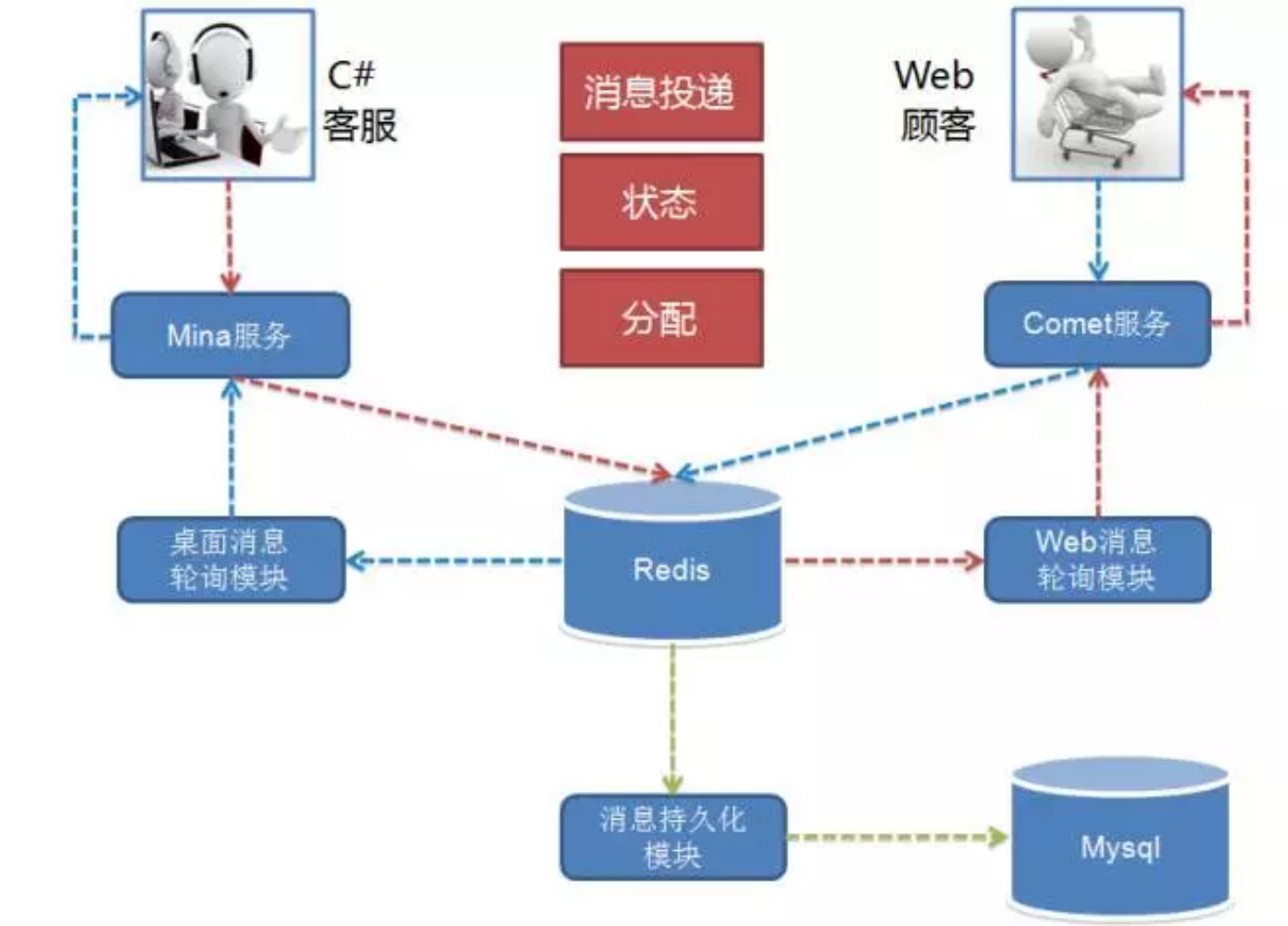
京东咚咚架构演进

原创：mindwind 瞬息之间 2015-12-03

咚咚是什么？咚咚之于京东相当于旺旺之于淘宝，它们都是服务于买家和卖家的沟通。自从京东开始为第三方卖家提供入驻平台服务后，咚咚也就随之诞生了。 我们首先看看它诞生之初是什么样的。

1.0 诞生（2010 - 2011)

为了业务的快速上线，1.0 版本的技术架构实现是非常直接且简单粗暴的。 如何简单粗暴法？请看架构图，如下。



1.0 的功能十分简单，实现了一个 IM 的基本功能，接入、互通消息和状态。 另外还有客服功能，就是顾客接入咨询时的客服分配，按轮询方式把顾客分配给在线的客服接待。 用开源 Mina 框架实现了 TCP 的长连接接入，用 Tomcat Comet 机制实现了 HTTP 的长轮询服务。 而消息投递的实现是一端发送的消息临时存放在 Redis 中，另一端拉取的生产消费模型。

