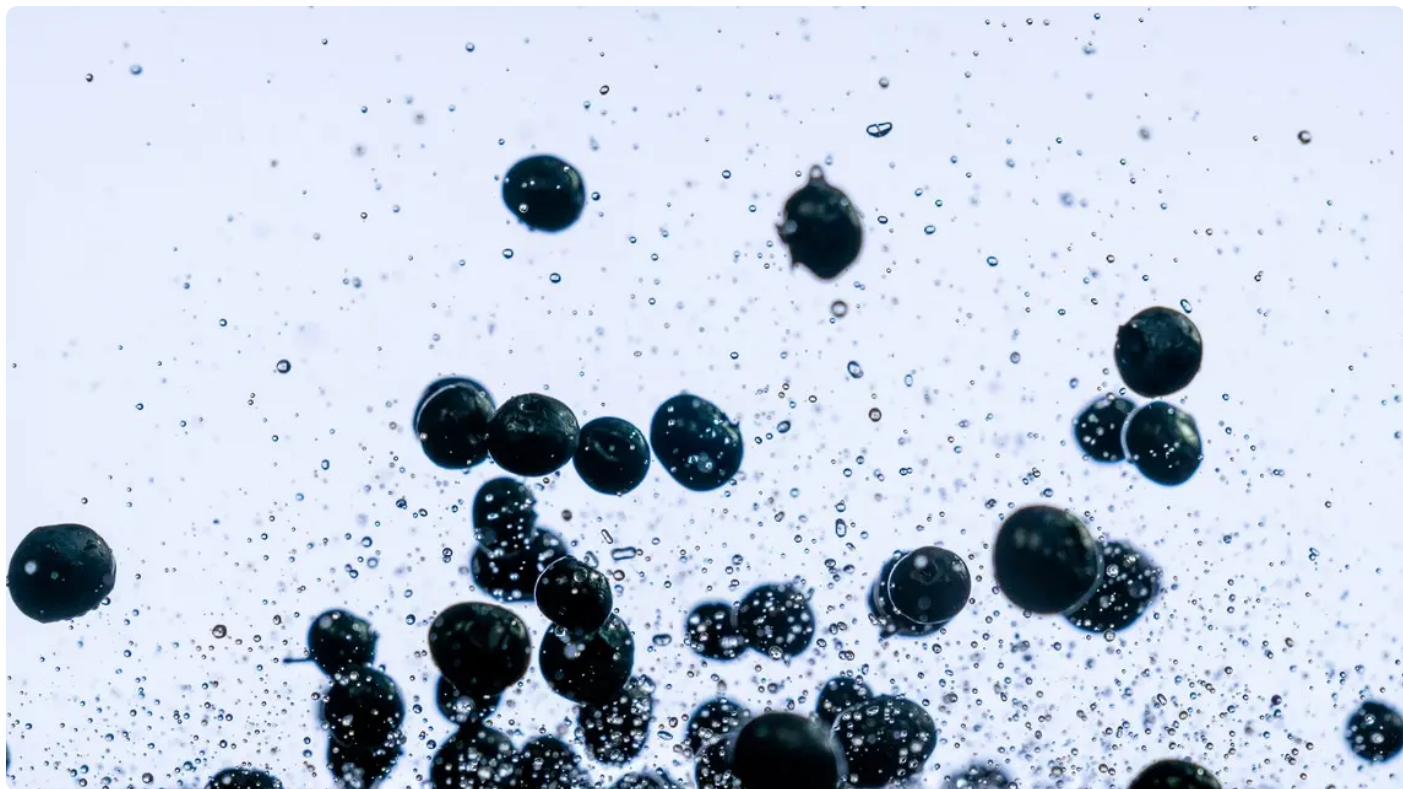


## 25 | 性能优化：如何设计一个合适的性能优化方案？

2022-05-25 蒋宏伟

《React Native 新架构实战课》

课程介绍 >



讲述：蒋宏伟

时长 17:38 大小 16.15M



你好，我是蒋宏伟。

性能问题是我们程序员绕不过的问题。有些人一遇到性能问题，首先想到的就是去网上寻找解决方案。看到别人做了拆包提升了性能，就认为拆包是解决性能问题的银弹；看到别人做了预加载提升了性能，就认为预加载能解决自己 App 的性能问题。

