Jvm 优化实践&数据库连接池&多级缓存实战

今日课程主要内容：

1、JVM 调优实践（fullgc,yong,old,s0s1）
2、吞吐量优先组合
3、响应时间优先垃圾回收器组合
4、G1
5、数据库调优（连接池参数）
6、服务分布式部署对性能影响
7、多级缓存（缓存架构，堆内存缓存，分布式缓存，内存字典，lua+redis 缓存）

1 JVM 调优实践

明确：

1、JVM 调优本质是 GC 垃圾回收；每次 GC 的时候，都会导致业务线程 STW, 因此频繁的 gc 会导致性能严重下降；

2、JVM 调优就是垃圾回收器参数进行一个设置，经过 JVM 内存模型，会进行内存参数设置，分配合理的内存；如果 JVM 堆内存分配合理，降低 gc 时间，gc 次数；

1.1 典型参数设置

典型的参数设置：4cpu,8GB

1、-Xmx4000M 设置 JVM 堆内存最大值（经验值的设置 3500MB-4000MB, 内存设置的大小，没有固定的设置，根据业务实际情况进行设置，根据压力测试情况，在线上运行一段时间（7 天，30 天），JVM 内存进行不断的调试）

2、-Xms4000M 设置 JVM 堆内存的初始化内存（一般情况下，都必须和最大内存设置为一致，防止内存抖动）

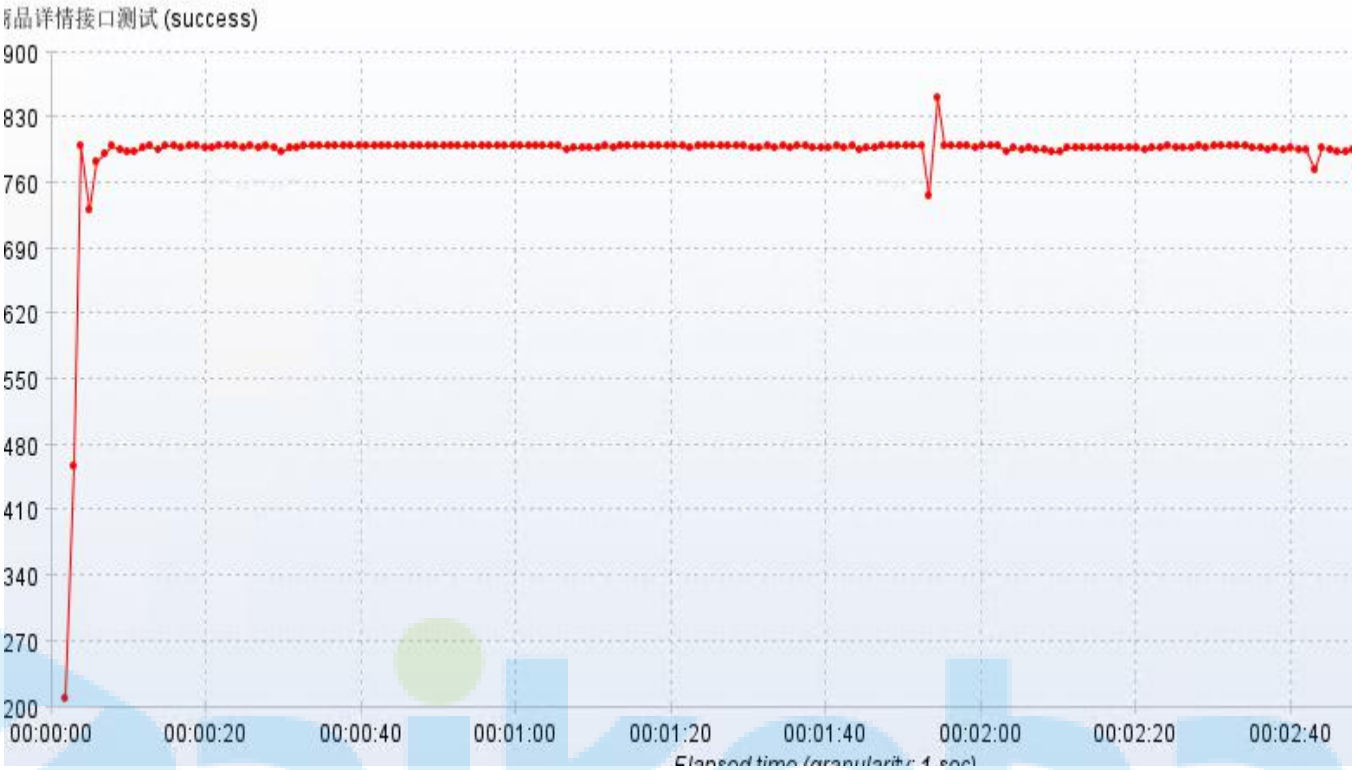
3、-Xmn2g 设置年轻代大小（eden,s0,s1）

4、-Xss256k 设置线程栈的大小，JDK1.5+版本线程栈默认是 1MB，相同的内存情况下，线程对象越小，操作系统会创建更多性能，系统性能会更好；

初始化优化参数设置：

```
nohup java -Xmx4000m -Xms4000m -Xmn2g -Xss256k -jar jshop-web-1.0-SNAPSHOT.jar --spring.config.addition-location=application.yaml > jshop.log 2>&1 &
```

压力测试情况：



根据压力测试结果，发现 JVM 参数的设置，和没有设置之前没有变化，因为没有设置默认分片 1G+ Yong 内存大小，而对于业务样本来说，产生的对象基本上不足以造成频繁的 gc, fullgc;

问题： 根据什么指标去调优？？

1、发生了几次 gc ,是否频繁的发生 GC

2、是否发生了 full gc (full gc 将会对整个堆内存进行垃圾回收 ，因此发生 full gc 比较耗时)

