Comment fonctionne YYText

L'effet d'ombre ci-dessus est réalisé avec le code suivant :

On peut voir que `YYTextShadow` est d'abord généré, puis assigné à `yy_textShadow` de `attributedString`. Ensuite, `attributedString` est assigné à `YYLabel`, et enfin, `YYLabel` est ajouté à `UIView` pour être affiché. En suivant `yy_textShadow`, on découvre que cela consiste principalement à lier `textShadow` à l'attribut de `NSAttributedString`, avec la clé `YYTextShadowAttributeName` et la valeur `textShadow`. Cela signifie que le shadow est d'abord stocké, puis utilisé ultérieurement. Utilisez Shift + Command + J pour accéder rapidement à la définition :

Il y a ici une fonction `addAttribute`, qui est définie dans le fichier `NSAttributedString.h` :

```
- (void)addAttribute:(NSString *)name value:(id)value range:(NSRange)range;
```

Cela signifie que vous pouvez lui assigner n'importe quelle paire clé-valeur. La définition de `YYTextShadowAttributeName` est une chaîne de caractères ordinaire, ce qui signifie que les informations de l'ombre sont d'abord stockées, puis utilisées ultérieurement. Faisons une recherche globale de `YYTextShadowAttributeName`.

Ensuite, nous arrivons à la fonction `YYTextDrawShadow` dans `YYTextLayout` :

`CGContextTranslateCTM` est utilisé pour modifier les coordonnées de l'origine dans un contexte, donc...

```
CGContextTranslateCTM(context, point.x, point.y);
```

Cela signifie qu'il faut déplacer le contexte de dessin vers le point `point`. Nous devons d'abord comprendre où `YYTextDrawShadow` est appelé, et nous constatons qu'il est appelé dans `drawInContext`.

Dans `drawInContext`, dessinez successivement la bordure du bloc, puis la bordure de fond, l'ombre, le soulignement, le texte, les accessoires, l'ombre intérieure, la ligne de suppression, la bordure du texte et les lignes de débogage.

Alors, où est-ce que `drawInContext` est utilisé exactement ? On peut voir qu'il y a un paramètre `YYTextDebugOption`, donc cette fonction n'est certainement pas un rappel système, mais plutôt une fonction appelée par YYText lui-même.

Maintenez la touche Ctrl + 1 enfoncée pour afficher les raccourcis clavier, et vous constaterez qu'il y a quatre endroits où ils sont utilisés.

`drawInContext:size:debug` reste un appel propre à YYText, car le type de `debug` est `YYTextDebugOption *`, qui est spécifique à YY. `newAsyncTask` ne ressemble pas à un appel système, et `addAttachmentToView:layer:` est similaire, donc il est fort probable que ce soit `drawRect:`.

En effet, en regardant l'aide rapide à droite, il y a une explication détaillée, et en dessous de l'aide, il est également indiqué que cela est défini dans `UIView`. Ensuite, en examinant `YYTextContainerView`, on voit qu'il hérite de `UIView`.

Donc `YYLabel` utilise `YYTextContainerView`, c'est ça ? Et ensuite, il laisse le système appeler `drawRect:` dans `YYTextContainerView` pour dessiner ?

Étrange, `YYLabel` hérite de `UIView`. Donc, dans YYText, il devrait y avoir deux ensembles de choses ! Un ensemble `YYLabel`, et un ensemble `YYTextView`, comme `UILabel` et `UITextView`. Ensuite, revenons à la méthode `newAsyncDisplayTask` de `YYLabel` que nous avons vue précédemment,

Très long, au milieu, il appelle `drawInContext` dans `YYTextLayout. newAsyncDisplayTask`, où est-il appelé à son tour ?

Sur la deuxième ligne, il est appelé. On peut donc simplement comprendre que `YYLabel` utilise un processus asynchrone pour dessiner le texte. Et `_displayAsync` est appelé par la méthode `display` ci-dessus. En regardant la documentation de `display`, il est dit que le système l'appellera au moment approprié pour mettre à jour le contenu du layer, et il ne faut pas l'appeler directement. Nous pouvons également y ajouter un point d'arrêt.

