

## Playwright-docs-api 中文文档

[Playwright | Playwright Python 官方文档--需要科学上网](#)

version: 1.20

language: Python

拷贝时间: 2022.03.31

拷贝完整度: > 99%

机翻: >90%

翻译: 有道词典, 小玉的玉

备注: 该文档针对 **typora** 的使用方式做了部分优化(超链接显示, 锚点跳转 [具体优化方式](#))

## Playwright

Playwright 模块提供了一个方法来启动浏览器实例。下面是一个使用Playwright驱动自动化的典型例子:

- Sync

```
from playwright.sync_api import sync_playwright

def run(playwright):
    chromium = playwright.chromium # or "firefox" or "webkit".
    browser = chromium.launch()
    page = browser.new_page()
    page.goto("http://example.com")
    # other actions...
    browser.close()

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

- Async

```
import asyncio
from playwright.async_api import async_playwright

async def run(playwright):
    chromium = playwright.chromium # or "firefox" or "webkit".
    browser = await chromium.launch()
    page = await browser.new_page()
    await page.goto("http://example.com")
    # other actions...
    await browser.close()

async def main():
    async with async_playwright() as playwright:
        await run(playwright)
asyncio.run(main())
```

- [playwright.stop\(\)](#)
- [playwright.chromium](#)
- [playwright.devices](#)
- [playwright.firefox](#)
- [playwright.request](#)
- [playwright.selectors](#)
- [playwright.webkit](#)

## # playwright.stop() #

---

- returns: <[NoneType](#)> #

终止这个剧作家实例，以防它是绕过Python上下文管理器创建的。这在REPL应用程序中非常有用。

```
from playwright.sync_api import sync_playwright

playwright = sync_playwright().start()

browser = playwright.chromium.launch()
page = browser.new_page()
page.goto("http://whatsmyuseragent.org/")
page.screenshot(path="example.png")
browser.close()

playwright.stop()
```

## # playwright.chromium#

---

- type: <[BrowserType](#)>

这个对象可以用来启动或连接到Chromium，返回[Browser](#)的实例。

## # playwright.devices#

---

- type: <[Dict](#)>

返回与[browser.new\\_context\(\\*\\*kwargs\)](#)或[browser.new\\_page\(\\*\\*kwargs\)](#)一起使用的设备的字典。

- Sync

```
from playwright.sync_api import sync_playwright

def run(playwright):
    webkit = playwright.webkit
    iphone = playwright.devices["iPhone 6"]
    browser = webkit.launch()
    context = browser.new_context(**iphone)
    page = context.new_page()
    page.goto("http://example.com")
    # other actions...
    browser.close()
```

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

- Async

```
import asyncio
from playwright.async_api import async_playwright

async def run(playwright):
    webkit = playwright.webkit
    iphone = playwright.devices["iPhone 6"]
    browser = await webkit.launch()
    context = await browser.new_context(**iphone)
    page = await context.new_page()
    await page.goto("http://example.com")
    # other actions...
    await browser.close()

async def main():
    async with async_playwright() as playwright:
        await run(playwright)
    asyncio.run(main())
```

## # playwright.firefox#

---

- type: <[BrowserType](#)>

此对象可用于启动或连接到Firefox，返回[Browser](#)的实例。

## # playwright.request#

---

- type: <[APIRequest](#)>

公开可用于Web API测试的API。

## # playwright.selectors#

---

- type: <[Selectors](#)>

选择器可用于安装自定义选择器引擎。有关更多信息，请参见[Working with selectors](#)。

## # playwright.webkit#

---

- type: <[BrowserType](#)>

这个对象可以用来启动或连接到WebKit，返回[Browser](#)的实例。

### APIRequest

公开可用于Web API测试的API。每个剧作家浏览器上下文都有一个APIRequestContext实例，它与页面上下文共享cookie。也可以手动创建一个新的APIRequestContext实例。更多信息请参见[此处](#)。

- [api\\_request.new\\_context\(\\*\\*kwargs\)](#)

## # api\_request.new\_context(\*\*kwargs)#

---

- `base_url` <[str](#)> [api\\_request\\_context.get\(url, \\*\\*kwargs\)](#) 之类的方法,通过使用 [URL\(\)](#) 构造函数来构建相应的url. 例子:<#>
  - `baseURL= http://localhost:3000` 时发送请求至 `/bar.html` 结果是 `http://localhost:3000/bar.html`
  - `baseURL= http://localhost:3000/foo/` 时发送请求至 `./bar.html` 结果是 `http://localhost:3000/foo/bar.html`
  - `baseURL= http://localhost:3000/foo` 时发送请求至(末尾不加斜杠) `./bar.html` 结果是 `http://localhost:3000/bar.html`
- `extra_http_headers` <[Dict](#)[[str](#), [str](#)]> 一个包含附加HTTP头的对象，每个请求都要发送。<#>
- `http_credentials` <[Dict](#)> [HTTP认证凭据](#)。<#>

- `username` `<str>`
- `password` `<str>`
- `ignore_https_errors` `<bool>` 发送网络请求时是否忽略HTTPS错误。默认值为 `false`.#
- `proxy` `<Dict>` 网络代理设置.#
  - `server` `<str>` 所有请求使用的代理。支持HTTP代理和SOCKS代理，例如: `http://myproxy.com:3128` or `socks5://myproxy.com:3128`.缩写形式 `myproxy.com:3128` 被认为是一个HTTP代理.
  - `bypass` `<str>` 可选，用逗号分隔的域，绕过代理，例如 `".com, chromium.org, .domain.com"`.
  - `username` `<str>`可选 username，当HTTP代理需要鉴权时使用.
  - `password` `<str>` 当HTTP代理需要鉴权时可选密码.
- `storage_state` `<Union[str, pathlib.Path]|Dict>` 用给定的存储状态填充上下文。这个选项可以用来通过 `browser_context.storage_state(**kwargs)` or `api_request_context.storage_state(**kwargs)`. 获得的登录信息初始化上下文。可以是保存存储的文件路径，也可以是 `browser_context.storage_state(**kwargs)` or `api_request_context.storage_state(**kwargs)` 方法返回的值.#
  - `cookies` `<List[Dict]>`
    - `name` `<str>`
    - `value` `<str>`
    - `domain` `<str>`
    - `path` `<str>`
    - `expires` `<float>` Unix 时间，单位为秒.
    - `httpOnly` `<bool>`
    - `secure` `<bool>`
    - `sameSite` `<"Strict"|"Lax"|"None">`
  - `origins` `<List[Dict]>`
    - `origin` `<str>`
    - `localStorage` `<List[Dict]>`
      - `name` `<str>`
      - `value` `<str>`
- `timeout` `<float>` 等待响应的最大时间，单位为毫秒。默认为 `30000` (30 seconds). 传递 `0` 以禁用超时.#
- `user_agent` `<str>` 在此上下文中使用的特定用户代理.#
- `returns:` `<APIRequestContext>` #

创建 [APIRequestContext](#) 新实例。

## APIRequestContext

此API用于Web API测试。您可以使用它来触发API端点、配置微服务、为您的端到端测试准备环境或服务。当在 [Page](#) 或 [BrowserContext](#) 中使用时，这个API将自动使用相应的 [BrowserContext](#) 中的cookie。这意味着如果您使用这个API登录，您的e2e测试将被登录，反之亦然。

- Sync

```
import os
from playwright.sync_api import sync_playwright

REPO = "test-repo-1"
USER = "github-username"
API_TOKEN = os.getenv("GITHUB_API_TOKEN")

with sync_playwright() as p:
    # This will launch a new browser, create a context and
    # page. When making HTTP
    # requests with the internal APIRequestContext (e.g.
    # `context.request` or `page.request`)
    # it will automatically set the cookies to the browser
    # page and vice versa.
    browser = p.chromium.launch()
    context =
    browser.new_context(base_url="https://api.github.com")
    api_request_context = context.request
    page = context.new_page()

    # Alternatively you can create a APIRequestContext
    # manually without having a browser context attached:
    # api_request_context =
    # p.request.new_context(base_url="https://api.github.com")

    # Create a repository.
    response = api_request_context.post(
        "/user/repos",
```

```

        headers={
            "Accept": "application/vnd.github.v3+json",
            # Add GitHub personal access token.
            "Authorization": f"token {API_TOKEN}",
        },
        data={"name": REPO},
    )
    assert response.ok
    assert response.json()["name"] == REPO

    # Delete a repository.
    response = api_request_context.delete(
        f"/repos/{USER}/{REPO}",
        headers={
            "Accept": "application/vnd.github.v3+json",
            # Add GitHub personal access token.
            "Authorization": f"token {API_TOKEN}",
        },
    )
    assert response.ok
    assert await response.body() == '{"status": "ok"}'

```

- Async

```

import os
import asyncio
from playwright.async_api import async_playwright, Playwright

REPO = "test-repo-1"
USER = "github-username"
API_TOKEN = os.getenv("GITHUB_API_TOKEN")

async def run(playwright: Playwright):
    # This will launch a new browser, create a context and
    # page. When making HTTP
    # requests with the internal APIRequestContext (e.g.
    # `context.request` or `page.request`)
    # it will automatically set the cookies to the browser
    # page and vice versa.
    browser = await playwright.chromium.launch()
    context = await
    browser.new_context(base_url="https://api.github.com")

```



```

api_request_context = context.request
page = await context.new_page()

# Alternatively you can create a APIRequestContext
manually without having a browser context attached:
# api_request_context = await
playwright.request.new_context(base_url="https://api.github.co
m")

# Create a repository.
response = await api_request_context.post(
    "/user/repos",
    headers={
        "Accept": "application/vnd.github.v3+json",
        # Add GitHub personal access token.
        "Authorization": f"token {API_TOKEN}",
    },
    data={"name": REPO},
)
assert response.ok
assert response.json()["name"] == REPO

# Delete a repository.
response = await api_request_context.delete(
    f"/repos/{USER}/{REPO}",
    headers={
        "Accept": "application/vnd.github.v3+json",
        # Add GitHub personal access token.
        "Authorization": f"token {API_TOKEN}",
    },
)
assert response.ok
assert await response.body() == '{"status": "ok"}'

async def main():
    async with async_playwright() as playwright:
        await run(playwright)

asyncio.run(main())

```

## # api\_request\_context.delete(url, \*\*kwargs)#

---

- `url` <str> 目标URL. #
- `data` <str|bytes|Serializable> 允许设置请求的post数据。如果data参数是一个对象，它将被序列化为json字符串，如果没有显式设置，`content-type` 头将被设置为 `application/json`。否则，如果没有显式设置 `content-type` 头将被设置为 `application/octet-stream`。 #
- `fail_on_status_code` <bool> • 是否对2xx和3xx以外的响应码抛出异常。默认情况下，返回所有状态码的响应对象. #
- `form` <Dict[str, str|float|bool]> 提供一个对象，该对象将使用 `application/x-www-form-urlencoded` 编码序列化为html form，并作为请求体发送。如果指定了这个参数，除非明确提供，否则 `content-type` 将被设置为 `application/x-www-form-urlencoded`。 #
- `headers` <Dict[str, str]> 允许设置HTTP头. #
- `ignore_https_errors` <bool> 发送网络请求时是否忽略HTTPS错误。默认值为 `false`。 #
- `multipart` <Dict[str, str|float|bool|[ReadStream]] Dict> 提供一个对象，该对象将使用 `multipart/form-data` 编码序列化为html形式，并作为请求体发送。如果指定了该参数，除非明确提供，否则 `content-type` 将被设置为 `multipart/form-data`。文件值可以作为fs来传递，`fs.ReadStream` 或作为文件类对象，包含文件名、mime类型及其内容. #
  - `name` <str> 文件名
  - `mimeType` <str> 文件类型
  - `buffer` <bytes> 文件内容
- `params` <Dict[str, str|float|bool]> 查询参数. #
- `timeout` <float> 请求超时时间，单位为毫秒。默认为 `30000` (30 seconds). 传递 `0` 以禁用超时. #
- returns: <APIResponse> #

发送HTTP(S) [DELETE](#) 请求并返回响应。该方法将从上下文填充请求cookie，并从响应更新上下文cookie。该方法将自动遵循重定向。

## # api\_request\_context.dispose()#

---

- returns: `<NoneType>` <#>

`api_request_context.get(url, **kwargs)` 返回的所有响应和类似的方法都存储在内存中，这样你以后就可以调用 `api_response.body()`。此方法丢弃所有存储的响应，并使 `api_response.body()` 抛出“Response dispose”错误。

## # `api_request_context.fetch(url_or_request, **kwargs)` <#>

---

- `url_or_request` `<str|Request>` 获取所有参数的目标URL或请求。 <#>
- `data` `<str|bytes|Serializable>` 允许设置请求的post数据。如果data参数是一个对象，它将被序列化为json字符串，如果没有显式设置，`content-type` 将被设置为 `application/json`，否则，如果没有显式设置，`content-type` 将被设置为 `application/octet-stream`。 <#>
- `fail_on_status_code` `<bool>` 是否对2xx和3xx以外的响应码排除。默认情况下，返回所有状态码的响应对象。 <#>
- `form` `<Dict[str, str|float|bool]>` 提供一个对象，该对象将使用 `application/x-www-form-urlencoded` 编码序列化为html form，并作为请求体发送。如果指定了这个参数，除非明确提供，否则 `content-type` 将被设置为 `application/x-www-form-urlencoded`。 <#>
- `headers` `<Dict[str, str]>` 允许设置HTTP头。 <#>
- `ignore_https_errors` `<bool>` 发送网络请求时是否忽略HTTPS错误。默认值为 `false`。 <#>
- `method` `<str>` 改变了获取方法 (e.g. `PUT` or `POST`)。如果未指定，则使用GET方法。 <#>
- `multipart` `<Dict[str, str|float|bool|[ReadStream]]Dict>` 提供一个对象，该对象将使用 `multipart/form-data` 编码序列化为html形式，并作为请求体发送。如果指定了该参数，除非明确提供，否则 `content-type` 将被设置为 `multipart/form-data`。文件值可以作为fs来传递，`fs.ReadStream` 或作为文件类对象，包含文件名、mime类型及其内容。 <#>
  - `name` `<str>` 文件名
  - `mimeType` `<str>` 文件类型
  - `buffer` `<bytes>` 文件内容
- `params` `<Dict[str, str|float|bool]>` 查询参数。 <#>

- `timeout` <float> 请求超时时间，单位为毫秒。默认为 `30000` (30 seconds). 传递 `0` 以禁用超时.#
- returns: <APIResponse>.#

发送HTTP(S)请求并返回响应。该方法将从上下文填充请求cookie，并从响应更新上下文cookie。该方法将自动遵循重定向。

## # api\_request\_context.get(url, \*\*kwargs)#

---

- `url` <str> 目标 URL.#
- `fail_on_status_code` <bool> 是否对2xx和3xx以外的响应码排除。默认情况下，返回所有状态码的响应对象.#
- `headers` <Dict[str, str]> 允许设置HTTP头.#
- `ignore_https_errors` <bool> 发送网络请求时是否忽略HTTPS错误。默认值为 `false`.#
- `params` <Dict[str, str|float|bool]> 查询参数.#
- `timeout` <float> 请求超时时间，单位为毫秒。默认为 `30000` (30 seconds). 传递 `0` 以禁用超时.#
- returns: <APIResponse>.#

发送HTTP(S) GET请求并返回响应。该方法将从上下文填充请求cookie，并从响应更新上下文cookie。该方法将自动遵循重定向。

## # api\_request\_context.head(url, \*\*kwargs)#

---

- `url` <str> 目标 URL.#
- `fail_on_status_code` <bool> 是否对2xx和3xx以外的响应码排除。默认情况下，返回所有状态码的响应对象.#
- `headers` <Dict[str, str]> 允许设置HTTP头.#
- `ignore_https_errors` <bool> 发送网络请求时是否忽略HTTPS错误。默认值为 `false`.#
- `params` <Dict[str, str|float|bool]> 查询参数.#
- `timeout` <float> 请求超时时间，单位为毫秒。默认为 `30000` (30 seconds). 传递 `0` 以禁用超时.#
- returns: <APIResponse>.#

发送HTTP(S) HEAD 请求并返回其响应。该方法将从上下文填充请求cookie，并从响应更新上下文cookie。该方法将自动遵循重定向。

## # api\_request\_context.patch(url, \*\*kwargs)#

---

- `url` <str> 目标 URL. #
- `data` <str|bytes|Serializable> 允许设置请求的post数据。如果data参数是一个对象，它将被序列化为json字符串，如果没有显式设置，`content-type` 将被设置为 `application/json` ,否则，如果没有显式设置，`content-type` 将被设置为 `application/octet-stream` .#
- `fail_on_status_code` <bool> 是否对2xx和3xx以外的响应码排除。默认情况下，返回所有状态码的响应对象. #
- `form` <Dict[str, str|float|bool]> 提供一个对象，该对象将使用 `application/x-www-form-urlencoded` 编码序列化为html form，并作为请求体发送。如果指定了这个参数，除非明确提供，否则 `content-type` 将被设置为 `application/x-www-form-urlencoded` .#
- `headers` <Dict[str, str]> 允许设置HTTP头. #
- `ignore_https_errors` <bool> 发送网络请求时是否忽略HTTPS错误。默认值为 `false` .#
- `multipart` <Dict[str, str|float|bool|[ReadStream]]Dict> 提供一个对象，该对象将使用 `multipart/form-data` 编码序列化为html形式，并作为请求体发送。如果指定了该参数，除非明确提供，否则 `content-type` 将被设置为 `multipart/form-data` . 文件值可以作为fs来传递，`fs.ReadStream` 或作为文件类对象，包含文件名、mime类型及其内容. #
  - `name` <str> 文件名
  - `mimeType` <str> 文件类型
  - `buffer` <bytes> 文件内容
- `params` <Dict[str, str|float|bool]> 查询参数. #
- `timeout` <float> 请求超时时间，单位为毫秒。默认为 `30000` (30 seconds). 传递 `0` 以禁用超时. #
- returns: <APIResponse> #

发送HTTP(S) [PATCH](#) 请求并返回响应。该方法将从上下文填充请求cookie，并从响应更新上下文cookie。该方法将自动遵循重定向。

## # api\_request\_context.post(url, \*\*kwargs)#

---

- `url` <`str`> 目标 URL. <#>
- `data` <`str`|`bytes`|`Serializable`> 允许设置请求的post数据。如果data参数是一个对象，它将被序列化为json字符串，如果没有显式设置，`content-type` 将被设置为 `application/json` ,否则，如果没有显式设置，`content-type` 将被设置为 `application/octet-stream` . <#>
- `fail_on_status_code` <`bool`> 是否对2xx和3xx以外的响应码排除。默认情况下，返回所有状态码的响应对象. <#>
- `form` <`Dict`[`str`, `str`|`float`|`bool`]> 提供一个对象，该对象将使用 `application/x-www-form-urlencoded` 编码序列化为html form，并作为请求体发送。如果指定了这个参数，除非明确提供，否则 `content-type` 将被设置为 `application/x-www-form-urlencoded` . <#>
- `headers` <`Dict`[`str`, `str`]> 允许设置HTTP头. <#>
- `ignore_https_errors` <`bool`> 发送网络请求时是否忽略HTTPS错误。默认值为 `false` . <#>
- `multipart` <`Dict`[`str`, `str`|`float`|`bool`][`ReadStream`]|`Dict`> 提供一个对象，该对象将使用 `multipart/form-data` 编码序列化为html形式，并作为请求体发送。如果指定了该参数，除非明确提供，否则 `content-type` 将被设置为 `multipart/form-data` . 文件值可以作为fs来传递，`fs.ReadStream` 或作为文件类对象，包含文件名、mime类型及其内容. <#>
  - `name` <`str`> 文件名
  - `mimeType` <`str`> 文件类型
  - `buffer` <`bytes`> 文件内容
- `params` <`Dict`[`str`, `str`|`float`|`bool`]> 查询参数. <#>
- `timeout` <`float`> 请求超时时间，单位为毫秒。默认为 `30000` (30 seconds). 传递 `0` 以禁用超时. <#>
- returns: <`APIResponse`> <#>

发送HTTP(S) [POST](#) 请求并返回响应。该方法将从上下文填充请求cookie，并从响应更新上下文cookie。该方法将自动遵循重定向。

## [# api\\_request\\_context.put\(url, \\*\\*kwargs\) #](#)

- `url` <`str`> 目标 URL. <#>

- `data` `<str|bytes|Serializable>` 允许设置请求的post数据。如果data参数是一个对象，它将被序列化为json字符串，如果没有显式设置，`content-type` 将被设置为 `application/json` ,否则，如果没有显式设置，`content-type` 将被设置为 `application/octet-stream` .#
- `fail_on_status_code` `<bool>` 是否对2xx和3xx以外的响应码排除。默认情况下，返回所有状态码的响应对象.#
- `form` `<Dict[str, str|float|bool]>`提供一个对象，该对象将使用 `application/x-www-form-urlencoded` 编码序列化为html form，并作为请求体发送。如果指定了这个参数，除非明确提供，否则 `content-type` 将被设置为 `application/x-www-form-urlencoded` .#
- `headers` `<Dict[str, str]>` 允许设置HTTP头.#
- `ignore_https_errors` `<bool>` 发送网络请求时是否忽略HTTPS错误。默认值为 `false` .#
- `multipart` `<Dict[str, str|float|bool|[ReadStream]]Dict>` 提供一个对象，该对象将使用 `multipart/form-data` 编码序列化为html形式，并作为请求体发送。如果指定了该参数，除非明确提供，否则 `content-type` 将被设置为 `multipart/form-data` . 文件值可以作为fs来传递，`fs.ReadStream` 或作为文件类对象，包含文件名、mime类型及其内容.#
  - `name` `<str>` 文件名
  - `mimeType` `<str>` 文件类型
  - `buffer` `<bytes>` 文件内容
- `params` `<Dict[str, str|float|bool]>` 查询参数 #
- `timeout` `<float>` 请求超时时间，单位为毫秒。默认为 `30000` (30 seconds). 传递 `0` 以禁用超时.#
- returns: `<APIResponse>` #

#

## api\_request\_context.storage\_state(\*\*kwargs) s)#

---

- `path` `<Union[str, pathlib.Path]>` 存储状态保存到的文件路径. 如果 `path` 是一个相对路径，那么它是相对于当前工作目录解析的。如果没有提供路径，存储状态仍然返回，但不会保存到磁盘.#
- returns: `<Dict>` #



- `cookies` <[List](#)[[Dict](#)]>
  - `name` <[str](#)>
  - `value` <[str](#)>
  - `domain` <[str](#)>
  - `path` <[str](#)>
  - `expires` <[float](#)> Unix 时间，单位为秒.
  - `httpOnly` <[bool](#)>
  - `secure` <[bool](#)>
  - `sameSite` <"Strict"|"Lax"|"None">
- `origins` <[List](#)[[Dict](#)]>
  - `origin` <[str](#)>
  - `localStorage` <[List](#)[[Dict](#)]>
    - `name` <[str](#)>
    - `value` <[str](#)>

返回此请求上下文的存储状态，如果它被传递给构造函数，则包含当前cookie和本地存储快照。

## APIResponse

[APIResponse](#) 类表示 [api\\_request\\_context.get\(url, \\*\\*kwargs\)](#) 等类似的方法返回的响应

- Sync

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    context = playwright.request.new_context()
    response = context.get("https://example.com/user/repos")
    assert response.ok
    assert response.status == 200
    assert response.headers["content-type"] ==
"application/json; charset=utf-8"
    assert response.json()["name"] == "foobar"
    assert response.body() == '{"status": "ok"}'
```



- Async

```
import asyncio
from playwright.async_api import async_playwright, Playwright

async def run(playwright: Playwright):
    context = await playwright.request.new_context()
    response = await
context.get("https://example.com/user/repos")
    assert response.ok
    assert response.status == 200
    assert response.headers["content-type"] =
"application/json; charset=utf-8"
    assert response.json()["name"] == "foobar"
    assert await response.body() == '{"status": "ok"}'

async def main():
    async with async_playwright() as playwright:
        await run(playwright)

asyncio.run(main())
```

## # api\_response.body()#

---

- returns: <[bytes](#)>#

返回带有响应体的缓冲区。

## # api\_response.dispose()#

---

- returns: <[NoneType](#)>#

处理此响应的正文。如果没有调用，则主体将留在内存中，直到上下文关闭。

## # api\_response.headers#

---

- returns: `<Dict[str, str]>#`

具有与此响应关联的所有响应HTTP头的对象。

## # api\_response.headers\_array#

---

- returns: `<List[Dict]>#`
  - `name` `<str>` 头属性名称.
  - `value` `<str>` 头属性值.

包含与此响应关联的所有请求HTTP头的数组。头属性名称不是小写的。具有多个条目的头属性，例如 `Set-Cookie`，在数组中出现多次。

## # api\_response.json()#

---

- returns: `<Serializable>#`

返回响应体的JSON表示。

如果响应体不能通过 `JSON.parse` 进行解析，则该方法将排除。

## # api\_response.ok#

---

- returns: `<bool>#`

包含一个布尔值，说明响应是否成功(状态在200-299之间)。

## # api\_response.status#

---

- returns: `<int>#`

包含响应的状态码(例如，200表示成功)。

## # api\_response.status\_text#

---

- returns: <str>#

包含响应的状态文本(例如, 通常一个“OK”表示成功)。

## # api\_response.text()#

---

- returns: <str>#

返回响应体的文本表示形式。

## # api\_response.url#

---

- returns: <str>#

包含响应的URL。

## Assertions

Playwright 为您提供了web优先的断言, 提供了创建断言的方便方法, 这些断言将等待并重试, 直到满足预期的条件。

- Sync

```
from playwright.sync_api import Page, expect

def test_status_becomes_submitted(page: Page) → None:
    # ..
    page.click("#submit-button")
    expect(page.locator(".status")).to_have_text("Submitted")
```

- Async

```
from playwright.async_api import Page, expect

async def test_status_becomes_submitted(page: Page) → None:
    # ..
    await page.click("#submit-button")
    await
    expect(page.locator(".status")).to_have_text("Submitted")
```

Playwright 会用选择器 `.status` 重新测试节点，直到取回的节点有 `"Submitted"` 文本。它将重新获取节点并一遍又一遍地检查它，直到满足条件或到达超时。您可以将此超时作为一个选项传递。

默认情况下，断言的超时被设置为5秒。

#

## `expect(locator).not_to_be_checked(**kwargs)`

---

- `timeout` `<float>` 重试断言的时间. #
- returns: `<NoneType>` #

与 `expect(locator).to_be_checked(**kwargs)` 相反.

#

## `expect(locator).not_to_be_disabled(**kwargs)`

---

- `timeout` `<float>` 重试断言的时间. #
- returns: `<NoneType>` #

与 `expect(locator).to_be_disabled(**kwargs)` 相反.

#

## `expect(locator).not_to_be_editable(**kwargs)` #

---

- `timeout` <float> 重试断言的时间.#
- returns: <NoneType>#

与 `expect(locator).to_be_editable(**kwargs)` 相反.

#

## `expect(locator).not_to_be_empty(**kwargs)` #

---

- `timeout` <float> 重试断言的时间.#
- returns: <NoneType>#

与 `expect(locator).to_be_empty(**kwargs)` 相反.

#

## `expect(locator).not_to_be_enabled(**kwargs)` #

---

- `timeout` <float> 重试断言的时间.#
- returns: <NoneType>#

与 `expect(locator).to_be_enabled(**kwargs)` 相反.

#

## `expect(locator).not_to_be_focused(**kwargs)` #

---

- `timeout` <float> 重试断言的时间.#
- returns: <NoneType>#

与 [expect\(locator\).to\\_be\\_focused\(\\*\\*kwargs\)](#) 相反.

#

## `expect(locator).not_to_be_hidden(**kwargs)` [\)#](#)

---

- `timeout` <float> 重试断言的时间.#
- returns: <NoneType>#

与 [expect\(locator\).to\\_be\\_hidden\(\\*\\*kwargs\)](#) 相反.

#

## `expect(locator).not_to_be_visible(**kwargs)` [\)#](#)

---

- `timeout` <float> 重试断言的时间.#
- returns: <NoneType>#

与 [expect\(locator\).to\\_be\\_visible\(\\*\\*kwargs\)](#) 相反.

#

## `expect(locator).not_to_contain_text(expected, **kwargs)` [\)#](#)

---

- `expected` <str|Pattern|List[str|Pattern]> 期望的子字符串或  
RegExp或它们的列表.#
- `timeout` <float> 重试断言的时间.#
- `use_inner_text` <bool> 检索DOM节点文本时使用  
`element.innerText` 而不是 `element.textContent` .#
- returns: <NoneType>#

与之相反 [expect\(locator\).to\\_contain\\_text\(expected, \\*\\*kwargs\)](#).

#

## **expect(locator).not\_to\_have\_attribute(name, value, \*\*kwargs)#**

---

- **name** <[str](#)> 属性名.#
- **value** <[str](#)|[Pattern](#)> 期望的属性值.#
- **timeout** <[float](#)> 重试断言的时间.#
- returns: <[NoneType](#)>#

与之相反 [expect\(locator\).to\\_have\\_attribute\(name, value, \\*\\*kwargs\)](#).

#

## **expect(locator).not\_to\_have\_class(expected, \*\*kwargs)#**

---

- **expected** <[str](#)|[Pattern](#)|[List](#)[[str](#)|[Pattern](#)]> 期望的类或RegExp或它们的列表.#
- **timeout** <[float](#)> 重试断言的时间.#
- returns: <[NoneType](#)>#

与之相反 [expect\(locator\).to\\_have\\_class\(expected, \\*\\*kwargs\)](#).

#

## **expect(locator).not\_to\_have\_count(count, \*\*kwargs)#**

---

- **count** <[int](#)> 期望的计数.#
- **timeout** <[float](#)> 重试断言的时间.#
- returns: <[NoneType](#)>#

与之相反 [expect\(locator\).to\\_have\\_count\(count, \\*\\*kwargs\)](#).

## # expect(locator).not\_to\_have\_css(name, value, \*\*kwargs) #

---

- `name` <[str](#)> CSS 属性名. #
- `value` <[str](#)|[Pattern](#)> CSS 属性值. #
- `timeout` <[float](#)> 重试断言的时间. #
- returns: <[NoneType](#)> #

与之相反 [expect\(locator\).to\\_have\\_css\(name, value, \\*\\*kwargs\)](#).

## # expect(locator).not\_to\_have\_id(id, \*\*kwargs) #

---

- `id` <[str](#)|[Pattern](#)> 元素 id. #
- `timeout` <[float](#)> 重试断言的时间. #
- returns: <[NoneType](#)> #

与之相反 [expect\(locator\).to\\_have\\_id\(id, \\*\\*kwargs\)](#).

## # expect(locator).not\_to\_have\_js\_property(name, value, \*\*kwargs) #

---

- `name` <[str](#)> 属性名. #
- `value` <[Any](#)> 属性值. #
- `timeout` <[float](#)> 重试断言的时间. #
- returns: <[NoneType](#)> #

与之相反 [expect\(locator\).to\\_have\\_js\\_property\(name, value, \\*\\*kwargs\)](#).



#

## **expect(locator).not\_to\_have\_text(expected, \*\*kwargs)#**

---

- `expected` <[str](#)|[Pattern](#)|[List](#)[[str](#)|[Pattern](#)]> 期望的子字符串或RegEx或它们的列表.#
- `timeout` <[float](#)> 重试断言的时间.#
- `use_inner_text` <[bool](#)> 检索DOM节点文本时使用 `element.innerText` 而不是 `element.textContent` .#
- returns: <[NoneType](#)>#

与之相反 [expect\(locator\).to\\_have\\_text\(expected, \\*\\*kwargs\)](#).

## **# expect(locator).not\_to\_have\_value(value, \*\*kwargs)#**

---

- `value` <[str](#)|[Pattern](#)> 期望值.#
- `timeout` <[float](#)> 重试断言的时间.#
- returns: <[NoneType](#)>#

与之相反 [expect\(locator\).to\\_have\\_value\(value, \\*\\*kwargs\)](#).

#

## **expect(locator).to\_be\_checked(\*\*kwargs)#**

---

- `checked` <[bool](#)>#
- `timeout` <[float](#)> 重试断言的时间.#
- returns: <[NoneType](#)>#

确保 [Locator](#) 指向选中的输入。

- Sync

```
from playwright.sync_api import expect

locator = page.locator(".subscribe")
expect(locator).to_be_checked()
```

- Async

```
from playwright.async_api import expect

locator = page.locator(".subscribe")
await expect(locator).to_be_checked()
```

#

## expect(locator).to\_be\_disabled(\*\*kwargs)#

- `timeout` <float> 重试断言的时间.#
- returns: <NoneType>#

确保 [Locator](#) 指向一个禁用的元素。

- Sync

```
from playwright.sync_api import expect

locator = page.locator("button.submit")
expect(locator).to_be_disabled()
```

- Async

```
from playwright.async_api import expect

locator = page.locator("button.submit")
await expect(locator).to_be_disabled()
```

#

## expect(locator).to\_be\_editable(\*\*kwargs)#

- `timeout` <float> 重试断言的时间.#
- returns: <NoneType>#

确保 [Locator](#) 指向一个可编辑的元素。

- Sync

```
from playwright.sync_api import expect

locator = page.locator(".input")
expect(locator).to_be_editable()
```

- Async

```
from playwright.async_api import expect

locator = page.locator(".input")
await expect(locator).to_be_editable()
```

## # expect(locator).to\_be\_empty(\*\*kwargs)#

- `timeout` <float> 重试断言的时间.#
- returns: <NoneType>#

确保 [Locator](#) 指向空的、可编辑元素或没有文本的DOM节点。

- Sync

```
from playwright.sync_api import expect

locator = page.locator("div.warning")
expect(locator).to_be_empty()
```

- Async

```
from playwright.async_api import expect

locator = page.locator("div.warning")
await expect(locator).to_be_empty()
```

#

## expect(locator).to\_be\_enabled(\*\*kwargs)#

- `timeout` <float> 重试断言的时间.#
- returns: <NoneType>#

确保 [Locator](#) 指向一个启用的元素。

- Sync

```
from playwright.sync_api import expect

locator = page.locator("button.submit")
expect(locator).to_be_enabled()
```

- Async

```
from playwright.async_api import expect

locator = page.locator("button.submit")
await expect(locator).to_be_enabled()
```

#

## expect(locator).to\_be\_focused(\*\*kwargs)#

- `timeout` <float> 重试断言的时间.#
- returns: <NoneType>#

确保 [Locator](#) 指向一个集中的DOM节点。

- Sync

```
from playwright.sync_api import expect

locator = page.locator('input')
expect(locator).to_be_focused()
```

- Async

```
from playwright.async_api import expect

locator = page.locator('input')
await expect(locator).to_be_focused()
```

## # expect(locator).to\_be\_hidden(\*\*kwargs)#

- `timeout` <float> 重试断言的时间.#
- returns: <NoneType>#

确保 [Locator](#) 指向一个隐藏的DOM节点，这与 [visible](#) 相反。

- Sync

```
from playwright.sync_api import expect

locator = page.locator('.my-element')
expect(locator).to_be_hidden()
```

- Async

```
from playwright.async_api import expect

locator = page.locator('.my-element')
await expect(locator).to_be_hidden()
```

## # `expect(locator).to_be_visible(**kwargs)` #

---

- `timeout` <float> 重试断言的时间. #
- returns: <NoneType> #

确保 [Locator](#) 指向一个 [可见 visible](#) 的DOM节点。

- Sync

```
from playwright.sync_api import expect

locator = page.locator('.my-element')
expect(locator).to_be_visible()
```

- Async

```
from playwright.async_api import expect

locator = page.locator('.my-element')
await expect(locator).to_be_visible()
```

## #

## `expect(locator).to_contain_text(expected, **kwargs)` #

---

- `expected` <[str](#)|[Pattern](#)|[List](#)[[str](#)|[Pattern](#)]> 期望的子字符串或 [RegExp](#)或它们的列表.<#>
- `timeout` <[float](#)> 重试断言的时间.<#>
- `use_inner_text` <[bool](#)> 当检索DOM节点文本时使用 `element.innerText` 而不是 `element.textContent` .<#>
- `returns`: <[NoneType](#)><#>

确保 [Locator](#) 指向包含给定文本的元素。您也可以为该值使用正则表达式。

- Sync

```
import re
from playwright.sync_api import expect

locator = page.locator('.title')
expect(locator).to_contain_text("substring")
expect(locator).to_contain_text(re.compile(r"\d messages"))
```

- Async

```
import re
from playwright.async_api import expect

locator = page.locator('.title')
await expect(locator).to_contain_text("substring")
await expect(locator).to_contain_text(re.compile(r"\d
messages"))
```

注意，如果array是作为预期值传递的，那么整个元素列表都可以被断言：

- Sync

```
import re
from playwright.sync_api import expect

locator = page.locator("list > .list-item")
expect(locator).to_contain_text(["Text 1", "Text 4", "Text
5"])
```

- Async

```
import re
from playwright.async_api import expect

locator = page.locator("list > .list-item")
await expect(locator).to_contain_text(["Text 1", "Text 4",
"Text 5"])
```

## # expect(locator).to\_have\_attribute(name, value, \*\*kwargs) #

---

- **name** <str> 属性名. #
- **value** <str|Pattern> 期望的属性值. #
- **timeout** <float> 重试断言的时间. #
- returns: <NoneType> #

确保 [Locator](#) 指向具有给定属性的元素。

- Sync

```
from playwright.sync_api import expect

locator = page.locator("input")
expect(locator).to_have_attribute("type", "text")
```

- Async

```
from playwright.async_api import expect

locator = page.locator("input")
await expect(locator).to_have_attribute("type", "text")
```



## # expect(locator).to\_have\_class(expected, \*\*kwargs) #

---

- `expected` <[str](#)|[Pattern](#)|[List](#)[[str](#)|[Pattern](#)]> • 期望的类或RegExp或它们的列表.#
- `timeout` <[float](#)> 重试断言的时间.#
- returns: <[NoneType](#)>#

确保 [Locator](#) 指向具有给定CSS类的元素。

- Sync

```
from playwright.sync_api import expect

locator = page.locator("#component")
expect(locator).to_have_class(re.compile(r"selected"))
```

- Async

```
from playwright.async_api import expect

locator = page.locator("#component")
await expect(locator).to_have_class(re.compile(r"selected"))
```

注意，如果array是作为预期值传递的，那么整个元素列表都可以被断言：

- Sync

```
from playwright.sync_api import expect

locator = page.locator("list > .component")
expect(locator).to_have_class(["component", "component
selected", "component"])
```

- Async

```
from playwright.async_api import expect

locator = page.locator("list > .component")
await expect(locator).to_have_class(["component", "component
selected", "component"])
```

## # expect(locator).to\_have\_count(count, \*\*kwargs) #

---

- `count` <int> 期望的计数. #
- `timeout` <float> 重试断言的时间. #
- returns: <NoneType> #

确保 [Locator](#) 解析为精确数量的DOM节点。

- Sync

```
from playwright.sync_api import expect

locator = page.locator("list > .component")
expect(locator).to_have_count(3)
```

- Async

```
from playwright.async_api import expect

locator = page.locator("list > .component")
await expect(locator).to_have_count(3)
```

## # expect(locator).to\_have\_css(name, value, \*\*kwargs) #

---

- `name` <[str](#)> CSS 属性名.<#>
- `value` <[str](#)|[Pattern](#)> CSS属性值.<#>
- `timeout` <[float](#)> 重试断言的时间.<#>
- returns: <[NoneType](#)><#>

确保 [Locator](#) 解析为具有给定计算的CSS样式的元素。

- Sync

```
from playwright.sync_api import expect

locator = page.locator("button")
expect(locator).to_have_css("display", "flex")
```

- Async

```
from playwright.async_api import expect

locator = page.locator("button")
await expect(locator).to_have_css("display", "flex")
```

## # expect(locator).to\_have\_id(id, \*\*kwargs)#

- `id` <[str](#)|[Pattern](#)> 元素 id.<#>
- `timeout` <[float](#)> 重试断言的时间.<#>
- returns: <[NoneType](#)><#>

确保 [Locator](#) 指向具有给定DOM节点ID的元素。

- Sync

```
from playwright.sync_api import expect

locator = page.locator("input")
expect(locator).to_have_id("lastname")
```

- Async

```
from playwright.async_api import expect

locator = page.locator("input")
await expect(locator).to_have_id("lastname")
```

#

## `expect(locator).to_have_js_property(name, value, **kwargs)`

---

- `name` <str> 属性名 #
- `value` <Any> 属性值. #
- `timeout` <float> 重试断言的时间. #
- returns: <NoneType> #

确保 [Locator](#) 指向具有给定JavaScript属性的元素。请注意，该属性可以是基本类型，也可以是可序列化的普通JavaScript对象。

- Sync

```
from playwright.sync_api import expect

locator = page.locator(".component")
expect(locator).to_have_js_property("loaded", True)
```

- Async

```
from playwright.async_api import expect

locator = page.locator(".component")
await expect(locator).to_have_js_property("loaded", True)
```

## # expect(locator).to\_have\_text(expected, \*\*kwargs) #

---

- `expected` <[str](#)|[Pattern](#)|[List](#)[[str](#)|[Pattern](#)]> 期望的子字符串或 [RegExp](#)或它们的列表. #
- `timeout` <[float](#)> 重试断言的时间. #
- `use_inner_text` <[bool](#)> 当检索DOM节点文本时使用 `element.innerText` 而不是 `element.textContent` . #
- returns: <[NoneType](#)> #

确保 [Locator](#) 指向具有给定文本的元素。您也可以为该值使用正则表达式。

- Sync

```
import re
from playwright.sync_api import expect

locator = page.locator(".title")
expect(locator).to_have_text(re.compile(r"Welcome, Test User"))
expect(locator).to_have_text(re.compile(r"Welcome, .*"))
```

- Async

```
import re
from playwright.async_api import expect

locator = page.locator(".title")
await expect(locator).to_have_text(re.compile(r"Welcome, Test User"))
await expect(locator).to_have_text(re.compile(r"Welcome, .*"))
```

注意，如果array是作为预期值传递的，那么整个元素列表都可以被断言：

- Sync

```
from playwright.sync_api import expect

locator = page.locator("list > .component")
expect(locator).to_have_text(["Text 1", "Text 2", "Text 3"])
```

- Async

```
from playwright.async_api import expect

locator = page.locator("list > .component")
await expect(locator).to_have_text(["Text 1", "Text 2", "Text 3"])
```

## # expect(locator).to\_have\_value(value, \*\*kwargs)#

---

- **value** <[str](#)|[Pattern](#)> 期望值 <#>
- **timeout** <[float](#)> 重试断言的时间. <#>
- returns: <[NoneType](#)> <#>

确保 [Locator](#) 指向具有给定输入值的元素。您也可以为该值使用正则表达式。

- Sync

```
import re
from playwright.sync_api import expect

locator = page.locator("input[type=number]")
expect(locator).to_have_value(re.compile(r"[0-9]"))
```

- Async

```
import re
from playwright.async_api import expect

locator = page.locator("input[type=number]")
await expect(locator).to_have_value(re.compile(r"[0-9]"))
```

#

## `expect(page).not_to_have_title(title_or_reg_exp, **kwargs)` <#>

---

- `title_or_reg_exp` [<str|Pattern>](#) 期望的标题或RegExp. <#>
- `timeout` [<float>](#) 重试断言的时间. <#>
- returns: [<NoneType>](#) <#>

与之相反 [expect\(page\).to\\_have\\_title\(title\\_or\\_reg\\_exp, \\*\\*kwargs\)](#).

#

## `expect(page).not_to_have_url(url_or_reg_exp, **kwargs)` <#>

---

- `url_or_reg_exp` [<str|Pattern>](#) 期望的子字符串或RegExp. <#>
- `timeout` [<float>](#) 重试断言的时间. <#>
- returns: [<NoneType>](#) <#>

与之相反 [expect\(page\).to\\_have\\_url\(url\\_or\\_reg\\_exp, \\*\\*kwargs\)](#).

#

## `expect(page).to_have_title(title_or_reg_exp, **kwargs)` <#>

---

- `title_or_reg_exp` <[str](#)|[Pattern](#)>期望的标题或RegExp.<#>
- `timeout` <[float](#)> 重试断言的时间.<#>
- returns: <[NoneType](#)><#>

确保 页面具有给定的标题。

- Sync

```
import re
from playwright.sync_api import expect

# ...
expect(page).to_have_title(re.compile(r".*checkout"))
```

- Async

```
import re
from playwright.async_api import expect

# ...
await expect(page).to_have_title(re.compile(r".*checkout"))
```

## **`# expect(page).to_have_url(url_or_reg_exp, **kwargs)`**<#>

---

- `url_or_reg_exp` <[str](#)|[Pattern](#)> 期望的子字符串或RegExp.<#>
- `timeout` <[float](#)> 重试断言的时间.<#>
- returns: <[NoneType](#)><#>

确保 页面被导航到给定的URL。

- Sync



```
import re
from playwright.sync_api import expect

# ...
expect(page).to_have_url(re.compile(".*checkout"))
```

- Async

```
import re
from playwright.async_api import expect

# ...
await expect(page).to_have_url(re.compile(".*checkout"))
```

## # `expect(api_response).not_to_be_ok()` #

---

- returns: <[NoneType](#)> #

与之相反 [expect\(api\\_response\).to\\_be\\_ok\(\)](#).

