01 官网

1.1 JDK8

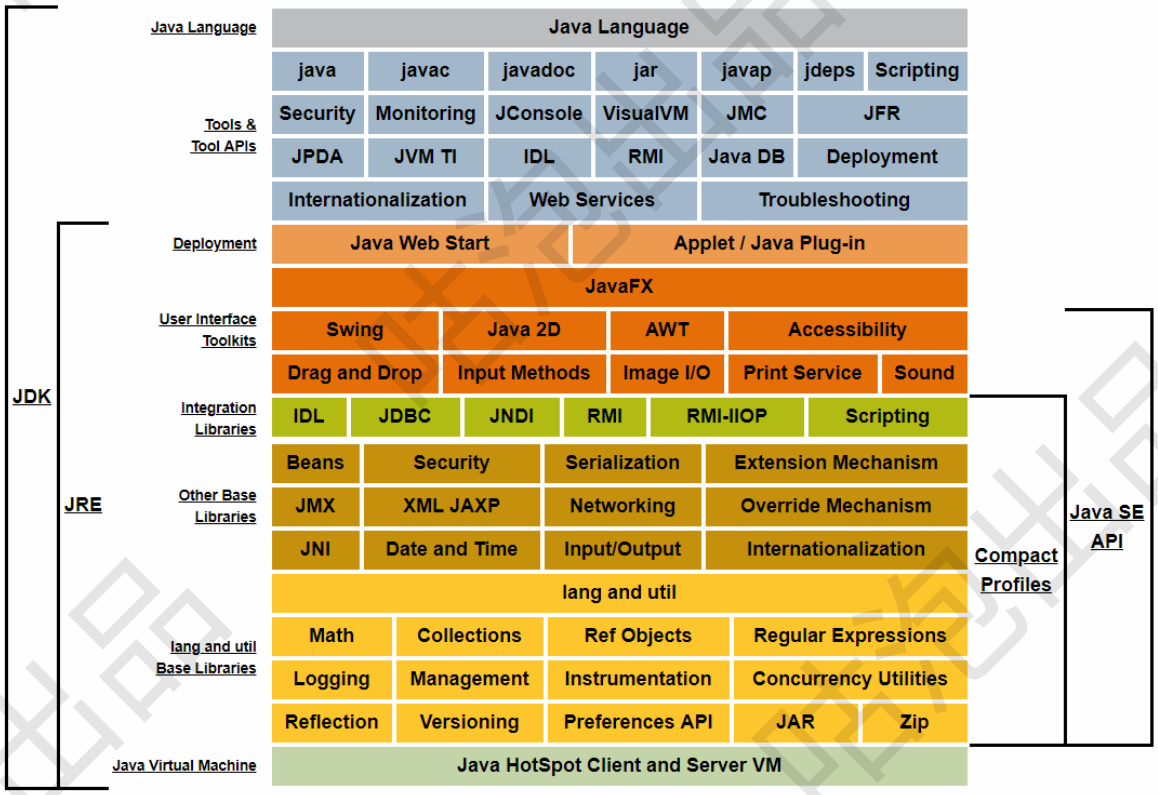
官网:https://docs.oracle.com/javase/8/

1.2 The relation of JDK/JRE/JVM

Reference -> Developer Guides -> 定位到:https://docs.oracle.com/javase/8/docs/index.html

```
Oracle has two products that implement Java Platform Standard Edition (Java SE)
8: Java SE Development Kit (JDK) 8 and Java SE Runtime Environment (JRE) 8.

JDK 8 is a superset of JRE 8, and contains everything that is in JRE 8, plus
tools such as the compilers and debuggers necessary for developing applets and
applications. JRE 8 provides the libraries, the Java Virtual Machine (JVM), and
other components to run applets and applications written in the Java programming
language. Note that the JRE includes components not required by the Java SE
specification, including both standard and non-standard Java components.
```



02 源码到类文件

2.1 源码

```
class Person{
    private String name;
    private int age;
    private static String address;
    private final static String hobby="Programming";
    public void say(){
        System.out.println("person say...");
    }
    public int calc(int op1,int op2){
        return op1+op2;
    }
}
```

> 编译: javac Person.java ---> Person.class

2.2 编译过程

> Person.java -> 词法分析器 -> tokens流 -> 语法分析器 -> 语法树/抽象语法树 -> 语义分析器
>
> -> 注解抽象语法树 -> 字节码生成器 -> Person.class文件

2.3 类文件(Class文件)

