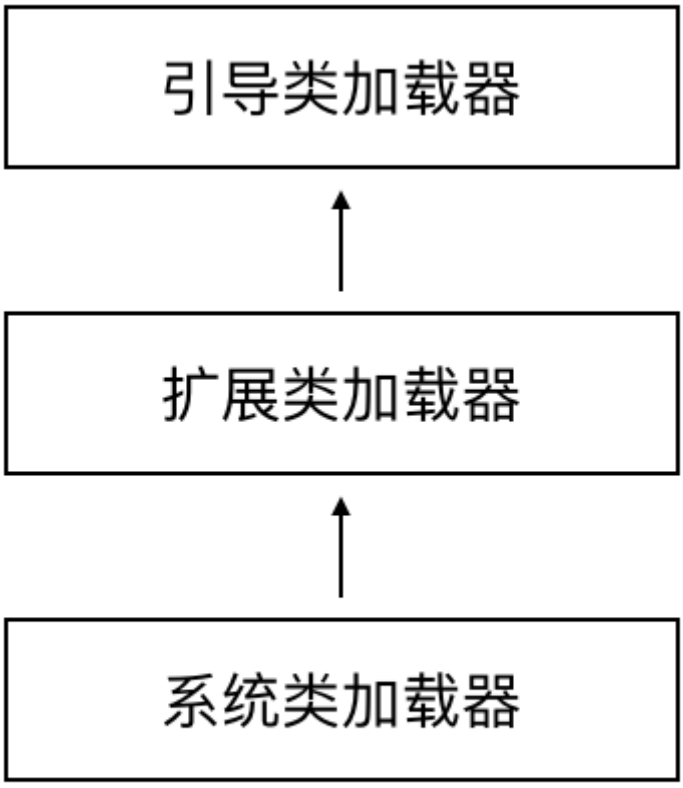
tomcat类加载机制剖析、https支持、tomcat调优

第一部分：tomcat类加载机制剖析

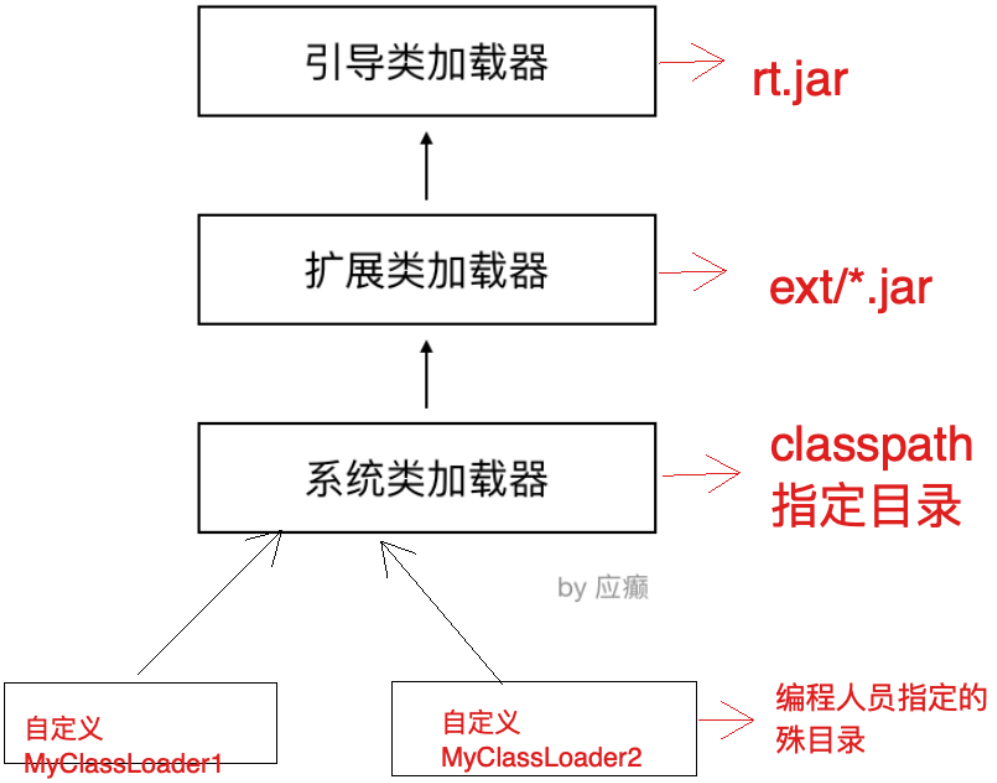
Java类（.java）—> 字节码文件(.class) —> 字节码文件需要被加载到jvm内存当中（这个过程就是一个类加载的过程）
类加载器（ClassLoader，说白了也是一个类，jvm启动的时候先把类加载器读取到内存当中去，其他的类（比如各种jar中的字节码文件，自己开发的代码编译之后的.class文件等等））
要说 Tomcat 的类加载机制，首先需要来看看 Jvm 的类加载机制，因为 Tomcat 类加载机制是在 Jvm 类加载机制基础之上进行了一些变动。

