# 云压测

# JavaScript API 列表

腾讯云

# 文档目录

Conn.recv
Conn.close

# JavaScript API 列表
# JavaScript API 列表概述

最近更新时间：2024-12-30 16:05:03

本部分介绍 PTS 提供的 JavaScript 模块及其 API，覆盖了编写压测脚本时所需的协议、函数、流程、断言的用法等内容。
关于 PTS 脚本模式压测的介绍，您可参见 脚本模式压测 。

# pts/global
# 模块概览

最近更新时间：2024-12-30 16:05:03

Javascript API 的 pts/global 模块实现了部分内置函数和对象。

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| open(filePath, mode?) | string 或 ArrayBuffer | 用于打开文件 |
| int64(v) | object | 用于实现 int64 类型 |
| uint64(v) | object | 用于实现 uint64 类型 |

## 对象

| 对象 | 描述 |
| --- | --- |
| BasicAuth | 配置 HTTP 中的 basicAuth 配置 |
| Certificate | 配置 TLSConfig 中的 certificates 配置 |
| HTTP | 全局参数配置 Option 中的 http 配置 |
| Option | 全局的配置参数 |
| TLSConfig | 全局参数配置 Option 中的 tlsConfig 配置项 |
| TRPC | 全局参数配置 Option 中的 trpc 配置 |
| WS | 全局参数配置 Option 中的 ws 配置 |

# open

最近更新时间：2024-12-30 16:05:03

**open 内置函数用于打开文件。**

```
open(filePath: string, mode?: '' | 'b'): string | ArrayBuffer
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| filePath | string | 文件的相对路径。 |
| mode? | '' 或 'b' | 打开模式（可选）。文本文件无需指定 mode，返回值为 string；二进制文件必须传入 'b' mode，返回值为 ArrayBuffer。 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string 或 ArrayBuffer | 文件数据。 |

## 样例

**打开文件：**

```
export default function () {
    let data = open('test1.json'); // 默认打开文本文件
    console.log(data); // {"a":"b"}
    data = open('test2.bin', 'b'); // 'b' 模式打开二进制文件
    console.log(data); // [object ArrayBuffer]
};
```

# int64

最近更新时间：2024-12-30 16:05:03

int64 内置函数用于实现 int64 类型。

```
int64(v: string): object
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| v | string | int64 类型数据的字符串格式 |

## 返回

| 类型 | 描述 |
| --- | --- |
| object | 返回的对象，具有以下方法：<br>• toString()，返回字符串；<br>• toJSON()，使用 JSON.stringify() 时会被调用； |

## 样例

### 使用 int64：

```
export default function () {
  let a = {
      k: int64("9223372036854775807")
  }
  // toJSON() 方法在 JSON.stringify() 时会被调用
  // {"k":"9223372036854775807"}
  console.log(JSON.stringify(a));
  // 9223372036854775807
  console.log(a.k.toString());
}
```

# uint64

最近更新时间：2025-01-03 14:17:23

uint64 内置函数用于实现 uint64 类型。

```
uint64(v: string): object
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| v | string | uint64 类型数据的字符串格式 |

## 返回

| 类型 | 描述 |
| --- | --- |
| object | 返回的对象，具有以下方法：<br>• toString()，返回字符串<br>• toJSON()，使用 JSON.stringify() 时会被调用 |

## 示例

### 使用 uint64：

```
export default function () {
  let a = {
      k: uint64("18446744073709551615")
  }
  // toJSON() 方法在 JSON.stringify() 时会被调用
  // {"k":"18446744073709551615"}
  console.log(JSON.stringify(a));
  // 18446744073709551615
  console.log(a.k.toString());
}
```

# BasicAuth

最近更新时间：2024-12-30 16:05:03

BasicAuth 是参数配置 HTTP 中的 basicAuth 配置。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| username | string | 用户名 |
| password | string | 密码 |

## 样例

### 使用 BasicAuth：

```
export const option = {
  http: {
    basicAuth: {
      username: 'username',
      password: 'password',
    }
     }
}
```

# Certificate

最近更新时间：2024-12-30 16:05:03

Certificate 是配置参数 TLSConfig 中的 certificates 配置项。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| cert | string | 证书 |
| key | string | 私钥 |

## 样例

### 使用 Certificate：

```
export const option = {
  tlsConfig: {
    'localhost': {
       insecureSkipVerify: false,
       rootCAs: [open('tool/tls/twoway/ca.crt')],
       certificates: [
         {
           cert: open('tool/tls/twoway/client.crt'),
           key: open('tool/tls/twoway/client.key')
         }
       ]
    }
  }
}
```

# HTTP

最近更新时间：2024-12-30 16:05:03

HTTP 是全局参数配置 Option 中的 http 配置。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| maxRedirects? | number | 可选，最大重定向跳转次数 |
| maxIdleConns? | number | 可选，单个 VU 最大活跃连接数 |
| maxIdleConnsPerHost? | number | 可选，单个 VU 单个域名最大活跃连接数 |
| disableKeepAlives? | boolean | 可选，是否禁用长连接 |
| headers? | Record<string, string> | 可选，请求头 |
| timeout? | number | 可选，请求超时时间，单位毫秒 |
| basicAuth? | BasicAuth | 可选，基础鉴权 |
| discardResponseBody? | boolean | 可选，是否丢弃回包 |
| http2? | boolean | 可选，是否启用 HTTP2 |

## 样例

### 使用 HTTP Option：

```
export const option = {
  http: {
      maxRedirects: 5,
      maxIdleConns: 50,
      maxIdleConnsPerHost: 10,
      disableKeepAlives: true,
      headers: {
        'key': 'value'
      },
      timeout: 3000,
      basicAuth: {
        username: 'user',
```

```
          password: 'passwd'
        },
        discardResponseBody: true,
        http2: true
    }
}
```

# Option

最近更新时间：2024-12-30 16:05:03

Option 是全局的配置参数，对于不同类型的请求可以设置不同的配置。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| setupTimeoutSeconds? | number | 可选，setup 函数的超时时间，单位为秒 |
| teardownTimeoutSeconds? | number | 可选，teardown 函数的超时时间，单位为秒 |
| tlsConfig? | Record<string, TLSConfig> | 可选，TLS 配置 |
| http? | HTTP | 可选，HTTP 选项配置 |
| trpc? | TRPC | 可选，TRPC 选项配置 |
| ws? | WS | 可选，WS 选项配置 |

# TLSConfig

最近更新时间：2024-12-30 16:05:03

TLSConfig 是全局参数配置 Option 中的 tlsConfig 配置项。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| insecureSkipVerify | boolean | 控制客户端是否验证服务器的证书链和主机名；如果为真，crypto/tls 接受服务器提供的任何证书以及该证书中的任何主机名 |
| rootCAs | Array | 根证书 |
| certificates | Array | 客户端证书列表 |
| serverName | string | 匹配证书里的主机名 |

## 样例

### 使用 TLSConfig：

```
export const option = {
  tlsConfig: {
    'localhost': {
       insecureSkipVerify: false,
       rootCAs: [open('ca.crt')],
       certificates: [
         {
           cert: open('client.crt'),
           key: open('client.key')
         }
       ],
       serverName: "xxx.com"
    }
  }
}
```

# TRPC

最近更新时间：2024-12-30 16:05:03

TRPC 是全局参数配置 Option 中的 trpc 配置。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| env? | string | 可选，123 环境名，例如 formal、pre、test |
| namespace? | string | 可选，环境类型，例如 Production、Development |
| sendOnly? | boolean | 可选，trpc 只发不收选项 |

## 样例

### 使用 TRPC Option：

```
export const option = {
  trpc: {
    env: 'formal',
    namespace: 'Development',
    sendOnly: true
  }
}
```

# WS

最近更新时间：2024-12-30 16:05:03

WS 是全局参数配置 Option 中的 ws 配置。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| writeControlTimeout? | number | 可选，写控制指令超时时间，单位毫秒，默认 10s |
| handshakeTimeout? | number | 可选，握手超时时间，单位毫秒，默认 30s |
| writeTimeout? | number | 可选，写消息超时时间，单位毫秒，默认不限制 |
| readTimeout? | number | 可选，读消息超时时间，单位毫秒，默认不限制 |

## 样例

### 使用 WS Option：

```
export const option = {
  ws: {
    writeControlTimeout: 10000,
    handshakeTimeout: 10000,
    writeTimeout: 3000,
    readTimeout: 3000,
  }
}
```

# pts/http
# 模块概览

最近更新时间：2025-01-03 14:17:23

Javascript API 的 pts/http 模块实现了 HTTP 相关的功能。

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| batch(requests, opt?) | BatchResponse[] | 批量发起 HTTP 请求 |
| delete(url, request?) | Response | 发起 DELETE 请求 |
| do(request) | Response | 发起 HTTP 请求 |
| file(data, name?, contentType?) | http.file | 构造 FormData 中上传的文件对象 |
| get(url, request?) | Response | 发起 GET 请求 |
| head(url, request?) | Response | 发起 HEAD 请求 |
| patch(url, body, request?) | Response | 发起 PATCH 请求 |
| post(url, body, request?) | Response | 发起 POST 请求 |
| put(url, body, request?) | Response | 发起 PUT 请求 |

## 对象

| 对象 | 描述 |
| --- | --- |
| BatchOption | 批量发起 HTTP 请求调用 http.batch 时的配置选项 |
| BatchResponse | 批量发起 HTTP 请求调用 http.batch 时得到的请求结果 |
| File | 用于 FormData 中上传的文件对象，由 http.file 生成 |
| FormData | 构造 form-data 类型的请求体 |

| | |
|---|---|
| Request | 发起请求时的请求 |
| Response | 发起请求后获得的请求结果 |

# http.batch

最近更新时间：2024-12-30 16:05:03

**http.batch 用于批量发起 HTTP 请求。**

```
batch(requests: Request[], opt?: BatchOption): BatchResponse[]
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| requests | Request[] | 请求对象数组 |
| opt? | BatchOption | 可选，批量发起请求的配置选项 |

## 返回

| 类型 | 描述 |
| --- | --- |
| BatchResponse[] | 批量请求结果数组 |

## 样例

**发起批量请求：**

```
import http from 'pts/http';

export default function () {
  const responses = http.batch([
    {
      method: 'GET',
      url: 'http://httpbin.org/get?a=1',
      headers: { a: '1, 2, 3' },
      query: { b: '2' },
    },
    {
      method: 'GET',
      url: 'http://httpbin.org/get?a=1',
      headers: { a: '1, 2, 3' },
      query: { b: '2' },
    },
  ]);
```

```
    console.log(JSON.stringify(responses));
}
```

# http.delete

最近更新时间：2025-01-03 14:17:23

**http.delete 用于发起 DELETE 请求。**

```
delete(url:string, request?: Request): Response
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| url | string | URL 字符串 |
| request | Request | 可选，请求对象，请求对象中的 method 和 url 字段不会覆盖当前方法 http.delete 中的 method 和 url |

## 返回

| 类型 | 描述 |
| --- | --- |
| Response | 响应对象 |

## 示例

**发起 DELETE 请求：**

```
import http from 'pts/http';

export default function () {
  const data = { user_id: '12345' };
  const req = {
    query: data,
  };
  const resp = http.delete('http://httpbin.org/delete', req);
  console.log(resp.json().args.user_id); // 12345
}
```

# http.do

最近更新时间：2024-12-30 16:05:03

**http.do 用于发起 HTTP 请求。**

```
do(request: Request): Response
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| request | Request | 请求对象 |

## 返回

| 类型 | 描述 |
| --- | --- |
| Response | 响应对象 |

## 样例

**发起 HTTP 请求：**

```
import http from 'pts/http';

export default function () {
  const req = {
    method: 'post',
    url: 'http://httpbin.org/post',
    headers: { 'Content-Type': 'application/json' },
    query: { a: '1' },
    body: { user_id: '12345' },
  };
  const resp = http.do(req);
  console.log(resp.json().args.a); // 1
  console.log(resp.json().json.user_id); // 12345
}
```

# http.file

最近更新时间：2024-12-30 16:05:03

**http.file 用于构造 FormData 中上传的文件对象。**

```
file(data: string | ArrayBuffer, name?: string, contentType?: string):
File
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| data | string 或 ArrayBuffer | 文件内容 |
| name? | string | 可选，文件名，默认为纳秒级时间戳 |
| contentType? | string | 可选，内容类型，默认为 application/octet-stream |

## 返回

| 类型 | 描述 |
| --- | --- |
| File | 文件对象 |

## 样例

**构造用于 FormData 上传的文件对象：**

```
import http from 'pts/http';

const data = open('./sample/tmp.js');

export default function () {
  const file = http.file(data);
  //@ts-ignore   忽略校验
  console.log(file.data.length); // 231
  console.log(file.name); // 1635403323707745000
  console.log(file.contentType); // application/octet-stream
}
```

**构造用于 FormData 上传的文件对象，并传入 name 和 contentType 参数：**

```
import http from 'pts/http';

const data = open('./sample/tmp.js');

export default function () {
  const file = http.file(data, 'data', 'application/json');
  //@ts-ignore    忽略校验
  console.log(file.data.length); // 231
  console.log(file.name); // data
  console.log(file.contentType); // application/json
}
```

# http.get

最近更新时间：2024-12-30 16:05:03

**http.get 用于发起 GET 请求。**

```
get(url:string, request?: Request): Response
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| url | string | URL 字符串 |
| request | Request | 可选，请求对象，请求对象中的 method 和 url 字段不会覆盖 GET 和 url 参数 |

## 返回

| 类型 | 描述 |
| --- | --- |
| Response | 响应对象 |

## 样例

**发起 GET 请求：**

```
import http from 'pts/http';

export default function () {
  const data = { user_id: '12345' };
  const req = {
    query: data,
  };
  const resp = http.get('http://httpbin.org/get', req);
  console.log(resp.json().args.user_id); // 12345
}
```

# http.head

最近更新时间：2024-12-30 16:05:03

**http.head 用于发起 HEAD 请求。**

```
head(url:string, request?: Request): Response
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| url | string | URL 字符串 |
| request | Request | 可选，请求对象，请求对象中的 method 和 url 字段不会覆盖 HEAD 和 url 参数 |

## 返回

| 类型 | 描述 |
| --- | --- |
| Response | 响应对象 |

## 样例

**发起 HEAD 请求：**

```
import http from 'pts/http';

export default function () {
  const data = { user_id: '12345' };
  const req = {
    query: data,
  };
  const resp = http.head('http://httpbin.org/get', req);
  console.log(resp.statusCode); // 200
}
```

# http.patch

最近更新时间：2024-12-30 16:05:03

**http.patch 用于发起 PATCH 请求。**

```
patch(url:string, body: string | object | Record<string, string>,
request?: Request): Response
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| url | string | URL 字符串 |
| body | string、object 或 Record<string, string> | 请求体 |
| request | Request | 可选，请求对象，请求对象中的 method、url 和 body 字段不会覆盖 HEAD 和 url、body 参数 |

## 返回

| 类型 | 描述 |
| --- | --- |
| Response | 响应对象 |

## 样例

**发起 PATCH 请求：**

```
import http from 'pts/http';

export default function () {
  const data = { user_id: '12345' };
  const headers = { 'Content-Type': 'application/json' };
  const request = {
    headers,
  };
  //@ts-ignore    忽略校验
  const resp1 = http.patch('http://httpbin.org/patch', data, request);
  const resp2 = http.patch('http://httpbin.org/patch', '123', request);
  console.log(resp1.json().json.user_id); // 12345
```

```
    console.log(resp2.json().json); // 123
}
```

# http.post

最近更新时间：2024-12-30 16:05:03

**http.post 用于发起 POST 请求。**

```
post(url:string, body: string | object | Record<string, string>,
request?: Request): Response
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| url | string | URL 字符串 |
| body | string、object 或 Record<string, string> | 请求体 |
| request | Request | 可选，请求对象，请求对象中的 method、url 和 body 字段不会覆盖 HEAD 和 url、body 参数 |

## 返回

| 类型 | 描述 |
| --- | --- |
| Response | 响应对象 |

## 样例

**发起 POST 请求：**