`官网The class File Format`:https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html

```
cafe babe 0000 0034 0027 0a00 0600 1809
0019 001a 0800 1b0a 001c 001d 0700 1e07
001f 0100 046e 616d 6501 0012 4c6a 6176
612f 6c61 6e67 2f53 7472 696e 673b 0100
0361 6765 0100 0149 0100 0761 6464 7265
......
```

> magic(魔数):
>
> > The `magic` item supplies the magic number identifying the `class` file format; it has the value `0xCAFEBABE`.

```
cafe babe
```

> minor_version, major_version

```
0000 0034    对应10进制的52，代表JDK 8中的一个版本
```

> constant_pool_count

```
0027    对应十进制27，代表常量池中27个常量
```

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
```

```
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

> .class字节码文件
>
> ```
> 魔数与class文件版本
> 常量池
> 访问标志
> 类索引、父类索引、接口索引
> 字段表集合
> 方法表集合
> 属性表集合
> ```

03 类文件到虚拟机(类加载机制)

3.1 装载(Load)

> 查找和导入class文件

```
(1)通过一个类的全限定名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Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口
```

3.2 链接(Link)

3.2.1 验证(Verify)

> 保证被加载类的正确性

- 文件格式验证
- 元数据验证
- 字节码验证
- 符号引用验证

3.2.2 准备(Prepare)

> 为类的静态变量分配内存，并将其初始化为默认值

3.2.3 解析(Resolve)

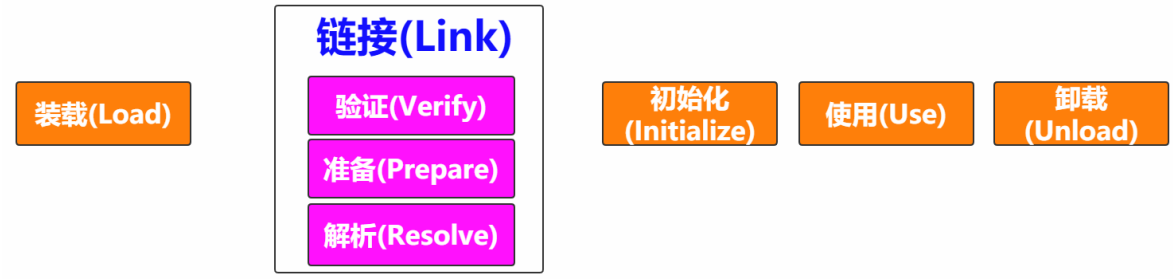
> 把类中的符号引用转换为直接引用

3.3 初始化(Initialize)

> 对类的静态变量，静态代码块执行初始化操作

3.4 类加载机制图解

> 使用和卸载不算是类加载过程中的阶段，只是画完整了一下



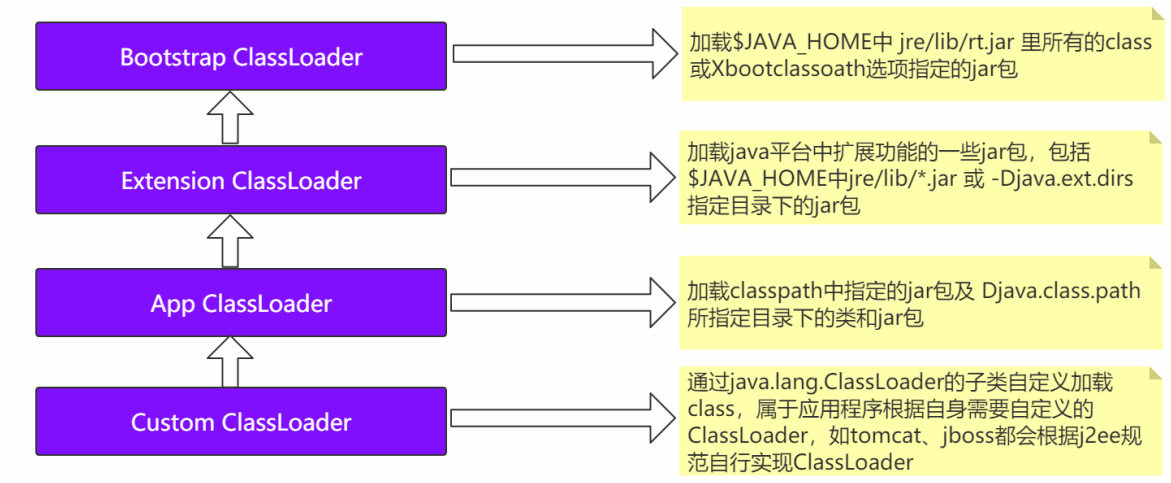
04 类装载器ClassLoader

> 在装载(Load)阶段，其中第(1)步:通过类的全限定名获取其定义的二进制字节流，需要借助类装载器完成，顾名思义，就是用来装载Class文件的。
>
> (1)通过一个类的全限定名获取定义此类的二进制字节流

