蚂蚁金服如何把前端性能监控做到极致?

原创 杨森 前端之巅 2019-02-23

作者 | 杨森
编辑 | 覃云

本文来自蚂蚁金服前端技术专家杨森在 ArchSummit 北京 2018 的分享，他将分享如何通过 Performance 相关的 API 准确的采集用户性能数据，并如何通过大数据计算加工最终产出用户性能分析产品，以及如何通过性能数据纵向衡量产品性能、发现性能瓶颈。（**文章经速记整理而来，有适当删减，文末有完整演讲视频**）

本文的主要从以下四个方面阐述：

- 前端性能监控的两种主要技术方案；
- 怎么样去做一个真实用户性能数据的采集；
- 怎么去分析这些数据；
- 这些性能数据在蚂蚁金服怎么应用。

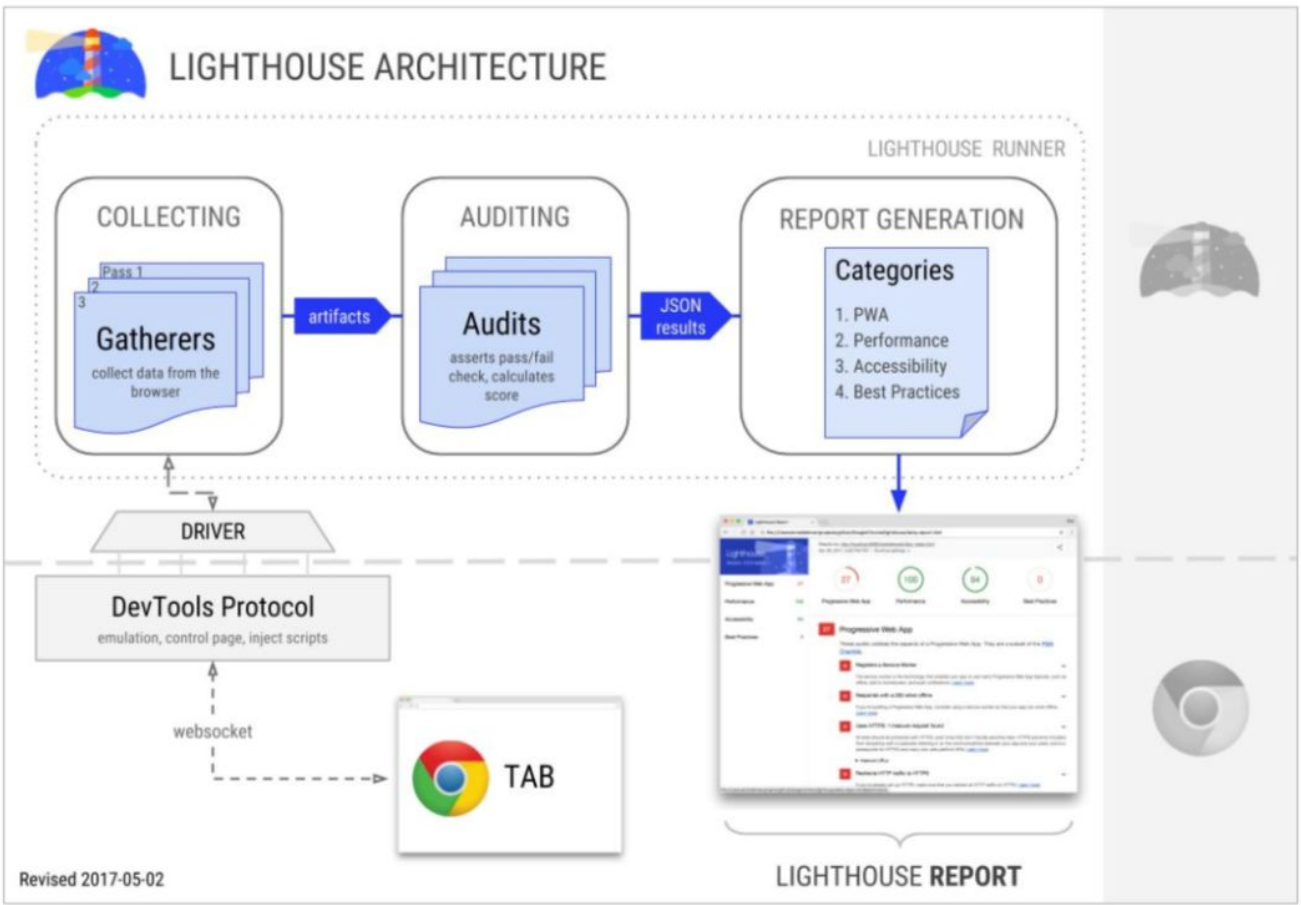
前端性能监控的两架马车

从技术方面来讲，前端性能监控主要分为两种方式，一种叫做合成监控（Synthetic Monitoring，SYN），另一种是真实用户监控（Real User Monitoring，RUM）。

