splash 中文文档 Documentation

发布 *0.1*

masimaro

2020 年 04 月 06 日

Contents

spalsh 提供 JavaScript 渲染服务，它是一个使用 Twisted 和 QT5 在 Python 3 中实现的支持 HTTP API 调用的轻量级的 web 浏览器。它使用 Twisted 和 QT 的反射机制以使服务完全异步并通过 QT 主循环以便利用 webkit 并发性

> 这段话的原文是"The (twisted) QT reactor is used to make the service fully asynchronous allowing to take advantage of webkit concurrency via QT main loop." 这里我实在是不知道怎么样把这个意思表达清楚，只能根据我的理解做适当的修改，可能表述不当

Splash 的部分特征:

- 并行处理多个页面
- 获取返回的 HTML 代码或者获取返回页面的截屏图片
- 通过禁止图片加载或者使用 Adblock Plus 插件来提高加载页面的速度
- 在页面的上下文中执行用户的 JavaScript 代码
- 编写 lua 脚本来操作浏览器
- 在 Splash-Jupyter 中支持 lua 脚本
- 在格式化的 HAR 数据中获取渲染的相关细节

CHAPTER 1

文档目录

1.1 安装

1.1.1 linux + dockeer

1. 下载 docker
2. 拉取镜像:

```
$ sudo docker pull scrapinghub/splash
```

3. 启动容器:

```
$ sudo docker run -p 8050:8050 -p 5023:5023 scrapinghub/splash
```

4. 现在 splash 在 0.0.0.0 这个 ip 上监听并绑定了端口 8050(http) 和 5023 (telnet)

1.1.2 OS X + Docker

1. 在 Mac 上安装 docker (请参考:https://docs.docker.com/docker-for-mac/)
2. 拉取镜像:

```
$ sudo docker pull scrapinghub/splash
```

3. 启动容器:

```
$ sudo docker run -p 8050:8050 -p 5023:5023 scrapinghub/splash
```

4. 现在 splash 在 0.0.0.0 这个 ip 上监听并绑定了端口 8050(http) 和 5023 (telnet)

1.1.3 Ubuntu 16.04(手工安装)

警告: 在桌面系统上使用 docker 来安装是一种更方便的方式，如果您选择使用手工安装，请至少先阅读项目中的 provision.sh 脚本文件

1. 从 GitHub 中克隆仓库:

```
git clone https://github.com/scrapinghub/splash/
```

2. 下载依赖项:

```
$ cd splash/dockerfiles/splash
$ sudo cp ./qt-installer-noninteractive.qs /tmp/script.qs
$ sudo ./provision.sh \
    prepare_install \
    install_msfonts \
    install_extra_fonts \
    install_deps \
    install_flash \
    install_qtwebkit_deps \
    install_official_qt \
    install_qtwebkit \
    install_pyqt5 \
    install_python_deps
```

3. 切回到 splash 的父目录比如 cd ~ 然后运行:

```
$ sudo pip3 install splash/
```

运行下面的命令来使服务启动起来:

```
python3 -m splash.server
```

运行 `python3 -m splash.server --help` 查看更多可能的操作默认情况下 splash API 在对应机器 IPv4 的 8050 端口监听，要修改这个端口请使用 `--port` 参数:

```
python3 -m splash.server --port=5000
```

注解: docker 官方镜像使用的是 Ubuntu 16.04；使用这种方式执行与使用 Dockerfile 执行，这二者命令行的使用方式基本相同，主要的不同点在于前者移除了 provision.sh 这个比较危险的脚本文件，这个文件中的相关命令也就不再使用了。它们二者都允许将相关内容保存在 docker 镜像中。从而打破在桌面系统中各种软件的相关依赖[6]

依赖的 python 包

```
# install PyQt5 (Splash is tested on PyQT 5.9)
# and the following packages:
twisted >= 15.5.0, < 16.3.0
qt5reactor
psutil
adblockparser >= 0.5
https://github.com/sunu/pyre2/archive/c610be52c3b5379b257d56fc0669d022fd70082a.zip#egg=re2
xvfbwrapper
Pillow > 2.0
# for scripting support
lupa >= 1.3
funcparserlib >= 0.3.6
```

1.1.4 Splash 版本

执行命令 `docker pull scrapinghub/splash` 将会得到 splash 的最新的稳定版，执行命令 `docker pull scrapinghub/splash:master` 可以获取到最新的开发版。当然，你也可以获取指定的版本比如执行命令 `docker pull scrapinghub/splash:2.3.3`

1.1.5 定制 docker 中的 splash[7]

定制启动

可以在 docker 的启动命令 `docker run` 在之后的镜像名称后加上对应的参数，就可以实现定制启动。比如执行下面的命令来降低日志等级

```
$ docker run -p 8050:8050 scrapinghub/splash -v3
```

[6] 后面一句的原文是:" they allow to save space in a Docker image, but can break unrelated software on a desktop system." 它的原本意思是什么我也不太清楚，不知道怎么才能表述清楚，我猜测它的意思应该是在 docker 中安装相关镜像只需要简单的拉取下来，而直接安装的话需要对应的依赖环境

[7] 原文是 Customizing Dockerized Splash, 这个 Dockerized 我不知道该怎么翻译

使用 --help 获取可用的选项，并不是所有的命令都在同一个 docker 环境中都适用：修改端口可能没有意义（使用命令的方式修改启动端口）。它的路径使用 docker 容器中的路径[8]

共享目录

使用 docker 的 -v 参数可以 自定义请求包过滤[9] ，首先在本地的文件系统[10] 中为请求过滤创建一个文件，然后为它赋对应的权限，以便容器能够访问它.

```
$ docker run -p 8050:8050 -v <my-filters-dir>:/etc/splash/filters scrapinghub/splash
```

将 <my-filters-dir> 替换成你用来进行请求过滤的文件所在的路径也可以使用 docker 容器的 Data Volume 作为对应的过滤文件，点击 https://docs.docker.com/userguide/dockervolumes/ 查看更多信息

代理 和 JavaScript 的相关设置 也可以使用相同的方式

```
$ docker run -p 8050:8050 \
    -v <my-proxy-profiles-dir>:/etc/splash/proxy-profiles \
    -v <my-js-profiles-dir>:/etc/splash/js-profiles \
    scrapinghub/splash
```

你可以在路径 /etc/splash/lua_modules 中挂载 你自己的模块 ，如果使用 'Lua sandbox <./scripting-tutorial.html#lua-sandbox>'_ （默认）不要忘了在命令参数 --lua-sandbox-allowed-modules 后列举上安全的模块

```
$ docker run -p 8050:8050 \
    -v <my-lua-modules-dir>:/etc/splash/lua_modules \
    scrapinghub/splash \
    --lua-sandbox-allowed-modules 'module1;module2'
```

警告: 在 OS X 和 Windows 平台上使用共享文件 (-v 选项) 还有一些问题 (参考链接: https://github.com/docker/docker/issues/4023) 如果你在使用时有相关问题，请尝试在该链接中提到的解决方法，或者克隆项目，修改 Dockerfile。

编译本地的 Docker 镜像

可以使用 git 的 checkout 命令检出 Splash 的 源代码 然后在 Splash 的根目录下执行命令来编译本地的 Docker 镜像

[8] 这里我觉得它的意思是不能通过命令行的方式修改端口和路径，它们的值应该是事先设定好的

[9] 请求包过滤是直译，这里的意思应该是对请求包进行过滤，确定哪些可以发，哪些不能发

[10] 在 Windows 中如果使用 docker toolbox 安装的，那么是在 docker 对应的虚拟机中，如果是直接安装的 docker，则是在本地操作系统中

```
$ docker build -t my-local-splash .
```

使用如下命令编译 'Splash-Jupyter <./kernel.html#splash-jupyter>'_ 的 Docker 镜像

```
$ docker build -t my-local-splash-jupyter -f  dockerfiles/splash-jupyter/Dockerfile .
```

如果你想基于本地的 Splash Docker 容器来编译, 可能需要修改 `dockerfiles/splash-jupyter/Dockerfile` 为对应的路径

1.2 Splash HTTP API

请先参阅 安装 部分的内容安装，然后启动 Splash

splash 是通过 HTTP API 来进行操作的。对于下面列举出来的所有端点都可以采用 GET 方式传入参数，或者将参数编码为 JSON 格式并使用 `Content-Type: application/json` 请求头使用 POST 方式发出

多数 splash 端点都提供了 run 和 execute 两种功能，这意味着它们能执行自定义的任意的 Lua 渲染脚本

另外的一些端点可能用于某些特殊的场合，比如 render.png 可以在你不提供任何进一步处理的情况下使用 PNG 格式返回一个网页截图，另外如果你不需要与页面进行交互那么 render.json 将会使程序编写变得更为方便[10]

1.2.1 render.html

返回一个经过 Javascript 渲染之后的页面的 HTML 代码参数:

url: string: required[11] 需要进行渲染的页面的 url，这个参数必须提供

baseurl : string : optional 用于呈现页面的基础 URL

> 基本 HTML 内容将从 url 参数中提供的 URL 中获取，而用于呈现页面的 HTML 文本中的相对引用资源是使用 baseurl 参数中给定的 URL 作为基础获取的[12] 。您可以在这个讨论中获取更多信息: render.html 返回好像被浏览器给破坏了

timeout : float : optional 渲染的超时值，以秒为单位 (默认为 30s)

> 默认情况下，允许的最大超时值为 90s，您也可以在启动时通过 `--wait-timeout` 参数来修改这个值，比如使用这个命令来使默认最大超时时间为 5 分钟
>
> ```
> $ docker run -it -p 8050:8050 scrapinghub/splash --max-timeout 300
> ```

resource_timeout : float : optional 单个网络请求的超时时间

[10] 从后面对它的介绍可以看到，它会将响应的相关内容转化为 JSON 格式，这样我们就能很方便的进行解析了

[11] 这里的格式为: 参数名: 数据类型: 参数类型, 参数类型有两种 required 表示必须提供，optional 表示可选

[12] 这里它针对的是相对路径的 URI，它会以 baseurl 作为基础最终拼接成一个完整的路径

更多信息请查看: splash:on_request 的 `request:set_timeout(timeout)` 方法和 splash.resource_timeout 属性

wait : float : optional 当收到响应包后等待的时长，单位为 s 默认为 0，如果您所请求的页面中包含一些异步与延时加载的 JavaScript 脚本时请添加上这个值，当等待时间为 0 时这些 JavaScript 代码不会执行。当你想要获取整个页面的 PNG 和 JPEG 图片的话，最好也加上此值（请查看*render_all* ）

这个等待值必须小于 timeout 这个超时值[13]

proxy : string : optional 指定代理配置文件的名称，或者代理 url。参阅代理配置

代理 url 的格式为: `[protocol://][user:password@]proxyhost[:port])`

其中 protocol 为 http 或者 socks5, 如果未指定端口，将会默认采用 1080 端口

js : string : optional JavaScript 配置文件名称，请参阅 JavaScript 配置

js_source : string : optional 可被页面环境执行的 JavaScript 代码。请参阅: 使用页面环境执行 JavaScript 代码

filters : string : optional 使用分号分隔的请求过滤的名称列表，请参阅请求包过滤

allowed_domains : string : optional 使用分号分隔的允许访问的域名列表。如果该值存在，Splash 将不会加载任何来自不在此列表中的域以及不在此列表中的域的子域的任何内容。

allowed_content_types : string : optional 使用分号分隔的允许内容类型列表。当该值存在时，如果请求包对应的响应包类型不在列表中那么该请求包将会被拒绝。允许内容的通配符使用 fnmatch 语法

forbidden_content_types 使用分号分隔的拒绝内容类型列表。当该值存在时，如果请求包对应的响应包类型在列表中那么该请求包将会被拒绝。允许内容的通配符使用 fnmatch 语法

viewport : string : optional 用来渲染 js 的浏览器视口的大小，主要是宽和高, 该值单位为像素, 格式为”< 宽 >x< 高 >”, 比如 800x600，默认值为 1024x768.

这个值在生成 PNG 和 JPEG 的情况下十分重要，所有渲染端点都支持这个参数，因为 JavaScript 代码的执行以视口大小为依据

出于向后兼容的考虑，它允许使用 full 为值 `viewport=full` 它的效果与使用 `render_all=1` 相同 (请参阅: render_all)

images : integer : optional 是否加载图片，当值为 1 时表示允许加载图片，为 0 时表示禁止加载

在某些情况下即使设置了值为 0，也会加载图片，你也可以使用请求包过滤的方式根据 url 来屏蔽不想看见的内容

headers : JSON array or object : optional 为首个发出去的 http 请求包设置请求头

这个参数仅仅在 `application/json` 类型的 POST 包中使用，它可以是使用 (header_name, header_value) 这种格式的数据组成的 json 对象，其中 header_name 表示请求头某项的键，header_value 表示请求头某项的值

其中“User-Agent”这个头比较特殊，它作用在所有请求包上而不仅仅是首个包

[13] 超时值是指发送请求到接受请求并返回的所有时间之和，也就是说它包含了 wait 值在内

body : string : optional 如果 HTTP 请求方式为 POST，那么该值将作为请求体，此时默认的 content-type 请求头为 `application/x-www-form-urlencoded`

http_method : string : optional 传出的 Splash 包的请求方法[14] ，默认的方法是 GET，当然 Splash 也支持 POST。

save_args : JSON 数据或者是一个以分号为分隔符的字符串: optional 这是一个放入缓存中的参数名称列表，Splash 将会把列表中对应的参数值放入到内部缓冲中，并通过 Splash 响应头的 `X-Splash-Saved-Arguments` 参数中进行返回，该参数会将对应值以 SHA1 列表的方式返回。这个返回值是一个以分号分隔的字符串，每个部分以键 = 哈希值这种方式展现

```
name1=9a6747fc6259aa374ab4e1bb03074b6ec672cf99;name2=ba001160ef96fe2a3f938fea9e6762e204a562b3
```

在客户端中可以使用 *load_args* 参数将响应包头部的对应哈希值转化为真实的参数值。当参数值较长而且不变的情况下使用这种方式将会是一个很好的选择（特别是在表示 js_source 和 lua_source 的时候）

load_args [JSON 对象或者是一个字符串][optional] 将参数值从缓存中加载出来，load_args 参数值必须是 `{"name": "<SHA1 hash>", ...}` 格式的 json 对象或者是响应包头的 `X-Splash-Saved-Arguments` 参数所对应的原始字符 (以分号分隔的 name=hash 格式的字符串)

针对每个存在在 load_args 中的参数，Splash 在取出对应的值的时候会使用 hash 值作为键值，从缓存中查找出对应的真实数据，如果对应的 hash 值在缓存中能够找到相应的值，那么会将这个找到的值作为参数的真实值，然后向往常一样处理请求

如果在缓存中没有找到对应的值，那么 Splash 会返回一个 HTTP 498 状态码。在这种情况下客户端需要再次使用 save_args 传入完整的参数值并提交 HTTP 请求

Splash 通过 load_args 和 save_args 参数的方式，在请求中不发送每个请求的大参数，以便达到节约网络流量的目的（通常在带有 js_source 和 lua_source 的参数中使用将会是一个很好的选择）

splash 使用 LUR 缓存来存储这些值, 在存储时限定了参数的条目数量，并且在每次重启 Splash 之后都会清理缓存，换句话说，Splash 中的缓存不是持久性的客户端应该要有重发这些参数的操作

html5_media : integer : optional 是否支持 H5 中的多媒体（比如 <video> 标签）。使用 1 表示支持，0 表示不支持，默认为 0

Splash 默认是不支持 H5 多媒体的，它可能会造成程序的不稳定。在未来的版本中可能会默认支持 H5，所以在那以后如果不需要使用 H5，那么请将参数设置为 0 `html5_media = 0`

更多信息请参阅 splash.html5_media_enabled .

示例

curl 示例

[14] 注意，这里的请求方法是指由 Splash 发出去的请求包的请求方法，使用 Splash 进行渲染的过程实际上是分两步走的，第一步是向 Splash 发送请求包，然后由 Splash 向对应目标发送请求包，接着由 Splash 接收响应包，并渲染最后返给程序。不要理解成了向 Splash 发包的请求方式

```
curl 'http://localhost:8050/render.html?url=http://domain.com/page-with-javascript.html&timeout=10&
↪wait=0.5'
```

返回的数据包都被编码为 UTF-8，render.html 端点会将返回的 HTML 也编码为 UTF-8，哪怕是在 HTML 的标签想下面这样中明确指定了编码方式

```
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
```

1.2.2 render.png

将页面渲染的结果以图片的方式返回（格式为 png）

参数:

它的许多参数都与 render.html 的相同, 相比较于前者它多出来下面几个参数

width : integer : optional 将生成图片宽度调整为指定宽度，以保持宽高比

height : integer : optional 将生成的图片裁剪到指定的高度，通常与 width 参数一起使用以生成固定大小的图片

render_all : int : optional 它可能的值有 0 和 1，表示在渲染前扩展视口以容纳整个 Web 页面 (即使整个页面很长)，默认值为 `render_all=0`

注解: render_all = 1 时需要一个不为 0 的 wait 值，这是一个不幸的限制，但是目前来看只能通过这种方式使得在 `render_all = 1` 这种情况下整个渲染变得可靠

---

scale_method : string : optional 可能的值有 `raster` (默认值) 和 `vector`, 如果值为 raster, 通过宽度执行的缩放操作是逐像素的，如果值为 vector, 在缩放是是按照元素在进行的[15]

注解: 基于矢量的重新缩放更加高效，并且会产生更清晰的字体和更锐利的元素边界，但是可能存在渲染问题，请谨慎使用

示例

curl 示例

```
# 使用超时值进行渲染
curl 'http://localhost:8050/render.png?url=http://domain.com/page-with-javascript.html&timeout=10'
```

(continues on next page)

[15] 这里我的理解是一个进行的是位图的变换，一个是进行矢量图的变换

(续上页)

```
# 将生成图片尺寸设置为:320x240
curl 'http://localhost:8050/render.png?url=http://domain.com/page-with-javascript.html&width=320&
↪height=240'
```

1.2.3 render.jpeg

将页面渲染的结果以图片的方式返回（格式为 jpeg）

参数:

它的参数与 render.png 大致相同，相比于前者，它多出一个参数

quality : integer : optional 该参数表示生成图片的质量，大小在 0~100 之前，默认值为 75

> **注解:** 该值应该尽量避免高于 95，当 `quality=100` 时，会禁用 JPEG 的相关压缩算法，导致大量的图片实际上得不到质量的提升

示例

curl 示例

```
# 生成默认质量的图片
curl 'http://localhost:8050/render.jpeg?url=http://domain.com/'

# 生成高质量的图片
curl 'http://localhost:8050/render.jpeg?url=http://domain.com/&quality=30'
```

1.2.4 render.har

以 HAR 格式返回 Splash 与目标站点的交互信息，里面包含了请求信息、响应信息、时间信息和头信息等等

您可以使用在线的 HAR 查看工具 来查看该端点返回的具体信息。这些信息与我们使用 Chrome 和 FirFox 等浏览器的 Network 工具得到的信息十分相似

目前这个端点不会公开原始的请求信息，目前只有一些元数据信息比如包头信息和时间信息是可用的，只有当‘response_body’参数被设置为 1 的时候才会包含响应体的信息

它的参数与 render.html 相似，多出来的参数如下:

response_body : int : optional 可选的值有 0 和 1，当值为 1 时，响应体的信息会被包含在返回的 HAR 数据中，默认情况下 `response_body = 0`

1.2.5 render.json

将经过 JavaScript 渲染的页面信息以 json 格式返回，它可以返回 HTML，PNG 等其他信息。返回何种信息由相关参数指定

参数:

参数与 render.jpeg 的参数相似，多余的参数如下:

html : integer : optional 返回值中是否包含 HTML，1 为包含，0 表示不包含，默认为 0

png : integer : optional 返回值中是否包含 PNG 图片，1 为包含，0 表示不包含，默认为 0

jpeg : integer : optional 返回值中是否包含 JPEG 图片，1 为包含，0 表示不包含，默认为 0

iframes : integer : optional 返回值中是否包含子 frame 的信息，1 为包含，0 表示不包含，默认为 0

script : integer : optional 是否在返回中包含执行的 javascript final 语句的结果（请参阅: 在页面上下文中执行用户自定义的 JavaScript 代码），可选择的值有 1（包含）0（不包含），默认是 0

history : integer : optional 返回值中是否包含主页面的历史请求/响应数据，可选择的值有 1（包含）0（不包含），默认是 0

使用该参数来获取 HTTP 响应码和对应的头信息，它只会返回最主要的请求/响应信息（也就是说页面加载的资源信息和对应请求的 AJAX 信息是不会返回的）要获取请求和响应的更详细信息请使用 har 参数

har : integer : optional 是否在返回中包含 HAR 信息，可选择的值有 1（包含）0（不包含），默认是 0，如果这个选项被打开，那么它将会在 har 键中返回与 render.har 一样的数据

默认情况下响应体未包含在返回中，如果要返回响应体，可以使用参数 response_body

response_body : int : optional 可选择的值有 1（包含）0（不包含），如果值为 1，那么将会在返回的 HAR 信息中包含响应体的内容。在参数 har 和 history 为 0 的情况下该参数无效

示例

默认情况下，返回当前页面的 url，请求 url，页面标题，主 frame 的尺寸[16]

```
{
    "url": "http://crawlera.com/",
    "geometry": [0, 0, 640, 480],
    "requestedUrl": "http://crawlera.com/",
    "title": "Crawlera"
}
```

设置参数 html=1 ，以便让 HTML 能够加入到返回值中

[16] 原文这里是 geometry 在这里根据给出的例子我感觉还是用尺寸更为合适一些

```
{
    "url": "http://crawlera.com/",
    "geometry": [0, 0, 640, 480],
    "requestedUrl": "http://crawlera.com/",
    "html": "<!DOCTYPE html><!--[if IE 8]>....",
    "title": "Crawlera"
}
```

设置参数 png=1 以便使渲染后的截图数据以 base64 的编码方式加入到返回值中

```
{
    "url": "http://crawlera.com/",
    "geometry": [0, 0, 640, 480],
    "requestedUrl": "http://crawlera.com/",
    "png": "iVBORw0KGgoAAAAN...",
    "title": "Crawlera"
}
```

同时设置 html=1 和 png=1，能同时获取到截图和 HTML 代码。这样就保证了截图与 HTML 相匹配

通过添加 iframes=1，能够在返回中得到对应的 frame 的信息

```
{
    "geometry": [0, 0, 640, 480],
    "frameName": "",
    "title": "Scrapinghub | Autoscraping",
    "url": "http://scrapinghub.com/autoscraping.html",
    "childFrames": [
        {
            "title": "Tutorial: Scrapinghub's autoscraping tool - YouTube",
            "url": "",
            "geometry": [235, 502, 497, 310],
            "frameName": "<!--framePath //<!--frame0-->-->",
            "requestedUrl": "http://www.youtube.com/embed/lSJvVqDLOOs?version=3&rel=1&fs=1&
↪showsearch=0&showinfo=1&iv_load_policy=1&wmode=transparent",
            "childFrames": []
      }
  ],
  "requestedUrl": "http://scrapinghub.com/autoscraping.html"
}
```

请注意，iframe 可以嵌套

同时设置 iframe=1 和 html=1, 以获取所有 iframe 和 HTML（包括 iframe 的 HTML 代码）

```
{
    "geometry": [0, 0, 640, 480],
    "frameName": "",
    "html": "<!DOCTYPE html...",
    "title": "Scrapinghub | Autoscraping",
    "url": "http://scrapinghub.com/autoscraping.html",
    "childFrames": [
        {
            "title": "Tutorial: Scrapinghub's autoscraping tool - YouTube",
            "url": "",
            "html": "<!DOCTYPE html>...",
            "geometry": [235, 502, 497, 310],
            "frameName": "<!--framePath //<!--frame0-->-->",
            "requestedUrl": "http://www.youtube.com/embed/lSJvVqDLOOs?version=3&rel=1&fs=1&
↪showsearch=0&showinfo=1&iv_load_policy=1&wmode=transparent",
            "childFrames": []
        }
    ],
    "requestedUrl": "http://scrapinghub.com/autoscraping.html"
}
```

与’html = 1’不同,’png = 1’不会影响 childFrame 中的数据。

当需要执行 JavaScript 代码的时候（请参阅：在页面上下文中执行用户自定义的 JavaScript 代码），设置‘script=1’以便在结果中返回代码执行的结果

```
{
    "url": "http://crawlera.com/",
    "geometry": [0, 0, 640, 480],
    "requestedUrl": "http://crawlera.com/",
    "title": "Crawlera",
    "script": "result of script..."
}
```

