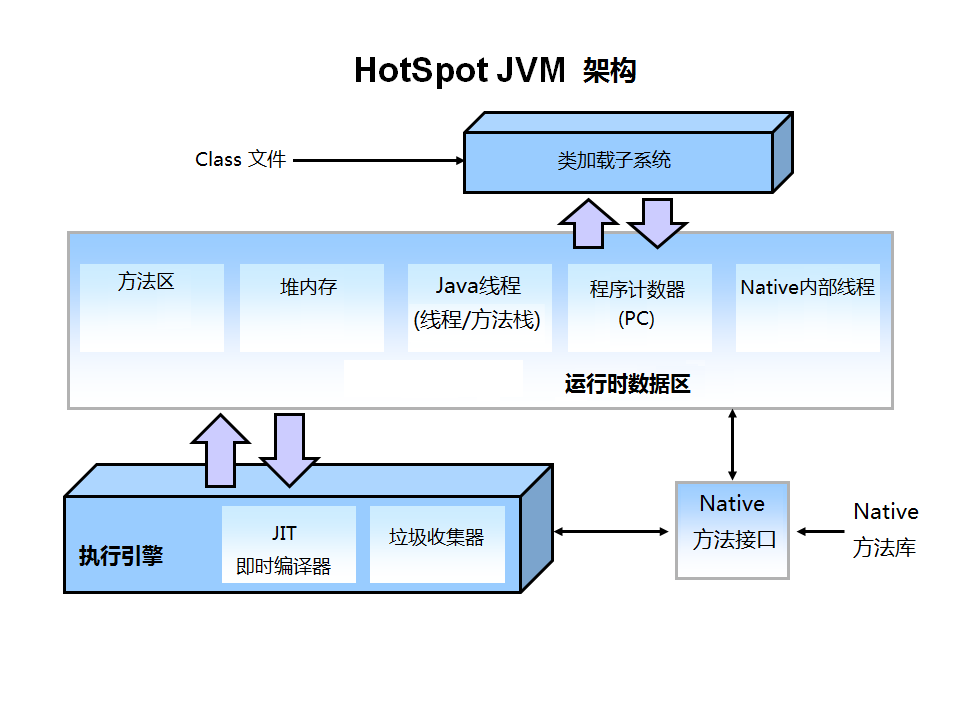
使用最广jvm sun hotspot(c++) vm



运行时数据区

1. 程序计数器：线程所执行的字节码的行号指示器（**线程私有**）
2. Java虚拟机栈：**线程私有，生命周期与线程相同**，每个方法在执行的同时会在其中创建**栈帧**存储：**局部变量表**、**操作数**（7 **+** 5）、动态链表、方法出口（调用压栈）。其中局部变量表包括存放**基本数据类型变量、对象引用、和returnaddress，局部变量表**（栈不是）所需的内存空间在**编译期间分配完成，其空间不会在运行期间改变。每个方法的调用对应一个栈帧的压栈出栈。**
3. 本地方法栈，与2类似，不过是提供给native（jni）方法服务的
4. Java堆：所有线程共享，几乎所有对象和全部数组都要在堆上分配内存，垃圾管理器主要管理的也是堆
5. 方法区：**存储类信息、常量、静态变量（如果是对象，其引用存储在方法区，对象存储在堆）、和jit代码**（堆的逻辑部分）
6. 直接内存

Class文件的常量池会存放**字面量和符号引用**（部分编译器解析变为直接引用），这部分内容在类加载后进入**方法区中**的**运行常量池（运行期间也会有常量）**的（**各个线程共享**）

（符号引用是一个字符串，它给出了被引用的内容的名字­­————如**类名，方法名**，在类加载解析）（运行期也可能将常量放入池中）

（局部变量，基础类型和reference，reference引用的对象不是局部变量）

“方法区”是JVM规范所描述的抽象概念。在实际的JVM实现中，它不一定是由单一的特殊区域所实现。

常量不是指Final，被final修饰不影响存储。

**成员变量**

类的成员变量在不同对象中各不相同，都有自己的存储空间(成员变量在**堆中的对象中**)，基本类型和引用类型的成员变量都在这个对象的空间中，作为一个整体存储在堆。

**对象的创建：**

1. new之后根据类名去常量池定位到符号引用，并检查该引用是否被加载、解析初始化过、未加载则加载
2. 分配内存（空闲列表或指针碰撞）
3. 内存空间初始化为0
4. 虚拟机对对象设置（如对象是哪个类的实例），将信息存放在**对象**的**objectheader**里
5. Init（构造方法初始化）

