# Node在个人直播中的应用

IMWEB-linkzhu

# 介绍

- 腾讯前端工程师——朱林(link)
- IMWEB团队成员
- 齐齐互动视频、花样直播、NOW直播
- 现在主要负责NodeJS在互动视频业务中的落地。

# Contents


01

Part one

**为啥要用Node？**

02

Part two

**异构系统中的通信实践**

03

Part three

**小结**

PART 1

# 为啥要用Node？

# Why Node ?

- 新应用NOW直播
- APP研发为主
- 直播内容秩序监管系统

# Why Node ?

| |  |  |  |
|---|---|---|---|
| SEO |  |  |  |
| 浏览器渲染负担 |  |  |  |
| 前后端耦合度 |  |  |  |
| 缓存利用率 |  |  |  |

# Why Node ?

javascript everywhere !



一种开发语言

一种数据结构

全局性思维

沟通成本低

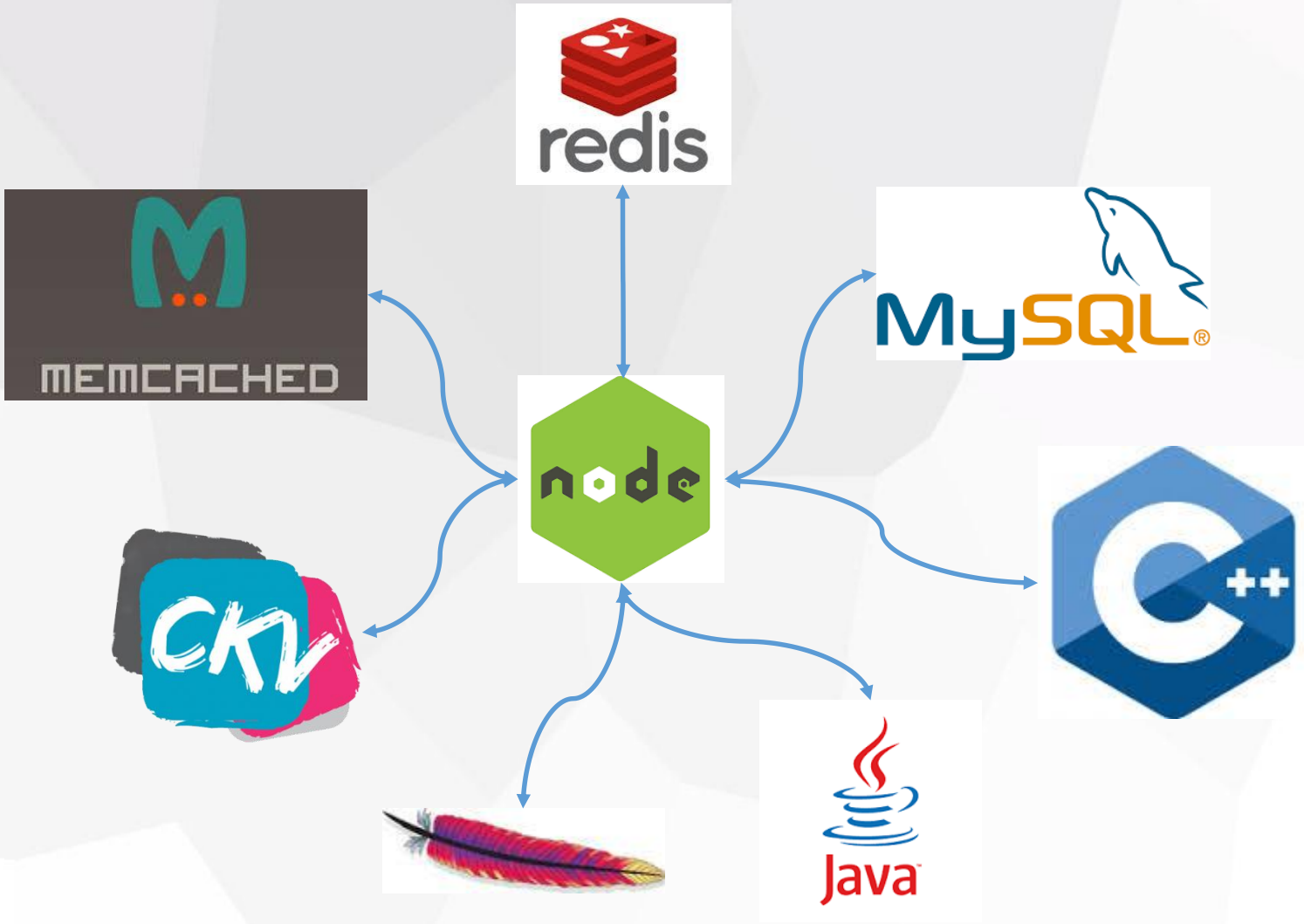
# 异构系统中的通信实践

PART 2

- Protocol Buffer
- CKV
- ZooKeeper

# 异构共存

# 异构共存



# 异构共存

**如何解决NodeJS应用与已有系统或服务的交互问题呢？**

# 异构共存

开播服务（C++）

登录服务（CKV/C++）

首页推荐服务（ZooKeeper）

私信/黑名单/...

协议（传输协议+数据序列化协议）

NOW直播秩序监管系统（Node）

# 异构共存

传输协议：

**HTTP**

**TCP**

**UDP**

**...**

# 异构共存

传输协议：

HTTP

TCP

UDP

...

# 传输协议

**HTTP：**

应用层协议

应用广泛

支持任何类型的数据对象传输

无状态，不保持长连接

文本协议

# 传输协议

**Socket：**

传输层协议

**TCP & UDP**

字节级传输数据

支持实时交互

# 传输协议

## HTTP：

## 文本协议

Server: Nginx \r\n

