Performance de React

¿Qué es el DOM virtual?

El DOM virtual (VDOM) es un concepto de programación donde una representación ideal o “virtual” de la IU se mantiene en memoria y en sincronía con el DOM “real”, mediante una biblioteca como ReactDOM. Este proceso se conoce como **reconciliación.**

Este enfoque hace posible la API declarativa de React: le dices a React en qué estado quieres que esté la IU, y se hará cargo de llevar el DOM a ese estado. Esto abstrae la manipulación de atributos, manejo de eventos y actualización manual del DOM que de otra manera tendrías que usar para construir tu aplicación.

Ya que “DOM virtual” es más un patrón que una tecnología específica, las personas a veces le dan significados diferentes. En el mundo de React, el término “DOM virtual” es normalmente asociado con elementos de React ya que son objetos representando la interfaz de usuario. Sin embargo, React también usa objetos internos llamados “fibers” para mantener información adicional acerca del árbol de componentes. Éstos pueden ser también considerados como parte de la implementación de “DOM virtual” de React.

¿Qué es React Fiber?

La explicación oficial de una frase es "React Fiber es una reimplementación del algoritmo central ". El ajuste de la arquitectura de fibra se anunció oficialmente muy temprano, pero tomó dos años lanzarlo oficialmente en la versión V16. La explicación oficial del concepto es demasiado general, de hecho, en resumen, React Fiber es una nueva tarea de reconciliación (reconciliación), que se explicará en detalle más adelante en este capítulo.

¿Por qué se llama "fibra"?

Todo el mundo debería conocer los conceptos de proceso y subproceso. El proceso es la unidad más pequeña de asignación de recursos del sistema operativo, y el subproceso es la unidad más pequeña de programación del sistema operativo. Existe otro concepto en informática llamado Fibra, que significa "" Fibra "significa un subproceso que es más delgado que Thread, que es un mecanismo de procesamiento concurrente que es más sofisticado que Thread. La fibra y la fibra React mencionadas anteriormente no son el mismo concepto, pero el equipo de React llamó a esta función Fiber, lo que significa un mecanismo de procesamiento más cercano, que es más detallado que Thread.

Reconciliation

El nombre oficial del algoritmo central de React es Reconciliación, sinodimo de "coordinación" La implementación del algoritmo React diff Está relacionado con eso.

Repasemos brevemente React Diff: React fue pionero en el concepto de "DOM virtual". La razón principal para que el "DOM virtual" se vuelva popular es que el concepto es una innovación revolucionaria en la optimización del rendimiento de front-end;

Los estudiantes de front-end que tienen un poco de comprensión del principio de cargar páginas en los navegadores saben que los problemas de rendimiento de las páginas web ocurren principalmente en operaciones frecuentes de nodos DOM;

Y React garantiza el rendimiento del front-end a través del algoritmo "DOM virtual" + React Diff;

Algoritmo Diff tradicional

Al comparar los nodos a su vez a través de ciclos recursivos, la complejidad del algoritmo llega a O (n ^ 3), donde n es el número de nodos en el árbol ¿Qué tan terrible es esto? ——Si desea mostrar 1000 nodos, debe realizar cientos de millones de comparaciones. . Incluso si la CPU puede ejecutar rápidamente 3 mil millones de comandos,Es difícil en un segundoCalcule la diferencia.

Algoritmo de React Diff

Convierta el árbol DOM virtual en el árbol DOM realProceso de operación mínimo Se llama **Reconciliación.**

Resumen de Virtual DOM:

- Es un patrón de comportamiento y React lo implementa con una tecnología llamada “Fiber”.
- En sí resulta ser todo lo que React sabe de tu aplicación y cada nodo o fibra.
- Esto es básicamente lo que React hace con el Virtual DOM: una representación virtual de la IU que se mantiene en memoria y en sincronía “reconciliado” con el DOM “real”.

¿Qué es npm?

npm, Node Package Manager, es un gestor de paquetes para JavaScript. Es el gestor de paquetes más popular ahora mismo, pertenece a GitHub (Microsoft), y es el predeterminado para el entorno de ejecución de Node.js.

