



# 第三章 高级功能(上海-悠悠)

## 第三章 高级功能(上海-悠悠)

### 3.1 new\_context上下文与新窗口操作

- 3.1.1 浏览器上下文
- 3.1.2 Context实现测试浏览器环境隔离
- 3.1.3 多标签页

### 3.2 page.goto(url) 导航生命周期

- 3.2.1 导航生命周期
- 3.2.2 HTML DOM 事件
- 3.2.3 事件状态
- 3.2.4 使用示例
- 3.2.5 等待元素
- 3.2.6 timeout 等待超时

### 3.3 expect\_popup() 处理新标签窗口

- 3.3.1 context.expect\_page() 处理新标签页
- 3.3.2 处理弹出窗口

### 3.4 页面交互wait\_for\_load\_state

- 3.4.1 自定义等待
- 3.4.2 显示等待
- 3.4.3 多重导航
- 3.4.4 加载弹出窗口
- 3.4.5 timeout 等待超时

### 3.5 强大的录制视频功能

- 3.5.1 录制视频
- 3.5.2 指定视频宽高
- 3.5.3 获取保存视频路径
- 3.5.4 使用示例

### 3.6 强大的Trace Viewer 测试追踪功能

- 3.6.1 Trace Viewer 追踪功能
- 3.6.2 查看
- 3.6.3 使用示例
- 3.6.4 查看追踪

### 3.7 保存cookie, 解决反复登录的问题

- 3.7.1 登录认证
- 3.7.2 storageState保存登录cookies
- 3.7.3 高级场景
- 3.8 强大的Mock 接口返回，模拟各种异常场景
  - 3.8.1 模拟网络请求
  - 3.8.2 修改 API
  - 3.8.3 模拟登录时候，服务器异常的场景
- 3.9 模拟手机浏览器测试
  - 3.9.1 设置手机模式
  - 3.9.2 playwright.devices 可支持的手机列表
  - 3.9.3 pytest-playwright 测试用例
- 3.10 获取页面完整 HTML 内容
  - 3.10.1 page.content() 获取 html 内容
  - 3.10.2 inner\_html() 与 inner\_text()
- 3.11 text\_content 与 inner\_text 获取元素文本
  - 3.11.1 text\_content() 与 inner\_text() 获取页面文本
  - 3.11.2 all\_inner\_texts() 与 all\_text\_contents()
- 3.12 bounding\_box 获取元素坐标与宽高
  - 3.12.1 使用示例
- 3.13 mouse 鼠标操作总结
  - 3.13.1 page.mouse 使用
  - 3.13.2 click 点击
  - 3.13.3 dblclick 双击
  - 3.13.4 mouse.down 按住鼠标
  - 3.13.5 mouse.move 移动
  - 3.13.6 mouse.up 释放鼠标
  - 3.13.7 wheel 滚轮
- 3.14 鼠标操作- drag\_to 拖拽
  - 3.14.1 场景
  - 3.14.2 locator.drag\_to 拖拽操作
  - 3.14.3 page.drag\_and\_drop 拖动
  - 3.14.4 手动拖动
  - 3.14.5 示例代码
- 3.15 鼠标操作- 滚轮操作mouse.wheel
  - 3.15.1 鼠标滚轮操作

- 3.15.2 使用示例

### 3.16 Keyboard 键盘基本操作

- 3.16.1 键盘操作示例

- 3.16.2 down 向下

- 3.16.3 insert\_text 插入文本

- 3.16.4 press 按住

- 3.16.5 Type 操作

- 3.16.6 up 方法

- 3.16.7 使用示例

### 3.17 模拟键盘操作 - press复制粘贴相关

- 3.17.1 press()方法 介绍

- 3.17.2 使用实例

- 3.17.3 复制到本地

- 3.17.4 获取playwright 复制到剪切板内容

### 3.18 page.evaluate()执行JavaScript脚本

- 3.18.1 page.evaluate()

- 3.18.2 操作 web 网页示例

- 3.18.3 page.evaluate\_handle()返回JSHandle

### 3.19 locator.evaluate()对定位的元素执行JS脚本

- 3.19.1 locator.evaluate() 对元素执行JavaScript

- 3.19.2 locator.evaluate\_all() 执行全部元素

### 3.20 grant\_permissions 设置默认允许麦克风和摄像头等权限

- 3.20.1 权限框

- 3.20.2 解决办法

- [网易云视频课程](#)

## 3.1 new\_context上下文与新窗口操作

`browser.new_context()` 创建一个新的浏览器上下文。它不会与其他浏览器上下文共享 cookies/缓存。

### 3.1.1 浏览器上下文

使用`browser.new_context()` 创建context对象，context之间是相互隔离的，可以理解为轻量级的

浏览器实例。

如需要不同用户登录同一个网页，不需要创建多个浏览器实例，只需要创建多个context即可

以下是在一个浏览器实例上打开2个标签页

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False, slow_mo=1000)
    context = browser.new_context() # 创建上下文，浏览器实例
    page = context.new_page()      # 打开标签页1
    page.goto("https://www.baidu.com/")
    page2 = context.new_page()     # 打开标签页2
    page2.goto("https://www.cnblogs.com/yoyoketang/")

    context.close()
    browser.close()
```

也可以通过context 上下文操作多个浏览器实例，它不会与其他浏览器上下文共享 cookies/缓存，适用于多用户同时登陆的场景。

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False, slow_mo=1000)
    context1 = browser.new_context() # 创建上下文，浏览器实例1
    context2 = browser.new_context() # 创建上下文，浏览器实例2
    page1 = context1.new_page()      # 打开标签页1
    page1.goto("https://www.baidu.com/")

    # 操作第二个浏览器窗口
    page2 = context2.new_page()     # 打开标签页1
    page2.goto("https://www.baidu.com/")
```

运行后会发现打开了2个浏览器窗口

### 3.1.2 Context实现测试浏览器环境隔离

使用 Playwright 编写的测试在称为浏览器上下文的隔离的全新环境中执行。这种隔离模型提高了可重复性并防止级联测试失败。

## 什么是测试隔离

测试隔离是指每个测试与另一个测试完全隔离。每个测试都独立于任何其他测试运行。这意味着每个测试都有自己的本地存储、会话存储、cookie 等。Playwright 使用BrowserContext实现了这一点，这相当于隐身式配置文件。它们的创建速度快、成本低，并且完全隔离，即使在单个浏览器中运行也是如此。Playwright 为每个测试创建一个上下文，并在该上下文中提供一个默认页面。

## 为什么测试隔离很重要

- 没有失败结转。如果一个测试失败，它不会影响另一个测试。
- 易于调试错误或不稳定，因为您可以根据需要多次运行单个测试。
- 并行运行、分片等时不必考虑顺序。

测试隔离有两种不同的策略：从头开始或在两者之间进行清理。在测试之间清理的问题是很容易忘记清理，有些东西是不可能清理的，比如“访问过的链接”。来自一个测试的状态可能会泄漏到下一个测试中，这可能会导致您的测试失败并使调试变得更加困难，因为问题来自另一个测试。从头开始意味着一切都是新的，因此如果测试失败，您只需查看该测试即可进行调试。

## Playwright 如何实现测试隔离

Playwright 使用浏览器上下文来实现测试隔离。每个测试都有自己的浏览器上下文。每次运行测试都会创建一个新的浏览器上下文。使用 Playwright 作为测试运行程序时，默认情况下会创建浏览器上下文。否则，您可以手动创建浏览器上下文。

```
browser = playwright.chromium.launch()
context = browser.new_context()
page = context.new_page()
```

浏览器上下文还可用于模拟涉及移动设备、权限、区域设置和配色方案的多页面场景

Playwright 可以在一个场景中创建多个浏览器上下文。当您想测试多用户功能（如聊天）时，这很有用。

```

from playwright.sync_api import sync_playwright

def run(playwright):
    # create a chromium browser instance
    chromium = playwright.chromium
    browser = chromium.launch()

    # create two isolated browser contexts
    user_context = browser.new_context()
    admin_context = browser.new_context()

    # create pages and interact with contexts independently

    with sync_playwright() as playwright:
        run(playwright)

```

### 3.1.3 多标签页

每个浏览器上下文可以承载多个页面（选项卡）。

- 每个页面都像一个聚焦的活动页面。不需要将页面置于最前面。
- 上下文中的页面遵循上下文级别的模拟，例如视口大小、自定义网络路由或浏览器区域设置。

```

# create two pages
page_one = context.new_page()
page_two = context.new_page()

# get pages of a browser context
all_pages = context.pages

```

使用示例，在page\_one 标签页打开百度，输入“上海-悠悠”， 在page\_two 标签页打开百度，输入“yoyoketang”

```
from playwright.sync_api import sync_playwright
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False, slow_mo=1000)
    context = browser.new_context() # 创建上下文, 浏览器实例1

    page1 = context.new_page() # 打开标签页1
    page2 = context.new_page() # 打开标签页2
    page1.goto("https://www.baidu.com/")
    page1.fill('#kw', "上海-悠悠")
    page1.wait_for_timeout(3000)

    page2.goto("https://www.baidu.com/")
    page2.fill('#kw', "yoyoketang")
    page2.wait_for_timeout(3000)
```

虽然你看不到第一个页面的操作，实际上它已经操作了，每个页面 page 对象都是聚焦的活动页面，不需要将页面置于最前面。

## 3.2 page.goto(url) 导航生命周期

Playwright 可以导航到 URL 并处理由页面交互引起的导航。本篇涵盖了等待页面导航和加载完成的常见场景。

### 3.2.1 导航生命周期

导航从更改页面 URL 或通过与页面交互（例如，单击链接）开始。导航意图可能会被取消，例如，在点击未解析的 DNS 地址或转换为文件下载时。

解析响应标头并更新会话历史记录后，将提交导航。只有在导航成功（提交）后，页面才会开始加载文档。

加载包括通过网络获取剩余的响应主体、解析、执行脚本和触发加载事件：

调用 page.goto(url) 后页面加载过程：

- page.url 设定新的 url
- document 文档内容通过网络加载并解析
- page.on("domcontentloaded") 事件触发
- 执行页面的 js 脚本，页面执行一些脚本并加载 css 和图像等资源

