腾讯云可观测平台

云压测

腾讯云

文档目录

云压测
云压测概述

最近更新时间：2024-11-08 15:33:22

云压测（Performance Testing Service, PTS）是一款分布式性能测试服务，可模拟海量用户的真实业务场景，全方位验证系统可用性和稳定性。支持按需发起压测任务，提供百万并发多地域流量发起能力。提供流量录制、场景编排、流量定制、高级脚本定制等功能，可快速根据业务模型定义压测场景，真实还原应用大规模业务访问场景，帮助用户提前识别应用性能问题。

产品背景

- 从逻辑复杂的大型单体服务到简单模块化的微服务，每个后台应用搭载的业务逻辑逐步简化，但整个分布式后台的系统结构却变得更加复杂。企业越来越重视如何保证整个系统的服务可用性。
- 在企业迅速增长过程或者突发流量（运营/大促活动，例如618、双11等），如何提前评估系统是否可用，稳定性如何，容量是否合理?
- 如何在快速开发迭代过程中，保持系统都是可靠运行?
- 新系统上线如何验证系统性能，及可承受的最大流量?
- 自研压测工具，成本太高? 遇到技术瓶颈无法实现?

PTS 模拟实际应用的软硬件环境及用户使用过程的系统负荷，长时间或超大负荷地运行应用系统，验证应用系统的性能、可靠性、稳定性等。

产品功能

高并发性能测试

提供百万并发多地域流量发起能力，设置不同地域用户每秒内发起的请求数。

支持根据业务模型自定义压测场景

需要发起一次性能压测，首先需要创建一个压测场景，进行业务场景编排。PTS 支持创建多种模式测试场景和多协议场景编排，您可以快速根据业务模型自定义压测场景。

专业性能测试报告

- 包括并发用户数、RPS、吞吐量、响应时延、请求总数等多维度统计，客观反映用户体验。
- 支持自定义可视化展示压测性能指标。
- 支持多指标历史数据对比，排查应用性能问题。

产品优势

配置灵活

提供灵活的压测场景配置，您可以通过 UI 配置压测用例，也可以通过脚本编写复杂的组合场景。

海量流量施压

依托于云服务算力，您可以动态调整流量，PTS 最大支持百万并发流量发起。

多协议支持

支持 HTTP、WebSocket、gRPC 等协议压测场景编排，可根据业务模型定义压测场景。

流量地域定制

可按比例分配流量，支持指定腾讯云全球地域同时发起施压流量，更加真实反映用户在各地域的体验。

资源按需分配

您无需提前准备压测资源，系统会根据用户的施压配置模式，自动计算资源用量，进行动态伸缩，按需创建压测资源。随用随建，为您节约测试成本。

专业报告分析

提供专业性能测试报告，系统将会进行 RPS 吞吐量、响应时延、资源使用等多维度统计，您也可以基于历史报告设定基线，进行多维度对比分析。

监控及 SLA 防护

支持根据压测 SLA 自动启停压测，消息触达通知，结合云上监控工具（例如：应用性能监控等），方便您实时分析压测数据，排除性能问题。

控制台操作指南
简单模式压测

最近更新时间：2024-08-20 16:12:31

前言

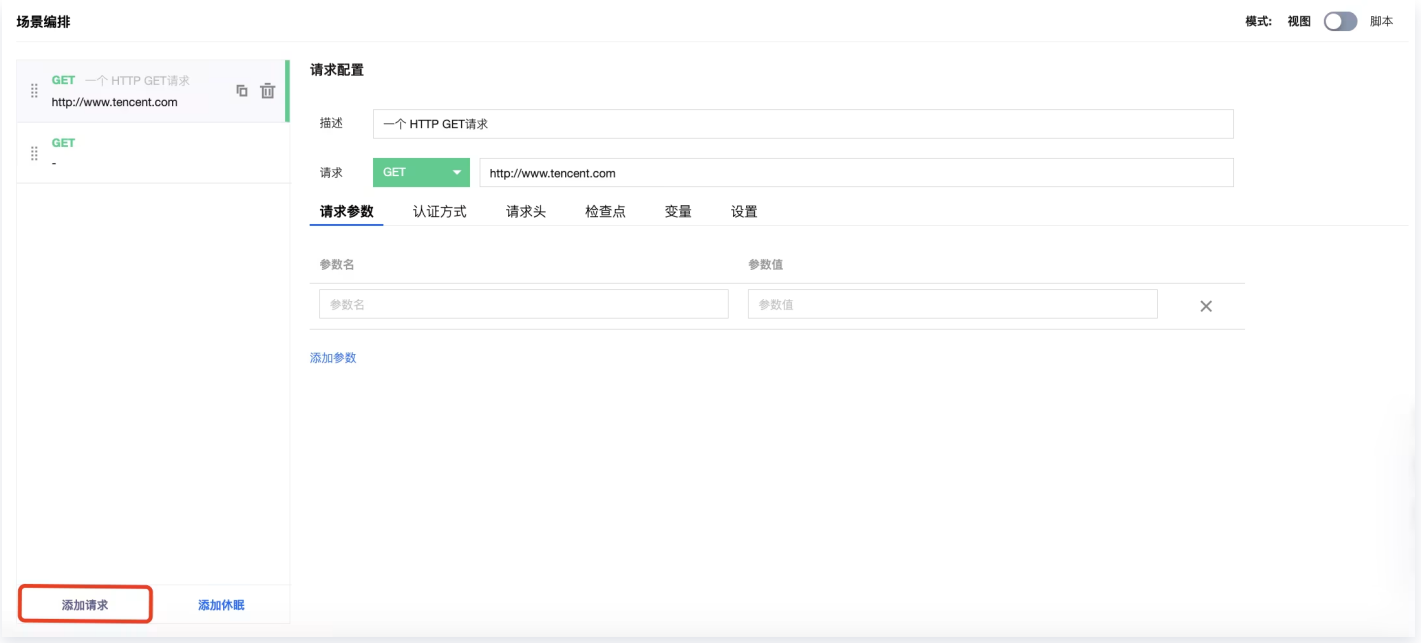
简单模式压测主要使用交互式 UI 组合 GET、POST、PUT、PATCH、 DELETE 等请求来压测场景，本文将详细介绍它的基本用法。

创建简单模式压测场景

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**云压测 > 测试场景**。
3. 在测试场景页面单击**新建场景**。
4. 在创建测试场景页面选择“简单模式”压测类型，并单击**开始**，创建压测场景。创建完成后，可进行下列操作。

添加 HTTP 请求

在场景编排模块，输入请求描述和请求地址，即可添加 HTTP 请求。



构建 HTTP 请求

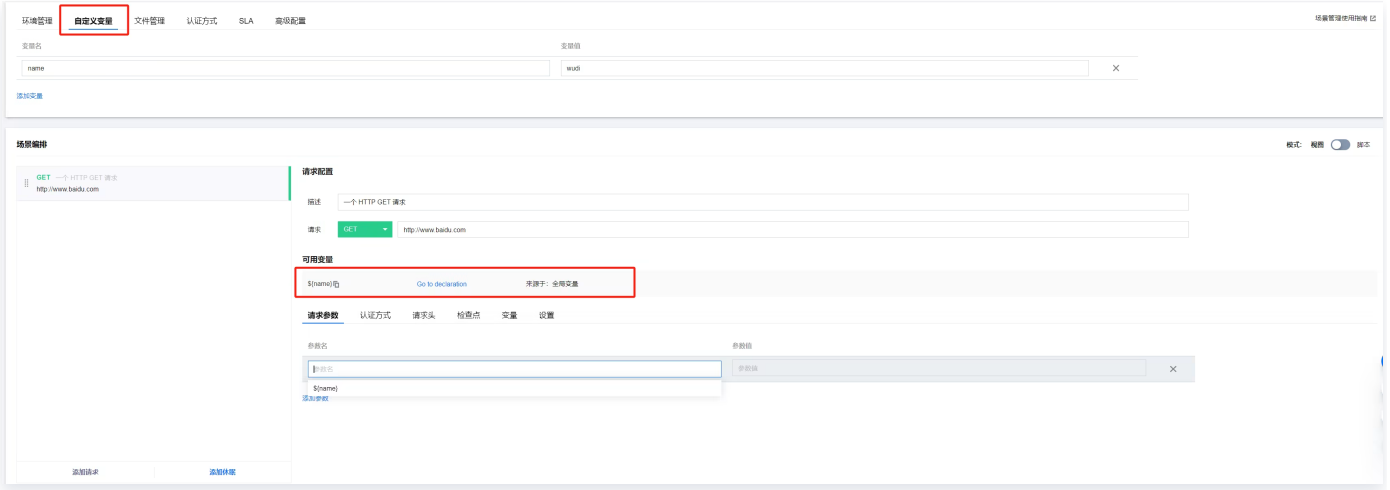
基本信息

以一个 HTTP Get 请求为例，您可以配置其 URL、请求参数（URL query string）、认证方式、请求头、检查点等。

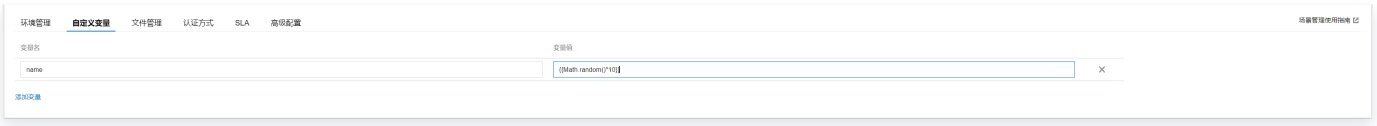
使用变量

在构建请求时，除了直接输入数据，您也可以先创建出“可用变量”，然后再在请求中引用该变量。
您可使用以下几种类型的变量：

- **自定义变量**：在请求的“自定义变量”中输入变量名和变量值，则新建请求后，请求的“可用变量”栏会展示该变量，供您在需要输入参数的地方，以 `${xx}` 的形式引用。



在设置自定义变量值时，除了直接输入数据，还可以用 `{{xx}}` 的形式，由函数计算得来。支持原生 JS 对象及函数调用，具体可参考 JS 原生内置对象 。例如引用 Math 生成随机数，如下图所示：



- **从参数文件中获取**：上传 csv 文件，并从中获取“可用变量”，供您在需要输入参数的地方，以 `${xx}` 的形式引用。关于参数文件的上传和使用，请参见 使用参数文件 。
- **从前序请求中获取**：可以从前序请求的响应中提取相关字段，生成可用变量，在后序请求的参数中引用该变量。
  例如：第一个请求的响应体为 JSON 格式，响应内容如下：

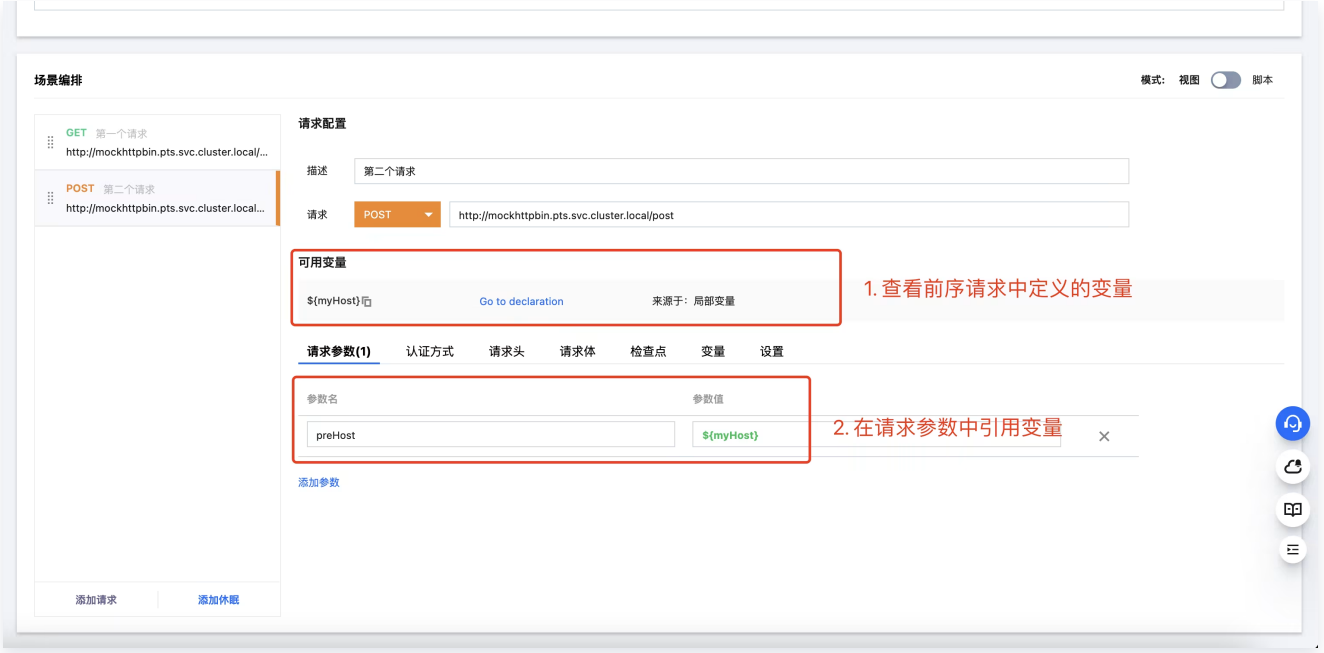
```
{
"args": {},
"headers": {
```

```
    "Accept": "*/*",
    "Accept-Encoding": "gzip",
    "Connection": "keep-alive",
    "Host": "mockhttpbin.pts.svc.cluster.local",
    "User-Agent": "PTSEngine",
    "X-Pts-Request-Id": "a19df018-555c-45a3-9eae-cc3cfc1d539a"
  },
  "origin": "127.0.0.1",
  "url": "http://mockhttpbin.pts.svc.cluster.local/get"
}
```

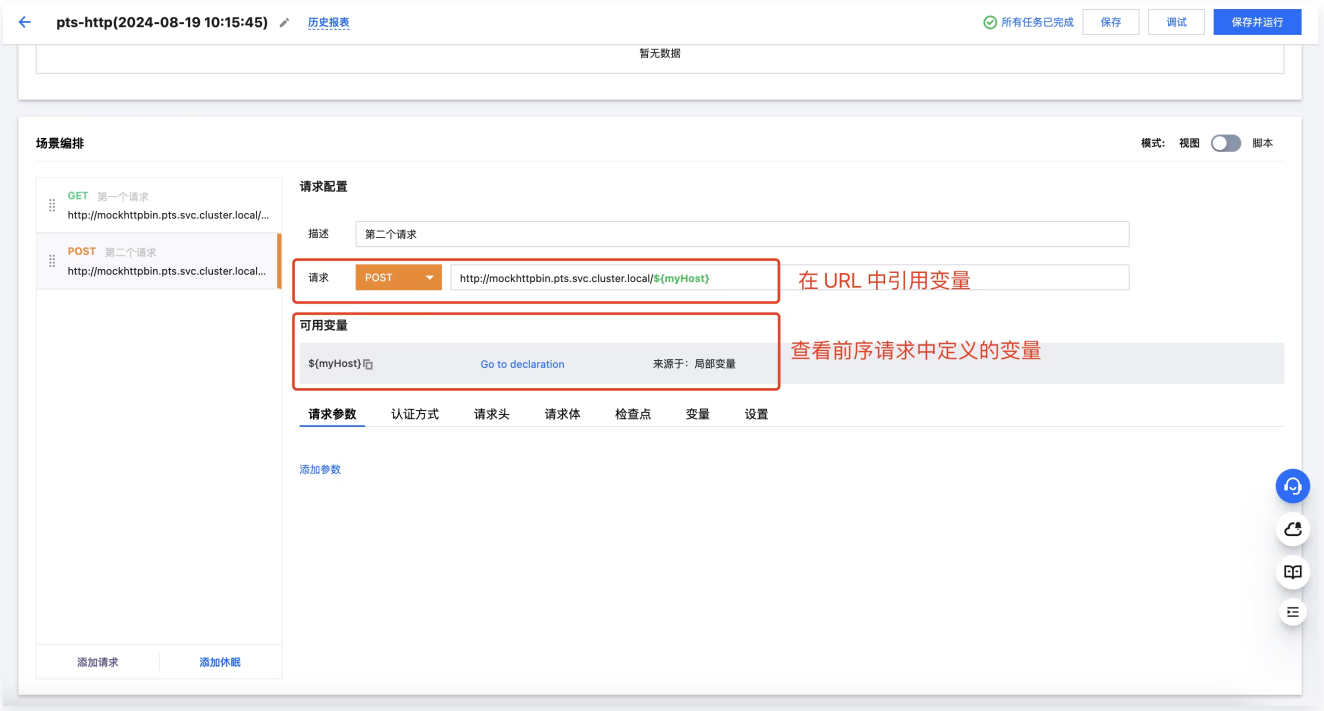
提取响应体中的 headers.Host，作为一个变量，变量名为 myHost，在下一个请求中使用。

则在后序请求中，即可以在请求的任意地方，以 `${xx}` 的形式引用该变量。例如，在请求参数与请求 URL 中引用变量：

- **在请求参数中引用变量：**



- 在请求 URL 中引用变量：



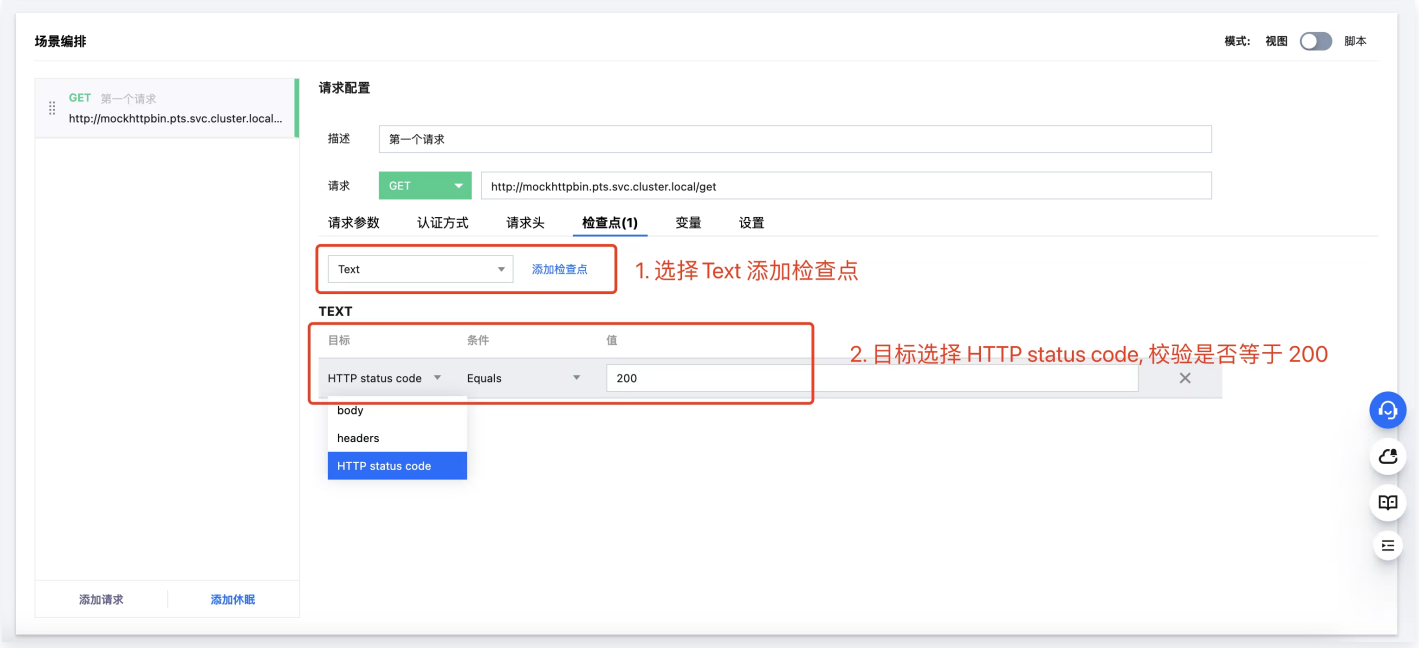
检查点配置

通过检查点可以校验请求响应内容是否符合预期。例如有一个请求，response status code 为 200， 响应内容如下：

```
{
  "args": {},
  "headers": {
```

```
    "Accept": "*/*",
    "Accept-Encoding": "gzip",
    "Connection": "keep-alive",
    "Host": "mockhttpbin.pts.svc.cluster.local",
    "User-Agent": "PTSEngine",
    "X-Pts-Request-Id": "a19df018-555c-45a3-9eae-cc3cfc1d539a"
  },
  "origin": "127.0.0.1",
  "url": "http://mockhttpbin.pts.svc.cluster.local/get"
}
```

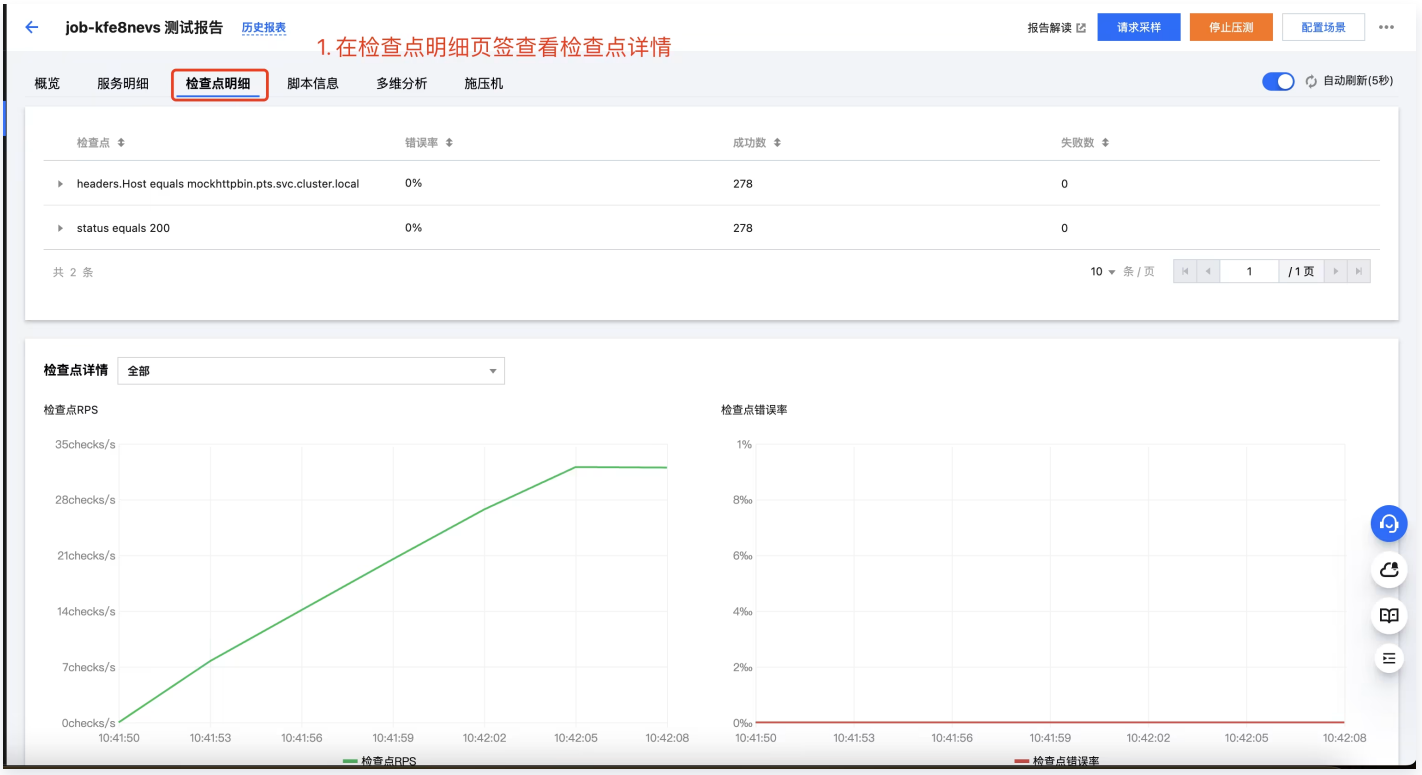
检查 response status code 是否为 200：



检查响应体某个字段是否符合预期：

如果检查响应体为 JSON 格式，校验响应体的 headers.Host 字段是否等于 "mockhttpbin.pts.svc.cluster.local"

单击右上角保存并运行启动压测后，在生成的压测历史报告中，您就可以观测到检查点的情况：



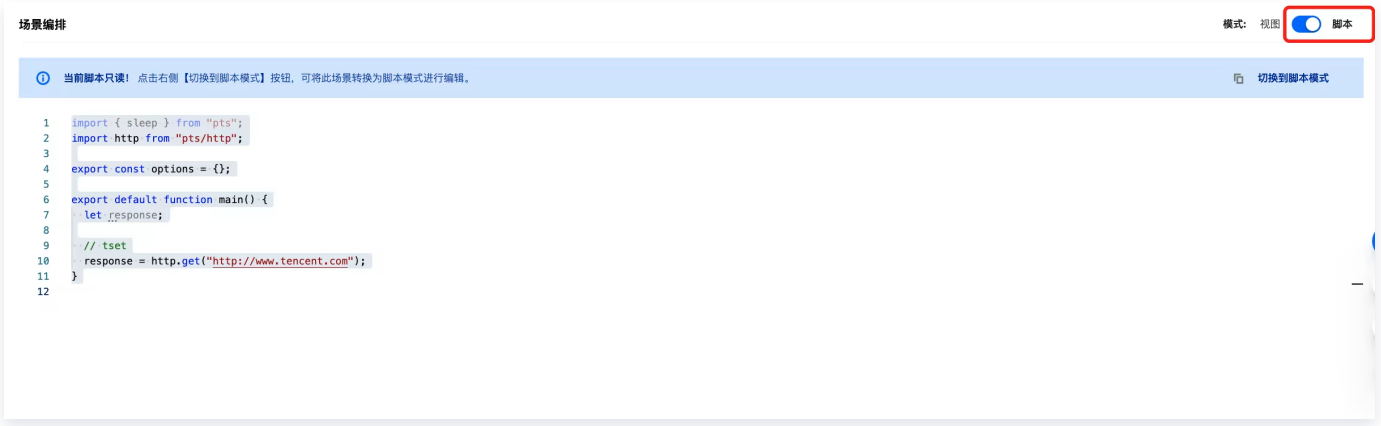
文件依赖

在压测场景里，您可上传以下几种类型的文件，提供压测执行时的状态数据：

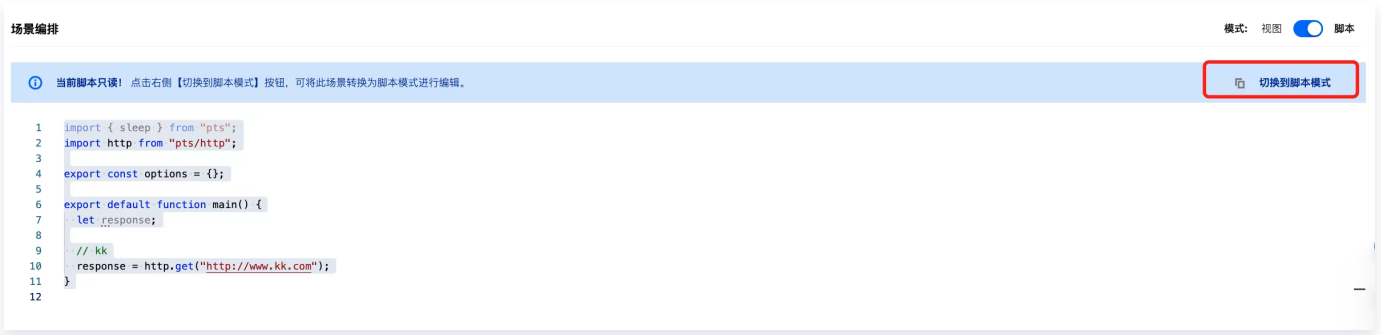
- 参数文件：以 csv 文件的形式，动态提供测试数据。也即，场景被每个并发用户（VU）执行时，会获取参数文件里的每行数据，作为测试数据的值，供脚本里的变量引用。具体使用方法参见：使用参数文件 。
- 请求文件：构建您的请求所需的文件，如需要上传的文件。具体使用方法参见：使用请求文件 。
- 协议文件：请求序列化所需要用到的文件。具体使用方法参见：使用协议文件 。

模式切换

- 若需看到场景详情，您可一键切换到脚本视图，该视图为只读模式：



- 若需直接修改脚本，可单击**切换到脚本模式**，编辑脚本内容。



> **注意：**
> 切换到脚本模式后，无法再回退到简单模式。

脚本模式压测
脚本概述

最近更新时间：2024-07-05 11:48:11

PTS 兼容 JavaScript ES2015(ES6)+ 语法，并提供额外函数，帮助您在脚本模式下，快速编排压测场景。
您可在控制台的在线编辑器里，用 JavaScript 代码描述您的压测场景所需的请求编排、变量定义、结果断言、通用函数等逻辑。（详细的 API 文档请参见：PTS API ）
PTS 还提供了各种类型的脚本模板（如各种协议的常见用法、以及一些常用函数等），在控制台脚本编辑器右侧的**脚本常用模板示例**里，供您参考以编写自己的脚本。

脚本结构

一个 PTS 场景脚本可由导入依赖模块、定义全局变量、定义全局选项 Options、定义函数、定义检查点组成，详细脚本内容请参考下文。

导入依赖模块

将所需模块导入后，才能使用模块中的 API。

定义全局变量

如果需要全局变量，可定义在函数外部。例如：

```
const globalVar = "var"
const globalObj = {
    "k": "v",
}

export default function () {
    console.log(globalVar); // var
    console.log(globalObj.k); // v
};
```

定义全局选项 Options

通过全局选项您可以控制引擎的默认行为。

```
export const option = {
   http: {
       http2: true,
       maxIdleConns: 50,
       basicAuth: {
            username: 'user',
            password: 'passwd',
        }
   },
   tlsConfig: {
        'localhost': {
            insecureSkipVerify: false,
            //需要用户在场景中上传请求文件ca.crt
            rootCAs: [open('ca.crt')],
            //需要用户在场景中上传请求文件client.crt, client.key
            certificates: [{cert: open('client.crt'), key:
open('client.key')}]
        }
    }
}
```

定义函数

每个并发用户（VU）每次迭代执行的逻辑，定义在主函数（default 函数）里。
除了主函数，您还可以定义预处理（setup）和后处理（teardown）函数，示例如下：

- 预处理函数在每次压测开始后运行一次。
- 后处理函数在每次压测结束前运行一次。

```
// 全局变量，定义在函数外
const global = { stage: "global" };

// 用 setup 函数做预处理，可返回自定义的键值对
export function setup() {
    return { stage: "setup" };
}

// 主函数（入参可接收 setup 函数返回的键值对）
export default function(data) {
    console.log(JSON.stringify(global)); // {"stage":"global"}
```

```
    console.log(JSON.stringify(data)); // {"stage":"setup"}
}

// 用 teardown 函数做后处理
export function teardown(data) {
    console.log(JSON.stringify(global)); // {"stage":"global"}
    console.log(JSON.stringify(data)); // {"stage":"setup"}
}
```

定义检查点

配置检查点可以从业务角度判断请求是否成功。

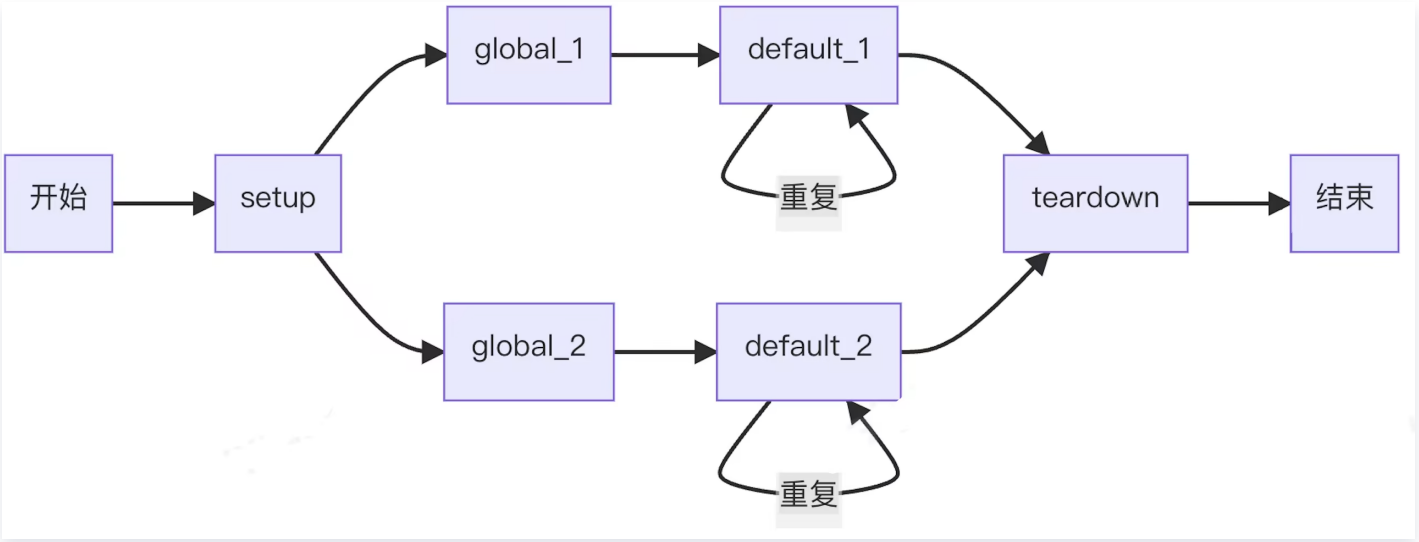
```
import http from 'pts/http';
import { check } from 'pts';

export default function () {
  // get request with headers and parameters
  const resp1 = http.get('http://httpbin.org/get', {
    headers: {
      Connection: 'keep-alive',
      'User-Agent': 'pts-engine',
    },
    query: {
      name1: 'value1',
      name2: 'value2',
    },
  });
  console.log(resp1.json().args.name1); // 'value1'
  check('status is 200', () => resp1.statusCode === 200);
  check('body.args.name1 equals value1', () => resp1.json().args.name1
=== 'value1');
}
```

生命周期

- 预处理（setup）和后处理（teardown）函数：每台压测机运行一次。
- 定义全局变量（global）的代码：每个 VU 运行一次。一些静态的文件读取等操作建议放到 global 中定义，这样一个并发仅需读取一次文件。避免场景迭代重复打开文件。

- 主函数（default）代码：每个 VU 的每次迭代运行一次，且每个 VU 在达到本次压测配置的时长上限或迭代上限之前，会持续不断地迭代执行。

例如：在一台压测机上，当有两个 VU 时的流程图如下：



说明：
关于 VU 的概念介绍，请参见 常见问题 。

脚本示例
基础语法

最近更新时间：2025-12-01 16:56:01

本文档介绍在云压测脚本模式中常用的 JavaScript 基础语法，包括变量声明、条件语句、错误处理（try/catch）、循环语句和数组操作等。这些基础语法是编写压测脚本的基础，掌握这些内容有助于构建更强大的压测脚本。

变量声明

在压测脚本中，可以使用 `var` 、 `let` 或 `const` 声明变量：

- `var`：函数作用域变量（不推荐使用）。
- `let`：块作用域变量，可以重新赋值。
- `const`：块作用域常量，声明后不能重新赋值。

使用前提

- 所有代码需放在 `export default function()` 函数中执行。
- 变量名需符合 JavaScript 标识符规范。
- 建议使用 `const` 声明不会改变的变量，使用 `let` 声明需要重新赋值的变量。

示例

示例1：变量声明和使用

本示例演示如何声明和使用变量，适用于需要在脚本中存储和操作数据的场景。

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

export default function () {
    // 使用 const 声明常量
    const baseUrl = 'http://example.com/api';
    const apiKey = 'your-api-key';

    // 使用 let 声明可变变量
    let requestCount = 0;
    let lastResponse = null;

    // 使用变量构建请求
    const resp = http.get(`${baseUrl}/users`, {
```

```
        headers: {
            'Authorization': `Bearer ${apiKey}`
        }
    });

    // 更新变量值
    requestCount++;
    lastResponse = resp;

    console.log(`请求次数: ${requestCount}`);
    console.log(`响应状态: ${lastResponse.statusCode}`);
}
```

示例2：变量作用域

本示例演示变量的作用域，适用于需要理解变量可见性的场景。

```
export default function () {
    // 函数作用域变量
    const globalVar = 'I am global in this function';

    if (true) {
        // 块作用域变量
        const blockVar = 'I am in block scope';
        let mutableVar = 'I can be changed';

        console.log(globalVar); // 可以访问
        console.log(blockVar); // 可以访问

        mutableVar = 'Changed value';
    }

    // console.log(blockVar); // 错误：blockVar 不可访问
    // console.log(mutableVar); // 错误：mutableVar 不可访问
}
```

条件语句

函数说明

条件语句用于根据不同的条件执行不同的代码，在压测脚本中常用于：

- 根据响应状态码执行不同逻辑。
- 根据数据内容进行分支处理。
- 实现错误处理和重试机制。
- 控制脚本执行流程。

使用前提

- 所有代码需放在 `export default function()` 函数中执行。
- 条件表达式需返回布尔值或可转换为布尔值的值。

语法说明

- `if (condition) { ... }`：单条件判断。
- `if (condition) { ... } else { ... }`：双分支判断。
- `if (condition1) { ... } else if (condition2) { ... } else { ... }`：多分支判断。
- `switch (value) { case ...: ... break; default: ... }`：多值判断。

示例

示例1：基本条件判断

本示例演示如何使用 if/else 进行条件判断，适用于根据响应状态执行不同逻辑的场景。

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

export default function () {
    const resp = http.get('http://example.com/api/user');

    // 根据状态码执行不同逻辑
    if (resp.statusCode === 200) {
        console.log('请求成功');
        const data = resp.json();
        console.log('用户数据:', data);
    } else if (resp.statusCode === 404) {
        console.log('用户不存在');
    } else if (resp.statusCode === 401) {
        console.log('未授权，需要登录');
    } else {
        console.log('请求失败，状态码:', resp.statusCode);
```

```
    }

    // 使用三元运算符
    const message = resp.statusCode === 200 ? '成功' : '失败';
    console.log('请求结果:', message);
}
```

示例2：使用 switch 语句

本示例演示如何使用 switch 语句处理多个值的情况，适用于需要根据特定值执行不同操作的场景。

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

export default function () {
    const resp = http.get('http://example.com/api/status');
    const status = resp.json().status;

    switch (status) {
        case 'active':
            console.log('状态: 活跃');
            // 执行活跃状态相关操作
            break;
        case 'inactive':
            console.log('状态: 非活跃');
            // 执行非活跃状态相关操作
            break;
        case 'pending':
            console.log('状态: 待处理');
            // 执行待处理状态相关操作
            break;
        default:
            console.log('未知状态:', status);
    }
}
```

示例3：复杂条件判断

本示例演示如何进行复杂的条件判断，适用于需要组合多个条件的场景。

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

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

export default function () {
    const resp = http.get('http://example.com/api/data');

    // 组合多个条件
    if (resp.statusCode === 200 && resp.body) {
        const data = resp.json();

        if (data.success === true && data.items && data.items.length >
0) {
            console.log('数据获取成功，共', data.items.length, '条记录');
        } else {
            console.log('数据为空或格式不正确');
        }
    } else {
        console.log('请求失败或响应为空');
    }

    // 使用逻辑运算符
    const isValid = resp.statusCode >= 200 && resp.statusCode < 300;
    const hasData = resp.body && resp.body.length > 0;

    if (isValid && hasData) {
        check('响应有效', () => true);
    }
}
```

错误处理（try/catch）

函数说明

`try/catch` 语句用于捕获和处理代码执行过程中可能出现的错误，在压测脚本中常用于：

- 捕获 JSON 解析错误。
- 处理网络请求异常。
- 处理数据格式错误。
- 实现优雅的错误处理和日志记录。
- 防止脚本因单个错误而中断执行。

使用前提

- 所有代码需放在 `export default function()` 函数中执行
- `try` 块中可能抛出错误的代码。
- `catch` 块用于捕获和处理错误。

语法说明

- `try { ... } catch (error) { ... }`：基本错误捕获
- `error.message`：获取错误消息。
- `error.stack`：获取错误堆栈信息（如果可用）。

示例

示例1：捕获 JSON 解析错误

本示例演示如何使用 try/catch 捕获 JSON 解析错误，适用于处理可能格式不正确的响应数据。

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

export default function () {
    const resp = http.get('http://example.com/api/data');

    try {
        // 尝试解析 JSON，如果格式不正确会抛出异常
        const data = JSON.parse(resp.body);
        console.log('解析成功:', data);

        // 使用解析后的数据
        if (data.items) {
            console.log('数据项数量:', data.items.length);
        }
    } catch (error) {
        // 捕获解析错误
        console.error('JSON 解析失败:', error.message);
        console.log('原始响应体:', resp.body);
    }
}
```

示例2：处理 HTTP 请求错误

本示例演示如何使用 try/catch 处理 HTTP 请求可能出现的错误，适用于需要优雅处理网络异常的场景。

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

export default function () {
    try {
        const resp = http.get('http://example.com/api/user');

        if (resp.statusCode === 200) {
            const userData = resp.json();
            console.log('用户信息:', userData);
        } else {
            console.log('请求失败，状态码:', resp.statusCode);
        }
    } catch (error) {
        console.error('请求处理出错:', error.message);
        // 可以在这里实现重试逻辑或记录错误日志
    }
}
```

示例3：多层错误处理

本示例演示如何进行多层错误处理，适用于需要区分不同类型错误的场景。

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

export default function () {
    try {
        const resp = http.get('http://example.com/api/data');

        try {
            const data = resp.json();

            try {
                // 访问可能不存在的嵌套属性
                const value = data.user.profile.email;
                console.log('邮箱:', value);
            } catch (error) {
                console.error('访问嵌套属性失败:', error.message);
                // 使用默认值
                console.log('使用默认邮箱');
```

```
            }
        } catch (error) {
            console.error('JSON 解析失败:', error.message);
        }
    } catch (error) {
        console.error('请求失败:', error.message);
    }
}
```

示例4：错误处理和重试机制结合

本示例演示如何将错误处理与重试机制结合，适用于需要自动重试失败请求的场景。

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

export default function () {
    const maxRetries = 3;
    let lastError = null;

    for (let attempt = 1; attempt <= maxRetries; attempt++) {
        try {
            console.log(`尝试第 ${attempt} 次请求`);

            const resp = http.get('http://example.com/api/data');

            if (resp.statusCode === 200) {
                const data = resp.json();
                console.log('请求成功:', data);
                lastError = null; // 清除错误
                break; // 成功则退出循环
            } else {
                // 状态码不是 200，记录错误信息
                lastError = new Error(`HTTP 状态码: ${resp.statusCode}`);
                console.error(`第 ${attempt} 次尝试失败:`,
lastError.message);
            }
        } catch (error) {
            lastError = error;
            console.error(`第 ${attempt} 次尝试失败:`, error.message);
```

```
        }

        if (lastError && attempt < maxRetries) {
            console.log(`等待 1 秒后重试...`);
            sleep(1);
        }
    }

    if (lastError) {
        console.error('所有重试均失败，最后错误:', lastError.message);
    }
}
```

循环语句

函数说明

循环语句用于重复执行代码，在压测脚本中常用于：

- 遍历数组或对象。
- 批量发送请求。
- 实现重试机制。
- 处理列表数据。

使用前提

- 所有代码需放在 `export default function()` 函数中执行。
- 循环条件需能正确终止，避免无限循环。
- 注意循环性能，避免在循环中进行耗时操作。

语法说明

- `for (init; condition; increment) { ... }`：传统 for 循环。
- `for (item of array) { ... }`：for...of 循环（遍历数组）。
- `for (key in object) { ... }`：for...in 循环（遍历对象）。
- `while (condition) { ... }`：while 循环。
- `array.forEach((item, index) => { ... })`：数组 forEach 方法。

示例

示例1：for 循环遍历数组

本示例演示如何使用 for 循环遍历数组，适用于需要逐个处理数组元素的场景。

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

export default function () {
    // 定义用户 ID 数组
    const userIds = ['1001', '1002', '1003', '1004', '1005'];

    // 使用传统 for 循环
    for (let i = 0; i < userIds.length; i++) {
        const userId = userIds[i];
        const resp = http.get(`http://example.com/api/users/${userId}`);
        console.log(`用户 ${userId} 信息:`, resp.json());
    }
}
```

示例2：for...of 循环

本示例演示如何使用 for...of 循环遍历数组，适用于需要简洁语法的场景。

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

export default function () {
    const endpoints = [
        '/api/users',
        '/api/products',
        '/api/orders'
    ];

    // 使用 for...of 循环
    for (const endpoint of endpoints) {
        const resp = http.get(`http://example.com${endpoint}`);
        console.log(`请求 ${endpoint}，状态码:`, resp.statusCode);
        sleep(1); // 每次请求间隔 1 秒
    }
}
```

示例3：forEach 方法

本示例演示如何使用 forEach 方法遍历数组，适用于需要对每个元素执行操作的场景。

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

export default function () {
    const products = [
        { id: 1, name: 'Product A' },
        { id: 2, name: 'Product B' },
        { id: 3, name: 'Product C' }
    ];

    // 使用 forEach 方法
    products.forEach((product, index) => {
        console.log(`处理产品 ${index + 1}:`, product.name);

        const resp = http.post('http://example.com/api/products', {
            body: JSON.stringify(product)
        });

        console.log(`产品 ${product.id} 创建结果:`, resp.statusCode);
    });
}
```

示例4：while 循环实现重试

本示例演示如何使用 while 循环实现重试机制，适用于需要重试失败请求的场景。

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

export default function () {
    let retryCount = 0;
    const maxRetries = 3;
    let resp = null;

    // 使用 while 循环实现重试
    while (retryCount < maxRetries) {
        resp = http.get('http://example.com/api/data');

        if (resp.statusCode === 200) {
            console.log('请求成功');
```

```
            break; // 成功则退出循环
        }

        retryCount++;
        console.log(`请求失败，第 ${retryCount} 次重试`);
        sleep(1); // 等待 1 秒后重试
    }

    if (resp && resp.statusCode !== 200) {
        console.log('重试失败，最终状态码:', resp.statusCode);
    }
}
```

数组操作

函数说明

数组是 JavaScript 中常用的数据结构，在压测脚本中常用于：

- 存储多个数据项。
- 批量处理数据。
- 参数化测试数据。
- 收集和处理响应数据。

使用前提

- 所有代码需放在 `export default function()` 函数中执行。
- 数组索引从0开始。
- 数组可以包含任意类型的元素。

常用方法

- `array.length`：获取数组长度。
- `array.push(item)`：向数组末尾添加元素。
- `array.pop()`：移除并返回数组最后一个元素。
- `array.shift()`：移除并返回数组第一个元素。
- `array.unshift(item)`：向数组开头添加元素。
- `array.map(fn)`：映射数组元素。
- `array.filter(fn)`：过滤数组元素。
- `array.find(fn)`：查找数组元素。
- `array.includes(item)`：检查数组是否包含元素。

示例

示例1：数组声明和基本操作

本示例演示如何声明数组并进行基本操作，适用于需要存储和操作多个数据的场景。

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

export default function () {
    // 声明数组的多种方式
    const numbers = [1, 2, 3, 4, 5];
    const strings = ['apple', 'banana', 'orange'];
    const mixed = [1, 'hello', true, { key: 'value' }];

    // 访问数组元素
    console.log('第一个数字:', numbers[0]); // 输出: 1
    console.log('数组长度:', numbers.length); // 输出: 5

    // 修改数组元素
    numbers[0] = 10;
    console.log('修改后的数组:', numbers);

    // 添加元素
    numbers.push(6);
    console.log('添加元素后:', numbers);
}
```

示例2：数组遍历和处理

本示例演示如何遍历数组并处理每个元素，适用于批量处理数据的场景。

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

export default function () {
    // 定义用户 ID 数组
    const userIds = ['1001', '1002', '1003'];
    const results = [];

    // 遍历数组并处理
    for (const userId of userIds) {
```

```
        const resp = http.get(`http://example.com/api/users/${userId}`);

        if (resp.statusCode === 200) {
            const userData = resp.json();
            results.push({
                userId: userId,
                name: userData.name,
                status: 'success'
            });
        } else {
            results.push({
                userId: userId,
                status: 'failed',
                statusCode: resp.statusCode
            });
        }
    }

    console.log('处理结果:', results);
    console.log('成功数量:', results.filter(r => r.status ===
'success').length);
}
```

示例3：数组方法的使用

本示例演示如何使用数组的高级方法，适用于需要转换、过滤和查找数据的场景。

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

export default function () {
    // 原始数据
    const products = [
        { id: 1, name: 'Product A', price: 100 },
        { id: 2, name: 'Product B', price: 200 },
        { id: 3, name: 'Product C', price: 150 }
    ];

    // 使用 map 转换数组
    const productNames = products.map(p => p.name);
    console.log('产品名称:', productNames);
```

```
    // 使用 filter 过滤数组
    const expensiveProducts = products.filter(p => p.price > 150);
    console.log('高价产品:', expensiveProducts);

    // 使用 find 查找元素
    const product = products.find(p => p.id === 2);
    console.log('找到的产品:', product);

    // 使用 includes 检查元素
    const hasProductA = productNames.includes('Product A');
    console.log('是否包含 Product A:', hasProductA);
}
```

示例4：动态构建数组

本示例演示如何动态构建数组，适用于需要根据条件或循环生成数据的场景。

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

export default function () {
    // 动态生成用户 ID 数组
    const userIds = [];
    for (let i = 1; i <= 10; i++) {
        userIds.push(`user_${i}`);
    }
    console.log('生成的用户 ID:', userIds);

    // 从响应中提取数据构建数组
    const resp = http.get('http://example.com/api/users');
    const users = resp.json().users || [];

    // 提取用户 ID 数组
    const extractedIds = users.map(user => user.id);
    console.log('提取的用户 ID:', extractedIds);

    // 使用数组存储请求结果
    const requestResults = [];
    for (const userId of userIds.slice(0, 5)) { // 只处理前 5 个
```

```
        const userResp =
http.get(`http://example.com/api/users/${userId}`);
        requestResults.push({
            userId: userId,
            statusCode: userResp.statusCode,
            success: userResp.statusCode === 200
        });
    }

    console.log('请求结果统计:', {
        total: requestResults.length,
        success: requestResults.filter(r => r.success).length,
        failed: requestResults.filter(r => !r.success).length
    });
}
```

HTTP

最近更新时间：2025-12-01 16:56:01

本文档提供了在云压测脚本模式中使用 HTTP 协议进行压测的完整示例。所有示例均基于 pts/http 模块提供的 HTTP 请求方法。

前置条件

在使用 HTTP 请求功能前，请确保：

- **导入模块**：在脚本开头使用 `import http from 'pts/http'` 导入 HTTP 模块。
- **脚本结构**：所有请求代码需放在 `export default function()` 函数中执行。
- **URL 格式**：确保目标 URL 格式正确，支持 `http://` 和 `https://` 协议。
- **网络连通性**：确保压测环境能够访问目标服务器。

注意事项

- **错误处理**：建议使用 `check()` 函数验证响应状态码和内容，确保请求成功。
- **超时设置**：对于可能响应较慢的接口，建议设置合理的 `timeout` 值。
- **响应体大小**：如果响应体很大且不需要检查内容，可以设置 `discardResponseBody: true` 以节省内存。
- **Content-Type**：发送 POST/PUT/PATCH 请求时，务必根据请求体格式设置正确的 `Content-Type`。
- **文件上传**：使用 FormData 上传文件时，文件必须提前上传到压测场景中，并使用 `open()` 函数读取。
- **并发控制**：批量请求时，注意合理设置 `parallel` 值，避免对目标服务器造成过大压力。
- **URL 编码**：查询参数和表单数据会自动进行 URL 编码，无需手动编码。
- **重定向**：默认会跟随重定向，可以通过 `maxRedirects` 控制最大重定向次数。

HTTP GET 请求

函数说明

`http.get()` 用于发起 HTTP GET 请求，适用于获取资源、查询数据等场景。该方法在脚本模式的压测场景中使用，支持设置请求头、查询参数等配置。

函数签名

```
http.get(url: string, options?: Request): Response
```

参数说明

- **url** (string，必填)：目标请求的完整 URL 地址，支持 `http://` 和 `https://` 协议。
- **options** (Request，可选)：请求配置对象，包含以下可选字段：

- `headers` (Record<string, string>)：自定义请求头，如 `{'User-Agent': 'pts-engine', 'Connection': 'keep-alive'}` 。
- `query` (Record<string, string>)：URL 查询参数，会自动拼接到 URL 后面。
- `timeout` (number)：请求超时时间，单位为毫秒，包括连接时间、任何重定向和读取响应正文的时间。
- `maxRedirects` (number)：最大重定向跳转次数。
- `discardResponseBody` (boolean)：是否丢弃响应体，适用于响应体太大且不需要进行 check 的场景。
- `service` (string)：服务标识，用于在报表中将不同 URL 的请求归类到同一个服务下。
- `basicAuth` (BasicAuth)：基础鉴权配置。
- `chunked` (function)：分块传输回调函数，函数签名为 `(body: string) => void` ，当数据以一系列分块的形式进行发送时，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。

返回值说明

返回 Response 对象，包含以下属性：

- `statusCode` (number)：HTTP 状态码，如200、404、500等。
- `status` (string)：HTTP 状态消息，如 "200 OK"。
- `body` (string)：响应体内容，原始字符串格式。
- `headers` (Record<string, string>)：响应头信息。
- `contentLength` (number)：服务器响应体长度。
- `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。
- `json()` (方法)：将响应体反序列化为 JSON 对象，仅当响应体为有效的 JSON 字符串时使用。

使用限制

- URL 必须包含完整的协议前缀（ `http://` 或 `https://` ）。
- 查询参数值必须是字符串类型。
- 如果响应体不是有效的 JSON 格式，调用 `json()` 方法会抛出异常。

示例

示例1：简单 GET 请求

本示例演示如何发起最简单的 GET 请求，并检查响应状态码和解析 JSON 响应体。

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

export default function () {
    // 发起简单的 GET 请求
    const resp1 =
http.get('http://mockhttpbin.pts.svc.cluster.local/get');

    // 输出原始响应体
    console.log(resp1.body);

    // 如果响应体是 JSON 格式，使用 json() 方法转换为对象
    console.log(resp1.json());

    // 检查响应状态码是否为 200
    check('status is 200', () => resp1.statusCode === 200);

    // 等待 1 秒
    sleep(1);
}
```

示例2：带请求头和查询参数的 GET 请求

本示例演示如何设置自定义请求头和 URL 查询参数，适用于需要传递特定头部信息或查询条件的场景。

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

export default function () {
    // GET 请求，包含自定义请求头和查询参数
    const resp2 =
http.get('http://mockhttpbin.pts.svc.cluster.local/get', {
        headers: {
            'Connection': 'keep-alive',
            'User-Agent': 'pts-engine',
            'Accept': 'application/json'
        },
        query: {
            'name1': 'value1',
```

```
            'name2': 'value2',
        }
    });

    // 从 JSON 响应中提取查询参数值
    console.log(resp2.json().args.name1); // 输出: 'value1'

    // 验证响应中的查询参数值
    check('body.args.name1 equals value1', () => resp2.json().args.name1
=== 'value1');
}
```

示例 3：带超时和重定向配置的 GET 请求

本示例演示如何设置请求超时时间和最大重定向次数，适用于需要控制请求行为的场景。

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

export default function () {
    const resp =
http.get('http://mockhttpbin.pts.svc.cluster.local/get', {
        headers: {
            'User-Agent': 'pts-engine'
        },
        query: {
            'page': '1',
            'size': '10'
        },
        timeout: 5000,          // 设置超时时间为 5 秒
        maxRedirects: 3         // 最大允许 3 次重定向
    });

    check('request success', () => resp.statusCode === 200);
}
```

HTTP POST 请求（JSON 格式）

函数说明

`http.post()` 用于发起 HTTP POST 请求，适用于创建资源、提交数据等场景。当请求体为对象时，默认会序列化为 JSON 格式发送。

函数签名

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

参数说明

- **url** (string，必填)：目标请求的完整 URL 地址
- **body** (string | object | Record<string, string>，可选)：请求体内容。
  - 如果为对象，会自动序列化为 JSON 字符串，并设置 `Content-Type: application/json`。
  - 如果为字符串，直接作为请求体发送。
  - 如果为 `Record<string, string>`，可以用于表单编码等场景。
- **options** (Request，可选)：请求配置对象，字段同 `http.get()` 方法，可设置 headers、query、timeout、maxRedirects、discardResponseBody、service 等。

返回值说明

返回 Response 对象，结构同 `http.get()` 方法。

使用限制

- 当 body 为对象时，系统会自动序列化为 JSON 字符串并设置 `Content-Type: application/json` 请求头，也可以手动设置。
- 如果响应体不是有效的 JSON 格式，调用 `json()` 方法会抛出异常。

示例

示例1：发送 JSON 格式的 POST 请求

本示例演示如何发送 JSON 格式的 POST 请求，适用于调用 RESTful API 创建或更新资源的场景。

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

export default function () {
    // 发送 POST 请求，body 为对象会自动序列化为 JSON
    const resp = http.post(
        'http://mockhttpbin.pts.svc.cluster.local/post',
        {
```

```
            user_id: '12345',
            username: 'testuser',
            email: 'test@example.com'
        },
        {
            headers: {
                'Content-Type': 'application/json',
            },
        }
    );

    // 解析响应 JSON 并提取数据
    console.log(resp.json().json.user_id); // 输出: 12345

    // 验证响应数据
    check('body.json.user_id equals 12345', () =>
resp.json().json.user_id === '12345', resp);
}
```

示例 2：发送字符串格式的 POST 请求

本示例演示如何发送字符串格式的请求体，适用于需要发送原始字符串数据的场景。

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

export default function () {
    // 发送字符串格式的 POST 请求
    const resp = http.post(
        'http://mockhttpbin.pts.svc.cluster.local/post',
        'raw string data',
        {
            headers: {
                'Content-Type': 'text/plain',
            },
        }
    );

    check('request success', () => resp.statusCode === 200);
    console.log(resp.json().data); // 输出: 'raw string data'
```

```
}
```

HTTP POST 请求（x-www-form-urlencoded 格式）

函数说明

当需要发送表单数据时，可以使用 `application/x-www-form-urlencoded` 格式。这种格式适用于传统的 HTML 表单提交场景。

参数说明

- **url** (string，必填)：目标请求的完整 URL 地址。
- **body** (object，必填)：表单数据对象，键值对会被编码为 URL 编码格式。
- **options** (Request，必填)：请求配置对象，必须设置 `Content-Type: application/x-www-form-urlencoded` 请求头。

使用限制

- 必须显式设置 `Content-Type: application/x-www-form-urlencoded` 请求头，否则服务器可能无法正确解析表单数据。
- body 对象中的值必须是字符串类型，非字符串值会被转换为字符串。

示例：发送表单编码的 POST 请求

本示例演示如何发送 `application/x-www-form-urlencoded` 格式的 POST 请求，适用于提交 HTML 表单数据的场景。

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

export default function () {
    // 发送表单编码格式的 POST 请求
    const resp = http.post(
        'http://mockhttpbin.pts.svc.cluster.local/post',
        {
            user_id: '12345',
            action: 'login',
            timestamp: '1234567890'
        },
        {
            headers: {
                'Content-Type': 'application/x-www-form-urlencoded',
```

```
            },
        }
    );

    // 验证 Content-Type 请求头
    console.log(resp.json().headers['Content-Type']); // 输出:
application/x-www-form-urlencoded

    // 从响应中提取表单数据
    console.log(resp.json().form.user_id); // 输出: 12345

    // 验证表单数据
    check('body.form.user_id equals 12345', () =>
resp.json().form.user_id === '12345', resp);
}
```

HTTP POST 请求（multipart/form-data 格式）

函数说明

`http.FormData` 用于构造 `multipart/form-data` 格式的请求体，适用于需要上传文件或同时发送文本和文件的场景。通过 `new http.FormData()` 创建实例，使用 `append()` 方法添加字段，最后通过 `body()` 方法获取请求体。

相关函数

- **new http.FormData()**：创建 FormData 实例。
- **formData.append(key, value)**：向 FormData 中添加字段，value 可以是字符串或 `http.file()` 返回的 File 对象。
- **formData.body()**：返回 FormData 的请求体内容（ArrayBuffer 格式），调用后不能再进行 append 操作。
- **formData.contentType()**：返回 FormData 的 Content-Type 值，包含 boundary 信息。
- **http.file(data, name?, contentType?)**：创建文件对象，用于上传文件。
  - `data`（string | ArrayBuffer，必填）：文件内容，通常使用 `open()` 函数的返回值。
  - `name`（string，可选）：文件名，默认为纳秒级时间戳。
  - `contentType`（string，可选）：文件内容类型，默认为 `application/octet-stream`。

使用限制

- 调用 `formData.body()` 后不能再调用 `append()` 方法。

- 必须使用 `formData.contentType()` 返回的值设置 `Content-Type` 请求头。
- 文件数据必须通过 `open()` 函数读取或使用 ArrayBuffer 格式。

示例

示例1：上传文本和文件

本示例演示如何同时上传文本字段和文件，适用于文件上传表单的场景。

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

// 读取文件内容（文件需提前上传到压测场景中）
const fileData = open("./sample/tmp.js");

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

    // 添加文本字段
    formData.append('data', 'some data');
    formData.append('description', 'This is a test file');

    // 添加文件字段，使用 http.file() 创建文件对象
    formData.append('file', http.file(fileData));

    // 发送 POST 请求，使用 formData.body() 作为请求体
    const resp =
http.post('http://mockhttpbin.pts.svc.cluster.local/post',
formData.body(), {
        headers: {
            'Content-Type': formData.contentType()  // 必须使用
formData.contentType() 返回的值
        }
    });

    // 验证 Content-Type 包含 multipart/form-data
    console.log(resp.json().headers['Content-Type']); // 输出:
multipart/form-data; boundary=xxxxx

    // 验证文本字段
```

```
    console.log(resp.json().form.data); // 输出: some data

    // 验证文件大小
    console.log(resp.json().files.file.length); // 输出文件字节数，如: 801

    // 检查表单数据
    check('body.form.data equals some data', () => resp.json().form.data
=== 'some data');
}
```

示例2：上传多个文件

本示例演示如何上传多个文件，适用于需要同时上传多个文件的场景。

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

const file1 = open("./sample/file1.txt");
const file2 = open("./sample/file2.jpg");

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

    // 添加多个文件，可以指定文件名和内容类型
    formData.append('document', http.file(file1, 'document.txt',
'text/plain'));
    formData.append('image', http.file(file2, 'photo.jpg',
'image/jpeg'));
    formData.append('category', 'uploads');

    const resp =
http.post('http://mockhttpbin.pts.svc.cluster.local/post',
formData.body(), {
        headers: {
            'Content-Type': formData.contentType()
        }
    });

    check('upload success', () => resp.statusCode === 200);
    console.log('Document size:', resp.json().files.document.length);
```

```
    console.log('Image size:', resp.json().files.image.length);
}
```

HTTP PUT 请求

函数说明

`http.put()` 用于发起 HTTP PUT 请求，适用于完整更新资源的场景。PUT 请求通常需要提供完整的资源数据。

函数签名

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

参数说明

参数与 `http.post()` 方法相同。

返回值说明

返回 Response 对象，结构同 `http.get()` 方法。

示例：更新资源

本示例演示如何使用 PUT 请求更新资源，适用于 RESTful API 中更新完整资源的场景。

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

export default function () {
    // 使用 PUT 请求更新资源
    const resp = http.put(
        'http://mockhttpbin.pts.svc.cluster.local/put',
        {
            id: '12345',
            name: 'Updated Name',
            status: 'active'
        },
        {
            headers: {
                'Content-Type': 'application/json',
```

```
            },
        }
    );

    check('update success', () => resp.statusCode === 200);
    console.log(resp.json().json.name); // 输出: Updated Name
}
```

HTTP DELETE 请求

函数说明

`http.delete()` 用于发起 HTTP DELETE 请求，适用于删除资源的场景。

函数签名

```
http.delete(url: string, options?: Request): Response
```

参数说明

- **url** (string，必填)：目标请求的完整 URL 地址。
- **options** (Request，可选)：请求配置对象，字段同 `http.get()` 方法，可设置 headers、query、timeout、maxRedirects、discardResponseBody、service 等。

返回值说明

返回 Response 对象，结构同 `http.get()` 方法。

示例：删除资源

本示例演示如何使用 DELETE 请求删除资源，适用于 RESTful API 中删除资源的场景。

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

export default function () {
    // 使用 DELETE 请求删除资源
    const resp =
http.delete('http://mockhttpbin.pts.svc.cluster.local/delete', {
        headers: {
            'Authorization': 'Bearer token123'
        },
```

```
        query: {
            'id': '12345'
        }
    });

    check('delete success', () => resp.statusCode === 200 ||
resp.statusCode === 204);
}
```

HTTP PATCH 请求

函数说明

`http.patch()` 用于发起 HTTP PATCH 请求，适用于部分更新资源的场景。PATCH 请求只需要提供需要更新的字段。

函数签名

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

参数说明

参数与 `http.post()` 方法相同。

返回值说明

返回 Response 对象，结构同 `http.get()` 方法。

示例：部分更新资源

本示例演示如何使用 PATCH 请求部分更新资源，适用于只需要更新部分字段的场景。

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

export default function () {
    // 使用 PATCH 请求部分更新资源
    const resp = http.patch(
        'http://mockhttpbin.pts.svc.cluster.local/patch',
        {
            status: 'inactive'  // 只更新 status 字段
```

```
        },
        {
            headers: {
                'Content-Type': 'application/json',
            },
        }
    );

    check('patch success', () => resp.statusCode === 200);
}
```

HTTP HEAD 请求

函数说明

`http.head()` 用于发起 HTTP HEAD 请求，只获取响应头信息，不返回响应体。适用于检查资源是否存在、获取资源元信息等场景。

函数签名

```
http.head(url: string, options?: Request): Response
```

参数说明

参数与 `http.get()` 方法相同。

返回值说明

返回 Response 对象，但 `body` 为空。可以通过 `headers` 属性获取响应头信息。

示例：检查资源

本示例演示如何使用 HEAD 请求检查资源是否存在，适用于不需要响应体内容的场景。

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

export default function () {
    // 使用 HEAD 请求检查资源
    const resp =
http.head('http://mockhttpbin.pts.svc.cluster.local/get', {
        headers: {
```

```
            'User-Agent': 'pts-engine'
        }
    });

    // 检查状态码
    check('resource exists', () => resp.statusCode === 200);

    // 获取响应头信息
    console.log('Content-Type:', resp.headers['Content-Type']);
    console.log('Content-Length:', resp.headers['Content-Length']);
}
```

HTTP 基础鉴权（Basic Authentication）

函数说明

HTTP 基础鉴权可以通过两种方式实现：

1. 在 URL 中包含用户名和密码：`http://username:password@host/path`。
2. 使用 `basicAuth` 配置项（需配合 `http.do()` 方法使用）。

使用限制

- URL 中的用户名和密码会以 Base64 编码形式发送。
- 仅适用于支持 HTTP Basic Authentication 的服务器。
- 生产环境建议使用更安全的认证方式。

示例：使用 URL 进行基础鉴权

本示例演示如何在 URL 中包含用户名和密码进行基础鉴权，适用于需要简单认证的场景。

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

export default function () {
    // 在 URL 中包含用户名和密码
    const user = 'user';
    const passwd = 'passwd';
    const resp =
http.get(`http://${user}:${passwd}@mockhttpbin.pts.svc.cluster.local/bas
ic-auth/user/passwd`);
```

```
    // 验证认证结果
    console.log(resp.json().authenticated); // 输出: true
    check('body.authenticated equals true', () =>
resp.json().authenticated === true);
}
```

HTTP Cookie 使用

函数说明

可以通过在请求头中设置 `Cookie` 字段来发送 Cookie，适用于需要维护会话状态的场景。

使用限制

- Cookie 值必须是字符串格式
- 多个 Cookie 可以使用分号分隔，如：`cookie1=value1; cookie2=value2`
- 如果需要自动管理 Cookie，建议使用 `http.do()` 方法配合相关配置

示例：发送 Cookie

本示例演示如何在请求中发送 Cookie，适用于需要传递会话信息的场景。

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

export default function () {
    // 在请求头中设置 Cookie
    const resp =
http.get('http://mockhttpbin.pts.svc.cluster.local/cookies', {
        headers: {
            cookie: 'k=v; session_id=abc123; user_id=456'
        }
    });

    // 验证 Cookie 是否正确传递
    console.log(resp.json().cookies.k); // 输出: v
    console.log(resp.json().cookies.session_id); // 输出: abc123
    check('body.cookies.k equals v', () => resp.json().cookies.k ===
'v');
}
```

通用请求方法 http.do()

函数说明

`http.do()` 是一个通用的 HTTP 请求方法，可以发送任意 HTTP 方法的请求。适用于需要更灵活控制请求参数或使用高级功能的场景。

函数签名

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

参数说明

request（Request，必填）：请求配置对象，包含以下字段：

- `method`（string，必填）：HTTP 方法，如 'GET'、'POST'、'PUT'、'DELETE'、'PATCH'、'HEAD' 等。
- `url`（string，必填）：目标请求的完整 URL 地址。
- `body`（string | object | ArrayBuffer，可选）：请求体内容，在使用 http.do 方法时才需要指定。
- `headers`（Record<string, string>，可选）：请求头。
- `query`（Record<string, string>，可选）：URL 查询参数。
- `timeout`（number，可选）：请求超时时间（毫秒），包括连接时间、任何重定向和读取响应正文的时间。
- `maxRedirects`（number，可选）：最大重定向跳转次数。
- `discardResponseBody`（boolean，可选）：是否丢弃响应体，适用于响应体太大且不需要进行 check 的场景。
- `service`（string，可选）：服务标识，用于在报表中将不同 URL 的请求归类到同一个服务下。
- `basicAuth`（BasicAuth，可选）：基础鉴权配置。
- `chunked`（function，可选）：分块传输回调函数，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。函数签名为 `(body: string) => void`。
- `contentLength`（number，可选）：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。
- `host`（string，可选）：host 或 host:port，通常不需要单独指定，使用 url 即可。
- `path`（string，可选）：路径，相对路径省略前导斜杠，通常不需要单独指定，使用 url 即可。
- `scheme`（string，可选）：协议，填写 "http" 或 "https"，通常不需要单独指定，使用 url 即可。

返回值说明

返回 Response 对象，结构同其他 HTTP 方法。

使用限制

- `method` 和 `url` 字段为必填项。
- 当使用 `body` 字段时，需要根据内容类型设置相应的 `Content-Type` 请求头。
- `host` 、 `path` 、 `scheme` 字段通常不需要单独指定，直接使用完整的 `url` 即可。
- `chunked` 回调函数仅在响应体以分块形式传输时才会被调用。

示例：使用 http.do() 发送请求

本示例演示如何使用 `http.do()` 方法发送请求，适用于需要完整控制请求参数的场景。

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

export default function () {
    // 使用 http.do() 发送 POST 请求
    const req = {
        method: 'post',
        url: 'http://mockhttpbin.pts.svc.cluster.local/post',
        headers: {
            'Content-Type': 'application/json'
        },
        query: {
            a: '1'
        },
        body: {
            user_id: '12345'
        },
        timeout: 5000,
        maxRedirects: 3
    };

    const resp = http.do(req);

    // 验证查询参数
    console.log(resp.json().args.a); // 输出: 1

    // 验证请求体
    console.log(resp.json().json.user_id); // 输出: 12345

    check('request success', () => resp.statusCode === 200);
```

```
}
```

批量请求 http.batch()

函数说明

`http.batch()` 用于批量发起多个 HTTP 请求，支持并行执行，适用于需要同时发起多个请求的场景，可以提高压测效率。

函数签名

```
http.batch(requests: Request[], options?: BatchOption): BatchResponse[]
```

参数说明

- **requests** (Request[]，必填)：请求配置对象数组，每个元素的结构与 `http.do()` 方法的参数相同，必须包含 `method` 和 `url` 字段。
- **options** (BatchOption，可选)：批量请求配置选项。
  - `parallel` (number，可选)：并行执行数，默认值为 20。

返回值说明

返回 `BatchResponse[]` 数组，每个元素包含以下字段：

- `error` (string)：错误信息，不为空则表示请求出错。
- `response` (Response)：请求结果，包含 statusCode、body、headers 等属性。

使用限制

- 批量请求的数量建议控制在合理范围内，避免过多请求导致性能问题。
- 并行数 `parallel` 的值会影响请求的并发度，需要根据实际情况调整。

示例：批量发送请求

本示例演示如何使用 `http.batch()` 批量发送多个请求，适用于需要同时测试多个接口的场景。

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

export default function () {
    // 批量发送多个请求
    const responses = http.batch(
        [
```

```
            {
                method: 'GET',
                url: 'http://mockhttpbin.pts.svc.cluster.local/get?a=1',
                headers: { 'User-Agent': 'pts-engine' },
                query: { b: '2' },
            },
            {
                method: 'POST',
                url: 'http://mockhttpbin.pts.svc.cluster.local/post',
                headers: { 'Content-Type': 'application/json' },
                body: { user_id: '12345' },
            },
            {
                method: 'GET',
                url: 'http://mockhttpbin.pts.svc.cluster.local/get?c=3',
                headers: { 'Authorization': 'Bearer token123' },
            },
        ],
        // 批量请求配置选项
        {
            parallel: 3,  // 并行执行 3 个请求
        }
    );

    // 处理每个响应
    responses.forEach((resp, index) => {
        if (resp.error) {
            console.log(`Request ${index + 1} error:`, resp.error);
        } else {
            console.log(`Request ${index + 1} status:`,
resp.response.statusCode);
            check(`Request ${index + 1} success`, () =>
resp.response.statusCode === 200);
        }
    });

    // 输出所有响应
    console.log(JSON.stringify(responses));
}
```

Response 对象常用方法

json() 方法

`Response.json()` 方法用于将响应体反序列化为 JSON 对象。

函数签名

```
json(): any
```

返回值

返回代表 `Response.body` 的 JavaScript 对象。

使用限制

- 仅当响应体为有效的 JSON 字符串时才能使用。
- 如果响应体不是 JSON 格式，调用此方法会抛出异常。
- 建议在使用前先检查响应体的 `Content-Type` 是否为 `application/json`。

示例：解析 JSON 响应

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

export default function () {
    const resp =
http.get('http://mockhttpbin.pts.svc.cluster.local/get', {
        query: { key: 'value' }
    });

    // 检查响应是否为 JSON 格式
    const contentType = resp.headers['Content-Type'] || '';
    if (contentType.includes('application/json')) {
        const data = resp.json();
        console.log('Parsed JSON:', data);
        check('has args', () => data.args !== undefined);
    } else {
        console.log('Response body:', resp.body);
    }
}
```

WebSocket

最近更新时间：2025-12-01 16:56:01

本文档提供了在云压测脚本模式中使用 WebSocket 协议进行压测的完整示例。所有示例均基于 `pts/ws` 模块提供的 WebSocket 连接和消息处理方法。

前置条件

在使用 WebSocket 功能前，请确保：

- **导入模块**：在脚本开头使用 `import ws from 'pts/ws'` 导入 WebSocket 模块。
- **脚本结构**：所有连接和消息处理代码需放在 `export default function()` 函数中执行。
- **URL 格式**：确保目标 WebSocket URL 格式正确，支持 `ws://` 和 `wss://` 协议。
- **网络连通性**：确保压测环境能够访问目标 WebSocket 服务器。
- **理解执行机制**：WebSocket 脚本的执行机制与 HTTP 脚本不同：
  - HTTP 脚本的每个 VU（虚拟用户）会持续迭代主函数，直到压测结束。
  - WebSocket 脚本的每个 VU 不会持续迭代主函数，因为主函数会被 `ws.connect` 方法阻塞，直到连接关闭。
  - 在连接建立后的回调函数中，会持续监听和处理异步事件，直到压测结束。

注意事项

- **连接生命周期**：WebSocket 连接建立后，主函数会被阻塞，直到连接关闭。所有业务逻辑都应在 `ws.connect()` 的回调函数中实现。
- **事件监听顺序**：应在连接建立后立即注册所有需要的事件监听器，确保不会遗漏任何消息。
- **消息发送时机**：虽然可以在回调函数的任何位置调用 `send()`，但应在 `open` 事件触发后或确认连接已建立后再发送消息。
- **资源清理**：使用 `setTimeout`、`setInterval` 和 `setLoop` 时，注意在连接关闭时清理这些定时器，避免资源泄漏。
- **消息格式**：根据服务器要求选择正确的消息格式（文本或二进制），并使用相应的方法发送。
- **连接保持**：对于长连接场景，建议定期发送 ping 消息或心跳消息，防止连接因超时被关闭。
- **并发控制**：在压测场景中，每个 VU 会建立独立的 WebSocket 连接，注意控制并发连接数，避免对服务器造成过大压力。
- **状态码检查**：连接建立后，应检查 `res.status` 是否为101，确认连接成功建立。

WebSocket 连接

函数说明

`ws.connect()` 用于建立 WebSocket 连接，是 WebSocket 压测的基础方法。该方法在脚本模式的压测场景中使用，支持建立持久连接并进行双向通信。连接建立后，主函数会被阻塞，直到连接关闭。

函数签名

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

参数说明

- **url** (string，必填)：目标 WebSocket 服务器的完整 URL 地址，支持 `ws://` 和 `wss://` 协议。
- **callback** (function，必填)：连接建立成功后的回调函数，接收一个 `Socket` 对象作为参数。
  - 回调函数会在连接建立后立即执行。
  - 在回调函数中定义所有的消息发送和事件监听逻辑。
  - 回调函数执行完成后， `ws.connect` 才会返回。
- **headers** (Record<string, string>，可选)：请求连接时的 headers 配置，用于设置自定义请求头。

返回值说明

返回 `Response` 对象，包含以下属性：

- `status` (number)：HTTP 状态码，WebSocket 连接成功时状态码为 101（Switching Protocols）。
- `body` (string)：响应包体内容。
- `headers` (Record<string, string>)：响应头参数。
- `url` (string)：请求地址。

使用限制

- URL 必须包含完整的协议前缀（ `ws://` 或 `wss://` ）。
- 回调函数是必需的，所有 Socket 操作都必须在回调函数中进行。
- 连接建立后，主函数会被阻塞，直到连接关闭或回调函数执行完成。
- 连接失败时， `status` 不是 101，需要检查连接状态。

示例

示例1：基本连接和消息收发

本示例演示如何建立 WebSocket 连接、发送消息和接收消息，适用于基本的 WebSocket 通信场景。

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

```
export default function () {
    // 建立 WebSocket 连接
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        // 监听连接建立事件
        socket.on('open', () => {
            console.log('WebSocket connected');
        });

        // 监听接收文本消息事件
        socket.on('message', (data) => {
            console.log('Message received:', data);
        });

        // 监听连接关闭事件
        socket.on('close', () => {
            console.log('WebSocket disconnected');
        });

        // 发送文本消息
        socket.send('Hello, WebSocket!');
    });

    // 检查连接状态码是否为 101（WebSocket 协议切换成功）
    check('status is 101', () => res.status === 101);
}
```

示例2：带错误处理的连接

本示例演示如何处理连接错误，适用于需要健壮错误处理的场景。

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        socket.on('open', () => {
```

```
            console.log('Connected successfully');
        });

        socket.on('message', (data) => {
            console.log('Received:', data);
        });

        socket.on('close', () => {
            console.log('Connection closed');
        });

        socket.send('Test message');
    });

    // 验证连接成功
    check('connection successful', () => res.status === 101);

    // 连接失败时输出错误信息
    if (res.status !== 101) {
        console.error('Connection failed with status:', res.status);
    }
}
```

示例3：带自定义请求头的连接

本示例演示如何在建立连接时设置自定义请求头，适用于需要传递认证信息或自定义头部字段的场景。

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

export default function () {
    // 定义自定义请求头
    const headers = {
        'X-MyApplication': 'PTS',
        'X-MyScript': 'WebSocket',
        'Authorization': 'Bearer token123'
    };

    // 建立连接并传入 headers
```

```
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        socket.on('open', () => {
            console.log('Connected with custom headers');
        });

        socket.on('message', (data) => {
            console.log('Received:', data);
        });

        socket.on('close', () => {
            console.log('Disconnected');
        });

        socket.send('Hello');
    }, headers);

    check('connection successful', () => res.status === 101);
}
```

Socket 对象事件监听

函数说明

`socket.on()` 用于监听 WebSocket 连接的各种事件，是处理异步消息和连接状态变化的核心方法。该方法在 `ws.connect()` 的回调函数中使用，用于注册事件处理函数。

函数签名

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

参数说明

- **event** (string，必填)：事件名称，支持以下事件：
  - `open`：建立连接时触发，无参数。
  - `close`：关闭连接时触发，无参数。
  - `message`：接收文本消息时触发，回调函数接收消息内容作为参数。
  - `binaryMessage`：接收二进制消息时触发，回调函数接收 ArrayBuffer 作为参数。
  - `ping`：接收 ping 消息时触发，无参数。

- `pong`：接收 pong 消息时触发，无参数。
- `error`：发生错误时触发，回调函数接收错误对象作为参数，错误对象包含 `error()` 方法用于获取错误信息。

- **callback** (function，必填)：事件回调函数，根据事件类型接收不同的参数。

返回值说明

无返回值（void）。

使用限制

- 必须在 `ws.connect()` 的回调函数中调用。
- 同一个事件可以注册多个监听器，都会被执行。
- 事件监听器应该在发送消息之前注册，以确保能接收到所有消息。
- `message` 和 `binaryMessage` 事件回调函数的参数类型不同，需要根据消息类型选择正确的事件。

示例

示例1：监听所有事件类型

本示例演示如何监听 WebSocket 支持的所有事件类型，适用于需要全面监控连接状态的场景。

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        // 监听连接建立
        socket.on('open', () => {
            console.log('Connection opened');
        });

        // 监听文本消息
        socket.on('message', (data) => {
            console.log('Text message received:', data);
        });

        // 监听二进制消息
        socket.on('binaryMessage', (data) => {
```

```
            console.log('Binary message received, length:',
data.byteLength);
        });

        // 监听 ping 消息
        socket.on('ping', () => {
            console.log('Ping received');
        });

        // 监听 pong 消息
        socket.on('pong', () => {
            console.log('Pong received');
        });

        // 监听连接关闭
        socket.on('close', () => {
            console.log('Connection closed');
        });

        // 监听错误事件
        socket.on('error', (e) => {
            console.log('Error happened:', e.error());
        });

        // 发送测试消息
        socket.send('Hello');
    });

    check('connection successful', () => res.status === 101);
}
```

示例2：处理不同类型的消息

本示例演示如何区分处理文本消息和二进制消息，适用于需要处理多种消息格式的场景。

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

export default function () {
```

```
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        socket.on('open', () => {
            console.log('Connected');
            // 发送文本消息
            socket.send('Text message');
            // 发送二进制消息
            socket.sendBinary(new ArrayBuffer(8));
        });

