大中型SNS系统设计漫谈

美图 洪小军
@XiaoJunHong

meitu

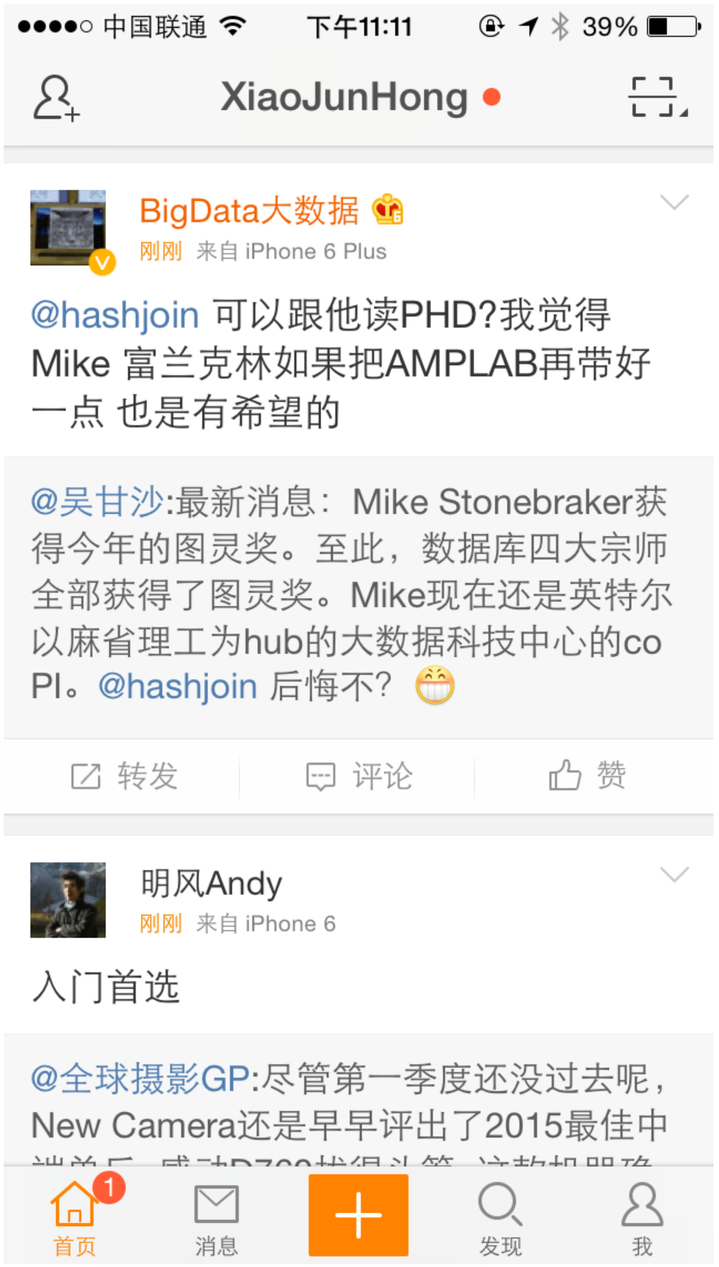
大中型SNS系统架构设计漫谈

- 个人简介
  - 洪小军 @XiaoJunHong
  - 2014年底加入美图：架构平台部门负责人
  - 2011年初加入新浪微博：平台架构团队负责人
  - 2009年初加入飞信：架构师

meitu

SNS系统简介







meitu

SNS系统简介



- Feed
- Counter
- Graph
- User
- IM
- ……

meitu

讨论大纲

- 业务系统架构演化
- 设计基本原则

业务系统架构演化

- 计数器服务
  - 美拍评论数、赞数
  - 微博评论数、转发数、赞数

计数器服务

- 快速上线阶段
  - 最“粗”的实现方式
    - select count(1) from table where id=xxx
  - 挑战点：数据库出现大量慢查询
    - 计数器服务往往是SNS系统中调用量最大的服务

meitu

计数器服务

- 独立维护计数器

  - memcached + mysql
    - 计数在mysql中独立维护
    - select count from table where id=xxx

