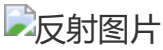
转载请注明链接：https://blog.csdn.net/feather_wch/article/details/81771443

> 介绍Java的反射机制，包括底层原理、使用场景、性能分析、性能优化、相关API的使用，以及最后的补充面试题。

Java 反射详解(76题)

版本：2018/8/23-1(22:30)



基本知识(12题)

反射是什么

1、反射是什么?

> 1. 反射是Java语言中一个相当重要的特性
> 2. 反射允许正在运行的Java程序去观测和修改程序的动态行为。
> 3. 通过 `java.lang.reflect.*` 保障的类可以实现反射。

2、Java中识别对象和类信息的两种方法?

> 传统RTTI(它假定我们在编译时已经知道了所有的类型信息)；反射机制(它允许我们在运行时发现和使用类的信息)

3、RTTI(RunTime Type Information)是什么?

> 1. 所有的类型信息都必须在编译时已知。
> 2. 会在所有类第一次使用的时候，将class对象(保存在.class文件)动态加载到JVM。

4、 反射机制实现方法

> 1. class类和reflect类库对反射进行了支持，通过里面的Field、Method和Constructor等类，能够确定对象信息，而在编译时不需要知道类的任何事情。
> 2. 但是反射的.class文件在运行的时候也必须是已知的。
> 3. .class文件可以在本地也可以通过网络获得。

5、 反射和RTTI的本质区别

> RTTI：编译器在编译时打开和检查.class文件
> 反射：运行时打开和检查.class文件

6、JVM加载对象的方法

> 类的信息会存在方法区，类加载器会通过方法区的类信息，在堆Heap上创建一个类对象(Class对象)，这个类对象是唯一的，后续的New等操作都是通过这个唯一的类对象作为模板去进行new等操作。

7、反射的特点

> 1. 允许程序在运行时取得任何一个已知名称的class的内部信息，包括包括其modifiers(修饰符)，fields(属性)，methods(方法)等，并可于运行时改变fields内容或调用methods。
> 2. 可以更灵活的编写代码，代码可以在运行时装配，无需在组件之间进行源代码链接，降低代码的耦合度
> 3. 反射使用不当会造成很高的资源消耗！

使用场景

8、通过Class对象可以枚举该类中的所有方法

9、通过Method.Accessible绕过Java语言的访问权限

> 1. 能在私有方法所在类之外的地方调用该方法
> 2. 该类位于 `java.lang.reflect` 包，该方法继承自 `AccessibleObject`

10、IDE中输入点号能动态展示方法和字段就是利用反射

11、Java调试器能够在调试过程中枚举某一对象所有字段的值。

12、web开发中，用于提升框架的可拓展性。

> 1. 借助Java的反射机制，根据配置来加载不同的类。
> 2. Spring框架的依赖反转(IOC), 就是依赖于反射。

反射机制(29题)

1、Method

> 1. 继承自抽象类Executable，需要实现其 9个方法

```
public final class Method extends Executable {
    @Override
    public Class<?> getDeclaringClass()  {...}
    @Override
    public String getName()  {...}
    @Override
    public int getModifiers()  {...}
    @Override
    public TypeVariable<?>[] getTypeParameters()  {...}
    @Override
    public Class<?>[] getParameterTypes()  {...}
    @Override
    public Class<?>[] getExceptionTypes()  {...}
    @Override
    public String toGenericString() {...}
    @Override
    public Annotation[][] getParameterAnnotations() {...}
    @Override
    public AnnotatedType getAnnotatedReturnType() {...}
}
```

