目录

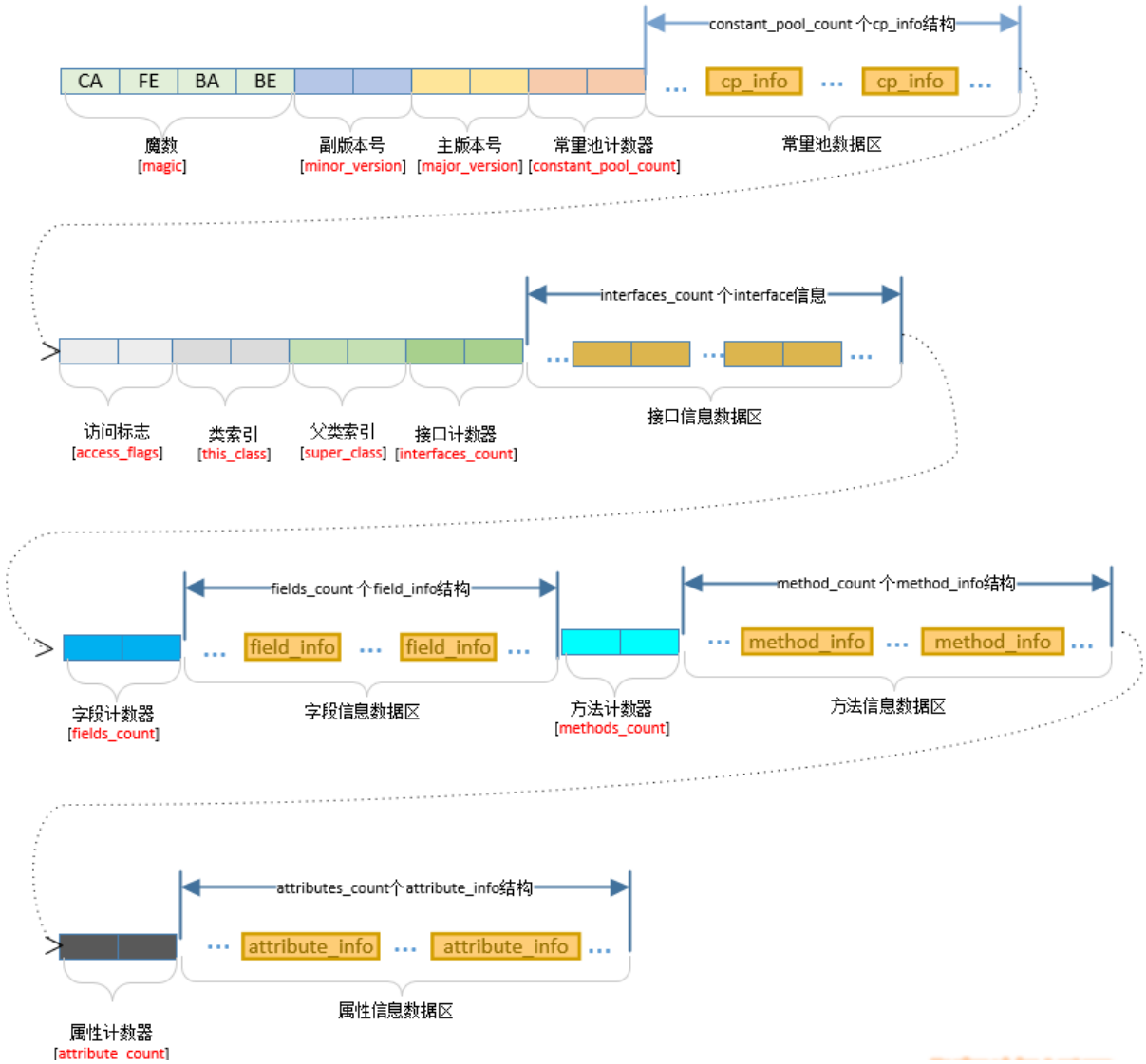
整理Java class的基本结构，整理Klass的相关数据结构及作用

1. class文件结构

根据 Java 虚拟机规范，一个Class文件由单个 ClassFile 结构组成

Class文件字节码结构组织示意图

注：被编译器编译成的.class字节码文件的字节流以及其组织结构如下如所示：




```
    ClassFile {
    u4    magic;     //Class 文件的标志，魔术
    u2    minor_version;    //Class 的附版本号
    u2    major_version;    //Class 的主版本号
    u2    constant_pool_count;    //常量池表项的数量
    cp_info    constant_pool[constant_pool_count-1];    //常量池表项，索引为
1~constant_pool_count-1
```

```
    u2    access_flags;    //Class 的访问标志（类访问修饰符）
    u2    this_class;    //表示当前类的引用
    u2    super_class;    //表示父类的引用
    u2    interfaces_count;    //实现接口数量
    u2    interfaces[interfaces_count];    //接口索引数组
    u2    fields_count;    //此类的字段表中的字段数量
    field_info    fields[fields_count];    //一个类会可以有多个字段，字段表
    u2    methods_count;    //此类的方法表中的方法数量
    method_info    methods[methods_count];    //一个类可以有个多个方法，方法表
    u2    attributes_count;    //此类的属性表中的属性数量
    attribute_info    attributes[attributes_count];    //属性表集合
}
```

1. 无符号数属于基本的数据类型，以 u1、u2、u4、u8来分别代表 1 个字节、2 个字节、4 个字节和 8个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者按照 UTF-8 编码结构构成的字符串值。对于字符串，则使用u1数组进行表示。
2. 表是由多个无符号数或者其他表作为数据项构成的复合数据类型，所有表都习惯性地以「_info」结尾。表用于描述有层次关系的复合结构的数据，整个 Class 文件就是一张ClassFile 表，它由下表中所示的数据项构成。

1.1. 魔数(magic)

所有的由Java编译器编译而成的class文件的前4个字节都是 `0xCAFEBABE` ,它的作用在于：当JVM在尝试加载某个文件到内存中来的时候，会首先判断此class文件有没有JVM认为可以接受的“签名”，即JVM会首先读取文件的前4个字节，判断该4个字节是否是 `0xCAFEBABE` ，如果是，则JVM会认为可以将此文件当作class文件来加载并使用。Java一直以咖啡为代言，CAFEBABE可以认为是 Cafe Babe，读音上和Cafe Baby很近。所以这个也许就是代表Cafe Baby的意思。

1.2. 版本号(minor_version,major_version)

```
u2    minor_version;    //Class 的附版本号
u2    major_version;    //Class 的主版本号
```

随着Java本身的发展，Java语言特性和JVM虚拟机也会有相应的更新和增强。目前我们能够用到的JDK版本如：1.5，1.6，1.7，还有现如今最新的1.8。发布新版本的目的在于：在原有的版本上增加新特性和相应的JVM虚拟机的优化。而随着主版本发布的次版本，则是修改相应主版本上出现的bug。我们平时只需要关注主版本就可以了。

主版本号和次版本号在class文件中各占两个字节，副版本号占用第5、6两个字节，而主版本号则占用第7，8两个字节。JDK1.0的主版本号为45，以后的每个新主版本都会在原先版本的基础上加1。若现在使用的是JDK1.7编译出来的class文件，则相应的主版本号应该是51,对应的7，8个字节的十六进制的值应该是 0x33。

一个 JVM实例只能支持特定范围内的主版本号 （Mi 至Mj） 和 0 至特定范围内 （0 至 m） 的副版本号。假设一个 Class 文件的格式版本号为 V， 仅当**Mi.0 ≤ v ≤ Mj.m**成立时，这个 Class 文件才可以被此 Java 虚拟机支持。不同版本的 Java 虚拟机实现支持的版本号也不同，高版本号的 Java 虚拟机实现可以支持低版本号的 Class 文件，反之则不成立。

JVM在加载class文件的时候，会读取出主版本号，然后比较这个class文件的主版本号和JVM本身的版本号，如果JVM本身的版本号 < class文件的版本号，JVM会认为加载不了这个class文件，会抛出我们经常见到的"java.lang.UnsupportedClassVersionError: Bad version number in .class file " **Error 错误**；反之，JVM会认为可以加载此class文件，继续加载此class文件。

JDK主版本号	Class主版本号	16进制
1.1	45.0	00 00 00 2D
1.2	46.0	00 00 00 2E
1.3	47.0	00 00 00 2F
1.4	48.0	00 00 00 30
1.5	49.0	00 00 00 31
1.6	50.0	00 00 00 32
1.7	51.0	00 00 00 33
1.8	52.0	00 00 00 34

1.3. 常量池计数器(constant_pool_count)

```
u2      constant_pool_count;      //常量池表项的数量
```

常量池是class文件中非常重要的结构，它描述着整个class文件的**字面量信息**。 常量池是由一组constant_pool结构体数组组成的，而数组的大小则由常量池计数器指定。常量池计数器constant_pool_count 的值=constant_pool表中的成员数+ 1。constant_pool表的索引值只有在大于 0 且小于constant_pool_count时才会被认为是有效的。

1.4. 常量池数据区(constant_pool[contstant_pool_count-1])

```
cp_info {
    u1      tag;
    info      info[];
}
```

