QCon 全球软件开发大会
INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE

BEIJING 2017

探究 Node.js 的服务端之路

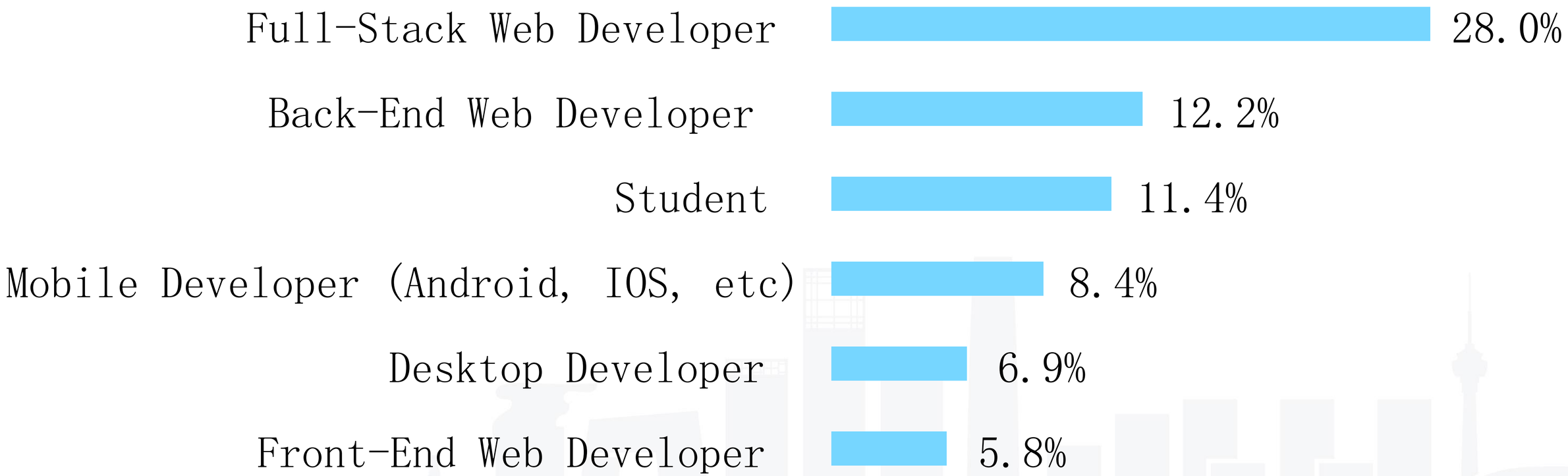
ElemeFe - 黄鼎恒

主办方 Geekbang. 极客邦科技 InfoQ

大纲

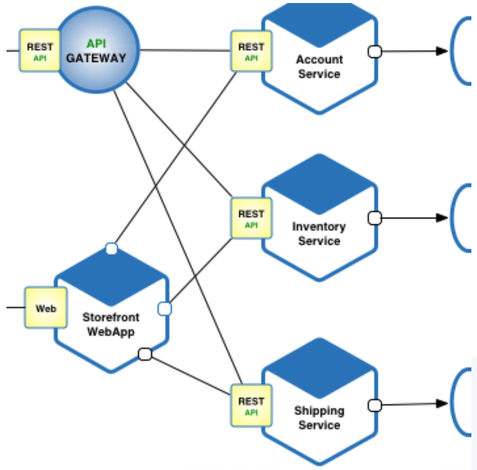
1. 前端 -> Node.js -> 后端
2. V8 内存的简介
3. Node.js 服务端开发的常见问题

