
Java
Performance Tuning Guide 1.3
By 江南白衣

ABOUT ME

70后，喜欢编码的架构师

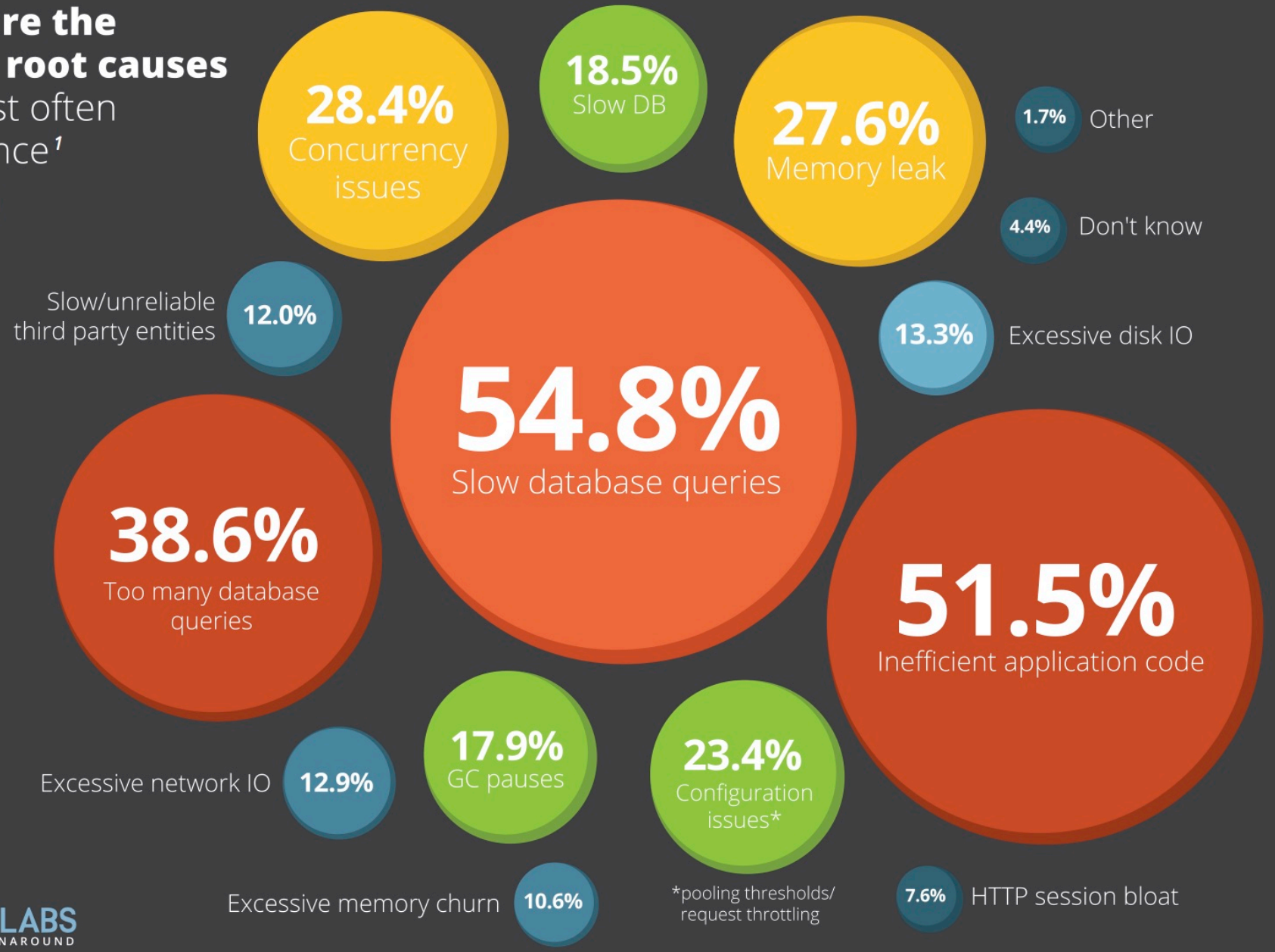
唯品会平台架构部

SpringSide 春天的旁边

才子词人，自是白衣卿相。
青春都一饷。忍把浮名，换了浅斟低唱。
--柳永《鹤冲天》

What are the typical root causes you most often experience[1]

Figure 1.16



REBELLABS by ZEROTURNAROUND

[1]*Answers were mulitple choice, so the numbers don't add up to 100%. Deal with it :)*