常量池，constant_pool是一种表结构,它包含 Class 文件结构及其子结构中引用的所有字符串常量、 类或接口名、字段名和其它常量。 常量池中的每一项都具备相同的格式特征——第一个字节作为类型标记用于识别该项是哪种类型的常量，称为 “tag byte” 。常量池的索引范围是 1 至constant_pool_count−1。常量池的具体细节我们会稍后讨论。

在JDK1.8中有14种常量池项目类型，每一种项目都有特定的表结构，这14种表有一个共同的特点：开始的第一位是一个 u1 类型的标志位 -tag 来标识常量的类型，代表当前这个常量属于哪种常量类型。**常量池tag类型表：**

常量类型	标志（tag）	描述
CONSTANT_utf8_info	1	UTF-8编码的字符串
CONSTANT_Integer_info	3	整形字面量
CONSTANT_Float_info	4	浮点型字面量
CONSTANT_Long_info	5	长整型字面量
CONSTANT_Double_info	6	双精度浮点型字面量
CONSTANT_Class_info	7	类或接口的符号引用
CONSTANT_String_info	8	字符串类型字面量
CONSTANT_Fieldref_info	9	字段的符号引用
CONSTANT_Methodref_info	10	类中方法的符号引用
CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
CONSTANT_NameAndType_info	12	字段或方法的符号引用
CONSTANT_MothodType_info	16	标志方法类型
CONSTANT_MethodHandle_info	15	表示方法句柄
CONSTANT_InvokeDynamic_info	18	表示一个动态方法调用点

每种常量类型均有自己的表结构，非常繁琐：

常量池项目结构表：

常量	描述	项目	类型	项目描述
CONSTANT_Utf8_info	UTF-8编码的字符串	tag	u1	值为1
		length	u2	UTF-8编码的字符串占用的字节数
		bytes[length]	u1	长度为length的UTF-8编码的字符串
CONSTANT_Integer_info	整型字面量	tag	u1	值为3
		bytes	u4	按照高位在前存储的int值
CONSTANT_Float_info	浮点型字面量	tag	u1	值为4
		bytes	u4	按照高位在前存储的float值
CONSTANT_Long_info	长整型字面量	tag	u1	值为5
		bytes	u8	按照高位在前存储的long值
CONSTANT_Double_info	双精度浮点型字面量	tag	u1	值为6
		bytes	u8	按照高位在前存储的double值
CONSTANT_Class_info	类或接口的符号引用	tag	u1	值为7
		name_index	u2	指向全限定名常量项的索引
CONSTANT_String_info	字符串类型字面量	tag	u1	值为8
		string_index	u2	指向字符串字面量的索引
CONSTANT_Fieldref_info	字段的符号引用	tag	u1	值为9
		class_index	u2	指向声明字段的类或者接口描述符CONSTANT_Class_info的索引项
		name_and_type_index	u2	指向字段描述符CONSTANT_NameAndType的索引项
CONSTANT_Methodref_info	类中方法的符号引用	tag	u1	值为10
		class_index	u2	指向声明方法的类描述符CONSTANT_Class_info的索引项
		name_and_type_index	u2	指向名称及类型描述符CONSTANT_NameAndType的索引项
CONSTANT_InterfaceMethodref_info	接口中方法的符号引用	tag	u1	值为11
		class_index	u2	指向声明方法的接口描述符CONSTANT_Class_info的索引项
		name_and_type_index	u2	指向名称及类型描述符CONSTANT_NameAndType的索引项
CONSTANT_NameAndType_info	字段或方法的部分符号引用	tag	u1	值为12
		name_index	u2	指向该字段或方法名称常量项的索引
		descriptor_index	u2	指向该字段或方法描述符常量项的索引
CONSTANT_MethodHandle_info	表示方法句柄	tag	u1	值为15
		reference_kind	u1	值必须在1~9范围，它决定了方法句柄的类型。方法句柄类型的值表示方法句柄的字节码行为
		reference_index	u2	值必须是对常量池的有效索引
CONSTANT_InvokeDynamic_info	表示一个动态方法调用点	tag	u1	值为18
		bootstrap_method_attr_index	u2	值必须是对当前Class文件中引导方法表的bootstrap_methods[]数组的的有效索引
		name_and_type_index	u2	值必须是对当前常量池的有效索引，常量池在该索引处的项必须是CONSTANT_NameAndType_info结构，表示方法名和方法描述符
CONSTANT_MethodType_info	标识方法类型	tag	u1	值为16
		descriptor_index	u2	值必须是对常量池的有效索引，常量池在该索引处的项必须是CONSTANT_Utf8_info结构，表示方法的描述符

1.5. 访问标志(access_flags)

访问标志，access_flags 是一种掩码标志，用于表示某个类或者接口的访问权限及基础属性。

表示名称	值	含义
ACC_PUBLIC	0x0001	是否为public类型
ACC_FINAL	0x0010	是否被声明为final类型，只有类可设置
ACC_SUPER	0x0020	是否允许使用invokespecial字节码指令的新语意，invokespecial指令的语意在JDB1.2之后发生过改变，为了区别这条指令使用哪种语意，JDK1.0.2之后编译出来的类的这个标志都必须为真。
ACC_INTERFACE	0x0200	标识这是一个接口
ACC_ABSTRACT	0x0400	是否为abstract类型，对于接口或者抽象类来说，此标志为真，其它类值为假
ACC_SYNTHETIC	0x1000	标识这个类并非由用户代码产生的
ACC_ANNOTATION	0x2000	标识这是一个注解
ACC_ENUM	0x4000	标识这是一个枚举

标记名	值	含义
ACC_PUBLIC	0x0001	可以被包的类外访问。
ACC_FINAL	0x0010	不允许有子类。
ACC_SUPER	0x0020	当用到 invokespecial 指令时，需要特殊处理的父类方法。
ACC_INTERFACE	0x0200	标识定义的是接口而不是类。
ACC_ABSTRACT	0x0400	不能被实例化。
ACC_SYNTHETIC	0x1000	标识并非 Java 源码生成的代码。
ACC_ANNOTATION	0x2000	标识注解类型
ACC_ENUM	0x4000	标识枚举类型

注：

- 带有 ACC_SYNTHETIC 标志的类，意味着它是由编译器自己产生的而不是由程序员编写的源代码生成的。
- 带有 ACC_ENUM 标志的类，意味着它或它的父类被声明为枚举类型。
- 带有 ACC_INTERFACE 标志的类，意味着它是接口而不是类，反之是类而不是接口。如果一个 Class 文件被设置了 ACC_INTERFACE 标志，那么同时也得设置 ACC_ABSTRACT 标志。同时它不能再设置 ACC_FINAL、ACC_SUPER 和 ACC_ENUM 标志。
- 注解类型必定带有 ACC_ANNOTATION 标记，如果设置了 ANNOTATION 标记，ACC_INTERFACE 也必须被同时设置。如果没有同时设置 ACC_INTERFACE 标记，那么这个 Class 文件可以具有表中的除 ACC_ANNOTATION 外的所有其它标记。当然 ACC_FINAL 和 ACC_ABSTRACT 这类互斥的标记除外。
- ACC_SUPER 标志用于确定该 Class 文件里面的 invokespecial 指令使用的是哪一种执行语义。目前 Java 虚拟机的编译器都应当设置这个标志。ACC_SUPER 标记是为了向后兼容旧编译器编译的 Class 文件而存在的，在 JDK1.0.2 版本以前的编译器产生的 Class 文件中，access_flag 里面没有 ACC_SUPER 标志。同时，JDK1.0.2 前的 Java 虚拟机遇到 ACC_SUPER 标记会自动忽略它。
- 在表中没有使用的 access_flags 标志位是为未来扩充而预留的 这些预留的标志为在编译器中会被设置为 0，Java 虚拟机实现也会自动忽略它们.