2016 Developer Occupations

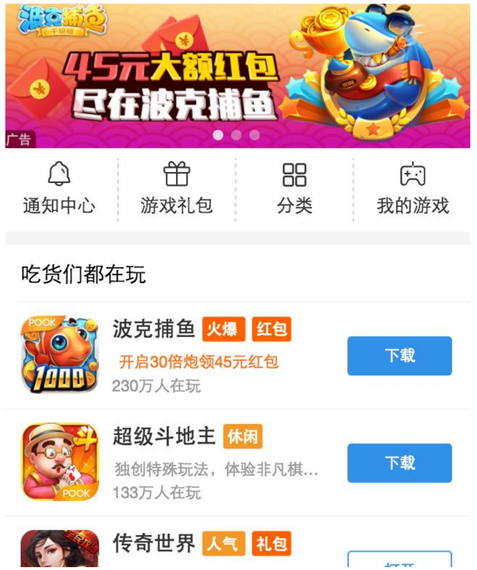


Node.js in Eleme

实时推送



中间件



快速建站

Front to Back

- 业务差异
- 环境差异
- 思想差异

状态管理

展示

面向DB

样式

推送

需求频繁变动

面向DOM操作

通信

业务差异

效果

面向View

交互

数据一致性

数据量大多对多

数据量小一对一

基于操作系统提供的接口

webview 兼容

Polyfill

基于浏览器提供的接口/对象

灾难备份

环境差异

浏览器兼容

响应式

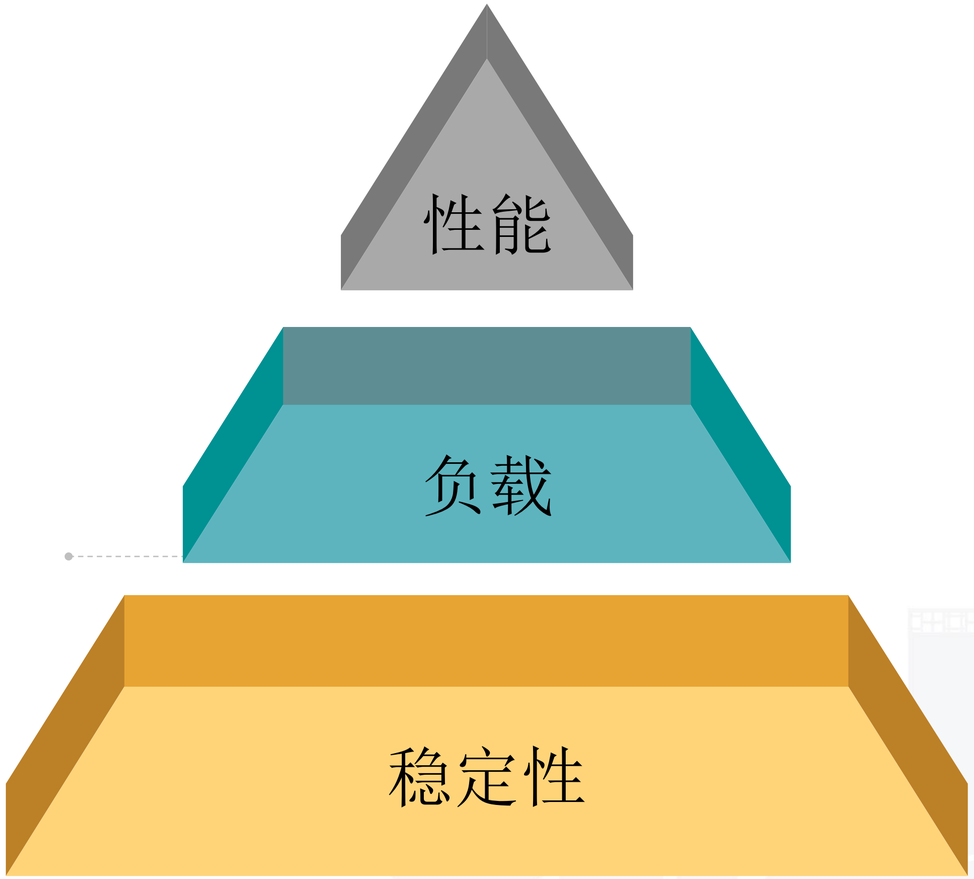
分布式

Scalable

Frontend	Backend
快速开发	稳定
快速渲染	性能
视觉效果	负载

思想差异

Backend’s Perspective



—— n% 流量丢失

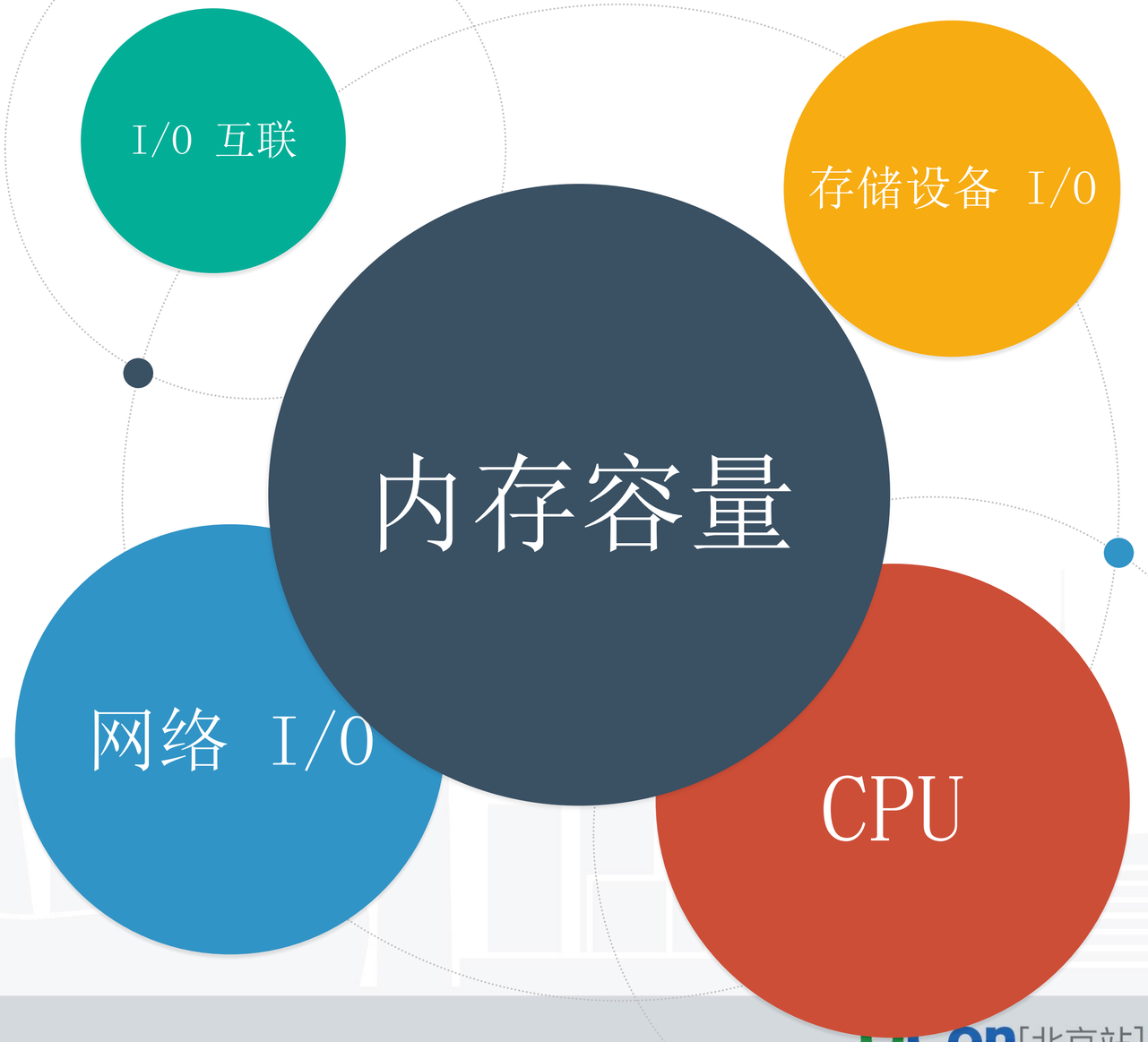
—— m% 流量丢失

—— 100% 流量丢失

Backend's Perspective

稳定性因素

- CPU
- 内存容量
- 网络 I/O
- 存储设备 I/O
- 存储容量
- 存储控制器
- 网络控制器
- CPU 互联
- 内存互联
- I/O 互联



V8 内存结构

Node.js - 后端

V8 JavaScript Engine

Chrome 浏览器 - 前端

```c
#include <stdio.h>

void init()
{
    int arr[5] = {};
    for (int i = 0; i < 5; ++i)
    {
        arr[i] = i;
    }
}

void test()
{
    int arr[5];
    for (int i = 0; i < 5; ++i)
    {
        printf("%d\n", arr[i]);
    }
}

int main(int argc, char const *argv[])
{
    init();
    test();
    return 0;
}
```

初始化 `int arr[5]`

主函数

输出未初始化的 `arr`

调用 `init`

调用 `test`

输出结果：

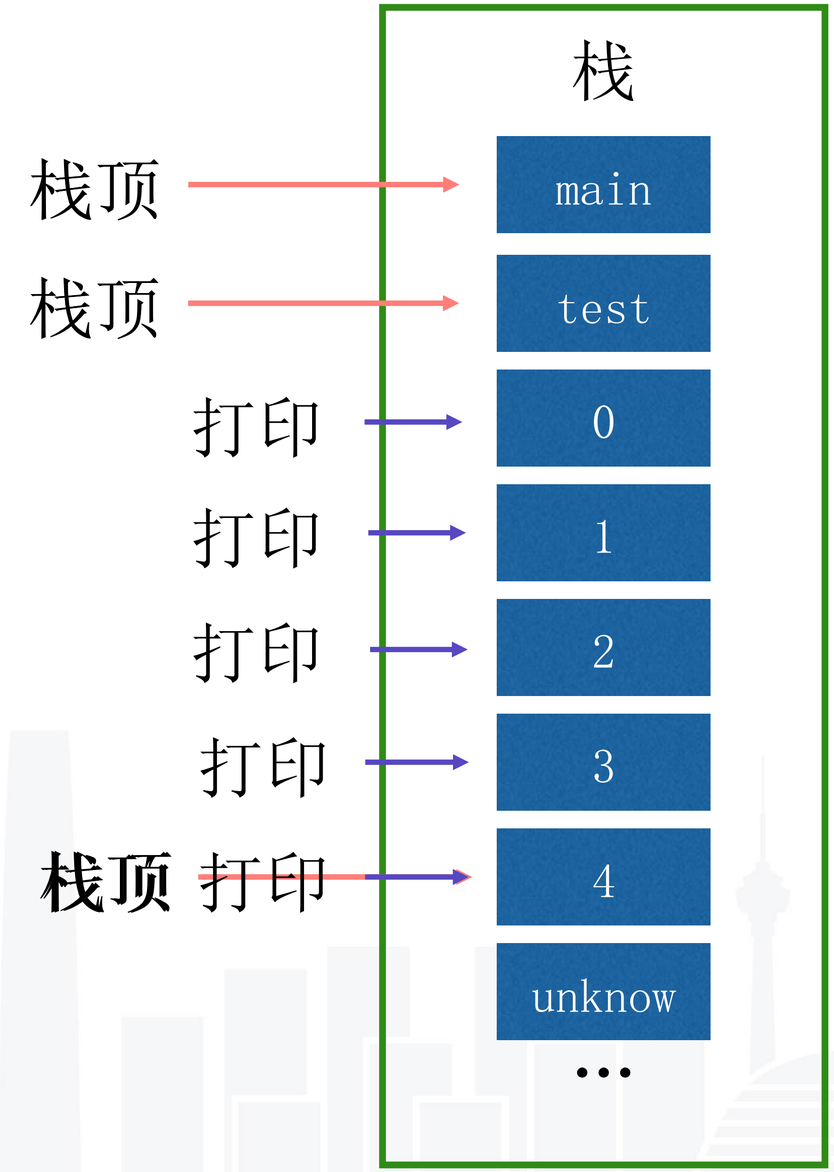
```
0
1
2
3
4
```

```c
#include <stdio.h>

void init()
{
    int arr[5] = {};
    for (int i = 0; i < 5; ++i)
    {
        arr[i] = i;
    }
}

void test()
{
    int arr[5];
    for (int i = 0; i < 5; ++i)
    {
        printf("%d\n", arr[i]);
    }
}

int main(int argc, char const *argv[])
{
    init();
    test();
    return 0;
}
```

