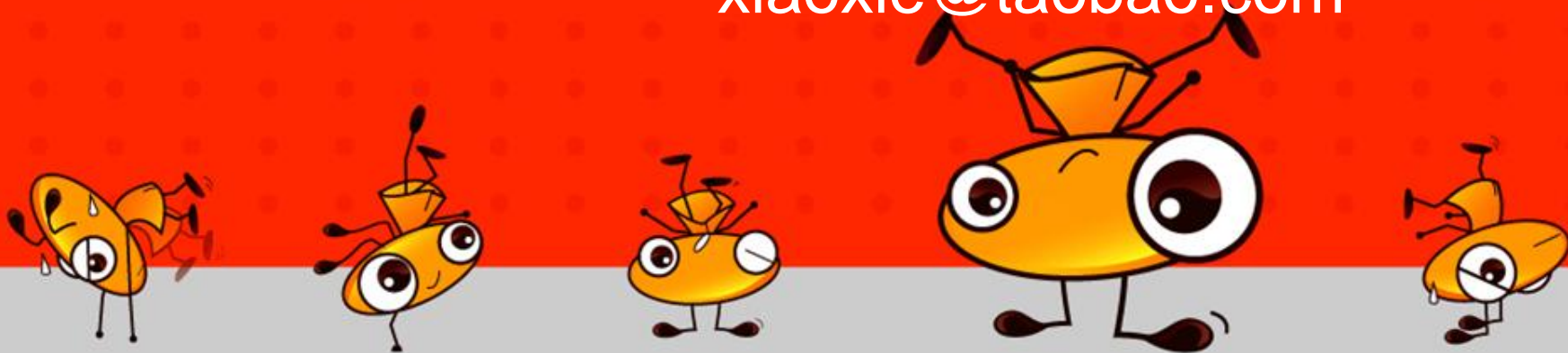


淘宝前台系统优化实践 “吞吐量优化”

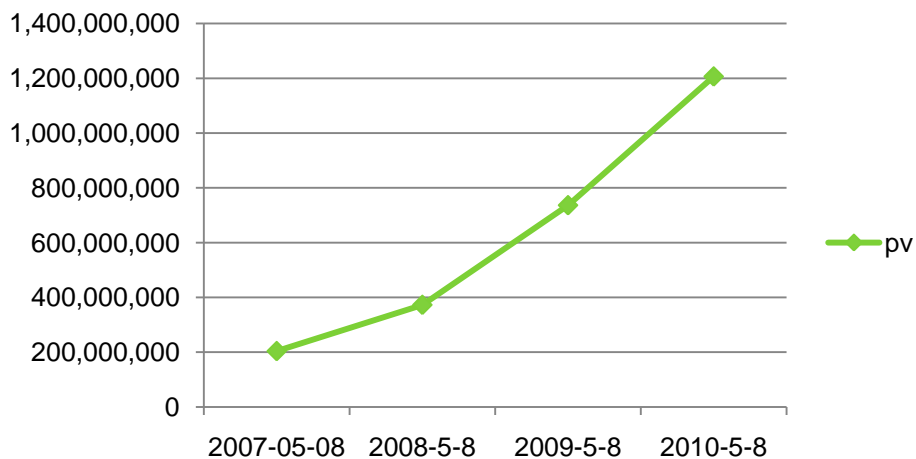
—— 淘宝网, 蒋江伟
—— xiaoxie@taobao.com



淘宝业务增加迅猛



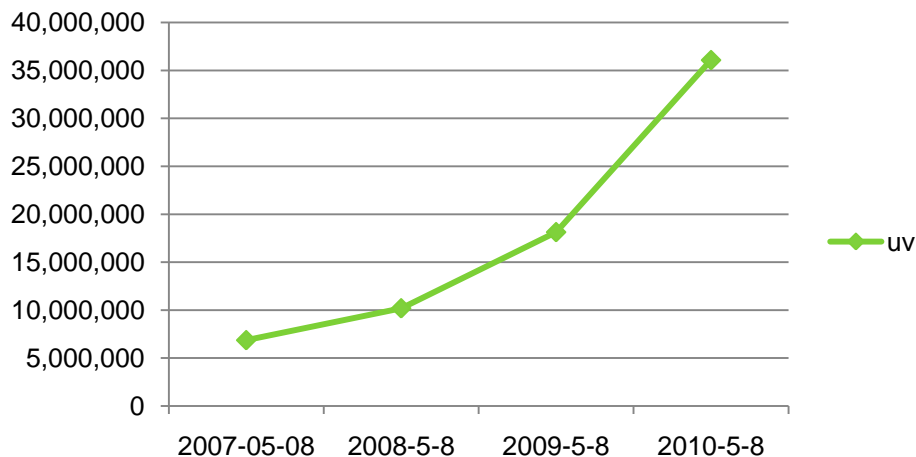
pv



淘宝网

淘宝商城
www.tmall.com

uv



某前台系统服务器数量



持续增加的服务器数量 VS 持续提升单机吞吐量



运维、管理
成本持续增加



增加单机吞吐量1倍，服务器数量减少1倍
在服务器数量达到一定级别的时候，非常合算

- 目的
 - 控制服务器增长数量
- 主题
 - 提升淘宝前台系统单服务器的QPS

- QPS (吞吐量) 三要素
- 优化模板
 - 至少提升50%
- 优化大数据的处理
 - 至少提升5%
- 优化jvm参数
 - 合理配置young区的大小 (0%~100%)
 - 减少GC的总时间
- 保持优化的成果
 - Daily load running
 - Daily hotspot code analysis

