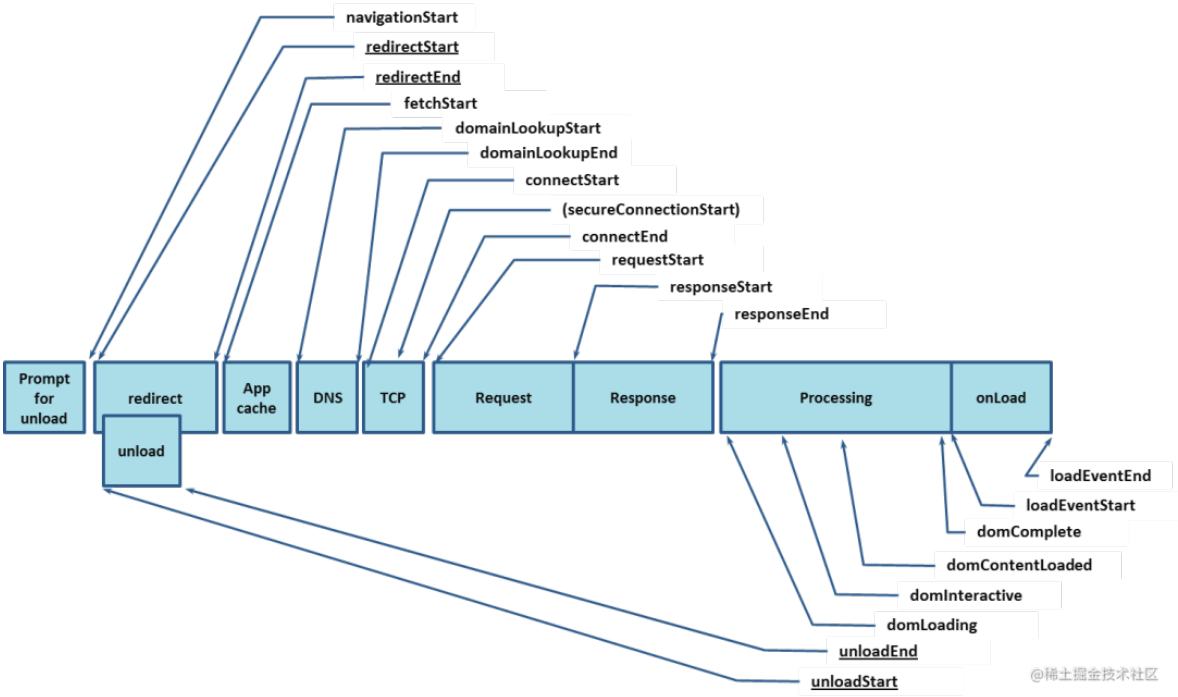
前端数据采集

前端性能指标采集

一、网络层面性能指标及采集

浏览器提供的能力: performance api
https://developer.mozilla.org/zh-CN/docs/Web/API/Navigation_timing_API



