node.js核心介绍与实战技巧

讲师：崔凯

2019年8月

开场

- 自我介绍
- 课程目标
  - 能学到：
    - 了解node.js的基本原理，以后能独立解决疑难问题、做简单的性能优化
    - 回去之后对工作中以前用过的框架，有更深入的了解
    - 打破对node.js的理解障碍，在学习新框架的时候更加得心应手
    - 面对新业务的技术选型，有更多底气和更准确的判断
  - 不能学到：
    - 具体的xxx框架怎么去使用
    - xx框架有哪些坑
- 课堂上
  - 欢迎随时打断提问

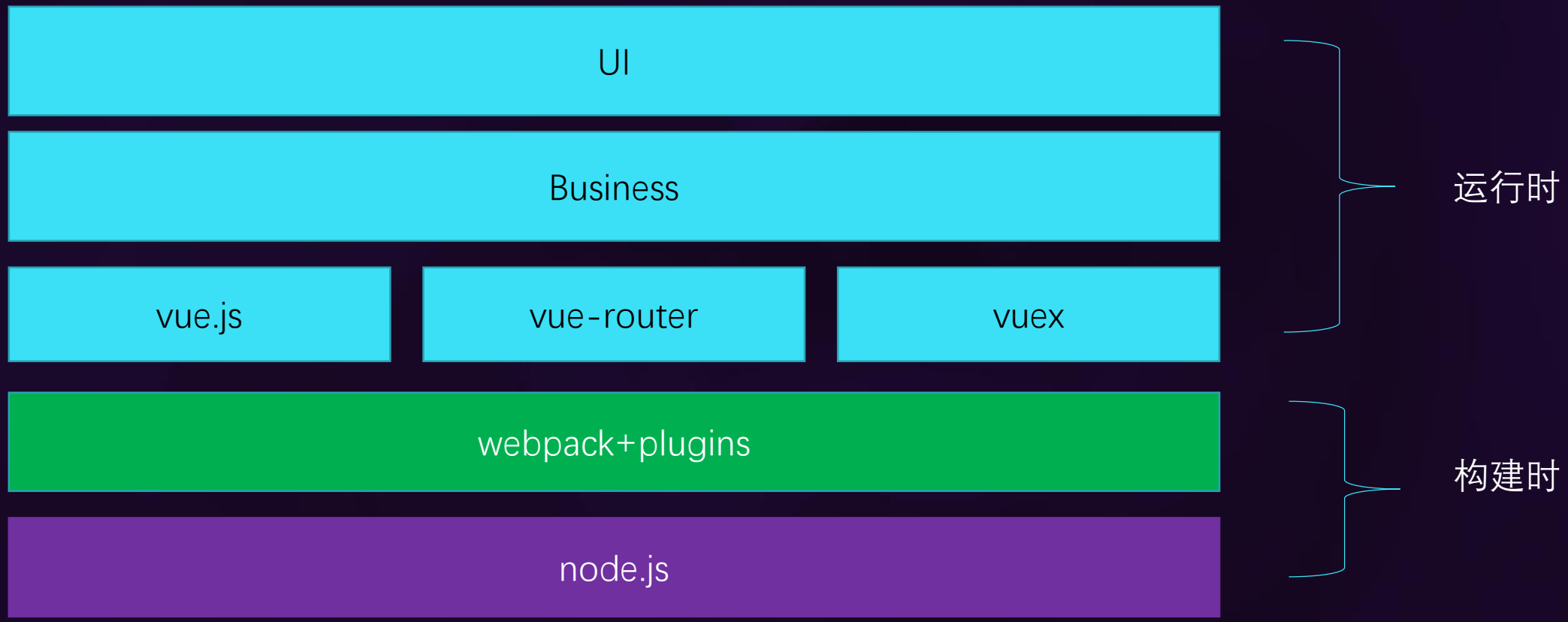
目录

CONTENTS

1 node.js架构概览

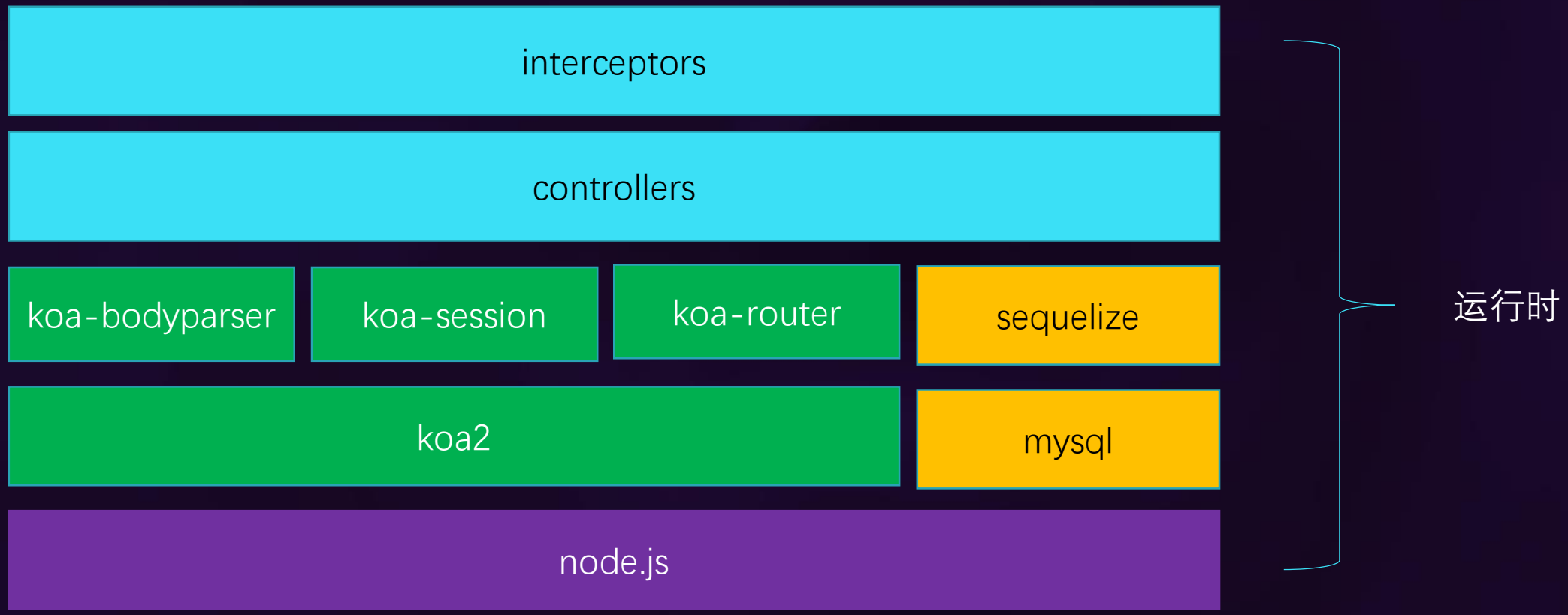
纯前端项目中的node.js

- 一个web项目的前端架构



后台服务项目中的node.js

- 一个web项目的后端架构

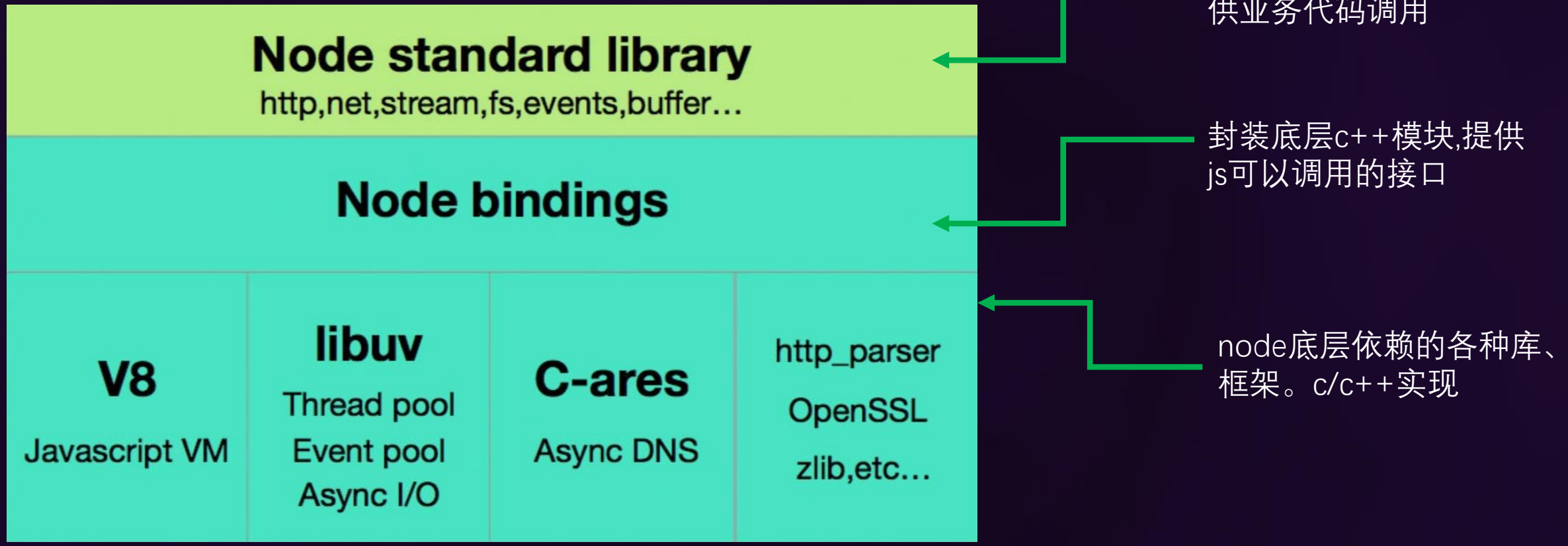


不同角色对node.js的了解

- 纯前端开发人员
  - 跑webpack要先安装node.js
  - 用npm需要先安装node.js
  - 曾经用node.js做过hello world
- node的web服务开发人员
  - 会用express，Koa等各种框架
  - 了解各大厂的一些定制框架和玩法
- 关于node.js内部
  - node.js到底是个啥?
  - 为啥他可以访问文件系统?
  - 单线程咋还能高并发?
  - 出现奇怪的问题怎么解决?

node.js架构概览

- 下面是node.js的内部架构图



2 IO模型与事件循环

IO模型与事件循环——IO模型

- 问题

1. 有做过android/iOS/Java/Winform/WPF/的小伙伴吗
2. C10K问题?
3. 现代浏览器要求音频播放必须用户行为触发，那为什么使用计时器来保持用户行为触发的回调，没有效果?

IO模型与事件循环——IO模型

- 问题二
  - 我们知道javascript是单线程的，那下面的代码中，node.js是如何处理并发请求任务的?
  - “同步/异步”和线程的”阻塞/非阻塞”如何理解?

```
async function (ctx, next) {
    const self = this;
    try {

        const toCreate = ctx.request.body;
        // 调用service获取返回数据
        const result = await sysService.createAccess(ctx,toCreate);
        ctx.body = result;
    } catch (e) {
        resp.failed({ desc: e.stack || e.toString() }, ctx);
    } finally {
        // 执行流程交给下一个middle-ware
        await next();
    }
}
```

