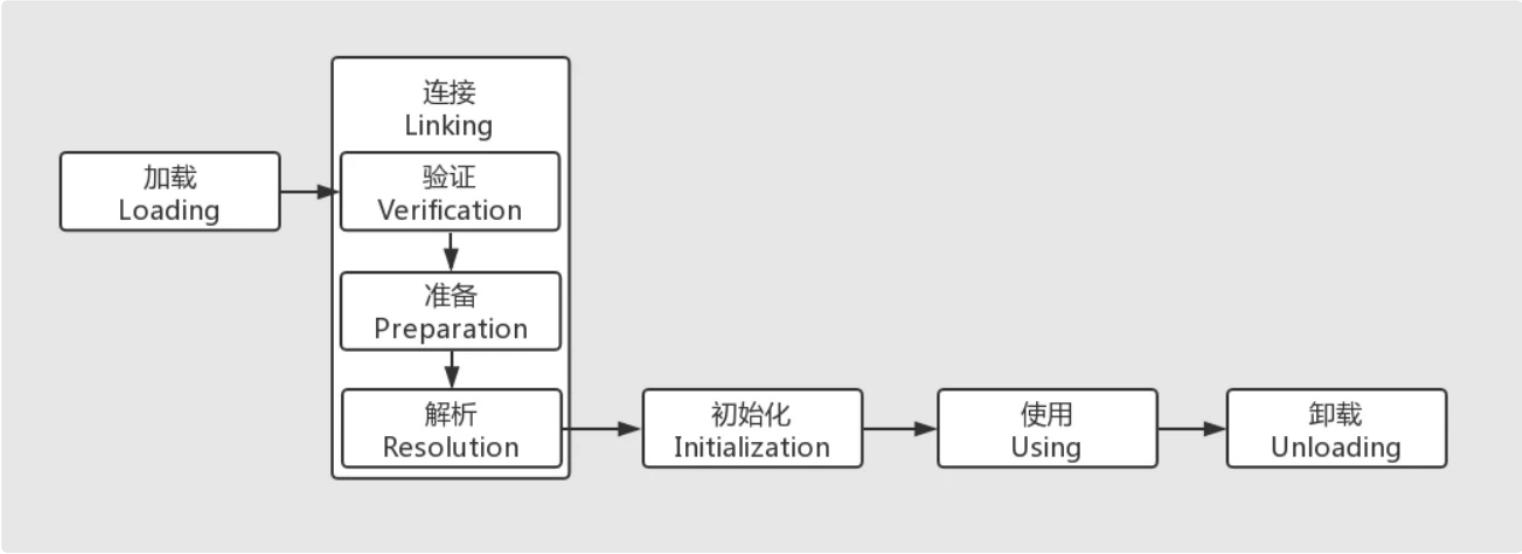


类加载分析

类的生命周期如下



Person.java

Java

```
package com.atao;

public class Person {
    public String name;
    public static int age;

    public Person(){
        System.out.println("调用无参构造方法");
    }

    public Person(String name){
        this.name = name;
        System.out.println("调用有参构造方法");
    }

    public static void staticAction(){
        System.out.println("调用静态方法");
    }

    public void Action(){
        System.out.println("调用方法");
    }

    {
        System.out.println("构造代码块");
    }

    static {
        System.out.println("静态代码块");
    }
}
```

例 1

Java

```
package com.atao;

public class Demo1 {
    public static void main(String[] args) throws Exception {
        new Person("atao");
    }
}
/*
静态代码块
构造代码块
调用有参构造方法
*/
```

这里直接 new 一个 Person 实例，可以看到它的调用顺序为 静态代码块 → 构造代码块 → 调用构造方法

例 2

Java

```
package com.atao;

public class Demo1 {
    public static void main(String[] args) throws Exception {
        // 1.
        Person.staticAction();

        // 2.
        Person.age =10;
    }
}

/*
1.
静态代码块
调用静态方法
*/

/*
2.
静态代码块
*/
```

