主讲老师：Fox

课前须知：

- 理解Tomcat是如何打破双亲委派机制的，如何利用多层次类加载器设计隔离web应用的
- 结合Tomcat热加载和热部署的实现，掌握利用后台线程的处理周期性任务的应用场景
- 有道云笔记地址：https://note.youdao.com/s/SBmJJ7Wj

# 1.Tomcat类加载机制详解

## 1.1 JVM类加载器

Java中有 3 个类加载器，另外你也可以自定义类加载器

- 引导（启动）类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的核心类库，比如rt.jar、charsets.jar等
- 扩展类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的ext扩展目录中的JAR类包
- 应用程序（系统）类加载器：负责加载ClassPath路径下的类包，主要就是加载你自己写的那些类
- 自定义加载器：负责加载用户自定义路径下的类包

```
public class ClassLoaderDemo {

    public static void main(String[] args) {
        // BootstrapClassLoader
        System.out.println(ReentrantLock.class.getClassLoader());
        // ExtClassLoader
        System.out.println(ZipInfo.class.getClassLoader());
        // AppClassLoader
        System.out.println(ClassLoaderDemo.class.getClassLoader());

        // AppClassLoader
        System.out.println(ClassLoader.getSystemClassLoader());
        // ExtClassLoader
        System.out.println(ClassLoader.getSystemClassLoader().getParent());
        // BootstrapClassLoader
        System.out.println(ClassLoader.getSystemClassLoader().getParent().getParent());

    }
}
```

## 1.2 双亲委派机制

JVM类加载器是有亲子层级结构的，如下图

这种类加载机制其实就是双亲委派机制，加载某个类时会先委托父加载器寻找目标类，找不到再委托上层父加载器加载，如果所有父加载器在自己的加载类路径下都找不到目标类，则在自己的类加载路径中查找并载入目标类。双亲委派机制说简单点就是，先找父亲加载，不行再由儿子自己加载。

### ClassLoader#loadClass源码分析

我们来看下应用程序类加载器AppClassLoader加载类的双亲委派机制源码，AppClassLoader的loadClass方法最终会调用其父类ClassLoader的loadClass方法，该方法的大体逻辑如下：

1. 首先，检查一下指定名称的类是否已经加载过，如果加载过了，就不需要再加载，直接返回。
2. 如果此类没有加载过，那么，再判断一下是否有父加载器；如果有父加载器，则由父加载器加载（即调用parent.loadClass(name, false);）.或者是调用bootstrap类加载器来加载。
3. 如果父加载器及bootstrap类加载器都没有找到指定的类，那么调用当前类加载器的findClass方法来完成类加载

```
public abstract class ClassLoader {



    // 每个类加载器都有个父加载器
    private final ClassLoader parent;

    public Class<?> loadClass(String name) {
        // 查找一下这个类是不是已经加载过了
        Class<?> c = findLoadedClass(name);
        // 如果没有加载过
        if( c == null ){
          // 先委托给父加载器去加载，注意这是个递归调用
          if (parent != null) {
              c = parent.loadClass(name);
          }else {
              // 如果父加载器为空，查找 Bootstrap 加载器是不是加载过了
              c = findBootstrapClassOrNull(name);
          }
        }

```

```
        // 如果父加载器没加载成功，调用自己的 findClass 去加载
        if (c == null) {
            c = findClass(name);
        }
        return c;
    }

    protected Class<?> findClass(String name){
       //1. 根据传入的类名 name，到在特定目录下去寻找类文件，把.class 文件读入内存
          ...
       //2. 调用 defineClass 将字节数组转成 Class 对象
       return defineClass(buf, off, len);
    }

    // 将字节码数组解析成一个 Class 对象，用 native 方法实现
    protected final Class<?> defineClass(byte[] b, int off, int len){
       ...
    }

}
```

思考：为什么要设计双亲委派机制？

- 沙箱安全机制：自己写的java.lang.String.class类不会被加载，这样便可以防止核心 API库被随意篡改
- 避免类的重复加载：当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次，保证被加载类的唯一性