IO模型与事件循环——IO模型

- 常见的误区——混淆IO模型与编程范式

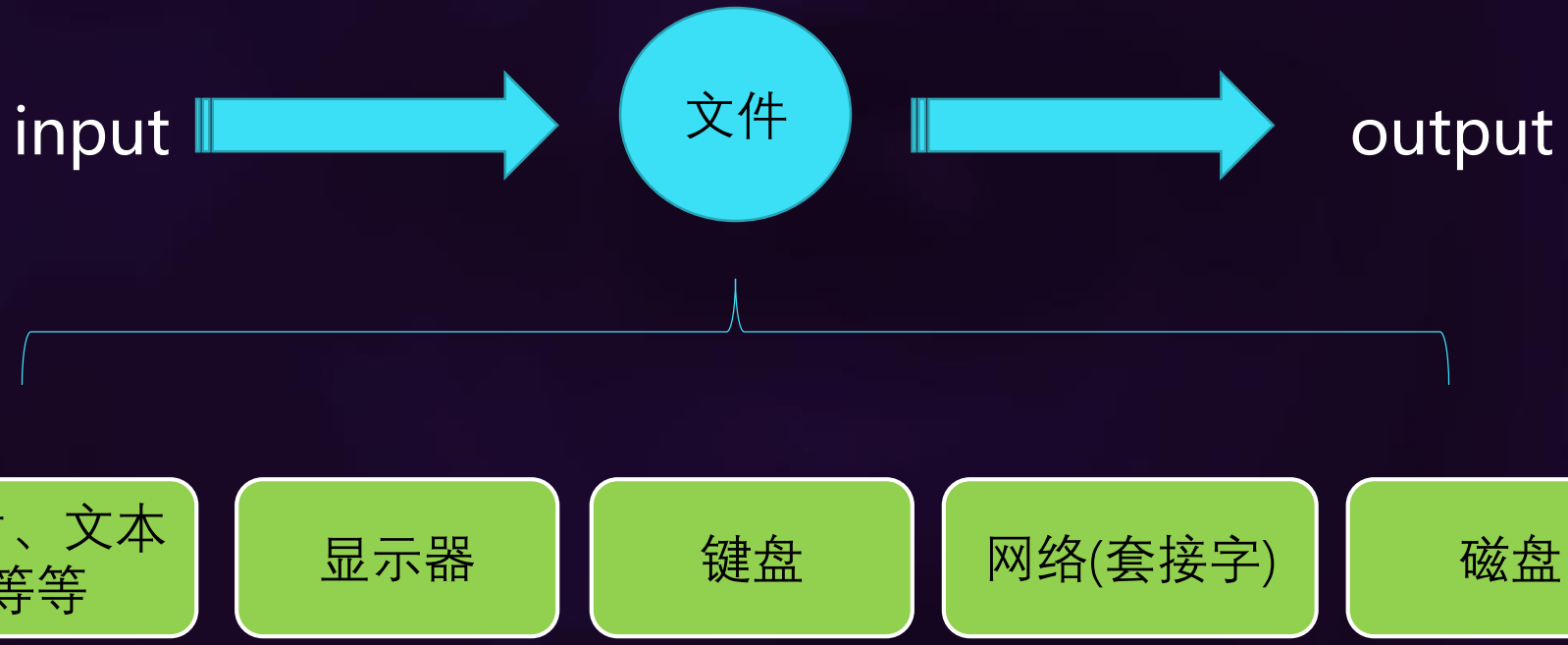
异步回调的编程范式

≠

异步IO

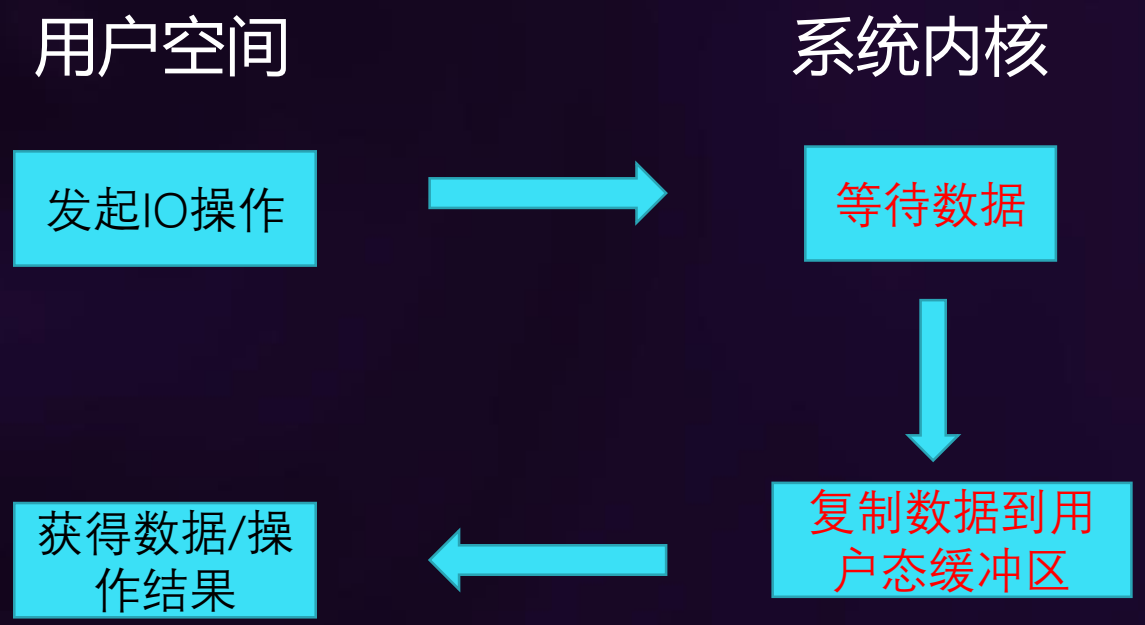
IO模型与事件循环——IO模型

- 什么是IO
  - I=Input
  - O=Output



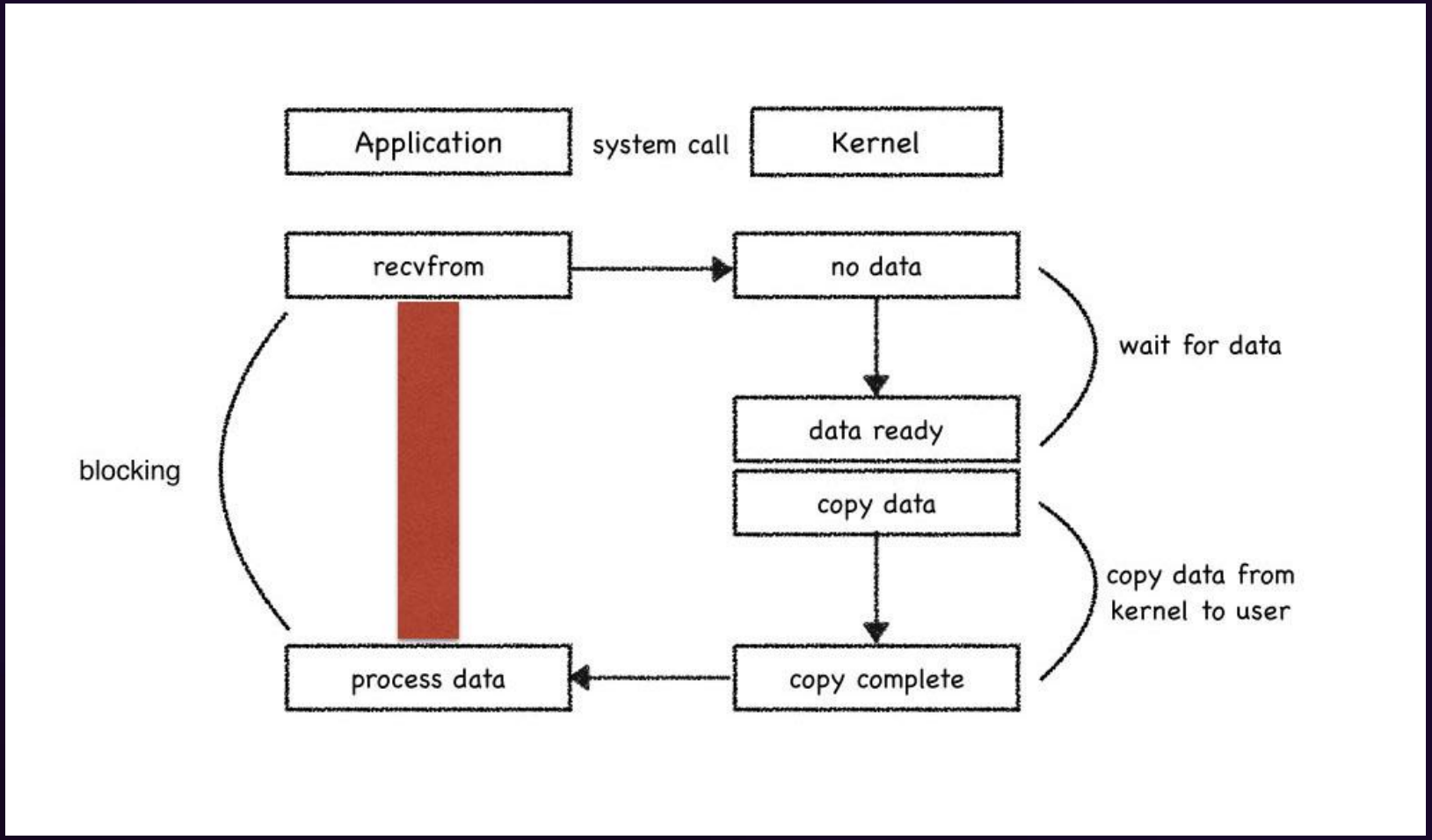
IO模型与事件循环——IO模型

- 什么是IO模型
  - 用户态线程从开始发起一个IO操作，到等待Kernel完成该操作并将数据copy到用户空间，所需要经过的步骤，以及所涉及的API
- 几种IO模型
  - 阻塞式IO
  - 非阻塞式IO
  - IO复用
  - 信号驱动式IO
  - 异步IO
- 区别
  - 等待数据的过程是否阻塞
  - 拷贝数据的过程是否阻塞



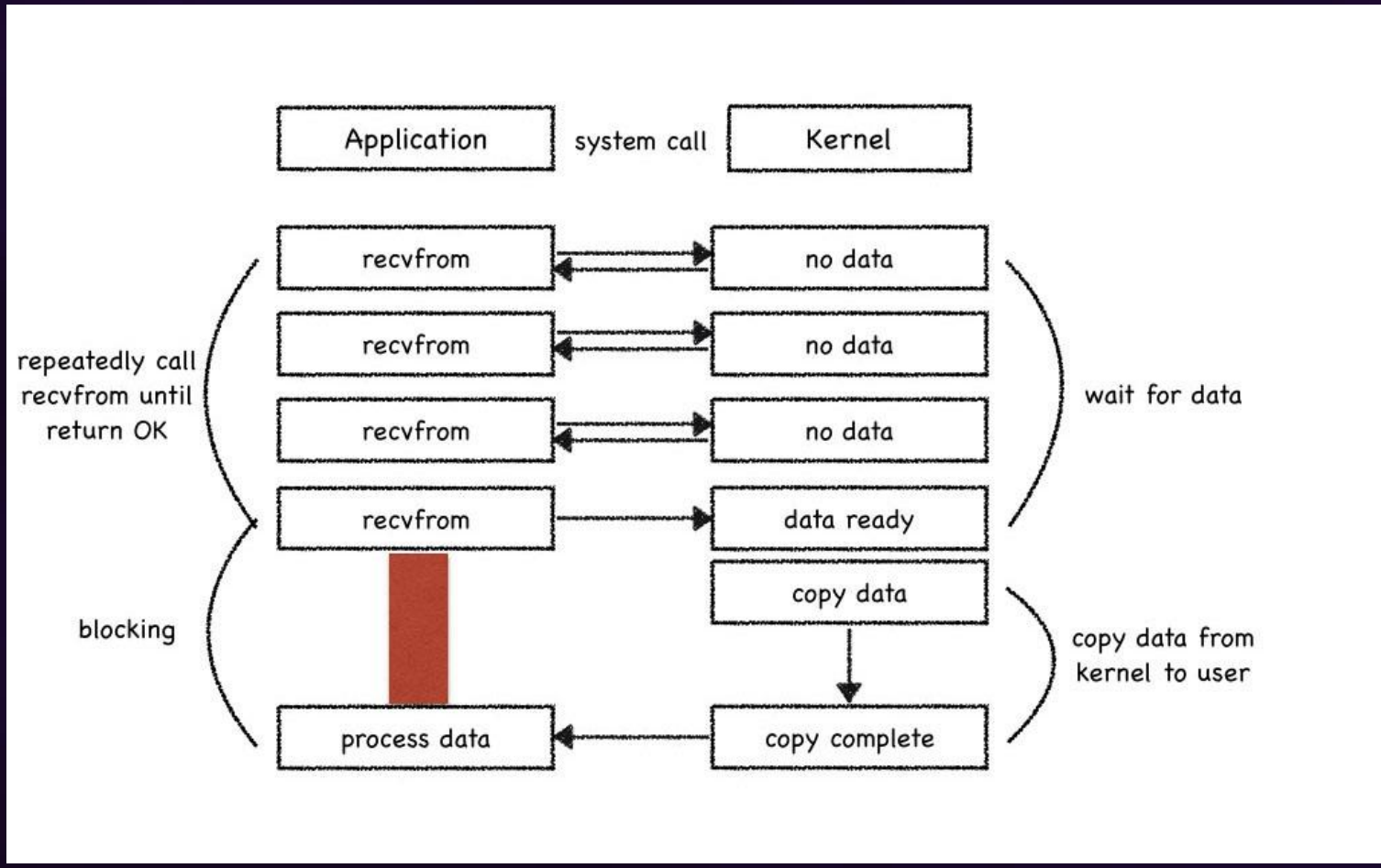
IO模型与事件循环——IO模型

- 阻塞IO(Blocking IO)
  - 用户线程一直阻塞
  - 数据到达，并且拷贝到用户态之后，用户线程可以继续后续操作



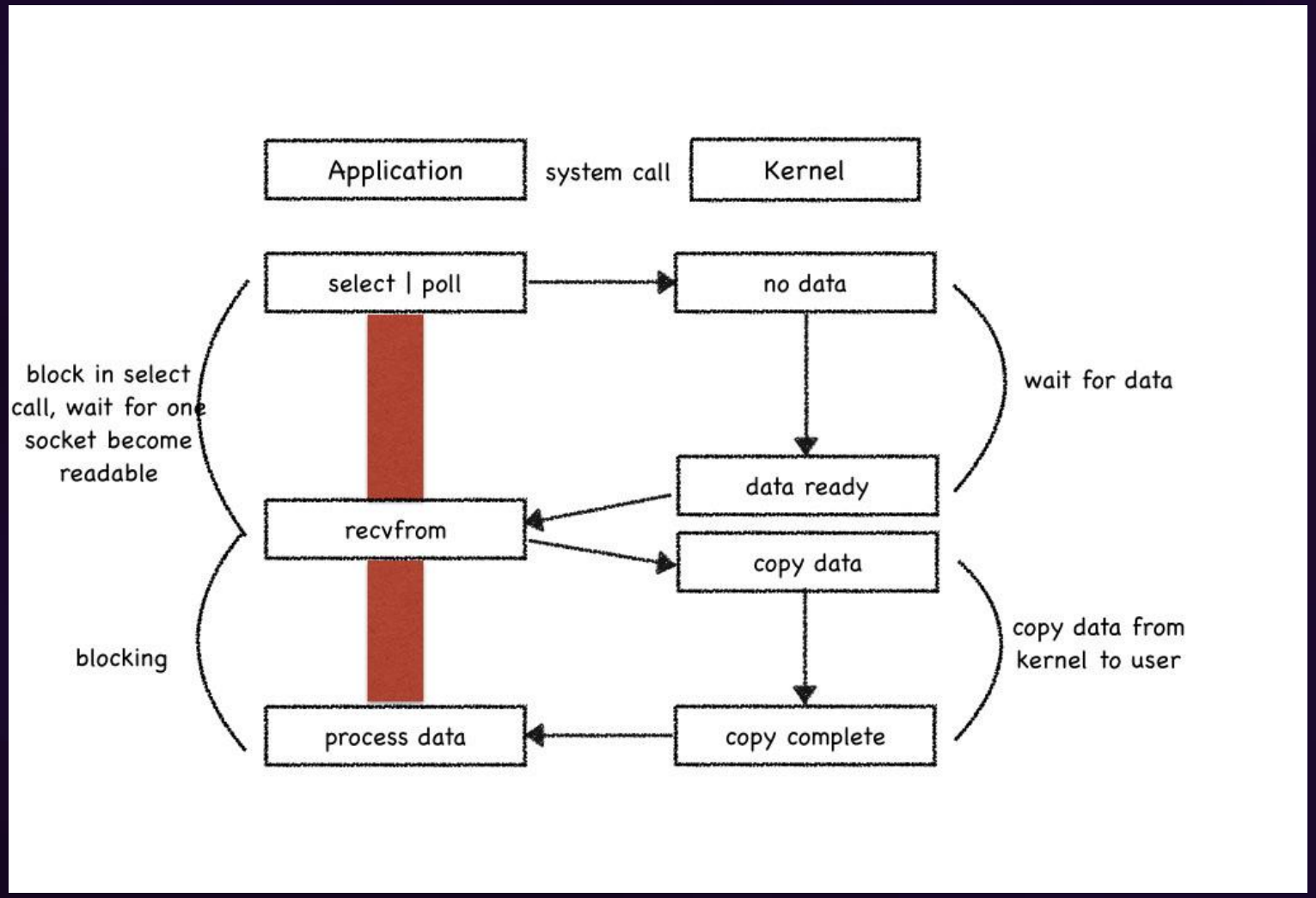
IO模型与事件循环——IO模型

- 非阻塞IO(Non-Blocking IO)
  - 数据未就绪时，用户线程不被阻塞
  - 数据就绪时，用户线程会阻塞，直到数据拷贝到用户态
  - 用户线程需要采用轮询的方式来获取数据的就绪状态



