1 What is Tomcat

```
The Apache Tomcat® software is an open source implementation of the Java Servlet, JavaServer
Pages, Java Expression Language and Java WebSocket technologies.
```

2 Version choose and reason

2.1 Version choose

Tomcat版本：Tomcat8.0.11

jdk版本：大于等于jdk1.7—>【`Download/Which version`】

tomcat各个版本下载地址：【`Download/Archives`】[各个版本产品和源码]

2.2 Reason

`在tomcat7.0中没有NIO2，在tomcat8.5中没有BIO，而在tomcat8.0中支持的比较丰富`

可以在源码中验证一下:AbstractEndpoint.bind()--->implementation

3 了解回顾

3.1 源码需要引入的pom.xml文件

`在tomcat源码的根目录新建pom.xml文件，将下面这段内容复制到pom.xml文件中`

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>Tomcat8.0</artifactId>
    <name>Tomcat8.0</name>
    <version>8.0</version>

    <build>
        <finalName>Tomcat8.0</finalName>
        <sourceDirectory>java</sourceDirectory>
        <testSourceDirectory>test</testSourceDirectory>
        <resources>
            <resource>
```

```
                <directory>java</directory>
            </resource>
        </resources>
        <testResources>
            <testResource>
                <directory>test</directory>
            </testResource>
        </testResources>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.3</version>
                <configuration>
                    <encoding>UTF-8</encoding>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.easymock</groupId>
            <artifactId>easymock</artifactId>
            <version>3.4</version>
        </dependency>
        <dependency>
            <groupId>ant</groupId>
            <artifactId>ant</artifactId>
            <version>1.7.0</version>
        </dependency>
        <dependency>
            <groupId>wsdl4j</groupId>
            <artifactId>wsdl4j</artifactId>
            <version>1.6.2</version>
        </dependency>
        <dependency>
            <groupId>javax.xml</groupId>
            <artifactId>jaxrpc</artifactId>
            <version>1.1</version>
        </dependency>
        <dependency>
            <groupId>org.eclipse.jdt.core.compiler</groupId>
            <artifactId>ecj</artifactId>
            <version>4.5.1</version>
```

```
        </dependency>
    </dependencies>
</project>
```

3.2 tomcat产品目录文件含义

（1）bin：主要用来存放命令，.bat是windows下，.sh是Linux下

（2）conf：主要用来存放tomcat的一些配置文件

（3）lib：存放tomcat依赖的一些jar包

（4）logs：存放tomcat在运行时产生的日志文件

（5）temp：存放运行时产生的临时文件

（6）webapps：存放应用程序

（7）work：存放tomcat运行时编译后的文件，比如JSP编译后的文件

这块咱们就不详细去说了，因为在Javaweb中都学过，即使忘了一些文件或者文件夹的作用，网上介绍的一大堆

3.3 tomcat额外需知

（1）Java语言写的

（2）servlet/jsp technologies

4 不妨手写一个mini的Tomcat

为什么要手写？既然上述提到了tomcat是java语言写的，又和servlet相关，那就自己设计一个试试，先不管作者的想法如何

4.1 确定tomcat作用

web服务器，说白了就是能够让客户端和服务端进行交互，比如客户端想要获取服务端某些资源，服务端可以通过tomcat去进行一些处理并且返回。

4.2 基于Socket进行网络通信

```
//基于网络编程socket套接字来做
class MyTomcat{
    ServerSocket server=new ServerSocket(8080);
    Socket socket=server.accept();
    InputStream in=socket.getInputStream();
    OutputStream out=socket.getOutputStream();
}
```

实际上就是通过serversocket在服务端监听一个端口，等待客户端的连接，然后能够获取到对应的输入输出流

4.3 优化

```
//优化1:将输入输出流封装到对象
class MyTomcat{
    ServerSocket server=new ServerSocket(8080);
    Socket socket=server.accept();
    InputStream in=socket.getInputStream();
    new Request(in);
    OutputStream out=socket.getOutputStream();
    new Response(out);
}
class Request{private String host;private String accept-language;}
class Response{}
```

发现一个比较靠谱的tomcat已经被我们写出来了，问题是这个tomcat如果使用起来方便吗？你会发现不方便，因为对应的request和response都放到了tomcat源码的内部，业务人员想要进行开发时，很难获得request对象，从而获得客户端传来的数据，也不能进行很好的返回，怎么办呢？

我们发现在JavaEE中有servlet这项技术，比如我们进行登录功能业务代码开发时,写过如下这段代码和配置

```
//优化2前奏:
class LoginServlet extends HttpServlet{
    doGet(request,response){}
    doPost(request,response){}
}
<servlet>
       <servlet-name>LoginServlet</servlet-name>
       <servlet-class>com.gupao.web.servlet.SimpleServlet</servlet-class>
</servlet>
<servlet-mapping>
       <servlet-name>LoginServlet</servlet-name>
       <url-pattern>/login</url-pattern>
</servlet-mapping>
```

所以不妨让tomcat也实现servlet规范，这时候手写的tomcat源码就可以做一个改变

