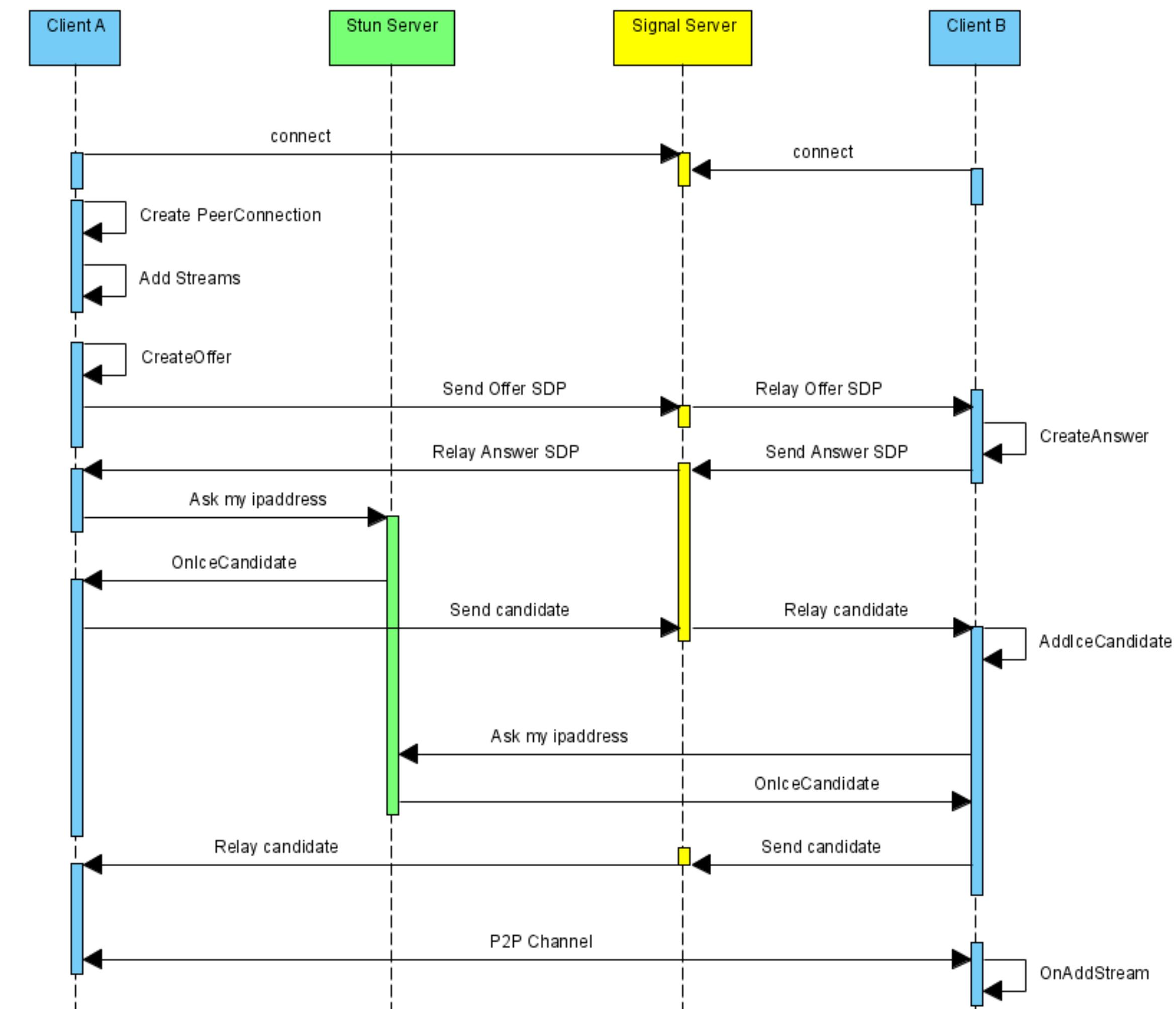
代码演示

小作业

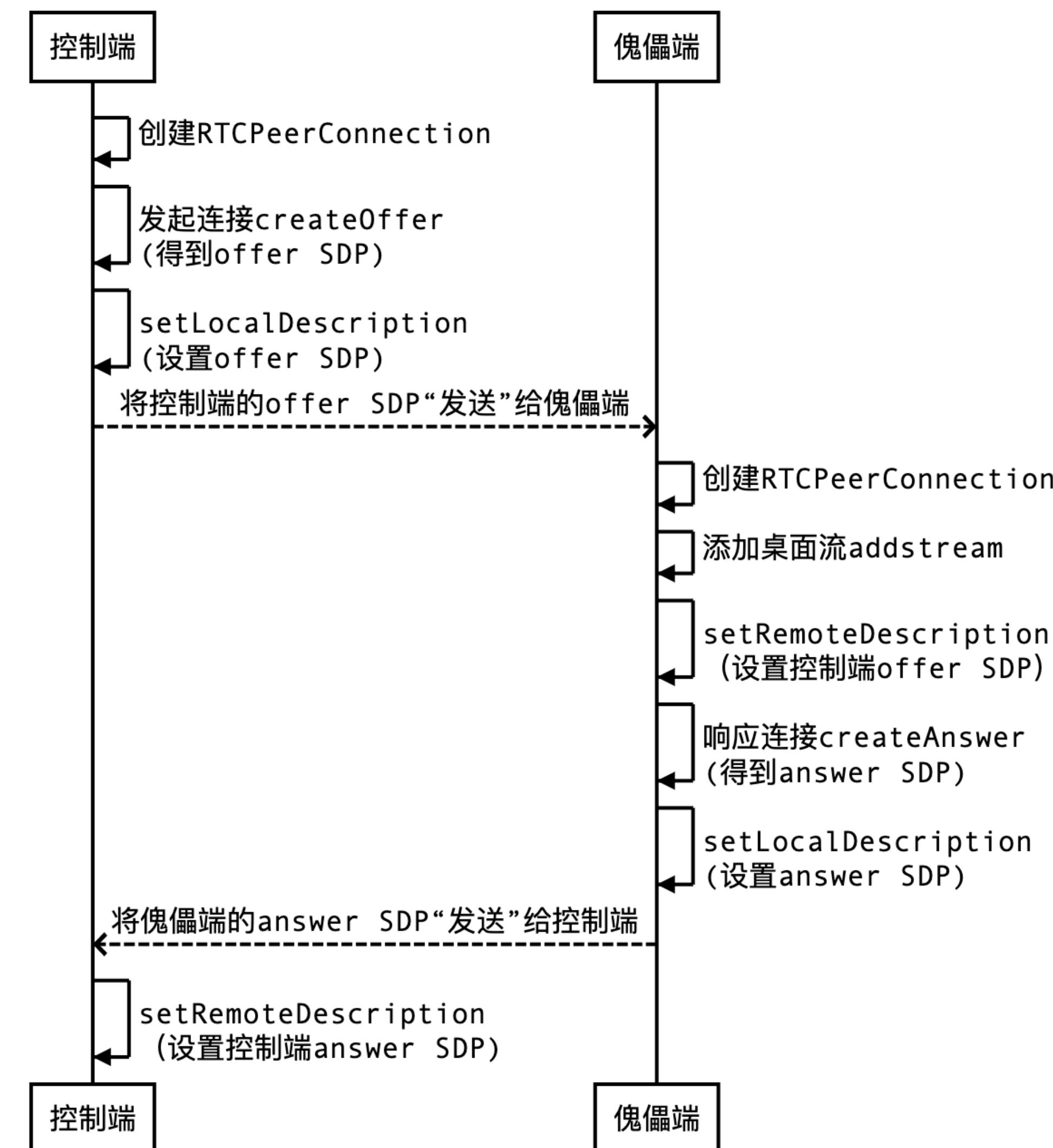
- 完善按键的响应
 - Enter、delete 等特殊字符
 - 利用 Electron globalShortcut 模块完成监听+响应打开控制台需求
 - Mac 打开控制台命令: CMD+OPTION+I
 - Windows 打开控制台命令: CTRL+SHIFT+I

2.7 视频传输：基于 WebRTC 传输视频流（上）





最简单的建立传输过程



SDP

- SDP (Session Description Protocol) 是一种会话描述协议，用来描述多媒体会话，主要用于协商双方通讯过程，传递基本信息。
- SDP的格式包含多行，每行为`<type>=<value>`
 - `<type>`: 字符，代表特定的属性，比如`v`，代表版本
 - `<value>`: 结构化文本，格式与属性类型有关，UTF8编码

