美团

2022美团技术年货

CODE A BETTER LIFE

—— 前端系列 ——

目录

前端

知识图谱可视化技术在美团的实践与探索

作者：巍耀

1. 知识图谱可视化基本概念

1.1 知识图谱技术的简介

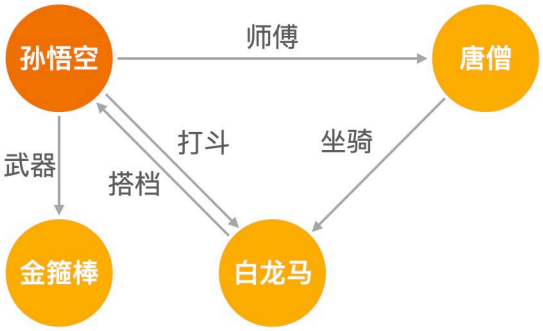
知识图谱（Knowledge Graph）是人工智能的重要分支，它是一种揭示实体之间关系的语义网络，可以对现实世界的事物及其相互关系进行形式化地描述。举个例子，“孙悟空的师傅是唐僧”就是一条知识。在这条知识里，有“孙悟空”和“唐僧”两个实体，“师傅”是描述这两个实体之间的关系，上述内容在知识图谱中就组成了一个 SPO 三元组（Subject-Predicate-Object）。

所以，对于现实世界中实体之间的关联关系，用知识图谱进行描述的话，就显得非常合适。正是由于知识图谱的这种优势，这项技术得到迅速普及，目前在搜索、推荐、广告、问答等多个领域都有相应的解决方案。

1.2 知识图谱可视化的简介

可视化，简单来说就是将数据以一种更直观的形式表现出来。其实，我们现在常用的折线图、柱状图、饼状图（下称折柱饼），甚至 Excel 表格，都属于数据可视化的一种。

以往，我们存储数据主要是以数据表的方式，但这种方式很难结构化地存储好知识类型的数据。对于关系类型的数据，如果用前文的例子为基础并补充一些相关信息，经过可视化后就能展示成这样：



西游记中人、物关系

这种信息就很难用“折柱饼”或者表格呈现出来，而用知识图谱可视化的方式呈现，就非常的清晰。

2. 场景分析与架构设计

2.1 场景需求分析

我们梳理后发现，在美团的各个业务场景中知识图谱可视化需求主要包含以下几类：

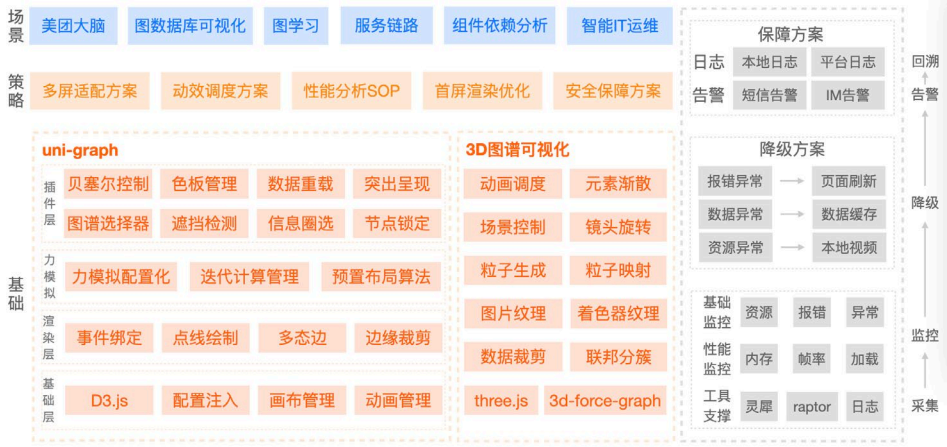
- **图查询应用**：以图数据库为主的图谱可视化工具，提供图数据的编辑、子图探索、顶点 / 边信息查询等交互操作。
- **图分析应用**：对业务场景中的关系类数据进行可视化展示，帮助业务同学快速了解链路故障、组件依赖等问题。
- **技术品牌建设**：通过知识图谱向大家普及人工智能技术是什么，以及它能做什么，让 AI 也具备可解释性。

2.2 技术选型与架构设计

在图关系可视化上，国内外有很多图可视化的框架，由于美团的业务场景中有很多个性化的需求和交互方式，所以选择了 D3.js 作为基础框架，虽然它的上手成本更高一些，但是灵活度也比较高，且功能拓展非常方便。D3.js 提供了力导向图位置计算的基础算法，同时也有很方便的布局干预手段。于是，我们基于 D3.js 封装了自己的知

识图谱可视化解决方案——uni-graph。

整体的功能与架构设计如下图所示，下面我们会介绍一些 uni-graph 的功能细节和可视化的通用技术策略。

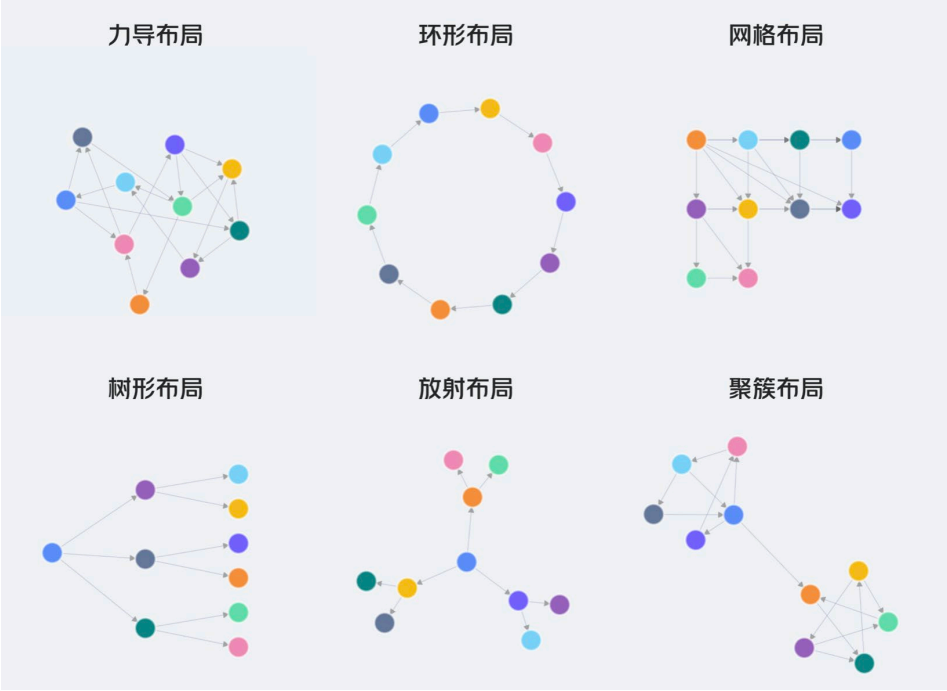


架构图

3. 技术挑战与方案设计

3.1 布局策略

在不同类型的知识图谱中，因数据差异较大，对布局效果的要求也有所不同。能让业务数据有合适的布局来做可视化呈现，是一项比较大的技术挑战。除了下面几种基本的布局之外，我们还探索了一些特定场景下的布局方案。

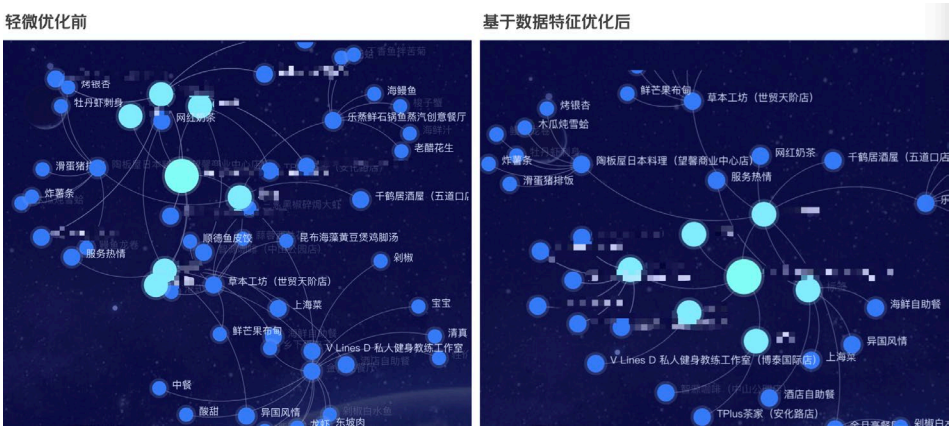


布局策略 – 基础布局

提取数据特征优化布局

D3.js 提供的力导向图模块（d3-force）实现了一个 velocity Verlet 数值积分器，用于模拟粒子的物理运动。在不做过多干预的情况下，会根据节点与边的关系模拟物理粒子的随机运动。D3.js 的力导向图提供的力学调参项主要包括 Centering（向心力）、Collision（碰撞检测）、Links（弹簧力）、Many-Body（电荷力）、Positioning（定位力）。

如何针对不同的节点进行合适的力学干预，是让布局更符合预期的关键。一般来讲，同一业务场景的图谱结构都具有一定的相似性，我们考虑针对业务特定的数据结构特征来做定制化的力学调优。这里举一个简单的场景进行说明，我们抽象出了在树中才有的层级和叶子节点的概念，虽然部分节点会互相成环，不满足树的定义，但是大部分数据是类似于树的结构，这样调试后，展示的关联关系就会比随机布局更加清晰，用户在寻找自己需要的数据时也会更快。



布局策略 – 基于数据特征优化

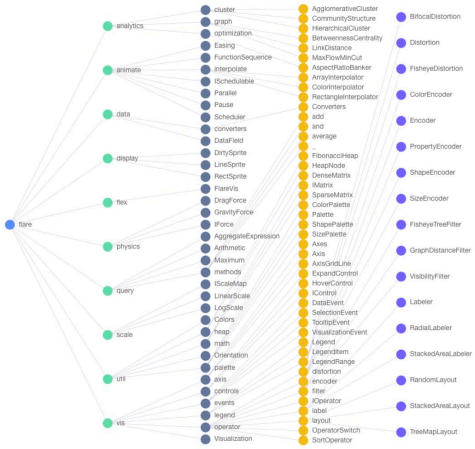
其实，美团的各个业务场景都有个性化定制布局的需求，这里只是拿其中一个最简单的场景进行说明，uni-graph 能够将力学参数调整的模块独立出来，并且梳理出一些常用的参数预设，可以支撑很多场景的布局优化。

层级数据布局方案

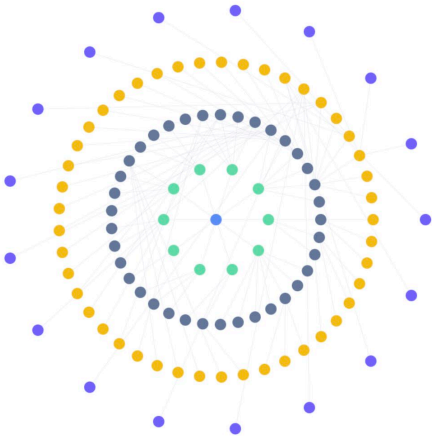
在很多业务场景中，用户更倾向于采用分层的方式来观察图谱数据，因为这样有利于理解和分析图谱数据，比如：根据用户探索路径分层、边关系聚合分层、业务属性归类分层、指定中心点路径分层等等，这些需求对图谱的样式和布局形式提出了更高的要求。

得益于 D3.js 力学布局的灵活性和拓展能力，我们在业务实践的过程中实现了几种常用的布局方案：

平铺层布局



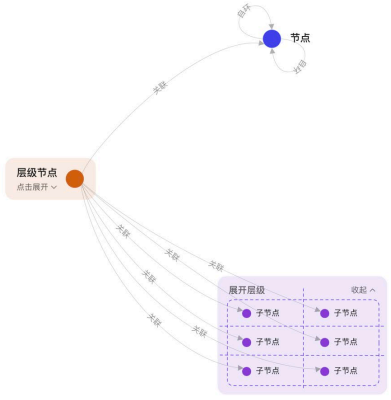
环形层布局



布局策略 – 层级布局 –1

聚簇层布局

网格层布局



布局策略 – 层级布局 –2

以聚簇层布局为例，我们简单介绍一下实现过程：

首先处理图谱数据，将中心节点关联的子节点按关联关系归类，生成聚簇边和聚簇边节点，同时将子节点分层。还需要自定义一种聚簇力，聚簇力包含三个参数ClusterCenter、Strength、Radius，对应聚簇中心、力的强度、聚簇半径。在

力学初始化时，我们为每个子节点定义聚簇中心节点和聚簇半径。最后在力学布局的 Tick 过程中，先计算子节点与其聚簇中心节点坐标偏移量，然后根据偏移量和聚簇半径的差值来判断节点的受力方向和大小，最终经过向量计算得出节点的坐标。

布局参数配置化

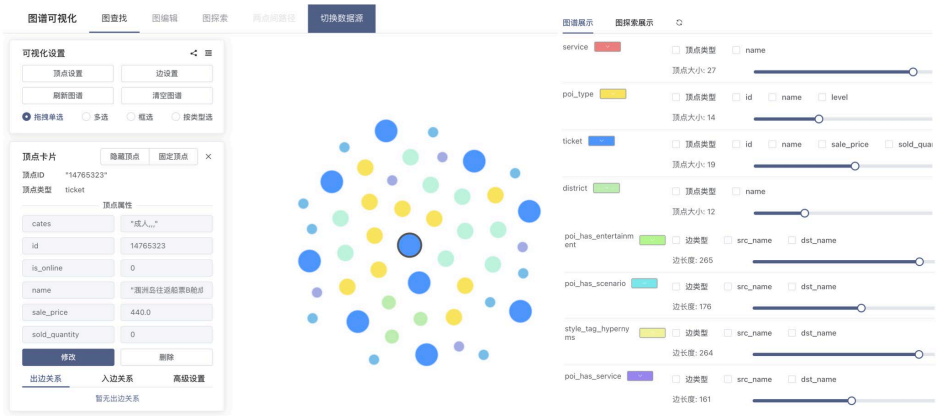
在特定领域的图谱可视化中，通常采用一两种布局即可满足用户的展示需求，因为这些场景下的图谱的关系结构相对固定。但作为平台性质的工具，就需要展示多个领域的图谱。为了更清晰地展现出各领域图谱的特点，布局形态就需要跟随图谱而变化。

针对这种场景，我们实现了多项布局参数的配置化，用户可以根据领域图谱的特点优化布局参数，并作为配置保存下来。

领域图谱可视化 – 网格布局参数调整

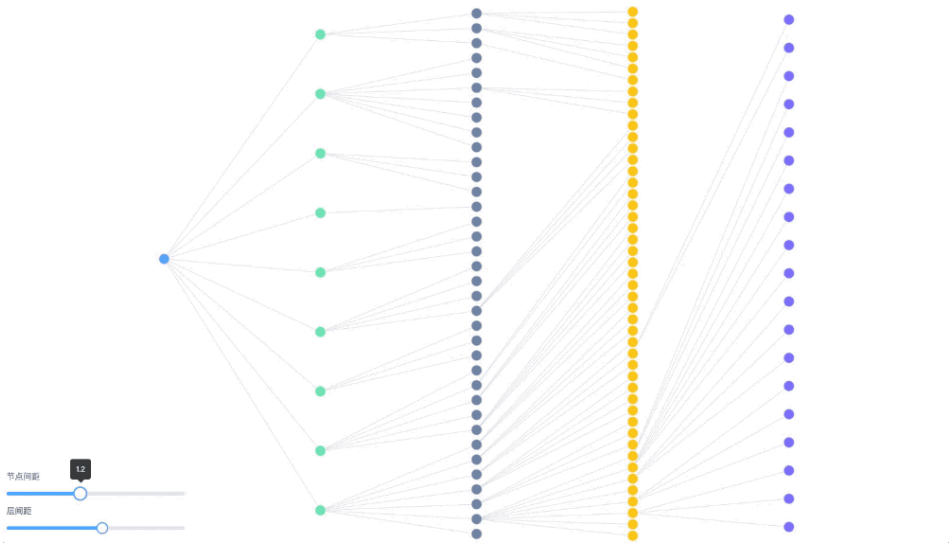
布局策略 – 参数配置化

图数据库可视化 – 布局样式参数调整



布局策略 – 图数据库

服务链路可视化 – 平铺层布局参数调整

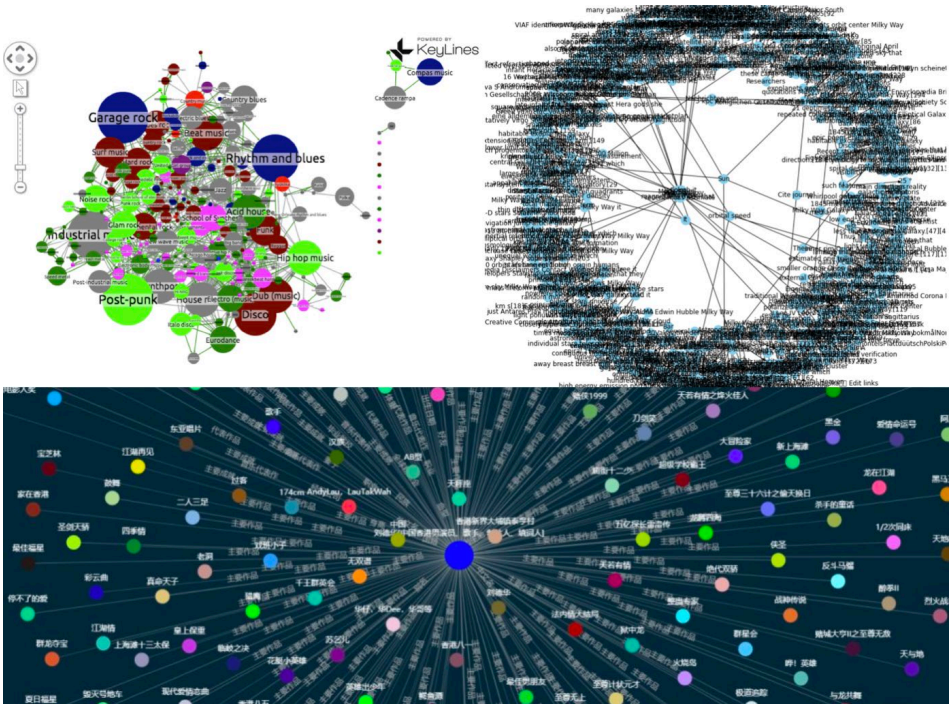


布局策略 – 服务链路

3.2 视觉降噪

在用户使用可视化应用时，文字 / 节点 / 边等元素内容混杂在一起，如果没有做好信

息的表达和呈现，会直接影响到用户的使用体验和使用效率。经过分析，我们发现这是因为在可视化过程中产生的视觉噪声太多，而通过可视化带来的有效信息太少。下面将举例展示什么叫做视觉噪声：

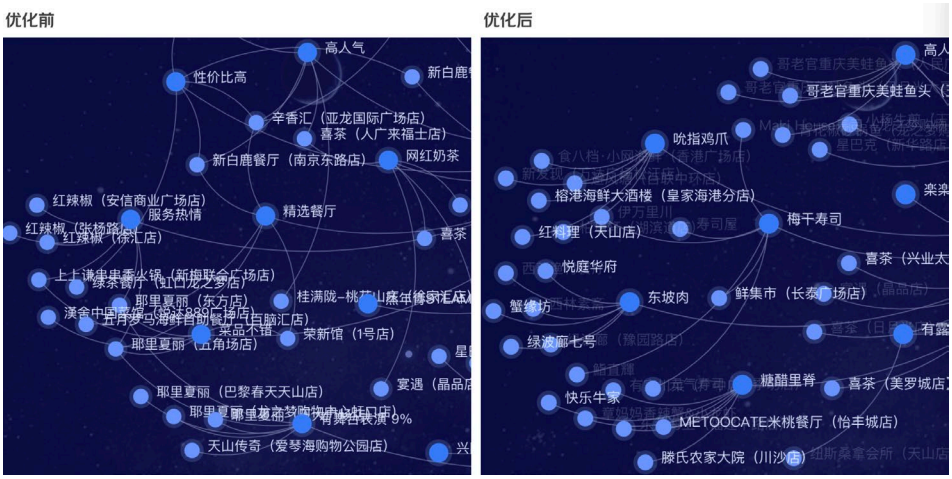


视觉降噪 – 视觉噪声

在以上几张图中，虽然将知识图谱的数据呈现了出来，但是元素数量非常多，信息杂乱，给用户的观感是“眼花缭乱”。下面我们会介绍针对这类问题的解决方案。

文字处理

文字主要用在节点和边的描述上，虽然它能提供非常重要的信息，但是节点多了后，文字会在所难免的相互重叠，而重叠后的文字很难快速识别出来，不仅起不到传递信息的作用，反而会造成很差的视觉体验。为此，我们需要对文字进行遮挡检测，根据文字的层叠关系，将置于底部的文字透明度调低，这样即使多层文字重叠后，置于顶层的文字依然能被快速识别。

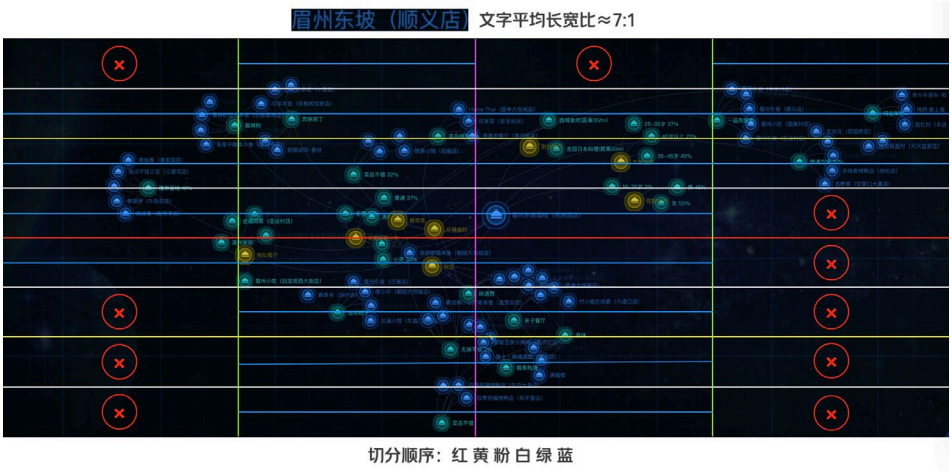


视觉降噪 – 文字 – 对比

但这种解法的时间复杂度会随着节点的增多逐渐变得不可控。假如我们有 100 个节点，最多需要 O(n!) 的时间复杂度才能计算完毕。我们这里采用栅格划分的方式来做优化，先对画布进行栅格划分，然后确定节点所在的一个或多个栅格，在进行碰撞检测的时候，只需要和自己同栅格的节点做对比即可，因为不同栅格内的节点一定不会出现碰撞的情况。

视觉降噪 – 文字 – 栅格划分

这种栅格划分的理论基础就是四叉树碰撞检测，我们在此基础上做了进一步的优化。由于需要进行遮挡检测的元素是文字类型的节点，这种节点的特点是长比宽大很多。如果按照传统的四叉树分割算法，就会造成很多文字节点横跨多个栅格，对比的次数较多。在检测前，我们先计算出所有文字节点的平均长宽比，每次栅格划分是横向还是纵向，取决于哪个方向划分后栅格的长宽比更靠近文字的平均长宽比，这样做就会减少文字节点横跨多个栅格的情况，从而减少了每次需要被碰撞检测的节点数量。



视觉降噪 – 文字 – 四叉树

边处理

多边散列排布

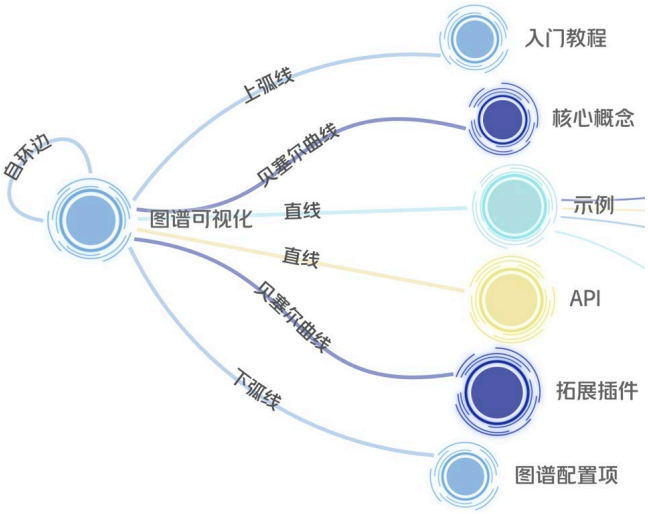
知识图谱中存在包含大量出（入）边的中心节点，在对这些中心节点的边进行可视化展示时，往往会出现边与中心节点联结处（Nexus）重叠交错在一起的情况，进而影响视觉体验。

针对这种特殊场景，我们设计了一种多边散列排布的算法，通过边夹角偏移量计算和节点半径裁剪，将 Nexus 分散排布在节点周围，减少边线重叠的情况，以达到更清晰的视觉效果：

边处理 – 散列排布

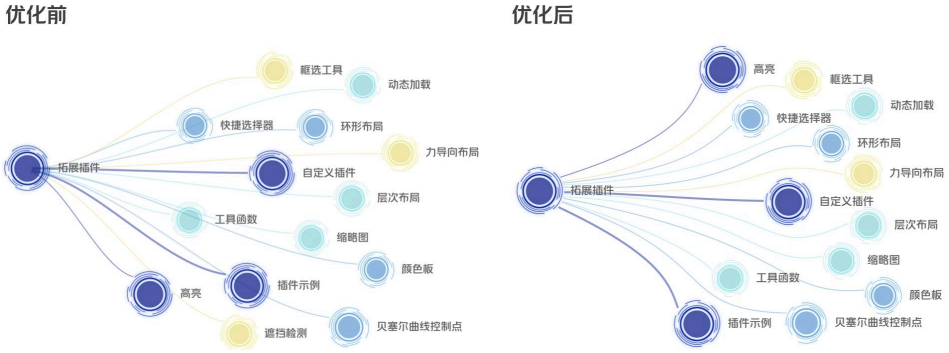
多类型可调节边

我们还实现了多种类型的边，并支持通过参数配置的方式来调整边的样式，比如：贝塞尔曲线控制点、弧度、自旋角度等参数，以满足各种复杂图谱的可视化场景。



边处理 – 多类型边

通过多边散列排布，改变边线类型，并调整样式参数，下面是我们将图谱中凌乱无序的边线优化后的效果：



边处理 – 最终对比

3.3 交互功能

合适的图谱布局能更好地表达出数据的含义，通过视觉降噪可以进一步让图谱传递出更多的有效信息。但是用户依然需要通过交互找到自己关心的信息，一个图谱可视化工具是否好用，交互功能会起到非常重要的作用。目前，我们实现了下面的基本交互功能：

- **画布操作**：拖动、缩放、动态延展、布局变换、多节点圈选。
- **元素（节点和边）操作**：样式配置、悬浮高亮、元素锁定、单击、双击、右键菜单、折叠 / 展开、节点拖动、边类型更改。
- **数据操作**：节点的增删改查、边的增删改查、子图探索、数据合并、更新重载。

除了上述的基础交互功能外，我们还探索了一些特殊场景的交互。在图谱可视化中交互的目的，是为了从庞大的知识图谱中找到自己关心数据的关联关系，同时也能够观察到这些关联关系在全局画布中的位置。

路径锁定

通过选取不同的节点，自动计算出节点之间的合适路径，做锁定展现，方便观察两个或多个节点是如何关联起来的。

交互功能 – 路径锁定

聚焦展现

对于当前不关注的图谱区域，默认布局可以密集一些来节省画布空间，关注某个区域后，会对当前关注的一小块区域重新布局，让节点排布分散一些，方便查看文字的内容。

交互功能 – 聚焦展现

其实，无论可视化的节点与边的数量有多庞大，当深入到业务细节中的时候，使用者关注的节点数量其实不多，重点是把使用者关心的数据从大量数据中筛选出来，并且

做好清晰地呈现表达。

3.4 美团大脑可视化

美团大脑 – 主界面

美团大脑是围绕吃喝玩乐等多种场景，构建的生活娱乐领域超大规模知识图谱，为用户和商家建立起全方位的链接。为了让美团大脑的能力更容易的被理解和使用，我们需要通过知识图谱可视化的方式让美团大脑更具象化，并开发出便捷的知识图谱查询应用。

在开发知识谱图可视化功能之前，还需要具备一些通用可视化能力。下面主要介绍一下多屏适配和动画相关的技术能力。

多屏适配方案

美团大脑呈现的终端场景非常复杂，有 PC/Mac 端屏幕、大屏电视、巨型宽屏等。各个屏幕的尺寸比例都有所不同，为了保持统一观感，不能出现滚动条、不能有边缘留白、不能压缩变形。同时在一些重要场合的巨型宽屏上，还要对细节做特定的适配。通过 sass 的函数和 mixin 功能可以较好地完成非等比缩放和个性化适配的需求。

非等比缩放

```scss
$design_width: 1920;
$design_height: 1080;

@function cvh ($px) {
    @return $px/$design_height*100vh;
}

@function cvw ($px) {
    @return $px/$design_width*100vw;
}
```

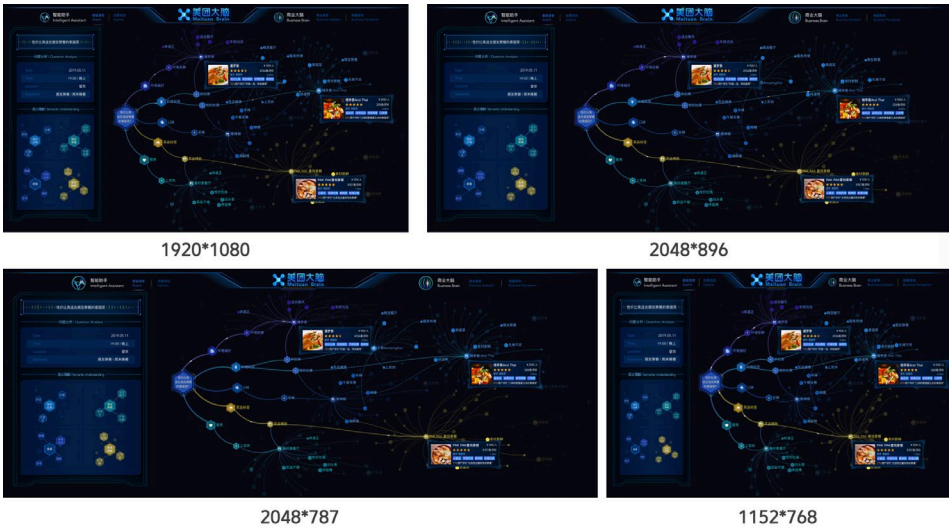
个性化适配

```scss
$media-command-center: '(min-aspect-ratio: 5760/2160)';

@mixin cc () {
    @media screen and #{$media-command-center} {
        @content;
    }
}

.panel-card {
    font-size: .14rem;
    margin-right: cvh(10);
    @include cc {
        font-size: .16rem;
    }
}
```

code-sass

- **非等比缩放**：在长宽都需要考虑的缩放场景中，使用基于视口百分比的单位 vh、vw 很合适，设计稿尺寸为 1920 * 1080，可以通过转换函数自动计算出对应的 vh、vw 值。其中为了保证字体大小在不同尺寸的屏幕上更符合预期，会用设计稿里的高为基础单位做 rem 的指导参数。
- **个性化适配**：在超宽的大屏尺寸下，按照比例自动缩放，在某些元素上视觉效果并不是特别完美，上面的 mixin 可以很方便地在 CSS 中对特定尺寸的屏幕做个性化适配。
- **像素级还原**：针对不同尺寸的设计稿校准时，有些元素会带有阴影效果或者是不规则图形，但是设计师只能按照矩形切图，导致 Sketch 自动标注的数据不准确。这时可以把浏览器的尺寸设置成与设计稿一致，再蒙上一层半透明的设计稿图片，逐个元素做对齐，就可以快速对不同尺寸屏幕的设计稿做像素级还原。

这套大屏适配技术方案支撑了美团大脑历次的版本迭代。此前在参展亚洲美食节时，由于会场搭建情况比较复杂，屏幕分辨率经历了多次变更，只花费了 0.5 人日就做到了各种不同分辨率的定制、开发和适配工作。



美团大脑 – 多屏适配

现场效果

美团大脑 – 多屏适配 – 现场

动画脚本自动化

与静态可视化界面相比，动态可视化或者交互式可视化有更好的视觉效果，并且能传递给观看者更多的信息。

静态效果

动态效果

静态效果对比动态效果

此外，美团大脑在展出过程中部分时间是无人值守的，而有了动态可视化后，还需要自动播放循环动画，因此就有了动画脚本自动化的需求：

- 在无人操作时，按照配置好的动画脚本循环执行。
- 用户与应用交互时，能够自动将动画停止。
- 有便捷的方式重新运行动画或进行任意场景的转跳。

美团大脑的动画效果具有以下几个特点：

- 动画类型多样化，包含 3D 类型、DOM Animation、SVG Animation、第三方 Canvas 组件、Vue 组件切换。
- 多个动画模块之间有衔接依赖，动画执行可以暂停和开始。
- 不同模块的动画之间需要相互通信。

我们将每个动画都封装成一个函数，初期使用了 setTimeout 和 async function 的方案：

setTimeout：可以管理简单的动画执行，但是只要前面的动画有时间上的变动，后续所有动画 setTimeout 的 delay 参数都需要改，非常麻烦。

async function：将动画都封装成返回 Promise 的函数，可以解决多个动画模块依赖的问题，这个方案对不同动画模块开发者的协作效率有很大的提升，但是依然无法暂停和取消动画。

setTimeout

```
let animate1 = () => {}; // 运行10S
let animate2 = () => {}; // 运行5S
let animate3 = () => {}; // 运行10S
let animate4 = () => {}; // 运行10S(2和3都运行结束后再运行)

setTimeout(animate1, 0);
setTimeout(animate2, 1000 * 10);
setTimeout(animate3, 1000 * 10);
setTimeout(animate4, 1000 * 20);
```

async function

```
let animate1 = async () => {}; // 10S后resolve
let animate2 = async () => {}; // 5S后resolve
let animate3 = async () => {}; // 10S后resolve
let animate4 = async () => {}; // 10S后resolve(2和3都运行结束后再运行)

let animateAll = async () => {
    await animate1();
    await Promise.all([animate2(), animate3()]);
    await animate4();
};
animateAll();
```

code-js 异步

async function 的方案已经比较好用了，但是主要问题是一旦执行就不能暂停或取消，因此我们基于 generator function 封装成了类 async function，可以做到随时暂停或取消，下面是使用封装的异步动画调度器与各种工具 helper 写的动画模块业务代码。

main.vue

```
*actionScript() {
    // 等待1秒
    yield this.$sleep(1000);
    // 背景渐入动画
    this.mainFrameClass = FLY_IN; // 设置渐入动画的class
    yield* this.$flyIn('main-frame'); // 自动监听CSS动画结束
    // 左右侧动画
    yield Promise.all([
        this.$bus.$emit('animate-left'), // 通知其他的Vue组件执行动画
        this.$bus.$emit('animate-right'), // 通知其他的Vue组件执行动画
    ]);
    // 主图谱动画
    yield this.$bus.$emit('animate-force-graph');
}

mounted() {
    // 初始化动画执行器
    this.cancelToken = new this.$ae.CancelToken(); // 取消执行器的token
    this.aef = this.$ae(this.actionScript, this.cancelToken);
    this.aef();
}
```

left-panel.vue

```
*actionScript() {
    yield animate1();
    yield animate2();
    yield animate3();
}

mounted() {
    // 初始化动画执行器
    this.cancelToken = new this.$ae.CancelToken(); // 取消执行器的token
    this.aef = this.$ae(this.actionScript, this.cancelToken);
    this.$autoOn('animate-left', async () => {
        await this.aef();
    });
}

beforeDestroy() {
    // 取消所有动画的执行，根据Vue Hook自动执行，不需要开发者手动写
    // this.cancelToken.abort();
    // this.$bus.$off('animate-left');
}
```

util.js

```
const $autoOn = function(eventName, listener) {
    // this.$bus是定制开发的支持异步的EventEmitter
    this.$bus.$on(eventName, listener);
    this.$on('hook:beforeDestroy', () => {
        // 自动取消所有
        if (this.cancelToken && !this.cancelToken.isAbort) {
            this.cancelToken.abort();
        }
        this.$bus.$off(eventName, listener);
    });
};

Vue.prototype.$autoOn = $autoOn;
```

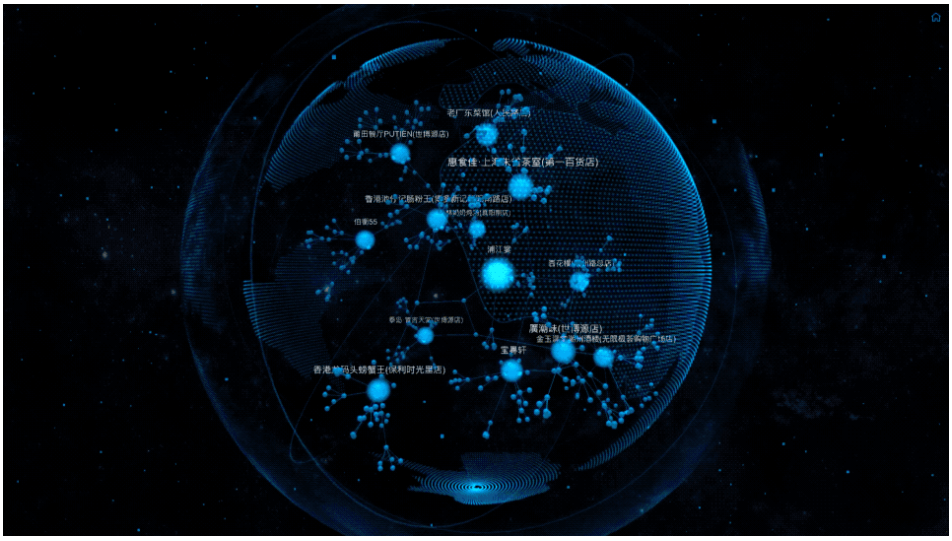
code-vue 实践

整体方案主要有以下几个功能：

- $ae 是基于 generator function 封装的异步工具库 async-eraser，CancelToken 是停止生成器运行的取消令牌。

- 定制开发了支持异步事件的 EventEmitter，emit 函数会返回一个 Promise，resolve 时就会得知 emit 的动画已经执行完毕，使 Vue 跨组件的动画调度更容易。
- Vue 组件卸载时会自动 off 监听的事件，同时也能自动停止当前组件内的动画调度器。
- 监听 DOM 的 transitionend 和 animationend 事件，自动获取 CSS 动画执行结束的时机。

通过上述方案，我们让开发动画模块的同学像写异步函数一样写动画模块，极大地提高了动画模块的开发效率，让每个同学的精力都放在动画细节调试上，下面是最终的实现效果：



美团大脑 – 总体效果

美团大脑功能交互

美团大脑 – 功能交互

因为美团大脑不仅要参加各类活动与展会，还要服务于同事们的日常工作，帮助大家便捷的查询出 POI 的知识图谱数据，最终效果如上图所示。它主要有以下功能和交互：

- **POI 信息查询**：星级、评论数、均价、地址、分项评分、推荐理由。
- **知识图谱可视化**：成簇布局、环路布局、节点寻路算法、画布的缩放与拖拽、节点锁定操作、聚焦呈现。
- **辅助信息**：推荐菜、菜品标签、店铺标签词云、情感曲线、细粒度情感分析、相似餐厅。

3.5 可视化叙事的探索

美团大脑是我们团队第一个知识图谱可视化项目，通过该项目的实践，我们积累了一些可视化基础能力和知识图谱可视化的优化策略，让开发效率得到了极大的提升，同时团队开始考虑在交互和表现形式上做进一步的突破。我们也搜集到一些反馈，发现很多人依然对知识图谱这项技术是什么和能做什么了解得不是很清楚。

经过团队的头脑风暴，我们认为核心原因是 AI 技术高深复杂，难以具象化，需要对真实场景进行还原。刚好，知识图谱相对于其他的技术而言其可解释性更强，于是我们决定进行可视化叙事的研发。

数据可视化叙事（Visual Data StoryTelling）是通过隐喻对数据进行可视化，并以可视化为手段，向受众讲述数据背后的故事。下面举个例子，来对比一下纯文字与可视化叙事的不同：

在用户搜索"性价比高适合朋友聚餐的川菜馆"时，根据搜索时间得知是周末的晚上，再通过语义理解分析得到"朋友聚餐"、"性价比高"、"川菜"、"周末聚餐"几个关键信息。美团App用知识图谱技术在环境、价格、口味、菜品、服务几个维度帮助用户搜索匹配的餐厅，通过知识图谱推断出一些餐厅标签，有环境安静、辛辣、海鲜棒、性价比高、菜品精致、上菜快、川菜等，最终基于用户的需求推荐出几家可能合适的餐厅。

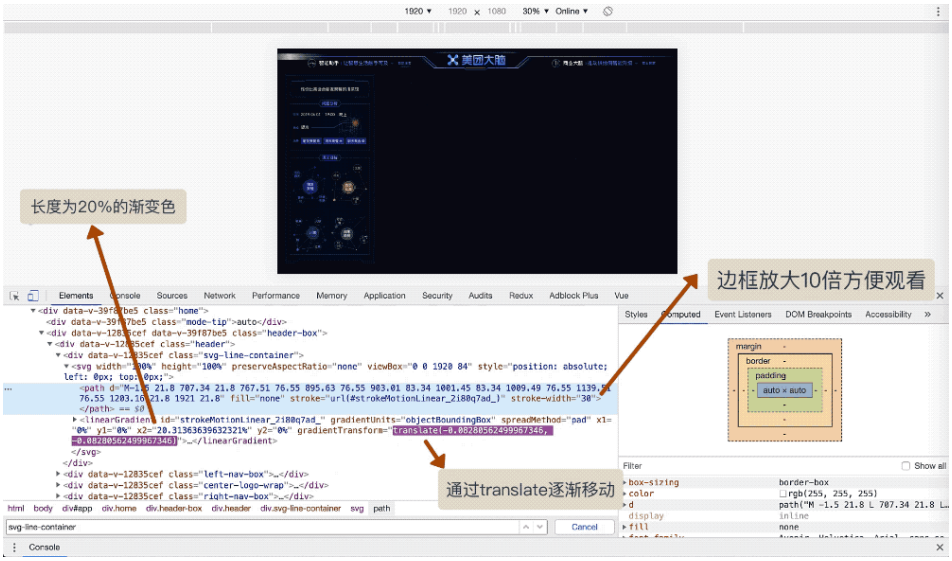
可视化叙事

可以看到，可视化叙事的形式要比文字更直观，能更清晰地让观看者了解数据背后的故事，还可以通过动效将重点信息呈现，引导用户按照顺序了解故事内容。下面我们会介绍几个在可视化叙事中开发动效的思路。

扫光效果

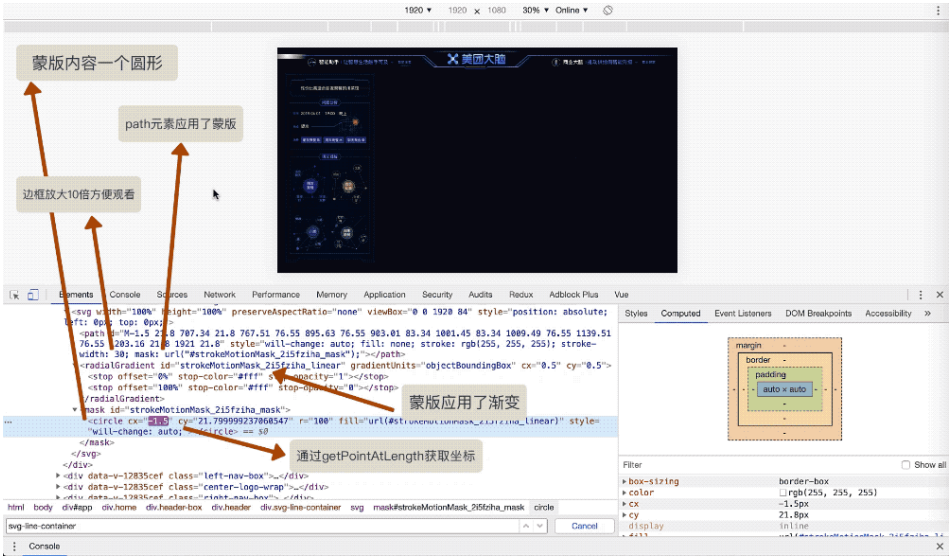
扫光效果对视觉观感的提升和视觉重点的强调非常有效，我们在做扫光效果的轮廓元素上，需要设计师提供两个文件，一个是轮廓的背景图片，一个是带有轮廓 path 的 svg。经过技术调研，我们发现可以通过 svg 渐变或者蒙版来进行实现。