但其实，**比技术方案更重要的是技术思路。**

如果你没有性能优化的思路，不能结合自己业务的实际情况分析，而是随便在网上找个方案，直接生搬硬套放在自己的 App 上，很容易吃力不讨好。

网上性能优化方案五花八门，各大厂之间也并不完全相同，就我所知，除了我们上一讲介绍的拆包方案之外，其他性能优化方案还有内置、Push、预加载、并行加载、热点资源定期拉取、按需加载等，这些方案都可以减少网络请求的耗时。

再比如，环境提前初始化、Bundle 预执行、Hermes 字节码、inlineRequire 等方案，也可以减少执行耗时。

难道这么多性能优化方案，我们都要上吗？应该根据什么标准来确定呢？先做哪个性能优化方案，后做哪个性能优化方案？你把性能优化上线后，又怎么和老板汇报自己的成绩呢？

## 以终为始的设计思路

我们以终为始的思路，先看下应该怎么和老板汇报自己的成绩，然后再根据汇报目的决定性能优化的方案。

在一个公司中，性能优化是一个非常好出成绩的点，因为性能优化能够产出可量化的指标。

在晋级的时候，如果你能拿出“将 A 业务的首屏耗时由 2s 优化到 1s，性能提升了 100%”这样的量化指标，比“花费 3 个月，完成 A 业务”这样不好量化的指标更容易出成绩。

但很多技术同学汇报的时候，就只是汇报到优化了多少秒，这是不够的。你的评委很难将多少秒的优化和对业务产生的影响联系起来，如果不能性能优化和转化率、销售额挂钩，是不好确定成绩的。

那我们应该怎么汇报呢？你可以这样汇报，比如：

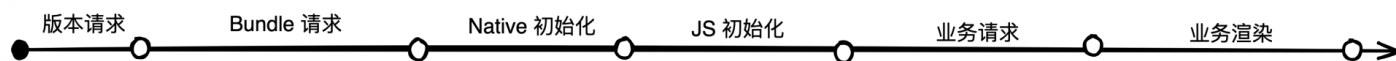
- 将首屏时间降低了 1s，访问转化率提高了 6.9%（数据来源，[🔗 58 同城 4 个 RN 业务的统计](#)）；
- 将主页性能提高了 60%，用户注册转换率提高了 40%（数据来源，[🔗 Pinterest](#)）；
- 将页面延迟降低了 100ms，销售额增加了 1%（数据来源，[🔗 亚马逊](#)）。

你看，把性能和业务指标挂钩后，这样汇报起来是不是更有说服力呢？

那么，既然在性能优化项目的最后，我们要汇报性能指标和业务指标。那么性能优化项目开始之前，我们最好能提前把性能埋点和业务关键指标埋点给埋上。这样一方面，在性能优化项目结束后，我们能够确定项目收益；另一方面，提前埋点也能帮助我们找到现有的性能瓶颈。

## 立项之前的指标统计

接下来，我以性能对页面到达率的影响为例，介绍一下我的统计方案：



在开始性能优化项目立项之前，我首先会统计当前业务的性能瓶颈和到达率。在本地确定各个关键时间点后，我会在这些关键点上埋上统计埋点。

以 **React Native** 混合应用为例，一个未优化的页面的主要加载流程包括：版本请求、**Bundle** 请求、**React Native** 框架 **Native** 部分的初始化、**React Native** 框架 **JavaScript** 公共包的初始化、业务的数据请求，以及最终的业务渲染。

当然，其中零散的耗时，由于耗时非常短，这里就进行了简化处理。比如用户点击业务入口，获取跳转协议开始路由的执行耗时就非常短，通常只有几毫秒，这些不关键耗时我这里就简化了。

那么，我们具体怎么统计耗时和到达率呢？我以版本请求阶段的耗时统计和到达率为例，和你介绍一下：

 复制代码

```
1 // 版本请求开始
2 { timestamp: 1652976000000, point: "version_request_start", uuid: "123e4567-e89
3 // 版本请求结束
4 { timestamp: 1652976000200, point: "version_request_end", uuid: "123e4567-e89b-
```

