# 在Garfish主应用实现的埋点 Copy

## 背景

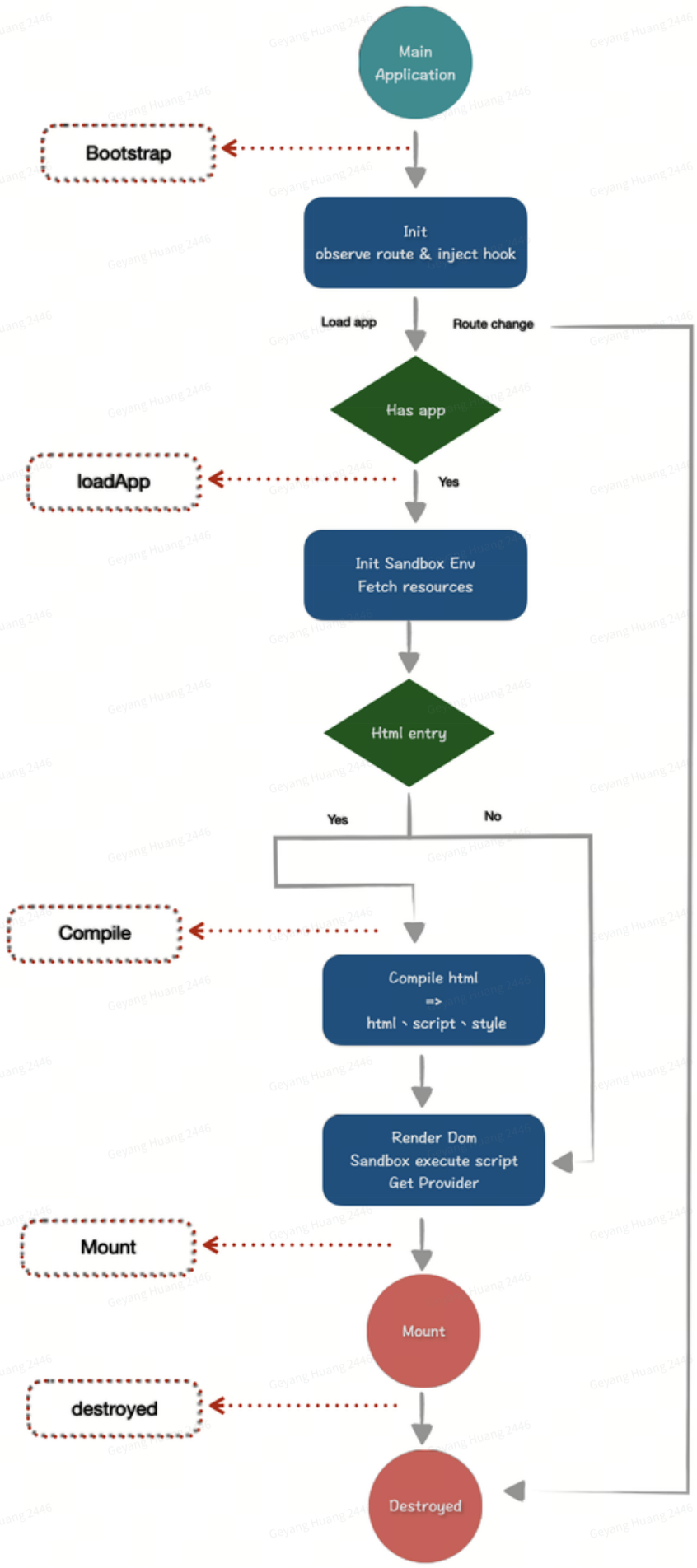
最近在做生产链路（PLM、版房、摄影、样品）的性能优化，并不是一开始就根据性能指标去做优化，而是先在整个链路的关键节点去做埋点监控，便于了解每个环节耗时情况。本次分享涉及内容：

- Garfish启动过程
- Garfish主应用实现的埋点链路
- 部分生命周期函数源码

> ❓ 为什么不直接运用Slardar的性能指标?
>
> 1、它是通用性指标，并不能完全符合需求；
>
> 2、指标相对零散，难以串联起来感知系统每个阶段的耗时；
>
> 3、指标准确性问题。