```
//优化2:
class MyTomcat{
    List list=new ArrayList();
    ServerSocket server=new ServerSocket(8080);
    Socket socket=server.accept();
    //也就是这个地方不是直接处理request和response
    //而是处理一个个servlets
    list.add(servlets);
}
```

4.4 手写版tomcat[servlets]真的可行吗?

换句话说：tomcat官方开发者对于用list集合保存项目中的servlets也是这样想的吗？我们可以从几个维度进行一下推测

4.4.1 servlet之业务代码

业务代码中关于servlet想必大家都配置过，或者用注解的方式，原本开发web应用就采用的是这样的方式。你的controller中有很多自己写的servlet，都继承了HttpServlet类，然后web.xml文件中配置过所有的servlets，也就是mapping映射，这个很简单。

4.4.2 servlet之产品角度

如果apache提供的tomcat也这么做了，势必也要跟servlet规范有关系，也就是要依赖servlet的jar包，我们来看一下在tomcat产品的bin文件夹之下有没有servlet.jar，发现有。

4.4.3 servlet之源码角度

`最后我们如果能够在tomcat源码中找到载入servlets的依据，就更加能说明问题了`

于是我们在idea中的tomcat8.0源码，关键是到哪里找呢？总得有个入口吧？源码中除了能够看到各种Java类型的文件之外，一脸懵逼，怎么办？

不妨先跳出来想想，如果我们是tomcat源码的设计者，也就是上述手写的代码，我们怎么将业务代码中的servlets加载到源码中？我觉得可以分为两步

（1）加载web项目中的web.xml文件，解析这个文件中的servlet标签，将其变成java中的对象

（2）在源码中用集合保存

`注意第（1）步，为什么是加载web.xml文件呢？因为要想加载servlets，一定是以web项目为单位的，而一个web项目中有多少个servlet类，是会配置在web.xml文件中的。`

`寻找和验证`

- 加载和解析web.xml文件

`加载`：ContextConfig.webConfig()—>getContextWebXmlSource()—>Constants.ApplicationWebXml

`解析`：ContextConfig.webConfig()—>configureContext(webXml)—>context.createWrapper()

- 将servlets加载到list集合中

StandardContext.loadOnStartup(Container children[])—>list.add(wrapper)

4.4.4 加载servlets的疑惑

`怎么知道上面找的过程的？`

我们会发现上面加载web.xml文件和添加servlets都和Context有点关系，因为都有这个单词，那这个Context大家眼熟吗？其实我们见过，比如你把web项目想要供外界访问时，你会添加web项目到webapps目录，这是tomcat的规定，除此之外，还可以在conf/server.xml文件中配置Context标签。

`按照经验之谈，一般框架的设计者都会提供一两个核心配置文件给我们，比如server.xml就是tomcat提供给我们的，而这些文件中的标签属性最终会对应到源码的类和属性`

4.5 手写版tomcat[监听端口]可行吗？

`换句话说：tomcat官方开发者对于监听端口也是这么设计的吗`

其实我们手写的tomcat这块有两个核心：第一是监听端口，第二是添加servlets，上面解决了添加servlets。

接下来显然我们有必要验证一下监听端口tomcat也是这么做的吗?

4.5.1 监听端口之画图

`上述的课堂中呈现出来的最好只有MyTomcat和Connector这块，其他先不管`

`在tomcat这块左边一定会监听在某个端口，等待客户端的连接，不然所有的操作都没办法进行交互`

4.5.2 监听端口之源码角度

Connector.initInternal()->protocolHandler.init()->AbstractProtocol.init()->endpoint.init()->bind()->Apr,JIo,NIO,NIO2->JIo即Socket实现方式

4.5.3 监听端口的疑惑

`为什么知道找Connector?`

再次回到conf/web.xml文件，发现有一个Connector标签，而且还可以配置port端口，我们能够联想到监听端口，按照配置文件到源码类的经验，源码中一定会有这样一个Connector类用于端口的监听。

4.6 完善自己的tomcat架构图

4.7 推导出tomcat架构图

`conclusion: 架构图<--->server.xml<--->源码 三者有一一对应的关系`

5 折腾Tomcat架构和源码

5.1 认识强化主要组件的含义

`官网`:Documentation/Tomcat8.0/Apache Tomcat Development/Architecture/Overview

- Server

  ```
  In the Tomcat world, a Server represents the whole container. Tomcat provides a default implementation of the Server interface which is rarely customized by users.
  ```

- Service

  ```
  A Service is an intermediate component which lives inside a Server and ties one or more Connectors to exactly one Engine. The Service element is rarely customized by users, as the default implementation is simple and sufficient: Service interface.Engine
  ```

- Engine

```
An Engine represents request processing pipeline for a specific Service. As a Service may
have multiple Connectors, the Engine receives and processes all requests from these
connectors, handing the response back to the appropriate connector for transmission to the
client. The Engine interface may be implemented to supply custom Engines, though this is
uncommon.
Note that the Engine may be used for Tomcat server clustering via the jvmRoute parameter.
Read the Clustering documentation for more information.
```

- Host

```
A Host is an association of a network name, e.g. www.yourcompany.com, to the Tomcat server.
An Engine may contain multiple hosts, and the Host element also supports network aliases
such as yourcompany.com and abc.yourcompany.com. Users rarely create custom Hosts because
the StandardHost implementation provides significant additional functionality.
```

- Connector

```
A Connector handles communications with the client. There are multiple connectors available
with Tomcat. These include the HTTP connector which is used for most HTTP traffic,
especially when running Tomcat as a standalone server, and the AJP connector which
implements the AJP protocol used when connecting Tomcat to a web server such as Apache
HTTPD server. Creating a customized connector is a significant effort.
```

- Context

```
A Context represents a web application. A Host may contain multiple contexts, each with a
unique path. The Context interface may be implemented to create custom Contexts, but this
is rarely the case because the StandardContext provides significant additional
functionality.
```

5.2 它们是如何协同工作的?

`换句话说：之前找了两个点，监听端口，加载servlets的调用过程是如何的?`

比如bind(),loadOnstartup()到底谁来调用?

此时大家还是要回归到最初的流程，客户端发起请求到得到响应来看。

`客户端角度`：发起请求，最终得到响应

`tomcat代码角度`：虽然是要监听端口和添加servlets进来，但是肯定有一个主函数，从主函数开始调用

说白了，如果我是源码设计者，既然架构图我都了解了，肯定是要把这些组件初始化出来，然后让它们一起工作，也就是:

- 初始化一个个组件
- 利用这些组件进行相应的操作

5.2.1 寻找源码开始的地方

一定有一个类，这个类中有main函数开始，这样才能有一款java源码到产品，一贯的作风。

感性的认知：`BootStrap`->main()->根据脚本命令->startd

- daemon.load()　加载
- daemon.start()　启动

`果然被我们找到了，先加载再启动，那就继续看咯`

5.2.2 加载:daemon.load()的过程

Bootstrap.main()->Bootstrap.load()->Catalina.load()->初始化的依据是什么？考虑coder的设计server.xml

->Lifecycle.init()->LifecycleBase.init()->LifecycleBase.initInternal()->StandardServer.initInternal()

->services[i].init()->StandardService.initInternal()->executor.init()/`connector.init()`

->LifecyleBase.initInternal()->Connector.initInternal()->protocolHandler.init()->AbstractProtocol.init()

->endpoint.init()->bind()->Apr,JIo,NIO,NIO2

`conclusion:请求目前没有来，只是内部的初始化工作`

5.2.3 启动:daemon.start()的过程

Bootstrap.start()->Catalina.start()->getServer.start()->LifecycleBase.start()->LifecycleBase.startInternal()

->StandardServer.startInternal()->services[i].start()->StandardService.startInternal()

->`container.start()[查看一下Container接口]`/executors.init()/connectors.start()->engine.start()->StandardEngine.startInternal()

查看一下StandardEngine类关系结构图，发现ContainerBase是它的爸爸，而这个爸爸有多少孩子呢?