2、Method的invoke方法

```
// Method.java---调用method方法
public Object invoke(Object obj, Object... args) throws IllegalAccessException, IllegalArgumentE
{
    // 1、权限检查，调用祖父类AccessibleObject的checkAccess检查权限
    if (!override) {
        if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {
            Class<?> caller = Reflection.getCallerClass();
            checkAccess(caller, clazz, obj, modifiers);
        }
    }
    /**=======================================
     * 2、 会委派给MethodAccessor来处理
     *  MethodAccessor是一个接口，具有两个具体实现：
     *   1. 通过本地方法来实现反射调用
     *   2. 使用委派模式来实现反射调用
     *=======================================*/
    MethodAccessor ma = methodAccessor;             // read volatile
    if (ma == null) {
        // 获取到MethodAccessor
        ma = acquireMethodAccessor();
    }
    // MA调用invoke方法
    return ma.invoke(obj, args);
}

// Method.java-获取到MethodAccessor
private MethodAccessor acquireMethodAccessor() {
    // 1、检查是否已经创建了MethodAccessor
    MethodAccessor tmp = null;
    if (root != null) tmp = root.getMethodAccessor();
    if (tmp != null) {
        // 2、已经存在就直接返回
        methodAccessor = tmp;
    } else {
        // 3、不存在MethodAccessor，制造一个并且使其连接到root上
        tmp = reflectionFactory.newMethodAccessor(this);
        setMethodAccessor(tmp);
    }
    // 4、返回MA
    return tmp;
}
```

本地实现

3、本地方法实现反射调用的原理?

> 1. 当进入到Java虚拟机内部之后，就拥有了 `Method` 实例所指向方法的 具体地址
> 2. 此时反射就是准备好参数，然后进入 目标方法 。
> 3. 本质是Java层切换到C++层，再切换到Java层去调用该方法。(根据如下打印的调用栈，可以看出)

```
// 1、Java层
at java.lang.reflect.Method.invoke(Method.java:483)

// 2、C++层Native方法
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

// 3、Java层：target方法
at Main.target(Main.java:19)
```

4、反射调用的轨迹

> 实例：

```
public class Main {
    // 1、需要反射的方法
    public static void target(int i){
     // 2、打印出轨迹
        new Exception("#" + i).printStackTrace();
    }

    public static void main(String[] args) {
         // 3、反射方法并且调用
            Class c = Class.forName("Main");
            Method method = c.getMethod("target", int.class);
            method.invoke(null, 0);
    }


}
```

> 轨迹如下：

```
java.lang.Exception: #0
        at Main.target(Main.java:19)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
// 进入本地实现
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
// 进入委派实现
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
// Method.invoke进行反射调用
        at java.lang.reflect.Method.invoke(Method.java:483)
        at Main.main(Main.java:27)
```

5、反射为什么要采用委派实现作为中间层？为什么不直接交给本地实现？

> 1. 因为Java反射机制还设立一种 动态生成字节码的实现(动态实现)
> 2. 动态实现能直接使用 invoke指令 调用目标方法。
> 3. 采用 委派实现 可以在 本地实现 和 动态实现 中切换

动态实现

6、动态实现伪代码

> 省略检查者调用、参数检查的字节码

```
public class GeneratedMethodAccessor1 extends ...{
  @Overrides
  public Object invoke(Object obj, Object[] args){
    Main.target((int)args[0]);
    return null;
  }
}
```

7、动态实现和本地实现的对比

> 1. 动态实现比本地实现的 运行效率 要快 20倍 。
> 2. 动态实现无需经过Java到C++再到Java的转换。
> 3. 但是生成字节码十分耗时，如果只调用一次，本地实现要快上 3、4倍

8、JVM关于反射调用的阀值

> 1. 许多反射调用只会执行一次一次，因此JVM设置了 阀值-15 (可以通过 -Dsun.reflect.inflationThreshold = xxx 进行调整)
> 2. 反射调用次数在 16次及以下(前16次，0~15) 时，采用本地实现。
> 3. 反射调用次数达到 17次 时，采用动态实现(动态生成字节码)。
> 4. 将委派实现的委派对象切换至动态实现的过程，称之为 inflation

9、阀值测试流程

> 循环调用20次：

```
Class c = Class.forName("Main");
Method method = c.getMethod("target", int.class);
for(int i = 1; i <= 20; i++){
    method.invoke(null, i);
}
```

