分布式系统消息

钟国英

简历

1. 2010 中山大学计算机科学与技术；
2. 运营业务研发(UC)：邮箱/社区分享/活动/美图，技术栈全面，有服务器端/前端/windows 应用开发经历；业余做过iOS应用，图片站、小说站；
3. MVC Web 开发框架(UC)，为研发人员提供开发 Web 开发解决方案；
4. 分布式爬虫平台(UC)；
5. UAE PaaS 私有云平台(UC)：从 0 开始搭建 PaaS 云平台，承载 UC 几乎所有 Web 服务；支持 UC 浏览器/阿里文学/神马搜索/支付中心/PP 助手/应用发行的 web 业务。 承载每日 HTTP 请求 200 亿，服务器 1000+台 。
6. 发现系统(UC) ： 流式日志统计平台，实时秒级延迟 。面向研发/运维/产品的实时数据分析、监控。每日处理日志 20TB+，峰值 80 万条每秒。

- web开发方案：FastCGI/路由配置/Session 管理/ 数据库 ORM/缓存管理/模版管理/IOC 注入/AOP，功能接近 Play 或 Ruby On Rails。
- 爬虫：支持自定义语言 DSL、可视化爬虫生命周期管理、结构化数 据存储、可视化 HTML 选择器、代理、HTTPS、anti 防盗链。 支持导航、新闻、视频、小说、应用发行、游戏业务。爬虫进程 100+、10 亿+ 页面抓取。
- 云平台：
    - 自动部署、动态扩容、进程状态监控、数据库/缓存服务监控、日志检索;
    - 灰度:地区灰度、参数分流;
    - 图片云:图片实时裁剪/圆角/水印/压缩/拼接/访问，包括 CDN 管理、存储、JS/CSS 拼接混淆、统计等一整套解决方案;
    - 静态资源平台:图片/文件/前端资源管理;
    - 高可用MySQL方案;
    - 分布式key-value数据库fooyun;
    - memcached缓存云;
- 发现系统：
    - PV/UV 统计、性能分析、流量分析、状态码分析、URL 排行、异常 IP 分析;
    - MySQL统计分析:连接数、慢查询、查询数;
    - 缓存:命中率分析;
    - Java 进程 profile 性能分析:ASM 字节码注入 + Java agent，树状调用栈性能分析;
    - 前端性能分析:白屏/DNS/TCP/HTTP;

关于技术追求

- understand things clearly
- explain them well

来自：https://colah.github.io/about.html

我活在世上，无非想要明白些道理，遇见些有趣的事。倘能如我所愿，我的一生就算成功。——王小波

保持内在驱动

- 性能：保持对性能的敏感度和自我要求；
- 寻找偶像：Bill Joy、Linus 、Jeff Dean、高纳德、多隆、David Cutler；

产品质量是在上线时决定的，取决于研发的实力和态度。之后的所有流程，包括测试、规范都只是保险和补丁(强调，并不是说测试不重要，把测试工作前倾，对个人成长和公司都更有价值)，不会产生质的差别。
ACM的经历对我的影响非常大，锻炼了严谨性、测试完备性。一道题目通常有上万个测试样例，既要考虑性能，又要考虑逻辑完备。

质量可以从性能问题入手，性能问题与产品质量息息相关，大多数线上问题由引起：
- 性能敏感包含调用栈逻辑、时序(异步)逻辑的反复复盘；
- 性能问题与扩展性，伸缩性相关，尽量保证每一个小模块的优美、简洁、稳固。无论是横向扩展，纵向分层，用稳固的石头建筑对比流沙有质的区别，不是堆量能够解决的。在性能不足的情况下，扩展/伸缩必然涉及到代码的重新改造，对性能的追求也必然会对加深对程序逻辑的理解；