合成监控

什么叫合成监控？就是在一个模拟场景里，去提交一个需要做性能审计的页面，通过一系列的工具、规则去运行你的页面，提取一些性能指标，得出一个审计报告。

合成监控中最近比较流行的是 Google 的 Lighthouse，下面我们就以 Lighthouse 为例。



Lighthouse 提供了很多种方案，但是我们这里演示的是一种以命令行工具的形式去对一个淘宝首页做性能审计。我们可以输出目标页面，它就会在我们这个模拟环境里面打开淘宝，然后去做一些性能数据的提取。

我们会看到一个生成的性能报告，从这个性能报告里我们可以看到，Lighthouse 生成的不仅仅是一些性能相关的数据，甚至包括 PWA，然后一些 SEO，甚至一些前端工程化的最佳实践等等。

当然其实业界对于 Lighthouse 也是评价有褒有贬，因为 Google 借助这个看似中立的性能评审工具也是在推行它的一些技术的方案。

合成监控的优缺点

现在，我们通过下表，来看看合成监控的优缺点：

👍 优点	👎 缺点
实现简单，解决方案成熟	合成条件配置复杂，无法全部还原真实场景
能采集到更丰富的数据，如硬件指标或瀑布图	登录等场景需要额外解决
不影响真实用户的访问性能	单次运行数据不够稳定
可以提供页面加载幻灯片等可视化分析途径	数据量较小，无法发挥更大价值

真实用用户监控（RUM）

所谓真实用户监控，就是用户在我们的页面上访问，访问之后就会产生各种各样的性能指标，我们在用户访问结束的时候，把这些性能指标上传到我们的日志服务器上，进行数据的提取清洗加工，最后在我们的监控平台上进行展示的一个过程。

真实用户监控 (RUM) 的优缺点

👍 优点	👎 缺点
无需配置模拟条件，完全还原真实场景	一定程度影响真实用户的访问性能及流量消耗
不存在登录等需要额外解决的场景	无法采集硬件相关指标
数据样本足够庞大，可以减少统计误差	受传输限制无法采集完整的资源加载瀑布图
性能数据可与其他数据关联产生更大价值	无法可视化展示页面加载过程

因为真实用户监控也是在运行时执行，所以这种真实用户监控比较难采集到一些硬件相关的指标，包括也很难去采集这个页面执行的幻灯片（即逐帧截图）。当然技术上，你可以用 JS 把当前页面保存成一个 Canvas，做一些逐帧对比，甚至把这些数据回传回去。但是在实践过程中，我们肯定不会这样做，因为这对用户的流量是极大的浪费。介绍完这两种监控方案我们来看一下他们两种方案的对比。

对比项	合成监控 SYN	真实用户监控 RUM
实现难度及成本	较低	较高
采集数据丰富度	丰富	基础
数据样本量	较小	大（视业务体量）
适合场景	支持团队自有业务，对性能做定性分析，或配合 CI 做小数据量的监控分析	作为中台产品支持前台业务，对性能做定量分析，结合业务数据进行深度挖掘

下文中，我们会着重介绍真实用户性能数据采集的方案。

真实用户性能数据采集方案

我认为，在真实用户性能数据采集时，要关注四个方面的东西：

- 使用标准的 API；
- 定义合适的指标；
- 采集正确的数据；
- 上报关联的维度。

下面我们来逐个解析一下。

使用标准的 API

如果大家有研究过前端性能监控，可能会知道浏览器会提供这么一个 API 叫 `performance.timing`，它会提供一个页面，从开始加载一直到加载完毕，中间各个阶段的一个模型，但这个 API 已经“废弃”了。为什么会被废弃？因为 W3C 给我们提供了更全面、更强大的一个性能分析矩阵，比单一的 `performance.timing` 更加强大，能帮助我们从各个方面分析前端页面性能。

Web Performance 相关 specs 一览

还有 High Resolution Time 这个基础的 API，可以为我们提供更精准的 timestamp。

- 使用标准的API —— High Resolution Time, HRTime

```
> Date.now()
<· 1543782619270
> performance.now()
<· 11222.600000008242
> performance.timeOrigin + performance.now()
<· 1543782634196.7979
```