SVG 渐变



扫光 – 渐变

SVG 蒙版



扫光 – 蒙版

渐变方案用在弯曲角度较小的轮廓元素或图谱的边上没有问题，不过渐变只能线性的从一侧到另一侧，如果应用到弯曲角度较大的边上，渐变效果会不连续。

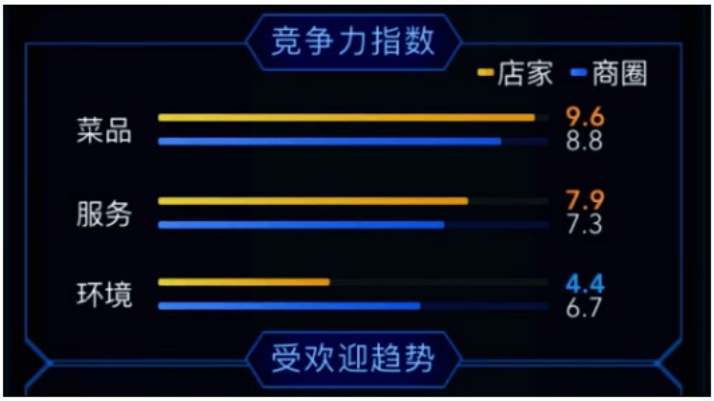
扫光 – 渐变 – 缺点

综合分析一下两种方案，蒙版方案更加灵活，渐变性能更好。由于我们的场景可以规避弧度过大的边，因此我们选择了性能更好的渐变方案。

动效节奏调试

一个动效是否有节奏，对于观看者的体验影响非常大，但是节奏感是一个非常难掌握的东西，这里推荐两个辅助工具：animejs 和贝塞尔调节。

这两个工具能够给大家带来很多灵感，同时可以让设计师自己利用工具调试出或者找到期望的动效，降低动画开发的协作成本，这里展示一个使用贝塞尔函数实现的动效：



贝塞尔

可视化叙事效果

我们为知识图谱的可视化叙事设计了几个典型场景，对日常生活中的找店游玩、商户经营分析等需求进行情景再现，直观地将知识图谱是如何服务真实场景的需求展现出来，以下是可视化叙事的效果：

可视化叙事 – 总体效果

3.6 3D 可视化场景的探索

上面介绍的都是在 2D 场景下知识图谱可视化的开发经验，为了实现更好的视觉效果，我们还探索了 3D 场景的技术方案。我们选择了 vasturiano 的 3d-force-graph，主要原因如下：

- 知识图谱布局库为 d3-force-3d，是基于 d3-force 开发的，延续了团队之前在 D3.js 方向的积累，使用起来也会更熟悉。
- 它是基于 three.js 做 3D 对象的渲染，并在渲染层屏蔽了大量的细节，又暴露出了 three.js 的原始对象，便于对 3D 场景的二次开发。

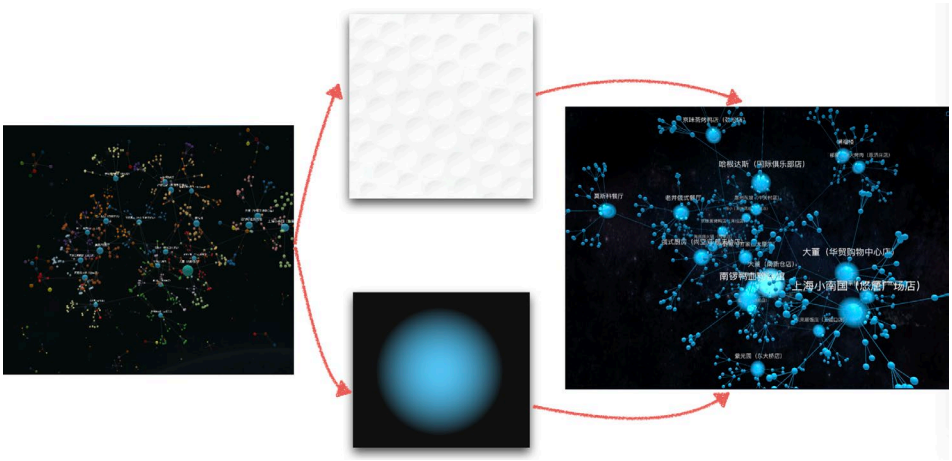
在产品与设计层面，因为我们团队在 3D 可视化上的经验比较少，就学习调研了很多优秀的作品，这里主要从 Earth 2050 项目获取了一些灵感。下面介绍我们在二次开发过程中主要的优化点。

节点样式优化

3d-force-graph 中默认节点就是基础的 SphereGeometry 3D 对象，视觉观感一般，需要更有光泽的节点，可以通过下面的方案实现。

- 用 shader 实现一个透明发光遮罩的材质。
- 用类似高尔夫的纹理让节点更有质感。

操作虽然比较简单，但是将关键节点“点亮”后，整体的视觉观感会好很多。



3D- 节点纹理

3D 动效

为了在 3D 场景下有更好的效果，还需要添加一些动效。

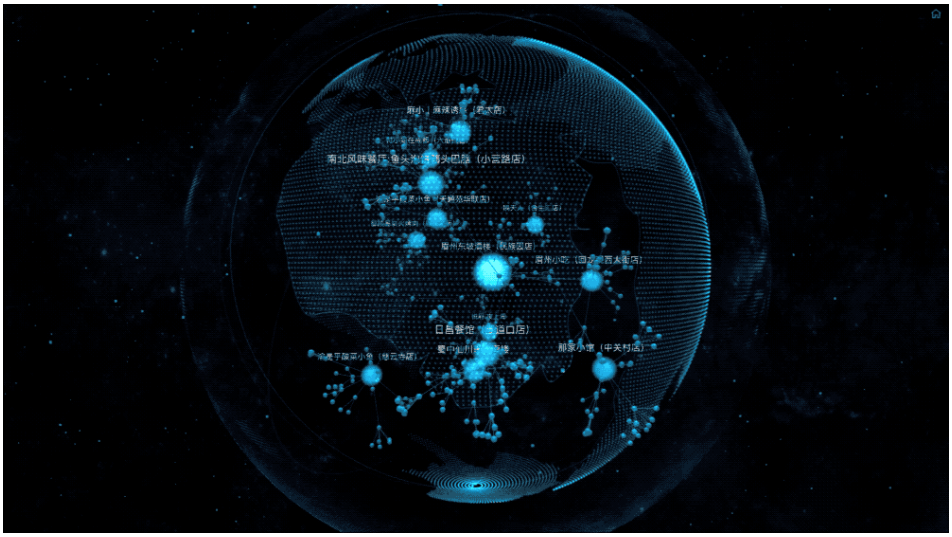
镜头游走

我们利用了内置的相机进行四元数的旋转计算。

3D- 镜头游走

粒子飞散

在飞散的时候，我们创建随机不可见的粒子，控制粒子数量缓慢出现，利用 requestAnimationFrame 向各自方向飞散。



3D- 粒子飞散

产品效果与场景思考

最终在 CES 会场效果如下：

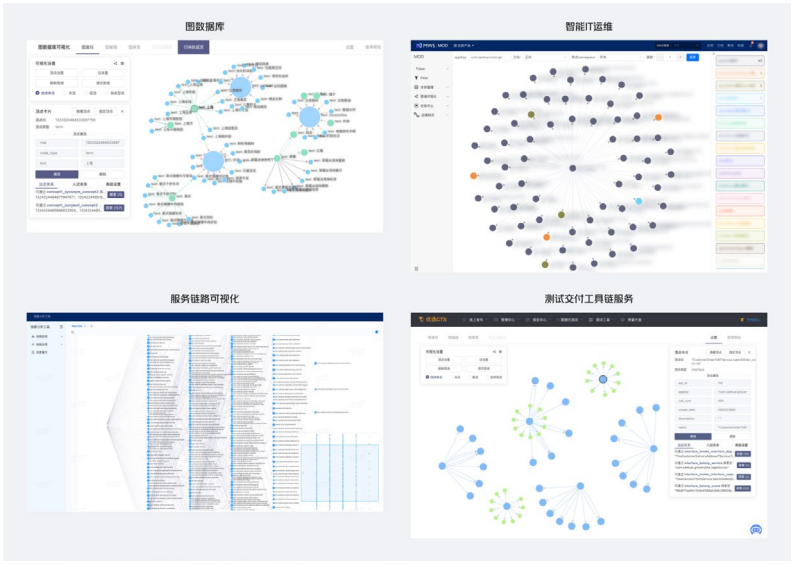
3D-CES 现场

我们在研发了主要应用在技术推广的 3D 知识图谱可视化后，还考虑迁移到工具类应用中，但是发现工具类应用目前更适合 2D 的展示与交互，3D 虽然对于空间利用率更大，但是用户交互方式也更复杂。如果后续能思考出更高效的交互方式，我们会再次尝试利用 3D 知识图谱可视化来提升工具类应用的产品体验。

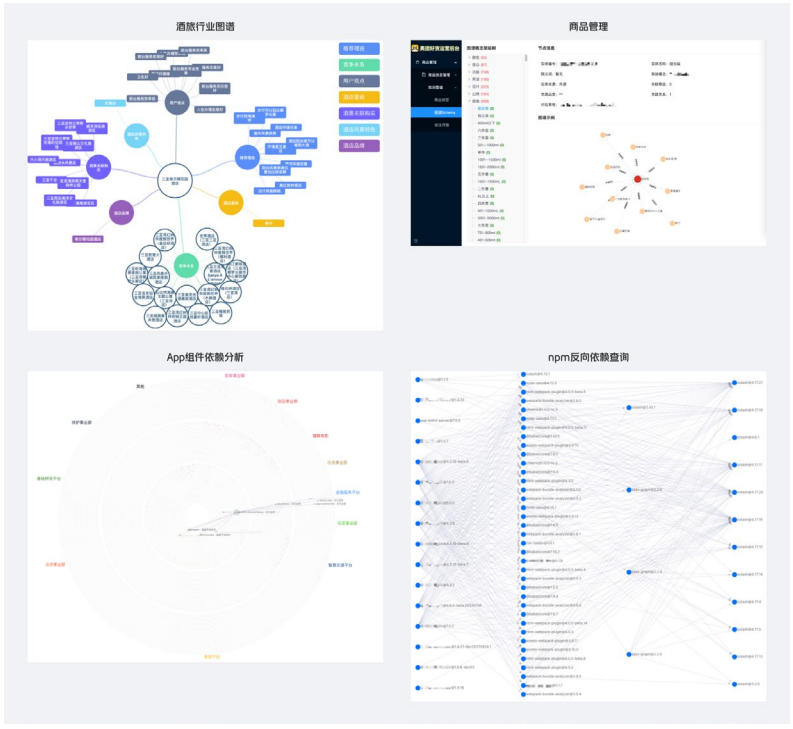
4. 落地场景

目前，知识图谱可视化技术方案已经应用在了美团多个业务场景中，包括美团大脑、图数据库、智能 IT 运维、组件依赖分析、商品标签管理、行业领域图谱等。未来，我们还将探索更多的应用场景。

落地场景 –1


图数据库
智能IT运维
服务链路可视化
测试交付工具链服务

落地场景 –2


酒旅行业图谱
商品管理
App组件依赖分析
npm反向依赖查询

落地场景 –3

5. 未来展望

最后，展望一下知识图谱可视化技术后面还可以探索的一些方向，我们从交互场景、效果呈现以及工具能力等三个维度进行说明。

交互场景

3D 场景中的交互：在 3D 场景中做知识图谱可视化视觉震撼程度更强，但是现阶段实用程度还不够，交互的效率也不如 2D 场景，高效的 3D 交互方式需要进一步探索。

虚拟现实：元宇宙的概念快速带动了 VR 等虚拟现实技术的发展，技术成熟后或许能带来更好的可视化体验。

效果呈现

大规模知识图谱可视化：在节点数量非常多的知识图谱可视化中，性能上的问题有 WebGL、WebGPU 等技术方案去解决，但是也仅限于能可视化出来，用户已经很难找到自己需要的信息了，如何既能呈现出成千上万的节点，又能让用户便捷的找到自己需要的关系数据信息很重要。

布局的智能化：目前知识图谱的布局合理性主要靠半人工干预的方式来保证，后面可以考虑针对不同的数据特征去自动匹配合适的力学布局策略，用算法智能预测出最合理的布局方式，减少开发者或用户的干预成本。

工具能力

通用查询工具：目前各大知识图谱数据存储引擎都有自己的可视化查询工具，互不通用，也互有优缺点，如果有统一的可视化查询语言，就能够让一种可视化工具适配多个存储引擎和应用，提高工具应用的效率。

本文作者

魏耀、诚威、宋奇、敏芳、曾亮，均为美团平台 / 搜索与 NLP 部前端工程师。

参考资料

- https://d3js.org/
- https://github.com/vasturiano/d3-force-3d
- https://github.com/vasturiano/3d-force-graph
- https://2050.earth/
- https://en.wikipedia.org/wiki/Quadtree
- https://github.com/getify/CAF
- https://github.com/tj/co
- https://animejs.com/documentation/#staggeringBasics
- https://cubic-bezier.com/

招聘信息

美团 / 搜索与 NLP 部 / 平台前端团队是一个创新、开放、对技术有热情的前端的团队，团队主要负责搜索平台、NLP 平台、知识图谱可视化、跨端框架、低代码工具等方向，长期诚聘实习、校招、社招，坐标北京 / 上海，欢迎感兴趣的同学发送简历至：zhangweiyao@meituan.com，也欢迎同行进行技术交流。

终端新玩法：技术栈无关的剧本式引导

作者：松涛　尚先　筱斌

背景

互联网行业节奏偏快，App 的更新愈发频繁，如何让用户跟上更新节奏，理解产品功能，完成认知迭代，是业务发展中不可忽视的一环。同时“低代码 / 零代码”的理念也逐步被大众认可，相关调研报告指出“低代码 / 零代码”可以加速企业的数字化转型。以美团到家事业群为例，在宅经济再度升温后，即时配送应用的增长速度高于其他配送时长的应用。大量新用户的涌入既是机遇，也是挑战。目前美团到家事业群已经涵盖了医药、团餐、闪购、跑腿、团好货、无人配送等 10+ 业务线。新的商业模式意味着新领域的尝试，主业务外卖平均数日也会上线新的功能模块，这些都需要关注用户心智建设与效率提升。

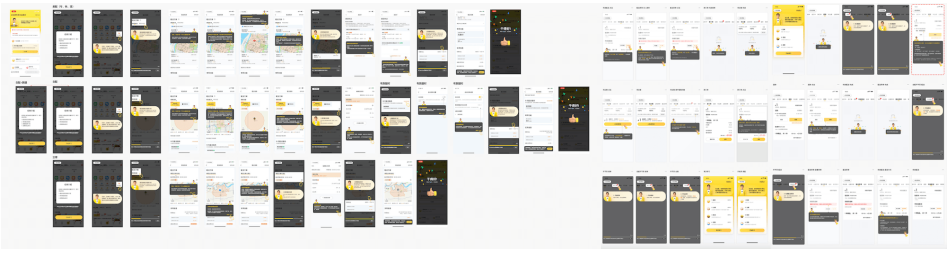
现状

在提升用户心智，获得服务认同方面，业界内也做了很多尝试，包括丰富多样的轻交互，也有“保姆式”的游戏引导教学。这些实现方式归结到技术层面，都是 App 中的功能引导，它可以让用户在短时间内快速了解产品特色以及产品使用方式。相对于“广告投放”、“口号传播”、“地推介绍”等传统方案，App 中的功能引导，具备成本低、覆盖准、可复用等特点。

常见的功能引导

App 功能引导是用户心智建设的“敲门砖”，只有让用户熟悉平台操作、了解产品特色作为前提，才能进一步借助情感化、场景识别、运营技巧等手段来做用户心智建设。随着 App 功能的不断迭代，在用户中逐渐出现了“用不明白”的现象，这个现象在美团外卖商家客户端尤为突出。作为商家生产运营的主要工具，客户端承载的业务功能复杂多样，设置项更是品类繁杂，如果商家用不明白，就会对整个运营体系造成非常不利的影响。

为了让商户“用得明白”，2021 年第一季度，美团外卖商家端在功能引导类需求层面耗费了大量人力，平台产品侧重点对商家进行了扶持，并试点了“情感化引导”等项目，虽然业务效果取得了正向收益，但由于后续的研发估时较大，空有想法却难以落地。类似的营销、广告、商品、订单等业务也由于快速迭代，也需要配套生产一系列产品功能的引导需求，也因为人力问题而一直处于积压状态。



部分引导类需求

目标与挑战

基于上述背景与现状，我们迫切需要提供一种解决方案，让业务方可以更快捷地落地自己的想法，在控制好成本的情况下，更好地建设用户心智。同时，解决目前积压的业务任务，包括但不限操作教学、功能介绍、情感化、严肃化等等场景。于是 ASG（Application Scripted Guidance）剧本式引导项目就应运而生了。

项目目标

我们的项目目标是搭建一套好用的剧本式引导工具，即便是非技术同学也能独立完成生产与投放，并且相比传统方案的成本更低、效果更好，目前主要应用在“操作引导”与“心智建设”等场景。

这里的“剧本”怎么理解？就是带入一个实际场景，模拟一个期望达成的目标，带领用户为此目标而进行一系列的操作引。，用户可感受整体流程以及其中的关联与时序关系。也可以理解为，这是一个预先安排好的小节目，一步一步展示给用户，可能需要交互也可能不需要。

而剧本化的引导方式，之前在游戏类 App 应用比较常见，比如遇到了一个火属性敌人，所以要去武器界面，选中某武器，换上水属性宝石。近两年，剧本化的引导逐步在展示类 App 与工具类 App 中也开始被使用起来。

此前，美团外卖商家端的“开门营业”、“模拟接单”等引导需求就使用了类似的思想，这种方式更加先进，但开发成本较高，所以导致后续引导类需求的积压。

收益测算逻辑

ASG 剧本式引导项目的收益测算逻辑是“降本增效”，这里的“效”既指“效率”也指“效果”，结果数据测算公式为：提效倍数 x =（ 1 /（ 1 − 成本缩减比 ））*（ 1 + 产品指标增长比），因此目标可拆解为如下两个方向：

- **更低的生产成本**，借助一些端能力和配置能力，通过简易的交互，就可以让产品与运营同学独立上线剧本。“零代码”与“技术栈无关”作为项目的核心竞争

力。我们提供标准化的框架，并通过一些参数与类型的调配来应对不同的需求场景，在大框架中提供有限的定制能力。

- **更高的应用效果**，相比于传统的功能引导，剧本式引导可以更加生动，能够融合更多元素（不僵硬的语音、恰逢时机的动效、和蔼的 IP 形象），从而带来沉浸式的体验，增强用户感知。更加关注与用户的交互 / 互动，操作后的反馈最好是真实页面的变化，加深用户的理解。时机更加可控，在满足规则后自动触发，后台可筛选特定特征的用户（比如用不明白的用户）定向下发剧本引导。

面临的挑战

1. 目前，Flutter/React Native/ 小程序 /PWA 等终端技术栈各有各的适用场景，App 大多数为几种技术栈的组合，如何抹平差异，做到技术栈无关？（即容器无关性 Containerless）。
2. 剧本执行的成功率与健壮性如何保证？（MVP 版 Demo 的成功率仅达到 50%，稳定版目标要达到 99% 以上）。
3. 怎样落实“零代码”的剧本生产方案，以支持产运独立发布？（之前类似单任务需要研发 20 ~ 50 人日）。

整体设计

展示形式选择

项目主体应该选择基于什么样的形式？我们的思路是先确定“好的效果”，再去尝试在此形式下做到“更低的成本”。

“好的效果”自然是期望体现在产品指标上，但是前期，在数据对比上不同的场景落地指标跨度较大，对于不同的形式也难以拉齐标准横向比较。所以我们从“学的越多才能会的越多”的角度推演，通过平台传递的信息能否更多的被用户接受，来衡量最终产品效果。

视频时长	新店必看（141 s）		新店配置活动（118 s）		勤看数据（90 s）		新店顾客分析（100 s）		维护评价（95 s）	
	播放次数	次均播放时长	播放次数	次均播放时长	播放次数	次均播放时长	播放次数	次均播放时长	播放次数	次均播放时长
日均	557.66	70.29	40.71	65.70	232.43	59.01	168.28	66.09	124.85	63.16

我们选取了一些之前含视频教学的业务数据，平均播放时长比例在 50% ~ 66% 左右，大多数用户没有看完整个视频。我们分析后认为，因为用户理解的速度有慢有快，稍长的视频内容如果吸引力不够大，或不能贴合用户理解的节奏，就很难被看完。同时，视频传播是单向的，缺乏互动，且不是剧本式思路。于是我们与产品商议后，在一些引导需求上试点了基于真实页面开发、带有一定剧本、可交互的引导（左上角设有常驻按钮，用户可以随时退出引导）。

剧本示例	新店在线联系（共 17 步）			新店配置商品（共 14 步）			新店换购活动创建 （共 10 步）		
	触发次数	次均完成步骤数	用户接受比	触发次数	次均完成步骤数	用户接受比	触发次数	次均完成步骤数	用户接受比
日均	616	14.26	83.9%	168	10.73	76.6%	57	8.11	81.1%

试点的结果符合我们的预期。基于真实页面开发且可交互的引导，的确可以更好地被用户所接受。引导完成步数比例达到 76% ~ 83%，相比于平均播放时长比例明显更高。

其实，常规的展示形式上还包括图片组，这个基本是强制用户点完才能进入该功能，可以应用于一些建议的引导场景，但对于一些中等复杂度及以上的引导案例，这里的数据就不具备参考意义了。 我们基于一些采集到的数据和基本认知，对以上三类做了一个对比，表格如下：

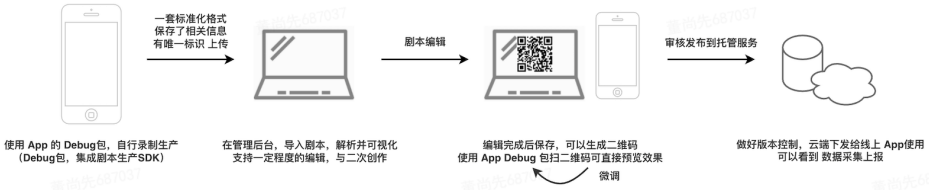
展示形式	研发成本	视觉与产运成本	互动	节奏可控	下发资源大小	用户体验	用户学习到的比例	维护成本
图片组	低	中	无	是	平均 1.2 MB	低	–	低
产运制作视频并配音	低	高	无	否	平均 8.6 MB	中	50% ~ 65%　avg 59%	中
基于真实页面开发&展示	高	中	支持	是	0.1 MB 以内	高	76% ~ 83%　avg 80%	较高

我们得到结论是：如果想要拿到更好的效果，想以用户为中心设计一些更能被用户所接受的引导，基于真实页面研发有着明显的优势，但是这么做的缺点是开发成本较

高。目前，简易的试点已经获得了不错的提升效果，所以产研同学有信心在引入更多客户端能力与调优后，整体效果还有更大的提升空间。

方案描述

ASG 剧本式引导项目的目标受众是产品运营同学，我们尝试从他们的角度思考了：**怎样才算是一个便捷且高效的“剧本式引导生产与投放工具”？**

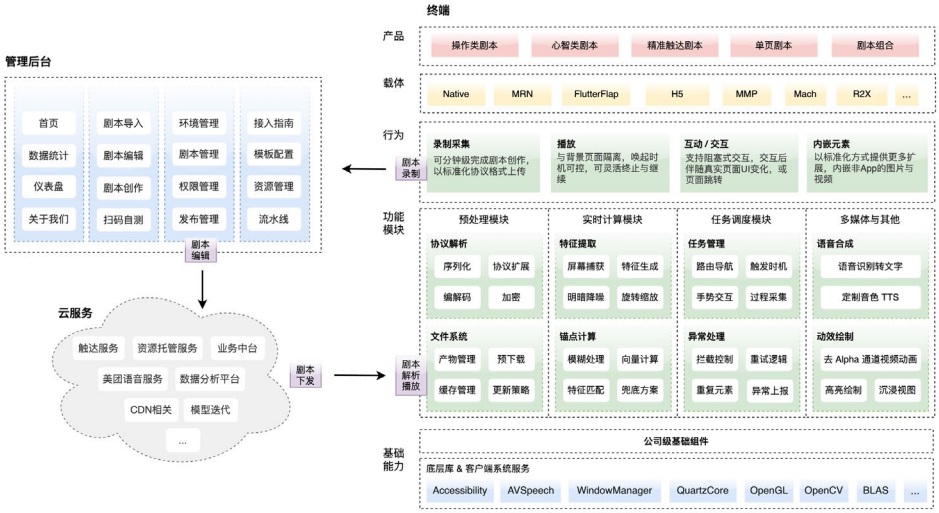


产品运营视角

如上图所示，我们提供给产品运营同学的交互仅有：录制、编辑、预览、发布等四个步骤，当产品运营同学需要在业务模块上线引导时，只需拟定一个剧本，然后四步即可完成这个“需求”，整个流程几乎不需要研发和设计同学的参与。

在具体的执行方案中，我们对剧本引导进行了模板化的设计编排，将每个引导动作抽象成一个事件，多个事件组合形成一个剧本。同时为保证不同终端的兼容性，我们设计了一套标准且易扩展的协议描述剧本元素，运行时 PC 管理后台和 App 可自动将剧本解析成可执行的事件（如坐标点击、页面导航、语音播放等）。

核心的功能模块在剧本的执行侧。为了保证更高的应用效果，我们要求引导过程与用户的交互，均操作在真实的业务页面，播放展示的元素也要求是实时计算与绘制的，这对系统性能与准确性提出了更高的要求。系统的全景图如下图所示，由终端侧、管理后台与云服务三个部分组成：



系统全景图

终端侧：包括两个职能，既具备剧本的录制能力，也具备剧本的播放能力，由四个功能模块构成。预处理模块负责剧本的资源下载、协议解析、编解码等操作，是保障剧本成功执行的前置环节；实时计算模块则通过屏幕捕获、特征匹配、图像智能，完成动态获取剧本锚点元素的信息，保证了剧本引导的精准展示，是实现剧本引导技术栈无关的核心环节；任务调度模块主要通过事件队列的实现方式，保证剧本有序、正确的执行；多媒体模块负责语音合成和动效绘制，在特定业务场景为剧本播放提供沉浸式的体验。同时 PC 端在客户端的基础上进行了能力的扩展，对于常见的 React/Vue/Svelte 网页应用都可以低成本地接入和使用。

管理后台：包括剧本编辑、导入和发布、权限控制、数据看板等功能模块。其中剧本编辑模块，承载了剧本协议的解析、编辑、预览等关键功能，操作界面按功能划分为以下区域：

- 事件流控制区域：以页面帧的形式展示剧本流程中的事件，提供动态添加与删除、调整页面帧顺序等编辑功能。
- 协议配置区域：依照剧本的标准协议，通过可视化页面帧配置项，生成满足需要的引导事件；同时提供丰富的物料，满足心智类剧本的情感化创作。

剧本预览区域：支持通过二维码扫描，实现便捷、无差别地效果预览，保证与最终呈现给用户的引导效果一致。

管理后台

云服务：依赖美团的底层云服务平台，在剧本编辑完成后，需要资源托管服务、CDN等进行资源的管理及分发，完成剧本的下发及更新。业务中台在端侧 SDK 和后台策略配置的共同作用下，提供了更细粒度的下发配置，更丰富的触达时机，满足业务侧按时间、城市、账号与门店、业务标签等维度配置的诉求。

部分技术方案剖析

基于视觉智能的区域定位方案

在引导过程中，需要对关键路径上目标区域设置高亮效果。在技术栈无关的前提下，基本思路是线下截取目标区域，线上运行时全屏截图，通过图像匹配算法，查找目标区域在全屏截图中的位置，从而获得该区域坐标，如下图所示：

高亮识别效果

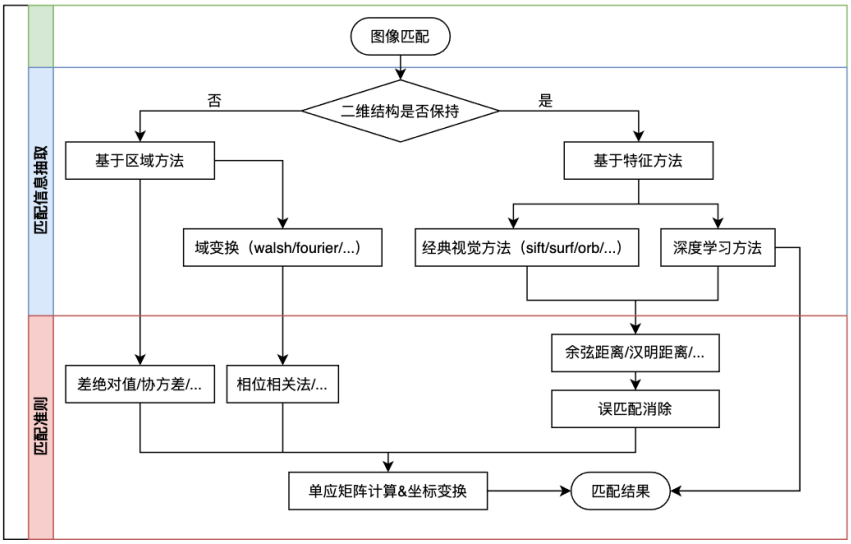
整体思路看起简单，但在具体的实践中却面临着诸多的挑战：

1. 圆角类图标的 UI 元素（`RadioButton`、`Switch`）在边缘区域能检测到的特征点过少，导致匹配成功率低。
2. 小字体的区域，在低分辨率情况下无法检测到足够的特征点，放大分辨率可以提升匹配精度，但是耗时也会成倍增加。
3. 在不提供初始位置的条件下，只能做全图检测和暴力匹配，需要检测和存储的特征点数量太庞大，尤其是复杂画面和高分辨率图像，移动设备上性能和内存开销无法接受。
4. 终端手机设备屏幕分辨率目前有几十种，算法需要适配多种分辨率。
5. 端侧部署，对算法库的包大小、性能、内存占用都有要求，例如 OpenCV，即使经过精心的裁剪之后仍然有 10 ~ 15 MB，无法直接集成到线上 App 中。

经过理论研究与实践试点，最终我们采用的是传统 CV（Computer Vision）+ AI 的解决方案，大部分场景可以基于传统 CV 的角点特征检测和匹配得到结果，未命中的则继续通过深度学习网络的检测和跟踪来获取结果。在工程部署方面也做了相应的优化。接下来将详细介绍这个方案的实现。

图像匹配流程概要

图像匹配算法由信息提取、匹配准则两部分组成。根据信息载体的二维结构特征是否保留，匹配算法可分为基于区域的信息匹配与基于特征的信息匹配，如下图所示：



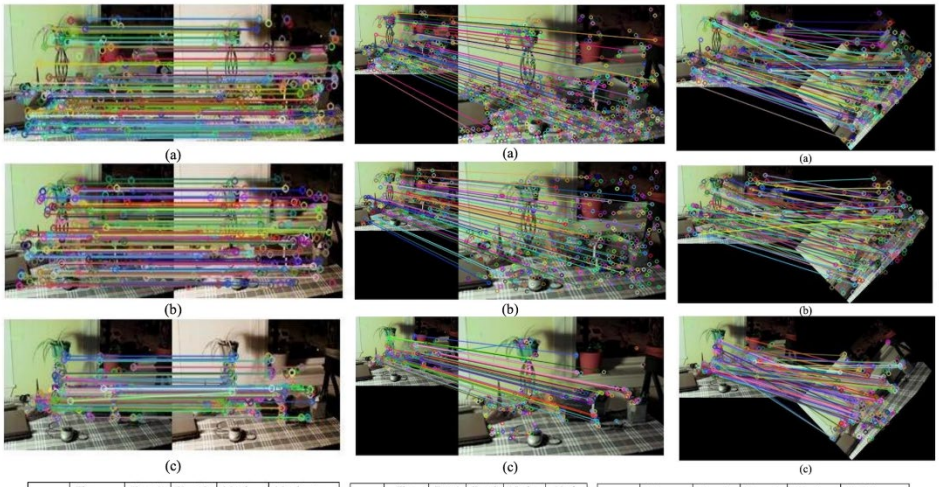
图像匹配流程概要

基于区域的图像匹配方法，采用原始图片或域变化后的图片作为载体，选取最小信息差异区域作为匹配结果，该方法对于图像形变、噪声敏感等处理不佳。而基于特征的图像匹配方法，丢弃了图像二维结构信息，提取图片的纹理、形状、颜色等特征及位置信息描述，进而得到匹配结果。基于特征的算法鲁棒性好、信息匹配步骤速度快、适应性强，应用也更加广泛。

基于传统 CV 特征的图像匹配

其实该项目的应用场景，属于典型的 ROI（Region Of Interesting）区域检测、定位，传统 CV 算法针对不同的使用场景已经有很多比较成熟的算法，比如轮廓特征、连通区域、基于颜色特征、角点检测等。角点特征是基于中心像素与周围像素亮度差异变化剧烈，且基本不受旋转、缩放、明暗等变化影响的特征点，经典的角点检测有

SIFT、SURF、ORB 等，相关研究业界已经有很多，E Karami [5] 等人在 2017 年发表的一份对比研究结果（如下图所示）表明：绝大多数情况下，ORB 最快，SIFT 匹配结果最好，ORB 特征点分布集中在图像中心区域，而 SIFT、SURF、FAST 则分布在整张图上。在美团到家的场景下，目标区域可能位于图片的中心、四角等任何位置，所以 ORB 对于边缘区域的目标区域匹配失败的概率会偏大，需要特殊处理。



	Time (sec)	Kpnts1	Kpnts2	Matches	Match rate (%)
SIFT	0.13	248	229	183	76.7
SURF	0.04	162	166	119	72.6
ORB	0.03	261	267	168	63.6

	Time (sec)	Kpnts1	Kpnts2	Matches	Match rate (%)
SIFT	0.25	248	1210	232	31.8
SURF	0.08	162	581	136	36.6
ORB	0.02	261	471	181	49.5

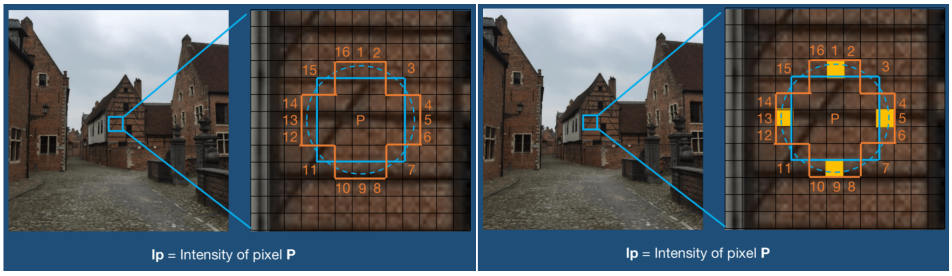
	Time (sec)	Kpnts1	Kpnts2	Matches	Match rate (%)
SIFT	0.16	248	260	166	65.4
SURF	0.03	162	271	110	50.8
ORB	0.03	261	423	158	46.2

(a) SIFT (b) SURF (c) ORB 的匹配结果：不同强度（左）缩放（中）旋转（右）

总体来讲，一个效果好的特征检测匹配算法，需要同时具备：尺度不变性、旋转不变性、亮度不变性，这样才能适应更多的应用场景，具有较好的鲁棒性。下面我们以 ORB 为例，来简单阐述一下算法的计算过程（感兴趣可以查阅更多的相关资料）。

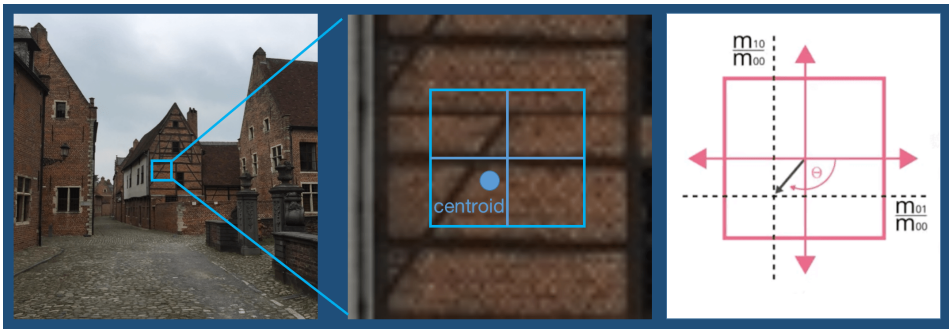
ORB = Oriented FAST + Rotated BRIEF（下文用 OFAST 与 rBRIEF 代替），ORB 融合了 FAST 特征检测和 BRIEF 特征描述算法，并做了一些改进，即采用改进的 OFAST 特征检测算法，使其具有方向性，并采用具有旋转不变性的 rBRIEF 特征描述子。FAST 和 BRIEF 都是非常快速的特征计算方法，因此 ORB 获得了比较明显的性能提升。

要想判断一个像素点 p 是不是 FAST 特征点，只需要判断其周围 7x7 邻域内的 16 个像素点中是否有连续 N 个点的灰度值与 p 的差的绝对值超出阈值。此外，FAST 之所以快，是因为首先根据上、下、左、右 4 个点的结果做判断，如果不满足角点条件则直接剔除，如果满足再计算其余 12 个点，由于图像中绝大多数像素点都不是特征点，所以这样做的结果，用深度学习”炼丹师“的话来说，就是“基本不掉点”，且计算时间大大减少。对于相邻的特征点存在重复的问题，可以采用极大值抑制来去除。



邻域 16 个点的位置（左）；上、下、左、右 4 个点（右）

改进后的 OFAST 会针对每个特征点计算一个方向向量。研究表明，通过从亮度中心至几何中心连接的向量作为特征点的方向，会比直方图算法和 MAX 算法有更好的效果。



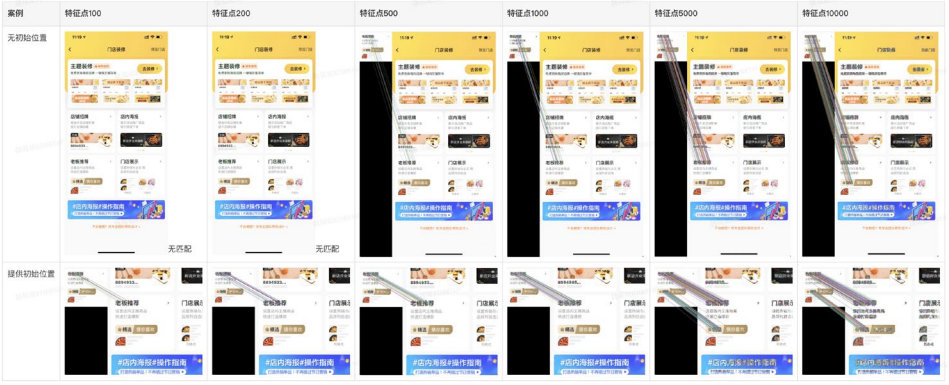
OFAST 方向向量的计算

ORB 算法的第二步是计算特征描述符。这一步采用的是 rBRIEF 算法，每个特征描述符是仅包含 1 和 0 的长度为 128 – 512 位的向量。得到特征点和特征描述符之后，

就可以做特征匹配了。此外，特征匹配算法也比较多，为了简化计算，我们这里采用了 LPM[6] 算法。得到筛选后的特征对后，计算它们的外接矩形包围框，反变换到原图坐标系就可以得到目标的区域位置坐标。

基于纯传统 CV 算法测试的结果表明，特征点数量对匹配的召回率有直接影响，特征点较少，召回率偏低无法满足业务需求；特征点超过 10000 点则会严重影响算法性能，尤其是在移动端设备上的性能，高端机型上耗时在 1 秒以上。我们针对目标区域小图和原图设定不同特征点数量，然后做匹配，这样可以兼顾性能和匹配精度。

不同配置参数实测的特征点和匹配结果如下图所示，针对大多数图像、文字内容的区域，特征点在 5000 以上，匹配结果不错，但还存在常见区域匹配失败的情况；特征点在 10000 以上，除了一些特殊 Case，大多数场景匹配结果都比较满意。如果不提供目标区域的大概的初始位置（真实情况），基本上大多区域需要 10000 ~ 20000 特征点才能匹配，端侧性能就是个问题。

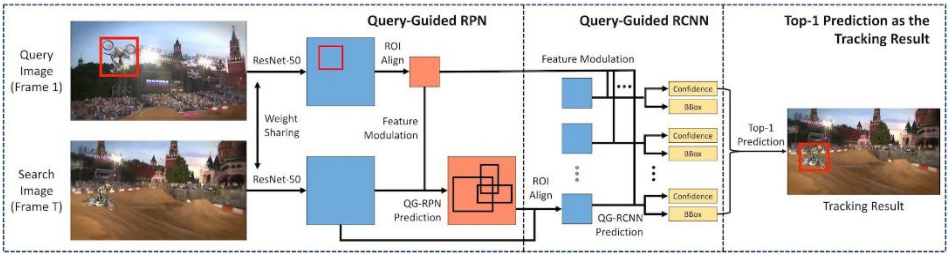


实测结果：匹配的召回率与特征点数量直接相关

基于深度学习的图像匹配

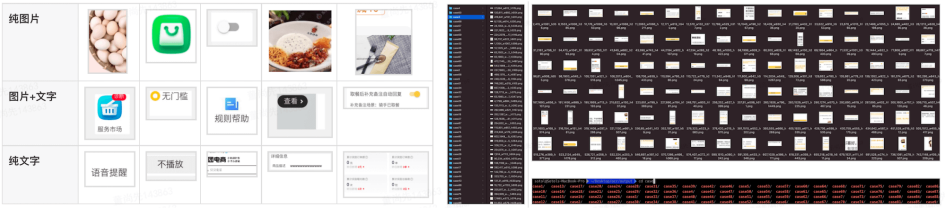
基于传统 CV 存在的弊端和一些无法解决的问题，我们需要具有更强图像特征表达能力的算法来进行图像匹配。近些年，深度学习算法取得了巨大的突破，同样在图像的特征匹配领域中也取得了较大的成功。在本应用场景中，我们需要算法在全屏截图中快速定位一个子区域的具体位置，即需要一个模型通过一个区域中局部区域的特征快

速定位其在全局特征中对应的位置。该问题看似可以使用目标检测的相关算法进行求解，但是一般目标检测算法需要目标的类别 / 语义信息，而我们这里需要匹配的是目标区域的表观特征。针对该问题，我们采用了基于目标检测的图像跟踪算法，即将目标区域视为算法需要跟踪的目标，在全屏截图中找到我们要跟踪的目标。在具体实现过程中，我们使用类似于 GlobalTrack[7] 的算法，首先会提取目标区域对应的特征，并使用目标区域的特征来对全屏截图的特征进行调制，并根据调制之后的特征来对目标区域进行定位。并根据移动端计算量受限的特性，我们在 GlobalTrack 的基础上设计了一个单阶段的目标检测器来对该过程进行加速。



GlobalTrack 示意图

由于我们直接使用目标区域的特征来引导目标检测的过程，所以其能够处理更为复杂的目标区域，比如纯文本、纯图像或图标、文字图像混编等，凡是能在 UI 上出现的元素都可能是目标区域，如下图所示的一些示例。



目标区域示例与包含不同尺寸类别组合的训练数据

结合业务场景，要求针对移动设备上 App UI 画面的任何局部区域做到精确定位。如上述的分析，该问题既可以看作是一个目标检测和匹配的问题，又可以看作是一个目标跟踪问题。同时算法需要能够适配不同内容的 ROI 区域、不同的屏幕分辨率、不

