Chapitre 3 : Introduction aux widgets

▼ Qu'est ce qu'un widget ?

L'idée centrale de **Flutter** est de construire votre interface utilisateur en utilisant uniquement des **widgets**.

Les **widgets** sont totalement configurables et décrivent comment leur vue doit apparaître à l'écran.

La plupart des widgets implémentent une fonction constructrice qui va décrire les widgets de plus bas niveau qu'ils utilisent. Dans ce cas, ils sont appelés container widgets car ils contiennent d'autres widgets.

Cependant, certains **widgets** bas niveau ne font qu'afficher quelque chose, par exemple le **widget Text**.

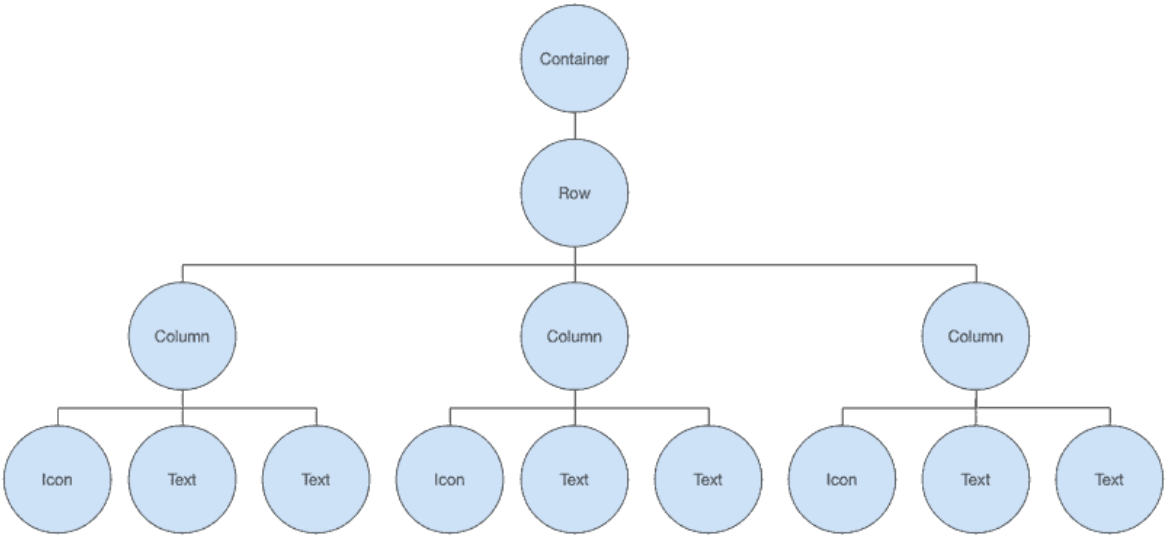
Il existe environ 150 **widgets** natifs créés par **Flutter** qui sont très configurable, et vous pouvez créer les vôtre comme nous le verrons.

L'arbre des widgets

Si vous venez d'un framework front, vous êtes familier de l'arbre des composants dont le fonctionnement est totalement similaire.

Les **widgets** sont organisés hiérarchiquement, avec un **widget** tout en haut appelé **widget racine**.

Voici à quoi ressemble un arbre :



Les widgets personnalisés avec ou sans état

Lorsque nous écrirons des applications, nous créerons des **widgets** personnalisés.

Tous les **widgets** personnalisés ont une fonction constructrice qui retourne un **widget**.

Ils héritent soit de la classe **StatelessWidget**, soit de la classe **StatefulWidget**.

La première classe sera réservée aux widgets **qui n'ont pas à gérer d'état**.

Ils ne sont dessinés sur le canvas qu'une seule fois lorsqu'ils sont chargés. Cela signifie qu'ils ne peuvent pas être rafraîchis si un événement ou une action utilisateur survient.

La deuxième classe sera réservée aux **widgets qui doivent être redessinés plusieurs fois**.

▼ Les propriétés des widgets

Les widgets sont des classes

Tous les widgets que nous utiliserons sont des instances de classes qui héritent de la classe **Widget**.

Les widgets **sont toujours configurables** avec des paramètres que nous leur passons.

Voici un exemple :

```
Text(
  'Bonjour, $_prenom ! Comment ça va ?',
  textAlign: TextAlign.center,
  overflow: TextOverflow.ellipsis,
  style: TextStyle(fontWeight: FontWeight.bold),
)
```

Nous étudierons en détails le **widget Text**, mais nous pouvons déjà faire plusieurs remarques.

Nous créons une instance de la classe **Text** (rappelez-vous qu'en Dart, le new est facultatif).

Les widgets sont configurés en passant des paramètres

Nous passons en premier paramètre une chaîne de caractères, puis un certain nombre de paramètres nommés pour configurer le **widget**.

Nous remarquons également l'utilisation d'**enum** pour toutes les propriétés nommés, par exemple **TextAlign.center**.

Nous verrons que cela permet de ne jamais se tromper sur le nom des propriétés et également de voir toutes les propriétés disponibles grâce à l'autocomplétion de votre éditeur de code.

▼ Le widget Text

Le code minimal permettant d'afficher quelque chose à l'écran est le suivant :

```
import 'package:flutter/material.dart';

void main() {
  runApp(
    const Center(
      child: Text(
        'Hello, world !',
        textDirection: TextDirection.ltr,
      ),
    ),
  );
}
```

Vous pouvez tester en remplaçant le contenu de main.dart dans un nouveau projet Flutter, puis en faisant :

```
flutter run
```

Nous allons étudier le code généré à la création d’un projet flutter.

La fonction runApp

La fonction **runApp()** prend en paramètre un Widget qui est en fait le **widget racine de votre application**. Il est au **sommet** de l'arbre des widgets.

Il ne faut donc utiliser cette fonction qu'une seul fois dans votre application, et lui passer le **widget** le plus haut dans l'arbre.

Arbre des widgets

Dans notre cas nous avons un **widget Center** comme widget racine.

