`1. 开发环境配置

1.1 文件夹结构

文件 / 文件夹	描述
src	存储源文件
dist	存储客户端代码打包文件
build	存储服务端代码打包文件
server.js	存储服务器端代码
webpack.config.server.js	服务端 webpack 配置文件
webpack.config.client.js	客户端 webpack 配置文件
babel.config.json	babel 配置文件
package.json	项目工程文件

创建 package.json 文件： `npm init -y`

1.2 安装项目依赖

开发依赖： `npm install webpack webpack-cli webpack-node-externals @babel/core @babel/preset-env @babel/preset-react babel-loader nodemon npm-run-all -D`

项目依赖： `npm install express`

依赖项	描述
webpack	模块打包工具
webpack-cli	打包命令
webpack-node-externals	打包服务器端模块时剔除 node_modules 文件夹中的模块
@babel/core	JavaScript 代码转换工具
@babel/preset-env	babel 预置，转换高级 JavaScript 语法
@babel/preset-react	babel 预置，转换 JSX 语法
babel-loader	webpack 中的 babel 工具加载器
nodemon	监控服务端文件变化，重启应用
npm-run-all	命令行工具，可以同时执行多个命令
express	基于 node 平台的 web 开发框架

1.3 环境配置

1.3.1 创建 web 服务器

```
// server.js
import express from "express"
const app = express()
app.use(express.static("dist"))
const template = `
  <html>
    <head>
      <title>React Fiber</title>
    </head>
    <body>
      <div id="root"></div>
                        <script src="bundle.js"></script>
    </body>
  </html>
`
app.get("*", (req, res) => {
  res.send(template)
})
app.listen(3000, () => console.log("server is running"))
```

1.3.2 服务端 webpack 配置

```
// webpack.config.server.js
const path = require("path")
const nodeExternals = require("webpack-node-externals")
module.exports = {
  target: "node",
  mode: "development",
  entry: "./server.js",
  output: {
    filename: "server.js",
    path: path.resolve(__dirname, "build")
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: "babel-loader"
        }
      }
    ]
  },
  externals: [nodeExternals()]
}
```

1.3.3 babel 配置

```
{
  "presets": ["@babel/preset-env", "@babel/preset-react"]
}
```

1.3.4 客户端 webpack 配置

```
const path = require("path")

module.exports = {
  target: "web",
  mode: "development",
  entry: "./src/index.js",
  output: {
    filename: "bundle.js",
    path: path.resolve(__dirname, "dist")
  },
  devtool: "source-map",
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: "babel-loader"
        }
      }
    ]
  }
}
```

1.3.5 启动命令

```
"scripts": {
  "start": "npm-run-all --parallel dev:*",
  "dev:server-compile": "webpack --config webpack.config.server.js --watch",
  "dev:server": "nodemon ./build/server.js",
  "dev:client-compile": "webpack --config webpack.config.client.js --watch"
},
```

2. requestIdleCallback

2.1 核心 API 功能介绍

利用浏览器的空余时间执行任务，如果有更高优先级的任务要执行时，当前执行的任务可以被终止，优先执行高级别任务。

```
requestIdleCallback(function(deadline) {
  // deadline.timeRemaining() 获取浏览器的空余时间
})
```

2.2 浏览器空余时间

页面是一帧一帧绘制出来的，当每秒绘制的帧数达到 60 时，页面是流畅的，小于这个值时， 用户会感觉到卡顿

1s 60帧，每一帧分到的时间是 1000/60 ≈ 16 ms，如果每一帧执行的时间小于16ms，就说明浏览器有空余时间

如果任务在剩余的时间内没有完成则会停止任务执行，继续优先执行主任务，也就是说 requestIdleCallback 总是利用浏览器的空余时间执行任务

2.3 API 功能体验

页面中有两个按钮和一个DIV，点击第一个按钮执行一项昂贵的计算，使其长期占用主线程，当计算任务执行的时候去点击第二个按钮更改页面中 DIV 的背景颜色。

使用 requestIdleCallback 就可以完美解决这个卡顿问题。

```
<div class="playground" id="play">playground</div>
<button id="work">start work</button>
<button id="interaction">handle some user interaction</button>


<style>
  .playground {
    background: palevioletred;
    padding: 20px;
    margin-bottom: 10px;
  }
</style>
```

```
var play = document.getElementById("play")
var workBtn = document.getElementById("work")
var interactionBtn = document.getElementById("interaction")
var iterationCount = 100000000
var value = 0

var expensiveCalculation = function (IdleDeadline) {
  while (iterationCount > 0 && IdleDeadline.timeRemaining() > 1) {
    value =
      Math.random() < 0.5 ? value + Math.random() : value + Math.random()
    iterationCount = iterationCount - 1
  }
  requestIdleCallback(expensiveCalculation)
}

workBtn.addEventListener("click", function () {
  requestIdleCallback(expensiveCalculation)
})

interactionBtn.addEventListener("click", function () {
  play.style.background = "palegreen"
})
```

3 Fiber

3.1 问题

React 16 之前的版本比对更新 VirtualDOM 的过程是采用循环加递归实现的，这种比对方式有一个问题，就是一旦任务开始进行就无法中断，如果应用中组件数量庞大，主线程被长期占用，直到整棵 VirtualDOM 树比对更新完成之后主线程才能被释放，主线程才能执行其他任务。这就会导致一些用户交互，动画等任务无法立即得到执行，页面就会产生卡顿, 非常的影响用户体验。

核心问题：递归无法中断，执行重任务耗时长。 JavaScript 又是单线程，无法同时执行其他任务，导致任务延迟页面卡顿，用户体验差。

3.2 解决方案

1. 利用浏览器空闲时间执行任务，拒绝长时间占用主线程
2. 放弃递归只采用循环，因为循环可以被中断
3. 任务拆分，将任务拆分成一个个的小任务

3.3 实现思路

在 Fiber 方案中，为了实现任务的终止再继续，DOM比对算法被分成了两部分：

1. 构建 Fiber (可中断)
2. 提交 Commit (不可中断)

DOM 初始渲染: virtualDOM -> Fiber -> Fiber[] -> DOM

DOM 更新操作: newFiber vs oldFiber -> Fiber[] -> DOM

3.4 Fiber 对象

```
{
  type          节点类型 (元素, 文本, 组件)(具体的类型)
  props         节点属性
  stateNode     节点 DOM 对象 | 组件实例对象
  tag           节点标记 (对具体类型的分类 hostRoot || hostComponent || classComponent || functionComponent)
  effects       数组, 存储需要更改的 fiber 对象
  effectTag     当前 Fiber 要被执行的操作 (新增, 删除, 修改)
  parent        当前 Fiber 的父级 Fiber
  child         当前 Fiber 的子级 Fiber
  sibling       当前 Fiber 的下一个兄弟 Fiber
  alternate     Fiber 备份 fiber 比对时使用
}
```