为什么需要动态iconfont

在应用iconfont之前，flutter开发者必须把大量的 图片资源放在工程内部，一定程度上会让编译包的体积增大，并且在不同的分辨率设备上有可能出现模糊的情况。

iconfont则可以把所有的图标资源统一管理，形成一个ttf文件，大大减少包体积，并且矢量图的特性使得图标在任何分辨率之下都**不会模糊**，而且还能自行定义图标的**渲染色**，避免了同一个图标不同颜色文件共存的情况。

但是，目前使用的iconfont的方式，一般都是将iconfont当成一个静态资源放在flutter工程内部，在yaml文件中注册，随后定义自己的IconFont类提供静态iconData常量给外部使用。有时候我们需要对某些图标进行修改，比如一个小改动，那么必须对app进行发版，流程很长。

想象一下，如果我们在线上对 iconfont的ttf文件进行托管，app负责与线上ttf进行同步，每次启动都能使用线上的最新资源，就能做到免发版（热更新都不用），就能直接更改生产环境上使用的图标。

实现动态iconfont的基础技术

官方支持

其实 flutter官方已经考虑到了这种情况，允许开发者在 app运行时动态加载 ttf资源，其核心 dart文件为 font\_loader.dart：

import 'dart:typed\_data';

import 'dart:ui';

import 'package:flutter/foundation.dart';

​

/// A class that enables the dynamic loading of fonts at runtime.

///

/// The [FontLoader] class provides a builder pattern, where the caller builds

/// up the assets that make up a font family, then calls [load] to load the

/// entire font family into a running Flutter application.

class FontLoader {

 /// Creates a new [FontLoader] that will load font assets for the specified

 /// [family].

 ///

 /// The font family will not be available for use until [load] has been

 /// called.

 FontLoader(this.family)

  : \_loaded = false,

     \_fontFutures = <Future<Uint8List>>[];

​

 /// The font family being loaded.

 ///

 /// The family groups a series of related font assets, each of which defines

 /// how to render a specific [FontWeight] and [FontStyle] within the family.

 final String family;

​

 /// Registers a font asset to be loaded by this font loader.

 ///

 /// The [bytes] argument specifies the actual font asset bytes. Currently,

 /// only OpenType (OTF) and TrueType (TTF) fonts are supported.

 void addFont(Future<ByteData> bytes) {

   if (\_loaded)

     throw StateError('FontLoader is already loaded');

​

   \_fontFutures.add(bytes.then(

      (ByteData data) => Uint8List.view(data.buffer, data.offsetInBytes, data.lengthInBytes),

  ));

}

​

 /// Loads this font loader's font [family] and all of its associated assets

 /// into the Flutter engine, making the font available to the current

 /// application.

 ///

 /// This method should only be called once per font loader. Attempts to

 /// load fonts from the same loader more than once will cause a [StateError]

 /// to be thrown.

 ///

 /// The returned future will complete with an error if any of the font asset

 /// futures yield an error.

 Future<void> load() async {

   if (\_loaded)

     throw StateError('FontLoader is already loaded');

   \_loaded = true;

​

   final Iterable<Future<void>> loadFutures = \_fontFutures.map(

      (Future<Uint8List> f) => f.then<void>(

          (Uint8List list) => loadFont(list, family),

      ),

  );

   await Future.wait(loadFutures.toList());

}

​

 /// Hook called to load a font asset into the engine.

 ///

 /// Subclasses may override this to replace the default loading logic with

 /// custom logic (for example, to mock the underlying engine API in tests).

 @protected

 @visibleForTesting

 Future<void> loadFont(Uint8List list, String family) {

   return loadFontFromList(list, fontFamily: family);

}

​

 bool \_loaded;

 final List<Future<Uint8List>> \_fontFutures;

}

解读一下它的核心逻辑：

构造函数 ： FontLoader(this.family) 必须提供一个family（String类型）实参对 不同的字体资源进行区分。

void addFont(Future<ByteData> bytes) addFont方法，加载外部字体文件的字节码，支持添加**多个**外部字体文件

Future<void> load() load方法，加载add好的 字体资源的字节码到内存中。值得注意的是, 这个 load方法使用了一个bool变量 \_loaded 防止了重复调用造成资源浪费。

Future<void> loadFont(Uint8List list, String family) 核心方法loadFont，加载字体资源的核心代码都在这里，进入观察一下：

/// Loads a font from a buffer and makes it available for rendering text.

