**1. 类的实例化顺序，比如父类静态数据，构造函数，字段，子类静态数据，构造函数，字段，他们的执行顺序**

1，父类中的static代码块，当前类的static

2，顺序执行父类的普通代码块

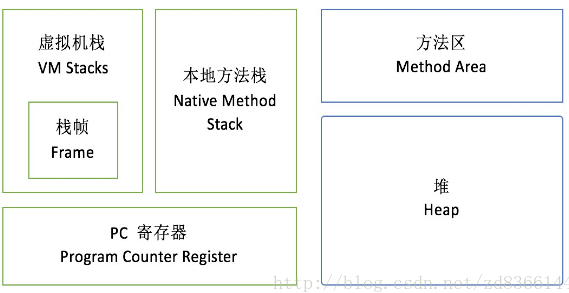
3，父类的构造函数

4，子类普通代码块

5，子类（当前类）的构造函数，按顺序执行。

6，子类方法的执行

**2. JVM内存分配**



**3.JVM垃圾回收机制，何时出发MinorGC，Full GC操作**

**当Eden区满时，触发Minor GC**

**新生代的垃圾收集**器命名为“**minor gc**”，**老生代**的GC命名为**”Full Gc 或者Major GC**”.其中用**System.gc()强制执行的是Full Gc**.

**Full GC触发条件：老年代空间不足，导致老年代空间不足的情况**

(1）调用System.gc时，系统建议执行Full GC，但是不必然执行

（2）老年代空间不足

（3）方法区空间不足

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

**5.jvm中一次完整的GC流程（从ygc到fgc）是怎么样的，重点讲讲对象如何晋升到老年代**

对象优先在新生代区中分配，若没有足够空间，Minor GC；

大对象（需要大量连续内存空间）直接进入老年态；长期存活的对象进入老年态。如果对象在新生代出生并经过第一次MGC后仍然存活，年龄+1，若年龄超过一定限制（15），则被晋升到老年态

**8.分析classloader、双亲委托机制**

ClassLoader：类加载器（class loader）用来加载 Java 类到 Java 虚拟机中。Java 源程序（.java 文件）在经过 Java 编译器编译之后就被转换成 Java 字节代码（.class 文件）。类加载器负责读取 Java 字节代码，并转换成 java.lang.Class 类的一个实例。

双亲委派机制：某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。

**13.jvm的常用参数**

堆设置，栈设置，垃圾回收器设置

**14.tomcat类加载流程**

①为什么tomcat要实现自己的类加载器

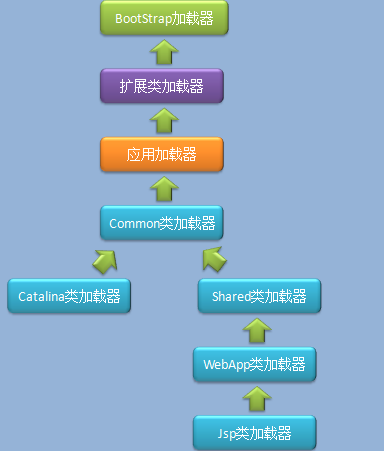
tomcat作为一个服务器，在它上面可以部署多个应用。默认情况下是使用应用加载器加载类的，但往往**部署的应用都会有多个jar依赖**，所以第一点**为了解决依赖问题**，必须保证**每个应用的类库独立**。 假设部署在tomcat上的两个应用依赖的数据库驱动版本有很大差异，这时如果应用类库不独立，那么使用应用类加载器就只会对一个版本的驱动进行加载，必然导致只能加载一个类库，这样另一个应用便无法启动。

