1. tomcat做过哪些优化

- 优化JVM，优化JVM的根本目的是尽可能的减少GC，不论是FULL GC还是MINOR GC，减少GC，特别是FULL GC的目的是尽量减少STW（Stop The World），SWT一旦发生，在客户端的影响就是，用户会发现卡顿
  - 优化堆内存
  - 优化年轻代和老年代的比例
  - 优化GC策略
- tomcat优化
  - 根据情况，高并发情况下，适当增加线程池数量
    - `<Connector port="8080" protocol="HTTP/1.1" maxThreads="200" />`
  - 使用NIO模型，提高并发能力，新版tomcat默认模型
    - `protocol="org.apache.coyote.http11.Http11NioProtocol"`
  - 启用长连接
    - `connectionTimeout="20000" keepAliveTimeout="60000"`
    - `maxKeepAliveRequests="100"`
  - 启用GZIP

```
<Connector port="8080" protocol="HTTP/1.1"
maxThreads="500"
acceptCount="200"
connectionTimeout="20000"
compression="on"
compressionMinSize="2048"
compressableMimeType="text/html,text/xml,text/plain,text/
css,application/json,application/javascript"
URIEncoding="UTF-8"
protocol="org.apache.coyote.http11.Http11NioProtocol"
redirectPort="8443" />
```

- 内核优化
  - 上调文件描述符数量

```
ulimit -n 65535
```

2. 什么是OOM，Java程序如何解决OOM问题

- OOM

OOM即Out Of Memory，"内存用完了"，的情况在java程序中比较常见。系统会选出一个进程将之杀死，在日志messages中看到类似下面的提示

```
# Jul 10 10:20:30 kernel: Out of memory: Kill process 9527(java) score 88 or sacrifice child
```

- 出现OOM的原因
  - `java.lang.OutOfMemoryError: Java heap space`: 堆内存不足，通常是内存泄漏或应用程序创建了太多对象
  - `java.lang.OutOfMemoryError: GC overhead limit exceeded`: GC运行频率过高且无法释放足够内存
  - `java.lang.OutOfMemoryError: Metaspace`：Metaspace空间不足，通常与类加载器泄漏或类的数量超出有关
  - `java.lang.OutOfMemoryError`: Unable to create native thread

- 解决方法
  - 检查JVM启动参数：
    - `java -Xms` 和 `java -Xmx`， 适当增加堆内存设置，例如 `-Xmx4g`
    - 调整eden区和幸存区的内存比例，增大青年代的堆内存大小
  - 缓存清理
  - 适当增加线程数
  - 更改垃圾回收器，比如高并发场景下，将CMS改为G1

3. 实验题：Tomcat基于MSM实现session共享

```
# 下载安装memcached
apt update; apt install memcached -y

# 查看memcached的状态
systemctl status memcached.service

# 启动脚本和配置文件
[root@ubuntu ~]# systemctl cat memcached.service | grep Exec
ExecStart=/usr/share/memcached/scripts/systemd-memcached-wrapper /etc/memcached.conf

# nginx节点测试 Memcached远程访问

# 安装python包管理器pypi
apt install -y python3-pip

# 安装python客户端工具
pip install python-memcached

# 写一个memcache测试脚本
```

```
vim test.py

#!/usr/bin/python3

import memcache
client = memcache.Client(['10.0.0.150:11211','10.0.0.151:11211'],debug=True)

for i in client.get_stats('items'):
    print(i)

print('-' * 35)

for i in client.get_stats('cachedump 5 0'):
    print(i)

# 执行python脚本
[root@mystical ~] $python3 test.py
('10.0.0.150:11211 (1)', {})
('10.0.0.151:11211 (1)', {})
-----------------------------------
('10.0.0.150:11211 (1)', {})
('10.0.0.151:11211 (1)', {})

# 显示无任何数据

# 配置（150的session写到151的memcached; 151的session写到150的memcached）
# tomcat节点配置session共享

# 10.0.0.150

# 下载相关jar包
cd /usr/local/tomcat/lib

wget https://repo1.maven.org/maven2/org/ow2/asm/asm/5.2/asm-5.2.jar
wget https://repo1.maven.org/maven2/com/esotericsoftware/kryo/3.0.3/kryo-
3.0.3.jar
wget https://repo1.maven.org/maven2/de/javakaffee/kryo-serializers/0.45/kryo-
serializers-0.45.jar
wget https://repo1.maven.org/maven2/de/javakaffee/msm/memcached-session-
manager/2.3.2/memcached-session-manager-2.3.2.jar
wget https://repo1.maven.org/maven2/de/javakaffee/msm/memcached-session-manager-
tc9/2.3.2/memcached-session-manager-tc9-2.3.2.jar
wget https://repo1.maven.org/maven2/com/esotericsoftware/minlog/1.3.1/minlog-
1.3.1.jar
wget https://repo1.maven.org/maven2/de/javakaffee/msm/msm-kryo-
serializer/2.3.2/msm-kryo-serializer-2.3.2.jar
wget https://repo1.maven.org/maven2/org/objenesis/objenesis/2.6/objenesis-2.6.jar
wget
https://repo1.maven.org/maven2/com/esotericsoftware/reflectasm/1.11.9/reflectasm-
1.11.9.jar
wget https://repo1.maven.org/maven2/net/spy/spymemcached/2.12.3/spymemcached-
2.12.3.jar

# 修改配置
cat /usr/local/tomcat/conf/context.xml
```

```
# 添加下列内容
# 10.0.0.150
<Manager className="de.javakaffee.web.msm.MemcachedBackupSessionManager"
memcachedNodes="m1:10.0.0.150:11211,m2:10.0.0.151:11211" failoverNodes="m1"
requestUriIgnorePattern=".*\.(ico|png|gif|jpg|css|js)$"
transcoderFactoryClass="de.javakaffee.web.msm.serializer.kryo.KryoTranscoderFacto
ry"/>

# 10.0.0.151
<Manager className="de.javakaffee.web.msm.MemcachedBackupSessionManager"
memcachedNodes="m1:10.0.0.150:11211,m2:10.0.0.151:11211" failoverNodes="m2"
requestUriIgnorePattern=".*\.(ico|png|gif|jpg|css|js)$"
transcoderFactoryClass="de.javakaffee.web.msm.serializer.kryo.KryoTranscoderFacto
ry"/>

# 150和151都重启tomcat服务
systemctl restart tomcat
```