///

/// \* `list`: A list of bytes containing the font file.

/// \* `fontFamily`: The family name used to identify the font in text styles.

/// If this is not provided, then the family name will be extracted from the font file.

Future<void> loadFontFromList(Uint8List list, {String? fontFamily}) {

 return \_futurize(

  (\_Callback<void> callback) {

     \_loadFontFromList(list, callback, fontFamily);

  }

).then((\_) => \_sendFontChangeMessage());

}

真正的加载过程是在native函数中：

void \_loadFontFromList(Uint8List list, \_Callback<void> callback, String? fontFamily) native 'loadFontFromList';

### 可运行Demo

目前Demo已经实现了读取线上iconfont并适用到运行中的程序。

关键代码如下：

import 'package:flutter/cupertino.dart';

import 'package:flutter/services.dart';

import 'package:http/http.dart' as http;

​

/// 动态iconfont

class DynamicIconFont {

 static const String iconFontURL = 'https://front-xps-cdn.xsyx.xyz/custom/day360/2022/04/19/2001349520.ttf';

 static const String \_family = 'dynamicIconFontFamily';

 static var fontLoader = FontLoader(\_family);

​

 static const IconData iconTest01 = IconData(0xe8c2, fontFamily: \_family);

 static const IconData iconTest02 = IconData(0xe8c3, fontFamily: \_family);

 static const IconData iconTest03 = IconData(0xe8c4, fontFamily: \_family);

​

 /// 获取 iconfont

 static Future<ByteData> \_fetchFont() async {

   try {

     final response = await http.get(Uri.parse(iconFontURL));

     if (response.statusCode == 200) {

       return ByteData.view(response.bodyBytes.buffer);

    } else {

       throw Exception('Failed to load font');

    }

  } catch (e) {

     throw Exception(e);

  }

}

​

 static Future<Object?>? addFont() async {

   try {

     fontLoader.addFont(\_fetchFont());

     await fontLoader.load();

  } catch (e) {

     debugPrint('addFont catch error: $e');

  }

   return true;

}

}

yaml中使用到的外部依赖只有 http: ^0.13.4

调用方，则只需要在启动时加上await DynamicIconFont.addFont();：

Future<void> main() async {

 WidgetsFlutterBinding.ensureInitialized();

 await DynamicIconFont.addFont();

 runApp(const MyApp());

}

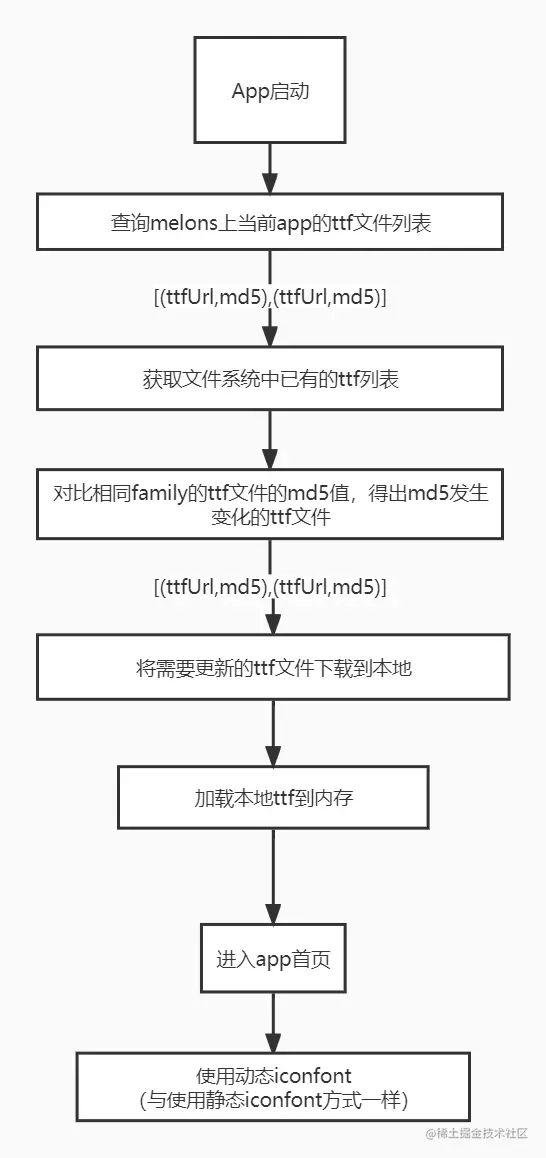
然后就能在任意位置使用动态iconfont中的图标了：

const Icon(DynamicIconFont.iconTest01, size: 33, color: Colors.greenAccent)