Connection: keep-alive \r\n

Content-Length: 707

http协议的接收端通过\r\n切分每一个字段，并通过字符串匹配得到每个字段的名字与取值。

# 传输协议

**Socket：**

二进制协议

# 传输协议

## Socket：

封解二进制包

```
// **********编制请求包
var buffer = new Buffer(28);
// 每个字段所占的长度，以这个字段可能出现的最长情况来定，这里8作为示例
buffer.write("Nginx");
buffer.write("keep-alive", 8);
buffer.writeInt32BE(707, 24);
// 到这里，buffer已经是一个完整的包

// **********解析返回包
var receiveBuffer = //do sth to get pack;
var result = {
  server: receiveBuffer.slice(0,8).toString('utf-8'),
  connection: receiveBuffer.slice(8,24).toString('utf-8'),
  contentLength: receiveBuffer.readInt32BE(24)
};
// 到这里，result就是一个解好的结果
```

# 异构共存

数据序列化协议：

**JSON**

**XML**

私有二进制协议

**Protocol Buffer**

**...**

# 异构共存

数据序列化协议：

**JSON**

**XML**

私有二进制协议

**Protocol Buffer**

**...**

# 数据序列化协议

## •二进制协议

- 二进制协议对于计算机来说**更容易解析**（通过位移计算），在解析速度上是JSON、XML这样的文本协议不可比拟的。
- 有tcp和udp**两种选择**，在一些场景下，udp传输的效率会更高。

# 数据序列化协议

•私有的二进制协议

- {int32 id: 101,  char[8] str:  ‘Hello’ }
- 101Hello:  <Buffer 0 0 0 65 0 0 0 48 65 6c 6c 6f>