你就可以在发版本请求之前，也就是 **version\_request\_start** 时间点，还有版本请求回来之后，也就是 **version\_request\_end** 时间点，进行埋点。

版本请求存在两种情况，一种请求是用户请求成功，另一种情况是用户请求失败。请求失败的原因可能有很多，有些情况可能是，网络请求速度慢，用户不想等了，中途退出了；有些情况是，用户误点了，直接中途点击 **back** 按钮取消跳转了，销毁了载体页，不再发送统计埋点了；还有可能是超时、报错等失败情况。

成功的情况下，版本请求开始埋点和版本请求结束埋点都发送成功，此时用户为“达到用户”；而在失败的情况下，版本请求开始埋点会发送成功，版本请求结束埋点会发送失败，此时用户为“未达到用户”。

从统计学的意义上讲，达到用户的占比，就是到达率，计算公式如下：

 复制代码

```
1 到达率 = count(version_request_end)/count(version_request_start) * 100%
```

另一方面，性能指标，也就是版本请求平均耗时的计算公式如下：

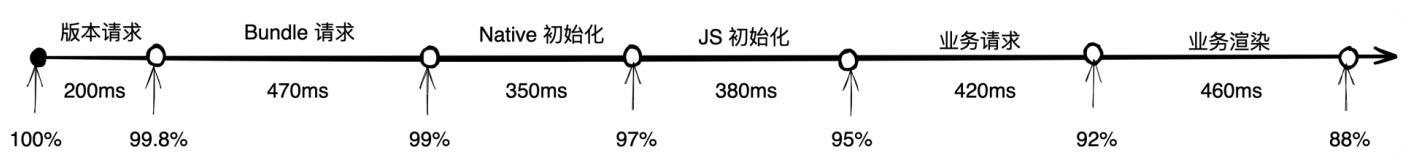
 复制代码

```
1 平均版本请求耗时 = SUM(MAX((version_request_end - version_request_start), 0)) / c
```

写 SQL 来统计版本请求平均耗时，还是稍微麻烦的，这里我就简化了一下。平均版本请求耗时就是到达用户的总耗时除以到达用户总数。

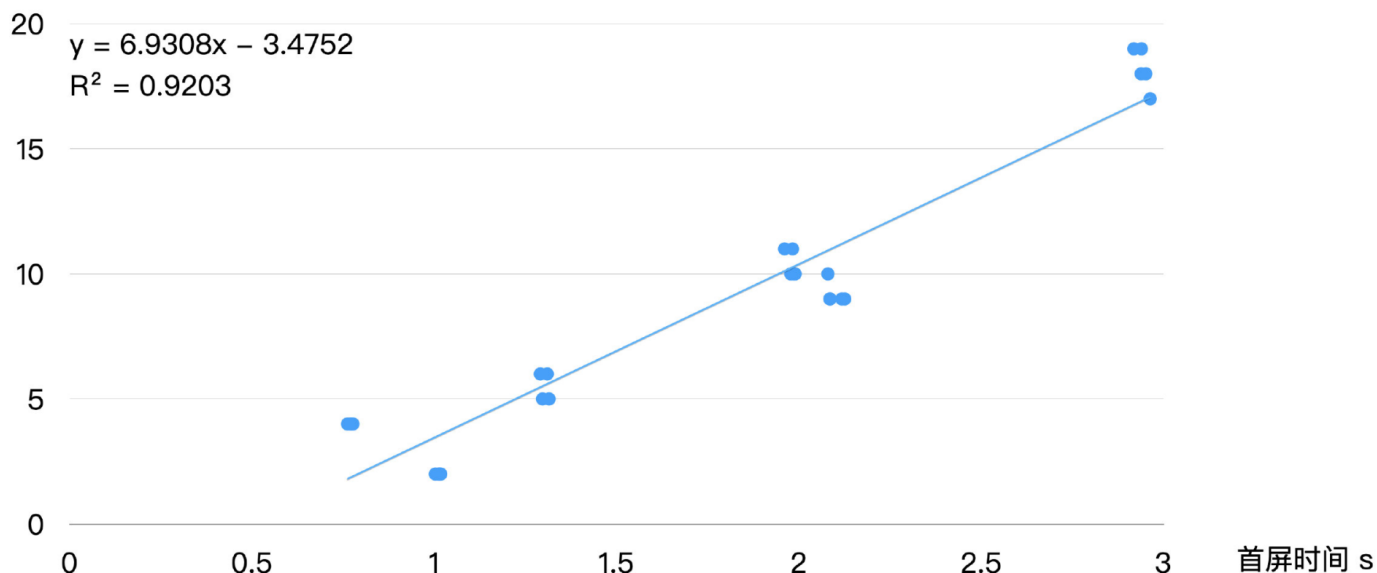
当你理解了版本请求到达率、耗时计算的统计原理后，Bundle 请求、React Native 框架 Native 部分的初始化、React Native 框架 JavaScript 公共包的初始化、业务的数据请求、业务渲染，这些的耗时、到达率统计方案也是类似，我就不啰嗦了。

