
Java
Performance Tuning Guide 1.8
By 江南白衣

ABOUT ME

70后，喜欢编码的架构师

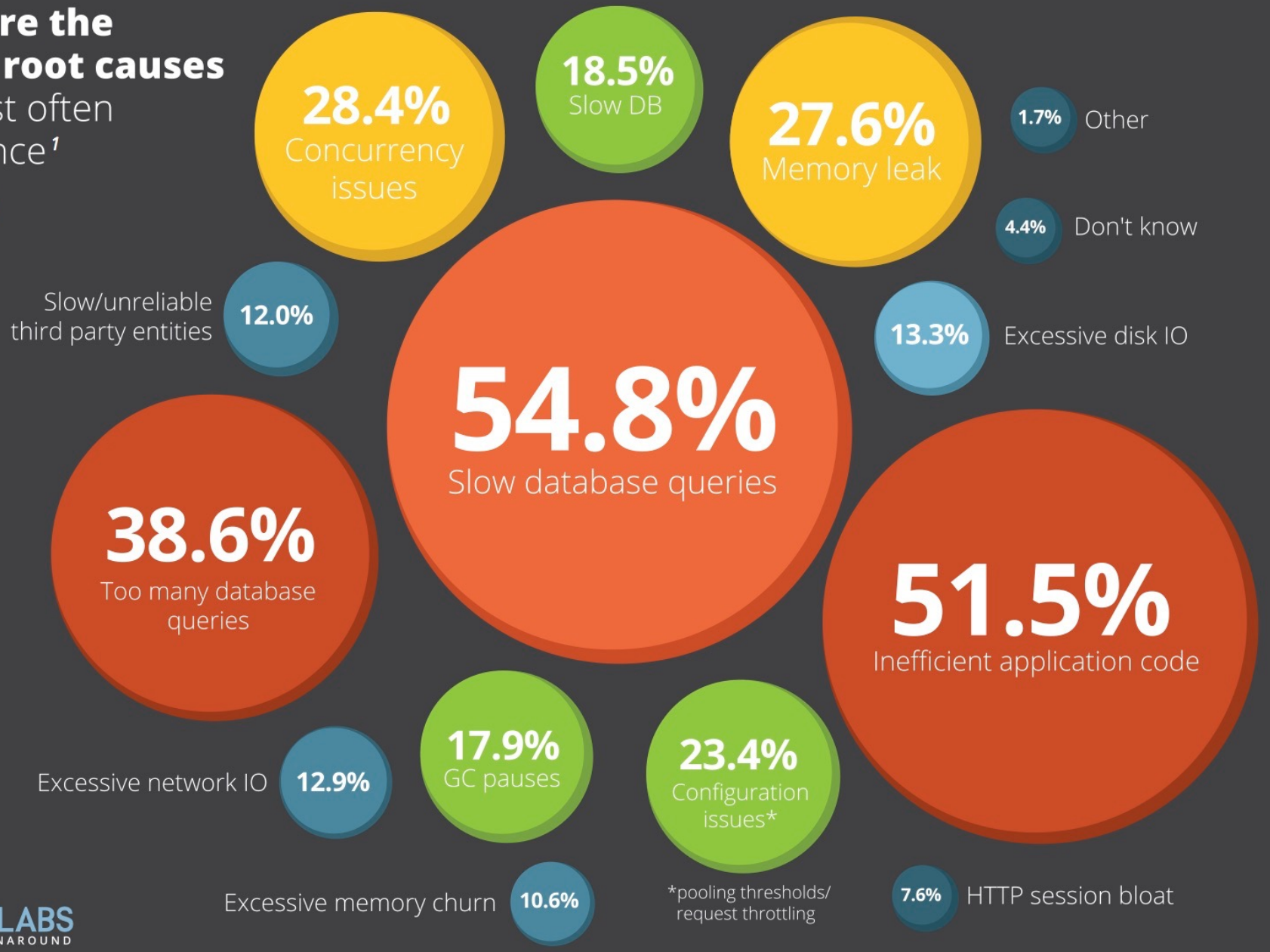
唯品会平台架构部

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。

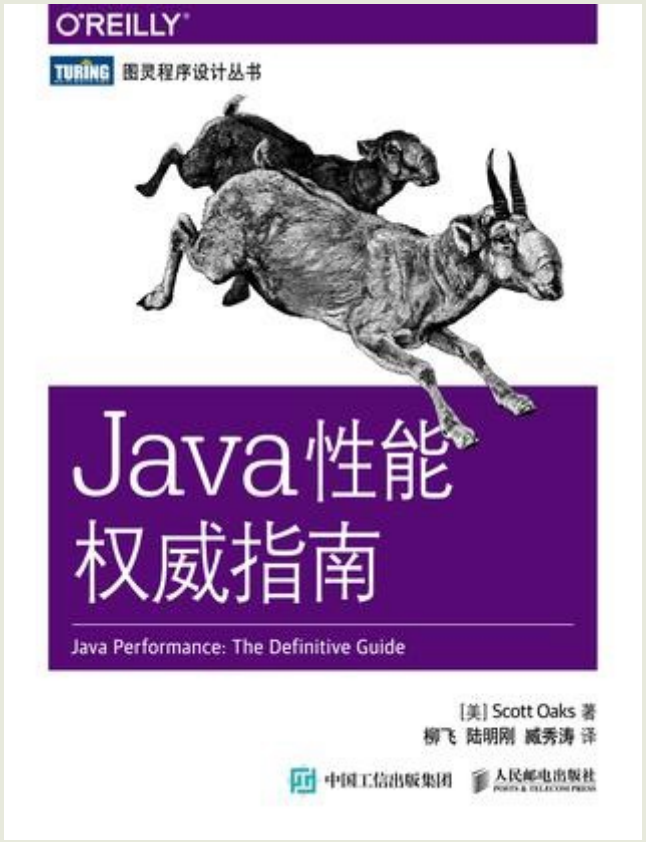
一些过时的经典语录