  - 挑战点：接口响应时间得不到保证，DB也有较多慢请求
    - 数据库有大量的update和select操作

meitu

计数器服务

- Memcached + Mysql
  - 数据库有大量的update操作
    - 热门消息并发转发和评论量巨大，造成大量的update操作
    - update table set count=count+1 where id=xxx
  - 数据库有大量的select操作
    - 有大量的消息是没有被转发和评论，要不要缓存这些计数为0的?
    - SNS长尾特性很明显，很难保证非常高命中率

meitu

计数器服务

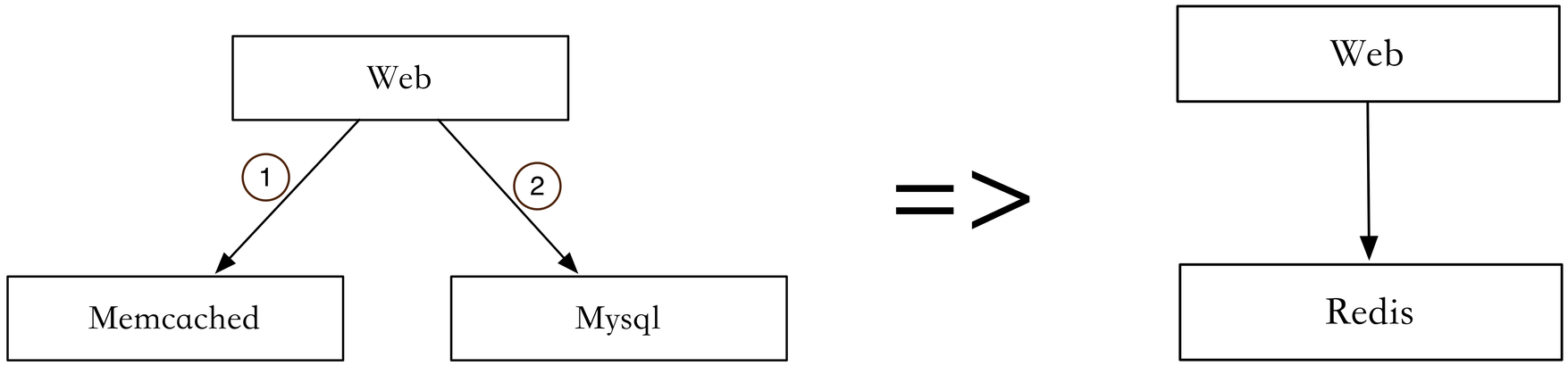
- Memcached + Mysql
  - 数据库合并写入
    - 程序端合并写入请求，有起到一定效果，但是不太明显
    - update table set count=count+5 where id=xxx
  - 缓存扩容尽可能存放更多数据
    - 由于长尾效应还是很难达到非常高命中率

meitu

计数器服务

- Redis内存存储方案

  - Redis作为内存存储
    - 存放所有计数器>0的数据
    - 高速的读取性能，保证接口的响应时间可控



计数器服务

- 架设在内存基础上的架构

	内存缓存	内存存储
请求访问策略	从cache获取为空穿透到db	只从内存获取
过期策略	LRU	不过期
命中率	取决于内存容量	100%
容量要求	适中？	高？（也取决于空数据比例、长尾数据量等）
SLA情况	适中	较好

meitu

计数器服务

- Redis内存存储方案
  - 挑战点：随着时间增长，海量长尾冷数据全量内存存储
    - 导致成本增大，资源利用率不高
  - 业界的典型解决方案
    - 冷热数据分离，热数据全量内存，冷数据使用SSD
    - 定制化redis，实现更紧凑的存储

meitu

Feed服务

- 推拉模式 - 推模式
  - 空间换时间
    - Insert into feeds(uid, author_id, feed_id)
      Select $uid, follower_id, $feed_id from followers
  - 获取消息
    - Select * from feeds where uid = $uid

uid	author_id	feed_id
10001	20001	1111111111
10002	20001	1111111111

meitu

Feed服务

- 推拉模式 - 推模式
  - 空间换时间：存储容量瓶颈?
  - 粉丝越多推送量越大：大V粉丝推送?
  - 变更通知成本高：加减关注？删除Feed?

