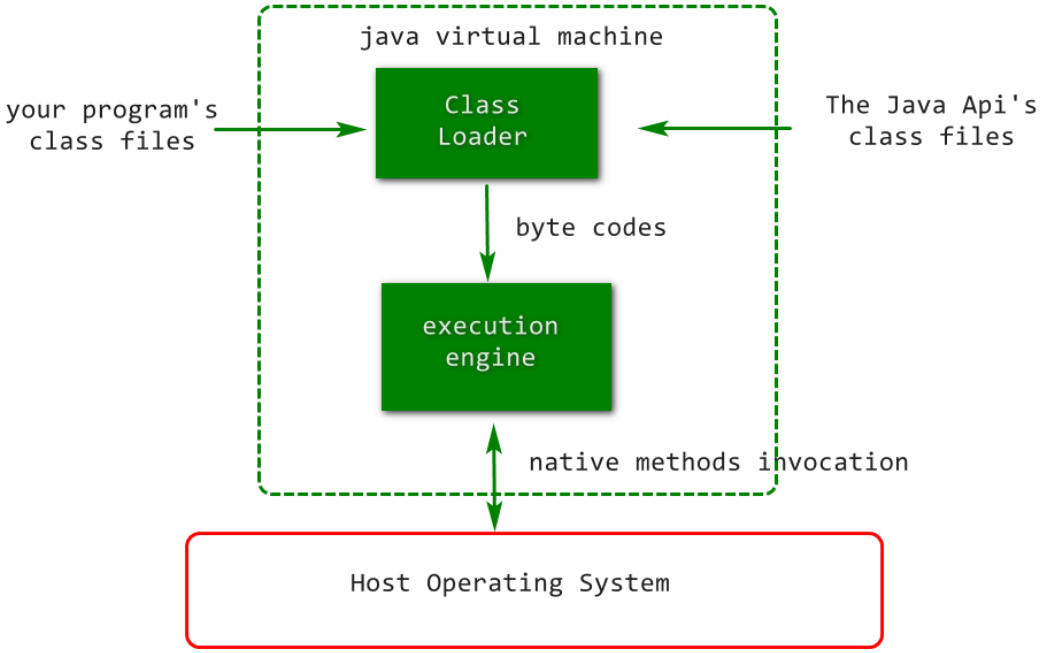
JVM 模块
类加载增强分析

1.类加载概要分析

1.1. 类加载概述

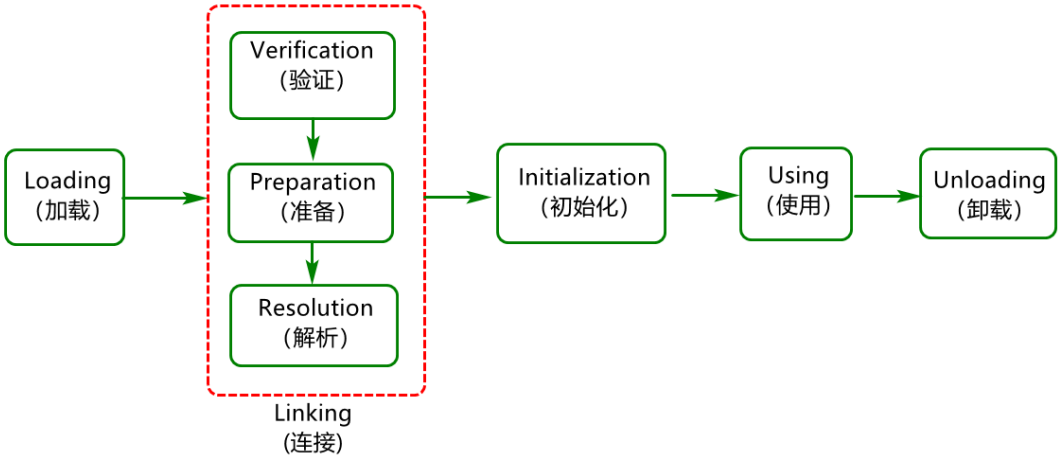
所谓类加载就是将类从磁盘或网络读到 JVM 内存，然后交给执行引擎执行。



说明：学习类加载有助于我们更深入地理解 JAVA 类成员的初始化过程，运行过程。并可以为后续的线上问题的解决及调优提供一种基础保障。

1.2. JAVA 类生命周期分析

类的生命周期指的是从加载到卸载的基本过程，此过程包含 7 个阶段，如下图所示：



说明：一个已经加载的类被卸载的几率很小，至少被卸载的时间是不确定的，假如需要卸载的话可尝试 System.exit(0);