QPS的3要素



- 线程
- 响应时间
- 瓶颈资源

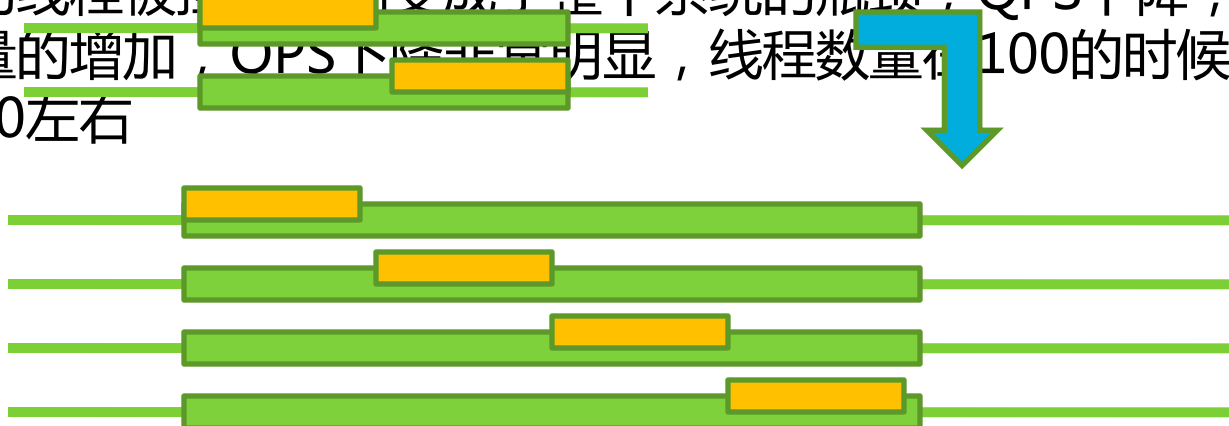
- 设置多少线程合适？

- 设置过少
- 设置过多

- 线程过多导致QPS下降

1、对象生命周期
2、内存占用总量

有个系统：线程数量在12~20之间的時候QPS几乎稳定在120左右，但是一旦线程数量超过30时候，FGC开始频繁，由于FGC导致的线程被挂起时间变成了整个系统的瓶颈，QPS下降，随着线程数量的增加，QPS下降非常明显，线程数量到100的时候，QPS只有60左右



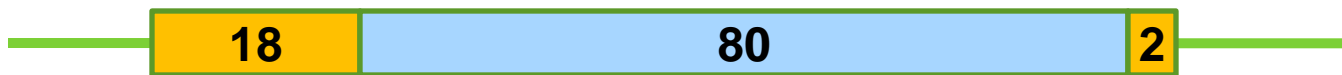
设置多少线程合适？

- CPU+1
- CPU-1

有这样一个模块

- cpu计算时间18ms (running)
- 查询数据库，网络io时间80ms (waiting)
- 解析结果2ms

如果服务器2CPU，大家看看这里多少线程合适？



充分利用CPU资源：

$$\text{线程数量} = 100/20 * 2 = 10$$

- 所以从CPU角度而言

线程数量 = ((CPU时间 + CPU等待时间) / CPU时间) * CPU数量

- 线程数量的设置就是由CPU决定的？

有这样一个模块：

- 线程同步锁(数据库事务锁) 50ms
- cpu时间 18ms
- 查询数据库，网络io时间 80ms
- 解析结果 2ms

如果服务器有2个CPU，这个模块线程多少合适？





- 以CPU计算为瓶颈，计算线程数量
 - 线程数 = $(18 + 2 + 50 + 80) / 20 * 2 = 15$
- 以线程同步锁为瓶颈，计算线程数
 - 线程数 = $(50 + 18 + 2 + 80) / 50 * 1/1 = 3$

公式1：

线程数量 = (线程总时间 / 瓶颈资源时间) * 瓶颈资源的线程并行数

准确的讲

瓶颈资源的线程并行数 = 瓶颈资源的总份数 / 单次请求占用瓶颈资源的份数

约束：

在计算的时候，对同一类资源的消耗时间进行合并

QPS的3要素2



- 响应时间

- $QPS = 1000 / \text{响应时间}$
- $QPS = 1000 / \text{响应时间} * \text{线程数量}$
- 响应时间决定QPS ?

→ 分析数据(10) → 搜索商城 (50) → 搜索产品(25) → 搜索商品(100) → 处理结果(15) →

cpu

waiting

waiting

waiting

cpu

线程数量 = 线程总时间 / 瓶颈资源时间 * 瓶颈资源并行数

线程数量 = $(10 + 50 + 25 + 100 + 15) / (10 + 15) * 4 = 32$

$QPS = \text{线程数量} * 1000 / \text{响应总时间}$

$QPS = 32 * 1000 / (10 + 50 + 25 + 100 + 15)$
 $= 160$

改进

→分析数据(10) →搜索商城 (50) →搜索产品(25) →搜索商品(100) →处理结果(15)→

cpu

waiting

waiting

waiting

cpu



→分析数据(10) →搜索商城 (50)
→搜索产品(25)
→搜索商品(100)

