# 继承结构

|  |
| --- |
|  |

# 重要属性-parent

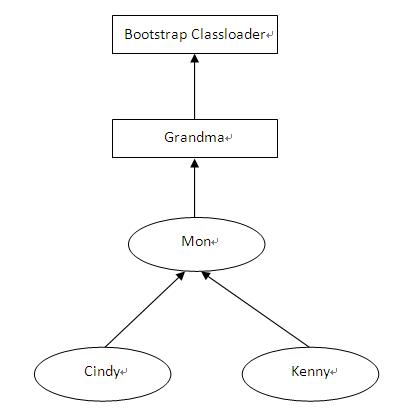
|  |
| --- |
| public abstract class ClassLoader {  //…  private final ClassLoader parent; //重要  //…  } |

# 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

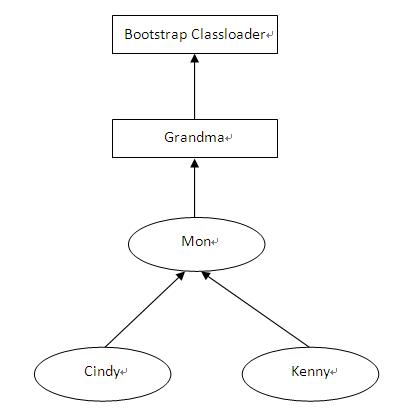


例如若果 要求cindy加载器去加载某目录（比如d://mon/lib）下的某个类，根据双亲委托机制，它会依次让bootstrap、grandma、mon、cindy来依次加载，但是最终却是由mon来加载，那么mon这个加载器就是定义类加载器。

总结：

如果某个类加载器能够真正加载一个类，那么该类加载器就称作：定义类加载器；

## 初始类加载器



例如若果 要求cindy加载器去加载某目录（比如d://grandma/lib）下的某个类，根据双亲委托机制，它会依次让bootstrap、grandma、mon、cindy来依次加载，但是最终却是由grandma来加载，那么mon和cindy这些加载器就是初始类加载器；同样若求求kenny来加载，情况类似；因此对于这个某个类来说mon、cindy、kenny都是初始类加载器。

换句话可以这样讲：

定义类加载器的所有子孙加载器都称作：初始类加载器。

## 自定义加载器

### 必须直接或间接继承自classloader

这没啥可说的，否则jvm就不会认定这个自定义加载器是类加载器。

|  |
| --- |
| ClassLoader#构造器-🡪  **protected** ClassLoader(ClassLoader parent) {//①  **this**(*checkCreateClassLoader*(), parent);  }  **protected** ClassLoader() {//②  **this**(*checkCreateClassLoader*(), *getSystemClassLoader*());  } |

### 自定义直接继承自Classloader时指定parent非空

从ClassLoader的①构造器可以看出，当parent为非null时，那么传入的parent就是当前待构造的加载器的parent。

### 自定义直接继承自classloader时指定parent为空

从ClassLoader的①构造器可以看出，当parent为null时，那么当前待构造的加载器的parent是boot类加载器（为什么？请看下面的代码（红色加重的））---有个前提就是我们采用的双亲委托机制；如果不采用双亲委托机制也就是说你重写类ClassLoader#loaderClass那这个parent究竟是谁，parent还有没有存在的价值就不好说了。

|  |
| --- |
| ClassLoader#loadClass🡪  **public** Class<?> loadClass(String name) **throws** ClassNotFoundException {  **return** loadClass(name, **false**);  }  **protected** **synchronized** Class<?> loadClass(String name, **boolean** resolve)  **throws** ClassNotFoundException  {  // First, check if the class has already been loaded  //从缓存中找  Class c = findLoadedClass(name);  //若缓存中没有，则让parent来加载  **if** (c == **null**) {  **try** {  //若parent不为null（），  **if** (parent != **null**) {  c = parent.loadClass(name, **false**);  } **else** {//若parent 为null，则证明其parent为boot，让boot来加载  **c = findBootstrapClassOrNull(name);**  }  } **catch** (ClassNotFoundException e) {  // ClassNotFoundException thrown if class not found  // from the non-null parent class loader  }  //若仍加载失败，则由自己来加载  **if** (c == **null**) {  // If still not found, then invoke findClass in order  // to find the class.  c = findClass(name);  }  }  **if** (resolve) {  resolveClass(c);  }  **return** c;  } |