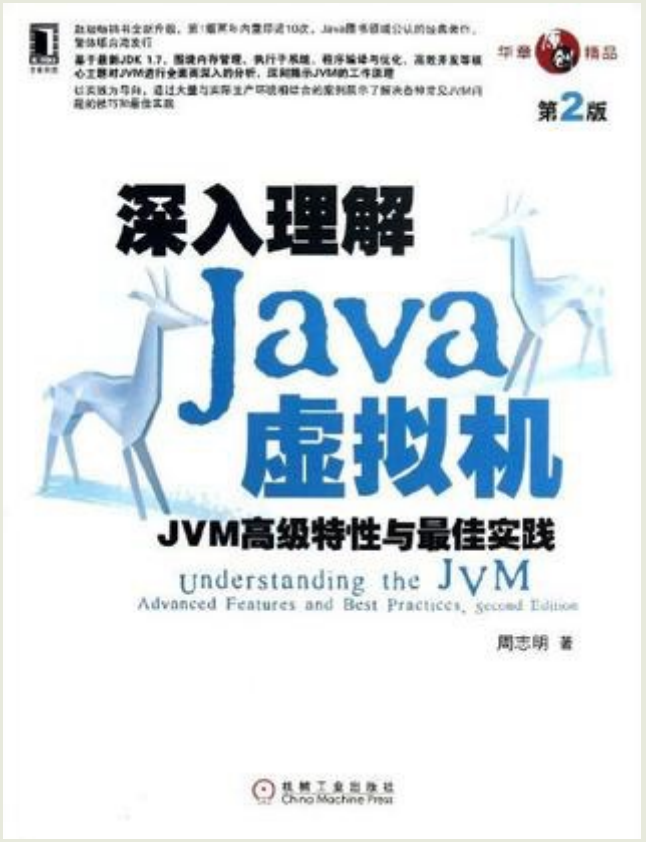
函数参数设为 final 提升性能

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

将变量设为 null 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

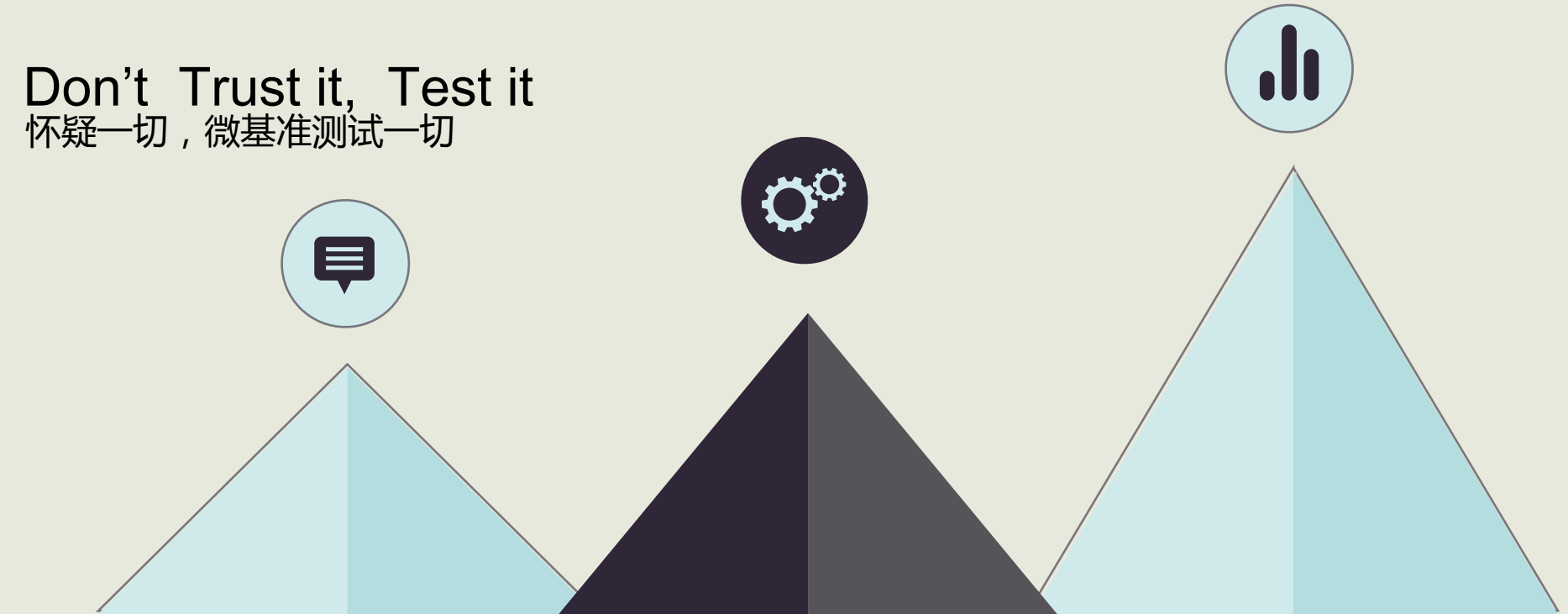
优化不优化?

