

# 高性能的Java代码及常见问题排查

林昊

<http://hellojava.info>

# 参考资料

- [Java代码的执行](#)
- [学习JVM的References](#)
- [Building memory-efficient Java Applications](#)
- [Achieving Rapid Response Times in Large Online Services](#)
- [highscalability.com](#)
- [Research at Google](#)
- [Facebook Engineering](#)

# 推荐书籍

- 《Java并发编程实战》
- 《Programming Concurrency on the JVM》
- 《JRockit:The Definitive Guide》
- 《The Garbage Collection Handbook》
- 《Memory Systems》
- 《Fundamental Networking in Java》
- 《Pro (IBM) WebSphere Application Server 7 Internals》
- 《虚拟机：系统与进程的通用平台》
- 《What every programmer should know about memory》
- 《Java Performance》

# Agenda

- 编写高性能Java代码
  - 并发
  - 通信
  - JVM
- Java常见问题排查
- 典型的互联网技术

# 编写高性能Java代码

- 通用技能
  - 算法
  - 数据结构
- 语言相关
  - 并发
  - 通信
  - JVM

# 编写高性能Java代码

- 并发
  - 线程
    - 创建一个线程到底耗多少内存呢？
    - 一台机器上到底能创建多少个线程呢？
      - 常见错误：Unable to create new native thread

# 编写高性能Java代码

- 并发
  - 线程池
    - `Executors.newCachedThreadPool();`
    - `new ThreadPoolExecutor(10,20,5,TimeUnit.MINUTES,new ArrayBlockingQueue<Runnable>(10));`
  - 线程池很容易带来的一些“副问题”
    - `ThreadLocal`累积...

# 编写高性能Java代码

- 并发
  - 线程之间的交互
    - wait/notify/notifyAll
    - CountdownLatch
      - 等一组动作完成
    - CyclicBarrier
      - 一组动作一同开始
    - Semaphore
      - 例如连接池类型的场景



# 编写高性能Java代码

- 并发

- 锁

- synchronized/ReentrantLock
    - java.util.concurrent展示了各种减少锁冲突技术
      - CAS
        - Atomic\*
        - 例如初始化代码可基于AtomicBoolean做优化
      - 拆分锁
        - ConcurrentHashMap
      - 读写锁
      - Non-Blocking
        - 通常基于CAS
        - ConcurrentLinkedQueue

# 编写高性能Java代码

- 一段基于j.u.c的优化代码Case

- `private Map<String,Object> caches=new HashMap<String,Object>();`
- `public Object getClient(String key,...){`
  - `synchronized(caches){`
    - `if(caches.containsKey(key)){`
      - `return caches.get(key);`
    - `}`
    - `else{`
      - `Object value=// create...`
        - `caches.put(key,value);`
      - `return value;`
    - `}`
  - `}`
- `}`

# 编写高性能Java代码

- 简单的优化

- `private Map<String,Object> caches=new HashMap<String,Object>();`
- `public Object getClient(String key,...){`
  - `if(caches.containsKey(key)){`
    - `return caches.get(key);`
  - `}`
  - `else{`
    - `synchronized(caches){`
      - `// double check`
      - `if(caches.containsKey(key)){`
        - `return caches.get(key);`
      - `}`
      - `Object value=// create...`
      - `caches.put(key,value);`
      - `return value;`
    - `}`
  - `}`
- `}`

# 编写高性能Java代码

- 基于j.u.c的优化

- `private ConcurrentHashMap<String,FutureTask<Object>> caches=new ConcurrentHashMap<String,FutureTask<Object>>();`
- `public Object getClient(String key,...){`
  - `if(caches.containsKey(key)){`
    - `return caches.get(key).get();`
  - `}`
  - `else{`
    - `FutureTask<Object> valueTask=new FutureTask<Object>(  
• new Callable<Object>(){`
      - `public Object call() throws Exception{`
        - `// create...`
      - `}`
    - `}`
    - `FutureTask<Object> current = caches.putIfAbsent(key, valueTask);`
    - `if(current == null){`
      - `valueTask.run();`
    - `}`
    - `else{`
      - `valueTask = current;`
    - `}`
    - `return valueTask.get();`
  - `}`
- `}`

# 编写高性能Java代码

- 并发
  - 并发相关的问题
    - 工欲善其事，必先利其器
      - jstack [-l]
      - 看懂线程dump
        - 给线程命名很重要
    - 不太好排查，人脑是串行的...