# 数据序列化协议

- 私有的二进制协议

**怎么扩展字段？？？**

# 数据序列化协议

你值得更好的！

更小！更快！更灵活！

# 异构共存

数据序列化协议：

**JSON**

**XML**

私有二进制协议

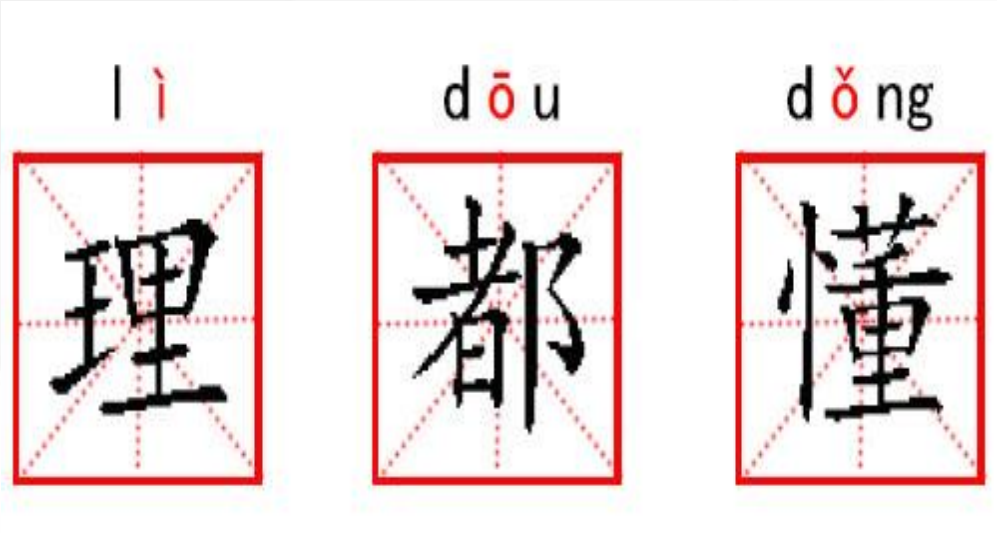
**Protocol Buffer**

**...**

# Protobuf是什么鬼？

## 官方定义：

Protocol Buffers 是一种轻便高效的**结构化数据存储格式**，可以用于结构化数据**序列化**，很适合做数据存储或 RPC 数据交换格式。它可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。目前提供了 C++、Java、Python 三种语言的 API。

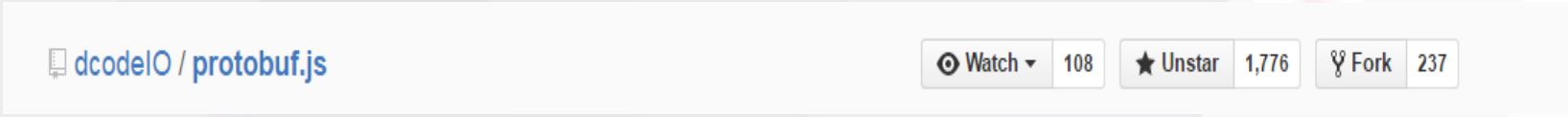



然而并没有什么卵用

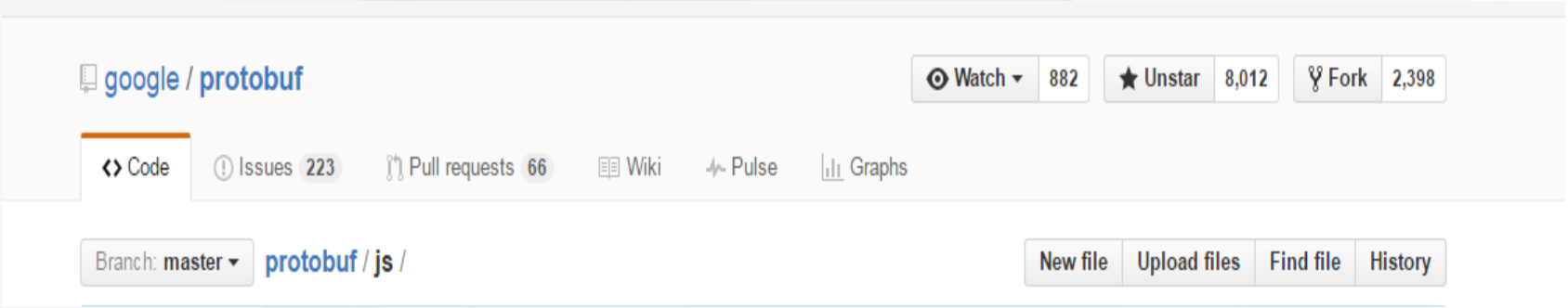

一脸懵逼

# 一个栗子

NodeJS解析protobuf文件的第三方模块：
protobuf.js：https://github.com/dcodeIO/ProtoBuf.js




Google protobuf js：
https://github.com/google/protobuf/tree/master/js




protocol-buffers：
https://github.com/mafintosh/protocol-buffers

# 一个简单的栗子帮助理解

# 一个栗子

书写protobuf文件

lm.message.proto

```
1  package lm;
2  message helloworld
3  {
4      required int32     id = 1;  // ID
5      required string    str = 2;  // str
6      optional int32     opt = 3;  //optional field
7  }
```

