# 继承结构

|  |
| --- |
|  |

# 重要属性-parent

|  |
| --- |
| public abstract class ClassLoader {  //…  private final ClassLoader parent; //重要  //…  } |

# 类加载器的分类

在双亲委托机制中：各个加载器按照父子关系形成了树形结构，除了根加载器以外，其他加载器有且只有一个父加载器。

自定义的类加载器时，如果没有指定它的父加载器，那么系统类加载器就将成为该类加载器的父加载器。

## 定义类加载器

如果某个类加载器能够加载一个类，那么该类加载器就称作：定义类加载器；

## 初始类加载器

定义类加载器及其所有子加载器都称作：初始类加载器。

# 类加载器的命名空间

每个类加载器都有自己的命名空间，命名空间由该类加载器及所有父加载器所加载的类组成。在同一个命名空间中，不会出现类的完整名字（包括类的包名）相同的两个类；在不同的命名空间中，有可能会出现类的完整名字（包括类的包名）相同的两个类。但不同命名空间下的类相互不可见。

# jvm默认使用的三个ClassLoader

java应用环境中不同的class分别由不同的ClassLoader负责加载。

一个jvm中默认的classloader有Bootstrap ClassLoader、ExtClassLoader、AppClassLoader，分别各司其职，通常情况下有这三个类加载器都已经可以了；

其中：

* Bootstrap ClassLoader是JVM级别的，由C++撰写；
* Extension ClassLoader、App ClassLoader都是java类，都继承自URLClassLoader超类。

初始化顺序：

* JVM 初始化BootstrapClassLoader
* BootstrapClassLoader初始化sun.misc.Launcher
* sun.misc.Launcher初始化ExtClassLoader、AppClassLoader

## BootstrapClassLoader

**BootstrapClassLoader** 负责加载java基础类，主要是 %JRE\_HOME/lib/ 目录下的rt.jar、resources.jar、charsets.jar和class等

## ExtClassLoader

**ExtClassLoader（ExtensionClassLoader）**负责加载java扩展类，主要是 %JRE\_HOME/lib/ext 目录下的jar和class

## AppClassLoader

**AppClassLoader（**SystemClassLoader**）** 负责加载当前java应用的classpath中的所有类。

# Jvm执行类加载的三个契机

* 包括隐式的new，

过程:

加载（使用classloader加载，我猜测应该是Class.forname(带验证)）、链接（验证（类是否标准？）、准备（static域指定jvm默认值）、解析（常量池符号链接修改为直接引用））、初始化（执行static块和static域）、实例化(堆中分配内存、实例变量值制定、调用构造函数、将对象在堆区的收地址返回给应用)；

所以此时 类的加载仅仅是new整个过程过的其中一个步骤。

注：上面所言过程是类还未加载的情况下的过程，实际情况可能是类已经处在初始化阶段，那么此时new的话，仅仅走实例化过程的。

* 显式的Class.forName()

过程

加载（使用classloader加载，我猜测应该是Class.forname(带验证)）、链接（验证（类是否标准？）、准备（static域指定jvm默认值）、解析（常量池符号链接修改为直接引用））、初始化（执行static块和static域）;

所以此时 类的加载仅仅是Class.forName()整个过程过的其中一个步骤。

* 显式的ClassLoader.loadCLass().

默认过程（使用jvm原生加载器，不使用自定义加载器）：

加载（使用classloader加载，我猜测应该是Class.forname(带验证)）、链接（验证（类是否标准？）、准备（static域指定jvm默认值））。

所以此时 类的加载仅仅是ClassLoader.loadCLass整个过程过的其中一个步骤。

# Jvm默认加载机制

Jvm默认加载机制遵循两个标准：

### 双亲委托机制

双亲委托机制可以描述为：

* 当前ClassLoader首先从自己已经加载的类中查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。