## # `expect(api_response).to_be_ok()` #

---

- returns: <[NoneType](#)> #

确保 确保响应状态码在[200..299]范围。

- Sync

```
import re
from playwright.sync_api import expect

# ...
expect(response).to_be_ok()
```

- Async

```
from playwright.async_api import expect

# ...

await expect(response).to_be_ok()
```

## Accessibility

可访问性类提供了检查Chromium的可访问性树的方法。辅助技术(如屏幕阅读器 [screen readers](#) 或开关 [switches](#))使用可访问性树。

可访问性是一个非常特定于平台的东西。在不同的平台上，不同的屏幕阅读器可能会产生非常不同的输出。

Chromium、Firefox和WebKit的渲染引擎都有一个“可访问性树”的概念，然后这些概念被翻译成不同的特定于平台的api。可访问性命名空间提供对此可访问性树的访问权。

当从内部浏览器AX树转换为特定于平台的AX- tree时，或者通过辅助技术本身，大多数可访问性树都会被过滤掉。默认情况下，剧作家会尝试近似过滤，只暴露树的“有趣”节点。

- [accessibility.snapshot\(\\*\\*kwargs\)](#)

## # accessibility.snapshot(\*\*kwargs)#

---

- `interesting_only` <bool> 从树中删除无兴趣节点。默认值为 `true`.#
- `root` <ElementHandle> 快照的根DOM元素。默认为整个page.#
- returns: <NoneType|Dict>#
  - `role` <str> 角色 [role](#).
  - `name` <str> 节点的人类可读名称.
  - `value` <str|float> 节点的当前值.
  - `description` <str> 一个额外的人类可读的节点描述，如果适用。
  - `keyshortcuts` <str> 与此节点相关联的键盘快捷键，如果适用的话。

- `roledescription` `<str>` 一个人类可读的角色替代者，如果适用的话。
- `valuetext` `<str>` 当前值的描述，如果适用的话。
- `disabled` `<bool>` 如果适用，节点是否被禁用。
- `expanded` `<bool>` 如果适用，节点是展开还是折叠。
- `focused` `<bool>` 如果适用，节点是否被聚焦。
- `modal` `<bool>` 如果适用，节点是否为模态。
- `multiline` `<bool>` 如果适用，节点文本输入是否支持多行。
- `multiselectable` `<bool>` 如果适用，是否可以选择多个子节点。
- `readonly` `<bool>` 是否为只读。
- `required` `<bool>` 是否需要该节点。
- `selected` `<bool>` 是否在其父节点中选中该节点。
- `checked` `<bool|"mixed">` 复选框是否选中，或是否为"mixed"(如果适用)
- `pressed` `<bool|"mixed">` 是否选中切换按钮，或如果适用，是"mixed"。
- `level` `<int>` 标题的级别，如果适用。
- `valuemin` `<float>` 节点中的最小值(如果适用)。
- `valuemax` `<float>` 节点中的最大值(如果适用)。
- `autocomplete` `<str>` 控件支持哪种类型的自动完成(如果适用)。
- `haspopup` `<str>` 当前节点显示的是哪种类型的弹出窗口，如果适用的话。
- `invalid` `<str>` 如果适用，该节点的值是否无效以及以何种方式无效。
- `orientation` `<str>` 如果适用，节点是水平方向还是垂直方向。
- `children` `<List[Dict]>` 子节点，如果有，如果适用。

捕获可访问性树的当前状态。返回的对象表示页面的根可访问节点。

#### NOTE

*Chromium*的可访问性树包含了在大多数平台和大多数屏幕阅读器上未使用的节点。*playwright*也会为了更容易处理树而丢弃它们，除非 `interesting_only` 被设置为 `false`。

转储整个可访问性树的示例:

- Sync

```
snapshot = page.accessibility.snapshot()  
print(snapshot)
```

- Async

```
snapshot = await page.accessibility.snapshot()  
print(snapshot)
```

记录被关注节点名称的示例:

- Sync

```
def find_focused_node(node):  
    if (node.get("focused")):  
        return node  
    for child in (node.get("children") or []):  
        found_node = find_focused_node(child)  
        return found_node  
    return None  
  
snapshot = page.accessibility.snapshot()  
node = find_focused_node(snapshot)  
if node:  
    print(node["name"])
```

- Async

```
def find_focused_node(node):
    if (node.get("focused"))
        return node
    for child in (node.get("children") or []):
        found_node = find_focused_node(child)
        return found_node
    return None

snapshot = await page.accessibility.snapshot()
node = find_focused_node(snapshot)
if node:
    print(node["name"])
```

## Browser

- extends: [EventEmitter](#)

Browser 是通过 [browser\\_type.launch\(\\*\\*kwargs\)](#) 创建的。使用 [Browser](#) 创建 [Page](#) 的例子:

- Sync

```
from playwright.sync_api import sync_playwright

def run(playwright):
    firefox = playwright.firefox
    browser = firefox.launch()
    page = browser.new_page()
    page.goto("https://example.com")
    browser.close()

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

- Async

```
import asyncio
from playwright.async_api import async_playwright
```

```
async def run(playwright):
    firefox = playwright.firefox
    browser = await firefox.launch()
    page = await browser.new_page()
    await page.goto("https://example.com")
    await browser.close()

async def main():
    async with async_playwright() as playwright:
        await run(playwright)

asyncio.run(main())
```

- [browser.on\("disconnected"\)](#)
- [browser.close\(\)](#)
- [browser.contexts](#)
- [browser.is\\_connected\(\)](#)
- [browser.new\\_browser\\_cdp\\_session\(\)](#)
- [browser.new\\_context\(\\*\\*kwargs\)](#)
- [browser.new\\_page\(\\*\\*kwargs\)](#)
- [browser.start\\_tracing\(\\*\\*kwargs\)](#)
- [browser.stop\\_tracing\(\)](#)
- [browser.version](#)

## # browser.on("disconnected") #

---

- type: <[Browser](#)>

当Browser与浏览器应用程序断开连接时触发。这可能是因为以下原因之一：

- Browser 程序关闭或崩溃。
- 调用了 [browser.close\(\)](#) 方法

## # browser.close() #

---

- returns: <[NoneType](#)> #

如果使用 [browser\\_type.launch\(\\*\\*kwargs\)](#) 获得此browser，则关闭browser及其所有page(如果有打开的page)。

如果这个浏览器连接到，清除所有创建的属于这个浏览器的上下文，并断开与浏览器服务器的连接。

[Browser](#) 对象本身被认为已被销毁，不能再使用。

## # browser.contexts#

---

- returns: <[List](#)[[BrowserContext](#)]>#

返回所有打开的浏览器上下文的数组。在新创建的浏览器中，这将不返回任何浏览器上下文

- Sync

```
browser = pw.webkit.launch()
print(len(browser.contexts())) # prints `0`
context = browser.new_context()
print(len(browser.contexts())) # prints `1`
```

- Async

```
browser = await pw.webkit.launch()
print(len(browser.contexts())) # prints `0`
context = await browser.new_context()
print(len(browser.contexts())) # prints `1`
```

## # browser.is\_connected()#

---

- returns: <[bool](#)>#

表示浏览器已连接。

## # browser.new\_browser\_cdp\_session()#

---

- returns: `<CDPSession>#`

#### NOTE

CDP会话仅支持基于chromium的浏览器。

返回新创建的浏览器会话。

## # browser.new\_context(\*\*kwargs)#

- `accept_downloads` `<bool>` 是否自动下载所有附件。在所有下载都被接受的地方默认为 True `#`
- `base_url` `<str>` 使用以下方法时 `page.goto(url, **kwargs)`, `page.route(url, handler, **kwargs)`, `page.wait_for_url(url, **kwargs)`, `page.expect_request(url_or_predicate, **kwargs)`, or `page.expect_response(url_or_predicate, **kwargs)` 通过使用 `URL()` 构造函数构建相应的URL:
  - `baseURL= http://localhost:3000` 时,导航到 `/bar.html` 结果是 `http://localhost:3000/bar.html`
  - `baseURL= http://localhost:3000/foo/` 时,导航到 `./bar.html` 结果是 `http://localhost:3000/foo/bar.html`
  - `baseURL= http://localhost:3000/foo` 时,导航到 (没有下划线) `./bar.html` 结果是 `http://localhost:3000/bar.html`
- `bypass_csp` `<bool>` 切换绕过页面的 Content-Security-Policy. `#`
- `color_scheme` `<"light"|"dark"|"no-preference">` 模拟 `'prefers-colors-scheme'` 的媒体特性, 支持的值为 `'light'`, `'dark'`, `'no-preference'`. 详情请参阅 `page.emulate_media(**kwargs)`. 默认值为 `'light'`. `#`
- `device_scale_factor` `<float>` 指定设备比例因子(可以认为是 dpr)。默认为 `1`. `#`
- `extra_http_headers` `<Dict[str, str]>` 一个包含附加HTTP头的对象, 每个请求都要发送. `#`
- `forced_colors` `<"active"|"none">` 模拟 `'forced-colors'` 媒体特性, 支持的值为 `'active'`, `'none'`. 详情请参阅 `page.emulate_media(**kwargs)`. 默认为 `'none'`. `#`



它在WebKit中不支持，请在他们的问题跟踪器中查看 [here](#)。

- `geolocation` `<Dict>#`
  - `latitude` `<float>` 纬度介于-90和90之间。
  - `longitude` `<float>` 经度介于-180和180之间。
  - `accuracy` `<float>` 非负精度值。默认值为 `0`。
- `has_touch` `<bool>` 指定视口是否支持触摸事件。默认值为 `false` `.#`
- `http_credentials` `<Dict>` HTTP认证凭据 [HTTP authentication](#) `.#`
  - `username` `<str>`
  - `password` `<str>`
- `ignore_https_errors` `<bool>` 发送网络请求时是否忽略HTTPS错误。默认值为 `false` `.#`
- `is_mobile` `<bool>` 是否考虑 `meta viewport` 标签，是否启用触摸事件。默认值为 `false`。Firefox中不支持 `.#`
- `java_script_enabled` `<bool>` 是否在上下文中启用JavaScript。默认值为 `true` `.#`
- `locale` `<str>` 指定用户的本地语言环境，例如 `en-GB`，`de-DE`，等。区域设置将影响导航器。 `navigator.language` 的值， `Accept-Language` 请求头值以及数字和日期格式规则 `.#`
- `no_viewport` `<bool>` 不强制固定viewport，允许在头部模式下调整窗口大小 `.#`
- `offline` `<bool>` 是否仿真网络离线。默认值为 `false` `.#`
- `permissions` `<List[str]>` 在此上下文中授予所有页面的权限列表。获取更多细节: [browser\\_context.grant\\_permissions\(permissions, \\*\\*kwargs\)](#) `.#`
- `proxy` `<Dict>` 在此上下文中使用的网络代理设置 `.#`
  - `server` `<str>` 所有请求使用的代理。支持HTTP代理和SOCKS代理，例如: `http://myproxy.com:3128` or `socks5://myproxy.com:3128`。缩写形式 `myproxy.com:3128` 被认为是一个HTTP代理。
  - `bypass` `<str>` 可选，用逗号分隔的域，绕过代理，例如 `".com, chromium.org, .domain.com"`。
  - `username` `<str>` 可选username，当HTTP代理需要鉴权时使用。
  - `password` `<str>` 当HTTP代理需要鉴权时可选密码。

## NOTE

对于Windows上的Chromium浏览器，需要使用全局代理启动该选项才能工作。如果所有上下文覆盖了代理，全局代理将永远不会被使用，可以是任何字符串，例如：`launch({ proxy: { server: 'http://per-context' } })`。

- `record_har_omit_content` `<bool>` 可选设置，控制是否从HAR忽略请求内容。默认值为 `false`。#
- `record_har_path` `<Union[str, pathlib.Path]>` 为文件系统中指定的 `HAR` 文件中所有页面启用HAR记录，如果没有指定，则不会记录HAR。确保调用 `browser_context.close()` 来保存HAR。#
- `record_video_dir` `<Union[str, pathlib.Path]>` 开启进入指定目录的所有页面的视频录制，如果没有指定，则不录制视频。确保调用 `browser_context.close()` 来保存视频。#
- `record_video_size` `<Dict>` 录制视频的尺寸。如果没有指定大小将等于 `viewport` 缩小到800x800。如果 `viewport` 没有显式配置，视频大小默认为800x450。每个页面的实际图片将按比例缩小，如果需要，以适应指定的大小。#
  - `width` `<int>` 视频帧宽度。
  - `height` `<int>` 视频帧高度。
- `reduced_motion` `<"reduce"|"no-preference">` 模拟 `'prefers-reduced-motion'` 媒体特性，支持的值为 `'reduce'`，`'no-preference'`。详情请参阅 `page.emulate_media(**kwargs)`。默认值为 `'no-preference'`。#
- `screen` `<Dict>` 通过 `window.screen` 在web页面中模拟一致的窗口屏幕大小。仅在 `viewport` 设置时使用。#
  - `width` `<int>` 页面宽度(px)。
  - `height` `<int>` 页面高度(px)。
- `storage_state` `<Union[str, pathlib.Path]|Dict>` 用给定的存储状态填充上下文。这个选项可以用来通过 `browser_context.storage_state(**kwargs)` 获得的登录信息初始化上下文。可以是保存了存储空间的文件路径，也可以是包含以下字段的对象：#
  - `cookies` `<List[Dict]>` cookie设置上下文
    - `name` `<str>`
    - `value` `<str>`
    - `domain` `<str>` Domain和path是必需的
    - `path` `<str>` Domain和path是必需的
    - `expires` `<float>` Unix 时间，单位为秒。
    - `httpOnly` `<bool>`

- `secure` <[bool](#)>
- `sameSite` <"Strict"|"Lax"|"None"> 相同站点标识
- `origins` <[List\[Dict\]](#)> localStorage 设置上下文
  - `origin` <[str](#)>
  - `localStorage` <[List\[Dict\]](#)>
    - `name` <[str](#)>
    - `value` <[str](#)>
- `strict_selectors` <[bool](#)> 它指定了，为这个上下文启用严格选择器模式。在严格的选择器模式中，当有多个元素匹配选择器时，所有对选择器的操作都将意味着只有一个目标DOM元素。请参阅 [Locator](#) 以了解更多关于严格模式。<#>
- `timezone_id` <[str](#)> 修改上下文的时区。查看 [ICU's metaZones.txt](#) 获取支持的时区id列表。<#>
- `user_agent` <[str](#)> 在此上下文中使用的特定用户代理。<#>
- `viewport` <[NoneType|Dict](#)> 为每个页面设置一个一致的视口。默认为1280x720视口。 `no_viewport` 禁用固定视口。<#>
  - `width` <[int](#)> 页面宽度(px).
  - `height` <[int](#)> 页面高度(px).
- returns: <[BrowserContext](#)> <#>

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cookie /缓存

- Sync

```
browser = playwright.firefox.launch() # or "chromium" or "webkit".
# create a new incognito browser context.
context = browser.new_context()
# create a new page in a pristine context.
page = context.new_page()
page.goto("https://example.com")
```

- Async

```

browser = await playwright.firefox.launch() # or "chromium" or
"webkit".
# create a new incognito browser context.
context = await browser.new_context()
# create a new page in a pristine context.
page = await context.new_page()
await page.goto("https://example.com")

```

## # browser.new\_page(\*\*kwargs)#

- `accept_downloads` <bool> 是否自动下载所有附件。在所有下载都被接受的地方默认为 True .#
- `base_url` <str> 使用以下方法时 `page.goto(url, **kwargs)`, `page.route(url, handler, **kwargs)`, `page.wait_for_url(url, **kwargs)`, `page.expect_request(url_or_predicate, **kwargs)`, or `page.expect_response(url_or_predicate, **kwargs)` 通过使用 `URL()` 构造函数构建相应的URL: #
  - `baseURL= http://localhost:3000` 时,导航到 `/bar.html` 结果是 `http://localhost:3000/bar.html`
  - `baseURL= http://localhost:3000/foo/` 时,导航到 `./bar.html` 结果是 `http://localhost:3000/foo/bar.html`
  - `baseURL= http://localhost:3000/foo` 时,导航到 (没有下划线) `./bar.html` 结果是 `http://localhost:3000/bar.html`
- `bypass_csp` <bool> 切换绕过页面的 Content-Security-Policy. #
- `color_scheme` <"light"|"dark"|"no-preference"> 模拟 `'prefers-colors-scheme'` 的媒体特性, 支持的值为 `'light'`, `'dark'`, `'no-preference'`. 详情请参阅 `page.emulate_media(**kwargs)`. 默认值为 `'light'`. #
- `device_scale_factor` <float> 指定设备比例因子(可以认为是 dpr)。默认为 1. #
- `extra_http_headers` <Dict[str, str]> 个包含附加HTTP头的对象, 每个请求都要发送. #
- `forced_colors` <"active"|"none">模拟 `'forced-colors'` 媒体特性, 支持的值为 `'active'`, `'none'`. 详情请参阅 `page.emulate_media(**kwargs)`. 默认为 `'none'`. #

## NOTE

它在WebKit中不支持，请在他们的问题跟踪器中查看 [here](#) .

- `geolocation` <Dict>#
  - `latitude` <float> 纬度介于-90和90之间。
  - `longitude` <float> 经度介于-180和180之间。
  - `accuracy` <float> 非负精度值。默认值为 `0` .
- `has_touch` <bool> 指定视口是否支持触摸事件。默认值为 `false` .#
- `http_credentials` <Dict> HTTP认证凭据 [HTTP authentication](#) .#
  - `username` <str>
  - `password` <str>
- `ignore_https_errors` <bool> 发送网络请求时是否忽略HTTPS错误。默认值为 `false` .#
- `is_mobile` <bool> 是否考虑 `meta viewport` 标签，是否启用触摸事件。默认值为 `false` . Firefox中不支持.#
- `java_script_enabled` <bool> 是否在上下文中启用JavaScript。默认值为 `true` .#
- `locale` <str> 指定用户的本地语言环境，例如 `en-GB` , `de-DE` , 等。区域设置将影响导航器. `navigator.language` 的值, `Accept-Language` 请求头值以及数字和日期格式规则.#
- `no_viewport` <bool> 不强制固定viewport，允许在头部模式下调整窗口大小.#
- `offline` <bool> 是否仿真网络离线。默认值为 `false` .#
- `permissions` <List[str]> 在此上下文中授予所有页面的权限列表。获取更多细节: [browser\\_context.grant\\_permissions\(permissions, \\*\\*kwargs\)](#) .#
- `proxy` <Dict> 在此上下文中使用的网络代理设置.#
  - `server` <str> 所有请求使用的代理。支持HTTP代理和SOCKS代理，例如: `http://myproxy.com:3128` or `socks5://myproxy.com:3128` .缩写形式 `myproxy.com:3128` 被认为是一个HTTP代理。
  - `bypass` <str> 可选，用逗号分隔的域，绕过代理，例如 `".com, chromium.org, .domain.com"` .
  - `username` <str> 可选username，当HTTP代理需要鉴权时使用。
  - `password` <str> 当HTTP代理需要鉴权时可选密码。

## NOTE

对于Windows上的Chromium浏览器，需要使用全局代理启动该选项才能工作。如果所有上下文覆盖了代理，全局代理将永远不会被使用，可以是任何字符串，例如：`launch({ proxy: { server: 'http://per-context' } })`。

- `record_har_omit_content` `<bool>` 可选设置，控制是否从HAR忽略请求内容。默认值为 `false`。#
- `record_har_path` `<Union[str, pathlib.Path]>` 为文件系统中指定的 `HAR` 文件中所有页面启用HAR 记录,如果没有指定，则不会记录HAR。确保调用 `browser_context.close()` 来保存HAR。#
- `record_video_dir` `<Union[str, pathlib.Path]>` 开启进入指定目录的所有页面的视频录制,如果没有指定，则不录制视频。确保调用 `browser_context.close()` 来保存视频。#
- `record_video_size` `<Dict>` 录制视频的尺寸。如果没有指定大小将等于 `viewport` 缩小到800x800。如果 `viewport` 没有显式配置，视频大小默认为800x450。每个页面的实际图片将按比例缩小，如果需要，以适应指定的大小。#
  - `width` `<int>` 视频帧宽度。
  - `height` `<int>` 视频帧高度。
- `reduced_motion` `<"reduce"|"no-preference">` 模拟 `'prefers-reduced-motion'` 媒体特性，支持的值为 `'reduce'`，`'no-preference'`。详情请参阅 `page.emulate_media(**kwargs)`。默认值为 `'no-preference'`。#
- `screen` `<Dict>` 通过 `window.screen` 在web页面中模拟一致的窗口屏幕大小。仅在 `viewport` 设置时使用。#
  - `width` `<int>` 页面宽度(px)。
  - `height` `<int>` 页面高度(px)。
- `storage_state` `<Union[str, pathlib.Path]|Dict>` 用给定的存储状态填充上下文。这个选项可以用来通过 `browser_context.storage_state(**kwargs)` 获得的登录信息初始化上下文。可以是保存了存储空间的文件路径，也可以是包含以下字段的对象：#
  - `cookies` `<List[Dict]>` cookie设置上下文
    - `name` `<str>`
    - `value` `<str>`
    - `domain` `<str>` Domain和path是必需的
    - `path` `<str>` Domain和path是必需的
    - `expires` `<float>` Unix 时间，单位为秒
    - `httpOnly` `<bool>`
    - `secure` `<bool>`

- `sameSite` `<"Strict"|"Lax"|"None">` 相同站点标识
- `origins` `<List[Dict]>` localStorage 设置上下文
  - `origin` `<str>`
  - `localStorage` `<List[Dict]>`
    - `name` `<str>`
    - `value` `<str>`
- `strict_selectors` `<bool>` 它指定了，为这个上下文启用严格选择器模式。在严格的选择器模式中，当有多个元素匹配选择器时，所有对选择器的操作都将意味着只有一个目标DOM元素。请参阅 [Locator](#) 以了解更多关于严格模式。<#>
- `timezone_id` `<str>` 修改上下文的时区。查看 [ICU's metaZones.txt](#) 获取支持的时区id列表。<#>
- `user_agent` `<str>` 在此上下文中使用的特定用户代理。<#>
- `viewport` `<NoneType|Dict>` 为每个页面设置一个一致的视口。默认为1280x720视口。 `no_viewport` 禁用固定视口。
  - `width` `<int>` 页面宽度(px).
  - `height` `<int>` 页面高度(px).
- returns: `<Page>` <#>

在新的浏览器上下文中创建一个新页面。关闭此页面也将关闭上下文。

这是一个方便的API，应该只用于单页场景和简短的片段。产品代码和测试框架应该显式地创建 [browser.new\\_context\(\\*\\*kwargs\)](#) ,然后再创建 [browser\\_context.new\\_page\(\)](#) 来控制它们的确切生命周期。

## # browser.start\_tracing(\*\*kwargs) <#>

- `page` `<Page>` 可选，如果指定，跟踪包含给定页面的截图。<#>
- `categories` `<List[str]>` 指定要使用的自定义类别而不是默认类别。<#>
- `path` `<Union[str, pathlib.Path]>` 文件写入的路径。<#>
- `screenshots` `<bool>` 捕获跟踪中的屏幕截图。<#>
- returns: `<NoneType>` <#>

### NOTE

这个API控制 [Chromium Tracing](#) 这是一个低级别的Chromium专用调试工具。控制 [Playwright Tracing](#) 的API可以在 [这里](#) 找到。



你可以使用 [browser.start\\_tracing\(\\*\\*kwargs\)](#) 和 [browser.stop\\_tracing\(\)](#) 来创建一个可以在 *Chrome DevTools* 性能面板中打开的跟踪文件。

- Sync

```
browser.start_tracing(page, path="trace.json")
page.goto("https://www.google.com")
browser.stop_tracing()
```

- Async

```
await browser.start_tracing(page, path="trace.json")
await page.goto("https://www.google.com")
await browser.stop_tracing()
```

## # browser.stop\_tracing() #

---

- returns: <[bytes](#)> #

### NOTE

这个API控制 [Chromium Tracing](#) 这是一个低级别的 *Chromium* 专用调试工具。控制 [Playwright Tracing](#) 的API可以在 [这里](#) 找到。

返回带有跟踪数据的缓冲区。

## # browser.version #

---

- returns: <[str](#)> #

返回浏览器版本



## BrowserContext

- extends: [EventEmitter](#)

BrowserContexts 提供了一种操作多个独立浏览器会话的方法。

如果一个页面打开另一个页面，例如with a `window.open` call，弹出窗口将属于父页面的浏览器上下文。

Playwright 允许使用 `browser.new_context(**kwargs)`方法创建“Incognito”浏览器上下文。“Incognito”浏览器上下文不会将任何浏览数据写入磁盘。

- Sync

```
# create a new incognito browser context
context = browser.new_context()
# create a new page inside context.
page = context.new_page()
page.goto("https://example.com")
# dispose context once it is no longer needed.
context.close()
```

- Async

```
# create a new incognito browser context
context = await browser.new_context()
# create a new page inside context.
page = await context.new_page()
await page.goto("https://example.com")
# dispose context once it is no longer needed.
await context.close()
```

## # browser\_context.on("backgroundpage")#

- type: [<Page>](#)

### NOTE

仅适用于Chromium 浏览器的持久上下文。

当在上下文中创建新的后台页面时触发。

- Sync

```
background_page = context.wait_for_event("backgroundpage")
```

- Async

```
background_page = await  
context.wait_for_event("backgroundpage")
```

## # browser\_context.on("close")#

---

- type: <[BrowserContext](#)>

当浏览器上下文关闭时触发。这可能是因为以下原因之一：

- 关闭浏览器上下文.
- 浏览器应用程序关闭或崩溃.
- 调用了 [browser.close\(\)](#) 方法.

## # browser\_context.on("page")#

---

- type: <[Page](#)>

当在BrowserContext中创建一个新的Page时，会触发该事件。页面可能仍在加载中。该事件也将为弹出页面而触发。请参见[page.on\("popup"\)](#) 接收与特定页面相关的关于弹出窗口的事件。

该页面可用的最早时刻是当它已导航到初始url。例如，当用 `window.open('http://example.com')`，打开一个弹出窗口时，这个事件将在网络请求"<http://example.com>" 完成并在弹出窗口中开始加载响应时触发。

- Sync

```
with context.expect_page() as page_info:
    page.click("a[target=_blank]"),
    page = page_info.value
print(page.evaluate("location.href"))
```

- Async

```
async with context.expect_page() as page_info:
    await page.click("a[target=_blank]"),
    page = await page_info.value
print(await page.evaluate("location.href"))
```

#### NOTE

使用 [`page.wait\_for\_load\_state\(\*\*kwargs\)`](#) 等待页面到达特定的状态(在大多数情况下不需要它).

## # browser\_context.on("request")#

- type: <[Request](#)>

当从通过此上下文创建的任何页面发出请求时触发。请求对象是只读的。要只监听来自特定页面的请求，请使用 [`page.on\("request"\)`](#).

为了拦截和修改请求，请参阅 [`browser\_context.route\(url, handler, \*\*kwargs\)`](#) or [`page.route\(url, handler, \*\*kwargs\)`](#).

## # browser\_context.on("requestfailed")#

- type: <[Request](#)>

当请求失败时触发，例如超时。要只监听来自特定页面的失败请求，请使用 [`page.on\("requestfailed"\)`](#).

#### NOTE

*HTTP*错误响应，如404或503，从*HTTP*的角度来看，仍然是成功的响应，因此请求将完成[browser\\_context.on\("requestfinished"\)](#)事件，而不是[browser\\_context.on\("requestfailed"\)](#)。

## # browser\_context.on("requestfinished")#

---

- type: <[Request](#)>

在下载响应体后，请求成功完成时触发。对于一个成功的响应，事件序列是 `request`，`response` and `requestfinished`。要监听来自特定页面的成功请求，请使用 [page.on\("requestfinished"\)](#)。

## # browser\_context.on("response")#

---

- type: <[Response](#)>

当收到请求的响应状态和报头时触发。对于一个成功的响应，事件序列是 `request`，`response` and `requestfinished`。要监听来自特定页面的响应事件，请使用 [page.on\("response"\)](#)。

## # browser\_context.on("serviceworker")#

---

- type: <[Worker](#)>

*NOTE*

*Service worker*只支持基于*chrome*的浏览器。

当在上下文中创建新的 *service worker* 时触发。

## # browser\_context.add\_cookies(cookies)#

---

- `cookies` <[List](#)[[Dict](#)]>#
  - `name` <[str](#)>
  - `value` <[str](#)>

- `url` `<str>` 完整的URL(带协议), `url` 和(`domain / path`) 之间必选其一.
- `domain` `<str>` 域(与`path`一起使用),不需要加协议. `url` 和(`domain / path`) 之间必选其一.
- `path` `<str>` 路径(与`domain` 一起使用),前面需要加 `/`. `url` 和(`domain / path`) 之间必选其一.
- `expires` `<float>` Unix 时间, 单位为秒. 可选的.
- `httpOnly` `<bool>` 可选的.
- `secure` `<bool>` 可选的.
- `sameSite` `<"Strict"|"Lax"|"None">` `<“严格”|“宽松”|“没有”>`可选的.
- returns: `<NoneType>#`

将cookie添加到此浏览器上下文中。这个上下文中的所有页面都将安装这些 cookie。cookie可以通过 `browser_context.cookies(**kwargs)` 获取。

```
# cookie_object like
{
    'name': 'aaa',
    'value': '132456',
    'domain': 'httpbin.org',
    'path': '/cookies'
}
# or
{
    'name': 'aaa',
    'value': '132456',
    'url': 'http://httpbin.org/cookies',
}
```

- Sync

```
browser_context.add_cookies([cookie_object1, cookie_object2])
```

- Async

```
await browser_context.add_cookies([cookie_object1,
cookie_object2])
```

#

## browser\_context.add\_init\_script(\*\*kwargs)

#

---

- `path` <[Union](#)[[str](#), [pathlib.Path](#)]> JavaScript 文件的路径. 如果 `path` 是一个相对路径, 那么它是相对于当前工作目录解析的. 可选的. <#>
- `script` <[str](#)> 在浏览器上下文中所有页面中执行的脚本. 可选的. <#>
- `returns`: <[NoneType](#)> <#>

添加一个脚本, 该脚本将在以下场景之一中进行评估:

- 当在浏览器上下文中创建或导航页面时.
- 当在浏览器上下文中的任何页面中附加或导航一个子框架时. 在这种情况下, 脚本将在新附加的框架的上下文中执行.

在创建文档之后, 但在运行文档的任何脚本之前, 对脚本进行计算. 这对于修改 JavaScript 环境是很有用的, 例如 `Math.random`.

一个重写 `Math.random` 的例子. 页面加载前:

```
// preload.js
Math.random = () => 42;
```

- Sync

```
# in your playwright script, assuming the preload.js file is
in same directory.
browser_context.add_init_script(path="preload.js")
```

- Async

```
# in your playwright script, assuming the preload.js file is
in same directory.
await browser_context.add_init_script(path="preload.js")
```

NOTE

通过 [browser\\_context.add\\_init\\_script\(\\*\\*kwargs\)](#) and [page.add\\_init\\_script\(\\*\\*kwargs\)](#) 安装的多个脚本的计算顺序没有定义.

## # browser\_context.background\_pages#

---

- returns: <[List](#)[[Page](#)]>#

### NOTE

背景页面仅支持基于*chrome*的浏览器.

所有现有的背景页在上下文中.

## # browser\_context.browser#

---

- returns: <[NoneType](#)|[Browser](#)>#

返回上下文的浏览器实例。如果它作为持久上下文启动，则返回null.

## # browser\_context.clear\_cookies()#

---

- returns: <[NoneType](#)>#

清除上下文 cookies.

## # browser\_context.clear\_permissions()#

---

- returns: <[NoneType](#)>#

清除浏览器上下文的所有权限覆盖.

- Sync

```
context = browser.new_context()
context.grant_permissions(["clipboard-read"])
# do stuff ..
context.clear_permissions()
```

- Async

```
context = await browser.new_context()
await context.grant_permissions(["clipboard-read"])
# do stuff ..
context.clear_permissions()
```

## # browser\_context.close()#

---

- returns: <[NoneType](#)>#

关闭浏览器上下文。属于浏览器上下文的所有页面都将被关闭。

### NOTE

无法关闭默认的浏览器上下文。

## # browser\_context.cookies(\*\*kwargs)#

---

- `urls` <[str](#)|[List](#)[[str](#)]> 可选url列表.#
- returns: <[List](#)[[Dict](#)]>#
  - `name` <[str](#)>
  - `value` <[str](#)>
  - `domain` <[str](#)>
  - `path` <[str](#)>
  - `expires` <[float](#)> Unix 时间，单位为秒.
  - `httpOnly` <[bool](#)>
  - `secure` <[bool](#)>
  - `sameSite` <"Strict"|"Lax"|"None"> <“严格”|“宽松”|“没有”>

如果没有指定url，则此方法返回所有cookie。如果指定了url，则只返回影响这些url的cookie。



## # browser\_context.expect\_event(event, \*\*kwargs)#

---

- `event` <str> 事件名称，相同的一个将传递给 `browserContext.on(event)` .#
- `predicate` <Callable> 接收事件数据，并在等待应该被解析时解析为 True .#
- `timeout` <float> 最大等待时间，单位为毫秒。默认为 `30000` (30 seconds). 传入 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` 来更改.#
- returns: <EventManager> .#

等待事件触发，并将其值传递给谓词函数。当谓词返回 `True` 时返回。如果上下文在触发事件之前关闭，则将抛出一个错误。返回事件数据值。

- Sync

```
with context.expect_event("page") as event_info:
    page.click("button")
page = event_info.value
```

- Async

```
async with context.expect_event("page") as event_info:
    await page.click("button")
page = await event_info.value
```

## # browser\_context.expect\_page(\*\*kwargs)#

---

- `predicate` <Callable[Page]:bool> 接收 `Page` 对象，并在等待应该解决时解析为 True .#
- `timeout` <float> 最大等待时间，单位为毫秒。默认为 `30000` (30 seconds). 传入 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` 来更改.#
- returns: <EventManager[Page]> .#

执行操作并等待在上下文中创建一个新的 `Page` 如果提供了 `predicate`，它将 `Page` 值传递给 `predicate` 函数，并等待 `predicate(event)` 返回一个真值。如果上下文在创建新页面之前关闭，将抛出一个错误。

## # browser\_context.expose\_binding(name, callback, \*\*kwargs)#

---

- **name** <str> window对象上的函数名.#
- **callback** <Callable> 在 playwright 的上下文中被调用的回调函数.#
- **handle** <bool> 是否将参数作为句柄传递，而不是按值传递。当传递句柄时，只支持一个参数。当传递值时，支持多个参数.#
- **returns:** <NoneType>#

该方法在上下文中每一页的每一帧的窗口对象上添加一个名为**name**的函数。当被调用时，函数执行回调并返回一个 [Promise](#) ,该Promise解析为回调的返回值。如果回调返回一个Promise，它将被等待

**callback** 函数的第一个参数包含调用者的信息: `{ browserContext: BrowserContext, page: Page, frame: Frame }`.

查看仅用于页面版本的 [page.expose\\_binding\(name, callback, \\*\\*kwargs\)](#).

一个将页面URL暴露给上下文中所有页面中的所有帧的例子:

- Sync

```
from playwright.sync_api import sync_playwright

def run(playwright):
    webkit = playwright.webkit
    browser = webkit.launch(headless=False)
    context = browser.new_context()
    context.expose_binding("pageURL", lambda source:
source["page"].url)
    page = context.new_page()
    page.set_content("""
<script>
    async function onClick() {
        document.querySelector('div').textContent = await
window.pageURL();
    }
</script>
<button onclick="onClick()">Click me</button>
<div></div>
""")
    page.click("button")
```

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

- Async

```
import asyncio
from playwright.async_api import async_playwright

async def run(playwright):
    webkit = playwright.webkit
    browser = await webkit.launch(headless=False)
    context = await browser.new_context()
    await context.expose_binding("pageURL", lambda source:
source["page"].url)
    page = await context.new_page()
    await page.set_content("""
<script>
    async function onClick() {
        document.querySelector('div').textContent = await
window.pageURL();
    }
</script>
<button onclick="onClick()">Click me</button>
<div></div>
""")
    await page.click("button")

async def main():
    async with async_playwright() as playwright:
        await run(playwright)
    asyncio.run(main())
```

传递元素句柄的例子:

- Sync

```
def print(source, element):
    print(element.text_content())

context.expose_binding("clicked", print, handle=true)
page.set_content("""
    \<script>
        document.addEventListener('click', event =>
window.clicked(event.target));
    \</script>
    \<div>Click me\</div>
    \<div>Or click me\</div>
    """)
```

- Async

```
async def print(source, element):
    print(await element.text_content())

await context.expose_binding("clicked", print, handle=true)
await page.set_content("""
    \<script>
        document.addEventListener('click', event =>
window.clicked(event.target));
    \</script>
    \<div>Click me\</div>
    \<div>Or click me\</div>
    """)
```

## # browser\_context.expose\_function(name, callback) #

---

- **name** <str> window对象上的函数名. #
- **callback** <Callable> 在 playwright 的上下文中被调用的回调函数. #
- returns: <NoneType> #

该方法在上下文中每一页的每一帧的窗口对象上添加一个名为name的函数。当被调用时，函数执行回调并返回一个 [Promise](#) ,该Promise解析为回调的返回值。

如果回调返回一个 [Promise](#) ,它将被等待。

查看仅页面版本的 [page.expose\\_function\(name, callback\)](#) .

在上下文中为所有页面添加一个 `sha256` 函数的例子:

- Sync

```
import hashlib
from playwright.sync_api import sync_playwright

def sha256(text):
    m = hashlib.sha256()
    m.update(bytes(text, "utf8"))
    return m.hexdigest()

def run(playwright):
    webkit = playwright.webkit
    browser = webkit.launch(headless=False)
    context = browser.new_context()
    context.expose_function("sha256", sha256)
    page = context.new_page()
    page.set_content("""
        \<script>
            async function onClick() {
                document.querySelector('div').textContent = await
window.sha256('PLAYWRIGHT');
            }
        \</script>
        \<button onclick="onClick()">Click me\</button>
        \<div>\</div>
    """)
    page.click("button")

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

- Async

```
import asyncio
import hashlib
from playwright.async_api import async_playwright
```

```

def sha256(text):
    m = hashlib.sha256()
    m.update(bytes(text, "utf8"))
    return m.hexdigest()

async def run(playwright):
    webkit = playwright.webkit
    browser = await webkit.launch(headless=False)
    context = await browser.new_context()
    await context.expose_function("sha256", sha256)
    page = await context.new_page()
    await page.set_content("""
        \<script>
            async function onClick() {
                document.querySelector('div').textContent = await
window.sha256('PLAYWRIGHT');
            }
        \</script>
        \<button onclick="onClick()">Click me\</button>
        \<div>\</div>
    """)
    await page.click("button")

async def main():
    async with async_playwright() as playwright:
        await run(playwright)
    asyncio.run(main())

```

#

## browser\_context.grant\_permissions(permissions, \*\*kwargs)#

---

- `permissions` `<List[str]>` 要授予的权限或权限数组。权限可以是以下值之一: `#`
  - `'geolocation'` 地理位置
  - `'midi'`
  - `'midi-sysex'` (system-exclusive midi)
  - `'notifications'` 通知

- `'camera'` 相机
- `'microphone'` 麦克风
- `'background-sync'`
- `'ambient-light-sensor'` 环境光传感器
- `'accelerometer'` 加速度计
- `'gyroscope'` 陀螺
- `'magnetometer'` 磁强计
- `'accessibility-events'`
- `'clipboard-read'`
- `'clipboard-write'`
- `'payment-handler'`
- `origin` `<str>` 要授予权限的起点，例如: "<https://example.com>".#
- returns: `<NoneType>` #

向浏览器上下文授予指定的权限。如果指定，则仅向给定的源授予相应的权限。

## # `browser_context.new_cdp_session(page)` #

- `page` `<Page|Frame>` 创建会话的目标。为了向后兼容，这个参数被命名为 `page`，但它可以是 `Page` or `Frame` 类型.#
- returns: `<CDPSession>` #

### NOTE

CDP会话仅支持基于`chromium`的浏览器

返回新创建的会话。

## # `browser_context.new_page()` #

- returns: `<Page>` #

在浏览器上下文中创建一个新页面。

## # `browser_context.pages` #

- returns: `<List[Page]>` #

返回上下文中所有打开的页面。

## # browser\_context.route(url, handler, \*\*kwargs) #

---

- `url` <[str](#)|[Pattern](#)|[Callable](#)[[URL](#)]:[bool](#)> 一个glob模式、regex模式或谓词在路由时接收要匹配的 [URL](#) ,当通过上下文选项提供了一个 `base_url` 并且传递的URL是一个路径时, 它会通过 [new URL\(\)](#) 构造函数合并. #
- `handler` <[Callable](#)[[Route](#), [Request](#)]> handler函数路由请求. #
- `times` <[int](#)> 一个路由应该使用的频率。默认情况下, 每次都会使用. #
- `returns`: <[NoneType](#)> #

路由提供了修改浏览器上下文中任何页面发出的网络请求的能力。一旦路由被启用, 每一个匹配url模式的请求都会停止, 除非它被继续、完成或中止。

### NOTE

[page.route\(url, handler, \\*\\*kwargs\)](#) 不会拦截被 *Service Worker* 拦截的请求. 详情查看 [这个问题](#) . 我们建议在使用请求拦截时禁用 *Service Workers*, 通过 `await context.addInitScript(() => delete window.navigator.serviceWorker);`

一个简单的处理程序的例子, 中止所有的图像请求:

- Sync

```
context = browser.new_context()
page = context.new_page()
context.route("**/*.{png,jpg,jpeg}", lambda route:
route.abort())
page.goto("https://example.com")
browser.close()
```

- Async



```
context = await browser.new_context()
page = await context.new_page()
await context.route("**/*.{png,jpg,jpeg}", lambda route:
route.abort())
await page.goto("https://example.com")
await browser.close()
```

或者使用regex模式替换相同的代码片段:

- Sync

```
context = browser.new_context()
page = context.new_page()
context.route(re.compile(r"(\.png$)|(\.jpg$)"), lambda route:
route.abort())
page = await context.new_page()
page = context.new_page()
page.goto("https://example.com")
browser.close()
```

- Async

```
context = await browser.new_context()
page = await context.new_page()
await context.route(re.compile(r"(\.png$)|(\.jpg$)"), lambda
route: route.abort())
page = await context.new_page()
await page.goto("https://example.com")
await browser.close()
```

可以通过检查请求来决定路由操作。例如，mock所有包含post数据的请求，并保留所有其他请求的原样:

- Sync

```
def handle_route(route):
    if ("my-string" in route.request.post_data)
        route.fulfill(body="mocked-data")
    else
        route.continue_()
context.route("/api/**", handle_route)
```

- Async

```
def handle_route(route):  
    if ("my-string" in route.request.post_data)  
        route.fulfill(body="mocked-data")  
    else  
        route.continue_()  
    await context.route("/api/**", handle_route)
```

当请求同时匹配两个处理程序时, [page.route\(url, handler, \\*\\*kwargs\)](#) 优先于浏览器上下文路由

要移除带有处理程序的路由, 你可以使用 [browser\\_context.unroute\(url, \\*\\*kwargs\)](#).

NOTE

启用路由将禁用 *http* 缓存.

## # browser\_context.service\_workers#

---

- returns: <[List](#) [[Worker](#)] >#

NOTE

*Service worker* 只支持基于 *chrome* 的浏览器。

上下文中的所有现有服务工作者。

## #

## browser\_context.set\_default\_navigation\_timeout(timeout)#

---

- `timeout` <[float](#)> 最大导航时间, 以毫秒为单位#
- returns: <[NoneType](#)>#

此设置将改变以下方法和相关快捷方式的默认最大导航时间:

- [`page.go\_back\(\*\*kwargs\)`](#)
- [`page.go\_forward\(\*\*kwargs\)`](#)
- [`page.goto\(url, \*\*kwargs\)`](#)
- [`page.reload\(\*\*kwargs\)`](#)
- [`page.set\_content\(html, \*\*kwargs\)`](#)
- [`page.expect\_navigation\(\*\*kwargs\)`](#)

#### NOTE

[`page.set\_default\_navigation\_timeout\(timeout\)`](#) 和 [`page.set\_default\_timeout\(timeout\)`](#) 优先于 [`browser\_context.set\_default\_navigation\_timeout\(timeout\)`](#).

## #

## `browser_context.set_default_timeout(timeout)` #

---

- `timeout` <[`float`](#)> 最大时间(毫秒) #
- returns: <[`NoneType`](#)> #

此设置将更改 **所有** 接受超时的方法的默认最大时间。

#### NOTE

[`page.set\_default\_navigation\_timeout\(timeout\)`](#), [`page.set\_default\_timeout\(timeout\)`](#) 和 [`browser\_context.set\_default\_navigation\_timeout\(timeout\)`](#) 优先于 [`browser\_context.set\_default\_timeout\(timeout\)`](#).

## #

## `browser_context.set_extra_http_headers(headers)` #

---

- `headers` <[`Dict\[str, str\]`](#)> 包含每个请求发送的附加HTTP头的对象。所有头文件的值必须是字符串。 #
- returns: <[`NoneType`](#)> #

额外的HTTP报头将与上下文中的任何页面发起的每个请求一起发送。这些标头与 [page.set\\_extra\\_http\\_headers\(headers\)](#) 设置的特定于页面的额外HTTP标头合并。如果页面覆盖了特定的标题，那么将使用特定于页面的标题值，而不是浏览器上下文标题值。

#### NOTE

[browser\\_context.set\\_extra\\_http\\_headers\(headers\)](#) 不能保证传出请求中报头的顺序。

## #

# browser\_context.set\_geolocation(geolocation)#

---

- `geolocation` [<NoneType|Dict>#](#)
  - `latitude` [<float>](#) 纬度介于-90和90之间。
  - `longitude` [<float>](#) 经度介于-180和180之间。
  - `accuracy` [<float>](#) 非负精度值。默认值为 `0`。
- returns: [<NoneType>#](#)

设置上下文的地理位置。传递 `null` or `undefined` 将模拟位置不可用。

- Sync

```
browser_context.set_geolocation({"latitude": 59.95,  
                                "longitude": 30.31667})
```

- Async

```
await browser_context.set_geolocation({"latitude": 59.95,  
                                       "longitude": 30.31667})
```

#### NOTE

考虑使用 [browser\\_context.grant\\_permissions\(permissions, \\*\\*kwargs\)](#) 授予浏览器上下文页面读取其地理位置的权限。

## # browser\_context.set\_offline(offline)#

---

- `offline` <[bool](#)> 是否为浏览器上下文模拟网络离线.#
- returns: <[NoneType](#)>#

#

## browser\_context.storage\_state(\*\*kwargs)#

---

- `path` <[Union](#)[[str](#), [pathlib.Path](#)]> 存储状态保存到的文件路径. 如果 `path` 是一个相对路径, 那么它是相对于当前工作目录解析的. 如果没有提供路径, 存储状态仍然返回, 但不会保存到磁盘.#
- returns: <[Dict](#)>#
  - `cookies` <[List](#)[[Dict](#)]>
    - `name` <[str](#)>
    - `value` <[str](#)>
    - `domain` <[str](#)>
    - `path` <[str](#)>
    - `expires` <[float](#)> Unix 时间, 单位为秒.
    - `httpOnly` <[bool](#)>
    - `secure` <[bool](#)>
    - `sameSite` <"Strict"|"Lax"|"None"> <“严格”|“宽松”|“没有”>
  - `origins` <[List](#)[[Dict](#)]>
    - `origin` <[str](#)>
    - `localStorage` <[List](#)[[Dict](#)]>
      - `name` <[str](#)>
      - `value` <[str](#)>

返回此浏览器上下文的存储状态, 包含当前cookie和本地存储快照.