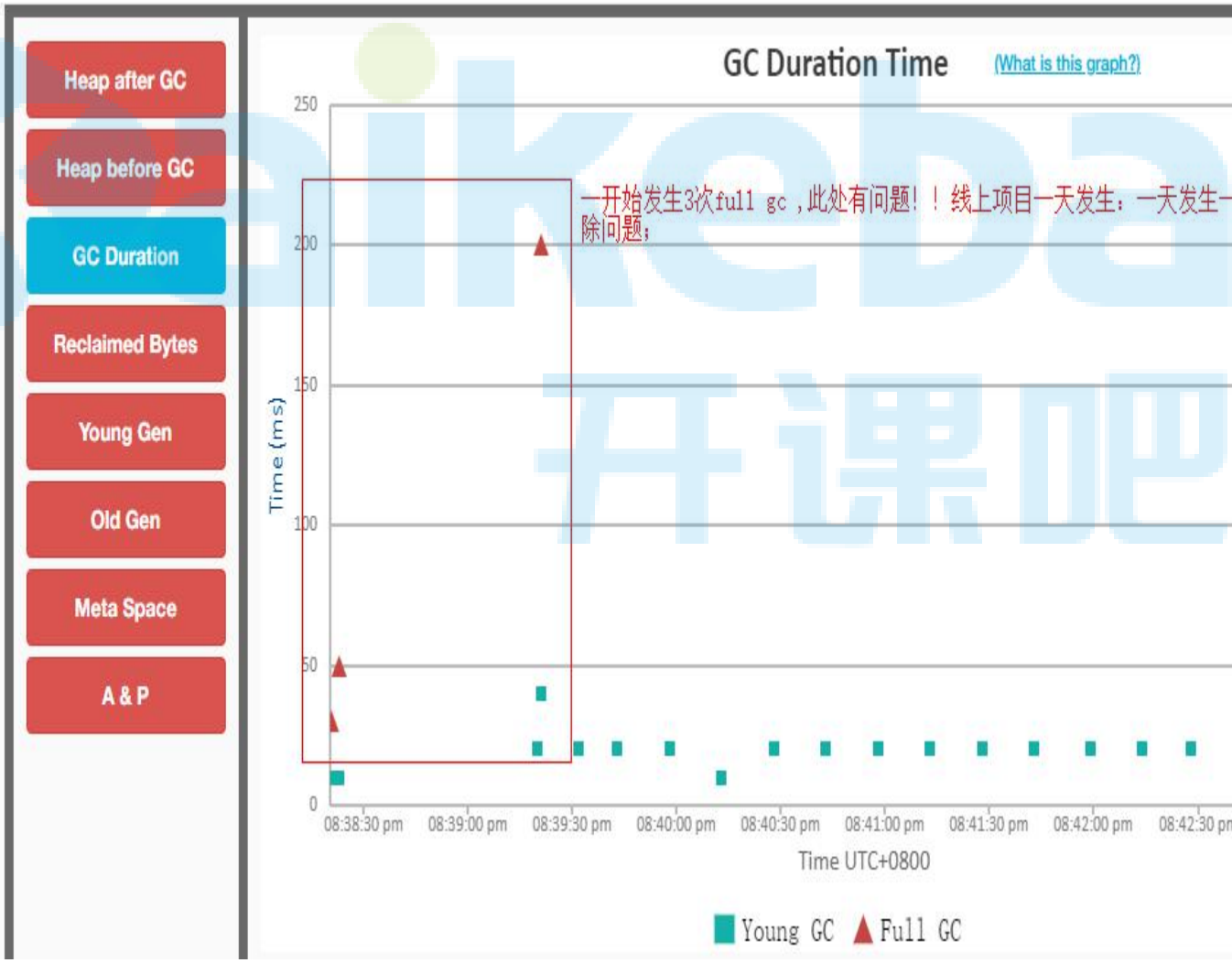
1.2 GC 日志输出

JVM 在 gc 的时候输出日志，把日志打印到一个文件中，然后使用相应的工具(gceasy.io)对日志进行在线分析；分析 jvm 可调优的空间；

```
nohup java -Xmx4000m -Xms4000m -Xmn2g -Xss256k -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -Xloggc:gc.log -jar jshop-web-1.0-SNAPSHOT.jar --spring.config.addition-location=application.yaml > jshop.log 2>&1 &
```

GC 日志可视化分析：

GC 情况： 一开始发生 3 次 full gc ,必须进行调优



Gc 可视化数据分析：

Total GC stats

Total GC count	24
Total reclaimed bytes	32.56 gb
Total GC time	690 ms
Avg GC time	28.7 ms
GC avg time std dev	36.7 ms
GC min/max time	10.0 ms / 200 ms
GC Interval avg time	13 sec 314 ms

Minor GC stats

Minor GC count	21
Minor GC reclaimed	32.55 gb
Minor GC total time	410 ms
Minor GC avg time	19.5 ms
Minor GC avg time std dev	5.75 ms
Minor GC min/max time	10.0 ms / 40.0 ms
Minor GC Interval avg	15 sec 311 ms

Full GC stats

Full GC Count	
Full GC reclaimed	
Full GC total time	
Full GC avg time	
Full GC avg time std dev	
Full GC min/max time	
Full GC Interval avg	

1.3 full gc 调优

排查一下原因： 为什么会一开始就发生 3 次 full gc ??
查看一下内存模型情况：可以看一下那些内存区域被占满了!! 占用比较多
排查命令： jstat -gcutil PID

```
root@qps004 java18]# jstat -gcutil 30579
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT
  0.00  99.71  34.14   7.17  95.04  91.81     21    0.449     3    0.2
root@qps004 java18]#
```

MetaSpace 元数据空间：初始化大小 20MB

Meta Space	1.05 gb

，
说明 MetaSpace 发生了扩容现象；MetaSpace 每扩容一次，就会发生一次 fullgc; 因此要避免由于 metaspace 引起的 fullgc,必须对 metaSpace 元数据空间大小进行设置一个合理的值；

MetaSpaceSize 设置： 防止因为 MetaSpaceSize 扩容而引起 full gc 现象；

```
nohup java -Xmx4000m -Xms4000m -Xmn2g -Xss256k -XX:MetaspaceSize=256m
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC
-Xloggc:gc.log -jar jshop-web-1.0-SNAPSHOT.jar
```

```
--spring.config.addition-location=application.yaml > jshop.log 2>&1 &
```

经过调优后，full gc 已经消除了： 同时节省了 gc 时间，提升服务性能

Minor GC stats

Minor GC count	23
Minor GC reclaimed	37 gb
Minor GC total time	510 ms
Minor GC avg time	22.2 ms
Minor GC avg time std dev	15.9 ms
Minor GC min/max time	0 / 60.0 ms
Minor GC Interval avg	11 sec 346 ms

Full GC stats

Full GC Count
Full GC reclaimed
Full GC total time
Full GC avg time
Full GC avg time std dev
Full GC min/max time
Full GC Interval avg

1.4 YONG&OLD 比例

问题： 年轻代，老年代 大小比例的设置，到底设置多少才更为合适呢？？？

比例设置参数： -XX: NewRatio = 4 ,

回答： 此比例设置必须根据业务类型进行判断，根据线上服务运行情况进行调优设置；无非是 yong 空间设置大一些，或者是 old 空间设置大一些；

参数：-XX: NewRatio = 4 ，年轻代：老年代 = 1:4 ，年轻代空间进一步缩小，更加频繁的 yong gc !!

