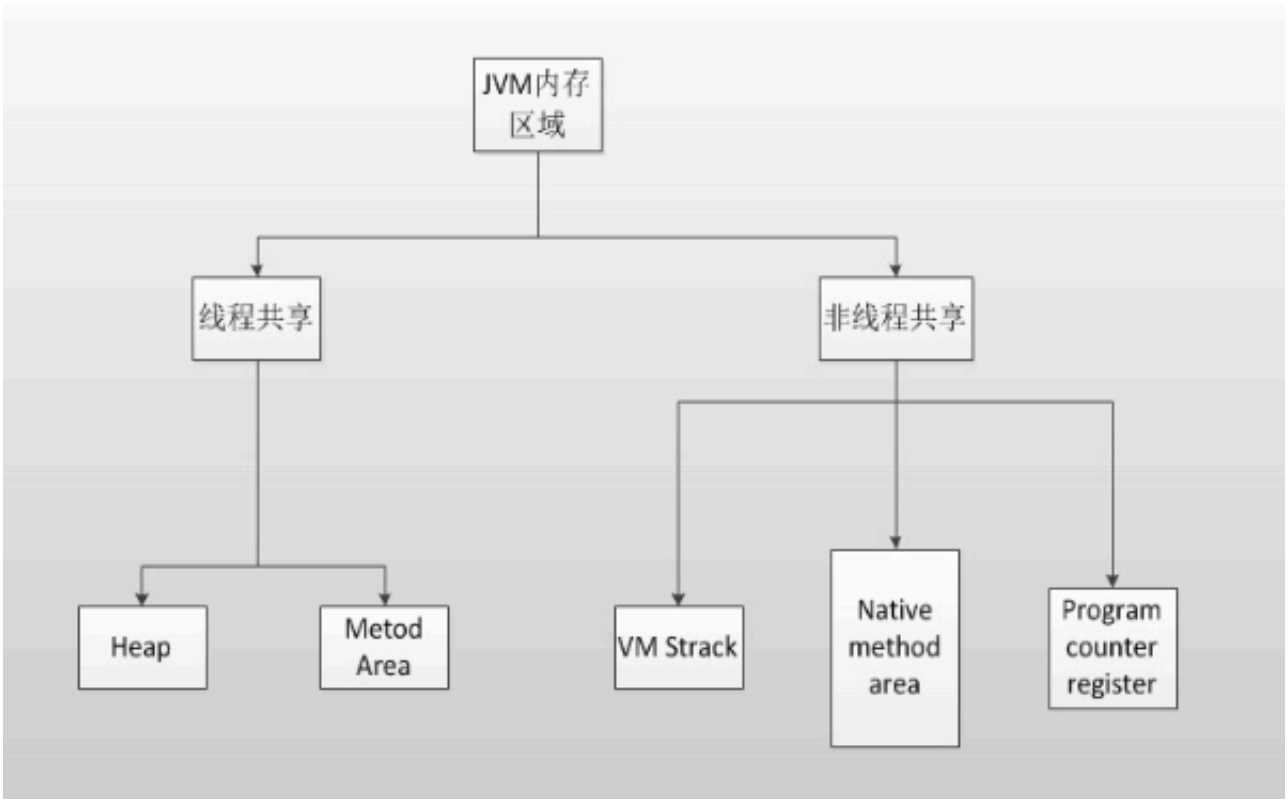
JVM内存模型



方法区(Method Area)、堆(Heap)
程序计数器(ProgramCounter Register)、Java栈(VM Stack)、本地方法栈(Native MethodStack)

JVM 将内存区域划分为 Method Area（Non–Heap）（方法区）,Heap（堆）,Program Counter Register（程序计数器）, VM Stack（虚拟机栈，也有翻译成JAVA 方法栈的）,Native Method Stack （本地方法栈），其中Method Area和Heap是线程共享的，VM Stack，Native Method Stack 和Program Counter Register是非线程共享的。

JVM初始运行的时候都会分配好Method Area（方法区）和Heap（堆），而JVM 每遇到一个线程，就为其分配一个Program Counter Register（程序计数器）, VM Stack（虚拟机栈）和Native Method Stack （本地方法栈），当线程终止时，三者（虚拟机栈，本地方法栈和程序计数器）所占用的内存空间也会被释放掉。非线程共享的那三个区域的生命周期与所属线程相同，而线程共享的区域与JAVA程序运行的生命周期相同，所以这也是系统垃圾回收的场所只发生在线程共享的区域（实际上对大部分虚拟机来说知发生在Heap上）的原因。

1. 程序计数器
   程序计数器是一块较小的内存区域，作用可以看做是当前线程执行的字节码的位置指示器。分支、循环、跳转、异常处理和线程恢复等基础功能都需要依赖这个计算器来完成
2. VM Strack
   先来了解下JAVA指令的构成：
   JAVA指令由 操作码 （方法本身）和 操作数 （方法内部变量） 组成。
   1. 方法本身是指令的操作码部分，保存在Stack中；
   2. 方法内部变量（局部变量）作为指令的操作数部分，跟在指令的操作码之后，保存在Stack中（实际上是简单类型（int,byte,short 等保存在Stack中），对象类型在Stack中保存地址，在Heap 中保存值；
   3. 虚拟机栈也叫栈内存，是在线程创建时创建，它的生命期是跟随线程的生命期，线程结束栈内存也就释放，对于栈来说不存在垃圾回收问题，只要线程一结束，该栈就 Over，所以不存在垃圾回收。也有一些资料翻译成JAVA方法栈，大概是因为它所描述的是java方法执行的内存模型，每个方法执行的同时创建帧栈（Strack Frame）用于存储局部变量表（包含了对应的方法参数和局部变量），操作栈（Operand Stack，记录出栈、入栈的操作），动态链接、方法出口等信息，每个方法被调用直到执行完毕的过程，对应这帧栈在虚拟机栈的入栈和出栈的过程。
   4. 局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象的引用（reference类型，不等同于对象本身，根据不同的虚拟机实现，可能是一个指向对象起始地址的引用指针，也可能是一个代表对象的句柄或者其他与对象相关的位置）和 returnAdress类型（指向下一条字节码指令的地址）。局部变量表所需的内存空间在编译期间完成分配，在方法在运行之前，该局部变量表所需要的内存空间是固定的，运行期间也不会改变。
3. Heap
   Heap（堆）是JVM的内存数据区。Heap 的管理很复杂，是被所有线程共享的内存区域，在JVM启动时候创建，专门用来保存对象的实例。在Heap 中分配一定的内存来保存对象实例，实际上也只是保存对象实例的属性值，属性的类型和对象本身的类型标记等，并不保存对象的方法（以帧栈的形式保存在Stack中）。而对象实例在Heap 中分配好以后，需要在Stack中保存一个4字节的Heap 内存地址，用来定位该对象实例在Heap 中的位置，便于找到该对象实例，是垃圾回收的主要场所。java堆处于物理不连续的内存空间中，只要逻辑上连续即可。
4. Method Area
   Object Class Data(加载类的类定义数据) 是存储在方法区的。除此之外，常量、静态变量、JIT(即时编译器)编译后的代码也都在方法区。正因为方法区所存储的数据与堆有一种类比关系，

所以它还被称为 Non-Heap。方法区也可以是内存不连续的区域组成的，并且可设置为固定大小，也可以设置为可扩展的，这点与堆一样。
垃圾回收在这个区域会比较少出现，这个区域内存回收的目的主要针对常量池的回收和类的卸载。

5. 运行时常量池（Runtime Constant Pool）
   方法区内部有一个非常重要的区域，叫做运行时常量池（Runtime Constant Pool，简称RCP）。在字节码文件（Class文件）中，除了有类的版本、字段、方法、接口等先关信息描述外，还有常量池（Constant Pool Table）信息，用于存储编译器产生的字面量和符号引用。这部分内容在类被加载后，都会存储到方法区中的RCP。值得注意的是，运行时产生的新常量也可以被放入常量池中，比如 String 类中的 intern() 方法产生的常量。
   常量池就是这个类型用到的常量的一个有序集合。包括直接常量(基本类型，String)和对其他类型、方法、字段的符号引用.例如：
   ◆类和接口的全限定名；
   ◆字段的名称和描述符；
   ◆方法和名称和描述符。