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。

一个简单的main()，也是 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(...) {
  int i =str.indexOf(f);
}
```

如果循环内的代码
对上下文都没有作用

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 - 比较SecureRandom两种实现的性能

```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
    …..
```

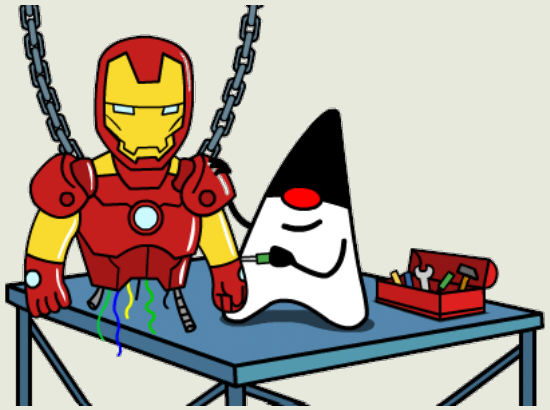
Benchmark

Native	2946835	count
Native·min	≈ 10^{-3}	ms/op
Native·p0.50	0.001	ms/op
Native·p0.9999	6.824	ms/op
Native·max	278.397	ms/op
SHA1	6151975	count
SHA1·min	≈ 10^{-4}	ms/op
SHA1·p0.50	0.002	ms/op
SHA1·p0.9999	6.298	ms/op
SHA1·max	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

遇到问题，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/