CONCENTS

BASIC RULES

Don’t Trust it, Test it

Show me the code

简单即正义

网上的信息很丰富 ？

Java已发展十几年，不同版本的新旧信息全堆在一起。

谁都可以把自己的理解放到网上，然后被各种网站转载，转载。。。但缺少一个纠错的机制。

这份幻灯片也会有同样的过时或有失偏颇的命运。

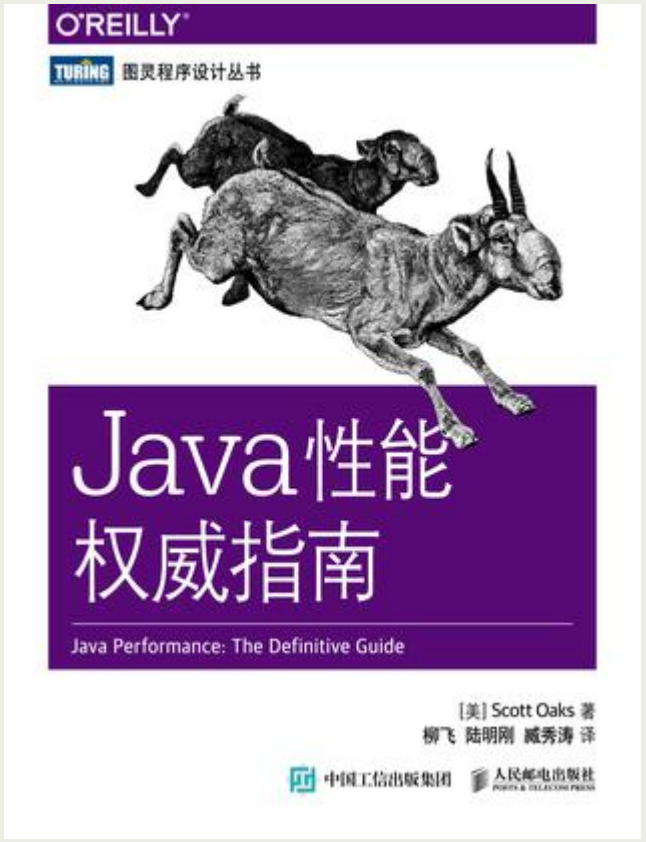
一些过时的经典语录

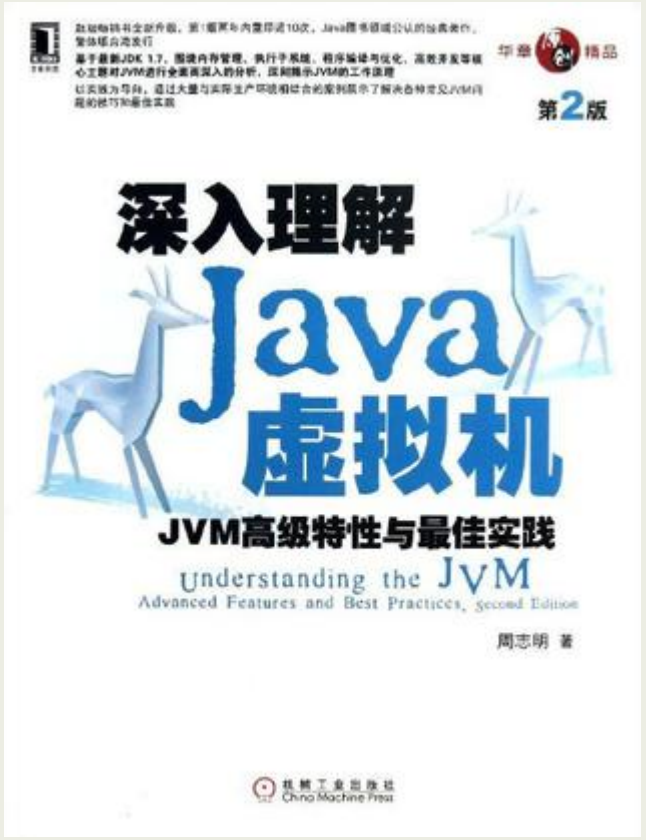
函数参数设为 final 能提升性能

不使用getter/setter，直接访问属性提升性能
设置+XX:+UseFastAccessors 提升getter性能

将变量设为Null能加快内存回收

可靠资料推荐





R大
RednaxelaFX，莫枢， Azul JVM
JavaEye，知乎

[知乎：R大是谁]

笨神
你假笨，寒泉子，阿里JVM
公众号：你假笨

毕玄
Bluedavy，阿里
公众号：HelloJava

那些老外大能

http://java-performance.info/

基本原则

简单即正义

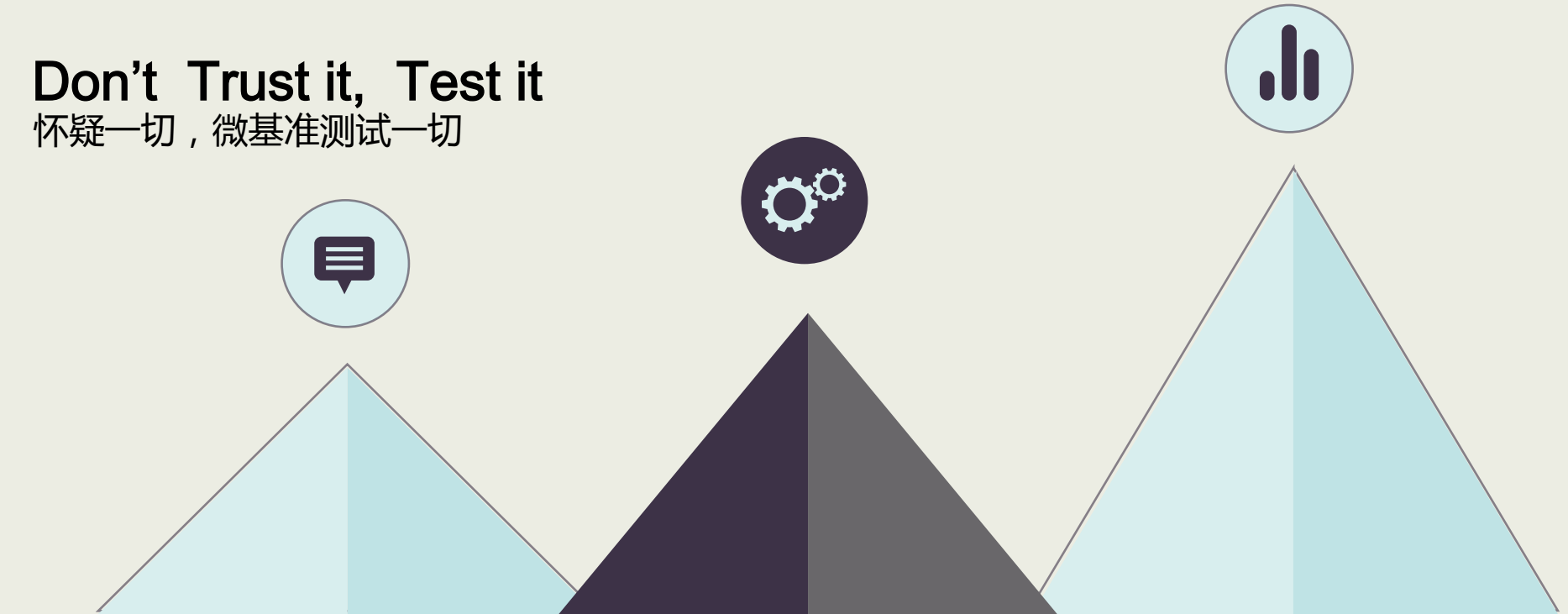
To be or not to be?