## # browser\_context.unroute(url, \*\*kwargs)#

---

- `url` <[str](#)|[Pattern](#)|[Callable](#)[[URL](#)]:[bool](#)> 用于向 [browser\\_context.route\(url, handler, \\*\\*kwargs\)](#) 注册路由的glob模式、regex模式或谓词接收url.#

- `handler` `<Callable[Route, Request]>` 可选处理函数，用于向 `browser_context.route(url, handler, **kwargs)` 注册路由。<#>
- returns: `<NoneType>` <#>

移除使用 `browser_context.route(url, handler, **kwargs)` 创建的路由。当未指定 `handler` 时，删除 `url` 的所有路由。

## # `browser_context.wait_for_event(event, **kwargs)` <#>

---

- `event` `<str>` 事件名称，与通常传递给 `*.on(event)` 的名称相同。<#>
- `predicate` `<Callable>` 接收事件数据，并在等待应该被解析时解析为真值。<#>
- `timeout` `<float>` 最大等待时间，单位为毫秒。默认为 `30000` (30 seconds)。传入 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` 来更改。<#>
- returns: `<Any>` <#>

### NOTE

在大多数情况下，你应该使用 `browser_context.expect_event(event, **kwargs)`。

等待给定 `event` 被触发。如果提供了 `predicate`，它会将事件的值传递给 `predicate(event)` 并等待其返回一个真值。如果浏览器上下文在触发事件之前关闭，则将抛出一个错误。

## # `browser_context.request` <#>

---

- type: `<APIRequestContext>`

与此上下文关联的API测试助手。使用此API发出的请求将使用上下文 cookie。

## # `browser_context.tracing` <#>

---

- type: `<Tracing>`

## BrowserType

BrowserType 提供了启动特定浏览器实例或连接到现有浏览器实例的方法。下面是一个使用Playwright 驱动自动化的典型例子：

- Sync

```
from playwright.sync_api import sync_playwright

def run(playwright):
    chromium = playwright.chromium
    browser = chromium.launch()
    page = browser.new_page()
    page.goto("https://example.com")
    # other actions...
    browser.close()

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

- Async

```
import asyncio
from playwright.async_api import async_playwright

async def run(playwright):
    chromium = playwright.chromium
    browser = await chromium.launch()
    page = await browser.new_page()
    await page.goto("https://example.com")
    # other actions...
    await browser.close()

async def main():
    async with async_playwright() as playwright:
```

```
await run(playwright)
asyncio.run(main())
```

## # browser\_type.connect(ws\_endpoint, \*\*kwargs) #

---

- `ws_endpoint` <[str](#)> 要连接的浏览器websocket端点。<#>
- `headers` <[Dict](#)[[str](#), [str](#)]> websocket 连接请求发送的额外HTTP头。可选的。<#>
- `slow_mo` <[float](#)> 使 playwright 操作变慢指定的毫秒数。很有用，这样你就能知道发生了什么。默认为0。<#>
- `timeout` <[float](#)> 等待连接建立的最大时间(单位:毫秒)。默认为 `30000` (30 seconds). 传递 `0` 以禁用超时。<#>
- returns: <[Browser](#)> <#>

该方法将playwright 附加到一个现有的浏览器实例。

## # browser\_type.connect\_over\_cdp(endpoint\_url, \*\*kwargs) #

---

- `endpoint_url` <[str](#)> 要连接CDP的 websocket端点或 http url。例如 `http://localhost:9222/` or `ws://127.0.0.1:9222/devtools/browser/387adf4c-243f-4051-a181-46798f4a46f4`。<#>
- `headers` <[Dict](#)[[str](#), [str](#)]> 连接请求发送的额外HTTP头。可选的。<#>
- `slow_mo` <[float](#)> 使 playwright 操作变慢指定的毫秒数。很有用，这样你就能知道发生了什么。默认为0。<#>
- `timeout` <[float](#)> 等待连接建立的最大时间(单位:毫秒)。默认为 `30000` (30 seconds). 传递 `0` 以禁用超时。<#>
- returns: <[Browser](#)> <#>

该方法使用Chrome DevTools协议将playwright附加到一个现有的浏览器实例。

默认的浏览器上下文可以通过 [browser.contexts](#) 访问。



## # browser\_type.executable\_path#

---

- returns: `<str>` #

Playwright 希望在这个路径中找到捆绑的浏览器可执行文件。

## # browser\_type.launch(\*\*kwargs)#

---

- `args` `<List[str]>` 传递给浏览器实例的其他参数。Chrome的列表可以在[这里](#)找到。#
- `channel` `<str>` 浏览器分发通道。支持的值为 "chrome", "chrome-beta", "chrome-dev", "chrome-canary", "msedge", "msedge-beta", "msedge-dev", "msedge-canary". 阅读更多关于使用[Google Chrome](#)和[Microsoft Edge](#)的方式。#
- `chromium_sandbox` `<bool>` 启用Chromium沙箱。默认为 `false`。#
- `devtools` `<bool>` **Chromium-only** 是否为每个选项卡自动打开开发人员工具面板。如果该选项为 `true`, 则 `headless` 选项将被设为 `false`。#
- `downloads_path` `<Union[str, pathlib.Path]>` 如果指定, 接受的下载将被下载到此目录, 否则, 将在关闭浏览器时创建并删除临时目录。在这两种情况下, 当创建下载的浏览器上下文关闭时, 下载就会被删除。#
- `env` `<Dict[str, str|float|bool]>` 指定浏览器可见的环境变量。默认为 `process.env`。#
- `executable_path` `<Union[str, pathlib.Path]>` 浏览器可执行文件的运行路径, 而不是绑定文件。如果 `executable_path` 是一个相对路径, 那么它是相对于当前工作目录进行解析的。请注意, playwright 只适用于捆绑的Chromium, Firefox或WebKit, 使用风险自负。#
- `firefox_user_prefs` `<Dict[str, str|float|bool]>` Firefox用户首选项。在[about:config](#)了解更多关于Firefox用户首选项的信息。#
- `handle_sighup` `<bool>` 在SIGHUP上关闭浏览器进程。默认值为 `true`。#

- `handle_sigint` `<bool>` 在Ctrl-C上关闭浏览器进程。默认值为 `true`.#
- `handle_sigterm` `<bool>` 关闭SIGTERM上的浏览器进程。默认值为 `true`.#
- `headless` `<bool>` 是否以无头模式运行浏览器。更多关于 [Chromium](#) and [Firefox](#) 的细节。默认为 `true` 除非 `devtools` 选项为 `true`.#
- `ignore_default_args` `<bool|List[str]>` 如果为 `true`, Playwright 不会传递自己的配置参数, 而只使用来自 `args` 的配置参数. 如果给出了数组, 则过滤掉给定的默认参数。危险的选择, 小心使用。默认值为 `false`.#
- `proxy` `<Dict>` 网络代理设置.#
  - `server` `<str>` 所有请求使用的代理。支持HTTP代理和SOCKS代理, 例如: `http://myproxy.com:3128` or `socks5://myproxy.com:3128`. 缩写形式 `myproxy.com:3128` 被认为是一个HTTP代理.
  - `bypass` `<str>` 可选, 用逗号分隔的域, 绕过代理, 例如 `".com, chromium.org, .domain.com"`.
  - `username` `<str>` 可选username, 当HTTP代理需要鉴权时使用.
  - `password` `<str>` 当HTTP代理需要鉴权时可选密码.
- `slow_mo` `<float>` 使 playwright 操作变慢指定的毫秒数。可以让您看到正在发生什么.#
- `timeout` `<float>` 浏览器实例启动的最大等待时间(单位为毫秒)。默认为 `30000` (30 seconds). 传递 `0` 以禁用超时.#
- `traces_dir` `<Union[str, pathlib.Path]>` 如果指定, 跟踪记录将被保存到该目录.#
- `returns:` `<Browser>` .#

返回浏览器实例.

你可以使用 `ignore_default_args` 从默认参数中过滤出 `--mute-audio`:

- Sync

```
browser = playwright.chromium.launch( # or "firefox" or
    "webkit".
    ignore_default_args=["--mute-audio"]
)
```

- Async

```
browser = await playwright.chromium.launch( # or "firefox" or
"webkit".
    ignore_default_args=["--mute-audio"]
)
```

**Chromium-only Playwright** 也可以用来控制谷歌Chrome或微软Edge浏览器, 但它最好的版本与 Chromium 捆绑. 不能保证它能与任何其他版本一起工作. 使用 `executable_path` 选项时要特别小心.

如果 Google Chrome (而不是 Chromium) 是首选, 则建议使用 [Chrome Canary](#) or [Dev Channel](#) .

像 Google Chrome 和 Microsoft Edge 这样的常用浏览器适合于需要专用媒体编解码器进行视频播放的测试. Chromium and Chrome 的其他区别见 [这篇文章](#). [本文](#) 描述了Linux用户的一些差异.

#

## `browser_type.launch_persistent_context(user_data_dir, **kwargs)` #

- `user_data_dir` <[Union](#)[[str](#), [pathlib.Path](#)]> 用户数据目录的路径, 用于存储浏览器会话数据, 如cookie和本地存储. 更多关于 [Chromium](#) and [Firefox](#) 的细节. 注意 Chromium 的用户数据目录是父目录的“配置文件路径” `chrome://version`. 传递一个空字符串来使用临时目录. #
- `accept_downloads` <[bool](#)> 是否自动下载所有附件. 在所有下载都被接受的地方默认为 `true` . #
- `args` <[List](#)[[str](#)]> 传递给浏览器实例的其他参数. Chromium 标志的列表可以在 [这里](#) 找到. #
- `base_url` <[str](#)> 使用 [page.goto\(url, \\*\\*kwargs\)](#), [page.route\(url, handler, \\*\\*kwargs\)](#), [page.wait\\_for\\_url\(url, \\*\\*kwargs\)](#), [page.expect\\_request\(url\\_or\\_predicate, \\*\\*kwargs\)](#), or [page.expect\\_response\(url\\_or\\_predicate, \\*\\*kwargs\)](#) 通过使用 `URL()` 构造函数构建相应的URL来考虑基URL : #
  - baseURL= `http://localhost:3000` 时, 导航到 `/bar.html` 的结果为

- `http://localhost:3000/bar.html`
  - baseURL= `http://localhost:3000/foo/` 时,导航到 `./bar.html` 的结果为 `http://localhost:3000/foo/bar.html`
  - baseURL= `http://localhost:3000/foo` 时,导航到(没有下划线) `./bar.html` 的结果为 `http://localhost:3000/bar.html`
- `bypass_csp` <[bool](#)> 切换绕过页面的 Content-Security-Policy. <#>
- `channel` <[str](#)> 浏览器分发通道。支持的值为 "chrome", "chrome-beta", "chrome-dev", "chrome-canary", "msedge", "msedge-beta", "msedge-dev", "msedge-canary". 阅读更多关于使用 [Google Chrome](#) 和 [Microsoft Edge](#). <#>
- `chromium_sandbox` <[bool](#)> 启用Chromium 沙箱. 默认为 `false`. <#>
- `color_scheme` <"light"|"dark"|"no-preference"> 模拟 '`prefers-colors-scheme`' 的媒体特性, 支持的值为 '`light`', '`dark`', '`no-preference`'. 详情请参阅: [page.emulate\\_media\(\\*\\*kwargs\)](#). 默认为 '`light`'. <#>
- `device_scale_factor` <[float](#)> 指定设备比例因子(可以认为是 dpr)。默认为 `1`. <#>
- `devtools` <[bool](#)> **Chromium-only** 是否为每个选项卡自动打开开发人员工具面板。如果该选项为 `true`, 则 `headless` 选项将被设为 `false`. <#>
- `downloads_path` <[Union](#)[[str](#), [pathlib.Path](#)]> 如果指定, 接受的下载将被下载到此目录, 否则, 将在关闭浏览器时创建并删除临时目录。在这两种情况下, 当创建下载的浏览器上下文关闭时, 下载就会被删除. <#>
- `env` <[Dict](#)[[str](#), [str](#)|[float](#)|[bool](#)]> 指定浏览器可见的环境变量。默认为 `process.env`. <#>
- `executable_path` <[Union](#)[[str](#), [pathlib.Path](#)]> 浏览器可执行文件的运行路径, 而不是绑定文件. 如果 `executable_path` 是一个相对路径, 那么它是相对于当前工作目录进行解析的。请注意, playwright 只适用于捆绑的Chromium, Firefox或WebKit, 使用风险自负. <#>
- `extra_http_headers` <[Dict](#)[[str](#), [str](#)]> 一个包含附加HTTP头的对象, 每个请求都要发送. <#>
- `forced_colors` <"active"|"none"> 模拟 '`forced-colors`' 媒体特性, 支持的值为 '`active`', '`none`'. 详情请参阅: [page.emulate\\_media\(\\*\\*kwargs\)](#). 默认为 '`none`'. <#>

它在WebKit中不支持，请在他们的[问题](#)跟踪器中查看

- `geolocation` [<Dict>#](#)
  - `latitude` [<float>](#) 纬度介于-90和90之间。
  - `longitude` [<float>](#) 经度介于-180和180之间。
  - `accuracy` [<float>](#) 非负精度值。默认值为 `0`。
- `handle_sighup` [<bool>](#) 在SIGHUP上关闭浏览器进程。默认值为 `true` [.#](#)
- `handle_sigint` [<bool>](#) 在Ctrl-C上关闭浏览器进程。默认值为 `true` [.#](#)
- `handle_sigterm` [<bool>](#) 关闭SIGTERM上的浏览器进程。默认值为 `true` [.#](#)
- `has_touch` [<bool>](#) 指定视口是否支持触摸事件。默认值为 `false` [.#](#)
- `headless` [<bool>](#) 是否以无头模式运行浏览器。更多关于 [Chromium](#) and [Firefox](#) 的细节。默认为 `true` 除非 `devtools` 选项为 `true` [.#](#)
- `http_credentials` [<Dict>](#) HTTP认证凭据 [HTTP authentication.#](#)
  - `username` [<str>](#)
  - `password` [<str>](#)
- `ignore_default_args` [<bool|List\[str\]>](#) 如果为 `true`，不会传递自己的配置参数，而只使用来自 `args` 的配置参数。如果给出了数组，则过滤掉给定的默认参数。危险的选择，小心使用。默认值为 `false` [.#](#)
- `ignore_https_errors` [<bool>](#) 发送网络请求时是否忽略HTTPS错误。默认值为 `false` [.#](#)
- `is_mobile` [<bool>](#) 是否考虑 `meta viewport` 标签，是否启用触摸事件。默认值为 `false`。Firefox中不支持 [.#](#)
- `javascript_enabled` [<bool>](#) 是否在上下文中启用JavaScript。默认值为 `true` [.#](#)
- `locale` [<str>](#) 指定用户的本地语言环境，例如 `en-GB`，`de-DE`，等。区域设置将影响 `navigator.language` 的值，`Accept-Language` 请求头值以及数字和日期格式规则 [.#](#)
- `no_viewport` [<bool>](#) 不强制固定viewport，允许在头部模式下调整窗口大小 [.#](#)
- `offline` [<bool>](#) 是否仿真网络离线。默认值为 `false` [.#](#)

- `permissions` `<List[str]>` 在此上下文中授予所有页面的权限列表。  
详细常看: [browser\\_context.grant\\_permissions\(permissions, \\*\\*kwargs\).](#)`#`
- `proxy` `<Dict>` 网络代理设置.`#`
  - `server` `<str>` 所有请求使用的代理。支持HTTP代理和SOCKS代理, 例如: `http://myproxy.com:3128` or `socks5://myproxy.com:3128`. 缩写 `myproxy.com:3128` 被认为是一个HTTP代理.
  - `bypass` `<str>` 可选, 用逗号分隔的域, 绕过代理, 例如 `".com, chromium.org, .domain.com"`.
  - `username` `<str>` 可选username, 当HTTP代理需要鉴权时使用.
  - `password` `<str>` 当HTTP代理需要鉴权时可选密码.
- `record_har_omit_content` `<bool>` 可选设置, 控制是否从HAR忽略请求内容。默认值为 `false` .`#`
- `record_har_path` `<Union[str, pathlib.Path]>` 为文件系统中指定的 `HAR` 文件中所有页面启用HAR记录. 如果没有指定, 则不会记录HAR。确保调用 [browser\\_context.close\(\)](#) 来保存HAR.`#`
- `record_video_dir` `<Union[str, pathlib.Path]>` 开启进入指定目录的所有页面的视频录制. 如果没有指定, 则不录制视频。确保调用 [browser\\_context.close\(\)](#) 来保存视频.`#`
- `record_video_size` `<Dict>` 录制视频的尺寸。如果没有指定大小将等于 `viewport` 缩小到800x800。如果 `viewport` 没有显式配置, 视频大小默认为800x450。每个页面的实际图片将按比例缩小, 如果需要, 以适应指定的大小.`#`
  - `width` `<int>` 视频帧宽度.
  - `height` `<int>` 视频帧高度.
- `reduced_motion` `<"reduce"|"no-preference">` 模拟 `'prefers-reduced-motion'` 媒体特性, 支持的值为 `'reduce'`, `'no-preference'`. 详情请参阅 [page.emulate\\_media\(\\*\\*kwargs\)](#). 默认为 `'no-preference'` .`#`
- `screen` `<Dict>` 通过 `window.screen` .在web页面中模拟一致的窗口屏幕大小。仅在设置 `viewport` 时使用.
  - `width` `<int>` 页面宽度(px).
  - `height` `<int>` 页面高度(px).
- `slow_mo` `<float>` 使 playwright 操作变慢指定的毫秒数。可以让您看到正在发生什么.`#`

- `strict_selectors` `<bool>` 它指定了，为这个上下文启用严格选择器模式。在严格的选择器模式中，当有多个元素匹配选择器时，所有对选择器的操作都将意味着只有一个目标DOM元素。请参阅 [Locator](#) 以了解更多关于严格模式。<#>
- `timeout` `<float>` 浏览器实例启动的最大等待时间(单位为毫秒)。默认为 `30000` (30 seconds). 传递 `0` 以禁用超时。<#>
- `timezone_id` `<str>` 修改上下文的时区。查看 [ICU's metaZones.txt](#) 获取支持的时区id列表。<#>
- `traces_dir` `<Union[str, pathlib.Path]>` 如果指定，跟踪记录将被保存到该目录。<#>
- `user_agent` `<str>` 在此上下文中使用的特定用户代理。<#>
- `viewport` `<NoneType|Dict>` 为每个页面设置一个一致的视口。默认为1280x720视口。 `no_viewport` 禁用固定视口。<#>
  - `width` `<int>` 页面宽度(px).
  - `height` `<int>` 页面高度(px).
- returns: `<BrowserContext>` <#>

返回持久的浏览器上下文实例。

启动浏览器，使用位于 `user_data_dir` 的持久存储，并返回唯一的上下文。关闭此上下文将自动关闭浏览器。

## `# browser_type.name` <#>

---

- returns: `<str>` <#>

返回浏览器的名称。例如: `'chromium'`, `'webkit'` or `'firefox'` .

## CDPSession

- extends: [EventEmitter](#)

`CDPSession` 实例用于谈论原始的Chrome Devtools协议:



- 协议方法可以用会话调用 `session.send` .
- 协议事件可以通过会话订阅 `session.on` 上的方法.

有用的链接:

- DevTools协议的文档可以在这里找到: [DevTools Protocol Viewer](#).
- 开始使用DevTools协议: <https://github.com/aslushnikov/getting-started-with-cdp/blob/master/README.md>
- Sync

```
client = page.context.new_cdp_session(page)
client.send("Animation.enable")
client.on("Animation.animationCreated", lambda:
print("animation created!"))
response = client.send("Animation.getPlaybackRate")
print("playback rate is " + str(response["playbackRate"]))
client.send("Animation.setPlaybackRate", {
    playbackRate: response["playbackRate"] / 2
})
```

- Async

```
client = await page.context.new_cdp_session(page)
await client.send("Animation.enable")
client.on("Animation.animationCreated", lambda:
print("animation created!"))
response = await client.send("Animation.getPlaybackRate")
print("playback rate is " + str(response["playbackRate"]))
await client.send("Animation.setPlaybackRate", {
    playbackRate: response["playbackRate"] / 2
})
```

- [cdp\\_session.detach\(\)](#)
- [cdp\\_session.send\(method, \\*\\*kwargs\)](#)

## # cdp\_session.detach()#

---

- returns: [<NoneType>](#) #



将CDPSession与目标分离。一旦分离，CDPSession对象将不会发出任何事件，也不能用于发送消息。

## # cdp\_session.send(method, \*\*kwargs)#

---

- `method` <str> 协议方法名#
- `params` <Dict> 可选方法参数#
- returns: <Dict>#

### ConsoleMessage

[ConsoleMessage](#) 对象通过 [page.on\("console"\)](#) 事件调度。

- [console\\_message.args](#)
- [console\\_message.location](#)
- [console\\_message.text](#)
- [console\\_message.type](#)

## # console\_message.args#

---

- returns: <List[[JSHandle](#)]>#

传递给 `console` 函数调用的参数列表。参见 [page.on\("console"\)](#)。

## # console\_message.location#

---

- returns: <Dict>#
  - `url` <str> 资源的url.
  - `lineNumber` <int> 0-based 资源的行号.
  - `columnNumber` <int> 0-based 资源的列号.

## # console\_message.text#

---

- returns: <[str](#)>#

控制台消息的文本.

## # console\_message.type#

---

- returns: <[str](#)>#

以下值之一: 'log', 'debug', 'info', 'error', 'warning', 'dir', 'dirxml', 'table', 'trace', 'clear', 'startGroup', 'startGroupCollapsed', 'endGroup', 'assert', 'profile', 'profileEnd', 'count', 'timeEnd'.

### Dialog

[Dialog](#) 对象通过 [page.on\("dialog"\)](#) 事件分配.

使用 [Dialog](#) 类的一个例子:

- Sync

```
from playwright.sync_api import sync_playwright

def handle_dialog(dialog):
    print(dialog.message)
    dialog.dismiss()

def run(playwright):
    chromium = playwright.chromium
    browser = chromium.launch()
    page = browser.new_page()
    page.on("dialog", handle_dialog)
    page.evaluate("alert('1')")
```

```
browser.close()

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

- Async

```
import asyncio
from playwright.async_api import async_playwright

async def handle_dialog(dialog):
    print(dialog.message)
    await dialog.dismiss()

async def run(playwright):
    chromium = playwright.chromium
    browser = await chromium.launch()
    page = await browser.new_page()
    page.on("dialog", handle_dialog)
    page.evaluate("alert('1')")
    await browser.close()

async def main():
    async with async_playwright() as playwright:
        await run(playwright)
    asyncio.run(main())
```

#### NOTE

除非有 [`page.on\("dialog"\)`](#) 监听器，否则对话框会自动被驳回。当监听器存在时，它必须要么 [`dialog.accept\(\*\*kwargs\)`](#) 要么 [`dialog.dismiss\(\)`](#) 否则页面将冻结等待对话框，像`click`这样的动作将永远不会结束。

- [`dialog.accept\(\*\*kwargs\)`](#)
- [`dialog.default\_value`](#)
- [`dialog.dismiss\(\)`](#)
- [`dialog.message`](#)
- [`dialog.type`](#)

## # dialog.accept(\*\*kwargs)#

---

- `prompt_text` `<str>` 提示符中要输入的文本。如果对话框的类型不是提示符，则不会造成任何影响。可选的.#
- returns: `<NoneType>` #

当对话框被接受时返回。

## # dialog.default\_value#

---

- returns: `<str>` #

如果对话框是提示符，则返回默认的提示值。否则，返回空字符串。

## # dialog.dismiss()#

---

- returns: `<NoneType>` #

当对话框被驳回时返回。

## # dialog.message#

---

- returns: `<str>` #

对话框中显示的消息。

## # dialog.type#

---

- returns: `<str>` #

返回对话框的类型，可以是 `alert`，`beforeunload`，`confirm` or `prompt`。

## Download

[Download](#) 对象通过 [page.on\("download"\)](#) 事件被页面分派。

当浏览器上下文关闭时，属于浏览器上下文的所有下载文件将被删除。

一旦下载开始，就会触发Download事件。下载完成后，下载路径变为可用：

- Sync

```
with page.expect_download() as download_info:
    page.click("a")
download = download_info.value
# wait for download to complete
path = download.path()
```

- Async

```
async with page.expect_download() as download_info:
    await page.click("a")
download = await download_info.value
# waits for download to complete
path = await download.path()
```

- [download.cancel\(\)](#)
- [download.delete\(\)](#)
- [download.failure\(\)](#)
- [download.page](#)
- [download.path\(\)](#)
- [download.save\\_as\(path\)](#)
- [download.suggested\\_filename](#)
- [download.url](#)

## # download.cancel()#

---

- returns: [<NoneType>](#)#

取消下载。如果下载已经完成或取消，则不会失败。成功取消后，

`download.failure()` 将解析为 `'canceled'`。

## # download.delete() #

---

- returns: <[NoneType](#)> #

删除下载的文件。如果需要，将等待下载完成。

## # download.failure() #

---

- returns: <[NoneType](#)|[str](#)> #

如果有，返回下载错误。如果需要，将等待下载完成。

## # download.page #

---

- returns: <[Page](#)> #

获取下载文件所属的页面。

## # download.path() #

---

- returns: <[NoneType](#)|[pathlib.Path](#)> #

如果下载成功，返回下载文件的路径。如果需要，该方法将等待下载完成。该方法在远程连接时排除。

请注意，下载的文件名是一个随机的GUID，使用 [download.suggested\\_filename](#) 获取建议文件名。

## # download.save\_as(path) #

---

- `path` <[Union](#)[[str](#), [pathlib.Path](#)]> 下载应该被复制的路径. #
- returns: <[NoneType](#)> #

将下载文件复制到用户指定的路径。在下载过程中调用此方法是安全的。如果需要，将等待下载完成。

## # download.suggested\_filename#

---

- returns: <[str](#)>#

返回此下载的建议文件名。它通常是由浏览器从 [Content-Disposition](#) 响应头或下载属性计算出来的。看规范是什么。不同的浏览器可以使用不同的逻辑来计算它。

## # download.url#

---

- returns: <[str](#)>#

返回下载url。

## ElementHandle

- extends: [JSHandle](#)

ElementHandle 表示一个页面内的DOM元素。可以用页面创建 [page.query\\_selector\(selector, \\*\\*kwargs\)](#) 方法。

*DISCOURAGED*

不鼓励使用 *ElementHandle*，而是使用 [Locator](#) 对象和 web 优先断言。

- Sync

```
href_element = page.query_selector("a")
href_element.click()
```

- Async

```
href_element = await page.query_selector("a")
await href_element.click()
```

ElementHandle prevents DOM element from garbage collection unless the handle is disposed with [js\\_handle.dispose\(\)](#). ElementHandles are auto-disposed when their origin frame gets navigated.

ElementHandle instances can be used as an argument in [page.eval\\_on\\_selector\(selector, expression, \\*\\*kwargs\)](#) and [page.evaluate\(expression, \\*\\*kwargs\)](#) methods.

The difference between the [Locator](#) and ElementHandle is that the ElementHandle points to a particular element, while [Locator](#) captures the logic of how to retrieve an element.

In the example below, handle points to a particular DOM element on page. If that element changes text or is used by React to render an entirely different component, handle is still pointing to that very DOM element. This can lead to unexpected behaviors.

- Sync

```
handle = page.query_selector("text=Submit")
handle.hover()
handle.click()
```

- Async

```
handle = await page.query_selector("text=Submit")
await handle.hover()
await handle.click()
```

With the locator, every time the `element` is used, up-to-date DOM element is located in the page using the selector. So in the snippet below, underlying DOM element is going to be located twice.

- Sync



```
locator = page.locator("text=Submit")
locator.hover()
locator.click()
```

- Async

```
locator = page.locator("text=Submit")
await locator.hover()
await locator.click()
```

- [element\\_handle.bounding\\_box\(\)](#)
- [element\\_handle.check\(\\*\\*kwargs\)](#)
- [element\\_handle.click\(\\*\\*kwargs\)](#)
- [element\\_handle.content\\_frame\(\)](#)
- [element\\_handle.dblclick\(\\*\\*kwargs\)](#)
- [element\\_handle.dispatch\\_event\(type, \\*\\*kwargs\)](#)
- [element\\_handle.eval\\_on\\_selector\(selector, expression, \\*\\*kwargs\)](#)
- [element\\_handle.eval\\_on\\_selector\\_all\(selector, expression, \\*\\*kwargs\)](#)
- [element\\_handle.fill\(value, \\*\\*kwargs\)](#)
- [element\\_handle.focus\(\)](#)
- [element\\_handle.get\\_attribute\(name\)](#)
- [element\\_handle.hover\(\\*\\*kwargs\)](#)
- [element\\_handle.inner\\_html\(\)](#)
- [element\\_handle.inner\\_text\(\)](#)
- [element\\_handle.input\\_value\(\\*\\*kwargs\)](#)
- [element\\_handle.is\\_checked\(\)](#)
- [element\\_handle.is\\_disabled\(\)](#)
- [element\\_handle.is\\_editable\(\)](#)
- [element\\_handle.is\\_enabled\(\)](#)
- [element\\_handle.is\\_hidden\(\)](#)
- [element\\_handle.is\\_visible\(\)](#)
- [element\\_handle.owner\\_frame\(\)](#)
- [element\\_handle.press\(key, \\*\\*kwargs\)](#)
- [element\\_handle.query\\_selector\(selector\)](#)
- [element\\_handle.query\\_selector\\_all\(selector\)](#)
- [element\\_handle.screenshot\(\\*\\*kwargs\)](#)
- [element\\_handle.scroll\\_into\\_view\\_if\\_needed\(\\*\\*kwargs\)](#)
- [element\\_handle.select\\_option\(\\*\\*kwargs\)](#)
- [element\\_handle.select\\_text\(\\*\\*kwargs\)](#)
- [element\\_handle.set\\_checked\(checked, \\*\\*kwargs\)](#)
- [element\\_handle.set\\_input\\_files\(files, \\*\\*kwargs\)](#)

- [`element\_handle.tap\(\*\*kwargs\)`](#)
- [`element\_handle.text\_content\(\)`](#)
- [`element\_handle.type\(text, \*\*kwargs\)`](#)
- [`element\_handle.uncheck\(\*\*kwargs\)`](#)
- [`element\_handle.wait\_for\_element\_state\(state, \*\*kwargs\)`](#)
- [`element\_handle.wait\_for\_selector\(selector, \*\*kwargs\)`](#)
- [`js\_handle.as\_element\(\)`](#)
- [`js\_handle.dispose\(\)`](#)
- [`js\_handle.evaluate\(expression, \*\*kwargs\)`](#)
- [`js\_handle.evaluate\_handle\(expression, \*\*kwargs\)`](#)
- [`js\_handle.get\_properties\(\)`](#)
- [`js\_handle.get\_property\(property\_name\)`](#)
- [`js\_handle.json\_value\(\)`](#)

## # `element_handle.bounding_box()` #

---

- returns: `<NoneType|Dict>#`
  - `x` `<float>` 元素的 X 坐标像素。
  - `y` `<float>` 元素的Y坐标，以像素为单位。
  - `width` `<float>` 元素的宽度，以像素为单位。
  - `height` `<float>` 元素的高度，以像素为单位。

此方法返回元素的边界框，如果元素不可见，则返回 `null`。边界框是相对于主帧视口计算的——主帧视口通常与浏览器窗口相同。

滚动会影响返回的绑定框，类似于 [`Element.getBoundingClientRect`](#)，这意味着 `x`和/或`y`可能是负的。

与 [`Element.getBoundingClientRect`](#) 不同，子`frame` 中的元素返回相对于主`frame` 的边界框。

假设页面是静态的，使用边界框坐标执行输入是安全的。例如，下面的代码片段应该单击元素的中心。

- Sync

```
box = element_handle.bounding_box()
page.mouse.click(box["x"] + box["width"] / 2, box["y"] +
box["height"] / 2)
```

- Async

```
box = await element_handle.bounding_box()
await page.mouse.click(box["x"] + box["width"] / 2, box["y"] +
box["height"] / 2)
```

## # `element_handle.check(**kwargs)` #

- `force` <bool> 是否绕过 [actionability](#) 检查。默认值为 `false` .#
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- `position` <Dict> 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点 .#
  - `x` <float>
  - `y` <float>
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改 .#
- `trial` <bool> 设置后，该方法只执行 [actionability](#) 检查，并跳过操作。默认值为 `false` 。在元素准备好时再执行动作是很有用的 .#
- `returns`: <NoneType> .#

这个方法通过执行以下步骤来检查元素：

1. 确保元素是一个复选框或单选输入。如果不是，则抛出此方法。如果元素已被选中，则该方法立即返回。
2. 等待元素的 [可操作性](#) 检查，除非设置了强制选项。
3. 如果需要，将元素滚动到视图中。
4. 使用 [page.mouse](#) 单击元素的中心。
5. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项。
6. 确保元素现在被选中。如果不是，则排除此方法。

如果元素在动作期间的任何时刻与DOM分离，此方法将抛出。

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#) 。传递零超时将禁用此功能。

## # element\_handle.click(\*\*kwargs) #

- `button` <"left"|"right"|"middle"> 默认左 `left` .#
- `click_count` <int> 默认为1, 详情查看 [UIEvent.detail](#) .#
- `delay` <float> `mousedown` 和 `mouseup` 之间的等待时间, 单位是毫秒。默认为0. #
- `force` <bool> 是否绕过 [actionability](#) 检查。默认值为 `false` .#
- `modifiers` <List["Alt"|"Control"|"Meta"|"Shift"]> `modifiers` 按键要按。确保在操作期间只按下这些修饰符, 然后恢复当前的修饰符。如果未指定, 则使用当前按下的修饰符. #
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生, 并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项, 比如导航到不可访问的页面。默认值为 `false` .#
- `position` <Dict> 相对于元素填充框的左上角使用的一个点。如果没有指定, 则使用元素的某个可见点. #
  - `x` <float>
  - `y` <float>
- `timeout` <float> 最大时间, 单位为毫秒, 默认为30秒, 通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改. #
- `trial` <bool> 设置后, 该方法只执行 [actionability](#) 检查, 并跳过操作。默认值为 `false` 。在元素准备好时再执行动作是很有用的. #
- `returns`: <NoneType> .#

该方法执行以下步骤单击元素:

1. 等待元素的 [可操作性](#) 检查, 除非设置了强制选项。
2. 如果需要, 将元素滚动到视图中。
3. 使用 [page.mouse](#) 单击元素的中心, or the specified `position` .
4. 等待已启动的导航成功或失败, 除非设置了 `no_wait_after` 选项。

如果元素在动作期间的任何时刻与DOM分离, 此方法将抛出。

如果在指定的超时期间, 所有步骤组合都没有完成, 则该方法将抛出一个 [TimeoutError](#) 。传递零超时将禁用此功能。

## # element\_handle.content\_frame()#

---

- returns: <NoneType|Frame>#

Returns the content frame for element handles referencing iframe nodes, or `null` otherwise

## # element\_handle.dblclick(\*\*kwargs)#

---

- `button` <"left"|"right"|"middle"> 默认左 `left`.#
- `delay` <float> `mousedown` 和 `mouseup` 之间的等待时间, 单位是毫秒。默认为0.#
- `force` <bool> 是否绕过 `actionability` 检查。默认值为 `false`.#
- `modifiers` <List["Alt"|"Control"|"Meta"|"Shift"]> `modifiers` 按键要按。确保在操作期间只按下这些修饰符, 然后恢复当前的修饰符。如果未指定, 则使用当前按下的修饰符.#
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生, 并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项, 比如导航到不可访问的页面。默认值为 `false`.#
- `position` <Dict> 相对于元素填充框的左上角使用的一个点。如果没有指定, 则使用元素的某个可见点.#
  - `x` <float>
  - `y` <float>
- `timeout` <float> 最大时间, 单位为毫秒, 默认为30秒, 通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改.#
- `trial` <bool> 设置后, 该方法只执行 `actionability` 检查, 并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的.#
- returns: <NoneType>#

该方法执行以下步骤双击元素:

1. 等待元素的 `可操作性` 检查, 除非设置了强制选项。
2. 如果需要, 将元素滚动到视图中。
3. 使用 `page.mouse` 方法, 双击元素中心位置。

4. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项.该方法执行以下步骤双击元素, 注意，如果 `dblclick()` 的第一次单击触发了一个导航事件，则该方法将抛出.

如果元素在动作期间的任何时刻与DOM分离，此方法将抛出.

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

#### NOTE

`elementHandle.dblclick()` dispatches two `click` events and a single `dblclick` event.

## # element\_handle.dispatch\_event(type, \*\*kwargs) #

- `type` <str> DOM事件类型: `"click"`, `"dragstart"` 等. #
- `event_init` <EvaluationArgument> 可选的特定于事件的初始化属性. #
- returns: <NoneType> #

下面的代码片段分派元素上的 `单击` 事件。无论元素的可见性状态如何，单击都将被分派。这相当于调用 [element.click\(\)](#)。

- Sync

```
element_handle.dispatch_event("click")
```

- Async

```
await element_handle.dispatch_event("click")
```

在底层，它根据给定的类型创建一个事件实例，使用 `event_init` 属性初始化它，并在元素上分派它。默认情况下，事件是组合的、可取消的和冒泡的。

由于 `event_init` 是特定于事件的，请参考事件文档中的初始属性列表：

- [DragEvent](#)

- [FocusEvent](#)
- [KeyboardEvent](#)
- [MouseEvent](#)
- [PointerEvent](#)
- [TouchEvent](#)
- [Event](#)

如果你想要将活动对象传递到事件中，你也可以指定 `jhandle` 作为属性值：

- Sync

```
# note you can only create data_transfer in chromium and
firefox
data_transfer = page.evaluate_handle("new DataTransfer()")
element_handle.dispatch_event("#source", "dragstart",
{"dataTransfer": data_transfer})
```

- Async

```
# note you can only create data_transfer in chromium and
firefox
data_transfer = await page.evaluate_handle("new
DataTransfer()")
await element_handle.dispatch_event("#source", "dragstart",
{"dataTransfer": data_transfer})
```

## #

## `element_handle.eval_on_selector(selector, expression, **kwargs)` <#>

---

- `selector` [<str>](#) 要查询的选择器。有关更多细节，请参阅[working with selectors](#). <#>
- `expression` [<str>](#) 要在浏览器上下文中计算的 `JavaScript` 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值. <#>
- `arg` [<EvaluationArgument>](#) 传递给 `expression` 的可选参数. <#>
- returns: [<Serializable>](#) <#>

返回表达式的返回值。

The method finds an element matching the specified selector in the `ElementHandle`'s subtree and passes it as a first argument to `expression`. See [Working with selectors](#) for more details. If no elements match the selector, the method throws an error.

如果 `expression` 返回 `Promise`，那么 `element_handle.eval_on_selector(selector, expression, **kwargs)` 将等待promise解析并返回它的值。

Examples:

- Sync

```
tweet_handle = page.query_selector(".tweet")
assert tweet_handle.eval_on_selector(".like", "node =>
node.innerText") = "100"
assert tweet_handle.eval_on_selector(".retweets", "node =>
node.innerText") = "10"
```

- Async

```
tweet_handle = await page.query_selector(".tweet")
assert await tweet_handle.eval_on_selector(".like", "node =>
node.innerText") = "100"
assert await tweet_handle.eval_on_selector(".retweets", "node
=> node.innerText") = "10"
```

#

## `element_handle.eval_on_selector_all(selector, expression, **kwargs)` <#>

---

- `selector` `<str>` 要查询的选择器。有关更多细节，请参阅[working with selectors](#). <#>
- `expression` `<str>` 要在浏览器上下文中计算的 `JavaScript` 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值。 <#>
- `arg` `<EvaluationArgument>` 传递给 `expression` 的可选参数。 <#>
- returns: `<Serializable>` <#>



返回表达式的返回值。

The method finds all elements matching the specified selector in the `ElementHandle`'s subtree and passes an array of matched elements as a first argument to `expression`. See [Working with selectors](#) for more details.

如果 `expression` 返回 [Promise](#)，那么 `element_handle.eval_on_selector_all(selector, expression, **kwargs)` 将等待 promise 解析并返回它的值。

Examples:

```
\<div class="feed">
  \<div class="tweet">Hello!\</div>
  \<div class="tweet">Hi!\</div>
\</div>
```

- Sync

```
feed_handle = page.query_selector(".feed")
assert feed_handle.eval_on_selector_all(".tweet", "nodes =>
nodes.map(n => n.innerText)") = ["hello!", "hi!"]
```

- Async

```
feed_handle = await page.query_selector(".feed")
assert await feed_handle.eval_on_selector_all(".tweet", "nodes
=> nodes.map(n => n.innerText)") = ["hello!", "hi!"]
```

## # `element_handle.fill(value, **kwargs)` #

- `value` <str> Value to set for the `\<input>`, `\<textarea>` or `[contenteditable]` element. #
- `force` <bool> 是否绕过 [actionability](#) 检查。默认值为 `false`. #
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false`. #

- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。`#`
- returns: `<NoneType>` `#`

这个方法等待 `可操作性` 检查，聚焦元素，填充它，并在填充后触发一个输入事件。请注意，您可以传递一个空字符串来清除输入字段。

如果目标元素不是 `\<input>`，`\<textarea>` or `[contenteditable]`，此方法将抛出一个错误。但是，如果该元素位于 `\<label>` 元素中，且该元素具有关联控件，则该控件将被填充。

To send fine-grained keyboard events, use `element_handle.type(text, **kwargs)`.

## `# element_handle.focus()` `#`

---

- returns: `<NoneType>` `#`

Calls `focus` on the element.

## `# element_handle.get_attribute(name)` `#`

---

- `name` `<str>` 属性名。`#`
- returns: `<NoneType|str>` `#`

返回元素属性值。

## `# element_handle.hover(**kwargs)` `#`

---

- `force` `<bool>` 是否绕过 `actionability` 检查。默认值为 `false`。`#`
- `modifiers` `<List["Alt"|"Control"|"Meta"|"Shift"]>` `modifiers` 按键要按。确保在操作期间只按下这些修饰符，然后恢复当前的修饰符。如果未指定，则使用当前按下的修饰符。`#`

- `position` `<Dict>` 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点.#
  - `x` `<float>`
  - `y` `<float>`
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改.#
- `trial` `<bool>` 设置后，该方法只执行 `actionability` 检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的.#
- returns: `<NoneType>` #

该方法通过执行以下步骤悬停在元素上：

1. 等待元素的 `可操作性` 检查，除非设置了强制选项。
2. 如果需要，将元素滚动到视图中。
3. 使鼠标停在元素中心或指定位置上。
4. 等待发起的导航成功或失败，除非设置了 `noWaitAfter` 选项。

如果元素在动作期间的任何时刻与DOM分离，此方法将抛出。

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 `TimeoutError`。传递零超时将禁用此功能。

## # `element_handle.inner_html()` #

---

- returns: `<str>` #

Returns the `element.innerHTML`.

## # `element_handle.inner_text()` #

---

- returns: `<str>` #

Returns the `element.innerText`.

## # element\_handle.input\_value(\*\*kwargs) #

---

- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。#
- returns: `<str>` #

返回输入 `\<input>` or `\<textarea>` or `\<select>` 元素的值, 排除非输入元素.

## # element\_handle.is\_checked() #

---

- returns: `<bool>` #

返回元素是否被选中。如果元素不是复选框或单选输入则排除。

## # element\_handle.is\_disabled() #

---

- returns: `<bool>` #

返回该元素是否被禁用，与启用 `enabled` 相反。

## # element\_handle.is\_editable() #

---

- returns: `<bool>` #

返回元素是否可编辑 `editable`。

## # element\_handle.is\_enabled() #

---

- returns: `<bool>` #

返回元素是否被启用 `enabled`。

## # element\_handle.is\_hidden()#

---

- returns: <[bool](#)>#

返回元素是否隐藏，与可见 [visible](#) 相反。

## # element\_handle.is\_visible()#

---

- returns: <[bool](#)>#

返回元素是否可见 [visible](#)。

## # element\_handle.owner\_frame()#

---

- returns: <[NoneType](#)|[Frame](#)>#

Returns the frame containing the given element.

## # element\_handle.press(key, \*\*kwargs)#

---

- **key** <[str](#)> 要按下的 **键名** 或要生成的字符，如 **ArrowLeft** 或 'a'.#
- **delay** <[float](#)> **keydown** 和 **keyup** 之间的等待时间，单位是毫秒。默认为0.#
- **no\_wait\_after** <[bool](#)> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 **false**.#
- **timeout** <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 **0** 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: <[NoneType](#)>#

聚焦元素，然后使用 [keyboard.down\(key\)](#) and [keyboard.up\(key\)](#)。

key可以指定想要的 [keyboardEvent.key](#) 或是单个字符生成的文本,这里可以找到键值的超集。键的例子如下:

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

还支持以下快捷键: Shift, Control, Alt, Meta, ShiftLeft.

按住 Shift 键将输入与大写键对应的文本.

如果 key 是单个字符,它是区分大小写的,因此值 a 和 A 将生成不同的文本.

也支持快捷键,如键:“Control+o”或键:“Control+Shift+T”。当用修饰符指定时,修饰符被按下并被保持,而随后的键被按下.

#

## element\_handle.query\_selector(selector)#

- selector <str> 要查询的选择器。有关更多细节,请参阅[working with selectors.#](#)
- returns: <NoneType|ElementHandle>#

The method finds an element matching the specified selector in the `ElementHandle`'s subtree. See [Working with selectors](#) for more details. If no elements match the selector, returns `null`.

#

## element\_handle.query\_selector\_all(selector)#

- selector <str> 要查询的选择器。有关更多细节,请参阅[working with selectors.#](#)
- returns: <List[ElementHandle]>#

The method finds all elements matching the specified selector in the `ElementHandle`'s subtree. See [Working with selectors](#) for more details. If no elements match the selector, returns empty array.

## # `element_handle.screenshot(**kwargs)` #

---

- `animations` <"disabled"> 当设置为 `"disabled"` 时, 停止CSS动画, CSS转换和Web动画。动画根据其持续时间得到不同的处理: #
  - 有限动画是快进到完成, 所以他们会触发 `transitionend` 事件。
  - 无限动画被取消到初始状态, 然后在屏幕截图后播放。
- `mask` <[List](#)[[Locator](#)]> 指定在截屏时应该被屏蔽的定位器。被屏蔽的元素将被一个粉红色的框覆盖#FF00FF, 完全覆盖该元素. #
- `omit_background` <[bool](#)> 隐藏默认的白色背景, 并允许透明捕捉屏幕截图。不适用于 `jpeg` 图像。默认值为 `false` . #
- `path` <[Union](#)[[str](#), [pathlib.Path](#)]> 保存镜像的文件路径, 屏幕截图类型将从文件扩展名推断。如果 `path` 是一个相对路径, 那么它是相对于当前工作目录解析的。如果没有提供路径, 映像将不会被保存到磁盘. #
- `quality` <[int](#)> 图像的质量, 在0-100之间。不适用于png图像. #
- `timeout` <[float](#)> 最大时间, 单位为毫秒, 默认为30秒, 通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改. #
- `type` <"png"|"jpeg">指定截图类型, 默认为 `png` . #
- returns: <[bytes](#)> #

返回带有捕获的截图的缓冲区。

这个方法等待[可操作性](#)检查, 然后在截屏之前将元素滚动到视图中。如果元素与DOM分离, 该方法将抛出一个错误。

## # `element_handle.scroll_into_view_if_needed(**kwargs)` #

---

- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。`#`
- returns: `<NoneType>` `#`

这个方法等待 [可操作性](#) 检查，然后尝试滚动元素到视图中，除非它是完全可见的，由 [IntersectionObserver](#) 的比率定义。

Throws when `elementHandle` does not point to an element [connected](#) to a Document or a ShadowRoot.

**#**

## element\_handle.select\_option(\*\*kwargs) `#`

- `force` `<bool>` 是否绕过 [actionability](#) 检查。默认值为 `false` `#`
- `no_wait_after` `<bool>` 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` `#`
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。`#`
- `element` `<ElementHandle|List[ElementHandle]>` 要选择的选项。可选的。`#`
- `index` `<int|List[int]>` 按索引进行选择的选项。可选的。`#`
- `value` `<str|List[str]>` 按值选择的选项。如果 `\<select>` 具有多个属性，则选择所有给定的选项，否则只选择与传递的选项之一匹配的的第一个选项。可选的。`#`
- `label` `<str|List[str]>` 按标签进行选择的选项。如果 `\<select>` 具有多个属性，则选择所有给定的选项，否则只选择与传递的选项之一匹配的的第一个选项。可选的。`#`
- returns: `<List[str]>` `#`

这个方法等待 [可操作性](#) 检查，直到所有指定的选项都出现在 `\<select>` 元素中，然后选择这些选项。

如果目标元素不是 `\<select>` 元素，此方法将抛出一个错误。但是，如果该元素位于 `\<label>` 元素中，且该元素具有关联控件，则将使用该控件。



Returns the array of option values that have been successfully selected.

一旦选择了所有提供的选项，就触发一个更改和输入事件。

- Sync

```
# single selection matching the value
handle.select_option("blue")
# single selection matching both the label
handle.select_option(label="blue")
# multiple selection
handle.select_option(value=["red", "green", "blue"])
```

- Async

```
# single selection matching the value
await handle.select_option("blue")
# single selection matching the label
await handle.select_option(label="blue")
# multiple selection
await handle.select_option(value=["red", "green", "blue"])
```

```
# sync

# single selection matching the value
handle.select_option("blue")
# single selection matching both the value and the label
handle.select_option(label="blue")
# multiple selection
handle.select_option("red", "green", "blue")
# multiple selection for blue, red and second option
handle.select_option(value="blue", { index: 2 }, "red")
```

## # element\_handle.select\_text(\*\*kwargs)#

- `force` <bool> 是否绕过 [actionability](#) 检查。默认值为 `false` .#
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用

[browser\\_context.set\\_default\\_timeout\(timeout\)](#) or  
[page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#

- returns: [<NoneType>](#) #

这个方法等待 [可操作性](#) 检查, then focuses the element and selects all its text content.

