React Fiber

被重寫的 React 核心

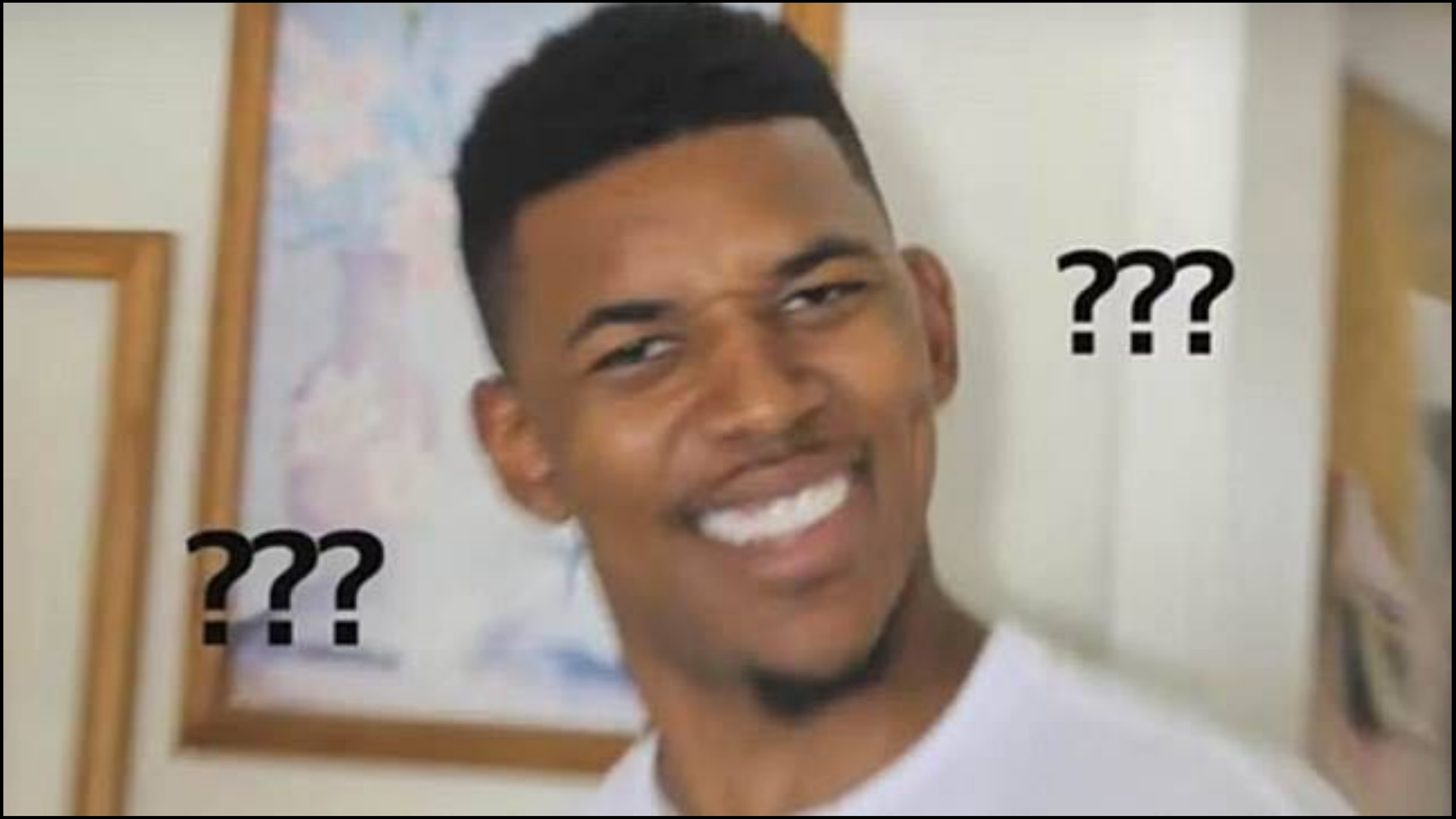
C. T. Lin

Engineering Lead@Yoctol

chentsulin@github

快速前情提要

Component, Element, Instance 的差別？



Component

有 Class Component 跟 Functional Component

```
// Class Component
class Block extends React.Component {
  render() {
    return <div className="block" />;
  }
}


// Functional Component
const Button = () =>
  <div className="block" />;
```