Oom：（out of memory）

堆溢出：部分对象生命周期过长，导致它引用的对象也无法释放造成恶性拓展。

拓展栈时无法申请到足够空间：创建过多线程

方法区和运行时常量池溢出

Sof：（stack over flow）

栈深度大于虚拟机允许最大深度

每个**方法在执行**的同时都会创建一个**栈帧**，每一个方法从调用直至执行完成的过程，就对应着一个**栈帧在虚拟机栈中入栈到出栈**的过程，错误**递归**则是一直压栈，超过栈的最大速度

**javac与jit**

Javac是编译成字节码，运行时还要一句一句转义，jit是即时编译成机器码

线程状态

**垃圾回收**

一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链相连（用图论的话来说，就是从GC Roots到这个对象不可达）时，则证明此对象是不可用的。



如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行finalize（）方法。当对象没有覆盖finalize（）方法，或者finalize（）方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

如果这个对象被判定为有必要执行finalize（）方法，那么这个对象将会放置在一个叫做F-Queue的队列之中，并在稍后由一个由虚拟机自动建立的、低优先级的Finalizer线程去执行它。如果对象要在finalize（）中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（this关键字）赋值给某个类变量或者对象的成员变量，那在第二次标记时它将被移除出“即将回收”的集合；如果对象这时候还没有逃脱，那基本上它就真的被回收了（和不需要执行finalize的一起，finalize只会执行一次只能逃一次）。

**Java类的加载、连接、初始化都是在运行区间完成的。（类加载的操作基本都是在方法区）**

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）7个阶段。其中验证、准备、解析3个部分统称为连接（Linking）



对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。

（主要是加载二进制流，生成class对象，验证语义）

**加载：**

1）通过一个类的全限定名来获取定义此类的**二进制字节流**。

2）将这个字节流所代表的静态存储结构转化为**方法区**的运行时数据结构。

3）在内存中生成一个代表这个类的**java.lang.Class对象**，作为方法区这个类的各种数据的访问入口。

对于数组类而言，情况就有所不同，数组类本身不通过类加载器创建，它是由Java虚拟机直接创建的。

**(（2）二进制字节流按照虚拟机所需的存储格式存储在方法区中，**

**（3）java.lang.Class对象存储在内存中（hotspot在方法区中），它将作为程序访问方法区中数据的外部接口)**

**验证：**

验证是连接阶段的第一步，验证是虚拟机对自身保护的一项重要工作

1.文件格式验证

2.元数据验证（对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语言规范的要求）

3.字节码验证 （通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的）

4.符号引用验证（符号引用中通过字符串描述的全限定名是否能找到对应的类及其方法字段）

**准备：**

准备阶段是正式为**类变量**（非对象变量--static）**分配内存**并设置类变量**初始值**的阶段

public static int value=123； 那变量value在准备阶段过后的初始值为0而不是123

**解析：**

解析阶段是虚拟机将**常量池内**的**符号引用替换为直接引用（方法：静态方法和私有方法，变量：static,private,final。编译器可知，运行期不可变）**的过程

符号引用（Symbolic References）：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。

直接引用（Direct References）：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。

**虚拟机规范之中并未规定解析阶段发生的具体时间**

**初始化：**

到了初始化阶段，才真正开始执行类中定义的Java程序代码（或者说是字节码）。在准备阶段，变量已经赋过一次系统要求的初始值，初始化阶段是执行类构造器＜clinit＞（）方法的过程

＜clinit＞（）方法是由编译器自动收集类中的所有**类变量的赋值动作**（**不包括方法与构造函数中**）和静态语句块（**static{}块**）中的语句合并产生的

