



# Node.js在YunOS中的最佳实践

叶敬福

# 摘要

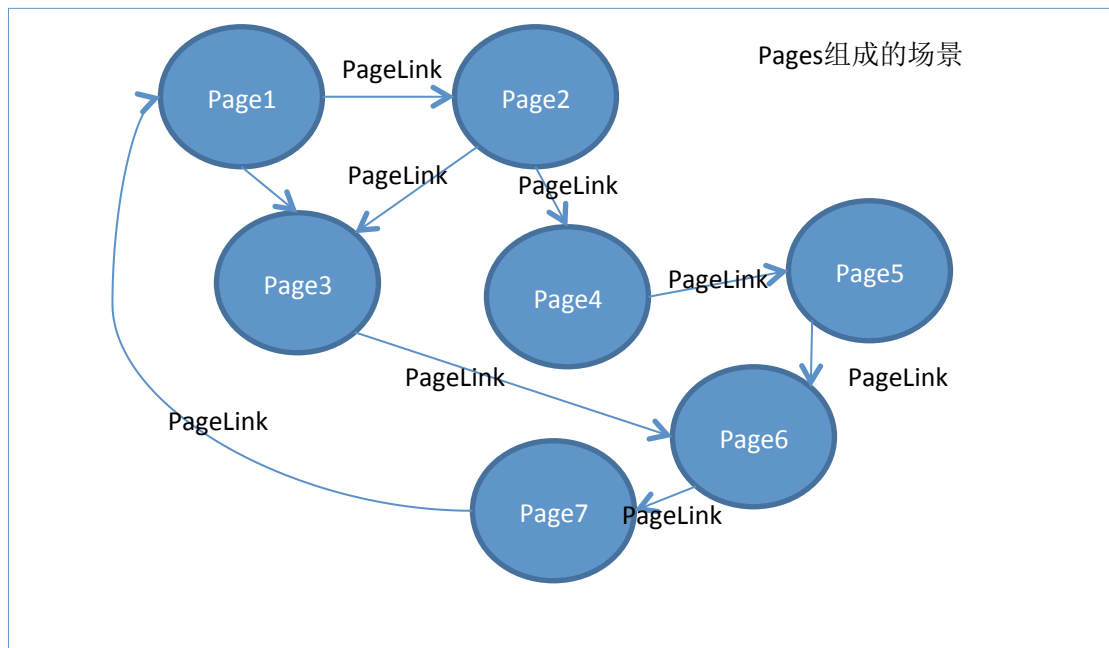
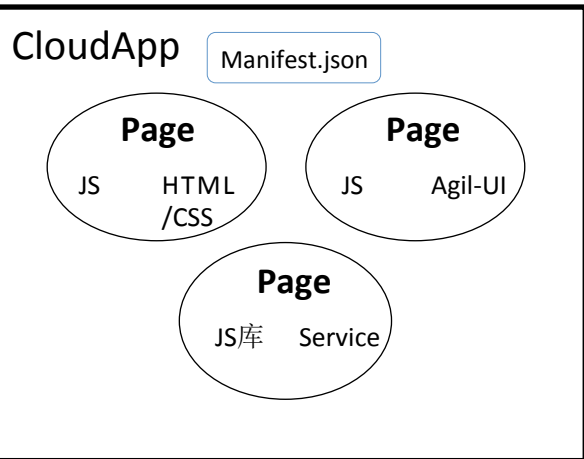
- YunOS: 万物互联
- YunOS选择Node.js
- Node.js在YunOS中的最佳实践
  - 对IO优先的Looper机制的改造和优化
  - SAB + Worker机制的实践
  - 性能优化的实践