> 打印结果：

```
java.lang.Exception: #16
        at Main.target(Main.java:19)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:483)
        at Main.main(Main.java:28)

java.lang.Exception: #17
        at Main.target(Main.java:19)
// 采用动态实现，直接调用invoke方法
        at sun.reflect.GeneratedMethodAccessor1.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:483)
        at Main.main(Main.java:28)
```

1. 在第17在第17次时，委派对象会切换到动态实现（inflation）
2. 反射调用的inflation可以通过参数： `-Dsun.reflect.noinflation = true` 进行关闭

性能开销

10、反射调用的三步骤的性能损耗

1. Class.forName(): 会调用本地方法
2. Class.getMethod(): 会遍历该类的共有方法，没有匹配到，会去遍历父类的共有方法。
3. Metjod.invoke():

11、getMethod的空间性能

1. 会有额外的堆空间损耗：getMethod()等系列方法操作，会将查找的结果进行拷贝，并将拷贝进行返回。
2. 必须要避免在热点代码中，使用返回Method数组的getMethods()或者getDeclaredMethods(),因为数据拷贝，会损耗大量堆空间。

12、如何解决getMethod的空间性能

1. 需要对 `Class.forName()` 、 `Class.getMethod()` 的结果在应用程序中缓存。

直接调用

13、直接调用的性能分析(V0)

直接调用空实现方法20亿次，测试每1亿次的时间消耗。

```
// 测试代码
public class Main {
    public static void target(int i){
        // 空实现
    }
    public static void main(String[] args) {
      long current = System.currentTimeMillis();
      long averageTime = 0;
      // 1、循环20亿次
      for(int i = 1; i <= 2_000_000_000; i++){
          // 2、1亿次测试一次时间
          if(i % 100_000_000 == 0){
              long temp = System.currentTimeMillis();
              // 3、累加得到总时间
              averageTime += temp - current;
              current = temp;
            }
            target(128);
      }
      // 4、得到平均时间
      System.out.println("averageTime = " + averageTime / 20);
    }
}
```

> 1. CPU：i5-6500 @ 3.20GHz
> 2. RAM: 8GB
> 3. OS: Window7 64bit

每次均测试20亿次	第1次	第2次	第3次
1亿次平均时间	114ms	114ms	114ms

14、直接调用的性能分析

> 1. 直接调用 `Main.target()`-本文实例中反射调用的方法 的时间消耗和不调用 `Main.traget()` 的时间消耗一致。
> 2. Main.target()代码属于 热循环 ， 会触发即时编译，
> 3. Main.target会在即时编译时被 内联 ， 从而消除了调用的开销。

反射调用

15、反射调用的性能测试(V1)

> 反射调用空实现方法20亿次，测试每1亿次的时间消耗。

```
public class Main {
    public static void target(int i){
        // 空实现
        // new Exception("#" + i).printStackTrace();
    }
    public static void main(String[] args) throws ClassNotFoundException, NoSuchMethodException,

        Class c = Class.forName("Main");
        Method method = c.getMethod("target", int.class);

        long current = System.currentTimeMillis();
        long averageTime = 0;
        // 1、循环20亿次
        for(int i = 1; i <= 2_000_000_000; i++){
            // 2、1亿次测试一次时间
            if(i % 100_000_000 == 0){
                long temp = System.currentTimeMillis();
                // 3、累加得到总时间
                averageTime += temp - current;
                current = temp;
            }
            method.invoke(null, 128);
        }
        // 4、得到平均时间
        System.out.println("averageTime = " + averageTime / 20);
    }

}
```

1. CPU：i5-6500 @ 3.20GHz
2. RAM: 8GB
3. OS: Window7 64bit

每次均测试20亿次	第1次	第2次	第3次	比较
1亿次平均时间	354ms	360ms	348ms	114ms基数的3.11倍

16、反射调用V1版本的性能问题?