```
import http from 'pts/http';

export default function () {
  const data = { user_id: '12345' };
  const headers = { 'Content-Type': 'application/json' };
  const request = {
    headers,
  };
  const resp1 = http.post('http://httpbin.org/post', data, request);
  const resp2 = http.post('http://httpbin.org/post', '123', request);
  console.log(resp1.json().json.user_id); // 12345
```

```
    console.log(resp2.json().json); // 123
}
```

# http.put

最近更新时间：2024-12-30 16:05:03

**http.put 用于发起 PUT 请求。**

```
put(url:string, body: string | object | Record<string, string>,
request?: Request): Response
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| url | string | URL 字符串 |
| body | string、object 或 Record<string, string> | 请求体 |
| request | Request | 可选，请求对象，请求对象中的 method、url 和 body 字段不会覆盖 HEAD 和 url、body 参数。 |

## 返回

| 类型 | 描述 |
| --- | --- |
| Response | 响应对象 |

## 样例

**发起 PUT 请求：**

```
import http from 'pts/http';

export default function () {
  const data = { user_id: '12345' };
  const headers = { 'Content-Type': 'application/json' };
  const request = {
    headers,
  };
  const resp1 = http.put('http://httpbin.org/put', data, request);
  const resp2 = http.put('http://httpbin.org/put', '123', request);
```

```
  console.log(resp1.json().json.user_id); // 12345
  console.log(resp2.json().json); // 123
}
```

# BatchOption

最近更新时间：2024-12-30 16:05:03

BatchOption 用于批量发起 HTTP 请求时的配置选项。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| parallel? | number | 可选，并行数，默认 20 |

## 使用样例

### 使用 BatchOption 进行批量请求的配置：

```
import http from 'pts/http';

export default function () {
  const responses = http.batch(
    [
      {
        method: 'GET',
        url: 'http://httpbin.org/get?a=1',
        headers: { a: '1, 2, 3' },
        query: { b: '2' },
      },
      {
        method: 'GET',
        url: 'http://httpbin.org/get?a=1',
        headers: { a: '1, 2, 3' },
        query: { b: '2' },
      },
    ],
    // BatchOption 配置选项
    {
      parallel: 1,
    },
  );
  console.log(JSON.stringify(responses));
}
```

# BatchResponse

最近更新时间：2024-12-30 16:05:03

BatchResponse 是利用 http.batch 批量发起 HTTP 请求得到的请求结果。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| error | string | 错误，不为空则表示请求出错。 |
| response | Response | 请求结果 |

## 样例

**批量发起请求并获得结果：**

```
import http from 'pts/http';

export default function () {
  const responses = http.batch(
    [
      {
        method: 'GET',
        url: 'http://httpbin.org/get?a=1',
        headers: { a: '1, 2, 3' },
        query: { b: '2' },
      },
      {
        method: 'GET',
        url: 'http://httpbin.org/get?a=1',
        headers: { a: '1, 2, 3' },
        query: { b: '2' },
      },
    ],
    {
      parallel: 1,
    },
  );

  // 200 OK
  // 200 OK
```

```
  for (const resp of responses) {
    console.log(resp.response.status);
  }
}
```

# File

最近更新时间：2024-12-30 16:05:03

File 表示用于 FormData 中上传的文件对象，由 http.file 生成。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| contentType | string | 内容类型，默认为 "application/octet-stream" |
| data | string 或 ArrayBuffer | 文件内容，通常使用 open() 的返回值 |
| name | string | 文件名，默认为纳秒级时间戳 |

## 样例

```
import http from 'pts/http';

const data = open('./sample/tmp.js');

export default function () {
  const file = http.file(data, 'data', 'application/json');
  console.log(file.data.length); // 231
  console.log(file.name); // data
  console.log(file.contentType); // application/json
}
```

# FormData
# FormData 概览

最近更新时间：2024-12-30 16:05:03

FormData 用于构造 form-data 类型的请求体。

### 构造函数

通过 new 进行对象实例的创建，如下所示：

```
new FormData(): FormData
```

### 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| append(key, value) | void | 向 form-data 中添加键值对数据 |
| body() | ArrayBuffer | 返回 form-data 内容，且不能再进行 append |
| contentType() | string | 返回 form-data 的 ContentType |

### 样例

```
import http from 'pts/http';

const data = open('./sample/tmp.js');

export default function () {
  // 通过 new 构造 FromData 实例
  const formData = new http.FormData();

  formData.append('text', 'text');
  formData.append('file', http.file(data, 'tmp.js'));

  console.log(formData.contentType());

  const resp = http.post('http://httpbin.org/post', formData.body(), {
    headers: {'Content-Type': formData.contentType()}
  });
  console.log('formData: ', resp.body);
```

```
};
```

# FormData.append

最近更新时间：2024-12-30 16:05:03

FormData.append 用于向 form-data 数据添加新的键值对数据。

```
append(key: string, value: string | File): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 添加数据的键 |
| value | string 或 File | 添加数据的值，可以是 string 或 File 类型 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

```
import http from 'pts/http';

const data = open('./sample/tmp.js');

export default function () {
  const formData = new http.FormData();

  // 添加 value 为 string 的数据
  formData.append('text', 'text');
  // 添加 value 为 File 的数据
  formData.append('file', http.file(data, 'tmp.js'));

  const resp = http.post('http://httpbin.org/post', formData.body(), {
    headers: {'Content-Type': formData.contentType()}
  });
  console.log('formData: ', resp.body);
};
```

# FormData.body

最近更新时间：2024-12-30 16:05:03

Form.body 用于返回 form-data 内容，且调用后不能再进行 append。

```
body(): ArrayBuffer
```

## 返回

| 类型 | 描述 |
| --- | --- |
| ArrayBuffer | 请求体内容 |

## 样例

```
import http from 'pts/http';

const data = open('./sample/tmp.js');

export default function () {
  const formData = new http.FormData();

  formData.append('text', 'text');
  formData.append('file', http.file(data, 'tmp.js'));

  // 输出 [object ArrayBuffer]
  console.log(formData.body());

  const resp = http.post('http://httpbin.org/post', formData.body(), {
    headers: {'Content-Type': formData.contentType()}
  });

  console.log('formData: ', resp.body);
};
```

# FormData.contentType

最近更新时间：2024-12-30 16:05:03

FormData.contentType 用于获取 form-data 的 ContentType 字段。

```
contentType(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | ContentType 字段值 |

## 样例

```
import http from 'pts/http';

const data = open('./sample/tmp.js');

export default function () {
  const formData = new http.FormData();

  formData.append('text', 'text');
  formData.append('file', http.file(data, 'tmp.js'));

  // 返回 "multipart/form-data; boundary=xxxxxx"
  console.log(formData.contentType());

  const resp = http.post('http://httpbin.org/post', formData.body(), {
    headers: {'Content-Type': formData.contentType()}
  });
  console.log('formData: ', resp.body);
};
```

# Request

最近更新时间：2024-12-30 16:05:03

Request 是发起请求时的请求结构。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| basicAuth? | BasicAuth | 基础鉴权 |
| body? | string、object 或 ArrayBuffer | 要发送的请求体，在使用 http.do 方法时才需要指定 |
| chunked? | function，(body string) => void | 当数据以一系列分块的形式进行发送时，如果指定了 chunked 函数，会按行读取响应体并进行回调函数的运行 |
| contentLength? | number | 记录关联内容的长度；-1 表示长度未知，>=0 表示可以从 body 中读取给定的字节数 |
| discardResponseBody? | boolean | 丢弃响应体，适用于响应体太大且不需要对象应体内容进行 check 的场景 |
| headers? | Record<string, string> | 请求头 |
| host? | string | host 或 host:port |
| maxRedirects? | number | 最大重定向跳转次数 |
| method? | string | 指定 HTTP 方法，如 GET、PUT、POST 等，只有当使用 http.do 方法时才需要指定 |
| path? | string | 路径，相对路径省略前导斜杠 |
| query? | Record<string, string> | 与请求一同发送的 URL 参数 |
| scheme? | "http" \| "https" | 协议，填写 "http" 或 "https" |
| service? | string | 在 PTS 中，按照 url 识别不同 service 并基于该维度生成报表；如当 url 为 http://demo.com/{id}，不同的 id 会被识别为不同的 service，可以通过指定 service，将此类请求在报表中归类到同一个服务下 |

| | | |
|---|---|---|
| timeout? | number | 客户端发出的请求的时间限制，超时包括连接时间、任何重定向和读取响应正文，单位为毫秒 |
| url? | string | 要访问的 URL，只有当使用 do 方法时才需要指定 |

## 样例

### 在 http.do 方法中使用 Request：

```
import http from 'pts/http';

export default function () {
  const req = {
    method: 'post',
    url: 'http://httpbin.org/post',
    headers: { 'Content-Type': 'application/json' },
    query: { a: '1' },
    body: { user_id: '12345' },
  };
  const resp = http.do(req);
  console.log(resp.json().args.a); // 1
  console.log(resp.json().json.user_id); // 12345
}
```

# Response
# Response 概览

最近更新时间：2024-12-30 16:05:03

Response 是发起请求后获得的请求结果。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| body | string | 服务器返回的响应 |
| contentLength | number | 服务器响应体长度 |
| headers | Record<string, string> | 服务器响应的 HTTP 头 |
| proto | string | 协议，如 "HTTP/1.0" |
| request | Request | 为获得此响应而发送的请求 |
| responseTimeMS | number | 请求的响应时间，单位为毫秒 |
| status | string | 来自服务器响应的 HTTP 状态消息，如 "200 OK" |
| statusCode | number | 来自服务器响应的 HTTP 状态代码，如 200 |

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| json() | any | 将 Response.body 反序列化为 json |

## 样例

```
import http from 'pts/http';

export default function () {
  const req = {
    method: 'post',
    url: 'http://httpbin.org/post',
    headers: { 'Content-Type': 'application/json' },
```

```
    query: { a: '1' },
    body: { user_id: '12345' },
  };
  const resp = http.do(req);

  console.log(resp.json()); // [Object object]
  console.log(resp.json().args.a); // 1
  console.log(resp.json().json.user_id); // 12345
}
```

# Response.json

最近更新时间：2024-12-30 16:05:03

Response.json 将 Response.body 反序列化为 json。

```
json(): any
```

## 返回

| 类型 | 描述 |
| --- | --- |
| any | 返回代表 Response.body 的对象 |

## 样例

```
import http from 'pts/http';

export default function () {
  const req = {
    method: 'post',
    url: 'http://httpbin.org/post',
    headers: { 'Content-Type': 'application/json' },
    query: { a: '1' },
    body: { user_id: '12345' },
  };
  const resp = http.do(req);

  console.log(resp.json()); // [Object object]
  console.log(resp.json().args.a); // 1
  console.log(resp.json().json.user_id); // 12345
}
```

# pts
# 模块概览

最近更新时间：2024-12-30 16:05:03

JavaScript API 中的 pts 模块实现了测试的基本功能。

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| check(name, callback, [interrupt]) | boolean | 针对请求返回的结果做进一步的检查，若返回失败，则代表当前检查不通过。 |
| sleep(seconds) | void | 暂停执行指定的时间长度。 |
| step(name, callback) | void | 将压测场景分为不同步骤，在最终的压测报告中实现步骤的区分。 |
| metadata() | Metadata | 返回压测任务的元数据。 |

## 对象

| 对象 | 描述 |
| --- | --- |
| Metadata | 包含了压测任务元数据的 Object，调用 pts.metadata() 方法时返回。 |

# pts.check

最近更新时间：2024-12-30 16:05:03

**在脚本执行过程中，针对请求返回的结果做进一步的检查，若返回失败则代表当前检查不通过。**

```
check(name: string, callback: () => boolean, response?: Response):
boolean
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| name | string | 检查点的名字 |
| callback | function | 用于检查的函数，该函数应返回 boolean 类型 |
| response（可选） | Response | 传入被检查的请求的响应，用于开启记录检查点日志 |

## 返回

| 类型 | 描述 |
| --- | --- |
| boolean | 检查结果；true 为检查通过，false 为检查不通过 |

## 样例

**检查 HTTP 请求的响应状态码是否为 200：**

```
import http from 'pts/http';
import { check } from 'pts';

export default function () {
    const resp =
http.get('http://mockhttpbin.pts.svc.cluster.local/get');
    check('statusCode is 200', () => resp.statusCode === 200); // 设置检查
点，以统计检查点指标
    check('statusCode is 200', () => resp.statusCode === 200, resp); //
设置检查点，以统计检查点指标、并记录检查点日志
};
```

# pts.metadata

最近更新时间：2024-12-30 16:05:03

**在脚本执行过程中，获取压测任务的元数据。**

```
metadata(): Metadata;
```

## 返回

| 类型 | 描述 |
| --- | --- |
| Metadata | Object，包含压测任务的元数据 |

## 样例

**获取当前压测任务的元数据：**

```
import { metadata } from 'pts';

export default function () {
    // md 为 Metadata 的 interface 对象
    let md = metadata();
    console.log(md.userID); // 123456
    console.log(md.appID); // 123456
    console.log(md.scenarioID); // scenario-xxxxxxxx
    console.log(md.region); // ap-guangzhou
    console.log(md.jobID); // job-xxxxxxxx
}
```

# pts.step

最近更新时间：2024-12-30 16:05:03

**在脚本执行的过程中，将执行过程分为不同步骤，在最终的压测报告中实现不同步骤的区分。**

```
step(name: string, callback: (() => void));
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| name | string | 步骤的名称标识信息 |
| callback | function | 步骤内执行的函数，由需要执行的脚本语句组成 |

## 样例

**将脚本执行过程划分为多个步骤：**

```
import http from 'pts/http';
import { step } from 'pts';

export default function () {
    step('get1', function () {
        http.get('http://httpbin.org/get');
    })

    step('get2', function () {
        http.get('http://httpbin.org/get');
    })

    step('get3', function () {
        http.get('http://httpbin.org/get');
    })
};
```

# pts.sleep

最近更新时间：2025-01-17 15:31:42

**在脚本执行过程中，暂停指定的时间长度。**

```
sleep(seconds: number);
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| seconds | number | 暂停执行的时间长度，单位为秒。 |

## 样例

**暂停10毫秒：**

```
import http from 'pts/http';
import { check, sleep } from 'pts';

export default function () {
  //const resp1 = http.get('http://httpbin.org/get');

  // 暂停10毫秒
  sleep(0.01);

  const resp1 = http.get('http://httpbin.org/get');
}
```

# Metadata

最近更新时间：2024-12-30 16:05:03

Metadata 是包含了压测任务元数据的 Object，通过调用 metadata() 方法时返回。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| userID | string | 用户 Uin |
| appID | string | 账户 AppID |
| scenarioID | string | 压测场景 ID |
| region | string | 压测任务所在地域 |
| jobID | string | 压测任务 ID |

## 样例

### 调用 metadata() 获取 Metadata 对象：

```
import { metadata } from 'pts';

export default function () {
    // md 为 Metadata 的 interface 对象
    let md = metadata();
    console.log(md.userID); // 123456
    console.log(md.appID); // 123456
    console.log(md.scenarioID); // scenario-xxxxxxxx
    console.log(md.region); // ap-guangzhou
    console.log(md.jobID); // job-xxxxxxxx
}
```

# pts/dataset
# 模块概览

最近更新时间：2024-12-30 16:05:03

JavaScript API 中的 pts/dataset 模块实现了参数文件相关的逻辑。

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| add(fileName, values) | void | 增加新的参数文件，并设置给定的参数值。 |
| forEach(fileName, callback) | void | 遍历指定参数文件，支持修改和删除。 |
| get(key) | string | 根据给定的列名，获取参数数据值。 |
| random(fileName) | Record<string, any> | 随机获取指定参数文件中的某行数据。 |

## 对象

| 对象 | 描述 |
| --- | --- |
| Item | 参数文件中的一行数据，以及其是否在本次脚本中是否被删除的标记，在 forEach 方法的回调函数中使用。 |

# dataset.add

最近更新时间：2024-12-30 16:05:03

dataset.add 方法会新增一个参数文件，并添加给定的参数，主要在 setup function 中使用。