一个比较好的习惯是认真对待 proto 文件的文件名。比如将命名规则定于如下：

**packageName.MessageName.proto**

在上例中，package 名字叫做 lm，定义了一个消息 helloworld，该消息有三个成员，类型为 int32 的 id，另一个为类型为 string 的成员 str。opt 是一个可选的成员，即消息中可以不包含该成员。

# 一个栗子

使用protobuf.js的命令行工具编译 .proto 文件
https://github.com/dcodeIO/ProtoBuf.js/wiki/pbjs

## pbjs

Daniel Wirtz edited this page on 13 Jun 2015 · 9 revisions

Since ProtoBuf.js 4.0.0 the library ships with the `pbjs` command line utility. With it it's possible to convert between .proto and JSON descriptors and even to generate the code required to access runtime structures as pure JS (classes), an AMD module or a CommonJS module.

```
 _ |_ . _
|_)|_)|_)          ProtoBuf.js v4.0.0-b3 https://github.com/dcodeIO/ProtoBuf.js
|     '

CLI utility to convert between .proto and JSON syntax / to generate classes.

Usage: pbjs <filename> [options] [> outFile]
```

```
Options:
  --help, -h        Show help  [boolean]
  --version, -v     Show version number  [boolean]
  --source, -s      Specifies the source format. Valid formats are:

                      json       Plain JSON descriptor
                      proto      Plain .proto descriptor

  --target, -t      Specifies the target format. Valid formats are:

                      amd        Runtime structures as AMD module
                      commonjs   Runtime structures as CommonJS module
                      js         Runtime structures
                      json       Plain JSON descriptor
                      proto      Plain .proto descriptor

  --using, -u       Specifies an option to apply to the volatile builder
                    loading the source, e.g. convertFieldsToCamelCase.
  --min, -m         Minifies the output.  [default: false]
  --path, -p        Adds a directory to the include path.
  --legacy, -l      Includes legacy descriptors from google/protobuf/ if
                    explicitly referenced.  [default: false]
  --quiet, -q       Suppresses any informatory output to stderr.  [default: false]
  --use, -i         Specifies an option to apply to the emitted builder
                    utilized by your program, e.g. populateAccessors.
  --exports, -e     Specifies the namespace to export. Defaults to export
                    the root namespace.
  --dependency, -d  Library dependency to use when generating classes.
                    Defaults to 'protobufjs' for CommonJS, 'ProtoBuf' for
                    AMD modules and 'dcodeIO.ProtoBuf' for classes.
```

# 一个栗子

使用protobuf.js的命令行工具编译 .proto 文件：

```
# ./pbjs ../../lm.message.proto  -t commonjs > ../../lm.message.js
```

得到编译后的符合commonjs规范的js文件：

lm.message.proto | lm.message.js

```
module.exports = require("protobufjs").newBuilder({})['import']({
    "package": "lm",
    "messages": [
        {
            "name": "helloworld",
            "fields": [
                {
                    "rule": "required",
                    "type": "int32",
                    "name": "id",
                    "id": 1
                },
                {
                    "rule": "required",
                    "type": "string",
                    "name": "str",
                    "id": 2
                },
                {
                    "rule": "optional",
                    "type": "int32",
                    "name": "opt",
                    "id": 3
                }
            ]
        }
    ]
}).build();
```

# 一个栗子

编写 **Writer**

```
var HelloWorld = require('./lm.message.js')['lm']['helloworld'];
var fs = require('fs');

var hw = new HelloWorld({
    'id': 101,
    'str': 'Hello'
})

var buffer = hw.encode();

fs.writeFile('./test.log', buffer.toBuffer(), function(err) {
    if(!err) {
        console.log('done!');
    }
});
```

# 一个栗子

编写**Reader**

Project Preferences Help

writer.js × reader.js × lm.message.js ×

```
1  var HelloWorld = require('./lm.message.js')['lm']['helloworld'];
2  var fs = require('fs');
3
4  var buffer = fs.readFile('./test.log', function(err, data) {
5      if(!err) {
6          console.log(data); // 来看看Node里的Buffer对象长什么样子。
7          var message = HelloWorld.decode(data);
8          console.log(message);
9      }
10 })
```