6. Native Method Stack
   与VM Strack相似，VM Strack为JVM提供执行JAVA方法的服务，Native Method Stack则为JVM提供使用native 方法的服务。

小结：

1. 分清什么是实例什么是对象。Class a= new Class();此时a叫实例，而不能说a是对象。实例在栈中，对象在堆中，操作实例实际上是通过实例的指针间接操作对象。多个实例可以指向同一个对象。

2. 栈中的数据和堆中的数据销毁并不是同步的。方法一旦结束，栈中的局部变量立即销毁，但是堆中对象不一定销毁。因为可能有其他变量也指向了这个对象，直到栈中没有变量指向堆中的对象时，它才销毁，而且还不是马上销毁，要等垃圾回收扫描时才可以被销毁。

3. 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)。而类的方法却是该类的所有对象共享的，只有一套，对象使用方法的时候方法才被压入栈，方法不使用则不占用内存。

4. 生命周期： 堆内存属于java 应用程序所使用, 生命周期与jvm一致；栈内存属于线程所私有的，它的生命周期与线程相同

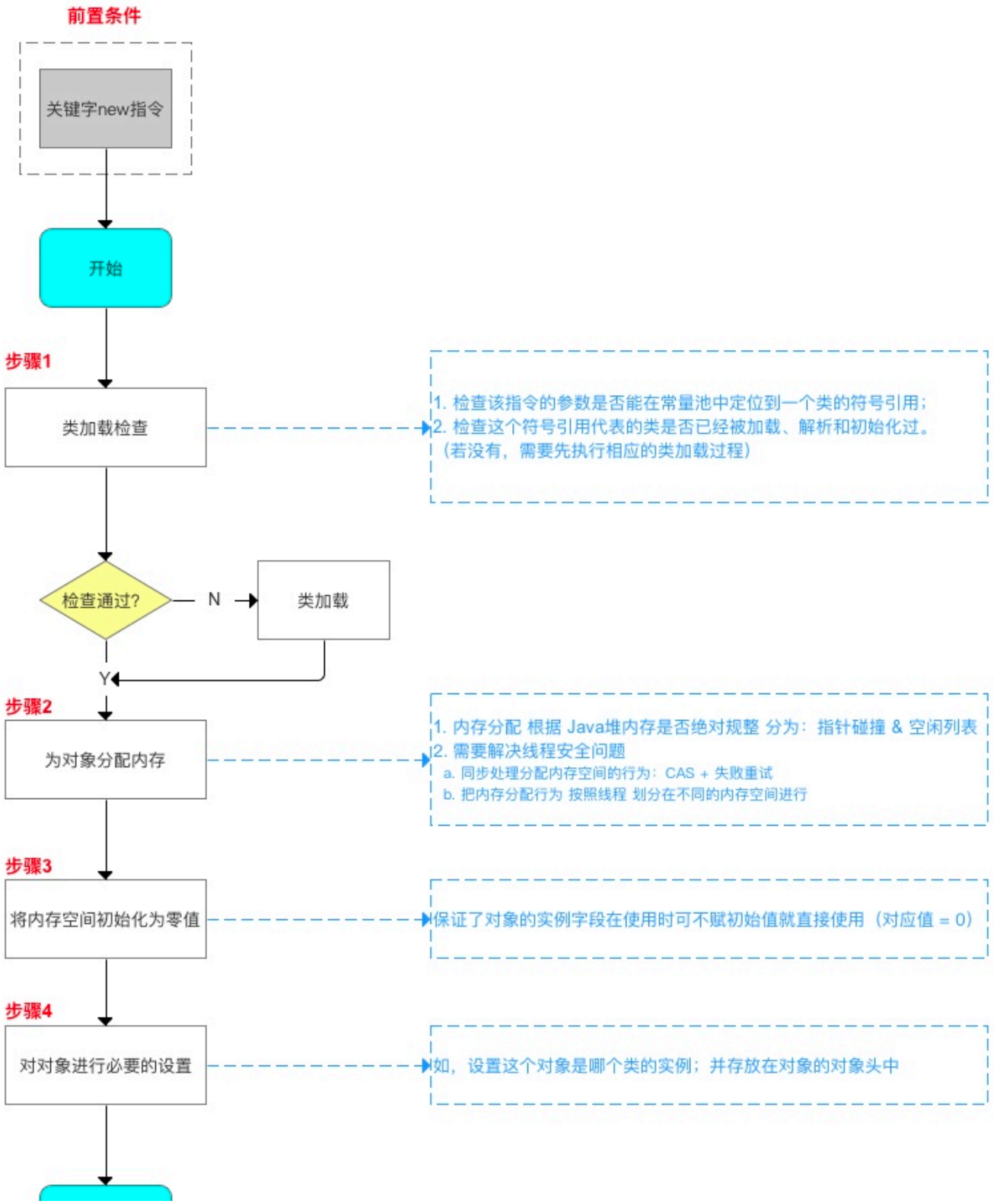
5. 引用：不论何时创建一个对象, 它总是存储在堆内存空间 并且栈内存空间包含对它的引用 . 栈内存空间只包含方法原始数据类型局部变量以及堆空间中对象的引用变量

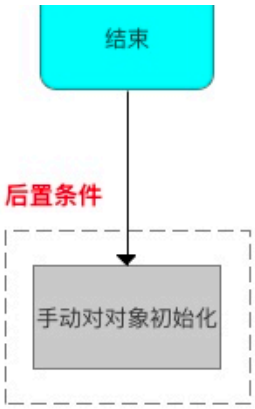
6. 在堆中的对象可以全局访问， 栈内存空间属于线程所私有

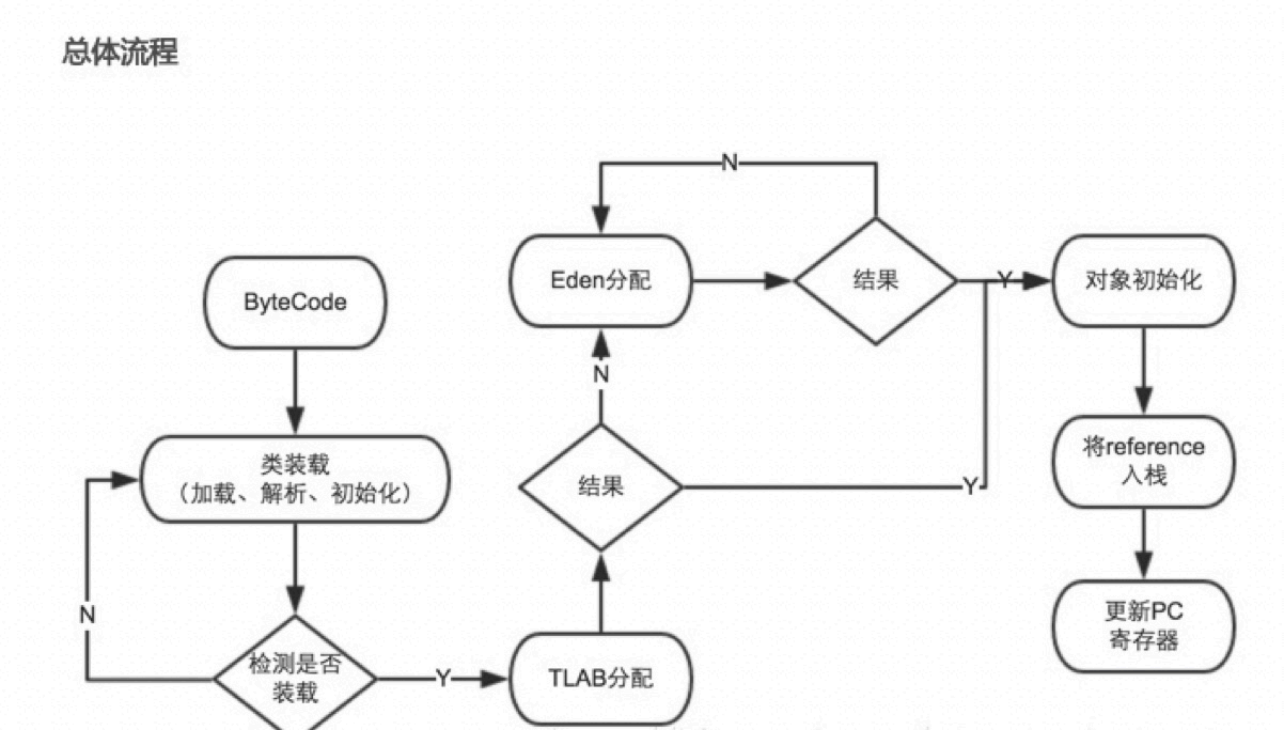
7. jvm 栈内存结构管理较为简单, 遵循LIFO 的原则, 堆空间内存管理较为复杂 , 细分为：新生代和老年代 etc..