4.1 分类

1）Bootstrap ClassLoader 负责加载$JAVA_HOME中 jre/lib/rt.jar 里所有的class或Xbootclassoath选项指定的jar包。由C++实现，不是ClassLoader子类。
2）Extension ClassLoader 负责加载java平台中扩展功能的一些jar包，包括$JAVA_HOME中jre/lib/*.jar 或 -Djava.ext.dirs指定目录下的jar包。
3）App ClassLoader 负责加载classpath中指定的jar包及 Djava.class.path 所指定目录下的类和jar包。
4）Custom ClassLoader 通过java.lang.ClassLoader的子类自定义加载class，属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader。

4.2 图解



4.3 加载原则

检查某个类是否已经加载：顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个Classloader已加载，就视为已加载此类，保证此类只所有ClassLoader加载一次。

加载的顺序：加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

双亲委派机制

> 定义：如果一个类加载器在接到加载类的请求时，它首先不会自己尝试去加载这个类，而是把这个请求任务委托给父类加载器去完成，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。
>
> 优势：Java类随着加载它的类加载器一起具备了一种带有优先级的层次关系。比如，Java中的Object类，它存放在rt.jar之中,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在各种类加载环境中都是同一个类。如果不采用双亲委派模型，那么由各个类加载器自己取加载的话，那么系统中会存在多种不同的Object类。
>
> 破坏：可以继承ClassLoader类，然后重写其中的loadClass方法，其他方式大家可以自己了解拓展一下。

05 运行时数据区(Run-Time Data Areas)

> 在装载阶段的第(2),(3)步可以发现有运行时数据，堆，方法区等名词
>
> (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Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口
>
> 说白了就是类文件被类装载器装载进来之后，类中的内容(比如变量，常量，方法，对象等这些数据得要有个去处，也就是要存储起来，存储的位置肯定是在JVM中有对应的空间)

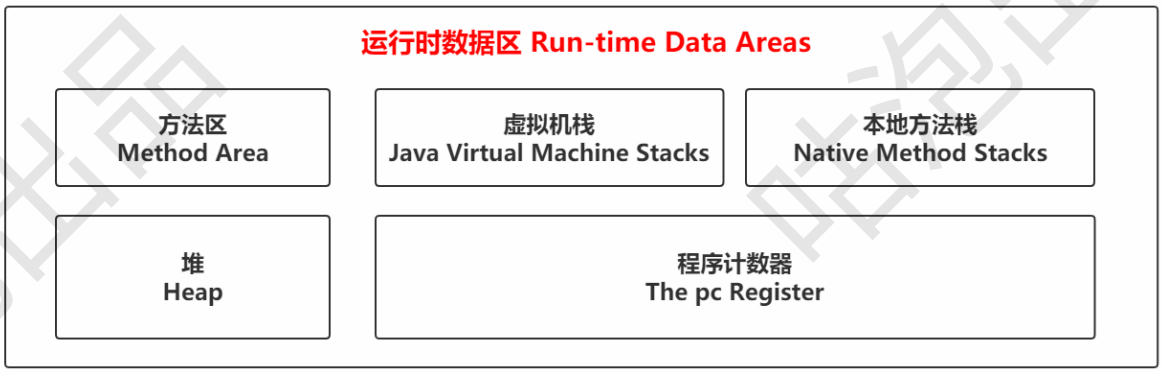
5.1 官网概括

https://docs.oracle.com/javase/specs/jvms/se8/html/index.html

> Summary

```
The Java Virtual Machine defines various run-time data areas that are used
during execution of a program. Some of these data areas are created on Java
Virtual Machine start-up and are destroyed only when the Java Virtual Machine
exits. Other data areas are per thread. Per-thread data areas are created when a
thread is created and destroyed when the thread exits.
```

