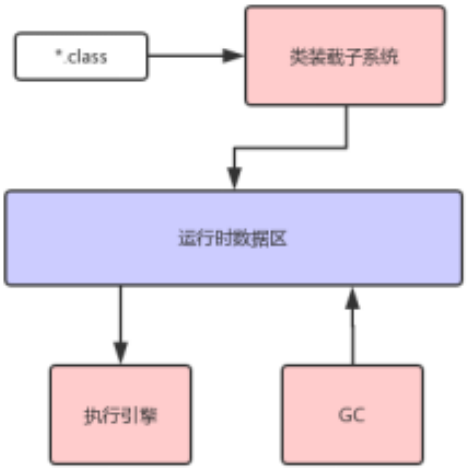
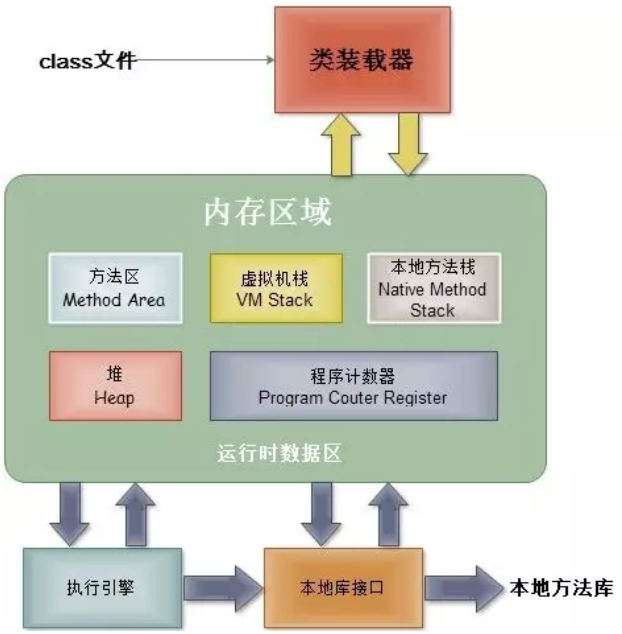
JVM主要组成

- 类加载子系统
- 执行引擎
- 垃圾回收子系统
- 运行时数据区

JVM运行时内存布局





加载

查找并加载类的二进制字节流数据。加载阶段完成三件事，①根据类的全名称限定名获取此类的二进制字节流。②将字节所流代表的这个静态结构转化为方法区的运行时数据结构。③生成该类的Class对象。

但Java虚拟机对这三点的规范并不算严格，并没有规定二进制字节流数据从哪获取，所以有很大的灵活性。比如：

1. 从网络红获取
2. 从ZIP中获取
3. 运行时生成

验证

确保class文件中包含的字节流符合当前虚拟机的要求，并且不会威胁虚拟机的安全。
验证阶段的工作量在虚拟机的类加载子系统中占了很大一部分。

验证包括四部分：文件格式验证、元数据验证、字节码验证、符号引用验证。

文件格式验证

- 开头是否为魔数
- 版本号是否符合要求
- 常量池内是否存在类型不符合要求的常量
- .....

元数据验证

对类的元数据进行语义校验，保证不存在不符合Java语言规范的元数据信息。

- 这个类是否有父类（除了Object，都应该有父类）
- 是否继承了不被允许继承的类（final修饰的类）
- 类中的字段、方法是否与父类产生了矛盾（覆盖了父类的final字段、不符合规则的重载）

字节码验证

保证其字节码指令都是安全的，在运行期间不会出现危害虚拟机安全的行为。

- 保证不会跳转到方法体以外的字节码指令上。

符号引用验证

此阶段验证发生在虚拟机将符号引用转为直接引用的过程中。这个转化动作将在解析阶段中发生。

- 是否可以通过全名称限定名找到对应的类
- 指定类中是否含有符合方法的字段描述符和简单名称所描述的方法和字段

准备

为类变量分配内存并设置初始值的阶段，这些内存都将在方法区中分配。但如果类字段的字段属性表中有ConstenValue，那么变量会被赋上ConstenValue指定的值。如：

```
public static final int a = 123;
```