        // 处理文本消息
        socket.on('message', (data) => {
            console.log('Received text:', data);
            // 可以在这里进行文本消息的业务处理
            if (data === 'ping') {
                socket.send('pong');
            }
        });

        // 处理二进制消息
        socket.on('binaryMessage', (data) => {
            console.log('Received binary, size:', data.byteLength);
            // 可以在这里进行二进制消息的业务处理
            const view = new Uint8Array(data);
            console.log('First byte:', view[0]);
        });

        socket.on('close', () => {
            console.log('Disconnected');
        });
    });

    check('connection successful', () => res.status === 101);
}
```

发送文本消息

函数说明

`socket.send()` 用于向 WebSocket 服务器发送文本消息，是 WebSocket 双向通信中客户端向服务器发送数据的主要方法。该方法在连接建立后的回调函数中使用。

函数签名

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

参数说明

msg (string，必填)：要发送的文本消息内容。

返回值说明

无返回值（void）。

使用限制

- 必须在连接建立后（`open` 事件触发后）调用，否则消息无法发送。
- 消息内容必须是字符串类型，非字符串值会被转换为字符串。
- 连接已关闭时，调用此方法不会报错，但消息不会发送。
- 应在 `open` 事件回调中或确认连接已建立后再发送消息。

示例

示例1：发送简单文本消息

本示例演示如何发送简单的文本消息，适用于基本的消息发送场景。

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        socket.on('open', () => {
            console.log('Connected');
            // 连接建立后立即发送消息
            socket.send('Hello, Server!');
        });

        socket.on('message', (data) => {
```

```
            console.log('Echo received:', data);
        });

        socket.on('close', () => {
            console.log('Disconnected');
        });
    });

    check('connection successful', () => res.status === 101);
}
```

示例2：发送 JSON 格式消息

本示例演示如何发送 JSON 格式的文本消息，适用于需要发送结构化数据的场景。

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        socket.on('open', () => {
            // 构造 JSON 对象并转换为字符串发送
            const message = {
                type: 'chat',
                user: 'testuser',
                content: 'Hello from WebSocket',
                timestamp: Date.now()
            };
            socket.send(JSON.stringify(message));
        });

        socket.on('message', (data) => {
            try {
                // 解析接收到的 JSON 消息
                const msg = JSON.parse(data);
                console.log('Received message:', msg);
            } catch (e) {
                console.log('Received non-JSON message:', data);
```

```
            }
        });

        socket.on('close', () => {
            console.log('Disconnected');
        });
    });

    check('connection successful', () => res.status === 101);
}
```

示例3：定时发送消息

本示例演示如何定时发送消息，适用于需要周期性发送数据的场景。

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        let messageCount = 0;

        socket.on('open', () => {
            console.log('Connected, starting to send messages');
        });

        socket.on('message', (data) => {
            console.log('Received:', data);
        });

        // 使用 setInterval 定时发送消息
        socket.setInterval(function () {
            messageCount++;
            const msg = `Message #${messageCount}`;
            socket.send(msg);
            console.log('Sent:', msg);
        }, 1000); // 每秒发送一条消息
```

```
        // 10 秒后关闭连接
        socket.setTimeout(function () {
            console.log('Closing connection after 10 seconds');
            socket.close();
        }, 10000);

        socket.on('close', () => {
            console.log('Disconnected');
        });
    });

    check('connection successful', () => res.status === 101);
}
```

发送二进制消息

函数说明

`socket.sendBinary()` 用于向 WebSocket 服务器发送二进制消息，适用于需要传输二进制数据（如图片、文件、协议数据等）的场景。该方法在连接建立后的回调函数中使用。

函数签名

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

参数说明

msg (ArrayBuffer，必填)：要发送的二进制数据，必须是 ArrayBuffer 类型。

返回值说明

无返回值（void）。

使用限制

- 必须在连接建立后调用。
- 参数必须是 ArrayBuffer 类型，不能是其他类型（如 Uint8Array、Buffer 等）。
- 从其他格式转换为 ArrayBuffer 时，使用 `util.toArrayBuffer()` 进行转换。
- 二进制消息的接收需要使用 `binaryMessage` 事件，而不是 `message` 事件。

示例

示例1：发送简单二进制数据

本示例演示如何发送简单的二进制数据，适用于基本的二进制消息发送场景。

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        socket.on('open', () => {
            console.log('Connected');
            // 创建一个包含 8 字节的 ArrayBuffer
            const buffer = new ArrayBuffer(8);
            const view = new Uint8Array(buffer);
            // 填充一些测试数据
            for (let i = 0; i < view.length; i++) {
                view[i] = i;
            }
            // 发送二进制数据
            socket.sendBinary(buffer);
            console.log('Binary message sent, size:',
buffer.byteLength);
        });

        // 监听二进制消息
        socket.on('binaryMessage', (data) => {
            console.log('Binary message received, size:',
data.byteLength);
            const view = new Uint8Array(data);
            console.log('First 4 bytes:', view.slice(0, 4));
        });

        socket.on('close', () => {
            console.log('Disconnected');
        });
    });

    check('connection successful', () => res.status === 101);
```

```
}
```

示例2：发送字符串转二进制

本示例演示如何将字符串转换为二进制数据发送，适用于需要以二进制格式传输文本的场景。

```
import ws from 'pts/ws';
import { check } from 'pts';
import util from 'pts/util';

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        socket.on('open', () => {
            // 使用 PTS 的 util.toArrayBuffer 将字符串转换为 ArrayBuffer
            const text = 'Hello, Binary World!';
            const buffer = util.toArrayBuffer(text);

            socket.sendBinary(buffer);
            console.log('Sent binary data from string:', text);
        });

        socket.on('binaryMessage', (data) => {
            // 读取二进制数据的字节信息
            console.log('Received binary message, size:',
data.byteLength);

            // 使用 Uint8Array 读取字节
            const view = new Uint8Array(data);
            console.log('First 5 bytes:', Array.from(view.slice(0, 5)));

            // 将 ArrayBuffer 转换为字符串：使用 base64 编码/解码
            // 注意：仅适用于 ArrayBuffer 内容是 UTF-8 编码文本的场景
            const base64 = util.base64Encoding(data);
            const text = util.base64Decoding(base64);
            console.log('Decoded text:', text);
        });

        socket.on('close', () => {
```

```
            console.log('Disconnected');
        });
    });

    check('connection successful', () => res.status === 101);
}
```

关闭连接

函数说明

`socket.close()` 用于主动关闭 WebSocket 连接，适用于需要控制连接生命周期的场景。调用此方法后，连接会正常关闭，并触发 `close` 事件。

函数签名

```
socket.close(): void
```

返回值说明

无返回值（void）。

使用限制

- 可以在连接建立后的任何时候调用
- 调用后连接会立即关闭，无法再发送或接收消息
- 关闭连接会触发 `close` 事件
- 连接已经关闭时，再次调用此方法不会报错，但不会有任何效果

示例

示例1：定时关闭连接

本示例演示如何在指定时间后关闭连接，适用于需要限制连接时长的场景。

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
```

```
        socket.on('open', () => {
            console.log('Connected');
            socket.send('Hello');
        });

        socket.on('message', (data) => {
            console.log('Received:', data);
        });

        // 5 秒后关闭连接
        socket.setTimeout(function () {
            console.log('5 seconds passed, closing connection');
            socket.close();
        }, 5000);

        socket.on('close', () => {
            console.log('Connection closed');
        });
    });

    check('connection successful', () => res.status === 101);
}
```

示例2：收到特定消息后关闭

本示例演示如何在收到特定消息后关闭连接，适用于需要根据服务器响应决定是否关闭连接的场景。

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        socket.on('open', () => {
            console.log('Connected');
            socket.send('start');
        });

        socket.on('message', (data) => {
```

```
            console.log('Received:', data);

            // 收到 "close" 消息时关闭连接
            if (data === 'close' || data === 'bye') {
                console.log('Received close signal, closing
connection');
                socket.close();
            } else {
                // 继续发送消息
                socket.send('continue');
            }
        });

        socket.on('close', () => {
            console.log('Connection closed');
        });
    });

    check('connection successful', () => res.status === 101);
}
```

发送 Ping 消息

函数说明

`socket.ping()` 用于向服务器发送 WebSocket ping 消息，用于保持连接活跃和检测连接状态。服务器通常会响应 pong 消息。该方法在连接建立后的回调函数中使用。

函数签名

```
socket.ping(): void
```

返回值说明

无返回值（void）。

使用限制

- 必须在连接建立后调用
- ping 消息是 WebSocket 协议层面的控制消息，不会触发 `message` 事件
- 服务器响应 pong 消息时，会触发 `pong` 事件

- 定期发送 ping 可以保持连接活跃，防止因超时被关闭

示例：定期发送 Ping 保持连接

本示例演示如何定期发送 ping 消息来保持连接活跃，适用于需要维持长连接的场景。

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        socket.on('open', () => {
            console.log('Connected');
        });

        // 监听 pong 响应
        socket.on('pong', () => {
            console.log('Received pong from server');
        });

        // 每 500 毫秒发送一次 ping
        socket.setInterval(function () {
            socket.ping();
            console.log('Sent ping');
        }, 500);

        socket.on('message', (data) => {
            console.log('Received message:', data);
        });

        // 30 秒后关闭连接
        socket.setTimeout(function () {
            console.log('Closing connection');
            socket.close();
        }, 30000);

        socket.on('close', () => {
            console.log('Disconnected');
```

```
        });
    });

    check('connection successful', () => res.status === 101);
}
```

设置定时器

函数说明

`socket.setTimeout()` 用于在 WebSocket 连接上设置一个定时器，在指定时间后执行回调函数。适用于需要在特定时间后执行某些操作（如关闭连接、发送消息等）的场景。

函数签名

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

参数说明

- **callback** (function，必填)：定时器到期后执行的回调函数，无参数。
- **intervalMs** (number，必填)：定时器延迟时间，单位为毫秒。

返回值说明

无返回值（void）。

使用限制

- 必须在连接建立后的回调函数中调用。
- 定时器是单次执行的，执行一次后不会自动重复。
- 需要重复执行时，可以在回调函数中再次调用 `setTimeout`，或使用 `setInterval`。
- 连接在定时器到期前关闭时，定时器不会执行。

示例

示例1：延迟发送消息

本示例演示如何使用定时器延迟发送消息，适用于需要延迟操作的场景。

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

export default function () {
```

```
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        socket.on('open', () => {
            console.log('Connected');

            // 2 秒后发送第一条消息
            socket.setTimeout(function () {
                socket.send('First message after 2 seconds');
            }, 2000);

            // 5 秒后发送第二条消息
            socket.setTimeout(function () {
                socket.send('Second message after 5 seconds');
            }, 5000);
        });

        socket.on('message', (data) => {
            console.log('Received:', data);
        });

        // 10 秒后关闭连接
        socket.setTimeout(function () {
            console.log('Closing connection');
            socket.close();
        }, 10000);

        socket.on('close', () => {
            console.log('Disconnected');
        });
    });

    check('connection successful', () => res.status === 101);
}
```

示例2：超时自动关闭

本示例演示如何使用定时器实现超时自动关闭连接，适用于需要限制连接时长的场景。

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

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        let isClosed = false;

        socket.on('open', () => {
            console.log('Connected, will close after 5 seconds');
            socket.send('Hello');
        });

        socket.on('message', (data) => {
            console.log('Received:', data);
        });

        // 5 秒后自动关闭连接
        socket.setTimeout(function () {
            if (!isClosed) {
                console.log('Timeout reached, closing connection');
                isClosed = true;
                socket.close();
            }
        }, 5000);

        socket.on('close', () => {
            console.log('Connection closed');
            isClosed = true;
        });
    });

    check('connection successful', () => res.status === 101);
}
```

设置轮询函数

函数说明

`socket.setInterval()` 用于在 WebSocket 连接上设置一个轮询函数，按照指定的时间间隔重复执行。适用于需要周期性执行某些操作（如定期发送消息、定期 ping 等）的场景。

函数签名

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

参数说明

- **callback** (function，必填)：每次轮询时执行的回调函数，无参数。
- **intervalMs** (number，必填)：轮询间隔时间，单位为毫秒。

返回值说明

无返回值（void）。

使用限制

- 必须在连接建立后的回调函数中调用
- 轮询会持续执行，直到连接关闭
- 回调函数执行时间超过间隔时间时，会影响定时精度
- 应在回调函数中避免耗时操作，或使用异步处理

示例

示例1：定期发送消息

本示例演示如何使用轮询函数定期发送消息，适用于需要周期性发送数据的场景。

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        let messageCount = 0;

        socket.on('open', () => {
            console.log('Connected, starting periodic messages');
        });
```

```
        socket.on('message', (data) => {
            console.log('Received:', data);
        });

        // 每 1 秒发送一条消息
        socket.setInterval(function () {
            messageCount++;
            const msg = `Periodic message #${messageCount}`;
            socket.send(msg);
            console.log('Sent:', msg);
        }, 1000);

        // 10 秒后关闭连接
        socket.setTimeout(function () {
            console.log('Closing connection');
            socket.close();
        }, 10000);

        socket.on('close', () => {
            console.log('Disconnected');
        });
    });

    check('connection successful', () => res.status === 101);
}
```

示例2：定期发送心跳

本示例演示如何使用轮询函数定期发送心跳消息，适用于需要保持连接活跃的场景。

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        socket.on('open', () => {
            console.log('Connected');
        });
```

```
        socket.on('message', (data) => {
            console.log('Received:', data);
        });

        // 每 2 秒发送一次心跳消息
        socket.setInterval(function () {
            const heartbeat = {
                type: 'heartbeat',
                timestamp: Date.now()
            };
            socket.send(JSON.stringify(heartbeat));
            console.log('Sent heartbeat');
        }, 2000);

        // 30 秒后关闭连接
        socket.setTimeout(function () {
            console.log('Closing connection');
            socket.close();
        }, 30000);

        socket.on('close', () => {
            console.log('Disconnected');
        });
    });

    check('connection successful', () => res.status === 101);
}
```

设置循环执行函数

函数说明

`socket.setLoop()` 用于在 WebSocket 连接上设置一个循环执行函数，该函数会持续执行直到连接关闭。与 `setInterval` 不同，`setLoop` 的执行频率由函数内部的 `sleep()` 调用控制，提供了更灵活的执行控制。

函数签名

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

参数说明

callback (function，必填)：循环执行的回调函数，无参数。函数内部应包含 `sleep()` 调用来控制执行频率。

返回值说明

无返回值（void）。

使用限制

- 必须在连接建立后的回调函数中调用
- 循环会持续执行，直到连接关闭
- 回调函数内部必须调用 `sleep()` 来控制执行频率，否则会无限快速执行
- 建议在回调函数中使用 `sleep()` 来避免 CPU 占用过高

示例

示例 1：循环发送消息

本示例演示如何使用循环函数持续发送消息，适用于需要持续发送数据的场景。

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        let messageCount = 0;

        socket.on('open', () => {
            console.log('Connected, starting loop');
        });

        socket.on('message', (data) => {
            console.log('Received:', data);
        });

        // 设置循环，每 0.5 秒发送一条消息
        socket.setLoop(function () {
            messageCount++;
            socket.send(`Loop message #${messageCount}`);
```

```
            console.log('Sent loop message #' + messageCount);
            // 必须调用 sleep 来控制执行频率
            sleep(0.5);
        });

        // 10 秒后关闭连接
        socket.setTimeout(function () {
            console.log('Closing connection');
            socket.close();
        }, 10000);

        socket.on('close', () => {
            console.log('Disconnected');
        });
    });

    check('connection successful', () => res.status === 101);
}
```

示例 2：循环处理业务逻辑

本示例演示如何在循环中处理复杂的业务逻辑，适用于需要持续处理业务的场景。

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        let state = 'idle';

        socket.on('open', () => {
            console.log('Connected');
            state = 'active';
        });

        socket.on('message', (data) => {
            console.log('Received:', data);
            // 根据接收到的消息更新状态
```

```
            if (data === 'start') {
                state = 'running';
            } else if (data === 'stop') {
                state = 'stopped';
            }
        });

        // 循环处理业务逻辑
        socket.setLoop(function () {
            if (state === 'running') {
                // 业务逻辑：发送数据
                const data = {
                    action: 'process',
                    timestamp: Date.now()
                };
                socket.send(JSON.stringify(data));
                console.log('Sent process data');
            } else if (state === 'stopped') {
                console.log('State is stopped, waiting...');
            }
            // 控制循环频率：每 1 秒执行一次
            sleep(1);
        });

        // 30 秒后关闭连接
        socket.setTimeout(function () {
            console.log('Closing connection');
            socket.close();
        }, 30000);

        socket.on('close', () => {
            console.log('Disconnected');
            state = 'closed';
        });
    });

    check('connection successful', () => res.status === 101);
}
```

综合示例：完整的 WebSocket 压测场景

本示例演示一个完整的 WebSocket 压测场景，包含连接建立、消息收发、心跳保持、错误处理和连接关闭等完整流程。

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

export default function () {
    const res =
ws.connect('ws://mockwebsocket.pts.svc.cluster.local/echo', function
(socket) {
        let messageCount = 0;
        let isActive = true;

        // 监听连接建立
        socket.on('open', () => {
            console.log('WebSocket connection established');
            check('connection opened', () => true);
        });

        // 监听文本消息
        socket.on('message', (data) => {
            messageCount++;
            console.log(`Message #${messageCount} received:`, data);

            // 验证消息内容
            if (data.includes('echo')) {
                check('message contains echo', () =>
data.includes('echo'));
            }
        });

        // 监听二进制消息
        socket.on('binaryMessage', (data) => {
            console.log('Binary message received, size:',
data.byteLength);
        });

        // 监听 pong 响应
```

```
        socket.on('pong', () => {
            console.log('Pong received from server');
        });

        // 监听连接关闭
        socket.on('close', () => {
            console.log('WebSocket connection closed');
            isActive = false;
        });

        // 连接建立后立即发送欢迎消息
        socket.send('Hello from WebSocket client');

        // 定期发送 ping 保持连接
        socket.setInterval(function () {
            if (isActive) {
                socket.ping();
                console.log('Sent ping');
            }
        }, 2000);

        // 循环发送业务消息
        socket.setLoop(function () {
            if (isActive) {
                const message = {
                    type: 'data',
                    sequence: messageCount + 1,
                    timestamp: Date.now(),
                    data: `Message content ${messageCount + 1}`
                };
                socket.send(JSON.stringify(message));
                console.log('Sent business message:', message.sequence);
            }
            sleep(1); // 每 1 秒发送一次
        });

        // 30 秒后关闭连接
        socket.setTimeout(function () {
            if (isActive) {
                console.log('Closing connection after 30 seconds');
```

```
                socket.close();
            }
        }, 30000);
    });

    // 验证连接状态
    check('WebSocket connection successful', () => res.status === 101);

    if (res.status !== 101) {
        console.error('WebSocket connection failed with status:',
res.status);
    }
}
```

常用函数

最近更新时间：2025-11-28 11:15:42

本文档介绍在云压测脚本模式中常用的 JavaScript 原生函数和 PTS 扩展库函数的使用方法。这些函数可用于数据处理、时间计算、随机数生成、字符串匹配等场景，帮助您构建更强大的压测脚本。

JS 公共库

云压测脚本模式支持 JavaScript 原生语法，详情请参见 JavaScript 标准内置对象 。以下为常用函数的说明和示例。

Date 对象

函数说明

`Date` 对象用于处理日期和时间，在压测脚本中常用于：

- 生成时间戳用于请求参数。
- 计算请求耗时。
- 格式化时间字符串用于日志记录。
- 实现时间相关的业务逻辑。

前提条件

- 所有代码需放在 `export default function()` 函数中执行
- 日期字符串格式需符合 ISO 8601 标准或 JavaScript 支持的格式

函数签名

```
new Date(value?: string | number): Date
```

常用方法

- `Date.now()`：返回当前时间戳（毫秒数）。
- `getTime()`：返回时间戳（毫秒数）。
- `toISOString()`：返回 ISO 8601 格式的字符串。
- `getFullYear()`、`getMonth()`、`getDate()` 等：获取日期各部分。

使用限制

- 时区默认为 UTC+8（中国标准时间）。
- 日期字符串解析可能因格式不同而产生差异。

示例1：创建日期对象并格式化

本示例演示如何创建日期对象并获取不同格式的时间信息，适用于需要在请求中使用时间参数的场景。

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

export default function () {
    // 创建日期对象
    const date1 = new Date('1995-12-17T03:24:00');
    console.log(date1); // Sun Dec 17 1995 11:24:00 GMT+0800 (CST)

    // 获取当前时间
    const now = new Date();
    console.log(now.toISOString()); // 2024-01-01T12:00:00.000Z

    // 获取时间戳（毫秒）
    const timestamp = Date.now();
    console.log(timestamp); // 1704096000000

    // 在 HTTP 请求中使用时间戳作为参数
    const resp = http.get('http://example.com/api', {
        query: {
            'timestamp': timestamp.toString()
        }
    });
}
```

示例2：计算请求耗时

本示例演示如何使用 Date 对象计算请求耗时，用于性能监控和调试。

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

export default function () {
    // 记录开始时间
    const startTime = Date.now();

    // 发起请求
    const resp = http.get('http://example.com/api');
```

```
    // 计算耗时
    const endTime = Date.now();
    const duration = endTime - startTime;

    console.log(`请求耗时: ${duration} 毫秒`);

    // 检查响应状态
    if (resp.statusCode === 200) {
        console.log('请求成功');
    }
}
```

JSON 对象

函数说明

`JSON` 对象提供 `parse()` 和 `stringify()` 方法，在压测脚本中用于：

- 解析 HTTP 响应体中的 JSON 数据。
- 将 JavaScript 对象序列化为 JSON 字符串用于请求体。
- 处理嵌套的 JSON 数据结构。
- 数据格式转换。

前提条件

- `JSON.parse()` 要求输入字符串必须是有效的 JSON 格式，否则会抛出异常。
- `JSON.stringify()` 可以处理对象、数组、字符串、数字、布尔值、null 等类型。
- 函数、undefined、Symbol 等类型在序列化时会被忽略或转换为 null。

函数签名

```
JSON.parse(text: string, reviver?: function): any
JSON.stringify(value: any, replacer?: function | array, space?: number |
string): string
```

参数说明

JSON.parse()

- `text` (string，必填)：要解析的 JSON 字符串。
- `reviver` (function，可选)：转换函数，用于在返回之前转换解析结果。

JSON.stringify()

- `value` (any，必填)：要序列化的值。

- `replacer` (function | array，可选)：用于控制序列化过程的函数或数组。
- `space` (number | string，可选)：用于缩进的空格数或字符串。

返回值说明

- `JSON.parse()`：返回解析后的 JavaScript 对象或值。
- `JSON.stringify()`：返回 JSON 字符串。

使用限制

- `JSON.parse()` 遇到无效 JSON 会抛出 `SyntaxError` 异常，建议使用 try-catch 处理。
- 循环引用的对象无法使用 `JSON.stringify()` 序列化。
- `undefined`、函数、Symbol 等类型在序列化时会被忽略。

示例1：解析响应 JSON 数据

本示例演示如何解析 HTTP 响应中的 JSON 数据并提取字段，适用于处理 API 响应的场景。

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

export default function () {
    // 发起请求获取 JSON 响应
    const resp = http.get('http://example.com/api/user');

    // 解析 JSON 响应体
    const jsonData = JSON.parse(resp.body);
    console.log(jsonData.name); // 输出用户名称

    // 检查响应数据
    check('用户 ID 存在', () => jsonData.id !== undefined);
    check('用户名称不为空', () => jsonData.name && jsonData.name.length >
0);
}
```

示例2：构建 JSON 请求体

本示例演示如何将 JavaScript 对象序列化为 JSON 字符串用于 POST 请求，适用于需要发送复杂数据结构的场景。

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

```
export default function () {
    // 构建请求数据对象
    const requestData = {
        name: "pts",
        language: "javascript",
        version: "1.0",
        features: ["http", "websocket", "grpc"]
    };

    // 序列化为 JSON 字符串
    const jsonStr = JSON.stringify(requestData);
    console.log(jsonStr); //
{"name":"pts","language":"javascript","version":"1.0","features":
["http","websocket","grpc"]}

    // 发送 POST 请求
    const resp = http.post('http://example.com/api/data', {
        headers: {
            'Content-Type': 'application/json'
        },
        body: jsonStr
    });

    console.log(resp.statusCode);
}
```

示例3：处理嵌套 JSON 和错误处理

本示例演示如何处理嵌套的 JSON 结构，并使用 try-catch 处理解析错误。

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

export default function () {
    const resp = http.get('http://example.com/api/complex');

    try {
        // 解析可能包含嵌套结构的 JSON
        const data = JSON.parse(resp.body);

        // 访问嵌套字段
```

```
        if (data.user && data.user.profile) {
            console.log(data.user.profile.email);
        }

        // 处理数组
        if (Array.isArray(data.items)) {
            data.items.forEach(item => {
                console.log(item.name);
            });
        }
    } catch (error) {
        console.error('JSON 解析失败:', error.message);
        console.log('原始响应:', resp.body);
    }
}
```

Math 对象

函数说明

`Math` 对象提供数学运算方法，在压测脚本中常用于：

- 生成随机数用于参数化测试。
- 数值计算和转换。
- 实现随机延迟、随机选择等逻辑。
- 数学运算和比较。

前提条件

- 所有方法都是静态方法，通过 `Math.方法名()` 调用。
- 随机数生成基于伪随机算法。

常用方法

- `Math.random()`：返回 0 到 1 之间的随机浮点数。
- `Math.floor(x)`：向下取整。
- `Math.ceil(x)`：向上取整。
- `Math.round(x)`：四舍五入。
- `Math.pow(x, y)`：返回 x 的 y 次幂。
- `Math.max(...values)`：返回最大值。
- `Math.min(...values)`：返回最小值。

返回值说明

- `Math.random()`：返回 `[0, 1)` 区间的浮点数。
- 其他方法根据具体功能返回相应的数值。

使用限制

- `Math.random()` 生成的是伪随机数，不适合用于加密场景。
- 数值运算需注意精度问题，浮点数运算可能存在精度误差。

示例1：生成随机数用于参数化

本示例演示如何生成随机整数和浮点数，用于参数化测试场景，如随机用户 ID、随机金额等。

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

export default function () {
    // 生成 0 到 9 之间的随机整数
    const randomInt = Math.floor(Math.random() * 10);
    console.log(randomInt);

    // 生成指定范围内的随机整数（如 100 到 999）
    const min = 100;
    const max = 999;
    const randomInRange = Math.floor(Math.random() * (max - min + 1)) +
min;
    console.log(randomInRange);

    // 在请求中使用随机参数
    const resp = http.get('http://example.com/api/user', {
        query: {
            'userId': randomInRange.toString()
        }
    });
}
```

示例2：数学运算和幂运算

本示例演示如何使用 Math 对象进行数学运算，适用于需要数值计算的场景。

```
export default function () {
    // 计算幂
```

```
    const result = Math.pow(2, 10);
    console.log(result); // 1024

    // 取最大值和最小值
    const numbers = [10, 20, 5, 30, 15];
    const max = Math.max(...numbers);
    const min = Math.min(...numbers);
    console.log(`最大值: ${max}, 最小值: ${min}`);

    // 四舍五入
    const rounded = Math.round(3.7);
    console.log(rounded); // 4
}
```

示例3：实现随机延迟和随机选择

本示例演示如何使用随机数实现随机延迟和随机选择逻辑，用于模拟真实用户行为。

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

export default function () {
    // 生成 1 到 5 秒之间的随机延迟
    const randomDelay = Math.floor(Math.random() * 4000) + 1000;
    sleep(randomDelay / 1000); // sleep 使用秒为单位

    // 随机选择操作
    const operations = ['read', 'write', 'delete'];
    const randomOperation = operations[Math.floor(Math.random() *
operations.length)];
    console.log(`执行操作: ${randomOperation}`);

    // 根据随机选择执行不同请求
    if (randomOperation === 'read') {
        http.get('http://example.com/api/read');
    } else if (randomOperation === 'write') {
        http.post('http://example.com/api/write', {
            body: JSON.stringify({ data: 'test' })
        });
    }
```

```
}
```

RegExp 对象

函数说明

`RegExp` 对象用于正则表达式匹配，在压测脚本中常用于：

- 从响应中提取特定数据。
- 验证数据格式。
- 字符串替换和转换。
- 模式匹配和验证。

前提条件

- 正则表达式语法需符合 JavaScript 标准。
- 可以使用字面量形式 `/pattern/` 或构造函数 `new RegExp()` 。

函数签名

```
/pattern/flags
new RegExp(pattern: string, flags?: string): RegExp
```

常用方法

- `test(str)` ：测试字符串是否匹配，返回布尔值。
- `exec(str)` ：执行匹配，返回匹配结果数组或 null。
- `match(regexp)` ：字符串方法，返回匹配结果。
- `replace(regexp, replacement)` ：字符串方法，替换匹配内容。
- `search(regexp)` ：字符串方法，返回匹配位置索引。

返回值说明

- `test()` ：返回 `true` 或 `false` 。
- `exec()` ：返回匹配结果数组（包含匹配内容和捕获组）或 `null` 。
- `match()` ：返回匹配结果数组或 `null` 。
- `replace()` ：返回替换后的新字符串。
- `search()` ：返回匹配位置的索引或 `-1` 。

使用限制

- 复杂正则表达式可能影响性能，建议优化表达式。

- 全局标志 `g` 会影响 `exec()` 和 `test()` 的行为。
- 需注意特殊字符的转义。

示例1：从响应中提取数据

本示例演示如何使用正则表达式从 HTTP 响应中提取特定数据，如提取 token、ID 等。

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

export default function () {
    const resp = http.get('http://example.com/api/data');

    // 使用正则表达式提取邮箱地址
    const emailRegex = /\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]
{2,}\b/g;
    const emails = resp.body.match(emailRegex);

    if (emails) {
        console.log('找到邮箱:', emails);
    }

    // 提取数字 ID
    const idRegex = /"id":\s*(\d+)/;
    const match = resp.body.match(idRegex);
    if (match) {
        const id = match[1];
        console.log('提取的 ID:', id);
    }
}
```

示例2：字符串替换和格式化

本示例演示如何使用正则表达式进行字符串替换，适用于数据格式转换场景。

```
export default function () {
    // 替换姓名格式：从 "John Smith" 转换为 "Smith, John"
    let re = /(\w+)\s(\w+)/;
    let str = "John Smith";
    let newStr = str.replace(re, "$2, $1");
    console.log(newStr); // Smith, John
```

```
    // 移除字符串中的空格
    const text = "Hello World Test";
    const noSpaces = text.replace(/\s/g, "");
    console.log(noSpaces); // HelloWorldTest

    // 格式化电话号码
    const phone = "13812345678";
    const formatted = phone.replace(/(\d{3})(\d{4})(\d{4})/,
"$1-$2-$3");
    console.log(formatted); // 138-1234-5678
}
```

示例3：数据验证和检查

本示例演示如何使用正则表达式验证数据格式，适用于参数校验场景。

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

export default function () {
    // 验证邮箱格式
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    const email = "user@example.com";
    const isValidEmail = emailRegex.test(email);
    check('邮箱格式正确', () => isValidEmail);

    // 验证手机号格式（11位数字）
    const phoneRegex = /^1[3-9]\d{9}$/;
    const phone = "13812345678";
    const isValidPhone = phoneRegex.test(phone);
    check('手机号格式正确', () => isValidPhone);

    // 在请求中使用验证后的数据
    if (isValidEmail && isValidPhone) {
        const resp = http.post('http://example.com/api/register', {
            body: JSON.stringify({
                email: email,
                phone: phone
            })
```

```
        });
    }
}
```

PTS 扩展库

PTS 扩展库提供了额外的工具函数，用于增强脚本功能。使用前需通过 `import util from 'pts/util'` 导入模块。

Base64编码解码

函数说明

`util.base64Encoding()` 和 `util.base64Decoding()` 用于 Base64 编码和解码，在压测脚本中常用于：

- 对请求参数进行 Base64 编码。
- 解码响应中的 Base64 数据。
- 处理二进制数据的文本传输。
- 实现认证和加密相关功能。

前提条件

- 需先导入 `pts/util` 模块：`import util from 'pts/util'`。
- 所有代码需放在 `export default function()` 函数中执行。
- 编码输入可以是字符串或 ArrayBuffer。
- 解码输出可以是字符串或 ArrayBuffer（通过 mode 参数控制）。

函数签名

```
base64Encoding(input: string | ArrayBuffer, encoding?: "std" | "rawstd"
| "url" | "rawurl"): string
base64Decoding(input: string, encoding?: "std" | "rawstd" | "url" |
"rawurl", mode?: "b"): string | ArrayBuffer
```

参数说明

util.base64Encoding()

- `input`（string | ArrayBuffer，必填）：要编码的字符串或字节数组。
- `encoding`（string，可选）：编码方式，可选值：
  - `"std"`：标准 Base64编码（默认），符合 RFC 4648标准。
  - `"rawstd"`：标准原始编码，无填充字符。

- `"url"`：URL 安全编码，适用于 URL 和文件名。
- `"rawurl"`：URL 安全原始编码，无填充字符。

util.base64Decoding()

- `input`（string，必填）：要解码的 Base64 字符串。
- `encoding`（string，可选）：编码方式，需与编码时使用的编码方式一致，默认为 `"std"`。
- `mode`（string，可选）：返回类型控制，不设置返回 string，设置为 `"b"` 返回 ArrayBuffer。

返回值说明

- `base64Encoding()`：返回 Base64 编码后的字符串。
- `base64Decoding()`：根据 `mode` 参数返回字符串或 ArrayBuffer。

使用限制

- 编码和解码需使用相同的 `encoding` 参数，否则可能无法正确解码。
- URL 编码方式适用于需要在 URL 中使用的场景。
- ArrayBuffer 模式适用于处理二进制数据。

示例1：基本编码解码

本示例演示基本的 Base64编码和解码操作，适用于简单的数据转换场景。

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

export default function () {
    // Base64 编码
    const base64Encoded = util.base64Encoding("Hello, world");
    console.log(base64Encoded); // SGVsbG8sIHdvcmxk

    // Base64 解码
    const base64Decoded = util.base64Decoding(base64Encoded);
    console.log(base64Decoded); // Hello, world
}
```

示例2：在 HTTP 请求中使用 Base64编码

本示例演示如何在 HTTP 请求中使用 Base64编码，适用于需要传递编码参数的场景，如 Basic 认证、API 签名等。

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

```
export default function () {
    // 对用户名和密码进行 Base64 编码用于 Basic 认证
    const credentials = "username:password";
    const encodedCredentials = util.base64Encoding(credentials);

    // 在请求头中使用
    const resp = http.get('http://example.com/api/protected', {
        headers: {
            'Authorization': `Basic ${encodedCredentials}`
        }
    });

    console.log(resp.statusCode);
}
```

示例3：使用不同的编码方式

本示例演示如何使用不同的 Base64编码方式，适用于不同场景的需求。

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

export default function () {
    const text = "http://www.example.com";

    // 标准编码（默认）
    const stdEncoded = util.base64Encoding(text);
    console.log('标准编码:', stdEncoded);

    // URL 安全编码（适用于 URL）
    const urlEncoded = util.base64Encoding(text, 'url');
    console.log('URL 编码:', urlEncoded);

    // 解码时需使用相同的编码方式
    const urlDecoded = util.base64Decoding(urlEncoded, 'url');
    console.log('URL 解码:', urlDecoded); // http://www.example.com
}
```

示例 4：处理 ArrayBuffer 数据

本示例演示如何使用 ArrayBuffer 模式处理二进制数据。

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

export default function () {
    // 编码字符串
    const encoded = util.base64Encoding("Hello, world");

    // 解码为 ArrayBuffer
    const arrayBuffer = util.base64Decoding(encoded, 'std', 'b');
    console.log('ArrayBuffer 类型:', arrayBuffer);

    // 如果需要，可以将 ArrayBuffer 转换回字符串
    // 注意：实际使用时需根据具体需求处理 ArrayBuffer
}
```

UUID 生成

函数说明

`util.uuid()` 用于生成 UUID（通用唯一标识符），在压测脚本中常用于：

- 生成唯一的请求 ID。
- 创建唯一的用户标识。
- 生成唯一的订单号、会话 ID 等。
- 实现去重和追踪功能。

前提条件

- 需先导入 `pts/util` 模块：`import util from 'pts/util'`。
- 所有代码需放在 `export default function()` 函数中执行。
- 生成的 UUID 符合 UUID v4 标准（随机 UUID）。

函数签名

```
uuid(): string
```

参数说明

无需参数。

返回值说明

返回 UUID v4 格式的字符串，格式为：`xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx`，其中：

- `x` 为十六进制数字。
- `y` 为 8、9、a 或 b 之一。

示例：`5fbf1e59-cabf-469b-9d9f-6622e97de1ec`。

使用限制

- 每次调用都会生成新的 UUID，不会重复。
- UUID 是随机生成的，不保证顺序性。
- 适用于需要唯一标识的场景，不适合用于加密。

示例1：生成唯一请求 ID

本示例演示如何生成 UUID 作为请求的唯一标识，适用于需要追踪和日志记录的场景。

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

export default function () {
    // 生成唯一请求 ID
    const requestId = util.uuid();
    console.log('请求 ID:', requestId); // 5fbf1e59-cabf-469b-9d9f-
6622e97de1ec

    // 在请求头中传递请求 ID
    const resp = http.get('http://example.com/api/data', {
        headers: {
            'X-Request-ID': requestId
        }
    });

    console.log('响应状态:', resp.statusCode);
}
```

示例2：创建唯一用户标识

本示例演示如何使用 UUID 创建唯一的用户标识，适用于模拟多用户压测场景。

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

```
export default function () {
    // 为每个虚拟用户生成唯一 ID
    const userId = util.uuid();
    console.log('用户 ID:', userId);

    // 使用唯一用户 ID 进行注册
    const registerResp = http.post('http://example.com/api/register', {
        body: JSON.stringify({
            userId: userId,
            username: `user_${userId.substring(0, 8)}`,
            email: `user_${userId.substring(0, 8)}@example.com`
        })
    });

    // 使用该用户 ID 进行后续操作
    if (registerResp.statusCode === 200) {
        const loginResp = http.post('http://example.com/api/login', {
            body: JSON.stringify({
                userId: userId
            })
        });
    }
}
```

示例3：生成唯一订单号和会话 ID

本示例演示如何使用 UUID 生成订单号、会话 ID 等业务唯一标识，适用于电商、支付等场景的压测。

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

export default function () {
    // 生成会话 ID
    const sessionId = util.uuid();
    console.log('会话 ID:', sessionId);

    // 创建购物车会话
    const cartResp = http.post('http://example.com/api/cart', {
        headers: {
            'X-Session-ID': sessionId
```

```
        },
        body: JSON.stringify({
            sessionId: sessionId,
            items: []
        })
    });

    // 生成订单号
    const orderId = util.uuid();
    console.log('订单号:', orderId);

    // 提交订单
    const orderResp = http.post('http://example.com/api/order', {
        body: JSON.stringify({
            orderId: orderId,
            sessionId: sessionId,
            amount: 100.00
        })
    });

    console.log('订单状态:', orderResp.statusCode);
}
```

HTTP 协议压测
基本用法

最近更新时间：2024-06-07 17:20:32

PTS 支持 HTTP 协议的 GET、POST、PUT、PATCH、OPTIONS、DELETE 和 HEAD 请求。

脚本编写

HTTP GET 请求

```
// Send a http get request
import http from 'pts/http';
import { check, sleep } from 'pts';

export default function () {
    // simple get request
    const resp1 = http.get('http://httpbin.org/get');
    console.log(resp1.body);
    // if resp1.body is a json string, resp1.json() transfer json format
body to a json object
    console.log(resp1.json());
    check('status is 200', () => resp1.statusCode === 200);

    // sleep 1 second
    sleep(1);

    // get request with headers and parameters
    const resp2 = http.get('http://httpbin.org/get', {
        headers: {
            'Connection': 'keep-alive',
            'User-Agent': 'pts-engine'
        },
        query: {
            'name1': 'value1',
            'name2': 'value2',
        }
    });
    console.log(resp2.json().args.name1); // 'value1'
```

```
    check('body.args.name1 equals value1', () => resp2.json().args.name1
=== 'value1');
};
```

HTTP POST 请求

```
// Send a post request
import http from 'pts/http';
import { check } from 'pts';

export default function () {
    const resp = http.post(
        'http://httpbin.org/post',
        {
            user_id: '12345',
        },
        {
            headers: {
                'Content-Type': 'application/json',
            },
        }
    );

    console.log(resp.json().json.user_id); // 12345
    check('body.json.user_id equals 12345', () =>
resp.json().json.user_id === '12345');
}
```

文件依赖

在压测场景里，您可上传以下几种类型的文件，提供压测执行时的状态数据：

- 参数文件：以 csv 文件的形式，动态提供测试数据。也即，场景被每个并发用户（VU）执行时，会获取参数文件里的每行数据，作为测试数据的值，供脚本里的变量引用。具体使用方法参见：使用参数文件 。
- 请求文件：构建您的请求所需的文件，如需要上传的文件。具体使用方法参见：使用请求文件 。
- 协议文件：请求序列化所需要用到的文件。具体使用方法参见：使用协议文件 。

配置选项

最近更新时间：2024-11-08 15:33:22

PTS 支持的配置选项

HTTP 的请求配置，可在全局变量 option 中定义。PTS 支持的配置选项如下表所示：

HTTP 请求配置选项	描述
maxRedirects	重定向跳转次数
maxIdleConns	单个 VU 最大活跃连接数
maxIdleConnsPerHost	单个 VU 单个域名最大活跃连接数
disableKeepAlives	禁用长连接
headers	公共请求头
timeout	请求超时时间，单位毫秒
basicAuth	基本认证
http2	是否开启 http2
discardResponseBody	是否丢弃回包，当压测业务不关注回报，可开启此开关，提升压测性能

说明：
完整选项列表请参见 HTTP global Options 。

配置 HTTP 请求的超时时间

```
import http from "pts/http";

export const option = {
    http: {
        timeout: 3000,
    }
}
```

```
export default function() {
    http.get("http://httpbin.org/get"); // Error: Get
"http://httpbin.org/get": net/http: request canceled while waiting for
connection
}
```

配置 HTTP 请求的基本认证

```
// HTTP basic authentication
import http from 'pts/http';
import { check } from 'pts';

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

export default function () {
  const resp = http.get(`http://httpbin.org/basic-auth/user/passwd`);
  console.log(resp.json().authenticated); // true
  check('body.authenticated equals true', () =>
resp.json().authenticated === true);
}
```

gRPC 协议压测

最近更新时间：2024-06-07 17:20:32

本文将介绍 gRPC 协议请求的编排方法。

基本用法

使用 pts/grpc API 提供的接口，您可以创建 gRPC client，发送 gRPC 请求。

协议上传

将您定义好的 proto 文件，通过**云压测 > 测试场景 > 新建测试场景 > 文件管理 > 协议文件**上传。

> **说明：**
> 关于 PTS 支持的协议类型及使用方法，请参见 使用协议文件 。

脚本编写

先创建一个 gRPC client，然后您可以使用 client 提供的以下方法编写您的逻辑：

- `load`：加载并解析您上传的 proto 文件。
- `connect`：与 gRPC 服务器建立连接。
- `invoke`：发起 RPC 调用并获得响应。
- `close`：关闭连接。

Proto 文件及场景脚本的示例如下：

```
// based on https://github.com/go-
kit/kit/blob/master/examples/addsvc/pb/addsvc.proto

syntax = "proto3";

package addsvc;

// The Add service definition.
service Add {
  // Sums two integers.
  rpc Sum (SumRequest) returns (SumReply) {}
}

// The sum request contains two parameters.
message SumRequest {
```

```
  int64 a = 1;
  int64 b = 2;
}

// The sum response contains the result of the calculation.
message SumReply {
  int64 v = 1;
  string err = 2;
}
```

场景脚本：

```
// GRPC API
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');

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

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

  client.close();
};
```

文件依赖

在压测场景里，您可上传以下几种类型的文件，提供压测执行时的状态数据：

- 参数文件：以 csv 文件的形式，动态提供测试数据。也即，场景被每个并发用户（VU）执行时，会获取参数文件里的每行数据，作为测试数据的值，供脚本里的变量引用。具体使用方法参见：使用参数文件 。
- 请求文件：构建您的请求所需的文件，如需要上传的文件。具体使用方法参见：使用请求文件 。
- 协议文件：请求序列化所需要用到的文件。具体使用方法参见：使用协议文件 。

Protobuf 协议压测

最近更新时间：2024-06-12 16:22:41

本文将介绍 Protobuf 序列化协议的使用方法。

协议上传

使用 Protobuf 协议需要用户上传 Proto 协议文件，压测引擎依赖协议文件完成请求的序列化。支持用户上传文件或目录，文件名需要保持唯一，同名文件将会被新上传的文件覆盖。如果用户上传 zip 文件，PTS 会解压文件，并展示解压后的文件结构。如果目录或者 zip 包中包含非 Proto 文件，PTS 将忽略这些文件。

> **说明：**
> 多协议文件请参见 使用协议文件 。

借助您上传的 proto 文件，您可对脚本中的对象做序列化/反序列化。如果 demo.proto 依赖其他 proto 文件，那么也需要一并上传（谷歌提供的标准 proto 文件：google/protobuf/*.proto 不需要额外上传，PTS 会自动加载）。用户只需要加载主 pb 即可，主 pb 依赖的其他 pb 文件，会自动递归加载。

示例

协议文件

duty.proto

```
syntax = "proto3";

message Duty {
  string time = 1;
```

```
  string work = 2;
}
```

student.proto

```
syntax = "proto3";
import "duty.proto";

package student;

message Student {
  string name = 1;
  Gender gender = 2;

  message GradeInfo {
    enum Grade {
      DEFAULT = 0;
      FIRST = 1;
      SECOND = 2;
      THIRD = 3;
    }
    Grade grade = 1;
  }
  GradeInfo gradeInfo = 3;

  map<string, int32> scores = 4;
  repeated Duty duties = 5;
}

enum Gender {
  DEFAULT = 0;
  FEMALE = 1;
  MALE = 2;
}

message SearchRequest {
  string id = 1;
}
```

```
message SearchResponse {
  message Result {
    Student student = 1;

  }
}

service SearchService {
  rpc SearchScores (SearchRequest) returns (SearchResponse);
}
```

脚本

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


// 加载协议文件根目录中的 student.proto，同时会加载 duty.proto
protobuf.load([], 'student.proto');


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


export default function () {
    let bodyBuffer = protobuf.marshal('student.Student', {
        'name': 'Alice',
        'gender': 1, // 或者 'FEMALE'，enum 直接设置具体值即可
        'gradeInfo': {
            'grade': 'THIRD'
        },
        'scores': {
            'Chinese': 116,
            'Math': 120,
            'English': 106
        },
        'duties': [
            {
                'time': 'time1',
```

```
                'work': 'work1'
            },
            {
                'time': 'time2',
                'work': 'work2'
            }
        ]
    });

    const value = protobuf.unmarshal('student.Student', bodyBuffer);
    // {"name":"Alice","gender":"FEMALE","gradeInfo":
{"grade":"THIRD"},"scores":
{"Math":120,"Chinese":116,"English":106},"duties":
[{"time":"time","work":"work"}]}
    console.log(JSON.stringify(value));
}
```

WebSocket 协议压测

最近更新时间：2025-08-26 11:02:52

本介绍基于 WebSocket 协议的压测脚本的编写方法。

> **说明：**
> 详细的 API 文档请参见 PTS API 。

概述

- WebSocket 是一种应用层通信协议，可在单个 TCP 连接上进行全双工通信。
- 不同于 HTTP 请求的客户端发起、服务端响应的一问一答模式，WebSocket 连接一旦建立，直到连接关闭之前，客户端、服务器之间都可源源不断地、双向地交换数据。因此，在压测场景中，基于 WebSocket 请求的脚本与基于 HTTP 请求的脚本，其结构和作用机制有所不同：
  - 执行 HTTP 脚本的每个 VU 会持续不断地迭代主函数（`export default function() { ... }`），直到压测结束。
  - 执行 WebSocket 脚本的每个 VU 不会持续迭代主函数，因为主函数会被建立连接的 `ws.connect` 方法阻塞，直到连接关闭。而在连接建立后的回调函数里（`function (socket) {...}`），会持续不断地监听和处理异步事件，直到压测结束。

脚本编写

PTS API 的 ws 模块提供了 WebSocket 协议的相关接口，请参见 pts/ws API 。

基本用法

- 用 `ws.connect` 方法建立连接，并在其回调函数里定义您的业务逻辑：
  - `ws.connect` 的必传参数为 URL 和回调函数。
  - 若连接建立成功，PTS 会将创建好的 `ws.Socket` 对象传入回调函数。您可在回调函数里，定义您的 WebSocket 请求逻辑。
  - 执行完回调函数，`ws.connect` 会返回 `ws.Response` 对象。
- `ws.Socket` 对象的常用方法：
  - `send`：发送文本消息。
  - `close`：关闭连接。
  - `on`：监听事件，并用回调函数处理事件。当前 PTS 支持的事件列表如下：

事件名	事件用途
open	建立连接
close	关闭连接

error	发生错误
message	接收文本消息
binaryMessage	接收二进制消息
pong	接收 pong 消息
ping	接收 ping 消息

代码示例如下：

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

文件依赖

在压测场景里，您可上传以下几种类型的文件，提供压测执行时的状态数据：

- 参数文件：以 csv 文件的形式，动态提供测试数据。也即，场景被每个并发用户（VU）执行时，会获取参数文件里的每行数据，作为测试数据的值，供脚本里的变量引用。具体使用方法请参见 使用参数文件 。
- 请求文件：构建您的请求所需的文件，如需要上传的文件。具体使用方法请参见 使用请求文件 。
- 协议文件：请求序列化所需要用到的文件。具体使用方法请参见 使用协议文件 。

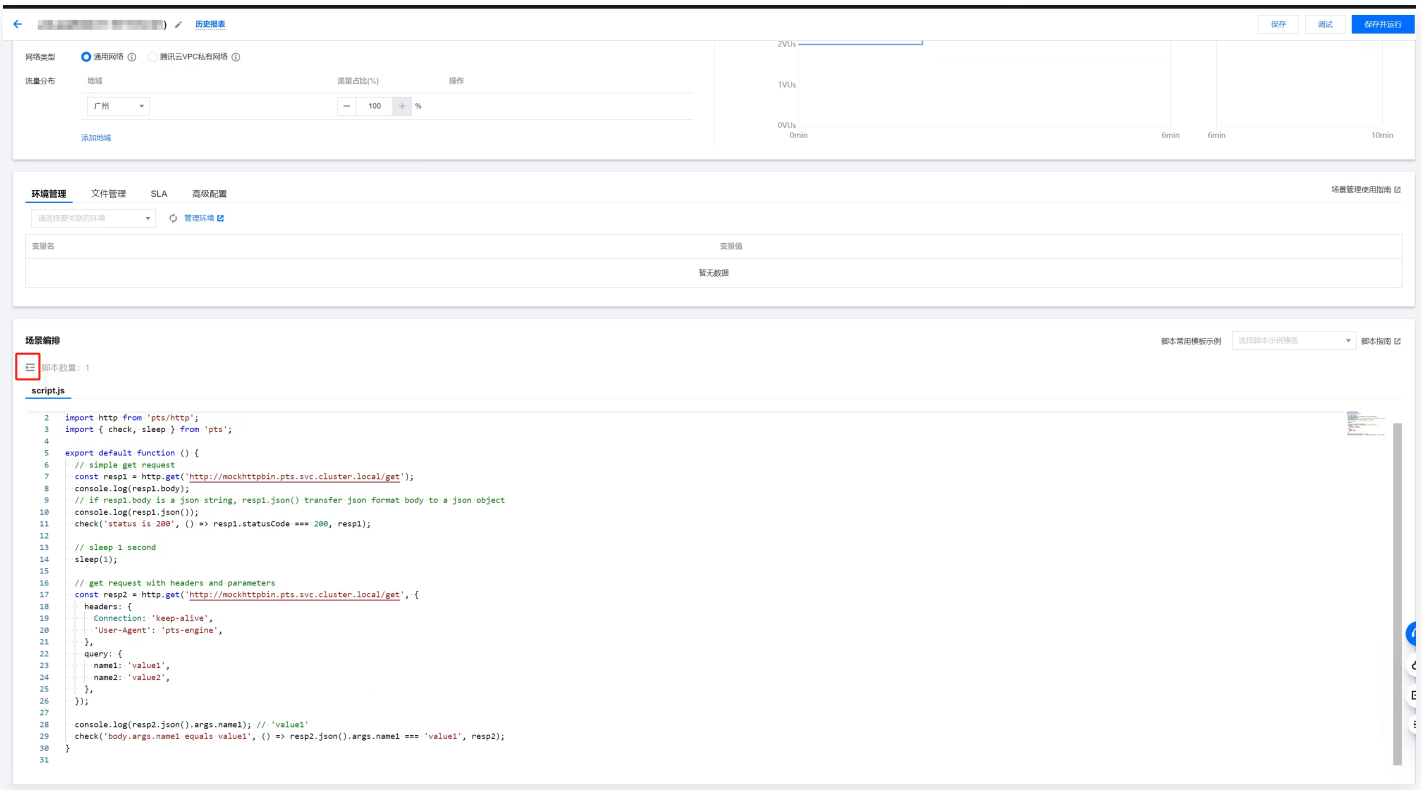
多脚本压测

最近更新时间：2024-08-16 15:02:21

在同一场景中，若您需要配置多个脚本，并且基于不同权重，按比例分配各脚本的 VU 数，则您可使用多脚本压测模式，实现更灵活的脚本定制及施压配置。

基本用法

登录 腾讯云可观测平台，在左侧导航栏选择**云压测 >测试场景**， 点击**新建场景**。在脚本模式的场景中，您可如下图所示，点开多脚本导航栏，添加新的脚本：



在新建脚本的页面，您需填写新脚本的名称以及权重：

若您需要修改原有脚本，可在多脚本导航栏右侧对该脚本进行编辑，还可以删除脚本。

压力分配规则

当压测任务执行时，PTS 将计算各个脚本的权重占所有脚本总权重的百分比，来分配施压力度：

- **在并发模式下，各脚本的 VU 将按该脚本权重占总权重的比例分配。**
- **在 RPS 模式下，各脚本的 RPS 将按该脚本权重占总权重的比例分配。**

SQL 数据库压测

最近更新时间：2024-07-15 15:26:51

本文介绍云压测 SQL 数据库的脚本编排方法，用于云压测支持 SQL 语言的关系型数据库（如 MySQL 等）。

基本用法

使用 pts/sql API 提供的接口，您可创建连接 SQL 数据库的客户端，发送 DDL 和 DML 请求，对数据库做基本的增删改查等操作。

数据库连接

建立数据库连接可调用 `new sql.Database(driverName: string, dataSourceName: string)` 方法。其中，`driverName` 参数用于指定数据库驱动程序，`dataSourceName` 参数用于指定数据源。
脚本示例如下：

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

const db = new sql.Database(sql.MySQL,
"user:passwd@tcp(ip:port)/database")

export default function () {
    // 向数据库发送请求
}
```

> **说明：**
> - 若数据库表中带有日期或时间字段，建立数据库连接时，建议在连接串中加入 parserTime 参数（例如：`user:passwd@tcp(ip:port)/database?parseTime=true`），避免时间解析出错。
> - 建议将上述建立数据库连接的语句，作为全局变量放在主函数外部（如上述示例），以供同一个 VU 在迭代执行主函数时能够复用连接，避免多次重复创建数据库连接，带来不必要的资源消耗。

SQL 查询

SQL 查询可调用 `db.query(sql string)` 方法，返回符合条件的数据库记录数组。其中，`sql` 参数代表传入的 SQL 查询语句。
脚本示例如下：

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

```
const db = new sql.Database(sql.MySQL,
"user:passwd@tcp(ip:port)/database")

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

SQL 执行

SQL 执行可调用 `db.exec(sql string)` 方法，传入执行语句，返回本次执行对数据库的影响（返回字段包括：最后插入行的 ID、所有受影响的行数）。
其中，`sql` 参数代表传入的 SQL 执行语句。
脚本示例如下：

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

const db = new sql.Database(sql.MySQL,
"user:passwd@tcp(ip:port)/database")

export default function () {
    // 修改数据
    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}

    // 插入数据
    let result1 = db.exec("insert into user (name, age) values
('wanger', 18)");
    console.log(JSON.stringify(result)); //
{"lastInsertId":66,"rowsAffected":1}
}
```

说明：

> `db.exec` 方法支持常见的 DDL 命令（如 create、drop、alter）和常见的 DML 命令（如 insert、update、delete）。

脚本示例

一个包含数据库基本操作及检查点使用的完整脚本示例如下：

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

const db = new sql.Database(sql.MySQL,
"user:passwd@tcp(ip:port)/database")

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

    // 新增数据
    let result = db.exec("insert into user (name, age) values ('wanger',
18)");
    console.log(JSON.stringify(result)); //
{"lastInsertId":66,"rowsAffected":1}

    // 删除数据
    let result1 = db.exec("delete from user where id > 8");
    console.log(JSON.stringify(result)); //
{"lastInsertId":0,"rowsAffected":2}

    // 修改数据
    let result2 = db.exec("UPDATE user SET age=? WHERE name='zhangsan'",
Math.floor(Math.random() * 100));
    console.log(JSON.stringify(result)); //
{"lastInsertId":0,"rowsAffected":1}

    // 设置检查点
    check("1 row returned", () => result.rowsAffected === 1);

    sleep(1)
```

```
}
```

脚本验证

若要验证脚本执行结果，可在正式压测前，先使用 PTS 调试功能，快速验证结果是否符合预期。
更多详情，可参见 调试场景 。

Socket.IO 框架压测

最近更新时间：2024-07-12 15:43:01

基本概念

Socket.IO

Socket.IO 是一个面向 Web 端即时通信技术的代码工具库，它主要基于 WebSocket 协议建立连接，同时也把 HTTP 长轮询 作为后备方案，支持即时、双向、基于事件的通信。其主要特性如下：

- **高性能：**在大多数情况下，它使用 WebSocket 协议建立连接，在客户端和服务器之间，提供双向、低延迟的通信通道。
- **可靠性：**在无法建立 WebSocket 连接时（例如浏览器不支持 WebSocket 协议、或 WebSocket 连接被代理或防火墙拦截等），它将使用 HTTP 长轮询作为替代方案。另外，如果连接丢失，客户端将自动重试连接。
- **可扩展：**支持将应用程序部署到多个服务器，向所有连接的客户端发送事件。

WebSocket

WebSocket 是一种应用层通信协议，可在单个 TCP 连接上进行全双工通信。
不同于 HTTP 协议的客户端发起请求、服务端响应的一问一答模式，WebSocket 连接一旦建立，直到连接关闭之前，客户端、服务器之间都可源源不断地、双向地交换数据。

HTTP 长轮询

HTTP 长轮询是一种基于 HTTP 协议的 Web 端即时通信技术。当服务器收到客户端发来的请求后，不会直接进行响应，而是先将这个请求挂起，判断服务器端数据是否有更新：

- 如果有数据更新，则服务器正常返回响应。
- 如果一直没有数据更新，则达到服务器端设置的超时时间后返回。

客户端则会在处理完服务器返回的响应后，再次发出请求，重新建立连接。
HTTP 长轮询和 HTTP 短轮询相比较，HTTP 长轮询在无数据更新的情况下，不会频繁发送请求，减少了不必要的请求次数、节约了资源。

Socket.IO 压测介绍

在 PTS 压测场景中，基于 Socket.IO 请求的脚本与基于 HTTP 请求的脚本，其结构和作用机制有所不同：

- 执行 HTTP 脚本的每个 VU 会持续不断地迭代主函数（`export default function() { ... }`），直到压测结束。
- 执行 Socket.IO 脚本的每个 VU 不会持续迭代主函数，因为主函数会被建立连接的 `io.connect` 方法阻塞，直到连接关闭。而在连接建立后的回调函数里（`function (socket) {...}`），会持续不断地监听和处理异步事件，直到压测结束。

脚本编写

PTS API 的 Socket.IO 模块提供了 Socket.IO 协议的相关接口，详情可参见 socketio API 。
使用这些接口，您可以建立 Socket.IO 连接、发送/收取事件消息。
基本用法：

- 用 `connect` 方法建立连接，并在其回调函数里定义您的业务逻辑：
  - 该方法的必传参数为待连接服务的 URL，和连接成功后需执行的回调函数。
  - 该方法的可选参数为调整配置的选项，包括：
    - `protocol`：协议类型，支持 polling（HTTP 长轮询）和 websocket（WebSocket）。
    - `headers`：请求头参数。
  - 若连接建立成功，PTS 会将创建好的 `SocketIO` 对象传入回调函数。您可在回调函数里，定义您的 Socket.IO 请求逻辑，发送/收取事件消息。
  - 执行完回调函数，`connect` 会返回 `Response` 对象，200 为成功返回码。
- `SocketIO` 对象的常用方法：
  - `emit`：发送事件。参数为事件名、消息数据、回调函数：
    - `event`：自定义事件的名称。
    - `msg`：文本消息或二进制数据。
    - `callback`：（可选）回调函数。
  - `close`：关闭连接。
  - `on`：监听事件，并用回调函数处理事件。PTS 支持监听的事件列表如下：

事件名	事件用途
open	建立连接
close	关闭连接
error	发生错误
message	接收文本消息
binaryMessage	接收二进制消息

  - `setTimeout`：等待 intervalMs 毫秒后执行函数。
  - `setInterval`：按照 intervalMs 毫秒定期执行函数。
  - `setLoop`：循环执行函数直至 context 结束或者连接关闭。

代码示例如下：

```
// 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);
}
```

脚本验证

若要验证脚本执行结果，可在正式压测前，先使用 PTS 调试功能，快速验证结果是否符合预期。更多详情可参见 调试场景 。

文件依赖

在压测场景里，您可上传以下几种类型的文件，提供压测执行时的状态数据：

- 参数文件：以 csv 文件的形式，动态提供测试数据。也即，场景被每个并发用户（VU）执行时，会获取参数文件里的每行数据，作为测试数据的值，供脚本里的变量引用。具体使用方法参见：使用参数文件 。
- 请求文件：构建您的请求所需的文件，如需要上传的文件。具体使用方法参见：使用请求文件 。
- 协议文件：请求序列化所需要用到的文件。具体使用方法参见：使用协议文件 。

TCP/UDP 协议压测

最近更新时间：2024-08-16 11:52:01

Socket 基本概念

Socket 是一种操作系统提供的进程之间的通信机制。操作系统会提供一组封装了 TCP/IP 协议的 Socket 接口，进程便可通过这些接口来使用 Socket、收发网络数据。
在使用 Socket 接口时，一个进程的 IP 地址加上端口构成一个 Socket 地址；客户端与服务器双方进程的 Socket 地址再加上传输协议（TCP 或 UDP），就构成了 Socket 五元组，标识一次网络通信。

脚本编写

- PTS API 的 socket 模块提供了支持 Socket 的相关接口，详见 Socket 概览 。
- 使用这些接口，您可以建立 Socket 实例，然后通过该实例发送或接收 TCP/UDP 数据。

基本用法：

1. 创建 Socket 实例

   通过 `new socket.Conn` 方法，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。
2. 使用 Socket 实例

- 发送数据：通过实例的 `send` 方法发送数据。参数为二进制数据，返回值为发送的字节数。
- 接收数据：通过实例的 `recv` 方法接收数据。参数为接收的字节数，返回值为接收的二进制数据。
- 关闭连接：通过实例的 `close` 方法关闭连接。

  使用 TCP 协议的脚本示例：

```
// tcp connect to send package
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);
}
```

使用 UDP 协议的脚本示例：

```
// udp connect to send package
import socket from "pts/socket";
import util from 'pts/util';

export default function main() {
    const udp_socket = new socket.Conn('udp', '127.0.0.1', 20001);
    const send_data = `test data`;
    udp_socket.send(util.toArrayBuffer(send_data));
    const bytes_read = udp_socket.recv(1024);
    udp_socket.close();
    console.log(bytes_read);
}
```

脚本验证

若要验证脚本执行结果，可在正式压测前，先使用 PTS 调试功能，快速验证结果是否符合预期。更多详情可参见 调试场景 。

文件依赖

在压测场景里，您可上传以下几种类型的文件，提供压测执行时的状态数据：

- 参数文件：以 csv 文件的形式，动态提供测试数据。也即，场景被每个并发用户（VU）执行时，会获取参数文件里的每行数据，作为测试数据的值，供脚本里的变量引用。具体使用方法参见：使用参数文件 。
- 请求文件：构建您的请求所需的文件，如需要上传的文件。具体使用方法参见：使用请求文件 。
- 协议文件：请求序列化所需要用到的文件。具体使用方法参见：使用协议文件 。

Redis 压测

最近更新时间：2025-03-24 10:32:02

本文介绍云压测 Redis 测试脚本编写方法，用于支持 Redis 数据库常用操作。

基本用法

使用 Client 概览 API 提供的接口，您可以创建 Redis 数据库的客户端并发送操作请求，进行数据库的基本操作。

数据库连接

建立 Redis 数据库连接可以调用 `new redis.Client(url: string)` 方法。其中，`url` 是目标 redis 的地址。
示例如下：

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

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

export default function () {
    // ...
}
```

> **说明：**
> 建议将上述建立数据库连接的语句，作为全局变量放在主函数外部（如示例），以供同一个 VU 在迭代执行主函数时能够复用连接，避免多次重复创建数据库连接，带来不必要的资源消耗。

数据库操作

连接数据库之后，可以通过 redis 方法进行数据库操作，如下方示例所示。其他方法请参考 JavaScript API 文档 pts/redis 。

```
// redis API
import redis from 'pts/redis';

// Create a redis Client instance.
let client = new redis.Client('redis://:<password>@<host>:6379/0');

export default function () {
```

```
    // set the value of the specified key.
    let resp = client.set('key', 'hello, world', 0);
    console.log(`redis set ${resp}`); // OK
    // obtain the value of the specified key.
    let val = client.get('key');
    console.log(`redis get ${val}`); // hello, world
    // delete an existing key.
    let cnt = client.del('key');
    console.log(`redis del ${cnt}`); // 1

    // insert one or more values at the head of the list.
    let lpushResp = client.lPush('list', 'foo');
    console.log(`redis lpush ${lpushResp}`); // 1
    // remove and obtain the first element of the list.
    let lpopResp = client.lPop('list');
    console.log(`redis lpop ${lpopResp}`); // foo
    // obtain the length of the list.
    let listLen = client.lLen('list');
    console.log(`redis llen ${listLen}`); // 0

    // set the fields and values in the hash table key.
    let hashSetResp = client.hSet('myhash', 'k', 1);
    console.log(`redis hset ${hashSetResp}`); // 1
    // obtain the value stored in the specified field of the hash table.
    let hashGetResp = client.hGet('myhash', 'k');
    console.log(`redis hget ${hashGetResp}`); // 1
    // delete one or more hash table fields.
    let hashDelResp = client.hDel('myhash', 'k');
    console.log(`redis hdel ${hashDelResp}`); // 1

    // add one or more members to the set.
    let setAddResp = client.sAdd('set', 'hello');
    console.log(`redis sadd ${setAddResp}`); // 1
    // randomly remove and return an element in the set.
    let setPopResp = client.sPop('set');
    console.log(`redis spop ${setPopResp}`); // hello
}
```

脚本验证

若要验证脚本执行结果，可在正式压测前，先使用 PTS 调试功能，快速验证结果是否符合预期。更多详情，请参见**调试场景**。

设置检查点

最近更新时间：2024-09-25 11:42:21

概述

您可通过自定义检查点，检查请求的响应结果是否符合业务预期。检查结果会被汇总到检查点指标里，供您在压测报告中查看明细。
此外，您还可以开启检查点与请求的关联日志，以在请求采样侧查看与该请求相关的检查点的信息。

用法

1. 检查点定义

PTS JavaScript API 提供了 `check` 方法来创建检查点。
`check` 方法的入参为：

- name：检查点的名字。
- callback：用于检查的函数，该函数应返回布尔类型。
- response（可选）：传入被检查的请求的响应，用于开启记录检查点日志。
- `check` 方法的返回值为布尔类型，代表本次检查的成功与否。

基本示例如下：

```
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
};
```

常用检查逻辑示例如下：

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

export default function () {
    check("is empty", () => "" === "") // true
```

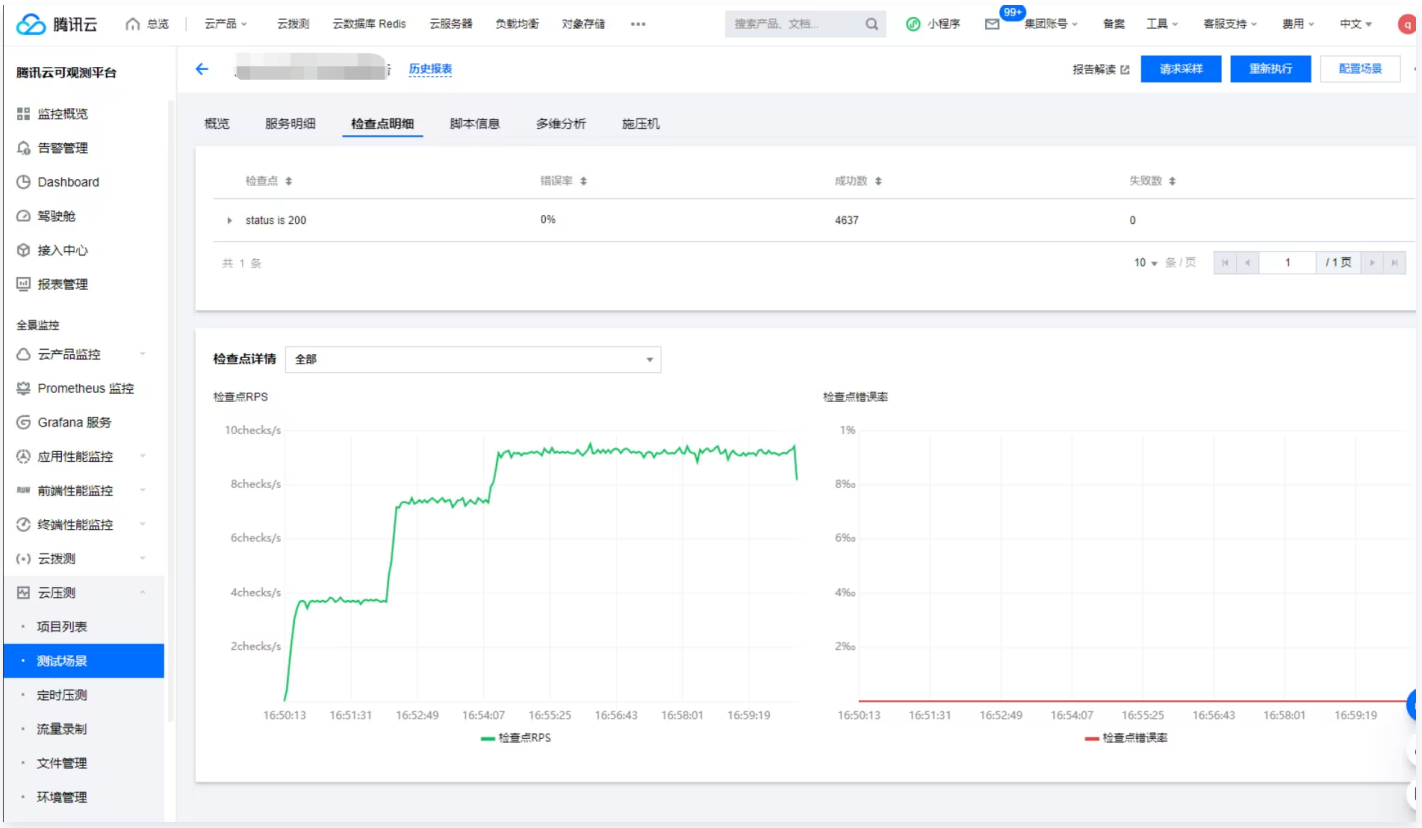
```
    //@ts-ignore
    check("is not empty", () => "str" !== "") // true
    check("equals", () => 1.00 == 1) // true
    check("not equal", () => 1.00 === 1) // true
    check("less than", () => 1 < 2) // true
    check("less or equal", () => 1 <= 1) // true
    check("greater than", () => 2 > 1) // true
    check("greater or equal", () => 2 >= 2) // true
    check("has key", () => ({key:"value"}).hasOwnProperty("key")) //
true
    check("string has value", () => "str".includes("s")) // true
    check("array has value", () => ["a", "b", "c"].includes("a")) //
true
};
```

> **说明**
> 更详细的 API 文档请参见 PTS API check 。

2. 检查结果查看

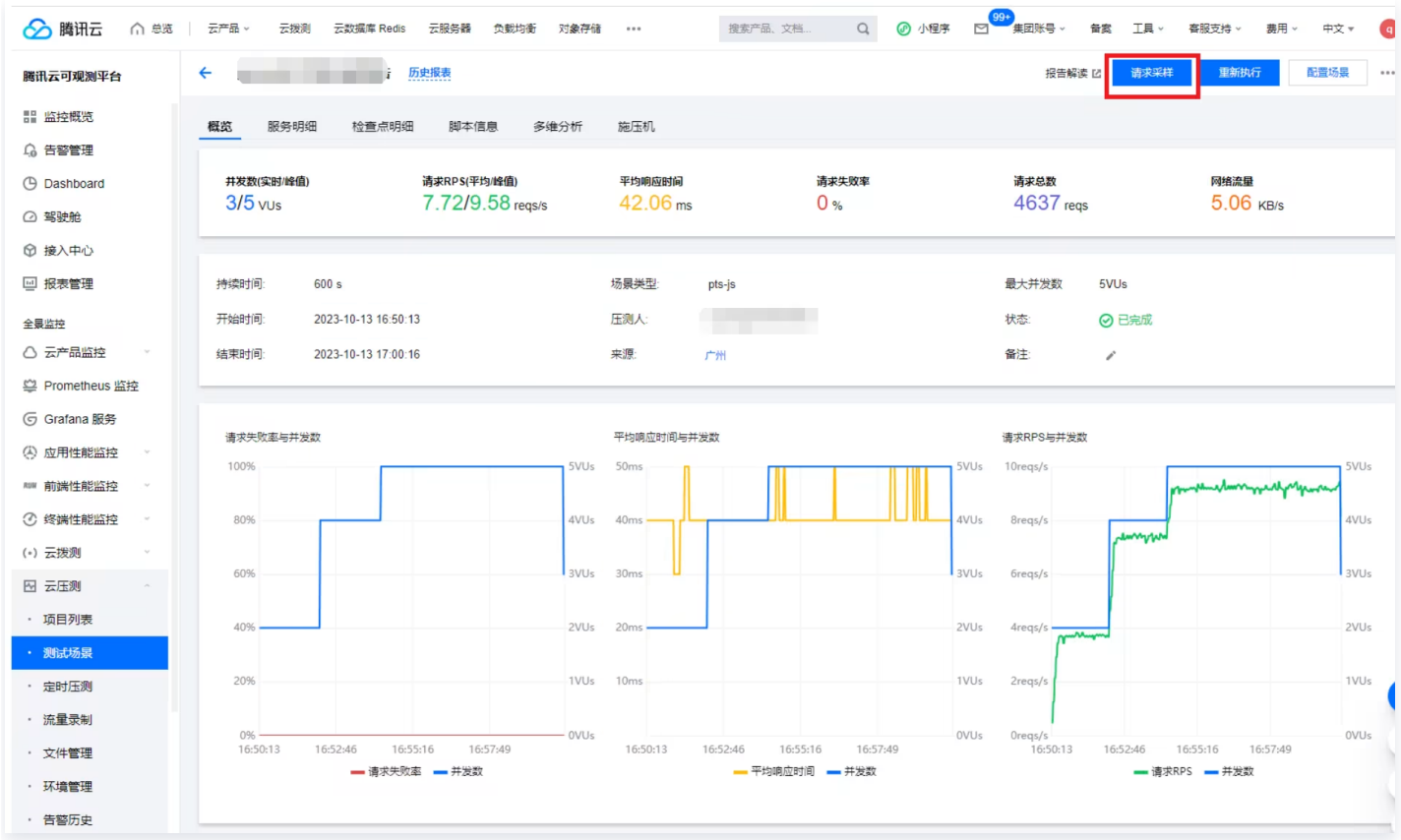
指标明细

登录 腾讯云可观测平台，在左侧导航栏选择**测试场景**，进入压测报告页面，单击**检查点明细**，可以观察到从所有检查结果汇总而来的多维度指标：

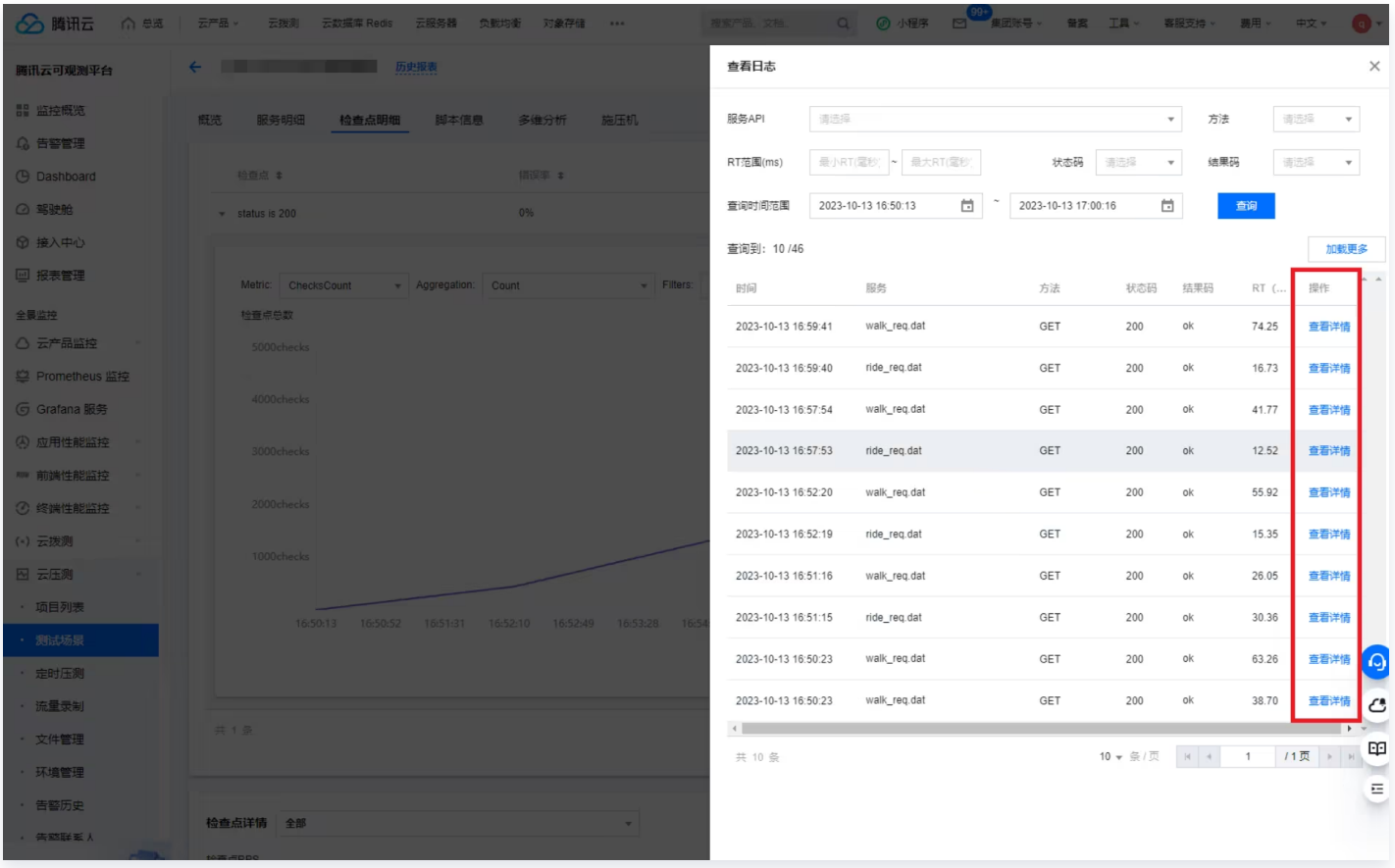


关联请求

调用 `check` 方法时，若您传入了可选的响应参数，则检查结果除了能体现在上述检查点指标里，还会被记录在请求采样日志里，可以进入请求采样页面查看：



单击选择需要查看的采样项：



再单击一条采样请求，进入详情页面，即可看到与该采样请求相关联的检查点内容：

检查点及对应请求的日志打印

检查点在逻辑上和请求是分离的，请求可以对应多个检查点，而检查点也可以检查非请求的内容；同时，请求正常响应（状态码为 200），检查点也可能不通过，这取决于用户配置的检查条件。

```
import http from 'pts/http';
import { check, sleep } from 'pts';
export default function () {
  const resp = http.get('http://mockhttpbin.pts.svc.cluster.local/get',
{
    headers: {
      Connection: 'keep-alive',
      'User-Agent': 'pts-engine',
    },
    query: {
      name1: 'value1',
      name2: 'value2',
    },
  });
  // 请求可以对应多个检查点
  check('status is 200', () => resp.statusCode === 200, resp);
  check('body.args.name1 equals value1', () => resp.json().args.name1
=== 'value1', resp);

  // 请求响应 200，检查点也可能不通过（取决于用户配置的检查条件）
  check('body.args.name1 equals value2', () => resp.json().args.name1
=== 'value2', resp);

  // 检查点可以检查非请求的内容
  let v = 1;
  check("v==1", () => v==1);
  check("v==2", () => v==2);
}
```

不过，在实际使用的过程中，检查点和请求往往组合使用，用于检查请求响应是否符合预期；因此，获取检查点和请求的对应关系非常重要。

在前文“关联请求”中，通过设置 `check` 中的响应参数，可以将检查点结果记录在请求采样的日志里面，满足了部分情况下对两者的关联需求。但在某些情况下，可能对细节有更多定制化的要求，此时可以在检查点的检查条件内将需要的内容打印到日志中，来查看更多的内容细节：

```
import http from 'pts/http';
import { check } from 'pts';
export default function () {
```

```
  const resp = http.get('http://mockhttpbin.pts.svc.cluster.local/get',
{
    query: {
      name1: 'value1',
    },
  });