1. Method.invoke()是一个变长参数方法，字节码层面最后一个参数是Object数组。Java编译器在方法调用出会生成Object[参数数量]数组，并将参数存储到该数组中。---可以用javap查看。
2. Object数组不能存储基本类型，Java编译器会对传入的基本类型参数进行自动装箱。
3. 这两个操作造成性能开销外，还会占用堆内存，导致GC更加频繁。---可以用虚拟机参数 `-XX:+PrintGC` 查看(IDEA中Edit Configuration可以配置)

反射优化

17、如何优化反射调用中自动装箱的问题？(V2)

1. Java缓存了[-128, 127]中所有整数的Integer对象。

> 2. 可以将范围扩大至128（ `-Djava.lang.Integer.IntegerCache.high=128` ）
> 3. 也可以在调用前，循环外手动对128进行装箱，生成Integer对象。

每次均测试20亿次	第1次	第2次	第3次	优化比	比较
1亿次平均时间	263ms	265ms	263ms	性能提升26%	114ms基数的2.3倍

18、反射调用的性能测试(V3)

> 1. 对自动装箱进行优化.
> 2. 对变长参数的Object数组问题进行优化--在反射前提前新建好Object数组。减少开销。

```
Class c = Class.forName("Main");
Method method = c.getMethod("target", int.class);

long current = System.currentTimeMillis();
long averageTime = 0;
Object[] objects = new Object[1];
objects[0] = new Integer(127);
// 1、循环20亿次
for(int i = 1; i <= 2_000_000_000; i++){
    // 2、1亿次测试一次时间
    if(i % 100_000_000 == 0){
        long temp = System.currentTimeMillis();
        // 3、累加得到总时间
        averageTime += temp - current;
        current = temp;
    }
    method.invoke(null, objects);
}
// 4、得到平均时间
System.out.println("averageTime = " + averageTime / 20);
```

> 1. 发现并没有想象中的优化效果。
> 2. 反而比解决自动装箱优化的性能更糟糕，

每次均测试20亿次	第1次	第2次	第3次	比较
1亿次平均时间	294ms	295ms	294ms	114ms基数的2.57倍

19、V2版本性能优化出错分析

> 1. 在解决自动装箱后，查看GC状况，会发现该程序不会触发GC。
> 2. 原因在于：原本的反射调用被内联，并将新建Object[]数组判定为 不逃逸的对象 。
> 3. 即使编译器会对不逃逸对象，进行栈分配甚至虚拟分配---不占用堆控件。
> 4. 在循环外新建数组，无法让即使编译器判断是否 逃逸---中途会被更改 ，因此无法优化掉访问数组的操作。

20、反射调用的inflation机制和权限检查优化(V4)

> 1. infaltion机制： 可以关闭该机制从而取消委派实现，并且直接使用动态实现。 `(-Dsun.reflect.noInflation=true)`
> 2. 权限检查：反射调用都会检查目标方法的权限，该检查可以再Java中关闭。 `method.setAccessible(true);`

```
Class c = Class.forName("Main");
Method method = c.getMethod("target", int.class);
// 关闭权限检查
method.setAccessible(true);
```

> 1. 已经提前进行了自动装箱优化
> 2. 关闭了inflation机制
> 3. 关闭了权限检查

每次均测试20亿次	第1次	第2次	第3次	比较
1亿次平均时间	162ms	163ms	164ms	114ms基数的1.4倍

性能优化瓶颈

21、反射调用为什么能从3.11倍优化到1.4倍?

> 1. 主要是因为 即使编译器 中的 方法内联

22、反射调用优化的瓶颈？内联的瓶颈?

> 1. 反射调用优化的瓶颈也就是内联的瓶颈。
> 2. 在关闭了inflation的情况下，内联的瓶颈在于 `Method.invoke()` 中对 `MethodAccessor.invoke()` 的调用。

23、方法内联的结论

> 1. 实际项目中，会由多个不同的反射调用，也会对应多个 `GenerateMethodAccessor---`也就是动态实现 。
> 2. Java虚拟机关于上述调用点(多个反射调用，多个动态实现)的类型profile，无法同时记录这么多类，可能会造成反射调用没有被内联的情况。

24、反射调用和动态实现的关系。

> 实际调用时：
>
> 1. 反射调用A，唯一对应于，动态实现A。
> 2. 反射调用B，唯一对应于，动态实现B。

25、类型profile是什么?