同的移动设备。

我们选择的方案

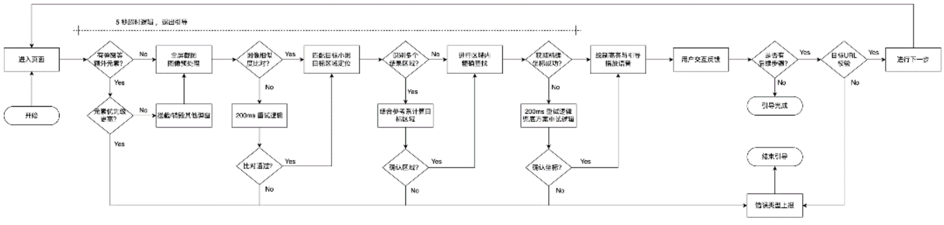
前文提到，我们采用的是 CV + AI 解决方案，这样的优点是：一方面解决传统 CV 检测无法覆盖全场景的问题；另一方面优化性能，减少移动端设备的耗时。

在工程部署方面，我们采用纯 C 实现检测和匹配算法，并且对 ORB 算法做了一些定制化修改。此外，我们采用多线程、Neon 优化等手段提升性能，从 800 毫秒优化到 100 毫秒左右。最终的版本不依赖 OpenCV 及第三方库，大大减小了算法库的包大小。深度学习模型基于 MTNN 端侧推理引擎获得了最优的推理性能和精度。在中高端机型上，可以启用异构硬件并行加速，CV 与 AI 并行计算，在 CPU 上执行特征检测的计算，同时在 GPU 或 NPU 上执行模型推理，然后做融合，这样可以在不增加 CPU 负载的情况下提升性能和准确率。

保证任务执行的健壮性

任务执行感知

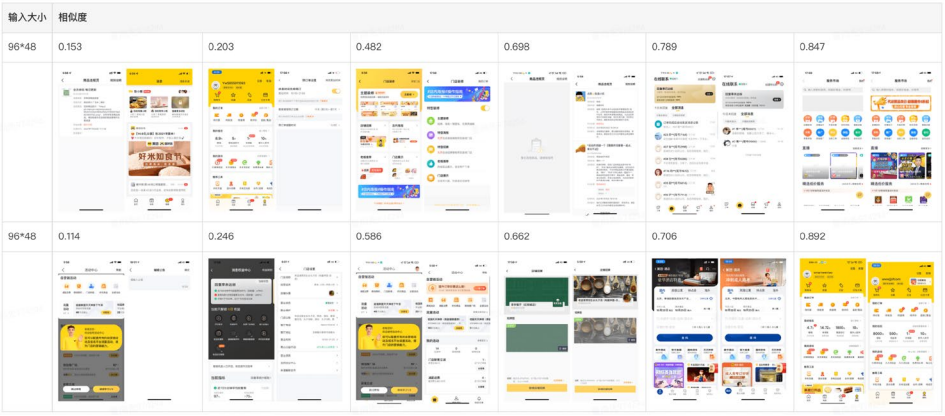
传统方案在做开发引导时，我们可以通过函数回调、广播、组件变化等多种方式获取任务执行状态。但在技术栈无关前提下，感知引导过程失败、感知用户执行 / 点击是否正确，相比之下就比较困难。同时还要精确甄别出错误类型，增加特定步骤的重试方案，尽可能保证剧本的执行通畅，在极少数遇到阻塞是错误时，需要及时确认，上报错误并退出引导，减少对用户的影响。



任务执行流程图

首先，比较优雅的“黑盒”方案是使用图像相似度对比技术，此能力模型在视觉智能中比较基础，在通过跳转来到目标页面后，会截图与目标特征进行比较，进行快速容错。根据线下的大量测试数据，去除一些极端情况，我们发现在不同的阈值下是有规律的：

- 相似度 80% 以上的区间，基本可以确定目标页面准确，受一些角标或图片区块加载的影响没达到更高。
- 相似度 60% ~ 80% 的区间，是在一些列表样式或背景图、Banner 图有些许差异导致的，可以模糊判定命中（上报数据但不用上报异常）。
- 相似度 40% ~ 60% 的区间，大概率遇到了对应模块的 UI 界面改版，或者有局部弹窗，这时就需要进行一些重试策略适时上报异常。
- 相似度 40% 以下，基本确定跳转的是错误页面，可以直接终止引导流程，并上报异常。



图片相似度部分 Case 实测效果

同时，我们在端侧也有一些判定规则来辅助图像对比的决策，比如**容器路由 URL 比对**，当图像对比不匹配但容器路由 URL 准确时，会有一些策略调整并进行重试逻辑。在确认页面准确后，才会进行高亮区域寻找以及后续的绘制逻辑。最后兜底可以通过**超时失败**的方式自然验证，一个剧本关键帧的完整判定流程，我们设置了 5 秒的超时策略。

关于尺度与旋转不变性

为了在尺度上具有更好的健壮性，计算过程首先会对图像做高斯模糊，去除噪声的影响，并且对图像做下采样生成多层图像金字塔，对每一层都做特征检测，所有特征点集合作为检测到的特征点结果输出，参与后续特征匹配计算。为了应对图像旋转的情况，可以加入 rBRIEF，rBRIEF 从给定特征点的 31 x 31 邻域内（称为一个 Patch）随机选择一个像素对。下图展示了采用高斯分布采样随机点对的方法，蓝色正方形像素，是从以关键点为中心的高斯分布中抽取的一个像素，标准偏差 σ；黄色正方形的像素，是随机对中的第二个像素，它是从以蓝色正方形像素为中心的高斯分布中抽取的像素，标准偏差为 σ/2，经验表明，这种高斯选择提高了特征匹配率。当然也有其它选择方式，我们这里就不一一列举了。首先，根据特征点方向向量构造旋转矩阵，并对 N 个点对做旋转变换，使得每个点对与该特征点的主方向一致，然后再根据点对来计算特征向量。因为特征向量的主方向与特征点一致，意味着 rBRIEF 可以在朝着任何角度旋转的图像中检测到相同的特征点。

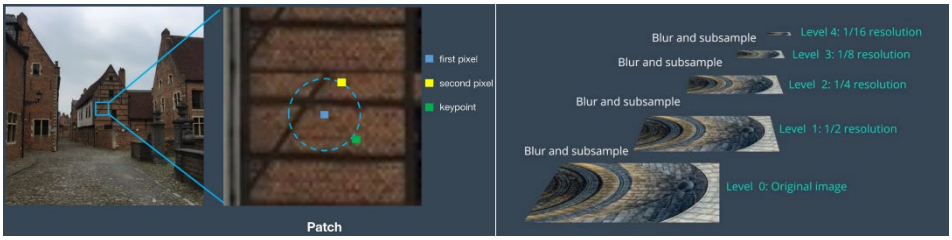


图 rBRIEF 随机像素对的选择（左）；图像金字塔（右）

其他容错处理

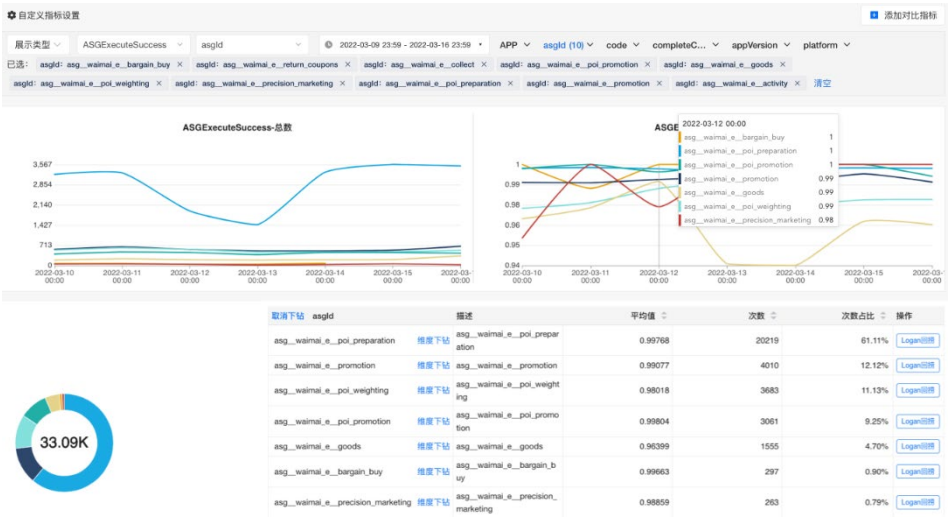
对于页面中存在多个相同或类似元素的场景，不能草率地选择任意一个区域。因此在进行目标区域定位时，我们需要在检索目标区域的基础上，结合目标周围信息，提供一个参考区域。运行时，提供目标区域图像信息及参考区域图像信息，查询到多个目标结果后，再查询参考区域所在位置，通过计算，离参考区域最近的目标区域则为最终目标区域。

对于页面中出现的不同技术栈弹窗场景，由于出现时机也不确定，一旦出现，容易

对目标区域造成遮挡，影响整个引导流程，需要对各类弹窗进行过滤和拦截。针对 Native 技术栈，我们通过对统一弹窗组件进行拦截，判断执行过程中禁止弹窗弹出，引导过程中业务认为非常重要的弹窗则通过加白处理。在 Flutter 上则采用全局拦截 `NavigatorObserver` 中的 `didPush` 过程，拦截及过滤 Flutter 的各类 `Widget`、`Dialog` 及 `Alert` 弹窗。关于 Web 上的处理，由于 Web 弹窗业务方比较多，没有特别统一的弹窗规范，特征比较难取；目前是在 Web 容器中注入一段 JavaScript 代码，给部分有弹窗特征和指定类型的组件设置隐藏，考虑到拓展性，JavaScript 代码设置成可动态更新。

对于部分页面元素复杂导致加载时间稍长的场景，剧本播放时也会基于录制侧提供的 delayInfo 字段，进行一些延迟判定策略。

基于前面的努力，剧本的执行链路成功率（如下图所示）基本可以达到 98% 以上，部分成功率较低的剧本可以根据维度下钻，查询具体的异常原因。



部分链路指标监控

零代码完成剧本创作与编辑

把一个剧本的生命周期划分为“生产”和“消费”两个阶段，“生产”阶段对应的是

剧本录制完成并上传至管理后台进行编辑的过程，“消费”阶段则对应下发与播放。如果说前两个挑战主要聚焦在“消费”，那么这里的挑战则主要聚焦在“生产”方面。接下来，我们将从“录制端侧赋能”与“标准协议设计”两个方面进行详细的介绍。

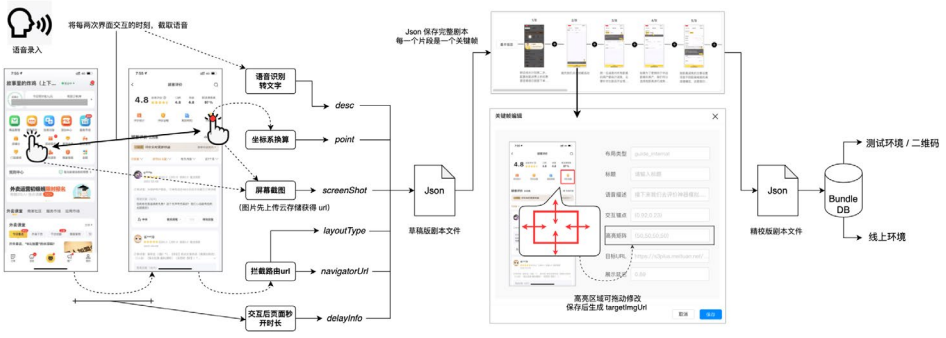
录制端侧赋能

集成录制 SDK 在移动端受限于屏幕尺寸，不易进行精细化创作，所以它的定位是进行基础剧本框架的创作与录制。

在此过程中，录制 SDK 首先要记录用户的操作信息和页面的基础信息，信息录入者在使用录制功能时，录制 SDK 会同步记录当前页面信息及与之相对应的音频录入，形成一个关键帧，后续录制以此类推，当所有信息录入完成之后，生成的多个关键帧会组成关键帧序列，结合一些基本信息，形成一个剧本框架，上传至服务器，便于录制者在后台进行精细化的创作。

同时录制 SDK 需要主动推断用户意图，减少录入者编辑。我们将关键帧的录入，按照是否产生页面跳转分为两种类型，对应不同类型，自动生成相异的路径。当录入者的操作产生页面跳转时，录制 SDK 在确定该操作的分类同时，主动将该处的语音输入标记为下一关键帧的描述，以减少录制者的操作。

录制全程，每个页面的打开时间也被作为关键帧的一部分记录下来，作为参考信息，帮助录入者调整剧本节奏。



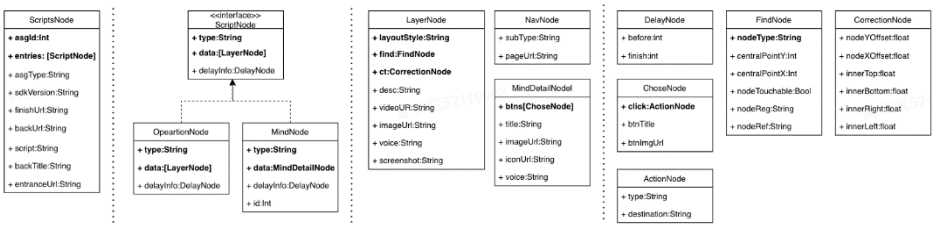
剧本录制侧示意图

标准协议设计

标准协议作为“零代码”的基石串联了录制到编辑的整个过程。

当前 App 中，操作类引导场景有数十种，我们通过传输模型和视图模型的结合，将核心字段提取，冗余字段剥离。在保证标准化与兼容性的前提下，将数十种场景抽象为四种通用事件类型，为关键帧的编排及业务场景的覆盖提供了便利。对心智类剧本而言，会随着用户的交互操作不断产生新的分支，最终成为一个复杂且冗余的二叉树结构。我们在设计此类协议时，将二叉树节点进行拍平，存储为一个 HashMap，两个关键帧的衔接可以以 id 为标识。

用户在使用 App 时，在某些需求指引下，会产生心智类和操作类剧本引导交替出现的情况。例如，商家（用户）打开推广页面后，出现一个心智类剧本——小袋动画伴随着语音：“老板好，小袋发现您开店 3 个月了，还没有使用过门店推广功能呢，请问您是不会操作还是担心推广效果不明显呢?”屏幕中会伴随两个按钮选择（1）不会操作；（2）担心推广效果。此时，如果用户点击了 1，会转向“操作类”剧本，所以我们在设计协议时，要尤为关注两种剧本的衔接。在这里，我们将协议进行了细化，将基础能力协议与展示类协议进行拆分。两种剧本共用一套基础能力协议，防止出现兼容性的问题。



部分协议节点设计

管理后台的编辑器引擎解析剧本协议后，完成内置逻辑的初始化，以及引导剧本中事件关键帧的渲染。编辑器引擎内部基于事件机制实现了可订阅的能力，当关键帧触发插入、编辑、调整顺序等事件时，所有其他的关键帧都可以订阅以上核心事件，实现完整的联动效果。编辑加工后的剧本协议，通过接入美团统一的动态下发平台，实现

剧本的灰度、全量、补丁的动态化发布能力。编辑器内建完整的生命周期，在操作的不同阶段暴露完整的事件勾子，支持良好的接入和扩展能力。

阶段成果

能力建设

我们抽象了如上图两种标准样式的剧本，线上使用较多的是操作引导类剧本，大多是之前积压的任务。目前，我们已经迭代出了一种标准化形态，接入方便，一般在新模块的提测期间，产运快速为此需求安排操作引导剧本跟随需求同步上线。也可以针对现有复杂模块设置引导，默认藏在 导航栏的“?”图标里，在合适的时机进行触发。

同时，用户心智建设不仅仅是常规的产品操作引导，我们也提供了**心智类剧本**（也叫概念类剧本），可以应用在需要“理念传递”或“概念植入”的场景里。在合适业务

场景以拟人化的方式给用户传递平台的制度与规范，让用户更容易接受平台的理念进而遵守经营的规范；比如可以在商家阅读差评时，执行一个情感化剧本（大概内容为差评是普遍现象，每 xx 条订单就容易产生一条差评，所以不用过于担心，平台也有公正的差评防护与差评申诉规则）；如果商家出现违规经营，也可以执行一个概念强化的严肃类剧本（大概内容为平台非常公正且有多重检查措施，不要试图在申诉中上传不实材料侥幸过关）。

值得一提的是，在这个过程中产出的图像特征定位、去 Alpha 通道的视频动画等能力也完成了技术储备，可以提供给其他场景使用。前文核心技术内容也申请了两项国家发明专利。

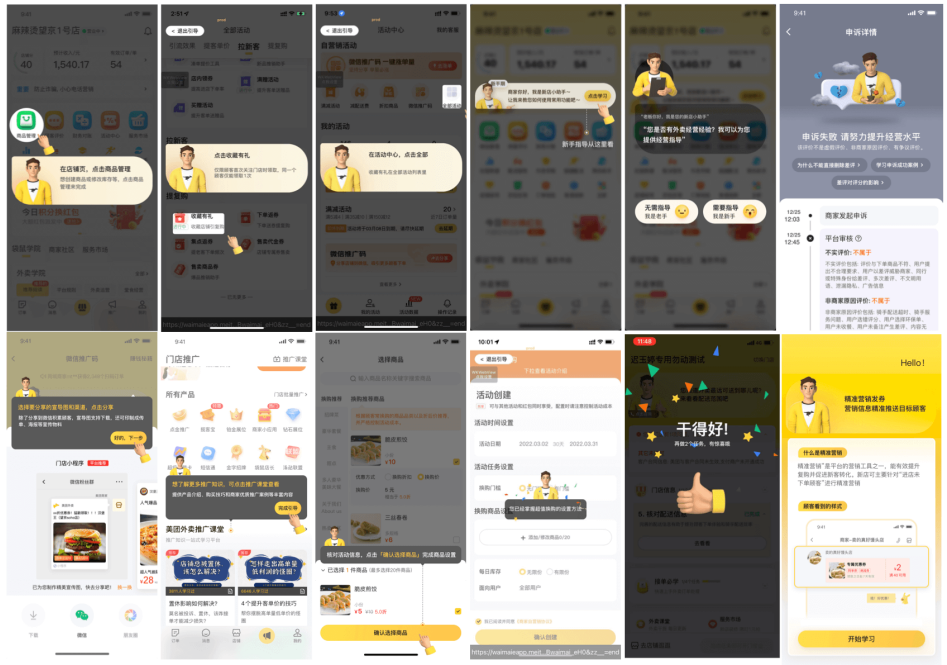
部分业务线上效果

- 新店成长计划，是剧本式引导应用的首个大需求。支撑新店成长计划项目顺利上线，目前的结果非常正向。ASG 支撑了整个项目 78.1% 的引导播放量，单个剧本开发成本 < 0.5d。综合观测指标“商家任务完成度”同比观测周期内，从 18% 提升至 35.7%，其他过程指标也有不同程度的提升。
- 超值换购，提供了心智类指引，是指导商家用最优的方式创建换购活动，结合过程数据预估访购率从 4% 提升至 5.5%，活动商家订单渗透率从 2.95% 提升至 4%，均有 35% 左右的涨幅。
- 配送信息任务引导，优化配送信息任务整体流程的行动点引导，避免引起商家行为阻塞，降低了用户操作成本与理解成本，提升商家在开门营业阶段满意度，同时提升商家对配送服务的认知度水平。
- …

自 2021 年 11 月上线以来，ASG 已经支撑了新店、活动、营销、广告等多个业务，在美团超过 20 个业务场景中进行了落地。总体看来，ASG 剧本式引导相比于传统引导方案，粗略估计的话，可以用约先前 1 / 10 的成本来提升约 20% 的效果。带入之前的结果测算公式，提效倍数（x = (1 / (1 − 90%)) * (1 + 20%) ），就是 12 倍。

从最终的结果来看，成本的降低，远比效果提升更加明显，所以本文对前者的论述篇

幅明显多于后者。目前，在效果提升层面，我们主要是对一些端能力比较基础的组合使用，对于效果提升我们并不担心，业内前沿的创新技术还有很多可以探索的可能，我们也会逐步跟进，使剧本效果更具有同理心、更加沉浸化。



总结与展望

本文介绍了美团外卖终端团队在用户心智建设领域的探索与实践。从业务现状与剧本式思维的思考出发，谈到了终端加管理后台的一站式设计，简化剧本接入门槛。后续，我们还谈到了传统 CV 与深度学习在剧本执行上起到的关键作用。整体看来，这个项目是基于终端能力拓展的一次大胆尝试，我们体会业务视角，通过不设限的跨团队的协作，完成对非技术人员的赋能。

结合目前的阶段性成果，我们验证了之前方向的正确性，下一步我们会继续从“更低的生产成本”与“更高的应用效果”两个角度进行深耕（例如组合元素剧本的易用性、剧本更新成本优化、引导时机结合规则引擎与意图猜测、折叠与再次唤醒逻辑等），以支撑更多类似场景的需求。并且，我们欣喜地看到终端的“容器无关性”收益杠杆明显，接下来还有很大的发挥空间。欢迎大家跟我们一起探讨交流。

作者简介

松涛、尚先、成浩、张雪、庆斌等，来自美团到家研发平台 / 外卖技术部；筱斌、民钦、德榜等，来自美团基础研发平台 / 视觉智能部。

参考文献

[1] App Annie. 2022 年移动市场报告
[2] HBR. When Low-Code/No-Code Development Works and When It Doesn’t
[3] Google. Compression Techniques
[4] Apple. Quartz 2D Programming Guide
[5] E. Karami, S. Prasad, M. Shehata “Image Matching Using SIFT, SURF, BRIEF and ORB: Performance Comparison for Distorted Images” Newfoundland Electrical and Computer Engineering Conference,St. johns, Canada, October, 2017
[6] Jiayi Ma, Ji Zhao, Junjun Jiang, Huabing Zhou, and Xiaojie Guo. “Locality Preserving Matching”, International Journal of Computer Vision, 127(5), pp. 512-531, May 2019.
[7] Huang, Lianghua, et al. Globaltrack: A simple and strong baseline for long-term tracking[C]//Proceedings of the AAAI Conference on Artificial Intelligence. 2020, 34(07): 11037-11044.

自动化测试在美团外卖的实践与落地

作者：少飞　闫旭　文文　军帅

1. 项目背景

美团外卖的业务场景比较多元化，除了外卖自身的业务，还作为平台承接了闪购、团好货、医药、跑腿等其他业务。除此之外，在全链路动态化的大基调下，外卖各个页面的技术形态也变得越来越复杂，除了 Native 代码，还包括 Mach（外卖自研动态化框架）、React Native、美团小程序、H5 等，不同技术栈的底层技术实现不同，渲染机制不同，进而对测试方式要求也有所不同，这也在无形中增加了测试的难度。下图汇总了美团多业务、多技术、多 App 的一些典型场景。

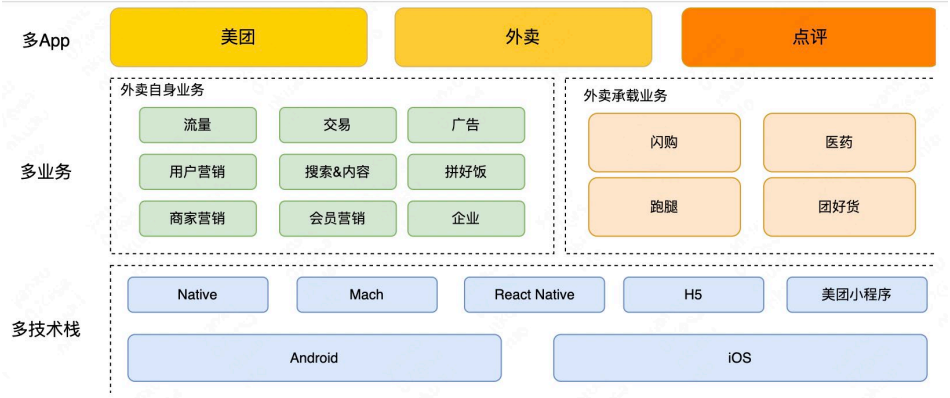


图 1　多业务、多技术栈、多 App

在产品交付上线的过程中，测试的占比也是非常大的，甚至大于总时长的 30%。如下图所示，整个测试包括了冒烟测试、新功能测试、二轮回归测试、三轮测试。然而，现在需求测试绝大部分还是采用非自动化的方式，这就使得人力成本变得非常之高。

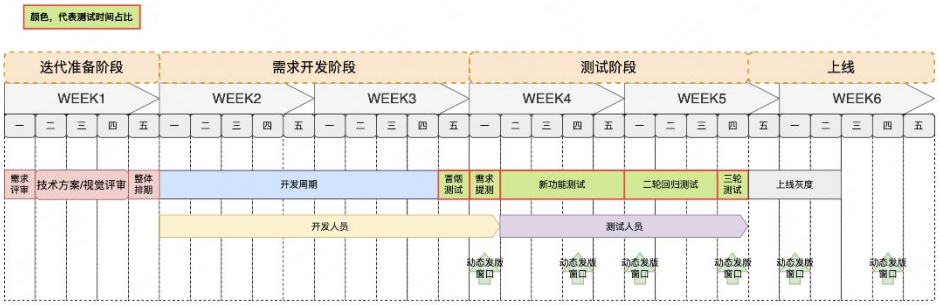


图 2　外卖迭代模型

另一方面，相比于 2018 年，2022 年的测试用例数量增长近 3 倍，已经超过 1 万 2 千条（如下图所示）。同时，外卖的业务是“三端复用”，除了外卖 App，还需要集成到美团 App 和大众点评 App 上，这样一来，测试工作量就翻了 3 倍，业务测试压力之大可想而知。如果按照当前的增长趋势持续下去，要保障外卖业务的稳定，就必须持续不断地投入大量的人力成本，所以引入能够支持外卖“多业务场景”、“多 App 复用”、“多技术栈”特点的自动化测试工具来提升人效和质量，势在必行。

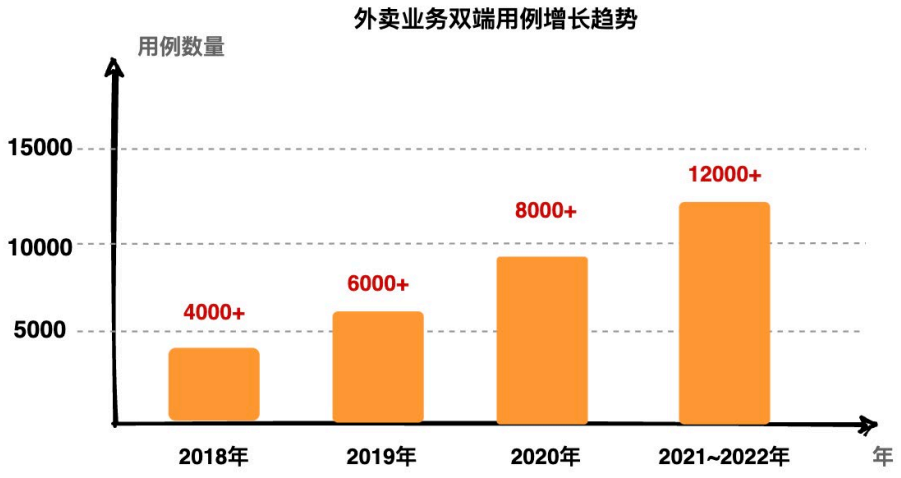


图 3　近几年用例增长变化

2. 项目目标

为了解决外卖面临的测试困境，我们尝试去探索一种零学习成本、低维护、高可用的自动化测试方案，能够支持外卖复杂多变的测试场景，它必须同时满足下面几点要求：

- **易用性**：工具 / 平台的上手难度，使用复杂度应该尽可能的低，因为自动化测试的目的是提效人力，而不是增加人力负担。
- **平台支持**：移动端至少需要覆盖 iOS 和 Android 双平台，同时基于外卖的业务特点，不仅需要对 Native 支持，也需要支持 Mach（自研局部动态化框架）、H5、React Native、美团小程序等技术栈。
- **稳定性**：自动化测试用例的执行需要有足够的稳定性和准确性，测试过程中不应因测试工具本身的不稳定而出现稳定性问题。
- **维护成本**：维护成本很大程度上决定了测试工作量的大小，因需求产生变动或架构重构等问题时，用例的维护成本应该尽可能的小。
- **可扩展性**：当测试方案不能满足测试需求时，工具 / 平台应具备可扩展的能力。

3. 方案选型

自动化测试工具那么多，自研是重复造轮子吗?

针对终端的 UI 自动化测试工具 / 平台可谓“屡见不鲜”，市面上也有很多相对成熟的方案，相信大家都有用过，或者至少有所耳闻，但这些方案是否能真的满足我们提效的诉求呢? 以下我们挑选了三类非常具有代表性的自动化测试工具 / 平台 – Appium、Airtest Project、SoloPi 进行了分析，来帮助大家对自动化测试技术建立一个认知：

—	Appium	Airtest Project	SoloPi
脚本语言	支持 Python，Java，JavaScript，PHP，C#，Ruby，OC 等	Python	/
数据记录（网络 / 本地）	不支持	不支持	不支持
环境模拟	不支持	不支持	不支持
上手难度	高，需要各种环境支持和语言学习	一般，不熟悉编程语言，也可以一定程度使用	低，用例即操作，不展示
问题溯源成本	高	高	高
维护成本	高	高	高
视图检索	基于 UI 控件的检索，支持 10 多种 UI 控件查找方式	基于图像识别和基于 UI 控件检索两种方式	基于图像识别和基于 UI 控件检索两种方式
源码集成	无需	可选	无需
WebView 支持	支持	支持	支持
用例编辑	支持	支持	支持
平台支持	iOS、Android、Windows	iOS、Android、Windows、游戏测试	Android

- Appium 是一个开源工具，用于自动化测试 iOS 手机、Android 手机和 Windows 桌面平台上的原生、移动 Web 和混合应用。它使用了各系统自带的自动化框架，无需 SDK 集成，Appium 把这些系统本身提供的框架包装进一套 API——WebDriver API 中，可以使用任何语言编写 Client 脚本向服务器发送适当的 HTTP 请求。这让不同技术栈的人员都能快速上手编写测试用例，可以选择自己最为熟悉的语言，但是对于没有语言开发基础的人来说，还是有一定学习成本，而且这种方式在多人协作时并没有太大作用，为了保证自动化用例的可维护性，团队内部应该需要统一脚本语言。值得一提的是：Appium 在 iOS、Android 和 Windows 测试套件之间可做的一定程度的复用代码。但是由于不同端界面及元素定位的差异，这往往是不现实的，更无法保证测试的准确性，所以这种所谓的“跨端”就变得毫无意义。

- Airtest Project 是由网易游戏推出的一款自动化测试平台，除了支持通过系统自带的自动化测试框架，还支持了通过图像识别的方式，对于非基于原生 UI 系统的一些游戏引擎提供了 SDK 的支持。其上手难度稍低，可以一定程度上通过 IDE 进行相关操作来生成简单的脚本指令。Airtest 虽然基于图像进行控件识别，为跨端提供了一定的可能性，然而图像识别并不能达到人眼识别的准确度，除此之外移动端页面的构成和游戏页面也存在不小的差别，页面元素的展示规则和样式受屏幕分辨率影响较大，单纯依靠图像识别来进行元素查找成功率不高，无法保证测试的准确性。
- SoloPi 是一个无线化、非侵入式的自动化测试工具，通过录制回放的方式进行 UI 自动化测试，SoloPi 虽然只支持 Android，但是在录制回放的这种方式中，还是极具代表性的。传统的自动化测试工具由于需要编写测试脚本，所以存在着一定的上手难度（Airtest 还是存在代码编辑的），便产生了 SoloPi 这种纯基于录制回放的自动化测试方式，将用例的所有操作事件进行录制，生成一个完整的录制脚本，通过对脚本的回放来还原所有的操作，从而进行自动化测试。但是，这种方式只能记录操作，而不能记录数据，在外卖这种数据驱动展示的场景下无法满足测试要求。并且外卖的业务要复用到美团 App 和大众点评 App 中，不同 App 存在部分视图和逻辑性的差异，SoloPi 也无法支持我们“一端录制多端回放”的测试场景。

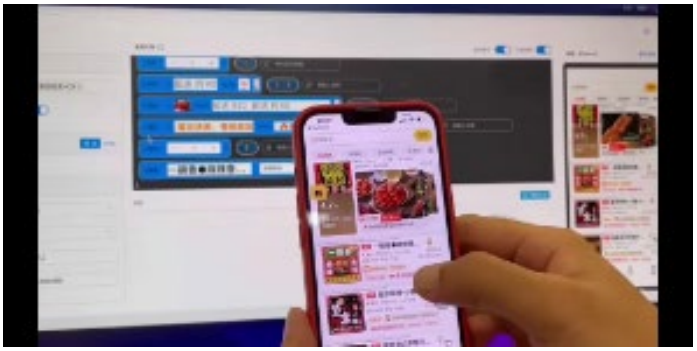
可以看出，以上这些自动化测试工具 / 平台对于数据记录，环境模拟、维护成本、跨 App 复用等方面，都是有所欠缺的。所以无论是哪种方案，在易用性、维护成本、稳定性、可扩展性以及最终的测试效果上，都无法满足我们对自动化测试的需求。我们并不是为了自动化而自动化，而是要解决实际的提效问题。

那么，怎样才能确定一个自动化工具 / 平台的可用性，并长期落地去使用自动化，带着上述提到的较高门槛的上手成本、操作繁琐的环境模拟、差强人意的测试成功率、定位模糊的测试缺陷、难以维护的用例脚本等几大重要痛点，**本文我们将介绍美团外卖自研的测试平台——AlphaTest**，都具备哪些能力以及是如何解决这些问题。

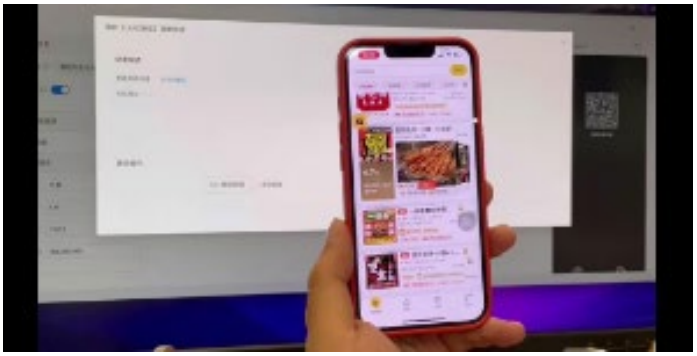
4. 实践和探索

一个自动化测试工具 / 平台能不能用起来，取决于他的上手成本和稳定性，即使工具的测试稳定性做的再好，使用的门槛高也会让人望而生却，反之亦然。所以 AlphaTest 平台为了上手简单，降低使用成本，采用了**基于录制回放**的方式进行设计，并且弥补了常规录制回放无法编辑的痛点，同时在手势操作的基础上增加了数据录制。整合美团系 App 的特性增加了环境模拟、跨 App 支持、混合技术栈的支持等能力，在使用简单的同时，也保障了用例的可维护性、测试的准确性等。我们先通过视频简单的了解一下：

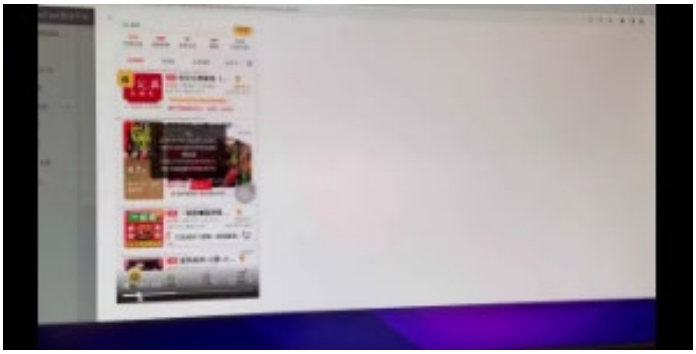
用例录制：



用例回放：



回放报告：



4.1 问题和挑战

注：这里我们将生成的自动化脚本统称为指令，将平台生成的用例统称自动化用例，将录制回放变成可视化的脚本指令，让用例变的易懂、易维护。

录制回放本身是一连串的操作数据的集合，是连续性的、不可拆分，因此几乎不具备可编辑性，这也就导致了用例维护成本极高。AlphaTest 虽然同样基于录制回放的方式生成自动化用例，但是我们将每一步的操作都具化成结构化的指令数据，并提供可视化指令编辑器，以支持查看编辑。

这些可视化的指令，完全通过录制自动生成，也不依赖于任何脚本语言。通过可视化用例指令编辑器，不仅为用例提供了编辑的可能性，同时大大地提高了用例的可阅读性，每一条测试用例在测试过程中每一步都做了什么、当时的界面是什么样的、都有哪些断言校验点，是显而易见的，不会存在像传统图文描述的测试用例那样，出现理解偏差。指令生成演示，手机录制与平台远端录制双模式支持：

图 4　指令编辑器

图 5　手机录制演示

图 6　平台远端录制演示

4.2 前置条件准备

一键环境模拟，解决操作繁琐的用例执行前的环境准备。

进行一个用例的测试之前，往往需要做大量的准备工作，比如切换 API 环境，定位到某个地点，登录指定账户等。这些需要准备的环境条件我们统称为前置条件。我们知道，前置条件的准备操作通常都不是一两个步骤就可以完成的，比如账号登录 / 切换：我们需要进入登录页，填写手机号 + 密码 / 验证码，点击登录等一系列动作来完成这个过程，非常繁琐，并且每次测试我们都需要准备，重复性高。因此，我们给 AlphaTest 设计了独立的前置条件模块，将用例拆成了两个部分：前置条件 + 操作步骤。

图 7　前置条件

与其它测试框架不同的是，AlphaTest 采用了 SDK 集成，但对业务无侵入的方式，因此可以通过编写白盒代码来实现前置条件的自动配置，只需要在平台添加需要的指令，下发到 SDK 后，即可根据相关指令完成前置条件的自动配置，不再需要重复进行相关的操作。并且这些前置条件支持复用，也不需要每次进行用例准备时的重复配置。AlphaTest 的前置条件，不仅有着基于美团内部服务及底层 Hook 的默认实现，也提供了 API 支持业务方自定义实现，比如实现不同的账号体系。

4.3 用例录制与回放的数据一致性

影响用例执行的不仅是代码，还有数据。

很多时候，自动化用例无法正常执行完成，可能是因为 App 回放时的本地数据及网络数据与录制时的不一致，从而导致用例执行流程的阻塞或 App 界面展示的不同。

这也是大多数自动化测试工具 / 平台测试通过率不高的主要因素，因此要保证测试成功率，我们需要控制变量，排除由数据产生的影响。

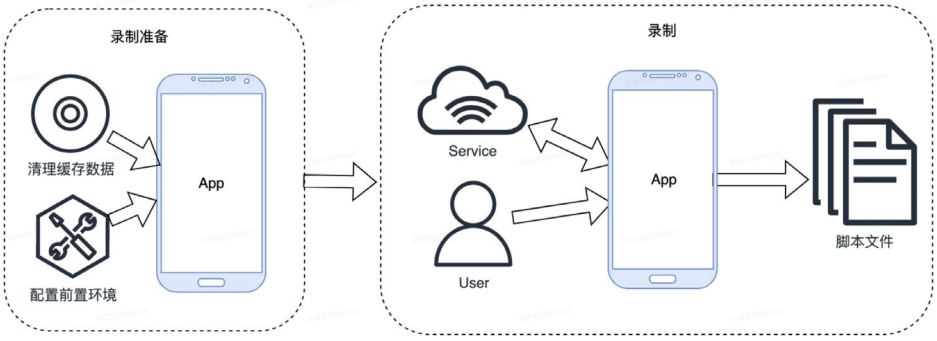


图 8　数据一致性

App 运行依赖的数据，有两部分——本地数据和网络数据：

- 本地数据是 App 在运行期间产生的缓存数据或持久化的存储数据。为了让用例在录制回放时都能够保持一致的本地数据环境，我们在录制和回放前都对 App 的本地数据进行了清理操作，这样用例在录制和回放的过程中，都可以保持一致的 App 初始化环境。
- 网络数据是驱动 App 交互呈现的基石，各种策略和 API 升级都会影响网络数据的响应，因此我们将用例录制过程中产生的网络数据也进行了录制，并将网络数据和对应的操作指令进行了关联和绑定，确定了数据产生的事件源。排除数据影响后，我们的自动化测试的成功率就取决于自动化操作的准确性了，这就回到了常见自动化框架范畴。

4.4 用例录制与回放的操作一致性

目标定位的准确性与手势定位的精准性。

UI 自动化测试的本质就是代替人去自动的做一步步的操作（点击、长按、输入、滑动等）。录制与回放过程的操作能否一致，是否精准，直接影响测试的成功率，决定了

工具 / 平台的可用性。

目标控件定位准确性：

操作行为是否一致首先需要确认操作目标是否一致。与一般测试工具 / 平台不同的是 AlphaTest 采用了 ViewPath + 图像 + 坐标的多重定位方案。得益于 SDK 集成的方式，我们的 ViewPath 可以记录更多的元素视图特征和执行不同的匹配策略。定位过程中会优先使用 ViewPath 进行目标控件检索，当目标控件查找异常时，会结合图像匹配和坐标匹配的方式进行兜底查找，来确保界面变化程度不大时，也能准确的查找到目标控件。

图 9　图像识别示意图

手势定位的精准性：

有了基于控件的目标定位之后，对于一些常用简单操作手势，比如点击、长按、断言、甚至输入都可以做到很好的支持，只需要找到对应的控件，在控件所在位置下发相应的触摸事件即可。我们知道，App 真正接收的触摸事件是屏幕上一个个精准的触摸点，在系统处理后，分发给当前 App 窗口，App 在接收事件后再继续分发，直到找到事件的最佳响应者，后续通过响应者链对事件消化处理。那我们要还原一个触摸事件的坐标点要如何确定呢？由于我们确定的只有控件，所以这个点自然而然就成了控件的中心点了。

大多数情况下，这些都可以很好地进行工作，但是对于一些多响应控件重叠的情况，可能会产生预想不到的操作误差。为了解决这样的问题，我们把控件定位与坐标定位进行了结合：基于纯坐标的定位是一种定位精准度非常高的定位方式，但是稳定性非常差，只有在屏幕分辨率完全一致且回放页面控件位置完全一致的情况下，才具备足够的可靠性，但这往往是不现实的，对测试环境机器量要求过高。

基于控件的定位，又存在着精准度不够的问题。使用坐标定位，如果定位区域足够小的话，那么受屏幕尺寸的影响就会越小，只需要确定在小范围内的相对位置即可。而基于控件目标的定位，恰恰可以把目标区域缩小到一个指定区域，我们刚好可以将二者结合起来，同时解决定位精准度和稳定性的问题。

对于复杂手势的支持，我们同样可以采用微分的方式，将一个复杂手势拆成多个简单手势的组成，比如我们可以将一个滑动操作的定位拆成两个部分：起始位置和终止位置，而这两个位置的定位，就变成了两个普通的单点手势操作定位了，可以通过上面提到的一个目标控件 + 相对坐标的形式进行定位。核心思想都是将基于屏幕坐标点的定位操作，缩小的目标控件的区域范围内，以达到不受设备分辨率的影响，实现操作行为一致的效果。

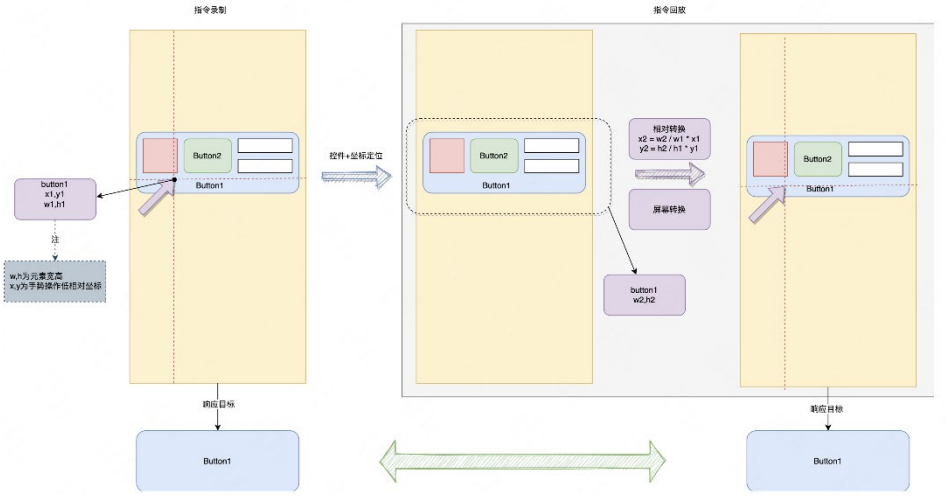


图 10　手势识别示意图

4.5 可溯源的自动化测试

测试全流程记录，问题溯源一键即达。

测试的目的是保证 App 运行的稳定，测试过程中出现 Bug 导致测试未通过时，需要溯源问题原因，发生的场景，乃至具体的执行步骤。这也是大多自动化测试工具 / 平台所欠缺的，即使发现了问题，排查工作也很困难；这个问题在手工测试的时候，更为严重，往往因为很多缺陷无法复现而难以定位。