可以在 JavaScript 代码中使用函数 console.log() 来记录相关信息，设置参数 console=1 以便在返回结果中包含控制台输出

```
{
    "url": "http://crawlera.com/",
    "geometry": [0, 0, 640, 480],
    "requestedUrl": "http://crawlera.com/",
    "title": "Crawlera",
    "script": "result of script...",
    "console": ["first log message", "second log message", ...]
}
```

curl 实例

```
# 返回完整的信息
curl 'http://localhost:8050/render.json?url=http://domain.com/page-with-iframes.html&png=1&html=1&
↪iframes=1'

# 页面自身的 HTML 代码，元数据信息以及所有的 iframe 信息
curl 'http://localhost:8050/render.json?url=http://domain.com/page-with-iframes.html&html=1&
↪iframes=1'

# 只返回元数据信息 (例如 页面/iframes 标题和 url)
curl 'http://localhost:8050/render.json?url=http://domain.com/page-with-iframes.html&iframes=1'

# 渲染页面并将页面裁剪为 320x240 的同时，不返回 iframe 的相关信息
curl 'http://localhost:8050/render.json?url=http://domain.com/page-with-iframes.html&html=1&png=1&
↪width=320&height=240'

# Render page and execute simple Javascript function, display the js output
curl -X POST -H 'content-type: application/javascript' \
-d 'function getAd(x){ return x; } getAd("abc");' \
'http://localhost:8050/render.json?url=http://domain.com&script=1'

# 渲染页面并执行简单的 JavaScript 代码，显示 js 的执行结果和在控制台的输出
curl -X POST -H 'content-type: application/javascript' \
    -d 'function getAd(x){ return x; }; console.log("some log"); console.log("another log"); getAd(
↪"abc");' \
    'http://localhost:8050/render.json?url=http://domain.com&script=1&console=1'
```

1.2.6 execute

执行自定义的渲染脚本并返回对应的结果

render.html, render.png, render.jpeg, render.har 和 render.json 已经涵盖了许多常见的情形，但是在某些时候这些仍然不够，这个端口允许用户编写自定义的脚本

参数:

lua_source : string : required 需要浏览器执行的脚本代码，请查看 Splash 脚本教程以获取更多信息

timeout : float : optional 与 render.html 中的 timeout 参数含义相同

allowed_domains : string : optional 与 render.html 中的 allowed_domains 参数含义相同

proxy : string : optional 与 render.html 中的 proxy 参数含义相同

filters : string : optional 与 render.html 中的 filters 参数含义相同

save_args : json 对象或者是以分号分隔的字符串: optional 与 render.html 中的 save_args 参数相同，请注意你不仅能保存 Splash 中的默认参数，也可以保存其他任何参数

load_args : JSON object or a string : optional 与 render.html 中的 load_args 参数相同，请注意你不仅能加载 Splash 中的默认参数，也可以加载其他任何参数

您可以传入任何类型的参数，所有在端点 execute 中传入的参数在脚本中都可以通过 splash.args 这个 table 对象来访问

1.2.7 run

这个端点与 execute 具有相同的功能，但是它会自动将 `lua_source` 包装在 `function main(splash, args) ... end` 结构中比如您在 execute 端点中传入脚本:

```
function main(splash, args)
    assert(splash:go(args.url))
    assert(splash:wait(1.0))
    return splash:html()
end
```

在使用 run 端点时只需要传入

```
assert(splash:go(args.url))
assert(splash:wait(1.0))
return splash:html()
```

1.2.8 在页面上下文中执行用户自定义的 JavaScript 代码

注解: 您也可以参考: 在 Splash 中执行 JavaScript 脚本

Splsh 支持在页面上下文中执行 JavaScript 代码，这些 JavaScript 代码在页面加载完成之后执行（包括由'wait'参数定义的等待时间）。但是它允许在页面渲染之前通过 Javascript 代码来修改渲染的结果。

可以通过 js_source 这个参数来执行 js 代码。参数中保存的是需要执行的 JavaScript 代码

请注意，浏览器和代理限制了可以使用 GET 发送的数据量，所以采用 POST 发送 `content-type: application/json` 类型的请求包将会是一个不错的选择

Curl example:

```
# 渲染页面并动态修改标题
curl -X POST -H 'content-type: application/json' \
    -d '{"js_source": "document.title=\"My Title\";", "url": "http://example.com"}' \
    'http://localhost:8050/render.html'
```

另一个发送 POST 请求的方式是设置请求包的 Content-Type 为 'application/javascript'，并在请求体中包含需要执行的 js 代码 curl:

```
# 渲染页面并动态修改标题
curl -X POST -H 'content-type: application/javascript' \
    -d 'document.title="My Title";' \
    'http://localhost:8050/render.html?url=http://domain.com'
```

可以通过使用 render.json 这个端点，并设置参数 script = 1 来获取 js 函数在页面上下文执行的结果

JavaScript 配置

Splash 允许使用 "JavaScript 配置" 的方式来预加载 JavaScript 文件，配置文件中的 JavaScript 代码将会在页面加载之后执行，但是会在请求中定义的 js 代码被执行之前

预加载的文件可以被包含在用户发送的 POST 请求中

为了开启 splash 对 JavaScript 文件的支持，在启动 splash 服务的时候可以使用参数 --js-profiles-path=<path to a folder with js profiles>

```
python3 -m splash.server --js-profiles-path=/etc/splash/js-profiles
```

注解: 请参阅 splash 版本

然后根据上面参数中给定的名称创建文件夹，在文件夹中创建需要加载的 js 文件（请注意，文件编码格式必须为 utf-8）这些文件都会在适当的时候被加载比如这样的一个目录结构

```
/etc/splash/js-profiles/
    mywebsite/
              lib1.js
```

为了应用这些 JavaScript 的配置，请在请求中添加参数 js=mywebsite

```
curl -X POST -H 'content-type: application/javascript' \
    -d 'myfunc("Hello");' \
    'http://localhost:8050/render.html?js=mywebsite&url=http://domain.com'
```

请注意，这个例子中假设 myfunc 是在 lib1.js 中定义的一个 JavaScript 函数

Javascript 安全

如果 splash 是通过 --js-cross-domain-access 的方式启动

```
$ docker run -it -p 8050:8050 scrapinghub/splash --js-cross-domain-access
```

此时将允许 JavaScript 代码访问非原始安全页面中的 iframe 中的内容（一般在浏览器中是不允许这个做）[17]。这个特性在爬取的时候非常有用，比如提取 iframe 中的 HTML 代码，它的一个使用的例子如下:

```
curl -X POST -H 'content-type: application/javascript' \
    -d 'function getContents(){ var f = document.getElementById("external"); return f.
↪contentDocument.getElementsByTagName("body")[0].innerHTML; }; getContents();' \
    'http://localhost:8050/render.html?url=http://domain.com'
```

这段 JavaScript 代码会查找一个 id 为”external”的 iframe，然后加载它的 HTML 代码

请注意：允许跨源调用 JavaScript 代码可能会造成一些安全问题，因为启用这些特性可能会泄漏一些敏感信息（例如 cookie），当禁用跨域安全时某些网站不会被加载，因此这个特性默认是关闭的

1.2.9 请求过滤

splash 允许通过 Adblock Plus 规则来对请求包进行过滤。您可以使用 EasyList 规则来过滤广告和跟踪代码（从而提高页面的渲染速度）。或者您也可以自己书写规则来过滤一些请求（例如书写规则来避免渲染 iframe，MP3，自定义字体等等）

要开启对请求的过滤，需要在启动 splash 的时候加上参数 `--filters-path`

```
python3 -m splash.server --filters-path=/etc/splash/filters
```

注解: 可以参阅 splash 版本

`filters-path` 所指向的目录中必须包含以 Adblock Plus 格式编写的规则的 `.txt` 文件您可以从 EasyList 的网站下载文件 `easylist.txt` 文件放到对应目录中，或者创建一个 `.txt` 文件编写自己的规则例如，让我们创建一个过滤器，以阻止加载的 ttf 和 woff 格式的自定义字体（在 Mac OS 中，可能会由于 qt 的 bug 导致 splash 产生一个段错误）

```
! put this to a /etc/splash/filters/nofonts.txt file
! comments start with an exclamation mark

.ttf|
.woff|
```

要使用这个规则您可以在请求包中添加参数 `filters=nofonts`

[17] 这里的原文是 then javascript code is allowed to access the content of iframes loaded from a security origin different to the original page (browsers usually disallow that)，翻译出来总感觉很别扭，我觉得这里的意思应该是跨站访问某些 iframe 并对它进行 js 渲染

```
curl 'http://localhost:8050/render.png?url=http://domain.com/page-with-fonts.html&filters=nofonts'
```

您可以添加多个规则并用逗号隔开它们

```
curl 'http://localhost:8050/render.png?url=http://domain.com/page-with-fonts.html&filters=nofonts,
↪easylist'
```

如果对应目录中存在一个 `default.txt` 那么文件里面的规则将会在默认情况下执行，即使您没有使用参数 `filters` 如果您不想使用默认的规则，您可以设置 `filters=none`

只有与情求相关的资源才会被过滤掉，加载主页的请求不会被过滤[18] 如果您确实想要这么做，请考虑在将 URL 发送到 Splash 之前使用 Adblock Plus 过滤器对 URL 进行检查（对 python 来说可以使用库 adblock-parser ）

您可以点击下面的链接来学习 Adblock Plus 过滤的语法

- https://adblockplus.org/en/filter-cheatsheet
- https://adblockplus.org/en/filters

splash 不能支持所有的 Adblock Plus 过滤规则，它有一些对应的限制

- 元素隐藏规则不受支持；过滤器可以过滤掉某些网络请求，但是并不能隐藏已加载页面的内容
- 只支持 `domain` 选项

splash 不支持的规则会被默默的丢弃

注解: 如果您想停止下载图片，请选择‘images’参数, 它不需要使用基于 url 的过滤器来进行过滤，它可以过滤掉那些使用基于 url 的过滤器很难过滤掉的图片

警告: 如果您的过滤器中含有大量的规则，您就得需要安装 pyre2 这个库。(这主要是针对从 EasyList 中下载下来的文件)

在 splash 中传统的 re 库会比 re2 慢上 1000x+ 的时间，当有大量的规则而未使用 re2 时下载文件会比过滤文件更快，但是使用 re2 会使规则的匹配更加迅速

您需要确认您未通过 PyPI 来下载 re2 的 0.2.20（这个版本已经被放弃了）；您应该使用最新版本

1.2.10 代理配置

splash 支持代理配置，它允许通过 `proxy` 参数来设置每个请求的代理处理规则

[18] 这里的意思是会过滤后续异步加载的请求而利用 url 针对主页面的请求不能被过滤，这样本省嵌入在主页中，与主页一起加载的内容不会被过滤掉

要支持代理配置，可以在启动 splash 的时候使用参数 `--proxy-profiles-path=<path to a folder with proxy profiles>` :

```
python3 -m splash.server --proxy-profiles-path=/etc/splash/proxy-profiles
```

注解: 如果您通过 docker 启动，请参数文件共享

然后在指定文件夹中创建一个以代理配置的规则编写的 INI 文件，例如在文件 `/etc/splash/proxy-profiles/mywebsite.ini` 中写下这些内容

```
[proxy]

; required
host=proxy.crawlera.com
port=8010

; optional, default is no auth
username=username
password=password

; optional, default is HTTP. Allowed values are HTTP and SOCKS5
type=HTTP

[rules]
; optional, default ".*"
whitelist=
    .*mywebsite\.com.*

; optional, default is no blacklist
blacklist=
  .*\.js.*
  .*\.css.*
  .*\.png
```

whitelist 和 blacklist 是以换行符分隔的正则表达式。如果 url 命中了白名单中的某项并且未命中黑名单中的任何一项，此时就使用在 `[proxy]` 节中定义的代理，否则就不使用代理

要使用对应的规则，可以在请求中添加 `proxy=mywebsite` 参数

```
curl 'http://localhost:8050/render.html?url=http://mywebsite.com/page-with-javascript.html&
↪proxy=mywebsite'
```

如果存在一个 `default.ini` 文件，那么会默认使用这个，即使你没有指定 `proxy` 参数，如果您有 `default.ini` 但是不想使用它，可以将 `proxy` 参数的值设置为 `none`

1.2.11 其他端点

_gc

可以向 /_gc 端点发送一个 POST 请求来回收一些内存

```
curl -X POST http://localhost:8050/_gc
```

它主要运行 python 的垃圾回收器，并清理 webkit 的缓存

_debug

可以向端点 /_debug 发送一个 GET 请求来获取 splash 历程的调试信息（RSS 的最大使用量、使用的文件描述符的数量、存活的请求、请求队列的长度、存活对象的个数）

```
curl http://localhost:8050/_debug
```

_ping

向 _ping 端点发送一个 GET 请求可以 ping splash 的历程

```
curl http://localhost:8050/_ping
```

如果 splash 历程存活，那么会返回 “ok” 状态和 RSS 的最大使用数

1.3 Splash 脚本教程

1.3.1 简介

splash 能够执行用户使用 Lua 语言编写的自定义渲染脚本，这就使我们能够像 PhantomJs 那样，将其作为一个浏览器自动化工具来使用

我们可以向 execute(或者 run) 端点发送请求，并设置上 lua_script 参数，以便执行脚本并获取返回值。在这个教程中主要使用 execute 端点

注解: 即使您之前没有 lua 的基础，您也可以很简单的看懂教程中的脚本示例。虽然它很简单，但也很值得学习，您可以使用 lua 语言编写 Redis , Nginx , Apache , 魔兽世界 的脚本可以使用 Corona 来创建手机应用或者使用当前最先进的深度学习框架 Torch7 。它很容易入门，并且在网上有许多很棒学习资源，像教程有 15 分钟学习 lua ，或者书籍 lua 编程语言

让我们从一个简单的例子开始:

```
function main(splash, args)
    splash:go("http://example.com")
    splash:wait(0.5)
    local title = splash:evaljs("document.title")
    return {title=title}
end
```

如果我们在将这个脚本填写到 `lua_script` 参数中并往 execute 端点上发送请求，那么 splash 会访问 example.com 这个站点，并等待它加载，会等待半秒，然后获取页面标题 (通过在页面上下文中执行 JavaScript 代码), 最后以 json 格式返回结果。

注解: Splash UI 提供了一种简便的方法来测试脚本，它里面有一个 lua 脚本的编辑框和一个将脚本提交到 execute 端点的按钮。您可以访问 http://127.0.0.1:8050/ (或者其他 splash 监听的主机和端口)

为了执行在您的编程环境中执行脚本，您需要弄清楚如何发送 HTTP 请求，您可以在问答模块中参考如何向 Splash API 发送 HTTP 请求，它包含了一些常见的方法和步骤 (比如，使用 Python + requests 库)

1.3.2 入口点——main 函数

脚本必须提供一个 main 函数供 splash 调用，执行结果会以 http 响应包的方式返回，脚本中可以包含其他有用的函数，但是 main 函数是必须的。在第一个例子中，main 函数返回一个 lua 的 table 结构（一个类似于 JavaScript 的 object 或者 Python 字典的一个关联数组)。这类结果将会以 json 的格式返回

下面的代码将会在 http 的响应中返回 `{"hello":"world!"}` 字符串

```
function main(splash)
    return {hello="world!"}
end
```

脚本也可以返回一个字符串

```
function main(splash)
    return 'hello'
end
```

字符串的返回值会原样的在响应体重返回 (它不会被编码成 json 格式)，请看下面的例子

```
$ curl 'http://127.0.0.1:8050/execute?lua_source=function+main%28splash%29%0D%0A++return+%27hello
↪%27%0D%0Aend'
hello
```

main 函数接收一个对象，该对象允许我们向操作浏览器选项卡那样操作 splash，splash 所有功能都被封装到此对象中，为了方便这个参数的名称约定俗成的被称为 “splash”，但是您不必遵守这条约定:

```
function main(please)
    please:go("http://example.com")
    please:wait(0.5)
    return "ok"
end
```

1.3.3 我们的回调在哪?

下面是我们第一个例子的部分代码

```
splash:go("http://example.com")
splash:wait(0.5)
local title = splash:evaljs("document.title")
```

这段代码就像传统的面相过程的代码，没有回调也没有花哨的控制流结构。但这并不意味这 splash 是以同步的方式运行。在引擎中它仍然是异步的。当代码执行到 `splash.wait(0.5)` 时，splash 从当前任务中跳出去执行其他任务，在 0.5s 之后再切换回来。

我们可以向一般的脚本语言一样使用条件、循环语句和函数，从而使编写的代码更加直观

下面来看一个 phantomjs 中的脚本的例子

```
// Render Multiple URLs to file

"use strict";
var RenderUrlsToFile, arrayOfUrls, system;

system = require("system");

/*
Render given urls
@param array of URLs to render
@param callbackPerUrl Function called after finishing each URL, including the last URL
@param callbackFinal Function called after finishing everything
*/
RenderUrlsToFile = function(urls, callbackPerUrl, callbackFinal) {
    var getFilename, next, page, retrieve, urlIndex, webpage;
    urlIndex = 0;
    webpage = require("webpage");
    page = null;
    getFilename = function() {
        return "rendermulti-" + urlIndex + ".png";
    };
```

(continues on next page)

(续上页)

```
    next = function(status, url, file) {
        page.close();
        callbackPerUrl(status, url, file);
        return retrieve();
    };

    retrieve = function() {
        var url;
        if (urls.length > 0) {
            url = urls.shift();
            urlIndex++;
            page = webpage.create();
            page.viewportSize = {
                width: 800,
                height: 600
            };
            page.settings.userAgent = "Phantom.js bot";
            return page.open("http://" + url, function(status) {
                var file;
                file = getFilename();
                if (status === "success") {
                    return window.setTimeout((function() {
                        page.render(file);
                        return next(status, url, file);
                    }), 200);
                } else {
                    return next(status, url, file);
                }
            });
        } else {
            return callbackFinal();
        }
    };
    return retrieve();
};

arrayOfUrls = null;

if (system.args.length > 1) {
    arrayOfUrls = Array.prototype.slice.call(system.args, 1);
} else {
    console.log("Usage: phantomjs render_multi_url.js [domain.name1, domain.name2, ...]");
    arrayOfUrls = ["www.google.com", "www.bbc.co.uk", "phantomjs.org"];
```

(continues on next page)

(续上页)

```
}

RenderUrlsToFile(arrayOfUrls, (function(status, url, file) {
    if (status !== "success") {
        return console.log("Unable to render '" + url + "'");
    } else {
        return console.log("Rendered '" + url + "' at '" + file + "'");
    }
}), function() {
    return phantom.exit();
});
```

平心而论这段代码写的很晦涩 RenderUrlsToFile ``函数通过创建一个回调链来实现循环，``page.open 函数并没有返回任何值（如果返回某些值的话实施起来会更加复杂）而是将返回值存入到磁盘中

下面是一个使用 splash 脚本更为简单的例子

```
function main(splash, args)
    splash.set_viewport_size(800, 600)
    splash.set_user_agent('Splash bot')
    local example_urls = {"www.google.com", "www.bbc.co.uk", "scrapinghub.com"}
    local urls = args.urls or example_urls
    local results = {}
    for _, url in ipairs(urls) do
        local ok, reason = splash:go("http://" .. url)
        if ok then
            splash:wait(0.2)
            results[url] = splash:png()
        end
    end
    return results
end
```

二者的功能有点不一样，这段代码没有保存页面的截图，而是将 png 图片的值使用 HTTP API 的功能返回到客户端

意见或建议

- 使用 page.open 函数并获取返回状态的这种方式有一个阻塞，作为替代可以使用 splash:go 并判断返回的标志是否为 “ok”
- 在 lua 中使用 loop 循环，而不是通过创建一个回调链来实现循环
- 拥有一些 lua 的知识有助于编写 lua 脚本，比如您可能对 ipairs 和 string 的连接符 .. 不太熟悉
- 错误处理是不同的，当发生 HTTP 的 4xx 或者 5xx 错误时，PhantomJS 虽然会得到一个页面的截图但是不会在 page.open 的回调中返回错误码，因为它的状态不为 “fail”，而在 splash 中会检测出这些

错误

- 为了不在控制台中打印返回或者将返回结果保存在文件中，我们可以使用与 json 相关的 Splash HTTP API
- PhantomJS 允许创建多个页面对象，以便在面板的 `page.open` 中提交多个请求，splash 只在 `main` 函数的 splash 参数中为脚本提供单个浏览器选项卡 (但是您可以自由的将多个包含 lua 脚本的请求并发的提交给 splash)

现在有许多很棒的针对 PhantomJS 的封装，像 CasperJS 、NightmareJS 它们提供了自定义的流程控制的微型语言，以便 PhantomJS 的脚本编写出来看起来像同步的方式，但是也多多少少存在一定的问题 (像循环，将代码移至帮助函数[3] ，错误处理) splash 则采用标准的 LUA 语言

注解: PhantomJS 和它对应的封装都很棒，很值得敬佩，不要因为上面的内容而抨击它们，它们比 splash 更加成熟，功能也更加完善 splash 尝试从另一个角度来看待问题，但是每一个独立的 splash 功能都有一个独特的 PhantomJS 功能与之对应

您想了解更多关于 Splash Lua API 的功能请参考 Splash Lua API 概览

1.3.4 在没有回调的情况下编写代码

注解: 您一定对 splash 引擎中使用的 lua 协程很好奇

其实在内部 main 函数是被 splash 作为一个协程在执行，像 `splash:foo()` 这类函数是用 `coroutine.yield` 来实现的，关于 lua 的协程，请参阅 http://www.lua.org/pil/9.html

---

在 splash 的脚本中并没有区分哪些是阻塞的哪些是同步的。这是对协程和小型组件的一些常见的批评 这篇文章 <> 对这个问题进行了很好的描述，您可以参考一下

但是这些问题并没有真正影响到 splash 脚本的执行，splash 的脚本一般是一段很小的代码，代码的共享状态缩减到最小，API 被设计成了同一时间内只执行单行代码。这些都意味着代码的执行流程是串行化的

如果您想要安全，可以把所有 splash 函数看做异步执行的。首先要考虑的是当您执行 `splash:foo()` 函数后，之前渲染的 web 页面就被更改了。这通常是调用这些方法的要点，`splash:wait(time)` 或者 `splash:go(url)` 这些函数只在这点上有意义，因为执行它们之后，web 页面就被更改了。[4] 您需要谨记这点

这里面有许多异步函数，像: splash:go , splash:wait , splash:wait_for_resume 。虽然大多数的 splash 函数都不是异步方式工作的，但是您将它们想象成异步的将使您的代码在未来它们被变成异步方式时也能正常工作

[3] 这块的原文是: moving code to helper functions? 暂时找不到合理的翻译方式

[4] 这里的原文是: splash:wait(time) or splash:go(url) only make sense because webpage changes after calling them

1.3.5 调用 splash 函数

与大多数语言不同，lua 中使用冒号 : 来调用类对象的中的方法，为了调用 splash 对象中的 `foo` 方法，需要写成 `splash:foo()`。更多细节请参考 http://www.lua.org/pil/16.html

在 splash 脚本中有两种方式来调用 lua 中的函数：按顺序传参和使用参数名传参；当使用按顺序传参的方式来调用函数时使用小括号作为形参列表 `splash:foo(val1, val2)`。当使用参数名传参的时候使用大括号来作为形参列表 `splash:foo{name1=val1, name2=val2}`

```
-- Examples of positional arguments:
splash:go("http://example.com")
splash:wait(0.5, false)
local title = splash:evaljs("document.title")

-- The same using keyword arguments:
splash:go{url="http://example.com"}
splash:wait{time=0.5, cancel_on_redirect=false}
local title = splash:evaljs{source="document.title"}

-- Mixed arguments example:
splash:wait{0.5, cancel_on_redirect=false}
```

为了方便，所有的 splash API 都被设计成接受这调用两种方式。但是针对在 lua 中大多数函数都是 没有参数参数名称的 Lua 函数 这样的一种情况 (包括大部分从标准库中导出的函数), 只能选择使用按参数顺序传参

1.3.6 错误处理

在 lua 中有两种报告错的方式，抛出一个异常、返回一个错误码。请参阅 http://www.lua.org/pil/8.3.html. 。

而在 splash 中有如下惯例: 1. 开发者自己的错误 (例如不正确的函数参数), 抛出异常 #. 开发者向外部调用者提供的错误 (无法访问的站点), 通过返回标志值的方式：比如函数可以返回 `ok, reason` 结构，而调用者可以选择忽略或者处理

如果 main 函数的结果中有有一个未处理的异常，splash 会返回 HTTP 400 并带上出错的信息

我们可以使用 lua 的 `error` 函数手工的抛出一个异常

```
error("A message to be returned in a HTTP 400 response")
```

您可以使用 lua 中的 `pcall` 函数来处理异常 (防止 splash 返回 HTTP 400 的错误)。请参阅 http://www.lua.org/pil/8.4.html

