关于对DevOps的认识以及案例分析

“DevOps”这个词是 development 和 operations 这两个词的组合。它是一种促进开发和运维团队之间的协作，以自动化和可重复的方式更快地将代码部署到生产中的**文化。**

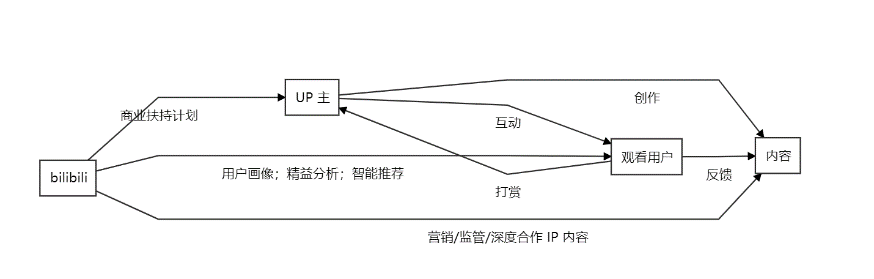
我对DevOps的认识就是，这个架构的用处是为了开发和运维能够进行更好的沟通与协作。DevOps和传统的运维有什么不同呢，我们假设有一个应用程序计划在2周内上线，代码完成80％。该应用程序是一个新的发布，从购买服务器开始

| **瀑布式开发** | **DevOps** |
| --- | --- |
| 订购新服务器后，开发团队需要进行测试。运维团队根据需求文档开始部署基础设施。 | 订购新服务器后，开发和运维团队根据需求文档共同调试部署新服务器。这样开发人员可以更好地了解服务器的基础架构。 |
| 关于故障转移，冗余策略，数据中心位置和存储要求的规划存在偏差，因为开发人员对应用程序有深入了解，但他们无法提供任何协助。 | 由于开发人员的加入，有关故障转移，冗余策略，灾难恢复，数据中心位置和存储要求的规划非常准确。 |
| 运维团队对开发团队的进展一无所知。只能根据运维团队理解制定监控计划。 | 在 DevOps 中，运维团队完全了解开发人员的进展。通过互动，共同制定满足运维和业务需求的监控计划。他们还使用应用程序性能监视（APM）工具以优化应用。 |
| 在上线之前，压力测试使应用程序崩溃。发布延迟了。 | 在上线之前，压力测试使应用程序有点慢。开发团队迅速解决了瓶颈问题。该应用程序按时发布。 |

我对于这个DevOps的案例分析选择的是B站。B站原名叫：Mikufans弹幕网。最早的雏形是“初音未来”的粉丝交流社区。于2010年1月改为提供视频资源的网站，并正式改名为Bilibili。

一．B站的生态环境和用户需求

B站的特点是以 " 原创内容 + 用户 + 弹幕 " 的交互方式所构筑的强粘性社区生态环境。



B站的用户价值以业务需求为载体，通过需求分析的筛选，快速地流入到研发并投产上线，并通过上线后的运营和用户来持续优化产品，实现整个业务价值的闭环。遵循DevOps的核心思想，持续快速的迭代反馈，特别是在用户需求的收集分析管理和用户反馈上，凸显出对用户价值的关注。接下来我们重点分析。首先是用户分析，我们看看B站核心用户画像。

* 二次元用户（核心目标用户）：可以追番剧，了解二次元文化，希望能认识更多同好。
* UGC创作者（核心目标用户）：希望有一个大平台可以让自己作品被看到，收到大家认同。
* 直播爱好者：希望有平台能展示自己的才能。
* 非二次元用户：只是单纯地希望能找到有序的视频，或跟随喜欢的UP主或主播而来。

有了清晰的用户画像，可以分析出B站的需求主要分为视频内容、用户体验、社区文化和氛围环境四类。

* 视频内容主要包括番剧、原创视频及UP主自制类视频功能，加上以用户为中心的页面布局、交互等用户体验设计，这两类需求针对核心用户占比会比较大，也呼应了前面的注重核心用户价值和体验。
* 社区文化类主要包括会员管理及支撑圈层的需求；氛围环境类包括弹幕和交流环境以及相关审核的需求，这两块为共同爱好者持续的交流分享营造一个良好的氛围，为核心功能提供支撑，这是B站重点响应的部分。