## 1.3 Tomcat 如何打破双亲委派机制

Tomcat 的自定义类加载器 WebAppClassLoader 打破了双亲委派机制，它首先自己尝试去加载某个类，如果找不到再代理给父类加载器，其目的是优先加载 Web 应用自己定义的类。具体实现就是重写 ClassLoader 的两个方法：findClass 和 loadClass。

### findClass方法

我们先来看看 findClass 方法的实现

```
public Class<?> findClass(String name) throws ClassNotFoundException {
    ...
    Class<?> clazz = null;
```

```
    try {
            //1. 先在 Web 应用目录下查找类
            clazz = findClassInternal(name);
    }  catch (RuntimeException e) {
           throw e;
    }

    if (clazz == null) {
    try {
            //2. 如果在本地目录没有找到，交给父加载器去查找
            clazz = super.findClass(name);
    }  catch (RuntimeException e) {
           throw e;
    }

    //3. 如果父类也没找到，抛出 ClassNotFoundException
    if (clazz == null) {
        throw new ClassNotFoundException(name);
     }
    return clazz;
}
```

在 findClass 方法里，主要有三个步骤：

1）先在 Web 应用本地目录下查找要加载的类。

2）如果没有找到，交给父加载器去查找，它的父加载器就是上面提到的系统类加载器 AppClassLoader。

3）如何父加载器也没找到这个类，抛出 ClassNotFound 异常。

## loadClass 方法

接着我们再来看 Tomcat 类加载器的 loadClass 方法的实现

```
public Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException {
    synchronized (getClassLoadingLock(name)) {
        Class<?> clazz = null;
        //1. 先在本地 cache 查找该类是否已经加载过
        clazz = findLoadedClass0(name);
        if (clazz != null) {
            if (resolve)
```

```
8                 resolveClass(clazz);
9             return clazz;
10        }
11
12        //2. 从系统类加载器的 cache 中查找是否加载过
13        clazz = findLoadedClass(name);
14        if (clazz != null) {
15            if (resolve)
16                resolveClass(clazz);
17            return clazz;
18        }
19
20        // 3. 尝试用 ExtClassLoader 类加载器类加载，为什么？
21        ClassLoader javaseLoader = getJavaseClassLoader();
22        try {
23            clazz = javaseLoader.loadClass(name);
24            if (clazz != null) {
25                if (resolve)
26                    resolveClass(clazz);
27                return clazz;
28            }
29        } catch (ClassNotFoundException e) {
30            // Ignore
31        }
32
33        // 4. 尝试在本地目录搜索 class 并加载
34        try {
35            clazz = findClass(name);
36            if (clazz != null) {
37                if (resolve)
38                    resolveClass(clazz);
39                return clazz;
40            }
41        } catch (ClassNotFoundException e) {
42            // Ignore
43        }
44
45        // 5. 尝试用系统类加载器 (也就是 AppClassLoader) 来加载
46            try {
47                clazz = Class.forName(name, false, parent);
```

```
                if (clazz != null) {
                    if (resolve)
                        resolveClass(clazz);
                    return clazz;
                }
            } catch (ClassNotFoundException e) {
                // Ignore
            }
        }

    //6. 上述过程都加载失败，抛出异常
    throw new ClassNotFoundException(name);
}
```

loadClass 方法稍微复杂一点，主要有六个步骤：

1）先在本地 Cache 查找该类是否已经加载过，也就是说 Tomcat 的类加载器是否已经加载过这个类。

2）如果 Tomcat 类加载器没有加载过这个类，再看看系统类加载器是否加载过。

3）如果都没有，就让ExtClassLoader去加载，这一步比较关键，目的防止 Web 应用自己的类覆盖 JRE 的核心类。因为 Tomcat 需要打破双亲委派机制，假如 Web 应用里自定义了一个叫 Object 的类，如果先加载这个 Object 类，就会覆盖 JRE 里面的那个 Object 类，这就是为什么 Tomcat 的类加载器会优先尝试用 ExtClassLoader 去加载，因为 ExtClassLoader 会委托给 BootstrapClassLoader 去加载，BootstrapClassLoader 发现自己已经加载了 Object 类，直接返回给 Tomcat 的类加载器，这样 Tomcat 的类加载器就不会去加载 Web 应用下的 Object 类了，也就避免了覆盖 JRE 核心类的问题。