一位优秀的技术人，不会在早晨醒来后说「今天就是我们性能优化的日子」，就像他不会早上醒来后决定开始呼吸一样。

偶像对人的精神驱动会强烈持久得多，浅了说是解决对职业生涯的内在驱动，深了说就是如何实现人生价值。

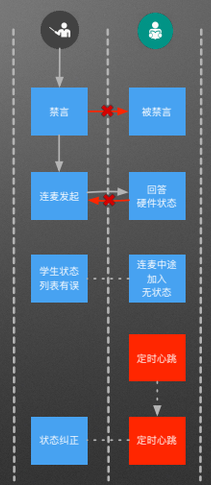
关于业务问题：
A. 弱化「锅」的概念，责任不是某一个个体的，而是全体的，端与端不是孤立的。解决问题为导向，而非找责任方，提倡主动发现与解决问题，互联网产品不可能没有问题，问题本身是不重要的，如何快速改进才是最重要的；
B. 在技术上尽量减少框架的约束，提供这样的平台和场景，让大家可以有足够的自由度、足够的权力去做决定，实现个人能力的提升，获取成就感；
C. 方案不应该是由固定某个人提出，而应该每个人都能提出问题和方案，方案的价值应该贴近业务，切实解决业务问题；

我(们)外放的能量/观点太少了，影响他人太少了，缺少不同方式思维、不同角度的碰撞。单一思维的优点是有限的，我们需要更全局的考虑。我会带头开始做这个事情。
包括后面的性能专题、架构专题、编程技巧、区块链基础；

- 通信问题
- 分布式系统
- 消息系统

通信问题

- 消息丢失
- 无回调确认
- 角色状态不一致
- 缺事件库



回调：经常需要实时回调，根据反馈以后续进行不同的逻辑处理；
状态一致性：没有一致性的处理逻辑，或者逻辑太复杂。比如在线人数的显示(退出通知失败)，答题结束显示；
没有事件库的情况下：时序逻辑很难控制。不够健壮，容错性/鲁棒性太弱。

其实都可以归结为一致性问题。

关于业务问题：
A. 弱化「锅」的概念，责任不是某一个个体的，而是全体的，端与端不是孤立的。解决问题为导向，而非找责任方，提倡主动发现与解决问题，互联网产品不可能没有问题，问题本身是不重要的，如何快速改进才是最重要的；
B. 在技术上尽量减少框架的约束，提供这样的平台和场景，让大家可以有足够的自由度、足够的权力去做决定，实现个人能力的提升，获取成就感；
C. 方案不应该是由固定某个人提出，而应该每个人都能提出问题和方案，方案的价值应该贴近业务，切实解决业务问题；

我(们)外放的能量/观点太少了，影响他人太少了，缺少不同方式思维、不同角度的碰撞。单一思维的优点是有限的，我们需要更全局的考虑。我会带头开始做这个事情。

分布式系统

- 难点
- CAP原理

一般的分布式系统指的是N个服务器的组合，但从用户视角看是一个单一系统。

我们这里会用到的知识，不会像阿里云或亚马逊，排除了存储和运算的复杂性，更偏老师端-学生端，学生端-学生端状态的一致性，会简单很多。

分布式系统 - 难点

- 异构
- 同步和一致性
- 可扩展
- 透明

异构是可以通过基础技术解决的，耗费工作量，不是大问题，但要考虑代价；
同步问题：很难获得一个同步的全局时钟，很多通信都得通过异步消息 + 回调去解决；
一致性，简单来说，就是个体的状态与预期的不一致；
可扩展，当个体越来越多，怎么样保证复杂度较低，**规则简单**干净稳固，不会有很多逻辑补丁，解耦做得比较漂亮；
透明，如何快速发现个体的异常状态；