## Garfish启动过程

| 图例 | |
| --- | --- |

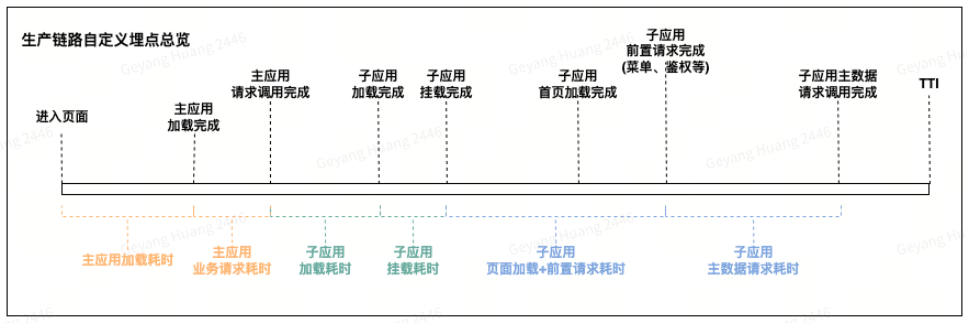
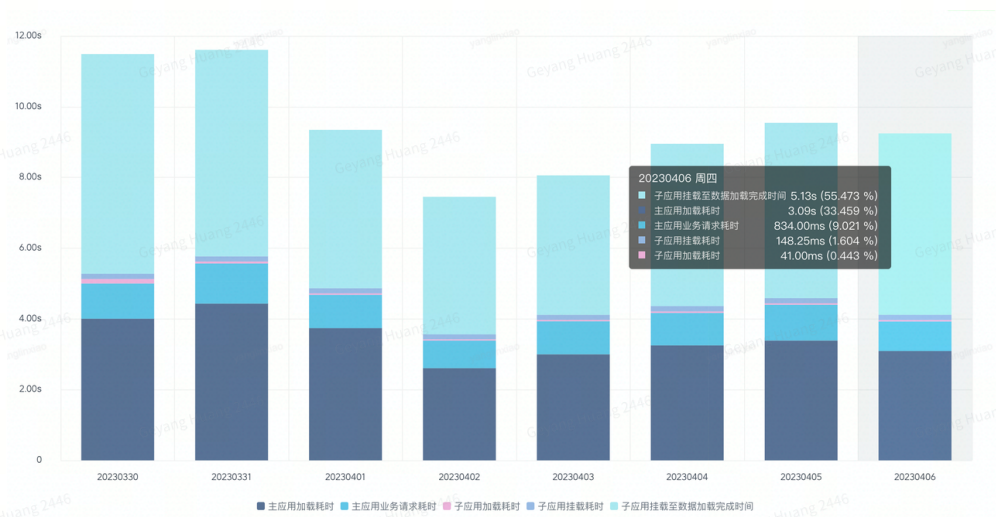



总结

- 渲染阶段
  - 主应用通过路由驱动或手动挂载的方式触发子应用渲染
  - 开始加载应用的资源内容，并初始化子应用的沙箱运行时环境
  - 判断入口类型
    - 若入口类型为 `HTML` 类型，将开始解析和拆解子应用资源
    - 若入口类型为 `JS`，创建子应用的挂载点 `DOM`
  - 将子应用存在”副作用“（对当前页面可能产生影响的内容）交由沙箱处理
  - 开始渲染子应用的 `DOM` 树
  - 触发子应用的渲染 `Hook`
- 销毁阶段

- 若路由变化离开子应用的激活范围或主动触发销毁函数，触发应用的销毁
- 清除应用在渲染时和运行时产生的副作用
- 移除子应用的 `DOM` 元素

# 埋点链路

| | |
|---|---|
| 链路图 |  |
| 效果图 |  |
| 埋点监控维度 | 主应用（加载、业务操作）和子应用（加载、挂载） |
| 上报位置 | 主应用 |
| 埋点数据验证手段 | • 和Slardar已有指标对比<br>• 业务代码中手动上报和Garfish透出钩子函数上报的对比<br>• 查看源码配合手动打点 |

