JVM中的方法调用

版本：2018/9/10-1(23:59)

问题汇总

1. 可变长参数方法的重载
2. 官方文档建议避免重载可变长参数方法
3. 什么是重载?
4. 如何绕开多个方法名字相同、参数类型相同的限制?
5. 当一个类(具有多个方法，方法名相同、参数类型相同、但是返回值类型不同)出现在Java编译器的用户路径上时，如何确定应该调用哪个方法?
6. 重载方法的识别是在哪个阶段完成的?
7. Java编译器如何根据传入参数的声明类型来选取重载方法的?
8. 声明类型和实际类型的区别?
9. Java编译器在同一个阶段中找到了多个适配的方法，如何选择一个最合适的方法?
10. 重载对于从父类继承来的非私有同名方法有效吗?
11. JVM中不存在重载这一概念

12. 如果子类定义的方法和继承自父类的方法同名，但是参数类型相同，这两个方法有什么关系?
13. 父类和子类具有两个同名方法，都是静态方法，都是public。子类中的方法会隐藏了父类中的方法。
14. 重写是什么?
15. 方法重写是如何体现了Java的多态?
16. JVM是如何识别方法的？关键在于三部分
17. 方法描述符是什么?
18. 如果一个类中出现了方法名相同，方法描述符相同的方法，会在类加载的什么阶段报错?
19. JVM和Java语言规范对于方法的限制有哪些不同处?
20. JVM的重写和Java语言的重写并不完全相同
21. 什么是桥接方法？编译器如何通过生成桥接方法解决了Java和JVM重写语义不同的问题?
22. 什么是静态绑定?什么是编译时多态?
23. 什么是动态绑定?
24. 对于JVM，什么是真正的静态绑定?
25. 对于JVM，什么是真正的动态绑定?
26. Java字节码中与调用相关的指令一共有五种
27. invokeinterface实例
28. invokestatic实例
29. invokevirtual实例
30. invokespecial实例
31. invokeastatic和invokespecial，JVM能直接识别具体的目标方法.
32. invokevirtual和invokeinterface，JVM需要在执行中，根据调用者的动态类型，来确定目标方法。
33. 符号引用的作用?
34. 符号引用存储在哪里?
35. 符号引用的分类
36. 实例中的符号引用
37. 符号引用什么时候需要被替换为实际引用?
38. JVM如何解析 非接口符号引用 ，并替换为实际引用?
39. 非接口符号引用的解析过程所得到的结论
40. 隐藏和重写的区别
41. JVM如何解析 接口符号引用 ，并替换为实际引用?
42. 实际引用是什么?
43. 如何将java文件翻译成字节码文件
44. 虚方法是什么？JVM中的虚方法调用?
45. 类有一个非静态的public方法，该方法是否是虚方法?
46. 虚方法和设计模式的关系？设计模式为什么需要大量采用虚方法来实现多态?
47. 虚方法的性能低下?
48. 哪些场景下虚方法的调用开销可以完全消除?
49. 为什么JVM中虚方法调用的开销很低?
50. 虚方法的性能消耗在哪?
51. 需要动态绑定确定目标方法的方法都是虚方法

52. 静态绑定的都是非虚方法
53. 虚方法中也会涉及到静态绑定？虚方法指向的是final方法会怎么样？
54. JVM中如何去提高动态绑定的性能消耗？
55. 方法表的作用
56. 方法表是何时构造的？类加载的准备阶段做了哪些事情？
57. 方法表的底层实现
58. 方法表中会存储private方法，或者static方法？
59. 子类的方法表中是否包含父类方法表中的所有方法？
60. 方法表的实例
61. 方法表的动态绑定和静态绑定相比有哪些开销？
62. 动态绑定的内存解引用操作的性能损耗可以忽略不计吗？
63. Java栈帧是什么？何时创建并且初始化的？
64. 方法表就是存储类的所有方法的表？
65. 内联缓存是什么？
66. 针对多态的优化手段中，具有单态、多态和超多态
67. 如何区分多态和超多态？
68. 内联缓存有几种？
69. JVM采用哪种内联缓存方式？
70. 如何没有命中内联缓存，对于内联缓存中的内容，JVM会如何处理？
71. 内联缓存并不是真正的内联，依旧有固定开销。
72. getter/setter等方法的固定开销超过了方法本身，这就需要方法内联的优化
73. HotSpot没有多态内联缓存？
74. 单态内联缓存和超多态内联缓存的性能差距？