```
Engine,Host,Context,Wrapper都是它的孩子
```

->super[ContainerBase].startInternal()->代码呈现

```
results.add(startStopExecutor.submit(new StartChild(children[i])))
```

关注到new StartChild(children[i])--->child.start()，也就是会调用Engine子容器的start方法，那子容器是什么呢?Host，child.start->LifecycleBase.start()->startInternal()->StandardHost.startInternal()

- Host将一个个web项目加载进来

StandardHost.startInternal()->ContainerBase.startInternal()->最后threadStart()

->new Thread(new ContainerBackgroundProcessor())->`run()[processChildren(ContainerBase.this)]`

->container.backgroundProcess()->ContainerBase.backgroundProcess()

->fireLifecycleEvent(Lifecycle.PERIODIC_EVENT, null)->listener.lifecycleEvent(event)

->interested[i].lifecycleEvent(event)->监听器HostConfig->HostConfig.lifecycleEvent(LifecycleEvent event)

->check()->`deployApps()`

```
// Deploy XML descriptors from configBase
deployDescriptors(configBase, configBase.list());
// Deploy WARs
deployWARs(appBase, filteredAppPaths);
// Deploy expanded folders
deployDirectories(appBase, filteredAppPaths);
```

回到`StandardHost.startInternal()`->super.startInternal()

`results.add(startStopExecutor.submit(new StartChild(children[i])));`

然后又会调用它的子容器->super.startInternal()->StandardContext.initInternal()

- StandardContext.startInternal()解析每个web项目

```
fireLifecycleEvent(Lifecycle.CONFIGURE_START_EVENT, null)->listener.lifecycleEvent(event)
interested[i].lifecycleEvent(event)->[找实现]ContextConfig.lifecycleEvent(LifecycleEvent event)-
configureStart()->webConfig()->解析每个web项目的xml文件了->getContextWebXmlSource()-
>Constants.ApplicationWebXml
```

`ContextConfig.webConfig()的step9解析到servlets包装成wrapper对象`

- 何时调用loadOnstartup()

  StandardContext.startInternal()->最终会调用`if (!loadOnStartup(findChildren()))`

5.3 官网验证上述流程

5.3.1 Server Startup

Documentation/Tomcat8.0/Apache Tomcat Development/Architecture/Server Startup:Server Startup

5.3.2 Request Process

Documentation/Tomcat8.0/Apache Tomcat Development/Architecture/Request Process:UML sequence diagram

6 Tomcat性能优化思路

6.1 优化思路过渡

上面说了这么多，接下来咱们就来聊聊tomcat的性能优化，那怎么进行优化？哪些方面需要进行优化？先有一个整体的认知。

其实还是要回归到问题的本质，一个客户端的连接请求响应的流程，看看这个过程经历了什么，哪些地方能够优化。

> 当然，我要补充的一点是，服务器的CPU、内存、硬盘等对性能有决定性的影响，硬件这块配置越高越好。

`再次看tomcat architecture`：

- 发现客户端的连接请求会和Connector打交道，对于Connector可以进行选择，比如Http Connector，AJP Connector。

  `整体介绍`:Documentation/Tomcat8.0/User Guide/21)Connectors链接

  `详细介绍`:Documentation/Tomcat8.0/Reference/Configuration/Connectors链接

- Executor

  `介绍`:Documentation/Tomcat8.0/Reference/Configuration/Executors链接

- Context

  `介绍`:Documentation/Tomcat8.0/Reference/Configuration/Containers/Context链接

- Context中加载web.xml文件时的源码

  处理一些过滤器，全局servlet，session等等这些有一个全局的web.xml文件，在conf目录下，源码中会将两者进行合并处理。

`conclusion:要想改变上面这些内容，适当进行调整，咱们去修改tomcat源码显然不合适，那怎么修改呢？tomcat给我们提供了可以进行定制自己组建的相关配置文件，比如说conf目录下的server.xml和web.xml文件,也就是说我们可以站在修改配置文件的角度进行性能优化`

`继续思考tomcat性能优化思路`

既然tomcat是Java写的，最终这些代码是会跑到jvm虚拟机中的，也就是说jvm的一些优化思路也可以在tomcat中进行落实。

6.2 配置优化

由前面的分析可以定位目前两个重要的配置文件`conf/server.xml` `conf/web.xml`

6.2.1 conf/server.xml核心组件

- Server

`官网描述`:Server interface which is rarely customized by users.【pass】

- Service

`官网描述`:The Service element is rarely customized by users.【pass】

- Connector

`官网描述`:Creating a customized connector is a significant effort.【`need`】

- Engine

`官网描述`:The Engine interface may be implemented to supply custom Engines, though this is uncommon.【pass】

- Host

`官网描述`:Users rarely create custom Hosts because the StandardHost implementation provides significant additional functionality.【pass】

- Context

官网描述:The Context interface may be implemented to create custom Contexts, but this is rarely the case because the StandardContext provides significant additional functionality. 【maybe】

Context既然代表的是web应用，是和我们比较接近的，这块我们考虑对其适当的优化

> conclusion:Connector and Context

6.2.2 conf/server.xml非核心组件

官网:Documentation/Reference/Configuration/Nested Components/xxx

- Listener

Listener(即监听器)定义的组件，可以在特定事件发生时执行特定的操作；被监听的事件通常是Tomcat的启动和停止。

```
<Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />
<!--监听内存溢出-->
<Listener className="org.apache.catalina.core.JreMemoryLeakPreventionListener" />
<Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
<Listener className="org.apache.catalina.core.ThreadLocalLeakPreventionListener" />
```

- Global Resources

GlobalNamingResources元素定义了全局资源，通过配置可以看出，该配置是通过读取$TOMCAT_HOME/ conf/tomcat-users.xml实现的。

The GlobalNamingResources element defines the global JNDI resources for the [Server](https://tomcat.apache.org/tomcat-8.0-doc/config/server.html)