# 一个栗子

运行结果

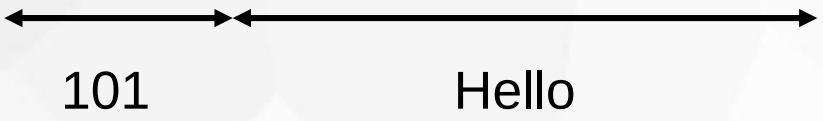
```
E:\doc\protobuf分享\example\helloworld>node writer.js
done!

E:\doc\protobuf分享\example\helloworld>node reader.js
<Buffer 08 65 12 05 48 65 6c 6c 6f>
{ id: 101, str: 'Hello', opt: null }
```

# 我们为什么要跟protobuf打交道？

•私有的二进制协议

- {int32 id: 101，char[8] str: ‘Hello’}
- 101Hello: <Buffer 0 0 0 65 0 0 0 48 65 6c 6c 6f>




**12**个字节

# 我们为什么要跟protobuf打交道？

•**Protocol Buffer**

- {int32 id: 101，string str: ‘Hello’}
- <Buffer 08 65 12 05 48 65 6c 6c 6f>

**9**个字节

# 我们为什么要跟protobuf打交道？

•JSON

- {id: 101, str: ‘Hello’}
- <Buffer 7b 22 69 64 22 3a 31 30 31 2c 22 73 74 72 22 3a 22 48 65 6c 6c 6f 22 7d>

**24**个字节

## 我们为什么要跟protobuf打交道？

# 12 vs 9 vs 24

兼具二进制的高性能和JSON的灵活

# 一个栗子


这个例子本身并无意义，但只要您稍加修改就可以将它变成更加有用的程序。比如将磁盘替换为网络 socket，那么就可以实现基于网络的数据交换任务。而存储和交换正是 Protobuf 最有效的应用领域。

# 动态链接库

# 动态链接库

- 避免造轮子
- 计算密集型任务
- npm install ffi


# CKV
# (Cloud Key-Value)

# 什么是CKV


Memcached++

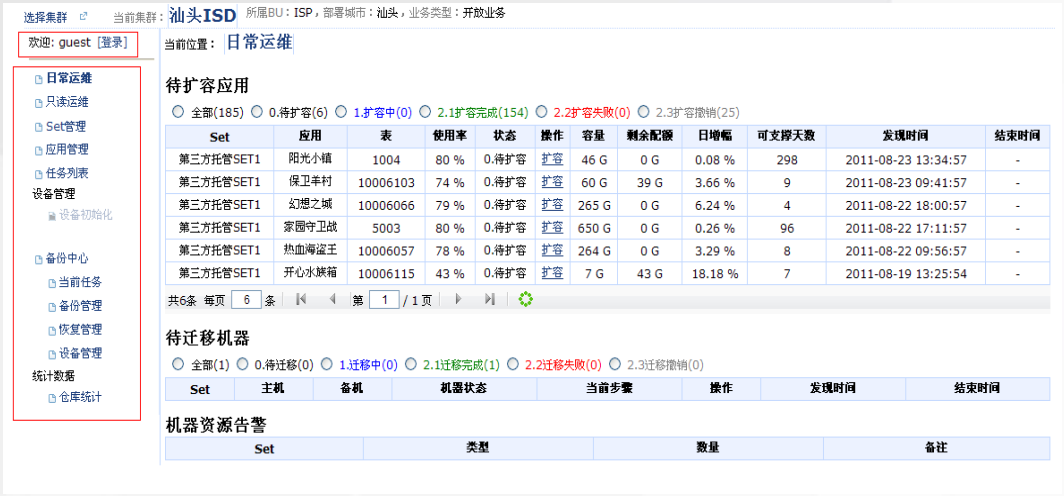
| | | | | |
|---|---|---|---|---|
| 专业运营团队 | 双点热备 | 集中备份，定点回档 | 负载均衡 | 高性能 |

# Why CKV

即时申请即时使用，开发者无需自行安装memcached。

-提供便利的运维系统，统计系统，拨测系统。

专业运营团队




业务永远不用关注扩容，机器故障等问题。

# Why CKV

双点热备

- 接入机镜像，相互容灾
- 存储机双机热备，自动切换
- 集中备份中心，任意时间回档
- 主备自动切换or人工切换

# Why CKV

集中备份，定点回档

- 可配备集中备份中心。
- 集中备份中心可完成定点回档。

镜像文件备份 + 流水旁路 + 恢复中心

# Why CKV

负载均衡

CKV后台有自动对负载过高的存储机进行负载调平，使存储机器达到最佳的使用率。不必担心一批访问量都非常低的业务分布再同一台机器上造成机器低负载或者使用率达不到要求，成本增加。