重载(11)

1、什么是重载？

> 1. Java中，如果同一个类中有多个方法：名字相同，参数类型相同。会无法通过编译。
> 2. 如果想要在同一个类中定义名字相同的方法，参数类型必须不同。
> 3. Java根据参数类型的不同，去选择对应的方法，称之为重载。

2、如何绕开多个方法名字相同、参数类型相同的限制？

> 1. 可以通过字节码工具绕开
> 2. 编译完成后,再向class文件中添加方法名相同、参数类型相同、但是返回值类型不同的方法。

3、当一个类(具有多个方法，方法名相同、参数类型相同、但是返回值类型不同)出现在Java编译器的用户路径上时，如何确定应该调用哪个方法？

> 1. 目前java编译器会直接选取第一个方法名以及参数类型匹配的方法

2. 并且根据返回值类型，判断是否可以通过编译，是否需要进行值转换

4、重载方法的识别是在哪个阶段完成的?

编译阶段

5、Java编译器如何根据传入参数的声明类型来选取重载方法的?

1. 第一阶段：在不考虑对基本类型自动装拆箱，以及可变长参数的情况下选取
2. 第二阶段：没有找到，在允许自动装拆箱，不允许可变长参数的情况下选取
3. 第三阶段：还没找到，在允许自动装拆箱，允许可变长参数的情况下选择

6、声明类型和实际类型的区别?

1. 声明类型：(Object)str，声明类型就是Object类型。
2. 实际类型：(Object)str，实际类型是String

```
String str = "str";
// 声明类型为String
invoke(str, 1);
// 声明类型为Object
invoke((Object)str, 2);
```

7、Java编译器在同一个阶段中找到了多个适配的方法，如何选择一个最合适的方法?

1. 这个符合程度的决定性因素就是 形式参数类型的继承关系
2. 比如传入参数null，可以是Object，也可以是String。因为String是子类，所以Java编译器认为String更符合。

```
public static void invoke(Object obj) {
        System.out.println("Object");
}
public static void invoke(String s) {
        System.out.println("String");
}
// null可以是Object，也可以是String
invoke(null);
```

8、重载对于从父类继承来的非私有同名方法有效吗?

1. 子类具有一个方法，和继承自父类的方法，名称相同，参数类型不同。这就属于重载。

9、JVM中不存在重载这一概念

1. 重载方法的区分处于编译阶段

可变长参数的方法重载

10、可变长参数方法的重载

```
void invoke(Object obj, Object... args) { ... }
void invoke(String s, Object obj, Object... args) { ... }

invoke(null, 1);// 调用第二个 invoke 方法
invoke(null, 1, 2);// 调用第二个 invoke 方法 invoke(null, 1, 2);
invoke(null, new Object[]{1});// 调用第一个 invoke 方法 ,只有手动绕开可变长参数的语法糖，才能调用第
invoke((Object)null, 1); //调用第一个invoke方法
```

> 1. 在具有Object和String参数的情况下，默认都会去调用第二个方法。

11、官方文档建议避免重载可变长参数方法

重写(4)

1、如果子类定义的方法和继承自父类的方法同名，但是参数类型相同，这两个方法有什么关系?

> 1. 如果这两个方法是静态方法，也不是私有的，子类中的方法隐藏了父类中的方法
> 2. 如果都不是静态方法，也不是私有的，子类就是重写了父类的方法。

2、父类和子类具有两个同名方法，都是静态方法，都是public。子类中的方法会隐藏了父类中的方法。

> 1-父类和子类

```
public class People {
    public static void print(){
        System.out.println("People");
    }
}
public class Student extends People{
    public static void print(){
        System.out.println("Student");
    }
}
```

> 2-无法通过子类去调用到父类的方法。

```
Student.print();
```

> 3-如果子类没有print()方法，可以通过子类类名来调用父类的静态方法

```
Student.print();
// 打印
People
```

3、重写是什么?

> 1. 子类和父类具有两个方法名相同，参数类型相同，并且都是非静态方法，非私有方法。这就是重写。
> 2. 无法通过子类对象去调用到父类该方法，只能在子类内部，通过super.xxx()来调用。

4、方法重写是如何体现了Java的多态?