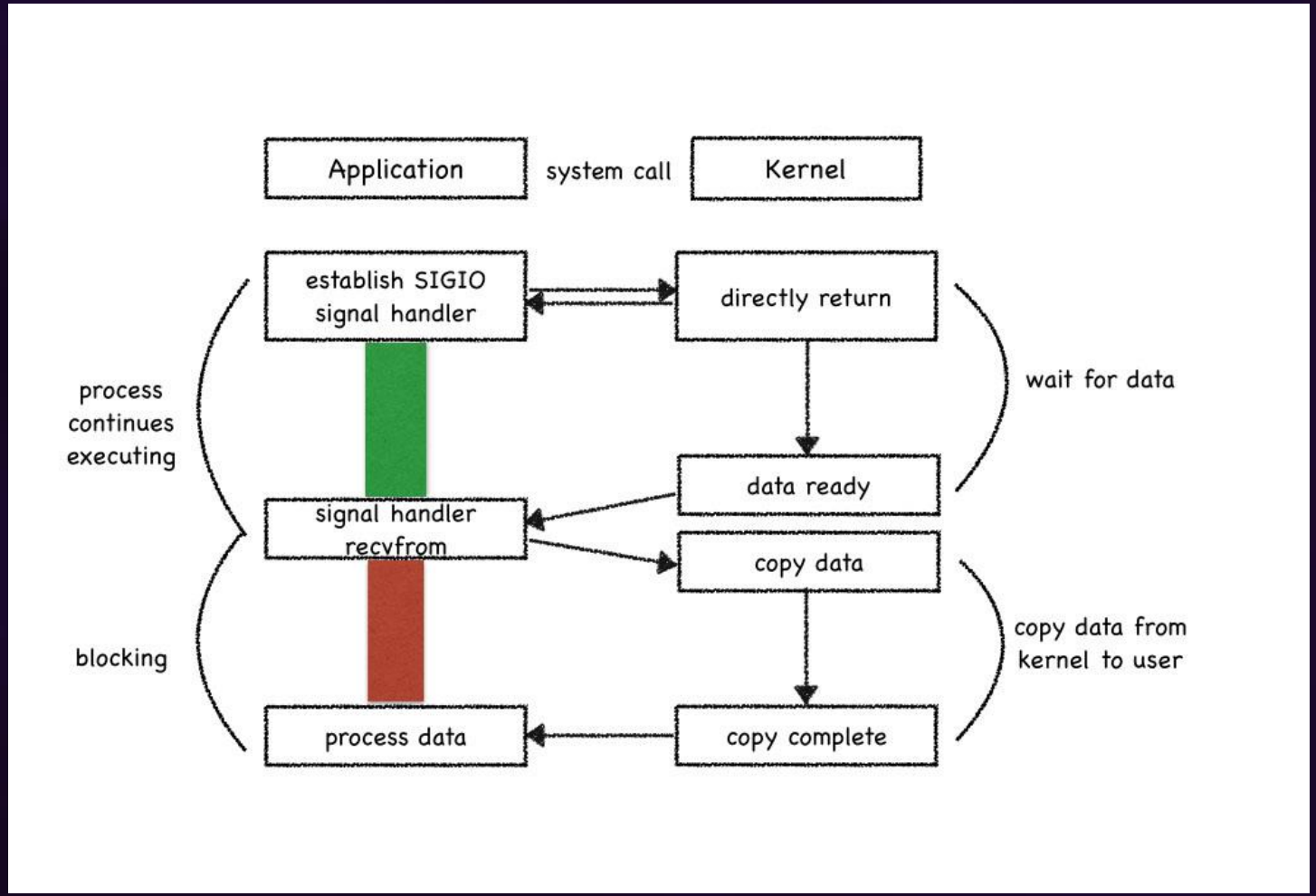
IO模型与事件循环——IO模型

- IO复用(IO Multiplexing)
  - 用户线程可以一次监听多个socket状态
  - 监听过程、数据拷贝过程都会阻塞当前线程



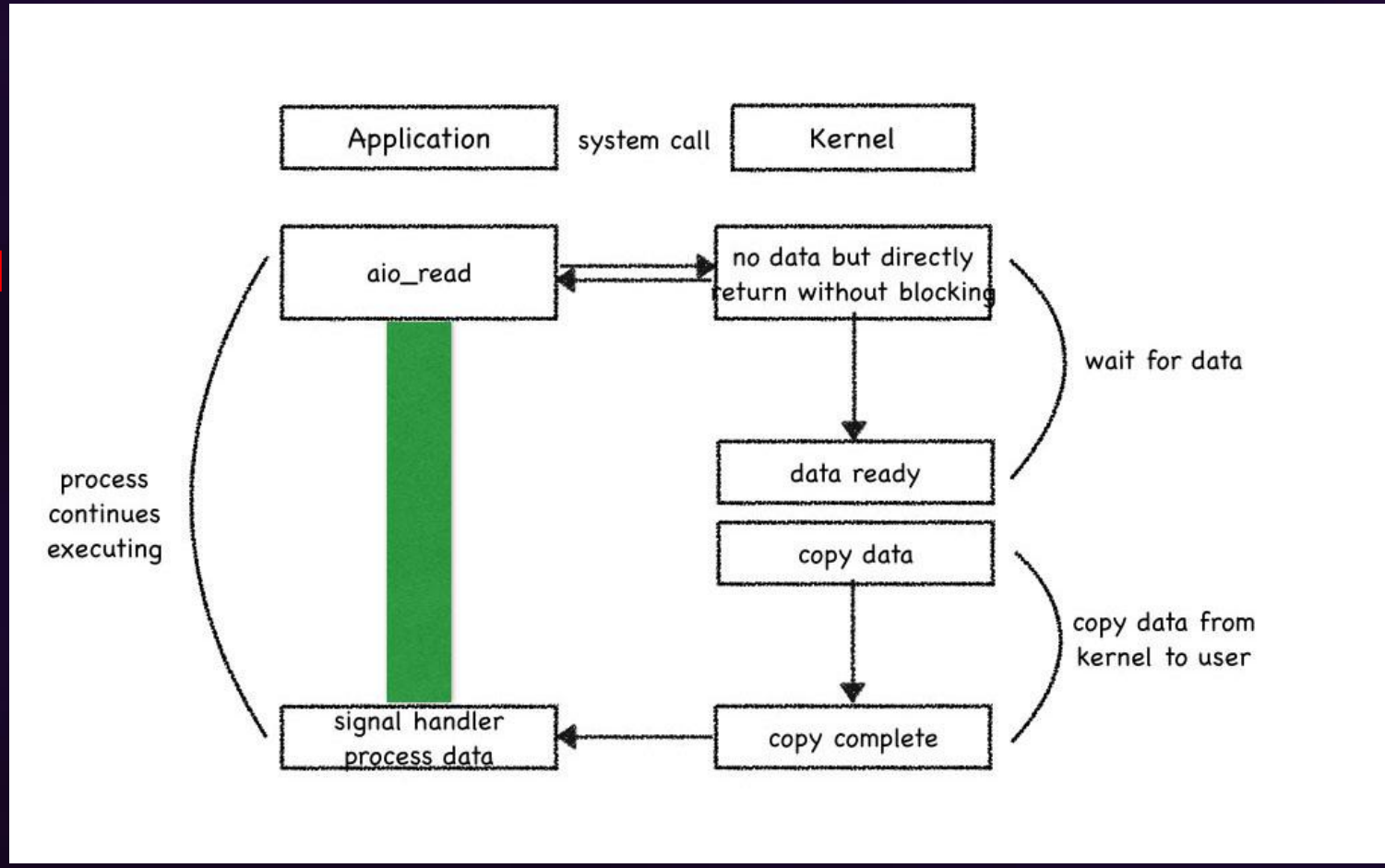
IO模型与事件循环——IO模型

- 信号驱动IO(Signal-driven I/O)
  - 用户线程在数据等待过程中不被阻塞
  - 拷贝数据到用户态的过程，仍然是阻塞的
  - 用户态线程无需轮询，通过注册signal handler收到kernel的通知



IO模型与事件循环——IO模型

- 异步IO(Asynchronous I/O)
  - POSIX定义的标准模型
  - 等待数据、拷贝数据到用户态的过程，都是非阻塞的
  - 用户态线程无需轮询，通过注册signal handler收到kernel的通知



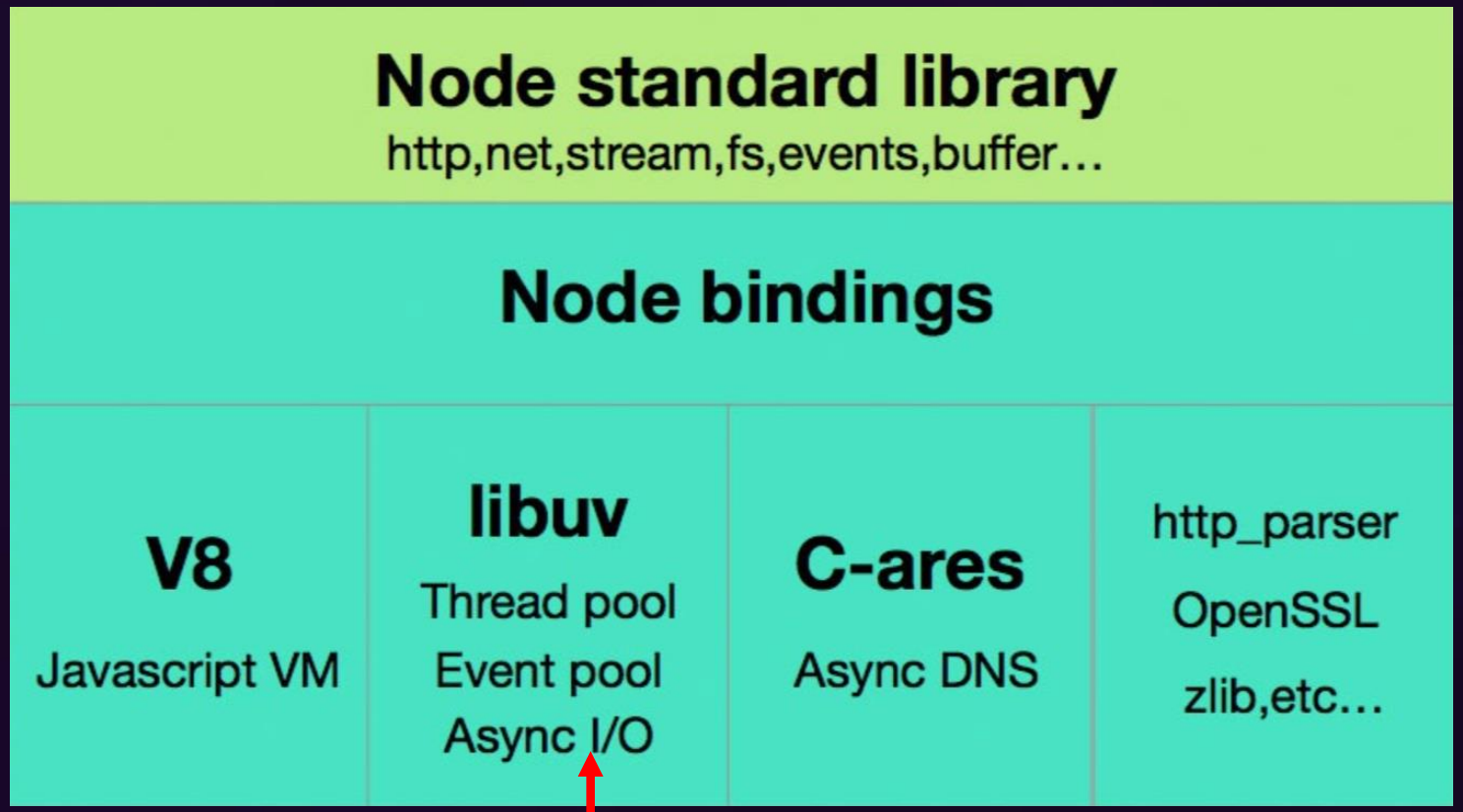
IO模型与事件循环——IO模型

- 形象一点的例子（打电话催快递场景）

Io模型	场景举例
阻塞IO	你有一个包裹，你打电话给快递员，快递员说东西还在送。你就这样一直拿着电话，等包裹送到你手上，挂断电话
非阻塞IO	你有一个包裹，你打电话给快递员，如果东西没到，你可以挂电话去干别的。 但是你每隔10分钟打个电话问到了没有，终于第100次电话中，对方说到了，你等我给你送上来，电话别挂！（此时你啥也干不了）
Io复用	你有20个包裹（分别20个快递员负责），你打电话给快递公司问到了没有，对方说还在送。 你就这样一直拿着电话，直到有一个快递员把包裹送到你手上，你挂了电话。（当然你还可以再去一个电话监听剩下的19个包裹）
信号驱动IO	你有一个包裹，你打电话给快递员，快递员说别急到了我回电话给你。你挂断后去忙别的，10分钟后快递员给你电话：“东西到楼下了，自己来拿吧”。你这时候自己去拿包裹。
异步IO	你有一个包裹，你打电话给快递员，快递员说别急到了我送上来给你。你挂断后去忙别的，10分钟后快递员带着包裹走到你的面前。

IO模型与事件循环——IO模型

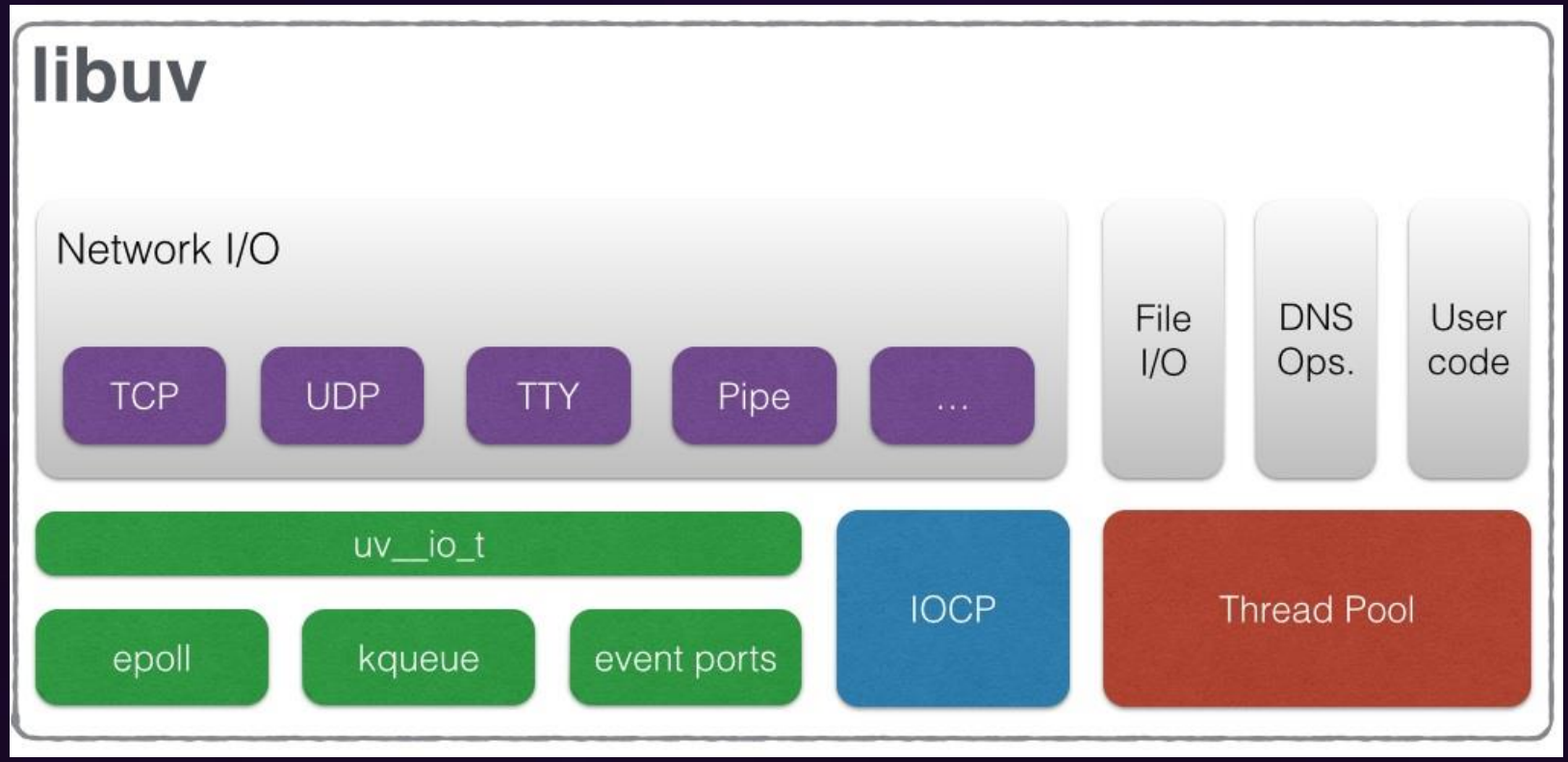
- node.js的IO模型是由libuv库实现



node借助libuv来实现网络和文件IO

IO模型与事件循环——IO模型

- libuv的架构



libuv对node.js层屏蔽了OS的细节，提供了统一标准的IO的API

IO模型与事件循环——小结

- 记住重点
  - AIO(异步IO)对用户线程的影响最小，其性能最好
  - AIO和Signal的方式很接近，同时他们也都需要用户线程有“注册回调handler”的能力
  - 编程范式的“异步”，并不代表IO模型的“异步”，是两个不同的概念
  - node.js使用libuv库，来实现网络/文件IO