总体响应时间变化
200ms—125ms

→处理结果(15)→

线程数量=线程总时间/瓶颈资源时间 瓶颈资源并行数

线程数量=(10 + 100 + 15)/ (10 + 15) * 4= 20

QPS = 20 * 1000 / (10 + 100 + 15)
= 160

线程数量 = 线程总时间 / 瓶颈资源时间 * 瓶颈资源并行数
QPS = 线程数量 * 1000 / 线程总时间



公式2：

QPS = 1000 / 瓶颈资源时间 * 瓶颈资源并行数

- **公式1：**

线程数量 = 线程**必须**总时间/瓶颈资源时间 * 瓶颈资源并行数

- **约束：**

在计算的时候，对同一类资源的消耗时间进行合并

- **公式2：**

$$QPS = 1000 / \text{瓶颈资源时间} * \text{瓶颈资源并行数}$$

QPS的3要素3

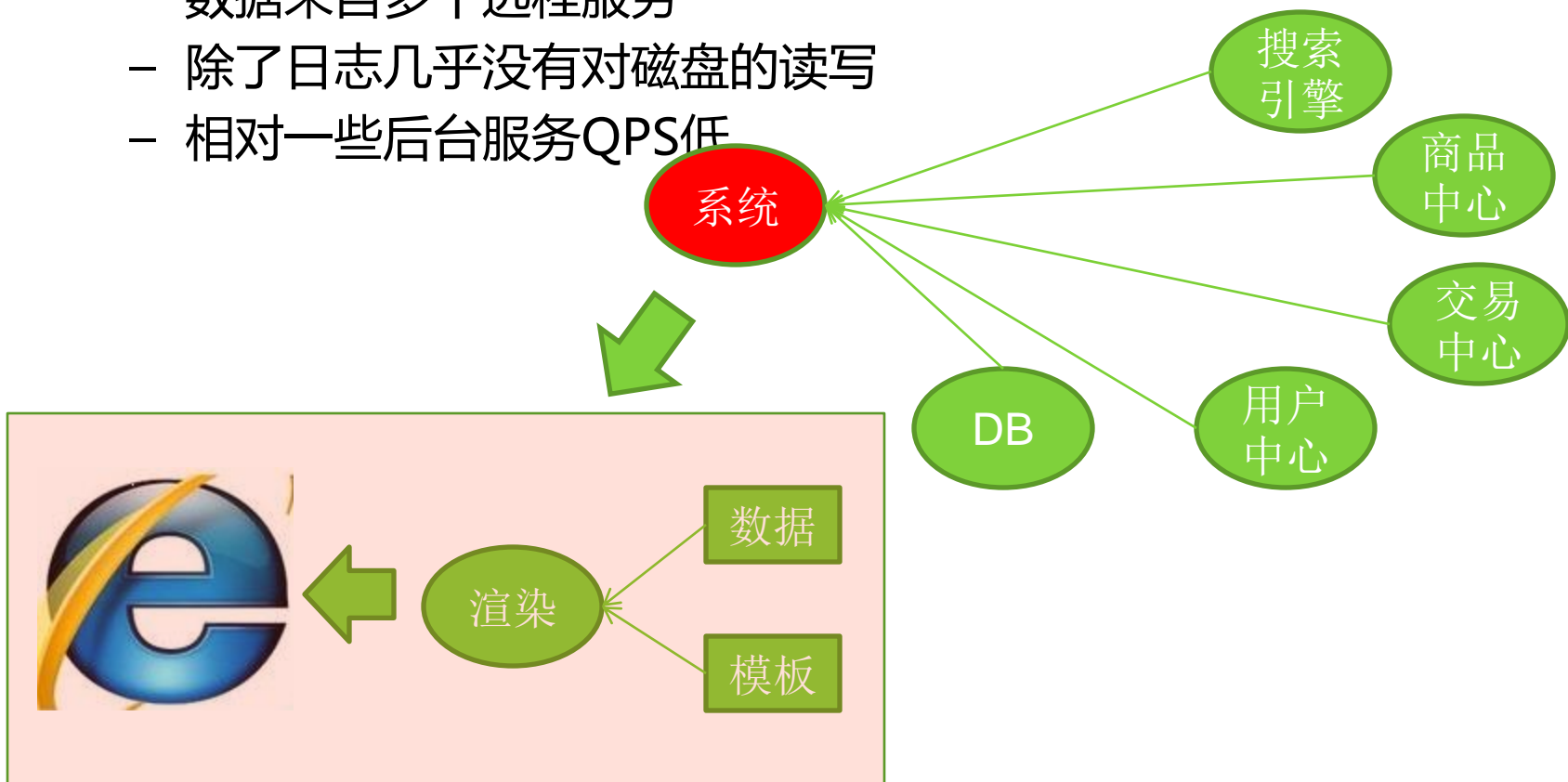
- 瓶颈资源
 - 淘宝的前台系统的瓶颈资源是什么？

CPU



• 淘宝前台系统特点

- 动态页面渲染输出
- 页面非常大
- 数据来自多个远程服务
- 除了日志几乎没有对磁盘的读写
- 相对一些后台服务QPS低



影响系统QPS的瓶颈

Net IO

Disk IO

CPU

内存GC hold

Threads limit

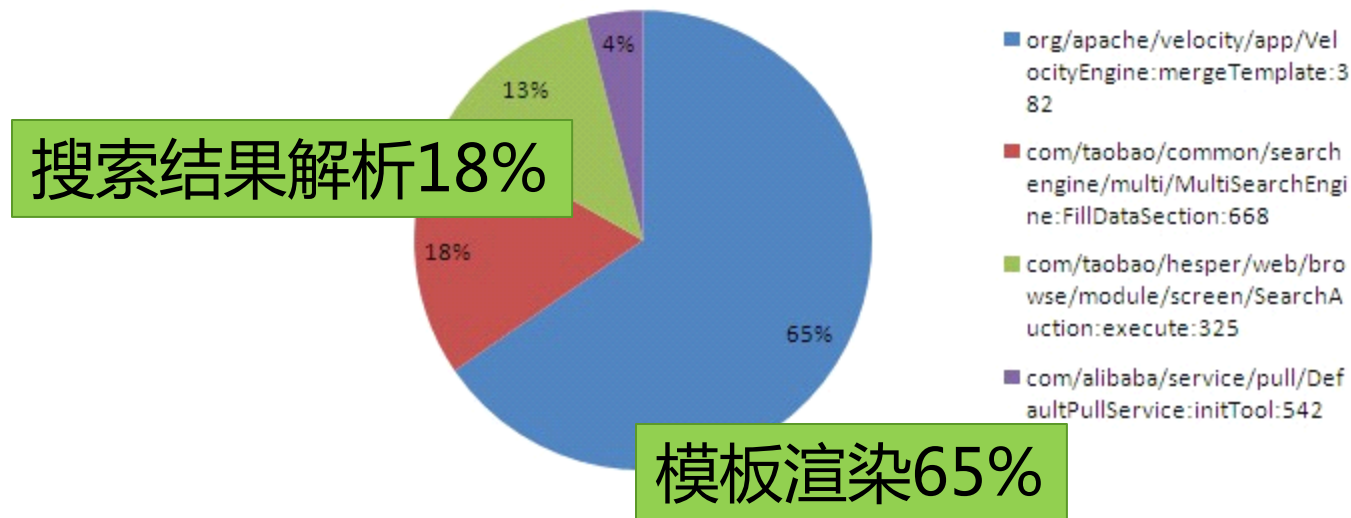
Thread Syn lock

Remoting Sys QPS limit



- AB 压测系统，系统的CPU基本上都跑到了85%以上

总耗时 (ms)



优化模板2式



- 第1式：char to byte

测试例子1：

```
private static String content = "...94k..." ;  
protected doGet(...){  
    response.getWriter().print(content);  
}
```


测试例子2：

```
private static String content = "...94k..." ;  
Private static byte[] bytes = content.getBytes();  
protected doGet(...){  
    response.getOutputStream().write(bytes);  
}
```

- 压测结果：



系统	页面大小 (K)	最高QPS
测试例子1 Servlet print	94	1800
测试例子2 Servlet byte	94	3500



```
String csn = (charsetName == null) ? "ISO-8859-1" : charsetName;
if ((se == null) || !(csn.equals(se.requestedCharsetName())
    || csn.equals(se.charsetName())) {
    se = null;
    try {
Charset cs = lookupCharset(csn);
if (cs != null)
    se = new StringEncoder(cs, csn);
    } catch (IllegalCharsetNameException x) {}
    if (se == null)
        throw new UnsupportedEncodingException (csn);
    set(encoder, se);
}
return se.encode(ca, off, len);
}
```

-----StringEncoder.class

```
byte[] encode(char[] ca, int off, int len) {
    int en = scale(len, ce.maxBytesPerChar());
    byte[] ba = new byte[en];
    if (len == 0)
return ba;
```

```
    ce.reset();
    ByteBuffer bb = ByteBuffer.wrap(ba);
    CharBuffer cb = CharBuffer.wrap(ca, off, len);
    try {
        CoderResult cr = ce.encode(cb, bb, true);
        if (!cr.isUnderflow())
            cr.throwException();
        cr = ce.flush(bb);
        if (!cr.isUnderflow())
            cr.throwException();
    } catch (CharacterCodingException x) {
        // Substitution is always enabled,
        // so this shouldn't happen
        throw new Error(x);
    }
    return safeTrim(ba, bb.position(), cs);
}
```