Gracias a npm, como desarrollador web, o agencia de desarrollo web, puedes beneficiarte de npm de muchas maneras. Sobre todo, puedes compartir código entre tus equipos o con otros programadores.

El objetivo de npm, Inc. es llevar el "desarrollo en JavaScript a la elegancia, productividad, y seguridad" y que no tengas que reinventar la rueda si quieres resolver problemas que la comunidad ya ha dado solución y otros proyectos con las mismas necesidades.

¿Cómo está organizado npm?

npm se organiza principalmente en tres partes, o componentes:

- El **portal**, un sitio web que te permite encontrar paquetes de terceros, gestionar tus paquetes o configurar perfiles
- La **interfaz de línea** de comandos de npm que te permite interactuar, instalar y publicar paquetes
- El **Registro**, es una colección pública de paquetes de código open-source para la mayoría de las necesidades de la comunidad JavaScript.

El fichero package.json

Como lograste entender, npm te permite instalar un nuevo paquete desde el Registro. Además, podes descubrir y publicar nuevos paquetes de Node.js.

Si ya has instalado npm en tu máquina, debes tener acceso a los comandos típicos de npm, por ejemplo "npm -v", para conocer la versión de tu gestor. Además, para que tu proyecto funcione correctamente con npm necesitarás tener un fichero llamado package.json en el directorio raíz.

Este fichero contiene información importante que npm usa para identificar el proyecto y gestionar las dependencias. Cuando instalas un paquete y este tiene dependencias, npm instalará también las dependencias de las dependencias.

Algunos comandos útiles de npm

npm install -g npm

npm install -g npm (***sirve para descargar la última versión de npm)***

npm install <nombre_del_paquete>

Para instalar un nuevo paquete, puedes utilizar npm install seguido del nombre del paquete. Puedes encontrar el nombre del paquete en el sitio web de npm. También puedes utilizar “npm i <paquete>” para instalar, “npm un <paquete>” para desinstalar, “npm up <paquete>” para actualizar.

npm init

Puedes utilizar npm init para configurar un paquete de npm nuevo o uno existente. npm init crea un fichero package.json.

¿Dónde se guardan los paquetes y módulos de npm?

Cuando instalas con éxito por primera vez un paquete, podrás ver un nuevo directorio llamado “/node_modules” creado en la raíz del proyecto. Los nuevos módulos que instales se guardarán en este directorio

Además, si abres el fichero package.json, verás que la sección de dependencias se ha actualizado, incluyendo el nuevo paquete.

Conclusión

Para finalizar, npm es una herramienta que cualquier desarrollador de aplicaciones web modernas utiliza de alguna forma. Si tu proyecto, por ejemplo, requiere funcionalidades para gestionar fechas y horas, npm te permite integrar bibliotecas como moment de forma fácil, y gestionar estas fechas.

Creo que vale la pena referir que existen otras herramientas y gestores de paquetes con el mismo objetivo. De todas formas, utilices npm u otro, te recomiendo que lo hagas de manera responsable.

NPM - Conceptos básicos sobre dependencias

A alto nivel, NPM no es muy diferente de otros gestores de paquetes de otros lenguajes de programación. Los paquetes dependen de otros paquetes, y estos expresan estas dependencias con rangos de versiones. NPM utiliza el esquema semver (o Semantic Versioning) para expresar esos rangos. Lo importante a destacar aquí es que estos paquetes pueden depender de un rango de versiones de dependencias, en lugar de una versión específica de una dependencia o paquete en si.

Esto es bastante importante para cualquier ecosistema, ya que bloquear una libreria a un rango específico de dependencias puede causar problemas relativamente importantes. No obstante, en NPM esto es un poco menos de problema si lo comparamos a otros sistemas de paquetes como composer en PHP.

En definitiva, normalmente es bastante seguro para el creador de una librería el limitar las dependencias de una versión específica sin afectar al comportamiento del resto de librerías o aplicaciones conectadas. La parte así un poco compleja es la de determinar cuando hacer esto es una práctica segura y cuando no, y aquí es donde noto (y la mayoría de desarrolladores con las que colaboro está de acuerdo) que la mayor parte de la gente no acaba de pillarlo (y yo al principio me incluyo).