# 编写高性能Java代码

- 并发
  - 线程不安全造成的问题
    - 经典的并发场景用HashMap造成cpu 100%的case
      - Velocity/Hessian等都犯过的错...
  - 表象：应用没反应
    - 死锁
      - Spring 3.1.4-版本的deadlock case
      - 一个static初始化的deadlock case
    - 处理线程不够用

# 编写高性能Java代码

- 通信
  - 数据库连接池高效吗?

# 编写高性能Java代码

- 通信
  - 基本知识
    - BIO/NIO/AIO
    - File Zero Transfer



# 编写高性能Java代码

- 通信
  - 典型的nio框架的实现
    - netty

# 编写高性能Java代码

- 通信
  - 连接
    - 长连 Vs 短连
    - 单个连接 Vs 连接池
  - 推荐：单个长连接
    - 容易碰到的问题
      - LB
  - 说说重连
    - 一个重连设计不好造成的严重故障Case
    - 连接状态检测

# 编写高性能Java代码

- 通信
  - 高性能Client编写的技巧
    - 选择一个靠谱的nio框架
      - netty
    - 单个长连接
    - 反序列化在业务线程里做
    - 高性能、易用的序列化 / 反序列化
      - PB
    - io线程批量通知
    - 尽量减少io线程的上下文切换

# 编写高性能Java代码

- 通信
  - 高性能Server编写技巧
    - 和client基本相同
    - 业务线程池
    - 反射method cache

# 编写高性能Java代码

- 通信
  - 通信协议的设计
    - 版本号
    - 扩展字段

# 编写高性能Java代码

- 通信

- 常见问题

- too many open files
    - 支持超高的并发连接数（10k，100k甚至1000k）
      - 8 core/8g，轻松支撑15k+
    - TIME\_WAIT(Client)、CLOSE\_WAIT(Server)
    - 网络通信慢
      - send buf/receive buf
      - 中断处理cpu均衡
      - tcpdump抓包分析

# 编写高性能Java代码

- JVM
  - 代码的执行
    - 编写源码时编译为bytecode
    - 启动后先解释执行
    - CI/C2编译
      - 经典的编译优化，相较静态而言更为高效
        - static final值
        - inline
        - 条件分支预测等
        - EA
    - TieredCompilation

# 编写高性能Java代码

- JVM
  - 代码的执行
    - 编写正确的MicroBenchMark
      - Warm
      - -XX:+PrintCompilation

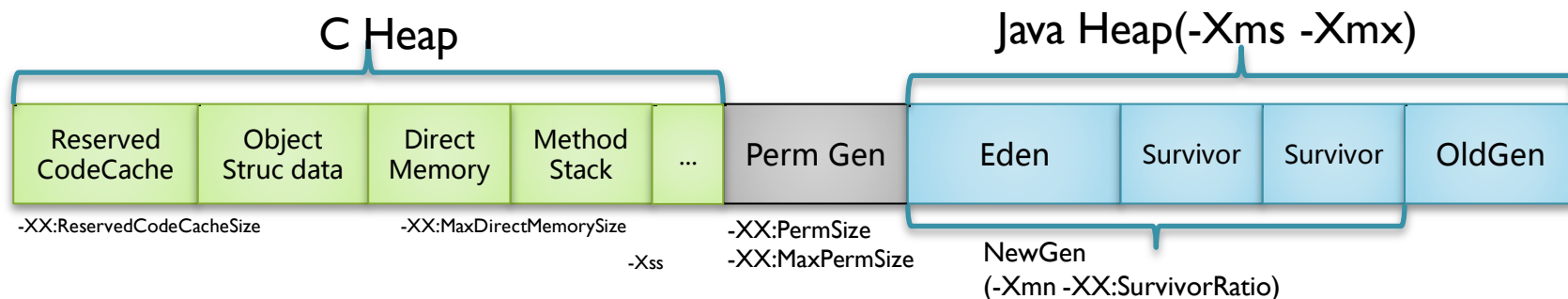


# 编写高性能Java代码

- JVM
  - 代码的执行
    - 传说中的大方法性能更差
      - 是有道理的
  - 一个性能狂降的case
    - CodeCache is full. Compiler has been disabled