1.6. 类索引(this_class)

```
u2      this_class;      //表示当前类的引用
u2      super_class;     //表示父类的引用
u2      interfaces_count;      //接口数量
u2      interfaces[interfaces_count];      //接口索引集合
```

类索引，this_class的值必须是对constant_pool表中项目的一个有效索引值。constant_pool表在这个索引处的项必须为CONSTANT_Class_info 类型常量，表示这个 Class 文件所定义的类或接口。

1.7. 父类索引(super_class)

父类索引，对于类来说，super_class 的值必须为 0 或者是对constant_pool 表中项目的一个有效索引值。如果它的值不为 0，那 constant_pool 表在这个索引处的项必须为CONSTANT_Class_info 类型常量，表示这个 Class 文件所定义的类的直接父类。当前类的直接父类，以及它所有间接父类的access_flag 中都不能带有 ACC_FINAL 标记。对于接口来说，它的Class文件的super_class项的值必须是对constant_pool表中项目的一个有效索引值。constant_pool表在这个索引处的项必须为代表 java.lang.Object 的 CONSTANT_Class_info 类型常量。如果 Class 文件的 super_class的值为 0，那这个Class文件只可能是定义的是java.lang.Object类，只有它是唯一没有父类的类。

1.8. 接口计数器(interfaces_count)

接口计数器，interfaces_count的值表示当前类或接口的直接父接口数量。(如果为零，后面的接口信息数据区则无)

1.9. 接口信息数据区(interfaces[interfaces_count])

接口表，interfaces[]数组中的每个成员的值必须是一个对constant_pool表中项目的一个有效索引值， 它的长度为 interfaces_count。每个成员 interfaces[i] 必须为 CONSTANT_Class_info类型常量，其中 **0 ≤ i <interfaces_count**。在interfaces[]数组中，成员所表示的接口顺序和对应的源代码中给定的接口顺序（从左至右）一样，即interfaces[0]对应的是源代码中最左边的接口。

1.10. 字段计数器(fields_count)

```
u2      fields_count;      //此类的字段表中的字段数量
field_info      fields[fields_count];      //一个类会可以有多个字段，字段表
```

字段计数器，fields_count的值表示当前 Class 文件 fields[]数组的成员个数。 fields[]数组中每一项都是一个field_info结构的数据项，它用于表示该类或接口声明的类字段或者实例字段。字段包括类级变量以及实例变量，但不包括在方法内部声明的局部变量。不会列出从父类或者父接口继承来的字段，但有可能列出原本Java代码没有的字段，比如在内部类中为了保持对外部类的访问性，会自动添加指向外部类实例的字段。

1.11. 字段信息数据区(fields[fields_count])

```
field_info {
    u2      access_flags;
    u2      name_index;
    u2      descriptor_index;
    u2      attributes_count;
    attribute_info      attributes[attributes_count];
}
```

- access_flags: 字段的作用域（public,private,protected修饰符），是实例变量还是类变量（static修饰符），可否被序列化（transient修饰符）,可变性（final）,可见性（volatile 修饰符，是否强制从主内存读写）。只有一个
- name_index:对常量池的引用，CONSTANT_utf8_info，表示的字段的名称；只有一个
- descriptor_index: 对常量池的引用，CONSTANT_utf8_info，表示字段和方法的描述符；只有一个
- attributes_count: 一个字段还会拥有一些额外的属性，表示attributes_count 存放属性的个数；只有一个
- attributes[attributes_count]: 存放具体属性具体内容集合。有attributes_count个

字段表，fields[]数组中的每个成员都必须是一个fields_info结构的数据项，用于表示当前类或接口中某个字段的完整描述。 fields[]数组描述当前类或接口声明的所有字段，但不包括从父类或父接口继承的部分。

在Java语言中字段是无法重载的，两个字段的数据类型、修饰符不管是否相同，都必须使用不一样的名称，但是对于字节码来讲，如果两个字段的描述符不一致，那字段重名就是合法的。

额外的属性表示一个字段还可能拥有一些属性， 用于存储更多的额外信息， 比如初始化值、一些注释信息等。属性个数存放在attributes_count 中， 属性具体内容存放于attributes数组。常量数据比如“public static finall”类型的字段，就有一个ConstantValue的额外属性。

字段访问和属性标识表：

标志名称	标志值	含义
ACC_PUBLIC	0x0001	字段为public
ACC_PRIVATE	0x0002	字段为private
ACC_PROTECTED	0x0004	字段为protected
ACC_STATIC	0x0008	字段为static
ACC_FINAL	0x0010	字段为final
ACC_VOLATILE	0x0040	字段为volatile
ACC_TRANSIENT	0x0080	字段为transient
ACC_SYNTHETIC	0x1000	字段由编译器自动产生的
ACC_ENUM	0x4000	字段是enum

attribute属性结构（在属性表部分会深入讲解）：

> 以常量属性为例，常量属性的结构为：
>
> ```
> ConstantValue_attribute {
>     u2 attribute_name_index;
>     u4 attribute_length;
>     u2 constantvalue_index;
> }
> ```

- attribute_name_index 为2 字节整数， 指向常量池的CONSTANT_Utf8_info类型, 并且这个字符串为“ ConstantValue" 。该属性是所有类型的字段都有的。
- attribute_length由4个字节组成， 表示这个属性的剩余长度为多少。对常量属性而言， 这个值恒定为2，表示从Ox00000002 之后的两个字节为属性的全部内容。该属性是所有类型的字段都有的。
- constantvalue_ index 表示属性值，但值并不直接出现在属性中，而是指向常量池引用，即存放在常量池中，不同类型的属性会和不同的常量池类型匹配。

1.12. 方法计数器(methods_count)

```
u2      methods_count;      //此类的方法表中的方法数量
method_info      methods[methods_count];      //一个类可以有个多个方法，方法表
```

方法计数器， methods_count的值表示当前Class 文件 methods[]数组的成员个数。Methods[]数组中每一项都是一个 method_info 结构的数据项。

1.13. 方法信息数据区(methods[methods_count])

```
method_info {
    u2        access_flags;
    u2        name_index;
    u2        descriptor_index;
    u2        attributes_count;
```

```
    attribute_info      attributes[attributes_count];
}
```

- name_index 表示方法的名称， 它是一个指向常量池的索引。
- descriptor_ index为方法描述符， 它也是指向常量池的索引， 是一个字符串， 用以表示方法的签名（参数、返回值等）。关于方法描述符，在最开始已经介绍了。
- 和字段表类似， 方法也可以附带若干个属性， 用于描述一些额外信息， 比如方法字节码等，attributes_count 表示该方法中属性的数量， 紧接着， 就是attributes_count个属性的描述。

attribute_info通用格式为：

```
attribute_info  {
    u2      attribute_name_index;
    u4      attribute_length;
    u1      info[attribute_length];
}
```

其中， attribute_name_ index 表示当前attribute的名称， attribute_length为当前attribute的剩余长度， 紧接着就是attribute_length个字节的byte 数组。常见的 attribute在属性表会有详细介绍。

方法表的结构如同字段表一样，依次包括了**访问标记、名称索引、描述符索引、属性表集合几项**。而方法中的具体代码则是存放在属性表中了。类似于字段表，子类不会记录父类未重写的方法，同上编译器可能自己加方法，比如< init > 、< clinit >。

方法表，methods[] 数组中的每个成员都必须是一个 method_info 结构的数据项，用于表示当前类或接口中某个方法的完整描述。如果某个method_info 结构的access_flags 项既没有设置 ACC_NATIVE 标志也没有设置 ACC_ABSTRACT 标志，那么它所对应的方法体就应当可以被 Java 虚拟机直接从当前类加载，而不需要引用其它类。 method_info结构可以表示类和接口中定义的所有方法，包括实例方法、类方法、实例初始化方法方法和类或接口初始化方法方法 。methods[]数组只描述当前类或接口中声明的方法，不包括从父类或父接口继承的方法。

1.14. 属性计数器(attributes_count)

```
u2      attributes_count;       //此类的属性表中的属性数量
attribute_info      attributes[attributes_count];       //属性表集合
```

属性计数器，attributes_count的值表示当前 Class 文件attributes表的成员个数。attributes表中每一项都是一个attribute_info 结构的数据项。

1.15. 属性信息数据区(attributes[attributes_count])

属性表，attributes 表的每个项的值必须是attribute_info结构。

属性表用于class文件格式中的ClassFile，field_info，method_info和Code_attribute结构，以用于描述某些场景专有的信息。与 Class 文件中其它的数据项目要求的顺序、长度和内容不同，属性表集合的限制稍微宽松一些，不再要求各个属性表具有严格的顺序，并且只要不与已有的属性名重复，任何人实现的编译器都可以向属性表中写 入自己定义的属性信息，Java 虚拟机运行时会忽略掉它不认识的属性。

对于每个属性，它的名称需要从常量池中引用一个CONSTANT_Utf8_info类型的常量来表示，而属性值的结构则是完全自定义的，只需要通过一个u4的长度去说明属性值所占用的位数即可。

属性表通用结构

```
attribute_info  {
    u2      attribute_name_index;
    u4      attribute_length;
    u1      info[attribute_length];
}
```

> 对于任意属性， attribute_name_index必须是对当前class文件的常量池的有效16位无符号索引。 常量池在该索引处的成员必须是CONSTANT_Utf8_info 结构， 用以表示当前属性的名字。 attribute_length项的值给出了跟随其后的信息字节的 长度， 这个长度不包括attribute_name_index 和 attribute_length项的6个字节。

1.16. 预定义属性

《java虚拟机规范 JavaSE8》中预定义23项虚拟机实现应当能识别的属性:

属性	可用位置	含义
SourceFile	ClassFile	记录源文件名称
InnerClasses	ClassFile	内部类列表
EnclosingMethod	ClassFile	仅当一个类为局部类或者匿名类时，才能拥有这个属性，这个属性用于表示这个类所在的外围方法
SourceDebugExtension	ClassFile	JDK1.6中新增的属性，SourceDebugExtension用于存储额外的调试信息。如在进行JSP文件调试时，无法通过Java堆栈来定位到JSP文件的行号，JSR-45规范为这些非Java语言编写，却需要编译成字节码运行在Java虚拟机汇中的程序提供了一个进行调试的标准机制，使用SourceDebugExtension就可以存储这些调试信息。
BootstrapMethods	ClassFile	JDK1.7新增的属性，用于保存invokedynamic指令引用的引导方法限定符
ConstantValue	field_info	final关键字定义的常量值
Code	method_info	Java代码编译成的字节码指令(即：具体的方法逻辑字节码指令)
Exceptions	method_info	方法声明的异常
RuntimeVisibleAnnotations	ClassFile, field_info, method_info	JDK1.5中新增的属性，为动态注解提供支持RuntimeVisibleAnnotations属性，用于指明哪些注解是运行时(实际上运行时就是进行反射调用)可见的。
RuntimeInvisibleAnnotations	ClassFile, field_info, method_info	JDK1.5中新增的属性，作用与RuntimeVisibleAnnotations相反用于指明哪些注解是运行时不可见的。
RuntimeVisibleParameterAnnotations	method_info	JDK1.5中新增的属性，作用与RuntimeVisibleAnnotations类似，只不过作用对象为方法的参数。
RuntimeInvisibleParameterAnnotations	method_info	JDK1.5中新增的属性，作用与RuntimeInvisibleAnnotations类似，只不过作用对象为方法的参数。
AnnotationDefault	method_info	JDK1.5中新增的属性，用于记录注解类元素的默认值
MethodParameters	method_info	52.0
Synthetic	ClassFile, field_info, method_info	标识方法或字段为编译器自动产生的
Deprecated	ClassFile, field_info, method_info	被声明为deprecated的方法和字段
Signature	ClassFile, field_info, method_info	JDK1.5新增的属性，这个属性用于支持泛型情况下的方法签名，在Java语言中，任何类、接口、初始化方法或成员的泛型签名如果包含了类型变量(Type Variables)或参数类型(Parameterized Types),则Signature属性会为它记录泛型签名信息。由于Java的泛型采用擦除法实现，在为了避免类型信息被擦除后导致签名混乱，需要这个属性记录泛型中的相关信息
RuntimeVisibleAnnotations	ClassFile, field_info, method_info	JDK1.5中新增的属性，为动态注解提供支持。RuntimeVisibleAnnotations属性，用于指明哪些注解是运行时(实际上运行时就是进行反射调用)可见的。

属性	可用位置	含义
RuntimeInvisibleAnnotations	ClassFile, field_info, method_info	JDK1.5中新增的属性，作用与RuntimeVisibleAnnotations相反用于指明哪些注解是运行时不可见的。
LineNumberTable	Code	Java源码的行号与字节码指令的对应关系
LocalVariableTable	Code	方法的局部变量描述
LocalVariableTypeTable	Code	JDK1.5中新增的属性，它使用特征签名代替描述符，是为了引入泛型语法之后能描述泛型参数化类型而添加
StackMapTable	Code	JDK1.6中新增的属性，供新的类型检查验证器(Type Checker)检查和处理目标方法的局部变量和操作数栈所需要的类型是否匹配
MethodParameters	method_info	JDK1.8中新加的属性，用于标识方法参数的名称和访问标志
RuntimeVisibleTypeAnnotations	ClassFile, field_info, method_info, Code	JDK1.8中新加的属性，在运行时可见的注释，用于泛型类型，指令等。
RuntimeInvisibleTypeAnnotations	ClassFile, field_info, method_info, Code	JDK1.8中新加的属性，在编译时可见的注释，用于泛型类型，指令等。

下面是一些常见的属性。

1.17. Code属性——方法体

Java方法体里面的代码经过Javac编译之后，最终变为字节码指令存储在Code属性内，Code属性出现在在method_info结构的attributes表中，但在接口或抽象类中就不存在Code属性（JDK1.8可以出现了）。一个方法中的Code属性值有一个。

在整个Class文件中，Code属性用于描述代码，所有的其他数据项目都用于描述元数据。

在字节码指令之后的是这个方法的显式异常处理表（下文简称异常表）集合，异常表对于Code属性来说并不是必须存在的。

Code属性的结构如下：

```
Code_attribute {
    u2      attribute_name_index;
    u4      attribute_length;
    u2      max_stack;
    u2      max_locals;
    u4      code_length;
    u1      code[code_length];
    u2      exception_table_length;
    {   u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2      attributes_count;
    attribute_info      attributes[attributes_count];
}
```

- Code属性的第一个字段attribute_name_index指定了该属性的名称，它是一个指向常量池的索引， 指向的类型为CONSTANT_Utf8_info, 对于Code 属性来说，该值恒为“Code"。
- attribute_length指定了Code属性的长度，该长度不包括前6个字节，即表示剩余长度。
- 在方法执行过程中，操作数栈可能不停地变化，在整个执行过程中，操作数栈存在一个最大深度，该深度由max_stack表示。
- 在方法执行过程中，局部变量表也可能会不断变化。在整个执行过程中局部变量表的最值由max_locals表示，它们都是2字节的无符号整数。也包括调用此方法时用于传递参数的局部变量。
- 在max_locals 之后，就是作为方法的最重要部分—字节码。它由code_length和code[code_length]两部分组成， code_length 表示字节码的字节数，为4字节无符号整数，必须大于0；code[code length]为byte数组，即存放的代码的实际字节内容本身。
- 在字节码之后，存放该方法的异常处理表。异常处理表告诉一个方法该如何处理字节码中可能抛出的异常。异常处理表亦由两部分组成：表项数量和内容。在Java虚拟机中，**处理异常（catch语句）不是由字节码指令来实现的，而是采用异常表来完成的。**
- exception_table_length表示异常表的表项数量，可以为0；
- exception_table[exception_table_length]结构为异常表的数据。
- 异常表中每一个数据由4部分组成，分别是start_pc、end_pc、handler_pc和catch_type。这4项表示从方法字节码的start_pc偏移量开始（包括）到end_pc 偏移量为止（不包括）的这段代码中，如果遇到了catch_type所指定的异常， 那么代码就跳转到handler_pc的位置执行，handler_pc即一个异常处理器的起点。
- 在这4项中， start_pc、end_pc和handlerpc 都是字节码的编译量， 也就是在code[code_length]中的位置， 而catch_type为指向常量池的索引，它指向一个CONSTANT_Class_info 类，表示需要处理的异常类型。如果catch_type值为0,那么将会在所有异常抛出时都调用这个异常处理器，这被用于实现finally语句。
- 至此， Code属性的主体部分已经介绍完毕， 但是Code属性中还可能包含更多信息， 比如行号表、局部变量表等。这些信息都以attribute属性的形式内嵌在Code属性中， 即除了字段、方法和类文件可以内嵌属性外，属性本身也可以内嵌其他属性。attributes_count表示Code属性的属性数量,attributes表示Code属性包含的属性。

1.18. exception_table---异常表

- 异常表实际上是Java代码的一部分， 编译器使用异常表而不是简单的跳转命令来实现Java异常及finally处理机制。（注：在JDK1.4.2之前的Javac编译器采用了jsr和ret指令实现finally语句。在JDK1.7中，已经完全禁止Class文件中出现jsr和ret指令，如果遇到这两条指令，虚拟机会在类加载的字节码校验阶段抛出异常）
- 当异常处理存在finally语句块时，编译器会自动在每一段可能的分支路径之后都将finally语句块的内容冗余生成一遍来实现finally语义。
- 在我们Java代码中，finally语句块是在最后的，但编译器在生成字节码时候，其实将finally语句块的执行指令移到了return指令之前，指令重排序了。所以，从字节码层面，我们解释了，为什么finally语句总会执行！初学Java的时候，我们有感到困惑，为什么方法已经return了，finally语句块里的代码还会执行呢？这是因为，在字节码中，它就是先执行了finally语句块，再执行return的，而这个变化是Java编译器帮我们做的。

1.19. LineNumberTable属性——记录行号

- 位于Code属性中，描述Java源码行号与字节码行号（字节码的偏移量）之间的对应关系。主要是如果抛出异常时，编译器会显示行号，比如调试程序时展示的行号，就是这个属性的作用。
- Code属性表中，LineNumberTable可以属性可以按照任意顺序出现。
- 在Code属性 attributes表中，可以有不止一个LineNumberTable属性对应于源文件中的同一行。也就是说，多个LineNumberTable属性可以合起来表示源文件中的某行代码，属性与源文件的代码行之间不必有一一对应的关系。

  LineNumberTable属性的格式如下：

```
LineNumberTable_ attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 line_number_table_length;
    {    u2    start_pc;
         u2    line_number;
    } line_number_table[line_number_table_length];
}
```

其中， attribute_name_index为指向常量池的索引， 在LineNumberTable 属性中， 该值为"LineNumberTable", attribute_length为4 字节无符号整数， 表示属性的长度（不含前6个字节），line_number_table_length 表明了表项有多少条记录。

line_number_table为表的实际内容，它包含line_number_table_length 个<start_pc, line_number>元组，其中，start_pc为字节码偏移量，必须是code数组的一个索引， line_number 为对应的源码的行号。

