主讲老师：Fox

课前须知：

1. 关于类加载机制JVM专题（jvm第一节）会用更深入的讲解，本节课重心是理解双亲委派机制，以及Tomcat是如何打破双亲委派机制的
2. 结合Tomcat热加载和热部署的实现，理解java后台线程的应用
3. 本节课后半段会补充上节课的知识（Tomcat线程模型），先学上节课，再来学这部分
4. 新加入的同学，不建议直接跟本节课，从Tomcat第一节课开始学习

5. Tomcat结束之后，是JVM专题

JVM：　录播（诸葛老师的JVM调优课程 ）+ 升级后的直播课（ZGC+Hotspot源码 ）

```
文档：3. Tomcat类加载机制&热加载和热部署实...
链接：http://note.youdao.com/noteshare?id=b5cd75383892b3f2db1837255e4bf2aa&sub=47D22C45128B481C8C1FCACB42D1F633
```


Tomcat类加载机制

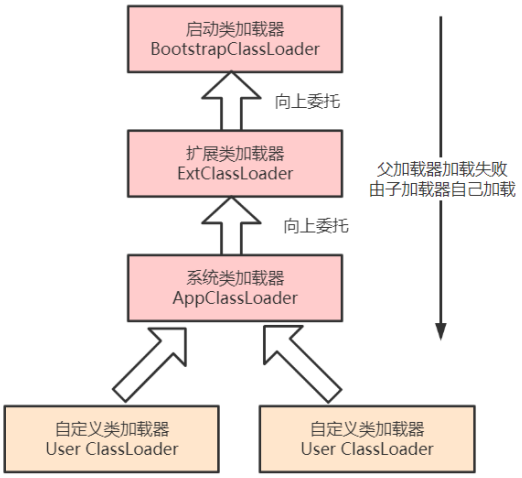
JVM类加载器

Java中有 3 个类加载器，另外你也可以自定义类加载器

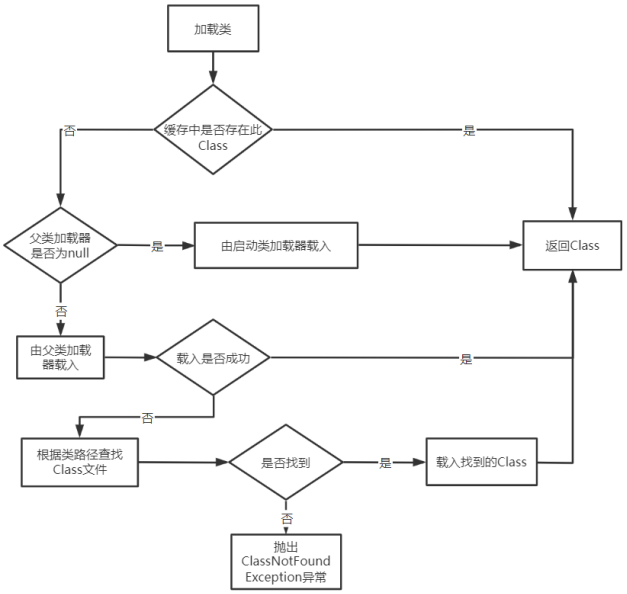
- BootstrapClassLoader 是启动类加载器，由 C 语言实现，用来加载 JVM 启动时所需要的核心类，比如rt.jar。
- ExtClassLoader 是扩展类加载器，用来加载\jre\lib\ext目录下 JAR 包。扩展加载器的 #getParent() 方法返回 null ，但实际上扩展类加载器的父类加载器就是启动类加载器。
- AppClassLoader 是系统类加载器，用来加载 classpath 下的类，应用程序默认用它来加载类。程序可以通过 #getSystemClassLoader() 来获取系统类加载器。
- 自定义类加载器，用来加载自定义路径下的类。

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。这就是双亲委派机制。



ClassLoader#loadClass加载过程



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

Tomcat 的类加载机制

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。

思考： Tomcat是如何隔离Web应用的?