Instance

- Class 的實體物件
- Component 裡面的 this
- 方便管理 LifeCycle Hook
- Functional Component 沒有 Instance

Element

通常由 React.createElement 或 JSX 得到

```
{
  type: Button,
  props: {
    color: 'blue',
    children: 'OK!'
  }
}

// JSX
<Button color="blue">OK!</Button>
```

React Core = Reconciler

React = Reconciler + Renderer

什麼是 Reconciler？

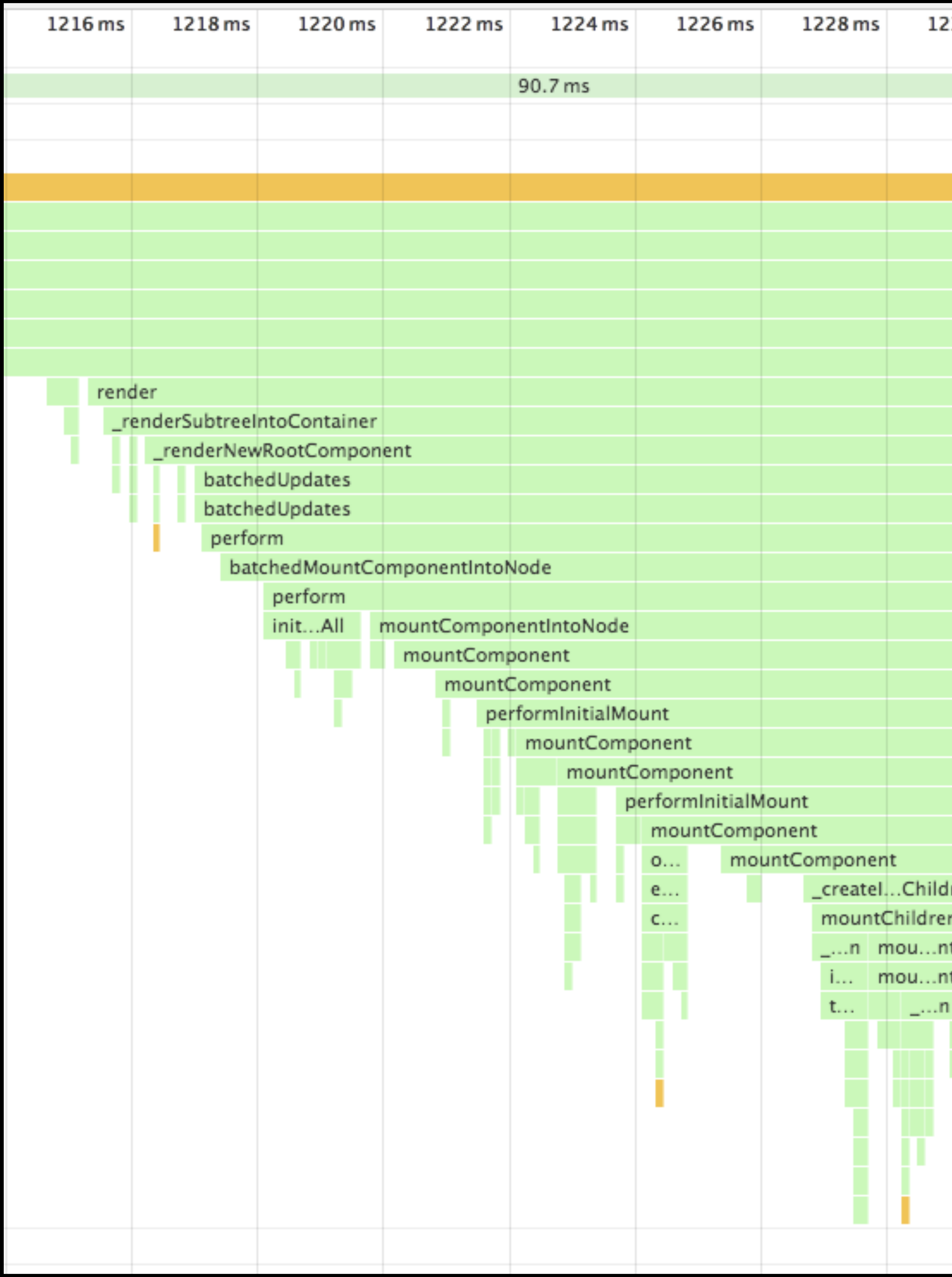
Reconciliation

- 為了提供 Declarative API 所產生的機制
- render 或 setState 後觸發
- createElement 做 Element 差異比對
- 通知 Renderer 並觸發 Instance Lifecycle Methods

Stack Reconciler

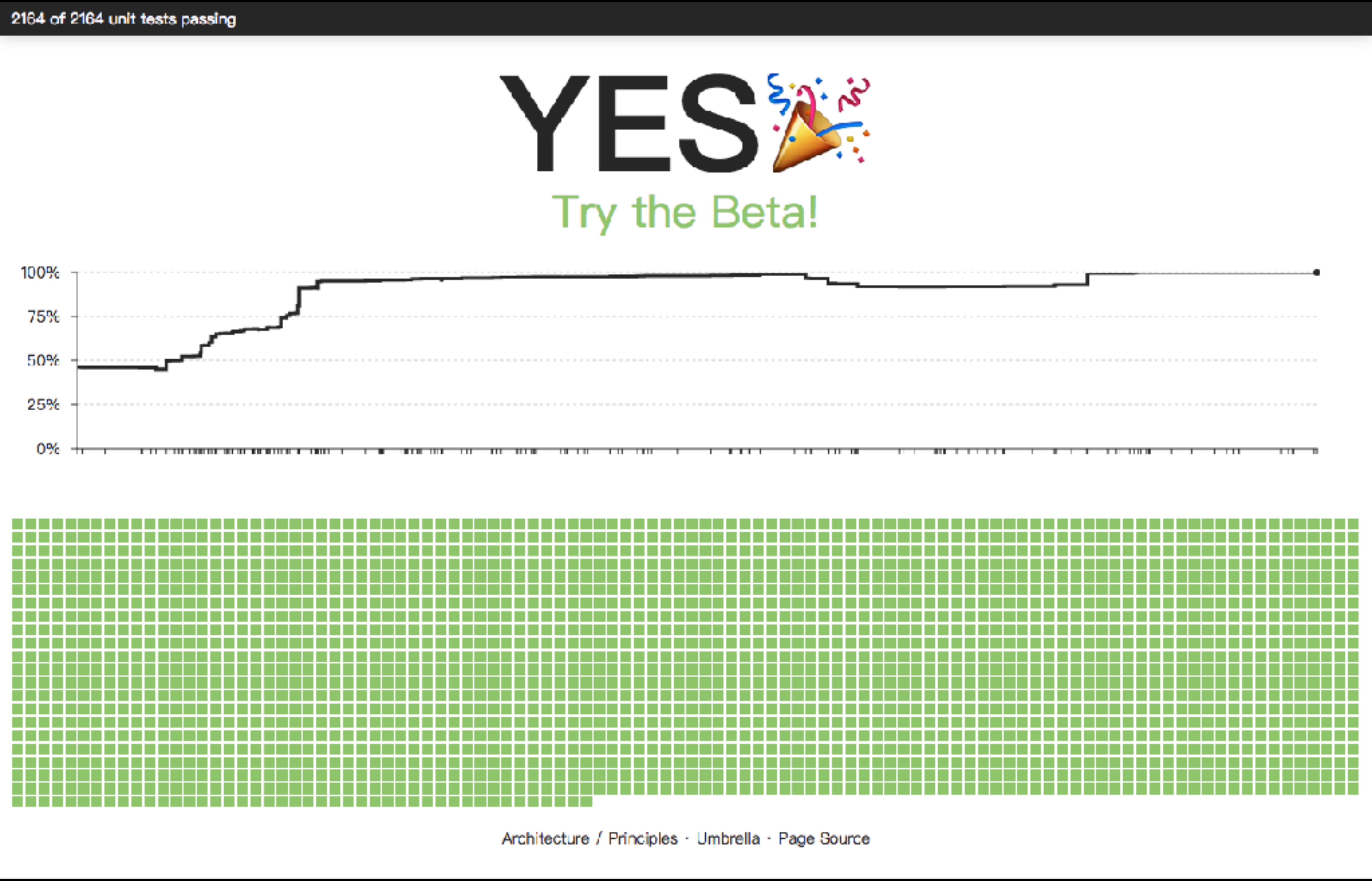
- React v15 以及之前所使用的 Reconciler
- 會一次 Recursive 的把整個 Tree 做完