SDP实例截取

v=0 // sdp版本号，一直为0, RFC4566规定

o=- 2588168131833388577 2 IN IP4 127.0.0.1

// RFC 4566 o=<username><sess-id><sess-version><nettype><addrtype><unicast-address>

t=0 0

//两个值分别是会话的起始时间和结束时间，这里都是0代表没有限制

m=video 9 UDP/TLS/RTP/SAVPF 96 97 98 99 100 101 102 122 127 121 125 107 108 109 124 120 123 119 114 115 116

// m=video说明本会话包含视频，9代表视频使用端口9来传输

// UDP, TLS, PTP代表使用UDP来传输RTP包，并使用TLS加密

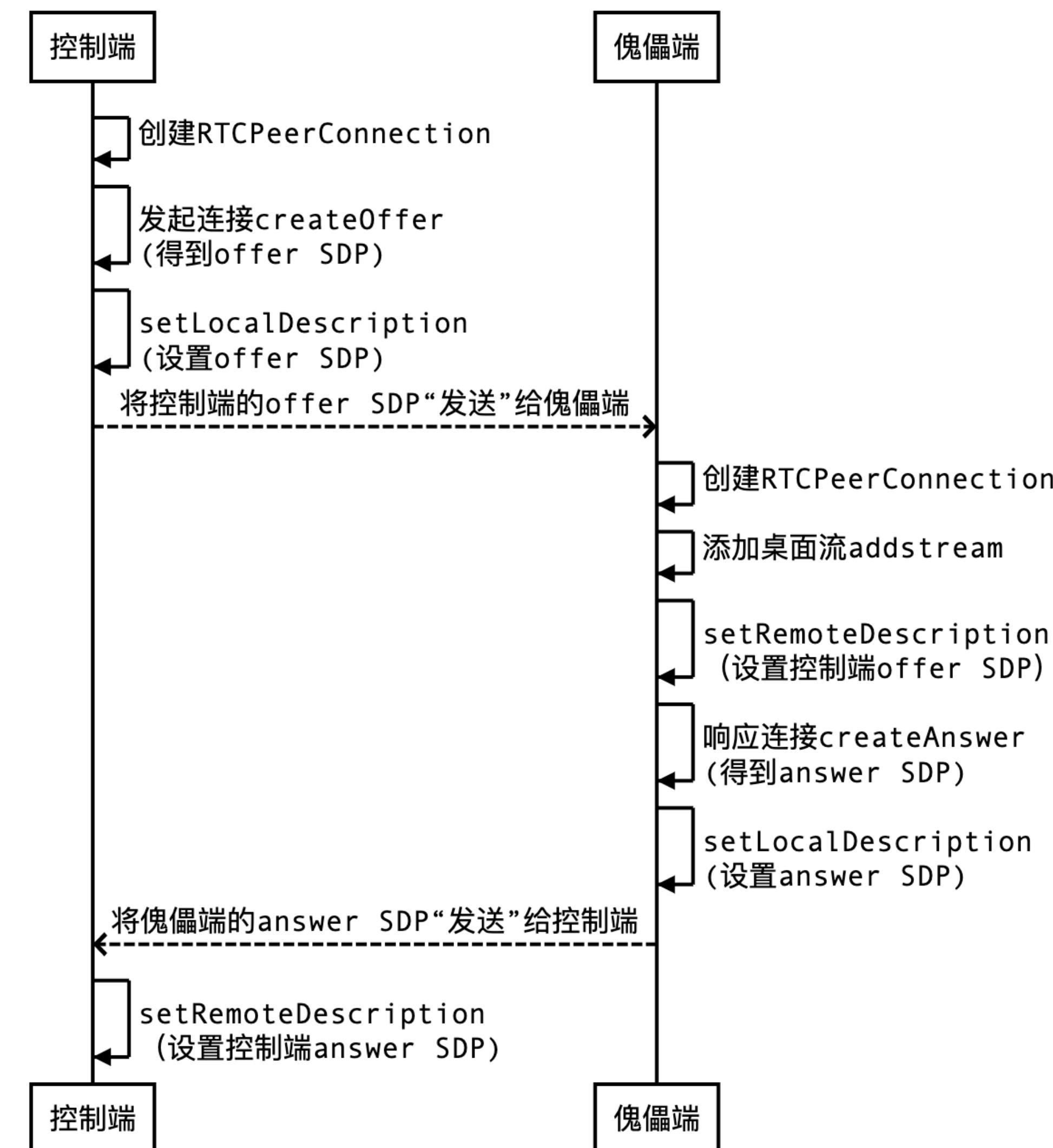
// SAVPF代表使用SRTCP

// 后面数字对应编码，比如video，122代表 H264/90000

a=msid-semantic: WMS qX0E8lLkHajwEZsXLKiu4FOIDOUqrqdqgEqER

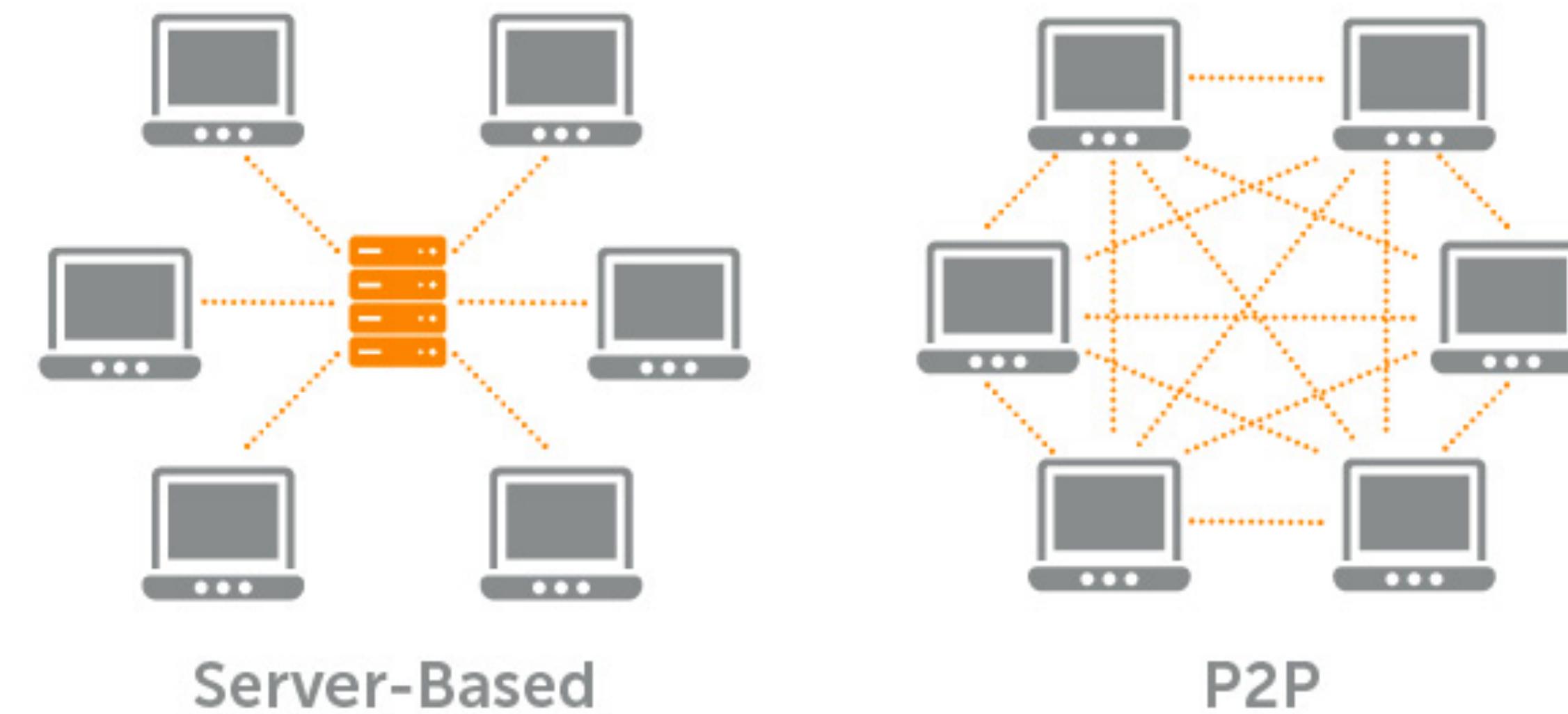
//WMS是WebRTC Media Stream简称，这一行对应的就是我们之前的media stream id