第 1 节 JVM 的类加载机制

JVM 的类加载机制中有一个非常重要的角色叫做类加载器（ClassLoader），类加载器有自己的体系，Jvm内置了几种类加载器，包括：引导类加载器、扩展类加载器、系统类加载器，他们之间形成父子关系，通过 Parent 属性来定义这种关系，最终可以形成树形结构。



by 应癫



类加载器	作用
引导启动类加载器 BootstrapClassLoader	c++编写，加载java核心库 java.*,比如rt.jar中的类，构造ExtClassLoader和AppClassLoader
扩展类加载器 ExtClassLoader	java编写，加载扩展库 JAVA_HOME/lib/ext目录下的jar中的类，如classpath中的jre ， javax.*或者java.ext.dir指定位置中的类
系统类加载器 SystemClassLoader/AppClassLoader	默认的类加载器，搜索环境变量 classpath 中指明的路径

另外：用户可以自定义类加载器（Java编写，用户自定义的类加载器，可加载指定路径的 class 文件）

当 JVM 运行过程中，用户自定义了类加载器去加载某些类时，会按照下面的步骤（父类委托机制）

1） 用户自己的类加载器，把加载请求传给父加载器，父加载器再传给其父加载器，一直到加载器树的顶层

2 ）最顶层的类加载器首先针对其特定的位置加载，如果加载不到就转交给子类（意思是先去引导类加载器查找，没有再去扩展类加载器找，以此类推，这就是**双亲委派机制**）

3 ）如果一直到底层的类加载都没有加载到，那么就会抛出异常 ClassNotFoundException

因此，按照这个过程可以想到，如果同样在 classpath 指定的目录中和自己工作目录中存放相同的 class，会优先加载 classpath 目录中的文件

第 2 节 双亲委派机制

2.1 什么是双亲委派机制

先加载上级类加载器，递归这个操作，如果上级的类加载器没有加载，自己才会去加载这个类。

2.2 双亲委派机制的作用

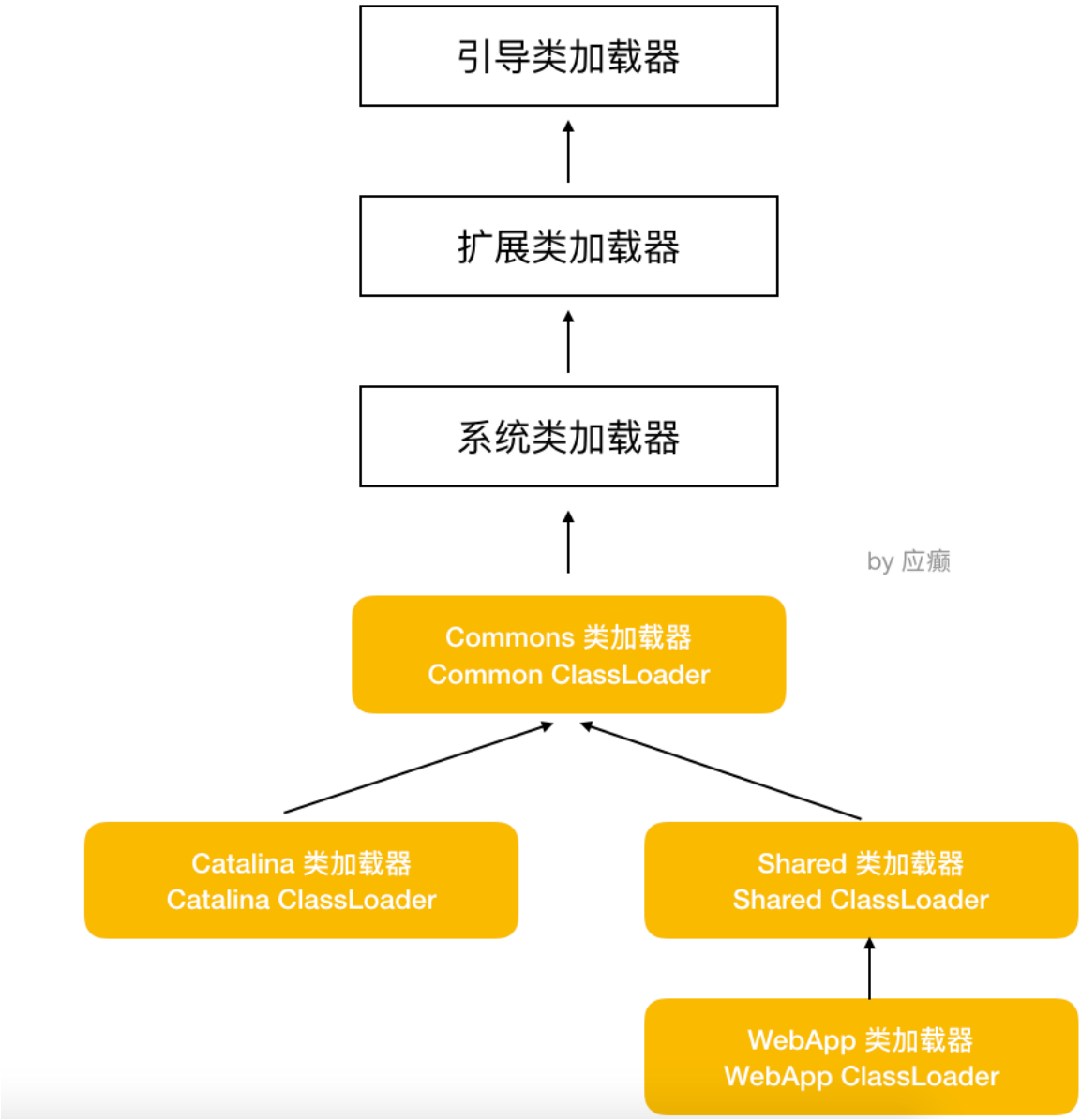
- **防止重复加载同一个.class。**通过委托去向上面问一问，加载过了，就不用再加载一遍。保证数据安全。
- **保证核心.class不能被篡改。**（如果先加载自己定义的，会出现问题的，那么真正的Object类就可能被篡改了）。