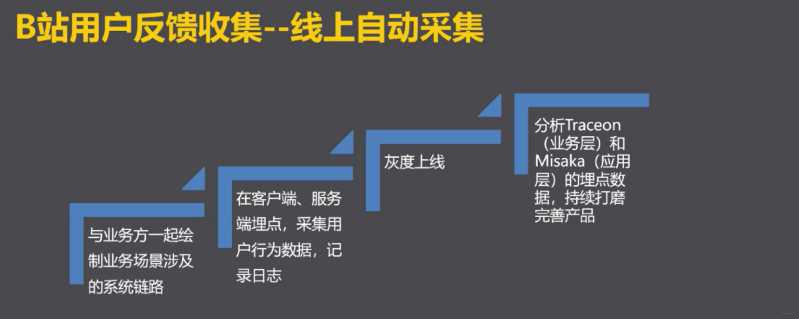
需求的收集主要从两个方面：

* 一方面通过全面分析商业价值、用户价值、分解转换为具体实现的需求；
* 另一方面在快速响应需求上线以后，收集用户反馈来持续完善产品的需求。

对于需求的优先级排序，B站有自己管理方法。

* 首先它是基于用户及商业价值的分析，基于核心用户的诉求和用户画像，优先考虑需求的迫切程度，通过良好的用户体验既维持老用户的粘度。对于系统运行稳定的一些非功能性的需求，也作为重点的排序考虑。
* 另外基于商业价值考量，针对付费用户，用付费情况来衡量需求价值，更量化且与商业接轨。
* 考虑用户与商业价值分析的同时，B站也会使用四象限法来进行需求的筛选和排序，通过需求的用户范围、发生频率和成本、效益，进行综合分析排序，确保用户价值价值较高的需求能够优先、快速地进入迭代循环，快速地上线并得到用户的反馈。

B站采用多种方式来收集用户的反馈，不仅包括了线下渠道的主动收集，而且还通过日志采集、设置埋点等多种手段，线上自动采集运营数据。对于线上的采集，采取了收集用户行为和用户数据的方式，首先与业务方一起绘制需采集业务场景涉及的系统链路，然后系统分布分别在客户端和服务端设置埋点，采集用户行为数据，记录日志。通过新功能灰度上线后目标用户数据采集和分析，形成对需求的快速反馈，持续打磨完善产品。



通过以线上与线下相结合的用户反馈收集机制，从用户画像分析、需求筛选排序到持续迭代反馈，用户的价值在DevOps循环中快速流动持续交付，B站实现了以用户价值为导向的需求管理的闭环。