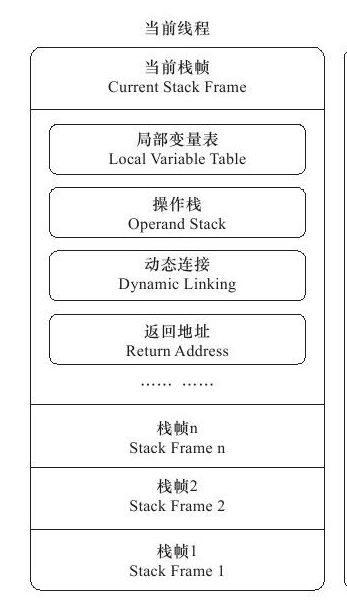
**栈帧**

栈帧（Stack Frame）是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈（Virtual Machine Stack）的栈元素。

每一个栈帧都包括了**局部变量表、操作数栈、动态连接、方法返回地址**和一些额外的附加信息。

一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而**仅仅取决于具体的虚拟机实现**。

编译是没有连接过程的，方法调用在class文件里面只是存储符号引用。



**局部变量表**

最小单位为slot，每个slot都应该能存放非long，double的基础类型(>=32位)

为了效率，一般slot为32位，而long和double为64位，故存储在两个slot中，非原子，但是不允许对单独访问slot单独一个，因此不要对long、double声明volidate

**方法调用**：

解析（静态解析：编译时确定，符号引用转直接引用—静态方法和私有方法（final方法解析阶段无法确认））。

分派（动态、静态）

调用（找到方法运行入口）

Java虚拟机里共提供了四条方法调用字节指令，分别是：

* invokestatic：调用静态方法。
* invokespecial：调用实例构造器<init>方法、私有方法和父类方法。
* invokevirtual：调用所有的虚方法。
* invokeinterface：调用接口方法，会在运行时再确定一个实现此接口的对象。

只要能被invokestatic和invokespecial指令调用的方法，都可以在解析阶段确定唯一的调用版本（静态方法、私有方法、实例构造器和父类方法），它们**在类加载时就会把符号引用解析为该方法的直接引用**。这些方法可以称为非虚方法（还包括final方法），其他方法就称为虚方法（final方法除外）。虽然调用final方法使用的是invokevirtual指令，但是由于它无法覆盖，没有其他版本，所以也无需对方发接收者进行多态选择。Java语言规范中明确说明了final方法是一种非虚方法。

**静态分派（重载）根据引用类型**

public void sayHello（Human guy）{

System.out.println（"hello,guy！"）；}

public void sayHello（Man guy）{

System.out.println（"hello,gentleman！"）；}

public void sayHello（Woman guy）{

System.out.println（"hello,lady！"）；}

Human man=new Man（）；

Human woman=new Woman（）；

StaticDispatch sr=new StaticDispatch（）；

sr.sayHello（man）；

sr.sayHello（woman）；

**调用的都是human的方法**

“Human”称为变量的**静态类型**（Static Type）编译期可知

后面的“Man”则称为变量的**实际类型**（Actual Type）运行期确认

**重载时**是通过参数的静态类型（引用类型）而不是实际类型作为判定依据的，虚拟机（准确地说是编译器）在重载时是通过参数的静态类型而不是实际类型作为判定依据的。并且静态类型是编译期可知的，因此，在编译阶段，**Javac编译器会根据参数的静态类型决定使用哪个重载版本，**所以选择了sayHello（Human）作为调用目标

**动态分派（覆盖override）根据堆中实际存放类型**

static abstract class Human{

protected abstract void sayHello（）；}

static class Man extends Human{

@Override

protected void sayHello（）{

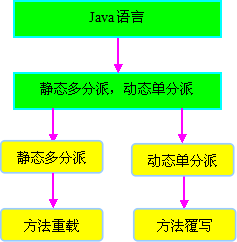
System.out.println（"man say hello"）；}

Human man=new Man（）；

man.sayHello（）

**调用的是man的方法**

因为是根据引用去寻找实际对象，在栈中，栈顶存储的是对象的引用



编译阶段编译器的选择过程，即静态分派过程。这时候选择目标方法的依据有两点：一是方法的接受者（即调用者）的静态类型，二是方法参数类型是Eat还是Drink。因为是根据两个宗量进行选择，所以Java语言的静态分派属于多分派类型。

运行阶段虚拟机的选择，即动态分派过程。由于编译期已经了确定了目标方法的参数类型（编译期根据参数的静态类型进行静态分派），因此唯一可以影响到虚拟机选择的因素只有此方法的接受者的实际类型是Father还是Child。因为只有一个宗量作为选择依据，所以Java语言的动态分派属于单分派类型。

