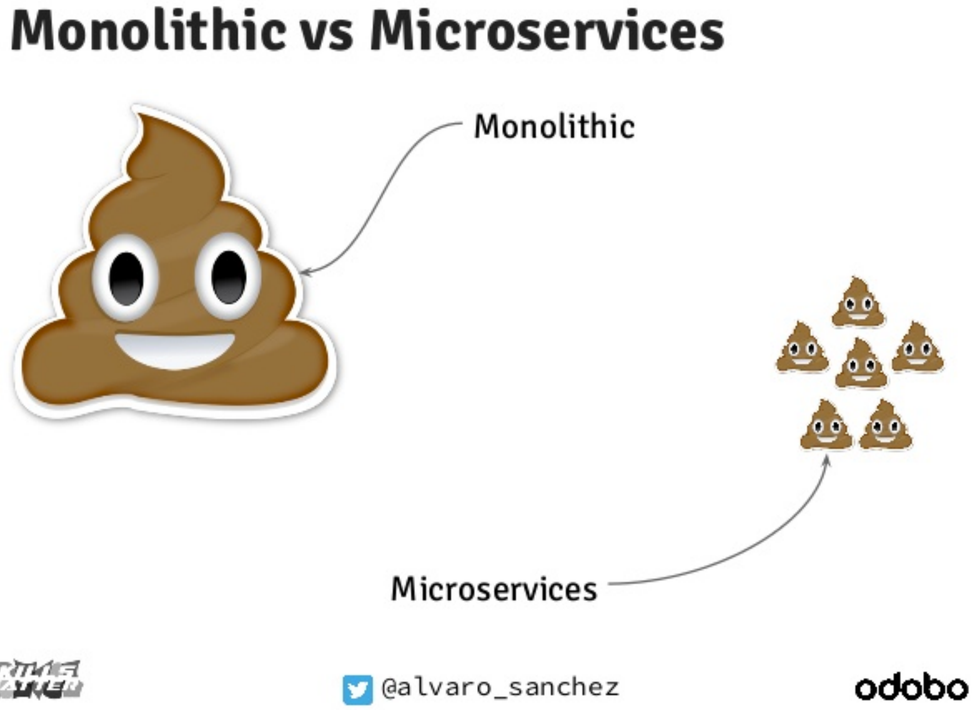
微服务架构中的消息队列

xh4n3@Coding

为什么大街小巷都在谈微服务？Why microservice?

因为：

1. 社会在不断向前发展
2. 业务需求迅速增加
3. 进行容量扩展
    - 垂直扩展（Scale Up）成本高昂，可预见的上限
    - 水平拓展（Scale Out）成本较为低廉
4. 水平拓展时
    - 单体应用（Monolithic）紧耦合，扩展的时候效率不按比例，较为粗犷
    - 微服务（Microservice）松耦合，细粒度，高效率



“劳动生产力最大的进步，以及劳动在任何地方的运用中体现的大部分的技能、熟练度和判断力似乎都是分工的结果。”

解耦

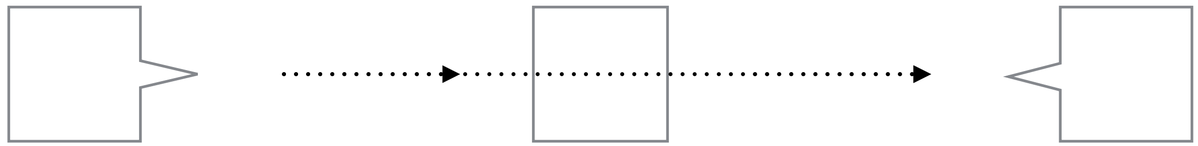
-《国富论》

解耦 Decoupling

解耦的过程：

- 是治疗患有重度洁癖强迫症的架构师的过程；
- 是将披萨分成一片一片的过程，却又享受地看着香浓芝士在其间相连的过程；
- 是将业务一步一步剥离开来，或把不那么核心的业务过程抛到外围的过程。

消息通信 Message Communication



潜在问题：

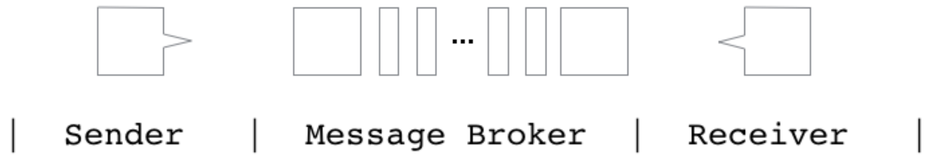
- Caller 发出消息后被阻塞
- Caller 和 Callee 必须同时在线
- 只适用于单播通信 unicast

“Distributed systems are all about trade-offs.”