# Why CKV

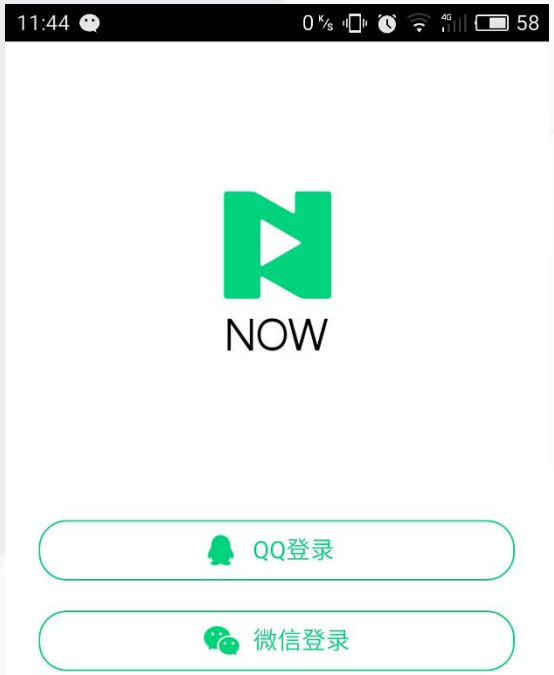
高性能

- 存储单机可同时提供19w/s的读和11w/s的写。
- 接入单机提供19w/s的穿过能力。

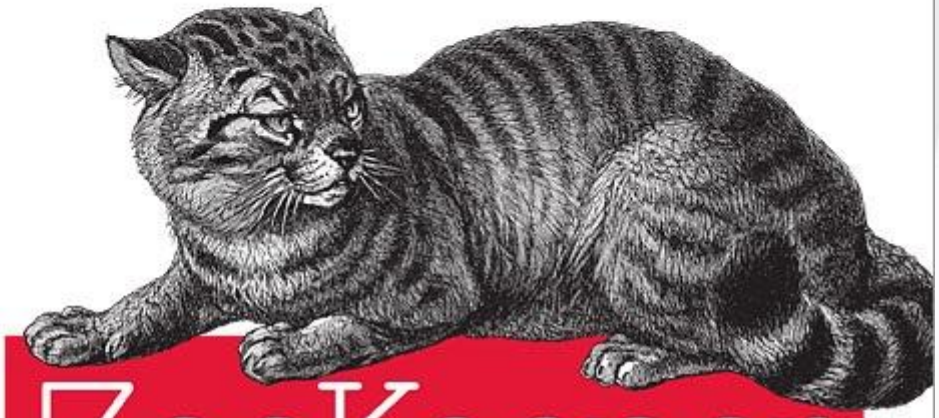
# 限制条件&适用场景

- Key长度不超过10K
- Value长度不超过10M

- 数据总量适合存储在内存中
- 任何Key-Value形式，写量很大的数据
- 延时敏感的业务



O'REILLY®



# ZooKeeper

DISTRIBUTED PROCESS COORDINATION

# 从入门到放弃

Flavio Junqueira & Benjamin Reed

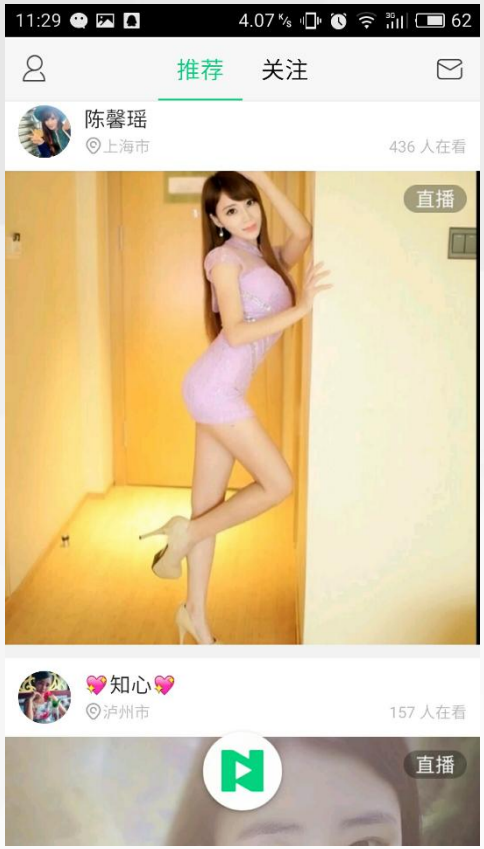
# 什么是ZooKeeper?