4. 基于客户端的分布式机集群和基于服务端的分布式集群有什么区别

客户端的分布式集群

客户端的分布式集群是指客户端负责**将请求分发到不同的服务器节点**，并自行**管理分布式逻辑**，而**服务器是无状态的**，客户端决定要与哪个服务器进行交互。

Memcached是客户端分布式集群

1. 无中心协调：Memcached服务器本身不维护关于其他服务器状态或数据存放位置的信息，它们独立运行，不需要彼此通信来处理缓存数据
2. 客户端决策：数据在哪里存储（哪台Memcached服务器）由客户端决定。客户端使用一致性hash等算法选择数据应该存储在哪个服务器上，这种方法减少了单点故障的风险，因为没有一个中央节点是关键的
3. 简单性和性能：这种模式使Memcached非常简单和高效。服务器仅仅响应客户端的请求，所有智能决策都由客户端完成。这减轻了服务器的计算负担，使系统能更快地响应。

服务端的分布式集群

服务端的分布式集群是指由**服务器来统一管理请求的分发和任务的调度**，客户端**只和一个负载均衡器或网关**进行通信，后端的分发逻辑和集群拓扑结构对客户端**透明**。

Web应用：前端请求通过 Nginx（或负载均衡）转发到后端的多个应用实例

5. 使用**jvisualvm**观察下面的Java程序在堆中的eden区，幸存区和老年区之间GC的过程，并截图

```
// HeapOom.java
import java.util.ArrayList;
import java.util.List;
public class HeapOom {
```

```
    public static void main(String[] args) {
        List<byte[]> list = new ArrayList<byte[]>();
        int i = 0;
        boolean flag = true;
        while(flag) {
            try {
                i++;
                list.add(new byte[1024 * 1024]); // 每次增加一个1M大小的数组对象
                Thread.sleep(1000);
            } catch (Throwable e ) {
                e.printStackTrace();
                flag = false;
                System.out.println("count="+i); // 记录运行次数
            }
        }
    }
}

// 使用如下指令编译代码执行
// javac HeapOom.java
// java HeapOom
```

使用jvisualvm监控内存

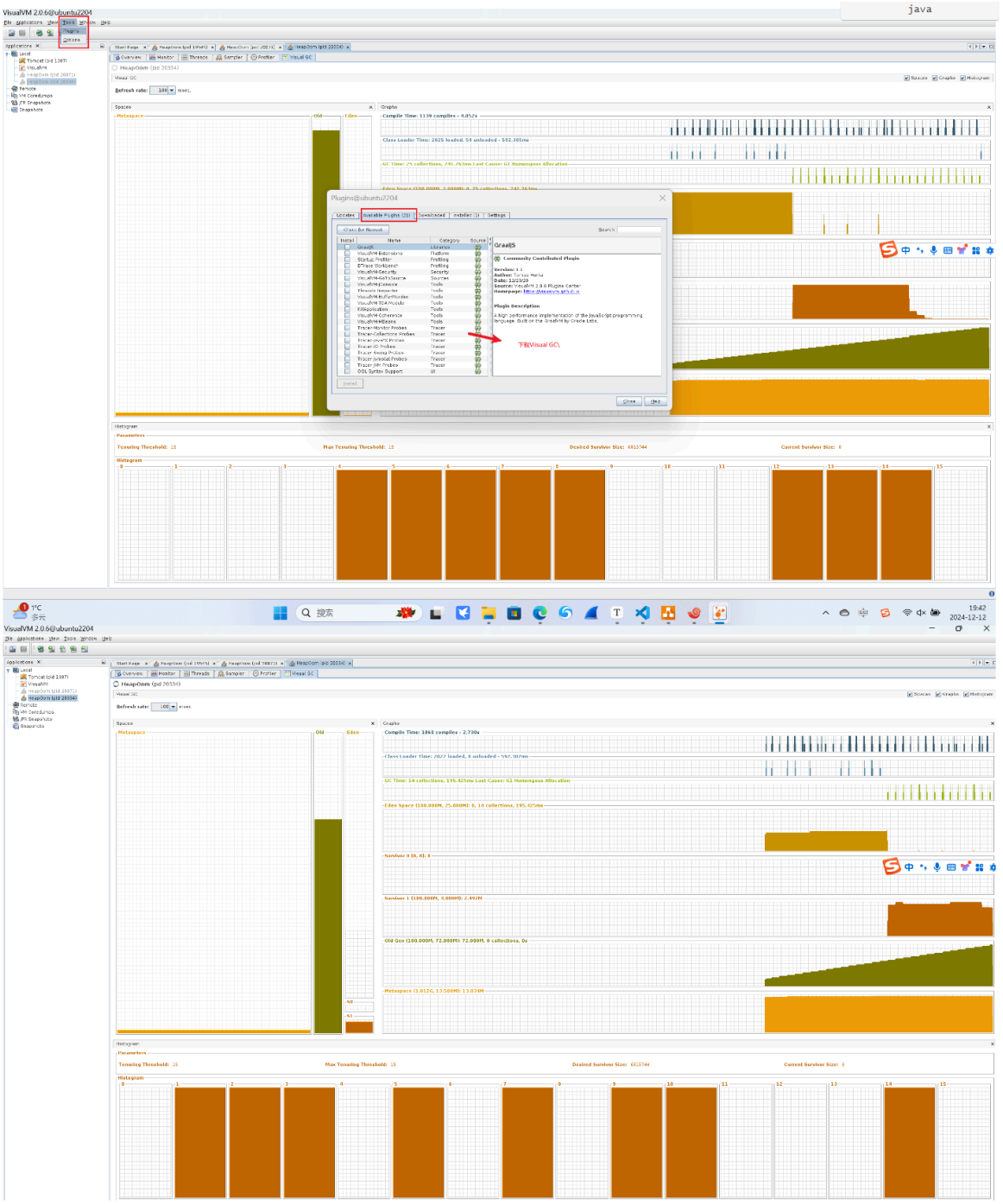
jvisualvm一款图形化的内存监控工具，在jdk-8u361之前的版本中是内置的组件，但在之后的JDK版本中已经取消了该组件，要单独下载并配置

```
# 安装依赖
apt install libxrender1 libxrender1 libxtst6 libxi6 fontconfig -y

wget https://github.com/oracle/visualvm/releases/download/2.1.8/visualvm_218.zip

unzip visualvm_218.zip

# 在windows中开启Xmanager - Passive
export DISPLAY=10.0.0.1:0.0

# 执行，在windows中能看到GUI界面，在GUI中点击Tools菜单，Plugins，然后安装VisualGC插件
```