meitu

Feed服务

- 推拉模式 - 拉模式
  - 写入消息
    - Insert into feeds(uid, feed_id) values($uid, $feed_id)
  - 获取消息
    - Select * from feeds where uid in
      (select following_id from following)

uid	feed_id
10001	1111111111
10002	222222222

meitu

Feed服务

- 推拉模式 - 拉模式
  - 时间换空间：聚合计算瓶颈?
  - 大量的计算，网卡带宽耗费也较大

Feed服务

- 推拉模式对比

	推	拉
获取Feed	简单、高效	实时计算量大，与好友数量相关
发表Feed	推送给所有粉丝	不推送
变更通知	加关注、取消关注、删除Feed等都需要变更	不需要
好友多、粉丝少	适合	不适合
好友少、粉丝多	不适合	适合
好友多、粉丝多	？	？

Feed服务

- 业界案例参考 – Twitter推拉结合
  - 普通用户基于推方式
    - 核心关注点为怎么尽快推送给关注者
  - 大V用户基于拉方式

Feed服务

- 业界案例参考 – Weibo推拉结合
  - 公开性微博基于拉方式
    - 核心关注点为怎么提高聚合效率，包括响应时间和成本
  - 私密性微博基于推方式

Feed服务

- 数据库设计

  - 挑战点：快速发展阶段频繁加字段

  - 需要保证查找性能的同时在线加新字段

uid	feed_id	text	.....
10001	1111111111	text1	?
10002	222222222	text2	?

Feed服务

- 数据库设计 - 索引和内容分离
  - 索引表：只存放需要索引的字段，需要保证高性能查找
  - 内容表：KV结构，V为二进制数据，便于字段变更
    - 内容紧凑型存储，使用可扩展的序列化，如Protocol Buffer

uid	feed_id
10001	1111111111
10002	222222222

feed_id	content
1111111111	[bytes]
222222222	[bytes]

meitu

Feed服务

- 数据库设计
  - 挑战点：面临很大读压力，很多读慢查询
  - 解决方案
    - 读写分离，部署更多的slave节点抗读
    - 数据库只是作为存储，不应该有任何逻辑

meitu

Feed服务

- 数据库设计

  - 挑战点：面临很大写压力
  - 解决方案：分库分表
  - 引出新的问题：
    - 之前依赖数据库自增id的方式如何解决?
    - 基于什么策略分库分表?

Feed服务

- 数据库设计 - 发号器服务
  - 问题场景：需要有全局唯一的id，并且是递增的
  - 发号器服务：
    - 程序自身带发号功能：重点需要保证id的唯一性
    - 全局统一的发号器服务：独立部署的服务

Feed服务

- 数据库设计 - 发号器还可以有很多用途
  - 64位的id里面究竟可以存放多少东西?
  - 假设业务上每秒峰值写入会有50万
    - 32位：精确到秒
    - 20位：可以存放52万数据
    - 剩余12位：还可以存放IDC信息、业务信息及其方便做HA