1.20. LocalVariableTable属性——保存局部变量和参数

- 描述栈帧中局部变量表中的变量与Java源码中定义的变量之间的关系。用处在于当别人使用这个方法是能够显示出方法定义的参数名。
- 它也不是运行时必需的属性，但默认会生成到Class文件之中。如果没有生成这项属性，最大的影响就是当其他人引用这个方法时，所有的参数名称都将会丢失

LocalVariableTable属性结构如下：

```
LocalVariableTable_attribute {
    u2      attribute_name_index;
    u4      attribute_length;
    u2      local_variable_table_length;
    {       u2      start_pc;
            u2      length;
            u2      name_index;
            u2      descriptor_index;
            u2      index;
    } local_variable_table[local_variable_table_length];
}
```

其中， attribute_name_index为当前属性的名字， 它是指向常量池的索引。对局部变量表而言， 该值为“LocalVariableTable", attribute_length 为属性的长度，local_variable_table_length为局部变量表表项条目。
局部变量表的每一条记录由以下几个部分组成：

1. start_pc、length：start_pc表示当前局部变量的开始位置，从0开始，length表示范围覆盖长度，两者结合就是这个局部变量在字节码中的作用域范围。
2. name_ index: 局部变量的名称， 这是一个指向常量池的索引，为CONSTANT_Utf8_info类型。
3. descriptor_index: 局部变量的类型描述， 指向常量池的索引。使用和字段描述符一样的方式描述局部变量，为CONSTANT_Utf8_info类型。
4. index，局部变量在当前帧栈的局部变量表中的槽位（solt）。当变量数据类型是64位时（long 和double）， 它们会占据局部变量表中的两个槽位，位置为index和index+1。

在JDK 1.5引入泛型后，LocalVariableTable属性增加了一个“姐妹属性”：**LocalVariableTypeTable**，这个新增的属性结构与LocalVariableTable非常相似，仅仅是把记录的字段描述符的descriptor_index替换成了字段的特征签名（Signature），对于非泛型类型来说，描述符和特征签名能描述的信息是基本一致的，但是泛型引入之后，由于描述符中泛型的参数化类型被擦出掉，描述符就不能准确地描述泛型类型了，因此出现了**LocalVariableTypeTable。**

1.21. StackMapTable属性——加快字节码校验

JDK 1.6 以后的类文件， 每个方法的Code 属性还可能含有一个StackMapTable 的属性结构。这是一个复杂的变长属性，位于Code属性的属性表中。这个属性会在虚拟机类加载的字节码验证阶段被新类型检查验证器（Type Checker）使用，目的在于代替以前比较消耗性能的基于数据流分析的类型推导验证器，加快字节码校验。

StackMapTable属性中包含零至多个栈映射帧（Stack Map Frames），每个栈映射帧都显式或隐式地代表了一个字节码偏移量，用于表示该执行到该字节码时局部变量表和操作数栈的验证类型。

该属性不包含运行时所需的信息，仅仅作为Class文件的类型检验。

StackMapTable属性结构如下：

```
StackMapTable_attribute {
    u2      attribute_name_index;
    u4      attribute_length;
    u2      number_of_entries;
    stack_map_frame entries[number_of_entries];
}
```

其中，attribute_name——index为常量池索引， 恒为“ Stack.MapTable"， attribute_length为该属性的长度，number_of_entries为栈映射帧的数量， 最后的stack_map_frame entries则为具体的内容，每一项为一个stack_map_ frame结构。

1.22. Exceptions属性——列举异常

列举出方法中可能抛出的受查异常，也就是方法描述时在throws关键字后面列举的异常。Exceptions与Code 属性中的异常表不同。Exceptions 属性表示一个方法可能抛出的异常，通常是由方法的throws 关键字指定的。而Code 属性中的异常表，则是异常处理机制，由try-catch语句生成。

Exceptions属性结构如下：

```
Exceptions_attribute {
    u2      attribute_name_index;
    u4      attribute_length;
    u2      number_of_exceptions;
    u2      exception_index_table[number_of_exceptions];
}
```

Exceptions 属性表中， attribute_name_index 指定了属性的名称， 它为指向常量池的索引，恒为“Exceptions"，attribute_lengt表示属性长度， number_of_exceptions表示表项数量即可能抛出的异常个数，最后exception_index_table项罗列了所有的异常，每一项为指向常量池的索引，对应的常量为CONSTANT_Class_info，表示为一个异常类型。

1.23. SourseFile属性——记录来源

SourseFile属性ClassFile，记录生成这个Class文件的源码文件名称，抛出异常时能够显示错误代码所属的文件名。

SourseFile属性结构如下：

```
SourceFile_attribute {
    u2      attribute_name_index;
    u4      attribute_length;
    u2      sourcefile_index;
}
```

SourseFile属性表中， attribute_name_index 指定了属性的名称， 它为指向常量池的索引，恒为“SourseFile",attribute_length表示属性长度，对SourseFile属性来说恒为2，sourcefile_index表示源代码文件名， 它是为指向常量池的索引，对应的常量为CONSTANT_Uft8_info类型。

1.24. InnerClass属性——记录内部类

InnerClass属性属性ClassFile，用于记录内部类与宿主类之间的关联。

InnerClass属性结构如下：

```
InnerClasses_attribute {
    u2      attribute_name_index;
    u4      attribute_length;
    u2      number_of_classes;
```

```
    {       u2      inner_class_info_index;
            u2      outer_class_info_index;
            u2      inner_name_index;
            u2      inner_class_access_flags;
    } classes[number_of_classes];
}
```

其中， attribute_name_index表示属性名称，为指向常量池的索引，这里恒为“TnnerClasses ”。attribute_ length为属性长度，number_of_classes 表示内部类的个数。classes[number_ of_ classes]为描述内部类的表格，每一条内部类记录包含4个字段，其中，inner_class_ info_ index为指向常量池的指针， 它指向一个CONSTANT_ Class_info, 表示内部类的类型。outer_class_ info_index表示外部类类型，也是常量池的索引。inner_name_ index表示内部类的名称， 指向常量池中的CONSTANT_ Utf8_info项。最后的inner_class_access_flags为内部类的访问标识符， 用于指示static、public等属性。

内部内标识符表如下：

标记名	值	含义
ACC_PUBLIC	0x0001	public
ACC_PRIVATE	0x0002	私有类
ACC_PROTECTED	0x0004	受保护的类
ACC_STATIC	0x0008	静态内部类
ACC_FINAL	0x0010	fmal类
ACC_INTERFACE	0x0200	接口
ACC_ABSTRACT	0x0400	抽象类
ACC_SYNTHETIC	0x1000	编译器产生的，而非代码产生的类
ACC_ANNOTATION	0x2000	注释
ACC_ENUM	0x4000	枚举

1.24.0.1. ConstantValue属性——static(常量)标识

onstantValue属性位于filed_info属性中，通知虚拟机自动为静态变量赋值，只有被static字符修饰的变量（类变量）才可以有这项属性。如果非静态字段拥有了这一属性，也该属性会被虚拟机所忽略。

对于非static类型的变量（也就是实例变量）的赋值是在实例构造器 < init > 方法中进行的；而对于类变量，则有两种方式可以选择：在类构造器 < clinit > 方法中或者使用ConstantValue属性。但是不同虚拟机有不同的实现。

目前Sun Javac编译器的选择是：如果同时使用final和static来修饰一个变量（按照习惯，这里称“常量”更贴切），并且这个变量的数据类型是基本类型或者java.lang.String的话，就生成ConstantValue属性来进行初始化，即编译的时候；如果这个变量没有被final修饰，或者并非基本类型及字符串，则将会选择在 < clinit > 方法中进行初始化，即类加载的时候。

ConstantValue属性结构如下：

```
ConstantValue_attribute {
    u2      attribute_name_index;
    u4      attribute_length;
    u2      constantvalue_index;
}
```

其中， attribute_name_index表示属性名称，为指向常量池的索引，这里恒为“ConstantValue_attribute ”。attribute_length表示后面还有几个字节，这里固定为2。constantvalue_index代表了常量池中一个字面常量的引用，根据字段类型不同，字面量可以是CONSTANT_Long_info、 CONSTANT_Float_info、CONSTANT_Double_info、CONSTANT_Integer_info、CONSTANT_String_info常量中的一种。

1.25. Signature属性

一个可选的定长属性，可以出现于ClassFile, field_info, method_info结构的属性表中。

任何类、接口、初始化方法或成员的泛型签名如果包含了类型变量（Type Variables）或参数化类型（Parameterized Types），则Signature属性会为它记录泛型签名信息。

Signature属性结构如下：

```
Signature_attribute {
    u2      attribute_name_index; //指向常量池的名称引用，固定为“Signature”
    u4      attribute_length; // 固定为2
    u2      signature_index; //指向常量池的类签名、方法类型前面、字段类型签名引用
}
```