分析结论： 年轻代：老年代 = 1:4 （4000MB） 800MB : 3200MB , yong 内存空间变小了，以为将会发生更多此的 yong gc

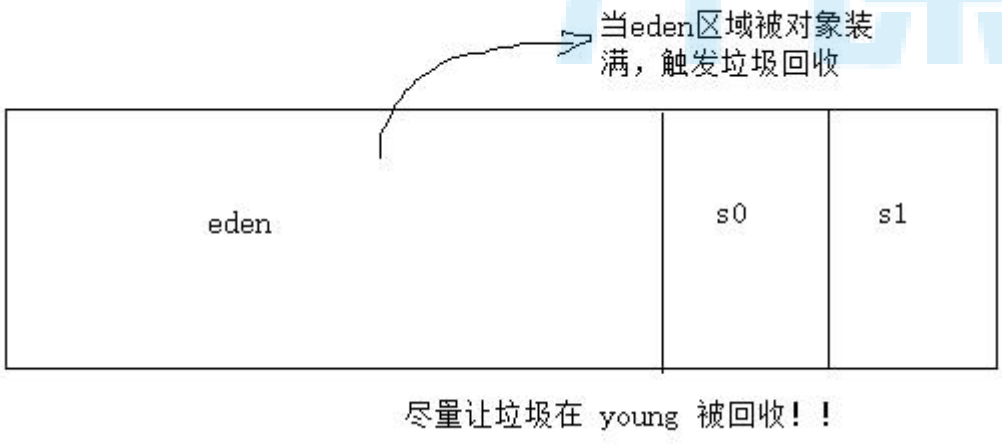
关于 JVM 调优： 年轻代，老年代大小比例设置必须根据线上项目运行情况进行比例的调整，寻找一种 balance;