在1和2代码中仅选择一个执行，可知对于类在初始化时会执行静态代码块，而对于构造代码块由于没有执行实例化操作并不会执行

例 3

```
package com.atao;

public class Demo1 {
    public static void main(String[] args) throws Exception {
        Class c = Person.class;
    }
}

/*

*/
```

这里在仅加载类，没有做其他操作，并没有返回任何内容，可以知道在加载类时不执行类中代码

例 4

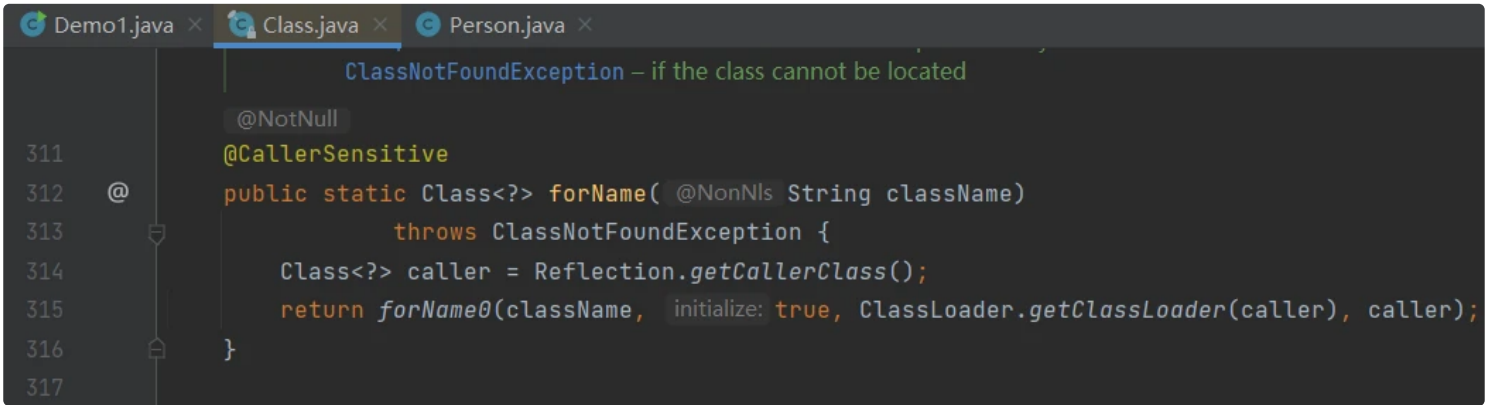
```
package com.atao;

public class Demo1 {
    public static void main(String[] args) throws Exception {
        Class.forName("com.atao.Person");
    }
}

/*
静态代码块
*/
```

利用 Class.forName 方法对类进行加载会类初始化

这里跟进 Class.forName 方法分析一下



跟进后，发现 Class 类中 forName 方法是有多多的，如果我们仅传入 String 变量，那它就直接调用上图这个方法，接着调用 forName0 方法，forName0 方法是一个底层方法就不继续跟进了。

但是 forName0 方法的第二个参数为 initialize，应该是一个和初始化有关的参数，默认为 true，这里我们利用 Class.forName 的另外一个方法进行尝试

```
Demo1.java x Class.java x Person.java x
@CallerSensitive
public static Class<?> forName( @Nonnull String name, boolean initialize,
                               @Nullable ClassLoader loader)
    throws ClassNotFoundException
{
    Class<?> caller = null;
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        // Reflective call to get caller class is only needed if a security manager
        // is present. Avoid the overhead of making this call otherwise.
        caller = Reflection.getCallerClass();
        if (loader == null) {
            ClassLoader ccl = ClassLoader.getClassLoader(caller);
            if (ccl != null) {
                sm.checkPermission(
                    SecurityConstants.GET_CLASSLOADER_PERMISSION);
            }
        }
    }
    return forName0(name, initialize, loader, caller);
}
```

```
package com.atao;

public class Demo1 {
    public static void main(String[] args) throws Exception {

        ClassLoader cl = ClassLoader.getSystemClassLoader();
        Class.forName("com.atao.Person", false, cl);
    }
}
/*
```

```
package com.atao;

public class Demo1 {
    public static void main(String[] args) throws Exception {
        ClassLoader cl = ClassLoader.getSystemClassLoader();
        cl.loadClass("com.atao.Person");
    }
}
```