1.25.0.1. Synthetic属性——编译器添加

标志类型的布尔属性，Synthetic属性用在ClassFile, field_info, method_info中，表示此字段或方法或类是由编译器自行添加的。

对于其他属性，可以查看官方文档：The Java® Virtual Machine Specification——Java SE 8 Edition

1.26. Deprecated属性——废弃

2. 描述符

标志类型的布尔属性，Deprecated表示类、方法、字段不再推荐使用，使用注解表示为@deprecated

首先，我们来说明一下：class文件只有两种数据类型： `无符号数` 和 `表` 。如下表所示：

数据类型	定义	说明
无符号数	无符号数可以用来描述数字、索引引用、数量值或按照utf-8编码构成的字符串值。	其中无符号数属于基本的数据类型。 以u1、u2、u4、u8来分别代表1个字节、2个字节、4个字节和8个字节
表	表是由多个无符号数或其他表构成的复合数据结构。	所有的表都以“_info”结尾。 由于表没有固定长度，所以通常会在其前面加上个数说明。

全限定名和非限定名

- Class文件中的类和接口，都是使用全限定名，又被称作Class的二进制名称。例如“com/ikang/JVM/classfile”是这个类的全限定名，仅仅是把类全名中的“.”替换成了“/”而已，为了使连续的多个全限定名之间不产生混淆，在使用时最后一般会加入一个“;”表示全限定名结束。
- 非限定名又被称作简单名称，Class文件中的方法、字段、局部变量、形参名称，都是使用简单名称，没有类型和参数修饰，例如这个类中的getK()方法和k字段的简单名称分别是“getK”和“m”。
- 非限定名不得包含ASCII字符. ; [/ ，此外方法名称除了特殊方法名称< init >和< clinit >方法之外，它们不能包含ASCII字符<或>，字段名称或接口方法名称可以是< init >或< clinit >，但是没有方法调用指令可以引用< clinit >，只有invokespecial指令可以引用< init >。
- 描述符的作用是用来描述字段的数据类型、方法的参数列表（包括数量、类型以及顺序）和返回值类型。

常量	具体的常量
字面量	文本字符串
	声明为final的常量值
符号引用	类和接口的全限定名
	字段的名称和描述符
	方法的名称和描述符

2.1. 字段描述符

根据描述符规则基本数据类型(byte、char、doublc、float, int、loog、shon, boolean)以及代表无返回伯的void类型都用一个大写字符来表示， 而对象类型则用字符L加对象的全限定名来表示,数组则用[

字段描述符的类型含义表

字段描述符	类型	含义
B	byte	基本类型byte
C	char	基本类型char
D	double	基本类型double
F	float	基本类型float
I	int	基本类型int
J	long	基本类型long
LClassName;	reference	对象类型，例如Ljava/lang/Object;
S	short	基本类型short
Z	boolean	基本类型boolean
[	reference	数组类型

- 对象类型的实例变量的字段描述符是L+类的二进制名称的内部形式。
- 对于数组类型，每一维度将使用一个前置的“[”字符来描述，如一个定义为“java.lang.String[][]”类型的二维数组，将被记录为：“[[Ljava/lang/String；”，一个整型数组“int[]”将被记录为“[I”

2.2. 方法描述符

它基于描述符标识字符含义表所示的字符串的类型表示方法， 同时对方法签名的表示做了一些规定。它将函数的参数类型写在一对小括号中， 并在括号右侧给出方法的返回值。比如， 若有如下方法：

```
Object m(int i, double d, Thread t) {… }
```

则它的方法描述符为：

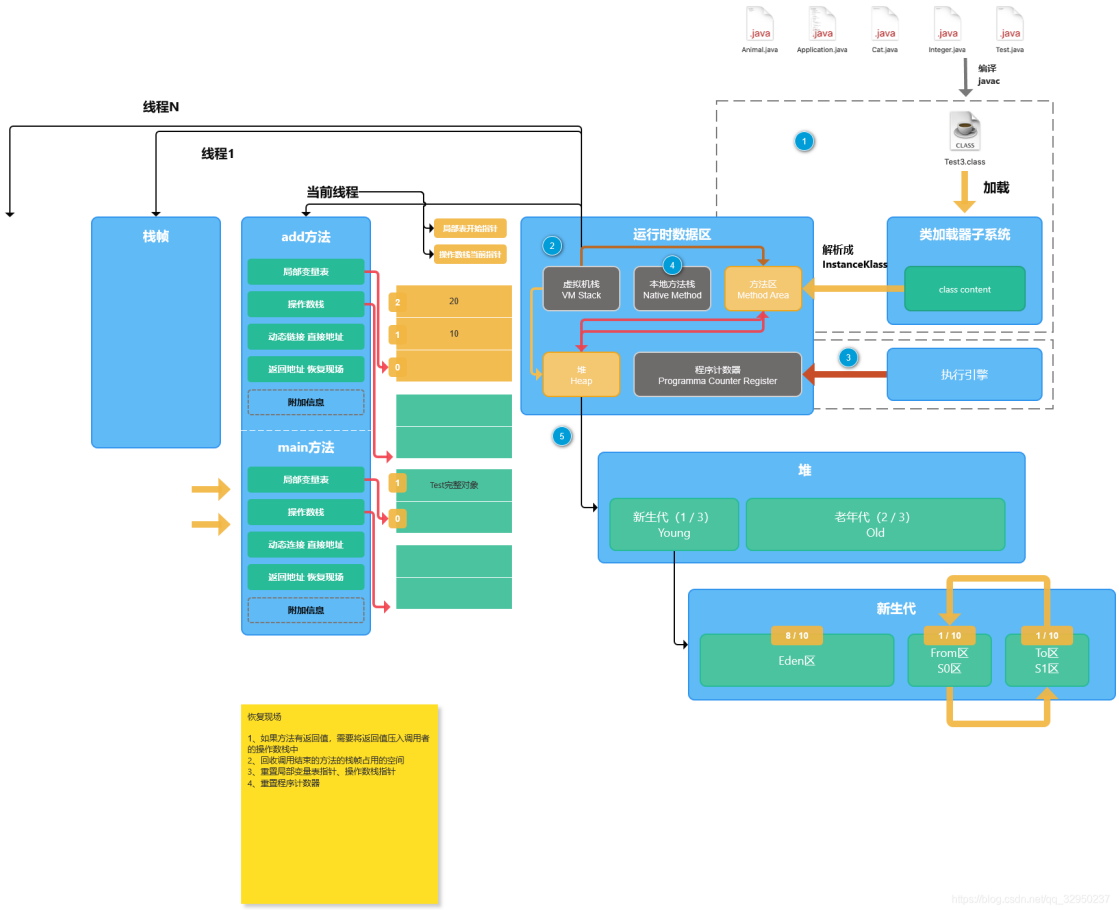
```
(IDLjava/lang/Thread;)Ljava/lang/Object;
```

可以看到， 方法的参数统一列在一对小括号中， “I”表示int ， “D”表示double， `“Ljava/lang/Thread;”` 表示Thread对象。小括号右侧的 `Ljava/lang/Object` ； 表示方法的返同值为Object对象类型。

3. oop-klass内存模型

- oop(ordinary object pointer)，用来描述对象实例信息。
- klass，用来描述 Java 类，是虚拟机内部Java类型结构的对等体 。

JVM内部定义了各种oop-klass，在JVM看来，不仅Java类是对象，Java 方法也是对象， 字节码常量池也是对象，一切皆是对象。JVM使用不同的oop-klass模型来表示各种不同的对象。 而在技术落地时，这些不同的模型就使用不同的 oop 类和 klass 类来表示 。由于JVM使用C/C++编写，因此这些 oop 和 klass 类便是各种不同的C++类。对于Java类型与实例对象，只叫使用 instanceOop 和 instanceKlass 这 2 个 C++类来表示。

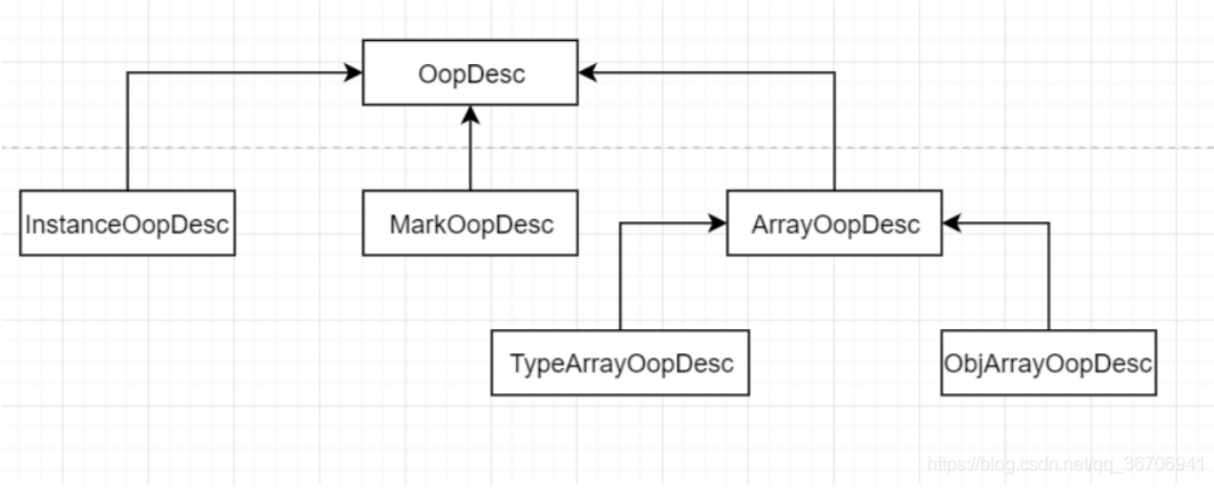


HotSpot JVM 并没有根据 Java 实例对象直接通过虚拟机映射到新建的 C++ 对象，而是设计了一个 oop-klass 模型。oop 指的是 **Ordinary Object Pointer**（普通对象指针，指向被创建的对象，具体通过什么结构创建对象请向下看…），它用来表示对象的**实例信息**，看起来像个指针实际上是藏在指针里的对象；而 klass 则包含**元数据和方法信息**，用来描述 Java 类。