Duplicación de dependencias en el arbol de dependencias

Si no lo habías descubierto ya, al contrario que otros gestores de paquetes, NPM instala un árbol de dependencias. Esto quiere decir, que cada paquete instalado obtiene su propio set de dependencias, en lugar de forzar a todos los paquetes a compartir el mismo set canónico de dependencias. Obviamente, prácticamente cualquier gestor de paquetes en existencia tiene que modelar un árbol de dependencias en algún punto de su desarrollo para su correcto funcionamiento, ya que es así como los programadores expresamos las dependencias entre diferentes paquetes.

Por ejemplo, consideramos dos paquetes, foo y bar. Cada uno de ellos tiene su propio set de dependencias, que se puede expresar como un árbol:

foo

-- hola ^0.1.2

-- mundo ^1.0.

bar

-- hola ^0.2.2

-- chau ^ 3.5.1

Imagina que una aplicación que depende tanto de foo como de bar. Claramente, las dependencias de mundo y chau carecen de ningún tipo de relación, por lo que la manera en la que NPM las almacena y organiza no es algo muy interesante para lo que queremos tratar en este artículo. No obstante, si consideramos el caso de hola, tanto foo como bar requieren versiones que entran en conflicto.

La mayoría de paquetes (incluyendo RubyGems, pip, Cabal...) sencillamente se detendrían aquí, dando algún tipo de error de versión de conflictos y ya te apañarás que seguro que te divierte arreglar esto. Esto ocurre porque en la mayoría de gestores de paquetes, tan solo puede existir una única versión de cualquier paquete. En ese sentido, la responsabilidad de cualquier gestor de paquetes es la de averiguar que set de paquetes y que versiones de cada uno de estos paquetes pueden utilizarse para satisfacer todas las necesidades de todos los paquetes simultáneamente.

Por lo contrario, NPM tiene un trabajo algo más sencillo. Para NPM no hay problema en instalar diferentes versiones del mismo paquete porque cada paquete tiene su propio árbol de dependencias. En el ejemplo que hemos puesto arriba, el listado de directorios de los paquetes tendría este aspecto.

node_modules/

├── foo/

| └── node_modules/

| ├── hola/

| └── mundo/

└── bar/

└── node_modules/

├── hola/

└── chau/

Como podemos observar, la estructura del directorio se parece mucho al árbol de dependencias original. El diagrama que mostramos arriba no es más que una simplificación. En la práctica, esto se traduce a que cada dependencia transitiva tendrá su propio directorio node_modules dentro de sí misma y as i sucesivamente.

Como pueden imaginar, la estructura del directorio se complicar a medida que las dependencias entre paquetes aumenten (Además, desde la versión 3 de npm que se realizan ciertas optimizaciones para tratar de compartir dependencias cuando se puedan, pero por el momento vamos a ignorar esto porque no es necesario saberlo ni estudiarlo para entender el modelo de NPM).

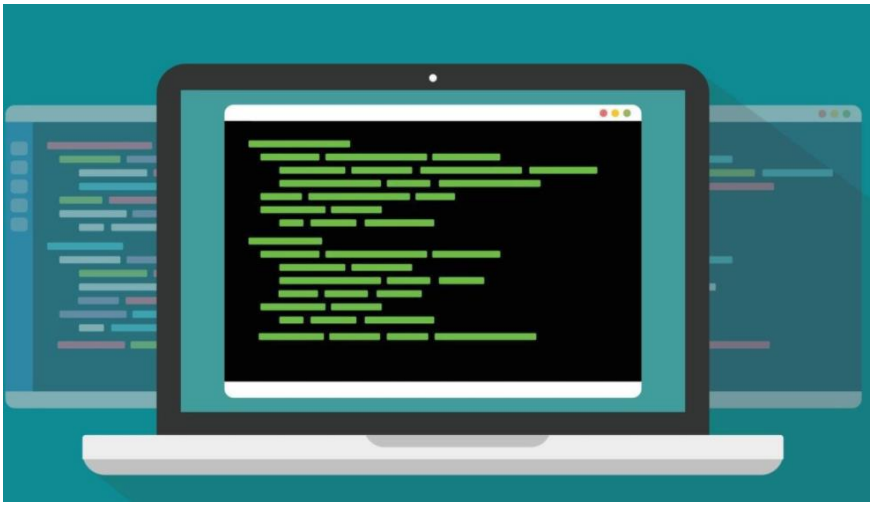
Esta forma de estructurar las dependencias la verdad que es bastante simple ¿no les parece? El efecto inmediato que podemos ver es que cada paquete tiene su propio "sandbox", que funciona muy bien para librerias , sobretodo funcionales.