每个类加载器都有自己的加载缓存，当一个类被加载了以后就会放入缓存，等下次加载的时候就可以直接返回了。

* 当前classLoader的缓存中没有找到被加载的类的时候，委托父类加载器去加载，父类加载器采用同样的策略，首先查看自己的缓存，然后委托父类的父类去加载，一直到bootstrp ClassLoader.
* 当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中，以便下次有加载请求的时候直接返回。

双亲委托机制深刻理解：

上层classLoder加载的class可以使用更上级classLoader加载的class，但无法使用其下级classloader加载的类。

双亲委托机制好处：

* 防止内存中出现多份同样的字节码，因为这样可以避免重复加载，当父亲已经加载了该类的时候，就没有必要子ClassLoader再加载一次。
* 考虑到安全因素，试想一下，如果不使用这种委托模式，那我们就可以随时使用自定义的String来动态替代java核心api中定义类型，这样会存在非常大的安全隐患， 而双亲委托的方式，就可以避免这种情况。

双亲委托机制不足：

上层classLoder加载的class可以使用更上级classLoader加载的class，但无法使用其下级classloader加载的class。但是如果上级classLoder加载的class要使用下级classLoder加载的class，怎么办？双亲加载机制就无能为力（这种情况虽然很少见，但并不是不存在）。

请注意：

“双亲委托机制”是Java推荐/默认的机制，但并不是强制的机制；我们可以继承java.lang.ClassLoader类，实现自己的类加载器：

* 如果想保持双亲委派模型，就应该重写findClass(name)方法()；
* 如果想破坏双亲委派模型，可以重写loadClass(name)方法。

### 全盘负责机制

同一个加载器->类A引用到类B，则由类A的加载器去加载类B，保证引用到的类由同一个加载器加载。

# 类加载器的使用

## Class#forName

使用调用者的ClassLoader加载被调用者的class。同时实例化被调用者class。

|  |
| --- |
| public static Class<?> forName(String className)  throws ClassNotFoundException {  Class<?> caller = Reflection.*getCallerClass*();  return *forName0*(className, true, ClassLoader.*getClassLoader*(caller), caller); } |

|  |
| --- |
| Reflection#getCallerClass  //获得调用者的Class  @CallerSensitive public static native Class getCallerClass();  ClassLoader#getClassLoader  //获得调用者的ClassLoader  static ClassLoader getClassLoader(Class<?> caller) {  // This can be null if the VM is requesting it  if (caller == null) {  return null;  }  // Circumvent security check since this is package-private  return caller.getClassLoader0(); }  //该方法是本地方法  //第二个参数(bool)表示装载类的时候是否初始化该类，即调用类的静态块的语句及初始化静态成员变量  private static native Class<?> forName0(String name, boolean initialize,  ClassLoader loader,  Class<?> caller)  throws ClassNotFoundException; |