实战编码



2.8 视频传输：基于 WebRTC 传输视频流（下）

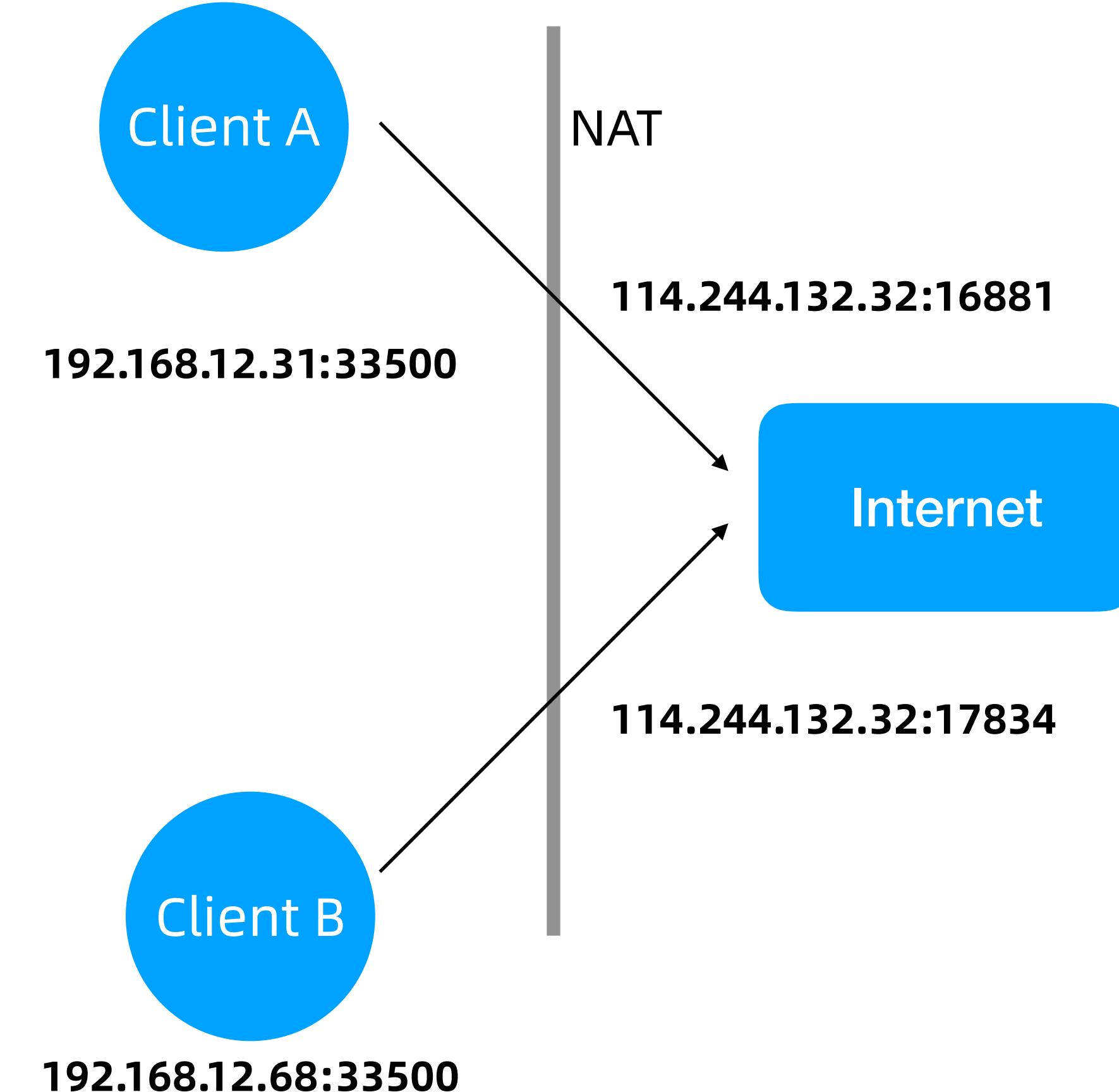
P2P



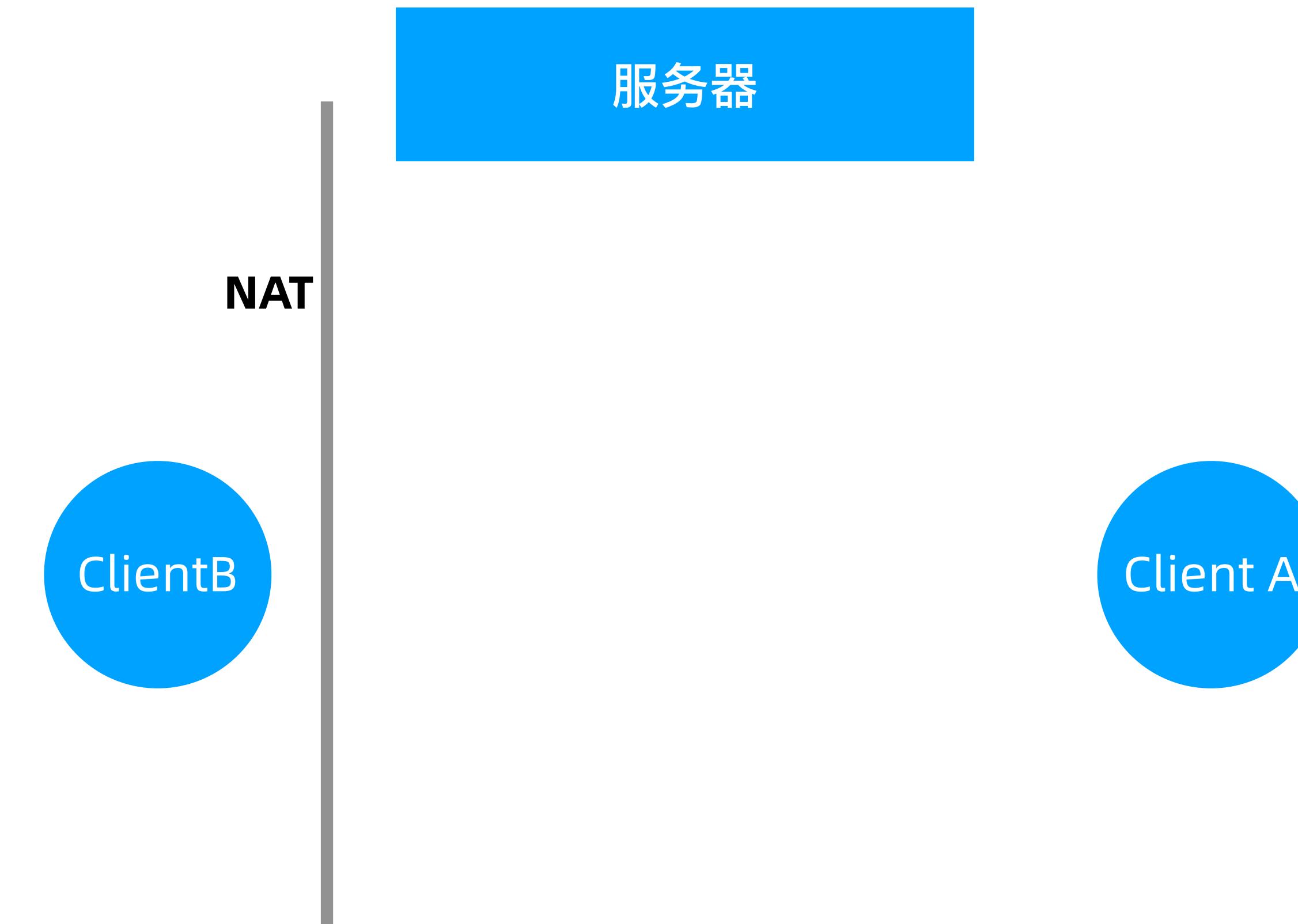
图片源自：<https://www.wowza.com/resources/guides/p2p-unicast-streaming>