计算完成后，你会得到一个平均耗时与流失率的关系图，如下：



以上是一个业务的平均耗时与流失率的关系数据。当你多统计几个业务的数据之后，并将这些业务数据使用 Excel/Number 的趋势线进行拟合后，你会得到一个趋势线，示例如下：

流失率 %



有了趋势线，就有了预期收益指标。这张图的大致意思是，🔗首屏耗时超过 1s 后，每增加 1s，用户到达率会降低 6.9%。

当然，每个业务都有自己的特点，性能与到达率的关系不一定就是 6.9% 这个值，但是这个统计思路你是可以借鉴的。

在性能优化项目立项之前，完成统计耗时与到达率的指标统计的目的，不仅是找到项目的性能瓶颈，更重要的是，它能帮你确定项目的预期成本和收益，比如采用 X 手段，将性能耗时降低 Y，用户到达率能够提高 Z。

在立项之前，完成预期成本和收益估算的作用是，它能帮你在项目前期和老板争取资源，或者项目后期让你的汇报、晋升更有说服力。



## 优化方案的确定

不过，当项目准备立项之前，你还有一项最重要的事情要做，就是选择优化方案。

你可以结合业务实际情况、业务的性能瓶颈数据，然后参考业内方案进行选择。能够参考的方案，除了各大厂 React Native 本身的优化方案外，你还可以参考业内的 Web、小程序、Native 这些领域中的成熟优化方案。

业务层的优化方案，我们就不介绍了。我们主要看看搞基建要做的底层优化方案，**第一类是网络请求类优化，第二类是执行耗时优化。**

网络请求类优化包括版本请求、**Bundle** 请求、业务请求的优化，主流的方向如下：

-  **HTTP**：编码效率优化、信道利用率优化、传输路径优化、信息安全优化；
-  **加载策略**：**NetWork Only**、**Cache First**、**Network First**、**Stale While Revalidate**（当次更新，下次生效）；
- 拆包策略：**patch** 拆包 / 模块拆包、内置包、增量更新；
- 预请求策略：首页预请求、上个页面预请求、热点资源定时更新、并行版本请求和业务请求的策略；
- 延后请求策略：**Dynamic Import**；
- 视觉策略：跳转动画、骨架屏。

第二类常用的执行耗时优化，包括 **React Native** 的 **Native** 代码初始化、**JavaScript** 代码初始化、业务渲染的优化方案如下：

- 引擎优化：**JSC** 引擎、**Hermes** 引擎、**V8** 引擎；
- 代码预执行：**Native** 环境预创建、**JavaScript** 公共代码预执行、**React Native SSR**；
- 代码懒执行：新架构、**inlineRequire**、**Dynamic Import**。

那么，以上这么多优化方案该怎么选呢？

简单来说，就是什么方案性价比最高，就优先选择什么方案。一般来说，能够直接复用业内开源工具的方案，实现成本更低，比如 **Hermes** 引擎、模块拆包、**Dynamic Import**、**inlineRequire**。这些方案基本上拿来直接适配一下就能用，你选一个预期成本低、收益高的方案做就行。