## ClassLoader#loadClass

|  |
| --- |
| ClassLoader #loadClass:  public Class<?> loadClass(String name) throws ClassNotFoundException {  return loadClass(name, false); }  ClassLoader #loadClass:  protected Class<?> loadClass(String name, boolean resolve)  throws ClassNotFoundException {  synchronized (getClassLoadingLock(name)) {  // First, check if the class has already been loaded  Class c = findLoadedClass(name);  if (c == null) {  long t0 = System.*nanoTime*();  try {  if (parent != null) {  c = parent.loadClass(name, false);  } else {  c = findBootstrapClassOrNull(name);  }  } catch (ClassNotFoundException e) {  // ClassNotFoundException thrown if class not found  // from the non-null parent class loader  }   if (c == null) {  // If still not found, then invoke findClass in order  // to find the class.  long t1 = System.*nanoTime*();  c = findClass(name);   // this is the defining class loader; record the stats  sun.misc.PerfCounter.*getParentDelegationTime*().addTime(t1 - t0);  sun.misc.PerfCounter.*getFindClassTime*().addElapsedTimeFrom(t1);  sun.misc.PerfCounter.*getFindClasses*().increment();  }  }  if (resolve) {  resolveClass(c);  }  return c;  } }  ClassLoader #findClass:  protected Class<?> findClass(String name) throws ClassNotFoundException {  throw new ClassNotFoundException(name); }  我们可以看出此方法默认的实现是直接抛出异常，其实这个方法就是留给我们应用程序来override的。那么具体的实现就看你的实现逻辑了，你可以从磁盘读取，也可以从网络上获取class文件的字节流，获取class二进制了以后就可以交给defineClass来实现进一步的加载。defineClass我们再下面再来描述。ok，通过上面的分析，我们可以得出如下结论：  我们在写自己的ClassLoader的时候，如果想遵循双亲委托机制，则仅需要override findClass.  例如  protected Class<?> findClass(final String name)  throws ClassNotFoundException {  try {  return AccessController.*doPrivileged*(  new PrivilegedExceptionAction<Class>() {  public Class run() throws ClassNotFoundException {  String path = name.replace('.', '/').concat(".class");  Resource res = ucp.getResource(path, false);  if (res != null) {  try {  return defineClass(name, res);  } catch (IOException e) {  throw new ClassNotFoundException(name, e);  }  } else {  throw new ClassNotFoundException(name);  }  }  }, acc);  } catch (java.security.PrivilegedActionException pae) {  throw (ClassNotFoundException) pae.getException();  } }  ClassLoader #defineClass:  protected final Class<?> defineClass(String name, byte[] b, int off, int len,  ProtectionDomain protectionDomain)  throws ClassFormatError {  protectionDomain = preDefineClass(name, protectionDomain);   Class c = null;  String source = defineClassSourceLocation(protectionDomain);   try {  c = defineClass1(name, b, off, len, protectionDomain, source);  } catch (ClassFormatError cfe) {  c = defineTransformedClass(name, b, off, len, protectionDomain, cfe,  source);  }   postDefineClass(c, protectionDomain);  return c; }  此方法被定义为了final，这也就意味着此方法不能被Override，其实这也是jvm留给我们的唯一的入口，通过这个唯 一的入口，jvm保证了类文件必须符合Java虚拟机规范规定的类的定义。此方法最后会调用native的方法来实现真正的类的加载工作。 |

## Class#forName和ClassLoader#loadClass的区别

装载器不同：

前者使用调用者装载器来装载；后者需指定；

是否初始化

前者完成了类生命周期的初始化阶段；后者默认情况（ClassLoader类默认resove=false）仅仅走到链接阶段的准备阶段(还差解析阶段未走完)—当然如果你自定义加载器就不说了；

### Jdbc的注册是否可以用classloader#loadClass?

我们常使用一下语句注册jdbc驱动（以msql为例）：

|  |
| --- |
| Class.forName("com.mysql.jdbc.Driver"); |

打开com.mysql.jdbc.Driver的源代码：

|  |
| --- |
| static {      try {          java.sql.DriverManager.registerDriver(new Driver());      } catch (SQLException E) {          throw new RuntimeException("Can't register driver!");      }  } |

该static块的执行是在类生命周期的初始化阶段完成的，而classLoader默认情况下仅走到链接阶段的验证步骤。故不可以使用classloader#loadClass。

注：类的生命周期：加载、链接（验证（符合类标准?）、准备(static 值指定为jvm默认值)、解析（常量池符号链接修改为直接引用））、初始化（静态块执行）、实例化、使用、卸载。

# 打破jvm默认的加载机制

## 为什么要打破默认的加载机制？

这其实是因为加载Class的默认机制在某些情况下不能满足现实的需要。

在大部分情况下，jvm的这个默认类加载体系已经能满足大部分情况的使用了。但是总有例外：比如jdk中的jdbc API 和具体数据库厂商的实现类SPI的类加载问题。在jdbc API的类是由BootStrap加载的，那么如果在jdbc API需要用到spi的实现类时，根据默机制2，则实现类也会由BootStrap加载，但是spi（server provide interface）实现类却没法由BootStrap加载，只能由 Ext或者App加载，如何解决这个问题？