	Blocking IO	Non Blocking IO	IO Multiplexing	Signal-driven I/O	IO(Asynchronous I/O
获取数据就绪状态	阻塞	非阻塞	阻塞	非阻塞	非阻塞
拷贝数据到用户态	阻塞	阻塞	阻塞	阻塞	非阻塞
主动查询/被动通知	主动	主动	主动	被动	被动

五种IO模型的关键区别

IO模型与事件循环——小结

- 疑问
  - 在上一节，我们知道：AIO需要注册signal handler，这个回调函数保存在哪里？谁来执行?
  - 我们写页面时，常用的：setTimeout、setInterval,在node.js里好像也可以用，为什么?
  - 下面的2段代码，为什么第二种可以让进程一直不退出?

event-loop-01.js > ...

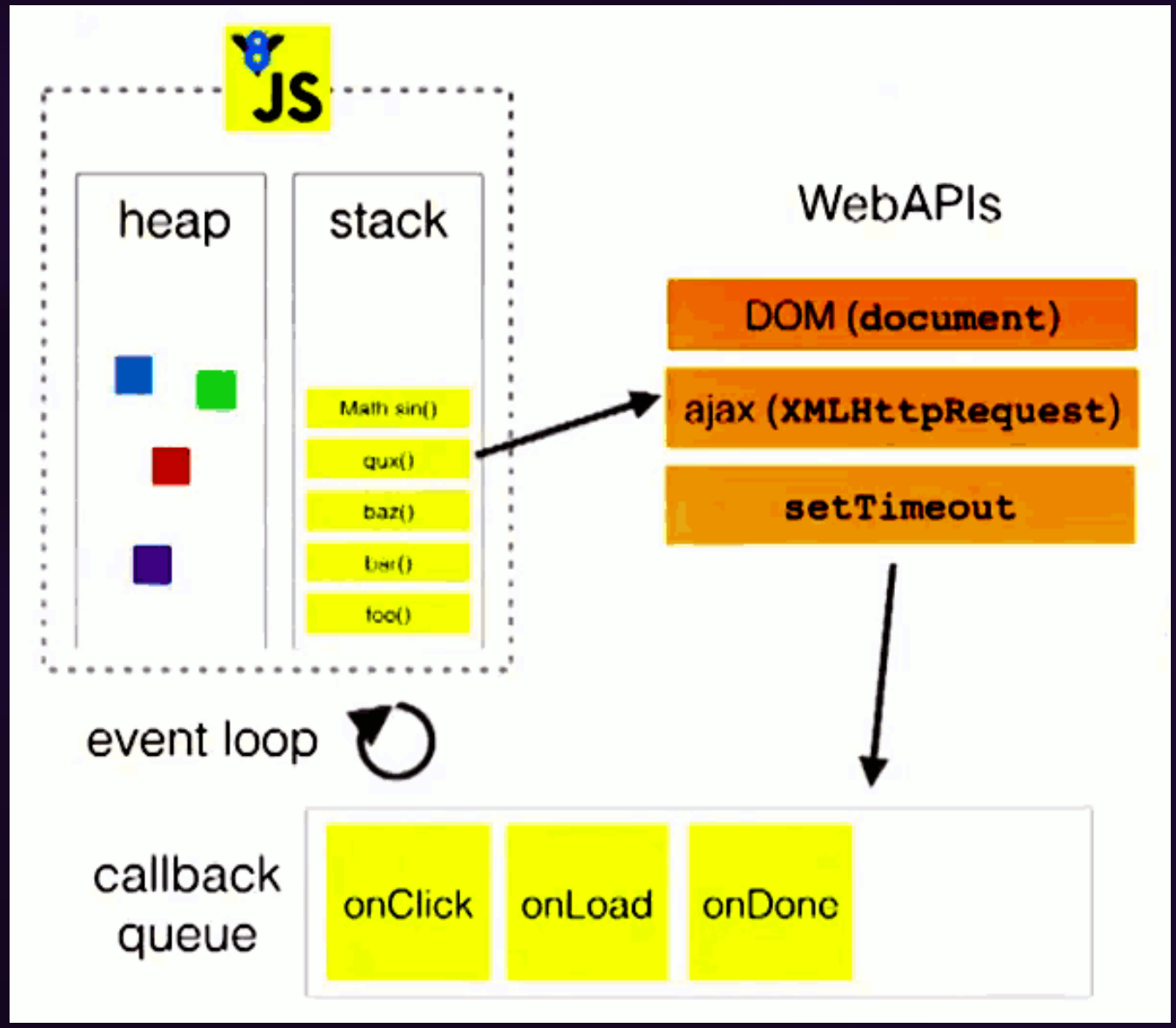
```
1   var a = 1;
2
3   console.log(a);
```

event-loop-02.js > ...

```
1   var a = 1;
2
3   setInterval(() => {
4     console.log(a);
5   }, 1000)
```

IO模型与事件循环——事件循环

- 回顾浏览器中的事件循环
  - js引擎负责哪些?
  - 浏览器负责哪些?



IO模型与事件循环——事件循环

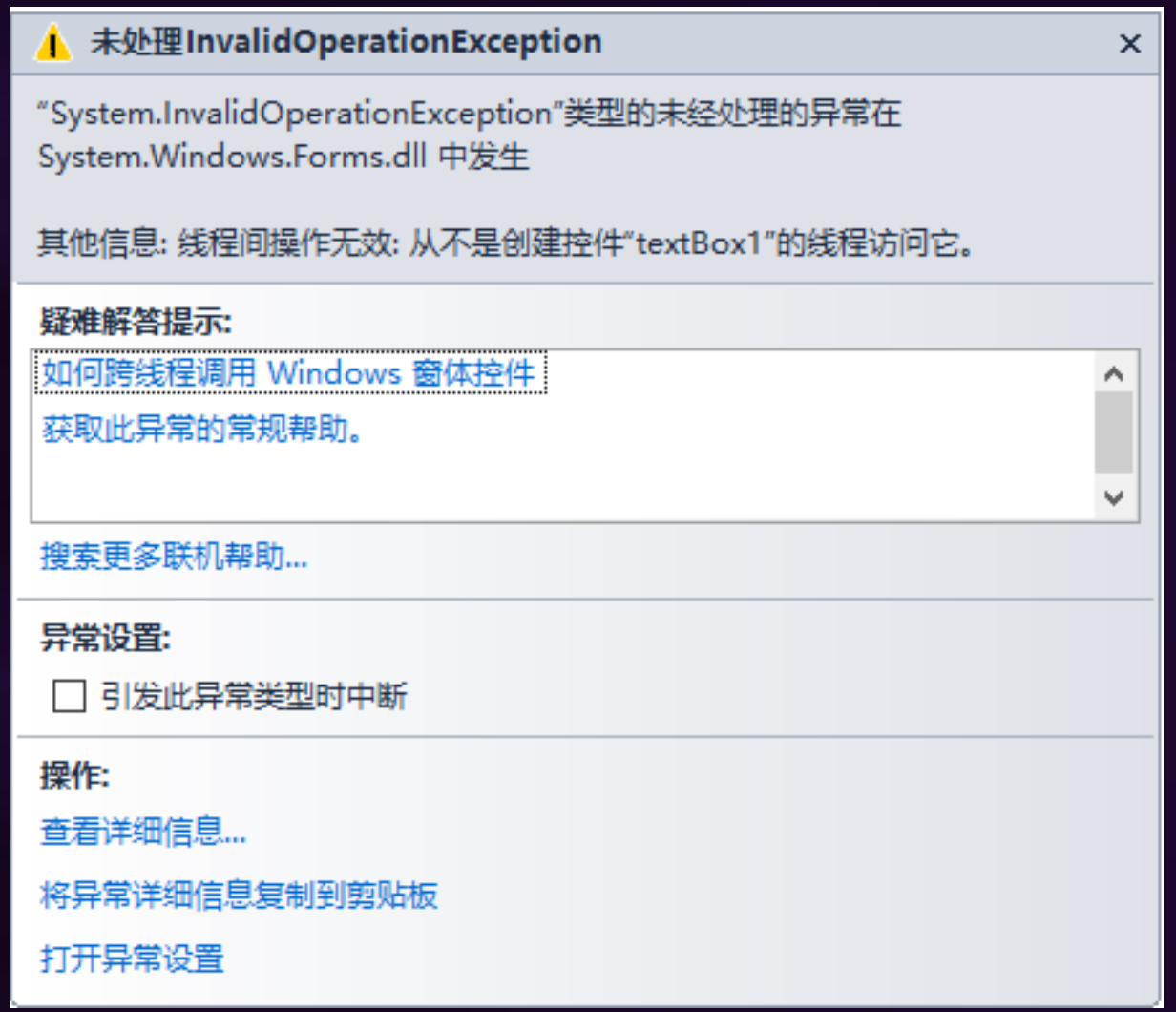
- 大家都很了解前端开发，不妨考虑2个问题：
  - javascript为啥最初被设计成单线程?
  - webworker为啥不允许访问DOM?

> 注意：参照 The Web Workers API landing page 获取workers的参考文档和更多指引。

在worker线程中你可以运行任何你喜欢的代码，不过有一些例外情况。比如：在worker内，不能直接操作DOM节点，也不能使用`window`对象的默认方法和属性。然而你可以使用大量window对象之下的东西，包括WebSockets，IndexedDB以及FireFox OS专用的Data Store API等数据存储机制。查看Functions and classes available to workers获取详情。

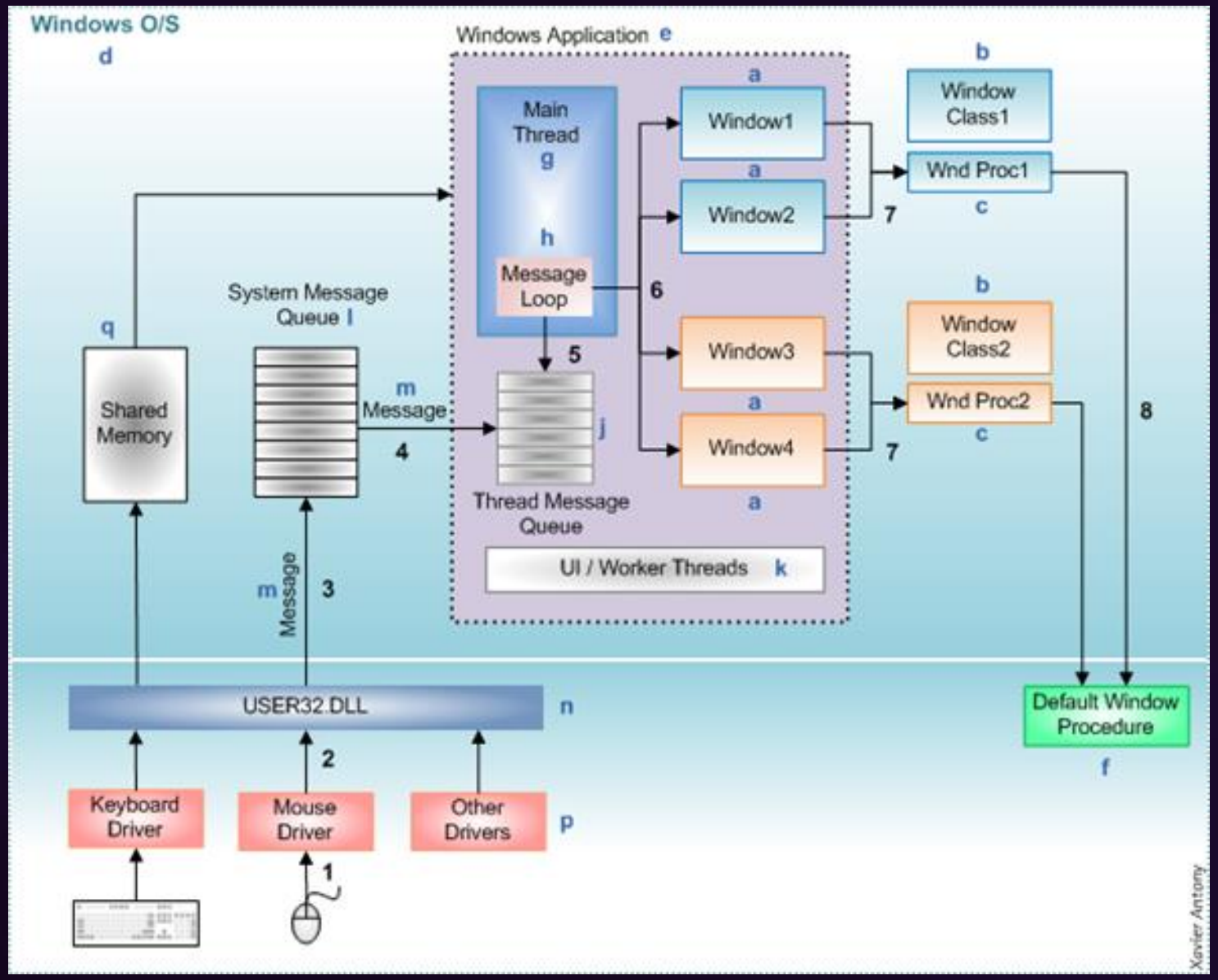
IO模型与事件循环——事件循环

- GUI开发鼻祖-windows
  - 问题总是惊人的相似
  - 那背后的设计是否有共通之处?