# 编写高性能Java代码

- JVM
  - 内存管理
    - 内存区域划分



# 编写高性能Java代码

- JVM

- 内存管理

- GC (Garbage Collector) : 负责内存的分配和回收
      - Serial
        - server端基本不用
      - Parallel
      - Concurrent
      - G1GC

# 编写高性能Java代码

- JVM

- 内存管理

- Parallel GC

- Stop-The-World (STW)
      - 回收时多线程执行
        - -XX:ParallelGCThreads
      - YoungGen采用Copy算法实现，Full采用Mark-Compact算法实现
      - 有两种可使用
        - -XX:+UseParallelGC (ServerVM默认)
        - -XX:+UseParallelOldGC (优化版本)

# 编写高性能Java代码

- JVM
  - 内存管理
    - Parallel GC
      - 相关参数
        - -XX:SurvivorRatio（默认无效），原因是...
        - -XX:MaxTenuringThreshold（默认无效），原因同上；
        - -XX:-UseAdaptiveSizePolicy
        - -XX:ParallelGCThreads

# 编写高性能Java代码

- JVM

- 内存管理

- Concurrent GC

- 简称CMS

- 新生代ParNew（Copy算法），旧世代CMS（采用Mark-Sweep算法），Full采用Serial

- Mostly Concurrent

- 分为CMS-initial-Mark、CMS-concurrent-mark、CMS-concurrent-preclean、CMS-remark、CMS-concurrent-sweep、CMS-concurrent-reset

- 其中CMS-initial-Mark、CMS-remark为STW；

- -XX:+UseConcMarkSweepGC

# 编写高性能Java代码

- JVM
  - 内存管理
    - Concurrent GC
      - 相关参数
        - -XX:SurvivorRatio
        - -XX:MaxTenuringThreshold
        - -XX:CMSInitiatingOccupancyFraction
        - -XX:CMSInitiatingPermOccupancyFraction
        - -XX:+UseCMSInitiatingOccupancyOnly

# 编写高性能Java代码

- JVM
  - 内存管理
    - G1GC
      - 6u23以后支持，不过目前还不成熟...



# 编写高性能Java代码

- JVM

- 内存管理

- `java.lang.OutOfMemoryError: {reason}`
      - GC overhead limit exceeded
      - Java Heap Space
      - Unable to create new native thread
      - PermGen Space
      - Direct buffer memory
      - request {} bytes for {}. Out of swap space?