## 主应用

### 加载

主应用在启动过程中Garfish对其初始化的耗时，比如主应用代码执行、自身资源拉取等。

### 上报

选取三个地方进行上报：

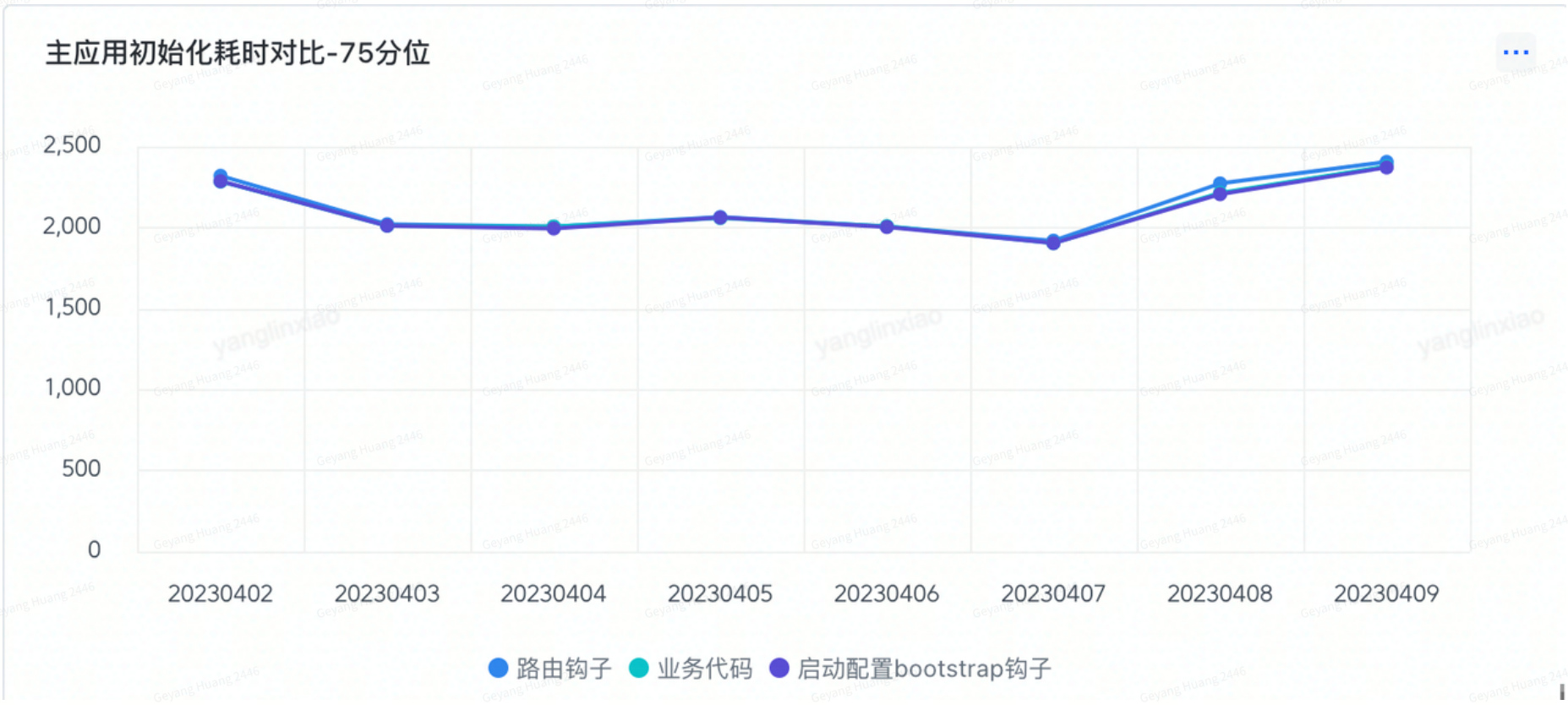
- Garfish路由钩子 `beforeEach` ：第一次匹配中子应用路由的时候上报，如何确定是第一次匹配？通过 `window.Garfish.cacheApps`
- Garfish启动配置中的 `bootstrap` 函数：默认在 `Garfish.run` 调用后触发，这个钩子函数并没有在官网文档透出，是通过源码发现的
- 在主应用 `useEffect` 的回调：手动在 `Garfish.run` 调用后触发

```
/**
 * 上报主应用初始化耗时
 * @param caller 调用方，在哪里进行上报的
```