ISO_8859_1.java

```
private CoderResult encodeArrayLoop(CharBuffer src,
    ByteBuffer dst)
{
    char[] sa = src.array();
    int sp = src.arrayOffset() + src.position();
    int sl = src.arrayOffset() + src.limit();
    assert (sp <= sl);
    sp = (sp <= sl ? sp : sl);
    byte[] da = dst.array();
    int dp = dst.arrayOffset() + dst.position();
    int dl = dst.arrayOffset() + dst.limit();
    assert (dp <= dl);
```

byte

110 9441

淘宝对Velocity进行了重构

- 利用 char to byte （ 100% ）
- 解析执行改成了编译后执行 （ 10% ）

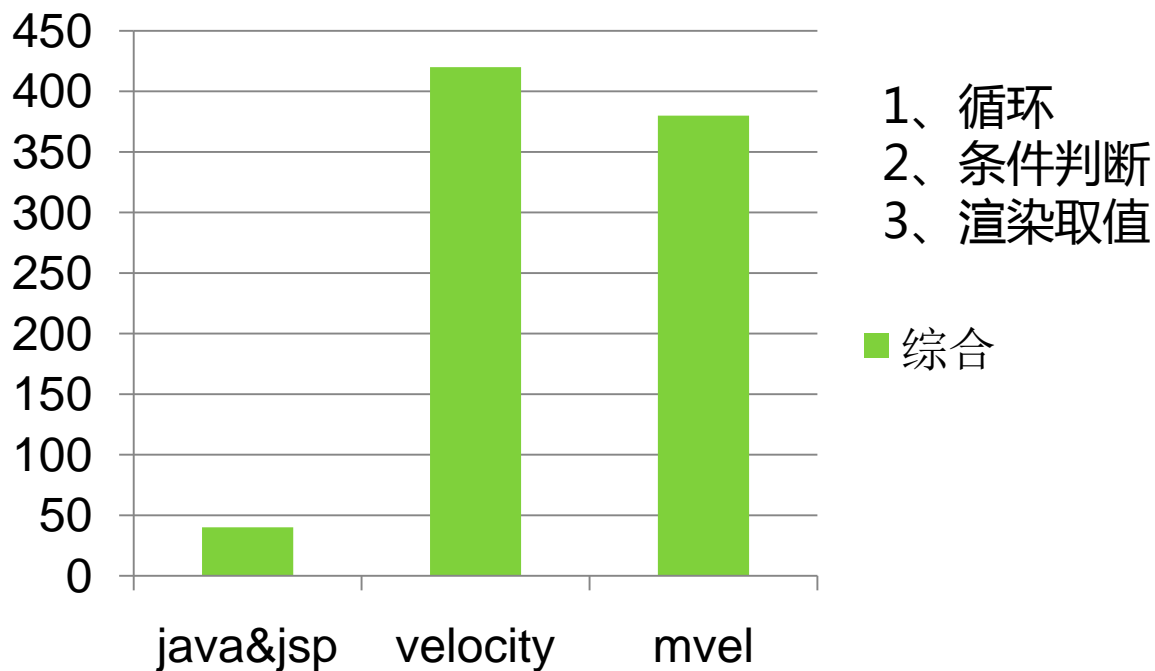
页面大小	优化前 QPS	优化前 RT(ms)	优化后的 QPS	优化后 RT(ms)	提升%
47355	319.05	109.7	455.87	76.776	43%
48581	306.85	114.061	445.39	78.582	45%
55735	296.65	117.983	437.46	80.007	47%
63484	193.69	180.608	302.55	115.684	56%
83152	180.88	193.333	236.1	148.305	30%
92890	170.68	205.867	219.27	163.342	26%
99732	103.64	337.707	161.46	216.77	56%
144292	108.76	321.81	148.18	236.199	36%
67144	148.49	235.714	268.07	130.565	81%
79703	124.51	281.1	243.64	143.657	96%
92537	123.85	282.595	190.8	183.44	54%
127047	117.52	297.829	164.1	213.284	40%
129479	105.36	332.197	155	225.8	47%

50%



- 解析执行转编译后执行的效果

综合

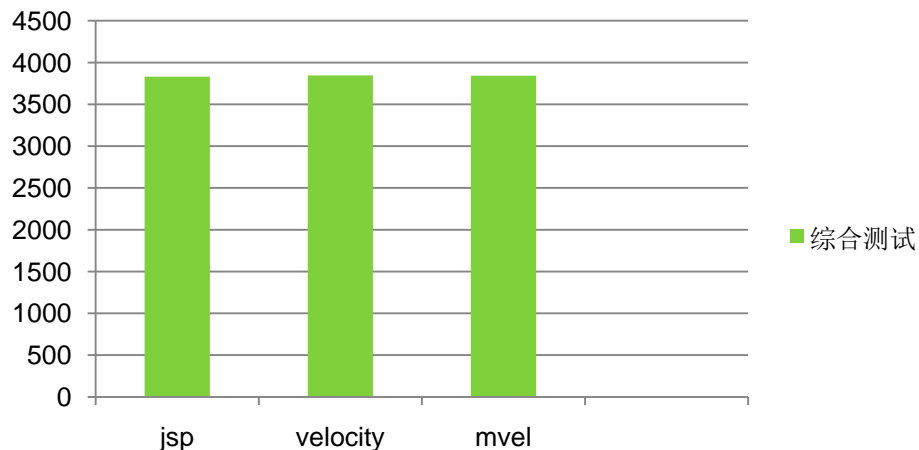



```
"Hello, my name is ${name.toUpperCase()}, "  
"#foreach($user in $group.users) - ${user.id} - ${user.name} #end "
```



```
"Hello, my name is ${name.toUpperCase()}, "  
" ...5k..."  
"#foreach($user in $group.users) - ${user.id} - ${user.name} #end "
```

综合测试



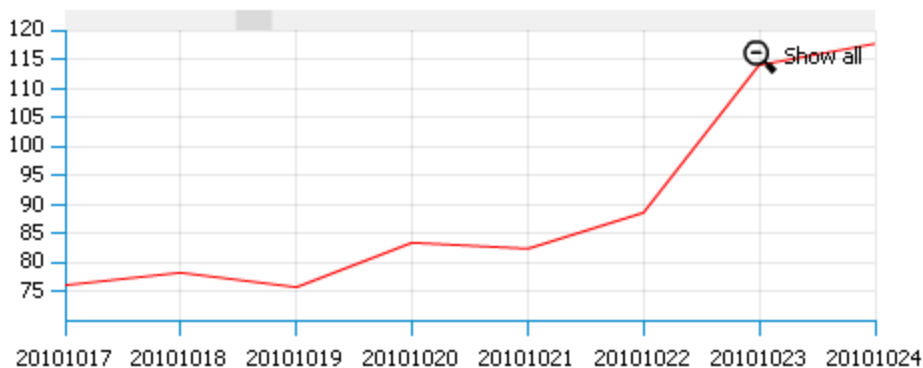
总结



- Char 2 byte
- 规模效益
 - 取决于脚本（判断，循环，比较，赋值）占整个模板的比例

优化模板2式

- 第2式：减少模板大小



35%

页面从170K下降到110K

- 减少模板大小的方法
 - 压缩模板的空白字符
 - 重复数据合并
 - 异步渲染

- 压缩空白字符
 - 压缩哪些
 - 何时压缩
 - 工具

- 重复数据合并

- 对一些系统非常有效，凡是代码里涉及到了循环，并且里面有静态内容输出均可以采用此方法

<a

href="http://www.taobao.com/go/act/sale/kade_pay.php?ad_id=&am_id=&cm_id=14001249574b437a411d&pm_id=" title="信用卡"

class="creditcard" target="_blank">信用卡

....