## # element\_handle.set\_checked(checked, \*\*kwargs) #

---

- `checked` [<bool>](#) 是否选中或不选中复选框.#
- `force` [<bool>](#) 是否绕过 [actionability](#) 检查。默认值为 `false` .#
- `no_wait_after` [<bool>](#) 启动导航的操作正在等待这些导航发生, 并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项, 比如导航到不可访问的页面。默认值为 `false` .#
- `position` [<Dict>](#) 相对于元素填充框的左上角使用的一个点。如果没有指定, 则使用元素的某个可见点.#
  - `x` [<float>](#)
  - `y` [<float>](#)
- `timeout` [<float>](#) 最大时间, 单位为毫秒, 默认为30秒, 通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- `trial` [<bool>](#) 设置后, 该方法只执行 [actionability](#) 检查, 并跳过操作。默认值为 `false` 。在元素准备好时再执行动作是很有用的.#
- returns: [<NoneType>](#) #

这个方法通过执行以下步骤检查或取消检查一个元素:

1. Ensure that element is a checkbox or a radio input. If not, this method throws.
2. 如果元素已经具有正确的选中状态, 则该方法立即返回.
3. 等待匹配元素的 [actionability](#) 检查, 除非设置了强制选项。如果在检查期间分离了元素, 则会重试整个操作.
4. 如果需要, 将元素滚动到视图中.
5. 使用 [page.mouse](#) 单击元素的中心.
6. 等待已启动的导航成功或失败, 除非设置了 `no_wait_after` 选项.

7. 确保元素现在被选中或取消选中。如果不是，则抛出此方法。

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

## # element\_handle.set\_input\_files(files, \*\*kwargs) #

---

- `files` <[Union](#)[[str](#), [pathlib.Path](#)]|[List](#)[[Union](#)[[str](#), [pathlib.Path](#)]]|[Dict](#)|[List](#)[[Dict](#)]> #
  - `name` <[str](#)> File name
  - `mimeType` <[str](#)> File type
  - `buffer` <[bytes](#)> File content
- `no_wait_after` <[bool](#)> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- `timeout` <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: <[NoneType](#)> #

This method expects `elementHandle` to point to an [input element](#).

将文件输入的值设置为这些文件路径或文件。如果某些 `filepath` 是相对路径，那么它们将相对于当前工作目录进行解析。对于空数组，清除选定的文件。

## # element\_handle.tap(\*\*kwargs) #

---

- `force` <[bool](#)> 是否绕过 [actionability](#) 检查。默认值为 `false` .#
- `modifiers` <[List](#)["Alt"|"Control"|"Meta"|"Shift"]> modifiers 按键要按。确保在操作期间只按下这些修饰符，然后恢复当前的修饰符。如果未指定，则使用当前按下的修饰符.#

- `no_wait_after` `<bool>` 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false`。#
- `position` `<Dict>` 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点。#
  - `x` `<float>`
  - `y` `<float>`
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。#
- `trial` `<bool>` 设置后，该方法只执行 `actionability` 检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的。#
- returns: `<NoneType>` #

这个方法通过执行以下步骤点击元素：

1. 等待元素的 `可操作性` 检查，除非设置了强制选项。
2. 如果需要，将元素滚动到视图中。
3. 点击页面中心或指定位置。
4. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项。

如果元素在动作期间的任何时刻与DOM分离，此方法将抛出。

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 `TimeoutError`。传递零超时将禁用此功能。

#### NOTE

`elementHandle.tap()` requires that the `hasTouch` option of the browser context be set to true.

## # `element_handle.text_content()` #

- returns: `<NoneType|str>` #

Returns the `node.textContent`.

## # `element_handle.type(text, **kwargs)` #

- `text` <[str](#)> 要输入到焦点元素中的文本. #
- `delay` <[float](#)> 按键之间的等待时间，单位是毫秒。默认为0. #
- `no_wait_after` <[bool](#)> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false`. #
- `timeout` <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改. #
- returns: <[NoneType](#)> #

聚焦元素，然后为文本中的每个字符发送 `keydown`，`keypress` / `input`，and `keyup` 事件。

To press a special key, like `Control` or `ArrowDown`, use [element\\_handle.press\(key, \\*\\*kwargs\)](#).

- Sync

```
element_handle.type("hello") # types instantly
element_handle.type("world", delay=100) # types slower, like a
user
```

- Async

```
await element_handle.type("hello") # types instantly
await element_handle.type("world", delay=100) # types slower,
like a user
```

An example of typing into a text field and then submitting the form:

- Sync

```
element_handle = page.query_selector("input")
element_handle.type("some text")
element_handle.press("Enter")
```

- Async

```
element_handle = await page.query_selector("input")
await element_handle.type("some text")
await element_handle.press("Enter")
```

## # `element_handle.uncheck(**kwargs)` #

- `force` <bool> 是否绕过 [actionability](#) 检查。默认值为 `false` .#
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- `position` <Dict> 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点.#
  - `x` <float>
  - `y` <float>
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- `trial` <bool> 设置后，该方法只执行 [actionability](#) 检查，并跳过操作。默认值为 `false` 。在元素准备好时再执行动作是很有用的.#
- `returns`: <NoneType> .#

这个方法通过执行以下步骤来检查元素：

1. 确保元素是一个复选框或单选输入。如果不是，则抛出此方法。如果元素已被选中，则此方法立即返回。
2. 等待元素的 [可操作性](#) 检查，除非设置了强制选项。
3. 如果需要，将元素滚动到视图中。
4. 使用 [page.mouse](#) 单击元素的中心。
5. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项。
6. 确保元素现在是未选中的。如果不是，则排除此方法。

如果元素在动作期间的任何时刻与DOM分离，此方法将抛出。

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#) 。传递零超时将禁用此功能。

#

## element\_handle.wait\_for\_element\_state(state, \*\*kwargs)#

---

- `state` <"visible"|"hidden"|"stable"|"enabled"|"disabled"|"editable"> A state to wait for, see below for more details.#
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: <NoneType>#

Returns when the element satisfies the `state`.

Depending on the `state` parameter, this method waits for one of the [actionability](#) checks to pass. This method throws when the element is detached while waiting, unless waiting for the `"hidden"` state.

- `"visible"` Wait until the element is [visible](#).
- `"hidden"` Wait until the element is [not visible](#) or [not attached](#).  
Note that waiting for hidden does not throw when the element detaches.
- `"stable"` Wait until the element is both [visible](#) and [stable](#).
- `"enabled"` Wait until the element is [enabled](#).
- `"disabled"` Wait until the element is [not enabled](#).
- `"editable"` Wait until the element is [editable](#).

If the element does not satisfy the condition for the `timeout` milliseconds, this method will throw.

#

## element\_handle.wait\_for\_selector(selector, \*\*kwargs)#

---

- `selector` <str> 要查询的选择器。有关更多细节，请参阅[working with selectors](#).#

- `state` <"attached"|"detached"|"visible"|"hidden"> 默认为 `"visible"`。可以是: #
  - `'attached'` - 等待元素出现在DOM中。
  - `'detached'` - 等待元素在DOM中不存在。
  - `'visible'` - 等待元素有非空的边界框 且 没有 `visibility:hidden`。注意，没有任何内容或带有 `display:none` 的元素有一个空的边界框，因此不被认为是可见的。
  - `'hidden'` - 等待元素从DOM中分离出来, 或有一个空的边界框或 `visibility:hidden`。这与 `"visible"` 选项相反。
- `strict` <`bool`> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。#
- `timeout` <`float`> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。#
- returns: <`NoneType`|`ElementHandle`> #

Returns element specified by selector when it satisfies `state` option.  
Returns `null` if waiting for `hidden` or `detached`.

Wait for the `selector` relative to the element handle to satisfy `state` option (either appear/disappear from dom, or become visible/hidden). If at the moment of calling the method `selector` already satisfies the condition, the method will return immediately. If the selector doesn't satisfy the condition for the `timeout` milliseconds, the function will throw.

- Sync

```
page.set_content("<div><span></span></div>")
div = page.query_selector("div")
# waiting for the "span" selector relative to the div.
span = div.wait_for_selector("span", state="attached")
```

- Async

```
await page.set_content("<div><span></span></div>")
div = await page.query_selector("div")
# waiting for the "span" selector relative to the div.
span = await div.wait_for_selector("span", state="attached")
```



## NOTE

*This method does not work across navigations, use [page.wait\\_for\\_selector\(selector, \\*\\*kwargs\)](#) instead.*

## Error

- extends: [Exception](#)

当某些操作异常终止时将引发错误，例如浏览器在运行时关闭  
[page.evaluate\(expression, \\*\\*kwargs\)](#)，所有playwright的Exception都继承自这个类

- [error.message](#)
- [error.name](#)
- [error.stack](#)

## # error.message #

---

- type: [<str>](#)

错误消息。

## # error.name #

---

- type: [<str>](#)

浏览器内部抛出的错误的名称。可选的。

## # error.stack #

---

- type: [<str>](#)

浏览器内部抛出的错误堆栈。可选的。

## FileChooser

[FileChooser](#) 对象由 [page.on\("filechooser"\)](#) 事件中的页面调度。

- Sync

```
with page.expect_file_chooser() as fc_info:
    page.click("upload")
file_chooser = fc_info.value
file_chooser.set_files("myfile.pdf")
```

- Async

```
async with page.expect_file_chooser() as fc_info:
    await page.click("upload")
file_chooser = await fc_info.value
await file_chooser.set_files("myfile.pdf")
```

- [file\\_chooser.element](#)
- [file\\_chooser.is\\_multiple\(\)](#)
- [file\\_chooser.page](#)
- [file\\_chooser.set\\_files\(files, \\*\\*kwargs\)](#)

## # file\_chooser.element#

---

- returns: <[ElementHandle](#)>#

返回与此文件选择器关联的输入元素。

## # file\_chooser.is\_multiple()#

---

- returns: <[bool](#)>#

返回此文件选择器是否接受多个文件。

## # `file_chooser.page` #

---

- returns: `<Page>` #

返回此文件选择器所属的页面。

## # `file_chooser.set_files(files, **kwargs)` #

---

- `files` `<Union[str, pathlib.Path]|List[Union[str, pathlib.Path]]|Dict|List[Dict]>` #
  - `name` `<str>` 文件名
  - `mimeType` `<str>` 文件类型
  - `buffer` `<bytes>` 文件内容
- `no_wait_after` `<bool>` 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改.#
- returns: `<NoneType>` #

设置与此选择器关联的文件输入的值。如果某些 `filePaths` 是相对路径，那么它们将相对于当前工作目录进行解析。对于空数组，清除选定的文件。

### Frame

在每个时间点，页面都会通过 `page.main_frame` and `frame.child_frames` 方法。

`Frame` 对象的生命周期由三个事件控制，在页面对象中分配：

- [page.on\("frameattached"\)](#) - 当框架被附加到页面时触发. 一个框架只能附加到页面一次.
- [page.on\("framenavigated"\)](#) - 当帧提交导航到另一个URL时触发
- [page.on\("framedetached"\)](#) - 当框架从页面分离时触发. **Frame**只能从页面分离一次.

一个倾倒框架树的例子:

- Sync

```
from playwright.sync_api import sync_playwright

def run(playwright):
    firefox = playwright.firefox
    browser = firefox.launch()
    page = browser.new_page()
    page.goto("https://www.theverge.com")
    dump_frame_tree(page.main_frame, "")
    browser.close()

def dump_frame_tree(frame, indent):
    print(indent + frame.name + '@' + frame.url)
    for child in frame.child_frames:
        dump_frame_tree(child, indent + "  ")

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

- Async

```
import asyncio
from playwright.async_api import async_playwright

async def run(playwright):
    firefox = playwright.firefox
    browser = await firefox.launch()
    page = await browser.new_page()
    await page.goto("https://www.theverge.com")
    dump_frame_tree(page.main_frame, "")
    await browser.close()

def dump_frame_tree(frame, indent):
```

```

print(indent + frame.name + '@' + frame.url)
for child in frame.child_frames:
    dump_frame_tree(child, indent + "    ")

async def main():
    async with async_playwright() as playwright:
        await run(playwright)
asyncio.run(main())

```

- [frame.add\\_script\\_tag\(\\*\\*kwargs\)](#)
- [frame.add\\_style\\_tag\(\\*\\*kwargs\)](#)
- [frame.check\(selector, \\*\\*kwargs\)](#)
- [frame.child\\_frames](#)
- [frame.click\(selector, \\*\\*kwargs\)](#)
- [frame.content\(\)](#)
- [frame.dblclick\(selector, \\*\\*kwargs\)](#)
- [frame.dispatch\\_event\(selector, type, \\*\\*kwargs\)](#)
- [frame.drag\\_and\\_drop\(source, target, \\*\\*kwargs\)](#)
- [frame.eval\\_on\\_selector\(selector, expression, \\*\\*kwargs\)](#)
- [frame.eval\\_on\\_selector\\_all\(selector, expression, \\*\\*kwargs\)](#)
- [frame.evaluate\(expression, \\*\\*kwargs\)](#)
- [frame.evaluate\\_handle\(expression, \\*\\*kwargs\)](#)
- [frame.expect\\_navigation\(\\*\\*kwargs\)](#)
- [frame.fill\(selector, value, \\*\\*kwargs\)](#)
- [frame.focus\(selector, \\*\\*kwargs\)](#)
- [frame.frame\\_element\(\)](#)
- [frame.frame\\_locator\(selector\)](#)
- [frame.get\\_attribute\(selector, name, \\*\\*kwargs\)](#)
- [frame.goto\(url, \\*\\*kwargs\)](#)
- [frame.hover\(selector, \\*\\*kwargs\)](#)
- [frame.inner\\_html\(selector, \\*\\*kwargs\)](#)
- [frame.inner\\_text\(selector, \\*\\*kwargs\)](#)
- [frame.input\\_value\(selector, \\*\\*kwargs\)](#)
- [frame.is\\_checked\(selector, \\*\\*kwargs\)](#)
- [frame.is\\_detached\(\)](#)
- [frame.is\\_disabled\(selector, \\*\\*kwargs\)](#)
- [frame.is\\_editable\(selector, \\*\\*kwargs\)](#)
- [frame.is\\_enabled\(selector, \\*\\*kwargs\)](#)
- [frame.is\\_hidden\(selector, \\*\\*kwargs\)](#)
- [frame.is\\_visible\(selector, \\*\\*kwargs\)](#)
- [frame.locator\(selector, \\*\\*kwargs\)](#)
- [frame.name](#)

- [frame.page](#)
- [frame.parent\\_frame](#)
- [frame.press\(selector, key, \\*\\*kwargs\)](#)
- [frame.query\\_selector\(selector, \\*\\*kwargs\)](#)
- [frame.query\\_selector\\_all\(selector\)](#)
- [frame.select\\_option\(selector, \\*\\*kwargs\)](#)
- [frame.set\\_checked\(selector, checked, \\*\\*kwargs\)](#)
- [frame.set\\_content\(html, \\*\\*kwargs\)](#)
- [frame.set\\_input\\_files\(selector, files, \\*\\*kwargs\)](#)
- [frame.tap\(selector, \\*\\*kwargs\)](#)
- [frame.text\\_content\(selector, \\*\\*kwargs\)](#)
- [frame.title\(\)](#)
- [frame.type\(selector, text, \\*\\*kwargs\)](#)
- [frame.uncheck\(selector, \\*\\*kwargs\)](#)
- [frame.url](#)
- [frame.wait\\_for\\_function\(expression, \\*\\*kwargs\)](#)
- [frame.wait\\_for\\_load\\_state\(\\*\\*kwargs\)](#)
- [frame.wait\\_for\\_selector\(selector, \\*\\*kwargs\)](#)
- [frame.wait\\_for\\_timeout\(timeout\)](#)
- [frame.wait\\_for\\_url\(url, \\*\\*kwargs\)](#)

## # [frame.add\\_script\\_tag\(\\*\\*kwargs\)](#) #

---

- `content` <[str](#)> 要注入帧的原始JavaScript内容。<#>
- `path` <[Union](#)[[str](#), [pathlib.Path](#)]> 要注入帧的JavaScript文件的路径, 如果 `path` 是一个相对路径, 那么它是相对于当前工作目录解析的。<#>
- `type` <[str](#)> 脚本类型。使用“module”来加载一个Javascript ES6模块。详情请参阅 [script](#)。<#>
- `url` <[str](#)> 要添加的脚本的url。<#>
- `returns:` <[ElementHandle](#)> <#>

当脚本的onload触发或脚本内容被注入帧时, 返回添加的标签。

添加一个 `\<script>` 标签到页面所需的url或内容。

## # [frame.add\\_style\\_tag\(\\*\\*kwargs\)](#) #

---

- `content` <[str](#)> 原始的CSS内容注入到帧。<#>

- `path` <[Union\[str, pathlib.Path\]](#)> 要注入帧的CSS文件的路径, 如果 `path` 是一个相对路径, 那么它是相对于当前工作目录解析的。<#>
- `url` <[str](#)> `\<link>` 标签的url。<#>
- `returns:` <[ElementHandle](#)> <#>

当样式表的onload触发时, 或者当CSS内容被注入框架时, 返回添加的标签。

添加一个 `\<link rel="stylesheet">` 标签到页面所需的url或 `\<style type="text/css">` 标签的内容。

## # `frame.check(selector, **kwargs)` <#>

- `selector` <[str](#)> 一个搜索元素的选择器。如果有多个元素满足选择器, 将使用第一个元素。有关更多细节, 请参阅[working with selectors](#)。<#>
- `force` <[bool](#)> 是否绕过[actionability](#)检查。默认值为 `false`。<#>
- `no_wait_after` <[bool](#)> 启动导航的操作正在等待这些导航发生, 并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项, 比如导航到不可访问的页面。默认值为 `false`。<#>
- `position` <[Dict](#)> 相对于元素填充框的左上角使用的一个点。如果没有指定, 则使用元素的某个可见点。<#>
  - `x` <[float](#)>
  - `y` <[float](#)>
- `strict` <[bool](#)> 当为true时, 调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素, 调用将抛出一个异常。<#>
- `timeout` <[float](#)> 最大时间, 单位为毫秒, 默认为30秒, 通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- `trial` <[bool](#)> 设置后, 该方法只执行[actionability](#)检查, 并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的。<#>
- `returns:` <[NoneType](#)> <#>

这个方法通过以下步骤检查元素匹配选择器:

1. 找到一个元素匹配选择器。如果没有, 则等待直到匹配的元素被附加到DOM.

2. 确保匹配的元素是一个复选框或单选输入。如果不是，则排除此方法。如果元素已被选中，则该方法立即返回。
3. 等待匹配元素的 [actionability](#) 检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作。
4. 如果需要，将元素滚动到视图中。
5. 使用 [page.mouse](#) 单击元素的中心。
6. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项。
7. 确保元素现在被选中。如果不是，则排除此方法。

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

## # frame.child\_frames#

---

- returns: [List\[Frame\]](#)>#

## # frame.click(selector, \*\*kwargs)#

---

- `selector` [<str>](#) 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅 [working with selectors](#) .#
- `button` [<"left"|"right"|"middle">](#) 默认左 `left` .#
- `click_count` [<int>](#) 默认为1. 查看 [UIEvent.detail](#) .#
- `delay` [<float>](#) `mousedown` 和 `mouseup` 之间的等待时间，单位是毫秒。默认为0.#
- `force` [<bool>](#) 是否绕过 [actionability](#) 检查。默认值为 `false` .#
- `modifiers` [<List\["Alt"|"Control"|"Meta"|"Shift"\]>](#) modifiers 按键要按。确保在操作期间只按下这些修饰符，然后恢复当前的修饰符。如果未指定，则使用当前按下的修饰符.#
- `no_wait_after` [<bool>](#) 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- `position` [<Dict>](#) 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点.#
  - `x` [<float>](#)



- `y` `<float>`
- `strict` `<bool>` 当为`true`时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。<#>
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- `trial` `<bool>` 设置后，该方法只执行 [actionability](#) 检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的。<#>
- returns: `<NoneType>` <#>

这个方法通过执行以下步骤点击元素匹配选择器：

1. 找到一个元素匹配选择器。如果没有，则等待直到匹配的元素被附加到DOM.
2. 等待匹配元素的 [actionability](#) 检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作.
3. 如果需要，将元素滚动到视图中.
4. 使用 [page.mouse](#) 单击元素的中心, or the specified `position`.
5. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项.

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

## # `frame.content()` <#>

---

- returns: `<str>` <#>

获取框架的完整HTML内容，包括文档类型。

## # `frame.dblclick(selector, **kwargs)` <#>

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅 [working with selectors](#) <#>
- `button` `<"left"|"right"|"middle">` 默认左 `left` <#>

- `delay` `<float>` `mousedown` 和 `mouseup` 之间的等待时间，单位是毫秒。默认为0.#
- `force` `<bool>` 是否绕过 `actionability` 检查。默认值为 `false`.#
- `modifiers` `<List["Alt"|"Control"|"Meta"|"Shift"]>` `modifiers` 按键要按。确保在操作期间只按下这些修饰符，然后恢复当前的修饰符。如果未指定，则使用当前按下的修饰符.#
- `no_wait_after` `<bool>` 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false`.#
- `position` `<Dict>` 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点.#
  - `x` `<float>`
  - `y` `<float>`
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改.#
- `trial` `<bool>` 设置后，该方法只执行 `actionability` 检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的.#
- `returns:` `<NoneType>` #

该方法通过执行以下步骤双击元素匹配选择器：

1. 找到一个元素匹配选择器。如果没有，则等待直到匹配的元素被附加到DOM.
2. 等待匹配元素的 `actionability` 检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作.
3. 如果需要，将元素滚动到视图中.
4. 使用 `page.mouse` 方法,双击元素中心位置.
5. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项.该方法执行以下步骤双击元素,注意，如果 `dblclick()` 的第一次单击触发了一个导航事件，则该方法将抛出.

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 `TimeoutError`。传递零超时将禁用此功能。

`frame.dblclick()` 分发两个 `click` 事件和一个 `dblclick` 事件。

## # `frame.dispatch_event(selector, type, **kwargs)` #

---

- `selector` <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#)。#
- `type` <str> DOM事件类型: `"click"`，`"dragstart"` 等。#
- `event_init` <EvaluationArgument> 可选的特定于事件的初始化属性。#
- `strict` <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。#
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用[browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。#
- returns: <NoneType> #

下面的代码片段分派元素上的 `单击` 事件。无论元素的可见性状态如何，单击都将被分派。这相当于调用 [element.click\(\)](#)。

- Sync

```
frame.dispatch_event("button#submit", "click")
```

- Async

```
await frame.dispatch_event("button#submit", "click")
```

在底层，它根据给定的类型创建一个事件实例，使用 `event_init` 属性初始化它，并在元素上分派它。默认情况下，事件是组合的、可取消的和冒泡的。

由于 `event_init` 是特定于事件的，请参考事件文档中的初始属性列表：

- [DragEvent](#)
- [FocusEvent](#)
- [KeyboardEvent](#)
- [MouseEvent](#)

- [PointerEvent](#)
- [TouchEvent](#)
- [Event](#)

如果你想要将活动对象传递到事件中，你也可以指定 `jhandle` 作为属性值：

- Sync

```
# note you can only create data_transfer in chromium and
firefox
data_transfer = frame.evaluate_handle("new DataTransfer()")
frame.dispatch_event("#source", "dragstart", { "dataTransfer":
data_transfer })
```

- Async

```
# note you can only create data_transfer in chromium and
firefox
data_transfer = await frame.evaluate_handle("new
DataTransfer()")
await frame.dispatch_event("#source", "dragstart", {
"dataTransfer": data_transfer })
```

## # frame.drag\_and\_drop(source, target, \*\*kwargs) #

---

- `source` <str> #
- `target` <str> #
- `force` <bool> 是否绕过 [actionability](#) 检查。默认值为 `false` .#
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- `source_position` <Dict> 此时相对于元素填充框的左上角单击源元素。如果没有指定，则使用元素的某个可见点.#
  - `x` <float>
  - `y` <float>

- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。<#>
- `target_position` `<Dict>` 此时相对于元素填充框的左上角落在目标元素上。如果没有指定，则使用元素的某个可见点。<#>
  - `x` `<float>`
  - `y` `<float>`
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- `trial` `<bool>` 设置后，该方法只执行 [actionability](#) 检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的。<#>
- returns: `<NoneType>` <#>

## # `frame.eval_on_selector(selector, expression, **kwargs)` <#>

---

- `selector` `<str>` 要查询的选择器。有关更多细节，请参阅[working with selectors](#)。<#>
- `expression` `<str>` 要在浏览器上下文中计算的 `JavaScript` 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值。<#>
- `arg` `<EvaluationArgument>` 传递给 `expression` 的可选参数。<#>
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。<#>
- returns: `<Serializable>` <#>

返回表达式的返回值。

### CAUTION

此方法不等待元素通过可操作性检查，因此可能导致不稳定的测试。使用 [locator.evaluate\(expression, \\*\\*kwargs\)](#)，其它 [Locator](#) 方法优先断言

该方法在框架中找到与指定选择器匹配的元素，并将其作为第一个参数传递给表达式。有关详细信息，请参阅 [Working with selectors](#)。如果没有匹配该选择器的元素，该方法将抛出错误

如果 `expression` 返回 [Promise](#), 那么 `frame.eval_on_selector(selector, expression, **kwargs)` 将等待promise解析并返回它的值。

例子:

- Sync

```
search_value = frame.eval_on_selector("#search", "el =>
el.value")
preload_href = frame.eval_on_selector("link[rel=preload]", "el
=> el.href")
html = frame.eval_on_selector(".main-container", "(e, suffix)
=> e.outerHTML + suffix", "hello")
```

- Async

```
search_value = await frame.eval_on_selector("#search", "el =>
el.value")
preload_href = await
frame.eval_on_selector("link[rel=preload]", "el => el.href")
html = await frame.eval_on_selector(".main-container", "(e,
suffix) => e.outerHTML + suffix", "hello")
```

## # `frame.eval_on_selector_all(selector, expression, **kwargs)` #

---

- `selector` <[str](#)> 要查询的选择器。有关更多细节, 请参阅[working with selectors](#).#
- `expression` <[str](#)> 要在浏览器上下文中计算的 `JavaScript` 表达式。如果它看起来像一个函数声明, 它会被解释为一个函数。否则, 作为表达式求值.#
- `arg` <[EvaluationArgument](#)> 传递给 `expression` 的可选参数.#
- returns: <[Serializable](#)>.#

返回表达式的返回值。

### NOTE

在大多数情况下, [locator.evaluate\\_all\(expression, \\*\\*kwargs\)](#), 其它 [Locator](#) 做得更好。

该方法在框架中查找与指定选择器匹配的所有元素，并将匹配元素的数组作为第一个参数传递给表达式。有关详细信息，请参阅[Working with selectors](#)

如果 `expression` 返回 [Promise](#)，那么 [frame.eval\\_on\\_selector\(selector, expression, \\*\\*kwargs\)](#) 将等待promise解析并返回它的值。

例子：

- Sync

```
divs_counts = frame.eval_on_selector_all("div", "(divs, min)
⇒ divs.length ≥ min", 10)
```

- Async

```
divs_counts = await frame.eval_on_selector_all("div", "(divs,
min) ⇒ divs.length ≥ min", 10)
```

## # [frame.evaluate\(expression, \\*\\*kwargs\)](#) #

- `expression` <[str](#)> 要在浏览器上下文中计算的 [JavaScript](#) 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值。<#>
- `arg` <[EvaluationArgument](#)> 传递给 `expression` 的可选参数。<#>
- returns: <[Serializable](#)> <#>

返回表达式的返回值。

如果函数传递给 [frame.evaluate\(expression, \\*\\*kwargs\)](#) 返回 [Promise](#)，那么 [frame.evaluate\(expression, \\*\\*kwargs\)](#) 将等待promise解析并返回其值。

如果函数传递给 [frame.evaluate\(expression, \\*\\*kwargs\)](#) 返回一个不可序列化的值-[Serializable](#)，那么 [frame.evaluate\(expression, \\*\\*kwargs\)](#) 返回 `undefined`。  
Playwright 还支持传递一些附加值，不能通过 [JSON](#) 序列化：`-0`，`NaN`，`Infinity`，`-Infinity`。

- Sync

```
result = frame.evaluate("([x, y]) => Promise.resolve(x * y)",  
[7, 8])  
print(result) # prints "56"
```

- Async

```
result = await frame.evaluate("([x, y]) => Promise.resolve(x *  
y)", [7, 8])  
print(result) # prints "56"
```

字符串也可以代替函数传入.

- Sync

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

- Async

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

可以将 [ElementHandle](#) 实例作为参数传递给框架 [frame.evaluate\(expression, \\*\\*kwargs\)](#):

- Sync

```
body_handle = frame.evaluate("document.body")  
html = frame.evaluate("([body, suffix]) => body.innerHTML +  
suffix", [body_handle, "hello"])  
body_handle.dispose()
```

- Async



```
body_handle = await frame.evaluate("document.body")
html = await frame.evaluate("([body, suffix]) =>
body.innerHTML + suffix", [body_handle, "hello"])
await body_handle.dispose()
```

## # frame.evaluate\_handle(expression, \*\*kwargs) #

---

- **expression** <str> 要在浏览器上下文中计算的 **JavaScript** 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值。#
- **arg** <EvaluationArgument> 传递给 **expression** 的可选参数。#
- returns: <JSHandle> #

返回表达式的返回值是 JSHandle。

框架之间唯一的区别 frame.evaluate(expression, \*\*kwargs) 和 frame.evaluate\_handle(expression, \*\*kwargs) 是 frame.evaluate\_handle(expression, \*\*kwargs) 返回 JSHandle。

如果函数，传递给 frame.evaluate\_handle(expression, \*\*kwargs)，返回 Promise，那么 frame.evaluate\_handle(expression, \*\*kwargs) 将等待promise解析并返回它的。

- Sync

```
a_window_handle =
frame.evaluate_handle("Promise.resolve(window)")
a_window_handle # handle for the window object.
```

- Async

```
a_window_handle = await
frame.evaluate_handle("Promise.resolve(window)")
a_window_handle # handle for the window object.
```

字符串也可以代替函数传入。

- Sync

```
a_handle = page.evaluate_handle("document") # handle for the
"document"
```

- Async

```
a_handle = await page.evaluate_handle("document") # handle for
the "document"
```

[JSHandle](#) 实例可以作为参数传递给 [frame.evaluate\\_handle\(expression, \\*\\*kwargs\)](#):

- Sync

```
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()
```

- Async

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

## # frame.expect\_navigation(\*\*kwargs)#

- **timeout** <float> 最大操作时间，单位为毫秒，默认为30秒，通过0表示禁止超时。默认值可以通过使用 [browser\\_context.set\\_default\\_navigation\\_timeout\(timeout\)](#), [browser\\_context.set\\_default\\_timeout\(timeout\)](#), [page.set\\_default\\_navigation\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来修改.#
- **url** <str|Pattern|Callable[URL]:bool> 一个glob模式、regex模式或谓词，在等待导航时接收匹配的url。注意，如果参数是一个不带通配符的字符串，该方法将等待导航到与该字符串完全相等的URL.#

- `wait_until` `<"load"|"domcontentloaded"|"networkidle"|"commit">`  
当认为操作成功时，默认为 `load`。事件可以是：<#>
  - `'domcontentloaded'` - 当 `domcontentloaded` 事件被触发时，认为操作已经完成。
  - `'load'` - 当触发 `load` 事件时，认为操作已经完成。
  - `'networkidle'` - 当至少 `500毫秒` 没有网络连接时，认为操作已经完成。
  - `'commit'` - 当接收到网络响应并开始加载文档时，认为操作已经完成。
- returns: `<EventManager[Response]>#`

等待框架导航并返回主资源响应。在多个重定向的情况下，导航将使用最后一个重定向的响应进行解析。如果导航到一个不同的锚或导航由于历史API的使用，导航将解析为 `null`。

该方法等待 `frame` 导航到一个新的URL。当你运行会间接导致框架导航的代码时，它很有用。考虑一下这个例子：

- Sync

```
with frame.expect_navigation():
    frame.click("a.delayed-navigation") # clicking the link
will indirectly cause a navigation
# Resolves after navigation has finished
```

- Async

```
async with frame.expect_navigation():
    await frame.click("a.delayed-navigation") # clicking the
link will indirectly cause a navigation
# Resolves after navigation has finished
```

#### NOTE

使用 [History API](#) 更改URL被视为导航。

**`# frame.fill(selector, value, **kwargs)#`**

- `selector` <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#).#
- `value` <str> Value to fill for the `\<input>`, `\<textarea>` or `[contenteditable]` element.#
- `force` <bool> 是否绕过[actionability](#)检查。默认值为 `false`.#
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false`.#
- `strict` <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: <NoneType>.#

这个方法等待元素匹配 `选择器`，等待[actionability](#)检查，聚焦元素，填充它，并在填充后触发一个输入事件。请注意，您可以 `传递` 一个空字符串来清除输入字段

如果目标元素不是 `\<input>`，`\<textarea>` 或者 `[contenteditable]` 元素，此方法将抛出一个错误。但是，如果该元素位于 `\<label>` 元素中，且该元素具有关联控件，则该控件将被填充。

要发送细粒度的键盘事件，请使用 [frame.type\(selector, text, \\*\\*kwargs\)](#).

## # `frame.focus(selector, **kwargs)`#

---

- `selector` <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#).#
- `strict` <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: <NoneType>.#

这个方法获取一个带有选择器的元素并聚焦于它。如果没有元素匹配选择器，该方法将等待，直到匹配元素出现在DOM中

## # `frame.frame_element()` #

---

- returns: `<ElementHandle>` #

返回与此frame相对应的frame或iframe元素句柄

这与 `element_handle.content_frame()` 相反. 注意，返回的句柄实际上属于父frame.

如果 frame 在 `frameElement()` 返回之前被分离，则此方法将抛出错误。

- Sync

```
frame_element = frame.frame_element()
content_frame = frame_element.content_frame()
assert frame == content_frame
```

- Async

```
frame_element = await frame.frame_element()
content_frame = await frame_element.content_frame()
assert frame == content_frame
```

## # `frame.frame_locator(selector)` #

---

- `selector` `<str>` 解析DOM元素时使用的选择器。有关更多细节，请参阅 [working with selectors](#) .#
- returns: `<FrameLocator>` #

在使用iframes时，您可以创建一个 frame 定位器，该定位器将进入iframe并允许选择该iframe中的元素。下面的代码片段在id为 `my-frame` 的iframe中定位到文本为"Submit"的元素，例如: `\<iframe id="my-frame">`:

- Sync

```
locator = frame.frame_locator("#my-  
iframe").locator("text=Submit")  
locator.click()
```

- Async

```
locator = frame.frame_locator("#my-  
iframe").locator("text=Submit")  
await locator.click()
```

## # frame.get\_attribute(selector, name, \*\*kwargs) #

---

- **selector** <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#). #
- **name** <str> 属性名. #
- **strict** <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常. #
- **timeout** <float> 最大时间，单位为毫秒，默认为30秒，通过 0 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改. #
- returns: <NoneType|str> #

返回元素属性值.

## # frame.goto(url, \*\*kwargs) #

---

- **url** <str> frame导航到的url。url应该包括协议，例如 [https://](#). #
- **referer** <str> referer头值。如果提供，它将优先于 [page.set\\_extra\\_http\\_headers\(headers\)](#) 设置的referer头值. #
- **timeout** <float> 最大操作时间，单位为毫秒，默认为30秒，通过0表示禁止超时。默认值可以通过使用 [browser\\_context.set\\_default\\_navigation\\_timeout\(timeout\)](#), [browser\\_context.set\\_default\\_timeout\(timeout\)](#),

[page.set\\_default\\_navigation\\_timeout\(timeout\)](#) or  
[page.set\\_default\\_timeout\(timeout\)](#) 方法来修改。<#>

- `wait_until` `<"load"|"domcontentloaded"|"networkidle"|"commit">`  
当认为操作成功时，默认为 `load`。事件可以是：<#>
  - `'domcontentloaded'` - 当 `domcontentloaded` 事件被触发时，认为操作已经完成。
  - `'load'` - 当触发 `load` 事件时，认为操作已经完成。
  - `'networkidle'` - 当至少 `500毫秒` 没有网络连接时，认为操作已经完成。
  - `'commit'` - 当接收到网络响应并开始加载文档时，认为操作已经完成。
- `returns:` `<NoneType | Response>`<#>

返回主资源响应。在多个重定向的情况下，导航将使用最后一个重定向的响应进行解析

如果以下情况，该方法将抛出一个错误：

- 有一个SSL错误(例如在自签名证书的情况下)。
- 目标URL无效。
- 导航过程中超时。
- 远程服务器没有响应或不可达。
- `main` 资源加载失败。

当远程服务器返回任何有效的HTTP状态码时，该方法不会抛出错误，包括404 "not Found"和500 "Internal server error"。这些响应的状态代码可以通过调用[response.status](#)来获取。

#### NOTE

该方法要么抛出错误，要么返回主资源响应。唯一的例外是导航到 `空白` 或导航到相同的URL与不同的哈希，这将成功并返回`null`。

#### NOTE

无头模式不支持PDF文档的导航。参见[upstream issue](#)。

---

## `# frame.hover(selector, **kwargs)`<#>

- **selector** <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#)。<#>
- **force** <bool> 是否绕过[actionability](#)检查。默认值为 **false**。<#>
- **modifiers** <List["Alt"|"Control"|"Meta"|"Shift"]> modifiers 按键要按。确保在操作期间只按下这些修饰符，然后恢复当前的修饰符。如果未指定，则使用当前按下的修饰符。<#>
- **position** <Dict> 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点。<#>
  - **x** <float>
  - **y** <float>
- **strict** <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。<#>
- **timeout** <float> 最大时间，单位为毫秒，默认为30秒，通过 **0** 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- **trial** <bool> 设置后，该方法只执行[actionability](#)检查，并跳过操作。默认值为 **false**。在元素准备好时再执行动作是很有用的。<#>
- returns: <NoneType> <#>

该方法通过执行以下步骤悬停在元素匹配 **选择器** 上：

1. 找到一个元素匹配选择器。如果没有，则等待直到匹配的元素被附加到DOM.
2. 等待匹配元素的[actionability](#)检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作.
3. 如果需要，将元素滚动到视图中.
4. 使鼠标停在元素中心或指定位置上.
5. 等待发起的导航成功或失败，除非设置了 **noWaitAfter** 选项.

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

## **# frame.inner\_html(selector, \*\*kwargs) #**

- **selector** <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#)。<#>



- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。<#>
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- returns: `<str>` <#>

返回 `element.innerHTML` .

## # `frame.inner_text(selector, **kwargs)` <#>

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#) .<#>
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。<#>
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- returns: `<str>` <#>

返回 `element.innerText` .

## # `frame.input_value(selector, **kwargs)` <#>

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#) .<#>
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。<#>
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- returns: `<str>` <#>

返回以下元素的输入值 `\<input>` or `\<textarea>` or `\<select>` .排除非输入元素.

## # `frame.is_checked(selector, **kwargs)` #

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#) .#
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用[browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: `<bool>` .#

返回元素是否被选中。如果元素不是复选框或单选输入则排除。

## # `frame.is_detached()` #

---

- returns: `<bool>` .#

如果 frame 已被分离，则返回 true，否则返回 false.

## # `frame.is_disabled(selector, **kwargs)` #

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#) .#
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用[browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: `<bool>` .#

返回该元素是否被禁用，与启用相反 [enabled](#).

## # `frame.is_editable(selector, **kwargs)` #

---

- `selector` <[str](#)> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#) .#
- `strict` <[bool](#)> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: <[bool](#)>.#

返回元素是否可编辑 [editable](#).

## # `frame.is_enabled(selector, **kwargs)` #

---

- `selector` <[str](#)> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#) .#
- `strict` <[bool](#)> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: <[bool](#)>.#

返回元素是否被启用 [enabled](#).

## # `frame.is_hidden(selector, **kwargs)` #

---

- `selector` <[str](#)> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#) .#

- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。<#>
- `timeout` `<float>` **DEPRECATED** 此选项将被忽略。  
[frame.is\\_hidden\(selector, \\*\\*kwargs\)](#) 立即返回,不会等待元素被隐藏。<#>
- returns: `<bool>` <#>

返回元素是否隐藏，与[可见](#)相反。不匹配任何元素的选择器被认为是隐藏的。

## # `frame.is_visible(selector, **kwargs)` <#>

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#)。<#>
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。<#>
- `timeout` `<float>` **DEPRECATED** 此选项将被忽略。  
[frame.is\\_visible\(selector, \\*\\*kwargs\)](#) 立即返回,不会等待元素变得可见。<#>
- returns: `<bool>` <#>

返回元素是否[可见](#)。不匹配任何元素的选择器被认为不可见

## # `frame.locator(selector, **kwargs)` <#>

---

- `selector` `<str>` 解析DOM元素时使用的选择器。有关更多细节，请参阅[working with selectors](#)。<#>
- `has` `<Locator>` 对selector选中的元素进行再次匹配，匹配目标中的子元素，例如：匹配子元素的 `text=Playwright` 的元素 `\<article>\<div>Playwright\</div>\</article>`。<#>  
 请注意，外部和内部定位器必须属于同一个 frame。内部定位器不能包含 [FrameLocator](#)。
- `has_text` `<str|Pattern>` 匹配包含指定文本的元素，可能在子元素或后代元素中,例如: `"Playwright"` 匹配 `\<article>\<div>Playwright\</div>\</article>`。<#>
- returns: `<Locator>` <#>

该方法返回一个元素定位器，可用于在`frame`中执行操作。在执行一个操作之前，`Locator`被立即解析为元素，因此同一定位器上的一系列操作实际上可以在不同的DOM元素上执行。如果这些动作之间的DOM结构发生了变化，就会发生这种情况。

## # `frame.name` #

---

- returns: `<str>` #

返回在标签中指定的 `frame` 的 `name` 属性。

如果名称为空，则返回 `id` 属性。

### NOTE

这个值在 `frame` 创建时计算一次，如果属性后来改变，这个值将不会更新。

## # `frame.page` #

---

- returns: `<Page>` #

返回包含此`frame`的页面。

## # `frame.parent_frame` #

---

- returns: `<NoneType|Frame>` #

父`frame`，如果有的话。分离的`frame`和主`frame`返回 `null`。

## # `frame.press(selector, key, **kwargs)` #

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#) #
- `key` `<str>` 要按下的 `键名` 或要生成的字符，如 `ArrowLeft` 或 `'a'` #

- `delay` `<float>` `keydown` 和 `keyup` 之间的等待时间，单位是毫秒。默认为0.#
- `no_wait_after` `<bool>` 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false`.#
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改.#
- returns: `<NoneType>` .#

`key`可以指定想要的 `keyboardEvent.key` 或是单个字符生成的文本,这里可以找到键值的超集。键的例子如下:

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

还支持以下快捷键: `Shift`, `Control`, `Alt`, `Meta`, `ShiftLeft`.

按住 `Shift` 键将输入与大写键对应的文本.

如果 `key` 是单个字符，它是区分大小写的，因此值 `a` 和 `A` 将生成不同的文本.

也支持快捷键，如键:“Control+o”或键:“Control+Shift+T”。当用修饰符指定时，修饰符被按下并被保持，而随后的键被按下.

## # `frame.query_selector(selector, **kwargs)` #

---

- `selector` `<str>` 要查询的选择器。有关更多细节，请参阅 [working with selectors](#).#
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- returns: `<NoneType|ElementHandle>` .#

返回指向frame元素的ElementHandle。

#### CAUTION

不鼓励使用 [ElementHandle](#)，而是使用 [Locator](#) 对象和web优先断言

该方法在框架中查找与指定选择器匹配的元素。有关详细信息，请参阅 [Working with selectors](#)。如果没有元素匹配该选择器，则返回 `null`

## # frame.query\_selector\_all(selector)#

- `selector` [<str>](#) 要查询的选择器。有关更多细节，请参阅 [working with selectors.#](#)
- returns: [<List\[ElementHandle\]>#](#)

返回指向frame元素的ElementHandles。

#### CAUTION

不鼓励使用 [ElementHandle](#)，而是使用 [Locator](#) 对象

该方法在框架中查找与指定选择器匹配的所有元素。有关详细信息，请参阅 [Working with selectors](#)。如果没有元素匹配该选择器，则返回 `空数组`

## # frame.select\_option(selector, \*\*kwargs)#

- `selector` [<str>](#) 要查询的选择器。有关更多细节，请参阅 [working with selectors.#](#)
- `force` [<bool>](#) 是否绕过 [actionability](#) 检查。默认值为 `false` [.#](#)
- `no_wait_after` [<bool>](#) 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` [.#](#)
- `strict` [<bool>](#) 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常 [.#](#)
- `timeout` [<float>](#) 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改 [.#](#)



- `element` <[ElementHandle](#)|[List](#)[[ElementHandle](#)]> 要选择的选项。  
可选的.<#>
- `index` <[int](#)|[List](#)[[int](#)]> 按索引进行选择的选项。可选的.<#>
- `value` <[str](#)|[List](#)[[str](#)]> 按值选择的选项。如果 `\<select>` 具有多个属性，则选择所有给定的选项，否则只选择与传递的选项之一匹配的的第一个选项。可选的.<#>
- `label` <[str](#)|[List](#)[[str](#)]> 按标签进行选择的选项。如果 `\<select>` 具有多个属性，则选择所有给定的选项，否则只选择与传递的选项之一匹配的的第一个选项。可选的.<#>
- `returns:` <[List](#)[[str](#)]><#>

这个方法等待元素匹配选择器，等待[可操作性](#)检查，直到所有指定的选项都出现在 `\<select>` 元素中，并选择这些选项。

如果目标元素不是 `\<select>` 元素，此方法将抛出一个错误。但是，如果该元素位于 `\<label>` 元素中，且该元素具有关联控件，则将使用该控件。

返回已成功选择的选项值的数组。

一旦选择了所有提供的选项，就触发一个更改和输入事件。

- Sync

```
# single selection matching the value
frame.select_option("select#colors", "blue")
# single selection matching both the label
frame.select_option("select#colors", label="blue")
# multiple selection
frame.select_option("select#colors", value=["red", "green", "blue"])
```

- Async

