并发通识&压力测试&服务端优化

一、并发&高并发

1、什么是并发?

在操作系统中，是指一个时间段中有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个处理机上运行。

就像前面提到的操作系统的时间片分时调度。打游戏和听音乐两件事情在同一个时间段内都是在同一台电脑上完成了从开始到结束的动作。那么，就可以说听音乐和打游戏是并发的。

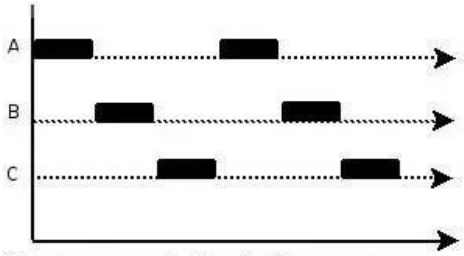
2、并发与并行

我们两个人在吃午饭。你在吃饭的整个过程中，吃了米饭、吃了蔬菜、吃了牛肉。吃米饭、吃蔬菜、吃牛肉这三件事其实就是并发执行的。

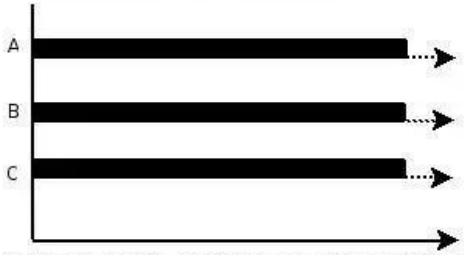
对于你来说，整个过程中看似是同时完成的的。但其实你是在吃不同的东西之间来回切换的。

还是我们两个人吃午饭。在吃饭过程中，你吃了米饭、蔬菜、牛肉。我也吃了米饭、蔬菜和牛肉。

我们两个人之间的吃饭就是并行的。两个人之间可以在同一时间点一起吃牛肉，或者一个吃牛肉，一个吃蔬菜。之间是互不影响的。



Concurrency : 1. Single Processor
2. logically simultaneous processing



Parallelism : 1. Multiprocessores, Multicore
2. Physically simultaneous processing

所以，并发是指在一段时间内宏观上多个程序同时运行。并行指的是同一个时刻，多个任务确实真的在同时运行。

并发和并行的区别：

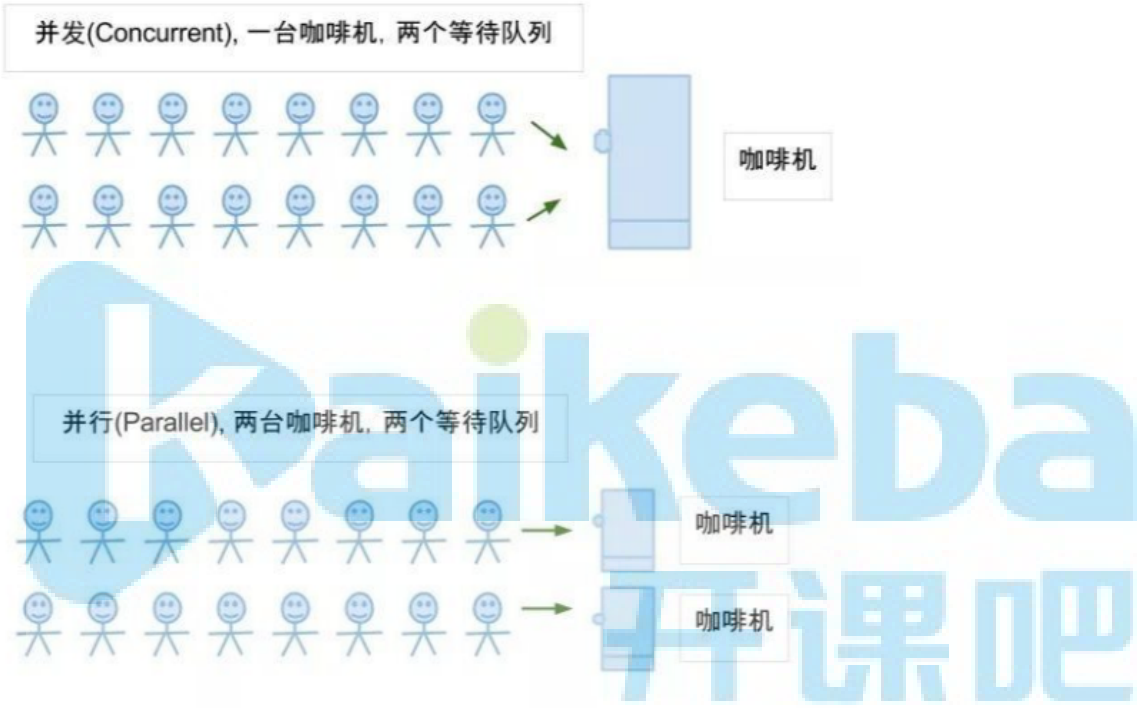
并发，指的是多个事情，在同一时间段内同时发生了。

并行，指的是多个事情，在同一时间点上同时发生了。

并发的多个任务之间是互相抢占资源的。

并行的多个任务之间是不互相抢占资源的、

只有在多CPU的情况中，才会发生并行。否则，看似同时发生的事情，其实都是并发执行的。



3、什么是高并发?

高并发和多线程"总是被一起提起，给人感觉两者好像相等，实则 高并发 ≠ 多线程（他们没有必然的直接联系）

多线程是完成任务的一种方法，高并发是系统运行的一种状态，通过多线程有助于系统承受高并发状态的实现。

高并发是一种系统运行过程中遇到的一种“短时间内遇到大量操作请求”的情况，主要发生在web系统集中大量访问或者socket端口集中性收到大量请求（例如：12306的抢票情况；天猫双十一活动）。

该情况的发生会导致系统在这段时间内执行大量操作，例如对资源的请求，数据库的操作等。如果高并发处理不好，不仅仅降低了用户的体验度（请求响应时间过长），同时可能导致系统宕机，严重的甚至导致OOM异常，系统停止工作等。

如果要想系统能够适应高并发状态，则需要从各个方面进行系统优化，包括，硬件、网络、系统架构、开发语言的选取、数据结构的运用、算法优化、数据库优化……而多线程只是其中解决方法之一。

实现高并发需要考虑：

（1）系统的架构设计，如何在架构层面减少不必要的处理（网络请求，数据库操作等）例如：使用Cache来减少IO次数，使用异步来增加单服务吞吐量，使用无锁数据结构来减少响应时间；

（2）网络拓扑优化减少网络请求时间、如何设计拓扑结构，分布式如何实现？

（3）系统代码级别的代码优化，使用什么设计模式来进行工作？哪些类需要使用单例，哪些需要尽量减少new操作？

（4）提高代码层面的运行效率、如何选取合适的数据结构进行数据存取？如何设计合适的算法？

（5）任务执行方式级别的同异步操作，在哪里使用同步，哪里使用异步？

（6）JVM调优，如何设置Heap、Stack、Eden的大小，如何选择GC策略,控制Full GC的频率？

（7）服务端调优（线程池，等待队列）

（8）数据库优化减少查询修改时间。数据库的选取？数据库引擎的选取？数据库表结构的设计？数据库索引、触发器等设计？是否使用读写分离？还是需要考虑使用数据仓库？

（9）缓存数据库的使用，如何选择缓存数据库？是Redis还是Memcache? 如何设计缓存机制？

（10）数据通信问题，如何选择通信方式？是使用TCP还是UDP，是使用长连接还是短连接？NIO还是BIO？netty、mina还是原生socket？

（11）操作系统选取，是使用winserver还是Linux？或者Unix？

（12）硬件配置？是8G内存还是32G，网卡10G还是1G? 例如：增加CPU核数如32核，升级更好的网卡如万兆，升级更好的硬盘如SSD，扩充硬盘容量如2T，扩充系统内存如128G；

以上的这些问题在高并发中都是必须要深入考虑的，就像木桶原理一样，只要其中的某一方面没有考虑到，都会造成系统瓶颈，影响整个系统的运行。而高并发问题不仅仅涉及面之广，同时又要求有足够的深度！！！

而多线程在这里只是在同/异步角度上解决高并发问题的其中的一个方法手段，是在同一时刻利用计算机闲置资源的一种方式。

多线程在解决高并发问题中所起到的作用就是使计算机的资源在每一时刻都能达到最大的利用率，不至于浪费计算机资源使其闲置。

4、你真的了解高并发吗?

高并发就是大家臆想的吹牛逼，其实大部分业务场景不存在并发竞争数据的情况，那么加服务 加机器基本上都能解决问题，你要事务压力大 那就分表，要是查询压力大 就主从 + 缓存，总有办法解决问题的。

```
# 所谓的亿级流量，100w级qps ，你真的懵逼了吗？？

例：某服务1天36万笔（粗粒度统计），平均RT在100ms，指标计算如下

**指标=(日量36万/10小时高峰期/3600秒)x5倍保险系数**=50qps、rt100ms

例：某服务近期监控请求峰值为1小时36万笔（粗粒度统计），平均RT在100ms，，指标计算如下

**指标=(时量36万/3600秒)x5倍保险系数**=500qps、rt100ms


# 衡量高并发的指标
Qps ， TPS ， RT ， 吞吐量
```

二、并发指标分析

1、吞吐量

在了解qps、tps、rt、并发数之前，首先我们应该明确一个系统的吞吐量到底代表什么含义，一般来说，系统吞吐量指的是系统的抗压、负载能力，代表一个系统每秒钟能承受的最大用户访问量。