AlphaTest 的自动化用例最小执行单元是操作指令，我们将测试过程的每一条指令的执行状况和过程中的界面快照进行了记录，并在指令执行失败时，对异常原因进行了初步分析。然后将整个用例的执行组合成了一份完整的测试报告，可快速溯源问题步骤。除此之外，我们还增加大量的日志上报，并将整个用例测试过程进行了视频录制，来进一步帮助疑难问题的排查。真正做到了用例回放测试可溯源。

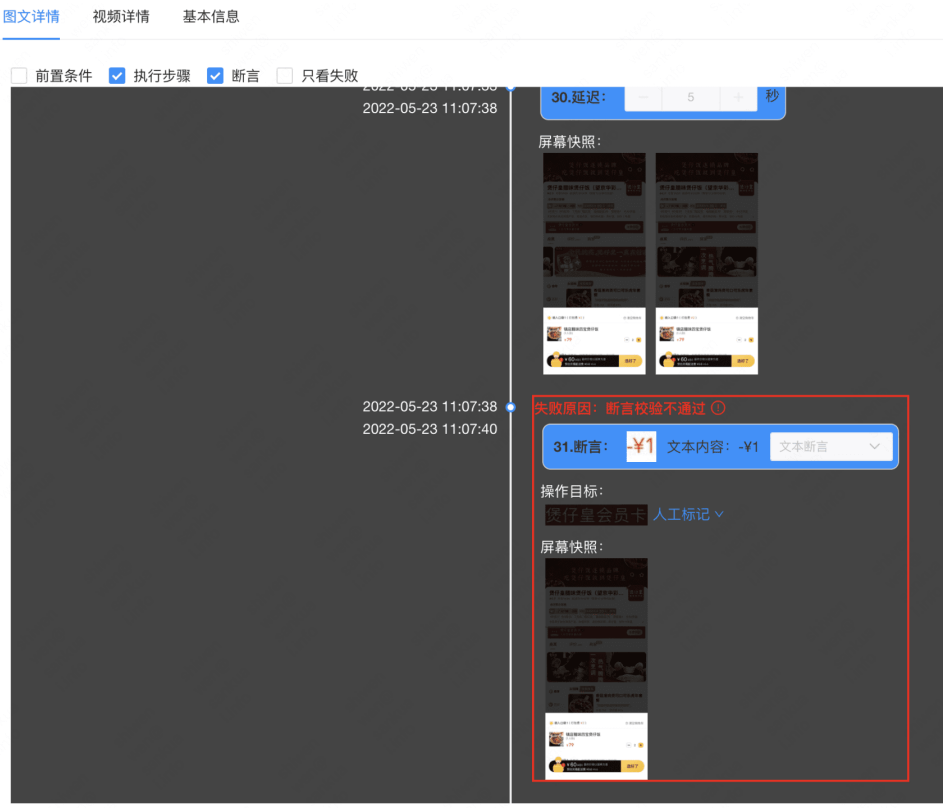


图 11　回放报告－图文详情

图12 回放报告-视频详情

图13 回放报告-基本信息

4.6 用例的维护

自动化用例需要持续地投入人力来维护么？架构升级，页面重构，用例需要全部重新录制么？

因自动化工具 / 平台众多，阻碍长期落地使用的一大问题是用例维护成本高，很多工具 / 平台让我们即便是使用上了自动化，但还需要持续投入人力维护用例的更新，最终的提效收益微乎其微。对于用例更新维护，我们可以梳理划分成三个场景：

- 需求发生重大变更，整体的业务执行流程及相关的校验点都需要进行大量的调

整。对于这种情况，无论是何种自动化测试工具 / 平台，都是需要正常进行用例变更重录以适应新的需求。

- 需求发生略微变更，业务流程基本一致，需要调整的校验点、操作以及数据或不影响整体流程的步骤。对于此场景，AlphaTest 通过指令编辑器与操作录制，支持指令增删改以及数据和场景的还原，帮助用户快速的进行用例调整，而无需重新录制用例。例如：修改网络数据字段、视图变更路径、断言替换目标等。

图 14　指令编辑

- 和业务需求不同，我们的技术实现也会发生迭代。随着 App 技术架构不断的演进，经常会面临着架构升级，页面重构甚至技术栈变迁等这样的技术升级。这些变动需要覆盖大量的测试用例，其中大量的自动化用例又可能会因为变动而导致失效，需要重新录制。为此，AlphaTest 设计一套利用相近分辨率机器进行用例自动修正的功能：利用图像 + 坐标进行二次识别定位，元素定位成功并校验通过后，生成新的 ViewPath，更新对应的用例指令，对用例进行自动修复，修复后可在任意回放。

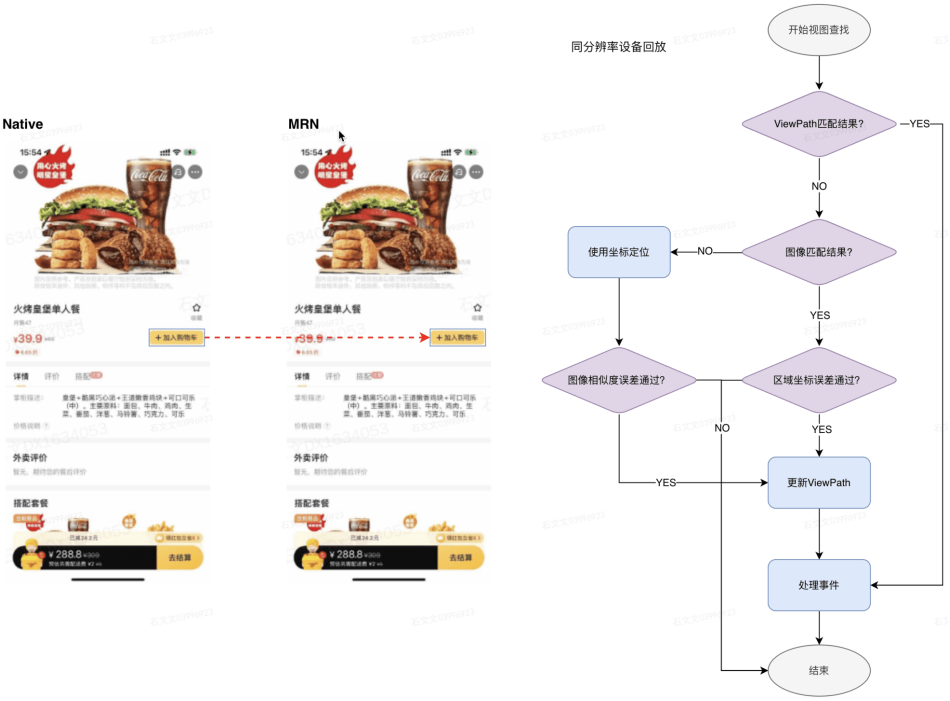


图 15　自修复能力

4.7 跨 App 回放用例

同一份代码运行在不同的 App 上，是否需要重新编写多份用例?

美团系的一些业务可能会复用在多个 App 上。比如外卖有独立 App，但同时也要复用到美团和点评 App 上，这些功能，几乎共用一份代码，而测试人员却不得不对每个 App 上的业务功能都进行测试，维护多份用例。由于业务本身实现是一致的，那我们可以通过适配不同 App 之间的差异，来让一个业务 Case 可以横跨多个 App 回放，这便可以将成本缩减好几倍，这些差异主要体现在：

- **前置条件和初始页面**：业务的初始页面进入路径不同，例如外卖 App 打开 App 就进入到了外卖首页，但是在美团 App 中就需要从美团首页跳转到外卖频道。同时由于不同 App 的样式风格、设计规范、业务特性等因素，也会造成首页代码逻辑和视图层级的差异。

- **AB 实验配置**：不同 App 所配置的实验可能不同，不同的实验会导致不同的样式和代码逻辑。
- **网路接口映射**：不同 App 中相同业务场景涉及的接口有所不同。
- **页面 Scheme 映射**：不同 App 中相同页面的跳转 Scheme 也不相同。

AlphaTest 平台支持 App 维度各项差异数据配置，当 SDK 检测用例回放环境与录制环境不一致时，会自动进行映射适配，从而让用例运行到了不同 App 上。

4.8 埋点的录制回放

除了功能测试，我们在日常开发和测试的工作中，还会面临另外一个比较重要的问题就是埋点测试。因此，我们在自动化的基础上扩展出埋点自动化测试。埋点自动化测试的核心思想是，通过对比录制时期和回放时期的埋点上报时机和上报参数进行判断。为了保证埋点自动化测试的稳定性，我们主要采用以下的障机制：

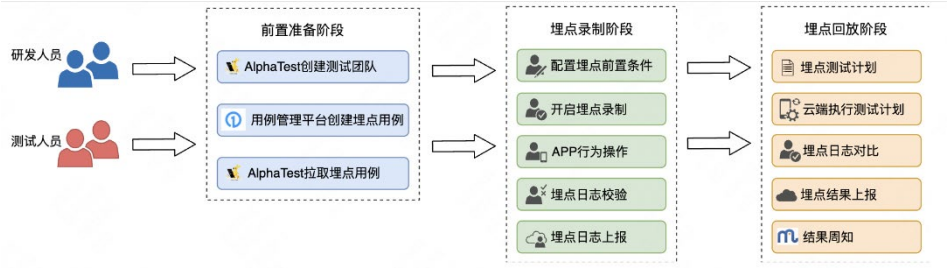


图 16　埋点自动化测试示意图

- **字段规则配置**：埋点自定义参数千姿百态，甚至有些字段每次代码执行都不一致，如果进行完全匹配结果注定是失败的，所以我们在 AlphaTest 平台提供了埋点字段规则配置功能，通过人为设置的方式来避免埋点自定义参数校验失败。App 重启进入录制状态时，用户就可以操作 App，平台会记录用户的操作行为，当产生相应的埋点日志的时候会将日志信息打印在日志区域（如下图 17 所示），在该过程中也会对埋点日志进行一定的校验。重点将操作时机、埋点日志一并保存到服务端。

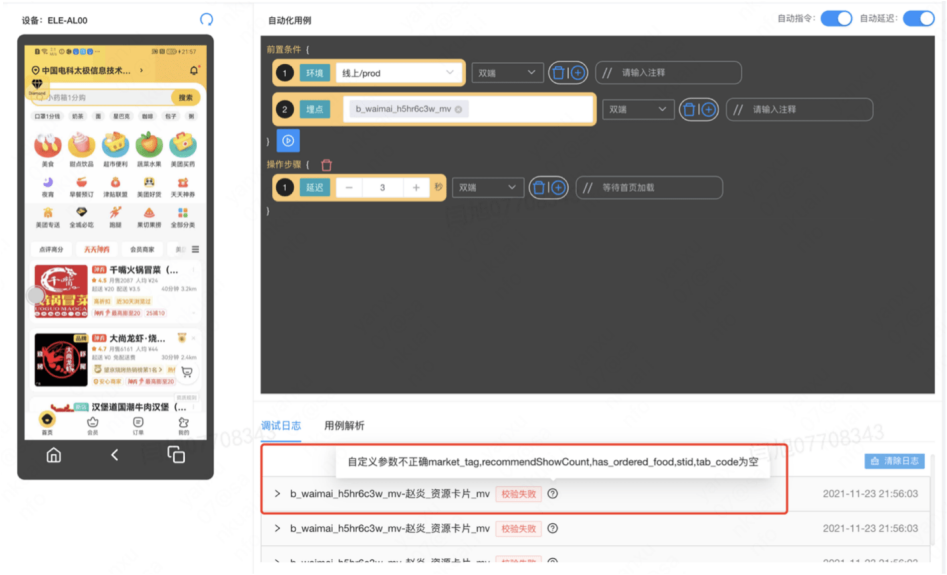


图 17　埋点上报数据控制台打印

- **埋点时机校验**：针对时机校验，程序并不支持埋点曝光的”1px 曝光”,”下拉刷新曝光”,”页面切换曝光”,”切前后台曝光”这些规则，主要的原因是每一个业务方在对埋点曝光的规则都是不一致的，而且该规则的实现会极大耦合业务代码。在针对时机校验我们目前只支持：

[1] 点击埋点上报时机校验，程序通过事件监听和埋点类型信息来判断点击埋点上报的时机是否是在点击的操作下产生的，如果不是则报错。

[2] 埋点重复上报校验，针对一般情况下用户一次操作不会产生两个相同的埋点上报，所以程序会校验某个事件下发生的所有埋点日志进行一一校验，检测是否具有 2 个或多个埋点日志完全一致，如有发生则会上报错误。

- **结果校验**：回放完成后，我们会对比录制和回放时的埋点数据，根据配置好的字段规则校验埋点上报是否符合预期。

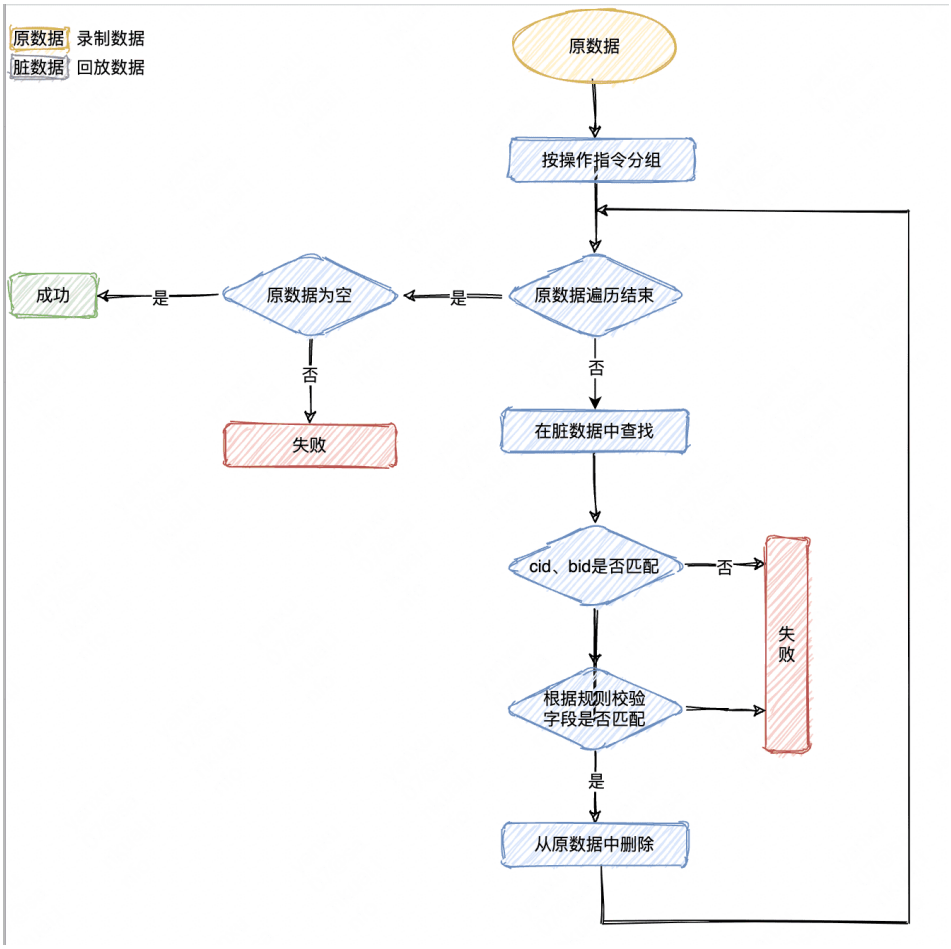


图 18　埋点校验流程图

5. 测试流程

AlphaTest 的核心测试流程始终聚焦在用例的录制与回放环节，整个流程涉及到自动化任务触发、回放集群调度、断言服务、消息推送等核心模块。

以 UI 自动化和埋点自动化的流程为例，AlphaTest 以业务团队为基本单元，可以和各团队的测试用例进行关联，定时同步状态。同时利用需求评审线上化做为基础，将自动化用例和研发流程中的 PR、集成打包、二轮回归等节点相结合，定时触发自动

化用例并将结果报告推送给相关负责人。

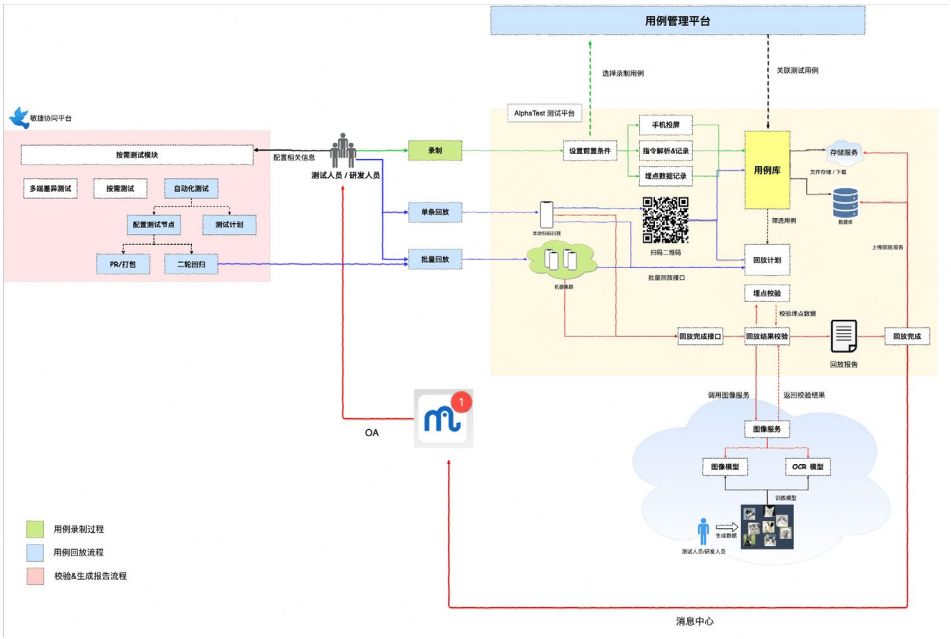


图 19　建设自动化测试流程闭环

录制用例：

[1] 首先在 AlphaTest 平台选择要录制的测试用例，打开待测试 App 进行扫码即可进入用例待录制状态，此时可以设置用例需要的前置条件（账号信息、Mock 数据、定位信息等），之后点击开始按钮后，手机便会自动重启，开始录制。

[2] 用户按照测试用例步骤，正常操作手机，AlphaTest 会将用户的操作行为全部记录下来，并自动生成语义化的描述语言显示在 AlphaTest 平台上，与此同时产生的网络数据、埋点数据等校验信息也会一并存储下来。

[3] 在录制的过程中可以快捷的打开断言模式，将页面上想要校验的元素进行文本提取 / 截图等操作记录下来，用于后续回放过程中对相同元素进行校验。

[4] 测试步骤全都执行完毕后，点击保存按钮即可生成本条自动化用例。

用例回放：

[1] 扫描对应自动化用例的二维码即可进行回放，回放过程中会将用户录制的行为、网络数据进行一比一还原，并且辅助有全过程视频录像，用于后续问题排查和溯源。

[2] 回放过程中碰到断言事件时，会将断言的元素进行文本提取 / 截图，上传至 AlphaTest 平台。回放完成后，会将回放时候的断言截图和录制时的断言截图进行图像对比，作为整个测试结果的一项。

[3] 回放过程中的埋点数据也会一并记录下来，并和录制时候的埋点数据和上报时机进行对比，自动提取出其中的差异项。

[4] 回放完成后，会生成完整的测试报告并将结果通过 OA 推送至相关人员。

回放计划：二轮回归测试中，回放用例数量多达几百条，为了做到全流程的自动化，我们提供了回放计划的概念，可以将多个自动化用例进行编组管理，每一组就是一个回放计划。触发一个计划的回放即可自动触发计划内的所有自动化用例。整个计划都执行完成后，会通知到指定的计划负责人或群组。

5.1 自动化任务触发

在整个外卖 C 端敏捷迭代的流程中，打包平台主要承接了业务需求发起到需求交付的流程，作为 AlphaTest 的上游平台，可以提供打包信息并触发自动化用例回放任务。以下简单展示 AlphaTest 与敏捷协同平台的交互流程：

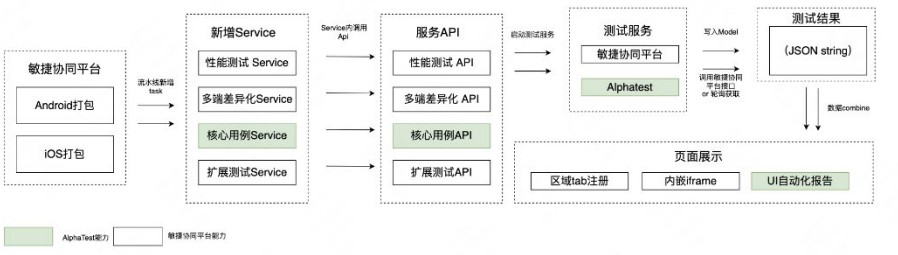


图 20　AlphaTest 与敏捷协同平台交互流程图

5.2 回放集群调度

整个测试过程真正的解放双手，才能算的上是自动化。因此，我们着手搭建了自己的自动化机器集群，可以 24 小时不间断的执行测试任务。为了保证任务回放能够顺利完成，我们在不同阶段增加了相应的保活策略。在极大程度上提高了任务执行完毕的成功率。

- **执行流程**：回放任务通过用户在平台手动触发或者二轮自动触发。新增的回放任务经过任务拆分系统拆分成 n 个子任务，加入到不同设备的回放任务队列中。每个子任务经过占用设备 -> 安装待测 App-> 应用授权 -> 打开 scheme-> 上报结果等步骤完成回放操作。
- **节点保活机制**：针对回放流程中每一个节点，失败后进行 N（默认为 3）次重试操作。减少因网络波动，接口偶现异常导致的回放失败数量。
- **子任务保活机制**：每个回放流程，失败后进行 N（默认为 3）次断点重试。减少因设备异常，SDK 心跳上报异常导致的回放失败数量。
- **父任务保活机制**：一个父任务会被拆分成 N 个子任务，当其中的一个子任务 S1 在节点保活机制和子任务保活机制下仍然执行失败之后，父任务保活机制会尝试将子任务 S1 中未执行完毕的用例转移到其他活跃状态的子任务中。减少因设备异常，设备掉线等问题导致的回放失败数量。

图 21　机器集群

5.3 断言服务

用例断言是整个自动化用例验证的核心步骤，我们的断言服务依据用例的实际情形可以分别进行文字与图像的断言。其中图像断言服务依托于自建的图像对比算法服务，可以高效进行录制回放断言图像的对比，图像对比准确率可以达到 99% 以上。

录制阶段：

[1] 录制时增加断言决策信息的自动采集。

[2] 和正常流程一样，提取区域的截图信息。

[3] 如果是文本组件，则提取文本内容，如果是图片组件，则提取图片二进制编码或图片 URL，同时提取区域内的布局信息。

回放阶段：

[1] 回放时，提取和录制时一致的内容（文本信息、图片编码、区域截图、布局信息）。

[2] 将回放时的断言信息上传至 AlphaTest 平台。

[3] AlphaTest 平台对断言结果进行校验，首先是基于模型的图像对比，如果判定为一致，则直接标记结果。

[4] 如果判定为不一致、则匹配“断言失败数据集”，如果能够匹配上，则标记结果。如果匹配不上，则需要人工选择匹配类型。

[5] 匹配类型为“文本校验”、“根据图片信息校验”、“人工校验”。如果前两项判定为一致，则直接标记结果。如果“人工校验”的结果为确实两张图不一致，则直接标记结果，结束。

[6] 如果“人工校验”结果为一致，既上述所有判定都不准确，则需要人工对两张图中判定错误的原因进行分类（具体类型待定），同时将断言存储到失败数据集。

[7] 模型自动训练，当数据集超过一定的阈值、通过定时触发、或者手动触发的方式，触发模型自动训练，训练完成后自动部署到 AlphaTest 平台，不断迭代。

图像服务：图像对比模型采用基于度量学习的对比算法，将图像对的一致性判别转换为图像语义的相似度量问题。度量学习（Metric Learning），也称距离度量学习（Distance Metric Learning，DML）属于机器学习的一种。其本质就是相似度的学习，也可以认为距离学习。因为在一定条件下，相似度和距离可以相互转换。比如在空间坐标的两条向量，既可以用余弦相似度的大小，也可以使用欧式距离的远近来衡量相似程度。度量学习的网络采用经典的 Siamese 结构，使用基于 resnext50 的主干网络提取图像的高级语义特征，后接 spplayer 完成多尺度特征融合，融合后的特征输出作为表达图像语义的特征向量，使用 ContrastiveLoss 进行度量学习。

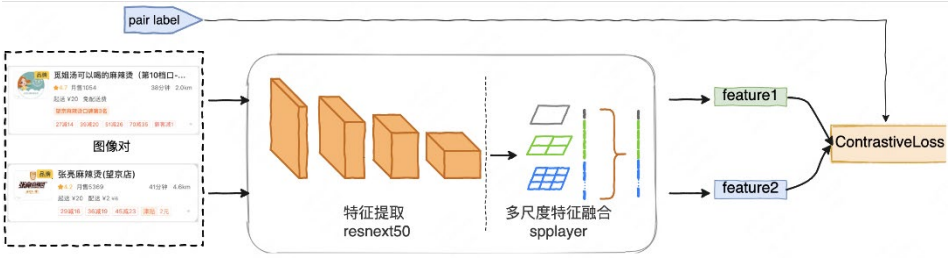


图 22　训练过程

[1] **预训练过程**：resnext50 网络是使用 ImageNet 的预训练模型。

[2] **数据增强**：为增加数据的丰富性、提高网络的泛化性能，数据增强的方式主要包括：图像右下部分的随机剪切和添加黑色蒙层（相应改变图像对的标签）。这种数据增强符合控键截图实际情况，不会造成数据分布的改变。

[3] **对比损失**：对比损失函数采用 ContrastiveLoss，它是一种在欧式空间的 pair based loss，其作用是减少一致图像对距离，保证不一致图像对的距离大于 margin，其中 margin=2。

$$L(x,x')=(1-y)\times\|f(x)-f(x')\|_2^2+y\times min(margin-\|f(x)-f(x')\|_2^2,0)$$

图 23　训练过程

[4] **相似度量**：相似度量也是采用计算图像对特征向量的欧式距离的方法，并归一化

到区间 [0, 1]，作为输出的图像对相似度。

5.4 消息推送

消息推送作为回放流程的最终环节，我们依赖于美团内部自建的消息队列服务与 OA SDK 消息推送能力，可以进行测试报告的实时推送。在此之上，还可以针对不同团队的推送诉求，做消息模板的定制化。

消息定制：消息推送与触达的核心，是满足业务诉求；不同业务对自动化测试报告中各项指标的关注点不同，这就需要 AlphaTest 具备消息推送定制的能力；将消息推送的模板以配置文件的形式提供出来，不同的业务使用不同的业务消息配置文件；再利用 OA 提供的图文、多媒体等消息推送能力，可以将自动化测试报告的各项指标自定义拆分；除此之外，消息还需要减少冗余，在这个信息泛滥的时代，我们愿意为无孔不入的消息、通知做减法，只将最重要、最核心的消息推送给最需要的人，既可以推动自动化测试流程的高效流转，又可以让各相关业务人员享受到自动化测试能力的便捷性。

一键触达：以往的研发人员冒烟测试，主要依赖于测试人员在用例管理平台建立测试计划，研发人员根据用例进行手工用例测试结果标记，之后去提测完成后续流程。这中间缺失的主要环节是，难以对研发人员冒烟测试的质量进行把控。而 AlphaTest 正可以解决此问题，流程转换为，研发人员在敏捷协同平台触发一键提测流程，调用 AlphaTest 的自动化测试能力对冒烟用例进行自动化测试回归，完成之后将测试生成的测试报告同步提测平台，作为研发人员冒烟的结论依据，同时在冒烟过程中发生的问题，也可以及时通知到对应的研发人员与测试人员进行改正。既保证了质量，又避免了人力空耗。

6. 落地与实践

外卖 C 端主要承担了用户在 App 端点餐、下单、配送的所有核心流程，场景繁多、业务复杂，这也给测试人员的版本测试带来了诸多挑战，其中最核心也最耗费人力的便是二轮回归测试环节。目前，C 端采用的双周敏捷迭代的开发方式，每个迭代周期

给测试人员用来进行二轮核心流程回归的时间为三天，为此 C 端测试团队投入了许多人力资源，但即便如此，仍难以覆盖全部流程；而 AlphaTest 的设计初衷也正是为解决此问题——UI 测试流程全覆盖及自动化验证。

6.1 业务共建

用例的转化与维护

[1] AlphaTest 在外卖 C 端测试团队的落地初期，我们采用了共建的模式，也就是业务研发人员与对应测试人员共同来进行用例录制与维护的工作；推荐这种工作模式的核心原因是，在 C 端功能迭代流程中的二轮周期的原有工作模式为，研发人员进行二轮冒烟测试，完成测试之后提交二轮包交由测试人员进行二轮回归测试，所以这本来就是一个双方都需要参与的环节；而二轮测试作为版本上线前的最重要一个测试流程，保证核心流程的正常也是测试人员与研发人员所关心重点。

[2] 经过多轮的使用与磨合之后，这种模式被证明是行之有效的，在整个 C 端二轮用例的转化过程中，测试人员主要负责了用例的录制与迭代流程，研发人员则主要负责版本回放数据的统计及问题用例的发现与解决。

外卖二轮落地情况

[1] 目前，AlphaTest 已经在外卖多个业务落地，支持了大于 15 个版本的二轮回归测试，用例覆盖率达到 70%。

[2] 覆盖了 Native、Mach、React Native、美团小程序、H5 技术栈的测试工作，能力上可进行支持：UI 自动化测试、埋点自动化测试、动态化加载成功率自动化测试、无障碍适配率自动化测试。

未来，我们会朝着“智能化”和“精准化”两个方向探索，覆盖更多测试场景的同时，更进一步提升测试人效。

6.2 实践效果

测试方向	同App回放成功率	跨App回放成功率
功能自动化	iOS:97.4%、Android: 94.7%	iOS:95.8%、Android: 91.1%
埋点自动化	iOS:96.3%、Android: 96%	iOS:95%、Android: 91%

7. 参考资料

[1] https://appium.io
[2] http://docs.seleniumhq.org/projects/webdriver
[3] http://airtest.netease.com/index.html
[4] https://github.com/alipay/SoloPi

深入理解函数式编程（上）

作者：俊杰

前言

本文分为上下两篇，上篇讲述函数式编程的基础概念和特性，下篇讲述函数式编程的进阶概念、应用及优缺点。函数式编程既不是简单的堆砌函数，也不是语言范式的终极之道。我们将深入浅出地讨论它的特性，以期在日常工作中能在对应场景中进行灵活应用。

1. 先览：代码组合和复用

在前端代码中，我们现有一些可行的模块复用方式，比如：

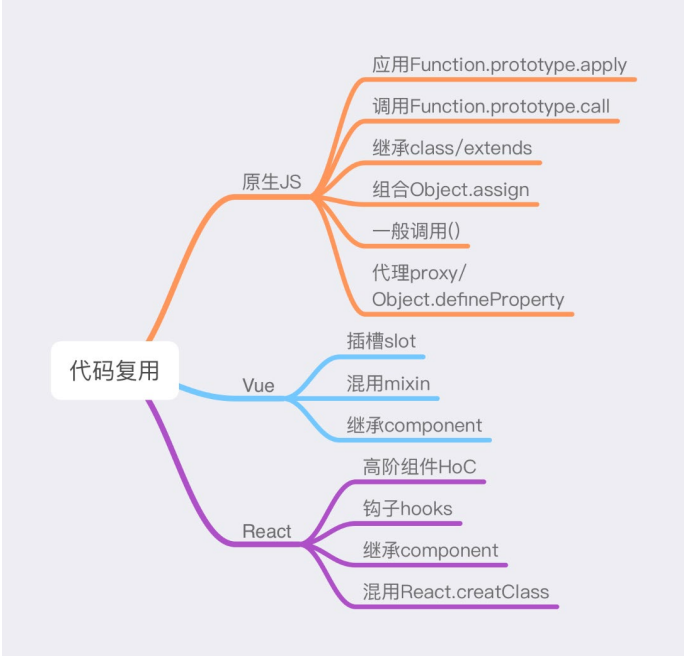


图 1

除了上面提到的组件和功能级别的代码复用，我们也可以在软件架构层面上，通过选择一些合理的架构设计来减少重复开发的工作量，比如说很多公司在中后台场景中大量使用的**低代码平台**。

可以说，在大部分软件项目中，我们都要去探索**代码组合和复用**。

函数式编程，曾经有过一段黄金时代，后来又因面向对象范式的崛起而逐步变为小众范式。但是，函数式编程目前又开始在不同的语言中流行起来了，像 Java 8、JS、Rust 等语言都有对函数式编程的支持。

今天我们就来探讨 JavaScript 的**函数**，并进一步探讨 **JavaScript 中的函数式编程**（关于函数式编程风格软件的**组织、组合和复用**）。

图 2

2. 什么是函数式编程?

2.1 定义

函数式编程是一种风格范式，没有一个标准的教条式定义。我们来看一下维基百科的定义：

函数式编程是一种编程范式，它将电脑运算视为函数运算，并且避免使用程序状态以及易变对象。其中，λ 演算是该语言最重要的基础。而且 λ 演算的函数可以接受函数作为输入的参数和输出的返回值。

我们可以直接读出以下信息：

1. 避免状态变更
2. 函数作为输入输出
3. 和 λ **演算**有关

那什么是 λ **演算**呢?

2.2 函数式编程起源：λ 演算

λ 演算（读作 lambda 演算）由数学家**阿隆佐 · 邱奇**在 20 世纪 30 年代首次发表，它从**数理逻辑**（Mathematical logic）中发展而来，使用变量绑定（binding）和代换规则（substitution）来研究函数如何抽象化定义（define）、函数如何被应用（apply）以及递归（recursion）的形式系统。

λ 演算和图灵机等价（图灵完备，作为一种研究语言又很方便）。

通常用这个定义形式来表示一个 λ **演算**。

```
x       变量                          用字符或字符串来表示参数或者数学上的值或者表示逻辑上的值
(λx.M)  抽象化                         一个完整的函数定义（M是一个 lambda 项），在表达式M中的`x` 都会绑定为变量 x。
(M N)   函数应用Function Application  将函数M作用于参数N。 M 和 N 是 lambda 项。
```

图 3

所以 λ **演算式**就三个要点：

1. **绑定关系**。变量任意性，x、y 和 z 都行，它仅仅是具体数据的代称。
2. **递归定义**。λ 项递归定义，**M** 可以是一个 λ 项。

3. 替换归约。λ 项可应用，空格分隔表示**对 M 应用 N**，N 可以是一个 λ 项。

比如这样的演算式：

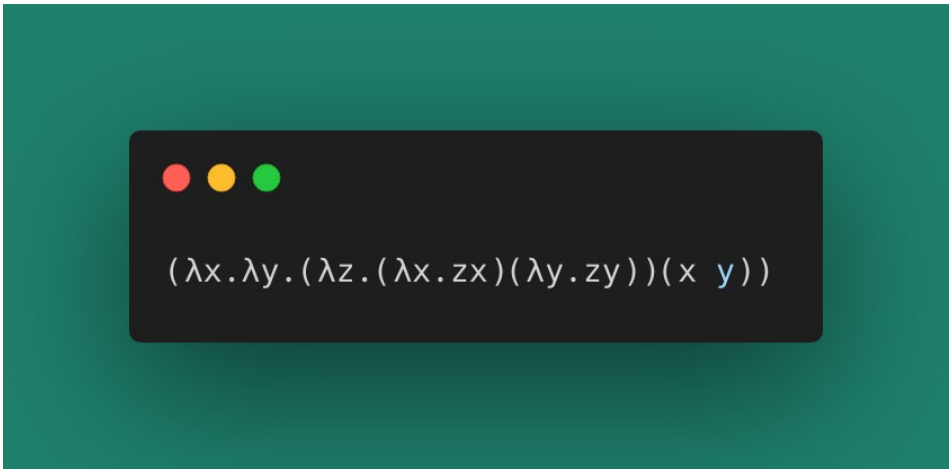


图 4

通过变量**代换（substitution）**和**归约（reduction）**，我们可以像化简方程一样处理我们的演算。

λ **演算**有很多方式进行，数学家们也总结了许多和它相关的规律和定律（可查看维基百科）。

举个例子，小时候我们学习整数就是学会几个数字，然后用加法 / 减法来推演其他数字。在函数式编程中，我们可以用**函数**来定义自然数，有很多定义方式，这里我们讲一种实现方式：

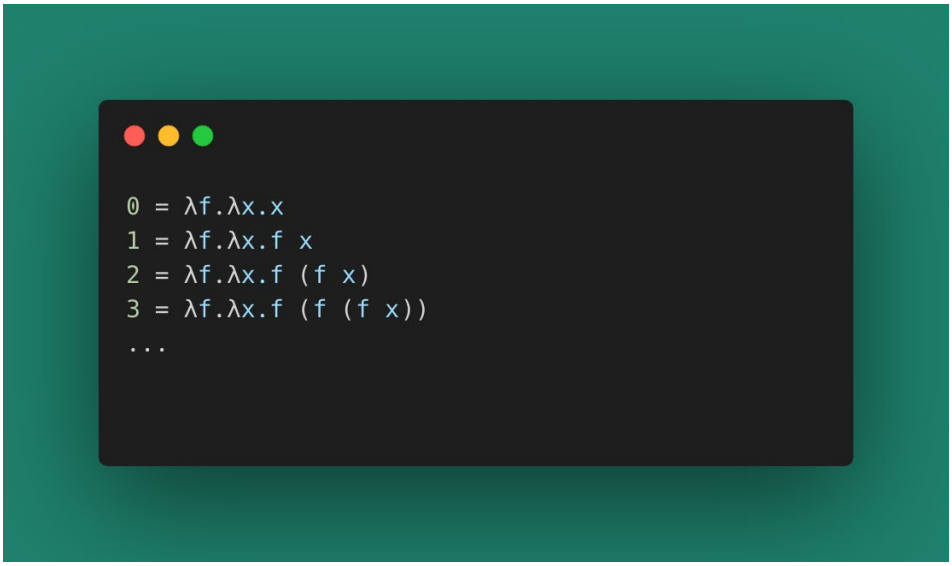


图 5

上面的演算式表示有一个函数 f 和一个参数 x。令 0 为 x，1 为 f x，2 为 f f x…

什么意思呢？这是不是很像我们数学中的幂：a^x（a 的 x 次幂表示 a 对自身乘 x 次）。相应的，我们理解上面的演算式就是**数字 n 就是 f 对 x 作用的次数**。有了这个数字的定义之后，我们就可以在这个基础上定义运算。

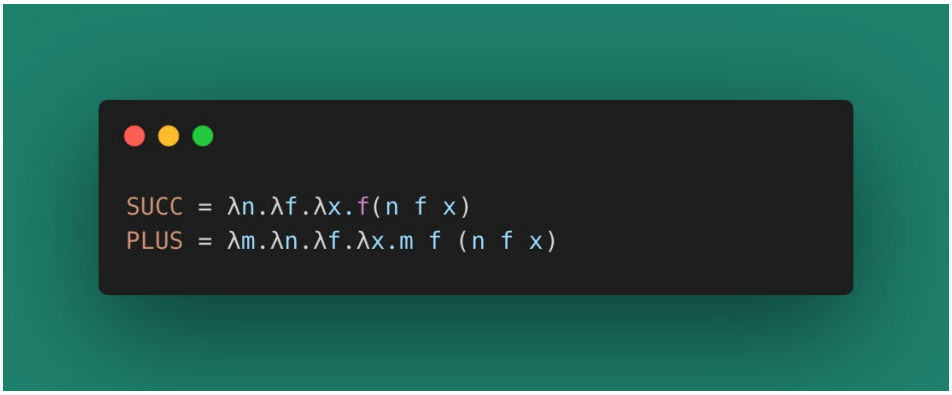


图 6

其中 SUCC 表示后继函数（+1 操作），PLUS 表示加法。现在我们来推导这个正确性。

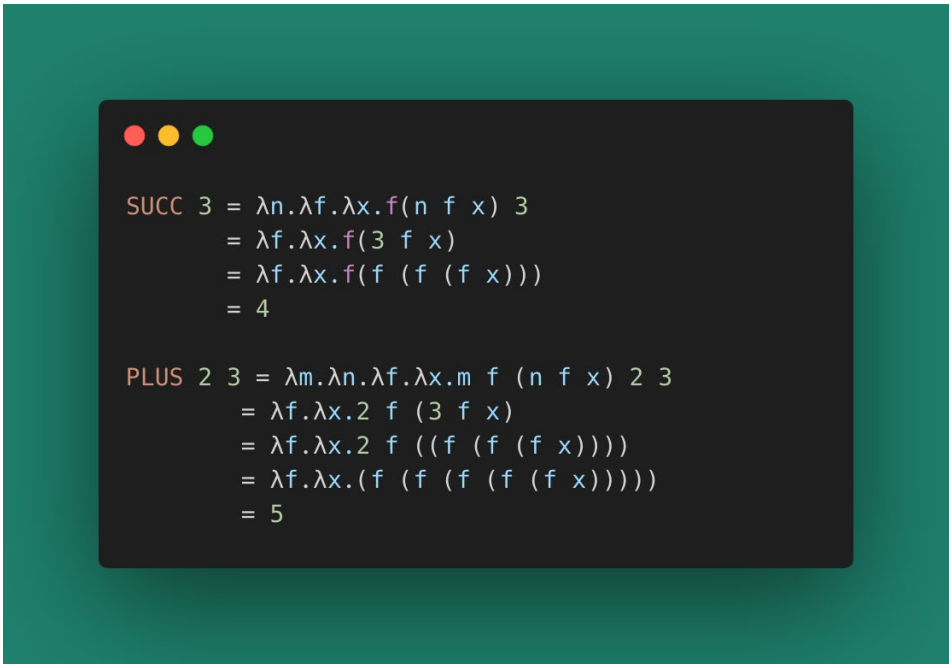


图 7

这样，进行 λ **演算**就像是方程的代换和化简，在一个已知前提（公理，比如 0/1，加法）下，进行规则推演。

2.2.1 演算：变量的含义

在 λ **演算**中我们的表达式只有一个参数，那它怎么实现两个数字的二元操作呢？比如加法 a + b，需要两个参数。

这时，我们要把函数本身也视为值，可以通过把一个变量绑定到上下文，然后返回一个新的函数，来实现数据（或者说是状态）的保存和传递，被绑定的变量可以在需要实际使用的时候从上下文中引用到。

比如下面这个简单的演算式：

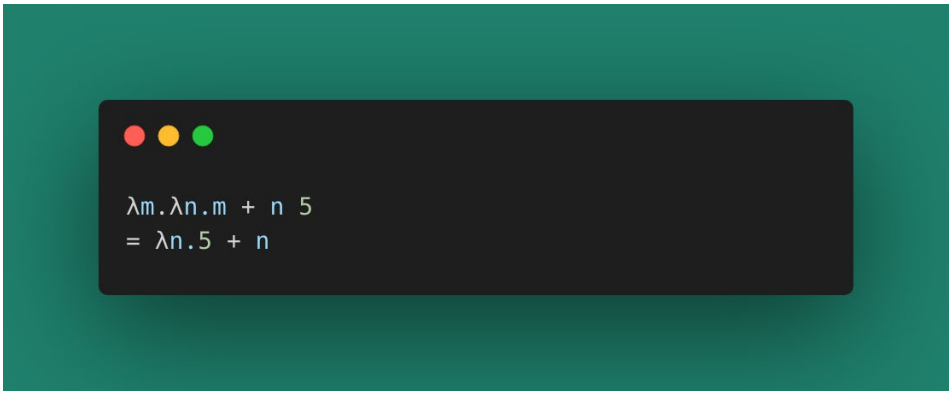


图 8

第一次函数调用传入 m=5，返回一个新函数，这个新函数接收一个参数 n，并返回 m + n 的结果。像这种情况产生的上下文，就是 Closure（**闭包，函数式编程常用的状态保存和引用手段**），我们称变量 m 是被**绑定**（binding）到了第二个函数的上下文。