IO模型与事件循环——事件循环

- windows的消息循环
  - OS->Application
  - Application->Threads



IO模型与事件循环——事件循环

- 事件循环的设计
  - 数据结构层面
    - 类似消息队列
  - 设计模式层面
    - 典型的生产者-消费者模型
    - 多线程写入
    - 单线程读取

IO模型与事件循环——事件循环

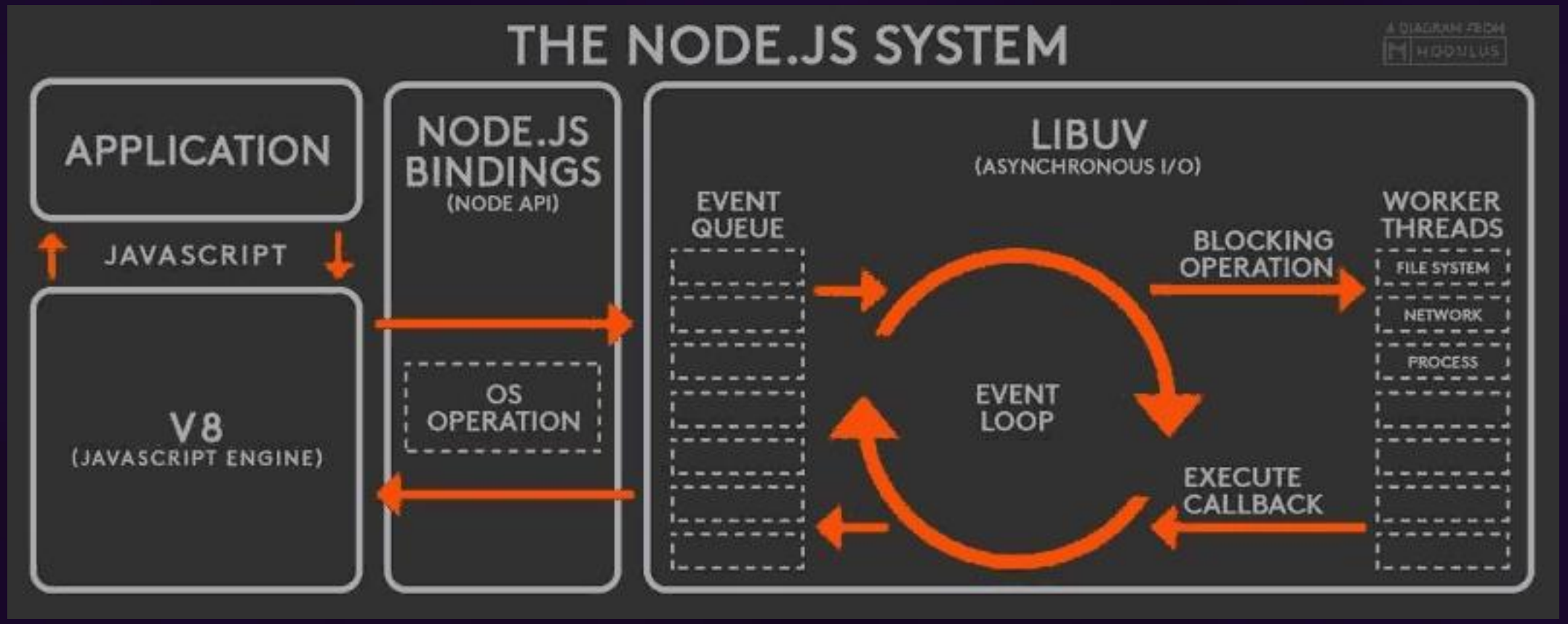
- 为什么GUI的架构设计偏爱单线程+事件循环
  - UI资源的频繁变化与多线程并发控制的矛盾
    - 诞生了主线程/UI线程概念
  - 简化编程范式
  - 降低开发门槛
  - 提高程序健壮性
- 子线程要想更新Ui怎么办
  - Winform:
    - Invoke/BeginInvoke
  - javascript(webworker):
    - postMessage

IO模型与事件循环——事件循环

- node.js的事件循环
  - 无需考虑GUI的问题，因为node.js本质上和GUI开发没啥关系
  - 多了各种文件IO,网络IO这些异步操作
  - 难道全部自己再实现一遍?

IO模型与事件循环——事件循环

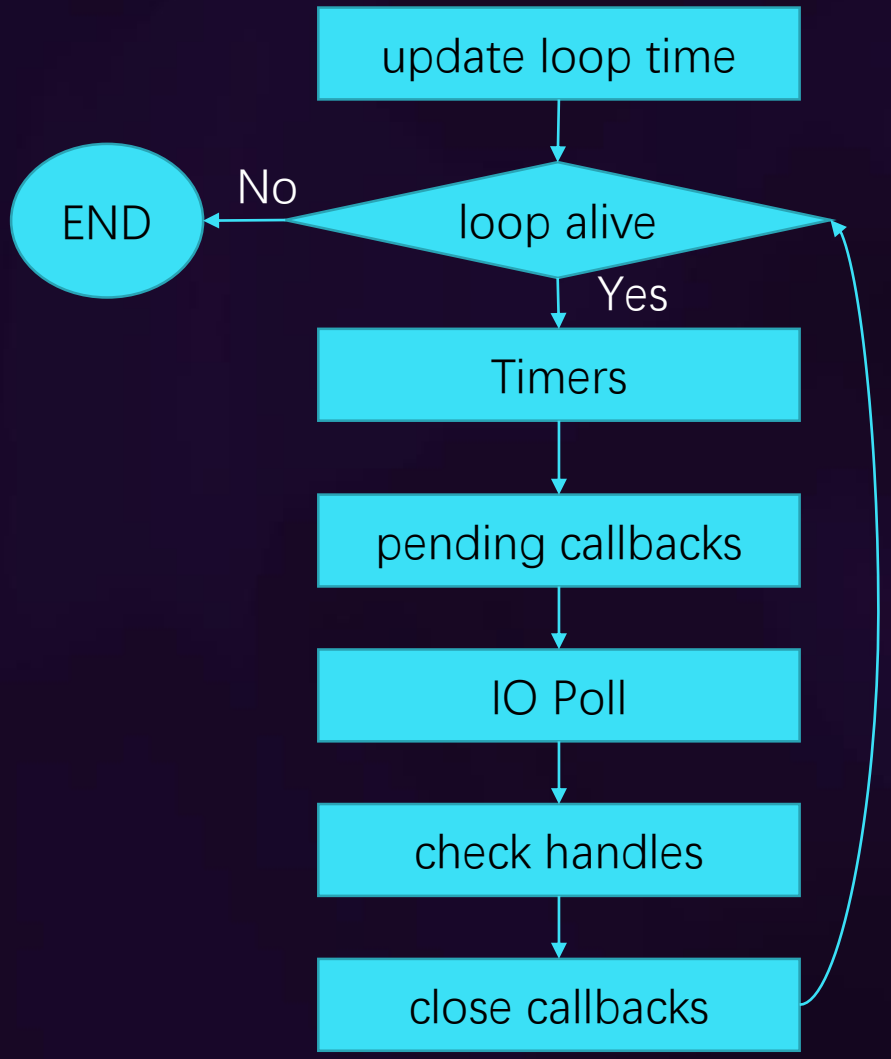
- node.js的事件循环



node直接使用libuv的默认事件循环

IO模型与事件循环——事件循环

- node.js的事件循环细节



deps > uv > src > unix > C core.c > ...

```
347
348  int uv_run(uv_loop_t* loop, uv_run_mode mode) {
349    int timeout;
350    int r;
351    int ran_pending;
352
353    r = uv__loop_alive(loop);
354    if (!r)
355      uv__update_time(loop);
356
357    while (r != 0 && loop->stop_flag == 0) {
358      uv__update_time(loop);
359      uv__run_timers(loop);
360      ran_pending = uv__run_pending(loop);
361      uv__run_idle(loop);
362      uv__run_prepare(loop);
363
364      timeout = 0;
365      if ((mode == UV_RUN_ONCE && !ran_pending) || mode == UV_RUN_DEFAULT)
366        timeout = uv_backend_timeout(loop);
367
368      uv__io_poll(loop, timeout);
369      uv__run_check(loop);
370      uv__run_closing_handles(loop);
371
```

IO模型与事件循环——事件循环

- node.js的事件循环细节

IO模型与事件循环——事件循环

- 思考问题
  - 如此跑事件循环，CPU难道不会100%吗

```
timeout = 0;
if ((mode == UV_RUN_ONCE && !ran_pending) || mode == UV_RUN_DEFAULT)
```

```
int uv_backend_timeout(const uv_
  if (loop->stop_flag != 0)
    return 0;

  if (!uv__has_active_handles(lo
    return 0;

  if (!QUEUE_EMPTY(&loop->idle_h
    return 0;

  if (!QUEUE_EMPTY(&loop->pendin
    return 0;

  if (loop->closing_handles)
    return 0;

  return uv__next_timeout(loop);
}
```

```
int uv__next
  const stru
  const uv_t
  uint64_t d

  heap_node
  if (heap_n
    return -

  handle = c
  if (handle
    return 0

  diff = han
  if (diff > INT_MAX)
    diff = INT_MAX;

  return (int) diff;
}
```

```
for (;;) {
  /* See the comment for max_safe_timeout for an explanation of why
   * this is necessary.  Executive summary: kernel bug workaround.
   */
  if (sizeof(int32_t) == sizeof(long) && timeout >= max_safe_timeout)
    timeout = max_safe_timeout;

  nfds = epoll_pwait(loop->backend_fd,
                     events,
                     ARRAY_SIZE(events),
                     timeout,
                     psigset);
```

IO模型与事件循环——事件循环

- macro-task与micro-task
  - micro-task实在当前调用栈结束之后立刻执行，其实是同步执行的
  - macro-task才是真正的异步执行

```
(function test() {
    setTimeout(function() {console.log(4)}, 0);
    new Promise(function executor(resolve) {
        console.log(1);
        for( var i=0 ; i<10000 ; i++ ) {
            i == 9999 && resolve();
        }
        console.log(2);
    }).then(function() {
        console.log(5);
    });
    console.log(3);
})()
```

```
1
2
3
5
4
```

IO模型与事件循环——事件循环

- node.js所涉及的2类任务

IO模型与事件循环——事件循环

- 关于process.nextTick
  - 其实它不在libuv的event loop里实现
  - 永远在当前调用栈执行结束之后立刻执行（比micro task还要早）

IO模型与事件循环——事件循环

- 小结
  - 各种异步任务的执行顺序

- 所有异步任务类型内部，处理基本原则是：先来先服务
- Macro Task执行时，创建一个新的当前执行栈，这里面的代码运行结束时，会紧接运行nextTick回调，最后是mico taks
- setImmediate从API层面，相当于承担setTimeout(cb,0)的职责(然而调用阶段不同)

IO模型与事件循环——事件循环

- 同样是macro task,他们谁先谁后?

event-loop-11-setImmediate-vs-setTimeout.js > ...

```
1  // timeout_vs_immediate.js
2  setTimeout(function timeout() {
3    console.log('timeout');
4  }, 0);
5
6  setImmediate(function immediate() {
7    console.log('immediate');
8  });
```

```
→  node-core-course node -v
v12.8.1
→  node-core-course node event-loop-11-setImmediate-vs-setTimeout.js
timeout
immediate
→  node-core-course node event-loop-11-setImmediate-vs-setTimeout.js
immediate
timeout
→  node-core-course node event-loop-11-setImmediate-vs-setTimeout.js
immediate
timeout
→  node-core-course node event-loop-11-setImmediate-vs-setTimeout.js
timeout
immediate
```

IO模型与事件循环——事件循环

- 换一个姿势
  - WHY?

event-loop-12-setImmediate-vs-setTimeout-2.js > ...

```
1   // timeout_vs_immediate.js
2   var fs = require('fs')
3
4   fs.readFile(__filename, () => {
5     setTimeout(() => {
6       console.log('timeout')
7     }, 0)
8     setImmediate(() => {
9       console.log('immediate')
10    })
11  })
```

```
→  node-core-course node -v
v12.8.1
→  node-core-course node event-loop-12-setImmediate-vs-setTimeout-2.js
immediate
timeout
→  node-core-course node event-loop-12-setImmediate-vs-setTimeout-2.js
immediate
timeout
→  node-core-course node event-loop-12-setImmediate-vs-setTimeout-2.js
immediate
timeout
→  node-core-course node event-loop-12-setImmediate-vs-setTimeout-2.js
immediate
timeout
→  node-core-course node event-loop-12-setImmediate-vs-setTimeout-2.js
immediate
timeout
→  node-core-course
```

IO模型与事件循环——事件循环

- 原因解析
  - 程序运行的耗时不定
  - 而IO回调函数执行后，最先遇到的是check环节，所以setImmediate先执行

```
while (r != 0 && loop->stop_flag == 0) {
  uv__update_time(loop);
  uv__run_timers(loop);
  ran_pending = uv__run_pending(loop);
  uv__run_idle(loop);
  uv__run_prepare(loop);

  timeout = 0;
  if ((mode == UV_RUN_ONCE && !ran_pending) || mode == UV_RUN_DEFAULT)
    timeout = uv_backend_timeout(loop);

  uv__io_poll(loop, timeout);
  uv__run_check(loop);
  uv__run_closing_handles(loop);
```