而比如自研加载策略、预请求策略、代码预执行、**HTTP** 优化、**SSR** 这些方面，业内只提供了思路，没有开源方案，开发成本很高，你可以后面再做。

## 整体加载策略

接下来，我以最关键版本请求、**Bundle** 请求的方案设计为例，帮你打开方案设计的思路。

我在设计 **58RN** 整体加载策略上，影响我最深的是  **Service Worker** 的加载策略。



Service Worker 和 React Native 混合应用很相似。Service Worker 的本质是充当 Web 应用程序与服务端之间的代理服务，Service Worker 代理服务会拦截 Web 应用的请求，并决定请求返回值是来自本地，还是来自服务端。

而 React Native 混合应用也是类似的。Native 充当了 React Native 应用与服务端之间的代理服务，Native 代理服务帮助 React Native 应用进行版本请求、资源请求，并决定版本请求、资源请求的返回值是来自本地，还是来自服务端。

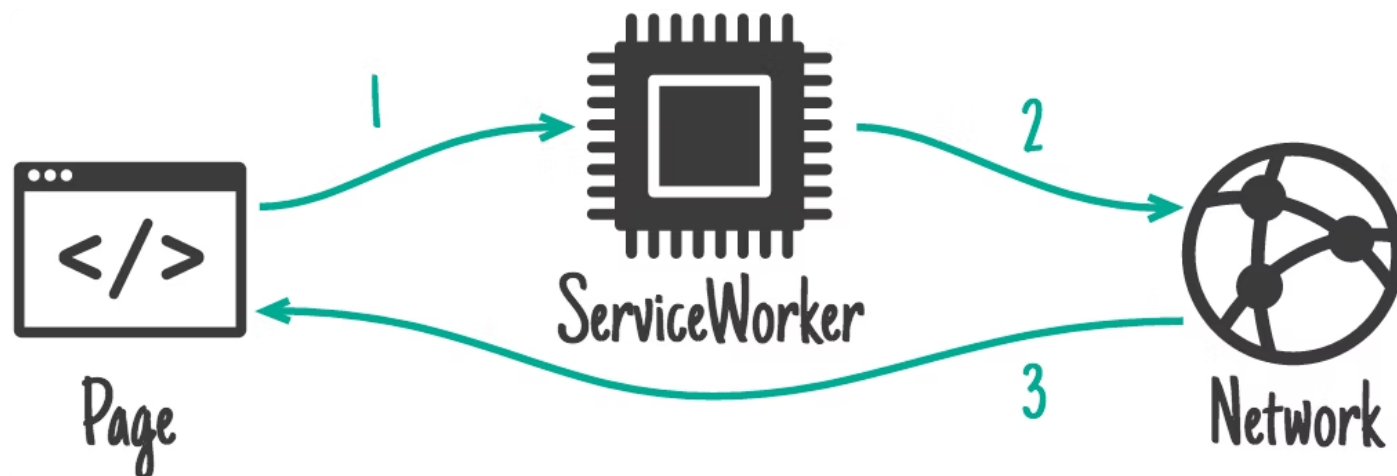
你看，Native 在版本请求、资源请求上的作用，是不是和 Service Worker 非常类似？不同的是，Native 的版本请求、资源请求是定制的，而 Service Worker 的请求是可自定义的。