4）如果 ExtClassLoader 加载器加载失败，也就是说 JRE 核心类中没有这类，那么就在本地 Web 应用目录下查找并加载。

5）如果本地目录下没有这个类，说明不是 Web 应用自己定义的类，那么由系统类加载器去加载。这里请你注意，Web 应用是通过Class.forName调用交给系统类加载器的，因为Class.forName的默认加载器就是系统类加载器。

6）如果上述加载过程全部失败，抛出 ClassNotFound 异常。

从上面的过程我们可以看到，Tomcat 的类加载器打破了双亲委派机制，没有一上来就直接委托给父加载器，而是先在本地目录下加载，为了避免本地目录下的类覆盖 JRE 的核心类，先尝试用 JVM 扩展类加载器 ExtClassLoader 去加载。那为什么不先用系统类加载器 AppClassLoader 去加载？很显然，如果是这样的话，那就变成双亲委派机制了，这就是 Tomcat 类加载器的巧妙之处。

## 1.4 Tomcat如何隔离Web应用

Tomcat 作为 Servlet 容器，它负责加载我们的 Servlet 类，此外它还负责加载 Servlet 所依赖的 JAR 包。并且 Tomcat 本身也是也是一个 Java 程序，因此它需要加载自己的类和依赖的 JAR 包。首先让我们思考这一下这几个问题：

1）假如我们在 Tomcat 中运行了两个 Web 应用程序，两个 Web 应用中有同名的 Servlet，但是功能不同，Tomcat 需要同时加载和管理这两个同名的 Servlet 类，保证它们不会冲突，因此 Web 应用之间的类需要隔离。

2）假如两个 Web 应用都依赖同一个第三方的 JAR 包，比如 Spring，那 Spring 的 JAR 包被加载到内存后，Tomcat 要保证这两个 Web 应用能够共享，也就是说 Spring 的 JAR 包只被加载一次，否则随着依赖的第三方 JAR 包增多，JVM 的内存会膨胀。

3）跟 JVM 一样，我们需要隔离 Tomcat 本身的类和 Web 应用的类。

思考：Tomcat 是如何解决这些问题的?

**Tomcat类加载器的层次结构**

为了解决这些问题，Tomcat 设计了类加载器的层次结构，它们的关系如下图所示：

- commonLoader：Tomcat最基本的类加载器，加载路径中的class可以被Tomcat容器本身以及各个Webapp访问；
- catalinaLoader：Tomcat容器私有的类加载器，加载路径中的class对于Webapp不可见；
- sharedLoader：各个Webapp共享的类加载器，加载路径中的class对于所有Webapp可见，但是对于Tomcat容器不可见；
- WebappClassLoader：各个Webapp私有的类加载器，加载路径中的class只对当前Webapp可见，比如加载war包里相关的类，每个war包应用都有自己的WebappClassLoader，实现相互隔离，比如不同war包应用引入了不同的spring版本，这样实现就能加载各自的spring版本；

### WebAppClassLoader

我们先来看**第 1 个问题**，假如我们使用 JVM 默认 AppClassLoader 来加载 Web 应用，AppClassLoader 只能加载一个 Servlet 类，在加载第二个同名 Servlet 类时，AppClassLoader 会返回第一个 Servlet 类的 Class 实例，这是因为在 AppClassLoader 看来，同名的 Servlet 类只被加载一次。

因此 Tomcat 的解决方案是自定义一个类加载器 WebAppClassLoader， 并且给每个 Web 应用创建一个类加载器实例。我们知道，Context 容器组件对应一个 Web 应用，因此，每个 Context 容器负责创建和维护一个 WebAppClassLoader 加载器实例。这背后的原理是，**不同的加载器实例加载的类被认为是不同的类**，即使它们的类名相同。

### SharedClassLoader

我们再来看**第 2 个问题**，本质需求是两个 Web 应用之间怎么共享库类，并且不能重复加载相同的类。Tomcat 的设计者又加了一个类加载器 SharedClassLoader，作为 WebAppClassLoader 的父加载器，专门来加载 Web 应用之间共享的类。如果 WebAppClassLoader 自己没有加载到某个类，就会委托父加载器 SharedClassLoader 去加载这个类，SharedClassLoader 会在指定目录下加载共享类，之后返回给 WebAppClassLoader，这样共享的问题就解决了。