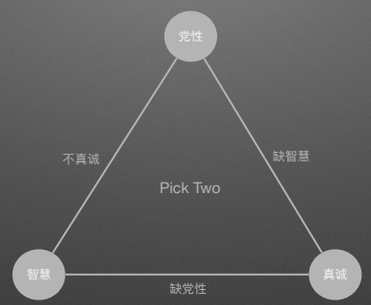
分布式系统

- 难点
- CAP原理

一般的分布式系统指的是N个服务器的组合，但从用户视角看是一个单一系统。

我们这里会用到的知识，不会像阿里云或亚马逊，排除了存储和运算的复杂性，更偏老师端-学生端，学生端-学生端状态的一致性，事务要求简单很多，关系上复杂很多。

分布式系统 - CAP原理

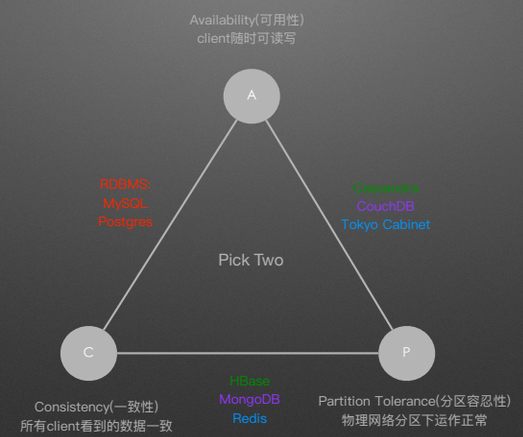


完美的事情只在童话故事里，我们成人的世界只有计算和权衡。

跟时间 - 空间的关系一样的，我们经常用空间去换时间。

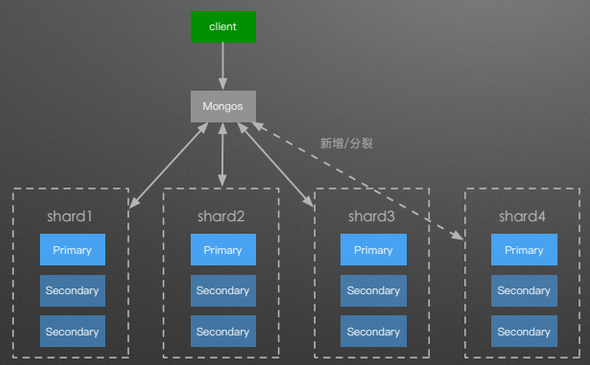
自然界是很奇妙的，所有算法都逃不出这一点，哈希、快排、堆、平衡树(B树、红黑树)、甚至神经网络，都只是在寻找时间与空间的平衡点。

分布式系统 - CAP原理

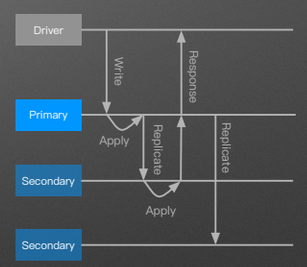


因为我们的物理分区是确定的，必须要通过网络交互，重点又在一致性上，所以选择自然是CP。
而可用性在时间上是可以延期的。
接下来讨论一致性。

分布式系统 - CAP原理 - 举例



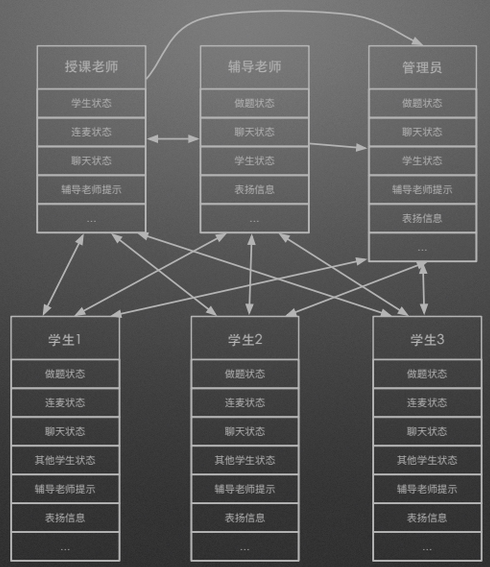
CAP不适用于shard之间，适用于Primary(Master)和Secondary(Slave)间，为什么?