```
<GlobalNamingResources>
    <Resource name="UserDatabase" auth="Container"
    type="org.apache.catalina.UserDatabase"
    description="User database that can be updated and saved"
    factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
    pathname="conf/tomcat-users.xml" />
</GlobalNamingResources>
```

- Valve

功能类似于过滤器Filter

```
<Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"
               prefix="localhost_access_log" suffix=".txt"
               pattern="%h %l %u %t &amp;quot;%r&amp;quot; %s %b" />
```

- Realm

Realm，可以把它理解成“域”；Realm提供了一种用户密码与web应用的映射关系，从而达到角色安全管理的作用。在本例中，Realm的配置使用name为UserDatabase的资源实现。而该资源在Server元素中使用GlobalNamingResources配置

`A Realm element represents a "database" of usernames, passwords, and roles (similar to Unix groups) assigned to those users.`

```
<Realm className="org.apache.catalina.realm.LockOutRealm">
    <!-- This Realm uses the UserDatabase configured in the global JNDI
    resources under the key "UserDatabase".  Any edits
    that are performed against this UserDatabase are immediately
    available for use by the Realm.  -->
    <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
    resourceName="UserDatabase"/>
</Realm>
```

6.2.3 conf/web.xml

全局的web.xml文件有些标签用不到的，可以删除掉，具体后面会说。

6.3 JVM优化

6.3.1 内存设置

为了防止内存不够用，显然可以设置一下内存的大小

6.3.2 GC算法

选择合适的GC算法，其实内存大小的设置也会影响GC

6.4 小结

- 减少相关配置->查看日志tomcat启动时间
- `项目方法`:Connector->BIO/NIO/APR->压测某个项目的方法观察Throughout
- `JVM`:jconsole,gceasy.io,jvisual

7 相关环境准备

7.1 windows

- jdk1.8
- maven
- git
- idea
- tomcat8.0

- Xshell
- jmeter

  `用于本地压测观察`
- ftp/rzsz

  `用于本地和远端文件交互`

7.2 centos7

- 本地搭建或者购买阿里云服务器安装centos系统

  ```
  这是我上课用的账号和密码，只有一个月有效期
  ip地址:39.98.168.189
  账户:root
  密码:AAbb1234
  ```

- jdk1.8

  `本地上传到生产环境,解压配置环境变量`
- maven

  `本地上传到生产环境,解压配置环境变量`
- tomcat8.0

  `本地上传到生产环境,解压配置环境变量`

  `如果tomcat启动慢，则catalina.sh的JAVA_OPTS加入-Djava.security.egd=file:/dev/./urandom`

7.3 一个web项目

```
项目名称:gp/index.html
项目介绍:
    其实就是一个静态页面，接下来就是对这个页面进行压测
```

7.4 JVisualVM监控java进程

（1）命令行输入jvisualvm

（2）选择本地的java进程【本地无需任何设置，直接连接即可】

（3）监控远程tomcat

输入远程ip地址，修改远端的catalina.sh文件，添加如下内容

```
JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote
-Djava.rmi.server.hostname=39.98.168.189 -Dcom.sun.management.jmxremote.port=8998
-Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.authenticate=true
-Dcom.sun.management.jmxremote.password.file=../conf/jmxremote.password
-Dcom.sun.management.jmxremote.access.file=../conf/jmxremote.access"
```

然后在jvisualvm中添加远程连接，JMX类型

`采坑指南`

查看hostname -i，修改/etc/hosts文件公网ip地址指向查询到的地址

lsof -i tcp:8998 得到PID

然后netstat -antup | grep PID 得到几个端口号，在阿里云安全组中添加相应端口

上述修改之后要重启远端的tomcat

添加JMX Connection，注意端口写8999

防火墙要记得添加对应的策略

在conf文件中添加两个文件jmxremote.access和jmxremote.password，内容分别为

```
guest readonly
manager readwrite
```

```
guest guest
manager manager
```

授予文件相应权限:`chmod 600 *jmxremot*`

7.5 tomcat-manager/probe

如果不想用jvisualvm来监控tomcat线程内存的信息，也可以选择tomcat自带的tomcat-manager或者probe来监控，只是有些功能没有那么完善。

7.6 课程中常用命令

- 启动停止tomcat,来到tomcat的bin文件夹
  - 启动: ./startup.sh
  - 停止:./shutdown.sh
- 查看tomcat启动日志

  cat ../logs/catalina.out

  tail -f ../logs/catalina.out 相当于监控该日志文件
- 解压

  tar -zxvf xxx
- 查看进程及端口号
  - 查看进程: ps -ef | grep tomcat/java
  - 端口号:lsof -i tcp:8080
- 杀掉进程

  kill PID
- 查看jvm内所有线程

jstack PID

官网线程状态描述

- maven打包

  mvn clean package

  mvn clean package -Dmaven.test.skip

- jinfo查看jvm某个参数是否启用

  jinfo -flag UseParallelGC PID

  jinfo -flag MaxHeapSize PID 查看最大内存

  jinfo -flag UseG1GC PID 查看垃圾回收器

  jinfo -flags PID 查看曾经赋过值的一些参数

- 查看java进程

  jps -l

- 查看jvm统计信息

  jstat -class/-gc PID 1000 10

- 导出内存信息

```
jmap -dump:format=b,file=heap.hprof pid
jmap -heap pid    打印出堆内存相关的信息
```

8 Tomcat性能优化

写的不错的一篇文章链接:资料

8.1 配置优化

8.1.1 减少web.xml/server.xml中标签

最终观察tomcat启动日志[时间/内容]，线程开销，内存大小，GC等

- DefaultServlet

官网:User Guide->Default Servlet

The default servlet is the servlet which serves static resources as well as serves the directory listings (if directory listings are enabled).

