Java中如何获得A<T>泛型中T的运行时类型及原理探究

文章来源：企鹅号 - 阿里技术

简介

如果经常写工具类，很大概率会遇到一个比较实际的问题，就是需要在泛型表达式A中获取T的运行时类型。获取它需要一些技巧。但这个技巧很少被透彻的解释过为什么会生效。在接下来的文章里，我们会从Java的泛型（Generics）谈起，结合JLS（Java语言标准，Java Language Specification）和JVMS（Java虚拟机标准，Java Virtual Machine Specification），通过javac编译过程对泛型处理的源码，结合JRE反射API源码的探索，最后以一种虚拟机的实现（OpenJDK8的hotspot）来验证，来从根本上解答这个问题。

引言

在写工具类或者使用泛型精简逻辑的时候，经常需要获取运行时的类型信息，用来做下一步的逻辑判断，比如业务里经常使用的插件式架构，或者反序列化时，获取类型的详细信息。比如，应该有不少新手比较会试图写如下代码来通过反射和泛型获取类型的运行时类型。

是的。以上的代码，可以在一定范围内工作的很好。但是，当遇到泛型的时候，就事与愿违了。可以说，Java从1.5引入泛型后，对于反射中泛型的处理，就一直是一个比较麻烦的事情。

泛型（Generics）

泛型是Java1.5引入的特性，主要是为了解决类型检查（Type Checking）问题 [1]，为我们编写通用代码（尤其是库）时提供强有力的类型约束，而不用担心1.5之前满天乱飘的cast导致heap pollution[13]。

为什么要引入泛型，泛型设计的思路，可以参考Gilad Bracha在1998年OOPLSA会议上的论文 [2]。后来形成了JSR14：Adding Generics to the Java Programming Language

add generics in java[14]，并最终在1.5进入JDK。

泛型的形式化定义参考JLS[3],[4],[5],[6]

泛型的非形式化定义可以参考下面的简单代码。其中比如很容易混淆的几个概念，Type Variable和Type Parameter和Type Argument也一并做了说明

反射（Reflection）

因为1.5引入了泛型，所以反射也针对新概念，做了相应的扩展7。

在实现上，反射引入了Type接口，以及派生接口和类，实现了泛型JLS的标准。它们的UML类型如下。

其中我们需要打最多交道的，就是ParameterizedType[10]

ParameterizedType可能是一个比较陌生的概念。但是经常用反射（Core Reflection ）API的开发者可能比较熟悉，Type类型的一个派生类就是ParameterizedType。这个概念的非形式化（大白话）解释，可以简单的类比为泛型类型Foo的一个实现。比如Foo，Foo就分别是Foo的ParameterizedType

同时伴随着泛型落地，反射API里增加了一系列名称中含有Generic的方法和类[8]，这些都是之后我们获得泛型运行时类型的基础。

类型擦除（Type Erasure）

虽然Java引入了泛型这个概念，但为了保持向前兼容（JVM层面的兼容，不需要改动Bytecode和JVM设计），和编译性能（相比C++的Template会以模板参数生成新的类型），Java引入了类型擦除[9]作为泛型的实现方案。通过类型擦除，Java不需要在虚拟机实现上做任何修改，同时也不会为ParameterizedType创建新的类。

类型擦除的缺陷

正如所有设计都存在权衡。Java泛型通过类型擦除获得了好处同时也导致了两个主要问题。

Java泛型是Compile-time（编译期的）的，也就是说在运行时，所有泛型信息都被抹除了。所以JVM无法感知类型的存在。

所以我们无法通过反射API，在运行期获得Type Variable所代表的类型。

但是这个特性导致我们在写工具类时会遇到一些困难。比如无法单独通过T来创建实例。如果T是一个非泛型类还好，我们可以通过直接传入类型信息进行一些操作

但当T是一个ParameterizedType时，上述接口里的tClass类型信息，也只能获得ParameterizedType里非泛型的类型信息。比如T位List时，Class就是List.class。在一些场景，比如反序列化时，会遇到一些麻烦。

获取泛型的运行时类型的技巧

引入TypeReference

那么，在Java中，就没有办法获得泛型的运行时类型了吗？答案是可以的。但是我们需要做一些改动。比如，我们可以在很多序列化框架（比如Jackson，Fastjson）里看到TypeReference或者类似的设计机制。在“基本”不改动函数签名的情况下获得T的运行时类型。

我们定义类

它非常简单，基本就类似一个包装类。然后再做一个简单的方法定义

最后通过一个小技巧，就是创建匿名派生类的实例，配合反射API，先获取superClass的泛型信息，如果是ParameterizedType，就尝试获取真实的Type Argument信息，就可以获取T的运行时类型了。

比如以下两个语句中，唯一的区别就是第2行创建了一个Wrapper的匿名类

但是最终运行后的结果，则分别打印

那么，为什么仅仅一个匿名类的实例就可以产生这么巨大的差别？还可以在类型擦除的框架下完成泛型类型的获取？其中的原理是什么？

其实它利用了JSR14[14]中，对类中的泛型信息，保存到类签名（Signature）的一个技巧。

Classfiles need to carry generic type information in a backwards compatible way. This is accomplished by introducing a new “Signature” attribute for classes, methods and fields.