使用ClassLoader类加载器对类进行加载，也不会进行类的初始化调用

插曲:调试类加载的过程

这里使用的环境：java version "1.8.0_191"

```
URLClassLoader.java x ClassLoader.java x Launcher.class x ClassNotFoundException.java x Demo.java x Ref
1 public class Demo {
2     public static void main(String[] args) throws Exception { args: []
3         ClassLoader cl = ClassLoader.getSystemClassLoader(); cl: Launcher$AppClassLoader@473
4         cl.loadClass(name: "Person"); cl: Launcher$AppClassLoader@473
5     }
6 }
```

最开始是使用了App加载器进行加载，跟进

```
URLClassLoader.java x ClassLoader.java x Launcher.class x ClassNotFoundException.java x Demo.java x Reflec
Throws: ClassNotFoundException – If the class was not found
356 public Class<?> loadClass(String name) throws ClassNotFoundException { name: "Person"
357     return loadClass(name, resolve: false); name: "Person"
358 }
359
```

这里由于AppClassLoader类没有loadClass(String name)这个方法，于是调用了父类的ClassLoader.loadClass方法，接着会调用AppClassLoader.loadClass(String var1, boolean var2)方法

```

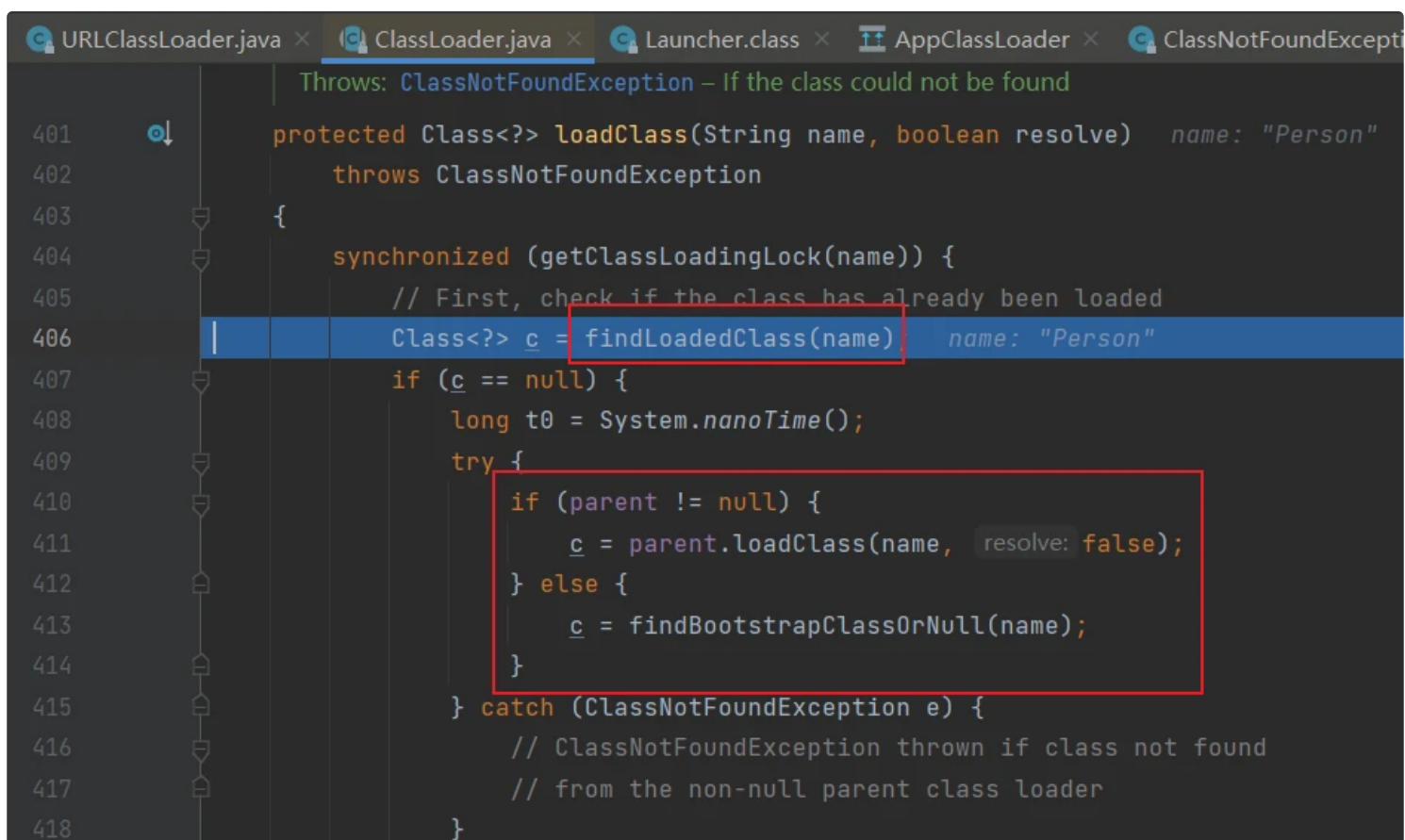
public Class<?> loadClass(String var1, boolean var2) throws ClassNotFoundException {
    int var3 = var1.lastIndexOf('ch: 46');    var1: "Person"
    if (var3 != -1) {
        SecurityManager var4 = System.getSecurityManager();
        if (var4 != null) {
            var4.checkPackageAccess(var1.substring(0, var3));
        }
    }

    if (this.ucp.knownToNotExist(var1)) {
        Class var5 = this.findLoadedClass(var1);
        if (var5 != null) {
            if (var2) {
                this.resolveClass(var5);
            }

            return var5;
        } else {
            throw new ClassNotFoundException(var1);
        }
    } else {
        return super.loadClass(var1, var2);
    }
}