A simple vista, podríamos decir que este sistema obviamente es mejor que cualquier alternativa de gestión de paquetes plana, siempre y cuando la ejecución de dichos programas soporte la carga de los módulos necesarios en este esquema (por ejemplo, que dos modulos diferentes con una misma dependencia pero en versiones diferentes no tenga que utilizar una variable global.

El mayor inconveniente es el aumento muy significativo del tamaño del código, ya que replicamos una y otra vez diferentes versiones de diferentes paquetes dentro de nuestro árbol de dependencias una y otra y otra vez. Un código que ocupa más y tiene más tamaño a menudo lleva aparejado una pérdida en el rendimiento de tu aplicación. Los programas más largos sencillamente no caben en la caché de una CPU tan fácilmente, y solamente por tener que paginar en cache un programa puede hacer que las cosas vayan bastante más lentas.

Hay que entender esto como una solución de compromiso. Estás sacrificando algo de rendimiento, no mantenimiento de código ni el hecho de que tu código sea mejor o peor.

¿Qué son las CLI?



Las CLI son comandos que ejecutamos en nuestra terminal para hacer algo. Si quieres una definición:

Una interfaz de línea de comandos procesa los comandos de un programa de computadora en forma de líneas de texto. El programa que maneja la interfaz se llama intérprete de línea de comandos o procesador de línea de comandos. Los sistemas operativos implementan una interfaz de línea de comandos en un shell para el acceso interactivo a las funciones o servicios del sistema operativo

¿Por qué son necesarias las CLI?

En el mundo moderno de las GUI (interfaces gráficas de usuario), podría preguntarse ¿por qué deberíamos conocer las CLI? ¿No se usaron en los 80? Estoy de acuerdo contigo en un 💯 por ciento. Están desactualizadas, pero muchas aplicaciones antiguas todavía usan CLI. El terminal / símbolo del sistema generalmente tiene más permisos y acceso en comparación con las aplicaciones GUI de forma predeterminada. Es una mala experiencia de usuario permitir 100 permisos para ejecutar una aplicación. Además, la gente bromea sobre ello como el "mejor amigo" de un desarrollador (que no sea Google).

Git Bash

En esencia, Git es un conjunto de programas de utilidades de líneas de comandos que están diseñados para ejecutarse en un entorno de líneas de comandos de estilo Unix. Los sistemas operativos modernos como Linux y macOS incluyen terminales de líneas de comandos Unix integrados. Esto convierte a Linux y a macOS en sistemas operativos complementarios cuando se trabaja con Git. En cambio, Microsoft Windows utiliza el símbolo del sistema de Windows, un entorno de terminal que no es Unix.

En entornos de Windows, Git normalmente se incluye en un paquete como parte de aplicaciones de interfaz gráfica de usuario de nivel superior. Las interfaces gráficas de usuario para Git podrían intentar abstraer y ocultar los lenguajes primitivos del sistema de control de versiones subyacente. Esto puede ser una ayuda excepcional para que los principiantes en Git contribuyan rápidamente a un proyecto. Una vez que los requisitos de colaboración de un proyecto aumentan con otros miembros del equipo, es fundamental ser consciente de cómo funcionan los métodos de Git de verdad sin procesar. En ese momento, puede ser beneficioso

disponer de una versión de interfaz gráfica de usuario para las herramientas de líneas de comandos. Se ofrece Git Bash para proporcionar una experiencia de Git en el terminal.

¿Qué es Git Bash?