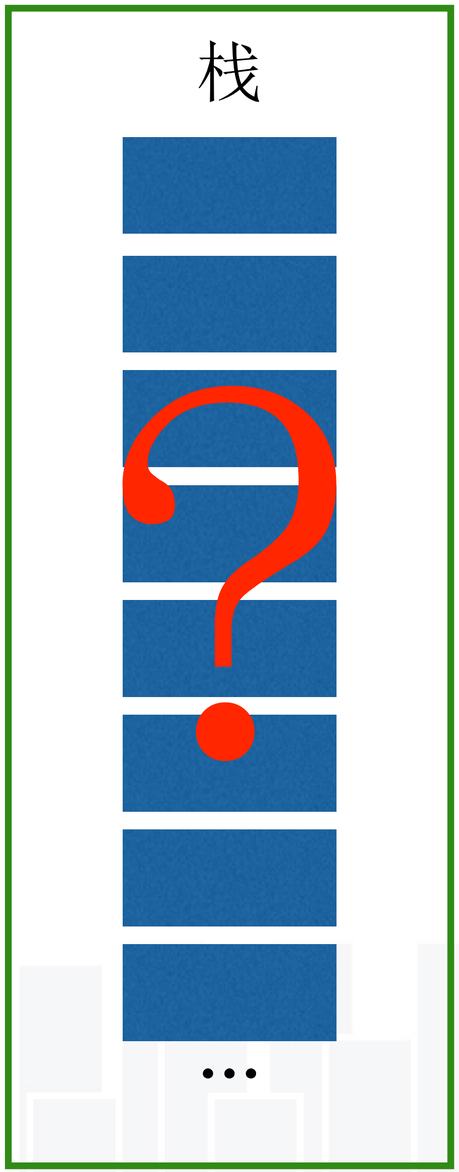
主函数

init

test

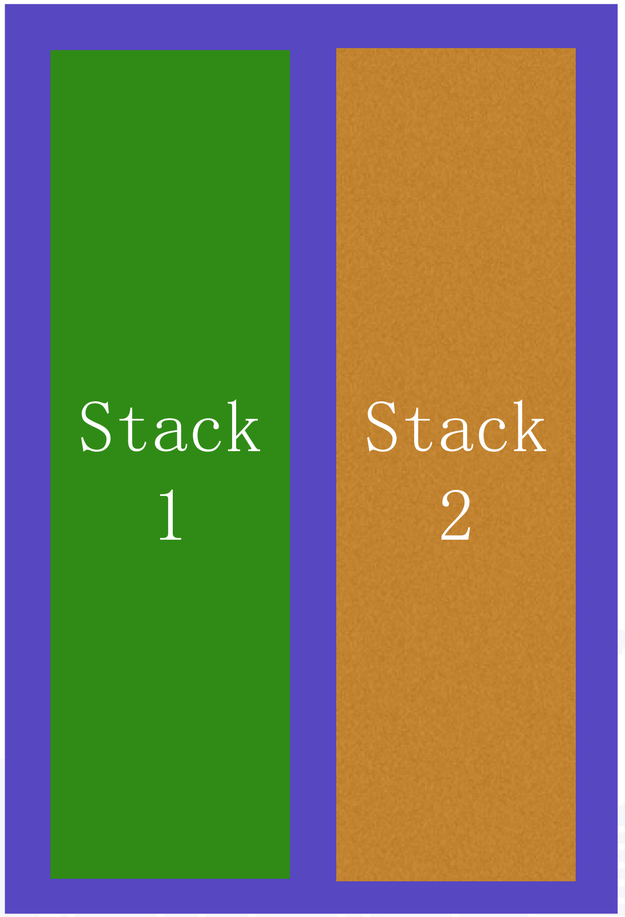


```
function test() {
  let number = 123;
  return number;
}

function init() {
  let arr = [1,2,3,4,5];
  return {
    get: (i) => arr[i]
  };
}

let {get} = init();
let n = test();
```

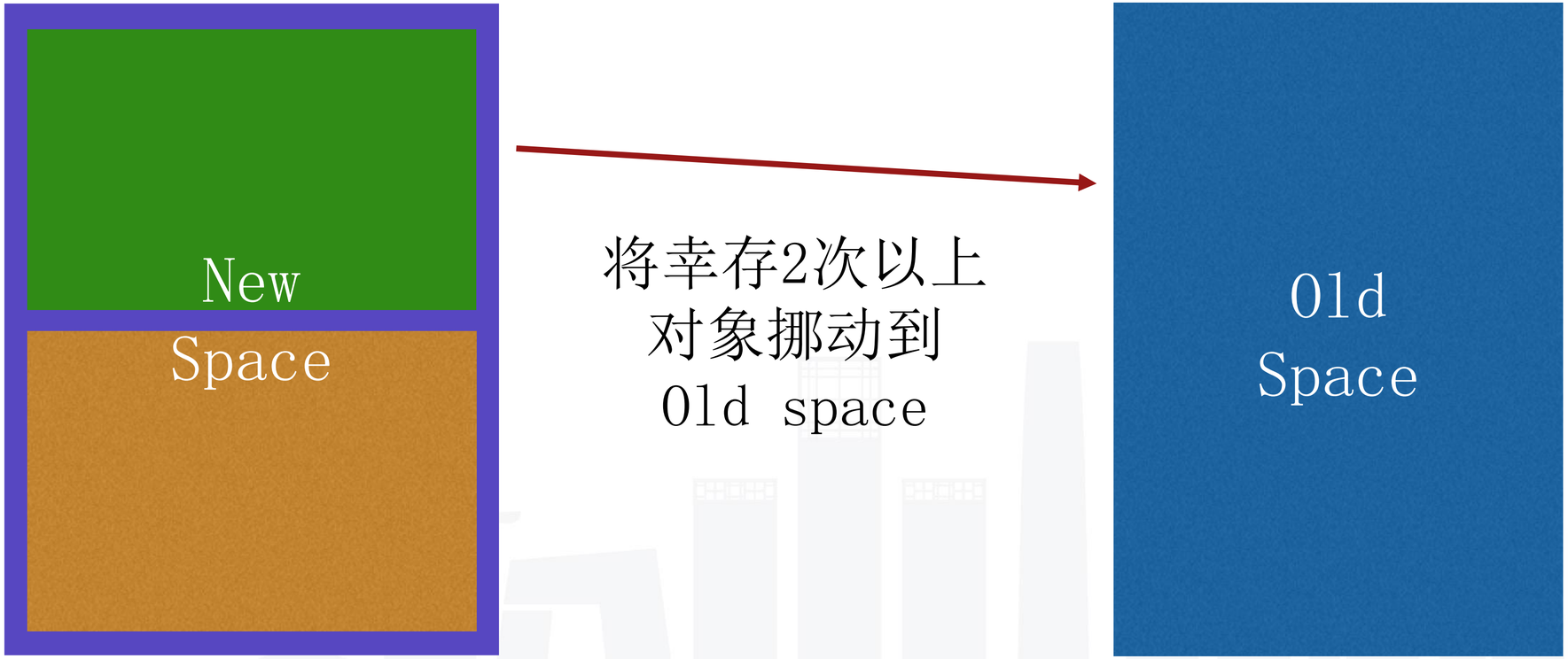


V8 New Space

1. 准备两个 Stack
2. 函数执行完不释放内存
3. 当正在使用的 Stack 快满，将不打算释放的内存到另外一个 Stack
4. 释放整个旧的 stack，使用新 Stack



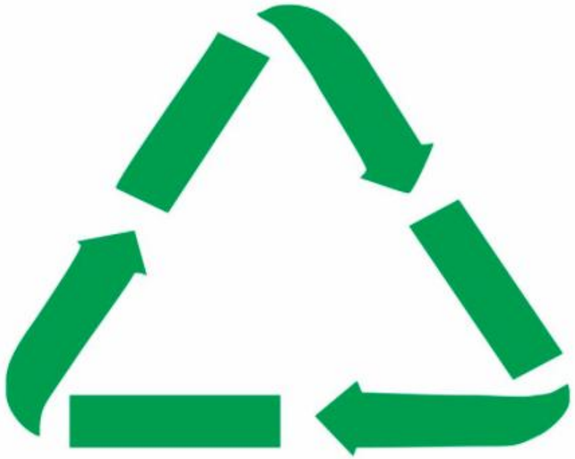
V8 Old Space



内存结构

新生代	New Space	新数据存储区
老生代	Old Space	老数据存储区
	Large Object Space	大对象存储区
	Map Space	对象结构信息
	Code Space	机器码存储区