所以...
Call Stack 會蠻深的



前情提要結束

接下來就進入主題
Fiber



2164 of 2164 unit tests passing

YES 🎉

Try the Beta!

Architecture / Principles · Umbrella · Page Source

http://isfiberreadyyet.com/

Fiber

- 於 2014 開始於 Facebook 內部討論
- 第一隻 PR 是 2016 / 5
- React v16 - beta (2017 / 7) 已經是採用 Fiber Reconciler
- 完整的使用 Flow Type
- 更小的 Bundle Size

為什麼需要 Fiber？

https://koba04.github.io/react-fiber-resources/examples/

Incremental Rendering

可以把工作拆成比較小的單位
並拆到多個 Frame 去執行

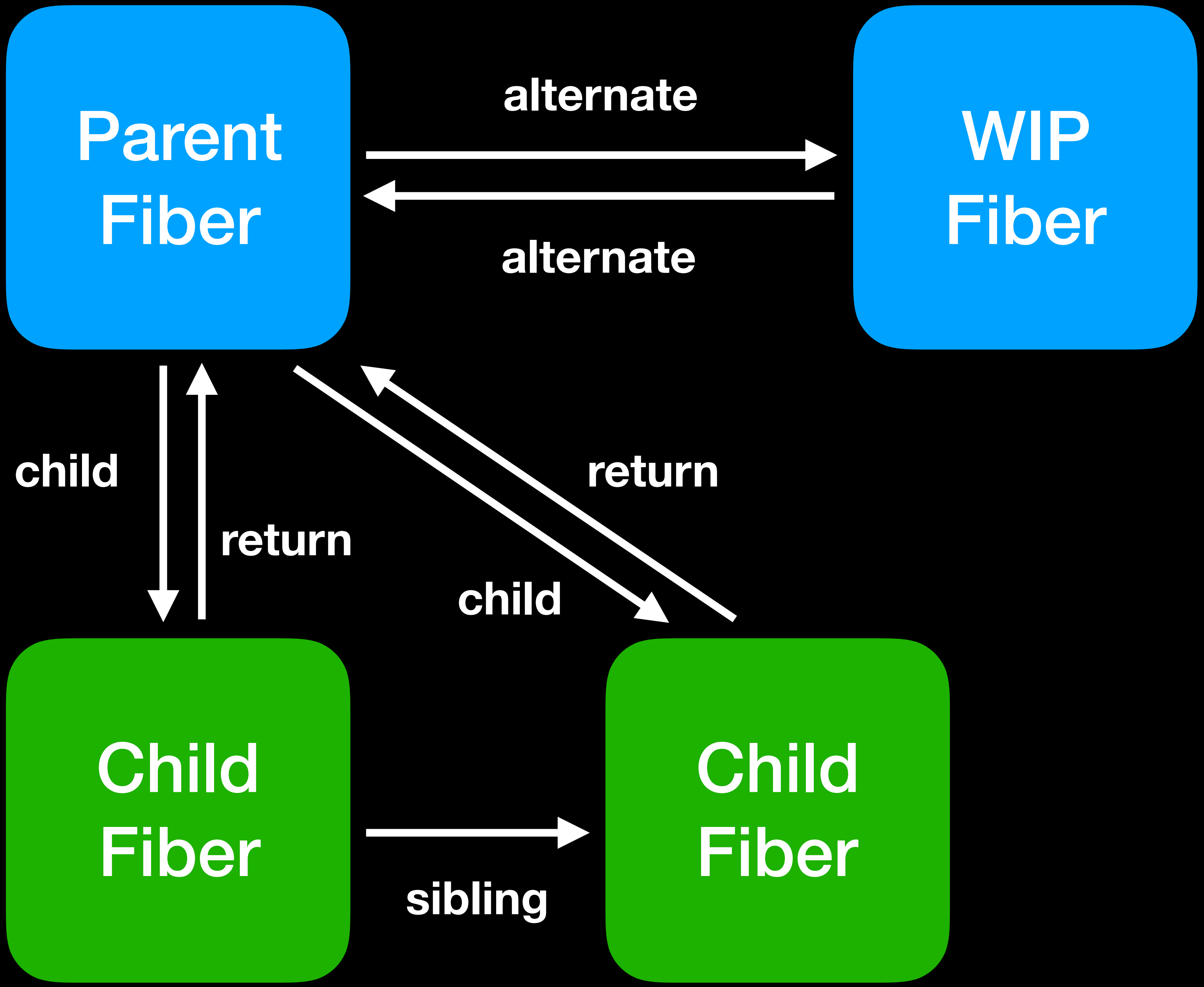
其他重要 Feature

- 可以暫停、丟棄、重用之前的工作
- 把各種更新指定不同的優先權
- Concurrent 執行
- 未來有可能加速 Initial Rendering

Fiber

是工作的最小單位

```
{
  type,
  key,
  child,
  sibling,
  return,
  alternate,
  pendingProps,
  memoizedProps,
  pendingWorkPriority,
  ..
}
```


Parent Fiber
alternate
alternate
WIP Fiber
child
return
return
child
Child Fiber
sibling
Child Fiber