一个系统的吞吐量通常由qps（tps）、并发数来决定，每个系统对这两个值都有一个相对极限值，只要某一项达到最大值，系统的吞吐量就上不去了。

所谓的系统吞吐量其实就是： 系统每秒请求数

2、QPS

Queries Per Second，每秒查询数，即是每秒能够响应的查询次数，注意这里的查询是指用户发出请求到服务器做出响应成功的次数，简单理解可以认为查询=请求request。

qps=每秒钟request数量

3、TPS

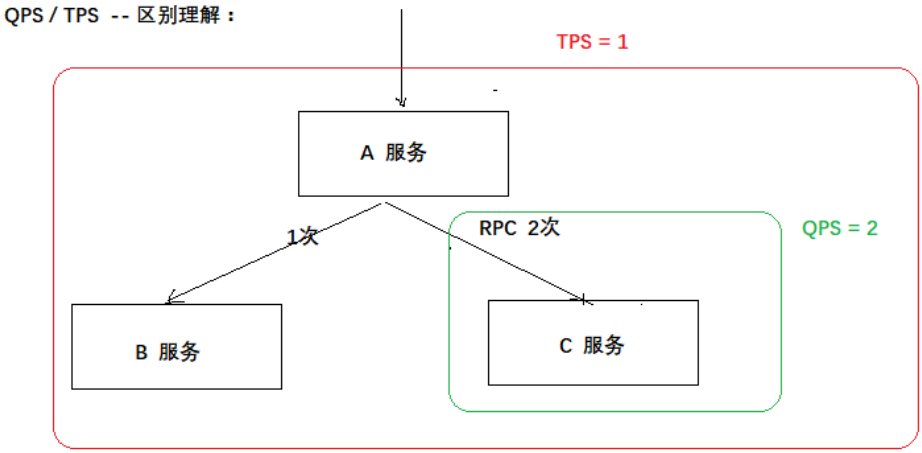
Transactions Per Second 的缩写，每秒处理的事务数。一个事务是指一个客户机向服务器发送请求然后服务器做出反应的过程。客户机在发送请求时开始计时，收到服务器响应后结束计时，以此来计算使用的时间和完成的事务个数。

> 针对单接口而言，TPS可以认为是等价于QPS的，比如访问一个页面/index.html，是一个TPS，而访问/index.html页面可能请求了3次服务器比如css、js、index接口，产生了3个QPS。

tps=每秒钟事务数量

> QPS和TPS区别个人理解如下：
>
> 1、Tps即每秒处理事务数，包括了用户请求服务器 服务器自己的内部处理 服务器返回给用户这三个过程，每秒能够完成N个这三个过程，Tps也就是N；
>
> 2、Qps基本类似于Tps，但是不同的是，对于一个页面的一次访问，形成一个Tps；但一次页面请求，可能产生多次对服务器的请求，服务器对这些请求，就可计入“Qps”之中。例子： 例如：访问一个页面会请求服务器3次，一次放，产生一个“T”，产生3个“Q”
>
> 例如：一个大胃王一秒能吃10个包子，一个女孩子0.1秒能吃1个包子，那么他们是不是一样的呢？答案是否定的，因为这个女孩子不可能在一秒钟吃下10个包子，她可能要吃很久。这个时候这个大胃王就相当于TPS，而这个女孩子则是QPS。虽然很相似，但其实是不同的。
>
> # 大部分情况不用纠结QPS TPS
> TPS=QPS（每秒请求数），所以大部分条件下，这两个概念不用纠结的!
>
> 举个例子，我需要进行一次查询，但这个查询需要调用A服务和B服务，而调用B服务需要2次调用，那么这种情况，以我查询这个场景成功作为一次事务的话，我一秒请求一笔就是1tps，当然对于A系统是1tps=1qps的.
>
> 但对于B系统而言，就是2qps，因为调用了两次（如果只看B服务的话，把每次请求当做一次事务的话2qps=2tps，还是可以等同的）所以仅仅是关注维度的不同，绝大多数时候我们不用去刻意区分的，毕竟我可以说我流程是1tps，B系统受到的双倍额压力是2tps的量(压测过程中也务必关注这样的流量放大服务，因为很有可能前面的服务抗的住，后面扛不住)，这样也是完全没有问题的。

如果单个接口请求，QPS = TPS ，但是观察这个变量的维度不一样，如果从请求的发起到请求的结束，且中间没有任何服务的远程调用，那么QPS = TPS 是没有任何疑义的，但是如果存在多个远程调用的话，就那么对于一台服务器 QPS = TPS 也是没有问题的，但是站在整个请求链路来说 ， QPS > TPS



4、RT

Response Time缩写，简单理解为系统从输入到输出的时间间隔，宽泛的来说，他代表从客户端发起请求到服务端接受到请求并响应所有数据的时间差。一般取平均响应时间。

对于RT，客户端和服务端是大不相同的，因为请求从客户端到服务端，需要经过广域网，所以客户端RT往往远大于服务端RT，同时客户端的RT往往决定着用户的真实体验，服务端RT往往是评估我们系统好坏的一个关键因素。

> 在开发过程中，我们一定面临过很多的线程数量的配置问题，这种问题往往让人摸不到头脑，往往都是拍脑袋给出一个线程池的数量，但这可能恰恰是不靠谱的，过小的话会导致请求RT极具增加，过大也一样RT也会升高。所以对于最佳线程数的评估往往比较麻烦。

三、压测报告瓶颈分析

等到服务上线后，在业务压力的冲击下，会发现程序运行非常的慢，或者是宕机，莫名其妙的出现各种问题，只会进行一些无脑的扩容，扩容真的能解决问题吗？？

可能能解决问题，但是同时也会带来一些其他的问题，因此在项目上线之前，还必须要有一步性能压力测试的步骤，以便于发现服务的一些问题，提前对服务的问题进行修复，优化等等。

1、应用部署

1.1、服务打包

项目打包： 可以使用idea直接打包上传，也可以在gitlab服务器直接通过maven进行打包，或者使用jenkins来进行打包，现在我们先使用idea的maven进行打包。

```
# 注意打包必须的依赖
<build>
        <plugins>
```

```
            <--此依赖必须有，否则项目的依赖包无法被打进项目：mysql.jar,spring*.jar都无法打包进入-->
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
            <!-- 编译项目，然后打包，只会打包自己的代码 -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

注意： 打包服务的时候必须注意服务的配套的ip地址,由于此时服务和mysql，redis都在同一个服务器上，因此连接访问地址设置为localhost即可。

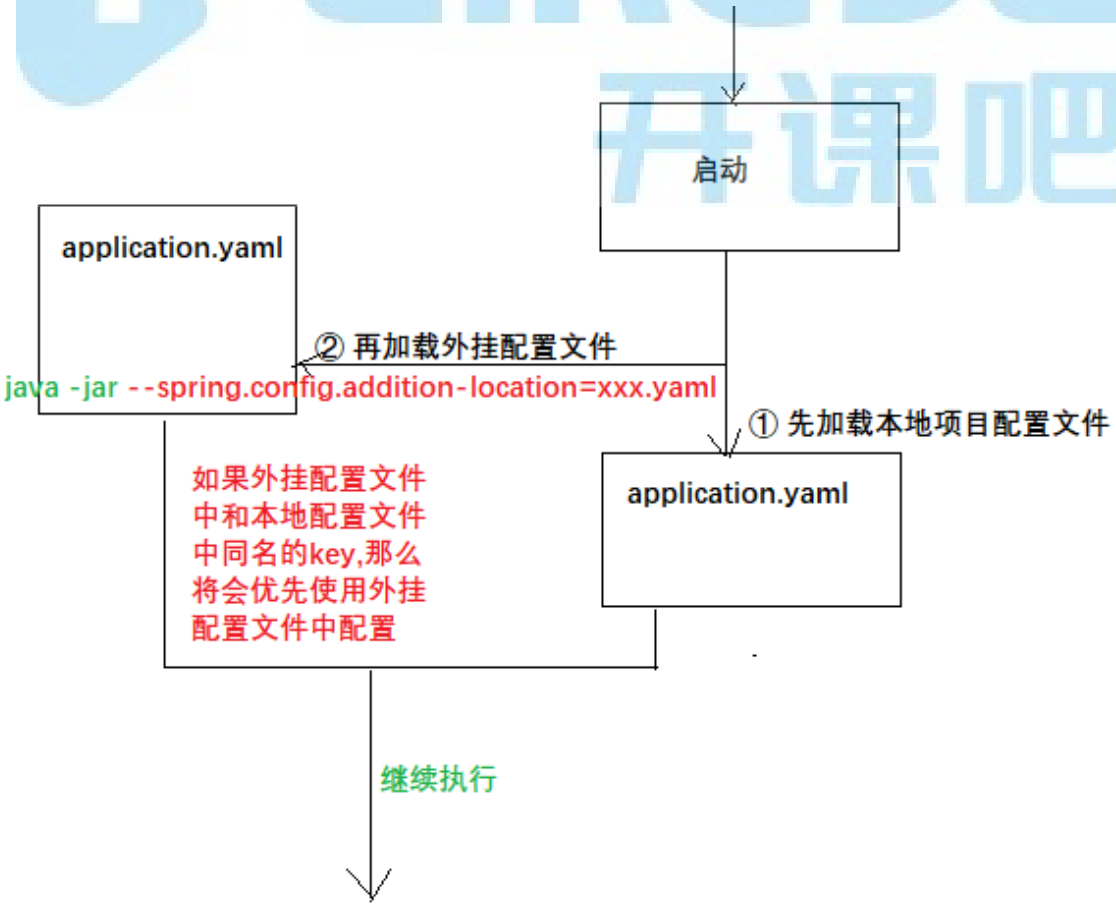
```
server:
  port: 9000
spring:
  application:
    name: sugo-seckill-web
  datasource:
    url: jdbc:mysql://127.0.0.1:3306/shop?useUnicode=true&characterEncoding=utf8&autoReconnect=true&allowMultiQueries=true
#    url: jdbc:mysql://47.113.81.149:3306/shop?useUnicode=true&characterEncoding=utf8&autoReconnect=true&allowMultiQueries=true
    username: root
    password: root
    driver-class-name: com.mysql.jdbc.Driver
    druid:
      #配置初始化大小、最小、最大
      initial-size: 1
      min-idle: 5
      max-active: 5
      max-wait: 20000
      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。
  redis:
    host: 127.0.0.1
    port: 6379
mybatis:
  type-aliases-package: com.supergo.pojo
mapper:
  not-empty: false
  identity: mysql
```

1.2、打包上传

启动命令： java -jar jshop-web-1.0-SNAPSHOT.jar

注意：服务器部署的时候，由于服务器环境的不同，往往都需要额外的修改服务的配置文件，重新编译打包，必须服务器ip地址，本地开发环境的ip和线上的ip是不一样的，部署的时候，每次都需要修改这些配置，非常麻烦。因此服务部署时候应该具有一个外挂配置文件的能力。