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，循环查找，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

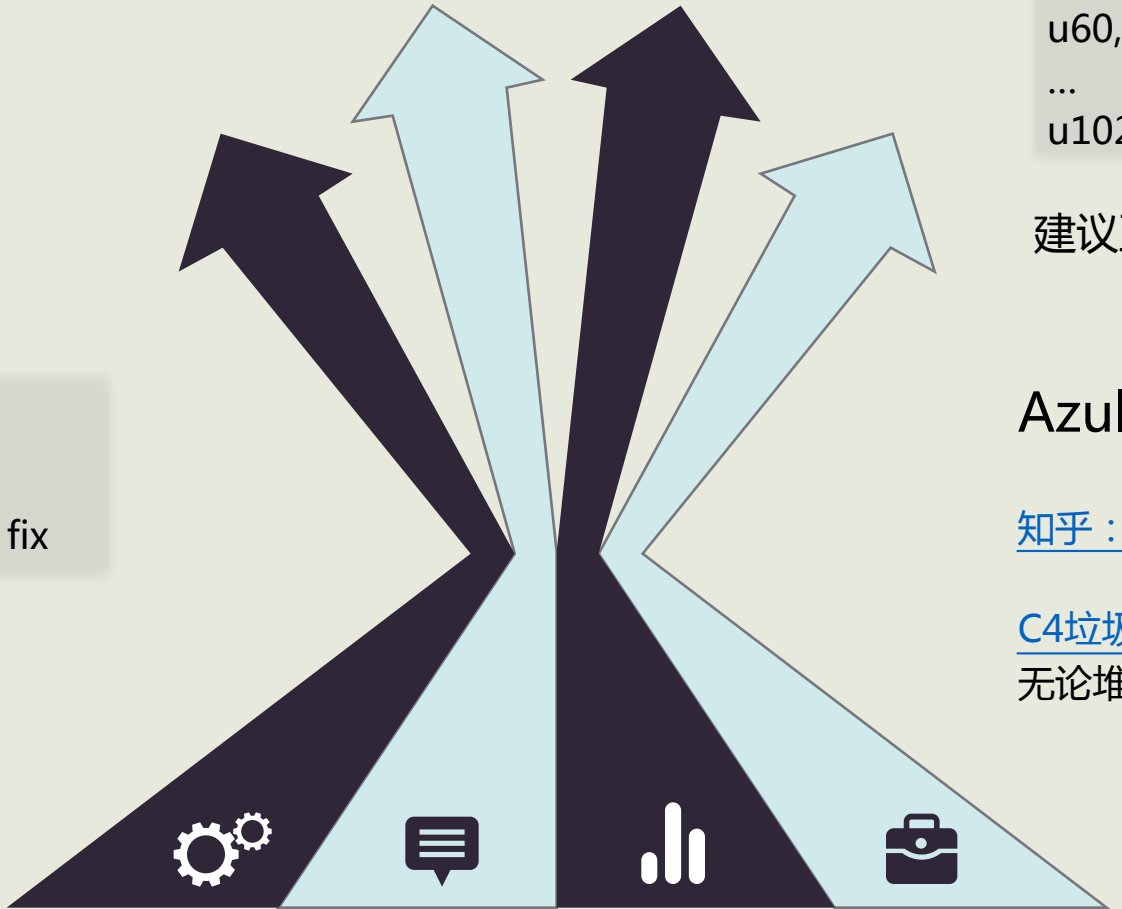
JVM参数

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

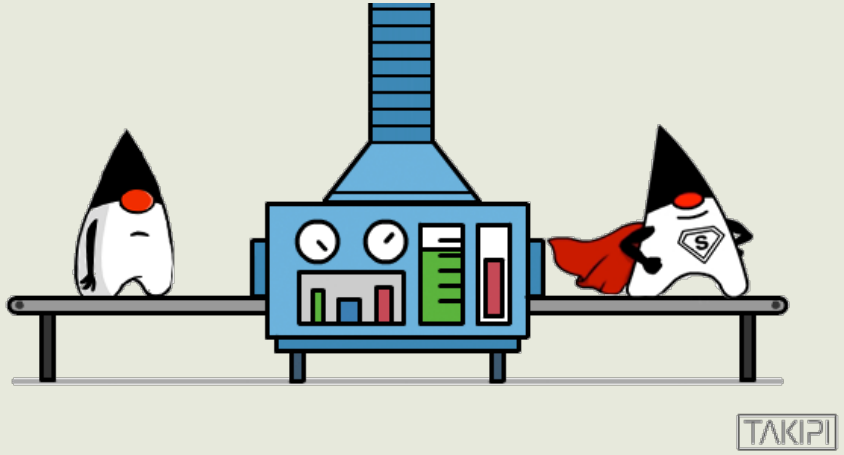
建议至少u60

Azul Zing

知乎：Azul Systems 是家什么样的公司

C4垃圾收集算法，超低的GC停顿时间：
无论堆多大， 不调优<10ms, 调优<1ms

JIT浅说



Java程序员需要知道的常识

JIT后，Java并不会比C++慢

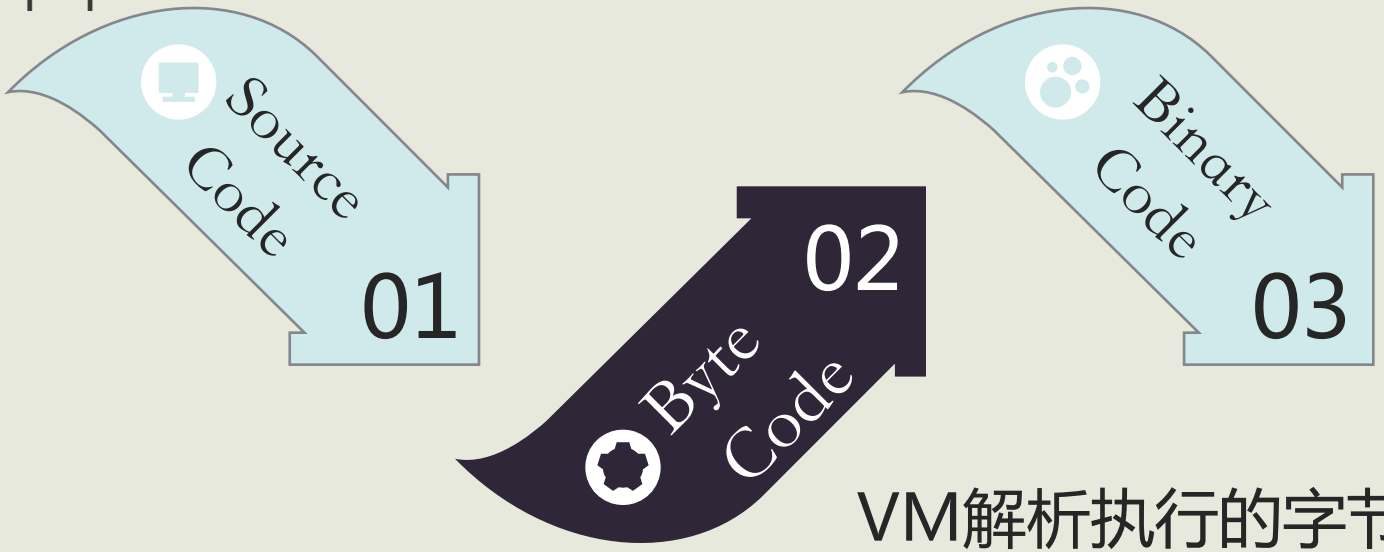
JIT浅说 – 编译

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 – 代码优化

JVM工程师十多年的积累与骄傲所在， 一些代码级的优化变得多余

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
微基准测试提到的无用代码是否真正无用

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，只创建对象的属性而不是对象