> 允许子类在继承父类部分功能的同时，拥有自己独特的行为

静态绑定和动态绑定(10)

1、JVM是如何识别方法的？关键在于三部分

> 1. 类名
> 2. 方法名
> 3. 方法描述符(method descriptor)

2、方法描述符是什么?

> 1. 由方法的类型参数，以及返回类型，所构成。
> 2. 如果同一个类中出现了名字相同、方法描述符相同的方法，JVM会在类的验证阶段报错

3、如果一个类中出现了方法名相同，方法描述符相同的方法，会在类加载的什么阶段报错?

> 类的验证阶段

4、JVM和Java语言规范对于方法的限制有哪些不同处?

> 1. Java不允许方法名、参数类型同时都相同。不能通过返回值类型进行区分。
> 2. JVM允许通过返回类型不同来区分方法。(字节码中的方法描述符，具有返回类型)

5、JVM的重写和Java语言的重写并不完全相同

> 在子类和父类中具有同名的方法，并且非静态，非私有的前提下。
>
> 1. JVM需要参数类型和返回类型都一致
> 2. Java语言只需要参数类型一致。
> 3. 这些区别，编译器会通过生成桥接方法来实现Java中的重写语义

6、什么是桥接方法？编译器如何通过生成桥接方法解决了Java和JVM重写语义不同的问题?

7、什么是静态绑定?什么是编译时多态?

> 1. 一般认为重载，就是静态绑定，也成为编译时多态。
> 2. 并不准确：某个类中的重载方法可能被子类所重写。
> 3. 此时Java编译器会将，所有，对实例的非私有方法的调用编译为需要动态绑定的类型。

8、什么是动态绑定?

> 1. 方法在执行过程中，JVM会获取到调用者的实际类型，在该实际类型的虚方法表中，根据索引值去获得目标方法。这个过程就是动态绑定。
> 2. 重写是动态绑定中的一种。

9、对于JVM，什么是真正的静态绑定?

> 1. 在解析时便能够直接识别目标方法的情况
> 2. 对于静态绑定的方法调用而言，实际引用是一个指向方法的指针。

10、对于JVM，什么是真正的动态绑定?

> 1. 在运行过程中，根据调用者的动态类型来识别目标方法的情况。
> 2. 对于动态绑定的方法调用而言，实际引用是一个方法标的索引。

调用指令(17)

1、Java字节码中与调用相关的指令一共有五种

> 1. invokestatic: 调用静态方法
> 2. invokespecial：调用实例的私有方法、构造器、以及super关键字所调用的父类的实例方法或者构造器，和所实现接口的默认方法
> 3. invokevirtual：调用实例的public方法
> 4. invokeinterface：调用接口方法
> 5. invokedynamic：调用动态方法

invokeinterface

2、invokeinterface实例

> 调用接口的方法

```
interface Listener{
    void onClick();
}
Listener listener = new Listener() {xxx};
// invokeinterface  InterfaceMethod Main$Listener.onClick:"()V",  1;
listener.onClick();
```

invokestatic

3、invokestatic实例

> 调用static方法

```
public class Main {
    public static void main(String args[]) {
      // invokestatic   Method printHello:"()V";
        printHello();
    }
    public static void printHello(){
        System.out.println("Hello World!");
    }
}
```

invokevirtual

4、invokevirtual实例

> 调用对象的public方法

```
public class Main {
    public static void main(String args[]) {
        Main main = new Main();
        // invokevirtual        Method printHello:"()V";
        main.printHello();
    }
    public void printHello(){
        System.out.println("Hello World!");
    }
}
```

invokespecial

5、invokespecial实例

> 1. 调用父类的构造方法
> 2. 调用父类的实例方法
> 3. 调用父类所实现的接口方法
> 4. 调用子类的构造方法
> 5. 调用子类的private实例方法

```
public class Student extends People{
    public Student(){
        // 1、调用父类的构造器
        super();
        // 2、调用父类的实例方法
        super.parentMethod();
        // 3、调用父类实现的接口方法
        super.run();
    }

    public Student(String name){
        // 4、子类的构造器
        this();
        // 5、子类的私有实例方法
        this.childMethod();
    }

    private void childMethod(){
    }
}
```

6、invokeastatic和invokespecial，JVM能直接识别具体的目标方法.