Show me the code

阅读JDK的代码，阅读一切的代码

Don’t Trust it, Test it

怀疑一切，微基准测试一切



测试一切 - 微基准测试的要点

没有测试数据证明的论断，都是 **可疑** 的。

没考虑到以下要点的测试，都是 **不可靠** 的。

01

预热

JIT编译优化

触发JIT的调用次数
后台编译的时间

02

防止JIT消除无用代码

```
for(...) {
    myMethod();
}
```

如果myMethod()对上下文代码没有产生作用

03

避免其他干扰

测试数据生成的时间
如调用random()

GC ，除非也要测它

简化基准测试编写 - JMH

像Junit那样，编写测试代码，标注annotation，JMH完成剩下一切

http://openjdk.java.net/projects/code-tools/jmh/

微基准的要点

运行预热循环
与被测函数交互防止无用代码
数据准备接口
迭代前主动GC

并发线程控制

同步开始与结束的时间

结果统计与打印

QPS，响应时间，分位数响应时间
监控GC，热方法等
Introduction to JMH Profilers

使用JMH进行微基准测试：不要猜，要测试！ - 译by 张洪亮

JMH Code Example

```java
@BenchmarkMode(Mode.SampleTime)
@Warmup(iterations = 5)
@Measurement(iterations = 10, time = 5, timeUnit = TimeUnit.SECONDS)
@Threads(24)
public class SecureRandomTest {
    private SecureRandom random;

    @Setup(Level.Trial)
    public void setup() {
        random = new SecureRandom();
    }

    @Benchmark
    public long randomWithNative() {
        return random.nextLong();
    }

    @Benchmark
    @Fork(jvmArgsAppend = "-Djava.security.egd=file:/dev/./urandom")
    public long randomWithSHA1() {
        return random.nextLong();
    }
}
```

JMH Result Example

```
# Warmup Iteration 1: 0.057 ±(99.9%) 0.003 ms/op
…
# Warmup Iteration 5: 0.053 ±(99.9%) 0.002 ms/op

Iteration   1: 0.062 ±(99.9%) 0.002 ms/op
…
Iteration  10: 0.057 ±(99.9%) 0.004 ms/op

 Histogram, ms/op:
    [ 0.000,  5.000) = 3997132
    [ 5.000, 10.000) = 3938
    [10.000, 15.000) = 1074
    …..
```

Benchmark

randomWithNative	2946835	count
randomWithNative·p0.00	≈ 10^{-3}	ms/op
randomWithNative·p0.50	0.001	ms/op
randomWithNative·p0.9999	6.824	ms/op
randomWithNative·p1.00	278.397	ms/op
randomWithSHA1	6151975	count
randomWithSHA1·p0.00	≈ 10^{-4}	ms/op
randomWithSHA1·p0.50	0.002	ms/op
randomWithSHA1·p0.9999	6.298	ms/op
randomWithSHA1·p1.00	391.459	ms/op

Show me the code - JDK源码

JDK的代码，其实比很多开源项目 工整，易读

是否看 com.sun.*， 与C写的native方法，是一个 分水岭

翻源码的例子： SecureRandom的江湖偏方与真实效果
Netty SSL性能调优

根据tags下载对应小版本的zip包
http://hg.openjdk.java.net/jdk7u/jdk7u/hotspot/
http://hg.openjdk.java.net/jdk7u/jdk7u/jdk/

ScheduledThreadPoolExecutor的故事

超时控制：发出请求时创建定时任务，成功收到返回时取消该任务

YoungGC 变慢 ？？

```
 num     #instances         #bytes  class name
----------------------------------------------
   1:         96891        6976152  java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask
   2:         26782        3528648  <constMethodKlass>
   3:         26782        3436768  <methodKlass>
   4:          2101        2517544  <constantPoolKlass>
   5:         96919        2326056  java.util.concurrent.Executors$RunnableAdapter
   6:         96890        2325360  com.vip.osp.core.base.SendMessage$2
   7:          4967        1806888  [Ljava.lang.Object;
   8:          2101        1571208  <instanceKlassKlass>
   9:          1806        1367680  <constantPoolCacheKlass>
```

为什么队列里的任务没被删除？
翻源码发现，基于成本考虑，task.cancel()默认只设置任务的标志位。
想真正在队列找出并删除一个任务，要设置Executor的RemoveOnCancelPolicy属性。

Slf4j的故事

```
logger.info(“Hello {}”,name);
```

有魔术吗？ 我得去学学 -- MessageFormatter类

```
for (L = 0; L < argArray.length; L++) {
        j = messagePattern.indexOf(“{}”, i);
        ……
}
```