首先，Java的编译器将泛型信息写入到ClassFile的Signature属性中。然后通过JRE的反射接口解析Signature中的字符串。最终“扒”出被隐藏的运行时类型信息。下面，我们从JVMS的定义开始，研究Java代码编译并产生ClassFile的过程，和JRE反射代码一探究竟。

原理分析

JVM的ClassFile标准

JVM的ClassFile就是Java源文件编译后产生的二进制格式。类似于Linux下的ELF或者Windows的COFF，可以简单的理解为JVM的“可执行文件”。JVM通过读取它，并执行bytecode，最终执行程序的运行。ClassFile的格式如下:

其中的attributes数组里，就是JSR14里提到的，泛型信息的保存所在。JVMS指出[11]

A Java compiler must emit a signature for any class, interface, constructor, method, or field whose declaration uses type variables or parameterized types

可以看到Java编译器需要把泛型类信息带到Signature这个attribute，然后存储于编译后的ClassFile里。

做一个小实验

我们简单继承一下Wrapper类，编译后通过javap验证下结论

javap后，可以看到42行类的Signature已有相应的类型信息了（Lcom/aliyun/cwz/model/Wrapper;）。基本验证了JVMS标准

那么Java编译器是如何操作的呢?

JavaCompiler探秘

根据《The Hitchhiker's Guide to javac》[15]，JavaCompiler是javac的driver。所以我们可以通过研究JavaCompiler的实现，来研究javac编译文件的过程（通过strace javac的编译过程，可以看到调用里含有JavaCompiler。基本可以得到相同的结论）。

我们使用OpenJDK1.8，看下JavaCompiler是如何编译ExtendedWrapper的。

首先在编译过程中JavaCompiler类调用了compile方法。这是核心编译方法，最终会把Java类输出为ClassFile。

```
2352        * Loading Classes
2353        ************************************************************************/
2354
                Define a new class given its name and owner.
2357        public ClassSymbol defineClass(Name name, Symbol owner) {
2358            ClassSymbol c = new ClassSymbol( flags: 0, name, owner);
2359            if (owner.kind == PCK)
2360                Assert.checkNull(classes.get(c.flatname), c);
2361            c.completer = thisCompleter;
2362            return c;
2363        }
2364
```

中将类型信息带到ExtendedWrapper类所产生的ClassSymbol的type变量里，并将ClassSymbol存入符号表。

在下一步的代码生成过程中

在方法

中通过符号表中ClassSymbol的type，通过方法

```
public void writeClassFile(OutputStream out, ClassSymbol c)
    throws IOException, PoolOverflow, StringOverflow {
    Assert.check( cond: (c.flags() & COMPOUND) == 0);
    databuf.reset();
    poolbuf.reset();
    signatureGen.reset();
    pool = c.pool;
    innerClasses = null;
    innerClassesQueue = null;
    bootstrapMethods = new LinkedHashMap<DynamicMethod, MethodHandle>();

    Type supertype = types.supertype(c.type);
    List<Type> interfaces = types.interfaces(c.type);
```

获得superClass的类型信息，并通过ClassWriter写入到ClassFile的Signature这个attribute里。

```
1698        boolean sigReq =
1699            typarams.length() != 0 || supertype.allparams().length() != 0;
1700        for (List<Type> l = interfaces; !sigReq && l.nonEmpty(); l = l.tail)
1701            sigReq = l.head.allparams().length() != 0;
1702        if (sigReq) {
1703            Assert.check(source.allowGenerics());
1704            int alenIdx = writeAttr(names.Signature);
1705            if (typarams.length() != 0) signatureGen.assembleParamsSig(typarams);
1706            signatureGen.assembleSig(supertype);
1707            for (List<Type> l = interfaces; l.nonEmpty(); l = l.tail)
1708                signatureGen.assembleSig(l.head);
1709            databuf.appendChar(pool.put(signatureGen.toName()));
1710            signatureGen.reset();
1711            endAttr(alenIdx);
```

JRE（Java Runtime Environment）源码

我们现在分析JRE中的反射API是如何把Signature字符串，转换为Type类型的。我们注意到之前代码主要用到的反射方法就是Class类下的getGenericSuperclass方法。我们从这里开始分析。这是1.5时引入的一个方法，用于返回带有泛型的基类。它的代码如下:

我们可以注意到核心是第2行产生的ClassRepository类型的变量，它代表类的泛型类型信息。具体如下

This class represents the generic type information for a class. The code is not dependent on a particular reflective implementation. It is designed to be used unchanged by at least core reflection and JDI.