```
<servlet>
        <servlet-name>default</servlet-name>
        <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
        <init-param>
            <param-name>debug</param-name>
            <param-value>0</param-value>
        </init-param>
        <init-param>
```

```
            <param-name>listings</param-name>
            <param-value>false</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
        <servlet-name>default</servlet-name>
        <url-pattern>/</url-pattern>
</servlet-mapping>
```

- JspServlet

```
<servlet>
    <servlet-name>jsp</servlet-name>
    <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
    <init-param>
        <param-name>fork</param-name>
        <param-value>false</param-value>
    </init-param>
    <init-param>
        <param-name>xpoweredBy</param-name>
        <param-value>false</param-value>
    </init-param>
    <load-on-startup>3</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>jsp</servlet-name>
    <url-pattern>*.jsp</url-pattern>
    <url-pattern>*.jspx</url-pattern>
</servlet-mapping>
```

- welcome-list-file

```
<welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

- mime-mapping移除响应的内容

支持的下载打开类型

```
<mime-mapping>
    <extension>123</extension>
    <mime-type>application/vnd.lotus-1-2-3</mime-type>
</mime-mapping>
<mime-mapping>
    <extension>3dml</extension>
    <mime-type>text/vnd.in3d.3dml</mime-type>
</mime-mapping>
```

- session-config

默认jsp页面有session，就是在于这个配置

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

8.1.2 调整优化server.xml中标签

8.1.2.1 Connector标签

- protocol属性

```
<Connector port="8080" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="8443" />
```

对于protocol="HTTP/1.1"，查看源码

构造函数

```
public Connector(String protocol) {
        setProtocol(protocol);
}
```

setProtocol(protocol)因为配置文件中传入的是HTTP/1.1

并且这里没有使用APR，一会我们会演示APR

```
else {
    if ("HTTP/1.1".equals(protocol)) {
    setProtocolHandlerClassName
    ("org.apache.coyote.http11.Http11NioProtocol");
    } else if ("AJP/1.3".equals(protocol)) {
    setProtocolHandlerClassName
    ("org.apache.coyote.ajp.AjpNioProtocol");
    } else if (protocol != null) {
    setProtocolHandlerClassName(protocol);
    }
}
```

`发现这里调用的是Http11NioProtocol，也就是说明tomcat8.0.x中默认使用的是NIO`

> 使用同样的方式看tomcat7和tomcat8.5，你会发现tomcat7默认使用的是BIO，tomcat8.5默认使用的是NIO

`针对BIO和NIO的方式进行压测`

（1）BIO

来到tomcat官网Configuration/HTTP/protocol

```
org.apache.coyote.http11.Http11Protocol - blocking Java connector
org.apache.coyote.http11.Http11NioProtocol - non blocking Java NIO connector
org.apache.coyote.http11.Http11Nio2Protocol - non blocking Java NIO2 connector
org.apache.coyote.http11.Http11AprProtocol - the APR/native connector.
```

`使用tomcat7取巧一下，默认是BIO，端口改成7070，将tomcat-optimize复制到tomcat7中`

（2）NIO

tomcat8.0中默认使用的是NIO

`针对上述BIO和NIO的方式，进行压测，调整并发数，看吞吐量`

（3）APR

【具体查看操作手册/APR安装方式】

```
下载以下内容：
apr
apr-iconv
apr-util
并且上传到centos中
```

先安装一些依赖库：`yum install apr* openssl-devel gcc make`

```
安装apr
解压apr，cd到源码目录
./configure --prefix=/usr/local/apr     指定安装的目录
make      编译
make install       安装
```

```
安装apr-iconv
解压apr-iconv，cd到源码目录
./configure --prefix==/usr/local/apr-iconv --with-apr=/usr/local/apr   使用了一下刚才的apr
make
make install
安装依赖包  yum install expat-devel
```

```
安装apr-util
解压apr-util,cd到源码目录
./configure --prefix=/usr/local/apr-util --with-apr=/usr/local/apr
make
make install
```

```
安装openssl   1.0.2k
解压，cd到源码目录
./config --prefix=/usr/local/openssl
修改Makefile文件，将CFLAG=-DOPENSSL_THREADS修改为CFLAG= -fPIC -DOPENSSL_THREADS
make
make install
```

```
来到tomcat的bin目录
解压tomcat-native.tar.gz
cd tomcat-native-1.x-src/jni/native
./configure --with-apr=/usr/local/apr --with-ssl=/usr/local/openssl
make
make install
```

```
配置bin/catalina.sh
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/apr/lib
export LD_LIBRARY_PATH
```

```
server.xml中的AprListener需要关闭SSL的方式，值设置为off
```

重新启动tomcat，看日志apr的方式

`也可以压测看一下效果`

- executor属性

`最佳线程数公式`:((线程等待时间+线程cpu时间)/线程cpu时间) * cpu数量

```
 The Executor represents a thread pool that can be shared between components in Tomcat.
 Historically there has been a thread pool per connector created but this allows you to share a
 thread pool, between (primarily) connector but also other components when those get configured
 to support executors
```

设置一些属性

`官网:https://tomcat.apache.org/tomcat-8.0-doc/config/http.html`

（1）acceptCount:达到最大连接数之后，等待队列中还能放多少连接，超过即拒绝，配置太大也没有意义

```
The maximum queue length for incoming connection requests when all possible request processing threads are in use. Any requests received when the queue is full will be refused. The default value is 100.
```

（2）maxConnections

达到这个值之后，将继续接受连接，但是不处理，能继续接受多少根据acceptCount的值

BIO:maxThreads

NIO/NIO2:10000 ——— AbstractEndpoint.maxConnections

APR:8192

```
The maximum number of connections that the server will accept and process at any given time. When this number has been reached, the server will accept, but not process, one further connection. This additional connection be blocked until the number of connections being processed falls below maxConnections at which point the server will start accepting and processing new connections again. Note that once the limit has been reached, the operating system may still accept connections based on the acceptCount setting. The default value varies by connector type. For BIO the default is the value of maxThreads unless an Executor is used in which case the default will be the value of maxThreads from the executor. For NIO and NIO2 the default is 10000. For APR/native, the default is 8192.

Note that for APR/native on Windows, the configured value will be reduced to the highest multiple of 1024 that is less than or equal to maxConnections. This is done for performance reasons.
If set to a value of -1, the maxConnections feature is disabled and connections are not counted.
```

（3）maxThreads:最大工作线程数，也就是用来处理request请求的，默认是200，如果自己配了executor，并且和Connector有关联了，则之前默认的200就会被忽略，取决于CPU的配置。监控中就可以看到所有的工作线程是什么状态，通过监控就能知道开启多少个线程合适

```
The maximum number of request processing threads to be created by this Connector, which therefore determines the maximum number of simultaneous requests that can be handled. If not specified, this attribute is set to 200. If an executor is associated with this connector, this attribute is ignored as the connector will execute tasks using the executor rather than an internal thread pool. Note that if an executor is configured any value set for this attribute will be recorded correctly but it will be reported (e.g. via JMX) as -1 to make clear that it is not used.
```