- `page.on("load")` 事件触发
- 页面执行动态加载的脚本
- `networkidle` 当 500 毫秒内没有新的网络请求时触发

## 3.2.2 HTML DOM 事件

HTML DOM 事件允许Javascript在HTML文档元素中注册不同事件处理程序。

事件名称	作用
onload	通常用于 <code>&lt;body&gt;</code> 元素，在页面完全载入后(包括图片、css文件等等。)执行脚本代码。
onunload	用户退出页面。( <code>&lt;body&gt;</code> 和 <code>&lt;frameset&gt;</code> )
onclick	当用户点击某个对象时调用的事件句柄。
onmouseover	鼠标移到某元素之上。
onmouseout	鼠标从某元素移开。
ondblclick	当用户双击某个对象时调用的事件句柄。
onkeydown	某个键盘按键被按下。
onkeypress	某个键盘按键被按下并松开。
onkeyup	某个键盘按键被松开。
onfocus	元素获取焦点时触发
onblur	元素失去焦点时触发
onchange	该事件在表单元素的内容改变时触发 <code>&lt;input&gt;</code> , <code>&lt;keygen&gt;</code> , <code>&lt;select&gt;</code> , 和 <code>&lt;textarea&gt;</code>
onfocus	元素获取焦点时触发
onsubmit	表单提交时触发

## 3.2.3 事件状态

导航到 URL 会自动等待页面触发事件 `load` 。如果页面之前进行了客户端重定向 `load` , `page.goto()`将自动等待重定向页面触发事件`load`。



从源码可以看到 `wait_until` 等待的事件可以支持`["commit", "domcontentloaded", "load", "networkidle"]` 四个参数，默认是等待 `load` 触发。

`wait_until:Union["commit", "domcontentloaded", "load", "networkidle", None]`当认为操作成功时，默认为“加载”。事件可以是：

- `commit`：考虑在接收到网络响应并且文档开始加载时完成操作。
- `domcontentloaded`：在触发“domcontentloaded”事件时完成操作。
- `load`：触发 `load`事件时完成操作。
- `networkidle`：当 500 毫秒内没有新的网络请求时触发，认为操作已完成。

```
def goto(
    self,
    url: str,
    *,
    timeout: typing.Optional[float] = None,
    wait_until: typing.Optional[
        Literal["commit", "domcontentloaded", "load", "networkidle"]
    ] = None,
    referer: typing.Optional[str] = None
) -> typing.Optional["Response"]:
```

如果我们希望ajax 也请求完成了，再继续下一步，那么可以覆盖默认行为以等待特定事件，例如 `networkidle`。

(对于 `click`、`fill` 等操作会自动等待元素出现。)

使用语法

```
# Navigate and wait until network is idle
page.goto("https://example.com", wait_until="networkidle")
```

## 3.2.4 使用示例

当我们打开一个国外的网站，访问会很慢时（有些js/css资源加载很慢很慢）

```

from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False, slow_mo=1000)
    context = browser.new_context(accept_downloads=True)
    page = context.new_page()
    page.goto("https://playwright.dev/")

    page.get_by_role("link", name="Community").click()

    page.pause()
    browser.close()

```

如果是按默认的等待 load 事件，会出现报错 `waiting until "load"`

```

    result = next(iter(done)).result()
playwright._impl._api_types.TimeoutError: Timeout 30000ms exceeded.
===== logs =====
navigating to "https://playwright.dev/", waiting until "load"
=====

```

其实页面元素在 `domcontentloaded` 阶段就已经加载到元素了，那么不用等待到load

```

from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False, slow_mo=1000)
    context = browser.new_context(accept_downloads=True)
    page = context.new_page()
    page.goto("https://playwright.dev/", wait_until="domcontentloaded")

    page.get_by_role("link", name="Community").click()

    page.pause()
    browser.close()

```

对于有些元素可能需要等待到 `networkidle` 状态才会出来，那么就需要使用

```

page.goto("xxx", wait_until="networkidle")

```

## 3.2.5 等待元素

在延迟加载的页面中，使用locator.wait\_for()等待元素可见是很有用的。或者，像page.click()这样的页面交互会自动等待元素。

```
# Navigate and wait for element
page.goto("https://example.com")
page.get_by_text("example domain").wait_for()

# Navigate and click element
# Click will auto-wait for the element
page.goto("https://example.com")
page.get_by_text("example domain").click()
```

## 3.2.6 timeout 等待超时

timeout 参数可以设置页面加载超时时间, 默认是30秒, 传递“0”以禁用超时。

```
timeout : Union[float, None]
    Maximum operation time in milliseconds, defaults to 30 seconds, pass `0` to disable timeout.
    The timeout can be changed by using the `browser_context.set_default_navigation_timeout()`,
    `browser_context.set_default_timeout()`, `page.set_default_navigation_timeout()` or
    `page.set_default_timeout()` methods.
```

更改默认值可以使用 以下方法更改

- browser\_context.set\_fault\_navigation\_timeout () 进行更改,
- browser\_context.set\_fault\_timeout () ,
- page.set\_fault\_navigation\_timeout)
- page.set\_fault\_timeout ()

## 3.3 expect\_popup() 处理新标签窗口

当我们点击一个带有 target="\_blank" 的链接时，会打开一个新的标签页，如果想继续在新的标签页上继续操作，那么有2种方式捕获到新页面的page对象

- 1. context.expect\_page() 获取新标签页对象
- 2. page.expect\_popup() 获取新标签页对象

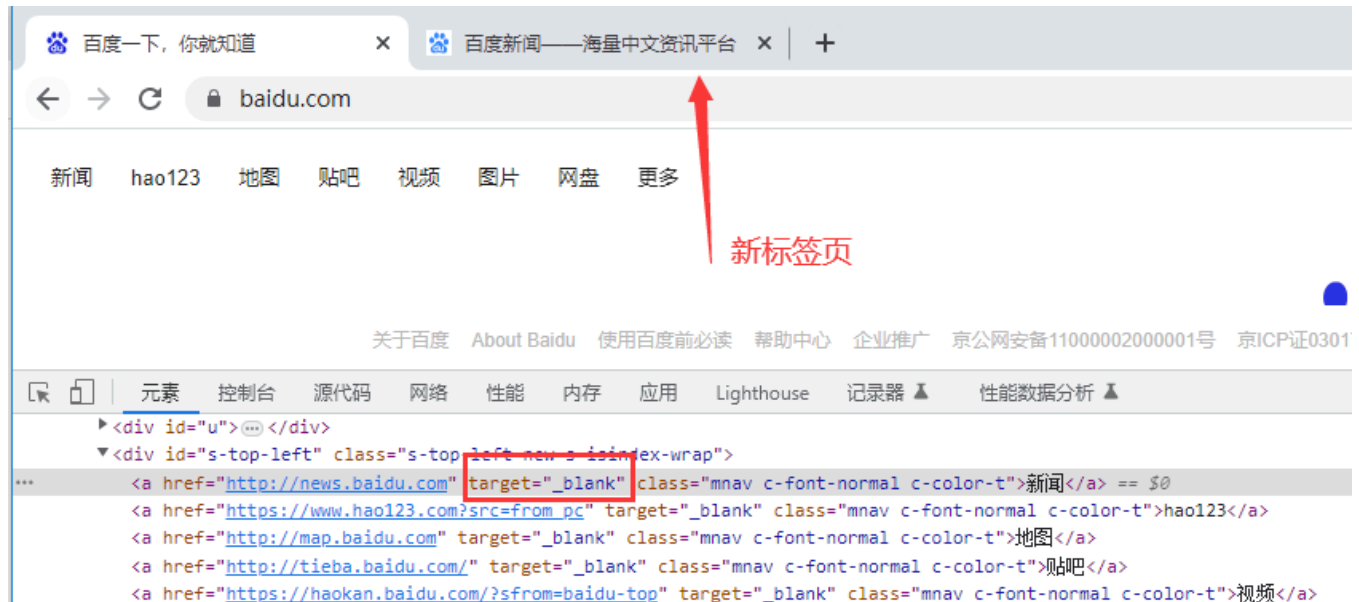
### 3.3.1 context.expect\_page() 处理新标签页

浏览器上下文中的事件page可用于获取在上下文中创建的新页面。这可用于处理通过 `target="_blank"` 链接打开的新页面。

```
# Get page after a specific action (e.g. clicking a link)
with context.expect_page() as new_page_info:
    page.get_by_text("open new tab").click() # Opens a new tab
new_page = new_page_info.value

new_page.wait_for_load_state()
print(new_page.title())
```

使用示例，打开百度页面的-新闻链接，会出现一个新标签页



```

from playwright.sync_api import sync_playwright
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False, slow_mo=1000)
    context = browser.new_context() # 创建上下文, 浏览器实例

    page = context.new_page() # 打开标签页
    page.goto("https://www.baidu.com/")
    print(page.title())
    # Get page after a specific action (e.g. clicking a link)
    with context.expect_page() as new_page_info:
        page.click('text=新闻') # Opens a new tab
    new_page = new_page_info.value

    new_page.wait_for_load_state() # 等待页面加载到指定状态
    print(new_page.title())

```

## 运行结果

百度一下, 你就知道  
 百度新闻—海量中文资讯平台

## 监听 context 的 page 事件

```

# Get all new pages (including popups) in the context
def handle_page(page):
    page.wait_for_load_state()
    print(page.title())

context.on("page", handle_page)

```

## 3.3.2 处理弹出窗口

如果页面打开一个弹出窗口（例如通过链接打开的页面），您可以通过监听页面上的事件 `target="_blank"` 来获取对它的引用。popup

除了 `browserContext.on('page')` 事件之外还会发出此事件，但仅针对与此页面相关的弹出窗口。

```
# Get popup after a specific action (e.g., click)
with page.expect_popup() as popup_info:
    page.get_by_text("open the popup").click()
    popup = popup_info.value

popup.wait_for_load_state()
print(popup.title())
```

## 监听page的popup事件

```
# Get all popups when they open
def handle_popup(popup):
    popup.wait_for_load_state()
    print(popup.title())

page.on("popup", handle_popup)
```

## 3.4 页面交互wait\_for\_load\_state

从A页面点击打开B页面，存在2种情况

- 1.带有 `target="_blank"` 的链接时，会打开一个新的标签页
- 2.直接在当前页面刷新另外一个地址

当打开一个新的页面后，可以使用 `wait_for_load_state` 等待页面加载到指定状态  
等待的事件可以支持["commit", "domcontentloaded", "load", "networkidle"] 四个参数

### 3.4.1 自定义等待

`locator.click`可以结合`page.wait_for_load_state()`来等待加载事件。

```
page.locator("button").click() # Click triggers navigation
page.wait_for_load_state("networkidle") # This waits for the "networkidle"
```

### 3.4.2 显示等待

显式调用`page.expect_navigation()` 等待新的页面加载

如果没有等待到新的导航页面，就会报超时异常。

```
# Waits for the next navigation. Using Python context manager
# prevents a race condition between clicking and waiting for a navigation.
with page.expect_navigation():
    # Triggers a navigation after a timeout
    page.get_by_text("Navigate after timeout").click()
```

### 3.4.3 多重导航

点击一个元素可能会触发多个导航。在这些情况下，建议将 `page.expect_navigation()` 显式指向特定的 url。

```
# Using Python context manager prevents a race condition
# between clicking and waiting for a navigation.
with page.expect_navigation(url="**/login"):
    # Triggers a navigation with a script redirect
    page.get_by_text("Click me").click()
```