因此，当初我的思路是定制几个常用的请求方式，然后业务可以根据具体情况进行配置。

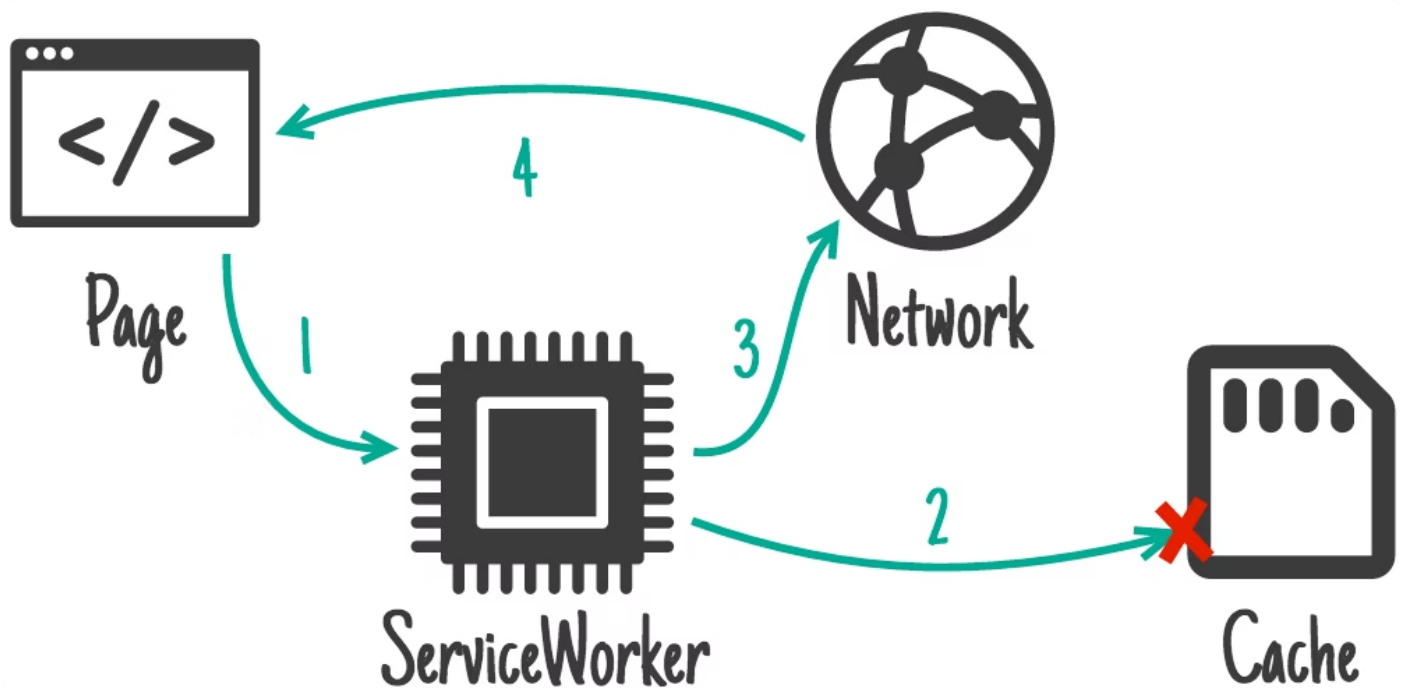
那 Service Worker 有哪些常用的策略可以参考呢？主要包括 4 种：

- NetWork Only;
- Cache First;
- Network First;
- Stale While Revalidate。

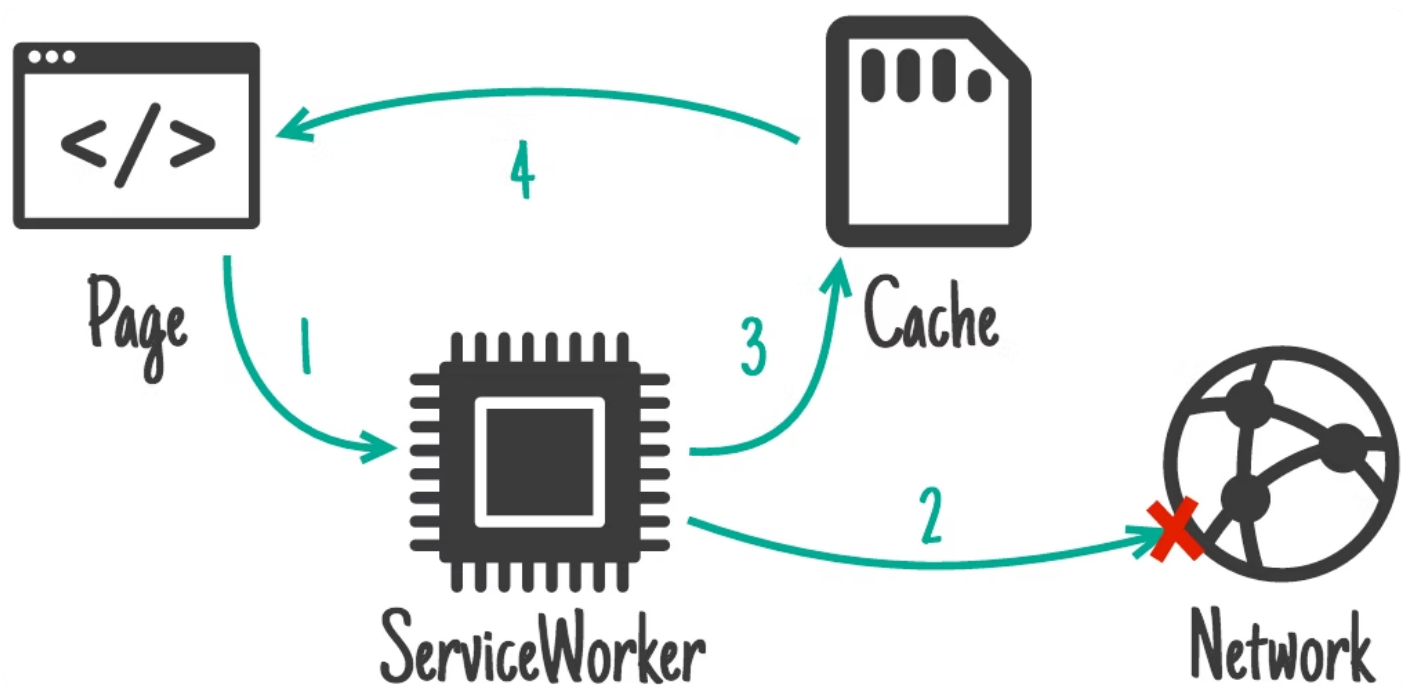
我们先来介绍下 NetWork Only 加载策略。这个方案中，页面请求只是过了一下 Service Worker 代理，然后直接请求服务端：