1秒 = 1,000毫秒 = 1,000,000微秒 = 1,000,000,000纳秒

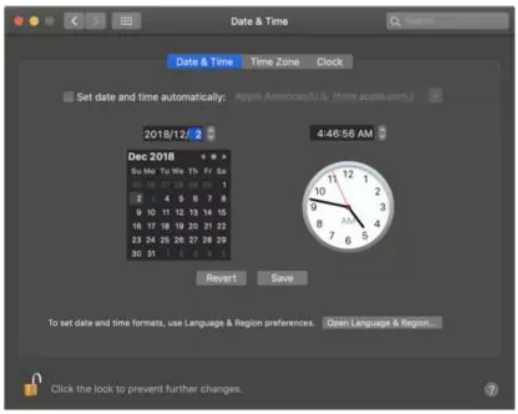
为什么所有的这些规范都是基于这个高精度时间的规范呢。大家写过 JS 都知道，可以用 Data.API 去提取当前的时间轴，单位是毫秒，这是我们新的高精度时间的定义，它可以通过 performance.now 来获取，它的单位也是毫秒。但是同样是毫秒， performance.now 返回的变量小数点后面 10 位以上是有的，这里面我列了一个简单的转换式，大家可以换算一下，是可以精确调纳秒级别的。这就是高精度时间的第一个特性。

为什么叫高精度时间，就是精度非常之高，也是为了适应现在前端一些复杂的一些性能衡量的场景，包括一些复杂的动画场景，需要的一些高精度的定义。

那么 High Resolution Time 的另外一个特性就是，当我们在用户界面获取当前时间，然后修改一下系统时间，再次调用同一个方法就可以获取当前时间。

- 使用标准的API —— High Resolution Time, HRTime

```
> Date.now()
<- 1543783578167
> performance.now()
<- 16289.9000000034
```



为了方便大家去对比，我列了一个辅助线，可以看到，同样是调用 Data.now 这个 API 获取系统时间，如果我们修改了系统时间，在获取时间的时候，我们拿到的是，就是当前对应的系统时间。比如说我们先获取一次，修改系统时间再获取一次，Data.now 就变成了昨天的时间，但是与此同时，HR.time 其实是不会受这个系统时间变更的影响的，它依然会单调的递增，这个是 HRtime 的第二个特性，它是一个单调递增的。

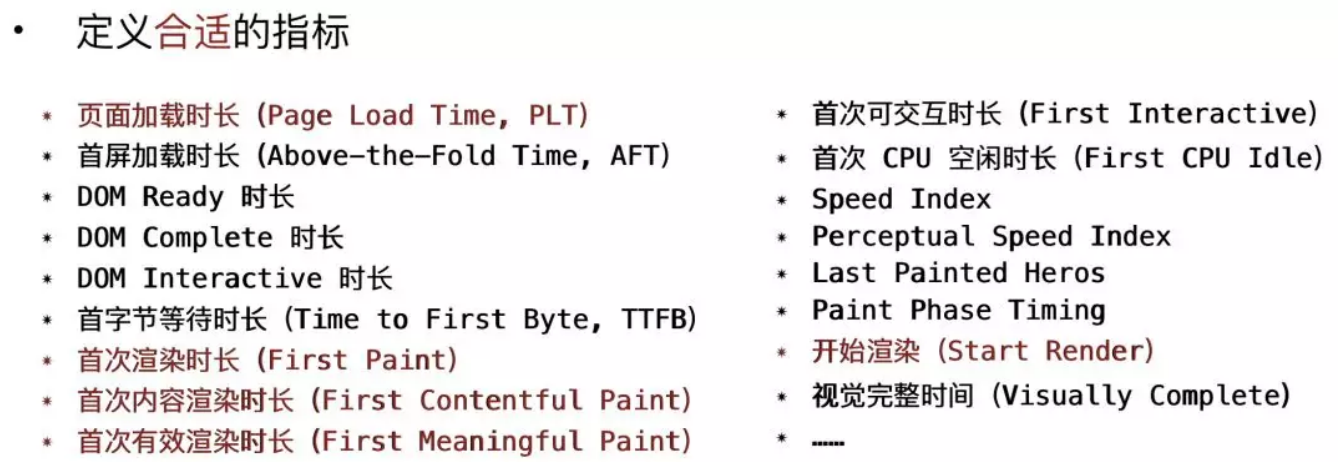
所以综合来说，HRtime 有两个特性，一个高精度，一个单调递增，保证了我们在提取性能指标的时候，不会受宿主环境的一些时间影响。

然后在 HRTime 之上，是一个叫 PerformanceTimeline 的 API，这个 API 很简单，它只是去定义了把各种各样的性能的情况。

在此，我们得出的最佳实践是：**采集性能数据时先抹平 Navigation Timing spec 差异，优先使用 PerformanceTimeline API(在复杂场景，亦可考虑优先使用 PerformanceObserver)**。