测试程序

```
// HeapOom.java
import java.util.ArrayList;
import java.util.List;
public class HeapOom {
    public static void main(String[] args) {
        List<byte[]> list = new ArrayList<byte[]>();
        int i = 0;
        boolean flag = true;
        while(flag) {
            try {
```

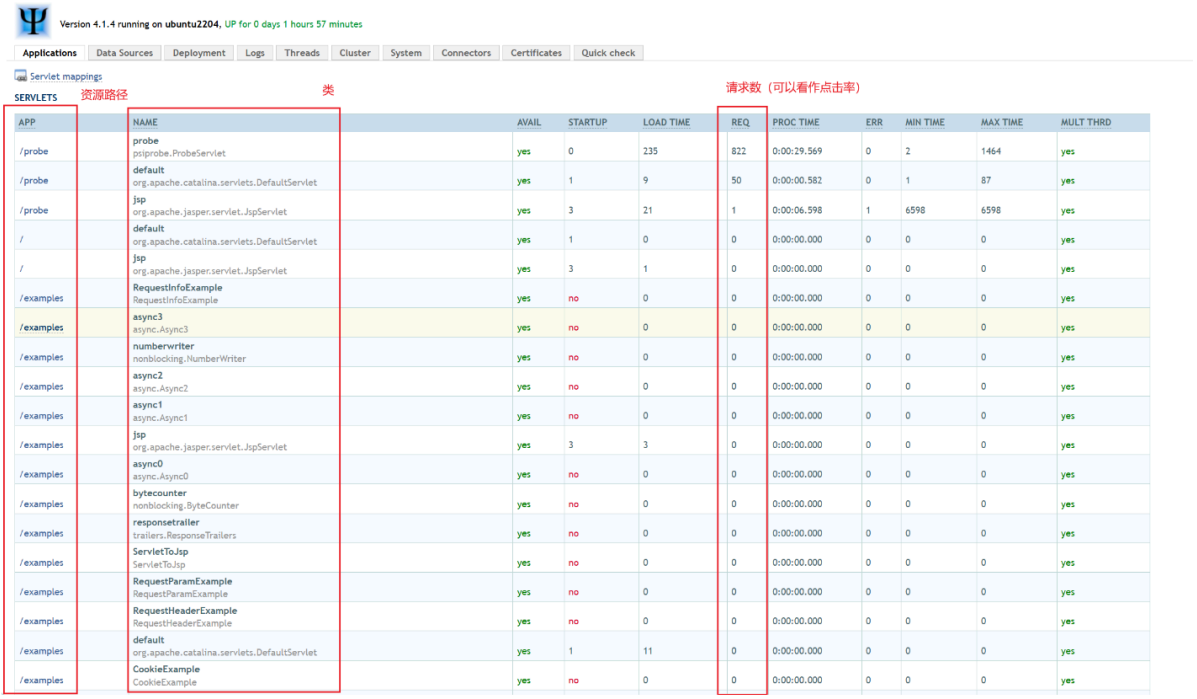
```
                i++;
                list.add(new byte[1024 * 1024]); // 每次增加一个1M大小的数组对象
                Thread.sleep(1000);
            } catch (Throwable e ) {
                e.printStackTrace();
                flag = false;
                System.out.println("count="+i); // 记录运行次数
            }
        }
    }
}
```