Garbage Collection



1

Scavenge

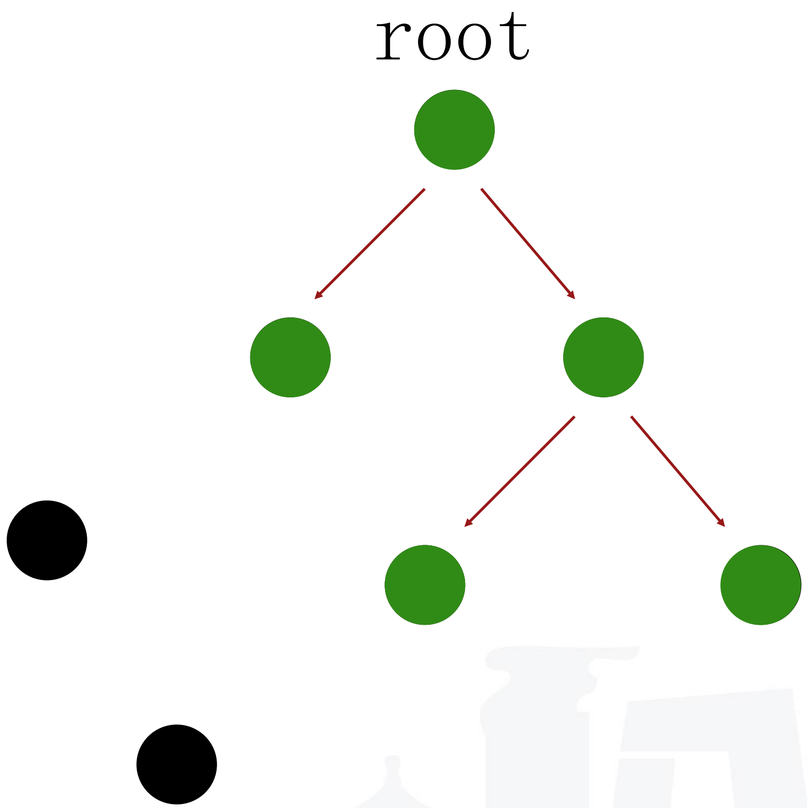
准备两份内存轮番使用，当某分内存即将用完时切换

2

Mark-Sweep/Compact

根据引用标记对象后视情况选择方法清理

Garbage Collection



V8 的 GC 的触发由 v8 自身决定（内含默认4个线程）。如果要需要主动触发 GC，可以在 node 启动时加上 -expose-gc 参数（不推荐）。

Memory Leak

当变量挂靠在 root 以及 module.exports 等全局变量上时，对象的内存始终不会释放

闭包因为写法的问题
容易使得闭包引用的作用域内存泄漏

异常产生之后没有正确恢复状态可能导致内存泄漏

对同一个事件重复监听或者忘记移除
容易导致内存泄漏

Memory Leak

Node.js 的事件监听也可能出现的内存泄漏。例如对同一个事件重复监听，忘记移除（removeListener），将造成内存泄漏。这种情况很容易在复用对象上添加事件时出现，所以事件重复监听可能收到如下警告：

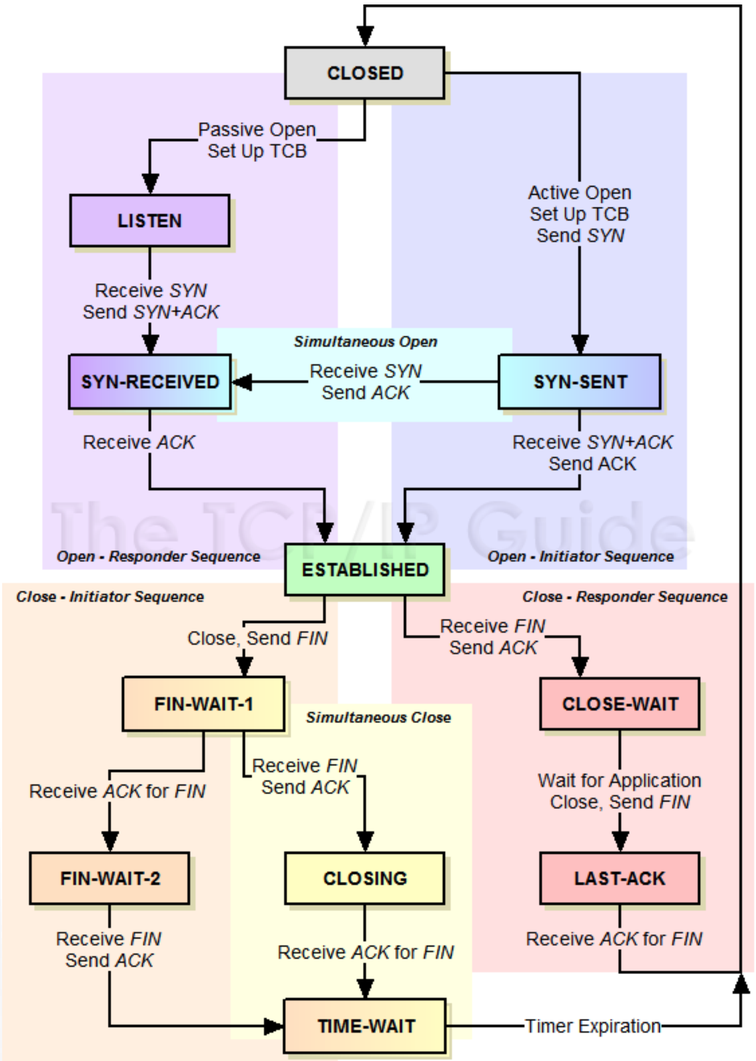
```
(node:2752) Warning: Possible EventEmitter memory leak detected。
11 myTest listeners added。Use emitter。setMaxListeners() to
increase limit
```

例如，Node.js 中 Http.Agent可能造成的内存泄漏。当 Agent keepAlive 为 true 或者短时间有大量情况的时候，都会复用之前使用过的 socket，如果此时在 socket 上添加事件监听，忘记清除的话，因为 socket 的复用，将导致事件重复监听从而产生内存泄漏。