除了绑定的变量，λ **演算**也支持自由的变量，比如下面这个 y：

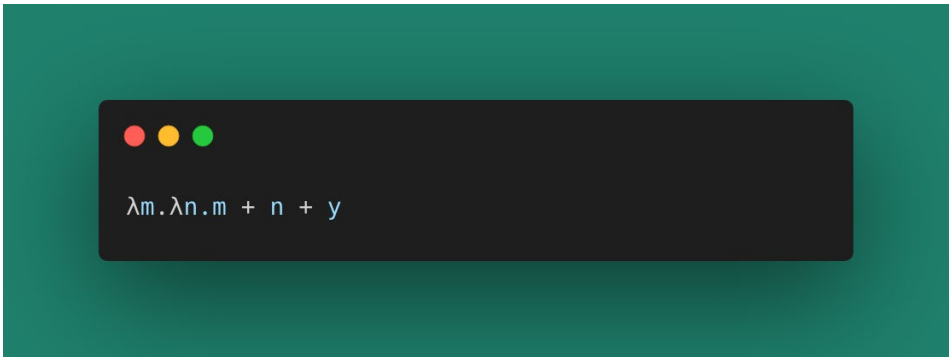


图 9

这里的 y 是一个没有绑定到参数位置的变量，被称为一个**自由变量**。

绑定变量和**自由变量**是函数的两种状态来源，一个可以被代换，另一个不能。实际程序中，通常把绑定变量实现为局部变量或者参数，自由变量实现为全局变量或者环境变量。

2.2.2 演算：代换和归约

演算分为 Alpha 代换和 Beta 归约。 前面章节我们实际上已经涉及这两个概念，下面来介绍一下它们。

Alpha 代换指的是变量的名称是不重要的，你可以写 λm.λn.m + n，也可以写 λx.λy.x + y，在演算过程中它们表示同一个函数。也就是说我们只**关心计算的形式**，而不关心细节用什么变量去实现。这方便我们不改变运算结果的前提下去修改变量命名，以方便在函数比较复杂的情况下进行化简操作。实际上，连整个 lambda 演算式的名字也是不重要的，我们只需要这种形式的计算，而不在乎这个形式的命名。

Beta 归约指的是如果你有一个**函数应用（函数调用）**，那么你可以对这个函数体中与标识符对应的部分做**代换（substitution）**，方式为使用参数（可能是另一个演算式）去替换标识符。听起来有点绕口，但是它实际上就是**函数调用的参数替换**。比如：

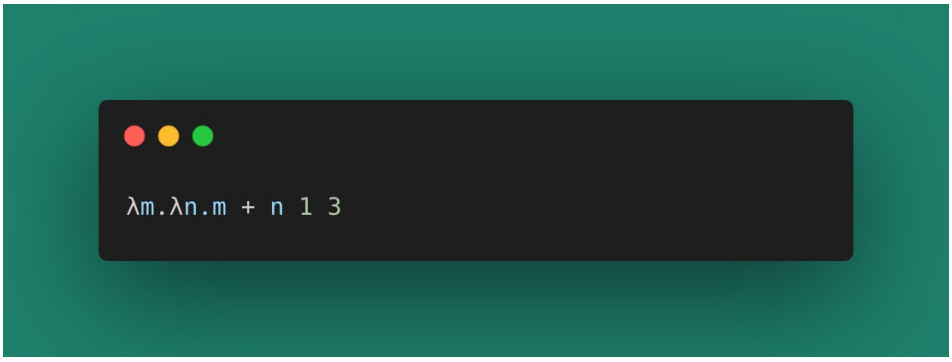


图 10

可以使用 1 替换 m，3 替换 n，那么整个表达式可以化简为 4。这也是函数式编程里面的引用透明性的由来。需要注意的是，这里的 1 和 3 表示表达式运算值，可以替换为其他表达式。比如把 1 **替换为（**λm.λn.m + n 1 3**）**，这里就需要做两次归约来得到下面的最终结果：

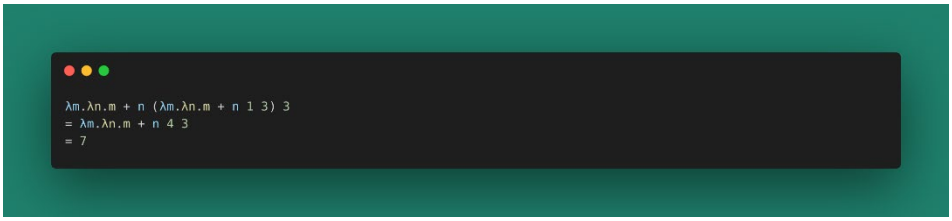


图 11

2.3 JavaScript 中的 λ 表达式：箭头函数

ECMAScript 2015 规范引入了箭头函数，它没有 **this**，没有 **arguments**。只能作为一个**表达式（expression）**而不能作为一个**声明式（statement）**，表达式产生一个箭头函数引用，该箭头函数引用仍然有 **name** 和 **length** 属性，分别表示箭头函数的名字、形参（parameters）长度。一个箭头函数就是一个单纯的运算式，箭头函数我们也可以称为 **lambda 函数**，它在书写形式上就像是一个 **λ 演算式**。

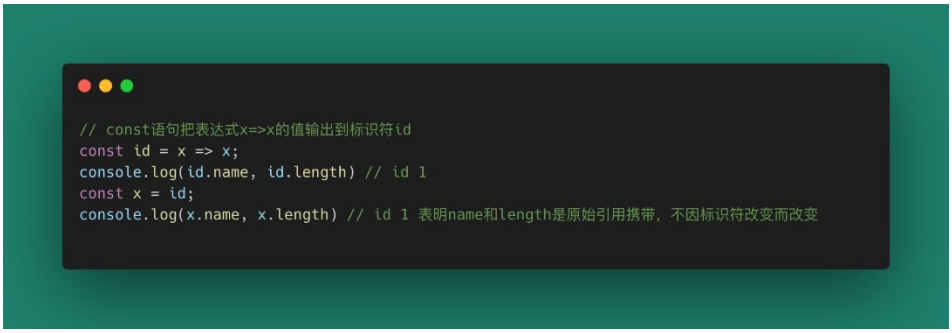


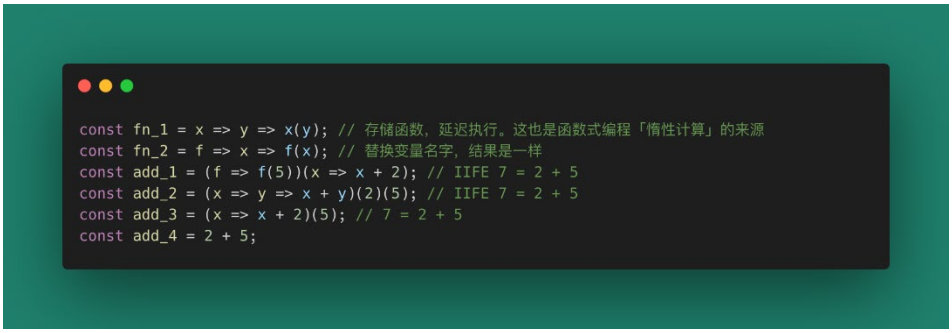
图 12

可以利用箭头函数做一些简单的运算，下例比较了四种箭头函数的使用方式：

```
// 加法
const add_1 = (x, y) => x + y;
const add_2 = x => x + y; // y来自外部(闭包引用外部参数)
const add_3 = x => y => x + y; // 可以用闭包串联参数(即柯里化)
const add_4 = b => a => a + b; // 参数命名和算式结果无关
```

图 13

这是直接针对数字（基本数据类型）的情况，如果是针对函数做运算（**引用数据类型**），事情就变得有趣起来了。我们看一下下面的示例：



```
const fn_1 = x => y => x(y); // 存储函数，延迟执行。这也是函数式编程「惰性计算」的来源
const fn_2 = f => x => f(x); // 替换变量名字，结果是一样
const add_1 = (f => f(5))(x => x + 2); // IIFE 7 = 2 + 5
const add_2 = (x => y => x + y)(2)(5); // IIFE 7 = 2 + 5
const add_3 = (x => x + 2)(5); // 7 = 2 + 5
const add_4 = 2 + 5;
```

图 14

fn_x 类型，表明我们可以利用函数内的函数，当函数被当作数据传递的时候，就可以对函数进行应用（apply），生成更高阶的操作。 并且 x => y => x(y) 可以有两种理解，一种是 x => y 函数传入 X => x(y)，另一种是 x 传入 y => x(y)。

add_x 类型表明，一个运算式可以有很多不同的路径来实现。

上面的 add_1/add_2/add_3 我们用到了 JavaScript 的立即运算表达式 IIFE。

λ **演算**是一种抽象的数学表达方式，我们不关心真实的运算情况，我们只关心这种**运算形式**。因此上一节的演算可以用 JavaScript 来模拟。下面我们来实现 λ **演算的 JavaScript 表示**。

```
// 定义自然数
const zero = f => x => f(x)
const one = f => x => f(f(x))
const two = f => x => f(f(f(x)))

// ...
// 为了表示函数累计的概念，引入两个辅助函数

// 同一个函数f对x（f f f f x...）执行n次
const times = n => f => new Array(n).fill(f).reduce((acc, f) => x => f(acc(x)));
const countTime = x => x + 1;

console.log('COUNT', times(8)(countTime)(0)) // COUNT 8

var SUCC = n => f => x => times(n)(f)(x)
var PLUS = m => n => f => x => times(m)(f)(times(n)(f)(x))

console.log('SUCC 4', SUCC(4)(countTime)(0)) // SUCC 4 4
console.log('PLUS 5 6', PLUS(5)(6)(countTime)(0)) // PLUS 5 6 11
```

图 15

我们把 λ **演算中**的 f 和 x 分别取为 countTime 和 x，代入运算就得到了我们的自然数。

这也说明了不管你使用**符号系统**还是 JavaScript **语言**，你想要表达的**自然数**是等价的。这也侧面说明 λ **演算**是一种**形式上的抽象（和具体语言表述无关的抽象表达）**。

2.4 函数式编程基础：函数的元、柯里化和 Point-Free

回到 JavaScript 本身，我们要探究**函数**本身能不能带给我们更多的东西？我们在 JavaScript 中有很多创建函数的方式：

```
// 函数声明
function log() {
    console.log(...arguments)
}
console.log(log.name) // log
console.log(log.length) // 0



// 函数表达式 logName可省略，不可被外部使用
const log = function logName () {
    // 可以使用logName
    console.log(...arguments)
}

console.log(log.name) // logName
console.log(log.length) // 0

// IIFE
// IIFE第一个括号是括号表达式，(expression)它产生一个值，比如var a = (1) // 1
// IIFE第二个括号是函数调用，值被传递到第一个表达式产生的函数上
(function IIFE() {
    console.log(...arguments)
})()

// 函数表达式-箭头函数 没有arguments
const arrowLog = (name) => console.log(name)
console.log(arrowLog.name) // arrowLog
console.log(arrowLog.length) // 1

// 运行时构造
const fn = new Function('a', 'b', 'c', `return a + b + c`)
console.log(fn.name) // anonymous
console.log(fn.length) // 3
```

图 16

虽然函数有这么多定义，但 function 关键字声明的函数带有 arguments 和 this 关键字，这让他们看起来更像是对象方法（method），而不是函数（function）。

况且 function 定义的函数大多数还能被构造（比如 new Array）。

接下来我们将只研究箭头函数，因为它更像是数学意义上的函数（仅执行计算过程）。

- 没有 arguments 和 this。
- 不可以被构造 new。

2.4.1 函数的元：完全调用和不完全调用

不论使用何种方式去构造一个函数，这个函数都有两个固定的信息可以获取：

- **name** 表示当前标识符指向的函数的名字。
- **length** 表示当前标识符指向的函数定义时的参数列表长度。

在数学上，我们定义 **f(x) = x** 是一个一元函数，而 **f(x, y) = x + y** 是一个二元函数。在 JavaScript 中我们可以使用函数定义时的 **length** 来定义它的元。

```
// 一元函数
const one = a => a;
// 二元函数
const two = (a, b) => a + b;
// 三元函数
const three = (a, b, c) => a + b + c;
```

图 17

定义**函数的元**的意义在于，我们可以对函数进行归类，并且可以明确一个函数需要的确切参数个数。函数的元在编译期（类型检查、重载）和运行时（异常处理、动态生成代码）都有重要作用。

如果我给你一个**二元函数**，你就知道需要传递两个参数。比如 + 就可以看成是一个二元函数，它的左边接受一个参数，右边接受一个参数，返回它们的和（或字符串连接）。

在一些其他语言中，+ 确实也是由抽象类来定义实现的，比如 Rust 语言的 **trait Add**。

但我们上面看到的 **λ 演算**，每个函数都只有一个元。为什么呢?

只有一个元的函数方便我们进行代数运算。λ 演算的参数列表使用 λx.λy.λz 的格

式进行分割，返回值一般都是函数，如果一个二元函数，调用时只使用了一个参数，则返回一个「不完全调用函数」。这里用三个例子解释“不完全调用”。

第一个，不完全调用，代换参数后产生了一个**不完全调用函数 λz.3 + z**。

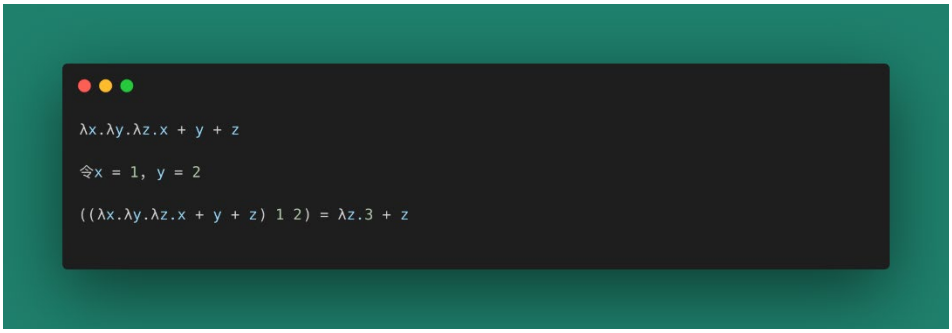


图 18

第二个，Haskell 代码，调用一个函数 **add**（类型为 **a -> a -> a**），得到另一个函数 **add 1**（类型为 **a -> a**）。

图 19

第三个，上一个例子的 JavaScript 版本。

图 20

“不完全调用”在 JavaScript 中也成立。实际上它就是 JavaScript 中闭包（Closure，上面我们已经提到过）产生的原因，一个函数还没有被销毁（调用没有完全结束），你可以在子环境内使用父环境的变量。

注意，上面我们使用到的是一元函数，如果使用三元函数来表示 addThree，那么函数一次性就调用完毕了，不会产生不完全调用。

函数的元还有一个著名的例子（面试题）：

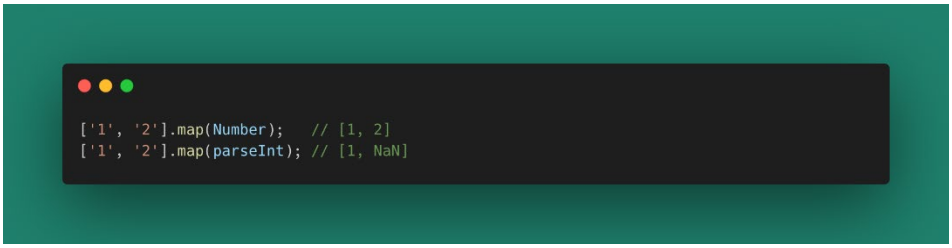


图 21

造成上述结果的原因就是，Number 是一元的，接受 map 第一个参数就转换得到返回值；而 parseInt 是二元的，第二个参数拿到进制为 1（map 函数为三元函数，第二个参数返回元素索引），无法输出正确的转换，只能返回 NaN。这个例子里面涉及到了一元、二元、三元函数，多一个元，函数体就多一个状态。如果世界上只有一元函数就好了！我们可以全通过一元函数和不完全调用来生成新的函数处理新的问题。

认识到函数是有元的，这是函数式编程的重要一步，多一个元就多一种复杂度。

2.4.2 柯里化函数：函数元降维技术

柯里化（curry）函数是一种把函数的元降维的技术，这个名词是为了纪念我们上文提到的数学家**阿隆佐 · 邱奇**。

首先，函数的几种写法是等价的（最终计算结果一致）。

```
const add = (a, b) => a + b
const add = a => b => a + b
```

图 22

这里列一个简单的把普通函数变为柯里化函数的方式（柯里化函数 Curry）。

```
const curry = function (fn) { // bind(绑定) fn
    let length = fn.length; // bind length
    let params = []; // bind params
    return function partial(x) {
        params.push(x); // use使用 params
        if (params.length === length) { // use length
            return fn(...params) // use fn
        } else {
            return partial
        }
    }
}

const curryAdd = curry((a, b, c) => a + b + c)

curryAdd(1)(2)(3) // 6
```

图 23

柯里化函数帮助我们把一个多元函数变成一个不完全调用，利用 Closure 的魔法，把

函数调用变成延迟的偏函数（不完全调用函数）调用。这在函数**组合、复用**等场景非常有用。比如：

```
// http.js
export default (url, params) => fetch(url, params)

// api.js
import api from './api.js'
export const fetchList = curry(api, '/api/getList')
export const fetchItem = curry(api, '/api/getItem')

// ui.js
async function UI() {
    const list = await fetchList({name: 'aa'})
    const all = list.forEach(async row => await fetchItem({id: row.id}))
    const res = Promise.all(all)
    return res;
}
```

图 24

虽然你可以用其他闭包方式来实现函数的延迟调用，但 Curry 函数绝对是其中形式最美的几种方式之一。

2.4.3 Point-Free | 无参风格：函数的高阶组合

函数式编程中有一种 **Point-Free** 风格，中文语境大概可以把 **point** 认为是参数点，对应 λ **演算**中的函数应用（Function Apply），或者 JavaScript 中的函数调用（Function Call），所以可以理解 **Point-Free 就指的是无参调用**。

来看一个日常的例子，把二进制数据转换为八进制数据：

```
var strNums = ['01', '10', '11', '1110']

strNums.map(x => parseInt(x, 2)).map(x => x.toString(8)) // ['1', '2', '3', '16']
```

图 25

这段代码运行起来没有问题，但我们为了处理这个转换，需要了解 **parseInt(x, 2)** 和 **toString(8)** 这两个函数（为什么有魔法数字 2 和魔法数字 8），并且要关心数据（函数类型 **a -> b**）在每个节点的形态（关心数据的流向）。有没有一种方式，可以让我们只关心入参和出参，不关心数据流动过程呢?

```
const toBinary = x => parseInt(x, 2); // 可以使用curry定义
const toStringOx = x => x.toString(8); // 可以使用curry定义
const pipe =  (...fns) =>  x => fns.reduce((acc, fn) => fn(acc), x);

// 数据流向没有显示出现
var strNums = ['01', '10', '11', '1110']
strNums.map(pipe(toBinary, toStringOx)) // ['1', '2', '3', '16']
```

图 26

上面的方法假设我们已经有了一些基础函数 **toBinary**（语义化，消除魔法数字 2）和 **toStringOx**（语义化，消除魔法数字 8），并且有一种方式（**pipe**）把基础函数组合（**Composition**）起来。如果用 **Typescript** 表示我们的数据流动就是：

```
type toBinary = (strNum: string) => number
type toStringOx = (num: number) => string

type pipe = (strNum: string) =>
    (handler1: toBinary, handler2: toStringOx) =>
    returnType<toStringOx>
```

图 27

用 Haskell 表示更简洁，后面都用 Haskell 类型表示方式，作为我们的符号语言。

```
[char] -> [int] -> [char]
```

图 28

值得注意的是，这里的 x-> [int] ->y 我们不用关心，因为 pipe(..) 函数帮我们处理了它们。pipe 就像一个盒子。

```
input ->  BOX -> output
```

图 29

BOX 内容不需要我们理解。而为了达成这个目的，我们需要做这些事：

- utils 一些特定的工具函数。

- **curry** 柯里化并使得函数可以被复用。
- **composition** 一个组合函数，像胶水一样粘合所有的操作。
- **name** 给每个函数定义一个见名知意的名字。

综上，Point-Free 风格是粘合一些基础函数，最终让我们的数据操作不再关心中间态的方式。这是函数式编程，或者说函数式编程语言中我们会一直遇到的风格，表明我们的基础函数是值得信赖的，我们仅关心数据转换这种形式，而不是过程。JavaScript 中有很多实现这种基础函数工具的库，最出名的是 Lodash。

可以说函数式编程范式就是在不停地组合函数。

2.5 函数式编程特性

说了这么久，都是在讲函数，那么究竟什么是函数式编程呢？在网上你可以看到很多定义，但大都离不开这些特性。

- **First Class** 函数：函数可以被应用，也可以被当作数据。
- **Pure** 纯函数，无副作用：任意时刻以相同参数调用函数任意次数得到的结果都一样。
- **Referential Transparency** 引用透明：可以被表达式替代。
- **Expression** 基于表达式：表达式可以被计算，促进数据流动，状态声明就像是一个暂停，好像数据到这里就会停滞了一下。
- **Immutable** 不可变性：参数不可被修改、变量不可被修改—宁可牺牲性能，也要产生新的数据（Rust 内存模型例外）。
- **High Order Function** 大量使用高阶函数：变量存储、闭包应用、函数高度可组合。
- **Curry** 柯里化：对函数进行降维，方便进行组合。
- **Composition** 函数组合：将多个单函数进行组合，像流水线一样工作。

另外还有一些特性，有的会提到，有的一笔带过，但实际也是一个特性（以 Haskell 为例）。

- **Type Inference** 类型推导：如果无法确定数据的类型，那函数怎么去组合？（常见，但非必需）
- **Lazy Evaluation** 惰性求值：函数天然就是一个执行环境，惰性求值是很自然的选择。
- **Side Effect** IO：一种类型，用于处理副作用。一个不能执行打印文字、修改文件等操作的程序，是没有意义的，总要有位置处理副作用。（边缘）

数学上，我们定义函数为集合 A 到集合 B 的映射。在函数式编程中，我们也是这么认为的。函数就是把数据从某种形态映射到另一种形态。注意理解“映射”，后面我们还会讲到。

图 30

2.5.1 First Class：函数也是一种数据

函数本身也是数据的一种，可以是参数，也可以是返回值。

图 31

通过这种方式，我们可以让函数作为一种可以保存状态的值进行流动，也可以充分利用不完全调用函数来进行函数组合。把函数作为数据，实际上就让我们有能力获取函数内部的状态，这样也产生了闭包。但函数式编程不强调状态，大部分情况下，我们的“状态”就是一个函数的元（我们从元获取外部状态）。

2.5.2 纯函数：无状态的世界

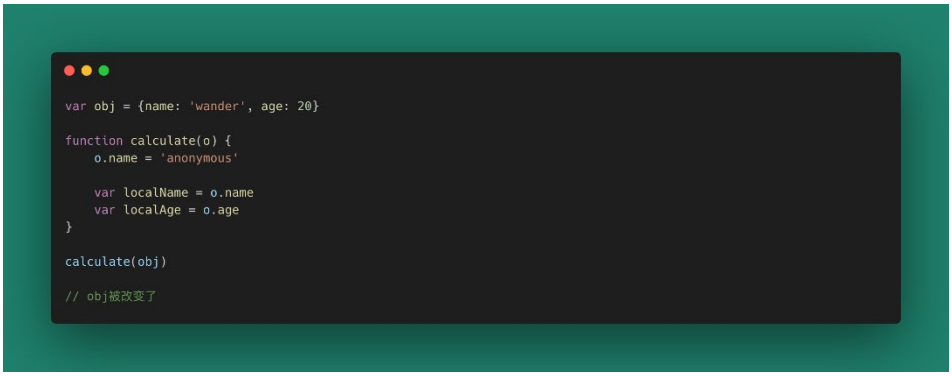
通常我们定义输入输出（IO）是不纯的，因为 IO 操作不仅操作了数据，还操作了这个数据范畴外部的世界，比如打印、播放声音、修改变量状态、网络请求等。这些操作并不是说对程序造成了破坏，相反，一个完整的程序一定是需要它们的，不然我们的所有计算都将毫无意义。

但纯函数是可预测的，引用透明的，我们希望代码中更多地出现纯函数式的代码，这样的代码可以被预测，可以被表达式替换，而更多地把 IO 操作放到一个统一的位置做处理。

```
const add = async  x => await fetch() + x

add(1).then(console.log)
```

图 32

这个 add 函数是不纯的，但我们把副作用都放到它里面了。任意使用这个 add 函数的位置，都将变成不纯的（就像是 async/await 的传递性一样）。需要说明的是抛开实际应用来谈论函数的纯粹性是毫无意义的，我们的程序需要和终端、网络等设备进行交互，不然一个计算的运行结果将毫无意义。但对于函数的元来说，这种纯粹性就很有意义，比如：



```
var obj = {name: 'wander', age: 20}

function calculate(o) {
    o.name = 'anonymous'

    var localName = o.name
    var localAge = o.age
}

calculate(obj)

// obj被改变了
```

图 33

当函数的元像上面那样是一个引用值，如果一个函数的调用不去控制自己的纯粹性，对别人来说，可能会造成毁灭性打击。因此我们需要对引用值特别小心。

```
var obj = {name: 'wander', age: 20}

function calculate(o) {
    var local = {...o, name: 'anonymous'}

    var localName = local.name
    var localAge = local.age
}

calculate(obj)

// obj没有改变
```

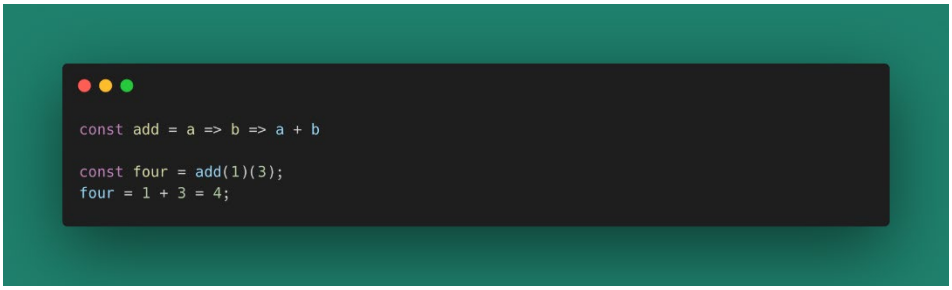
图 34

上面这种解构赋值的方式仅解决了第一层的引用值，很多其他情况下，我们要处理一个引用树、或者返回一个引用树，这需要更深层次的解引用操作。请小心对待你的引用。

函数的纯粹性要求我们时刻提醒自己降低状态数量，把变化留在函数外部。

2.5.3 引用透明：代换

通过表达式替代（也就是 λ 演算中讲的归约），可以得到最终数据形态。



```
const add = a => b => a + b

const four = add(1)(3);
four = 1 + 3 = 4;
```

图 35

也就是说，调用一个函数的位置，我们可以使用函数的调用结果来替代此函数调用，产生的结果不变。

一个引用透明的函数调用链永远是可以被合并式代换的。

2.5.4 不可变性：把简单留给自己

一个函数不应该去改变原有的引用值，避免对运算的其他部分造成影响。

图 36

一个充满变化的世界是混沌的，在函数式编程世界，我们需要强调参数和值的不可变性，甚至在很多时候我们可以为了不改变原来的引用值，牺牲性能以产生新的对象来进行运算。牺牲一部分性能来保证我们程序的每个部分都是可预测的，任意一个对象从创建到消失，它的值应该是固定的。

一个元如果是引用值，请使用一个副本（克隆、复制、替代等方式）来得到状态变更。

2.5.5 高阶：函数抽象和组合

JS 中用的最多的就是 Array 相关的高阶函数。实际上 Array 是一种 Monad（后面讲解）。

图 37

通过高阶函数传递和修改变量：

图 38

高阶函数实际上为我们提供了注入环境变量（或者说绑定环境变量）提供了更多可能。React 的高阶组件就从这个思想上借用而来。一个高阶函数就是使用或者产生另一个函数的函数。高阶函数是函数组合（Composition）的一种方式。

高阶函数让我们可以更好地组合业务。常见的高阶函数有：

- map
- compose
- fold
- pipe
- curry
- ….

2.5.6 惰性计算：降低运行时开销

惰性计算的含义就是在真正调用到的时候才执行，中间步骤不真实执行程序。这样可以让我们在运行时创建很多基础函数，但并不影响实际业务运行速度，唯有业务代码真实调用时才产生开销。

图 39

map(addOne) 并不会真实执行 +1，只有真实调用 exec 才执行。当然这个只是一个简单的例子，强大的惰性计算在函数式编程语言中还有很多其他例子。比如：

图 40

“无穷”只是一个字面定义，我们知道计算机是无法定义无穷的数据的，因此数据在 take 的时候才真实产生。

惰性计算让我们可以无限使用函数组合，在写这些函数组合的过程中并不产生调用。但这种形式，可能会有一个严重的问题，那就是产生一个非常长的调用栈，而虚拟机或者解释器的函数调用栈一般都是有上限的，比如 2000 个，超过这个限制，函数调用就会栈溢出。虽然函数调用栈过长会引起这个严重的问题，但这个问题其实不是函数式语言设计的逻辑问题，因为调用栈溢出的问题在任何设计不良的程序中都有可能出现，惰性计算只是利用了函数调用栈的特性，而不是一种缺陷。

记住，任何时候我们都可以重构代码，通过良好的设计来解决栈溢出的问题。

2.5.7 类型推导

当前的 JS 有 TypeScript 的加持，也可以算是有类型推导了。

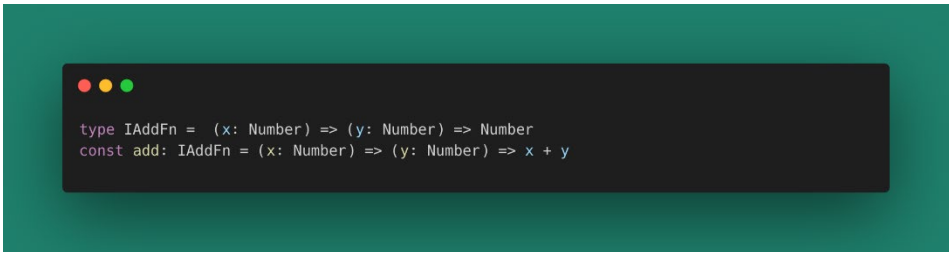


图 41

没有类型推导的函数式编程，在使用的时候会很不方便，所有的工具函数都要查表查示例，开发中效率会比较低下，也容易造成错误。

但并不是说一门函数式语言必须要类型推导，这不是强制的。像 Lisp 就没有强制类型推导，JavaScript 也没有强制的类型推导，这不影响他们的成功。只是说，有了类型推导，我们的编译器可以在编译器期间提前捕获错误，甚至在编译之前，写代码的时候就可以发现错误。类型推导是一些语言强调的优秀特性，它确实可以帮助我们提前发现更多的代码问题。像 Rust、Haskell 等。

2.5.8 其他

你现在也可以总结一些其他的风格或者定义。比如强调函数的组合、强调 Point-Free 的风格等等。

图 42

3. 小结

函数式有很多基础的特性，熟练地使用这些特性，并加以巧妙地组合，就形成了我们的“函数式编程范式”。这些基础特性让我们对待一个 function，更多地看作**函数**，而不是一个**方法**。在许多场景下，使用这种范式进行编程，就像是在做数学推导（或者说是类型推导），它让我们像学习数学一样，把一个个复杂的问题简单化，再进行累加 / 累积，从而得到结果。

同时，函数式编程还有一块大的领域需要进入，即副作用处理。不处理副作用的程序是毫无意义的（仅在内存中进行计算），下篇我们将深入函数式编程的应用，为我们的工程应用发掘出更多的特性。

4. 作者简介

俊杰，美团到家研发平台 / 医药技术部前端工程师。

深入理解函数式编程（下）

作者：俊杰

1. 前文回顾

在上篇中，我们分析了函数式编程的起源和基本特性，并通过每一个特性的示例来演示这种特性的实际效果。首先，函数式编程起源于数理逻辑，起源于 λ **演算**，这是一种演算法，它定义一些基础的数据结构，然后通过归约和代换来实现更复杂的数据结构，而函数本身也是它的一种数据。其次，我们探讨了很多函数式编程的特性，比如：

- First Class
- 纯函数
- 引用透明
- 表达式
- 高阶函数
- 柯里化
- 函数组合
- point-free
- …

但我们也指出了一个实际问题：不能处理副作用的程序是毫无意义的。我们的计算机程序随时都在产生副作用。我们程序里面有大量的网络请求、多媒体输入输出、内部状态、全局状态等，甚至在提倡“碳中和”的今天，电脑的发热量也是一个不容小觑的副作用。那么我们应该如何处理这些问题呢?

2. 本文简介

本文通过深入函数式编程的副作用处理及实际应用场景，提供一个学习和使用函数式编程的视角给读者。一方面，这种副作用管理方式是一种高级的抽象形式，不易理解；另一方面，我们在学习和使用函数式编程的过程中，几乎都会遇到类似的副作用问题需要解决，能否解决这个问题也决定了一门函数式编程语言最终是否能走上成功。

本文主要分为三个部分：

- 副作用处理方式
- 函数式编程的应用
- 函数式编程的优缺点比较

3. 副作用处理：单子 Monad，一种不可避免的抽象

上面说的，都是最基础的 JavaScript 概念 + 函数式编程概念。但我们还留了一个“坑”。

如何去处理 IO 操作?

我们的代码经常在和副作用打交道，如果要满足纯函数的要求，几乎连一个需求都完成不了。不用急，我们来看一下 React Hooks。React Hooks 的设计是很巧妙的，以 useEffect 为例：

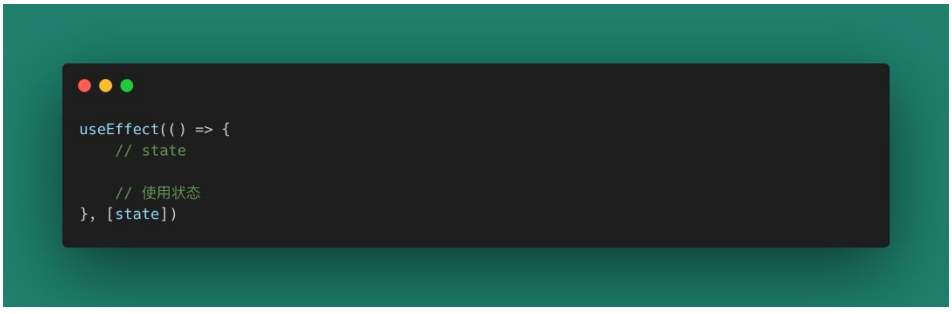


图 43

在函数组件中，useState 用来产生状态，在使用 useEffect 的时候，我们需要挂载这个 state 到第二个参数，而第一个参数给到的运行函数在 state 变更的时候被调用，被调用时得到最新的 state。

这里面有一个状态转换：

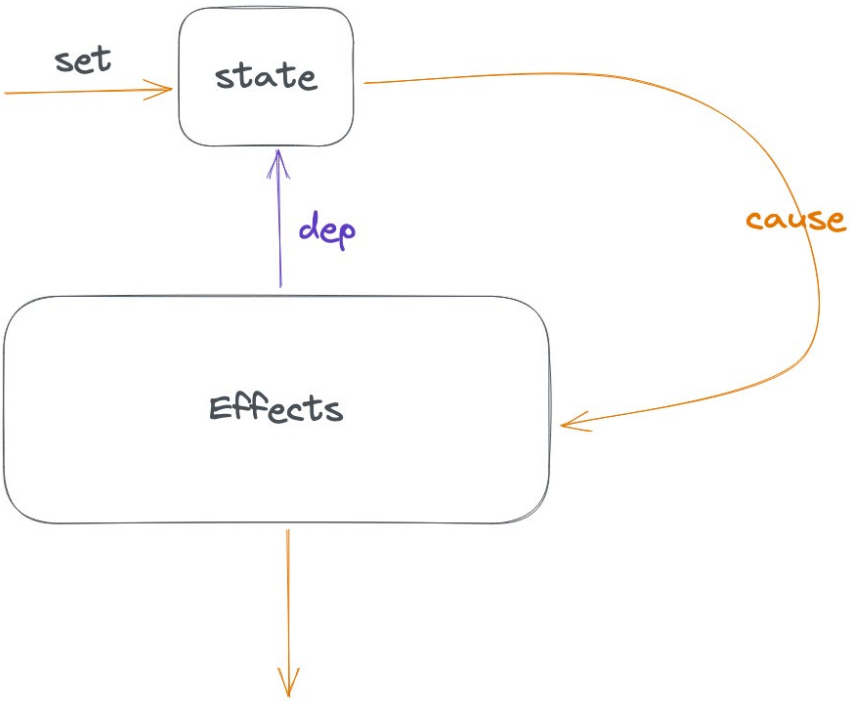


图 44

React Hooks 给我们的启发是，副作用都被放到一个状态节点里面去被动触发，行程一个单向的数据流动。而实际上，函数式编程语言确实也是这么做的，把副作用包裹到一个特殊的函数里面。

如果一个函数既包含了我们的值，又封装了值的统一操作，使得我们可以在它限定的范围内进行任意运算，那么，我们称这种函数类型为 Monad。Monad 是一种高级别的思维抽象。

3.1 什么是 Monad？

先思考一个问题，下面两个定义有什么区别?

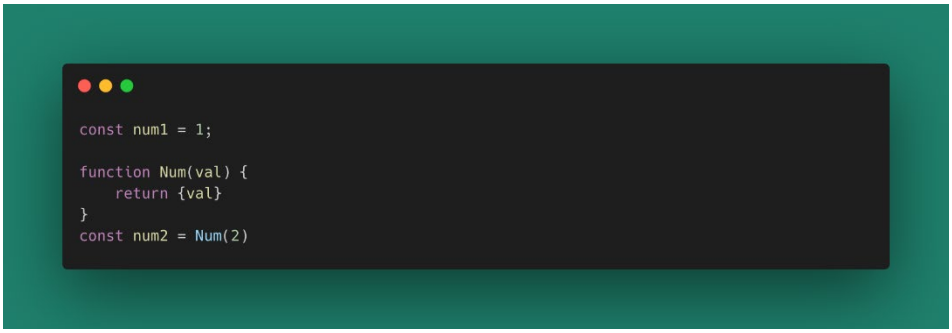
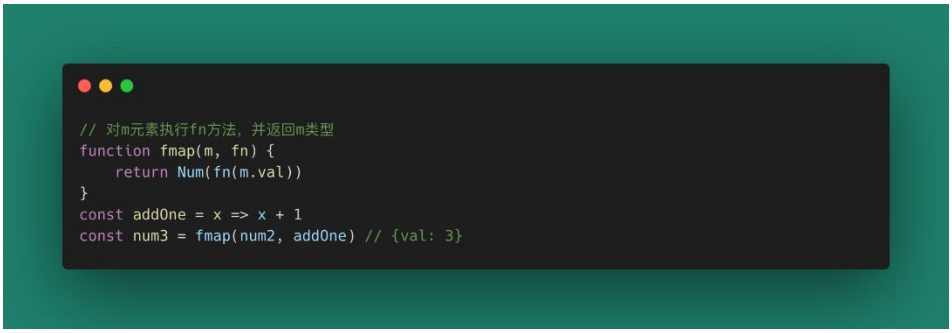


图 45

num1 是数字类型，而 num2 是对象类型，这是一个直观的区别。

不过，不仅仅如此。利用类型，我们可以做更多的事。因为作为数字的 num1 是支持加减乘除运算的，而 num2 却不行，必须要把它视为一个对象 {val: 2}，并通过属性访问符 num2.val 才能进行计算 num2.val + 2。但我们知道，函数式编程是不能改变状态的，现在为了计算 num2.val 被改变了，这不是我们期望的，并且我们使用属性操作符去读数据，更像是在操作对象，而不是操作函数，这与我们的初衷有所背离。

现在我们把 num2 当作一个独立的数据，并假设存在一个方法 fmap 可以操作这个数据，可能是这样的。



```
// 对m元素执行fn方法，并返回m类型
function fmap(m, fn) {
    return Num(fn(m.val))
}
const addOne = x => x + 1
const num3 = fmap(num2, addOne) // {val: 3}
```

图 46

得到的还是对象，但操作通过一个纯函数 addOne 去实现了。

上面这个例子里面的 Num，实际上就是一个最简单的 Monad，而 fmap 是属于 Functor（**函子**）的概念。我们说函数就是从一个数据到另一个数据的映射，这里的 fmap 就是一个映射函数，在范畴论里面叫做**态射**（后面讲解）。

由于有一个包裹的过程，很多人会把 Monad 看作是一个盒子类型。但 Monad 不仅是一个盒子的概念，它还需要满足一些特定的运算规律（后面涉及）。

但是我们直接使用数字的加减乘除不行吗？为什么一定要 Monad 类型？

首先，fmap 的目的是把数据从一个类型映射到另一个类型，而 JavaScript 里面的 map 函数实际上就是这个功能。

```
[1,2,3].map(v => v * 2)
```

图 47

我们可以认为 Array 就是一个 Monad 实现，map 把 Array< T > 类型映射到 Array< K > 类型，操作仍然在数组范畴，数组的值被映射为新的值。如果用 TypeScript 来表示，会不会更清晰一点？

```
type MapType<T, K> = (a: T) => K

interface Functor<T> {
    fmap: <K>(fn: MapType<T, K>) => Monad<K>
}

class Monad<T> implements Functor<T> {
    private _value: T;
    constructor(value: T) {
        this._value = value;
    }
    fmap<K>(fn: MapType<T, K>){
       return new Monad<K>(fn(this._value))
    }
}

const num1 = new Monad(1);
const num2 = num1.fmap((x => x + 1))
```

图 48

看起来 **Monad** 只是一个实现了 **fmap** 的对象（**Functor** 类型，mappable 接口）而已。但 **Monad** 类型不仅是一个 **Functor**，它还有很多其他的工具函数，比如：

- bind 函数
- flatMap 函数
- liftM 函数

这些概念在学习 Haskell 时可以遇到，本文不作过多提及。这些额外的函数可以帮助我们操作被封装起来的值。

3.2 范畴、群、幺半群

范畴论是一种研究抽象数学形式的科学，它把我们的数学世界抽象为两个概念：

- 对象
- 态射

为什么说这是一种**形式**上的抽象呢？因为很多数学的概念都可以被这种形式所描述，

比如集合，对集合范畴来说，一个集合就是一个范畴对象，从集合 A 到集合 B 的映射就是集合的态射，再细化一点，整数集合到整数集合的加减乘操作构成了整数集合的态射（除法会产生整数集合无法表示的数字，因此这里排除了除法）。又比如，三角形可以被代数表示，也可以用几何表示、向量表示，从代数表示到几何表示的运算就可以视为三角形范畴的一种态射。

总之，**对象描述了一个范畴中的元素，而态射描述了针对这些元素的操作**。范畴论不仅可以应用到数学科学里面，在其他科学里面也有一些应用，实际上，范畴论就是我们描述客观世界的一种方式（抽象形式）。

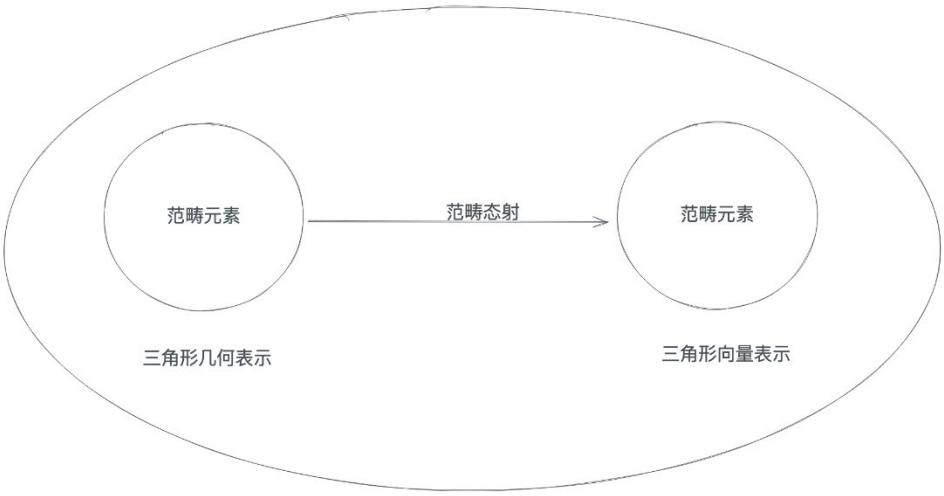


图 49

相对应的，**函子就是描述一个范畴对象和另一个范畴对象间关系的态射**，具体到编程语言中，函子是一个帮助我们映射一个范畴元素（比如 Monad）到另一个范畴元素的函数。