Git Bash es una aplicación para entornos de Microsoft Windows que ofrece una capa de emulación para una experiencia de líneas de comandos de Git. Bash es el acrónimo en inglés de Bourne Again Shell. Una shell es una aplicación de terminal que se utiliza como interfaz con un sistema operativo mediante comandos escritos. Bash es una shell predeterminada popular en Linux y macOS. Git Bash es un paquete que instala Bash, algunas utilidades comunes de bash y Git en un sistema operativo Windows.

Linux NO es difícil

A veces parece que el uso de terminal es muy dificil, pero para nada lo es, de hecho, es mucho más difícil programar en Windows que programar en Linux (y es por eso que Windows tiene a WSL) esto es porque con Linux tenemos todo a la mano, es un sistema operativo más dev-friendly, Windows está más pensado para el usuario final, es por eso que nos toca virtualizar todo ahí, y que de repente algo no funciona.

Otra cosa cursiosa es que de hecho sí puedes invocar demonios en la terminal “literalmente”. En el mundo de la terminal, hay algunos procesos especiales a los que se les conoce como “demonios”, básicamente son procesos que se están ejecutando en el background o en la misma terminal.

¿Alguna vez has usado Nodemon mientras trabajabas con Node.js? Bueno, Nodemon es un demonio, su propio nombre lo dice “No… demon”, y es básicamente un demonio porque cuando lo ejecutas se queda aparando la terminal, es decir, se queda corriendo un proceso.

De hecho, como dato curioso, la terminal y todos los comandos que pones ahí son básicamente un lenguaje de programación llamado “Bash”, sí, puedes programar en Bash usando la terminal. Saber usar la terminal es una de las principales habilidades que debe tener un programador para ser un profesional.

Comandos Básicos:

Comando “man”: Desplega una descripcion del comando indicado (uso, parametros y argumentos).

Sintaxis: man [comando]

Ejemplos: *man cd *muestra la ayuda referente al comando “cd”*

Comando “ls”: Crea una lista las capetas y archivos que hay en el directorio seleccionado

Sintaxis: ls [ruta] -[modificadores] [parametros]

Ejemplos: *ls -lh *ver objetos en el directorio actual estructurados de manera ordenada*

ls usr/bin | wc -l *cuenta cantidad de objetos en el directorio “usr/bin”

Modificadores:

-l lista las capetas y archivos con su información básica

-h ver y ordenar la información de forma que sea fácil de entender (para humanos)

-a ver archivos ocultos

ver más con el comando –help

Comando "pwd": Muestra el directorio en el que se ejecuta el comando, util en los casos en los que el "Prompt" no indica la ruta.

Sintaxis: pwd

Modificadores: Este comando no tiene modificadores

Comando "cd": Navega por los directorios del sistema.

Sintaxis: cd [ruta]

Ejemplos:

*cd .. *retrocede un directorio.*

*cd ~*se mueve al directorio "home".*

Modificadores: ver mas con el comando --help

Comando "mkdir": Crear una carpeta si no existe.

Sintaxis: mkdir -[modificadores] [nombre_carpeta]

Ejemplos:

*mkdir uauario *crea carpeta llamada "usuario"*

Modificadores: ver mas con el comando --help

Comando "touch": Crea un archivo si no existe, de lo contrario cambia la fecha de modificación (el argumento "{1, 2, 3}.txt" crea varios archivos de texto).

Sintaxis: touch -[modificadores] [nombre_archivo]

Ejemplos:

*touch uauario.txt *crea un archivo de texto llamado "usuario"*

Modificadores: ver mas con el comando --help

Comando "mv": Mueve y/o renombra el archivo indicado alojandolo en una ruta destino.

Sintaxis: mv [ruta/archivo_origen] [ruta/archivo_destino]

Ejemplos:

*mv archivo.txt C:/ *mueve el archivo archivo.txt desde la ruta actual a la ubicacion "C:/"*

*mv archivo.txt nuevo.txt *renombra el archivo sin moverlo.*

Modificadores: ver más con el comando --help

Comando "cp": Crea una copia del archivo indicado alojandolo en una ruta destino.