### CatalinaClassloader

我们来看**第 3 个问题**，如何隔离 Tomcat 本身的类和 Web 应用的类？我们知道，要共享可以通过父子关系，要隔离那就需要兄弟关系了。兄弟关系就是指两个类加载器是平行的，它们可能拥有同一个父加载器，但是两个兄弟类加载器加载的类是隔离的。基于此 Tomcat 又设计一个类加载器 CatalinaClassloader，专门来加载 Tomcat 自身的类。这样设计有个问题，那 Tomcat 和各 Web 应用之间需要共享一些类时该怎么办呢？

### CommonClassLoader

老办法，还是再增加一个 CommonClassLoader，作为 CatalinaClassloader 和 SharedClassLoader 的父加载器。CommonClassLoader 能加载的类都可以被 CatalinaClassLoader 和 SharedClassLoader 使用，而 CatalinaClassLoader 和 SharedClassLoader 能加载的类则与对方相互隔离。WebAppClassLoader 可以使用 SharedClassLoader 加载到的类，但各个 WebAppClassLoader 实例之间相互隔离。

## Spring 的加载问题

### 全盘负责委托机制

“全盘负责”是指当一个ClassLoder装载一个类时，除非显示的使用另外一个ClassLoder，该类所依赖及引用的类也由这个ClassLoder载入。

比如 Spring 作为一个 Bean 工厂，它需要创建业务类的实例，并且在创建业务类实例之前需要加载这些类。Spring 是通过调用Class.forName来加载业务类的，我们来看一下 forName 的源码：

```
1 public static Class<?> forName(String className) {
2     Class<?> caller = Reflection.getCallerClass();
3     return forName0(className, true, ClassLoader.getClassLoader(caller), caller);
4 }
```

可以看到在 forName 的函数里，会用调用者也就是 Spring 的加载器去加载业务类。

我们在前面提到，Web 应用之间共享的 JAR 包可以交给 SharedClassLoader 来加载，从而避免重复加载。Spring 作为共享的第三方 JAR 包，它本身是由 SharedClassLoader 来加载的，Spring 又要去加载业务类，按照前面那条规则，加载 Spring 的类加载器也会用来加载业务类，但是业务类在 Web 应用目录下，不在 SharedClassLoader 的加载路径下，这该怎么办呢？

**线程上下文加载器**

于是线程上下文加载器登场了，它其实是一种类加载器传递机制。为什么叫作“线程上下文加载器”呢，因为这个类加载器保存在线程私有数据里，只要是同一个线程，一旦设置了线程上下文加载器，在线程后续执行过程中就能把这个类加载器取出来用。因此 Tomcat 为每个 Web 应用创建一个 WebAppClassLoarder 类加载器，并在启动 Web 应用的线程里设置线程上下文加载器，这样 Spring 在启动时就将线程上下文加载器取出来，用来加载 Bean。Spring 取线程上下文加载的代码如下：

```
cl = Thread.currentThread().getContextClassLoader();
```

线程上下文加载器不仅仅可以用在 Tomcat 和 Spring 类加载的场景里，核心框架类需要加载具体实现类时都可以用到它，比如我们熟悉的 JDBC 就是通过上下文类加载器来加载不同的数据库驱动的

## 2. Tomcat热加载和热部署

在项目开发过程中，经常要改动Java/JSP 文件，但是又不想重新启动Tomcat，有两种方式:热加载和热部署。热部署表示重新部署应用，它的执行主体是Host。 热加载表示重新加载class，它的执行主体是Context。

- 热加载：在server.xml -> context 标签中 设置 reloadable="true"

```
<Context docBase="D:\mvc" path="/mvc"  reloadable="true" />
```

- 热部署：在server.xml -> Host标签中 设置  autoDeploy="true"

```
<Host name="localhost"  appBase="webapps"
      unpackWARs="true" autoDeploy="true">
```

它们的区别是：

- 热加载的实现方式是 Web 容器启动一个后台线程，定期检测类文件的变化，如果有变化，就重新加载类，在这个过程中不会清空 Session ，一般用在开发环境。
- 热部署原理类似，也是由后台线程定时检测 Web 应用的变化，但它会重新加载整个 Web 应用。这种方式会清空 Session，比热加载更加干净、彻底，一般用在生产环境。

思考：Tomcat 是如何用后台线程来实现热加载和热部署的?