## 如何打破?

### **Thead#contextClassLoade**r

我们常使用一下语句注册jdbc驱动（以msql为例）：

|  |
| --- |
| Class.forName("com.mysql.jdbc.Driver"); |

打开com.mysql.jdbc.Driver的源代码：

|  |
| --- |
| static {      try {          java.sql.DriverManager.registerDriver(new Driver());      } catch (SQLException E) {          throw new RuntimeException("Can't register driver!");      }  } |

**分别由谁来加载？**

* java.sql.DriverManager.registerDriver是由BootstrapClassLoader来加载（因为该类属于jdk jar中的类）
* com.mysql.jdbc.Driver由谁来加载，暂时打个问号。

**问题描述**

1. 从com.mysql.jdbc.Driver角度分析：

在类com.mysql.jdbc.Driver中存在java.sql.DriverManager类;根据双亲委托机制(可以使用上级加载器加载的类)，com.mysql.jdbc.Driver是可以使用java.sql.DriverManager.registerDriver的，因此没有问题的。

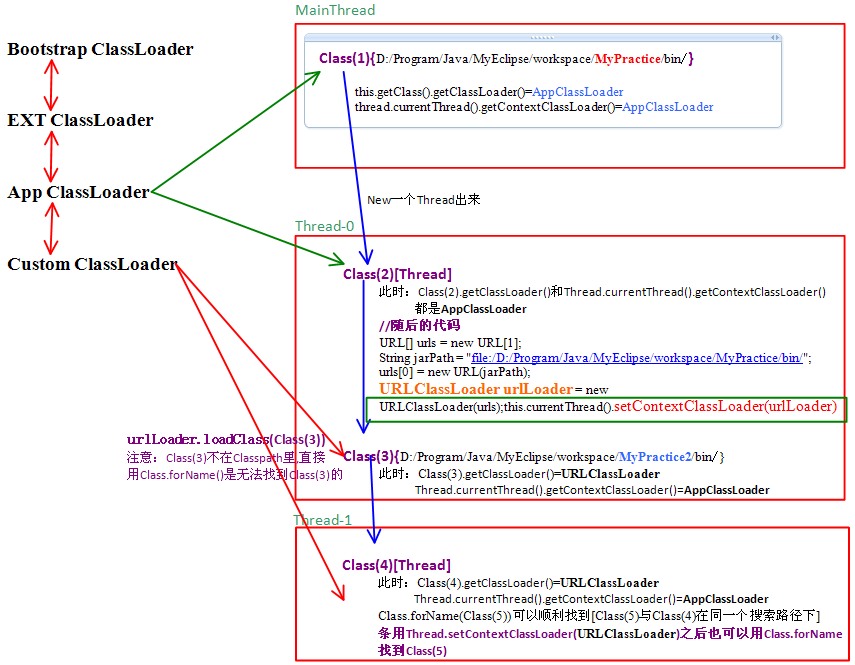
1. 从java.sql.DriverManager角度分析

当执行方法registerDriver(new Driver())时，是在java.sql.DriverManager中执行的：

* 根据双亲负责机制com.mysql.jdbc.Driver要么由更上级的加载器来加载（这里不可能，因为当前加载器已是最顶层），要么由当前加载器来加载（这里是BootstrapClassLoader），但又由于com.mysql.jdbc.Driver是不可能由当前加载器来加载的（因为该类不属于jdk jar中的类）。
* 根据全盘负责机制com.mysql.jdbc.Driver应由当前加载器来加载（这里是BootstrapClassLoader）来加载，，但又由于com.mysql.jdbc.Driver是不可能由当前加载器来加载的（因为该类不属于jdk jar中的类）。