```
add(filename: string, values: Record<string, any>[]): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| filename | string | 新增参数文件的名称 |
| values | Record<string, any>[] | 需要添加到参数文件的给定参数 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 使用样例

### 增加新的参数文件和参数：

```
import dataset from 'pts/dataset';

export function setup () {
  dataset.add("user", [
    {"id": 1, "name": "zhangsan", "age": 1},
    {"id": 2, "name": "lisi", "age": 2},
 ]);
}
```

# dataset.forEach

最近更新时间：2024-12-30 16:05:03

在脚本执行的过程中，dataset.forEach 能够遍历给定的参数文件，支持修改和删除的操作，主要在 setup function 中使用。

```
forEach(fileName: string, callback: (item: Item, i?: number) => void):
void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| fileName | string | 遍历的参数文件名 |
| callback | function | 回调函数；item 为 Item 类型，代表参数文件中的一行数据；i 为数字类型，代表该行数据的行号 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

**遍历参数文件，并进行修改和删除：**

```
import dataset from 'pts/dataset';

export function setup() {
  // 遍历名为 'test.csv' 的参数文件
  dataset.forEach('test.csv', (item) => {
    // 将数据行 item 中键名为 'key5' 的数据值改为 '555'
    item.data.key5 = '555';

    // 若数据行 item 中键名为 'key1' 的数据值为 '1'，则将其标记为删除，在本脚本执
行过程中不会被使用
    if (item.data.key1 === '1') {
      item.delete();
    }
  });
```

```
}
```

**遍历参数文件，在回调函数中包含 i 参数：**

```
import dataset from 'pts/dataset';

export function setup() {
  // 遍历名为 'test.csv' 的参数文件
  dataset.forEach('test.csv', (item, i) => {
    // 输出
    // 0: {"name":"1","value":"111"}
    // 1: {"name":"2","value":"222"}
    // 2: {"name":"3","value":"333"}
    console.log(i, ': ', JSON.stringify(item.data));
  });
}
```

# dataset.get

最近更新时间：2024-12-30 16:05:03

dataset.get 能够根据给定的列名，获取参数文件中的数据值，主要在主函数中使用。

```
get(key: string): string
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 列名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 数据值 |

## 使用样例

### 获取参数文件中的数据值：

```
import dataset from 'pts/dataset';

export default function () {
  // 获取 dataset 中列名为 'key1' 的数据值，假设值为 'value1'
  const value = dataset.get('key1');
  // 输出 'key1 => value1'
  console.log(`key1 => ${value}`);
}
```

# dataset.random

最近更新时间：2024-12-30 16:05:03

dataset.random 能够随机获取参数文件的一行，主要在主函数中使用。

```
random(filename: string): Record<string, any>
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| fileName | string | 获取的参数文件名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| Record<string, any> | 随机获取的一行参数 |

## 使用样例

### 随机获取某一参数文件的一行：

```
import dataset from 'pts/dataset';

export default function () {
  // 参数文件 'test.csv'
  // name,value
  // 1,111
  // 2,222
  // 3,333
  const record = dataset.random('test.csv');
  // 输出 '{"name":"2","value":"222"}'
  console.log(JSON.stringify(record));
}
```

# Item
# Item 概览

最近更新时间：2024-12-30 16:05:03

Item 代表了参数文件中的一行数据，以及其是否在本次脚本中是否被删除的标记，在 dataset.forEach 方法中使用到。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| data | Record<string, string> | 该行参数的列名和数据值 |

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| delete() | void | 标记该行数据为“删除”，即在当前脚本执行过程中不会使用 |

## 使用样例

**遍历参数文件，并进行修改和删除：**

```
import dataset from 'pts/dataset';

export function setup() {
  // 遍历名为 'test.csv' 的参数文件
  dataset.forEach('test.csv', (item) => {
    // 将数据行 item 中键名为 'key5' 的数据值改为 '555'
    item.data.key5 = '555';

    // 若数据行 item 中键名为 'key1' 的数据值为 '1'，则将其标记为删除，在本脚本执行过程中不会被使用
    if (item.data.key1 === '1') {
      item.delete();
    }
  });
}
```

# Item.delete

最近更新时间：2024-12-30 16:05:03

Item.delete 将该 Item 代表的数据行标记为删除，并在当前脚本的执行中不被使用。

```
datele(): void
```

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 使用样例

**将参数文件中的某行数据标记为删除：**

```
import dataset from 'pts/dataset';

export function setup() {
  // 遍历名为 'test.csv' 的参数文件
  dataset.forEach('test.csv', (item) => {
    // 若数据行 item 中键名为 'key1' 的数据值为 '1'，则将其标记为删除，在本脚本执
行过程中不会被使用
    if (item.data.key1 === '1') {
      item.delete();
    }
  });
}

export default function() {
  // 在脚本执行过程中不会访问到 'key1' 列为 '1' 的数据行
  const record = dataset.random('test.csv');
  console.log(JSON.stringify(record));
}
```

# pts/grpc
# 模块概览

最近更新时间：2024-12-30 16:05:03

Javascript API 中的 pts/grpc 模块实现了 gRPC 相关的功能。

## 对象

| 对象 | 描述 |
| --- | --- |
| Client | gRPC 客户端 |
| DialOption | Client.connect 建立连接过程中的可选配置 |
| InvokeOption | Client.invoke 执行方法过程中的可选配置 |
| Response | Client.invoke 执行方法的返回结果 |

# Client
# Client 概览

最近更新时间：2024-12-30 16:05:03

grpc.Client 代表了 gRPC 客户端，能够和 gRPC 服务端进行交互。

### 构造函数

```
new Client(): Client
```

### 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| load(importPaths, ...filenames) | void | 加载 pb 文件 |
| connect(target, option?) | void | 建立连接 |
| invoke(method, request, option?) | Response | 执行方法 |
| close() | void | 关闭连接 |

### 样例

**创建 gRPC Client 并使用：**

```
import grpc from 'pts/grpc';

// 创建新的 grpc Client
const client = new grpc.Client();

// 加载协议文件根目录中的 addsvc.proto
client.load([], 'addsvc.proto');

export default () => {
  // 建立连接
  client.connect('grpcb.in:9000', { insecure: true });

  // 调用方法
```

```
  const rsp = client.invoke('addsvc.Add/Sum', {
    a: 1,
    b: 2,
  });
  console.log(rsp.data.v); // 3

  // 关闭连接
  client.close();
};
```

# Client.load

最近更新时间：2024-12-30 16:05:03

Client.load 用于加载 pb 文件。

```
load(importPaths: string[], ...filenames: string[]): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| importPaths | string[] | 用于搜索 proto 源文件的 import 语句中引用的依赖项路径；若没有提供导入路径，则当前目录被假定为唯一的导入路径 |
| ...filenames | string[] | pb 文件名列表，支持单个文件名的调用 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

### 加载协议文件根目录中的文件：

```
import grpc from 'pts/grpc';

const client = new grpc.Client();

// 加载协议文件根目录中的 addsvc.proto
client.load([], 'addsvc.proto');
```

### 加载协议文件某个目录中多个文件：

```
import grpc from 'pts/grpc';

const client = new grpc.Client();

// 加载中协议文件 dirName 目录中的 addsvc.proto 和 example.proto
client.load(['dirName'], 'addsvc.proto', 'example.proto');
```

# Client.connect

最近更新时间：2024-12-30 16:05:03

Client.connect 用于建立连接。

```
connect(target: string, option?: DialOption): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| target | string | 连接建立的目标地址 |
| option? | DialOption | 可选，DialOption 对象，建立连接的可选配置 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

**调用方法建立连接：**

```
import grpc from 'pts/grpc';

// 创建新的 grpc Client
const client = new grpc.Client();

// 加载协议文件根目录中的 addsvc.proto
client.load([], 'addsvc.proto');

export default () => {
  // 建立连接
  client.connect('grpcb.in:9000', { insecure: true });

  // 关闭连接
  client.close();
};
```

# Client.invoke

最近更新时间：2024-12-30 16:05:03

Client.invoke 用于执行方法。

```
invoke(method: string, request: any, option?: InvokeOption): Response
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| method | string | 完整的 Path 路径 |
| request | any | 业务的请求内容 |
| option? | InvokeOption | 可选，InvokeOption 对象，执行方法的选项 |

## 返回

| 类型 | 描述 |
| --- | --- |
| Response | 执行结果 |

## 样例

### 调用方法进行指定 method 的执行：

```
import grpc from 'pts/grpc';

// 创建新的 grpc Client
const client = new grpc.Client();

// 加载协议文件根目录中的 addsvc.proto
client.load([], 'addsvc.proto');

export default () => {
  // 建立连接
  client.connect('grpcb.in:9000', { insecure: true });

  // 调用方法
  const rsp = client.invoke('addsvc.Add/Sum', {
```

```
    a: 1,
    b: 2,
  });
  console.log(rsp.data.v); // 3

  // 关闭连接
  client.close();
};
```

# Client.close

最近更新时间：2025-01-03 14:17:23

Client.close 关闭连接。

```
close(): void
```

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

**调用方法关闭连接：**

```
import grpc from 'pts/grpc';

// 创建新的 grpc Client
const client = new grpc.Client();

// 加载协议文件根目录中的 addsvc.proto
client.load([], 'addsvc.proto');

export default () => {
  // 建立连接
  client.connect('grpcb.in:9000', { insecure: true });

  // 调用方法
  const rsp = client.invoke('addsvc.Add/Sum', {
    a: 1,
    b: 2,
  });
  console.log(rsp.data.v); // 3

  // 关闭连接
  client.close();
};
```

# DialOption

最近更新时间：2024-12-30 16:05:04

DialOption 是 Client.connect 建立连接的过程中的可选项。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| insecure? | boolean | 是否不加密，true 不加密，false 加密 |
| timeout? | number | 超时时间，单位为毫秒 |

## 样例

### 使用 DialOption 设置连接的建立：

```
import grpc from 'pts/grpc';

// 创建新的 grpc Client
const client = new grpc.Client();

// 加载协议文件根目录中的 addsvc.proto
client.load([], 'addsvc.proto');

export default () => {
  // 建立连接
  client.connect('grpcb.in:9000', { insecure: true, timeout: 3000 });

  // 关闭连接
  client.close();
};
```

# InvokeOption

最近更新时间：2024-12-30 16:05:04

InvokeOption 是 Client.invoke 执行方法过程中的可选配置。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| headers? | Record<string, string[]> | 请求头 |
| timeout? | number | 超时时间，单位为毫秒 |

## 样例

### 使用 InvokeOption 设置执行的可选配置：

```
export default () => {
  // 建立连接
  client.connect('grpcb.in:9000', { insecure: true });

  // 调用方法
  const rsp = client.invoke(
    'addsvc.Add/Sum',
    { a: 1, b: 2 },
    // 配置 InvokeOption
    {
      headers: {
        example: ['a', 'b'],
      },
      timeout: 5000,
    },
  );
  console.log(rsp.data.v); // 3

  // 关闭连接
  client.close();
};
```

# Response

最近更新时间：2024-12-30 16:05:04

Response 是 Client.invoke 执行方法的返回对象。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| code | number | 状态码 |
| data | any | 业务返回数据 |
| headers | Record<string, string[]> | 请求 Header 元数据 |
| message | string | 错误信息 |
| trailers | Record<string, string[]> | 请求 Trailer 元数据 |

## 样例

### 调用 Client.invoke 获得请求返回对象：

```
import grpc from 'pts/grpc';

// 创建新的 grpc Client
const client = new grpc.Client();

// 加载协议文件根目录中的 addsvc.proto
client.load([], 'addsvc.proto');

export default () => {
  // 建立连接
  client.connect('grpcb.in:9000', { insecure: true });

  // 调用方法，获得请求的返回对象 rsp
  const rsp = client.invoke('addsvc.Add/Sum', {
    a: 1,
    b: 2,
  });
  console.log(rsp.data.v); // 3

  // 关闭连接
  client.close();
```

```
};
```

# pts/jsonpath
# 模块概览

最近更新时间：2024-12-30 16:05:04

JavaScript API 中的 pts/jsonpath 实现了对 JSON 序列化字符串进行操作的部分功能。

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| get(json, path) | number、string、boolean 或 object | 根据 path 获取 json 字符串中的值 |

# jsonpath.get

最近更新时间：2024-12-30 16:05:04

**jsonpath.get 用于从 JSON 字符串中获取对应路径的值。**

```
get(json: string, path: string): string | number | boolean | object
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| json | string | JSON 字符串 |
| path | string | 取值路径 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string、number、boolean 或 object | 取值得到的数据 |

## 样例

**获取给定路径的值：**

```
import jsonpath from 'pts/jsonpath';

export default function () {
  const json = JSON.stringify({
    name: { first: 'Tom', last: 'Anderson' },
    age: 37,
    children: ['Sara', 'Alex', 'Jack'],
    'fav.movie': 'Deer Hunter',
    friends: [
      { first: 'Dale', last: 'Murphy', age: 44, nets: ['ig', 'fb', 'tw']
},
      { first: 'Roger', last: 'Craig', age: 68, nets: ['fb', 'tw'] },
      { first: 'Jane', last: 'Murphy', age: 47, nets: ['ig', 'tw'] },
    ],
  });

  console.log(jsonpath.get(json, 'name.last')); // Anderson
```

```
  console.log(jsonpath.get(json, 'age')); // 37
  console.log(jsonpath.get(json, 'children')); // Sara,Alex,Jack
  console.log(jsonpath.get(json, 'children[*]')); // Sara,Alex,Jack
  console.log(jsonpath.get(json, 'children.[0]')); // Sara
  console.log(jsonpath.get(json, 'children[1:2]')); // Alex,Jack
  console.log(jsonpath.get(json, 'friends[:].first')); //
Dale,Roger,Jane
  console.log(jsonpath.get(json, 'friends[1].last')); // Craig
  console.log(jsonpath.get(json, 'friends[?(@.age > 45)].last')); //
Craig,Murphy
  console.log(jsonpath.get(json, 'friends[?(@.first =~ /D.*e/)].last'));
// Murphy
}
```

# pts/protobuf
# 模块概览

最近更新时间：2024-12-30 16:05:04

JavaScript API 中的 pts/protobuf 模块实现了 protobuf 相关的功能。

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| load(importPaths, ...fileNames) | void | 加载 pb 文件 |
| marshal(message, value) | ArrayBuffer | 进行 pb 序列化 |
| unmarshal(message, data, filename?) | any | 进行 pb 反序列化 |

## 样例

```
import protobuf from 'pts/protobuf';

// 加载协议文件根目录中的 demo.proto
protobuf.load([], 'demo.proto');

// 加载中协议文件 dirName 目录中的 demo.proto
// protobuf.load(['dirName'], 'demo.proto');

export default function () {
  // 调用 marshal 进行序列化
  const data = protobuf.marshal('xxxx.xxx.demo.stSayHelloReq', { msg:
'pts' });
  console.log(data); // [object ArrayBuffer]

  // 调用 unmarshal 进行反序列化
  const value = protobuf.unmarshal('xxxx.xxx.demo.stSayHelloReq', data);
  console.log(JSON.stringify(value)); // {"msg":"pts"}
}
```

# protobuf.load

最近更新时间：2024-12-30 16:05:04

rotobuf.load 用于加载 pb 文件。

```
load(importPaths: string[], ...filenames: string[]): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| importPaths | string[] | 用于搜索 proto 源文件的 import 语句中引用的依赖项路径；若没有提供导入路径，则当前目录被假定为唯一的导入路径。 |
| ...filenames | string[] | pb 文件名列表，支持单个文件名的调用。 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

### 加载协议文件根目录中的文件：

```
import protobuf from 'pts/protobuf';

// 加载协议文件根目录中的 addsvc.proto
protobuf.load([], 'addsvc.proto');
```