```
#启动命令,注意：配置文件的名称必须是application.yaml，或者application.properties
java -jar xxx.jar --spring.config.addition-
location=/usr/local/src/application.yaml
```

1.3、启动脚本

创建deploy.sh 这样一个shell脚本文件，执行java程序的后端启动工作

```
#使用nohup启动，使得Java进程在后台以进程模试运行
nohup java -Xms500m -Xmx500m -XX:NewSize=300m -XX:MaxNewSize=300m -jar jshop-
web-1.0-SNAPSHOT.jar --spring.config.addition-location=application.yaml >
log.log 2>&1 &

#授权
chmod 777 deploy.sh
```

浏览器接口测试访问，发现服务已经启动成功：

2、开始压测

2.1、压测准备

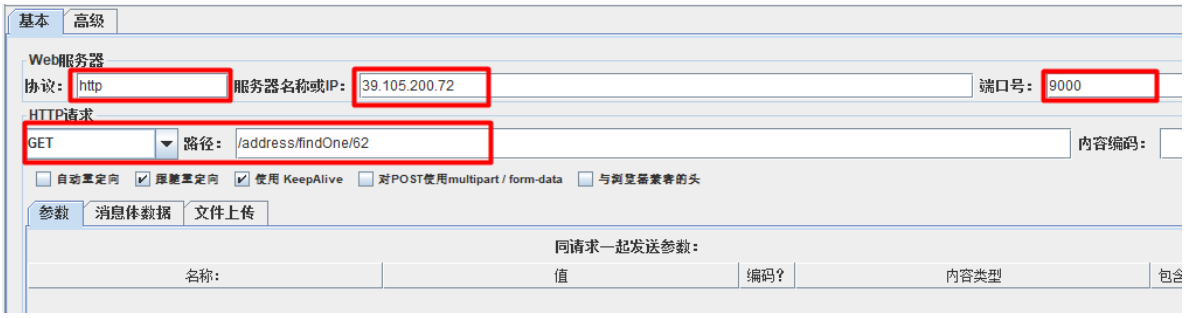
压测目标：

线程梯度：500、1000，1500,2000,2500，3000个线程，即模拟这些数目的用户并发;

时间设置：Ramp-up period(inseconds)的值设为1(即1s，5s,10s启动500、1000，1500,2000,2500，3000并发访问)

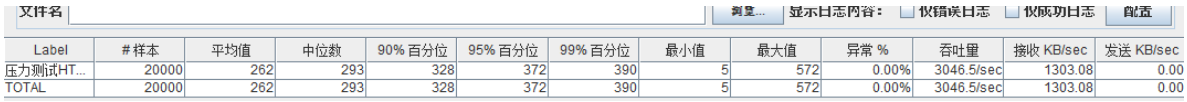
循环次数：50次

1）设置压测请求



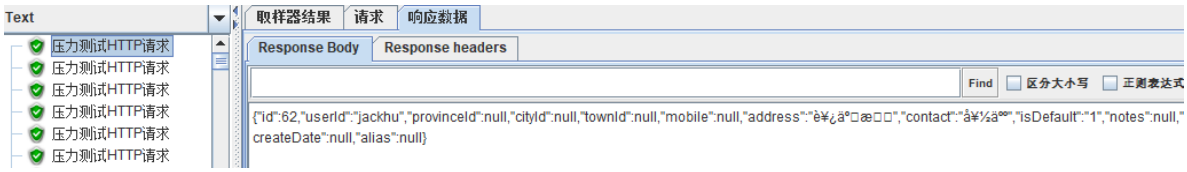
2.2、添加监听器

聚合报告：添加聚合报告

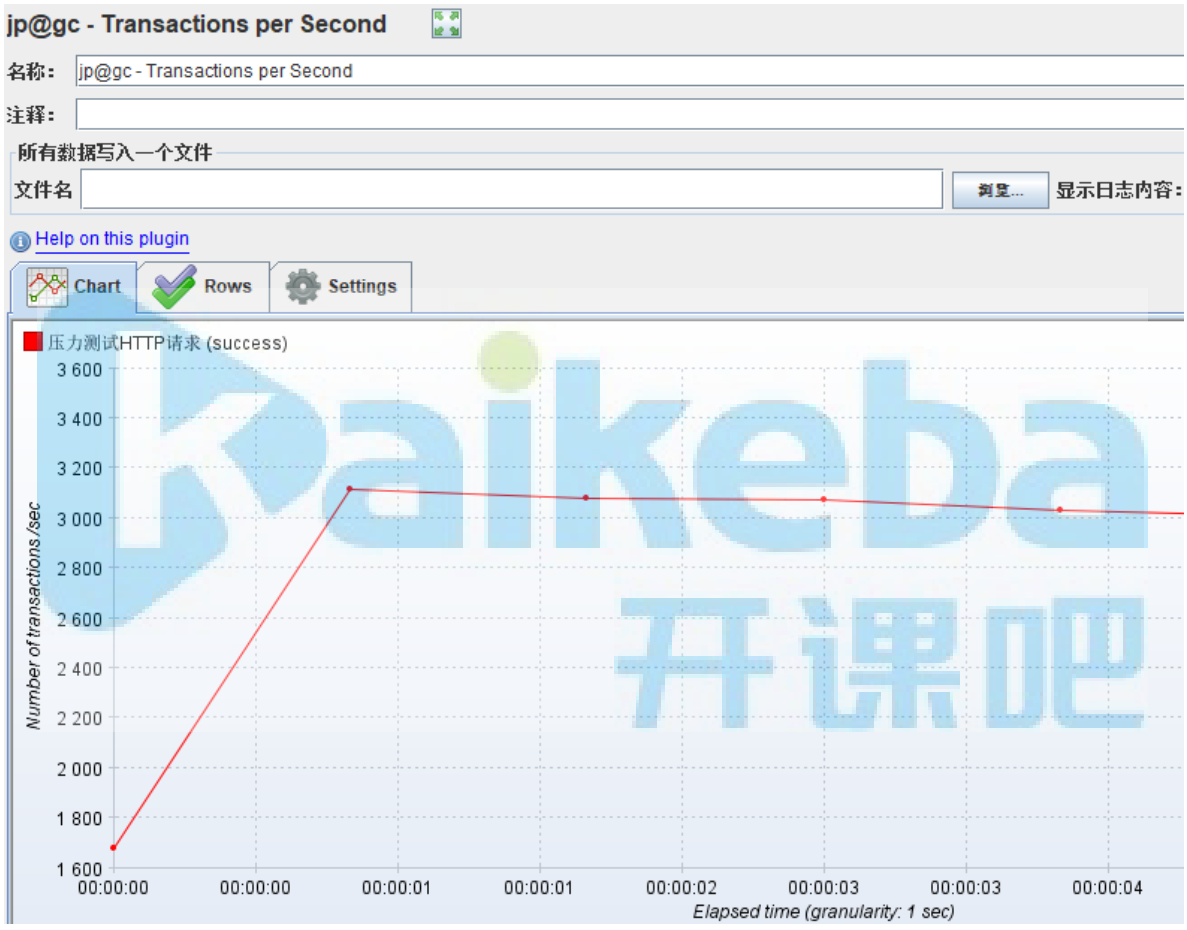


Label	# 样本	平均值	中位数	90% 百分位	95% 百分位	99% 百分位	最小值	最大值	异常 %	吞吐量	接收 KB/sec	发送 KB/sec
压力测试HT...	20000	262	293	328	372	390	5	572	0.00%	3046.5/sec	1303.08	0.00
TOTAL	20000	262	293	328	372	390	5	572	0.00%	3046.5/sec	1303.08	0.00

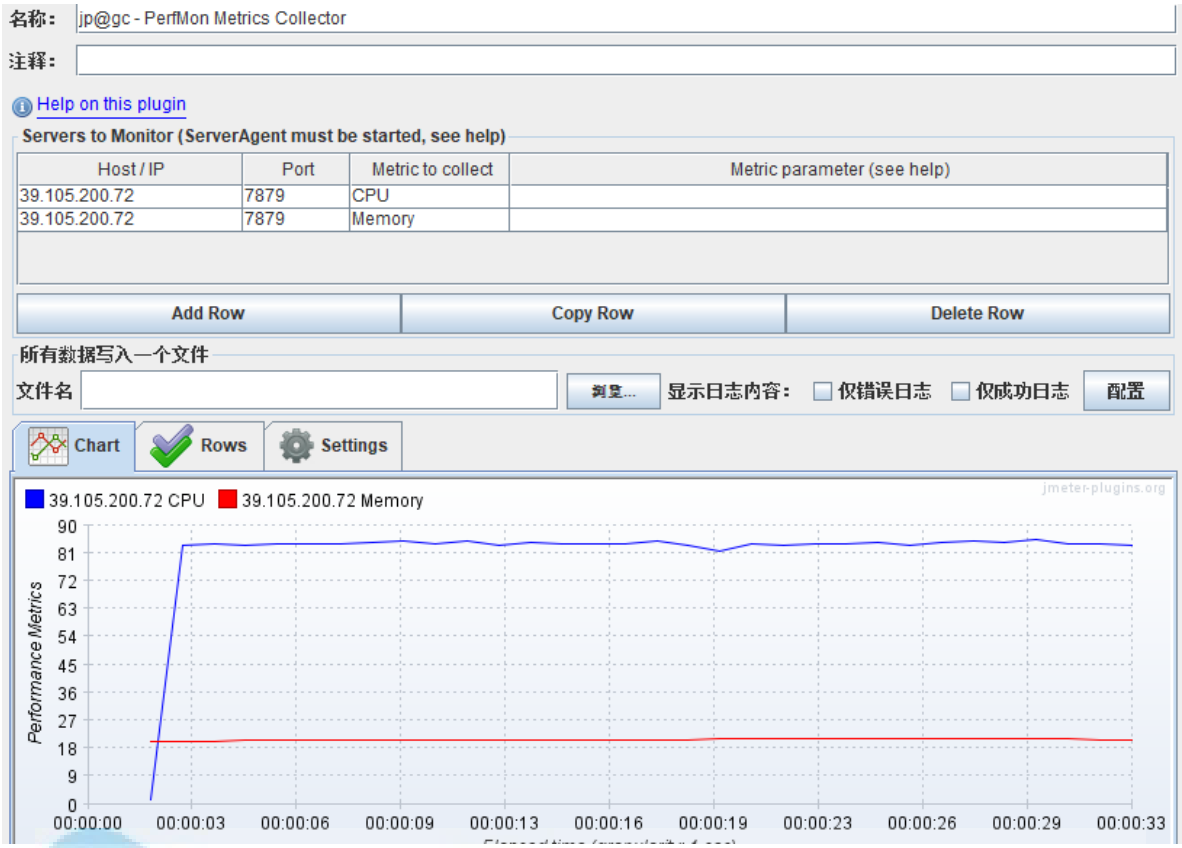
查看结果树：添加查看结果树



TPS统计分析：每秒事务树



服务器性能监控：CPU、内存、IO



注意： 对于jmeter的cpu监控来说，只能监控单核cpu，没有太大的参考价值，真实测试可以使用top指令观察cpu使用情况。

3、参数原理

我们设置线程数 n = 5，循环次数a = 1000，请求www.google.com，得到聚合报告如图：

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %
HTTP请求	5000	206	205	219	228	295	165	729	0
总体	5000	206	205	219	228	295	165	729	0

图中得到谷歌首页的平均请求时间大约为t = 0.2秒

这里，我们为了方便分析，将Ramp-Up Period 设置为T = 10秒（实际合理的时间后面会说明）

依然是n = 5，得到 S = (T- T/n) = 8 ，也就是说，从第一个线程启动到第8秒的时候，最后一个线程开始启动，若需要在最后一个线程启动的时候第一个线程仍未关闭，则需要满足 a·t > S ，已知S = 8,t = 0.2，得到 a > 40 。