```
# single selection matching the value
await frame.select_option("select#colors", "blue")
# single selection matching the label
await frame.select_option("select#colors", label="blue")
# multiple selection
await frame.select_option("select#colors", value=["red", "green", "blue"])
```



## # frame.set\_checked(selector, checked, \*\*kwargs)#

---

- **selector** <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#)。[.#](#)
- **checked** <bool> 是否选中或不选中复选框。[.#](#)
- **force** <bool> 是否绕过[actionability](#)检查。默认值为 **false**。[.#](#)
- **no\_wait\_after** <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 **false**。[.#](#)
- **position** <Dict> 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点。[.#](#)
  - **x** <float>
  - **y** <float>
- **strict** <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。[.#](#)
- **timeout** <float> 最大时间，单位为毫秒，默认为30秒，通过 **0** 禁用超时。默认值可以通过使用[browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。[.#](#)
- **trial** <bool> 设置后，该方法只执行[actionability](#)检查，并跳过操作。默认值为 **false**。在元素准备好时再执行动作是很有用的。[.#](#)
- **returns:** <NoneType> [.#](#)

这个方法通过执行以下步骤检查或取消检查元素匹配选择器：

1. 找到一个元素匹配选择器。如果没有，则等待直到匹配的元素被附加到DOM.
2. 确保匹配的元素是一个复选框或单选输入。如果不是，则排除此方法.
3. 如果元素已经具有正确的选中状态，则该方法立即返回.
4. 等待匹配元素的[actionability](#)检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作.
5. 如果需要，将元素滚动到视图中.
6. 使用 [page.mouse](#) 单击元素的中心.
7. 等待已启动的导航成功或失败，除非设置了 **no\_wait\_after** 选项.
8. 确保元素现在被选中或取消选中。如果不是，则抛出此方法.

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

## # `frame.set_content(html, **kwargs)` #

---

- `html` <[str](#)> 要分配给页面的html标记. #
- `timeout` <[float](#)> 最大操作时间，单位为毫秒，默认为30秒，通过0表示禁止超时。默认值可以通过使用 [browser\\_context.set\\_default\\_navigation\\_timeout\(timeout\)](#), [browser\\_context.set\\_default\\_timeout\(timeout\)](#), [page.set\\_default\\_navigation\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来修改. #
- `wait_until` <"load"|"domcontentloaded"|"networkidle"|"commit"> 当认为操作成功时，默认为 `load`。事件可以是: #
  - `'domcontentloaded'` - 当 `domcontentloaded` 事件被触发时，认为操作已经完成。
  - `'load'` - 当触发 `load` 事件时，认为操作已经完成。
  - `'networkidle'` - 当至少 `500毫秒` 没有网络连接时，认为操作已经完成。
  - `'commit'` - 当接收到网络响应并开始加载文档时，认为操作已经完成。
- returns: <[NoneType](#)> #

## # `frame.set_input_files(selector, files, **kwargs)` #

---

- `selector` <[str](#)> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅 [working with selectors](#). #
- `files` <[Union](#)[[str](#), [pathlib.Path](#)]|[List](#)[[Union](#)[[str](#), [pathlib.Path](#)]]|[Dict](#)|[List](#)[[Dict](#)]> #
  - `name` <[str](#)> File name
  - `mimeType` <[str](#)> File type
  - `buffer` <[bytes](#)> File content

- `no_wait_after` `<bool>` 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` `.#`
- `strict` `<bool>` 当为`true`时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常 `.#`
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改 `.#`
- `returns:` `<NoneType>` `.#`

这个方法期望选择器指向一个 [输入元素](#)。

将文件输入的值设置为这些文件路径或文件。如果某些 `filepath` 是相对路径，那么它们将相对于当前工作目录进行解析。对于空数组，清除选定的文件。

## # `frame.tap(selector, **kwargs)` `.#`

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅 [working with selectors](#) `.#`
- `force` `<bool>` 是否绕过 `actionability` 检查。默认值为 `false` `.#`
- `modifiers` `<List["Alt"|"Control"|"Meta"|"Shift"]>` `modifiers` 按键要按。确保在操作期间只按下这些修饰符，然后恢复当前的修饰符。如果未指定，则使用当前按下的修饰符 `.#`
- `no_wait_after` `<bool>` 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` `.#`
- `position` `<Dict>` 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点 `.#`
  - `x` `<float>`
  - `y` `<float>`
- `strict` `<bool>` 当为`true`时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常 `.#`

- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。<#>
- `trial` `<bool>` 设置后，该方法只执行 `actionability` 检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的。<#>
- returns: `<NoneType>` <#>

这个方法通过执行以下步骤点击元素匹配选择器：

1. 找到一个元素匹配选择器。如果没有，则等待直到匹配的元素被附加到 DOM.
2. 等待匹配元素的 `actionability` 检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作.
3. 如果需要，将元素滚动到视图中.
4. 点击页面中心或指定位置.
5. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项.

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 `TimeoutError`。传递零超时将禁用此功能。

#### NOTE

`frame.tap()` 要求浏览器上下文的 `hasTouch` 选项被设置为 `True`。

## # `frame.text_content(selector, **kwargs)` <#>

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅 [working with selectors](#)。<#>
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。<#>
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。<#>
- returns: `<NoneType|str>` <#>

返回 `element.textContent`。

## # `frame.title()` #

---

- returns: `<str>` #

返回页面标题。

## # `frame.type(selector, text, **kwargs)` #

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅 [working with selectors](#) .#
- `text` `<str>` 要输入到焦点元素中的文本.#
- `delay` `<float>` 按键之间的等待时间，单位是毫秒。默认为0.#
- `no_wait_after` `<bool>` 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: `<NoneType>` #

为文本中的每个字符发送 `keydown` , `keypress` / `input` , and `keyup` 事件。`frame.type` 可用于发送细粒度键盘事件。要在表单字段中填充值，请使用 [frame.fill\(selector, value, \\*\\*kwargs\)](#) .

要按一个特殊的键，如 `Control` 或 `ArrowDown` ，使用 [keyboard.press\(key, \\*\\*kwargs\)](#) .

- Sync

```
frame.type("#mytextarea", "hello") # types instantly
frame.type("#mytextarea", "world", delay=100) # types slower,
like a user
```

- Async

```
await frame.type("#mytextarea", "hello") # types instantly
await frame.type("#mytextarea", "world", delay=100) # types
slower, like a user
```

## # `frame.uncheck(selector, **kwargs)` #

- `selector` <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#) .#
- `force` <bool> 是否绕过[actionability](#)检查。默认值为 `false` .#
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- `position` <Dict> 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点 .#
  - `x` <float>
  - `y` <float>
- `strict` <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常 .#
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用[browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改 .#
- `trial` <bool> 设置后，该方法只执行[actionability](#)检查，并跳过操作。默认值为 `false` 。在元素准备好时再执行动作是很有用的 .#
- returns: <NoneType> .#

这个方法通过以下步骤检查元素匹配选择器::

1. 找到一个元素匹配选择器。如果没有，则等待直到匹配的元素被附加到DOM.
2. 确保匹配的元素是一个复选框或单选输入。如果不是，则排除此方法。  
If the element is already unchecked, this method returns immediately.
3. 等待匹配元素的[actionability](#)检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作.
4. 如果需要，将元素滚动到视图中.
5. 使用 [page.mouse](#) 单击元素的中心.

6. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项。
7. 确保元素现在是未选中的。如果不是，则排除此方法。

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

## # `frame.url` #

---

- returns: [<str>](#) #

返回 frame's url.

## # `frame.wait_for_function(expression, **kwargs)` #

---

- `expression` [<str>](#) 要在浏览器上下文中计算的 `JavaScript` 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值。#
- `arg` [<EvaluationArgument>](#) 传递给 `expression` 的可选参数。#
- `polling` [<float|'raf'>](#) 如果 `polling` 是 'raf'，那么表达式会在 `requestAnimationFrame` 回调中持续执行。如果 `polling` 是一个数字，那么它将被视为执行函数的毫秒间隔。默认为 `raf`。#
- `timeout` [<float>](#) 最大等待时间，以毫秒为单位。默认为 `30000` (30 秒)。通过 0 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) 来更改。#
- returns: [<JSHandle>](#) #

当表达式返回真值时，返回该值。

[frame.wait\\_for\\_function\(expression, \\*\\*kwargs\)](#) 可用于观察视口大小的变化：

- Sync



```

from playwright.sync_api import sync_playwright

def run(playwright):
    webkit = playwright.webkit
    browser = webkit.launch()
    page = browser.new_page()
    page.evaluate("window.x = 0; setTimeout(() => { window.x = 100 }, 1000);")
    page.main_frame.wait_for_function("() => window.x > 0")
    browser.close()

with sync_playwright() as playwright:
    run(playwright)

```

- Async

```

import asyncio
from playwright.async_api import async_playwright

async def run(playwright):
    webkit = playwright.webkit
    browser = await webkit.launch()
    page = await browser.new_page()
    await page.evaluate("window.x = 0; setTimeout(() => { window.x = 100 }, 1000);")
    await page.main_frame.wait_for_function("() => window.x > 0")
    await browser.close()

async def main():
    async with async_playwright() as playwright:
        await run(playwright)
asyncio.run(main())

```

将参数传递给 `frame.waitForFunction` 函数:

- Sync

```

selector = ".foo"
frame.wait_for_function("selector => !!document.querySelector(selector)", selector)

```



- Async

```
selector = ".foo"
await frame.wait_for_function("selector =>
!!document.querySelector(selector)", selector)
```

## # frame.wait\_for\_load\_state(\*\*kwargs)#

- **state** <"load"|"domcontentloaded"|"networkidle"> 可选加载状态，等待，默认为 **load**。如果在加载当前文档时已经达到该状态，该方法将立即进行解析。可以是：#
  - **'load'** - 等待 **load** 事件被触发。
  - **'domcontentloaded'** - 等待 **domcontentloaded** 事件被触发。
  - **'networkidle'** - 等待至少500毫秒没有网络连接。
- **timeout** <float>最大操作时间，单位为毫秒，默认为30秒，通过0表示禁止超时。默认值可以通过使用 [browser\\_context.set\\_default\\_navigation\\_timeout\(timeout\)](#), [browser\\_context.set\\_default\\_timeout\(timeout\)](#), [page.set\\_default\\_navigation\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来修改。#
- returns: <NoneType> #

等待到达所需的加载状态。

当 **frame** 达到所需的加载状态时返回，默认为加载。该导航必须在调用此方法时已提交。如果当前文档已经达到了所需的状态，则立即进行解析。

- Sync

```
frame.click("button") # click triggers navigation.
frame.wait_for_load_state() # the promise resolves after
"load" event.
```

- Async

```
await frame.click("button") # click triggers navigation.
await frame.wait_for_load_state() # the promise resolves after
"load" event.
```

## # frame.wait\_for\_selector(selector, \*\*kwargs) #

---

- `selector` <str> 要查询的选择器。有关更多细节，请参阅[working with selectors](#). #
- `state` <"attached"|"detached"|"visible"|"hidden"> 默认为 `"visible"`。可以是: #
  - `'attached'` - 等待元素出现在DOM中。
  - `'detached'` - 等待元素在DOM中不存在。
  - `'visible'` - 等待元素有非空的边界框 且 没有 `visibility:hidden`。注意，没有任何内容或带有 `display:none` 的元素有一个空的边界框，因此不被认为是可见的。
  - `'hidden'` - 等待元素从DOM中分离出来，或有一个空的边界框或 `visibility:hidden`。这与 `"visible"` 选项相反。
- `strict` <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。 #
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。 #
- returns: <[NoneType](#) | [ElementHandle](#)> #

当选择器指定的元素满足状态选项时返回。如果等待 `隐藏` 或 `分离`，则返回 `null`。

### NOTE

*Playwright* 在执行操作之前，自动等待元素准备就绪。使用 [Locator](#) 对象和 `web-first` 断言会使代码不需要等待选择器。

等待 `'selector'` 满足 `'state'` 选项(要么从dom中出现/消失，要么变成可见/隐藏)。如果在调用方法 `'selector'` 的时候已经满足条件，该方法将立即返回。如果选择器不满足 `'timeout'` 毫秒的条件，函数将排除。

此方法适用于多个导航：

- Sync

```

from playwright.sync_api import sync_playwright

def run(playwright):
    chromium = playwright.chromium
    browser = chromium.launch()
    page = browser.new_page()
    for current_url in ["https://google.com",
                        "https://bbc.com"]:
        page.goto(current_url, wait_until="domcontentloaded")
        element = page.main_frame.wait_for_selector("img")
        print("Loaded image: " +
              str(element.get_attribute("src")))
    browser.close()

with sync_playwright() as playwright:
    run(playwright)

```

- Async

```

import asyncio
from playwright.async_api import async_playwright

async def run(playwright):
    chromium = playwright.chromium
    browser = await chromium.launch()
    page = await browser.new_page()
    for current_url in ["https://google.com",
                        "https://bbc.com"]:
        await page.goto(current_url,
                        wait_until="domcontentloaded")
        element = await
page.main_frame.wait_for_selector("img")
        print("Loaded image: " + str(await
element.get_attribute("src")))
        await browser.close()

async def main():
    async with async_playwright() as playwright:
        await run(playwright)
    asyncio.run(main())

```

## # frame.wait\_for\_timeout(timeout)#

---

- `timeout` <[float](#)> 等待超时时间#
- returns: <[NoneType](#)>#

等待给定超时(以毫秒为单位).

注意, `frame.waitForTimeout()` 应该只用于调试。在生产中使用计时器的测试将是不可靠的。使用信号, 如网络事件, 选择器变得可见和其他方式。

## # frame.wait\_for\_url(url, \*\*kwargs)#

---

- `url` <[str](#)|[Pattern](#)|[Callable](#)[[URL](#)]:[bool](#)> 一个glob模式、regex模式或谓词, 在等待导航时接收匹配的url。注意, 如果参数是一个不带通配符的字符串, 该方法将等待导航到与该字符串完全相等的URL.#
- `timeout` <[float](#)>最大操作时间, 单位为毫秒, 默认为30秒, 通过0表示禁止超时。默认值可以通过使用 [browser\\_context.set\\_default\\_navigation\\_timeout\(timeout\)](#), [browser\\_context.set\\_default\\_timeout\(timeout\)](#), [page.set\\_default\\_navigation\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来修改.#
- `wait_until` <"load"|"domcontentloaded"|"networkidle"|"commit"> 当认为操作成功时, 默认为 `load`。事件可以是:#
  - `'domcontentloaded'` - 当 `domcontentloaded` 事件被触发时, 认为操作已经完成.
  - `'load'` - 当触发 `load` 事件时, 认为操作已经完成.
  - `'networkidle'` - 当至少 `500毫秒` 没有网络连接时, 认为操作已经完成.
  - `'commit'` - 当接收到网络响应并开始加载文档时, 认为操作已经完成.
- returns: <[NoneType](#)>#

等待frame导航到给定的URL.

- Sync

```
frame.click("a.delayed-navigation") # clicking the link will
indirectly cause a navigation
frame.wait_for_url("**/target.html")
```

- Async

```
await frame.click("a.delayed-navigation") # clicking the link
will indirectly cause a navigation
await frame.wait_for_url("**/target.html")
```

## FrameLocator

FrameLocator表示页面上 `iframe` 的视图。它捕获的逻辑足以检索 `iframe` 并定位该 `iframe` 中的元素。framelocator可以用 [page.frame\\_locator\(selector\)](#) or [locator.frame\\_locator\(selector\)](#) 方法创建。

- Sync

```
locator = page.frame_locator("my-
frame").locator("text=Submit")
locator.click()
```

- Async

```
locator = page.frame_locator("#my-
frame").locator("text=Submit")
await locator.click()
```

## Strictness

框架定位是严格的。这意味着，如果有多个元素匹配给定的选择器，那么帧定位器上的所有操作都会抛出。

- Sync

```
# Throws if there are several frames in DOM:
page.frame_locator('.result-frame').locator('button').click()

# Works because we explicitly tell locator to pick the first
frame:
page.frame_locator('.result-frame').first.locator('button').click()
```

- Async

```
# Throws if there are several frames in DOM:
await page.frame_locator('.result-frame').locator('button').click()

# Works because we explicitly tell locator to pick the first
frame:
await page.frame_locator('.result-frame').first.locator('button').click()
```

## Converting Locator to FrameLocator

如果你有一个指向 `iframe` 的 `Locator` 对象，可以使用以下方法将其转换为 `FrameLocator`，使用 `:scope` CSS 选择器：

- Sync

```
frameLocator = locator.frame_locator(":scope");
```

- Async

```
frameLocator = locator.frame_locator(":scope");
```

- [frame\\_locator.first](#)
- [frame\\_locator.frame\\_locator\(selector\)](#)
- [frame\\_locator.last](#)
- [frame\\_locator.locator\(selector, \\*\\*kwargs\)](#)
- [frame\\_locator.nth\(index\)](#)

## # frame\_locator.first#

---

- returns: <[FrameLocator](#)>#

返回第一个匹配 frame 的定位符.

## # frame\_locator.frame\_locator(selector)#

---

- **selector** <[str](#)> 解析DOM元素时使用的选择器。有关更多细节, 请参阅[working with selectors](#) .#
- returns: <[FrameLocator](#)>#

在使用iframes时, 您可以创建一个frame 定位器, 该定位器将进入iframe并允许选择该iframe中的元素.

## # frame\_locator.last#

---

- returns: <[FrameLocator](#)>#

返回最后一个匹配frame 的定位符.

## # frame\_locator.locator(selector, \*\*kwargs)#

---

- **selector** <[str](#)> 解析DOM元素时使用的选择器。有关更多细节, 请参阅[working with selectors](#) .#
- **has** <[Locator](#)> 对selector选中的元素进行再次匹配, 匹配目标中的子元素, 例如: 匹配子元素的 **text=Playwright** 的元素 `\<article>\<div>Playwright\</div>\</article>` .#

请注意, 外部和内部定位器必须属于同一个 frame. 内部定位器不能包含 [FrameLocator](#).

- **has\_text** <[str](#)|[Pattern](#)> 匹配包含指定文本的元素, 可能在子元素或后代元素中, 例如: `"Playwright"` 匹配 `\<article>\<div>Playwright\</div>\</article>` .#

- returns: <[Locator](#)>#

该方法在framocator的子树中找到与指定选择器匹配的元素。

## # frame\_locator.nth(index)#

---

- `index` <[int](#)>#
- returns: <[FrameLocator](#)>#

返回第n个匹配frame 的定位符。

## JSHandle

JSHandle表示一个页内JavaScript对象。JSHandles可以用[page.evaluate\\_handle\(expression, \\*\\*kwargs\)](#) 方法创建。

- Sync

```
window_handle = page.evaluate_handle("window")
# ...
```

- Async

```
window_handle = await page.evaluate_handle("window")
# ...
```

jhandle防止被引用的JavaScript对象被垃圾收集，除非该句柄被[js\\_handle.dispose\(\)](#) 指定。当JSHandles的原始frame 被导航或者父上下文被销毁时，JSHandles会被自动销毁。

JSHandle实例可以当作 [page.eval\\_on\\_selector\(selector, expression, \\*\\*kwargs\)](#), [page.evaluate\(expression, \\*\\*kwargs\)](#) and [page.evaluate\\_handle\(expression, \\*\\*kwargs\)](#) 方法中的一个参数。



- [js\\_handle.as\\_element\(\)](#)
- [js\\_handle.dispose\(\)](#)
- [js\\_handle.evaluate\(expression, \\*\\*kwargs\)](#)
- [js\\_handle.evaluate\\_handle\(expression, \\*\\*kwargs\)](#)
- [js\\_handle.get\\_properties\(\)](#)
- [js\\_handle.get\\_property\(property\\_name\)](#)
- [js\\_handle.json\\_value\(\)](#)

## # js\_handle.as\_element()#

---

- returns: <[NoneType](#)|[ElementHandle](#)>#

如果对象句柄是[ElementHandle](#)的实例，则返回 `null` 或对象句柄本身。

## # js\_handle.dispose()#

---

- returns: <[NoneType](#)>#

`jsHandle.dispose` 方法停止引用元素句柄。

## # js\_handle.evaluate(expression, \*\*kwargs)#

---

- `expression` <[str](#)> 要在浏览器上下文中计算的 `JavaScript` 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值。#
- `arg` <[EvaluationArgument](#)> 传递给 `expression` 的可选参数。#
- returns: <[Serializable](#)>#

返回表达式的返回值。

此方法将此句柄作为第一个参数传递给表达式。

如果表达式返回[Promise](#)，则 `handle.evaluate` 将等待promise解析并返回其值

例子:

- Sync

```
tweet_handle = page.query_selector(".tweet .retweets")
assert tweet_handle.evaluate("node ⇒ node.innerText") == "10 retweets"
```

- Async

```
tweet_handle = await page.query_selector(".tweet .retweets")
assert await tweet_handle.evaluate("node ⇒ node.innerText") == "10 retweets"
```

## # js\_handle.evaluate\_handle(expression, \*\*kwargs) #

---

- `expression` <str> 要在浏览器上下文中计算的 `JavaScript` 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值。#
- `arg` <EvaluationArgument> 传递给 `expression` 的可选参数。#
- returns: <JSHandle> #

返回表达式的返回值是一个 [JSHandle](#)。

此方法将此句柄作为第一个参数传递给表达式。

`js_handle.evaluate` 和 `js_handle.evaluate_handle` 两者之间的唯一区别是: `js_handle.evaluate_handle` 返回 [JSHandle](#)。

如果函数传递给 `js_handle.evaluate_handle` 返回一个 [Promise](#), 则 `js_handle.evaluate_handle` 将等待promise解析并返回它的值。

更多细节参考: [page.evaluate\\_handle\(expression, \\*\\*kwargs\)](#)。

## # js\_handle.get\_properties() #

---

- returns: <[Map][[str](<https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str>), [JSHandle](#)]>#

该方法返回一个映射，其中包含自己的属性名作为键，jshandle实例作为属性值。

- Sync

```
handle = page.evaluate_handle("{window, document}")
properties = handle.get_properties()
window_handle = properties.get("window")
document_handle = properties.get("document")
handle.dispose()
```

- Async

```
handle = await page.evaluate_handle("{window, document}")
properties = await handle.get_properties()
window_handle = properties.get("window")
document_handle = properties.get("document")
await handle.dispose()
```

## # js\_handle.get\_property(property\_name)#

- `property_name` <[str](#)> 属性获取#
- returns: <[JSHandle](#)>#

从被引用的对象中获取单个属性。

## # js\_handle.json\_value()#

- returns: <[Serializable](#)>#

返回对象的JSON表示。如果对象有toJSON函数，它将不会被调用。

NOTE

如果引用的对象不是可字符串化的，该方法将返回一个空的JSON对象。  
如果对象有循环引用，它将抛出一个错误。

## Keyboard

Keyboard提供了一个用于管理虚拟键盘的api。高级api是 [keyboard.type\(text, \\*\\*kwargs\)](#)，它接受原始字符并在页面上生成适当的keydown、keypress/input和keyup事件。

为了进行更精细的控制，可以使用 [keyboard.down\(key\)](#)，[keyboard.up\(key\)](#)，and [keyboard.insert\\_text\(text\)](#) 手动触发事件，就像它们是从真正的键盘中生成的一样。

按住 **Shift** 键来选择和删除一些文本的例子：

- Sync

```
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!"
```

- Async

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

一个按大写字母 **A** 的例子

- Sync

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

- Async

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

一个用键盘触发全部选择的例子

- Sync

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

- Async

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

- [keyboard.down\(key\)](#)
- [keyboard.insert\\_text\(text\)](#)
- [keyboard.press\(key, \\*\\*kwargs\)](#)
- [keyboard.type\(text, \\*\\*kwargs\)](#)
- [keyboard.up\(key\)](#)

---

**# keyboard.down(key) #**

- `key` <[str](#)> 要按下的 `键名` 或要生成的字符, 如 `ArrowLeft` 或 `'a'.`<#>
- returns: <[NoneType](#)><#>

分派一个按键事件.

`key` 可以指定想要的 [keyboardEvent.key](#) 或是单个字符生成的文本, 这里可以找到键值的超集. 键的例子如下:

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

还支持以下快捷键: `Shift`, `Control`, `Alt`, `Meta`, `ShiftLeft`.

按住 `Shift` 键将输入与大写键对应的文本.

如果 `key` 是单个字符, 它是区分大小写的, 因此值 `a` 和 `A` 将生成不同的文本.

如果 `key` 是一个修饰键, `Shift`, `Meta`, `Control`, or `Alt`, 随后的按键将发送该修饰器激活. 要释放修饰键, 请使用 [keyboard.up\(key\)](#).

按下该键一次后, 随后调用 [keyboard.down\(key\)](#) 将 [重复](#) 设置为 `true`. 释放键, 使用 [keyboard.up\(key\)](#).

#### NOTE

修改键会产生影响 `keyboard.down`. 按住 `Shift` 键将输入大写的文本.

## # keyboard.insert\_text(text)<#>

- `text` <[str](#)> 设置输入为指定的文本值.<#>
- returns: <[NoneType](#)><#>

只分派输入事件, 不发出 `keydown`, `keyup` or `keypress` 等事件.

- Sync

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

- Async

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

#### NOTE

修饰键不影响 `keyboard.insertText`. 按下 `Shift` 键不会输入大写的文本.

## # keyboard.press(key, \*\*kwargs) #

- `key` <str> 要按下的 `键名` 或要生成的字符, 如 `ArrowLeft` 或 `'a'.` #
- `delay` <float> `keydown` 和 `keyup` 之间的等待时间, 单位是毫秒。默认为0. #
- returns: <NoneType> #

`key`可以指定想要的 [keyboardEvent.key](#) 或是单个字符生成的文本,这里可以找到键值的超集。键的例子如下:

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

还支持以下快捷键: `Shift`, `Control`, `Alt`, `Meta`, `ShiftLeft`.

按住 `Shift` 键将输入与大写键对应的文本.

如果 `key` 是单个字符, 它是区分大小写的, 因此值 `a` 和 `A` 将生成不同的文本.

也支持快捷键, 如键:“Control+o”或键:“Control+Shift+T”. 当用修饰符指定时, 修饰符被按下并被保持, 而随后的键被按下.

- Sync

```

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+0")
page.screenshot(path="o.png")
browser.close()

```

- Async

```

page = await browser.new_page()
await page.goto("https://keycode.info")
await page.keyboard.press("a")
await page.screenshot(path="a.png")
await page.keyboard.press("ArrowLeft")
await page.screenshot(path="arrow_left.png")
await page.keyboard.press("Shift+0")
await page.screenshot(path="o.png")
await browser.close()

```

[keyboard.down\(key\)](#) and [keyboard.up\(key\)](#) 的快捷键.

## # keyboard.type(text, \*\*kwargs) #

- `text` <[str](#)> 要输入到焦点元素中的文本. #
- `delay` <[float](#)> 按键之间的等待时间，单位是毫秒。默认为0. #
- returns: <[NoneType](#)> #

为文本中的每个字符发送 `keydown`, `keypress` / `input`, and `keyup` 事件.

要按一个特殊的键，如 `Control` 或 `ArrowDown`，使用 [keyboard.press\(key, \\*\\*kwargs\)](#).

- Sync



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

- Async

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

#### NOTE

修饰键不影响 `keyboard.type`。按下 `Shift` 键不会输入大写的文本。

#### NOTE

对于非美式键盘上的字符，只会发送一个 `输入` 事件。

## # keyboard.up(key) #

- `key` <str> 要按下的 `键名` 或要生成的字符，如 `ArrowLeft` 或 `'a'.` #
- returns: <NoneType> #

分派一个 `keyup` 事件。

## Locator

`locator 定位器` 是 playwright 自动等待和重试能力的核心部分。简而言之，定位器表示一种随时在页面上查找元素的方法。可以使用页面创建定位器 `page.locator(selector, **kwargs)` 。

[Learn more about locators.](#)

- [locator.all\\_inner\\_texts\(\)](#)
- [locator.all\\_text\\_contents\(\)](#)
- [locator.bounding\\_box\(\\*\\*kwargs\)](#)

- [locator.check\(\\*\\*kwargs\)](#)
- [locator.click\(\\*\\*kwargs\)](#)
- [locator.count\(\)](#)
- [locator.dbclick\(\\*\\*kwargs\)](#)
- [locator.dispatch\\_event\(type, \\*\\*kwargs\)](#)
- [locator.drag\\_to\(target, \\*\\*kwargs\)](#)
- [locator.element\\_handle\(\\*\\*kwargs\)](#)
- [locator.element\\_handles\(\)](#)
- [locator.evaluate\(expression, \\*\\*kwargs\)](#)
- [locator.evaluate\\_all\(expression, \\*\\*kwargs\)](#)
- [locator.evaluate\\_handle\(expression, \\*\\*kwargs\)](#)
- [locator.fill\(value, \\*\\*kwargs\)](#)
- [locator.first](#)
- [locator.focus\(\\*\\*kwargs\)](#)
- [locator.frame\\_locator\(selector\)](#)
- [locator.get\\_attribute\(name, \\*\\*kwargs\)](#)
- [locator.highlight\(\)](#)
- [locator.hover\(\\*\\*kwargs\)](#)
- [locator.inner\\_html\(\\*\\*kwargs\)](#)
- [locator.inner\\_text\(\\*\\*kwargs\)](#)
- [locator.input\\_value\(\\*\\*kwargs\)](#)
- [locator.is\\_checked\(\\*\\*kwargs\)](#)
- [locator.is\\_disabled\(\\*\\*kwargs\)](#)
- [locator.is\\_editable\(\\*\\*kwargs\)](#)
- [locator.is\\_enabled\(\\*\\*kwargs\)](#)
- [locator.is\\_hidden\(\\*\\*kwargs\)](#)
- [locator.is\\_visible\(\\*\\*kwargs\)](#)
- [locator.last](#)
- [locator.locator\(selector, \\*\\*kwargs\)](#)
- [locator.nth\(index\)](#)
- [locator.page](#)
- [locator.press\(key, \\*\\*kwargs\)](#)
- [locator.screenshot\(\\*\\*kwargs\)](#)
- [locator.scroll\\_into\\_view\\_if\\_needed\(\\*\\*kwargs\)](#)
- [locator.select\\_option\(\\*\\*kwargs\)](#)
- [locator.select\\_text\(\\*\\*kwargs\)](#)
- [locator.set\\_checked\(checked, \\*\\*kwargs\)](#)
- [locator.set\\_input\\_files\(files, \\*\\*kwargs\)](#)
- [locator.tap\(\\*\\*kwargs\)](#)
- [locator.text\\_content\(\\*\\*kwargs\)](#)
- [locator.type\(text, \\*\\*kwargs\)](#)
- [locator.uncheck\(\\*\\*kwargs\)](#)
- [locator.wait\\_for\(\\*\\*kwargs\)](#)

## # locator.all\_inner\_texts()#

---

- returns: `<List[str]>#`

返回一个 `node.innerText` 值为所有匹配节点。

## # locator.all\_text\_contents()#

---

- returns: `<List[str]>#`

返回一个节点数组值为所有节点的 `node.textContent` 。

## # locator.bounding\_box(\*\*kwargs)#

---

- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。`#`
- returns: `<NoneType|Dict>#`
  - `x` `<float>` 元素的 X 坐标像素。
  - `y` `<float>` 元素的Y坐标，以像素为单位。
  - `width` `<float>` 元素的宽度，以像素为单位。
  - `height` `<float>` 元素的高度，以像素为单位。

此方法返回元素的边界框，如果元素不可见，则返回 `null`。边界框是相对于主帧视口计算的——主帧视口通常与浏览器窗口相同。

滚动会影响返回的绑定框，类似于 `Element.getBoundingClientRect`，这意味着 `x`和/或`y`可能是负的。

与 `Element.getBoundingClientRect` 不同，子`frame`中的元素返回相对于主`frame`的边界框。

假设页面是静态的，使用边界框坐标执行输入是安全的。例如，下面的代码片段应该单击元素的中心。

- Sync

```
box = element.bounding_box()
page.mouse.click(box["x"] + box["width"] / 2, box["y"] +
box["height"] / 2)
```

- Async

```
box = await element.bounding_box()
await page.mouse.click(box["x"] + box["width"] / 2, box["y"] +
box["height"] / 2)
```

## # locator.check(\*\*kwargs)#

---

- `force` <bool> 是否绕过 [actionability](#) 检查。默认值为 `false` .#
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- `position` <Dict> 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点.#
  - `x` <float>
  - `y` <float>
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- `trial` <bool> 设置后，该方法只执行 [actionability](#) 检查，并跳过操作。默认值为 `false` 。在元素准备好时再执行动作是很有用的.#
- returns: <NoneType> .#

这个方法通过执行以下步骤来检查元素：

1. 确保元素是一个复选框或单选输入。如果不是，则抛出此方法。如果元素已被选中，则该方法立即返回。
2. 等待元素的 [可操作性](#) 检查，除非设置了强制选项。
3. 如果需要，将元素滚动到视图中。
4. 使用 [page.mouse](#) 单击元素的中心。
5. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项。
6. 确保元素现在被选中。如果不是，则排除此方法。

如果元素在动作期间的任何时刻与DOM分离，此方法将抛出。

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

## # locator.click(\*\*kwargs)#

---

- `button` <"left"|"right"|"middle"> 默认左 `left` [.#](#)
- `click_count` <[int](#)> 默认为1, 详情查看 [UIEvent.detail.#](#)
- `delay` <[float](#)> `mousedown` 和 `mouseup` 之间的等待时间，单位是毫秒。默认为0. [.#](#)
- `force` <[bool](#)> 是否绕过 [actionability](#) 检查。默认值为 `false` [.#](#)
- `modifiers` <[List](#)["Alt"|"Control"|"Meta"|"Shift"]> `modifiers` 按键要按。确保在操作期间只按下这些修饰符，然后恢复当前的修饰符。如果未指定，则使用当前按下的修饰符. [.#](#)
- `no_wait_after` <[bool](#)> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` [.#](#)
- `position` <[Dict](#)> 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点. [.#](#)
  - `x` <[float](#)>
  - `y` <[float](#)>
- `timeout` <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改. [.#](#)
- `trial` <[bool](#)> 设置后，该方法只执行 [actionability](#) 检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的. [.#](#)
- `returns`: <[NoneType](#)> [.#](#)

该方法执行以下步骤单击元素：

1. 等待元素的 [可操作性](#) 检查，除非设置了强制选项。
2. 如果需要，将元素滚动到视图中。
3. 使用 [page.mouse](#) 单击元素的中心, or the specified `position` .
4. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项。

如果元素在动作期间的任何时刻与DOM分离，此方法将抛出。

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

## # locator.count()#

---

- returns: <[int](#)>#

返回匹配给定选择器的元素数。

## # locator.dblclick(\*\*kwargs)#

---

- **button** <"left"|"right"|"middle"> 默认左 **left** .#
- **delay** <[float](#)> **mousedown** 和 **mouseup** 之间的等待时间，单位是毫秒。默认为0.#
- **force** <[bool](#)> 是否绕过 [actionability](#) 检查。默认值为 **false** .#
- **modifiers** <[List](#)["Alt"|"Control"|"Meta"|"Shift"]> modifiers 按键要按。确保在操作期间只按下这些修饰符，然后恢复当前的修饰符。如果未指定，则使用当前按下的修饰符.#
- **no\_wait\_after** <[bool](#)> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 **false** .#
- **position** <[Dict](#)> 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点.#
  - **x** <[float](#)>
  - **y** <[float](#)>
- **timeout** <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 **0** 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- **trial** <[bool](#)> 设置后，该方法只执行 [actionability](#) 检查，并跳过操作。默认值为 **false**。在元素准备好时再执行动作是很有用的.#
- returns: <[NoneType](#)>#

该方法执行以下步骤双击元素：

1. 等待元素的 [可操作性](#) 检查，除非设置了强制选项。
2. 如果需要，将元素滚动到视图中。
3. 使用 [page.mouse](#) 方法，双击元素中心位置。
4. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项。该方法执行以下步骤双击元素，注意，如果 `dblclick()` 的第一次单击触发了一个导航事件，则该方法将抛出。

如果元素在动作期间的任何时刻与DOM分离，此方法将抛出。

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

#### NOTE

`element.dblclick()` 分发两个 `click` 事件和一个 `dblclick` 事件。

## # locator.dispatch\_event(type, \*\*kwargs) #

- `type` <[str](#)> DOM事件类型: `"click"`，`"dragstart"` 等。<#>
- `event_init` <[EvaluationArgument](#)> 可选的特定于事件的初始化属性。<#>
- `timeout` <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- `returns`: <[NoneType](#)> <#>

下面的代码片段分派元素上的 `单击` 事件。无论元素的可见性状态如何，单击都将被分派。这相当于调用 [element.click\(\)](#)。

- Sync

```
element.dispatch_event("click")
```

- Async

```
await element.dispatch_event("click")
```

在底层，它根据给定的类型创建一个事件实例，使用 `event_init` 属性初始化它，并在元素上分派它。默认情况下，事件是组合的、可取消的和冒泡的。

由于 `event_init` 是特定于事件的，请参考事件文档中的初始属性列表：

- [DragEvent](#)
- [FocusEvent](#)
- [KeyboardEvent](#)
- [MouseEvent](#)
- [PointerEvent](#)
- [TouchEvent](#)
- [Event](#)

如果你想要将活动对象传递到事件中，你也可以指定 `jhandle` 作为属性值：

- Sync

```
# note you can only create data_transfer in chromium and
firefox
data_transfer = page.evaluate_handle("new DataTransfer()")
element.dispatch_event("#source", "dragstart",
{"dataTransfer": data_transfer})
```

- Async

```
# note you can only create data_transfer in chromium and
firefox
data_transfer = await page.evaluate_handle("new
DataTransfer()")
await element.dispatch_event("#source", "dragstart",
{"dataTransfer": data_transfer})
```

## # locator.drag\_to(target, \*\*kwargs)#

- `target` [<Locator>](#) 要拖动到的元素的定位器。<#>
- `force` [<bool>](#) 是否绕过 [actionability](#) 检查。默认值为 `false`。<#>
- `no_wait_after` [<bool>](#) 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false`。<#>



- `source_position` <[Dict](#)> 此时相对于元素填充框的左上角单击源元素。如果没有指定，则使用元素的某个可见点。<#>
  - `x` <[float](#)>
  - `y` <[float](#)>
- `target_position` <[Dict](#)> 此时相对于元素填充框的左上角落在目标元素上。如果没有指定，则使用元素的某个可见点。<#>
  - `x` <[float](#)>
  - `y` <[float](#)>
- `timeout` <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- `trial` <[bool](#)> 设置后，该方法只执行 [actionability](#) 检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的。<#>
- returns: <[NoneType](#)> <#>

## [# locator.element\\_handle\(\\*\\*kwargs\) #](#)

---

- `timeout` <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- returns: <[ElementHandle](#)> <#>

解析给定的定位器到第一个匹配的DOM元素。如果没有匹配查询的元素可见，等待它们直到给定的超时。如果多个元素匹配该选择器，则抛出。

## [# locator.element\\_handles\(\) #](#)

---

- returns: <[List](#)[[ElementHandle](#)]> <#>

解析所有匹配的DOM元素。

## [# locator.evaluate\(expression, \\*\\*kwargs\) #](#)

---

- `expression` <[str](#)> 要在浏览器上下文中计算的 `JavaScript` 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值。<#>
- `arg` <[EvaluationArgument](#)> 传递给 `expression` 的可选参数。<#>
- `timeout` <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- returns: <[Serializable](#)> <#>

返回表达式的返回值。

此方法将此句柄作为第一个参数传递给表达式。

如果表达式返回 [Promise](#)，则 `handle.evaluate` 将等待promise解析并返回其值。

例子：

- Sync

```
tweets = page.locator(".tweet .retweets")
assert tweets.evaluate("node ⇒ node.innerText") = "10
retweets"
```

- Async

```
tweets = page.locator(".tweet .retweets")
assert await tweets.evaluate("node ⇒ node.innerText") = "10
retweets"
```

## `# locator.evaluate_all(expression, **kwargs)` <#>

---

- `expression` <[str](#)> 要在浏览器上下文中计算的 `JavaScript` 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值。<#>
- `arg` <[EvaluationArgument](#)> 传递给 `expression` 的可选参数。<#>
- returns: <[Serializable](#)> <#>

该方法查找与指定定位器匹配的所有元素，并将匹配元素的数组作为第一个参数传递给表达式。返回表达式调用的结果。

如果表达式返回 [Promise](#)，则 [locator.evaluate\\_all\(expression, \\*\\*kwargs\)](#) 将等待 promise 解析并返回其值。

例子：

- Sync

```
elements = page.locator("div")
div_counts = elements("(divs, min) => divs.length >= min", 10)
```

- Async

```
elements = page.locator("div")
div_counts = await elements("(divs, min) => divs.length >= min", 10)
```

## # [locator.evaluate\\_handle\(expression, \\*\\*kwargs\)](#) #

---

- [expression](#) <[str](#)> 要在浏览器上下文中计算的 [JavaScript](#) 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值。<#>
- [arg](#) <[EvaluationArgument](#)> 传递给 [expression](#) 的可选参数。<#>
- [timeout](#) <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 [0](#) 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- returns: <[JSHandle](#)> <#>

返回表达式的返回值 as a [JSHandle](#) .

此方法将此句柄作为第一个参数传递给表达式。

[locator.evaluate\(expression, \\*\\*kwargs\)](#) and [locator.evaluate\\_handle\(expression, \\*\\*kwargs\)](#) 之间唯一不同的是 [locator.evaluate\\_handle\(expression, \\*\\*kwargs\)](#) 返回 [JSHandle](#) .

如果函数传递给 `locator.evaluate_handle(expression, **kwargs)` 返回一个 `Promise`, 则 `locator.evaluate_handle(expression, **kwargs)` 将等待promise解析并返回其值.

详情查看 `page.evaluate_handle(expression, **kwargs)`.

## # locator.fill(value, \*\*kwargs) #

---

- `value` <str> Value to set for the `\<input>`, `\<textarea>` or `[contenteditable]` element. #
- `force` <bool> 是否绕过 `actionability` 检查。默认值为 `false`. #
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生, 并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项, 比如导航到不可访问的页面。默认值为 `false`. #
- `timeout` <float> 最大时间, 单位为毫秒, 默认为30秒, 通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改. #
- returns: <NoneType> #

这个方法等待 `可操作性` 检查, 聚焦元素, 填充它, 并在填充后触发一个输入事件。请注意, 您可以传递一个空字符串来清除输入字段。

如果目标元素不是 `\<input>`, `\<textarea>` or `[contenteditable]`, 此方法将抛出一个错误。但是, 如果该元素位于 `\<label>` 元素中, 且该元素具有关联控件, 则该控件将被填充。

要发送细粒度的键盘事件, 请使用 `locator.type(text, **kwargs)`.

## # locator.first #

---

- returns: <Locator> #

返回第一个匹配元素的定位符。

## # locator.focus(\*\*kwargs) #

---

- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。<#>
- returns: `<NoneType>` <#>

Calls [focus](#) on the element.

## # `locator.frame_locator(selector)` <#>

- `selector` `<str>` 解析DOM元素时使用的选择器。有关更多细节，请参阅 [working with selectors](#) <#>
- returns: `<FrameLocator>` <#>

当使用 `iframe` 时，你可以创建一个frame 定位器，它将进入iframe 并允许选择该 `iframe`中的元素：

- Sync

```
locator = page.frame_locator("iframe").locator("text=Submit")
locator.click()
```

- Async

```
locator = page.frame_locator("iframe").locator("text=Submit")
await locator.click()
```

## # `locator.get_attribute(name, **kwargs)` <#>

- `name` `<str>` 属性名。<#>
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。<#>
- returns: `<NoneType|str>` <#>

返回元素属性值。

## # locator.highlight()#

---

- returns: <NoneType>#

在屏幕上突出显示相应的元素。这对调试很有用，不要提交使用 [locator.highlight\(\)](#) 的代码。

## # locator.hover(\*\*kwargs)#

---

- **force** <bool> 是否绕过 [actionability](#) 检查。默认值为 **false**。#
- **modifiers** <List["Alt"|"Control"|"Meta"|"Shift"]> modifiers 按键要按。确保在操作期间只按下这些修饰符，然后恢复当前的修饰符。如果未指定，则使用当前按下的修饰符。#
- **position** <Dict> 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点。#
  - **x** <float>
  - **y** <float>
- **timeout** <float> 最大时间，单位为毫秒，默认为30秒，通过 **0** 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。#
- **trial** <bool> 设置后，该方法只执行 [actionability](#) 检查，并跳过操作。默认值为 **false**。在元素准备好时再执行动作是很有用的。#
- returns: <NoneType>#

该方法通过执行以下步骤悬停在元素上：

1. 等待元素的 [可操作性](#) 检查，除非设置了强制选项。
2. 如果需要，将元素滚动到视图中。
3. 使鼠标停在元素中心或指定位置上。
4. 等待发起的导航成功或失败，除非设置了 **noWaitAfter** 选项。

如果元素在动作期间的任何时刻与DOM分离，此方法将抛出。

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

## # locator.inner\_html(\*\*kwargs)#

---

- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: <str>#

Returns the `element.innerHTML`.

## # locator.inner\_text(\*\*kwargs)#

---

- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: <str>#

Returns the `element.innerText`.

## # locator.input\_value(\*\*kwargs)#

---

- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: <str>#

返回输入 `\<input>` or `\<textarea>` or `\<select>` 元素的值, 排除非输入元素.

## # locator.is\_checked(\*\*kwargs)#

---

- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用


[browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#

- returns: [<bool>](#) #

返回元素是否被选中。如果元素不是复选框或单选输入则排除。

## # locator.is\_disabled(\*\*kwargs) #


---

- **timeout** [<float>](#) 最大时间，单位为毫秒，默认为30秒，通过  禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: [<bool>](#) #

返回该元素是否被禁用，与启用 [enabled](#) 相反。

## # locator.is\_editable(\*\*kwargs) #


---

- **timeout** [<float>](#) 最大时间，单位为毫秒，默认为30秒，通过  禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: [<bool>](#) #

返回元素是否可编辑 [editable](#)。

## # locator.is\_enabled(\*\*kwargs) #

---

- **timeout** [<float>](#) 最大时间，单位为毫秒，默认为30秒，通过  禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: [<bool>](#) #

返回元素是否被启用 [enabled](#)。



## # locator.is\_hidden(\*\*kwargs)#

---

- `timeout` <float> **DEPRECATED** This option is ignored. `locator.is_hidden(**kwargs)` does not wait for the element to become hidden and returns immediately. #
- returns: <bool> #

返回元素是否隐藏，与可见 [visible](#) 相反。

## # locator.is\_visible(\*\*kwargs)#

---

- `timeout` <float> **DEPRECATED** This option is ignored. `locator.is_visible(**kwargs)` does not wait for the element to become visible and returns immediately. #
- returns: <bool> #

返回元素是否可见 [visible](#)。

## # locator.last#

---

- returns: <Locator> #

返回最后一个匹配元素的定位符。

## # locator.locator(selector, \*\*kwargs)#

---

- `selector` <str> 解析DOM元素时使用的选择器。有关更多细节，请参阅 [working with selectors](#) . #
- `has` <Locator> 对selector选中的元素进行再次匹配，匹配目标中的子元素，例如：匹配子元素的 `text=Playwright` 的元素 `\<article>\<div>Playwright\</div>\</article>` . #

请注意，外部和内部定位器必须属于同一个 frame. 内部定位器不能包含 [FrameLocator](#)。

- `has_text` `<str|Pattern>` 匹配包含指定文本的元素，可能在子元素或后代元素中,例如: `"Playwright"` 匹配 `\<article>\<div>Playwright\</div>\</article>`.#
- returns: `<Locator>` .#

该方法在 `Locator` 的子树中查找与指定选择器匹配的元素.

## # `locator.nth(index)` .#

---

- `index` `<int>` .#
- returns: `<Locator>` .#

返回第n个匹配元素的定位符.

## # `locator.page` .#

---

- returns: `<Page>` .#

这个定位器所属的页面.

## # `locator.press(key, **kwargs)` .#

---

- `key` `<str>` 要按下的 `键名` 或要生成的字符, 如 `ArrowLeft` 或 `'a'`.#
- `delay` `<float>` `keydown` 和 `keyup` 之间的等待时间, 单位是毫秒。默认为0.#
- `no_wait_after` `<bool>` 启动导航的操作正在等待这些导航发生, 并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项, 比如导航到不可访问的页面。默认值为 `false`.#
- `timeout` `<float>` 最大时间, 单位为毫秒, 默认为30秒, 通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改.#
- returns: `<NoneType>` .#

聚焦元素, 然后使用 `keyboard.down(key)` and `keyboard.up(key)`.

key可以指定想要的 [keyboardEvent.key](#) 或是单个字符生成的文本,这里可以找到键值的超集。键的例子如下:

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

还支持以下快捷键: `Shift`, `Control`, `Alt`, `Meta`, `ShiftLeft`.

按住 `Shift` 键将输入与大写键对应的文本.

如果 `key` 是单个字符, 它是区分大小写的, 因此值 `a` 和 `A` 将生成不同的文本.

也支持快捷键, 如键:“Control+o”或键:“Control+Shift+T”。当用修饰符指定时, 修饰符被按下并被保持, 而随后的键被按下.

## # locator.screenshot(\*\*kwargs) #

---

- `animations` <"disabled"> 当设置为 `"disabled"` 时, 停止CSS动画, CSS转换和Web动画。动画根据其持续时间得到不同的处理: #
  - 有限动画是快进到完成, 所以他们会触发 `transitionend` 事件.
  - 无限动画被取消到初始状态, 然后在屏幕截图后播放.
- `mask` <[List](#)[[Locator](#)]> 指定在截屏时应该被屏蔽的定位器。被屏蔽的元素将被一个粉红色的框覆盖#FF00FF, 完全覆盖该元素. #
- `omit_background` <[bool](#)> 隐藏默认的白色背景, 并允许透明捕捉屏幕截图。不适用于 `jpeg` 图像。默认值为 `false`. #
- `path` <[Union](#)[[str](#), [pathlib.Path](#)]> 保存镜像的文件路径, 屏幕截图类型将从文件扩展名推断。如果 `path` 是一个相对路径, 那么它是相对于当前工作目录解析的。如果没有提供路径, 映像将不会被保存到磁盘. #
- `quality` <[int](#)> 图像的质量, 在0-100之间。不适用于png图像. #
- `timeout` <[float](#)> 最大时间, 单位为毫秒, 默认为30秒, 通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改. #
- `type` <"png"|"jpeg">指定截图类型, 默认为 `png`. #
- returns: <[bytes](#)> #

返回带有捕获的截图的缓冲区。

这个方法等待 [可操作性](#) 检查，然后在截屏之前将元素滚动到视图中。如果元素与DOM分离，该方法将抛出一个错误。

#

## locator.scroll\_into\_view\_if\_needed(\*\*kwargs)#

---

- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- returns: <NoneType> <#>

这个方法等待 [可操作性](#) 检查，然后尝试滚动元素到视图中，除非它是完全可见的，由 [IntersectionObserver](#) 的比率定义。

## # locator.select\_option(\*\*kwargs)#

---

- `force` <bool> 是否绕过 [actionability](#) 检查。默认值为 `false` <#>
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` <#>
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- `element` <ElementHandle|List[ElementHandle]> 要选择的选项。<#>
- `index` <int|List[int]> 按索引进行选择的选项。可选的。<#>
- `value` <str|List[str]> 按值选择的选项。如果 `\<select>` 具有多个属性，则选择所有给定的选项，否则只选择与传递的选项之一匹配的[第一个选项](#)。可选的。<#>
- `label` <str|List[str]> 按标签进行选择的选项。如果 `\<select>` 具有多个属性，则选择所有给定的选项，否则只选择与传递的选项之一匹配的[第一个选项](#)。可选的。<#>

- returns: `<List[str]>#`

这个方法等待[可操作性](#)检查，直到所有指定的选项都出现在 `<select>` 元素中，然后选择这些选项。

如果目标元素不是 `<select>` 元素，此方法将抛出一个错误。但是，如果该元素位于 `<label>` 元素中，且该元素具有关联控件，则将使用该控件。

返回已成功选择的选项值的数组。

一旦选择了所有提供的选项，就触发一个更改和输入事件。

- Sync

