作业中的资料来源：

<https://github.com/hollischuang/Architecture-Evolution>

和

<https://blog.csdn.net/JF_1994/article/details/53870534>

并阅读了一部分《架构之美》

知乎：

知乎在规模上是仅次于百度贴吧和豆瓣的中文互联网最大的UGC(用户生成内容) 社区。知乎创业三年来，从0 开始，到现在已经有了100 多台服务器。目前知乎的注册用户超过了1100 万，每个月有超过8000 万人使用；网站每个月的PV 超过2.2 亿，差不多每秒钟的动态请求超过2500。

在 ArchSummit 北京 2014 大会上，知乎联合创始人兼 CTO 李申申带来了知乎创业三年多来的首次全面技术分享

初期架构选型：

知乎的主力开发语言是Python。因为Python 简单且强大，能够快速上手，开发效率高，而且社区活跃，团队成员也比较喜欢。

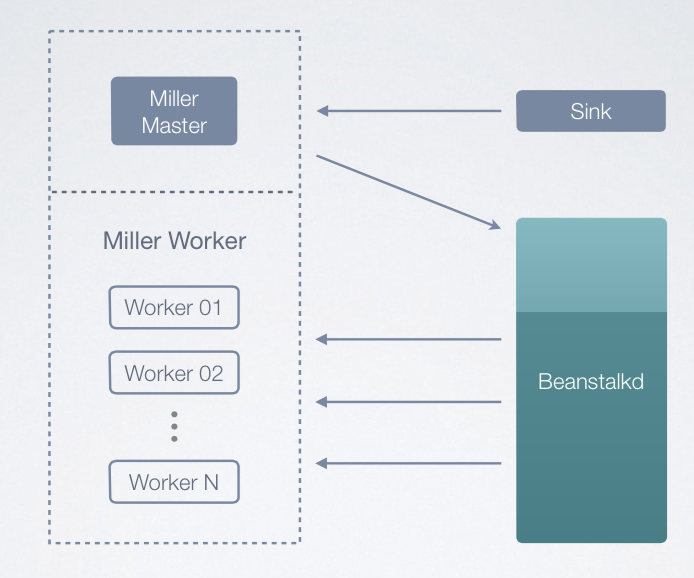
知乎使用的是 Tornado 框架。因为它支持异步，很适合做实时 Comet 应用，而且简单轻量，学习成本低，再就是有 FriendFeed 的成熟案例，Facebook 的社区支持。知乎的产品有个特性，就是希望跟浏览器端建立一个长连接，便于实时推送 Feed 和通知，所以 Tornado 比较合适。

Tornado: Tornado是一个Python Web框架和异步网络库，最初是由FriendFeed开发的。通过使用非阻塞网络I / O，Tornado可以扩展到成千上万的开放连接，使其非常适合 长时间轮询， WebSocket和其他需要与每个用户建立长期连接的应用程序。

## 事件驱动的架构

知乎这个产品有一个特点，最早在添加一个答案后，后续的操作其实只有更新通知、更新动态。但是随着整个功能的增加，又多出了一些更新索引、更新计数、内容审查等操作，后续操作五花八门。如果按照传统方式，维护逻辑会越来越庞大，维护性也会非常差。这种场景很适合事件驱动方式，所以开发团队对整个架构做了调整，做了事件驱动的架构。

这时首先需要的是一个消息队列，它应该可以获取到各种各样的事件，而且对一致性有很高的要求。针对这个需求，知乎开发了一个叫 Sink 的小工具。它拿到消息后，先做本地的保存、持久化，然后再把消息分发出去。如果那台机器挂掉了，重启时可以完整恢复，确保消息不会丢失。然后它通过 Miller 开发框架，把消息放到任务队列。Sink 更像是串行消息订阅服务，但任务需要并行化处理， Beanstalkd 就派上了用场，由其对任务进行全周期管理。架构如下图所示：



举例而言，如果现在有用户回答了问题，首先系统会把问题写到 MySQL 里面，把消息塞到 Sink，然后把问题返回给用户。Sink 通过 Miller 把任务发给 Beanstalkd，Worker 自己可以找到任务并处理。