# 编写高性能Java代码

- JVM

- 内存管理

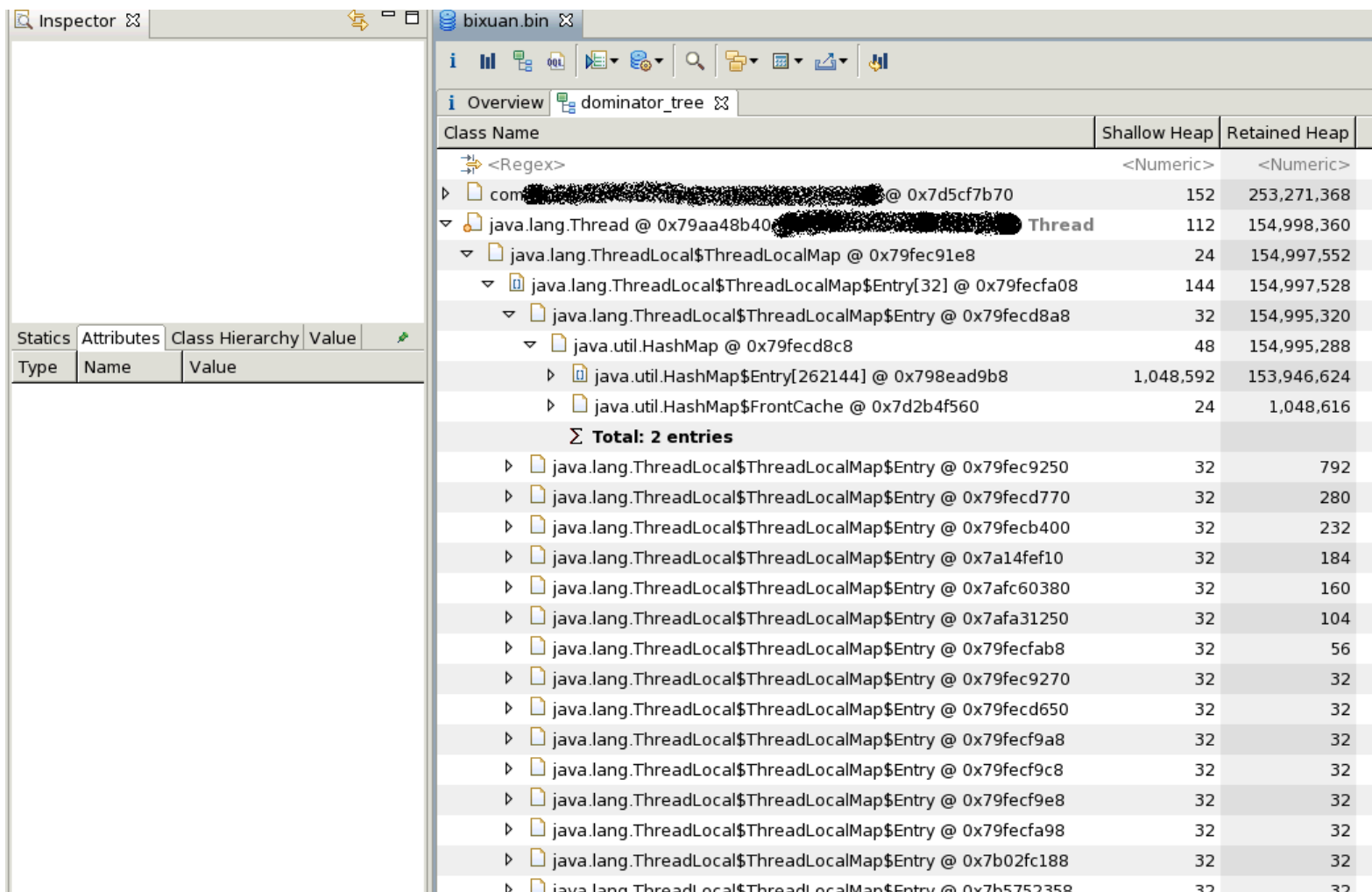
- 工欲善其事，必先利其器

- ps
      - -XX:+PrintGCDateStamps -XX:+PrintGCDetails -Xloggc:<gc 文件位置>
      - -XX:+PrintFlagsFinal (6u2l+)
      - -XX:+HeapDumpOnOutOfMemoryError
      - jinfo -flag
      - jstat
      - jmap
      - [MAT](#)
      - [btrace](#)
      - [google perf-tools](#)

# 编写高性能Java代码

- JVM
  - 内存管理
    - OOM
      - GC overhead limit exceeded || Java Heap Space
        - 首先要获取到heap dump文件
          - -XX:+HeapDumpOnOutOfMemoryError
          - jmap -dump:file=<>,format=b [pid]
          - from core dump
        - 接下去的解决步骤
          - Cases show