### 加载协议文件某个目录中多个文件：

```
import protobuf from 'pts/protobuf';

// 加载中协议文件 dirName 目录中的 addsvc.proto 和 example.proto
protobuf.load(['dirName'], 'addsvc.proto', 'example.proto');
```

# protobuf.marshal

最近更新时间：2024-12-30 16:05:04

**protobuf.marshal 用于进行 pb 序列化。**

```
marshal(message: string, value: any): ArrayBuffer
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| message | string | 结构体名 |
| value | any | JSON 化的请求体 |

## 返回

| 类型 | 描述 |
| --- | --- |
| ArrayBuffer | 序列化得到的二进制请求体 |

## 样例

**调用方法进行 pb 序列化：**

```
import protobuf from 'pts/protobuf';

// 加载协议文件根目录中的 demo.proto
protobuf.load([], 'demo.proto');

// 加载中协议文件 dirName 目录中的 demo.proto
// protobuf.load(['dirName'], 'demo.proto');

export default function () {
  // 调用 marshal 进行序列化
  const data = protobuf.marshal('xxxx.xxx.demo.stSayHelloReq', { msg:
'pts' });
  console.log(data); // [object ArrayBuffer]

  // 调用 unmarshal 进行反序列化
  const value = protobuf.unmarshal('xxxx.xxx.demo.stSayHelloReq', data);
  console.log(JSON.stringify(value)); // {"msg":"pts"}
```

```
}
```

# protobuf.unmarshal

最近更新时间：2024-12-30 16:05:04

**protobuf.marshal 用于进行 pb 反序列化。**

```
unmarshal(message: string, data: ArrayBuffer, filename?: string): any
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| message | string | 结构体名 |
| data | ArrayBuffer | 二进制请求体 |
| filename? | string | 可选，参数文件名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| any | 反序列化得到的结果 |

## 样例

**调用方法进行 pb 反序列化：**

```
import protobuf from 'pts/protobuf';

// 加载协议文件根目录中的 demo.proto
protobuf.load([], 'demo.proto');

// 加载中协议文件 dirName 目录中的 demo.proto
// protobuf.load(['dirName'], 'demo.proto');

export default function () {
  // 调用 marshal 进行序列化
  const data = protobuf.marshal('xxxx.xxx.demo.stSayHelloReq', { msg:
'pts' });
  console.log(data); // [object ArrayBuffer]

  // 调用 unmarshal 进行反序列化
```

```
    const value = protobuf.unmarshal('xxxx.xxx.demo.stSayHelloReq', data);
    console.log(JSON.stringify(value)); // {"msg":"pts"}
}
```

# pts/sql
# 模块概览

最近更新时间：2025-01-03 14:17:23

API 中的 pts/sql 模块实现了 sql 相关的功能。

## 对象

| 对象 | 描述 |
| --- | --- |
| Database | 数据库实例，能够与数据库进行交互 |
| Result | 调用 Database.exec 返回的结果 |

# Database
# Database 概览

最近更新时间：2024-11-29 15:25:42

Database 代表了数据库实例，能够与数据库进行交互。

## 创建

通过 new 进行数据库实例的创建，如下所示：

```
new (driverName: string, dataSourceName: string): Database
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| driverName | string | 驱动名，目前支持 'mysql' |
| dataSourceName | string | 数据源 |

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| exec(query, ...args) | Result | 执行查询但不返回行数据 |
| query(query, ...args) | Record<string, any>[] | 执行查询并返回行结果，通常是 SELECT |

## 样例

创建 Database 实例并进行交互：

```
import sql from 'pts/sql';

// 通过 new 创建数据库实例
const db = new sql.Database(sql.MySQL,
"user:passwd@tcp(ip:port)/database")

export default function () {
```

```
  let result = db.exec("UPDATE user SET age=? WHERE name='zhangsan'",
Math.floor(Math.random() * 100));
  console.log(JSON.stringify(result)); //
{"lastInsertId":0,"rowsAffected":1}

  let rows = db.query("SELECT * FROM user");
  console.log(JSON.stringify(rows)); //
[{"id":1,"name":"zhangsan","age":23},{"id":2,"name":"lisi","age":2}]
}
```

# Database.exec

最近更新时间：2024-11-29 15:25:42

**Database.exec 用于执行查询，但不返回数据库数据。**

```
exec(query: string, ...args: any[]): Result
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| query | string | 查询语句 |
| ...args | any[] | 用于查询中的占位符参数 |

## 返回

| 类型 | 描述 |
| --- | --- |
| Result | 查询返回结果 |

## 样例

**使用 exec 进行数据库查询：**

```
import sql from 'pts/sql';

// 通过 new 创建数据库实例
const db = new sql.Database(sql.MySQL,
"user:passwd@tcp(ip:port)/database")

export default function () {
  let result = db.exec("UPDATE user SET age=? WHERE name='zhangsan'",
Math.floor(Math.random() * 100));
  console.log(JSON.stringify(result)); //
{"lastInsertId":0,"rowsAffected":1}
}
```

# Database.query

最近更新时间：2024-11-29 15:25:42

**Database.query 用于执行查询，并返回数据库数据。**

```
query(query: string, ...args: any[]): Record<string, any>[]
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| query | string | 查询语句 |
| ...args | any[] | 用于查询中的占位符参数 |

## 返回

| 类型 | 描述 |
| --- | --- |
| Record<string, any>[] | 查询返回结果，包含数据库数据 |

## 样例

**使用 query 进行数据库查询：**

```
import sql from 'pts/sql';

// 通过 new 创建数据库实例
const db = new sql.Database(sql.MySQL,
"user:passwd@tcp(ip:port)/database")

export default function () {
  let rows = db.query("SELECT * FROM user");

  // [{"id":1,"name":"zhangsan","age":23},
{"id":2,"name":"lisi","age":2}]
  console.log(JSON.stringify(rows));
}
```

# Result

最近更新时间：2024-12-30 16:05:04

Result 是 Database.exec 返回的结果。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| lastInsertId? | number | 返回数据库为响应命令而生成的整数；通常这将来自插入新行时的“自动增量”列 |
| rowsAffected? | number | 返回受更新、插入或删除影响的行数 |

## 样例

### 使用 exec 进行数据库查询返回 Result：

```
import sql from 'pts/sql';

// 通过 new 创建数据库实例
const db = new sql.Database(sql.MySQL,
"user:passwd@tcp(ip:port)/database")

export default function () {
  let result = db.exec("UPDATE user SET age=? WHERE name='zhangsan'",
Math.floor(Math.random() * 100));
  console.log(JSON.stringify(result)); //
{"lastInsertId":0,"rowsAffected":1}
}
```

# pts/url
# 模块概览

最近更新时间：2024-11-29 15:25:42

JavaScript API 中的 pts/url 模块实现了 url 相关的功能。

## 对象

| 对象 | 描述 |
| --- | --- |
| URL | 用于 URL 相关操作 |
| URLSearchParams | 用于处理 URL 的查询字符串 |

# URL
# URL 概览

最近更新时间：2024-12-30 16:05:04

url.URL 能够用于 URL 的相关操作。

### 构造函数

通过 new 进行对象实例创建，如下所示：

```
new URL(url: string, base?: string | URL): URL
```

### 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| url | string | 统一资源定位符 |
| base? | string 或 URL | 统一资源定位符中的协议和主机部分，若 url 中不包含，则对其进行覆盖 |

### 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| hash() | string | 获取网址的片段部分 |
| setHash(hash) | void | 设置网址的片段部分 |
| host() | string | 获取网址的主机部分 |
| setHost(host) | void | 设置网址的部分 |
| hostname() | string | 获取网址的主机名部分 |
| setHostname(hostname) | void | 设置网址的主机名部分 |
| setHostname(hostname) | string | 获取序列化的网址 |
| setHref(href) | void | 设置序列化的网址 |

| | | |
|---|---|---|
| origin() | string | 获取网址的源的只读的序列化 |
| password() | string | 获取网址的密码部分 |
| setPassword(password) | void | 设置网址的密码部分 |
| pathname() | string | 获取网址的路径部分 |
| setPathname(pathname) | void | 设置网址的路径部分 |
| port() | string | 获取网址的端口部分 |
| setPort(port) | void | 设置网址的端口部分 |
| protocol() | string | 获取网址的协议部分 |
| protocol() | void | 设置网址的协议部分 |
| search() | string | 获取网址的序列化的查询部分 |
| setSearch(search) | void | 设置网址的序列化的查询部分 |
| searchParams() | URLSearchParams | 获取表示网址查询参数的 URLSearchParams 对象 |
| username() | string | 获取网址的用户名部分 |
| setUsername(username) | void | 设置网址的用户名部分 |
| toJSON() | string | 返回序列化的网址，当 URL 对象用 JSON.stringify() 序列化时，会自动调用此方法 |
| toString() | string | 返回序列化的网址 |

## 样例

### 创建 URL 对象，不使用 base 参数：

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?name=xxx&age=18#worker';
```

```
  const u0 = new url.URL(u)
  console.log(u0.toString()); //
http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker
}
```

**创建 URL 对象，使用 base 参数：**

```
import url from 'pts/url';

export default function () {
  const u = '/test/index.html?name=xxx&age=18#worker';

  const u0 = new url.URL(u, 'https://console.cloud.tencent.com:8080')
  console.log(u0.toString()); //
https://console.cloud.tencent.com:8080/test/index.html?
name=xxx&age=18#worker
}
```

**使用 URL 对象进行相关操作：**

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const u1 = new url.URL(u);
  console.log('hash1 ', u1.hash()); // #worker
  u1.setHash('hash');
  console.log('hash2 ', u1.hash()); // #hash

  const u2 = new url.URL(u);
  console.log('host1 ', u2.host()); // www.example.com:8080
  u2.setHost('host');
  console.log('host2 ', u2.host()); // host

  const u3 = new url.URL(u);
  console.log('hostname1 ', u3.hostname()); // www.example.com
  u3.setHostname('hostname');
  console.log('hostname2 ', u3.hostname()); // hostname
```

```
  const u4 = new url.URL(u);
  console.log('href1 ', u4.href()); //
http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker
  u4.setHref('https://console.cloud.tencent.com');
  console.log('href2 ', u4.href()); //
https://console.cloud.tencent.com/

  const u5 = new url.URL(u);
  console.log('origin1 ', u5.origin()); // http://www.example.com:8080

  const u6 = new url.URL(u);
  console.log('pathname1 ', u6.pathname()); // /test/index.html
  u6.setPathname('pathname');
  console.log('pathname2 ', u6.pathname()); // pathname

  const u7 = new url.URL(u);
  console.log('port1 ', u7.port()); // 8080
  u7.setPort('80');
  console.log('port2 ', u7.port()); // 80

  const u8 = new url.URL(u);
  console.log('protocol1 ', u8.protocol()); // http:
  u8.setProtocol('protocol');
  console.log('protocol2 ', u8.protocol()); // protocol:

  const u9 = new url.URL(u);
  console.log('search1 ', u9.search()); // ?age=18&name=xxx
  u9.setSearch('search');
  console.log('search2 ', u9.search()); // ?search=

  const u10 = new url.URL(u);
  console.log('searchParams1 ', u10.searchParams()); // [object Object]

  const u11 = new url.URL(u);
  console.log('username1 ', u11.username()); // user
  u11.setUsername('username');
  console.log('username2 ', u11.username()); // username

  const u12 = new url.URL(u);
  console.log('password1 ', u12.password()); // pass
  u12.setPassword('password');
  console.log('password2 ', u12.password()); // password
```

```
  const u13 = new url.URL(u);
  console.log('toJSON1 ', u13.toJSON()); //
http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker

  const u14 = new url.URL(u);
  console.log('toString1 ', u14.toString()); //
http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker
}
```

# URL.hash

最近更新时间：2024-12-30 16:05:04

URL.hash 用于获取网址的片段部分。

```
hash(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 网址的片段部分 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log('hash1 ', uu.hash()); // #worker
}
```

# URL.setHash

最近更新时间：2024-12-30 16:05:04

URL.setHash 用于设置网址的片段部分。

```
setHash(hash: string): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| hash | string | 要设置的网址片段部分 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.hash()); // #worker
  uu.setHash('hash');
  console.log('hash2 ', uu.hash()); // #hash
}
```

# URL.host

最近更新时间：2024-12-30 16:05:04

URL.host 用于获取网址的主机部分。

```
host(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 网址的主机部分 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.host()); // www.example.com:8080
}
```

# URL.setHost

最近更新时间：2024-12-30 16:05:04

URL.setHost 用于网址的主机部分。

```
setHost(host: string): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| host | string | 要设置的网址主机部分 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.host()); // www.example.com:8080
  uu.setHost('host');
  console.log(uu.host()); // host
}
```

# URL.hostname

最近更新时间：2024-12-30 16:05:04

URL.hostname 用于获取网址的主机名部分。

```
hostname(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 网址的主机名部分 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.hostname()); // www.example.com
}
```

# URL.setHostname

最近更新时间：2024-12-30 16:05:04

URL.setHostname 用于设置网址的部分。

```
setHostname(hostname: string): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| hostname | string | 要设置的网址主机名部分 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.hostname()); // www.example.com
  uu.setHostname('hostname');
  console.log(uu.hostname()); // hostname
}
```

# URL.href

最近更新时间：2024-12-30 16:05:04

URL.href 用于获取序列化的网址。

```
href(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 序列化的网址 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.href()); //
http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker
}
```

# URL.setHref

最近更新时间：2024-12-30 16:05:04

URL.setHref 用于设置序列化的网址。

```
setHref(href: string): void新增
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| href | string | 要设置的序列化网址 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.href()); //
http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker
  uu.setHref('https://console.cloud.tencent.com');
  console.log(uu.href()); // https://console.cloud.tencent.com/
}
```

# URL.origin

最近更新时间：2024-12-30 16:05:04

URL.origin 用于获取网址的源的只读的序列化。

```
origin(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 网址的源的只读的序列化 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.origin()); // http://www.example.com:8080
}
```

# URL.pathname

最近更新时间：2024-12-30 16:05:04

URL.pathname 用于获取网址的路径部分。

```
pathname(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 网址的路径部分 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.pathname()); // /test/index.html
}
```

# URL.setPathname

最近更新时间：2024-12-30 16:05:04

URL.setPathname 用于设置网址的路径部分。

```
setPathname(pathname: string): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| pathname | string | 要设置网址的路径部分 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.pathname()); // /test/index.html
  uu.setPathname('pathname');
  console.log(uu.pathname()); // pathname
}
```

# URL.password

最近更新时间：2024-12-30 16:05:04

URL.password 用于获取网址的密码部分。

```
password(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 网址的密码部分 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.password()); // pass
}
```

# URL.setPassword

最近更新时间：2024-12-30 16:05:04

URL.setPassword 用于设置网址的密码。

```
setPassword(password: string): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| password | string | 要设置网址的密码 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.password()); // pass
  uu.setPassword('password');
  console.log(uu.password()); // password
}
```

# URL.port

最近更新时间：2024-12-30 16:05:04

URL.port 用于获取网址的端口部分。

```
port(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 网址的端口部分 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.port()); // 8080
}
```

# URL.setPort

最近更新时间：2025-01-03 14:17:23

URL.setPort 用于设置网址的端口部分。

```
setPort(port: string): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| port | string | 要设置网址的端口部分 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.port()); // 8080
  uu.setPort('80');
  console.log(uu.port()); // 80
}
```

# URL.protocol

最近更新时间：2024-12-30 16:05:04

URL.protocol 用于获取网址的协议部分。

```
protocol(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 网址的协议部分 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.protocol()); // http:
}
```

# URL.setProtocol

最近更新时间：2024-12-30 16:05:04

URL.setProtocol 用于设置网址的协议部分。

```
setProtocol(protocol: string): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| protocol | string | 要设置网址的协议部分 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.protocol()); // http:
  uu.setProtocol('protocol');
  console.log(uu.protocol()); // protocol:
}
```

# URL.search

最近更新时间：2024-12-30 16:05:04

URL.search 用于获取网址的序列化的查询部分。