Cooperative Scheduling

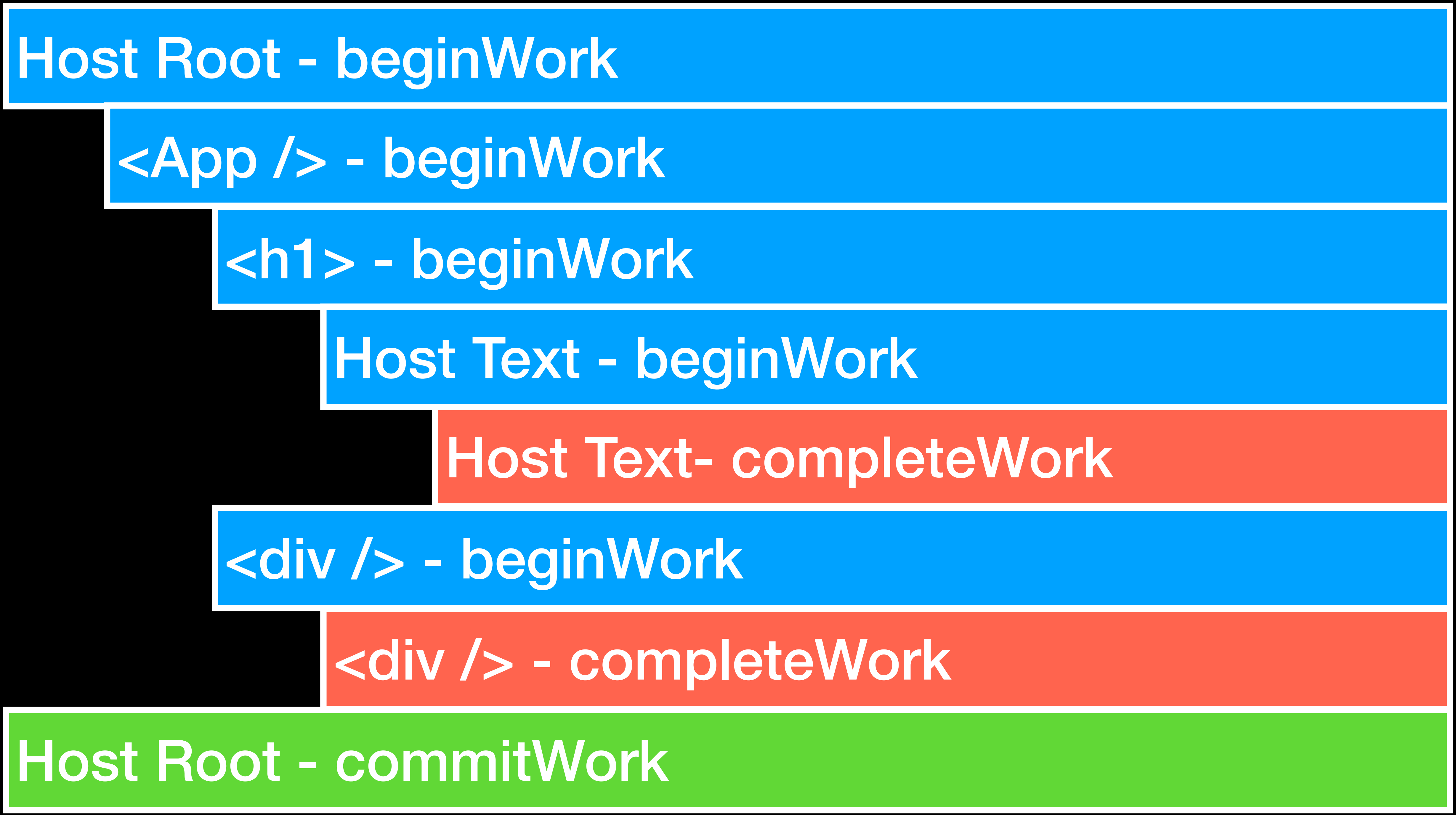
等待 Fiber 完成或讓出

Phases

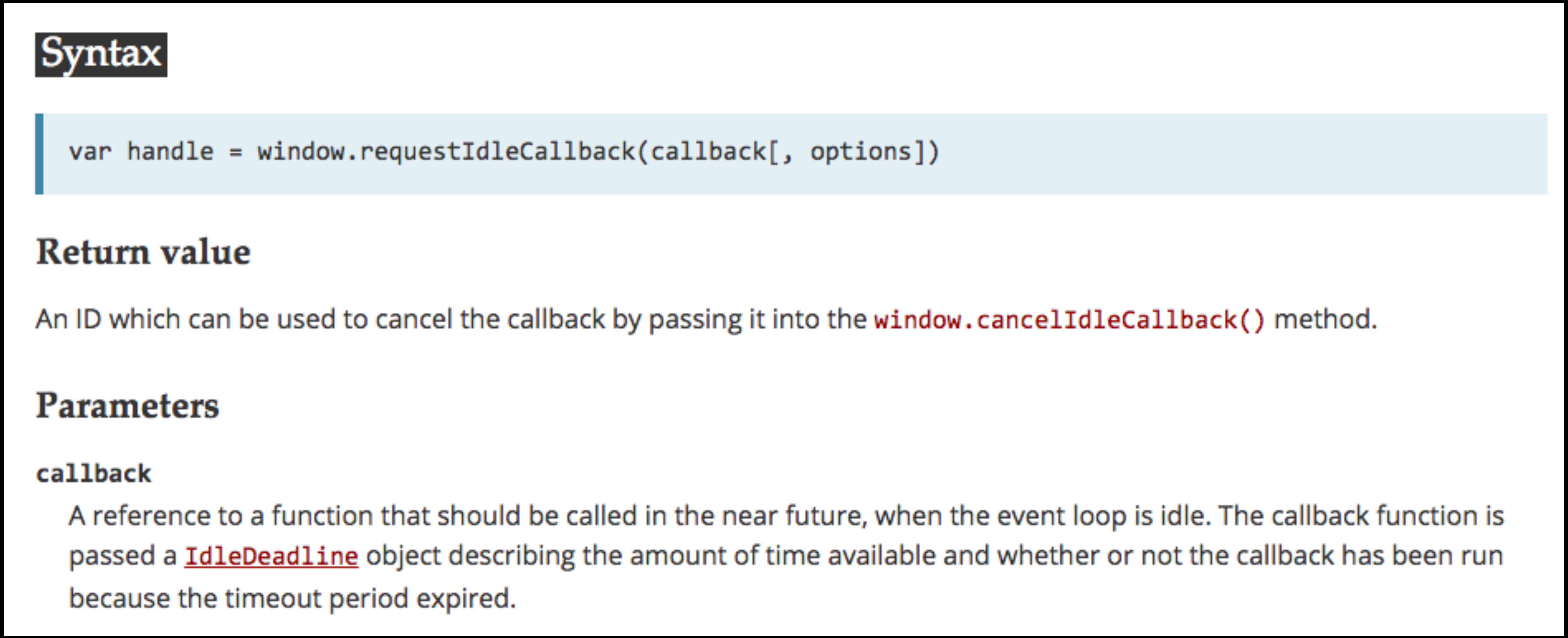
- 以 Fiber 為單位 (可以中斷)
  1. beginWork
