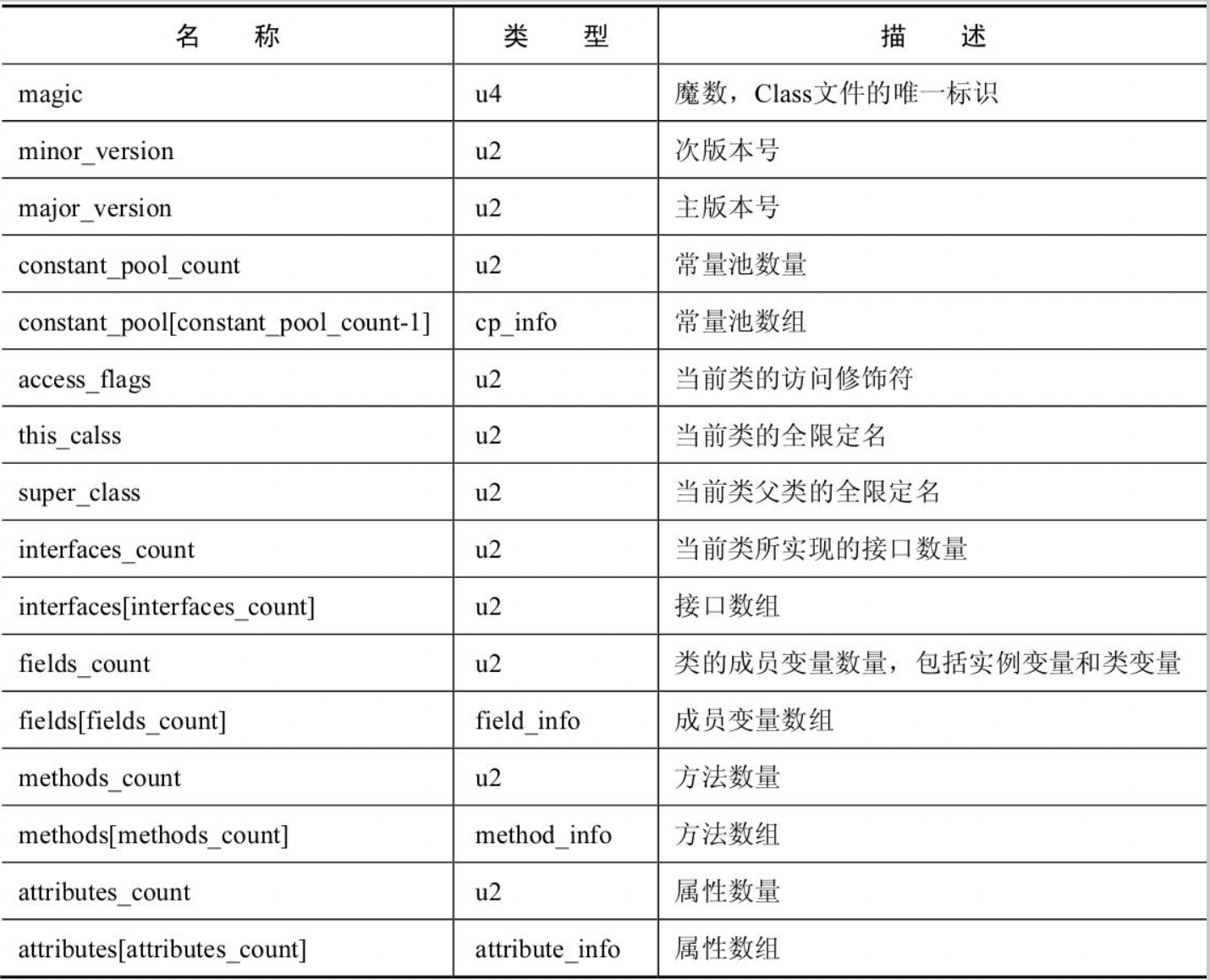
# class文件结构



u1、u2、u4和u8来分别表示1个字节、2个字节、4个字节和8个字节的无符号数，以“\_info”结尾的类型都表示表类型。读取与写入class文件都是根据该表有严格的顺序

## 魔数及版本号

每个Class文件的头4个字节被称为魔数（Magic Number），它的唯一作用是确定这个文件是否为一个能被虚拟机接受的Class文件。紧接着魔数的4个字节存储的是Class文件的版本号：第5和第6个字节是次版本号（Minor Version），第7和第8个字节是主版本号（Major Version）。