P2P的难题

- NAT (Network Address Translation)网络地址转换
- 怎么获得真正的IP和端口？

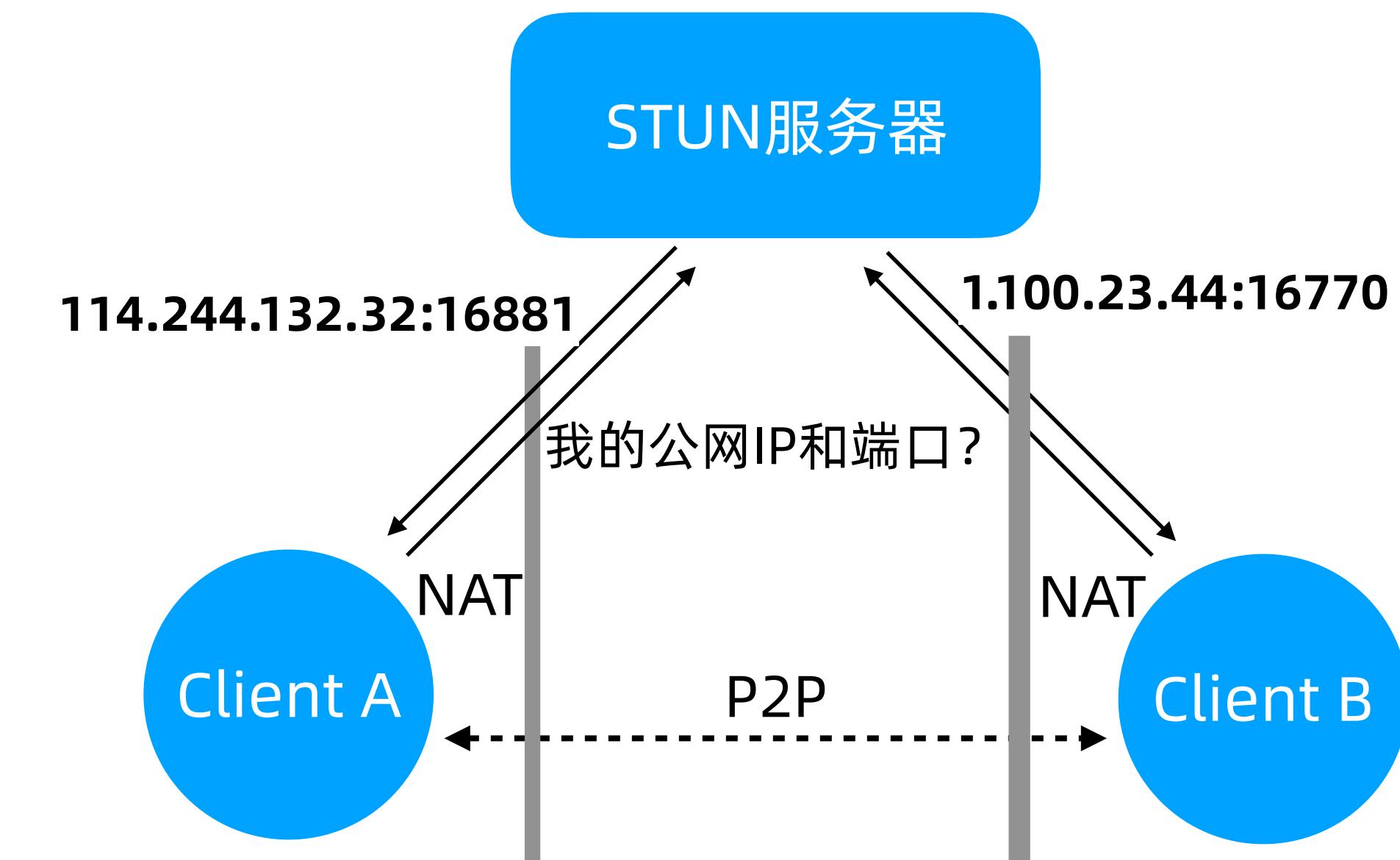


NAT打洞

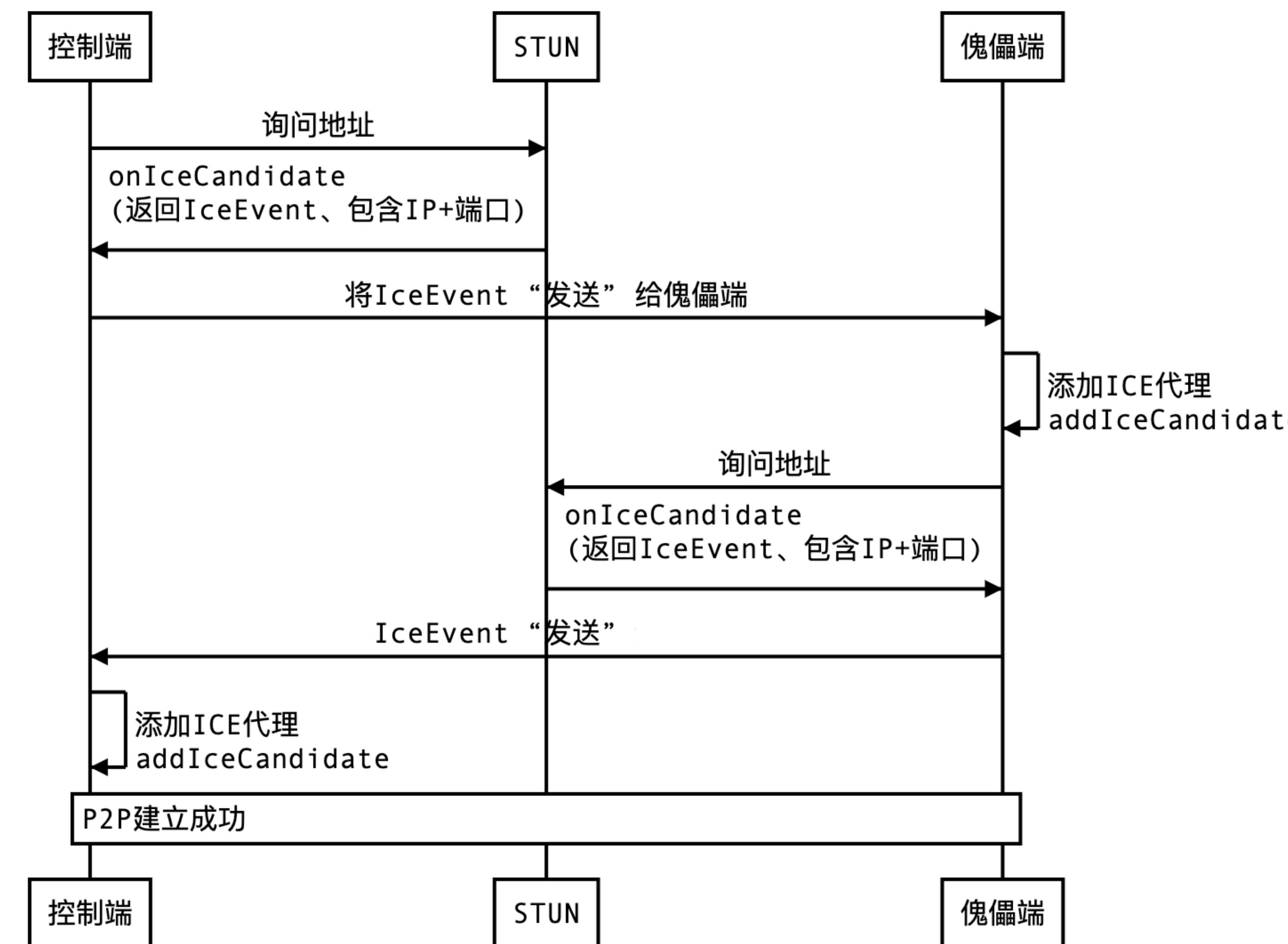


WebRTC NAT穿透：ICE

- ICE (Interactive Connectivity Establishment) 交互式连接创建
 - 优先STUN (Session Traversal Utilities for NAT), NAT会话穿越应用程序
 - 备选TURN (Traversal Using Relay NAT), 中继NAT实现的穿透
 - Full Cone NAT - 完全锥形NAT
 - Restricted Cone NAT - 限制锥形NAT
 - Port Restricted Cone NAT 端口限制锥形NAT
 - Symmetric NAT 对称NAT