```
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？
可能有些函数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/

让函数更容易被内联

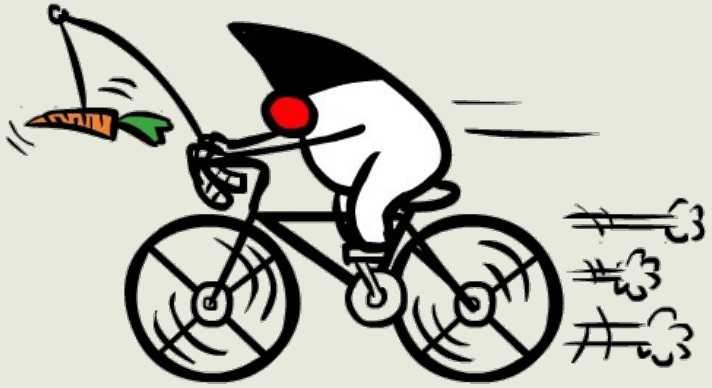
怎样让你的代码更好的被JVM JIT Inlining By 戎码一生

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等等优化

JVM参数调优



兼顾性能，稳定性，问题排查的便捷性

JVM参数 – 原则

有没有一些开源的例子？ Cassandra的 jvm.options

反面例子：无参裸奔的ZooKeeper

JDK的默认值，在小版本间不断变化，参数间互相影响

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，禁止它

数字对象的缓存：-XX:AutoBoxCacheMax=20000

int<->Integer，默认缓存-128～127，设置后我的应用提升4%QPS

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的启用

JVM参数 - 内存大小设置

一个2G堆大小的JVM，可能占多少内存？

堆内存 + 线程数 * 线程栈 + 永久代 + 二进制代码 + 堆外内存

2G + 1000 * 1M + 256M + 48/240M + (~2G) = 5.5G (3.5G)

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，默认最大值约为堆内存大小

Netty之Java堆外内存扫盲贴

GC停顿

Stop The World, Java程序员最头痛的事情

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

新生代的大小：GC间隔 大于 临时对象的生命周期，GC时只有少量存活对象

新生代的热身：把真正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

G1从理论到实现已忙活六七年，细节太复杂。

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 CMS GC(正常) 与 Full GC(坏的)

GC问题排查 - 停顿时间

1. GC的停顿， 不止是纯垃圾收集的时间

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 cpu time / GC线程数
- application stopped time ≈ real cpu time
- GC线程数 = 8 + (Processor - 8) (5/8)， e.g. 24核时为18