好失望，查找，拼接再加上各种兼容处理，比自己拼字符串慢一截。

只有日志不确定会否输出时，才使用此经典写法。

要不要优化？ - 简单即正义

始终编写 **清晰，直接，易理解** 的代码

如果违背了简单性，则考虑 **复杂度** 与性能提升的 **性价比**

TUNNING JVM

JIT浅说

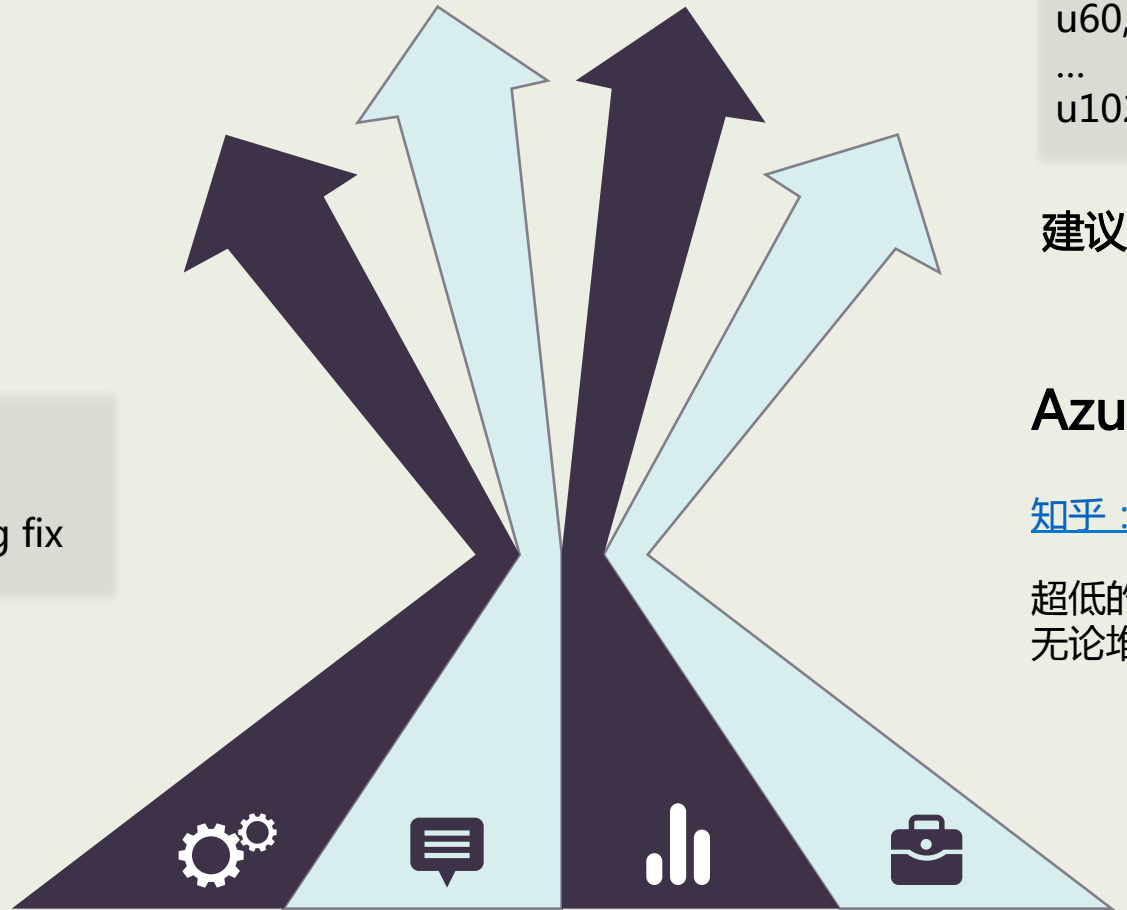
JVM参数

GC停顿

JDK的版本选择

Oracle JDK 大小版本总览

https://en.wikipedia.org/wiki/Java_version_history



JDK7

u40，集成JMC，621 bug fix
u60，性能优化完成, 130 bug fix
u80，最后一个免费版本, 104 bug fix

建议至少u60

JDK8

u20, 669 bug fix
u40, 645 bug fix
u60, 480 bug fix
…
u102, 118 bug fix

建议至少u60

Azul Zing

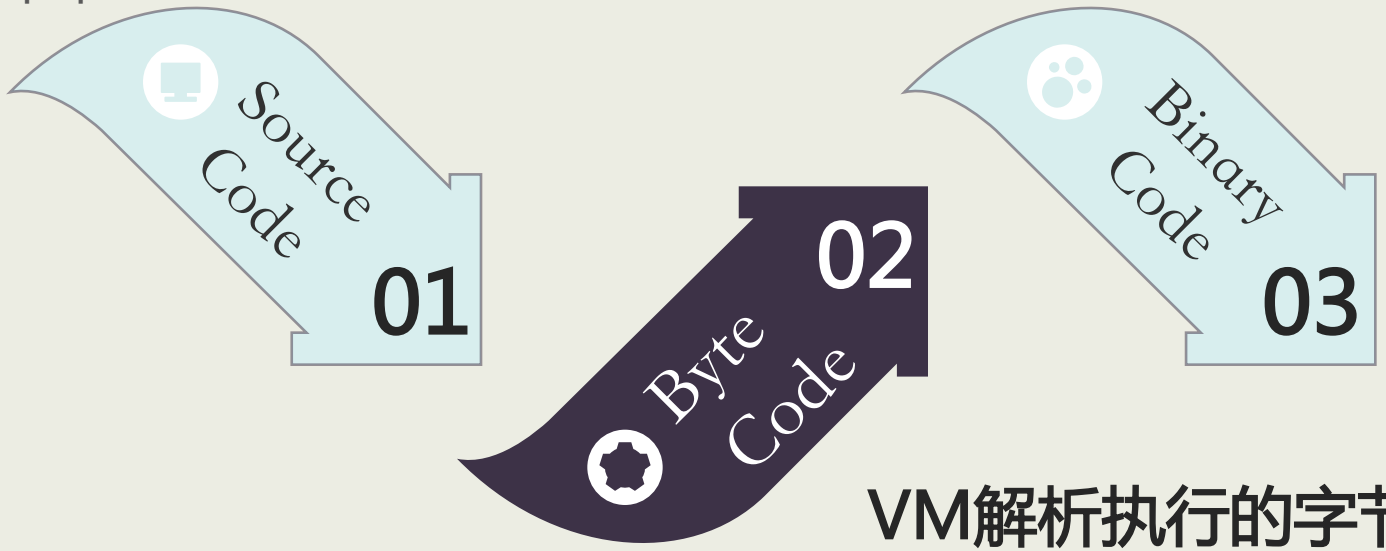
知乎：Azul Systems 是家什么样的公司

超低的GC停顿时间：
无论堆多大， 不调优<10ms, 调优 <1ms

JIT浅说 – 编译

文本源码

*.java
*.py
*.php



JIT - 机器直接执行的二进制码

Java 原生支持
Python 某些非官方版本
PHP 鸟哥在捣鼓

VM解析执行的字节码

*.class
*.pyc
opCache (php5.5+)

JIT浅说 – 优化

JIT更重要是代码的优化，JVM工程师十多年的积累与骄傲所在

函数内联

每个函数只有几行的 **优雅代码** 的基础

R大：内联是**JIT优化之母**

e.g. 没有充分内联就无法判断
微基准测试提到的代码是否真正无用

逃逸分析

对象有否逃逸出当前线程和方法

1.**同步消除**，e.g. StringBuffer

2.**标量替换**，只创建用到的对象属性
但Hotspot还未支持栈上分配对象

```
A a = new A(); //堆上分配的对象
a.b = 1;
 - >
int b = 1;        //栈上分配的原子类型局部变量
```

更多

无用代码消除
循环展开
空值检测消除
数组边界检查消除
公共子表达式消除
……

JIT优化项一览(2009)

JIT浅说 – 编译器

C1 编译器

无采样，立即编译
轻量优化

C2 编译器

64位 JVM默认编译器
采集1万次调用样本后深度优化

但每次GC，计算器会衰减一半
温热，总是达不到阀值，始终解析执行

禁止衰减：-XX:-UseCounterDecay

JDK8 多层编译

启动时，以C1编译
样本足够后，以C2编译

但，有时JDK8 反而比JDK7 略差？
有些函数C1编译后C2不再编译了

禁止多层编译：-XX:-TieredCompilation

JIT浅说 – 内联

内联条件

第一次访问：-XX:MaxInlineSize=35 Byte
频繁的访问：-XX:FreqInlineSize=325 Byte

最多18层内联，还有其他条件....

JITWatch

可视化JIT及内联的情况，提供优化的建议