Common Problem

异步

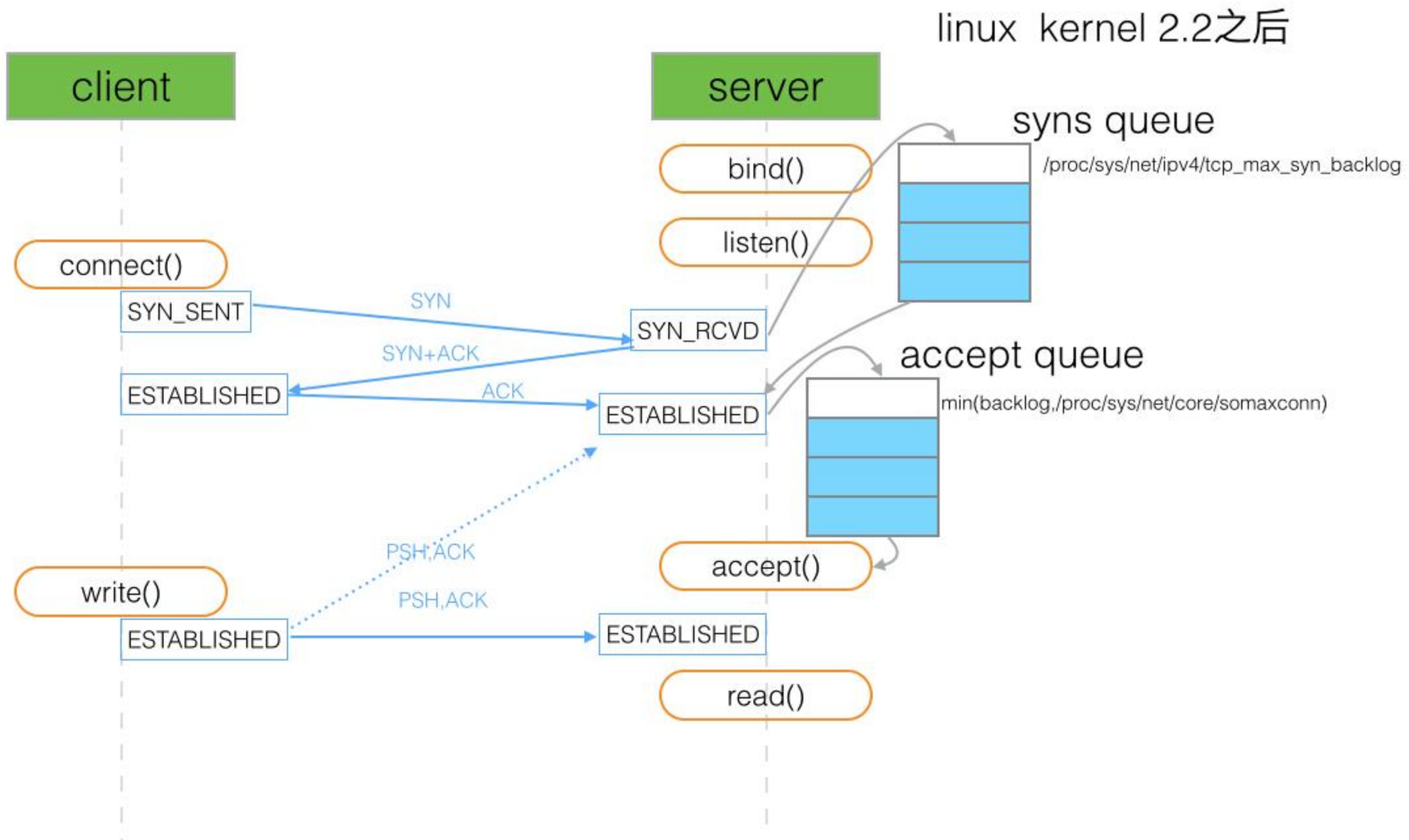
1. 流程处理曲折
2. 闭包持有引用的释放问题
3. 异步队列的问题
4. 不适合处理复杂的状态机



异步

1. 使用 Promise, Async/await 等现代方式
2. 学习 V8 内存结构了解闭包释放问题
3. 遵循 immutable 等思想减少操作副作用
4. 使用队列结构进行可控的异步并发

异步队列


linux kernel 2.2之后
client
server
syns queue
/proc/sys/net/ipv4/tcp_max_syn_backlog
bind()
listen()
connect()
SYN_SENT
SYN
SYN_RCVD
SYN+ACK
ESTABLISHED
ACK
ESTABLISHED
accept queue
min(backlog,/proc/sys/net/core/somaxconn)
accept()
PSH,ACK
write()
PSH,ACK
ESTABLISHED
ESTABLISHED
read()