Il a un seul widget enfant dans la clé **child** : un **widget Text**.

```
import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
      return const Center(
        child: Text(
          'Hello, world !',
      );
  }
}
```

Le widget Center

Nous le reverrons en détails, mais le **widget Center** permet de centrer le **widget enfant** passé en paramètre nommé **child**.

Le widget Text

Le **widget Text** permet d'afficher une chaîne de caractères en utilisant un seul style.

Le constructeur de la classe **Text** est le suivant :

```
Text(
    this.data, {
    Key key,
    this.style,
    this.strutStyle,
    this.textAlign,
    this.textDirection,
    this.locale,
    this.softWrap,
    this.overflow,
    this.textScaleFactor,
    this.maxLines,
    this.semanticsLabel,
  })
```

Nous voyons qu'il peut être configuré avec de nombreuses propriétés.

this.data est la chaîne de caractères passée en premier paramètre, elle ne peut être nulle.

Tous les autres paramètres sont nommés et permettent de configurer l'affichage du texte.

Le paramètre style

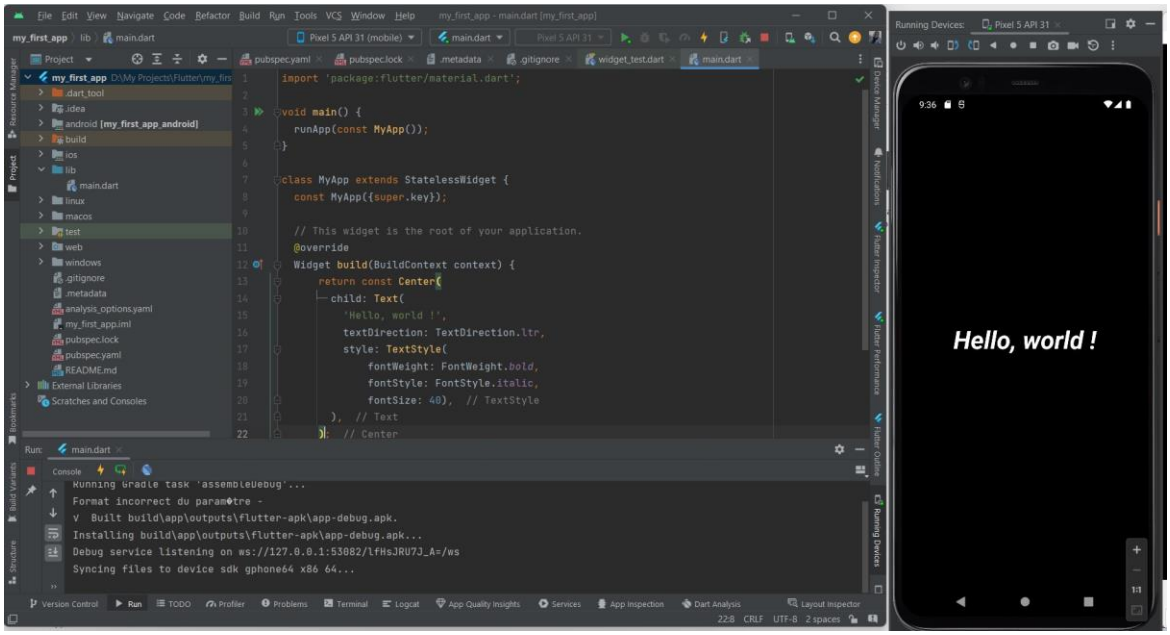
Nous allons commencer par le style que nous pouvons passer avec le paramètre nommé **style**.

Le **style** peut recevoir un **TextStyle** qui a pour constructeur :

```
const TextStyle({
    this.inherit = true,
    this.color,
    this.backgroundColor,
    this.fontSize,
    this.fontWeight,
    this.fontStyle,
    this.letterSpacing,
    this.wordSpacing,
    this.textBaseline,
    this.height,
    this.locale,
    this.foreground,
    this.background,
    this.shadows,
    this.decoration,
    this.decorationColor,
    this.decorationStyle,
    this.decorationThickness,
    this.debugLabel,
    String fontFamily,
    List<String> fontFamilyFallback,
    String package,
  })
```

Vous pouvez ainsi rendre le texte en italique, en gras et avec une taille de 40 en faisant :

```
import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
      return const Center(
        child: Text(
          'Hello, world !',
          textDirection: TextDirection.ltr,
          style: TextStyle(
              fontWeight: FontWeight.bold,
              fontStyle: FontStyle.italic,
              fontSize: 40),
        ),
      );
  }
}
```



Avec un style un peu plus complexe :

```
import 'package:flutter/material.dart';
```

```
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
      return Center(
        child: Text(
          'Hello, world !',
          textDirection: TextDirection.ltr,
          style: TextStyle(
              fontWeight: FontWeight.bold,
              fontStyle: FontStyle.italic,
              fontSize: 50,
              backgroundColor: Colors.purple,
              foreground: Paint()
                ..color = const Color(0xffA8CBFD)
                ..strokeWidth = 2
                ..style = PaintingStyle.stroke,
          ),
        ),
      );
  }
}
```

Nous passons en outre, la couleur du fond à rose en utilisant l'une des nombreuses couleurs disponibles en **enum** sur la classe **Colors**.

Nous définissons également la propriété **foreground** qui prend une instance de la classe **Paint()** en paramètre.

Cette propriété permet de styliser le texte en premier plan (par opposition au fond, le **background**).

Cette classe très bas niveau permet de définir le style à utiliser pour dessiner quelque chose sur le **Canvas**.

Nous utilisons ensuite **la notation en cascade** `..` qui **permet d'effectuer une séquence d'opération sur le même objet.**

Il s'agit d'une notation raccourcie permettant de ne pas devoir définir de variable intermédiaire et de ne pas répéter la cible des opérations.

