### 1. 什么是Java反射，有什么用？

反射使程序代码能够接入装载到JVM中的类的内部信息，允许在编写与执行时，而不是源代码中选定的类协作的代码，是以开发效率换运行效率的一种手段。这使反射成为构建灵活应用的主要工具。

反射可以：

1. 调用一些私有方法，实现黑科技。比如双卡短信发送、设置状态栏颜色、自动挂电话等。
2. 实现序列化与反序列化，比如PO的ORM，Json解析等。
3. 实现跨平台兼容，比如JDK中的SocketImpl的实现
4. 通过xml或注解，实现依赖注入(DI)，注解处理，动态代理，单元测试等功能。比如Retrofit、Spring或者Dagger

### 2. Java Class文件的结构

在\*.class文件中，以Byte流的形式进行Class的存储，通过一系列Load，Parse后，Java代码实际上可以映射为下图的结构体，这里可以用javap命令或者IDE插件进行查看。

typedef struct {

u4 magic;/\*0xCAFEBABE\*/

u2 minor\_version; /\*网上有表可查\*/

u2 major\_version; /\*网上有表可查\*/

u2 constant\_pool\_count;

cp\_info constant\_pool[constant\_pool\_count-1];

u2 access\_flags;

u2 this\_class;

u2 super\_class;

u2 interfaces\_count;

u2 interfaces[interfaces\_count];

//重要

u2 fields\_count;

field\_info fields[fields\_count];

//重要

u2 methods\_count;

method\_info methods[methods\_count];

u2 attributes\_count;

attribute\_info attributes[attributes\_count];

}ClassBlock;

常量池(constant pool):类似于C中的DATA段与BSS段，提供常量、字符串、方法名等值或者符号（可以看作偏移定值的指针）的存放

access\_flags: 对Class的flag修饰

typedef enum {

ACC\_PUBLIC = 0x0001,

ACC\_FINAL = 0x0010,

ACC\_SUPER = 0x0020,

ACC\_INTERFACE = 0x0200,

ACC\_ACSTRACT = 0x0400

}AccessFlag

this class/super class/interface: 一个长度为u2的指针，指向常量池中真正的地址，将在Link阶段进行符号解引。

filed: 字段信息，结构体如下

typedef struct fieldblock {

char \*name;

char \*type;

char \*signature;

u2 access\_flags;

u2 constant;

union {

union {

char data[8];

uintptr\_t u;

long long l;

void \*p;

int i;

} static\_value;

u4 offset;

} u;

} FieldBlock;

method: 提供descriptor, access\_flags, Code等索引，并指向常量池：

它的结构体如下，详细在这里

method\_info {

u2 access\_flags;

u2 name\_index;

//the parameters that the method takes and the

//value that it return

u2 descriptor\_index;

u2 attributes\_count;

attribute\_info attributes[attributes\_count];

}

### 3. Java Class加载的过程

lass的加载主要分为两步

第一步通过ClassLoader进行读取、连结操作

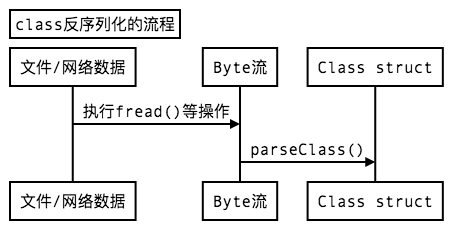
第二步进行Class的<clinit>()初始化。

#### 3.1. Classloader加载过程

ClassLoader用于加载、连接、缓存Class，可以通过纯Java或者native进行实现。在JVM的native代码中，ClassLoader内部维护着一个线程安全的HashTable<String,Class>，用于实现对Class字节流解码后的缓存，如果HashTable中已经有了缓存，则直接返回缓存；反之，在获得类名后，通过读取文件、网络上的class字节流反序列化为JVM中native的C结构体，接着malloc内存，并将指针缓存在HashTable中。

下面是非数组情况下ClassLoader的流程

find/load: 将文件反序列化为C结构体。



Class反序列化的流程

link: 根据Class结构体常量池进行符号的解引。比如对象计算内存空间，创建方法表，native invoker，接口方法表，finalizer函数等工作。

#### 3.2. 初始化过程

当ClassLoader加载Class结束后，将进行Class的初始化操作。主要执行<clinit()>的静态代码段与静态变量（取决于源码顺序）。

在完成初始化后，就是Object的构造<init>了，

### 4. 反射在native的实现

反射在Java中可以直接调用，不过最终调用的仍是native方法，以下为主流反射操作的实现。

#### 4.1. Class.forName的实现

Class.forName可以通过包名寻找Class对象，比如Class.forName("java.lang.String")。

在JDK的源码实现中，可以发现最终调用的是native方法forName0()，它在JVM中调用的实际是findClassFromClassLoader()，原理与ClassLoader的流程一样，具体实现已经在上面介绍过了。

#### 4.2. getDeclaredFields的实现

在JDK源码中，可以知道class.getDeclaredFields()方法实际调用的是native方法getDeclaredFields0()，它在JVM主要实现步骤如下

根据Class结构体信息，获取field\_count与fields[]字段，这个字段早已在load过程中被放入了

根据field\_count的大小分配内存、创建数组

将数组进行forEach循环，通过fields[]中的信息依次创建Object对象

返回数组指针

主要慢在如下方面

创建、计算、分配数组对象

对字段进行循环赋值

#### 4.3. Method.invoke的实现

以下为无同步、无异常的情况下调用的步骤

创建Frame

如果对象flag为native，交给native\_handler进行处理

在frame中执行java代码

弹出Frame

返回执行结果的指针

主要慢在如下方面

需要完全执行ByteCode而缺少JIT等优化

检查参数非常多，这些本来可以在编译器或者加载时完成

#### 4.4. class.newInstance的实现

检测权限、预分配空间大小等参数

创建Object对象，并分配空间

通过Method.invoke调用构造函数(<init>())

返回Object指针

主要慢在如下方面

参数检查不能优化或者遗漏

<init>()的查表

Method.invoke本身耗时