https://github.com/AdoptOpenJDK/jitwatch/

让代码更容易被内联

怎样让你的代码更好的被JVM JIT Inlining By 戎码一生

1. R大最认可的缩短的方式 ：拆分不常访问的路径

```
if(most case){
   .....
} else {
   //e.g. 异常处理
    seldomAccessPath();
}
```

2. Final关键字有助内联？No，JIT有CHA等优化

JVM参数 – 原则

有没有一些好的开源例子？ Cassandra的 jvm.options

反面例子：无参裸奔的ZooKeeper

JDK的默认值不断变化， 参数间也互相影响

确认最终值的方法：

```
Java [生产环境参数] -XX:+PrintFlagsFinal –version | grep [待查证的参数]
```

JVM参数选释 – 性能

取消偏向锁：-XX:-UseBiasedLocking

JDK的偏向锁优化，但只适用于非多线程高并发应用

数字对象AutoBoxing的缓存：-XX:AutoBoxCacheMax=20000

JDK的默认值为-127- 128的数字，设置后我的程序增加2000QPS

关键业务系统的JVM启动参数推荐

JVM参数选释 – 监控

不忽略重复异常的栈 -XX:-OmitStackTraceInFastThrow

JDK的优化，大量重复的JDK异常不再打印其StackTrace。
但如果日志滚动了，当前日志里的NPE不知如何引起的

OOM时打印HeapDump : -XX:+HeapDumpOnOutOfMemoryError

Crash时输出Dump: -XX:ErrorFile 与 CoreDump

JVM参数 – 内存大小设置

一个2G堆大小的JVM，要预留多少内存？

堆内存 + 线程数 * 线程栈 + 永久代 + 二进制代码 + 堆外内存

2G + 1000 * 1M + 512M + 48/240M + (2G) = 5.75G (3.75G)

堆内存： 存储Java对象，默认为物理内存的1/64
线程栈： 存储局部变量（原子类型，引用）及其他，默认为1M
永久代： 存储类定义及常量池，注意JDK7/8的区别
二进制代码：JDK7与8，打开多层编译时的默认值不一样，从48到240M
堆外内存： 被Netty，堆外缓存等使用，默认最大值为堆内存大小

GC问题排查： 基本设置

Young GC

停顿时间与GC后剩下对象的多少，成 正比 关系

新生代的热身：把长久对象的升到老生代　　-XX:MaxTenuringThreshold，默认为15

Old GC

CMS　vs　G1　？　-　R大推荐 8G 为界

System.gc()

不要禁止:　-XX:+DisableExplicitGC（No！）
要CMS:　　-XX:+ExplicitGCInvokesConcurrent

常见误解

1. YGC不会STW，关注FULL GC即可
2. 混淆Old GC 与 Full GC

GC问题排查： 停顿时间

1. GC的停顿， 不止是垃圾收集的时间
2. JVM的停顿，不止是GC

   JIT，Class Redefinition(AOP)，取消偏向锁，Thread Dump...