6. 你在工作中监控过Java程序的哪些指标

查看资源请求情况详情

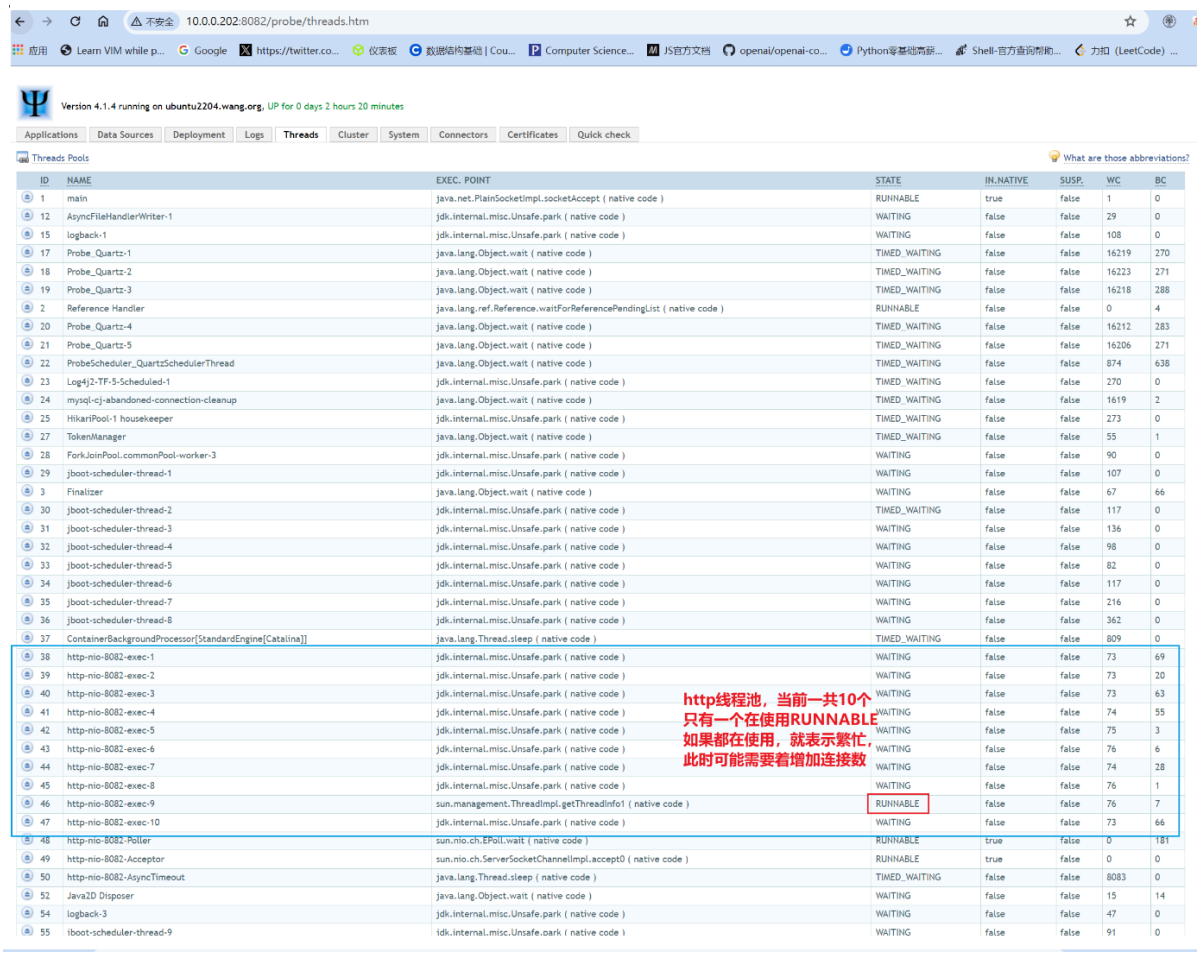
通过request查看



Session

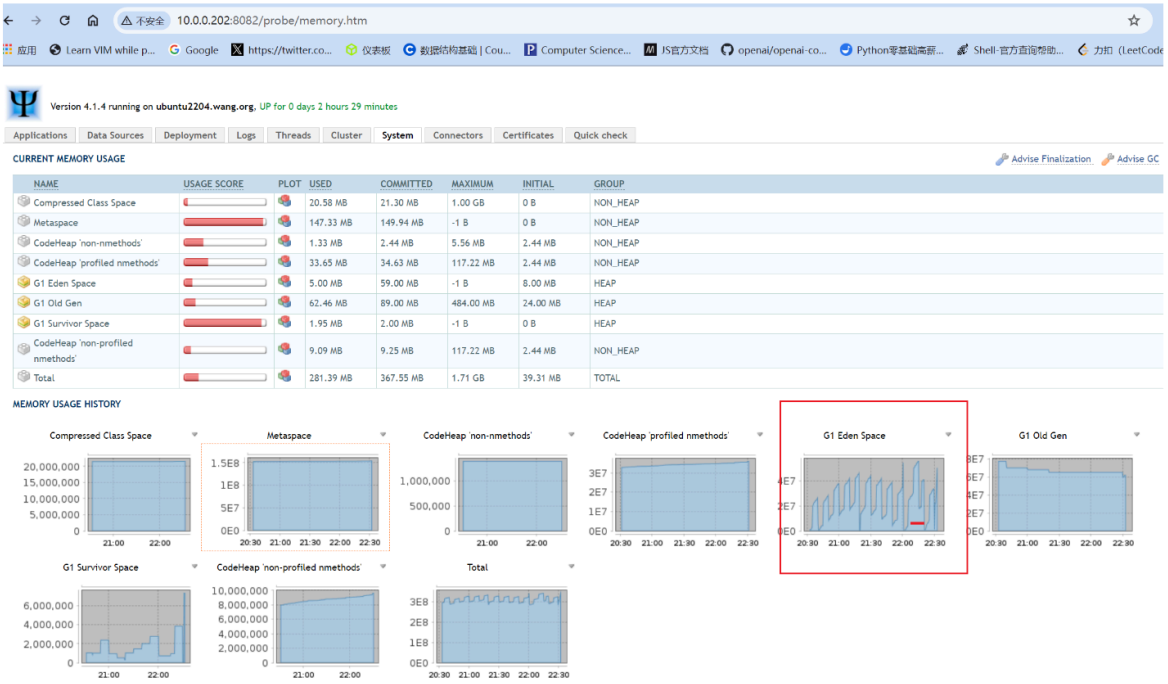
线程池数据（连接数）

- 根据下列指标即可看出线程池是否够用，是否需要增加线程池的线程数量



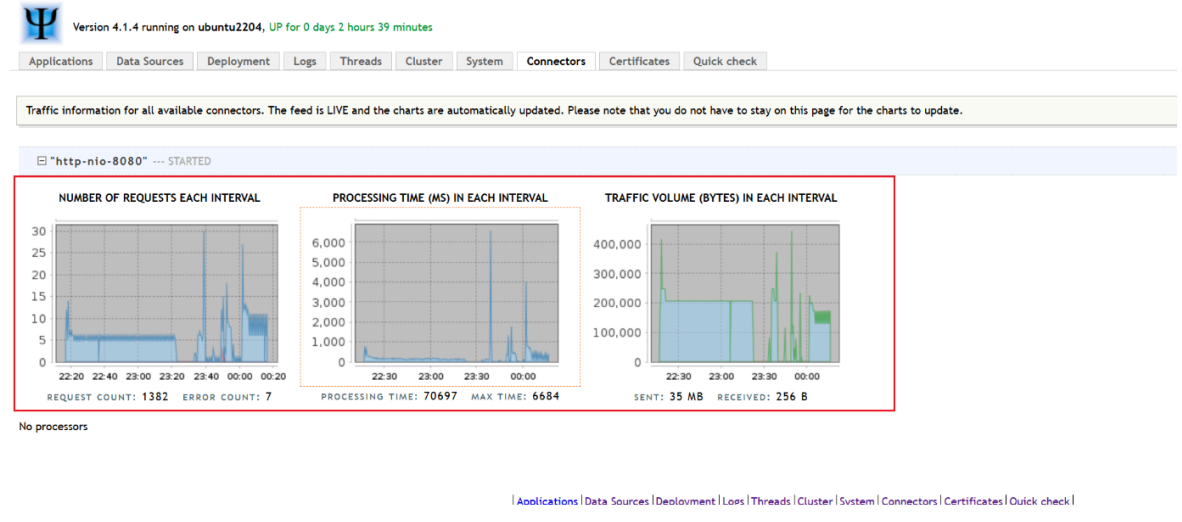
ID	NAME	EXEC. POINT	STATE	IN.NATIVE	SUSP.	WC	BC
1	main	java.net.PlainSocketImpl.socketAccept (native code)	RUNNABLE	true	false	1	0
12	AsyncFileHandlerWriter-1	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	29	0
15	logback-1	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	108	0
17	Probe_Quartz-1	java.lang.Object.wait (native code)	TIMED_WAITING	false	false	16219	270
18	Probe_Quartz-2	java.lang.Object.wait (native code)	TIMED_WAITING	false	false	16223	271
19	Probe_Quartz-3	java.lang.Object.wait (native code)	TIMED_WAITING	false	false	16218	288
2	Reference Handler	java.lang.ref.Reference.waitForReferencePendingList (native code)	RUNNABLE	false	false	0	4
20	Probe_Quartz-4	java.lang.Object.wait (native code)	TIMED_WAITING	false	false	16212	283
21	Probe_Quartz-5	java.lang.Object.wait (native code)	TIMED_WAITING	false	false	16206	271
22	ProbeScheduler_QuartzSchedulerThread	java.lang.Object.wait (native code)	TIMED_WAITING	false	false	874	638
23	Log4j2-TF-5-Scheduled-1	jdk.internal.misc.Unsafe.park (native code)	TIMED_WAITING	false	false	270	0
24	mysql-cj-abandoned-connection-cleanup	java.lang.Object.wait (native code)	TIMED_WAITING	false	false	1619	2
25	HikariPool-1 housekeeper	jdk.internal.misc.Unsafe.park (native code)	TIMED_WAITING	false	false	273	0
27	TokenManager	java.lang.Object.wait (native code)	TIMED_WAITING	false	false	55	1
28	ForkJoinPool.commonPool-worker-3	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	90	0
29	jboot-scheduler-thread-1	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	107	0
3	Finalizer	java.lang.Object.wait (native code)	WAITING	false	false	67	66
30	jboot-scheduler-thread-2	jdk.internal.misc.Unsafe.park (native code)	TIMED_WAITING	false	false	117	0
31	jboot-scheduler-thread-3	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	136	0