### 3.4.4 加载弹出窗口

打开弹出窗口时，显式调用 `page.wait_for_load_state()` 可确保将弹出窗口加载到所需状态。

```
with page.expect_popup() as popup_info:
    page.get_by_text("Open popup").click() # Opens popup
popup = popup_info.value
popup.wait_for_load_state("load")
```

### 3.4.5 timeout 等待超时

`timeout` 参数可以设置页面加载超时时间，默认是30秒，传递“0”以禁用超时。

```
timeout : Union[float, None]
    Maximum operation time in milliseconds, defaults to 30 seconds, pass `0` to disable timeout. The default
    be changed by using the `browser_context.set_default_navigation_timeout()`,
    `browser_context.set_default_timeout()`, `page.set_default_navigation_timeout()` or
    `page.set_default_timeout()` methods.
```

更改默认值可以使用以下方法更改

- `browser_context.set_fault_navigation_timeout()` 进行更改,

- `browser_context.set_fault_timeout ()` ,
- `page.set_fault_navigation_timeout()`
- `page.set_fault_timeout ()`

## 3.5 强大的录制视频功能

使用 Playwright，您可以为测试录制视频。

### 3.5.1 录制视频

视频在浏览器上下文关闭时保存。如果您手动创建浏览器上下文，请确保 `browser_context.close()`，会在调用close的时候保存视频。

```
context = browser.new_context(record_video_dir="videos/")

# 确保调用 close， videos视频才会保存
context.close()
```

执行完成后，会保存到videos/目录下

名称	日期	类型	大小	时长
 652e10d397b491...	2023/2/27 13:40	WEBM 文件	11 KB	00:00:01
 532750b41682f48...	2023/2/27 13:42	WEBM 文件	471 KB	00:00:41

视频录制40秒左右，也才400k大小，一般用户3-5秒可以执行完，所以完全不用担心视频占用太多的空间。

### 3.5.2 指定视频宽高

您还可以指定视频大小。视频大小默认为缩小以适合 800x800 的视口大小。

视口的视频放置在输出视频的左上角，必要时按比例缩小以适合。您可能需要设置视口大小以匹配您想要的视频大小。



```
context = browser.new_context(  
    record_video_dir="videos/",  
    record_video_size={"width": 640, "height": 480}  
)
```

### 3.5.3 获取保存视频路径

保存的视频文件将出现在指定的文件夹中。它们都生成了唯一的名称。对于多页面场景，可以通过 `page.video` 访问页面关联的视频文件。

```
path = page.video.path()  
print(path) # videos\b880519a32528f80d64a2cb6769f2162.webm
```

### 3.5.4 使用示例

```
from playwright.sync_api import sync_playwright  
  
with sync_playwright() as p:  
    browser = p.chromium.launch(headless=False, slow_mo=1000)  
    context = browser.new_context(  
        record_video_dir="videos/",  
        record_video_size={"width": 640, "height": 480}  
    )  
    page = context.new_page()  
    page.goto("http://47.108.155.10/login.html")  
    page.get_by_placeholder("请输入用户名").fill("test")  
    page.get_by_placeholder("请输入密码").fill("123456")  
    page.get_by_role("button", name="立即登录 >").click()  
  
    path = page.video.path()  
    print(path) # videos\b880519a32528f80d64a2cb6769f2162.webm  
  
    # 确保调用 close, videos视频才会保存  
    context.close()  
    browser.close()
```

## 3.6 强大的Trace Viewer 测试追踪功能

在执行自动化用例的过程中，出现一些不稳定偶然性的bug，需要复现bug，还原bug出现的过

程。于是需要追踪用例执行的过程。

Playwright Trace Viewer 是一个 GUI 工具，可让您探索记录的 Playwright 测试跟踪，这意味着您可以在测试的每个操作中前后移动，并直观地查看每个操作期间发生的情况。

### 3.6.1 Trace Viewer 追踪功能

可以使用 `browser_context.tracing` API 记录跟踪，如下所示：

```
browser = chromium.launch()
context = browser.new_context()

# Start tracing before creating / navigating a page.
context.tracing.start(screenshots=True, snapshots=True, sources=True)

page = context.new_page()
page.goto("https://playwright.dev")

# Stop tracing and export it into a zip archive.
context.tracing.stop(path = "trace.zip")
```

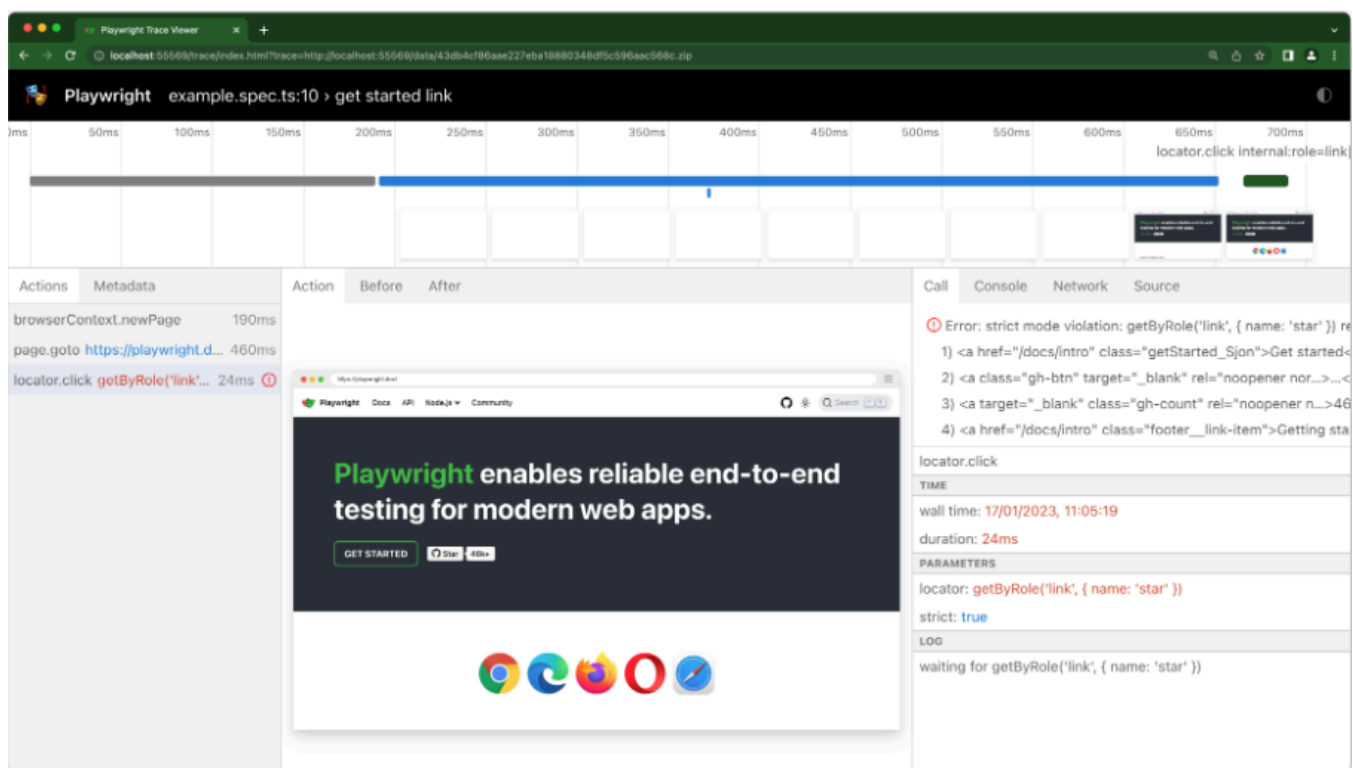
这将记录跟踪并将其放入名为 `trace.zip`。

您可以使用 Playwright CLI 或在您的浏览器中打开保存的跟踪 `trace.playwright.dev`。

```
playwright show-trace trace.zip
```

### 3.6.2 查看

通过单击每个操作或使用时间轴悬停来查看测试的痕迹，并查看操作前后页面的状态。在测试的每个步骤中检查日志、源和网络。跟踪查看器创建一个 DOM 快照，因此您可以与其完全交互，打开 devtools 等。



### 3.6.3 使用示例

打开网站登录，记录操作过程

```
from playwright.sync_api import Playwright, sync_playwright, expect
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

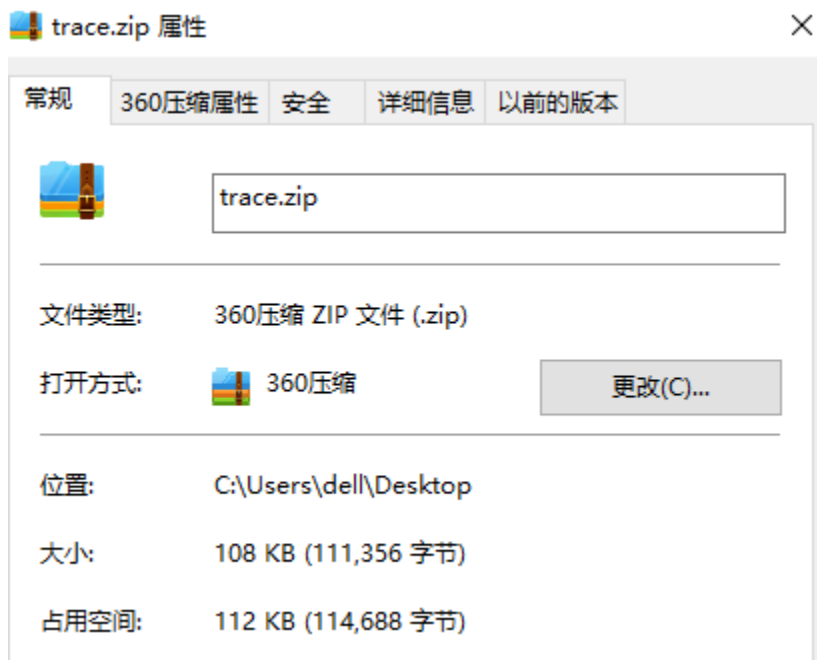
def run(playwright: Playwright) -> None:
    browser = playwright.chromium.launch(headless=False)
    context = browser.new_context()

    # Start tracing before creating / navigating a page.
    # 配置追踪开始
    context.tracing.start(screenshots=True, snapshots=True, sources=True)
    # 打开页面
    page = context.new_page()
    page.goto('http://127.0.0.1:8000/login.html')
    page.get_by_label("用户名:").fill("yoyo")
    page.get_by_label("密码:").fill("123456")
    page.locator("text=立即登录").click()

    context.tracing.stop(path="trace.zip") # 结束追踪的地方, 添加 tracing 的结束配置。
    context.close()
    browser.close()

with sync_playwright() as playwright:
    run(playwright)
```