5.2 图解



5.3 常规理解

5.3.1 Method Area(方法区)

方法区是各个线程共享的内存区域，在虚拟机启动时创建。

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。

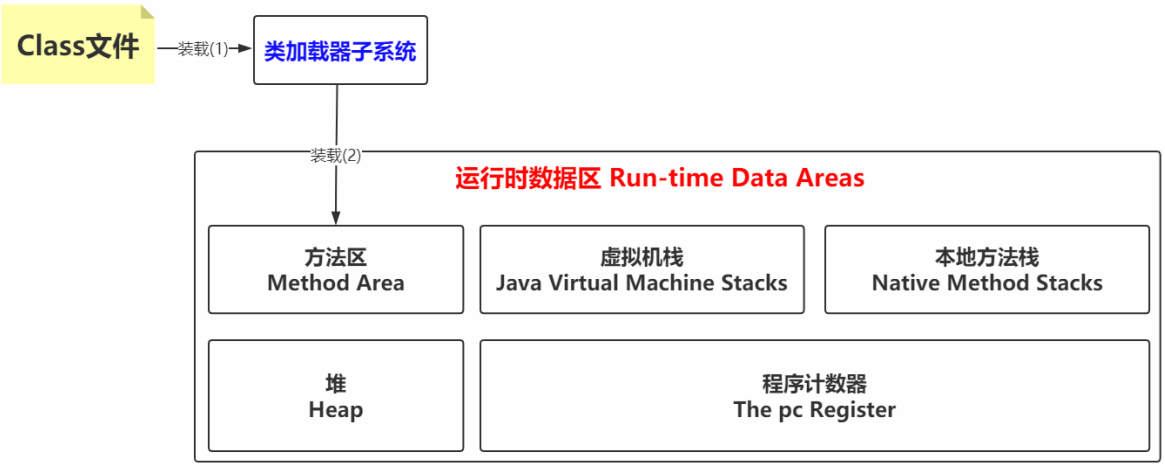
虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却又一个别名叫做Non-Heap(非堆)，目的是与Java堆区分开来。

当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。

```
The Java Virtual Machine has a method area that is shared among all Java Virtual
Machine threads.
The method area is created on virtual machine start-up.
Although the method area is logically part of the heap,......
If memory in the method area cannot be made available to satisfy an allocation
request, the Java Virtual Machine throws an OutOfMemoryError.
```

> 此时回看装载阶段的第2步：(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

如果这时候把从class文件到装载的第(1)和(2)步合并起来理解的话，可以画个图



> 值得说明的

(1)方法区在JDK 8中就是Metaspace，在JDK6或7中就是Perm Space

(2)Run-Time Constant Pool

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。

```
Each run-time constant pool is allocated from the Java Virtual Machine's method
area (§2.5.4).s
```

5.3.2 Heap(堆)

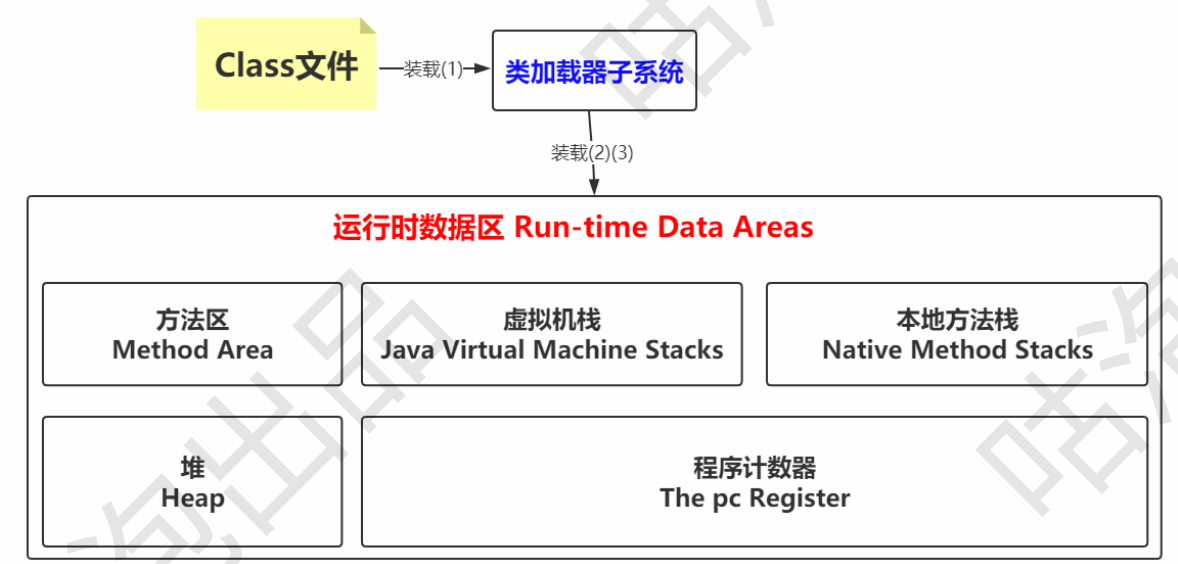
Java堆是Java虚拟机所管理内存中最大的一块，在虚拟机启动时创建，被所有线程共享。

Java对象实例以及数组都在堆上分配。

```
The Java Virtual Machine has a heap that is shared among all Java Virtual
Machine threads. The heap is the run-time data area from which memory for all
class instances and arrays is allocated.
The heap is created on virtual machine start-up.
```