Ainsi :

```
foreground: Paint()
              ..color = const Color(0xffA8CBFD)
              ..strokeWidth = 2
              ..style = PaintingStyle.stroke,
```

Equivaut à :

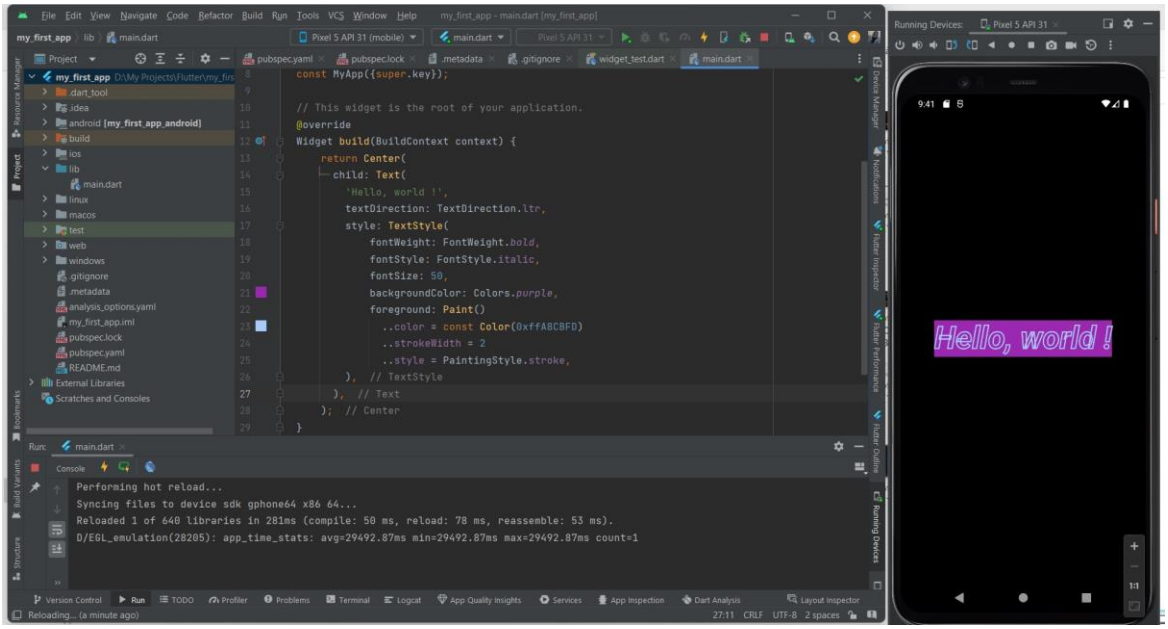
```
var p = Paint();
p.color = const Color(0xffA8CBFD);
p.strokeWidth = 2;
p.style = PaintingStyle.stroke;
```

```
...

foreground: p
```

Ainsi :

Nous configurons donc la couleur du texte, son remplissage (ici non rempli en utilisant `PaintingStyle.stroke`, l'inverse étant `PaintingStyle.fill`), ainsi que l'épaisseur du trait.

Voici le résultat :



Il est également possible de paramétrer de nombreuse choses comme l'espacement entre les lettres (police (`fontFamily`) etc.

Prenons un autre exemple en utilisant `decoration` :

```
import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    var p = Paint();
    p.color = const Color(0xffA8CBFD);
    p.strokeWidth = 2;
    p.style = PaintingStyle.stroke;
      return Center(
        child: Text(
          'Hello, world !',
          textDirection: TextDirection.ltr,
          style: TextStyle(
            fontWeight: FontWeight.bold,
```

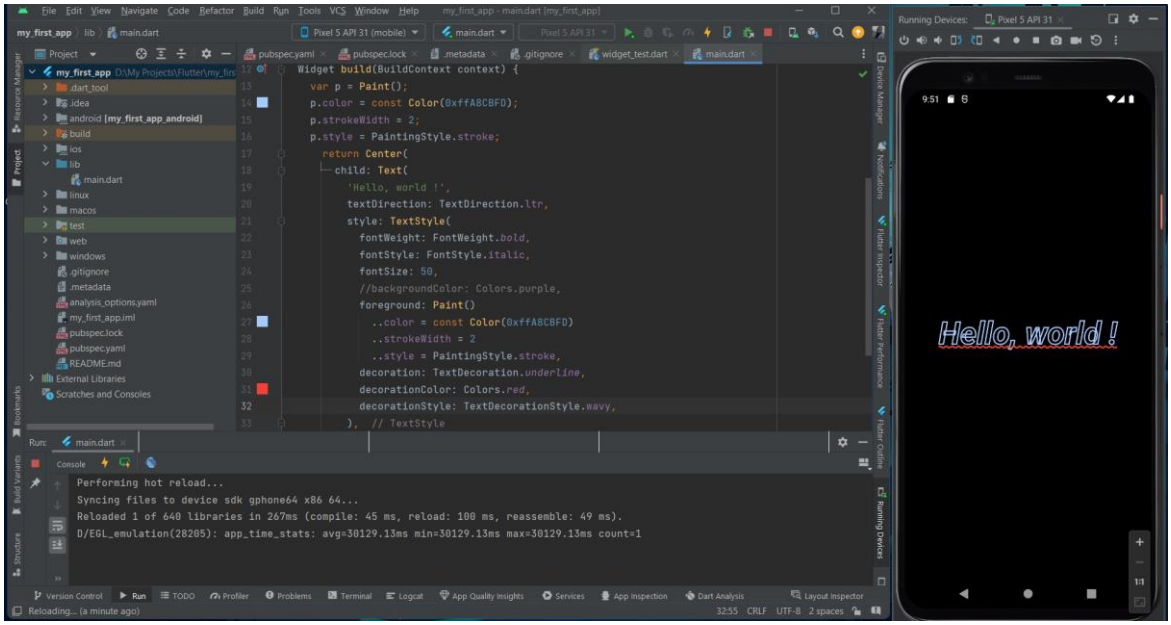
`letterSpacing`), entre les mots (`wordSpacing`), des effets d'ombre complexes (`shadows`), la

```
            fontStyle: FontStyle.italic,
            fontSize: 50,
            //backgroundColor: Colors.purple,
            foreground: Paint()
              ..color = const Color(0xffA8CBFD)
              ..strokeWidth = 2
              ..style = PaintingStyle.stroke,
            decoration: TextDecoration.underline,
            decorationColor: Colors.red,
            decorationStyle: TextDecorationStyle.wavy,
          ),
        ),
      );
  }
}
```