32	jboot-scheduler-thread-4	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	98	0
33	jboot-scheduler-thread-5	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	82	0
34	jboot-scheduler-thread-6	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	117	0
35	jboot-scheduler-thread-7	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	216	0
36	jboot-scheduler-thread-8	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	362	0
37	ContainerBackgroundProcessor[StandardEngine[Catalina]]	java.lang.Thread.sleep (native code)	TIMED_WAITING	false	false	809	0
38	http-nio-8082-exec-1	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	73	69
39	http-nio-8082-exec-2	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	73	20
40	http-nio-8082-exec-3	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	73	63
41	http-nio-8082-exec-4	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	74	55
42	http-nio-8082-exec-5	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	75	3
43	http-nio-8082-exec-6	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	76	6
44	http-nio-8082-exec-7	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	74	28
45	http-nio-8082-exec-8	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	76	1
46	http-nio-8082-exec-9	sun.management.ThreadImpl.getThreadInfo1 (native code)	RUNNABLE	false	false	76	7
47	http-nio-8082-exec-10	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	73	66
48	http-nio-8082-Poller	sun.nio.ch.EPoll.wait (native code)	RUNNABLE	true	false	0	181
49	http-nio-8082-Acceptor	sun.nio.ch.ServerSocketChannelImpl.accept0 (native code)	RUNNABLE	true	false	0	0
50	http-nio-8082-AsyncTimeout	java.lang.Thread.sleep (native code)	TIMED_WAITING	false	false	8083	0
52	Java2D Disposer	java.lang.Object.wait (native code)	WAITING	false	false	15	14
54	logback-3	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	47	0
55	jboot-scheduler-thread-9	jdk.internal.misc.Unsafe.park (native code)	WAITING	false	false	91	0