# 编写高性能Java代码



Inspector

bixuan.bin

Overview dominator\_tree

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com. [REDACTED] @ 0x7d5cf7b70	152	253,271,368
java.lang.Thread @ 0x79aa48b40	112	154,998,360
java.lang.ThreadLocal\$ThreadLocalMap @ 0x79fec91e8	24	154,997,552
java.lang.ThreadLocal\$ThreadLocalMap\$Entry[32] @ 0x79fecfa08	144	154,997,528
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x79fecd8a8	32	154,995,320
java.util.HashMap @ 0x79fecd8c8	48	154,995,288
java.util.HashMap\$Entry[262144] @ 0x798ead9b8	1,048,592	153,946,624
java.util.HashMap\$FrontCache @ 0x7d2b4f560	24	1,048,616
Σ Total: 2 entries		
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x79fec9250	32	792
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x79fecd770	32	280
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x79fecb400	32	232
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x7a14fef10	32	184
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x7afc60380	32	160
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x7afa31250	32	104
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x79fecfab8	32	56
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x79fec9270	32	32
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x79fecd650	32	32
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x79fecf9a8	32	32
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x79fecf9c8	32	32
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x79fecf9e8	32	32
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x79fecfa98	32	32
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x7b02fc188	32	32
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x7b5757358	32	32

# 编写高性能Java代码

- JVM
  - 内存管理
    - OOM
      - 比较难排查的java heap space oom Case...
        - 两种...

# 编写高性能Java代码

- JVM
  - 内存管理
    - OOM
      - PermGen Space
        - PermSize小了
        - ClassLoader使用不当
          - 经典的Groovy Case
        - 排查方法
          - -XX:+TraceClassLoading
          - btrace

# 编写高性能Java代码

- JVM
  - 内存管理
    - OOM
      - Direct buffer memory
        - -XX:MaxDirectMemorySize
        - 只有ByteBuffer.allocateDirect这里有可能抛出
          - 排查起来不会太复杂

# 编写高性能Java代码

- JVM
  - 内存管理
    - OOM
      - request {} bytes for {}. Out of swap space?
        - 只有Java crash才会看到: `hs_err_pid[$pid].log`
        - 原因可能是
          - 地址空间不够用
            - 32 bit
          - C Heap内存泄露
            - google perf-tools
            - 经典的Inflater/Deflater Case
            - Direct ByteBuffer Case



# 编写高性能Java代码

- JVM
  - 内存管理
    - GC调优
      - 到底什么算GC频繁?
      - 如何选择GC?
      - 为什么heap size $\leq$ 3G下不建议采用CMS GC?

# 编写高性能Java代码

- JVM

- 内存管理

- GC调优Cases

- 通常是明显的GC参数问题

- CMS GC

- 触发比率设置不合理导致CMS GC频繁的案例

- -Xmn设置不合理导致CMS-remark时间长的case

- swap case

- promotion failed cases

- 大对象分配

- 碎片

- Parallel GC

- Survivor区域大小调整的案例

- 悲观策略造成GC频繁的案例

# 编写高性能Java代码

- JVM

- 内存管理

- 编写GC友好的代码

- 限制大小的集合对象；
      - 避免Autobox；
      - 慎用ThreadLocal；
      - 限制提交请求的大小，尤其是批量处理；
      - 限制数据库返回的数据数量；
      - 合理选择数据结构。

# Java常见问题排查

- 类加载问题
- CPU高
- 内存问题
- Java进程退出
  
- 知其因 + 经验

# Java常见问题排查

- 类加载问题
  - ClassNotFoundException
  - NoClassDefFoundError
  - ClassCastException
  - 一个集群里有部分node成功、部分node失败的case
    - Java应用在linux环境下的常见问题...
  - 重要工具
    - -XX:+TraceClassLoading
    - 上面参数不work的时候btrace

# Java常见问题排查

- CPU高

- us高

- 特殊字符串引起的系统us高的case
      - top -H
      - 将看到的pid做十六进制转化
      - jstack | grep nid=0x上面的值，即为对应的处理线程
      - btrace看看有哪些输入的字符串

# Java常见问题排查

- CPU高

- us高

- 一行代码引发的杯具
    - public class CustomException extends Exception{
    - private Throwable cause;
    - public Throwable getCause(){
    - return cause;
    - }
    - }

# Java常见问题排查

- CPU高
  - us高
    - 还有可能
      - top,then l 如看到一直是其中一个cpu高，有可能会是gc问题，看看gc log
  - 最麻烦的case是cpu使用比较平均，每个线程耗一些，而且是动态的...