在GC日志打印一切原因的，真实的停顿时间

```
-XX:+PrintGCApplicationStoppedTime
```

[安全点日志 - JVM的Stop The World，安全点，黑暗的地底世界]

GC问题排查：一条正常的YGC日志

[Times: user=0.29， sys=0.00，real=0.015 secs]
Total time for which application threads were stopped: 0.0154 seconds

sys cpu time = 0

real cpu time ≈ user time / GC线程数

application stopped time = real GC time

GC线程数 = 8 +（Processor - 8）（5/8）， e.g. 24核时为18

GC问题排查：一些意外的GC停顿

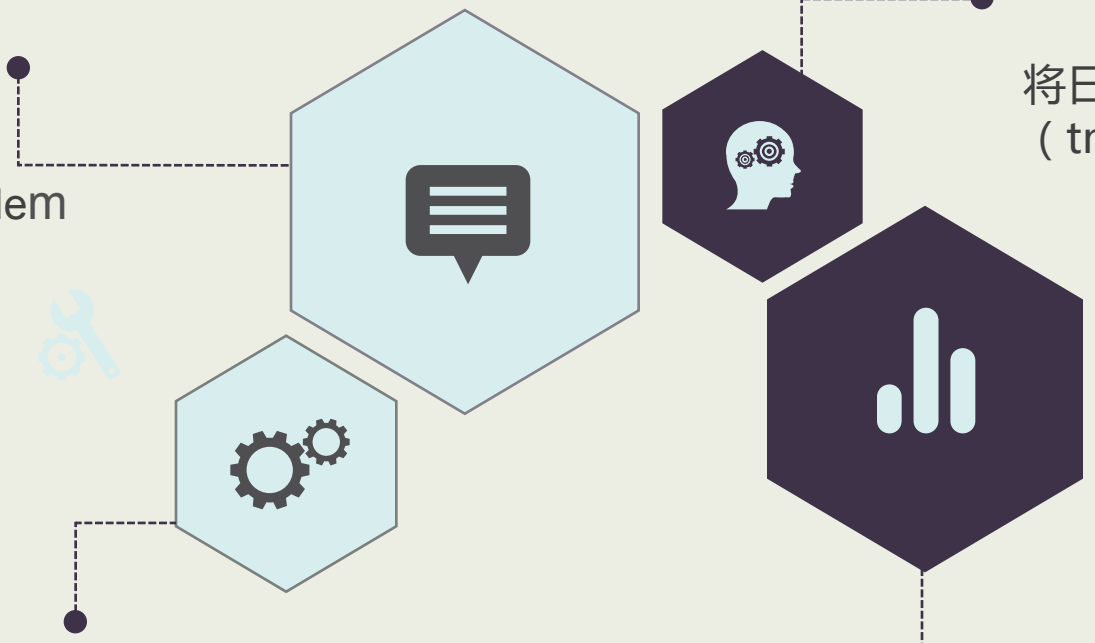
写入/tmp/hsperfdata文件被锁？

使用JMX代替jstat

-XX:+PerfDisableSharedMem

写入GC日志被锁？

将日志放入 /dev/shm目录
（ tmpfs内存文件系统 ）



使用了Swap？

vm.swappiness = 1

抢不到CPU？

检查有无其他后台辅助工具
偶发大量消耗CPU，用cgroup限制

[服务时延过长的三个追查方向]

JVM参数 – 完整例子

性能相关

```
-XX:-UseBiasedLocking  -XX:-UseCounterDecay  -XX:AutoBoxCacheMax=20000
-XX:+PerfDisableSharedMem -XX:+AlwaysPreTouch  -Djava.security.egd=file:/dev/./urandom
```

内存大小相关(JDK7)

```
-Xms4096m -Xmx4096m -Xmn2038m  -XX:MaxDirectMemorySize=4096m
-XX:PermSize=256m -XX:MaxPermSize=512m  -XX:ReservedCodeCacheSize=240M
```

CMS GC 相关

```
-XX:+UseConcMarkSweepGC  -XX:CMSInitiatingOccupancyFraction=75
-XX:+UseCMSInitiatingOccupancyOnly  -XX:MaxTenuringThreshold=6
-XX:+ExplicitGCInvokesConcurrent  -XX:+ParallelRefProcEnabled
```

JVM参数 – 完整例子(2)

GC 日志 相关

```
-Xloggc:/dev/shm/app-gc.log -XX:+PrintGCApplicationStoppedTime
-XX:+PrintGCDateStamps  -XX:+PrintGCDetails
```

异常 日志 相关

```
-XX:-OmitStackTraceInFastThrow -XX:ErrorFile=${LOGDIR}/hs_err_%p.log
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=${LOGDIR}/
```