内存数据



连接器Connector

- 观察（http）连接器里的点击率和吞吐量



7. java对象进入老年代的原因（最少说3种）

三种对象从青年代移动到老年代的原因

- **对象的年龄超过阈值（默认15，具体看JVM实现）**
  - 当对象在 Survivor 区的年龄超过了阈值，JVM 会将对象移动到老年代
  - 过程解释

        - 在年轻代中，内存划分为 Eden、Survivor from (S0) 和 Survivor to (S1) 三部分。
        - 新创建的对象首先分配在 Eden 区
        - 当 Eden 区内存满时，Minor GC 触发
        - 存活的对象将被移入 Survivor S0 区。
        - 在接下来的 GC 中，存活的对象会从 S0 → S1，它们的"年龄"会增加 1。
        - 当对象的年龄达到某个阈值 (默认为15)，JVM 就会将该对象移入老年代。
    - **年龄阈值**
        - JVM 中的对象年龄由 MaxTenuringThreshold 决定，默认值是15。
        - 也就是说，如果一个对象的年龄达到 15（在 S0/S1 之间存活了 15 次 GC），这个对象就会被移到老年代
    - **修改阈值**

```
-XX:MaxTenuringThreshold=10
# 将对象的"最大生存代数"调整为 10。也就是说，当对象在 Survivor 区经历了 10 次 GC 后，它就会被移入老年代。
```

- 对象的年龄超过阈值
    - 大对象（通常是大数组或大字符串）会直接分配到老年代，而不会经历年轻代的过程。
    - **为什么要直接进入老年代?**
        - 如果一个大对象（比如一个大数组或大字符串）被分配到 Eden 区，可能会很快填满 Eden 区，频繁触发 GC。这种频繁的 GC 会导致性能问题。为了解决这个问题，JVM 提供了一种机制，大对象直接进入老年代。
    - **如何定义大对象的大小?**
        - JVM 使用 `PretenureSizeThreshold` 参数来决定大对象的最小大小。
        - 只要对象的大小大于 PretenureSizeThreshold，它就会被直接分配到老年代。
    - **如何设置大对象的阈值?**

```
# 表示大于 1MB 的对象将直接分配到老年代。
-XX:PretenureSizeThreshold=1M
```

- 特殊情况：**Survivor 区满了的情况**
    - 如果 Survivor 区满了，存活的对象也会被直接送入老年代。
    - **为什么会发生?**
        - 在 GC 过程中，如果 Survivor 区中的空间不足以存放所有 Eden 和 Survivor 中存活的对象，那么一部分对象会直接被分配到老年代。
        - 这有点像应急转移的机制。
    - 这种情况在高并发场景中常见，比如网络高并发请求，瞬时大量对象创建导致 Survivor 区不够用。

堆满时，老年代如何分配?

- 当老年代也满了时，JVM 将触发 Full GC。
- Full GC 触发后，如果无法回收老年代中的对象，JVM 会抛出：

```
java.lang.OutOfMemoryError: Java heap space
```

8. 创建Nexus服务，并配置Maven、apt、yum的私有仓，并在客户端测试，看是否生效

Nexus安装和配置Maven仓库

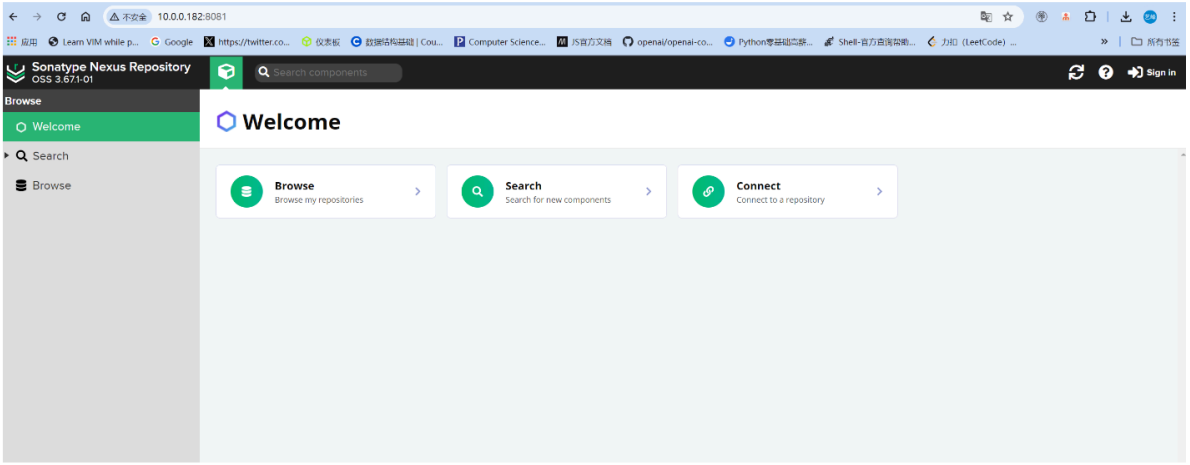
实验准备

- 设备1
  - IP：10.0.0.150
  - 部署服务：java, Maven
  - 备注：将Maven中的仓库配置指向自建的私有仓库
- 设备2
  - IP：10.0.0.182（重点）
  - 部署服务：Nexus Server
  - 内存需要4G以上，实现Maven的仓库，apt仓库，yum仓库