STUN 过程



代码演示

延伸资料

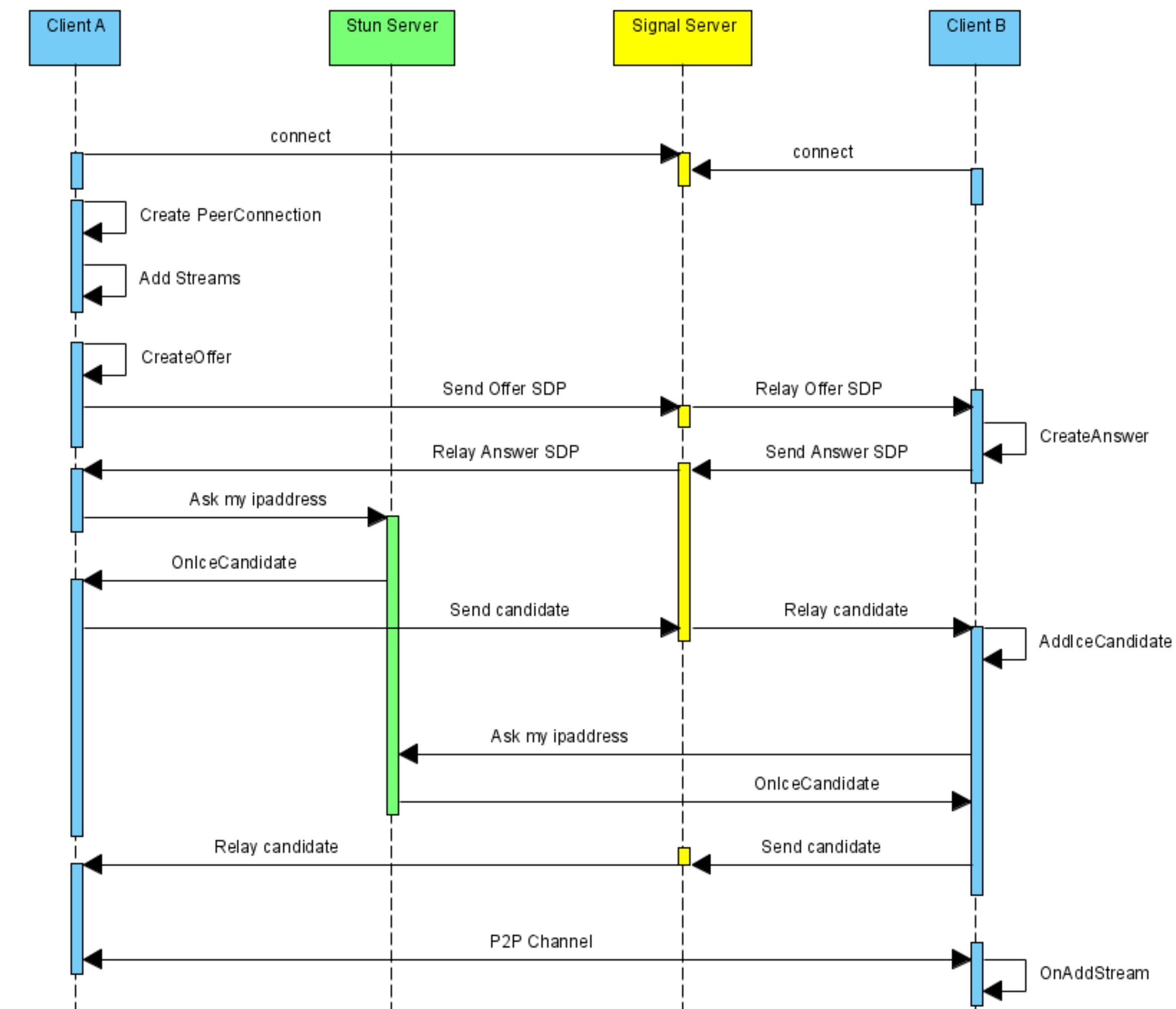
- <https://www.html5rocks.com/en/tutorials/webrtc/basics/>
- <https://cloud.tencent.com/developer/article/1005489>
- <https://url.cn/5lucoBr>

代码演示

2.9 信令服务：如何连接两端（上）

什么是信令服务？

- WebRTC 客户端（对等端）之间传递消息的服务器



服务端需求

- 处理业务逻辑
 - 建立端与控制码的联系
 - 通过控制码找到用户
- 转发 offer SDP、answer SDP、iceCandidate
 - 处理客户端请求
 - 主动推送消息给客户端

技术选型

	短轮询	长轮询	WebSocket 	sse
通讯方式	http	http	基于 TCP 长连接通讯	http
触发方式	轮询	轮询	事件	事件
优点	简单，兼容性好	相对短轮询资源占用少	全双工通讯，性能好，安全，扩展性	实现简单，开发成本低
缺点	安全性差，资源占用高	安全性差，资源占用高	传输数据需要进行二次解析，有一定开发门槛	适用于高级浏览器
适用范围	B/S服务	B/S服务	网游、支付、IM等	服务端到客户端推送 (如新消息推送)

服务端实现 WebSocket 服务器（基于 Node.js）

- npm install ws —save
- 基本使用

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 8010 });
wss.on('connection', function connection(ws, req) {
  ws.on('message', function incoming(msg) {
    // 响应客户端send事件
  })
  ws.on('close', function() {
    // 响应客户端close事件
  })
  ws.send('推送内容') // 发送内容到客户端
})
```

业务逻辑实现

- 处理业务逻辑
 - 建立端与控制码的联系
 - 通过控制码找到用户

测试 Websocket

- <https://www.websocket.org/echo.html>

转发需求

- 转发 offer SDP、answer SDP、iceCandidate
 - 处理客户端请求
 - 主动推送消息给客户端

代码演示

盘点埋下的「坑」

- 业务逻辑
 - ipcMain login
 - ipcMain control
- 信令逻辑
 - window.createAnswer
 - window.setRemote
 - window.addIceCandidate
 - robot

2.10 信令服务：如何连接两端（下）

浏览器使用 WebSocket

```
var ws = new WebSocket( "websocket地址" );

ws.onopen = function(e) {
    // websocket 成功建立回调
};

ws.onmessage = function(e) {
    // 收到服务端推送消息
    var data = event.data;
    console.log("Received Message: " + data);
};

ws.onclose = function(evt) {
    // 处理断开逻辑
};

ws.close() //主动断开
ws.send('hello') // 支持文本、blob、ArrayBuffer
```

客户端使用 WebSocket

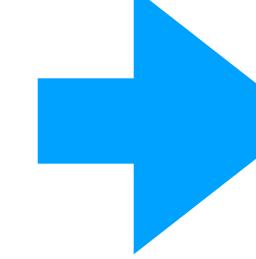
- npm install ws —save
- 基本使用