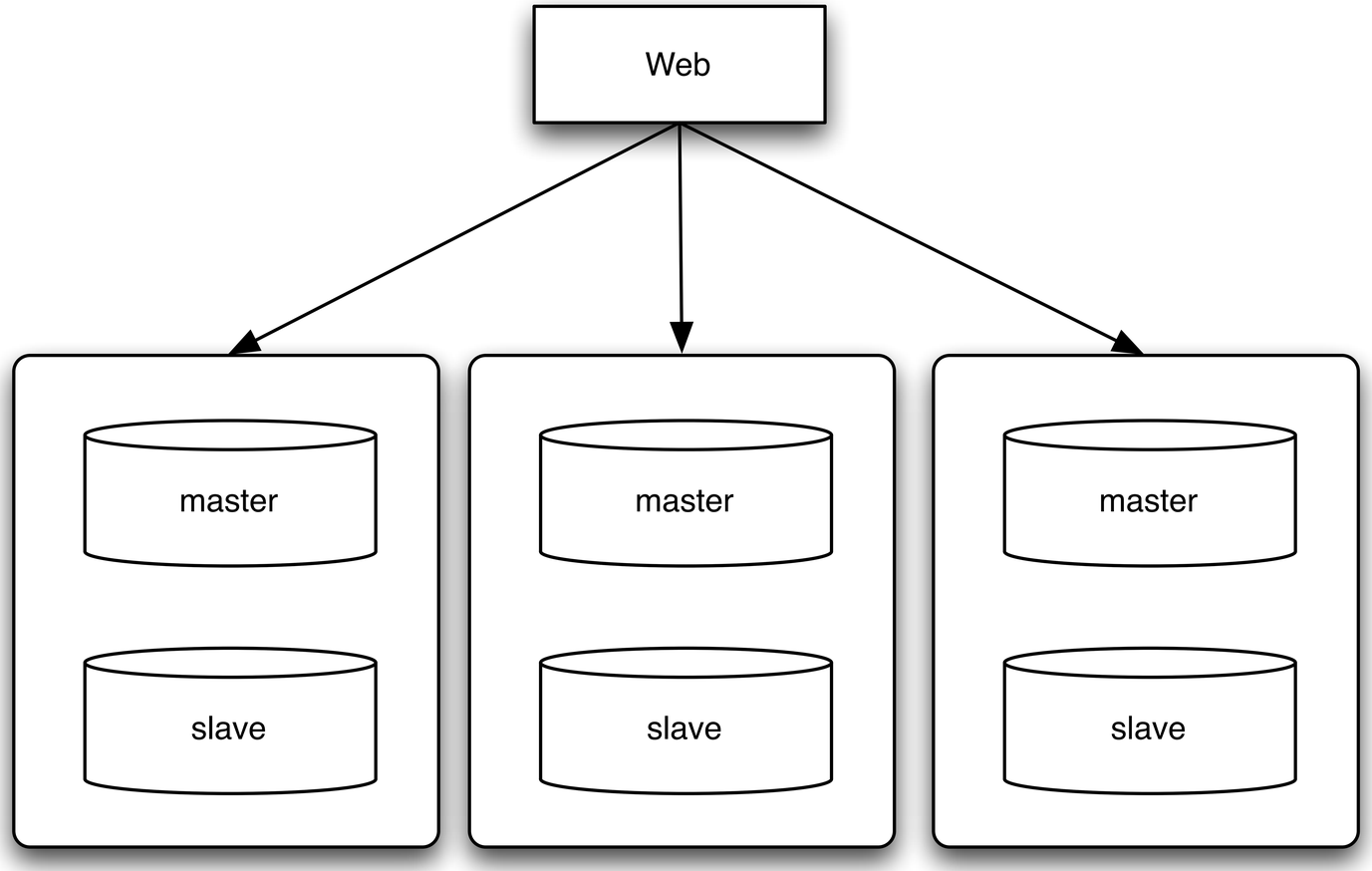
meitu

Feed服务

- 数据库设计 – 分库分表策略
  - 分库：物理库、逻辑库
  - 分表：Hash分布和时间分布

Feed服务

- 数据库设计 – 分库策略



Feed服务

- 数据库设计 – 基于hash分表
  - 每个库N张表，基于Hash定位
  - 最常见和使用最多的场景
  - 系统中前期比较推荐的做法

Feed服务

- 数据库设计 – 基于hash分表
  - 挑战1：特殊用户查询性能下降很厉害
    - 发现这些用户发布的内容太多，长期积累起来单用户数据量太大
  - 挑战2：想升级使用更好硬件（SSD），但是总成本太大
    - 有大量的历史数据，如果升级需要整体升