二．高性能微服务实践

B站曾经的架构中存在很多的问题

* 代码维护难度大
* 运维配置复杂，很难扩展
* 监控不准，定位问题慢

微服务的优缺点以及架构如下：

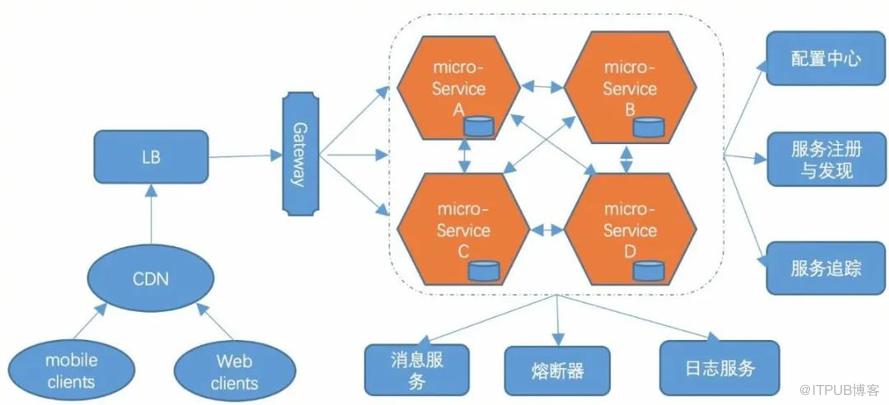
**1）优势**

* 各个服务独立开发。
* 分工明确，各自迭代。
* 单独模块独立部署。
* 可独立进行测试。

**2）挑战**

* 依赖链条长，测试常常遇阻，影响测试结果。
* 各服务的一致性难以保证。
* 运维成本高，错误难定位。
* 基础设施，网络等要求都比较高。

针对业务逻辑进行垂直划分切割，将一个巨大的服务体系，按业务逻辑切割成单元相对独立的服务，如：评论、硬币、稿件、收藏、feed等，服务间依赖标准采用RPC调用。



接下来是有关RPC框架：

B站一开始是以PHP为主流开发语言，后来为了快速支撑业务的发展，Node、Java、Python等开发语言也相继出现，导致B站的核心技术栈无法统一，对于微服务的落地，是兼容所有的语言基于现有架构改造，还是选择统一的技术栈进行重写？B站选择了后者。归根结底，重写后台工程（主要是账号相关的业务）是哔哩哔哩统一技术栈的一次尝试，至于最后为啥选择了用Go来实现RPC，在2017全球架构师峰会上，毛剑解释很简单：“主要就是我比较喜欢Go”，看似简单的一句回答，其实支撑其选择的原因还在于Go的诸多优势：

* 强大的标准库，解决了B站在视频方面的问题。
* 和Docker容器化良好的支持。
* 二进制发布，优秀的执行效率和开发效率。
* 易学易用上手快。
* 背靠Google大厂，丰富的开发者生态。
* 支持几乎所有主流的框架。

又或者说，选择Go非要有个原因么？为什么不能是Go呢？



**1）B站的RPC框架**

* 基于Go原生的网络库“net/rpc”。
* Gob进行struct的序列化，不需要拷贝额外的代码，使用较为方便。
* 方法级的超时控制。
* 上下文context的支持。

**2）服务注册与发现Discovery**

* 及时同步服务上下线事件。
* 服务发现节点自我发现。
* CP模式：依赖ZK的心跳进行广播（ZK心跳遇到抖动时候，容易出现全服务下线 ，所以B站会根据服务节点变化数进行判断是否放弃本次变更，进行容错）。
* AP模式：polling + ping（优先保证高可用，牺牲部分一致性，但最终达到一致）。

**3）配置中心**

* 利用mysql做存储管理相关配置信息。
* 通过http long polling来检查配置变更确保及时生效。
* 获取服务ip和版本，记录meta信息，实现服务发现的功能。