群论（Group）研究的是**群**这种代数结构，怎么去理解群呢？比如一个三角形有三个顶点 A/B/C，那么我们可以表示一个三角形为 ABC 或者 ACB，三角形还是这个三角形，但是从 ABC 到 ACB 一定是经过了某种变换。这就像范畴论，三角形的表示

是范畴对象，而一个三角形的表示变换到另一个形式，就是范畴的态射。而我们说这些三角形表示方式的集合为一个**群**。**群论主要是研究变换关系**，群又可以分为很多种类，也有很多规律特性，这不在本文研究范围之内，读者可以自行学习相关内容。

科学解释一个 Monad 为自函子范畴上的幺半群。如果没有学习群论和范畴论的话，我们是很难理解这个解释的。

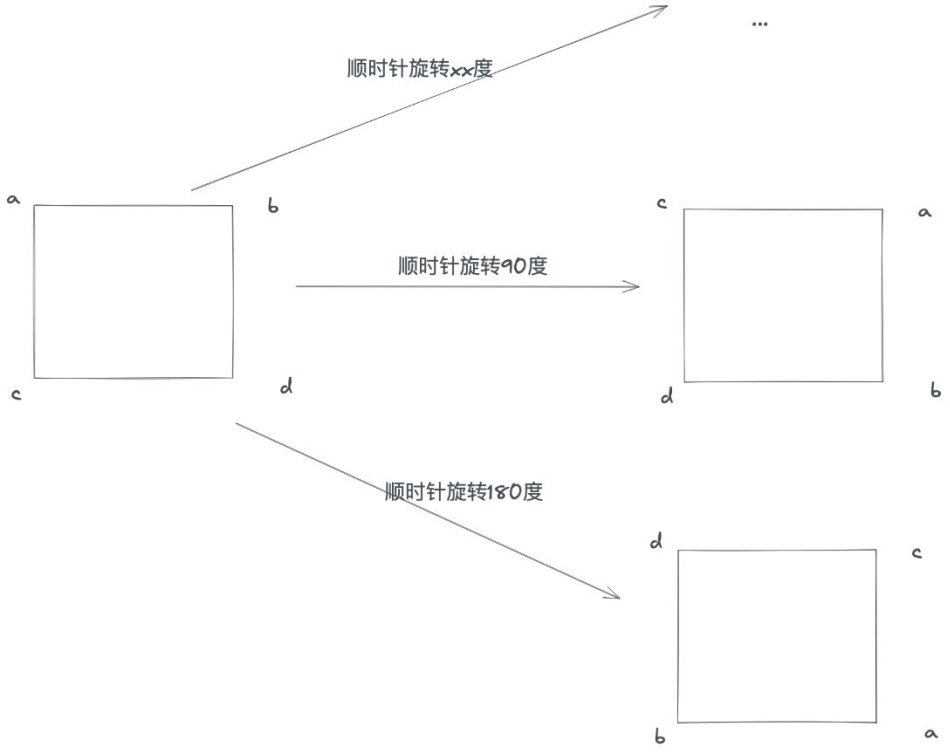


图 50

简单来说先固定一个正方形 abcd，它和它的几何变换方式（旋转 / 逆时针旋转 / 对称 / 中心对称等）形成的其他正方形一起构成一个群。从这个角度来说，群研究的事物是同一类，只是性质稍有不一样（态射后）。

另外一个理解群的概念就是**自然数（构成一个群）和加法（群的二元运算，且满足结合律，半群）**。

图 51

到此，我们可以理解 Monad 为：

1. 满足自函子运算（从 A 范畴态射到 A 范畴，fmap 是在自己空间做映射）。
2. 满足含幺半群的结合律。

很多函数式编程里面都会实现一个 Identity 函数，实际就是一个幺元素。比如 JavaScript 中对 Just 满足二元结合律可以这么操作：

```
function Just(val) {
    return {
        val,
        fmap(fn) {
            return Just(fn(val))
        }
    }
}
function Identity (x) {
        return x
}

// Just(1).fmap(Identity) =相等= Just(Identity(1))
```

图 52

3.3 Monad 范畴：定律、折叠和链

我们要在一个更大的空间上讨论这个范畴对象（Monad）。就像 Number 封装了数字类型，Monad 也封装了一些类型。

```
// Just是一个自函子
function Just(__value) {
    return {
        fmap(fn) {
            return Just(fn(__value)) // 态射到Just范畴，自函子
        },
        __inspect() {
            return __value
        }
    }
}
```

图 53

Monad 需要满足一些定律：

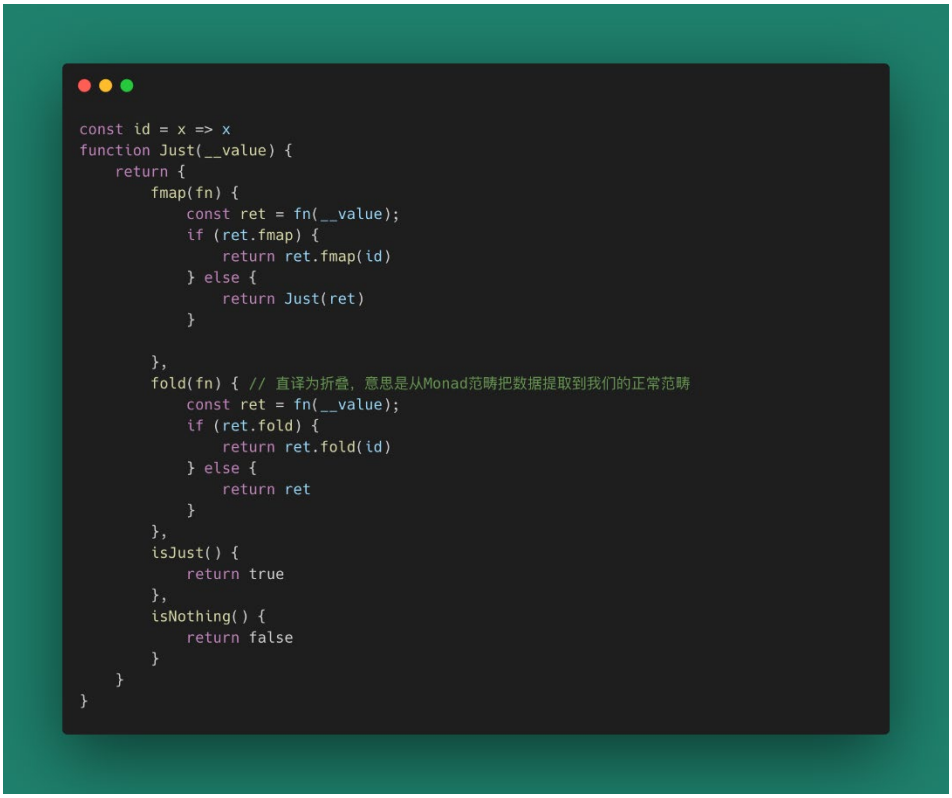
- **结合律**：比如 a · b · c = a · (b · c)。
- **幺元**：比如 a · e = e · a = a。

一旦定义了 Monad 为一类对象，fmap 为针对这种对象的操作，那么定律我们可以很容易证明：

```
const id = x => x
function Just(__value) {
    return {
        fmap(fn) {
            const ret = fn(__value);
            if (ret.fmap) {
                return ret.fmap(id)
            } else {
                return Just(ret)
            }

        },
    }
}

const addOne = x => Just(x + 1)
const multThree = x => Just(x * 3)

// 结合律
Just(1).fmap(addOne).fmap(multThree) // Just(6)
Just(1).fmap(x => addOne(x).fmap(multThree)) // Just(6)

// 幺元，Just本身作为幺元 类似const id = x => x函数
Just(1).fmap(addOne) // Just(2)
Just(2).fmap(Just) // Just(2)
const identity = x => x
Just(1).fmap(identity) // Just(1)
Just(identity(1)) // Just(1)
```

图 54

我们可以通过 **Monad Just** 上挂载的操作来对数据进行计算，这些运算是限定在了 **Just** 上的，也就是说你只能得到 **Just(..)** 类型。要获取原始数据，可以基于这个定义一个 fold 方法。



```
const id = x => x
function Just(__value) {
    return {
        fmap(fn) {
            const ret = fn(__value);
            if (ret.fmap) {
                return ret.fmap(id)
            } else {
                return Just(ret)
            }

        },
        fold(fn) { // 直译为折叠，意思是从Monad范畴把数据提取到我们的正常范畴
            const ret = fn(__value);
            if (ret.fold) {
                return ret.fold(id)
            } else {
                return ret
            }
        },
        isJust() {
            return true
        },
        isNothing() {
            return false
        }
    }
}
```

图 55

fold（折叠，对应能力我们称为 foldable）的意义在于你可以将数据从一个特定范畴映射到你的常用范畴，比如面向对象语言的 toString 方法，就是把数据从对象域转换到字符串域。

JavaScript 中的 Array.prototype.reduce 其实就是一个 fold 函数，它把数据从 Array 范畴映射到其他范畴。

一旦数据类型被我们锁定在了 Monad 空间（范畴），那我们就可以在这个范畴内连续调用 fmap（或者其他这个空间的函数）来进行值操作，这样我们就可以**链式处理**我们的数据。

```
const append = tail => head => head + tail
Just('a').fmap(append('b')).fmap(append('c')) // Just('abc')
```

图 56

3.4 Maybe 和 Either

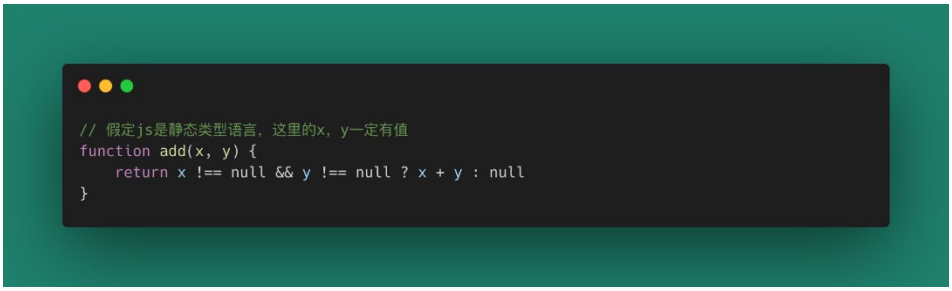
有了 Just 的概念，我们再来学习一些新的 Monad 概念。比如 Nothing。

```
function Nothing(val) {
    return {
        fmap(fn) {
            return Nothing()
        },
        fold(fn) {
            return fn(null)
        },
        isJust() {
            return false
        },
        isNothing() {
            return true
        }
    }
}
```

图 57

Nothing 表示在 Monad 范畴上没有的值。和 Just 一起正好描述了所有的数据情况，合称为 Maybe，我们的 Maybe Monad 要么是 Just，要么是 Nothing。这有什么意义呢?

其实这就是模拟了其他范畴内的“有”和“无”的概念，方便我们模拟其他编程范式的空值操作。比如：



```
// 假定js是静态类型语言，这里的x，y一定有值
function add(x, y) {
    return x !== null && y !== null ? x + y : null
}
```

图 58

这种情况下我们需要去判断 x 和 y 是否为空。在 Monad 空间中，这种情况就很好表示：

```
function Maybe(val) {
    return {
        fmap(fn) {
            if (val === null) {
                return Nothing()
            } else {
                return Just(val).fmap(fn)
            }
        },
        fold(fn) {
            return fn(val)
        }
    }
}

const addM = x => y => x.fmap(i => y.fmap(j => i + j))

addM(Maybe(1))(Maybe(2)) // Just(3)
addM(Maybe(null))(Maybe(3)) // Nothing
```

图 59

我们在 Monad 空间中消除了烦人的 !== null 判断，甚至消除了三元运算符。一切都只有函数。实际使用中一个 Maybe 要么是 Just 要么是 Nothing。因此，这里用 Maybe(..) 构造可能让我们难以理解。

如果非要理解的话，可以理解 Maybe 为 Nothing 和 Just 的抽象类，Just 和

Nothing 构成这个抽象类的两个实现。实际在函数式编程语言实现中，Maybe 确实只是一个类型（称为代数类型），具体的一个值有具体类型 Just 或 Nothing，就像数字可以分为有理数和无理数一样。

除了这种值存在与否的判断，我们的程序还有一些分支结构的方式，因此我们来看一下在 Monad 空间中，分支情况怎么去模拟?

图 60

假设我们有一个代数类型 Either，Left 和 Right 分别表示当数据为错误和数据为正确情况下的逻辑。

```
// 这次我们用常见的方式定义
function Either() {}
Either.from = function (val, [left, right]) {
    if (val.fmap) {
        if (val.isNothing()) {
            return Just(left)
        } else {
            return val.fmap(x => x ? right : left)
        }
    } else {
        return Just(val ? right : left)
    }

}

const addOne = x => x + 1
Either.from(true, [1, 2]).fmap(addOne) // Just(3)
Either.from(false, [1, 2]).fmap(addOne) // Just(2)
Either.from(Just(true), [1, 2]).fmap(addOne) // Just(3)
Either.from(Just(false), [1, 2]).fmap(addOne) // Just(2)
Either.from(Nothing(), [1, 2]).fmap(addOne) // Just(2)
```

图 61

这样，我们就可以使用“函数”来替代分支了。这里的 Either 实现比较粗糙，因为 Either 类型应该只在 Monad 空间。这里加入了布尔常量的判断，目的是好理解一些。其他的编程语言特性，在函数式编程中也能找到对应的影子，比如循环结构，我们往往使用函数递归来实现。

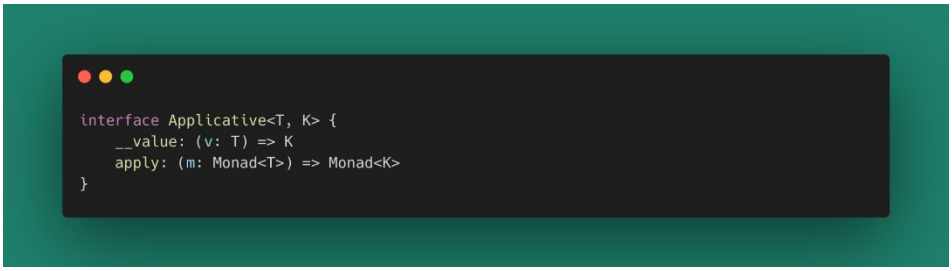
3.5 IO 的处理方式

终于到 IO 了，如果不能处理好 IO，我们的程序是不健全的。到目前为止，我们的 Monad 都是针对数据的。这句话对也不对，因为函数也是一种数据（函数是第一公民）。我们先让 Monad Just 能存储函数。

```
function Just(__value) {
    return {
        fmap(fn) {
            const ret = fn(__value);
            if (ret.fmap) {
                return ret.fmap(id)
            } else {
                return Just(ret)
            }

        },
        fold(fn) {
            const ret = fn(__value);
            if (ret.fold) {
                return ret.fold(id)
            } else {
                return ret
            }
        },
        isJust() {
            return true
        },
        isNothing() {
            return false
        },
        /** 增加方法 */
        apply(monad) {
            if (typeof __value === 'function') {
                return monad.fmap(__value)
            } else {
                return Nothing()
            }
        }
    }
}
```

图 62

你可以想象为 **Just** 增加了一个抽象类实现，这个抽象类为：



```
interface Applicative<T, K> {
    __value: (v: T) => K
    apply: (m: Monad<T>) => Monad<K>
}
```

图 63

这个抽象类我们称为“应用函子”，它可以保存一个函数作为内部值，并且使用 **apply** 方法可以把这个函数作用到另一个 **Monad** 上。到这里，我们完全可以把 Monad 之间的各种操作（接口，比如 **fmap** 和 **apply**）视为契约，也就是数学上的态射。

现在，如果我们有一个单子叫 **IO**，并且它有如下表现：

```
function IO(fn) {
    return {
        fmap(io) {
            return IO(() => io.run(fn))
        },
        fold() {
            return fn
        },
        apply(io) {
            return io.run(fn)
        },
        run(next = () => {}) {
            next(fn())
        }
    }
}

const one = IO(() => console.log(1))
const two = IO(() => console.log(2))

two.fmap(one).run()
// 1
// 2
```

图 64

我们把这种类型的 **Monad** 称为 **IO**，我们在 **IO** 中处理打印（副作用）。你可以把之前我们学习到的类型合并一下，得到一个示例：

```
function main() {
    const addOne = x => x + 1; // 纯函数
    const one = Just(1);  // monad

    const main = IO(() => {
        const three = Just(addOne).apply(one.fmap(addOne))
        three.fold(console.log) // 3
    })

    main.run();
}

main()
```

图 65

通常一个程序会有一个主入口函数 main，这个 main 函数返回值类型是一个 IO，我们的副作用现在全在 IO 这个范畴下运行，而其他操作，都可以保持纯净（类型运算）。

IO 类型让我们可以在 Monad 空间处理那些烦人的副作用，这个 **Monad** 类型和 **Promise**（限定副作用到 Promise 域处理，可链式调用，可用 then 折叠和映射）很像。

4. 函数式编程的应用

除了上面我们提到的一些示例，函数式编程可以应用到更广的业务代码开发中，用来替代我们的一些基础业务代码。这里举几个例子。

4.1 设计一个请求模块

```
// 假设我们有一个基础请求函数
// 我们要开发我们每个业务模块的代码
function req(url, params, options) {
    // ...
}

// module.js
// 1. 不完全调用 划分模块
const myReq = options => (url, params) => req(url, params, options)
const moduleOne = myReq({module: 'one'})
const moduleTwo = myReq({module: 'two'})

// api.js
// 2. 不完全调用 划分基础请求
const getList = partial(moduleOne, '/api/get')
const update = partial(moduleOne, '/api/update')

// validate.js
// 3. 分支检查 either(Maybe, left, right) 如果Maybe为Just，调用right，如果Maybe为Nothing，调用left
const checkNull = (fn) => (params) => either(Maybe.from(params), () => Promise.resolve('参数不能为空'), fn)
const checkId = (fn) => (params) => either(Maybe.from(params?.id), () => Promise.resolve('Id不能为空'), fn)
const toGetList = checkId(checkNull(getList))
const toUpdate = checkId(checkNull(update))

// business.js
// 4. 业务代码
toGetList(params)
toUpdate(params)
```

图 66

用这种方式构建的模块，组合和复用性很强，你也可以利用 lodash 的其他库对 req 做一个其他改造。我们调用业务代码的时候只管传递 params，分支校验和错误检查就教给 validate.js 里面的高阶函数就好了。

4.2 设计一个输入框

```
// 假设你有这么一个已知函数，可以用来处理dom的监听和取消监听
function listen(el, type, callback) {
    const dom = document.querySelector(el)
    if (!dom) {
        return
    }

    dom.addEventListener(type, callback)

    return () => dom.removeEventListener(type, callback)
}

// 1. 反转参数列表
const reverseListen = reverse(listen)

// 2. 不完全调用 产生一个处理input的高阶函数
const listenInput = callback => partial(reverseListen, callback, 'input')

// 3. debounce
const callback = e => fetch('/api/xxx', {input: e.target.value})
const debounced = debounce(callback, 200)

// 4. 监听
const subscribe = listenInput(debounced)
const unsubscribe = subscribe('#input')

// 5. 取消监听
if (unsubscribe) {
    unsubscribe()
}
```

图 67

这个例子也是来源于前端常见的场景。我们使用函数式编程的思想，把多个看似不相关的函数进行组合，得到了业务需要的 subscribe 函数，但同时，上面的任意一个函数都可以被用于其他功能组合。比如 callback 函数可以直接给 dom 回调，listenInput 可以用于任意一个 dom。

这种通过高阶组件不停组合得到最终结果的方式，我们可以认为就是函数式的。（尽管它没有像上一个例子一样引入 IO/Monad 等概念）

4.3 超长文本省略：Ramdajs 为例

```
// 模拟接口请求
const getList = () => Promise.resolve(['12345', '1234567890000'])
// 比较函数
const gt10 = R.gt(R.__, 10)
// 过滤函数
const pickLengthGt10 = R.propSatisfies(gt10, 'length')
// 业务逻辑处理
const appendEllipsis = R.pipe(
    R.take(10),
    R.append('…'),
    R.join('')
  )
const truncate = R.when(pickLengthGt10, appendEllipsis);


getList()
.then(R.map(truncate, R.__))
.then(console.log)
// ["12345","1234567890…"]
```

图 68

这个也是常见的前端场景，当文本长度大于 X 时，显示省略号，这个实现使用 Ramdajs。这个过程中你就像是在搭积木，很容易就把业务给“搭建”完成了。

5. 函数式编程库、语言

函数式编程的库可以学习：

- Ramda.js：函数式编程库
- lodash.js：函数工具
- immutable.js：数据不可变
- rx.js：响应式编程
- partial.lenses：函数工具
- monio.js：函数式编程工具库 /IO 库
- …

你可以结合起来使用。下面是 Ramda.js 示例：

```
// Ramda.js R表示Ramda函数库
const makeQuery = email => ({ query: { email }});
const fetchMember = request =>
  Promise.resolve({ firstName: 'Bob', lastName: 'Loblaw', id: 42 });

// 类型定义 getMemberName :: String -> Promise ({ firstName, lastName })
const getMemberName = R.pipe( // 函数组合
  makeQuery,
  fetchMember,
  R.andThen(R.pick(['firstName', 'lastName'])) // 偏函数应用
);

getMemberName('bob@gmail.com').then(console.log);
```

图片 69

而纯函数式语言，有很多：

- Lisp 代表软件 emacs…
- Haskell 代表软件 pandoc…
- Ocaml …
- …

6. 总结

函数式编程并不是什么“黑科技”，它已经存在的时间甚至比面向对象编程更久远。希望本文能帮助大家理解什么是函数式编程。

现在我们来回顾先览，实际上，函数式编程也是程序实现方式的一种，它和面向对象是殊途同归的。在函数式语言中，我们要构建一个个小的基础函数，并通过一些通用的流程把他们粘合起来。举个例子，面向对象里面的继承，我在函数式编程中可以使用组合 compose 或者高阶函数 hoc 来实现。

尽管在实现上是等价的，但和面向对象的编程范式对比，函数式编程有很多优点值得大家去尝试。

6.1 优点

除了上面提到的风格和特性之外，函数式编程相对其他编程范式，有很多优点：

- **函数纯净** 程序有更少的状态量，编码心智负担更小。随着状态量的增加，某些编程范式构建的软件库代码复杂度可能呈几何增长，而函数式编程的状态量都收敛了，对软件复杂度带来的影响更小。
- **引用透明性** 可以让你在不影响其他功能的前提下，升级某一个特定功能（一个对象的引用需要改动的话，可能牵一发而动全身）。
- **高度可组合** 函数之间复用方便（需要关注的状态量更少），函数的功能升级改造也更容易（高阶组件）。
- **副作用隔离** 所有的状态量被收敛到一个盒子（函数）里面处理，关注点更加集中。
- **代码简洁 / 流程更清晰** 通常函数式编程风格的程序，代码量比其他编程风格的少很多，这得益于函数的高度可组合性以及大量的完善的基础函数，简洁性也使得代码更容易维护。
- **语义化** 一个个小的函数分别完成一种小的功能，当你需要组合上层能力的时候，基本可以按照函数语义来进行快速组合。
- **惰性计算** 被组合的函数只会生成一个更高阶的函数，最后调用时数据才会在函数之间流动。
- **跨语言统一性** 不同的语言，似乎都遵从类似的函数式编程范式，比如 Java 8 的 lambda 表达式，Rust 的 collection、匿名函数；而面向对象的实现，不同语言可能千差万别，函数式编程的统一性让你可以舒服地跨语言开发。
- **关键领域应用** 因为函数式编程状态少、代码简洁等特点，使得它在交互复杂、安全性要求高的领域有重要的应用，像 Lisp 和 Haskell 就是因上一波人工智能热而火起来的，后来也在一些特殊的领域（银行、水利、航空航天等）得到了较大规模的应用。
- …

6.2 不足

当然，函数式编程也存在一些不足之处：

- **陡峭的学习曲线** 面向对象和命令式编程范式都是贴近我们的日常习惯的方式，而函数式编程要更抽象一些，要想更好地管理副作用，你可能需要学习很多新的概念（响应式、Monad 等），这些概念入门很难，而且是一个长期积累的过程。
- **可能的调用栈溢出问题** 惰性计算在一些电脑或特种程序架构上可能有函数调用栈错误（超长调用链、超长递归），另外许多函数式编程语言需要编译器支持尾递归优化（优化为循环迭代）以得到更好的性能。
- **额外的抽象负担** 当程序有大量可变状态、副作用时，用函数式编程可能造成额外的抽象负担，项目开发周期可能会延长，这时可能用其他抽象方式更好（比如 OOP）。
- **数据不变性的问题** 为了数据不变，运行时可能会构建生成大量的数据副本，造成时间和空间消耗更大，拖慢性能；同时数据不可变性可能会造成构建一些基础数据结构的时候语法不简洁，性能也更差（比如 LinkedList、HashMap 等数据结构）。
- **语义化的问题** 往往为了开发一个功能，去造许多的基础函数，大量业务组件想要语义化的命名，也会带给开发人员很多负担；并且功能抽象能力因人而异，公共函数往往不够公用或者过度设计。
- **生态问题** 函数式编程在工业生产领域因其抽象性和性能带来的问题，被许多开发者拒之门外，一些特定功能的解决方案也更小众（相比其他编程范式），生态也一直比较小，这成为一些新的开发人员学习和使用函数式编程的又一个巨大障碍。
- …

日常业务开发中，往往我们需要取长补短，在适合的领域用适合的方法 / 范式。大家只要要记住，软件开发并没有“银弹”。

7. FAQ

Q：你觉得 Promise 是不是一种 Monad IO 模型?

A：我认为是的。纯函数是没有异步概念的，Promise 用了一种很棒的方式把异步和 IO 转化为了 .then 函数。你仍然可以在 .then 函数中写纯粹的函数，也可以在 .then 函数中调用其他的 Promise，这就和 `IO Monad` 的行为非常像。

Q：你愿意在生产中使用 Haskell/Lisp/Clojure 等纯函数式语言吗?

A：不论是否愿意使用，现在很多语言都开始引入函数式编程语法了。并不是说函数式编程一定是优秀的，但它至少没有那么恐怖。有一点可以肯定的是，学习函数式编程可以扩展我们的思维，增加我们看问题的角度。

Q：有没有一些可以预见的好处?

A：有的。比如强制你写代码的时候去关注状态量（多少、是否引用值、是否变更等），这或多或少可以帮助你写代码的时候减少状态量的使用，也慢慢地能复合一些状态量，写出更简洁的代码。

Q：函数式编程能给业务带来什么好处?

A：业务拆分的时候，函数式的思维是单向的，我们会通过实现，想到它对应需要的基础组件，并递归地思考下去，功能实现从最小粒度开始，上层逐步通过函数组合来实现。相比于面向对象，这种方式在组合上更方便简洁，更容易把复杂度降低，比如面向对象中可能对象之间的相互引用和调用是没有限制的，这种模式带来的是思考逻辑的时候思维会发散。

这种对比在业务复杂的情况下更加明显，面向对象必须要优秀的设计模式来实现控制代码复杂度增长不那么快，而函数式编程大多数情况下都是单向数据流 + 基础工具库就减少了大量的复杂度，而且产生的代码更简洁。

8. 作者简介

俊杰，美团到家研发平台 / 医药技术部前端工程师。

9. 参考文献

1. 维基百科：函数式编程 /lambda 演算 / 范畴论 / 集合论 / 群论。
2. Github：getify/Functional-Light-JS
3.《Learn You A Haskell For Great Good!》
4.《Deep JavaScript》
5. 其他：各类在线博客

Android 对 so 体积优化的探索与实践

作者：洪凯　常强

1. 背景

应用安装包的体积影响着用户的下载时长、安装时长、磁盘占用空间等诸多方面，因此减小安装包的体积对于提升用户体验和下载转化率都大有益处。Android 应用安装包其实是一个 zip 文件，主要由 dex、assets、resource、so 等各类型文件压缩而成。目前业内常见的包体积优化方案大体分为以下几类：

- 针对 dex 的优化，例如 Proguard、dex 的 DebugItem 删除、字节码优化等；
- 针对 resource 的优化，例如 AndResGuard、webp 优化等；
- 针对 assets 的优化，例如压缩、动态下发等；
- 针对 so 的优化，同 assets，另外还有移除调试符号等。

随着动态化、端智能等技术的广泛应用，在采用上述优化手段后，so 在安装包体积中的比重依然很高，我们开始思索这部分体积是否能进一步优化。

经过一段时间的调研、分析和验证，我们逐渐摸索出一套可以将应用安装包中 so 体积进一步减小 30% ~ 60% 的方案。该方案包含一系列纯技术优化手段，对业务侵入性低，通过简单的配置，可以快速部署生效，目前美团 App 已在线上部署使用。为让大家能知其然，也能知其所以然，本文将先从 so 文件格式讲起，结合文件格式分析哪些内容可以优化。

2. so 文件格式分析

so 即动态库，本质上是 ELF（Executable and Linkable Format）文件。可以从两个维度查看 so 文件的内部结构：链接视图（Linking View）和执行视图（Execution View）。链接视图将 so 主体看作多个 section 的组合，该视图体现的是 so 是如何组

装的，是编译链接的视角。而执行视图将 so 主体看作多个 segment 的组合，该视图告诉动态链接器如何加载和执行该 so，是运行时的视角。鉴于对 so 优化更侧重于编译链接角度，并且通常一个 segment 包含多个 section（即链接视图对 so 的分解粒度更小），因此我们这里只讨论 so 的链接视图。

通过 `readelf -S` 命令可以查看一个 so 文件的所有 section 列表，参考 ELF 文件格式说明，这里简要介绍一下本文涉及的 section：

- `.text`：存放的是编译后的机器指令，C/C++ 代码的大部分函数编译后就存放在这里。这里只有机器指令，没有字符串等信息。
- `.data`：存放的是初始值不为零的一些可读写变量。
- `.bss`：存放的是初始值为零或未初始化的一些可读写变量。该 section 仅指示运行时需要的内存大小，不会占用 so 文件的体积。
- `.rodata`：存放的是一些只读常量。
- `.dynsym`：动态符号表，给出了该 so 对外提供的符号（导出符号）和依赖外部的符号（导入符号）的信息。
- `.dynstr`：字符串池，不同字符串以‘\0’分割，供 `.dynsym` 和其他部分使用。
- `.gnu.hash` 和 `.hash`：两种类型的哈希表，用于快速查找 `.dynsym` 中的导出符号或全部符号。
- `.gnu.version`、`.gnu.version_d`、`.gnu.version_r`：这三个 section 用于指定动态符号表中每个符号的版本，其中 `.gnu.version` 是一个数组，其元素个数与动态符号表中符号的个数相同，即数组每个元素与动态符号表的每个符号是一一对应的关系。数组每个元素的类型为 `Elfxx_Half`，其意义是索引，指示每个符号的版本。`.gnu.version_d` 描述了该 so 定义的所有符号的版本，供 `.gnu.version` 索引。`.gnu.version_r` 描述了该 so 依赖的所有符号的版本，也供 `.gnu.version` 索引。因为不同的符号可能具有相同的版本，所以采用这种索引结构，可以减小 so 文件的大小。

在进行优化之前，我们需要对这些 section 以及它们之间的关系有一个清晰的认识，下图较直观地展示了 so 中各个 section 之间的关系（这里只绘制了本文涉及的 section）：

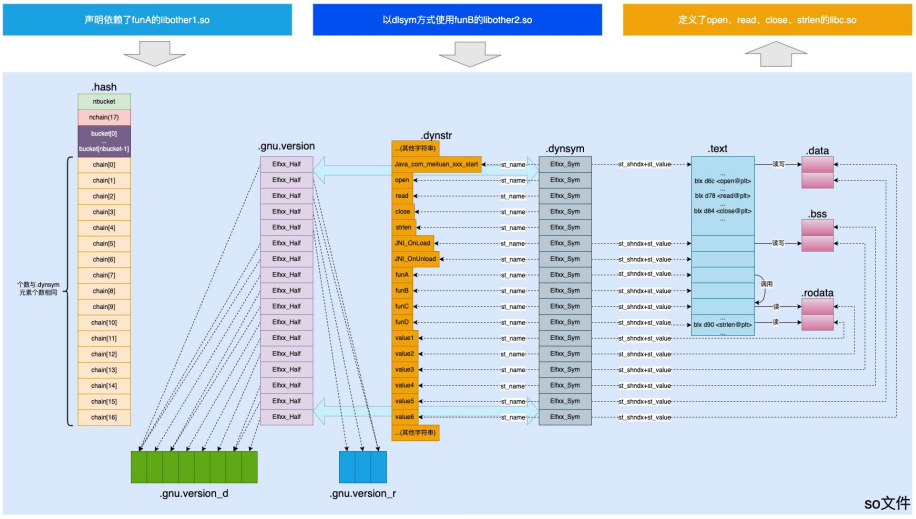


图 1　so 文件结构示意图

结合上图，我们从另一个角度来理解 so 文件的结构：想象一下，我们把所有的函数实现体都放到 `.text` 中，`.text` 中的指令会去读取 `.rodata` 中的数据，读取或修改 `.data` 和 `.bss` 中的数据。看上去 so 中有这些内容也足够了。但是这些函数怎样执行呢？也就是说，只把这些函数和数据加载进内存是不够的，这些函数只有真正去执行，才能发挥作用。

我们知道想要执行一个函数，只要跳转到它的地址就行了。那外界调用者（该 so 之外的模块）怎样知道它想要调用函数的地址呢？这里就涉及一个函数 ID 的问题：外部调用者给出需要调用的函数的 ID，而动态链接器（Linker）根据该 ID 查找目标函数的地址并告知外部调用者。所以 so 文件还需要一个结构去存储“ID- 地址”的映射关系，这个结构就是动态符号表的所有导出符号。

具体到动态符号表的实现，ID 的类型是“字符串”，可以说动态符号表的所有导出符

号构成了一个“字符串 – 地址“的映射表。调用者获取目标函数的地址后，准备好参数跳转到该地址就可以执行这个函数了。另一方面，当前 so 可能也需要调用其他 so 中的函数（例如 libc.so 中的 read、write 等），动态符号表的导入符号记录了这些函数的信息，在 so 内函数执行之前动态链接器会将目标函数的地址填入到相应位置，供该 so 使用。所以动态符号表是连接当前 so 与外部环境的“桥梁”：导出符号供外部使用，导入符号声明了该 so 需要使用的外部符号（注：实际上 `.dynsym` 中的符号还可以代表变量等其他类型，与函数类型类似，这里就不再赘述）。

结合 so 文件结构，接下来我们开始分析 so 中有哪些内容可以优化。

3. so 可优化内容分析

在讨论 so 可优化内容之前，我们先了解一下 Android 构建工具（Android Gradle Plugin，下文简称 AGP）对 so 体积做的 strip 优化（移除调试信息和符号表）。AGP 编译 so 时，首先产生的是带调试信息和符号表的 so（任务名为 externalNative-BuildRelease），之后对刚产生的带调试信息和符号表的 so 进行 strip，就得到了最终打包到 apk 或 aar 中的 so（任务名为 stripReleaseDebugSymbols）。

strip 优化的作用就是删除输入 so 中的调试信息和符号表。这里说的符号表与上文中的“动态符号表”不同，符号表所在 section 名通常为 .symtab，它通常包含了动态符号表中的全部符号，并且额外还有很多符号。调试信息顾名思义就是用于调试该 so 的信息，主要是各种名字以 `.debug_` 开头的 section，通过这些 section 可以建立 so 每条指令与源码文件的映射关系（也就是能够对 so 中每条指令找到其对应的源码文件名、文件行号等信息）。 之所以叫 strip 优化，是因为其实际调用的是 NDK 提供的的 strip 命令（所用参数为 –strip-unneeded）。

注：为什么 AGP 要先编译出带调试信息和符号表的 so，而不直接编译出最终的 so 呢（通过添加 `-s` 参数是可以做到直接编译出没有调试信息和符号表的 so 的）？原因就在于需要使用带调试信息和符号表的 so 对崩溃调用栈进行还原。删除了调试信息和符号表的 so 完全可以正常运行，但是当它发生崩溃时，只能保证获取到崩溃调

用栈的每个栈帧的相应指令在 so 中的位置，不一定能获取到符号。但是排查崩溃问题时，我们希望得知 so 崩溃在源码的哪个位置。带调试信息和符号表的 so 可以将崩溃调用栈的每个栈帧还原成其对应的源码文件名、文件行号、函数名等，大大方便了崩溃问题的排查。所以说，虽然带调试信息和符号表的 so 不会打包到最终的 apk 中，但它对排查问题来说非常重要。

AGP 通过开启 strip 优化，可以大幅缩减 so 的体积，甚至可以达到十倍以上。以一个测试 so 为例，其最终 so 大小为 14 KB，但是对应的带调试信息和符号表的 so 大小为 136 KB。不过在使用中，我们需要注意的是，如果 AGP 找不到对应的 strip 命令，就会把带调试信息和符号表的 so 直接打包到 apk 或 aar 中，并不会打包失败。例如缺少 armeabi 架构对应的 strip 命令时提示信息如下：

```
Unable to strip library 'XXX.so' due to missing strip tool for ABI
'ARMEABI'. Packaging it as is.
```

除了上述 Android 构建工具默认为 so 体积做的优化，我们还能做哪些优化呢？首先明确我们优化的原则：

- 对于必须保留的内容考虑进行缩减，减小体积占用；
- 对于无需保留的内容直接删除。

基于以上原则，可以从以下三个方面对 so 继续进行深入优化：

- **精简动态符号表**：上文已经提到，动态符号表是 so 与外部进行连接的“桥梁”，其中的导出表相当于是 so 对外暴露的接口。哪些接口是必须对外暴露的呢？在 Android 中，大部分 so 是用来实现 Java 的 native 方法的，对于这种 so，只要让应用运行时能够获取到 Java native 方法对应的函数地址即可。要实现这个目标，有两种方法：一种是使用 RegisterNatives 动态注册 Java native 方法，一种是按照 JNI 规范定义 `java_***` 样式的函数并导出其符号。RegisterNatives 方式可以提前检测到方法签名不匹配的问题，并且可以减少导出符号的数量，这也是 Google 推荐的做法。所以在最优情况下只需导出

JNI_OnLoad（在其中使用 RegisterNatives 对 Java native 方法进行动态注册）和 JNI_OnUnload（可以做一些清理工作）这两个符号即可。如果不希望改写项目代码，也可以再导出 java_*** 样式的符号。除了上述类型的 so，剩余的 so 通常是被应用的其他 so 动态依赖的，对于这类 so，需要确定所有动态依赖它的 so 依赖了它的哪些符号，仅保留这些被依赖的符号即可。另外，这里应区分符号表项与实现体，符号表项是动态符号表中相应的 Elfxx_Sym 项（见上图），实现体是其在 .text、.data、.bss、.rodata 等或其他部分的实体。删除了符号表项，实现体不一定要被删除。结合上文 so 文件结构示意图，可以预估出删除一个符号表项后 so 减小的体积为：符号名字符串长度 + 1 + Elfxx_Sym + Elfxx_Half + Elfxx_Word 。

- **移除无用代码**：在实际的项目中，有一些代码在 Release 版中永远不会被使用到（例如历史遗留代码、用于测试的代码等），这些代码被称为 DeadCode。而根据上文分析，只有动态符号表的导出符号直接或间接引用到的所有代码才需要保留，其他剩余的所有代码都是 DeadCode，都是可以删除的（注：事实上 .init_array 等特殊 section 涉及的代码也要保留）。删除无用代码的潜在收益较大。
- **优化指令长度**：实现某个功能的指令并不是固定的，编译器有可能能用更少的指令完成相同的功能，从而实现优化。由于指令是 so 的主要组成部分，因此优化这一部分的潜在收益也比较大。

so 可优化内容如下图所示（可删除部分用红色背景标出，可优化部分是 .text），其中 funC、value2、value3、value6 由于分别被需保留部分使用，所以需要保留其实现体，只能删除其符号表项。funD、value1、value4、value5 可删除符号表项及其实现体（注：因为 value4 的实现体在 .bss 中，而 .bss 实际不占用 so 的体积，所以删除 value4 的实现体不会减小 so 的体积）。

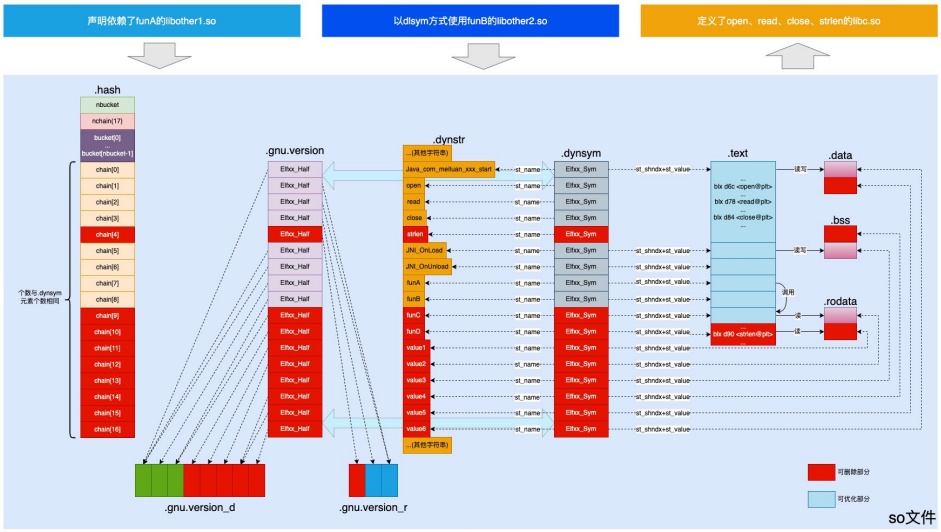


图 2　so 可优化部分

在确定了 so 中可以优化的内容后，我们还需要考虑优化时机的问题：是直接修改 so 文件，还是控制其生成过程？考虑到直接修改 so 文件的风险与难度较大，控制 so 的生成过程显然更稳妥。为了控制 so 的生成过程，我们先简要介绍一下 so 的生成过程：

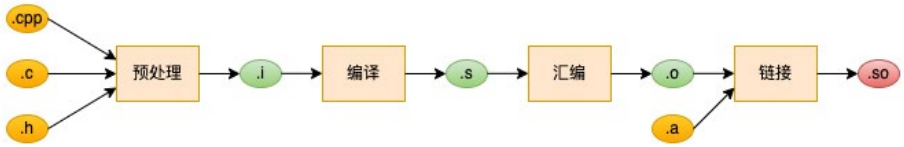


图 3　so 文件的生成过程

如上图所示，so 的生成过程可以分为四个阶段：

- **预处理**：将 include 头文件处扩展为实际文件内容并进行宏定义替换。
- **编译**：将预处理后的文件编译成汇编代码。

- **汇编**：将汇编代码汇编成目标文件，目标文件中包含机器指令（大部分情况下是机器指令，见下文 LTO 一节）和数据以及其他必要信息。
- **链接**：将输入的所有目标文件以及静态库（.a 文件）链接成 so 文件。

可以看出，预处理和汇编阶段对特定输入产生的输出基本是固定的，优化空间较小。所以我们的优化方案主要是针对编译和链接阶段进行优化。

4. 优化方案介绍

我们对所有能控制最终 so 体积的方案都进行调研，并验证了其效果，最后总结出较为通用的可行方案。

4.1 精简动态符号表

使用 visibility 和 attribute 控制符号可见性

可以通过给编译器传递 `-fvisibility=VALUE` 控制全局的符号可见性，VALUE 常取值为 default 和 hidden：

- **default**：除非对变量或函数特别指定符号可见性，所有符号都在动态符号表中，这也是不使用 -fvisibility 时的默认值。
- **hidden**：除非对变量或函数特别指定符号可见性，所有符号在动态符号表中都不可见。

CMake 项目的配置方式：

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fvisibility=hidden")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fvisibility=hidden")
```

ndk-build 项目的配置方式：

```
LOCAL_CFLAGS += -fvisibility=hidden
```

另一方面，针对单个变量或函数，可以通过 attribute 方式指定其符号可见性，示例

如下：