第 3 节 tomcat 的类加载机制

Tomcat 的类加载机制相对于 Jvm 的类加载机制做了一些改变。
但是没有严格的遵从双亲委派机制，也可以说打破了双亲委派机制

比如：有一个tomcat，webapps下部署了两个应用
app1/lib/a-1.0.jar com.lagou.edu.Abc
app2/lib/a-2.0.jar com.lagou.edu.Abc

不同版本中Abc类的内容是不同的，代码是不一样的。



- 系统类加载器正常情况下加载的是 CLASSPATH 下的类，但是 Tomcat 的启动脚本并未使用该变量，而是加载tomcat启动的类，比如bootstrap.jar，**通常在catalina.bat或者catalina.sh中指定。位于CATALINA_HOME/bin下。(载CATALINA_HOME/bin下的jar包)**
- Common 通用类加载器加载Tomcat使用以及应用通用的一些类，位于CATALINA_HOME/lib下，比如servlet-api.jar。**(加载CATALINA_HOME/lib下的jar包)**
- Catalina ClassLoader 用于加载服务器内部可见类，这些类应用程序不能访问 **(加载Tomcat自带jar包)**
- Shared ClassLoader 用于加载应用程序共享类，这些类服务器不会依赖。 **(加载部署的几个应用共享的jar包)**
- Webapp ClassLoader，每个应用程序都会有一个独一无二的Webapp ClassLoader，他用来加载本应用程序 /WEB-INF/classes 和 /WEB-INF/lib 下的类。 **(加载应用内部独有的jar包)**

tomcat 8.5 默认改变了严格的双亲委派机制

- 首先从 Bootstrap Classloader加载指定的类（引导类加载器）

- 如果未加载到，则从 /WEB-INF/classes加载（WebApp类加载器）
- 如果未加载到，则从 /WEB-INF/lib/*.jar 加载（WebApp类加载器）
- 如果未加载到，则依次从 System、Common、Shared 加载（在这最后一步，遵从双亲委派机制）