1、youg gc --- 尽量让垃圾在 young 被回收

2、尽量减少 full gc

1.5 Eden&S0&S1

思考： 您认为 eden ,s0 ,s1 区域内存比例应该如何进行设置？？？

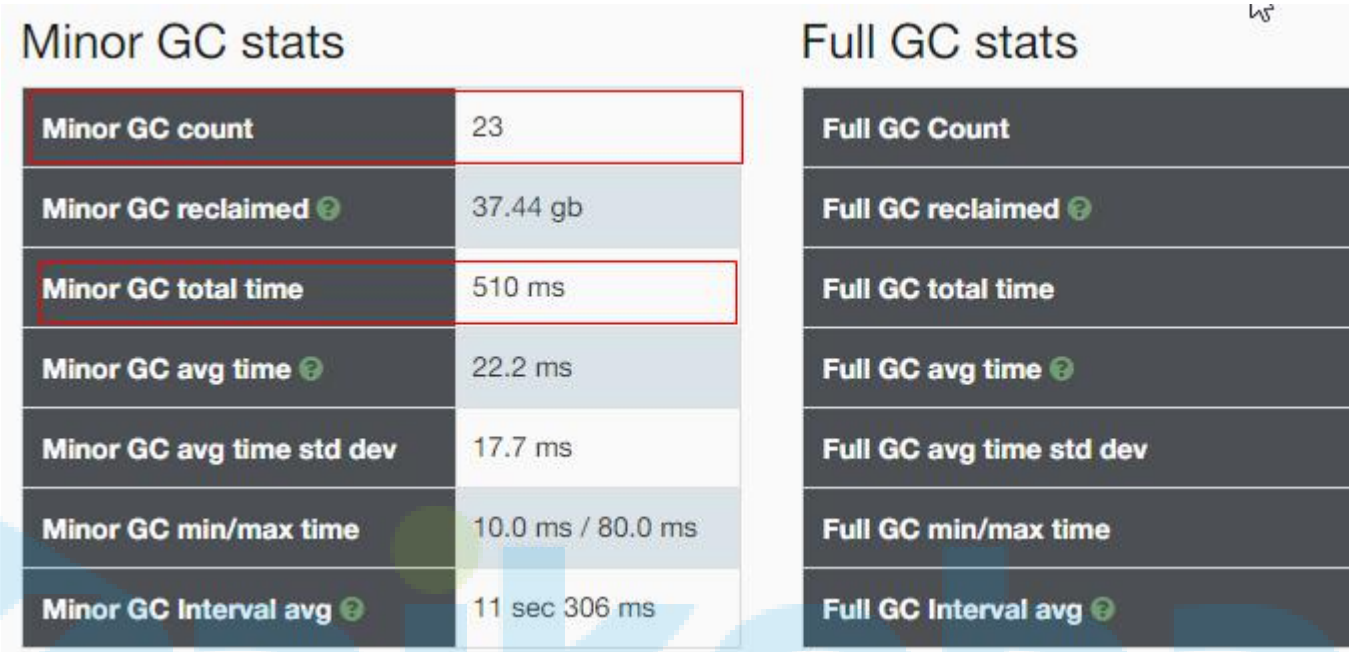


官方： eden:s0:s1 = 8:1:1 设置方式： -XX:SurvivorRatio=8

```
nohup java -Xmx4000m -Xms4000m -Xss256k -Xmn2g -XX:SurvivorRatio=8
-XX:MetaspaceSize=256m -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -Xloggc:gc.log -jar jshop-web-1.0-SNAPSHOT.jar
```

```
--spring.config.addition-location=application.yaml > jshop.log 2>&1 &
```

-XX:SurvivorRatio=8 : 尽可能让对象在年轻代被回收；



2 GC 经典组合

2.1 吞吐量优先

并行垃圾回收器组合，并行垃圾回收器有哪些？？

年轻代： ParNew , Parallel Scavenge

老年代： Parallel Old

对于吞吐量优先的垃圾回收器来说，就是并行垃圾回收器

使用的垃圾回收器组合： ps + po ------ 就是 Jdk8 默认的垃圾回收器；

当然也可以进行显示的配置：

```
nohup java -Xmx4000m -Xms4000m -Xss256k -Xmn2g -XX:SurvivorRatio=8 -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:MetaspaceSize=256m -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -Xloggc:gc.log -jar jshop-web-1.0-SNAPSHOT.jar --spring.config.addition-location=application.yaml > jshop.log 2>&1 &
```

2.2 响应时间优先

响应时间优先垃圾回收器： 并发垃圾回收器（业务线程，gc 线程交叉执行，减少业务线程 stw 时间），垃圾回收器组合：parNew+CMS

```
nohup java -Xmx4000m -Xms4000m -Xss256k -Xmn2g -XX:SurvivorRatio=8 -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:MetaspaceSize=256m -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -Xloggc:gc.log -jar jshop-web-1.0-SNAPSHOT.jar --spring.config.addition-location=application.yaml > jshop.log 2>&1 &
```

XX:+UseParNewGC: 年轻代垃圾回收器，并行垃圾回收器

XX:+UseConcMarkSweepGC： 并发垃圾回收器，老年代的垃圾回收器

2.3 G1 组合

```
nohup java -Xmx4000m -Xms4000m -Xss256k -Xmn2g -XX:SurvivorRatio=8 -XX:+UseG1GC -XX:MetaspaceSize=256m -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -Xloggc:gc.log -jar jshop-web-1.0-SNAPSHOT.jar --spring.config.addition-location=application.yaml > jshop.log 2>&1 &
```

可以发现垃圾回收器： gc 次数减少了，但是 gc 总耗时增加了 70ms; 因此可以发现 ps+po 是比较好的组合；

Other GC stats

Other GC count	15
Other GC reclaimed	28.9 gb
Other GC total time	580 ms
Other GC avg time	38.7 ms
Other GC avg time std dev	19.3 ms
Other GC min/max time	20.0 ms / 90.0 ms
Other GC Interval avg	16 sec 872 ms

Full GC stats

Full GC Count
Full GC reclaimed
Full GC total time
Full GC avg time
Full GC avg time std dev
Full GC min/max time
Full GC Interval avg

3 连接池调优

3.1 为什么进行数据库调优？

1、避免网站网页出现错误

Timeout 5xx 错误

慢查询导致业务无法加载

阻塞操作数据无法提交

2、增加数据库稳定性（很多数据库问题： 低效的查询语句造成）

3、优化用户体系

流畅的业务访问效果

良好网站功能体验

3.2 是什么影响了数据库性能?