```
线程数：n=5
循环次数：a = 1000
平均响应时间： t = 0.2s
Ramp-Up Period T=10s
S = （T-T/n） = 8
# 循环次数 * 平均响应时间
a * t > S  ==>  a > S/t = 40

Ramp-Up Period:
```

【1】决定多长时间启动所有线程。如果使用10个线程，ramp-up period是100秒，那么JMeter用100秒使所有10个线程启动并运行。每个线程会在上一个线程启动后10秒（100/10）启动。Ramp-up需要要充足长以避免在启动测试时有一个太大的工作负载，并且要充足小以至于最后一个线程在第一个完成前启动。 一般设置ramp-up=线程数启动，并上下调整到所需的。

【2】用于告知JMeter 要在多长时间内建立全部的线程。默认值是0。如果未指定ramp-up period ，也就是说ramp-up period 为零， JMeter 将立即建立所有线程。假设ramp-up period 设置成T 秒， 全部线程数设置成N个， JMeter 将每隔T/N秒建立一个线程。

【3】Ramp-Up Period(in-seconds)代表隔多长时间执行，0代表同时并发

4、性能参数分析

4.1、TPS吞吐能力

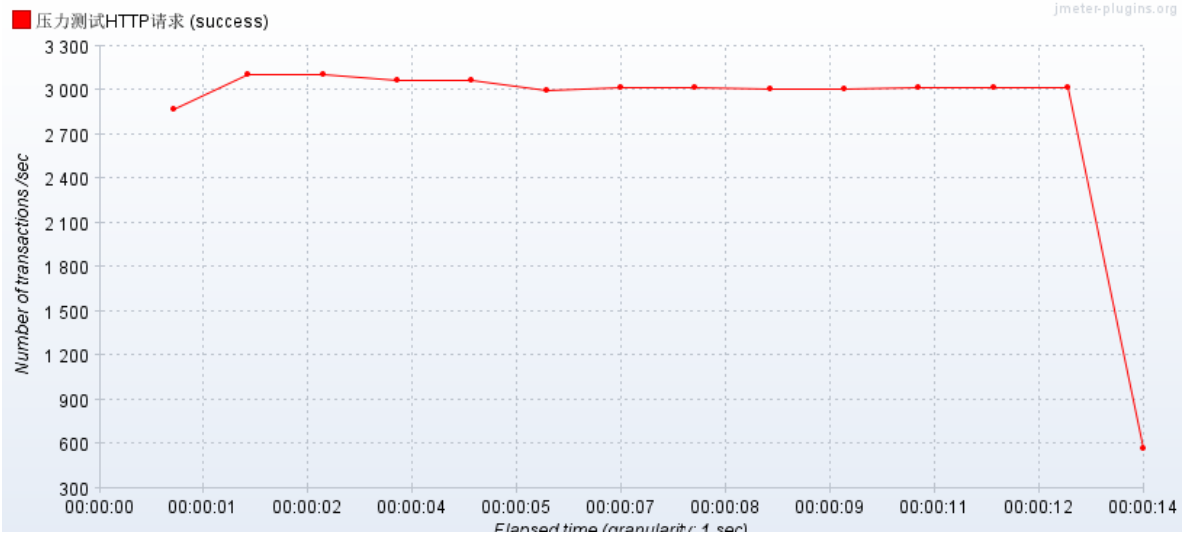
聚合报告：TPS 3039，但是在这里看不见TPS的峰值是多少，需要进行改进

Label	#样本	平均值	中位数	90% 百分...	95% 百分...	99% 百分...	最小值	最大值	异常 %	吞吐量	接收 KB/s...	发送 KB/s...
压力测试...	40000	339	374	520	530	584	4	763	0.00%	3039.5/sec	1300.10	0.00
TOTAL	40000	339	374	520	530	584	4	763	0.00%	3039.5/sec	1300.10	0.00

样本（sample）：发送请求的总样本数量
平均值（average）：平均的响应时间
中位数（median）：中位数的响应时间，50%请求的响应时间
90%百分位（90% Line）：90%的请求的响应时间，意思就是说90%的请求是<=1765ms返回，另外10%的请求是大于等于1765ms返回的。

95%百分位（95% Line）：95%的请求的响应时间，95%的请求都落在1920ms之内返回的
99%百分位（99% Line）：99%的请求的响应时间
最小值(min)：请求返回的最小时间，其中一个用时最少的请求
最大值(max)：请求返回的最大时间，其中一个用时最大的请求
异常（error）：出现错误的百分比，错误率=错误的请求的数量/请求的总数
吞吐量TPS（throughout）：吞吐能力，这个才是我们需要的并发数
Received KB/sec----每秒从服务器端接收到的数据量
Sent KB/sec----每秒从客户端发送的请求的数量

使用Tps监控曲线图：



可以看见，TPS在1s+的位置，TPS能力最大，其他时候TPS都一直平稳过渡。

随着并发压力的加大，以及时间延长，系统性能所发生的变化。正常情况下，平均采样响应时长曲线应该是平滑的，并大致平行于图形下边界。

4.2、性能曲线

随着并发压力的加大，以及时间延长，系统性能所发生的变化。正常情况下，平均采样响应时长曲线应该是平滑的，并大致平行于图形下边界。

可能存在性能问题：

①平均值在初始阶段跳升，而后逐渐平稳起来

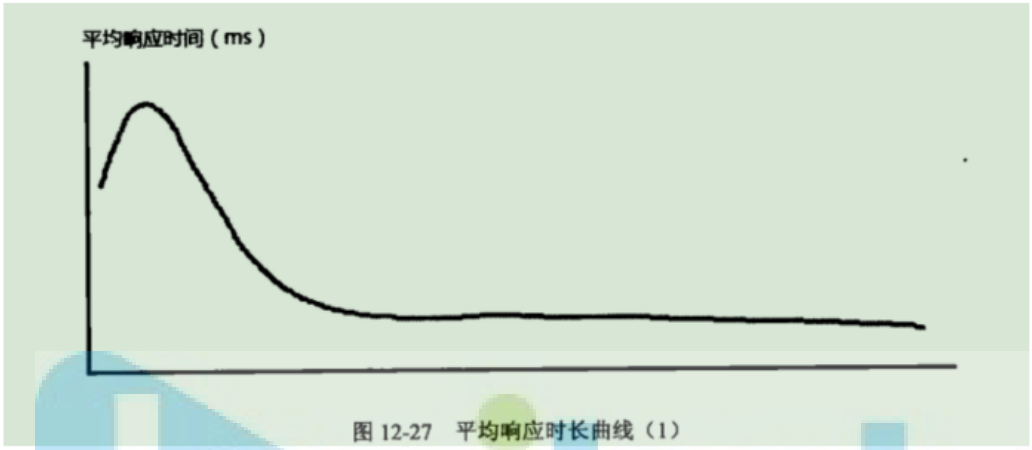


图 12-27　平均响应时长曲线（1）

一是系统在初始阶段存在性能缺陷，需要进一步优化，如数据库查询缓慢

二是系统有缓存机制，而性能测试数据在测试期间没有变化，如此一来同样的数据在初始阶段的响应时长肯定较慢；这属于性能测试数据准备的问题，不是性能缺陷，需调整后在测试

三是系统架构设计导致的固有现象，例如在系统接收到第一个请求后，才去建立应用服务器到数据库的链接，后续一段时间内不会释放连接。

②平均值持续增大，图片变得越来越陡峭

一是可能存在内存泄漏，此时可以通过监控系统日志、监控应用服务器状态等常见方法，来定位问题。

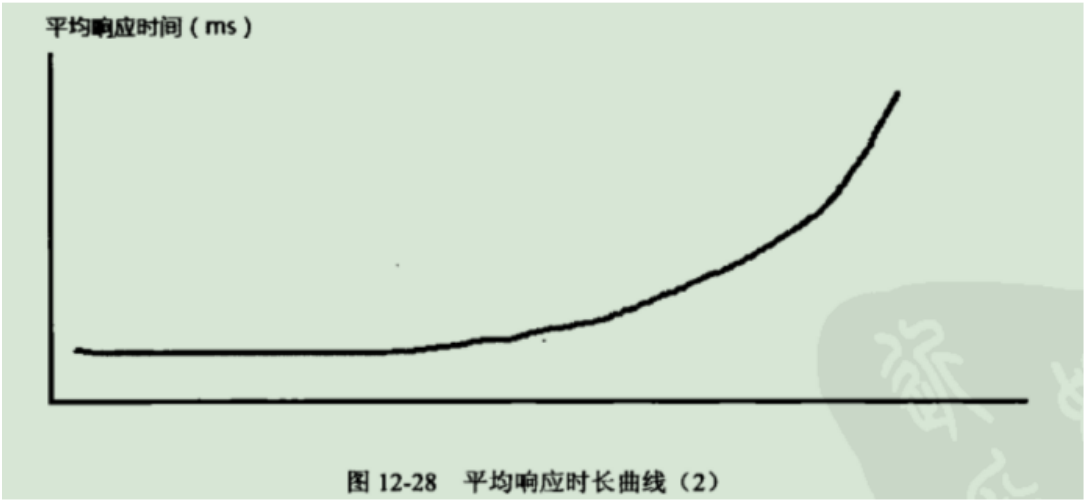


图 12-28　平均响应时长曲线（2）

③平均值在性能测试期间，突然发生跳变，然后又恢复正常

一是可能存在系统性能缺陷

二是可能由于测试环境不稳定所造成的（检查应用服务器状态【CPU占用、内存占用】或者检查测试环境网络是否存在拥塞）

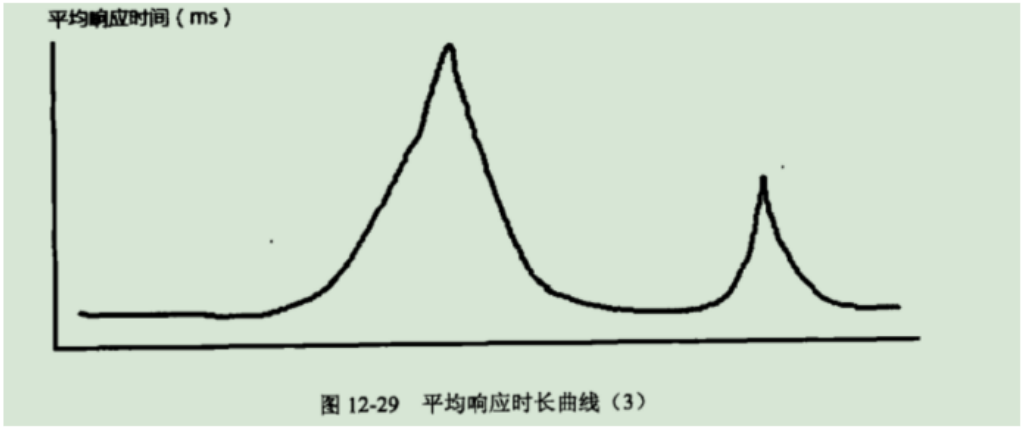


图 12-29　平均响应时长曲线（3）

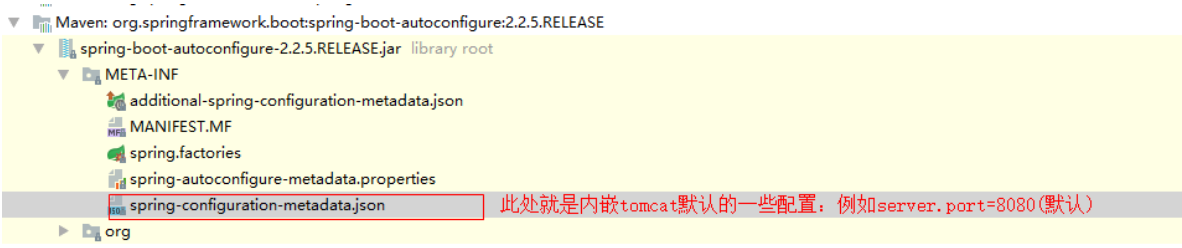
四、服务优化

1、线程数量提升