**invokevirtual指令的动态查找的步骤：**

1. 找到操作数栈顶的第一个元素所指向的**对象的实际类型**，记作C。
2. 如果在类型C中找到与常量中的描述符和简单名称都相符的方法，则进行访问**权限校验**，如果通过则返回这个方法的直接引用，查找过程结束；如果不通过
3. 否则，按照继承关系从下往上依次对C的各个父类进行第2步的搜索和验证过程。
4. 如果始终没有找到合适的方法，则抛出java.lang.AbstractMethodError异常。

核心：根据引用找对象的实际类型，然后得到直接引用，再去调用方法。

**解析和分派并不是二选一，是在不同层面进行筛选。**

**指令介绍**

iconst\_1 //把常量1压入栈

iconst\_1

iadd //把栈顶两个元素弹出

istore\_0 //把栈顶的值存放在局部变量表第0个slot

**class类文件**

**理解class类文件要与存储区区分开，此是代码编译的结果，而不是运行时的数据区域**

**Class类文件的结构：**

**魔数与Class文件的版本**

**常量池 （这部分文件在类加载后进入方法去的运行时常量池存放）**

（常量池中主要存放两大类常量：**字面量（Literal）和符号引用（Symbolic References）**。**字面量**比较接近于Java语言层面的常量概念，如**文本字符串**等。而符号引用则属于编译原理方面的概念，包括了下面三类常量： 类和接口的全限定名（Fully Qualified Name） 字段的名称和描述符（Descriptor） 方法的名称和描述符）

**访问标志 （类的）**

**类索引、父类索引与接口索引集合**

**字段表集合 字段修饰符、名称、类型..**

**方法表集合 （方法的访问标志、名称索引、属性表集合…）**

**属性表集合**

在Class文件、字段表、方法表都可以携带自己的属性表集合，以用于描述某些场景专有的信息，Java程序方法体中的代码经过Javac编译器处理后，最终变为**字节码指令存储在Code属性内**。Code属性出现在方法表的属性集合之中，但并非所有的方法表都必须存在这个属性，譬如接口或者抽象类中的方法就不存在Code属性

**编译（前端编译：如javac）**

编译过程大致可以分为3个过程，分别是：

解析与填充符号表过程。

(符号表所登记的内容将用于语义检查和产生中间代码)

插入式注解处理器的注解处理过程。

分析与字节码生成过程。

**后期编译（jit）**

解释器与编译器两者各有优势：当程序需要迅速启动和执行的时候，解释器可以首先发挥作用，省去编译的时间，立即执行。在程序运行后，随着时间的推移，编译器逐渐发挥作用，把越来越多的代码编译成本地代码之后，可以获取更高的执行效率。

**语法糖与泛型擦除**

语法糖：在计算机语言中添加的某种语法，这种语法对语言的功能并没有影响

Java语言中的泛型，它只在程序源码中存在，在编译后的字节码文件中，就已经替换为原来的原生类型（Raw Type，也称为裸类型）了，并且在相应的地方插入了**强制转型代码**，因此，对于运行期的Java语言来说，ArrayList＜int＞与ArrayList＜String＞就是同一个类，所以泛型技术实际上是Java语言的一颗语法糖，Java语言中的泛型实现方法称为类型擦除，基于这种方法实现的泛型称为伪泛型。

**工作内存与主内存**

将运算需要使用到的数据复制到缓存中，让运算能快速进行，当运算结束后再从缓存同步回内存之中，这样处理器就无须等待缓慢的内存读写了



Java内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成



**线程实现方式**

实现线程主要有3种方式：使用内核线程实现、使用用户线程实现和使用用户线程加轻量级进程混合实现。

线程模型替换为基于操作系统原生线程模型来实现。因此，在目前的JDK版本中，操作系统支持怎样的线程模型，在很大程度上决定了Java虚拟机的线程是怎样映射的

**Java内存模型（jmm）**

**主内存+每个线程有着自己的工作内存**

**主内存与工作内存同步方式**

**Lock/unlock**

**Use assign**

**Read(from内存) load(强绑定)**

**Store(to工作内存) write（强绑定）**

**Gc算法**

“**标记-清除**”（Mark-Sweep）算法

如同它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。(**问题：产生碎片**)