问题： 双 11，双 12 网络有很大流量，后端服务器面临很大的压力 ---- 扩容，架构; web 服务器扩容非常简单的，每一个 web 服务器都是无状态，因此扩容非常简单，那么数据库扩容如何发生？？

答案： 数据库扩容非常困难，因为数据库是有状态服务服务，不能随意进行扩容的，影响数据完整性，数据一致性；

解决方案：

项目架构中，对项目进行很多的优化，真实落在数据库服务器上请求非常少；因此数据库不需要进行大规模的扩容；在大多数企业中，数据库采用主从架构；

以上问题反馈的条件： 数据库不能随意扩容，因此当流量来到数据库的时候，数据库要做一些优化处理？？

影响数据库性能因素：

1、服务器硬件

2、操作系统

3、存储引擎

4、数据库表结构设计

5、SQL 语句

6、磁盘 IO

7、网卡流量

8、慢查询

总结来看：

1、低效 SQL

2、并发 cpu 的问题（SQL 并不支持多核心的 cpu 并发运算， 一个 SQL 只能在一个

cpu 上运行）

3、连接数据：max_connections

4、超高的 cpu 使用率

5、大表（数据多，字段多）

6、大事务

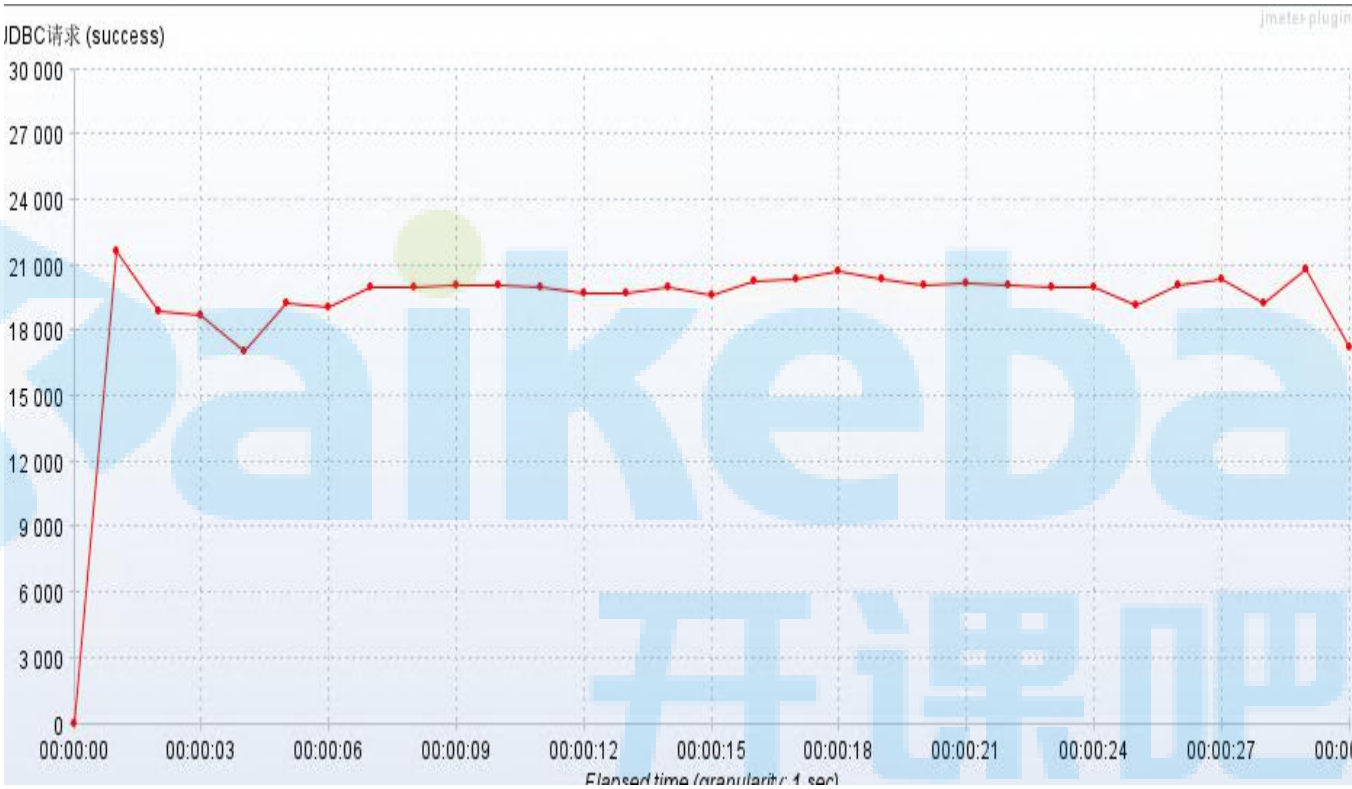
3.3 连接池相关参数

```
druid:
  #配置初始化大小、最小、最大
  initial-size: 1
  min-idle: 5
  max-active: 30
  max-wait: 10000
  time-between-eviction-runs-millis: 600000
  # 配置一个连接在池中最大空闲时间，单位是毫秒
  min-evictable-idle-time-millis: 300000
  # 设置从连接池获取连接时是否检查连接有效性，true 时，每次都检查;false 时，不检查
  test-on-borrow: true
  #设置往连接池归还连接时是否检查连接有效性，true 时，每次都检查;false 时，不检查
  test-on-return: true
  # 设置从连接池获取连接时是否检查连接有效性，true 时，如果连接空闲时间超过 minEvictableIdleTimeMillis 进行检查，否则不检查;false 时，不检查
  test-while-idle: true
  # 检验连接是否有效的查询语句。如果数据库 Driver 支持 ping()方法，则优先使用 ping()方法进行检查，否则使用 validationQuery 查询进行检查。(Oracle jdbc Driver 目前不支持 ping 方法)
  validation-query: select 1 from dual
  keep-alive: true
  remove-abandoned: true
  remove-abandoned-timeout: 80
  log-abandoned: true
  #打开 PSCache，并且指定每个连接上 PSCache 的大小，Oracle 等支持游标的数据库，打开此开关，会以数量级提升性能，具体查阅 PSCache 相关资料
  pool-prepared-statements: true
  max-pool-prepared-statement-per-connection-size: 20
  # 配置间隔多久启动一次 DestroyThread，对连接池内的连接才进行一次检
```

```
测，单位是毫秒。
  #检测时:
  #1.如果连接空闲并且超过minIdle 以外的连接，如果空闲时间超过
minEvictableIdleTimeMillis 设置的值则直接物理关闭。
  #2.在minIdle 以内的不处理。
```