安装Nexus，并配置Maven仓库

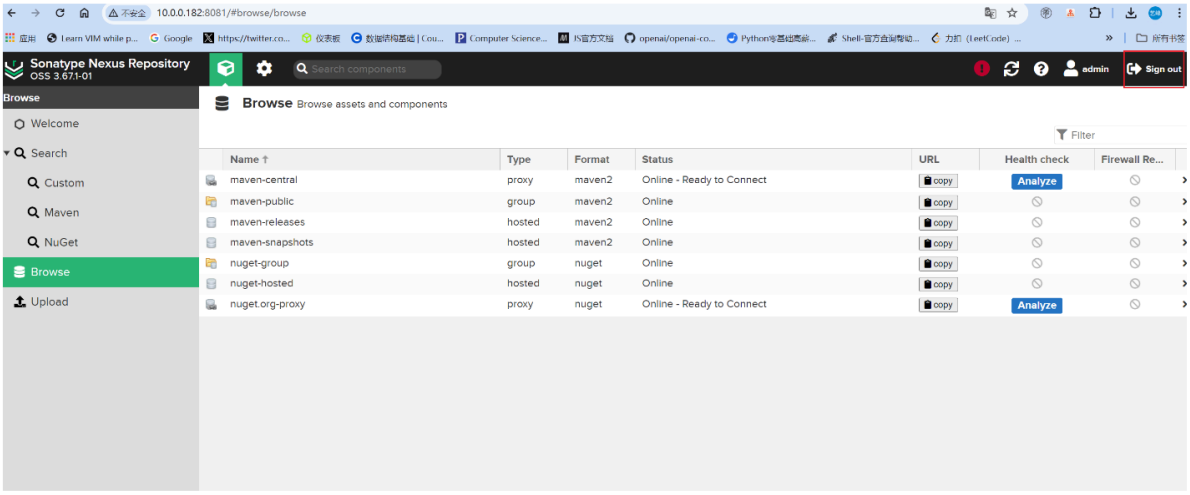
```
# 安装jdk-11
dpkg -i jdk-11.0.23_linux-x64_bin.deb

wget https://download.sonatype.com/nexus/3/nexus-3.67.1-01-java11-unix.tar.gz

tar xf nexus-3.67.1-01-java11-unix.tar.gz -C /usr/local/

ln -sv /usr/local/nexus-3.67.1-01/ /usr/local/nexus

# 主配置文件，指定监听的端口号和IP
## DO NOT EDIT - CUSTOMIZATIONS BELONG IN $data-dir/etc/nexus.properties
##
# Jetty section
application-port=8081
application-host=0.0.0.0
nexus-args=${jetty.etc}/jetty.xml,${jetty.etc}/jetty-http.xml,${jetty.etc}/jetty-
requestlog.xml
nexus-context-path=/

# Nexus section
nexus-edition=nexus-pro-edition
nexus-features=\
 nexus-pro-feature

nexus.hazelcast.discovery.isEnabled=true

# 主要文件
[root@mystical /usr/local/nexus] $tree  bin
bin
├── contrib
|   ├── karaf-service.sh
```

```
|   ├── karaf-service-template.conf
|   ├── karaf-service-template.init
|   ├── karaf-service-template.init-debian
|   ├── karaf-service-template.init-redhat
|   ├── karaf-service-template.solaris-smf
|   ├── karaf-service-template.systemd
|   ├── karaf-service-template.systemd-instances
|   ├── karaf-service-win.exe
|   └── karaf-service-win.xml
├── nexus                  # 可执行二进制脚本文件，主程序
├── nexus.rc               # 配置运行时用户身份
└── nexus.vmoptions        # 配置服务启动时的Java选项

# 可以使用nexus.rc 指定执行程序的身份
run_as_user="root"

# 将二进制执行文件加入全局变量
ln -sv /usr/local/nexus/bin/nexus /usr/bin/

# 启用nexus

# 在浏览器访问10.0.0.182:8081
```




```
# 配置服务脚本，使用systemctl 来进行服务管理
# 参考：https://help.sonatype.com/en/installation-and-upgrades.html

cat lib/systemd/system/nexus.service
[Unit]
Description=nexus service
After=network.target1

[Service]
Type=forking
LimitNOFILE=65536
ExecStart=/usr/local/nexus/bin/nexus start
ExecStop=/usr/local/nexus/bin/nexus stop
User=root
#User=nexusvim v
Restart=on-abort

[Install]
```

```
WantedBy=multi-user.target
```