8. 二者抛出异常的方式： 如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常, 堆内存抛出OutOfMemoryError异常

内存分配







逃逸分析就是 分析Java对象的动态作用域。当一个对象被定义之后，可能会被外部对象引用，称之为「方法逃逸」；也有可能被其他线程所引用，称之为「线程逃逸」。如果经过逃逸分析后发现，一个对象并没有逃逸出方法的话，那么就可能被优化成栈上分配。这样就无需在堆上分配内存，也无须进行垃圾回收了。

对象头

HotSpot虚拟机中，对象在内存中存储的布局可以分为三块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

HotSpot虚拟机的对象头(Object Header)包括两部分信息，第一部分用于存储对象自身的运行时数据， 如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等等，这部分数据的长度在32位和64位的虚拟机（暂 不考虑开启压缩指针的场景）中分别为32个和64个Bits，官方称它为“Mark Word”。

对象需要存储的运行时数据很多，其实已经超出了32、64位Bitmap结构所能记录的限度，但是对象头信息是与对象自身定义的数据无关的额 外存储成本，考虑到虚拟机的空间效率，Mark Word被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息，它会根据对象的状态复用自己的存储空间。例如在32位的HotSpot虚拟机 中对象未被锁定的状态下，Mark Word的32个Bits空间中的25Bits用于存储对象哈希码（HashCode），4Bits用于存储对象分代年龄，2Bits用于存储锁标志位，1Bit固定为0，在其他状态（轻量级锁定、重量级锁定、GC标记、可偏向）下对象的存储内容如下表所示。

锁状态	25 bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
无锁	对象的hashCode		对象分代年龄	0	01
偏向锁	线程ID	Epoch	对象分代年龄	1	01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC标记	空				11

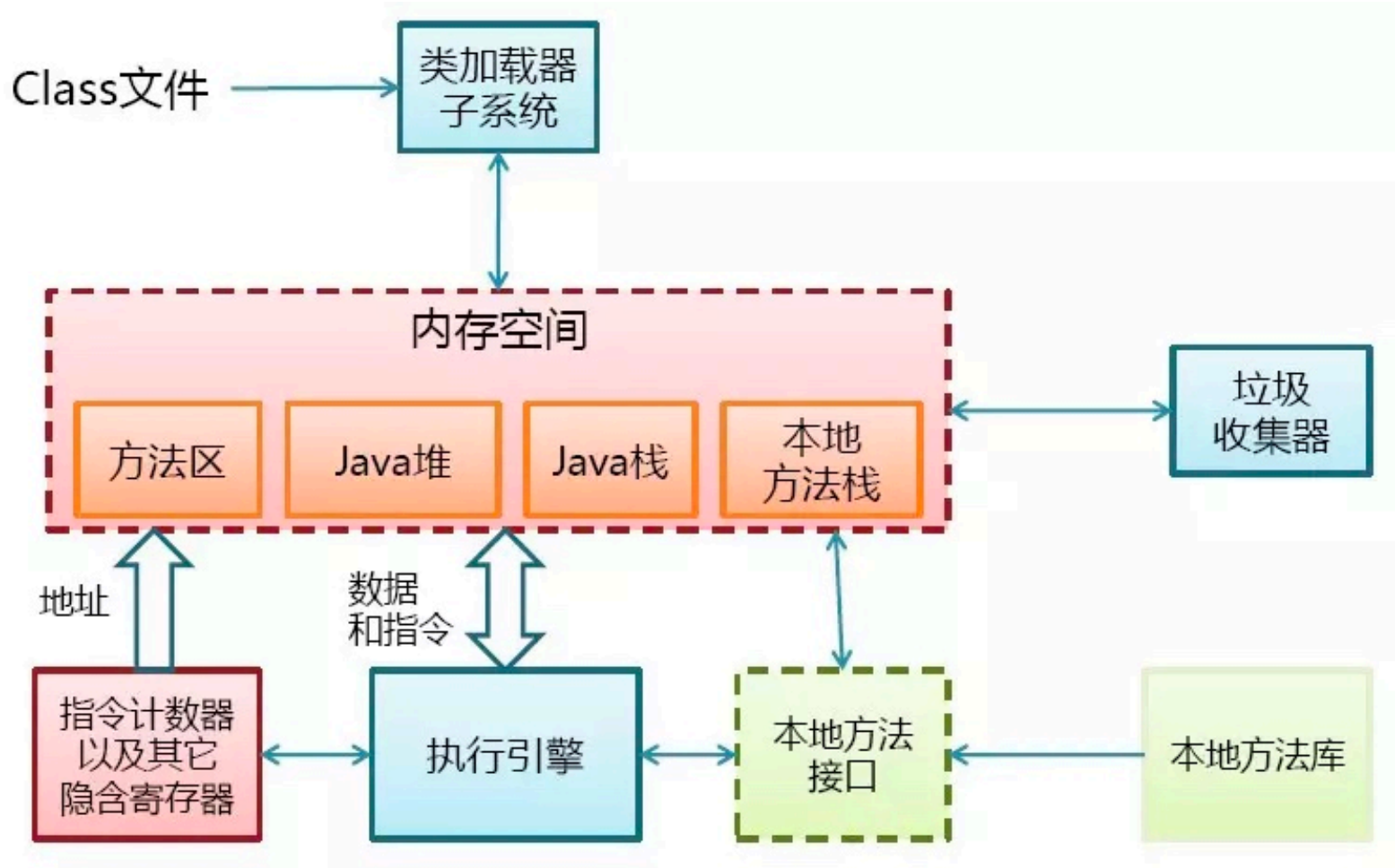
对象头的另外一部分是类型指针，这一部分用于存储对象的类型指针，该指针指向它的类元数据，JVM通过这个指针确定对象是哪个类的实例。该指针的位长度为JVM的一个字大小，即32位的JVM为32位，64位的JVM为64位。
如果应用的对象过多，使用64位的指针将浪费大量内存，为了节约内存可以使用选项+UseCompressedOops开启指针压缩，
如果对象是一个数组，那么对象头还需要有额外的空间用于存储数组的长度，这部分数据的长度也随着JVM架构的不同而不同：32位的JVM上，长度为32位；64位JVM则为64位。64位JVM如果开启+UseCompressedOops选项，该区域长度也将由64位压缩至32位。

这里要特别关注的是锁标志位，锁标志位与是否偏向锁对应到唯一的锁状态。
所以锁的状态保存在对象头中，所以再理解
Synchronized锁的到底是什么，锁住的是代码还是对象）（答案锁的是对象）？
synchronized 有 4 种锁状态，从低到高分别为：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，锁可以升级但不能降级。

偏向锁，就是在锁对象的对象头中有个ThreaddId字段，这个字段如果是空的，第一次获取锁的时候，就将自身的ThreadId写入到锁的ThreadId字段内，将锁头内的是否偏向锁的状态位置1.这样下次获取锁的时候，直接检查ThreadId是否和自身线程Id一致， 如果一致，则认为当前线程已经获取了锁，因此不需再次获取锁，略过了轻量级锁和重量级锁的加锁阶段。提高了效率。

在Hotspot JVM中，32位机器下，Integer对象的大小是int的几倍?
Integer只有一个int类型的成员变量value，所以其对象实际数据部分的大小是4个字节，markword 4字节，指针4字节，然后再在后面填充4个字节达到8字节的对齐，所以可以得出Integer对象的大小是16个字节。