附：CMS GC日志

只观察不是 CMS-concurrent-* 的阶段

了解CMS 垃圾回收日志

GC问题排查 - 一些意外的GC停顿

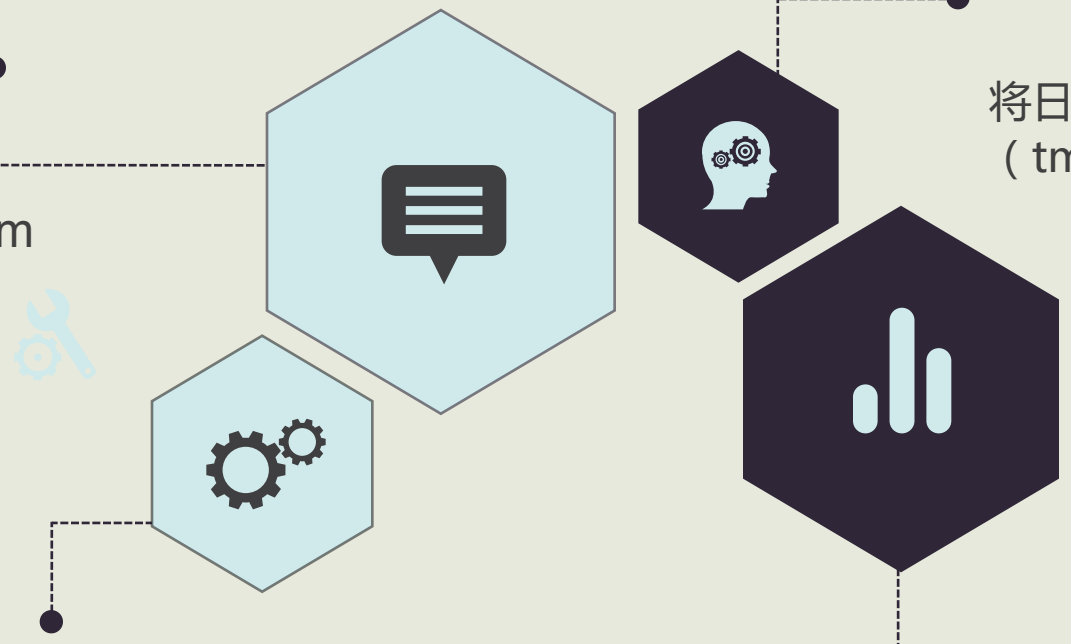
写入/tmp/hsperfdata文件被锁？

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，还有无数种的可能，八卦网上每一篇实战分析

JVM参数：完整例子

性能相关

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

内存大小相关(JDK7)

```
-Xms4096m -Xmx4096m -Xmn2048m  -XX:MaxDirectMemorySize=4096m
-XX:PermSize=128m -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
```

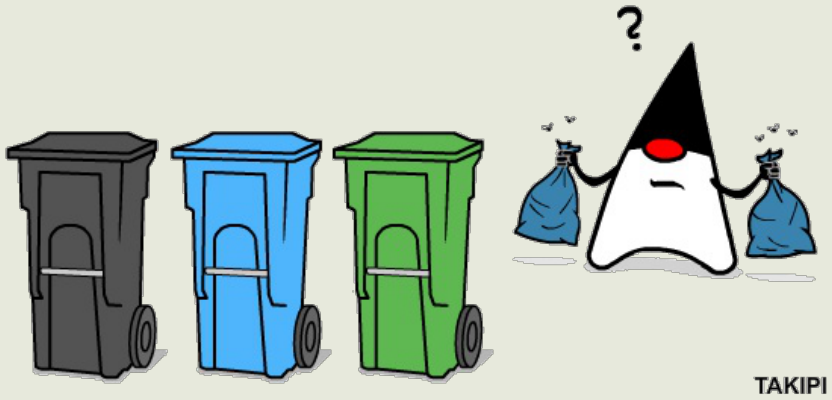
TUNING CODE

面向GC编程

并发，并发

杂项

面向GC编程



降低内存需求

减少GC频率

面向GC的编程

Array Base的集合必须指定初始化大小

e.g. ArrayList, HashMap

容量不足时会+50%复制式扩容

注意Map的75%加载因子

引用设置为Null的传说

缩小对象范围 (局部 vs 属性)

对象定义贴近使用的逻辑分支

WeakReference

流式处理

Big JSON

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, 如JSON Factory
非Thread Safe对象，ThreadLocal重用, 如SimpleDateFormat

不可变对象的好处

老生代对象的属性的生命周期一致
GC时减少扫描新生代
实际上的不变，与final无关

内存分配的优化

没有魔术，每条线程在堆区划分TLAB来减少分配冲突

栈上分配局部变量，无需GC
堆上分配静态变量，属性变量，对象

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
检查API接口，避免无谓转换

TAKIPI

AtomicIntegerFieldUpdater *vs* AtomicInteger

Netty大量使用此方法，代码非常复杂。
海量对象，多个数字属性时：int + 静态的updater vs AtoimicInteger

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)

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]

面向GC的编程 – Map家族

EnumMap

以枚举的oridnal()为下标来访问的数组，性能与空间俱佳

但，Map.get(Object o)接口导致多余的类型判断

GuavaCache

支持并发的WeakReferenceHashMap

优化hashCode()与equals()

只比较必要的属性
不可变对象缓存hashCode (e.g. String)

原子类型集合

大型HashMap评估：JDK、FastUtil、Goldman Sachs、HPPC、Koloboke与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，性能与空间俱佳

遗憾，不支持并发

高性能场景下，Map家族的优化使用建议

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DoubleCheckLocking

防止重排序，变量定义为volatile
还要把volatile变量存到临时对象提高性能

Java 单例真的写对了么? by 飒然hang

面向GC的编程 – 一些美丽的诱惑

String.intern() 字符串去重

字符串池，将重复的字符串转化为唯一引用

没有魔术，一个HashMap形式的字符串池
只适合海量长期存在的对象中，有重复的字符串

String.inter()怯魅

堆外内存是新大陆吗？

没有GC，没有堆大小限制了？

申请与释放的成本大，必须 池化

ByteBuffer.bytes()<-> Object的 序列化 or 按位访问的黑科技

并发与锁

减少锁

减少上下文切换

并发，并发，锁

01 缩短锁

synchronized 尽量短的代码块

保护数据而不是保护代码

但如果有多个断续的同步块
可考虑合并粗化

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 分散锁

ConcurrentHashMap

分散成16把锁

LongAdder(JDK8)