**复制**：

为了解决效率问题，一种称为“复制”（Copying）的收集算法出现了。将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor。当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8:1，也就是每次新生代中可用内存空间为整个新生代容量的90%（80%+10%），只有10%的内存会被“浪费”。**（问题：如果对象都存活不能使用—老年代）**

Gc都要从gc root进行可达性分析，整个分析期间不可以出现对象引用关系变化。故必须停顿所有线程。**Stop the world。**

**标记整理**

标记操作和“标记-清除”算法一致，后续操作不只是直接清理对象，而是在清理无用对象完成后让所有存活的对象都向一端移动，并更新引用其对象的指针。主要缺点：在标记-清除的基础上还需进行对象的移动，成本相对较高，好处则是不会产生内存碎片

**分代收集**

把堆分为新生代老生代，不同代采用不同的收集算法。

老年代：存活率高 (标记整理)

新生代：死去的多 (复制)

分代办法：每次gc，没被回收age+1.jvm配置分代

Jvm还可配置大对象直接进入老年代（避免eden survior大量内存复制/新生代复制策略）

收集器典型：G1,CMS,SERIAL

分代收集

基于“标记-整理”，两个region之间是基于“复制”

低停顿

介绍：化整为零，将java堆分为多个region,并保留新生代老生代概念（但不物理隔离）

优先回收价值大的region

Region直接有互相引用，要单独回收每一个需要有一个remembered set记录对其他region的引用

步骤：

初始标记（标记gcroot直接可达对象，停顿）

并发标记（用户程序继续运行，可达性分析）

最终标记（根据remebered set修改标记）

筛选回收（停顿用户筛选）

对象创建时如何根据直接引用找到类信息

从年轻代空间（包括 Eden 和 Survivor 区域）回收内存被称为 Minor GC。Major GC 是清理老年代。 Full GC 是清理整个堆空间—包括年轻代和老年代。

**类加载器：**

通过一个类的**全限定名来获取描述此类的二进制字节流**”这个动作放**到Java虚拟机外部**去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为“类加载器”。

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在Java虚拟机中的唯一性

从Java虚拟机的角度来讲，只存在两种不同的类加载器：一种是启动类加载器（Bootstrap ClassLoader），这个类加载器使用C++语言实现，是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都由Java语言实现，独立于虚拟机外部，并且**全都继承自抽象类java.lang.ClassLoader**。

绝大部分Java程序都会使用到以下3种系统提供的类加载器

启动类加载器（Bootstrap ClassLoader）：

这个类将器负责将存放在＜JAVA\_HOME＞\lib目录中的

扩展类加载器（Extension ClassLoader）：

这个加载器由sun.misc.Launcher $ExtClassLoader实现，它负责加载＜JAVA\_HOME＞\lib\ext目录中的。开发者可以直接使用扩展类加载器。

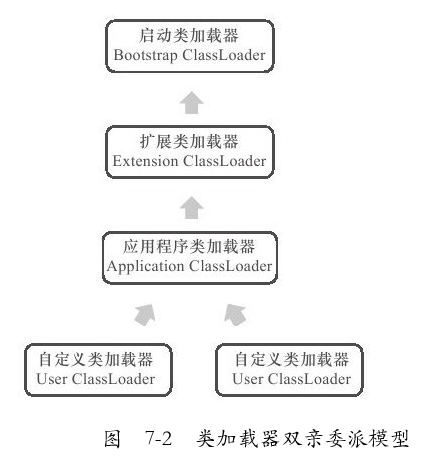
应用程序类加载器（Application ClassLoader）：

这个类加载器由sun.misc.Launcher $App-ClassLoader实现。开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一**般情况下这个就是程序中默认的类加载器**。

双亲委派模型（图7-4）要求除了顶层的启动类加载器外，**其余的类加载器都应当有自己的父类加载器**。这里类加载器之间的父子关系一般不会以继承（Inheritance）的关系来实现，而是都使用组合（Composition）关系来复用父加载器的代码。

双亲委派模型的工作过程是：如果一个类加载器收到了类加载的请求，它**首先**不会自己去尝试加载这个类，而是**把这个请求委派给父类加载器去完成**，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，**只有当父加载器反馈自己无法完成这个加载请求**（它的**搜索范围中没有**找到所需的类）时，**子加载器才会尝试自己去加载**。