深入getGenericInfo，我们可以看一下info到底如何产生

可以看到是来自方法getGenericSignature0字符串signature，通过ClassRepository的处理，最终产生了info。那么，这个字符串从哪里来呢？我们跟踪一下就可以发现，这是一个native方法。来自JVM实现。

既然是JVM的方法，那么我们可以翻一下源码来验证下是否符合之前提到的JVMS标准。

OpenJDK源码验证

通过JVMS的学习，结合JRE的Class#getGenericSignature0，我认为有必要探索一下JVM里面的具体实现。基于之上使用的JavaCompiler源码，所以最终决定参考OpenJDK的实现。我们参考内部使用更广泛的jdk8[12]，

搜索函数名getGenericSignature0，可以发现有一个JNI方法的配置数组在文件./jdk/src/share/native/java/lang/Class.c里

其中getGenericSignature0对应JNINativeMethod这个struct的一个对象。{"getGenericSignature0", "()"STR, (void *)&JVM_GetClassSignature}

JNINativeMethod定义如下

可见getGenericSignature0对应的jvm实现是JVM_GetClassSignature，一个函数指针。

这个函数的实现在./hotspot/src/share/vm/prims/jvm.cpp，包裹在JVM_ENTRY宏里。

可以看到，最终getGenericSignature0是从InstanceKlass::cast(k)->generic_signature方法获得。而这个方法使用_generic_signature_index这个序号从ClassFile的Symbol数组里获取相关数据。与javac编译过程的源码和JVMS标准是符合的。

对应jvm里Class文件格式里的Signature[11]

结论

我们从Java的泛型开始，研究反射针对泛型的扩展，类型擦除的影响。然后通过生成匿名类实例的小技巧，获得了泛型的运行时类型的技巧。

然后根据JVM标准，javac的编译过程，通过JRE源码入手，研究了JVM对泛型获取的实现，了解了底层原理。比较好的解决了这个问题。

参考链接：

【1】: https://docs.oracle.com/javase/tutorial/java/generics/why.html

【2】: https://homepages.inf.ed.ac.uk/wadler/gj/Documents/gj-oopsla.pdf

【3】: https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.1.2

【4】: https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4.4

【5】: https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.8.4

【6】: https://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.1.2

【7】: https://docs.oracle.com/javase/1.5.0/docs/guide/reflection/enhancements.html

【8】: https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/class-use/Type.html

【9】: https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.6

【10】: https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.5

【11】: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.9.1

【12】: https://github.com/openjdk/jdk8u

【13】: https://docs.oracle.com/javase/tutorial/java/generics/nonReifiableVarargsType.html#heap_pollution

【14】https://jcp.org/aboutJava/communityprocess/review/jsr014/index.html

【15】https://openjdk.org/groups/compiler/doc/hhgtjavac/index.html

发表于: 2023-05-25
原文链接：https://kuaibao.qq.com/s/20230525A01GMR00?refer=cp_1026

上一篇：酷开会员丨版权时代，酷开科技打造更多优质内容服务消费者
下一篇：能够写论文的AI有哪些？推荐几种好用的AI工具

同媒体快讯

- Java编程技巧之单元测试用例简化方法（内含案例） 2023-09-20
- Java 缺失的特性：扩展方法 2023-09-20
- 蚂蚁实时计算团队的AntFlink提交攻坚之路 2023-09-20
- 写出易维护的代码｜React开发的设计模式及原则 2023-09-20
- 干货总结｜快速构造String对象及访问其内部成员的技巧 2023-09-20
- 足球黑科技之AI与足球智能分析 2023-09-20

相关快讯

- Java的泛型编程与多态，重载的同与不同 2023-09-20
- Kotlin第七讲-泛型在Java和Kotlin上的差异 2023-09-20
- Kotlin的独门秘籍Reified实化类型参数 2023-09-20
- NET进阶篇-语言章-1-Generic泛型深入 2019-10-16
- 深入理解 Java 泛型 2023-09-20