定义合适的指标

讲完使用标准的 API，我们来看一看，怎么定义合适的指标，假设有一天你老板问你说，我们页面性能怎么样，你回答他，反正我觉得打开挺快的，这是一个非常不严谨的论述。我们到底怎么样定义我们的页面性能呢，其实业界有非常多的方案，这里只是列举了十几个相对来说比较常见的一些的性能指标定义方式。



那么在这么多方案里边，到底哪些指标是适合我们的，我挑了几个具有典型代表意义的指标例子，给大家做个详细的讲解。

首先我们来看一看大家最熟悉的页面加载时长，页面加载时长是被清晰的标在这个页面的底部的。它是指 DOM load 事件触发完成，它的优点有：

- 原生 API；
- 接受度高；
- 感知明显（浏览器 Tab 停止 loading）。

缺点是：

- 无法准确反映页面加载性能；
- 易受特殊情况影响。

为了解决这个问题，W3C 的工作小组引入了首次渲染 / 首次内容渲染。首次渲染是指第一个非网页背景像素渲染，首次内容渲染是指第一个文本、图像、背景图片或非白色 canvas/SVG 渲染。

这里以打开 YouTube 为例，因为打开这个网站可能相对反应会慢一点，我们能更容易发现这种区别。当你去采集这两个指标的时候，你会发现，或者说你在大部分时候会发现，这两个指标没有任何差异。

根据我们实际采集到的性能数据也证实了我们的观点，在 70% 的情况下，First Paint 和 First Contentful Paint 他们两个指标之间的差异小于一百毫秒，为什么？因为现在一个前端网站的架构设计是倾向于单页应用的，它原本的 HTML 结构非常小巧。在渲染的时候，尤其是在第一次渲染的时候，它就已经能够把文本和背景一起渲染出来了，这两个指标差异会非常之小。

我们可以总结一下，这两个指标相比于页面加载时长它更聚焦于页面元素的渲染，相对来说更客观，但同时可以看到，页面上有象素被渲染出来，并不一定代表着用户去看到了它关心的主要内容，在实际的经验中也可以看到，大多数时候，这两个指标的相差并不是特别大。

那么页面加载时长会有异常情况，First Paint 和 First Contentful Paint 又会有各种差异不大，或者是不能够完全代表这个页面性能的情况，于是就有下面一个算法，通过算法计算出来的指标，叫做 First Meaningful Paint，首次有效渲染时长，这个指标最早是由 Google 提出的，它的一个核心的想法是渲染并不一定代表着用户看到了主要内容，Load 也不一定代表用户看到主要内容，那用户什么时候能够看到主要内容呢？我们假设当一个网页的 DOM 结构发生剧烈的变化的时候，就是这个网页主要内容出现的时候，那么在这样的一个时间点上，就是用户看到主要内容的一个时间点。

具体，可以查看此链接：

https://docs.google.com/document/d/1BR94tJdZLsin5poeet0XoTW60M0SjvOJQttKT-JK8HI/view#heading=h.ycg9fbz776q3

它的优点是相对校准的估算出内容渲染时间，贴近用户感知。但缺点是无原生 API 支持，算法推导时 DOM 节点不含权重。

然后最后一种指标叫做开始渲染时间，Start Render Time，这个指标时间是没有办法或者很难通过 JS 在用户环境执行来获取的。因为它的定义非常简单，就是假设我们开始加载页面，一直拿个照相机对着屏幕拍，直到拍到有一帧，发现这个页面不一样了，那么这个时间就是页面的一个开始渲染时间。这个定义看起来和那个刚才讲的一个 First Paint 非常像，但实际上，因为它的采集方式不一样，First Paint 还是通过浏览器的渲染引擎来计算出来的时间，而 Start Render Time 就是客观观察到页面的一个加载变化的时间，所以可能 Start Render Time 能够更客观地反映我们页面的加载情况，但是因为这个指标很难通过 JS 计算，所以是仅仅作为一个参考。