解析

解析阶段是将常量池内的符号引用替换为直接引用的过程。虚拟机规范中**并未规定解析阶段的具体时间**，只要求了用于操作符号引用的字节码指令之前，先对他们所使用的符号引用进行解析。

解析动作主要针对类或接口、接口方法，字段、类方法、方法类型、方法句柄、调用点限定符。

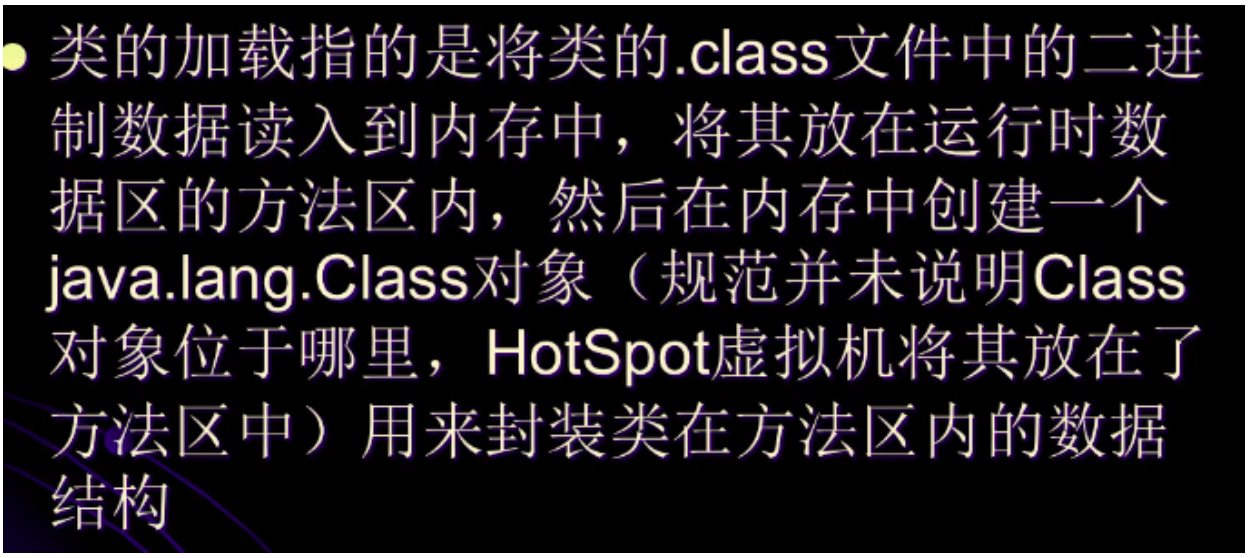
此阶段将符号引用转化为直接引用。

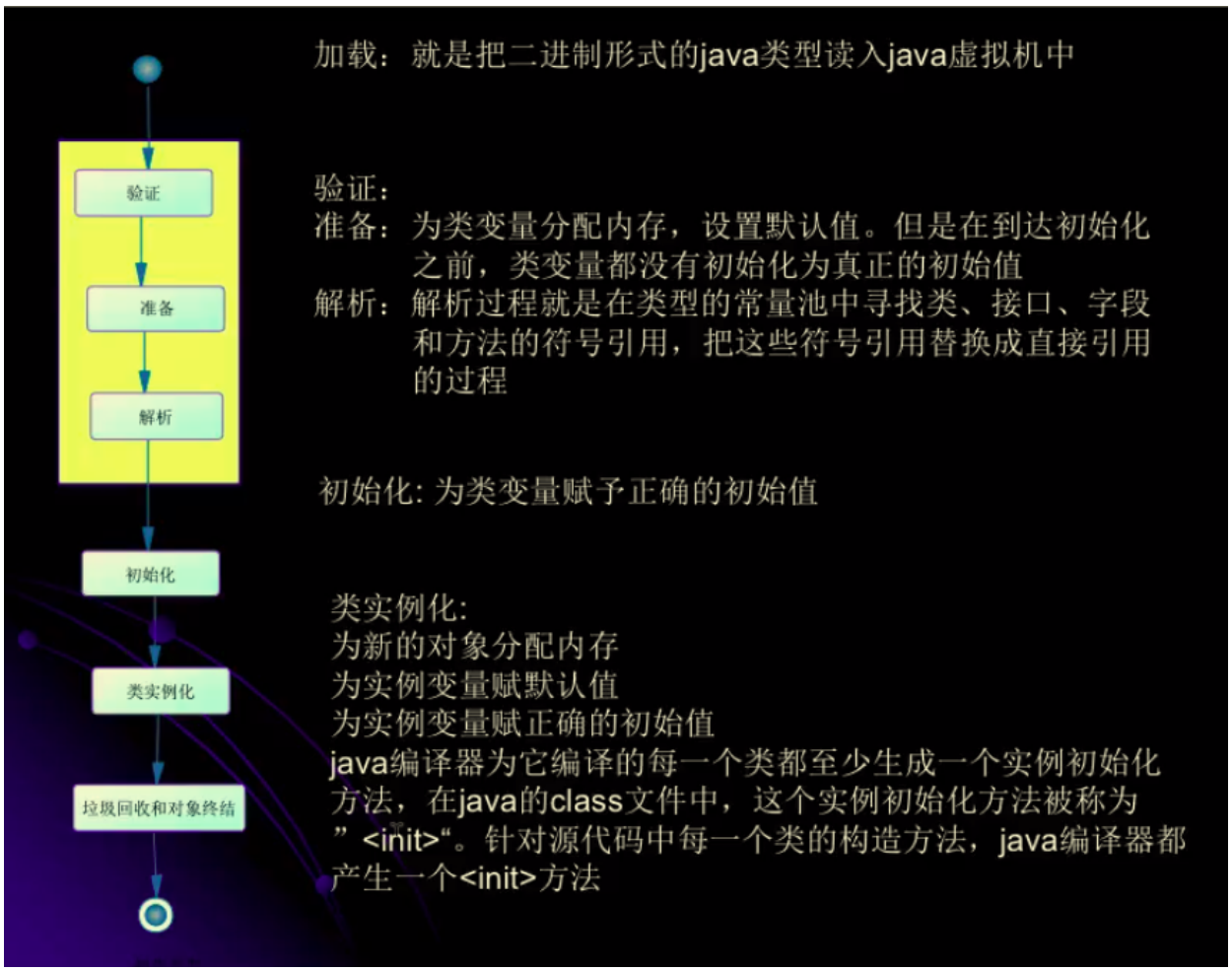
初始化

为类的静态变量赋予正确的初始值。会执行静态代码块(静态类属性)。

实例化：

- 为对象分配内存
- 实例变量赋默认值
- 为实例变量赋正确的值





根据上面规律，写出结果：

```
public class Test2 {
    public static void main(String[] args) {
        Test3 test3 = Test3.single();
        System.out.println(Test3.num1);//1
        System.out.println(Test3.num2);//0
    }
}

class Test3{
    public static int num1 = 0;
    private static Test3 test3 = new Test3();
    public static int num2 = 0;

    private Test3(){
        num1++;
        num2++;
    }

    public static Test3 single(){
        return test3;
    }
}
```

类的初始时机

- 当 java 虚拟机初始化一个类时，要求它的所有父类已经完成初始化。但此条规则不适用与接口。
- 在初始化一个类时，并不会先初始化它所实现的接口。
- 在初始化一个接口时，并不会先初始化它的 父接口。

注意，接口里的变量默认 public static final。

因此，一个父接口并不会因为它的子接口或者实现类的初始化而初始化，只有当程序首次使用特定接口的静态变量时，才会导致该接口的初始化。

练习

- 当一个常量的值在编译期不能确认的时候，这个值就不会放入调用类的常量池中。程序运行时，会导致主动使用这个常量的所在类，造成常量所在类的初始化。

```
public class Test1{
    public static void main(String[] args) {
        System.out.println(Test2.s1);//不会造成 Test2 初始化
```

```
        System.out.println(Test2.s2);//会造成 Test2 初始化
    }
}
class Test2{
    public final static int s1 = 1;
    public final static String s2 = new String("123");
   static {
        System.out.println("Test2 init");
    }
}
//注意，接口里的变量默认 public static final
```