总结：等于说tomcat执行顺序和JVM不同的是使用**引导类加载器后使用的是WebApp类加载器**，其他都和JVM一样。

第二部分：tomcat对Https支持

第一节：HTTPS 简介

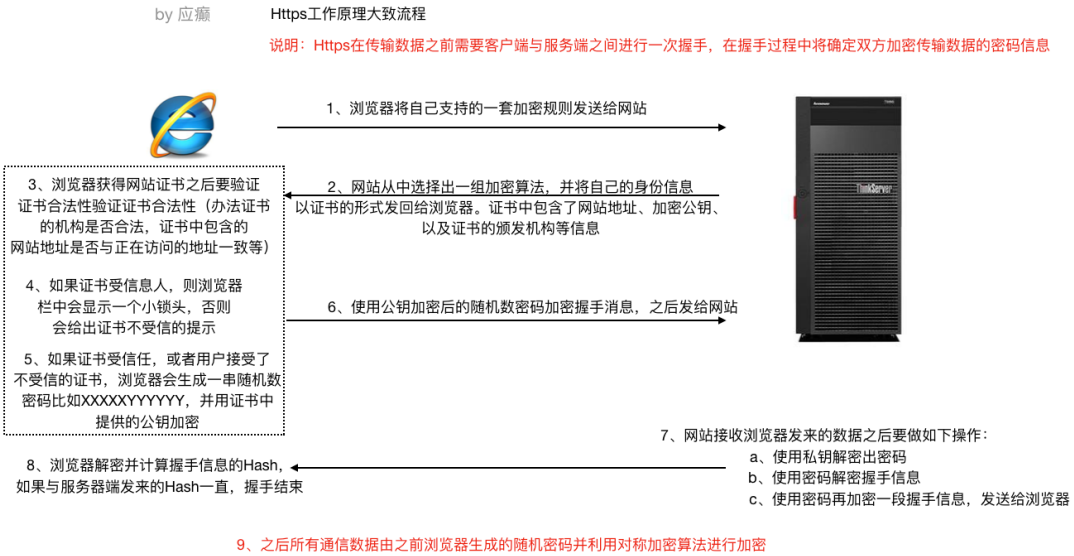
https可使用的协议：

ssl协议
TLS(transport layer security)协议（比ssl协议晚出现）

HTTPS和HTTP的主要区别

- HTTPS协议使用时需要到电子商务认证授权机构（CA）申请SSL证书
- HTTP默认使用8080端口，HTTPS默认使用8443端口
- HTTPS则是具有SSL加密的安全性传输协议，对数据的传输进行加密，效果上相当于HTTP的升级版
- HTTP的连接是无状态的，不安全的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比HTTP协议安全

第二节：HTTPS 工作原理

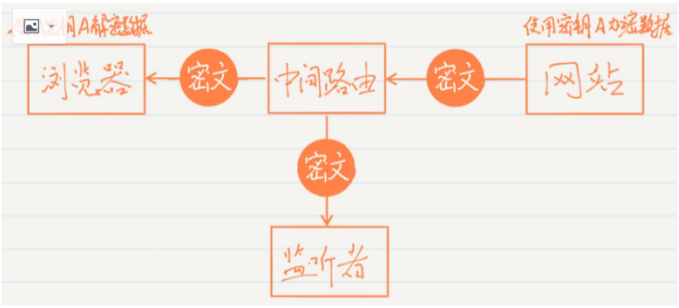


https第一次进行非对称加密（如图2.1），后面进行对称加密（如图2.2）

- **对称加密：**

优点：效率高

缺点：首次通信时无法商定密钥（浏览器和网站的首次通信过程必定是明文的。这就意味着，按照上述的工作流程，我们始终无法创建一个安全的对称加密密钥。）

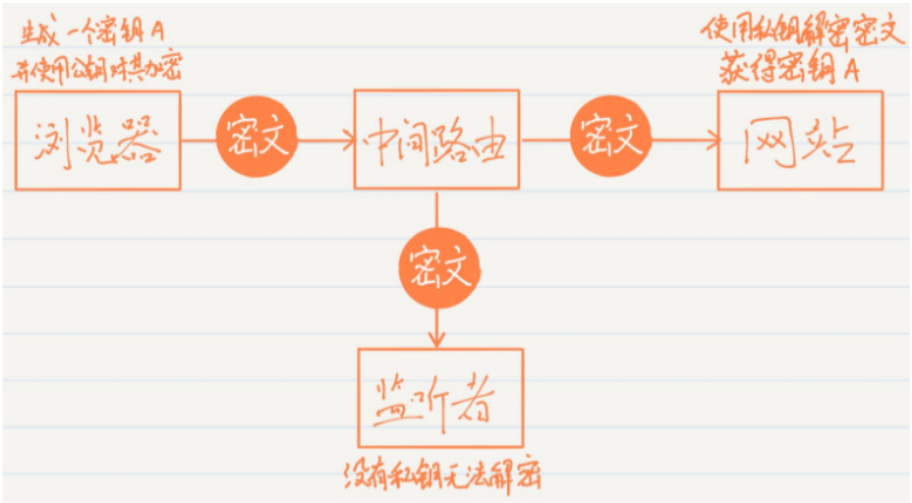


2.1

- **非对称加密：**

优点：安全性高

缺点：效率低



2.2

第三节：tomcat配置Https

1） 使用 JDK 中的 keytool 工具生成免费的秘钥库文件(证书)。

```
//命令执行如下命令：-genkey：产生密钥库文件；-alias：别名 -keyalg：密钥算法 -keystor：指定证书的名称
keytool -genkey -alias lagou -keyalg RSA -keystore lagou.keystore
```

```
localhost:conf yingdian$ keytool -genkey -alias lagou -keyalg RSA -keystore lagou.keystore
输入密钥库口令：
再次输入新口令：
您的名字与姓氏是什么？
  [Unknown]:  www.abc.com
您的组织单位名称是什么？
  [Unknown]:  lagou
您的组织名称是什么？
  [Unknown]:  lagou
您所在的城市或区域名称是什么？
  [Unknown]:  beijing
您所在的省/市/自治区名称是什么？
  [Unknown]:  beijing
该单位的双字母国家/地区代码是什么？
  [Unknown]:  cn
CN=www.abc.com, OU=lagou, O=lagou, L=beijing, ST=beijing, C=cn是否正确？
  [否]:  y
```