> 1. JVM对于invokevirtual或者invokeinterface，会记录下调用者的具体类型。就称之为 类型`profile`

26、污染Profile导致的性能损伤(V5)

> 对profile进行污染：提前进行大量的其他反射调用，导致类型Profile无法记录测试的内容，导致无法内联。

```
public class Main {

    public static void target(int i){
        // 空实现
    }
    public static void main(String[] args) throws ClassNotFoundException, NoSuchMethodException,
        Class c = Class.forName("Main");
        Method method = c.getMethod("target", int.class);

        // === 调用前污染Profil e===
        polluteProfile();

        /**=======================
         *  原来的测试内容
         *====================*/
    }

// 污染Profile
    public static void polluteProfile() throws NoSuchMethodException, InvocationTargetException,
        Method method1 = Main.class.getMethod("target1", int.class);
        Method method2 = Main.class.getMethod("target2", int.class);
        for(int i = 0; i < 2000; i++){
            method1.invoke(null, 0);
            method2.invoke(null, 0);
        }
    }

    public static void target1(int i){}
    public static void target2(int i){}
}
```

> 1. 已经提进行了自动装箱优化、inflation机制优化、权限检查优化
> 2. 进行了profile污染

每次均测试20亿次	第1次	第2次	第3次	比较
1亿次平均时间	1006ms	1004ms	1005ms	114ms基数的8.8倍

27、Profile污染性能问题优化(V6)

> 1. 原因：没有内联、逃逸分析不再有效
> 2. 逃逸分析优化：采用(V3)关于变长参数的Object数组优化---循环外构造Object数组。
> 3. 提高JVM关于每个调用能够记录的类型数目： `-XX:TypeProfileWidth = xxx`

每次均测试20亿次	第1次	第2次	第3次	比较

每次均测试20亿次	第1次	第2次	第3次	比较
1亿次平均时间	828ms	828ms	828ms	114ms基数的7.26倍

28、反射调用和直接调用的性能差距

	直接调用(V0)	反射调用(V1)	反射调用(V2)	反射调用(V3)	反射调用(V4)	反射调用(V5)	反射调用(V6)
平均时间(1亿次)	114ms	354ms	264ms	294ms	163ms	1005ms	828ms
比例	1倍	3.11倍	2.3倍	2.57倍	1.4倍	8.8倍	7.26倍

> 1. 参照性能：直接调用(V0)
> 2. 性能最差：反射调用(V5)-提前大量调用其他反射，污染了类型Profile。这种情况可以从变长参数的Object[]数组进行优化，提高JVM关于每个调用能够记录的类型数目。
> 3. 性能最好：反射调用(V4)-没有污染Profiel，关闭Inflation，优化自动装箱，关闭权限检查。
> 4. 优化总结：
>   1. 自动装箱优化：能够有效提升效率。(V2)
>   2. Object数组优化：在有逃逸分析下，性能反而会变差。但是在没有逃逸分析的最差情况，该方法能实现优化。(V3变差，V6变好)
>   3. Inflation优化： 关闭Inflation能够尽早使用“动态实现”，但是如果只运行几次，性能整体变差。(V4)
>   4. 权限检查优化；关闭反射调用的权限检查。(V4)
>   5. 类型Profile：增加JVM关于每个调用能够记录的类型数目。(V6)

优化方法总结

29、反射调用优化有哪几个方面？(5)

> 1. 自动装箱优化：能够有效提升效率。
> 2. Object数组优化：在有逃逸分析下，性能反而会变差。但是在没有逃逸分析的最差情况，该方法能实现优化。
> 3. Inflation优化： 关闭Inflation能够尽早使用“动态实现”，但是如果只运行几次，性能整体变差。
> 4. 权限检查优化；关闭反射调用的权限检查。
> 5. 类型Profile：增加JVM关于每个调用能够记录的类型数目。

