¿Cómo funciona YYText?

El efecto de sombra anterior se logró con el siguiente código:

Se puede observar que primero se genera `YYTextShadow`, luego se asigna a `yy_textShadow` de `attributedString`, y después se asigna `attributedString` a `YYLabel`. A continuación, se añade `YYLabel` a `UIView` para mostrarlo. Al rastrear `yy_textShadow`, se descubre que principalmente se vincula `textShadow` al atributo de `NSAttributedString`, donde la clave es `YYTextShadowAttributeName` y el valor es `textShadow`. Es decir, primero se almacena el shadow y luego se utiliza. Usando Shift + Command + J, se puede saltar rápidamente a la definición:

Aquí hay un `addAttribute`, que está definido en `NSAttributedString.h`:

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

En español, el método se traduciría como:

```
- (void)agregarAtributo:(NSString *)nombre valor:(id)valor rango:(NSRange)rango;
```

La explicación indica que se pueden asignar pares clave-valor arbitrarios. La definición de `YYTextShadowAttributeName` es una cadena de texto común, lo que significa que primero se almacena la información de la sombra y luego se utiliza más adelante. Vamos a realizar una búsqueda global de `YYTextShadowAttributeName`.

Luego llegamos a la función `YYTextDrawShadow` dentro de `YYTextLayout`:

`CGContextTranslateCTM` se refiere a cambiar las coordenadas del origen en un contexto, por lo que

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

Se refiere a mover el contexto de dibujo al punto `point`. Primero, debemos aclarar dónde se llama a `YYTextDrawShadow`, y descubrimos que se llama dentro de `drawInContext`.

En `drawInContext`, se dibuja en orden el borde del bloque, luego el borde del fondo, la sombra, la línea subrayada, el texto, los accesorios, la sombra interior, la línea de tachado, el borde del texto y las líneas de depuración.

Entonces, ¿dónde se usa exactamente `drawInContext`? Puedes ver que hay un parámetro llamado `YYTextDebugOption`, lo que significa que esta función definitivamente no es una devolución de llamada del sistema, sino que es llamada internamente por YYText.

Mantén presionado Ctrl + 1 para abrir el menú de atajos de teclado, y notarás que hay cuatro lugares donde se invoca.

`drawInContext:size:debug` sigue siendo una llamada propia de YYText, ya que el tipo de `debug` es `YYTextDebugOption *`, que es propio de YY. `newAsyncTask` no parece ser una llamada del sistema, y lo mismo ocurre con `addAttachmentToView:layer:`, por lo que es muy probable que sea `drawRect:`.

Efectivamente, al mirar la ayuda rápida en el lado derecho, hay una explicación detallada, y debajo de la ayuda también se indica que está definido en `UIView`. Si observamos `YYTextContainerView`, vemos que hereda de `UIView`.

¿Entonces `YYLabel` utiliza `YYTextContainerView`? ¿Y luego hace que el sistema llame al método `drawRect:` dentro de `YYTextContainerView` para dibujarlo?

Es curioso, `YYLabel` hereda de `UIView`. Por lo tanto, en YYText debería haber dos conjuntos de cosas: uno para `YYLabel` y otro para `YYTextView`, similar a cómo están `UILabel` y `UITextView`. Luego, si volvemos a revisar el `newAsyncDisplayTask` de `YYLabel` que mencionamos antes,

Es bastante largo, en la parte central se llama a `drawInContext` dentro de `YYTextLayout`. ¿Dónde se llama a `newAsyncDisplayTask`?

En la segunda línea se llamó. Por lo tanto, se puede entender simplemente que `YYLabel` utiliza asincronía para dibujar el texto. Y `_displayAsync` fue llamado por el `display` anterior. Mirando la documentación de `display`, dice que el sistema lo llamará en el momento adecuado para actualizar el contenido del layer, y no debes llamarlo directamente. También podemos ponerle un punto de interrupción.

La explicación es que `display` se llama dentro de una transacción de `CALayer`. ¿Por qué se usa una transacción? Probablemente para realizar actualizaciones en lote, lo que sería más eficiente, ¿no? No parece ser un requisito de reversión como en una base de datos.

La documentación del sistema de `display` también menciona que, si deseas que tu capa se dibuje de manera diferente, puedes sobrescribir este método para implementar tu propio dibujo.

Entonces, tenemos una idea básica. `YYLabel` sobrescribe el método `display` de `UIView` para dibujar de manera asíncrona varios efectos como sombras, etc. Los efectos de sombra se almacenan primero en los atributos del `attributedText` de `YYLabel`, y luego se recuperan durante la ejecución del método `display` para dibujarlos. Durante el dibujo, se utiliza el framework CoreGraphics del sistema.

Después de aclarar algunas ideas, te darás cuenta de que lo que realmente es poderoso es, por un lado, la capacidad de organizar tantos efectos, llamadas asíncronas, etc., y por otro lado,

el manejo experto del framework CoreGraphics subyacente. Así que, después de comprender un poco la organización del código anterior, profundizaremos en el framework CoreGraphics. Veamos cómo se realiza el dibujo.

Volvamos a `YYTextDrawShadow`.