Common Problem

弱计算

Node.js 擅长 IO 密集型应用

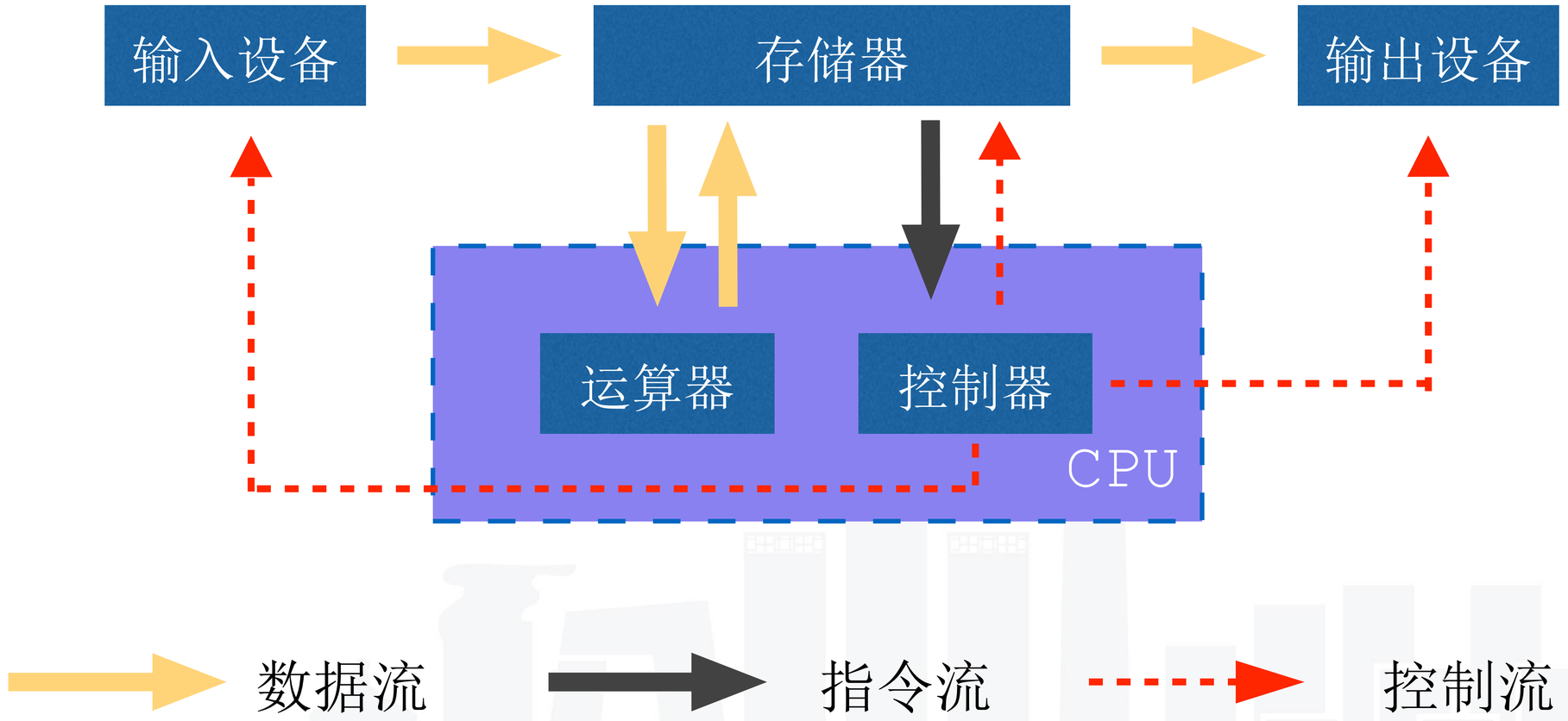
不擅长 CPU 密集型

弱计算

什么是 IO 密集型？

什么是 CPU 密集型？

弱计算



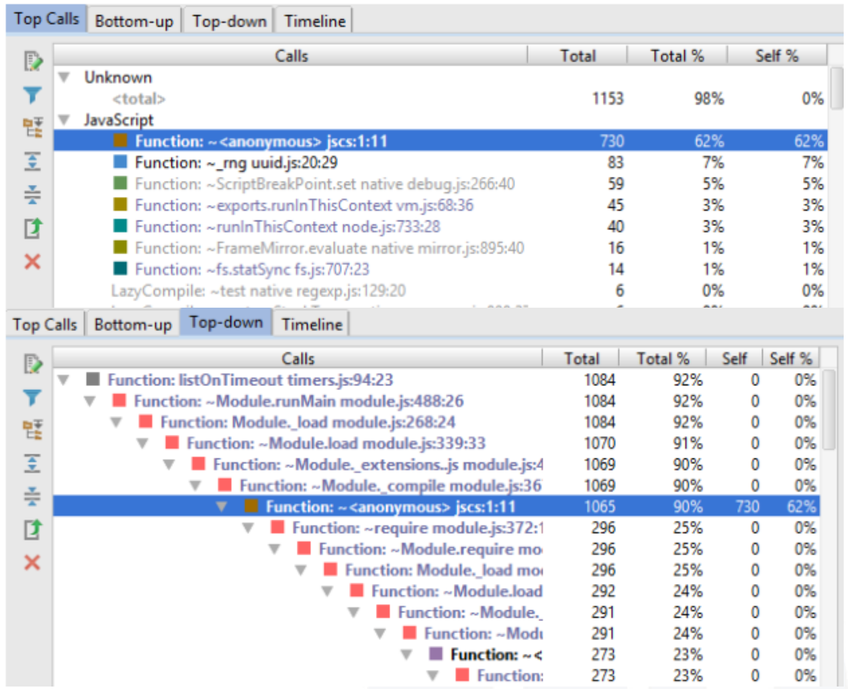
弱计算

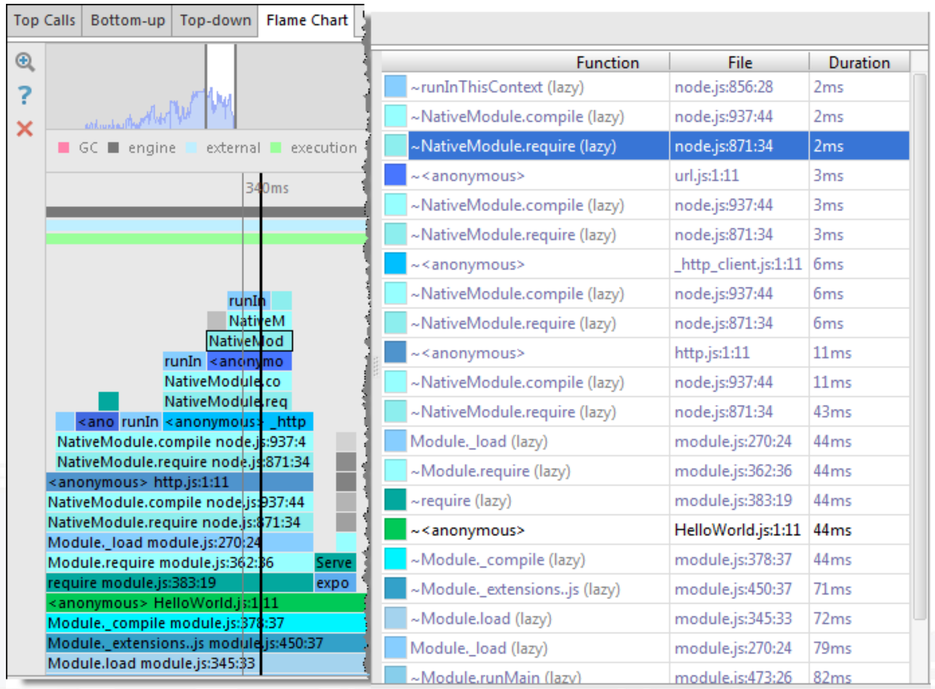
1. 确认业务类型
   - CPU密集型
   - IO密集型
2. 了解 CPU 在干什么
   - 死循环　　GC
   - 序列化对象　　Crypt...
3. 常规监控
   - 利用率
   - 饱和度

弱计算

CPU Profilling

火焰图





webstorm 工具 https://www.jetbrains.com/help/webstorm/2016.2/v8-cpu-and-memory-profiling.html

Common Problem

内存泄漏

- **1．引用问题：** 不论是闭包编写，还是事件监听没注意释放。你需要知道什么情况下，你的引用是被持有的，什么情况下又不是。所有引用问题归结到最后都是 v8 内存释放原理了解程度的问题。

- **2．队列问题：** 一个流程调用的过程中可能经过了非常多个过程，包括通信层、业务层、数据层等，其中每一个层级对于事务的处理都存在的队列的问题，你需要避免整个流程中某个环节的负载超过其能处理的上限。

- **3．CPU问题：** 引用的释放（GC）、队列设置的不合理（超过负载）、业务逻辑的编写问题（死循环）都可能导致 CPU 资源紧张，而 CPU 资源紧张同样会导致内存泄漏（没有足够的 CPU 执行 GC 操作，释放速度赶不上生产速度）。

- **4．错误处理：** 错误出现之后，如果没有对当前流程进行正确的状态恢复，可能导致错误状态下的内存始终得不到释放从而导致内存泄漏。

Common Problem

部署

Node.js 的 child_process.fork

与

POSIX 的 fork

部署

- **1. exec:** 启动一个子进程来执行命令，调用 bash 来解释命令，所以如果有命令有外部参数，则需要注意被注入的情况。

- **2. spawn:** 更安全的启动一个子进程来执行命令，使用 option 传入各种参数来设置子进程的 stdin、stdout 等。通过内置的命名管道来与子进程建立 IPC 通信。

- **3. fork:** spawn 的特殊情况，专门用来产生 worker 或者 worker 池。 返回值是 ChildProcess 对象可以方便的与子进程交互。

Node.js 的 child_process.fork() 不像 POSIX fork(2) 系统调用，不会克隆当前父进程的空间，也不需要手动的等待子进程结束回收。

部署

1. IPC 通信问题

RPC

通过 Node.js 的 fork 创建的进程使用的是其内置的 IPC（进程间通信）功能，该功能基于管道实现。

目前在实践使用过程中发现其自带的 IPC 通信功能较弱，在传输较大数据（1MB以上）是存在性能问题，所以不推荐使用。

架构调整

2. swap 内存异常

master 健康检查

在线上部署的过程中出现机器的 swap 内存爆满情况，排查发现多进程模式中存在 master 死亡后没有通知到 worker 终止进程，使得 worker 成为孤儿进程被系统 init 领养，在长时间无请求的情况下将 worker 的内存折叠进入 swap 内存。

节点注册（zookeeper等）

Common Problem

分布式

1. cluster 的负载均衡

- 句柄共享 (win)

由主进程创建 socket 监听端口后，将 socket 句柄直接分发给相应的 worker，然后当连接进来时，就直接由相应的 worker 来接收连接并处理。 多个 worker 之间会存在竞争关系，产生“惊群效应”

- round-robin (*nix)

通过时间片轮转法（round-robin）分发连接．主进程监听端口，接收到新连接之后，通过时间片轮转法来决定将接收到的客户端的 socket 句柄传递给指定的 worker 处理．至于每个连接由哪个 worker 来处理，完全由内置的循环算法决定．

分布式

2. 数据一致性

- Mongodb 弱项
  - MonogoDB 的事务能力分常弱，如果你的业务对数据一致性有要求，请更换事务能力更强的数据库，比如 mysql 的 innodb
- 异步的先天弱势
  - 异步的写法以及异步的流程，使得失败操作的回滚存在先天的困难。

Common Problem

维护

- **1.日志思想：** 与传统语言不同，Node.js 操作系统资源的形式几乎都是异步，不像 PHP 等传统的情况可以使用 Dtrace 等动态分析工具。并且异步的情况下，错误栈存在曲折、截断的情况。