然后再来看 Cache First 策略。页面请求到 Service Worker 代理，然后优先读取 Cache 值返回。如果 Cache 不存在，或则 Cache 读取失败，则接着请求网络，返回相应值：



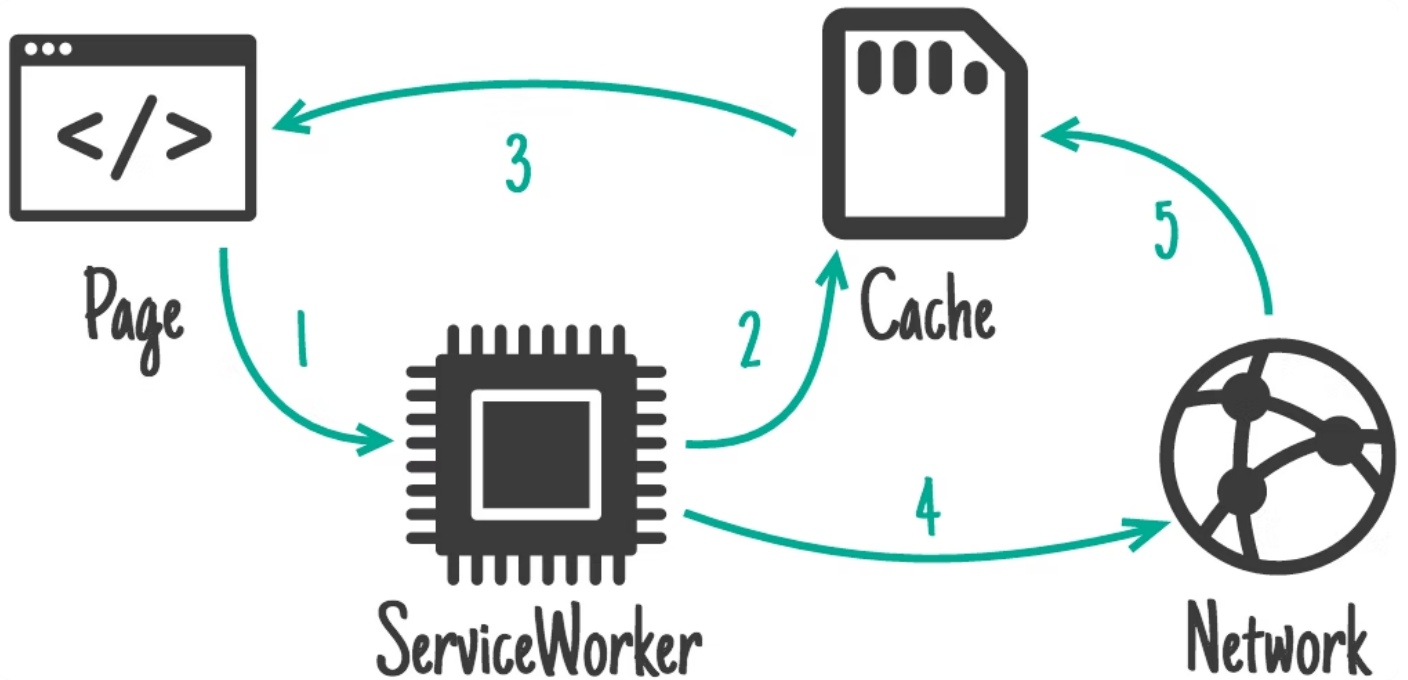
第三种是 Network First 加载策略。不同于 Cache First 和 Network First 加载策略，这个策略会优先从服务端请求返回值，只有服务端请求失败时才会从 Cache 中读取返回值：