和网站域名保持一致

2） 配置conf/server.xml

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11NioProtocol"
maxThreads="150" schema="https" secure="true" SSLEnabled="true">
<SSLHostConfig>
<Certificate certificateKeystoreFile="/Users/yingdian/workspace/servers/apache-
tomcat-8.5.50/conf/lagou.keystore" certificateKeystorePassword="lagou123"
type="RSA"
/>
</SSLHostConfig>
</Connector>

SSLEnabled:是否开启ssl
certificateKeystoreFile：证书的位置
certificateKeystorePassword：密钥库文件密码
type：加密算法
```

3）使用https协议访问8443端口（https://localhost:8443）。

第三部分：tomcat的JVM调优

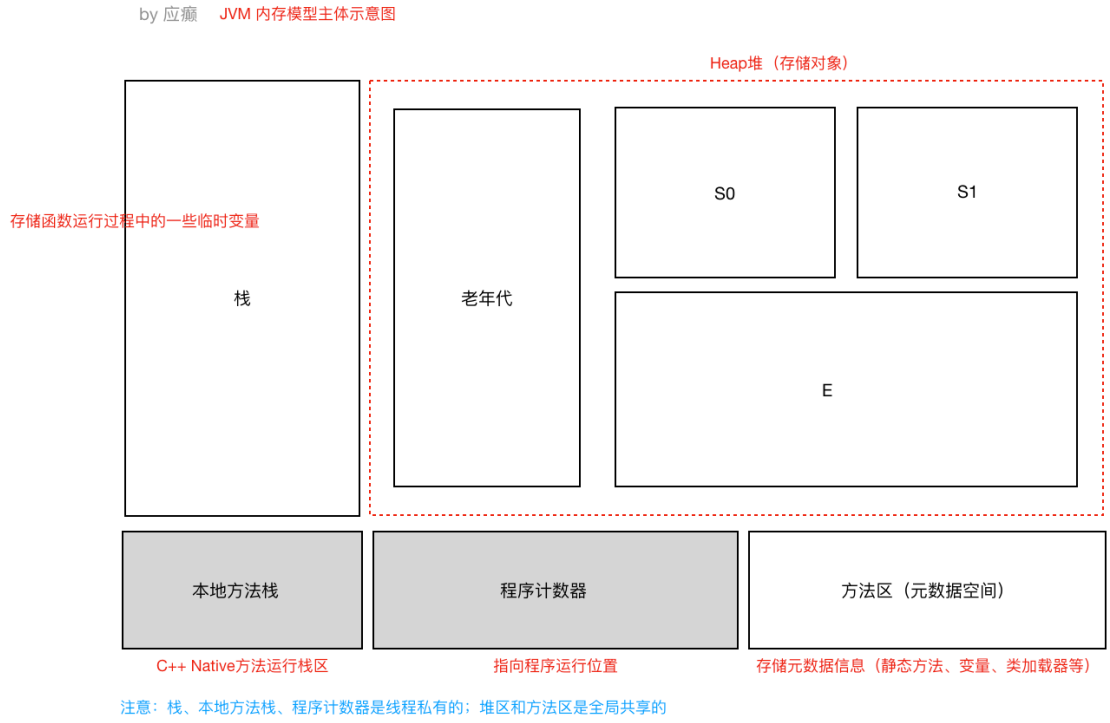
系统性能的衡量指标，主要是响应时间和吞吐量。
1）响应时间：执行某个操作的耗时；
2) 吞吐量：系统在给定时间内能够支持的事务数量，单位为TPS（Transactions PerSecond的缩写，也就是事务数/秒，一个事务是指一个客户机向服务器发送请求然后服务器做出反应的过程。
Tomcat优化从两个方面进行
1）JVM虚拟机优化（优化内存模型）
2）Tomcat自身配置的优化（比如是否使用了共享线程池？IO模型？）
学习优化的原则
提供给大家优化思路，没有说有明确的参数值大家直接去使用，必须根据自己的真实生产环境来进行调整，调优是一个过程

3.1 虚拟机运行优化（参数调整）

Java 虚拟机的运行优化主要是内存分配和垃圾回收策略的优化：

- 内存直接影响服务的运行效率和吞吐量
- 垃圾回收机制会不同程度地导致程序运行中断（垃圾回收策略不同，垃圾回收次数和回收效率都是不同的）

JVM内存模型回顾



Java 虚拟机内存相关参数

参数	参数作用	优化建议
-server	启动Server，以服务端模式运行	服务端模式建议开启（**默认client开启，Server模式启动较耗性能，运行时效率较高**）
-Xms	最小堆内存	建议与-Xmx设置相同（**假如最小和最大设置的值不同，动态调整会耗费资源，所以设置相同最好**）
-Xmx	最大堆内存	建议设置为可用内存的80%
-XX:MetaspaceSize	元空间初始值	
-XX:MaxMetaspaceSize	元空间最大内存	默认无限
-XX:NewRatio	年轻代和老年代大小比值，取值为整数，默认为2	不需要修改
-XX:SurvivorRatio	Eden区与Survivor区大小的比值，取值为整数，默认为8	不需要修改

根据以上参数信息配置最优方案(配置在catalina.bat或catalina.sh的注释下):