最开始上线时，每秒钟有 10 个消息，然后有 70 个任务产生。现在每秒钟有 100 个事件，有 1500 个任务产生，就是通过现在的事件驱动架构支撑的。

## 面向服务的架构（SOA）

随着知乎的功能越来越庞杂，整个系统也越来越大。知乎是怎么做的服务化呢？

首先需要一个最基本的 RPC 框架，RPC 框架也经历了好几版演进。

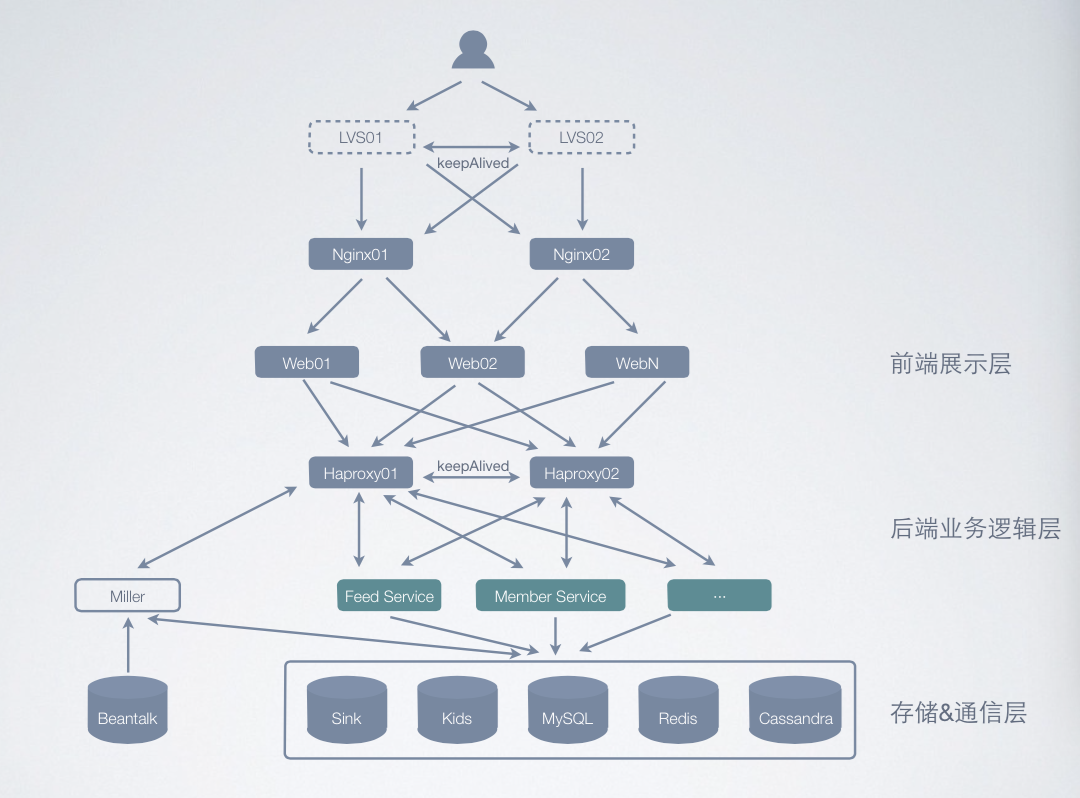
第一版是 Wish，它是一个严格定义序列化的模型。传输层用到了 STP，这是自己写的很简单的传输协议，跑在 TCP 上。一开始用的还不错，因为一开始只写了一两个服务。但是随着服务增多，一些问题开始出现，首先是 ProtocolBuffer 会 生成一些描述代码，很冗长，放到整个库里显得很丑陋。另外严格的定义使其不便使用。这时有位工程师开发了新的 RPC 框架——Snow。它使用简单的 JSON 做数据序列化。但是松散的数据定义面对的问题是，比如说服务要去升级，要改写数据结构，很难知道有哪几个服务在使用，也很难通知它们，往往错误就发生了。于是又出了第三个 RPC 框架，写 RPC 框架的工程师，希望结合前面两个框架的特点，首先保持 Snow 简单，其次需要相对严格的序列化协议。这一版本引入了 Apache Avro。同时加入了特别的机制，在传输层和序列化协议这一层都做成了可插拔的方式，既可以用 JSON，也可以用 Avro，传输层可以用 STP，也可以用二进制协议。

