**03-Promise**

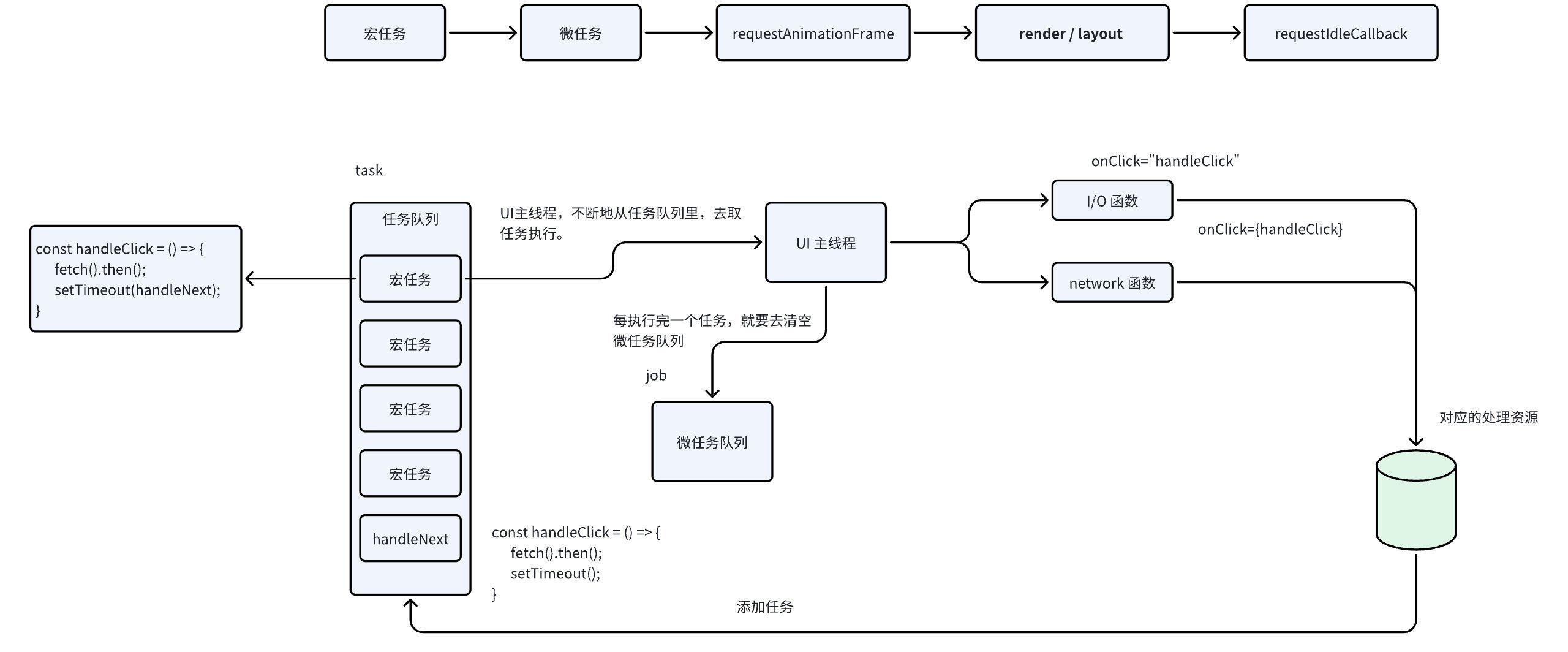
*Async / await 简介*

*Promise A+ 规范*

*实现 Promise*

1. **异步的逻辑**

**浏览器**



**为什么会有微任务？**

* 主线程执行消息队列的宏任务，粒度有点不够，微任务的时效性强；
* 微任务可以改变当前的编程模型；
* 如果数据量大，可以解决一些异步时机不可控的问题。

**异步的发展**

|  |
| --- |
| JavaScript function foo() {  const bar = "bar"; }  // 我如果想要通过调用 foo， 拿到这个 bar。很简单： function foo() {  const bar = "bar";  return bar; }  // 我如果想要通过调用 foo 的 1000ms 以后，再拿到这个 bar， 怎么办？ function foo(cb: Function) {  const bar = "bar";  setTimeout(() => {  cb(bar);  }, 1000) };  const handleFoo = (res) => {  // res 就是我们想要的 bar;  // 但是，我想基于得到 bar 的值以后，再写新的逻辑，只能在这里写了。   }  foo(handleFoo); |