相关类(19题)

Class

1、Class常用API(一共64种)

> 1. getName()：获得类的完整名字。

2. getFields()：获得类的public类型的属性。
3. getDeclaredFields()：获得类的所有属性。包括private 声明的和继承类
4. getMethods()：获得类的public类型的方法。
5. getMethod(String name, Class[] parameterTypes)：获得类的特定方法，name参数指定方法的名字，parameterTypes 参数指定方法的参数类型。
6. getDeclaredMethods()：获得类的所有方法。包括private 声明的和继承类
7. getConstructors()：获得类的public类型的构造方法。
8. getConstructor(Class[] parameterTypes)：获得类的特定构造方法，parameterTypes 参数指定构造方法的参数类型。
9. newInstance()：通过类的不带参数的构造方法创建这个类的一个对象。
10. getComponentType();

2、Class的获取方法

1. Class.forName()
2. obj.getClass()
3. 类.class

3、通过各种方法获取到的同一类的Class实例，equals相比较的结果?

1. 一个类在 JVM 中只会有一个 Class 实例
2. 因此将c1,c2,c3进行 equals 比较，会发现为true。

4、Class的获取方法的性能哪个最高?

类.class ： 直接通过类的静态成员变量来获取。最可靠，最稳定，性能最高。

获取Class对象

forName

5、Class.forName()

```
//1、通过Class的forName方法来获取。但可能抛出 ClassNotFoundException 异常。
try {
    Class c3 = Class.forName("java.lang.String");
    Log.i("reflect", "c3 Is = " +  c3.getName());
} catch (ClassNotFoundException e) {
    e.printStackTrace();
    Log.i("reflect", "c3 's Class Not Found ");
}
```

getClass

6、obj.getClass()

1. 常用于获取到Object但是不知道其具体类型

```
String msg = "Message";

//1、通过对象调用getClass()方法来获取,通常应用在传过来一个Object对象,但是不知道其具体类型。
Object obj = msg;
Class c1 = obj.getClass();

Log.i("reflect", "c1 Is = " +  c1.getName());
```

.class

7、类.class

```
//1、直接通过类的静态成员变量来获取。最可靠，最稳定，性能最高。
Class c2 = String.class;
Log.i("reflect", "c2 Is = " +  c2.getName());
```

8、基本类型的包装类型的“TYPE”静态字段可以获取到Class对象

> 1. TYPE是final静态字段，指向基本类型对应的Class对象
> 2. `Integer.TYPE -> int.class`
> 3. 数组：`[].class` 如 `int[].class`

数组元素的Class对象

9、数组类的Class对象获取方法?

```
Class arrayClass1 = array.getClass();
Class arrayClass2 = int[].class;
```

10、获取到数组元素的类型

```
Class c = arrayClass.getComponentType();
```

构造实例

11、通过Class对象构造实例的方法?

```
// 1、构造实例
c.newInstance();
// 2、构造数组实例
Array.newInstance(arrayClass, 10);
// 3、判断对象是否是Class对象的实例。
c.isInstance(obj);
```

getField

12、getField()相关API

1. 获取成员变量。
2. Declared：返回自己的全部成员变量(包括public、private、protected、default)，不返回任何父类的成员变量。
3. 非Declared：返回自己和父类的 `public` 成员变量 ， 包括祖父类等层层继承的public 成员变量。

```
c.getField("Field name");
c.getFields();
c.getDeclaredField("Field name");
c.getDeclaredFields();
```