  2. completeWork

- 所有 Fiber 處理完才會執行
  3. commitWork

Work In Process



RequestIdleCallback



Syntax

```
var handle = window.requestIdleCallback(callback[, options])
```

Return value

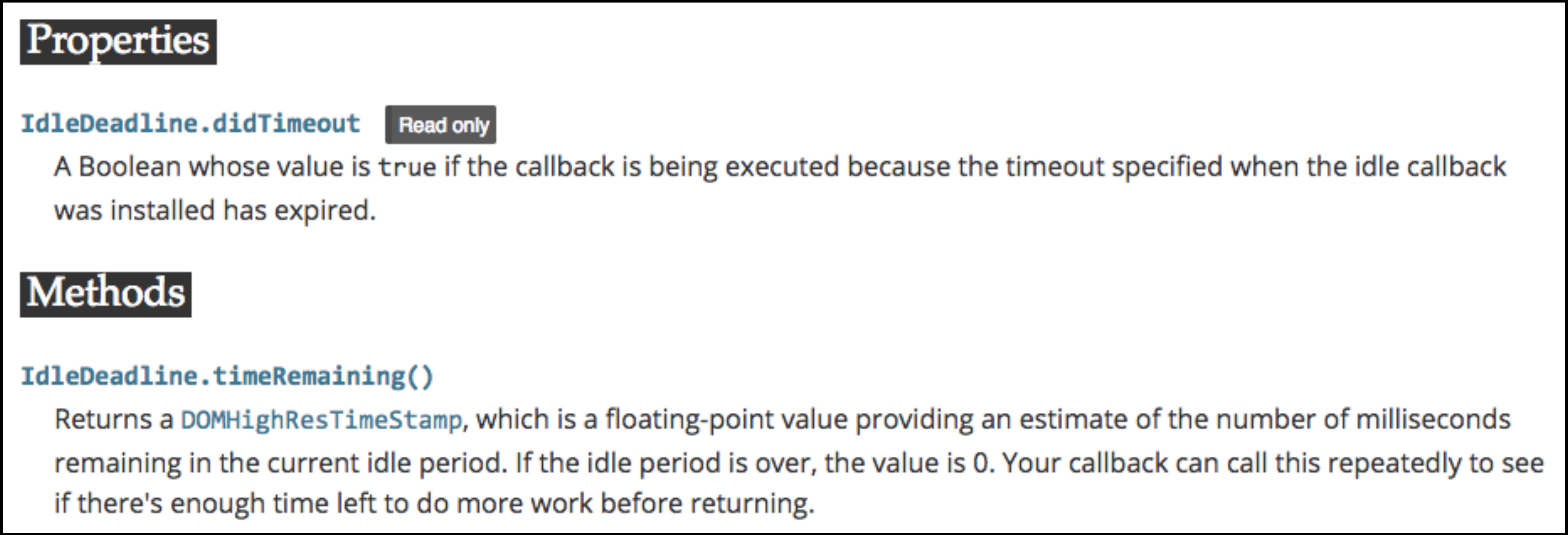
An ID which can be used to cancel the callback by passing it into the `window.cancelIdleCallback()` method.