运行结束后在本地保存一个 `trace.zip` 功能, 并且文件只有100K左右, 占用空间很小



### 3.6.4 查看追踪

有2种方法可以查看追踪日志

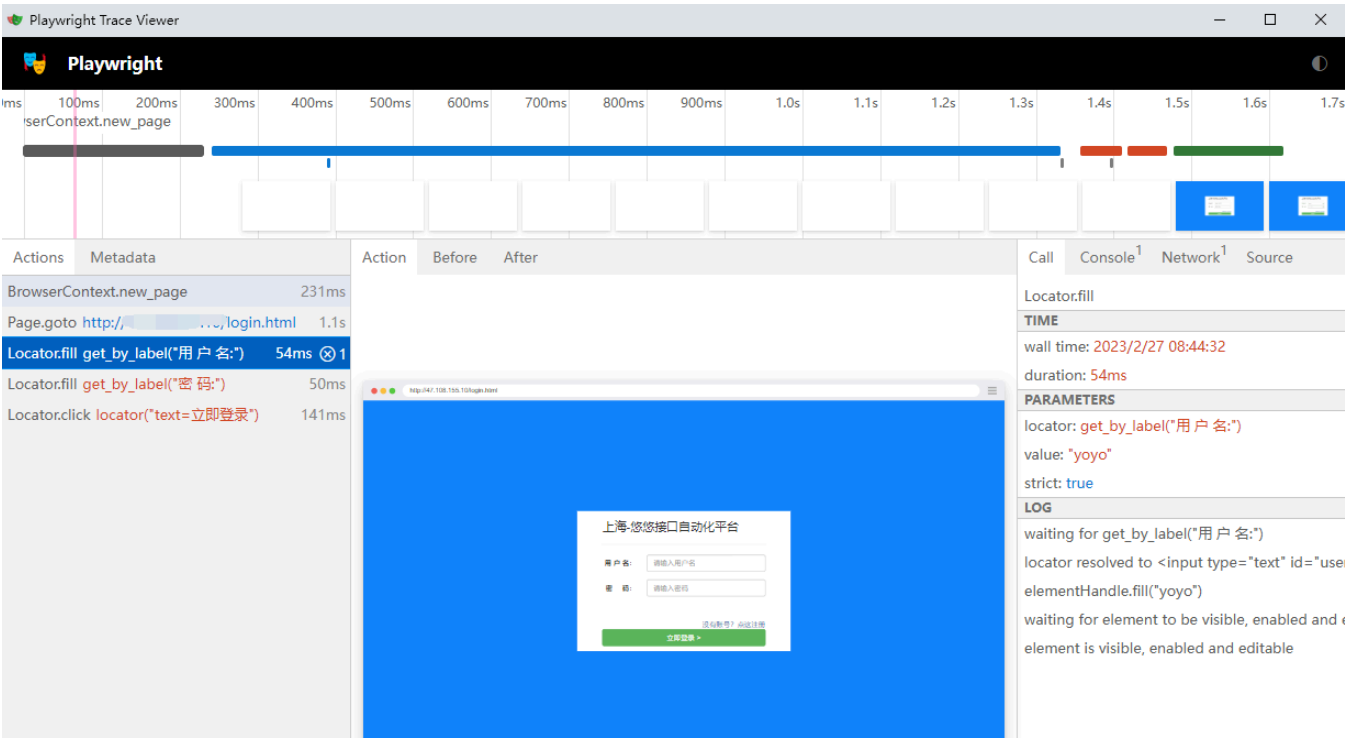
方法1. 通过命令行查看操作过程追踪

```
>playwright show-trace trace.zip
```

方法2.访问 <https://trace.playwright.dev/> 选择录制好的trace.zip文件即可打开




它可以清晰的记录代码的每个步骤



并且还能记录每个动作前和后的对比


Actions	Metadata	Action	Before	After	Call	Console <sup>1</sup>	Network <sup>1</sup>	Source
BrowserContext.new_page	231ms				Locator.fill			
Page.goto http://47.108.155.10/login.html	1.1s				TIME			
Locator.fill get_by_label("用户名:")	54ms				wall time: 2023/2/27 08:44:32			
Locator.fill get_by_label("密码:")	50ms				duration: 54ms			
Locator.click locator("text=立即登录")	141ms				PARAMETERS			
					locator: get_by_label("用户名:")			
					value: "yoyo"			
					strict: true			
					LOG			
					waiting for get_by_label("用户名:")			
					locator resolved to <input type="text" id="use			
					elementHandle.fill("yoyo")			
					waiting for element to be visible, enabled and			
					element is visible, enabled and editable			

输入用户名之前



Actions	Metadata	Action	Before	After	Call	Console <sup>1</sup>	Network <sup>1</sup>	Source
BrowserContext.new_page	231ms				Locator.fill			
Page.goto http://47.108.155.10/login.html	1.1s				TIME			
Locator.fill get_by_label("用户名:")	54ms				wall time: 2023/2/27 08:44:32			
Locator.fill get_by_label("密码:")	50ms				duration: 54ms			
Locator.click locator("text=立即登录")	141ms				PARAMETERS			
					locator: get_by_label("用户名:")			
					value: "yoyo"			
					strict: true			
					LOG			
					waiting for get_by_label("用户名:")			
					locator resolved to <input type="text" id="use			
					elementHandle.fill("yoyo")			
					waiting for element to be visible, enabled and			
					element is visible, enabled and editable			

输入之后



Playwright 的 Trace Viewer 功能非常强大，执行过程中出现的问题可以快速定位

## 3.7 保存cookie，解决反复登录的问题

在写web自动化的时候，很多用例是需要先登录的，为了做到每个用例的隔离，数据互不影响，一般会创建一个新的page对象。

很多用例是需要先登录的，可以先登录后把cookie保存到本地，通过加载cookie的方式解决重复登录的问题。

### 3.7.1 登录认证

Playwright 在称为浏览器上下文的隔离环境中执行测试。这种隔离模型提高了可重复性并防止级联测试失败。

测试可以加载现有的经过身份验证的状态。这消除了在每个测试中进行身份验证的需要并加快了测试执行速度。

我们建议创建 `auth` 目录并将其添加到您的 `.gitignore`。您的身份验证例程将生成经过身份验证的浏览器状态并将其保存到此 `auth` 目录中的文件中。稍后，测试将重用此状态并开始已通过身份验证。

windows 操作命令

```
md auth
echo. >> .gitignore
echo "auth" >> .gitignore
```

以下示例登录到 GitHub。执行这些步骤后，浏览器上下文将通过身份验证。

```
page = context.new_page()
page.goto('https://github.com/login')

# Interact with login form
page.get_by_label("Username or email address").fill("username")
page.get_by_label("Password").fill("password")
page.get_by_role("button", name="Sign in").click()
# Continue with the test
```

为每个测试重做登录会减慢测试执行速度。为了缓解这种情况，请改用现有的身份验证状态

## 3.7.2 storageState保存登录cookies

Playwright 提供了一种在测试中重用登录状态的方法。这样您就可以只登录一次，然后跳过所有测试的登录步骤。

Web 应用程序使用基于 cookie 或基于令牌的身份验证，其中经过身份验证的状态存储为 cookie 或本地存储。Playwright 提供 `browserContext.storageState([options])` 方法，可用于从经过身份验证的上下文中检索存储状态，然后创建具有预填充状态的新上下文。

Cookie 和本地存储状态可以跨不同的浏览器使用。它们取决于您的应用程序的身份验证模型：某些应用程序可能需要 cookie 和本地存储。

以下代码片段从经过身份验证的上下文中检索状态，并使用该状态创建一个新上下文。

```

from playwright.sync_api import Playwright, sync_playwright, expect
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

def run(playwright: Playwright) -> None:
    browser = playwright.chromium.launch(headless=False)
    context = browser.new_context()
    page = context.new_page()
    page.goto('https://github.com/login')

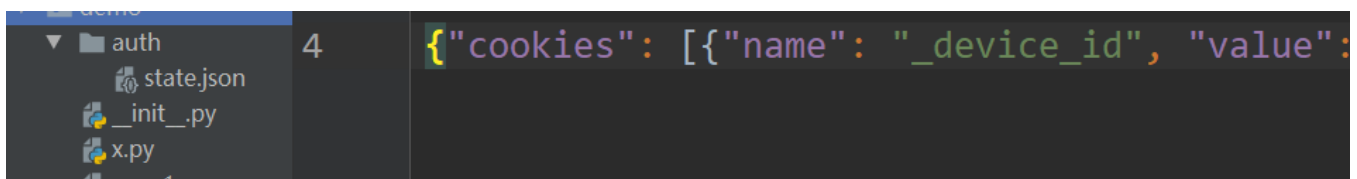
    # Interact with login form
    page.get_by_label("Username or email address").fill("yoyo")
    page.get_by_label("Password").fill("*****")
    page.get_by_role("button", name="Sign in").click()
    # Continue with the test

    # 保存storage state 到指定的文件
    storage = context.storage_state(path="auth/state.json")

    # -----
    context.close()
    browser.close()

with sync_playwright() as playwright:
    run(playwright)

```



于是在本地会保存一个state.json文件

这样在其它地方就可以使用本地的cookies，实现免登录了

```

# Create a new context with the saved storage state.
context = browser.new_context(storage_state="state.json")

```

验证下是不是已经不需要登录了



```
from playwright.sync_api import Playwright, sync_playwright, expect
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

def run(playwright: Playwright) -> None:
    browser = playwright.chromium.launch(headless=False)
    # 加载本地cookies, 免登陆
    context = browser.new_context(storage_state="auth/state.json")

    # 打开页面继续操作
    page = context.new_page()
    page.goto('https://github.com/')
    page.pause() # 打断点看是不是已经登录了

    context.close()
    browser.close()

with sync_playwright() as playwright:
    run(playwright)
```

### 3.7.3 高级场景

重用经过身份验证的状态包括cookie和基于本地存储的身份验证。会话存储很少用于存储与登录状态相关的信息。会话存储特定于特定域，并且不会跨页面加载持续存在。Playwright 不提供持久会话存储的 API，但以下代码片段可用于保存/加载会话存储。

使用添加到环境变量的方式

```
import os
# Get session storage and store as env variable
session_storage = page.evaluate("() => JSON.stringify(sessionStorage)")
os.environ["SESSION_STORAGE"] = session_storage

# Set session storage in a new context
session_storage = os.environ["SESSION_STORAGE"]
context.add_init_script("""(storage => {
  if (window.location.hostname === 'example.com') {
    const entries = JSON.parse(storage)
    for (const [key, value] of Object.entries(entries)) {
      window.sessionStorage.setItem(key, value)
    }
  }
}))(session_storage + session_storage + '')")
```