### 自定义直接继承自classloader时 使用无参构造

从ClassLoader的②构造器可以看出，当使用无参构造器时，其 parent其实是系统加载器。

# 如何使用类加载器

## 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。

# 隐式/显式使用类加载

* 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。

# 常用的三个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。

# 双亲加载机制

通常情况下，我们使用的都是双亲加载机制，这个加载机制也是jvm推荐的，但是它是可以更改的（更改），也就是说可以不使用这种机制。比如osgi，比如tomcat等

## 双亲委托机制流程描述

* 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，若没有依旧委托父类加载器去加载，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。若当前类加载器加载失败则抛出未找到异常。

## 双亲委托机制代码描述

|  |
| --- |
| ClassLoader#loadClass🡪  **public** Class<?> loadClass(String name) **throws** ClassNotFoundException {  **return** loadClass(name, **false**);  }  **protected** **synchronized** Class<?> loadClass(String name, **boolean** resolve)  **throws** ClassNotFoundException  {  // First, check if the class has already been loaded  //从缓存中找  Class c = findLoadedClass(name);  //若缓存中没有，则让parent来加载  **if** (c == **null**) {  **try** {  //若parent不为null（），  **if** (parent != **null**) {  c = parent.loadClass(name, **false**);  } **else** {//若parent 为null，则证明其parent为boot，让boot来加载  c = findBootstrapClassOrNull(name);  }  } **catch** (ClassNotFoundException e) {  // ClassNotFoundException thrown if class not found  // from the non-null parent class loader  }  //若仍加载失败，则由自己来加载  **if** (c == **null**) {  // If still not found, then invoke findClass in order  // to find the class.  c = findClass(name);  }  }  **if** (resolve) {  resolveClass(c);  }  **return** c;  } |
| **缓存中加载🡪**  **protected** **final** Class<?> findLoadedClass(String name) {  **if** (!checkName(name))  **return** **null**;  **return** findLoadedClass0(name);  }  **private** **native** **final** Class findLoadedClass0(String name); |

## 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，就可以防止子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

本级的classLoder加载的class可以与本级别加载的class、上级classLoader加载的class，...直至顶级classloader加载的class进行互通，但本级的classLoder无法与其下级classloader加载的class互通。正常情况下没有这样的需求，但我们也说了是正常情况下，肯定存在非正常情况🡪上级classloader要使用下级classloader加载的class，此时双亲加载机制就无能为力!；不存在直接或间接关系命名空间下的类互相访问，此时双亲加载机制也无能为力！

## 如何放弃使用双亲加载机制

“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。

# 双亲加载机制下类加载器的命名空间

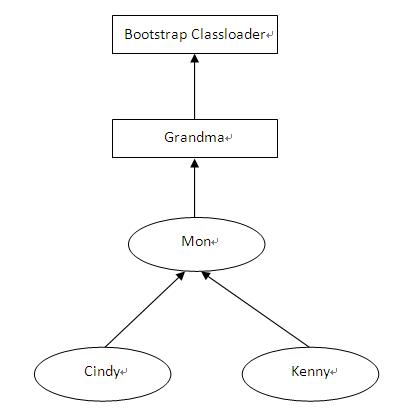
每个类加载器都有自己的命名空间，它维护着自己加载的class。在同一个命名空间中，不会出现类的完整名字（包括类的包名）相同的两个类，他们之间可以直接交互；在不同的命名空间中，有可能会出现类的完整名字（包括类的包名）相同的两个类，他们之间是否可以交互取决于这两个命名空间的类加载器是否存在直接或间接的父子关系：