           a)、要保证部署在tomcat上的**每个应用依赖的类库相互独立**，不受影响。

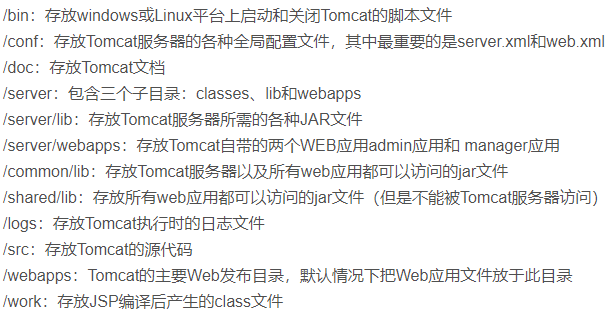
           b)、由于tomcat是采用java语言编写的，它自身也有类库依赖，为了安全考虑，**tomcat使用的类库要与部署的应用的类库相互独立**。

           c)、**有些类库tomcat与部署的应用可以共享，比如说servlet-api**

由于存在上述问题，tomcat实现了自己的类加载器，



其中蓝色的类加载器为tomcat自己实现的类加载器。



 Common类加载器：负责加载/common目录的类库，这儿存放的类库可被tomcat以及所有的应用使用。

         Catalina类加载器：负责加载/server目录的类库，只能被tomcat使用。

         Shared类加载器：负载加载/shared目录的类库，可被所有的web应用使用，但tomcat不可使用。

--------------6.x以上的tomcat已经将上述两个加载器用commonclassloader来代替

         WebApp类加载器：负载加载单个Web应用下classes目录以及lib目录的类库，只能当前应用使用。

         Jsp类加载器：负责加载Jsp，每一个Jsp文件都对应一个Jsp加载器。

         Tomcat运行期间，Webapp类加载器与Jsp类加载器都是多个的。所以应用之间可以互不影响的启动。

启动流程：通过bootstrap加载器的main方法进行启动

第一步：判断bootstrap是否已经初始化了？如果没有则进行初始化；

设置catalina.base和catalina.home路径；初始化需要的类加载器；利用反射机制执行setParentClassLoader方法设置Tomcat类加载体系的父类加载器；

第二步：调用Bootstrap的load()来加载server.xml文件，最终还是使用catalina类的load方法，对server.xml文件进行载入和解析

第三步：调用bootstrap类的start()方法启动Tomcat；

依然使用catalina类的start方法；判断catalina的server实例是否初始化成功了，没有的话继续调用catalina的load方法；成功的话，调用server的start方法，server的启动又会带动service的启动等等，一级一级使得整个组件都启动起来。最后，调用catalina自身类的await方法循环等待shutdown命令；该方法一直等待socket连接，并接受shutdown命令，如果没有收到就会一直阻塞在await方法中的serverSocket.accept方法；一旦接受到shutdown命令，就会结束await方法，执行start方法里面的stop方法；内部就是对于shutdown钩子的移除，以及调用stop和destroy方法分别关闭服务和释放资源。

**18．为什么要采用双亲委托**

1. 当前ClassLoader首先从自己已经加载的类中查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。

每个类加载器都有自己的加载缓存，当一个类被加载了以后就会放入缓存，等下次加载的时候就可以直接返回了。

2.  当前classLoader的缓存中没有找到被加载的类的时候，委托父类加载器去加载，父类加载器采用同样的策略，首先查看自己的缓存，然后委托父类的父类去加载，一直到bootstrp ClassLoader.

3.  当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中，以便下次有加载请求的时候直接返回。

防止内存中出现多份同样的字节码

假设如果类A要使用扩展类加载器的一个字节码C

①如果不用委托，自己加载，那么类A就会加载一份C字节码，这就出现两份System字节码；如果采用父委托，只需要加载一次即可

② java核心api中定义类型不会被随意替换，假设通过网络传递一个名为java.lang.Integer的类，通过双亲委托模式传递到启动类加载器，而启动类加载器在核心Java API发现这个名字的类，发现该类已被加载，并不会重新加载网络传递的过来的java.lang.Integer。

**19. JVM内存泄漏**

当被分配的对象可达但已无用，并且没有对其进行回收，那么就会引起内存泄漏