  // 在检查点的检查条件内打印用户日志
  check('body.args.name1 equals value2', () => {
    if (resp.json().args.name1 === 'value2') {
      return true
    };
    console.log(resp.body);
    console.log(`check not pass, name1 need value2 but
${resp.json().args.name1}`);
    return false;
  });
}
```

三方包引用

最近更新时间：2024-09-11 09:00:51

PTS 脚本压测模式支持原生 JavaScript ES5.1 语法以及绝大部分 ES6 语法。云压测推荐使用 云压测 Javascript 原生库 来实现您的功能。
三方包引用作为实验功能，如果原生库不满足您的需求，您可以**尝试**从远端加载远程 HTTP(S) 模块作为三方包进行引用，云压测不保证您引用的三方包语法一定能被正确解析。
压测引擎启动时，将下载三方脚本及其依赖信息，并执行用户脚本。

> **说明：**
> - 引入的包应当足够的轻量，确保压测脚本能够顺利执行。
> - PTS 不是 NodeJS，所以三方脚本中加载 NodeJS 库不会生效。
> - 不支持 TypeScript 语法。
> - 不支持浏览器中的函数，例如：setTimeout，XMLRequest 等。

常用搜索远程模块的网站

请单击 模块网站 。

常用远程模块

crypto

详细说明请参见 crypto-js 。

```
import crypto from 'https://cdnjs.cloudflare.com/ajax/libs/crypto-
js/4.1.1/crypto-js.js'

export default function () {
    console.log(crypto.MD5('Message')); //
4c2a8fe7eaf24721cc7a9f0175115bd4
    console.log(crypto.SHA1('Message')); //
68f4145fee7dde76afceb910165924ad14cf0d00
    console.log(crypto.SHA256('Message')); //
2f77668a9dfbf8d5848b9eeb4a7145ca94c6ed9236e4a773f6dcafa5132b2f91
    console.log(crypto.SHA512('Message')); //
4fb472dfc43def7a46ad442c58ac532f89e0c8a96f23b672f5fd637652eab158d4d58944
4ef7530a34e6626b40830b4e1ec5364611ae31c599bffa958e8b4c4e
```

```
    console.log(crypto.SHA384('Message')); //
b526d8394134b853bd071719bc99d42b669bc9252baa82dcafabc1f322a3841c57cc0c82
f080fd331b1666112b27a329
    console.log(crypto.RIPEMD160('Message')); //
85eab2fe4383a869da13d51f4b91506924b1f821

    console.log(crypto.HmacMD5('Message', 'Secret Passphrase')); //
5e03d0c1b42ef0b7e61fb333f3993949
    console.log(crypto.HmacSHA1('Message', 'Secret Passphrase')); //
e90f713295ea4cc06c92c9248696ffafc5d01faf
    console.log(crypto.HmacSHA256('Message', 'Secret Passphrase')); //
32c647602ab4c4c7543e9c50ae25e567c3354e1532b11649ce308e6e2568d205
    console.log(crypto.HmacSHA512('Message', 'Secret Passphrase')); //
c03f82cd6f9d03920d95caeedfa722d4e42325a18b049942ee5560787ad2aa394be6b958
49c563ecdd37495726cd6236529a721b563b9778dd6119939bcab7e1
    console.log(crypto.HmacSHA384('Message', 'Secret Passphrase')); //
84b318cc0232a370c1f8b8746afcb575fc2debc680122c7422fd425638896d0dcf9e905b
8cd9c1d7aed8d5439a2a2328
    console.log(crypto.HmacRIPEMD160('Message', 'Secret Passphrase'));
// d1b4088aba7f4897444c1423c0b1f056605473ab

    let words = crypto.enc.Base64.parse('SGVsbG8sIFdvcmxkIQ==');
    console.log(words); // 48656c6c6f2c20576f726c6421

    let base64 = crypto.enc.Base64.stringify(words);
    console.log(base64); // SGVsbG8sIFdvcmxkIQ==

    words = crypto.enc.Latin1.parse('Hello, World!');
    console.log(words); // 48656c6c6f2c20576f726c6421

    let latin1 = crypto.enc.Latin1.stringify(words);
    console.log(latin1); // Hello, World!

    words = crypto.enc.Hex.parse('48656c6c6f2c20576f726c6421');
    console.log(words); // 48656c6c6f2c20576f726c6421

    let hex = crypto.enc.Hex.stringify(words);
    console.log(hex); // 48656c6c6f2c20576f726c6421

    words = crypto.enc.Utf8.parse('?');
```

```
    console.log(words); // f094ada2

    let utf8 = crypto.enc.Utf8.stringify(words);
    console.log(utf8); // ?

    words = crypto.enc.Utf16.parse('Hello, World!');
    console.log(words); //
004800650006c006c006f002c00200057006f0072006c00640021

    let utf16 = crypto.enc.Utf16.stringify(words);
    console.log(utf16); // Hello, World!

    words = crypto.enc.Utf16LE.parse('Hello, World!');
    console.log(words); //
4800650006c006c006f002c00200057006f0072006c0064002100

    utf16 = crypto.enc.Utf16LE.stringify(words);
    console.log(utf16); // Hello, World!
}
```

aes

详细说明请参见 相关文档 。

```
import aesjs from 'https://cdnjs.cloudflare.com/ajax/libs/aes-js/4.0.0-
beta.2/index.js'

export default function () {
    // An example 128-bit key (16 bytes * 8 bits/byte = 128 bits)
    var key = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16];

    // Convert text to bytes
    var text = 'Text may be any length you wish, no padding is
required.';
    var textBytes = aesjs.utils.utf8.toBytes(text);

    // The counter is optional, and if omitted will begin at 1
    var aesCtr = new aesjs.ModeOfOperation.ctr(key, new
aesjs.Counter(5));
    var encryptedBytes = aesCtr.encrypt(textBytes);
```

```
    // To print or store the binary data, you may convert it to hex
    var encryptedHex = aesjs.utils.hex.fromBytes(encryptedBytes);
    console.log(encryptedHex);
    // "a338eda3874ed884b6199150d36f49988c90f5c47fe7792b0cf8c7f77eeffd87
    //  ea145b73e82aefcf2076f881c88879e4e25b1d7b24ba2788"

    // When ready to decrypt the hex string, convert it back to bytes
    var encryptedBytes = aesjs.utils.hex.toBytes(encryptedHex);

    // The counter mode of operation maintains internal state, so to
    // decrypt a new instance must be instantiated.
    var aesCtr = new aesjs.ModeOfOperation.ctr(key, new
aesjs.Counter(5));
    var decryptedBytes = aesCtr.decrypt(encryptedBytes);

    // Convert our bytes back into text
    var decryptedText = aesjs.utils.utf8.fromBytes(decryptedBytes);
    console.log(decryptedText);
    // "Text may be any length you wish, no padding is required."
}
```

设置全局 Options

最近更新时间：2024-06-07 17:20:32

若要定制压测引擎在执行压测任务时的行为（传入自定义参数值、覆盖默认配置），您可设置全局 Options，来控制诸如 TLS/HTTP/WebSocket 通信的配置参数、预处理/后处理的超时时间等。
要设置全局 Options，您可在脚本最外层定义一个 option 对象（`export const option = {...}`），再根据您的需求，在 option 对象中定义 `tlsConfig`、`http` 等字段。

> **说明：**
> 参数详情可参见 PTS JavaScript API 文档：Option。

HTTP 全局配置

通过 `option` 里的 `http` 字段，您可配置压测引擎作为 HTTP 客户端的相关参数，这些参数对本次压测任务的所有 HTTP 请求全局生效。
常用参数如下：

- `headers`：设置请求头。
- `basicAuth`：使用 HTTP basic auth 认证时，通过该参数传入用户名和密码。
- `disableKeepAlives`：若要禁用长连接，可将该字段设置为 true。
- `discardResponseBody`：若要丢弃服务端返回的响应包体，可将该字段设置为 true。
- `http2`：若要启用 HTTP2 协议，可将该字段设置为 true。
- `maxIdleConns`：单个 VU 的最大连接数，默认值为 100个。
- `maxIdleConnsPerHost`：单个 VU 针对单个 host（地址+端口）的最大连接数，默认值为 2。
- `maxRedirects`：重定向跳转的最大次数，默认值为 10次。
- `timeout`：请求超时时间，单位为毫秒，默认值为 10000（10 秒）。

脚本示例：

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

export const option = {
    http: {
        maxRedirects: 10,
        maxIdleConns: 100,
        headers: {
            'key': 'value'
        }
```

```
    }
}
export default function () {
  // get request with headers and parameters
  const resp1 = http.get('http://httpbin.org/get', {
    headers: {
      Connection: 'keep-alive',
      'User-Agent': 'pts-engine',
    },
    query: {
      name1: 'value1',
      name2: 'value2',
    },
  });

  console.log(resp1.json().args.name1); // 'value1'
  check('status is 200', () => resp1.statusCode === 200);
  check('body.args.name1 equals value1', () => resp1.json().args.name1
=== 'value1');
  check('headers.key equals value', () => resp1.json().headers.key ===
'value')
}
```

WebSocket 全局配置

通过 `option` 里的 `ws` 字段，您可配置压测引擎作为 WebSocket 客户端的相关参数，这些参数对本次压测任务的所有 WebSocket 请求全局生效。

常用参数如下：

- `handshakeTimeout`：握手超时时间，单位为毫秒，默认值为 30 秒。
- `readTimeout`：读消息超时时间，单位为毫秒，默认不限制。
- `writeControlTimeout`：写控制指令超时时间，单位为毫秒，默认为 10 秒。
- `writeTimeout`：写消息超时时间，单位为毫秒，默认不限制。

示例：

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

```
}
```

TLS 全局配置

压测引擎作为客户端，在建立 TLS（Transport Layer Security/传输层安全协议）连接时，支持以下配置选项：

- `insecureSkipVerify`：是否验证服务器的证书链和主机名。若设置为 true 则不验证（默认为 false）。
- `rootCAs`：在验证服务器证书时，使用的一组根证书颁发机构。 若为空，则默认使用主机的根 CA 集。
- `certificates`：在双向 TLS 认证中，客户端提供的供服务端验证的证书列表（证书文件可通过场景的 `文件管理 > 请求文件` 上传）。

示例：

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

预处理与后处理配置

- setupTimeoutSeconds：预处理（setup）步骤的超时时间。默认为 60 秒。 示例：

```
export const option = { setupTimeoutSeconds: 30 }
```

- `teardownTimeoutSeconds`：后处理（teardown）步骤的超时时间。默认为 60 秒。

```
export const option = {
    teardownTimeoutSeconds: 30
}
```

> **说明：**
> 关于脚本的预处理/后处理的详情，参见： 脚本概述 。

运行时元数据

最近更新时间：2024-10-16 16:33:31

在脚本模式中，我们可以通过 metadata 模块，获取压测运行时的元数据。

元数据字段说明

元数据字段	元数据描述
userID	执行压测用户的 uin
appID	执行压测用户的 appID
scenarioID	压测任务所属场景 ID
jobID	压测任务 ID
region	引擎所属地域。各个地域 region 值如下： • 广州：ap-guangzhou • 上海：ap-shanghai • 北京：ap-beijing • 南京：ap-nanjing • 成都：ap-chengdu

示例

通过调用 metadata() 获取 Metadata 对象，通过 Metadata 对象属性获取元数据。基本示例如下：

```
// get metadata
import { metadata } from 'pts'

var meta = metadata()

export default function () {
  console.log(meta.userID) // 123456
  console.log(meta.appID) // 123456
  console.log(meta.scenarioID) // scenario-123456
  console.log(meta.jobID) // job-123456
  console.log(meta.region) // ap-guangzhou
}
```

说明：
更详细的 API 文档请查看 PTS API Metadata 。

JMeter 模式压测
JMeter 模式概述

最近更新时间：2024-11-08 15:33:22

PTS 支持 JMeter 模式压测，为您提供与原生 JMeter 压测一致的使用体验。PTS 目前仅支持 HTTP、TCP/UDP、Tars、WebSocket 压测请求的报表查看。

基本用法

创建 JMeter 压测场景

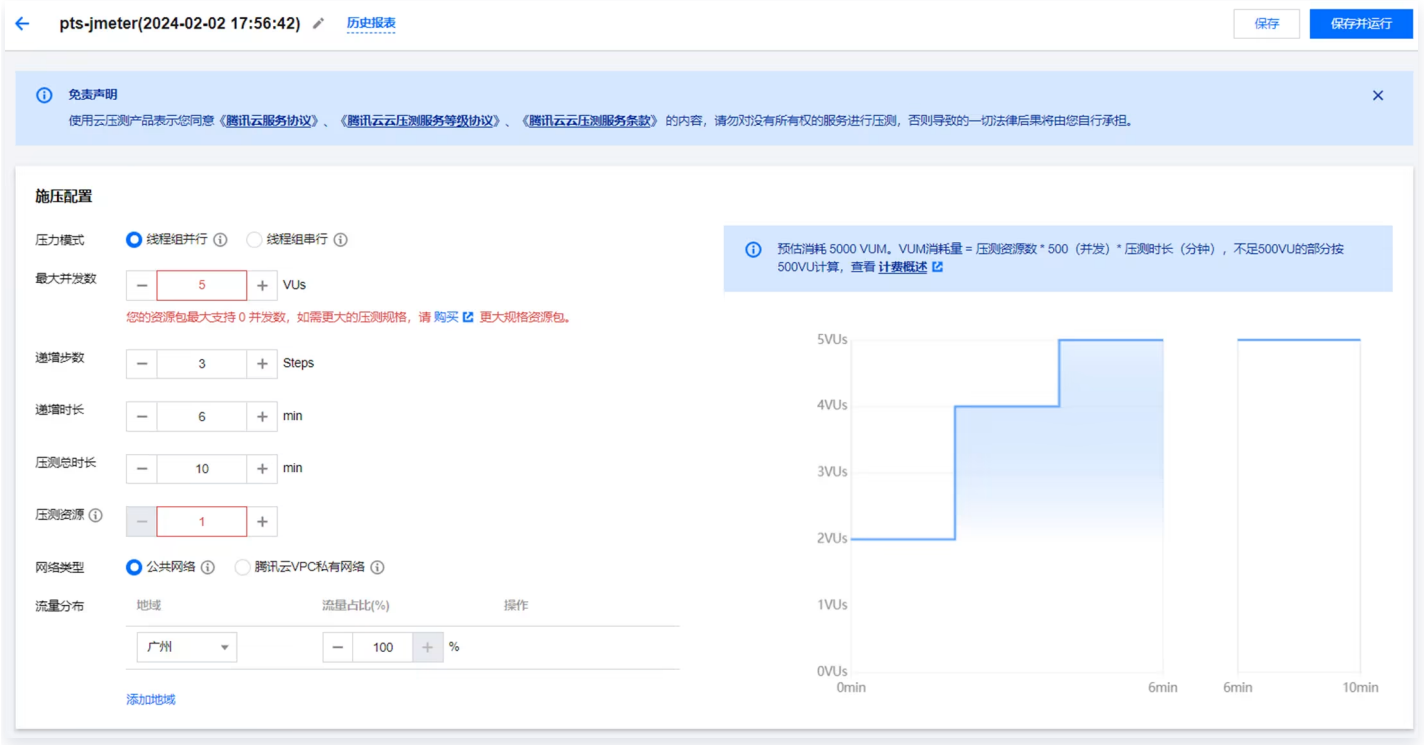
1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**云压测** > **测试场景**。
3. 在创建测试场景页面选择 **JMeter** 压测类型，并单击**开始**，创建压测场景。

上传 jmx 脚本等文件

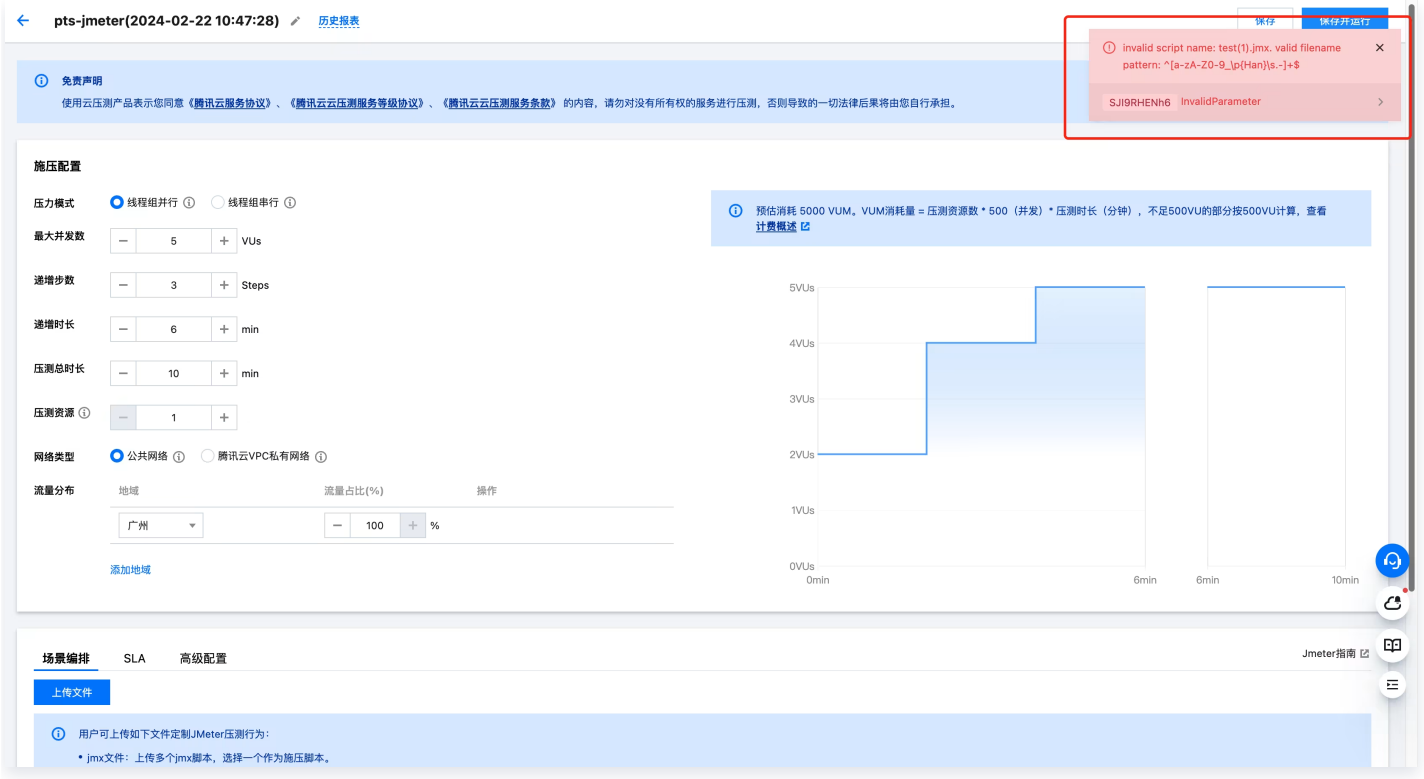
- 必选：上传 jmx 压测脚本。
- 可选：支持上传 csv、jar 等文件，以使用 JMeter 扩展功能。

运行压测脚本

单击右上角**保存并运行**，开始执行压测任务、生成实时报告。



在**保存**的过程中，如果出现如下弹框，用户需要检查上传文件的名称，去掉文件名中的非法特殊字符。



使用限制

从安全角度考虑，PTS 限制用户通过 BeanShell exec 执行外部命令。beanshell exec 命令会被拦截并在引擎中输出错误提示。

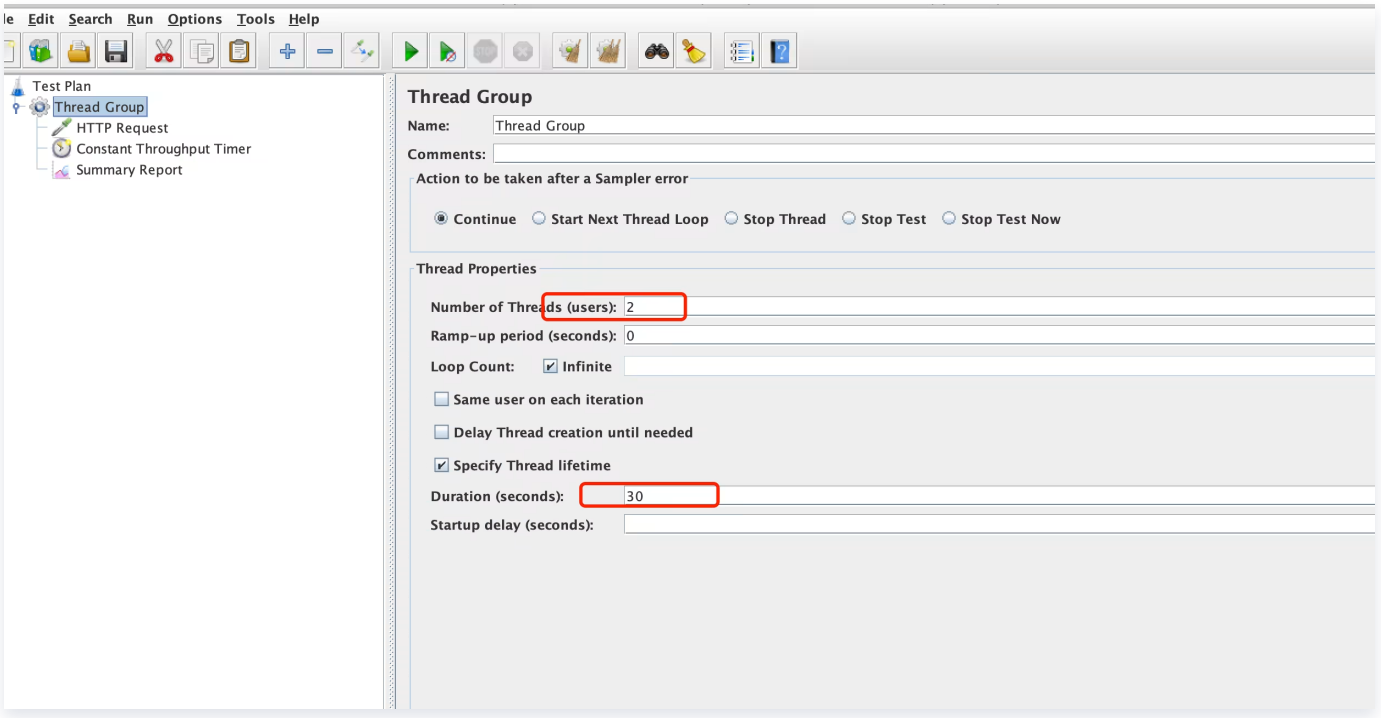
JMeter 配置 RPS 限制

最近更新时间：2024-08-16 15:02:21

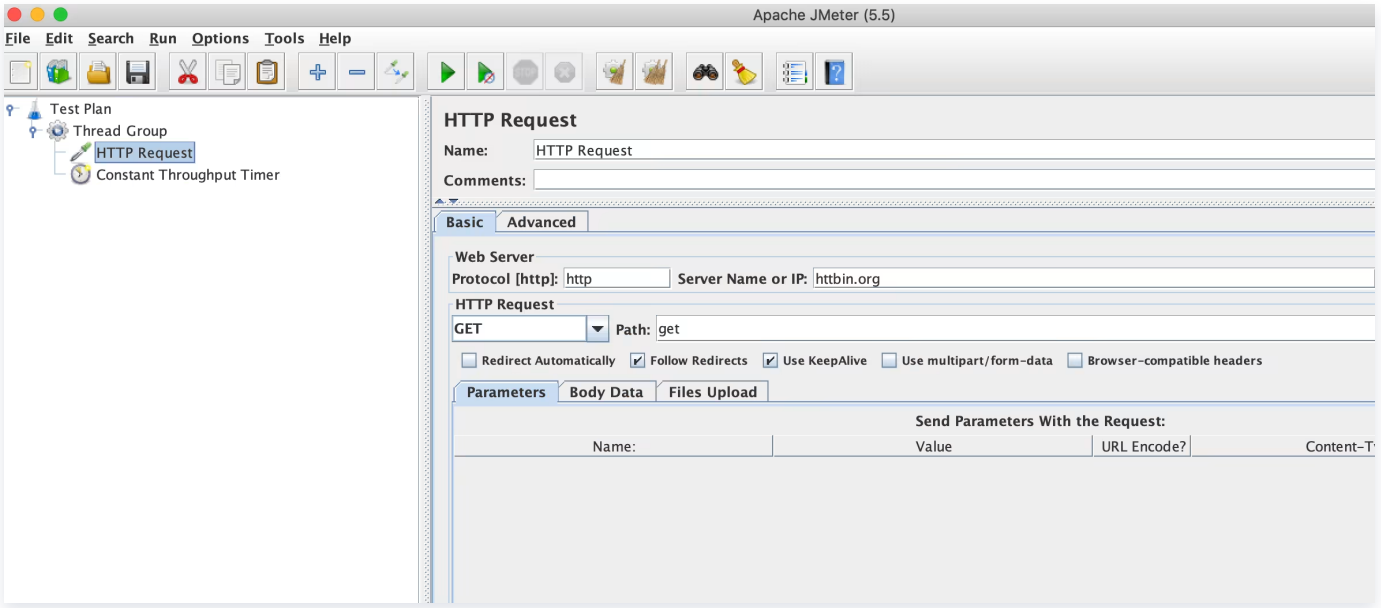
RPS (Request Per Second)一般用来衡量服务端的吞吐量，相比于并发模式，更适合用来摸底服务端的性能。我们可以通过使用 JMeter 原生的 Constant Throughput Timer 来限制每个线程的RPS。

配置 RPS 限制

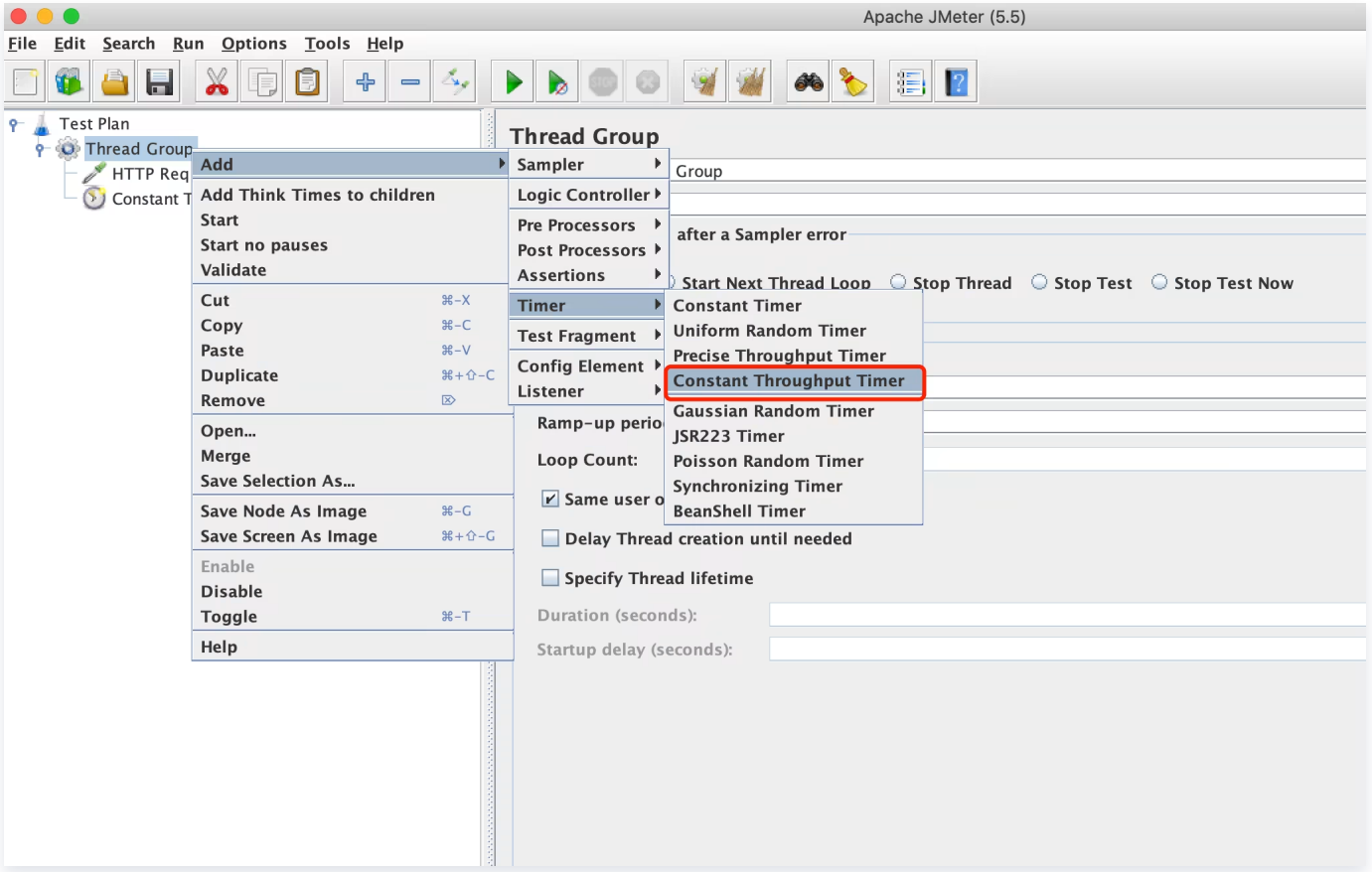
RPS 取决于压测的并发数以及服务的响应时间，并发数过高，可能压力过大压垮后端服务，并发数过低，可能压不到指定的 RPS。
为了避免压力过大压垮后端服务以及摸底后端服务性能上限，可以通过设置 Constant Throughput timer 来限制线程的 RPS 上限。
以下示例中：我们通过 JMeter 添加一个线程组，包含2个线程，每个线程 RPS 上限为1。
运行30s，查看最终总的 RPS 是否为 1reqs/s/thread × 2 thread = 2 reqs/s
1. 设置线程组并发数为2，运行30s。



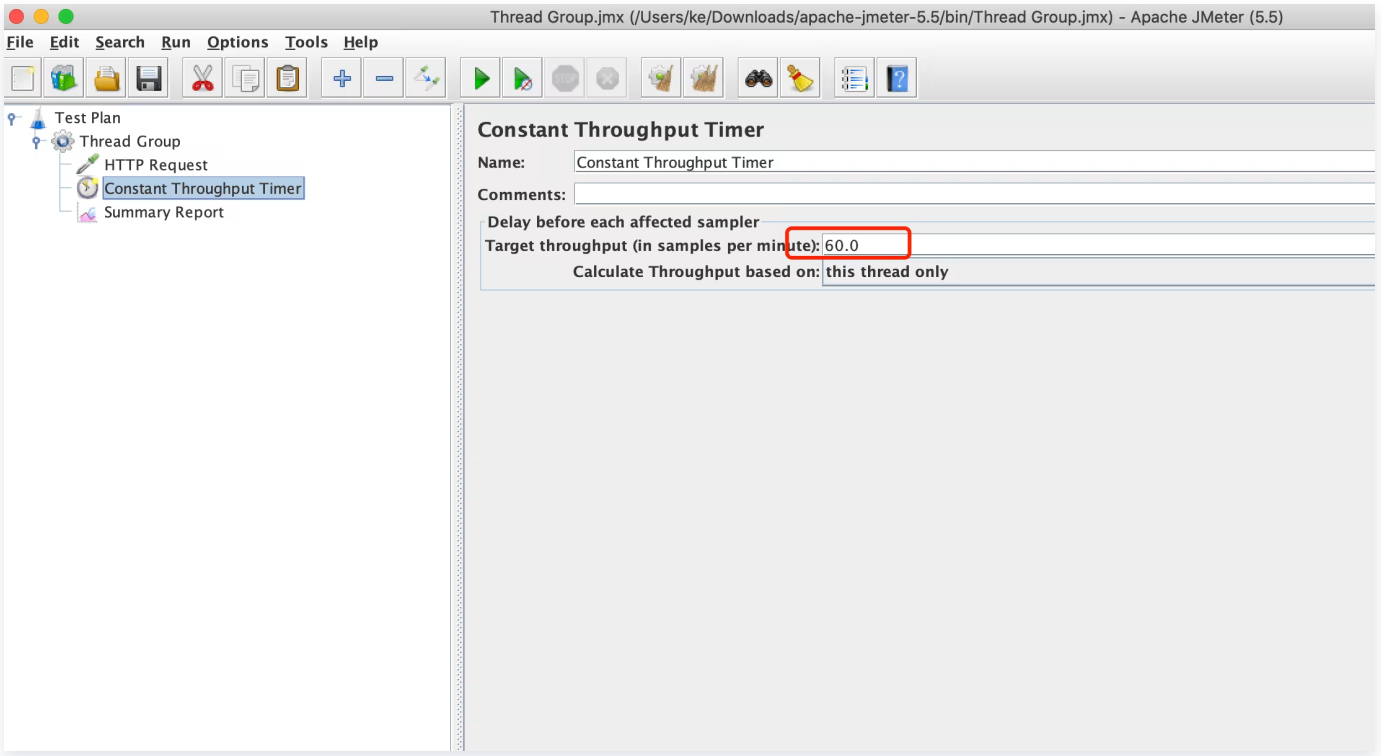
2. 添加HTTP sampler，模拟发送 HTTP 请求。



3. 右键单击线程组，选择 Timer > Constant Throughput Timer。



4. 配置每个线程组 RPS上限为1 req/s， 即: 60req/min。



> ⚠ **注意:**
>
> 1. Target throughput **默认单位是分钟**，如果想设置单线程 RPS 是 1reqs/s， 则 JMeter 表单中必须填写60，即60req/min。
> 2. Calculate Throughput based on 必须设置为 **this thread only**，即设置单个线程的 RPS。当用户配置的并发较大时，PTS 将压测任务切分后分发到多个 JMeter 引擎执行。其他选项仅对单个引擎生效，无法全局生效。
> 3. 如果要制定全局 RPS 限制，可使用全局 RPS 除以并发数，得到单个 RPS 的限制。

5. 查看结果报告，确认是否接近2req/s。

JMeter 使用 CSV 参数文件

最近更新时间：2024-06-27 14:06:51

在使用 JMeter 做压力测试的时候，我们可以使用 JMeter CSV Data Set Config 元件来实现参数化，每次请求发送不同的测试数据，模拟更真实的用户场景。
用户可通过 CSV 文件提前构造压测数据。JMeter 线程组循环时每次从 CSV 文件中读取一行作为测试数据，每次请求使用不同的测试用例。

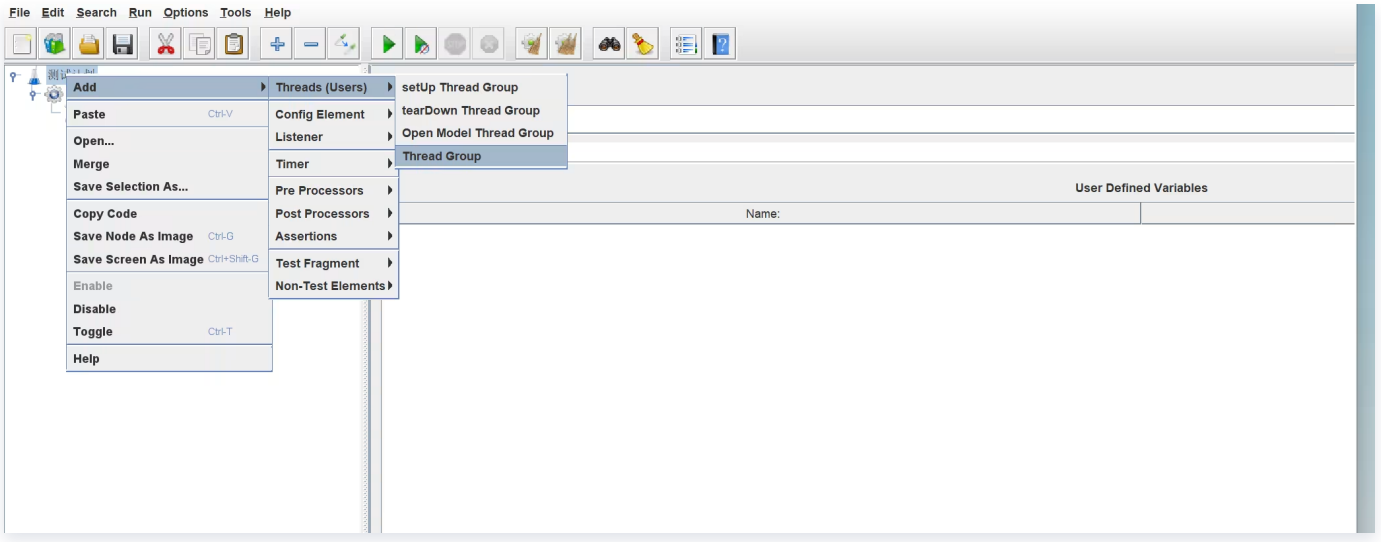
使用 JMeter 配置参数文件

假设需要模拟给不同用户发送邮件的场景，我们提前准备好一份 CSV 格式文件，第一行作为表头，表示参数名。其他行作为测试数据。
在这个文件中我们定义了2个参数：name、email。我们可以在 JMeter 请求中通过${name}, ${email} 引用参数。

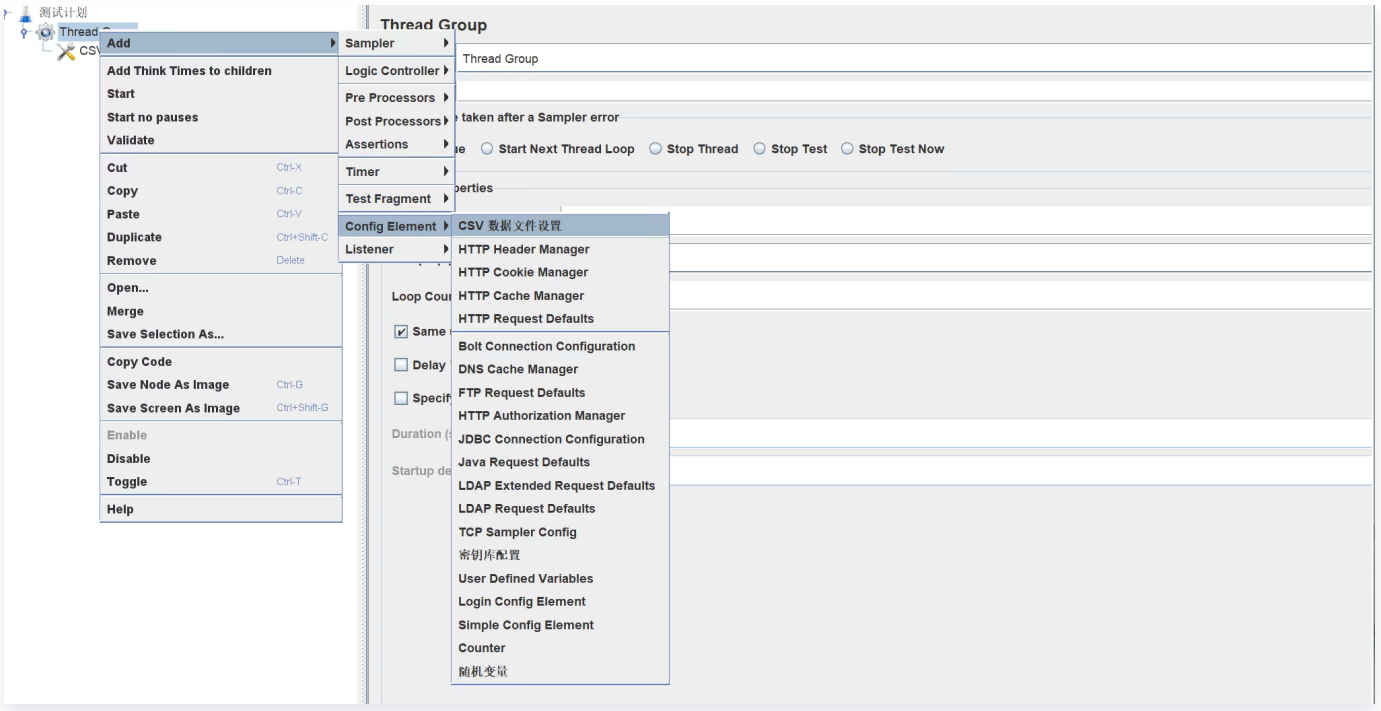
```
name, email
lyli,lyli@test.com
lucky,lucky@test.com
lucas,lucas@test.com
```

如果 CSV 文件中首行不包含参数名，则需要在 CSV Data Set Config 中额外设置参数名。

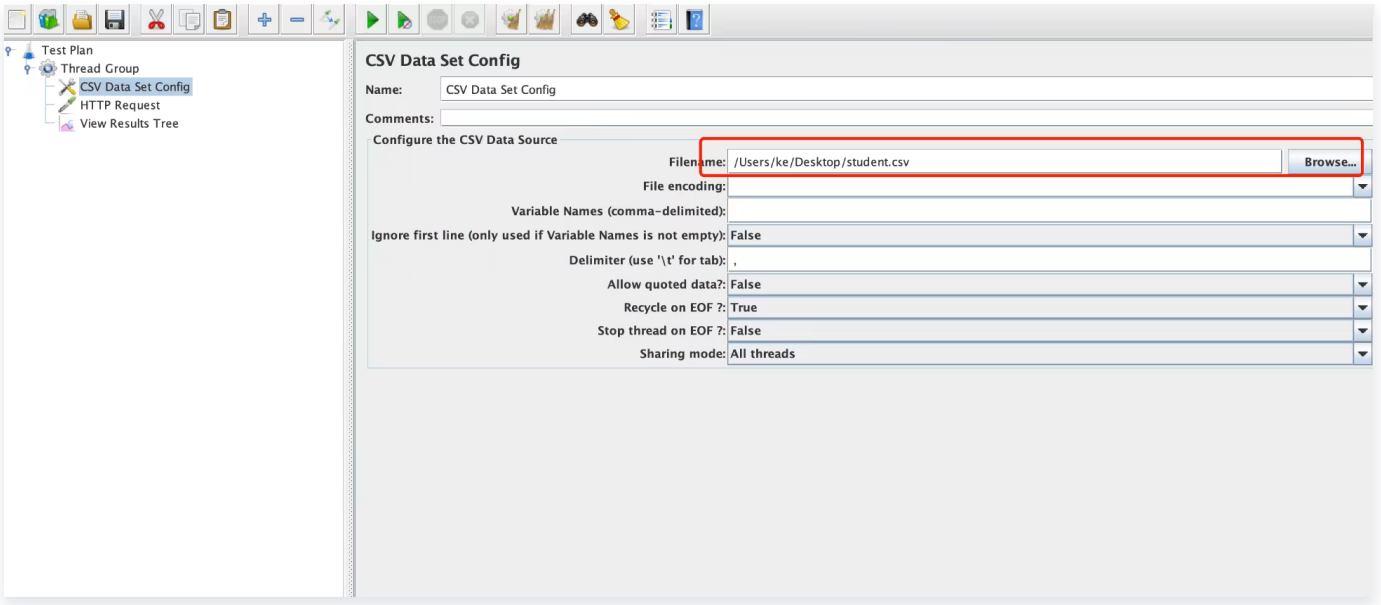
1. 右键单击**测试计划**，选择 **Add > Threads(Users) > Thread Group**。



2. 右键单击**线程组**，选择 **Add > Config Element > CSV 数据文件设置**。

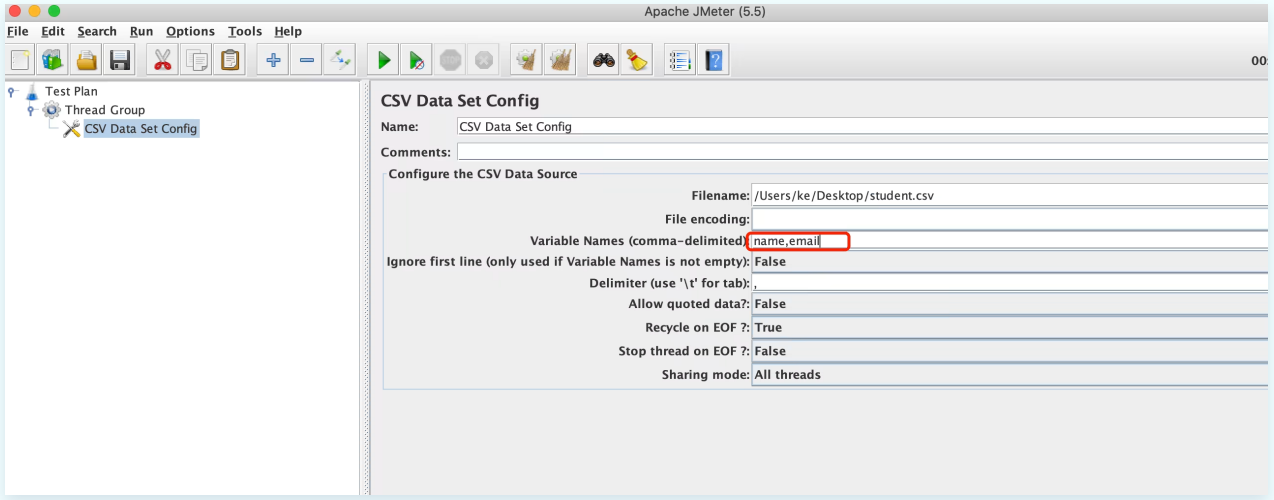


3. 配置 CSV Data Set Config. 在我们的案例中，导入 CSV 文件，其他的保留默认配置即可。此时默认以首行(name, email)作为参数名，我们可以在请求中引用 name、email 变量。



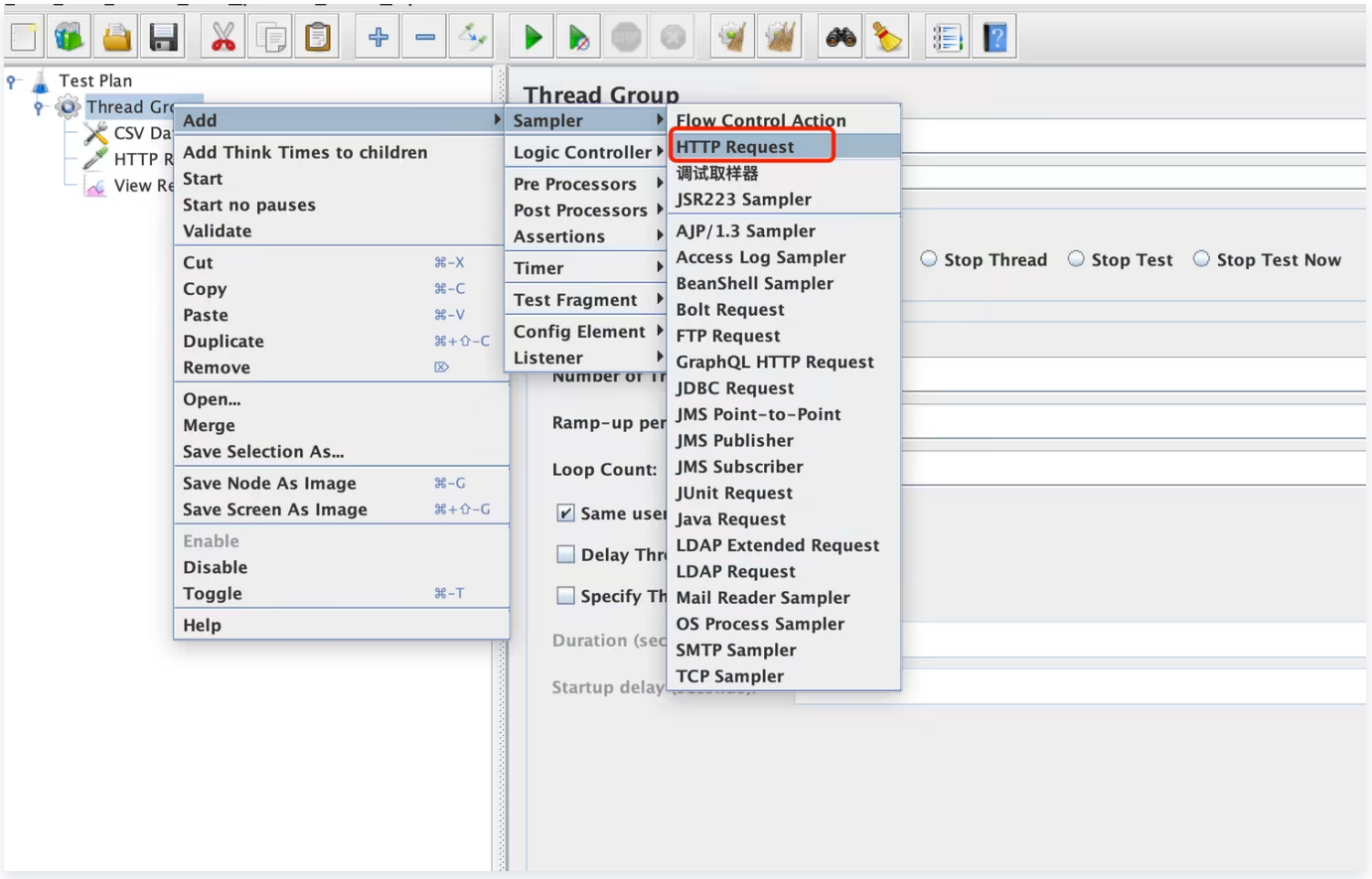
注意：

如果您的 CSV 文件首行不包含参数名，则需要设置参数名，配置如下：

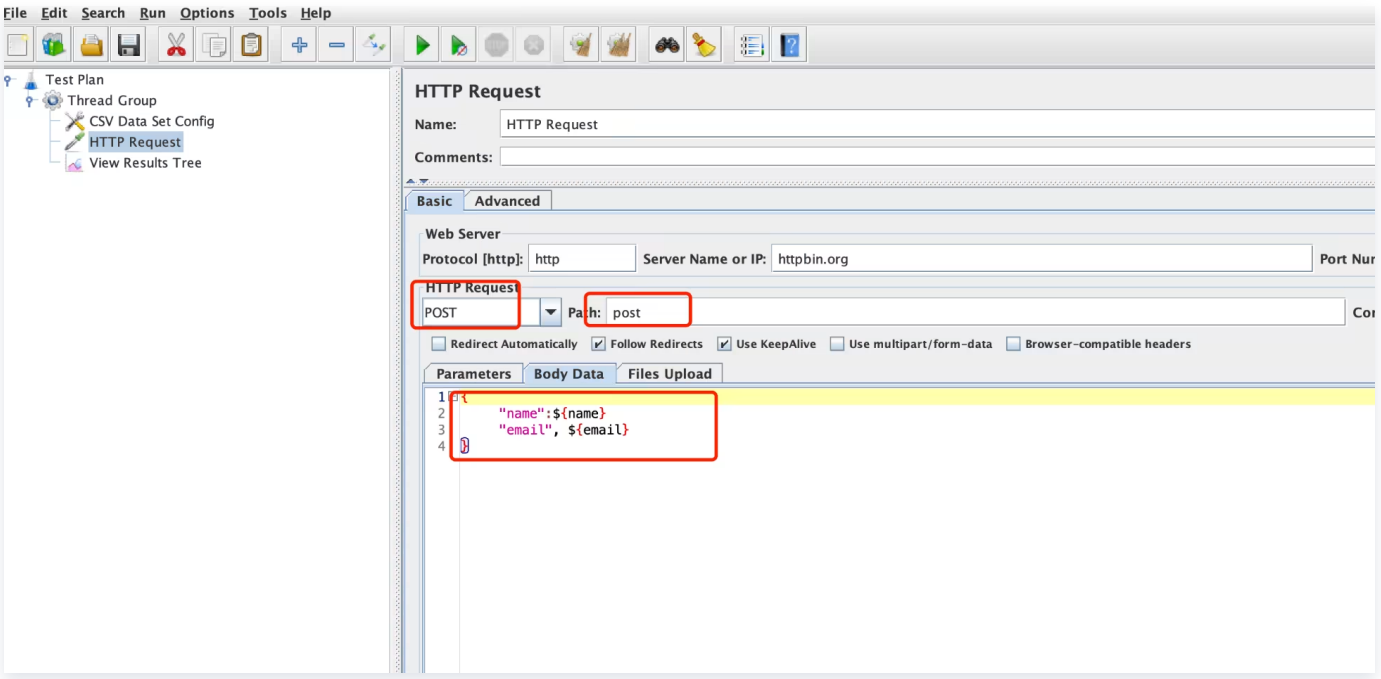


在请求中引用 CSV Data Set Config 设置的参数

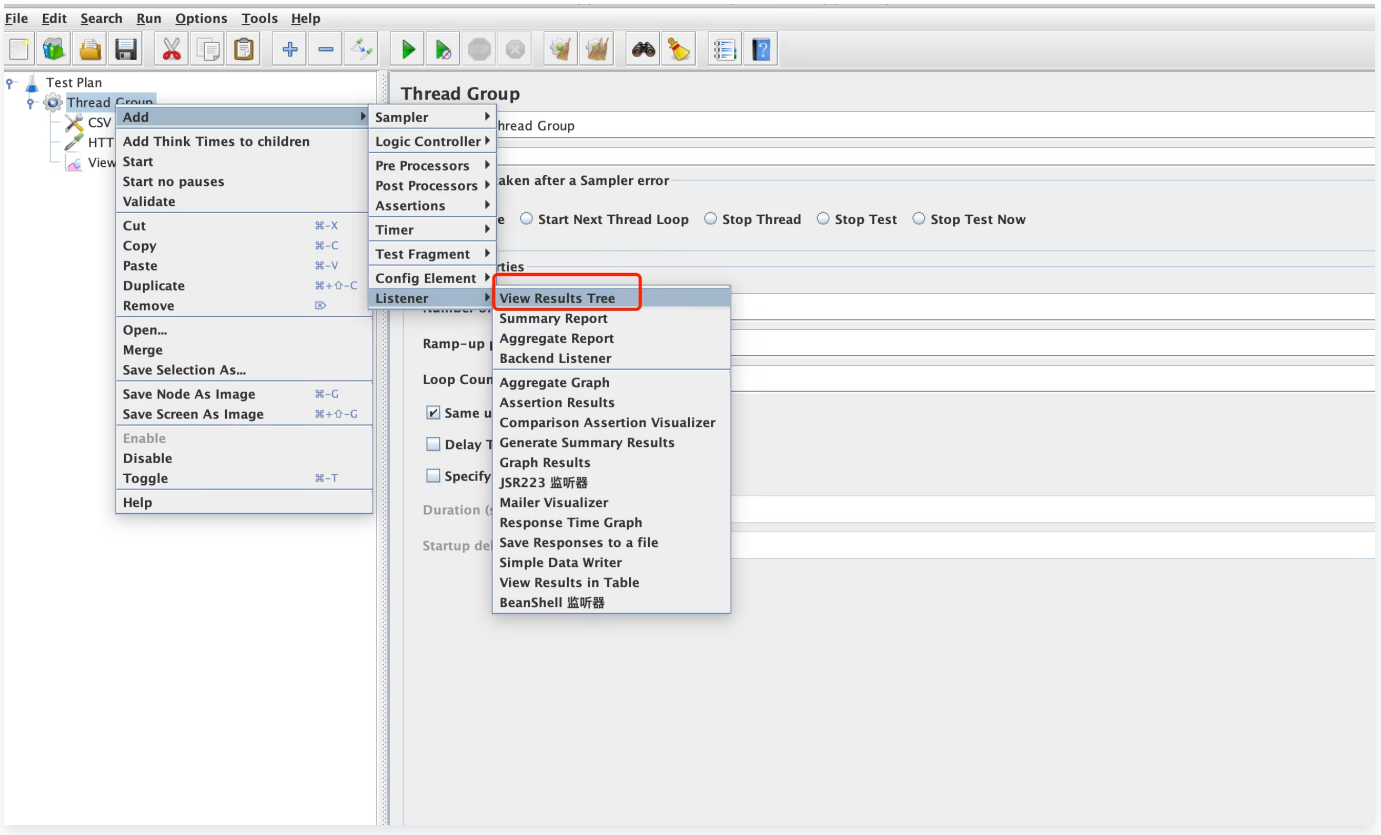
1. 右键单击**线程组**，选择 **Sampler > HTTP Request**。



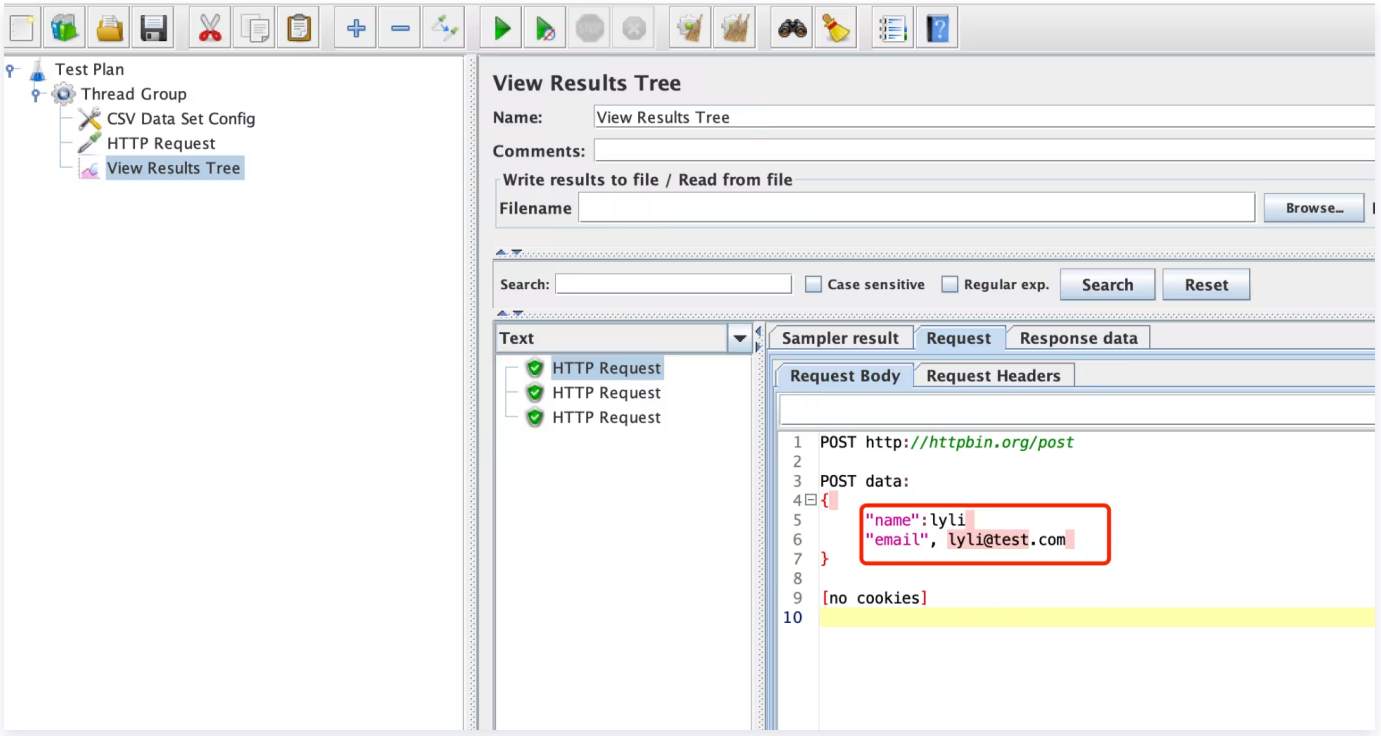
2. 配置 HTTP Sampler，发送 POST 请求。在 body 中使用${name}, ${email}引用 CSV 中配置的参数。



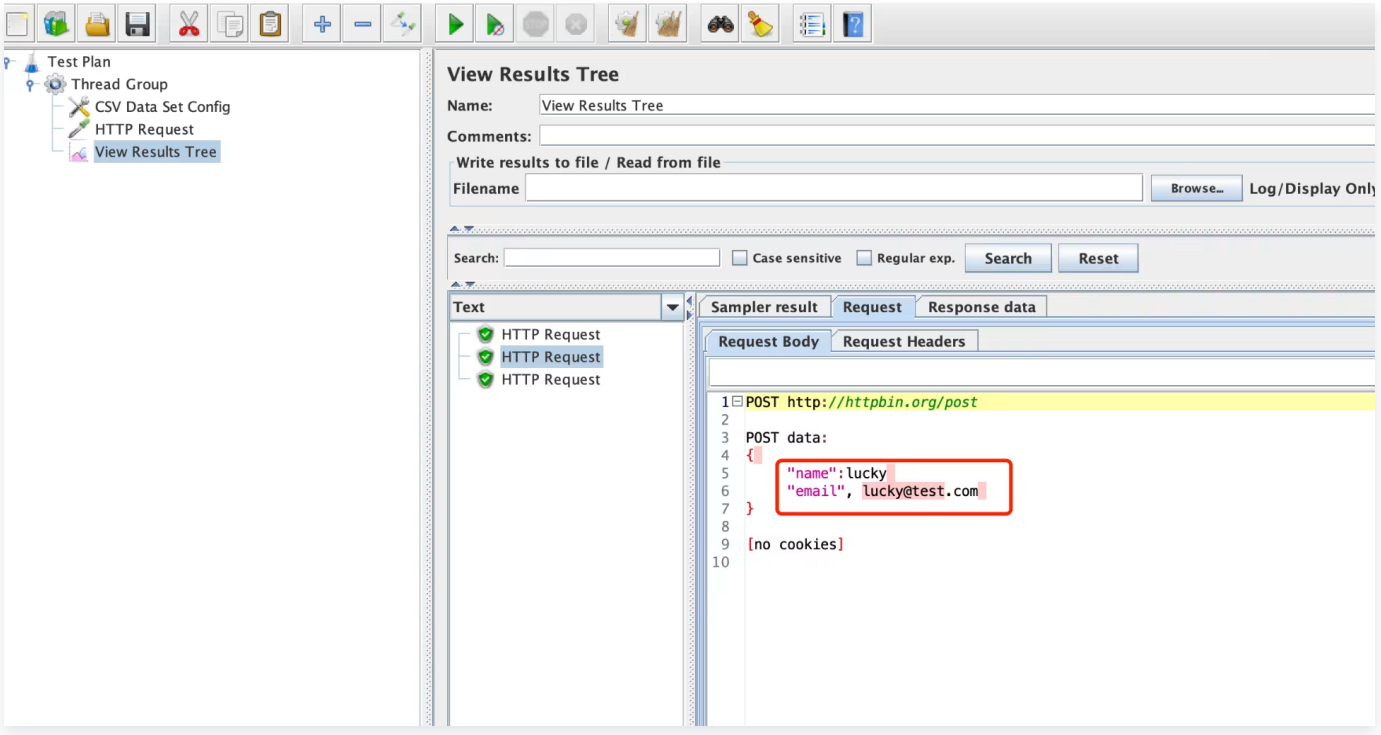
3. 右键单击**线程组**，选择 **Listener > View Result Tree**。通过 View Result Tree 我们可以查看每次请求发送和返回的数据，来确认以上配置的 CSV Data Set Config 是否生效。



4. 运行线程组，查看 View Result Tree 中请求发送的数据。我们可以看到一共发送了两个请求，每个请求 request Body 都不一样。
   - 第一个请求：



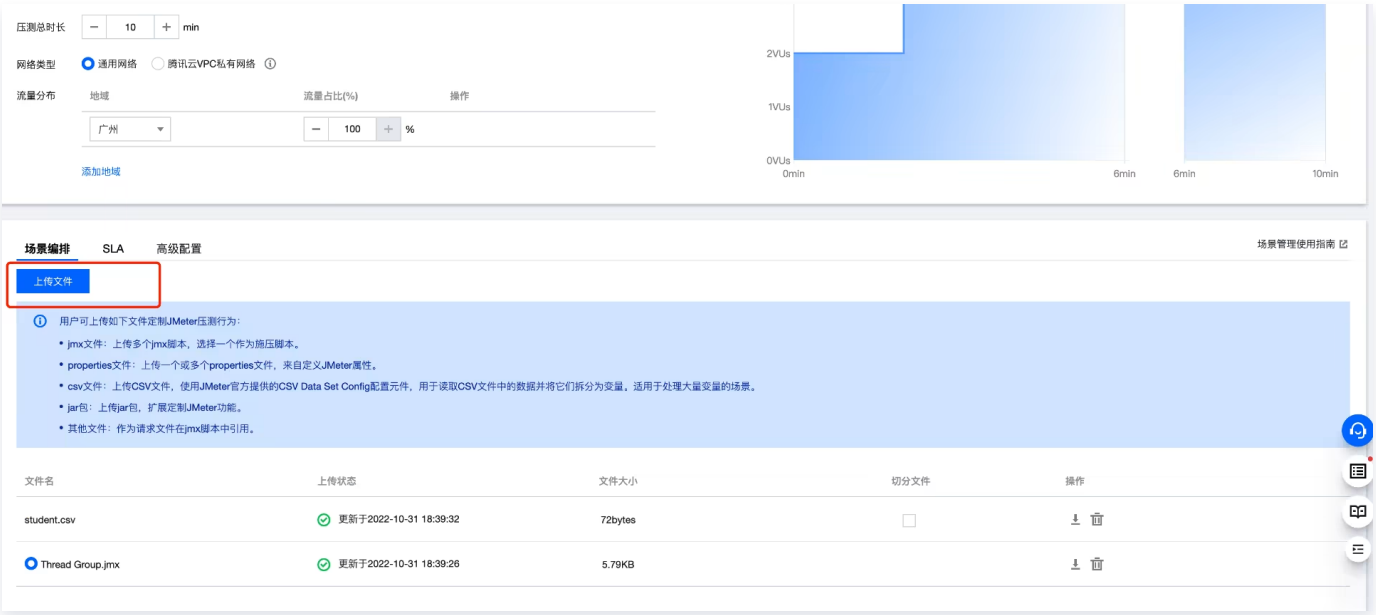
- 第二个请求：



在 PTS 中使用 CSV 参数文件

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**云压测 > 测试场景**，再点击**新建场景**。
3. 在创建测试场景页面选择“JMeter”压测类型，并单击**开始**，创建压测场景。

4. 上传刚配置的 Jmx 文件和 CSV 文件。



> **说明：**
> PTS 在运行压测时，会自动识别 Jmx 中配置的 CSV 文件路径，并替换成用户上传的同名文件。例如：假设 Jmx 中配置的 CSV 文件路径为 `/Users/ke/Desktop/student.csv`，在 PTS 中只需上传 `student.csv` 文件即可。PTS 在运行时会自动帮您识别同名文件，无需您更改 Jmx 中引用的文件路径。

5. CSV 文件切分：PTS 会根据用户配置的并发数，自动启动多个压测引擎，以集群化模式运行 JMeter 压测脚本。如果您希望每个引擎执行单独的测试用例，则需要用到 CSV 文件切分功能。

   5.1 假设用户上传的 CSV 文件为：

```
name,email
lyli,lyli@test.com
lucky,lucky@test.com
lucas,lucas@test.com
```

5.2 勾选切分文件选项

如果用户配置的并发数较大，需要调度2个引擎执行该压测任务，则 CSV 测试数据会在2个引擎间均分。

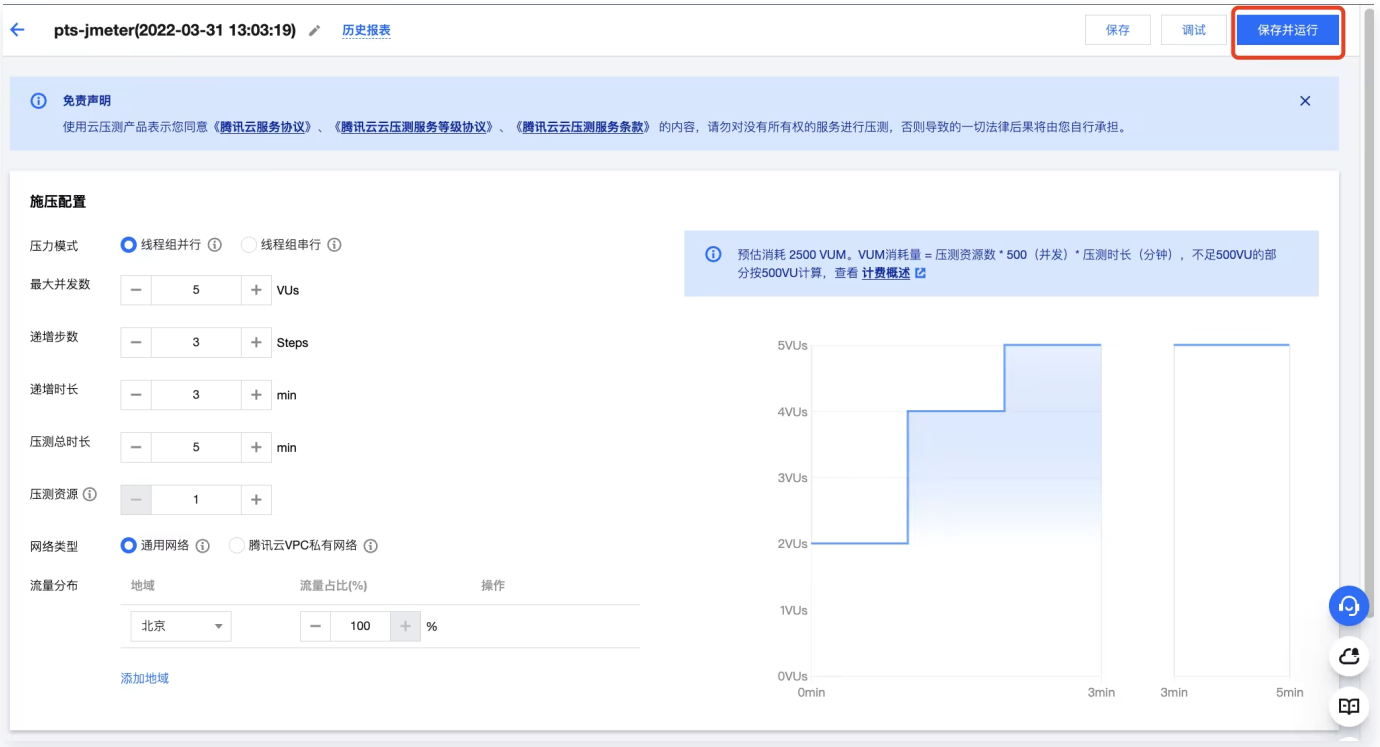
引擎1获取的 CSV 文件为：

```
name,email
lyli,lyli@test.com
lucky,lucky@test.com
```

引擎2获取的 CSV 文件为：

```
name,email
lucas,lucas@test.com
```

6. 单击**保存并运行**，即可开始压测。



JMeter 多线程组

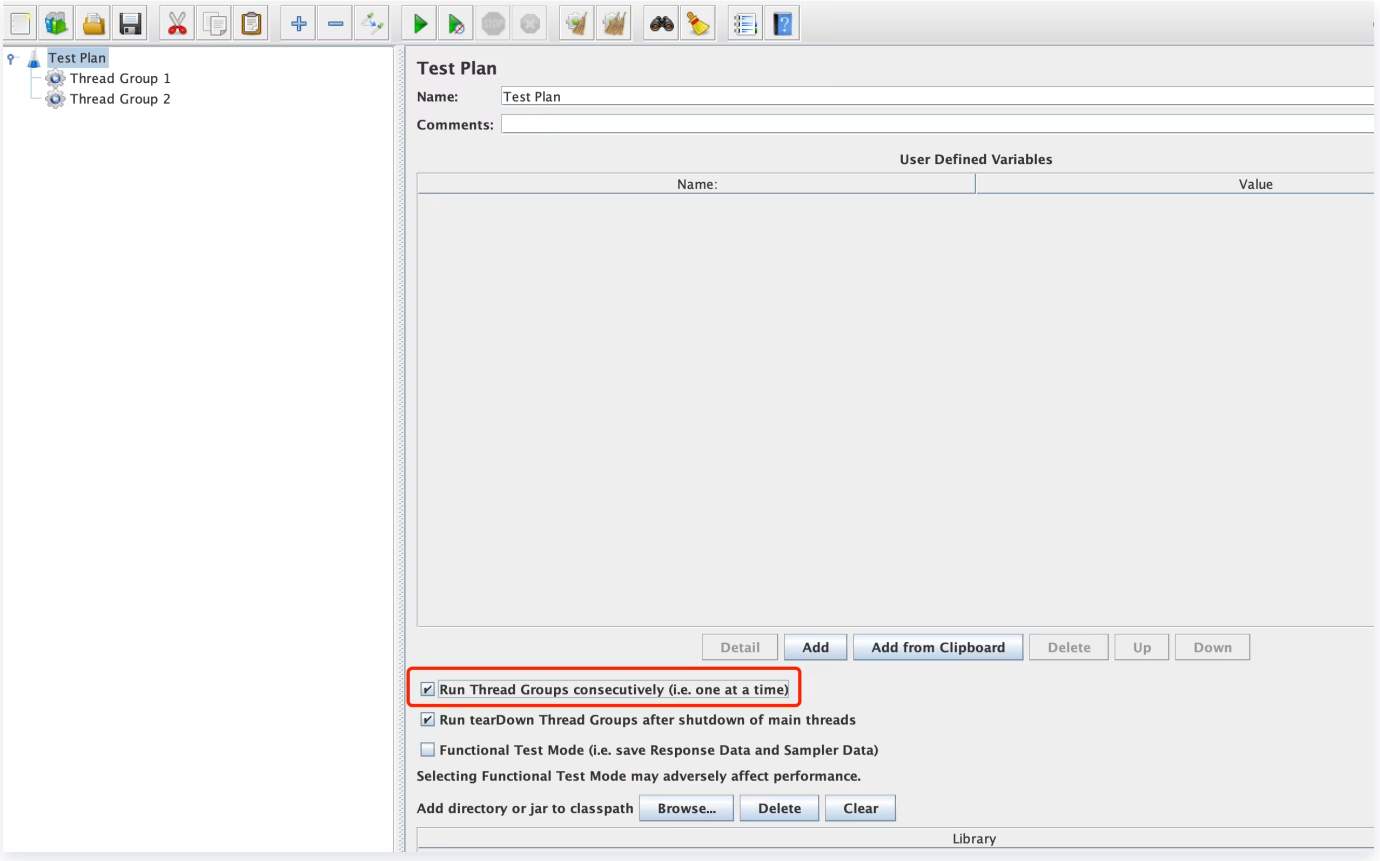
最近更新时间：2024-06-26 14:23:32

当您有多个相关的场景需要在一个脚本中运行，可以通过配置 JMeter 多线程组实现，每个线程组中配置一个用户场景。
JMeter 支持线程组并行运行和串行运行两种模式。

JMeter 线程组并行或串行

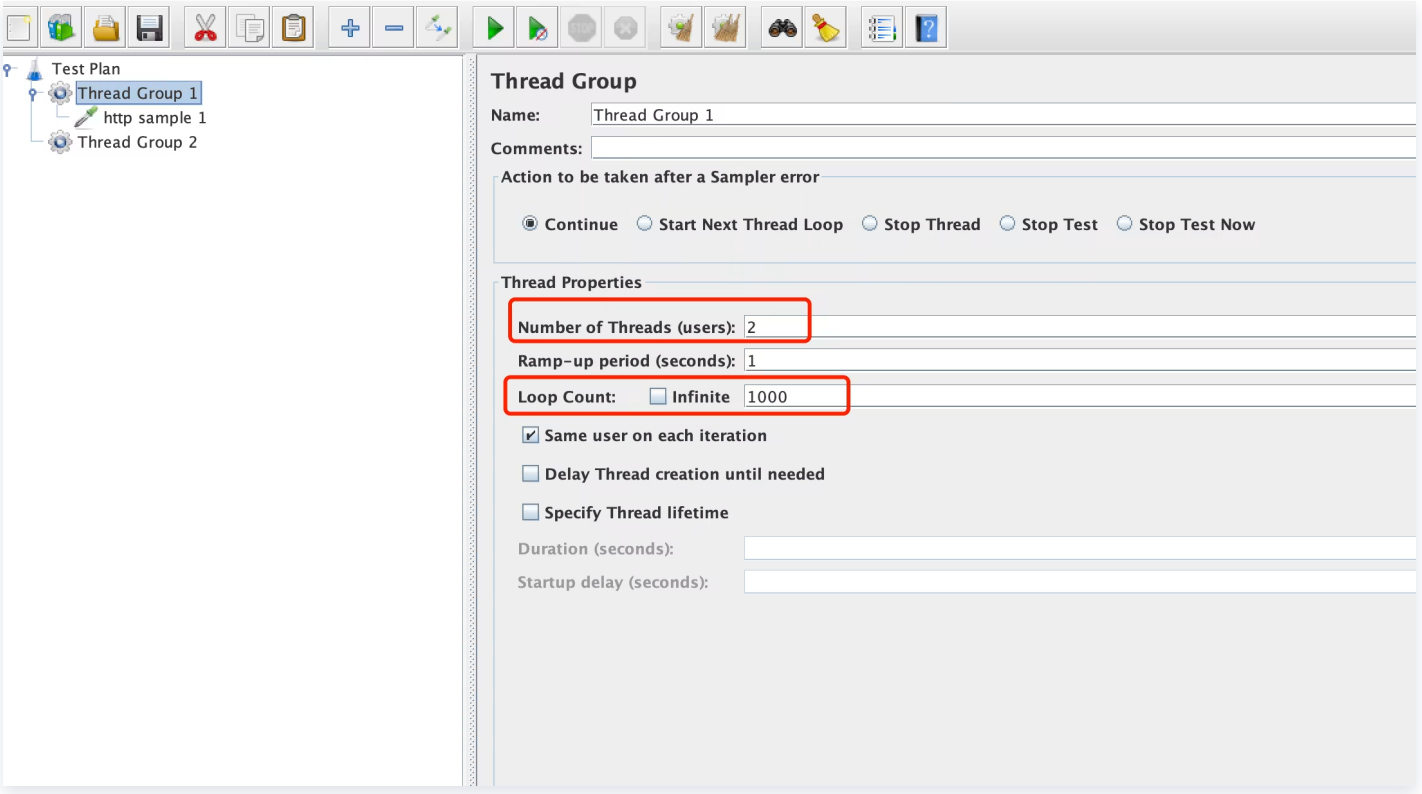
在测试计划中可以设置多线程组并行或串行执行。

- **线程组串行：** 勾选 Run Thread Groups consecutively (i.e. one at a time)。线程组串行，指的是测试计划中存在多个线程组时，第一个线程组执行完成后再执行下一个线程组。
- **线程组并行：** 不勾选 Run Thread Groups consecutively (i.e. one at a time)。线程组并行，指的是测试计划中存在多个线程组时，多个线程组同时运行。



在 JMeter 中为线程设置循环次数

当线程组串行时，当前线程组执行完成，下一个线程组才能执行。因此我们需要为线程组设置循环次数，以便当前线程组能够正常退出，下一个线程组获取执行时间。
以下示例中：Thread Group 1 包含 2个线程，**每个线程**执行1000次循环。 HTTP sample 1被执行2000次后（每个线程执行1000次），Thread Group 1 退出，Thread Group 2 开始执行。



使用 PTS 设置 JMeter 压力模型

PTS 支持 JMeter 线程组并行、线程组串行两种压力模式。
在 PTS 中配置的压力模型，会重写 JMX 文件中主线程组的配置，不会影响 setUp 和 tearDown 线程组。

PTS 配置线程组并行

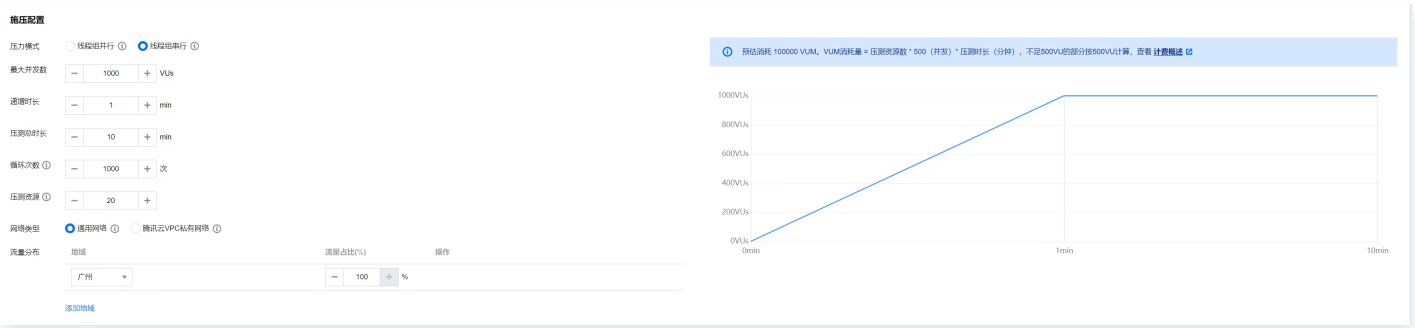
压力模式勾选**线程组并行**，设置期望的最大并发数1000，递增步数3，压测时长10分钟。右侧会渲染出对应的压力模型。

- 选择线程组并行：会覆盖 JMeter Test Plan 中的 **Run Thread Groups consecutively (i.e. one at a time)配置为不勾选**，代表线程组并行。
- 最大并发数1000： **会在多个主线程组中按比例分配**。假设用户 JMX 脚本中有两个线程组，A 线程组设置线程数量为10，B 线程组设置线程数量是20。那么压测时，A 线程组分配的线程数为334， B 线程组分配的线程数为667（小数向上取整）。
- 递增步数，递增时长，压测总时长会应用到每个主线程组。多个主线程组的压力模型合并，即等价于用户在PTS上配置的压力模型。

PTS 配置线程组串行

压力模式选择**线程组串行**，设置最大并发数1000，递增时长1min, 压测总时长10min，循环次数1000次。

- 选择线程组串行：会覆盖 JMeter Test Plan 中的 **Run Thread Groups consecutively (i.e. one at a time)配置为勾选**，代表线程组串行。一个线程组执行完成后，再执行下一个线程组。
- **并发配置会应用到每个主线程组上。**在本案例中每个主线程组的最大并发数都是1000，递增时长1min, 压测总时长10min, 循环次数1000次。
- 循环次数作用于每个线程，**代表每个线程执行循环的次数**。循环次数和压测时长，有一个达到设置值，就会停止当前并发。
- 线程组串行模式下，必须设置循环次数，以便当前线程组达到循环次数后能够退出，下个线程组获得执行时间。
- 一个线程组所有并发退出后，当前线程组执行完成，开始执行下一个线程组。



> **注意：**
> 线程组串行模式下，必须设置循环次数，以便当前线程组达到循环次数后能够退出，下个线程组获得执行时间。

JMeter 进行 WebSocket 压测

最近更新时间：2024-06-07 17:20:32

本文介绍如何通过引入插件，使用 JMeter 进行 WebSocket 压测。

背景

WebSocket 是常见的网络通信协议，随着实时 Web 应用程序的普及，确保 WebSocket 连接的性能变得至关重要。压测 WebSocket 协议有助于评估业务在高负载和高并发连接下的表现，确保在大量用户同时使用时仍能保持良好的响应速度和数据传输质量。
虽然 JMeter 没有原生支持对 WebSocket 协议的压测，但 JMeter 支持通过插件扩展，额外增加对 WebSocket 协议的压测。其中最常用的插件是 WebSocket Samplers by Peter Doornbosch，该插件提供 6 种采样器，可以满足绝大部分 WebSocket 的压测需求。

插件版本

云压测支持 WebSocket 压测，提供与原生 JMeter 压测一致的使用体验，对稳定的插件版本 jmeter-websocket-samplers-1.2.8.jar 进行了埋点，支持压测过程中的数据上报；需注意插件版本为 1.2.8，其他版本的插件可能会出现缺少埋点导致数据不完整的情况。插件地址：https://bitbucket.org/pjtr/jmeter-websocket-samplers/downloads/?tab=downloads 。

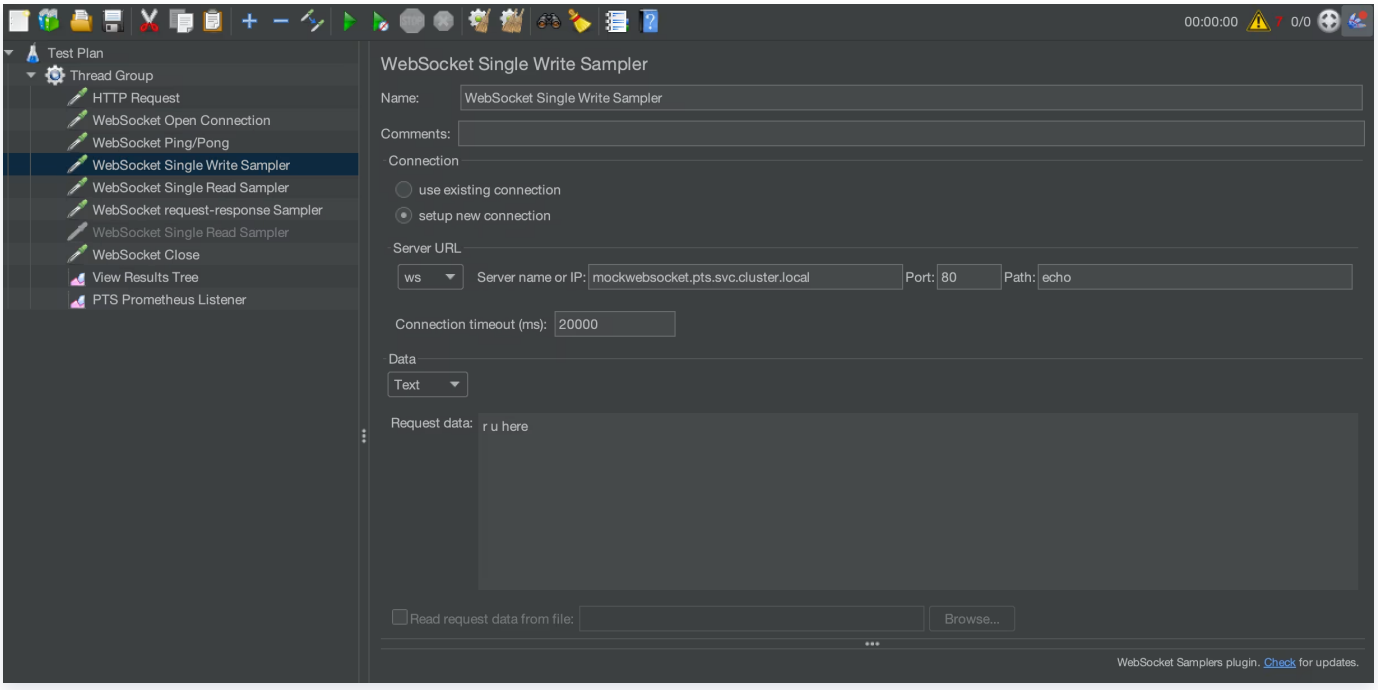
使用方法

新建 JMeter 压测场景

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中点击**云压测** > **测试场景**。
3. 单击**新建场景**，在**新建测试场景**页面选择 **JMeter** 类型的测试场景。

上传压测文件

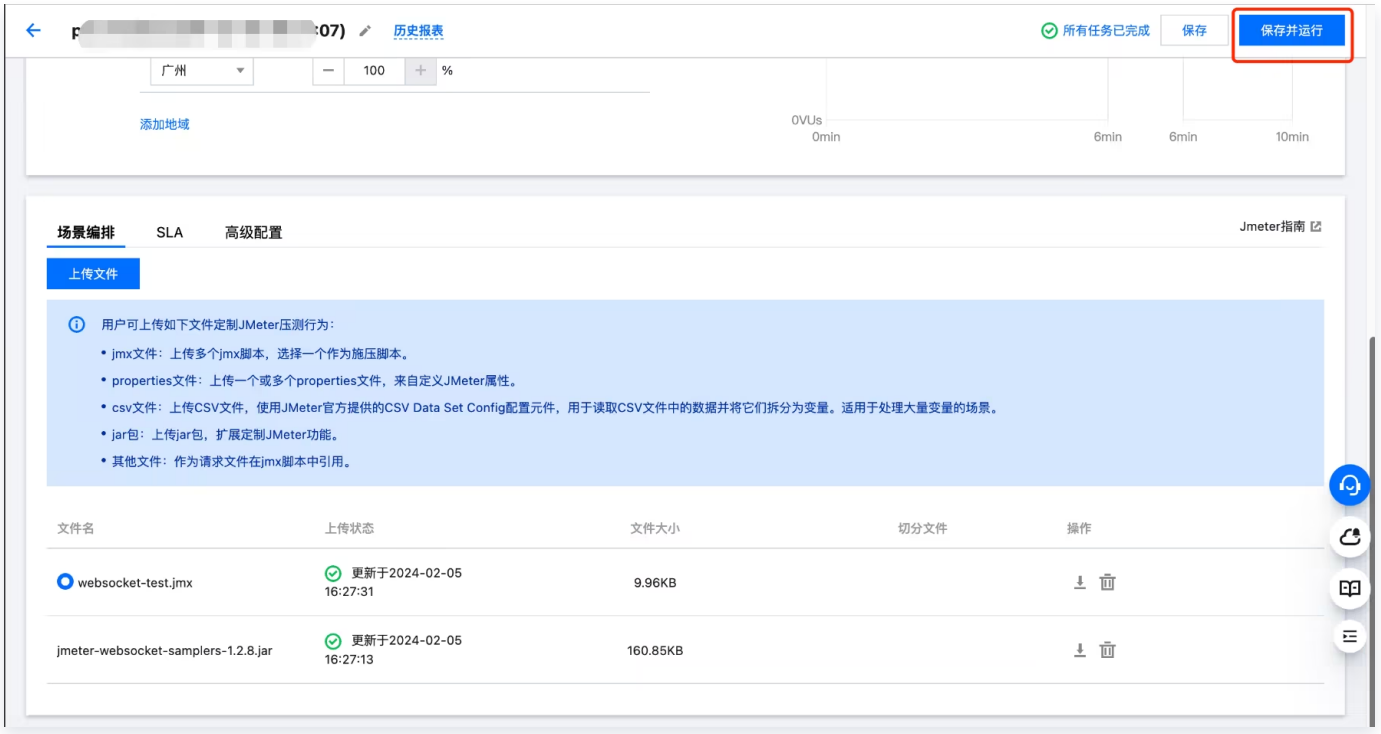
1. 在本地 JMeter 中根据业务需要进行压测计划的配置；



2. 本地调试成功后，将对应的 JMX 脚本和 WebSocket 插件以及其他需要的文件上传到控制台。

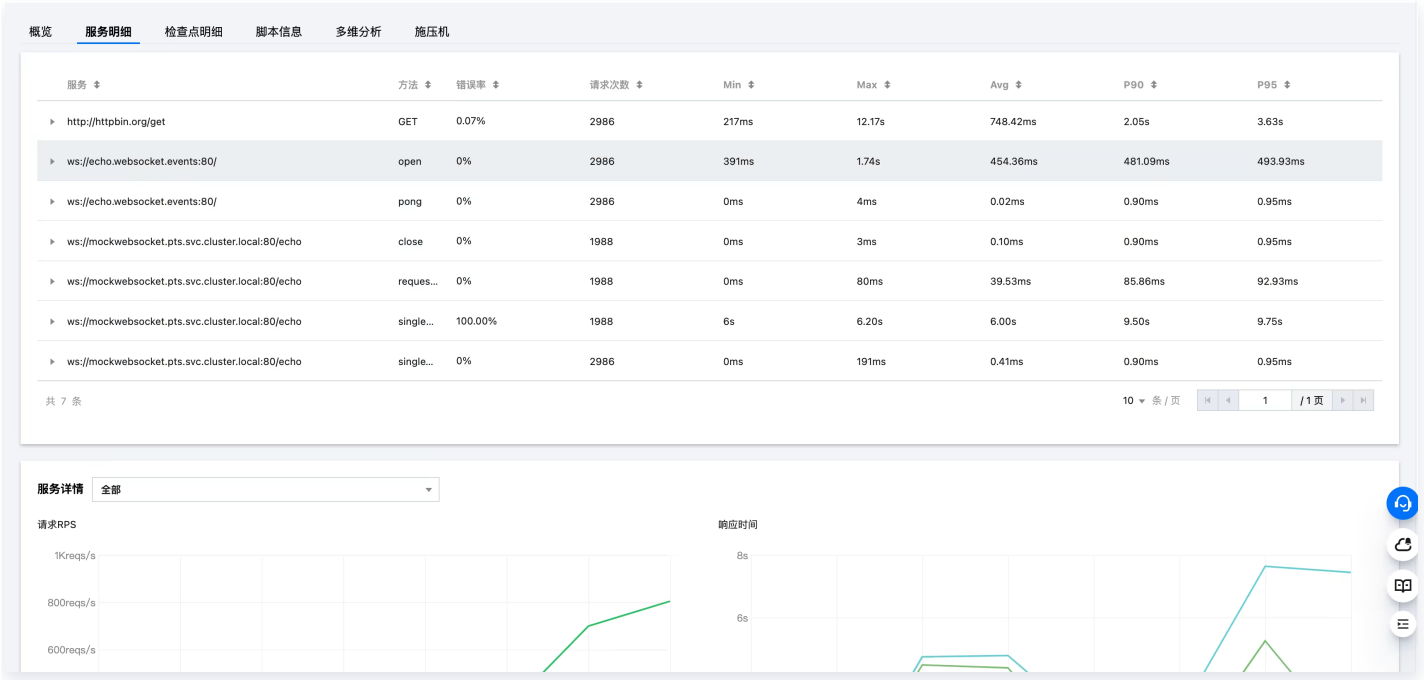
运行压测脚本

点击右上角的**保存并运行**，开始执行压测任务，并生成实时报告。



查看压测结果

云压测对 WebSocket 插件中各采样器的结果进行了埋点，压测过程中在控制台上可以看到对应不同方法的压测数据，可以根据业务需要进行查看。


概览
服务明细
检查点明细
脚本信息
多维分析
施压机
服务
方法
错误率
请求次数
Min
Max
Avg
P90
P95
http://httpbin.org/get GET 0.07% 2986 217ms 12.17s 748.42ms 2.05s 3.63s
ws://echo.websocket.events:80/ open 0% 2986 391ms 1.74s 454.36ms 481.09ms 493.93ms
ws://echo.websocket.events:80/ pong 0% 2986 0ms 4ms 0.02ms 0.90ms 0.95ms
ws://mockwebsocket.pts.svc.cluster.local:80/echo close 0% 1988 0ms 3ms 0.10ms 0.90ms 0.95ms
ws://mockwebsocket.pts.svc.cluster.local:80/echo reques... 0% 1988 0ms 80ms 39.53ms 85.86ms 92.93ms
ws://mockwebsocket.pts.svc.cluster.local:80/echo single... 100.00% 1988 6s 6.20s 6.00s 9.50s 9.75s
ws://mockwebsocket.pts.svc.cluster.local:80/echo single... 0% 2986 0ms 191ms 0.41ms 0.90ms 0.95ms
共 7 条
10 条 / 页
1
/ 1 页
服务详情
全部
请求RPS
1Kreqs/s
800reqs/s
600reqs/s
响应时间
8s
6s

JMeter 请求和检查点日志打印

最近更新时间：2024-06-24 11:46:11

在 PTS 中，请求和检查点的总体情况，可以在 云压测控制台 > 测试场景 的**服务明细**和**检查点明细**中查看；单个请求的发送和接收详情，可以在控制台的**请求采样**中查看；具体的查看方法可以参考 解读报告 中的对应内容。
如果除了**解读报告**的内容外，对请求或检查点还有其他的查看需求，例如：

- 查看请求采样中未记录的其他请求细节；
- 查看设置的检查断言失败消息；
- 查看检查断言失败时的请求内容；
- 等等...