- 对于静态字段来说，只有直接定义了该字段的类才会被初始化。 一个类在初始化时，要求其父类已完成初始化。

```
public class Test{
    public static void main(String[] args) {
        System.out.println(s1.str1);
    }
}

class f1{
    public static String str1 = "1";
    static {
        System.out.println("f1");
    }
}

class s1 extends f1{
    public static String str2 = "2";
    static {
        System.out.println("s1");
    }
}
```

类加载器

- 类加载器不必等到类首次使用时再加载该类。
- 加载时如果发现该类的class文件不存在，不会立马报错，而是在各类首次使用时才会报告错误(LinkageError)。如果该类一直没有被使用，类加载器就不会报告错误。

除了根加载器，其它的加载器都有且只有一个父加载器。

自定义类加载器只需要覆写findClass()方法即可。

```
int[] ints = new int[2]
sout(ints.getClass().getClassLoader)// 1.null

String[] str = new int[2]
sout(str.getClass().getClassLoader)// 2.null
```
1: 原生类型的classloader是null
2：启动类加载器在Hotspot中是null

ClassLoader loadClass 源码

Subclass of ClassLoader are encouraged overridefind Class method, rather than this method.

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class<?> c = findLoadedClass(name);//检查是否已被加载
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) {//会一直向上尝试用父类加载器加载
                    c = parent.loadClass(name, false);
                } else {//父级到顶都不能加载，尝试使用BootStrap类加载器加载
                    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
                // to find the class.
                long t1 = System.nanoTime();
                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) {
            resolveClass(c);
```

```
        }
        return c;
    }
}
```

JVM类加载机制

1. 全盘负责：一个类负责加载某个Class时，该Class所依赖的类也由该类加载器尝试加载。（也就是，从该类加载器往上委托）
2. 双亲委托：
3. 缓存机制：保证所有加载过的Class都会被缓存。当程序需要使用某个Class时，先从缓存中查找，找不到才会读取2进制数据进行加载，加载完后丢进缓存。这也就是为什么修改Class后需要重新编译的原因。

命名空间：

- 每个类加载器都有自己的命名空间，命名空间是由该加载器及所有父加载器加载的类构成的
- 同一命名空间中一个类不会被重复加载
- 不同命名空间中，类可以被重复加载
- 子加载器可以看到父加载器加载的类，父加载器看不到子加载器加载的类

类的卸载：

由Java虚拟机自带类加载器加载的类，在虚拟机生命周期内，不会卸载。Java虚拟机自带的类加载器有根加载器、扩展类加载器、系统类加载器。Java虚拟机本身会始终引用这些类加载器，而这些类加载器始终引用被它们加载的类的Class对象，因此，这些class对象始终是可以被触及的。

由用户自定义的类加载器加载的类是可以被卸载的。

System.gc()

```
System.out.println(System.getProperty("sun.boot.class.path"));
System.out.println(System.getProperty("java.ext.dirs"));
System.out.println(System.getProperty("java.class.path"));
```

自己指定系统类加载器

java -Djava.system.class.loader=MyClassLoader.MyClassLoader Test1

指定MyClassLoader.MyClassLoader为系统类加载器，并运行Test1

```
scl = AccessController.doPrivileged(new SystemClassLoaderAction(scl));

class SystemClassLoaderAction
    implements PrivilegedExceptionAction<ClassLoader> {
    private ClassLoader parent;

    //parent 就是系统类加载器
    SystemClassLoaderAction(ClassLoader parent) {
        this.parent = parent;
    }

    public ClassLoader run() throws Exception {
        String cls = System.getProperty("java.system.class.loader");
        if (cls == null) {//如果没自定义系统类加载器，直接返回parent
            return parent;
        }