> 此时回看装载阶段的第3步：(3)在Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口

此时装载(1)(2)(3)的图可以改动一下



5.3.3 Java Virtual Machine Stacks(虚拟机栈)

> 经过上面的分析，类加载机制的装载过程已经完成，后续的链接，初始化也会相应的生效。
>
> 假如目前的阶段是初始化完成了，后续做啥呢？肯定是Use使用咯，不用的话这样折腾来折腾去有什么意义？那怎样才能被使用到？换句话说里面内容怎样才能被执行？比如通过主函数main调用其他方法，这种方式实际上是main线程执行之后调用的方法，即要想使用里面的各种内容，得要以线程为单位，执行相应的方法才行。
>
> 那一个线程执行的状态如何维护？一个线程可以执行多少个方法？这样的关系怎么维护呢？

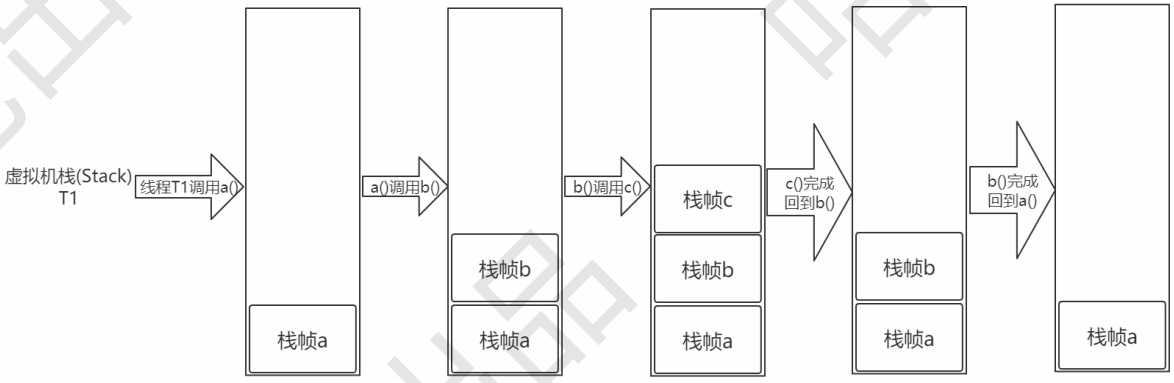
虚拟机栈是一个线程执行的区域，保存着一个线程中方法的调用状态。换句话说，一个Java线程的运行状态，由一个虚拟机栈来保存，所以虚拟机栈肯定是线程私有的，独有的，随着线程的创建而创建。

每一个被线程执行的方法，为该栈中的栈帧，即每个方法对应一个栈帧。

调用一个方法，就会向栈中压入一个栈帧；一个方法调用完成，就会把该栈帧从栈中弹出。

```
Each Java Virtual Machine thread has a private Java Virtual Machine stack,
created at the same time as the thread. A Java Virtual Machine stack stores
frames (§2.6).
```

画图理解栈和栈帧



5.3.4 The pc Register(程序计数器)

> 我们都知道一个JVM进程中有多个线程在执行，而线程中的内容是否能够拥有执行权，是根据CPU调度来的。
>
> 假如线程A正在执行到某个地方，突然失去了CPU的执行权，切换到线程B了，然后当线程A再获得CPU执行权的时候，怎么能继续执行呢？这就是需要在线程中维护一个变量，记录线程执行到的位置。

程序计数器占用的内存空间很小，由于Java虚拟机的多线程是通过线程轮流切换，并分配处理器执行时间的方式来实现的，在任意时刻，一个处理器只会执行一条线程中的指令。因此，为了线程切换后能够恢复到正确的执行位置，每条线程需要有一个独立的程序计数器(线程私有)。

如果线程正在执行Java方法，则计数器记录的是正在执行的虚拟机字节码指令的地址；

如果正在执行的是Native方法，则这个计数器为空。

```
The Java Virtual Machine can support many threads of execution at once (JLS §17). Each Java Virtual Machine thread has its own pc (program counter) register. At any point, each Java Virtual Machine thread is executing the code of a single method, namely the current method (§2.6) for that thread. If that method is not native, the pc register contains the address of the Java Virtual Machine instruction currently being executed. If the method currently being executed by the thread is native, the value of the Java Virtual Machine's pc register is undefined. The Java Virtual Machine's pc register is wide enough to hold a returnAddress or a native pointer on the specific platform.
```

5.3.5 Native Method Stacks(本地方法栈)

如果当前线程执行的方法是Native类型的，这些方法就会在本地方法栈中执行。