* 若果不存在直接或间接父子关系，则不可以直接交互。
* 若存在直接或间接父子关系，则
* 子中的class可以与父中的class进行直接交互；
* 父中的class无法与子中的class进行直接交互。

注，这里说的是直接交互，什么是直接交互？请先阅读8.1。

## 不存在直接或间接关系命名空间下的class如何才能访问彼此？

### 把问题说明白



假定CindyClassLoader load的是“d://cindy//lib”中的cindy.jar，其中有个类是Cindy

假定KennyClassLoader load的是”d://kenny//lib”中的kenny.jarr，其中有个类是Kenny

那么Cindy.class和Kenny.Class在不同的命名空间下，假定Cindy类要使用Kenny类，

大家都不会采用的方式是这样：

让CindyClassLoader load也加载d://kenny//lib，这样Cindy类也可以使用Kenny类了，但是此时Cindy类使用的是CindyClassLoader命名空间中的Kenny类，它和KennyClassLoader命名空间中的中Cindy类除了名字相同外就没有任何关系了---如果Cindy类此时持有一个KennyClassLoader命名空间下Kenny类实例的一个引用，如果你想转型为CindyClassLoader命名空间下Kenny类型是会包转型失败的；同时这种方式也不太友好，至少它太浪费内存了，同样一份二进制代码，jvm中出现了两份。

大家都采用的方式：

让各自的classloader加载各自的类，避免重复加载；

这里就会出现一个问题，如果Cindy类想构造一个Kenny类实例，那么就让Cindy类想办法能够拿到KennyClassLoader，怎么拿到？通常做法是采用Thread#contextClassLoader方式。

拿到之后怎么使用？

不能直接调用（即强转为Kenny类后直接调用其方法或属性，强转会报转型失败异常，因为CindyClassLoader命名空间中没有Kenny.class）；

只能使用反射（反射先构造出Object对象，再找出method、fied利用Object对象进行访问）；

或者使用引用（这里的引用类型有个限制，那就是这个引用类型.class必须是CindyClassLoader命名空间和KennyClassLoader共有的class，这很难吗？至少Object.class是CindyClassLoader 和KennyClassLoader共有的类型吧（虚拟机中独有一份，它处于bootStrapClassloader命名空间中），所以引用类型你可以选择Object类型，但我们知道Object类型没有多大意义，这时我们就得想办法，构造一个有意义且是CindyClassLoader 和KennyClassLoader共有的类型，比如我们构造一个接口（假定为Comm），这个接口由Mom来加载，那么此时我们就可以将这个引用强转为Comm类型，我们使用Comm来调用方法或属性。

引用和反射是直指本质的（请看8.1.2），上面我们讨论了各类的Class对象都拥有自己的命名空间，那么不同命名空间（包括”不存在直接或间接关系命的命名空间”）中的Class对象对应的实例（比如Student.class对象对应的具体的10个、100个等等Student实例）（在堆中）有命名空间之说吗？没有，正因为没有我们才能使用反射使用引用进行访问啊！！！！！