Parameters

`callback`

A reference to a function that should be called in the near future, when the event loop is idle. The callback function is passed a `IdleDeadline` object describing the amount of time available and whether or not the callback has been run because the timeout period expired.

IdleDeadline



Properties

`IdleDeadline.didTimeout` Read only

A Boolean whose value is `true` if the callback is being executed because the timeout specified when the idle callback was installed has expired.

Methods

`IdleDeadline.timeRemaining()`

Returns a `DOMHighResTimeStamp`, which is a floating-point value providing an estimate of the number of milliseconds remaining in the current idle period. If the idle period is over, the value is 0. Your callback can call this repeatedly to see if there's enough time left to do more work before returning.

```
// react/src/renderers/shared/fiber/ReactFiberScheduler.js

// ...

} else if (deadline !== null) {
  // Flush asynchronous work until the deadline expires.
  while (nextUnitOfWork !== null && !deadlineHasExpired) {
    if (deadline.timeRemaining() > timeHeuristicForUnitOfWork) {
      nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
      // ...
    }
  }
}
```

Priority

```
// react/src/renderers/shared/fiber/ReactPriorityLevel.js

export type PriorityLevel = 0 | 1 | 2 | 3 | 4 | 5;

module.exports = {
  NoWork: 0, // No work is pending.
  SynchronousPriority: 1, // For controlled text inputs.
Synchronous side-effects.
  TaskPriority: 2, // Completes at the end of the current
tick.
  HighPriority: 3, // Interaction that needs to complete
pretty soon to feel responsive.
  LowPriority: 4, // Data fetching, or result from updating
stores.
  OffscreenPriority: 5, // Won't be visible but do the work
in case it becomes visible.
};
```

未來幾個版本可能造成的影響？

React 16

不會啟用 Async Scheduling

對使用者來說幾乎沒有什麼影響

(但 Renderer 的作者應該會覺得難受....)

Error Boundary

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  componentDidCatch(error, info) {
    // Display fallback UI
    this.setState({ hasError: true });
    // You can also log the error to an error reporting
service
    logErrorToMyService(error, info);
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}
```

Return String, Array

```
// 可以回傳 String
class StringComponent extends React.Component {
  render() {
    return 'A string';
  }
}


// 也可以回傳 Array
class ArrayComponent extends React.Component {
  render() {
    return [
      <div>One</div>,
      <div>Two</div>
    ];
  }
}
```

setState

```
// 可以回傳 String
class StringComponent extends React.Component {
  state = {
    count: 0,
  };

  handleClick = () => {
    // 用 function 來 setState 比較好
    this.setState(state => ({
      count: state.count + 1,
    }));

    // 用物件來 setState 之後可能會被拿掉
    // this.setState({ count: this.state.count + 1 });
  };

  render() {
    return <div onClick={this.handleClick} />;
  }
}
```

React 17 ~

可能會啟用 Async Scheduling

(componentWillXXX 的 API 可能會被 call 很多次....)

END