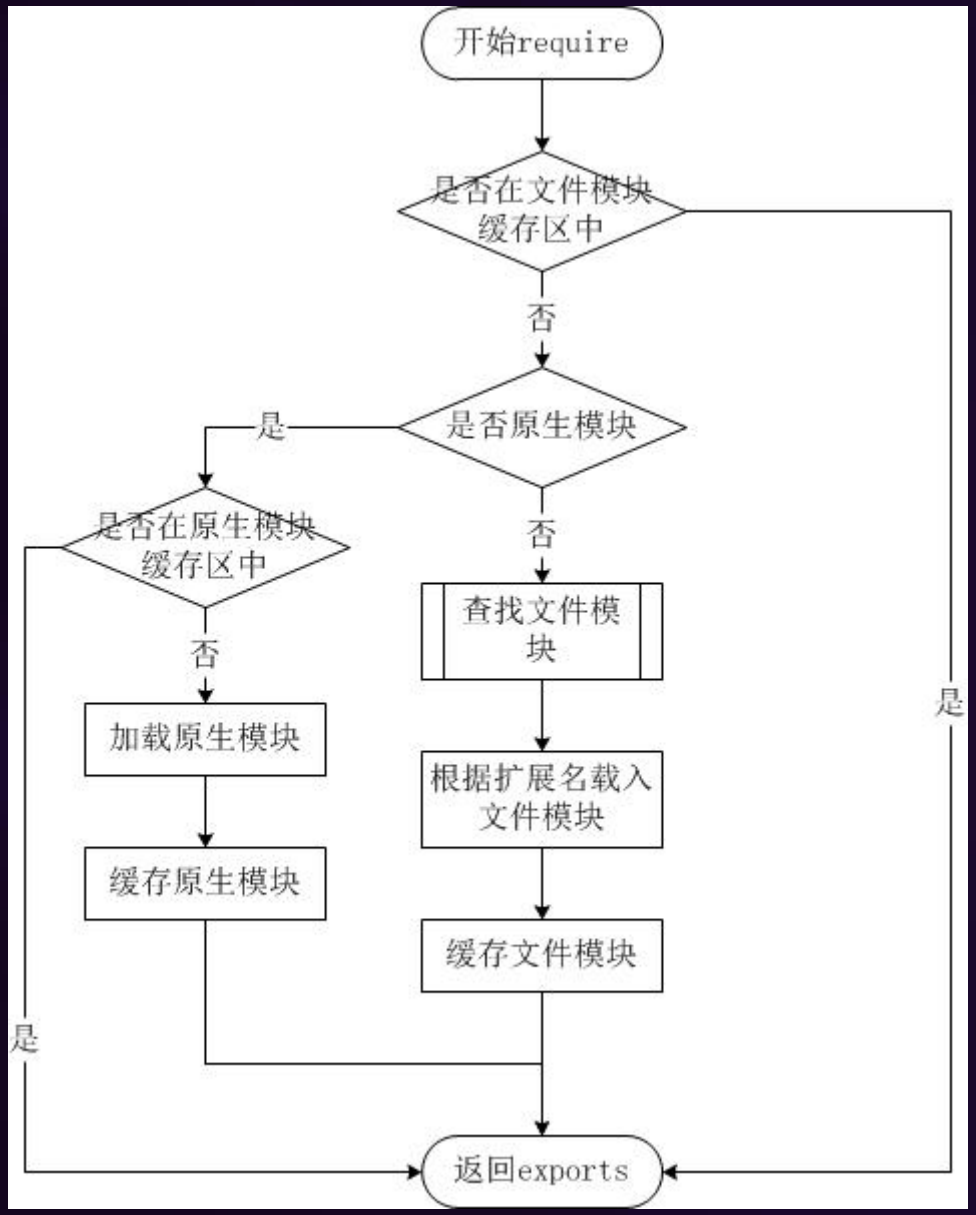
3 node.js模块管理

node.js模块管理

- 基本概念
  - commonjs
    - require
    - exports
    - module
  - 模块分类
    - 核心(原生)模块
      - fs
      - http
      - …
    - 文件模块
      - ‘./xx/xx’
      - ‘../../xx’
      - ‘xx’

node.js模块管理

- 模块的查找策略
  - 原生模块优先
  - 缓存优先



node.js模块管理

- 文件模块的查找方式
  - module.path变量

```
console.log(module.paths);
```

```
[
  '/Users/kaicui/Documents/work/技术分享/2019-集团前端中级培训课程/node核心介绍代码/node-core-course/node_modules',
  '/Users/kaicui/Documents/work/技术分享/2019-集团前端中级培训课程/node核心介绍代码/node_modules',
  '/Users/kaicui/Documents/work/技术分享/2019-集团前端中级培训课程/node_modules',
  '/Users/kaicui/Documents/work/技术分享/node_modules',
  '/Users/kaicui/Documents/work/node_modules',
  '/Users/kaicui/Documents/node_modules',
  '/Users/kaicui/node_modules',
  '/Users/node_modules',
  '/node_modules'
]
```

node.js模块管理

- 文件模块的查找方式
  1. 从 module path 数组中取出第一个目录作为查找基准。
  2. 直接从目录中查找该文件，如果存在，则结束查找。如果不存在，则进行下一条查找。
  3. 尝试添加.js、.json、.node 后缀后查找，如果存在文件，则结束查找。如果不存在，则进行下一条。
  4. 尝试将 require 的参数作为一个包来进行查找，读取目录下的 package.json 文件，取得 main 参数指定的文件。
  5. 尝试查找该文件，如果存在，则结束查找。如果不存在，则进行第 3 条查找。
  6. 如果继续失败，则取出 module path 数组中的下一个目录作为基准查找，循环第 1 至 5 个步骤。
  7. 如果继续失败，循环第 1 至 6 个步骤，直到 module path 中的最后一个值。
  8. 如果仍然失败，则抛出异常。

node.js模块管理

- 模块编写实践经验建议
  - 相对标准的、功能相对复杂的、可复用范围广的通用模块，建议编写为NPM package
    - 使用者通过npm install安装
    - 模块查找首次加载时间相对较长（基本可忽略）
  - 比较简单的、复用范围仅限项目内部的模块，直接项目里建一个lib目录保存即可
    - 使用者直接通过相对路径require
    - 模块首次加载速度最快

node.js模块管理

- 反模式：模块的循环依赖
  - node.js在第一次编译文件模块的时候，就已经纳入了缓存
  - 循环依赖不会导致编译报错，但是会发生一些不符合预期的行为

module-recycle > a.js > ...

```
1  let b = require('./b');
2
3  console.log('A: before logging b');
4  console.log(b);
5  console.log('A: after logging b');
6
7  module.exports = {
8    A: 'this is a Object'
9  };
```

循环依赖

module-recycle > b.js > ...

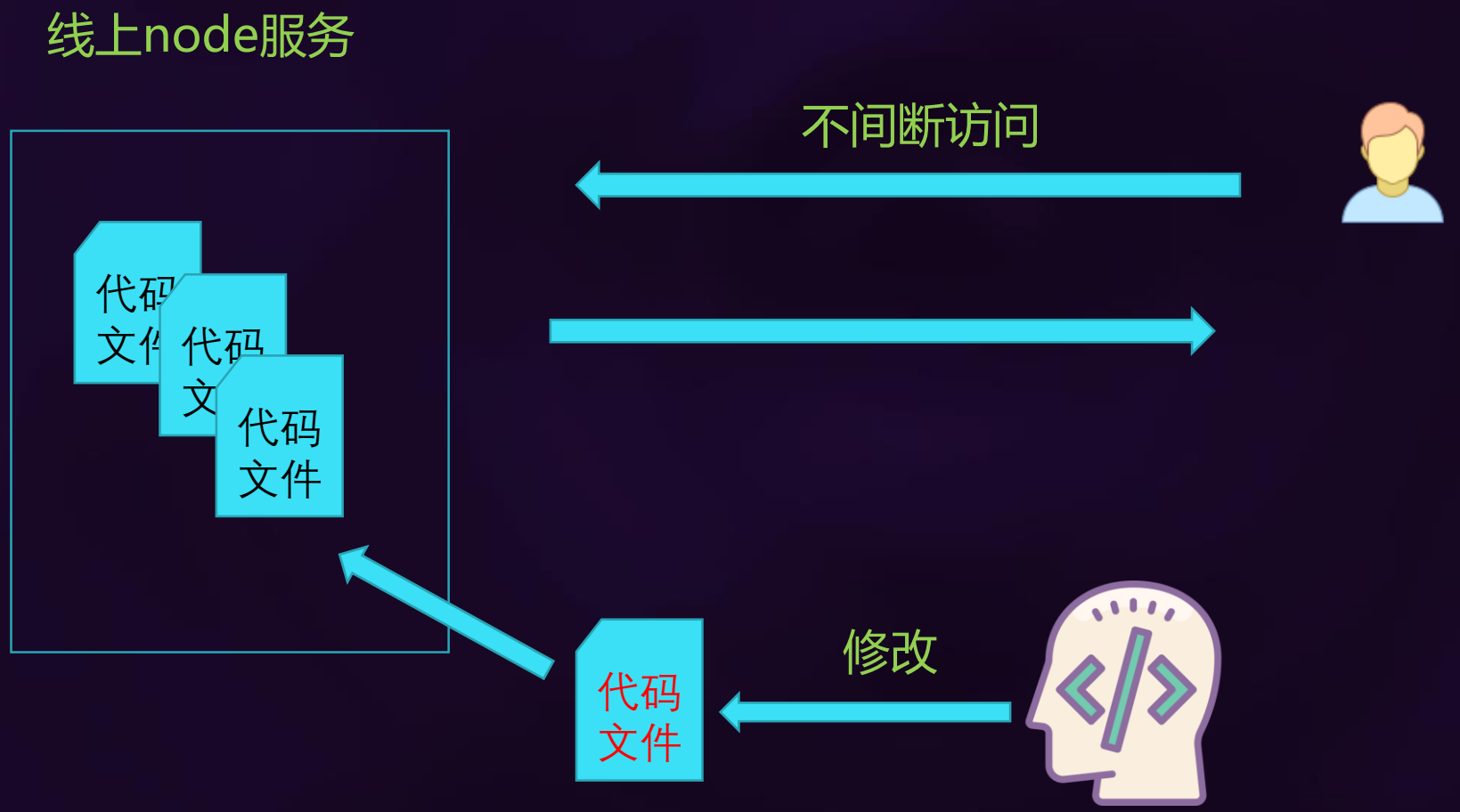
```
1  let a = require('./a');
2
3  console.log('B: before logging a');
4  console.log(a);
5  console.log('B: after logging a');
6
7  module.exports = {
8    B: 'this is b Object'
9  };
```

```
→  module-recycle node a.js
B: before logging a
{}
B: after logging a
A: before logging b
{ B: 'this is b Object' }
A: after logging b
```

b模块中获取的a模块是空对象

node.js模块管理

- 模块管理应用案例实践
  - node编写的服务如何不重启进程来实现热更新?



node.js模块管理-案例实践

- 基本思路
  - 文件拆分，使得需要热更新的模块有单独的js文件保存
  - 使用fs.watch或者其他api监听文件内容变化
  - 从cache中找到要更新的模块，使缓存失效
  - 释放老模块的资源

module-hotreload > index.js > setInterval() callback

```
1   function cleanCache(modulePath) {
2     var module = require.cache[modulePath];
3     // remove reference in module.parent
4     if (module && module.parent) {
5       module.parent.children.splice(module.parent.children.indexOf(module), 1);
6     }
7     require.cache[modulePath] = undefined;
8     console.log('remove cache')
9   }
10
```

找到模块的绝对路径

释放老模块在其他地方的引用

从缓存中删除老模块

node.js模块管理-案例实践

- DEMO演示
  - 还有一些其他情况（比如setInterval)会导致模块资源的释放较为复杂
  - 重要项目的生产环境，还是建议重启进程，或用PM2等第三方库管理

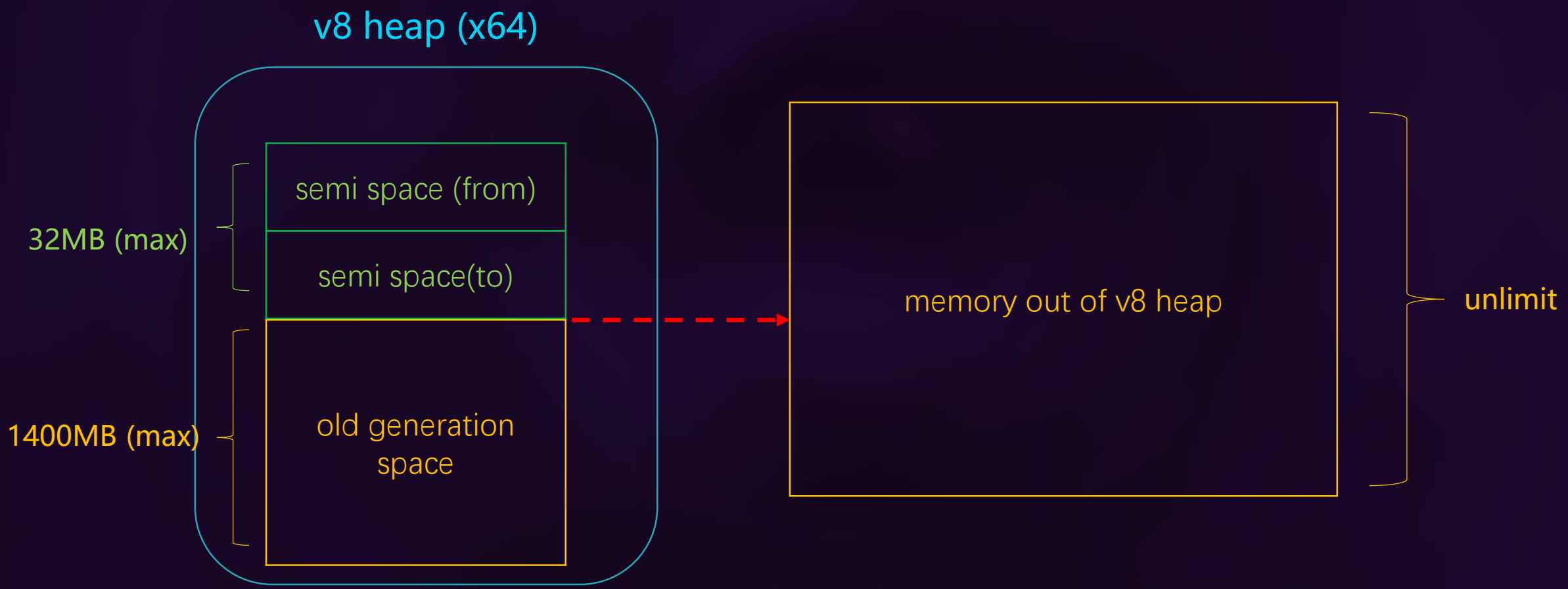
4 node.js的内存管理

node.js的内存管理

- 了解内存管理的必要性
  - node.js开发的程序，很多都是常驻内存提供长时间服务的
  - 网页应用出内存问题，影响的可能是局部重度用户。node.js服务出问题，影响整个业务可用性。
  - 内存泄露是家常便饭，如何查找、定位问题很关键
  - 做架构、方案设计时提前规避重大风险

node.js的内存管理

- 内存管理概览



node.js的内存管理

- 内存相关指标
  - heapTotal
    - v8堆中总共申请的内存
  - heapUsed
    - 已经使用的v8堆内存
  - rss
    - resident set size
    - 进程的常驻内存部分
- 获取进程的内存指标
  - process.memoryUsage

node.js的内存管理

- v8的内存限制
  - 默认情况
    - 1.4GB(64位)
    - 0.7GB(32位)
  - 自定义
    - --max-old-space-size(老生代)
    - --max-new-space-size (新生代)
- Why
  - 使用的内存越大，GC的停顿越大
- 内存真的不够怎么办
  - 多进程架构
  - 自定义启动参数

node.js的内存管理

- v8的垃圾回收机制
  - 新生代：Scavenge算法
    - 空间一分为2，from,to
    - 先在from中分配内存，GC时把存活对象copy到to,然后清空from
      - 如果被复制对象已经经历过回收，或to空间占用率超过25%，则晋升到old generation
    - from 和 to 角色互换
    - 如此往复
  - 优点
    - 回收速度快
  - 缺点
    - 内存使用率低（只能使用1/2的内存）
  - Why
    - 新生代对象存活周期较短

node.js的内存管理

- v8的垃圾回收机制
  - 老生代： Mark-Sweep
    - 遍历所有对象，标记存活的对象
    - 清理死亡的对象
  - 老生代： Mark-Compact
    - 在Mark-Sweep基础上，增加对象的移动
    - 目的是腾出更多的连续内存空间
  - 优点
    - 内存可以充分利用
  - 缺点
    - 一次清理耗时较长
  - Why
    - 新生代对象存活周期较长，存活对象占大多数

node.js的内存管理

- 内存泄露
  - 什么是内存泄露
    - 不再使用的对象，并没有被GC回收掉
    - 泄露的对象，是本来该“死掉”，但仍然“存活”的对象
  - 哪些场景容易导致泄露
    - 全局对象使用
    - 闭包
    - 长连接应用

node.js的内存管理

- 内存泄露典型案例场景
  - 一个基于长连接的实时服务
  - 用户可以通过网页连接，加入房间
  - 需要实现各种形式的消息组播
    - 用户群消息
    - 房间消息
    - ...
  - 问题演示DEMO

```
io.on('connection', function (socket) {
  console.log('a user connected');
  sendToRoom('room1', `user:${socket.id} comming!`)

  allUsers.room1.push(socket);
```

```
async function main() {
  for (let i = 0; i < 10000 * 10000; i++) {
    makeNewConn();
    await sleep(20);
  }
}

main();
```