getMethod

13、getMethod()相关API

1. 获取方法。
2. Declared：返回自己的全部方法(包括public、private、protected、default)，不返回任何父类的方法。
3. 非Declared：返回自己和父类的 `public` 方法 (包括祖父类等层层继承的public 方法)。
4. Enclosing: 该类是在哪个方法中定义的，比如方法中定义的匿名内部类.

```
// 1、指定Method
c.getMethod("Method Name");
c.getDeclaredMethod("Method Name");

// 2、Methods
c.getMethods();
c.getDeclaredMethods();

// 3、c为局部或者匿名类的Class对象时，返回该类是在哪个方法中定义的。
c.getEnclosingMethod();
```

getConstructor

14、getConstructor

```
// 1、指定构造方法
c.getConstructor(); //为构造方法的参数类型，如int.class, String.class
c.getDeclaredConstructor();

// 2、Constructors
c.getConstructors();
c.getDeclaredConstructors();

// 3、c为局部或者匿名类的Class对象时，返回该类是在哪个构造函数中定义的。
c.getEnclosingConstructor();
```

setAccessible

15、setAccessible-绕开Java语言的访问限制

```
field.setAccessible(true);
method.setAccessible(true);
constructor.setAccessible(true);
```

Field

16、Filed访问字段值

```
Class c = Class.forName("Main");
Field field = c.getField("field");
Main obj = new Main();
// 从Obj这个对象中，获取到该Field的值。
field.get(obj);

// 将新值赋值到Obj对象对应的Field字段上。
field.set(obj, "新数值");
```

Constructor

17、生成类的实例

```
Constructor constructor = c.getConstructor(构造的参数类型的class);
constructor.newInstance(构造方法的参数);
```

Method

枚举所有方法

18、通过Class对象可以枚举该类中的所有方法

> getMethods()：获得类的public类型的方法。

```
try {
    Class c = Class.forName("java.lang.String");
    Method[] methods = c.getMethods();

    for (Method method : methods) {
        System.out.println(method.getName() + "");
    }
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

调用方法

19、Method.invoke()-调用某个方法

```
public class Main {
    // 需要被反射的方法
    public static void target(int i){
        new Exception("#" + i).printStackTrace();
    }

    public static void main(String[] args) {
        // 1、获取到Class对象
            Class c = Class.forName("Main");
        // 2、获取到方法，根据方法名和参数类型
            Method method = c.getMethod("target", int.class);
        // 3、调用该方法
            method.invoke(null, 0);
    }


}
```

补充题(16题)

1、Class对象存在的意义

> 1. 涉及到JVM的类加载机制。
> 2. 加载过程将类的class文件中的二进制数据读取到内存中, 会在堆区创建一个java.lang.Class对象。
> 3. Class对象封装了类在方法区内的数据结构。
> 4. 该Class对象向程序员提供了访问方法区内的数据结构的接口。

2、Class对象放在哪里的?

> 1. 堆区：在类的加载阶段，将类的class文件中的二进制数据读取到内存中, 会在堆区创建一个java.lang.Class对象。

3、Class.forName()执行的过程