## 3.8 强大的Mock 接口返回， 模拟各种异常场景

web 自动化主要测前端UI 的功能， 有很多异常的场景， 我们很难造真实的场景去触发， 比如服务器异常时候， 前端的提示语。

这时候就可以使用mock 功能， 模拟接口的返回， 测试前端的功能。

### 3.8.1 模拟网络请求

Web API 通常作为 HTTP 端点实现。Playwright 提供 API 来模拟和修改网络流量， 包括 HTTP 和 HTTPS。页面执行的任何请求， 包括XHR和获取请求， 都可以被跟踪、修改和模拟。

以下代码将拦截对的所有调用https://dog.ceo/api/breeds/list/all并将返回测试数据。不会向https://dog.ceo/api/breeds/list/all端点发出任何请求。

```
async def handle(route):
    json = { message: { "test_breed": [] } }
    route.fulfill(json=json)

page.route("https://dog.ceo/api/breeds/list/all", handle)
```

### 3.8.2 修改 API

有时， 必须发出 API 请求， 但需要修补响应以允许可重现的测试。在这种情况下， 可以执行请求并使用修改后的响应来完成请求， 而不是模拟请求。

```
def handle(route):
    response = route.fulfill()
    json = response.json()
    json["message"]["big_red_dog"] = []
    # Fulfill using the original response, while patching the response body
    # with the given JSON object.
    route.fulfill(response=response, json=json)

page.route("https://dog.ceo/api/breeds/list/all", handle)
```

### 3.8.3 模拟登录时候，服务器异常的场景



当登录的接口返回状态码是500 的时候，前端才会触发：服务器异常！

于是可以用 `page.route()` 拦截请求，修改状态码即可触发

```

from playwright.sync_api import Playwright, sync_playwright, expect
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

def handle(route):
    # 状态码改成500 模拟服务器异常
    route.fulfill(status=500)

def run(playwright: Playwright) -> None:
    browser = playwright.chromium.launch(headless=False)
    context = browser.new_context()
    page = context.new_page()
    page.goto("http://127.0.0.0:8000/login.html")
    page.get_by_placeholder("请输入用户名").click()
    page.get_by_placeholder("请输入用户名").fill("yoyo")
    page.get_by_placeholder("请输入密码").click()
    page.get_by_placeholder("请输入密码").fill("aa123456")

    page.route("http://47.108.155.10/api/login", handle)

    page.get_by_role("button", name="立即登录 >").click()
    page.pause() # 断点
    # -----
    context.close()
    browser.close()

with sync_playwright() as playwright:
    run(playwright)

```

这就给我们测试前端的各种异常场景带来了很大的方便，可以模拟出任何我们希望返回的接口数据。这对于前后端分离的项目测试是非常关键的。

## 3.9 模拟手机浏览器测试

使用 Playwright，您可以在任何浏览器上测试您的应用程序，也可以模拟真实设备，例如手机或平板电脑。

### 3.9.1 设置手机模式

只需配置您想要模拟的设备，Playwright 就会模拟浏览器行为，例

如"user\_agent"、"screen\_size"以及"viewport"是否"has\_touch"启用。  
您还可以为所有测试或特定测试模拟,以及设置以"geolocation"显示通知或更改."locale""timezone""permissions""colorScheme"

Playwright使用 `playwright.devices` 为选定的台式机、平板电脑和移动设备提供设备参数注册表。

它可用于模拟特定设备的浏览器行为,例如用户代理、屏幕尺寸、视口以及是否启用了触摸。所有测试都将使用指定的设备参数运行。

## 使用示例

```
from playwright.sync_api import sync_playwright
# 上海悠悠 wx:283340479
# blog: https://www.cnblogs.com/yoyoketang/

def run(playwright):
    iphone_12 = playwright.devices['iPhone 12']
    browser = playwright.chromium.launch(headless=False)
    context = browser.new_context(
        **iphone_12,
    )
    page = context.new_page()
    page.goto('https://m.baidu.com')
    page.pause()

with sync_playwright() as playwright:
    run(playwright)
```

需注意手机设备参数必须是playwright 支持的设备列表里面的手机型号才可以。

运行后就是以手机模式打开的



### 3.9.2 playwright.devices 可支持的手机列表

有2种方式可以查看 playwright.devices 所支持的全部设置

第一种，直接打印 playwright.devices

```
with sync_playwright() as p:
    import pprint
    pprint.pprint(p.devices)
```

第二种，查看deviceDescriptorsSource.json源码

playwright可以支持的手机列表可以在GitHub 上查看到

<https://github.com/microsoft/playwright/blob/main/packages/playwright-core/src/server/deviceDescriptorsSource.json>

查看到 iPhone 12 手机相关配置参数

```
'iPhone 12': {'default_browser_type': 'webkit',
              'device_scale_factor': 3,
              'has_touch': True,
              'is_mobile': True,
              'user_agent': 'Mozilla/5.0 (iPhone; CPU iPhone OS 14_4 like Mac '
                            'OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) '
                            'Version/16.4 Mobile/15E148 Safari/604.1',
              'viewport': {'height': 664, 'width': 390}},
```

### 3.9.3 pytest-playwright 测试用例

在pytest-playwright 测试用例中可以通过重写browser\_context\_args 来实现

```
# conftest.py

import pytest
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

@pytest.fixture(scope="session")
def browser_context_args(browser_context_args, playwright):
    iphone_11 = playwright.devices['iPhone 11 Pro']
    return {
        **browser_context_args,
        **iphone_11,
    }
```

用例部分

```
from playwright.sync_api import Page

def test_m_baidu(page: Page):
    page.goto("https://m.baidu.com/")
```

这样可以指定手机型号运行用例

## 3.10 获取页面完整 HTML 内容

selenium 里面有个driver.page\_source 可以获取整个html页面的内容， playwright里面也有类似的方法 使用 page.content()

什么场景会用到：做测试基本用不到，如果是爬虫相关的，会用的到。

### 3.10.1 page.content() 获取 html 内容

使用示例

```
from playwright.sync_api import sync_playwright
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)
    context = browser.new_context()
    page = context.new_page()

    page.goto("https://www.cnblogs.com/yoyoketang/")
    print(page.content())
```

```
<!DOCTYPE html><html lang="zh-cn" style="--olcb-folder-code-block-max-height:80vh;"><head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta name="referrer" content="origin-when-cross-origin">

  <meta http-equiv="Cache-Control" content="no-transform">
  <meta http-equiv="Cache-Control" content="no-siteapp">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <title>上海-悠悠 - 博客园</title>
  <link id="favicon" rel="shortcut icon" href="//common.cnblogs.com/favicon.svg" type="image/svg+xml"/>

  <style>#home :not(.cnblogs_code):not(.cnblogs_Highlighter)>pre:not([highlighted]):not([class*="
    padding: 13px;
    border: 1px solid #ccc; margin: 6px 6px 6px 6px;
  </style>
```

对于喜欢爬虫的小伙伴就可以去抓取页面上的内容了。

### 3.10.2 inner\_html() 与 inner\_text()

page.content() 是获取整个页面的HTML,如果我们只需获取某个元素的HTML，如下图



上海-悠悠

2023年第 3 期《Python 测试平台开发》进阶课程（3月5号开学）

2023年第 14期《Python接口自动化+Playwright》课程，4月2号开学（课程全面升级！）！

报名联系weixin/qq: 283340479

正在连接 www.google-analytics.com...

查看器 控制台 调试器 网络 样式编辑器 性能 内存 存储 无障碍环境 应用程序

搜索 HTML

```
<!DOCTYPE html>
<html style="--olcb-folder-code-block-max-height: 80vh;" lang="zh-cn">
  <head>
  </head>
  <body class="skin-coffee has-navbar">
    <a name="top"></a>
    <div id="top_nav" class="navbar forpc">
    </div>
    <div id="home">
      <div id="header">
        <div id="blogTitle">
          <a href="https://www.cnblogs.com/yoyoketang/">
            
          </a>
          <!--done-->
          <h1>
          </h1>
          <h2>
          </h2>
        </div>
      </div>
    </div>
  </body>
</html>
```

获取此元素的内容

```
元素 {
}
#blogTitle {
  height: 178px;
  clear: both;
  background: url('/skins/coffee/images/bg_title.gif') 170px 40px no-repeat;
}
* {
  margin: 0;
  padding: 0;
}
继承自 body
body {
  background-color: #f0f0f0;
}
```

可以使用locator().inner\_html() 方法获取

- inner\_html() 获取元素的整个html源码内容
- inner\_text() 获取元素的文本内容

```
from playwright.sync_api import sync_playwright
```

```
# 上海悠悠 wx:283340479
```

```
# blog:https://www.cnblogs.com/yoyoketang/
```

```
with sync_playwright() as p:
```

```
    browser = p.chromium.launch(headless=False)
```

```
    context = browser.new_context()
```

```
    page = context.new_page()
```

```
    page.goto("https://www.cnblogs.com/yoyoketang/")
```

```
    # print(page.content())
```

```
    # 获取某个元素的HTML
```

```
    blog = page.locator('#blogTitle')
```

```
    print(blog.inner_html())
```

```
    print('-----上海-悠悠-----')
```

```
    print(blog.inner_text())
```

```

<a href="https://www.cnblogs.com/yoyoketang/">
<h2>2023年第 3 期《Python 测试平台开发》进阶课程（3月5号开学） <br>
2023年第 14期《Python接口自动化+Playwright 》课程，4月2号开学（课程全面升级！）！ <br>报名联

```

-----上海-悠悠-----

上海-悠悠

2023年第 3 期《Python 测试平台开发》进阶课程（3月5号开学）

2023年第 14期《Python接口自动化+Playwright 》课程，4月2号开学（课程全面升级！）！

报名联系weixin/qq: 283340479

## 3.11 text\_content 与 inner\_text 获取元素文本

text\_content() 用来获取某个元素内所有文本内容，包含子元素内容，隐藏元素也能获取。

inner\_text() 的返回值会被格式化 ,但是text\_content()的返回值不会被格式化

最重要的区别 inner\_text()返回的值, 依赖于页面的显示, text\_content()依赖于代码的内容

### 3.11.1 text\_content() 与 inner\_text() 获取页面文本

text\_content() 用来获取某个元素内所有文本内容，包含子元素内容，隐藏元素也能获取。

inner\_text() 的返回值会被格式化 ,但是text\_content()的返回值不会被格式化

```

from playwright.sync_api import sync_playwright
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)
    context = browser.new_context()
    page = context.new_page()

    page.goto("https://www.cnblogs.com/yoyoketang/")
    # print(page.content())
    # 获取某个元素的HTML
    blog = page.locator('#blogTitle')
    # print(blog.inner_html())
    # print('-----上海-悠悠-----')
    # print(blog.inner_text())
    print(blog.text_content())