La `decoration` permet de souligner, de barrer ou de placer une barre au dessus d'un mot. Ici, nous le soulignons (`underline`)

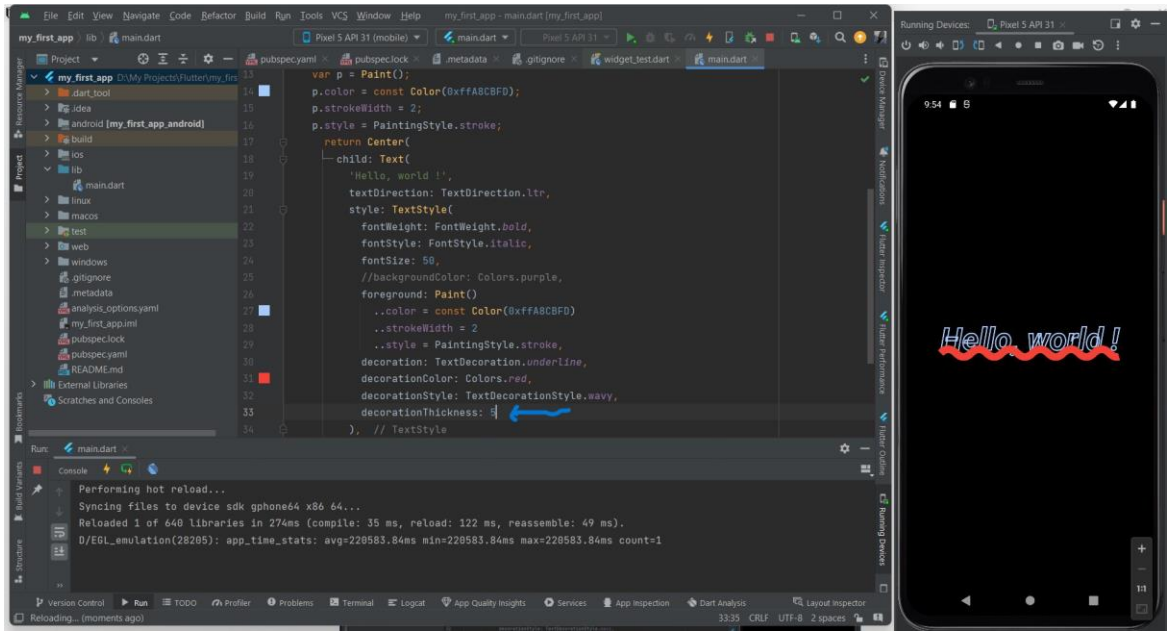
Nous spécifions ensuite la couleur de la décoration et lui attribuons un style avec `decorationStyle` (`wavy` pour vaguelettes). Les autres valeurs possibles sont `double` , `dashed` , `dotted` et `solid` .

Voici le résultat :



Nous pourrions également utiliser `decorationThickness` pour spécifier en utilisant un double, la taille du trait de la décoration.

Voici le résultat :



Utilisation de textAlign

Ce paramètre nommé permet de contrôler l'alignement horizontal du texte.

Il peut prendre comme valeur : `center` , `end` , `justify` , `left` , `right` et `start` .

Example: `textAlign: TextAlign.left,`

> NB: Dans notre cas ceci n'aura aucun n'effet puisque notre Text Widget à comme parent le Widget Center qui va le forcer à s'aligner au milieu

Utilisation de textDirection

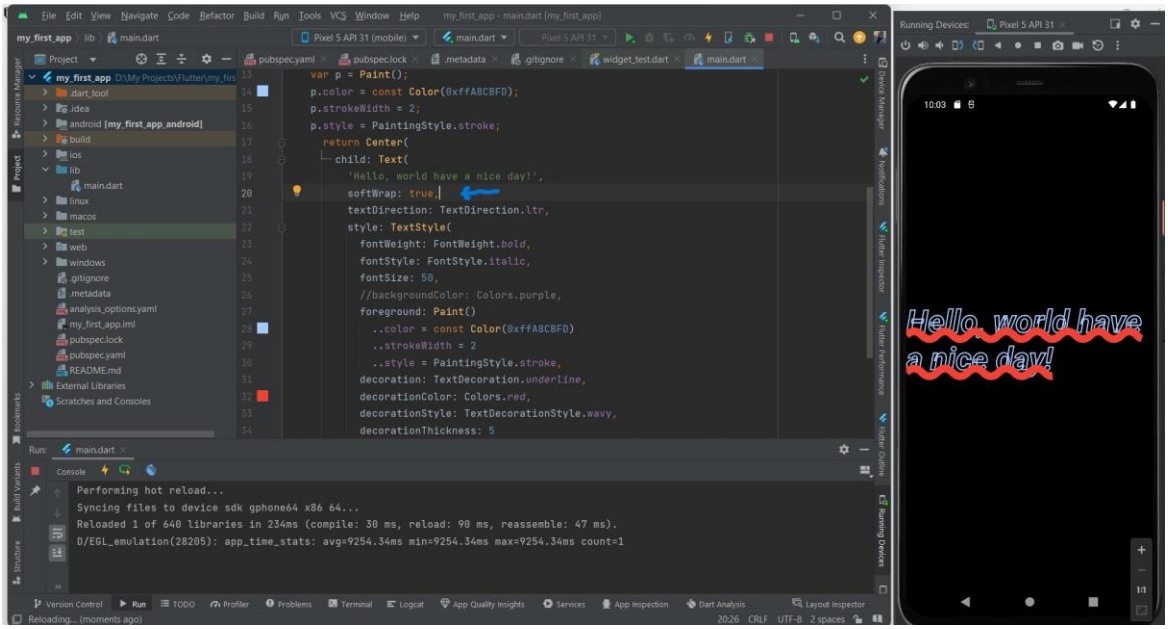
Ce paramètre nommé permet de contrôler la direction du texte : c'est-à-dire gauche à droite (français, anglais etc) ou droite à gauche (arabe, hébreux etc).