JMX 相关

```
-Dcom.sun.management.jmxremote.port=${JMX_PORT} -Dcom.sun.management.jmxremote
-Djava.rmi.server.hostname=127.0.0.1 -Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

TUNING CODE

面向GC编程

并发，并发

杂项

面向GC的编程

Array Base的集合必须指定初始化大小

e.g. ArrayList, HashMap
大小不足时成倍复制扩容
注意Map的加载因子

引用设置为Null的传说

设为null已无用
缩小对象范围

对象重用

创建局部对象的成本远比想象的低
线程池，连接池，Netty堆外内存池

Thread Safe对象，全局重用, Gson
非Thread Safe对象，ThreadLocal重用, SimpleDateFormat

不可变对象的好处

老生代GC时减少扫描新生代
实际上的不变，与修饰符无关

内存分配的优化

栈上分配原始类型与引用
堆上分配对象
线程私有的TLAB堆区减少冲突

面向GC的Java编程 by 沐剑

面向GC的编程 – 更多抠门的优化

int vs Integer

4 bytes vs 16 bytes。
Java对象最小16 bytes， 12 bytes的固定header，按8 bytes对齐

AtomicIntegerFieldUpdater vs AtomicInteger

Netty大量使用此方法，代码非常复杂。
但如果海量对象，多个的AtomicInteger属性，用updater时可改为使用int

ArrayList vs LinkedList

Array-Based的容器，直接在数组里存放对象（4 bytes per element)，而且是连续性存储的(缓存行预加载)
Pointer-Based的容器，每个元素是Node对象，里面含真正对象及前后节点的指针(24 bytes per element)

面向GC的编程 – 高级Map

EnumMap

以枚举的oridnal()为下标来访问的数组，性能与空间俱佳

GuavaCache

支持并发的WeakReferenceHashMap

原子类型集合

Large HashMap overview: JDK, FastUtil, Goldman Sachs, HPPC, Koloboke, Trove

key为原子类型
哈希冲突从数组 + 链表 的 链表法
改为双数组的开放地址法

遗憾，不支持并发

高性能场景下，HashMap的优化使用建议

面向GC的编程 – 延时初始化

访问时始终有判断是否为空的成本， 适用于不一定需要创建的变量

1.Holder类静态变量

推荐，利用ClasssLoader加载类时的锁

```
private static class LazyObjectHolder {
      static final LazyObject instance
        = new LazyObject ();
}
```

2. 枚举

更好支持序列化，写法稍复杂

3. 正确写法的DoubleCheck

防止重排序，变量定义为volatile
使用临时变量，减少访问volatile变量

面向GC的编程 – 一些美丽的诱惑

String.intern()

字符串池，将重复的字符串转化为唯一引用

只适合长期存在的对象中，有海量重复的字符串

String.inter()怯魅

堆外内存是新大陆吗？

申请与释放的成本大，必须池化

ByteBuf.bytes<-> Object的序列化 or 按位访问的黑科技

StringBuilder的故事

生成 129 个字符的字符串要花多少内存？

```
StringBuilder sb = new StringBuilder();
for(int i=0;i<129;i++){
        sb.append("a");
}
```

从默认16字符开始，append()过程中，4次扩容复制，16 + 32+ 64 +128 + 256 = 496

还没完……

```
StringBuilder.toString() == return new String(value, 0, count)
```

总共需要 496+129 = 525 字符，并若干次 内存复制

StringBuilder的故事（2）

Liferay的StringBundler：

append 时先不往char[] 里添加，而是用String[] 暂存，
最后计算一个总长度再申请char[]，避免了扩容。

BigDecimal的StringBuilderHolder：

完全重用同一个StringBuilder，同一个char[]，
每次重置count属性即可。

[StringBuilder在高性能场景下的正确用法]

并发，并发，锁

01 分散锁

ConcurrentHashMap

分散成16把锁

LongAdder(JDK8)

代替AtomicLong
记数时分散多个AtomicLong
取值时将所有AtomicLong求和

02 分离锁

ReadWriteLock

多线程并发读锁，单线程写锁

BlockingQueue

ArrayBlockingQueue ：全局一把锁
LinkedBlockingQueue : 队头队尾两把锁

03 缩短锁

synchronized 尽量短的代码

但如果有多个断断续续的同步块，可考虑粗化合并

JODD Cache的故事

AbstractCacheMap

```
public V get(K key) {
    readLock.lock();  // 读写锁，可被并发读
    CacheObject<K,V> co = cacheMap.get(key);
    readLock.unlock();
}

public V put(K key, V object) {
    writeLock.lock();
    CacheObject<K,V> co = new CacheObject(…);
    // 循环遍历清理Key，但链表结构已被破坏
    if (isReallyFull())
        pruneCache();
    cacheMap.put(key, co);
    writeLock.unlock();
}
```

LRUCache基于LinkedHashMap