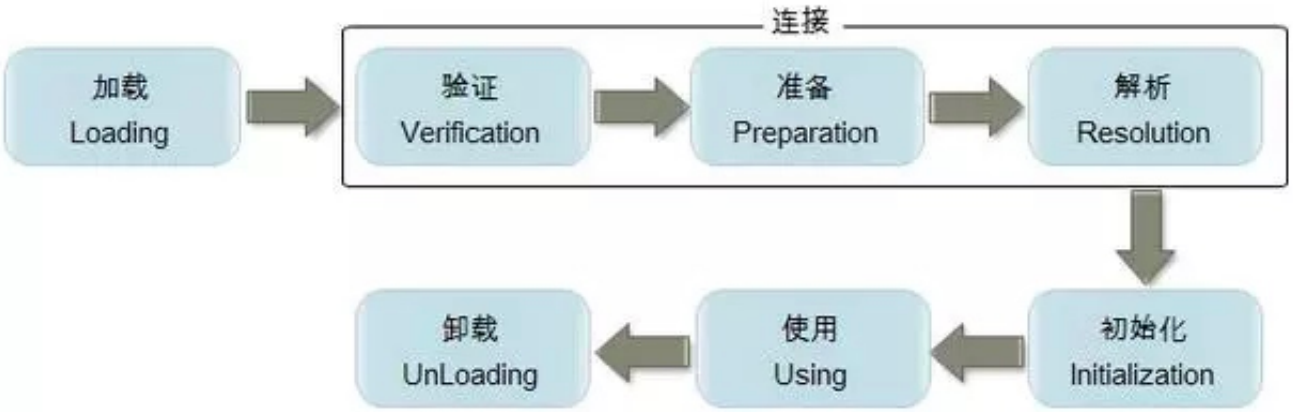
JVM整体结构



类加载

类加载机制概念

Java虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型，这就是虚拟机的加载机制。*
Class文件由类装载器装载后，在JVM中将形成一份描述Class结构的元信息对象，通过该元信息对象可以获知Class的结构信息：如构造函数，属性和方法等，Java允许用户借由这个Class相关的元信息对象间接调用Class对象的功能,这里就是我们经常能见到的Class类。



1. 加载
   类的装载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。
   在这个阶段，JVM主要完成三件事：

   1. 通过一个类的全限定名（包名与类名）来获取定义此类的二进制字节流（Class文件）。而获取的方式，可以通过jar包、war包、网络中获取、JSP文件生成等方式。
   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。这里只是转化了数据结构，并未合并数据。（方法区就是用来存放已被加载的类信息，常量，静态变量，编译后的代码的运行时内存区域）
   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。这个Class对象并没有规定是在Java堆内存中，它比较特殊，虽为对象，但存放在方法区中。

2. 类的连接
   类的加载过程后生成了类的java.lang.Class对象，接着会进入连接阶段，连接阶段负责将类的二进制数据合并入JRE（Java运行时环境）中。类的连接大致分三个阶段。

   1. 验证：验证被加载后的类是否有正确的结构，类数据是否会符合虚拟机的要求，确保不会危害虚拟机安全。
   2. 准备：为类的静态变量（static filed）在方法区分配内存，并赋默认初值（0值或null值）。如static int a = 100;
      1. 静态变量a就会在准备阶段被赋默认值0。
      2. 对于一般的成员变量是在类实例化时候，随对象一起分配在堆内存中。
      3. 另外，静态常量（static final filed）会在准备阶段赋程序设定的初值，如static final int a = 666; 静态常量a就会在准备阶段被直接赋值为666，对于静态变量，这个操作是在初始化阶段进行的。
   3. 解析：将类的二进制数据中的符号引用换为直接引用。

3. 类的初始化

类初始化是类加载的最后一步，除了加载阶段，用户可以通过自定义的类加载器参与，其他阶段都完全由虚拟机主导和控制。到了初始化阶段才真正执行Java代码。
类的初始化的主要工作是为静态变量赋程序设定的初值。
如static int a = 100;在准备阶段，a被赋默认值0，在初始化阶段就会被赋值为100。
Java虚拟机规范中严格规定了有且只有4种情况必须对类进行初始化：

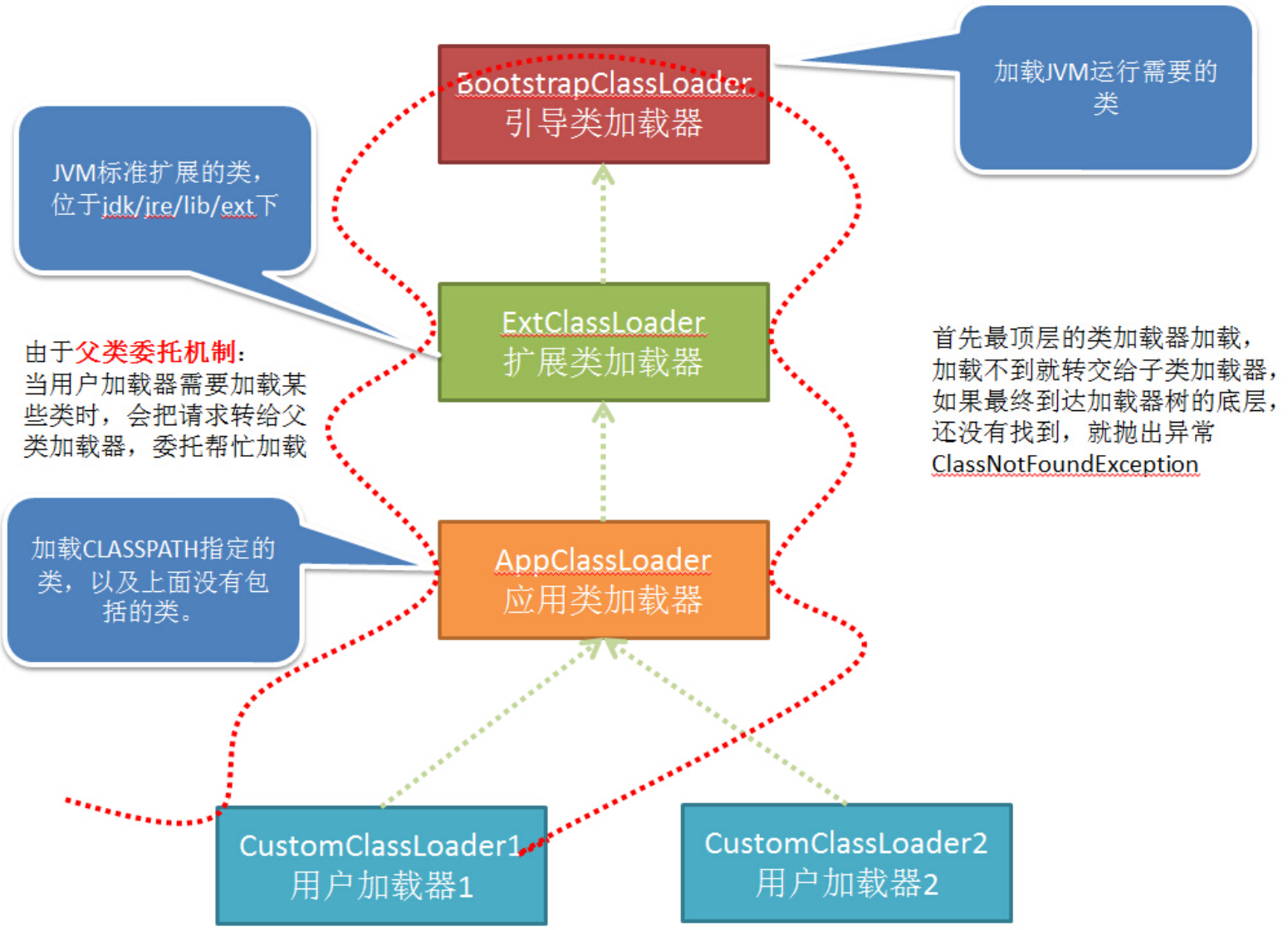
1. 使用new字节码指令创建类的实例，或者使用getstatic、putstatic读取或设置一个静态字段的值（放入常量池中的常量除外），或者调用一个静态方法的时候，对应类必须进行过初始化。
2. 通过java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则要首先进行初始化。
3. 当初始化一个类的时候，如果发现其父类没有进行过初始化，则首先触发父类初始化。对于静态字段，只有直接定义这个字段的类才会被初始化，因此，通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化
4. 当虚拟机启动时，用户需要指定一个主类（包含main()方法的类），虚拟机会首先初始化这个类。

5. 例子

```
class ConstClass {
    static {
        System.out.println("ConstClass init");
    }
    public static final String HELLOWORLD = "hello world";
}
public class Test {
    public static void main(String[] args) {
        System.out.println(ConstClass.HELLOWORLD);// 调用类常量
    }
}
```

双亲委派模型

在JVM中并不是一次性把所有的文件都加载到，而是一步一步的，按照需要来加载。
比如JVM启动时，会通过不同的类加载器加载不同的类。当用户在自己的代码中，需要某些额外的类时，再通过加载机制加载到JVM中，并且存放一段时间，便于频繁使用。
对于任何一个类，都需要由加载它的类加载器和这个类来确立其在JVM中的唯一性。也就是说，两个类来源于同一个Class文件，并且被同一个类加载器加载，这两个类才相等。