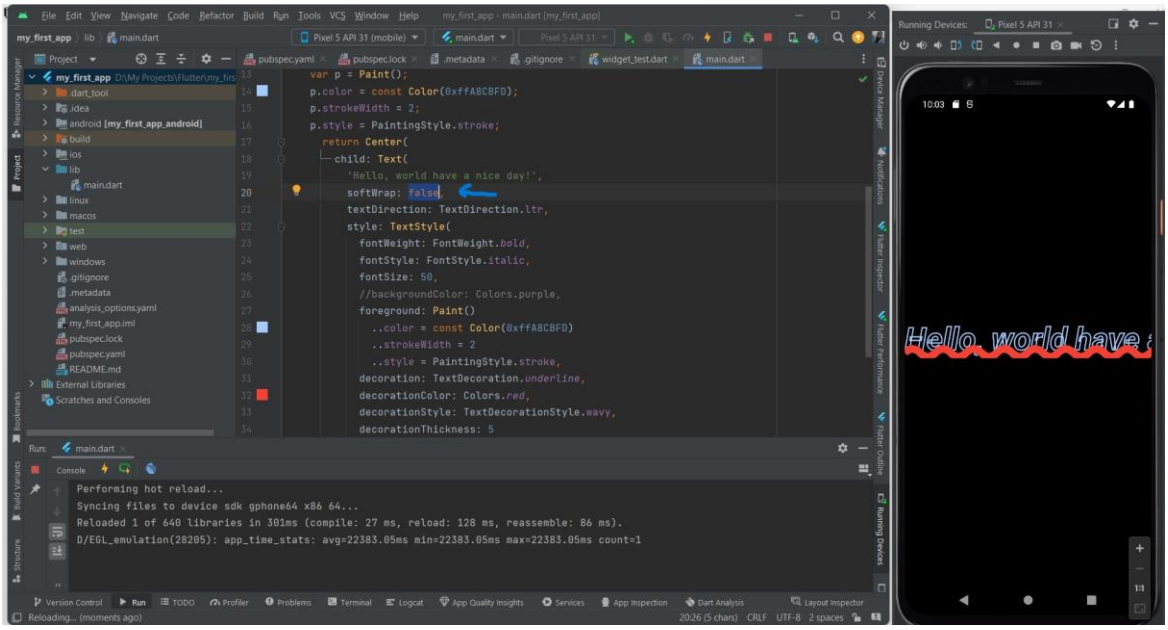
Il peut prendre deux valeurs `ltr` (pour `left to right`) ou rtl (pour `right to left`).

Utilisation de softWrap

Ce paramètre nommé permet de contrôler si il faut aller à la ligne (`wrap`) ou ne pas afficher la totalité si il n'y a pas assez de place.

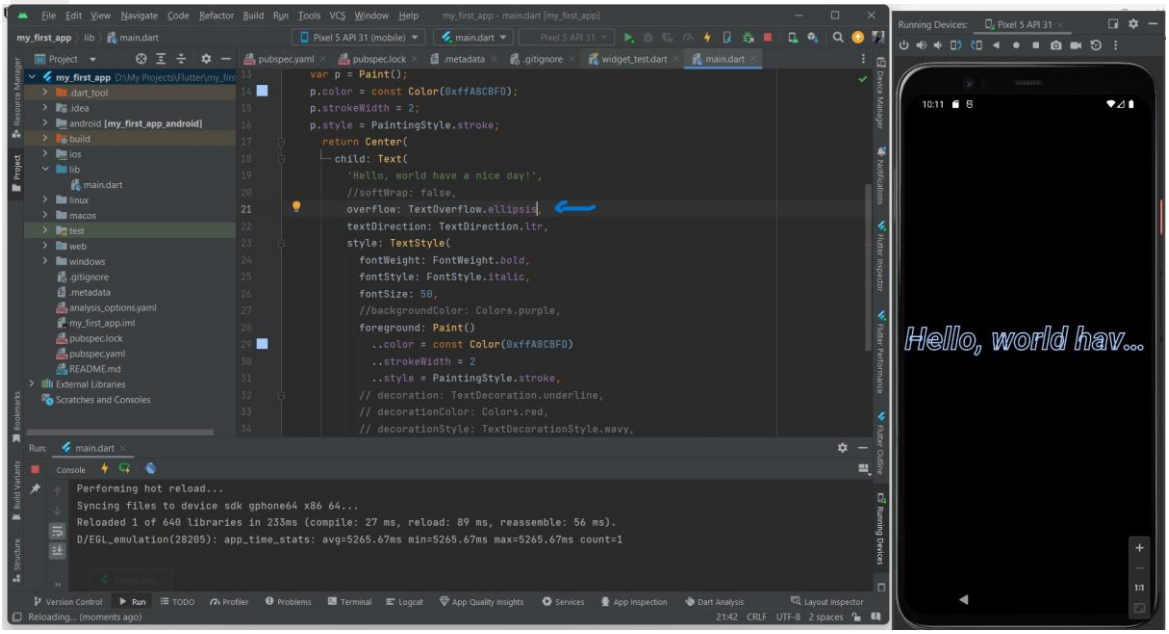
Il prend un booléen avec `true` pour aller à la ligne ou `false` pour ne pas afficher le texte si il n'y a pas assez de place