之后就涉及到了交付的问题，B站的快速交付，离不开需求分析之后，按照需求的优先级经过研发实施后进入运营监控的措施。

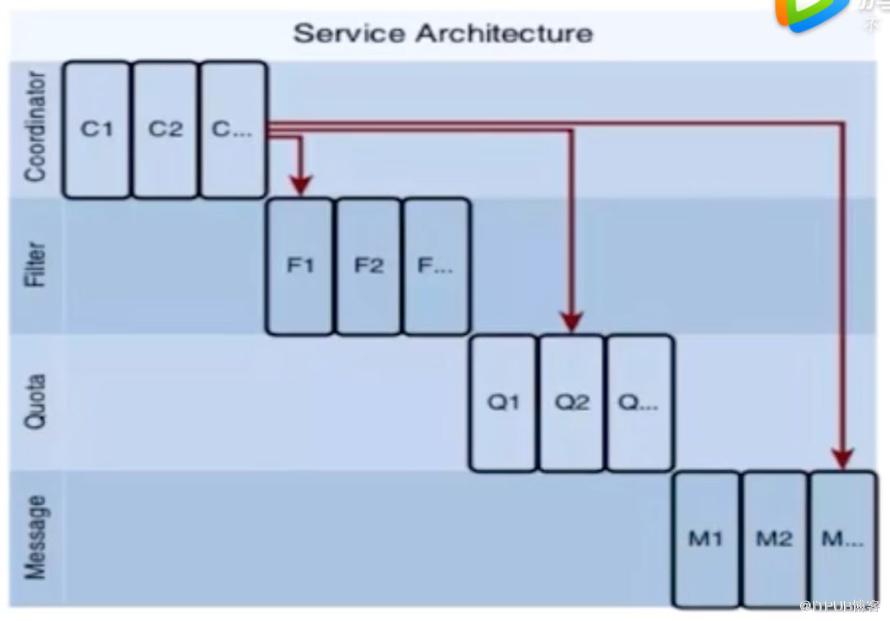
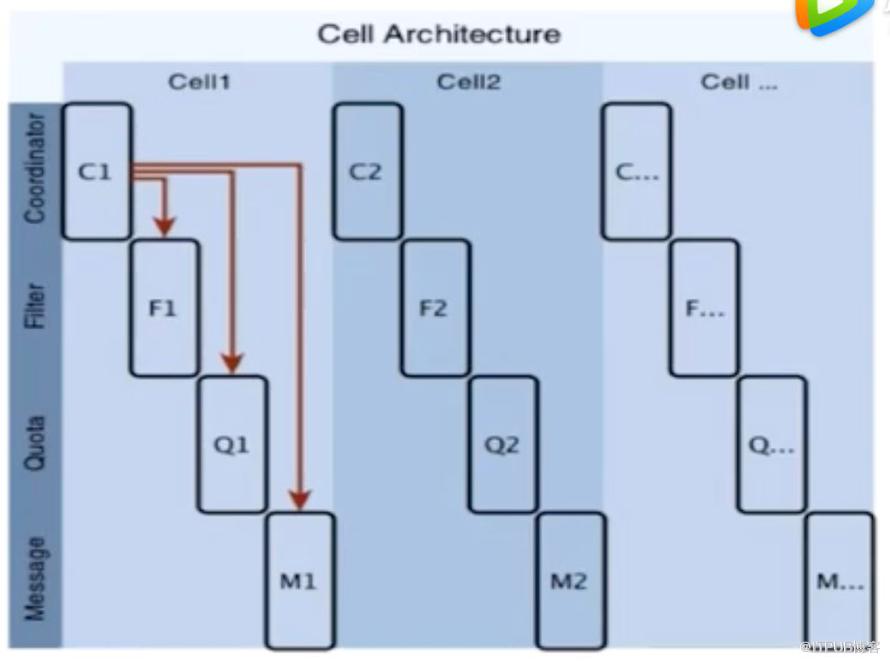
通过以线上与线下相结合的用户反馈收集机制，从用户画像分析、需求筛选排序到持续迭代反馈，用户的价值在DevOps循环中快速流动持续交付，B站实现了以用户价值为导向的需求管理的闭环。

微服务化以后，曾经的本地调用变成了远程调用，曾经的多节点对等变成了分布式的多节点，部署越来越复杂，调用链条越来越长，跨机房、跨网络、跨机架等都可能造成性能损失。**1）链路加速（客户端&服务端）**

* 慢、流量贵（移动端）。
* HTTP DNS & Server List。
* 多CDN加速（规避风险）。
* 协议优化（压缩、聚合、根据网速选择资源等）。

**2）网关优化（go自研，灵活可控）**

* 动态配置
* 协议装载
* 业务拦截
* 限流保护
* 缓存加速
* 鉴权
* 反作弊
* 业务服务
* 统计&监控

**3）优化服务部署架构**多个组，小集群方式部署，每个组依赖更少的节点，这样依赖资源有多套，相互之间也达到了隔离的作用。（一旦某个服务出现问题，全体受影响）（小集群，多个组，相互隔离，互不影响）

**可用性**

**1）隔离**

* 服务隔离：压力分流，稳定性高，物理隔离。
* 轻重隔离：核心稳定，快慢分离，流量迁移。
* 物理隔离：进程隔离，集群隔离，机房隔离。e.g. 机器隔离，容器隔离