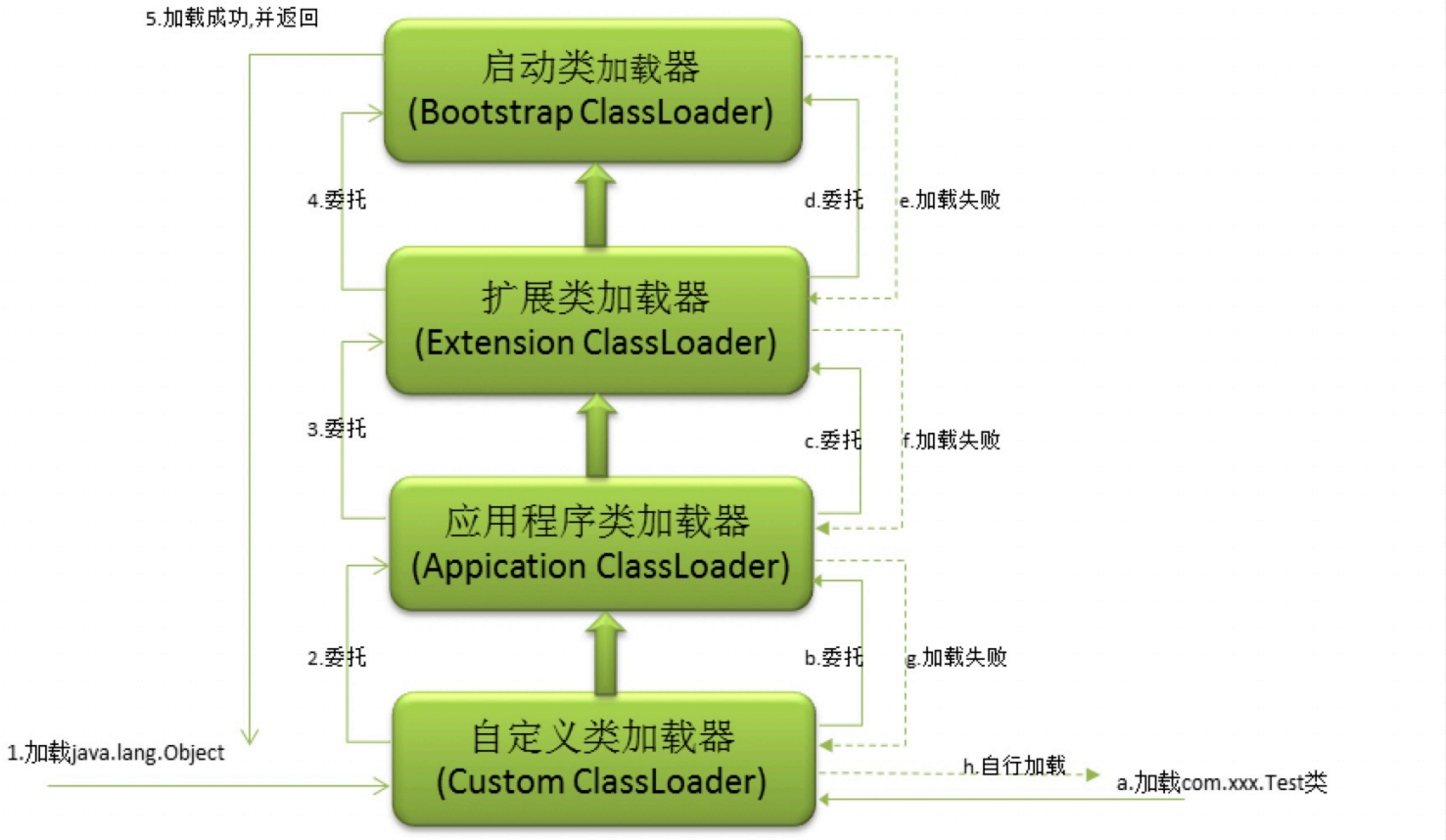


虚拟机提供了3种类加载器3种类加载器，引导（Bootstrap）类加载器、扩展（Extension）类加载器、系统（System）类加载器（也称应用类加载器）；以及父类加载器为AppClassLoader的自定义类加载器。在Launcher中实现

1. 启动（Bootstrap）类加载器（由C++实现，没有父类）
   启动类加载器主要加载的是JVM自身需要的类，这个类加载使用C++语言实现的，是虚拟机自身的一部分，它负责将 /lib路径下的核心类库或–Xbootclasspath参数指定的路径下的jar包加载到内存中，注意必由于虚拟机是按照文件名识别加载jar包的，如rt.jar，如果文件名不被虚拟机识别，即使把jar包丢到lib目录下也是没有作用的(出于安全考虑，Bootstrap启动类加载器只加载包名为java、javax、sun等开头的类)。
2. 扩展（ExtClassLoader）类加载器（由Java语言实现，父类加载器为null）
   扩展类加载器是指Sun公司(已被Oracle收购)实现的sun.misc.Launcher$ExtClassLoader类，由Java语言实现的，是Launcher的静态内部类，它负责加载/lib/ext目录下或者由系统变量–Djava.ext.dir指定位路径中的类库，开发者可以直接使用标准扩展类加载器。
3. 系统（AppClassLoader）类加载器（由Java语言实现，父类加载器为ExtClassLoader）

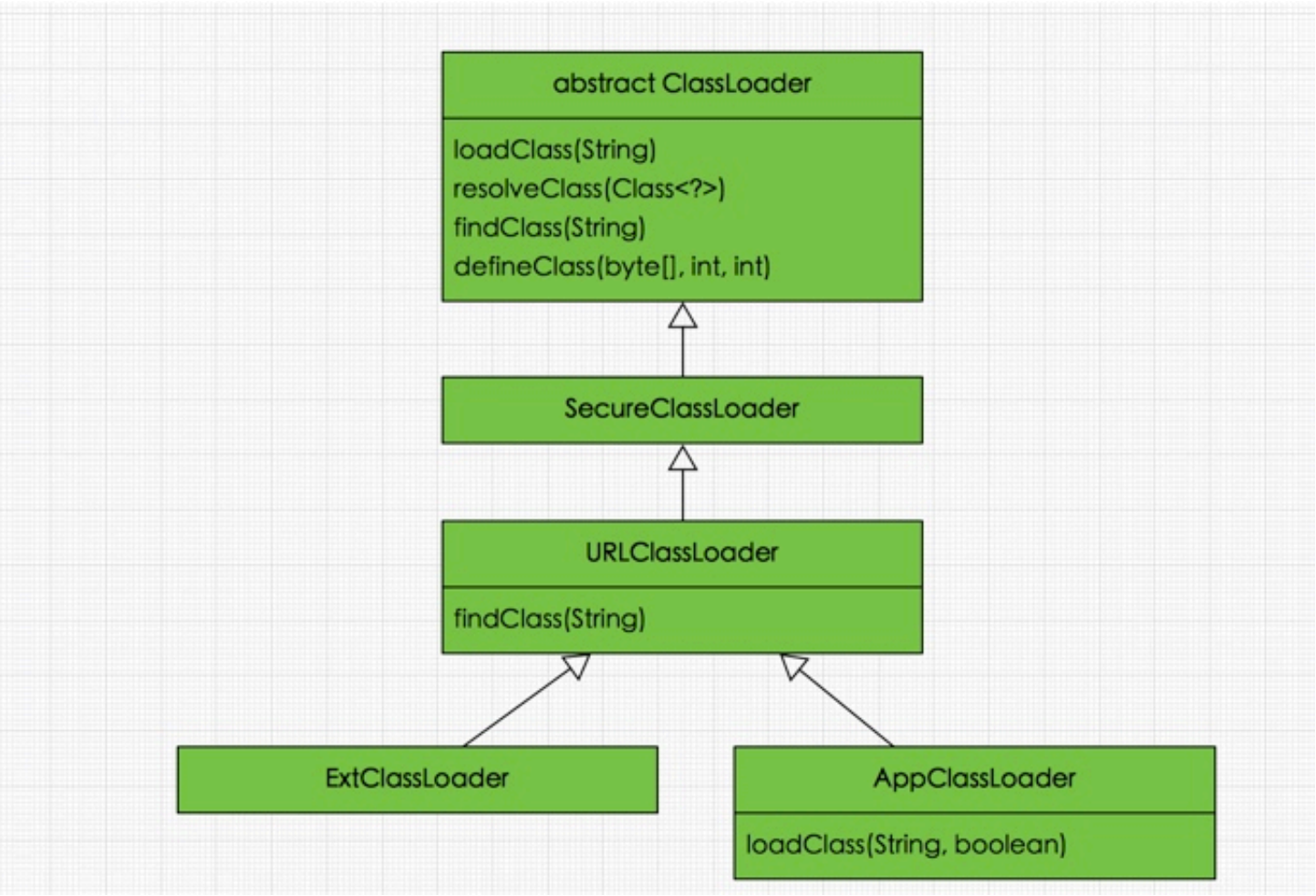
也称应用程序加载器,它负责加载系统类路径java -classpath或-D java.class.path 指定路径下的类库，也就是我们经常用到的classpath路径，开发者可以直接使用系统类加载器，一般情况下该类加载是程序中默认的类加载器，通过ClassLoader#getSystemClassLoader()方法可以获取到该类加载器。