2.类加载进阶分析

2.1. 加载分析（loading）

我们知道类的加载过程中大致可分为加载、验证、准备、解析、初始化几大阶段，但这几个阶段的执行顺序又是怎样的呢？JVM 规范中是这样说的：

1)加载、验证、准备和初始化发生的顺序是确定的，而解析阶段则不一定.

2)加载、验证、准备和初始化这四个阶段按顺序开始不一定按顺序完成。

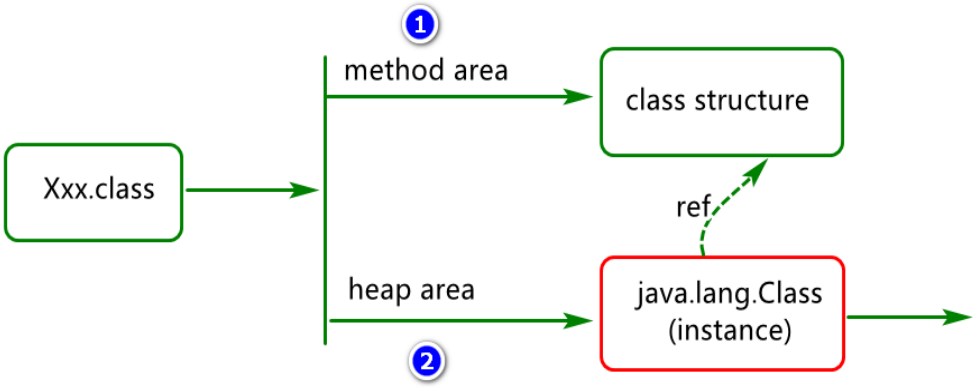
2.1.1. 加载基本步骤分析

1）通过一个类的全限定名（类全名）来获取其定义的二进制字节流。

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。



2.1.2. 加载路径分析

JVM 从何处加载我们要使用的类呢？主要从如下三个地方：

1）JDK 类库中的类(lib\jar,lib\ext)

2）第三方类库中的类

3）应用程序类库中的类

2.1.3. 加载方式及时机分析

JVM 中的类加载方式主要两种：隐式加载和显式加载

1）隐式加载

a）访问类的静态成员(例如类变量，静态方法)

b）构建类的实例对象(例如使用 new 关键字构建对象或反射构建对象)

c）构建子类实例对象（构建类的对象时首先会加载父类类型)

2)显式加载

a) ClassLoader.loadClass(..)
b) Class.forName(…)

3）代码分析：

```
class ClassA{
    static {
        System.out.println("ClassA");
    }
}
public class TestClassLoader02 {

    public static void main(String[] args)throws Exception {

        //ClassLoader loader=TestClassLoader02.class.getClassLoader();
        //loader.loadClass("cgb.java.jvm.loader.ClassA");
        Class.forName("cgb.java.jvm.loader.ClassA");
        //Class.forName("cgb.java.jvm.loader.ClassA", true, loader);
    }

}
```

说明:

1）通过 ClassLoader 对象的 loadClass 方法加载类不会执行静态代码块。

2）可通过指定运行参数，查看类的加载顺序。

```
-XX:+TraceClassLoading
```

2.2. 连接分析（linking）

2.2.1. 验证（Verification）

这一阶段的目的是为了确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

验证阶段大致会完成 4 个阶段的检验动作:

1）文件格式的验证

2）元数据验证

3）字节码合法性验证

4）符号引用验证（Class 文件中以 CONSTANT_Class_info、CONSTANT_Fieldref_info 等常量形式出现）

说明：验证阶段是非常重要的，但不是必须的，它对程序运行期没有影响，如果

所引用的类经过反复验证，那么可以考虑采用-Xverify:none 参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