```
# single selection matching the value
element.select_option("blue")
# single selection matching both the label
element.select_option(label="blue")
# multiple selection
element.select_option(value=["red", "green", "blue"])
```

- Async

```
# single selection matching the value
await element.select_option("blue")
# single selection matching the label
await element.select_option(label="blue")
# multiple selection
await element.select_option(value=["red", "green", "blue"])
```

```
# sync
```

```
# single selection matching the value
element.select_option("blue")
# single selection matching both the value and the label
element.select_option(label="blue")
# multiple selection
element.select_option("red", "green", "blue")
# multiple selection for blue, red and second option
element.select_option(value="blue", { index: 2 }, "red")
```

## # locator.select\_text(\*\*kwargs) #

---

- `force` <bool> 是否绕过 [actionability](#) 检查。默认值为 `false` .#
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: <NoneType> .#

这个方法等待 [可操作性](#) 检查, 然后关注元素并选择其所有文本内容

## # locator.set\_checked(checked, \*\*kwargs) #

---

- `checked` <bool> 是否选中或不选中复选框.#
- `force` <bool> 是否绕过 [actionability](#) 检查。默认值为 `false` .#
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- `position` <Dict> 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点.#
  - `x` <float>
  - `y` <float>
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- `trial` <bool> 设置后，该方法只执行 [actionability](#) 检查，并跳过操作。默认值为 `false` 。在元素准备好时再执行动作是很有用的.#
- returns: <NoneType> .#

这个方法通过执行以下步骤检查或取消检查一个元素：

1. 确保匹配的元素是一个复选框或单选输入。如果不是，则排除此方法。
2. 如果元素已经具有正确的选中状态，则该方法立即返回。
3. 等待匹配元素的 [actionability](#) 检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作。
4. 如果需要，将元素滚动到视图中。

5. 使用 [page.mouse](#) 单击元素的中心。
6. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项。
7. 确保元素现在被选中或取消选中。如果不是，则抛出此方法。

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

## # locator.set\_input\_files(files, \*\*kwargs) #

---

- `files` <[Union](#)[[str](#), [pathlib.Path](#)]|[List](#)[[Union](#)[[str](#), [pathlib.Path](#)]]|[Dict](#)|[List](#)[[Dict](#)]> #
  - `name` <[str](#)> 文件名
  - `mimeType` <[str](#)> 文件类型
  - `buffer` <[bytes](#)> 文件内容
- `no_wait_after` <[bool](#)> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- `timeout` <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- `returns`: <[NoneType](#)> #

这个方法期望元素指向一个输入元素 [input element](#)。

将文件输入的值设置为这些文件路径或文件。如果某些 `filepath` 是相对路径，那么它们将相对于当前工作目录进行解析。对于空数组，清除选定的文件。

## # locator.tap(\*\*kwargs) #

---

- `force` <[bool](#)> 是否绕过 [actionability](#) 检查。默认值为 `false` .#
- `modifiers` <[List](#)["Alt"]|["Control"]|["Meta"]|["Shift"]> modifiers 按键要按。确保在操作期间只按下这些修饰符，然后恢复当前的修饰符。如果未指定，则使用当前按下的修饰符.#

- `no_wait_after` `<bool>` 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false`。<#>
- `position` `<Dict>` 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点。<#>
  - `x` `<float>`
  - `y` `<float>`
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- `trial` `<bool>` 设置后，该方法只执行 [actionability](#) 检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的。<#>
- returns: `<NoneType>` <#>

这个方法通过执行以下步骤点击元素：

1. 等待元素的 [可操作性](#) 检查，除非设置了强制选项。
2. 如果需要，将元素滚动到视图中。
3. 点击页面中心或指定位置。
4. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项。

如果元素在动作期间的任何时刻与DOM分离，此方法将抛出。

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

#### NOTE

`element.tap()` 要求浏览器上下文的 `hasTouch` 选项设置为 `True`。

## # locator.text\_content(\*\*kwargs)<#>

- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- returns: `<NoneType|str>` <#>



Returns the `node.textContent`.

## # locator.type(text, \*\*kwargs)#

---

- `text` <str> 要输入到焦点元素中的文本.#
- `delay` <float> 按键之间的等待时间，单位是毫秒。默认为0.#
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false`.#
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: <NoneType>#

聚焦元素，然后为文本中的每个字符发送 `keydown`，`keypress`/`input`，and `keyup` 事件。

要按一个特殊的键，如 `Control` or `ArrowDown`，使用 [locator.press\(key, \\*\\*kwargs\)](#)。

- Sync

```
element.type("hello") # types instantly
element.type("world", delay=100) # types slower, like a user
```

- Async

```
await element.type("hello") # types instantly
await element.type("world", delay=100) # types slower, like a user
```

一个在文本框中输入然后提交表单的例子：

- Sync

```
element = page.locator("input")
element.type("some text")
element.press("Enter")
```

- Async

```
element = page.locator("input")
await element.type("some text")
await element.press("Enter")
```

## # locator.uncheck(\*\*kwargs) #

---

- `force` <bool> 是否绕过 [actionability](#) 检查。默认值为 `false` .#
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- `position` <Dict> 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点 .#
  - `x` <float>
  - `y` <float>
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改 .#
- `trial` <bool> 设置后，该方法只执行 [actionability](#) 检查，并跳过操作。默认值为 `false` 。在元素准备好时再执行动作是很有用的 .#
- returns: <NoneType> #

这个方法通过执行以下步骤来检查元素：

1. 确保元素是一个复选框或单选输入。如果不是，则抛出此方法。如果元素已被选中，则此方法立即返回。
2. 等待元素的 [可操作性](#) 检查，除非设置了强制选项。
3. 如果需要，将元素滚动到视图中。
4. 使用 [page.mouse](#) 单击元素的中心。
5. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项。
6. 确保元素现在是未选中的。如果不是，则排除此方法。

如果元素在动作期间的任何时刻与DOM分离，此方法将抛出。

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

## # locator.wait\_for(\*\*kwargs)#

---

- **state** <"attached"|"detached"|"visible"|"hidden"> 默认为 **"visible"**。可以是：
  - **'attached'** - 等待元素出现在DOM中。
  - **'detached'** - 等待元素在DOM中不存在。
  - **'visible'** - 等待元素有非空的边界框 且 没有 **visibility:hidden**。注意，没有任何内容或带有 **display:none** 的元素有一个空的边界框，因此不被认为是可见的。
  - **'hidden'** - 等待元素从DOM中分离出来，或有一个空的边界框或 **visibility:hidden**。这与 **"visible"** 选项相反。
- **timeout** <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 **0** 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- returns: <[NoneType](#)> <#>

当 locator 指定的元素满足状态选项时返回。

如果目标元素已经满足条件，则该方法立即返回。否则，将等待 **超时** 毫秒，直到满足条件。

- Sync

```
order_sent = page.locator("#order-sent")
order_sent.wait_for()
```

- Async

```
order_sent = page.locator("#order-sent")
await order_sent.wait_for()
```

## Mouse

Mouse类操作的是相对于视口左上角的CSS 像素.

每个 [页面](#) 对象都有自己的鼠标, 可以通过 [page.mouse](#).

- Sync

```
# 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()
```

- Async

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

- [mouse.click\(x, y, \\*\\*kwargs\)](#)
- [mouse.dblclick\(x, y, \\*\\*kwargs\)](#)
- [mouse.down\(\\*\\*kwargs\)](#)
- [mouse.move\(x, y, \\*\\*kwargs\)](#)
- [mouse.up\(\\*\\*kwargs\)](#)
- [mouse.wheel\(delta\\_x, delta\\_y\)](#)

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

---

- `x` <float>#
- `y` <float>#
- `button` <"left"|"right"|"middle"> 默认左 `left`.#
- `click_count` <int> 默认为1,详情查看 [UIEvent.detail](#).#
- `delay` <float> `mousedown` 和 `mouseup` 之间的等待时间, 单位是毫秒。默认为0.#
- returns: <NoneType>#

[mouse.move\(x, y, \\*\\*kwargs\)](#), [mouse.down\(\\*\\*kwargs\)](#), [mouse.up\(\\*\\*kwargs\)](#) 的三合一快捷键

## # mouse.dblick(x, y, \*\*kwargs)#

---

- `x` <float>#
- `y` <float>#
- `button` <"left"|"right"|"middle"> 默认左 `left`.#
- `delay` <float> `mousedown` 和 `mouseup` 之间的等待时间, 单位是毫秒。默认为0.#
- returns: <NoneType>#

[mouse.move\(x, y, \\*\\*kwargs\)](#), [mouse.down\(\\*\\*kwargs\)](#), [mouse.up\(\\*\\*kwargs\)](#), [mouse.down\(\\*\\*kwargs\)](#) and [mouse.up\(\\*\\*kwargs\)](#) 的快捷键

## # mouse.down(\*\*kwargs)#

---

- `button` <"left"|"right"|"middle"> 默认左 `left`.#
- `click_count` <int> 默认为1,详情查看 [UIEvent.detail](#).#
- returns: <NoneType>#

发送一个 `mousedown` 事件.

## # mouse.move(x, y, \*\*kwargs)#

---

- `x` <float>#

- `y` <[float](#)>#
- `steps` <[int](#)> 默认为1. 发送 `mousemove` 事件.#
- returns: <[NoneType](#)>#

发送一个 `mousemove` 事件.

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

---

- `button` <"left"|"right"|"middle"> 默认左 `left`.#
- `click_count` <[int](#)> 默认为1, 详情查看 [UIEvent.detail](#).#
- returns: <[NoneType](#)>#

发送一个 `mouseup` 事件.

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

---

- `delta_x` <[float](#)> 像素水平滚动.#
- `delta_y` <[float](#)> 垂直滚动的像素.#
- returns: <[NoneType](#)>#

发送一个 `wheel` 事件.

### NOTE

如果 *Wheel* 事件未被处理, 则可能导致滚动, 并且此方法在返回之前不会等待滚动完成

## Page

- extends: [EventEmitter](#)

页面提供了与 [Browser](#) 中的单个标签交互的方法, 或在Chromium的 [extension background page](#)。一个 [Browser](#) 实例可能有多个 [Page](#) 实例

这个例子创建了一个页面, 将它导航到一个URL, 然后保存了一个截图:

- Sync

```
from playwright.sync_api import sync_playwright

def run(playwright):
    webkit = playwright.webkit
    browser = webkit.launch()
    context = browser.new_context()
    page = context.new_page()
    page.goto("https://example.com")
    page.screenshot(path="screenshot.png")
    browser.close()

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

- Async

```
import asyncio
from playwright.async_api import async_playwright

async def run(playwright):
    webkit = playwright.webkit
    browser = await webkit.launch()
    context = await browser.new_context()
    page = await context.new_page()
    await page.goto("https://example.com")
    await page.screenshot(path="screenshot.png")
    await browser.close()

async def main():
    async with async_playwright() as playwright:
        await run(playwright)

asyncio.run(main())
```

Page类会发出各种各样的事件(如下所述), 这些事件可以使用Node的任何本地 `EventEmitter` 方法来处理, 比如 `on`、`once` 或 `removeListener`。

下面的例子记录了单个页面加载事件的消息:

```
page.once("load", lambda: print("page loaded!"))
```

要取消订阅事件，请使用 `removeListener` 方法：

```
def log_request(intercepted_request):  
    print("a request was made:", intercepted_request.url)  
    page.on("request", log_request)  
    # sometime later...  
    page.remove_listener("request", log_request)
```

- [page.on\("close"\)](#)
- [page.on\("console"\)](#)
- [page.on\("crash"\)](#)
- [page.on\("dialog"\)](#)
- [page.on\("domcontentloaded"\)](#)
- [page.on\("download"\)](#)
- [page.on\("filechooser"\)](#)
- [page.on\("frameattached"\)](#)
- [page.on\("framedetached"\)](#)
- [page.on\("framenavigated"\)](#)
- [page.on\("load"\)](#)
- [page.on\("pageerror"\)](#)
- [page.on\("popup"\)](#)
- [page.on\("request"\)](#)
- [page.on\("requestfailed"\)](#)
- [page.on\("requestfinished"\)](#)
- [page.on\("response"\)](#)
- [page.on\("websocket"\)](#)
- [page.on\("worker"\)](#)
- [page.add\\_init\\_script\(\\*\\*kwargs\)](#)
- [page.add\\_script\\_tag\(\\*\\*kwargs\)](#)
- [page.add\\_style\\_tag\(\\*\\*kwargs\)](#)
- [page.bring\\_to\\_front\(\)](#)
- [page.check\(selector, \\*\\*kwargs\)](#)
- [page.click\(selector, \\*\\*kwargs\)](#)
- [page.close\(\\*\\*kwargs\)](#)
- [page.content\(\)](#)
- [page.context](#)
- [page.dblclick\(selector, \\*\\*kwargs\)](#)
- [page.dispatch\\_event\(selector, type, \\*\\*kwargs\)](#)
- [page.drag\\_and\\_drop\(source, target, \\*\\*kwargs\)](#)
- [page.emulate\\_media\(\\*\\*kwargs\)](#)
- [page.eval\\_on\\_selector\(selector, expression, \\*\\*kwargs\)](#)
- [page.eval\\_on\\_selector\\_all\(selector, expression, \\*\\*kwargs\)](#)



- [page.evaluate\(expression, \\*\\*kwargs\)](#)
- [page.evaluate\\_handle\(expression, \\*\\*kwargs\)](#)
- [page.expect\\_console\\_message\(\\*\\*kwargs\)](#)
- [page.expect\\_download\(\\*\\*kwargs\)](#)
- [page.expect\\_event\(event, \\*\\*kwargs\)](#)
- [page.expect\\_file\\_chooser\(\\*\\*kwargs\)](#)
- [page.expect\\_navigation\(\\*\\*kwargs\)](#)
- [page.expect\\_popup\(\\*\\*kwargs\)](#)
- [page.expect\\_request\(url\\_or\\_predicate, \\*\\*kwargs\)](#)
- [page.expect\\_request\\_finished\(\\*\\*kwargs\)](#)
- [page.expect\\_response\(url\\_or\\_predicate, \\*\\*kwargs\)](#)
- [page.expect\\_websocket\(\\*\\*kwargs\)](#)
- [page.expect\\_worker\(\\*\\*kwargs\)](#)
- [page.expose\\_binding\(name, callback, \\*\\*kwargs\)](#)
- [page.expose\\_function\(name, callback\)](#)
- [page.fill\(selector, value, \\*\\*kwargs\)](#)
- [page.focus\(selector, \\*\\*kwargs\)](#)
- [page.frame\(\\*\\*kwargs\)](#)
- [page.frame\\_locator\(selector\)](#)
- [page.frames](#)
- [page.get\\_attribute\(selector, name, \\*\\*kwargs\)](#)
- [page.go\\_back\(\\*\\*kwargs\)](#)
- [page.go\\_forward\(\\*\\*kwargs\)](#)
- [page.goto\(url, \\*\\*kwargs\)](#)
- [page.hover\(selector, \\*\\*kwargs\)](#)
- [page.inner\\_html\(selector, \\*\\*kwargs\)](#)
- [page.inner\\_text\(selector, \\*\\*kwargs\)](#)
- [page.input\\_value\(selector, \\*\\*kwargs\)](#)
- [page.is\\_checked\(selector, \\*\\*kwargs\)](#)
- [page.is\\_closed\(\)](#)
- [page.is\\_disabled\(selector, \\*\\*kwargs\)](#)
- [page.is\\_editable\(selector, \\*\\*kwargs\)](#)
- [page.is\\_enabled\(selector, \\*\\*kwargs\)](#)
- [page.is\\_hidden\(selector, \\*\\*kwargs\)](#)
- [page.is\\_visible\(selector, \\*\\*kwargs\)](#)
- [page.locator\(selector, \\*\\*kwargs\)](#)
- [page.main\\_frame](#)
- [page.opener\(\)](#)
- [page.pause\(\)](#)
- [page.pdf\(\\*\\*kwargs\)](#)
- [page.press\(selector, key, \\*\\*kwargs\)](#)
- [page.query\\_selector\(selector, \\*\\*kwargs\)](#)
- [page.query\\_selector\\_all\(selector\)](#)

- [page.reload\(\\*\\*kwargs\)](#)
- [page.route\(url, handler, \\*\\*kwargs\)](#)
- [page.screenshot\(\\*\\*kwargs\)](#)
- [page.select\\_option\(selector, \\*\\*kwargs\)](#)
- [page.set\\_checked\(selector, checked, \\*\\*kwargs\)](#)
- [page.set\\_content\(html, \\*\\*kwargs\)](#)
- [page.set\\_default\\_navigation\\_timeout\(timeout\)](#)
- [page.set\\_default\\_timeout\(timeout\)](#)
- [page.set\\_extra\\_http\\_headers\(headers\)](#)
- [page.set\\_input\\_files\(selector, files, \\*\\*kwargs\)](#)
- [page.set\\_viewport\\_size\(viewport\\_size\)](#)
- [page.tap\(selector, \\*\\*kwargs\)](#)
- [page.text\\_content\(selector, \\*\\*kwargs\)](#)
- [page.title\(\)](#)
- [page.type\(selector, text, \\*\\*kwargs\)](#)
- [page.uncheck\(selector, \\*\\*kwargs\)](#)
- [page.unroute\(url, \\*\\*kwargs\)](#)
- [page.url](#)
- [page.video](#)
- [page.viewport\\_size](#)
- [page.wait\\_for\\_event\(event, \\*\\*kwargs\)](#)
- [page.wait\\_for\\_function\(expression, \\*\\*kwargs\)](#)
- [page.wait\\_for\\_load\\_state\(\\*\\*kwargs\)](#)
- [page.wait\\_for\\_selector\(selector, \\*\\*kwargs\)](#)
- [page.wait\\_for\\_timeout\(timeout\)](#)
- [page.wait\\_for\\_url\(url, \\*\\*kwargs\)](#)
- [page.workers](#)
- [page.accessibility](#)
- [page.keyboard](#)
- [page.mouse](#)
- [page.request](#)
- [page.touchscreen](#)

## # [page.on\("close"\)](#) #

---

- type: <[Page](#)>

页面关闭时触发.

## # page.on("console")#

---

- type: <[ConsoleMessage](#)>

当页面中的JavaScript调用控制台API方法之一时触发，例如 `console.log` 或 `console.dir`。当页面抛出错误或警告时也会触发。

传递到 `console.log` 的参数显示为事件处理程序上的参数。

一个处理控制台事件的例子：

- Sync

```
def print_args(msg):
    for arg in msg.args:
        print(arg.json_value())

page.on("console", print_args)
page.evaluate("console.log('hello', 5, {foo: 'bar'})")
```

- Async

```
async def print_args(msg):
    values = []
    for arg in msg.args:
        values.append(await arg.json_value())
    print(values)

page.on("console", print_args)
await page.evaluate("console.log('hello', 5, {foo: 'bar'})")
```

## # page.on("crash")#

---

- type: <[Page](#)>

当页面崩溃时触发。如果试图分配过多的内存，浏览器页面可能会崩溃。当页面崩溃时，将抛出正在进行的和后续的操作。

处理崩溃最常见的方法是捕获一个异常：

- Sync

```
try:
    # crash might happen during a click.
    page.click("button")
    # or while waiting for an event.
    page.wait_for_event("popup")
except Error as e:
    # when the page crashes, exception message contains
    "crash".
```

- Async

```
try:
    # crash might happen during a click.
    await page.click("button")
    # or while waiting for an event.
    await page.wait_for_event("popup")
except Error as e:
    # when the page crashes, exception message contains
    "crash".
```

## # page.on("dialog")#

---

- type: <[Dialog](#)>

当JavaScript对话框出现时触发，例如 `alert`，`prompt`，`confirm` or `beforeunload`。必须要么 `dialog.accept(**kwargs)` 要么 `dialog.dismiss()` 对话框-否则页面会冻结等待对话框，像click这样的动作将永远不会结束。

### NOTE

当没有 `page.on("dialog")` 监听器时，所有的对话框都会被自动取消。

## # page.on("domcontentloaded")#

---

- type: [<Page>](#)

当发送 JavaScript `DOMContentLoaded` 事件时触发。

## # `page.on("download")` #

---

- type: [<Download>](#)

附件下载开始时发出。用户可以通过传递的 [Download](#) 实例访问已下载内容的基本文件操作。

## # `page.on("filechooser")` #

---

- type: [<FileChooser>](#)

当一个文件选择器应该出现时发出，例如在单击 `<input type=file>`。Playwright 可以通过使用 [file\\_chooser.set\\_files\(files, \\*\\*kwargs\)](#) 设置输入文件来响应它。

```
page.on("filechooser", lambda file_chooser:
    file_chooser.set_files("/tmp/myfile.pdf"))
```

## # `page.on("frameattached")` #

---

- type: [<Frame>](#)

附加 frame 时触发。

## # `page.on("framedetached")` #

---

- type: [<Frame>](#)

分离 frame 时触发。

## # page.on("framenavigated")#

---

- type: <[Frame](#)>

当一个frame 导航到一个新url时触发.

## # page.on("load")#

---

- type: <[Page](#)>

当分派JavaScript加载 [load](#) 事件时触发

## # page.on("pageerror")#

---

- type: <[Error](#)>

当页面内发生未捕获的异常时触发.

## # page.on("popup")#

---

- type: <[Page](#)>

当页面打开新选项卡或窗口时触发。这个事件是在 [browser\\_context.on\("page"\)](#) 之外触发的，但只针对与该页相关的弹出窗口。

该页面可用的最早时刻是当它已导航到初始url。例如，当用 `window.open('http://example.com')` 打开一个弹出窗口时，这个事件将在网络请求 "<http://example.com>" 完成并在弹出窗口中开始加载响应时触发。

- Sync

```
with page.expect_event("popup") as page_info:
    page.evaluate("window.open('https://example.com')")
    popup = page_info.value
    print(popup.evaluate("location.href"))
```

- Async

```
async with page.expect_event("popup") as page_info:
    page.evaluate("window.open('https://example.com')")
popup = await page_info.value
print(await popup.evaluate("location.href"))
```

#### NOTE

使用 [`page.wait\_for\_load\_state\(\*\*kwargs\)`](#) 等待页面到达特定的状态(在大多数情况下不需要它).

## # `page.on("request")` #

---

- type: [`<Request>`](#)

当页面发出请求时触发。 [`request`](#) 对象是只读的。要拦截和修改请求，请参阅页面 [`page.route\(url, handler, \*\*kwargs\)`](#) or [`browser\_context.route\(url, handler, \*\*kwargs\)`](#).

## # `page.on("requestfailed")` #

---

- type: [`<Request>`](#)

当请求失败时触发，例如超时.

#### NOTE

`HTTP` 错误响应，如 `404` 或 `503`，从 `HTTP` 的角度来看，仍然是成功的响应，因此请求将以 [`page.on\("requestfinished"\)`](#) 事件完成，而不是 [`page.on\("requestfailed"\)`](#). 只有当客户端无法从服务器获得 `HTTP` 响应时，请求才会被认为失败，例如由于网络错误 `net::ERR_FAILED`.

## # `page.on("requestfinished")` #

---

- type: [`<Request>`](#)

在下载响应体后，请求成功完成时触发。对于一个成功的响应，事件序列是 `request`，`response` and `requestfinished`。

## # `page.on("response")` #

---

- type: <[Response](#)>

当收到请求的 [Response](#) 状态和报头时触发。对于一个成功的响应，事件序列是 `request`，`response` and `requestfinished`。

## # `page.on("websocket")` #

---

- type: <[WebSocket](#)>

[WebSocket](#) 请求发送时触发。

## # `page.on("worker")` #

---

- type: <[Worker](#)>

当页面生成一个专用的 [WebWorker](#) 时触发。

## # `page.add_init_script(**kwargs)` #

---

- `path` <[Union](#)[[str](#), [pathlib.Path](#)]> JavaScript 文件的路径. 如果 `path` 是一个相对路径，那么它是相对于当前工作目录解析的。可选的. #
- `script` <[str](#)> 在浏览器上下文中所有页面中计算的脚本。可选的. #
- returns: <[NoneType](#)> #

添加一个脚本，该脚本将在以下场景之一中进行评估：

- 当页面被导航时。
- 当附加或导航子框架时。在这种情况下，脚本将在新附加的框架的上下文中计算。



在创建文档之后，但在运行文档的任何脚本之前，对脚本进行计算。这对于修改JavaScript环境是很有用的，例如 `Math.random`。

一个重写 `Math.random` 的例子：

```
// preload.js
Math.random = () => 42;
```

- Sync

```
# in your playwright script, assuming the preload.js file is
in same directory
page.add_init_script(path="./preload.js")
```

- Async

```
# in your playwright script, assuming the preload.js file is
in same directory
await page.add_init_script(path="./preload.js")
```

#### NOTE

通过 [`browser\_context.add\_init\_script\(\*\*kwargs\)`](#) and [`page.add\_init\_script\(\*\*kwargs\)`](#) 安装的多个脚本的计算顺序没有定义。

## # `page.add_script_tag(**kwargs)` #

- `content` <str> 要注入帧的原始JavaScript内容。#
- `path` <Union[str, pathlib.Path]> 要注入帧的JavaScript文件的路径，如果 `path` 是一个相对路径，那么它是相对于当前工作目录解析的。#
- `type` <str> JS脚本类型。使用“module”来加载一个JavaScript ES6模块。详情请参阅 [`script`](#)。#
- `url` <str> 要添加的脚本的url。#
- returns: <ElementHandle> #

添加一个 `<script>` 标签到页面所需的url或内容。当脚本的onload触发或脚本内容被注入帧时，返回添加的标签。

主frame [`frame.add\_script\_tag\(\*\*kwargs\)`](#) 的快捷方式。

## # page.add\_style\_tag(\*\*kwargs)#

---

- `content` <str> 原始的CSS内容注入到帧.#
- `path` <Union[str, pathlib.Path]> 要注入帧的CSS文件的路径, 如果path是一个相对路径, 那么它是相对于当前工作目录解析的.#
- `url` <str> URL of the `<link>` tag.#
- returns: <ElementHandle>#

添加一个 `<link rel="stylesheet">` 标签到页面所需的url 或 `<style type="text/css">` 标签时. 当样式表的onload触发时, 或者当CSS内容被注入框架时, 返回添加的标签.

主框架的 [frame.add\\_style\\_tag\(\\*\\*kwargs\)](#) 快捷方式.

## # page.bring\_to\_front()#

---

- returns: <NoneType>#

将页面放到前面(激活标签).

## # page.check(selector, \*\*kwargs)#

---

- `selector` <str> 一个搜索元素的选择器。如果有多个元素满足选择器, 将使用第一个元素。有关更多细节, 请参阅[working with selectors](#) .#
- `force` <bool> 是否绕过[actionability](#)检查。默认值为 `false` .#
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生, 并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项, 比如导航到不可访问的页面。默认值为 `false` .#
- `position` <Dict> 相对于元素填充框的左上角使用的一个点。如果没有指定, 则使用元素的某个可见点.#
  - `x` <float>
  - `y` <float>
- `strict` <bool> 当为true时, 调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素, 调用将抛出一个异常.#

- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。<#>
- `trial` `<bool>` 设置后，该方法只执行 `actionability` 检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的。<#>
- `returns`: `<NoneType>` <#>

这个方法通过以下步骤检查元素匹配选择器::

1. 找到一个元素匹配选择器。如果没有，则等待直到匹配的元素被附加到 DOM.
2. 确保匹配的元素是一个复选框或单选输入。如果不是，则排除此方法。如果元素已被选中，则该方法立即返回.
3. 等待匹配元素的 `actionability` 检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作.
4. 如果需要，将元素滚动到视图中.
5. 使用 `page.mouse` 单击元素的中心.
6. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项.
7. 确保元素现在被选中。如果不是，则排除此方法.

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 `TimeoutError`。传递零超时将禁用此功能。

主框架 `frame.check(selector, **kwargs)` 的快捷方式。

## [# page.click\(selector, \\*\\*kwargs\)](#)<#>

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅 [working with selectors](#) <#>
- `button` `<"left"|"right"|"middle">` 默认左 `left` <#>
- `click_count` `<int>` 默认为1,详情查看 [UIEvent.detail](#) <#>
- `delay` `<float>` `mousedown` 和 `mouseup` 之间的等待时间，单位是毫秒。默认为0.<#>
- `force` `<bool>` 是否绕过 `actionability` 检查。默认值为 `false` <#>
- `modifiers` `<List["Alt"|"Control"|"Meta"|"Shift"]>` `modifiers` 按键要按。确保在操作期间只按下这些修饰符，然后恢复当前的修饰符。如果未指定，则使用当前按下的修饰符。<#>

- `no_wait_after` `<bool>` 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` `.#`
- `position` `<Dict>` 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点 `.#`
  - `x` `<float>`
  - `y` `<float>`
- `strict` `<bool>` 当为`true`时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常 `.#`
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改 `.#`
- `trial` `<bool>` 设置后，该方法只执行 `actionability` 检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的 `.#`
- `returns:` `<NoneType>` `.#`

这个方法通过执行以下步骤点击元素匹配选择器：

1. 找到一个元素匹配选择器。如果没有，则等待直到匹配的元素被附加到DOM.
2. 等待匹配元素的 `actionability` 检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作.
3. 如果需要，将元素滚动到视图中.
4. 使用 `page.mouse` 单击元素的中心, or the specified `position` .
5. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项.

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 `TimeoutError`。传递零超时将禁用此功能。

主框架 `frame.click(selector, **kwargs)` 的快捷方式.

## # `page.close(**kwargs)` #

- `run_before_unload` `<bool>` 默认为 `false`。是否运行卸载前页面处理程序 `before unload` `.#`
- `returns:` `<NoneType>` `.#`

如果 `run_before_unload` is `false`, 则不运行任何卸载处理程序, 并等待关闭页面. 如果 `run_before_unload` is `true` 该方法将运行卸载处理程序, 但 **不会等待** 页面关闭.

默认情况下, `page.close()` 不会在卸载处理程序之前运行.

#### NOTE

如果 `run_before_unload` 被传递为`true`, 一个 `beforeunload` 对话框可能会被调用, 并且应该通过 `page.on("dialog")` 事件手动处理.

## # page.content()#

---

- returns: `<str>` #

获取页面的完整HTML内容, 包括文档类型.

## # page.context#

---

- returns: `<BrowserContext>` #

获取页面所属的浏览器上下文.

## # page.dblclick(selector, \*\*kwargs)#

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器, 将使用第一个元素。有关更多细节, 请参阅 [working with selectors](#) .#
- `button` `<"left"|"right"|"middle">` 默认左 `left` .#
- `delay` `<float>` `mousedown` 和 `mouseup` 之间的等待时间, 单位是毫秒。默认为0.#
- `force` `<bool>` 是否绕过 `actionability` 检查。默认值为 `false` .#
- `modifiers` `<List["Alt"|"Control"|"Meta"|"Shift"]>` modifiers按键要按。确保在操作期间只按下这些修饰符, 然后恢复当前的修饰符。如果未指定, 则使用当前按下的修饰符.#

- `no_wait_after` `<bool>` 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false`。#
- `position` `<Dict>` 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点。#
  - `x` `<float>`
  - `y` `<float>`
- `strict` `<bool>` 当为`true`时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。#
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。#
- `trial` `<bool>` 设置后，该方法只执行 `actionability` 检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的。#
- `returns:` `<NoneType>` #

该方法通过执行以下步骤双击元素匹配选择器：

1. 找到一个元素匹配选择器。如果没有，则等待直到匹配的元素被附加到DOM.
2. 等待匹配元素的 `actionability` 检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作.
3. 如果需要，将元素滚动到视图中.
4. 使用 `page.mouse` 方法,双击元素中心位置.
5. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项.该方法执行以下步骤双击元素,注意，如果 `dblclick()` 的第一次单击触发了一个导航事件，则该方法将抛出.

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 `TimeoutError`。传递零超时将禁用此功能。

#### NOTE

`page.dblclick()` 分发两个 `click` 事件和一个 `dblclick` 事件.

主框架 `frame.dblclick(selector, **kwargs)` 的快捷方式.

## # page.dispatch\_event(selector, type, \*\*kwargs) #

---

- `selector` <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#) .#
- `type` <str> DOM事件类型: `"click"` , `"dragstart"` 等.#
- `event_init` <EvaluationArgument> 可选的特定于事件的初始化属性.#
- `strict` <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: <NoneType>.#

下面的代码片段分派元素上的 `单击` 事件。无论元素的可见性状态如何，单击都将被分派。这相当于调用 [element.click\(\)](#) .

- Sync

```
page.dispatch_event("button#submit", "click")
```

- Async

```
await page.dispatch_event("button#submit", "click")
```

在底层，它根据给定的类型创建一个事件实例，使用 `event_init` 属性初始化它，并在元素上分派它。默认情况下，事件是组合的、可取消的和冒泡的。

由于 `event_init` 是特定于事件的，请参考事件文档中的初始属性列表：

- [DragEvent](#)
- [FocusEvent](#)
- [KeyboardEvent](#)
- [MouseEvent](#)
- [PointerEvent](#)
- [TouchEvent](#)
- [Event](#)



如果你想要将活动对象传递到事件中，你也可以指定 `jhandle` 作为属性值：

- Sync

```
# note you can only create data_transfer in chromium and
firefox
data_transfer = page.evaluate_handle("new DataTransfer()")
page.dispatch_event("#source", "dragstart", { "dataTransfer":
data_transfer })
```

- Async

```
# note you can only create data_transfer in chromium and
firefox
data_transfer = await page.evaluate_handle("new
DataTransfer()")
await page.dispatch_event("#source", "dragstart", {
"dataTransfer": data_transfer })
```

## # `page.drag_and_drop(source, target,` `**kwargs)` #

---

- `source` <str> #
- `target` <str> #
- `force` <bool> 是否绕过 [actionability](#) 检查。默认值为 `false` .#
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- `source_position` <Dict> 此时相对于元素填充框的左上角单击源元素。如果没有指定，则使用元素的某个可见点.#
  - `x` <float>
  - `y` <float>
- `strict` <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `target_position` <Dict> 此时相对于元素填充框的左上角落在目标元素上。如果没有指定，则使用元素的某个可见点.#



- `x` `<float>`
- `y` `<float>`
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。<#>
- `trial` `<bool>` 设置后，该方法只执行 `actionability` 检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的。<#>
- returns: `<NoneType>` <#>

## # `page.emulate_media(**kwargs)` <#>

- `color_scheme` `<NoneType|'light'|'dark'|'no-preference'>` 模拟 `'prefers-colors-scheme'` 媒体特性，支持的值为 `'light'`, `'dark'`, `'no-preference'`。传递 `null` 将禁用颜色方案仿真。<#>
- `forced_colors` `<NoneType|'active'|'none'>` 模拟 `'forced-colors'` 媒体特性，支持的值为 `'active'` and `'none'`。传递 `null` 将禁用强制颜色模拟。<#>

### NOTE

它在WebKit中不支持，请在他们的问题跟踪器中查看 [here](#)。

- `media` `<NoneType|'screen'|'print'>` 修改页面的CSS媒体类型。唯一允许的值是 `'screen'`, `'print'` and `null`。传递 `null` 将禁用CSS媒体模拟。<#>
- `reduced_motion` `<NoneType|'reduce'|'no-preference'>` 模拟 `'prefers-reduced-motion'` 媒体特性，支持的值为 `'reduce'`, `'no-preference'`。传递 `null` 将禁用减少的运动仿真。<#>
- returns: `<NoneType>` <#>

这个方法通过media参数改变 `CSS media type`，或者使用 `colorScheme` 参数改变 `'prefers-colors-scheme'` 的媒体特性。

- Sync

```
page.evaluate("matchMedia('screen').matches")
# → True
page.evaluate("matchMedia('print').matches")
```

```
# → False

page.emulate_media(media="print")
page.evaluate("matchMedia('screen').matches")
# → False
page.evaluate("matchMedia('print').matches")
# → True

page.emulate_media()
page.evaluate("matchMedia('screen').matches")
# → True
page.evaluate("matchMedia('print').matches")
# → False
```

- Async

```
await page.evaluate("matchMedia('screen').matches")
# → True
await page.evaluate("matchMedia('print').matches")
# → False

await page.emulate_media(media="print")
await page.evaluate("matchMedia('screen').matches")
# → False
await page.evaluate("matchMedia('print').matches")
# → True

await page.emulate_media()
await page.evaluate("matchMedia('screen').matches")
# → True
await page.evaluate("matchMedia('print').matches")
# → False
```

- Sync

```

page.emulate_media(color_scheme="dark")
page.evaluate("matchMedia('(prefers-color-scheme:
dark)').matches")
# → True
page.evaluate("matchMedia('(prefers-color-scheme:
light)').matches")
# → False
page.evaluate("matchMedia('(prefers-color-scheme: no-
preference)').matches")

```

- Async

```

await page.emulate_media(color_scheme="dark")
await page.evaluate("matchMedia('(prefers-color-scheme:
dark)').matches")
# → True
await page.evaluate("matchMedia('(prefers-color-scheme:
light)').matches")
# → False
await page.evaluate("matchMedia('(prefers-color-scheme: no-
preference)').matches")
# → False

```

## # page.eval\_on\_selector(selector, expression, \*\*kwargs)#

---

- **selector** <str> 要查询的选择器。有关更多细节，请参阅[working with selectors](#).#
- **expression** <str> 要在浏览器上下文中计算的 **JavaScript** 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值.#
- **arg** <EvaluationArgument> 传递给 **expression** 的可选参数.#
- **strict** <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- returns: <Serializable>.#

CAUTION

此方法不等待元素通过可操作性检查，因此可能导致不稳定的测试。使用 [locator.evaluate\(expression, \\*\\*kwargs\)](#)，其它 [Locator](#) 其他 [Locator](#) 助手方法或 [web](#) 优先断言。

该方法在页面中找到与指定选择器匹配的元素，并将其作为第一个参数传递给表达式。如果没有匹配该选择器的元素，该方法将抛出错误。返回表达式的值。

如果 `expression`` 返回 [Promise](#)，则 [page.eval\\_on\\_selector\(selector, expression, \\*\\*kwargs\)](#) 将等待 `promise` 解析并返回它的值。

例子:

- Sync

```
search_value = page.eval_on_selector("#search", "el =>
el.value")
preload_href = page.eval_on_selector("link[rel=preload]", "el
=> el.href")
html = page.eval_on_selector(".main-container", "(e, suffix)
=> e.outer_html + suffix", "hello")
```

- Async

```
search_value = await page.eval_on_selector("#search", "el =>
el.value")
preload_href = await
page.eval_on_selector("link[rel=preload]", "el => el.href")
html = await page.eval_on_selector(".main-container", "(e,
suffix) => e.outer_html + suffix", "hello")
```

主框架 [frame.eval\\_on\\_selector\(selector, expression, \\*\\*kwargs\)](#) 的快捷方式。

## # `page.eval_on_selector_all(selector, expression, **kwargs)` #

- `selector` [<str>](#) 要查询的选择器。有关更多细节，请参阅 [working with selectors.#](#)
- `expression` [<str>](#) 要在浏览器上下文中计算的 [JavaScript](#) 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作

为表达式求值.#

- `arg` <EvaluationArgument> 传递给 `expression` 的可选参数.#
- returns: <Serializable>#

#### NOTE

在大多数情况下, `locator.evaluate_all(expression, **kwargs)`, 其它 `Locator` 助手方法和 `web-first`断言做得更好.

该方法查找页面中与指定选择器匹配的所有元素, 并将匹配元素的数组作为第一个参数传递给表达式. 返回表达式调用的结果.

如果 `expression` 返回 `Promise`, 那么 `page.eval_on_selector_all(selector, expression, **kwargs)` 将等待promise解析并返回它的值.

例子:

- Sync

```
div_counts = page.eval_on_selector_all("div", "(divs, min) =>
divs.length >= min", 10)
```

- Async

```
div_counts = await page.eval_on_selector_all("div", "(divs,
min) => divs.length >= min", 10)
```

## # `page.evaluate(expression, **kwargs)`#

- `expression` <str> 要在浏览器上下文中计算的 `JavaScript` 表达式. 如果它看起来像一个函数声明, 它会被解释为一个函数. 否则, 作为表达式求值.#
- `arg` <EvaluationArgument> 传递给 `expression` 的可选参数.#
- returns: <Serializable>#

返回表达式调用的值.

如果函数传递给 `page.evaluate(expression, **kwargs)` 返回 `Promise`, 那么 `page.evaluate(expression, **kwargs)` 将等待promise解析并返回它的值.

如果函数传递给 `page.evaluate(expression, **kwargs)` 返回一个不可序列化( `non-Serializable` ) 的值, 那么 `page.evaluate(expression, **kwargs)` 解析为 `undefined`. Playwright 还支持传递一些附加值, 不能通过 `JSON` 序列化: `-0`, `NaN`, `Infinity`, `-Infinity`.