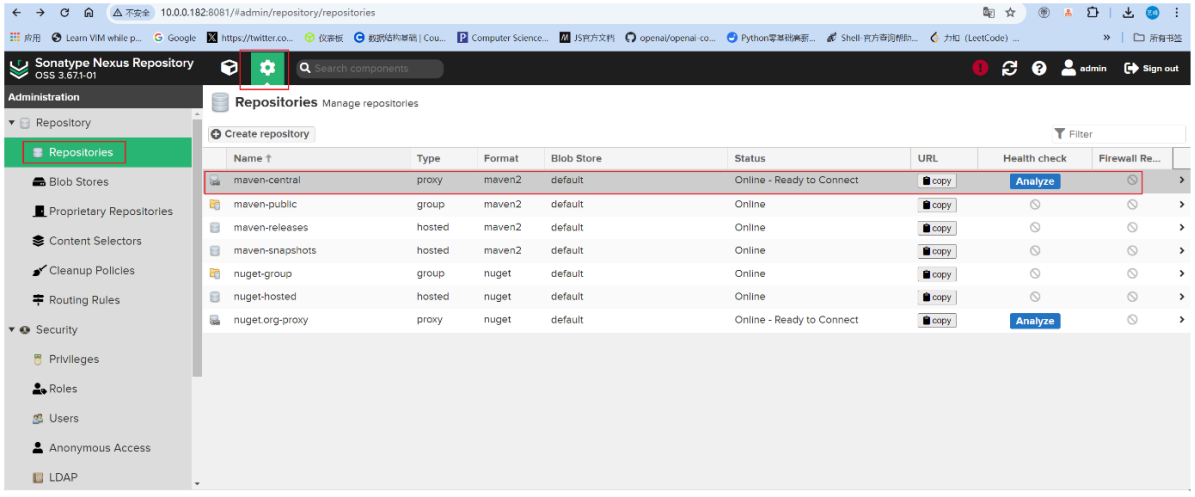
首次登录，用户名为admin; 密码在/usr/local/sonatype-work/nexus3/admin.password文件中

首次登录成功后，可以修改新的密码

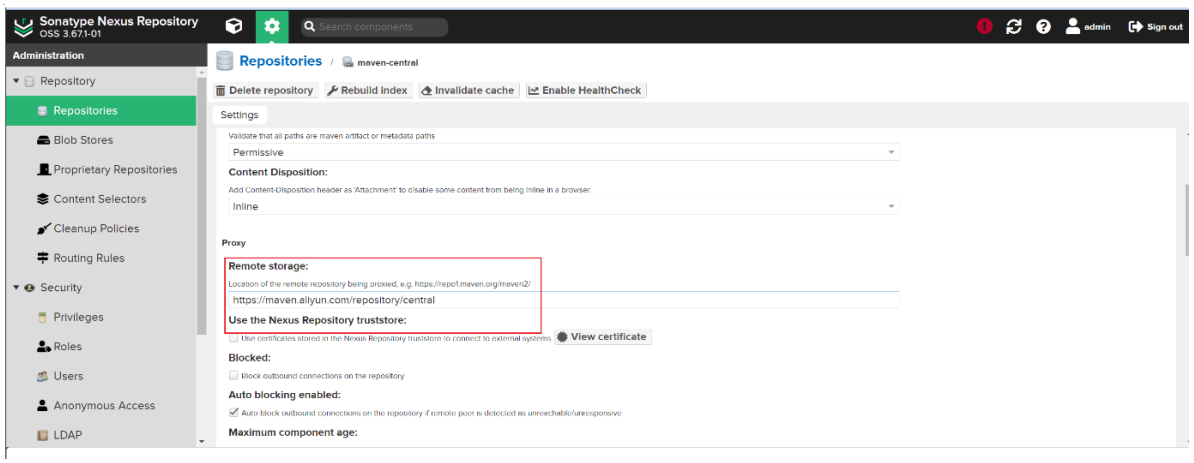
```
# 首次登录成功后修改密码，并选择支持匿名访问，做成公开库

# proxy    代理仓，如果nexus服务器上没有，则先去maven的官方仓库下载回来
# hosted   本地仓，如果nexus服务器上没有，不会去公网下载
# group    多个仓库集合，及支持maven仓库，yum和apt不能合并
#          https://maven.aliyun.com 也有集合
```

```
# 配置maven-central的公网地址以为国内阿里云
https://maven.aliyun.com/repository/central
```

- 点击



- 添加阿里的镜像仓库



在Maven主机上测试自建仓库(10.0.0.150)

```
cat /usr/local/maven/conf/setting.xml

<mirror>
    <id>nexus-maven</id>
    <mirrorOf>*</mirrorOf>
    <name>自建仓库</name>
    <url>http://10.0.0.182:8081/repository/maven-central/</url>
</mirror>
```

```
#在nexus 服务器上，对应的包文件放在此目录中
[root@ubuntu ~]# du -sh /usr/local/sonatype-work/nexus3/blobs/default/
 56M /usr/local/sonatype-work/nexus3/blobs/default/
 #但并不是可读的友好格式，这是基于安全方面的考虑，保有在 web 页面可见
[root@ubuntu ~]# tree /usr/local/sonatype-work/nexus3/blobs/default/
```

Nexus配置Apt仓库

1. 创建目录/data/blobs
2. 进入nexus配置界面，在Blob Stores页面点击Create Blob Store，创建一个存储，Type选择File，Name填Ubuntu2204, Path填/data/blobs/ubuntu2204，点击保存
3. 在Repositories页面点击Create repository,选择apt(proxy)，在新页面中Name填Ubutun2204, Distributionj填jammy,Remote storage 填https://mirror.aliyun.com/ubuntu/, Blob store 选择ubuntu2204, 点击保存
4. 在Browse页面选择刚创建的ubuntu2204，点击copy，复制原地址，然后修改一台ubuntu主机的/etc/apt/sources.list,将源指向自建的nextus服务器中的原地址
5. 测试

Nexus配置yum仓库

同上