2.2.2. 准备（Preparation）

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。

1）类变量（static）内存分配
2）按类型进行初始默认值分配（如 0、0L、null、false 等）。

例如：
假设一个类变量的定义为：public static int value = 3；
那么变量 value 在准备阶段过后的初始值为 0，而不是 3，把 value 赋值为 3 的动作将在初始化阶段才会执行。

3）如果类字段的字段属性表中存在 ConstantValue 属性，即同时被 final 和 static 修饰，那么在准备阶段变量 value 就会被初始化为 ConstValue 属性所指定的值。

例如：
假设上面的类变量 value 被定义为： public static final int value = 3；
编译时 Javac 将会为 value 生成 ConstantValue 属性，在准备阶段虚拟机就会根据 ConstantValue 的设置将 value 赋值为 3

2.2.3. 解析（Resolution）

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的过程，其中：
1)符号引用：就是一组符号（例如 CONSTANT_Fieldref_info）来描述目标，可以是任何字面量。
2)直接引用：就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

说明：相同的符号引用不同 JVM 机器上对应的直接引用可能不同，直接引用一般对应已加载到内存中的一个具体对象。

2.3. 初始化分析（Initialization）

此阶段为类加载的最后一个阶段，这个阶段我们让自定义类加载器参与进来，其余阶段完全由 JVM 主导。例如 JVM 负责对类进行初始化，主要对类变量进行初始化。

在 Java 中，对类变量进行初始值的设定有两种方式：

1)声明类变量时指定初始值。
2)使用静态代码块为类变量指定初始值

说明：只有当对类的主动使用的时候才会导致类的初始化

Java 程序对类的使用方式可以分为两种：

主动使用：会执行加载、连接、初始化静态域
被动使用：只执行加载、连接，不初始化类静态域

如何理解被动使用呢?

如通过子类引用父类的静态字段，为子类的被动使用，不会导致子类初始化，例如：

```
class A{
    public static int a=10;
    static {
        System.out.println("A.a="+a);
    }
}
class B extends A{
    static {
        System.out.println("B");
    }
}
```

```
public class TestClassLoader03 {
    public static void main(String[] args) {
        System.out.println(B.a);
    }
}
```

当通过 B 对象访问 A 类的 a 属性时不会执行 B 类的静态代码块。

3.类加载器应用分析

3.1. 类加载器概要分析

3.1.1. 类加载器简介

类加载器是在类运行时负责将类读到内存的一个对象，其类型为 ClassLoader 类型，此类型为抽象类型，通常以父类形式出现。

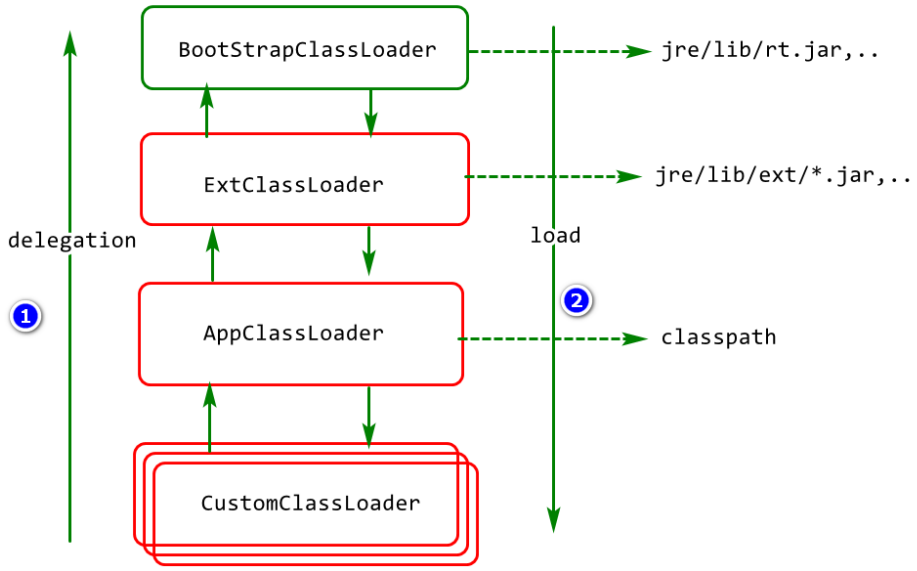
类加载器对象常用方法说明：

1) getParent() 返回类加载器的父类加载器。
2) loadClass(String name)加载名称为 name 的类.
3) findClass(String name)查找名称为 name 的类.
4) findLoadedClass(String name) 查找名称为 name 的已经被加载过的类
5) defineClass(String name, byte[] b, int off, int len) 把字节数组 b 中的内容转换成 Java 类。
6) ……