**Callback**

当没有 Promise 的时候，大量的异步逻辑回调，都依赖于，callback

在 node 中，大量这种用法

|  |
| --- |
| JavaScript fs.readFile("a.txt", "utf-8", function(err, data) {  fs.readFile("b.txt", "utf-8", function(err, data) {  fs.readFile("c.txt", "utf-8", function(err, data) {    })  }) }) |

**Promise**

应用 -> fetch， webpack，

|  |
| --- |
| JavaScript //   export const getData = () => post('xxxx/xxx');   getData().then(res => {  // vue   this.dataList = res.data;  // react   setData(res.data) })   // -------------------------------   function post(url) {  return new Promise((resolve, reject) => {  setTimeout(() => {  resolve({data: [1,2,3]})  }, 1000)  }) } |

**Generator**

* 协程，是一种比线程更小的机制，但是本质上使用很少，所以一般也不会问。

|  |
| --- |
| JavaScript function \*gen() {  yield "1st mession";  yield "2nd mession";  let res = yield "3rd mession";  return res; }  let result = gen();  console.log(result.next()); console.log(result.next()); console.log(result.next()); console.log(result.next("over")); |

**async / await**

异步编程的一种重大改进，提供了一种在不阻塞主线程的情况下，使用同步代码直接执行异步的逻辑。

|  |
| --- |
| JavaScript  // -------------------------------   async function post(url) {  return new Promise((resolve, reject) => {  setTimeout(() => {  resolve({data: [1,2,3], url})  }, 1000)  }) }   const getData = async () => await post('xxxx/xxx');  const run = async () => {  console.log('starting...')  const res = await getData();  console.log(res) };  run(); |

**Promise 深入理解**

**初探 promise**

|  |
| --- |
| JavaScript export const getData = () => post('xxxx/xxx'); getData()**.then(res => {**  **// vue**   **this.dataList = res.data;**  **// react**   **setData(res.data)** **}， err => {**   **})**  function post(url) {  **return new Promise((resolve, reject) => {**  **setTimeout(() => {**  **resolve({data: [1,2,3]})**  **}, 1000)**  **})** } |

**规则**

|  |
| --- |
| * Promise 是一个构造函数； * Promise 接受一个函数作为参数，这个函数的参数，是两个函数(resolve, reject) * Promise 返回一个对象，这个对象包含一个 then 函数，这个then 函数，接收两个参数，这两个参数，也都是函数。 * Promise 的 status： * pending * 初始的状态，可以改变 * 一个 Promise 在 resolve 或者 reject 之前，都处于这个状态 * 我们可以通过调用 resolve 或者 reject 方法，让这个 Promise 变成 fulfilled 或者 rejected 的状态。 * fulfilled * 不可变状态 * 在 resolve 之后，变成这个状态，拥有一个 value * rejected * 不可变状态 * 在 reject 之后，变成这个状态，拥有一个 reason * then 函数 * 参数 * onFulfilled, onRejected 必须是函数类型，如果不是，应该被忽略； * onFulfilled 和 onRejected 的特性 * 在 promise 变成 fulfilled / rejected 状态的时候，应该调用 onFulfilled / onRejected； * 在 promise 变成 fulfilled / rejected 状态之前，不应该被调用 * 只能调用一次。 |