```
JAVA_OPTS="-server -Xms2048m -Xmx2048m -XX:MetaspaceSize=256m -
XX:MaxMetaspaceSize=512m"
```

调整后查看可使用JDK提供的内存映射工具

```
localhost:bin yingdian$ jhsdb jmap --heap --pid 8481
Attaching to process ID 8481, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 11.0.5+10-LTS

using thread-local object allocation.
Garbage-First (G1) GC with 4 thread(s)

Heap Configuration:
   MinHeapFreeRatio         = 40
   MaxHeapFreeRatio         = 70
   MaxHeapSize              = 2147483648 (2048.0MB)
   NewSize                  = 1363144 (1.2999954223632812MB)
   MaxNewSize               = 1287651328 (1228.0MB)
   OldSize                  = 5452592 (5.1999969482421875MB)
   NewRatio                 = 2
   SurvivorRatio            = 8
   MetaspaceSize            = 268435456 (256.0MB)
   CompressedClassSpaceSize = 528482304 (504.0MB)
   MaxMetaspaceSize         = 536870912 (512.0MB)
   G1HeapRegionSize         = 1048576 (1.0MB)

Heap Usage:
G1 Heap:
   regions  = 2048
   capacity = 2147483648 (2048.0MB)
   used     = 52320840 (49.89704132080078MB)
   free     = 2095162808 (1998.1029586791992MB)
   2.4363789707422256% used
G1 Young Generation:
Eden Space:
   regions  = 39
   capacity = 101711872 (97.0MB)
   used     = 40894464 (39.0MB)
   free     = 60817408 (58.0MB)
   40.20618556701031% used
Survivor Space:
   regions  = 11
   capacity = 11534336 (11.0MB)
   used     = 11534336 (11.0MB)
   free     = 0 (0.0MB)
   100.0% used
G1 Old Generation:
   regions  = 0
   capacity = 2034237440 (1940.0MB)
   used     = 0 (0.0MB)
   free     = 2034237440 (1940.0MB)
   0.0% used
```

使用内存映射工具查看tomcat中jvm内存配置

3.2 垃圾回收（GC）策略

垃圾回收性能指标

- 吞吐量：工作时间（排除GC时间）占总时间的百分比， 工作时间并不仅是程序运行的时间，还包含内存分配时间。
- 暂停时间：由垃圾回收导致的应用程序停止响应次数/时间。

垃圾收集器

- 串行收集器（Serial Collector）
  单线程执行所有的垃圾回收工作， 适用于单核CPU服务器
  工作进程-----|（单线程）垃圾回收线程进行垃圾收集|---工作进程继续
- 并行收集器（Parallel Collector）
  工作进程-----|（多线程）垃圾回收线程进行垃圾收集|---工作进程继续
  又称为吞吐量收集器（关注吞吐量）， 以并行的方式执行年轻代的垃圾回收， 该方式可以显著降低垃圾回收的开销(指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态)。适用于多处理器或多线程硬件上运行的数据量较大的应用
- 并发收集器（Concurrent Collector）
  以并发的方式执行大部分垃圾回收工作，以缩短垃圾回收的暂停时间。适用于那些响应时间优先于

吞吐量的应用， 因为该收集器虽然最小化了暂停时间(指用户线程与垃圾收集线程同时执行,但不一定是并行的，可能会交替进行)， 但是会降低应用程序的性能

- CMS收集器（Concurrent Mark Sweep Collector）
  并发标记清除收集器， 适用于那些更愿意缩短垃圾回收暂停时间并且负担的起与垃圾回收共享处理器资源的应用
- G1收集器（Garbage-First Garbage Collector）
  适用于大容量内存的多核服务器， 可以在满足垃圾回收暂停时间目标的同时， 以最大可能性实现高吞吐量(JDK1.7之后)

垃圾回收器参数

参数	描述
-XX:+UseSerialGC	启用串行收集器
-XX:+UseParallelGC	启用并行垃圾收集器，配置了该选项，那么 -XX:+UseParallelOldGC默认启用
-XX:+UseParNewGC	年轻代采用并行收集器，如果设置了 -XX:+UseConcMarkSweepGC选项，自动启用
-XX:ParallelGCThreads	年轻代及老年代垃圾回收使用的线程数。默认值依赖于JVM使用的CPU个数
-XX:+UseConcMarkSweepGC（CMS）	对于老年代，启用CMS垃圾收集器。 当并行收集器无法满足应用的延迟需求是，推荐使用CMS或G1收集器。启用该选项后， -XX:+UseParNewGC自动启用。
-XX:+UseG1GC	启用G1收集器。 G1是服务器类型的收集器， 用于多核、大内存的机器。它在保持高吞吐量的情况下，高概率满足GC暂停时间的目标。

在bin/catalina.sh的脚本中 , 追加如下配置 :