工具的支持？

信用卡

....

+ javascript

¥ 639.00 运费：10.00 信用卡
¥ 799.00 运费：10.00 信用卡
¥ 27.00 运费：5.00 信用卡

异步渲染

- 将页面中静态并且相对不重要的内容抽取出来
- 利用专用服务器的优势

异步加载结合专用Server

价格: 到

店内搜索

宝贝详情

评价详情(138) 成交记录(303件) 服务质量 掌柜推荐 售后服务 留言簿

淘宝商城承诺: 您付款之后, 如果卖家缺货, 可以获得商品价格5% (不大于30元) 的赔付金, 详见。

裤长: 长裤 (穿起裤长至脚踝下)	货号: 08501-0297	裤型: 直筒裤 (膝盖围=裤口围)
腰型: 低腰	风格: 时尚休闲	风格细分: 日常休闲
颜色: 蓝色	尺码: 29 32 28 32 31 32 30 32 33	牛仔面料: 全棉牛仔布
裤门襟: 拉链开襟	工艺处理: 褶皱	适合群体: 青年
价格区间: 500元以上	品牌: Levi's/李维斯	

发货物流: 本店使用快递为国内物流服务最优, 品质最好, 速度最快的顺风快递和网点最广的EMS (拍下商品的时候选择快递为顺风快递, 选择EMS则为EMS, 注: 包邮活动默认快递EMS)

实物拍摄展示 Taken in display



客服中心

客服旺旺 (一号)

客服旺旺 (二号)

客服旺旺 (三号)

客服旺旺 (售后1)

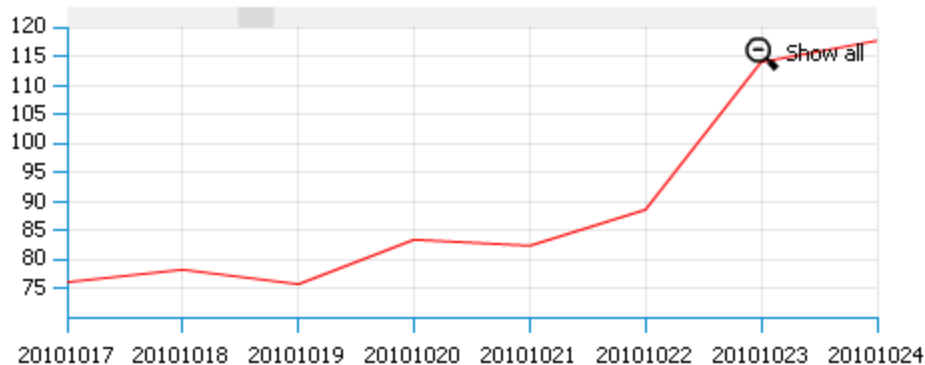
客服旺旺 (售后2)

服务时间

周一至周日 9:00~23:00

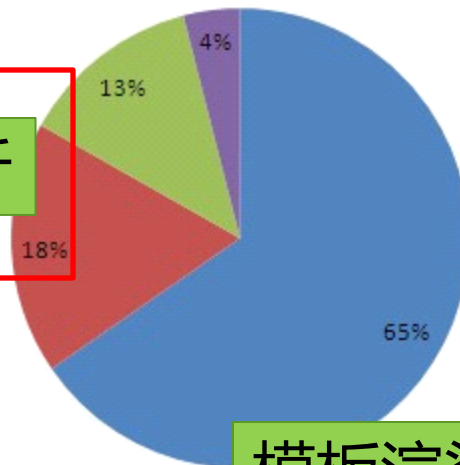
CDN化, Cache化

- 减少模板大小
 - 压缩空白字符
 - 合并相同数据
 - 异步渲染，利用专用服务器的优势
- 10%~100%以上内容的节省
- QPS的提升10%~80%



总耗时 (ms)

搜索结果解析



模板渲染

- org/apache/velocity/app/VelocityEngine:mergeTemplate:382
- com/taobao/common/searchengine/multi/MultiSearchEngine:FillDataSection:668
- com/taobao/hesper/web/browse/module/screen/SearchAuction:execute:325
- com/alibaba/service/pull/DefaultPullService:initTool:542