        //如果指定了系统类加载器
        //获取指定的系统类加载器的参数为ClassLoader.class 的构造方法，
        Constructor<?> ctor = Class.forName(cls, true, parent)
            .getDeclaredConstructor(new Class<?>[] { ClassLoader.class });
        //以系统类加载器为父加载器加载这个加载器
        ClassLoader sys = (ClassLoader) ctor.newInstance(
            new Object[] { parent });
        //将指定的系统类加载器设置为线程上下文加载器
        Thread.currentThread().setContextClassLoader(sys);
        return sys;
    }
}
```

上面就是 ClassLoader.getSystemClassLoader() 文档中下面这句话的原因

If the system property "java.system.class.loader" is defined when this method is first invoked then the value of that property is taken to be the name of a class that will be returned as the system class loader. The class is loaded using the default system class loader and must define a public constructor that takes a single parameter of type ClassLoader which is used as the delegation parent.

SPI 线程上线文类加载器

双亲委托机制在父类加载器加载的类中访问子类加载器加载的类时会出现问题，比如JDBC。JDBC中规定，Driver(数据库驱动)必须向 DriverManage 注册自己，而

DriverManage 是BootStrapClassloader加载的，所以DriverManage 中是无法加载到具体的Driver。

具体数据库的 Driver

```
static {
    try {
        //会向DriverManager注册自己，注册时会先完成DriverManager的加载和初始化
        DriverManager.registerDriver(new Driver());
    } catch (SQLException var1) {
        throw new RuntimeException("Can't register driver!");
    }
}
```

DriverManager 的初始化

```
static {
    loadInitialDrivers();
    println("JDBC DriverManager initialized");
}

private static void loadInitialDrivers() {
    ......
    AccessController.doPrivileged(new PrivilegedAction<Void>() {
        public Void run() {
            //下面两行 打破了双亲委托机制
            ServiceLoader<Driver> loadedDrivers = ServiceLoader.load(Driver.class);
            Iterator<Driver> driversIterator = loadedDrivers.iterator();
            try{
                while(driversIterator.hasNext()) {
                    driversIterator.next();//此时会加载具体Class
                }
            } catch(Throwable t) {
            }
            return null;
       }
    });
    .......
}
```

上面两行可以转换为：

```
//只需要下面两行代码，便会加载具体的mysql Driver
ServiceLoader<Driver> loader = ServiceLoader.load(Driver.class);
Iterator<Driver> iterator = loader.iterator();

System.out.println(iterator.hasNext());
```

```
if(iterator.hasNext()){
    Driver driver = iterator.next();
    sout(driver.getClass());
}
```

ServiceLoader.load() 方法 （重点）

```
//ServiceLoader是由BootStrap Classloader 加载的，所以类中引用的其它类
//也会由BootStrap 尝试去加载，而BootStrap 已是最顶层的类加载器，不会再向上委托，
//如果它也不能加载，那就加载不到类了。
public static <S> ServiceLoader<S> load(Class<S> service) {
    //ServiceLoader中会尝试用BootStrap 加载具体的Mysql Driver，
    //但ServiceLoader中是不可见的，这样就无法加载。
    //所以取出当前线程的上下文类加载器即appCL，用于后面加载具体的Mysql Driver
    //即打破了双亲委托机制
    ClassLoader cl = Thread.currentThread().getContextClassLoader();
   return ServiceLoader.load(service, cl);
}
public static <S> ServiceLoader<S> load(Class<S> service,ClassLoader loader){
    return new ServiceLoader<>(service, loader);
}
private ServiceLoader(Class<S> svc, ClassLoader cl) {
    service = Objects.requireNonNull(svc, "Service interface cannot be null");
    //loader 为ServiceLoader的私有常量，在后面加载具体实现类时会用该加载器进行加
载。
    // loader 在构造方法内赋了值，即为上文取到的线程上下文类加载器。
    loader = (cl == null) ? ClassLoader.getSystemClassLoader() : cl;
    acc = (System.getSecurityManager() != null) ? AccessController.getContext() :
null;
    reload();
}
```

ServiceLoaderd将类的加载延迟到了DriverManager调用的时候（也就是所谓的Lazy）。即

```
//不会加载Class，只是找出需要加载的Class，存到LazyIterator lookupIterator
ServiceLoader<Driver> loader = ServiceLoader.load(Driver.class);
```

调用 driversIterator.next() ——>lookupIterator.next() ——> nextService 此时就会根据驱动名字具体实例化各个实现类了。