```
# server端线程数到245就已经上不去了，导致服务端的性能提升不上去
# pstree 查询线程数量
jps -l
# 查询所有线程
pstree -p pid
# 统计线程数量
pstree -p pid | wc -l
```

2、内嵌配置

Springboot开发的服务使用内嵌的tomcat服务器来启动服务，那么tomcat配置使用的是默认配置，我们需要对tomcat配置进行一些适当的优化，让tomcat性能得以提升。



当然内嵌tomcat内嵌线程池的配置也是比较小的，我们可以通过外挂配置文件，把tomcat的相关配置进行改写，然后重新启动服务器进行测试。修改配置如下所示：

Tomcat的maxConnections、maxThreads、acceptCount三大配置，分别表示最大连接数，最大线程数、最大的等待数，可以通过application.yml配置文件来改变这个三个值，一个标准的示例如下：

```yaml
server:
  tomcat:
    uri-encoding: UTF-8
    #最大工作线程数，默认200，4核8g内存，线程数经验值800
    #操作系统做线程之间的切换调度是有系统开销的，所以不是越多越好。
    max-threads: 1000
    # 等待队列长度，默认100
   accept-count: 1000
   max-connections: 20000
    # 最小工作空闲线程数，默认10，适当增大一些，以便应对突然增长的访问量
   min-spare-threads: 100
```

1)、accept-count：最大等待数

官方文档的说明为：当所有的请求处理线程都在使用时，所能接收的连接请求的队列的最大长度。当队列已满时，任何的连接请求都将被拒绝。accept-count的默认值为100。
详细的来说：当调用HTTP请求数达到tomcat的最大线程数时，还有新的HTTP请求到来，这时tomcat会将该请求放在等待队列中，这个acceptCount就是指能够接受的最大等待数，默认100。如果等待队列也被放满了，这个时候再来新的请求就会被tomcat拒绝（connection refused）。

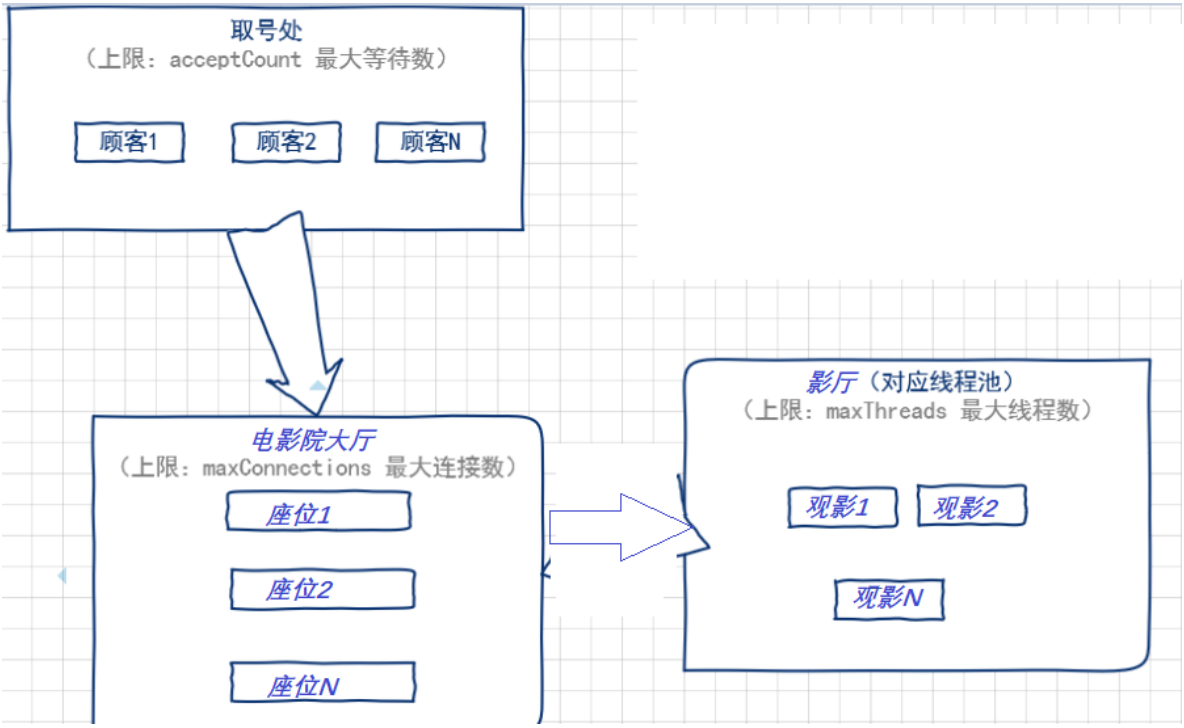
2)、maxThreads：最大线程数

每一次HTTP请求到达Web服务，tomcat都会创建一个线程来处理该请求，那么最大线程数决定了Web服务容器可以同时处理多少个请求。maxThreads默认200，肯定建议增加。但是，增加线程是有成本的，更多的线程，不仅仅会带来更多的线程上下文切换成本，而且意味着带来更多的内存消耗。JVM中默认情况下在创建新线程时会分配大小为1M的线程栈，所以，更多的线程异味着需要更多的内存。线程数的经验值为：1核2g内存为200，线程数经验值200；4核8g内存，线程数经验值800。

3)、maxConnections：最大连接数

官方文档的说明为：

这个参数是指在同一时间，tomcat能够接受的最大连接数。对于Java的阻塞式BIO，默认值是maxthreads的值；如果在BIO模式使用定制的Executor执行器，默认值将是执行器中maxthreads的值。对于Java 新的NIO模式，maxConnections 默认值是10000。
对于windows上APR/native IO模式，maxConnections默认值为8192，这是出于性能原因，如果配置的值不是1024的倍数，maxConnections 的实际值将减少到1024的最大倍数。
如果设置为-1，则禁用maxconnections功能，表示不限制tomcat容器的连接数。
maxConnections和accept-count的关系为：当连接数达到最大值maxConnections后，系统会继续接收连接，但不会超过acceptCount的值。



3、性能对比

未优化测试性能对比表：

并发数 / 5s	样本数	平均响应时间（ms）	吞吐量(TPS)	错误率	KB/sec
200	10000	7	1849	0	791
300	15000	14	2654	0	1135
500	25000	71	2925	0	1251
700	35000	141	2865	0	1225
1000	50000	244	2891	0	1236
1500	75000	418	2910	0	1224
2000	100000	553	2920	0	1249
3000	150000	889	2533	0.05	1086
3200					
3500					
4000					
4500					
5000					

优化后性能对比表：