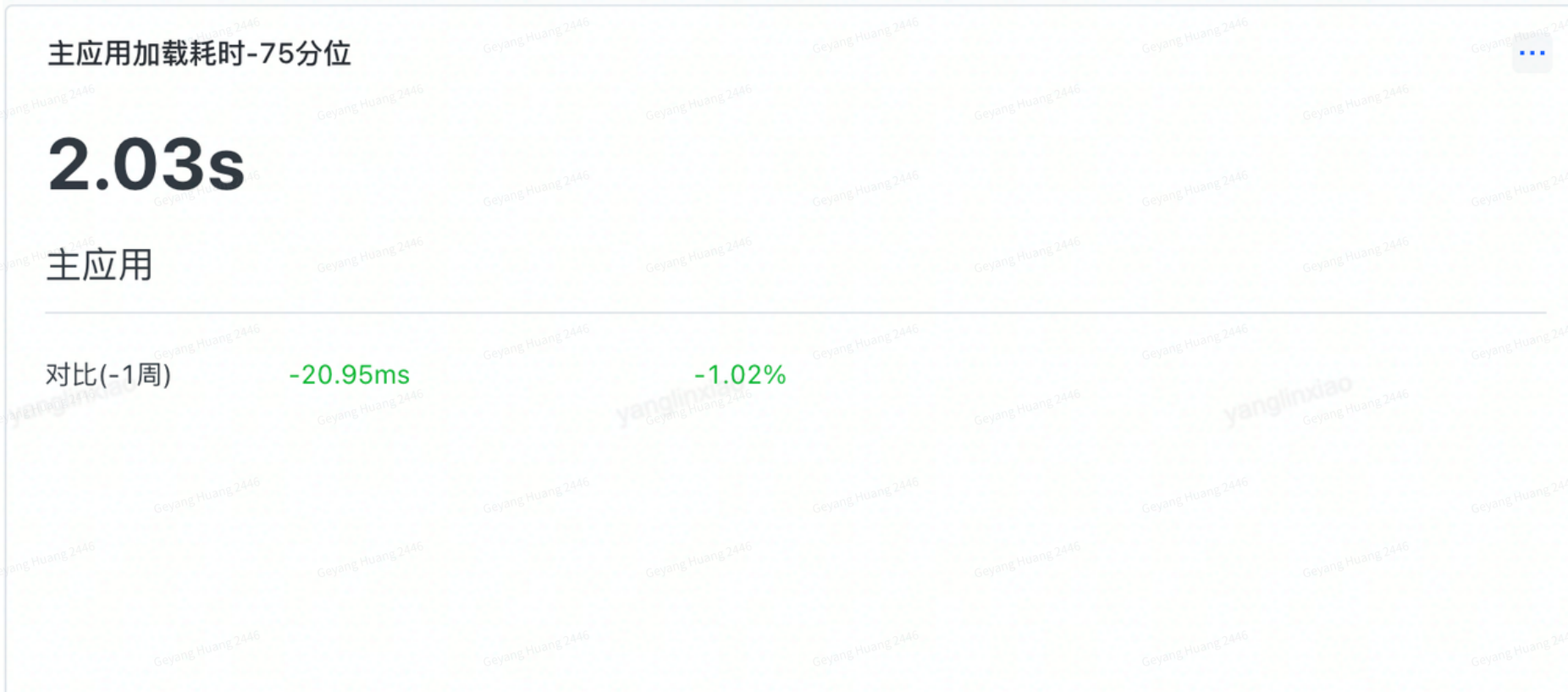
```
   * @param config 透传Slardar默认的事件配置
   */
export function reportMainAppInitUseTime(
  caller: MainAppInitCaller,
  config?: Omit<CustomEventData, 'name'>,
) {
  const duration = config?.metrics?.duration || performance.now();
  slardarInstance.sendEvent?.({
    name: SlardarCustomEvent.MainAppBeforeLoadSubAppUseTime,
    metrics: {
      [caller]: duration,
      ...config?.metrics,
    },
    ...config?.categories,
  });
}
```