可以在执行过程中通过日志打印的方式，将需要查看的内容在**施压机 > 日志**下的**引擎输出**中打印出来。

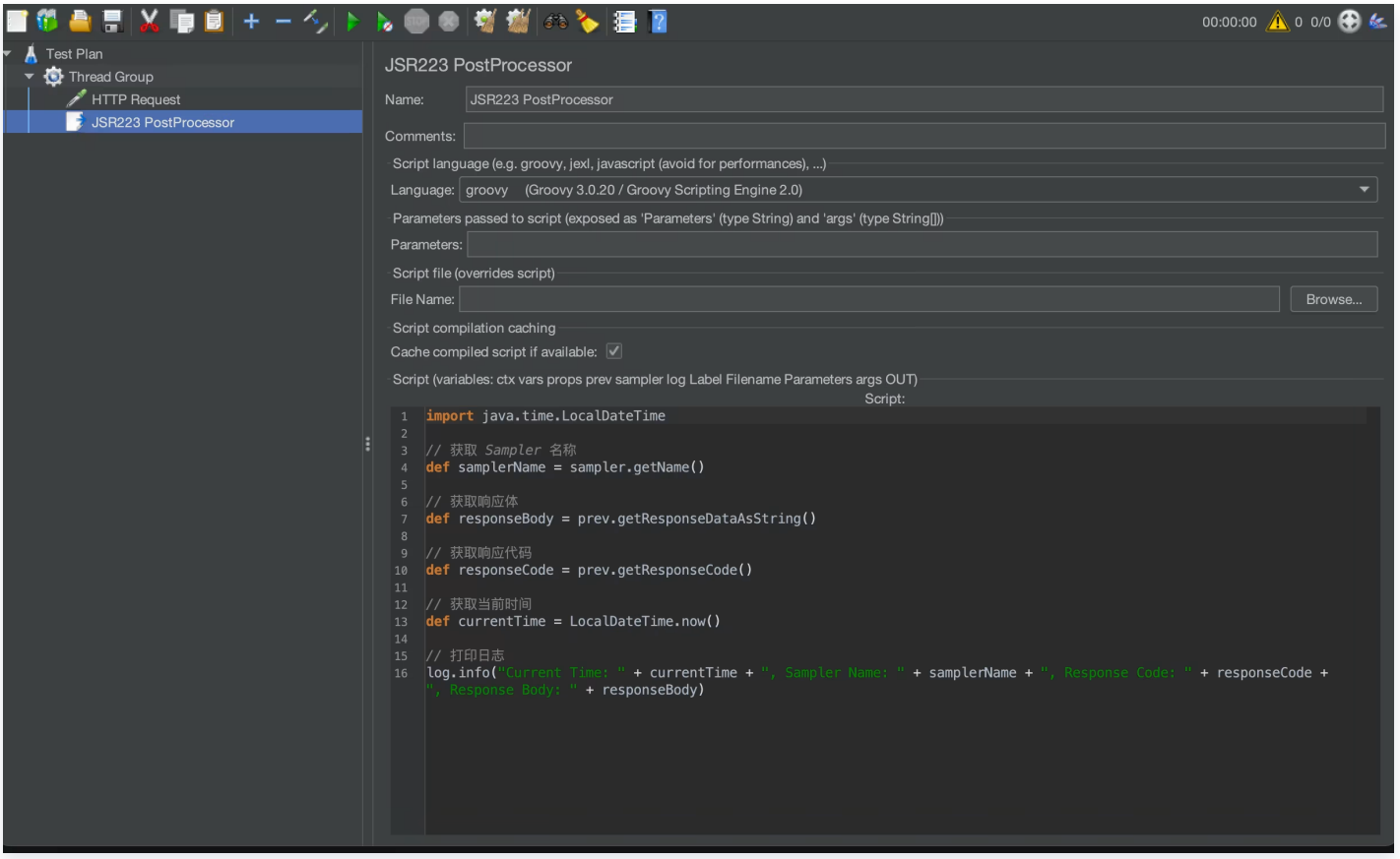
> ⚠ **注意：**
> 在压测任务执行的过程中打印额外的日志，会**占用压测机的资源**，且**日志的采集和展示在控制台上的速率是有限的**，如果没有必要，**在正式压测的时候不建议这样使用**。

请求日志

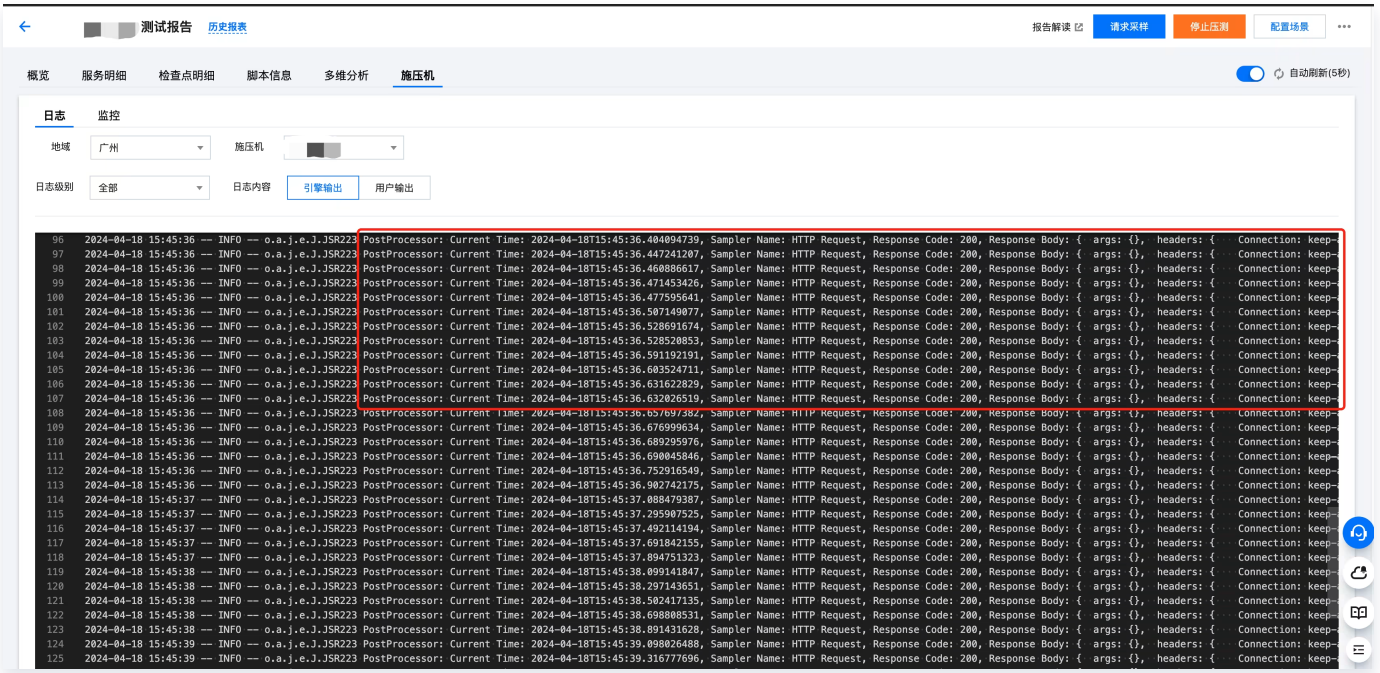
根据 JMeter 的 执行顺序，在请求采样 Sampler 之后的阶段都可以知道请求的执行情况，因此可以在 JMeter 的 Sampler 后面添加“JSR223 PostProcessor”，顾名思义，利用 JSR223 后置处理器在 Sampler 之后，通过脚本将请求细节打印到引擎日志中查看。
以下是 Groovy 脚本样例，其中 `prev` 可以代表请求采样的结果 SampleResult，对应的方法可以参考 JMeter 官方文档；如果有其他内容需要打印，用户可以自行进行数据获取和输出。

```
import java.time.LocalDateTime

// 获取 Sampler 名称
def samplerName = sampler.getName()

// 获取响应体
def responseBody = prev.getResponseDataAsString()

// 获取响应代码
def responseCode = prev.getResponseCode()

// 获取当前时间
def currentTime = LocalDateTime.now()
```

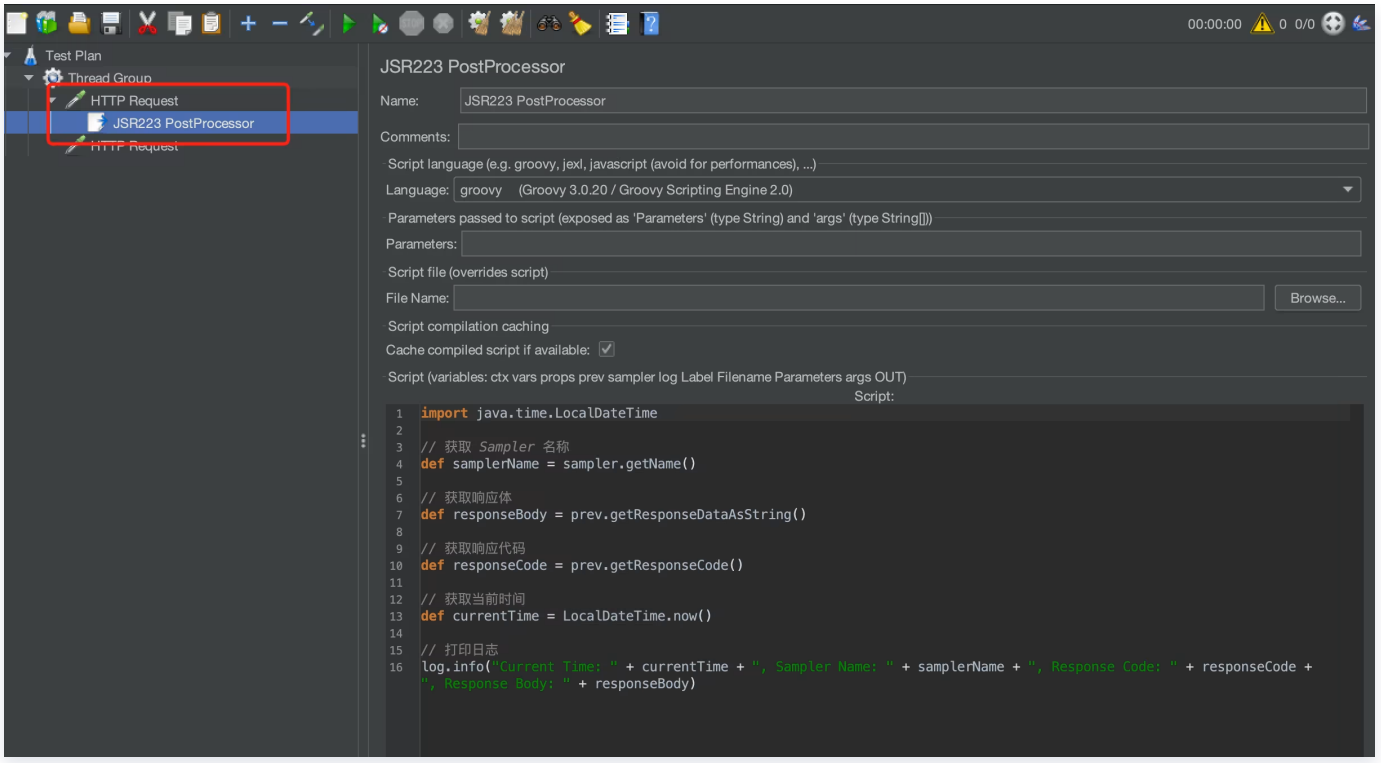
```
// 打印日志
log.info("Current Time: " + currentTime + ", Sampler Name: " +
samplerName + ", Response Code: " + responseCode + ", Response Body: " +
responseBody)
```



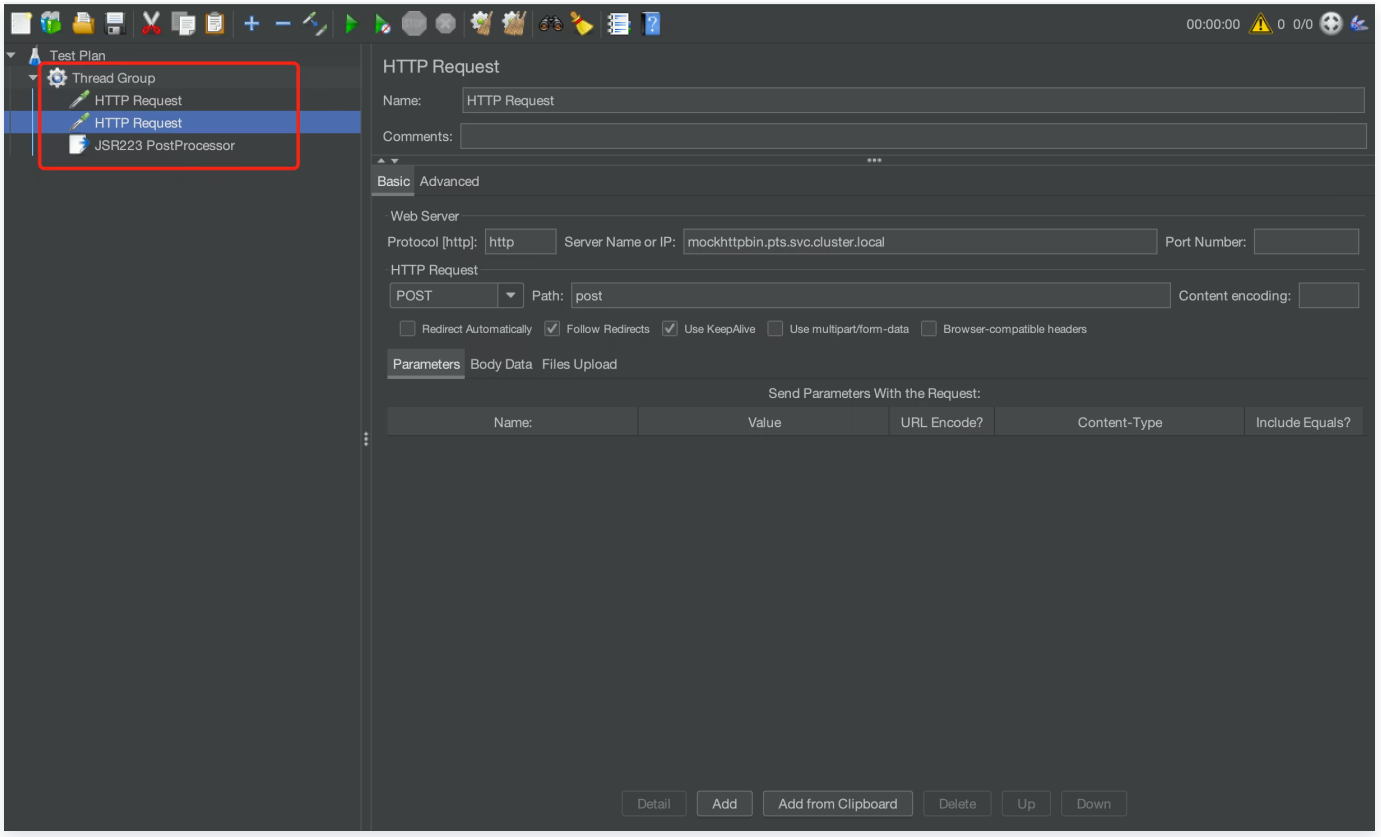
- 将该 JMX 脚本在 PTS 上执行，在控制台的**施压机**标签的**引擎输出**中，可以看到我们打印出来的请求日志：



- 如果脚本中有多个请求，但是只需要打印单个请求的细节查看，不需要全部打印，那么可以将 Post Processor 放在对应的请求里面，如图所示：



- 如果需要把所有的请求细节都打印出来，那么可以将 Post Processor 放在和请求并列的位置，如图所示：

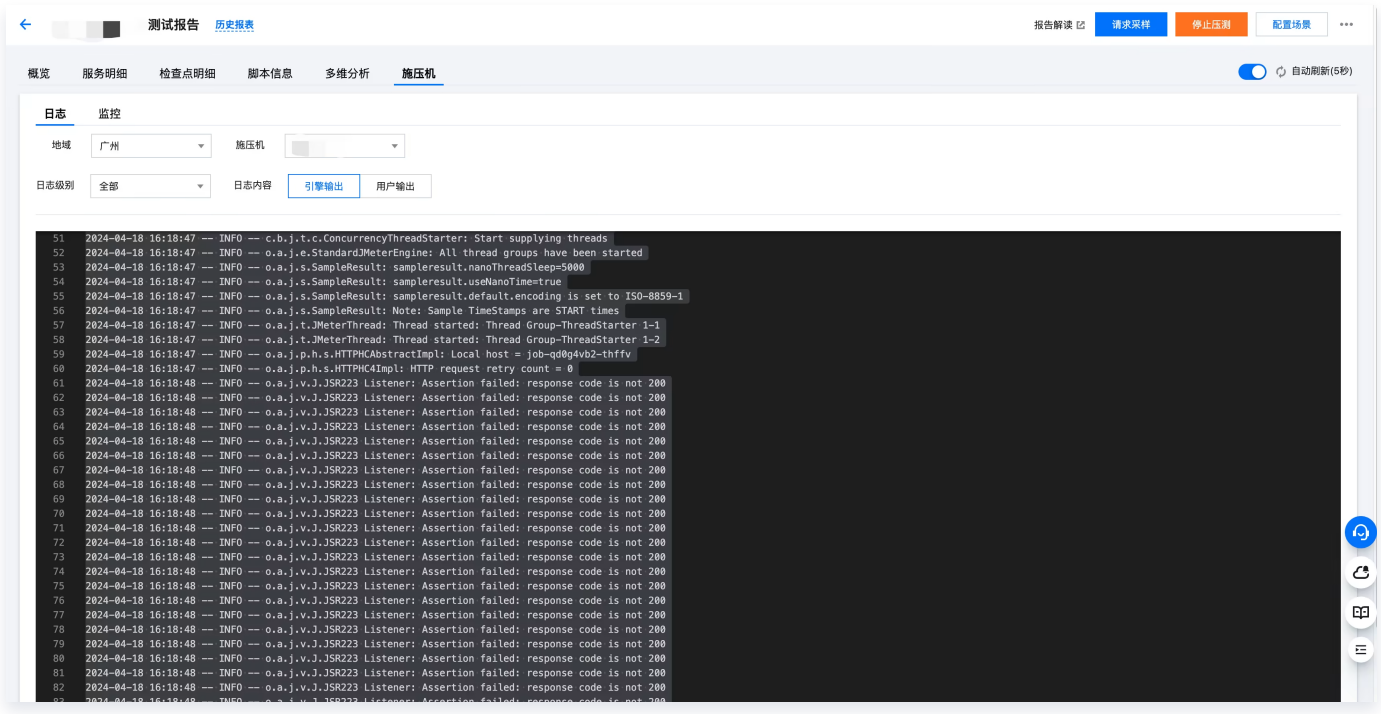


检查点日志

根据 JMeter 的 执行顺序，对于检查点的结果，不能使用 Post Processor 来进行获取，因为该阶段执行的时候，检查断言还没有执行，只有在 Listener 阶段才能知道检查断言的结果；因此，可以使用 Listener 来通过脚本将检查点细节打印到引擎日志中查看，使用 JSR223 Listener，以下是 Groovy 脚本样例，方法 `getAssertionResults()` 返回的 `AssertionResult` 可以参考 JMeter 官方文档。

```
import org.apache.jmeter.assertions.AssertionResult;

// 获取断言结果
AssertionResult[] results = prev.getAssertionResults();

// 遍历断言结果
for (int i = 0; i < results.length; i++) {
    AssertionResult result = results[i];
    if (result.isFailure() || result.isError()) {
        // 打印断言失败或错误信息
        log.info("Assertion failed: " + result.getFailureMessage());
    }
}
```

- 将该 JMX 脚本在 PTS 上执行，在控制台的**施压机**标签的**引擎日志**中，可以看到我们打印出来的日志：



- 这里只打印了检查断言的结果和检查断言的 Failure Message，如果有需要，也可以参考前文的请求日志将检查断言失败的请求细节打印出来。

```
import org.apache.jmeter.assertions.AssertionResult;

// 获取断言结果
AssertionResult[] results = prev.getAssertionResults();

// 遍历断言结果
for (int i = 0; i < results.length; i++) {
    AssertionResult result = results[i];
    if (result.isFailure() || result.isError()) {
        // 打印断言失败或错误信息
        log.info("Assertion failed: " + result.getFailureMessage());

        // 获取 Sampler 名称
        def samplerName = sampler.getName()

        // 获取响应码
        def responseCode = prev.getResponseCode()

        log.info("Sampler Name: " + samplerName + ", Response Code: " +
responseCode)
    }
}
```

- 如果 JMX 脚本中有多个检查断言，需要打印单个或需要全部打印的情况，可以参考前文有多个请求的情况下打印日志的方式，将 Listener 放置在不同的位置即可。

管理项目
项目概述

最近更新时间：2024-11-08 15:33:22

概念介绍

在 PTS 中，您可以用项目（Project）来组织压测资源、管理资源权限。

- **组织压测资源：**
  - 一个压测项目的资源，可包含多个压测场景（Scenario）、多个告警联系人等。
  - 项目与项目之间的资源是互相隔离的。
  - 您可从每个页面顶端的列表里切换项目，来管理该项目下的所有资源。
- **管理资源权限：**
  - PTS 以项目资源为粒度，对用户进行鉴权。
  - 鉴权通过腾讯云标签实现。您可以为您的项目绑定一些标签，对子用户进行鉴权（子用户必须对其中任一标签具备权限，才能访问该项目的资源）。

新建项目

最近更新时间：2024-10-31 18:11:52

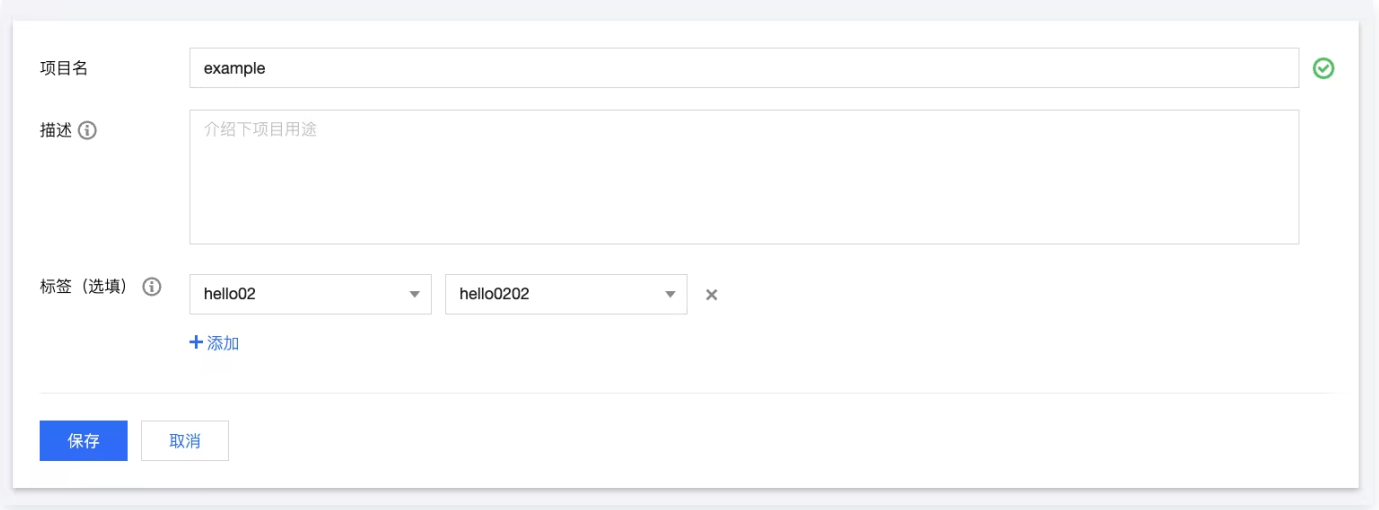
操作场景

云压测的项目模块用于集中管理测试场景，一个项目可包含多个测试场景。本文将为您介绍如何新建云压测项目。

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**云压测** > **项目列表** > **新建项目**，进入新建项目页面。

3. 填写项目名、描述以及标签，单击**保存**。您还可为项目绑定标签用于鉴权，详情请参见 标签管理 。



编辑项目

最近更新时间：2024-10-31 18:11:52

本文将为您介绍如何修改云压测项目信息，包括项目名、项目描述和标签等。

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**云压测** > **项目列表**。
3. 在新建项目管理页，找到需要编辑的项目，在操作列中单击**编辑**。

4. 在编辑页面输入需要修改的信息，修改完后单击**保存**即可。

删除项目

最近更新时间：2024-10-31 18:11:52

本文将为您介绍如何删除云压测项目。

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**云压测 > 项目列表**。
3. 在新建项目管理页，找到需要删除的项目，在操作列中单击**删除图标**并确定删除。

管理场景
场景概述

最近更新时间：2024-07-05 11:48:11

概念介绍

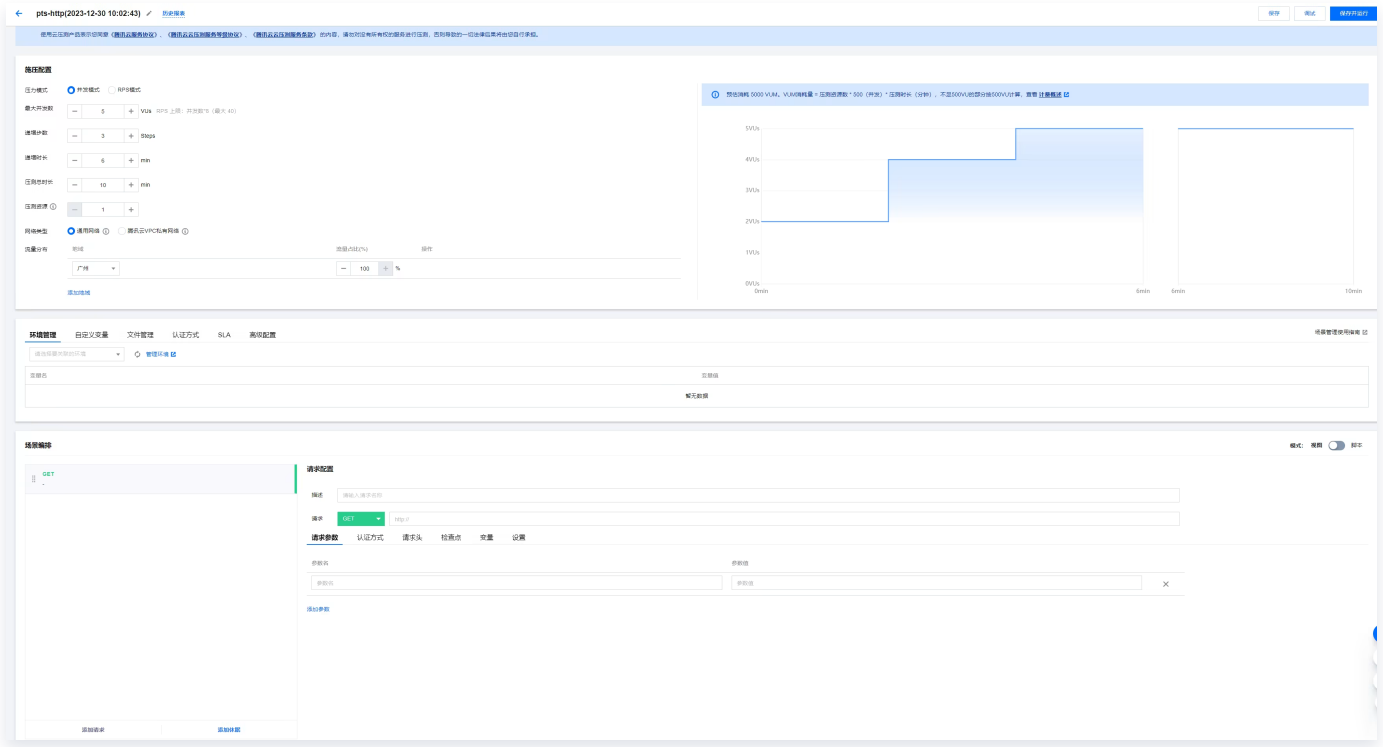
在 PTS 里，场景（Scenario）是对一个真实业务场景的压力状况的模拟，也是管理一次压测的配置、资源、生命周期的最小单元。
根据编排方式的不同，PTS 提供以下几种场景模式：

- 简单模式：简单直观，适合快速上手。
  - 通过控制台的图形界面，可视化、零代码实现请求链路编排。
  - 支持 HTTP 协议的 GET，POST，PUT，PATCH，DELETE 请求。
- 脚本模式：逻辑编排灵活、协议支持灵活、有脚本模板可供参考。
  - 在控制台的在线代码编辑器里，编写 JavaScript 脚本，以 as code 模式实现对请求链路的编排。
  - 支持 HTTP、WebSocket、TRPC、GOFree 等网络传输协议，和 Protobuf、JCE 等数据序列化/反序列化协议。
  - PTS 提供了多种类别的脚本模板，供您参考以编写自己的脚本，方便地满足个性化需求。
- JMeter 模式：原生支持 JMeter 压测。
  - 上传 jmx 脚本，无缝迁移 JMeter 压测体验。
  - 还支持上传 csv、jar 等文件，以使用 JMeter 扩展功能。

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**云压测>测试场景**。
3. 在测试场景页面单击**新建场景**。
4. 在创建测试场景页面选择合适的类型并单击**开始**进行创建。

5. 完成对应的场景配置后，单击右上角的**保存**，或者**调试**，或者**保存并运行**。


pts-http(2023-12-30 10:02:43)
历史报表
保存
调试
保存并运行
压测配置
压力模式
并发模式
RPS模式
最大并发数
VUs
递增步数
Steps
递增时长
min
压测总时长
min
压测资源
网络类型
通用网络
腾讯云VPC私有网络
流量分布
地域
流量占比(%)
操作
广州
100
%
添加地域
预估消耗 5000 VUM，VUM消耗量 = 压测资源数 * 500（并发）* 压测时长（分钟），不足500VU的部分按500VU计算。查看 计费概述
5VUs
4VUs
3VUs
2VUs
1VUs
0VUs
0min
6min
6min
10min
环境管理
自定义变量
文件管理
认证方式
SLA
高级配置
场景管理使用指南
管理环境
变量名
变量值
暂无数据
场景编排
模式：
脚本
GET
请求配置
描述
请输入请求名称
请求
GET
http://
请求参数
认证方式
请求头
检查点
变量
设置
参数名
参数值
添加参数
添加请求
添加分组

施压配置

最近更新时间：2024-12-18 10:02:33

施压配置用于控制压测过程中的流量，来模拟真实业务场景中的流量变化及流量分布。
在场景的**施压配置**部分，您可以配置以下针对施压模式、施压时长、压力来源等的参数，来管理压测流量。
关于 VU、RPS 等相关概念的介绍，请参见 常见问题 。

压力模式

并发模式

并发指虚拟并发用户数。从业务角度，也可以理解为同时在线的用户数。可用于摸底业务系统能够承载的最大实时并发数。

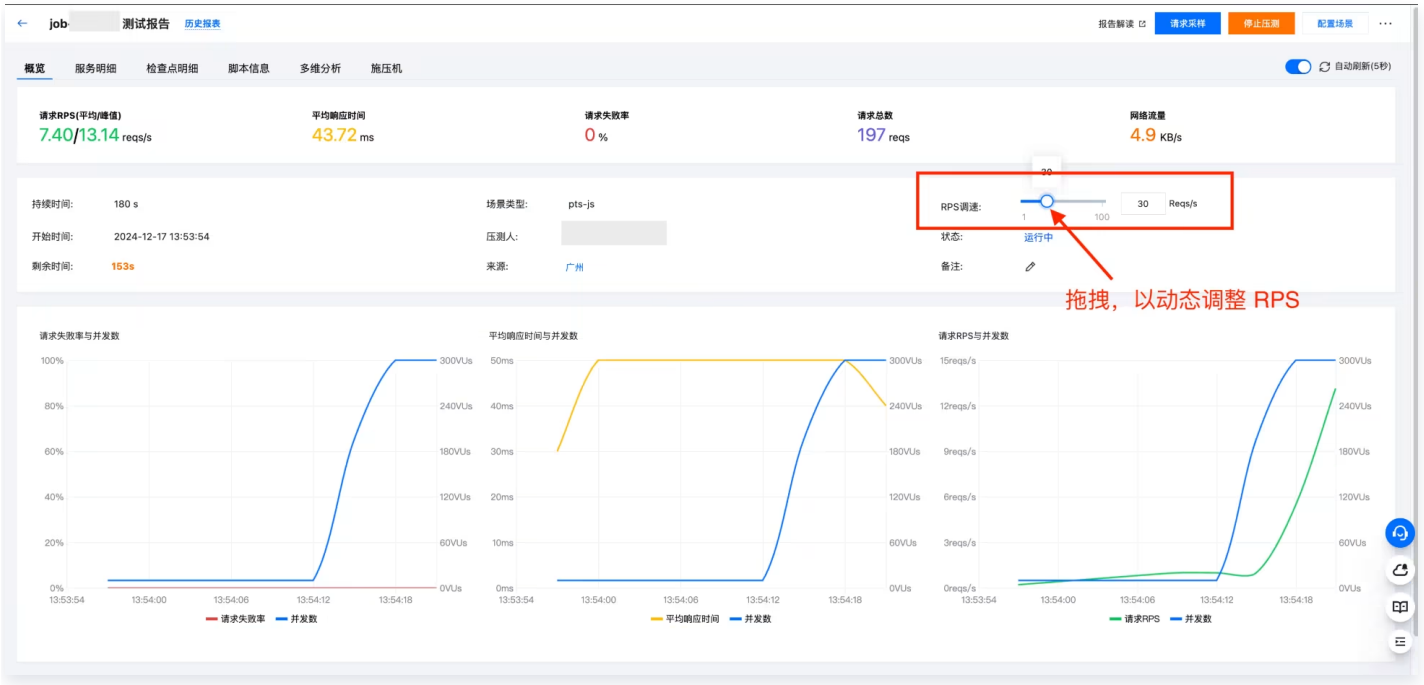
- 最大并发数（Max VUs）：虚拟用户（VUs）的最大数量。施压端通常用一个线程实现一个 VU，每个 VU 重复执行压测脚本。
- 递增步数：在 VU 递增的过程中，需将递增时长平均分成几个阶段，来达到最大并发数。（**建议3 - 5步**，步数太高变化频繁会影响线程预热启动）
- 总梯度时长：VU 递增过程的时长。
- 压测总时长：一次压测的总时长，包括 VU 递增时的运行时长，和达到最大 VU 数后的稳定运行时长。

RPS 模式

RPS 即每秒请求数量，用于衡量服务端的吞吐量。免去并发数到 RPS 的繁琐转换，来帮忙用户更好的摸底业务性能瓶颈。

- 最大 RPS：压测 RPS 上限，用来摸底业务系统的目标吞吐量。PTS 会根据最大 RPS 为压测任务分配合理的施压资源。
- 起始 RPS：压测起始 RPS，用户可以在压测过程中手工调整发压 RPS，并观察报表指标变化。
- 压测总时长：一次压测的总时长。
- 压测资源：PTS 会根据用户设置的最大 RPS，合理分配压测资源池。如果您的请求响应较慢，您可以通过适当扩大压测资源池，来确保达到目标吞吐量。
- 流量分布：将压测总流量以一定的百分比，分布于多个地域，以模拟真实场景中，来自不同地域的用户带来的流量。

RPS 压测模式下， 用户可在 测试场景 的压测报表页面调整 RPS，并实时观察系统整体指标的变化。



说明：

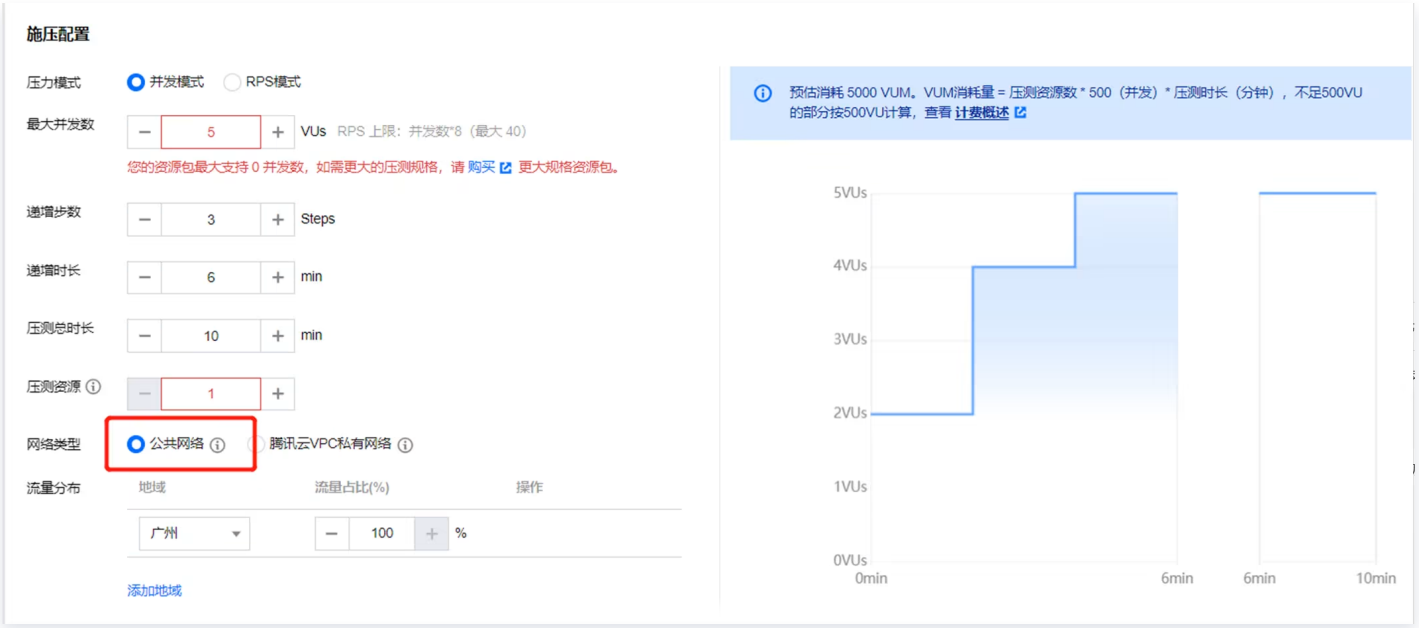
- 简单模式和脚本模式的场景：从 PTS 控制台，创建场景时在施压配置里设定 RPS 上下限；运行压测时在报告页动态拖拽调整 RPS。
- JMeter 模式的场景：编写 JMeter 压测脚本时，使用 JMeter 原生的 RPS 模式，再将该脚本上传至 PTS。详情请参见 JMeter 配置 RPS 限制 。

网络流量配置

根据压测流量来源的不同，PTS 支持**公共网络**和**腾讯云 VPC 私有网络**两种网络类型。
若您选择公共网络，则压测流量将由 PTS 为您分配的公共网络资源发出；若您选择腾讯云 VPC 私有网络，则需要您手工指定被压服务所在的 VPC 及子网，压测流量将由该 VPC 内网发出。

公共网络

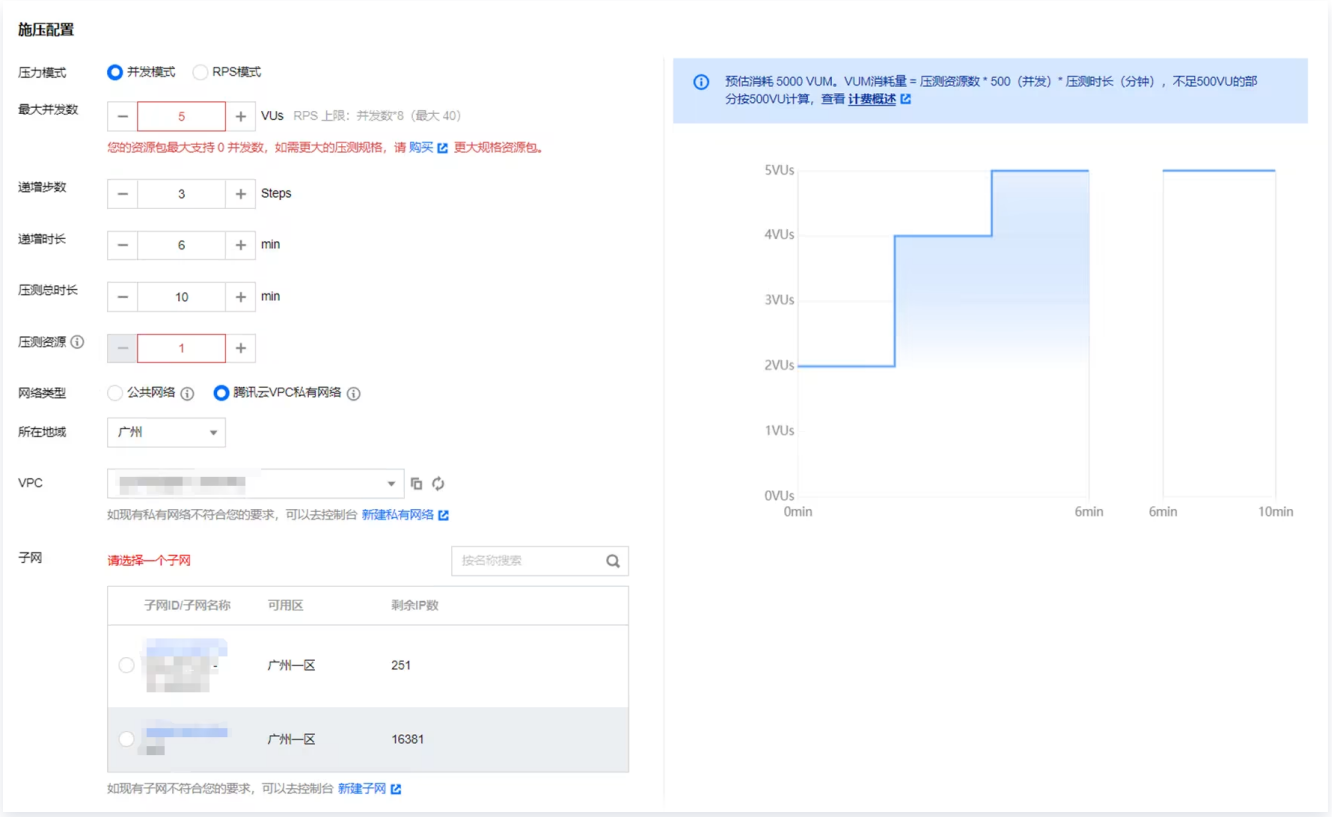
若您选择公共网络，则可压测支持公网访问的服务地址。
流量分布：将压测总流量以一定的百分比，分布于多个地域，以模拟真实场景中，来自不同地域的用户带来的流量。



腾讯云 VPC 私有网络

若您选择腾讯云 VPC 私有网络，则可压测您的 VPC 内的服务地址。

- VPC 内网性能测试完全在客户 VPC 环境进行，客户服务无需暴露服务到公网，安全性更高。另外，对于 RPC 类型的微服务，有些不方便暴露公网地址，针对 VPC 内网的每个微服务执行性能测试，可以大幅提升性能测试的效率，节省性能测试成本。
- VPC 内网性能测试模式下，压测机仅能访问客户 VPC 环境内的地址。**业务选好 VPC 与子网后，初次压测需要进行资源初始化，预计需要花5分钟时间准备资源，请耐心等待。**
  - VPC：选择被压服务所在的 VPC。
  - 子网：选择压测机使用的子网网段，子网网段的可用 IP 数量需要大于压测机节点数量，否则会导致压测任务启动失败。若子网不可选，则代表该子网所在可用区暂不支持性能测试服务。



压测资源

在 PTS 中，压测资源是引擎调度的基本单位。

一个压测资源提供500并发，以及100Mb网络带宽（上行下行各100Mb）。压测资源在 PTS 中是一个虚拟概念，代表一组资源集合，PTS 保障压测资源调度后互相隔离。利用虚拟化技术，多个压测资源可能调度到同一个物理机上共享 IP，也可能独占 IP。

您可以在施压报告中查看您的压测资源的利用率。

文件管理
使用参数文件

最近更新时间：2025-09-23 09:43:42

上传参数文件

通过上传 csv 参数文件，您可以动态引用其中的测试数据，供脚本里的变量使用。
这样，当施压机并发执行这段代码，每条请求能动态、逐行获取 csv 里的每行数据，作为请求参数使用。

参数定义

- 默认 csv 首行作为参数名。在该模式下，PTS 读取数据时，会跳过第一行。
- 若不用 csv 文件首行做参数名，则可如下图所示，取消勾选**首行作为参数名**，然后勾选该参数文件所在的行，页面会展开一个参数名编辑框，供您自行编辑参数名。

参数使用

- 在代码中，您可以用参数名作为变量名，获取变量值。
- 每个 VU 每次迭代会按照顺序取 csv 的一行数据。
- 当 csv 文件被读取完最后一行数据后，下次会回到首行，继续循环读取。

参数文件组合与切分

参数切分的逻辑

- 同施压机多 VU：所有 VU 共同逐行读取施压机上的 csv 文件，每个 VU 每次迭代会依次取一行数据。例如：施压机有一个参数文件 user.csv，在压测启动后，VU-1的第1次迭代读取到了第1行，VU-2的第1次迭代则读取到第2行，以此类推。
- 多施压机：若参数文件很大，用户可勾选**切分文件**。PTS 会将大文件逐行切分，按照流量占比分配给多个施压机使用。例如：user.csv 有4行数据，用户勾选了**切分文件**，且在广州地域有2个压测资源（施压机），则在压测启动前，施压机-1获取到包含第1、3行数据的 csv，施压机-2获取到包含第2、4行数据的 csv。

参数组合的逻辑

一个场景可上传多个 csv 参数文件，进行跨文件参数组合。

- 不同 csv 文件的列名（参数名）需保持全局唯一。
- 每个 VU 每次迭代会跨文件取到多个文件的同一行的数据。
- 若不同 csv 文件的行数不同，默认采用行数大的作为基准，行数少的 csv 文件会自行复制到跟基准文件相同行数，保证每个 VU 每次迭代都能取到多个文件的同一行的数据。

在场景中使用参数

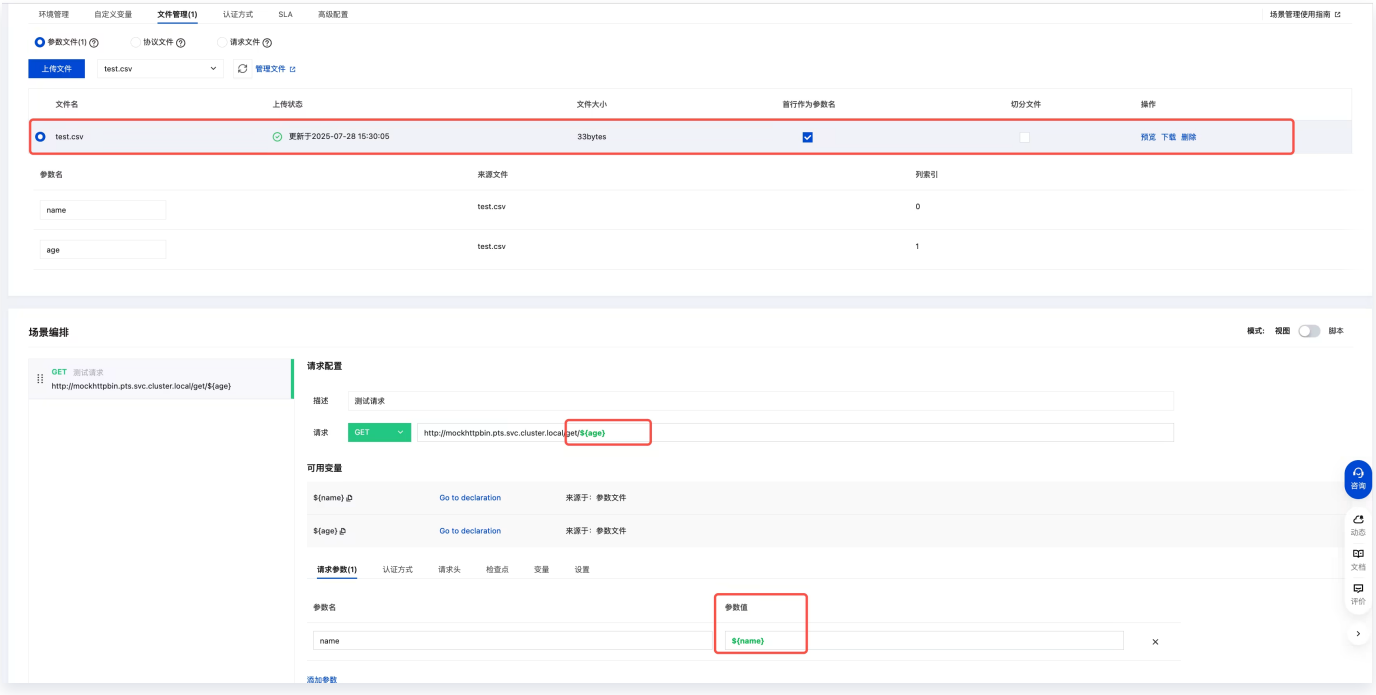
简单模式场景

在简单模式的场景中，您可以用 `${}` 的形式，使用参数文件里的参数。以下面的 dataset.csv 为例：

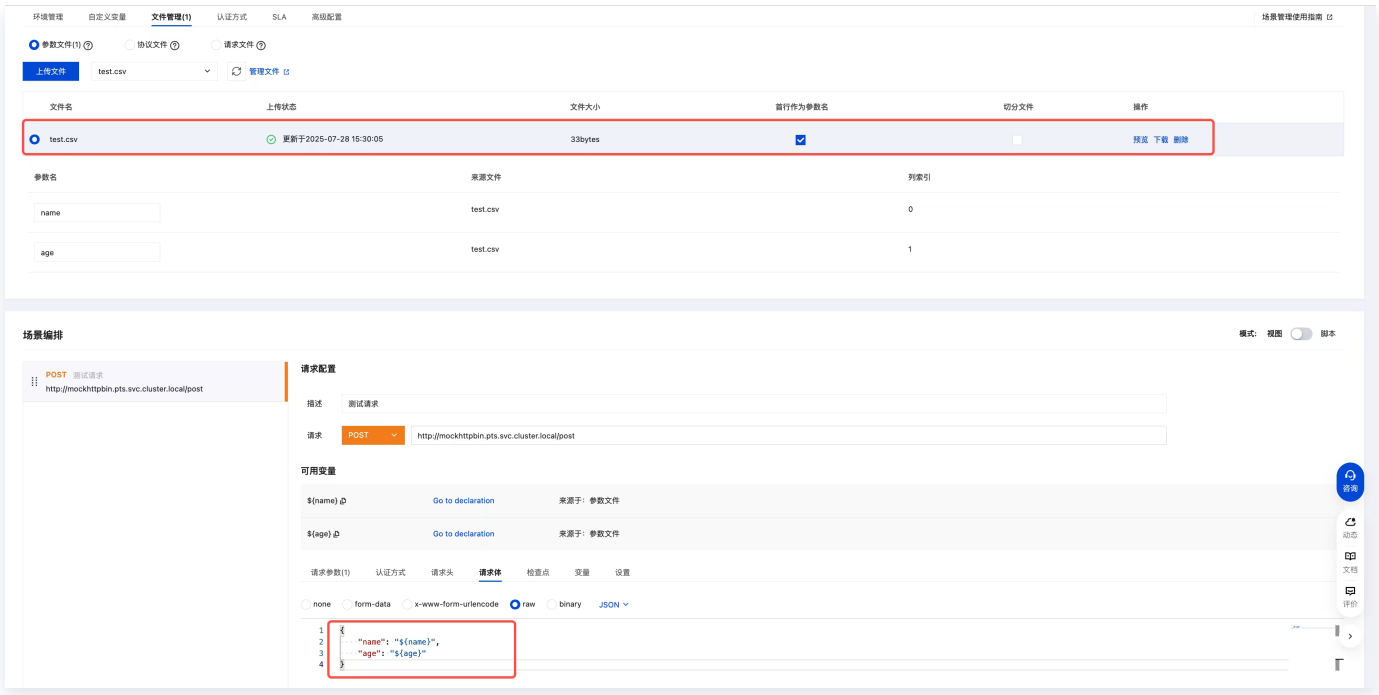
```
name,age
xiaohong,18
xiaoming,19
```

上传以上参数文件后，您即可在**请求的任何部分**通过${name}, ${age}引用参数。如下图所示：

- 在 GET 请求的 url 通过 `${age}` 引用参数文件中定义的 age 参数，在请求参数中通过 `${name}` 引用参数文件中定义的 name 参数。



- 在 POST 请求的 body中通过 `${name}`, `${age}` 引用参数文件中定义的参数：



脚本模式场景

代码示例如下，用 `dataset.get("MyKey")`，可从 csv 文件获取参数名/列名为 `MyKey` 的参数值，作为请求体里的 `value` 值。
如果期望参数文件的数据轮询一遍后，自动结束压测，可以设置全局参数：stopAfterDataConsumption: true。

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

// Global load configuration example
export const option = {
  load: {
    // Stop the test when the dataset is fully consumed. Default: false.
    stopAfterDataConsumption: true,
    // Total iterations to run. Uncomment to enable. Default: unlimited.
    // targetIterations: 100
  }
};

export default function () {
    const value = dataset.get("MyKey")
    //@ts-ignore
    const postResponse = http.post("http://httpbin.org/post", {data:
value});
    console.log(postResponse)
```

```
};
```

使用请求文件

最近更新时间：2024-06-07 15:31:51

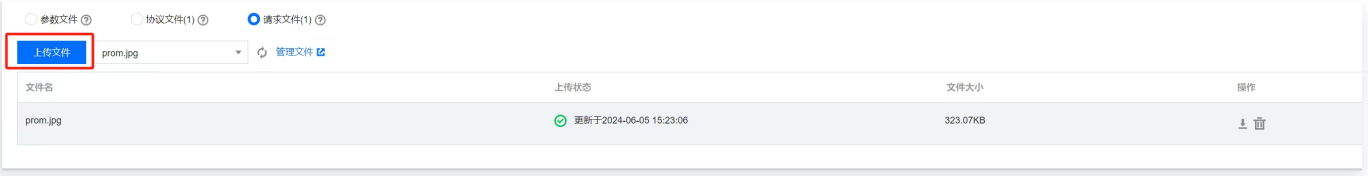
请求文件主要指代在压测场景中，构建您的请求需要使用到的文件，如压测接口中需要上传的文件。用户可以在场景中直接操作这些请求文件。

例如：

- QQ 空间用户发图片说说的压测场景，需要压测模拟用户上传图片的场景。
- 银行部分场景需要在客户端安装证书。模拟这些场景需要在客户端打开并加载证书。

使用请求文件

1. 上传请求文件：



2. 在脚本中使用请求文件：

- 定义全局变量（global）的代码：每个并发运行一次。
- 主函数（default）代码：每个并发的每次迭代运行一次，且每个 VU 在达到本次压测配置的时长上限或迭代上限之前，会持续不断地迭代执行。

因此，一些静态的文件读取等操作建议放到 global 中定义，一个并发仅需读取一次文件。避免在主函数中定义读取文件，会导致每个迭代中读取文件，带来压测性能损耗。

```
// send a post request
import http from "pts/http";

const makefile = open("Makefile");

export const options = {};

export default function main() {
  let response;

  response = http.post("https://httpbin.org/post", makefile, {
    headers: {
      "Content-Type": "application/octet-stream",
    },
  });
```

```
}
```

3. open 函数提供两种 mode, 第二个参数为空返回字符串， 为‘b’则返回 ArrayBuffer。

```
export default function main() {
 let data = open('Makefile');
  console.log(data); // SHELL  := /bin/bash ...
  data = open('Makefile', 'b');
  console.log(data); // [object ArrayBuffer]
}
```

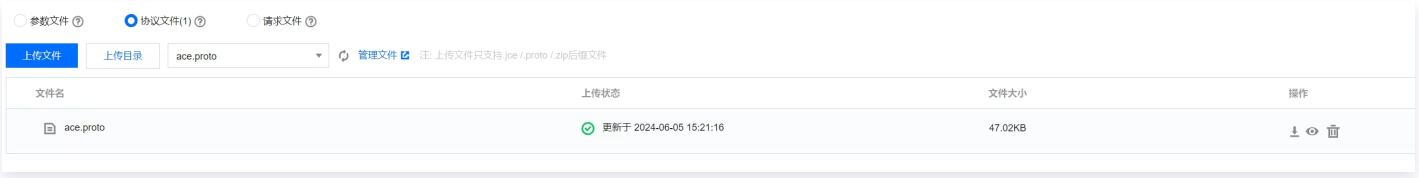
使用协议文件

最近更新时间：2024-06-07 15:31:51

gRPC 场景使用协议文件

gRPC 等协议需要用户上传协议文件，压测引擎依赖协议文件完成请求的序列化。支持用户上传文件或目录，文件名需要保持唯一，同名文件将会被新上传的文件覆盖。

- 如果用户上传 zip 文件，PTS 会解压文件，并展示解压后的文件结构。
- 如果目录或者 zip 包中包含非 Proto 文件，PTS 将忽略这些文件。



如果主 pb 文件依赖其他 proto 文件，那么也需要一并上传（谷歌提供的标准 proto 文件：google/protobuf/*.proto 不需要额外上传，PTS 会自动加载）。
用户只需要加载主 pb 即可，主 pb 依赖的其他 pb 文件，会根据主 pb 文件中 import 的路径自动递归加载。

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

const client = new grpc.Client();
client.load([],'addsvc/addsvc.proto');

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

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

    client.close();
};
```

SLA 配置

最近更新时间：2024-11-08 15:33:22

配置场景

服务等级定义（Service Level Agreement，缩写为 SLA）是您为压测场景定义的具体目标，也是判断压测是否异常的重要依据。
当您为场景配置了 SLA 规则，PTS 在运行压测任务时，会将 SLA 指标与压测过程中收集到的相关数据进行比较，然后确定目标的 SLA 状态，并根据您指定的方式做出相应处理（例如停止压测、发出通知等）。

配置 SLA

1. 登录 腾讯云可观测平台 控制台。
2. 在左侧菜单栏中单击**云压测 > 测试场景**。
3. 在**测试场景**页面单击**新建场景**。
4. 在创建测试场景页面选择合适的类型并单击**开始**进行创建。
5. 在场景配置的 **SLA** 页面，您可以为该场景创建 SLA 规则。

SLA 规则配置

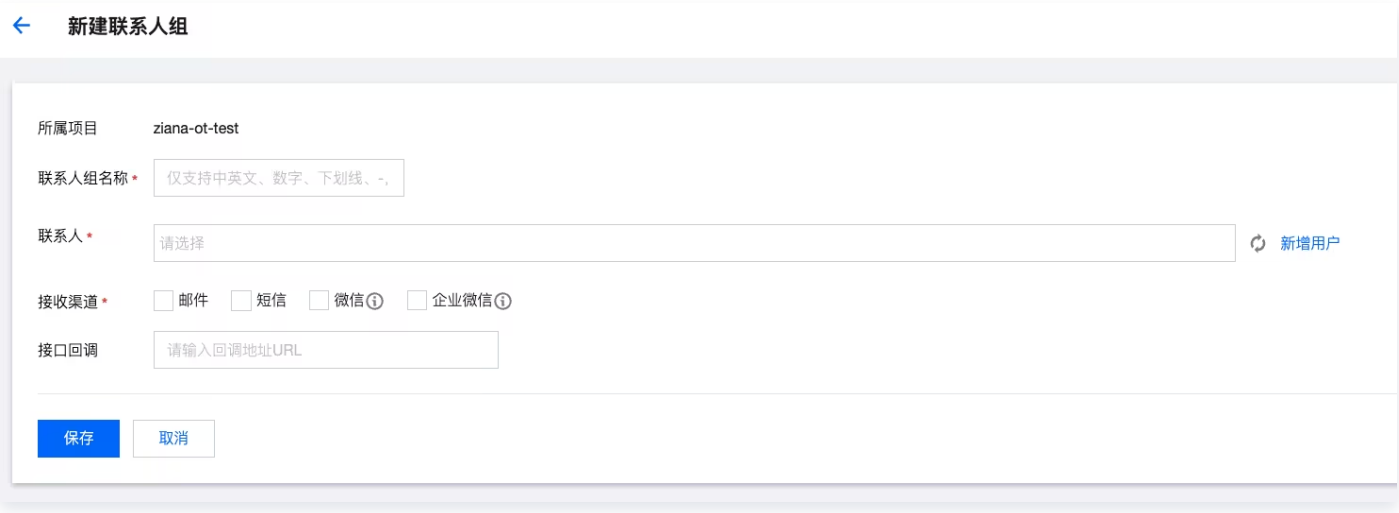
一条 SLA 规则主要包含以下信息：

- **SLA 规则表达式**：包含 SLA 指标、聚合方式、条件、阈值。
- **适用对象**：被发送请求的所有服务接口 URL。
- **是否停止压测**：当规则被触发时，是否中止压测任务。

告警联系人配置

当 SLA 规则被触发时，除了能及时停止压测，您还可以配置告警联系人，接收告警消息。PTS 能借助您的腾讯云账号下已有的通知渠道，向您发送告警消息。在场景的 SLA 配置页面，您可单击**选择联系人组**，弹出的列表页会展示出当前项目下，所有场景共用的联系人组：

若当前列表为空，您需要先新建联系人组，并添加相应的腾讯云账号作为联系人、使用该联系人账号下已有的通知渠道（例如短信、邮件等）：



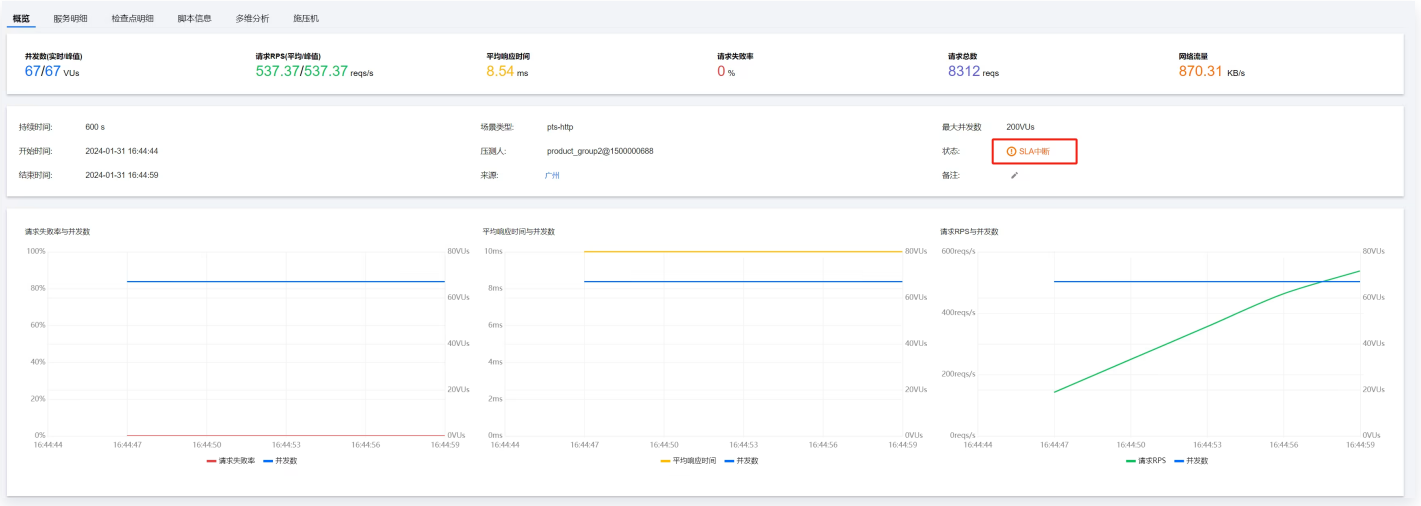
若您想要选择的联系人不存在，您可单击**新增用户**，跳转到腾讯云访问管理相关页面，维护您账号下的用户信息及通知渠道。

除了腾讯云账号下的现有通知渠道之外，您还可以自行创建企业微信机器人，并将其 Webhook 填入接收渠道，您就可以通过企业微信群，接收由该机器人发出的告警消息。

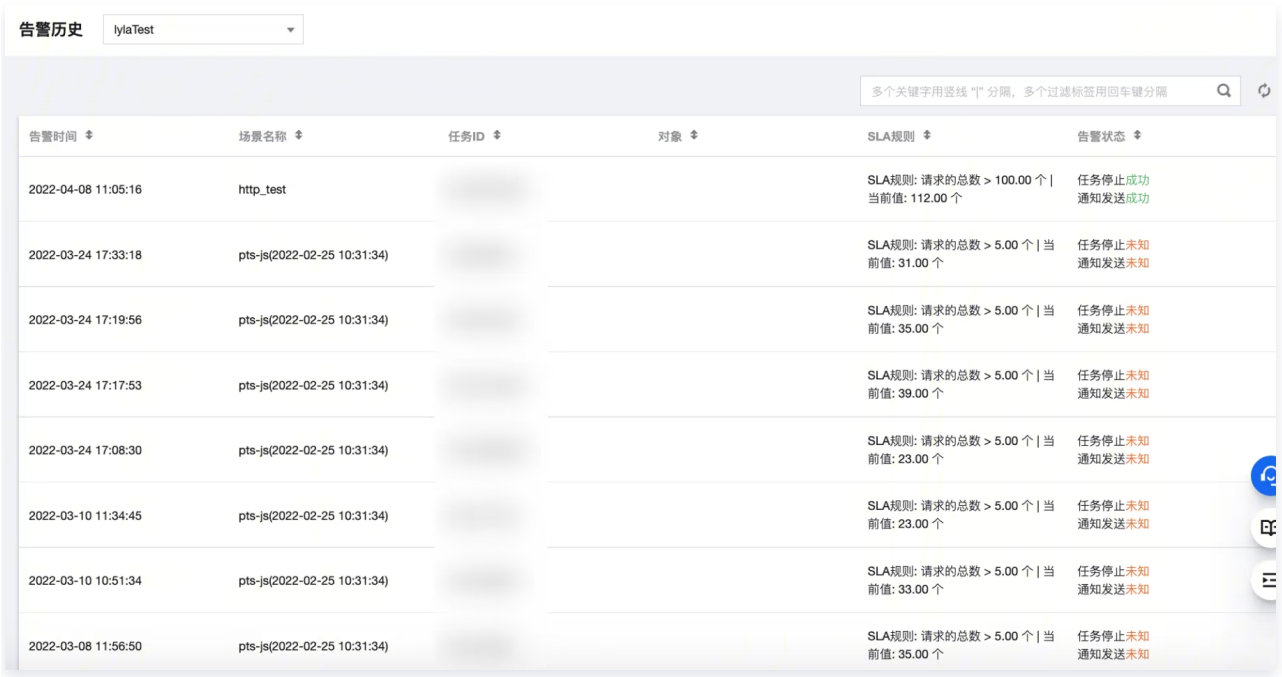
配置效果

如图所示，SLA 规则中配置当请求数大于100时，停止压测并发送通知：

压测时请求数超过100，任务中断：



告警通知也会发往相应的联系人和渠道，例如短信、邮件等。此外，还可在告警历史页面，查看同一项目下所有的告警历史记录。



高级配置
域名解析

最近更新时间：2024-11-08 15:33:22

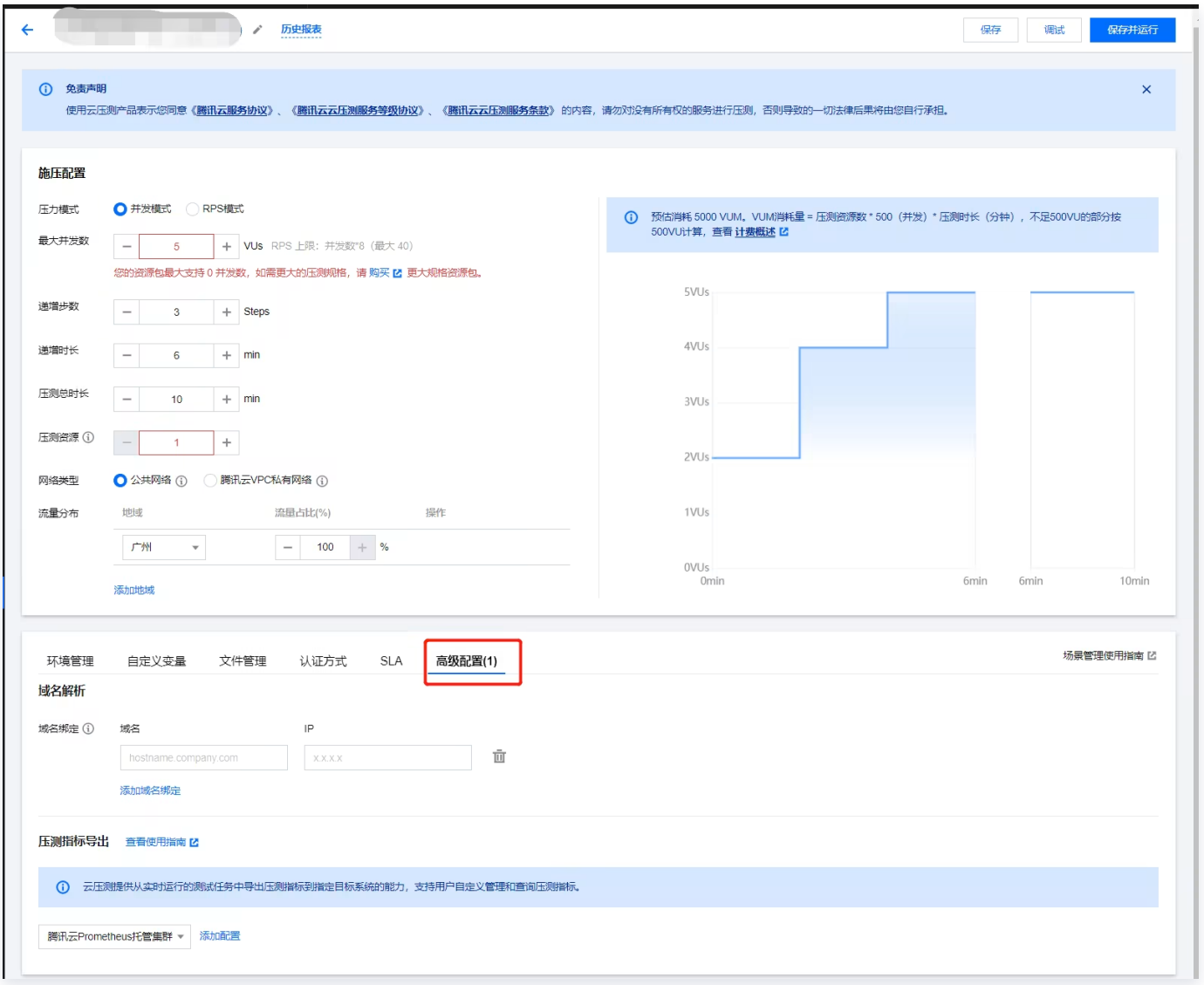
操作场景

简单理解相当于在施压机 /etc/hosts 文件中增加条目。
域名绑定是指将域名与指定的 IP 地址关联。压测时，压测流量将直接访问绑定的 IP 地址（优先级高于 DNS 服务器对域名的解析），实现对目标设施的压测。