```

|

上海-悠悠

2023年第 3 期《Python 测试平台开发》进阶课程（3月5号开学）

2023年第 14期《Python接口自动化+Playwright 》课程，4月2号开学（课程全面升级！）！

### 3.11.2 all\_inner\_texts() 与 all\_text\_contents()

all\_inner\_texts() 和 all\_text\_contents() 也是用于获取页面上的文本，但是返回的是list列表

```

from playwright.sync_api import sync_playwright
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)
    context = browser.new_context()
    page = context.new_page()

    page.goto("https://www.cnblogs.com/yoyoketang/")
    # print(page.content())
    # 获取某个元素的HTML
    blog = page.locator('#blogTitle')
    # print(blog.inner_html())
    # print('-----上海-悠悠-----')
    # print(blog.inner_text())
    # print(blog.text_content())

    print(blog.all_inner_texts())
    print('-----上海-悠悠-----')
    print(blog.all_text_contents())

```

```

['上海-悠悠\n2023年第 3 期《Python 测试平台开发》进阶课程（3月5号开学）\n2023年第
-----上海-悠悠-----
['\n\t\t\t\t\t\n\t\t\t\t\t\n上海-悠悠\n\n2023年第 3 期《Python 测试平台开发》进阶课程

```

## 3.12 bounding\_box 获取元素坐标与宽高

bounding\_box 可以获取元素的坐标与宽高

使用场景：一般在登录页面滑块操作的时候用到

### 3.12.1 使用示例

以下图登录按钮为例

元素的坐标是以页面左上角为起点，横向是x轴距离

纵向是y轴距离

每个元素都有宽高，bounding\_box()方法可以获取到元素的宽高



```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False, slow_mo=1000)
    context = browser.new_context()
    page = context.new_page()
    page.goto("http://47.108.155.10/login.html")

    btn = page.locator('#loginBtn')
    print(btn.bounding_box())
```

## 运行结果

```
{'x': 477.2124938964844, 'y': 455.8000183105469, 'width': 325.5500183105469, 'height': 33.5999755859375}
```

具体使用场景参照：

1.12 登录-页面滑动解锁

1.16 登录-滑块拼图验证码

## 3.13 mouse 鼠标操作总结

Mouse 鼠标操作是基于page对象去调用。常用的鼠标操作有

- 单击 click

- 双击 dblclick
- 按住 down
- 移动 move
- 释放 up
- 滚轮操作 wheel
- 鼠标悬停 locator.hover
- 拖拽 locator.drag\_to

什么场景会用到鼠标事件？

1. 登录页面滑块操作用的比较多
2. 鼠标悬停，拖拽，滚动条操作
3. 页面上能定位到某个元素，但是click无效（selenium上经常遇到），可以用鼠标的点击事件

### 3.13.1 page.mouse 使用

Mouse 类在相对于视口左上角的主框架 CSS 像素中运行。

每个page对象都有mouse方法，可通过page.mouse访问。

```
# using 'page.mouse' to trace a 100x100 square.  
page.mouse.move(0, 0)  
page.mouse.down()  
page.mouse.move(0, 100)  
page.mouse.move(100, 100)  
page.mouse.move(100, 0)  
page.mouse.move(0, 0)  
page.mouse.up()
```

### 3.13.2 click 点击

鼠标click 点击是mouse.move()、mouse.down()、mouse.up()的快捷方式。

```

def click(
    self,
    x: float,
    y: float,
    *,
    delay: typing.Optional[float] = None,
    button: typing.Optional[Literal["left", "middle", "right"]] = None,
    click_count: typing.Optional[int] = None
) -> None:
    """Mouse.click

    Shortcut for `mouse.move()`, `mouse.down()`, `mouse.up()`.

    Parameters
    -----
    x : float
    y : float
    delay : Union[float, None]
        Time to wait between `mousedown` and `mouseup` in milliseconds. Defaults to 0.
    button : Union["left", "middle", "right", None]
        Defaults to `left`.
    click_count : Union[int, None]
        defaults to 1. See [UIEvent.detail].
    """

```

参数详解：

- x 横向坐标位置
- y 纵向坐标位置
- delay 是 `mousedown` 和 `mouseup` 事件中间的等待时间，单位是毫秒，默认是0
- button 是点击元素的位置："left", "middle", "right"，默认参数是left
- click\_count 是点击次数

```

mouse.click(x, y)
mouse.click(x, y, **kwargs)

```

鼠标事件点击元素按钮示例

(需注意，如果直接点击元素左上角位置，可能无效，因为有些元素是圆角)

```

from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False, slow_mo=1000)
    context = browser.new_context()
    page = context.new_page()
    page.goto("http://47.108.155.10/login.html")

    btn = page.locator('#loginBtn')
    box = btn.bounding_box()
    print(box)
    # 鼠标事件点击元素
    page.mouse.click(x=box['x']+box['width']/2, y=box['y'])

    page.pause()

```

### 3.13.3 dblclick 双击

鼠标双击是mouse.move()、mouse.down()、mouse.up()、mouse.down()和mouse.up()的快捷方式。

```

# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

mouse.dblclick(x, y)
mouse.dblclick(x, y, **kwargs)

```

### 3.13.4 mouse.down 按住鼠标

调度一个mousedown事件。

有2个参数

- button 是点击元素的位置："left", "middle", "right", 默认参数是left
- click\_count 是点击次数

```

mouse.down()
mouse.down(**kwargs)

```



### 3.13.5 mouse.move 移动

调度一个 `mousemove` 事件。

```
def move(self, x: float, y: float, *, steps: typing.Optional[int] = None) -> None:
    """Mouse.move

    Dispatches a `mousemove` event.

    Parameters
    -----
    x : float
    y : float
    steps : Union[int, None]
        Defaults to 1. Sends intermediate `mousemove` events.
    """
```

用法

```
mouse.move(x, y)
mouse.move(x, y, **kwargs)
```

### 3.13.6 mouse.up 释放鼠标

调度一个 `mouseup` 事件。

```
mouse.up()
mouse.up(**kwargs)
```

### 3.13.7 wheel 滚轮

调度一个 `wheel` 事件。（滚轮事件如果不处理可能会导致滚动，该方法不会等待滚动结束才返回。）

```
mouse.wheel(delta_x, delta_y)
```

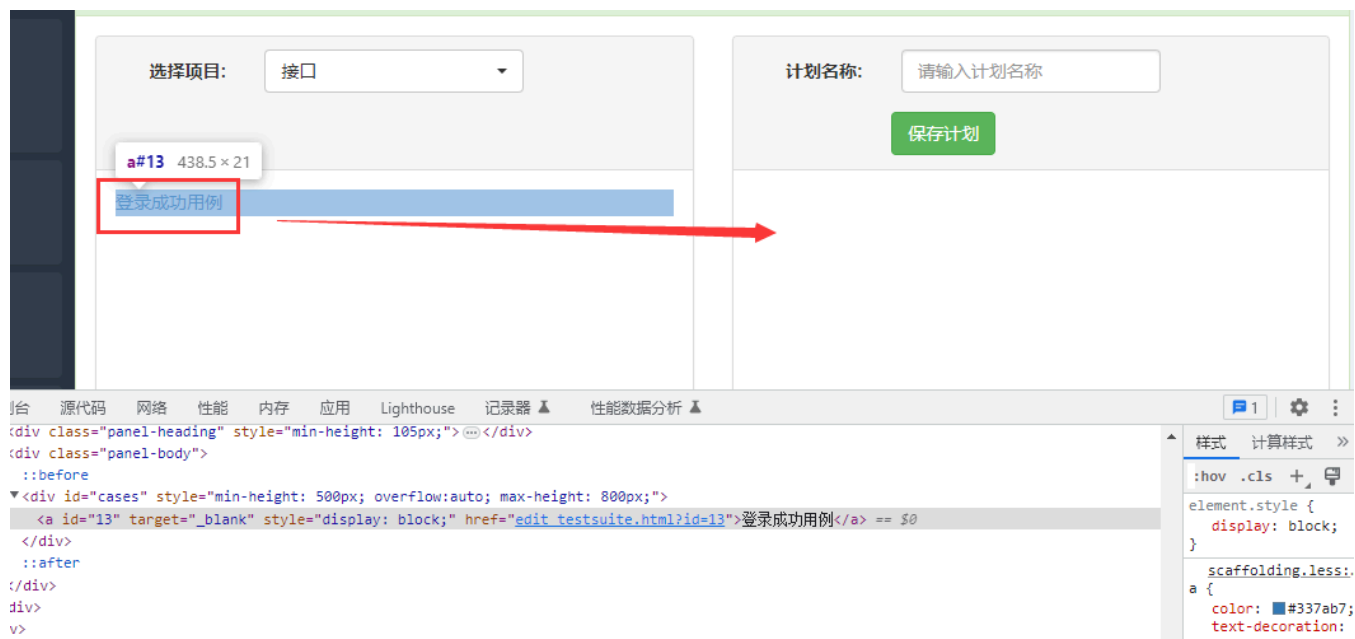
## 3.14 鼠标操作- drag\_to 拖拽

按住元素从页面的一个位置拖动到另外一个位置，有3种方式可以实现

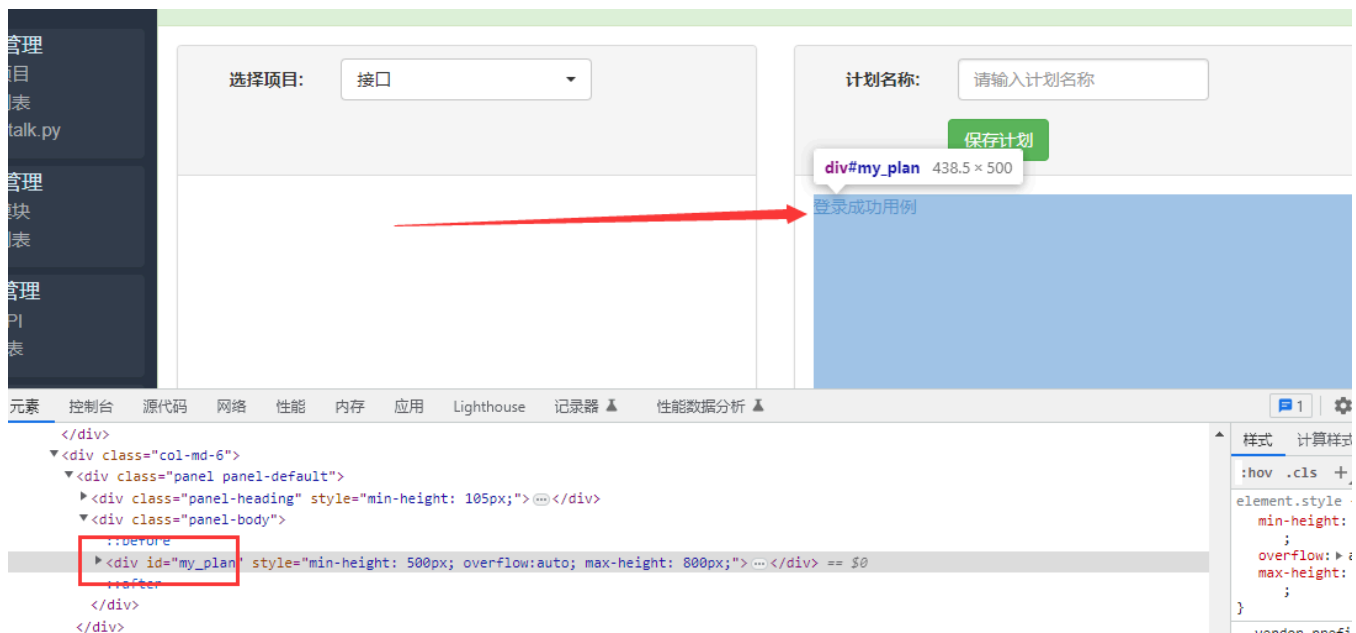
- locator.drag\_to(target: locator) 先定位元素，调用drag\_to方法到目标元素
- page.drag\_and\_drop(source: str, target: str) page对象直接调用
- 自己拖动 hover->down->hover->up 方式拖动