设计方案

初步方案

上一章节的Demo只是一个单机版的演示，要想在项目中运用到，则需要与后台联动，后台负责 字体资源托管，app与后台进行ttf文件同步，同步完毕之后将 最新ttf通过 上一节中的 fontLoader 加载到内存中，之后再进入到flutter主页，此后app就能使用到线上的最新字体资源了。上述思路具象化成流程图如下：



上面第一种方案，该方案可以快速实现iconfont动态化，每一次启动app都能使用最新字体资源。但是有几个问题：

1. App的启动速度会受到 melons请求接口响应时间的影响，如果接口响应慢，那么app的启动速度相应变慢，而且如果接口出现问题，app会白屏很长时间。App端必须设计出相应的接口容灾策略，在接口出现不同程度问题的情况下也能保证不错的体验，方案才算完整。
2. app可以同时加载多个 ttf到内存中，无论是否将来会使用到所有ttf。一定程度上造成了内存浪费，最理想的做法还是 ttf按需加载，一个ttf只加载一次。
3. 目前我们使用iconfont，基本上都是用 unicode为key，去读取图标，最多在自己的代码中写死一套 常量，让外部代码去使用常量，这种方案到了动态iconfont则不太适用，因为无法再写死常量了。

### 改良方案

* 针对问题1，目前没有太好的改良方向，只能做容灾策略，一旦app请求时发现接口问题，app提示 建议切换好点的网络或者转移到网络较好的地方重启app。
* 问题2，其实能够做到按需加载，解决方案为: 利用 FutureBuilder+UniqueKey 改造 Icon。

/// Icon的扩展方法，主要实现Icon组件的动态刷新

/// [dynamic] 方法主要通过[FutureBuilder]实现动态加载的核心原理

extension DynamicIconExtension on Icon {

 /// 用来监听新icon字体加载成功后的回调及时刷新icon，

 Widget get dynamic {

   /// 没有使用动态iconfont的情况下直接返回

   if (icon is! DynamicIconDataMixin) return this;

   final loadedKey = UniqueKey();

   return FutureBuilder(

     future: (icon as DynamicIconDataMixin).dynamicIconFont.addFont(icon!.codePoint),

     builder: (context, snapshot) {

       /// 由于icon的配置未发生变化但实际上其使用的字体已经发生了变化，所以这里通过使用不同的key让其强制刷新

       return KeyedSubtree(

         key: snapshot.hasData ? loadedKey : null,

         child: this,

      );

    },

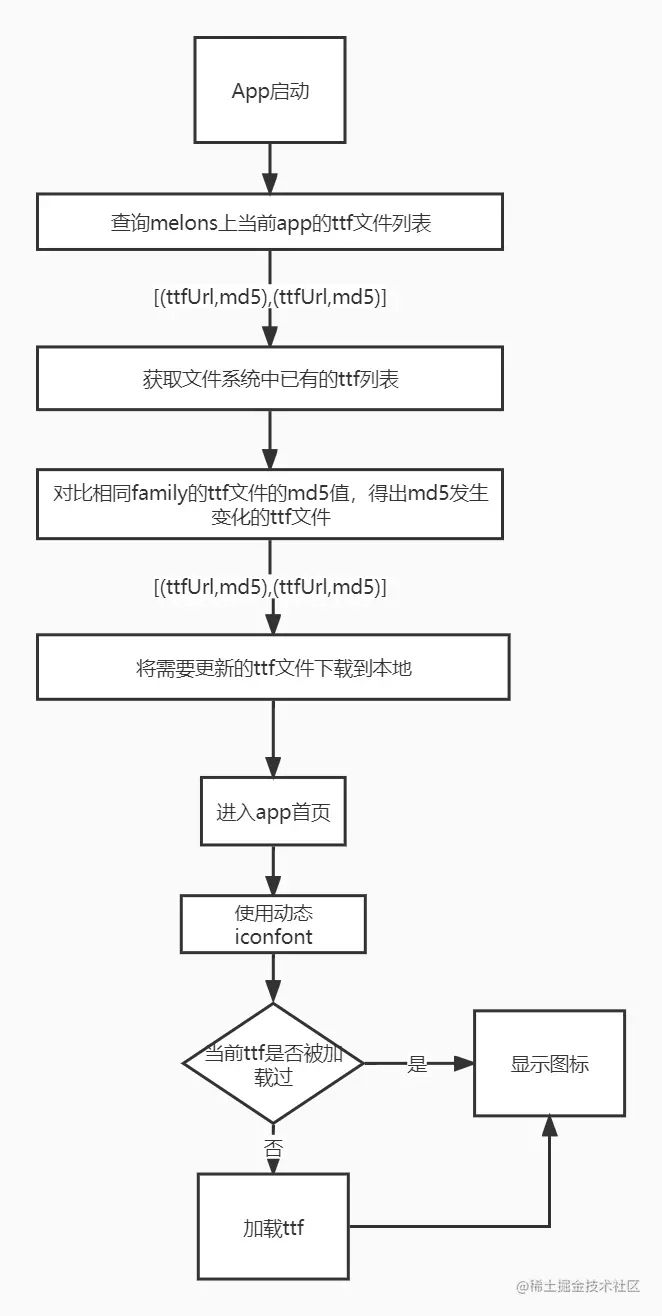
  );