```
# 优化的参数配置
server:
  tomcat:
    accept-count: 800
    max-connections: 20000
    max-threads: 800
    min-spare-threads: 100
```

性能测试对比：

并发数 / 5s	样本数	平均响应时间（ms）	吞吐量(TPS)	错误率	KB/sec
200	10000	11	1836	0	785
300	15000	17	2486	0	1063
500	25000	85	2693	0	1152
700	35000	151	2769	0	1184
1000	50000	260	2736	0	1171
1500	75000	445	2766	0	1183
2000	100000	582	2813	0	1203
3000	150000	937	2794	0	1195
3200	160000	1147	2429	0	1039
3500	175000	1150	2796	0.05	1198
4000	200000	1292	2852	0	1220
4500	225000	1428	2825	0	1208
5000	250000	1581	2867	0.03	1228

4、千万级数据

模拟千万级别的数据规模，使得查询耗时增大，然后再压力测试，才能看出效果。否则这个操作基本是一个不耗时操作，优化是没有效果的。

```
package com.sugo.seckill.web.batch;

/**
 * @ClassName DataDemoTest
 * @Description
 * @Author ithubin
 * @Date 2020/8/11 10:06
 * @Version V1.0
 **/
```

```
public class DataDemoTest {
    public static void main(String[] args) throws ClassNotFoundException,
SQLException {
        final String url = "jdbc:mysql://39.105.204.66:3306/shop?
useUnicode=true&characterEncoding=utf8&autoReconnect=true&allowMultiQueries=true
" ;
        final String name = "com.mysql.jdbc.Driver" ;
        final String user = "root" ;
        final String password = "root" ;
        Connection conn = null ;
        Class.forName(name); //指定连接类型
        conn = DriverManager.getConnection(url, user, password); //获取连接
        if (conn!= null ) {
            System.out.println( "获取连接成功" );
            insert(conn);
        } else {
            System.out.println( "获取连接失败" );
        }
    }
    public static void insert(Connection conn) {
        // 开始时间
        Long begin = new Date().getTime();
        // sql前缀


        String prefix = "insert  into
tb_seckill_goods(product_id,image,images,mark," +

 "title,info,price,cost_price,ot_price,give_integral,sort,stock,stock_count," +

 "sales,unit_name,postage,description,start_time,stop_time,add_time,status," +

 "is_postage,is_hot,is_del,num,is_show,end_time_date,start_time_date,time_id,ver
sion) values" ;
        try {
            // 保存sql后缀
            StringBuffer suffix = new StringBuffer();
            // 设置事务为非自动提交
            conn.setAutoCommit( false );
            // 比起st，pst会更好些
            PreparedStatement pst = (PreparedStatement) conn.prepareStatement(
"" );
            //准备执行语句
            // 外层循环，总提交事务次数
            for ( int i = 1 ; i <= 100 ; i++) {
                suffix = new StringBuffer();
                // 第j次提交步长
                for ( int j = 1 ; j <= 100000 ; j++) {
                    // 构建SQL后缀
                   String str =  "
(30,'https://img.kaikeba.com/under_line_shalong_one.png','https://img.kaikeba.co
m/under_line_shalong_one.png',NULL,'开课吧java架构师','开课吧课
程',9.90,50.00,99.00,0.00,1,100,375,0,'开课吧 秒杀实战落地开启最强实战!',10.00,'开课吧
课程',1590940800,1593446400,'1592997034',1,0,0,0,1,1,'2020-06-30 00:00:00','2020-
06-01 00:00:00',212,0),";
                    suffix.append(str);
                }
                // 构建完整SQL
```

```
                String sql = prefix + suffix.substring( 0 , suffix.length() - 1
);
                // 添加执行SQL
                pst.addBatch(sql);
                // 执行操作
                pst.executeBatch();
                // 提交事务
                conn.commit();
                // 清空上一次添加的数据
                suffix = new StringBuffer();
            }
            // 头等连接
            pst.close();
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        // 结束时间
        Long end = new Date().getTime();
        // 耗时
        System.out.println( "1000万条数据插入花费时间 : " + (end - begin) / 1000 +
" s" );
        System.out.println( "插入完成" );
    }


}
```

5、keepalive

长连接会消耗大量资源，如果连接不能及时释放，系统的TPS就提升不上去，因此我们需要改造web服务，提升web服务的长连接的性能。

```
package com.sugo.seckill.web.config;

//当Spring容器内没有TomcatEmbeddedServletContainerFactory这个bean时，会吧此bean加载进
spring容器中
@Component
public class WebServerConfig implements
WebServerFactoryCustomizer<ConfigurableWebServerFactory> {
    @Override
    public void customize(ConfigurableWebServerFactory
configurableWebServerFactory) {
            //使用对应工厂类提供给我们的接口定制化我们的tomcat connector

((TomcatServletWebServerFactory)configurableWebServerFactory).addConnectorCustom
izers(new TomcatConnectorCustomizer() {
            @Override
            public void customize(Connector connector) {
                Http11NioProtocol protocol = (Http11NioProtocol)
connector.getProtocolHandler();
```