Feed服务

- 数据库设计 – 基于时间分表
  - 数据按月存放，增加二级索引表

timeline	
uid	long
id	long

timeline_si	
uid	long
stat_date	date
count	int

Feed服务

- 数据库设计 – 基于时间分表
  - 通过二级索引表获取月份信息
  - 通过一级索引获取具体数据

timeline_si		
uid	stat_date	count
100001	2015-03	2
100001	2015-02	1
100001	2015-01	1
……	……	……

timeline_1503	
uid	id
100001	555555
100001	444444

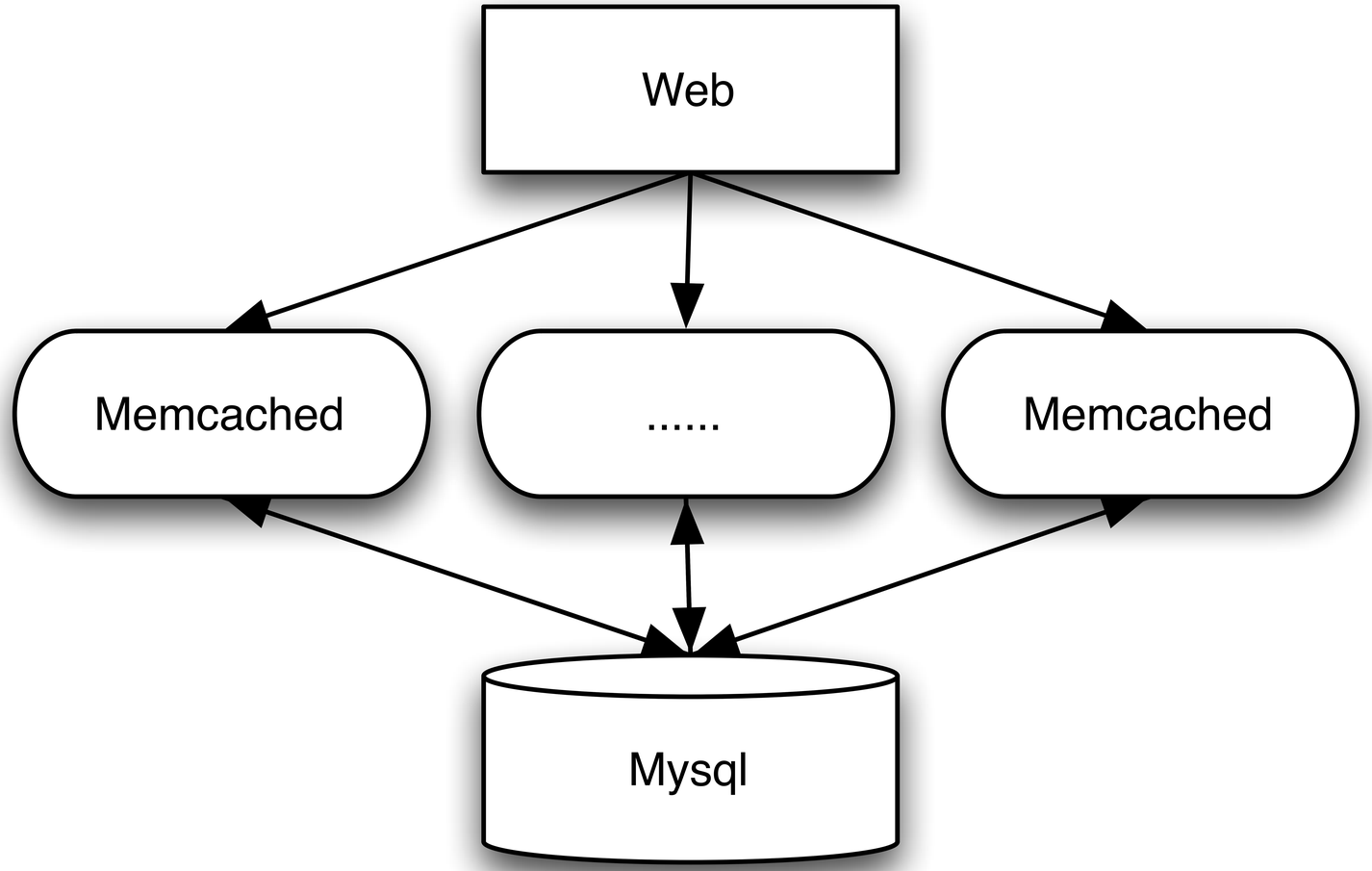
timeline_1502	
uid	id
100001	333333

timeline_1501	
uid	id
100001	222222

meitu

Feed服务

- 缓存设计 - 分布式缓存



Feed服务

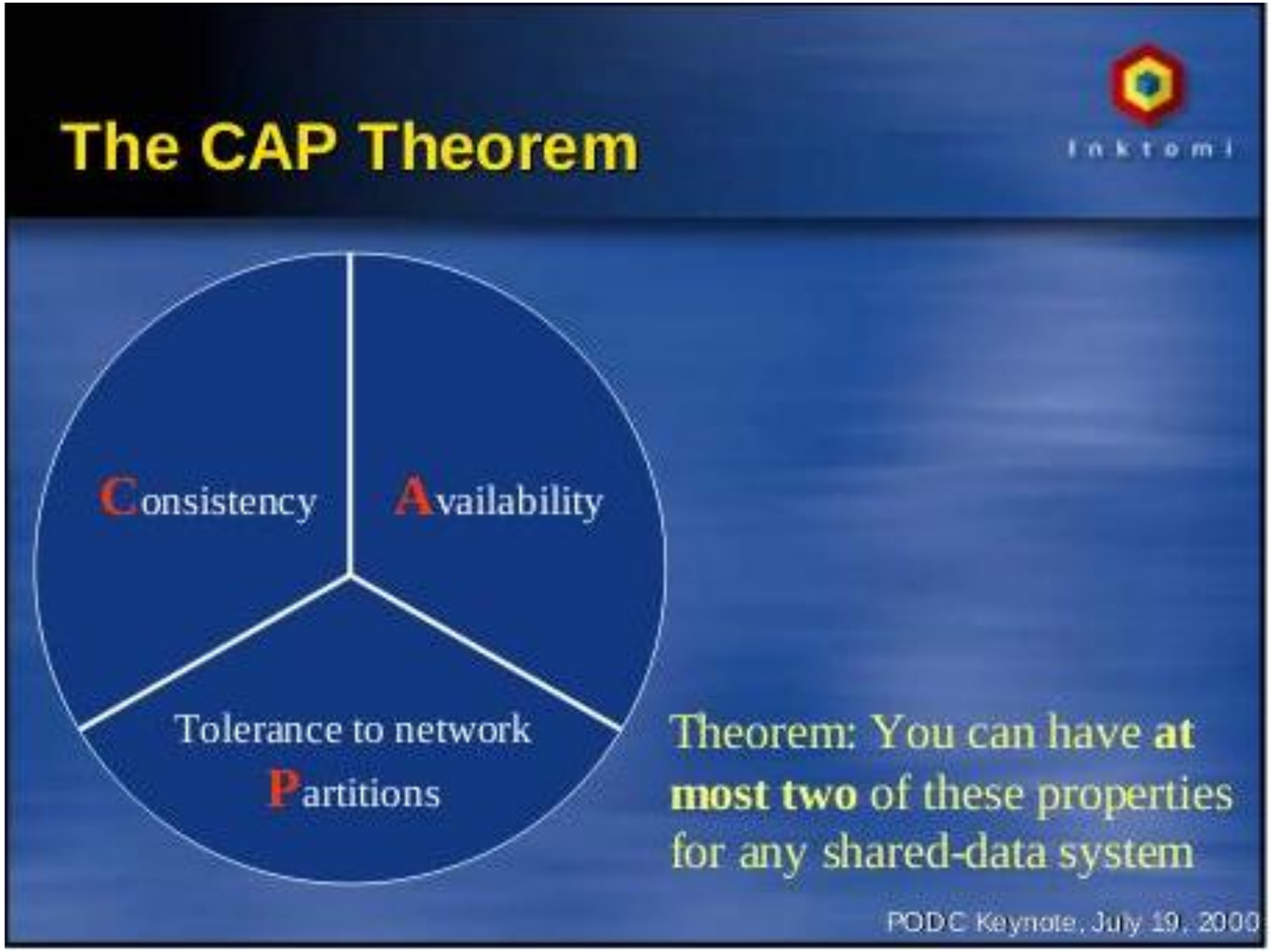
- 缓存设计
  - 挑战：缓存节点出现故障，将可能出现雪崩问题
  - 解决方案：Master/Slave部署