（4）minSpareThreads

最小空闲线程数

```
The minimum number of threads always kept running. This includes both active and idle threads. If not specified, the default of 10 is used. If an executor is associated with this connector, this attribute is ignored as the connector will execute tasks using the executor rather than an internal thread pool. Note that if an executor is configured any value set for this attribute will be recorded correctly but it will be reported (e.g. via JMX) as -1 to make clear that it is not used.
```

可以实践一下，Connector配合自定义的线程池

```
 <Connector executor="tomcatThreadPool"
               port="8080" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="8443" />
```

```
<Executor name="tomcatThreadPool" namePrefix="catalina-exec-"
        maxThreads="150" minSpareThreads="4"/>
```

其实这块最好的方式是结合BIO来看，因为BIO是一个request对应一个线程

```
值太低，并发请求多了之后，多余的则进入等待状态。
值太高，启动Tomcat将花费更多的时间。
比如可以改成250。
```

- enableLookups

设置为false

- 删掉AJP的Connector

8.1.2.2 Host标签

autoDeploy :Tomcat运行时，要用一个线程拿出来进行检查，生产环境之下一定要改成false

```
This flag value indicates if Tomcat should check periodically for new or updated web applications
while Tomcat is running. If true, Tomcat periodically checks the appBase and xmlBase directories
and deploys any new web applications or context XML descriptors found. Updated web applications
or context XML descriptors will trigger a reload of the web application. The flag's value
defaults to true. See Automatic Application Deployment for more information.
```

8.1.2.3 Context标签

reloadable:false

```
reloadable:如果这个属性设为true，tomcat服务器在运行状态下会监视在WEB-INF/classes和WEB-INF/lib目录下
class文件的改动，如果监测到有class文件被更新的，服务器会自动重新加载Web应用。
在开发阶段将reloadable属性设为true，有助于调试servlet和其它的class文件，但这样用加重服务器运行负荷，建议
在Web应用的发存阶段将reloadable设为false。
```

```
Set to true if you want Catalina to monitor classes in /WEB-INF/classes/ and /WEB-INF/lib for
changes, and automatically reload the web application if a change is detected. This feature is
very useful during application development, but it requires significant runtime overhead and is
not recommended for use on deployed production applications. That's why the default setting for
this attribute is false. You can use the Manager web application, however, to trigger reloads of
deployed applications on demand.
```

8.2 JVM优化

8.2.1 JVM优化过渡

为什么会有JVM这块的优化？因为tomcat是java语言写的，那么对于jvm这块的优化在tomcat中就是适用的。比如修改一些参数，调整内存大小，选择合适的垃圾回收算法等等。

现在有个问题，修改JVM参数在哪里修改会对tomcat生效？还是在bin文件夹之下，有一个catalina.sh，找到JAVA_OPTS即可，当然不建议对此文件进行直接修改，一般是在外面新建一个文件，然后引入进来，我们就不这样做了，直接修改`bin/catalina.sh`文件。

8.2.2 运行时数据区和内存结构

既然要对内存的大小做调整设置，你得认知一下jvm这块的内容，这里之前James老师的公开课和VIP课中讲过，当然你没听过也没关系，可以回头听一下，而且后面大白老师也会和大家讲这块的内容。

`结论`:接下来我也站在我的角度和大家做一个简单的分享，这有利于接下来我们tomcat的jvm调优。

`运行时数据区是一个规范，内存结构是一个实际的实现`

- 运行时数据区

`官网`:官网

（1）程序计数器The pc Register

JVM支持多线程同时执行，每一个线程都有自己的pc register，线程正在执行的方法叫做当前方法。如果是java代码，pc register中存放的就是当前正在执行的指令的地址，如果是c代码，则为空。

（2）Java虚拟机栈Java Virtual Machine Stacks

Java虚拟机栈是线程私有的，它的生命周期和线程相同。虚拟机栈描述的是Java方法执行的内存模型:每个方法在执行的同时都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直到执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

（3）堆Heap

Java堆是Java虚拟机所管理的内存中最大的一块。对是被所有线程共享的一块内存区域，在虚拟机启动时创建。次内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

Java对可以处于物理上不连续的内存空间中，只要逻辑上市连续的即可。

（4）方法区Method Area

方法区和Java堆一样，是各个线程共享的内存区域，也是在虚拟机启动时创建。它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做Non-Heap(非堆)，目的是与Java堆区分开来。

`jdk1.8中就是metaspace`

`jdk1.6或者1.7中就是perm space`

运行时常量池Runtime Constant Pool是方法区的一部分，Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池，用于存放编译时期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

（5）本地方法栈Native Method Stacks

本地方法栈和虚拟机栈锁发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈执行Java方法服务，而本地方法栈则为虚拟机使用到的native方法服务。

- 内存结构

上面对运行时数据区描述了很多，其实重点存储数据的是堆和方法区(非堆)，所以我们内存结构的设计也是着重从这两方面展开的。

一块是非堆区，一块是堆区。

堆区分为两大块，一个是Old区，一个是Young区。

Young区分为两大块，一个是Survival区（S0+S1），一块是Eden区。 Eden:S0:S1=8:1:1

S0和S1一样大，也可以叫From和To。

在同一个时间点上，S0和S1只能有一个区有数据，另外一个是空的。

8.2.3 垃圾回收算法

- 为什么需要学习垃圾回收算法?

Java是做自动内存管理的，自动垃圾回收。

- 如何确定一个对象是否是垃圾，从而确定是否需要回收?