这个模型的做法导致需要以一种高频率的方式来轮询 Redis 遍历属于自己连接的关联会话消息。 这个模型很简单，简单包括多个层面的意思：理解起来简单；开发起来简单；部署起来也简单。 只需要一个 Tomcat 应用依赖一个共享的 Redis，简单的实现核心业务功能，并支持业务快速上线。

但这个简单的模型也有些严重的缺陷，主要是效率和扩展问题。 轮询的频率间隔大小基本决定了消息的延时，轮询越快延时越低，但轮询越快消耗也越高。 这个模型实际上是一个高功耗低效能的模型，因为不活跃的连接在那做高频率的无意义轮询。 高频有多高呢，基本在 100 ms 以内，你不能让轮询太慢，比如超过 2 秒轮一次，人就会在聊天过程中感受到明显的会话延迟。 随着在线人数增加，轮询的耗时也线性增长，因此这个模型导致了扩展能力和承载能力都不好，一定会随着在线人数的增长碰到性能瓶颈。

1.0 的时代背景正是京东技术平台从 .NET 向 Java 转型的年代，我也正是在这期间加入京东并参与了京东主站技术转型架构升级的过程。 之后开始接手了京东咚咚，并持续完善这个产品，进行了三次技术架构演进。

2.0 成长（2012）

我们刚接手时 1.0 已在线上运行并支持京东 POP（开放平台）业务，之后京东打算组建自营在线客服团队并落地在成都。 不管是自营还是 POP 客服咨询业务当时都起步不久，1.0 架构中的性能和效率缺陷问题还没有达到引爆的业务量级。 而自营客服当时还处于起步阶段，客服人数不足，服务能力不够，顾客咨询量远远超过客服的服务能力。 超出服务能力的顾客咨询，当时我们的系统统一返回提示客服繁忙，请稍后咨询。 这种状况导致高峰期大量顾客无论怎么刷新请求，都很可能无法接入客服，体验很差。 所以 2.0 重点放在了业务功能体验的提升上，如下图所示。

针对无法及时提供服务的顾客，可以排队或者留言。 针对纯文字沟通，提供了文件和图片等更丰富的表达方式。 另外支持了客服转接和快捷回复等方式来提升客服的接待效率。 总之，整个 2.0 就是围绕提升客服效率和用户体验。 而我们担心的效率问题在 2.0 高速发展业务的时期还没有出现，但业务量正在逐渐积累，我们知道它快要爆了。 到 2012 年末，度过双十一后开始了 3.0 的一次重大架构升级。

3.0 爆发（2013 - 2014）

经历了 2.0 时代一整年的业务高速发展，实际上代码规模膨胀的很快。 与代码一块膨胀的还有团队，从最初的 4 个人到近 30 人。 团队大了后，一个系统多人开发，开发人员层次不一，规范难统一，系统模块耦合重，改动沟通和依赖多，上线风险难以控制。 一个单独 tomcat 应用多实例部署模型终于走到头了，这个版本架构升级的主题就是服务化。

服务化的第一个问题如何把一个大的应用系统切分成子服务系统。 当时的背景是京东的部署还在半自动化年代，自动部署系统刚起步，子服务系统若按业务划分太细太多，部署工作量很大且难管理。 所以当时我们不是按业务功能分区服务的，而是按业务重要性级别划分了 0、1、2 三个级别不同的子业务服务系统。 另外就是独立了一组接入服务，针对不同渠道和通信方式的接入端，见下图。

更细化的应用服务和架构分层方式可见下图。

这次大的架构升级，主要考虑了三个方面：稳定性、效率和容量。 做了下面这些事情：

1. 业务分级、核心、非核心业务隔离
2. 多机房部署，流量分流、容灾冗余、峰值应对冗余
3. 读库多源，失败自动转移
4. 写库主备，短暂有损服务容忍下的快速切换

5. 外部接口，失败转移或快速断路
6. Redis 主备，失败转移
7. 大表迁移，MongoDB 取代 MySQL 存储消息记录
8. 改进消息投递模型

前 6 条基本属于考虑系统稳定性、可用性方面的改进升级。 这一块属于陆续迭代完成的，承载很多失败转移的配置和控制功能在上面图中是由管控中心提供的。 第 7 条主要是随着业务量的上升，单日消息量越来越大后，使用了 MongoDB 来单独存储量最大的聊天记录。 第 8 条是针对 1.0 版本消息轮询效率低的改进，改进后的投递方式如下图所示：

不再是轮询了，而是让终端每次建立连接后注册接入点位置，消息投递前定位连接所在接入点位置再推送过去。 这样投递效率就是恒定的了，而且很容易扩展，在线人数越多则连接数越多，只需要扩展接入点即可。 其实，这个模型依然还有些小问题，主要出在离线消息的处理上，可以先思考下，我们最后再讲。