}

}

上文中扩展了 Icon 类，提供了一个新的 dynamic成员变量，其中利用FutureBuilder的 future属性去指派要做的异步操作，也就是 加载ttf，在加载完毕之后，重新赋予key，强制它刷新自身。用到了哪个ttf，就只去加载这个字体，并且 fontLoader.load() 自身已经支持了避免重复load。这样，就不用每次都去一梭子先把ttf加载到内存中造成资源浪费。

这样，流程图就改成了，



问题3，优化开发者对于动态iconfont的使用体验，我们可以将 icon的unicode与name的映射下发到内存中，然后让开发者能够直接使用 family+name的组合去定位到一个确定的icon。

比如，以前的使用方式是这样的，

IconData iconTest01 = IconData(0xe677, fontFamily: family)

Icon(iconTest01);

现在你就可以这样用：

Icon(DynamicIconDataMixin('icon\_name', family)).dynamic;

托管平台支持

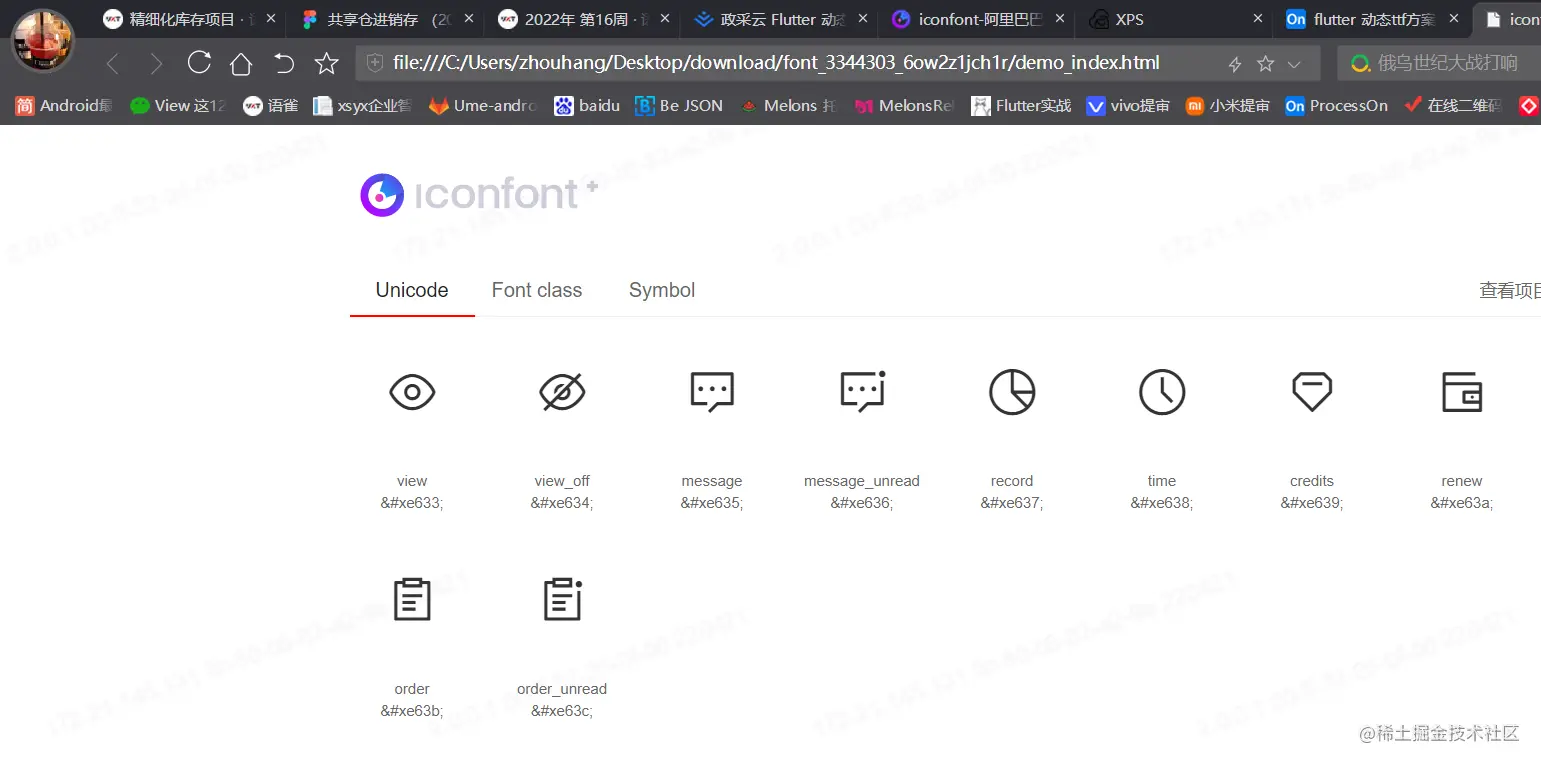
上面的都是app要做的事情，托管平台则需要负责以下工作：

上传iconfont资源（直接采用iconfont官网上下载的完整资源包作为单位去上传）

如下图所示，从iconfont官网上下载下来的资源包解压之后就这样：



使用浏览器打开html文件之后，就能看到此时这个ttf的描述页面：



而iconfont.json其实是，图形的对应关系：

{

 "id": "3344303",

 "name": "test002",

 "font\_family": "dynamicIconFontFamily",

 "css\_prefix\_text": "icon-",

 "description": "测试动态ttf的项目2",

 "glyphs": [

  {

     "icon\_id": "8776631",

     "name": "view",

     "font\_class": "icon-test",

     "unicode": "e633",

     "unicode\_decimal": 58931

  },

  {

     "icon\_id": "8776676",

     "name": "view\_off",

     "font\_class": "icon-test1",