**好处**就是Java类随着它的类加载器一起具备了一种**带有优先级的层次关系**。如java.lang.Object，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的**启动类加载器**进行加载，因此Object类在程序的各种类加载器环境中都是**同一个类**。



双亲委派代码：

protected synchronized Class＜?＞loadClass（String name,boolean resolve）throws ClassNotFoundException

{

//首先，检查请求的类是否已经被加载过了

Class c=findLoadedClass（name）；

if（c==null）{

try{

if（parent！=null）{

c=parent.loadClass（name,false）；

}else{

c=findBootstrapClassOrNull（name）；

}

}catch（ClassNotFoundException e）{

//如果父类加载器抛出ClassNotFoundException

//说明父类加载器无法完成加载请求

}

if（c==null）{

//在父类加载器无法加载的时候

//再调用本身的findClass方法来进行类加载

c=findClass（name）；

}

}

if（resolve）{

resolveClass（c）；

}

return c；

}

破坏双亲委派模型

原因：

1基础类用回用户的代码

2.用户对程序动态性的追求而导致的

“动态性”指的是：代码热替换（HotSwap）、模块热部署（Hot Deployment）等（不停止服务重新加载类）。OSGi实现模块化热部署的关键则是它自定义的类加载器机制的实现。每一个程序模块（OSGi中称为Bundle）都有一个自己的类加载器，**当需要更换一个Bundle时，就把Bundle连同类加载器一起换掉以实现代码的热替换**。

1）将以java.\*开头的类委派给父类加载器加载。

2）否则，将委派列表名单内的类委派给父类加载器加载。

3）否则，将Import列表中的类委派给Export这个类的Bundle的类加载器加载。

4）否则，查找当前Bundle的ClassPath，使用自己的类加载器加载。

5）否则，查找类是否在自己的Fragment Bundle中，如果在，则委派给Fragment Bundle的类加载器加载。

6）否则，查找Dynamic Import列表的Bundle，委派给对应Bundle的类加载器加载。

7）否则，类查找失败。

Tomcat与Apache HTTP Server和Nginx区别：

另外两本身**不支持生成动态页面**，但它们可以**通过其他模块来支持（例如通过Shell、PHP、Python脚本程序来动态生成内容）**。如果想要使用Java程序来动态生成资源内容，使用这一类HTTP服务器很难做到。Java Servlet技术以及衍生的Java Server Pages技术可以让Java程序也具有处理HTTP请求并且返回内容（由程序动态控制）的能力，Tomcat正是支持运行Servlet/JSP应用程序的容器（Container）。（tomcat是运行动态页面的容器，always running）

虽然Tomcat也可以认为是HTTP服务器，但通常它仍然会和Nginx配合在一起使用.

 **动静态资源分离**——运用Nginx的反向代理功能分发请求：所有动态资源的请求交给Tomcat，而静态资源的请求（例如图片、视频、CSS、JavaScript文件等）则直接由Nginx返回到浏览器，这样能大大减轻Tomcat的压力。

 负载均衡，当业务压力增大时，可能一个Tomcat的实例不足以处理，那么这时可以启动多个Tomcat实例进行水平扩展，而**Nginx的负载均衡功能**可以把请求通过算法分发到各个不同的实例进行处理  
Tomcat运行在JVM之上，它和HTTP服务器一样，绑定IP地址并监听TCP端口，同时还包含以下指责：