```

继续往下走是调用它父类的loadClass方法

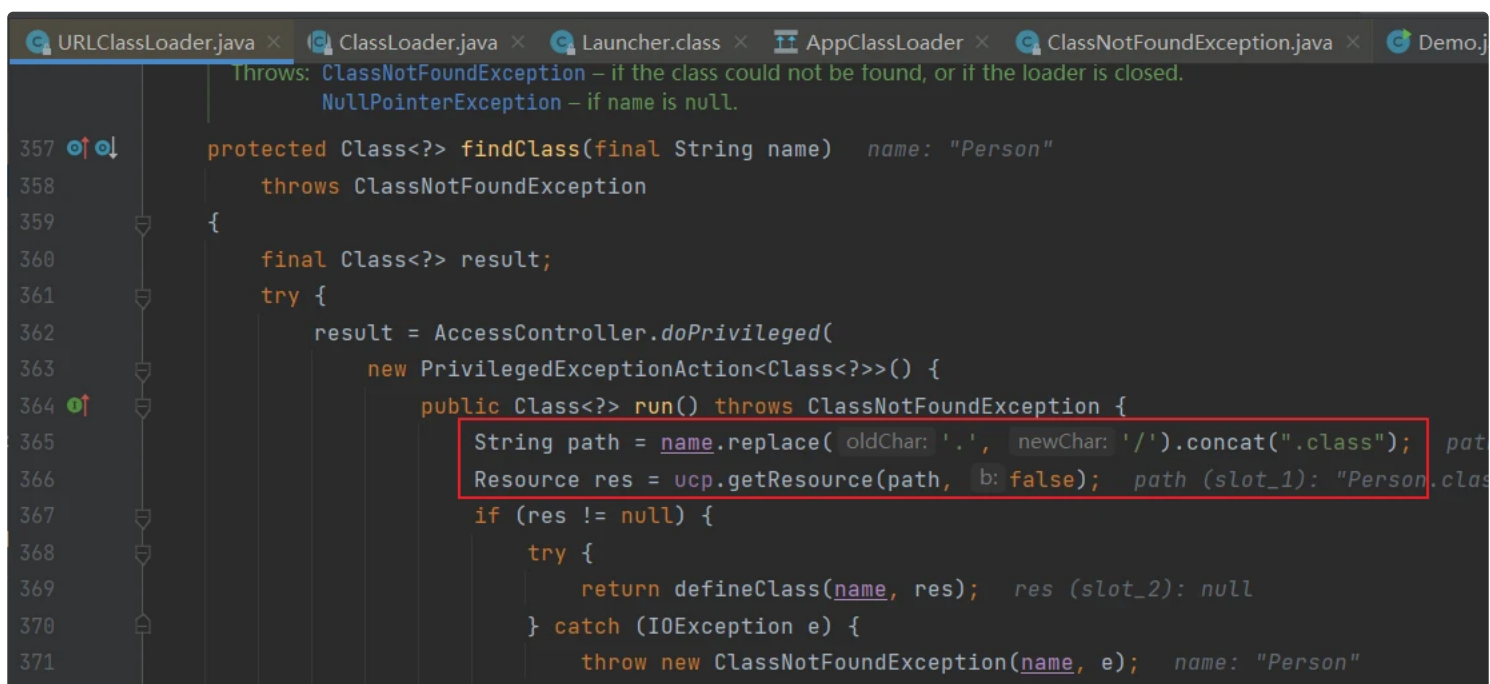


```

protected Class<?> loadClass(String name, boolean resolve)    name: "Person"
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class<?> c = findLoadedClass(name)    name: "Person"
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
                    c = parent.loadClass(name, resolve: false);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }
        }
    }
}