您可以使用 `assert` 将错误标志转化为异常, 比如您想使一个站点一直运行，但是又不想手工的处理这个错误，当您指定的错误发生时您可以使用 `assert` 来停止当前进程并使 splash 返回 HTTP 400 的错误

```
local ok, msg = splash:go("http://example.com")
if not ok then
    -- handle error somehow, e.g.
    error(msg)
end

-- a shortcut for the code above: use assert
assert(splash:go("http://example.com"))
```

1.3.7 沙盒

默认情况下，spalsh 脚本在受限制的环境下运行，在这种情况下并非所有的 Lua 模块和函数都是有效的。例如，`require` 函数被限制了，同时也针对一些资源的数量进行了限制 (虽然这个限制很松)。

您可以通过参数 `--disable-lua-sandbox` 来启动 splash 的沙盒

```
$ docker run -it -p 8050:8050 scrapinghub/splash --disable-lua-sandbox
```

1.3.8 超时

默认情况下，在超时后 splash 会停止脚本的执行 (默认超时值是 30s)。这对于比较长的脚本是一个常见的问题。更多详情请参考: *求助：我有一个 504 超时错误 <> 和 splash lua 脚本需要做很多工作 <>*

1.4 Splash lua API 概览

Splash 提供了许多方法、函数和属性。这些内容分别被写进了文档：Splash 脚本参考 , 可用的 lua 库 , 元素对象 , 请求对象 , 响应对象 和 使用二进制数据 。下面将对这些内容做一些简单的描述

1.4.1 脚本作为 Splash HTTP API 端点

每一个 splash 的 lua 脚本都可以看做是一个 HTTP API 端点，具有输入参数和结构化的返回值。您可以使用 lua 脚本来模拟一个 render.png 端点，包括它所使用的参数。

- splash.args 可以从脚本中获取数据[2]
- splash:set_result_status_code 可以在返回值中修改 HTTP 的状态码
- splash:set_result_content_type 运许更改返回给客户端的 Content-Type
- splash:set_result_header 可以在返回值中增加自定义的 HTTP 头
- Working with Binary Data 描述了如何在 splash 中使用非文本数据、比如如何给客户端返回二进制数

[2] 这里的脚本指的是 Python 脚本

- treat 运许将在返回结果时将自定义数据序列化为 json 格式

1.4.2 导航

- splash:go 在浏览器中加载指定的 url
- splash:set_content 在浏览器中加载特定的内容（通常是 HTML）
- splash:lock_navigation 与 splash:unlock_navigation 锁定与解锁导航
- splash:set_user_agent 修改请求中使用的 User-Agent
- splash:set_custom_headers 允许设置 splash 默认使用的 HTTP 请求头
- splash:on_request 允许过滤或者替换对相应资源的请求，它允许在请求前设置 HTTP 或者 SOCKS5 代理服务
- splash:on_response_headers 允许通过请求头来过滤请求，比如通过 Content-Type
- splash:init_cookies ， splash:add_cookie ， splash:get_cookies ， splash:clear_cookies 与 splash:delete_cookies 允许对 cookie 进行管理

1.4.3 延迟

- splash:wait 允许等待特定的时间
- splash:call_later 后续执行一项任务
- splash:wait_for_resume 等待某些 js 事件发生
- splash:with_timeout 允许设置代码执行的超时时间

1.4.4 在页面中提取相关的信息

- splash:html 返回页面通过浏览器渲染后的 HTML 内容
- splash:url 返回当前浏览器中加载的 url
- splash:evaljs 和 splash:jsfunc 允许在页面中通过 js 来提取数据
- splash:select 和 splash:select_all 允许在页面中使用 CSS 选择器。它们会返回对应元素的对象，这些对象在许多方法和后续的处理上都很有用 (请参阅 元素对象)
- element:text 返回 DOM 对象的文本内容
- element:bounds 返回元素的边界框
- element:styles 返回自定义的元素样式
- element:form_values 返回表单元素的值

- DOM 中的 ‘ HTMLElement <https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement>‘_ 对象的绝大多数属性和方法都被支持，您可以参阅 DOM 属性 和 DOM 方法

1.4.5 截图

- splash:png, splash:jpeg 得到一个 PNG 和 JPEG 的截图
- splash:set_viewport_full 通过修改视口的大小（通常在使用 splash:png, *splash:jpeg <./scripting-ref.html#splash-jpeg>* 之前调用）可以获得页面完整的截图
- splash:set_viewport_size 修改视口大小
- element:png 和 element:jpeg 截取单个 DOM 元素的截图

1.4.6 与页面交互

- splash:runjs , splash:evaljs 和 splash:jsfunc 允许在页面上下文中执行任意的 js 脚本
- splash:autoload 允许在页面开始渲染的时候预加载一些 JavaScript 库或者执行 JavaScript 代码
- splash:mouse_click , splash:mouse_hover , splash:mouse_press , splash:mouse_release 允许向页面指定的坐标发送鼠标消息
- splash:send_keys 和 splash:send_text 允许向页面发送键盘事件
- element:send_keys and element:send_text 向指定的 DOM 元素发送键盘事件
- 您可以通过使用 element:form_values 函数来设置表单的初始值, 使用 element:fill 来更新表单值, 使用 ‘ element:submit <./scripting-element-object.html#splash-element-submit>‘_ 来提交表单
- splash.scroll_position 允许您滚动页面
- DOM 中的 ‘ HTMLElement <https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement>‘_ 对象的绝大多数属性和方法都被支持，您可以参阅 DOM 属性 和 DOM 方法

1.4.7 构造 HTTP 请求

- splash:http_get 发送一个 HTTP 的 GET 请求，并获取一个未经过渲染的原始的响应
- splash:http_post 发送一个 HTTP 的 POST 请求，并获取一个未经过渲染的原始的响应

1.4.8 检查网络流量

- splash:har 在 HAR 结构体中返回所有的 HTTP 请求和响应
- splash:history 返回有关重定向和加载到主浏览器窗口的页面的信息;

- splash:on_request 允许捕捉浏览器页面或者脚本中的错误
- splash:on_response_headers 允许在获取响应头后，进行检查（或者丢弃）
- splash:on_response 允许检查接受到的原始响应信息（包括相应的资源新）
- splash.response_body_enabled 允许接受 splash:har 和 *splash:on_response <./scripting-ref.html#splash-on-response>* 中全部的响应信息
- 您可以参阅 Response Object 和 Request Object 来获取关于请求和响应的更详细的信息

1.4.9 浏览选项

- splash.js_enabled 允许关闭 JavaScript 的支持
- splash.private_mode_enabled 允许关闭专有模式 (它在有些情况下是有用的，比如 Webkit 在专有模式是没有本地存储)
- splash.images_enabled 允许禁止下载图片
- splash.plugins_enabled 允许禁止加载插件 (默认情况下 docker 中的镜像允许加载 Flash)
- splash.resource_timeout 允许在超时后减缓或者挂起相关请求
- splash.indexeddb_enabled 允许打开 IndexedDB
- splash.webgl_enabled 允许关闭 WebGL
- splash.html5_media_enabled 允许打开对 HTML5 的支持 (比如播放 `<video>` 标签)
- splash.media_source_enabled 允许关闭对多媒体扩展 API 的支持

1.5 Splash 脚本参考

注解: 虽然这个参考十分全面，但是它很难作为入门的内容。如果您是初学状态，或者看不懂下面的内容，请您首先查阅 Splash lua API 概览 这部分的内容

`splash` 作为 main 函数的第一个参数被传进来，通过这个对象脚本可以控制浏览器。您可以将它看做一个单一浏览器标签的 API[9]

1.5.1 属性

[9] 这是由于 Splash 一次只能加载一个页面，在加载新页面之前的老页面不会被保存

splash.args

`splash.args` 是一个传入参数的 table 类型，它包含了来自 GET 请求的原始 url 的合并值和数据类型为 `application/json` 的 POST 请求的请求体的值

例如您在使用 Splash HTTP API 的时候在脚本的参数中添加了一个 url 参数，那么 `splash.args.url` 就包含了您传入的这个 URL[10]

您也可以将 `splash.args` 作为 main 函数的第二个参数传入:

```
function main(splash, args)
    local url = args.url
    -- ...
end
```

它等效于:

```
function main(splash)
    local url = splash.args.url
    -- ...
end
```

使用 `args` 或者*splash.args* 是向 Splash 脚本传递参数的首选。另一个方式是通过字符串格式化将参数作为嵌入的变量来构造 lua 脚本的字符串。但是相比与使用 splash.args 来说，这种方法有两个问题

1. 数据必须以合理的方式组织，以便它们不会破坏 lua 脚本
2. 嵌入变量这种方式使得脚本无法高效的使用缓存 (您可以参考 HTTP API 部分的 save_args 和 load_args)

splash.js_enabled

允许或者禁止执行嵌入到页面中的 JavaScript 代码

原型: `splash.js_enabled = true/false`

默认情况下允许执行 JavaScript 代码

splash.private_mode_enabled

开启或者关闭浏览器的私有模式 (匿名模式)

原型: `splash.private_mode_enabled = true/false`

默认情况下匿名模式是开启的，除非您使用 `--disable-private-mode` 选项来启动 Splash，请注意如果您关闭了匿名模式，某些请求数据可能会在浏览器中持续存在 (但是它并不影响 cookie)

[10] 这个意思是说我们在使用 API 时传入的参数都可以通过这个参数获取到

您也可以参考 如果关闭匿名模式

splash.resource_timeout

第二次设定网络请求的超时值

原型: `splash.resource_timeout = number`

例如，下面的例子演示了当请求远端的资源超过 10s 时异常告警

```
function main(splash)
    splash.resource_timeout = 10.0
    assert(splash:go(splash.args.url))
    return splash:png()
end
```

当值为 0 或者 `nil` 时表示没有设置超时值

在设置请求的超时值事，在*splash:on_request* 函数中使用 `request:set_timeout` 设置的超时值要优先于*splash.resource_timeout*

splash.images_enabled

是否加载图片

原型: `splash.images_enabled = true/false`

默认情况下运行加载图片, 禁止加载图片能节省大量的网络带宽 (通常在 50% 左右) 并且能提高渲染的速度。请注意这个选项可能会影响页面中 JavaScript 代码的执行：禁止加载图片可能会影响 DOM 元素的坐标或者大小，而在脚本中可能会读取并使用它们

splash 在使用缓存的情况下，如果禁止了图片的加载，当缓存中存储了对应的图片，图片就会被加载。所以可能造成的问题是当您在允许图片加载的情况下加载了一个页面，然后禁止图片加载，接着再加载一个新页面，此时您可能会看到，第一个页面加载了图片，而第二个页面加载的部分图片（被加载的是与第一个页面相同的图片）。在同一个进程中 splash 的缓存是可以被不同的脚本共享的，所以您可能会在某些页面中看到部分图片即使您在一开始就禁止了图片的加载

示例

```
function main(splash, args)
    splash.images_enabled = false
    assert(splash:go(splash.args.url))
    return {png=splash:png()}
end
```

splash.plugins_enabled

允许或者禁止浏览器插件 (例如 Falsh)

原型: `splash.plugins_enabled = true/false`

默认情况下插件是被禁止的

splash.response_body_enabled

启用或者禁止响应内容追踪

原型: `splash.response_body_enabled = true/false`

从效率上考虑，默认情况下 Splash 不会在内存中保存每个请求的响应内容。这就意味着在函数*splash:on_response* 的回调函数中，我们无法获取到 response.body 属性，同时也无法从 HAR 中获取到响应的对应内容。可以通过在 lua 脚本中设置 `splash.response_body_enabled = true` 来使响应内容变得有效

请注意，不管*splash.response_body_enabled* 是否设置，在:ref:*splash:http_get* <*splash-http-get*> 和*splash:http_post* 中总是能获取到 response.body 的内容

您可以通过在函数*splash:on_request* 的回调中设置 request:enable_response_body 来启用每个请求的响应内容跟踪

splash.scroll_position

设置或者获取当前滚动的位置

原型: `splash.scroll_position = {x=..., y=...}`

这个属性允许我们设置或者获取当前主窗口的滚动的位置

将窗口滚动到内容以外是没有意义的，例如您设置 `splash.scroll_position` 为 `{x=-100, y=-100}` 效果与 `splash.scroll_position` 默认的 `{x=0, y=0}` 相同

在设置滚动位置的时候，您不用写全 (例如, `splash.scroll_position = {x=100, y=200}`) 您可以简写成 `splash.scroll_position = {100, 200}`。即使您使用的简写的方式，属性值也会被作为一个键为 `x` 和 `y` 的 table

当然，您也可以省略您不想改变的坐标值，例如 `splash.scroll_position = {y=200}` 是将 y 的值改为 200，而 x 的值保持不变

splash.indexeddb_enabled

允许或者禁止 IndexedDB

原型: `splash.indexeddb_enabled = true/false`

默认情况下 IndexedDB 是被禁止的。您可以使用 `splash.indexeddb_enabled = true` 来开启它

注解: 在当前默认情况下 IndexedDB 是被禁止的，这是因为它在 WebKit 中存在一些问题，可能在未来它会被默认打开

splash.webgl_enabled

启用或者禁用 WebGL

原型: `splash.webgl_enabled = true/false`

WebGL 默认是启用的，您可以通过 `splash.webgl_enabled = false` 来禁用

splash.html5_media_enabled

禁止或者启用 HTML5 多媒体, 包括 HTML5 中的 video 和 audio (例如 `<video>` 标签进行回放)

原型: `splash.html5_media_enabled = true/false`

默认情况下 HTML5 标签是被禁用的，您可以设置 `splash.html5_media_enabled = true` 来启用

注解: 默认情况下 HTML5 被禁止，因为它在某些环境下会使 WebKit 在访问某些网站时崩溃。在未来它可能会被设置为 `true` 。如果在您的程序中不需要使用 HTML5，请您明确的设置它为 `false`

您也可以参考 splash.html5_media_enabled 这个 HTTP API 参数的内容

splash.media_source_enabled

允许或者禁止 多媒体资源扩展 API

原型: `splash.media_source_enabled = true/false`

多媒体资源在默认情况下是打开的，您可以使用 `splash.media_source_enabled = false` 来关闭它

1.5.2 方法

splash:go

跳转到一个 URL, 它的效果类似于在浏览器的地址栏中输入一个 url，然后按回车键等待页面加载

原型:

```
ok, reason = splash:go{url, baseurl=nil, headers=nil, http_method="GET", body=nil, formdata=nil}
```

参数:

- url: 需要加载的页面的 url
- baseurl: 这个参数为可选参数。当给定了 `baseurl` 参数后，页面仍然从 `url` 参数中加载，但是它呈现为，页面中资源的相对路径是相对于 baseurl 来说的。，而且浏览器会认为 baseurl 在地址栏中。
- headers: 一个由 lua table 结构表示的 http 请求头，它被用来新增或者替换初始请求中的头信息
- http_method：可选参数，它使用一个字符来表示如果请求 url 所表示的页面，默认为 GET，splash 同样支持 POST
- body：可选参数，它是 POST 请求中的 body 部分的字符串
- formdata: 可选参数，类型为 lua 中的 table，当 POST 请求中的 `Content-Type` 为 `content-type: application/x-www-form-urlencoded` 时，它会进行相应的编码，并作为 POST 请求的 body 部分。

返回值: `ok`, `reason` 元组[12]。如果在加载页面的过程中发生错误那么 `ok` 为空。`reason` 将会保存错误的类型信息

异步: 为异步操作，除非导航被锁

将会报告的 5 种错误类型 (`ok` 会为 `nil` 的 5 种情况)

1. 发生网络错误, 主机不存在, 失去与远程服务端的连接等等。在这种情况下 `reason` 为 `"network<code>"` 。可以在 QT 的文档中找到对应的错误码，比如 ‘‘network3‘‘ 表示 NDS 错误 (无效的主机名称)
2. spalsh 返回带有 4xx 或者 5xx 状态码的 HTTP 响应信息。在这种情况下 `reason` 的值为 `http<code>` 。例如当发生 HTTP 404 Not Found 时，`reason` 的值为 `"http404"`
3. 导航被锁住 (请参阅*splash:lock_navigation*) 。此时 `reason` 的值为 `"navigation_locked"`
4. splash 不能加载主页面 (例如第一个请求被丢弃) `reason` 的值为 `render_error`
5. 如果 splash 不能确定是哪种错误，会简单的返回一个 `error`

请看下面的例子

```
local ok, reason = splash:go("http://example.com")
if not ok then
    if reason:sub(0,4) == 'http' then
        -- handle HTTP errors
    else
        -- handle other errors
    end
end
-- process the page

-- assert can be used as a shortcut for error handling
assert(splash:go("http://example.com"))
```

只有当主页[11] 请求失败时才会上报一个错误 (ok==nil)。如果针对其中的相关资源的请求失败 `splash:go` 不

[12] 这里的原文是 pair，由于我对 lua 不是很了解，这里翻译为元组可能更好理解

[11] 这里的主页是指参数中 url 对应的页面，而页面中的资源等等不包含在里面

会上报错误。为了确定上述错误是否发生或者处理这些错误 (像 image/js/css 等链接加载失败，ajax 请求失败), 您可以使用*splash.har* 和*splash:on_response*

`splash:go` 在返回结果之前会一直跟随 HTTP 的重定向去请求其中的链接，但是它不会跟踪像 `<meta http-equiv="refresh" ...>` 这样在 HTML 中定义的重定向或者在 JavaScript 代码中的跳转。此时您可以使用方法*splash:wait* 进行等待，以便使浏览器跳转到对应的页面上

`headers` 参数允许添加或者修改初始 HTTP 请求中的 header 值, 您可以使用*splash:set_custom_headers* 和*splash:on_request* 来为以后所有的请求设置 header 的值 (包括后续针对对应资源文件的请求)

下面是一个自定义设置 header 的例子:

```
local ok, reason = splash:go{"http://example.com", headers={
    ["Custom-Header"] = "Header Value",
}})
```

headers 中的 User-Agent 项比较特殊, 一旦使用他将被保留并用于进一步的请求。这是一个实现的细节，我们可能在未来的版本中对这个特性进行修改。在设置 User-Agent 时，推荐使用方法*splash:set_user_agent*

splash:wait

等待对应的时间 (单位为秒), 使程序等待 WebKit 对网页进行进一步的处理

原型: `ok, reason = splash:wait{time, cancel_on_redirect=false, cancel_on_error=true}`

参数:

- time: 等待的时间, 单位为 s
- cancel_on_redirect: 如果它为 true(默认为 false) 并且在加载主页面的时候发生了重定向，`splash:wait` 函数会提前返回，返回值为 `nil "redirect"` . 但是在 HTML 中通过 `<meta http-equiv="refresh" ...>` 定义的重定向或者在 JavaScript 中的重定向不受影响
- cancel_on_error: 如果它为 true(默认为 true) 在等待时如果发生相关错误导致页面无法加载 (像 WebKit 内部的错误或者重定向到了一个无法解析的主机上), 此时 `splash:wait` 会提前退出并返回 `nil, "<error string>"`

返回值: `ok, reason` 元组, 如果 `ok` 的值为 `nil` ，函数可能会提前退出，`reason` 可能会返回一个包含错误信息的字符串

是否为异步: 异步

使用示例

```
-- go to example.com, wait 0.5s, return rendered html, ignore all errors.
function main(splash)
    splash:go("http://example.com")
    splash:wait(0.5)
```

(continues on next page)

(续上页)

```
    return {html=splash:html()}
end
```

默认情况下当重定向发生的时候计时器会继续计时，cancel_on_redirect 选项可以在每次重定向发生的时候让计时器重新计时. 例如下面这个函数演示了如何利用 cancel_on_redirect 来实现每次重定向并加载对应页面后等待对应的时间

```
function wait_restarting_on_redirects(splash, time, max_redirects)
    local redirects_remaining = max_redirects
    while redirects_remaining > 0 do
        local ok, reason = self:wait{time=time, cancel_on_redirect=true}
        if reason ~= 'redirect' then
            return ok, reason
        end
        redirects_remaining = redirects_remaining - 1
    end
    return nil, "too_many_redirects"
end
```

splash:jsfunc

将 JavaScript 函数转化为 lua 可调用的函数

原型: `lua_func = splash:jsfunc(func)`

参数:

- func: 包含 js 函数代码的字符串

返回值: 返回一个函数, 该函数可以被 lua 执行并且可以在页面上下文中执行 JavaScript 代码

是否为异步: 否

例子

```
function main(splash, args)
  local get_div_count = splash:jsfunc([[
  function () {
    var body = document.body;
    var divs = body.getElementsByTagName('div');
    return divs.length;
  }
  ]])
  splash:go(args.url)
```

(continues on next page)

(续上页)

```
  return ("There are %s DIVs in %s"):format(
    get_div_count(), args.url)
end
```

请注意，如果您了解 lua 字符串中关于 [[ ]] 的语法知识将会对您理解这些代码有一定的帮助

JavaScript 代码也可以接受参数

```
local vec_len = splash:jsfunc([[
    function(x, y) {
       return Math.sqrt(x*x + y*y)
    }
]])
return {res=vec_len(5, 4)}
```

全局的 JavaScript 函数可以被直接包含进来

```
local pow = splash:jsfunc("Math.pow")
local twenty_five = pow(5, 2)  -- 5#2 is 25
local thousand = pow(10, 3)    -- 10#3 is 1000
```

lua 类型到 JavaScript 之间的转化规则

Lua	JavaScript
string	string
number	number
boolean	boolean
table	Object or Array, see below
nil	undefined
Element	DOM node

lua 中的 strings, numbers, booleans 和 tables 类型的数据可以直接作为参数传入，它们会被转化成 js 的 strings/numbers/booleans/objects 类型。element 也是被支持的。但是这类数据不能被放到 lua 的 table 中

就目前来说，不能传递其他的 lua 对象，比如不能往一个闭包的 JavaScript 函数中穿入一个闭包的 JavaScript 函数或者正常的 lua 函数作为参数

默认情况下 lua 中的 table 类型被转化为 JavaScript 中的 object 类型。如果您要将一个 table 类型转化为数组类型，可以使用函数 treat.as_array

JavaScript 类型到 lua 对象的转化

Lua	JavaScript
string	string
number	number
boolean	boolean
Object	table
Array	table, 将其转化为数组 (请参阅 treat.as_array)
undefined	nil
null	“” (一个空字符串)
Date	string: 一个使用 ISO8601 标砖表示的日期字符串, 例如. 1958-05-21T10:12:00.000Z
Node	Element 实例
NodeList	一个由 Element 对象组成的 table
function	nil
circular object	nil
host object	nil

函数会将返回结果中的 JavaScript 类型转化为 lua 类型，它只支持一些简单的 JavaScript 类型的转化，例如如果从闭包中返回一个函数或者 jQuery 选择器，这种操作不被支持

当需要返回一个节点 (html DOM 元素的一个引用) 或者节点的列表 (document.querySelectorAll 函数返回的结果) 时，只能返回节点或者节点列表这样单一的内容，它们不能被包含进数组或者其他的结构中

注解: 经验法则：如果参数或者返回值能被序列化为 json 格式的话，那样最好，当然您也可以在函数中返回节点或者节点的列表，但是它们不能被包含在其他结构中[13] 。

请注意目前您不能返回 jQuery $ 结果或者从 JavaScript 到 lua 的类似结构[14] 。要传递数据必须将您感兴趣的属性提取为普通的字符串/数字/对象/数组

```
-- 这个函数假设 jQuery 已经在页面中加载
local get_hrefs = splash:jsfunc([[
function(sel){
    return $(sel).map(function(){return this.href}).get();
}
]])
local hrefs = get_hrefs("a.story-title")
```

当然您也可以在代码中使用 Element 对象和*splash:select_all*

```
local elems = splash:select_all("a.story-title")
local hrefs = {}
```

(continues on next page)

[13] 我觉得这里官方应该是鼓励我们尽量返回能够被序列化的数据，而不要返回类似 DOM 元素的内容

[14] 这里的意思应该是不要使用 JavaScript 的复杂结构

(续上页)

```
for i, elem in ipairs(elems) do
    hrefs[i] = elem.node:getAttribute("href")
end
```

函数的参数和返回值都是按值传递, 比如说您在 JavaScript 函数中修改了某个参数的值，作为函数调用者的 lua 代码是不知道的，您在 js 代码中返回某个全局的 js 对象，并在 lua 中对它进行修改也不会影响到页面上下文。Element 对象是一个例外，它里面有一些可变的字段。

如果 JavaScript 抛出一个错误，它会作为一个 lua 错误，处理它最好的方式是在 JavaScript 代码中使用 try/catch ，因为在 JavaScript 转 lua 的过程中可能存在数据的丢失

您也可以参考: *splash:runjs* , *splash:evaljs* , *splash:wait_for_resume* , *splash:autoload* , treat.as_array , Element Object , *splash:select* , *splash:select_all* .

splash:evaljs

在页面上下文中执行 JavaScript 代码并返回最后一条语句的结果

原型: `result = splash:evaljs(snippet)`

参数:

- snippet : 一段可以执行的 JavaScript 代码的字符串

返回值: 返回 snippet 中最后一条语句的结果, 并将这个结果有 JavaScript 的类型转化为 lua 对应的类型。如果发生 JavaScript 异常或者语法错误，则会引发错误。

是否异步: 否

JavaScript 到 lua 的转化规则与*splash:jsfunc* 相同

在只需要执行一小段代码而不用涉及到闭包函数的时候，使用 `splash.eveljs` 将会十分方便, 例如

```
local title = splash:evaljs("document.title")
```

当您不需要返回值的时候不要使用*splash:jsfunc* 。这种方式十分低效，而且可能会带来一些问题可以使用*splash:runjs* 来代替。例如下面这段无辜的代码 (使用 using jQuery) 可能会做无用功

```
splash:evaljs("$(console.log('foo'));")
```

这段代码的一个问题是，它允许链接 jQuery $ 函数返回一个巨大的对象, 接着*splash:evaljs* 尝试对其进行序列化并将它转化为 lua 对象，这是一种对资源的浪费，但是*splash:jsfunc* 不会出现这个问题

如果您经过评估发现您的代码需要使用参数，相比于使用*splash:evaljs* 和格式化字符串的方式来说，使用*splash:jsfunc* 将会是一种更好的选择。请对比下面这段代码

```
function main(splash)

    local font_size = splash:jsfunc([[
        function(sel) {
            var el = document.querySelector(sel);
            return getComputedStyle(el)["font-size"];
        }
    ]])

    local font_size2 = function(sel)
        -- FIXME: escaping of `sel` parameter!
        local js = string.format([[
            var el = document.querySelector("%s");
            getComputedStyle(el)["font-size"]
        ]], sel)
        return splash:evaljs(js)
    end

    -- ...
end
```

请参考: *splash:runjs*, *splash:jsfunc* , *splash:wait_for_resume*, *splash:autoload* , Element Object, *splash:select*, *splash:select_all* .

splash:runjs

在页面上下文中执行 JavaScript 代码

原型: `ok, error = splash:runjs(snippet)`

参数:

- snippet: 一段可以被执行的 JavaScript 代码的字符串

返回值: `ok, error` 的元组, 如果执行结果正常 `ok` 的值为 `True` , 如果 JavaScript 代码发生错误, `ok` 的值为 `nil` 并且 "error" 是一个描述错误信息的字符串

是否异步: 否

示例

```
assert(splash:runjs("document.title = 'hello';"))
```

请注意使用语法 `function foo(){}` 定义的函数的作用返回并不是全局的

```
assert(splash:runjs("function foo(){return 'bar'}"))
local res = splash:evaljs("foo()")  -- 此处会返回一个错误
```

这是一个实现的细节：传递给*splash:runjs* 的代码是在闭包中执行的，您可以使用下面的方式来定义全局函数和变量

```
assert(splash:runjs("foo = function (){return 'bar'}"))
local res = splash:evaljs("foo()")  -- 这个位置将会返回 'bar'
```

如果代码中需要参数，使用*splash:jsfunc* 将会是一个更好的选择

请对比下面的代码

```
function main(splash)

    -- 滚动窗口到 (x, y) 位置的 lua 函数.
    function scroll_to(x, y)
        local js = string.format(
            "window.scrollTo(%s, %s);",
            tonumber(x),
            tonumber(y)
        )
        assert(splash:runjs(js))
    end

    -- 一个简单的使用 splash:jsfunc 的示例代码
    local scroll_to2 = splash:jsfunc("window.scrollTo")

    -- ...
end
```

您也可以参考: *splash:runjs* , *splash:jsfunc*, *splash:autoload* , *splash:wait_for_resume*

splash:wait_for_resume

以异步的方式在页面上下文中执行 JavaScript 代码，直到 JavaScript 代码告诉它恢复，Lua 脚本才会恢复然后继续执行

原型: `result, error = splash:wait_for_resume(snippet, timeout)`

参数:

- snippet: 一个可以被执行的 JavaScript 代码的字符串，这段代码必须包含一个可被调用的 main 函数,main 函数的第一个参数是一个包含了 `resume``和 ``error` 属性的对象，resume 是一个可以用来恢复 lua 执行的函数，它传入一个可选参数，这个参数将会以 `result.value` 的形式返回到 lua 中，`error` 是一个函数，当发生错误时会调用这个函数并返回一段包含错误信息字符串
- timeout: 它是一个数值，它决定了在强制返回到 lua 代码之前运行 JavaScript 代码运行多长时间 (以秒为单位)，默认值是 0，这表示将禁用这个超时

返回值: `result, error` 的元组。当代码成功执行，`result` 是一个 table，如果返回的值在 JavaScript 中未定义，则会在 `result` 中包含一个由 `splash.resume(…)` 返回的键值。`result` 也可以包含由 `splash.set(…)` 添加进来的键值对。如果执行 JavaScript 代码超时, `result` 的值将会为 `nil` 并且 `error` 会包含一个错误信息的字符串

是否为异步: 是

例子:

第一个小例子主要演示如何将代码执行的控制权由 lua 转移到 JavaScript，最后返回到 lua 中。这个命令主要告诉 JavaScript 代码休眠 3s，然后返回到 lua 中来，请注意：它使用异步的方式来执行，lua 事件循环和 JavaScript 事件循环将在暂停 3s 之后再运行。但是 lua 代码会一直等到 JavaScript 代码调用 ‘‘splash.resume()‘‘ 才会继续执行当前函数

```
function main(splash)
    local result, error = splash:wait_for_resume([[
        function main(splash) {
            setTimeout(function () {
                splash.resume();
            }, 3000);
        }
    ]])

    -- 返回值为 {}
    -- 错误值为 nil

end
```

返回值被设置为空的 table，通过 `splash.resume` 函数未返回任何值。即使 JavaScript 代码未返回任何值，您也可以使用 `assert(splash:wait_for_resume(…))` 因为对 `assert()` 返回空表表示为真

注解: 请注意，您的 JavaScript 代码必须包含 main 函数，如果您不包含它，将会得到一个错误。当然 main 函数的第一个参数的名称您可以随便取在这份文档中为了方便我们将称它为 splash

---

第二个例子将演示如果通过 JavaScript 代码返回对应的值到 lua 中，您可以返回布尔类型、数字类型、字符串类型、数组类型和 JavaScript 对象

```
function main(splash)

    local result, error = splash:wait_for_resume([[
        function main(splash) {
            setTimeout(function () {
                splash.resume([1, 2, 'red', 'blue']);
            }, 3000);
```

(continues on next page)

(续上页)

```
        }
    ]])

    -- result is {value={1, 2, 'red', 'blue'}}
    -- error is nil

end
```

注解: 与*splash:evaljs* 类似, 注意不要返回太大的 JavaScript 对象，类似于 jQuery 中的 `$` 。太大的对象在转化为 lua 时会消耗大量的时间和内存

您也可以通过使用函数 `splash.set(key, value)` 来在 JavaScript 代码中添加键值对，新增的键值对将会被包含在 `result` 中返回到 lua。下面的代码就演示了这种情况

```
function main(splash)

    local result, error = splash:wait_for_resume([[
        function main(splash) {
            setTimeout(function () {
                splash.set("foo", "bar");
                splash.resume("ok");
            }, 3000);
        }
    ]])

    -- result is {foo="bar", value="ok"}
    -- error is nil

end
```

下面的例子将演示一种 `splash:wait_for_resume()` 错误的调用方式, JavaScript 代码中不包含 main 函数, 此时 `result` 值为 `nil` 因为 `splash.resume()` 函数永远不会被调用, `error` 会返回一条包含错误信息的字符串来说明这个错误

```
function main(splash)

    local result, error = splash:wait_for_resume([[
        console.log('hello!');
    ]])

    -- result is nil
    -- error is "error: wait_for_resume(): no main() function defined"
```

(continues on next page)

(续上页)

```
end
```

下面一个例子将展示如何进行错误处理, 如果 splash.error(…) 代替 splash.resume() 被调用, result 返回值将会是 nil 而 error 将会包含一个由 splash.error(…) 给出的错误信息

```
function main(splash)
    local result, error = splash:wait_for_resume([[
        function main(splash) {
            setTimeout(function () {
                splash.error("Goodbye, cruel world!");
            }, 3000);
        }
    ]])

    -- result is nil
    -- error is "error: Goodbye, cruel world!"

end
```

您的 JavaScript 代码在某一时刻只能调用 splash.resume() 或者 splash.error() 中的任意一个。随后对这两个函数的调用都不起作用。下面这个例子展示了这一特性

```
function main(splash)

    local result, error = splash:wait_for_resume([[
        function main(splash) {
            setTimeout(function () {
                splash.resume("ok");
                splash.resume("still ok");
                splash.error("not ok");
            }, 3000);
        }
    ]])

    -- result is {value="ok"}
    -- error is nil

end
```

下面的例子将演示 timeout 参数的影响。我们将超时设置为 1s，但是 JavaScript 代码中 splash.resume() 函数将在 3s 后执行。这样确保 splash:wait_for_resume() 一定会超时.

当超时发生时 result 将会为 nil, error 将会包含一个字符串来解释超时, 并且 lua 代码将会继续执行, 在超时后调用 splash.resume() 或者 splash.error() 不起任何作用

```
function main(splash)

    local result, error = splash:wait_for_resume([[
        function main(splash) {
            setTimeout(function () {
                splash.resume("Hello, world!");
            }, 3000);
        }
    ]], 1)

    -- result is nil
    -- error is "error: One shot callback timed out while waiting for resume() or error()."

end
```

注解: 超时值必须要 >= 0，如果超时值为 0 `splash:wait_for_resume()` 永远不会超时 (但是 Splash’s HTTP API 中设置的超时仍然有用)

请确保您的 JavaScript 代码没有因为超时而被强制结束, 它可能会一直执行，直到 splash 关闭浏览器的页面上下文。

您可以参考: *splash:runjs* , *splash:jsfunc* , *splash:evaljs*

splash:autoload

设置 JavaScript 代码在每个页面加载时自动加载

原型: `ok, reason = splash:autoload{source_or_url, source=nil, url=nil}`

参数:

- source_or_url: 可以是一段 JavaScript 代码的字符串或者是 JavaScript 代码所对应的 url，以便在页面加载时执行
- source: 一段 JavaScript 代码的字符串
- url: 一个用于加载 JavaScript 源码的 url

返回值: `ok, reason` 元组

是否异步: 是。只有当远程资源的 url 被传递时采用的是异步

splash:autoload 允许在每个页面被加载时执行 JavaScript 代码，*splash:autoload* 并不会自己执行 JavaScript 代码，如果您想只执行一次 JavaScript 代码或者在页面加载完成之后执行，请使用*splash:runjs* 或者*splash:jsfunc*

splash.autoload 可以在页面加载之前预加载有用的 JavaScript 库，或者在页面要使用某些 JavaScript 对象之前先替换这些对象

例子

```
function main(splash, args)
  splash:autoload([[
    function get_document_title(){
      return document.title;
    }
  ]])
  assert(splash:go(args.url))

  return splash:evaljs("get_document_title()")
end
```

为了方便，当*splash.autoload* 第一个参数以 “http” 或者以 “https://” 开头时，认为它传进来的是一个 URL. 示例 2-确定某个远程的库是可达的

```
function main(splash, args)
  assert(splash:autoload("https://code.jquery.com/jquery-2.1.3.min.js"))
  assert(splash:go(splash.args.url))
  local version = splash:evaljs("$.fn.jquery")

  return 'JQuery version: ' .. version
end
```

您可以使用 “url” 或者 “source” 参数来防止函数自己判断参数

```
splash:autoload{url="https://code.jquery.com/jquery-2.1.3.min.js"}
splash:autoload{source="window.foo = 'bar';"}
```

当参数不变时通过这种方式来禁止函数进行参数判断是一个很好的使用方式

如果*splash.autoload* 多次被调用，那么所有的脚本都会在页面被加载时调用

如果不想每次页面加载都执行这段 JavaScript 代码可以使用*splash:autoload_reset*

您可以参考: *splash:evaljs* , *splash:runjs*, *splash:jsfunc*, *splash:wait_for_resume* , *splash:autoload_reset*

splash:autoload_reset

取消先前通过函数*splash:autoload* 向页面上下文中注册的所有 JavaScript 代码

原型: `splash:autoload_reset()`

返回值: nil

是否为异步: 否

当调用了*splash:autoload_reset* 之后，之前使用*splash:autoload* 注册的函数在后续的请求中将不再被执行。您可以再次使用*splash:autoload* 来设置一组不同的脚本代码

已经加载了的脚本将不会被移出当前的页面上下文

您可以参考*splash:autoload*

splash:call_later

安排一些回调函数在对应的延迟时间过后再调用。

原型: `timer = splash:call_later(callback, delay)`

参数:

- callback: 需要被执行的函数
- delay: 延迟时间

返回值: 一个允许取消挂起计时器的句柄或者注册回调时产生的异常

是否异步: 否

例子 1-在页面开始加载后的 1.5s 和 2.5s 分别获取一段 HTML 代码

```
function main(splash, args)
  local snapshots = {}
  local timer = splash:call_later(function()
    snapshots["a"] = splash:html()
    splash:wait(1.0)
    snapshots["b"] = splash:html()
  end, 1.5)
  assert(splash:go(args.url))
  splash:wait(3.0)
  timer:reraise()

  return snapshots
end
```

splash:call_later 返回一个句柄 (计时器) 如果需要取消任务，可以使用 `timer:cancel()`。如果一个回调已经开始，调用 `timer:cancel()` 将不会起作用

默认情况下，当通过*splash:call_later* 注册的回调中发生错误，回调将会停止执行，但是这不影响 main 函数中脚本的执行，您可以使用 `timer:reraise()` 来抛出异常

splash:call_later 定义的回调函数将会在后续执行，即使延迟时间为 0 它们也不会立即执行，当延迟为 0 时，不早于当前函数产生的事件循环，即不早于某些异步函数被调用

splash:http_get

发送一个 HTTP 的 GET 请求，并返回一个未经过浏览器加载的响应

原型: `response = splash:http_get{url, headers=nil, follow_redirects=true}`[15]

参数:

- url: 请求的 url
- headers: 一个用户新增或者替换初始请求头的 lua table 类型
- follow_redirects: 是否跟随重定向

返回值: 返回一个 Response 对象

是否异步: 是

例子

```
local reply = splash:http_get("http://example.com")
```

该函数调用不会修改当前页面中的上下文环境和 URL，如果要通过浏览器来加载 web 页面，请调用函数*splash:go*

您可以参考: *splash:http_post* , Response Object

splash:http_post

发送一个 HTTP 的 POST 请求，并返回一个未经过浏览器加载的响应

原型: `response = splash:http_post{url, headers=nil, follow_redirects=true, body=nil}`

参数:

- url: 请求的 url
- headers: 一个用户新增或者替换初始请求头的 lua table 类型
- follow_redirects: 是否跟随重定向
- body: 用字符串表示的请求的请求体，如果您打算将数据提交到表单，body 应该进行 url 编码

返回值: 返回一个 Response 对象

是否异步: 是一个提交 form 表单的例子

```
local reply = splash:http_post{url="http://example.com", body="user=Frank&password=hunter2"}
-- reply.body contains raw HTML data (as a binary object)
-- reply.status contains HTTP status code, as a number
-- see Response docs for more info
```

[15] lua 函数调用有两种形式，带形参名称的使用 ‘‘{}‘‘ 不带的使用 ()

一个关于 JSON 的例子

```
json = require("json")

local reply = splash:http_post{
    url="http://example.com/post",
    body=json.encode({alpha="beta"}),
    headers={["content-type"]="application/json"}
}
```

该函数调用不会修改当前页面中的上下文环境和 URL，如果要通过浏览器来加载 web 页面，请调用函数*splash:go*

您可以参考: *splash:http_post* , json , Response Object

splash:set_content

设置当前页面的上下文环境，并且等待直到页面加载

原 型: `ok, reason = splash:set_content{data, mime_type="text/html; charset=utf-8", baseurl=""}`

参数:

- data: 新的页面上下文
- mime_type: 上下文的 MIME 类型
- baseurl: 页面中引用的外部对象的相对路径通过 baseurl 来定位

返回值: `ok`，`reason` 的元组, 如果 `ok` 为空，表明在加载页面的时候发生了一些异常，`reason` 包含的发生的错误的类型信息

是否为异步: 是

例子

```
function main(splash)
    assert(splash:set_content("<html><body><h1>hello</h1></body></html>"))
    return splash:png()
end
```

splash:html

返回整个页面的 HTML 代码 (以字符串的形式返回)

原型: `html = splash:html()`

返回值: 页面内容 (以字符串的形式)

是否异步: 否

例子:

```
-- A simplistic implementation of render.html endpoint
function main(splash)
    splash:set_result_content_type("text/html; charset=utf-8")
    assert(splash:go(splash.args.url))
    return splash:html()
end
```

没有什么能阻止我们获取多个 HTML 快照。例如我们先加载一个站点的 3 个页面，为每个页面存储它初始的 HTML 快照，然后等待 0.5s，再加载下一个

```
treat = require("treat")

-- Given an url, this function returns a table
-- with the page screenshoot, it's HTML contents
-- and it's title.
function page_info(splash, url)
  local ok, msg = splash:go(url)
  if not ok then
    return {ok=false, reason=msg}
  end
  local res = {
    html=splash:html(),
    title=splash:evaljs('document.title'),
    image=splash:png(),
    ok=true,
  }
  return res
end

function main(splash, args)
  -- visit first 3 pages of hacker news
  local base = "https://news.ycombinator.com/news?p="
  local result = treat.as_array({})
  for i=1,3 do
    local url =  base .. i
    result[i] = page_info(splash, url)
  end
  return result
end
```

splash:png

返回当前页面指定尺寸的屏幕截图

原 型: `png = splash:png{width=nil, height=nil, render_all=false, scale_method='raster', region=nil}`

参数:

- width: 可选值，以像素为单位的截图的宽
- height: 可选值, 以像素为单位的截图的高
- render_all: 可选值, 如果为 `true` 则返回整个页面的截图
- scale_method: 可选值, 调整图片大小时所以用的方法，取值为 `'raster'``(位图) 和 ``'vector'` 矢量图
- region: 可选值, `{left, top, right, bottom}` 表示的裁剪矩形的坐标

返回值: 以 binary object 形式返回的 PNG 截图，如果结果为空会返回 `nil`

是否为异步: 否

如果不传参数，`splash:png()` 则会返回当前视框的截图

width 参数设置返回图片的宽度，如果视口的宽度与设置的宽度不同，图片会放大或者缩小，匹配指定的图片大小。例如假设视口的宽度为 1024px 而设置 `splash:png{width=100}` 将会返回一个完整视口的截图，但是图片会被缩小到宽度为 100px

height 参数设置返回图片的高度，视口的高度与设置的高度不同，图片会被裁剪或者扩展以便匹配指定大小，但是不调整图片内容的大小。通过这种扩展创建的区域是透明的。

您可以使用*splash:set_viewport_size* , *splash:set_viewport_full* 或者使用参数 render_all 来对视口进行设置 `render_all = true` 相当于在开始渲染前调用 `splash:set_viewport_full()` 然后再恢复视口大小

您可以使用 region 参数来指定截取渲染页面的哪个部分, 它是一个由 `{left, top, right, bottom}` 组成的 table 对象。它的坐标值与当前滚动的位置有关，目前传入的坐标值必须在视口中。您可以在渲染前调用*splash:set_viewport_full* 接着再调用 `splash:png` 来确保您传入的坐标值永远在视口中。在后续的 splash 版本中可能会修复这个问题

使用 `region` 参数或者使用一小段 JavaScript 代码很容易实现只渲染某一个 HTML DOM 元素，例如下面的例子

```
-- This in an example of how to use lower-level
-- Splash functions to get element screenshot.
--
-- In practice use splash:select("a"):png{pad=32}.



-- this function adds padding around region
```

(continues on next page)

(续上页)

```
function pad(r, pad)
  return {r[1]-pad, r[2]-pad, r[3]+pad, r[4]+pad}
end

function main(splash, args)
  -- this function returns element bounding box
  local get_bbox = splash:jsfunc([[
    function(css) {
      var el = document.querySelector(css);
      var r = el.getBoundingClientRect();
      return [r.left, r.top, r.right, r.bottom];
    }
  ]])

  -- main script
  assert(splash:go(splash.args.url))
  assert(splash:wait(0.5))

  -- don't crop image by a viewport
  splash:set_viewport_full()

  -- let's get a screenshot of a first <a>
  -- element on a page, with extra 32px around it
  local region = pad(get_bbox("a"), 32)
  return splash:png{region=region}
end
```

另一种简单的方法是使用 element:png

```
splash:select('#my-element'):png()
```

scale_method 参数的值必须是 `'raster'` 或者 `'vector'` 其中的一个，当 `scale_method='raster'` 时，图像是按照像素来调整大小的，当 `scale_method='vector'` 时，在渲染过程中图像按照每个元素来调整大小. 矢量缩放更加高效，而且图像更加清晰, 但是它可能导致渲染失真，所以请谨慎的使用这种方式

`splash:png` 返回的是一个二进制对象 binary object，因此您可以直接将其作为返回值在 main 函数中返回，它将作为二进制图像数据与适当的 content-type 头一起返回：

```
-- A simplistic implementation of render.png
-- endpoint.
function main(splash, args)
  assert(splash:go(args.url))

  return splash:png{
```

(continues on next page)

(续上页)

```
    width=args.width,
    height=args.height
  }
end
```

splash:png() 的结果将会作为一个 table 对象进行返回, 它会以 base64 的方式进行编码以便嵌入到 json 数据中，在客户端中创建一个 data:uri

```
function main(splash)
    assert(splash:go(splash.args.url))
    return {png=splash:png()}
end
```

当图片为空时，*splash:png* 返回 `nil` ，如果您想 splash 抛出一个错误，请使用 `assert`

```
function main(splash)
    assert(splash:go(splash.args.url))
    local png = assert(splash:png())
    return {png=png}
end
```

您可以参考*splash:jpeg* , Binary Objects , *splash:set_viewport_size* , *splash:set_viewport_full* , element:jpeg , element:png

splash:jpeg

返回指定尺寸的屏幕截图，以 JPEG 格式返回

原型: `jpeg = splash:jpeg{width=nil, height=nil, render_all=false, scale_method='raster', quality=75, region=nil}`

参数:

- width: 可选值，以像素为单位的截图的宽
- height: 可选值, 以像素为单位的截图的高
- render_all: 可选值, 如果为 `true` 则返回整个页面的截图
- scale_method: 可选值, 调整图片大小时所以用的方法，取值为 `'raster'``(位图) 和 ``'vector'` 矢量图
- quality: 可选值，返回 JPEG 图片的质量, 取 1 到 100 的整数值
- region: 可选值, `{left, top, right, bottom}` 表示的裁剪矩形的坐标

返回值: 以 binary object 形式返回的 JPEG 截图，

width 参数设置返回图片的宽度，如果视口的宽度与设置的宽度不同，图片会放大或者缩小，匹配指定的图片大小。例如假设视口的宽度为 1024px 而设置 `splash:jpeg{width=100}` 将会返回一个完整视口的截图，但是图片会被缩小到宽度为 100px

height 参数设置返回图片的高度，视口的高度与设置的高度不同，图片会被裁剪或者扩展以便匹配指定大小，但是不调整图片内容的大小。通过这种扩展创建的区域是透明的。

您可以使用*splash:set_viewport_size* , *splash:set_viewport_full* 或者使用参数 render_all 来对视口进行设置 `render_all = true` 相当于在开始渲染前调用 `splash:set_viewport_full()` 然后再恢复视口大小

您可以使用 region 参数来指定截取渲染页面的哪个部分，它是一个由 `{left, top, right, bottom}` 组成的 table 对象。它的坐标值与当前滚动的位置有关，目前传入的坐标值必须在视口中。您可以在渲染前调用*splash:set_viewport_full* 接着再调用 `splash:jpeg` 来确保您传入的坐标值永远在视口中。在后续的 splash 版本中可能会修复这个问题

使用一小段 JavaScript 代码配合 `region` 参数可以实现只截取单一 HTML DOM 元素图片的功能，您可以参考*splash:png* 文档中的*例子* 。另一种方法是使用 element:jpeg

scale_method 参数的值必须是 `'raster'` 或者 `'vector'` 其中的一个，当 `scale_method='raster'` 时，图像是按照像素来调整大小的，当 `scale_method='vector'` 时，在渲染过程中图像按照每个元素来调整大小. 矢量缩放更加高效，而且图像更加清晰, 但是它可能导致渲染失真，所以请谨慎的使用这种方式

quality 参数的值必须是 `0` 到 `100` 之间的整数值, 当这个数值超过 `95` 时需要特别注意, `quality=100` 会禁用部分 JPEG 的压缩算法, 并会产生一个大文件，而这个文件对图像质量的增益几乎没有任何效益

`splash:jpeg` 返回的是一个二进制对象 binary object ，因此您可以直接将其作为返回值在 main 函数中返回，它将作为二进制图像数据与适当的 content-type 头一起返回：

```
-- A simplistic implementation of render.jpeg endpoint
function main(splash, args)
    assert(splash:go(args.url))
    return splash:jpeg{
       width=args.width,
       height=args.height
    }
end
```

`splash:jpeg()` 的结果将会作为一个 table 对象进行返回, 它会以 base64 的方式进行编码以便嵌入到 json 数据中，在客户端中创建一个 data:uri

```
function main(splash)
    assert(splash:go(splash.args.url))
    return {jpeg=splash:jpeg()}
end
```

当图片为空时，*splash:jpeg* 返回 `nil` ，如果您想 splash 抛出一个错误，请使用 `assert`

```
function main(splash)
    assert(splash:go(splash.args.url))
    local jpeg = assert(splash:jpeg())
    return {jpeg=jpeg}
end
```

您可以参考*splash:png* , Binary Objects , *splash:set_viewport_size* , *splash:set_viewport_full* , element:jpeg , element:png

请注意在 1.2..5x 版本之后，`splash:jpeg` 的速度要优于 `splash:png`

splash:har

原型: `har = splash:har{reset=false}`

参数:

- reset: 可选值, 当值为 `true` 时，每次拍快照之前会清除之前的记录

返回值: 返回页面加载的相关信息，发生的事件，发送的网络请求以及请求的响应信息，这些信息都被存储成 HAR 格式

是否异步: 否

您可以使用*splash:har* 来获取关于网络请求的信息和 splash 的其他行为。如果您的脚本在顶层的 “har” 键中返回了 splash:har() 得到的结果，在 splash UI 中将会以很好的格式展示这些数据 (就像 Firefox 中的 “Network” 选项卡或者 Chrome 中的开发者工具)

```
function main(splash)
    assert(splash:go(splash.args.url))
    return {har=splash:har()}
end
```

默认情况下当某些请求被创建 (例如*splash:go* 函数被多次调用), HAR 数据会被累计并组合成单个对象。(日志仍然按页面进行分组)

如果您只想更新对应信息，请使用 `reset` 参数，它会清理之前所有已存在的日志，然后重新开始记录

```
function main(splash, args)
    assert(splash:go(args.url1))
    local har1 = splash:har{reset=true}
    assert(splash:go(args.url2))
    local har2 = splash:har()
    return {har1=har1, har2=har2}
end
```

默认情况下，返回的 HAR 数据不包含响应体的内容，您可以使用*splash.response_body_enabled* 属性或者 request:enable_response_body 方法

您也可以参考: *splash:har_reset* , *splash:on_response* , *splash.response_body_enabled* , request:enable_response_body .

splash:har_reset

原型: `splash:har_reset()`

返回值: `nil`

是否异步: 否

删除所有内部存储的 HAR 记录, 它与使用 `splash:har{reset=true}` 相似，但是不返回任何值

您可以参考*splash:har*

splash:history

原型: `entries = splash:history()`

返回值: 返回页面加载时的请求与响应信息，以 HAR entries 格式返回

是否异步: 否

`splash:history` 返回中不包含相关资源的信息，像图片，脚本，样式或者 AJAX 请求，如果您需要这方面的信息，请使用*splash:har* 或者*splash:on_response* .