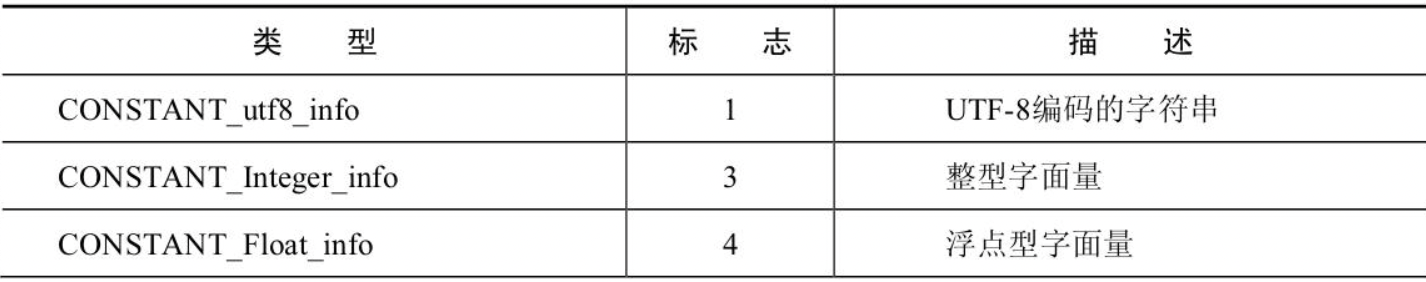
魔数及版本号的定义

魔数的定义，来源：com.sun.tools.javac.jvm.ClassFile  
 public final static int JAVA\_MAGIC = 0xCAFEBABE;  
 版本号的定，来源：com.sun.tools.javac.jvm.Target  
 // 主版本号  
 public final int majorVersion;   
 // 次版本号  
 public final int minorVersion;  
 private Target(\_, int majorVersion, int minorVersion) {  
 ...  
 this.majorVersion = majorVersion;  
 this.minorVersion = minorVersion;  
 }

魔数及版本号的使用

来源：com.sun.tools.javac.jvm.ClassWriter  
 public void writeClassFile(OutputStream out, ClassSymbol c)  
 throws IOException, PoolOverflow, StringOverflow {  
 ...  
 // poolbuf是常量池缓冲区对象   
 poolbuf.appendInt(JAVA\_MAGIC);  
 // 次版本号  
 poolbuf.appendChar(target.minorVersion);  
 // 主版本号  
 poolbuf.appendChar(target.majorVersion);  
 ...  
 }

## 常量池





来源：com.sun.tools.javac.jvm.ClassWriter#writePool  
void writePool(Pool pool) throws PoolOverflow, StringOverflow {  
 ...  
 while (i < pool.pp) {  
 ...  
 // 往常量池中追加各种信息  
 if (value instanceof MethodSymbol) {  
 // CONSTANT\_Methodref\_info/CONSTANT\_InterfaceMethodref\_info根据格式入常量池缓冲  
 } else if (value instanceof VarSymbol) {  
 // CONSTANT\_Fieldref\_info根据格式入常量池  
 } else if (value instanceof Name) {  
 // CONSTANT\_Utf8\_info根据格式入常量池  
 } else if (value instanceof ClassSymbol) {  
 // CONSTANT\_Class\_info根据格式入常量池  
 } else if (value instanceof NameAndType) {  
 // CONSTANT\_NameandType\_info根据格式入常量池  
 } else if (value instanceof Integer) {  
 // CONSTANT\_Integer\_info根据格式入常量池  
 } else if (value instanceof Long) {  
 // CONSTANT\_Long\_info根据格式入常量池  
 } else if (value instanceof Float) {  
 // CONSTANT\_Float\_info根据格式入常量池  
 } else if (value instanceof Double) {  
 // CONSTANT\_Double\_info根据格式入常量池  
 } else if (value instanceof String) {  
 // CONSTANT\_String\_info根据格式入常量池  
 } else if (value instanceof Type) {  
 // CONSTANT\_Class\_info根据格式入常量池  
 }  
 ...  
 }  
 // 写入常量池常量个数及常量池数组  
 putChar(poolbuf, poolCountIdx, pool.pp);  
}