Utilisation de overflow

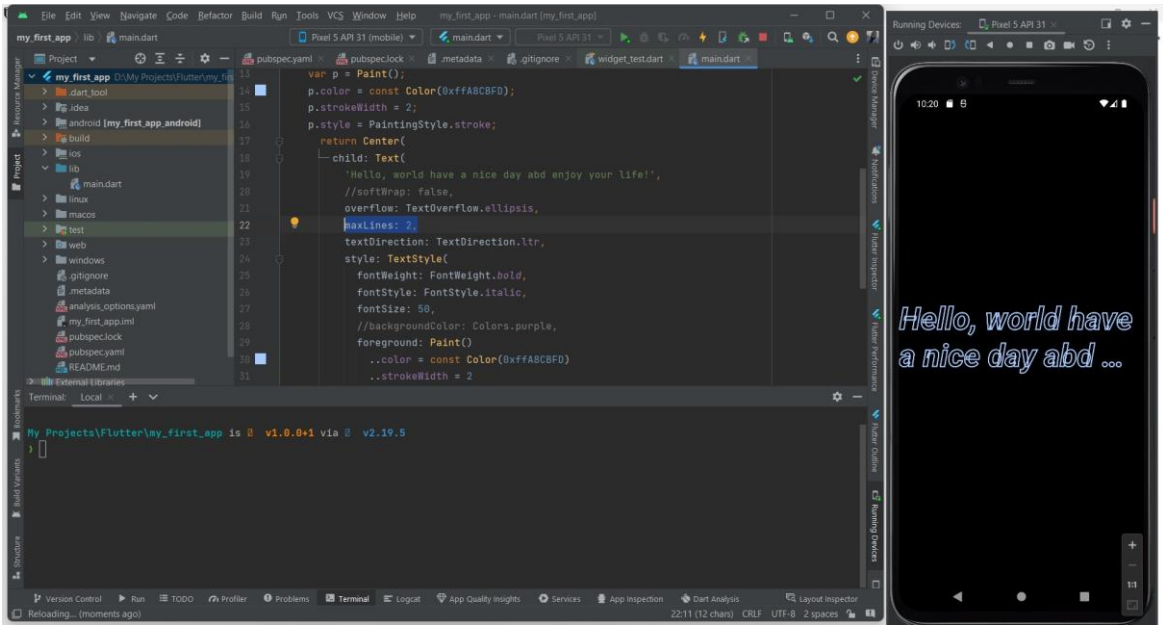
Ce paramètre nommé permet de contrôler `l'overflow` , c'est à dire comment gérer le dépassement du texte autrement qu'aller à la ligne : par exemple le couper avec … (`ellipsis`), le faire dépasser du `widget container` si il y en a un (`visible`), faire un effet de décoloration (`fade`) ou le couper (`clip`).

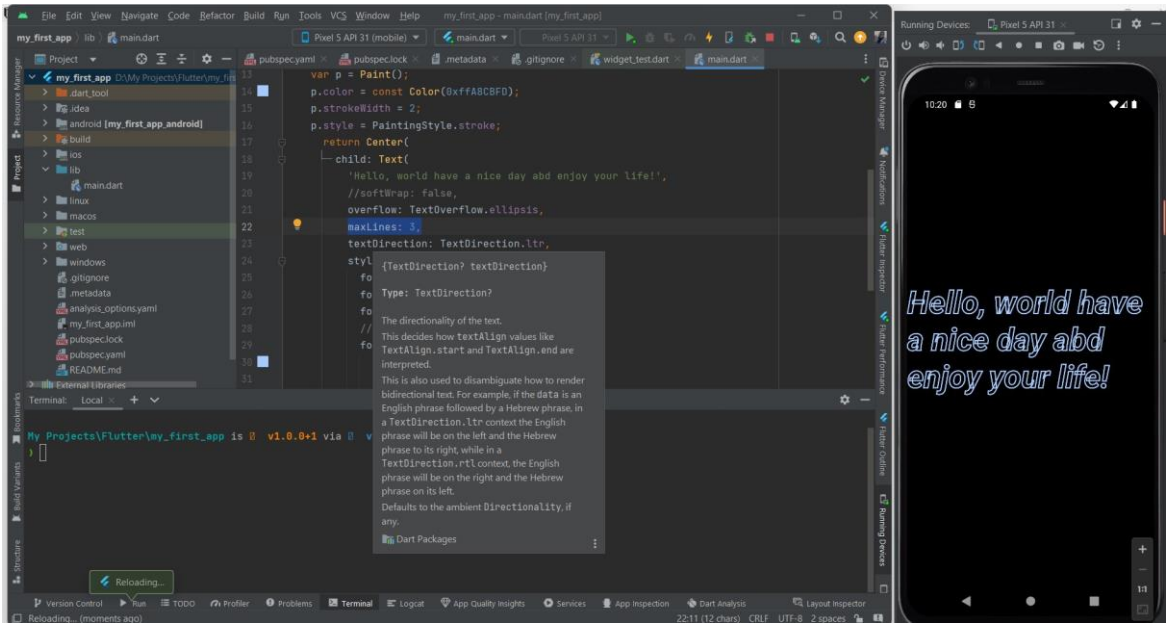


Utilisation de maxLines

Ce paramètre permet de passer un `int` qui est le nombre maximal de lignes d'un texte.

Si le nombre est supérieur à 1, le texte ira à la ligne jusqu'au nombre maximal de lignes spécifié. Ensuite, il sera coupé en respectant l'`overflow` spécifié.





Le constructeur nommé Text.rich

Si vous voulez configurer de manière complexe un texte, en lui appliquant différents styles suivant les caractères etc, il vous faudra utiliser le constructeur nommé `Text.rich()` .

Ce constructeur prend en **premier paramètre positionnel un élément** `TextSpan` .

Il prend ensuite en **paramètres nommés** les mêmes que vue précédemment.

Dans un `TextSpan` vous pouvez passer uniquement **des paramètres nommés**.

Les principaux sont `style` , `text` et `children` :

- Le `style` permet d'appliquer un `TextStyle` à tous les éléments `children` et au `text` .
- `text` permet d'afficher un texte auquel sera appliqué les paramètres nommés passés à `TextSpan` ou à `Text.rich()` .
- Les `children` sont une `list` de `TextSpan` qui prennent un paramètre nommé `text` et auxquelles vous pouvez appliquer des `style` différents.