第四个策略是 Stale While Revalidate 加载策略。这个策略会稍微复杂一点，我会解释得详细一些。在第一次请求时，没有 Cache，会先从服务端请求返回值，并将资源存放在 Cache 中。在第二次有 Cache 时，会先读取 Cache 并进行返回，在读取 Cache 的同时，会发起一个异步请求去更新 Cache。

Stale While Revalidate 不是很好翻译，我们内部常用的翻译有静默更新、异步更新，它的返回值是当次更新，下次使用的。示意图如下：





好了，Service Worker 的介绍就到这里。接下来我们的关键是怎么把 Service Worker 的思路用到 React Native 的版本请求、Bundle 请求中来。

我们第一版的设计思路是，版本请求用 **Stale While Revalidate** 策略，Bundle 请求用 **Cache First** 策略。该方案的优势是，只要用户访问过一次，那他下次访问时版本请求和 Bundle 请求都走的是 Cache。

但这样的话，由于用户每次都使用的是上次缓存的资源，而不是线上最新的资源，就会存在一种风险，如果一个带严重 Bug 的版本被用户缓存下来了，那这个 Bug 是不能即使得到更新的，因为任何访问过的用户永远使用的是 Cache 资源。

因此，我们接着又设计了第二版方案。第二版本方案，版本请求用 **Network Only** 策略，Bundle 请求依旧用的是 **Cache First** 策略。这样，在用户每次进入 React Native 页面时，都会发起一个版本请求，因此用户永远获取到的是最新版，彻底避免的缓存严重 Bug 风险。

第二版方案也不是没有缺陷，第二版方案的缺陷就是性能较差，因为每次都会有会网络请求。

为了解决性能较差的问题，我们又设计了第三版方案，也就是 **Bundle 预加载** 方案。由于首页是 **Native** 页面，因此可以在首页加载完成的空闲时间，把 Tab 页面的 Bundle 资源给提前下载并缓存了。然后再把 Tab 页面的 **Stale While Revalidate** 策略开启，这样用户在进入 Tab 页面的时候就能够直接使用 Cache 的 Bundle 进行加载了。

当然，并不是所有的 **Bundle** 资源都适合预加载，你需要在页面性能提升和用户流量浪费之间做取舍，每个页面请求都不一样，没办法统一处理，只能 **Case By Case** 地、一个个地配置预加载功能。

## 总结

今天，我们介绍了如何设计一个合适的性能优化方案的基本思路。

性能优化项目这类技术类项目有着明确的、可量化的技术指标和业务指标，因此在立项之初，我们就应该以用户为出发点，评估项目的成本和收益。有了明确的预期数据后，无论是在立项还是总结汇报时，都会有很大的优势。

接着，在项目设计环节，不要立刻着手开始开发，应该结合业务实际的性能瓶颈，调研业内的成熟方案，然后产出适合自己的技术方案进行落地。

性能优化要做的事情很多，对于热更新 **React Native** 应用而言，最关键的优化环节是版本请求和 **Bundle** 请求。你可以参考四种常用 **Service Worker** 策略，包括 **NetWork Only**、**Cache First**、**Network First**、**Stale While Revalidate**，进行设计，产出适合你热更新策略。


## 作业

请你思考一下，针对业务请求的性能瓶颈，应该怎么设计性能优化方案？

欢迎在评论区留言分享，咱们下一节课见。

分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 20 元

 生成海报并分享

 赞 2  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

## 精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。