### 3.14.1 场景

目标元素



拖动到指定位置



### 3.14.2 locator.drag\_to 拖拽操作

您可以使用locator.drag\_to()执行拖放操作。此方法将：

- 将鼠标悬停在要拖动的元素上。
- 按鼠标左键。
- 将鼠标移动到将接收放置的元素。
- 松开鼠标左键。

使用示例

```
page.locator("#item-to-be-dragged").drag_to(page.locator("#item-to-drop-at"))
```

先分别定为目标元素和拖动的位置元素，调用drag\_to 方法拖拽操作

```
# 目标元素
source = page.get_by_text('登录成功用例')
# 终点位置
target = page.locator('#my_plan')
# 拖拽操作
source.drag_to(target)
```

### 3.14.3 page.drag\_and\_drop 拖动

通过page对象调用drag\_and\_drop ,部分源码如下

```
def drag_and_drop(
    self,
    source: str,
    target: str,
    *,
    source_position: typing.Optional[Position] = None,
    target_position: typing.Optional[Position] = None,
    force: typing.Optional[bool] = None,
    no_wait_after: typing.Optional[bool] = None,
    timeout: typing.Optional[float] = None,
    strict: typing.Optional[bool] = None,
    trial: typing.Optional[bool] = None
) -> None:
```

source 和 target 是字符串格式，也就是传selector 选择器的方法

```
page.drag_and_drop('text=登录成功用例', '#my_plan')
```

### 3.14.4 手动拖动

如果您想精确控制拖动操作，请使用较低级别的方法，如locator.hover()、mouse.down()、mouse.move()和mouse.up()。

```
page.get_by_text('登录成功用例').hover()
page.mouse.down()
page.locator("#my_plan").hover()
page.mouse.up()
```

如果您的页面依赖于dragover正在调度的事件，则您至少需要移动两次鼠标才能在所有浏览器中触发它。

要可靠地发出第二次鼠标移动，请重复mouse.move()或locator.hover()两次。

操作顺序是：悬停拖动元素，鼠标向下，悬停放置元素，第二次悬停放置元素，鼠标向上。

### 3.14.5 示例代码

```
"""
drag_to 拖拽操作
"""

from playwright.sync_api import sync_playwright, expect

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)
    context = browser.new_context()
    page = context.new_page()
    page.goto("http://47.108.155.10/login.html")
    page.get_by_placeholder("请输入用户名").fill("yoyo")
    page.get_by_placeholder("请输入密码").fill("aa123456")
    page.get_by_role("button", name="立即登录 >").click()

    page.get_by_text('我的计划').click()
    page.locator('#btn-add-task').click()

    page.select_option('#project', '接口')

    # 方法一
    source = page.get_by_text('登录成功用例')
    # 终点位置
    target = page.locator('#my_plan')
    # 拖拽操作
    source.drag_to(target)

    # 方法二
    # page.drag_and_drop('text=登录成功用例', '#my_plan')

    # 方法三
    # page.get_by_text('登录成功用例').hover()
    # page.mouse.down()
    # page.locator("#my_plan").hover()
    # page.mouse.up()

    page.pause()
```

### 3.15 鼠标操作- 滚轮操作mouse.wheel

有些网站是动态加载的，当拖动页面右侧滚动条后会自动加载网页下面的内容，或者通过鼠标滚轮操作。

使用场景：网页动态加载，拖动滚动条继续加载内容

(大部分情况下，click可以直接点到元素，或者使用hover方法滚动到元素出现位置)

### 3.15.1 鼠标滚轮操作

鼠标滚轮操作调用page.mouse.wheel() 方法

- delta\_x 横向移动距离
- delta\_y 纵向移动距离

```
def wheel(self, delta_x: float, delta_y: float) -> None:
    """Mouse.wheel

    Dispatches a `wheel` event.

    **NOTE** Wheel events may cause scrolling if they are not handled, and this method does not
    to finish before returning.

    Parameters
    -----
    delta_x : float
        Pixels to scroll horizontally.
    delta_y : float
        Pixels to scroll vertically.
    """
```

### 3.15.2 使用示例

一边滚动一边加载网页

```
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/
from playwright.sync_api import Playwright, sync_playwright

with sync_playwright() as playwright:
    browser = playwright.chromium.launch(headless=False)
    page = browser.new_page()
    page.goto('https://www.jianshu.com/')

    for i in range(50):
        page.mouse.wheel(0, 100)
        page.wait_for_timeout(500)

    page.pause()
    browser.close()
```

## 3.16 Keyboard 键盘基本操作

Keyboard 提供了一个用于管理虚拟键盘的 API。高级 api 是 `keyboard.type()`，它接受原始字符并在您的页面上生成适当的 `keydown`、`keypress/input` 和 `keyup` 事件。

为了更好地控制，您可以使用 `keyboard.down()`、`keyboard.up()` 和 `keyboard.insert_text()` 手动触发事件，就好像它们是从真实键盘生成的一样。

页面上输入框输入文本内容总结下有四种方式

1. `page.fill()` 输入字符串
2. `page.type()` 字符一个个敲入
3. `page.keyboard.type()` 键盘事件，模拟键盘上按钮操作
4. `page.keyboard.insert_text()` 键盘事件，插入一段文本

### 3.16.1 键盘操作示例

按住 Shift 以选择和删除某些文本的示例：

```
page.keyboard.type("Hello World!")
page.keyboard.press("ArrowLeft")
page.keyboard.down("Shift")
for i in range(6):
    page.keyboard.press("ArrowLeft")
page.keyboard.up("Shift")
page.keyboard.press("Backspace")
# result text will end up saying "Hello!"
```

按大写字母A 的例子

```
page.keyboard.press("Shift+KeyA")
# or
page.keyboard.press("Shift+A")
```

按 Ctrl+A 选择全部

```
# on windows and linux
page.keyboard.press("Control+A")
# on mac_os
page.keyboard.press("Meta+A")
```

## 3.16.2 down 向下

调度一个keydown事件。

key可以指定预期的keyboardEvent.key值或单个字符来为其生成文本。可以在此处key找到这些值的超集。键的例子是：

F1- F12, Digit0- Digit9, KeyA- KeyZ, Backquote, Minus, Equal, Backslash, Backspace, Tab, Delete, Escape, ArrowDown, End, Enter, Home, , , Insert, 等  
\_PageDownPageUpArrowRightArrowUp

还支持以下修改快捷方式：Shift, Control, Alt, Meta, ShiftLeft.

按住将键入对应于大写字母Shift的文本。key

如果key是单个字符，则区分大小写，因此值a和A将生成各自不同的文本。

如果key是修饰键、Shift、Meta、Control或Alt，则后续按键将在该修饰键激活的情况下发送。



要释放修饰键，请使用`keyboard.up()`。

按下一次键后，对`keyboard.down()`的后续调用会将`repeat`设置为 `true`。要释放键，请使用`keyboard.up()`。

笔记：修改键确实影响`keyboard.down`。按住Shift将以大写形式键入文本。

用法

```
keyboard.down(key)
```

### 3.16.3 insert\_text 插入文本

仅调度input事件，不发出`keydown`,`keyup`或`keypress`事件。

```
page.keyboard.insert_text("嗨")
```

笔记：修改键不影响`keyboard.insertText`。按住Shift不会以大写形式键入文本。

### 3.16.4 press 按住

`key`可以指定预期的`keyboardEvent.key`值或单个字符来为其生成文本。可以在此处`key`找到这些值的超集。键的例子是：

F1- F12, Digit0- Digit9, KeyA- KeyZ, Backquote, Minus, Equal, Backslash, Backspace, Tab, Delete, Escape, ArrowDown, End, Enter, Home, , , Insert, 等  
\_PageDownPageUpArrowRightArrowUp

还支持以下修改快捷方式：Shift, Control, Alt, Meta, ShiftLeft.

按住将键入对应于大写字母Shift的文本。`key`

如果`key`是单个字符，则区分大小写，因此值`a`和`A`将生成各自不同的文本。

也支持`key`: "Control+o"或等快捷方式。`key`: "Control+Shift+T"当使用修饰符指定时，修饰符被按下并在按下后续键时按住。

```
page = browser.new_page()
page.goto("https://keycode.info")
page.keyboard.press("a")
page.screenshot(path="a.png")
page.keyboard.press("ArrowLeft")
page.screenshot(path="arrow_left.png")
page.keyboard.press("Shift+O")
page.screenshot(path="o.png")
browser.close()
```

keyboard.down()和keyboard.up()的快捷方式。

### 3.16.5 Type 操作

为文本中的每个字符发送keydown、keypress/input和事件。keyup

要按特殊键，例如Control或ArrowDown，请使用keyboard.press()。

使用示例

```
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

page.keyboard.type("Hello") # types instantly
page.keyboard.type("World", delay=100) # types slower, like a user
```

### 3.16.6 up 方法

调度一个keyup事件。

```
keyboard.up(key)
```

## 3.16.7 使用示例

```
from playwright.sync_api import sync_playwright, expect

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)
    context = browser.new_context()
    page = context.new_page()
    page.goto("http://47.108.155.10/login.html")

    user = page.locator('#username')
    # 鼠标聚焦输入框
    user.focus()
    # type 输入内容
    page.keyboard.type('yoyoketang')
    # press 按键操作
    page.keyboard.press('Backspace')
    # Control+A快捷键
    page.keyboard.press('Control+A')

    page.pause()
```

## 3.17 模拟键盘操作 - press 复制粘贴相关

playwright 可以模拟键盘操作，定位到元素使用press()方法

- page.keyboard.press() 直接操作键盘
- page.locator.press() 定位元素后操作键盘

### 3.17.1 press()方法 介绍

locator.press ()方法聚焦所选元素并产生单个击键。它接受在键盘事件的keyboardEvent.key属性中发出的逻辑键名称：

```
Backquote, Minus, Equal, Backslash, Backspace, Tab, Delete, Escape,
ArrowDown, End, Enter, Home, Insert, PageDown, PageUp, ArrowRight,
ArrowUp, F1 - F12, Digit0 - Digit9, KeyA - KeyZ, etc.
```

使用示例

```
# 敲 Enter 键
page.get_by_text("Submit").press("Enter")

# 按住 Control+右箭头→
page.get_by_role("textbox").press("Control+ArrowRight")

# 按键盘上的 $ 符号
page.get_by_role("textbox").press("$")
```

您也可以指定要生成的单个字符，例如"a"或"#"。

还支持以下修改快捷方式：Shift, Control, Alt, Meta.