复制过程

- 如果新增一个shard，是否适用CAP原理？分裂呢?
- MySQL的表现会完全不同

分布式系统 - CAP原理 - 举例



其中，学生节点可能上万，要求数据基本保持一致，允许时间上的延迟。

- 好处是不需要支持事务，不需要保持数据的强一致性，但是角色间的数据同步关系比存储复杂的多。
- 任意一个角色的数据逻辑修改可能都会影响另外一个角色，我们急切需要一个解耦利器，把这些关系管理起来，对于每一个角色，只需对接一个规范化的接口即可。解决方案之一就是中心化的pubsub。
- 分布式系统常用的技术：
    - 一致性哈希；
    - vector clock、时钟向量；
    - MVCC
    - Paxos
    - Gossip
    - lease
    - Map-Reduce

分布式系统 - CAP原理 - 一致性模型

- 强一致性(Strong Consitency)：新的数据一旦写入，在任意副本任意时刻都能读到新值；
- 弱一致性(Week Consistency)：不同副本上的值有新有旧，需要client做更多的工作获取最新值；
- 最终一致性(Evantual Consistency)：一旦更新成功，各副本的数据最终将达到一致。

- 文件系统，RDBMS都是强一致性的。
- 比如Dynamo；
- 其中最重要的变体是Read-your-Writes Consistency。特别适用于数据的更新同步，用户的修改马上对自己可见，但是其他用户可以看到他老的版本。Facebook的数据同步就是采用这种原则。