### 数据验证

通过上面三个上报时机的交叉验证

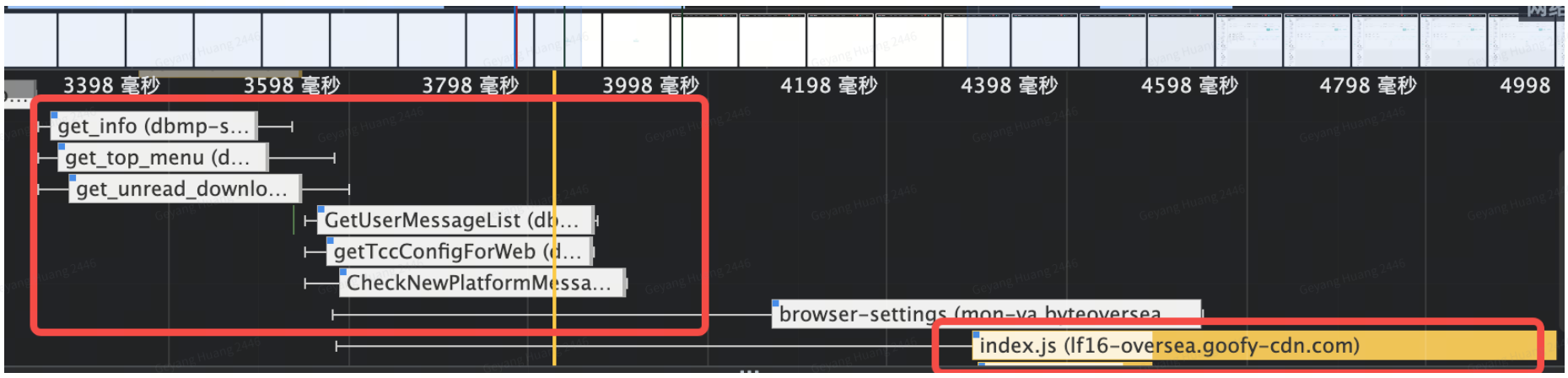



### 耗时



## 业务操作

主应用阻塞子应用渲染耗时，比如主应用的一些接口请求，子应用依赖接口的返回才开始渲染。




## 上报

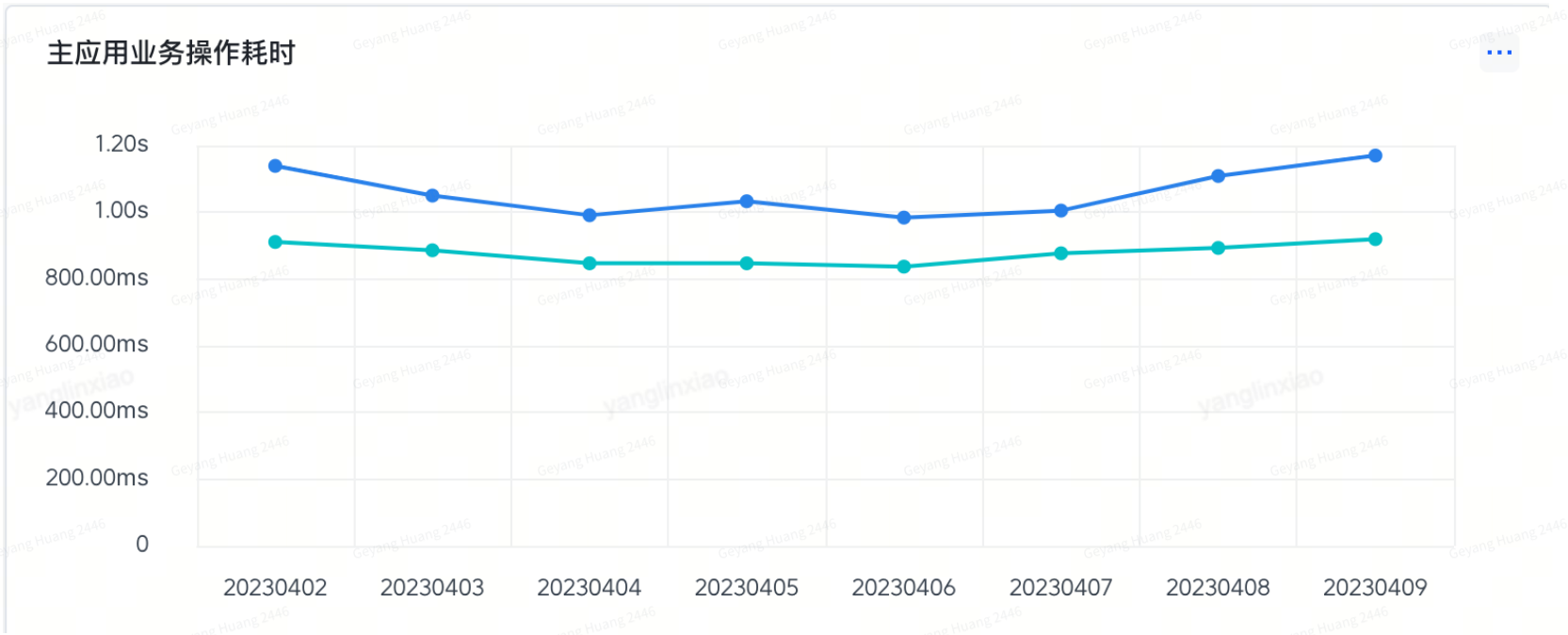
选取了两个地方进行上报：

- 在主应用获取用户信息接口后上报：只有获取用户信息后，子应用都才开始渲染，在这之前都处于阻塞状态
- Garfish启动配置中的 `afterMount` 函数：选取子应用挂载完毕的时间点，分别减去以下耗时：
  - 主应用加载时间点
  - 子应用加载耗时
  - 子应用挂载耗时

```
    afterMount(appInfo) {
      // 子应用信息
      const { mountStartTime, loadStartTime, isFirstLoad } = map[appInfo.name];
      // 加载、挂载总耗时
      const loadAndMountSubAppUseTime = Date.now() - loadStartTime;
      if (isFirstLoad) {
        // 首次加载，记录下主应用在业务操作的耗时
        const routeJump = Reflect.get(
          window.Garfish.getGlobalObject(),
          GarfishGlobalKey.RouteJump,
        );
        if (routeJump?.type === RouteJumpType.Init) {
          const mainAppOtherUseTime =
            Date.now() - routeJump.timestamp - loadAndMountSubAppUseTime;
          reportMainAppOtherUseTime(MainAppOtherCaller.AfterMount, {
            metrics: {
              duration: mainAppOtherUseTime,
            },
            categories,
          });
        }
      }
    },
```

## 数据验证

上面两个上报时机的交叉验证




耗时


## 子应用

### 加载

下载子应用入口文件资源的耗时。

### 上报

通过 `Garfish.run` 中 `beforeLoad` 和 `afterLoad` 中各打一个时间戳的点，相减后进行上报

```
afterLoad(appInfo) {
    const { loadStartTime, isFirstLoad } = map[appInfo.name];
    // 加载子应用耗时
    const loadSubAppUseTime = Date.now() - loadStartTime;
    // 上报加载子应用的耗时
    slardarInstance.sendEvent?.({
      name: SlardarCustomEvent.LoadSubAppUseTime,
      metrics: {
        // 每次都上传duration可以统一记录加载耗时，不区分首次和非首次
        duration: loadSubAppUseTime,
        // 区分首次和非首次上报
        [getDurationMetricsKey(isFirstLoad)]: loadSubAppUseTime,
      },
      categories: {
```