### “引用”和“反射”直指本质

|  |
| --- |
| **public** **class** Sample **implements** Driver{  **public** String f1 = "i am is Sample field";  **public** Sample() {}  **public** Connection connect(String url, Properties info) **throws** SQLException {  **return** **null**;  }  **public** **boolean** acceptsURL(String url) **throws** SQLException {  **return** **false**;  }  **public** DriverPropertyInfo[] getPropertyInfo(String url, Properties info)  **throws** SQLException {  **return** **null**;  }  **public** **int** getMajorVersion() {  **return** 0;  }  **public** **int** getMinorVersion() {  **return** 0;  }  **public** **boolean** jdbcCompliant() {  **return** **false**;  }  } |
| **public** **class** SmallSample **extends** Sample{  **public** String f1 = "i am is SmallSample field";  **public** SmallSample() {}  **public** Connection connect(String url, Properties info) **throws** SQLException {  **return** **null**;  }  **public** **boolean** acceptsURL(String url) **throws** SQLException {  **return** **false**;  }  **public** DriverPropertyInfo[] getPropertyInfo(String url, Properties info)  **throws** SQLException {  **return** **null**;  }  **public** **int** getMajorVersion() {  **return** 1;  }  **public** **int** getMinorVersion() {  **return** 1;  }  **public** **boolean** jdbcCompliant() {  **return** **false**;  }  **public** **static** **void** main(String args[]){  //引用部分  SmallSample ss=**new** SmallSample();  Object ss\_o=ss;  System.*out*.println(ss\_o);//com.cht.SmallSample@10b30a7--不是object，不是Driver.class,也不是Sample.class  Driver ss\_d=(Driver)(ss\_o);  System.*out*.println(ss\_d.getMajorVersion());//1 -–调用的是SmallSample，不是Sample；driver引用也可直接使用。  Sample ss\_s=(Sample)ss\_o;  System.*out*.println(ss\_s.getMajorVersion());//1 -–调用的是SmallSample，不是Sample。  。  //反射部分  Class clazz=ss\_o.getClass();  Method method=clazz.getMethod("getMajorVersion", **null**);  System.*out*.println(method.invoke(ss\_o, **null**)); //1 -–调用的是SmallSample，不是Sample。  }  } |

## 父命名空间访问子命名空间中的calss

“父命名空间访问子命名空间中的calss”，是”不存在直接或间接关系命名空间下的class如何才能访问彼此” 的特例，这里不再细说。

### 实实在在的例子---jdbc驱动

它采用了引用方式来调用子命名空间中的类。

代码如下：

我们常使用一下语句注册jdbc驱动（以msql为例）-🡪存在于我们自定义工具类或第三方jar中：

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

打开com.mysql.jdbc.Driver的源代码-🡪第三方jar：

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

Java.sql.DriverManger---存在于jdk中:

|  |
| --- |
| **public** **class** DriverManager {  **public** **static** **synchronized** **void** registerDriver(java.sql.Driver driver)  **throws** SQLException {  **if** (!*initialized*) {  *initialize*();  }    DriverInfo di = **new** DriverInfo();  di.driver = driver;  di.driverClass = driver.getClass();  di.driverClassName = di.driverClass.getName();  // Not Required -- drivers.addElement(di);  *writeDrivers*.addElement(di);  *println*("registerDriver: " + di);    /\* update the read copy of drivers vector \*/  *readDrivers* = (java.util.Vector) *writeDrivers*.clone();  } |
| **public** **static** Connection getConnection(String url,  String user, String password) **throws** SQLException {  java.util.Properties info = **new** java.util.Properties();  // Gets the classloader of the code that called this method, may  // be null.  ClassLoader callerCL = DriverManager.*getCallerClassLoader*();  **if** (user != **null**) {  info.put("user", user);  }  **if** (password != **null**) {  info.put("password", password);  }  **return** (*getConnection*(url, info, callerCL));  } |
| // Worker method called by the public getConnection() methods.  **private** **static** Connection getConnection(  String url, java.util.Properties info, ClassLoader callerCL) **throws** SQLException {  java.util.Vector drivers = **null**;  /\*  \* When callerCl is null, we should check the application's  \* (which is invoking this class indirectly)  \* classloader, so that the JDBC driver class outside rt.jar  \* can be loaded from here.  \*/  **synchronized**(DriverManager.**class**) {  // synchronize loading of the correct classloader.  **if**(callerCL == **null**) {  callerCL = Thread.*currentThread*().getContextClassLoader();  }  }    **if**(url == **null**) {  **throw** **new** SQLException("The url cannot be null", "08001");  }    *println*("DriverManager.getConnection(\"" + url + "\")");    **if** (!*initialized*) {  *initialize*();  }  **synchronized** (DriverManager.**class**){  // use the readcopy of drivers  drivers = *readDrivers*;  }  // Walk through the loaded drivers attempting to make a connection.  // Remember the first exception that gets raised so we can reraise it.  SQLException reason = **null**;  **for** (**int** i = 0; i < drivers.size(); i++) {  DriverInfo di = (DriverInfo)drivers.elementAt(i);    // If the caller does not have permission to load the driver then  // skip it.  **if** ( *getCallerClass*(callerCL, di.driverClassName ) != di.driverClass ) {  *println*(" skipping: " + di);  **continue**;  }  **try** {  *println*(" trying " + di);  Connection result = di.driver.connect(url, info);  **if** (result != **null**) {  // Success!  *println*("getConnection returning " + di);  **return** (result);  }  } **catch** (SQLException ex) {  **if** (reason == **null**) {  reason = ex;  }  }  }    // if we got here nobody could connect.  **if** (reason != **null**) {  *println*("getConnection failed: " + reason);  **throw** reason;  }    *println*("getConnection: no suitable driver found for "+ url);  **throw** **new** SQLException("No suitable driver found for "+ url, "08001");  } |
| /\* Returns the caller's class loader, or null if none \*/  **private** **static** **native** ClassLoader getCallerClassLoader();  } |