- 通信问题
- 分布式系统
- 消息系统

消息系统

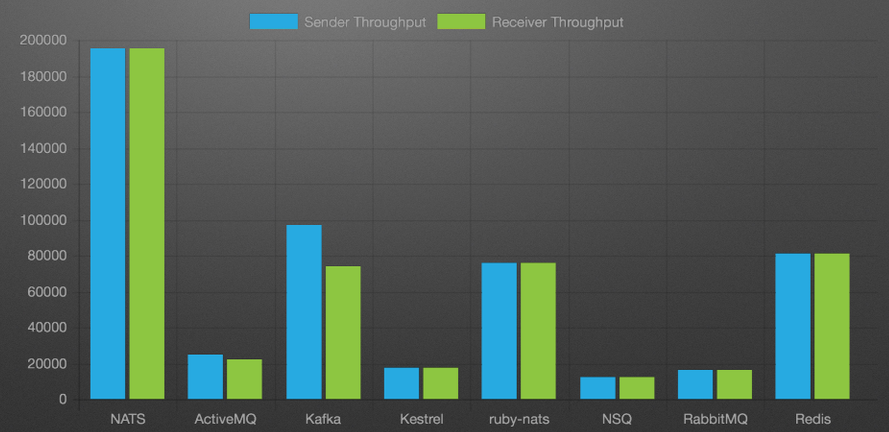
- 消息中间件
- Reactor Pattern

消息系统

- 消息中间件
- Reactor Pattern

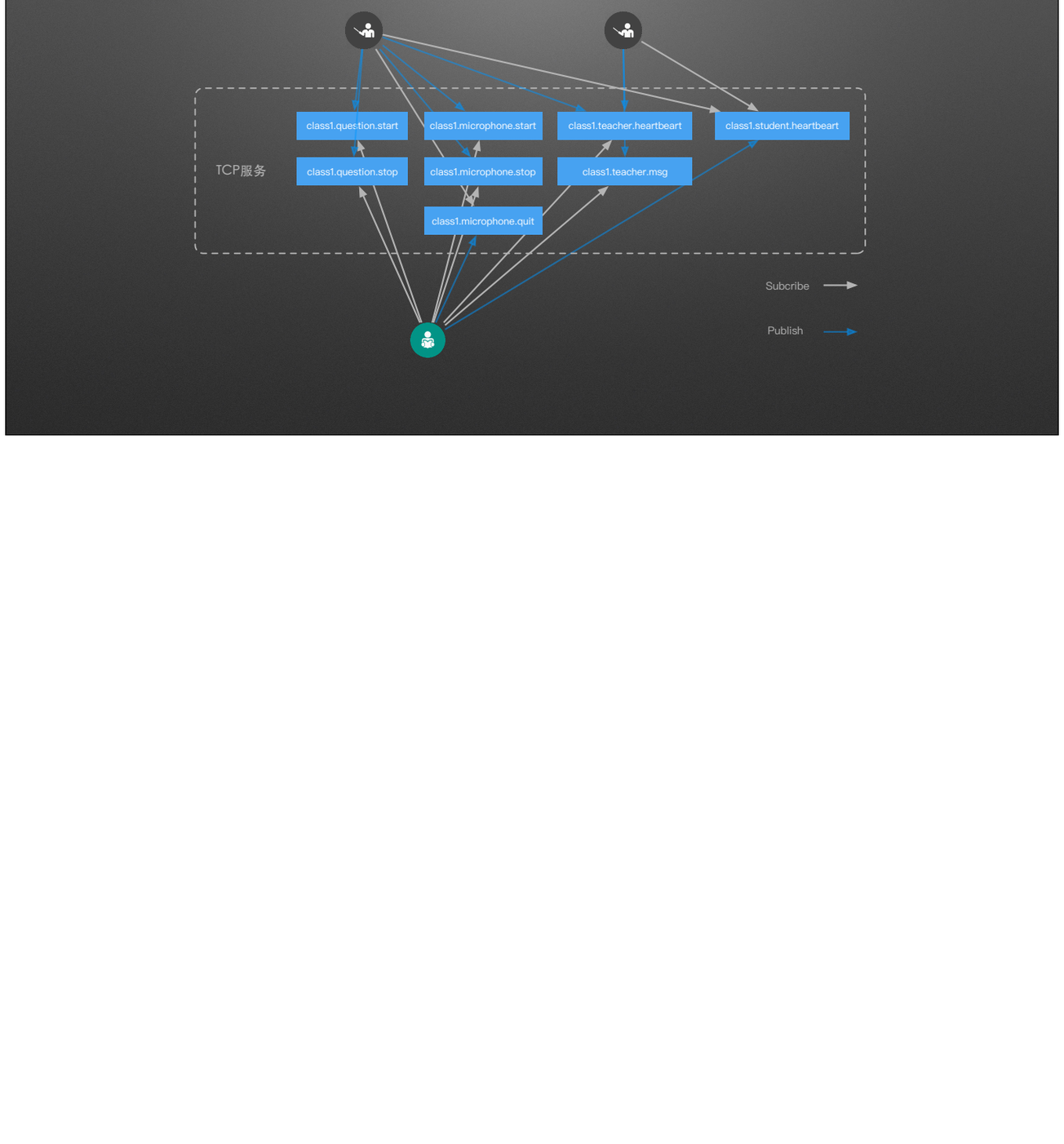
消息系统 - 消息中间件(Message broker)