Sintaxis: cp [ruta/archivo] [ruta/archivo_destino]

Ejemplos:

*cp archivo.txt C:/ *crea una copia del archivo archivo.txt que se encuentra en la ruta actual a la ubicación "C:/"*

Modificadores: ver mas con el comando --help

Comando "rm": Elimina el archivo indicado

Sintaxis:

rm -[modificadores] [nombre_archivo]

rm -rf [directorio]

Ejemplos:

*mv c:/archivo.txt *elimina el archivo "archivo.txt" ubicado en "C:/"*

*rm -rf ejercicio *elimina un directorio/carpeta "ejercicio" recursivamente.*

Modificadores:

-r: remover directorio y contenido de manera recursiva.

-f: ignora archivos no existentes y argumentos, no pregunta.

ver mas con el comando --help

Comando "open": Abrir el archivo indicado, en windows se utiliza "start"

Sintaxis: open -[modificadores] [ruta_archivo]

Ejemplos:

open archivo.txt *abre el archivo archivo.txt utilizando el programa por defecto.

Modificadores: ver más con el comando --help

Comando "cat": Imprimir todo el contenido de un archivo en pantalla.

Sintaxis: cat -[modificadores] [ruta_archivo]

Ejemplos:

cat -v C:/archivo.txt *imprime el contenido del archivo "archivo.txt" en la ubicacion "c:/" mostrando los caracteres no imprimibles.

Modificadores: ver más con el comando --help

Instalación de Node JS

Para instalar React JS debemos tener instalado nodejs, dejamos el link para la instalación (recuerda siempre descargar la versión LTS):

https://nodejs.org/es/

Pasos para instalar React JS

Levantamos la ventana de Comando Git Bash y ejecutamos las siguientes instrucciones o comandos.

node --version.- Verificamos la instalación

npm --version.- Verificamos que se haya instalado correctamente el gestor de dependencias de Javascript

Importante antes de crear un proyecto, debemos posicionarnos en la carpeta en la cual vamos a trabajar, recuerden organizar sus carpetas y proyectos.

Una vez posicionados en el lugar indicado ejecutar las siguientes líneas de comando:

npx create-react-app miaplicacion

Con este comando iniciamos la instalación de un proyecto con React JS.

Este proceso suele tardar algunos minutos, una vez terminada la instalación, eberemos ingresar a nuestra carpeta con los siguientes comandos

cd miaplicacion

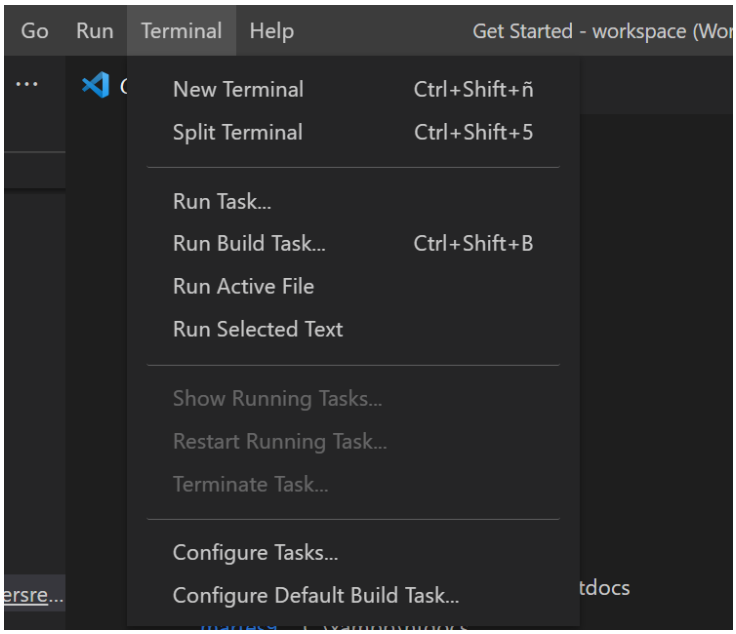
Ingresamos al directorio de la aplicación

Nos aseguramos estar dentro de la carpeta raíz del proyecto y ahí ejecutar la siguiente línea de comandos

npm start

Finalmente, iniciamos la compilación del proyecto para visualizarlo en un navegador web

Les recordamos que también podemos ejecutar la terminal de línea de comandos desde Visual Estudio Code



Recomendamos también utilizar la terminal bash en visual studio en lugar del predeterminado power shell