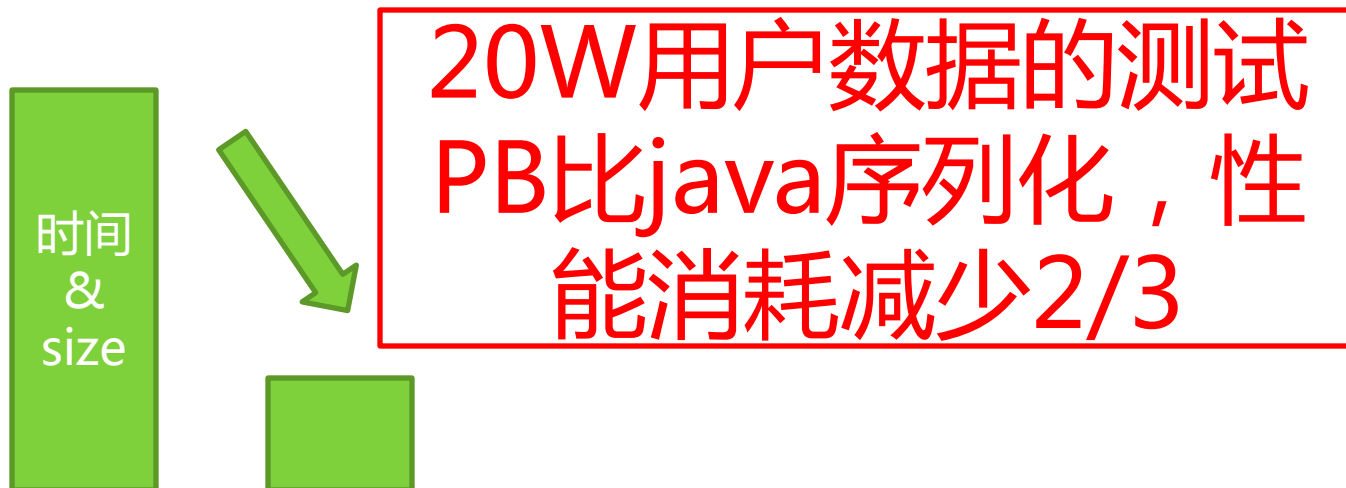
- 优化大数据的处理

- 搜索结果解析案例

- Byte 2 char

- 序列化方式的选择

- thrift , protobuf vs java序列化

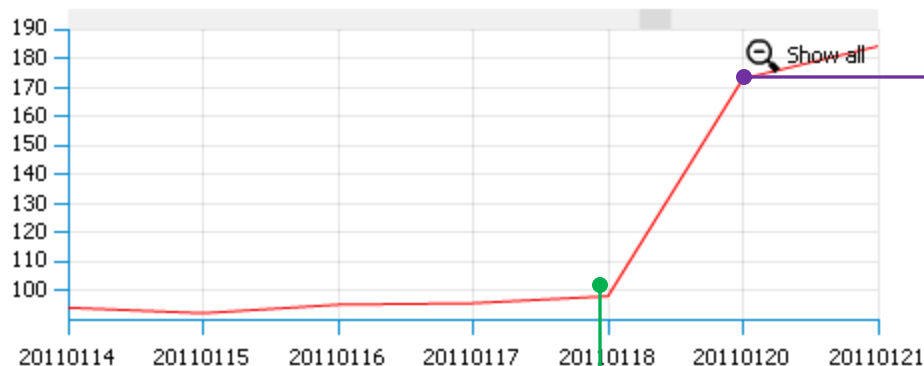


优化jvm的参数

- Yong区的比例调整
- 减少GC的总时间



- 某系统操作系统32位升级到64位
- Yong区从500M增加到-Xmn2560m



1.-Xmn2560m
2.QPS=170

1.-Xmn500m
2.QPS=100

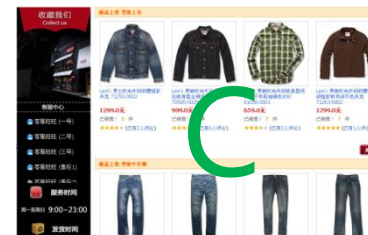
70%



- 对其他系统进行yong区调整，QPS并没有有效的提升

根据每日的压测报表信息，统计得到一些规律：

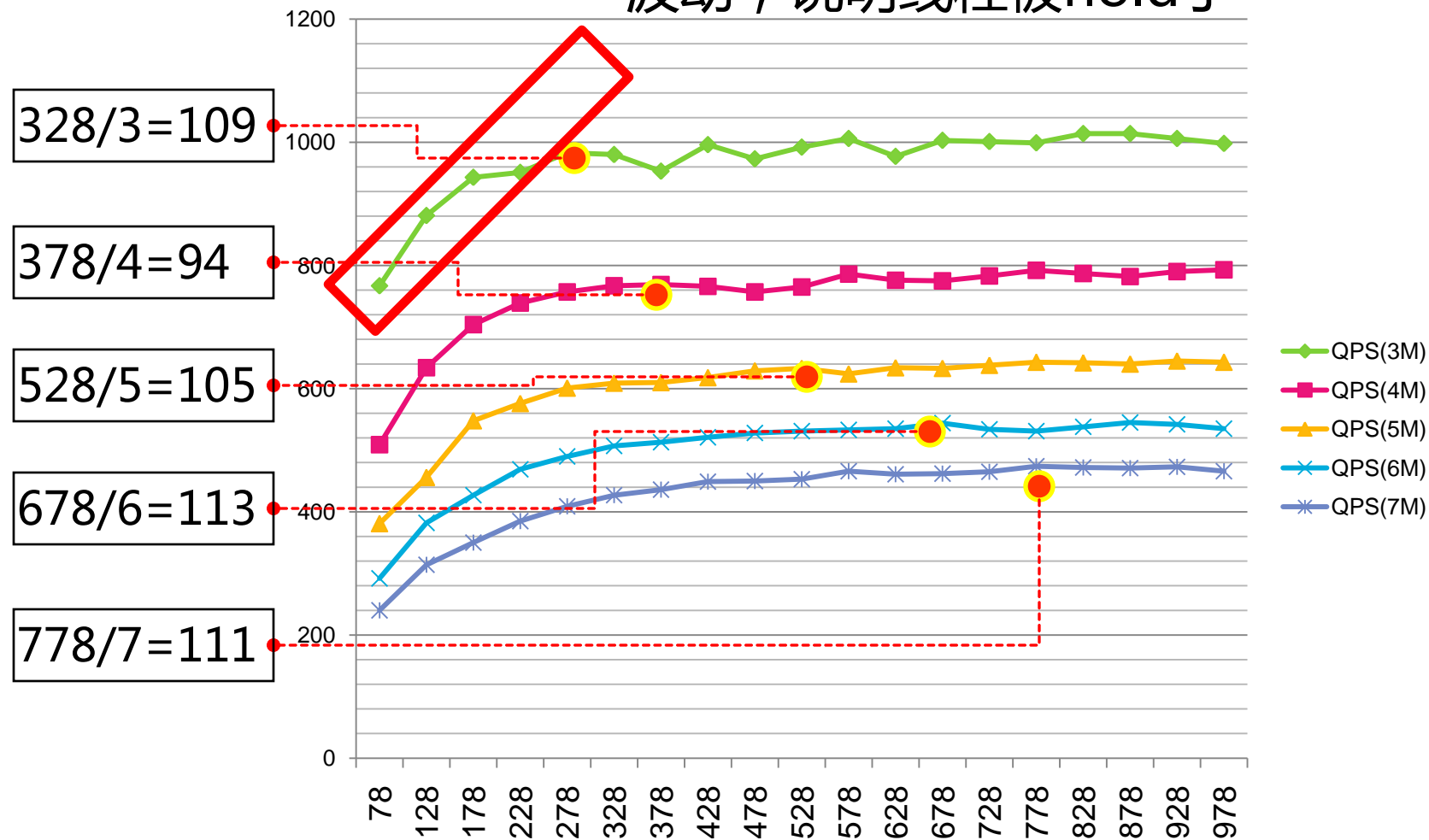
系统	平均 页面大小	平均每次 请求消耗内存	Young 区大小
A（效果明显）	110K	12M	500M
B（不明显）	66K	5.2 M	500M
C（不明显）	65K	5.4 M	500M