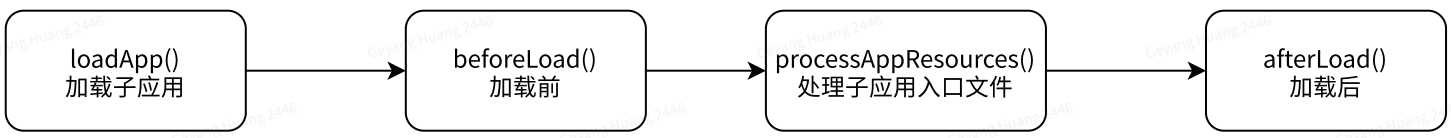
```
15          // 根据子应用分类上报
16          app: appInfo.name,
17        },
18      });
19    },
```

### 数据验证

采取两种验证方式，分别是查看源码和手动打点。

- 查看源码

调用流程




源码

`loadApp()`：执行 `beforeLoad` 和 `afterLoad` 钩子函数，处理子应用入口文件。

```
loadApp(appName, options) {
  assert4(appName, "Miss appName.");
  let appInfo = generateAppOptions(appName, this, options);
  const asyncLoadProcess = async () => {
    const stop = await this.hooks.lifecycle.beforeLoad.emit(appInfo);
    if (stop === false) {
      warn5(`Load ${appName} application is terminated by beforeLoad.`);
      return null;
    }
    appInfo = generateAppOptions(appName, this, options);
    assert4(appInfo.entry, `Can't load unexpected child app "${appName}", Please provide
    let appInstance = null;
    const cacheApp = this.cacheApps[appName];
    if (appInfo.cache && cacheApp) {
      appInstance = cacheApp;
    } else {
      try {
        const [manager, resources, isHtmlMode] = await processAppResources(this.loader,
        appInstance = new App(this, appInfo, manager, resources, isHtmlMode, appInfo.cust
        for (const key in this.plugins) {
          appInstance.hooks.usePlugin(this.plugins[key]);
        }
        if (appInfo.cache) {
          this.cacheApps[appName] = appInstance;
        }
      } catch (e) {
        (typeof process !== "undefined" && process.env && process.env.NODE_ENV ? process.
        this.hooks.lifecycle.errorLoadApp.emit(e, appInfo);
      }
    }
    await this.hooks.lifecycle.afterLoad.emit(appInfo, appInstance);
    return appInstance;
  };
  if (!this.loading[appName]) {
    this.loading[appName] = asyncLoadProcess().finally(() => {
      delete this.loading[appName];
    });
  }
  return this.loading[appName];
}
};
```



`processAppResources()`：通过 `fetch` 拉取子应用入口文件。

```
async function processAppResources(loader, appInfo) {
  let isHtmlMode = false, fakeEntryManager;
  const resources = { js: [], link: [], modules: [] };
  assert3(appInfo.entry, `[${appInfo.name}] Entry is not specified.`);
  const { resourceManager: entryManager } = await loader.load({
    scope: appInfo.name,
    url: transformUrl3(location.href, appInfo.entry)
  });
  if (entryManager instanceof TemplateManager2) {
    isHtmlMode = true;
    const [js, link, modules] = await fetchStaticResources(appInfo.name, loader, entryMana
    resources.js = js;
    resources.link = link;
    resources.modules = modules;
  } else if (entryManager instanceof JavaScriptManager) {
    isHtmlMode = false;
    const mockTemplateCode = `<script src="${entryManager.url}"><\/script>`;
    fakeEntryManager = new TemplateManager2(mockTemplateCode, entryManager.url);
    entryManager.setDep(fakeEntryManager.findAllJsNodes()[0]);
    resources.js = [entryManager];
  } else {
    error2(`Entrance wrong type of resource of "${appInfo.name}".`);
  }
  return [fakeEntryManager || entryManager, resources, isHtmlMode];
}
```

加载子应用入口文件

- 手动打点

比较平均的「埋点上报耗时」、「Slardar请求耗时」、「Chrome实际耗时」

| 系统 | 埋点上报耗时（ms） | Slardar请求耗时（ms） | Chorme实际耗时（ms） |
| --- | --- | --- | --- |
| PLM | 815.91 | 747.03 | 723.4（781/284/1490/729/333） |
| | 330.4 | 295.56 | 285.8（424/224/215/381/185） |
| 摄影 | 635.4 | 602.64 | 597.6（1510/744/227/236/271） |
| | 574.25 | 542.35 | 510（1400/249/283/228/393） |
| 样品 | 628.2 | 297.24 | 293.2（481/435/201/174/175） |
| | 1120 | 769.96 | 765.8(2040+303+705+491+290) |
| | 1290 | 1010 | 1288.8（1850/2640/557/572/825） |
| 版房 | 567.6 | 528.32 | 523.5（915/894/300/243/267） |
| | 859.2 | 808.04 | 788.2（1840/899/511/344/347） |