```
__attribute__((visibility("hidden")))
int hiddenInt=3;
```

其常用值也是 default 和 hidden，与 visibility 方式意义类似，这里不再赘述。

attribute 方式指定的符号可见性的优先级，高于 visibility 方式指定的可见性，相当于 visibility 是全局符号可见性开关，attribute 方式是针对单个符号的可见性开关。这两种方式结合就能控制源码中每个符号的可见性。

需要注意的是上面这两种方式，只能控制变量或函数是否存在于动态符号表中（即是否删除其动态符号表项），而不会删除其实现体。

使用 static 关键字控制符号可见性

在 C/C++ 语言中，static 关键字在不同场景下有不同意义，当使用 static 表示“该函数或变量仅在本文件可见”时，那么这个函数或变量就不会出现在动态符号表中，但只会删除其动态符号表项，而不会删除其实现体。static 关键字相当于是增强的 hidden（因为 static 声明的函数或变量编译时只对当前文件可见，而 hidden 声明的函数或变量只是在动态符号表中不存在，在编译期间对其他文件还是可见的）。在项目开发中，使用 static 关键字声明一个函数或变量“仅在本文件可见”是很好的习惯，但是不建议使用 static 关键字控制符号可见性：无法使用 static 关键字控制一个多文件可见的函数或变量的符号可见性。

使用 exclude libs 移除静态库中的符号

上述 visibility 方式、attribute 方式和 static 关键字，都是控制项目源码中符号的可见性，而无法控制依赖的静态库中的符号在最终 so 中是否存在。exclude libs 就是用来控制依赖的静态库中的符号是否可见，它是传递给链接器的参数，可以使依赖的静态库的符号在动态符号表中不存在。同样，也是只能删除符号表项，实现体仍然会存在于产生的 so 文件中。

CMake 项目的配置方式：

```
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -Wl,--
exclude-libs,ALL")# 使所有静态库中的符号都不被导出
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -Wl,--
exclude-libs,libabc.a")# 使 libabc.a 的符号都不被导出
```

ndk-build 项目的配置方式：

```
LOCAL_LDFLAGS += -Wl,--exclude-libs,ALL # 使所有静态库中的符号都不被导出
LOCAL_LDFLAGS += -Wl,--exclude-libs,libabc.a # 使 libabc.a 的符号都不被导出
```

使用 version script 控制符号可见性

version script 是传递给链接器的参数，用来指定动态库导出哪些符号以及符号的版本。该参数会影响到上面“so 文件格式”一节中 `.gnu.version` 和 `.gnu.version_d` 的内容。我们现在只使用它的指定所有导出符号的功能（即符号版本名使用空字符串）。开启 version script 需要先编写一个文本文件，用来指定动态库导出哪些符号。示例如下（只导出 usedFun 这一个函数）：

```
{
    global:usedFun;
    local:*;
};
```

然后将上述文件的路径传递给链接器即可（假定上述文件名为 `version_script.txt`）。

CMake 项目的配置方式：

```
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -Wl,-
-version-script=${CMAKE_CURRENT_SOURCE_DIR}/version_script.txt")
#version_script.txt 与当前 CMakeLists.txt 同目录
```

ndk-build 项目的配置方式：

```
LOCAL_LDFLAGS += -Wl,--version-script=${LOCAL_PATH}/version_script.txt
#version_script.txt 与当前 Android.mk 同目录
```

看上去，version script 是明确地指定需要保留的符号，如果通过 visibility 结合 attribute 的方式控制每个符号是否导出，也能达到 version script 的效果，但是 version script 方式有一些额外的好处：

1. version script 方式可以控制编译进 so 的静态库的符号是否导出，visibility 和 attribute 方式都无法做到这一点。
2. visibility 结合 attribute 方式需要在源码中标明每个需要导出的符号，对于导出符号较多的项目来说是很繁杂的。version script 把需要导出的符号统一地放到了一起，能够直观方便地查看和修改，对导出符号较多的项目也非常友好。
3. version script 支持通配符，`*` 代表 0 个或者多个字符，`?` 代表单个字符。比如 `my*;` 就代表所有以 my 开头的符号。有了通配符的支持，配置 version script 会更加方便。
4. 还有非常特殊的一点，version script 方式可以删除 `__bss_start` 这样的一些符号（这是链接器默认加上的符号）。

综上所述，version script 方式优于 visibility 结合 attribute 的方式。同时，使用了 version script 方式，就不需要使用 exclude libs 方式控制依赖的静态库中的符号是否导出了。

4.2 移除无用代码

开启 LTO

LTO 是 Link Time Optimization 的缩写，即链接期优化。LTO 能够在链接目标文件时检测出 DeadCode 并删除它们，从而减小编译产物的体积。DeadCode 举例：某个 if 条件永远为假，那么 if 为真下的代码块就可以移除。进一步地，被移除代码块所调用的函数也可能因此而变为 DeadCode，它们又可以被移除。能够在链接期做优化的原因是，在编译期很多信息还不能确定，只有局部信息，无法执行一些优化。但是链接时大部分信息都确定了，相当于获取了全局信息，所以可以进行一些优化。GCC 和 Clang 均支持 LTO。LTO 方式编译的目标文件中存储的不再是具体机器的

指令，而是机器无关的中间表示（GCC 采用的是 GIMPLE 字节码，Clang 采用的是 LLVM IR 比特码）。

CMake 项目的配置方式：

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -flto")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -flto")
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -O3 -flto")
```

ndk-build 项目的配置方式：

```
LOCAL_CFLAGS += -flto
LOCAL_LDFLAGS += -O3 -flto
```

使用 LTO 时需要注意几点：

1. 如果使用 Clang，编译参数和链接参数中都要开启 LTO，否则会出现无法识别文件格式的问题（NDK22 之前存在此问题）。使用 GCC 的话，只需要编译参数中开启 LTO 即可。
2. 如果项目工程依赖了静态库，可以使用 LTO 方式重新编译该静态库，那么编译动态库时，就能移除静态库中的 DeadCode，从而减小最终 so 的体积。
3. 经过测试，如果使用 Clang，链接器需要开启非 0 级别的优化，LTO 才能真正生效。经过实际测试（NDK 为 r16b），O1 优化效果较差，O2、O3 优化效果比较接近。
4. 由于需要进行更多的分析计算，开启 LTO 后，链接耗时会明显增加。

开启 GC sections

这是传递给链接器的参数，GC 即 Garbage Collection（垃圾回收），也就是对无用的 section 进行回收。注意，这里的 section 不是指最终 so 中的 section，而是作为链接器的输入的目标文件中的 section。

简要介绍一下目标文件，目标文件（扩展名 .o ）也是 ELF 文件，所以也是由 section 组成的，只不过它只包含了相应源文件的内容：函数会放到 `.text` 样式的

section 中，一些可读写变量会放到 `.data` 样式的 section 中，等等。链接器会把所有输入的目标文件的同类型的 section 进行合并，组装出最终的 so 文件。

GC sections 参数通知链接器：仅保留动态符号（及 `.init_array` 等）直接或者间接引用到的 section，移除其他无用 section。这样就能减小最终 so 的体积。但开启 GC sections 还需要考虑一个问题：编译器默认会把所有函数放到同一个 section 中，把所有相同特点的数据放到同一个 section 中，如果同一个 section 中既有需要删除的部分又有需要保留的部分，会使得整个 section 都要保留。所以我们需要减小目标文件 section 的粒度，这需要借助另外两个编译参数 `-fdata-sections` 和 `-ffunction-sections`，这两个参数通知编译器，将每个变量和函数分别放到各自独立的 section 中，这样就不会出现上述问题了。实际上 Android 编译目标文件时会自动带上 `-fdata-sections` 和 `-ffunction-sections` 参数，这里一并列出来，是为了突出它们的作用。

CMake 项目的配置方式：

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fdata-sections -ffunction-sections")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fdata-sections -ffunction-
sections")
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -Wl,--gc-
sections")
```

ndk-build 项目的配置方式：

```
LOCAL_CFLAGS += -fdata-sections -ffunction-sections
LOCAL_LDFLAGS += -Wl,--gc-sections
```

4.3 优化指令长度

使用 Oz/Os 优化级别

编译器根据输入的 -Ox 参数决定编译的优化级别，其中 O0 表示不开启优化（这种情况主要是为了便于调试以及更快的编译速度），从 O1 到 O3，优化程度越来越强。Clang 和 GCC 均提供了 Os 的优化级别，其与 O2 比较接近，但是优化了生成产物

的体积。而 Clang 还提供了 Oz 优化级别，在 Os 的基础上能进一步优化产物体积。

综上，编译器是 Clang，可以开启 Oz 优化。如果编译器是 GCC，则只能开启 Os 优化（注：NDK 从 r13 开始默认编译器从 GCC 变为 Clang，r18 中正式移除了 GCC。GCC 不支持 Oz 是指 Android 最后使用的 GCC4.9 版本不支持 Oz 参数）。Oz/Os 优化相比于 O3 优化，优化了产物体积，性能上可能有一定损失，因此如果项目原本使用了 O3 优化，可根据实际测试结果以及对性能的要求，决定是否使用 Os/Oz 优化级别，如果项目原本未使用 O3 优化级别，可直接使用 Os/Oz 优化。

CMake 项目的配置方式（如果使用 GCC，应将 Oz 改为 Os）：

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Oz")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Oz")
```

ndk-build 项目的配置方式（如果使用 GCC，应将 Oz 改为 Os）：

```
LOCAL_CFLAGS += -Oz
```

4.4 其他措施

禁用 C++ 的异常机制

如果项目中没有使用 C++ 的异常机制（例如 `try...catch` 等），可以通过禁用 C++ 的异常机制，来减小 so 的体积。

CMake 项目的配置方式：

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-exceptions")
```

ndk-build 默认会禁用 C++ 的异常机制，因此无需特意禁用（如果现有项目开启了 C++ 的异常机制，说明确有需要，需仔细确认后才能禁用）。

禁用 C++ 的 RTTI 机制

如果项目中没有使用 C++ 的 RTTI 机制（例如 typeid 和 dynamic_cast 等），可以通过禁用 C++ 的 RTTI ，来减小 so 的体积。

CMake 项目的配置方式：

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-rtti")
```

ndk-build 默认会禁用 C++ 的 RTTI 机制，因此无需特意禁用（如果现有项目开启了 C++ 的 RTTI 机制，说明确有需要，需仔细确认后才能禁用）。

合并 so

以上都是针对单个 so 的优化方案，对单个 so 进行优化后，还可以考虑对 so 进行合并，能够进一步减小 so 的体积。具体来讲，当安装包内某些 so 仅被另外一个 so 动态依赖时，可以将这些 so 合并为一个 so。例如 liba.so 和 libb.so 仅被 libx.so 动态依赖，可以将这三个 so 合并为一个新的 libx.so。合并 so 有以下好处：

1. 可以删除部分动态符号表项，减小 so 总体积。具体来讲，就是可以删除 liba.so 和 libb.so 的动态符号表中的所有导出符号，以及 libx.so 的动态符号表中从 liba.so 和 libb.so 中导入的符号。
2. 可以删除部分 PLT 表项和 GOT 表项，减小 so 总体积。具体来讲，就是可以删除 libx.so 中与 liba.so、libb.so 相关的 PLT 表项和 GOT 表项。
3. 可以减轻优化的工作量。如果没有合并 so，对 liba.so 和 libb.so 做体积优化时需要确定 libx.so 依赖了它们的哪些符号，才能对它们进行优化，做了 so 合并后就不需要了。链接器会自动分析引用关系，保留使用到的所有符号的对应内容。
4. 由于链接器对原 liba.so 和 libb.so 的导出符号拥有了更全的上下文信息，LTO 优化也能取得更好的效果。

可以在不修改项目源码的情况下，在编译层面实现 so 的合并。

提取多 so 共同依赖库

上面“合并 so”是减小 so 总个数，而这里是增加 so 总个数。当多个 so 以静态方式依赖了某个相同的库时，可以考虑将此库提取成一个单独的 so，原来的几个 so 改

为动态依赖该 so。例如 liba.so 和 libb.so 都静态依赖了 libx.a，可以优化为 liba.so 和 libb.so 均动态依赖 libx.so。提取多 so 共同依赖库，可以对不同 so 内的相同代码进行合并，从而减小总的 so 体积。

这里典型的例子是 libc++ 库：如果存在多个 so 都静态依赖 libc++ 库的情况，可以优化为这些 so 都动态依赖于 `libc++_shared.so`。

4.5 整合后的通用方案

通过上述分析，我们可以整合出普通项目均可使用的通用的优化方案，CMake 项目的配置方式（如果使用 GCC，应将 Oz 改为 Os）：

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Oz -flto -fdata-sections
-ffunction-sections")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Oz -flto -fdata-sections
-ffunction-sections")
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -O3 -flto
-Wl,--gc-sections -Wl,--version-script=${CMAKE_CURRENT_SOURCE_DIR}/
version_script.txt") #version_script.txt 与当前 CMakeLists.txt 同目录
```

ndk-build 项目的配置方式（如果使用 GCC，应将 Oz 改为 Os）：

```
LOCAL_CFLAGS += -Oz -flto -fdata-sections -ffunction-sections
LOCAL_LDFLAGS += -O3 -flto -Wl,--gc-sections -Wl,--version-
script=${LOCAL_PATH}/version_script.txt #version_script.txt 与当前
Android.mk 同目录
```

其中 `version_script.txt` 较为通用的配置如下，可根据实际情况添加需要保留的导出符号：

```
{
    global:JNI_OnLoad;JNI_OnUnload;Java_*;
    local:*;
};
```

说明：version script 方式指定所有需要导出的符号，不再需要 visibility 方式、attribute 方式、static 关键字和 exclude libs 方式控制导出符号。是否禁用 C++ 的异常机制和 RTTI 机制、合并 so 以及提取多 so 共同依赖库取决于具体项目，不具

有通用性。

至此，我们总结出一套可行的 so 体积优化方案。但在工程实践中，还有一些问题要解决。

5. 工程实践

支持多种构建工具

美团有众多业务使用了 so，所使用的构建工具也不尽相同，除了上述常见的 CMake 和 ndk-build，也有项目在使用 Make、Automake、Ninja、GYP 和 GN 等各种构建工具。不同构建工具应用 so 优化方案的方式也不相同，尤其对大型工程而言，配置复杂性较高。

基于以上原因，每个业务自行配置 so 优化方案会消耗较多的人力成本，并且有配置无效的可能。为了降低配置成本、加快优化方案的推进速度、保证配置的有效性和正确性，我们在构建平台上统一支持了 so 的优化（支持使用任意构建工具的项目）。业务只需进行简单的配置即可开启 so 的体积优化。

配置导出符号的注意事项

注意事项有以下两点：

1. 如果一个 so 的某些符号，被其他 so 通过 dlsym 方式使用，那么这些符号也应该保留在该 so 的导出符号中（否则会导致运行时异常）。
2. 编写 `version_script.txt` 时需要注意 C++ 等语言对符号的修饰，不能直接把函数名填写进去。符号修饰就是把一个函数的命名空间（如果有）、类名（如果有）、参数类型等都添加到最终的符号中，这也是 C++ 语言实现重载的基础。有两种方式可以把 C++ 的函数添加到导出符号中：第一种是查看未优化 so 的导出符号表，找到目标函数被修饰后的符号，然后填写到 `version_script.txt` 中。例如有一个 MyClass 类：

```
class MyClass{
   void start(int arg);
   void stop();
};
```

要确定 start 函数真正的符号可以对未优化的 libexample.so 执行以下命令。因为 C++ 对符号修饰后，函数名是符号的一部分，所以可以通过 grep 加快查找：

```
➜  ~ nm -D --defined-only libexample.so | grep start
000006e8 T _ZN7MyClass5startEi
```

图 4　查找 start 函数真正符号

可以看到 start 函数真正的符号是 `_ZN7MyClass5startEi`。如果想导出该函数，`version_script.txt` 相应位置填入 `_ZN7MyClass5startEi` 即可。

第二种方式是在 `version_script.txt` 中使用 extern 语法，如下所示：

```
{
    global:
      extern "C++" {
            MyClass::start*;
        "MyClass::stop()";
      };
    local:*;
};
```

上述配置可以导出 MyClass 的 start 和 stop 函数。其原理是，链接时链接器对每个符号进行 demangle（解构，即把修饰后的符号还原为可读的表示），然后与 extern “C++”中的条目进行匹配，如果能与任一条目匹配成功就保留该符号。匹配的规则是：有双引号的条目不能使用通配符，需要全字符串完全匹配才可以（例如 stop 条目，如果括号之间多一个空格就会匹配失败）。对于没有双引号的条目能够使用通配符（例如 start 条目）。

查看优化后 so 的导出符号

业务对 so 进行优化之后，需要查看最终的 so 文件中保留了哪些导出符号，验证优

化效果是否符合预期。在 Mac 和 Linux 下均可使用下述命令查看 so 保留了哪些导出符号：