## 理想中的分布式应用，各个服务各司其职

实际中的分布式应用，惊群、死锁、数据一致性...

# ZooKeeper解决了什么问题？

## 核心：解决分布式系统中的一致性问题



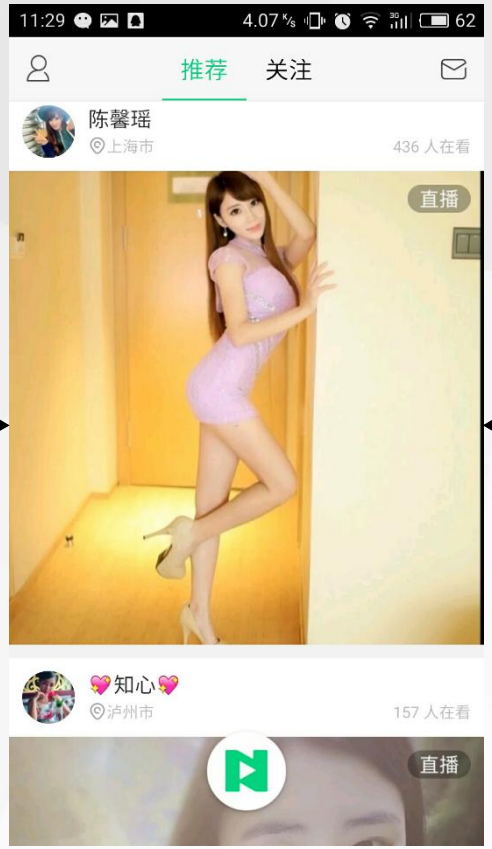
# ZooKeeper解决了什么问题？

## 核心：解决分布式系统中的一致性问题

评分系统

个性化推荐

主播类型




秩序监管

开播状态

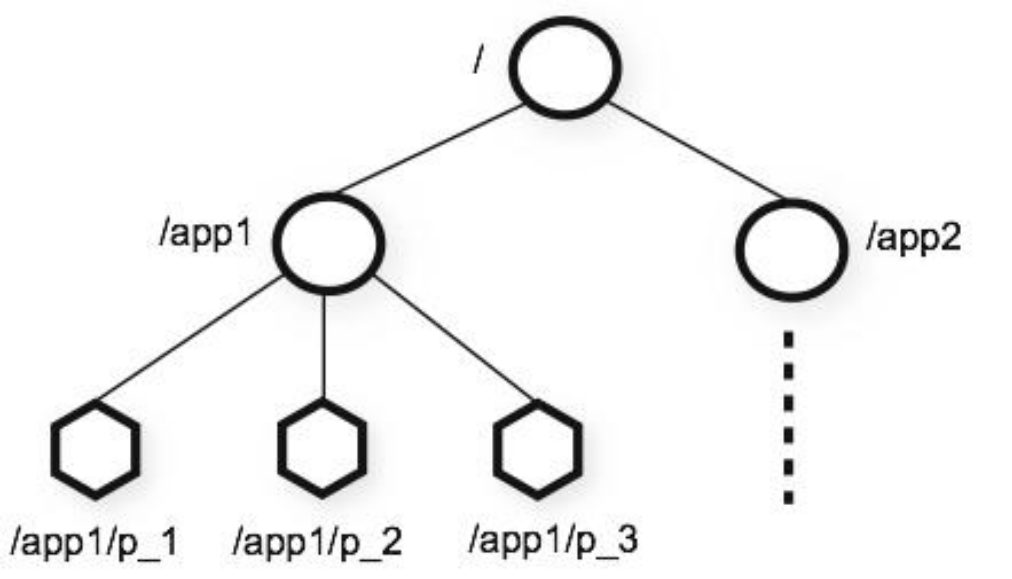
…

# ZooKeeper解决了什么问题？

HOW?