Feed服务

- 缓存设计
  - 挑战：网卡带宽/调用量成为瓶颈
  - 解决方案1：扩容节点数?
    - multiget hole
  - 解决方案2：线性扩展方式 - 部署多套集群

设计基本原则

- CAP原则



设计基本原则

- SNS系统的考虑
  - 可扩展性?
  - 可用性?
  - 一致性?

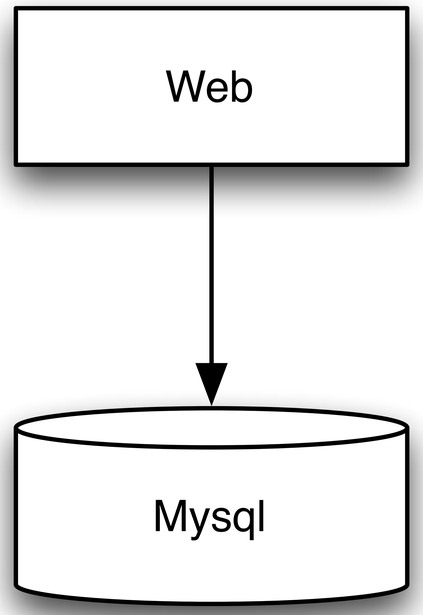
=>

- 选择可扩展性、可用性
- 妥协一致性 – 最终一致

meitu

设计基本原则

- 系统演化 - 前期
  - 快速搭建并推进上线，验证产品可行性



设计基本原则

- 系统演化 – 中期
  - 高可扩展性
    - Scale Up：硬件升级、单机性能优化
    - Scale Out：分布式支持
  - 路线图
    - Scale Up -> Scale Out -> Scale Up

meitu

设计基本原则

- 高可扩展设计
  - 支撑高并发读取
    - 架设在内存之上
  - 支撑峰值写入
    - 异步化处理
  - 分布式部署
    - 多集群、多机房

设计基本原则

- 支撑高并发读取
  - 支撑大量级接口调用
    - 对于资源的调用会扩大化
  - 保证较低的响应时间
    - 满足SLA要求

设计基本原则

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K w/cheap compression algorithm	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Google

meitu

设计基本原则

- 架设在内存基础上的架构
  - 内存缓存
  - 内存存储

设计基本原则

- 架设在内存基础上的架构

	内存缓存	内存存储
内存命中率	取决于内存容量	100%
请求访问策略	从cache获取为空穿透到db	只从内存获取
过期策略	LRU	不过期
容量要求	适中？	高？（也取决于空数据比例）
数据一致性	强	适中
SLA情况	适中	较好