```
                //定制化keepalivetimeout,设置30秒内没有请求则服务端自动断开keepalive链
接
                protocol.setKeepAliveTimeout(30000);
                //当客户端发送超过10000个请求则自动断开keepalive链接
                protocol.setMaxKeepAliveRequests(10000);
            }
        });
    }
}
```

五、常用性能分析方法

Java 语言是当前互联网应用最为广泛的语言，作为一名 Java 程序猿，当业务相对比较稳定之后平常工作除了 coding 之外，大部分时间（70%~80%）是会用来排查突发或者周期性的线上问题。

由于业务应用 bug(本身或引入第三方库)、环境原因、硬件问题等原因，Java 线上服务出现故障 / 问题几乎不可避免。例如，常见的现象包括部分请求超时、用户明显感受到系统发生卡顿等等。

尽快线上问题从系统表象来看非常明显，但排查深究其发生的原因还是比较困难的，因此对开发测试或者是运维的同学产生了许多的困扰。

排查定位线上问题是具有一定技巧或者说是经验规律的，排查者如果对业务系统了解得越深入，那么相对来说定位也会容易一些。

不管怎么说，掌握 Java 服务线上问题排查思路并能够熟练排查问题常用工具 / 命令 / 平台是每一个 Java 程序猿进阶必须掌握的实战技能。

1、Java 服务常见线上问题

所有 Java 服务的线上问题从系统表象来看归结起来总共有四方面：CPU、内存、磁盘、网络。例如 CPU 使用率峰值突然飚高、内存溢出 (泄露)、磁盘满了、网络流量异常、FullGC 等等问题。

基于这些现象我们可以将线上问题分成两大类: 系统异常、业务服务异常。

1.1、 系统异常

常见的系统异常现象包括：CPU 占用率过高、CPU 上下文切换频率次数较高、磁盘满了、磁盘 I/O 过于频繁、网络流量异常 (连接数过多)、系统可用内存长期处于较低值 (导致 oom killer) 等等。

这些问题可以通过 top(cpu)、free(内存)、df(磁盘)、dstat(网络流量)、pstack、vmstat、strace(底层系统调用) 等工具获取系统异常现象数据。

此外，如果对系统以及应用进行排查后，均未发现异常现象的更笨原因，那么也有可能是外部基础设施如 IAAS 平台本身引发的问题。

例如运营商网络或者云服务提供商偶尔可能也会发生一些故障问题，你的引用只有某个区域 如北京用户访问系统时发生服务不可用现象，那么极有可能是这些原因导致的。

1.2、业务服务异常

常见的业务服务异常现象包括: PV 量过高、服务调用耗时异常、线程死锁、多线程并发问题、频繁进行 Full GC、异常安全攻击扫描等。

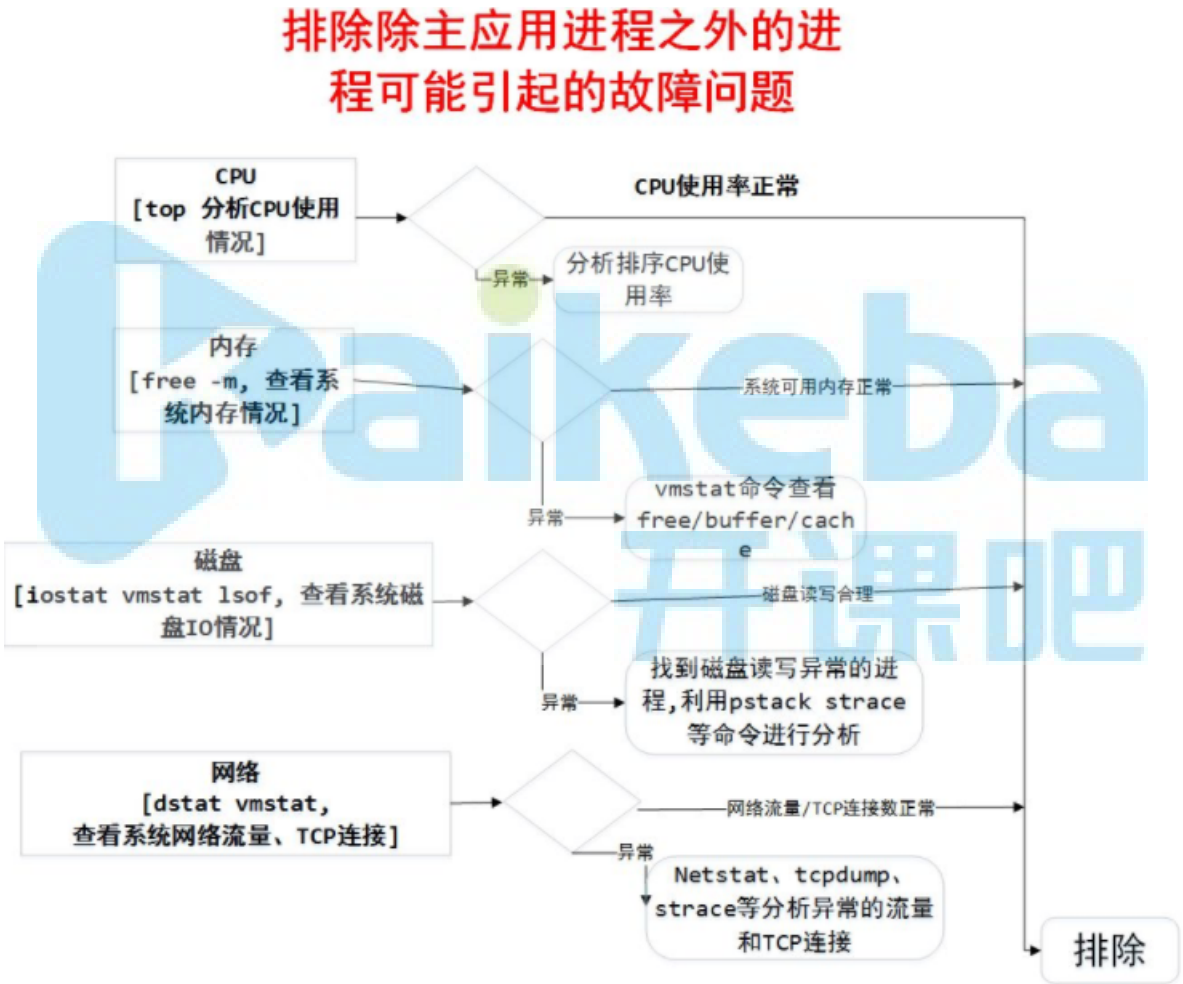
2、问题定位

我们一般会采用排除法，从外部排查到内部排查的方式来定位线上服务问题。

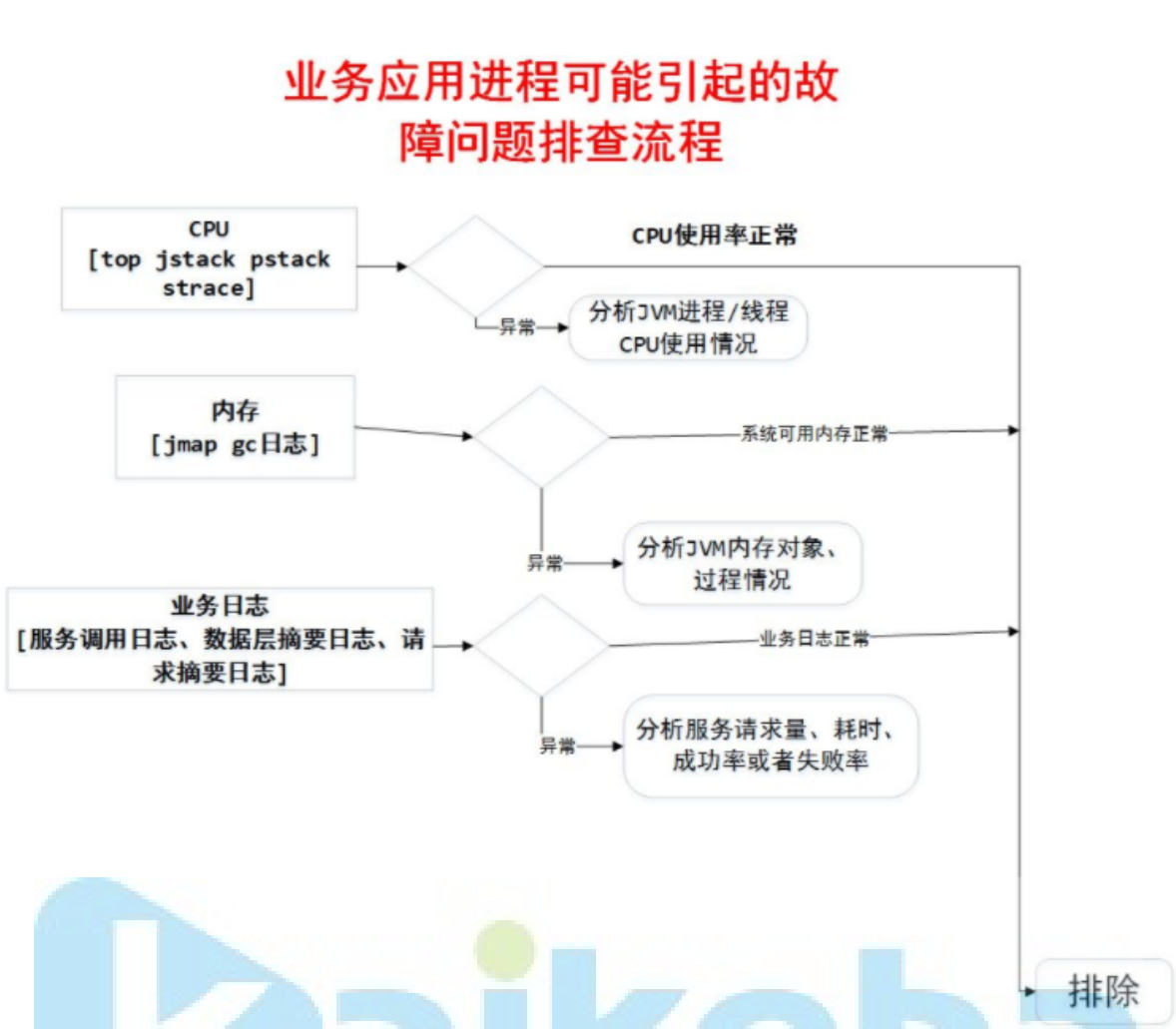
- 首先我们要排除其他进程 (除主进程之外) 可能引起的故障问题；
- 然后排除业务应用可能引起的故障问题；
- 可以考虑是否为运营商或者云服务提供商所引起的故障。

2.1、系统异常排查流程

问题定位流程，在linux系统中排查问题的方法，流程。



2.2、业务应用排查流程



压力测试&瓶颈分析&JVM优化实践

1、 Jvm调优原理

1.1、思考问题

Jvm为什么要调优？？？

1、垃圾太多，内存占满了，程序跑不动了！！！---- 内存溢出