让我们来获取一个 JSON 数组，其中包含我们想显示的响应头的信息

```
function main(splash)
    assert(splash:go(splash.args.url))
    local entries = splash:history()
    -- #entries means "entries length"; arrays in Lua start from 1
    local last_entry = entries[#entries]
    return {
       headers = last_entry.response.headers
    }
end
```

您可以参考: *splash:har*, *splash:on_response*

splash:url

原型: `url = splash:url()`

返回值: 当前的 url

是否异步: 否

splash:get_cookies

原型: `cookies = splash:get_cookies()`

返回值: 返回 CookieJar 的内容, 一个包含所有脚本可用的 cookie 的数组，结果以 HAR cookies 格式返回

是否异步: 否

一个返回值的例子

```
[
    {
        "name": "TestCookie",
        "value": "Cookie Value",
        "path": "/",
        "domain": "www.example.com",
        "expires": "2016-07-24T19:20:30+02:00",
        "httpOnly": false,
        "secure": false,
    }
]
```

splash:add_cookie

添加一个 cookie

原 型: `cookies = splash:add_cookie{name, value, path=nil, domain=nil, expires=nil, httpOnly=nil, secure=nil}`

是否异步: 否

例子

```
function main(splash)
    splash:add_cookie{"sessionid", "237465ghgfsd", "/", domain="http://example.com"}
    splash:go("http://example.com/")
    return splash:html()
end
```

splash:init_cookies

通过传入的 cookie 来重新设置当前 cookie

原型: `splash:init_cookies(cookies)`

参数:

- cookies: 一个用 lua 的 table 表示的需要设置的 cookie 值，与*splash:get_cookies* 返回值的格式相同

返回值: nil

是否异步: 否

例 1：保存并重新设置 cookie

```
local cookies = splash:get_cookies()
-- ... do something ...
splash:init_cookies(cookies)  -- restore cookies
```

例 2：手工初始化 cookie

```
splash:init_cookies({
    {name="baz", value="egg"},
    {name="spam", value="egg", domain="example.com"},
    {
        name="foo",
        value="bar",
        path="/",
        domain="localhost",
        expires="2016-07-24T19:20:30+02:00",
        secure=true,
        httpOnly=true,
    }
})

-- do something
assert(splash:go("http://example.com"))
```

splash:clear_cookies

清理所有的 cookie

原型: n_removed = splash:clear_cookies()

返回值: 返回被删除 cookie 的数量

是否异步: 否

如果只删除指定的 cookie 请使用*splash:delete_cookies*

splash:delete_cookies

删除指定的 cookie

原型: n_removed = splash:delete_cookies{name=nil, url=nil}

参数:

- name: 字符串类型，可选参数；所有 name 值为此值的 cookie 都将被删除
- url: 字符串类型，可选参数，所有应该发送到此地址的 cookie 将被删除

返回值: 被删除的 cookie 的个数

当 name 和 url 都为 nil 的时候，这个函数不做任何事情，您可以使用*splash:clear_cookies* 来删除所有 cookie

splash:lock_navigation

锁定导航栏

原型: `splash:lock_navigation()`

是否异步: 否

当调用这个函数之后，页面导航就不再离开当前页面，页面被锁定在当前 url 上

splash:unlock_navigation

解锁导航栏

原型: `splash:unlock_navigation()`

是否异步: 否

当调用这个函数之后，就允许导航离开当前页面。请注意之前由于*splash:lock_navigation* 限制的待处理的请求不会被重新加载

splash:set_result_status_code

设置返回给客户端的 HTTP 的状态码

原型: `splash:set_result_status_code(code)`

参数:

- HTTP 状态码 (是一个 >= 200 <= 999 的整数)

返回值: nil

是否为异步: 否

通过此功能可以向 splash 客户端发送 HTTP 状态码，以便向客户端报告相关错误

例子:

```
function main(splash)
    local ok, reason = splash:go("http://www.example.com")
    if reason == "http500" then
        splash:set_result_status_code(503)
```

(continues on next page)

(续上页)

```
        splash:set_result_header("Retry-After", 10)
        return ''
    end
    return splash:png()
end
```

在使用这个函数时要注意: 某些代码可能配置成会根据不同的响应码来进行不同的处理。例如您可以查看 nginx 的 proxy_next_upstream 选项

当存在未处理的 lua 错误时，不论之前调用*splash:set_result_status_code* 设置了何值, 最终都会返回 400.

您也可以参考: *splash:set_result_content_type*, *splash:set_result_header*

splash:set_result_content_type

设置返回给客户端结果的 Content-Type 值

原型: `splash:set_result_content_type(content_type)`

参数:

- content_type: 一个表示头中 Content-Type 键值的字符串

返回值: `nil`

是否异步: 否

如果 main 函数的返回值是一个 table，那么这个函数将不起作用: Content-Type 会被设置成 `application/json`

这个函数并不是设置使用*splash:go* 发送原始请求的 Content-Type，而是设置返回给客户端的结果的 Content-Type

例子:

```
function main(splash)
    splash:set_result_content_type("text/xml")
    return [[
       <?xml version="1.0" encoding="UTF-8"?>
       <note>
           <to>Tove</to>
           <from>Jani</from>
           <heading>Reminder</heading>
           <body>Don't forget me this weekend!</body>
       </note>
    ]]
end
```

您可以参考:

- *splash:set_result_header* 这个函数允许用户自定义任意的头信息，而不仅仅是 Content-Type
- Binary Objects 它有它自己的方法来设置返回值的 Content-Type

splash:set_result_header

设置返回给客户端结果的 header 值

原型: `splash:set_result_header(name, value)`

参数:

- name: 返回 header 的键名称
- value: 对应的键值

返回值: nil

这个函数并不是设置使用*splash:go* 发送原始请求的 headers, 而是设置通过 splash 返回到客户端的 headers

例 1：在 header 中设置 “foo=bar”

```
function main(splash)
    splash:set_result_header("foo", "bar")
    return "hello"
end
```

例 2：获取将屏幕快照转化为 PNG 图片所需要的时间，并通过 header 返回

```
function main(splash)

    -- this function measures the time code takes to execute and returns
    -- it in an HTTP header
    function timeit(header_name, func)
        local start_time = splash:get_perf_stats().walltime
        local result = func()  -- it won't work for multiple returned values!
        local end_time = splash:get_perf_stats().walltime
        splash:set_result_header(header_name, tostring(end_time - start_time))
        return result
    end

    -- rendering script
    assert(splash:go(splash.args.url))
    local screenshot = timeit("X-Render-Time", function()
       return splash:png()
    end)
    splash:set_result_content_type("image/png")
```

(continues on next page)

(续上页)

```
    return screenshot
end
```

您也可以看看: *splash:set_result_status_code* , *splash:set_result_content_type* .

splash:get_viewport_size

获取浏览器视口的尺寸

原型: `width, height = splash:get_viewport_size()`

返回值: 以像素为单位返回两个数值——视口的宽和高

是否异步: 否

splash:set_viewport_size

设置浏览器视口的大小

原型: `splash:set_viewport_size(width, height)`

参数:

- width: 整数值，视口的宽
- height: 整数值, 视口的高

返回值: nil

是否异步: 否

这个函数可能会修改可见区域大小和随后的渲染命令, 例如它可能会修改之后调用*splash:png* 将产生具有指定大小的图像

splash:png 将会使用这个视口大小

例子

```
function main(splash)
    splash:set_viewport_size(1980, 1020)
    assert(splash:go("http://example.com"))
    return {png=splash:png()}
end
```

注解: 这将重新布局所有的 document 元素并影响 geometry 变量, 像 `window.innerWidth` 和 `window.innerHeight` 。然而 `window.onresize` 事件回调将会在下一次异步操作时被调用，很明显*splash:png* 属于同

步操作。因此如果您希望在修改视口大小并希望它在执行截屏之前作出相应的调整，您可以使用*splash:wait*来等待。

---

splash:set_viewport_full

调整浏览器视口大小以便适应整个页面

原型: `width, height = splash:set_viewport_full()`

返回值: 返回两个数值，当前视口被设置的宽和高，以像素为单位

是否异步: 否

`splash:set_viewport_full` 只有在页面被加载之后才能调用,有时候需要等待一段时间 (使用 `splash:wait <splash-wait>`)。这是一个糟糕的限制，但是这是使自动调整大小这一行为工作正常的唯一办法

您可以参考*splash:set_viewport_size* 来获取与 JS 交互的相关内容

splash:png 将会使用这个设置的视口大小

例子:

```
function main(splash)
    assert(splash:go("http://example.com"))
    assert(splash:wait(0.5))
    splash:set_viewport_full()
    return {png=splash:png()}
end
```

splash:set_user_agent

重新设置后续所有请求头中的 User-Agent 的值

原型: `splash:set_user_agent(value)`

参数:

- value: 一个表示 HTTP 请求头中 UA 的字符串

返回值: nil

是否异步: 否

splash:set_custom_headers

设置每个请求的 HTTP 请求头

原型: `splash:set_custom_headers(headers)`

参数:

- headers: 一个表示请求头的 LUA 的 table 数据

返回值: nil

是否异步: 否

这里设置的 headers 将会与 WebKit 默认的 headers 合并, 在发生冲突时会覆盖 WebKit 中的值

使用 `splash:set_custom_headers` 设置的请求值并不会作用在*splash:go* 的请求上，也就是说值不会进行合并。*splash:go* 使用的 headers 拥有更高的优先级

例子

```
splash:set_custom_headers({
   ["Header-1"] = "Value 1",
   ["Header-2"] = "Value 2",
})
```

注解: 这个函数不支持使用参数名传参的方式

您也可以参考: *splash:on_request*

splash:get_perf_stats

返回一个与统计相关的表现资料

原型: `stats = splash:get_perf_stats()`

返回值: 返回一个对行为分析有用的 table

是否异步: 否

就目前来说，这个 table 包含如下值:

- `walltime` : float 型，返回 epoch 时间 (从 1970 0 点开始的时间), 类似于 lua 中的 `os.clock`
- `cputime` : float 型, splash 进程消耗的 CPU 时间
- `maxrss` : int 型, splash 消耗的内存字节数的高位

splash:on_request

注册一个函数，每当发送 HTTP 请求的时候调用

原型: `splash:on_request(callback)`

参数:

- callback: 每次发送 HTTP 请求前被调用的函数

是否异步: 否

splash:on_request 的回调接收一个 splash 参数 (一个 Request 对象)

获取更多关于请求的信息，您可以使用请求对象的 属性 , 如果要在发起请求前修改或者丢弃对应请求，请使用请求对象的相关 方法

通过*spash:on_request* 注册的回调函数中不能调用 Splash 中的异步函数，像*splash:go* 或者*splash:wait*

例 1: 使用 request.url 属性记录所有的请求 URL

```
treat = require("treat")

function main(splash, args)
  local urls = {}
  splash:on_request(function(request)
    table.insert(urls, request.url)
  end)

  assert(splash:go(splash.args.url))
  return treat.as_array(urls)
end
```

例 2: 通过 request.info 属性来记录请求信息，但是不直接存储

```
treat = require("treat")
function main(splash)
    local entries = treat.as_array({})
    splash:on_request(function(request)
        table.insert(entries, request.info)
    end)
    assert(splash:go(splash.args.url))
    return entries
end
```

例 3: 丢弃所有针对 ".css" 资源的请求 (请参考: request:abort)

```
splash:on_request(function(request)
    if string.find(request.url, ".css") ~= nil then
        request.abort()
    end
end)
```

例 4: 替换资源 (请参阅: request:set_url)

```
splash:on_request(function(request)
    if request.url == 'http://example.com/script.js' then
```

(continues on next page)

(续上页)

```
        request:set_url('http://mydomain.com/myscript.js')
    end
end)
```

例 5：设置自定义的代理服务器，并将相关凭据传递给 Splash 的 HTTP 请求 (请参阅: request:set_proxy)

```
splash:on_request(function(request)
    request:set_proxy{
        host = "0.0.0.0",
        port = 8990,
        username = splash.args.username,
        password = splash.args.password,
    }
end)
```

例 6：丢弃响应时间超过 5s 的请求，但是针对第一个请求允许它不超过 15s(请参阅 request:set_timeout)

```
local first = true
splash.resource_timeout = 5
splash:on_request(function(request)
    if first then
        request:set_timeout(15.0)
        first = false
    end
end)
```

注解: *splash:on_request* 不能使用命名方式传参

您也可以参考: *splash:on_response* , *splash:on_response_headers* , *splash:on_request_reset* , treat , Request Object

splash:on_response_headers

注册一个回调函数，这个函数在接收响应头之后，但是在接收到响应体之前被调用

原型: `splash:on_response_headers(callback)`

参数:

- callback: 在接收到响应头之后，但是在接收到响应体之前被调用的 Lua 函数

返回值: nil

是否异步: 否

splash:on_response_headers 传入单个的 `response` 参数 (一个 Response Object)

response.body 在*splash:on_response_headers* 中无效，因为此时并没有接收到响应体，*splash:on_response_headers* 方法使用的关键在于您可以根据具体情况调用 response:abort 选择放弃接收响应体

在*splash:on_response_headers* 定义的回调函数中无法使用 Splash 中的异步函数，像*splash:go* 或者*splash:wait* 。`response` 对象在退出回调函数后被清理。所以您在回调函数之外无法使用这个对象

例 1：记录在渲染时获取到的所有响应头的 content-type 值

```
function main(splash)
    local all_headers = {}
    splash:on_response_headers(function(response)
        local content_type = response.headers["Content-Type"]
        all_headers[response.url] = content_type
    end)
    assert(splash:go(splash.args.url))
    return all_headers
end
```

例 2：放弃接收响应头中 Content-Type 值为 `text/css` 的响应体

```
function main(splash, args)
  splash:on_response_headers(function(response)
    local ct = response.headers["Content-Type"]
    if ct == "text/css" then
      response.abort()
    end
  end)

  assert(splash:go(args.url))
  return {
    png=splash:png(),
    har=splash:har()
  }
end
```

例 3：在不获取响应体的情况下提取站点中的所有 cookie

```
function main(splash)
local cookies = ""
splash:on_response_headers(function(response)
    local response_cookies = response.headers["Set-cookie"]
    cookies = cookies .. ";" .. response_cookies
    response.abort()
```

(continues on next page)

(续上页)

```
end)
assert(splash:go(splash.args.url))
return cookies
end
```

注解: *splash:on_response_headers* 不能使用命名方式传参

您也可以参考: *splash:on_request* , *splash:on_response* , *splash:on_response_headers_reset* , Response Object .

splash:on_response

注册一个回调函数，当响应下载完成之后调用

原型: `splash:on_response(callback)`

参数:

- callback: 每当下载完成之后需要调用的函数

返回值: nil

是否为异步: 否

splash:on_response 回调函数接受单个 `response` 参数 (一个 Response 对象)

默认情况下，`response` 对象没有 response.body 属性。如果您想开启这个属性，请使用*splash.response_body_enabled* 选项或者调用 request:enable_response_body 方法

注解: *splash:on_response* 不允许使用参数名称进行传参

您可以参考: *splash:on_request* , *splash:on_response_headers* , *splash:on_response_reset* , Response Object , *splash.response_body_enabled* , request:enable_response_body .

splash:on_request_reset

删除由*splash:on_request* 函数注册的所有回调函数

原型: `splash:on_request_reset()`

返回值: nil

是否异步: 否

splash:on_response_headers_reset

清理所有由*splash:on_response_headers* 注册的回调函数

原型: `splash:on_response_headers_reset()`

返回值: nil

是否异步: 否

splash:on_response_reset

移除所有由*splash:on_response* 函数注册的回调函数

原型: `splash:on_response_reset()`

返回值: nil

是否异步: 否

splash:get_version

获取 Splash 的主版本和次版本号

原型: `version_info = splash:get_version()`

返回值: 一个包含版本信息的 table 结构

是否异步: 否

目前，这个 table 主要包含如下内容:

- `splash` : (string) Splash 版本
- `major` : (int) Splash 主版本
- `minor` : (int) Splash 次版本
- `python` : (string) Python 版本号
- `qt` : (string) QT 的版本
- `webkit`: (string) WebKit 版本
- `sip` : (string) SIP 版本
- `twisted`: (string) Twisted 版本

示例:

```
function main(splash)
    local version = splash:get_version()
    if version.major < 2 and version.minor < 8 then
```

(continues on next page)

(续上页)

```
        error("Splash 1.8 or newer required")
    end
end
```

splash:mouse_click

在 Web 页面中触发一个鼠标点击的消息

原型: `splash:mouse_click(x, y)`

参数:

- x: 需要点击元素的 x 坐标的值 (距左侧的距离，相对于当前视口)
- y: 需要点击元素的 y 坐标的值 (距上方的距离，相对于当前视口)

返回值: nil

是否异步: 否

鼠标事件的坐标必须与当前视口相关联

如果您想在某个元素上点击鼠标，一个简单的办法是使用*splash:select* 和 element:mouse_click

```
local button = splash:select('button')
button:mouse_click()
```

您也可以使用*splash:mouse_click* 来实现它，使用 JavaScript 代码来获取对应元素的坐标

```
-- Get button element dimensions with javascript and perform mouse click.
function main(splash)
    assert(splash:go(splash.args.url))
    local get_dimensions = splash:jsfunc([[
        function () {
            var rect = document.getElementById('button').getClientRects()[0];
            return {"x": rect.left, "y": rect.top}
        }
    ]])
    splash:set_viewport_full()
    splash:wait(0.1)
    local dimensions = get_dimensions()
    -- FIXME: button must be inside a viewport
    splash:mouse_click(dimensions.x, dimensions.y)

    -- Wait split second to allow event to propagate.
    splash:wait(0.1)
```

(continues on next page)

(续上页)

```
    return splash:html()
end
```

与 element:mouse_click 不同，*splash:mouse_click* 不是异步的，鼠标消息不会立即得到响应，为了查看鼠标点击事件执行后页面的变化，您必须要在调用*splash:mouse_click* 之后调用*splash:wait*

执行该操作的元素必须在视口之内 (必须对用户可见)，如果元素在视口之外，您需要滚动视口以便让其可见，您可以选择滚动该元素 (使用 JavaScript 代码、*splash.scroll_position* 或者 `element:scrollIntoViewIfNeeded()`) 或者使用函数*splash:set_viewport_full* 来将视口设置为整个页面大小

注解: 与*splash:mouse_click* 不同, element:mouse_click 会自动滚动屏幕

在 splash 引擎中, *splash:mouse_click* 会先执行*splash:mouse_press* 再执行*splash:mouse_release*

目前只支持鼠标左键点击事件

您可以参考: element:mouse_click , *splash:mouse_press* , *splash:mouse_release* , *splash:mouse_hover*, *splash.scroll_position*

splash:mouse_hover

触发 Web 页面中鼠标悬停消息 (JavaScript 中的 mouseover 消息)

原型: `splash:mouse_hover(x, y)`

参数:

- x: 需要触发悬停事件的元素所在位置的 x 坐标 (距左边的距离，相对于当前视口来说)
- y: 需要触发悬停事件的元素所在位置的 y 坐标 (距上边的距离，相对于当前视口来说)

返回值: nil

是否异步: 否

请在*splash:mouse_click* 中参阅相关的鼠标事件

您也可以参考: element:mouse_hover

splash:mouse_press

触发页面中鼠标按下的事件

原型: `splash:mouse_press(x, y)`

参数:

- x: 需要触发鼠标左键按下事件的元素所在位置的 x 坐标 (距左边的距离，相对于当前视口来说)
- y: 需要触发鼠标左键按下事件的元素所在位置的 y 坐标 (距上边的距离，相对于当前视口来说)

返回值: nil

是否异步: 否

请在*splash:mouse_click* 中参阅相关的鼠标事件

splash:mouse_release

触发页面中鼠标键抬起的事件

原型: `splash:mouse_release(x, y)`

参数:

- x: 需要触发鼠标左键抬起事件的元素所在位置的 x 坐标 (距左边的距离，相对于当前视口来说)
- y: 需要触发鼠标左键抬起事件的元素所在位置的 y 坐标 (距上边的距离，相对于当前视口来说)

返回值: nil

是否异步: 否

请在*splash:mouse_click* 中参阅相关的鼠标事件

splash:with_timeout

设置执行函数的超时值

原型: `ok, result = splash:with_timeout(func, timeout)`

参数:

- func: 需要设置超时值的函数
- timeout: 超时值, 单位为秒

返回值: `ok, result` 的元组; 如果 `ok` 的值不为 `true` 表明在执行函数的过程中出现对应的错误，或者超时。`result` 将会保存错误的类型信息, 如果 `result` 等于 `timeout` 则说明指定的超时时间已过。当 `ok` 为 `true` 时, `result` 中包含函数执行的结果。如果您的函数返回多个值, 它们会被返回到后面多个 `result` 变量中

是否异步: 是

例 1:

```
function main(splash, args)
  local ok, result = splash:with_timeout(function()
    -- try commenting out splash:wait(3)
    splash:wait(3)
```

(continues on next page)

(续上页)

```
    assert(splash:go(args.url))
  end, 2)

  if not ok then
    if result == "timeout_over" then
      return "Cannot navigate to the url within 2 seconds"
    else
      return result
    end
  end
  return "Navigated to the url within 2 seconds"
end
```

例 2：函数返回多个值:

```
function main(splash)
    local ok, result1, result2, result3 = splash:with_timeout(function()
        splash:wait(0.5)
        return 1, 2, 3
    end, 1)

    return result1, result2, result3
end
```

请注意，如果函数执行时间超过了设置的超时值，那么 splash 会尝试中断函数的执行。但是 splash 是以 协作式多任务 的方式在运行，因此在某些时候 Spalash 无法停止执行超时的函数。换句话说，协作式多任务模式意味着管理程序 (在这个例子中管理程序是 Splash 脚本引擎) 在对应子程序没有要求的情况下不会主动结束子程序。在 splash 脚本中，只有在调用某些异步操作时，这些操作才会被结束[16] 。反之，同步函数无法被停止

注解: splash 是以 协作式多任务 的方式在运行。在进行同步调用的时候，您需要额外的小心

让我们在例子中看一下它们的不同

```
function main(splash)
    local ok, result = splash:with_timeout(function()
        splash:go(splash.args.url) -- 执行到此处时任务可以被停止
        splash:evaljs(long_js_operation) -- 在执行 js 函数期间不能被停止
        local png = splash:png() -- 执行到同步操作的时候，不能被停止
        return png
    end, 0.1)
```

(continues on next page)

[16] Splash 只能管理异步函数而无权管理这些同步函数，只能被动的等待函数执行完成

(续上页)

```
    return result
end
```

splash:send_keys

相对应的页面上下文环境发送键盘事件

原型: `splash:send_keys(keys)`

参数:

- keys：表示要作为键盘事件发送的键的字符串

返回值: nil

是否异步: 否

指定的键值将会被使用 emacs edmacro 语法中的一个小子集来进行序列化

- 空格键将会被忽略，它仅仅用来分隔不同的键值
- 字符值将会原样的保留
- 括号内的词代表对应的功能键, 像 `<Return>` 、`<Home>` 、`<Left>` 。您可以查看 QT 的帮助文档 来获取完整的功能键列表. `<Foo>` 会尝试匹配 `Qt::Key_Foo`

下面这个表格展示了一些输入宏的例子，以及他们最终被转化的结果

宏	结果
`Hello World`	`HelloWorld`
`Hello <Space> World`	`Hello World`
`< S p a c e >`	`<Space>`
`Hello <Home> <Delete>`	`ello`
`Hello <Backspace>`	`Hell`

键盘事件不会立即处理，而是要等到下一次进行消息循环。因此必须要调用*splash:wait* 来等待事件的响应执行

您也可以参考: element:send_keys , *splash:send_text* .

splash:send_text

发送一个文本，作为页面上下文的输入，字面上逐个字符输入

原型: `splash:send_text(text)`

参数:

- text: 作为输入的字符串

返回值: nil

是否异步: 否

键盘事件不会立即处理，而是要等到下一次进行消息循环。因此必须要调用*splash:wait* 来等待事件的响应执行

这个函数与*splash:send_keys* 一起涵盖了键盘输入的大部分需求，像自动填充和提交表单。

例 1：选中第一个输入框，填充并提交表单:

```
function main(splash)
    assert(splash:go(splash.args.url))
    assert(splash:wait(0.5))
    splash:send_keys("<Tab>")
    splash:send_text("zero cool")
    splash:send_keys("<Tab>")
    splash:send_text("hunter2")
    splash:send_keys("<Return>")
    -- 请注意如何使用 splash:send_keys 来改写程序
    -- splash:send_keys("<Tab> zero <Space> cool <Tab> hunter2 <Return>")
    assert(splash:wait(0))
    -- ...
end
```

例 2：使用 JavaScript 代码或者*splash:mouse_click* 来选中输入框

我们不能总是假定使用 <Tab> 会选中对应的输入框 <Enter> 会提交表单，选中输入框还可以通过点击它或者将焦点移动到它来完成。提交表单也可以通过触发表单中的提交事件或者点击提交按钮来完成.