```

在这里调用了findLoadedClass先判断父类加载器是否有加载过这个类。如果没有的话，就会继续调用它的父类查看，这是Java中的双亲委派（不断向上询问是否加载，如果都没有加载，则自己进行加载）这里会查看AppClassLoader父类加载器ExtClassLoader，最后调用Bootstrap加载器



```

protected Class<?> findClass(final String name)    name: "Person"
    throws ClassNotFoundException
{
    final Class<?> result;
    try {
        result = AccessController.doPrivileged(
            new PrivilegedExceptionAction<Class<?>>() {
                public Class<?> run() throws ClassNotFoundException {
                    String path = name.replace('oldChar: '.', 'newChar: '/').concat(".class");    pat
                    Resource res = ucp.getResource(path, b: false);    path (slot_1): "Person.clas
                    if (res != null) {
                        try {
                            return defineClass(name, res);    res (slot_2): null
                        } catch (IOException e) {
                            throw new ClassNotFoundException(name, e);    name: "Person"
                        }
                    }
                }
            }
        );
    }
}

```

接着会查看Bootstrap加载器的Path路径能否加载该类，Bootstrap是系统加载器是加载的类一般不为普通类，所以这里是无法加载Person类的

然后换成使用AppClassLoader加载器加载Person类，但是由于AppClassLoader是没有写findClass方法，所以是调用父类URLClassLoader.findClass方法


```

443  */
444  @private Class<?> defineClass(String name, Resource res) throws IOException {    name: "Person"    res
445      long t0 = System.nanoTime();    t0 (slot_3): 170181681840400
446      int i = name.lastIndexOf( ch: '.' );    i (slot_5): -1
447      URL url = res.getCodeSourceURL();    url (slot_6): "file:/D:/AtaoStudy/Java_Safety/java1.8_Demo/o
448      if (i != -1) {
449          String pkgname = name.substring(0, i);    i (slot_5): -1
450          // Check if package already loaded.
451          Manifest man = res.getManifest();
452          definePackageInternal(pkgname, man, url);
453      }
454      // Now read the class bytes and define the class
455      java.nio.ByteBuffer bb = res.getByteBuffer();    bb (slot_7): null
456      if (bb != null) {
457          // Use (direct) ByteBuffer:
458          CodeSigner[] signers = res.getCodeSigners();
459          CodeSource cs = new CodeSource(url, signers);
460          sun.misc.PerfCounter.getReadClassBytesTime().addElapsedTimeFrom(t0);
461          return defineClass(name, bb, cs);    bb (slot_7): null
462      } else {
463          byte[] b = res.getBytes();    b (slot_8): [-54, -2, -70, -66, 0, 0, 0, 52, 0, 49, +914 more]
464          // must read certificates AFTER reading bytes.
465          CodeSigner[] signers = res.getCodeSigners();    res: URLClassPath$FileLoader$1@559    signers