再就是搭了一个服务注册发现，只需要简单的定义服务的名字就可以找到服务在哪台机器上。同时，知乎也有相应的调优的工具，基于 Zipkin 开发了自己的 Tracing 系统。

按照调用关系，知乎的服务分成了 3 层：**聚合层、内容层和基础层**。按属性又可以分成 3 类：数据服务、逻辑服务和通道服务。数据服务主要是一些要做特殊数据类型的存储，比如图片服务。逻辑服务更多的是 CPU 密集、计算密集的操作，比如答案格式的定义、解析等。通道服务的特点是没有存储，更多是做一个转发，比如说 Sink。

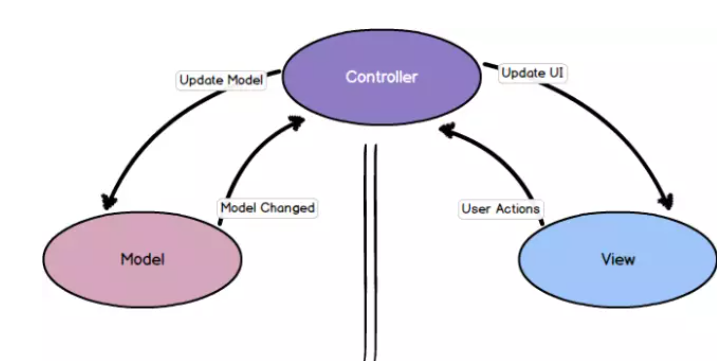


这是引入服务化之后整体的架构。



## 饿了么移动APP的架构演进:

在饿了么业务发展的早期，移动APP经历了从无到有的阶段。为了快速上线抢占市场，传统移动APP开发的MVC架构成了“短平快”思路的首选：



这种架构因简单清晰，容易开发而被大多数人所接受。

在MVC的体系架构中，Controller层负责整个APP中主要逻辑功能的实现；Model层则负责数据结构的描述以及数据持久化的功能；而View层作为展现层负责渲染整个APP的UI。分工清晰，简洁明了。此外，这种系统架构在语言框架层就得到了Apple的支持，所以非常适用于APP的startup开发。

然后，这种架构在开发的后期会由于其超高耦合性，造成Controller层庞大，而这也是一直被人们所诟病。最终的MVC都从Model-View-Controller走向了Massive-View-Controller的终点。

“短平快”的MVC架构在早期可以满足饿了么移动APP的快速开发迭代，但是随着代码量的不断增加，臃肿的Controller层也在渐露头角；而业务上，饿了么移动APP也从单一APP发展为多APP齐头并进的格局。这时候，降低耦合，复用已有模块便成了架构的第一要务。

架构中，模块复用的第一要求便是代码的功能组件化。组件化意味着拥有独立功能的代码从系统中进行抽象并剥离，再以“插件”的形式插回原有系统中。这样剥离出来的功能组件，便可以供其他APP使用，从而降低系统中模块与模块之间的耦合性；也同时提高了APP之间代码的复用性。

饿了么移动对于组件有两种定义：公有组件和业务组件。公有组件指的是封装得比较好的一些SDK，包括一些第三方组件和自己内部使用的组件。如iOS中最著名的网络SDK AFNetworking，Android下OKHttp，都是这类组件的代表。业务组件，则定义为包含了一系列业务功能的整体，例如登录业务组件，注册业务组件，即为此类组件的典型代表。

对于公有组件，饿了么移动采取了版本化的管理方式，而这在iOS和Android平台上早有比较成熟的解决方案。例如，对于iOS平台，CocoaPods基本上成为了代码组件化管理的标配；在Android平台上，Gradle也是非常成熟和稳健的方案。采用以上管理工具的另一个原因在于，对企业开发而言，代码也是一种商业机密。基于保密的目的，支持内网搭建私有服务器成为了必需。以上的管理工具都能够很好地支持这些操作。