（1）引用计数

对于某个对象而言，只要应用程序中持有该对象的引用，就说明该对象不是垃圾，如果一个对象没有任何指针对其引用，它就是垃圾。

`弊端`:AB相互持有引用，导致永远不能被回收。

（2）枚举根节点做可达性分析

`能作为根节点的`:类加载器、Thread、虚拟机栈的本地变量表、static成员、常量引用、本地方法栈的变量等。

- 常量的垃圾回收算法

`能够确定一个对象是垃圾之后，怎么回收？得要有对应的算法`

（1）标记清除

先标记所有需要回收的对象，然后统一回收。

`缺点`:效率不高，标记和清除两个过程的效率都不高，容易产生碎片，碎片太多会导致提前GC。

（2）复制

将内存按容量划分为大小相等的两块(S0和S1)，每次只使用其中一块。

当这块使用完了，就讲还存活的对象复制到另一块上，然后再把已经使用过的内存空间一次性清除掉【Young区此采用的是复制算法】

`优缺点`:实现简单，运行高效，但是空间利用率低。

（3）标记整理

标记需要回收的对象，然后让所有存活的对象移动到另外一端，直接清理掉端边界意外的内存。

- JVM中采用的是分代垃圾回收

换句话说，堆中的Old区和Young区采用的垃圾回收算法是不一样的。

（1）Young区：复制算法

（2）Old区：标记清除或标记整理

`对象在被分配之后，可能声明周期比较短，Young区复制效率比较高。`

`Old区对象存活时间比较长，复制来复制去没必要，不如做个标记。`

- 对象分配方式
  - 对象优先分配在Eden区
  - 大对象直接进入老年代，多大的对象称为大对象？可以通过JVM参数指定 -XX:PretenureSizeThreshold
  - 长期存活对象进入老年代

8.2.4 垃圾收集器

- 串行收集器Serial:Serial、Serial Old

  一个线程跑，停止，启动垃圾回收线程，回收完成，继续执行刚才暂停的线程。适用于内存比较小的嵌入式设备中。

- 并行收集器Parallel:Parallel Scavenge、Parallel Old，吞吐量优先

  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，适合科学计算、后台处理等弱交互场景

- 并发收集器Concurrent:CMS、G1，停顿时间优先

  用户线程和垃圾收集线程同时执行(但不一定是并行的，可能会交替执行)，垃圾收集线程在执行的时候不会停顿用户程序的运行。适合于对相应时间有要求的场景，比如Web。

- 吞吐量和停顿时间解释
  - 吞吐量:花在垃圾收集的时间和花在应用程序时间的占比
  - 停顿时间:垃圾收集器做垃圾回收终端应用执行的时间

  `小结：`评价一个垃圾回收器的好坏，其实调优的时候就是在观察者两个变量

- 开启垃圾收集器

```
 (1) 串行:    -XX: +UseSerialGC   -XX: +UseSerialOldGC      新老生代
 (2) 并行(吞吐量优先):
-XX: +UseParallelGC
-XX: +UseParallelOldGC
 (3) 并发收集器(响应时间优先)
CMS:    -XX: +UseConcMarkSweepGC
G1:     -XX: +UseG1GC
```

- Young区和Old区适用的垃圾回收器

jdk1.8中比较推荐使用G1垃圾回收器，性能比较高。

- 常用的G1 Collector

`jdk1.7开始使用，jdk1.8非常成熟，jdk1.9默认的垃圾收集器`

`要求`:>=6GB,停顿时间小于0.5秒

适用于新老生代

是否需要用G1的判断依据

（1）50%以上的堆被存活对象占用

（2）对象分配和晋升的速度变化非常大

（3）垃圾回收时间比较长

- 如何选择合适的垃圾回收器

（1）优先调整堆的大小让服务器自己来选择

（2）如果内存小于100M，使用串行收集器

（3）如果是单核，并且没有停顿时间要求，使用串行或JVM自己选

（4）如果允许停顿时间超过1秒，选择并行或JVM自己选

（5）如果响应时间最重要，并且不能超过1秒，使用并发收集器

8.2.5 两款GC日志分析工具

评价一个垃圾回收器的好坏：吞吐量和停顿时间

要想分析，得把GC日志打印出来才行，可以在tomcat中catalina.sh JAVA_OPTS配置相关参数

```
XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -
Xloggc:$CATALINA_HOME/logs/gc.log
```

然后重启tomcat，下载下来看看内容

- 在线:http://gceasy.io

  上述日志直接看比较费力，不妨借助工具，把gc.log下载到本地，然后上传到gceasy.io

  可以比较不同的垃圾回收器的日志情况

- GCViewer

8.2.6 内存模型和GC联系

```
Minor GC:新生代
Major GC:老年代
Full GC:新生代+老年代
```

- 一个对象的一辈子-概要

一般情况下,新创建的对象都会被分配到Eden区(一些大对象特殊处理),这些对象经过第一次Minor GC后,如果仍然存活,将会被移到Survivor区。对象在Survivor区中每熬过一次Minor GC,年龄就会增加1岁,当它的年龄增加到一定程度时,就会被移动到年老代中。

- 一个对象的一辈子-理论

在GC开始的时候,对象只会存在于Eden区和名为“From”的Survivor区,Survivor区“To”是空的。紧接着进行GC,Eden区中所有存活的对象都会被复制到“To”,而在“From”区中,仍存活的对象会根据他们的年龄值来决定去向。年龄达到一定值(年龄阈值,可以通过-XX:MaxTenuringThreshold来设置)的对象会被移动到年老代中,没有达到阈值的对象会被复制到“To”区域。经过这次GC后,Eden区和From区已经被清空。这个时候,“From”和“To”会交换他们的角色,也就是新的“To”就是上次GC前的“From”,新的“From”就是上次GC前的“To”。不管怎样,都会保证名为To的Survivor区域是空的。Minor GC会一直重复这样的过程,直到“To”区被填满,“To”区被填满之后,会将所有对象移动到年老代中。

- 一个对象的一辈子-案例

我是一个普通的Java对象,我出生在Eden区,在Eden区我还看到和我长的很像的小兄弟,我们在Eden区中玩了挺长时间。有一天Eden区中的人实在是太多了,我就被迫去了Survivor区的“From”区,自从去了Survivor区,我就开始漂了,有时候在Survivor的“From”区,有时候在Survivor的“To”区,居无定所。直到我18岁的时候,爸爸说我成人了,该去社会上闯闯了。于是我就去了年老代那边,年老代里,人很多,并且年龄都挺大的,我在这里也认识了很多人。在年老代里,我生活了20年(每次GC加一岁),然后被回收。