466          CodeSource cs = new CodeSource(url, signers);    url (slot_6): "file:/D:/AtaoStudy/Java_Safet
467          sun.misc.PerfCounter.getReadClassBytesTime().addElapsedTimeFrom(t0);    t0 (slot_3): 17018168
468          return defineClass(name, b, off: 0, b.length, cs);    name: "Person"    b (slot_8): [-54, -2,
469      }

```

进入到URLClassLoader.defineClass方法中，开始主要是一些安全检查，主要是最后一步的调用，继续跟进

```

138  @protected final Class<?> defineClass(String name,    name: "Person"
139      byte[] b, int off, int len,    b:
140      CodeSource cs)    cs: "(file:/D:/A
141  {
142      return defineClass(name, b, off, len, getProtectionDomain(cs));
143  }

```

来到SecureClassLoader.defineClass方法中

```

757  @protected final Class<?> defineClass(String name, byte[] b, int off, int len,
758      ProtectionDomain protectionDomain)    pro
759      throws ClassFormatError
760  {
761      protectionDomain = preDefineClass(name, protectionDomain);
762      String source = defineClassSourceLocation(protectionDomain);    source (sl
763      Class<?> c = defineClass1(name, b, off, len, protectionDomain, source);
764      postDefineClass(c, protectionDomain);
765      return c;
766  }
767

```

来到ClassLoader.defineClass方法中

```

856
857  private native Class<?> defineClass0(String name, byte[] b, int off, int len,
858      ProtectionDomain pd);
859
860  private native Class<?> defineClass1(String name, byte[] b, int off, int len,
861      ProtectionDomain pd, String source);
862
863  private native Class<?> defineClass2(String name, java.nio.ByteBuffer b,
864      int off, int len, ProtectionDomain pd,
865      String source);
866

```

最后是调用了这个ClassLoader.defineClass1方法，后面就是底层写的不会了。到此我们大概了解了一下整个类加载的过程

总结

1. 类的继承关系(左子类右父类): AppClassLoader → URLClassLoader → SecureClassLoader → ClassLoader
2. 类加载时的方法调用: loadClass → findClass → defineClass
3. findClass是判断该路径下能否加载该类，defineClass是通过字节码加载类

例 6

在上面我们已经知道了大致的类加载过程，接着实现类加载的代码。这里使用 URLClassLoader 类，通过它完成类加载

准备一个 Demo1.java

```
public class Demo1 {
    {
        System.out.println("Hello Word!!!");
    }
}
```

Java

将上面代码编译为 class 文件

```
import java.net.URL;
import java.net.URLClassLoader;

public class Demo {
    public static void main(String[] args) throws Exception {
        //URLClassLoader urlClassLoader = new URLClassLoader(new URL[]{new URL("file://class文件路径\\")});
        URLClassLoader urlClassLoader = new URLClassLoader(new URL[]{new URL("http://URL/")});
        Class c = urlClassLoader.loadClass("Demo1");
        c.newInstance();
    }
}
/*
Hello Word!!!
*/
```

Java

通过上述代码是可以直接加载本地和远程的 class 文件

结语

参考视频：https://www.bilibili.com/video/BV16h411z7o9?p=4&share_source=copy_web

动手跟着分析了类加载过程，以及了解了类加载过程中与安全相关的一些问题

如果那里有错误，希望能联系笔者进行修改