问题：浏览器请求一个url到底经历了什么?
描述了浏览器在请求一个资源的完整流程，从发起请求到加载完毕的各个阶段的性能耗时，**以加载速度来衡量性能**。

各字段含义

采集方法

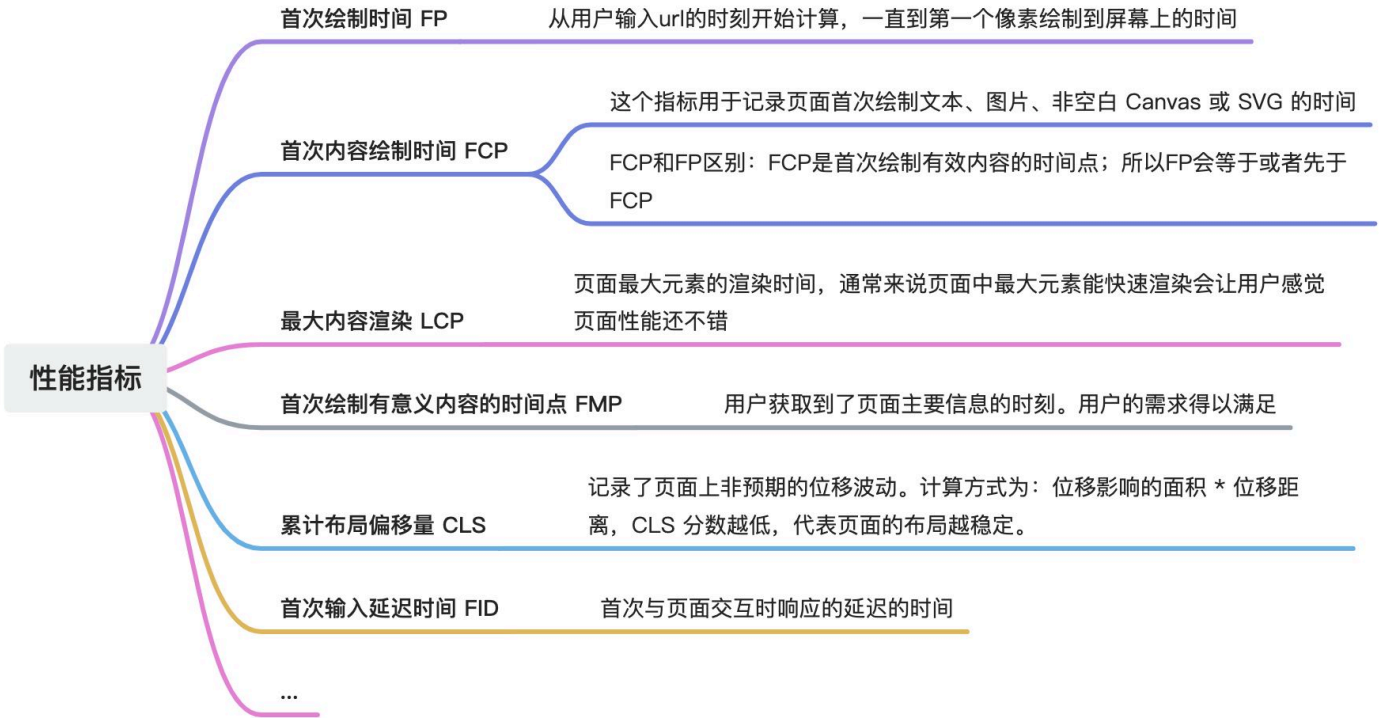
使用上面navigation timging 的相关属性，我们可以计算许多重要的指标

字段	描述	计算公式
redirect	重定向	redirectEnd – redirectStart
app cache	缓存	domLookupStart – fetchStart
DNS	DNS查询耗时	domainLookupEnd – domainLookupStart
TCP	TCP连接耗时	connectEnd – connectStart
request	请求耗时	connectEnd – secureConnectionStart
response	内容传输耗时	responseEnd – responseStart

DOMParse	DOM解析耗时	domInteractive – responseEnd
processing	文档解析过程	domComplete – domLoading
load	页面完全加载耗时	loadEventStart – navigationStart
DomReady	dom加载耗时	domContentLoaded – fetchStart
DOMParse	dom解析耗时	domInteractive – responseEnd
TTFB	首字节时间	reponseStart – navigationStart
resouseceLoad	剩余资源加载耗时	domComplete – domContentLoaded

二、以用户为中心的指标及采集

页面展示层面的指标以用户为中心的性能指标衡量页面显示有用内容的速度，用户是否可以交互，以及这些交互是否流畅



1. FP、FCP

```
const entryHandler = list => {
    for (const entry of list.getEntries()) {
        if (entry.name === 'first-contentful-paint') {
            observer.disconnect()
            // 计算首次内容绘制时间
            let FCP = entry.startTime
            console.log('fcp', FCP)
```

```
                }
            }
        }
        const observer = new PerformanceObserver(entryHandler)
        observer.observe({ type: 'paint', buffered: true })
```

2.FID

```
function getFID() {
    new PerformanceObserver((entryList, observer) => {
        let firstInput = entryList.getEntries()[0];
        if (firstInput) {
            const FID = firstInput.processingStart - firstInput.startTime;
            console.log('FID', FID);
        }
        observer.disconnect();
    }).observe({type: 'first-input', buffered: true});
}
```

3.LCP

```
const entryHandler = (list) => {
    if (observer) {
        observer.disconnect()
    }
    for (const entry of list.getEntries()) {
        // 最大内容绘制时间
       let LCP = entry.startTime
      console.log(LCP)
    }
}
const observer = new PerformanceObserver(entryHandler)
observer.observe({ type: 'largest-contentful-paint', buffered: true })
```