对于业务的组件化，我们采取了业务模块注册机制来达到解耦合的目的。每个业务模块对外提供相应的业务接口，同时在系统启动的时候向Excalibur系统注册自己模块的Scheme（Excalibur是饿了么移动用来保存Scheme与模块之间映射的系统，同时能根据Scheme进行Class反射返回）。 当其他业务模块对该业务模块有依赖时，从Excalibur系统中获取相关实例，并调用相应接口来实现调用，从而实现了业务模块之间的解耦目的。

而在业务组件，即业务模块的内部，则可以根据不同开发人员的偏好，来实现不同的代码架构。如现在讨论得比较火的MVVM, MVP等，都可以在模块内部进行而不影响整体系统架构。

这时候的架构看起来更像是这样：

## 

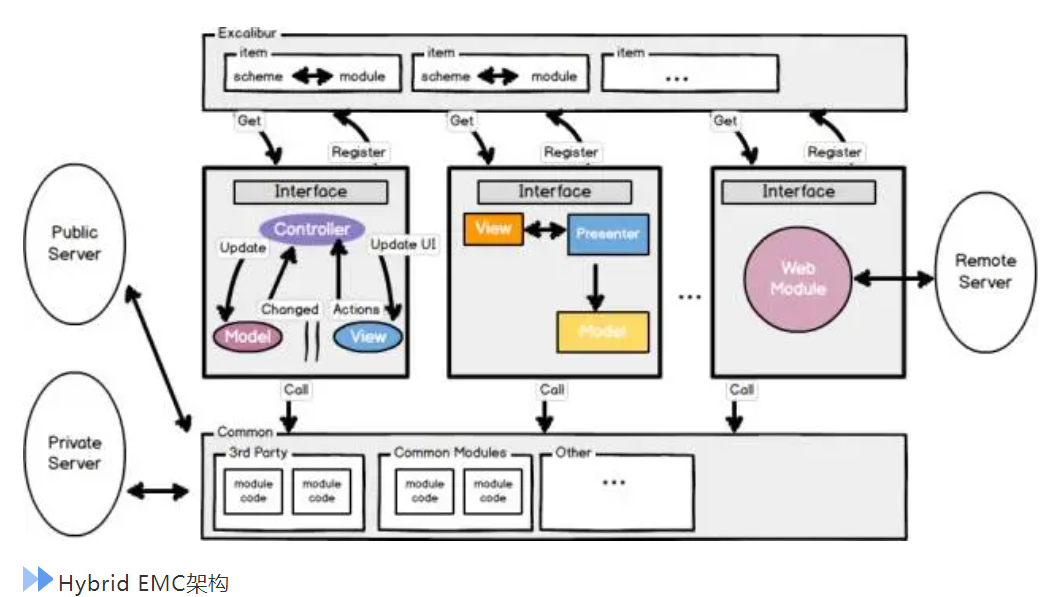
移动APP的开发有两种不同的路线，NativeAPP和Web APP。这两种路线的区别类似于PC时代开发应用程序时的C/S架构和 B/S架构。

以上我们谈到的都属于典型的Native APP，即所有的程序都由本地组件渲染完成。这类APP优点是显而易见的，渲染速度快、用户体验好；缺点同时也十分突出：出现了错误一定要等待下一次用户进行APP更新才能够修复。

Web APP的优点恰好就是Native APP的缺点所在，其页面全部采用H5撰写并存放在服务器端。每次进行页面渲染时都从服务器请求最新的页面。一旦页面有错误，服务器端进行更新便能立刻解决。不过其弊端也容易窥见：每次页面都需要请求服务器，造成渲染时等待时间过长，从而导致的用户体验不够完美，并且性能上较Native APP慢了1-2个数量级；与此同时还会导致更多的用户流量消耗。另一个缺点则在于，Web APP在移动端上调用本地的硬件设备存在一定的不便。不过这些弊端也都有相应的解决方案，如PhoneGap将网页提前打包在本地以减少网络的请求时间；同时也提供一系列的插件来访问本地的硬件设备。然而，尽管如此，其渲染速度上还是存在一定的差距。