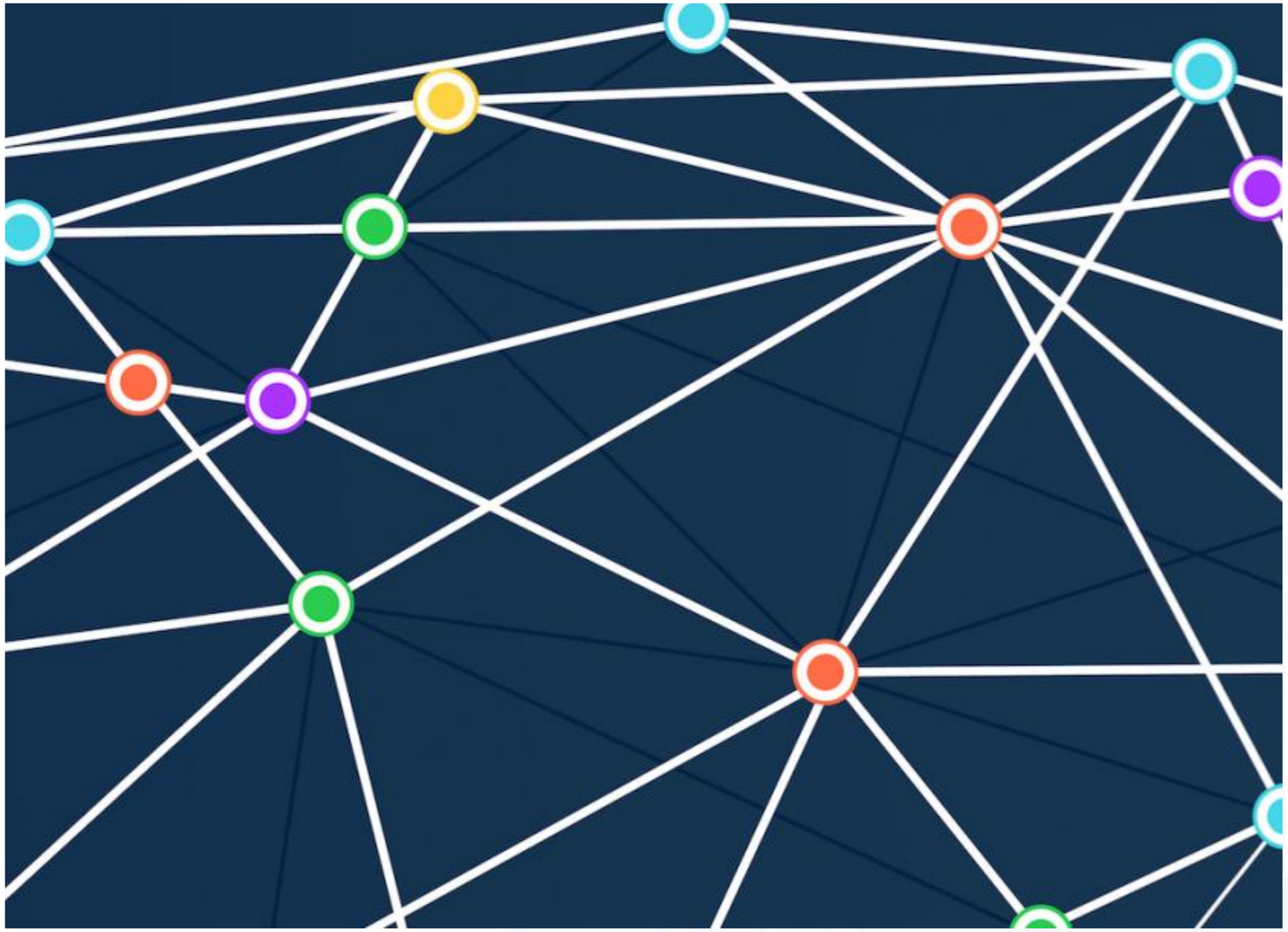
- **2.异常处理：** 上文中有提到过，异常出现之后，如果没有进行正确的恢复操作可能导致内存泄漏。其根本原因在于一些异常产生之后 Node.js 的行为是未定义的。官方有明确表示 process 的 uncaughtException 事件是为了让你准备之后再 exit 进程。

- **3.依赖管理：** 2016年的 left-pad 事件让大家意识到了使用第三方依赖存在的安全隐患，同时如果版本未做明确限制的话依赖的模块如果出现 breaking change 可能导致项目构建失败。

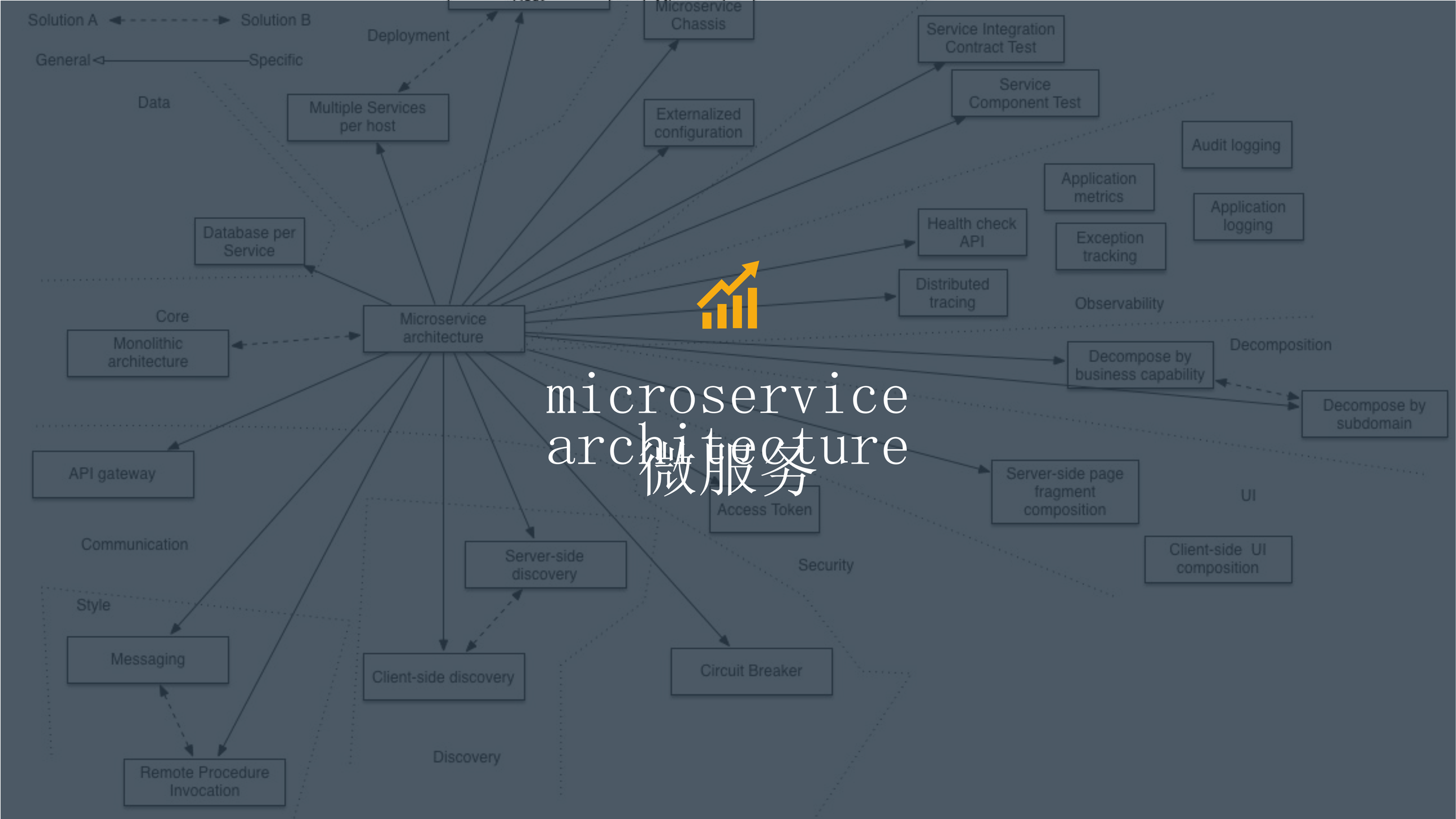
维护并不是一个孤立的问题，包括前面提到的异步、CPU、内存、部署、分布式，某一个环节如果架构的不好，必然导致后期维护困难。