```
search(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 网址的序列化的查询部分 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.search()); // ?age=18&name=xxx
}
```

# URL.setSearch

最近更新时间：2024-12-30 16:05:04

URL.setSearch 用于设置网址的序列化的查询部分。

```
setSearch(search: string): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| search | string | 要设置网址的序列化的查询部分 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.search()); // ?age=18&name=xxx
  uu.setSearch('search=1');
  console.log(uu.search()); // ?search=1
}
```

# URL.searchParams

最近更新时间：2024-12-30 16:05:04

URL.searchParams 用于获取表示网址查询参数的 URLSearchParams 对象。

```
searchParams(): URLSearchParams
```

## 返回

| 类型 | 描述 |
| --- | --- |
| URLSearchParams | 网址查询参数的 URLSearchParams 对象 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.searchParams()); // [object Object]
}
```

# URL.username

最近更新时间：2024-12-30 16:05:04

URL.username 用于获取网址的用户名部分。

```
username(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 网址的用户名部分 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.username()); // user
}
```

# URL.setUsername

最近更新时间：2024-12-30 16:05:04

URL.setUsername 用于设置网址的用户名。

```
setUsername(username: string): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| username | string | 要设置网址的用户名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.username()); // user
  uu.setUsername('username');
  console.log(uu.username()); // username
}
```

# URL.toJSON

最近更新时间：2024-12-30 16:05:04

URL.toJSON 用于获取序列化的网址，当 URL 对象用 JSON.stringify() 序列化时，会自动调用此方法。

```
toJSON(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 序列化的网址 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.toJSON()); //
http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker
}
```

# URL.toString

最近更新时间：2024-12-30 16:05:04

URL.toString 用于返回序列化的网址。

```
toString(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 序列化的网址 |

## 样例

```
import url from 'pts/url';

export default function () {
  const u = 'http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker';

  const uu = new url.URL(u);
  console.log(uu.toString()); //
http://user:pass@www.example.com:8080/test/index.html?
name=xxx&age=18#worker
}
```

# URLSearchParams
# URLSearchParams 概览

最近更新时间：2024-12-30 16:05:04

url.URLSearchParams 定义了一些实用的方法处理 URL 的查询字符串。

### 构造函数

通过 new 进行对象实例创建，如下所示：

```
new URLSearchParams(params: string): URLSearchParams
```

### 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| params | string | 查询字符串 |

### 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| append(key, value) | void | 增加指定的键/值对作为搜索参数 |
| delete(key) | void | 删除搜索参数列表指定的键及其对应的值 |
| entries() | string[][] | 获取查询参数中所有的键值对 |
| forEach(callback) | void | 通过回调函数遍历 URLSearchParams 上的所有键值对 |
| get(key) | null 或 string | 获取指定搜索参数对应的第一个值 |
| getAll(key) | string[] | 获取指定搜索参数对应的所有值 |
| has() | boolean | 判断是否存在该键对应的搜索参数 |
| keys() | string[] | 获取搜索参数中所有的键名 |
| set(key, value) | void | 设置新的搜索参数，如果原来有该键值则覆盖 |

| | | |
|---|---|---|
| toString() | string | 获取搜索参数组成的字符串，可直接使用于 URL |
| values() | string[] | 获取搜索参数中所有的值 |

## 样例

**构造实例并进行操作：**

```
import url from 'pts/url';

export default function() {
  const params = new url.URLSearchParams('key1=value1&key2=value2');

  // 调用 append
  params.append('key3', 'value3');
  console.log(params.toString()); // key1=value1&key2=value2&key3=value3

  // 调用 delete
  params.delete('key3');
  console.log(params.toString()); // key1=value1&key2=value2

  // 调用 entries
  // key1, value1
  // key2, value2
  for(var pair of params.entries()) {
    console.log(pair[0]+ ', '+ pair[1]);
  }

  // 调用 forEach
  // value1, key1, [object Object]
  // value2, key2, [object Object]
  params.forEach(function(value, key, searchParams) {
    console.log(value, ', ', key, ', ', searchParams);
  });

  // 调用 get
  console.log(params.get('key1')); // value1
  console.log(params.get('key3')); // null

  // 调用 getAll
  params.append('key1', 1);
  console.log(params.getAll('key1')); // value1,1
```

```
  // 调用 has
  console.log(params.has('key1')); // true
  console.log(params.has('key3')); // false

  // 调用 keys
  console.log(params.keys()); // key1,key2

  // 调用 set
  params.set('key3', 'value3');
  params.set('key1', 'value1');
  // key1, value1
  // key2, value2
  // key3, value3
  for(var pair of params.entries()) {
    console.log(pair[0]+ ', '+ pair[1]);
  }

  // 调用 toString
  console.log(params.toString()); // key1=value1&key2=value2&key3=value3

  // 调用 values
  console.log(params.values()); // value1,value2,value3
}
```

# URLSearchParams.append

最近更新时间：2024-12-30 16:05:04

URLSearchParams.append 用于增加指定的键值对作为搜索参数。

```
append(key: string, value: string | number): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键 |
| value | string 或 number | 值 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

### 增加指定键值对作为搜索参数：

```
import url from 'pts/url';

export default function() {
  const params = new url.URLSearchParams('key1=value1');

  params.append('key2', 'value2');
  console.log(params.toString()); // key1=value1&key2=value2
}
```

# URLSearchParams.delete

最近更新时间：2024-12-30 16:05:04

URLSearchParams.delete 用于删除搜索参数列表指定的键及其对应的值。

```
delete(key: string): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

**删除搜索参数列表指定的键及其对应的值：**

```
import url from 'pts/url';

export default function() {
  const params = new url.URLSearchParams('key1=value1&key2=value2');

  params.delete('key2');
  console.log(params.toString()); // key1=value1
}
```

# URLSearchParams.entries

最近更新时间：2024-12-30 16:05:04

URLSearchParams.entries 用于获取搜索参数中所有的键值对。

```
entries(): string[][]
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string[][] | 搜索参数中所有的键值对 |

## 样例

**获取搜索参数列表所有的键值对：**

```
import url from 'pts/url';

export default function() {
  const params = new url.URLSearchParams("key1=value1&key2=value2");

  // 键/值对
  // key1, value1
  // key2, value2
  for(var pair of params.entries()) {
    console.log(pair[0]+ ', '+ pair[1]);
  }
}
```

# URLSearchParams.forEach

最近更新时间：2024-12-30 16:05:04

URLSearchParams.forEach 用于通过回调函数遍历 URLSearchParams 上的所有键值对。

```
forEach(callback: (value: string, key: string, parent: URLSearchParams)
=> void): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| callback | function | 回调函数，value 和 key 分别为搜索参数的值和键名，parent 为当前调用 forEach 的 URLSearchParams 实例对象 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

### 通过回调函数遍历 URLSearchParams 上的所有键值对：

```
import url from 'pts/url';

export default function() {
  const params = new url.URLSearchParams('key1=value1&key2=value2');

  // value1, key1, [object Object]
  // value2, key2, [object Object]
  params.forEach(function(value, key, searchParams) {
    console.log(value, ', ', key, ', ', searchParams);
  });
}
```

# URLSearchParams.get

最近更新时间：2024-12-30 16:05:04

URLSearchParams.get 用于获取指定搜索参数对应的第一个值。

```
get(key: string): null | string
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键 |

## 返回

| 类型 | 描述 |
| --- | --- |
| null 或 string | 指定键对应的第一个值，若没有找到则为 null |

## 样例

**获取指定搜索参数对应的第一个值：**

```
import url from 'pts/url';

export default function() {
  const params = new url.URLSearchParams('key1=value1&key2=value2');

  const value1 = params.get('key1');
  console.log(value1); // value1
  const value3 = params.get('key3');
  console.log(value3); // null
}
```

# URLSearchParams.getAll

最近更新时间：2024-12-30 16:05:04

URLSearchParams.getAll 用于获取指定搜索参数对应的所有值。

```
getAll(key: string): string[]
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string[] | 指定搜索参数对应的所有值 |

## 样例

**获取指定搜索参数对应的所有值：**

```
import url from 'pts/url';

export default function() {
  const params = new url.URLSearchParams('key1=value1&key2=value2');

  params.append('key1', 1);
  console.log(params.getAll('key1')); // value1,1
}
```

# URLSearchParams.has

最近更新时间：2024-12-30 16:05:04

URLSearchParams.has 用于判断是否存在该键对应的搜索参数。

```
has(key: string): boolean
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键 |

## 返回

| 类型 | 描述 |
| --- | --- |
| boolean | 是否存在该键对应的搜索参数 |

## 样例

### 判断是否存在该键对应的搜索参数：

```
import url from 'pts/url';

export default function() {
  const params = new url.URLSearchParams('key1=value1&key2=value2');

  console.log(params.has('key1')); // true
  console.log(params.has('key3')); // false
}
```

# URLSearchParams.keys

最近更新时间：2024-12-30 16:05:04

URLSearchParams.keys 用于获取搜索参数中所有的键名。

```
keys(): string[]
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string[] | 搜索参数中所有的键名 |

## 样例

**获取指定搜索参数对应的所有键名：**

```
import url from 'pts/url';

export default function() {
  const params = new url.URLSearchParams('key1=value1&key2=value2');

  console.log(params.keys()); // key1,key2
}
```

# URLSearchParams.set

最近更新时间：2024-12-30 16:05:04

URLSearchParams.set 用于设置新的搜索参数，如果原来有该搜索参数则覆盖其值。

```
set(key: string, value: string | number): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键 |
| value | string 或 number | 值 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回数据 |

## 样例

### 设置新的搜索参数：

```
import url from 'pts/url';

export default function() {
  const params = new url.URLSearchParams('key1=value1&key2=value2');

  params.set('key3', 'value3');
  params.set('key1', 1);

  // key1, 1
  // key2, value2
  // key3, value3
  for(var pair of params.entries()) {
    console.log(pair[0]+ ', '+ pair[1]);
  }
}
```

# URLSearchParams.toString

最近更新时间：2024-12-30 16:05:04

URLSearchParams.toString 用于获取搜索参数组成的字符串，可直接使用于 URL。

```
toString(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 搜索参数组成的字符串 |

## 样例

### 获取搜索参数组成的字符串：

```
import url from 'pts/url';

export default function() {
  const params = new url.URLSearchParams('key1=value1&key2=value2');

  params.set('key3', 'value3');
  console.log(params.toString()); // key1=value1&key2=value2&key3=value3
}
```

# URLSearchParams.values

最近更新时间：2024-12-30 16:05:04

URLSearchParams.values 用于获取搜索参数中所有的值。

```
values(): string[]
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string[] | 搜索参数中所有的值 |

## 样例

**获取指定搜索参数对应的所有值：**

```
import url from 'pts/url';

export default function() {
  const params = new url.URLSearchParams('key1=value1&key2=value2');

  console.log(params.values()); // value1,value2
}
```

# pts/util
# 模块概览

最近更新时间：2025-01-03 14:17:23

base64Encoding(input, encoding?)Javascript API 中的 pts/util 模块实现了部分常用的工具。

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| base64Encoding(input, encoding?) | string | base64 编码 |
| base64Decoding(input, encoding?) | string \| ArrayBuffer | base64 解码 |
| cloudAPISignatureV3(param) | string | 腾讯云 API 签名方法 V3 |
| md5Sum(data) | string | md5 加密 |
| sloginEncrypt(salt, pwd, vcode) | string | QQ slogin 加密 |
| toArrayBuffer(data) | ArrayBuffer | 转换为字节数组 |
| uuid() | string | 全局唯一标识 UUID v4 版本 |

## 对象

| 对象 | 描述 |
| --- | --- |
| CloudAPISignatureV3Param | 用于调用腾讯云 API 签名方法时需要的参数 |

# util.base64Encoding

最近更新时间：2024-12-30 16:05:04

在脚本执行过程中，util.base64Encoding 用于 base64 编码。

```
base64Encoding(input: string | ArrayBuffer, encoding?: "std" | "rawstd"
| "url" | "rawurl"): string
```

## 背景

### base64 不同的编码方式：

- StdEncoding 是标准的 base64 编码，见 RFC 4648 中定义。
- RawStdEncoding 是标准的原始、未填充的 base64 编码，见 RFC 4648 第 3.2 节中定义；与 StdEncoding 相同，但省略了填充字符。
- URLEncoding 是 RFC 4648 中定义的备用 base64 编码，通常用于 URL 和文件名。
- RawURLEncoding 是 RFC 4648 中定义的未填充的替代 base64 编码，通常用于 URL 和文件名；与 URLEncoding 相同，但省略了填充字符。

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| input | string 或 ArrayBuffer | 要编码的字符串或字节数组 |
| encoding（可选） | "std"、"rawstd"、"url" 或 "rawurl" | 可选，代表前文所述的不同编码方式，不填默认为 std |

## 返回

| 类型 | 描述 |
| --- | --- |
| string | base64 编码后的结果 |

## 使用样例

### 不指定 encoding 使用 base64Encoding 方法：

```
import util from 'pts/util';

export default function () {
  // SGVsbG8sIHdvcmxk
```

```
  console.log(util.base64Encoding('Hello, world'));
}
```

**指定 encoding 使用 base64Encoding 方法：**

```
import util from 'pts/util';

export default function () {
  // aHR0cDovL3d3dy5leGFtcGxlLmNvbQ==
  console.log(util.base64Encoding('http://www.example.com', 'url'));
}
```

# util.base64Decoding

最近更新时间：2024-12-30 16:05:04

在脚本执行过程中，util.base64Decoding 用于 base64 解码。

```
base64Decoding(input: string, encoding?: "std" | "rawstd" | "url" |
"rawurl", mode?: "b"): string | ArrayBuffer
```

## 背景

### base64 不同的编码方式：

- StdEncoding 是标准的 base64 编码，见 RFC 4648 中定义。
- RawStdEncoding 是标准的原始、未填充的 base64 编码，见 RFC 4648 第 3.2 节中定义；与 StdEncoding 相同，但省略了填充字符。
- URLEncoding 是 RFC 4648 中定义的备用 base64 编码，通常用于 URL 和文件名。
- RawURLEncoding 是 RFC 4648 中定义的未填充的替代 base64 编码，通常用于 URL 和文件名；与 URLEncoding 相同，但省略了填充字符。

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| input | string | 要解码的字符串 |
| encoding（可选） | string | 可选，代表前文所述的不同编码方式；可选值包括 "std"、"rawstd"、"url" 或 "rawurl"，不设置该值默认为 "std" |
| mode（可选） | string | 可选，不设置则结果为 string 类型，设置为"b"则结果为 ArrayBuffer 类型 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string 或 ArrayBuffer | base64 解码得到的结果 |

## 使用样例

### 不指定 encoding 使用 base64Decoding 方法：

```
import util from 'pts/util';

export default function () {
  // Hello, world
  console.log(util.base64Decoding('SGVsbG8sIHdvcmxk'));
}
```

**指定 encoding 使用 base64Decoding 方法：**

```
import util from 'pts/util';

export default function () {
  // http://www.example.com
  console.log(util.base64Decoding('aHR0cDovL3d3dy5leGFtcGxlLmNvbQ==',
'url'));
}
```

**指定 mode 使用 base64Decoding 方法：**

```
import util from 'pts/util';

export default function () {
  // [object ArrayBuffer]
  console.log(util.base64Decoding('SGVsbG8sIHdvcmxk', 'std', 'b'));
}
```

# util.cloudAPISignatureV3

最近更新时间：2024-12-30 16:05:04

在脚本执行过程中，util.cloudAPISignatureV3 用于调用腾讯云 API 进行签名方法 v3 签名；
详情参考腾讯云 API 签名方法 v3 文档 。

```
cloudAPISignatureV3(param: CloudAPISignatureV3Param): string
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| param | CloudAPISignatureV3Param | 签名参数 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 签名结果 |

## 使用样例

### 调用方法进行签名并访问云 API：