```
nm -D --defined-only xxx.so
```

例如：

```
➜  ~ nm -D --defined-only libexample.so
00000658 T JNI_OnLoad
00000668 T Java_com_example_MainActivity_stringFromJNI
```

图 5　nm 命令查看 so 文件的导出符号

可以看出，libexample.so 的导出符号有两个：`JNI_OnLoad` 和 `Java_com_example_MainActivity_stringFromJNI`。

解析崩溃堆栈

本文的优化方案会移除非必要导出的动态符号，那 so 如果发生崩溃的话是不是就无法解析崩溃堆栈了呢？答案是完全不会影响崩溃堆栈的解析结果。

“so 可优化内容分析”一节已经提过，使用带调试信息和符号表的 so 解析线上崩溃，是分析 so 崩溃的标准方式（这也是 Google 解析 so 崩溃的方式）。本文的优化方案并未修改调试信息和符号表，所以可以使用带调试信息和符号表的 so 对崩溃堆栈进行完整的还原，解析出崩溃堆栈每个栈帧对应的源码文件、行号和函数名等信息。业务编译出 release 版的 so 后将相应的带调试信息和符号表的 so 上传到 crash 平台即可。

6. 方案收益

优化 so 对安装包体积和安装后占用的本地存储空间有直接收益，收益大小取决于原 so 冗余代码数量和导出符号数量等具体情况，下面是部分 so 优化前后占用安装包体积的对比：

so	优化前大小	优化后大小	优化百分比
A 库	4.49 MB	3.28 MB	27.02%
B 库	995.82 KB	728.38 KB	26.86%
C 库	312.05 KB	153.81 KB	50.71%
D 库	505.57 KB	321.75 KB	36.36%
E 库	309.89 KB	157.08 KB	49.31%
F 库	88.59 KB	62.93 KB	28.97%

下面是上述 so 优化前后占用本地存储空间的对比：

so	优化前大小	优化后大小	优化百分比
A 库	10.67 MB	7.04 MB	34.05%
B 库	2.35 MB	1.61 MB	31.46%
C 库	898.14 KB	386.31 KB	56.99%
D 库	1.30 MB	771.47 KB	41.88%
E 库	890.13 KB	398.30 KB	55.25%
F 库	230.30 KB	146.06 KB	36.58%

7. 总结与后续计划

对 so 体积进行优化不仅能够减小安装包体积，而且能获得以下收益：

- 删除了大量的非必要导出符号从而提升了 so 的安全性。
- 因为 `.data` `.bss` `.text` 等运行时占用内存的 section 减小了，所以也能减小应用运行时的内存占用。
- 如果优化过程中减少了 so 对外依赖的符号，还可以加快 so 的加载速度。

我们对后续工作做了如下的规划：

- 提升编译速度。因为使用 LTO、gc sections 等会增加编译耗时，计划调研 ThinLTO 等方案对编译速度进行优化。
- 详细展示保留各个函数 / 数据的原因。
- 进一步完善平台优化 so 的能力。

8. 参考资料

- https://www.cs.cmu.edu/afs/cs/academic/class/15213-f00/docs/elf.pdf
- https://llvm.org/docs/LinkTimeOptimization.html
- https://gcc.gnu.org/onlinedocs/gccint/LTO-Overview.html
- https://sourceware.org/binutils/docs/ld/VERSION.html
- https://clang.llvm.org/docs
- https://gcc.gnu.org/onlinedocs/gcc

9. 本文作者

洪凯、常强，来自美团平台 /App 技术部。

从 0 到 1：美团端侧 CDN 容灾解决方案

作者：魏磊　心澎　陈彤

1. 前言

作为业务研发，你是否遇到过因为 CDN 问题导致的业务图片加载失败，页面打开缓慢，页面布局错乱或者页面白屏？你是否又遇到过某些区域 CDN 域名异常导致业务停摆，客诉不断，此时的你一脸茫然，不知所措？作为 CDN 运维，你是否常常被业务方反馈的各种 CDN 问题搞得焦头烂额，一边顶着各种催促和压力寻求解决方案，一边抱怨着服务商的不靠谱？今天，我们主要介绍一下美团外卖技术团队端侧 CDN 的容灾方案，经过实践，我们发现该产品能有效减少运维及业务开发同学的焦虑，希望我们的这些经验也能够帮助到更多的技术团队。

2. 背景

CDN 因能够有效解决因分布、带宽、服务器性能带来的网络访问延迟等问题，已经成为互联网不可或缺的一部分，也是前端业务严重依赖的服务之一。在实际业务生产中，我们通常会将大量的静态资源如 JS 脚本、CSS 资源、图片、视频、音频等托管至 CDN 服务，以享受其边缘节点缓存对静态资源的加速。但是在享用 CDN 服务带来更好体验的同时，也经常会被 CDN 故障所影响。比如因 CDN 边缘节点异常，CDN 域名封禁等导致页面白屏、排版错乱、图片加载失败。

每一次的 CDN 故障，业务方往往束手无策，只能寄希望于 CDN 团队。而 CDN 的监控与问题排查，对 SRE 也是巨大的难题和挑战。一方面，由于 CDN 节点的分布广泛，边缘节点的监控就异常困难。另一方面，各业务汇聚得到的 CDN 监控大盘，极大程度上隐匿了细节。小流量业务、定点区域的 CDN 异常往往会被淹没。SRE 团队也做了很多努力，设计了多种方案来降低 CDN 异常对业务的影响，也取得了一定的效果，但始终有几个问题无法很好解决：

- **时效性**：当 CDN 出现问题时，SRE 会手动进行 CDN 切换，因为需要人为操作，响应时长就很难保证。另外，切换后故障恢复时间也无法准确保障。
- **有效性**：切换至备份 CDN 后，备份 CDN 的可用性无法验证，另外因为 Local DNS 缓存，无法解决域名劫持和跨网访问等问题。
- **精准性**：CDN 的切换都是大范围的变更，无法针对某一区域或者某一项目单独进行。
- **风险性**：切换至备份 CDN 之后可能会导致回源，流量剧增拖垮源站，从而引发更大的风险。

当前，美团外卖业务每天服务上亿人次，即使再小的问题在巨大的流量面前，也会被放大成大问题。外卖的动态化架构，70% 的业务资源都依赖于 CDN，所以 CDN 的可用性严重影响着外卖业务。如何更有效的进行 CDN 容灾，降低 CDN 异常对业务的影响，是我们不断思考的问题。

既然以上问题 SRE 侧无法完美地解决，端侧是不是可以进行一些尝试呢？比如将 CDN 容灾前置到终端侧。不死鸟（Phoenix）就是在这样的设想下，通过前端能力建设，不断实践和完善的一套端侧 CDN 容灾方案。该方案不仅能够有效降低 CDN 异常对业务的影响，还能提高 CDN 资源加载成功率，现已服务整个美团多个业务和 App。

3. 目标与场景

3.1 核心目标

为降低 CDN 异常对业务的影响，提高业务可用性，同时降低 SRE 同学在 CDN 运维方面的压力，在方案设计之初，我们确定了以下目标：

- **端侧 CDN 域名自动切换**：在 CDN 异常时，端侧第一时间感知并自动切换 CDN 域名进行加载重试，减少对人为操作的依赖。
- **CDN 域名隔离**：CDN 域名与服务厂商在区域维度实现服务隔离且服务等效，

保证 CDN 切换重试的有效性。

- **更精准有效的 CDN 监控**：建设更细粒度的 CDN 监控，能够按照项目维度实时监控 CDN 可用性，解决 SRE CDN 监控粒度不足，告警滞后等问题。并根据容灾监控对 CDN 容灾策略实施动态调整，减少 SRE 切换 CDN 的频率。
- **域名持续热备**：保证每个 CDN 域名的持续预热，避免流量切换时导致回源。

3.2 适用场景

适用所有依赖 CDN ，希望降低 CDN 异常对业务影响的端侧场景，包括 Web、SSR Web、Native 等技术场景。

4. Phoenix 方案

一直以来，CDN 的稳定性是由 SRE 来保障，容灾措施也一直在 SRE 侧进行，但仅仅依靠链路层面上的保障，很难处理局部问题和实现快速止损。用户终端作为业务的最终投放载体，对资源加载有着天然的独立性和敏感性。如果将 CDN 容灾前置到终端侧，无论从时效性，精准性，都是 SRE 侧无法比拟的。在端侧进行容灾，就需要感知 CDN 的可用性，然后实现端侧自动切换的能力。我们调研整个前端领域，并未发现业内在端侧 CDN 容灾方面有所实践和输出，所以整个方案的实现是从无到有的一个过程。

4.1 总体设计

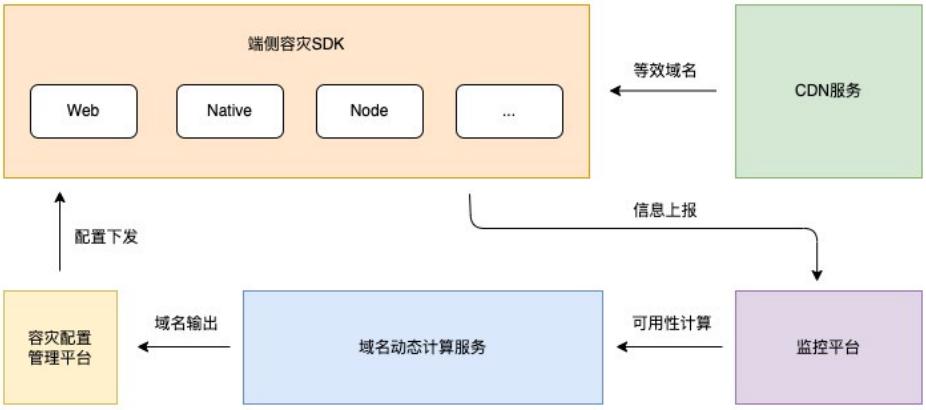


图 1

Phoenix 端侧 CDN 容灾方案主要由五部分组成：

- **端侧容灾 SDK**：负责端侧资源加载感知，CDN 切换重试，监控上报。
- **动态计算服务**：根据端侧 SDK 上报数据，对多组等效域名按照城市、项目、时段等维度定时轮询计算域名可用性，动态调整流量至最优 CDN。同时也是对 CDN 可用性的日常巡检。
- **容灾监控平台**：从项目维度和大盘维度提供 CDN 可用性监控和告警，为问题排查提供详细信息。
- **CDN 服务**：提供完善的 CDN 链路服务，在架构上实现域名隔离，并为业务方提供等效域名服务，保证端侧容灾的有效性。等效域名，就是能够通过相同路径访问到同一资源的域名，比如：cdn1.meituan.net/src/js/test.js 和 cdn2.meituan.net/src/js/test.js 能够返回相同内容，则 cdn1.meituan.net 和 cdn2.meituan.net 互为等效域名。
- **容灾配置平台**：对项目容灾域名进行配置管理，监控上报策略管理，并提供 CDN 流量人工干预等措施。

4.2 容灾流程设计

为保证各个端侧容灾效果和监控指标的一致性，我们设计了统一的容灾流程，整体流程如下：

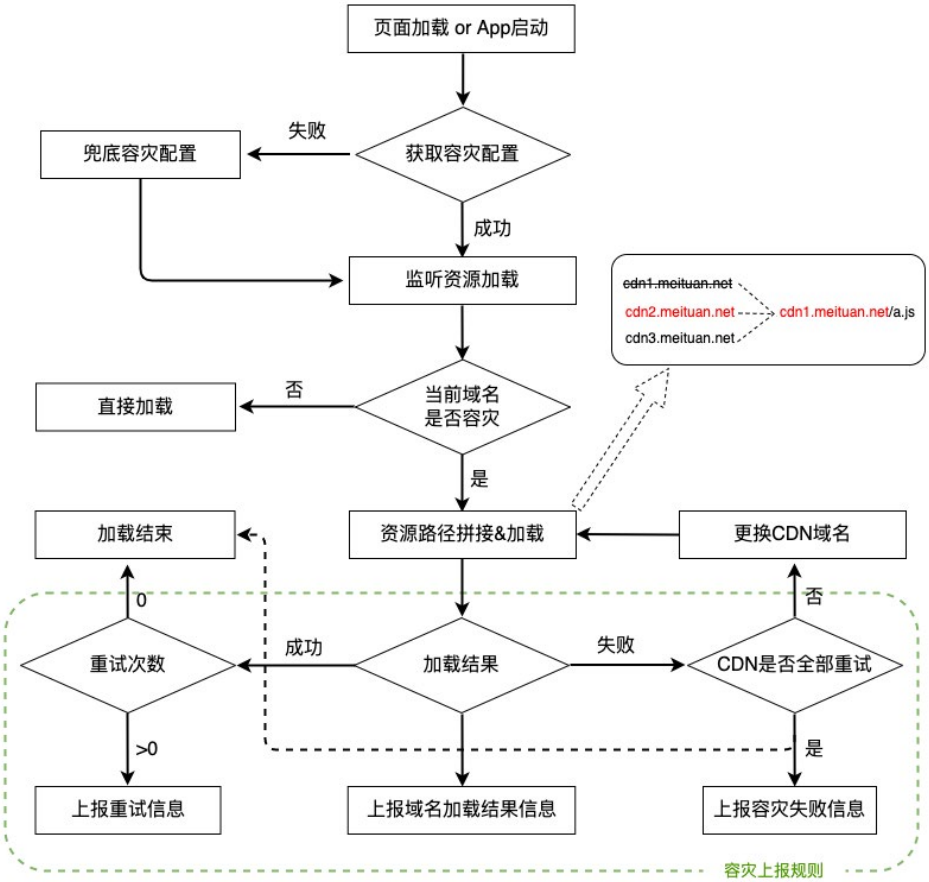


图 2

4.3 实现原理

4.3.1 端侧容灾 SDK

Web 端实现

Web 端的 CDN 资源主要是 JS、CSS 和图片，所以我们的容灾目标也聚焦于这些。

在 Web 侧的容灾，我们主要实现了对静态资源，异步资源和图片资源的容灾。

实现思路

要实现资源的容灾，最主要的问题是感知资源加载结果。通常我们是在资源标签上面添加错误回调来捕获，图片容灾可以这样实现，但这并不适合 JS，因为它有严格的执行顺序。为了解决这一问题，我们将传统的标签加载资源的方式，换成 **XHR** 来实现。通过 **Webpack** 在工程构建阶段把同步资源进行抽离，然后通过 **Phoenix-Loader** 来加载资源。这样就能通过网络请求返回的状态码，来感知资源加载结果。

在方案的实现上，我们将 SDK 设计成了 Webpack Plugin，主要基于以下四点考虑：

1. **通用性**：美团前端技术栈相对较多，要保证容灾 SDK 能够覆盖大部分的技术框架。
2. **易用性**：过高的接入成本会增加开发人员的工作量，不能做到对业务的有效覆盖，方案价值也就无从谈起。
3. **稳定性**：方案要保持稳定可靠，不受 CDN 可用性干扰。
4. **侵入性**：不能侵入到正常业务，要做到即插即用，保证业务的稳定性。

通过调研发现，前端有 70% 的工程构建都离不开 Webpack，而 Webpack Plugin 独立配置，即插即用的特性，是实现方案的最好选择。整体方案设计如下：

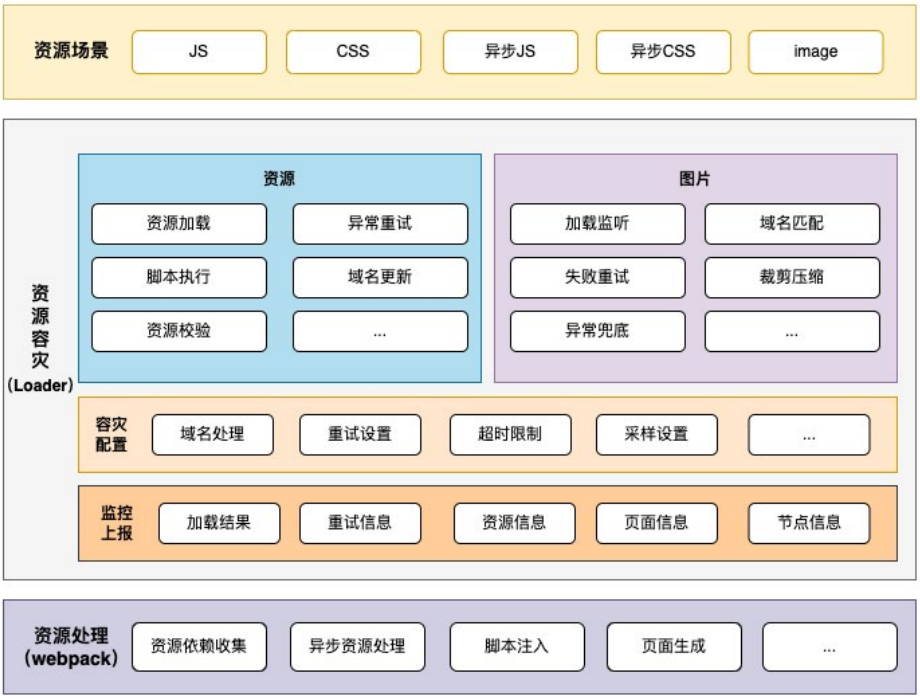


图 3

当然，很多团队在做性能优化时，会采取代码分割，按需引入的方式。这部分资源在同步资源生成的过程中无法感知，但这部分资源的加载结果，也关系到业务的可用性。在对异步资源的容灾方面，我们主要是通过对 Webpack 的异步资源处理方式进行重写，使用 **Phoenix Loader** 接管资源加载，从而实现异步资源的容灾。整体分析过程如下图所示：

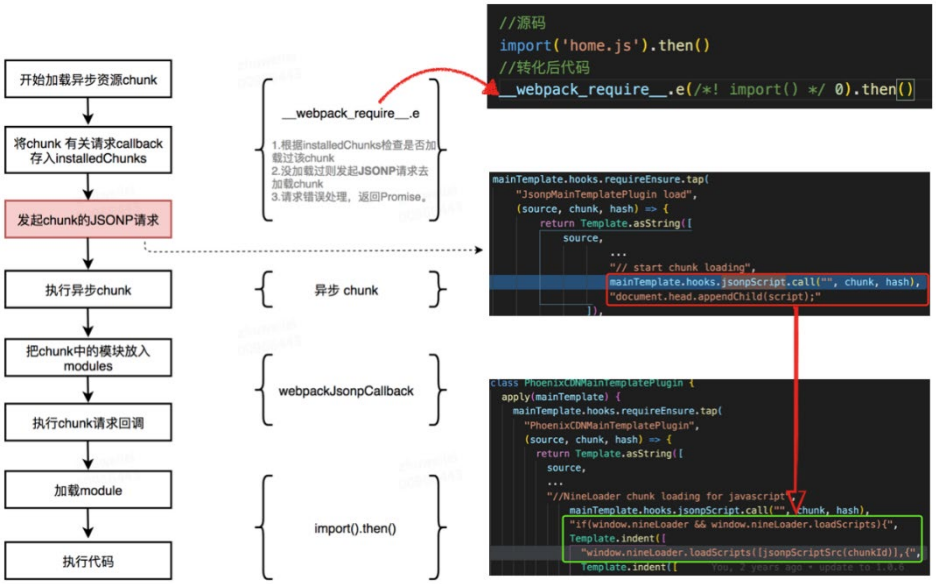


图 4

CSS 资源的处理与 JS 有所差别，但原理相似，只需要重写 **mini-css-extract-plugin** 的异步加载实现即可。

Web 端方案资源加载示意：

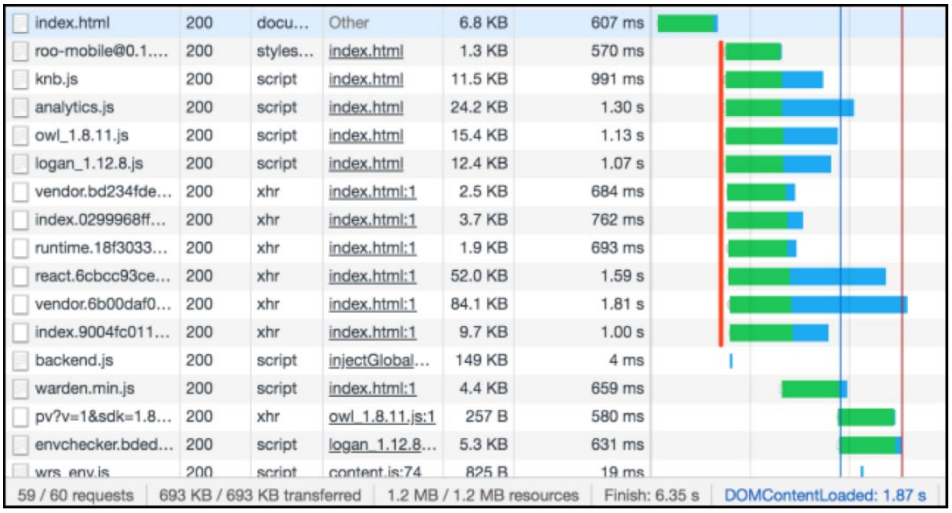


图 5

容灾效果

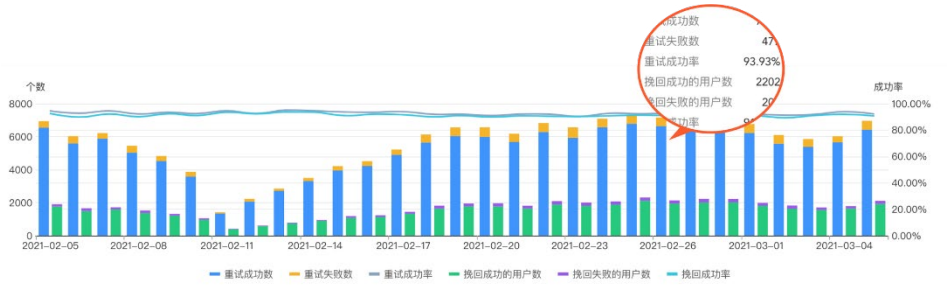


图 6　容灾大盘

图 7　容灾案例

Native 端容灾

客户端的 CDN 资源主要是图片，音视频以及各种动态化方案的 bundle 资源。Native 端的容灾建设也主要围绕上述资源展开。

实现思路

重新请求是 Native 端 CDN 容灾方案的基本原理，根据互备 CDN 域名，由 Native 容灾基建容灾域名重新进行请求资源，整个过程发生在原始请求失败后。Native 容灾基建不会在原始请求过程中进行任何操作，避免对原始请求产生影响。原始请求失

败后，Native 容灾基建代理处理失败返回，业务方仍处于等待结果状态，重请新求结束后向业务方返回最终结果。整个过程中从业务方角度来看仍只发出一次请求，收到一次结果，从而达到业务方不感知的目的。为将重新请求效率提升至最佳，必须尽可能的保证重新请求次数趋向于最小。

调研业务的关注点和技术层面使用的网络框架，结合 Phoenix 容灾方案的基本流程，在方案设计方面，我们主要考虑以下几点：

- **便捷性**：接入的便捷性是 SDK 设计时首先考虑的内容，即业务方可以用最简单的方式接入，实现资源容灾，同时也可以简单无残留拆除 SDK。
- **兼容性**：Android 侧的特殊性在于多样的网络框架，集团内包括 Retrofit 框架，okHttp 框架，okHttp3 框架及已经很少使用的 URLConnection 框架。提供的 SDK 应当与各种网络框架兼容，同时业务方在即使变更网络框架也能够以最小的成本实现容灾功能。而 iOS 侧则考虑复用一个 NSURLProtocol 去实现对请求的拦截，降低代码的冗余度，同时实现对初始化项进行统一适配。
- **扩展性**：需要在基础功能之上提供可选的高级配置来满足特殊需求，包括监控方面也要提供特殊的监控数据上报能力。

基于以上设计要点，我们将 Phoenix 划分为以下结构图，图中将整体的容灾 SDK 拆分为两部分 Phoenix-Adaptor 部分与 Phoenix-Base 部分。

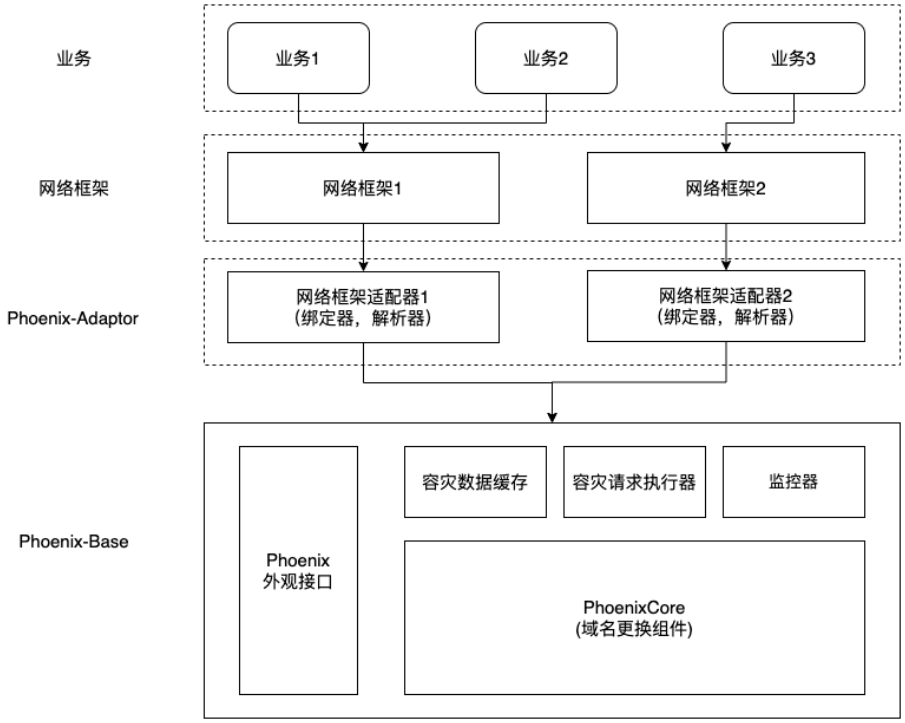


图 8

Phoenix-Base

Phoenix-Base 是整个 Phoenix 容灾的核心部分，其包括容灾数据缓存，域名更换组件，容灾请求执行器（区别于原始请求执行器），监控器四个对外不可见的内部功能模块，并包含外部接入模块，提供外部接入功能。

- **容灾数据缓存**：定期获取及更新容灾数据，其产生的数据只会被域名更换组件使用。
- **域名更换组件**：连接容灾数据缓存，容灾请求执行器，监控器的中心节点，负责匹配原始失败 Host，过滤错误码，并向容灾请求执行器提供容灾域名，向监控器提供整个容灾过程的详细数据副本。
- **容灾执行器**：容灾请求的真正请求者，目前采用内部 OkHttp3Client，业务方也可以自主切换至自身的执行器。

- **监控器**：分发容灾过程的详细数据，内置数据大盘的上报，若有外部自定义的监控器，也会向自定义监控器分发数据。

Phoenix-Adaptor

Phoenix-Adaptor 是 Phoenix 容灾的扩展适配部分，用于兼容各种网络框架。

- **绑定器**：生成适合各个网络框架的拦截器并绑定至原始请求执行者。
- **解析器**：将网络框架的 Request 转换为 Phoenix 内部执行器的 Request，并将 Phoenix 内部执行器的 Response 解析为外部网络框架 Response，以此达到适配目的。

容灾效果

① 业务成功率

以外卖图片业务为例，Android 业务成功率对比（同版本 7512，2021.01.17 未开启 Phoenix 容灾，2021.01.19 晚开启 Phoenix 容灾）。

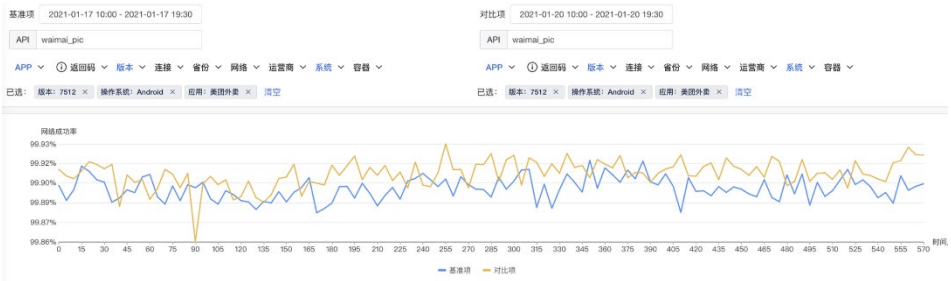


图 9

iOS 业务成功率对比（同版本 7511，2021.01.17 未开启 Phoenix 容灾，2021.01.19 晚开启 Phoenix 容灾）。

图 10

② 风险应对

以外卖与美团图片做为对比，在 CDN 服务出现异常时，接入 Phoenix 的外卖 App 和未接入的美团 App 在图片成功率方面的对比。

图 11

4.3.2 动态计算服务

端侧的域名重试，会在某一域名加载资源失败后，根据容灾列表依次进行重试，直至成功或者失败。如下图所示：

图 12

如果域名 A 大范围异常，端侧依然会首先进行域名 A 的重试加载，这样就导致不必要的重试成本。如何让资源的首次加载更加稳定有效，如何为不同业务和地区动态提供最优的 CDN 域名列表，这就是动态计算服务的要解决的问题。

计算原理

动态计算服务通过域名池和项目的 Appkey 进行关联，按照不同省份、不同地级市、不同项目、不同资源等维度进行策略管理。通过获取 5 分钟内对应项目上报的资源加载结果进行**定时轮询计算**，对域名池中的域名按照地区（城市 && 省份）的可用性监控。计算服务会根据域名可用性动态调整域名顺序并对结果进行输出。下图是一次完整的计算过程：

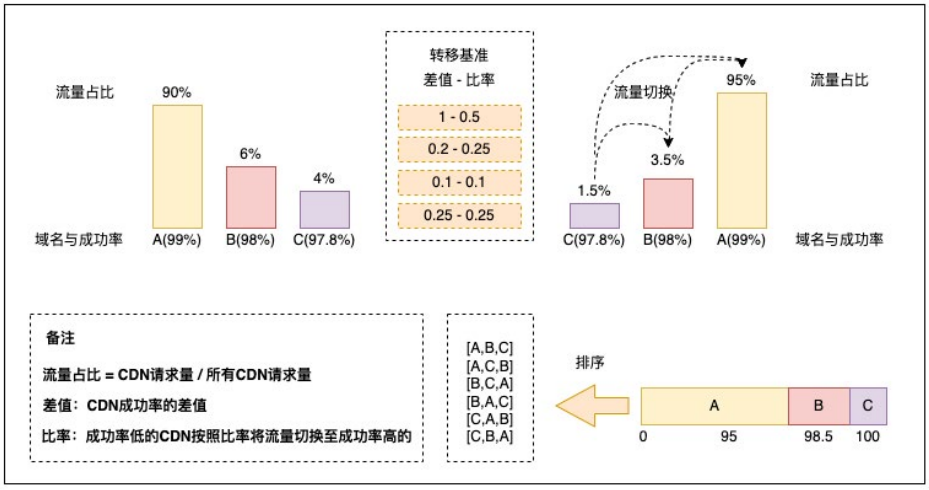


图 13

假设有 A、B、C 三个域名，成功率分别是 99%、98%、97.8%，流量占比分别是 90%、6%、4%。基于转移基准，进行流量转移，比如，A 和 B 成功率差值是 1，B 需要把自己 1/2 的流量转移给 A，同时 A 和 C 的成功率差值大于 1，C 也需要把自己 1/2 的流量转移给 A，同时 B 和 C 的差值是 0.2，所以 C 还需要把自己 1/4 的流量转移给 B。最终，经过计算，A 的流量占比是 95%，B 是 3.5%，C 是 1.5%。最

后，经过排序和随机计算后将最终结果输出。

因为 A 的占比最大，所以 A 优先被选择；通过随机，B 和 C 也会有一定的流量；基于转移基准，可以实现流量的平稳切换。

异常唤起

当某个 CDN 无法正常访问的时候，该 CDN 访问流量会由计算过程切换至等效的 CDN B。如果 SRE 发现切换过慢可以进行手动干预分配流量。当少量的 A 域名成功率上升后，会重复计算过程将 A 的流量加大。直至恢复初始态。

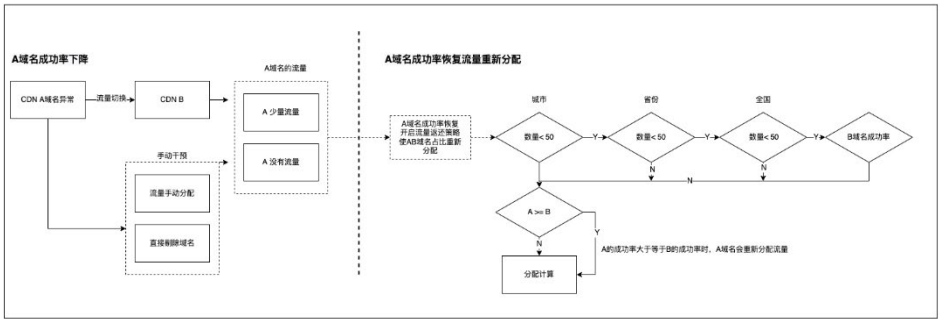


图 14

服务效果

动态计算服务使得资源的首次加载成功率由原来的 **99.7% 提升至 99.9%**。下图为接入动态计算后资源加载成功率与未接入加载成功率对比。

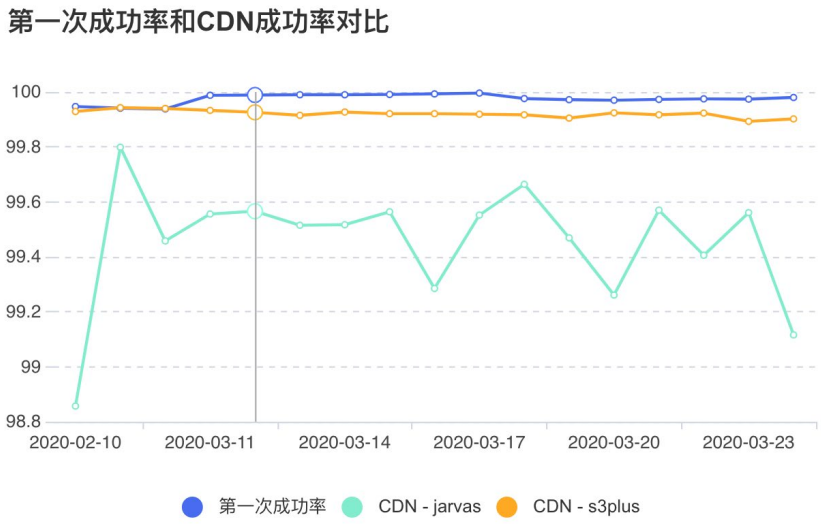


图 15

4.3.3 容灾监控

在监控层面，SRE 团队往往只关注域名、大区域、运营商等复合维度的监控指标，监控流量巨大，对于小流量业务或者小范围区域的 CDN 波动，可能就无法被监控分析识别，进而也就无法感知 CDN 边缘节点异常。容灾监控建设，主要是为了解决 SRE 团队的 CDN 监控告警滞后和监控粒度问题。监控整体设计如下：

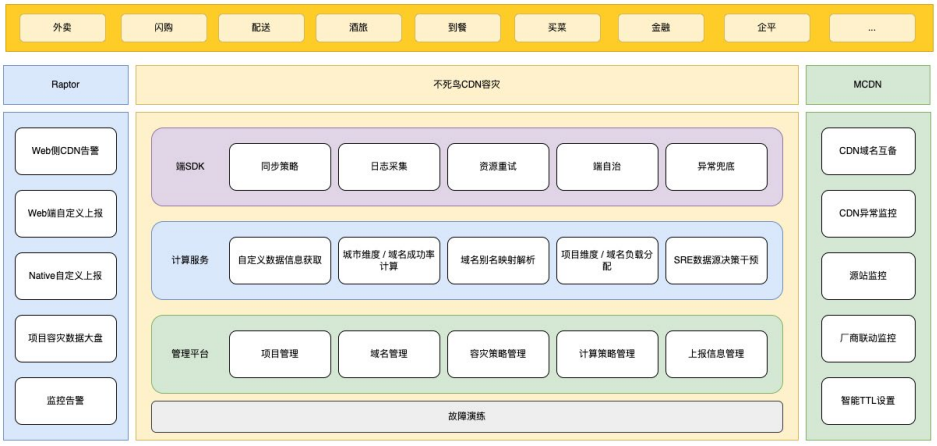


图 16

流程设计

端侧容灾数据的上报，分别按照**项目、App、资源、域名**等维度建立监控指标，将 CDN 可用性变成项目可用性的一部分。通过计算平台对数据进行分析聚合，形成 CDN 可用性大盘，按照域名、区域、项目、时间等维度进行输出，与天网监控互通，建立分钟级别的监控告警机制，大大提升了 CDN 异常感知的灵敏性。同时，SRE 侧的天网监控，也会对动态计算服务结果产生干预。监控整体流程如下：

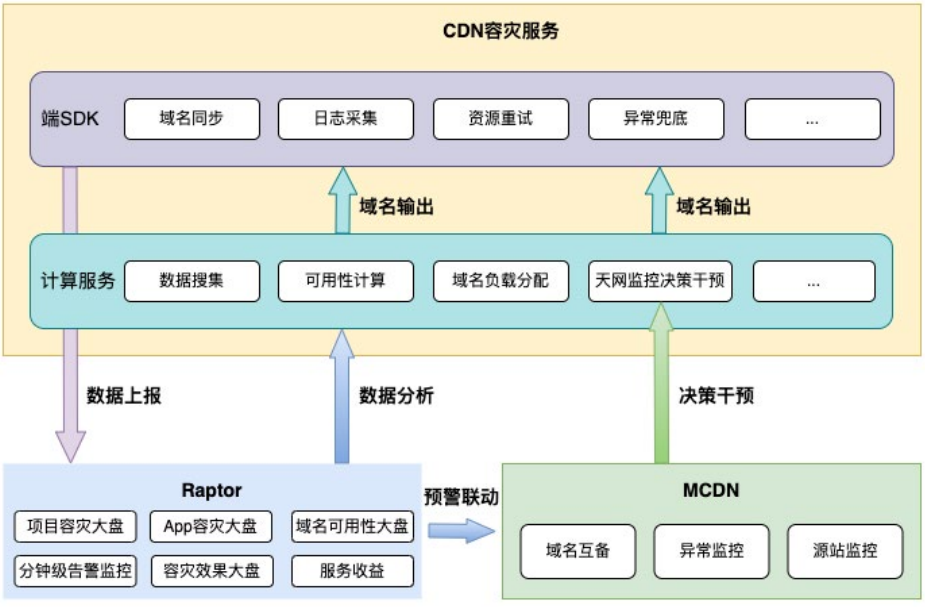


图 17

监控效果

CDN 监控不仅从项目维度更加细粒度的监测 CDN 可用性，还为 CDN 异常排查提供了区域、运营商、网络状况、返回码等更丰富的信息。在监控告警方面，实现了分钟级异常告警，灵敏度也高于美团内部的监控系统。

图 18

4.3.4 CDN 服务

端侧域名切换的有效性，离不开 CDN 服务的支持。在 CDN 服务方面，在原有 SRE 侧容灾的基础上，对 CDN 服务整体做了升级，实现域名隔离，解决了单域名对应多 CDN 和多域名对应单 CDN 重试无效的弊端。

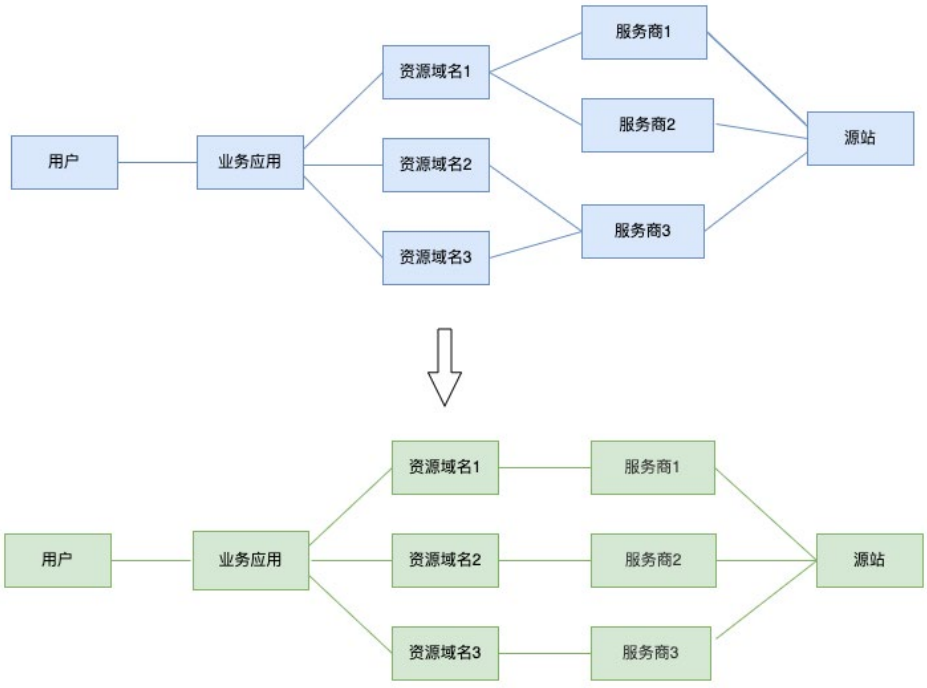


图 19

5. 总结与展望

经过一年的建设与发展，Phoenix CDN 容灾方案日趋成熟，现已成为美团在 CDN 容灾方面唯一的公共服务，在多次 CDN 异常中发挥了巨大的作用。在端侧，当前该方案日均容灾资源 **3000 万 +**，挽回用户 **35 万 +**，覆盖外卖，酒旅，餐饮，优选，买菜等业务部门，服务 200+ 个工程，**外卖 App**、**美团 App**、**大众点评 App** 均已接入。

在 SRE 侧，实现了项目维度的分钟级精准告警，同时丰富了异常信息，大大提高了 SRE 问题排查效率。自从方案大规模落地以来，CDN 异常时鲜有手动切换操作，极大减轻了 SRE 同学的运维压力。

由于前端技术的多样性和复杂性，我们的 SDK 无法覆盖所有的技术方案，所以在接下来的建设中，我们会积极推广我们的容灾原理，公开动态计算服务，希望更多的框架和服务在我们的容灾思想上，贴合自身业务实现端侧的 CDN 容灾。另外，针对方案本身，我们会不断优化资源加载性能，完善资源验签，智能切换等能力，也欢迎对 Phoenix CDN 容灾方案有兴趣的同学，跟我们一起探讨交流。同时更欢迎加入我们，文末附招聘信息，期待你的邮件。

6. 作者简介

魏磊、陈彤、张群、粤俊等，均来自美团外卖平台 – 大前端团队，丁磊、心澎，来自美团餐饮 SaaS 团队。

7. 招聘信息

美团外卖平台 – 大前端团队是一个开放、创新、无边界的团队，鼓励每一位同学追求自己的技术梦想。团队长期招聘 Android、iOS、FE 高级 / 资深工程师和技术专家。欢迎感兴趣的同学投递简历至：wangxiaofei03@meituan.com（邮件标题请注明：美团外卖大前端）。

美团高性能终端实时日志系统建设实践

作者：洪坤　徐博　陈成　少星

1. 背景

1.1 Logan 简介

Logan 是美团面向终端的统一日志服务，已支持移动端 App、Web、小程序、IoT 等多端环境，具备日志采集、存储、上传、查询与分析等能力，帮助用户定位研发问题，提升故障排查效率。同时，Logan 也是业内开源较早的大前端日志系统，具有写入性能高、安全性高、日志防丢失等优点。

1.2 Logan 工作流程

为了方便读者更好地理解 Logan 系统是如何工作的，下图是简化后的 Logan 系统工作流程图。主要分为以下几个部分：

- **主动上报日志**：终端设备在需要上报日志时，可以通过 HTTPS 接口主动上传日志到 Logan 接收服务，接收服务再把原始日志文件转存到对象存储平台。
- **日志解密与解析**：当研发人员想要查看主动上报的日志时会触发日志下载与解析流程，原始加密日志从对象存储平台下载成功后进行解密、解析等操作，然后再投递到日志存储系统。
- **日志查询与检索**：日志平台支持对单设备所有日志进行日志类型、标签、进程、关键字、时间等维度的筛选，同时也支持对一些特定类型的日志进行可视化展示。

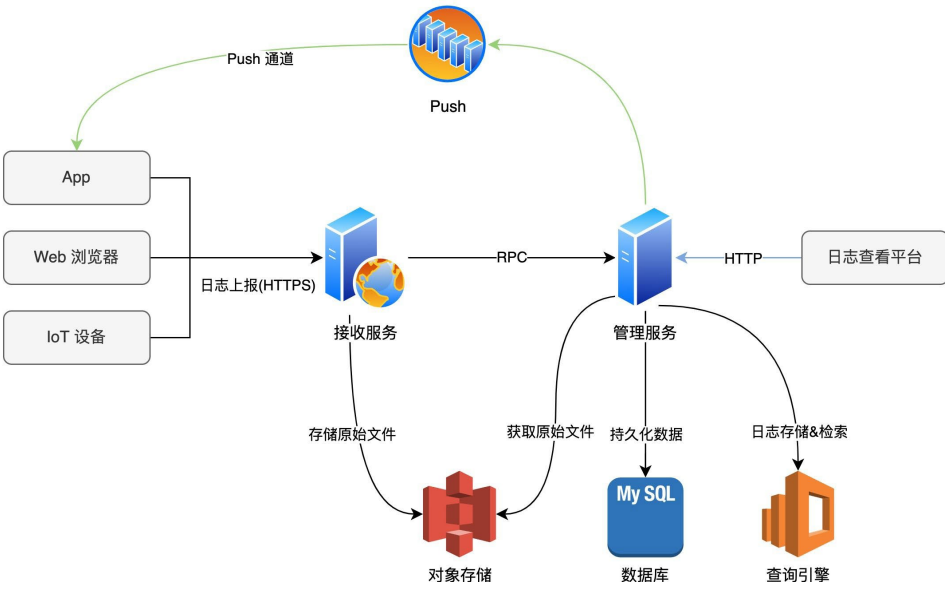


图 1　Logan 系统工作流程图

1.3 为什么需要实时日志?

如前文所述，这套“本地存储 + 主动上报”的模式虽然解决了大前端场景下基础的日志使用需求，但是随着业务复杂度的不断增加，用户对日志的要求也越来越高，当前 Logan 架构存在的问题也变得越来越突出，主要体现在以下几个方面：

1. **部分场景上报日志受限**：由于在 Web 与小程序上用户一般的使用场景是用完即走，当线上出现问题时再联系用户主动上报日志，整个处理周期较长，有可能会错过最佳排查时间。
2. **缺少实时分析和告警能力**：当前缺少实时分析和告警的能力，用户曾多次提到过想要对线上异常日志进行监控，当有符合规则的异常日志出现时能收到告警信息。
3. **缺少全链路追踪能力**：当前多端的日志散落在各个系统中，研发人员在定位问题时需要手动去关联日志，操作起来很不方便，美团内部缺乏一个通用的全链路追踪方案。

针对以上痛点问题，我们提出了建设 Logan 实时日志，旨在提供统一的、高性能的实时日志服务，以解决美团现阶段不同业务系统想要日志上云的需求。

1.4 Logan 实时日志是什么?

Logan 实时日志是服务于移动端 App、Web、小程序、IoT 等终端场景下的实时日志解决方案，旨在提供高扩展性、高性能、高可靠性的实时日志服务，包括日志采集、上传、加工、消费、投递、查询与分析等能力。

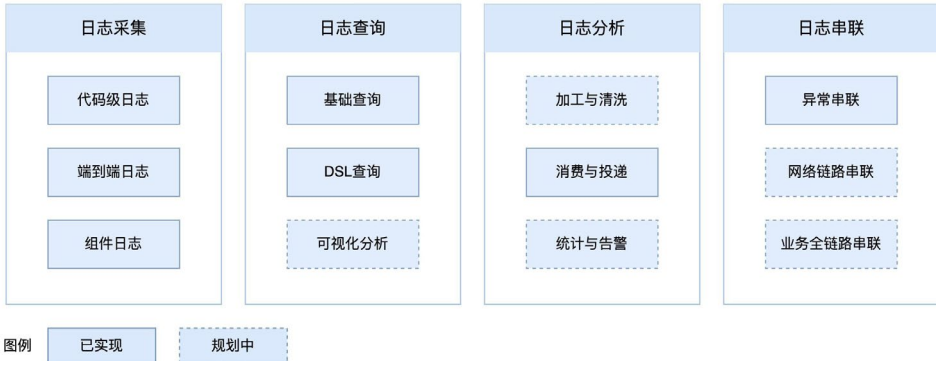


图 2　Logan 实时日志产品功能图

2. 设计实现

2.1 整体架构

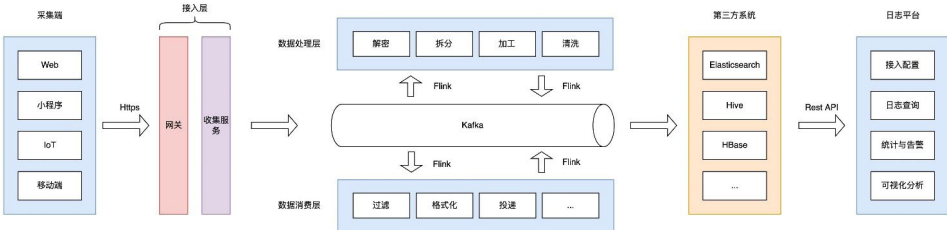


图 3　Logan 实时日志整体架构图

如上图所示，整体架构主要分为五个部分，它们分别是：

- **采集端**：负责端上日志数据的采集、加密、压缩、聚合和上报等。
- **接入层**：负责提供日志上报接口，接收日志上报数据，并将数据转发到数据处理层。
- **数据处理层**：负责日志数据的解密、拆分、加工和清洗等。
- **数据消费层**：负责日志数据的过滤、格式化、投递等。
- **日志平台**：负责日志数据查询与分析、业务系统接入配置、统计与告警等。

2.2 采集端

通用采集端架构，解决跨平台复用

采集端 SDK 用于端侧日志收集，需要在多种终端环境落地，但是由于端和平台较多、技术栈和运行环境也不一致，多端开发和维护成本会比较高。因此，我们设计了一套核心逻辑复用的通用采集端架构，具体的平台依赖代码则单独进行适配。我们目前上线了微信、MMP、Web、MRN 端侧，逻辑层代码做到了完全复用。采集端架构设计图如下：

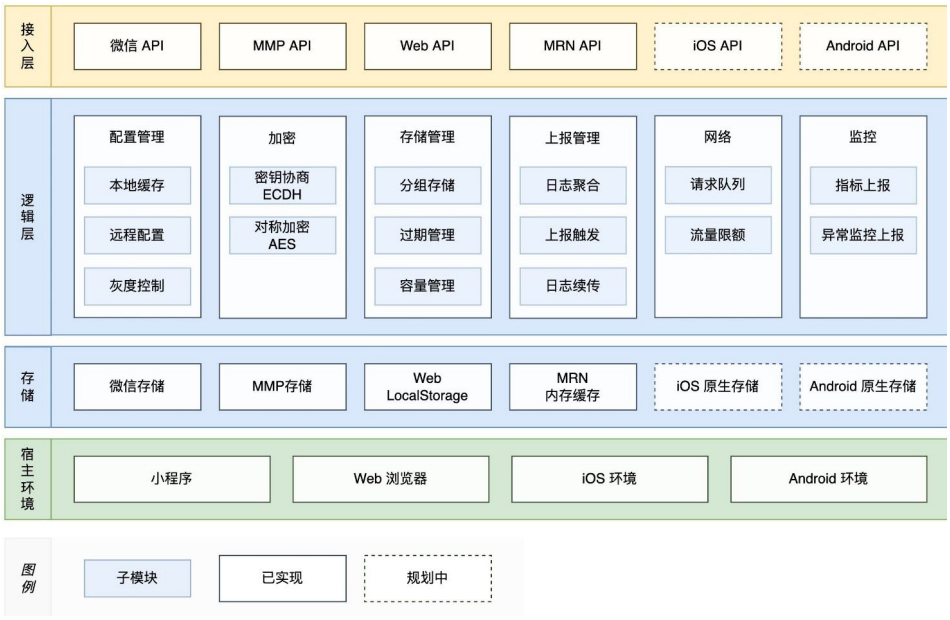


图 4　采集端 SDK 架构图

重点模块介绍：

- **配置管理**：采集端初始化完成后，首先启动配置管理模块，拉取和刷新配置信息，包括上报限流配置、指标采样率、功能开关等，支持对关键配置进行灰度发布。
- **加密**：所有记录的日志都使用 ECDH + AES 方案加密，确保日志信息不泄漏。Web 版加密模块使用浏览器原生加密 API 进行适配，可实现高性能异步加密，其他平台则使用纯 JS 实现。
- **存储管理**：线上数据表明，由于页面关闭导致的日志丢失占比高达 1%，因此我们设计了日志落盘功能，当日志上传失败后会先缓存在本地磁盘，等到页面下一次启动时再重新恢复上传。
- **队列管理**：需要发送的日志首先进行分组聚合，如果等待分组过多则需要丢弃一部分请求，防止在弱网环境或者日志堆积太多时造成内存持续上涨而影响用户。

落盘缓存 + 上报恢复，防止日志丢失

为了方便读者更好地理解端上日志采集过程，下面将具体介绍下采集端流程设计。当采集端初始化 API 开始调用时，先创建 Logger、Encryptor、Storage 等实例对象，并异步拉取环境配置文件。初始化完成之后，先检查是否有成功落盘，但是上报失败的日志，如果有的话立即开始恢复上传流程。当正常调用写日志 API 时，原始日志被加密后加入当前上报组，等到有上报事件（时间、条数、导航等）触发时，当前上报组内的所有日志被加入上报队列并开始上传。采集端详细流程图如下：

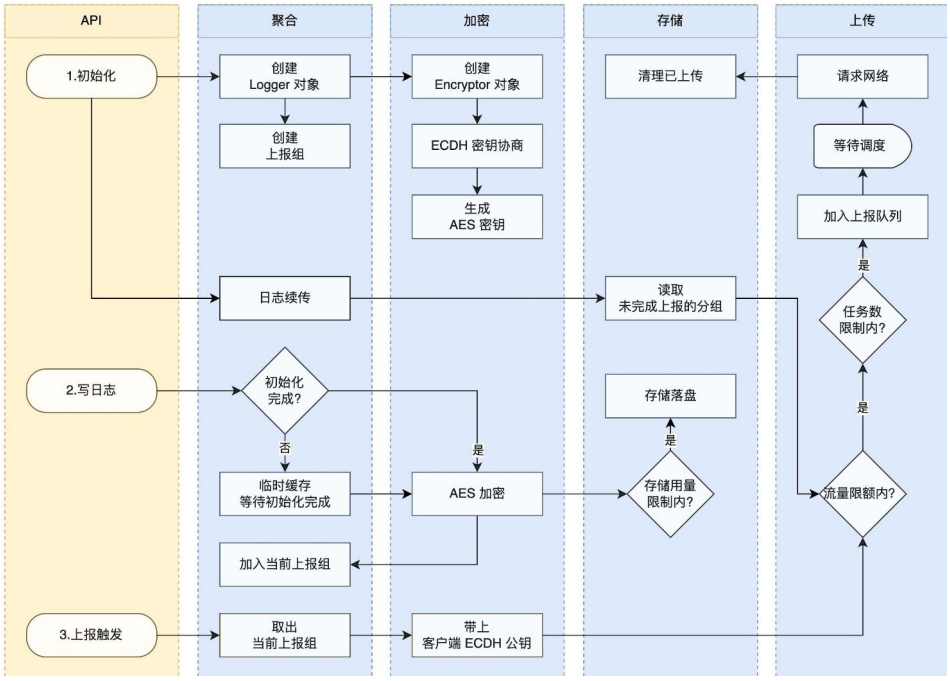


图 5　采集端 SDK 流程图

2.3 数据接入层

对于实时日志系统来讲，接入层需要满足以下几点要求：(1) 支持公网上报域名；(2) 支持高并发处理；(3) 具备高实时性，延迟是分钟级；(4) 支持投递数据到 Kafka 消息队列。

经过对比，美团内的统一日志收集通道均满足以上需求，因此我们选用了统一日志收集通道作为接入层。采集端 SDK 通过独立的公网域名上报日志后，收集通道将日志数据汇总好并投递到指定的 Kafka 消息队列。如果读者公司没有类似的日志收集通道，那么可以参考以下流程搭建实时日志系统接入层。

图 6　接入层流程图

2.4 数据处理层

在数据处理框架的技术选型上，我们先后考虑了三种方案，分别是传统架构（Java 应用）、Storm 架构、Flink 架构，这三种方案在不同维度的对比数据如下：

方案	成熟度	稳定性	扩展性	容错性	延迟	吞吐量	开发成本	运维成本
传统架构	高	高	低	低	高	低	高	高
Storm 架构	高	中	高	高	中	中	中	中
Flink 架构	中	中	高	高	低	高	中	中

表 1　技术选型对比表

综合来看，虽然传统架构有比较好的成熟度与灵活性，但是在扩展性、容错性以及性能上均不能满足系统要求，而 Flink 架构与 Storm 架构都有比较优秀的扩展性与容错性，但是 Flink 架构在延迟与吞吐量上表现要更好，因此我们最终选择了使用 Flink 架构作为数据处理框架。

Flink：业内领先的流式计算引擎，具有高吞吐、低延迟、高可靠和精确计算等优点，对事件窗口有很好的支持，被业内很多公司作为首选的流式计算引擎。

在日志处理流程设计上，日志数据通过接入层处理后被投递到汇总 topic 里面，然后再通过 Flink 作业的逻辑处理后分发到下游。处理流程如下图所示：

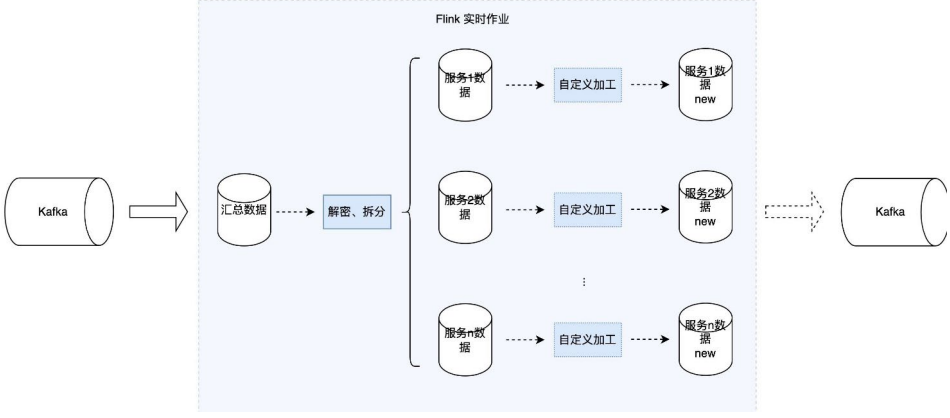


图 7　日志处理层流程图

汇总后的日志数据处理和分发依赖于实时计算平台的实时作业能力，底层使用 Flink 数据处理引擎，主要负责日志数据的解析、日志内容的解密以及拆分到下游等。

1. **元数据解析**：通过实时作业能力完成原始日志数据解析为 JSON 对象数据。
2. **内容解密**：对加密内容进行解密，此处使用非对称协商计算出对称加密密钥，然后再进行解密。
3. **服务维度拆分**：通过 topic 字段把日志分发到各业务系统所属的 topic 里面，从而实现业务日志相互隔离。
4. **数据自定义加工**：根据用户自定义的加工语法模版，从服务 topic 实时消费并加工数据到自定义 topic 中。

2.5 数据消费层

对大部分用户来说 Logan 实时日志提供的日志采集、加工、检索能力已经能满足大部分需求。但是在与用户沟通过程中我们发现还有一些更高阶的需求，比如指标监控、前后端链路串联、离线数据计算等。于是我们将 Logan 标准化后的日志统一投递到 Kafka 流处理平台，并提供一些通用的数据转换能力，方便用户按需接入到不同的第三方系统。数据消费层设计如下图所示：

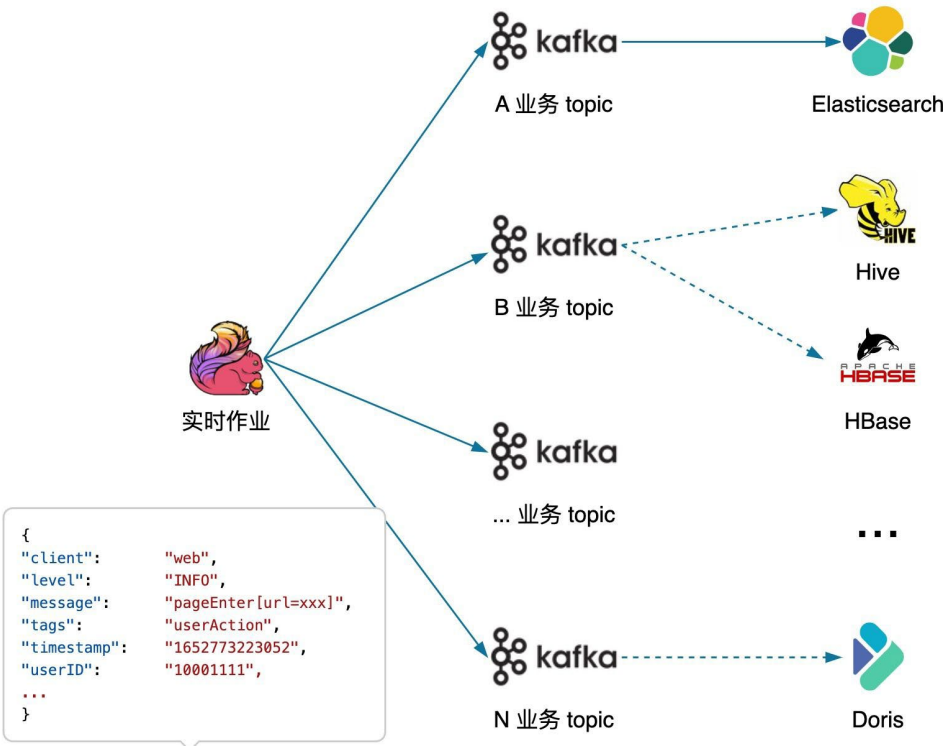


图 8　数据消费层设计图

数据消费层的一些典型的应用场景：

1. **网络全链路追踪**：现阶段前后端的日志可能分布在不同的系统中，前端日志系统记录的主要是代码级日志、端到端日志等，后端日志系统记录的是链路关系、服务耗时等信息。通过 Logan 实时日志开放的数据消费能力，用户可以根据自己的需求来串联多端日志，从而实现网络全链路追踪。
2. **指标聚合统计 & 告警**：实时日志也是一种实时的数据流，可以作为指标数据上报的载体，如果把日志数据对接到数据统计平台就能实现指标监控和告警了。
3. **离线数据分析**：如果在一些需求场景下需要对数据进行长期化保存或者离线分析，就可以将数据导入到 Hive 中来实现。

2.6 日志平台

日志平台的核心功能是为用户提供日志检索支持，日志平台提供了用户标识、自定义标签、关键字等多种检索过滤方式。在日志底层存储架构的选择上，目前业界广泛使用的是 Elasticsearch，考虑到计费与运维成本的关系，美团内部已经有一套统一的框架可以使用，所以我们也选用了 Elasticsearch 架构。同时，我们也支持通过单独的接口层适配其他存储引擎，日志查询流程如下：

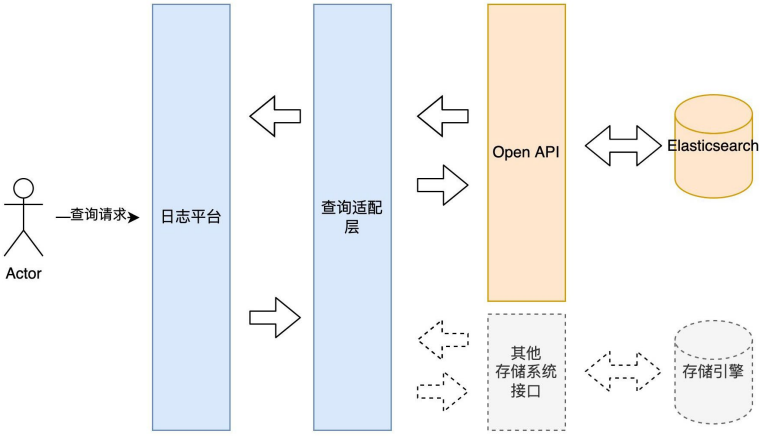


图 9　日志查询流程设计图

Elasticsearch：是一个分布式的开源搜索和分析引擎，具有接入成本低、扩展性高和近实时性等优点，比较适合用来做大数据量的全文检索服务，例如日志查询等。

3. 稳定性保障

3.1 核心监控

为了衡量终端实时日志系统的可用性，我们制定了以下核心 SLA 指标：

指标名称	指标定义	目标
端侧上报成功率	端侧日志上报请求成功次数 / 上报请求总次数	99.5%
服务可用性	服务周期内系统可用时长 / 服务周期总时长	99.9%
日志平均延迟	日志从产生到可以被消费的平均延迟时长	< 1 min

表 2　核心 SLA 指标表格

除了核心指标监控之外，我们还建设了全流程监控大盘，覆盖了分端上报成功率、域名可用性、域名 QPS、作业吞吐量、平均聚合条数等重要观测指标，并且针对上报成功率、域名 QPS、作业吞吐量等配置了兜底告警，当线上有异常时可以第一时间发现并进行处理。

3.2 蓝绿发布

实时日志依赖于实时作业的处理计算能力，但是目前实时作业的发布和部署不能无缝衔接，中间可能存在真空的情况。当重启作业时，需要先停止原作业，再启动新的作业。如果遇到代码故障或系统资源不足等情况时则会导致作业启动失败，从而直接面临消息积压严重和数据延时升高的问题，这对于实时日志系统来说是不能忍受的。

蓝绿发布（Blue Green Deployment）是一种平滑过渡的发布模式。在蓝绿发布模式中，首先要将应用划分为对等的蓝绿两个分组，蓝组发布新产品代码并引入少许线上流量，绿组继续运行老产品代码。当新产品代码经线上运行观察没有问题后，开始逐步引入更多线上流量，直至所有流量都访问蓝组新产品。因此，蓝绿发布可以保证整体系统的稳定，在产品发布前期就可以发现并解决问题，以保证其影响面可控。

目前美团已有的实时作业蓝绿部署方案各不相同，由于 Logan 实时日志接入业务系统较多，且数据量较大，经过综合考量后，我们决定自己实现一套适合当前系统的蓝绿部署方案。为了保证系统的稳定性，在作业运行过程中，启动另外一个相同的作业，当新作业运行没有问题后，再完成新老作业切换。蓝绿发布流程图如下：

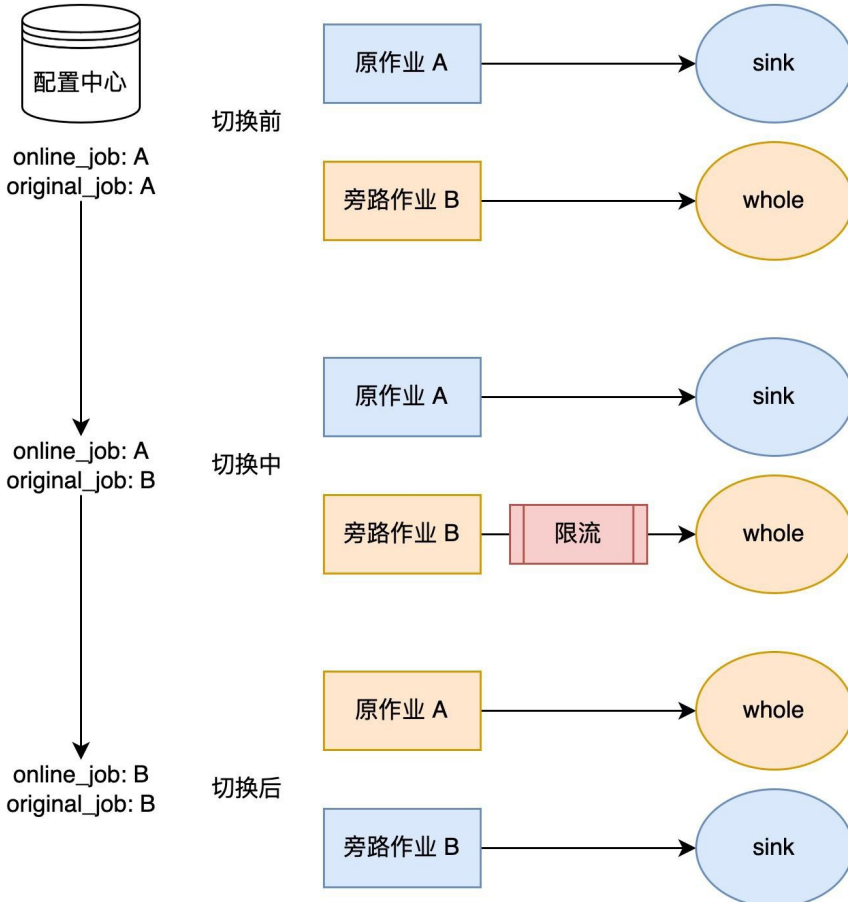


图 10　蓝绿发布流程图

使用蓝绿部署后，彻底解决了由于资源不足或参数不对导致的上线失败问题，平均部署切换耗时也保持在 1 分钟以内，基本避免了因发布带来的日志消费延迟问题。

4. 落地成果

Logan 实时日志在建设初期就受到了各个业务的广泛关注，截止到 2022 年第 3 季度，Logan 实时日志接入并上线的业务系统数量已多达二十余个，其中包括美团小程序、优选商家、餐饮 SaaS 等大体量业务。下面是一些业务系统接入的典型使用场景，供大家参考：

1. **核心链路还原**：到家某 C 端小程序使用 Logan 实时日志记录程序核心链路中的关键日志与异常日志，当线上有客诉问题发生时，可以第一时间查看实时日志并定位问题。项目上线后，平均客诉定位时间从之前的 10 分钟减少到 3 分钟以内，排障效率有明显提升。
2. **内测阶段排障**：企业平台某前端项目由于 2.0 改版改动较大，于是使用 Logan 实时日志在内测阶段添加更多的调试日志，方便定位线上问题。项目上线后，每次排查问题除了节省用户上报日志时间 10-15 分钟，还杜绝了因为存储空间不足而拿不到用户日志的情况。
3. **日志数据分析**：美团到店某团队使用 Logan 实时日志分析前后端交互过程中的请求头、请求参数、响应体等数据是否符合标准化规范。经过一个多月时间的试运行，一期版本上线后共覆盖 300+ 前端页面和 500+ 前端接口，共计发现 1000+ 规范问题。

5. 未来规划

Logan 实时日志经过半年的建设与推广，已经完成了系统基础能力的建设，能满足用户对于实时日志的基本诉求。但是对于日志数据深加工与清洗、日志统计与告警等高阶需求还不支持，因此为了更好地发挥日志价值，提升终端故障排查效率，Logan 实时日志有以下几个方面的规划：

- **完善功能**：支持更多终端类型，建设日志加工与清洗、日志统计与告警、全链路追踪等功能。
- **提升性能**：支持百万并发 QPS、日志上报成功率提升至 99.9% 等。
- **提升稳定性**：建设限流熔断机制、应急响应与故障处理预案等。

6. 本文作者

洪坤、徐博、陈成、少星等，均来自美团 - 基础技术部 - 前端技术中心。

7. 招聘信息

美团基础技术部 – 前端技术中心，诚招高级、资深技术专家，Base 上海、北京。我们致力于为美团海量业务建设高性能、高可用、高体验的前端基础技术服务，涵盖终端通信、终端安全、资源托管、可观测性、研发协同、设计工具、低代码、Node 基建等技术领域，欢迎有兴趣的同学投送简历至：edp.itu.zhaopin@meituan.com。