- 申请3Mbyte内存，-XX:NewSize=78M~978M，统计QPS
 - NewSize 越大QPS越高
 - NewSize 越大GC时间越短

NewSize(M)	QPS(3M)	QPS(4M)	QPS(5M)	QPS(6M)	QPS(7M)	GC(count)	GC(real)
78	767	509	381	292	240	13590	137.19
128	881	634	456	382	314	8098	94.2
178	943	704	548	427	350	5533	60.77
228	951	739	576	469	385	4219	37.72
278	983	757	601	490	409	3316	28.62
328	980	767	609	507	427	2734	23.24
378	953	769	610	513	436	2306	19.28
428	996	766	618	521	449	2002	16.8
478	973	757	629	528	450	1756	15.78
528	992	765	633	531	453	1576	13.23
578	1006	786	624	533	466	1431	11.66
628	977	776	634	535	461	1304	11.03
678	1003	775	633	544	462	1202	10.2
728	1001	783	638	534	465	1115	9.19
778	999	792	643	531	474	1037	8.68
828	1014	787	642	538	472	971	8.18
878	1014	782	640	545	471	913	7.59
928	1006	790	645	542	473	859	7.12
978	998	793	643	535	466	812	6.76

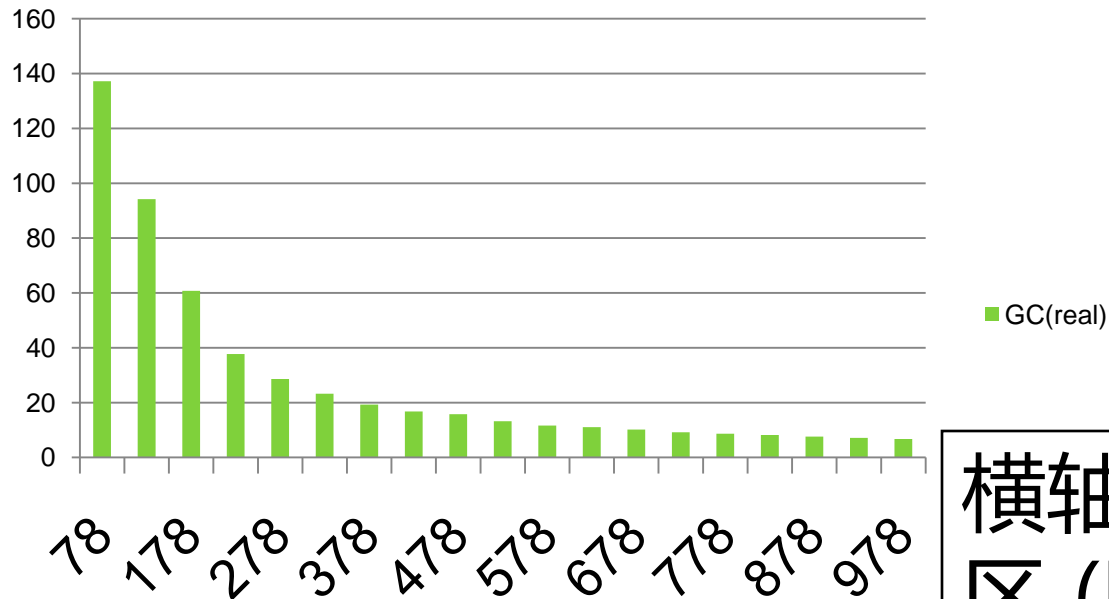
- 每个点，理论上响应时间应该是一样，波动，说明线程被hold了



GC时间包括Full GC
时间和minorGC时间

GC总时间

GC总时间(毫秒)



横轴：young
区 (M)

- Yong区大小至少要大于每次请求内存消耗的100倍
 - Old区被挤占的问题
 - 单次minorgc是否会变长的问题

Jvm参数优化-减少GC总时间

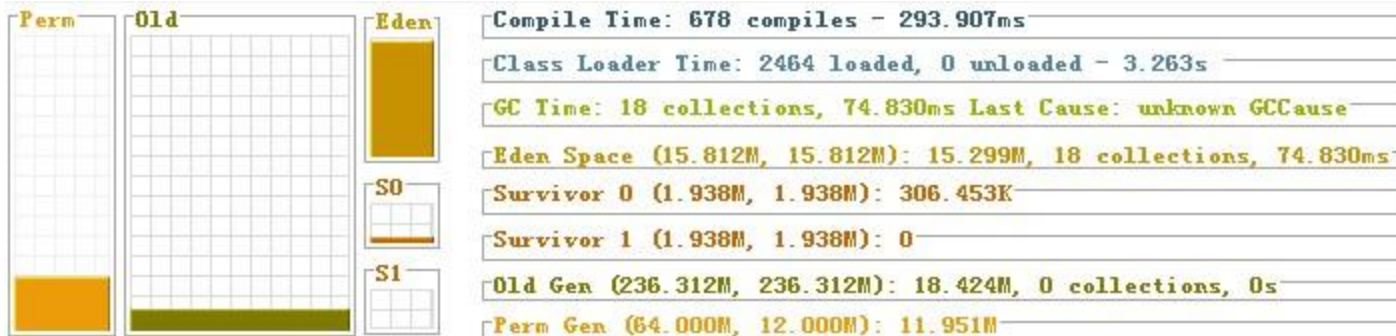
- 减少GC总时间



对于淘宝前台系统而言，年老代区一般存放的内容是什么？

垃圾回收step

- 1、对象在Eden区完成内存分配
- 2、当Eden区满了，再创建对象，会因为申请不到空间，触发minorGC，进行young(eden+1survivor)区的垃圾回收
- 3、minorGC时，Eden不能被回收的对象被放入到空的survivor（Eden肯定会被清空），另一个survivor里不能被GC回收的对象也会被放入这个survivor，始终保证一个survivor是空的
- 4、当做第3步的时候，如果发现survivor满了，则这些对象被copy到old区，或者survivor并没有满，但是有些对象已经足够Old，也被放入Old区 XX:MaxTenuringThreshold
- 5、当Old区被放满的之后，进行完整的垃圾回收



- 数据进入年老代的3个途径
 - 直接进入Old区
 - 超过指定size的数据
 - 比较少见，一般一下子申请大片缓冲区
 - **minorGC触发时**，交换分区S0或者S1放不下
 - 缓存数据
 - 因为线程执行周期缓慢导致未释放的对象的量太多了
 - **足够老的数据**，在交换区拷贝次数超过了上限（
XX:MaxTenuringThreshold=15）
 - 缓存数据
 - 因为线程执行周期缓慢导致未释放的对象

example1



- -Xmx256m -Xmn15m -XX:SurvivorRatio=6

- Eden 11.2M

- S0 1.87M

JSP Old 241M

<%

```
int size = (int)(1024 * 1024 * m);  
byte[] buffer = new byte[size];  
Thread.sleep(s);
```