讲了这么多性能指标的定义方法，我们做一个简单的总结。

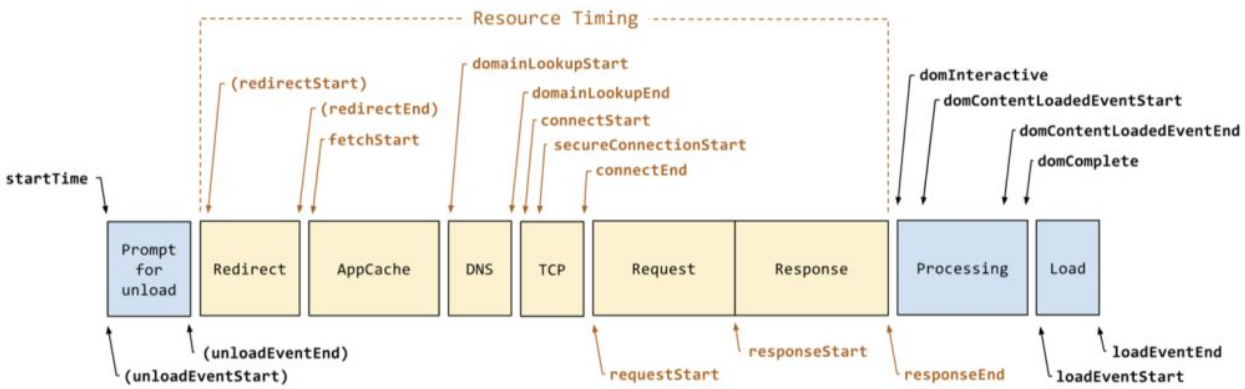
性能指标类型	👍 优点	👎 缺点	🤠 典型代表
DOM 相关	兼容性高，易采集	无法准确反映性能	load、dom ready、dom complete
Spec 相关	官方支持，相对易采集	部分场景无法准确反映性能	fp、fcp
算法相关	兼容性高，易采集，相对准确	部分场景存在计算误差情况	fmp、speed index
视觉相关	最直观的反馈页面加载性能	无法通过 RUM 方式采集	start render、visual complete

这么多的性能指标，我们到底应该怎么选呢？我们得出了最佳实践：**根据业务特性及性能监控方案选择最适合的性能指标，必要情况下可使用自定义性能点位。**

怎样采集正确的数据？

定义完合适的指标，我们来看看怎么样采集正确的数据。

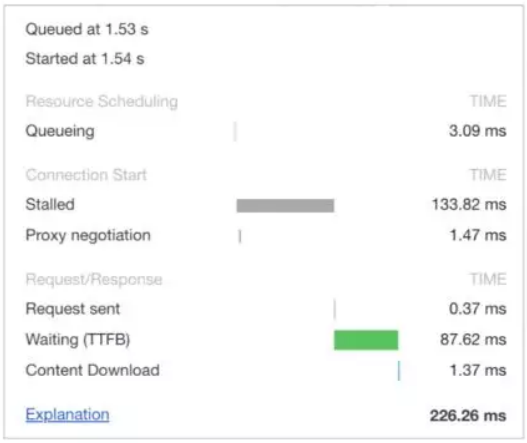
不同阶段之间是连续的吗？



这是目前最规范的一个 RUM 性能模型，可以看到我们的页面加载被定义成了很多个阶段。

很多做性能监控的方案都是计算阶段再上报数据，当然这可以减少数据传输量，但是在实际的应用中我们会发现，这样做是有问题的。

比如说 DNS 查询完，马上就开始建立 TCP 连接了吗？TCP 连 接完马上就开始去发送这个请求了吗？实际上是不一定的，除了我们刚才模型定义的阶段之外，会有一个叫做 stalled 的时间，那么这个阶段发生了什么事情呢？

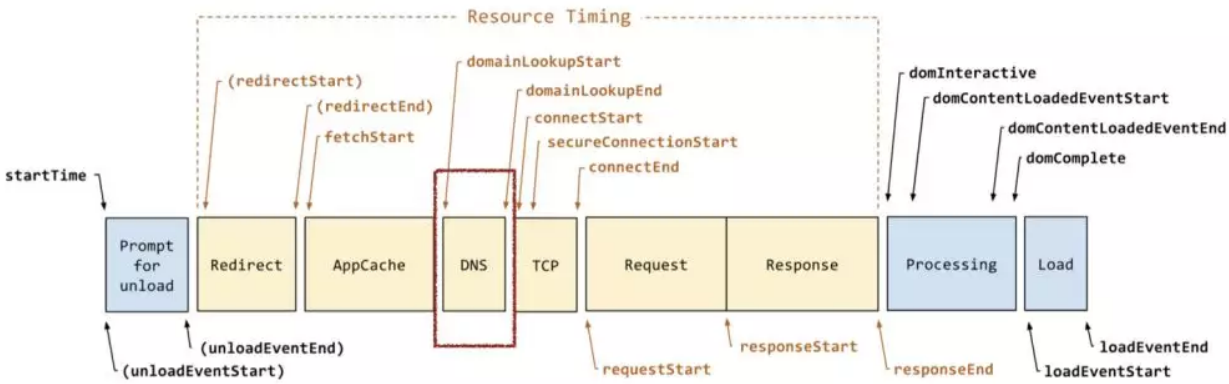


- 同域名 TCP 连接并发限制
- 浏览器正在分配缓存空间
- 当前有更高优先级请求正在处理
- ……

当你观察到有这种 blocked 或者 stalled 的时间出现的时候，大部分情况下，都是因为被同域名 TCP 并发连接限制了，就比如说，我们一个页面可能加载了 CDN 上的十个资源，那么在 TCP 连接会有限制，比如 Chrome 它就会有限制。单页面、单个域名的 TCP 连接，并发数是 6，也就是说加载十个资源，只有六个连接被真正建立了，剩下四个资源是要等待这六个连接消费完成之后才能被复用的。