- ActiveMQ
- Kafka
- Redis
- Kestrel
- RabbitMQ
- NATS



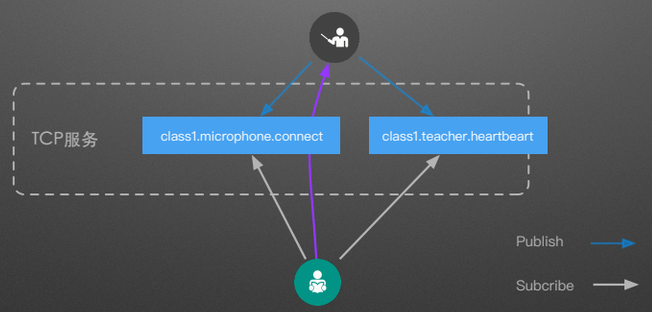
其实我们的选择不多，因为还要看支持的语言。

消息系统 - 消息中间件 - PubSub - 1对多



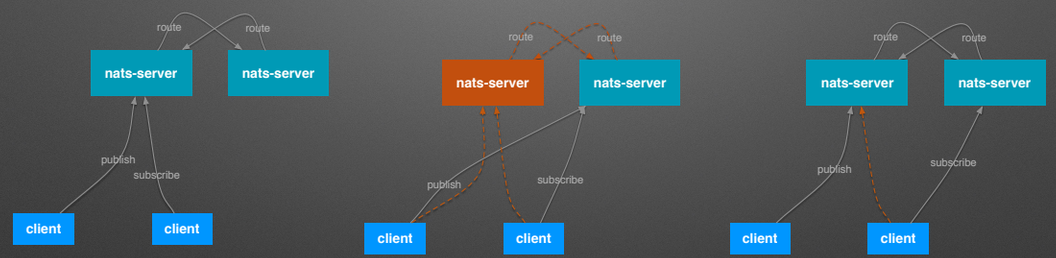
- publish的是任意字符串，一般是JSON

消息系统 - 消息中间件 - PubSub - p2p



- point to point

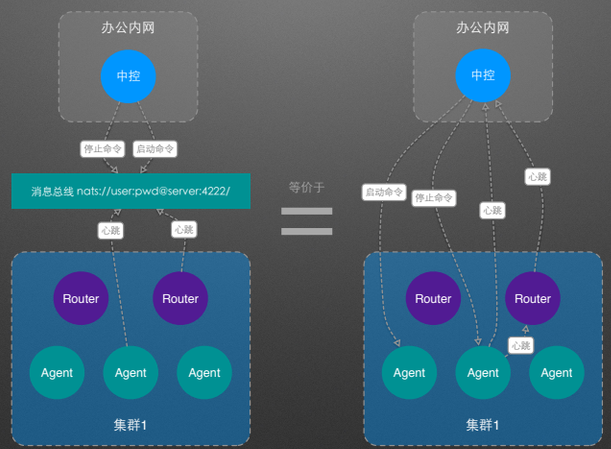
消息系统 - 消息中间件 - PubSub - 单点问题



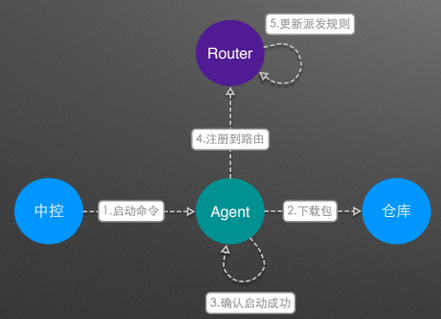
消息系统 - 消息中间件 - PubSub

1. 大部分频道(channel/topic/subject/theme)都是即时通信，比如连麦、下发题目。小部分允许延迟，比如学生心跳，允许老师学生去更新班群列表，实现自发现，在网络出问题的时候，自动纠正教师状态；
2. 支持Request-Reply；
3. 支持队列；
4. 除了固定的频道名，一般也支持通配符PubSub，如*.teacher.heartbeat；
5. 支持报错、加密；
6. 支持集群模式，防单点问题；