Java.sql.Driver（spi，例如mysql的驱动需要实现该接口。）---存在于jdk中:

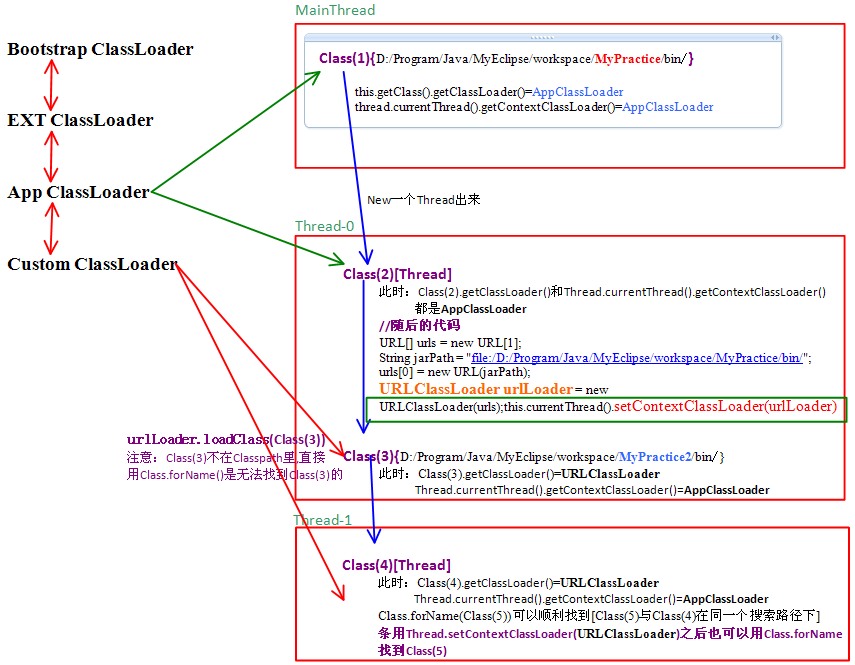
|  |
| --- |
| **public** **interface** Driver {  Connection connect(String url, java.util.Properties info)  **throws** SQLException;  **boolean** acceptsURL(String url) **throws** SQLException;  DriverPropertyInfo[] getPropertyInfo(String url, java.util.Properties info)  **throws** SQLException;  **int** getMajorVersion();  **int** getMinorVersion();  **boolean** jdbcCompliant();  } |

# **Thead#contextClassLoade**r

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.

为什么不设定，默认是system classloader呢？请看以下源码，另外：ContextClassLoader具有连带性，比如A线程的ContextClassLoader是classCloaderA，A线程没有重设ContextClassLoader，则B线程的ContextClassLoader依旧是classCloaderA。

|  |
| --- |
| 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 } |



# 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.