- 在 VPC 内网压测时，用户可以为域名绑定 VPC 内地址。在不改变压测场景配置的前提下，实现对 VPC 内网服务的压测。
- 在公网压测时，如果用户有多套环境，可通过为域名绑定不同的 IP 地址，实现对不同环境进行压测。

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**云压测 > 测试场景**。
3. 在测试场景页面单击**新建场景**。
4. 在创建测试场景页面选择合适的类型并单击**开始**进行创建。
5. 在场景页面配置中选择**高级配置**，填写需要绑定的域名和 IP 即可。


历史报表
保存
调试
保存并运行
免责声明
使用云压测产品表示您同意《腾讯云服务协议》、《腾讯云云压测服务等级协议》、《腾讯云云压测服务条款》的内容，请勿对没有所有权的服务进行压测，否则导致的一切法律后果将由您自行承担。
施压配置
压力模式
并发模式
RPS模式
最大并发数
5
VUs
RPS 上限：并发数*8（最大 40）
您的资源包最大支持 0 并发数，如需更大的压测规格，请 购买 更大规格资源包。
递增步数
3
Steps
递增时长
6
min
压测总时长
10
min
压测资源
1
网络类型
公共网络
腾讯云VPC私有网络
流量分布
地域
流量占比(%)
操作
广州
100
%
添加地域
预估消耗 5000 VUM。VUM消耗量 = 压测资源数 * 500（并发）* 压测时长（分钟），不足500VU的部分按500VU计算，查看 计费概述
5VUs
4VUs
3VUs
2VUs
1VUs
0VUs
0min
6min
6min
10min
环境管理
自定义变量
文件管理
认证方式
SLA
高级配置(1)
场景管理使用指南
域名解析
域名绑定
域名
IP
hostname.company.com
x.x.x.x
添加域名绑定
压测指标导出
查看使用指南
云压测提供从实时运行的测试任务中导出压测指标到指定目标系统的能力，支持用户自定义管理和查询压测指标。
腾讯云Prometheus托管集群
添加配置

压测指标导出
压测指标导出使用指南

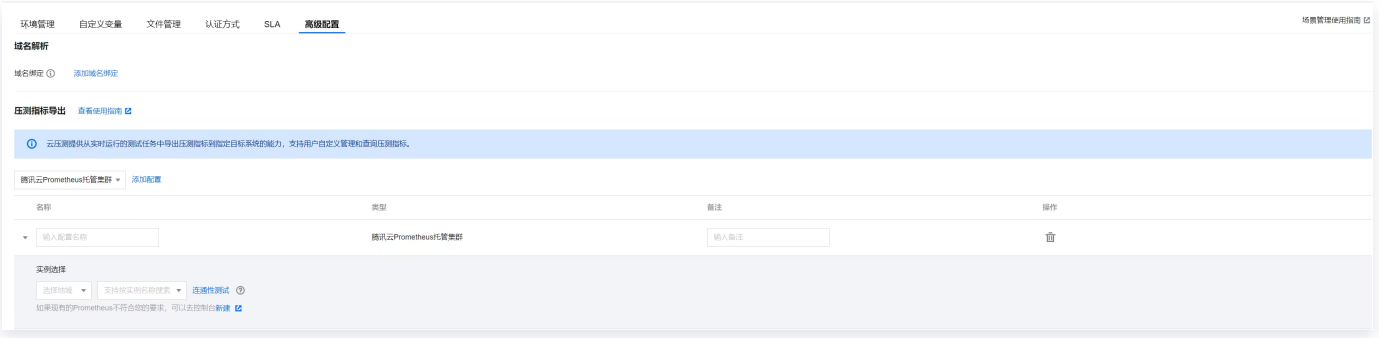
最近更新时间：2024-11-08 15:33:22

云压测提供从实时运行的测试任务中导出压测指标到指定系统的能力，支持用户自定义管理和查询压测指标的需求。
支持导出的压测指标文档： 压测指标介绍 。

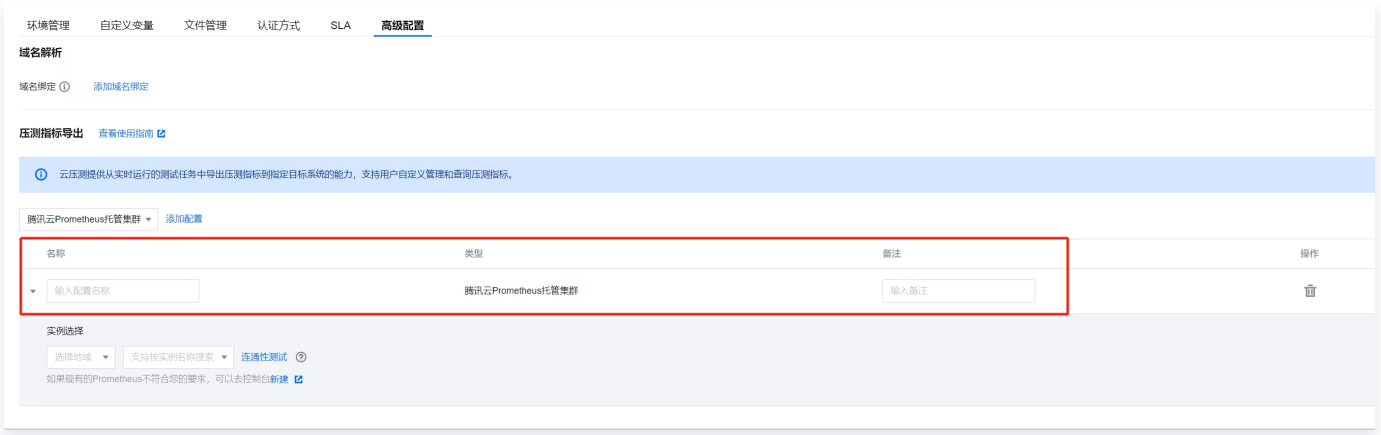
压测指标导出

1. 登录 云压测控制台 ，选择**测试场景**，创建压测场景并单击**高级配置**菜单，可以看到**压测指标导出**的配置选项。

2. 选择压测指标导出的目标系统类型并单击**添加配置**，目前，云压测支持将压测指标导出到**腾讯云 Prometheus 托管集群**。若要查看账号已有的 Prometheus 托管集群，请 前往控制台 查看。

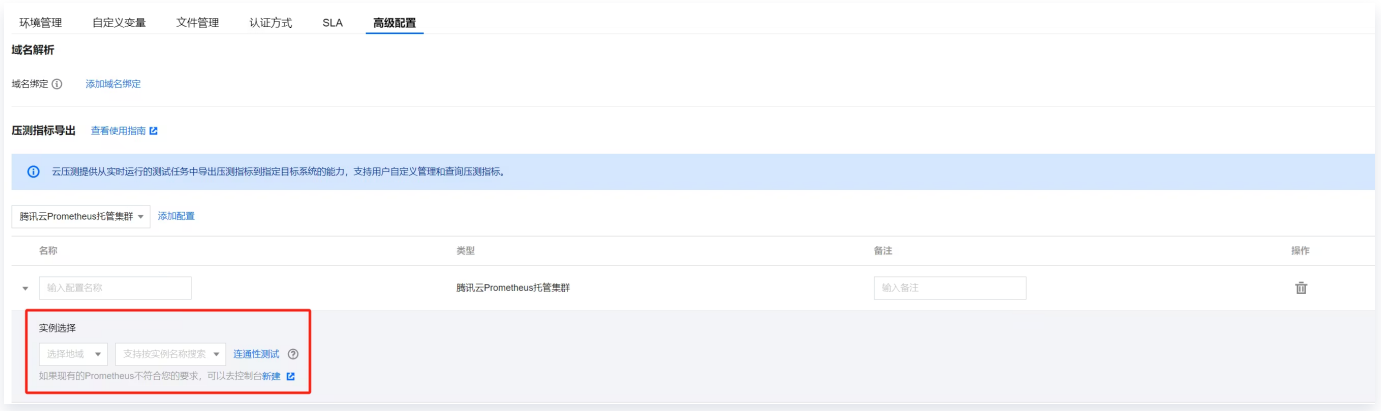


3. 进行配置基础信息的填写，其中**配置名称**是必填项、配置备注是选填项。



4. 选择 Prometheus 实例，指定地域后，选择压测指标导出的 Prometheus 实例。

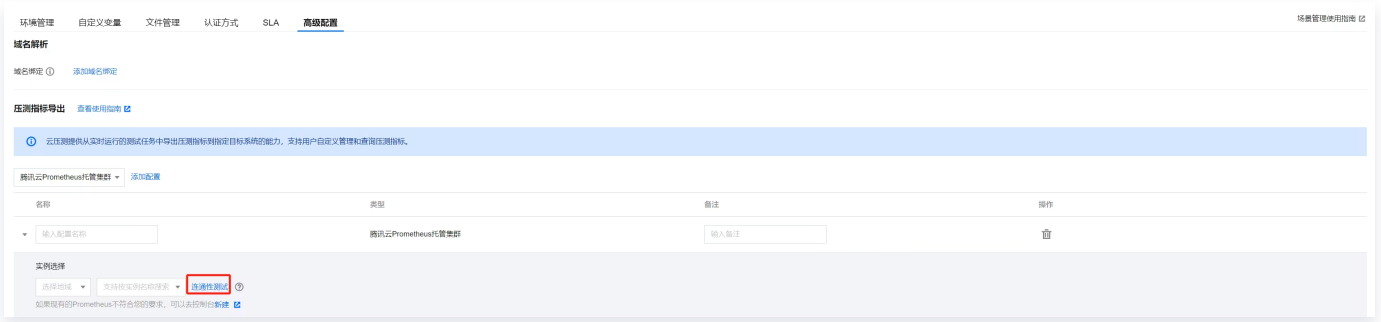
   若现有的 Prometheus 实例不符合您的要求，可以单击**新建**跳转链接到 Prometheus 控制台新建实例。



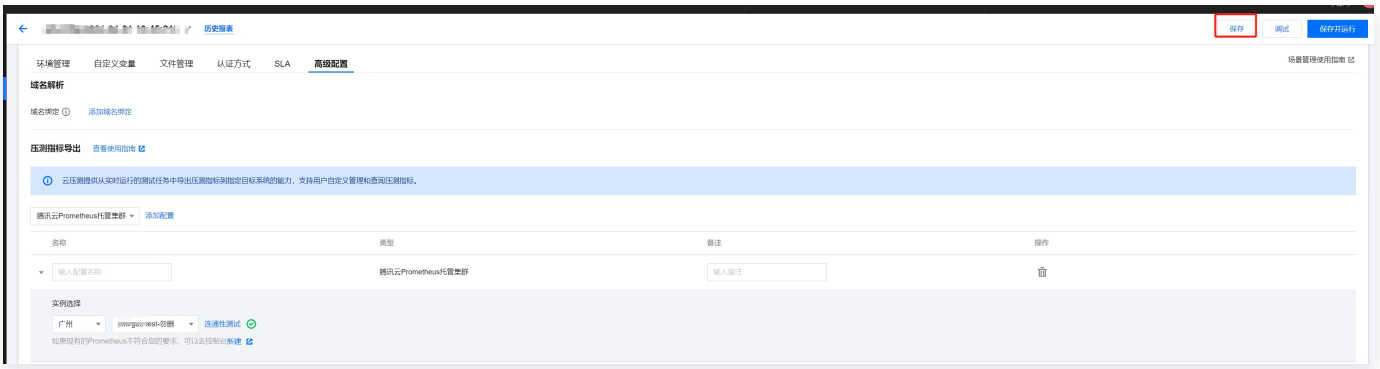
5. 单击**连通性测试**，检测云压测后台能否访问已选定的 Prometheus 实例。

   若显示为绿色的对勾，则说明通过连通性测试；若显示橙色的叹号，则说明云压测后台无法访问到对应的实例。

> **注意：**
>
> “连通性测试”仅对当前的实例状态进行检测，在未来的压测任务执行的过程中，若实例出现故障或销毁等情况导致云压测后台无法访问，则也无法将压测指标导出到对应实例中。



6. 单击**保存**即可保存场景，在任务运行时就可以将压测指标导入到配置的 Prometheus 中；目前云压测支持导出的**压测指标详情请参见** 压测指标介绍 。



压测指标查看

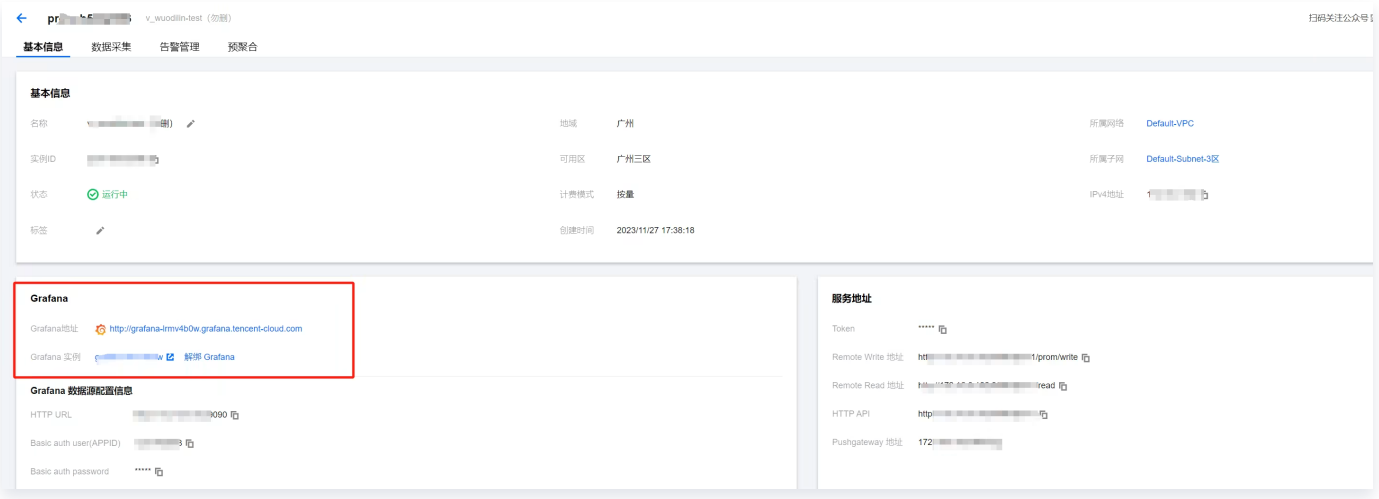
云压测将压测指标导出到指定的 Prometheus 实例中，支持用户自定义地查询和管理压测指标数据。

在 Prometheus 中，常用的数据查看方法包括，通过 API 请求或 Grafana 中添加数据源并查看数据，请参见 监控数据查询 。

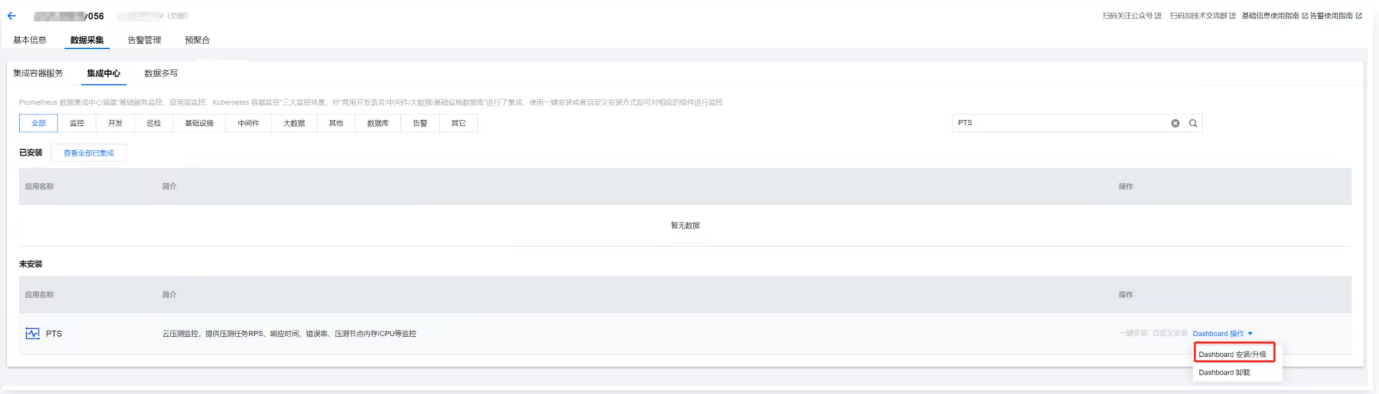
以下主要介绍如何在 **Grafana** 中查看云压测导出的压测指标数据：

1. 进入 Prometheus 实例的“基本信息”页面，确认是否已绑定可用的 Grafana 实例。

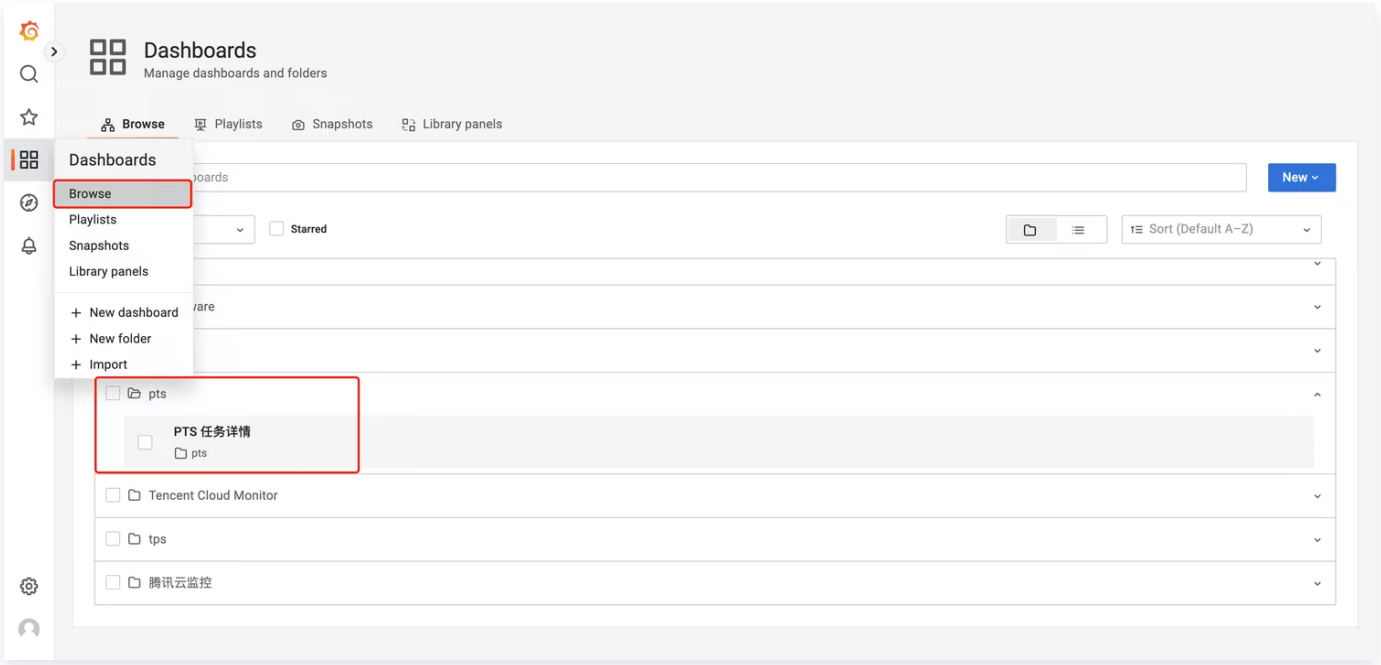
   Prometheus 监控服务与 Grafana 服务高度集成，Prometheus 绑定 Grafana 的方式请参见 Grafana 服务 。



2. 在 Prometheus 实例的**集成中心**页面，搜索“PTS”找到 **PTS 集成**，单击 **Dashboard 操作** > **Dashboard 安装/升级**并单击确定，可以在 Grafana 中自动安装云压测 Dashboard。

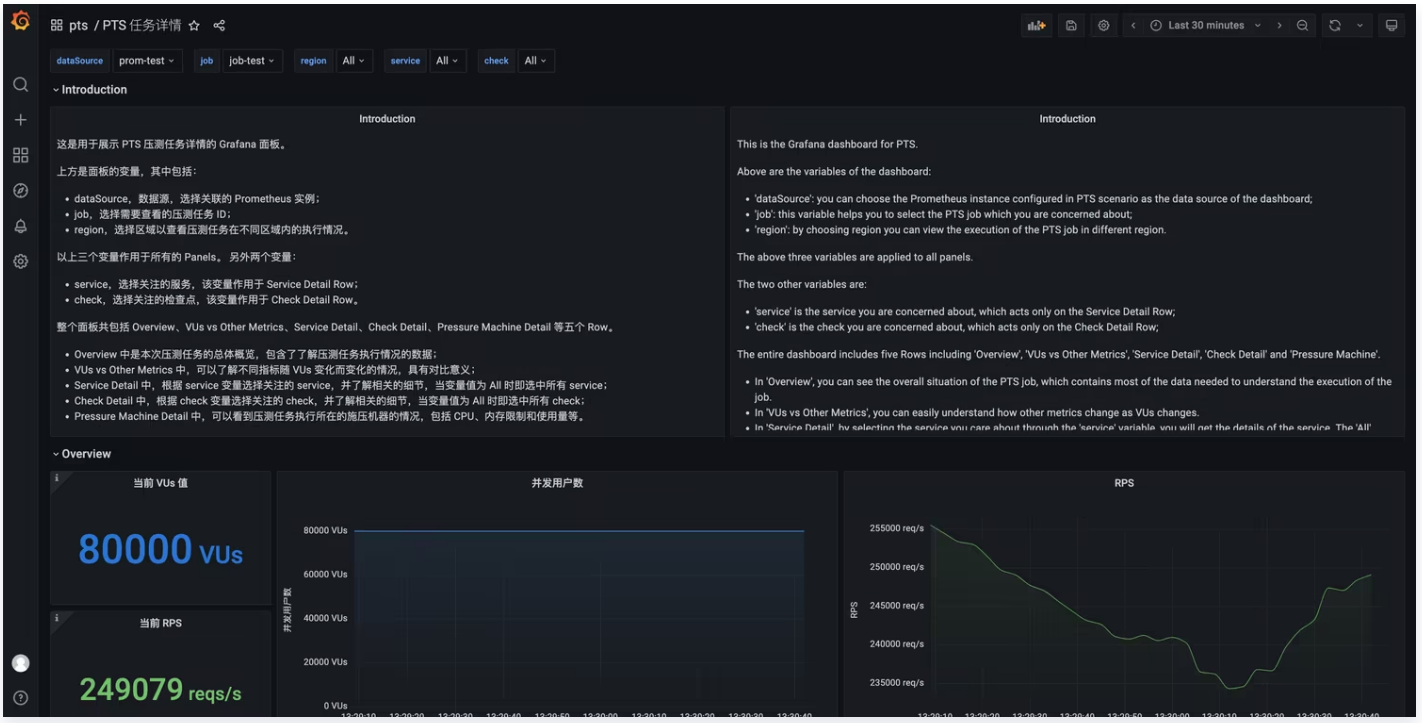


3. 在 Prometheus 实例的“基本信息”页面，单击绑定的 Grafana 并登录进入 Grafana 界面，进入“Dashboards”–“Browse”–“pts”–“PTS 任务详情”即可看到安装的云压测 Dashboard。

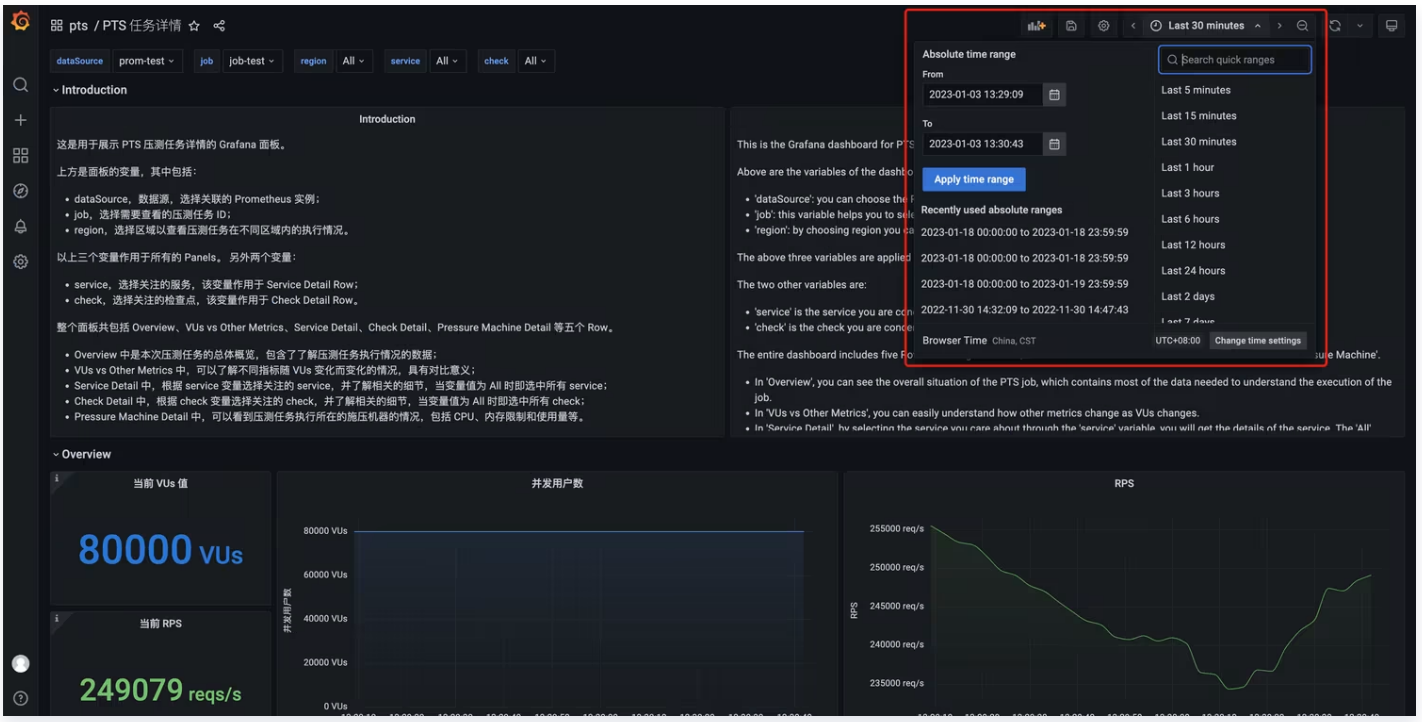


监控面板使用

面板概览图：



其中，右上方的时间栏代表了**查询数据的时间范围**，通过下拉填写能够改变面板展示不同时间范围的数据。



时间的变化也可以在面板里面选择，如图，在需要的时间开始处**按住并拖动**即可：

界面的正上方是该 Dashboard 包含的**变量**，包括：

- **dataSource**，数据源，可以选择关联的监控指标导出的 Prometheus 实例。
- **job**，选择需要查看的压测任务 ID。
- **region**，选择区域以查看压测任务在不同区域内的执行情况。

以上三个变量作用于后面所有的图表。

另外两个变量是：

- **service**，选择关注的服务，该变量作用于 Service Detail Row 栏。
- **check**，选择关注的检查点，该变量作用于 Check Detail Row 栏。

整个面板共包括 Overview、VUs vs Other Metrics、Service Detail、Check Detail、Pressure Machine Detail 等五个栏。

- **Overview** 中是压测任务的总体概览，包含了解压测任务总体执行情况的数据。
- **VUs vs Other Metrics** 中，您可以了解不同指标随 VUs 变化而变化的情况，具有对比意义。
- **Service Detail** 中，根据 service 变量选择关注的 service，并了解相关的细节，当变量值为 All 时即选中所有 service。
- **Check Detail** 中，根据 check 变量选择关注的 check，并了解相关的细节，当变量值为 All 时即选中所有 check。
- **Pressure Machine Detail** 中，可以看到压测任务执行所在的施压机器的情况，包括 CPU、内存限制和使用量等。

图表栏 Overview

Overview 中是压测任务的总体概览，通过该栏能够**大致了解压测任务执行的情况**。

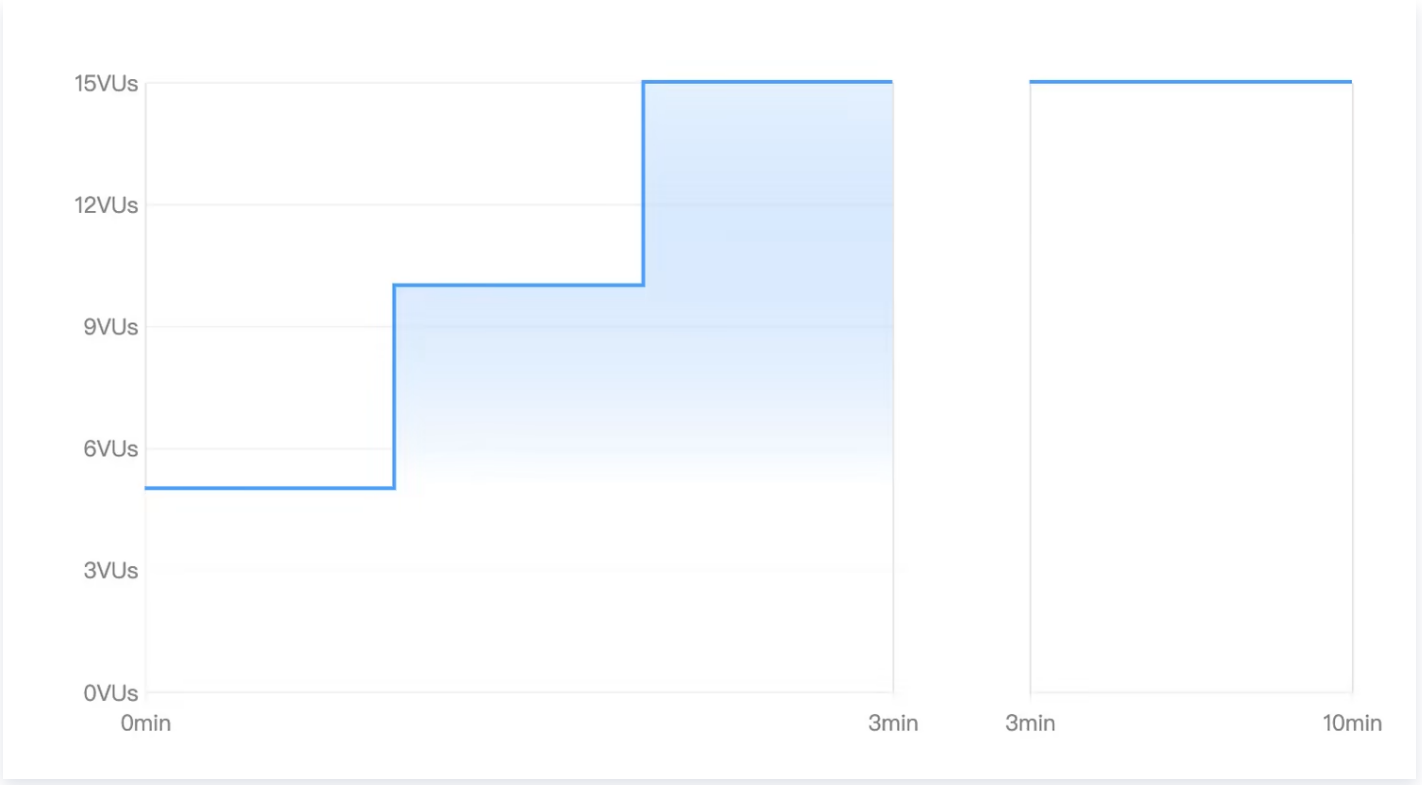
左侧为六个**“瞬时”**数值，代表当前任务运行的状态，右侧为六个时序曲线图表。
分别代表：

- 当前执行压测的 VU 数量及其随时间的变化。
- 当前的 RPS 数值及其随时间的变化。
- 当前执行请求的平均响应时间及其随时间的变化。
- 当前网络流量及其随时间的变化，包括入流量和出流量的加和。
- 当前请求错误率及其随时间的变化。
- 当前检查点未通过率及其随时间的变化。

图表栏 VUs vs Other Metrics

VUs vs Other Metrics 中，可以了解不同指标随 VUs 变化而变化的情况，具有**对比意义**。

当压测场景中配置了 VU 梯度时，如图所示：

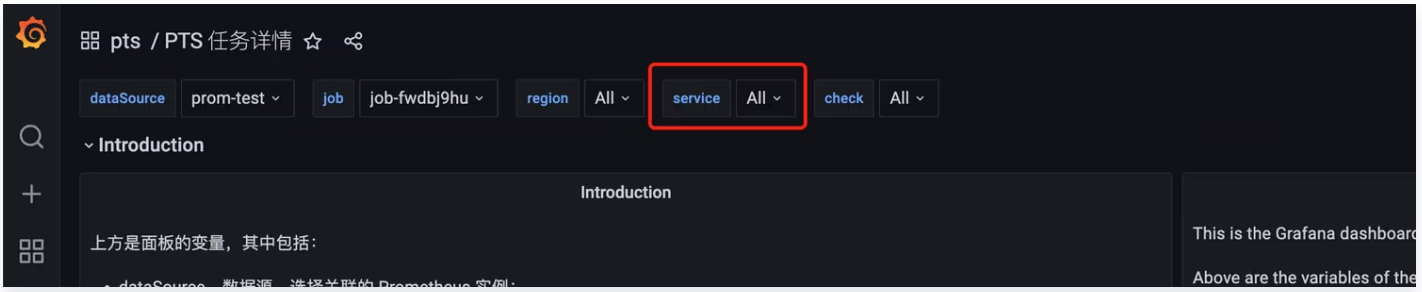


对应的施压力度会随着时间根据不同的梯度进行变化，因此，在不同 VU 数值下观察关键指标的变化具有重要的意义。

在该栏中，包括了 RPS 吞吐量、平均响应时间、请求错误率、检查点未通过率等和 VU 的对比时序曲线图，能够反映随 VU 变化时被压端执行请求的情况。

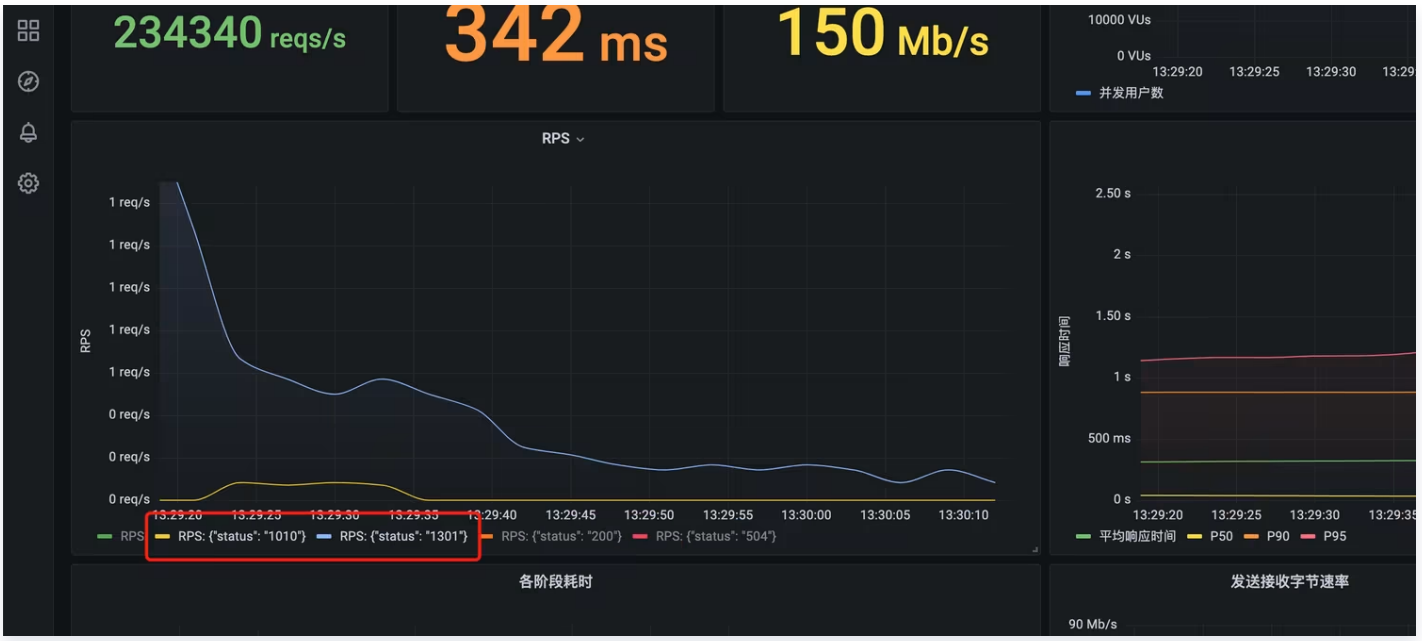
图表栏 Service Detail

Service Detail 中，根据 service 变量选择关注的 service，并了解相关的细节，当**变量值为 All 时即选中所有 service**。



选择关注的 service，获得该 service 的信息，包括当前的瞬时量，例如 RPS、平均响应时间、网络流量等。以及 RPS 和不同状态码、响应时间的不同百分位数、请求不同阶段的耗时、发送和接收的流量和速率等的时序曲线。

以 RPS 为例，如果想看某个状态码的曲线，单击下方图例即可；若要**查看多条曲线**，按住 ctrl 或 shift 再单击多个图例即可：

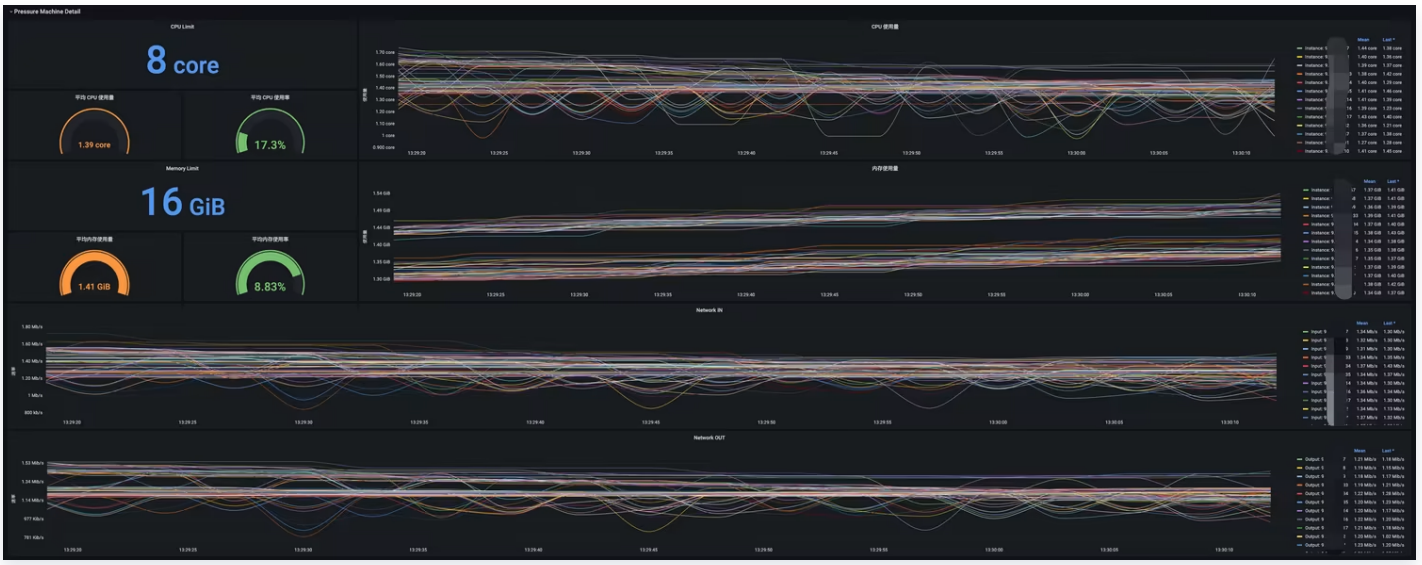


图表栏 Check Detail

Check Detail 中，根据 check 变量选择关注的 check，并了解相关的细节，当**变量值为 All 时即选中所有 check。**

该栏中，包括了检查点当前的执行次数、执行速率、未通过次数、未通过速率和未通过率，以及其随时间变化的情况等信息。

图表栏 Pressure Machine Detail



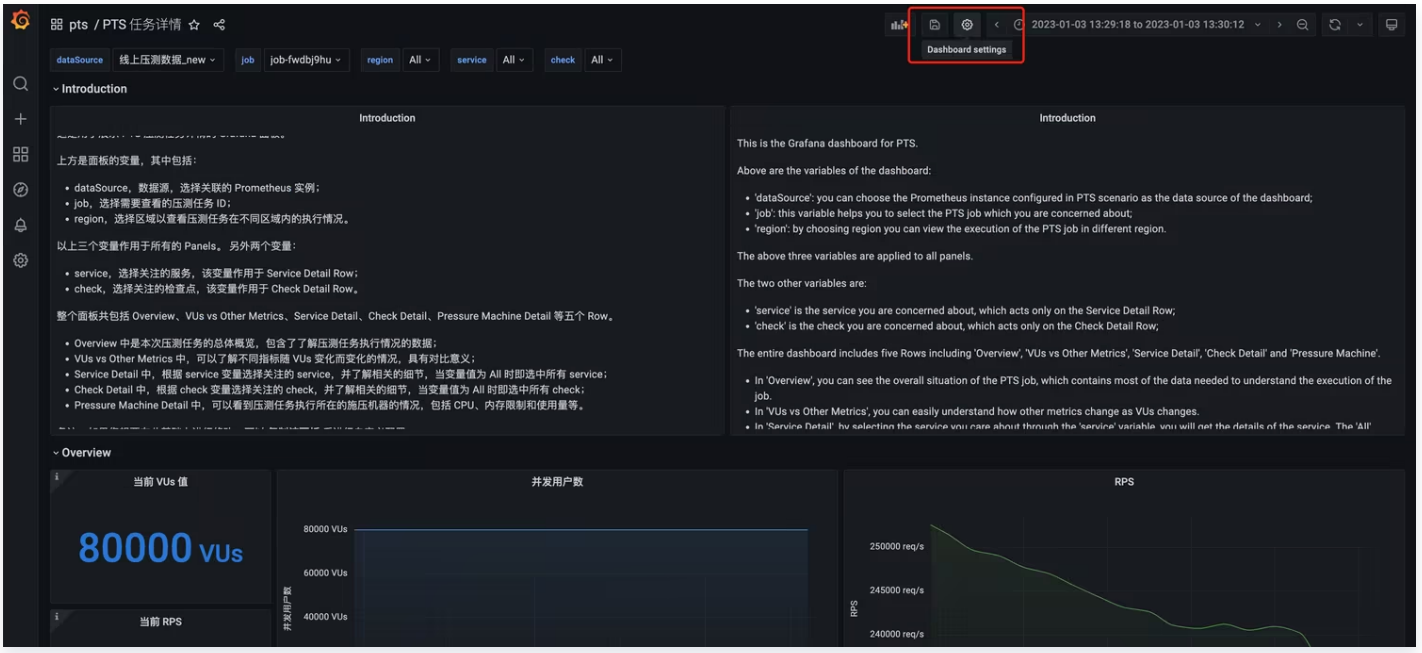
Pressure Machine Detail 中，可以看到压测任务执行所在的施压机器的情况，包括 CPU、内存限制和使用量等。

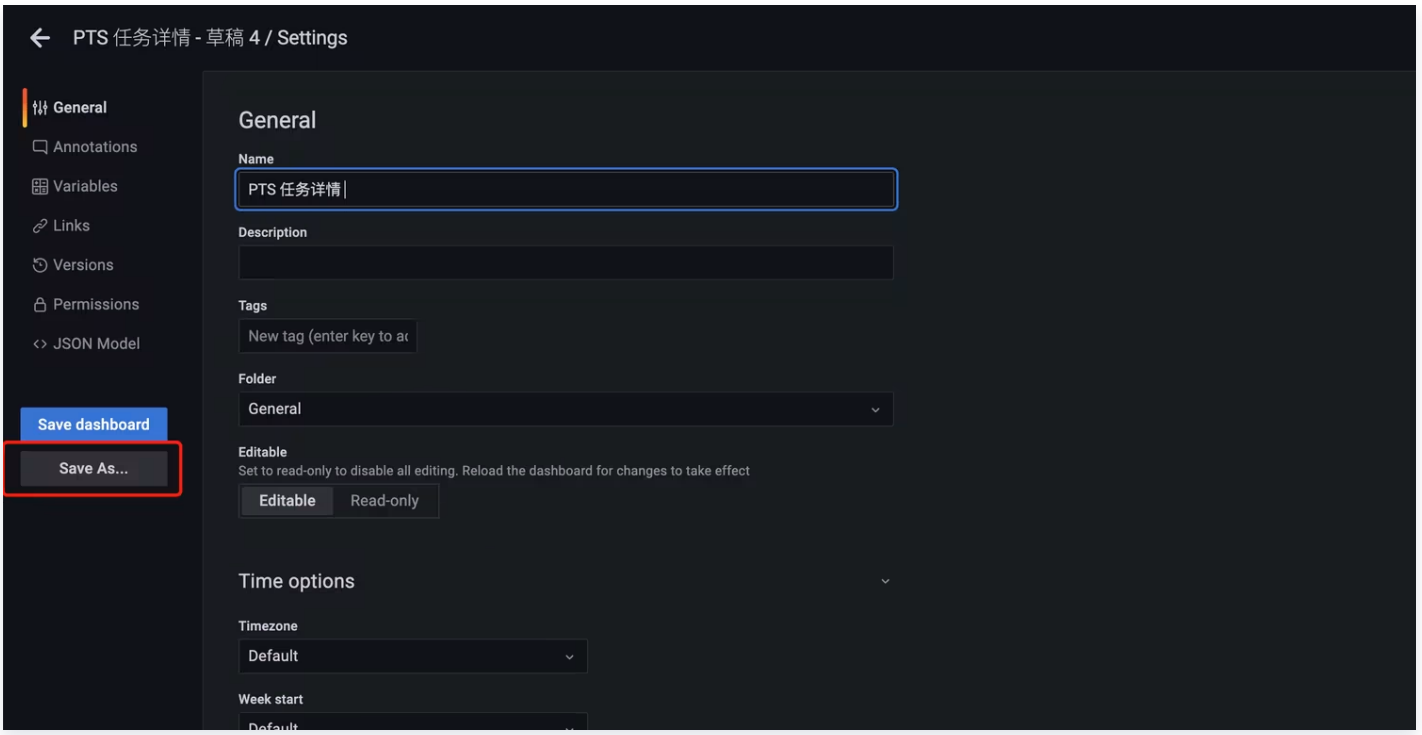
根据用户在压测场景中的配置，云压测会拉起不同配置和数量的 Pod 进行压测任务执行，通过施压机监控可以查看任务运行过程中 CPU 或内存的情况。

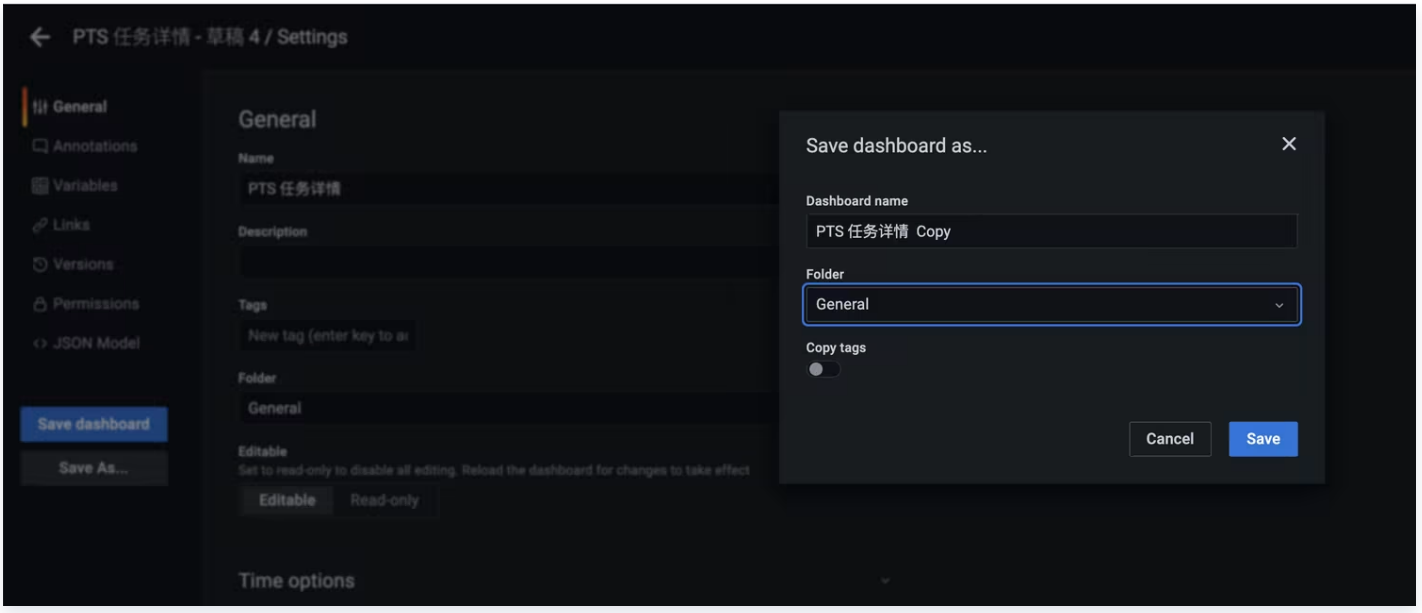
备注：

如果您想要在此面板的基础上进行修改，可以**复制该面板**后进行自定义配置。

单击**设置**符号，并单击 **Save As...**，设置并保存到想要的名称和目录，单击 **Save** 即可。





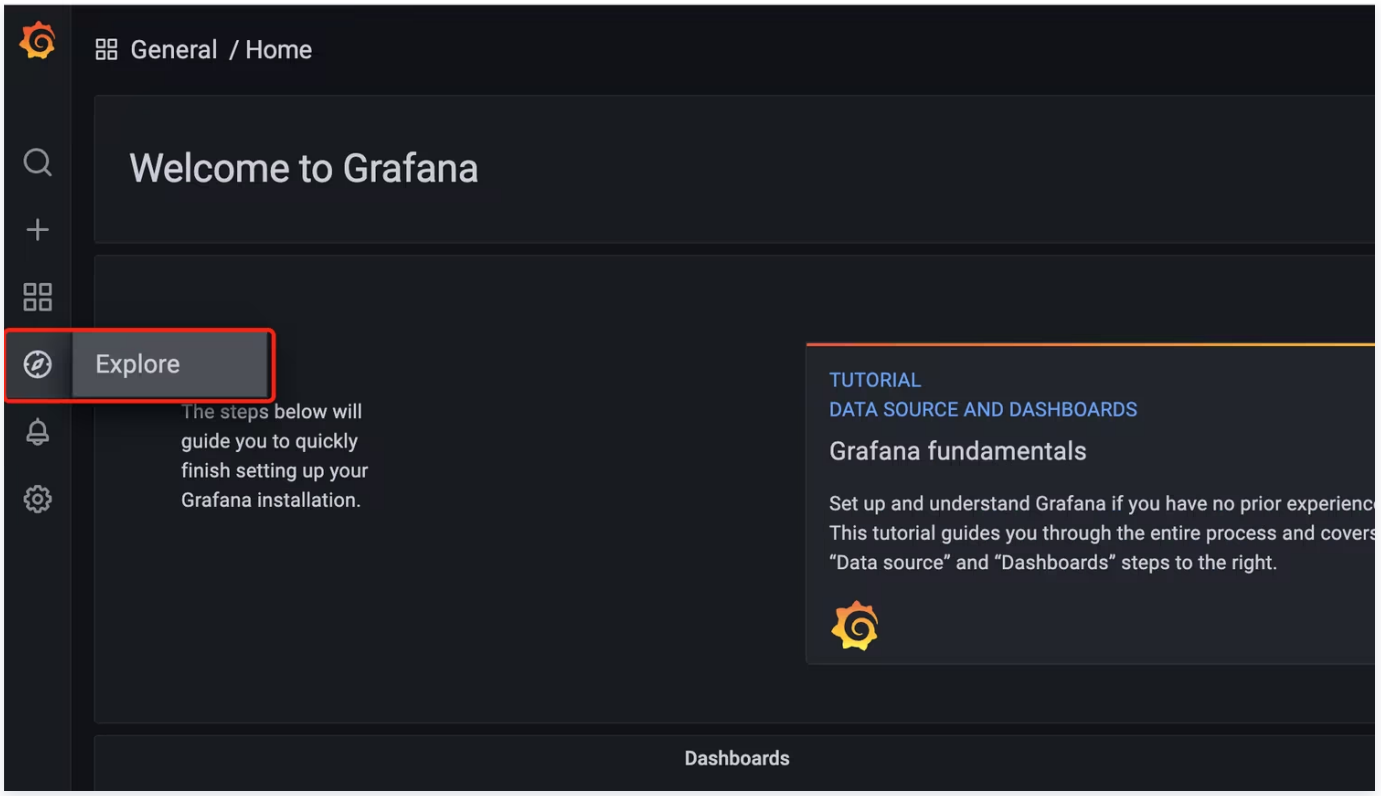


之后，Grafana 会跳转到复制后的 Dashboard 界面，该 Dashboard 即可以由用户自定义修改和删除。

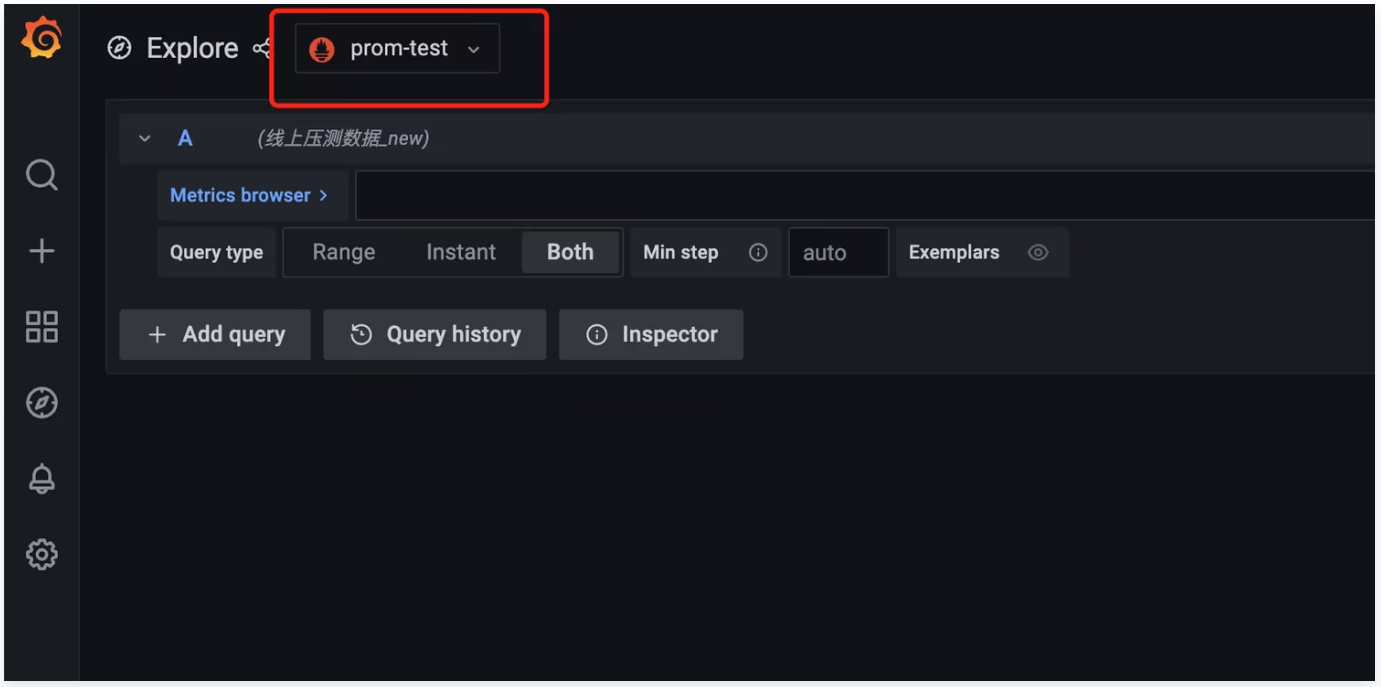
压测指标查询

除了 Dashboard，Grafana 还支持通过 PromQL 语句进行临时的指标查询，通过自定义 PromQL 语句，来查询聚合出丰富的业务指标。

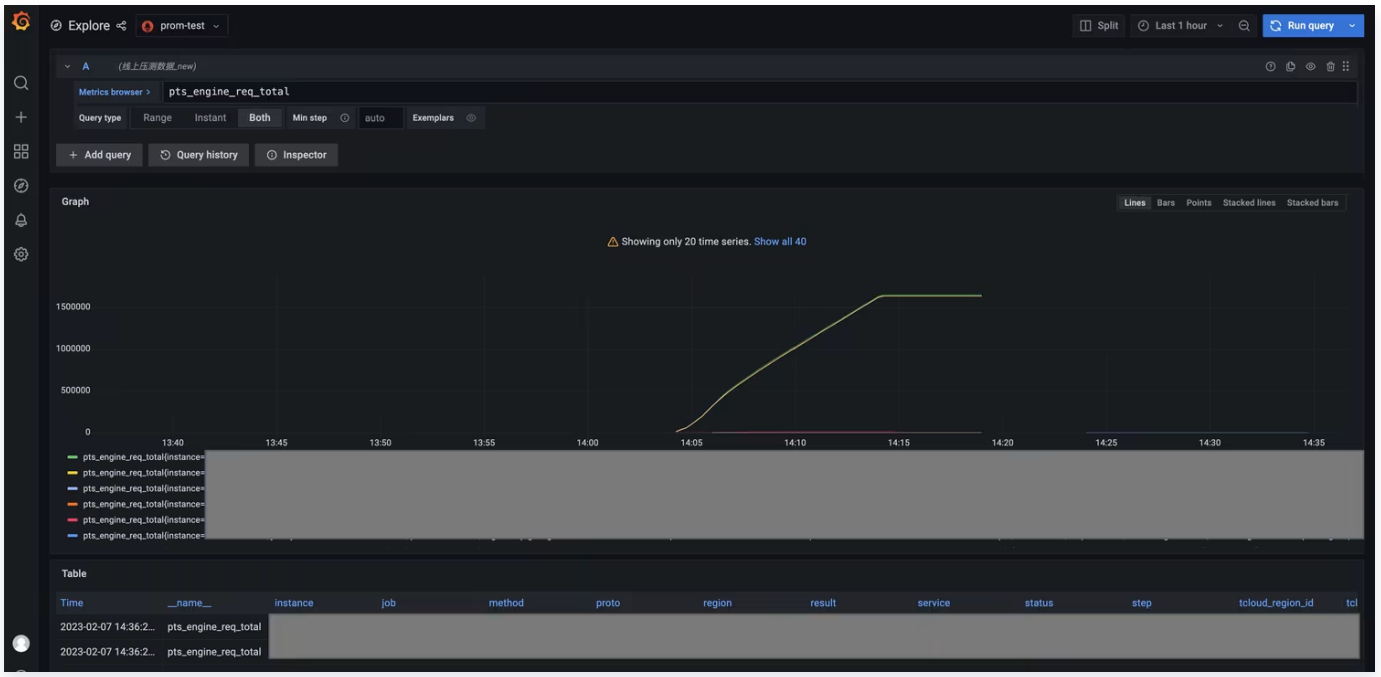
1. 在 Grafana 中单击 Explore。



选择监控指标导入的 Prometheus 实例。



2. 输入想要查询的 PromQL 模式和语句即可，具体请参见 Grafana 官方文档 。



压测指标文档

最近更新时间：2024-11-08 15:33:22

本文档介绍 PTS 支持输出到 Prometheus 的压测指标，供用户接入时使用参考。

标签

默认标签

PTS 输出的所有压测指标均包含以下三个标签：

- instance：施压机的 IP 地址。
- job：PTS 压测任务 ID，如 job-xxx。
- region：压测任务运行所在的地域，如 ap-guangzhou。

其他标签

部分压测指标包含的其他标签：

- method：请求方法名称，以 HTTP 协议为例，method 为 GET、POST、PUT 等。
- proto：协议名称，以 HTTP 协议为例，proto 为 HTTP/1.1、HTTP/2 等。
- service：服务名，以 HTTP 协议为例，service 为请求 url，如 http://httpbin.org/get 等。
- status：响应状态码，以 HTTP 协议为例，状态码包括 200、404、500 等。
- result：响应详情，通过 result 判断请求成果或失败。
  - 请求正常，result 标签值为 ok。
  - 请求失败，result 标签携带错误码和描述。
  - 详细的错误码手册：错误代码手册 。
- check：检查名，标签值为用户设置的检查点名称。

> **注意：**
> 在 websocket 中，event 对应 method 标签，包括：
> - 上行消息：sendPing、sendPong、sendMessage、sendBinaryMessage。
> - 下行消息：ping、pong、message、binaryMessage、error、open、close。

指标

metric name	type	labels	help info	description
pts_engine_req_total	counter	method, proto,	Total number of	请求总次数

		service, status, result, instance, job, region	requests sent to server	
pts_engine_req_duration_seconds	histogram	method, proto, service, status, result, instance, job, region	Duration of request(second)	每次请求耗时
pts_engine_req_max_duration_seconds	gauge	method, proto, service, status, result, instance, job, region	Max duration of request(second)	请求最大耗时
pts_engine_req_min_duration_seconds	gauge	method, proto, service, status, result, instance, job, region	Min duration of request(second)	请求最小耗时
pts_engine_req_send_duration_seconds	histogram	method, proto, service, status, result, instance, job, region	Duration of request sending(second)	发送请求耗时
pts_engine_req_wait_duration_seconds	histogram	method, proto, service, status, result, instance, job, region	Duration of request waiting(second)	读取第一个响应字节耗时

pts_engine_req_receive_duration_seconds	histogram	method, proto, service, status, result, instance, job, region	Duration of request receiving(second)	读取完整响应耗时
pts_engine_req_block_duration_seconds	histogram	method, proto, service, status, result, instance, job, region	Duration of request blocked(second)	发起请求之前被阻塞耗时
pts_engine_req_connect_duration_seconds	histogram	method, proto, service, status, result, instance, job, region	Duration of request connecting(second)	与远程主机建立连接耗时
pts_engine_req_tls_handshake_duration_seconds	histogram	method, proto, service, status, result, instance, job, region	Duration of request tls handshaking(second)	与远程主机握手耗时
pts_engine_req_dns_lookup_duration_seconds	histogram	method, proto, service, status, result, instance, job, region	Duration of request dns lookup(second)	DNS 寻址耗时
pts_engine_send_bytes_total	counter	method, proto, service, status, result,	Total number of bytes sent	发送字节数

		instance, job, region		
pts_engine_receive_bytes_total	counter	method, proto, service, status, result, instance, job, region	Total number of bytes received	接收字节数
pts_engine_checks_total	counter	check, result, instance, job, region	Total number of checks in requests	检查点执行总次数
container_cpu_usage_seconds_total	counter	instance, job, region	Cumulative cpu time consumed	累计 CPU 占用时间
container_memory_usage_bytes	gauge	instance, job, region	Current memory usage, including all memory regardless of when it was accessed	当前内存使用量
container_network_receive_bytes_total	counter	instance, job, region	Cumulative count of bytes received	累计接收字节数
container_network_transmit_bytes_total	counter	instance, job, region	Cumulative count of bytes transmitted	累计发送字节数
kube_pod_resource_cpu_limits	gauge	instance, job, region	CPU limit in cores	CPU 核心上限
kube_pod_resource_mem_limits	gauge	instance, job, region	Memory limit in bytes	内存上限

> **注意：**
> histogram 指标的 bucket 为 {0.001, 0.002, 0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1, 2, 5, 10, 25, 50}。

响应数据提取

最近更新时间：2024-06-27 14:06:51

概述

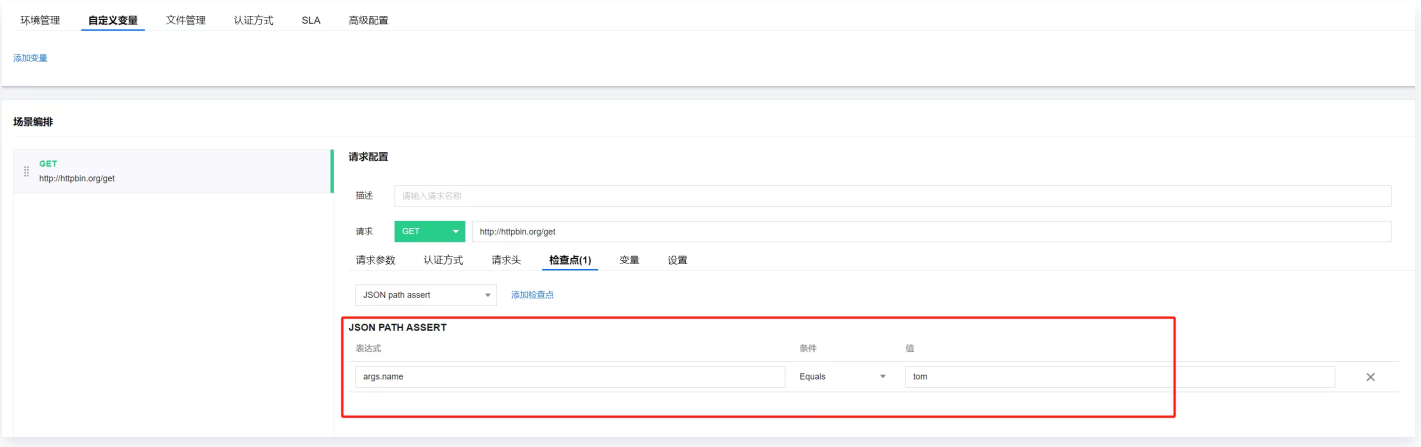
若要从请求的响应中动态提取数据，PTS 除了支持字符串的简单匹配（相等或包含），还支持 Jsonpath 提取器、正则表达式提取器，供您实现更加灵活的提取逻辑。

基本用法

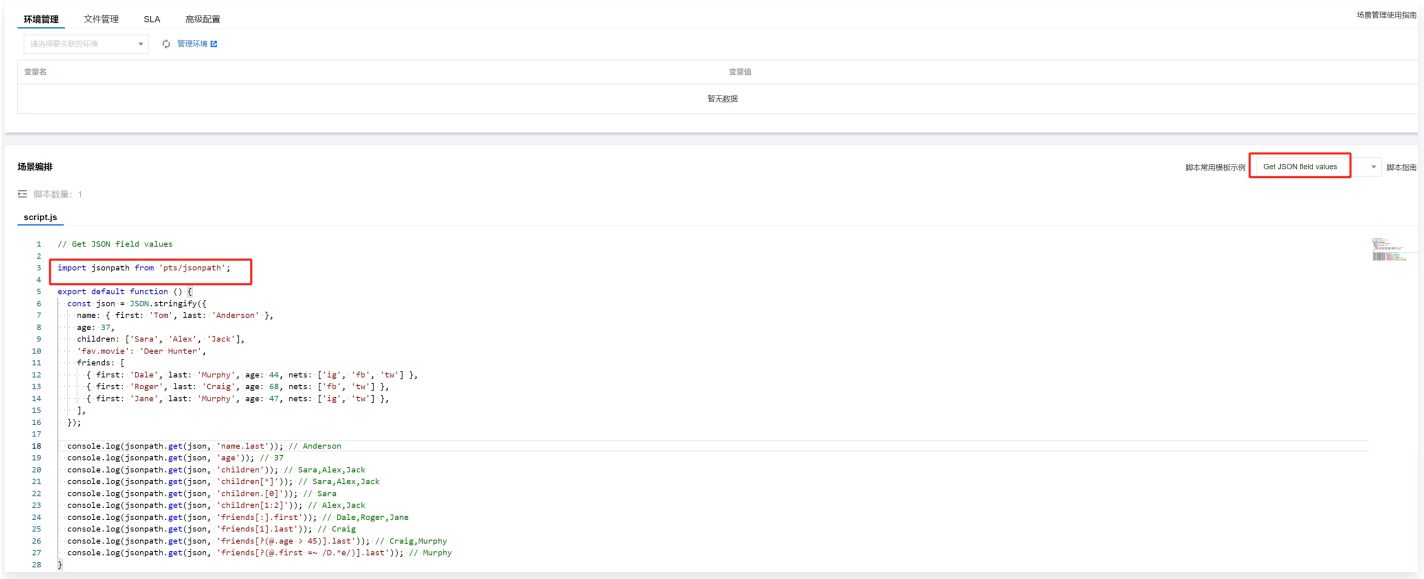
您在 PTS 压测场景中编排或调试请求时，若需从响应中动态提取数据值，用于为当前请求设置检查点、或为后续请求注入参数、或在调试模式下调试具体响应字段，则您可使用数据提取器，解析和提取响应数据中的具体字段。
响应数据提取需在编排场景或者调试场景时使用。如何编排或调试场景，详情请参见 简单模式 、脚本模式 和 调试场景 。

使用场景

简单模式的场景下，您可从列表中选择具体的提取方式，然后在文本框输入符合提取器语法的表达式。以 Jsonpath 提取器为例：



脚本模式的场景下，您可使用 Get JSON field values，在脚本中实现提取数据的逻辑。以 Jsonpath 提取器为例：



提取器类型

Jsonpath 提取器

Jsonpath 提取器适用于从 JSON 类型的响应体中提取数据。Jsonpath 表达式的常用语法：

运算符	描述
@	当前节点
*	通配符，可匹配任意节点名或索引值
.<name>	用 . 匹配下级节点
[<number> (, <number>)]	用 [] 检索数组中的一个或多个元素
[start:end]	数组切片
[?(<expression>)]	用布尔表达式筛选数据

在脚本模式下，Jsonpath 的用法及其结果示例如下：

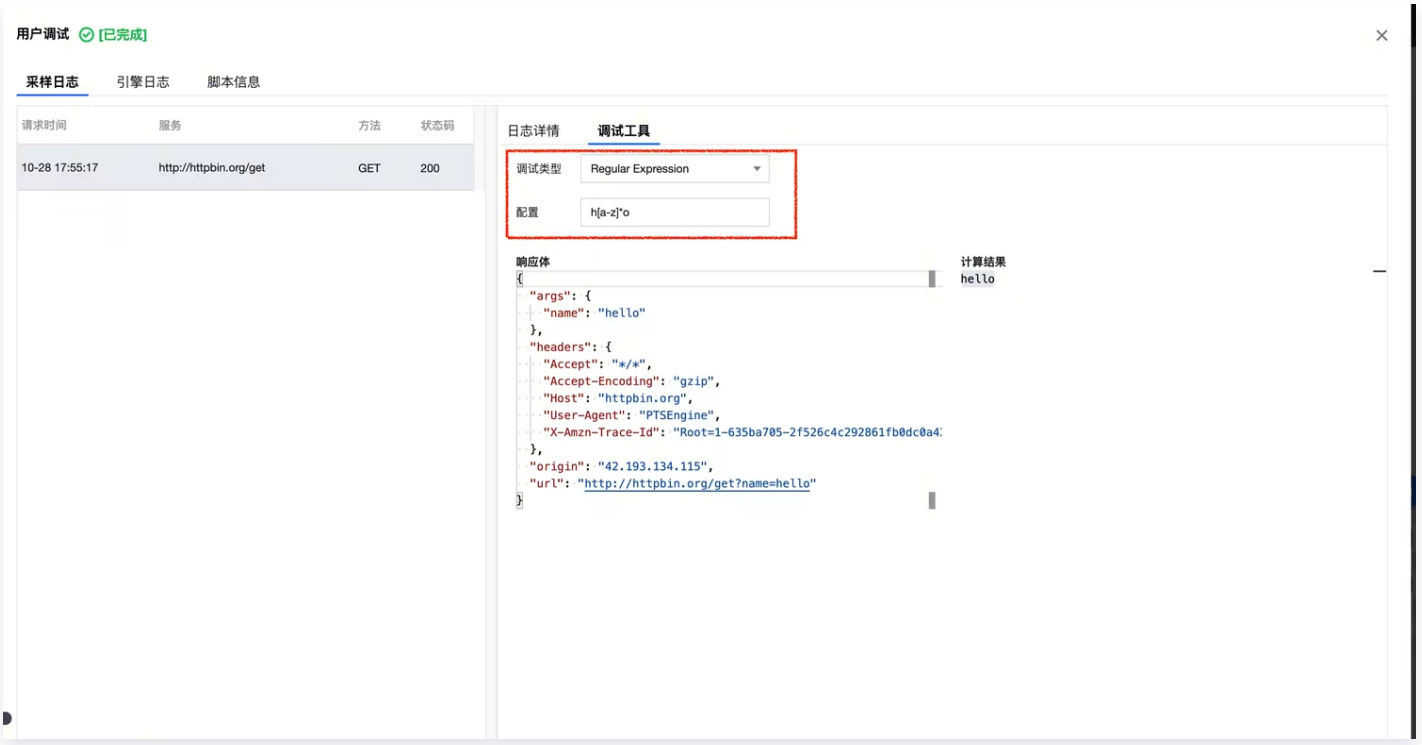
```
import jsonpath from 'pts/jsonpath';
export default function () {
```

```
  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, '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
};
```

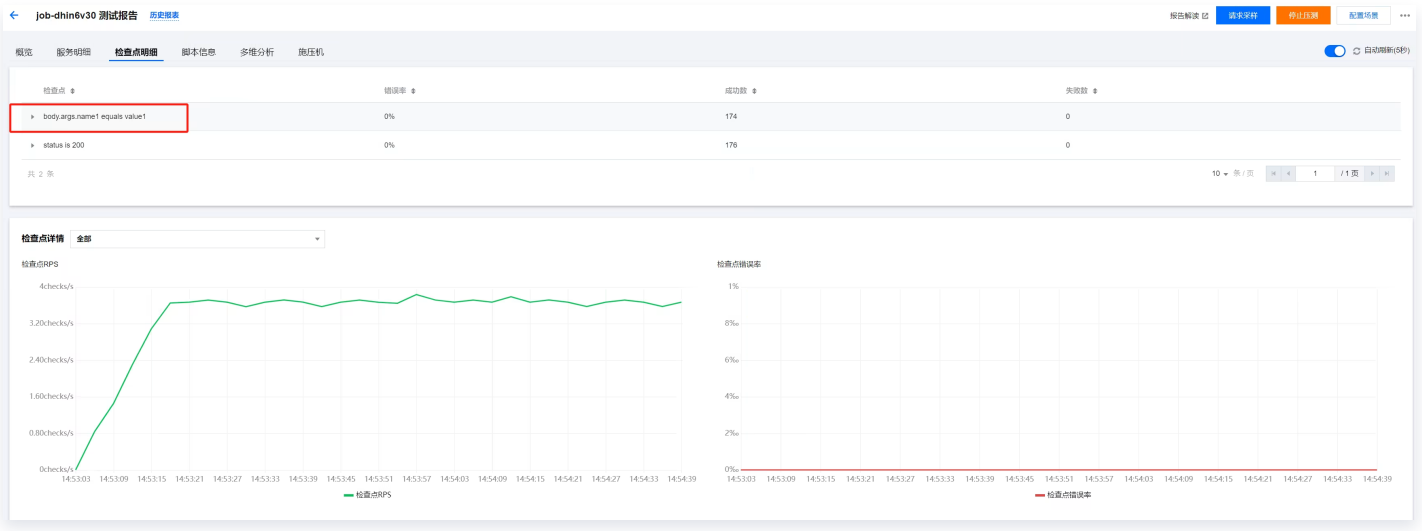
正则表达式提取器

正则表达式提取器适用于从文本类型的响应中提取数据。PTS 正则表达式符合 JavaScript 原生语法，详情请参见 正则表达式 。

以 PTS 场景调试为例，正则表达式提取器用法如下：



以脚本模式场景设置检查点为例，正则表达式提取器用法如下：




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

export default function main() {
  let response;

  response = http.get("http://mockhttpbin.pts.svc.cluster.local/get?
name=hello");
  check("body matches /h[a-z]*o/", () => {
```