7、invokevirtual和invokeinterface，JVM需要在执行中，根据调用者的动态类型，来确定目标方法。

符号引用

8、符号引用的作用?

> 1. 编译过程中，不知道目标方法的具体内存地址
> 2. 编译器会暂时用符号引用来表示该目标方法
> 3. 该符号引用会包括：目标方法所在类或者接口的名字、目标方法的方法名、方法描述符

9、符号引用存储在哪里?

> 1. class文件的常量池中
> 2. javap -v xxx.class可以打印出常量池.

```
// Main.java文件
public class Main {
    interface Listener{
        void onClick();
    }
    public static void main(String args[]) {
        Listener listener = new Listener() {xxx};
        listener.onClick();
    }
}
```

```
// Main.class文件的常量池
Constant pool:
   #1 = Methodref          #6.#18         // java/lang/Object."<init>":()V
   #2 = Class              #19            // Main$1
   #3 = Methodref          #2.#18         // Main$1."<init>":()V
   #4 = InterfaceMethodref #7.#20         // Main$Listener.onClick:()V
   #5 = Class              #21            // Main
   #6 = Class              #22            // java/lang/Object
   #7 = Class              #23            // Main$Listener
   #8 = Utf8               Listener
   #9 = Utf8               InnerClasses
  #10 = Utf8               <init>
  #11 = Utf8               ()V
  #12 = Utf8               Code
  #13 = Utf8               LineNumberTable
  #14 = Utf8               main
  #15 = Utf8               ([Ljava/lang/String;)V
  #16 = Utf8               SourceFile
  #17 = Utf8               Main.java
  #18 = NameAndType        #10:#11        // "<init>":()V
  #19 = Utf8               Main$1
  #20 = NameAndType        #24:#11        // onClick:()V
  #21 = Utf8               Main
  #22 = Utf8               java/lang/Object
  #23 = Utf8               Main$Listener
  #24 = Utf8               onClick
```

10、符号引用的分类

1-接口符号引用

```
 #4 = InterfaceMethodref #7.#20         // Main$Listener.onClick:()V
```

2-非接口符号引用

```
   #3 = Methodref          #2.#18         // Main$1."<init>":()V
```

11、实例中的符号引用

1. `#3` 就是符号引用， `invokespecial #3` 就是调用 `#3` 所表示的方法

```
public static void main(java.lang.String[]);
    xxx
         4: invokespecial #3                  // Method Main$1."<init>":()V
         7: astore_1
         8: aload_1
         9: invokeinterface #4,  1            // InterfaceMethod Main$Listener.onClick:()V
        14: return
    xxx
}
```

实际引用

12、符号引用什么时候需要被替换为实际引用?

> 1. 执行使用了符号引用的字节码前,JVM需要解析符号引用，并替换为实际引用
> 2. 类加载的解析阶段

13、JVM如何解析 非接口符号引用 ，并替换为实际引用?

> 1. 在指向的类C中查找名字符合、描述符符合的方法
> 2. 如果没有找到，在类C的父类中继续搜索，直至Object类。
> 3. 如果还是没有找到，会在类C直接或者间接实现的接口中搜索。
> 4. 第三步搜索到的目标方法必须是public、非static的方法。
> 5. 第三步如果是间接实现的接口中，则需要满足类C和该接口之间没有其他符合条件的目标方法。(比如类C实现了接口1，接口1继承接口2，接口2继承接口3.那么类C和接口3之间，就隔着接口1)。如果有多个符合条件的目标方法，则返回其中任意一个。

14、非接口符号引用的解析过程所得到的结论

> 1. 静态方法也可以通过子类来调用
> 2. 子类的静态方法会隐藏父类中同名、同描述符的静态方法。

15、隐藏和重写的区别

> 1. 隐藏的方法：都需要是static、public的同名、同描述符方法。
> 2. 重写的方法：都需要是非static、public的同名、同描述符方法。

16、JVM如何解析 接口符号引用 ，并替换为实际引用?

> 1. 在接口I中查找名字符合、描述符符合的方法
> 2. 如果没有找到，在Object类中的public、非static方法(实例方法)中搜索
> 3. 如果没有找到，则在接口I的超接口中搜索。
> 4. 第三步搜索到的目标方法必须是public、非static的方法。

17、实际引用是什么?