下面这个例子将会先选中输入框，然后通过函数*splash:mouse_click* 来点击提交按钮完成表单的提交。这里假设 splash 传入两个参数:username 和 password

```
function main(splash, args)
    function focus(sel)
        splash:select(sel):focus()
    end

    assert(splash:go(args.url))
    assert(splash:wait(0.5))
    focus('input[name=username]')
    splash:send_text(args.username)
    assert(splash:wait(0))
    focus('input[name=password]')
    splash:send_text(args.password)
    splash:select('input[type=submit]'):mouse_click()
```

(continues on next page)

(续上页)

```
    assert(splash:wait(0))
    -- Usually, wait for the submit request to finish
    -- ...
end
```

您也可以参考: element:send_text , *splash:send_keys*

splash:select

利用 CSS 选择器，在 HTML DOM 中选择第一个匹配的元素

原型: `element = splash:select(selector)`

参数:

- selector: 有效的 CSS 选择器

返回值: Element 对象的值

是否异步: 否

使用*splash:select* 您可以通过 CSS 选择器来获取对应的元素，就好像在浏览器中使用 document.querySelector 一样。它返回的元素对象是 Element 。这个对象包含许多有用的属性和方法，这些属性和方法与在 JavaScript 中类似

如果使用的 CSS 选择器不能找到对应的元素，函数将会返回 `nil` 。如果您输入的 CSS 选择器无效，将会抛出一个错误.

例 1：选择类名为 `element` 的元素并且返回它所有兄弟元素的类名:

```
local treat = require('treat')

function main(splash)
    assert(splash:go(splash.args.url))
    assert(splash:wait(0.5))

    local el = splash:select('.element')
    local seen = {}
    local classNames = {}

    while el do
      local classList = el.node.classList
      if classList then
        for _, v in ipairs(classList) do
          if (not seen[v]) then
            classNames[#classNames + 1] = v
            seen[v] = true
```

(continues on next page)

(续上页)

```
          end
        end
      end

      el = el.node.nextSibling
    end

    return treat.as_array(classNames)
end
```

例 2：判断返回的元素是否存在

```
function main(splash)
    -- ...
    local el = assert(splash:select('.element'))
    -- ...
end
```

splash:select_all

在页面中通过 CSS 选择器来选择对应的 DOM 元素，并以列表的形式返回匹配的元素

原型: `elements = splash:select_all(selector)`

参数:

- selector: 有效的 CSS 选择器

返回值: 包含 Element 对象的列表

是否异步: 否

与*splash:select* 不同的是，它会在 table 中返回所有匹配到的元素的列表。

如果没有元素被匹配到，它会返回一个 `{}` ，如果传入的选择器无效，则会抛出一个错误

下面这个例子是选择所有 `<img />` 元素并获取它们的 `src` 属性

```
local treat = require('treat')

function main(splash)
    assert(splash:go(splash.args.url))
    assert(splash:wait(0.5))

    local imgs = splash:select_all('img')
    local srcs = {}
```

(continues on next page)

(续上页)

```
    for _, img in ipairs(imgs) do
      srcs[#srcs+1] = img.node.attributes.src
    end

    return treat.as_array(srcs)
end
```

1.6 响应对象

Respons Object 作为某些 Splash 方法执行的结果被返回 (像 splash:http_get 或者 splash:http_post); 它也被传递给某些回调函数 (例如 splash:on_response 和 splash:on_response_headers 回调) 这个对象包含响应的相关内容

1.6.1 response.url

响应的 URL，在发生重定向的情况下，这个值是请求中的最后一个 url

这个字段为只读的字段

1.6.2 response.status

响应的 HTTP 状态码

这个字段是一个只读字段

1.6.3 response.ok

成功获取到响应时为 `true` 否则为 `false`

例如:

```
local reply = splash:http_get("some-bad-url")
-- reply.ok == false
```

这个字段是一个只读字段

1.6.4 response.headers

一个用 Lua 中 table 结构表示的 HTTP 的响应头 (一个由名称到键值的映射); 键值为响应头对应的名称, 值为响应头对应的值.

查询时是不区分大小写的，因此 `response.headers['content-type']` 与 `response.headers['Content-Type']` 是相同的.

这个字段是一个只读字段

1.6.5 response.info

它是一个 lua 中的 table 结构，以 HAR response 的格式来表示响应数据

该字段是一个只读字段

1.6.6 response.body

原始响应信息 (为一个 二进制对象)

如果您想通过 Lua 来处理这个响应体，可以使用 treat.as_string 来将二进制数据转化为字符串

默认情况下*response.body* 属性在回调 splash:on_response 您可以使用 splash.response_body_enabled 或者 request:enable_response_body 来启用它

1.6.7 response.request

一个对应的请求对象

这个字段是一个只读字段

1.6.8 response:abort

原型: `response:abort()`

返回值: nil

是否异步: 否

放弃接收响应体, 这个函数仅仅在还没有开始下载响应体时起作用，您只能在 splash:on_response_headers 定义的回调中使用

1.7 请求对象

请求对象通过 splash:on_request 定义的回调函数接收, 它们也可以作为 response.request

1.7.1 属性

请求对象中用许多属性来描述 HTTP 请求的相关信息. 这些字段仅仅作为一种信息，即使对他们进行修改也不会对最终发送的 HTTP 请求产生任何影响

request.url

请求的 url

request.method

使用大写字母表示的请求的方法, 例如 “GET”

request.headers

一个用 Lua 中 table 结构表示的 HTTP 的请求头 (一个由名称到键值的映射); 键值为响应头对应的名称, 值为响应头对应的值.

查询时是不区分大小写的，因此 `request.headers['content-type']` 与 `request.headers['Content-Type']` 是相同的.

request.info

它是一个 lua 中的 table 结构，以 HAR request 的格式来表示请求数据

1.7.2 方法

为了修改或者丢弃某个请求，您可以使用下面列举的 `request` 相关方法。请注意这些方法只有在请求发送之前有效 (如果一个请求已经被发送，那么修改它也就没有意义)。目前您只能在 `splash:on_request <splash-on-request>` 定义的回调函数中使用

request:abort

丢弃一个请求

原型: `request:abort()`

返回值: `nil`

是否异步: 否

request:enable_response_body

启用对响应内容的追踪 (例如启用之后您可以使用 response.body)

原型: `request:enable_response_body()`

返回值: `nil`

是否异步: 否

当 splash.response_body_enabled 被设置为 false 时, 这个函数将会开启每个请求的响应内容追踪, 您可以在 `splash:on_request <splash-on-request>` 定义的回调函数中使用

request:set_url

修改请求的 URL 为用户指定的 url

原型: `request:set_url(url)`

参数:

- url: 一个请求的新的 url

返回值: `nil`

是否异步: 否

request:set_proxy

在当前的请求中设置代理服务器的相关信息

原型: `request:set_proxy{host, port, username=nil, password=nil, type='HTTP'}`

参数:

- host
- port
- username
- password
- type: 代理类型; 允许的类型有‘HTTP’和‘SOCKS5’.

返回值: nil

是否异步: 否

如果一个代理服务不需要进行认证，那么 `username` 和 `password` 可以忽略

如果代理类型设置为“HTTP” HTTPS 的代理也能正常工作, 它是使用 CONNECT 命令还实现的

request:set_timeout

您可以针对这个请求设置一个超时时间

原型: `request:set_timeout(timeout)`

参数:

- timeout：超时时间，单位为秒

返回值: nil

是否异步: 否

如果发生超时后，响应仍然没有接收完成，此时会丢弃请求。您可以参考: splash.resource_timeout

request:set_header

设置请求的请求头

原型: `request:set_header(name, value)`

参数:

- name: 请求头的名称
- value: 请求头的值

返回值: nil

是否异步: 否

您可以参考: splash:set_custom_headers

1.8 Element 对象

Element 对象是对 JavaScript DOM 对象的封装, 它们在一些函数返回任意类型的 DOM 节点时被创建 (节点、元素、HTML 元素等等)。

splash:select 和 splash:select_all 返回 Element 对象，splash:evaljs 也可能返回 Element 对象, 但是目前它们不能被放到数组或其他对象中一起返回。它们必须作为顶层节点或者节点的列表被返回。

1.8.1 方法

要修改或者检索有关元素的信息，您可以使用下列的方法

element:mouse_click

在元素上触发鼠标点击消息

原型: `ok, reason = element:mouse_click{x=nil, y=nil}`

参数:

- x: 可选值, 相对于元素左上角的 x 的坐标
- y: 可选值, 相对于元素左上角的 y 的坐标

返回值: ok, reason 元组, 当函数在执行过程中发生错误的时候 ok 的值为 nil. reason 包含了一个表示错误类型的字符串

是否异步: 是

如果没有指定 x 和 y 的值，默认为元素宽的一半和高的一半，点击事件将在元素的中间被触发。

这个坐标可以是负值，这意味着点击事件将在元素之外触发。

例 1：在靠近元素左上角的位置触发点击事件

```
function main(splash)
    -- ...
    local element = splash:select('.element')
    local bounds = element:bounds()
    assert(element:mouse_click{x=bounds.width/3, y=bounds.height/3})
    -- ...
end
```

例 2：在元素上方 10 个像素的位置触发点击事件

```
function main(splash)
    -- ...
    splash:set_viewport_full()
    local element = splash:select('.element')
    assert(element:mouse_click{y=-10})
    -- ...
end
```

与 splash:mouse_click 不同，*element:mouse_click* 会一直等待直到点击事件执行。所以如果要看点击事件执行之后在页面上产生的效果，您不需要调用 splash:wait 进行等待

如果当前元素不在视口范围之内，视口会被自动滚动到合适的位置，以便让元素可见，如果需要滚动页面，在点击事件被触发之后，页面不会自动滚回原来的位置.

您可以在 splash:mouse_click 中查阅更多与鼠标事件相关的内容

element:mouse_hover

在对应元素上触发鼠标悬停事件 (JavaScript mouseover 事件)

原型: `ok, reason = element:mouse_hover{x=0, y=0}`

参数:

- x: 可选值, 相对于元素左上角的 x 的坐标
- y: 可选值, 相对于元素左上角的 y 的坐标

返回值: ok，reason 元组，当函数在执行过程中发生错误的时候 ok 的值为 nil. reason 包含了一个表示错误类型的字符串

是否异步: 否

如果没有指定 x 和 y 的值，默认为元素宽的一半和高的一半，鼠标悬停事件将在元素的中间被触发。

这个坐标可以是负值，这意味着鼠标悬停事件将在元素之外触发。

例 1：在靠近元素左上角的位置触鼠标悬停事件

```
function main(splash)
    -- ...
    local element = splash:select('.element')
    assert(element:mouse_hover{x=0, y=0})
    -- ...
end
```

例 2：在元素上方 10 个像素的位置触发鼠标悬停事件

```
function main(splash)
    -- ...
    splash:set_viewport_full()
    local element = splash:select('.element')
    assert(element:mouse_hover{y=-10})
    -- ...
end
```

与 splash:mouse_hover 不同，*element:mouse_hover* 会一直等待直到点击事件执行。所以如果要看点击事件执行之后在页面上产生的效果，您不需要调用 splash:wait 进行等待

如果当前元素不在视口范围之内，视口会被自动滚动到合适的位置，以便让元素可见，如果需要滚动页面，在点击事件被触发之后，页面不会自动滚回原来的位置.

您可以在 splash:mouse_hover 中查阅更多与鼠标事件相关的内容

element:styles

获取元素的所有样式信息

原型: `styles = element:styles()`

返回值: style 是一个描述获取到的样式的 table 结构

是否异步: 否

这个方法返回在元素上调用 window.getComputedStyle() 获取到的结果

例子: 获取元素所有的样式，并返回 font-size 属性

```
function main(splash)
    -- ...
    local element = splash:select('.element')
    return element:styles()['font-size']
end
```

element:bounds

返回元素 bounding 框的矩形

原型: `bounds = element:bounds()`

返回值: `bounds` 是一个 table 结构，里面包含了 bounding 框的 `top` , `right` , `bottom` 和 `left` . 同时它也包含了 bounding 框的 `width` 和 `height` 值

是否异步: 否

例子: 获取元素的边界

```
function main(splash)
    -- ..
    local element = splash:select('.element')
    return element:bounds()
    -- e.g. bounds is { top = 10, right = 20, bottom = 20, left = 10, height = 10, width = 10 }
end
```

element:png

以 png 格式返回元素的截图

原型: `shot = element:png{width=nil, scale_method='raster', pad=0}`

参数:

- width: 可选值, 截图的宽，以像素为单位
- scale_method: 可选值，缩放图片的方式, 可选的值有 `'raster'` 和 `'vector'`
- pad: 可选值, 整型或者是 `{left, top, right, bottom}` 结构的 padding 值

返回值: 以 二进制对象 表示的 PNG 截图的数据, 如果结果为空 (例如元素在 DOM 中不存在, 或者不可见) 会返回 nil

是否异步: 否

pad 参数设置返回图片的 padding 值, 如果是一个整数值，那么 padding 中所有的边距都相同, 如果值为正数，那么截图会在原来元素的基础上扩充指定像素大小, 如果未负值，截图会在原来元素的基础上压缩指定像素大小.

例子: 返回一个填充指定大小的元素截图

```
function main(splash)
    -- ..
    local element = splash:select('.element')
    return element:png{pad=10}
end
```

如果元素不在视口中, 视口会暂时滚动以便让元素可见，然后再滚回到原来的位置

您可以参阅 splash:png

element:jpeg

以 jpeg 格式返回元素的截图

原型: `shot = element:jpeg{width=nil, scale_method='raster', quality=75, region=nil, pad=0}`

参数:

- width: 可选值, 截图的宽，以像素为单位
- scale_method: 可选值，缩放图片的方式, 可选的值有 `'raster'` 和 `'vector'`
- quality: 可选值, 生成 jpeg 图片的质量, 为 `0` 到 `100` 的整数值
- pad: 可选值, 整型或者是 `{left, top, right, bottom}` 结构的 padding 值

返回值: 以 二进制对象 表示的 jpeg 截图的数据, 如果结果为空 (例如元素在 DOM 中不存在, 或者不可见) 会返回 nil

是否异步: 否

pad 参数设置返回图片的 padding 值, 如果是一个整数值，那么 padding 中所有的边距都相同, 如果值为正数，那么截图会在原来元素的基础上扩充指定像素大小, 如果未负值，截图会在原来元素的基础上压缩指定像素大小.

如果元素不在视口中, 视口会暂时滚动以便让元素可见，然后再滚回到原来的位置

您可以参阅 splash:jpeg

element:visible

判断元素是否可见

原型: `visible = element:visible()`

返回值: `visible` 表示元素是否可见

是否异步: 否

element:focused

检测元素是否获取到焦点

原型: `focused = element:focused()`

返回值: `focused` 表示元素是否获取到焦点

是否异步: 否

element:text

从元素中获取文本信息

原型: `text = element:text()`

返回值: `text` 表示元素的文本内容

是否异步: 否

它会尝试返回以下 JavaScript 节点的属性值:

- textContent
- innerText
- value

如果这些值都为空，则会返回空值

element-info

获取元素的有用信息

原型: `info = element:info()`

返回值: `info` 表示元素的相关信息

是否异步: 否

info 是一个包含以下字段的 table 结构

- nodeName - 以小写字母表示的节点的名称 (比如 h1)
- attributes - 以属性名和属性值为键值对的 table 结构
- tag - 表示该元素的 HTML 的字符串
- html - 元素内部的 HTML 代码
- text - 元素内部的文本值
- x - 元素的 x 坐标
- y - 元素的 y 坐标

- width - 元素的宽
- height - 元素的高
- visible - 元素是否可见的一个标志

element:field_value

获取元素字段的值 (针对 input, select, textarea, button)

原型: `ok, value = element:field_value()`

返回值: `ok, value` 的元组, 如果 ok 为 `nil` 表明在执行函数的过程中有错误发生, 此时 `value` 是一个包含错误类型信息的字符串。如果没有错误发生, `ok` 为 `true` `value` 为对应元素字段的值

是否异步: 否

这些方法按照以下方式工作:

- **如果当前元素是 `select` :**
  - 如果元素的 `multiple` 属性为 `true` 它会以 table 结构返回所有被选中的值
  - 否则返回被选中的一个值
- **如果元素存在属性 `type="radio"` :**
  - 如果它被选中，则返回选中的值
  - 否则返回 `nil`
- 如果元素存在属性 `type="checkbox"` 则返回 bool 值
- 否则会返回元素 `value` 属性的值，或者当 `value` 属性不存在的时候返回空字符串

element:form_values

如果元素类型为表单，它会以 table 结构的方式返回表单中每项的值

原型: `form_values, reason = element:form_values{values='auto'}`

参数:

- values: 返回值的类型，可以是 `'auto'` 、`'list'` 或者 `'first'` 中的一个

返回值: `form_values, reason` 元组，如果 `form_values` 为 nil，则表明当前执行函数时发生错误，或者当前元素不是表单类型。`reason` 返回错误类型信息, `form_values` 返回以表单元素名为键值，元素值为键值的 table 结构

是否异步: 否

函数的返回值取决于参数 `values` 的值，它的取值有下列三种

`'auto'`

返回值将是 table 或者是一个单一的值，这将取决于表单的类型

- 如果元素类型为 `<select multiple>` 返回值将会是一个包含所有选中项的 table 结构, 如果属性值不存在则返回一个文本内容
- 如果表单中多个元素都有相同的 `name` 属性, 则返回一个包含该元素所有值的 table 结构
- 否则将会是一个字符串 (针对 text 框或者单选按钮框) 或者是一个 bool 类型 (多选框), 或者返回 `nil`

如果您需要在 lua 中使用这个返回值，这种返回值类型将会很方便

`'list'`

返回值将一直是 table 结构 (针对 lists), 即使在表单元素是一个单值的情况下。这一特性也可用于具有未知结构的表

- 如果元素是一个 checkbox ，并且有 value 属性，那么 table 中将使用这个属性作为键值
- 如果元素是一个 `<select multiple>` ，并且有一些还具有相同的名称, 那么他们的值将于之前的值连接到一起
- 在编写通用的表单处理代码时这个类型的返回值将会十分有用, 与 `'auto'` 不同，您不需要考虑多种数据类型

`'first'` 返回的值都是单一值, 即使表单元素可以选择多个值, 如果表单元素可以选择多个值，它总会返回被选中的第一个值。

例 1：返回下列登录表单的值

```
<form id="login">
    <input type="text" name="username" value="admin" />
    <input type="password" name="password" value="pass" />
    <input type="checkbox" name="remember" value="yes" checked />
</form>
```

```
function main(splash)
    -- ...
    local form = splash:select('#login')
    return assert(form:form_values())
end

-- returned values are
{ username = 'admin', password = 'pass', remember = true }
```

例 2：当 values 的值为 `'list'`

```
function main(splash)
    -- ...
```

(continues on next page)

(续上页)

```
    local form = splash:select('#login')
    return assert(form:form_values{values='list'}))
end

-- returned values are
{ username = ['admin'], password = ['pass'], remember = ['checked'] }
```

例 3: 当 values 的值为 'first' 的时候返回下列表单的值

```
<form>
    <input type="text" name="foo[]" value="coffee"/>
    <input type="text" name="foo[]" value="milk"/>
    <input type="text" name="foo[]" value="eggs"/>
    <input type="text" name="baz" value="foo"/>
    <input type="radio" name="choice" value="yes"/>
    <input type="radio" name="choice" value="no" checked/>
    <input type="checkbox" name="check" checked/>

    <select multiple name="selection">
        <option value="1" selected>1</option>
        <option value="2">2</option>
        <option value="3" selected>2</option>
    </select>
</form>
```

```
function main(splash)
    -- ...
    local form = splash:select('form')
    return assert(form:form_values(false))
end

-- returned values are
{
    ['foo[]'] = 'coffee',
    baz = 'foo',
    choice = 'no',
    check = false,
    selection = '1'
}
```

element:fill

利用提供的值来填充表单

原型: `ok, reason = element:fill(values)`

参数:

- values: 以表单项名称为键值，表单项的值作为键值的 table 结构

返回值: `ok, reason` 元组, 如果 `ok` 为 nil 表明在函数执行过程中有错误发生, `reason` 返回错误类型的字符串信息

是否异步: 否

为了填充表单，您的输入框需要有 `name` 属性, 而且需要事先选中表单

例 1：获取表单的值，并修改 password 项

```
function main(splash)
    -- ...
    local form = splash:select('#login')
    local values = assert(form:form_values())
    values.password = "l33t"
    assert(form:fill(values))
end
```

例 2：填充复杂的表单:

```
<form id="signup" action="/signup">
    <input type="text" name="name"/>
    <input type="radio" name="gender" value="male"/>
    <input type="radio" name="gender" value="female"/>

    <select multiple name="hobbies">
        <option value="sport">Sport</option>
        <option value="cars">Cars</option>
        <option value="games">Video Games</option>
    </select>

    <button type="submit">Sign Up</button>
</form>
```

```
function main(splash)
  assert(splash:go(splash.args.url))
  assert(splash:wait(0.1))

  local form = splash:select('#signup')
  local values = {
    name = 'user',
    gender = 'female',
```

(continues on next page)

(续上页)

```
    hobbies = {'sport', 'games'},
  }

  assert(form:fill(values))
  assert(form:submit())
  -- ...
end
```

element:send_keys

向元素发送键盘消息

原型: `ok, reason = element:send_keys(keys)`

参数:

- keys: 要作为键盘事件发送的多个字符组成的字符串

返回值: `ok, reason` 元组, 如果 `ok` 为 nil 则表明在调用函数期间发生错误, `reason` 提供错误类型信息

是否异步: 否

这个方法主要进行这样几个操作

- 点击元素
- 向元素发送对应的键盘事件

更多信息，您可以参阅: splash:send_keys

element:send_text

向元素发送键盘消息

原型: `ok, reason = element:send_text(text)`

参数:

- text: 将要被发送并作为元素输入的字符串

返回值: `ok, reason` 元组, 如果 `ok` 为 nil 则表明在调用函数期间发生错误, `reason` 提供错误类型信息

是否异步: 否

这个方法主要进行这样几个操作

- 点击元素
- 向元素发送指定的字符串, 并作为元素的输入值

更多信息，您可以参阅: splash:send_text

element:submit

提交表单

原型: `ok, reason = element:submit()`

返回值: `ok, reason` 元组, 如果 `ok` 为 nil 则表明在调用函数期间发生错误 (例如您尝试提交的元素不是表单), `reason` 提供错误类型信息

是否异步: 否

例：先获取表单的值，然后设置对应的值，最后提交

```
<form id="login" action="/login">
    <input type="text" name="username" />
    <input type="password" name="password" />
    <input type="checkbox" name="remember" />
    <button type="submit">Submit</button>
</form>
```

```
function main(splash)
    -- ...
    local form = splash:select('#login')
    assert(form:fill({ username='admin', password='pass', remember=true }))
    assert(form:submit())
    -- ...
end
```

element:exists

在 DOM 中检查对应元素是否存在。如果不存在，某些方法可能会执行失败，并返回对应的标志

原型: `exists = element:exists()`

返回值: `exists` 表明当前元素是否存在

是否异步: 否

注解: 不要使用 `splash:select(..):exists()` 来检查元素是否存在, 如果选择器没有选择任何值那么 splash:select 会返回 `nil` , 此时应该检查返回值是否为 `nil`

`element:exists()` 应该用于这样的情况：您之前确定有这么一个元素，但是不清楚后续它是否被从当前的 DOM 中移出。

下面将列举几种会将元素移出 DOM 的原因，其中一个理由是它可能被某些 JavaScript 代码给移除了

例 1：元素被 js 代码移除

```
function main(splash)
    -- ...
    local element = splash:select('.element')
    assert(splash:runjs('document.write("<body></body>")'))
    assert(splash:wait(0.1))
    local exists = element:exists() -- exists will be `false`
    -- ...
end
```

另一个原因是元素是通过 JavaScript 代码创建的但是并没有加入到 DOM 中

例 2：元素未被插入到 DOM 中

```
function main(splash)
    -- ...
    local element = splash:select('.element')
    local cloned = element.node:cloneNode() -- the cloned element isn't in DOM
    local exists = cloned:exists() -- exists will be `false`
    -- ...
end
```

1.8.2 DOM 方法

除了特定的 Splash 自定义的方法，DOM 元素也支持许多常见的 DOM HTMLElement 类的方法

用法

您只需要在 `element` 对象上调用，例如为了确定某个 DOM 元素是否具有某个特定的属性，您可以调用 hasAttribute

```
function main(splash)
    -- ...
    if splash:select('.element'):hasAttribute('foo') then
        -- ...
    end
    -- ...
end
```

另一个例子：为了确保元素在视口中，您可以使用 `scrollIntoViewIfNeeded` 方法

```
function main(splash)
    -- ...
    splash:select('.element'):scrollIntoViewIfNeeded()
```

(continues on next page)

(续上页)

```
    -- ...
end
```

被支持的 DOM 方法

继承自 EventTarget 的方法:

- addEventListener
- removeEventListener

继承自 HTMLElement 的方法

- blur
- click
- focus

继承自 Element 的方法:

- getAttribute
- getAttributeNS
- getBoundingClientRect
- getClientRects
- getElementsByClassName
- getElementsByTagName
- getElementsByTagNameNS
- hasAttribute
- hasAttributeNS
- hasAttributes
- querySelector
- querySelectorAll
- releasePointerCapture
- remove
- removeAttribute
- removeAttributeNS
- requestFullscreen

- requestPointerLock
- scrollIntoView
- scrollIntoViewIfNeeded
- setAttribute
- setAttributeNS
- setPointerCapture

继承自 Node 的方法

- appendChild
- cloneNode
- compareDocumentPosition
- contains
- hasChildNodes
- insertBefore
- isDefaultNamespace
- isEqualNode
- isSameNode
- lookupPrefix
- lookupNamespaceURI
- normalize
- removeChild
- replaceChild

这些方法应该作为 JS 的对应物, 在 Lua 中使用

例如, 您可以通过 `element:addEventListener(event, listener)` 方法来为元素添加对应事件的响应程序

```
function main(splash)
    -- ...
    local element = splash:select('.element')
    local x, y = 0, 0

    local store_coordinates = function(event)
        x = event.clientX
        y = event.clientY
    end
```

(continues on next page)

(续上页)

```
    element:addEventListener('click', store_coordinates)
    assert(splash:wait(10))
    return x, y
end
```

1.8.3 属性

element.node

element.node 可以对外暴露 DOM 元素所有可公开的属性和方法，但是不包括 Splash 自定义的属性和方法。如果您想要更准确的使用它，请使用只读的方式。在将来他可能会允许避免可能出现的命名冲突

例如，您需要获取元素的 innerHTML 值，您可以使用 .node.innerHTML

```
function main(splash)
    -- ...
    return {html=splash:select('.element').node.innerHTML}
end
```

element.inner_id

元素内部表示的 ID。它对于相等的元素实例进行比较可能很有用

例如:

```
function main(splash)
    -- ...
    local same = element2.inner_id == element2.inner_id
    -- ...
end
```

1.8.4 DOM 属性

用法

Element 对象也提供了对几乎所有 DOM 元素属性的支持，例如您可以获取 DOM 元素的节点名称 (p, div, a 等等)

```
function main(splash)
    -- ...
```

(continues on next page)

(续上页)

```
    local tag_name = splash:select('.foo').nodeName
    -- ...
end
```

大部分元素属性不光是可读，它们也是可写的，例如您可以设置元素 innerHTML 属性的值

```
function main(splash)
    -- ...
    splash:select('.foo').innerHTML = "hello"
    -- ...
end
```

被支持的 DOM 属性

下面将列举被支持的 DOM 属性 (某些是可读写的，某些是只读的)

继承自 HTMLElement 的属性

- accessKey
- accessKeyLabel (只读)
- contentEditable
- isContentEditable (只读)
- dataset (只读)
- dir
- draggable
- hidden
- lang
- offsetHeight (只读)
- offsetLeft (只读)
- offsetParent (只读)
- offsetTop (只读)
- spellcheck
- style - 用一个 table 表示，table 的值可以被修改
- tabIndex
- title
- translate

继承自 Element 的属性

- attributes (只读) - 一个 table 结构表示的元素的属性
- classList (只读) - 一个 table 结构表示的元素类名
- className
- clientHeight (只读)
- clientLeft (只读)
- clientTop (只读)
- clientWidth (只读)
- id
- innerHTML
- localeName (只读)
- namespaceURI (只读)
- nextElementSibling (只读)
- outerHTML
- prefix (只读)
- previousElementSibling (只读)
- scrollHeight (只读)
- scrollLeft
- scrollTop
- scrollWidth (只读)
- tabStop
- tagName (只读)

继承自 Node 的属性

- baseURI (只读)
- childNodes (只读)
- firstChild (只读)
- lastChild (只读)
- nextSibling (只读)
- nodeName (只读)
- nodeType (只读)

- nodeValue
- ownerDocument (只读)
- parentNode (只读)
- parentElement (只读)
- previousSibling (只读)
- rootNode (只读)
- textContent

当然您也可以通过属性在特殊的事件上添加指定的事件处理函数. 当这个处理函数被调用时，他将会接收到一个 `event` 的 table 结构, 这个结构包含所有可用的方法和属性

```
function main(splash)
    -- ...
    local element = splash:select('.element')

    local x, y = 0, 0

    element.onclick = function(event)
        event:preventDefault()
        x = event.clientX
        y = event.clientY
    end

    assert(splash:wait(10))

    return x, y
end
```

如果您希望在一个事件上添加多个事件处理程序，请使用 `element:addEventListener()`

1.9 使用二进制数据

1.9.1 动机

Splash 假设脚本中所有字符串被以 UTF-8 格式进行编码。即使 HTTP 响应信息不是通过 UTF-8 编码的，它对其使用 UTF-8 的方式还原为字符串。由于内部的浏览器使用 UTF-8，所以通过 splash:html 返回的 HTML 代码也是 UTF-8 的

如果您在 `main` 函数中通过 table 返回一组数据,Splash 会将它编码为一个 json 数据。json 是一个无法处理二进制的文本协议。所以在通过 json 返回数据时，Splash 认为里面所有的字符串都是 UTF-8 格式的

但是在某些场合下使用二进制数据十分重要, 例如这个二进制数据可以是通过 splash:png 方法获取的图片的原始数据, 或者是通过 splash:http_get 获取到的未经 UTF-8 编码的原始的 HTTP 响应信息

1.9.2 二进制对象

为了将非 UTF-8 数据传递给 Splash(在 main 函数中返回数据, 或者向 Splash 方法传递参数), 在脚本中可以使用 treat.as_binary 来将数据作为二进制对象进行标记

一些函数本身就返回二进制对象, 像: *splash:png <./scripting-ref.html#splash-png>* , splash:jpeg ; response.body

二进制对象可以直接通过 main 函数返回。这也是为什么下面的例子可以运行 (render.png 的基础例子)

```
-- 模拟基础的 render.png 的实现
function main(splash)
    assert(splash:go(splash.args.url))
    return splash:png()
end
```

所有的二进制对象都添加了 Content-Type 字段, 例如 splash:png 会拥有的 Content-Type 值为 image/png

如果直接返回, 二进制对象会被作为响应体的值来使用, HTTP 响应体中 Content-Type 的值会被设置成二进制对象所代表的数据类型。所以在上面的例子中响应的 Content-Type 会被设置成 PNG 图片的类型, 也就是 image/png

如果要构造您自己的二进制对象, 请使用 treat.as_binary 函数。例如让我们将一个 1x1 像素大小的黑色 GIF 图片作为响应内容

```
treat = require("treat")
base64 = require("base64")

function main(splash)
    local gif_b64 = "AQABAIAAAAAAAAAAACH5BAAAAAAALAAAAAABAAEAAAICTAEAOw=="
    local gif_bytes = base64.decode(gif_b64)
    return treat.as_binary(gif_bytes, "image/gif")
end
```

当 main 函数返回时, 二进制对象自身的 Content-Type 值优于通过 splash:set_result_content_type 设置的值。为了覆盖二进制对象自身的 Content-Type 值, 您可以新建一个二进制对象并重新指定 Content-Type 值

```
lcoal treat = require("treat")
function main(splash)
    -- ...
    local img = splash:png()
    return treat.as_binary(img, "image/x-png") -- default was "image/png"
end
```

当一个二进制值需要被序列化为 json 对象时, 在序列化之前会自动将二进制值采用 base64 进行编码，例如当需要在 `main` 函数中返回一个 table

```
function main(splash)
    assert(splash:go(splash.args.url))

    -- result is a JSON object {"png": "...base64-encoded image data"}
    return {png=splash:png()}
end
```

1.10 可使用的 lua 库

当 Splash 的 沙盒模式 关闭时，可以支持所有的 lua 标准库，而当沙盒模式开启 (默认开启) 时，只有部分标准库可以调用。您可以参阅*Standard Library* 以获取更多信息

splash 默认提供多个非标准模块

- *json*：对 json 数据进行编码/解码
- *base64*：对数据进行 Base64 编码/解码
- *treat*：将 lua 变量进行相应的类型转化，以便适应 Splash[2]

与标准模块不同，扩展模块在使用前需要引用, 例如

```
base64 = require("base64")
function main(splash)
    return base64.encode('hello')
end
```

您可以参阅自定义 *lua* 模块 的内容，来为 Splash 添加更多 lua 模块

1.10.1 lua 标准库

当 splash 的沙盒模式开启时 (默认),splash 支持 lua 5.2 版本中的下列标准库:

- string
- table
- math
- os

上述库是预先导入的，您不需要显示的调用 `require`

[2] 这段话的原文是 “fine-tune the way Splash works with your Lua varaibles and returns the result” 这里为了理解方便采用意译

注解: 目前当沙盒模式开启时，上述库中并不是的所有方法都可以使用。

1.10.2 json

json 库可以将一个数据序列化为 json 数据，或者将 json 数据反序列化为 lua 中的 table 结构。库中提供了两个方法*json.encode* 和*json.decode*

json.encode

将数据编码为 JSON 格式

原型: `result = json.encode(obj)`

参数:

- obj: 将被转化为 json 格式的对象

返回值: 转化完成后的一个 json 字符串

目前不支持针对二进制数据的 json 格式化, json.encode 函数在处理二进制对象的时候，先将其使用 Base64 方式的加密

json.decode

将 json 对象反序列化为普通对象

原型: `decoded = json.decode(s)`

参数:

- s: 一个 json 字符串

返回值: 反序列化完成之后产生的 Lua 对象

示例

```
json = require("json")

function main(splash)
    local resp = splash:http_get("http:/myapi.example.com/resource.json")
    local decoded = json.decode(resp.content.text)
    return {myfield=decoded.myfield}
end
```

请注意，不同于*json.encode* 函数, *json.decode* 函数没有针对 二进制对象 的支持。这意味着如果您希望将之前使用*json.encode* 得到的 json 数据还原成最初的二进制对象，您需要自己使用 base64 的方式进行解码，您可以通过使用 lua 中的*base64* 模块来完成这个操作

1.10.3 base64

使用 base64 方式对字符串进行编码/解码, 它提供了两个函数*base64.encode* 和*base64.decode* 。如果您需要在 json 请求或者响应中传递一些二进制数据，那么这些函数将会很有用

base64.encode

将一个字符串使用 base64 编码

原型: `encoded = base64.encode(s)`

参数:

- s: 需要进行编码的字符串或者 二进制对象

返回值: `s` 通过 base64 编码后的字符串

base64.decode

将一个字符串通过 base64 解码

原型: `data = base64.decode(s)`

参数:

- s: 需要进行解码的字符串

返回值: 解码后的 lua 字符串

请注意，base64.decode 可能会返回一个非 UTF-8 格式的字符串, 这个字符串在传入 splash 时可能不太安全 (作为 `main` 的返回值或者传入其他 splash 函数)。如果您知道原始数据是 UTF-8 格式或者 ASCII 格式会非常好，但是如果您不知道原始数据的格式或者原始数据根本就不是 UTF-8 格式的，您可以在*base64.encoded* 函数的返回结果上调用*treat.as_binary*

例子: 返回一个黑色的 1x1 像素大小的 gif 图片

```
treat = require("treat")
base64 = require("base64")

function main(splash)
    local gif_b64 = "AQABAIAAAAAAAAAAACH5BAAAAAAALAAAAAABAAEAAAICTAEAOw=="
    local gif_bytes = base64.decode(gif_b64)
    return treat.as_binary(gif_bytes, "image/gif")
end
```

1.10.4 treat

treat.as_binary

将字符串转化为 二进制对象

原型: `bytes = treat.as_binary(s, content_type="application/octet-stream")`

参数:

- s: 字符串
- content_type: `s` 的 content_type 值

返回值: 一个 二进制对象

treat.as_binary 返回一个二进制对象，这个对象不能再在 lua 中做进一步处理，但是它可以作为 main 函数的返回值。

treat.as_string

从 二进制对象 的原始数据中得到一个字符串

原型: `s, content_type = treat.as_string(bytes)`

参数:

- bytes: 一个原始的 二进制对象

返回值: `(s, content_type)` 元组, 转化后得到的字符串以及它对应的 content_type 值

treat.as_string 会 “解开” 一个 二进制对象 并得到一个普通的 lua 字符串，这个字符串可以被 lua 程序进一步处理。如果返回的字符串没有使用 UTF-8 进行解码，那么它仍然可以被 lua 做进一步的处理, 但是如果将其作为 `main` 函数的返回值或者作为其他 splash 函数的参数, 将会很不安全, 如果您想在 Splash 中将其还原成二进制对象，请使用函数*treat.as_binary*

treat.as_array

将 lua 中的 table 标记为数组 (针对编码过后的 json 数据或者 table 到 js 的转换)

原型: `tbl = treat.as_array(tbl)`

参数:

- tbl: lua table

返回值: 与参数相同的 table

json 可以表示数组和对象, 但是在 lua 中没法对他们进行区分, 不管是键值对还是数组在 lua 中都是以 table 的格式存储。

默认情况下, 从 Splash `main` 函数中返回结果，或者使用*json.encode* 函数和 splash.jsfunc 函数时，Lua table 将会转化为 json

```
function main(splash)
    -- client gets {"foo": "bar"} JSON object
    return {foo="bar"}
end
```

使用类似于数组的 lua table 将会带来意想不到的结果

```
function main(splash)
    -- client gets {"1": "foo", "2": "bar"} JSON object
    return {"foo", "bar"}
end
```

treat.as_array 将 table 标记为 Json 数组。

```
treat = require("treat")

function main(splash)
    local tbl = {"foo", "bar"}
    treat.as_array(tbl)

    -- client gets ["foo", "bar"] JSON object
    return tbl
end
```

此函数在其中修改其参数, 但是为了方便，它仍然返回原始的 table 对象, 它允许我们简化代码

```
treat = require("treat")
function main(splash)
    -- client gets ["foo", "bar"] JSON object
    return treat.as_array({"foo", "bar"})
end
```

注解: 针对 table，每有什么方法可以检测它具体的类型, 因为 `{}` 符号本身可以表示多种含义, 它既可以作为 json 数组，也可以作为 json 对象。当 table 的具体类型不确定的时候就容易产生一个错误的输出: 即使某些数据总是一个数组，当数组为空时，它可能会被当做一个对象, 为了避免这种错误，Splash 提供了一个工具函数*treat.as_array*

1.10.5 添加您自己的模块

Splash 提供了一个通过 HTTP API 等方式在脚本中使用自定义 Lua 模块的功能 (存储在服务端)。这个功能允许:

1. 重用代码，而不是通过网络一遍又一遍的发送
2. 使用第三方 lua 模块
3. 实现需要不安全代码的功能，并能够在沙盒中安全的使用它

注解: 您可以访问 http://lua-users.org/wiki/ModulesTutorial 来学习关于 lua 模块的知识, 请爱上最新的编写模块的方法，因为它会让您的模块在沙盒模式下更好的工作, 这里有一个很好的 lua 模块样式指南: http://hisham.hm/2014/01/02/how-to-write-lua-modules-in-a-post-module-world/

配置

请您进行如下配置，以便使用 lua 的自定义模块

1. 配置 lua 模块的路径，让后在对应目录下添加您自己的模块
2. 告诉 Splash，在沙盒中可以使用哪些自定义模块
3. 在 lua 脚本中使用 `require` 来导入一个库

您可以在启动 Splash 的时候加上参数 `--lua-package-path` 来指定 lua 库的路径。`--lua-package-path` 的值应该是以分号分隔的多个路径的字符串，每一个路径应该有一个 `?` 它被用来替代模块名称

```
$ python3 -m splash.server --lua-package-path "/etc/splash/lua_modules/?.lua;/home/myuser/splash-
↪modules/?.lua"
```

注解: 如果您使用 docker 来安装 splash，请参考 文件共享 这部分内容来获取与设置文件路径相关的内容

注解: `--lua-package-path` 中的值会被添加到 Lua 的 `package.path`

当您使用 Lua 沙盒的时候 (默认情况下开启) . 在脚本中 `require` 会收到一定的限制, 它只能加载在白名单中的模块。白名单在默认情况下是空的，也就是说在默认情况下您不能加载任何东西, 为了让您的模块可用，您可以在启动 Splash 的时候设置 `--lua-sandbox-allowed-modules` 选项。它应该包含以分号为分隔符的字符串，其中每个子串代表允许被加载的模块的名称

```
$ python3 -m splash.server --lua-sandbox-allowed-modules "foo;bar" --lua-package-path "/etc/splash/
↪lua_modules/?.lua"
```

设置完成之后，您就可以通过 require 来加载您的自定义模块

```
local foo = require("foo")
function main(splash)
    return {result=foo.myfunc()}
end
```

模块的编写

一个基础的模块看起来像这样:

```
-- mymodule.lua
local mymodule = {}

function mymodule.hello(name)
    return "Hello, " .. name
end

return mymodule
```

在脚本中可以这样来使用

```
local mymodule = require("mymodule")

function main(splash)
    return mymodule.hello("world!")
end
```

在许多情况下，模块可能会使用 splash 对象, 这里给出一些办法来编写这类模块。最简单的办法就是让函数允许传入 splash 对象作为参数。

```
-- utils.lua
local utils = {}

-- wait until `condition` function returns true
function utils.wait_for(splash, condition)
    while not condition() do
        splash:wait(0.05)
    end
end

return utils
```

用法:

```
local utils = require("utils")

function main(splash)
    splash:go(splash.args.url)

    -- wait until <h1> element is loaded
    utils.wait_for(splash, function()
       return splash:evaljs("document.querySelector('h1') != null")
    end)

    return splash:html()
end
```

另一个编写此类模块的方法是，往一个 splash 对象中添加对应的方法。您可以通过向 Splash 类来添加方法。这种方法在 Ruby 中被称之为“打开类”，而在 python 中被叫做补丁。

```
-- wait_for.lua

-- Sandbox is not enforced in custom modules, so we can import
-- internal Splash class and change it - add a method.
local Splash = require("splash")

function Splash:wait_for(condition)
    while not condition() do
        self:wait(0.05)
    end
end

-- no need to return anything
```

用法:

```
require("wait_for")

function main(splash)
    splash:go(splash.args.url)

    -- wait until <h1> element is loaded
    splash:wait_for(function()
       return splash:evaljs("document.querySelector('h1') != null")
    end)

    return splash:html()
end
```

具体使用哪种风格，取决于您的个人喜好，函数的方式更加明确，更加容易组合。使用补丁的方式可以使代码更加紧凑，无论您使用哪种办法来向 splash 对象中添加函数，都可以直接使用 require

如前面的例子所示。标准的 lua 模块和函数的沙盒限制不适用于自定义的 Lua 模块。换句话说自定义的 lua 模块不受限制，您可以发挥出 lua 的全部功效，这将令我们可以使用第三发模块，利用 lua 来写出更加高级的功能。但是在使用的时候需要特别小心。例如让我们来使用 os 模块

```
-- evil.lua
local os = require("os")
local evil = {}

function evil.sleep()
    -- Don't do this! It blocks the event loop and has a startup cost.
    -- splash:wait is there for a reason.
    os.execute("sleep 2")
end

function evil.touch(filename)
    -- another bad idea
    os.execute("touch " .. filename)
end

-- todo: rm -rf /

return evil
```

1.11 Splash 和 Jupyter

Splash 为 Lua 提供了一个自定义的 Jupyter 内核 (以前被称为 IPython)。与 Jupyter 编辑器 前端一起，它为 Splash JavaScript 构建了一个基于 Web 的开发环境，具有语法高亮，智能代码自动补全，上下文感知帮助，内联图像支持，以及启用 Web 检查器来查看实时的 WebKit 浏览器窗口。它可以通过 Jupyter 编辑器来进行控制

1.11.1 安装

在 docker 中您可以执行下列命名来安装 Splash-Jupyter

```
$ docker pull scrapinghub/splash-jupyter
```

然后启动容器

```
$ docker run -p 8888:8888 -it scrapinghub/splash-jupyter
```

注解: 请注意，如果不加 `-it` 标志，您将不能通过 CTRL+C 来停止容器

上面的命令应该会打印出类似下面的内容

```
Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
    http://localhost:8888/?token=e2435ae336d22b23d5e868d03ce728bc33e73b6159e391ba
```

想要查看 Jupyter，请在浏览器中打开上面给出的地址，它应该会显示 Jupyter 编辑器的概述页面

注解: 在较旧的 Docker 中 (例如在 OS X 系统中使用 boot2docker 的版本), 您可能需要将 localhost 替换成 docker 能够使用的 IP，比如在 boot2docker 中使用 `boot2docker ip` 得到的 IP 地址，或者在进入 docker-machine 时在屏幕上出现的 `docker-machine ip <your machine>` 字样中的 IP[3]

点击 "New" 这个按钮，然后在下拉框中选择 Splash，这样 Splash 编译器应该就会被打开 Splash 编辑器看起来就像是 IPython 编辑器或者是以其他基于 Jupyter 的编辑环境，它允许在内部运行和开发 Splash 的 lua 脚本。例如您可以在里面输入 `splash:go("you-favorite-website")`，然后运行，接着您在另一个单元格中输入 `splash:png()`。也像之前那样允许他，这时您应该可以得到一个内联显示的网站截图

1.11.2 保持 Jupyter 环境

默认情况下 Jupyter 编辑环境存储在 Docker 容器中，一旦重启镜像，环境就会被删除。为了保存当前编辑器环境，您可以挂载一个本地的文件到 `/notebooks` 中，例如让我们使用当前目录来保存编辑器环境

```
$ docker run -v `/bin/pwd`/notebooks:/notebooks -p 8888:8888 -it splash-jupyter
```

1.11.3 实时的 WebKit 窗口

当 Splash-Jupyter 在 Docker 中执行的时候您可以使用 Web 检查器来查看实时的 WebKit 窗口。您需要在启动 Jupyter 时向 Docker 传递额外的参数来向 Docker 容器共享主机系统的 X 服务。您需要在启动的时候使用 `--disable-xvfb` 标志

```
$ docker run -e DISPLAY=unix$DISPLAY \
             -v /tmp/.X11-unix:/tmp/.X11-unix \
             -v $XAUTHORITY:$XAUTHORITY \
             -e XAUTHORITY=$XAUTHORITY \
             -p 8888:8888 \
             -it scrapinghub/splash-jupyter --disable-xvfb
```

[3] 在旧版的 Docker 中，启动时会出现 "docker is configured to use the default machine with IP xxx.xxx.xxx.xxx" 的字样，我在使用 DockerToolbox 的时候一般这个 IP 都是 192.168.99.100

1.11.4 从编辑器到 HTTP API

当您能够在 Splash 编辑器中进行程序开发之后，您可能会希望将其转化为合适的可以提交到 HTTP API 的脚本代码 (参见 execute 和 run)

要达到这样的效果，您可以复制-粘贴所有对应的代码 (或者直接下载，通过以此点击 “File -> Download as -> .lua”)。针对 run 这个端点，使用 `return` 语句来返回结果

```
-- Script code goes here,
-- including all helper functions.
return {...}  -- return the result
```

针对 execute 端点，需要添加 `return` 语句，并将代码放入到 `main` 函数中

```
function main(splash)
    -- Script code goes here,
    -- including all helper functions.
    return {...}  -- return the result
end
```

为了使代码更加通用，您可以将其中硬编码的常量通过 *splash.args <./scripting-ref.html#splash-args>* 来进行替换。另外如果您需要处理多个不同的页面，您可以考虑使用不同的参数提交多个请求，而不是在脚本中运行循环。这是一种简单的优化操作.