这3种类加载器之间存在着父子关系(区别于java里的继承)，子加载器保存着父加载器的引用。当一个类加载器需要加载一个目标类时，会先委托父加载器去加载，然后父加载器会在自己的加载路径中搜索目标类，父加载器在自己的加载范围中找不到时，才会交还给子加载器加载目标类。
采用双亲委托模式可以避免类加载混乱，而且还将类分层次了，例如java中lang包下的类在jvm启动时就被启动类加载器加载了，而用户一些代码类则由应用程序类加载器(AppClassLoader)加载，基于双亲委托模式，就算用户定义了与lang包中一样的类，最终还是由应用程序类加载器委托给启动类加载器去加载，这个时候启动类加载器发现已经加载过了lang包下的类了，所以两者都不会再重新加载。
当然，如果使用者通过自定义的类加载器可以强行打破这种双亲委托模型，但也不会成功的，java安全管理器抛出将会抛出java.lang.SecurityException异常。



实现原理

Java中定义的类加载器及其双亲委派模式的实现，它们类图关系



loadClass()方法是ClassLoader类自己实现的，该方法中的逻辑就是双亲委派模式的实现，顶层的类加载器是ClassLoader类，它是一个抽象类，其后所有的类加载器都继承自ClassLoader（不包括启动类加载器），这里我们主要介绍ClassLoader中几个比较重要的方法。

1. loadClass(String name, boolean resolve)是一个重载方法，resolve参数代表是否生成class对象的同时进行解析相关操作。

```
protected Class<?> loadClass(String name, boolean resolve)
  throws ClassNotFoundException
{
  synchronized (getClassLoadingLock(name)) {
      // 先从缓存查找该class对象，找到就不用重新加载
      Class<?> c = findLoadedClass(name);
      if (c == null) {
          long t0 = System.nanoTime();
          try {
              if (parent != null) {
                  //如果找不到，则委托给父类加载器去加载
                  c = parent.loadClass(name, false);
              } else {
              //如果没有父类，则委托给启动加载器去加载
```

```
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }
            if (c == null) {
                // If still not found, then invoke findClass in order
                // 如果都没有找到，则通过自定义实现的findClass去查找并加载
                c = findClass(name);
                // this is the defining class loader; record the stats
                sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);
                sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                sun.misc.PerfCounter.getFindClasses().increment();
            }
        }
        if (resolve) {//是否需要在加载时进行解析
            resolveClass(c);
        }
        return c;
    }
}
```

当类加载请求到来时，先从缓存中查找该类对象，如果存在直接返回，如果不存在则交给该类加载去的父加载器去加载，倘若没有父加载则交给顶级启动类加载器去加载，最后倘若仍没有找到，则使用findClass()方法去加载（关于findClass()稍后会进一步介绍）。从loadClass实现也可以知道如果不想重新定义加载类的规则，也没有复杂的逻辑，只想在运行时加载自己指定的类，那么我们可以直接使用this.getClass().getClassLoder.loadClass("className")，这样就可以直接调用ClassLoader的loadClass方法获取到class对象。

2. 自定义的类加载逻辑写在findClass()方法中,findClass()方法是在loadClass()方法中被调用的，当loadClass()方法中父加载器加载失败后，则会调用自己的findClass()方法来完成类加载，这样就可以保证自定义的类加载器也符合双亲委托模式,defineClass()方法通常与findClass()方法一起使用，一般情况下，在自定义类加载器时，会直接覆盖ClassLoader的findClass()方法并编写加载规则，取得要加载类的字节码后转换成流，然后调用defineClass()方法生成类的Class对象，简单例子如下：

```
protected Class<?> findClass(String name) throws ClassNotFoundException {
  // 获取类的字节数组
  byte[] classData = getClassData(name);
  if (classData == null) {
      throw new ClassNotFoundException();
```

```
    } else {
        //使用defineClass生成class对象
        return defineClass(name, classData, 0, classData.length);
    }
}
```

直接调用defineClass()方法生成类的Class对象，这个类的Class对象并没有解析(也可以理解为链接阶段，毕竟解析是链接的最后一步)，其解析操作需要等待初始化阶段进行。

3. resolveClass(Class<?> c)
   使用该方法可以使用类的Class对象创建完成也同时被解析。前面我们说链接阶段主要是对字节码进行验证，为类变量分配内存并设置初始值同时将字节码文件中的符号引用转换为直接引用。

上述4个方法是ClassLoader类中的比较重要的方法，也是我们可能会经常用到的方法。接看SercureClassLoader扩展了 ClassLoader，新增了几个与使用相关的代码源(对代码源的位置及其证书的验证)和权限定义类验证(主要指对class源码的访问权限)的方法，一般我们不会直接跟这个类打交道，更多是与它的子类URLClassLoader有所关联，前面说过，ClassLoader是一个抽象类，很多方法是空的没有实现，比如 findClass()、findResource()等。而URLClassLoader这个实现类为这些方法提供了具体的实现，并新增了URLClassPath类协助取得Class字节码流等功能，在编写自定义类加载器时，如果没有太过于复杂的需求，可以直接继承URLClassLoader类，这样就可以避免自己去编写findClass()方法及其获取字节码流的方式，使自定义类加载器编写更加简洁

双亲委派模型的好处：

1. 主要是为了安全性，避免用户自己编写的类动态替换 Java的一些核心类，比如 String。
2. 同时也避免了类的重复加载，因为 JVM中区分不同类，不仅仅是根据类名，相同的 class文件被不同的 ClassLoader加载就是不同的两个类。双亲委派模型的好处：

在JVM中表示两个class对象是否为同一个类对象存在两个必要条件

1. 类的完整类名必须一致，包括包名。
2. 加载这个类的ClassLoader(指ClassLoader实例对象)必须相同。

自定义类加载器

自定义类的应用场景：

1. 加密：Java代码可以轻易的被反编译，如果你需要把自己的代码进行加密以防止反编译，可以先将编译后的代码用某种加密算法加密，类加密后就不能再用Java的ClassLoader去加载类了，这时就需要自定义ClassLoader在加载类的时候先解密类，然后再加载。
2. 从非标准的来源加载代码：如果你的字节码是放在数据库、甚至是在云端，就可以自定义类加载器，从指定的来源加载类。

3. 以上两种情况在实际中的综合运用：比如你的应用需要通过网络来传输 Java 类的字节码，为了安全性，这些字节码经过了加密处理。这个时候你就需要自定义类加载器来从某个网络地址上读取加密后的字节代码，接着进行解密和验证，最后定义出在Java虚拟机中运行的类。

如何自定义类加载器：从上面对于java.lang.ClassLoader的loadClass(String name, boolean resolve)方法的解析来看，可以得出以下2个结论：

1. 如果不想打破双亲委派模型，那么只需要重写findClass方法即可
2. 如果想打破双亲委派模型，那么就重写整个loadClass方法

实例

```
class HClassLoader extends ClassLoader {

    private String classPath;

    public HClassLoader(String classPath) {
        this.classPath = classPath;
    }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        try {
            byte[] data = loadByte(name);
            return defineClass(name, data, 0, data.length);
        } catch (Exception e) {
            e.printStackTrace();
            throw new ClassNotFoundException();
        }

    }

    /**
     * 获取.class的字节流
     *
     * @param name
     * @return
     * @throws Exception
     */
    private byte[] loadByte(String name) throws Exception {
        name = name.replaceAll("\\.", "/");
        FileInputStream fis = new FileInputStream(classPath + "/" + name + ".class");
        int len = fis.available();
        byte[] data = new byte[len];
```

```
            fis.read(data);
            fis.close();

            // 字节流解密
            data = DESInstance.deCode("1234567890qwertyuiopasdf".getBytes(), data);

            return data;
        }
    }
```