```
import util from 'pts/util';
import http from 'pts/http';

export default function () {
  const timestamp = parseInt(new Date().getTime() / 1000);
  const body = {
    EnvironmentId: 'wtp',
    TopicName: 'access_server',
    ClusterId: 'pulsar-vgb3w9ezndvx',
  };
  const headers = {
    'Content-Type': 'application/json',
    Host: 'tdmq.tencentcloudapi.com',
    'X-TC-Action': 'DescribeSubscriptions',
    'X-TC-Version': '2020-02-17',
    'X-TC-Timestamp': timestamp.toString(),
    'X-TC-Region': 'ap-guangzhou',
```

```
  };
  // 调用方法
  headers.Authorization = util.cloudAPISignatureV3({
    secretID: 'xxx',
    secretKey: 'xxx',
    service: 'tdmq',
    method: 'POST',
    timestamp,
    headers,
    body,
  });
  const resp = http.post('https://tdmq.tencentcloudapi.com', body, {
    headers,
  });
  console.log(resp.body);
}
```

# util.md5Sum

最近更新时间：2024-12-30 16:05:04

在脚本执行过程中，util.md5Sum 用于进行 md5 加密。

```
md5Sum(data: string | ArrayBuffer): string
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| data | string 或 ArrayBuffer | 要加密的数据 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 加密结果 |

## 样例

### 调用方法进行 md5 加密

```
import util from 'pts/util';

export default function () {
  console.log(util.md5Sum('12345')); // 827ccb0eea8a706c4c34a16891f84e7b
}
```

# util.sloginEncrypt

最近更新时间：2024-12-30 16:05:04

**在脚本执行过程中，util.sloginEncrypt 用于 QQ slogin 加密。**

```
sloginEncrypt(salt: number, pwd: string, vcode: string): string
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| salt | number | QQ 号码数字 |
| pwd | string | 用户的明文密码 |
| vcode | string | appid，即 aid 字段 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 加密后的密码 |

## 样例

**调用方法进行 qq slogin 加密：**

```
import util from 'pts/util';

export default function () {
  // lXYi46n51i4I2E6rFgaR75Lnp9kt4S4ZTq9ZTCxPv-
Ce0jWsjCss2uCl9Hed163KGkCLUxFivS9BTGRyR7YuWrDa9*tGcqal6q3BW2jxPR2M3Si3Q2
prGGIM5sIgwaaBeQWo1w-67Hgd-Qt*N4fszGRSS55VDl-
b4THwmOAp6eKA*sG80HEzbLRUWmNnfmg8wdmtyxiZisYtyWI2HJozH1EKuN2u9byOvFnMdzC
MlL7kPIZACk3zt84DM5byfCVpBII5N1EM6IMZ*u7A2WOd2c2RerWbVwAyu1raYoZwTODeOx1
8xw2uTnGi8aLJTz4PIG*3svujqwMayIgtzhq1IQ__
  console.log(util.sloginEncrypt(123456, 'abcdef', '14'));
}
```

# util.toArrayBuffer

最近更新时间：2024-12-30 16:05:04

**在脚本执行过程中，util.toArrayBuffer 用于将参数转换成 JS 中的字节数组。**

```
toArrayBuffer(data: string | ArrayBuffer): ArrayBuffer
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| data | string 或 ArrayBuffer | 要转换的数据 |

## 返回

| 类型 | 描述 |
| --- | --- |
| ArrayBuffer | 转换后的结果 |

## 样例

**调用方法进行数据的转换：**

```
import util from 'pts/util';

export default function () {
  console.log(util.toArrayBuffer('12345')); // [object ArrayBuffer]
}
```

# util.uuid

最近更新时间：2024-12-30 16:05:04

**在脚本执行过程中，util.uuid 用于生成 uuid，使用 uuid v4 版本。**

```
uuid(): string
```

## 返回

| 类型 | 描述 |
| --- | --- |
| string | uuid 字符串 |

## 样例

**调用 util.uuid：**

```
import util from 'pts/util';

export default function () {
  console.log(util.uuid()) // 5fbf1e59-cabf-469b-9d9f-6622e97de1ec
}
```

# CloudAPISignatureV3Param

最近更新时间：2024-12-30 16:05:04

CloudAPISignatureV3Param 是调用 util.cloudAPISignatureV3 方法进行签名时的参数 Object。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| secretID | string | 密钥 ID，标识 API 调用者身份 |
| secretKey | string | 密钥 Key，验证 API 调用者的身份 |
| service | string | 产品名称 |
| method | string | 调用方法，如 "POST" |
| timestamp | string | 时间戳 |
| body | string、object 或 ArrayBuffer | 请求体 |
| query | Record<string, string> | 请求参数 |
| headers | Record<string, string> | 请求头 |

## 样例

### 调用 util.cloudAPISignatureV3 方法进行签名：

```
import util from 'pts/util';
import http from 'pts/http';

export default function () {
  const timestamp = parseInt(new Date().getTime() / 1000);
  const body = {
    EnvironmentId: 'wtp',
    TopicName: 'access_server',
    ClusterId: 'pulsar-vgb3w9ezndvx',
  };
  const headers = {
    'Content-Type': 'application/json',
    Host: 'tdmq.tencentcloudapi.com',
    'X-TC-Action': 'DescribeSubscriptions',
```

```
      'X-TC-Version': '2020-02-17',
      'X-TC-Timestamp': timestamp.toString(),
      'X-TC-Region': 'ap-guangzhou',
    };
    // 调用 util.cloudAPISignatureV3，内部的参数即 CloudAPISignatureV3Param
    headers.Authorization = util.cloudAPISignatureV3({
      secretID: 'xxx',
      secretKey: 'xxx',
      service: 'tdmq',
      method: 'POST',
      timestamp,
      headers,
      body,
    });
    const resp = http.post('https://tdmq.tencentcloudapi.com', body, {
      headers,
    });
    console.log(resp.body);
}
```

# pts/ws
# 模块概览

最近更新时间：2024-12-30 16:05:04

JavaScript API 中的 pts/ws 模块用于实现基于 Websocket 协议的基本功能。

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| connect(url, callback, [headers]) | Response | 建立 Websocket 连接，并在回调函数中定义业务逻辑，执行完回调函数后，ws.connect 返回 ws.Response 对象。 |

## 对象

| 对象 | 描述 |
| --- | --- |
| Socket 概览 | ws.connect 建立成功后，传递 ws.Socket 对象进入 callback 回调函数，用以定义 Websocket 的请求逻辑。 |
| Response | ws.connect 执行完回调函数后，返回的 ws.Response 对象。 |

## 样例

```
import ws from 'pts/ws';
import { check, sleep } from 'pts';

export default function () {
  const res = ws.connect('ws://localhost:8080/echo', function (socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('message received: ',
data));
    socket.on('close', () => console.log('disconnected'));
    socket.on('ping', () => console.log('ping'));
    socket.on('pong', () => console.log('pong'));
    socket.on('error', (e) => console.log('error happened', e.error()));
    socket.send('message');
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
      socket.close();
```

```
    }, 3000);
    socket.setInterval(function () {
        socket.ping();
    }, 500);
    socket.setLoop(function () {
        sleep(0.1)
        socket.send('loop message')
    });
  });
  check('status is 101', () => res.status === 101);
};
```

# ws.connect

最近更新时间：2024-12-30 16:05:04

ws.connect 根据指定参数建立 Websocket 连接，并执行用户定义的逻辑，返回 Response 对象。

```
connect(url: string, callback: (socket: Socket) => void,
headers?: Record<string, string>): Response
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| url | string | 请求连接的地址。 |
| callback | function | 回调函数，在完成连接后将 ws.Socket 对象传入该回调函数，用户可以在该函数中定义 Websocket 请求逻辑。 |
| headers（可选） | Record<string, string> | 请求连接时的 headers 配置。 |

## 返回

| 类型 | 描述 |
| --- | --- |
| Response | object，包含 ws.connect 返回的响应结果。 |

## 样例

### 建立连接：

```
import ws from 'pts/ws';

export default function () {
  const res = ws.connect("ws://localhost:8080/echo", function (socket) {
    socket.on('open', () => {
      console.log('connected');
      socket.close();
    });
  });
}
```

### 建立连接，并指定 headers 参数：

```
import ws from 'pts/ws';

export default function () {
  const headers = {
    'X-MyApplication': 'PTS',
    'X-MyScript': 'Websocket',
  }
  const res = ws.connect("ws://localhost:8080/echo", function (socket) {
    socket.on('open', () => {
      console.log('connected');
      socket.close();
    });
  }, headers);
}
```

# Response

最近更新时间：2024-12-30 16:05:04

ws.Response 是 ws.connect 返回的响应结果。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| body | string | 响应包体内容 |
| headers | Record<string, string> | 响应头参数 |
| status | number | 状态码 |
| url | string | 请求地址 |

# Socket
# Socket 概览

最近更新时间：2024-12-30 16:05:05

在 Websocket 连接建立成功后，PTS 将创建好的 ws.Socket 对象传递进入回调函数，用户通过调用 Socket 的方法进行 WebSocket 请求逻辑的定义。

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| close() | void | 关闭连接 |
| on(event, callback) | void | 消息事件 event 监听，并根据 callback 处理事件；目前 PTS 支持的事件列表如下：<br>• open：建立连接<br>• close：关闭连接<br>• message：接受文本消息<br>• binaryMessage：接受二进制消息<br>• ping：接收 ping 消息 |
| ping() | void | 发送 ping 消息 |
| send(msg) | void | 发送文本消息 |
| sendBinary(msg) | void | 发送二进制消息 |
| setInterval(callback, intervalMs) | void | 设置轮询函数 |
| setLoop(callback) | void | 设置循环执行函数 |
| setTimeout(callback, intervalMs) | void | 设置定时函数 |

# Socket.close

最近更新时间：2024-12-30 16:05:05

Socket.close 用于关闭连接。

```
close(): void
```

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

```
import ws from 'pts/ws';
import { check, sleep } from 'pts';

export default function () {
  const res = ws.connect('ws://localhost:8080/echo', function (socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('Message received: ',
data));
    socket.on('close', () => console.log('disconnected'));
    socket.send('message');
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
      // 关闭连接
      socket.close();
    }, 3000);
    socket.setInterval(function () {
      socket.ping();
    }, 500);
    socket.setLoop(function () {
      sleep(0.1);
      socket.send('loop message');
    });
  });
  check('status is 101', () => res.status === 101);
}
```

# Socket.on

最近更新时间：2024-12-30 16:05:05

**Socket.on 用于消息事件监听。**

```
on(event: string, callback: ((...args: any[]) => void)): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| event | string | 事件名，支持的事件列表如下：<br>• open，建立连接；<br>• close，关闭连接；<br>• message，接受文本消息；<br>• binaryMessage，接受二进制消息；<br>• pong，接收 pong 消息；<br>• ping，接收 ping 消息； |
| callback | function | 回调函数 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

```
import ws from 'pts/ws';
import { check, sleep } from 'pts';

export default function () {
  const res = ws.connect('ws://localhost:8080/echo', function (socket) {
    // 消息事件监听
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('Message received: ',
data));
    socket.on('close', () => console.log('disconnected'));
    socket.send('message');
    socket.setTimeout(function () {
```

```
      console.log('3 seconds passed, closing the socket');
      socket.close();
    }, 3000);
    socket.setInterval(function () {
      socket.ping();
    }, 500);
    socket.setLoop(function () {
      sleep(0.1);
      socket.send('loop message');
    });
  });
  check('status is 101', () => res.status === 101);
}
```

# Socket.ping

最近更新时间：2024-12-30 16:05:05

Socket.ping 用于发送 ping 消息。

```
ping(): void
```

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

```
import ws from 'pts/ws';
import { check, sleep } from 'pts';

export default function () {
  const res = ws.connect('ws://localhost:8080/echo', function (socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('Message received: ',
data));
    socket.on('close', () => console.log('disconnected'));
    socket.send('message');
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
      socket.close();
    }, 3000);
    socket.setInterval(function () {
    // 发送 ping 消息
      socket.ping();
    }, 500);
    socket.setLoop(function () {
      sleep(0.1);
      socket.send('loop message');
    });
  });
  check('status is 101', () => res.status === 101);
}
```

# Sokcet.send

最近更新时间：2024-12-30 16:05:05

**Socket.send 用于文本消息发送。**

```
send(msg: string): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| msg | string | 文本内容 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

```
import ws from 'pts/ws';
import { check, sleep } from 'pts';

export default function () {
  const res = ws.connect('ws://localhost:8080/echo', function (socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('Message received: ',
data));
    socket.on('close', () => console.log('disconnected'));
    // 文本消息发送
    socket.send('message');
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
      // 关闭连接
      socket.close();
    }, 3000);
    socket.setInterval(function () {
      socket.ping();
    }, 500);
    socket.setLoop(function () {
```

```
        sleep(0.1);
        socket.send('loop message');
      });
    });
    check('status is 101', () => res.status === 101);
}
```

# Socket.sendBinary

最近更新时间：2024-12-30 16:05:05

**Socket.sendBinary 用于文本消息发送。**

```
sendBinary(msg: ArrayBuffer): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| msg | ArrayBuffer | 二进制内容 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

```
import ws from 'pts/ws';
import { check, sleep } from 'pts';

export default function () {
  const res = ws.connect('ws://localhost:8080/echo', function (socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('Message received: ',
data));
    socket.on('close', () => console.log('disconnected'));
    socket.send('message');
    // 二进制消息发送
    socket.sendBinary(new ArrayBuffer(1));
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
      socket.close();
    }, 3000);
    socket.setInterval(function () {
      socket.ping();
    }, 500);
    socket.setLoop(function () {
```

```
        sleep(0.1);
        socket.send('loop message');
      });
    });
    check('status is 101', () => res.status === 101);
}
```

# Socket.setInterval

最近更新时间：2024-12-30 16:05:05

Socket.setInterval 用于设置轮询函数。

```
setInterval(callback: (() => void), intervalMs: number): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| callback | function | 回调函数 |
| intervalMs | number | 设置时间，单位为毫秒 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

```
import ws from 'pts/ws';
import { check, sleep } from 'pts';

export default function () {
  const res = ws.connect('ws://localhost:8080/echo', function (socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('Message received: ',
data));
    socket.on('close', () => console.log('disconnected'));
    socket.send('message');
    socket.sendBinary(new ArrayBuffer(1));
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
      socket.close();
    }, 3000);
    // 设置轮询函数
    socket.setInterval(function () {
      socket.ping();
    }, 500);
```

```
    socket.setLoop(function () {
      sleep(0.1);
      socket.send('loop message');
    });
  });
  check('status is 101', () => res.status === 101);
}
```

# Socket.setLoop

最近更新时间：2024-12-30 16:05:05

**Socket.setLoop 用于设置循环执行函数。**

```
setLoop(callback: (() => void)): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| callback | function | 回调函数 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

```
import ws from 'pts/ws';
import { check, sleep } from 'pts';

export default function () {
  const res = ws.connect('ws://localhost:8080/echo', function (socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('Message received: ',
data));
    socket.on('close', () => console.log('disconnected'));
    socket.send('message');
    socket.sendBinary(new ArrayBuffer(1));
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
      socket.close();
    }, 3000);
    socket.setInterval(function () {
      socket.ping();
    }, 500);
    // 设置循环执行函数
    socket.setLoop(function () {
```

```
        sleep(0.1);
        socket.send('loop message');
      });
    });
    check('status is 101', () => res.status === 101);
}
```

# Socket.setTimeout

最近更新时间：2024-12-30 16:05:05

Socket.setTimeout 用于设置定时函数。

```
setTimeout(callback: (() => void), intervalMs: number): void
```

## 参数新增

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| callback | function | 回调函数 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

```
import ws from 'pts/ws';
import { check, sleep } from 'pts';

export default function () {
  const res = ws.connect('ws://localhost:8080/echo', function (socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('Message received: ',
data));
    socket.on('close', () => console.log('disconnected'));
    socket.send('message');
    socket.sendBinary(new ArrayBuffer(1));
    // 设置定时函数
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
      socket.close();
    }, 3000);
    socket.setInterval(function () {
      socket.ping();
    }, 500);
    socket.setLoop(function () {
```

```
        sleep(0.1);
        socket.send('loop message');
      });
    });
    check('status is 101', () => res.status === 101);
}
```