这里也有一些陷阱

- 当您在编译器单元中运行一段代码，紧接着运行下一个单元中的代码时，在不同单元中代码的执行是有延迟的，这个延迟类似于在每个单元之间调用了 splash:wait 。[4]
- 在编辑器中是不考虑沙盒的设置的，因为在 Jupyter 编辑器中是没有沙盒的，一般情况下这不是什么问题，但是请注意，有些函数在沙盒下可能并不被支持

1.12 问答

1.12.1 如何向 Splash HTTP API 发送请求

我们推荐的方式是使用 `application/json` 类型的 POST 请求。因为这种方式可以保留原始的数据类型，并且对请求大小没有限制.

在 python 中可以使用 `requests` 库

在 Python 中使用 requests 库发送 HTTP 请求是非常流行的。它提供了一种发送 JSON POST 类型请求的快捷方式。让我们来看一个使用 run 端口执行 lua 脚本的例子

[4] 这也就是说我们不能直接将所有代码简单的复制粘贴，还需要做进一步修改，在该等待的时候加上 splash:wait 函数

```
import requests

script = """
splash:go(args.url)
return splash:png()
"""
resp = requests.post('http://localhost:8050/run', json={
    'lua_source': script,
    'url': 'http://example.com'
})
png_data = resp.content
```

python + scrapy

Scrapy 是一个非常流行的爬虫框架，针 Splash + Scrapy 对这种情形，您可以使用 scrapy-splash 库

R 语言

有一个简单的第三方库可以在 R 语言中简单的使用 Splash https://github.com/hrbrmstr/splashr

curl

```
curl --header "Content-Type: application/json" \
     -X POST \
     --data '{"url":"http://example.com","wait":1.0}' \
     'http://localhost:8050/render.html'
```

httpie

httpie 是一个用来发送 HTTP 请求的实用命令行工具。它有一个很好的用于发送 JSON POST 请求的 API

```
http POST localhost:8050/render.png url=http://example.com width=200 > img.png
```

html

您可以直接在 HTML 页面中嵌入 Splash 返回的结果。这种方法不一定是最好的解决办法，因为它每次打开页面都会重新渲染网站。但是您仍然可以使用它

```
<img src="http://splash-url:8050/render.jpeg?url=http://example.com&width=300"/>
```

1.12.2 我得到一个 504 的错误，请帮帮我!

HTTP 504 错误代表您的 Splash 请求在指定超时时间 (默认是 30s) 之内未得到返回。Splash 在达到超时时间的时候会直接停止 Lua 脚本的执行。您可以使用 “*timeout*” *<./api.html#arg-timeout>* 来重新指定超时时间。

请注意这个 `timeout` 不得超过我们设置的最大超时时间, 这个最大超时时间默认为 60s。换句话说您不能指定 `?timeout=300` 的脚本执行时间，如果您这么做将会得到一个错误.

您可以在启动 Splash 的时候使用选项 `--max-timeout` 来增加这个默认的最大超时时间

```
$ docker run -it -p 8050:8050 scrapinghub/splash --max-timeout 3600
```

如果您使用 Docker，请使用上面的命令

```
$ python3 -m splash.server --max-timeout 3600
```

下一个问题在于为什么您的请求需要花 10 分钟来进行渲染? 下面列举了 3 条理由:

1. 加载较慢的站点

有的站点确实很慢，或者它会请求一些远程的资源，这些资源的加载很费时间

如果它真的很慢，那么就只能增加超时时间以及尽量减少对该站的请求。但是在大部分情况下，都是在加载广告和一些不可靠的资源上消耗大量的时间。默认情况下 Splash 会在所有资源都加载完成之后返回，但是在这种情况下最好不要等待它加载这些资源。

您可以在它加载页面之前设置资源的加载超时时间，在资源的超时时间达到之后终止对资源的加载。针对 render 系列的端点，您可以使用 ‘resource_timeout’ 参数。针对 *run <./api.html#run>* 和 execute 您可以使用 splash.resource_timeout 或者 “request:set_timeout“ (请参见: splash:on_request)

永远在代码中设置 resource_timeout 是一个很好的做法，大部分情况下可以设置 `resource_timeout=20`

2. splash lua 脚本执行的任务太多

当脚本需要加载过多页面，或者存在大量等待的时候，超时是不可避免的。在有些时候您必须这么干的话，可以通过选项 `--max-timeout` 来增加最大超时值，并使用较大的超时值

但是在增加超时值前，请先考虑将您的代码分隔为每个短小的步骤，然后一个个的提交到 Splash 上，例如您需要加载 100 个站点，不要在脚本中写一个包含 100 个 URL 的列表，然后循环的提交它们，您可以写一段脚本加载 1 个页面，然后发送 100 次请求到 Splash。这种方法有许多好处：它使脚本更加简单，更健壮，并且能支持并行处理。

3. Splash 实例超载

当 Splash 实例超载的时候，也会产生 504 错误

Splash 是以并行的方式来呈现请求的，但是并不意味着 Splash 会在相同的时间段内渲染所有的请求。这个并发的数量是在启动 Splash 时通过参数 `--slots` 来设置的。当所有线程都被请求占用，新的请求任务会放入任务队列。这个问题是一旦 Splash 接收到请求它就会针对这个请求开始计算时间，而不是等到 Splash 开始处理它的时候计算。所以如果一个请求在内部的任务队列中待太久的话，即使他请求的站点非常快，也会造成超时。为了解决任务队列的这个问题和提高渲染速度，您可以使用多个 Splash 实例，并使用能够维护自己任务队列的负载均衡器。HAProxy 拥有所有这些特征，您可以 点击这里 查看它的配置的例子。使用负载均衡器中的共享队列也有助于提高可靠性，如果需要重启某个 Splash 实例，您不会丢失请求

注解： Nginx (另外一个比较流行的负载均衡器)。仅在商业版本中提供内部的任务队列。Nginx Plus 版

1.12.3 如何在发布版产品中运行 Splash

简单的办法

如果您想快速的入门，请参阅 Aquarium (它是一个简单的 Splash 配置程序)。或者使用像 ScrapingHub 这样的托管服务平台。

不要忘了在您的客户端代码中使用资源的超时时间 (请参见*1. Slow website*)。如果 Splash 返回 5xx 的错误，那么重试几次给这个超时时间设置一个合理的值是十分有意义的事。

困难的方式

如果您希望自己在生成环境中配置，这里有几个小小的清单供您参考:

- Splash 应该作为守护进程，并在产品启动时候启动
- 如果出现错误或者段错误，必须能够重启 Splash
- 必须减少内存的消耗
- 应该启动多个 Splash 实例，以便能够使用所有 CPU 的核或者多个服务器上运行
- 请求队列应该放到负载均衡里面，以便使 Splash 更加健壮 (请参阅*3. Splash 实例超载*)

当然，配置监控、设置管理等等其他平常的东西也可以考虑。

为了守护 Splash 需要在程序启动时启动守护进程，并且在 Splash 崩溃时重启 Splash，您可以考虑使用 Docker。从 Docker 的 1.2 版本以后，您可以同时使用 `--restart` 和 `-d` 选项。您也可以使用一些标准的工具，像 upstart, systemd or supervisor

---

注解： Docker 中 `--restart` 如果不与 `-d` 选项一起，将无法正常工作

Splash 使用未绑定的内存缓冲，因此它最终会占用所有的内存。一个解决的办法是在它占用过量内存时进行重启。Splash 中的 `--maxrss` 参数正是这个作用。您还可以在 Docker 中添加 `--memory` 选项。

在正式产品中固定使用同一个版本的 Splash 会是一个好的做法。相比于使用 `scrapinghub/splash` 来说使用像 `scrapinghub/splash:2.0` 这样的可能会更好

如果您希望设置 Splash 使用的最大内存为 4GB，并且加上守护进程，崩溃重启这些特性，您可以使用下面的命令

```
$ docker run -d -p 8050:8050 --memory=4.5G --restart=always scrapinghub/splash:3.1 --maxrss 4000
```

当然，您可能需要一个负载均衡。这样您可以在 Splash 中进行与 Aquarium 或者 HAProxy 相关的配置

使用 Ansible 的方式

Splash 与 Ansible 的结合相关内容可以通过第三方项目获得:https://github.com/nabilm/ansible-splash.

1.12.4 页面未正常呈现

某些网站通过 Splash 不能正常呈现, 可能的原因如下:

- 没有足够的等待时间，解决方案：多等待一段时间 (请参阅: splash:wait))
- 在私有模式下，本地存储未正常工作。这是一个常见的问题，例如一个网站基于 AngularJS 搭建。如果未正常加载，请关闭私有模式 (请参阅我如何关闭私有模式)
- 某些时候响应体时惰性加载的，或者是在用户产生动作时候才加载 (例如滚动页面)。尝试增加视口大小并等待一定时间以便让所有内容都能够呈现 (请参阅 splash:set_viewport_full)。您可能也需要模拟键盘和鼠标事件 (请参阅：与页面交互)
- Splash 使用的 WebKit 缺少某些功能。现在 Splash 使用 https://github.com/annulen/webkit ，他比 QT WebKit 提供的功能要多得多。我们使用的 WebKit 将与 annulen 的 WebKit 一同更新
- QT 或者 WebKit 的 bug 导致 Splash 挂起或者崩溃。通常 Webkit 对所有的网站都有效，但是针对某些特殊的 js 代码 (或者其他的内容) 会导致这个问题。针对这种情况，您可以在 verbose 模式中重启 Splash(例如: `docker run -it -p8050:8050 scrapinghub/splash -v2`) 。请注意它最终下载了哪些无关紧要的资源并使用 splash:on_request 或者 Request Filters 过滤它们
- 某些崩溃可以通过关闭 HTML5 的支持来解决 (splash.html5_media_enabled 属性或者 html5_media HTTP API 参数)。请注意在默认情况下它们是打开的
- 站点可能会根据 UA 或者代理 IP 的地址来显示不同的信息。您可以使用 splash:set_user_agent 来修改默认的 UA，如果您的 Splash 在云上运行，并且没有得到正常的返回结果。请尝试在本地重现它，以防止站点根据 IP 来返回内容。
- 站点会请求 Flash，您可以通过 splash.plugins_enabled 来允许加载插件
- 站点请求 IndexedDB 。您可以使用 splash.indexeddb_enabled 来开启对它的支持
- 如果没有视频或者其他多媒体文件，请使用 html5_media 参数，或者 splash.html5_media_enabled 来打开对 HTML5 多媒体的支持，或者通过 splash.plugins_enabled 参数来打开对 Flash 的支持

- 网站与 Splash 正在使用的 WebKit 存在兼容性问题。一个快速的 (虽然不太精确) 的解决办法是尝试在 Safari 中打开并检查

如果您在使用 Splash 时出现问题，请尝试在 https://stackoverflow.com 中提问。如果您认为这是 Splash 的一个 bug 请将问题提交到 https://github.com/scrapinghub/splash/issues

1.12.5 如何关闭私有模式

在 Splash>=2.0 的版本中，您可以关闭私有模式 (默认开启)。主要有两种方法

在启动时通过 `--disable-private-mode` 参数，例如如果您在 Docker 中启动

```
$ sudo docker run -it -p 8050:8050 scrapinghub/splash --disable-private-mode
```

如果是在运行状态下，您可以使用 `/execute` 端点，并设置 splash.private_mode_enabled 参数为 `false`

请注意，如果您关闭了私有模式，那么不同请求之间可能会使用同样的浏览器信息 (cookie 不受影响)。如果您下共享环境下使用 Splash，您发送请求中的相关信息可能会影响其他用户发送的请求。

有时您仍然需要关闭私有模式，WebKit 的本地存储在开启私有模式时不能正常工作。并且可能无法为本地缓存提供 JavaScript 填充程序。因此对于某些站点 (某些基于 AngularJS 站) 您需要关闭私有模式。

1.12.6 为什么 Splash 被首先创建

请参阅: kmike 在 reddit 上的回答

1.12.7 为何 Splash 会使用 Lua 做脚本而不是 Python 或者 JavaScript

您可以在 GitHub Issue 找到答案

1.12.8 render.html 返回的值在浏览器上看起来不太正常

当您在浏览器中输入 `http://<splash-server>:8050/render.html?url=<url>` 来检查渲染结果的时候，可能会出现样式和资源无法加载的情况。当资源是采取相对定位的时候，可能会出现这种情况，此时浏览器在加载这些相对定位时采用的基地址是 `http://<splash-server>:8050/render.html?url=<url>` 而不是 `url` 。这不是 Splash 的 bug 而是浏览器的正常行为。

如果您想看看这个页面经过渲染后是什么样子的，您可以使用 render.png 或者 render.jpeg 端点。如果您不想通过截屏的方式查看，但是仍然想在浏览器中查看 HTML 的效果，您可以使用基地址来将相对定位的 url 转化为绝对定位。然后再使用浏览器加载 HTML 代码。

在这种情况下 Splash 的 baseurl 参数不能起到实质性的作用。它可以正常呈现另一台主机上的页面，就好像在原始机器上的页面一样。比如说您可以拷贝一个 HTML 页面到您的机器上，但是使用 baseurl 指向原来的主机。这样 Splash 将会使用原始的 URL 来解析相对的 URL _[#1] 。这样您就可以正确的读取到对应的屏幕截图或者执行 JavaScript 代码。

但是通过传递 baseurl，您需要明确的指示 Splash 来使用它，但是在浏览器中做不到这点。它不会改变 DOM 中相对的 url 的基地址。浏览器在使用这些 url 的时候会将地址栏中的地址作为基地址。

在 DOM 树中更改绝对链接与浏览器在运用基本的 URL 时所作的操作不同。如果您使用 JS 代码来查看链接的 href 属性，它仍然包含相对值，即使您使用了 `<base>` 标签。render.html 返回 DOM 的快照。因此这些链接也不会被改变。

当您在浏览器中加载 render.html 得到的 HTML 页面时，是由您的浏览器来进行相对地址的定位，而不是通过 Splash，所以它的加载可能不太完整。

1.13 scrapy-splash 教程

scrapy-splash 是为了方便 scrapy 框架使用 splash 而进行的封装。它能与 scrapy 框架更好的结合，相比较于在 python 中使用 requests 库或者使用 scrapy 的 Request 对象来说，更为方便，而且能更好的支持异步。

1.13.1 安装

针对 python 来说可以直接使用:

```
$ pip install scrapy-splash
```

1.13.2 配置

1. 首先需要启动 Splash，启动命令如下

```
$ docker run -p 8050:8050 scrapinghub/splash
```

关于 Splash 安装等更多信息请参考 splash 安装文档

#. 然后在对应 scrapy 项目的 settings 里面配置 Splash 服务的地址, 例如:

```
SPLASH_URL = 'http://192.168.59.103:8050'
```

#. 在 settings 中的 DOWNLOADER_MIDDLEWARES 加上 splash 的中间件，并设置 HttpCompressionMiddleware 对象的优先级

```
DOWNLOADER_MIDDLEWARES = {
    'scrapy_splash.SplashCookiesMiddleware': 723,
    'scrapy_splash.SplashMiddleware': 725,
    'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware': 810,
}
```

#. 在 SPIDER_MIDDLEWARES 中安装 splash 的 SplashDeduplicateArgsMiddleware 中间件

```
SPIDER_MIDDLEWARES = {
    'scrapy_splash.SplashDeduplicateArgsMiddleware': 100,
}
```

#. 您还可以设置对应的过滤中间件——DUPEFILTER_CLASS

```
DUPEFILTER_CLASS = 'scrapy_splash.SplashAwareDupeFilter'
```

#. 您可以设置 scrapy.contrib.httpcache.FilesystemCacheStorage 来使用 Splash 的 HTTP 缓存

```
HTTPCACHE_STORAGE = 'scrapy_splash.SplashAwareFSCacheStorage'
```

1.13.3 用法

如果要使用 Splash 来对页面进行渲染，您可以使用 SplashRequest 来代替原始 scrapy 中的 Request，示例如下:

```
yield SplashRequest(url, self.parse_result, callback # 任务完成之后对应的回调函数
    #args 设置的是端点 API 的参数，关于 API 参数问题，请参考: `Splash HTTP API <./api.html>`_
    args={
        # 可选参数，表示 spalsh 在执行完成之后会等待一段时间后返回
        'wait': 0.5,
        #url 是一个必须的参数，表明将要对哪个 url 进行请求
        'url' : "http://www.example.com",
        #http_method: 表示 Splash 将向目标 url 发送何种请求
        'http_method': 'GET'
        # 'body' 用于 POST 请求，作为请求的请求体
        # 'lua_source' 如果需要执行 lua 脚本，那么这个参数表示对应 lua 脚本的字符串
    },
    endpoint='render.json', # optional; default is render.html
    splash_url='<url>',     # optional; overrides SPLASH_URL
    slot_policy=scrapy_splash.SlotPolicy.PER_DOMAIN,  # optional,
    # "meta" 是一个用来向回调函数传入参数的方式，在回调函数中的 response.meta 中可以取到这个地方传入的
参数
)
```

如果在 splash 中使用 lua 脚本，那么 args 中的内容会通过 main 函数的 `splash.args` 参数传入，其余的内容会通过第二个参数 `args` 传入。

比如下面有一个简单的用户登录的例子:

```
lua_script= '''
function main(splash, args)
```

(continues on next page)

(续上页)

```
local ok, reason = splash:go(args.url)
user_name = args.user_name
user_passwd = args.user_passwd
user_text = splash:select("#email")
pass_text = splash:select("#pass")
login_btn = splash:select("#loginbutton")
if (user_text and pass_text and login_btn) then
    user_text:send_text(user_name)
    pass_text:send_text(user_passwd)
    login_btn:mouse_click{}
end

splash:wait(math.random(5, 10))
return {
    url = splash:url(),
    cookies = splash:get_cookies(),
    headers = splash.args.headers,
 }
end'''

yield SplashRequest(
    url=self.login_url,
    endpoint="execute",
    args={
        "wait": 30,
        "lua_source": lua_script,
        "user_name": "xxxx",  # 在 Lua 脚本中这个参数可用通过 args.user_name 取得
        "user_passwd": "xxxx",
    },
    meta = {"user_name" : "xxxx"},
    callback=self.after_login,
    errback=self.error_parse,
)
```

上述代码提交了一个 Splash 的请求，在脚本中首先获取用户名和密码的输入框元素和对应的提交按钮元素，接着填入用户名和密码，最后点击提交并返回对应的 cookie。回调函数 after_login 的代码如下:

```
def after_login(self, response):
    # 首先根据一定条件判断登录是否成功
    self.login_user = response.meta["user_name"] # 保存当前登录用户
    self.cookie = response.data["cookies"]  # 保存 cookie
```

在回调函数中，可以通过 response.data 来获取 lua 脚本中返回的内容，而对应的 HTML 代码的获取方式与使用传统的 Request 方式相同。

另外在回调函数中可以通过 response.meta 来获取 Request 中 meta 传入的参数。

上述示例演示了如何使用 SplashRequest 来像 Splash 发送渲染请求，以及如何在回调函数中获取 lua 脚本中的返回、以及如何在回调函数中获取 lua 脚本中的返回、如何向回调函数传递参数。

当然您也可以使用常规的 scrapy.Request 来向 Splash 发送请求，发送的示例如下:

```
yield scrapy.Request(url, self.parse_result, meta={
    'splash': {
        'args': {
            # 在此处设置端点 API 的参数
            'html': 1,
            'png': 1,

            # 'url' is prefilled from request url
            # 'http_method' is set to 'POST' for POST requests
            # 'body' is set to request body for POST requests
        },

        # optional parameters
        'endpoint': 'render.json',  # optional; default is render.json
        'splash_url': '<url>',      # optional; overrides SPLASH_URL
        'slot_policy': scrapy_splash.SlotPolicy.PER_DOMAIN,
        'splash_headers': {},       # optional; a dict with headers sent to Splash
        'dont_process_response': True, # optional, default is False
        'dont_send_headers': True,  # optional, default is False
        'magic_response': False,    # optional, default is True
    }
})
```

splash 参数中的内容是用于 splash 的，使用这个参数表明我们希望向 splash 发送渲染请求。

最终它们会被组织成 `request.meta['splash']` 。在 scrapy 处理这些请求的时候根据这个来确定是否创建 spalsh 的中间件，最终请求会被中间件以 HTTP API 的方式转发到 splash 中。

splash 中各个参数的作用如下:

- **meta[‘splash’][‘args’] 是最终发送到 splash HTTP API 的参数**
  - `url` 表示目标站点的 url
  - `http_method` 表示向 url 发送的 HTTP 的请求方式
  - `body` 是采用 POST 方式发送请求时，请求体的内容
- meta[‘splash’][‘cache_args’] 表示将要被作为缓冲的参数的列表字符串，以分号分隔
- meta[‘splash’][‘endpoint’] 表示对应的端点
- meta[‘splash’][‘splash_url’] 与 settings 文件中的 SPLASH_URL 作用相同，但是会优先采用这里的设置

- meta[‘splash’][‘splash_headers’] 即将发送到 splash 服务器上的请求头信息，注意，这里它不是最终发送到对应站点的请求头信息

由于本人水平有限以及当时项目需求并没有对它的用法做很深入的了解，更为详细的用法请参见：https://github.com/scrapy-plugins/scrapy-splash

1.14 写在最后的话

在投入到这个翻译项目的这段时间中，我也感觉到了自己的英文水平实在太差，很多地方自己在当初阅读时产生了一定的理解失误，导致后来在写程序时出现各种问题，对于这类问题，有些地方我没有采用直译的方式，或者一时找不到好的语言来描述，这个时候我会在下面写上我的注解，希望对各位有一定的帮助。

有的内容我自己没有仔细的研究过，也没有写过对应的 DEMO，所以有的地方采用直译或者意译，这些地方我都是按照自己的理解去写的，可能会出现理解上的偏差和错误，有时候也会误导参考这篇中文文档的朋友。在这里对这类情况表示歉意，各位朋友可以结合原来的英文文档来查阅这篇中文文档，减少由于本人水平的问题造成的误解，另外也请各位发现问题后与我联系，以便能对该文档进行修改。

这里在原文档后面有关于如何提问和如何参与 splash 这个开源项目的内容，这里不再翻译了，作为补偿也是作为之前对项目的一点总结，我新加了一篇 scrapy-splash 的简单的使用说明，希望能对大家有用

联系方式

- 邮箱：liu1793222129@163.com
- QQ: 654170359
- 博客: http://www.masimaroweb.com
- GitHub: https://github.com/aMonst