Java底层方法记录

1. 反射

1.1 Reflection#getCallerClass - link

作用：得到调用者的类

参数：

```
1. `0`/`负整数` - 返回Reflection.class
2. `正整数` - 层层向上查找调用类
```

补充（2019-03-05）：发现 `jdk.internal.reflect.Reflection` 类已经修改，并且不能被直接引入（import）；

1.2 返回修饰符

方法：Class#**getModifiers**

解释：修饰符都被包装成一个int类型的数字，这样每个修饰符都是一个位标识(flag bit)。

使用：可以使用Modifier#isXxx(int)来判断是修饰符

```
public static boolean isPrivate(int mod) {
    return (mod & PRIVATE) != 0;
}

public static final int PUBLIC           = 0x00000001;
public static final int PRIVATE          = 0x00000002;
public static final int PROTECTED        = 0x00000004;
public static final int STATIC           = 0x00000008;
```

1.3 包信息

方法：Class#**getPackage()**

1.4 父类

方法：Class#**getSuperclass()**

1.5 实现接口

方法：Class#**getInterfaces()**

1.6 变量

方法：Class#**getFields**

解释：在通常的观点中从对象的外部访问私有变量以及方法是不允许的，但是 Java 反射机制可以做到这一点，要想获取私有变量你可以调用 Class.getDeclaredField(String name)方法或者 Class.getDeclaredFields()方法。

获取私有变量示例：

```
public class PrivateObject {

  private String privateString = null;

  public PrivateObject(String privateString) {
```

```
    this.privateString = privateString;
  }
}
PrivateObject privateObject = new PrivateObject("The Private Value");

Field privateStringField = PrivateObject.class.
           getDeclaredField("privateString");

/* 注意 privateStringField.setAccessible(true)这行代码，通过调用 setAccessible()方法会关闭指定类 Field 实例的反射访问检查，这行代码执行之后不论是私有的、受保护的以及包访问的作用域，你都可以在任何地方访问，即使你不在他的访问权限作用域之内。*/
// privateStringField.setAccessible(true)等于彻底放开的权限，
// 既可以通过(String) privateStringField.get(privateObject)获取属性值
// 也可以通过privateObject.privateString直接访问属性值
privateStringField.setAccessible(true);

String fieldValue = (String) privateStringField.get(privateObject);
System.out.println("fieldValue = " + fieldValue);
```

1.7 方法

访问一个私有方法你需要调用 Class.getDeclaredMethod(String name, Class[] parameterTypes)或者 Class.getDeclaredMethods() 方法。 Class.getMethod(String name, Class[] parameterTypes)和 Class.getMethods()方法，只会返回公有的方法，无法获取私有方法。

```
public class PrivateObject {

  private String privateString = null;

  public PrivateObject(String privateString) {
    this.privateString = privateString;
  }

  private String getPrivateString(){
    return this.privateString;
  }
}
PrivateObject privateObject = new PrivateObject("The Private Value");

Method privateStringMethod = PrivateObject.class.
        getDeclaredMethod("getPrivateString", null);

// 和上面的私有变量同样的原理
privateStringMethod.setAccessible(true);

String returnValue = (String)
        privateStringMethod.invoke(privateObject, null);

System.out.println("returnValue = " + returnValue);
```

1.8 泛型 link

说明：当你声明一个类或者接口的时候你可以指明这个类或接口可以被参数化， java.util.List 接口就是典型的例子。你可以运用泛型机制创建一个标明存储的是 String 类型 list，这样比你创建一个 Object 的list 要更好。
当你想在运行期参数化类型本身，比如你想检查 java.util.List 类的参数化类型，你是没有办法能知道他具体的参数化类型是什么。这样一来这个类型就可以是一个应用中所有的类型。

示例：

```
public class MyClass {

  protected List<String> stringList = ...;

  public List<String> getStringList(){
    return this.stringList;
  }
}
```

```
Method method = MyClass.class.getMethod("getStringList", null);

Type returnType = method.getGenericReturnType();

if(returnType instanceof ParameterizedType){
    ParameterizedType type = (ParameterizedType) returnType;
    Type[] typeArguments = type.getActualTypeArguments();
    for(Type typeArgument : typeArguments){
        Class typeArgClass = (Class) typeArgument;
        System.out.println("typeArgClass = " + typeArgClass);
    }
}
```