- @tyler_treat

现实世界的抉择 Make your choice

- Sender 需要知道准确运行结果 -> 强一致性 -> 同步
  - 通过事务保证数据准确性
  - 分布式架构中时效性低下



- Sender 不在乎运行结果 -> 弱一致性/最终一致性 -> 异步
  - 最终一致性 - "Everything will be okay."
  - 一致性程度由框架和业务代码决定

同步和异步通信 Synchronous vs. Asynchronous

同步形式的通信，即各种形态的 Remote Procedure Call：

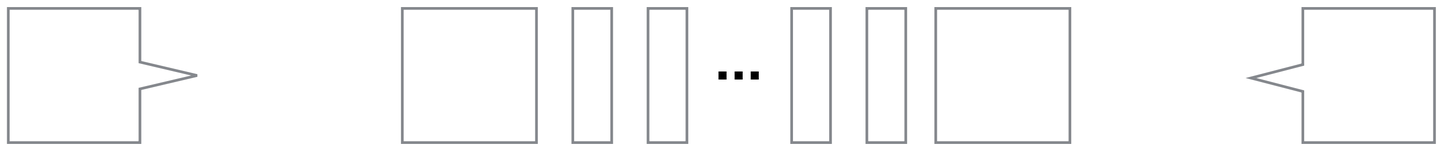
- Google's gRPC
- Apache Thrift
- Alibaba's Dubbo
- Plain RESTFul API …

异步形式的通信：

- **消息队列 Message Queue**
- 事件总线 EventBus
- 其它形式的消息的队列：数据库，文件等

消息队列架构 Message Queue Topology - 1

- 三类角色设定



Sender	Message Broker	Receiver
Producer	Message Broker	Consumer
Publisher	Message Broker	Subscriber

消息队列架构 Message Queue Topology - 2

广义地来说以下都是消息队列的形式：

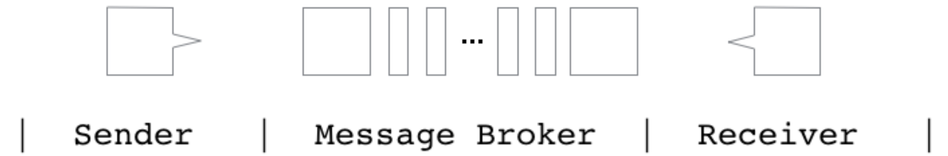
- HTML5 Web Worker
    - 当主线程调用 postMessage(msg) 时，WebKit 会把 msg 存入内存中的临时队列，当 worker 线程准备完毕时，再从队列中取得消息开始处理。
- 数据库中的一张表
    - A 服务把注册完成的用户手机号码填入数据库中的一张表，B 服务每隔几分钟从这张表中取出一部分手机号码进行短信推送。

消息队列特性 Message Queue Feature

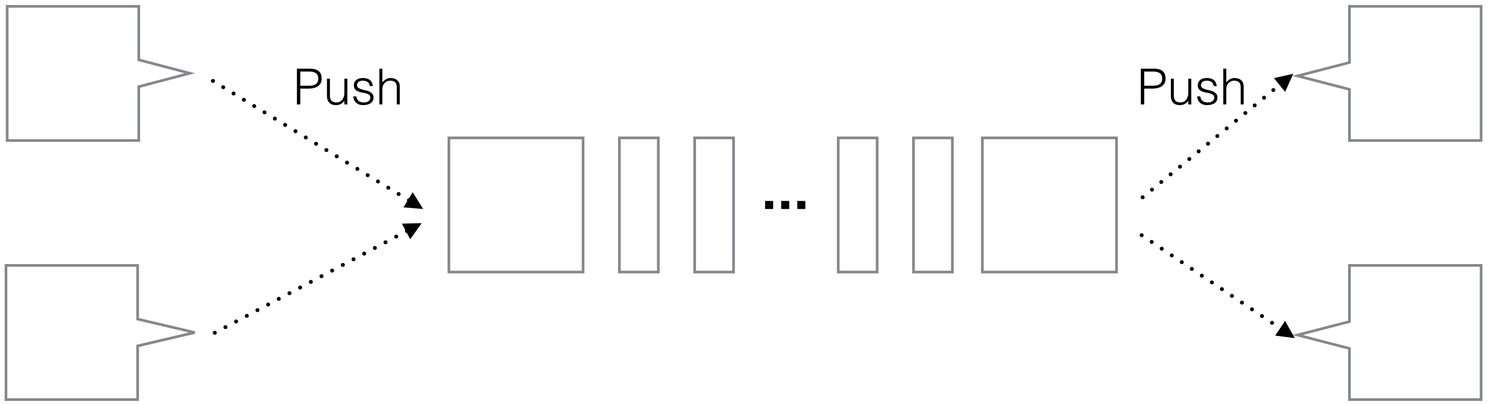
- 一致性 - 消息确认 Acknowledgements
- 可靠性 - 推还是拉 Push and Pull
- 有序性 - 顺序消息 Message Order