简单版本产生单个字符。这个字符是区分大小写的，所以"a"和"A"会产生不同的结果。

```
# <input id=name>
page.locator('#name').press('Shift+A')

# <input id=name>
page.locator('#name').press('Shift+ArrowLeft')
```

也支持"Control+o"或"Control+Shift+T" 等快捷方式，当使用修饰符指定时，修饰符被按下并在按下后续键时按住。

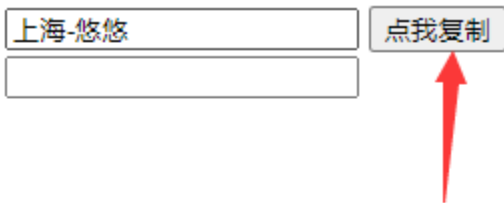
请注意，您仍然需要指定大写字母A以Shift-A生成大写字符。Shift-a产生一个小写的，就好像你有CapsLock切换。

## 3.17.2 使用实例

如下网页中

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
  <div>
    <input id="demoInput" value="上海-悠悠">
    <button id="btn">点我复制</button>
  </div>
  <div>
    <input id="copy" value="">
  </div>
</body>
<script>
  const btn = document.querySelector('#btn');
  btn.addEventListener('click', () => {
    const input = document.querySelector('#demoInput');
    input.select();
    if (document.execCommand('copy')) {
      document.execCommand('copy');
      console.log('复制成功');
    }
  })
</script>
</html>
```

点按钮复制到粘贴板



使用快捷键"Control+V" 粘贴到页面其它发位置

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)
    page = browser.new_page()
    page.goto("file:///C:/Users/dell/Desktop/demo.html")

    # 点击按钮后复制到粘贴板
    page.locator('#btn').click()

    # Ctrl + v 粘贴到另外一个地方
    page.locator('#copy').press('Control+V')

    page.pause()
    browser.close()
```

### 3.17.3 复制到本地

如果需要把粘贴板的内容复制到本地文件，需用到python的第三方库保存到本地

```
pip install pyperclip
```

使用示例

```
import pyperclip

x = "上海-悠悠"

# 复制到剪切板
pyperclip.copy(x)

# 获取剪切板的内容
y = pyperclip.paste()
print(f"粘贴的内容: {y}")
```

### 3.17.4 获取playwright 复制到剪切板内容

playwright 结合 pyperclip 使用

```

from playwright.sync_api import sync_playwright
import pyperclip

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)
    page = browser.new_page()
    page.goto("file:///C:/Users/dell/Desktop/demo.html")

    # 点击按钮后复制到粘贴板
    page.locator('#btn').click()

    # 获取剪切板的内容
    y = pyperclip.paste()
    print(f"粘贴的内容: {y}")

    # # Ctrl + v 粘贴到另外一个地方
    # page.locator('#copy').press('Control+V')
    browser.close()

```

## 3.18 page.evaluate()执行JavaScript脚本

大家在学selenium的时候，对于页面上的有些元素不好操作的时候，可以使用 `driver.execute_script()` 方法执行JavaScript脚本。

在playwright 中也有类似的方法，使用`page.evaluate()`执行JavaScript脚本。

`page.evaluate()`和`page.evaluate_handle()`之间的唯一区别是`page.evaluate_handle()`返回JSHandle。

- `page.evaluate()` 返回调用执行的结果
- `page.evaluate_handle()`返回JSHandle

### 3.18.1 page.evaluate()

此方法返回`evaluate()` 返回执行JavaScript脚本的结果，使用示例

简单示例

```
print(page.evaluate("1 + 2")) # prints "3"
x = 10
print(page.evaluate(f"1 + {x}")) # prints "11"
```

也可以是执行一个函数

```
res = page.evaluate("() => 'Hello World!'",)
print(res) # Hello World!
```

函数也可以带上参数

```
res = page.evaluate("([a, b]) => a+b+'world'", ["hello", 'xx'])
print(res) # helloxxworld
```

如果传递给page.evaluate()的函数返回一个不可序列化的值, 则page.evaluate()解析为undefined

## 3.18.2 操作 web 网页示例

执行 `document.title` 获取页面的title

```
page.goto("https://www.baidu.com/")

title = page.evaluate('document.title')
print(title)
page.pause()
```

操作元素的几个常用JavaScript方法

1.输入框输入内容

```
document.getElementById('xx').value='yoyo';
```

2.清空输入框的内容

```
document.getElementById('xx').value='yoyo';
```

3.点击元素



```
document.getElementById('loginBtn').click();
```

## JavaScript 中定位元素的几种方法

- document.getElementById 通过元素的id 属性定位，返回单个元素对象
- document.getElementsByName 通过name属性定位，返回的是一组元素
- document.getElementsByClassName 通过class属性定位，返回一组元素
- document.getElementsByTagName 通过标签定位，返回一组元素
- document.querySelector 选择器定位，可以支持css语法，返回单个元素
- document.querySelectorAll 选择器定位，可以支持css语法，返回一组元素

其中只有document.getElementById 和 document.querySelector 返回的是单个元素，其它都是返回一组元素

## 使用 JavaScript 登录网站示例

```
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

page = browser.new_page()
page.goto("http://127.0.0.1/login.html")

js = """
document.getElementById('username').value='yoyo';
document.getElementById('password').value='*****';
document.getElementById('loginBtn').click();
"""

page.evaluate(js)
```

page.evaluate() 方法一般用于页面上操作元素，无法正常操作的情况，可以用执行JavaScript脚本协助解决。

## 3.18.3 page.evaluate\_handle()返回JSHandle

page.evaluate()和page.evaluate\_handle()之间的唯一区别是page.evaluate\_handle()返回JSHandle。

```
a_handle = page.evaluate_handle("document.body")
result_handle = page.evaluate_handle("body => body.innerHTML", a_handle)
print(result_handle.json_value())
result_handle.dispose()
```

## 3.19 locator.evaluate()对定位的元素执行JS脚本

page.evaluate() 是直接在页面对象上执行JavaScript脚本

locator.evaluate() 是对定位的元素执行JavaScript

locator.evaluate\_all() 对定位到的所有元素执行JavaScript

### 3.19.1 locator.evaluate() 对元素执行JavaScript

先用locator 方法定位到元素, 再对元素执行JavaScript

```
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

page = browser.new_page()
page.goto("http://127.0.0.1/login.html")

username = page.locator('#username')
# 输入框输入内容
username.evaluate('node => node.value="yoyo"')
# 获取输入框内容
input_value = username.evaluate('node => node.value')
print(input_value) # yoyo
```

### 3.19.2 locator.evaluate\_all() 执行全部元素

在页面中执行 JavaScript 代码, 将所有匹配的元素作为参数。

```
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch(
        headless=False
    )
    page = browser.new_page()
    page.goto("https://www.baidu.com/")

    links = page.locator('#s-top-left>a')
    # 定位全部元素
    res = links.evaluate_all('nodes => nodes.length')
    print(res) # 7
```

定位百度页面上的链接, 执行 `nodes.length` 获取元素个数

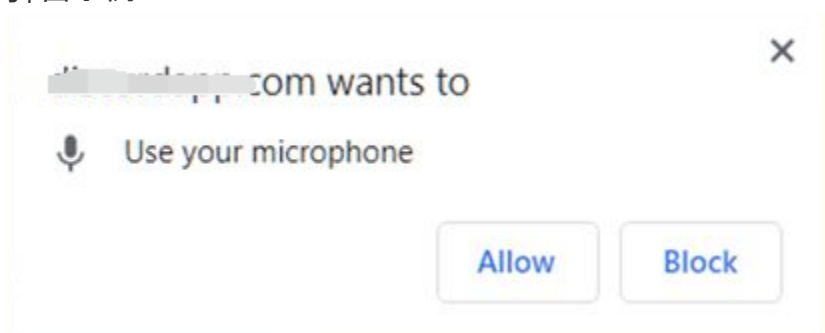
## 3.20 grant\_permissions 设置默认允许麦克风和摄像头等权限

有些场景在使用的时候, 会弹出一些权限框, 比如麦克风和摄像头等, 通过监听alert 是没法捕获的。

正确做法是给浏览器设置默认允许麦克风和摄像头等权限, 不让弹窗出来。使用context 的 `grant_permissions` 方法加权限。

### 3.20.1 权限框

弹窗示例





这种弹窗是权限窗，不是alert

### 3.20.2 解决办法

context 有个 `grant_permissions` 方法可以加权限，常见的权限如下

```

def grant_permissions(
    self, permissions: typing.List[str], *, origin: typing.Optional[str] = None
) -> None:
    """BrowserContext.grant_permissions

    Grants specified permissions to the browser context. Only grants corresponding permissions
    specified.

    Parameters
    -----
    permissions : List[str]
        A permission or an array of permissions to grant. Permissions can be one of the follow
        - ``'geolocation'``
        - ``'midi'``
        - ``'midi-sysex'`` (system-exclusive midi)
        - ``'notifications'``
        - ``'camera'``
        - ``'microphone'``
        - ``'background-sync'``
        - ``'ambient-light-sensor'``
        - ``'accelerometer'``
        - ``'gyroscope'``
        - ``'magnetometer'``
        - ``'accessibility-events'``
        - ``'clipboard-read'``
        - ``'clipboard-write'``
        - ``'payment-handler'``
    origin : Union[str, None]
        The [origin] to grant permissions to, e.g. "https://example.com".
    """

```

添加'camera', 'microphone' 权限示例

```

from playwright.sync_api import sync_playwright
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

with sync_playwright() as playwright:
    browser = playwright.chromium.launch(headless=False)
    context = browser.new_context()
    # 设置允许 'camera', 'microphone' 权限
    context.grant_permissions(['camera', 'microphone'])
    page = context.new_page()
    page.goto('https://www.xxx/xx.html')

```

# 网易云视频课程

网易云视频完整课程地址<https://study.163.com/course/courseMain.htm?courseId=1213382811&share=2&shareId=480000002230338>