那么为何要设计这样一个一分为二的对象模型呢？这是因为 HotSopt JVM 的设计者不想让每个对象中都含有一个 vtable（虚函数表），所以就把对象模型拆成 klass 和 oop，其中 oop 中不含有任何虚函数，而 klass 就含有虚函数表，可以进行 method dispatch。这个模型其实是参照的 Strongtalk VM 底层的对象模型。

3.1. OOP

```
//定义了oops共同基类
typedef class    oopDesc*                            oop;
//表示一个Java类型实例
typedef class    instanceOopDesc*            instanceOop;
//定义了数组OOPS的抽象基类
typedef class    arrayOopDesc*                    arrayOop;
//表示持有一个OOPS数组
typedef class    objArrayOopDesc*            objArrayOop;
//表示容纳基本类型的数组
typedef class    typeArrayOopDesc*            typeArrayOop;
```



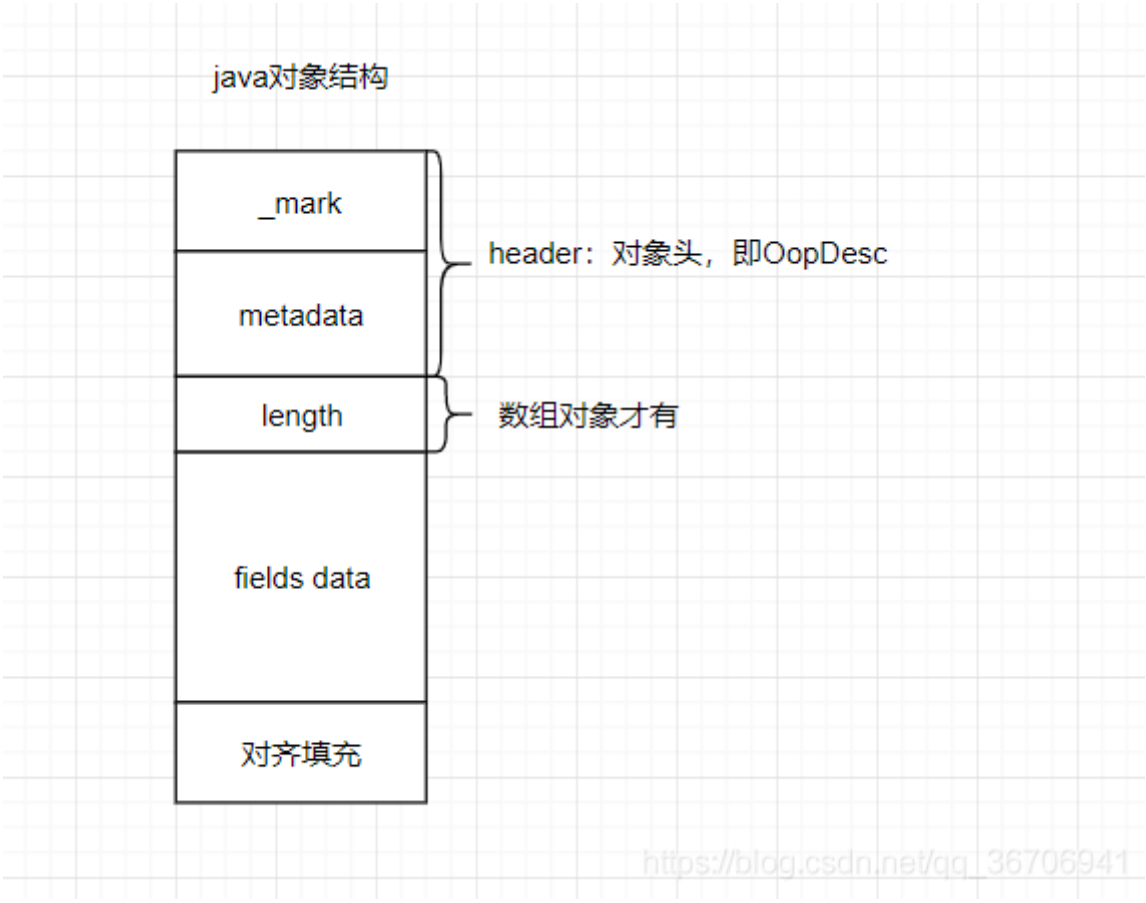
上面是整个oops模块的组成结构，其中包括多个子模块，每个子模块对应一个类型，每个类型的oop都代表一个在JVM内部使用的特定对象的类型。比如 **instanceOopDesc** 表示类实例，arrayOopDesc 表示数组。也就是说，当我们用new创建一个Java对象实例的时候，JVM会创建一个 instanceOopDesc 对象来表示这个Java对象。同理，对于数组实例则是 arrayOopDesc。

而各种 oop 类的共同基类为 **oopDesc** 类：

```
class oopDesc {
    friend class VMStructs;
    friend class JVMCIVMStructs;
  private:
      volatile markOop  _mark; //线程状态 、并发锁 、GC 分代信息等内部标识，这些标识全都打在_mark变量上
      union _metadata {
        Klass*      _klass;//普通指针
         narrowOop       _compressed_klass;//压缩类指针
      } _metadata;
```

InstanceOopDesc主要包含markOop _mark和union _metadata，实例数据则保存在oopDesc中定义的各种field中。

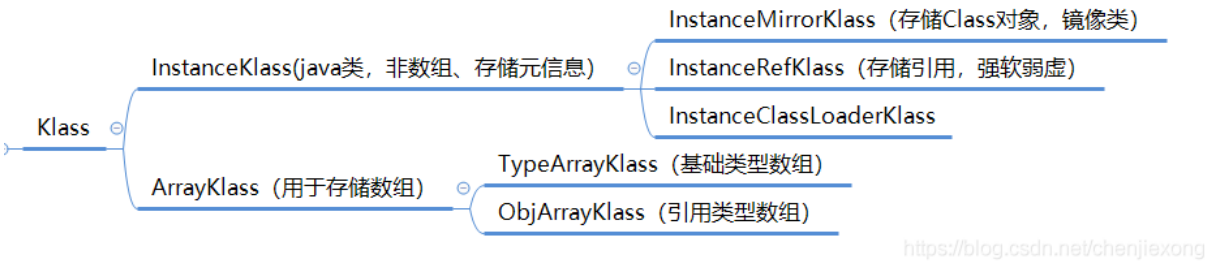
> 1.**Mark Word**：用于存储对象的运行时记录信息，如哈希值、GC 分代年龄(Age)、锁状态标志（偏向锁、轻量级锁、重量级锁）、线程持有的锁、偏向线程 ID、偏向时间戳等。
> 2.**元数据指针**：`OopDesc` 中的 `_metadata` 成员，它是联合体，可以表示未压缩的 Klass 指针(`_klass`)和压缩的 Klass 指针（narrowOop）。对应的 klass 指针指向一个存储类的元数据的 Klass 对象。



- header：由*mark*和 metadata组成， *mark*（即*MarkOopDesc*对象）中保存了*Java*对象的一些信息，如*GC*年龄，锁状态等， metadata存储了指向方法区对象类型数据的指针 *klass*或 compressed_klass指向Klass实例。
- fields data：实例字段没有在OopDesc类中定义相应属性存储，所以只能申请一定大小的空间，按顺序存储，对象字段存放紧跟着OopDesc实例本身占用的内存空间之后，获取时只能通过偏移来取值。
- length：长度，数组对象才有。
- 对齐填充：对齐填充部分不是必须的，只起占位符作用，没有其他含义。HotSpot虚拟机要求对象大小必须是8字节的整数倍，对象头是8字节整数倍，所以填充是对实例数据没有对齐的情况来说的。

3.2. Klass

```
//klassOop的一部分，用来描述语言层的类型
class  Klass;
//在虚拟机层面描述一个Java类
class   instanceKlass;
//专有instantKlass，表示java.lang.Class的Klass
class     instanceMirrorKlass;
//表示类加载器，主要用于遍历类加载器加载的类
class     InstanceClassLoaderKlass;
//专有instantKlass，表示java.lang.ref.Reference的子类的Klass
class     instanceRefKlass;
//表示array类型的抽象基类
class   arrayKlass;
//表示objArrayOop的Klass
class     objArrayKlass;
//表示typeArrayOop的Klass
class     typeArrayKlass;
```