**2）超时**

* 设置超时：连接超时，读取超时，写入超时。e.g. 避免挤压，防止雪崩
* 合理超时：避免过短，避免过长，动态设置。

**3）限流**

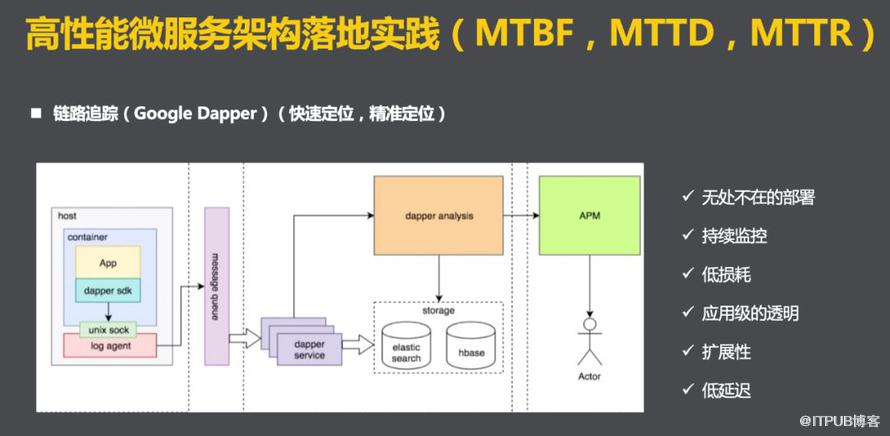
* 流量限流：accept，connection，thread。
* 资源限流：连接池，线程池。
* 请求限流：总数，时间窗口，平滑限流。
* 分布式限流：redis + lua，nginx + lua。
* 接入层限流：nginx limit\_req，nginx limit\_conn

**4)容错**

* 重试容错：简单重试，主备重试，成功率重试，快速失败。
* 熔断容错：动态剔除，异常恢复。

**5）降级**

* 调用链路：UI降级，UI异步请求降级，功能降级，读/写降级，接入层降级，应用层降级。
* 自动降级：超时降级，统计失败降级，服务故障降级，限流降级。
* 手动降级：功能开关，只读缓存，写异步化降级。

每个服务自身拥有比较健壮的服务能力，基本每个对外服务在代码层都能兼顾到降级、限流、容错、熔断、安全、健康检测。通过链路追踪保证，对错误能快速定位和精准定位，降低感知时间和修复时间。**6）Playbook 运维操作手册**

* 依赖的接口，必须加上熔断。
* 当接口出现故障，自动熔断，普罗米修斯出熔断数据曲线图，并且报警。
* 当超过N分钟，服务仍然不恢复，可以使用配置中心的推送功能，打开强制熔断，不再依赖接口。
* 当依赖方告知服务恢复，重新关闭熔断开关，变成自动熔断状态。

**7）运维层面，定期故障演练，确保流程可被正确可靠执行。**

**一致性**

**1）存储一致性**

* MySQL本地事务
* 本地事务+消息队列
* Binlog

**2）服务一致性**

* 消息队列
* 幂等
* 努力送达
* 事务补偿

**微服务部署发布**

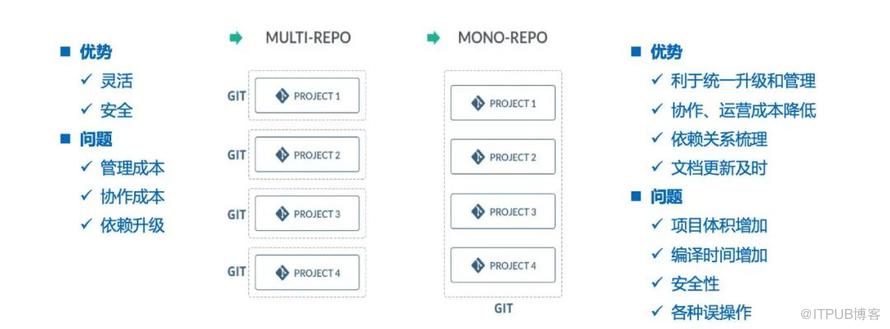
****  
**1）灰度发布 - 染色**

* 在PaaS平台上选择灰度发布-染色，定义染色标签。
* 在IaaS平台上开启“环境代理”虚拟机。
* 镜像选择环境代理hasson-base最小配置即可。
* 在hasson的虚拟机录入染色标签，按需配置测试接口。
* DNS指向hasson IP，即可开始测试。

RPC meta info & contextserviceA(Red) -> serviceB -> serviceC(Red)**2）灰度发布 - feature flag**

* 尽早的进入主干，可以跟灰度发布结合使用（金丝雀发布），需要使用统一的flag机制。
* 需要清理不再需要的flags，需要开发者养成良好习惯，正确使用flags，否则容易显得很乱。