**Hybrid APP**则是综合了二者优缺点的解决方案。饿了么移动对于此二类APP的观点在于，纯粹展示性的模块会更适合使用Web页面来达到渲染的目的；而更多的数据操作性、动画渲染性的模块则更适合采用Native的方式。

基于之前的EMC架构，我们将部分模块重新进行了架构：



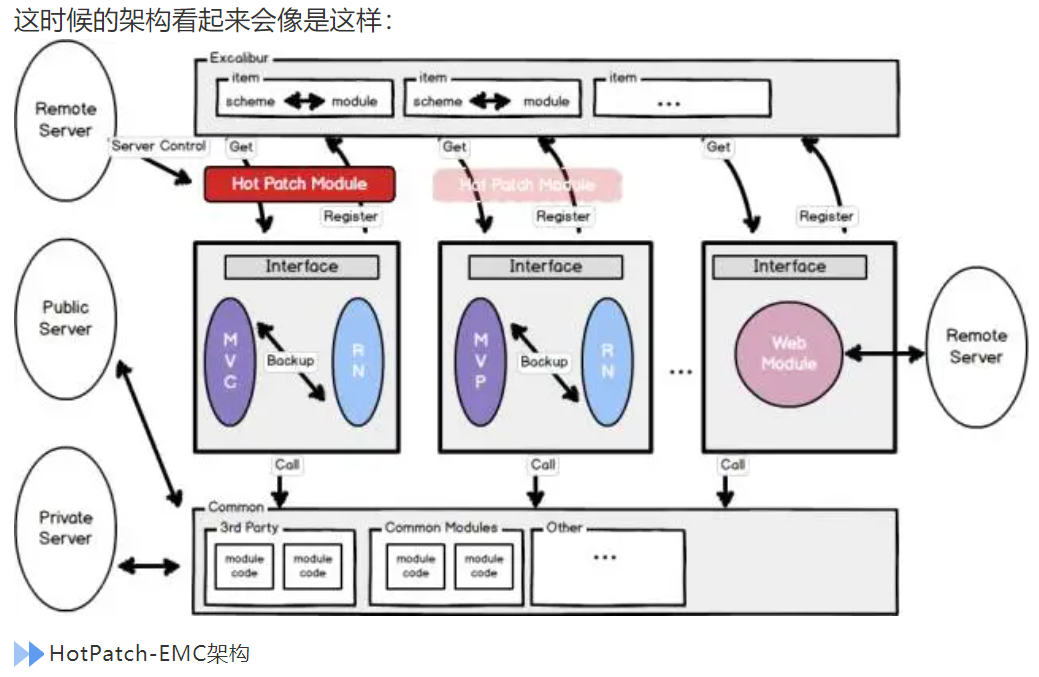
Hybrid-EMC架构中，Web作为一个子模块，注册加入到整个系统中，从而让业务上需要快速迭代的模块实现实时更新。

经过这些年的业务发展，Hybrid提供的展示界面更新方案也逐渐地无法满足APP更新迭代的需要。因此越来越多动态部署的方案被提了出来，比如iOS下的JSPatch, waxPatch，Android下的Dexpose, AndFix, ClassLoader，都是比较成熟Hot Patch动态部署解决方案。这些方案的思路都是通过下载远程服务器的代码来动态更新本地的代码行为。

React-Native则属于另一种动态部署的方案，其核心原理在于通过JavaScript来调用本地组件进行界面的渲染。

而饿了么移动APP发展到今天，各个APP综合用户量已经过亿。因此一个非常小的Bug所带来的问题都可能会直接影响到几万人的使用。为了保证APP的稳定性和健壮性，Hot Patch方案也就成了当下最有待解决的问题。

根据80%的用户访问20%页面这一80/20原则，保证这20%访问最频繁的页面的稳定性就是保证了80%的APP的稳定性。因此，饿了么移动对于部分访问最频繁的模块进行了React-Native备份。当这部分页面出现问题时，APP可以通过服务器的配置，自动切换成React-Native的备份页面；而与此同时开发人员开发一个小而精的Hot Patch来修复出现的问题。当Hot Patch完成修补后，再切换回Native APP的原生功能。

这时候的架构看起来会像是这样：

**HotPatch-EMC**的架构主要目标在于解决移动APP的稳定性问题。通过RN与Native的主备，可以减少系统APP出错带来的失误。