Tomcat 自定义了一个类加载器 WebAppClassLoader， 并且给每个 Web 应用创建一个类加载器实例，每个 Context 容器负责创建和维护一个 WebAppClassLoader 加载器实例。其实现的原理就是不同的类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（不同类加载器实例加载的类是互相隔离的）。

Tomcat 的自定义类加载器 WebAppClassLoader 打破了双亲委托机制，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。

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 需要打破双亲委托机制，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。

Tomcat类加载器的层次结构

Tomcat 拥有不同的自定义类加载器，以实现对各种资源库的控制。 Tomcat 主要用类加载器解决以下 4 个问题：

- 同一个 Web 服务器里，各个 Web 项目之间各自使用的 Java 类库要互相隔离。
- 同一个 Web 服务器里，各个 Web 项目之间可以提供共享的 Java 类库 。
- 为了使服务器不受 Web 项目的影响，应该使服务器的类库与应用程序的类库互相独立。
- 对于支持 JSP 的 Web 服务器，应该支持热插拔（ HotSwap ）功能 。

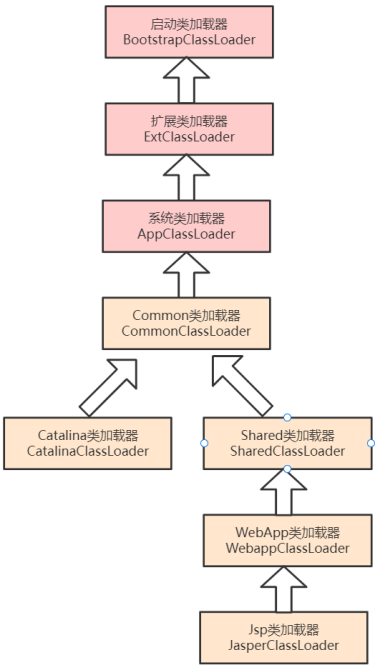
Tomcat提供了四组目录供用户存放第三方类库：

- 放置在/common目录中：类库可被Tomcat和所有的 Web应用程序共同使用。
- 放置在/server目录中：类库可被Tomcat使用，对所有的Web应用程序都不可见。

- 放置在/shared目录中：类库可被所有的Web应用程序共同使用，但对 Tomcat自己不可见。
- 放置在/WebApp/WEB-INF目录中：类库仅仅可以被此Web应用程序使用，对Tomcat和其他Web应用程序都不可见。

Tomcat自定义了多个类加载器，CommonClassLoader、CatalinaClassLoader、SharedClassLoader和WebappClassLoader则是Tomcat自己定义的类加载器，它们分别加载/common/、/server/、/shared/和/WebApp/WEB-INF/中的Java类库。其中WebApp类加载器和Jsp类加载器通常会存在多个实例，每一个Web应用程序对应一个WebApp类加载器，每一个JSP文件对应一个Jsp类加载器。

- **CommonClassLoader** : Tomcat 通用类加载器, 加载的资源可被 Tomcat 和所有的 Web 应用程序共同获取
- **CatalinaClassLoader** : Tomcat 类加载器, 加载的资源只能被 Tomcat 获取(但所有 WebappClassLoader 不能获取到 catalinaLoader 加载的类)
- **SharedClassLoader** : Tomcat 各个Context的父加载器, 这个类是所有 WebappClassLoader 的父类, sharedLoader 所加载的类将被所有的 WebappClassLoader 共享获取
- **WebappClassLoader** ：每个Context 对应一个 WebappClassloader, 主要用于加载 WEB-INF/lib 与 WEB-INF/classes 下面的资源



线程上下文加载器

在 JVM 的实现中有一条隐含的规则，默认情况下，如果一个类由类加载器 A 加载，那么这个类的依赖类也是由相同的类加载器加载。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。

思考：如果spring作为共享第三方jar包，交给SharedClassLoader加载，但是业务类在web目录下，不在SharedClassLoader的加载路径下，那spring如何加载web应用目录下的业务bean呢?

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。

线程上下文加载器是一种类加载器传递机制，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。

```
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

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?

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。

```
        backgroundProcessorFuture = Container.getService( container: this).getServer()
.getUtilityExecutor()
                .scheduleWithFixedDelay(new ContainerBackgroundProcessor(),
                        backgroundProcessorDelay, backgroundProcessorDelay,
                        TimeUnit.SECONDS);
```

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。

```
//调用当前容器的 backgroundProcess 方法
container.backgroundProcess();
Container[] children = container.findChildren();
//遍历所有的子容器，递归调用 processChildren
for (Container child : children) {
    if (child.getBackgroundProcessorDelay() <= 0) {
        processChildren(child);
    }
}
```

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。

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。

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 DeployedAppli
cation[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);
  }
  }
  }
```

```
20  // Check for old versions of applications that can now be undeployed
21  if (host.getUndeployOldVersions()) {
22  checkUndeploy();
23  }
24
25  // 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。