**实现**

|  |
| --- |
| JavaScript  function LPromise(execute) {  this.status = "pending";  this.value = null;  this.reason = null;    const resolve = (value) => {  if(this.status === "pending") {  this.value = value;  this.status = "fulfilled";  }  }    const reject = (reason) => {  if(this.status === "pending") {  this.reason = reason;  this.status = "rejected";  }  }    execute(resolve, reject); }  LPromise.prototype.then = function(onFulfilled, onRejected) {  onFulfilled = typeof onFulfilled === "function" ? onFulfilled: (data) => { return data };  onRejected = typeof onRejected === "function" ? onRejected: (error) => { throw error };    if(this.status === "fulfilled") {  onFulfilled(this.value);  }    if(this.status === "rejected") {  onRejected(this.reason);  } } |

**测试：**

|  |
| --- |
| JavaScript new LPromise((resolve, reject) => {  resolve('hello luyi') }).then(res => {  console.log(res) }) |

**问题：**

|  |
| --- |
| JavaScript new LPromise((resolve, reject) => {  setTimeout(() => {  resolve('hello luyi');  }, 1000) }).then(res => {  console.log(res) }) |

啥也没有了。

因为我们在 resolve 执行的时候，then 函数已经执行过了。

* then 是不是要等到我们 resolve 的时候再执行？？？
* 所以，我们要在一个合适的时机，去执行 then 的 onFulfilled
* **发布订阅。**

**Promise 进阶**

**规则**

|  |
| --- |
| * resolve / reject 执行了之后，再执行 onfulfilled 和 onjected； * onfulfilled 和 onjected 应该是微任务。 |

**实现**

|  |
| --- |
| JavaScript  function LPromise(execute) {  this.status = "pending";  this.value = null;  this.reason = null;   this.onFulfilledArray = [];  this.onRejectedArray = [];    const resolve = (value) => {  queueMicrotask(() => {  if(this.status === "pending") {  this.value = value;  this.status = "fulfilled";  this.onFulfilledArray.forEach(func => func(value))  }  })  }    const reject = (reason) => {  queueMicrotask(() => {  if(this.status === "pending") {  this.reason = reason;  this.status = "rejected";  this.onRejectedArray.forEach(func => func(reason))  }  })  }    execute(resolve, reject); }  LPromise.prototype.then = function(onFulfilled, onRejected) {  onFulfilled = typeof onFulfilled === "function" ? onFulfilled: (data) => { return data };  onRejected = typeof onRejected === "function" ? onRejected: (error) => { throw error };    if(this.status === "fulfilled") {  onFulfilled(this.value);  }    if(this.status === "rejected") {  onRejected(this.reason);  }   if(this.status === "pending") {  this.onFulfilledArray.push(onFulfilled);  this.onRejectedArray.push(onRejected);  } } |

**数组 push 的作用和直接拿过来用有啥区别吗？？？**

|  |
| --- |
| JavaScript const p = new LPromise((resolve, reject) => {  setTimeout(() => {  resolve('hello luyi');  console.log("settimeout")  }, 1000) }); p.then(res => {  console.log(res);  return res + "luyi" }); p.then(res => {  console.log(res+"2");  return res + "luyi" }) |



**问题：**

|  |
| --- |
| JavaScript new LPromise((resolve, reject) => {  setTimeout(() => {  resolve('hello luyi');  console.log("settimeout")  }, 1000) }).then(res => {   console.log(res); // hello luyi   return res + "luyi" }).then(res => {   console.log(res) // hello luyi luyi  }) |

以上不work。

**Promise 的链式调用**

**规则**

|  |
| --- |
| * then 方法，应该返回一个 Promise   promise2 = promise1.then(onFulfilled, onRejected)   * onFulfilled / onRejected 的执行结果，为 x， 调用 resolvePromise * 如果 onFulfilled / onRejected 执行时抛出异常，我们 promise2 需要被 reject * 如果 onFulfilled / onRejected 不是一个函数，promise2 以 promise1 的 value 或者 reason 触发 fulfilled 和 rejected   **promise1 中 onfulfilled 返回了一个值，这个值需要被 promise2 进行 resolve ，才能出现在下一个 then(res)。** |