3.1.2. 类加载器的层次架构

Java 中类加载器大致可分为两种，一类是系统提供，一类是自己定义，其层级

结构如下图所示：



类加载器加载类的过程分析：

1）首先会查看自己是否已加载过此类，有则返回，没有则将加载任务委托给父类(parent)加载器加载，依次递归。
2）父类加载器无法完成此加载任务时，自己去加载。
3）自己也无法完成加载时就会抛出异常。

说明：类加载时首先委托父类加载的这种机制称之为双亲委派机制。基于这种机制实现了类加载时的优先级层次关系，同时也可以保证同一个类只被一个加载器加载（例如 Object 类只会被 BootstrapClassLoader 加载），这样更有利于 java 程序的稳定运行。

代码分享：获取类的加载器对象

```
    public static void doMethod01() {
          ClassLoader loader=ClassLoader.getSystemClassLoader();
          System.out.println(loader);
          System.out.println(loader.getParent());
          System.out.println(loader.getParent().getParent());
    }
```

```
    public static void doMethod02() {
        ClassLoader loader =Thread.currentThread().getContextClassLoader();
```

```
        System.out.println(loader);
        System.out.println(loader.getParent());
        System.out.println(loader.getParent().getParent());
    }
```

3.2. 自定义类加载器

JVM 自带的类加载器只能加载默认 classpath 下的类，如果我们需要加载应用程序之外的类文件呢？比如网络上的某个类文件，这种情况一般就要使用自定义加载器了。自定义类加载器可以自己指定类加载路径，可以实现系统在线升级（热替换）等操作。在我们使用的 tomcat 服务器，spring 框架，mybatis 框架等其内部都自己定义了类加载器。

我们如何自己定义类加载器呢？我们自己写类加载器一般需要直接或间接继承 ClassLoader 类，然后重写相关方法，具体过程可参考后续小节内容。

3.2.1. 准备工作

在指定包中创建一个自己写的类，例如：

```
package pkg;
public class Search {
     static {
        System.out.println("search static");
     }
     public Search() {
         System.out.println("search constructor");
     }
}
```

说明：后续可将此类拷贝到指定目录，由自己指定的类加载器进行加载。

3.2.2. 基于 ClassLoader 创建

代码实现：

```
class MyClassLoader01 extends ClassLoader {
    private String baseDir;
```

```
    public MyClassLoader01(String baseDir) {
        this.baseDir=baseDir;
    }
    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException
{
        byte[] classData = loadClassBytes(name);
        if (classData == null) {
            throw new ClassNotFoundException();
        } else {
            return defineClass(name, classData, 0, classData.length);
        }
    }
    /**自己定义*/
    private byte[] loadClassBytes(String className) {//pkg.Search
        String fileName =baseDir+className.replace('.', File.separatorChar)
+ ".class";
        System.out.println("fileName="+fileName);
        InputStream ins=null;
        try {
            ins= new FileInputStream(fileName);
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            int bufferSize = 1024;
            byte[] buffer = new byte[bufferSize];
            int length = 0;
            while ((length = ins.read(buffer)) != -1) {
                baos.write(buffer, 0, length);
            }
            return baos.toByteArray();
        } catch (IOException e) {
             e.printStackTrace();
             throw new RuntimeException(e);
         }finally {
        if(ins!=null)try{ins.close();}catch(Exception e) {}
        }
    }
}
```

说明:自己写类加载器一般不建议重写 loadClass 方法,当然不是不可以重写。

定义测试方法:假如使用自定义类加载器加载我们指定的类,要求被加载的类应与当前类不在同一个命名空间范围内,否则可能直接使用 AppClassLoader 进行类加载。

```
public class TestMyClassLoader01 {
    public static void main(String[] args) throws Exception{
         String baseDir="F:\\WORKSPACE\\";
         MyClassLoader01 classLoader = new MyClassLoader01(baseDir);
         //此类不要和当前类放相同目录结构中
         String pkgCls="pkg.Search";
         Class<?> testClass = classLoader.loadClass(pkgCls);
         Object object = testClass.newInstance();
         System.out.println(object.getClass());
         System.out.println(object.getClass().getClassLoader());
    }
}
```