# pts/redis
# 模块概览

最近更新时间：2024-11-29 15:25:42

JavaScript API 中的 pts/redis 模块用于建立同 Redis 实例的连接 Client，并通过该 Client 进行操作。

## 对象

| 对象 | 描述 |
| --- | --- |
| Client | Redis client 实例 |

# Client
# Client 概览

最近更新时间：2024-11-29 15:25:42

通过 `new redis.Client` 方法，您可以创建一个 Client 实例。该方法的参数为目标 redis 的地址。

## 构造函数

```
new Client(url: string): Client
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| url | string | 目标 redis 的地址，例如 redis://<user>:<password>@<host>:<port>/<db_number> |

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| get(key) | string | 获取指定 key 的值 |
| set(key, value, expiration?) | string | 设置指定 key 的值 |
| del(...keys) | number | 删除已存在的 key |
| lPush(key, ...values) | number | 将一个或多个值插入到列表头部 |
| rPush(key, ...values) | number | 将一个或多个值插入到列表尾部 |
| lPop(key) | string | 移除并获取列表的第一个元素 |
| rPop(key) | string | 移除并获取列表的最后一个元素 |
| lRange(key, start, stop) | string[] | 获取列表指定范围内的元素 |
| lIndex(key, index) | string | 通过索引获取列表中的元素 |
| lLen(key) | number | 获取列表长度 |

| | | |
|---|---|---|
| lSet(key, index, value) | string | 通过索引设置列表元素的值 |
| lRem(key, count, value) | number | 移除列表元素 |
| hSet(key, ...members) | number | 设置哈希表 key 中的字段和值 |
| hGet(key, field) | string | 获取存储在哈希表中指定字段的值 |
| hDel(key, ...fields) | number | 删除一个或多个哈希表字段 |
| hLen(key) | number | 获取哈希表中字段的数量 |
| sAdd(key, ...members) | number | 向集合添加一个或多个成员 |
| sRem(key, ...members) | number | 移除集合中一个或多个成员 |
| sIsMember(key, member) | boolean | 判断 member 元素是否是集合 key 的成员 |
| sMembers(key) | string[] | 返回集合中的所有成员 |
| sRandMember(key) | string | 随机返回集合中一个元素 |
| sPop(key) | string | 随机移除并返回集合中的一个元素 |

## 示例

同 Redis 建立连接并进行操作。

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let resp = client.set("key", "hello, world", 0);
    console.log(`redis set ${resp}`); // OK
    let val = client.get("key");
    console.log(`redis get ${val}`); // hello, world
    let cnt = client.del("key");
    console.log(`redis del ${cnt}`); // 1

    let lpushResp = client.lPush("list", "foo");
    console.log(`redis lpush ${lpushResp}`); // OK
    let lpopResp = client.lPop("list");
```

```
    console.log(`redis lpop ${lpopResp}`); // foo
    let listLen = client.lLen("list");
    console.log(`redis llen ${listLen}`); // 0

    let hashSetResp = client.hSet("hash", "k", 1); // [k1, v1, k2, v2,
...]
    console.log(`redis hset ${hashSetResp}`); // 1
    let hashGetResp = client.hGet("hash", "k");
    console.log(`redis hget ${hashGetResp}`); // 1
    let hashDelResp = client.hDel("hash", "k");
    console.log(`redis hdel ${hashDelResp}`); // 1

    let setAddResp = client.sAdd("set", "hello");
    console.log(`redis sadd ${setAddResp}`); // 1
    let setPopResp = client.sPop("set");
    console.log(`redis spop ${setPopResp}`); // hello
}
```

# Client.get

最近更新时间：2024-11-29 15:25:42

**get 方法用于获取指定 key 的值。**

```
get(key: string): string
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 值 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://<password>@<host>:6379/0");

export default function main() {
    let setResp = client.set("key", "hello, world", 0);
    console.log(`redis set ${setResp}`); // OK
    let getResp = client.get("key");
    console.log(`redis get ${getResp}`); // hello, world
}
```

# Client.set

最近更新时间：2024-11-29 15:25:42

**set 方法用于设置指定 key 的值。**

```
set(key: string, value: string, expiration?: number): string
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |
| value | string | 值 |
| expiration | number | 可选，过期时间，单位秒。不填表示不设置过期时间，-1 表示 keepttl |

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 成功时，返回 "OK" |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let setResp = client.set("key", "hello, world", 0);
    console.log(`redis set ${setResp}`); // OK
    let getResp = client.get("key");
    console.log(`redis get ${getResp}`); // hello, world
}
```

# Client.del

最近更新时间：2024-11-29 15:25:42

del 方法用于删除已存在的 key。

```
del(...keys: string[]): number
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| ...keys | string[] | 键名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| number | 成功，返回被删除条目的数量 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let setResp = client.set("key", "hello, world", 0);
    console.log(`redis set ${setResp}`); // OK
    let delResp = client.del("key");
    console.log(`redis del ${delResp}`); // 1
}
```

# Client.lPush

最近更新时间：2024-11-29 15:25:42

**lPush 方法用于将一个或多个值插入到列表头部。**

```
lPush(key: string, ...values: (string | number)[]): number
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |
| ...values | (string \| number)[] | 值 |

## 返回

| 类型 | 描述 |
| --- | --- |
| number | 成功时，返回插入后 list 的长度 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let lPushResp = client.lPush("list", "foo");
    console.log(`redis lPush ${lPushResp}`); // 1
}
```

# Client.rPush

最近更新时间：2024-11-29 15:25:42

**rPush 方法用于将一个或多个值插入到列表尾部。**

```
rPush(key: string, ...values: (string | number)[]): number
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |
| ...values | (string \| number)[] | 值 |

## 返回

| 类型 | 描述 |
| --- | --- |
| number | 成功时，返回插入后 list 的长度 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let rPushResp = client.rPush("list", "foo");
    console.log(`redis rPush ${rPushResp}`); // 1
}
```

# Client.lPop

最近更新时间：2024-11-29 15:25:42

**lPop 方法用于移除并获取列表的第一个元素。**

```
lPop(key: string): string
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 成功时，返回列表中的第一个元素 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let lPushResp = client.lPush("list", "foo");
    console.log(`redis lPush ${lPushResp}`); // 1
    let lPopResp = client.lPop("list");
    console.log(`redis lPop ${lPopResp}`); // foo
}
```

# Client.rPop

最近更新时间：2024-11-29 15:25:42

**rPop 方法用于移除并获取列表的最后一个元素。**

```
rPop(key: string): string
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 成功时，返回列表中的最后一个元素 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let rPushResp = client.rPush("list", "foo");
    console.log(`redis rPush ${rPushResp}`); // 1
    let rPopResp = client.rPop("list");
    console.log(`redis rPop ${rPopResp}`); // foo
}
```

# Client.lRange

最近更新时间：2024-11-29 15:25:42

**lRange 方法用于获取列表指定范围内的元素。**

```
lRange(key: string, start, stop: number): string[]
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |
| start | string | 起始位置 |
| stop | string | 结束位置 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string[] | 成功时，返回 [start, stop] 区间的元素 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let lPushResp = client.lPush("list", "foo", "bar", "zoo");
    console.log(`redis lPush ${lPushResp}`); // 3
    let lRangeResp = client.lRange("list", 0, 1);
    console.log(`redis lRange ${lRangeResp}`); // zoo,bar
}
```

# Client.lIndex

最近更新时间：2024-11-29 15:25:42

**lIndex 方法用于通过索引获取列表中的元素。**

```
lIndex(key: string, index: number): string
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |
| index | number | 索引 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 成功时，返回索引对应的元素 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let lPushResp = client.lPush("list", "foo", "bar", "zoo");
    console.log(`redis lPush ${lPushResp}`); // 3
    let lIndexResp = client.lIndex("list", 1);
    console.log(`redis lIndex ${lIndexResp}`); // bar
}
```

# Client.lLen

最近更新时间：2024-11-29 15:25:42

**lLen 方法用于获取列表长度。**

```
lLen(key: string): number
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| number | 成功时，返回列表的长度 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let lPushResp = client.lPush("list", "foo", "bar", "zoo");
    console.log(`redis lPush ${lPushResp}`); // 3
    let lLenResp = client.lLen("list");
    console.log(`redis lLen ${lLenResp}`); // 3
}
```

# Client.lSet

最近更新时间：2024-11-29 15:25:42

**lSet 方法用于通过索引设置列表元素的值。**

```
lSet(key: string, index: number, value: string | number): string
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |
| index | number | 索引 |
| value | string \| number | 值 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 成功时，返回“OK” |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let lPushResp = client.lPush("list", "foo", "bar", "zoo");
    console.log(`redis lPush ${lPushResp}`); // 3
    let lSetResp = client.lSet("list", 1, "bar2");
    console.log(`redis lSet ${lSetResp}`); // OK
    let lIndexResp = client.lIndex("list", 1);
    console.log(`redis lIndex ${lIndexResp}`); // bar2
}
```

# Client.lRem

最近更新时间：2024-11-29 15:25:42

**lRem 方法用于移除列表元素。**

```
lRem(key: string, count: number, value: string | number): number
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |
| count | number | 移除的数量，正数表示从表头开始，负数表示从表尾开始，0 代表全部移除 |
| value | string \| number | 移除元素的值 |

## 返回

| 类型 | 描述 |
| --- | --- |
| number | 成功时，返回移除元素的数量 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let lPushResp = client.lPush("list", "foo", "bar", "zoo", "foo",
"bar", "zoo");
    console.log(`redis lPush ${lPushResp}`); // 6
    let lRemResp1 = client.lRem("list", 1, "bar");
    console.log(`redis lRem ${lRemResp1}`); // 1
    let lRemResp2 = client.lRem("list", -1, "foo");
    console.log(`redis lRem ${lRemResp2}`); // 1
    let lRangeResp = client.lRange("list", 0, 4);
    console.log(`redis lRange ${lRangeResp}`); // zoo,foo,zoo,bar
```

```
}
```

# Client.hSet

最近更新时间：2024-11-29 15:25:42

**hSet 方法用于设置哈希表 key 中的字段和值。**

```
hSet(key: string, ...members: (string | number)[]): number
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |
| ...members | (string \| number)[] | hash 成员，以 key1、value1、key2、value2 形式输入 |

## 返回

| 类型 | 描述 |
| --- | --- |
| number | 成功时，返回添加成功的条数 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let hSetResp = client.hSet("hash", "k", 1); // [k1, v1, k2, v2, ...]
    console.log(`redis hSet ${hSetResp}`); // 1
}
```

# Client.hGet

最近更新时间：2024-11-29 15:25:42

**hGet 方法用于获取存储在哈希表中指定字段的值。**

```
hGet(key: string, field: string): string
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |
| field | string | 字段名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 成功时，返回对应字段的值 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let hSetResp = client.hSet("hash", "k", 1, "k1", 2); // [k1, v1, k2,
v2, ...]
    console.log(`redis hSet ${hSetResp}`); // 2
    let hGetResp = client.hGet("hash", "k");
    console.log(`redis hGet ${hGetResp}`); // 1
}
```

# Client.hDel

最近更新时间：2024-11-29 15:25:42

**hDel 方法用于删除一个或多个哈希表字段。**

```
hDel(key: string, ...fields: string[]): number
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |
| ...fields | string[] | 字段名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| number | 成功时，返回被删除条目的数量 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let hSetResp = client.hSet("hash", "k", 1, "k1", 2);
    console.log(`redis hSet ${hSetResp}`); // 2
    let hDelResp = client.hDel("hash", "k");
    console.log(`redis hDel ${hDelResp}`); // 1
}
```

# Client.hLen

最近更新时间：2024-11-29 15:25:42

**hLen 方法用于获取哈希表中字段的数量。**

```
hLen(key: string): number
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| number | 成功时，返回条目的数量 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let hSetResp = client.hSet("hash", "k", 1, "k1", 2);
    console.log(`redis hSet ${hSetResp}`); // 2
    let hLenResp = client.hLen("hash");
    console.log(`redis hLen ${hLenResp}`); // 2
}
```

# Client.sAdd

最近更新时间：2024-11-29 15:25:42

**sAdd 方法用于向集合添加一个或多个成员。**

```
sAdd(key: string, ...members: (string | number)[]): number
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |
| ...members | (string \| number)[] | 待添加的成员 |

## 返回

| 类型 | 描述 |
| --- | --- |
| number | 成功时，返回添加成功的数量 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let sAddResp = client.sAdd("set", "hello");
    console.log(`redis sAdd ${sAddResp}`); // 1
}
```

# Client.sRem

最近更新时间：2024-11-29 15:25:42

**sRem 方法用于移除集合中一个或多个成员。**

```
sRem(key: string, ...members: (string | number)[]): number
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |
| ...members | (string \| number)[] | 待删除的成员 |

## 返回

| 类型 | 描述 |
| --- | --- |
| number | 成功时，返回删除成功的数量 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let sAddResp = client.sAdd("set", "hello", "world");
    console.log(`redis sAdd ${sAddResp}`); // 2
    let sRemResp = client.sRem("set", "hello");
    console.log(`redis sRem ${sRemResp}`); // 1
}
```

# Client.sIsMember

最近更新时间：2024-11-29 15:25:42

**sIsMember 方法用于判断 member 元素是否是集合 key 的成员。**

```
sIsMember(key: string, member: string | number): boolean
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |
| member | string \| number | 待查询的成员 |

## 返回

| 类型 | 描述 |
| --- | --- |
| boolean | 存在时返回 true，否则返回 false |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let sAddResp = client.sAdd("set", "hello", "world");
    console.log(`redis sAdd ${sAddResp}`); // 2
    let sIsMemberResp = client.sIsMember("set", "hello");
    console.log(`redis sIsMember ${sIsMemberResp}`); // true
}
```

# Client.sMembers

最近更新时间：2024-11-29 15:25:42

**sMembers 方法用于返回集合中的所有成员。**

```
sMembers(key: string): string[]
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string[] | 成功时，返回所有的元素 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let sAddResp = client.sAdd("set", "hello", "world");
    console.log(`redis sAdd ${sAddResp}`); // 2
    let sMembersResp = client.sMembers("set");
    console.log(`redis sMembers ${sMembersResp}`); // hello,world
}
```

# Client.sRandMember

最近更新时间：2024-11-29 15:25:42

**sRandMember 方法用于随机返回集合中一个元素。**

```
sRandMember(key: string): string
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 成功时，随机返回一个元素 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let sAddResp = client.sAdd("set", "hello", "world");
    console.log(`redis sAdd ${sAddResp}`); // 2
    let sRandMemberResp = client.sRandMember("set");
    console.log(`redis sRandMember ${sRandMemberResp}`); // world
}
```

# Client.sPop

最近更新时间：2024-11-29 15:25:42

**sPop 方法用于随机移除并返回集合中的一个元素。**

```
sPop(key: string): string
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| key | string | 键名 |

## 返回

| 类型 | 描述 |
| --- | --- |
| string | 成功时，随机删除一个元素并返回 |

## 样例

```
import redis from "pts/redis";

let client = new redis.Client("redis://:<password>@<host>:6379/0");

export default function main() {
    let sAddResp = client.sAdd("set", "hello", "world");
    console.log(`redis sAdd ${sAddResp}`); // 2
    let sPopResp = client.sPop("set");
    console.log(`redis sPop ${sPopResp}`); // world
}
```

# pts/socketio
# 模块概览

最近更新时间：2024-12-30 16:05:05

JavaScript API 中的 pts/socketio 模块实现了 socketio 相关的功能。

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| connect(url, callback, [option]) | Response | 建立 Socket.IO 连接，并在回调函数中定义业务逻辑，执行完回调函数后，返回 Response 对象。 |

## 对象

| 对象 | 描述 |
| --- | --- |
| Option | 使用 connect 方法建立连接时的可选配置项。 |
| socketio | 若连接建立成功，创建好的 SocketIO 对象会被传入 callback 回调函数。您可在回调函数里，定义您的请求逻辑，发送/收取事件消息。 |
| Response | 执行完回调函数，connect 方法会返回 Response 对象。 |

## 样例