     "unicode": "e634",

     "unicode\_decimal": 58932

  }

]

}

上传完成之后要能够展示 该资源下的所有 图标

资源包中有一个html文件，就上面那个html文件，提供一个链接打开它即可

下发ttf资源

app请求接口时需要传入 appId, 让每个app都能有自己的iconfont线上配置。返回值如下格式：

{

 "iconfonts": [

  {

     "family": "iconfont的业务代码，同一个app上可能同时存在多个iconfont用这个区分",

     "url": "http://www.xxxxxXXXXxxx.ttf",

     "md5": "用于校验文件正确性的MD5，也能作为文件的唯一标识区分",

     "glyphs": [

      {

         "name": "view",

         "unicode": "e633"

      },

      {

         "name": "view\_off",

         "unicode": "e634"

      }

    ]

  },

  {

     "family": "iconfont的业务代码，同一个app上可能同时存在多个iconfont用这个区分",

     "url": "http://www.xxxxxXXXXxxx.ttf",

     "md5": "用于校验文件正确性的MD5，也能作为文件的唯一标识区分",

     "glyphs": [

      {

         "name": "view",

         "unicode": "e633"

      },

      {

         "name": "view\_off",

         "unicode": "e634"

      }

    ]

  }

]

}

## 最终成果预期

预计最终会形成一个 纯flutter的SDK

提供一个初始化方法，阻塞执行，获取远端的iconfont最新配置

提供使用动态图标的新写法：

Icon(DynamicIconDataMixin('icon\_name', family)).dynamic;

iconFont 蜜汁操作，希望知道的解答一下

经过测试发现，下图中的font-family设置没发现什么作用，代码中使用ttf的时候，不管family写的是不是和设置中一样，都不影响图标的引用。很诡异。