示例二：

```
public class MyClass {
  protected List<String> stringList = ...;

  // 参数是参数化类型
  public void setStringList(List<String> list){
    this.stringList = list;
  }
}
Method method = Myclass.class.getMethod("setStringList", List.class);

Type[] genericParameterTypes = method.getGenericParameterTypes();

for(Type genericParameterType : genericParameterTypes){
    if(genericParameterType instanceof ParameterizedType){
        ParameterizedType aType = (ParameterizedType) genericParameterType;
        Type[] parameterArgTypes = aType.getActualTypeArguments();
        for(Type parameterArgType : parameterArgTypes){
            Class parameterArgClass = (Class) parameterArgType;
            System.out.println("parameterArgClass = " + parameterArgClass);
        }
    }
}
```

2. Java安全模型 参考博文

2.1 概念

2.1.1 本地和远程代码

在 Java 中将执行程序分成本地和远程两种，本地代码默认视为可信任的，而远程代码则被看作是不受信的。

2.1.2 包

主要类所在包：java.security

2.2 衍进

- 而对于非授信的远程代码在早期的 Java 实现中，安全依赖于沙箱 (Sandbox) 机制。沙箱机制就是将 Java 代码限定在虚拟机 (JVM) 特定的运行范围中，并且严格限制代码对本地系统的资源访问，通过这样的措施来保证对远程代码的有效隔离，防止对本地系统造成破坏。
- Java1.1 版本中，针对安全机制做了改进，增加了安全策略，允许用户指定代码对本地资源的访问权限。
- 在 Java1.2 版本中，再次改进了安全机制，增加了代码签名。不论本地代码或是远程代码，都会按照用户的安全策略设定，由类加载器加载到虚拟机中权限不同的运行空间，来实现差异化的代码执行权限控制。
- 最新的安全机制实现，则引入了域 (Domain) 的概念

  虚拟机会把所有代码加载到不同的系统域和应用域，系统域部分专门负责与关键资源进行交互，而各个应用域部分则通过系

统域的部分代理来对各种需要的资源进行访问。虚拟机中不同的受保护域 (Protected Domain)，对应不一样的权限 (Permission)。存在于不同域中的类文件就具有了当前域的全部权限，

2.3 继承

> 有一种特殊的情况，就是访问控制上下文的继承问题。当一个线程创建另一个新线程时，会同时创建新的堆栈。如果创建新线程时没有保留当前的安全上下文，也就是线程相关的安全信息，则新线程调用 AccessController.checkPermission 检验权限时，安全访问控制机制只会根据新线程的上下文来决定安全性问题，而不会考虑其父线程的相应权限。这个清除堆栈的做法本身并不会给系统带来安全隐患，但它会使源代码，尤其是系统代码的编写容易出现错误。例如，对安全框架实现不熟悉编程人员可能会很自然地认为，子线程执行的信任代码继承了父线程执行的不可信任代码的安全限制特性。当从子线程内访问受控制的资源时，如果父线程的安全上下文信息并未保存，就会导致意外的安全漏洞。因为丢失的父线程中安全限制数据会使子线程将资源传递给一些不可信任的代码。因此，在创建新线程时，必须确保利用父线程创建，或利用其他形式创建代码。总之，要保证让子线程自动继承父线程的安全性上下文，这样子线程中的后续 AccessController.checkPermission 调用就会考虑所继承的父线程的安全特性。

- 需要注意是 AccessController 类的 checkPermission 方法将在当前执行线程的上下文，包括继承的上下文中进行安全检查。

3. 类加载器

3.1 加载资源

```
// 获取资源url
Url url = ClassLoader.getSystemClassLoader().getResources("application.yml");
// 获取资源properties
Properties properties = PropertiesLoaderUtils.loadProperties(new UrlResource(url));
```