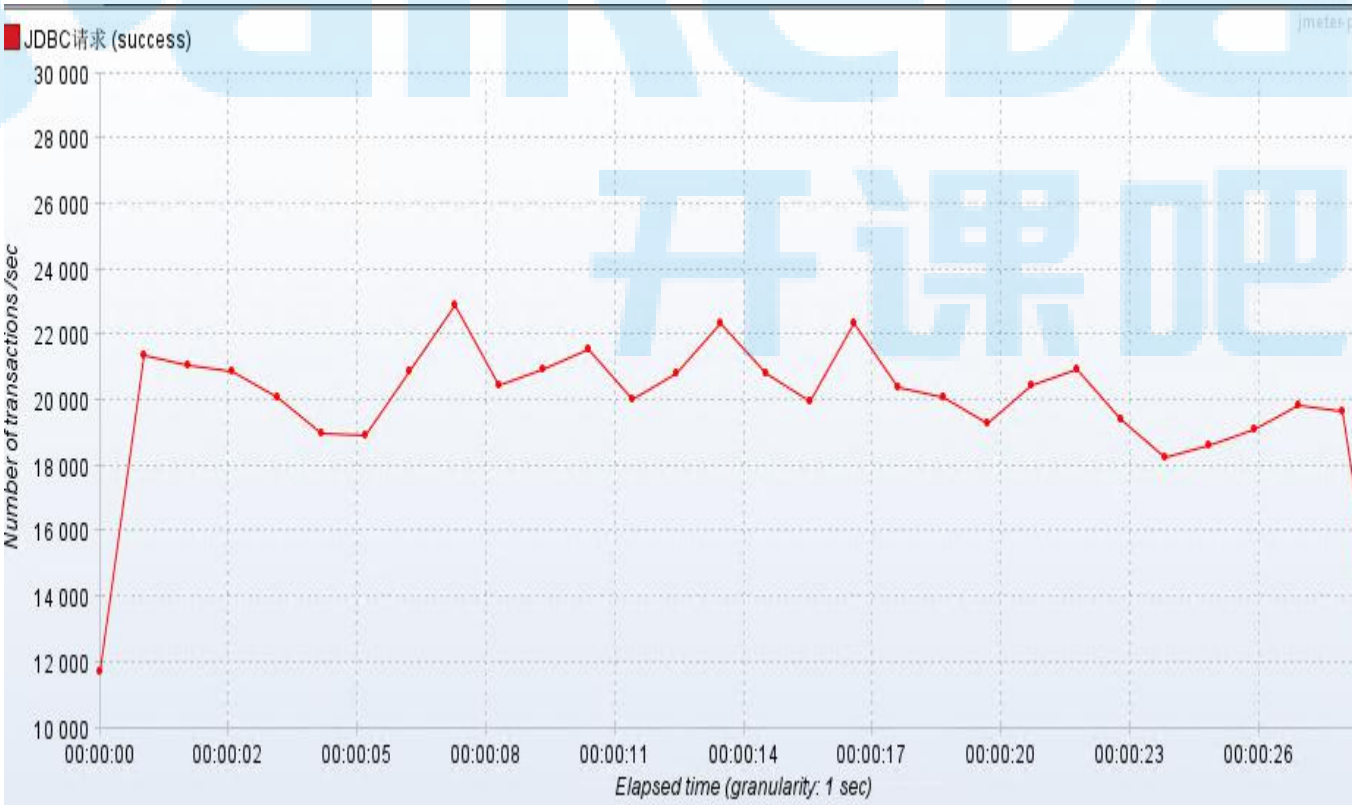
连接参数对象查询数据库的性能的影响： 连接数量最合适应该设置多少，才能满足数据库查询性能的需求？