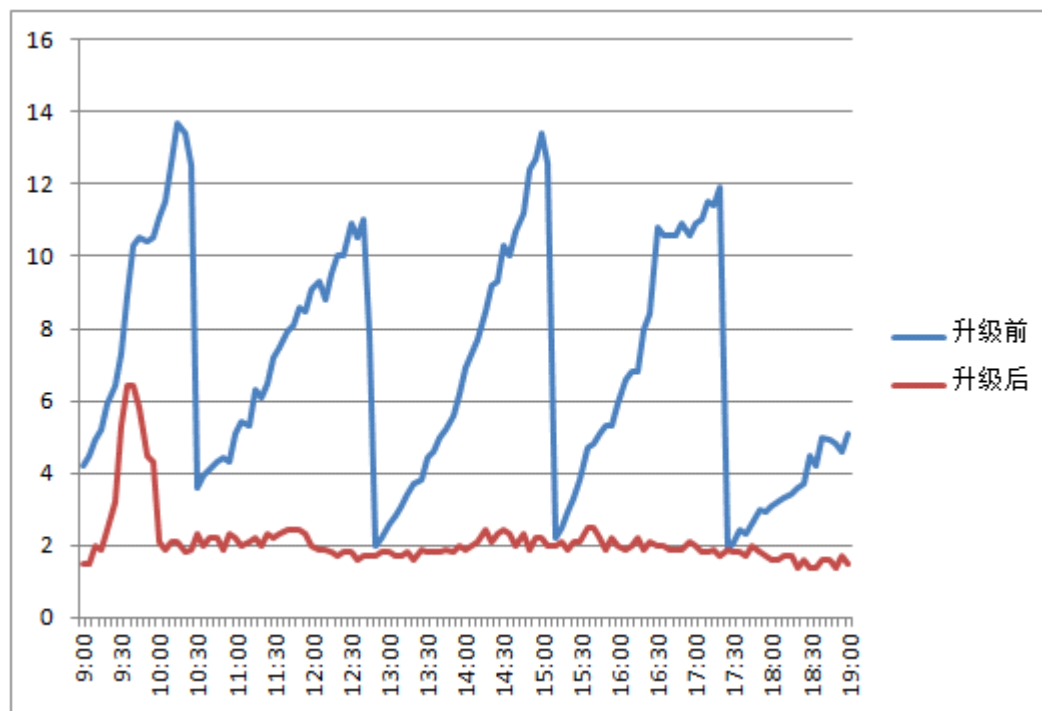


# Java常见问题排查

- CPU高

- us高

- 一个诡异的cpu us消耗的case



# Java常见问题排查

- CPU高

- sy高

- 线程上下文切换会造成sy高
      - 线程多
        - 一个误用netty client造成sy高的case
      - 锁竞争激烈
      - 主动的切换
        - 一个Case...
    - linux 2.6.32高精度定时器引发的sy高case

# Java常见问题排查

- CPU高
  - iowait高
    - 一个iowait高的排查case
      - 工具
      - 硬件

# Java常见问题排查

- Java进程退出
  - 原因非常的多
  - 首先要确保core dump已打开
  - dmesg
  - crash demo
    - jinfo -flag FLSLargestBlockCoalesceProximity <pid>

# Java常见问题排查

- Java进程退出

- native stack溢出导致java进程退出的case
- 编译不了某些代码导致的Java进程退出的case
  - -  
XX:CompileCommand=exclude,the/package/and/Class,methodName
- 内存问题导致的进程退出的case
- JVM自身bug导致退出的case

# 典型的互联网技术

- Google的发展历程
  - 1997年
    - Index Servers + Doc Servers
  - 1999年
    - Cache Cluster + Index Servers Cluster + Doc Servers Cluster
    - 自行设计服务器
    - Borg(可能是这年)
  - 2000年
    - 自行设计DataCenter, 降低PUE
  - 2001年
    - Index全部放入内存
  - 2003年
    - 经典的Google Cluster Architecture文章
    - GFS论文 (2001年上线)