```
const WebSocket = require('ws')
const ws = new WebSocket(“websocket地址”);
```

盘点埋下的「坑」

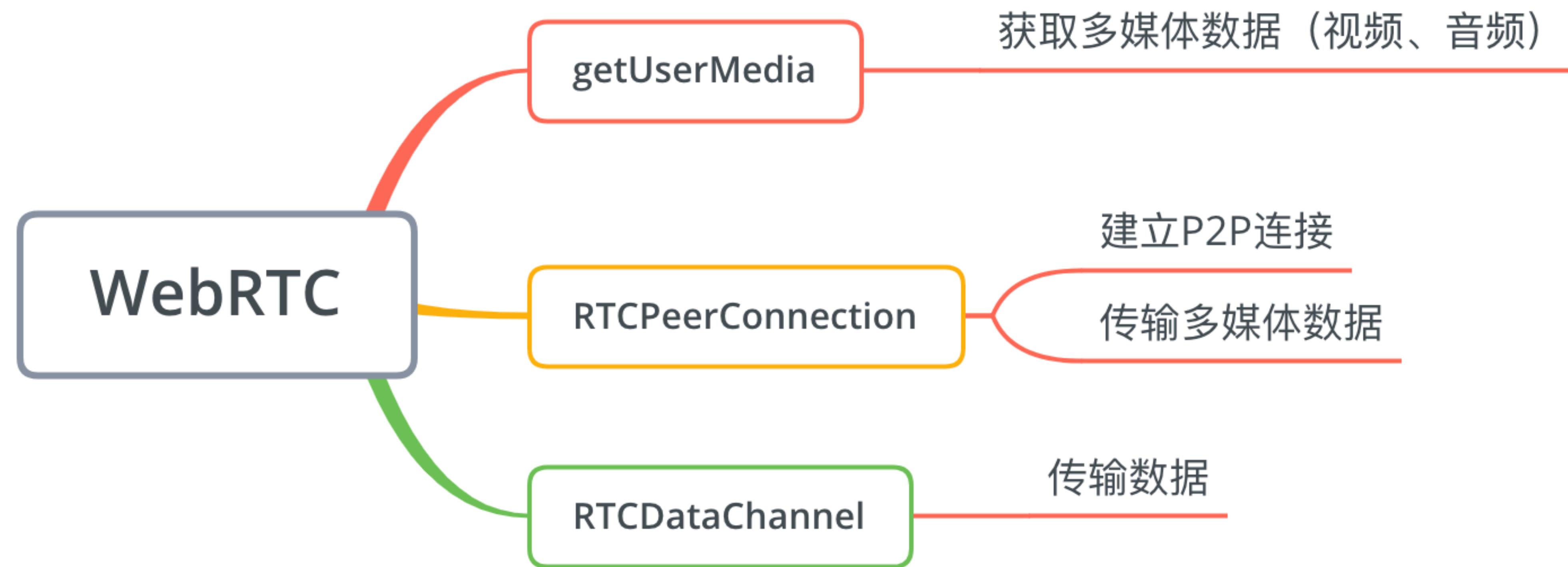
- 业务逻辑
 - ipcMain login
 - ipcMain control
- 信令逻辑
 - window.createAnswer
 - window.setRemote
 - window.addIceCandidate
 - robot

盘点埋下的「坑」

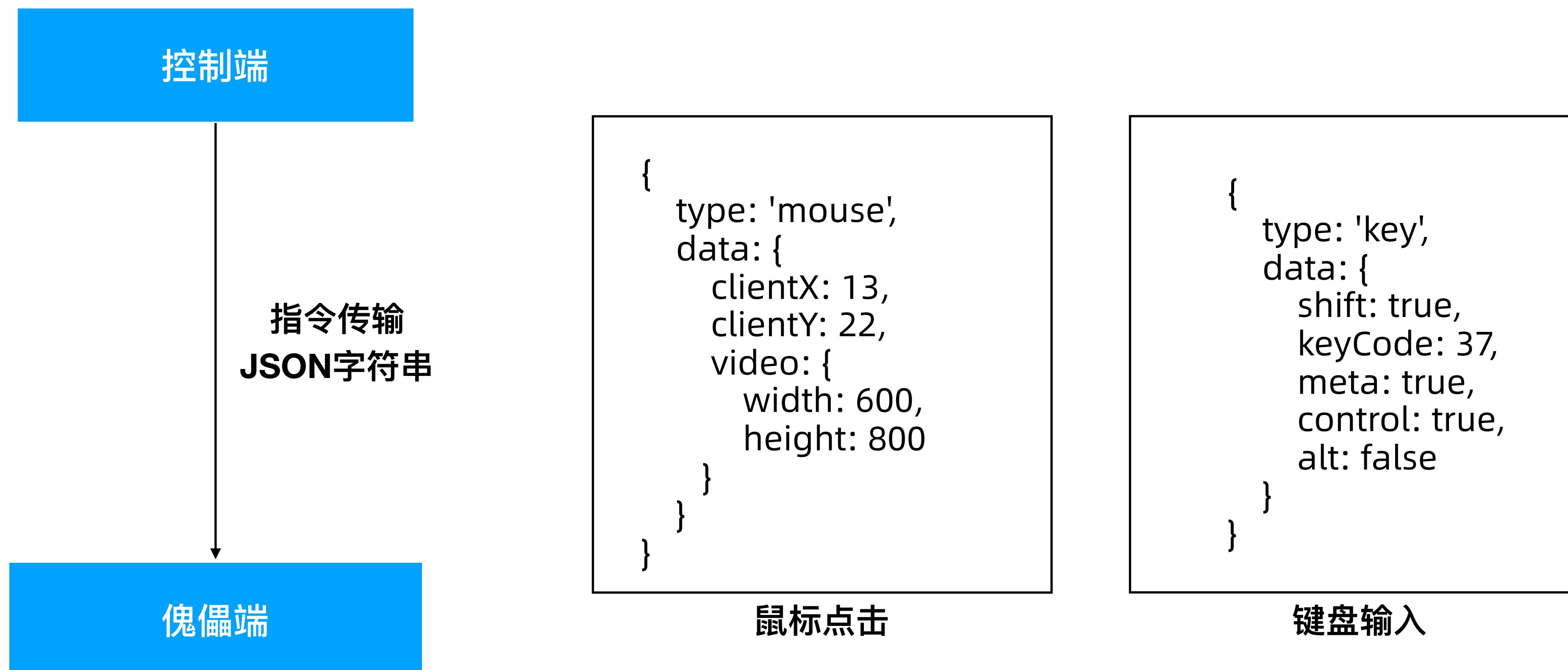
- 业务逻辑
 - ipcMain login signal.send('login') && signal.on('logined')
 - ipcMain control signal.on('controlled') && signal.on('be-controlled')
- 信令逻辑
 - window.createAnswer signal.send('forward') && signal.on('offer')
 - window.setRemote signal.send('forward') && signal.on('answer')
 - window.addIceCandidate signal.send('forward') && signal.send('iceCandidate')

代码演示

2.11 指令传输实现：如何建立数据传输



指令传输



传输方式

- 通过信令服务
- 基于 WebRTC 的 RTCDataChannel✓
 - 无服务端依赖，P2P 传输
 - 基于SCTP（传输层，有着 TCP、UDP 的优点）

基本用法

```
var pc = new RTCPeerConnection();

var dc = pc.createDataChannel("robotchannel"); // 创建一个 datachannel

dc.onmessage = function (event) { //接收消息
    console.log("received: " + event.data);
};

dc.onopen = function () { // 建立成功
    console.log("datachannel open");
};

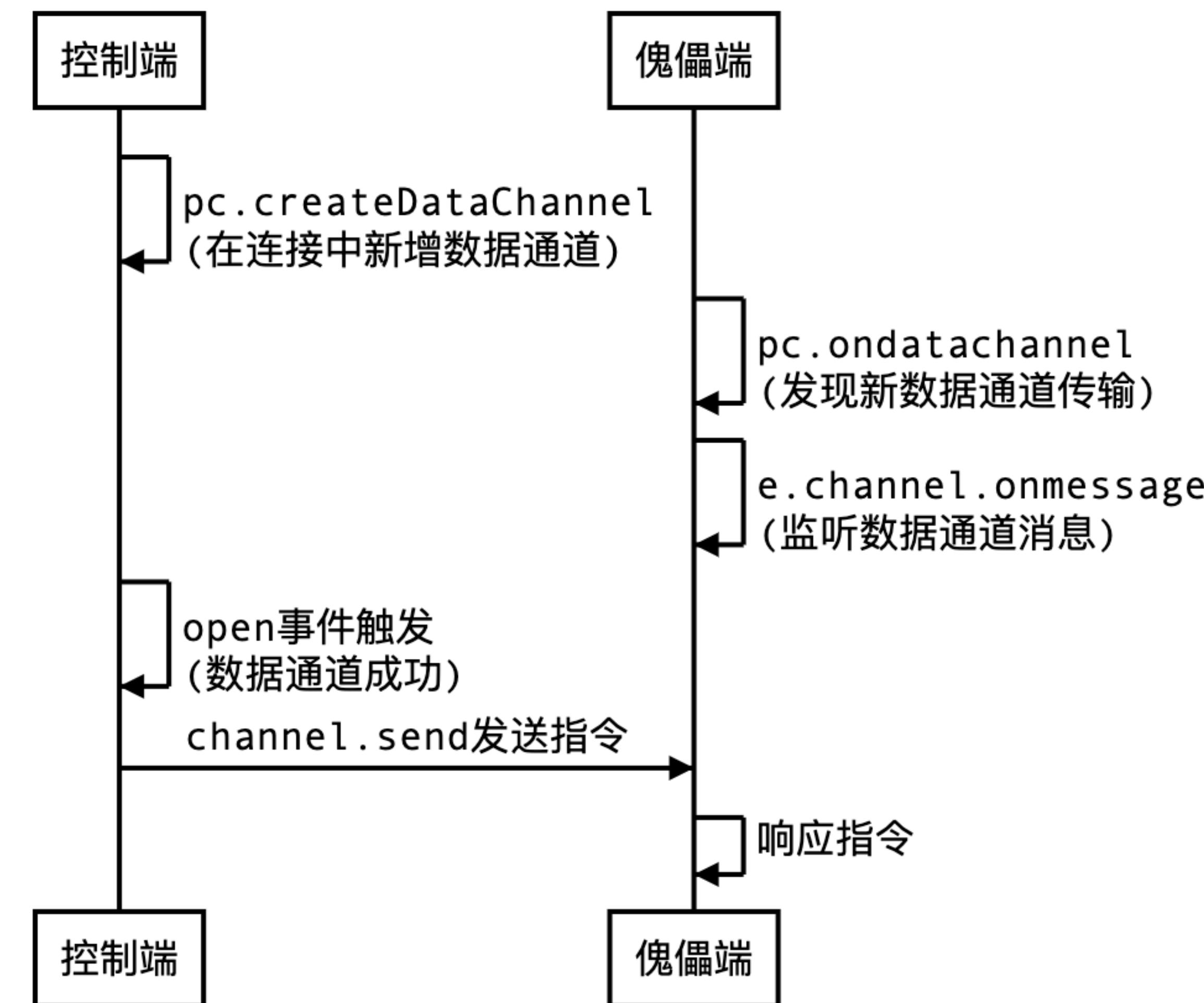
dc.onclose = function () { // 关闭
    console.log("datachannel close");
};