万物互联

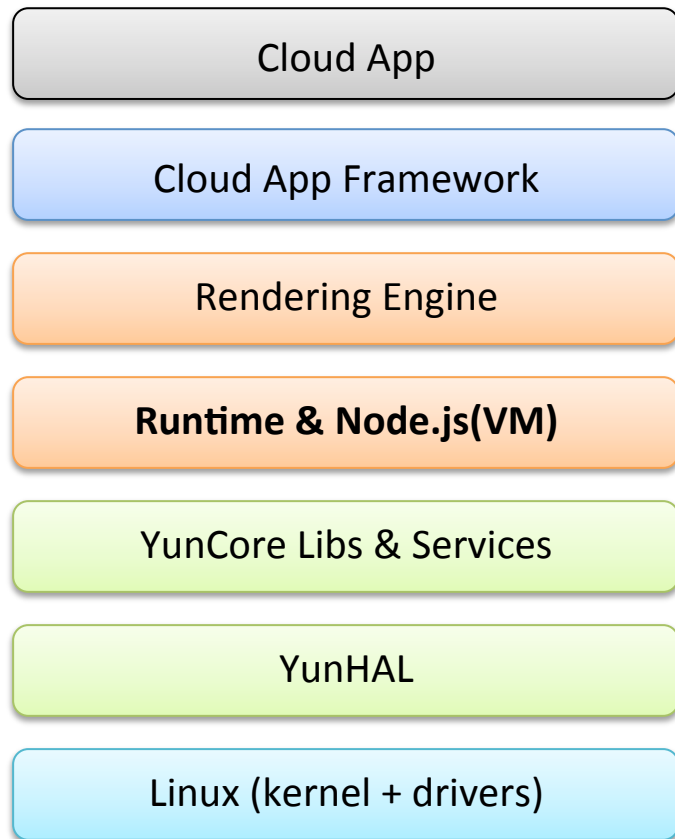


# Cloud App应用场景化



# YunOS选择Node.js

# 基础技术架构



# Node.js对YunOS的助力

技术

模块化、Native互调、事件模型

社区

开发者生态成熟

跨端

计算无处不在

分发

云端一体，即点即用





# Node.js的不足



作为JavaScript虚拟机还不成熟  
(缺少完整的libcore)



与V8强绑定，解耦或升级很困难  
(native调用强依赖V8接口)



CPU密集的计算场景中存在性能问题  
(单线程，事件驱动，GC效率低)

# Node.js在YunOS中的最佳实践

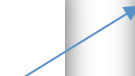
# 对IO优先的Looper机制的改造和优化

难点:

1. Node.js的消息循环机制完全由IO事件来驱动
2. 终端设备的场景在UI渲染及非IO任务的及时响应的需求

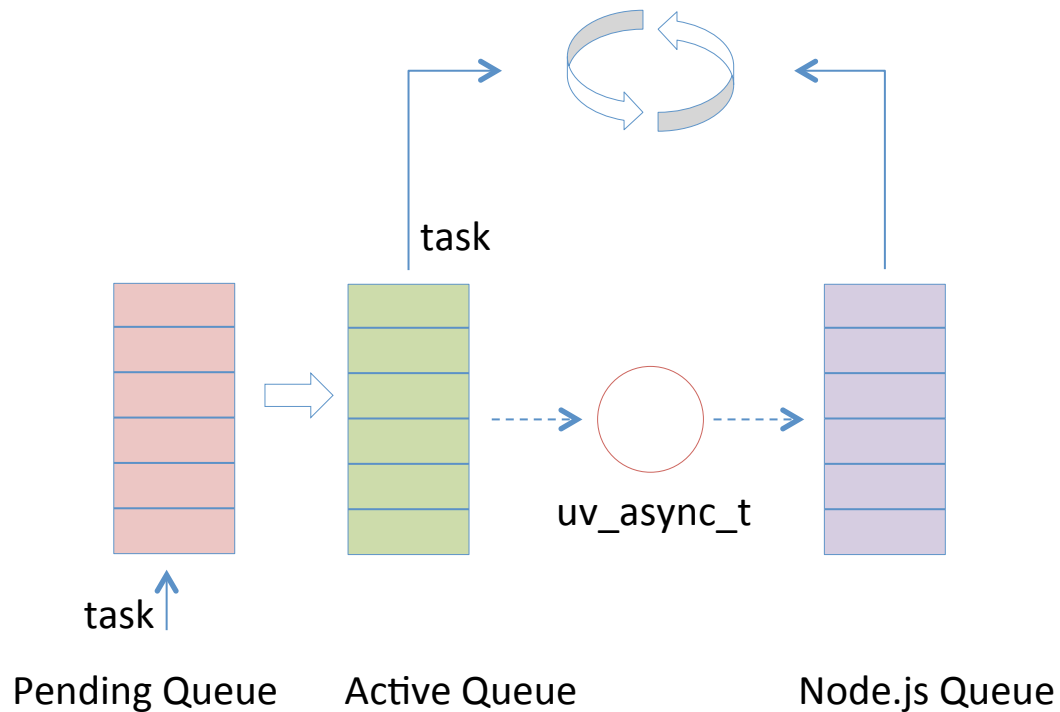
Node.js的消息循环:

```
bool more;  
do {  
    more = uv_run(env->event_loop(), UV_RUN_ONCE);  
} while (more);
```



```
int r = uv__loop_alive(loop);  
while (r != 0 && loop->stop_flag == 0) {  
    uv__io_poll(loop, timeout);  
    r = uv__loop_alive(loop);  
}
```

# YunOS对Looper改造的实践



1. 系统任务与Node.js的IO任务融合，两者地位基本平等，系统任务优先级略高

2. 系统任务通过异步事件接入Node.js的队列中

# SAB + Worker机制的实践

YunOS对多任务的需求:

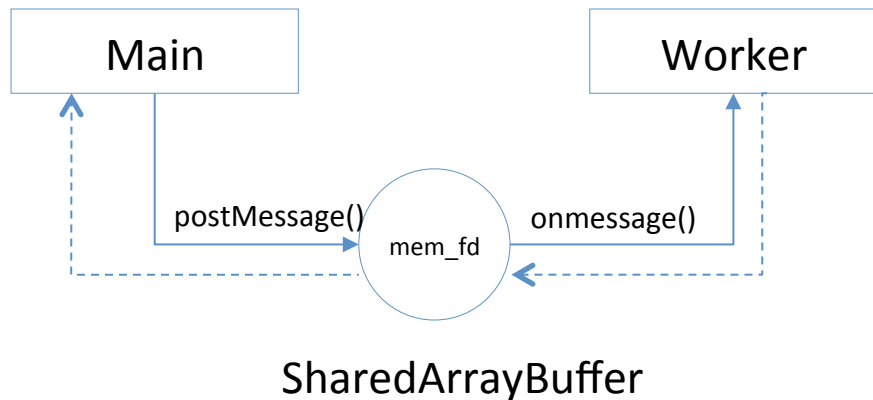
1. 安全沙箱对多进程的限制(不能使用多进程)
2. 实际应用以计算为主(CPU密集), 对多任务并行有需求
3. 并行任务在通信上的性能要求

Node.js对多任务的支持:

1. 多进程及多进程集群(cluster)
2. Node.js Addon + pthread
3. (JavaScript Webworker + postMessage + onMessage)

## YunOS实现了Worker + SAB:

1. 基于复制进程的方式实现Worker的创建
2. Worker内部支持Node.js的接口
3. SAB(SharedArrayBuffer)基于共享内存的方式实现，实现高效率通信
4. SAB支持转换到TypedArray (如Int32Array等)，方便使用
5. SAB实现了Atomics的接口规范，支持互斥访问和同步消息
6. 应用场景：将计算密集型任务从UI线程分离出来，计算完成后通知UI线程



## Worker + SAB举例:

```
// main.js
var Worker = require("worker");
var workerDemo = new Worker("workerdemo.js");

var sab = new SharedArrayBuffer(4096);
var i32a = new Int32Array(sab);
for(var i = 0; i < i32a.length; i++) {
    i32a[i] = i;
}
workerDemo.postMessage(sab);
workerDemo.onmessage = function(env) {
    console.log('main received: ', env.data);
};
```

```
// workerdemo.js
...
global.onmessage = function(env) {
    var sab = env.data;
    // view sab buffer in int32 array
    var i32a = new Int32Array(sab);
    var msg = i32a.join(', ');
    console.log('worker received:', msg);
    postMessage('done');
}
```

# 性能优化的实践(一)

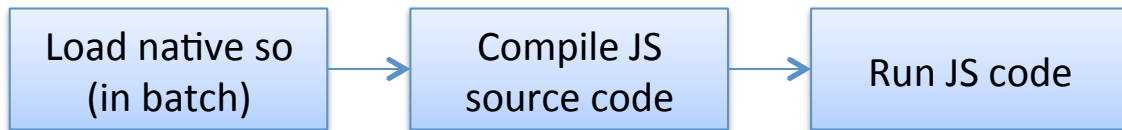
## JavaScript to native:

将系统级JS模块合并加载，大幅减少IO次数，提高系统整体效率。

Compile time:



Run time:



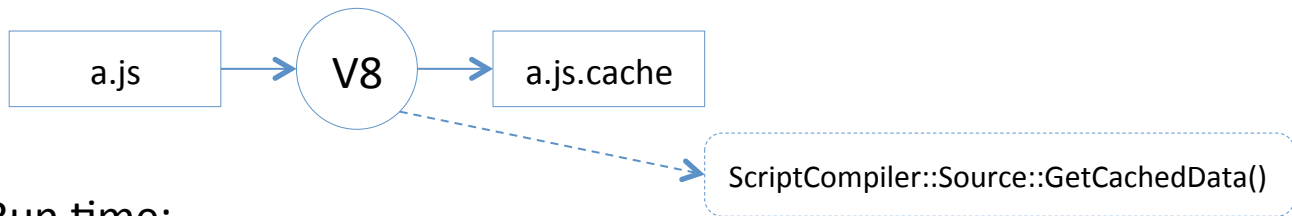


# 性能优化的实践(二)

## Code cache:

充分利用V8引擎的能力，预先将JavaScript代码编译好生成cache文件  
运行时加载cache文件，跳过编译过程，直接运行预编译的代码，提高运行效率

Compile time:



Run time:



# 性能优化的实践(三)

## Lazy Load:

`require()`时不真正加载模块，而是在第一次访问对象的时候才真正加载模块  
通过将加载时间分散到运行时刻，减轻启动时的压力，优化启动时间

```
var obj = require('test.js'); // load
obj.foo();
obj.bar;
```



```
var obj = require('test.js', 'lazy'); // not load
...
obj.foo(); // load at first access
obj.bar;
```

# 性能优化的实践(四)

## 其他

- Timer coalescing
- 模块可卸载
- 模块裁剪
- ...

Thanks!