```
    const expr = new RegExp("h[a-z]*o");
    return expr.test(response.body);
  });
}
```

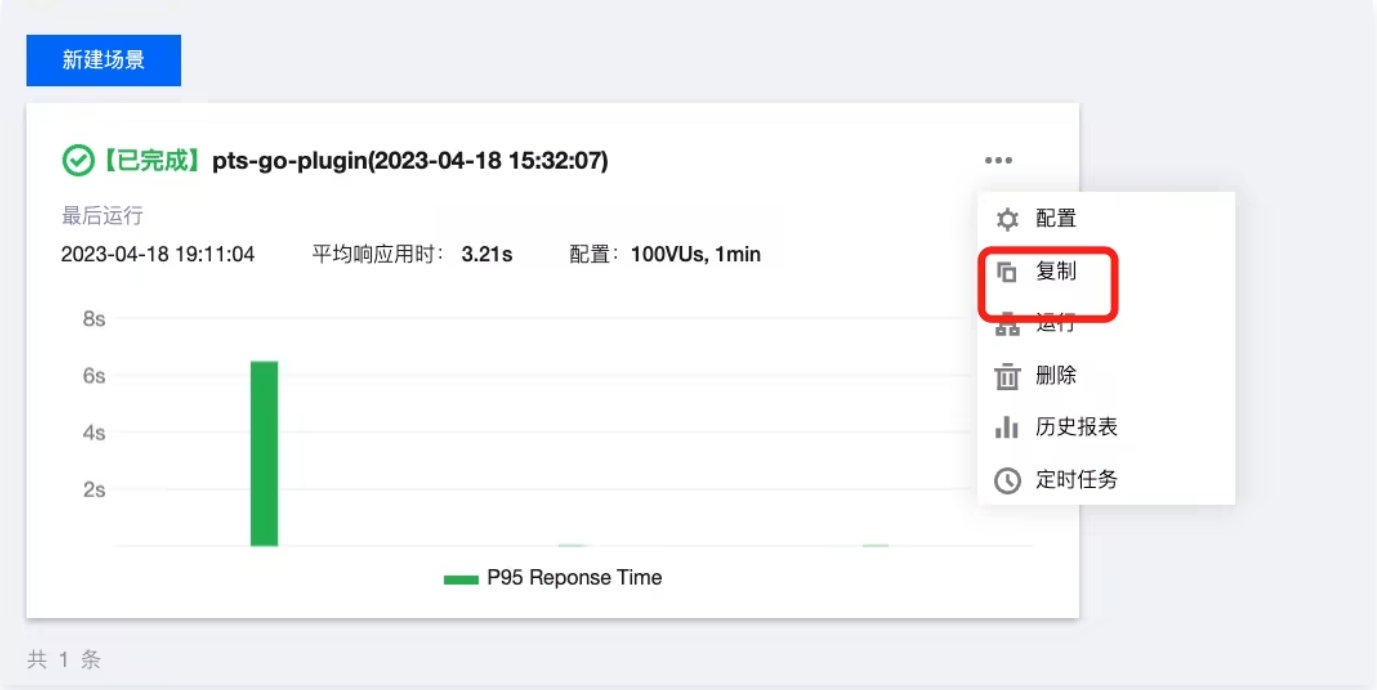
复制场景

最近更新时间：2024-01-10 20:03:21

本文将为您介绍如何在同一个项目下快速复制测试场景。

操作步骤

1. 登录 腾讯云可观测平台 控制台。
2. 在左侧菜单栏中单击**云压测 > 测试场景**。
3. 单击场景名称，进入测试场景详情页，单击 ··· **> 复制**。



4. 等待执行成功后，修改场景名称并单击**确认**（场景名称为必填字段，默认与原场景同名），即可创建出与原场景的配置完全一致的新场景。

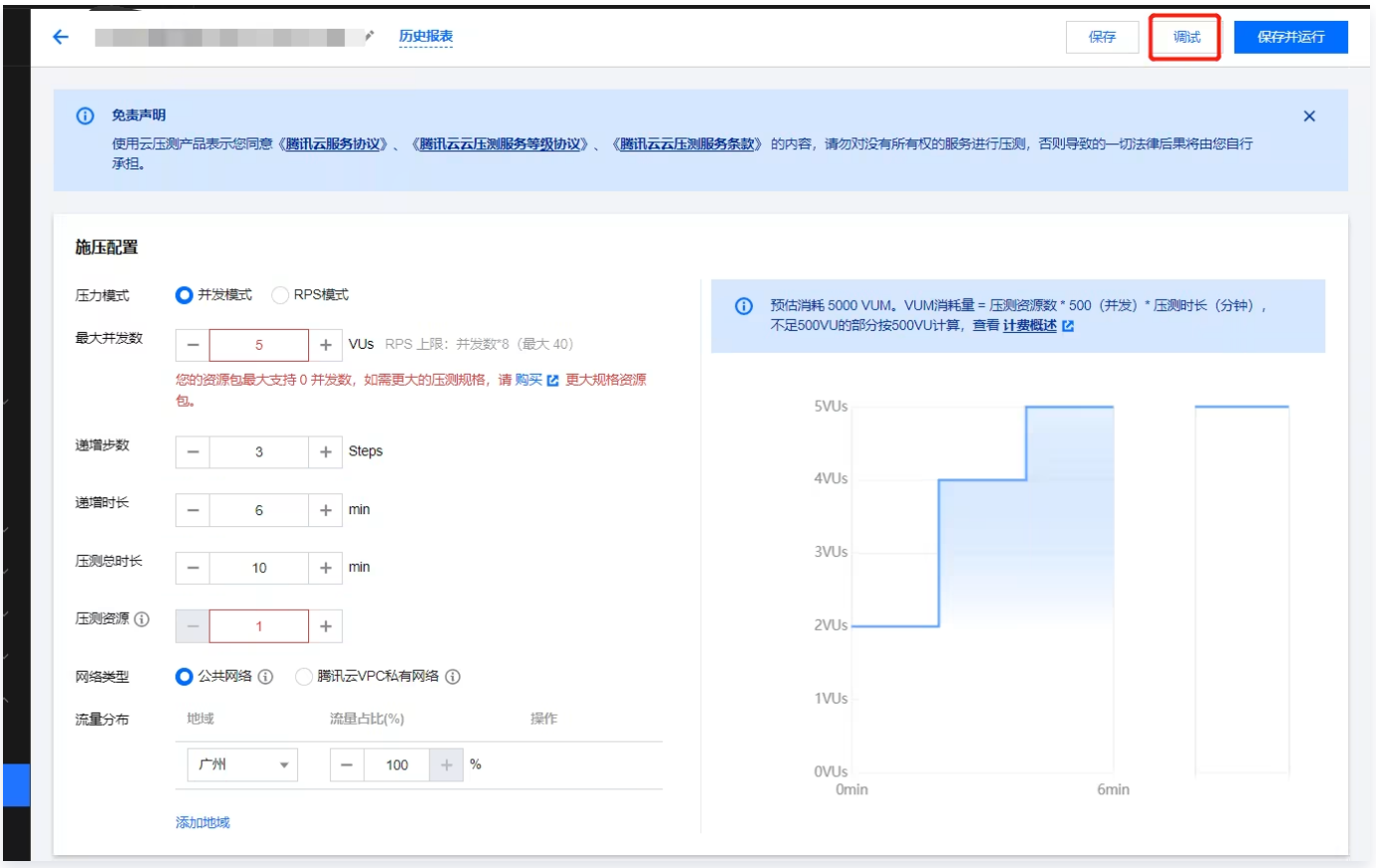
调试场景

最近更新时间：2024-10-31 18:11:52

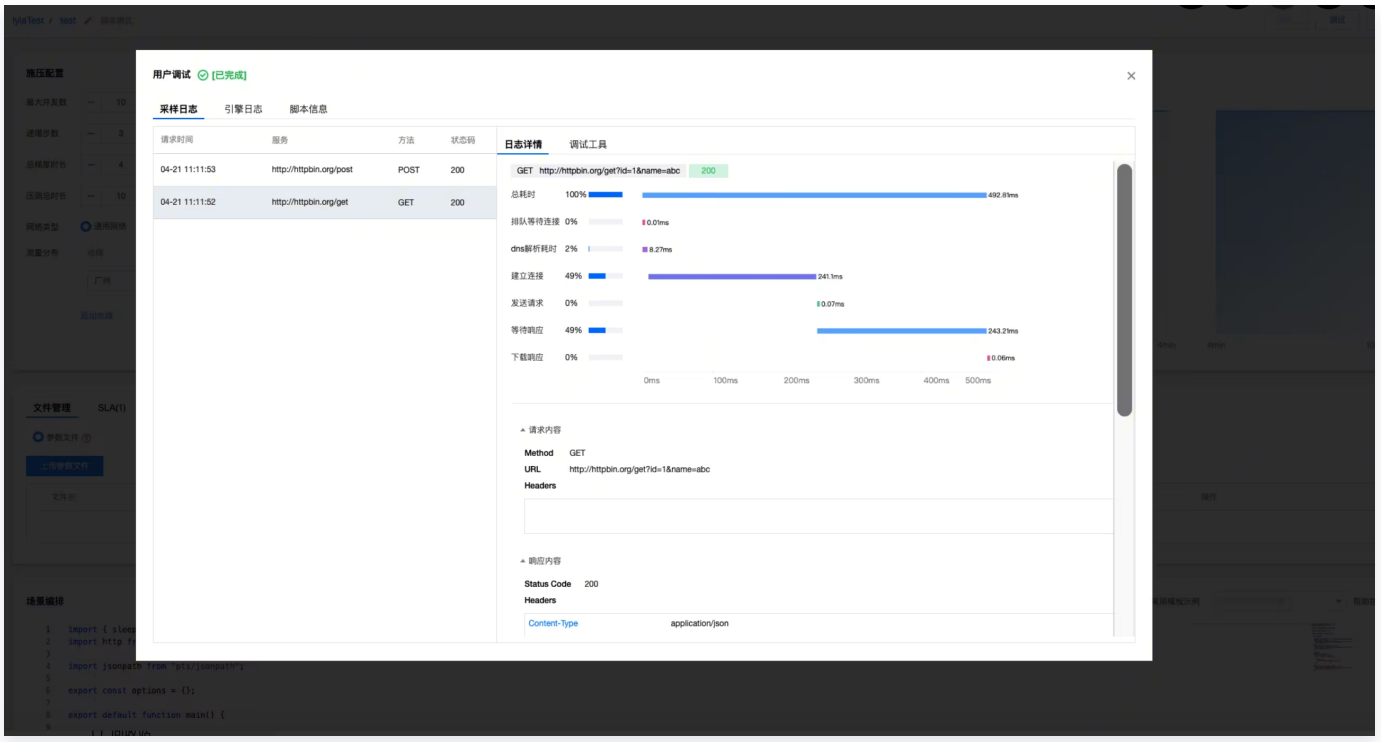
在创建场景之后、正式压测之前，您可以先借助调试模式，快速地校验您的场景、排查和修复错误，以保证正式压测时，您的场景是符合预期的。

操作步骤

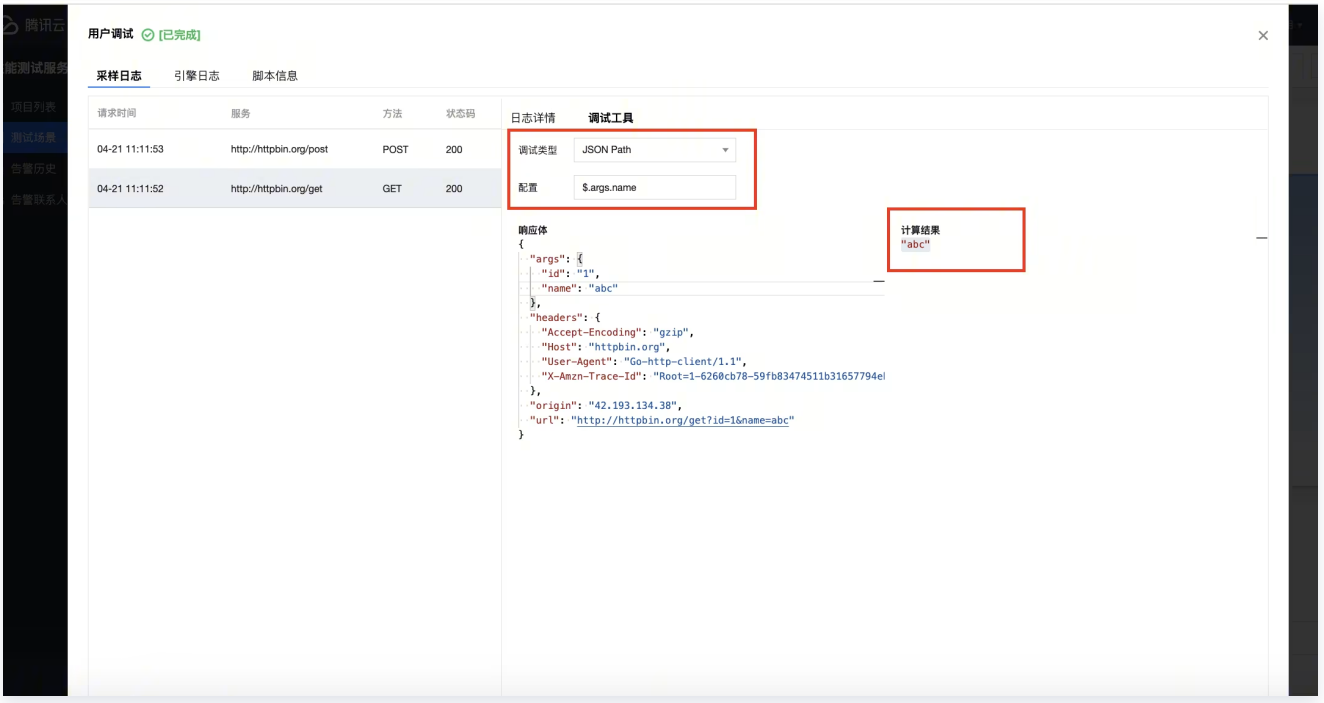
1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**云压测 > 测试场景**。
3. 点击**新建场景**，选择测试场景模式，然后在新建测试场景页面填写基础信息后，点击**调试**，进入调试模式。



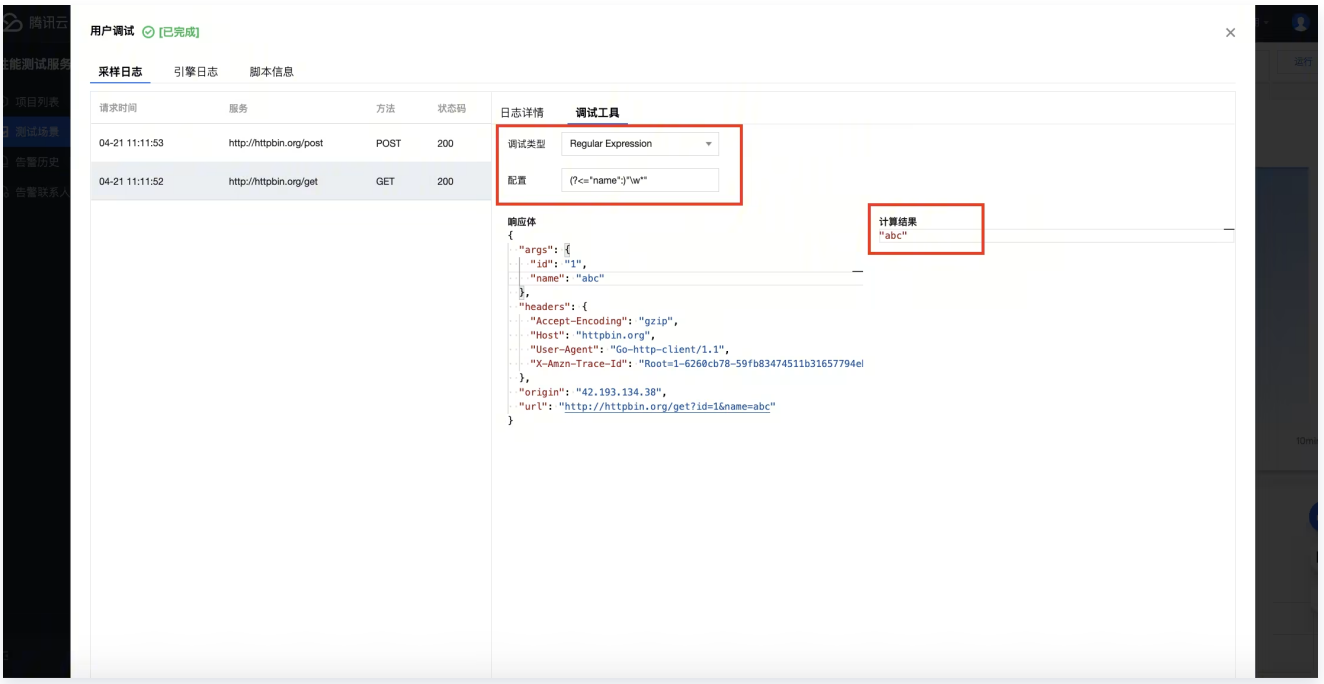
4. 在弹出的调试页面上，您可以查看所有请求的采样日志、引擎日志和脚本信息，并可以使用调试工具，调试分析您的请求数据。



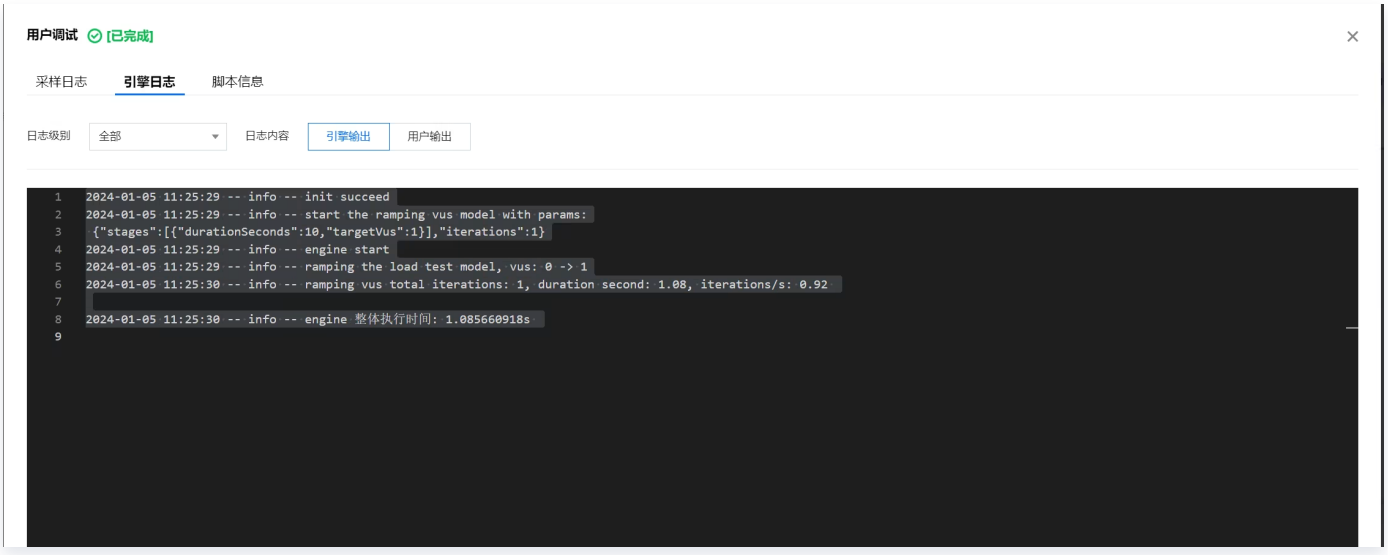
- **在采样日志页，您可以单击左侧请求列表，查看某个请求及其响应的具体信息和耗时瀑布流；还可以单击调试工具标签页，输入 JSON Path 表达式或者正则表达式，从响应结果中提取所需数据。**
  - **使用 JSON Path 提取数据的示例如下：**



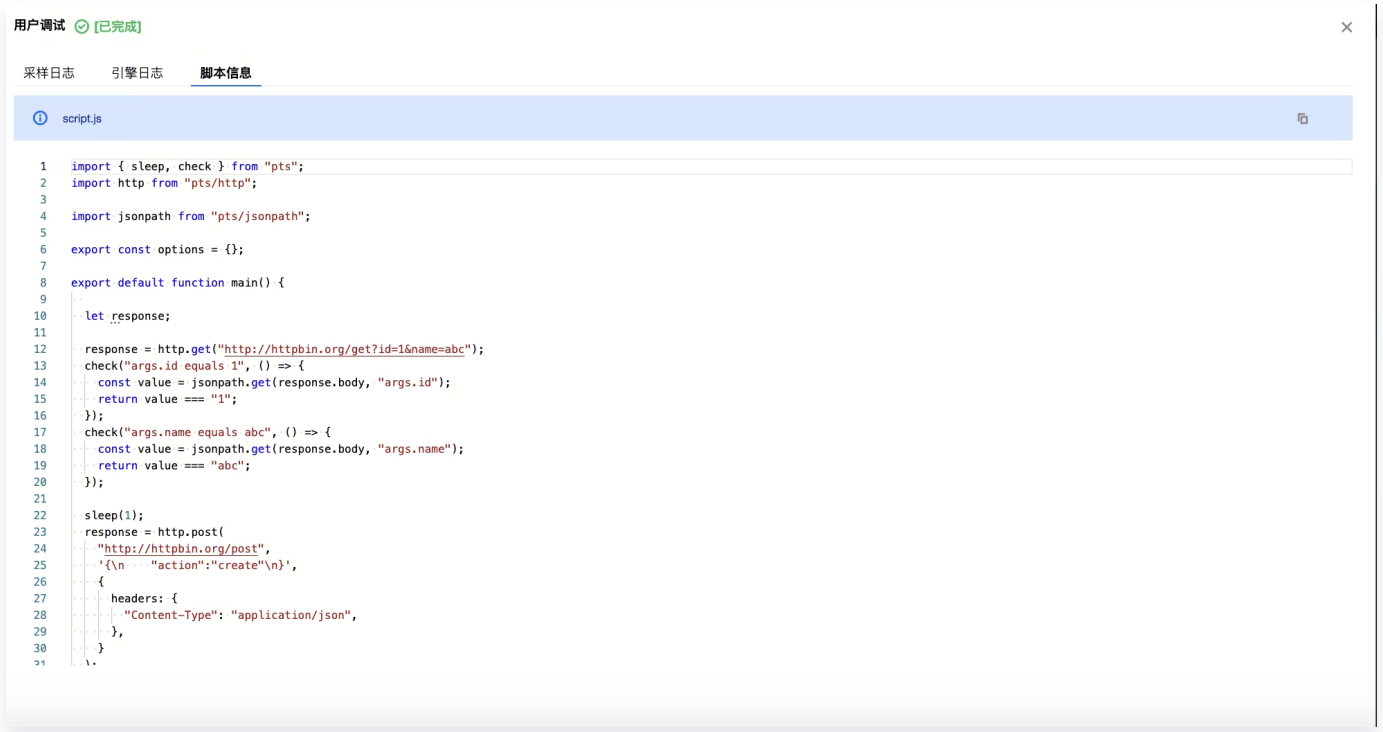
  - **使用正则表达式提取数据的示例如下：**



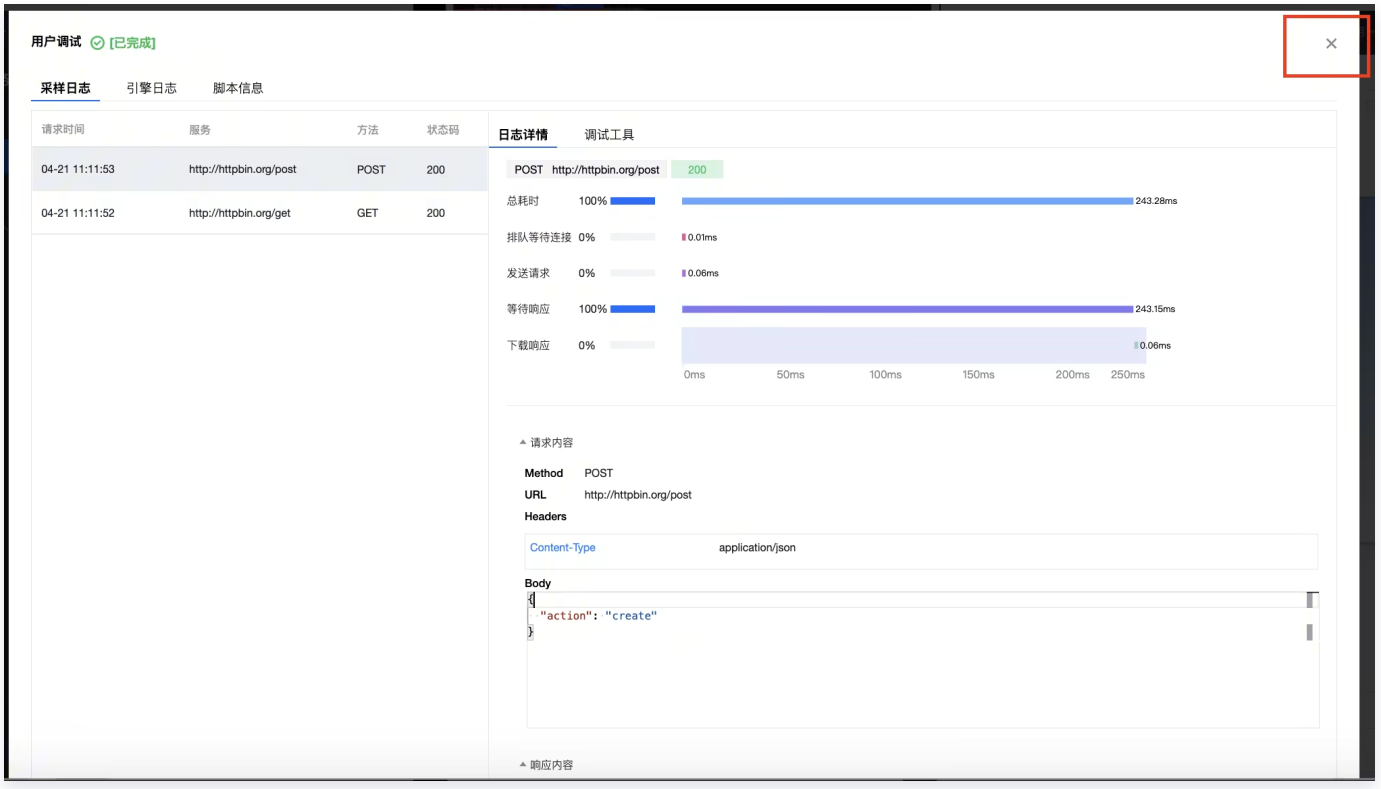
- 在引擎日志页，您可以选择日志级别和日志来源，查看引擎输出的日志：



- 在脚本信息页，您可以浏览本次压测时刻所使用的场景脚本快照：



5. 若要退出调试模式，可单击右上角“关闭”图标，返回场景页。



流量录制
浏览器流量录制

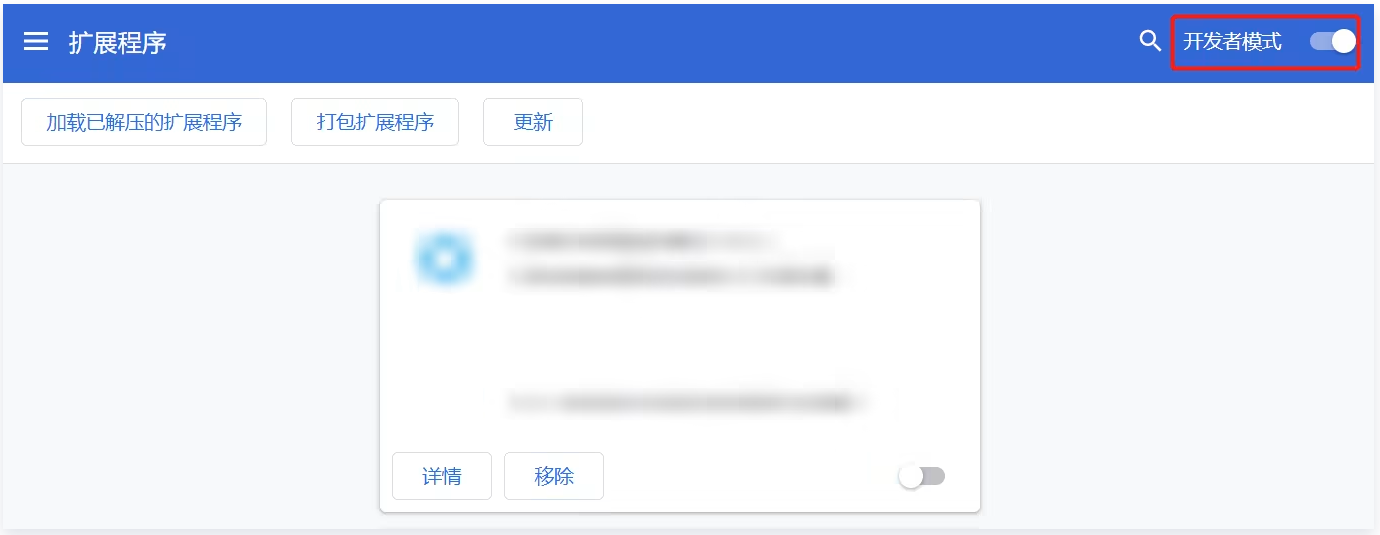
最近更新时间：2024-10-31 18:11:52

您可以在 Chrome 上安装 Tencent PTS Recorder 流量录制插件。PTS Recorder 将会录制您在浏览器上的操作和发送的请求以及对应的响应内容。并自动生成 PTS 压测场景。
本文介绍如何使用 PTS 流量录制功能，帮助您快速模拟业务场景，发起压测。

操作步骤

安装插件

1. 下载 PTS Recorder 插件，下载到本地，并解压。
2. 打开 Chrome 浏览器，地址栏输入 `chrome://extensions/` ，进入扩展程序管理页面。单击扩展页右上角按钮，切换到开发者模式。



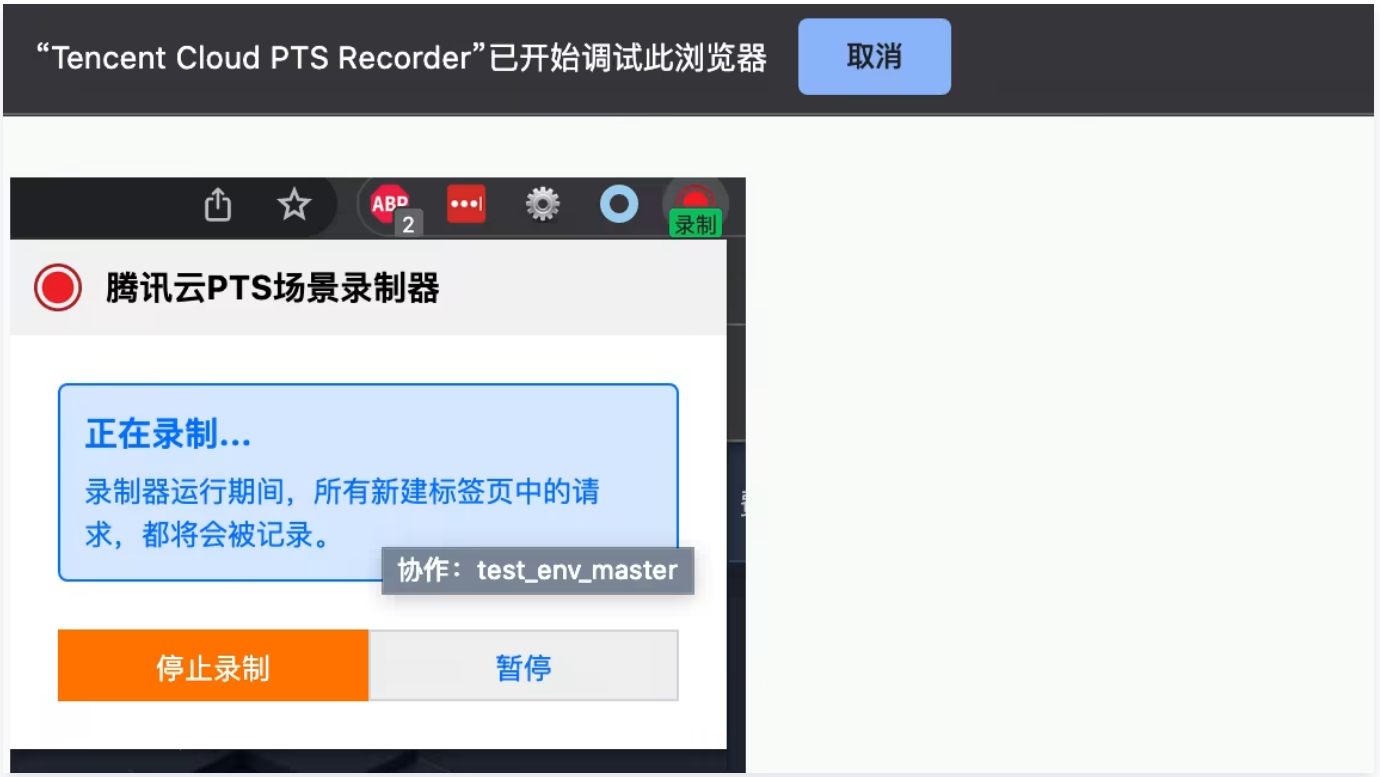
3. 单击左上角**加载已解压的扩展程序**，选择下载并解压后 PTS Recorder 插件。
4. Chrome 浏览器插件列表中出现 Tencent Cloud PTS Recorder，即表示安装成功。
5. 成功安装插件后，请单击**刷新页面** 后，重新单击**流量录制**。
6. 插件安装完成后，您可以在 Chrome 浏览器顶部菜单栏，单击 图标，再单击 PTS 录制器插件旁的 图标，固定此插件。

使用插件

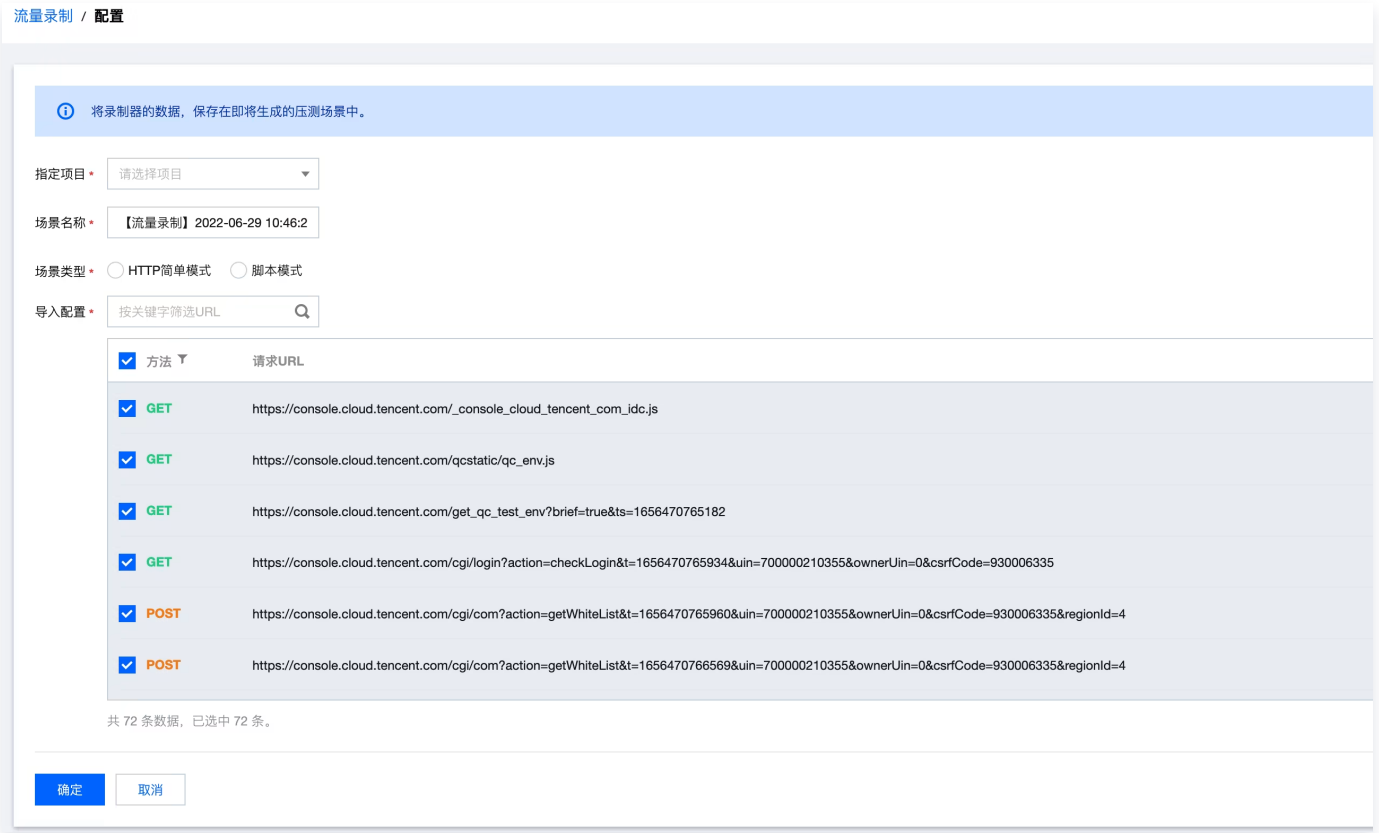
1. 插件安装完成后，填写下列信息，并单击 开始录制。
   - 起始页面：录制器启动后跳转的页面，在该页面开始录制用户操作流量。
   - URL 筛选：通过 URL 筛选，可以录制您感兴趣的流量。例如您只关心发送到域名为 console.cloud.tencent.com 的请求，那么在 URL 筛选中填入该域名即可。
   - 类型筛选：选择仅录制您感兴趣的请求类型。

   > **说明：**
   > 录制类型：
   > - XHR： application/json,text/xml,text/plain,application/xml
   > - JS：application/javascript,text/javascript,application/x-javascript
   > - HTML：text/html
   > - CSS：text/css
   > - Image：image/*
   > - 其他：others/*,application/json,text/xml,text/plain,application/xml,-application/javascript,text/javascript,application/x-javascript,text/html,text/css,image/*

2. 在录制页面上执行您的操作，PTS 将自动记录您的操作。您可以浏览器底部菜单栏下看到 PTS 录制中的提示。单击 PTS 插件，也可看到录制中的提示。



3. 操作完成后，浏览器底部菜单栏下方的提示中单击**取消**，或者在 PTS 插件中单击**停止录制**。在流量录制页面中筛选出您感兴趣的请求，生成压测场景。



环境管理

最近更新时间：2024-07-01 16:33:21

简介

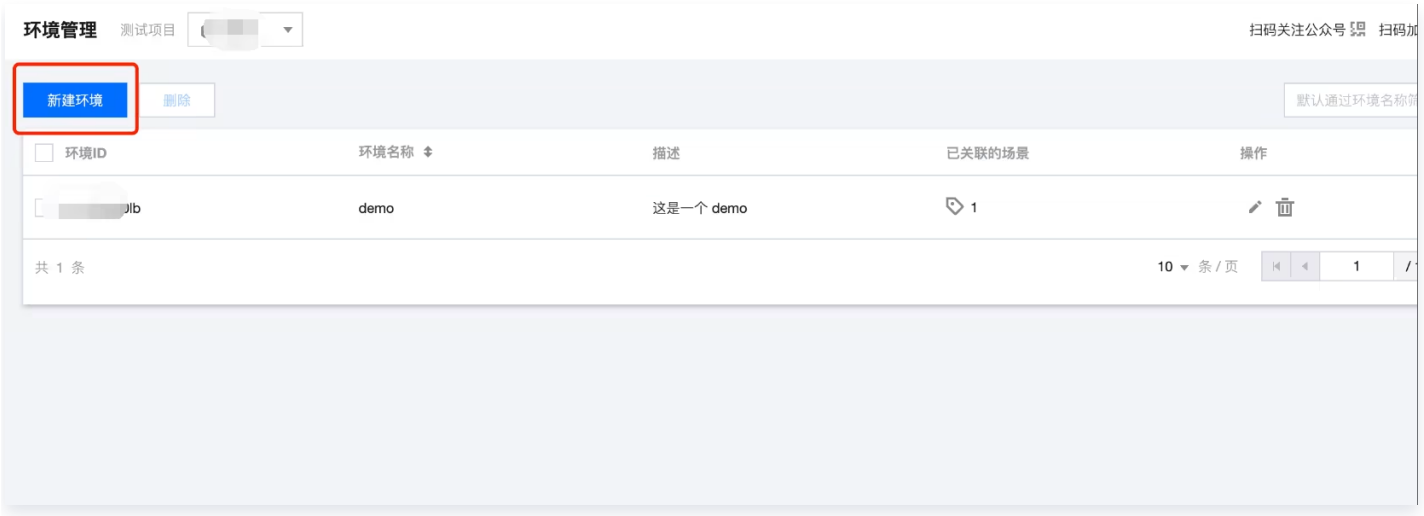
环境管理支持用户创建不同的环境变量组，每个变量组中可以创建不同的环境变量。在压测过程中，可以在压测脚本中使用环境变量。

环境管理

PTS 支持环境配置的增删改查操作。

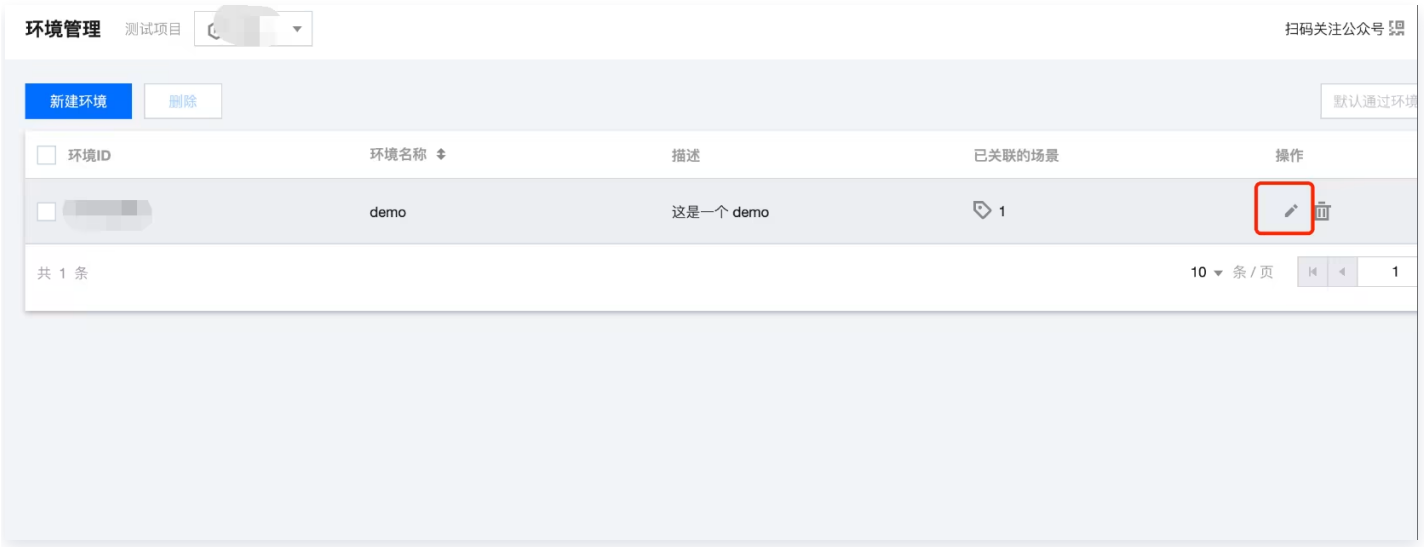
环境创建

登录 腾讯云可观测平台，在左侧导航栏中选择**云压测 > 环境管理**，点击**新建环境**，创建环境和变量，并进行保存。



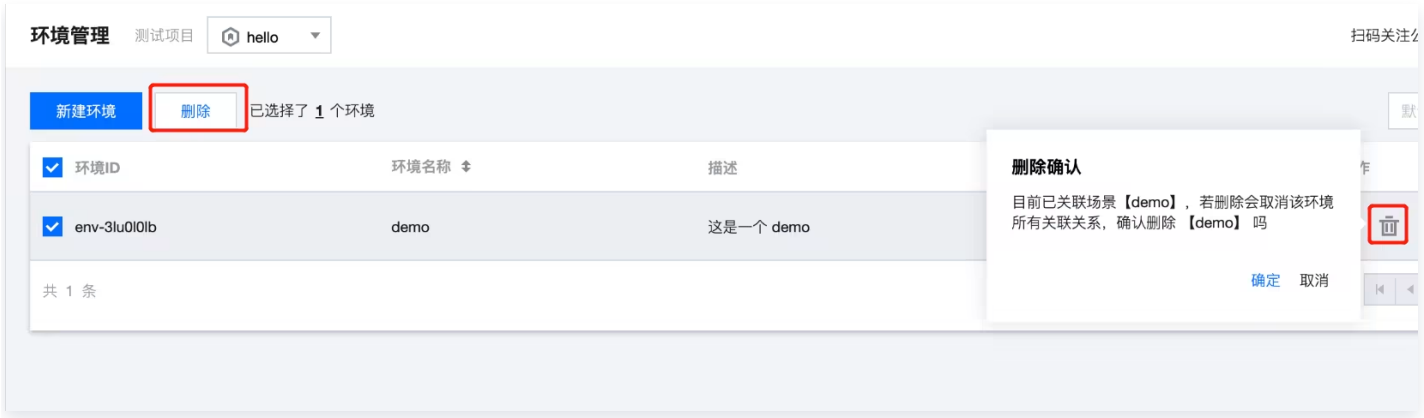
环境编辑

登录 腾讯云可观测平台，在左侧导航栏中选择**云压测 > 环境管理**，选择目标环境的 ID，点击 ，即可对已有的环境进行编辑处理。



环境删除

登录 腾讯云可观测平台，在左侧导航栏中选择**云压测 > 环境管理**，选择目标环境的 ID，点击**删除**或者 时，会提示已经关联的测试场景，然后点击弹框中的**确定**即可删除。



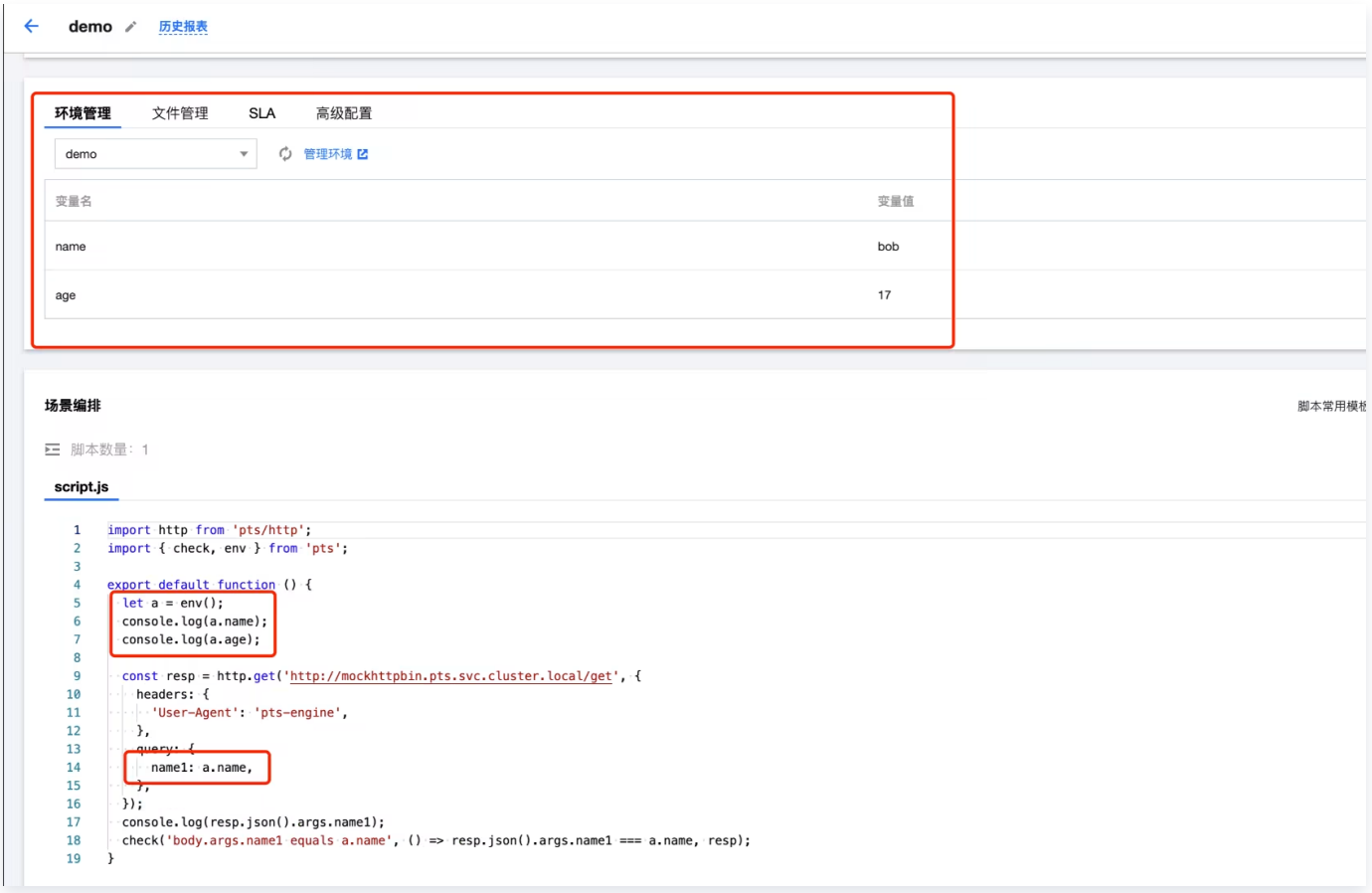
如何使用

创建完环境和变量后，可以在简单测试场景和 js 脚本中进行引用。

脚本模式

1. 登录 腾讯云可观测平台，在左侧导航栏中选择**云压测 > 测试场景**，点击**新建场景**。
2. 选择**脚本模式**。
3. 在测试场景中进行施压配置，配置详情可参见 施压配置 。

4. 点击**环境管理**，管理具体的环境，然后在脚本中，使用 env 函数即可。

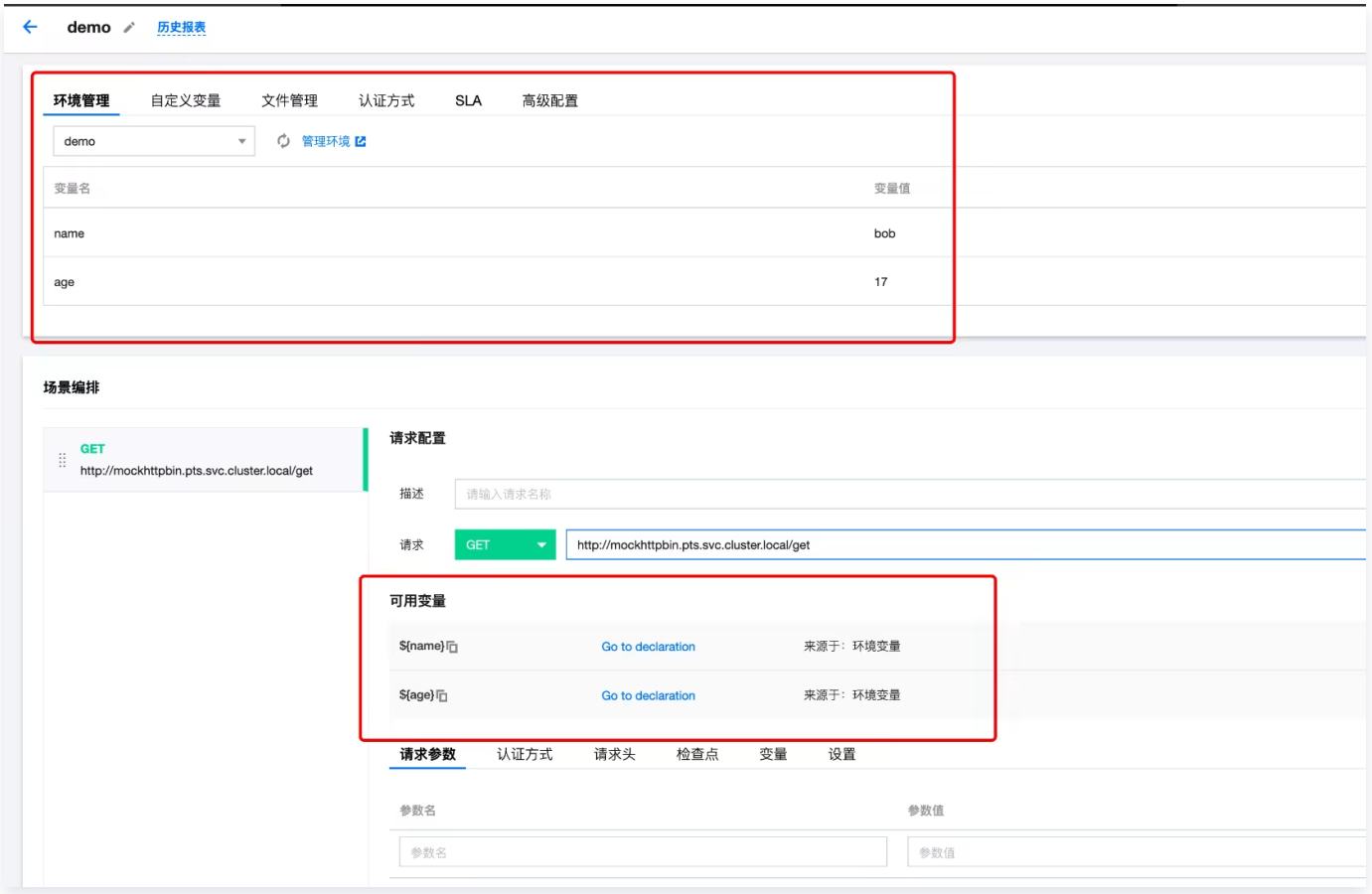



```
import {env} from 'pts'

let a = env()
console.log(a.name)
console.log(a.age)
```

简单模式

1. 登录 腾讯云可观测平台 ，在左侧导航栏中选择**云压测 > 测试场景**，点击**新建场景**。
2. 选择**简单模式**。
3. 在测试场景中进行施压配置，配置详情可参见 施压配置 。

4. 点击**环境管理**，管理具体的环境，简单模式会自动生成变量引用，直接使用即可。



定时压测

最近更新时间：2024-10-31 18:11:52

操作场景

当您需要定期执行压测任务时，可以使用定时压测功能。定时压测可以指定压测任务的执行时间、频率、通知对象等。本文将介绍如何使用定时压测功能，帮助您快速上手，发起定时压测。

使用前提

- 为了确保压测任务的可行性，您需提前创建并调试了您的测试场景，详情可参见 相关操作指引 。
- 提前 新建告警联系人 ，告警联系人用于在压测任务启动和结束时的通知。

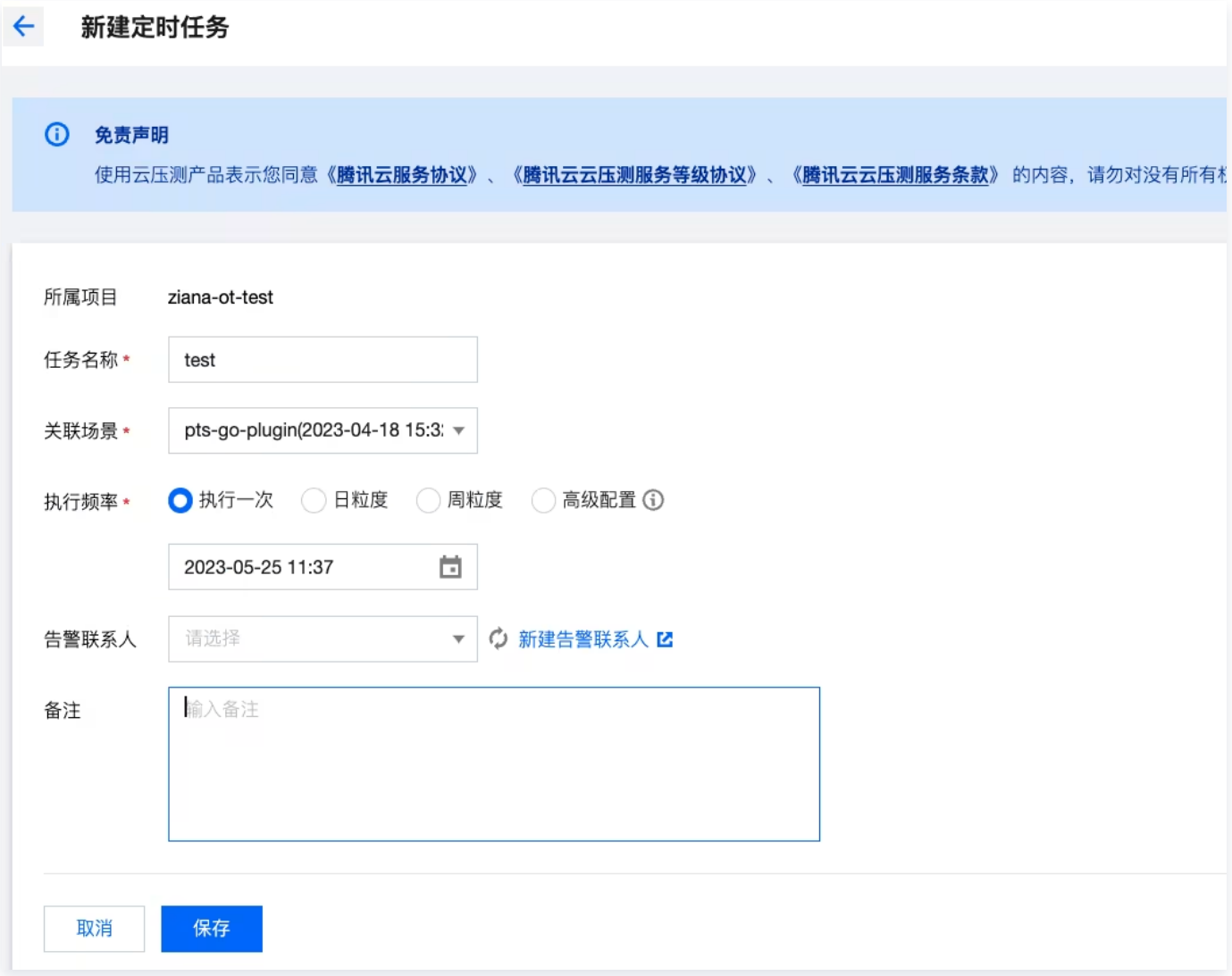
使用限制

在使用定时压测功能之前，您需要知悉定时压测的使用限制：

- 一个测试场景只能用于一个定时压测任务。建议您单独为定时压测创建场景。
- 已经用于定时压测任务的测试场景不可再编辑，只有删除该定时压测任务后，场景的限制才会解除。

操作指南

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**云压测 > 定时压测 > 创建定时任务**
3. 进入**新建定时任务**页面，根据下列描述配置定时压测任务。



配置项	说明
任务名称	自定义定时任务名称
关联场景	选择您需要关联的场景名称
执行频率	支持执行一次，日粒度（每个月的某日执行），周粒度（每周的星期几执行），高级配置（若前三种方式无法满足您的需求，您可以使用 cron 表达式来定义执行频率），详情可参见 执行频率介绍 。
告警联系人	选择对应的告警联系人，若无可进入 告警联系人 页面进行创建，用于接收压测任务启动和结束相关的通知。

4. 创建完后您可以在定时压测任务列表。查看定时压测任务的基本信息，包括关联的测试场景、以文字形式描述的执行频率、任务状态等。

- 若您想要快速找到某一个定时压测任务，可以通过排序或任务名称模糊搜索实现。
- 若您想要查看已经执行的压测任务的报表，可以单击对应场景快速跳转到报表界面。

执行频率介绍

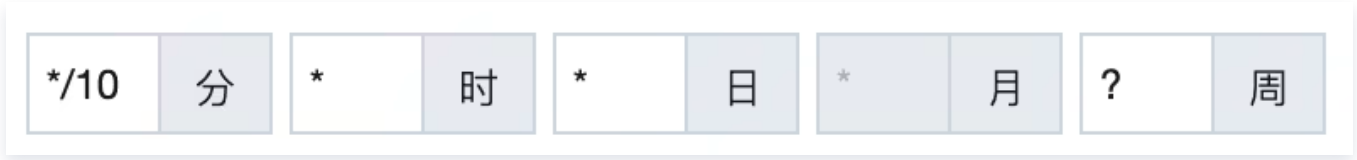
- **执行一次**：您需要选择压测任务执行的时间，且该时间必须大于当前时间。定时压测任务会在指定时间运行一次压测任务，并将状态更新为已完成。
- **日粒度**：如果您想在每个月的某些天执行一次压测任务，可以选择日粒度，如图选择2号、4号的16:35，则定时压测任务会在每个月2号和4号的16:35执行一次压测任务。另外，您可以填写**结束时间**（可不填），定时压测任务会在到达结束时间时更新状态为已完成。
- **周粒度**：如果您想在每周的某些天执行一次压测任务，可以选择周粒度。
- **高级配置**：如果前三种方式都没法满足您的需求，那您可以直接使用 cron 表达式来定义执行频率（cron 表达式的介绍放在下一节），在您填写 cron 表达式的过程中，我们会检查您的表达式是否正确，这能辅助您正确填写。如果填写正确，我们会展示前5次执行的时间，方便您检查该表达式是否符合您的需求。

cron 表达式

- 只能选择按照日或者周，指定执行压测时间。 日和周只能二选一，指定日或周之后，另外一个输入框内必须填写?，表示不受限制。
- 只能使用英文符号，不能使用中文符号（若您认为填写正确，但提示填写错误，可以检查 * 和?是否是英文）。
- 表示多个值时，使用英文逗号连接。例如， 日输入框中填写 1,5 表示每月一号和五号。
- 表示区间值时，用短横线连接。例如， 日输入框中填写 1-5 表示每月一号至五号。
- 表示指定频率时，使用斜杠连接。例如， 日输入框中填写 */2 表示每隔两天。

示例：

- 每隔10分钟执行一次。



- 每个月一号至五号的2点和8点每隔半小时执行一次。详情可看图中的最近5次执行时间。

*/30	分	2,8	时	1-5	日	*	月	?	周

注意：当定时任务运行的时间间隔，小于关联场景的持续时间时，定时任务任务会并发运行。

Cron表达式: */30 2,8 1-5 * ?

最近5次执行时间：

2022-08-01 02:00

2022-08-01 02:30

2022-08-01 08:00

2022-08-01 08:30

2022-08-02 02:00

压测报告
解读报告

最近更新时间：2024-12-27 11:58:52

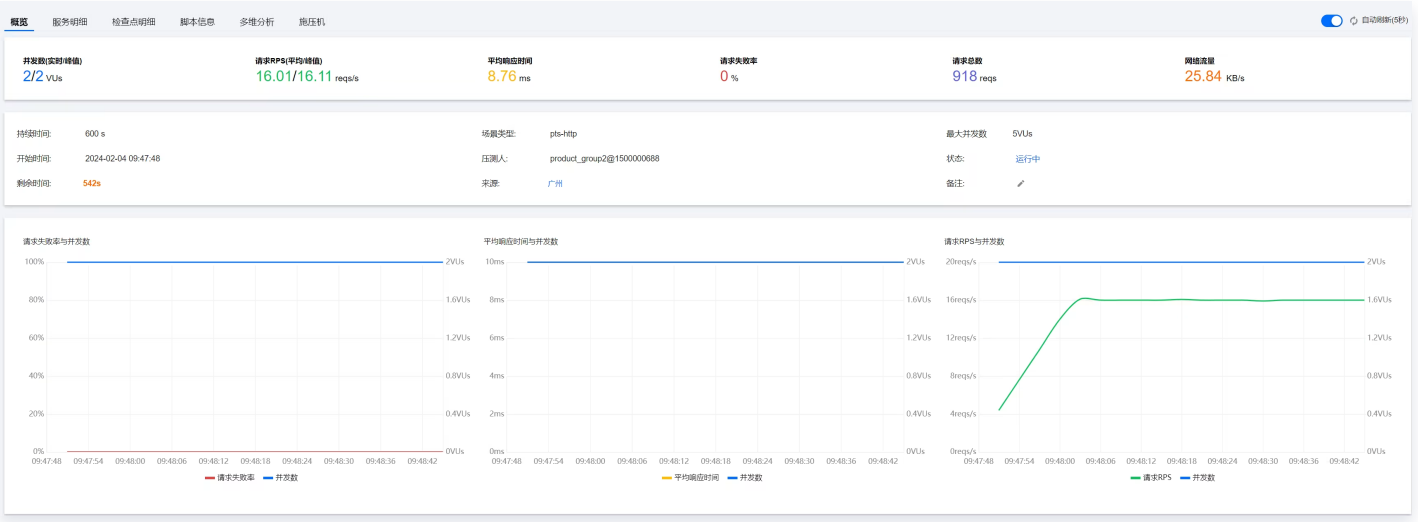
概述

云压测会将一次压测的结果展示在压测报表中。压测报表分为实时报表和历史报表两种状态，前者供您在压测过程中实时查看数据，后者供您在压测结束后查看历史数据。

> **说明：**
> 云压测历史报表保留期限为45天，**45天后将自动清理过期报告**。您可以在报告过期前下载 PDF 格式压测报告作为备份。

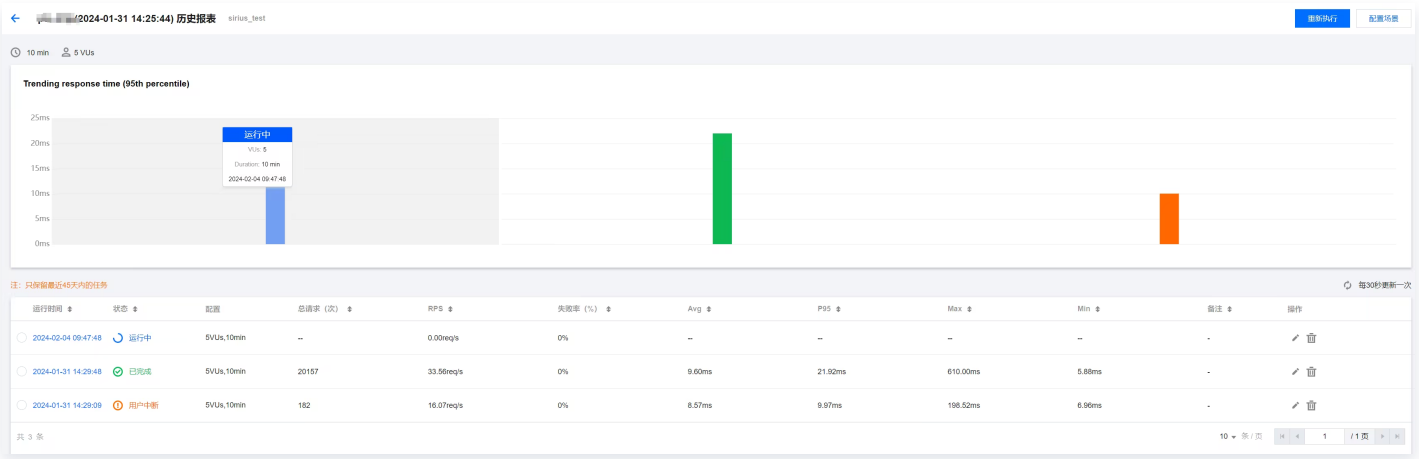
实时报表

当您触发运行您的压测场景，PTS 经过一些资源准备步骤后，会为您创建出一个压测任务。进入 测试场景 ，选择并点击需查看的压测任务，页面动态展示该任务的压测数据，并以一定的频率实时刷新。



历史报表

当您的压测场景的一次压测任务完成后，您可进入 测试场景 ，单击压测任务的右上角的 ··· > **历史报表**，即可进入历史报表总览页面。找到并单击需查看的历史报表，即可查看历史数据。



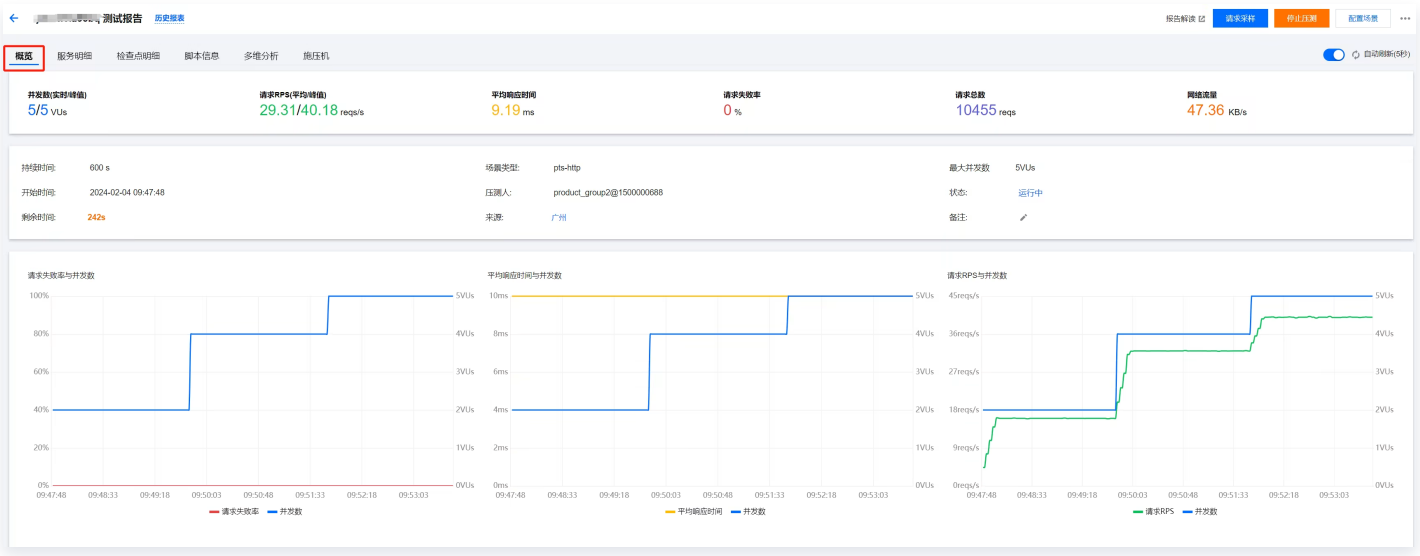
报表数据

概览

概览页展示一些最核心的概览数据，如压测任务本身的元数据及压测结果里最常用的指标及其图表（如 VU、RPS、平均响应时间）。

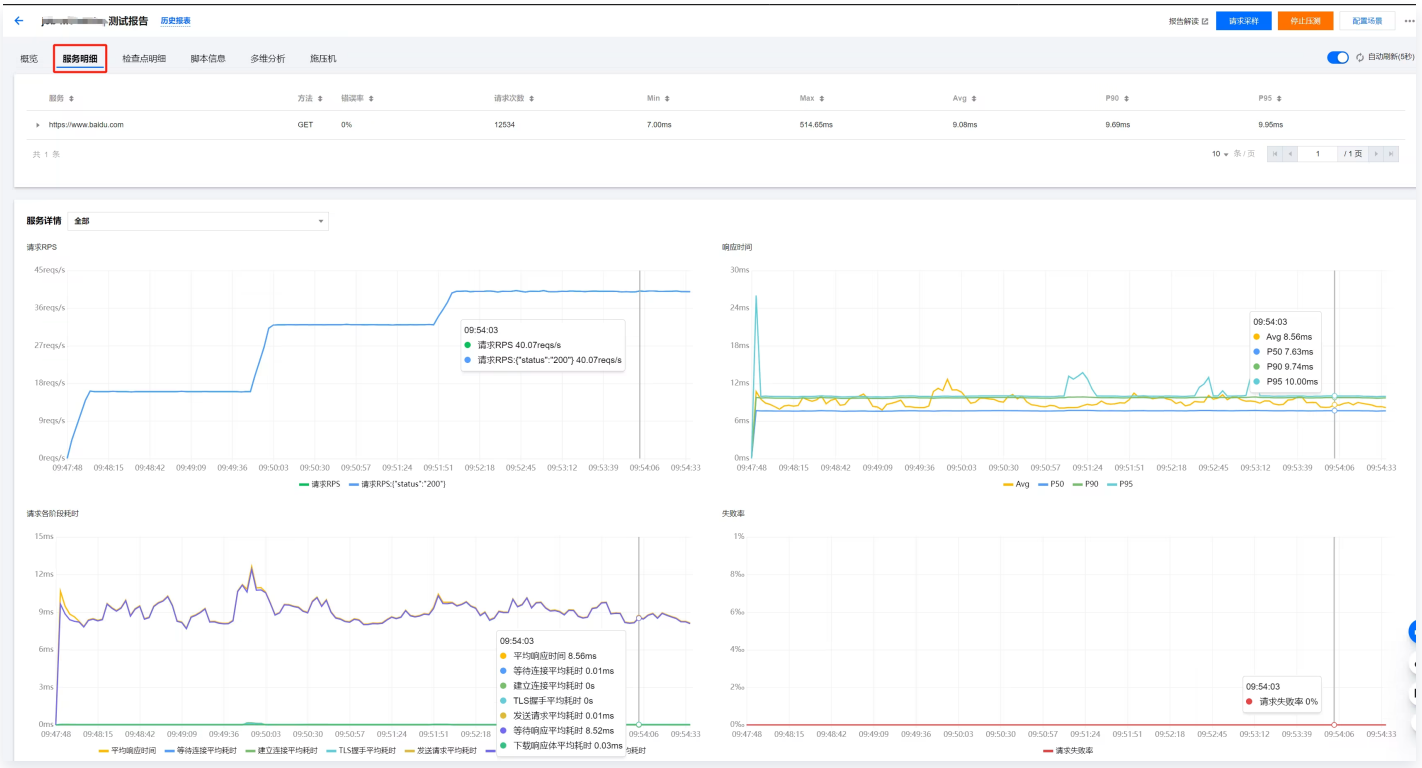
- 概览页最上方一栏，为压测任务的总览数据，其中：
  - 并发数、请求总数，为压测任务运行时刻的瞬时值。
  - RPS、平均响应时间、失败率、网络流量，为压测任务运行期间的平均值。
- 概览页中间一栏，为压测任务的持续时间、压测人、状态等元数据。
- 概览页最下方一栏，为压测任务的实时曲线，展示各指标在各时间点的瞬时值。

> **说明**
>
> 关于并发用户数、RPS、响应时间的概念介绍及其之间的关系，请参见 常见问题 。



服务明细

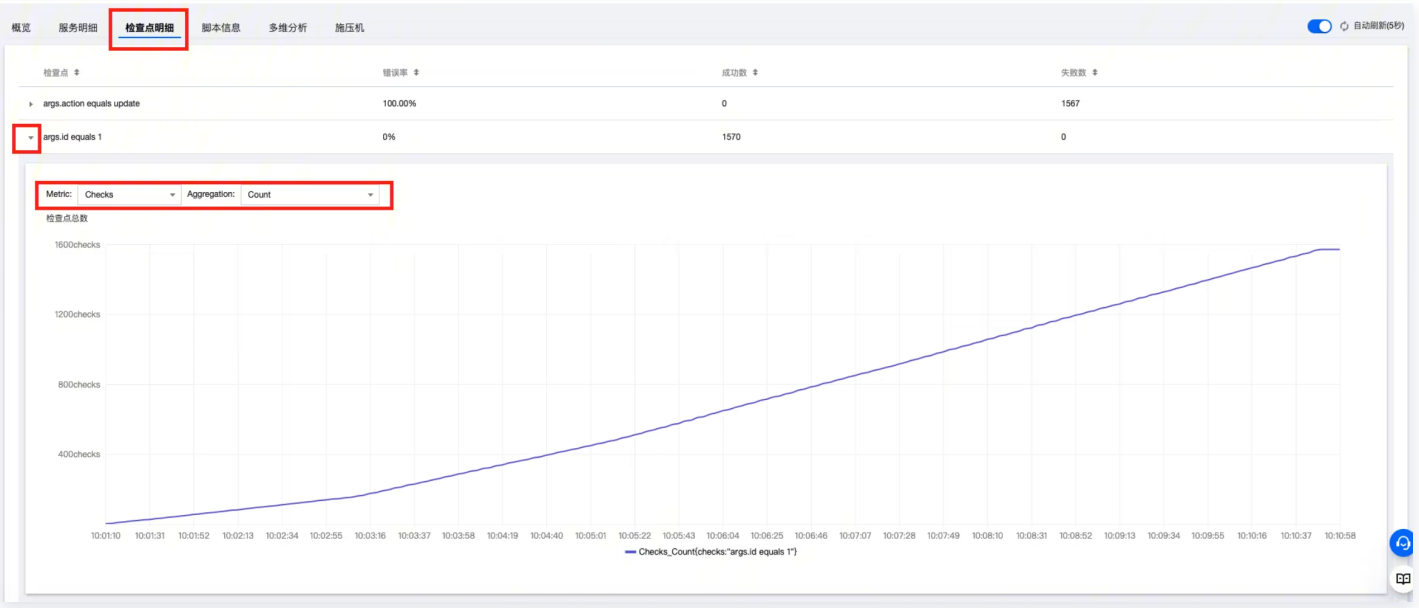
服务明细页默认将每个 URL 归类为一个“服务”，展示压测期间发送的所有请求的明细信息。
您可单击展开每个服务的详情，查看其数据及图表。在图表中，您可点击切换指标名（Metric）或聚合方式（Aggregation），来切换查看。



> **说明**
> 在 PTS 中，不同服务默认是按照不同的 URL 来归类的。若您需要自定义服务归类，可在脚本模式的场景中，指定 `http.Request` 中的 `service` 属性。请参见 JavaScript API 列表 。

检查点明细

在**检查点明细**页面，您可查看您在场景中所设置的检查点的结果明细。

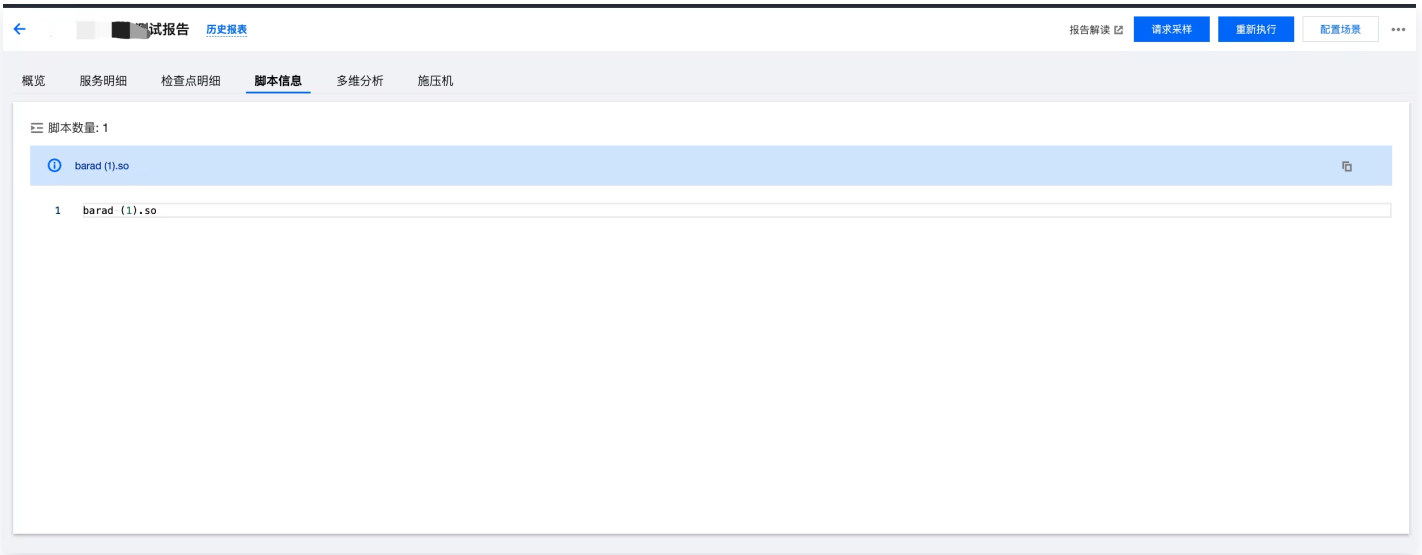


说明

关于如何设置检查点，请参见 设置检查点 。

脚本信息

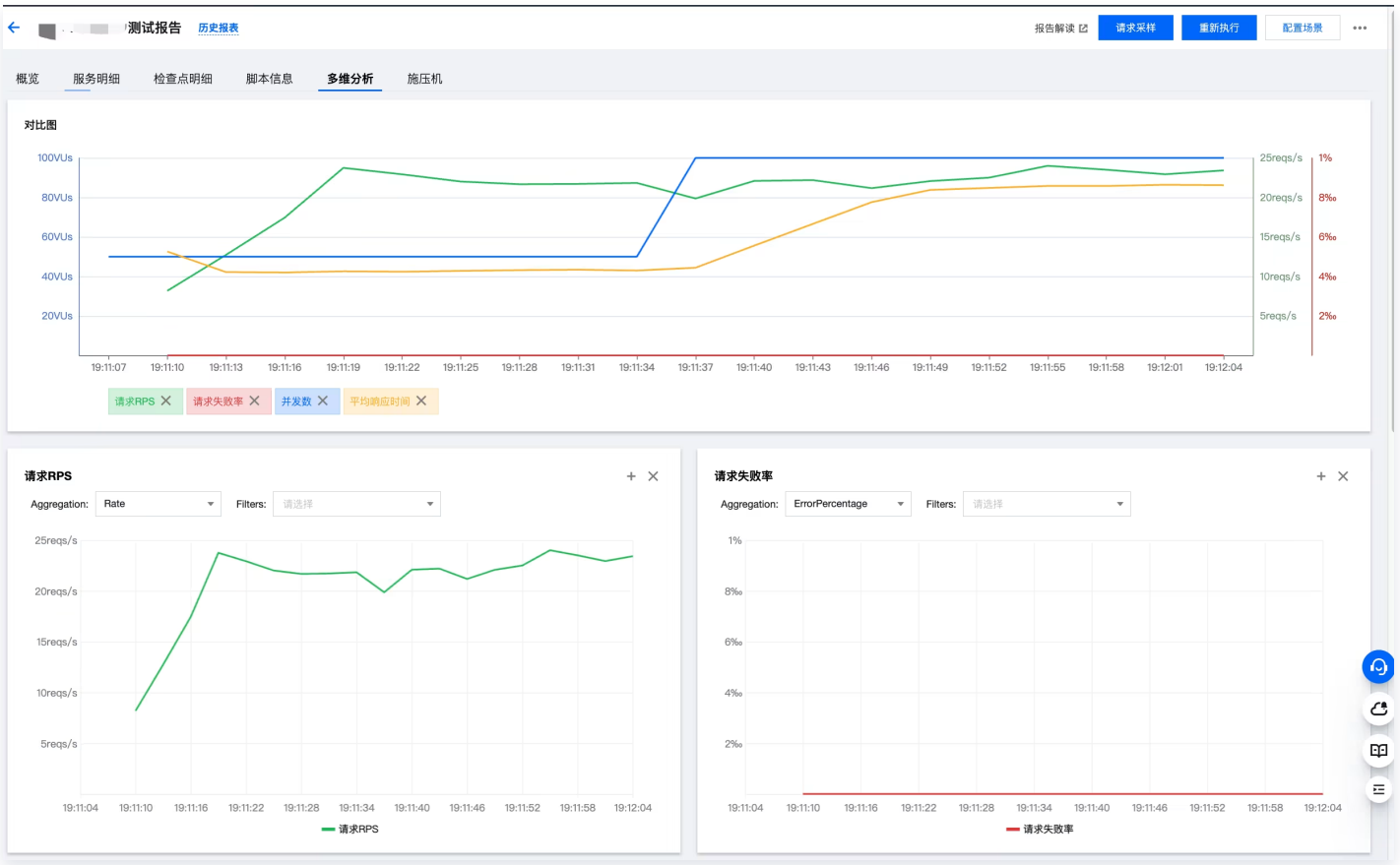
在**脚本信息**页面，您可查看压测任务执行时所使用的场景脚本的快照。



多维分析

在**多维分析**页面，您可交互式地切换查看多种压测结果数据的图表组合。您可单击切换指标名（Metric）或聚合方式（Aggregation），来切换查看不同图表。

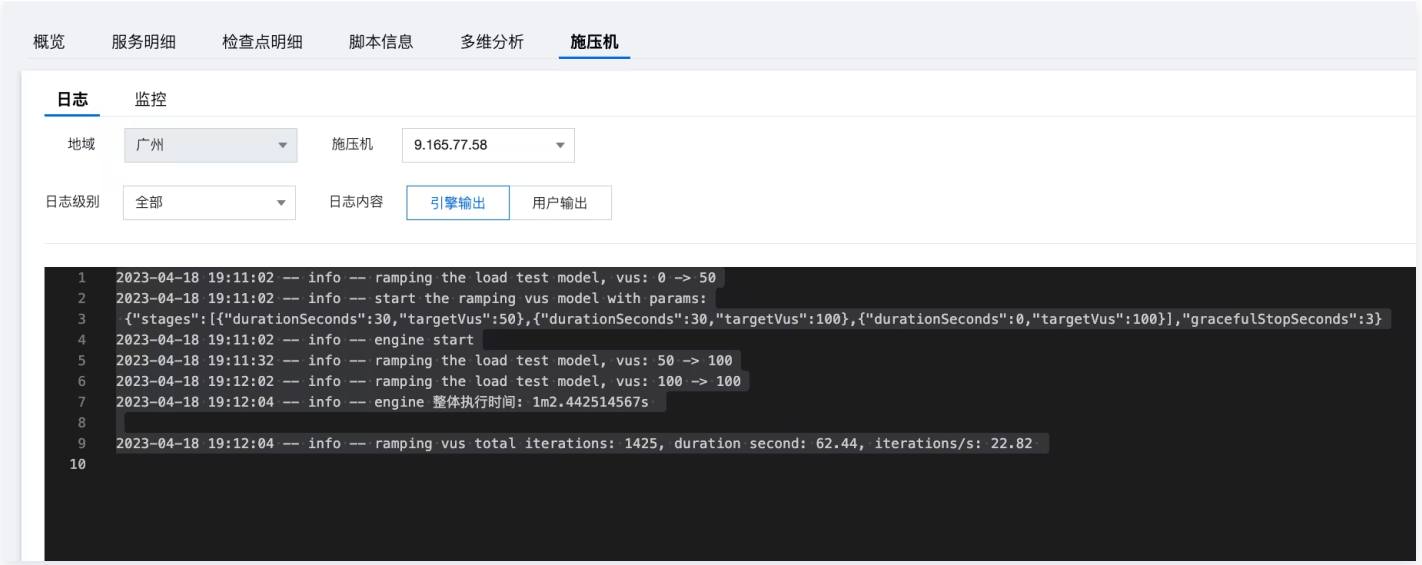
您还可单击页面下方**添加指标**，新建您所需的数据图表。



施压机

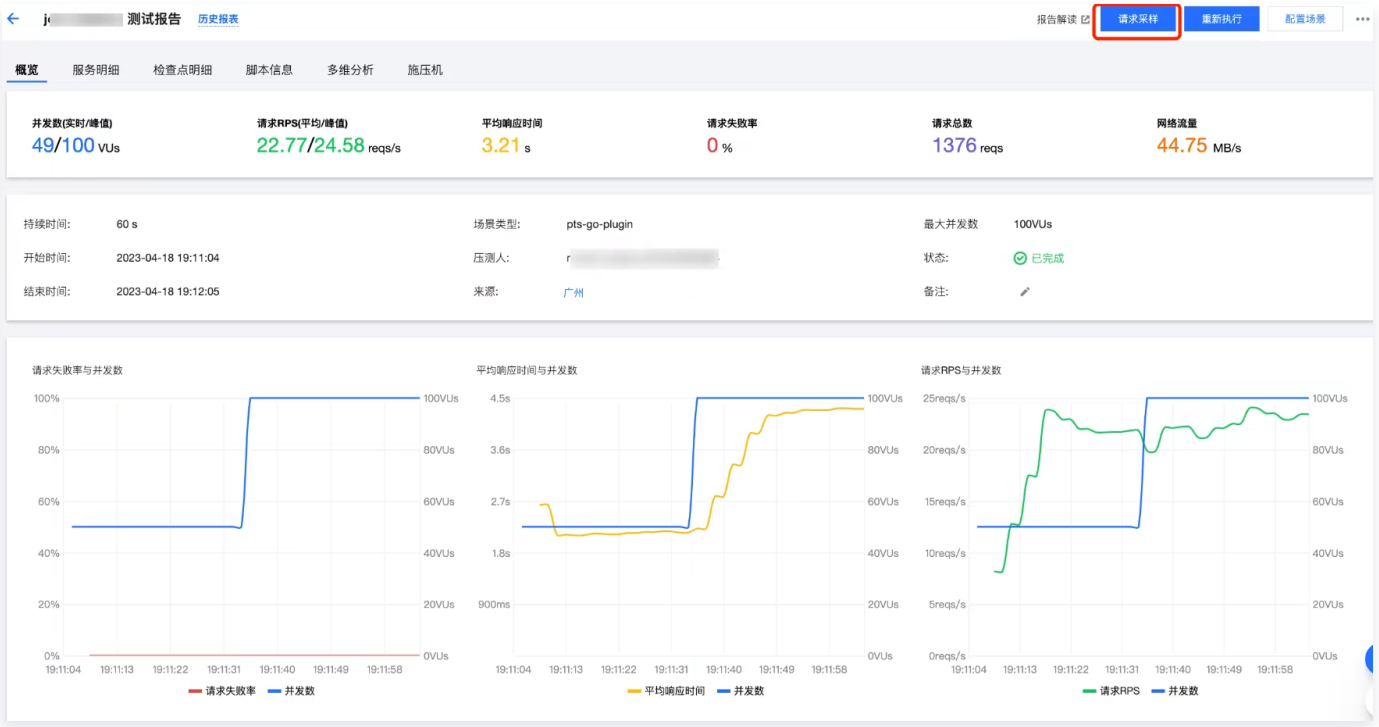
在**施压机**页面，您可查看该压测任务的施压机的基本信息、在压测过程中输出的日志、施压机本身的资源使用状况。其中，压测日志可按日志级别（debug/info/error）和日志内容（用户输出/引擎输出）分类，您可在下拉列表中切换。

- 用户自行打印的日志，将展示在**用户输出**标签页。
- PTS 打印的通用日志，将展示在**引擎输出**标签页。

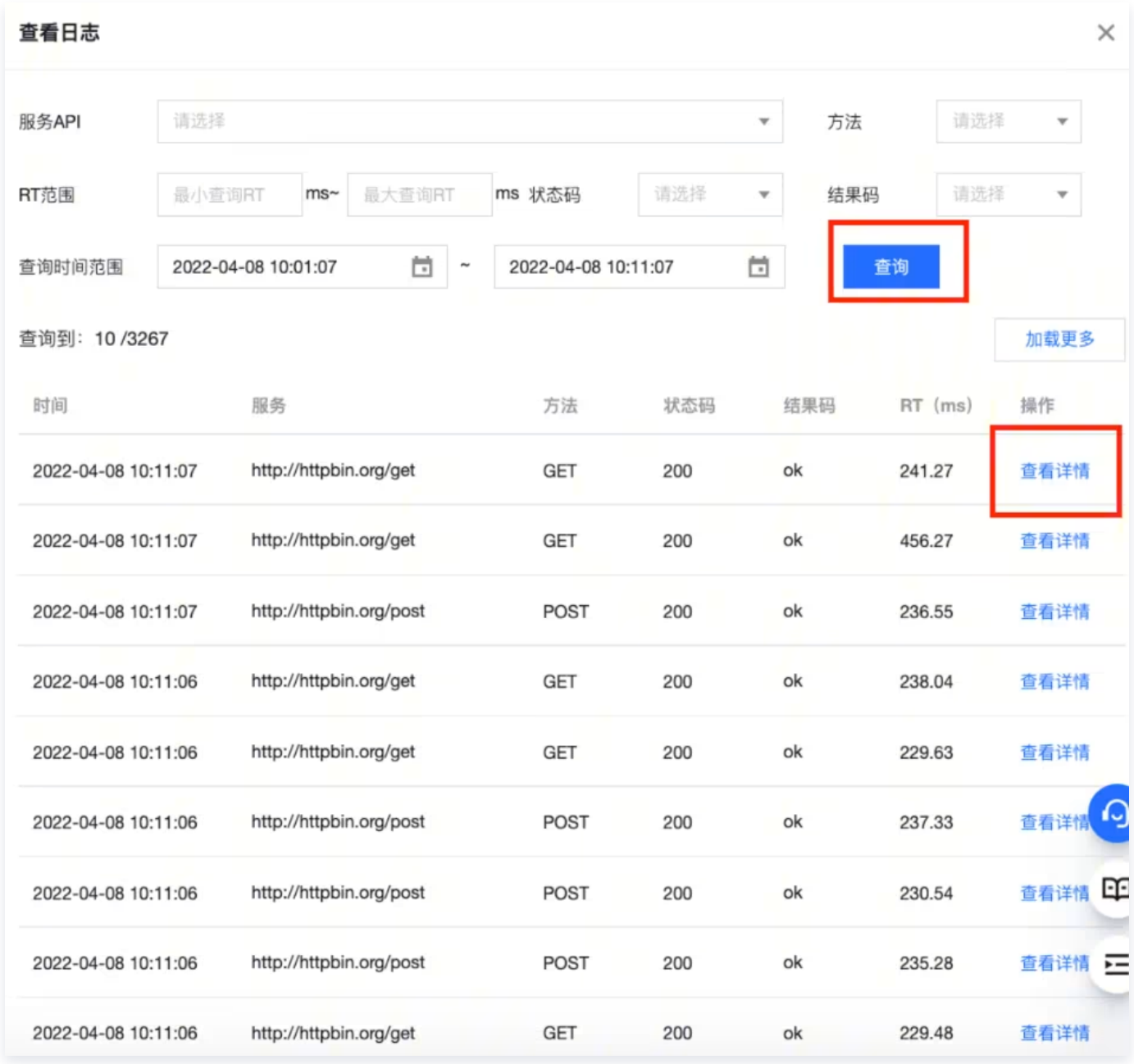


请求采样

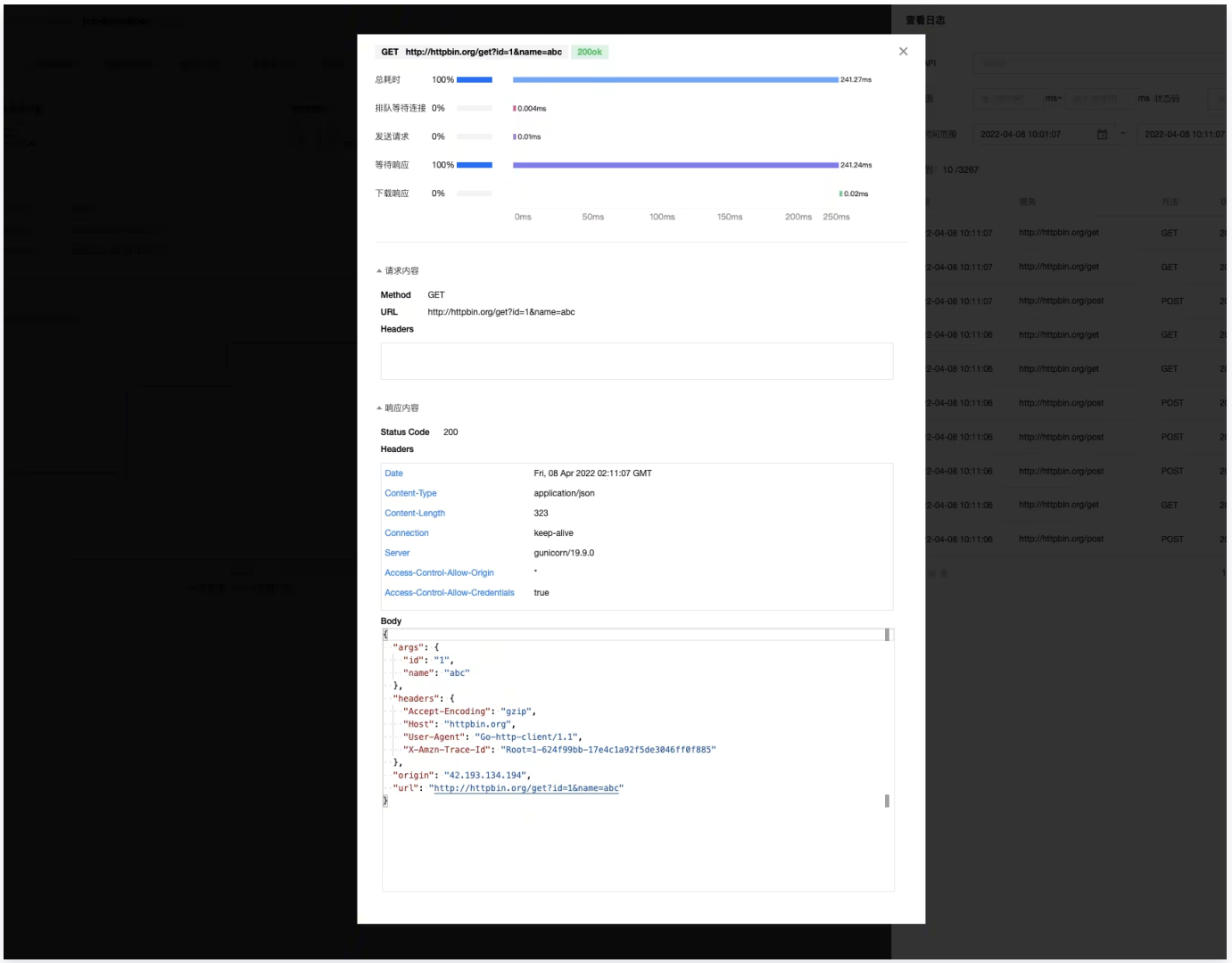
1. 单击**请求采样**，您可查看施压端采样选取的部分请求的详细信息。



2. 输入相应条件，筛选所需请求。在请求列表中，单击**查看详情**，可展开单条请求的详情页。



在单条请求的详情页中，您可查看它的请求和响应的详细信息，以及请求耗时分布的瀑布图。


GET http://httpbin.org/get?id=1&name=abc 200ok
总耗时 100% 241.27ms
排队等待连接 0% 0.004ms
发送请求 0% 0.01ms
等待响应 100% 241.24ms
下载响应 0% 0.02ms
0ms 50ms 100ms 150ms 200ms 250ms
请求内容
Method GET
URL http://httpbin.org/get?id=1&name=abc
Headers
响应内容
Status Code 200
Headers
Date Fri, 08 Apr 2022 02:11:07 GMT
Content-Type application/json
Content-Length 323
Connection keep-alive
Server gunicorn/19.9.0
Access-Control-Allow-Origin *
Access-Control-Allow-Credentials true
Body
{
"args": {
"id": "1",
"name": "abc"
},
"headers": {
"Accept-Encoding": "gzip",
"Host": "httpbin.org",
"User-Agent": "Go-http-client/1.1",
"X-Amzn-Trace-Id": "Root=1-624f99bb-17e4c1a92f5de3046ff0f885"
},
"origin": "42.193.134.194",
"url": "http://httpbin.org/get?id=1&name=abc"
}

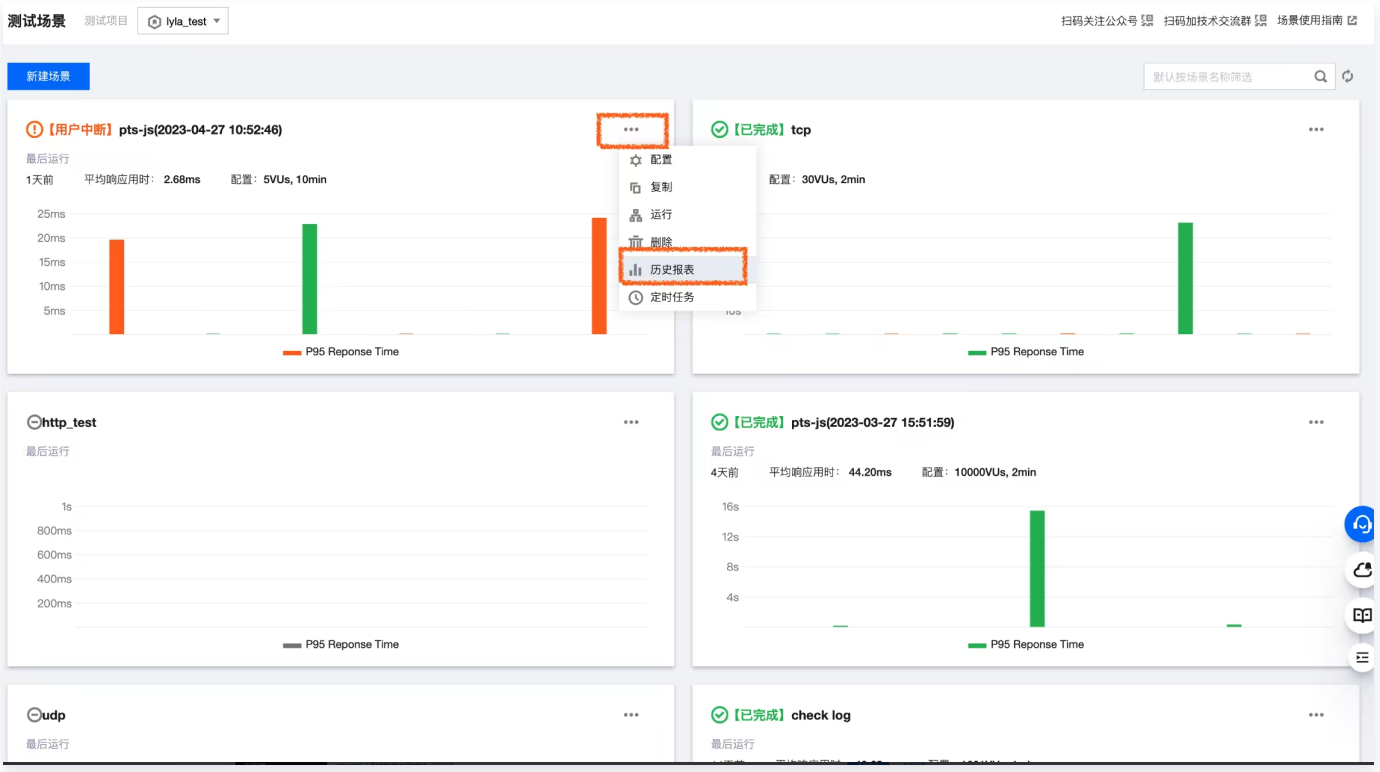
下载报告

最近更新时间：2024-08-16 11:52:01

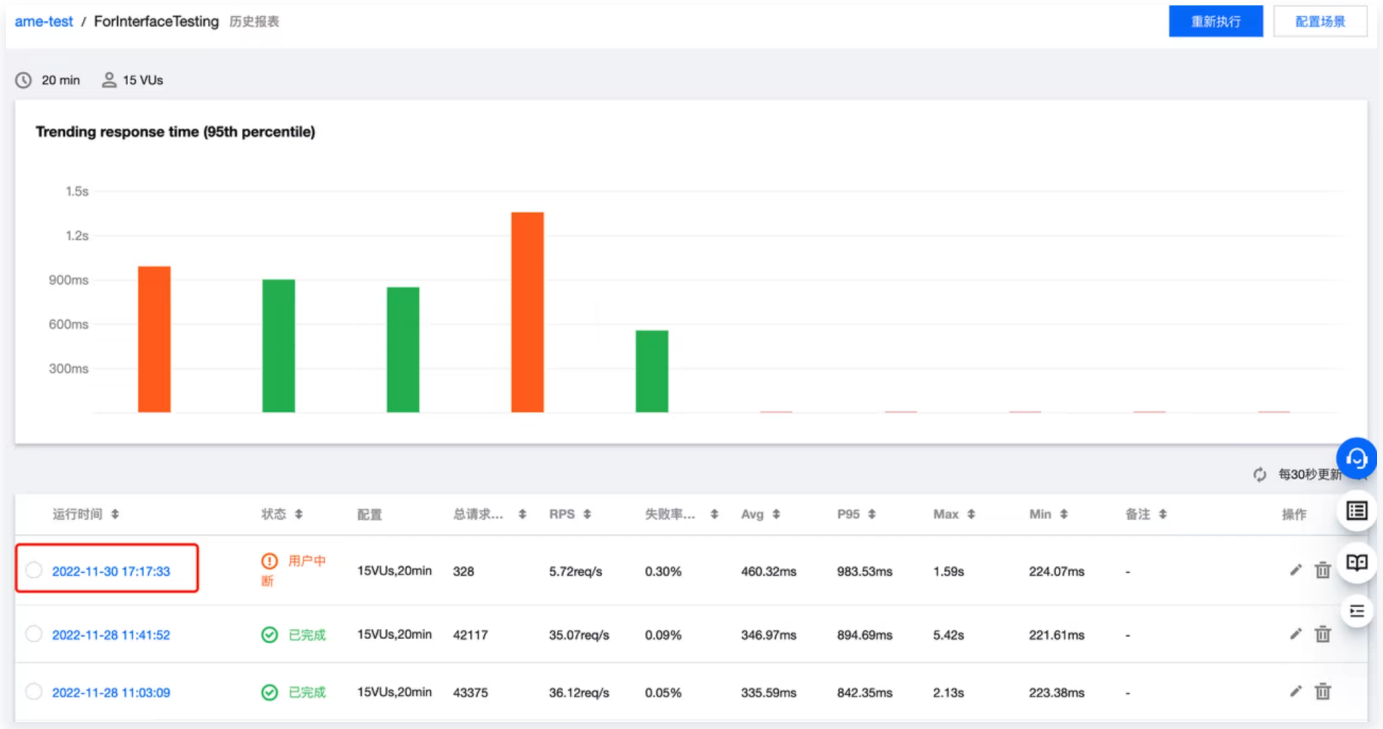
本文将为您介绍如何下载一次已完成的压测任务的历史报告。

操作步骤

1. 登录 腾讯云可观测平台控制台 。
2. 在左侧菜单栏选择**云压测 > 测试场景**页面。
3. 在测试场景的右上角，单击 ••• **> 历史报表**，进入历史报表页面。



4. 在历史报表页面，单击想要下载的历史报表。



5. 在报表页点击右上角 ••• > **下载压测报告**，您可下载一次已完成压测任务的历史报告。

访问控制

概述

最近更新时间：2024-04-22 18:05:51

如果您在腾讯云中使用到了性能测试服务，该服务由不同的人管理，但都共享您的云账号密钥，将存在以下问题：

- 您的密钥由多人共享，泄密风险高。
- 您无法限制其他人的访问权限，易产生误操作造成安全风险。

此时，您就可以通过子账号实现不同的人员管理不同的服务，来规避以上的问题。默认情况下，子账号无使用性能测试权限。因此，我们需要创建策略来允许子账号使用他们所需要资源的权限。

简介

访问管理（Cloud Access Management，CAM）是腾讯云提供的一套 Web 服务，它主要用于帮助客户安全管理腾讯云账户下的资源的访问权限。通过 CAM，您可以创建、管理和销毁用户（组），并通过身份管理和策略管理控制哪些人可以使用哪些腾讯云资源。
当您使用 CAM 时，可以将策略与一个用户或一组用户关联起来，策略能够授权或者拒绝用户使用指定资源完成指定任务。有关 CAM 策略的更多相关基本信息，请参见 策略语法 。有关 CAM 策略的更多相关使用信息，请参见 策略 。

授权方式

云压测支持资源级授权和按标签授权两种方式：

- 资源级授权：您可以通过策略语法或默认策略授予子账号单个资源的管理权限，详细请参见 策略语法 和 策略授予 。
- 按标签授权：您可以通过给资源标记标签，实现给子账号对应的标签下资源的管理权限，详细请参见 资源标签 。

> **说明：**
> 若您无需对子账号进行性能测试服务相关资源的访问管理，您可以跳过此章节。跳过该部分不会影响您对文档中其余部分的理解和使用。

策略授予

最近更新时间：2024-09-14 11:48:11

子账号默认没有性能测试任何权限。需要主账号授予子账号相关权限，子账号才能正常访问云压测资源。

操作前提

使用拥有管理员权限（AdministratorAccess）或者拥有访问管理全读写权限（QcloudCamFullAccess）的子账号登录腾讯云控制台，请参考 新建子用户 创建子账户。

自定义策略

1. 使用拥有管理员权限（AdministratorAccess）或者拥有访问管理全读写权限（QcloudCamFullAccess）子账号进入 访问管理 > 策略 。
2. 单击**新建自定义策略 > 按策略语法创建**，选择空白模板。根据 策略语法 完成策略编辑。

策略授权

说明：

> 云压测为您创建默认策略 QcloudPTSFullAccess（云压测（PTS）全读写访问权限）和 QcloudPTSReadOnlyAccess（云压测（PTS）只读访问权限），您可以通过搜索策略名称快速进行默认策略授权。也可以对自定义策略进行授权。授权成功后，子账号才能正常访问相关资源。

1. 使用拥有管理员权限（AdministratorAccess）或者拥有访问管理全读写权限（QcloudCamFullAccess）权限的子账号进入 访问管理 > 策略 。
2. 进入策略管理页，在策略名称搜索框中输入对应的策略名称。
3. 选择只读访问或全读写访问权限，在操作列中单击**关联用户/组/角色**。

支持资源级授权的 API 列表

API 操作	API 描述
AbortJob	停止任务
CreateProject	创建项目
CreateScenario	创建场景
DeleteJobs	删除任务
DeleteProjects	删除项目
DeleteScenarios	删除场景
DescribeCheckSummary	查询检查点汇总信息
DescribeJobs	查询任务列表
DescribeLabelValues	查询标签内容
DescribeProjects	查询项目列表
DescribeRegions	查询地域列表
DescribeSampleBatchQuery	批量查询指标，返回固定时间点指标内容
DescribeSampleQuery	查询指标，返回固定时间点指标内容

DescribeSampleStreamBatchQuery	批量查询指标序列
DescribeSampleStreamQuery	查询一段时间范围内的指标序列
DescribeScenarioWithJobs	查询场景配置并附带已经执行的任务内容
DescribeScenarios	查询场景列表
DescribeServiceSummary	查询服务汇总信息
DescribeZones	查询可用区列表
GenerateTmpKey	生成临时 COS 凭证
StartJob	创建并启动任务
UpdateJob	更新任务
UpdateProject	更新项目
UpdateScenario	更新场景

不支持资源级授权的 API 列表

针对不支持资源级权限的云压测 API 操作，您仍可以向用户授予使用该操作的权限，但策略语句的资源（resource）元素必须指定为 * 。

API 操作	API 描述
CreateProject	创建 PTS 服务实例

策略语法

最近更新时间：2024-10-31 18:11:52

概述

访问策略可用于授予访问云压测相关的权限。访问策略使用基于 JSON 的访问策略语言。您可以通过访问策略语言授权指定委托人（principal）对指定的云压测资源执行指定的操作。
访问策略语言描述了策略的基本元素和用法，有关策略语言的说明请参见 CAM 策略管理 。

策略语法

CAM 策略