还有其他的一些情况也会导致这个页面 stalled 的时间产生，如果我们只是用刚才那个性能去计算某一个阶段的时间的话，就会把这个 stalled 的时间漏掉，这样会导致我们真正把各个阶段加起来去算这个页面总的加载时间的时候，出现一个误差。在实际观察中我们会发现，这种 stalled 时间是非常常见的。

每个阶段都一定会发生吗？



一个最典型的例子，DNS 查询，每次页面访问都会去查 DNS 吗？我们知道现在浏览器会对 DNS 查询结构做缓存了，你第一次访问了淘宝，你第二次再访问的时候，很大概率 DNS 是不会再查询了的，那么这个阶段可能算出来就是 0，那么在各种特殊的情况下，我们怎样去采集这个正确的数据呢？

我们目前得出的结论是：**上报页面加载开始时间，以及后续各时间点相对增量，在数据端进行阶段清洗和异常处理。**

比如说页面开始加载时间是 0 秒，那么从 0 项开始时间是一毫秒，TCP 开始时间是两毫秒，把各个时间点相对的增量算出来。

最后在数据端进行一个阶段的清洗，以及异常的处理，这样一方面能够保证在规范定义中没有被体现出来的阶段不会被遗漏掉，另一方面也能够让我们去掌控这个页面加载的分布到底是怎样的，也不会因为我们在前端就把这个页面算好，遗漏导致数据的丢失。

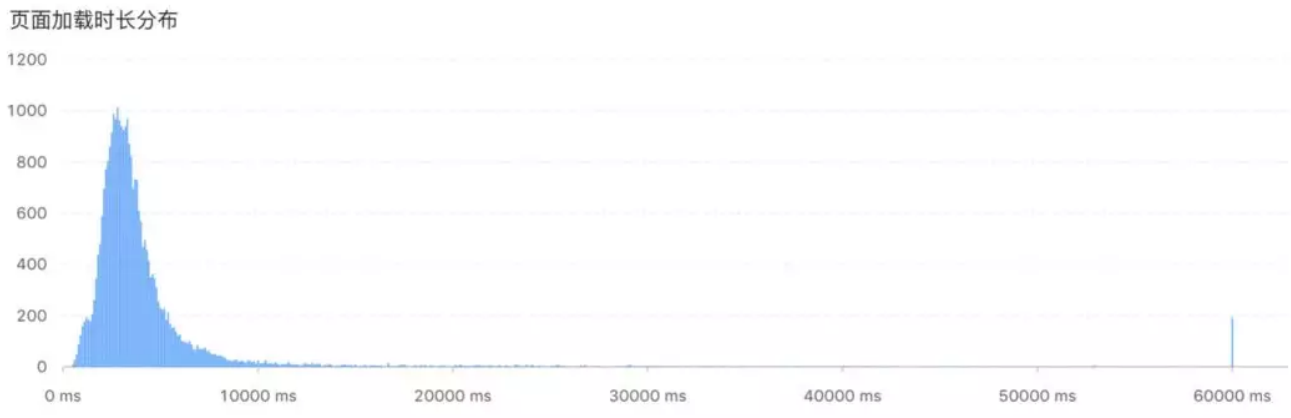
采集完合适的指标，最后一个是上报关联的维度。

我们都知道在做前端的数据采集的时候，维度数据是非常重要的，除了我们刚才定义的各种度量，怎样采集到合适的相关维度，也能够极大地帮助我们分析页面性能的效果。

上图提到的点都是必须要采集的，但是在分析页面性能的时候，有很多相对专业的维度是会被大家忽略掉的，比如说当前页面是否可见，这个页面加载方式是怎么样的，它是直接打开，还是刷新打开，还是前进后退打开等等。就是通过后面的数据分析，我们会发现，不同的页面操作，页面打开方式都会对我们页面加载的性能会有影响，以及一些更复杂的，比如说是否启用 HTTP2、Service Worker 等等，这些数据我们都应该尽可能采集到，从而能够更好的去分析我们的页面性能。

准确分析性能数据及影响因素

讲完前端数据的一些采集，我们来看一看怎么样去分析这些性能数据。这个是我们采集到真实的蚂蚁的某一个业务的页面加载时长数据。可以看到大部分的页面都会在 10 秒内完成这个页面的加载，但是仔细观察就会发现，还会有大量的长尾值存在，可以看到从 10 秒到 30 秒，甚至到 60 秒，都会采集到这样的数据，这也是一个非常真实的情况。