1、申请1M内存
2、等待10秒钟

%>

1个线程进行压测：

ab -c1 -n1000 "http://localhost/perf.jsp?m=1&s=10"

2个线程进行压测：

ab -c2 -n1000 "http://localhost/perf.jsp?m=1&s=10"

3个线程进行压测：

ab -c3 -n1000 "http://localhost/perf.jsp?m=1&s=10"

example2



JSP

<%

```
int size = (int)(1024 * 1024 * 1.2);
```

```
byte[] buffer = new byte[size];
```

```
buffer = null;
```

```
Thread.sleep(10);
```

%>

2个线程进行压测：

```
ab -c1 -n10000 "http://localhost/perf.jsp"
```

3个线程进行压测：

```
ab -c3 -n10000 "http://localhost/perf.jsp"
```

20个线程进行压测：

```
ab -c20 -n10000 "http://localhost/perf.jsp"
```

编写GC有好的代码

```
{  
StringBuffer a = new StringBuffer( "....." );  
a.append( "....." );  
...  
}
```

```
Object c = SearchManager.search(b); // waiting 100ms
```

这里触发gc的概率99%以上

```
.....  
}
```

- 1、对象a的生命周期=方法的生命周期
- 2、被gc的时间至少>100ms

改进之后

```
{  
StringBuffer a = new StringBuffer( "....." );  
a.append( "....." );  
.....  
}
```

```
a = null;
```

```
Object c = SearchManager.search(b); // waiting 100ms
```

```
.....  
}
```

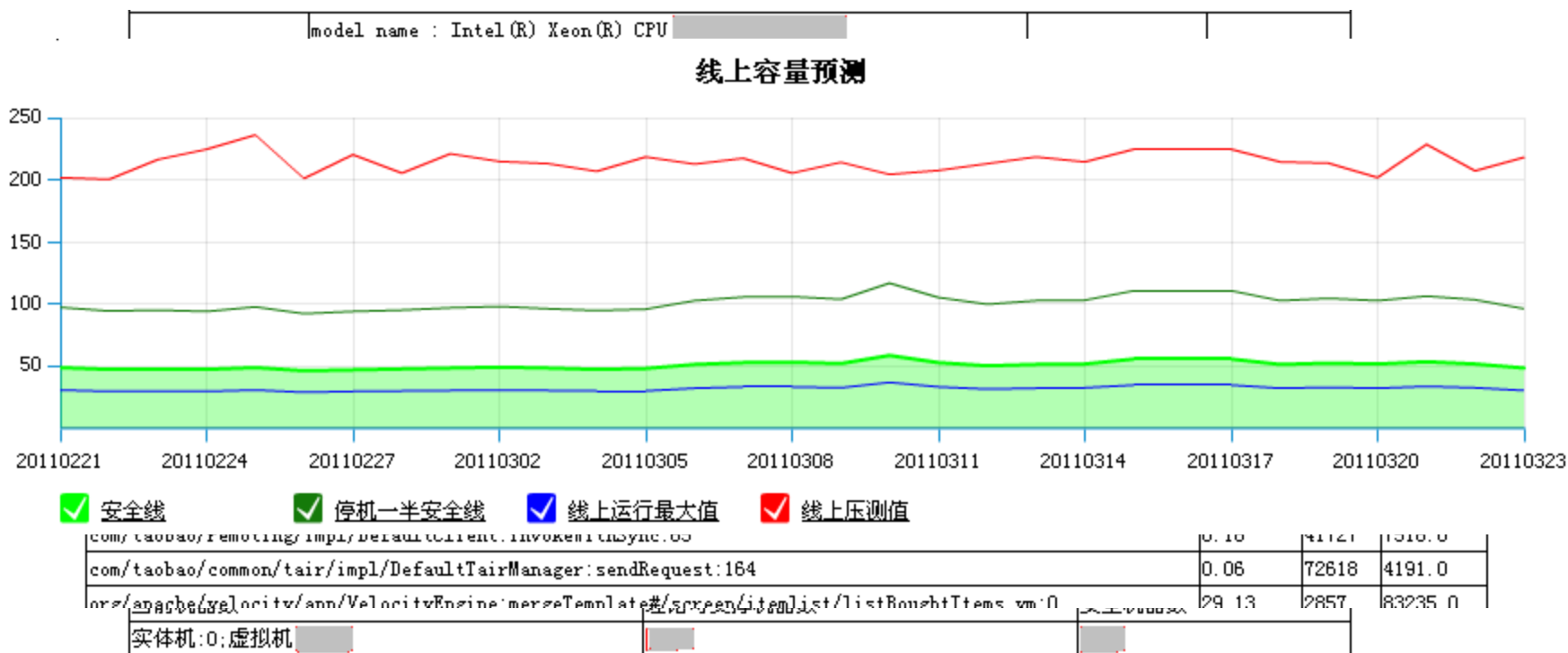
- 1、对象a的生命周期在a=null之后结束
- 2、可以被随时回收
- 3、一般认为一个耗时的方法之前的对象尽可能对GC优化

- 调整yong区的大于
 - 大于每次请求的消耗内存的100倍
- 减少GC的总时间
 - 最佳实践，在一些远程调用方法之前，尽量释放掉对象的引用

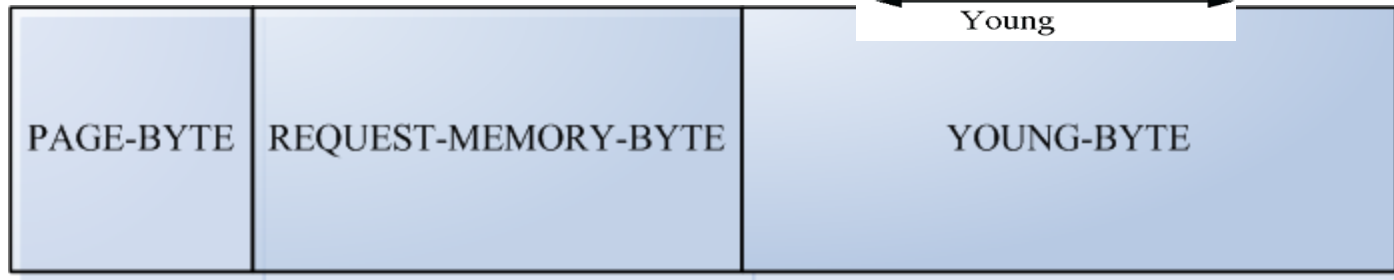
保持优化成果



- Daily load running
- Daily hotspot code analysis



淘宝前台系统的一些规律



<100倍

>100倍

xiaoxie@taobao.com

THANK YOU



附录：平均单次请求内存消耗计算方法

- 每个请求占用的内存 = $\text{Eden} / (\text{QPS} * \text{minorGC的平均间隔时间(秒)})$