dc.send('text') // 发送消息

dc.close() // 关闭

pc.ondatachannel = function(e) {} // 发现新的datachannel
```

RTCDATAChannel 过程



盘点埋下的「坑」

- 业务逻辑
 - ipcMain login
 - ipcMain control
- 信令逻辑
 - window.createAnswer
 - window.setRemote
 - window.addIceCandidate
 - **robot**

代码演示

2.12 项目完善与总结（上）：App特性

项目完善（原生体验）

App特性

- 起名 + Logo
- 关于窗口
- 禁止多开
- 窗口假关闭

原生GUI

- 托盘（点击唤起）
- 顶部菜单
- 右键菜单（复制）

熵

Kubernetes

MySQL

MariaDB MaxDB

Nike

Hermès

起名字

Mercurius

寓意：传递信息的使者



Mercurius Logo



+



=



Mercurius Logo



+



=



关于窗口

图标 + 产品名字 + 版本号 + 版权 + ...



使用electron-about-window

```
1 const openAboutWindow = require('about-window').default;
2 const path = require('path')
3 const create = () => openAboutWindow({
4     icon_path: path.join(__dirname, 'icon.png'),
5     package_json_dir: path.resolve(__dirname + '../../..'),
6     copyright: 'Copyright (c) 2020 dragon',
7     homepage: 'https://github.com/dengyaolong/geektime-electron',
8     bug_report_url: 'https://github.com/dengyaolong/geektime-electron/issues',
9 })
10 module.exports = {create}
```

<https://github.com/rhysd/electron-about-window>

禁止多开

```
const gotTheLock = app.requestSingleInstanceLock()

if (!gotTheLock) {
  app.quit()
} else {
  app.on('second-instance', (event, commandLine, workingDirectory) => {
    // 当运行第二个实例时,将会聚焦到myWindow这个窗口
    showMainWindow()
  })
  app.on('ready', () => ...
})
}
```

窗口假关闭

- 用户点击窗口关闭按钮时候，应用只是隐藏
- 点击「退出应用」时才真正关闭窗口

代码演示-APP特性

2.13 项目完善与总结（下）：原生 GUI

Menu/MenuItem (菜单/菜单项)



● 新建菜单

```
const menu = new Menu()
```

● 新建菜单项

```
const menuItem1 = new MenuItem({ label: '复制', role: 'copy' })
const menuItem2 = new MenuItem({ label: '菜单项名', click: handler, enabled, visible,
  type: normal | separator | submenu | checkbox | radio,
  role: copy | paste | cut | quit | ...
})
```



● 添加菜单项

```
menu.append(menuItem1)
menu.append(new MenuItem({ type: 'separator' }))
menu.append(menuItem2)
```

● 弹出右键菜单

```
menu.popup({ window: remote.getCurrentWindow() })
```



● 设置应用菜单栏

```
app.applicationMenu = appMenu;
```



Tray (托盘)

方法

- **创建托盘**

```
tray = new Tray('/path/to/my/icon')
```

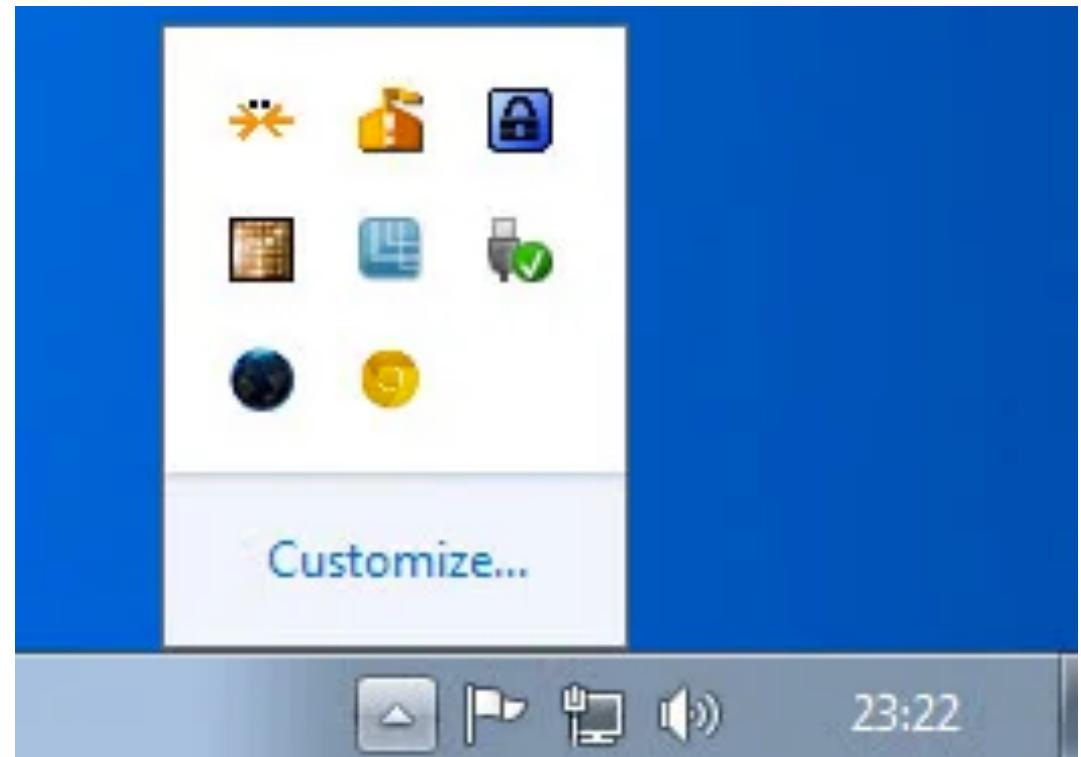
- Mac图片建议保留 1倍图 (32 * 32) , 2倍图@2x (64 * 64)
- Windows使用ico格式
- 大部分Mac托盘都是偏黑色的、Windows则是彩色的



Mac

- **弹出托盘菜单**