**代码管理**

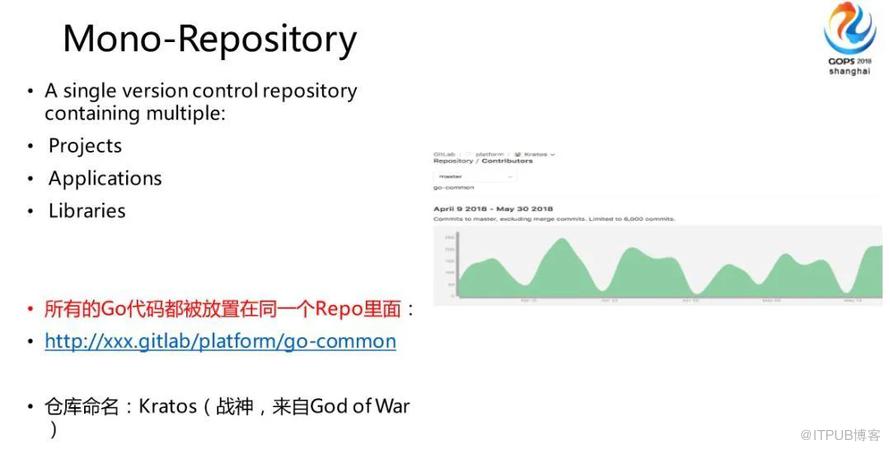


* 随着业务发展，基础库变更频繁，推进困难。
* 虽然重视代码质量，但流程不够自动化，完全靠自觉。
* 测试都积压在发版前，质量难保证，发版风险大。
* 版本管理非常复杂，代码复用率低下，维护成本高。

通过比对分散式管理方式和集中式管理方式，显而易见，大仓库的优势在于统一版本依赖、增强协作、最大化复用代码以及减少版本不一致所引发的各种线上运营风险。

**1）大仓库管理代码带来的好处：**

* 统一版本控制
* 广泛的代码共享和重用
* 简化依赖管理，避免菱形依赖
* 原子修改
* 大规模重构
* 跨团队协作
* 灵活的团队边界和代码所有权
* 代码可见性以及清晰的树形结构提供了隐含的团队命名空间

(引自：Google为什么要把数十亿行代码放到一个库中？)****  
同时，大仓库所带来的的挑战也很明显，如何保证安全性，如何防止误操作，如何减少每次checkout代码的时间和编译的时间，B站进行了如下的应对。

**1）目录级权限管理**

**2）统一的构建系统Bazel**

* 支持多种开发语言。
* 依赖分析增量编译。
* 可按需进行扩展。

**3）更高的代码质量要求**

* 统一的代码规范，便于协作。
* 单元测试覆盖率，高质量传承。

**4）注重CodeReview，鼓励沟通**

* 保证业务逻辑代码质量。
* 信息传递，扩充人员备份。
* buildup competence，提升集体战斗力。
* Ownership机制，责任到人。

以下是有关运维：



优点：

**加快**[上市](https://en.wikipedia.org/wiki/Time_to_market)时间：缩短从构思构思到将其出售或投入生产所需的时间。

**构建正确的产品** ：开发人员可以通过更频繁地发布和实时测试想法来从用户那里获得更快的反馈

**降低成本** ：报告平均降低20％。

更高的**生产率** ：通过持续交付，开发人员和测试人员可以节省设置和修复工作环境的时间。 而且，部署速度明显更快(稍后在“与Jenkins的持续集成”部分中将对此进行详细介绍)。

**可靠的发行版** ： **发行版**较小且更频繁时，代码中的更改(因此引入的错误及其影响)也较小。

**提高产品质量** ：公司报告显着减少了打开的错误和其他问题(在某些情况下，降低了90％以上)。

**更高的客户满意度** ：毫无疑问，这是以前所有改进的副产品。

缺点：

第一个是在**组织**级别。 由于消除了开发和部署方面的限制，并且程序员和系统管理员具有更大的独立性，因此相关人员将需要采用不同的思维方式，并且需要建立从操作到开发的反馈循环的适当机制，例如问题跟踪程序，讨论区(例如[Slack](https://slack.com/) )等。

第二个关键挑战是关于**流程** 。 您不再希望开发人员和服务器管理员自己花费时间来进行更改后的单独测试。 您想要的是使测试过程自动化，以便您可以允许不同的团队进行更改并快速检查一切是否仍在进行中，并在出现问题时撤消这些更改。

最后，存在**技术**挑战。 如前所述，一旦您的组织从头到尾仔细审查了流程，您可能需要采用或创建一种技术，以解决更适合您的流程和组织的自动化和反馈回路。