- 为什么会有Survival区

如果没有Survivor,Eden区每进行一次Minor GC,存活的对象就会被送到老年代。老年代很快被填满,触发Major GC(因为Major GC一般伴随着Minor GC,也可以看做触发了Full GC)。老年代的内存空间远大于新生代,进行一次Full GC消耗的时间比Minor GC长得多。你也许会问,执行时间长有什么坏处?频发的Full GC消耗的时间是非常可观的,这一点会影响大型程序的执行和响应速度,更不要说某些连接会因为超时发生连接错误了。
增加老年代空间　更多存活对象才能填满老年代。降低Full GC频率　随着老年代空间加大,一旦发生Full GC,执行所需要的时间更长
减少老年代空间　Full GC所需时间减少　老年代很快被存活对象填满,Full GC频率增加
Survivor的存在意义,就是减少被送到老年代的对象,进而减少Full GC的发生,Survivor的预筛选保证,只有经历16次Minor GC还能在新生代中存活的对象,才会被送到老年代。

- 为什么会有两个Survival区

设置两个Survivor区最大的好处就是解决了碎片化,下面我们来分析一下。

为什么一个Survivor区不行?第一部分中,我们知道了必须设置Survivor区。假设现在只有一个survivor区,我们来模拟一下流程:
刚刚新建的对象在Eden中,一旦Eden满了,触发一次Minor GC,Eden中的存活对象就会被移动到Survivor区。这样继续循环下去,下一次Eden满了的时候,问题来了,此时进行Minor GC,Eden和Survivor各有一些存活对象,如果此时把Eden区的存活对象硬放到Survivor区,很明显这两部分对象所占有的内存是不连续的,也就导致了内存碎片化。
永远有一个survivor space是空的,另一个非空的survivor space无碎片。

8.2.7 JVM常见参数

无论是设置内存大小还是选用不同的GC Collector都可以通过JVM参数的形式，所以我们有必要了解一下JVM参数相关的内容。

- 标准参数

```
-help
-server   -client
-version    -showversion
-cp     -classpath
```

- X参数

```
非标准参数，也就是在jvm各个版本中可能会变
-Xint       解释执行
-Xcomp       第一次使用就编译成本地代码
-Xmixed        混合模式，JVM自己来决定是否编译成本地代码
```

- XX参数

```
平时用的最多的参数类型
非标准化参数，相对不稳定，主要用于JVM调优和Debug
a.Boolean类型
格式：-XX:[+-]<name>  表示启用或者禁用name属性
比如：-XX:+UseConcMarkSweepGC     表示启用CMS类型的垃圾回收器
  -XX:+UseG1GC      表示启用CMS类型的垃圾回收器
b.非Boolean类型
格式：-XX<name>=<value>表示name属性的值是value
比如：-XX:MaxGCPauseMillis=500
```

- 特殊参数

```
-Xmx   -Xms     设置最大最小内存的
不是X参数，而是XX参数
-Xms等价于-XX:InitialHeapSize
-Xmx等价于-XX:MaxHeapSize
-Xss等价于-XX:ThreadStackSize
```

- 查看JVM运行时参数

得先知道当前的值是什么，然后才能设置调优

=表示默认值

:=表示被用户或JVM修改后的值

查看PID: jps -l，专门用来查看java进程的

```
jinfo   查看已经运行的jvm里面的参数值
jinfo -flag MaxHeapSize PID      查看最大内存
jinfo -flag UseG1GC PID      查看垃圾回收器
jinfo -flags PID     查看曾经赋过值的一些参数
```

- jstat查看JVM统计信息

（1）类装载

jstat -class PID 1000 10

PID进程ID，1000每个一秒钟，10输出10次

（2）垃圾收集

jstat -gc PID 1000 10

```
  S0C     S1C   S0U    S1U     EC        EU       OC         OU        GCT
2560.0  2560.0  0.0   2556.0 30720.0  29499.0   40960.0    11559.8   16512.0

  MC      MU     CCSC     CCSU    YGC   YGCT     FGC      FGCT
15968.7 1920.0  1795.4      3    0.038   0      0.000     0.038
```

8.2.8 内存溢出和优化

内存不够用主要分为两个方面：堆和非堆

所以这时候就要去手动设置堆或者非堆的大小，然后程序中不停使用相对应的区域，等待内存溢出。

关键是内存溢出之后，怎么得到溢出信息进行分析，有两种做法

- 参数设置自动

```
-XX:+HeapDumpOnOutOfMemoryError
-XX:HeapDumpPath=./
```

- jmap手动

```
查看当前进程id PID
jmap -dump:format=b,file=heap.hprof PID
jmap -heap PID    打印出堆内存相关的信息
```

> 当内存信息打印出来之后，发现看不懂，怎么办呢？得要有工具帮助我们看这块的信息，比如MAT

小结:这块可以适当增加内存的大小，这样防止内存溢出，减少垃圾回收的频率

8.2.9 GC调优

（1）查看目前VM使用的垃圾回收器

```
[root@pretty ~]# jinfo -flag UseParallelGC 6925
-XX:+UseParallelGC      --->发现使用了ParallelGC
[root@pretty ~]# jinfo -flag UseG1GC 6925
-XX:-UseG1GC            --->发现没有使用G1GC
```

（2）将垃圾回收器修改为G1

```
-XX:+UseG1GC
```

```
[root@pretty ~]# jinfo -flag UseG1GC 7158
-XX:+UseG1GC
```

（3）打印出日志详情信息和日志输出目录文件

```
PrintGCDetails:打印日志详情信息
PrintGCTimeStamps:输出GC的时间戳(以基准时间的形式)
```

```
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -
Xloggc:$CATALINA_HOME/logs/g1gc.log
```

（4）将日志用工具来分析，看相应的参数

8.2.10 JVM调优小结

内存大小设置——>dump出日志 使用MAT工具分析

垃圾收集器选择———>dump出GC日志 gceasy或者GCViewer

8.3 其他优化

- Connector

  配置压缩属性compression="500"，文件大于500bytes才会压缩

- 数据库优化

  减少对数据库访问等待的时间，可以从数据库的层面进行优化，或者加缓存等等各种方案。

- 开启浏览器缓存，nginx静态资源部署

9 嵌入式Tomcat主类寻找

9.1 maven

```
<dependency>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <version>2.0</version>
</dependency>
```

寻找:Tomcat7RunnerCli类，寻找main函数

9.2 springboot

org.springframework.boot.context.embedded.tomcat.EmbeddedServletContainerCustomizer

```
// 相当于 new TomcatContextCustomizer(){}
factory.addContextCustomizers((context) -> { // Lambda
    if (context instanceof StandardContext) {
        StandardContext standardContext = (StandardContext) context;
        // standardContext.setDefaultWebXml(); // 设置
    }
});
```