> 1. 对于静态绑定的方法调用而言，实际引用是一个指向方法的指针。
> 2. 对于动态绑定的方法调用而言，实际引用是方法表的索引值。

虚方法(28)

1、虚方法是什么？JVM中的虚方法调用?

> 1. invokevirtual 调用指令调用的方法。
>   - 也就是对象的public方法，就是虚方法。

> 2. invokeinterface 调用指令调用的方法。
>   - 调用接口的方法。

2、类有一个非静态的public方法，该方法是否是虚方法？

> 是的！需要在运行时才能确定具体的目标方法

2、虚方法和设计模式的关系？设计模式为什么需要大量采用虚方法来实现多态？

> 1. 设计模式大量采用虚方法来实现多态

3、虚方法的性能低下？

> 1. 虚方法有一定的性能开销
> 2. 但是在JVM中虚方法调用的性能开销微乎其微，一定场景下甚至可以完全消除。

4、哪些场景下虚方法的调用开销可以完全消除？

> 1. 内联缓存，但是仍然会有固定开销
> 2. 进行方法内联，能完全消除固定开销

5、为什么JVM中虚方法调用的开销很低？

> 1. 方法表的开销，并不是很高。
> 2. 通过内联缓存、方法内联，能进一步降低虚方法调用的开销。

6、虚方法的性能消耗在哪？

> 1. 虚方法需要JVM根据调用者的动态类型，来确定虚方法调用的目标方法
> 2. 这个过程就是 动态绑定 ，相比于静态绑定的非虚方法调用，肯定会有一定的额外开销。

7、需要动态绑定确定目标方法的方法都是虚方法

8、静态绑定的都是非虚方法

> 1. 静态绑定的方法都是非虚方法。
> 2. invokestatic指令，调用的方法，就是非虚方法。
> 3. invokespecial指令，调用的方法，也是非虚方法。(子类构造器、子类对象的私有方法、super调用的父类构造器、super调用的父类的public实例方法、super调用的父类实现的接口方法)

9、虚方法中也会涉及到静态绑定？虚方法指向的是final方法会怎么样？

> 1. 如果一个虚方法，指向的目标方法，使用 `final` 修饰
> 2. JVM会去静态绑定该虚方法的目标方法

方法表

10、JVM中如何去提高动态绑定的性能消耗?

> 1. JVM采用 空间换时间 的策略去实现动态绑定。
> 2. 为每个类生成一张方法表，用于快速定位目标方法

11、方法表的作用

> 1. 这个数据结构，是JVM实现动态绑定的关键所在。
> 2. 每个类都有一个方法表，用于快速定位目标方法

12、方法表是何时构造的？类加载的准备阶段做了哪些事情?

> 1. 类加载的准备阶段，会为 静态字段 分配内存。
> 2. 准备阶段，还会构造与该类相关联的 方法表

13、方法表的底层实现

> 1. 本质是一个数组，每个数组元素是一个 指针 ，指向当前类和祖先类中public的实例方法(非static)
> 2. 子类方法表中，包含父类方法表中的所有方法。
> 3. 子类方法如果重写了父类方法，在方法表中的索引值(index)，和所重写的父类方法的索引值相同。

14、方法表中会存储private方法，或者static方法?

> 错误!
>
> 1. private的方法，都不是虚方法。
> 2. static方法，对应invokestatic指令，都不是虚方法。

15、子类的方法表中是否包含父类方法表中的所有方法?

> 是的

16、方法表的实例

乘客的方法表

0	乘客.toString()
1	乘客.出境()（备注：抽象方法，不可执行）

外国人的方法表

0	乘客.toString()
1	外国人.出境()

中国人的方法表

0	乘客.toString()
1	中国人.出境()
2	中国人.买买买()

> 1. 遇到新乘客时，会先去判断是中国人还是外国人。这里是 获取动态类型
> 2. 根据动态类型，找到该目标的方法表。
> 3. 然后根据 索引，例如1 ，找到需要执行的方法。

17、方法表的动态绑定和静态绑定相比有哪些开销?

> 多出几个内存解引用操作:
>
> 1. 访问栈上的调用者，读取调用者的动态类型
> 2. 读取该类型的方法表
> 3. 读取方法表中某个索引(index)所对应的目标方法

18、动态绑定的内存解引用操作的性能损耗可以忽略不计吗?