2、垃圾回收线程太多(回收垃圾太频繁)，gc线程本身也会占有资源，gc线程太多，太频繁，消耗性能。

因此才需要对jvm去调优。

如果操作系统：32位，每一个进程最大使用内存空间是多少？

- 2~32 == 4g（理论上值），操作会占用一半资源。Jvm能使用的资源就是：2G
- 64位，每一个进程最大占用的内存空间是多少？？
- 2~64=16384PB，操作系统占用：2G ，jvm进程理论上可以使用无限的内存空间。

Jvm调优原则：

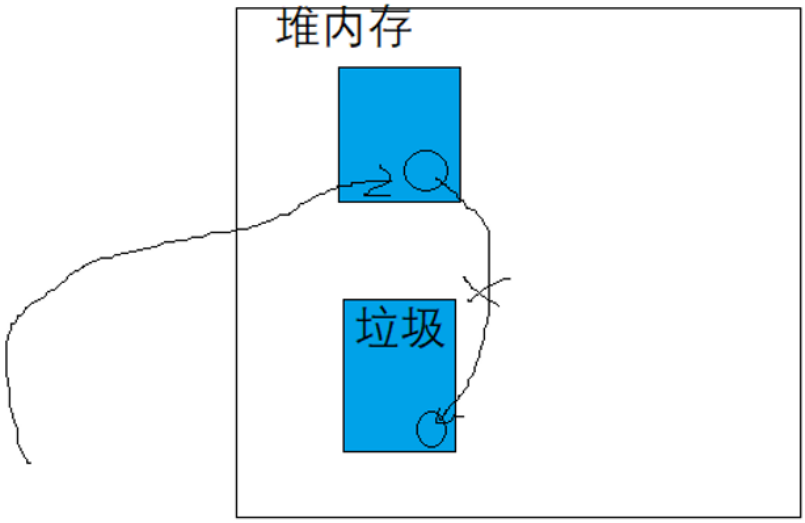
1、GC时间足够小 （堆内存设置小）

2、GC次数足够少 （堆内存设置大）

3、发生full GC的周期足够长

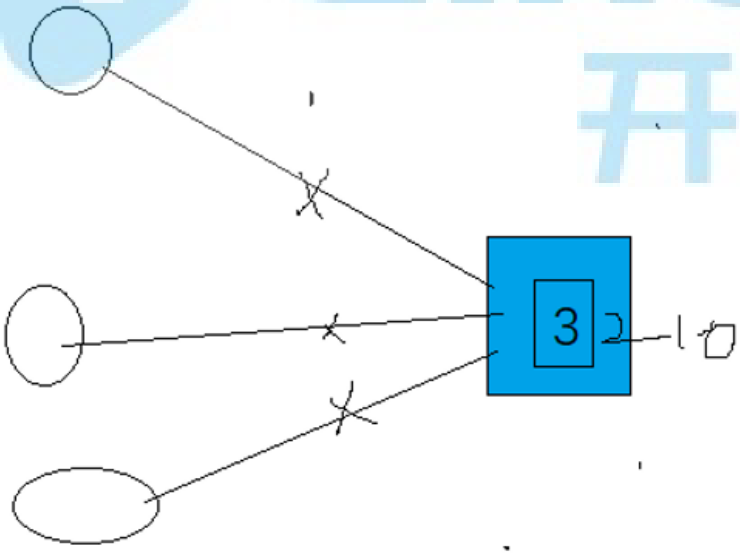
2.2、什么是垃圾?

概念： 没有被引用的对象就是垃圾。在堆内存中，一个对象引用没有了，这个空间就会被回收。

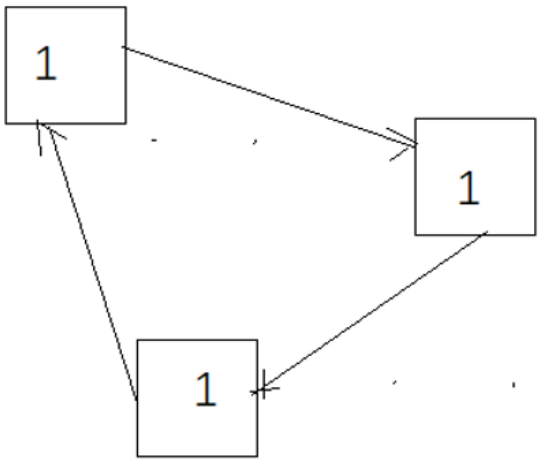


2.3、如何找垃圾?

1）引用计算



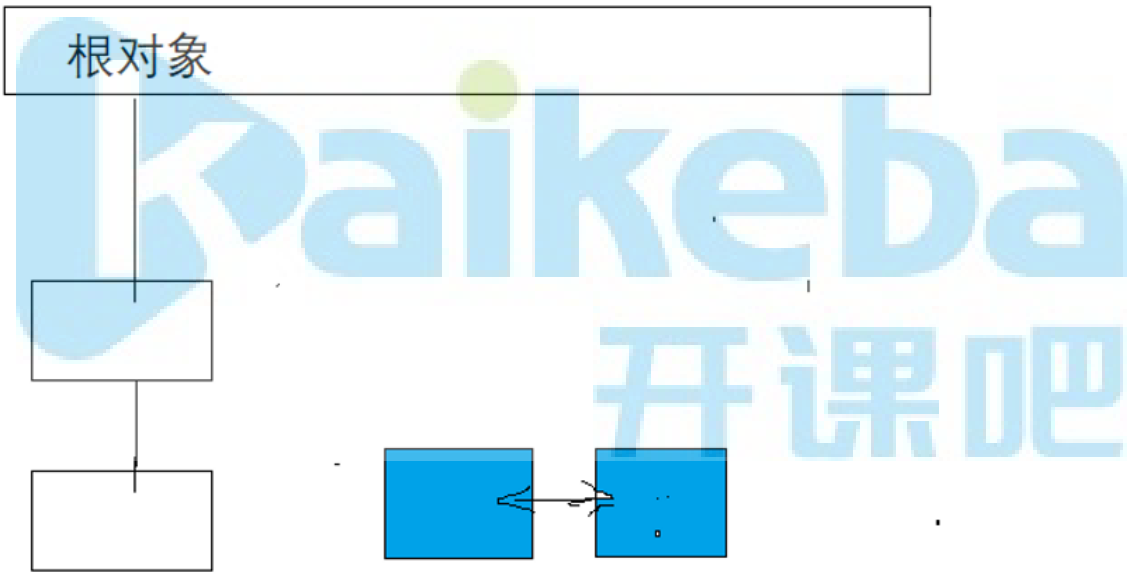
引用计算算法不能解决循环引用问题！！ 多个对象存在循环引用的关系，但是没有外部对象引用这几个循环引用对象，实际上这几个对象都是垃圾，因为没有外部对象引用，只是再相互引用。



2）根可达性算法

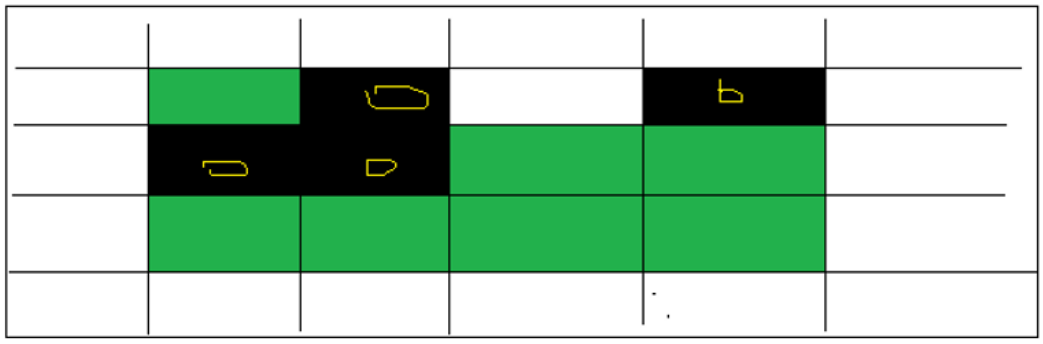
Hotspot 目前使用的主要的垃圾回收算法就是：根可达算法

Taobao.vm



2.4、垃圾回收算法

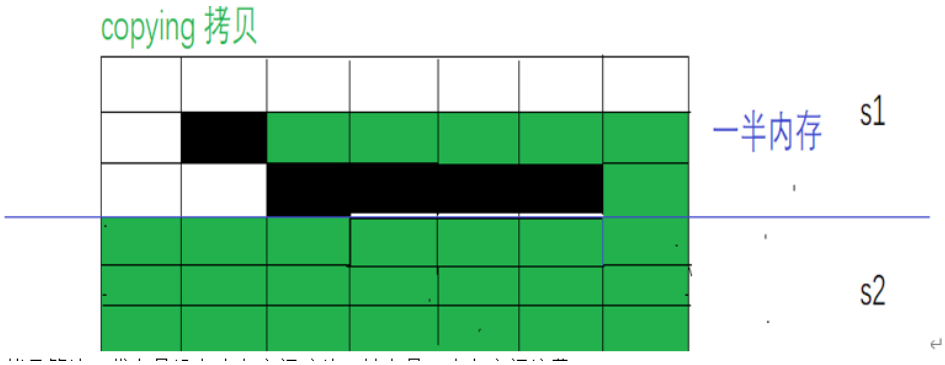
1）mark-sweep (标记清除)



找到垃圾，然后对垃圾直接进行标记，然后直接清除。但是清除垃圾后，内存空间会存在大量的内存碎片，降低后续处理性能。

终极解决方案：重启服务器

2）Copying (拷贝)



拷贝算法：优点是没有内存空间碎片，缺点是：内存空间浪费

3）mark-compact (标记压缩)

标记压缩算法：把存活对象拷贝到可回收的空间中（垃圾），然后把存活对象空间进行压缩，串联起来，防止空间碎片化。

把未使用对象空间链接起来，然后把使用空间队列链接起来。

优点： 没有空间碎片

缺点： 拷贝+压缩 消耗时间，性能比较低。

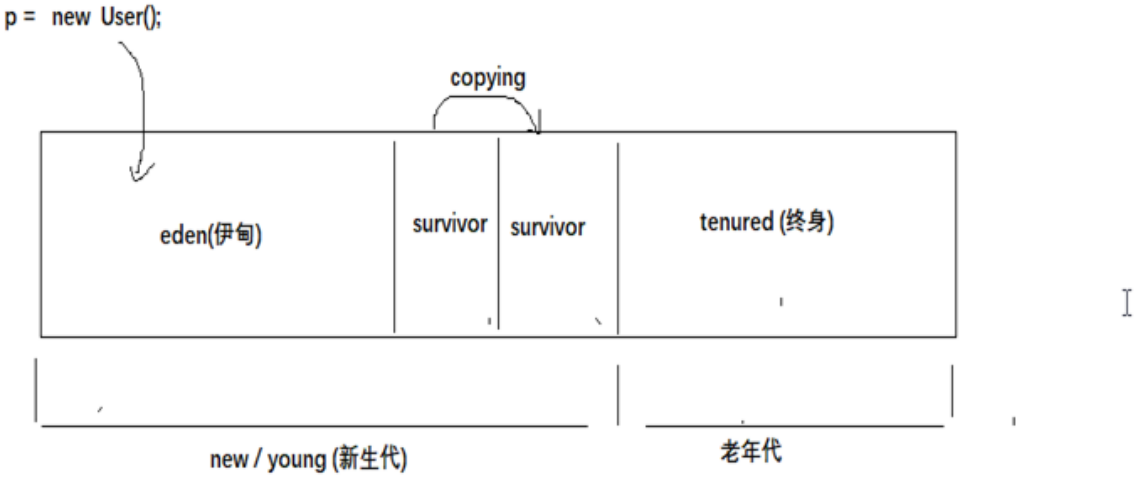
2.5、常用垃圾回收器

响应时间优先：parNew + CMS

内存较大组合： parallel Scavenge + Parallel Old

内存较小： serial + serial Old

2.6、分代模型



新生代：90%+对象第一次就被回收了。10% 对象没有被回收。
回收流程：经过第一次回收后，此对象没有被回收掉，此对象就进入survivor1区域，再次对此对象回收，还是没有回收掉，此时对象年龄变为2，进入suvivor2区域
object : 3 ---> survivor 1
object: 4 ----> survivor 2
.....
jdk1.8默认设置：对象年龄到15后，进入老年代区域。

问题：何时触发垃圾回收？？

有的垃圾回收器只有空间满了以后，才会进行垃圾回收。有的垃圾回收器空间占用到一定比例，就开始回收。

1、当年轻代存储满了以后，才开始垃圾回收

2、当老年代也存储满了，产生一次full gc

3、STW --- stop the world (单线程收集垃圾，多线程收集垃圾)

4、CMS --- 一边干活，一边收集垃圾