```
// SocketIO API
import socketio from 'pts/socketio';
import { check } from 'pts';
import util from 'pts/util';

export default function () {
  const res = socketio.connect('http://localhost:8080', function
(socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('message received: ',
data));
    socket.on('binaryMessage', (data) => console.log('binaryMessage
received: ', data));
```

```
    socket.on('close', () => console.log('disconnected'));
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
      socket.close();
    }, 3000);
    // 设置定时任务
    socket.setTimeout(function () {
      socket.emit('message', 'hello');
      socket.emit('binaryMessage', util.base64Decoding('aGVsbG8=',
'std', 'b'));
      socket.emit('ackMessage',  'hello ack', function(msg) {
        console.log('ack message received: ', msg)
      })
    }, 500);
    // 设置定期执行的任务
    socket.setInterval(function(){
      socket.emit('message', 'interval message');
    }, 500);
  }, {
    // 支持 polling、websocket 协议
    protocol:'websocket',
    headers: {
      token: 'ZER3XSR',
    }
  });
  check('status is 200', () => res.status === 200);
}
```

# socketio.connect

最近更新时间：2025-01-03 14:17:23

socketio.connect 根据指定参数建立 socketio 连接，并执行用户定义的逻辑，返回 Response 对象。

```
connect(url: string, callback: (socketIO: SocketIO) => void,
option?: Option): Response
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| url | string | 请求连接的地址。 |
| callback | function | 回调函数，在完成连接后将 socketio 对象传入该回调函数，用户可以在该函数中定义请求逻辑。 |
| option | Option | 可选，配置参数。 |

## 返回

| 类型 | 描述 |
| --- | --- |
| Response | object，包含 ws.connect 返回的响应结果。 |

## 样例

发起 socketio connect 请求。

```
import socketio from 'pts/socketio';
import { check, sleep } from 'pts';
import util  from 'pts/util';

export default function () {
    const res = socketio.connect('http://localhost:8080', function
(socket) {
        socket.on('open', () => console.log('connected'));
```

```
        socket.on('message', (data) => console.log('message received: ',
data));
        socket.on('binaryMessage', (data) => console.log('binaryMessage
received: ', data));
        socket.on('close', () => console.log('disconnected'));
        socket.on('error', (e) => console.log('error happened',
e.error()));
        socket.setTimeout(function () {
            console.log('3 seconds passed, closing the socket');
            socket.close();
        }, 3000);
        socket.setInterval(function () {
            socket.emit('message', 'interval message');
            socket.emit('binaryMessage', util.base64Decoding('aGVsbG8=',
'std', 'b'));
            socket.emit('ackMessage', 'ack message', function (msg) {
                console.log('received ackMessage: ', msg)
            })
        }, 500);
    }, {
       headers: {
         token: 'VR23EQ2R',
       },
       protocol: 'webscoket'
    });
    check('status is 200', () => res.status === 200);
};
```

# Option

最近更新时间：2024-12-30 16:05:05

**Option 是配置参数，对于不同类型的请求可以设置不同的配置。**

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| headers | Record<string,string> | 请求头 |
| protocol | string | 协议类型，支持 polling/websocket |

## 样例

```
// SocketIO API
import socketio from 'pts/socketio';
import { check, sleep } from 'pts';
import util  from 'pts/util';

export default function () {
  const res = socketio.connect('http://localhost:8080', function
(socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('message received: ',
data));
    socket.on('binaryMessage', (data) => console.log('binaryMessage
received: ', data));
    socket.on('close', () => console.log('disconnected'));
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
      socket.close();
    }, 3000);
    // 设置定时任务
    socket.setTimeout(function () {
      socket.emit('message', 'hello');
      socket.emit('binaryMessage', util.base64Decoding('aGVsbG8=',
'std', 'b'));
      socket.emit('ackMessage',  'hello ack', function(msg) {
        console.log('ack message received: ', msg)
      })
```

```
    }, 500);
    // 设置定期执行的任务
    socket.setInterval(function(){
      socket.emit('message', 'interval message');
    }, 500);
    // 设置循环执行任务，socket/context 关闭自然退出
    socket.setLoop(function () {
      sleep(0.1);
      socket.emit('message', 'loop message');
    });
  }, {
    // 支持 polling、websocket 协议
    protocol:'websocket',
    headers: {
      token: 'xxx',
    }
  });
  check('status is 200', () => res.status === 200);
}
```

# socketio
# socketio.close

最近更新时间：2024-12-30 16:05:05

**socketio.close 用于关闭连接。**

```
close(): void
```

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

```
// SocketIO API
import socketio from 'pts/socketio';
import { check, sleep } from 'pts';
import util  from 'pts/util';

export default function () {
  const res = socketio.connect('http://localhost:8080', function
(socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('message received: ',
data));
    socket.on('binaryMessage', (data) => console.log('binaryMessage
received: ', data));
    socket.on('close', () => console.log('disconnected'));
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
      socket.close();
    }, 3000);
    // 设置定时任务
    socket.setTimeout(function () {
      socket.emit('message', 'hello');
      socket.emit('binaryMessage', util.base64Decoding('aGVsbG8=',
'std', 'b'));
      socket.emit('ackMessage',  'hello ack', function(msg) {
```

```
          console.log('ack message received: ', msg)
        })
    }, 500);
    // 设置定期执行的任务
    socket.setInterval(function(){
      socket.emit('message', 'interval message');
    }, 500);
    // 设置循环执行任务，socket/context 关闭自然退出
    socket.setLoop(function () {
      sleep(0.1);
      socket.emit('message', 'loop message');
    });
  }, {
    // 支持 polling、websocket 协议
    protocol:'websocket',
    headers: {
      token: 'xxx',
    }
  });
  check('status is 200', () => res.status === 200);
}
```

# socketio.emit

最近更新时间：2024-12-30 16:05:05

**socketio.emit 发送文本/二进制消息。**

```
emit(event: string, msg: any, callback?: (...args: any[]) => void): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| event | string | 事件 |
| msg | any | 文本内容/二进制文件 |
| callback | function | 可选，回调函数 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

```
// SocketIO API
import socketio from 'pts/socketio';
import { check, sleep } from 'pts';
import util  from 'pts/util';

export default function () {
  const res = socketio.connect('http://localhost:8080', function
(socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('message received: ',
data));
    socket.on('binaryMessage', (data) => console.log('binaryMessage
received: ', data));
    socket.on('close', () => console.log('disconnected'));
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
```

```
      socket.close();
    }, 3000);
    // 设置定时任务
    socket.setTimeout(function () {
      socket.emit('message', 'hello');
      socket.emit('binaryMessage', util.base64Decoding('aGVsbG8=',
'std', 'b'));
      socket.emit('ackMessage',  'hello ack', function(msg) {
        console.log('ack message received: ', msg)
      })
    }, 500);
    // 设置定期执行的任务
    socket.setInterval(function(){
      socket.emit('message', 'interval message');
    }, 500);
    // 设置循环执行任务，socket/context 关闭自然退出
    socket.setLoop(function () {
      sleep(0.1);
      socket.emit('message', 'loop message');
    });
  }, {
    // 支持 polling、websocket 协议
    protocol:'websocket',
    headers: {
      token: 'xxx',
    }
  });
  check('status is 200', () => res.status === 200);
}
```

# socketio.on

最近更新时间：2024-12-30 16:05:05

**socketio.on 监听消息事件。**

```
on(event: string, callback: (...args: any[]) => void): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| event | string | 事件 |
| callback | function | 回调函数 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

```
// SocketIO API
import socketio from 'pts/socketio';
import { check, sleep } from 'pts';
import util  from 'pts/util';

export default function () {
  const res = socketio.connect('http://localhost:8080', function
(socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('message received: ',
data));
    socket.on('binaryMessage', (data) => console.log('binaryMessage
received: ', data));
    socket.on('close', () => console.log('disconnected'));
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
      socket.close();
    }, 3000);
```

```
    // 设置定时任务
    socket.setTimeout(function () {
      socket.emit('message', 'hello');
      socket.emit('binaryMessage', util.base64Decoding('aGVsbG8=',
'std', 'b'));
      socket.emit('ackMessage',  'hello ack', function(msg) {
        console.log('ack message received: ', msg)
      })
    }, 500);
    // 设置定期执行的任务
    socket.setInterval(function(){
      socket.emit('message', 'interval message');
    }, 500);
    // 设置循环执行任务，socket/context 关闭自然退出
    socket.setLoop(function () {
      sleep(0.1);
      socket.emit('message', 'loop message');
    });
  }, {
    // 支持 polling、websocket 协议
    protocol:'websocket',
    headers: {
      token: 'xxx',
    }
  });
  check('status is 200', () => res.status === 200);
}
```

# socketio.setInterval

最近更新时间：2024-12-30 16:05:05

socketio.setInterval 设置轮询函数。

```
setInterval(callback: () => void, intervalMs: number): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| callback | function | 回调函数 |
| intervalMs | number | 设置时间，单位毫秒 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

```
// SocketIO API
import socketio from 'pts/socketio';
import { check, sleep } from 'pts';
import util  from 'pts/util';
```

# socketio.setLoop

最近更新时间：2024-12-30 16:05:05

socketio.setLoop 循环执行函数。

```
setLoop(callback: () => void): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| callback | function | 回调函数 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

```
// SocketIO API
import socketio from 'pts/socketio';
import { check } from 'pts';
import util from 'pts/util';

export default function () {
  const res = socketio.connect('http://localhost:8080', function
(socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('message received: ',
data));
    socket.on('binaryMessage', (data) => console.log('binaryMessage
received: ', data));
    socket.on('close', () => console.log('disconnected'));
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
      socket.close();
    }, 3000);
    // 设置定时任务
    socket.setTimeout(function () {
```

```
      socket.emit('message', 'hello');
      socket.emit('binaryMessage', util.base64Decoding('aGVsbG8=',
'std', 'b'));
      socket.emit('ackMessage',  'hello ack', function(msg) {
        console.log('ack message received: ', msg)
      })
    }, 500);
    // 设置定期执行的任务
    socket.setInterval(function(){
      socket.emit('message', 'interval message');
    }, 500);
  }, {
    // 支持 polling、websocket 协议
    protocol:'websocket',
    headers: {
      token: 'ZER3XSR',
    }
  });
  check('status is 200', () => res.status === 200);
}
```

# socketio.setTimeout

最近更新时间：2024-12-30 16:05:05

socketio.setTimeout 设置定时函数。

```
setTimeout(callback: () => void, intervalMs: number): void
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| callback | function | 回调函数 |
| intervalMs | number | 设置时间，单位毫秒 |

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

```
// SocketIO API
import socketio from 'pts/socketio';
import { check, sleep } from 'pts';
import util  from 'pts/util';

export default function () {
  const res = socketio.connect('http://localhost:8080', function
(socket) {
    socket.on('open', () => console.log('connected'));
    socket.on('message', (data) => console.log('message received: ',
data));
    socket.on('binaryMessage', (data) => console.log('binaryMessage
received: ', data));
    socket.on('close', () => console.log('disconnected'));
    socket.setTimeout(function () {
      console.log('3 seconds passed, closing the socket');
      socket.close();
    }, 3000);
```

```
    // 设置定时任务
    socket.setTimeout(function () {
      socket.emit('message', 'hello');
      socket.emit('binaryMessage', util.base64Decoding('aGVsbG8=',
'std', 'b'));
      socket.emit('ackMessage',  'hello ack', function(msg) {
        console.log('ack message received: ', msg)
      })
    }, 500);
    // 设置定期执行的任务
    socket.setInterval(function(){
      socket.emit('message', 'interval message');
    }, 500);
    // 设置循环执行任务，socket/context 关闭自然退出
    socket.setLoop(function () {
      sleep(0.1);
      socket.emit('message', 'loop message');
    });
  }, {
    // 支持 polling、websocket 协议
    protocol:'websocket',
    headers: {
      token: 'xxx',
    }
  });
  check('status is 200', () => res.status === 200);
}
```

# Response

最近更新时间：2024-12-30 16:05:05

socketio 是 socketio.connect 返回的响应结果。

## 字段

| 字段 | 类型 | 描述 |
| --- | --- | --- |
| body | string | 响应包体内容 |
| headers | Record<string, string> | 响应头参数 |
| status | number | 状态码 |
| url | string | 请求地址 |

# pts/socket
# 模块概览

最近更新时间：2024-12-30 16:05:05

JavaScript API 中的 pts/socket 模块用于建立 Socket 实例，然后通过该实例发送或接收 TCP/UDP 数据。

## 对象

| 对象 | 描述 |
| --- | --- |
| Conn | Socket 实例 |

# Conn

# Conn 概览

最近更新时间：2025-01-03 14:17:23

通过 `new socket.Conn` 方法，您可以创建一个 Socket 实例。该方法的参数为协议名（ `tcp` 或 `udp` ）、服务地址、服务端口。

## 构造函数

```
new Conn(): Conn
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| network | string | 用于建立连接的协议名（tcp 或 udp） |
| host | string | 服务的 IP 地址 |
| port | number | 服务的端口 |

## 方法

| 方法 | 返回类型 | 描述 |
| --- | --- | --- |
| send() | number | 发送请求数据 |
| recv() | ArrayBuffer | 接收响应数据 |
| close() | void | 关闭连接 |

## 样例

建立 socket 连接发起 tcp/udp 请求。

```
import socket from "pts/socket";
import util from 'pts/util';
import {sleep} from 'pts';

export default function () {
```

```
    const tcp_socket = new socket.Conn('tcp', '127.0.0.1', 80);
    const send_data = `GET /get HTTP/1.1
Host: 127.0.0.1
User-Agent: pts-engine
\r\n`;
    tcp_socket.send(util.toArrayBuffer(send_data));
    const bytes_read = tcp_socket.recv(512);
    tcp_socket.close();
    console.log(bytes_read);
    sleep(1);
}
```

# Conn.send

最近更新时间：2024-12-30 16:05:05

**send 方法用于发送请求数据。**

```
send(b: ArrayBuffer): number
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| b | ArrayBuffer | 待发送的二进制数据 |

## 返回

| 类型 | 描述 |
| --- | --- |
| number | 发送的字节数 |

## 样例

**发送请求数据：**

```
import socket from "pts/socket";
import util from 'pts/util';

export default function () {
    const tcp_socket = new socket.Conn('tcp', '127.0.0.1', 80);
    const data = `GET /get HTTP/1.1
Host: 127.0.0.1
\n`;
    const sent_bytes = tcp_socket.send(util.toArrayBuffer(data));
    console.log(sent_bytes); // 35
}
```

# Conn.recv

最近更新时间：2024-12-30 16:05:05

**recv 用于接收数据。**

```
recv(size: number): ArrayBuffer
```

## 参数

| 参数 | 类型 | 描述 |
| --- | --- | --- |
| size | number | 可接收的最大字节数限制 |

## 返回

| 类型 | 描述 |
| --- | --- |
| ArrayBuffer | 接收到的二进制数据 |

## 样例

**接收请求的响应数据：**

```
import socket from "pts/socket";
import util from 'pts/util';

export default function () {
    const tcp_socket = new socket.Conn('tcp', '127.0.0.1', 80);
    const send_data = `GET /get HTTP/1.1
Host: 127.0.0.1
\n`;
    tcp_socket.send(util.toArrayBuffer(send_data));
    tcp_socket.recv(512);
    tcp_socket.close();
}
```

# Conn.close

最近更新时间：2024-12-30 16:05:05

close 用于关闭连接。

```
close(): void
```

## 返回

| 类型 | 描述 |
| --- | --- |
| void | 无返回内容 |

## 样例

关闭连接：

```
import socket from "pts/socket";
import util from 'pts/util';

export default function () {
    const tcp_socket = new socket.Conn('tcp', '127.0.0.1', 80);
    const send_data = `GET /get HTTP/1.1
Host: 127.0.0.1
\n`;
    tcp_socket.send(util.toArrayBuffer(send_data));
    tcp_socket.recv(512);
    tcp_socket.close();
}
```