```
{
        "version":"2.0",
        "statement":
        [
            {
                "effect":"effect",
                "action":["action"],
                "resource":["resource"],
                 "condition": {"key":{"value"}}
            }
        ]
}
```

元素用法

- **版本 version**：必填项，目前仅允许值为"2.0"。
- **语句 statement**：描述一条或多条权限的详细信息，该元素包括 effect、action、resource，condition 等多个其他元素的权限或权限集合。一条策略有且仅有一个 statement 元素。
  - **影响 effect**：必填项，描述声明产生的结果为“允许”或“显式拒绝”，包括 allow（允许）和 deny（显式拒绝）两种情况。
  - **操作 action**：必填项，描述允许或拒绝的操作。操作可以是 API（以 name 前缀描述）或者功能集（一组特定的 API，以 permid 前缀描述）。
  - **资源 resource**：必填项，授权的具体数据。资源是用六段式描述。每款产品的资源定义详情会有所区别。有关如何指定资源的信息，请参见您编写的资源声明所对应的产品文档。

- **生效条件 condition**：非必填项，描述策略生效的约束条件。条件包括操作符、操作键和操作值组成。条件值可包括时间、IP 地址等信息。有些服务允许您在条件中指定其他值。

指定效力

如果没有显式授予（允许）对资源的访问权限，则隐式拒绝访问。同时，也可以显式拒绝（deny）对资源的访问，从而确保用户无法访问该资源，即使有其他策略授予了访问权限的情况下也无法访问。下面是指定允许效力的示例：

```
"effect" : "allow"
```

指定操作

云压测定义了可在策略中指定一类控制台的操作，指定的操作按照操作性质分为读取部分接口 `pts:Describe\*` 和全部接口 `pts:\*` 。
指定允许操作的示例如下：

```
"action": [
  "name/pts:Describe*"
]
```

指定资源

资源（resource）元素描述一个或多个操作对象，如性能测试服务等。所有资源均可采用下述的六段式描述方式。

```
qcs:project_id:service_type:region:account:resource
```

参数说明如下：

参数	描述	是否必选
qcs	qcloud service 的简称，表示是腾讯云的云服务	是
project_id	描述项目信息，仅为了兼容 CAM 早期逻辑，一般不填	否
service_type	产品简称，此处为 pts	是
account	描述资源拥有者的主账号信息，即主账号的 ID，表示为 `uin/${OwnerUin}`，如 uin/100000000001	是

resource	描述具体资源详情，前缀为 instance	是

下面是性能测试服务的四段式示例：

```
"resource":["qcs::pts:uin/1250000000:ProjectId/project-bx123456"]
```

实际案例

基于资源 ID，分配指定资源的读写权限，主账号 ID 为 1250000000。
示例：为子账号分配查询项目（ID：project-bx123456）权限。

```
{
    "version": "2.0",
    "statement": [
        {
            "effect": "allow",
            "action": [
                "pts:DescribeProjects"
            ],
            "resource": [
                "qcs::pts:uin/1250000000:ProjectId/project-bx123456"
            ]
        }
    ]
}
```

支持资源级授权的 API 列表

API 操作	API 描述
API 操作	API 描述
AbortJob	停止任务
CreateProject	创建项目
CreateScenario	创建场景
DeleteJobs	删除任务
DeleteProjects	删除项目

DeleteScenarios	删除场景
DescribeAllLabels	查询所有指标的labels
DescribeCheckSummary	查询检查点汇总信息
DescribeJobs	查询任务列表
DescribeLabelValues	查询标签内容
DescribeProjects	查询项目列表
DescribeRegions	查询地域列表
DescribeSampleBatchQuery	批量查询指标，返回固定时间点指标内容
DescribeSampleQuery	查询指标，返回固定时间点指标内容
DescribeSampleStreamBatchQuery	批量查询指标序列
DescribeSampleStreamQuery	查询一段时间范围内的指标序列
DescribeScenarioWithJobs	查询场景配置并附带已经执行的任务内容
DescribeScenarios	查询场景列表
DescribeServiceSummary	查询服务汇总信息
DescribeZones	查询可用区列表
GenerateTmpKey	生成临时COS凭证
StartJob	创建并启动任务
UpdateJob	更新任务
UpdateProject	更新项目
UpdateScenario	更新场景

不支持资源级授权的 API 列表

针对不支持资源级权限的云压测 API 操作，您仍可以向用户授予使用该操作的权限，但策略语句的资源（resource）元素必须指定为 * 。

API 操作	API 描述
CreateProject	创建 PTS 服务实例

告警管理
告警联系人

最近更新时间：2024-08-16 11:52:01

使用场景

- SLA 使用场景：当 SLA 规则被触发时，您还可以配置告警联系人，接收告警消息。PTS 能借助您的腾讯云账号下已有的通知渠道，向您发送 SLA 规则被触发的告警消息。
- 定时压测使用场景：您可以在设置定时压测时，绑定告警联系人，在压测任务启动和结束时的将会根据您筛选的渠道进行通知。

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**云压测 > 告警联系人 > 添加联系人组。**
3. 自定义联系人组名称，选择联系人，并选择接收渠道即可。若联系人列表不符合您的要求，您可以单击右侧的**新增用户**，参见 新建消息接收人 指引新增联系人即可。

   接口回调：填写回调地址，例如：`http://my.service.example.com`，接口回调具备将告警信息通过 HTTP 的 POST 请求推送到可访问公网 URL 的功能，您可基于接口回调推送的告警信息做进一步的处理。

告警历史

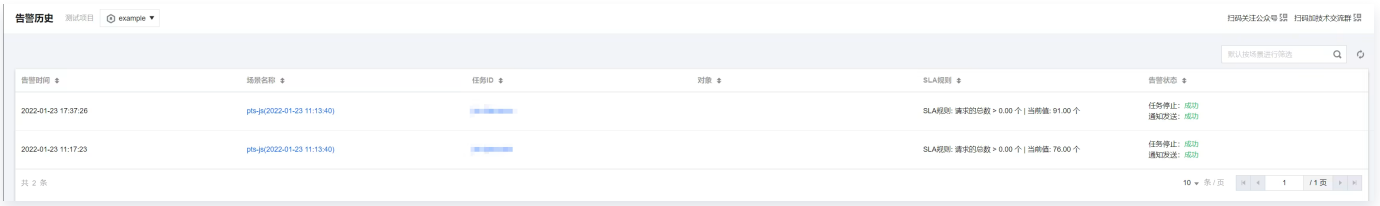
最近更新时间：2024-08-16 11:52:01

操作场景

当您在 SLA 规则或定时压测中绑定了告警联系人，当 SLA 规则被触发或定时压测任务启动和结束时，将会根据您筛选的渠道进行告警通知，历史的告警记录将会保存在告警历史中。

操作步骤

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**云压测 > 告警历史**，即可查看告警历史内容。



标签管理
标签概述

最近更新时间：2024-04-19 16:18:11

简介

标签是腾讯云提供的用于标识云上资源的标记，是一个键-值对（Key-Value）。您可以根据各种维度（例如业务、用途、负责人等）使用标签对 PTS 项目资源进行分类管理。
通过标签可便捷地筛选**过滤出对应的资源**，也可基于**标签对资源进行授权**。
标签键值对会严格按字符串进行解析匹配，腾讯云不会使用您设定的标签，标签仅用于您对资源的管理。
以下通过一个具体的案例来介绍标签的使用。

案例背景

某公司在腾讯云上拥有10 个 云项目，分属电商、游戏、文娱三个运营部门，服务于营销活动、游戏 A、后期制作等业务，三个部门对应的负责人为张三、李四、王五。

设置标签

为了方便管理，该公司使用标签分类管理对应的 PTS 项目资源，定义了下述标签键和值。

标签键	标签值
运营部门	电商、游戏
运营产品	广告营销中心、天涯明月刀
一级业务	资讯推荐平台、游戏运营平台
二级业务	推送流、春节推广活动
负责人	张三、李四

使用限制

最近更新时间：2024-10-31 18:11:52

标签是一个键-值对（Key-Value），您可以在 PTS 云压测控制台，通过对项目设置标签实现资源的分类管理。通过标签，可以非常方便筛选过滤出对应的资源。

数量限制

每个云资源允许的最大标签数是50。

标签键限制

- qcloud、tencent、project 开头为系统预留标签键，禁止创建。
- 只能为字母、数字、空格或汉字，支持 `+-=._:/@()[]（）【】,;><`
- 标签键长度最大为127个字符。

标签值限制

- 只支持字母、数字、空格、汉字或特殊符号。
- 标签值最大长度为255个字符。

绑定标签

最近更新时间：2024-10-31 18:11:52

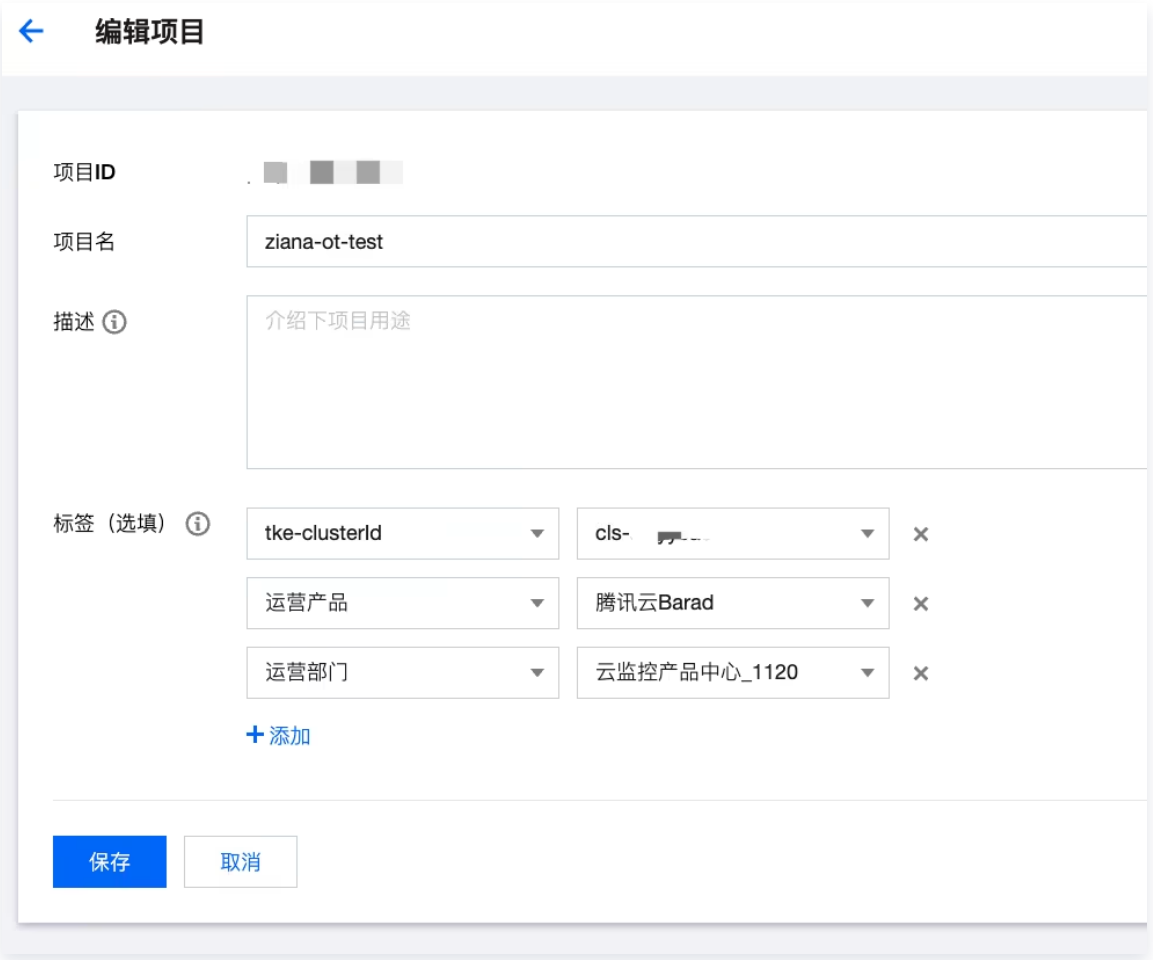
本文将为您介绍云压测项目如何绑定标签。

操作前提

已创建标签，详情可参见 创建标签 指引新建标签。

操作步骤

1. 登录 腾讯云可观测平台控制台 。
2. 在左侧菜单栏中单击**云压测 > 项目列表**，进入项目列表页，在列表中选择您需要操作的项目，单击操作列的编辑按钮。
3. 进入编辑页面后，单击**+添加**，添加完后保存即可。



使用标签

最近更新时间：2024-10-31 18:11:52

本文指导您在腾讯云可观测平台的控制台中，根据标签对实例进行资源筛选，过滤出对应的资源。

前提条件

已创建标签，若未创建可参见 创建标签 进行创建。

操作步骤

1. 登录 腾讯云可观测平台控制台 。
2. 在左侧菜单栏中单击**云压测 > 项目列表**。
3. 在项目列表右上角的搜索框，单击空白处弹出标签过滤选择框，选择**标签**，如下图所示：

4. 在标签过滤选择框中选择对应的条件，单击**确定**进行过滤。
5. 如果需要调整对应的标签条件，单击搜索框里的**标签**后面的标签内容进行编辑即可。
6. 同时也支持直接单击项目列表中对应的标签值来进行过滤，如下图所示：

错误代码手册

最近更新时间：2024-12-02 10:04:42

前言

PTS 错误码主要有两种：

- 引擎发请求包时产生的自定义错误码。
- 引擎透传的 HTTP、gRPC 等协议的错误码。

错误码只代表大致的错误类型；若要定位具体错误原因，请务必参考错误的详细信息。

引擎自定义错误码

引擎自定义的错误码，是用来进一步细分脚本可能导致的错误，例如：设置的 HTTP 超时时间不合理（默认10秒）、域名不存在等，方便您排查问题。

错误码	英文描述	错误码含义
999	unkown	通用错误码，具体原因见请求采样、施压机引擎日志。
1010	context deadline	请求超时，通常与1301状态码同时出现，均表示超时。
1100	具体的 DNS 错误信息	DNS 查询出错的通用错误码。
1101	lookup: no such host	DNS 域名解析错误。
1200	具体的 TCP 错误信息	请求在成功建立网络连接后出错的通用错误码。
1201	具体的 TCP op error 错误信息	请求出错，且该请求使用的不是 TCP 连接。
1202	connection reset by peer	请求出错，且该错误发生于数据的读取或写入场景，服务端关闭了连接，可能是服务端异常、或负载过高等原因。
1203	broken pipe	请求出错，且该错误发生于数据的写入场景，服务端关闭了连接，可能是服务端异常、或负载过高等原因。
1204	具体的系统调用错误信息	请求出错，且错误来源于系统调用。
1210	具体的 dial 错误信息	请求在建立网络连接时出错的通用错误码。
1211	unknown errno on with message	请求出错，且该错误来源于未被识别的系统调用。

1212	dial: i/o timeout	建立网络连接超时。
1301	request canceled	请求取消，通常是因为超时，且超时情况下通常与1010错误码同时出现。
2000	具体错误信息	数据库操作出错。
2001	具体错误信息	检查点结果为 False。

HTTP 协议常见错误码

单击查看官网说详细说明

状态码	状态码英文名称	中文描述
100	Continue	继续。客户端应继续其请求。
101	Switching Protocols	切换协议。服务器根据客户端的请求切换协议。只能切换到更高级的协议，例如，切换到 HTTP 的新版本协议。
200	OK	请求成功。一般用于 GET 与 POST 请求。
201	Created	已创建。成功请求并创建了新的资源。
202	Accepted	已接受。已经接受请求，但未处理完成。
203	Non-Authoritative Information	非授权信息。请求成功。但返回的 meta 信息不在原始的服务器，而是一个副本。
204	No Content	无内容。服务器成功处理，但未返回内容。在未更新网页的情况下，可确保浏览器继续显示当前文档。
205	Reset Content	重置内容。服务器处理成功，用户终端（例如：浏览器）应重置文档视图。可通过此返回码清除浏览器的表单域。
206	Partial Content	部分内容。服务器成功处理了部分 GET 请求。
300	Multiple Choices	多种选择。请求的资源可包括多个位置，相应可返回一个资源特征与地址的列表用于用户终端（例如浏览器）选择。

301	Moved Permanently	永久移动。请求的资源已被永久的移动到新 URI，返回信息会包括新的 URI，浏览器会自动定向到新 URI。今后任何新的请求都应使用新的 URI 代替。
302	Found	临时移动。与301类似。但资源只是临时被移动。客户端应继续使用原有 URI。
303	See Other	查看其它地址。与301类似。使用 GET 和 POST 请求查看。
304	Not Modified	未修改。所请求的资源未修改，服务器返回此状态码时，不会返回任何资源。客户端通常会缓存访问过的资源，通过提供一个头信息指出客户端希望只返回在指定日期之后修改的资源。
305	Use Proxy	使用代理。所请求的资源必须通过代理访问。
306	Unused	已经被废弃的 HTTP 状态码。
307	Temporary Redirect	临时重定向。与302类似。使用 GET 请求重定向。
400	Bad Request	客户端请求的语法错误，服务器无法理解。
401	Unauthorized	请求要求用户的身份认证。
402	Payment Required	保留，将来使用。
403	Forbidden	服务器理解请求客户端的请求，但是拒绝执行此请求。
404	Not Found	服务器无法根据客户端的请求找到资源（网页）。通过此代码，网站设计人员可设置"您所请求的资源无法找到"的个性页面。
405	Method Not Allowed	客户端请求中的方法被禁止。
406	Not Acceptable	服务器无法根据客户端请求的内容特性完成请求。
407	Proxy Authentication Required	请求要求代理的身份认证，与401类似，但请求者应当使用代理进行授权。

408	Request Time-out	服务器等待客户端发送的请求时间过长，超时。
409	Conflict	服务器完成客户端的 PUT 请求时可能返回此代码，服务器处理请求时发生了冲突。
410	Gone	客户端请求的资源已经不存在。410不同于404，如果资源以前有现在被永久删除了可使用410代码，网站设计人员可通过301代码指定资源的新位置。
411	Length Required	服务器无法处理客户端发送的不带 Content-Length 的请求信息。
412	Precondition Failed	客户端请求信息的先决条件错误。
413	Request Entity Too Large	由于请求的实体过大，服务器无法处理，因此拒绝请求。为防止客户端的连续请求，服务器可能会关闭连接。如果只是服务器暂时无法处理，则会包含一个 Retry-After 的响应信息。
414	Request-URI Too Large	请求的 URL 过长（ URL 通常为网址），服务器无法处理。
415	Unsupported Media Type	服务器无法处理请求附带的媒体格式。
416	Requested range not satisfiable	客户端请求的范围无效。
417	Expectation Failed	服务器无法满足 Expect 的请求头信息。
500	Internal Server Error	服务器内部错误，无法完成请求。
501	Not Implemented	服务器不支持请求的功能，无法完成请求。
502	Bad Gateway	作为网关或者代理工作的服务器尝试执行请求时，从远程服务器接收到了一个无效的响应。
503	Service Unavailable	由于超载或系统维护，服务器暂时的无法处理客户端的请求。延时的长度可包含在服务器的 Retry-After 头信息中。
504	Gateway Time-out	充当网关或代理的服务器，未及时从远端服务器获取请求。

505	HTTP Version not supported	服务器不支持请求的 HTTP 协议的版本，无法完成处理。

gRPC 协议常见错误码

点此查看官网说详细说明

英文描述	状态码	说明
OK	0	不是错误；成功后返回。
CANCELLED	1	该操作通常被调用方取消。
UNKNOWN	2	未知错误。例如，当从另一个地址空间接收的 Status 值属于此地址空间中未知的错误空间时，可能会返回此错误。此外，API 引发的错误如果没有返回足够的错误信息，则可能会转换为此错误。
INVALID_ARGUMENT	3	客户端指定的参数无效。请注意，这与 FAILED_PRECONDITION 不同。INVALID_ARGUMENT 表示无论系统状态如何都有问题的参数（例如，格式错误的文件名）。
DEADLINE_EXCEEDED	4	在操作完成之前，截止日期已过期。对于更改系统状态的操作，即使操作已成功完成，也可能返回此错误。例如，来自服务器的成功响应可能会延迟很长时间。
NOT_FOUND	5	找不到某些请求的实体（例如，文件或目录）。服务器开发人员注意：如果对整个用户类别的请求被拒绝，例如逐步推出功能或未记录的 allowlist，则可以使用 NOT_FOUND。如果某类用户中的某些用户的请求被拒绝，例如基于用户的访问控制，则必须使用 PERMISSION_denied。
ALREADY_EXISTS	6	客户端试图创建的实体（例如，文件或目录）已存在。
PERMISSION_DENIED	7	调用者没有执行指定操作的权限。permission_DENIED 不能用于因耗尽某些资源而导致的拒绝（对于这些错误，请使用 resource_EXHAUSTED）。如果无法识别调用者，则不得使用 PERMISTION_DENIED（对于这些错误，请使用 UNAUTHENTICATED）。此错误代码并不意味着请求有效，或者请求的实体存在或满足其他先决条件。
RESOURCE_EXHAUSTED	8	某些资源已用尽，可能是每个用户的配额，也可能是整个文件系统空间不足。
FAILED_PRECONDITION	9	操作被拒绝，因为系统未处于执行操作所需的状态。例如，要删除的目录是非空的，rmdir 操作应用于非目录等。服务实现者可以使用以下

		准则来决定 FAILED_PRECONDITION、ABORTED 和 UNAVAILABLE： • 如果客户端可以重试失败的调用，请使用 UNAVAIALLE。 • 如果客户端应在更高级别重试（例如，当客户端指定的测试和设置失败时，表示客户端应重新启动读-修改-写序列），请使用 ABORTED。 • 如果在显式修复系统状态之前客户端不应重试，请使用 FAILED_PRECONDITION。例如，如果“rmdir”因目录非空而失败，则应返回 FAILED_PRECONDITION，因为除非从目录中删除文件，否则客户端不应重试。
ABORTED	10	操作被中止，通常是由于并发问题，如序列器检查失败或事务中止。请参阅上面的指导原则，以确定FAILED_PRECONDITION、ABORTED和UNAVAILABLE。
OUT_OF_RANGE	11	尝试的操作超出了有效范围。 例如，查找或读取文件末尾。与 INVALID_ARGUMENT 不同，此错误表示一个问题，如果系统状态发生变化，则可以修复该问题。例如，如果要求32位文件系统以不在[0,2^32-1]范围内的偏移量进行读取，则会生成 INVALID_ARGUMENT，但如果要求以超过当前文件大小的偏移量读取，则将生成 OUT_OF_range。 FAILED_PRECONDITION 和 OUT_of_RANGE 之间有一点重叠。我们建议在应用时使用 OUT_OF_RANGE（更具体的错误），以便在空间中迭代的调用者可以很容易地查找 OUT_OF-RANGE 错误，以便在完成时进行检测。
UNIMPLEMENTED	12	此服务未实现或不支持/启用该操作。
INTERNAL	13	内部错误。这意味着底层系统预期的一些不变量已被打破。此错误代码是为严重错误保留的。
UNAVAILABLE	14	该服务当前不可用。这很可能是一种瞬态情况，可以通过回退重试来纠正。请注意，重试非幂等操作并不总是安全的。
DATA_LOSS	15	无法恢复的数据丢失或损坏。
UNAUTHENTICATED	16	请求没有该操作的有效身份验证凭据。

实践教程
使用 Prometheus 观测性能压测指标

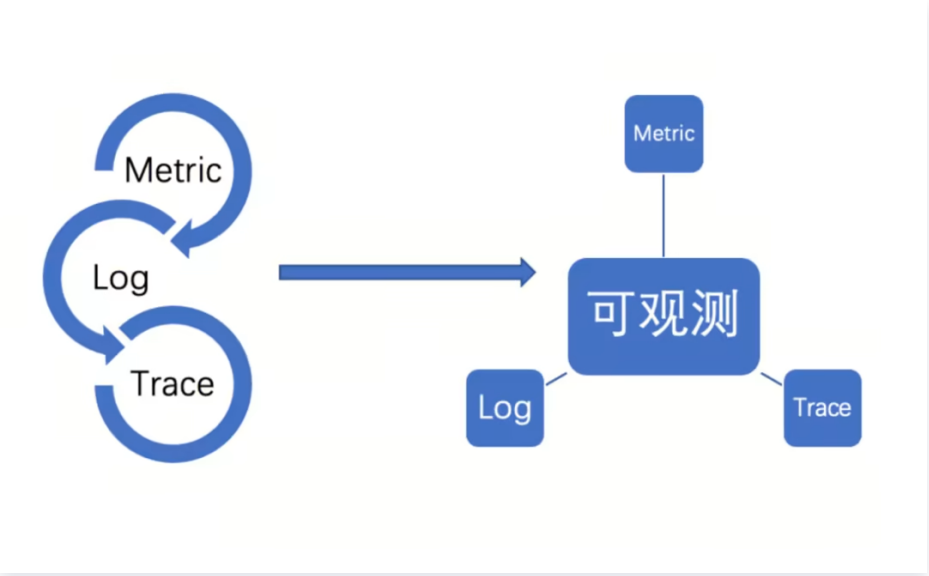
最近更新时间：2024-11-11 10:15:12

性能压测是一个充满挑战的领域，性能评估和优化贯穿开发、测试、部署、上线等各个阶段，直接影响系统运行效率和用户使用体验。本文将为您介绍在压测过程中 Prometheus 存储指标，并使用 Grafana 将指标可视化。通过观测指标的动态变化，发现系统瓶颈。

性能压测可观测

可观测体系主要包含3类指标：Metrics、Logs、Traces。三者既可独立工作，也可相辅相成，推导出系统整体的状况。

- Metrics：是一种聚合的度量数值，能够量化系统各个维度指标，常用于提供系统全局视图，一般包括 Counter、Gauge、Histogram 等指标类型。
- Logs：应用程序运行过程中产生的日志或者事件，提供系统运行的上下文信息，例如：某个变量值、发生错误详情等。
- Traces：提供请求从发送到完成响应整个链路。在分布式系统中，一个请求完成需要经过多个服务，Trace 提供请求在链路各个环节的响应时间、响应体、是否报错等。通过 Trace 更方便分析出请求中的异常环节。



指标概述

Counter：只增不减的计数器

Counter 是一种累积度量指标，用于表示一个只能增加的值，如果重置系统或服务，Counter 可能会从0开始重新计数。Counter 最常见的用途是计数器，用于记录发生的事件数量，例如请求的数量、完成的任务数、错误的数量等。例如 Counter 类型指标：http_requests_total，用于记录请求次数。

通过 rate()函数获取请求 QPS:

```
rate(http_requests_total[5m])
```

查询系统访问量前10的 http 请求:

```
topk(10, http_requests_total)
```

Gauge：有增有减的度量衡

Gauge 与 Counter 不同，Counter 用来反映事件发生次数，而 Gauge 用来反映系统当前的状态，例如当前的温度、服务器 CPU/内存使用率、剩余可用内存等。例如：通过 Gauge 指标，查看节点剩余可用内存:

```
# HELP node_memory_MemAvailable_bytes Memory information field
MemAvailable_bytes.
# TYPE node_memory_MemAvailable_bytes gauge
node_memory_MemAvailable_bytes
```

查看剩余可用内存比例（这个语句通常可用来设置警报，当剩余可用内存低于某个阈值时进行通知，这样您就可以采取行动避免系统出现内存不足的情况）:

```
(node_memory_MemFree_bytes/node_memory_MemTotal_bytes)*100
```

Histogram/Summary：分析数据分布

Histogram 和 Summary 主要用于统计和分析样本的分布情况。
以 Histogram 为例，通常使用 histogram_quantile 函数计算百分位数。例如，要计算请求响应时间的第90百分位数:

```
histogram_quantile(0.9, rate(http_response_duration_seconds_bucket[5m]))
```

这个查询会返回过去5分钟内所有记录的请求持续时间的第90百分位数。通过这种方式，Histogram 为您提供了强大的工具来监控和理解您的系统性能。

设计观测指标

可以通过性能压测暴露系统瓶颈，通过选择合适可观测工具、设计科学的指标监控，根据指标变化来帮助您定位系统瓶颈。

核心指标

压测过程中，从施压方看主要需要关注以下核心指标：

- 请求响应时间：包括平均响应时间、响应时间百分比水位。
- 请求 RPS
- 请求成功率，失败率。
- 请求错误原因
- 压测并发数
- 检查点成功数，失败数。
- 请求发送接收字节数
- 施压机内存/CPU 使用率等

指标维度

每个指标最好能够区分不同的维度，例如：

1. 按返回码统计不同请求的 QPS，例如返回码为200的请求的 QPS 是多少，返回码为500的请求的 QPS 是多少?
2. 统计发往不同服务的请求的 QPS，例如压测 `www.test1.com` 和 `www.ok1.com` 的请求的 QPS 分别是多少?

以 http 请求为例，常见的维度划分如下：

- job：压测任务标识，每一次压测都是一个不同的 job。
- method：请求方法，例如 GET、POST、HEAD 等。
- proto：请求协议，例如 http、https、http2。
- service：请求地址，例如 `https://www.test1.com` 。
- status：请求响应码，例如200、404、500等。
- result：标识请求响应码，例如200或者其他小于400对应 OK，404对应 Not Found，500对应 Internal Error 。
- check：检查点/断言名字，一般用于简单描述该断言的作用。

指标设计

基于此我们设计了一套最基本、常用的指标体系，能够覆盖绝大部分用户的使用需求，详情请参见下表。您也可以基于如下指标体系进行拓展，以满足不同的需求。

Metric	Type	Labels	Description
req_total	Counter	job,method, proto,service, status, result	请求次数
req_duration_seconds	Histogram	job,method, proto,service, status,	请求耗时

		result	
checks_total	Counter	job, check, result	检查点
send_bytes_total	Counter	job,method, proto,service, status, result	发送字节数
receive_bytes_total	Counter	job,method, proto,service, status, result	接收字节数
num_vus	Gauge	job	并发数，可用于指代虚拟用户数量。如果是 JMeter压测，也可用于指代线程数

示例

查看实时的并发用户数：

```
sum(num_vus{job=~"$job"})
```

查看请求的 QPS：

```
# 查看不同请求的qps
sum(rate(req_total{job=~"$job"}[1m])) by (service)
# 查看成功请求的qps
sum(rate(req_total{job=~"$job",result="ok"}[1m]))
# 查看指定http://www.test1.com服务的qps
sum(rate(req_total{job=~"$job",service="http://www.test1.com"}[1m]))
```

查看请求失败率：

```
# 请求总体失败率
sum(rate(req_total{job=~"$job",result!="ok"}
[1m]))/sum(rate(req_total{job=~"$job"}[1m]))
# 基于请求维度，查看各请求的失败率
sum(rate(req_total{job=~"$job",result!="ok"}[1m])) by
(service)/sum(rate(req_total{job=~"$job"}[1m])) by (service)
```

查询请求响应时间：

```
# 查询请求平均响应时间
sum(rate(req_duration_seconds_sum{job=~"$job"}
[15s]))/sum(rate(req_duration_seconds_count{job=~"$job"}[15s]))
# 查询请求中位数（50百分位）响应时间
histogram_quantile(0.50,
sum(rate(req_duration_seconds_bucket{job=~"$job"}[1m])) by (le))
# 查询请求90百分位响应时间
histogram_quantile(0.90,
sum(rate(req_duration_seconds_bucket{job=~"$job"}[1m])) by (le))
```

查询请求出入带宽：

```
# 请求出带宽汇总
sum(rate(send_bytes_total{job=~"$job",region=~"$region"}[1m]))
# 请求入带宽汇总
sum(rate(pts_engine_receive_bytes_total{job=~"$job",region=~"$region"}
[1m]))
```

检查点成功率（用户自定义的 test 语句，在JMeter 中对应断言）：

```
# 检查点成功率
sum(rate(checks_total{job=~"$job", result="ok"}
[1m]))/sum(rate(checks_total{job=~"$job"}[1m]))
# 指定检查点成功率
sum(rate(checks_total{job=~"$job", result="ok",check="response contains
hello"}[1m]))/sum(rate(checks_total{job=~"$job", check="response
contains hello"}[1m]))
```

说明：

- 用户也可以基于以上 demo 进行灵活扩充，对被压测服务进行同样的监控。在压测过程中，对比查看压测平台生成的指标报告与用户服务的指标报告，综合分析排查问题。
- 使用 Prometheus 存储以上指标，使用查询语句在 Prometheus 中查询数据，最后通过 Grafana 可视化展示以上数据。一边压测，一边实时观测服务性能指标变化。

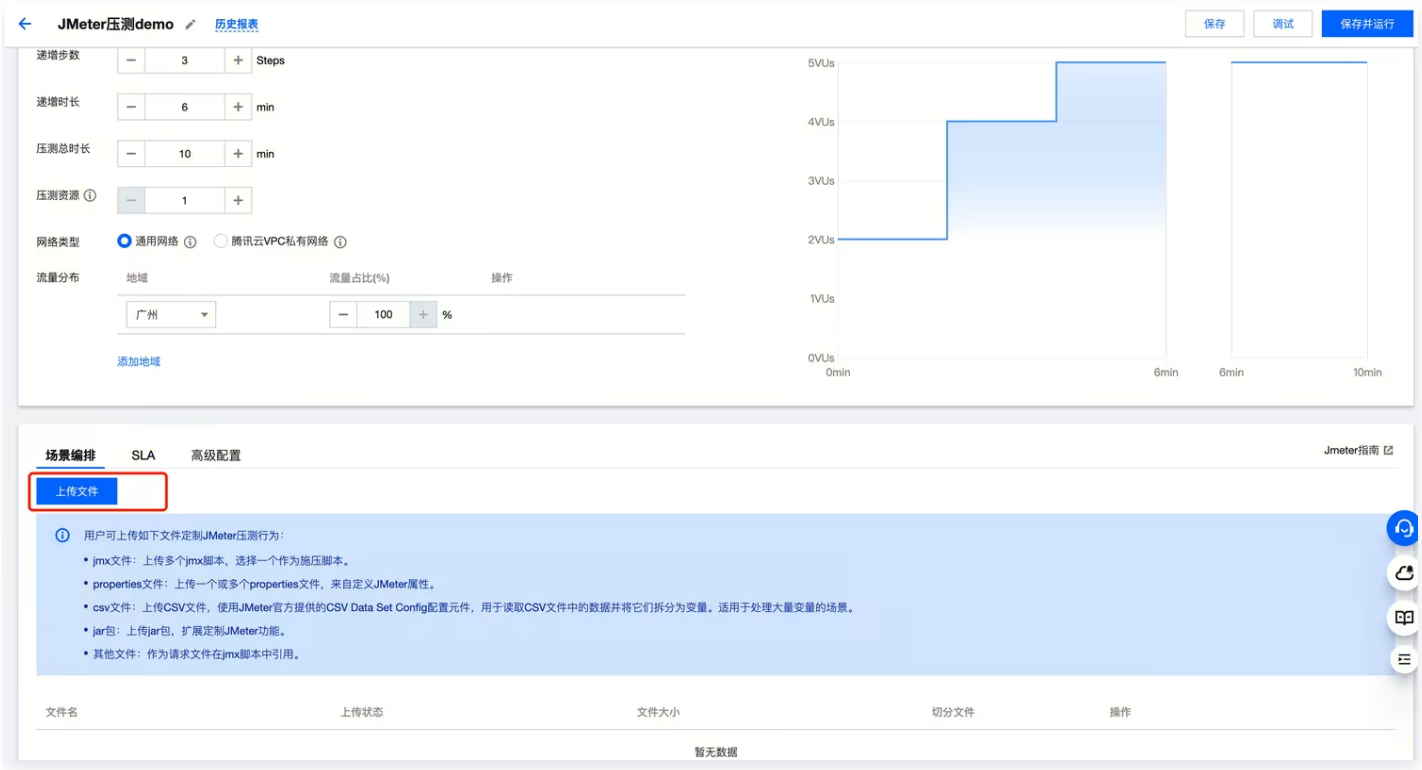
操作步骤

1. 登录 腾讯云可观测平台 。

2. 在左侧菜单栏中单击**云压测 > 测试场景**。
3. 在测试场景页面单击**新建场景**。
4. 在创建测试场景页面选择“JMeter”压测类型，并单击**开始**，创建压测场景。

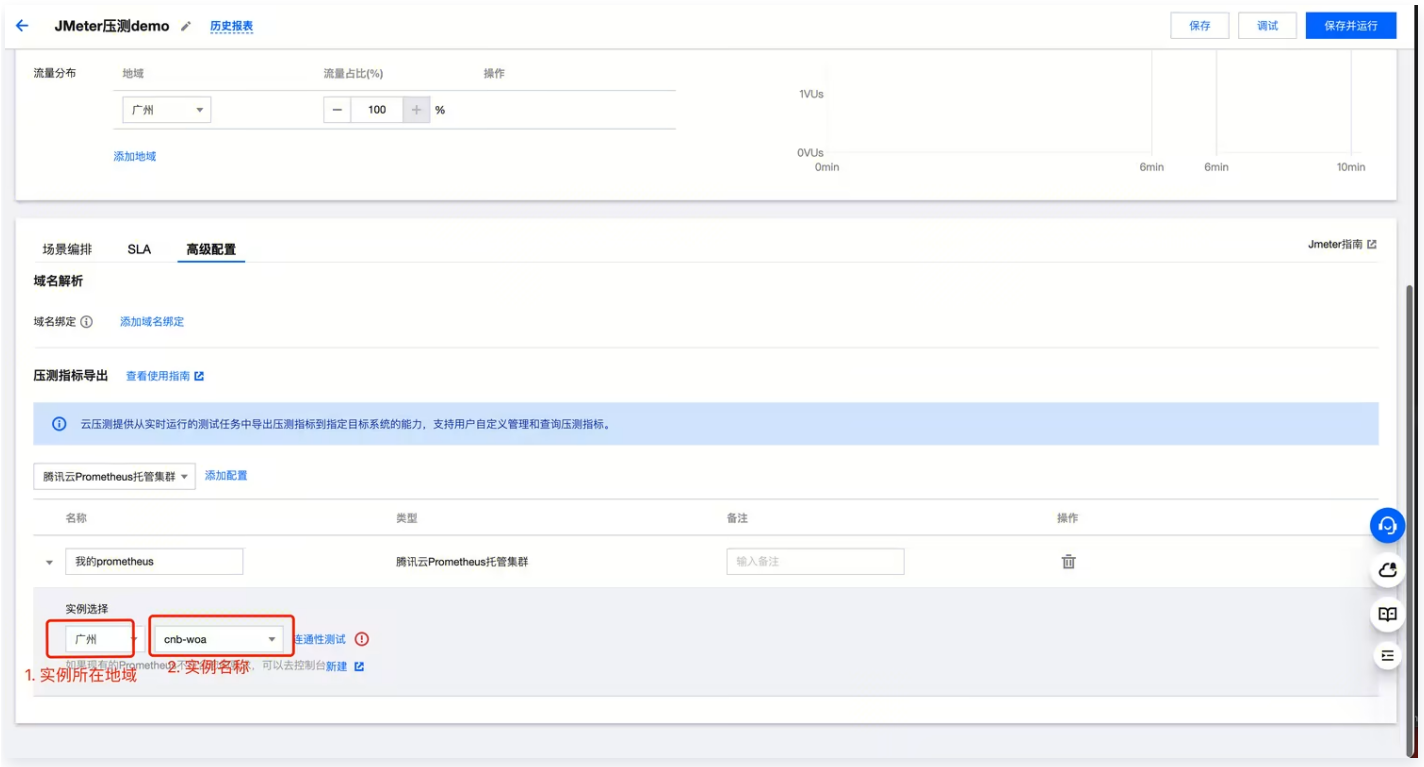
上传 jmx 脚本

- 必选：上传 jmx 脚本
- 可选：
  - 上传 Jar 包：如果您的脚本中使用了 JMeter 三方插件，您可以上传对应 jar 包，来拓展 JMeter 功能。
  - 上传 properties 文件：在原生 jmeter.properties 文件基础上，自定义 JMeter 属性。
  - csv 文件：在 Jmeter 中读取 csv 文件中的数据，作为变量在脚本中引用。
  - 其他文件：任何在 jmx 中脚本引用的其他文件



将报告导出到腾讯云 Prometheus

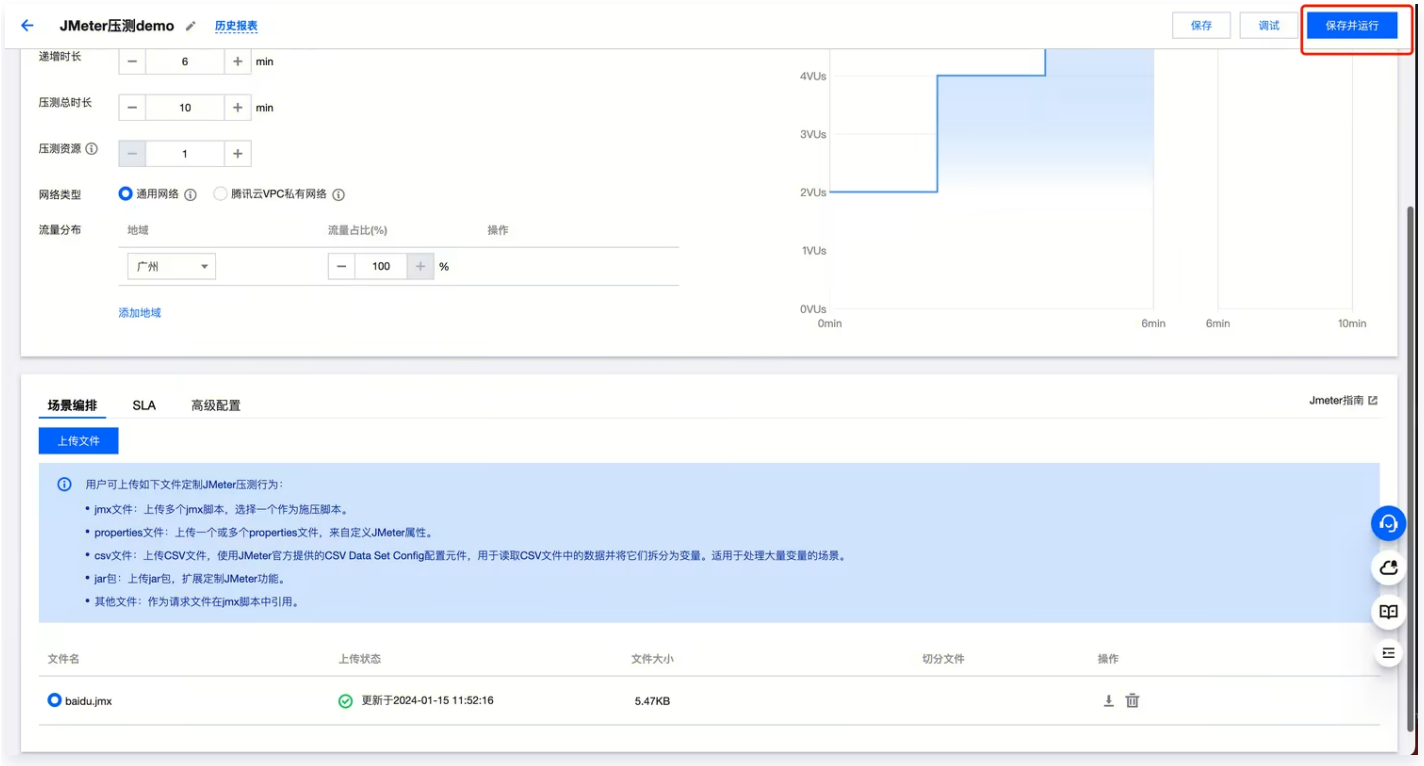
在**高级配置 > 压测指标导出 > 腾讯云 Prometheus 托管集群**中，单击**添加配置**， 选择您的实例：



如果您在该地域没有实例，您也可以单击**新建**。

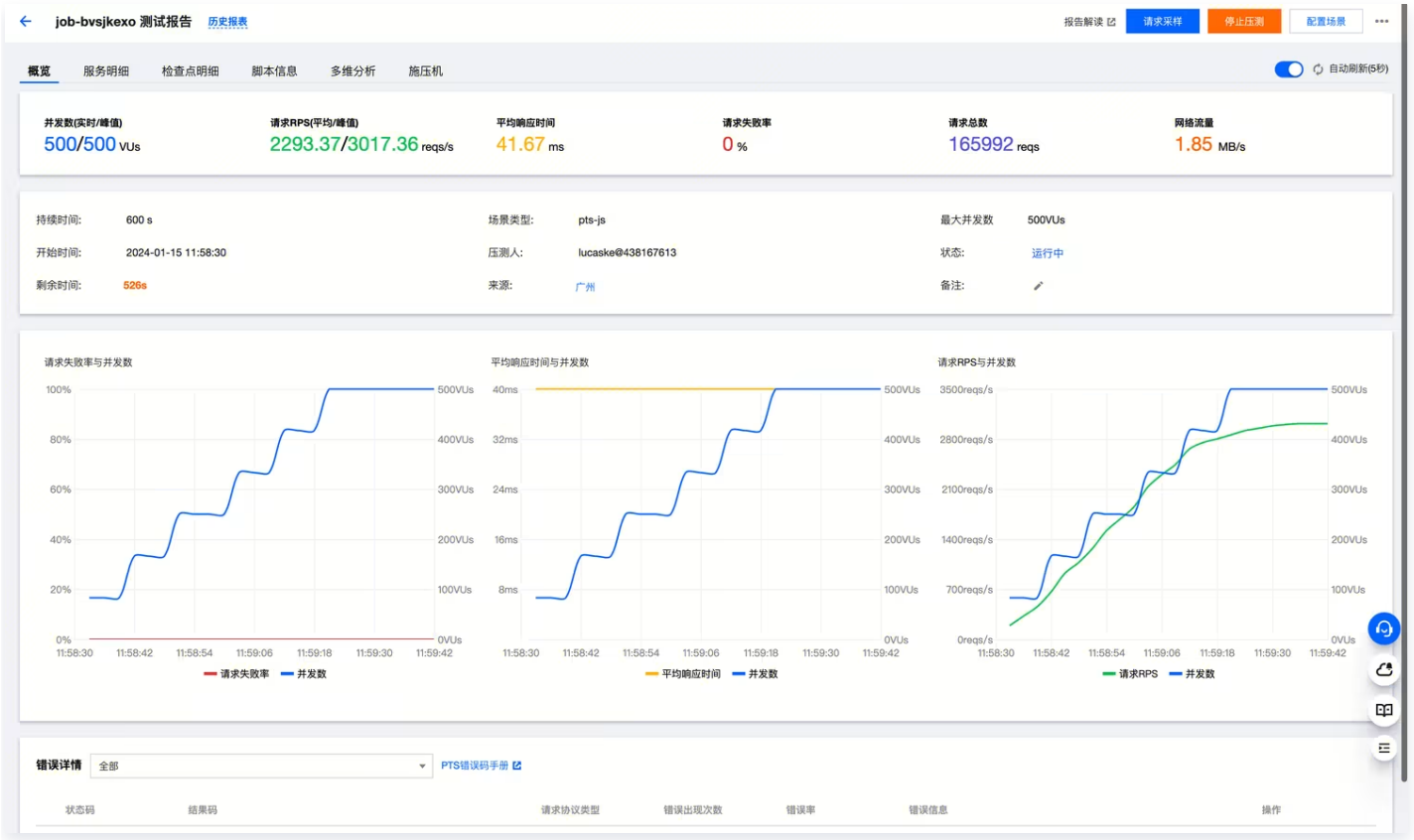
运行压测脚本

单击**保存并运行**即可开始压测。



查看实时报告

您可以选择不同的页签，查看不同维度的报告详情。

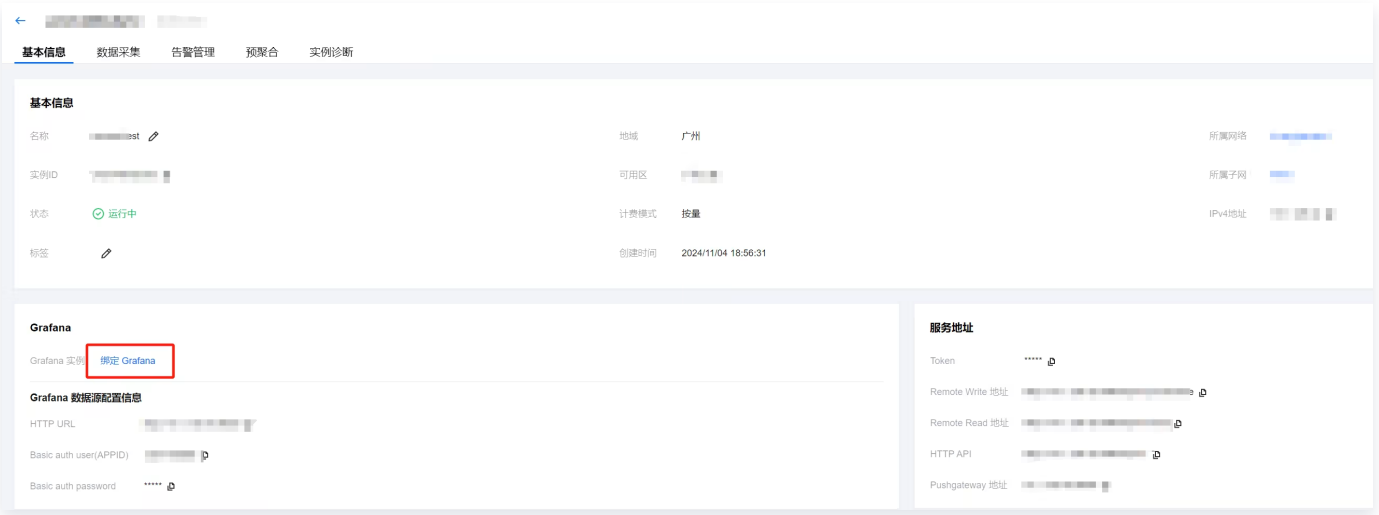


在 Prometheus 中查看压测报告

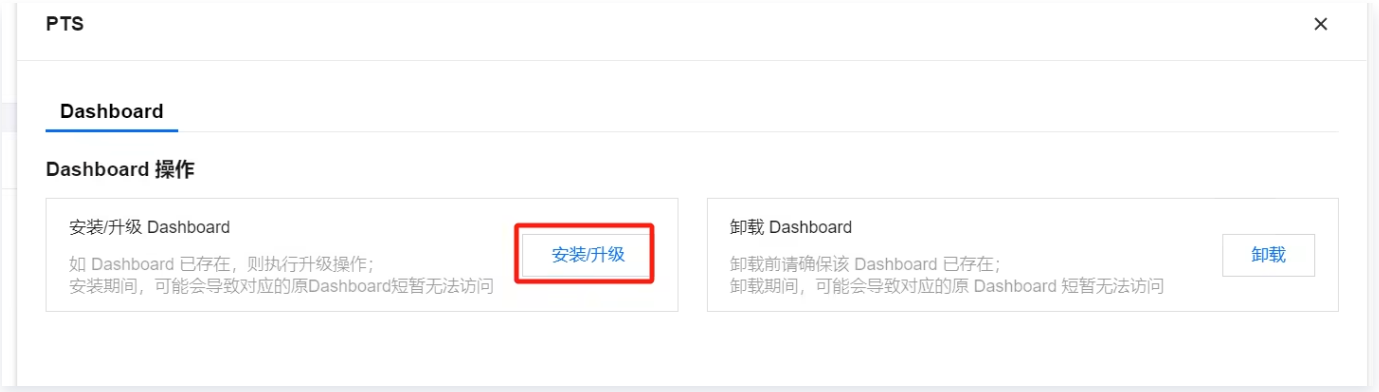
在压测过程中，指标会同步发送到腾讯云 Prometheus 中。您也可以在 Prometheus 中查询指标信息，自定义查询语句。

1. 登录 Prometheus 控制台，在搜索栏根据实例名称找到您的 Prometheus 实例。

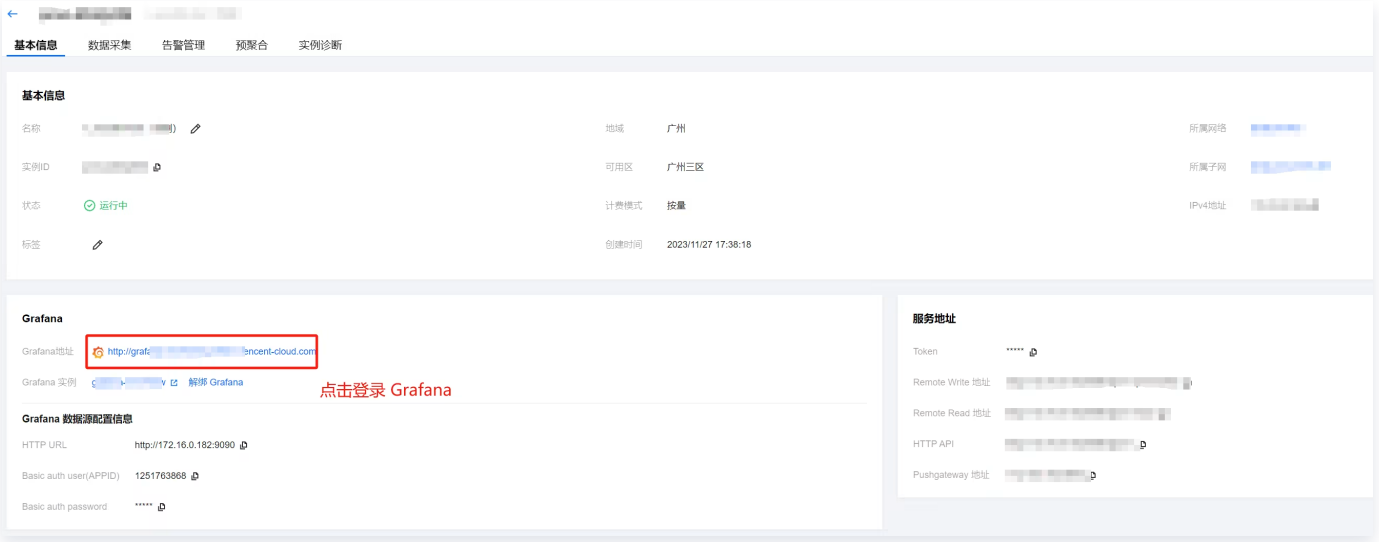
2. 单击实例 ID，进入实例详情。
   如果您的 Prometheus 实例没有对应的 Grafana，可以单击绑定 Grafana。通过 Grafana 来展示 Prometheus 指标。



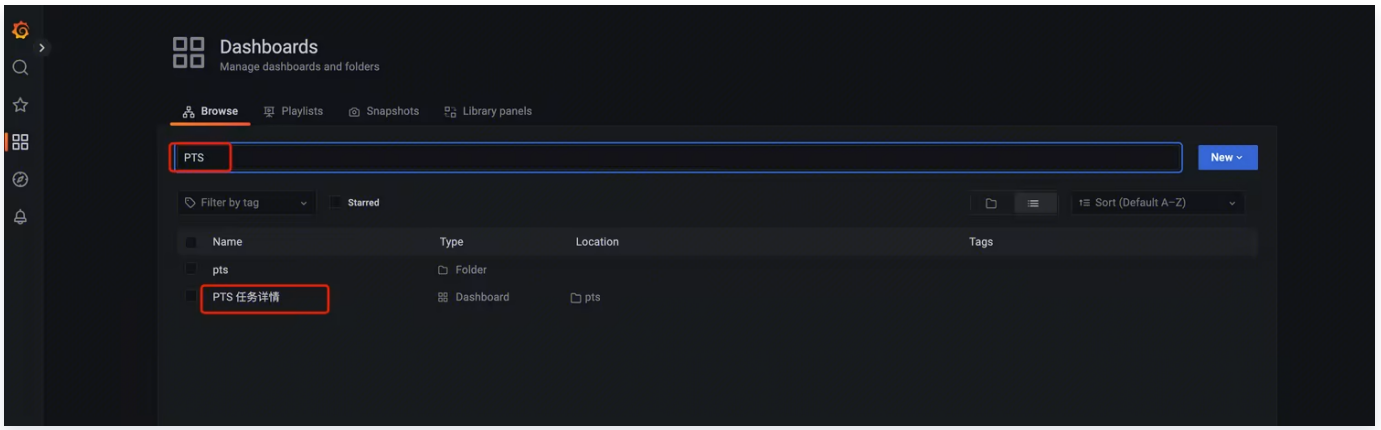
3. 选择**数据采集 > 集成中心**，搜索云压测 PTS 应用，单击它即会弹出一个窗口，单击 **Dashboard > 安装/升级**， 将云压测监控面板安装到 Prometheus 绑定的 Grafana 上。



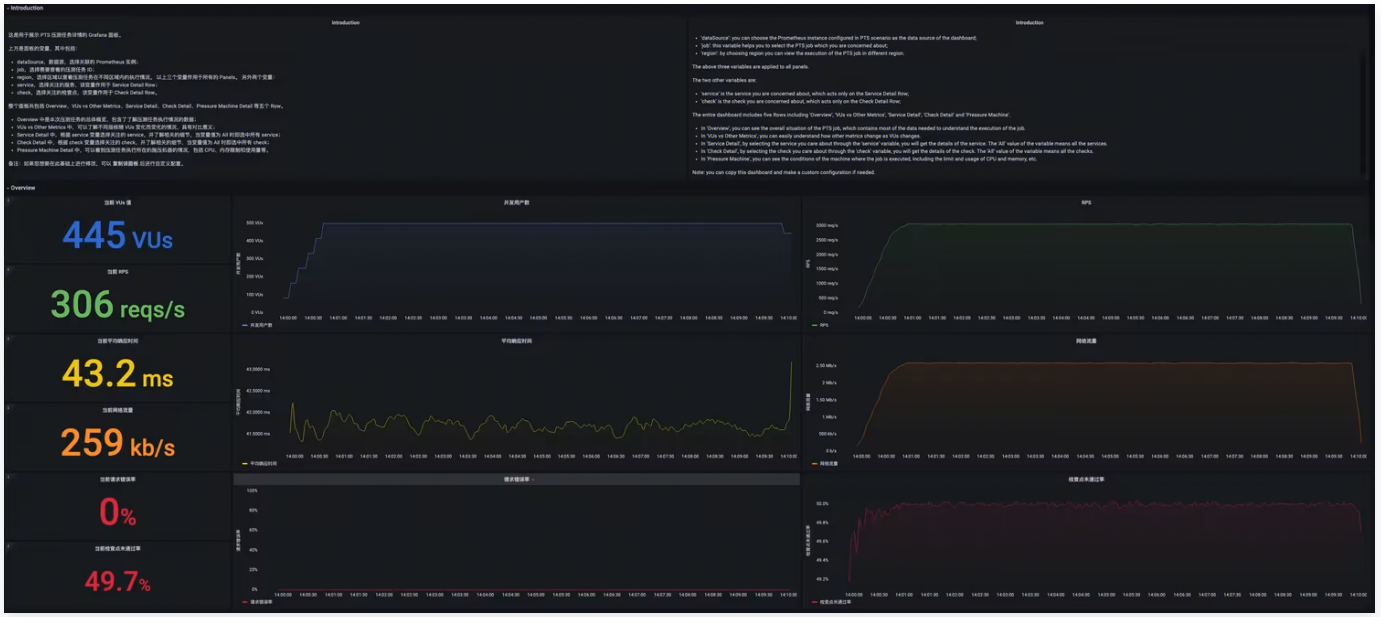
4. 在 Grafana 中查看压测报告，登录 Prometheus 绑定的 Grafana 页面。

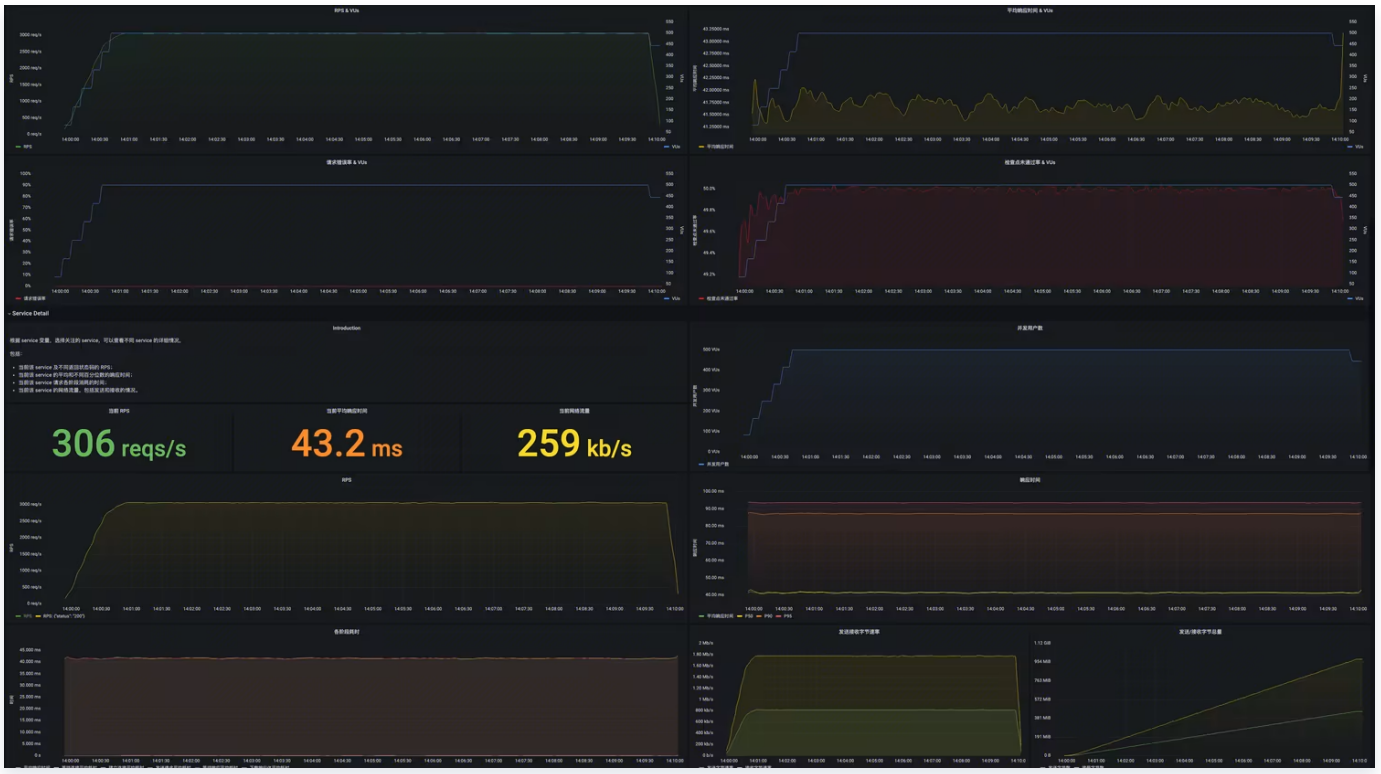


5. 搜索 PTS，查看云压测压测报告。



6. 单击 PTS 任务详情，查看报告详情。





压测时如何评估系统瓶颈

在压测时，主要关注系统吞吐量（RPS，网络带宽）、响应时间、并发用户数等。公式如下：

RPS = VU（并发数）/ 平均响应时间

如何理解压测公式?

以一个线程循环去执行某个请求为例：如果请求平均响应时间是10ms，那么一个并发每秒可执行100个请求，对应的 RPS = 100req/s。
从这个公式可以推导，如果想提升压测的总体 RPS，有以下几种方法：

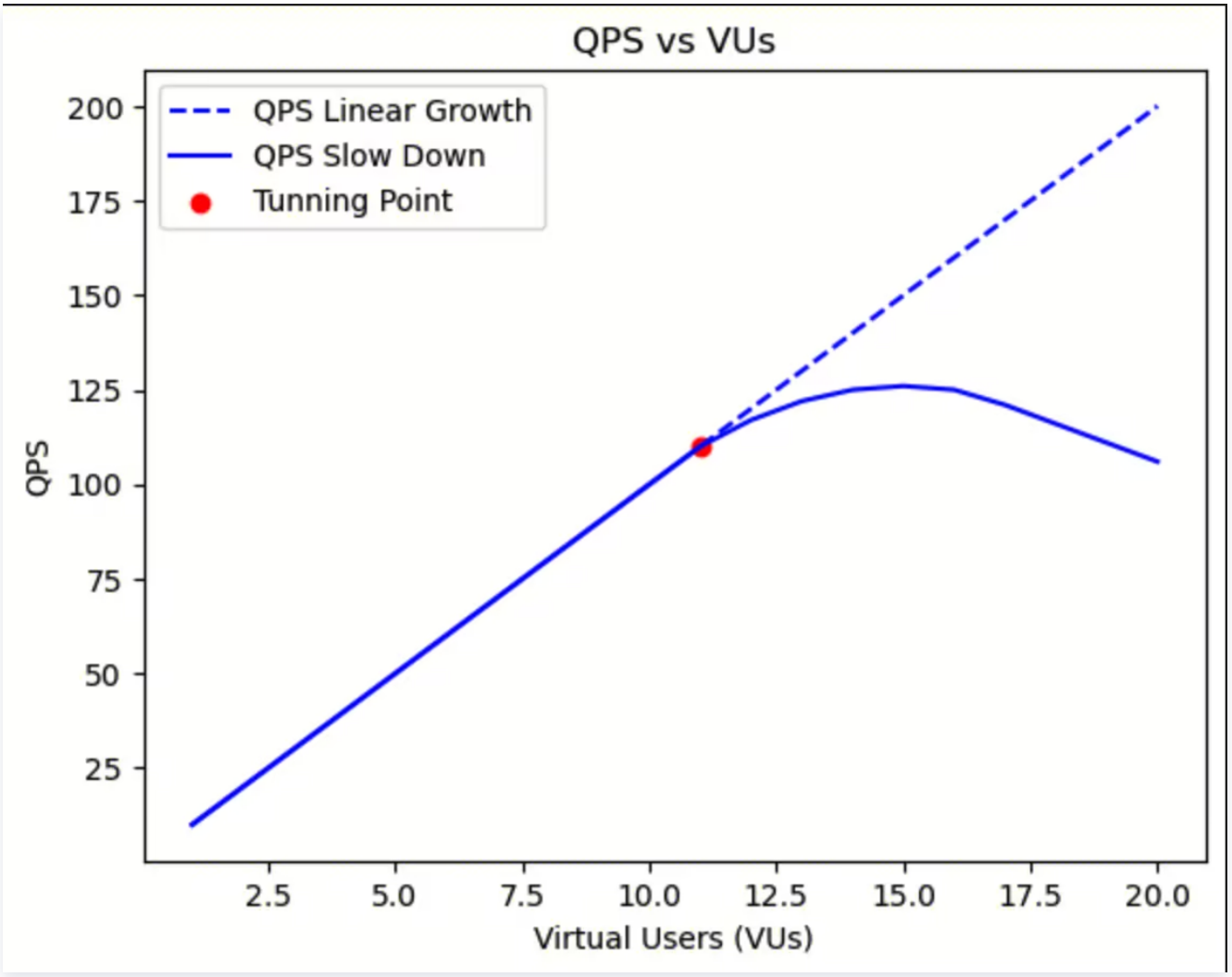
- 增加并发数，且请求平均响应时间保持不变。
  这种情况就是被压服务还没有饱和，压测 RPS 随着并发压力的增加而增加。
- 降低请求平均响应时间， VU 保持不变。
  这种情况比较理想，请求平均响应时间降低，代表被压服务进行了优化，单个 VU 单位时间内能够发送更多的请求。

在现实压测中，很可能随着 VU 增加，被压系统压力增加，响应时间也随之增加。

- 如果响应时间增加系数小于 VU 增加系数，总体 RPS 还是在变大，系统还未达到瓶颈。
- 如果响应时间增加系数大于 VU 增加系数，压测的表现就是随着 VU 增大，总体 RPS 反而降低，此时系统已经达到瓶颈。

评估系统性能拐点

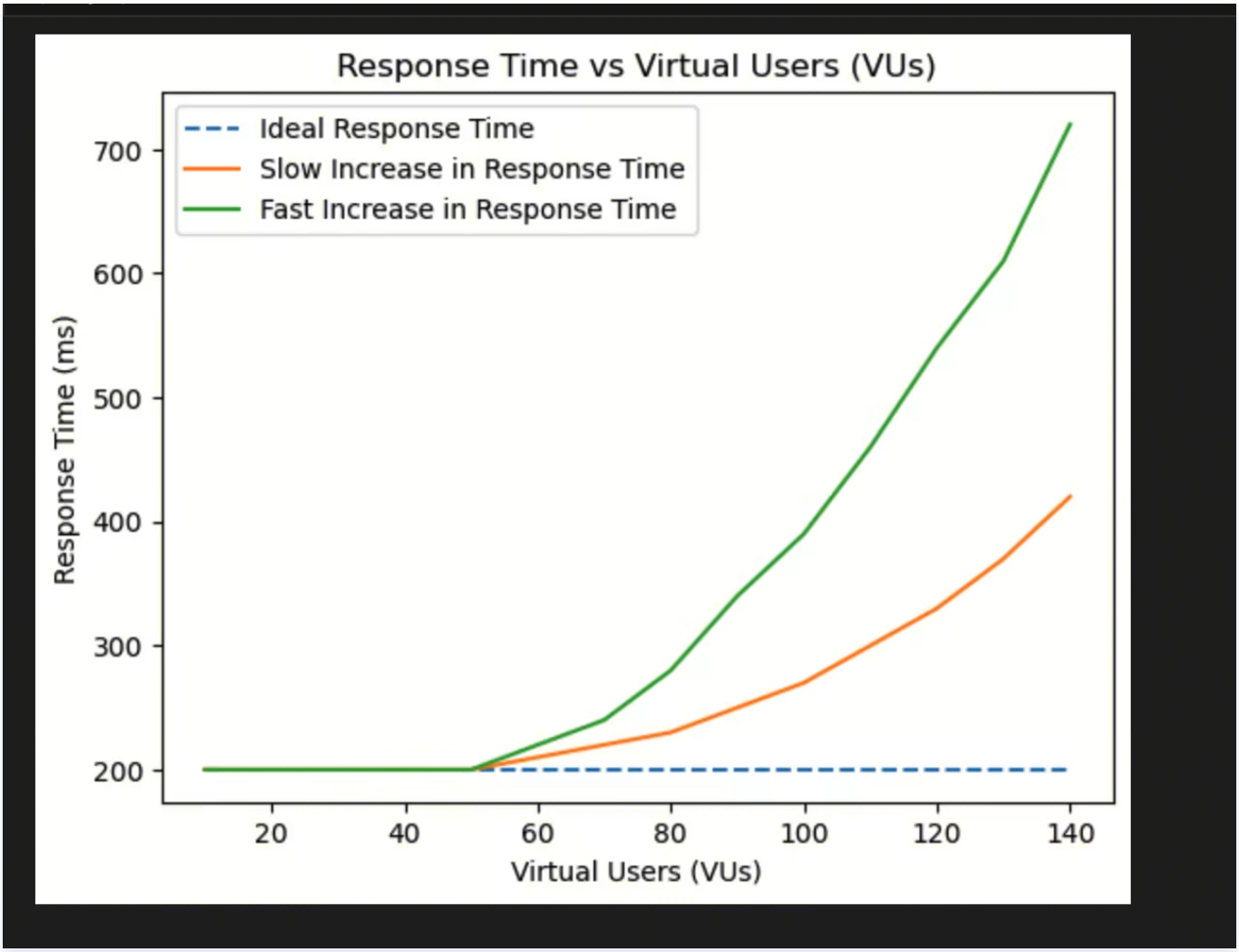
压测就是在压力不断增加情况下，找到业务系统扩展性的拐点，即系统瓶颈/最大容量。
在一定的阶段，我们观测到扩展性是线性变化的。到达某一点时，此时对于资源的争夺开始影响性能。这一点也可以认为是系统拐点。作为曲线分界，过了拐点，整体的吞吐量会随着资源争夺加剧偏离线性扩展。最终，资源争夺的开销反而导致完成的请求数变少，吞吐量反而下降。



这种情况可能在系统负载到达 100% 使用率（饱和点）之后发生，也可能在接近100% 使用率的时候，这个时候排队比较明显。

例如：有一个计算密集型系统，在更多请求进来时候，需要更多的线程来执行请求。当CPU使用率接近100%时，由于CPU调度延时增加，性能开始下降。在性能达到峰值后，整体吞吐量反而会随着更多线程加入而下降。线程加入会导致更多上下文切换，消耗CPU资源，实际完成的任务反而变少。

性能的非线性变化，我们也可以通过响应时间的变化来看出来：



性能下降的原因非常多，除了上面提到的频繁上下文切换外，还有如下原因：

- 系统内存不够，开始频繁的换页（swap）来补充内存。
- 随着系统磁盘 IO 增加，磁盘 IO 可能进入缓冲排队。
- 系统内部实现队列算法，来进行削峰操作，导致请求处理等待时间变长。

使用云压测回放 GoReplay 录制的请求

最近更新时间：2024-12-02 10:04:42

前言

本文将通过一个实例演示如何使用 GoReplay 录制 Nginx 网关接收到的请求，并将请求各个字段保存成 CSV 文件。在云压测中，通过上传 CSV 参数文件，指定期望的并发数，分布式回放请求到用户指定的地址。

GoRePlay 简介

GoReplay 是一个开源的流量录制回放工具。主要用于捕获实时流量并将其复制到测试环境中。
由于 GoReplay 本身并不提供一个分布式运行方案，只能在单机上运行。在流量录制完成后，受限于单机资源瓶颈，我们很难大规模的重放录制的流量，无法 有效的模拟真实用户流量的压测行为以及极限测试。而腾讯云云压测是一款分布式性能测试服务，可模拟海量用户的真实业务场景。因此我们可以引入云压测，使用云压测来回放 GoReplay 录制的真实流量。

常见 GoReplay 使用场景

- 性能测试：通过复制生产环境的流量到测试环境，可以在不影响真实用户的情况下对应用程序进行压力测试和性能评估。
- 故障排除和调试：当生产环境出现问题时，可以捕获相关的流量并在一个隔离的环境中重放，以便开发人员可以安全地调试问题而不会影响实际服务。
- 回归测试：在发布新版本之前，可以使用 GoReplay 捕获的流量来验证更改是否会引入新的错误或性能问题。
- A/B 测试：可以将流量同时发送到两个服务版本，比较它们的表现，以便做出数据驱动的决策。
- 通过在回放时候，加大回放请求的倍数，模拟高流量情况，可以帮助确定在不同负载下所需的资源量。

GoReplay 流量录制原理

GoReplay 流量录制是监听指定端口流量，录制成 gor 文件（或者发送到其他目的端），方便后续回放。