消息确认 Acknowledgements

- Broker 告诉 Sender 接收完/处理完消息
- Receiver 告诉 Broker 接收完/处理完消息
- 保证了 at-least-once delivery
  - 狂发消息给接收者，直到收到消息确认
  - 消息可能重复接收，业务操作需要保证是幂等操作 Idempotent Operation
  - 如果做不到幂等操作，需要做去重过滤
- 如果没有消息确认，只能保证 at-most-once delivery



推还是拉 Push and Pull - 1

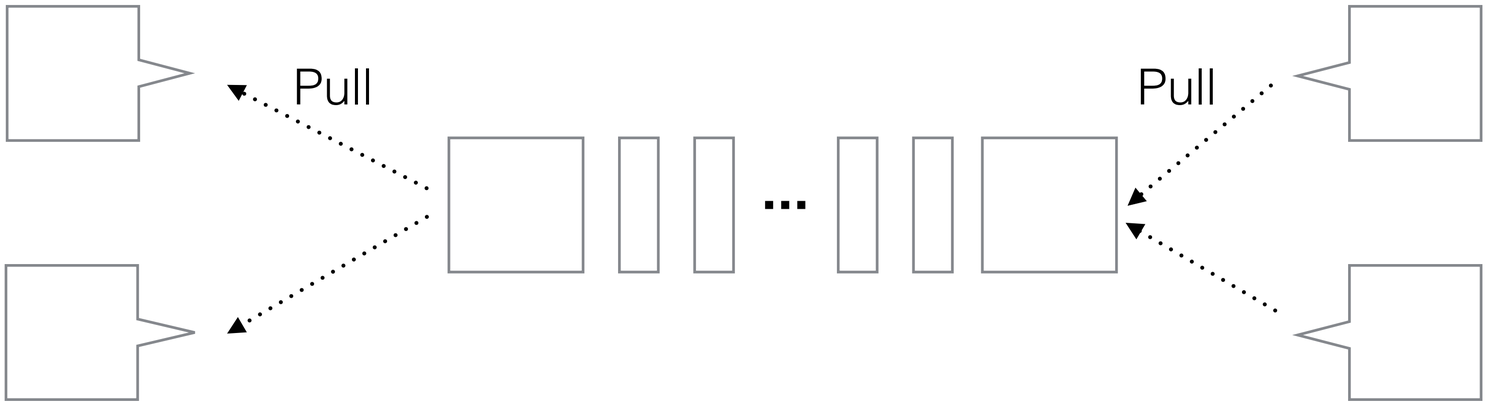


推 Push 模式：

- Broker 一收到消息就会给 Receiver 推送

- 如果 Receiver 处理消息速度慢，而消息持续产生，发送给 Receiver 时，Receiver 并不能处理消息，会返回 reject 或者 error。

推还是拉 Push and Pull - 2



拉 Pull 模式：

- Receiver 定时主动向 Broker 请求消息
- 可以按需地拉取数据，类似于 Web 中的轮询，很多请求是白费功夫

推还是拉 Push and Pull - 3

长轮询 Long-Polling，Receiver 主动请求消息，Broker 只在有消息时或超时后返回：

- 服务器保持连接会消耗一定资源，但接近于实时
- 需要设置一个超时重试
- Kafka/RocketMQ

顺序消息 Message Order

- 为了保证业务的一致性和实现事务，消息系统难点之一在于消息有序性
- 只有前一个消息得到消息确认后才发送后一个消息
- 如果只有单线程单实例，就不会有顺序错乱问题
- 但是，效率低下以及带来单实例部署导致低可用性问题