### 2.1 Tomcat开启后台线程执行周期性任务

Tomcat 通过开启后台线程ContainerBase.ContainerBackgroundProcessor，使得各个层次的容器组件都有机会完成一些周期性任务。我们在实际工作中，往往也需要执行一些周期性的任务，比如监控程序周期性拉取系统的健康状态，就可以借鉴这种设计。

Tomcat9 是通过ScheduledThreadPoolExecutor来开启后台线程的，它除了具有线程池的功能，还能够执行周期性的任务。

此后台线程会调用当前容器的 backgroundProcess 方法，以及递归调用子孙的 backgroundProcess 方法，backgroundProcess 方法会触发容器的周期性任务。

有了 ContainerBase 中的后台线程和 backgroundProcess 方法，各种子容器和通用组件不需要各自弄一个后台线程来处理周期性任务，这样的设计显得优雅和整洁。

## 2.2 Tomcat热加载实现原理

有了 ContainerBase 的周期性任务处理“框架”，作为具体容器子类，只需要实现自己的周期性任务就行。而 Tomcat 的热加载，就是在 Context 容器中实现的。Context 容器的 backgroundProcess 方法是这样实现的：

```
//  StandardContext#backgroundProcess

//WebappLoader 周期性的检查 WEB-INF/classes 和 WEB-INF/lib 目录下的类文件
// 热加载
Loader loader = getLoader();
if (loader != null) {
    loader.backgroundProcess();
}
```

WebappLoader 实现热加载的逻辑：它主要是调用了 Context 容器的 reload 方法，先stop Context容器，再start Context容器。具体的实现：

1）停止和销毁 Context 容器及其所有子容器，子容器其实就是 Wrapper，也就是说 Wrapper 里面 Servlet 实例也被销毁了。

2）停止和销毁 Context 容器关联的 Listener 和 Filter。

3）停止和销毁 Context 下的 Pipeline 和各种 Valve。

4）停止和销毁 Context 的类加载器，以及类加载器加载的类文件资源。

5）启动 Context 容器，在这个过程中会重新创建前面四步被销毁的资源。

在这个过程中，类加载器发挥着关键作用。一个 Context 容器对应一个类加载器，类加载器在销毁的过程中会把它加载的所有类也全部销毁。Context 容器在启动过程中，会创建一个新的类加载器来加载新的类文件。

## 2.3 Tomcat热部署实现原理

热部署跟热加载的本质区别是，热部署会重新部署 Web 应用，原来的 Context 对象会整个被销毁掉，因此这个 Context 所关联的一切资源都会被销毁，包括 Session。

Host 容器并没有在 backgroundProcess 方法中实现周期性检测的任务，而是通过监听器 HostConfig 来实现的

```
// HostConfig#lifecycleEvent
// 周期性任务
if (event.getType().equals(Lifecycle.PERIODIC_EVENT)) {
    check();
}
protected void check() {
    if (host.getAutoDeploy()) {
        // Check for resources modification to trigger redeployment
        DeployedApplication[] apps = deployed.values().toArray(new DeployedApplication[0]);
        for (DeployedApplication app : apps) {
            if (tryAddServiced(app.name)) {
                try {
                    // 检查 Web 应用目录是否有变化
                    checkResources(app, false);
                } finally {
                    removeServiced(app.name);
                }
            }
        }
        // Check for old versions of applications that can now be undeployed
        if (host.getUndeployOldVersions()) {
            checkUndeploy();
        }

        // Hotdeploy applications
```

```
26          //热部署
27          deployApps();
28      }
```

HostConfig 会检查 webapps 目录下的所有 Web 应用：

- 如果原来 Web 应用目录被删掉了，就把相应 Context 容器整个销毁掉。
- 是否有新的 Web 应用目录放进来了，或者有新的 WAR 包放进来了，就部署相应的 Web 应用。

因此 HostConfig 做的事情都是比较“宏观”的，它不会去检查具体类文件或者资源文件是否有变化，而是检查 Web 应用目录级别的变化。