**耗时**

子应用加载耗时-75分位

| 分组指标 | 平均值 | 20230402 | 20230403 | 20230404 |
| --- | --- | --- | --- | --- |
| 子应用 sample | 1.28s | 1.57s | 93.00ms | 1.75s |
| 子应用 photograph | 418.78ms | 53.00ms | 96.50ms | 437.00ms |
| 子应用 plm-pattern-pc | 38.00ms | 34.00ms | 37.00ms | 41.00ms |
| 子应用 plm | 119.81ms | 33.00ms | 52.25ms | 57.00ms |

### 挂载

子应用入口文件执行耗时。

### 上报

上报方式和「加载耗时」差不多，通过 `Garfish.run` 中 `beforeMount` 和 `afterMount` 中各打一个时间戳的点，相减后进行上报。

```
afterMount(appInfo) {
    // 子应用信息
    const { mountStartTime, loadStartTime, isFirstLoad } = map[appInfo.name];
    // 挂载子应用耗时
    const mountSubAppUseTime = Date.now() - mountStartTime;
    const key = getDurationMetricsKey(isFirstLoad);
    const categories = {
      // 根据子应用分类上报
      app: appInfo.name,
    };
    // 上报挂载子应用的耗时
    slardarInstance.sendEvent?.({
      name: SlardarCustomEvent.MountSubAppUseTime,
      metrics: {
        // 每次都上传duration可以统一记录加载耗时，不区分首次和非首次
        duration: mountSubAppUseTime,
        // 区分首次和非首次上报
        [key]: mountSubAppUseTime,
      },
      categories,
    });
  },
```

### 数据验证

通过查看源码去验证。

调用流程


源码

`mount()`：子应用挂载过程。

```
async mount() {
    if (!this.canMount())
      return false;
    // 1、调用 app.options.beforeMount 钩子
    this.hooks.lifecycle.beforeMount.emit(this.appInfo, this, false);
    this.active = true;
    this.mounting = true;
    try {
      // 2、将 app set 到 window.Garfish.activeApps 中，Garfish.activeApps
      this.context.activeApps.push(this);
      // 3、创建 app 容器并添加到文档流上，编译子应用的代码
      // compileAndRenderContainer() -> renderTemplate() -> execScript() -> runCode()
      const { asyncScripts } = await this.compileAndRenderContainer();
      if (!this.stopMountAndClearEffect())
        return false;
      const provider = await this.getProvider(); // 4、获取子应用provider
      if (!this.stopMountAndClearEffect())
        return false;
      this.callRender(provider, true); // 5、调用子应用provider.render
      // 6、将 app.display 和 app.mounted 设置为 true
      this.display = true;
      this.mounted = true;
      this.hooks.lifecycle.afterMount.emit(this.appInfo, this, false); // 7、调用
app.options.afterMount 钩子
      await asyncScripts; // 异步加载的js文件
      if (!this.stopMountAndClearEffect())
        return false;
    } catch (e) {
      this.entryManager.DOMApis.removeElement(this.appContainer);
      this.hooks.lifecycle.errorMountApp.emit(e, this.appInfo);
      return false;
    } finally {
      this.mounting = false;
    }
    return true;
  }
```

`compileAndRenderContainer()`：挂载入口文件和执行异步加载的JS文件。

```
async compileAndRenderContainer() {
  await this.renderTemplate();
  return {
    asyncScripts: new Promise((resolve) => {
      setTimeout(() => {
        if (this.stopMountAndClearEffect()) {
          for (const jsManager of this.resources.js) {
            if (jsManager.async) {
              try {
                this.execScript(jsManager.scriptCode, {}, jsManager.url || this.appInfo.er
                  async: false,
                  noEntry: true
                });
              } catch (e) {
                this.hooks.lifecycle.errorMountApp.emit(e, this.appInfo);
              }
            }
          }
        }
        resolve();
      });
    })
  };
}
```