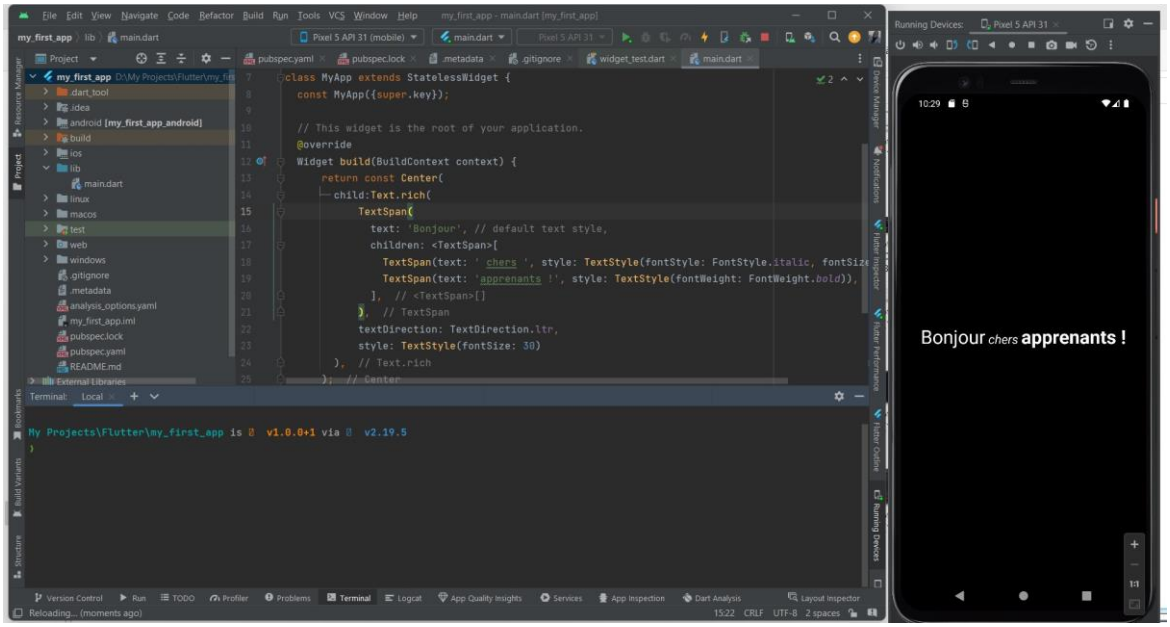
Prenons un exemple :

```
import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
      return const Center(
        child:Text.rich(
            TextSpan(
              text: 'Bonjour', // default text style,
              children: <TextSpan>[
                TextSpan(text: ' chers ', style: TextStyle(fontStyle: FontStyle.italic, fontSize: 20)),
```

```
                TextSpan(text: 'apprenants !', style: TextStyle(fontWeight: FontWeight.bold)),
              ],
            ),
            textDirection: TextDirection.ltr,
            style: TextStyle(fontSize: 30)
        ),
      );
  }
}
```

Voici le résultat :

Les paramètres



`style` et `textDirection` passés à `Text.rich()` seront appliqués à tous les `widgets enfants` , **sauf si d'autres paramètres sont appliqués à plus bas niveau**.

En fait, les paramètres définis sont appliqués par défaut à tous les éléments enfants. Mais ils sont remplacés par des paramètres appliqués à des `widgets` de plus bas niveau.

C'est ici le cas pour le `TextSpan` qui contient `chers` : nous lui passons un paramètre nommé `style` et nous remplaçons la `fontSize` utilisée.

Le widget Image

Le `widget Image` permet d'afficher une image avec `Flutter` .

Il existe plusieurs sources pour une image :

- `asset` : Dans un dossier qui sera mis dans un bundle assets lors de la compilation
- `network` : En utilisant une URL et être connecté sur le réseau internet, `file` : En utilisant un fichier `memory` : En utilisant une image depuis la mémoire.

Parmi ces quatre possibilités on préfère le plus souvent utiliser l' `asset` ou le `network`

La source asset

Dans `Flutter` , les `assets` sont gérés dans le fichier `pubspec.yaml` .

Pour inclure des `assets` dans votre application, commencez par créer un dossier `assets` au même niveau que le fichier `pubspec.yaml` .

Pour les images, il est recommandé de créer un dossier `images` dans `assets` .

Utilisez le lien qui suit pour télécharger des images:

Image libre de droit, banque d'image gratuite et photos gratuites

Images libres de droit et des vidéos gratuites à utiliser partout.✓Haute qualité ✓Entièrement gratuite ✓Aucune attribution nécessaire

https://www.pexels.com/fr-fr/

Vous pouvez ensuite inclure toutes les images en indiquant dans `pubspec.yaml` :

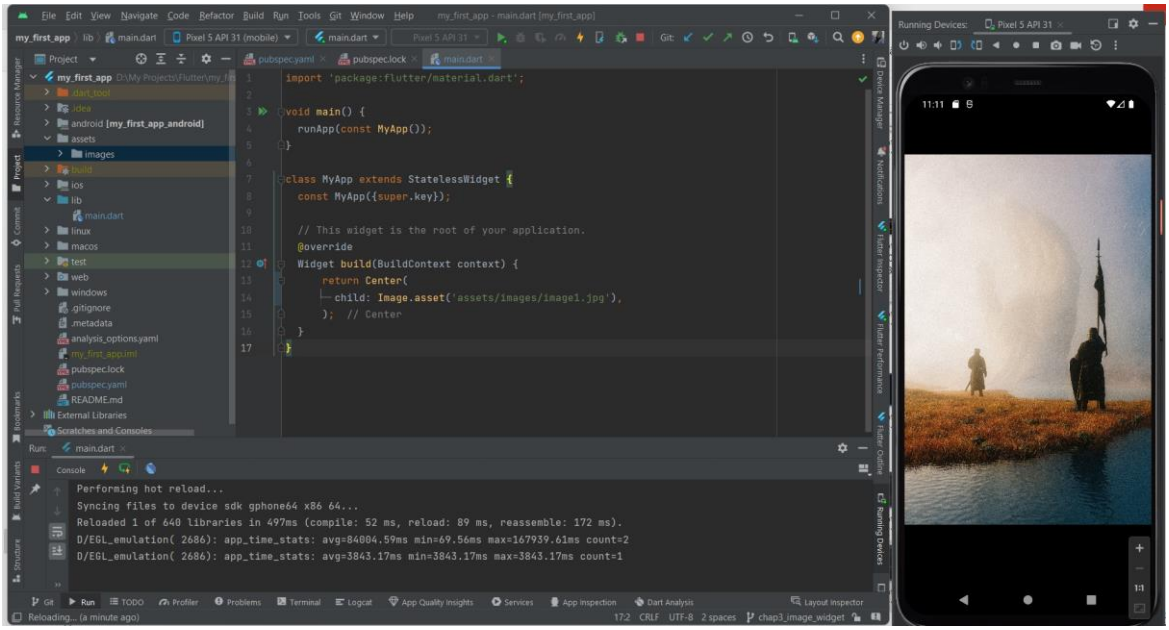
```
flutter:
  assets:
    - assets/images/
```