> 1. 理论上，这些操作相当于创建并且初始化Java栈帧的操作来说开销可以忽略不计。
> 2. 理论上的低开销，只存在于 解释执行 中，或者存在于 即使编译代码 的最坏情况。
> 3. 实际上，即时编译拥有性能更好的 内联缓存-inlining cache 和 方法内联-method inlining

19、Java栈帧是什么？何时创建并且初始化的?

20、方法表就是存储类的所有方法的表?

> 错误!
>
> 1. 只存储虚方法
> 2. 其他方法，都在静态绑定后，类加载的解析阶段(将符号引用替换为实际引用)时指向了目标方法。不需要方法表。

内联缓存

21、内联缓存是什么?

> 1. 一种加快动态绑定的优化技术
> 2. 会去缓存虚方法调用中 调用者的动态类型 ，以及 该类型所对应的目标方法
> 3. 之后的执行中，如果碰到已缓存的类型，会直接调用该类型对应的目标方法。
> 4. 如果没有碰到已缓存的类型，就会去使用原始的基于方法表的动态绑定。

22、针对多态的优化手段中，具有单态、多态和超多态

> 1. 单态-monomorphic，仅有一种状态的情况。
> 2. 多态-polymorphic，有限数量种状态的情况。二态-bimorphic，是多态中的一种。
> 3. 超多态-megamorphic，更多种状态的情况。

23、如何区分多态和超多态?

> 有一个阈值

24、内联缓存有三种

> 1. 单态内联缓存: 只缓存了一种动态类型和所对应的目标方法。如果动态类型命中，直接调用目标方法。
> 2. 多态内联缓存：缓存了多个动态类型和所对应的目标方法。会将当前动态类型和缓存的动态类型依次比较，命中就调用目标方法。
> 3. 超多态内联缓存

25、JVM采用哪种内联缓存方式?

> 1. 实践中会将热门的动态类型，放到前面。
> 2. 此外，由于大部分虚方法都是单态的，所以只有一种动态类型
> 3. 为节省内存空间，JVM采用单态内联缓存

26、如何没有命中内联缓存，对于内联缓存中的内容，JVM会如何处理?

> 1. 第一种：替换单态内联缓存中的记录，类似于CPU的数据缓存，需要在一定时间内，调用者的动态类型要保持一致。最坏情况下，会导致内联缓存完全失效。
> 2. 第二种：劣化为超多态，放弃了优化的机会，直接访问方法表，来动态绑定目标方法。节省了写缓存的额外开销。这是JVM的具体实现。

27、内联缓存并不是真正的内联，依旧有固定开销。

> 1. 保存程序在该方法中的执行位置
> 2. 新建、压入、弹出新方法所使用的栈帧
> 3. 这些都是性能开销，除非方法被内联，才不会有性能开销。

28、getter/setter等方法的固定开销超过了方法本身，这就需要方法内联的优化

1. 方法内联不仅消除了方法调用的固定开销
2. 方法内联还进一步增加了优化的可能性。

29、HotSpot没有多态内联缓存

1. 只有单态：单态内联缓存
2. 超多态：方法表+索引

30、单态内联缓存和超多态内联缓存的性能差距

1. 单态内联缓存：10亿次，平均每1亿次，时间130ms
2. 超多态内联缓存：10亿次，平均每1亿次，时间164ms
3. 超多态内联缓存的性能，相比于单态内联缓存，下降了26%

```
public class Main {
    public static abstract class Passenger{
        abstract void exit();
    }
    public static class Chinese extends Passenger{
        @Override
        void exit() {
        }
    }

    public static class American extends Passenger{
        @Override
        void exit() {
        }
    }
    public static void main(String args[]) {
        Passenger a = new Chinese ();
        Passenger b = new American();
        long current = System.currentTimeMillis();
        for (int i = 1; i <= 2_000_000_000; i++) {
            if (i % 100_000_000 == 0) {
                long temp = System.currentTimeMillis();
                System.out.println(temp - current);
                current = temp;
            }
            Passenger c = (i < 1_000_000_000) ? a : b;
            c. exit ();
        }
    }
}
```

JVM参数，关掉方法内联。

```
-XX:CompileCommand=dontinline
```

知识扩展

1、如何将java文件翻译成字节码文件

```
// 生成class文件
javac Student.java
// 将class文件转换成字节码文件
java -cp .\asmtools.jar org.openjdk.asmtools.jdis.Main Student.class >Student.jasm
```