L'explication est que `display` est appelé dans une transaction de `CALayer`. Pourquoi utiliser une transaction ? Probablement pour mettre à jour en masse, ce qui serait plus efficace, non ? Cela ne ressemble pas à un besoin de rollback comme dans une base de données.

La documentation système de `display` mentionne également que si vous souhaitez que votre layer soit dessiné différemment, vous pouvez surcharger cette méthode pour implémenter votre propre dessin.

Ainsi, nous avons une idée simple. `YYLabel` utilise la méthode `display` de `UIView` pour dessiner de manière asynchrone ses effets tels que les ombres, etc. Les effets d'ombre sont d'abord stockés dans les attributs de `attributedText` de `YYLabel`, puis récupérés lors du dessin dans la méthode `display`. Pour le dessin, le framework CoreGraphics du système est utilisé.

Après avoir clarifié certaines idées, on se rend compte de ce qui est vraiment puissant : d'un côté, c'est la capacité à organiser autant d'effets, d'appels asynchrones, etc., et de l'autre, c'est la maîtrise approfondie du framework CoreGraphics sous-jacent. Ainsi, après avoir acquis une certaine compréhension de l'organisation du code précédent, nous allons maintenant

plonger plus profondément dans le framework CoreGraphics. Voyons comment le dessin est effectué.

Revenons à `YYTextDrawShadow`.

Ici, `CGContextSaveGState` et `CGContextRestoreGState` encadrent un bloc de code de dessin. `CGContextSaveGState` signifie que l'état actuel du dessin est copié et placé dans la pile de dessin. Chaque contexte de dessin maintient une pile de dessin. Je ne suis pas sûr de la manière exacte dont la pile est gérée. Pour l'instant, comprenons simplement qu'il faut appeler `CGContextSaveGState` avant de dessiner dans le contexte, et `CGContextRestoreGState` après, afin que le dessin intermédiaire apparaisse correctement dans le contexte. `CGContextTranslateCTM` déplace le contexte à une position spécifique. Il se déplace d'abord à `point.x` et `point.y`, les coordonnées de dessin, puis à 0 et `size.height`, ce qui n'est pas clair pour le moment, nous verrons cela plus tard. Ensuite, les `lines` sont récupérées et une boucle `for` est exécutée.

`lines` est un tableau qui contient les lignes de texte générées lors de la création d'une mise en page de texte (`YYTextLayout`). Dans la méthode `(YYTextLayout *)layoutWithContainer:(YYTextContainer *)container text:(NSAttributedString *)text range:(NSRange)range`, ce tableau est rempli avec les lignes de texte qui sont calculées en fonction du conteneur (`container`), du texte (`text`) et de la plage spécifiée (`range`).

Chaque élément du tableau `lines` représente une ligne de texte dans la mise en page, et ces lignes sont utilisées pour afficher le texte correctement formaté à l'écran.

Ensuite, naviguez jusqu'à la définition de cette fonction :

Cette fonction est très longue, de la ligne 367 à la ligne 861, soit 500 lignes de code ! En regardant le début et la fin, on peut voir que son utilité est d'obtenir ces variables. Comment `lines` est-il obtenu ?

On peut voir que dans une grande boucle `for`, chaque `line` est ajoutée une par une dans `lines`. Mais comment est obtenu le `lineCount` ?

À la ligne 472, un objet `framesetter` est créé, avec le paramètre `text` étant un `NSAttributedString`. Ensuite, un `CTFrameRef` est créé à partir de l'objet `frameSetter`, puis les `lines` sont obtenues à partir du `CTFrameRef`. Mais qu'est-ce exactement qu'une `line` ? Mettons un point d'arrêt pour l'examiner.

On a découvert que pour le mot `shadow`, la valeur de `lineCount = 2` ne correspond pas au nombre de lettres comme on pourrait s'y attendre.

Donc, on peut supposer que le `Shadow` blanc est en fait une seule `line`, et l'ombre est également une seule `line` ?

Dans YYText, il y a plusieurs exemples, mais un seul effet est affiché, les autres codes étant commentés. J'ai remarqué quelque chose d'étrange : pour `Shadow`, `lineCount = 2`, et pour `Multiple Shadows`, `lineCount` est également égal à 2. Cependant, `Multiple Shadows` a également une ombre intérieure, donc cela devrait être 3, non ?