Ensuite pour utiliser l'image dans un widget de votre application, il suffit d'utiliser le constructeur nommé Image.asset() :

```
import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
      return Center(
        child: Image.asset('assets/images/image1.jpg'),
      );
  }
}
```

> NB: A chaque fois que vous ajoutez une nouvelle image vous devez impérativement recompiler votre application pour pouvoir l'utiliser

Voici le résultat :



La source network

Cette source permet de charger une image à partir d'une URL réseau. Vous devez fournir l'URL de l'image à charger.

Pour uploader une image en ligne vous pouvez utiliser le lien suivant:

> ImgBB — Upload Image — Free Image Hosting
>
> Free image hosting and sharing service, upload pictures, photo host. Offers integration solutions for uploading images to forums.
>
> https://imgbb.com/

Pour ce faire vous pouvez utilisez deux propriétés `Image.network` et `NetworkImage`

`Image.network` est un widget Flutter qui affiche une image à partir d'une URL réseau spécifiée. Vous pouvez l'utiliser directement dans votre arborescence de widgets pour afficher l'image.

`NetworkImage` est une classe dans Flutter qui représente une image chargée à partir d'une URL réseau. Elle est utilisée comme argument dans le constructeur du widget `Image` pour spécifier la source de l'image.

En résumé, `Image.network` est le widget qui utilise `NetworkImage` en interne pour charger et afficher une image à partir d'une URL réseau. Vous pouvez utiliser directement `Image.network` si vous souhaitez afficher une image à partir d'une URL réseau sans avoir à créer manuellement une instance de `NetworkImage` . Si vous avez besoin d'une instance de `NetworkImage` pour une autre utilisation, vous pouvez l'utiliser comme argument dans le constructeur du widget `Image` .

Voici un exemple en utilisant le constructeur nommé `NetworkImage` :

```
import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
```

```
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
      return const Center(
        child: Image(
          image: NetworkImage('https://i.ibb.co/HG70pXB/image4.jpg'),
        ),
      );
  }
}
```

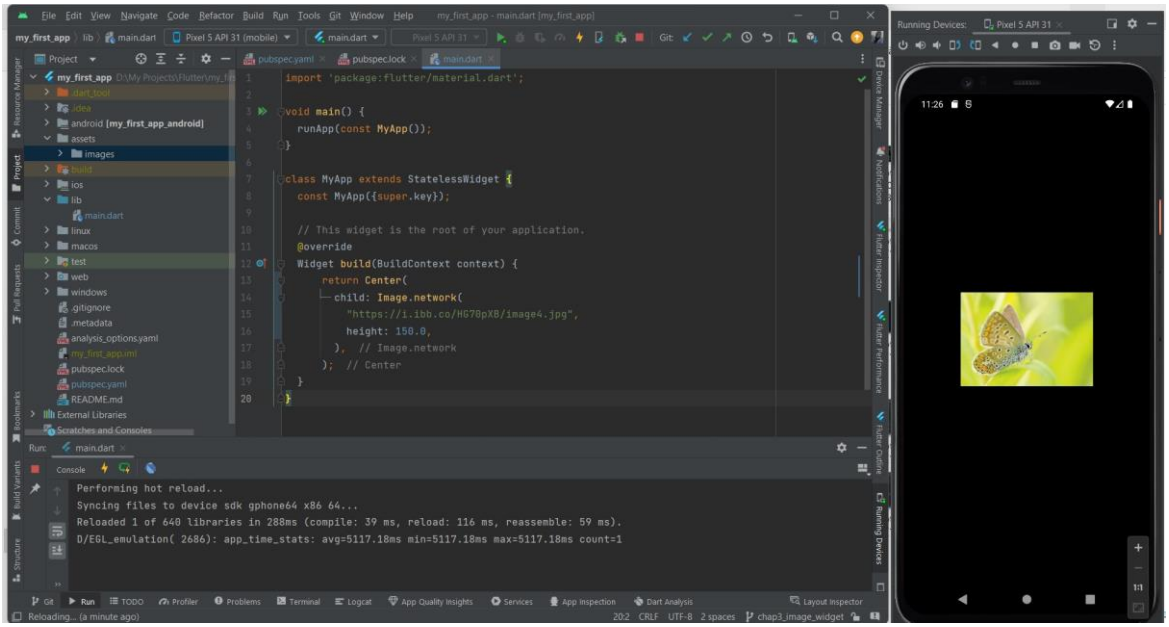
Voici un exemple en utilisant le constructeur nommé `Image.network` :

```
import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
      return Center(
        child: Image.network("https://i.ibb.co/HG70pXB/image4.jpg"),
      );
  }
}
```

Vous pouvez spécifier plusieurs paramètres nommés que nous verrons dans la formation, les principaux que vous pourriez avoir besoin sont `height` et `width` :

```
import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
      return Center(
        child: Image.network(
          "https://i.ibb.co/HG70pXB/image4.jpg",
          height: 150.0,
        ),
      );
  }
}
```

Ce qui donne :


import 'package:flutter/material.dart';
void main() {
runApp(const MyApp());
}
class MyApp extends StatelessWidget {
const MyApp({super.key});
// This widget is the root of your application.
@override
Widget build(BuildContext context) {
return Center(
child: Image.network(
"https://i.ibb.co/HG70pX8/image4.jpg",
height: 150.0,
), // Image.network
); // Center
}
}
Performing hot reload...
Syncing files to device sdk gphone64 x86 64...
Reloaded 1 of 640 libraries in 288ms (compile: 39 ms, reload: 116 ms, reassemble: 59 ms).
D/EGL_emulation(2686): app_time_stats: avg=5117.18ms min=5117.18ms max=5117.18ms count=1
11:26