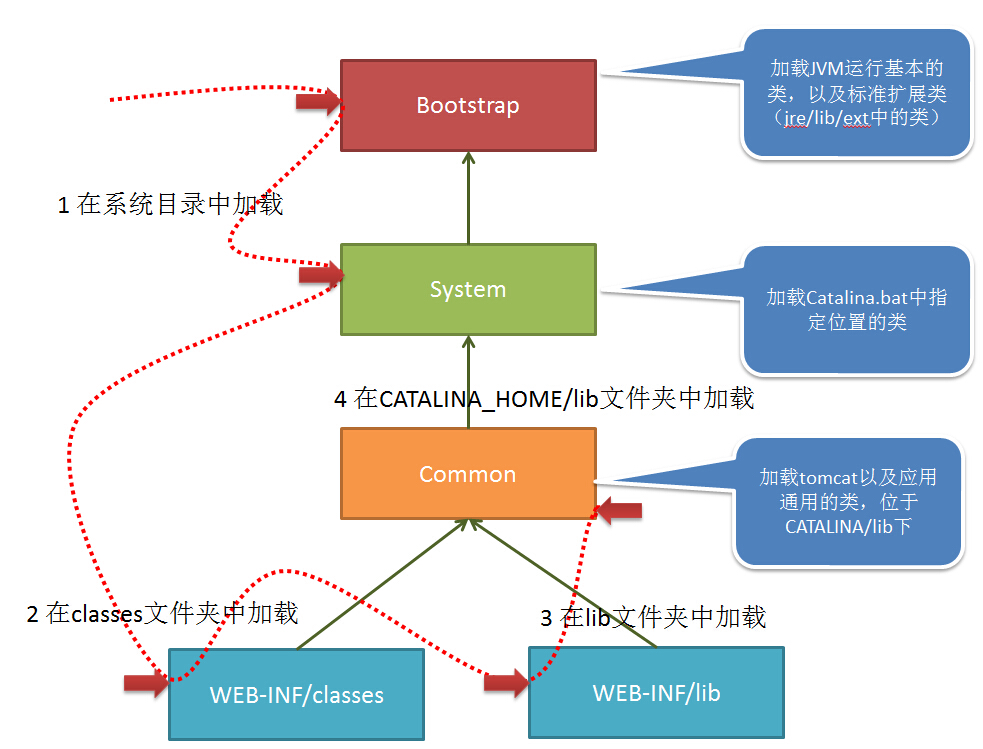
1.管理Servlet程序的生命周期

2.将URL映射到指定的Servlet进行处理

3.与Servlet程序合作处理HTTP请求,根据HTTP请求生成HttpServletRequest对象并传递给Servlet进行处理，将Servlet的HttpServletResponse对象生成的内容返回给浏览器

Tomcat容器运行时本身就是个大进程，通过线程调度生成request发送给servlet，接收response返回给client。**maxConnections决定请求连接量。**

Tomcat模型：



每个应用在部署后，都会创建一个唯一的类加载器。该类加载器会加载位于WEB-INF/lib下的jar文件中的class和WEB-INF/classes下的class文件。

当应用需要到某个类时，则会按照下面的顺序进行类加载：

1 使用**bootstrap**引导类加载器加载

2 使用**system**系统类加载器加载

3 使用应用类加载器在**WEB-INF/classes**中加载

4 使用应用类加载器在**WEB-INF/lib**中加载

5 使用common类加载器在**CATALINA\_HOME/lib**中加载

<https://stackoverflow.com/questions/5474765/order-of-loading-jar-files-from-lib-directory>

能不能自己写个类叫java.lang.System？

答案：通常不可以，但可以采取另类方法达到这个需求。

解释：为了不让我们写System类，类加载采用委托机制，这样可以保证爸爸们优先，爸爸们能找到的类，儿子就没有机会加载。而System类是Bootstrap加载器加载的，就算自己重写，也总是使用Java系统提供的System，自己写的System类根本没有机会得到加载。 但是，我们可以自己定义一个类加载器来达到这个目的，为了避免双亲委托机制，这个类加载器也必须是特殊的。由于系统自带的三个类加载器都加载特定目录下的类，如果我们自己的类加载器放在一个特殊的目录，那么系统的加载器就无法加载，也就是最终还是由我们自己的加载器加载。

**GC调优**

1. 产生一大堆大对象，（如读入文档生成序列化对象）。堆又非常大，导致full gc频繁停止时间久

解答：与其使用64位大内存，不如使用若干个32位虚拟机建立逻辑集群。而且不应该有成批时间长的大对象生成。

1. 阻塞型io服务者和调用者处理速度不对称，导致socket越来越多导致崩溃

解答：生产者消费者。

1. 采用复制算法对象却寿命过长导致复制成本高。

解答：调低进入老年代的minor gc次数

（选择合适的收集器）

**线程调度**

协同式调度和抢占式调度

协同式：运行完了再通知系统切换

抢占式：系统分配时间（可以设置优先级）