Aquí, `CGContextSaveGState` y `CGContextRestoreGState` encierran un bloque de código de dibujo. `CGContextSaveGState` significa copiar el estado actual de dibujo y guardarlo en la pila de dibujo. Cada contexto de dibujo mantiene una pila de dibujo. No estoy seguro de cómo se maneja exactamente la pila internamente. Por ahora, entiéndase que antes de dibujar en el contexto, se debe llamar a `CGContextSaveGState`, y después de dibujar en el contexto, se debe llamar a `CGContextRestoreGState`. Luego, el dibujo en el medio aparecerá efectivamente en el contexto. `CGContextTranslateCTM` mueve el contexto a la posición correspondiente. Primero se mueve a `point.x` y `point.y`, que son las coordenadas correspondientes para el dibujo. En cuanto al movimiento posterior a 0 y `size.height`, no estoy seguro, lo revisaré más adelante. Luego se obtienen las `lines` y se ejecuta un bucle `for`.

¿Qué es `lines`? Se encuentra en `YYTextLayout` dentro del método `(YYTextLayout *)layoutWithContainer:(YYText *)container text:(NSAttributedString *)text range:(NSRange)range`, donde se asigna un valor.

Luego, navega hasta la definición de esta función:

Esta función es extremadamente larga, ¡va desde la línea 367 hasta la 861, con 500 líneas de código! Después de revisar el principio y el final, se puede ver que su propósito es obtener estas variables. ¿Cómo se obtiene `lines`?

Se puede observar que dentro de un gran bucle `for`, se agrega una por una cada `line` a `lines`. Entonces, ¿cómo se obtiene el `lineCount`?

En la línea 472, se crea un objeto `framesetter`, donde el parámetro `text` es un `NSAttributedString`. Luego, se crea un `CTFrameRef` dentro del objeto `frameSetter`, y a partir de este `CTFrameRef`, se obtienen las `lines`. ¿Qué es exactamente `line`? Vamos a ponerle un punto de interrupción para averiguarlo.

Descubrí que la palabra `shadow` tiene `lineCount = 2`, lo cual no es el número de letras que imaginábamos.

Así que supongo que el `Shadow` blanco es una sola `line`, y la sombra también es una `line`?

En YYText hay varios ejemplos, solo se muestra uno de los efectos y los demás códigos están comentados. Noté algo extraño: el `lineCount` de Shadow es 2, y el `lineCount` de Multiple Shadows también es 2, pero Multiple Shadows tiene una sombra interna, ¿no debería ser 3?

Al buscar la documentación de Apple sobre CTLine, se dice que CTLine representa una línea de texto, y un objeto CTLine contiene un conjunto de `glyph runs`. ¡Así que simplemente se trata del número de líneas! Mirando la captura de pantalla del punto de interrupción anterior, la razón por la que `shadow` era 2 es porque su texto era `shadow\n\n`. Como se puede ver, `\n\n` se agregó intencionalmente para mejorar la presentación visual:

Por lo tanto, `shadow\n\n` son dos líneas de texto. CTLine es lo que comúnmente llamamos una línea. Luego, volvemos a nuestro `lineCount`:

Aquí obtenemos el arreglo `CTLines`, contamos el número de elementos dentro de él, y si `lineCount` es mayor que 0, obtenemos el origen de las coordenadas de cada línea. Bien, ahora que tenemos `lineCount`, continuamos con el bucle `for`.

Obtén `CTLine` del array `ctLines`, luego obtén el objeto `YYTextLine` y agrégalo al array `lines`. Después, realiza algunos cálculos de `frame` para la línea. El constructor de `YYTextLine` es bastante simple, primero guarda la posición, si es un diseño vertical y el objeto `CTLine`:

Una vez que hayas entendido `lines`, volvamos a la función `YYTextDrawShadow` anterior:

Ahora el código es más sencillo. Primero obtenemos el número de líneas, lo recorremos, luego obtenemos el array `GlyphRuns` y lo recorremos también. `GlyphRun` puede entenderse como un primitivo gráfico o una unidad de dibujo. Luego, obtenemos el array `attributes` y, utilizando nuestro `YYTextShadowAttributeName` anterior, obtenemos el `shadow` que asignamos al principio. Finalmente, comenzamos a dibujar la sombra:

Un bucle `while` que dibuja continuamente sombras secundarias. Se llama a `CGContextSetShadowWithColor` para configurar el desplazamiento, el radio y el color de la sombra. Luego se llama a `YYTextDrawRun` para realizar el dibujo real. `YYTextDrawRun` es llamado desde tres lugares:

Se utiliza para dibujar sombras internas, sombras de texto y el texto en sí. Esto indica que es un método genérico utilizado para dibujar el objeto `Run`.

Primero, se obtiene la matriz de transformación del texto y se utiliza `runTextMatrixIsID` para verificar si permanece sin cambios. Si no es una disposición vertical o no se ha configurado una transformación de gráficos primitivos, se procede directamente a dibujar. Se llama a `CTRunDraw` para dibujar el objeto `run`. Luego, al establecer un punto de interrupción, se descubre que al dibujar la sombra inicial solo se entra en el bloque `if`, pero no en el bloque `else`.

¡Así que eso es todo para nuestro dibujo de sombras!

En resumen, `YYLabel` primero guarda efectos como sombras en los atributos del `attributedText`, sobrescribe el método `display` de `UIView`, y realiza el dibujo asíncrono dentro de `display`. Utiliza el framework `CoreText` para obtener objetos `CTLine` y `CTRun`, y luego extrae los atributos desde

`CTRun`. Finalmente, basándose en los atributos obtenidos, utiliza el framework `CoreGraphics` para dibujar el objeto `CTRun` en el Contexto.

Aún no lo entiendo del todo, volveré a leerlo más adelante. No puedo evitar admirar lo increíblemente poderoso que es YY. Hoy he organizado mis ideas, escribiendo y leyendo código al mismo tiempo para no aburrirme, y también para que otros puedan consultarlo. Es hora de ir a dormir.