模拟创建用户不停进入、退出的场景

```
Process: heapTotal 16.41 MB heapUsed 8.03 MB rss 35.66 MB
-----------------------------------------------------------
Process: heapTotal 15.12 MB heapUsed 10.95 MB rss 40.22 MB
-----------------------------------------------------------
Process: heapTotal 22.12 MB heapUsed 15.39 MB rss 48.16 MB
-----------------------------------------------------------
Process: heapTotal 31.87 MB heapUsed 22.26 MB rss 58.53 MB
-----------------------------------------------------------
Process: heapTotal 54.37 MB heapUsed 22.17 MB rss 63.18 MB
-----------------------------------------------------------
```

node.js的内存管理

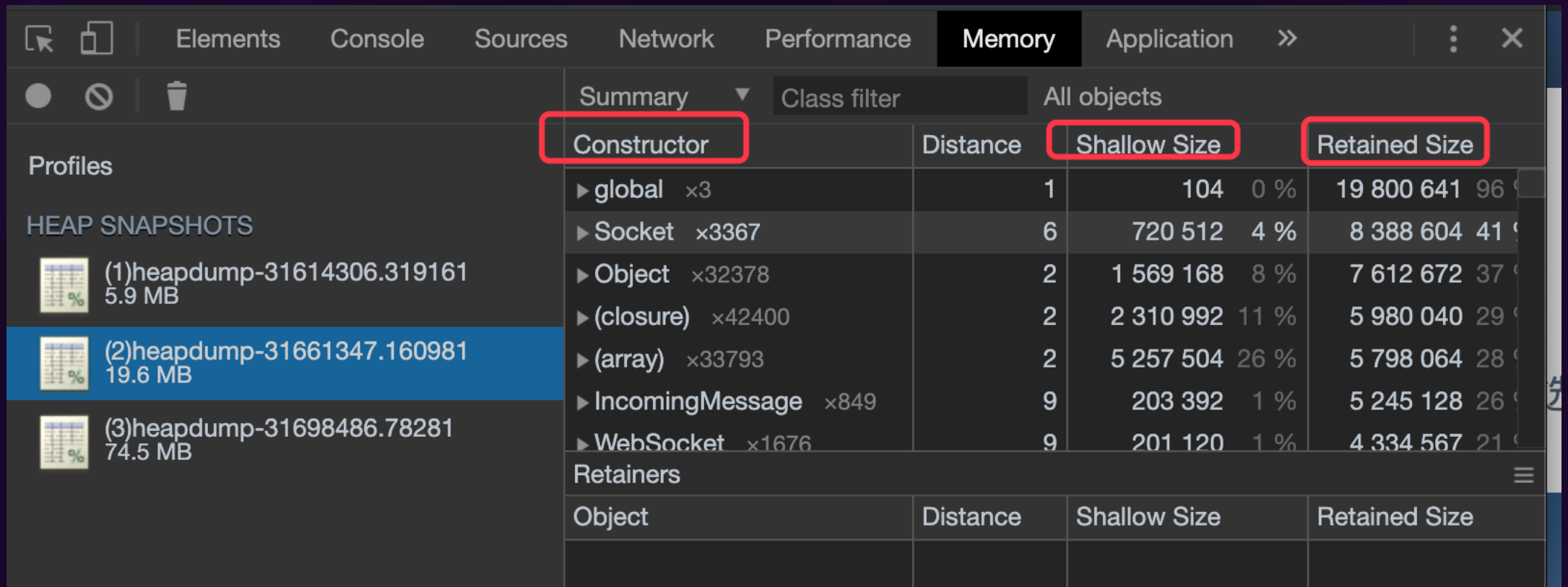
- 内存泄露的排查
  - 思路
    - 定位泄露对象的类型
    - 通过不同泄露前后的内存快照，查找泄露源头
    - 对代码的熟悉也很重要，很多问题看代码可以提前规避泄露风险
  - 工具
    - node-heapdump

node.js的内存管理

- 内存泄露的排查
  - 在服务代码中，install heapdump
  - 在服务运行开始，以及泄露发生后，分别使用kill -USR2 <PID> 的方式，通知服务打内存dump
  - 在chrome的memory面板中分析内存文件，找出问题

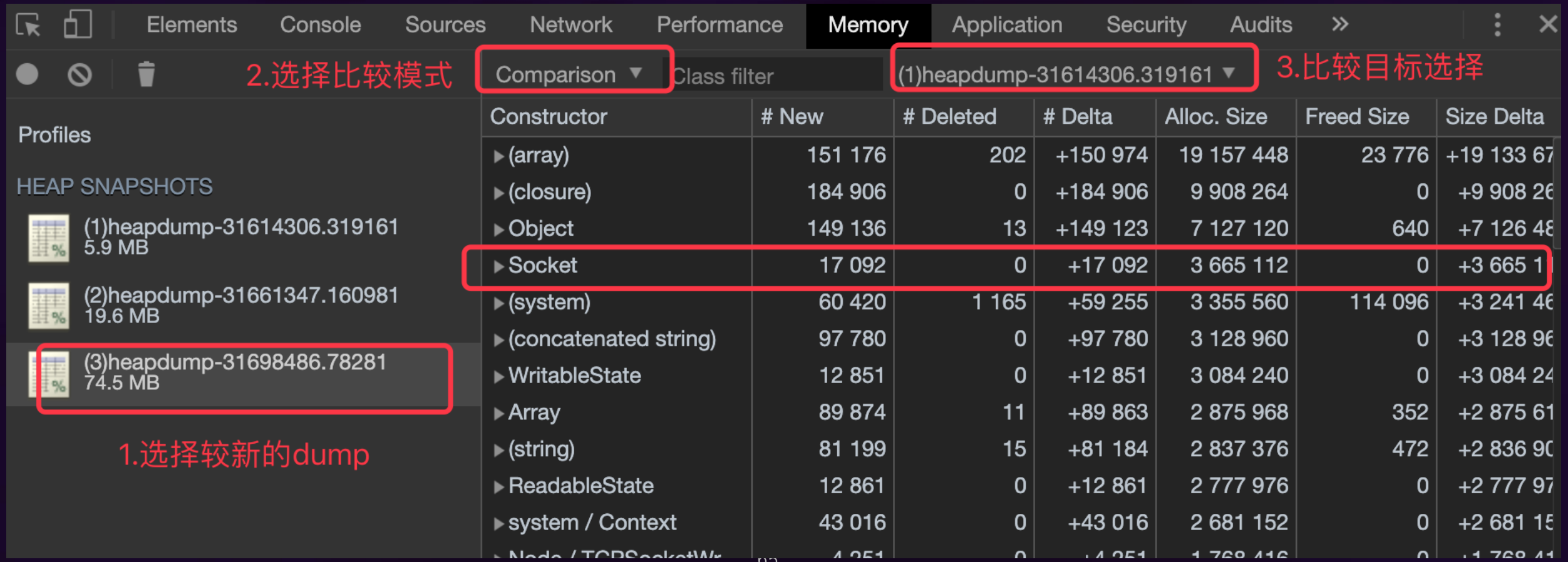
node.js的内存管理

- 内存泄露的排查——chrome的memory profiles
  - Shallow Size
    - 对象自身占用的大小
  - Retained Size
    - 对象以及自身引用的其他对象在一起的大小



node.js的内存管理

- 内存泄露的排查——chrome的memory profiles
  - comparison视图
    - 比较泄露后的文件和泄露前的差异



node.js的内存管理

- 一些最佳实践
  - 减少匿名函数的使用
  - 需要频繁创建的对象，最好使用OOP方式创建
  - 注意事件监听的闭包里所使用的外部对象

node.js的内存管理

- Buffer与内存管理
  - Buffer特点
    - Buffer的分配不在v8里控制,但是也会 统计到rss里
    - Buffer可以不受v8的内存限制
    - v8虽然不分配Buffer的内存，但是Buffer对象本身是在v8里管理
      - 所以v8的GC也会顺带释放Buffer占据的空间
  - Buffer的Demo

Buffer泄露

```
Process: heapTotal 4.84 MB heapUsed 2.88 MB rss 86.13 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.09 MB rss 105.93 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.09 MB rss 125.93 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.09 MB rss 145.93 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.10 MB rss 165.93 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.05 MB rss 185.14 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.05 MB rss 205.14 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.06 MB rss 225.15 MB
-------------------------------------------------------
```

Buffer回收

```
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.09 MB rss 105.83 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.10 MB rss 105.83 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.10 MB rss 105.83 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.05 MB rss 124.95 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.05 MB rss 124.96 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.06 MB rss 124.96 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.06 MB rss 124.96 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.05 MB rss 124.99 MB
-------------------------------------------------------
Process: heapTotal 4.34 MB heapUsed 2.05 MB rss 124.99 MB
```

node.js的内存管理

- Buffer与ArrayBuffer
  - 区别
    - Buffer的内存分配在v8 heap之外
    - ArrayBuffer的内存是v8分配的，在v8 heap上
    - 部分api的实现不同，比如slice方法
  - 相似
    - 都是对uint8Array的实现
    - 都适合处理2进制数据

ArrayBuffer的内存分配

```
------------------------------------------------------------
Process: heapTotal 45.55 MB heapUsed 11.76 MB rss 34.91 MB
------------------------------------------------------------
Process: heapTotal 66.56 MB heapUsed 32.77 MB rss 56.10 MB
------------------------------------------------------------
Process: heapTotal 85.12 MB heapUsed 51.32 MB rss 74.68 MB
------------------------------------------------------------
Process: heapTotal 75.16 MB heapUsed 41.59 MB rss 64.94 MB
------------------------------------------------------------
Process: heapTotal 96.17 MB heapUsed 62.60 MB rss 86.33 MB
------------------------------------------------------------
```

5 进程管理与IPC

进程管理与IPC

- 为什么要了解进程管理

可靠性

利用多核

进程互操作

集群管理

进程管理与IPC

- process对象的属性/方法
  - argv、env、pid
  - stdin/stdout
  - exit/kill/abort

进程管理与IPC

- 创建子进程
  - spaw
  - exec
  - execFile
  - fork

函数名	结果回调	进程类型	参数类型	支持超时
spaw	✕	任意	命令	✕
exec	✓	任意	命令	✓
execFile	✓	任意	可执行的文件	✓
fork	✕	Node	js文件	✕

进程管理与IPC

- node.js里主子进程关系
  - 当使用ctl+c关闭主进程时，子进程也会自动退出

process > master-fork.js > ...

```
1  var cp = require('child_process');
2  var cpus = require('os').cpus();
3  for (var i = 0; i < cpus.length; i++) {
4    let subprocess = cp.fork('./worker.js');
5  }
6  setInterval(() => {
7    console.log(`master:${process.pid} running~`)
8  }, 1000)
```

```
6888  6597 node master-fork.js
6889  6888 /Users/kaicui/.nvm/versions/node/v12.8.1/bin/node ./worker.js
6890  6888 /Users/kaicui/.nvm/versions/node/v12.8.1/bin/node ./worker.js
6891  6888 /Users/kaicui/.nvm/versions/node/v12.8.1/bin/node ./worker.js
6892  6888 /Users/kaicui/.nvm/versions/node/v12.8.1/bin/node ./worker.js
6893  6888 /Users/kaicui/.nvm/versions/node/v12.8.1/bin/node ./worker.js
6894  6888 /Users/kaicui/.nvm/versions/node/v12.8.1/bin/node ./worker.js
6895  6888 /Users/kaicui/.nvm/versions/node/v12.8.1/bin/node ./worker.js
6896  6888 /Users/kaicui/.nvm/versions/node/v12.8.1/bin/node ./worker.js
~ ps -o pid,ppid,command
```

进程管理与IPC

- 进程间IPC
  - pipe,named pipe
  - socket
  - semaphore
  - shared memory
  - mq
  - domain socket

进程管理与IPC

- Domain Socket
  - 可靠传输
  - 性能比socket要好得多，没有网络协议栈处理
- 用法
  - master
    - childProcess.send
    - childProcess.on（‘message’）
  - worker
    - process.send
    - process.on（‘message’）

进程管理与IPC

- 在主子进程间传递数据消息
  - 注意
    - 无法直接发送字符串
    - 发送和接收过程涉及对象的序列化和反序列化，对象本身无法直接传输

process > worker.js > ...

```
12  setInterval(() => {
13    console.log(`worker:${process.pid} running~`);
14    process.send(
15      'hello world'
16    );
17    process.send({
18      hello: 'world'
19    });
20  }, 3000)
```

发送字符串

发送js对象

```
master received message from worker:3799:
{ hello: 'world' }
```

进程管理与IPC

- 在主子进程间传递句柄(handle)
  - net.Socket
    - TCP套接字
  - net.Server
    - TCP服务器
  - net.Native
  - dgram.Socket
  - dgram.Native

```
const subprocess = require('child_process').fork('subprocess.js');

// Open up the server object and send the handle.
const server = require('net').createServer();
server.on('connection', (socket) => {
  socket.end('handled by parent');
});
server.listen(1337, () => {
  subprocess.send('server', server);
});
```

发送TCP Server到子进程

```
process.on('message', (m, server) => {
  if (m === 'server') {
    server.on('connection', (socket) => {
      socket.end('handled by child');
    });
  }
});
```

子进程处理TCP链接

进程管理与IPC

- 传递句柄的内部实现
  - 发送端判断发送的handle类型
  - 组装成特殊的内部消息结构

```
// Package messages with a handle object
if (handle) {
  // This message will be handled by an internalMessage event handl
  message = {
    cmd: 'NODE_HANDLE',    传递句柄
    type: null,
    msg: message
  };

  if (handle instanceof net.Socket) {
    message.type = 'net.Socket';
  } else if (handle instanceof net.Server) {
    message.type = 'net.Server';
  } else if (handle instanceof TCP || handle instanceof Pipe) {
    message.type = 'net.Native';
  } else if (handle instanceof dgram.Socket) {
    message.type = 'dgram.Socket';
  } else if (handle instanceof UDP) {
    message.type = 'dgram.Native';
  } else {
    throw new ERR_INVALID_HANDLE_TYPE();
  }
```

进程管理与IPC

- 传递句柄的内部实现
  - 接收端收到发送过来的句柄信息
  - 重新调用net.Server的listen方法建立监听

```
// and back again.
const handleConversion = {
  'net.Native': {…
  },

  'net.Server': {
    simultaneousAccepts: true,

    send(message, server, options) {
      return server._handle;
    },

    got(message, handle, emit) {
      const server = new net.Server();
      server.listen(handle, () => {
        emit(server);
      });
    }
```

进程管理与IPC

- 思考问题
  - 为何进程间IPC要特殊支持handle的传递?