代替AtomicLong
计数时分散多个cell
取值时将所有cell求和

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的位置，不支持并发读
        if (accessOrder)
            afterNodeAccess(e);
        return e.value;
}
```

无锁(lock –free)/无等待(wait-free)

01 CAS

Compare And Set, 乐观锁

读无消耗，竞争时写有等待

Atomic* 系列

ConcurrentLinkedQueue
ConcurrentSkipListMap

JDK8中CAS的增强

02 ThreadLocal

ThreadLocalRandom(JDK8)

SimpleDateFormat in ThreadLocal

03 不可变对象

不修改对象属性
直接替换为新对象

CopyOnWriteArrayList

异步日志与无锁队列实现

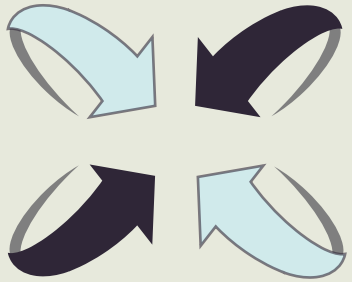
同步 日志是 堵塞之源

Logback异步日志的差劲实现

ArrayBlockingQueue
每次插入前还询问Queue剩余容量，建议重写。

Log4j2的三种无锁队列

- 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

剖析Disruptor:为什么会这么快？

- 锁的缺点
- 神奇的缓存行填充
- 揭秘内存屏障

并发的其它话题（1）

ThreadLocal的代价

没有银弹，开放地址法的HashMap
Netty的 FastThreadLocal, index原子自增的数组

重载initialValue()来初始终化值

volatile的代价

Happen Before

volatile *vs* Atomic*，可见性 *vs* 原子性

访问开销比同步块小，但也不可忽视
存到临时变量使用，需要确保可见性时再次赋值

并发的其它话题（2） - 线程池

1. 合理设置线程数

内存，线程调度的消耗不可忽略
CPU time +IO time/CPU time* 核数

2. JDK线程池

Java ThreadPool的正确打开方式

3. Tomcat的线程池

Tomcat，更符合想象可扩展线程池

4. 无队列锁的线程池

ForkJoinPool：每线程独立任务队列
Netty的EventLoop， 由一组容量为1的线程池组成

并发的其它话题（3） - 异步

1. Future style：伪异步

支持并行调用

调用Future.get() 时，阻塞调用线程

2. Callback Style：烧脑

响应式编程

Guava ListenableFuture，JDK8 CompletableFuture

RxJava

3. Quasar 协程：支持同步风格编码

少量线程处理海量请求

Java中的纤程库 - Quasar

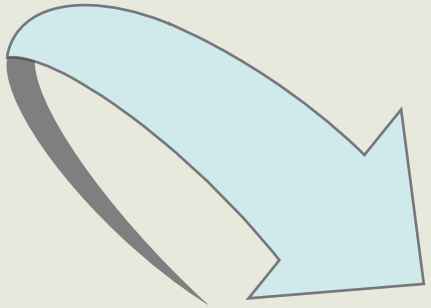
遇上阻塞，主动让出线程去做其他事情

阻塞返回，随便拉一条线程来再续前缘

并发的其它话题（4）

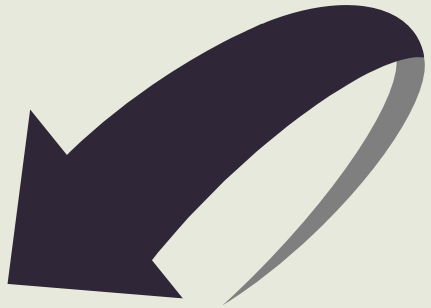
了解三方包的底层

System.getProperty( )



HashTable

Common Lang 3.4 FastDateFormat



StringBuffer

其他

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主要用于调用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

反射会慢吗？

很多框架都使用了反射，至少不要每次重新反射获取Method

Dozer vs Apache BeanUtils（very slow）

性能优化的其它话题（2）

1.字符串处理

很多操作的消耗都很大

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

字符 vs 字符串 vs 预编译的正则表达式

Commons Lang, Guava的实现

ASCII vs UTF-8

String.getBytes(Chartsets.UTF8) vs
String.getBytes("UTF-8")

2.异常处理

创建异常的Stack Trace消耗非常大

1. 静态创建异常，设一层栈
   错误信息会变化？ Clone该静态异常
2. 重载fillStackTrace()， 无栈

The hidden performance costs of instantiating Throwables

3. 正确的集合类型

关于Java集合的小抄

为90%的场景优化算法

if(a||b)

A与B的顺序，尽量让判断条件短路

延迟表达式的计算

SLF4J的著名例子：logger.info(“hello {}”,name);
Linkedin工程师是如何优化他们的Java代码的

另一个例子：带权重的随机负载均衡算法

先判断权重是否一致，99%情况下是一致的
直接使用不带权重的简单随机算法

for each 语法糖的故事