```
JAVA_OPTS="-XX:+UseConcMarkSweepGC"
```

3.3 Tomcat 配置调优

Tomcat自身相关的调优

- 调整tomcat线程池

- 调整tomcat的连接器
  调整tomcat/conf/server.xml 中关于链接器的配置可以提升应用服务器的性能。

参数	说明

参数	说明
maxConnections	最大连接数，当到达该值后，服务器接收但不会处理更多的请求， 额外的请求将会阻塞直到连接数低于maxConnections 。可通过ulimit -a 查看服务器限制。对于CPU要求更高(计算密集型)时，建议不要配置过大; 对于CPU要求不是特别高时，建议配置在2000左右(受服务器性能影响)。 当然这个需要服务器硬件的支持
maxThreads	最大线程数,需要根据服务器的硬件情况，进行一个合理的设置
acceptCount	最大排队等待数,当服务器接收的请求数量到达maxConnections ， 此时Tomcat会将后面的请求，存放在任务队列中进行排序， acceptCount指的就是任务队列中排队等待的请求数 。 一台Tomcat的最大的请求处理数量，是maxConnections+acceptCount

- 禁用 A JP 连接器

```
<!-- Define an AJP 1.3 Connector on port 8009 -->
<!--禁用AJP协议-->
<!--
<Connector port="8009" protocol="AJP/1.3"  executor="commonThreadPool"  redirectPort="8443" />
-->
```

- 调整 IO 模式
  Tomcat8之前的版本默认使用BIO（阻塞式IO），对于每一个请求都要创建一个线程来处理，不适合高并发；Tomcat8以后的版本默认使用NIO模式（非阻塞式IO）

```
<!--org.apache.coyote.http11.Http11NioProtocol ,  非阻塞式 Java NIO 链接器-->
    <Connector port="8080" protocol="HTTP/1.1"  executor="commonThreadPool"
               connectionTimeout="20000"
               redirectPort="8443" />
```

IO模型

当Tomcat并发性能有较高要求或者出现瓶颈时，我们可以尝试使用APR模式，APR（Apache Portable Runtime）是从操作系统级别解决异步IO问题，使用时需要在操作系统上安装APR和Native（因为APR原理是使用使用JNI技术调用操作系统底层的IO接口）

- 动静分离

  可以使用Nginx+Tomcat相结合的部署方案，Nginx负责静态资源访问，Tomcat负责Jsp等动态资源访问处理（因为Tomcat不擅长处理静态资源）。

3.4 io模型比较

总结上述几种IO模型，将其功能和特性进行对比：https://blog.csdn.net/szxiaohe/article/details/81542605

	同步阻塞IO（BIO）	伪异步IO	非阻塞IO（NIO）	异步IO（AIO）
客户端个数：IO线程	1：1	M：N	M：1（1个IO线程处理多个客户端连接）	M：0（不需要启动额外的IO线程，被动回调）
IO类型（阻塞）	阻塞IO	阻塞IO	非阻塞IO	非阻塞IO
IO类型（同步）	同步IO	同步IO	同步IO（IO多路复用）	异步IO
API使用难度	简单	简单	非常复杂	复杂
调试难度	简单	简单	复杂	复杂
可靠性	非常差	差	高	高
吞吐量	低	中	高	高

虽然AIO有很多优势，但并不意味着所有Java网络编程都必须选择AIO，具体选择什么IO模型或者框架，还是要基于业务的实际应用场景和性能诉求，如果客户端并发连接数不多，服务器的负载也不重，则完全没必要选择AIO做服务器，毕竟AIO编程难度相对BIO来说更大

Tomcat Servlet容器处理流程

当用户请求某个URL资源时
1）HTTP服务器会把请求信息使用ServletRequest对象封装起来
2）进一步去调用Servlet容器中某个具体的Servlet
3）在 2）中，Servlet容器拿到请求后，根据URL和Servlet的映射关系，找到相应的Servlet
4）如果Servlet还没有被加载，就用反射机制创建这个Servlet，并调用Servlet的init方法来完成初始化
5）接着调用这个具体Servlet的service方法来处理请求，请求处理结果使用ServletResponse对象封装
6）把ServletResponse对象返回给HTTP服务器，HTTP服务器会把响应发送给客户端

Tomcat6升到8分别都有什么变化?

nginx进程之间如何通信的

Spring和nginx进程模型有什么区别

nginx是如何控制并发数量的

分为**限制并发连接数**和**限制并发请求数**:

- **限制并发连接数**

  示例配置:

```
Copyhttp {
    limit_conn_zone $binary_remote_addr zone=addr:10m;
    #limit_conn_zone $server_name zone=perserver:10m;

    server {
        limit_conn addr 1;
        limit_conn_log_level warn;
        limit_conn_status 503;
    }
}
```

  limit_conn_zone key zone=name:size; 定义并发连接的配置

  - 可定义的模块为http模块。
  - key关键字是根据什么变量来限制连接数，示例中有binary_remote_addr、$server_name，根据实际业务需求。
  - zone定义配置名称和最大共享内存，若占用的内存超过最大共享内存，则服务器返回错误

  示例中的`$binary_remote_addr`是二进制的用户地址，用二进制来节省字节数，减少占用共享内存的大小。

  limit_conn zone number; 并发连接限制

- 可定义模块为http、server、location模块
- zone为指定使用哪个limit_conn_zone配置
- number为限制连接数，示例配置中限制为 1 个连接。

limit_conn_log_level info | notice | warn | error ; 限制发生时的日志级别

- 可定义模块为http、server、location模块

limit_conn_status code; 限制发生时的返回错误码，默认503

- 可定义模块为http、server、location模块

- **限制并发请求数**

limit_req_zone key zone=name:size rate=rate； 定义限制并发请求的配置。

- 若占用的内存超过最大共享内存，则服务器返回错误响应
- rate定义的是请求速率，如10r/s 每秒传递10个请求，10r/m 每分钟传递10个请求

limit_req zone=name [burst=number] [nodelay | delay=number];

- zone 定义使用哪个 limit_req_zone配置
- burst=number 设置桶可存放的请求数，就是请求的缓冲区大小
- nodelay burst桶的请求不再缓冲，直接传递，rate请求速率失效。
- delay=number 第一次接收请求时，可提前传递number个请求。
- 可定义模块为http、server、location模块

limit_req_log_level info | notice | warn | error; 限制发生时的日志级别

- 可定义模块为http、server、location模块

limit_req_status *code*；限制发生时的错误码

- 可定义模块为http、server、location模块

示例配置1

```
Copyhttp {
    limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
    limit_req zone=one burst=5;
}
```

请求速率为每秒传递1个请求。burst桶大小可存放5个请求。超出限制的请求会返回错误。

示例配置2

```
Copyhttp {
    limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
    limit_req zone=one burst=5 nodelay;
}
```

示例配置2是在示例配置1当中添加了`nodelay`选项。那么rate请求速率则不管用了。会直接传递burst桶中的所有请求。超出限制的请求会返回错误。

示例配置3

```
Copyhttp {
    limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
    limit_req zone=one burst=5 delay=3;
}
```

示例配置3是在示例配置1当中添加了`delay=3`选项。表示前3个请求会立即传递，然后其他请求会按请求速率传递。超出限制的请求会返回错误。

- 若占用的内存超过最大共享内存，则服务器返回错误响应

- rate定义的是请求速率，如10r/s 每秒传递10个请求，10r/m 每分钟传递10个请求

limit_req zone=name [burst=number] [nodelay | delay=number];

- zone 定义使用哪个 limit_req_zone配置
- burst=number 设置桶可存放的请求数，就是请求的缓冲区大小
- nodelay burst桶的请求不再缓冲，直接传递，rate请求速率失效。
- delay=number 第一次接收请求时，可提前传递number个请求。
- 可定义模块为http、server、location模块

limit_req_log_level info | notice | warn | error; 限制发生时的日志级别

- 可定义模块为http、server、location模块

limit_req_status *code*；限制发生时的错误码

- 可定义模块为http、server、location模块

示例配置1

```
Copyhttp {
    limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
    limit_req zone=one burst=5;
}
```

请求速率为每秒传递1个请求。burst桶大小可存放5个请求。超出限制的请求会返回错误。

示例配置2

```
Copyhttp {
    limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
    limit_req zone=one burst=5 nodelay;
}
```

示例配置2是在示例配置1当中添加了`nodelay`选项。那么rate请求速率则不管用了。会直接传递burst桶中的所有请求。超出限制的请求会返回错误。

示例配置3

```
Copyhttp {
    limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
    limit_req zone=one burst=5 delay=3;
}
```

示例配置3是在示例配置1当中添加了`delay=3`选项。表示前3个请求会立即传递，然后其他请求会按请求速率传递。超出限制的请求会返回错误。