`renderTemplate()`：创建子应用容器和根据不同标签创建对应的DOM节点挂载。

```
689   }
690   async renderTemplate() {
691     const { appInfo, entryManager, resources } = this;
692     const { url: baseUrl, DOMApis } = entryManager;
693     const { htmlNode, appContainer } = createAppContainer(appInfo);
694     this.htmlNode = htmlNode;
695     this.appContainer = appContainer;
696     await this.addContainer();
697     const customRenderer = {
698       meta: () => null,
699       img: (node) => {
700         baseUrl && entryManager.toResolveUrl(node, "src", baseUrl);
701         return DOMApis.createElement(node);
702       },
703       video: (node) => {
704         baseUrl && entryManager.toResolveUrl(node, "src", baseUrl);
705         return DOMApis.createElement(node);
706       },
707       audio: (node) => {
708         baseUrl && entryManager.toResolveUrl(node, "src", baseUrl);
709         return DOMApis.createElement(node);
710       },
711       body: (node) => {
712         if (!this.strictIsolation) {
713           node = entryManager.cloneNode(node);
714           node.tagName = "div";
715           node.attributes.push({
716             key: __MockBody__,
717             value: null
718           });
719         }
720         return DOMApis.createElement(node);
721       },
722       head: (node) => {
723         if (!this.strictIsolation) {
724           node = entryManager.cloneNode(node);
725           node.tagName = "div";
726           node.attributes.push({
727             key: __MockHead__,
728             value: null
729           });
730         }
731         return DOMApis.createElement(node);
732       },
733       script: (node) => {
734         const mimeType = entryManager.findAttributeValue(node, "type");
735         const isModule = mimeType === "module";
736         if (mimeType) {
737           if (!isModule && !isJsType({ type: mimeType })) {
738             return DOMApis.createElement(node);
739           }
740         }
741         const jsManager = resources.js.find((manager) => {
742           return !manager.async ? manager.isSameOrigin(node) : false;
743         });
744         if (jsManager) {
745           const { url, scriptCode } = jsManager;
746           const mockOriginScript = document.createElement("script");
747           node.attributes.forEach((attribute) => {
748             if (attribute.key) {
749               mockOriginScript.setAttribute(attribute.key, attribute.value || "");
750             }
751           });
752           this.execScript(scriptCode, {}, url || this.appInfo.entry, {
753             isModule,
754             async: false,
755             isInline: jsManager.isInlineScript(),
756             noEntry: toBoolean(entryManager.findAttributeValue(node, "no-entry")),
757             originScript: mockOriginScript
758           });
```

创建子应用容器

执行子应用入口文件代码

`execScript()` 和 `runCode()`：执行JS代码。

```
466   }
467   execScript(code, env, url, options) {
468     env = __spreadValues(__spreadValues({}, this.getExecScriptEnv(options == null ? void 0 : options.no
469     this.scriptCount++;
470     const args = [this.appInfo, code, env, url, options];
471     this.hooks.lifecycle.beforeEval.emit(...args);
472     try {
473       this.runCode(code, env, url, options);
474     } catch (err) {
475       this.hooks.lifecycle.errorExecCode.emit(err, ...args);
476       throw err;
477     }
478     this.hooks.lifecycle.afterEval.emit(...args);
479   }
480   runCode(code, env, url, options) {
481     if (options && options.isModule) {
482       this.esmQueue.add(async (next) => {
483         await this.esModuleLoader.load(code, env, url, options);
484         next();
485       });
486     } else {
487       const revertCurrentScript = setDocCurrentScript(this.global.document, code, true, url, options ==
488       code += url ? `
489 //# sourceURL=${url}
490 ` : "";
491       if (!hasOwn(env, "window")) {
492         env = __spreadProps(__spreadValues({}, env), {
493           window: this.global
494         });
495       }
496       evalWithEnv(`;${code}`, env, this.global);
497       setTimeout(revertCurrentScript, 0);
498     }
499   }
```

耗时

子应用挂载耗时-75分位

| 分组指标 | 平均值 | 20230402 | 20230403 | 20230404 |
| --- | --- | --- | --- | --- |
| 子应用 plm | 157.81ms | 168.50ms | 151.00ms | 145.00ms |
| 子应用 sample | 157.19ms | 148.00ms | 156.25ms | 167.50ms |
| 子应用 plm-pattern-pc | 144.25ms | 139.00ms | 144.00ms | 151.00ms |
| 子应用 photograph | 107.88ms | 97.00ms | 101.00ms | 105.00ms |

1

📌 在挂载源码中，发现通过 `Garfish.run` 中的 `beforeEval` 和 `afterEval` 可以计算子应用的每个资源文件的「执行耗时」，它们是嵌套挂载过程中的，作用类似于 `performance.getEntries()` 。

```
execScript(code, env, url, options) {
    env = __spreadValues(__spreadValues({}, this.getExecScriptEnv(options == null ? void 0 :
options.noEntry)), env || {});
    this.scriptCount++;
    const args = [this.appInfo, code, env, url, options];
    this.hooks.lifecycle.beforeEval.emit(...args);
    try {
      this.runCode(code, env, url, options);
    } catch (err) {
      this.hooks.lifecycle.errorExecCode.emit(err, ...args);
      throw err;
    }
```

```
12      this.hooks.lifecycle.afterEval.emit(...args);
13    }
```

## 插件封装

Garfish 框架引入了插件化机制，目的是为了让开发者能够通过编写插件的方式扩展更多功能，或为自身业务定制个性化功能。目前在主应用的埋点已封装成插件 `@s_op/main-app-track-plugin`，待稳定后其它主应用即可引入使用：

```
1  import { MainAppTrackPlugin } from '@s_op/main-app-track-plugin';
2  Garfish.usePlugin(MainAppTrackPlugin({ slardarInstance }))
```