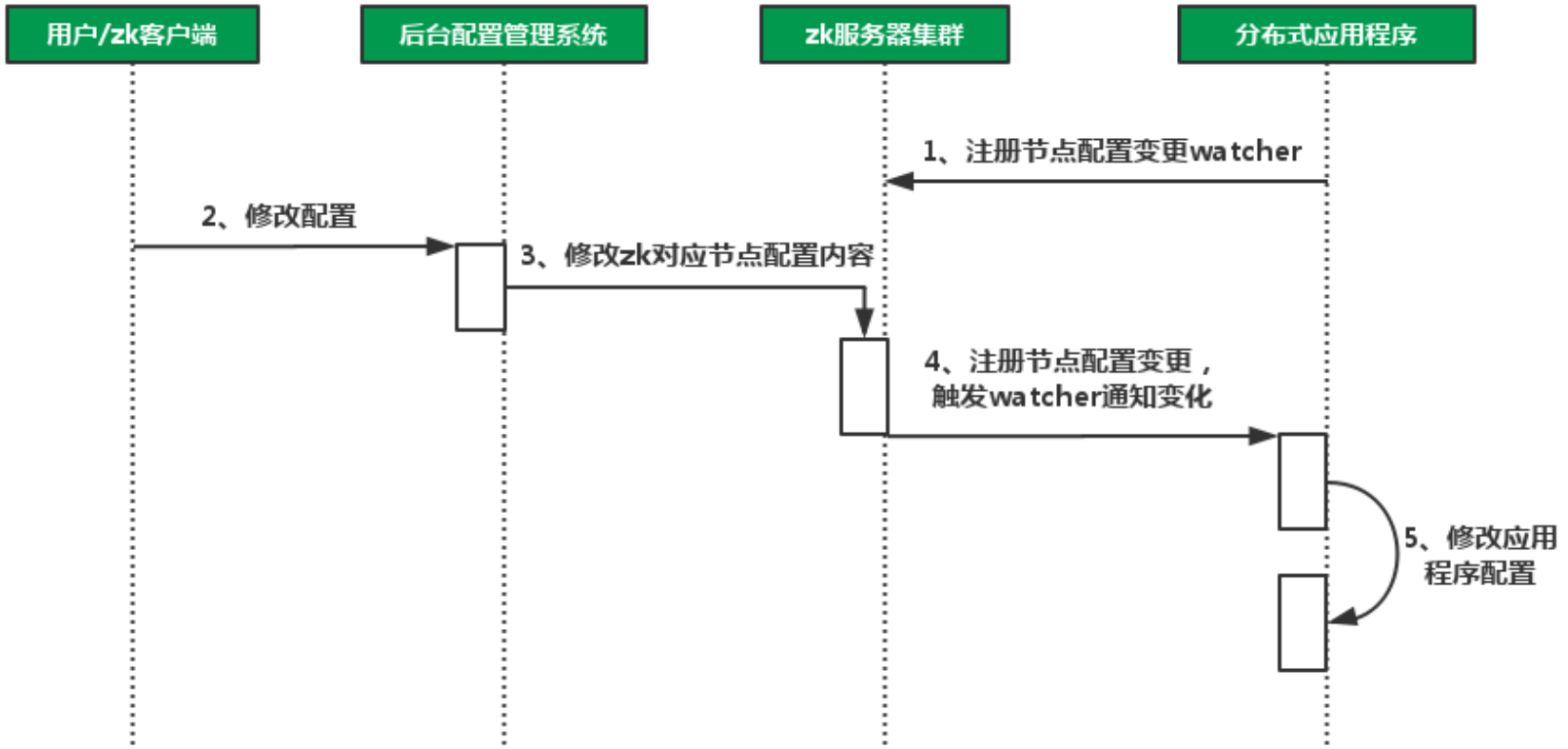
# ZooKeeper解决了什么问题？

zookeeper会维护一个目录结点树，如下图：



# ZooKeeper解决了什么问题？

每个节点znode可以被监控，包括监控某个目录中存储的数据变化，子目录节点的变化，一旦变化可以通知设置监控的客户端。如下图：



PART 3

# 小结

Q & A

THANKS!

# JS大法！一统江湖！

写在最后！