### Eventloop

#### 1.进程and线程

进程：指程序在操作系统中运行的实例。实例是指同一个程序可以同时在操作系统里实际的运行。

线程：进程是由一个或多个线程组成，在linux中线程被实现为 “轻量级进程”

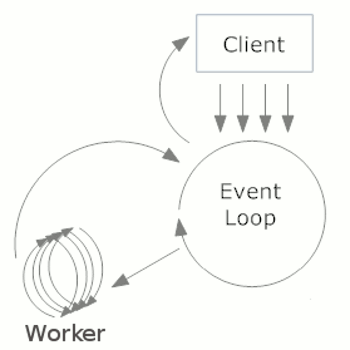
进程是资源（CPU、内存等）分配的基本单位。

每个线程有自己的堆栈和局部变量。线程由CPU独立调度执行，在多CPU环境下就允许多个线程同时运行。

|  |  |
| --- | --- |
| 进程： | 线程： |
| 一个应用程序一个进程 | 一个进程可以有多个线程 |
| 资源分配的最小单位 | 执行程序的最小单元 |
| 通过多个线程占据系统资源 | CPU调度和分配的基本单位 |
| 进程之间的虚拟内存彼此隔离 | 线程之间共享同一虚拟内存空间 |

#### 2.Event Loop

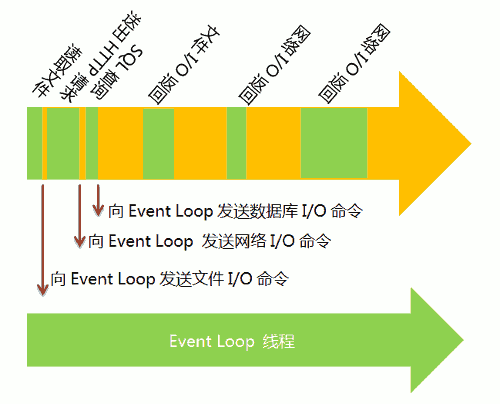
EventLoop是一个程序结构，用于等待和发送消息事件。



JavaScript是一种单线程语言，所有任务都在一个线程上完成，一旦遇到大量任务或者遇到一个耗时的任务，网页就会出现“假死”。

Event Loop就是为了解决一下两个问题而出现的，单线程阻塞问题和多线程浪费资源问题。

程序中设置两个线程，一个负责程序本身的运行，成为“主线程”；另一个负责主线程与其他进程的通信，称为“Event Loop线程”。



主线程绿色部分表示运行时间，橙色部分表示空闲时间。每当遇到I/O的时候，主线程就让Event Loop线程去通知相应的I/O程序，然后主线程接着往下进行，不存在阻塞问题，等到I/O程序完成操作，Event Loop线程再把结果返回主线程，主线程直接调用回调函数，完成任务。这就是异步。

#### 3.微任务与宏任务

优先级：主线程>微任务>宏任务

主线程任务完成以后，去微任务队列读取程序，然后执行微任务，微任务队列为空时在去读取宏任务队列。

//  同步任务

console.log('hello')

//  宏任务

    setTimeout(() => {

      console.log('定时器');

},0 );

//  微任务

    Promise.resolve()

    .then(value => {

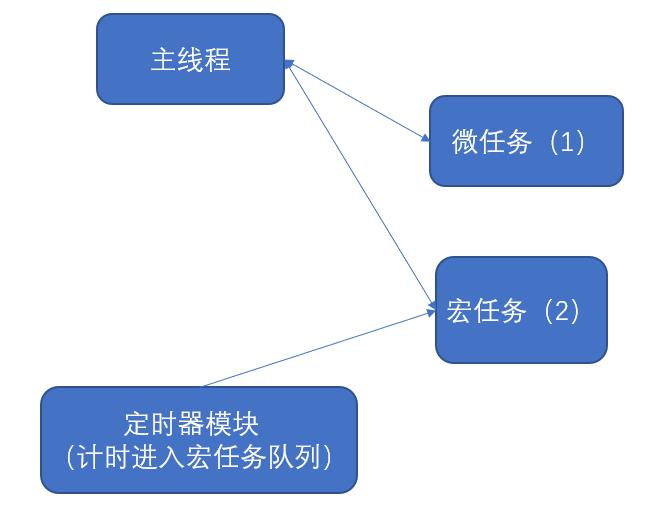
      console.log('promise');

})

//  hello promise 定时器

定时器任务编排：

定时器函数有一个定时器模块，读取时先进入定时器模块，按照时间长短依次进入宏任务队列，当主线程中没有任务时接着执行定时器函数。



// 两个定时器函数根据时间输出

setTimeout(() => {

  console.log('定时器1');

}, 2000);

setTimeout(() => {

  console.log('定时器2');

}, 1000);

console.log('hello');

for (let i = 0; i < 1000; i++) {

  console.log('');

}

//  hello  定时器2  定时器1

任务是共享内存的：任务是一个一个执行的，并且共享一个内存，所以后面使用的是前面处理的结果。

//共享内存

let i = 0

setTimeout(() => {

  console.log(++i);

}, 1000);

setTimeout(() => {

  console.log(++i);

}, 1000);

//  1   2

Promise微任务处理逻辑：

还是一样的逻辑，先处理同步任务，在处理微任务，最后处理宏任务，此时宏任务里继续递归，先处理里面的同步任务，微任务，宏任务。

setTimeout(() => {

  console.log('定时器');

  new Promise(resolve => {

    console.log('setTimeout Promise');

    resolve()

  }).then(() => {

    console.log('setTimeout then');

  })

}, 0);

new Promise((resolve) => {

  console.log('Promise');

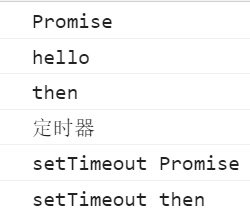
  resolve()

}).then(() => {

  console.log('then');

})

console.log('hello');



DOM渲染任务也是宏任务，所以先把加载script标签放在渲染的后面，视觉效果更好。

如果任务计算时间比较长，不想影响同步任务的执行，可以使用setTimeout宏任务或者promise.then()微任务。

// 使用promise微任务

async function hd(num) {

  let res = await Promise.resolve().then(\_ => {

    let count = 0

    for(let i = 0; i < num; i++){

      count += num--

    }

    return count

  })

  console.log(res);

}

hd(9876543221)

console.log('hello');

// hello  36579789756012884000

使用setTimeout宏任务

// 使用setTimeout宏任务

function sum(num) {

  return new Promise(resolve => {

    setTimeout(() => {

     let count = 0

     for(let i = 0; i < num; i++){

       count += num--

     }

     resolve(count)

    });

  })

}

async function hd(num) {

  let res = await sum(num)

  console.log(res);

}

hd(9876543221)

console.log('hello');

// hello  36579789756012884000