```
sudo gor --input-raw :8080 --output-file requests.gor
```

开始录制回放用户 Nginx 网关

本文以录制回放 Nginx 网关为例，其他所有类型的网关都可以按照相同的方式来录制请求，然后使用云压测来回放用户请求。

环境准备

- Nginx 网关：Nginx 网关上有源源不断的用户请求，需要在 Nginx 网关录制下这些请求。
- GoReplay：请求录制回放工具。

安装 GoReplay 至网关所在机器上，如果网关所在机器是 Linux 或 macOS，可以使用以下命令：

```
  # 从官方 GitHub 仓库下载最新的二进制文件
curl -L
https://github.com/buger/goreplay/releases/download/1.3.3/gor_1.3.3_x6
4.tar.gz | tar xz

# 将二进制文件移动到你的PATH目录中，例如/usr/local/bin
mv gor /usr/local/bin/
```

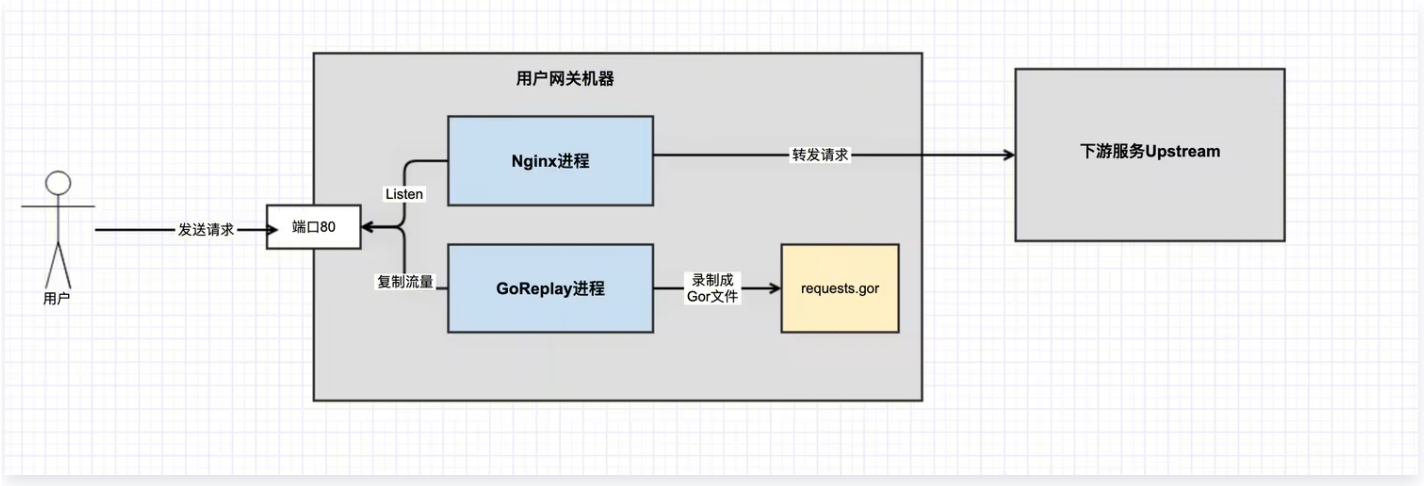
> **说明：**
> 确保替换上面的 URL 中的版本号为最新的版本，仓库地址：
> https://github.com/buger/goreplay 。

- CSV 生成服务：接收 HTTP 请求，将接收到的请求各个字段写入 CSV 文件中。
- 云压测：基于用户上传的 CSV 文件，回放用户录制的所有请求。

实验流程

将 Nginx 上的请求录制成 Gor 文件

本节参与组件（其他组件仅做完整场景展示）：Nginx 网关、GoReplay。
整体架构图如下：



开始录制流量前，需要在网关所在服务器上运行 GoReplay。以下是一个基本的命令示例，它会监听网关上的80端口，并将捕获的流量保存到一个文件中：

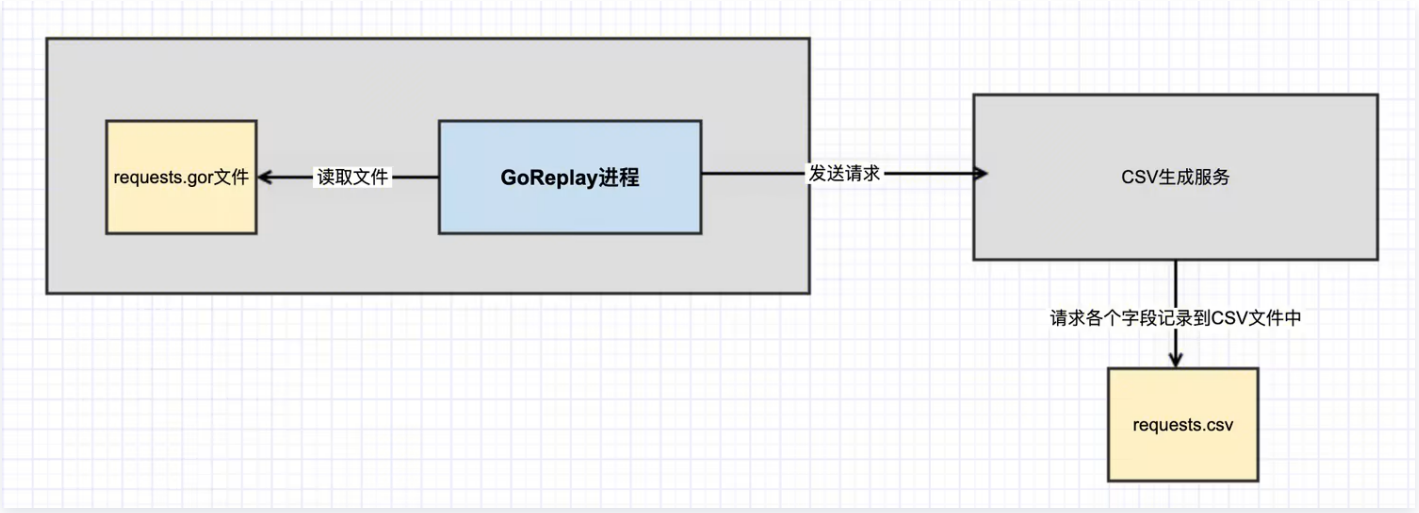
```
sudo gor --input-raw :80 --output-file requests.gor
```

这个命令会捕获所有通过端口80的流量，并将其保存到当前目录下的 requests.gor 文件中。

> **注意：**
> 需要 sudo 权限来监听80端口。

将 Gor 文件转换成 CSV 参数文件

本节使用 GoReplay 回放 Gor 文件中记录的请求到 CSV 生成服务。参与组件：GoReplay、CSV 生成服务。整体架构如下：



在 CSV 文件中会记录下请求各个字段，例如 scheme、host、uri、method、base64Body。下面是一个简单 CSV 文件示例：

scheme	host	uri	method	jsonHeaders	base64Body
http	mockhttpbin.pts.svc.cluster.local	/get?page=1	get	{"name":"kk"}	-
http	mockhttpbin.pts.svc.cluster.local	/post	post	{"Hello": 'world',}	dGhpcyBpcyBnb29kCg==

```
scheme,host,uri,method,jsonHeaders,base64Body
http,mockhttpbin.pts.svc.cluster.local,/get?page=1,get,{"name":"kk"},
http,mockhttpbin.pts.svc.cluster.local,/post,post,
{"hello":"world"},dGhpcyBpcyBnb29kCg==
```

> **说明：**
> 使用 base64Body，而不是直接记录 body 是因为有些请求的 body 发送的二进制文件，直接写入 CSV 文件会展示成乱码。写成 base64 后，方便后续转换成原来的结构体。

CSV 生成服务代码

编写服务代码，并保存为 main.go 文件。

```
package main

import (
    "encoding/base64"
    "encoding/csv"
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "os"
)

func main() {
    http.HandleFunc("/", requestHandler) // 设置处理函数
    log.Println("Server starting on port 8080...")
    log.Fatal(http.ListenAndServe(":8080", nil))
}

func requestHandler(w http.ResponseWriter, r *http.Request) {
    // 获取请求信息
    scheme := "http" // 默认为http，因为Go的http包不支持直接获取scheme
    if r.TLS != nil {
        scheme = "https"
    }
    host := r.Host
    uri := r.RequestURI
    method := r.Method

    // 将headers转换为JSON格式
    headersJson, err := json.Marshal(r.Header)
    if err != nil {
        http.Error(w, "Error converting headers to JSON",
http.StatusInternalServerError)
        return
    }
```

```
    // 读取请求体
    body, err := ioutil.ReadAll(r.Body)
    if err != nil {
        http.Error(w, "Error reading request body",
http.StatusInternalServerError)
        return
    }
    defer r.Body.Close()

    // Base64编码请求体
    base64Body := base64.StdEncoding.EncodeToString(body)

    // 写入CSV文件
    record := []string{scheme, host, uri, method, string(headersJson),
base64Body}
    err = writeToCSV(record)
    if err != nil {
        http.Error(w, "Error writing to CSV",
http.StatusInternalServerError)
        return
    }

    // 发送响应
    fmt.Fprintf(w, "Request logged")
}

func writeToCSV(record []string) error {
    file, err := os.OpenFile("requests.csv",
os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0666)
    if err != nil {
        return err
    }
    defer file.Close()

    writer := csv.NewWriter(file)
    defer writer.Flush()

    return writer.Write(record)
}
```

编译并运行 CSV 生成服务

1. 将上述文件保存成 main.go 文件
2. 直接运行代码。

```
go run main.go
```

回放流量到 CSV 生成服务上，用来生成 CSV 文件。

```
gor --input-file requests.gor --output-http "http://csv-server-
address:8080" --http-original-host true
```

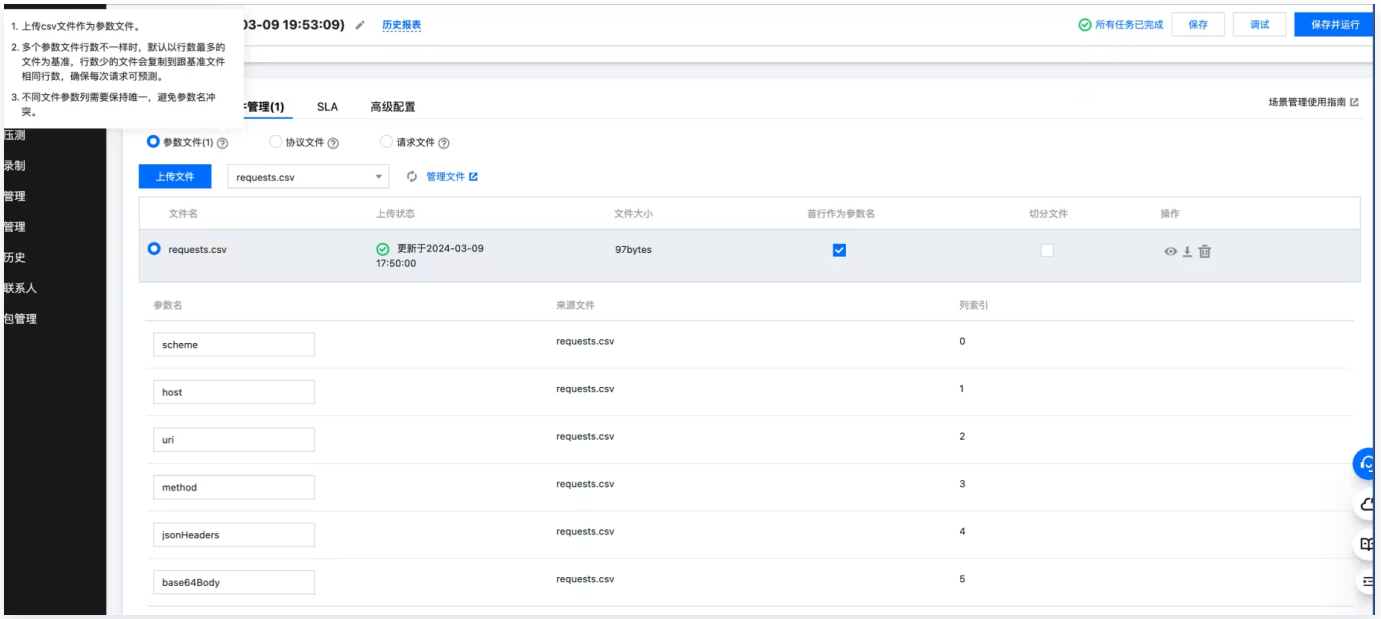
这个命令会读取 requests.gor 文件中的流量，并将其回放到CSV生成服务上，CSV 生成服务默认会将接收到的请求写成 requests.csv 文件里；且生成的流量 host 为请求原本的 host 而非 CSV 服务的地址。

在云压测上使用 CSV 参数文件回放请求

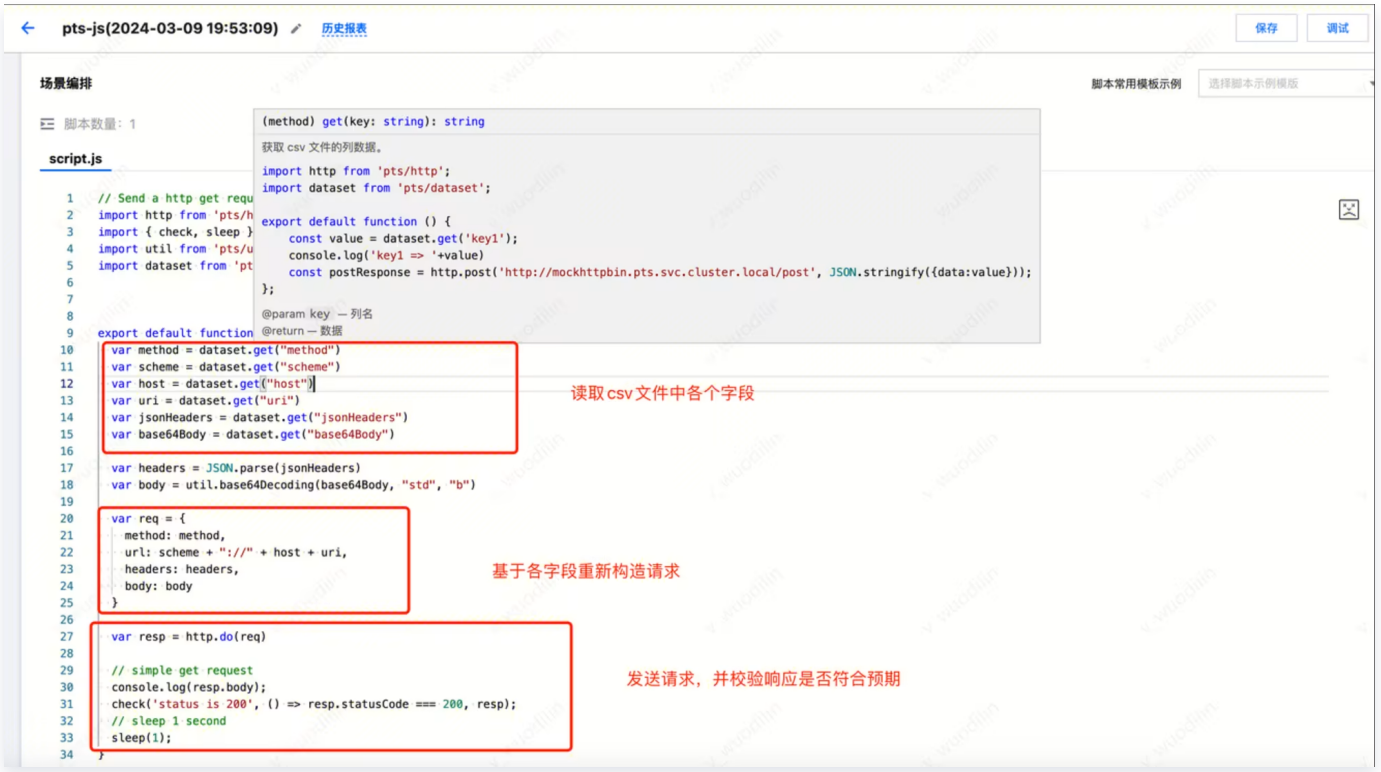
云压测支持用户上传 CSV 文件作为参数文件。您可以动态引用其中的测试数据，供脚本里的变量使用。这样，当施压机并发执行这段代码，每条请求能动态、逐行获取 CSV 里的每行数据，作为请求参数使用。参数文件具体用法可参见 使用参数文件 。

1. 登录 云压测控制台 ，云压测对于首次使用的用户提供一个免费的压测资源包。
2. 在左侧导航栏中选择**测试场景**，单击**新建场景**，选择**脚本模式**。

   云压测脚本模式支持原生 JavaScript ES2015(ES6)+ 语法，并提供额外函数，帮助您在脚本模式下，快速编排压测场景。您可在控制台的在线编辑器里，用 JavaScript 代码描述您的压测场景所需的请求编排、变量定义、结果断言、通用函数等逻辑。（详细的 API 文档请参见：PTS API ）。
3. 上传之前录制的 CSV 文件，作为参数文件。



4. 编写压测脚本，施压机每次执行压测脚本时候，读取 CSV 文件中下一行，利用 CSV 文件中记录的字段重新构造出原始请求。

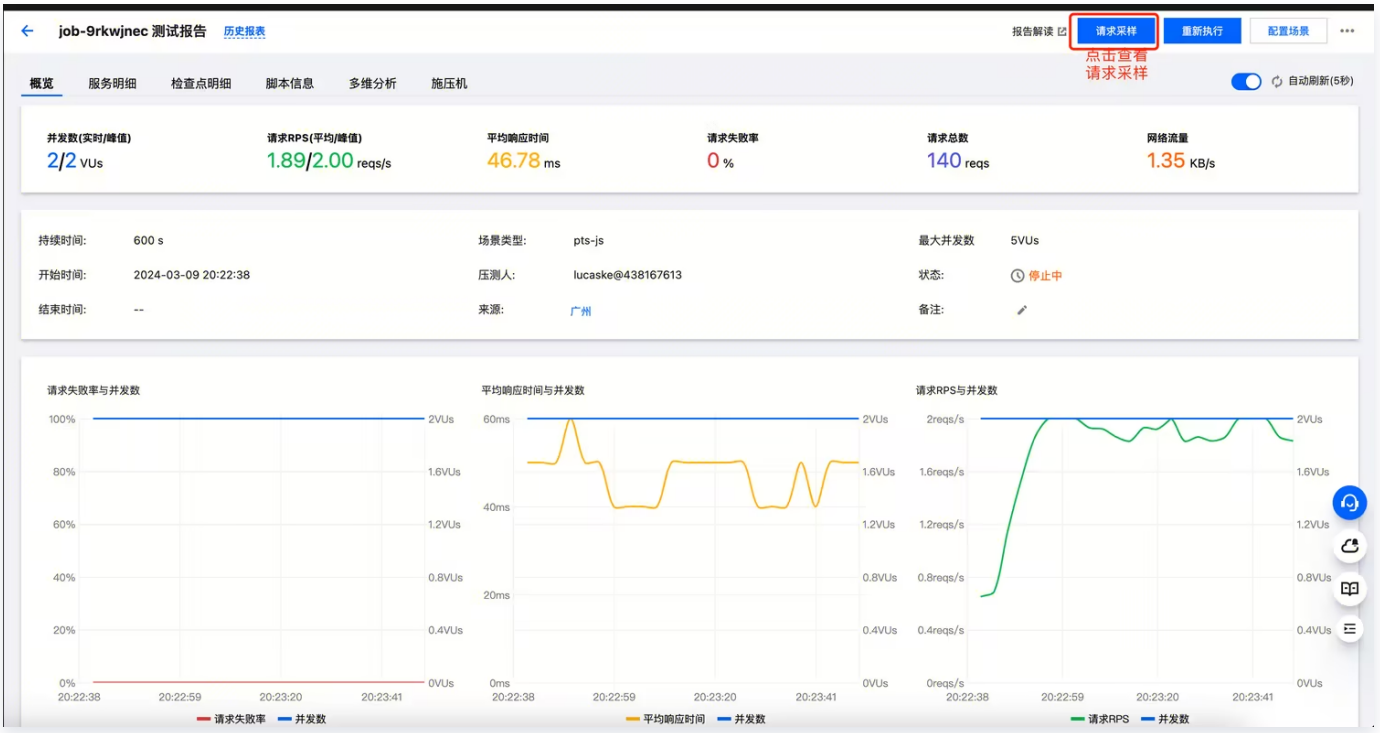


压测脚本如下：

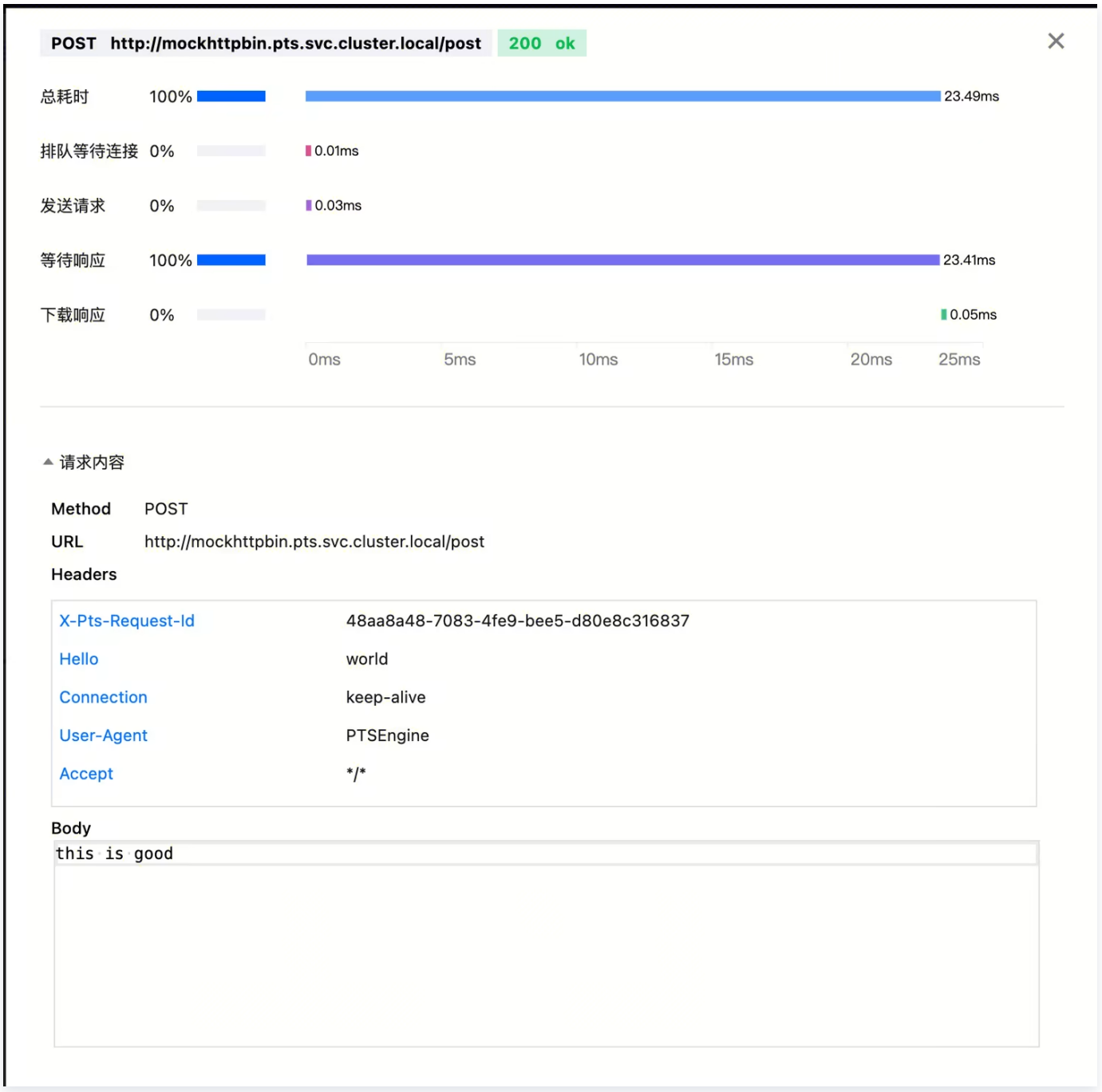
```
// Send a http get request
import http from 'pts/http';
import { check, sleep } from 'pts';
import util from 'pts/util';
import dataset from 'pts/dataset';


export default function () {
  // 读取csv文件各个字段
  var method = dataset.get("method")
  var scheme = dataset.get("scheme")
  var host = dataset.get("host")
  var uri = dataset.get("uri")
  var jsonHeaders = dataset.get("jsonHeaders")
  var base64Body = dataset.get("base64Body")

  var headers = JSON.parse(jsonHeaders)
  var body = util.base64Decoding(base64Body, "std", "b")
```

```
  // 构造请求
  var req = {
    method: method,
    url: scheme + "://" + host + uri,
    headers: headers,
    body: body
  }

  // 发送请求
  var resp = http.do(req)

  // simple get request
  console.log(resp.body);
  check('status is 200', () => resp.statusCode === 200, resp);
  // sleep 1 second
  sleep(1);
}
```

5. 点击**保存并运行**，即可运行压测脚本，回放流量。查看压测报告及**请求采样**，观察请求是否符合预期。



请求采样：


POST http://mockhttpbin.pts.svc.cluster.local/post 200 ok
总耗时 100% 23.49ms
排队等待连接 0% 0.01ms
发送请求 0% 0.03ms
等待响应 100% 23.41ms
下载响应 0% 0.05ms
0ms 5ms 10ms 15ms 20ms 25ms
请求内容
Method POST
URL http://mockhttpbin.pts.svc.cluster.local/post
Headers
X-Pts-Request-Id 48aa8a48-7083-4fe9-bee5-d80e8c316837
Hello world
Connection keep-alive
User-Agent PTSEngine
Accept */*
Body
this is good

JavaScript API 列表
JavaScript API 列表概述

最近更新时间：2024-10-22 14:10:21

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：2025-10-31 16:33:21

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
Load	全局参数配置 Option 中的 load 配置

open

最近更新时间：2023-05-17 10:10:08

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：2023-05-17 10:10:08

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： • toString()，返回字符串； • 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 14:11:02

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： • toString()，返回字符串 • 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：2023-05-09 17:33:19

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：2023-05-09 17:33:19

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：2023-05-17 10:10:09

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：2023-05-17 10:10:09

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：2024-06-26 11:12:51

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：2023-05-17 10:10:09

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：2023-05-09 17:33:19

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,
  }
}
```

Load

最近更新时间：2025-10-31 16:33:21

Load 是全局参数配置 Option 中的 load 配置。用于控制参数文件 dataset 的消费策略与压测迭代行为，例如在数据集耗尽时是否停止，以及全局目标迭代总数等。

字段

字段	类型	描述
stopAfterDataConsumption	boolean	是否在数据集耗尽后停止
targetIterations	number	压测脚本总迭代次数

样例

使用 Load Option：

```
export const option = {
  load: {
    stopAfterDataConsumption: true,
    targetIterations: 1000
  }
}
```

pts/http
模块概览

最近更新时间：2025-01-03 14:11:02

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：2023-05-17 10:10:09

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 14:11:02

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：2023-05-17 10:10:09

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：2024-06-19 11:04:52

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：2023-05-17 10:10:10

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：2023-05-17 10:10:10

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：2024-06-19 11:04:52

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：2023-05-17 10:10:10

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：2023-05-17 10:10:10

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：2023-05-17 10:10:10

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：2023-05-17 10:10:10

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：2023-05-17 10:10:10

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：2023-05-17 10:10:11

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：2023-05-17 10:10:11

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：2023-05-17 10:10:11

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：2023-05-17 10:10:11

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：2023-05-17 10:10:11

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：2023-05-17 10:10:11

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：2023-05-17 10:10:11

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：2023-05-17 10:10:11

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：2024-09-09 18:18:21

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：2023-05-17 10:10:12

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：2023-05-17 10:10:12

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 15:32:42

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：2023-05-17 10:10:12

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：2023-05-17 10:10:12

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：2023-05-17 10:10:12

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：2024-06-19 11:04:52

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：2023-05-17 10:10:12

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：2023-05-17 10:10:13

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：2023-05-17 10:10:13

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：2023-05-17 10:10:13

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：2023-05-17 10:10:13

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：2023-05-17 10:10:13

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：2023-05-17 10:10:13

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：2023-05-17 10:10:13

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：2023-05-17 10:10:14

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 14:11:02

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：2023-05-17 10:10:14

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：2023-05-17 10:10:14

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：2023-05-17 10:10:14

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：2023-05-17 10:10:14

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：2023-05-17 10:10:14

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：2023-05-17 10:10:14

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：2023-05-17 10:10:15

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：2023-05-17 10:10:15

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：2024-06-20 10:39:01

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 14:11:02

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：2023-05-17 10:10:15

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：2024-06-19 11:04:52

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 11:04:53

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：2023-05-17 10:10:15

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 11:04:53

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：2023-05-17 10:10:16

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：2023-05-17 10:10:16

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：2023-05-17 10:10:16

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：2023-05-17 10:10:16

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：2023-05-17 10:10:16

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：2023-05-17 10:10:16

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：2023-05-17 10:10:16

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：2023-05-17 10:10:17

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：2023-05-17 10:10:17

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：2023-05-17 10:10:17

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：2023-05-17 10:10:17

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：2024-06-19 11:04:52

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：2023-05-17 10:10:17

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：2023-05-17 10:10:17

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：2023-05-17 10:10:17

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 14:11:02

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：2023-05-17 10:10:17

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：2024-06-19 11:04:52

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：2023-05-17 10:10:18

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：2023-05-17 10:10:18

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：2023-05-17 10:10:18

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：2023-05-17 10:10:18

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：2023-05-17 10:10:18

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：2023-05-17 10:10:18

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：2023-05-17 10:10:18

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：2023-05-17 10:10:18

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：2023-05-17 10:10:19

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：2024-06-19 11:04:52

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：2024-06-24 17:37:21

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：2023-05-17 10:10:19

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：2023-05-17 10:10:19

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：2023-05-17 10:10:19

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：2023-05-17 10:10:19

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：2023-05-17 10:10:19

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：2023-05-17 10:10:19

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：2023-05-17 10:10:19

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：2023-05-17 10:10:19

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 14:11:02

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：2024-06-20 17:35:41

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：2024-06-20 17:35:41

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：2025-07-10 17:42:21

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：2023-05-17 10:10:20

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：2023-05-17 10:10:20

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：2023-05-17 10:10:20

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：2023-05-17 10:10:20

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：2023-05-09 17:33:25

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：2023-05-17 10:10:21

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：2024-06-19 11:04:52

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：2023-05-17 10:10:21

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：2024-12-27 11:58:52

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： • open：建立连接 • close：关闭连接 • message：接受文本消息 • binaryMessage：接受二进制消息 • 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：2023-05-17 10:10:21

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：2023-05-17 10:10:21

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： • open，建立连接； • close，关闭连接； • message，接受文本消息； • binaryMessage，接受二进制消息； • pong，接收 pong 消息； • 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：2023-05-17 10:10:21

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：2023-05-17 10:10:22

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：2023-05-17 10:10:22

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：2023-05-17 10:10:22

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：2023-05-17 10:10:22

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：2023-05-17 10:10:22

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:52

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 11:04:53

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 11:04:53

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 11:04:53

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 11:04:53

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 11:04:53

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 11:04:53

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 11:04:53

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：2024-06-24 11:46:11

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 14:11:02

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：2024-06-24 11:46:11

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：2024-10-22 14:10:21

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：2024-06-24 11:46:11

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：2024-10-21 22:06:01

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：2024-10-22 14:10:22

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：2024-06-24 17:37:21

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：2024-10-22 14:10:22

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：2024-10-22 14:10:22

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：2024-10-22 14:10:22

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 14:11:03

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：2024-10-22 14:10:22

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：2024-10-22 14:10:22

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：2024-06-19 11:04:52

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();
}
```

常见问题

最近更新时间：2024-11-01 14:55:01

常用术语

VU（并发用户数）

VU（Virtual User）：虚拟用户数。用来模拟真实场景中，在同时执行操作的用户数量，所以也叫“并发用户数”。

- VU 代表了施压端向被压端施压的能力。
- 压测系统通常用一个线程实现一个 VU，每个 VU 重复执行压测脚本。因此，当多线程/多 VU 并发时，就能模拟真实场景中，多个用户同时执行操作的情形。
- 每个 VU 执行脚本的次数：一般靠压测时长和迭代次数来规定，任一参数达到上限即停止。例如：压测时长为 1 小时，则每个 VU 在1小时内持续反复执行脚本，直到1小时结束。（在 PTS 里，支持配置时长，暂不支持配置迭代次数。）
- VU 跟真实用户的区别：一个 VU 执行完一次脚本，会继续重复执行。其关注点不在于代表某个固定的真实用户，而在于跟其他 VU 一起，在每个时刻模拟出足够的并发用户数量。也即，施压端会按照施压配置，在相应的时刻，保证满足所配置的 VU 数量、对被压端产生足够压力。
- VU 跟在线用户的区别：在线用户不一定在做操作；而 VU 一定在做脚本里的相关操作，持续不断地给被压端造成压力。

在 PTS 中，VU 的数值是在场景的施压配置中提前设置好的。

- 并发模式下：直接设置 VU，可按时间梯度递增。
- RPS 模式下：1 个压测资源 = 500VU。

RT（响应时间）

从客户端发出请求，到客户端完全接收服务器响应的时间消耗。
为了衡量 RT 指标，施压端会采集一个时间窗口内的所有请求从发出到收到响应的耗时，再聚合计算这批数据，得到多种维度的特征值，例如：平均值、最大值、最小值、分位值（50/90/95/99 百分位）。
在 PTS 压测报告中：

- 概览里的响应时间，是以压测任务的整个时长为时间窗口、以平均值为特征值，计算整个压测任务期间所有请求的平均响应时间。
- 各个实时曲线里的响应时间，是以一个很小的时间窗口随着时间轴移动、以平均值为特征值，模拟计算压测任务期间的各个时刻，实时的平均响应时间。

RPS（每秒请求数）

RPS（Requests per Second），每秒请求数，也叫“吞吐量”。在 PTS 中，有两处用到了 RPS：

- 施压配置 > RPS 模式下的调速参数，用于配置 PTS 每秒发出请求的上限；
- 压测报告里的 RPS 性能指标，用于体现 PTS 收到服务端响应的速度。

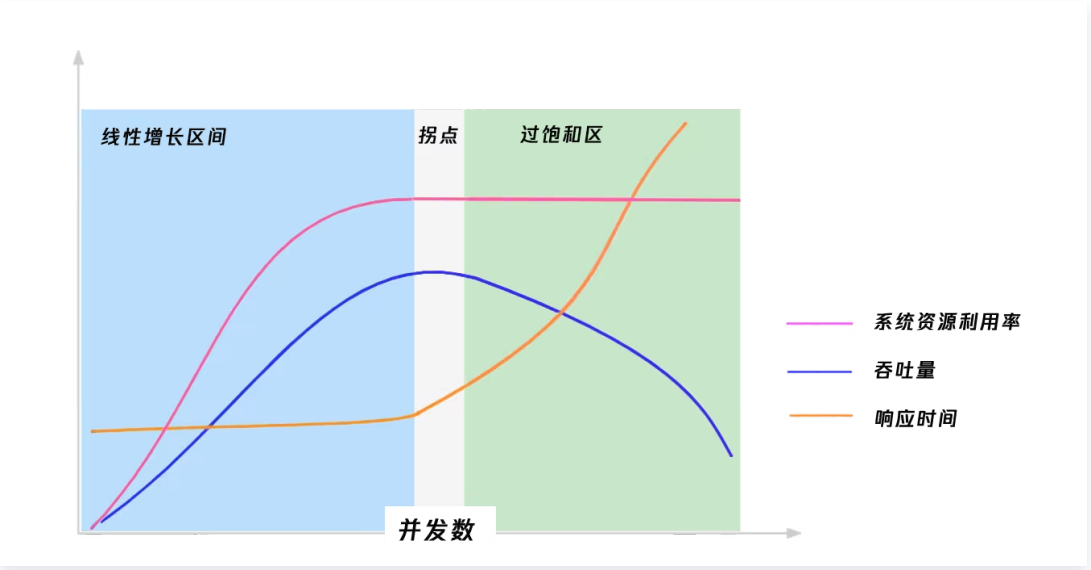
详细解释如下：

- 在绝大多数情况下，压测领域的 RPS 是指收响应的速度，用作性能指标。（若将“发请求+收响应”定义为一个“事务”，则也可将该指标称作 TPS/Transaction Per Second/每秒事务数。）
  - 在施压端统计每秒收到响应的请求数，来反映被压系统的处理能力。
  - 每秒响应结束的请求数，包括施压端作为客户端正常收到服务端响应的请求、以及主动结束的请求。
  - 在 PTS 中，您可观察压测报告里的 RPS 指标，得知 RPS 的概览值和实时值：
    - 概览里的 RPS，是以压测任务的整个时长为时间窗口，计算整个压测任务期间的 RPS。
    - 各个实时曲线里的 RPS，是以一个很小的时间窗口随着时间轴移动，模拟计算压测任务期间的各个时刻，实时的 RPS。
  - RPS 的值跟 VU 和 RT 密切相关，详见下文对三者关系的描述。
- 在 PTS 中，除了上述反映被压系统处理能力的 RPS 指标，还存在一个控制施压端每秒发出请求数的 RPS 调速参数。
  - 在 RPS 模式的施压配置中，起始 RPS/最大 RPS/动态调速 RPS，都属于 RPS 调速参数。
  - PTS 实现该 RPS 调速参数的方式，是在发请求时配置了限流。所以该参数本质上是每秒发出请求数的上限。
  - 在配置 RPS 调速参数时，PTS 会自动调整压测资源数（1 压测资源 = 500 VU），来保证施压端有能力在每秒发出足够的请求。

受限于被压系统的处理能力是否平稳、网络状况是否平稳、带宽资源是否充足等条件，客户端发请求的速度上限，不一定等于客户端收到服务端响应的速度。因此，施压时配置的 RPS 调速参数，不一定等于呈现在压测报告里的 RPS 性能指标。

VU、RPS、RT 的关系

VU = RPS × RT（也即：并发数 = 吞吐量 × 响应时间）
此公式基于 Little 定律得出。Little 定律的完整表述是：在系统的稳定状态下（尚未达到系统资源过载的拐点、响应时间基本稳定、到达系统的 RPS = 离开系统的 RPS），系统中平均同时服务的用户数量 = 用户请求到达系统的速度 × 每个用户请求平均在系统中等待的时间。
例如：假设系统某接口的响应时间为100ms，那么在 1 秒内，施压端的 1 个 VU 能连发10个请求并获得响应，反映了被压端的系统吞吐量是10个请求每秒（RPS 为 10）；那么，当同时做操作的用户数翻了100倍，也即100个 VU 同时并发施压，如果被压接口的响应耗时仍为100ms，则在 1 秒内，每个 VU 都能发送10次请求并获得响应，反映了被压端在 1 秒内处理了 1000 个请求，也即 RPS 达到 1000。
以上换算建立在被压系统表现稳定、响应时间保持不变的理想状况下。然而实际上，随着并发数增大、系统负载升高，被压接口的响应时间不一定能保持在100 ms，而可能呈现以下的增大趋势：



1. 刚开始为 “线性增长区”，此时响应时间（RT）基本稳定，吞吐量（RPS）随着并发用户数（VU）的增加而增加，三者关系符合 Little 定律：VU = RPS × RT。
2. 随着 VU 增大、系统的资源利用率饱和，系统到达 “拐点”，若继续增大 VU，响应时间开始增大，RPS 开始下降。
3. 继续增加 VU，系统超负荷、进入过饱和区，此时响应时间急剧增大、RPS 急剧下降。

失败率

一批请求中结果出错的请求所占比例，以校验响应结果是否符合期望。（不同系统对错误率的要求不同，但一般不超出千分之六，即成功率不低于99.4%。）

- PTS 通过统计一批请求中失败响应码所占比例，来计算请求失败率。响应码大于或等于 400，视为请求失败。（其中包含 PTS 端认为被压端不可达而主动取消请求的情况，相关响应码详见 错误代码手册 。）
- 请求失败率不包含检查点断言失败的情况（检查点情况参见检查点明细）。

在 PTS 压测报告中：

- 概览里的失败率，是以压测任务的整个时长为时间窗口，计算整个压测任务期间所有请求的失败率。
- 各个实时曲线里的失败率，是以一个很小的时间窗口随着时间轴移动，模拟计算压测任务期间的各个时刻，实时的失败率。

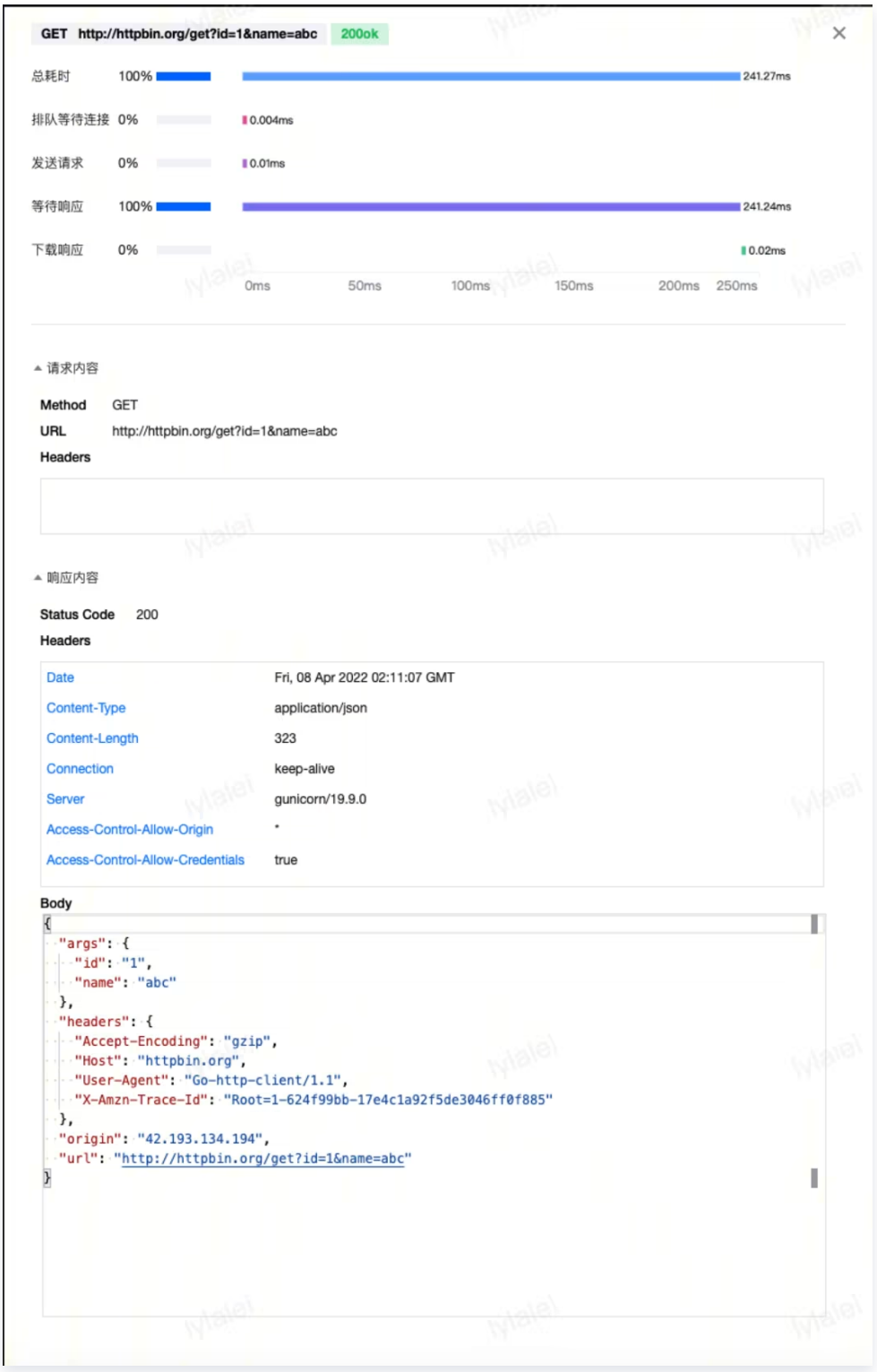
常见问题

PTS 调试失败/没有日志，如何去定位问题?

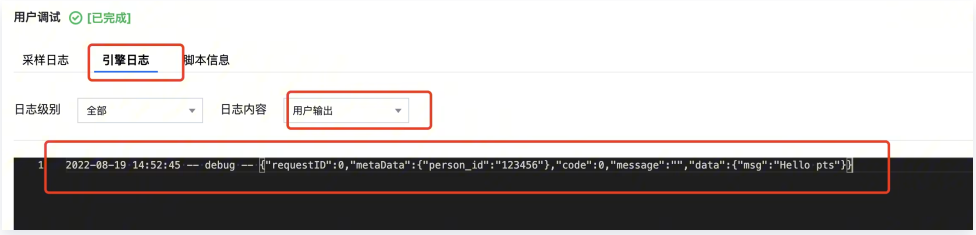
PTS 提供了全面的日志定位手段，分别为引擎日志/用户日志/请求日志，主要有三个阶段：

- **引擎日志**：JavaScript 脚本编写出现语法错误、空指针错误等问题，导致引擎解析脚本失败，可以按照日志提示的报错点进行修复。

- **请求日志**：已经产生网络 I/O 请求，对应请求包已经发到服务端，但服务端解析失败。可以根据请求采样中、或调试功能中的请求/响应包体的详细信息，定位哪个协议字段存在问题。



- **用户日志**：您在脚本中通过 console.log 打印相关变量，可以在用户日志中查看，调试相关变量的结构体/返回值定义。



测试报表会在云压测中保存多久?

测试报表包含指标数据及日志数据，默认保留45天，45天后将自动清理过期数据。在过期前，用户可下载测试报表，在本地进行保存。用户也可将测试报表设置为基线报表，基线报表将永久保存。

如何保护被压端服务，防止被压端服务异常影响业务可用性?

当被压端服务异常时，通过实时测试报表，您可以看到请求 RT 变高，甚至出现请求失败。
为了防止服务异常，您可以在测试场景编排中，设置被压服务 SLA（服务可用性指标），例如：限制响应 RT<100ms，请求失败率<0.1%。当压测指标触发被压服务 SLA 水位线时，可通过告警通知到您，也可根据设置自动停止压测任务。
另外为避免服务异常，也建议您：

- 设置合理的压力模型。
- 将起步压力设置较低，通过梯度模型或者手动逐步调高压力，观察服务整体可用性。

PTS 支持 JMeter 压测吗?

用户只需要在场景编排中导入 jmx 文件，即可以原生方式运行 JMeter 压测。PTS 支持以分布式方式运行 JMeter 引擎，提供便捷的横向扩容能力和实时测试报表。

HTTP 服务请求失败率高，返回大量的 `net/http: request canceled` 错误信息?

在压测报告可以看到详细的错误率，用户可以在采样日志看具体的耗时分布，如果是服务端返回超时，可自定义配置全局 option http timeout 参数，默认为 10s。

```
export const option = {
  http: {
    // 单位ms
    timeout: 10000,
  }
}
```

HTTP 请求出现 `x509: cannot validate certificate` 返回错误?

支持两种解决方案：

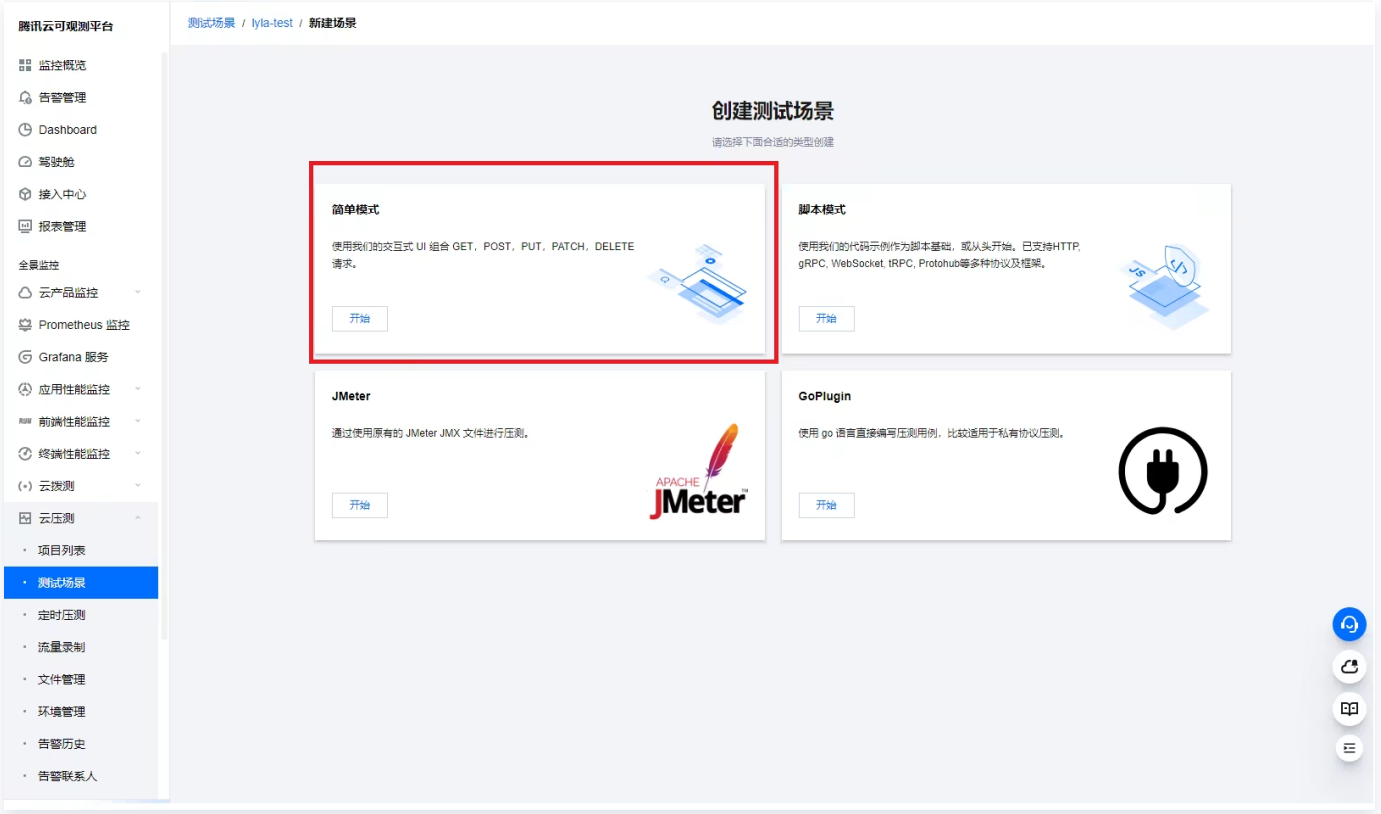
- 全局配置参数 insecureSkipVerify:true

```
export const option = {
  tlsConfig: {
    'localhost': {
      insecureSkipVerify: true
    }
  }
}
```

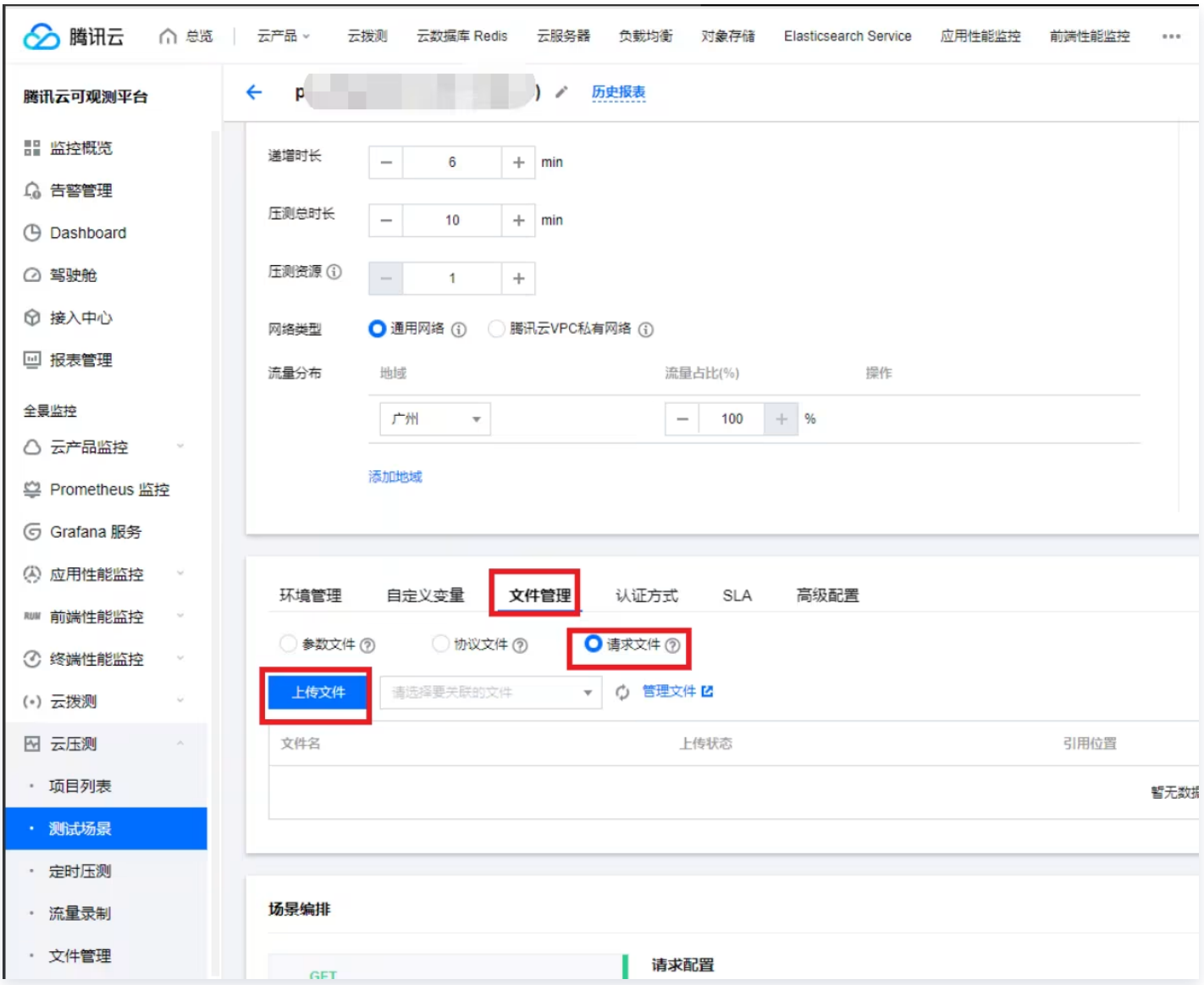
- 上传单独 TLS 证书

这种方法是将服务器返回的 TLS 证书上传至压测工具中，并在发送 HTTP 请求时指定证书路径，从而避免证书验证失败。这种方法相对更加安全，但需要手动上传证书文件。在使用腾讯云云压测时，可以按照以下步骤上传证书文件：

1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中单击**云压测 > 测试场景**。
3. 在项目列表中单击**新建场景 > 简单模式**。



4. 在项目列表页单击**文件管理 > 请求文件 > 上传文件**。



5. **上传 TLS 证书**，如下示例：

例如，在使用 JMeter 进行压测时，可以在 HTTP 请求中设置 sslManager 属性，指定证书路径：

比如上传了根域名证书：`ca.crt`, **客户端证书：**`client.crt`，**客户端**`key`: `client.key`

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

```
        serverName: "xxx.com"
      }
    }
}
```

> **注意：**
> 上传证书文件需要一定的时间进行审核和生效，因此在上传证书后可能需要等待一段时间才能使用。

PTS 支持哪些扩展方法，具体的参数定义去哪里查看?

- PTS 脚本示例，包括 HTTP、WebSocket 等常用协议。
- JavaScript API 列表概述。
- PTS 常用工具函数，包括随机数/base64 编解码/math 函数等。
- PTS 支持完整 ES6 语法，还支持 第三方包引用（如 crypto.js 这种 PTS 暂时没有集成的能力）。

PTS 如何从测试文件读取数据?

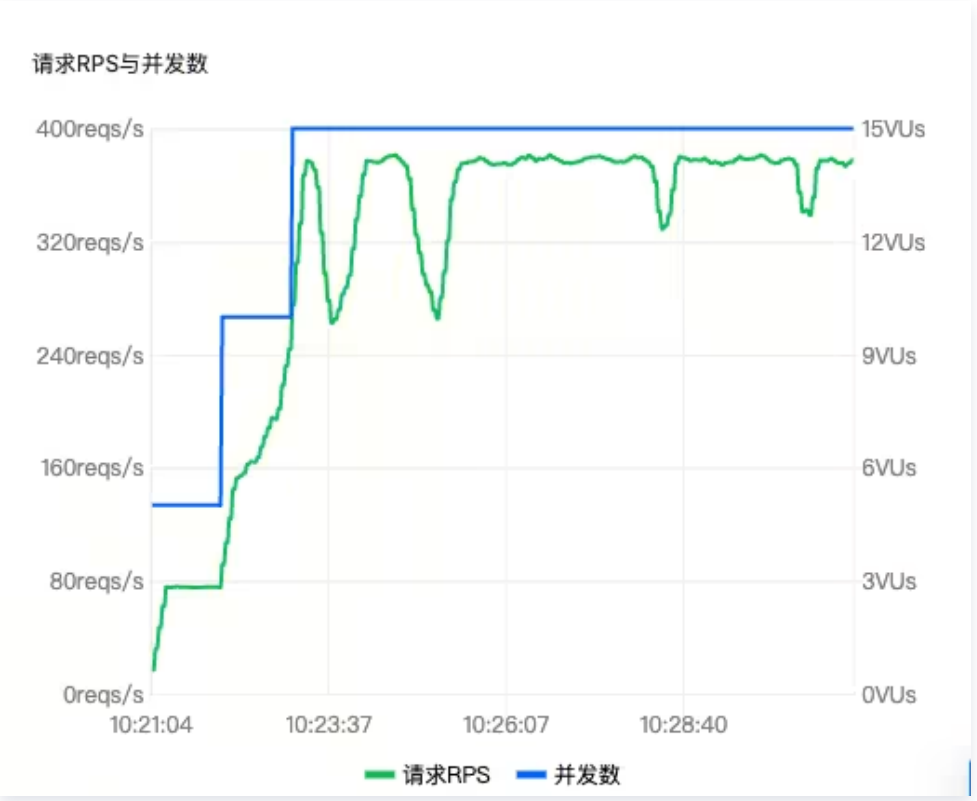
PTS 支持 dataset 读取测试数据，用户在压测场景完成文件上传，引擎会解析 csv 文件并按行轮询进行读取，具体的语法如下：

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

export default function () {
    const value = dataset.get("MyKey")
    //@ts-ignore    忽略校验
    const postResponse = http.post("http://httpbin.org/post", {data:
value});
    console.log(postResponse)
};
```

> **说明**
> 详细使用指引：使用参数文件。

PTS 报表显示 VU = 0，或者跟施压配置的值对应不上?

VU=0：PTS 报表显示请求的 VU（并发用户数）为瞬时指标，当处于任务结束时，其瞬时值有可能为 0。
跟施压配置的值对应不上：由于大部分用户设置的是梯度发压模型，VU 值会随时间梯度变化，其瞬时值应以图表显示的 VU 曲线变化为准。



导入的 csv 出现乱码，如何解决?

含中文的 csv 导入后乱码的问题：
因为 Windows 默认导出的 csv 使用的是 GBK 编码，并且旧版本的 Excel 2016 前会不保存 Bom (byte order mark)。
解决方法：将 csv 导出为 utf-8 格式：

- Windows 可以使用记事本打开 csv 文件后，另存为 utf-8 格式。
- Mac 上使用 `iconv -f GBK -t UTF-8 xxx.csv > utf-8.csv` 。

状态码 999 是什么错误，如何排查?

施压端没能从被压服务端得到有效的 HTTP 响应状态码，则会将状态码置为 999 Unknown。这些请求会被视为错误请求，计入压测报告的错误率。
错误原因可能是请求本身的协议/地址等有误，或者是网络原因、服务端的 DNS/防火墙/SSL 证书/超时断连等原因，导致服务不可达。
如需排查，可参考请求采样里的错误信息、施压机日志里的报错信息，还可使用调试模式调试请求。
常见原因如下：

- 施压端没能正常发出请求。
  - 请求采样里的报错信息：`Error net/http: request canceled while waiting for connection` 。可能的原因：
    - 施压端到被压端服务端口之间的网络不通。
    - 被压端服务的 DNS/防火墙/SSL 证书等配置错误。

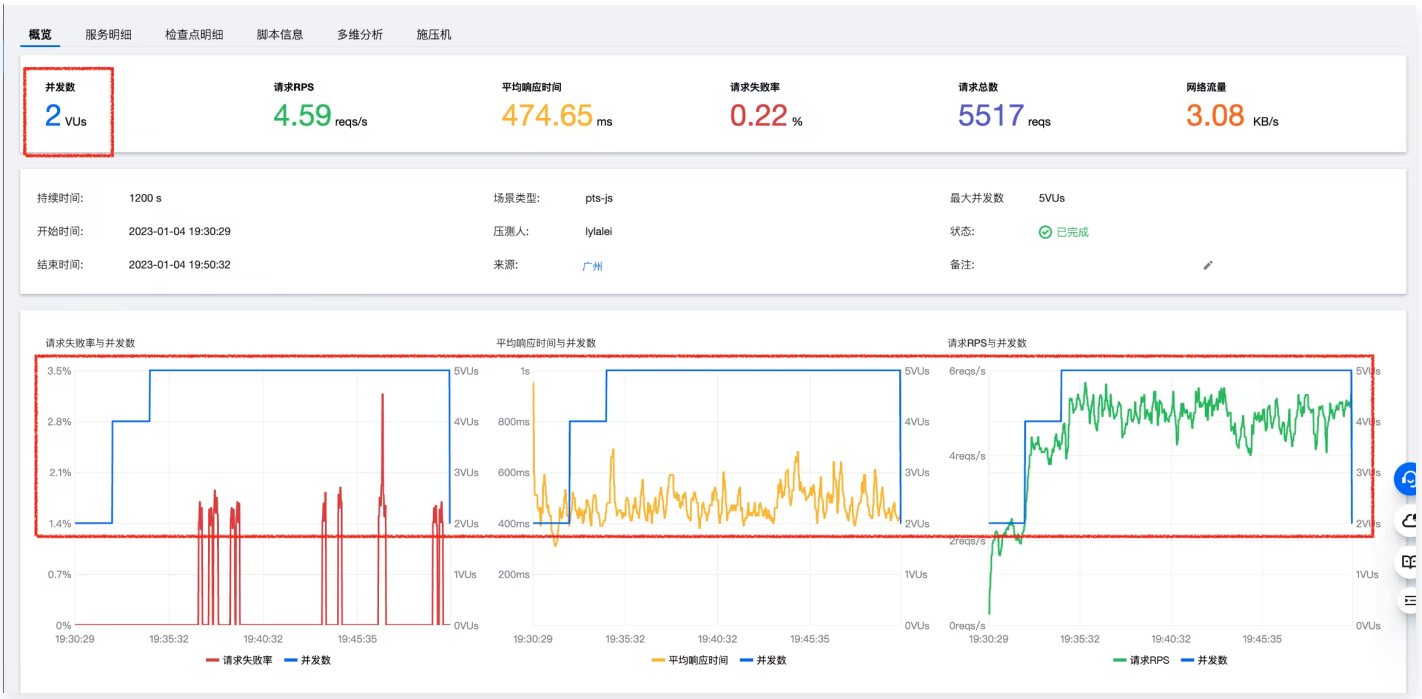
- 施压端已正常发出请求，但没能在超时时间内获得有效的响应状态码。
  - 例如，目前 HTTP 协议默认 10 秒钟超时，可观察压测报告里的响应时间是否已超过 10 秒、请求采样里的错误信息是否为 `Error net/http: request canceled` 。若确实是超时导致，可排查为何被压服务响应慢、优化其处理请求的能力。
  - 若需调大 HTTP 超时时间，可在脚本模式下配置，详见：配置选项 。
- 压测任务结束释放资源时，若有部分请求尚未完成，则会被施压端自动取消掉，此时请求采样里的报错信息为：`Error context deadline exceeded` 。

调试模式下，为什么我的请求只执行了一部分?

PTS 在调试模式下，压测引擎最多执行10秒，之后会自动退出。
如果您场景里编排的请求无法在10秒内全部完成，则会表现为只执行了部分请求。
建议直接以较小的 VU 数运行压测任务，来代替调试模式，避免其10秒无法执行全部请求的问题。

压测结束时，概览里的并发数（VUs）为什么突然下降了?

压测运行时，在报告页的概览栏里，并发数（VUs）的数值是实时值，与图表里代表并发数的蓝色梯度线在每个时刻的值是一致的。
压测结束时，PTS 会将资源回收，所以实时 VU 可能表现为瞬间下降，这是符合预期的正常行为。
您可参考图表里的蓝色线，观察并发数（VUs）随时间轴的变化，可以发现它是在将您配置的梯度发压如期完成后，在压测结束时刻才下降的。



采样日志的采样策略是什么样的，采样比例是多少?

PTS 使用首次采样与比例采样结合的方法来对用户请求进行采样。

- 首次采样策略

  我们将请求[service, method, status, result] 四个维度组合起来作为请求特征， 如果一组请求特征没有被记录过，那么这样的请求会被采样记录下来。
- 比例采样策略

  按千分之一的比例采样用户请求。首次采样策略命中的请求不计入该比例中。比例采样策略简化后：采样第1个请求，采样第1001个请求，以此类推。

以压测 http get 请求 `https://mockhttpbin.pts.svc.cluster.local/get` 请求为例:

- 第1个请求状态码返回200， 请求特征["https://mockhttpbin.pts.svc.cluster.local/get", "get", "200", "ok"]，这个特征首次出现，请求将被采样。
- 第2个请求返回status 200，比例采样策略命中该请求，该请求被记录下来。
- 第10个请求时候，出现了500错误，请求特征["https://mockhttpbin.pts.svc.cluster.local/get", "get", "500", "internal error"]，首次采样策略观测到这是首次出现的特征，该请求也会被采样。
- 第1002个请求返回 status 200，比例采样策略命中该请求，该请求也被记录下来。

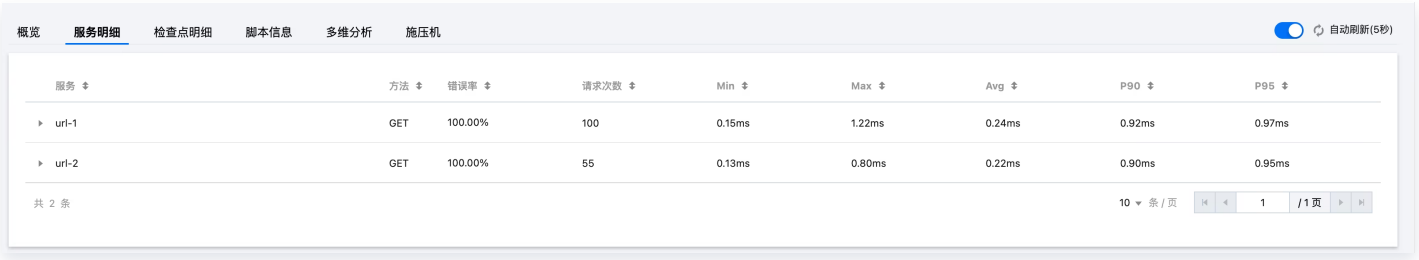
报告中服务明细内出现 tooManyService 是什么意思，该如何处理?

PTS 压测报告的服务明细中，默认将每个 URL 归类为一个“**服务 service**”，展示压测期间发送的所有请求的明细信息。如果 URL 数量较多，控制台仅展示 64 个不同的服务 service，更多的会以 **tooManyService** 标签汇总起来。

服务 service 数量过大，不对 URL 进行分类，对于报告的解读和分析有不利的影响；因此，把同一类的请求 service 汇总起来，对于统计分析具有较大的帮助。通过设置请求中 Request 的 service 字段，可以把同一类的请求分到同一个服务明细的展示行里面；如何配置，可以参考 Request 。

```
// Send a http get request
import http from 'pts/http';
import { check, sleep } from 'pts';


export default function () {
  const resp1 = http.get("http://mockhttpbin.pts.svc.cluster.local/get",
{
```

```
    service: "url-1",
  });
  const resp2 =
http.post("http://mockhttpbin.pts.svc.cluster.local/post", "", {
    service: "url-2",
  });
}
```



PTS 使用的是同步压测还是异步压测？为什么？

PTS 使用的是同步压测，即 PTS 的一个 VU（并发用户）会在发送请求后等待响应，直到接收到响应后才会发送下一个请求。

而异步压测是指一个 VU（并发用户）会在发送一个请求后，立即继续发送下一个请求，而不等待前一个请求的响应。

同步压测具有以下优点：

- 简单易懂；
- 更接近真实用户的行为（用户通常会等待请求的响应）；
- 易于调试（由于每个请求都是顺序执行的，调试和分析问题相对容易）。

异步压测缺点：

- 实现和调试相对复杂；
- 单个 VU（并发用户）可能占用过多资源；
- 无法准确模拟真实用户的行为（尤其是在用户需要等待响应的场景中）。

综上，PTS 平台选择同步压测的方式，以更好地模拟真实用户的行为。