# 典型的互联网技术

- Google的发展历程
  - 2004年
    - 发表MapReduce论文
  - 2006年
    - 发表BigTable论文（2003年上线）
  - 2007年
    - build索引时间缩短到分钟级
    - Service
  - after 2009
    - Colossus(NextGen GFS)、Spanner(NextGen Bigtable)、实时搜索、Omega(NextGen Borg)
    - Http协议改进、TCP/IP协议改进、图片格式改进

# 典型的互联网技术

- Facebook的发展历程
  - 成立之初
    - LAMP
  - LAMP + Memcached
  - LAMP + Memcached + Services
  - 2007年
    - HipHop
  - 2009年
    - BigPipe、Scribe、Haystack、Cassandra
    - 自行设计DataCenter
  - 2010年
    - HBase



# 典型的互联网技术

- Twitter的发展历程
  - 2006年诞生时
    - Ruby On Rails + MySQL
  - 2007年
    - 增加Memcached
  - 2008年
    - 往Java/Scala迁移
    - Service化
    - 尝试Cassandra、Redis
  - 2010年
    - 自建DataCenter

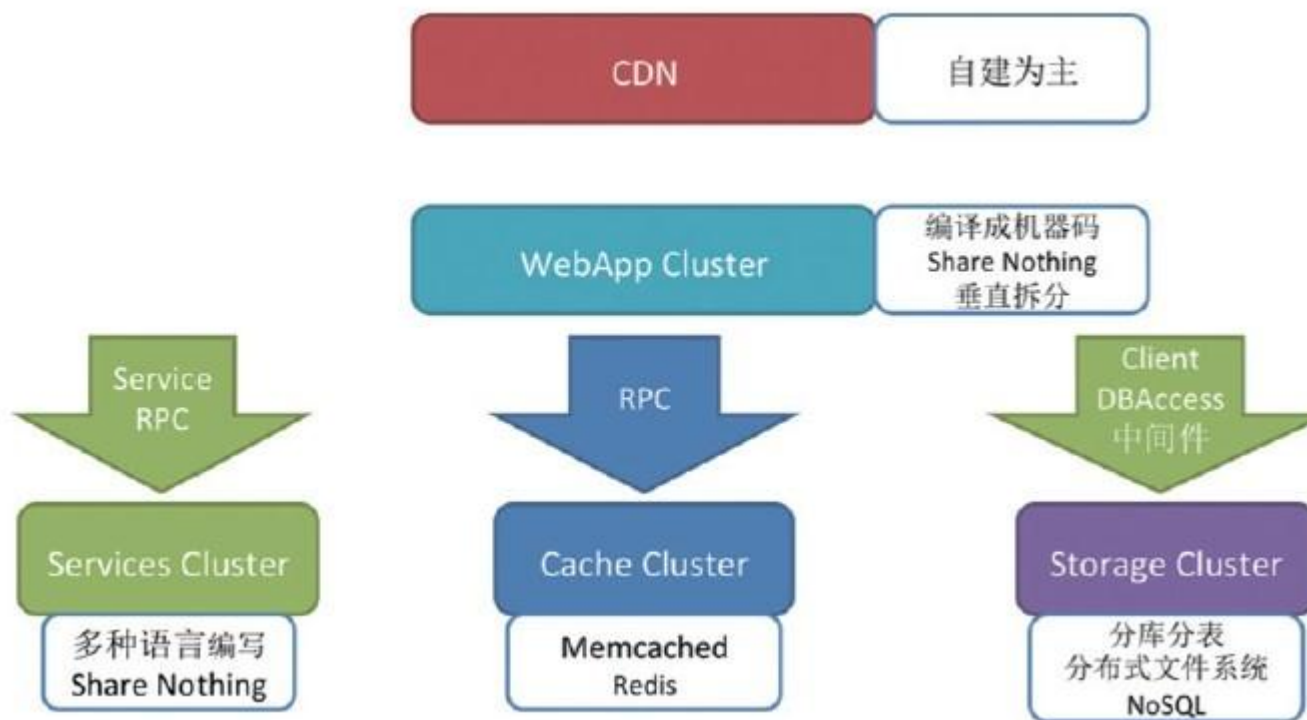
# 典型的互联网技术

- eBay的发展历程
  - 1995年
    - CGI + GDBM, 最多5w在线商品
  - 1997年
    - freebsd迁移到windows, GDBM-->Oracle
  - 1999年
    - Cluster + 小型机
  - 2001年
    - 分库分表, 小型机升级
  - 2002年
    - 迁移到Java