Common Problem

大型应用的短板



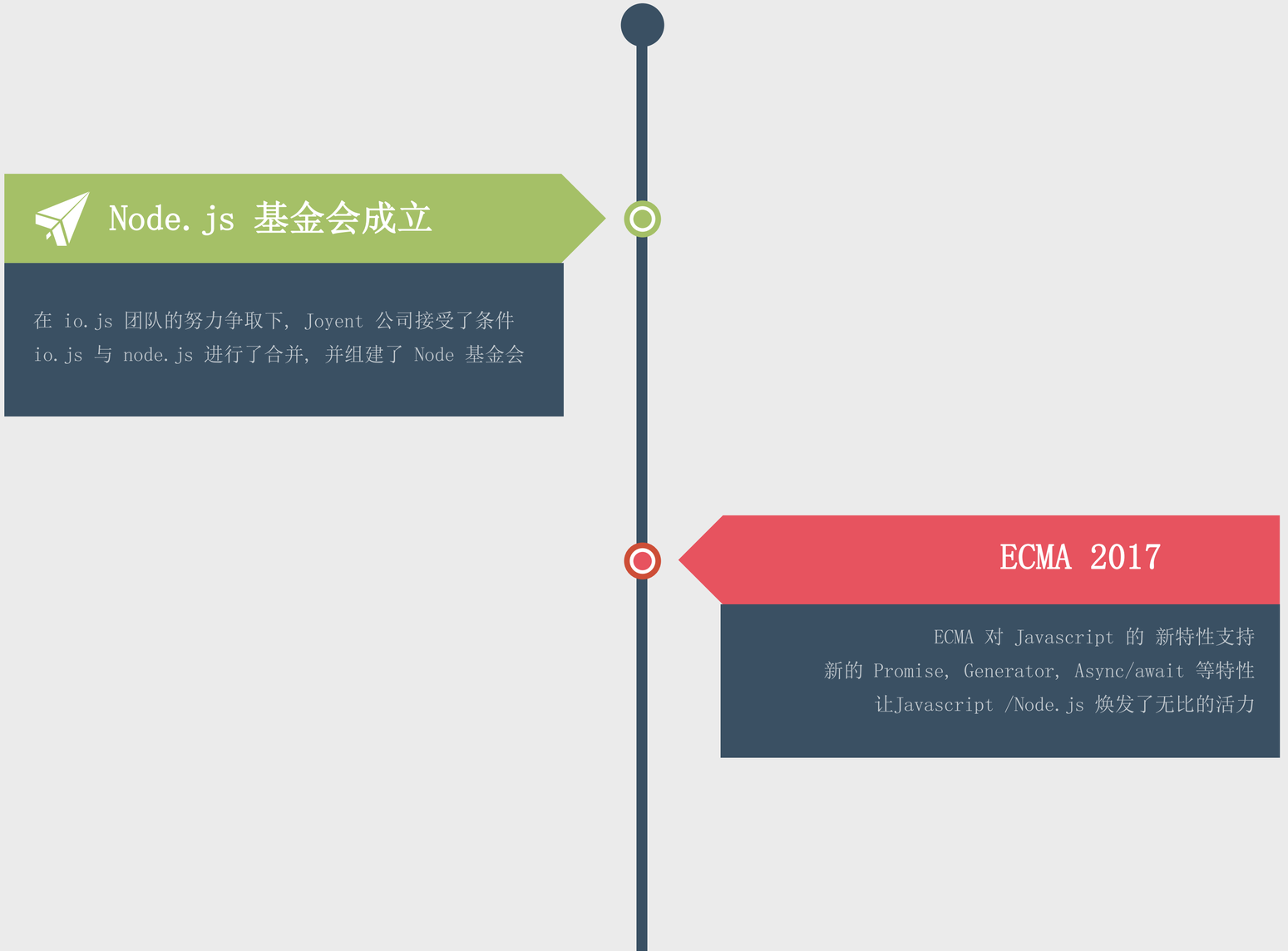

It doesn't work..... why?
It works..... why?
VIA 9GAG.COM
9GAG is your best source of fun.



microservice architecture

微服务

Step by Step


Node.js 基金会成立
在 io.js 团队的努力争取下，Joyent 公司接受了条件
io.js 与 node.js 进行了合并，并组建了 Node 基金会
ECMA 2017
ECMA 对 Javascript 的 新特性支持
新的 Promise，Generator，Async/await 等特性
让Javascript /Node.js 焕发了无比的活力

JavaScript 称霸 Github

Node.js 的模块生态圈，NPM 已然成为世界上轮子最多的地方， e sem .

NPM 数目破 40万

Node.js 的生态圈 NPM 模块系统
已然成为世界上轮子最多的地方
JS开发者对造轮子的热情大大超乎人们的预料

官方 Security 平台

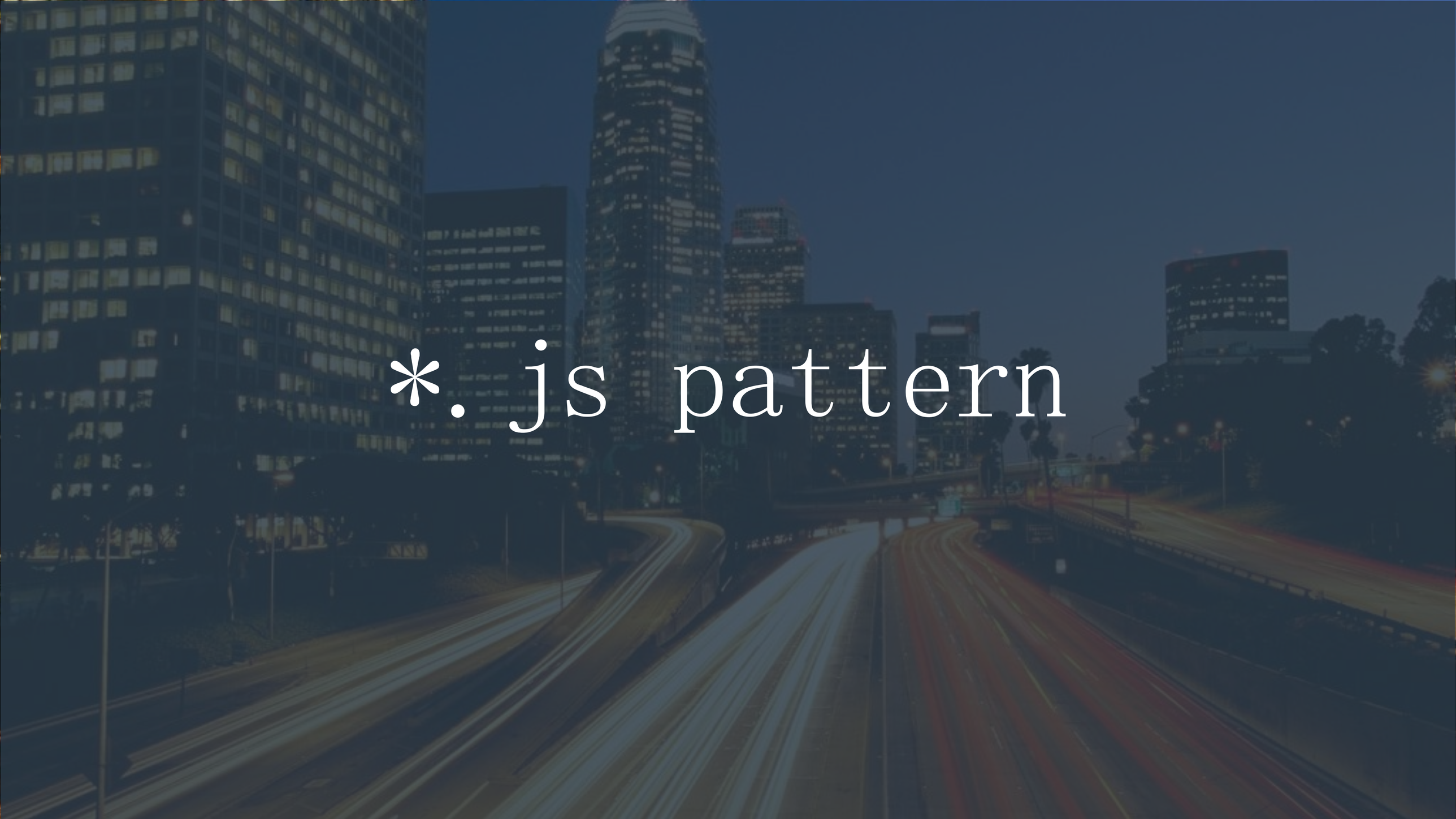
leftpad 事件，greenkeeper
npms.io 与 nodesecurity.io 的开放

加入 Linux 基金会

正式加入Linux 的大家族
这是一个分常重要的转折
意味着后续 Node.js 可能得到 Linux 内核方面的支持

Node.js v7.6 发布

async/await 可以正式使用
使用新版 V8 引擎
内存优化 性能优化



*.js pattern

Node.js 面试github.com/ElemeFE/node-interview

微博@Lellansin

Thanks!

INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE

主办方 Geekbang. 极客邦科技 InfoQ