```
const contextMenu = Menu.buildFromTemplate([
  { label: '显示', click: () => {showMainWindow()}},
  { label: '退出', role: 'quit'}
])
tray.popUpContextMenu(contextMenu)
```



Windows

事件

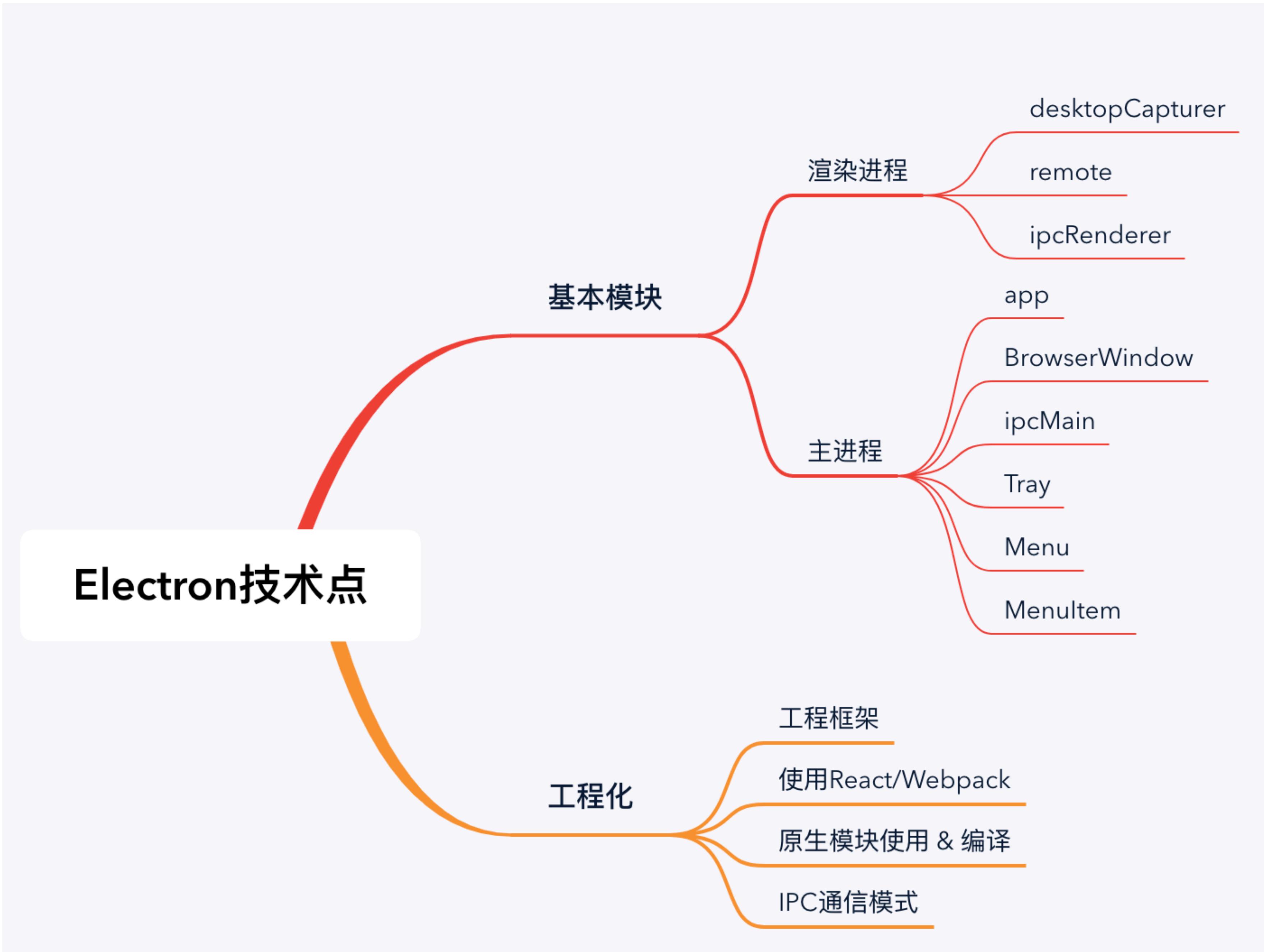
'click': 点击托盘

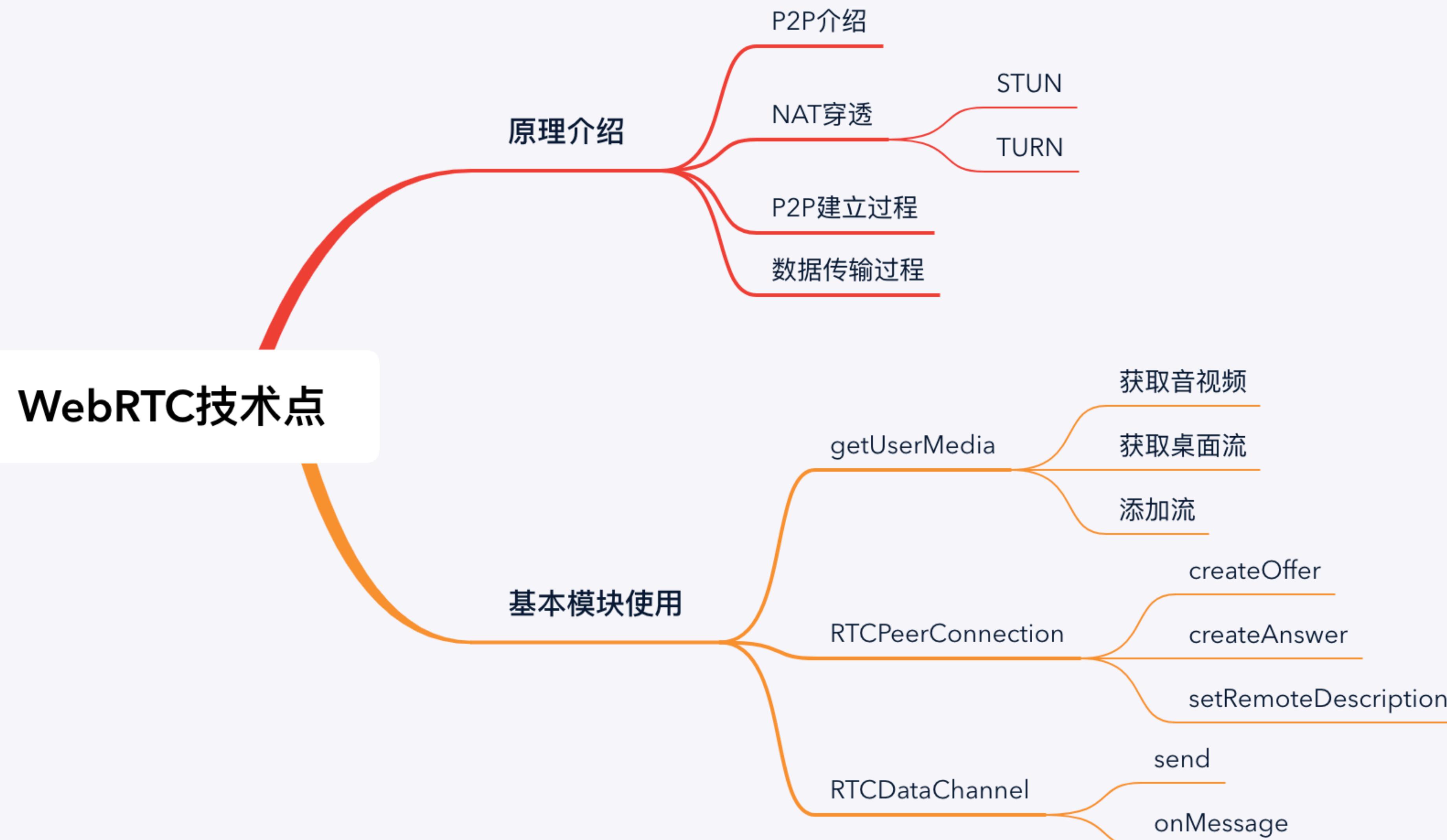
'right-click': 右击托盘

'drop-files': 文件拖拽。类似的还有drop-text

'balloon-click': 托盘气泡被点击 (Windows特性)

代码演示-GUI使用



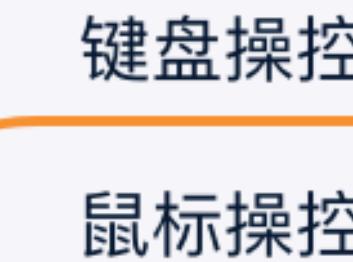


其他技术点

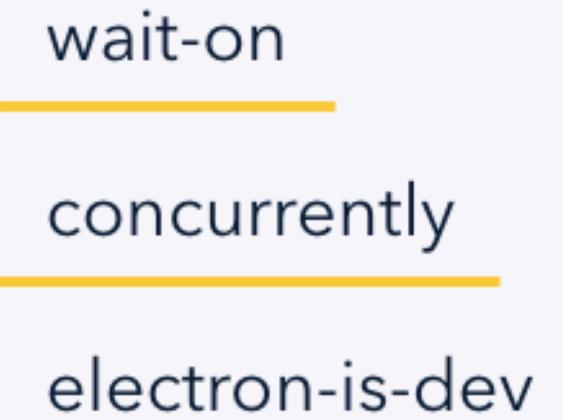
WebSocket



robotjs



开发常用库的使用





扫码试看/订阅

《Electron 开发实战》视频课程