```
Class.forName("ClassName");
```

> forName内部是走到Native方法，本质涉及到类的加载流程：
>
> 1. 加载：
>   1. 通过“类全名”定位到类文件，并获取该类文件的二进制字节流
>   2. class文件二进制字节流中的静态数据结构转换为方法区中动态的运行时数据结构
>   3. 在java堆中生成一个代表这个类的java.lang.Class对象，作为方法区中该类相关数据的访问入口
> 2. 连接-验证：目的在于，确保被加载类能够满足JVM的约束条件
>   1. 验证文件的各式是否正确：是否以魔数0xCAFEBABE开头，紧接着魔数是不是正确的主次版本号，主版本号最小45等等验证

2. 验证元数据是否合法：该类是否有父类（除了java.lang.object外都当有父类），是否继承了final修饰的类等等
3. 验证字节码是否安全：确保被验证类的方法在运行时不会做出危害虚拟机安全的行为，如保证跳转命令不会跳转到方法体以外的字节码命令上
4. 符号引用验证：对当前类意外的信息进行匹配性验证，如通过“类全限定名”是否可找到对应的类，方法是否可访问等等。
   - 需要重视的是：Java类在编译期会被检查验证(保证加载的类满足约束条件)，但是我们无法保证class文件不会被篡改，所以虚拟机的验证是非常重要的。

3. 连接-准备：
   1. 为类成员变量分配内存设置类型默认值（比如int，则设为0），这些内存都将在方法区中进行分配，但需要注意的是，若静态成员变量被final修饰，则此时会直接赋值，而不是取类型默认值。

4. 连接-解析：
   1. 将类的符号应用（单纯的符号字面量，不涉及jvm内存）转为直接应用（直接定位到目标的内存地址）；

5. 初始化：
   1. “初始化”是类加载过程的最后一步，在此阶段，类的静态成员将会被赋值（静态方法将会被执行）；

4、方法区是什么?

> 方法区是jvm运行时数据结构中的一部分，为所有线程共享（线程安全）。

5、方法区中存放的类信息有哪些?

1. 类的全限定名
2. 直接超类的全限定名（如果是Object，则没有超类）
3. 类的类型（类还是接口）
4. 类的访问修饰符（public，abstract，final等）
5. 所有的直接接口的全限定名
6. 常量池：主要存放该类的 字段，方法信息，类变量信息，装在该类的装载器的引用，类型引用等。

6、反射的意义

1. 允许正在运行的Java程序去观测和修改程序的动态行为。
2. 能增加灵活性：能在运行状态中，知道任意类的所有属性和方法。并且进行实例化、调用方法、获取属性的工作。

7、反射和动态代理的关系？Java代码是怎么运行的?

8、Method.invoke何时会被内联?

1. `Method.invoke` 一直会被内联。
2. 但是其内部的 `MethodAccessor.invoke()` 则不一定会被内联。

9、本地方法实现反射调用时，什么叫做Java层->C++层->Java层的切换?

> 根据“本地实现”时打印的调用栈:
>
> 1. Method.invoke()等层层调用的就是Java层代码。
> 2. NativeMethodAccessorImpl.invoke0(): C++层代码。
> 3. Main.target(): 最终切换到我们要调用的目标方法，属于Java层。

10、本地实现和动态实现的阀值是如何存储的?

> 作为 `NativeMethodAccessorImpl` 的一个字段来存储。

11、不逃逸的数组可以优化访问，这是如何实现的?

> 1. 不逃逸：就是指能确保中途不会改变。
> 2. “读数组”这个操作，会替换为之前写入数组的值。也就直接优化掉 数组相关的读写操作

12、为什么要避免在热点代码中使用返回Method数组的getMethods和getDeclaredMethods?

> 这些操作会获取一个Method数组，然后又将数组的所有内容复制一份，然后返回给调用者。会造成大量的堆空间消耗。

知识储备

自动装箱

13、自动装箱是什么?

方法内联

14、方法内联是什么?

> 1. 出现在编译阶段
> 2. 方法内联是指编译器在编译一个方法时，将某个方法调用的目标方法也纳入编译范围，并且用其返回值替代原方法的过程。

ASM

15、ASM是什么?

> 1. ASM是一个java字节码操纵框架.
> 2. 能用来动态生成类或者增强既有类的功能。
> 3. ASM 可以直接产生二进制 class 文件，也可以在类被加载入 Java 虚拟机之前动态改变类行为。
> 4. Java class 被存储在严格格式定义的 .class文件里，这些类文件拥有足够的元数据来解析类中的所有元素：类名称、方法、属性以及 Java 字节码（指令）。

> 5. ASM从类文件中读入信息后，能够改变类行为，分析类信息，甚至能够根据用户要求生成新类。

16、反射在达到阀值后从本地实现切换到动态实现时，是如何生成字节码的?

> 1. 新版本的JDK全都是使用 `ASM` 来生成字节码的。

参考资料

1. 反射简单介绍和简单实例
2. Class.forname和ClassLoader.loadClass的源码分析
3. Java动态代理与反射的关系
4. 深入剖析Java中的装箱和拆箱
5. 关于java字节码框架ASM的学习