测试类

```
@Test
public void testClassLoader() throws Exception {
    HClassLoader myClassLoader = new HClassLoader("e:/temp/a");
    Class clazz = myClassLoader.loadClass("com.demo.Car");
    Object o = clazz.newInstance();
    Method print = clazz.getDeclaredMethod("print", null);
    print.invoke(o, null);
}
```

实体类

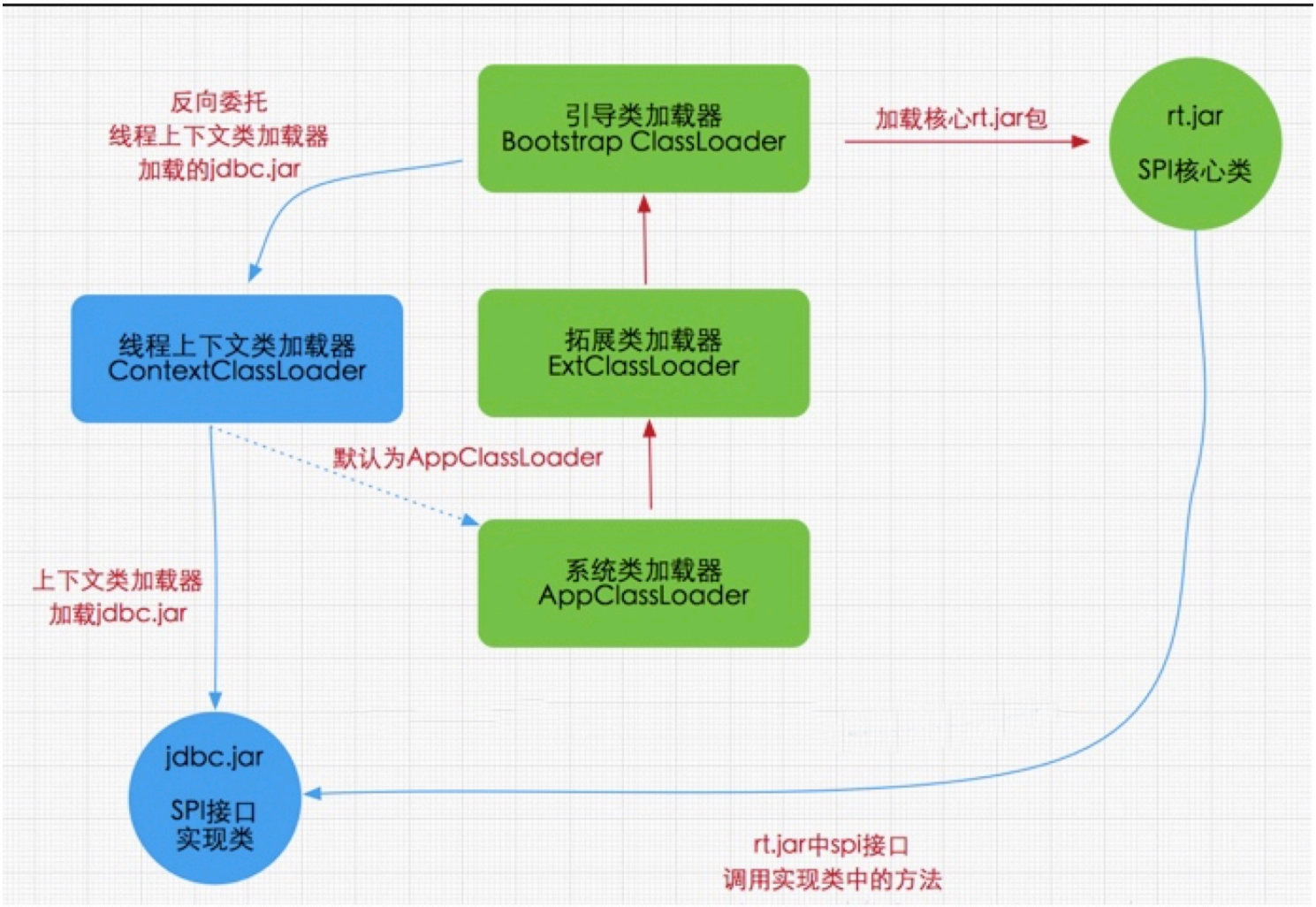
```
public class Car {

    public Car() {
        System.out.println("Car:" + getClass().getClassLoader());
        System.out.println("Car Parent:" + getClass().getClassLoader().getParent());
    }

    public String print() {
        System.out.println("Car:print()");
        return "carPrint";
    }
 }
```

双亲委派模型的破坏者-线程上下文类加载器

在Java应用中存在着很多服务提供者接口（Service Provider Interface，SPI），这些接口允许第三方为它们提供实现，如常见的 SPI 有 JDBC、JNDI等，这些 SPI 的接口属于 Java 核心库，一般存在rt.jar包中，由Bootstrap类加载器加载，而 SPI 的第三方实现代码则是作为Java应用所依赖的 jar

包被存放在classpath路径下，由于SPI接口中的代码经常需要加载具体的第三方实现类并调用其相关方法，但SPI的核心接口类是由引导类加载器来加载的，而Bootstrap类加载器无法直接加载SPI的实现类，同时由于双亲委派模式的存在，Bootstrap类加载器也无法反向委托AppClassLoader加载器SPI的实现类。在这种情况下，我们就需要一种特殊的类加载器来加载第三方的类库，而线程上下文类加载器就是很好的选择。

线程上下文类加载器（contextClassLoader）是从 JDK 1.2 开始引入的，我们可以通过java.lang.Thread类中的getContextClassLoader()和 setContextClassLoader(ClassLoader cl)方法来获取和设置线程的上下文类加载器。如果没有手动设置上下文类加载器，线程将继承其父线程的上下文类加载器，初始线程的上下文类加载器是系统类加载器（AppClassLoader）,在线程中运行的代码可以通过此类加载器来加载类和资源，如下图所示，以jdbc.jar加载为例

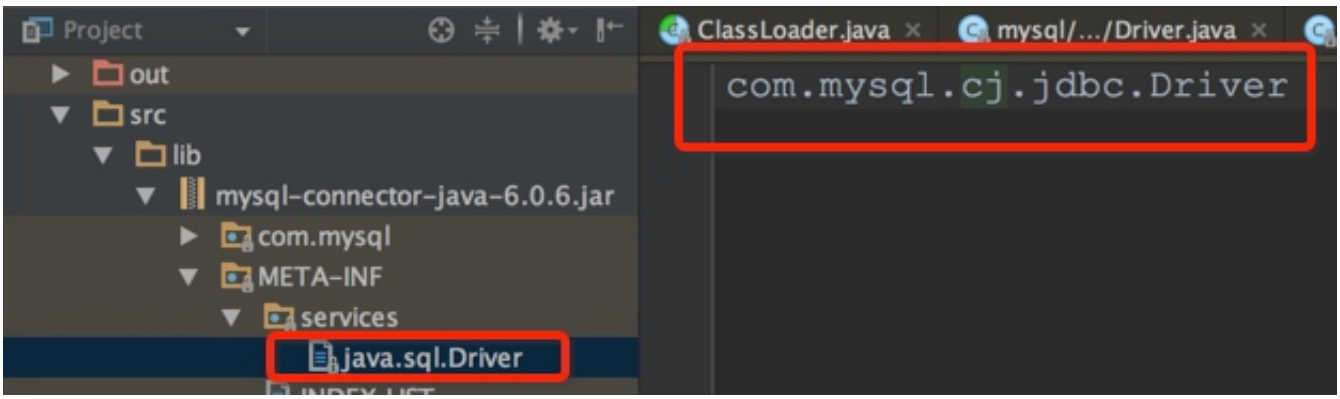


从图可知rt.jar核心包是有Bootstrap类加载器加载的，其内包含SPI核心接口类，由于SPI中的类经常需要调用外部实现类的方法，而jdbc.jar包含外部实现类(jdbc.jar存在于classpath路径)无法通过Bootstrap类加载器加载，因此只能委派线程上下文类加载器把jdbc.jar中的实现类加载到内存以便SPI相关类使用。显然这种线程上下文类加载器的加载方式破坏了“双亲委派模型”，它在执行过程中抛弃双亲委派加载链模式，使程序可以逆向使用类加载器，当然这也使得Java类加载器变得更加灵活。为了进一步证实这种场景，不妨看看DriverManager类的源码，DriverManager是Java核心rt.jar包中的类，该类用来管理不同数据库的实现驱动即Driver，它们都实现了Java核心包中的java.sql.Driver接