meitu

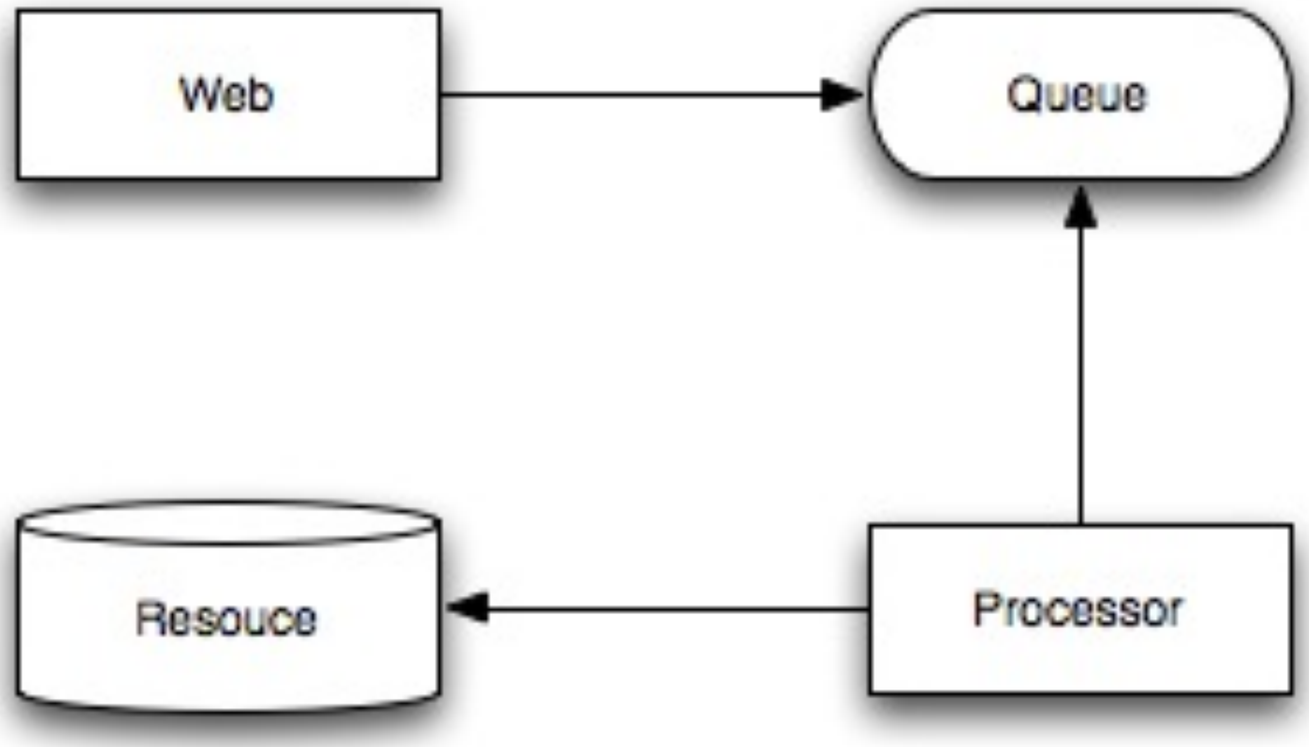
设计基本原则

- 支撑峰值写入
  - 满足时事热点时大量的并发写入
  - 保证一直可写，即使存储等出现问题

meitu

设计基本原则

- 异步化处理
  - 前后端分离，前端永远可写



设计基本原则

- 多机房部署
  - 可靠的消息同步方式
  - 避免跨机房读调用

设计基本原则

- 典型存储/缓存服务的消息同步方式
  - Mysql：相对成熟和可靠的master/slave方式
  - Memcached：自身无同步方式，依靠消息更新
  - Redis：有条件的master/slave方式
    - 异常重连2.8+可有条件支持增量同步，否则为全量同步

meitu

设计基本原则

- 系统演化 – 中后期

  - 高可用性
    - Failover：避免单点故障
    - Failfast：保证响应时间
    - 降级机制：柔性可用
    - 演练机制：及早发现系统瓶颈

meitu

Q&A

美图期待牛人加盟（hxj@meitu.com)