En consultant la documentation Apple pour `CTLine`, il est indiqué que `CTLine` représente une ligne de texte, et un objet `CTLine` contient un ensemble de `glyph runs`. Donc, il s'agit simplement du nombre de lignes ! En regardant la capture d'écran du point d'arrêt ci-dessus, la raison pour laquelle `shadow` avait une valeur de 2 est que son texte était `shadow\n\n`. Comme vous pouvez le voir, `\n\n` a été ajouté intentionnellement pour des raisons esthétiques :

Donc, `shadow\n\n` représente deux lignes de texte. CTLine correspond à ce que nous appelons habituellement une ligne. Revenons maintenant à notre `lineCount` :

Ici, nous obtenons un tableau de `CTLines`, puis nous comptons le nombre d'éléments dans ce tableau. Si `lineCount` est supérieur à 0, nous obtenons l'origine des coordonnées pour chaque ligne. Maintenant que nous avons `lineCount`, passons à la boucle `for`.

Depuis le tableau `ctLines`, on obtient un `CTLine`, puis on crée un objet `YYTextLine` qui est ensuite ajouté au tableau `lines`. Ensuite, on effectue quelques calculs de frame pour la `line`. Le constructeur de `YYTextLine` est assez simple : il sauvegarde d'abord la position, si le texte est en mode vertical, et l'objet `CTLine` :

```
- (instancetype)initWithCTLine:(CTLineRef)ctLine position:(CGPoint)position vertical:(BOOL)vertical {
    self = [super init];
    if (self) {
        _ctLine = CFRetain(ctLine);
        _position = position;
        _vertical = vertical;
    }
    return self;
}
```

Une fois que vous avez bien compris `lines`, revenons à la fonction `YYTextDrawShadow` précédente :

Le code est maintenant plus simple. D'abord, on récupère le nombre de lignes, on les parcourt, puis on obtient le tableau `GlyphRuns`, qu'on parcourt également. Un `GlyphRun` peut être considéré comme un élément graphique ou une unité de dessin. Ensuite, on extrait le tableau `attributes`, on utilise notre `YYTextShadowAttributeName` précédemment défini pour récupérer

l'ombre (`shadow`) que nous avons attribuée au départ, et enfin on commence à dessiner l'ombre :

Une boucle `while` qui dessine continuellement des ombres portées. Appelle `CGContextSetShadowWithColor` pour définir le déplacement, le rayon et la couleur de l'ombre. Ensuite, appelle `YYTextDrawRun` pour effectuer le dessin réel. `YYTextDrawRun` est appelé à trois endroits :

Utilisé pour dessiner des ombres intérieures, des ombres de texte ainsi que le texte lui-même. Cela indique qu'il s'agit d'une méthode générique utilisée pour dessiner l'objet `Run`.

On commence par obtenir la matrice de transformation du texte, puis on utilise `runTextMatrixIsID` pour vérifier si elle reste inchangée. Si ce n'est pas une mise en page verticale ou si aucune transformation graphique n'est définie, on passe directement au dessin. On appelle `CTRunDraw` pour dessiner l'objet `run`. Ensuite, en mettant un point d'arrêt, on constate que lors du dessin de l'ombre initiale, le programme n'entre que dans le bloc `if` et ne passe pas dans le bloc `else`.

Ainsi, notre dessin d'ombres est terminé !

En résumé, `YYLabel` stocke d'abord les effets tels que les ombres dans les attributs de `attributedText`, puis redéfinit la méthode `display` de `UIView`. Dans `display`, il effectue un rendu asynchrone en utilisant le framework `CoreText` pour obtenir des objets `CTLine` et `CTRun`. Ensuite, il récupère les attributs à partir de `CTRun` et, en fonction des propriétés contenues dans ces attributs, utilise le framework `CoreGraphics` pour dessiner l'objet `CTRun` dans le contexte.

La compréhension n'est pas encore suffisante, je reviendrai lire cela plus tard. Je ne peux m'empêcher de m'exclamer que YY est vraiment trop fort ! Aujourd'hui, j'ai organisé mes idées, me permettant d'écrire tout en lisant le code, pour ne pas que cela devienne ennuyeux, et en même temps pour offrir une référence à tout le monde. Il est temps d'aller dormir.