**实现**

|  |
| --- |
| JavaScript  function LPromise(execute) {  this.status = "pending";  this.value = null;  this.reason = null;   this.onFulfilledArray = [];  this.onRejectedArray = [];    const resolve = (value) => {  queueMicrotask(() => {  if(this.status === "pending") {  this.value = value;  this.status = "fulfilled";  this.onFulfilledArray.forEach(func => func(value))  }  })  }    const reject = (reason) => {  queueMicrotask(() => {  if(this.status === "pending") {  this.reason = reason;  this.status = "rejected";  this.onRejectedArray.forEach(func => func(reason))  }  })  }  // try catch   execute(resolve, reject); }  LPromise.prototype.then = function(onFulfilled, onRejected) {  onFulfilled = typeof onFulfilled === "function" ? onFulfilled: (data) => { return data };  onRejected = typeof onRejected === "function" ? onRejected: (error) => { throw error };    let promise2;   if(this.status === "fulfilled") {  return promise2 = new LPromise((resolve, reject) => {  queueMicrotask(() => {  try {  // promise1 中 onfulfilled 返回了一个值，这个值需要被 promise2 进行 resolve ，才能出现在下一个 then(res)  let result = onFulfilled(this.value);  resolve(result);  } catch(e) {  reject(e)  }  })  })    }    if(this.status === "rejected") {  return promise2 = new LPromise((resolve, reject) => {  queueMicrotask(() => {  try {  // promise1 中 onfulfilled 返回了一个值，这个值需要被 promise2 进行 resolve ，才能出现在下一个 then(res)  let result = onRejected(this.reason);  resolve(result);  } catch(e) {  reject(e)  }  })  })  }   if(this.status === "pending") {  return promise2 = new LPromise((resolve, reject) => {  this.onFulfilledArray.push(() => {  try {  let result = onFulfilled(this.value);  resolve(result);  } catch(e) {  reject(e)  }  });  this.onRejectedArray.push(() => {  try {  // promise1 中 onfulfilled 返回了一个值，这个值需要被 promise2 进行 resolve ，才能出现在下一个 then(res)  let result = onRejected(this.reason);  resolve(result);  } catch(e) {  reject(e)  }  });  })    } } |

**选读：resolvePromise 规范**

**规则**

resolvePromise(promise2, x, resolve, reject)

|  |
| --- |
| * 如果 promise2 和 x 相等，那么 reject error; * 如果 promise2 是一个 promise * 如果 x 是一个pending 状态，那么 promise2 必须要再 pending, 直到 x 变成 fulfilled / rejected * 如果 x 被 fulfilled， fulfill promise with the same value * 如果 x 被 rejected， reject promise with the same reason * 如果 x 是一个 object 或者 function * Let thenable = x.then * 如果 x.then 这一步出错，那么 reject promise with e as the reason * 如果 then 是一个函数，then.call(x, resolvePromiseFn, rejectPromiseFn) * resolvePromiseFn 的入参是y, 执行 resolvePromise(promise2, y, resolve, reject) * rejectPromiseFn 的入参是 r, reject promise with r * 如果 resolvePromiseFn 和 rejectPromiseFn 都调用了，那么第一个调用优先，后面的忽略 * 如果调用then 抛出异常 * 如果 resolvePromise 或 rejectPromise 已经被调用，可以忽略 * 如果 then 不是一个 function， fulfill promise with x |