进程管理与IPC

- node多进程利用多核
  - 多进程监听同一端口
- 问题
  - 多个进程无法监听同一端口

```
→  error node master.js
events.js:180
      throw er; // Unhandled 'error' event
      ^

Error: listen EADDRINUSE: address already in use :::3000
    at Server.setupListenHandle [as _listen2] (net.js:1228:14)
    at listenInCluster (net.js:1276:12)
    at Server.listen (net.js:1364:7)
```

```
const subprocess1 = require('child_process').fork('subprocess.js');
const subprocess2 = require('child_process').fork('subprocess.js');

setInterval(() => {
  console.log('master is running')
}, 1000)
```

master启动2个子进程

master.js

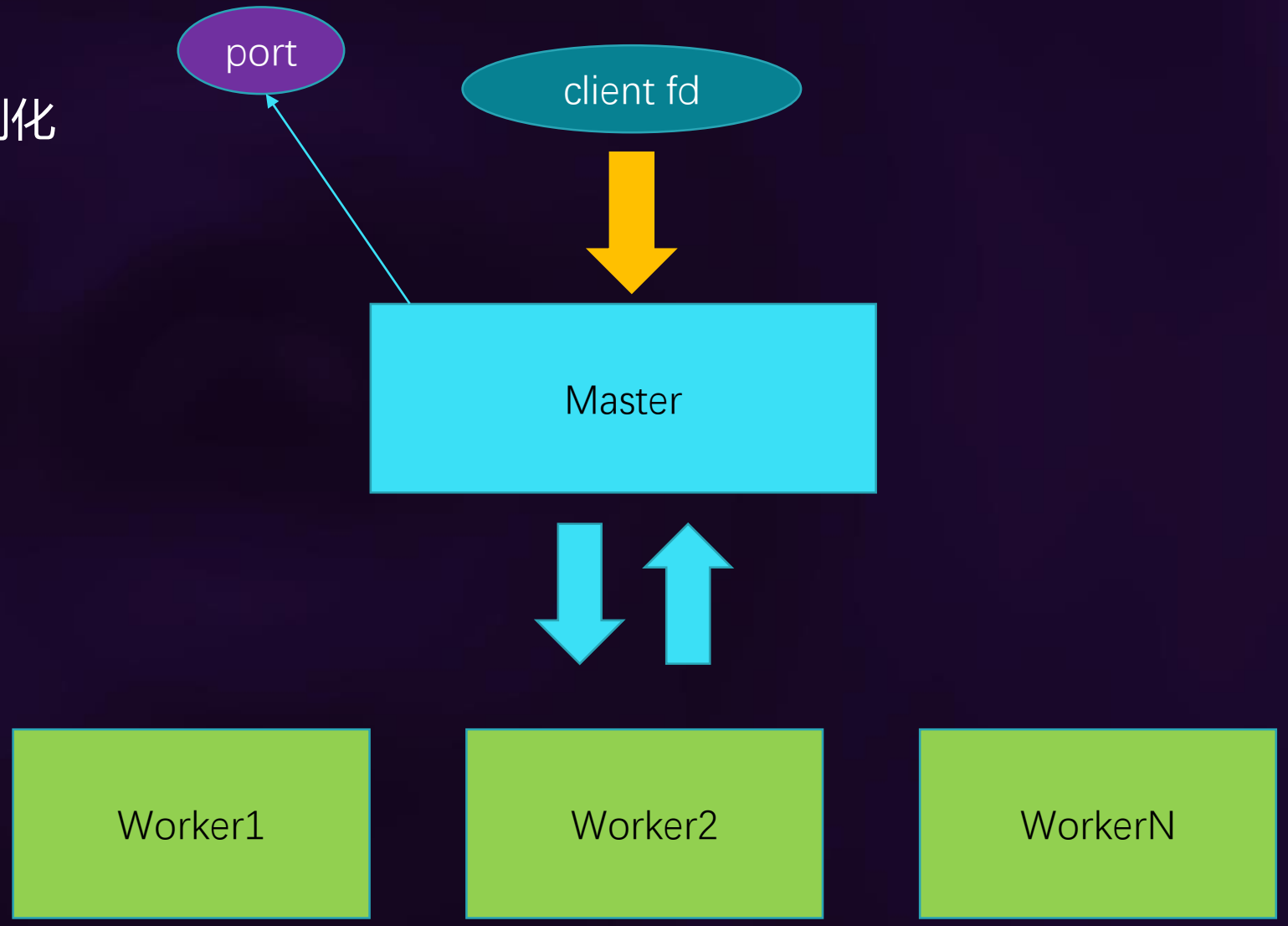
```
const server = require('net').createServer();
server.on('connection', (socket) => {
  socket.end('handled by parent');
});
server.listen(3000, () => {
  console.log(`sub:${process.pid} is listening`)
});
```

子进程尝试监听端口，提供服务

worker.js

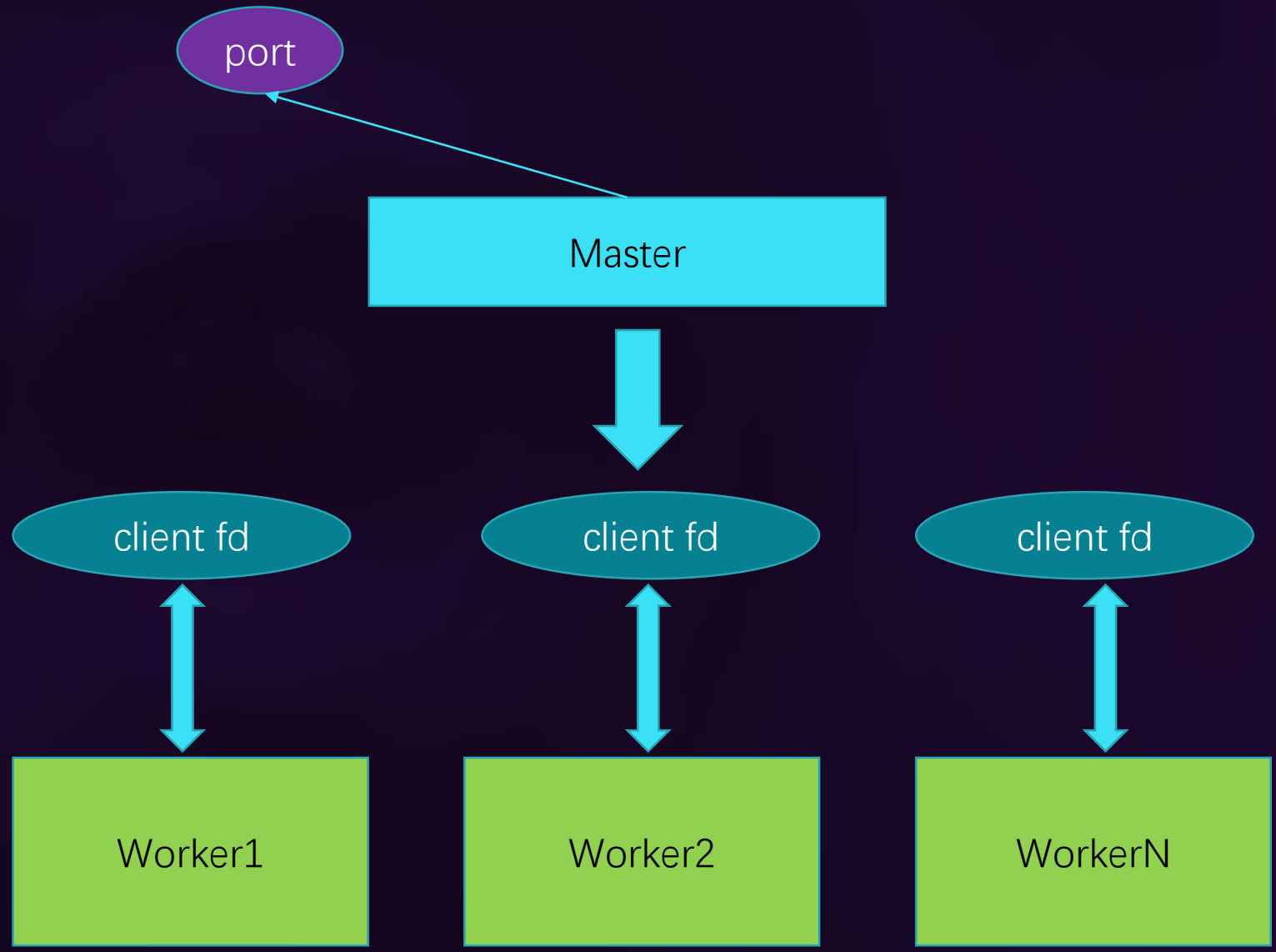
进程管理与IPC

- master代理模式
  - 多了2次消息的序列化、反序列化
  - Master实际上成为了单点瓶颈



进程管理与IPC

- Cluster模式
  - 利用node.js的进程通信可以传递handle的特点
  - 由Master进程监听端口，并把监听到的socket的fd传递给子进程
  - Master进程还实现了一些负载均衡策略，比如RoundRobbin
- 优点
  - 充分利用多核
  - 子进程直接通过socket与client通信
  - 负载均衡实现灵活



进程管理与IPC

- Cluster模式实现细节

```
RoundRobinHandle.prototype.distribute = function(err, handle) {
  this.handles.push(handle);
  const worker = this.free.shift();

  if (worker)
    this.handoff(worker);
};

RoundRobinHandle.prototype.handoff = function(worker) {
  if (this.all.has(worker.id) === false) {
    return;  // Worker is closing (or has closed) the server.
  }

  const handle = this.handles.shift();

  if (handle === undefined) {
    this.free.push(worker);  // Add to ready queue again.
    return;
  }

  const message = { act: 'newconn', key: this.key };

  sendHelper(worker.process, message, handle, (reply) => {
    if (reply.accepted)
      handle.close();
    else
      this.distribute(0, handle);  // Worker is shutting down. Send to

    this.handoff(worker);
  });
```

从队列中取一个

再把这个worker塞回队列尾部

给这个worker进程发送client的socket的fd

```
function listenInCluster(server, address, port, addressType,
                         backlog, fd, exclusive, flags) {
  exclusive = !!exclusive;

  if (cluster === undefined) cluster = require('cluster');

  if (cluster.isMaster || exclusive) {
    // Will create a new handle
    // _listen2 sets up the listened handle, it is still named like this
    // to avoid breaking code that wraps this method
    server._listen2(address, port, addressType, backlog, fd, flags);
    return;
  }

  const serverQuery = {
    address: address,
    port: port,
    addressType: addressType,
    fd: fd,
    flags,
  };

  // Get the master's server handle, and listen on it
  cluster._getServer(server, serverQuery, listenOnMasterHandle);
```

master进程直接监听

child进程实际上是获取master进程监听到的socket

进程管理与IPC

- 问题
  - 为什么不直接让子进程自己监听端口，获取连接?

消息负载不可控

资源竞争效率低下

6 node.js的适用场景

node.js的适用场景

- node.js的特点
  - 庞大的jser用户群，几乎无缝过渡的编程体验
  - v8的性能保证
  - web前端开发工程化工具链的事实标准
  - 繁荣的生态
    - GUI开发
      - nw.js->electron.js
    - Web服务
      - express.js->Koa->Thinkjs,egg.js,...
    - 实时通讯应用
      - socket.io
    - Headless Browser & Test Automation
      - Puppeteer,nightmare
    - 测试
      - karma,Mocha

node.js的适用场景

- 适合
  - 业务中台
  - 业务管理系统
  - 业务服务的中间层
  - 爬虫、自动化测试工具
  - 交互式命令行工具
  - ...
- 不适合
  - cpu密集型服务
  - 需要长时间常驻内存，对稳定性要求非常高的核心服务

node.js的适用场景

- 其他node.js值得关注的话题
  - Stream的使用
  - C++ Addon的编写和使用场景
  - Buffer的使用以及与TypedArray的结合
  - 各种有意思的库和框架

作业

- 编写一个简易的http接口mock服务
  - 具备以下功能：
    - 监听本地8080端口
    - 只要收到请求（不管什么url）,就读取本项目中/data/user.json的内容，作为http响应返回
    - 在本地创建一个日志文件，每次收到请求就把请求的url和当前时间记录到日志文件里
      - 随着请求次数增加，日志文件里的内容也增加(而不是出现新的文件)
- 作业要求
  - 提供源码以及运行启动的.sh脚本（windows请提供.bat脚本）
  - 请自行编译和测试通过可运行
  - 不可以使用任何第三方库，只能使用node.js内置的库
  - 服务可以稳定运行，不崩溃

作业2（扩展练习）

- 编写一个简易的http server
  - 具备以下功能：
    - 可以利用多核心，支持多进程模式（只能使用一个端口）
    - 支持拦截器，即可以通过插入新脚本的方式，在请求处理流程中增加一个切片逻辑，且拦截器可以选择不把执行权交给下个拦截器，而直接返回
    - 每隔1小时自动给每个进程打一个内存dump文件，存放在本地的一个文件夹里
    - 无论请求什么内容，响应都是一个字符串，里面包含请求的header和body内容（body可以一律用utf-8展示）
  - 不需要具备的功能：
    - 无需处理请求body的格式
    - 不需要支持response的其他类型（如静态文件访问），只支持string返回即可
- 作业要求
  - 提供源码以及运行启动的.sh脚本（windows请提供.bat脚本）
  - 请自行编译和测试通过可运行
  - 不可以使用任何第三方库，只能使用node.js内置的库

THANKS

前端赋能 共创卓越