- Klass表示Java类在JVM中的存在形式
  - InstanceKlass表示类的元信息
    - InstanceMirrorKlass表示类的Class对象
    - InstanceClassLoaderKlass 表示类加载器，主要用于遍历类加载器加载的类
  - InstanceRefKlass表示 java.lang.ref.Reference的子类
  - ArrayKlass表示数组类的元信息
    - TypeArrayKlass表示基本数组类的元信息
    - ObjArrayKlass表示引用数组类的元信息
- oopDesc表示JAVA对象在JVM中的存在形式
  - instanceOopDesc表示普通类对象（非数组类对象）
    - arrayOopDesc表示数组类对象
  - typeArrayOopDesc表示基本数组类对象
    - objArrayOopDesc表示引用数组类对象

InstanceKlass

```
// An InstanceKlass is the VM level representation of a Java class.
// It contains all information needed for at class at execution runtime.
```

```
class InstanceKlass: public Klass {
friend class VMStructs;
friend class ClassFileParser;
friend class CompileReplay;
protected:
// Constructor
InstanceKlass(int vtable_len,
int itable_len,
int static_field_size,
int nonstatic_oop_map_size,
ReferenceType rt,
AccessFlags access_flags,
bool is_anonymous);
...
protected:
// Annotations for this class
Annotations*    _annotations;
Klass*          _array_klasses;
ConstantPool* _constants;Array* inner_classes;
char*           _source_debug_extension;
Symbol*         _array_name;
int             _nonstatic_field_size;
int             _static_field_size;    // number words used by static fields (oop
and non-oop) in this klass
u2              _generic_signature_index;
u2              _source_file_name_index;
```

```
u2              _static_oop_field_count;// number of static oop fields in this
klass
u2              _java_fields_count;    // The number of declared Java fields
int             _nonstatic_oop_map_size;// size in words of nonstatic oop map
blocks
bool            _is_marked_dependent;  // used for marking during flushing and
deoptimization
bool            _has_unloaded_dependent;
enum {
_misc_rewritten                = 1 << 0, // methods rewritten.
_misc_has_nonstatic_fields     = 1 << 1, // for sizing with UseCompressedOops
_misc_should_verify_class      = 1 << 2, // allow caching of preverification
misc_is_anonymous             = 1 << 3, // has embedded host_klass field
misc_is_contended             = 1 << 4, // marked with contended annotation
misc_has_default_methods      = 1 << 5, // class/superclass/implemented
interfaces has default methods
misc_declares_default_methods = 1 << 6  // directly declares default methods (any
access)
};
u2              misc_flags;
u2              minor_version;        // minor version number of class file
u2              major_version;        // major version number of class file
Thread*         init_thread;          // Pointer to current thread doing
initialization (to handle recusive initialization)
int             vtable_len;           // length of Java vtable (in words)
int             itable_len;           // length of Java itable (in words)
OopMapCache*    volatile oop_map_cache;   // OopMapCache for all methods in the
klass (allocated lazily)
MemberNameTable* member_names;        // Member names
JNIid*          jni_ids;              // First JNI identifier for static fields
in this class
jmethodID*      methods_jmethod_ids;  // jmethodIDs corresponding to
method_idnum, or NULL if none
nmethodBucket*  dependencies;         // list of dependent nmethods
nmethod*        osr_nmethods_head;    // Head of list of on-stack replacement
nmethods for this class
BreakpointInfo* breakpoints;          // bpt lists, managed by Method*
GrowableArray<PreviousVersionNode > previous_versions;
JvmtiCachedClassFileData* cached_class_file;
volatile u2     _idnum_allocated_count;         change
u1              _init_state;                    // state of class
u1              _reference_type;                // reference type
JvmtiCachedClassFieldMap* _jvmti_cached_class_field_map;  // JVMTI: used during
heap iteration
NOT_PRODUCT(int _verify_count;)  // to avoid redundant verifies
Array<Method> _methods;
Array<Method> _default_methods;
Array<Klass> _local_interfaces;
Array<Klass> transitive_interfaces;
Array*     method_ordering;Array*     default_vtable_indices;
Array*      _fields;
```

_annotations：保存该类的所有注解
_array_klasses：保存数组元素所关联的klass指针
_constants：保存该类的常量池指针
_inner_classes：保存内部类相关的信息
_array_name：如果该类是数组，就会生成数组类名词，如“[Ljava/lang/String;”
_nonstatic_field_size：非静态字段数量
_static_field_size：静态字段数量

_generic_signature_index：泛型签名在常量池中的索引
_source_file_name_index：文件名在常量池中的索引
_static_oop_field_count：该类包含的静态的引用类型字段个数
_java_fields_count：已声明的Java字段数量
_nonstatic_oop_map_size：非静态oop映射块的大小(以字为单位)
_is_marked_dependent：用于刷新和反优化期间打标
_minor_version：主版本号
_major_version：次版本号
_init_thread：初始化此类的线程
_vtable_len：虚函数表的大小
_itable_len：接口函数表的大小
_oop_map_cache：该类所有方法的OopMapCache(延迟分配)
_member_names：MemberNameTable指针
_jni_ids：存放jni_id单向链表的首地址（什么是jni_id?）
_methods_jmethod_ids：与method_idnum对应的jmethodIDs，如果没有，则为NULL
_dependencies：存放nmethod的Bucket的首地址
_osr_nmethods_head：栈上替换nmethods的链表的首地址
_breakpoints：断点链表首地址
_previous_versions：此实例的前一个版本的有趣部分的数组。请参见下面的PreviousVersionWalker
_cached_class_file：缓存的类文件
_idnum_allocated_count：已经分配的idnum的个数
_init_state：该类的状态，值：allocated（已分配内存但未链接）、loaded（加载并插入到类层次结构中但仍未链接）、linked（验证及链接成功但未初始化）、being_initialized（正在初始化）、fully_initialized（已完成初始化）、initialization_error（初始化出错）
_reference_type：引用类型
_methods：存储该类的所有方法对象的指针的数组指针
_default_methods：存储从接口继承的所有方法对象的指针的数组指针
_local_interfaces：数组指针，存储所有实现的接口的指针
_transitive_interfaces：数组，存储直接实现的接口指针+接口间继承实现的接口指针
_method_ordering：包含类文件中方法的原始顺序的Int数组，JVMTI需要用到
_default_vtable_indices：默认构造方法在虚表中的索引
_fields：类的成员属性

InstanceMirrorKlass

```
// An InstanceMirrorKlass is a specialized InstanceKlass for
// java.lang.Class instances.  These instances are special because
// they contain the static fields of the class in addition to the
// normal fields of Class.  This means they are variable sized
// instances and need special logic for computing their size and for
// iteration of their oops.
```

```
class InstanceMirrorKlass: public InstanceKlass {
  friend class VMStructs;
  friend class InstanceKlass;
 private:
  static int _offset_of_static_fields;
  // Constructor
  InstanceMirrorKlass(int vtable_len, int itable_len, int static_field_size, int
nonstatic_oop_map_size, ReferenceType rt, AccessFlags access_flags,  bool
is_anonymous)
    : InstanceKlass(vtable_len, itable_len, static_field_size,
nonstatic_oop_map_size, rt, access_flags, is_anonymous) {}
 public:
  InstanceMirrorKlass() { assert(DumpSharedSpaces || UseSharedSpaces, "only for
CDS"); }
  // Type testing
  bool oop_is_instanceMirror() const             { return true; }
```

> • InstanceKlass中定义了Java运行时环境类所需的所有数据信息，比如 constants 常量池、methods 方法 等（An InstanceKlass is the VM level representation of a Java class. It contains all information needed for at class at execution runtime.）
> • InstanceMirrorKlass是InstanceKlass的一个子类
> • InstanceMirrorKlass是java.lang.Class类专用的InstanceKlass（An InstanceMirrorKlass is a specialized InstanceKlass for java.lang.Class instances.）
> 简单总结一下，InstanceKlass包含了Java运行时环境中类所需的所有数据信息，在类加载这一步，类加载器会将.class文件读入类加载器的class content，然后以InstanceKlass的形式写入JVM内存区域的方法区中，而InstanceMirrorKlass是类所对应的Class对象（java.lang.Class）的InstanceKlass

oop 和 Klass 存储信息比较

typeArrayOop.hpp

> // A typeArrayOop is an array containing basic types (non oop elements).
> // It is used for arrays of {characters, singles, doubles, bytes, shorts, integers, longs}

typeArrayKlass.hpp

> // A TypeArrayKlass is the klass of a typeArray
> // It contains the type and size of the elements

4. 总结:

字节码文件在类加载过程中被解析成Oop-Klass模型，其中Oop存储实例对象及实例数据，Klass存储类的结构信息。

参考:

https://blog.csdn.net/weixin_43767015/article/details/105310047

https://www.cnblogs.com/blwy-zmh/p/11852916.html

https://www.cnblogs.com/blwy-zmh/p/11857953.html

https://blog.csdn.net/qq_32950237/article/details/111397432?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_title~default-5.control&spm=1001.2101.3001.4242

https://github.com/JetBrains/jdk8u_hotspot/blob/master/src/share/vm/oops