这个数据可以从两个方面进行解读，第一，由于页面加载时长会受很多异常因素的影响，你的用户可能已经在正常地使用你的网站业务，只不过一个 icon 没加载出来，一个小图片没加载出来，就会导致这个指标变得超长，这都是合理的情况。而另外一种情况就是在分析性能数据的时候，我们会发现，长尾的数据是非常常见的，因为影响外部性能因素太多了，除了前端自己的原因，除了我们 JS 写的太大，静态资源太多，没有压缩等等，比如用户的网络不好，服务器压力突然很高，甚至 IO 堵塞了，都可能导致最后这个页面没有被渲染出来。

在蚂蚁内部，我们开玩笑说，一些体量比较小的业务，可能有一天它的用户蹲在厕所里面用手机去刷了一下它的网站，页面加载时长统计出来的数据就要暴涨，这都是非常真实可能存在的情况。

我们理解这些长尾数据之后，来看一看还会有哪些影响我们这个页面加载性能的因素，我刚才在那个上报维度里面讲到了，页面的可见状态是一个非常大的影响因素，从 Chrome 68 开始，Chrome 启用了一种全新的浏览器的架构模式，在这种模式下，其实它对不可见 Tab 的硬件资源的分配做了一个极大的限制。

你会发现，在你的不可见 Tab 下，可能你的动画会被减慢。那么比如说用户他在后台打开你的页面，就是说它在前一个页面，比如说右键，先打开，后面右键后台先打开，这个时候在

你这个新打开的页面去提取性能数据的时候，会发现那个性能就没有正常打开，相对来说就没有那么好。

另外一种页面加载方式也会影响我们页面性能，比如我们正常打开一个页面会去加载各种各样的资源，如果刷新打开，可能有一些缓存，浏览器帮我们做掉了，一些静态资源，304 没有更新，游览器就不会重新下载，比如说前进后退的时候，甚至浏览器会帮我们做一些页面内容的缓存，有些页面前进后退都是秒出的，根本没有出现那种页面完整刷新的效果，就是这种不同的页面加载方式，会影响到我们页面的性能数据的采集结果。

最后一种就是 Service Worker，大家如果实践过就会发现，Service Worker 能力非常强大，它可以做非常强大的缓存，这是可能会影响到整个页面的一个性能。

数据是如何撒谎的?

最后一个我想讲的是，我们性能数据是怎么样撒谎的，还是以刚才这个采集到的性能直方图为例，老板问说，你性能采集到了，你给我一个数，你们这个页面性能到底怎么样？我说我算了一下，大概是 6 秒左右。

老板不开心了，说这个数据怎么这么大？然后我们可以通过一些小技巧，一些不同的加工方式，又算了一下，算出来是 4.2 秒左右，老板说还可以再好一点。又换一种加工方式，2 秒 5 左右，这个数字不错，同样的一份数据，原始的直方图就摆在这里，同样的一份数据，为什么会得出三个截然不同的结论呢?

这是因为我们指标计算的口径不一样，第一个数据，特别大的数据是 95 分位数，所谓 95 分位数就是把我们所有的页面加载时长，从小到大排列，取第 95% 的位置，这个值作为我们的性能指标，那么使用 95 分位数，隐含的意义就是，我们承诺 95% 的人访问我们页面的时候，页面加载时长是小于 6 秒的。后面一种鸡贼的方法就是我们取平均数，缓存特别好的，可能几十毫秒就打开了，缓存不好的几十秒才打开，我们取平均一下，看起来也很中立，4 秒多；如果我们再鸡贼一点，比如说我们长尾数据特别长，我们还可以取 10% 的截尾平均数，掐头去尾取平均，这样的数据看起来就非常漂亮。

但是在实际上，如果我们做的是一个给自己看的性能产品，我们应该是对自己的性能数据负责任，所以，在实际分析性能指标的时候，我们是建议 **分析性能指标时建议关注百分位数(percentile)，对性能的要求越高，使用越大的百分位数。**