3.0 经过了两年的迭代式升级，单纯从业务量上来说还可以继续支撑很长时间的增长。 但实际上到 2014 年底我们面对的不再是业务量的问题，而是业务模式的变化。 这直接导致了一个全新时代的到来。

4.0 涅槃（2015 至今）

2014 年京东的组织架构发生了很大变化，从一个公司变成了一个集团，下设多个子公司。 原来的商城成为了其中一个子公司，新成立的子公司包括京东金融、京东智能、京东到家、拍拍、海外事业部等。 各自业务范围不同，业务模式也不同，但不管什么业务

总是需要客服服务。 如何复用原来为商城量身订做的咚咚客服系统并支持其他子公司业务快速接入成为我们新的课题。

最早要求接入的是拍拍网，它是从腾讯收购的，所以是完全不同的账户和订单交易体系。由于时间紧迫，我们把为商城订做的部分剥离，基于 3.0 架构对接拍拍又单独订做了一套，并独立部署，像下面这样。

虽然在业务要求的时间点前完成了上线，但这样做也带来了明显的问题：

1. 复制工程，定制业务开发，多套源码维护成本高
2. 独立部署，至少双机房主备外加一个灰度集群，资源浪费大

以前我们都是面向业务去架构系统，如今新的业务变化形势下我们开始考虑面向平台去架构，在统一平台上跑多套业务，统一源码，统一部署，统一维护。 把业务服务继续拆分，剥离出最基础的 IM 服务，IM 通用服务，客服通用服务，而针对不同的业务特殊需求做最小化的定制服务开发。 部署方式则以平台形式部署，不同的业务方的服务跑在同一个平台上，但数据互相隔离。 服务继续被拆分的更微粒化，形成了一组服务矩阵（见下图）。

而部署方式，只需要在双机房建立两套对等集群，并另外建一个较小的灰度发布集群即可，所有不同业务都运行在统一平台集群上，如下图。

更细粒度的服务意味着每个服务的开发更简单，代码量更小，依赖更少，隔离稳定性更高。 但更细粒度的服务也意味着更繁琐的运维监控管理，直到今年公司内部弹性私有云、缓存云、消息队列、部署、监控、日志等基础系统日趋完善， 使得实施这类细粒度划分的微服务架构成为可能，运维成本可控。 而从当初 1.0 的 1 种应用进程，到 3.0 的 6、7 种应用进程，再到 4.0 的 50+ 更细粒度的不同种应用进程。 每种进程再根据承载业务流量不同分配不同的实例数，真正的实例进程数会过千。 为了更好的监控和管理这些进程，为此专门定制了一套面向服务的运维管理系统，见下图。

统一服务运维提供了实用的内部工具和库来帮助开发更健壮的微服务。 包括中心配置管理，流量埋点监控，数据库和缓存访问，运行时隔离，如下图所示是一个运行隔离的图示：

细粒度的微服务做到了进程间隔离，严格的开发规范和工具库帮助实现了异步消息和异步 HTTP 来避免多个跨进程的同步长调用链。 进程内部通过切面方式引入了服务增强容器 Armor 来隔离线程， 并支持进程内的单独业务降级和同步转异步化执行。而所有这些工具和库服务都是为了两个目标：

1. 让服务进程运行时状态可见
2. 让服务进程运行时状态可被管理和改变

最后我们回到前文留下的一个悬念，就是关于消息投递模型的缺陷。 一开始我们在接入层检测到终端连接断开后，消息无法投递，再将消息缓存下来，等终端重连接上来再拉取离线消息。 这个模型在移动时代表现的很不好，因为移动网络的不稳定性，导致经常断链后重连。 而准确的检测网络连接断开是依赖一个网络超时的，导致检测可能不准确，引发消息假投递成功。 新的模型如下图所示，它不再依赖准确的网络连接检测，投递前待确认消息 id 被缓存，而消息体被持久存储。 等到终端接收确认返回后，该消息才算投妥，未确认的消息 id 再重新登陆后或重连接后作为离线消息推送。 这个模型不会产生消息假投妥导致的丢失，但可能导致消息重复，只需由客户终端按消息 id 去重即可。

京东咚咚诞生之初正是京东技术转型到 Java 之时，经历这些年的发展，取得了很大的进步。 从草根走向专业，从弱小走向规模，从分散走向统一，从杂乱走向规范。 本文主要重心放在了几年来咚咚架构演进的过程，技术架构单独拿出来看我认为没有绝对的好与不好， 技术架构总是要放在彼时的背景下来看，要考虑业务的时效价值、团队的规模和能力、环境基础设施等等方面。 架构演进的生命周期适时匹配好业务的生命周期，才可能发挥最好的效果。