输出的类加载名称应该为我们自己定义的类加载器名称。

3.2.3. 基于 URLClassLoader 创建

URLClassLoader 继承 ClassLoader，可以从指定目录，jar 包，网络中加载指定的类资源。

代码示例：基于此类加载器可加载网络中的

```
class MyClassLoader02 extends URLClassLoader {
    public MyClassLoader02(URL[] urls) {
        super(urls,null);// 指定父加载器为 null
    }
}
```

编写测试类

```
public class TestMyClassLoader02 {
    public static void main(String[] args)throws Exception {
        File file=new File("f:\\workspace\\");
        //File to URI
        URI uri=file.toURI();
        URL[] urls={uri.toURL()};
        ClassLoader classLoader = new MyClassLoader02(urls);
        Class<?> cls = classLoader.loadClass("pkg.Search");
        System.out.println(classLoader);
        Object obj = cls.newInstance();
        System.out.println(obj);
```

```
    }
}
```

3.3. 基于类加载器实现热替换

当我们的项目运行时假如需要实现在线升级（也就是常说的热替换），可以通过自定义类加载实现，例如

自定义类加载器

```
class MyClassLoader03 extends ClassLoader {
    private String basedir; // 需要该类加载器直接加载的类文件的基目录
    private HashSet<String> loadClasses; // 需要由该类加载器直接加载的类名

    public MyClassLoader03(String basedir, String[] classes)
        throws IOException {
        // 指定父类加载器为 null，打破双亲委派原则
        super(null);
        this.basedir = basedir;
        loadClasses = new HashSet<String>();
        customLoadClass(classes);
    }
 // 获取所有文件完整路径及类名，刷入缓存
    private void customLoadClass(String[] classes) throws IOException {
        for (String classStr : classes) {
            loadDirectly(classStr);
            loadClasses.add(classStr);
        }
    }
    // 拼接文件路径及文件名
    private void loadDirectly(String name) throws IOException {
        StringBuilder sb = new StringBuilder(basedir);
        String classname = name.replace('.', File.separatorChar) + ".class";
        sb.append(File.separator).append(classname);
        File classF = new File(sb.toString());
        instantiateClass(name,new FileInputStream(classF),
                classF.length());
    }
    // 读取并加载类
    private void instantiateClass(String name, InputStream fin, long len)
        throws IOException {
        byte[] raw = new byte[(int) len];
```

```
        fin.read(raw);
        fin.close();
        defineClass(name, raw, 0, raw.length);
    }
    @Override
    protected Class<?> loadClass(String name, boolean resolve)
            throws ClassNotFoundException {
        Class<?> cls;
        // 判断是否已加载(在名字空间中寻找指定的类是否已存在)
        cls = findLoadedClass(name);
        if(!this.loadClasses.contains(name) && cls == null)
            cls = getSystemClassLoader().loadClass(name);
        if (cls == null)throw new ClassNotFoundException(name);
        if (resolve)resolveClass(cls);
        return cls;
    }
}
```

编写测试类

```
public class TestMyClassLoader03 {
    public static void main(String[] args) throws Exception {
        MyClassLoader03 loader=new MyClassLoader03("f:\\workspace\\",
            new String[] {"pkg.Search"});
        Class<?> cls = loader.loadClass("pkg.Search");
        System.out.println(cls.getClassLoader());
        Object search = cls.newInstance();
        System.out.println(search);
        Thread.sleep(20000);
        loader=new MyClassLoader03("f:\\workspace\\",
            new String[] {"pkg.Search"});
        cls=loader.loadClass("pkg.Search");
        System.out.println(cls.getClassLoader());
        search = cls.newInstance();
        System.out.println(search);
    }
}
```

说明：程序可运行期间可以将不需要的目标类删除，然后将新的目标类放到原先的地方，以实现热替换操作。