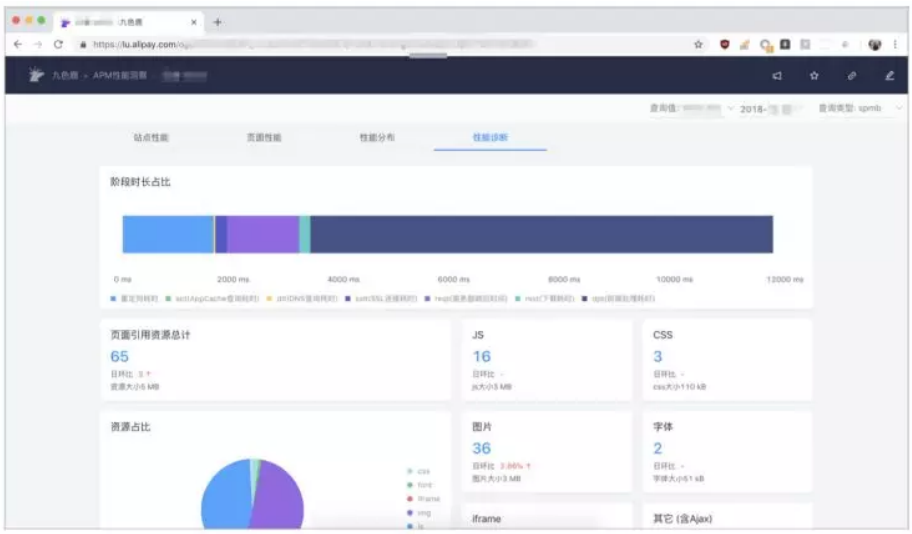
比如我们要承诺 99% 的用户都要小于 5 秒，我们看页面加载时长时候就应该看 99 分位数。如果我们现在精力不够，我们只能承诺 50% 的人页面加载时长小于 5 秒，实际上 50 分位数，就是中位数，就是 50% 的访问能够不小于这个时间打开这个页面。

总结

最后一点，我想讲的是在蚂蚁金服，我们是怎么用这些性能数据的。

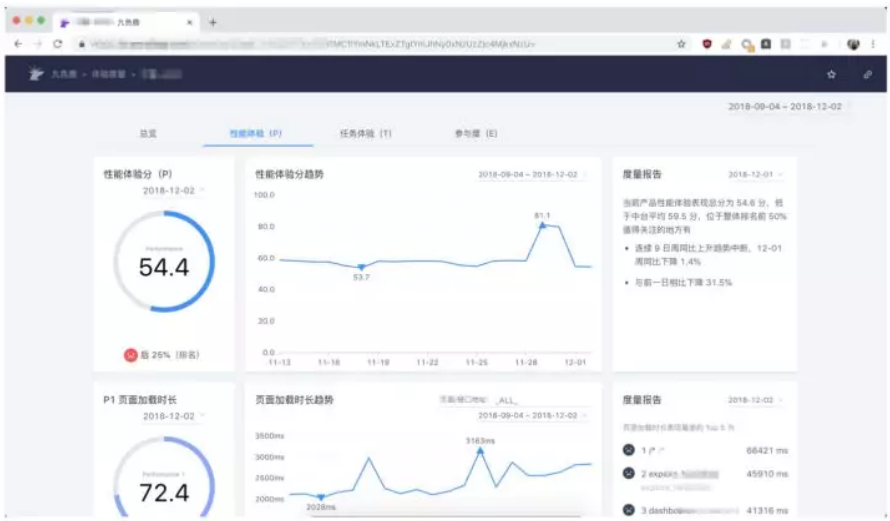
首先性能数据采集当然是要给用户一个分析的结果，因为这是一个性能分析的产品，我们要从各种维度，各种时间和刚才讲到的关联的维度，给用户一个分析的抓手，告诉它你的性能瓶颈在哪里，你的综合的情况怎么样的，让用户感知他的业务到底是快还是慢。

一些客观的数据能够了解自己业务的综合情况，但是看到自己的数据还不够，我们在蚂蚁金服在做另外一件事情，就是对所有中后台产品做一个性能数据的横向比分，当然不仅仅是性能数据，这个产品是综合的一个叫体验分的东西，体验包括了性能数据，同时也包括了页面的美观度，用户的感知度，以及一些任务的转化率等等，其中这个性能的分数可以通过我们采集到的性能的指标来反馈。然后当然你的可能有人就会觉得说，我们业务形态不一样，我们的前端架构方式不一样，你把这些分数横向比较有什么意义？



蚂蚁金服体验分析产品
九色鹿

这个时候就涉及到性能指标的定义，应该尽可能的选取对，如果我们是一个性能监控的中台，而不是一个简单的对自己业务性能的一个监控，我们应该尽可能的选取一个中立的指标去衡量，就是客观的衡量各种业务的一个展示形态，同时在做一个横向比较的时候，我们也会换用不同的性能计算方式，比如说在这个体验分项目里面，我们就会用截尾平均值，目的并不是为了让这个数据变得好看，是为了让这个数据变得平稳，让异常值不会产生太大的影响。同时我们在一些横向比较中，或者在这种大盘的比较项目中，我们也会尽可能的去让用户关注自身性能的一个提升，你可以不关心你比别人快还是慢，但是你要关心你比之前快了还是慢了。



性能数据助力蚂蚁金服中后台产品体验升级

以上就是整个性能数据的分享以及在蚂蚁金服的应用，谢谢大家。

演讲视频