请求监控

接口监控的实现原理：针对浏览器内置的 XMLHttpRequest、fetch 对象，重写两种方法，进行代理，实现对请求的接口拦截，获取接口报错的情况并上报。

拦截XMLHttpRequest

代理的核心在于使用 apply 重新执行原有方法，并且在执行原有方法之前进行监听操作。

拦截open方法

- 获取请求方法
- 请求路径

拦截send方法

- 获取请求开始的时间和请求体

- 响应时间和响应体

```
function hookXhr() {
        let xhr = window.XMLHttpRequest
        // 拦截open,send方法
        let _open = xhr.prototype.open
        xhr.prototype.open = function (method, url) {
            this.logData = {
                method,
                url,
            }
            return _open.apply(this, arguments)
        }
        let _send = xhr.prototype.send
        xhr.prototype.send = function (body) {
            let start = Date.now()
            let fn = type => () => {
                console.log({
                    kind: 'stability',
                    type: 'xhr',
                    eventType: type,
                    duration: Date.now() - start,
                    status: this.status,
                    responseSize: this.responseText.length,
                    body,
                    ...this.logData,
                })
            }
            this.addEventListener('load', fn('load'), false)
            this.addEventListener('error', fn('error'), false)
            this.addEventListener('abort', fn('abort'), false)
            return _send.apply(this, arguments)
        }
}
```

拦截fetch

```
function fetchReplace() {
        const _fetch = window.fetch
        const newFetch = function (url, config) {
            let start = Date.now()
            const logData = {
                url,
                ...config,
            }
            const onResolve = res => {
                // 请求成功
                console.log({
                    kind: 'stability',
                    type: 'fetch',
                    eventType: 'load',
                    duration: Date.now() - start,
                    status: res.status,
                    ...logData,
                })
```

```
                return res
            }
            const onReject = err => {
                // 请求失败
                console.log({
                    kind: 'stability',
                    type: 'fetch',
                    eventType: 'error',
                    duration: Date.now() - start,
                    status: 0,
                    ...logData,
                })

                throw err
            }
            return _fetch.apply(this, arguments).then(onResolve, onReject)
        }
        window.fetch = newFetch
}
```

错误采集

错误信息是最基础也是最重要的数据，错误信息主要分为下面几类：

- JS 代码运行错误、语法错误等
- 异步错误等
- 静态资源加载错误

一、异常捕获方式

window.addEventListener 全局捕获

当静态资源加载失败时，会触发 error 事件，图片、script、css加载错误，都能被捕获

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
</head>
<script>
  window.addEventListener('error', (error) => {
    console.log('捕获到异常：', error);
  }, true)
</script>

<!-- 图片、script、css加载错误，都能被捕获 -->
<img src="https://test.cn/xxx.png">
<script src="https://test.cn/xxx.js"></script>
<link href="https://test.cn/xxx.css" rel="stylesheet" />
</html>
```

Promise错误

当 Promise 被 reject 且没有 reject 处理器的时候，无法被 window.onerror、try/catch、 error 事件捕获到，可通过 unhandledrejection 事件来处理

```
// unhandledrejection 可以捕获Promise中的错误
window.addEventListener(”unhandledrejection”, function(e) {
  console.log(”捕获到异常”, e);
  // preventDefault阻止传播，不会在控制台打印
  e.preventDefault();
});
```

二、注意点

跨域问题

- 原因

为了提升加载速度,大部分产品都有CDN部署,将资源部署到不同的域名上

浏览器出于安全性考虑，只允许同源的脚本捕获具体的错误，不同源脚本隐藏具体的错误信息，仅返回”Script error”

报错信息

- 解决方案

前端script加crossorigin，后端配置跨域头 Access-Control-Allow-Origin， 后可以捕获到完整的报错信息

```
<script src=”https://www.test.com/index.js” crossorigin></script>
```

堆栈解析sourmap

- 原因

监听捕获到错误的堆栈，文件名，行列号都是根据打包编辑混淆后的产物抛出的，无法快速定位到原始出现错误的具体代码。

- 解决方案

sourcemap 把混淆后的代码行列和原代码行列做了映射

混淆后的行列号 ===>源代码的行列号

混淆后的文件名 ===》源代码的文件名