```
for(String name :list){
}
```

```
for (Iterator i=list.iterator(); i.hasNext(); ){
    String name = i.next();
}
```

语法糖编译后，等价于此写法

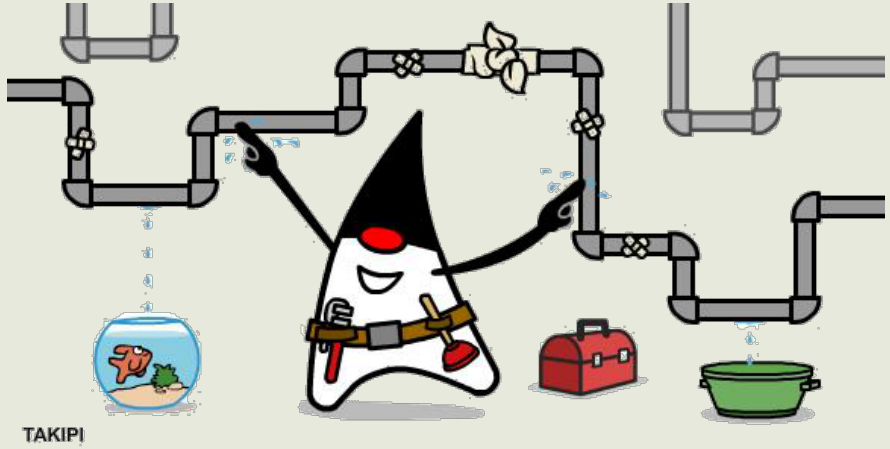
```
for(int i=0,size=list.size(); i<size; i++){
    String name = list.get(i);
}
```

RandomAccess接口的集合类此写法更快

节约一个Iterator的创建
ArrayList.get(i) vs ArrayList$Itr.next()

以防万一，循环开始前获取List的size

SonarQube



还有太多规则不能尽述

代码质量自动检查平台

http://www.sonarqube.org

规则集：

SonarQube Java Plugin : 代替 CheckStyle + PMD

FindBugs Plugin : FindBugs + FaceBook Contrib

从 Performance + Multi threading tag中自定义规则

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

改进实现，减少调用次数，替换实现

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。

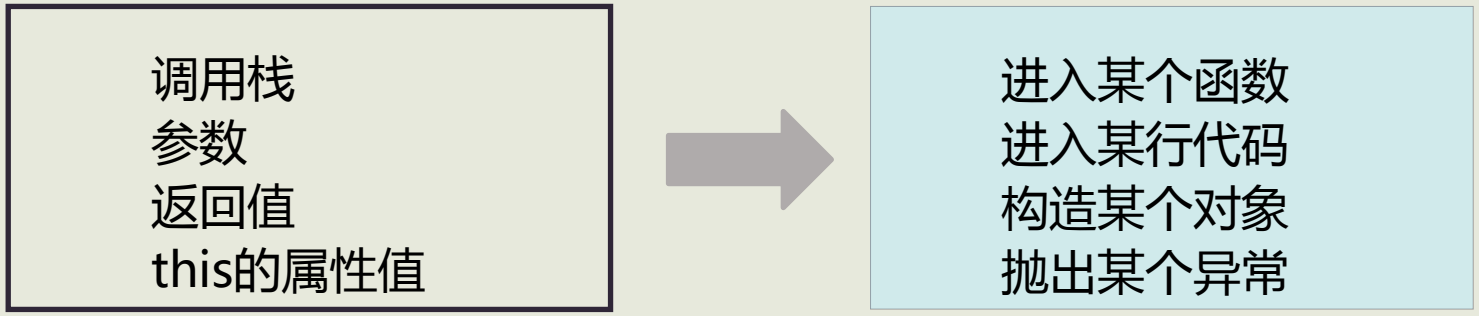
只要定义脚本时不作大死，直接在 生产环境 打开也没问题。

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？

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

Java火焰图 by 鸟窝

Java问题排查工具箱 by 毕玄

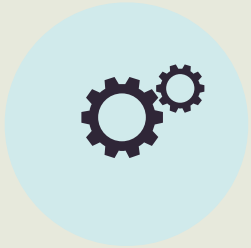
Java应用线上问题排查的常用工具和方法 by 沐剑

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

春天的旁边

微信号：jnby1978

主要更新记录

1.6 & 1.8 -2016.10.30

- P31 增加CMS日志说明
- P37 细化各种GC优化的说明
- P41 细化hashCode优化
- P47 细化CAS说明
- P52 细化线程池与异步，并拆分两页
- P55 细化异常处理的说明
- P56 增加延迟表达式计算
- P58 增加SonarQube
- PPT美化