```
public V get(Object key) {
        Node<K,V> e=getNode(hash(key), key)
        //按访问次序调整元素在链表位置，不支持并发读
        if (accessOrder)
            afterNodeAccess(e);
        return e.value;
}
```

无锁(lock –free)

01 CAS

Compare And Set 自旋

Atomic* 系列

ConcurrentLinkedQueue

02 ThreadLocal

ThreadLocalRandom(JDK8)

Random 全局锁
ThreadLocalRandom 无锁

03 不可变对象

不修改对象属性，
直接替换为新对象

CopyOnWriteArrayList

[Java8中CAS的增强]

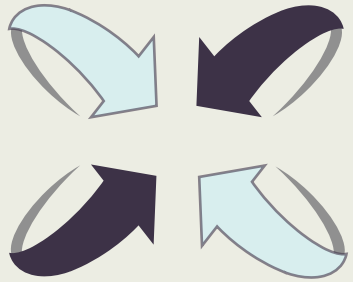
异步日志与无锁队列实现

同步的日志是应用的重要堵塞根源

Logback异步日志的差劲实现

使用ArrayBlockingQueue，每次插入前都询问Queue剩余容量

旧版只使用了堵塞的queue.put()



Log4j2的三种无锁队列选择

- JCToolsQueue
- DistruptorQueue
- LinkedTransferList

JCTools

- SPSC ： 单生产者单消费者队列
- MPSC： 多生产者单消费者队列
- SPMC： 单生产者多消费者队列
- MPMC： 多生产者多消费者队列

Distruptor

并发框架Disruptor译文 by 方腾飞

并发的其它话题（1）

ThreadLocal的代价

没有魔术，开放地址法的HashMap

Netty的 **FastThreadLocal**, index原子自增的数组

volatile的代价

Happen Before，保证可见性

每次访问比同步块开销小，但也不可忽视
可存到临时变量使用，直到可见性时才再次访问它

并发的其它话题（2）

ForkJoinPool的其他优势: 每线程独立任务队列

Netty的EventLoop， 同样由一组大小为1的线程池组成

Quasar 纤程：线程遇到阻塞时，主动去做其他事情

Java中的纤程库 - Quasar

性能优化的其它话题（1）

JNI调用C代码会比Java快吗？

写得好的Java代码JIT后，至少是和C代码一样快的，跨越JNI边界消耗更大

JNI主要用于操作系统的特定函数

匿名内部类会慢吗？

是否匿名，在字节码层面没有区别

是否静态内部类，区别是函数调用时多一个this的参数，影响不大

反射会慢吗？

很多框架都使用了反射，不要每次重新反射获取Method，Property

Dozer vs Apache BeanUtils（very slow）

性能优化的其它话题（2）

字符串处理

1. 很多操作的消耗都很大

format(), split(), indexOf(), replace()

避免或减少调用，预处理正则表达式
对比Commons Lang, Guava的实现

2. Encoding

UTF-8 vs ASCII
“UTF-8” vs Charset.UTF8

异常处理

1. 静态异常

创建异常时，获取StackTrace的代价非常大

Nety静态创建异常

The hidden performance costs of instantiating Throwables

TOOL BOX

JMC

BTrace

Profiler工具

调优，必须有 **针对性**

不能以 **黑盒的方式** ，盲目调优

锁，热点方法，对象内存分配

薛定谔的猫

Profiler工具的两大分类

Instrumentation - based

AOP式植入代码

能准确统计每个方法的调用次数，时间

性能成量级的衰退，使结果失去意义

植入代码导致方法膨胀，无法内联

Sampling - based

Thread Dump式采样 stack trace上的方法

只能统计方法的热点程度，相对百分比

性能损耗可忽略不计，< 10%

线下定位 - JMC

Java Mission Control， 原BEA JRockit 拳头产品

采样型的Profiler 工具， JDK7 u40 以上自带

开发环境免费，生产环境收费

Oracle Java Mission Control Overview

线上定位 – Btrace

BTrace是神器，每一个需要每天解决线上问题，
但完全不用BTrace的Java工程师，都是可疑的。
- by 凯尔文。肖

通过自己编写的脚本，attach进应用获得一切调用信息。

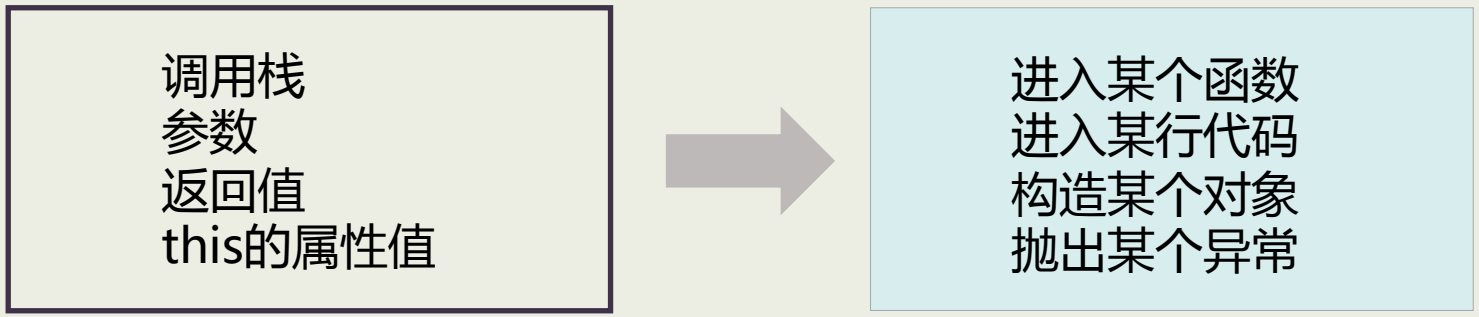
不再需要修改代码，加上System.out.println(),
然后重启，然后重启，然后 重启！！！

严格的约束，保证消耗特别小。
只要定义脚本时不作大死，直接在 生产环境 打开也没问题。

线上定位 – Btrace –典型的场景

1. 服务慢，能找出慢在哪一步，哪个函数里么？

2. 下列情况发生时，上下文是怎样的？



e.g.

什么情况下进入了这个处理分支？
谁调用了System.gc()？
谁构造了一个超大的ArrayList？

线上定位 – Btrace – 示例

打印实现了OspFilter接口的Filter链中，执行时间超过了1毫秒的Filter

执行命令：./btrace $pid HelloWorld.java

```
@OnMethod(clazz = "+com.vip.demo.OspFilter", method = "doFilter", location = @Location(Kind.RETURN))
public static void onDoFilter(@ProbeClassName String pcn, @Duration long duration) {
    if (duration / 1000000 > 1)
        println(pcn + ".doFilter:" + (duration / 1000000));
}
```

[Btrace入门到熟练小工完全指南]

其他工具

《另一份Java应用调优指南之 - 前菜》

《Java火焰图》 by 鸟窝

《Java问题排查工具箱》 by 毕玄

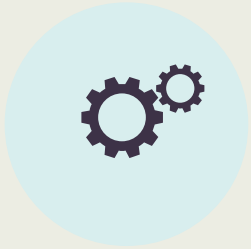
《Java应用线上问题排查的常用工具和方法》 by 沐剑

最后的话

调优是艺术，因为它源于深厚的知识，丰富的经验，和敏锐的直觉

\- 《Java性能权威指南》

本次分享只为大家在调优知识上作一个梳理



完成延伸阅读文章

养成微基准测试一切的习惯

养成阅读源码的习惯

公众号：春天的旁边