将参数传递给表达式:

- Sync

```
result = page.evaluate("([x, y]) => Promise.resolve(x * y)",  
[7, 8])  
print(result) # prints "56"
```

- Async

```
result = await page.evaluate("([x, y]) => Promise.resolve(x *  
y)", [7, 8])  
print(result) # prints "56"
```

字符串也可以代替函数传入:

- Sync

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

- Async

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

可以将 `ElementHandle` 实例作为参数传递给 `page.evaluate(expression, **kwargs)`:

- Sync

```
body_handle = page.evaluate("document.body")
html = page.evaluate("([body, suffix]) => body.innerHTML +
suffix", [body_handle, "hello"])
body_handle.dispose()
```

- Async

```
body_handle = await page.evaluate("document.body")
html = await page.evaluate("([body, suffix]) => body.innerHTML
+ suffix", [body_handle, "hello"])
await body_handle.dispose()
```

主框架 [frame.evaluate\(expression, \\*\\*kwargs\)](#) 的快捷方式。

## # page.evaluate\_handle(expression, \*\*kwargs) #

---

- **expression** <[str](#)> 要在浏览器上下文中计算的 [JavaScript](#) 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值。<#>
- **arg** <[EvaluationArgument](#)> 传递给 **expression** 的可选参数。<#>
- returns: <[JSHandle](#)> <#>

以jhandle形式返回表达式调用的值。 [JSHandle](#)。

[page.evaluate\(expression, \\*\\*kwargs\)](#) and [page.evaluate\\_handle\(expression, \\*\\*kwargs\)](#) 之间唯一的区别是 [page.evaluate\\_handle\(expression, \\*\\*kwargs\)](#) 返回 [JSHandle](#)。

如果函数传递给 [page.evaluate\\_handle\(expression, \\*\\*kwargs\)](#) 返回一个 [Promise](#)，那么 [page.evaluate\\_handle\(expression, \\*\\*kwargs\)](#) 将等待promise解析并返回它的值。

- Sync

```
a_window_handle =
page.evaluate_handle("Promise.resolve(window)")
a_window_handle # handle for the window object.
```

- Async

```
a_window_handle = await  
page.evaluate_handle("Promise.resolve(window)")  
a_window_handle # handle for the window object.
```

字符串也可以代替函数传入:

- Sync

```
a_handle = page.evaluate_handle("document") # handle for the  
"document"
```

- Async

```
a_handle = await page.evaluate_handle("document") # handle for  
the "document"
```

[JSHandle](#) 实例可以作为参数传递给 [page.evaluate\\_handle\(expression, \\*\\*kwargs\)](#):

- Sync

```
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()
```

- Async

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



#

## page.expect\_console\_message(\*\*kwargs)#

- `predicate` <Callable[[ConsoleMessage](#)]:bool> 接收 [ConsoleMessage](#) 对象，并在等待应该解决时解析为真值。<#>
- `timeout` <float> 最大等待时间，单位为毫秒。默认为 `30000` (30 秒)。通过0禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) 来更改。<#>
- returns: <[EventManager](#)[[ConsoleMessage](#)]><#>

执行操作并等待在页面中记录一个 [ConsoleMessage](#)，如果提供了 `predicate`，则它将 [ConsoleMessage](#) 值传递给 `predicate` 函数，并等待 `predicate(message)` 返回一个真值。如果页面在触发 [page.on\("console"\)](#) 事件之前关闭，将抛出一个错误。

## # page.expect\_download(\*\*kwargs)#

- `predicate` <Callable[[Download](#)]:bool> 接收 [Download](#) 对象，并在等待应该解决时解析为true值。<#>
- `timeout` <float> 最大等待时间，单位为毫秒。默认为 `30000` (30 秒)。通过0禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) 来更改。<#>
- returns: <[EventManager](#)[[Download](#)]><#>

执行操作并等待新的 [Download](#)。如果提供了 `provided`，它将 [Download](#) 的值传递给 `predicate` 函数并等待 `predicate(download)` 返回一个真值。如果页面在下载事件被触发之前关闭，将抛出一个错误。

## # page.expect\_event(event, \*\*kwargs)#

- `event` <str> 事件名称，与通常传递给 `*.on(event)` 的名称相同。<#>
- `predicate` <Callable> 接收事件数据，并在等待应该被解析时解析为真值。<#>
- `timeout` <float> 最大等待时间，单位为毫秒。默认为 `30000` (30 秒)。通过0禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) 来更改。<#>
- returns: <[EventManager](#)><#>

等待事件触发，并将其值传递给 `predicate` 函数。当 `predicate()` 返回真值时返回。如果在触发事件之前页面已关闭，则将抛出一个错误。返回事件数据值。

- Sync

```
with page.expect_event("framenavigated") as event_info:
    page.click("button")
frame = event_info.value
```

- Async

```
async with page.expect_event("framenavigated") as event_info:
    await page.click("button")
frame = await event_info.value
```

## # `page.expect_file_chooser(**kwargs)` #

---

- `predicate` `<Callable[[FileChooser]:bool>` 接收 `FileChooser` 对象，并在等待应该解决时解析为true值。#
- `timeout` `<float>` 最大等待时间，单位为毫秒。默认为 `30000` (30秒)。通过0禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` 来更改。#
- `returns:` `<EventManager[FileChooser]>` #

执行操作并等待创建一个新的 `FileChooser`。如果提供了 `provided`，则它将 `FileChooser` 值传递给 `predicate` 函数，并等待 `predicate(fileChooser)` 返回一个真值。如果在打开文件选择器之前关闭该页，则将抛出一个错误。

## # `page.expect_navigation(**kwargs)` #

---

- `timeout` `<float>` 最大操作时间，单位为毫秒，默认为30秒，通过0表示禁止超时。默认值可以通过使用 `browser_context.set_default_navigation_timeout(timeout)`, `browser_context.set_default_timeout(timeout)`, `page.set_default_navigation_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来修改。#

- `url` <[str](#)|[Pattern](#)|[Callable](#)[[URL](#)]:[bool](#)> 一个glob模式、regex模式或谓词，在等待导航时接收匹配的url。注意，如果参数是一个不带通配符的字符串，该方法将等待导航到与该字符串完全相等的URL。<#>
- `wait_until` <"load"|"domcontentloaded"|"networkidle"|"commit"> 当认为操作成功时，默认为 `load`。事件可以是:<#>
  - `'domcontentloaded'` - 当 `domcontentloaded` 事件被触发时，认为操作已经完成。
  - `'load'` - 当触发 `load` 事件时，认为操作已经完成。
  - `'networkidle'` - 当至少 `500毫秒` 没有网络连接时，认为操作已经完成。
  - `'commit'` - 当接收到网络响应并开始加载文档时，认为操作已经完成。
- returns: <[EventManager](#)[[Response](#)]><#>

等待主框架导航并返回主资源响应。在多个重定向的情况下，导航将使用最后一个重定向的响应进行解析。如果导航到一个不同的锚或导航由于历史API的使用，导航将解析为null。

当页面导航到一个新的URL或重新加载时，这个问题就会解决。当您运行将间接导致页面导航的代码时，它非常有用。例:点击目标有一个 `onclick` 处理程序，通过 `setTimeout` 触发导航。考虑一下这个例子：

- Sync

```
with page.expect_navigation():
    page.click("a.delayed-navigation") # clicking the link
    will indirectly cause a navigation
    # Resolves after navigation has finished
```

- Async

```
async with page.expect_navigation():
    await page.click("a.delayed-navigation") # clicking the
    link will indirectly cause a navigation
    # Resolves after navigation has finished
```

#### NOTE

使用 [History API](#) 更改URL被视为导航。

主框架 [frame.expect\\_navigation\(\\*\\*kwargs\)](#) 的快捷方式。

## # page.expect\_popup(\*\*kwargs)#

---

- `predicate` <[Callable](#)[[Page](#)]:[bool](#)> 接收 [Page](#) 对象，并在等待应该解决时解析为true值。<#>
- `timeout` <[float](#)> 最大等待时间，单位为毫秒。默认为 `30000` (30秒)。通过0禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) 来更改。<#>
- returns: <[EventContextManager](#)[[Page](#)]><#>

执行动作并等待弹出 [Page](#)。如果提供了 `predicate`，它将 [Popup] 的值传递给 `predicate` 函数，并等待 `predicate(page)` 返回一个真值。如果页面在弹出事件被触发之前关闭，将抛出一个错误。

## # page.expect\_request(url\_or\_predicate, \*\*kwargs)#

---

- `url_or_predicate` <[str](#)|[Pattern](#)|[Callable](#)[[Request](#)]:[bool](#)> 请求 URL 字符串、`regex`或`predicate`接收 [Request](#) 对象。当通过上下文选项提供了一个 `base_url` 并且传递的URL是一个路径时，它会通过 [new URL\(\)](#) 构造函数合并。<#>
- `timeout` <[float](#)> 最大等待时间，单位为毫秒，缺省值为30秒，通过 `0` 表示不允许超时。默认值可以通过使用 [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- returns: <[EventContextManager](#)[[Request](#)]><#>

等待匹配的请求并返回它。有关事件的更多细节，请参阅等待事件 [waiting for event](#)。

- Sync

```
with page.expect_request("http://example.com/resource") as first:
    page.click('button')
first_request = first.value

# or with a lambda
with page.expect_request(lambda request: request.url ==
    "http://example.com" and request.method == "get") as second:
    page.click('img')
second_request = second.value
```

- Async

```

async with page.expect_request("http://example.com/resource")
as first:
    await page.click('button')
first_request = await first.value

# or with a lambda
async with page.expect_request(lambda request: request.url ==
"http://example.com" and request.method == "get") as second:
    await page.click('img')
second_request = await second.value

```

## # page.expect\_request\_finished(\*\*kwargs)#

- **predicate** <Callable[Request]:bool> 接收 [Request](#) 对象，并在等待应该解决时解析为true值。<#>
- **timeout** <float> 最大等待时间，单位为毫秒。默认为 **30000** (30秒)。通过0禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) 来更改。<#>
- returns: <[EventContextManager](#)[Request]><#>

执行动作并等待一个[Request](#) 完成加载. 如果提供了 predicate ,它将 [Request](#) 的值传递给 **predicate** 函数,并等待 **predicate(request)** 返回一个真值。如果页面在[page.on\("requestfinished"\)](#) 事件之前关闭，将抛出一个错误。

## # page.expect\_response(url\_or\_predicate, \*\*kwargs)#

- **url\_or\_predicate** <str|Pattern|Callable[Response]:bool> 请求 URL字符串、regex或predicate接收 [Response](#) 对象。当通过上下文选项提供了一个 **base\_url** 并且传递的URL是一个路径时，它会通过 [new URL\(\)](#) 构造函数合并。<#>
- **timeout** <float> 最大等待时间，单位为毫秒，缺省值为30秒，通过 **0** 表示不允许超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- returns: <[EventContextManager](#)[Response]><#>

返回匹配的响应。有关事件的更多细节，请参阅等待事件 [waiting for event](#)

- Sync

```
with page.expect_response("https://example.com/resource") as
    response_info:
        page.click("input")
    response = response_info.value
    return response.ok

# or with a lambda
with page.expect_response(lambda response: response.url ==
    "https://example.com" and response.status == 200) as
    response_info:
        page.click("input")
    response = response_info.value
    return response.ok
```

- Async

```
async with
    page.expect_response("https://example.com/resource") as
        response_info:
            await page.click("input")
        response = await response_info.value
        return response.ok

# or with a lambda
async with page.expect_response(lambda response: response.url
    == "https://example.com" and response.status == 200) as
        response_info:
            await page.click("input")
        response = await response_info.value
        return response.ok
```

## **# page.expect\_websocket(\*\*kwargs)#**

- `predicate` `<Callable[WebSocket]:bool>` 接收 `WebSocket` 对象，并在等待应该解决时解析为true值。<#>

- `timeout` <float> 最大等待时间，单位为毫秒。默认为 `30000` (30秒)。通过0禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` 来更改。<#>
- returns: <[EventManager](#) [[WebSocket](#)]> <#>

执行操作并等待一个新的 [WebSocket](#)。如果提供了 `predicate`，它将 [WebSocket](#) 的值传递给 `predicate` 函数，并等待 `predicate(webSocket)` 返回一个真值。如果页面在 [WebSocket](#) 事件被触发之前关闭，将抛出一个错误。

## # `page.expect_worker(**kwargs)` <#>

---

- `predicate` <[Callable](#) [[Worker](#)]:[bool](#)> 接收 [Worker](#) 对象，并在等待应该解决时解析为true值。<#>
- `timeout` <float> 最大等待时间，单位为毫秒。默认为 `30000` (30秒)。通过0禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` 来更改。<#>
- returns: <[EventManager](#) [[Worker](#)]> <#>

执行动作并等待一个新的 [Worker](#)。如果提供了 `predicate`，它将 [Worker](#) 的值传递给 `predicate` 函数，并等待 `predicate(worker)` 返回一个真值。如果页面在触发 `worker` 事件之前关闭，则将抛出一个错误。

## # `page.expose_binding(name, callback, **kwargs)` <#>

---

- `name` <[str](#)> window 对象上的函数名。<#>
- `callback` <[Callable](#)> Playwright 的上下文中被调用的回调函数。<#>
- `handle` <[bool](#)> 是否将参数作为句柄传递，而不是按值传递。当传递句柄时，只支持一个参数。当传递值时，支持多个参数。<#>
- returns: <[NoneType](#)> <#>

该方法为页面中每一个 `frame` 的窗口对象添加一个名为 `name` 的函数。当被调用时，函数执行回调并返回一个 [Promise](#)，该 [Promise](#) 解析为回调的返回值。如果回调返回一个 [Promise](#)，它将被等待。

`callback` 函数的第一个参数包含调用者的信息: `{ browserContext: BrowserContext, page: Page, frame: Frame }`。

查看 [browser\\_context.expose\\_binding\(name, callback, \\*\\*kwargs\)](#) 上下文级版本。

#### NOTE

Functions installed via [page.expose\\_binding\(name, callback, \\*\\*kwargs\)](#) survive navigations.

一个将页面URL暴露给页面中所有 frame 的例子:

- Sync

```
from playwright.sync_api import sync_playwright

def run(playwright):
    webkit = playwright.webkit
    browser = webkit.launch(headless=False)
    context = browser.new_context()
    page = context.new_page()
    page.expose_binding("pageURL", lambda source:
source["page"].url)
    page.set_content("""
<script>
    async function onClick() {
        document.querySelector('div').textContent = await
window.pageURL();
    }
</script>
<button onclick="onClick()">Click me</button>
<div></div>
""")
    page.click("button")

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

- Async

```
import asyncio
from playwright.async_api import async_playwright

async def run(playwright):
    webkit = playwright.webkit
```



```

browser = await webkit.launch(headless=false)
context = await browser.new_context()
page = await context.new_page()
await page.expose_binding("pageURL", lambda source:
source["page"].url)
await page.set_content("""
<script>
    async function onClick() {
        document.querySelector('div').textContent = await
window.pageURL();
    }
</script>
<button onclick="onClick()">Click me</button>
<div></div>
""")
await page.click("button")

async def main():
    async with async_playwright() as playwright:
        await run(playwright)
asyncio.run(main())

```

传递 element handle 的例子:

- Sync

```

def print(source, element):
    print(element.text_content())

page.expose_binding("clicked", print, handle=true)
page.set_content("""
<script>
    document.addEventListener('click', event =>
window.clicked(event.target));
</script>
<div>Click me</div>
<div>Or click me</div>
""")

```

- Async

```

async def print(source, element):
    print(await element.text_content())

await page.expose_binding("clicked", print, handle=true)
await page.set_content("""
    \<script>
        document.addEventListener('click', event =>
window.clicked(event.target));
    \</script>
    \<div>Click me\</div>
    \<div>Or click me\</div>
""")

```

## # `page.expose_function(name, callback)` #

- `name` <str> window 对象上的函数名 #
- `callback` <Callable> 回调函数，将在playwright的上下文中被调用.#
- returns: <NoneType> #

该方法在页面中每一frame 的 `window` 对象上添加一个名为 `name` 的函数. 当被调用时，函数执行回调并返回一个 `Promise` ,该Promise解析为回调的返回值.

如果回调返回一个 `Promise`，它将被等待

查看 [browser\\_context.expose\\_function\(name, callback\)](#) 用于上下文范围的公开函数

### NOTE

Functions installed via [page.expose\\_function\(name, callback\)](#) survive navigations.

一个在页面中添加 `sha256` 函数的例子:

- Sync

```

import hashlib
from playwright.sync_api import sync_playwright

def sha256(text):

```

```

    m = hashlib.sha256()
    m.update(bytes(text, "utf8"))
    return m.hexdigest()

def run(playwright):
    webkit = playwright.webkit
    browser = webkit.launch(headless=False)
    page = browser.new_page()
    page.expose_function("sha256", sha256)
    page.set_content("""
        <script>
            async function onClick() {
                document.querySelector('div').textContent = await
window.sha256('PLAYWRIGHT');
            }
        </script>
        <button onclick="onClick()">Click me</button>
        <div></div>
    """)
    page.click("button")

with sync_playwright() as playwright:
    run(playwright)

```

- Async

```

import asyncio
import hashlib
from playwright.async_api import async_playwright

def sha256(text):
    m = hashlib.sha256()
    m.update(bytes(text, "utf8"))
    return m.hexdigest()

async def run(playwright):
    webkit = playwright.webkit
    browser = await webkit.launch(headless=False)
    page = await browser.new_page()
    await page.expose_function("sha256", sha256)

```

```

await page.set_content("""
    \<script>
        async function onClick() {
            document.querySelector('div').textContent = await
window.sha256('PLAYWRIGHT');
        }
    \</script>
    \<button onclick="onClick()">Click me\</button>
    \<div>\</div>
""")
await page.click("button")

async def main():
    async with async_playwright() as playwright:
        await run(playwright)
asyncio.run(main())

```

## # page.fill(selector, value, \*\*kwargs) #

- **selector** <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#) .#
- **value** <str> Value to fill for the `\<input>`, `\<textarea>` or `[contenteditable]` element.#
- **force** <bool> 是否绕过[actionability](#)检查。默认值为 `false` .#
- **no\_wait\_after** <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- **strict** <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- **timeout** <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- **returns:** <NoneType>.#

这个方法等待元素匹配选择器，等待[可操作性](#)检查,聚焦元素，填充它，并在填充后触发一个输入事件。请注意，您可以传递一个空字符串来清除输入字段。

如果目标元素不是 `\<input>`, `\<textarea>` or `[contenteditable]`, 此方法将抛出一个错误。但是, 如果该元素位于 `\<label>` 元素中, 且该元素具有关联控件, 则该控件将被填充。

要发送细粒度的键盘事件, 请使用 [page.type\(selector, text, \\*\\*kwargs\)](#)。

主框架 [frame.fill\(selector, value, \\*\\*kwargs\)](#) 快捷方式。

## # page.focus(selector, \*\*kwargs)#

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器, 将使用第一个元素。有关更多细节, 请参阅[working with selectors](#)。`#`
- `strict` `<bool>` 当为true时, 调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素, 调用将抛出一个异常。`#`
- `timeout` `<float>` 最大时间, 单位为毫秒, 默认为30秒, 通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。`#`
- returns: `<NoneType>` `#`

这个方法获取一个带有选择器的元素并聚焦于它。如果没有元素匹配选择器, 该方法将等待, 直到匹配元素出现在DOM中。

主框架 [frame.focus\(selector, \\*\\*kwargs\)](#) 快捷方式。

## # page.frame(\*\*kwargs)#

---

- `name` `<str>` 在iframe的name属性中指定的frame名。可选的。`#`
- `url` `<str|Pattern|Callable[URL]:bool>` 一个glob模式, regex模式或谓词接收frame的URL作为url对象。可选的。`#`
- returns: `<NoneType|Frame>` `#`

返回匹配指定条件的frame。必须指定 `name` 或 `url`。

```
frame = page.frame(name="frame-name")
```

```
frame = page.frame(url=r".*domain.*")
```

## # page.frame\_locator(selector) #

---

- `selector` <str> 解析DOM元素时使用的选择器。有关更多细节，请参阅[working with selectors](#) .#
- returns: <FrameLocator> #

在使用iframes时，您可以创建一个帧定位器，该定位器将进入iframe并允许选择该iframe中的元素。下面的代码片段在id为 `my-frame` 的iframe中定位到文本为"Submit"的元素，例如 `\<iframe id="my-frame">`：

- Sync

```
locator = page.frame_locator("#my-  
iframe").locator("text=Submit")  
locator.click()
```

- Async

```
locator = page.frame_locator("#my-  
iframe").locator("text=Submit")  
await locator.click()
```

## # page.frames #

---

- returns: <List[Frame]> #

附在页面上的所有frame的数组。

## # page.get\_attribute(selector, name, \*\*kwargs) #

---

- `selector` <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#) .#
- `name` <str> 属性名.#
- `strict` <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#

- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来更改。`#`
- returns: `<NoneType|str>` `#`

返回元素属性值。

## # `page.go_back(**kwargs)` `#`

---

- `timeout` `<float>` 最大操作时间，单位为毫秒，默认为30秒，通过0表示禁止超时。默认值可以通过使用 `browser_context.set_default_navigation_timeout(timeout)`, `browser_context.set_default_timeout(timeout)`, `page.set_default_navigation_timeout(timeout)` or `page.set_default_timeout(timeout)` 方法来修改。`#`
- `wait_until` `<"load"|"domcontentloaded"|"networkidle"|"commit">` 当认为操作成功时，默认为 `load`。事件可以是：`#`
  - `'domcontentloaded'` - 当 `domcontentloaded` 事件被触发时，认为操作已经完成。
  - `'load'` - 当触发 `load` 事件时，认为操作已经完成。
  - `'networkidle'` - 当至少 `500毫秒` 没有网络连接时，认为操作已经完成。
  - `'commit'` - 当接收到网络响应并开始加载文档时，认为操作已经完成。
- returns: `<NoneType|Response>` `#`

返回主资源响应。在多个重定向的情况下，导航将使用最后一个重定向的响应进行解析。如果不能返回，返回 `null`。

导航到历史记录的前一页。

## # `page.go_forward(**kwargs)` `#`

---

- `timeout` `<float>` 最大操作时间，单位为毫秒，默认为30秒，通过0表示禁止超时。默认值可以通过使用 `browser_context.set_default_navigation_timeout(timeout)`, `browser_context.set_default_timeout(timeout)`,

[page.set\\_default\\_navigation\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来修改.#

- `wait_until` `<"load"|"domcontentloaded"|"networkidle"|"commit">`  
当认为操作成功时，默认为 `load`。事件可以是:#
  - `'domcontentloaded'` - 当 `domcontentloaded` 事件被触发时，认为操作已经完成。
  - `'load'` - 当触发 `load` 事件时，认为操作已经完成。
  - `'networkidle'` - 当至少 `500毫秒` 没有网络连接时，认为操作已经完成。
  - `'commit'` - 当接收到网络响应并开始加载文档时，认为操作已经完成。
- `returns: <NoneType|Response>#`

返回主资源响应。在多个重定向的情况下，导航将使用最后一个重定向的响应进行解析。如果不能前进，返回null。

导航到历史记录的前一页。

## # `page.goto(url, **kwargs)#`

---

- `url` `<str>` 页面导航到的url。url应该包括协议，例如 `https://`。当通过上下文选项提供了一个 `base_url` 并且传递的URL是一个路径时，它会通过 `new URL()` 构造函数合并.#
- `referer` `<str>` referer头值。如果提供，它将优先于 [page.set\\_extra\\_http\\_headers\(headers\)](#) 设置的referer头值.#
- `timeout` `<float>` 最大操作时间，单位为毫秒，默认为30秒，通过0表示禁止超时。默认值可以通过使用 [browser\\_context.set\\_default\\_navigation\\_timeout\(timeout\)](#), [browser\\_context.set\\_default\\_timeout\(timeout\)](#), [page.set\\_default\\_navigation\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来修改.#
- `wait_until` `<"load"|"domcontentloaded"|"networkidle"|"commit">`  
当认为操作成功时，默认为 `load`。事件可以是:#
  - `'domcontentloaded'` - 当 `domcontentloaded` 事件被触发时，认为操作已经完成。
  - `'load'` - 当触发 `load` 事件时，认为操作已经完成。
  - `'networkidle'` - 当至少 `500毫秒` 没有网络连接时，认为操作已经完成。



- `'commit'` - 当接收到网络响应并开始加载文档时，认为操作已经完成。
- `returns: <NoneType|Response>#`

返回主资源响应。在多个重定向的情况下，导航将使用最后一个重定向的响应进行解析。

如果以下情况，该方法将抛出一个错误::

- 有一个SSL错误(例如在自签名证书的情况下)。
- 目标URL无效。
- 导航过程中超时。
- 远程服务器没有响应或不可达。
- `main` 资源加载失败。

当远程服务器返回任何有效的HTTP状态码时，该方法不会抛出错误，包括404 "not Found"和500 "Internal server error"。这些响应的状态代码可以通过调用 [`response.status`](#) 来获取。

#### NOTE

该方法要么抛出错误，要么返回主资源响应。唯一的例外是导航到 `空白` 或导航到相同的URL与不同的哈希，这将成功并返回`null`。

#### NOTE

无头模式不支持PDF文档的导航。参见 [`upstream issue`](#)。

主框架 [`frame.goto\(url, \*\*kwargs\)`](#) 的快捷方式

## # `page.hover(selector, **kwargs)#`

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅 [`working with selectors`](#) `.#`
- `force` `<bool>` 是否绕过 [`actionability`](#) 检查。默认值为 `false` `.#`
- `modifiers` `<List["Alt"|"Control"|"Meta"|"Shift"]>` `modifiers` 按键要按。确保在操作期间只按下这些修饰符，然后恢复当前的修饰符。如果未指定，则使用当前按下的修饰符 `.#`

- `position` `<Dict>` 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点。<#>
  - `x` `<float>`
  - `y` `<float>`
- `strict` `<bool>` 当为`true`时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。<#>
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- `trial` `<bool>` 设置后，该方法只执行 [actionability](#) 检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的。<#>
- returns: `<NoneType>`<#>

该方法通过执行以下步骤悬停在元素匹配选择器上：

1. 找到一个元素匹配选择器。如果没有，则等待直到匹配的元素被附加到DOM.
2. 等待匹配元素的 [actionability](#) 检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作.
3. 如果需要，将元素滚动到视图中.
4. 使鼠标停在元素中心或指定位置上.
5. 等待发起的导航成功或失败，除非设置了 `noWaitAfter` 选项.

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

主框架 [frame.hover\(selector, \\*\\*kwargs\)](#) 的快捷方式。

## [# page.inner\\_html\(selector, \\*\\*kwargs\)#](#)

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅 [working with selectors](#) [.#](#)
- `strict` `<bool>` 当为`true`时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。<#>
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>

- returns: `<str>#`

Returns `element.innerHTML`.

## # `page.inner_text(selector, **kwargs)#`

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#).#
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: `<str>#`

Returns `element.innerText`.

## # `page.input_value(selector, **kwargs)#`

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#).#
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: `<str>#`

返回以下元素的输入值 `\<input>` or `\<textarea>` or `\<select>` .排除非输入元素.

## # `page.is_checked(selector, **kwargs)#`

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#).#
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: `<bool>` .#

返回元素是否被选中。如果元素不是复选框或单选输入则排除。

## # `page.is_closed()` #

---

- returns: `<bool>` .#

表示该页面已关闭。

## # `page.is_disabled(selector, **kwargs)` #

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#).#
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: `<bool>` .#

返回该元素是否被禁用，与启用[enabled](#)相反。

## # `page.is_editable(selector, **kwargs)` #

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#).#
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: `<bool>` .#

返回元素是否可编辑[editable](#).

## # `page.is_enabled(selector, **kwargs)` #

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#).#
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: `<bool>` .#

返回元素是否被启用[enabled](#).

## # `page.is_hidden(selector, **kwargs)` #

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#).#
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` `<float>` **DEPRECATED** 此选项将被忽略。[page.is\\_hidden\(selector, \\*\\*kwargs\)](#) 不会等待元素被隐藏并立即返回.#
- returns: `<bool>` .#

返回元素是否隐藏，与可见 [visible](#) 相反. 不匹配任何元素的选择器被认为是隐藏的.

## # `page.is_visible(selector, **kwargs)` #

---

- `selector` <[str](#)> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅 [working with selectors](#) .#
- `strict` <[bool](#)> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` <[float](#)> **DEPRECATED** 此选项将被忽略。  
[page.is\\_visible\(selector, \\*\\*kwargs\)](#) 不会等待元素变得可见并立即返回.#
- returns: <[bool](#)>.#

返回元素是否可见 [visible](#). 不匹配任何元素的选择器被认为不可见.

## # `page.locator(selector, **kwargs)` #

---

- `selector` <[str](#)> 解析DOM元素时使用的选择器。有关更多细节，请参阅 [working with selectors](#) .#
- `has` <[Locator](#)> 对selector选中的元素进行再次匹配, 匹配目标中的子元素, 例如: 匹配子元素的 `text=Playwright` 的元素 `\<article>\<div>Playwright\</div>\</article>` .#  
请注意，外部和内部定位器必须属于同一个 frame. 内部定位器不能包含 [FrameLocator](#) .
- `has_text` <[str](#)|[Pattern](#)> 匹配包含指定文本的元素，可能在子元素或后代元素中, 例如: `"Playwright"` 匹配 `\<article>\<div>Playwright\</div>\</article>` .#
- returns: <[Locator](#)>.#

该方法返回一个元素定位器，可用于在页面上执行操作。在执行一个操作之前，`Locator`被立即解析为元素，因此同一定位器上的一系列操作实际上可以在不同的DOM元素上执行。如果这些动作之间的DOM结构发生了变化，就会发生这种情况。

主框架 [frame.locator\(selector, \\*\\*kwargs\)](#) 的快捷方式.

## # page.main\_frame#

---

- returns: <[Frame](#)>#

页面的主框架。页面保证有一个在导航期间持久存在的主框架。

## # page.opener()#

---

- returns: <[NoneType](#)|[Page](#)>#

返回弹出页面的 opener 其他的为空 如果 opener 已经关闭，则返回 `null`。

## # page.pause()#

---

- returns: <[NoneType](#)>#

暂停脚本执行。Playwright 将停止执行脚本，等待用户在页面覆盖中按下 'Resume' 按钮，或者在DevTools控制台中调用 `playwright.resume()`。

用户可以在暂停时检查选择器或执行手动步骤。Resume将从暂停的位置继续运行原始脚本。

### NOTE

这个方法要求 Playwright 以 有头模式启动, 及 [browser\\_type.launch\(\\*\\*kwargs\)](#) 里 `headless=False`。

## # page.pdf(\*\*kwargs)#

---

- `display_header_footer` <[bool](#)> 显示页眉和页脚。默认值为 `false`。#
- `footer_template` <[str](#)> 打印页脚的HTML模板。应该使用与 `header_template` 相同的格式。#
- `format` <[str](#)> 论文格式。如果设置，优先级高于 `宽度` 或 `高度` 选项。默认为 'Letter'。#



- `header_template` `<str>` 打印头的HTML模板。应该是有效的HTML标记, 使用以下类注入打印值: `#`
  - `'date'` 格式化打印日期
  - `'title'` 文档标题
  - `'url'` 文档的位置
  - `'pageNumber'` 当前页码
  - `'totalPages'` 文件中的总页数
- `height` `<str|float>` 纸张高度, 接受单位标记的值. `#`
- `landscape` `<bool>` 纸张朝向。默认值为 `false`. `#`
- `margin` `<Dict>` 纸边距, 默认为 `none`. `#`
  - `top` `<str|float>` 顶部空白, 接受单位标记的值。默认值为 `0`.
  - `right` `<str|float>` 右边距, 接受单位标记的值。默认值为 `0`.
  - `bottom` `<str|float>` 底部边距, 接受单位标记的值。默认值为 `0`.
  - `left` `<str|float>` 左边距, 接受带有单位标记的值。默认值为 `0`.
- `page_ranges` `<str>` 打印的纸张范围, 例如: '1-5, 8, 11-13'. 默认为空字符串, 这意味着打印所有页面. `#`
- `path` `<Union[str, pathlib.Path]>` PDF文件保存到的路径. 如果`path`是一个相对路径, 那么它是相对于当前工作目录解析的。如果没有提供路径, PDF将不会被保存到磁盘. `#`
- `prefer_css_page_size` `<bool>` 指定任何在 `页面` 中声明的 CSS 尺寸优先于在宽度、高度或格式选项中声明的 CSS 尺寸. 默认为 `false`, 它将缩放内容以适应纸张大小. `#`
- `print_background` `<bool>` 打印背景图形。默认值为 `false`. `#`
- `scale` `<float>` 网页渲染的比例。默认为 `1`。刻度必须在0.1到 2 之间. `#`
- `width` `<str|float>` 纸张宽度, 接受单位标记的值. `#`
- `returns:` `<bytes>` `#`

返回PDF缓冲区。

#### NOTE

生成pdf目前只支持铬无头模式。

`page.pdf()` 生成一个带有打印CSS媒体的PDF页面, 在调用 `page.pdf()` 之前, 调用 `page.emulate_media(**kwargs)` 来生成带有屏幕媒体的pdf:



## NOTE

默认情况下, `page.pdf()` 会生成一个经过修改的`pdf`以供打印。使用 `-webkit-print-color-adjust` 属性强制渲染精确的颜色。

- Sync

```
# generates a pdf with "screen" media type.  
page.emulate_media(media="screen")  
page.pdf(path="page.pdf")
```

- Async

```
# generates a pdf with "screen" media type.  
await page.emulate_media(media="screen")  
await page.pdf(path="page.pdf")
```

宽度、高度和边距选项接受用单位标记的值。未标记的值被视为像素。

几个例子:

- `page.pdf({width: 100})` - 打印宽度设置为100像素
- `page.pdf({width: '100px'})` - 打印宽度设置为100像素
- `page.pdf({width: '10cm'})` - 打印宽度设置为10厘米。

所有可能的单位是:

- `px` - 像素
- `in` - 英寸
- `cm` - 厘米
- `mm` - 毫米

格式选项如下:

- `Letter` : 8.5in x 11in
- `Legal` : 8.5in x 14in
- `Tabloid` : 11in x 17in
- `Ledger` : 17in x 11in
- `A0` : 33.1in x 46.8in
- `A1` : 23.4in x 33.1in
- `A2` : 16.54in x 23.4in

- **A3** : 11.7in x 16.54in
- **A4** : 8.27in x 11.7in
- **A5** : 5.83in x 8.27in
- **A6** : 4.13in x 5.83in

#### NOTE

**header\_template** and **footer\_template** 标记有以下限制: > 1. 模板内的脚本标记不会被计算. > 2. 页面样式在模板中是不可见的.

## # page.press(selector, key, \*\*kwargs)#

- **selector** <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#) .#
- **key** <str> 要按下的 **键名** 或要生成的字符，如 **ArrowLeft** 或 **'a'** .#
- **delay** <float> **keydown** 和 **keyup** 之间的等待时间，单位是毫秒。默认为0. #
- **no\_wait\_after** <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 **false** .#
- **strict** <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常. #
- **timeout** <float> 最大时间，单位为毫秒，默认为30秒，通过 **0** 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改. #
- returns: <NoneType> .#

聚焦元素，然后使用 [keyboard.down\(key\)](#) and [keyboard.up\(key\)](#) .

**key**可以指定想要的 [keyboardEvent.key](#) 或是单个字符生成的文本,这里可以找到键值的超集。键的例子如下:

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

还支持以下快捷键: **Shift** , **Control** , **Alt** , **Meta** , **ShiftLeft** .

按住 `Shift` 键将输入与大写键对应的文本。

如果 `key` 是单个字符，它是区分大小写的，因此值 `a` 和 `A` 将生成不同的文本。

也支持快捷键，如键：“Control+o”或键：“Control+Shift+T”。当用修饰符指定时，修饰符被按下并被保持，而随后的键被按下。

- Sync

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

- Async

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

## # page.query\_selector(selector, \*\*kwargs) #

- `selector` `<str>` 要查询的选择器。有关更多细节，请参阅[working with selectors](#).#
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- returns: `<NoneType|ElementHandle>` #

CAUTION

不鼓励使用 [ElementHandle](#) 而是使用 [Locator](#) 对象和web优先断言

该方法查找页面中与指定选择器匹配的元素。如果没有元素匹配选择器，返回值解析为 `null`。若要等待页面上的元素，请使用 [locator.wait\\_for\(\\*\\*kwargs\)](#)。

主框架 [frame.query\\_selector\(selector, \\*\\*kwargs\)](#) 的快捷方式。

## # page.query\_selector\_all(selector) #

- `selector` <[str](#)> 要查询的选择器。有关更多细节，请参阅[working with selectors](#)。<#>
- returns: <[List](#)[[ElementHandle](#)]><#>

### CAUTION

不鼓励使用 [ElementHandle](#) 而是使用 [Locator](#) 对象和web优先断言

该方法查找页面中与指定选择器匹配的所有元素。如果没有元素匹配选择器，返回值解析为 `[]`。

主框架 [frame.query\\_selector\\_all\(selector\)](#) 的快捷方式。

## # page.reload(\*\*kwargs) #

- `timeout` <[float](#)>最大操作时间，单位为毫秒，默认为30秒，通过0表示禁止超时。默认值可以通过使用 [browser\\_context.set\\_default\\_navigation\\_timeout\(timeout\)](#), [browser\\_context.set\\_default\\_timeout\(timeout\)](#), [page.set\\_default\\_navigation\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来修改。<#>
- `wait_until` <"load"|"domcontentloaded"|"networkidle"|"commit"> 当认为操作成功时，默认为 `load`。事件可以是:<#>
  - `'domcontentloaded'` - 当 `domcontentloaded` 事件被触发时，认为操作已经完成。
  - `'load'` - 当触发 `load` 事件时，认为操作已经完成。
  - `'networkidle'` - 当至少 `500毫秒` 没有网络连接时，认为操作已经完成。

- `'commit'` - 当接收到网络响应并开始加载文档时，认为操作已经完成。
- `returns: <NoneType|Response>#`

这个方法重新加载当前页面，就像用户触发了浏览器刷新一样。返回主资源响应。在多个重定向的情况下，导航将使用最后一个重定向的响应进行解析。

## # `page.route(url, handler, **kwargs)#`

- `url` `<str|Pattern|Callable[URL]:bool>` 一个glob模式、regex模式或谓词在路由时接收要匹配的 `URL`. 当通过上下文选项提供了一个 `base_url` 并且传递的URL是一个路径时，它会通过 `new URL()` 构造函数合并。`#`
- `handler` `<Callable[Route, Request]>` handler函数路由请求。`#`
- `times` `<int>` 一个路由应该使用的频率。默认情况下，每次都会使用。`#`
- `returns: <NoneType>#`

路由提供了修改由页面发出的网络请求的功能。

一旦路由被启用，每一个匹配url模式的请求都会停止，除非它被继续、完成或中止。

### NOTE

只有当响应是重定向时，才会为第一个url调用处理程序。

### NOTE

`page.route(url, handler, **kwargs)` 不会拦截被Service Worker拦截的请求。查看 [这个问题](#)。我们建议在使用请求拦截时禁用 Service Workers。通过

```
await context.addInitScript(() => delete
window.navigator.serviceWorker);
```

一个简单的处理程序的例子，中止所有的图像请求：

- Sync

```
page = browser.new_page()
page.route("**/*.{png,jpg,jpeg}", lambda route: route.abort())
page.goto("https://example.com")
browser.close()
```

- Async

```
page = await browser.new_page()
await page.route("**/*.{png,jpg,jpeg}", lambda route:
route.abort())
await page.goto("https://example.com")
await browser.close()
```

或者使用regex模式替换相同的代码片段:

- Sync

```
page = browser.new_page()
page.route(re.compile(r"(\.png$)|(\.jpg$)"), lambda route:
route.abort())
page.goto("https://example.com")
browser.close()
```

- Async

```
page = await browser.new_page()
await page.route(re.compile(r"(\.png$)|(\.jpg$)"), lambda
route: route.abort())
await page.goto("https://example.com")
await browser.close()
```

可以通过检查请求来决定路由操作。例如，mock所有包含post数据的请求，并保留所有其他请求的原样:

- Sync

```
def handle_route(route):
    if ("my-string" in route.request.post_data)
        route.fulfill(body="mocked-data")
    else
        route.continue_()
page.route("/api/**", handle_route)
```

- Async

```
def handle_route(route):
    if ("my-string" in route.request.post_data)
        route.fulfill(body="mocked-data")
    else
        route.continue_()
await page.route("/api/**", handle_route)
```

当请求匹配两个处理程序时,页面路由优先于浏览器上下文路由(使用 [browser\\_context.route\(url, handler, \\*\\*kwargs\)](#) 设置)

要移除带有其处理程序的路由, 你可以使用 [page.unroute\(url, \\*\\*kwargs\)](#).

#### NOTE

启用路由将禁用`http`缓存.

## # page.screenshot(\*\*kwargs) #

- `animations` <"disabled"> 当设置为 `"disabled"` 时, 停止CSS动画, CSS转换和Web动画。动画根据其持续时间得到不同的处理: <#>
  - 有限动画是快进到完成, 所以他们会触发 `transitionend` 事件.
  - 无限动画被取消到初始状态, 然后在屏幕截图后播放.
- `clip` <[Dict](#)> 指定对结果图像进行剪辑的对象。应该有以下字段: <#>
  - `x` <[float](#)> 剪辑区域左上角的X坐标
  - `y` <[float](#)> 剪辑区域左上角的Y坐标
  - `width` <[float](#)> 剪辑区域的宽度
  - `height` <[float](#)> 剪辑区域的高度
- `full_page` <[bool](#)> 当为`true`时, 获取完整的可滚动页面的截图, 而不是当前可见的视口。默认值为 `false` [.#](#)

- `mask` <[List](#)[[Locator](#)]> 指定在截屏时应该被屏蔽的定位器。被屏蔽的元素将被一个粉红色的框覆盖#FF00FF，完全覆盖该元素。<#>
- `omit_background` <[bool](#)> 隐藏默认的白色背景，并允许透明捕捉屏幕截图。不适用于 `jpeg` 图像。默认值为 `false`。<#>
- `path` <[Union](#)[[str](#), [pathlib.Path](#)]> 保存镜像的文件路径，屏幕截图类型将从文件扩展名推断。如果 `path` 是一个相对路径，那么它是相对于当前工作目录解析的。如果没有提供路径，映像将不会被保存到磁盘。<#>
- `quality` <[int](#)> 图像的质量，在0-100之间。不适用于png图像。<#>
- `timeout` <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- `type` <"png"|"jpeg">指定截图类型，默认为 `png`。<#>
- returns: <[bytes](#)> <#>

返回带有捕获的截图的缓冲区。

## [# page.select\\_option\(selector, \\*\\*kwargs\)#](#)

- `selector` <[str](#)> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#)。<#>
- `force` <[bool](#)> 是否绕过[actionability](#)检查。默认值为 `false`。<#>
- `no_wait_after` <[bool](#)> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false`。<#>
- `strict` <[bool](#)> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。<#>
- `timeout` <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- `element` <[ElementHandle](#)|[List](#)[[ElementHandle](#)]> 要选择的选项。<#>
- `index` <[int](#)|[List](#)[[int](#)]> 按索引进行选择的选项。可选的。<#>
- `value` <[str](#)|[List](#)[[str](#)]> 按值选择的选项。如果 `<select>` 具有多个属性，则选择所有给定的选项，否则只选择与传递的选项之一匹配的第一个选项。可选的。<#>



- `label <str|List[str]>` 按标签进行选择的选项。如果 `\<select>` 具有多个属性，则选择所有给定的选项，否则只选择与传递的选项之一匹配的选项。可选的 `#`
- `returns: <List[str]>#`

这个方法等待元素匹配选择器，等待[可操作性](#)检查，直到所有指定的选项都出现在 `\<select>` 元素中，并选择这些选项。

如果目标元素不是 `\<select>` 元素，此方法将抛出一个错误。但是，如果该元素位于 `\<label>` 元素中，且该元素具有关联控件，则将使用该控件。

返回已成功选择的选项值的数组。

一旦选择了所有提供的选项，就触发一个更改和输入事件。

- Sync

```
# single selection matching the value
page.select_option("select#colors", "blue")
# single selection matching both the label
page.select_option("select#colors", label="blue")
# multiple selection
page.select_option("select#colors", value=["red", "green",
"blue"])
```

- Async

```
# single selection matching the value
await page.select_option("select#colors", "blue")
# single selection matching the label
await page.select_option("select#colors", label="blue")
# multiple selection
await page.select_option("select#colors", value=["red",
"green", "blue"])
```

主框架 [frame.select\\_option\(selector, \\*\\*kwargs\)](#) 的快捷方式。

**`# page.set_checked(selector, checked, **kwargs)#`**

---

- `selector` `<str>` 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#).#
- `checked` `<bool>` 是否选中或不选中复选框.#
- `force` `<bool>` 是否绕过[actionability](#)检查。默认值为 `false`.#
- `no_wait_after` `<bool>` 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false`.#
- `position` `<Dict>` 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点.#
  - `x` `<float>`
  - `y` `<float>`
- `strict` `<bool>` 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` `<float>` 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- `trial` `<bool>` 设置后，该方法只执行[actionability](#)检查，并跳过操作。默认值为 `false`。在元素准备好时再执行动作是很有用的.#
- `returns:` `<NoneType>`.#

这个方法通过执行以下步骤检查或取消检查元素匹配选择器：

1. 找到一个元素匹配选择器。如果没有，则等待直到匹配的元素被附加到DOM.
2. 确保匹配的元素是一个复选框或单选输入。如果不是，则排除此方法.
3. 如果元素已经具有正确的选中状态，则该方法立即返回.
4. 等待匹配元素的[actionability](#)检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作.
5. 如果需要，将元素滚动到视图中.
6. 使用 [page.mouse](#) 单击元素的中心.
7. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项.
8. 确保元素现在被选中或取消选中。如果不是，则抛出此方法.

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个[TimeoutError](#)。传递零超时将禁用此功能。

主框架 [frame.set\\_checked\(selector, checked, \\*\\*kwargs\)](#) 的快捷方式.

## # `page.set_content(html, **kwargs)` #

---

- `html` <[str](#)> 要分配给页面的html标记. #
- `timeout` <[float](#)> 最大操作时间，单位为毫秒，默认为30秒，通过0表示禁止超时。默认值可以通过使用 [browser\\_context.set\\_default\\_navigation\\_timeout\(timeout\)](#), [browser\\_context.set\\_default\\_timeout\(timeout\)](#), [page.set\\_default\\_navigation\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来修改. #
- `wait_until` <"load"|"domcontentloaded"|"networkidle"|"commit"> 当认为操作成功时，默认为 `load`。事件可以是: #
  - `'domcontentloaded'` - 当 `domcontentloaded` 事件被触发时，认为操作已经完成.
  - `'load'` - 当触发 `load` 事件时，认为操作已经完成.
  - `'networkidle'` - 当至少 `500毫秒` 没有网络连接时，认为操作已经完成.
  - `'commit'` - 当接收到网络响应并开始加载文档时，认为操作已经完成.
- returns: <[NoneType](#)> #

## # `page.set_default_navigation_timeout(timeout)` #

---

- `timeout` <[float](#)> 最大导航时间，以毫秒为单位 #
- returns: <[NoneType](#)> #

此设置将改变以下方法和相关快捷方式的默认最大导航时间：

- [page.go\\_back\(\\*\\*kwargs\)](#)
- [page.go\\_forward\(\\*\\*kwargs\)](#)
- [page.goto\(url, \\*\\*kwargs\)](#)
- [page.reload\(\\*\\*kwargs\)](#)
- [page.set\\_content\(html, \\*\\*kwargs\)](#)
- [page.expect\\_navigation\(\\*\\*kwargs\)](#)
- [page.wait\\_for\\_url\(url, \\*\\*kwargs\)](#)

NOTE

[page.set\\_default\\_navigation\\_timeout\(timeout\)](#) 优先于  
[page.set\\_default\\_timeout\(timeout\)](#),  
[browser\\_context.set\\_default\\_timeout\(timeout\)](#) 和  
[browser\\_context.set\\_default\\_navigation\\_timeout\(timeout\)](#).

## # [page.set\\_default\\_timeout\(timeout\)](#)

- `timeout` <[float](#)> 最大时间(毫秒) <#>
- returns: <[NoneType](#)> <#>

此设置将更改所有接受超时的方法的默认最大时间。

### NOTE

[page.set\\_default\\_navigation\\_timeout\(timeout\)](#) 优先于  
[page.set\\_default\\_timeout\(timeout\)](#).

## # [page.set\\_extra\\_http\\_headers\(headers\)](#)

- `headers` <[Dict\[str, str\]](#)> 包含每个请求发送的附加HTTP头的对象。  
所有头文件的值必须是字符串. <#>
- returns: <[NoneType](#)> <#>

额外的HTTP报头将随页面发起的每个请求一起发送。

### NOTE

[page.set\\_extra\\_http\\_headers\(headers\)](#) 不能保证传出请求中报头的顺序。

## # [page.set\\_input\\_files\(selector, files, \\*\\*kwargs\)](#)

- `selector` <[str](#)> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#). <#>

- `files` <[Union\[str, pathlib.Path\]](#)|[List\[Union\[str, pathlib.Path\]\]](#)|[Dict](#)|[List\[Dict\]](#)>#
  - `name` <[str](#)> File name
  - `mimeType` <[str](#)> File type
  - `buffer` <[bytes](#)> File content
- `no_wait_after` <[bool](#)> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false`。#
- `strict` <[bool](#)> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。#
- `timeout` <[float](#)> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。#
- returns: <[NoneType](#)>#

这个方法期望选择器指向一个[输入元素](#)。

将文件输入的值设置为这些文件路径或文件。如果某些 `filepath` 是相对路径，那么它们将相对于当前工作目录进行解析。对于空数组，清除选定的文件。

## # [page.set\\_viewport\\_size\(viewport\\_size\)](#)#

- `viewport_size` <[Dict](#)>#
  - `width` <[int](#)> 页面宽度(像素).
  - `height` <[int](#)> 页面高度(像素).
- returns: <[NoneType](#)>#

在一个浏览器中有多个页面的情况下，每个页面可以有自己的视口大小。然而，[browser.new\\_context\(\\*\\*kwargs\)](#) 允许在上下文中一次为所有页面设置视口大小(和更多)。

[page.set\\_viewport\\_size\(viewport\\_size\)](#) 将调整页面的大小. 很多网站不希望手机改变大小，所以你应该在导航到页面之前设置viewport的大小。

[page.set\\_viewport\\_size\(viewport\\_size\)](#) 也将重置屏幕大小，如果你需要更好地控制这些属性，使用 [browser.new\\_context\(\\*\\*kwargs\)](#) 与屏幕和viewport参数。

- Sync

```
page = browser.new_page()
page.set_viewport_size({"width": 640, "height": 480})
page.goto("https://example.com")
```

- Async