**解决问题：**

上面的问题其实可以概括为上级加载器加载的类需要到了使用下级加载器加载的类。



# 查看某个类是由哪个加载器加载

|  |
| --- |
| getClass().getClassLoader(); |

# 附录

源码解读

Launcher 由bootstrap classloader来调用

Launcher的部分代码：

|  |
| --- |
| 1 public class Launcher  {  2     public Launcher() {  3         ExtClassLoader extclassloader;  4         try {  5             //初始化extension classloader  6             extclassloader = ExtClassLoader.getExtClassLoader();  7         } catch(IOException ioexception) {  8             throw new InternalError("Could not create extension class loader");  9         } 10         try { 11             //初始化system classloader，parent是extension classloader 12             loader = AppClassLoader.getAppClassLoader(extclassloader); 13         } catch(IOException ioexception1) { 14             throw new InternalError("Could not create application class loader"); 15         } 16         //将system classloader设置成当前线程的context classloader（将在后面加以介绍） 17         Thread.currentThread().setContextClassLoader(loader); 18         http://www.blogjava.net/Images/dot.gifhttp://www.blogjava.net/Images/dot.gif 19     } 20     public ClassLoader getClassLoader() { 21         //返回system classloader 22         return loader; 23     } 24 } |

extension classloader的部分代码：

|  |
| --- |
| 1 static class Launcher$ExtClassLoader extends URLClassLoader {  2   3     public static Launcher$ExtClassLoader getExtClassLoader()  4         throws IOException  5     {  6         File afile[] = getExtDirs();  7         return (Launcher$ExtClassLoader)AccessController.doPrivileged(new Launcher$1(afile));  8     }  9    private static File[] getExtDirs() { 10         //获得系统属性“java.ext.dirs” 11         String s = System.getProperty("java.ext.dirs"); 12         File afile[]; 13         if(s != null) { 14             StringTokenizer stringtokenizer = new StringTokenizer(s, File.pathSeparator); 15             int i = stringtokenizer.countTokens(); 16             afile = new File; 17             for(int j = 0; j < i; j++) 18                 afile[j] = new File(stringtokenizer.nextToken()); 19  20         } else { 21             afile = new File[0]; 22         } 23         return afile; 24     } 25 } |

system classloader的部分代码：

|  |
| --- |
| 1 static class Launcher$AppClassLoader extends URLClassLoader  2 {  3   4     public static ClassLoader getAppClassLoader(ClassLoader classloader)  5         throws IOException  6     {  7         //获得系统属性“java.class.path”  8         String s = System.getProperty("java.class.path");  9         File afile[] = s != null ? Launcher.access$200(s) : new File[0]; 10         return (Launcher$AppClassLoader)AccessController.doPrivileged(new Launcher$2(s, afile, classloader)); 11     } 12 } |

通过源码我们可：

extension classloader是使用系统属性“java.ext.dirs”设置类搜索路径的，并且没有parent。system classloader是使用系统属性“java.class.path”设置类搜索路径的，并且有一个parent classloader。Launcher初始化extension classloader，system classloader，并将system classloader设置成为context classloader，但是仅仅返回system classloader给JVM

这里怎么又出来一个context classloader呢？它有什么用呢？我们在建立一个线程Thread的时候，可以为这个线程通过setContextClassLoader方法来 指定一个合适的classloader作为这个线程的context classloader，当此线程运行的时候，我们可以通过getContextClassLoader方法来获得此context classloader，就可以用它来载入我们所需要的Class。默认的是system classloader。利用这个特性，我们可以“打破”classloader委托机制了，父classloader可以获得当前线程的context classloader，而这个context classloader可以是它的子classloader或者其他的classloader，那么父classloader就可以从其获得所需的 Class，这就打破了只能向父classloader请求的限制了。这个机制可以满足当我们的classpath是在运行时才确定,并由定制的 classloader加载的时候,由system classloader(即在jvm classpath中)加载的class可以通过context classloader获得定制的classloader并加载入特定的class(通常是抽象类和接口,定制的classloader中是其实现),例 如web应用中的servlet就是用这种机制加载的.