# 典型的互联网技术

- 有趣的现象
  - 语言方面
    - 混合
    - VM
    - 团队基因
  - Cache
  - Service体系
  - 存储方面
    - MySQL（分库分表）、NoSQL、自研发的
    - 分布式文件系统
  - 硬件
    - 大量廉价的机器
    - 自建DataCenter
  - 对业界的技术发展起到了巨大的推动作用。

# 典型的互联网技术



# 典型的互联网技术

- 面临的问题
  - 大访问量、大数据量
    - 可伸缩、成本控制
  - 访问速度
    - 高性能
  - 高可用

# 典型的互联网技术

- 可伸缩
  - 垂直伸缩
    - 可根据硬件自动调整
      - 例如 `Runtime.getRuntime().availableProcessors()`
  - 应用水平伸缩
    - 无状态
      - 最典型的问题：用户 session
      - 状态放在存储中
    - SOA
    - 技术较为成熟

# 典型的互联网技术

- 可伸缩
  - 存储可伸缩
    - 极度复杂
    - 分库分表
    - NoSQL的自动伸缩，例如HBase
      - 增加dataserver即可增加存储空间
      - 增加regionserver即可增加支撑的qps
      - 但真的这么完美吗？

# 典型的互联网技术

- 高性能
  - Cache，能Cache的全部Cache
    - 绕过Service
  - 静态化，能静态的全部静态
    - 全静态
    - 动静结合
  - CDN
  - 根据鼠标行为的加载
  - 适当的“欺骗”
    - youtube的技巧



# 典型的互联网技术

- 高可用
  - SPoF (Single Point of Failure)
    - HA
    - 集群
      - 负载均衡 (硬件、软件)
    - 还有更广的概念
      - 单点机柜、网络、机房、运营商、城市

# 典型的互联网技术

- 高可用
  - 监控
    - 系统的运行状况
    - 异常状况的准确体现
    - 并不好做
      - 准实时的数据采集/分析
      - 报警：如何有效的报警
        - 单机/集群合并
        - 多故障合并等等

# 典型的互联网技术

- 高可用
  - 灰度

# 典型的互联网技术

- 高可用
  - 异步（解耦）
    - 主逻辑同步，次要逻辑异步
      - 前端： ajax
        - case show
      - 后端： 消息

# 典型的互联网技术

- 高可用
  - Keep Simple
    - 最高境界，但也最难做到
    - 不采用很炫但复杂的技术
    - 一个故障N多的系统改造成几乎无故障的系统的Case

# 典型的互联网技术

- 高可用

- 隔离

- 分离系统中的各种操作
      - 拆分系统（重要功能和不重要功能分开）
        - 一个不重要系统引发的重要系统挂掉的Case
        - 系统要做分级，重要级别的系统不能依赖非重要的系统
      - 七层路由
        - 分离耗资源的操作和不怎么耗资源的
          - 查询拖S整个系统的Case
        - 分离重要功能和不重要功能
          - 给内部用的功能拖S整个系统的Case

# 典型的互联网技术

- 高可用
  - 容灾
    - 超时
      - 太多这类case了
    - 降级
      - 有效的降级保障系统不受影响的Case
      - 少写一个catch造成的严重故障Case
      - 自动降级保障系统的Case
      - 用户体验降级保障活动的Case
    - 自恢复
      - 自动重连等
    - 更广范围的容灾
      - 硬件、机房、城市

# 典型的互联网技术

- 高可用
  - 自我保护
    - 处理能力保护
      - 容量规划
    - 负载保护



# 典型的互联网技术

- 高可用
  - Full Stack
    - 从头到尾的技术掌握

# 典型的互联网技术

- 成本控制
  - 有些性能优化通常会带来成本下降
    - 例如像google改进tcp/ip协议、webP等
  - 虚拟化
    - 关键手段
  - 自定义硬件/DataCenter
  - “云”