口，如mysql驱动包中的com.mysql.jdbc.Driver，这里主要看看如何加载外部实现类，在DriverManager初始化时会执行如下代码

```
//DriverManager是Java核心包rt.jar的类
public class DriverManager {
    //省略不必要的代码
    static {
        loadInitialDrivers();//执行该方法
        println("JDBC DriverManager initialized");
    }

//loadInitialDrivers方法
 private static void loadInitialDrivers() {
     sun.misc.Providers()
     AccessController.doPrivileged(new PrivilegedAction<Void>() {
            public Void run() {
                //加载外部的Driver的实现类
                ServiceLoader<Driver> loadedDrivers = ServiceLoader.load(Driver.class);
              //省略不必要的代码......
            }
        });
    }
```

在DriverManager类初始化时执行了loadInitialDrivers()方法,在该方法中通过ServiceLoader.load(Driver.class);去加载外部实现的驱动类，ServiceLoader类会去读取mysql的jdbc.jar下META-INF文件的内容，如下所示



加载meta-inf的过程:
实现延迟服务提供者查找
DriverManager.loadInitialDrivers-> ServiceLoader.load->reload()->
lookupIterator = new LazyIterator(service, loader);
加载meta-inf，初始化驱动
loadedDrivers.iterator()->driversIterator.hasNext()->hasNextService-

>ClassLoader.getSystemResources(fullName);
这样ServiceLoader会帮助我们处理一切，并最终通过load()方法加载

```
public static <S> ServiceLoader<S> load(Class<S> service) {
     //通过线程上下文类加载器加载
      ClassLoader cl = Thread.currentThread().getContextClassLoader();
      return ServiceLoader.load(service, cl);
  }
```

核心包的SPI类对外部实现类的加载都是基于线程上下文类加载器执行的，通过这种方式实现了Java核心代码内部去调用外部实现类。

简而言之就是ContextClassLoader默认存放了AppClassLoader的引用，由于它是在运行时被放在了线程中，所以不管当前程序处于何处（BootstrapClassLoader或是ExtClassLoader等），在任何需要的时候都可以用Thread.currentThread().getContextClassLoader()取出应用程序类加载器来完成需要的操作

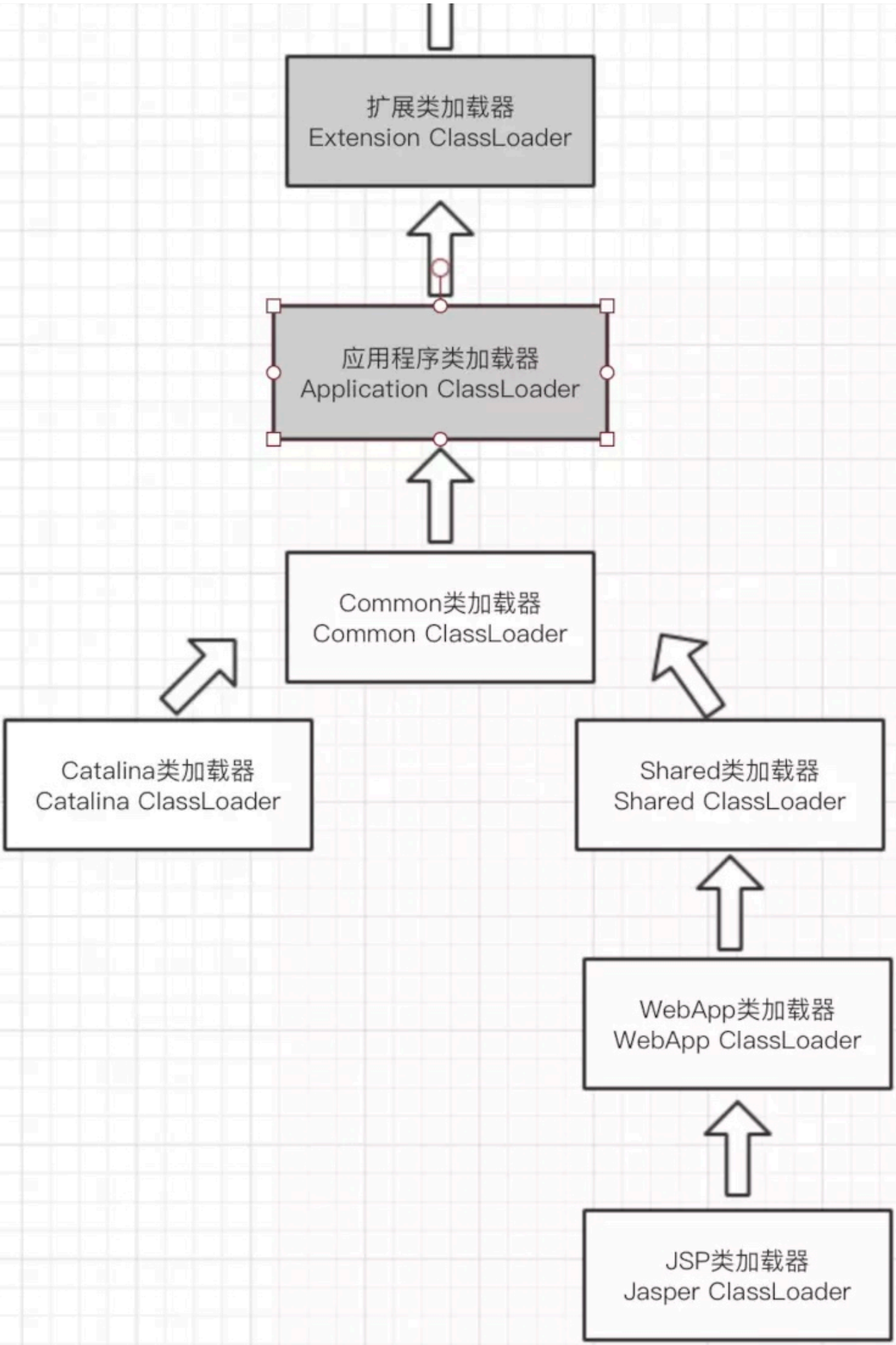
Tomcat类加载器结构

Tomcat作为一个web容器需要解决下面几个问题：
1）部署在同一个Web容器上的两个Web应用程序所使用的Java类库可以实现相互隔离。这是最基本的需求，两个不同的应用程序可能会依赖同一个第三方类库的不同版本，不能要求一个类库在一个服务器中只有一份，服务器应当保证两个应用程序的类库可以互相独立使用。

2）部署在同一个Web容器上的两个Web应用程序所使用的Java类库可以互相共享。这个需求也很常见，例如，用户可能有10个使用Spring组织的应用程序部署在同一台服务器上，如果把10份Spring分别存放在各个应用程序的隔离目录中，将会是很大的资源浪费——这主要倒不是浪费磁盘空间的问题，而是指类库在使用时都要被加载到Web容器的内存，如果类库不能共享，虚拟机的方法区就会很容易出现过度膨胀的风险。

3）Web容器需要尽可能地保证自身的安全不受部署的Web应用程序影响。目前，有许多主流的Java Web容器自身也是使用Java语言来实现的。因此，Web容器本身也有类库依赖的问题，一般来说，基于安全考虑，容器所使用的类库应该与应用程序的类库互相独立。

Tomcat类加载架构如下图：


扩展类加载器
Extension ClassLoader
应用程序类加载器
Application ClassLoader
Common类加载器
Common ClassLoader
Catalina类加载器
Catalina ClassLoader
Shared类加载器
Shared ClassLoader
WebApp类加载器
WebApp ClassLoader
JSP类加载器
Jasper ClassLoader

Web应用类加载器默认的加载顺序是：

(1).先从缓存中加载；
(2).如果没有，则从JVM的Bootstrap类加载器加载；
(3).如果没有，则从当前类加载器加载（按照WEB-INF/classes、WEB-INF/lib的顺序）；
(4).如果没有，则从父类加载器加载，以加载顺序是AppClassLoader、Common、Shared。