RabbitMQ: Messaging that just works

- Advanced Message Queuing Protocol (AMQP) 实现
  - 一个开放的高级消息队列协议，为通用消息队列架构提供通用的构建工具
  - 满足协议的不同客户端可以互相通信，此外还有 Apache ActiveMQ
- 支持多种模式
  - Work queues：分配任务，通常一个 Sender 发布消息，多个 Receiver 处理消息
  - Pub/Sub：引入消息中转 Exchange，Exchange 中采用 fanout 规则将消息转发给每一个 Receiver
  - Routing：采用 direct 规则，发布消息是可指定 Routing Key，供 Exchange 转发到对应的队列中
  - Topics：采用 topic 规则，发布消息时指定 Topic，以通配符的方式将消息转发到不同队列中
  - RPC：同步模式，阻塞直到获得结果

Apache Kafka: Distributed Message Queue

- 每个 Topic 的内容被根据一定算法被分配到不同分区 Partition，每个 Partition 内部消息保证有序，并发数量与分区数成正比
- 主要解决单点故障问题，采用分布式架构，由 ZooKeeper 提供高可用性
  - 采用 Active/Passive 模式，每个分区 Partition 有多个备份 Replicas，其中一份为 Leader 承载消息读写，由 ZooKeeper 选举产生，其余的是 Follower，只同步消息。
  - 看起来是单实例，但是这个单实例故障后会有新的 Leader 接管
- 将一定时间内的消息持久化到硬盘中，支持重放 Replay

ZeroMQ: Brokerless Message Queue - 1

- 歌颂了 Broker 的优点
  - Sender 和 Receiver 无关性：A 不需要知道 B 所在，可以通过 Broker 来寻找 B 所在
  - Sender 和 Receiver 生命周期不需要重叠：A 发出消息后，就可以关机
  - 一定程度上减少业务故障：A 发出消息后，宕机，但消息还在，B 业务正常处理
- 吐槽了 Broker 的缺点
  - 父业务需要调用子业务，甚至孙子业务时，考虑到请求和返回是双向的，内网网络链接数量非常大
  - 即使采用流水线形式（Pipelined Fashion）的业务调用，由于 Broker 的存在，也好不到哪里去
  - 由于 Broker 是中心化节点一样的存在，在业务量较大时，Broker 成为整个架构的瓶颈

ZeroMQ: Brokerless Message Queue - 2

- 提出了神思路：去掉 Broker
  - 用目录服务（Directory Service）代替 Broker，类似服务注册与发现模块
  - Sender 用目录服务找到 Receiver，进行直接通信
- 可选地加入了 Broker 的优点：
  - 在中心化的目录服务基础上，加上分布式的 Broker
  - 顺便又把中心化的目录服务做成分布式的，避免了单点故障
- 总结：
  - Brokerless 模式下性能超高
  - 高度可定制化，可以搭配多种模式使用
  - 信息默认存在内存中

其他消息队列 Other Message Queues

- RocketMQ: Alibaba's MQ, also aliyun ONS.
  - 部分性能比 Kafka 更优，见文档
- NSQ: A realtime distributed messaging platform
  - 消息默认不持久化
  - 保证 at-least-once delivery
  - 消息无序

参考资料 References

分布式RPC框架性能大比拼

http://colobu.com/2016/09/05/benchmarks-of-popular-rpc-frameworks/

Reliability Guide

https://www.rabbitmq.com/reliability.html

Message Delivery Reliability

http://doc.akka.io/docs/akka/snapshot/general/message-delivery-reliability.html

You Cannot Have Exactly-Once Delivery

http://bravenewgeek.com/you-cannot-have-exactly-once-delivery/

Kafka 0.10.0 Documentation

http://kafka.apache.org/documentation.html#design

Kafka Partitions Problem

https://github.com/alibaba/RocketMQ/blob/master/wiki/kafka_partitions_problem.md

ZeroMQ Frequently Asked Questions

http://zeromq.org/area:faq

Message Queue Shootout!

http://mikehadlow.blogspot.com/2011/04/message-queue-shootout.html

Broker vs. Brokerless

http://zeromq.org/whitepapers:brokerless

RocketMQ vs. Kafka

https://github.com/alibaba/RocketMQ/blob/master/wiki/rmq_vs_kafka.md

分布式开放消息系统(RocketMQ)的原理与实践

http://www.jianshu.com/p/453c6e7ff81c

NSQ FEATURES & GUARANTEES

http://nsq.io/overview/features_and_guarantees.html

Dissecting Message Queues

http://bravenewgeek.com/dissecting-message-queues/

Top 10 Uses For A Message Queue

https://www.iron.io/top-10-uses-for-message-queue/