1） 10 个链接，60w 测试样本



2）15 连接，可发发现性能提升

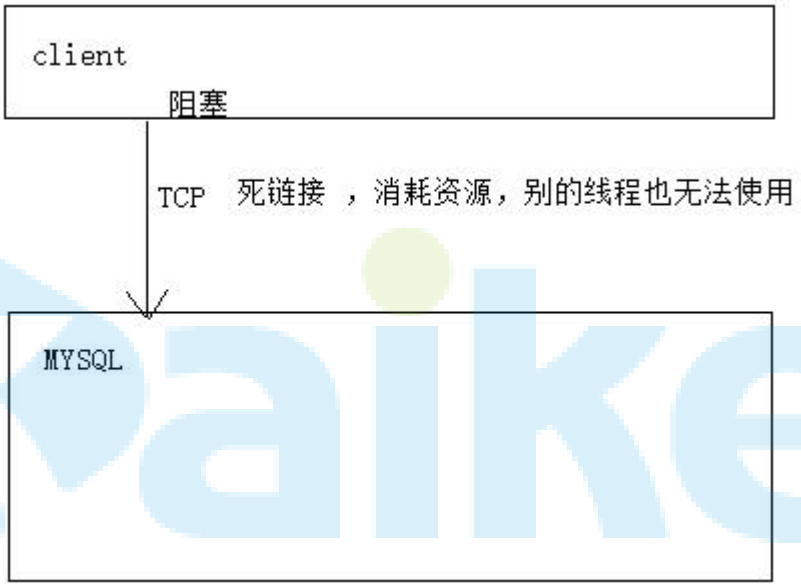
```
mysql> show variables like 'max_connections';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 151   |
+-----------------+-------+
1 row in set (0.00 sec)

mysql> set global max_connections = 600;
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like 'max_connections';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 600   |
+-----------------+-------+
1 row in set (0.00 sec)
```



总结： 经过测试，发现数据库最大连接数据控制在 10~15 个链接之间，才能达到数据库性能的最大化；

3.4 连接属性设置问题

connectionTimeout： 配置建立 TCP 连接的超时时间的设置属性参数，追加 JDBC 连接的后面
socketTimeout： 配置发送请求后等待响应的超时时间；



Jdbc:mysql://……&connectionTimeout=3000&socketTimeout=1200 (及时释放连接，不可用的连接)