**实现**

|  |
| --- |
| JavaScript  const resolvePromise = (promise2, result, resolve, reject) => {  // 当 result 和 promise2 相等时，也就是说 onfulfilled 返回 promise2 时，进行 reject  if (result === promise2) {  reject(new TypeError('error due to circular reference'))  }    // 是否已经执行过 onfulfilled 或者 onrejected  let consumed = false  let thenable    if (result instanceof LPromise) {  if (result.status === 'pending') {  result.then(function(data) {  resolvePromise(promise2, data, resolve, reject)  }, reject)  } else {  result.then(resolve, reject)  }  return  }    let isComplexResult = target => (typeof target === 'function' || typeof target === 'object') && (target !== null)    // 如果返回的是疑似 Promise 类型  if (isComplexResult(result)) {  try {  thenable = result.then  // 如果返回的是 Promise 类型，具有 then 方法  if (typeof thenable === 'function') {  thenable.call(result, function(data) {  if (consumed) {  return  }  consumed = true    return resolvePromise(promise2, data, resolve, reject)  }, function(error) {  if (consumed) {  return  }  consumed = true    return reject(error)  })  }  else {  resolve(result)  }    } catch(e) {  if (consumed) {  return  }  consumed = true  return reject(e)  }  }  else {  resolve(result)  } }  function LPromise(execute) {  this.status = "pending";  this.value = null;  this.reason = null;   this.onFulfilledArray = [];  this.onRejectedArray = [];    const resolve = (value) => {  queueMicrotask(() => {  if(this.status === "pending") {  this.value = value;  this.status = "fulfilled";  this.onFulfilledArray.forEach(func => func(value))  }  })  }    const reject = (reason) => {  queueMicrotask(() => {  if(this.status === "pending") {  this.reason = reason;  this.status = "rejected";  this.onRejectedArray.forEach(func => func(reason))  }  })  }  // try catch   execute(resolve, reject); }  LPromise.prototype.then = function(onFulfilled, onRejected) {  onFulfilled = typeof onFulfilled === "function" ? onFulfilled: (data) => { return data };  onRejected = typeof onRejected === "function" ? onRejected: (error) => { throw error };    let promise2;   if(this.status === "fulfilled") {  return promise2 = new LPromise((resolve, reject) => {  queueMicrotask(() => {  try {  // promise1 中 onfulfilled 返回了一个值，这个值需要被 promise2 进行 resolve ，才能出现在下一个 then(res)  let x = onFulfilled(this.value);  **resolvePromise(promise2, x, resolve, reject)**  } catch(e) {  reject(e)  }  })  })    }    if(this.status === "rejected") {  return promise2 = new LPromise((resolve, reject) => {  queueMicrotask(() => {  try {  // promise1 中 onfulfilled 返回了一个值，这个值需要被 promise2 进行 resolve ，才能出现在下一个 then(res)  let x = onRejected(this.reason);  **resolvePromise(promise2, x, resolve, reject)**  } catch(e) {  reject(e)  }  })  })  }   if(this.status === "pending") {  return promise2 = new LPromise((resolve, reject) => {  this.onFulfilledArray.push(() => {  try {  let x = onFulfilled(this.value);  **resolvePromise(promise2, x, resolve, reject)**  } catch(e) {  reject(e)  }  });  this.onRejectedArray.push(() => {  try {  // promise1 中 onfulfilled 返回了一个值，这个值需要被 promise2 进行 resolve ，才能出现在下一个 then(res)  let x = onRejected(this.reason);  **resolvePromise(promise2, x, resolve, reject)**  } catch(e) {  reject(e)  }  });  })    } } |

**Promise 的一些理解**

**链式执行**

*100 个 promise，10个先执行，每 resolve 一个，加一个进去。形成 stream.*

|  |
| --- |
| JavaScript const promiseArrGenerator = (num) =>   new Array(num).fill(0).map((item, index) => () => new Promise((resolve, reject) => {  setTimeout(() => {  resolve(index)  }, Math.random() \* 1000)  }))  let arr = promiseArrGenerator(100);  // arr.map((fn) => { // fn().then(console.log) // }) // Promise.all(arr.map(fn => fn())).then(res => console.log(res))  // 设计一个 promise Chain 链式调用  const promiseChain = (arr) => {  arr.reduce((proChain, pro) => proChain.then(res => {  ~res && console.log(res);  return pro()  }), Promise.resolve(-1)) }  promiseChain(arr) |

**手动触发**

*sleep 函数，halk 函数*

|  |
| --- |
| JavaScript const engine = (cb) => {  let \_resolve;   new Promise((resolve, reject) => {  \_resolve = resolve;  }).then(res => {  cb()  })   return {  start: () => {  \_resolve()  }  } }  let e = engine(() => {  console.log("engine") })  e.start() |