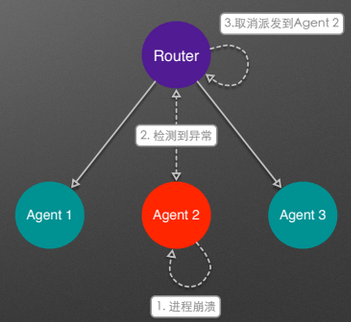
消息系统-消息中间件-UAE例子



消息系统-消息中间件-UAE例子



启动过程



失效转移

failover，或者叫故障转移

消息系统 - 消息中间件 - PubSub

- 回头看前面提出的问题，均可解决；
- 松耦合：各角色只需要定义自己的Pub/Sub的规范即可。
  - 不需要知道每个角色有多少实例，不需要知道哪些实例在线，哪些实例失效。
  - 好的规范设计下，不需要强制启动顺序。比如学生上线request-reply请求老师状态，不会丢失老师已经发起的的连麦/做题状态；
  - 定时Pub/Sub心跳，纠正状态，很容易实现最终一致性；
- 动态发现实例，可寻址通信；
- 下发控制命令非常简单、实时下发配置也同理；
- 透明：监控任意角色的状态也非常简单，订阅即可，且任意客户端都可以发起；
- 可横向扩展；
- 高性能，且有较强的中断恢复的容错能力；

消息系统

- 消息中间件
- Reactor Pattern

反应器模式的应用：ActiveMQ、NATS、Kafka、Redis。
这个设计模式还跟著名的c10k problem性能问题、IO多路复用有关，后面还有性能专题应该会提到。
问题：

- IO很慢，包括网络IO、磁盘IO、内存IO，需要等待，CPU不应该等待，而应该压榨干CPU的能力；
- 如果用多线程处理，线程会变得不可控制：线程频繁创建和销毁的开销，内存空间的占用，线程间数据传输；
- 阻塞式的编程很方便，符合人类的时序逻辑，但是很难有效利用资源，很难扩展；

Reactor Pattern是事件驱动的编程：

- 单线程；
- IoC；
- blocking event loop + dispatcher + handler

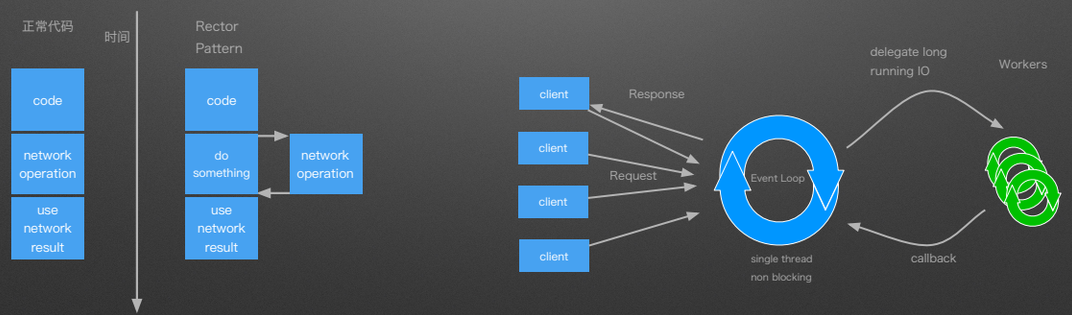
消息系统 - Reactor Pattern

- libevent(C)
- Netty(Java)
- Node.js(JS)
- Twisted(Python)
- EventMachine(Ruby)

- 非阻塞IO，高并发，事件驱动；
- 降低多IO编程复杂度；

PS: Node的几个特点，单线程、异步、非阻塞都跟Reactor Pattern有关。

消息系统 - Reactor Pattern



对Node熟悉的同学应该非常熟悉左侧这个模式了。
右侧的实际模块比这里要稍微复杂一点：Rector、Event Demultiplexer(dispatcher)、Event Handler，可以参考libevent的实现。

具体的高并发IO多路复用机制：

- select
- poll
- epoll

再回头看问题：

- IO很慢，包括网络IO、磁盘IO、内存IO，需要等待，CPU不应该等待，而应该压榨干CPU的能力；
- 如果用多线程处理，线程会变得不可控制：线程频繁创建和销毁的开销，内存空间的占用，线程间数据传输；
- 阻塞式的编程很方便，符合人类的时序逻辑，但是很难有效利用资源，很难扩展；