```
page = await browser.new_page()
await page.set_viewport_size({"width": 640, "height": 480})
await page.goto("https://example.com")
```

## # page.tap(selector, \*\*kwargs) #

- **selector** <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#)。[.#](#)
- **force** <bool> 是否绕过[actionability](#)检查。默认值为 **false**。[.#](#)
- **modifiers** <List["Alt"|"Control"|"Meta"|"Shift"]> modifiers 按键要按。确保在操作期间只按下这些修饰符，然后恢复当前的修饰符。如果未指定，则使用当前按下的修饰符。[.#](#)
- **no\_wait\_after** <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 **false**。[.#](#)
- **position** <Dict> 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点。[.#](#)
  - **x** <float>
  - **y** <float>
- **strict** <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。[.#](#)
- **timeout** <float> 最大时间，单位为毫秒，默认为30秒，通过 **0** 禁用超时。默认值可以通过使用[browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。[.#](#)
- **trial** <bool> 设置后，该方法只执行[actionability](#)检查，并跳过操作。默认值为 **false**。在元素准备好时再执行动作是很有用的。[.#](#)
- returns: <NoneType> [.#](#)

这个方法通过执行以下步骤点击元素匹配选择器：

1. 找到一个元素匹配选择器。如果没有，则等待直到匹配的元素被附加到 DOM.
2. 等待匹配元素的 [actionability](#) 检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作.
3. 如果需要，将元素滚动到视图中.
4. 点击页面中心或指定位置.
5. 等待已启动的导航成功或失败，除非设置了 `no_wait_after` 选项.

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个 [TimeoutError](#)。传递零超时将禁用此功能。

#### NOTE

[page.tap\(selector, \\*\\*kwargs\)](#) 需要将浏览器上下文的 `has_touch` 选项设置为 `true`.

主框架 [frame.tap\(selector, \\*\\*kwargs\)](#) 的快捷方式.

## # page.text\_content(selector, \*\*kwargs) #

---

- `selector` [<str>](#) 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅 [working with selectors](#) .#
- `strict` [<bool>](#) 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` [<float>](#) 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: [<NoneType|str>](#) .#

Returns `element.textContent` .

## # page.title() #

---

- returns: [<str>](#) .#

返回页面标题. 主框架 [frame.title\(\)](#) 的快捷方式.



## # `page.type(selector, text, **kwargs)` #

- `selector` <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#) .#
- `text` <str> 要输入到焦点元素中的文本.#
- `delay` <float> 按键之间的等待时间，单位是毫秒。默认为0.#
- `no_wait_after` <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 `false` .#
- `strict` <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常.#
- `timeout` <float> 最大时间，单位为毫秒，默认为30秒，通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改.#
- returns: <NoneType> #

为文本中的每个字符发送 `keydown` , `keypress` / `input` , and `keyup` 事件. `page.type` can be used to send fine-grained keyboard events. To fill values in form fields, use [page.fill\(selector, value, \\*\\*kwargs\)](#).

要按一个特殊的键，如 `Control` 或 `ArrowDown` ，使用 [keyboard.press\(key, \\*\\*kwargs\)](#).

- Sync

```
page.type("#mytextarea", "hello") # types instantly
page.type("#mytextarea", "world", delay=100) # types slower,
like a user
```

- Async

```
await page.type("#mytextarea", "hello") # types instantly
await page.type("#mytextarea", "world", delay=100) # types
slower, like a user
```

主框架 [frame.type\(selector, text, \\*\\*kwargs\)](#) 的快捷方式.



## # page.uncheck(selector, \*\*kwargs) #

- **selector** <str> 一个搜索元素的选择器。如果有多个元素满足选择器，将使用第一个元素。有关更多细节，请参阅[working with selectors](#)。[.#](#)
- **force** <bool> 是否绕过[actionability](#)检查。默认值为 **false**。[.#](#)
- **no\_wait\_after** <bool> 启动导航的操作正在等待这些导航发生，并等待页面开始加载。你可以通过设置这个标志来选择不等待。只有在特殊情况下才需要这个选项，比如导航到不可访问的页面。默认值为 **false**。[.#](#)
- **position** <Dict> 相对于元素填充框的左上角使用的一个点。如果没有指定，则使用元素的某个可见点。[.#](#)
  - **x** <float>
  - **y** <float>
- **strict** <bool> 当为true时，调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素，调用将抛出一个异常。[.#](#)
- **timeout** <float> 最大时间，单位为毫秒，默认为30秒，通过 **0** 禁用超时。默认值可以通过使用[browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。[.#](#)
- **trial** <bool> 设置后，该方法只执行[actionability](#)检查，并跳过操作。默认值为 **false**。在元素准备好时再执行动作是很有用的。[.#](#)
- **returns:** <NoneType> [.#](#)

这个方法通过执行以下步骤取消对元素匹配选择器的检查：

1. 找到一个元素匹配选择器。如果没有，则等待直到匹配的元素被附加到DOM.
2. 确保匹配的元素是一个复选框或单选输入。如果不是，则排除此方法。  
If the element is already unchecked, this method returns immediately.
3. 等待匹配元素的[actionability](#)检查，除非设置了强制选项。如果在检查期间分离了元素，则会重试整个操作.
4. 如果需要，将元素滚动到视图中.
5. 使用 [page.mouse](#) 单击元素的中心.
6. 等待已启动的导航成功或失败，除非设置了 **no\_wait\_after** 选项.
7. 确保元素现在是未选中的。如果不是，则排除此方法.

如果在指定的超时期间，所有步骤组合都没有完成，则该方法将抛出一个[TimeoutError](#)。传递零超时将禁用此功能。

主框架 [frame.uncheck\(selector, \\*\\*kwargs\)](#) 的快捷方式.

## # page.unroute(url, \*\*kwargs)#

---

- `url` <[str](#)|[Pattern](#)|[Callable](#)[[URL](#)]:[bool](#)> 一个glob Pattern, regex Pattern或predicate在路由时接收要匹配的url.#
- `handler` <[Callable](#)[[Route](#), [Request](#)]> Optional handler函数路由请求.#
- returns: <[NoneType](#)>#

移除使用 [page.route\(url, handler, \\*\\*kwargs\)](#) 创建的路由. 当未指定handler时, 删除url的所有路由.

## # page.url#

---

- returns: <[str](#)>#

主框架 [frame.url](#) 的快捷方式.

## # page.video#

---

- returns: <[NoneType](#)|[Video](#)>#

与此页相关联的视频对象.

## # page.viewport\_size#

---

- returns: <[NoneType](#)|[Dict](#)>#
  - `width` <[int](#)> 页面宽度(像素).
  - `height` <[int](#)> 页面高度(像素).

## # page.wait\_for\_event(event, \*\*kwargs)#

---

- `event` <[str](#)> 事件名称，与通常传递给 `*.on(event)` 的名称相同。<#>
- `predicate` <[Callable](#)> 接收事件数据，并在等待应该被解析时解析为真值。<#>
- `timeout` <[float](#)> 最大等待时间，单位为毫秒。默认为 `30000` (30秒)。通过0禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) 来更改。<#>
- returns: <[Any](#)> <#>

#### NOTE

在大多数情况下，您应该使用 [page.expect\\_event\(event, \\*\\*kwargs\)](#)。

等待给定事件被触发。如果提供了 `predicate`，它将 `event's` 的值传递给 `predicate` 函数，并等待 `predicate(event)` 返回一个真值。如果触发事件之前页面已关闭，则将抛出一个错误。

## # `page.wait_for_function(expression, **kwargs)` <#>

- `expression` <[str](#)> 要在浏览器上下文中计算的 `JavaScript` 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值。<#>
- `arg` <[EvaluationArgument](#)> 传递给 `expression` 的可选参数。<#>
- `polling` <[float](#)|`"raf"`> 如果 `polling` 是 `'raf'`，那么表达式会在 `requestAnimationFrame` 回调中持续执行。如果 `polling` 是一个数字，那么它将被视为执行函数的毫秒间隔。默认为 `raf`。<#>
- `timeout` <[float](#)> 最大等待时间，以毫秒为单位。默认为30000(30秒)。通过0禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) 来更改。<#>
- returns: <[JSHandle](#)> <#>

当表达式返回真值时返回。它解析为真值的 `jshandle`。

[page.wait\\_for\\_function\(expression, \\*\\*kwargs\)](#) 可用于观察视口大小的变化：

- Sync

```

from playwright.sync_api import sync_playwright

def run(playwright):
    webkit = playwright.webkit
    browser = webkit.launch()
    page = browser.new_page()
    page.evaluate("window.x = 0; setTimeout(() => { window.x = 100 }, 1000);")
    page.wait_for_function("() => window.x > 0")
    browser.close()

with sync_playwright() as playwright:
    run(playwright)

```

- Async

```

import asyncio
from playwright.async_api import async_playwright

async def run(playwright):
    webkit = playwright.webkit
    browser = await webkit.launch()
    page = await browser.new_page()
    await page.evaluate("window.x = 0; setTimeout(() => { window.x = 100 }, 1000);")
    await page.wait_for_function("() => window.x > 0")
    await browser.close()

async def main():
    async with async_playwright() as playwright:
        await run(playwright)

asyncio.run(main())

```

将参数传递给 [page.wait\\_for\\_function\(expression, \\*\\*kwargs\)](#) 函数的 predicate :

- Sync

```

selector = ".foo"
page.wait_for_function("selector =>
!!document.querySelector(selector)", selector)

```

- Async

```
selector = ".foo"
await page.wait_for_function("selector =>
!!document.querySelector(selector)", selector)
```

主框架 [frame.wait\\_for\\_function\(expression, \\*\\*kwargs\)](#) 的快捷方式。

## # page.wait\_for\_load\_state(\*\*kwargs) #

- **state** <"load"|"domcontentloaded"|"networkidle"> 可选加载状态，等待，默认为 **load**。如果在加载当前文档时已经达到该状态，该方法将立即进行解析。可以是：
  - **'load'** - 等待 **load** 事件被触发。
  - **'domcontentloaded'** - 等待 **domcontentloaded** 事件被触发。
  - **'networkidle'** - 等待至少500毫秒没有网络连接。
- **timeout** <float>最大操作时间，单位为毫秒，默认为30秒，通过0表示禁止超时。默认值可以通过使用 [browser\\_context.set\\_default\\_navigation\\_timeout\(timeout\)](#), [browser\\_context.set\\_default\\_timeout\(timeout\)](#), [page.set\\_default\\_navigation\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来修改。#
- **returns:** <[NoneType](#)> #

当达到所需的加载状态时返回

当页面达到所需的加载状态时，此问题将被解决，默认情况下加载。该导航必须在调用此方法时已提交。如果当前文档已经达到了所需的状态，则立即进行解析。

- Sync

```
page.click("button") # click triggers navigation.
page.wait_for_load_state() # the promise resolves after "load"
event.
```

- Async

```
await page.click("button") # click triggers navigation.
await page.wait_for_load_state() # the promise resolves after
"load" event.
```

- Sync

```
with page.expect_popup() as page_info:
    page.click("button") # click triggers a popup.
popup = page_info.value
# Following resolves after "domcontentloaded" event.
popup.wait_for_load_state("domcontentloaded")
print(popup.title()) # popup is ready to use.
```

- Async

```
async with page.expect_popup() as page_info:
    await page.click("button") # click triggers a popup.
popup = await page_info.value
# Following resolves after "domcontentloaded" event.
await popup.wait_for_load_state("domcontentloaded")
print(await popup.title()) # popup is ready to use.
```

主框架 [frame.wait\\_for\\_load\\_state\(\\*\\*kwargs\)](#) 的快捷方式。

## # page.wait\_for\_selector(selector, \*\*kwargs) #

---

- **selector** <[str](#)> 要查询的选择器。有关更多细节，请参阅[working with selectors](#)。<#>
- **state** <"attached"|"detached"|"visible"|"hidden"> 默认为 **"visible"**。可以是：
  - **'attached'** - 等待元素出现在DOM中。
  - **'detached'** - 等待元素在DOM中不存在。
  - **'visible'** - 等待元素有非空的边界框 且 没有 **visibility:hidden**。注意，没有任何内容或带有 **display:none** 的元素有一个空的边界框，因此不被认为是可见的。

- `'hidden'` - 等待元素从DOM中分离出来, 或有一个空的边界框或 `'visibility:hidden'`。这与 `"visible"` 选项相反。
- `strict` `<bool>` 当为true时, 调用要求选择器解析为单个元素。如果给定的选择器解析为多于一个元素, 调用将抛出一个异常。<#>
- `timeout` `<float>` 最大时间, 单位为毫秒, 默认为30秒, 通过 `0` 禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来更改。<#>
- returns: `<NoneType|ElementHandle>` <#>

当选择器指定的元素满足状态选项时返回。如果等待隐藏或分离, 则返回 `null`。

#### NOTE

*Playwright* 在执行动作前,会自动等待元素准备好。使用 [Locator](#) 对象和 *web-first*断言可以使代码不需要等待选择器。

等待选择器满足状态选项 (要么从dom中出现/消失, 要么变成可见/隐藏). 如果在调用方法选择器的时刻已经满足条件, 则该方法将立即返回。如果选择器不满足超时毫秒的条件, 函数将抛出。

此方法适用于多个导航:

- Sync

```
from playwright.sync_api import sync_playwright

def run(playwright):
    chromium = playwright.chromium
    browser = chromium.launch()
    page = browser.new_page()
    for current_url in ["https://google.com",
                        "https://bbc.com"]:
        page.goto(current_url, wait_until="domcontentloaded")
        element = page.wait_for_selector("img")
        print("Loaded image: " +
              str(element.get_attribute("src")))
    browser.close()

with sync_playwright() as playwright:
```

```
run(playwright)
```

- Async

```
import asyncio
from playwright.async_api import async_playwright

async def run(playwright):
    chromium = playwright.chromium
    browser = await chromium.launch()
    page = await browser.new_page()
    for current_url in ["https://google.com",
                        "https://bbc.com"]:
        await page.goto(current_url,
                        wait_until="domcontentloaded")
        element = await page.wait_for_selector("img")
        print("Loaded image: " + str(await
            element.get_attribute("src")))
        await browser.close()

async def main():
    async with async_playwright() as playwright:
        await run(playwright)
    asyncio.run(main())
```

## # `page.wait_for_timeout(timeout)` #

---

- `timeout` <float> 等待超时时间 #
- returns: <NoneType> #

等待给定超时(以毫秒为单位).

注意, `page.waitForTimeout()` 应该只用于调试。在生产中使用计时器的测试将是不可靠的。使用信号, 如网络事件, 选择器变得可见和其他.

- Sync

```
# wait for 1 second
page.wait_for_timeout(1000)
```



- Async

```
# wait for 1 second
await page.wait_for_timeout(1000)
```

主框架 [frame.wait\\_for\\_timeout\(timeout\)](#) 的快捷方式.

## # [page.wait\\_for\\_url\(url, \\*\\*kwargs\)](#) #

- **url** <[str](#)|[Pattern](#)|[Callable](#)[[URL](#)]:[bool](#)> 一个glob模式、regex模式或谓词，在等待导航时接收匹配的url。注意，如果参数是一个不带通配符的字符串，该方法将等待导航到与该字符串完全相等的URL。<#>
- **timeout** <[float](#)>最大操作时间，单位为毫秒，默认为30秒，通过0表示禁止超时。默认值可以通过使用 [browser\\_context.set\\_default\\_navigation\\_timeout\(timeout\)](#), [browser\\_context.set\\_default\\_timeout\(timeout\)](#), [page.set\\_default\\_navigation\\_timeout\(timeout\)](#) or [page.set\\_default\\_timeout\(timeout\)](#) 方法来修改。<#>
- **wait\_until** <"load"|"domcontentloaded"|"networkidle"|"commit"> 当认为操作成功时，默认为 **load**。事件可以是:<#>
  - **'domcontentloaded'** - 当 **domcontentloaded** 事件被触发时，认为操作已经完成。
  - **'load'** - 当触发 **load** 事件时，认为操作已经完成。
  - **'networkidle'** - 当至少 **500毫秒** 没有网络连接时，认为操作已经完成。
  - **'commit'** - 当接收到网络响应并开始加载文档时，认为操作已经完成。
- returns: <[NoneType](#)><#>

等待主框架导航到给定的URL.

- Sync

```
page.click("a.delayed-navigation") # clicking the link will
indirectly cause a navigation
page.wait_for_url("**/target.html")
```

- Async

```
await page.click("a.delayed-navigation") # clicking the link
will indirectly cause a navigation
await page.wait_for_url("**/target.html")
```

主框架 [frame.wait\\_for\\_url\(url, \\*\\*kwargs\)](#) 的快捷方式.

## # page.workers#

---

- returns: <[List](#)[[Worker](#)]>#

此方法返回与该页面关联的所有专用 [WebWorkers](#) .

*NOTE*

它不包含 *ServiceWorkers*

## # page.accessibility#

---

- type: <[Accessibility](#)>

## # page.keyboard#

---

- type: <[Keyboard](#)>

## # page.mouse#

---

- type: <[Mouse](#)>

## # page.request#

---

- type: <[APIRequestContext](#)>

与此页面相关的API测试助手。使用此API发出的请求将使用页面cookie.

## # page.touchscreen#

---

- type: <[Touchscreen](#)>

### Request

当页面向网络资源发送请求时，会触发以下事件序列 [Page](#)：

- [page.on\("request"\)](#) 当页面发出请求时.
- [page.on\("response"\)](#) 在/如果收到请求的响应状态和报头时触发.
- [page.on\("requestfinished"\)](#) 在下载响应体并完成请求时触发.

如果请求在某一时刻失败，则会触发 [page.on\("requestfailed"\)](#) 而不是'requestfinished'事件(可能也不是'response'事件).

#### NOTE

*HTTP*错误响应，如404或503，从*HTTP*的角度来看仍然是成功的响应，所以请求将通过'*requestfinished*'事件完成.

如果请求得到了一个“重定向”响应，该请求就会通过“requestfinished”事件成功完成，并向一个重定向url发出一个新请求.

- [request.all\\_headers\(\)](#)
- [request.failure](#)
- [request.frame](#)
- [request.header\\_value\(name\)](#)
- [request.headers](#)
- [request.headers\\_array\(\)](#)
- [request.is\\_navigation\\_request\(\)](#)
- [request.method](#)
- [request.post\\_data](#)
- [request.post\\_data\\_buffer](#)

- [request.post\\_data\\_json](#)
- [request.redirected\\_from](#)
- [request.redirected\\_to](#)
- [request.resource\\_type](#)
- [request.response\(\)](#)
- [request.sizes\(\)](#)
- [request.timing](#)
- [request.url](#)

## # request.all\_headers()#

---

- returns: <[Dict](#)[[str](#), [str](#)]>#

一个包含与此请求相关的所有请求HTTP头的对象。头属性名称是小写的。

## # request.failure#

---

- returns: <[NoneType](#)|[str](#)>#

该方法返回 `null`，除非请求失败，如`requestfailed`事件所报告的那样。

所有失败请求的日志记录示例：

```
page.on("requestfailed", lambda request: print(request.url + " " + request.failure))
```

## # request.frame#

---

- returns: <[Frame](#)>#

返回发起此请求的[Frame](#)。

## # request.header\_value(name)#

---

- `name` <[str](#)> 请求头名称.#

- returns: [`<NoneType|str>`](#) [`#`](#)

返回与名称匹配的头的值。名称不区分大小写。

## # request.headers#

---

- returns: [`<Dict\[str, str\]>`](#) [`#`](#)

**DEPRECATED** 由渲染引擎看到的不完整的头文件列表。使用 [`request.all\_headers\(\)`](#) 。

## # request.headers\_array()#

---

- returns: [`<List\[Dict\]>`](#) [`#`](#)
  - `name` [`<str>`](#) 请求头名称。
  - `value` [`<str>`](#) 请求头值。

包含与此请求相关的所有请求HTTP头的数组。与 [`request.all\_headers\(\)`](#) 不同，头属性名称不是小写的。具有多个条目的头属性，例如 `Set-Cookie`，在数组中会出现多次

## # request.is\_navigation\_request()#

---

- returns: [`<bool>`](#) [`#`](#)

这个请求是否驱动框架的导航。

## # request.method#

---

- returns: [`<str>`](#) [`#`](#)

请求的方法(GET、POST等)

## # request.post\_data#

---

- returns: <[NoneType](#)|[str](#)>#

请求的post body, 如果有的话

## # request.post\_data\_buffer#

---

- returns: <[NoneType](#)|[bytes](#)>#

请求体以二进制形式(如果有的话)表示

## # request.post\_data\_json#

---

- returns: <[NoneType](#)|[Serializable](#)>#

返回url编码的 `form-urlencoded` 和 JSON 解析后的请求体作为备用

当响应是 `application/x-www-form-urlencoded` 时, 值的键/值对象将被返回。否则它将被解析为JSON

## # request.redirected\_from#

---

- returns: <[NoneType](#)|[Request](#)>#

请求被服务器重定向到这个, 如果有的话

当服务器响应重定向时, Playwright 创建一个新的 [Request](#) 对象。这两个请求由 `redirectedFrom()` and `redirectedTo()` 方法连接。当发生多个服务器重定向时, 可以通过重复调用 `redirectedFrom()` 来构造整个重定向链。

例如, `http://example.com` 网站跳转到 `https://example.com`:

- Sync

```
response = page.goto("http://example.com")
print(response.request.redirected_from.url) #
"http://example.com"
```

- Async

```
response = await page.goto("http://example.com")
print(response.request.redirected_from.url) #
"http://example.com"
```

如果网站 `https://google.com` 没有重定向:

- Sync

```
response = page.goto("https://google.com")
print(response.request.redirected_from) # None
```

- Async

```
response = await page.goto("https://google.com")
print(response.request.redirected_from) # None
```

## # request.redirected\_to #

---

- returns: <[NoneType](#)|[Request](#)> #

如果服务器响应重定向，浏览器发出的新请求。

这个方法与 [request.redirected\\_from](#) 相反:

```
assert request.redirected_from.redirected_to == request
```

## # request.resource\_type #

---

- returns: <[str](#)> #

包含呈现引擎感知到的请求资源类型. `ResourceType` 将是以下其中之一:

`document`, `stylesheet`, `image`, `media`, `font`, `script`, `texttrack`,  
`xhr`, `fetch`, `eventsourcing`, `websocket`, `manifest`, `other`.

## # `request.response()` #

---

- returns: `<NoneType|Response>#`

返回匹配的 `Response` 对象, 如果由于错误没有接收到响应, 则返回 `null`

## # `request.sizes()` #

---

- returns: `<Dict>#`
  - `requestBodySize` `<int>` 请求正文(POST数据负载)的大小, 以字节为单位。如果没有正文, 则设置为0.
  - `requestHeadersSize` `<int>` 从HTTP请求消息开始直到(包括)正文前的双CRLF的总字节数
  - `responseBodySize` `<int>` 接收到的响应体大小(编码后), 单位为字节
  - `responseHeadersSize` `<int>` 从HTTP响应消息开始到(包括)正文前的双CRLF的总字节数

返回给定请求的资源大小信息

## # `request.timing` #

---

- returns: `<Dict>#`
  - `startTime` `<float>` 请求开始时间, 单位是毫秒, 自UTC 1月1日1970 00:00:00开始
  - `domainLookupStart` `<float>` 浏览器启动资源域名查找的紧接时间。该值以相对于 `startTime` 的毫秒为单位给出, 如果不可用则为-1.
  - `domainLookupEnd` `<float>` 浏览器启动资源域名查找后的即时时间。该值以相对于 `startTime` 的毫秒为单位给出, 如果不可用则为-1.



- `connectStart` <float> 用户代理开始建立与服务器的连接以检索资源之前的时间。该值以相对于 `startTime` 的毫秒为单位给出，如果不可用则为-1。
- `secureConnectionStart` <float> 浏览器启动握手进程以确保当前连接安全的时间。该值以相对于 `startTime` 的毫秒为单位给出，如果不可用则为-1。
- `connectEnd` <float> 用户代理开始建立与服务器的连接以检索资源之前的时间。该值以相对于 `startTime` 的毫秒为单位给出，如果不可用则为-1。
- `requestStart` <float> 浏览器从服务器、缓存或本地资源开始请求资源的时间。该值以相对于 `startTime` 的毫秒为单位给出，如果不可用则为-1。
- `responseStart` <float> 浏览器从服务器、缓存或本地资源开始请求资源的时间。该值以相对于 `startTime` 的毫秒为单位给出，如果不可用则为-1。
- `responseEnd` <float> 浏览器接收到资源的最后一个字节后或传输连接关闭前的时间，以先到的那个为例。该值以相对于 `startTime` 的毫秒为单位给出，如果不可用则为-1。

返回给定请求的资源计时信息。大多数计时值在响应时可用，`responseEnd` 在请求完成时可用。详情查看 [Resource Timing API](#)。

- Sync

```
with page.expect_event("requestfinished") as request_info:
    page.goto("http://example.com")
    request = request_info.value
    print(request.timing)
```

- Async

```
async with page.expect_event("requestfinished") as
    request_info:
        await page.goto("http://example.com")
    request = await request_info.value
    print(request.timing)
```

**# request.url#**

---

- returns: [<str>#](#)

请求的URL.

## Response

[Response](#) 表示页面接收的响应.

- [response.all\\_headers\(\)](#)
- [response.body\(\)](#)
- [response.finished\(\)](#)
- [response.frame](#)
- [response.header\\_value\(name\)](#)
- [response.header\\_values\(name\)](#)
- [response.headers](#)
- [response.headers\\_array\(\)](#)
- [response.json\(\)](#)
- [response.ok](#)
- [response.request](#)
- [response.security\\_details\(\)](#)
- [response.server\\_addr\(\)](#)
- [response.status](#)
- [response.status\\_text](#)
- [response.text\(\)](#)
- [response.url](#)

### # response.all\_headers()#

---

- returns: [<Dict\[str, str\]>#](#)

具有与此响应关联的所有响应HTTP头的对象

### # response.body()#

---

- returns: [<bytes>#](#)

返回带有响应体的缓冲区

## # response.finished()#

---

- returns: <[NoneType](#)|[str](#)>#

等待此响应完成，总是返回 `null`

## # response.frame#

---

- returns: <[Frame](#)>#

返回发起此响应的 [Frame](#) .

## # response.header\_value(name)#

---

- `name` <[str](#)> 请求头名称.#
- returns: <[NoneType](#)|[str](#)>#

返回与名称匹配的头的值。名称不区分大小写。如果多个报头有相同的名称(除了 `set-cookie`), 它们将被返回一个由 `,` 分隔的列表。对于 `set-cookie`, 使用 `\n` 分隔符。如果没有找到对应的响应头, 则返回 `null` .

## # response.header\_values(name)#

---

- `name` <[str](#)> 请求头名称.#
- returns: <[List](#)[[str](#)]>#

返回与名称匹配的头的值, 例如 `set-cookie` . 名称不区分大小写

## # response.headers#

---

- returns: <[Dict](#)[[str](#), [str](#)]>#

由渲染引擎看到的不完整的响应头列表。建议使用 [response.all\\_headers\(\)](#) 。

## # response.headers\_array()#

---

- returns: <[List](#)[[Dict](#)]>#
  - `name` <[str](#)> 请求头名称.
  - `value` <[str](#)> 请求头值.

包含与此响应关联的所有请求HTTP头的数组。与 [response.all\\_headers\(\)](#) 不同，头文件名称不是小写的。具有多个条目的头文件，例如 `Set-Cookie`，在数组中出现多次

## # response.json()#

---

- returns: <[Serializable](#)>#

返回响应体的JSON表示

如果响应体不能通过JSON.parse进行解析，则该方法将抛出。

## # response.ok#

---

- returns: <[bool](#)>#

包含一个布尔值，说明响应是否成功(状态在200-299之间)。

## # response.request#

---

- returns: <[Request](#)>#

返回匹配的 [Request](#) 对象。

## # response.security\_details()#

---

- returns: <[NoneType](#)|[Dict](#)>#
  - `issuer` <[str](#)> issuer字段的Common Name组件。从证书。这应该只用于信息目的。可选的
  - `protocol` <[str](#)> 所使用的TLS协议。(如TLS 1.3)。可选的
  - `subjectName` <[str](#)> Subject字段的Common Name组件。这应该只用于信息目的。可选的。
  - `validFrom` <[float](#)> Unix时间戳(以秒为单位), 指定此证书何时生效。可选的。
  - `validTo` <[float](#)> Unix时间戳(以秒为单位), 指定证书何时失效。可选的

返回SSL和其他安全信息

## # response.server\_addr()#

---

- returns: <[NoneType](#)|[Dict](#)>#
  - `ipAddress` <[str](#)> 服务器IPv4或IPV6地址
  - `port` <[int](#)>

返回服务器的IP地址和端口

## # response.status#

---

- returns: <[int](#)>#

包含响应的状态码(例如, 200表示成功)

## # response.status\_text#

---

- returns: <[str](#)>#

包含响应的状态文本(例如, 通常一个“OK”表示成功)

## # response.text()#

---

- returns: <[str](#)>#

返回响应体的文本表示形式

## # response.url#

---

- returns: <[str](#)>#

包含响应的URL

### Route

当一个网络路由页面被设置为 [page.route\(url, handler, \\*\\*kwargs\)](#) or [browser\\_context.route\(url, handler, \\*\\*kwargs\)](#), route对象允许处理该路由

- [route.abort\(\\*\\*kwargs\)](#)
- [route.continue\\_\(\\*\\*kwargs\)](#)
- [route.fulfill\(\\*\\*kwargs\)](#)
- [route.request](#)

## # route.abort(\*\*kwargs)#

---

- `error_code` <[str](#)> 可选错误码。默认为 `failed`, 可能是以下其中之一: #
  - `'aborted'` - 操作被中止(由于用户操作)
  - `'accessdenied'` - 访问资源的权限, 而不是网络, 被拒绝
  - `'addressunreachable'` - IP地址不可达。这通常意味着没有到指定主机或网络的路由
  - `'blockedbyclient'` - 客户端选择阻塞请求
  - `'blockedbyresponse'` - 请求失败, 因为响应是与不满足的需求一起交付的(例如'`X-Frame-Options`'和'`Content-Security-Policy`'祖先检查)

- `'connectionaborted'` - 由于没有收到ACK而导致的连接超时.
- `'connectionclosed'` - 连接被关闭(对应于一个TCP FIN).
- `'connectionfailed'` - 连接尝试失败.
- `'connectionrefused'` - 连接请求被拒绝.
- `'connectionreset'` - 连接被重置(对应于TCP RST).
- `'internetdisconnected'` - 网络连接中断.
- `'namenotresolved'` - 无法解析主机名.
- `'timedout'` - 操作超时.
- `'failed'` - 发生了一个普遍的故障.
- returns: `<NoneType>#`

中止路由的请求

## # route.continue\_(\*\*kwargs)#

---

- `headers` `<Dict[str, str]>` 改变请求的HTTP头。头文件的值将被转换为字符串.#
- `method` `<str>` 改变请求的方法(例如GET或POST)#
- `post_data` `<str|bytes>` 改变请求的post数据#
- `url` `<str>` 如果设置改变请求的url。新URL必须具有与原始URL相同的协议.#
- returns: `<NoneType>#`

用可选的覆盖继续路由的请求

- Sync

```
def handle(route, request):
    # override headers
    headers = {
        **request.headers,
        "foo": "bar" # set "foo" header
        "origin": None # remove "origin" header
    }
    route.continue_(headers=headers)
}

page.route("**/*", handle)
```

- Async

```
async def handle(route, request):
    # override headers
    headers = {
        **request.headers,
        "foo": "bar" # set "foo" header
        "origin": None # remove "origin" header
    }
    await route.continue_(headers=headers)
}

await page.route("**/*", handle)
```

## # route.fulfill(\*\*kwargs) #

- **body** <str|bytes> 响应正文. #
- **content\_type** <str> 设置 **Content-Type** 响应头. #
- **headers** <Dict[str, str]> 响应头。头文件的值将被转换为字符串. #
- **path** <Union[str, pathlib.Path]> 响应的文件路径。内容类型将从文件扩展名推断出来。如果path是一个相对路径，那么它是相对于当前工作目录解析的. #
- **response** <APIResponse> **APIResponse** 满足路由的请求。响应的各个字段(如报头)可以使用fill选项覆盖. #
- **status** <int> 响应状态码，默认为 **200**. #
- **returns:** <NoneType> #

用给定的响应来满足路由的请求

用 **404** 响应来完成所有请求的例子：

- Sync

```
page.route("**/*", lambda route: route.fulfill(
    status=404,
    content_type="text/plain",
    body="not found!"))
```

- Async



```
await page.route("**/*", lambda route: route.fulfill(
    status=404,
    content_type="text/plain",
    body="not found!"))
```

An example of serving static file:

- Sync

```
page.route("**/xhr_endpoint", lambda route:
    route.fulfill(path="mock_data.json"))
```

- Async

```
await page.route("**/xhr_endpoint", lambda route:
    route.fulfill(path="mock_data.json"))
```

## # route.request#

---

- returns: <[Request](#)>#

A request to be routed.

## Selectors

选择器可用于安装自定义选择器引擎。有关更多信息，请参见使用选择器 [Working with selectors](#)

- [selectors.register\(name, \\*\\*kwargs\)](#)

## # selectors.register(name, \*\*kwargs)#

---

- `name` <[str](#)> 在选择器中作为前缀使用的名称，例如 `{name: 'foo'}` 启用 `foo=myselectorbody` 选择器。只能包含 `[a-zA-`

`20-9_] 字符.#`

- `content_script` `<bool>` 是否在独立的JavaScript环境中运行该选择器引擎。这个环境可以访问相同的DOM，但不能访问框架脚本中的任何JavaScript对象。默认值为 `false`。请注意，当这个引擎与其他已注册的引擎一起使用时，不能保证作为一个内容脚本运行.#
- `path` `<Union[str, pathlib.Path]>` JavaScript文件的路径。如果path是一个相对路径，那么它是相对于当前工作目录解析的.#
- `script` `<str>` 原始脚本内容.#
- `returns:` `<NoneType>` #

注册选择器引擎的一个例子，它根据标签名查询元素：

- Sync

```
from playwright.sync_api import sync_playwright

def run(playwright):
    tag_selector = """
    {
        // Returns the first element matching given selector
        in the root's subtree.
        query(root, selector) {
            return root.querySelector(selector);
        },
        // Returns all elements matching given selector in
        the root's subtree.
        queryAll(root, selector) {
            return
Array.from(root.querySelectorAll(selector));
        }
    }"""

    # Register the engine. Selectors will be prefixed with
    "tag=".
    playwright.selectors.register("tag", tag_selector)
    browser = playwright.chromium.launch()
    page = browser.new_page()
    page.set_content('<div><button>Click me</button></div>')

    # Use the selector prefixed with its name.
    button = page.locator('tag=button')
    # Combine it with other selector engines.
```

```

page.click('tag=div >> text="Click me"')
# Can use it in any methods supporting selectors.
button_count = page.locator('tag=button').count()
print(button_count)
browser.close()

with sync_playwright() as playwright:
    run(playwright)

```

- Async

```

import asyncio
from playwright.async_api import async_playwright

async def run(playwright):
    tag_selector = """
    {
        // Returns the first element matching given selector
in the root's subtree.
        query(root, selector) {
            return root.querySelector(selector);
        },
        // Returns all elements matching given selector in
the root's subtree.
        queryAll(root, selector) {
            return
Array.from(root.querySelectorAll(selector));
        }
    }"""

    # Register the engine. Selectors will be prefixed with
"tag=".
    await playwright.selectors.register("tag", tag_selector)
    browser = await playwright.chromium.launch()
    page = await browser.new_page()
    await page.set_content('\<div>\<button>Click me\</button>\
</div>')

    # Use the selector prefixed with its name.
    button = await page.query_selector('tag=button')
    # Combine it with other selector engines.
    await page.click('tag=div >> text="Click me"')

```

```
# Can use it in any methods supporting selectors.
button_count = await page.locator('tag=button').count()
print(button_count)
await browser.close()

async def main():
    async with async_playwright() as playwright:
        await run(playwright)

asyncio.run(main())
```

## TimeoutError

- extends: [Error](#)

每当某些操作因超时而终止时，就会触发TimeoutError，例如 [locator.wait\\_for\(\\*\\*kwargs\)](#) or [browser\\_type.launch\(\\*\\*kwargs\)](#).

- Sync

```
from playwright.sync_api import sync_playwright, TimeoutError
as PlaywrightTimeoutError

with sync_playwright() as p:
    browser = p.chromium.launch()
    page = browser.new_page()
    try:
        page.click("text=Example", timeout=100)
    except PlaywrightTimeoutError:
        print("Timeout!")
    browser.close()
```

- Async

```
import asyncio
from playwright.async_api import async_playwright,
TimeoutError as PlaywrightTimeoutError
```

```

async def run(playwright):
    browser = await playwright.chromium.launch()
    page = await browser.new_page()
    try:
        await page.click("text=Example", timeout=100)
    except PlaywrightTimeoutError:
        print("Timeout!")
    await browser.close()

async def main():
    async with async_playwright() as playwright:
        await run(playwright)

asyncio.run(main())

```

## Touchscreen

触摸屏类操作的是相对于视口左上角CSS像素。触屏上的方法只能在已经初始化并设置为true的浏览器上下文中使用

- [touchscreen.tap\(x, y\)](#)

## # touchscreen.tap(x, y)#

---

- x <[float](#)>#
- y <[float](#)>#
- returns: <[NoneType](#)>#

在位置(x,y)分派一个触摸开始和触摸结束事件。

# Tracing

API用于收集和保存 Playwright的痕迹. Playwright 跟踪可以在 Playwright 脚本运行后在 [Trace Viewer](#) 中打开

在执行操作之前开始记录跟踪。最后，停止跟踪并将其保存到一个文件中

- Sync

```
browser = chromium.launch()
context = browser.new_context()
context.tracing.start(screenshots=True, snapshots=True)
page = context.new_page()
page.goto("https://playwright.dev")
context.tracing.stop(path = "trace.zip")
```

- Async

```
browser = await chromium.launch()
context = await browser.new_context()
await context.tracing.start(screenshots=True, snapshots=True)
page = await context.new_page()
await page.goto("https://playwright.dev")
await context.tracing.stop(path = "trace.zip")
```

- [tracing.start\(\\*\\*kwargs\)](#)
- [tracing.start\\_chunk\(\\*\\*kwargs\)](#)
- [tracing.stop\(\\*\\*kwargs\)](#)
- [tracing.stop\\_chunk\(\\*\\*kwargs\)](#)

## # tracing.start(\*\*kwargs) #

---

- **name** <str>如果指定了，跟踪将被保存到 [browser\\_type.launch\(\\*\\*kwargs\)](#) 中指定的 **traces\_dir** 文件夹中的文件中.#
- **screenshots** <bool> 跟踪时是否截图。截图是用来构建时间线预览的.#
- **snapshots** <bool> 如果该选项为true，跟踪将#

- 在每个动作上捕获DOM快照
- 记录网络活动
- `sources` `<bool>` 是否包含跟踪动作的源文件。<#>
- `title` `<str>` 要显示在跟踪查看器中的跟踪名称。<#>
- returns: `<NoneType>`<#>

开始跟踪

- Sync

```
context.tracing.start(name="trace", screenshots=True,
snapshots=True)
page = context.new_page()
page.goto("https://playwright.dev")
context.tracing.stop(path = "trace.zip")
```

- Async

```
await context.tracing.start(name="trace", screenshots=True,
snapshots=True)
page = await context.new_page()
await page.goto("https://playwright.dev")
await context.tracing.stop(path = "trace.zip")
```

## [# tracing.start\\_chunk\(\\*\\*kwargs\)#](#)

- `title` `<str>` 要显示在跟踪查看器中的跟踪名称。<#>
- returns: `<NoneType>`<#>

启动一个新的跟踪块。如果你想在同一个 [BrowserContext](#), 上记录多个跟踪, 只使用一次 [tracing.start\(\\*\\*kwargs\)](#) 然后使用 [tracing.start\\_chunk\(\\*\\*kwargs\)](#) and [tracing.stop\\_chunk\(\\*\\*kwargs\)](#) 创建多个跟踪块.

- Sync

```

context.tracing.start(name="trace", screenshots=True,
snapshots=True)
page = context.new_page()
page.goto("https://playwright.dev")

context.tracing.start_chunk()
page.click("text=Get Started")
# Everything between start_chunk and stop_chunk will be
recorded in the trace.
context.tracing.stop_chunk(path = "trace1.zip")

context.tracing.start_chunk()
page.goto("http://example.com")
# Save a second trace file with different actions.
context.tracing.stop_chunk(path = "trace2.zip")

```

- Async

```

await context.tracing.start(name="trace", screenshots=True,
snapshots=True)
page = await context.new_page()
await page.goto("https://playwright.dev")

await context.tracing.start_chunk()
await page.click("text=Get Started")
# Everything between start_chunk and stop_chunk will be
recorded in the trace.
await context.tracing.stop_chunk(path = "trace1.zip")

await context.tracing.start_chunk()
await page.goto("http://example.com")
# Save a second trace file with different actions.
await context.tracing.stop_chunk(path = "trace2.zip")

```

## # tracing.stop(\*\*kwargs)#

---

- `path` <[Union](#)[[str](#), [pathlib.Path](#)]> 将跟踪导出到指定路径下的文件中。<#>
- returns: <[NoneType](#)><#>



## # tracing.stop\_chunk(\*\*kwargs) #

---

- `path` <[Union\[str, pathlib.Path\]](#)> 将上次 [tracing.start\\_chunk\(\\*\\*kwargs\)](#) 调用后收集的跟踪信息导出到指定路径的文件中. #
- returns: <[NoneType](#)> #

停止跟踪块。有关多个跟踪块的详细信息，请参阅 [tracing.start\\_chunk\(\\*\\*kwargs\)](#)

### Video

当使用`recordVideo`选项创建浏览器上下文时，每个页面都有一个与之关联的视频对象

- Sync

```
print(page.video.path())
```

- Async

```
print(await page.video.path())
```

- [video.delete\(\)](#)
- [video.path\(\)](#)
- [video.save\\_as\(path\)](#)

## # video.delete() #

---

- returns: <[NoneType](#)> #

删除视频文件。如有必要，将等待视频完成

## # video.path() #

---

- returns: <[pathlib.Path](#)>#

返回视频录制到的文件系统路径。在关闭浏览器上下文时，保证视频被写入文件系统。此方法在远程连接时抛出

## # video.save\_as(path) #

---

- `path` <[Union](#)[[str](#), [pathlib.Path](#)]> 视频应该保存的路径.#
- returns: <[NoneType](#)>#

将视频保存到用户指定的路径。在视频仍在播放或页面已关闭时调用此方法是安全的。此方法将一直等待，直到页面关闭并完全保存视频

## WebSocket

[WebSocket](#) 表示页面中的WebSocket连接

- [web\\_socket.on\("close"\)](#)
- [web\\_socket.on\("framereceived"\)](#)
- [web\\_socket.on\("framesent"\)](#)
- [web\\_socket.on\("socketerror"\)](#)
- [web\\_socket.expect\\_event\(event, \\*\\*kwargs\)](#)
- [web\\_socket.is\\_closed\(\)](#)
- [web\\_socket.url](#)
- [web\\_socket.wait\\_for\\_event\(event, \\*\\*kwargs\)](#)

## # web\_socket.on("close") #

---

- type: <[WebSocket](#)>

websocket关闭时触发

## # web\_socket.on("framereceived")#

---

- type: <[Dict](#)>
  - `payload` <[str](#)|[bytes](#)> frame 有效负载

当websocket接收到一个frame 时触发.

## # web\_socket.on("framesent")#

---

- type: <[Dict](#)>
  - `payload` <[str](#)|[bytes](#)> frame 有效负载

当websocket发送一个frame 时触发

## # web\_socket.on("socketerror")#

---

- type: <[String]>

当websocket有错误时触发

## # web\_socket.expect\_event(event, \*\*kwargs)#

---

- `event` <[str](#)> 事件名称, 相同的将传递到 `WebSocket.on(event)`.#
- `predicate` <[Callable](#)> 接收事件数据, 并在等待应该被解析时解析为真值.#
- `timeout` <[float](#)> 最大等待时间, 单位为毫秒。默认为 `30000` (30秒)。通过0禁用超时。默认值可以通过使用 `browser_context.set_default_timeout(timeout)` 来更改.#
- returns: <[EventManager](#)>.#

等待事件触发, 并将其值传递给 `predicate` 函数.当 `predicate` 返回真值时返回。如果WebSocket在事件被触发之前关闭, 将抛出一个错误。返回事件数据值

## # web\_socket.is\_closed()#

---

- returns: <[bool](#)>#

web socket已经关闭

## # web\_socket.url#

---

- returns: <[str](#)>#

包含WebSocket的URL

## # web\_socket.wait\_for\_event(event, \*\*kwargs)#

---

- **event** <[str](#)> 事件名称，与通常传递给 **\*.on(event)** 的名称相同.#
- **predicate** <[Callable](#)> 接收事件数据，并在等待应该被解析时解析为真值.#
- **timeout** <[float](#)> 最大等待时间，单位为毫秒。默认为 **30000** (30秒)。通过0禁用超时。默认值可以通过使用 [browser\\_context.set\\_default\\_timeout\(timeout\)](#) 来更改.#
- returns: <[Any](#)>#

### NOTE

在大多数情况下，你应该使用 [web\\_socket.expect\\_event\(event, \\*\\*kwargs\)](#)。

等待给定事件被触发。如果提供了 **predicate**，它将 **event's** 的值传递给 **predicate** 函数，并等待 **predicate(event)** 返回一个真值。如果套接字在触发事件之前关闭，则将抛出一个错误

## Worker

Worker 类代表一个 [WebWorker](#). 在页面对象上发出Worker事件，以通知一个Worker创建。关闭事件在worker对象消失时被触发

```
def handle_worker(worker):
    print("worker created: " + worker.url)
    worker.on("close", lambda: print("worker destroyed: " +
    worker.url))

page.on('worker', handle_worker)

print("current workers:")
for worker in page.workers:
    print("    " + worker.url)
```

- [worker.on\("close"\)](#)
- [worker.evaluate\(expression, \\*\\*kwargs\)](#)
- [worker.evaluate\\_handle\(expression, \\*\\*kwargs\)](#)
- [worker.url](#)

### # worker.on("close")#

---

- type: <[Worker](#)>

当这个专用的[WebWorker](#) 被终止时触发.

### # worker.evaluate(expression, \*\*kwargs)#

---

- **expression** <[str](#)> 要在浏览器上下文中计算的 [JavaScript](#) 表达式。如果它看起来像一个函数声明，它会被解释为一个函数。否则，作为表达式求值.#
- **arg** <[EvaluationArgument](#)> 传递给 **expression** 的可选参数.#
- returns: <[Serializable](#)>#

返回表达式的返回值.

如果函数传递给 `worker.evaluate(expression, **kwargs)` 返回一个 `Promise`, 则 `worker.evaluate(expression, **kwargs)` 将等待promise解析并返回它的值.

如果函数传递给 `worker.evaluate(expression, **kwargs)` 返回一个不可序列化 non-`Serializable` 的值, 则 `worker.evaluate(expression, **kwargs)` 返回 `undefined`. Playwright 还支持传递一些附加值, 不能通过JSON序列化: `-0`, `NaN`, `Infinity`, `-Infinity`.

## # `worker.evaluate_handle(expression, **kwargs)` #

---

- `expression` `<str>` 要在浏览器上下文中计算的 `JavaScript` 表达式。如果它看起来像一个函数声明, 它会被解释为一个函数。否则, 作为表达式求值. #
- `arg` `<EvaluationArgument>` 传递给 `expression` 的可选参数. #
- returns: `<JSHandle>` #

返回表达式的返回值是一个 `JSHandle`.

`worker.evaluate(expression, **kwargs)` and `worker.evaluate_handle(expression, **kwargs)` 之间唯一的区别是 `worker.evaluate_handle(expression, **kwargs)` 返回 `JSHandle`.

如果函数传递给 `worker.evaluate_handle(expression, **kwargs)` 返回一个 `Promise`, 则 `worker.evaluate_handle(expression, **kwargs)` 将等待promise解析并返回它的值.

## # `worker.url` #

---

- returns: `<str>` #