## 类定义的基本信息

### 访问表示符

来源：com.sun.tools.javac.jvm.ClassWriter#writeClassFile  
public void writeClassFile(OutputStream out, ClassSymbol c)  
 throws IOException, PoolOverflow, StringOverflow {  
 ...  
 // 调用adjustFlags()方法调整ClassSymbol对象c的flags\_field变量的值  
 int flags = adjustFlags(c.flags());  
 // 计算类的标识符  
 if ((flags & PROTECTED) != 0)  
 flags |= PUBLIC;  
 flags = flags & ClassFlags & ~STRICTFP;  
 if ((flags & INTERFACE) == 0)  
 flags |= ACC\_SUPER;  
 if (c.isInner() && c.name.isEmpty())  
 flags &= ~FINAL;  
 // 调用databuf.appendChar()方法将flags追加到databuf缓冲中  
 databuf.appendChar(flags);  
 ...  
}

### 类、父类及接口集合

来源：com.sun.tools.javac.jvm.ClassWriter#writeClassFile  
public void writeClassFile(OutputStream out, ClassSymbol c)  
 throws IOException, PoolOverflow, StringOverflow {  
 ...  
 // 当前类符号放入常量池并将在常量池中的索引追加到databuf中  
 databuf.appendChar(pool.put(c));  
 // 父类放入常量池并追加到databuf中  
 // 如果当前类没有父类，如Object类没有父类时保存0值，而0指向常量池中第0项，表示不引用任何常量池项  
 databuf.appendChar(supertype.tag == CLASS ? pool.put(supertype.tsym) : 0);  
 // 接口数量  
 databuf.appendChar(interfaces.length());  
 for (List<Type> l = interfaces; l.nonEmpty(); l = l.tail)  
 // 循环追加接口在常量池中的索引  
 databuf.appendChar(pool.put(l.head.tsym));  
 ...  
}

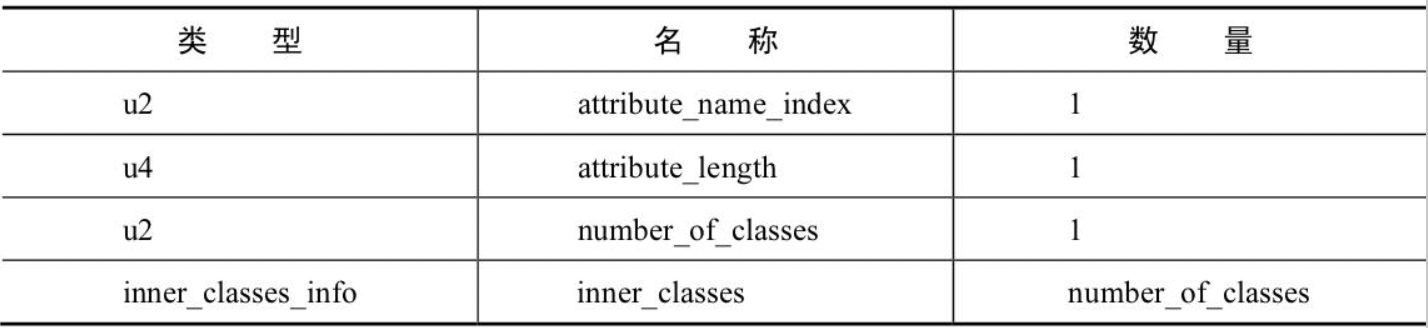
## 字段集合、方法集合

来源：com.sun.tools.javac.jvm.ClassWriter#writeClassFile  
public void writeClassFile(OutputStream out, ClassSymbol c)  
 throws IOException, PoolOverflow, StringOverflow {  
 ...  
 int fieldsCount = 0; // 字段个数  
 int methodsCount = 0; // 方法个数  
 // 循环处理成员  
 for (Scope.Entry e = c.members().elems; e != null; e = e.sibling) {  
 switch (e.sym.kind) {  
 // 成员是变量  
 case VAR:  
 // 计算c中定义的字段数量并追加到databuf中  
 fieldsCount++; break;  
 // 成员是方法  
 case MTH:  
 if ((e.sym.flags() & HYPOTHETICAL) == 0)  
 // 计算方法的个数  
 methodsCount++;  
 break;  
 ...  
 }  
 }  
 // 字段个数  
 databuf.appendChar(fieldsCount);  
 // 处理字段集合  
 writeFields(c.members().elems);  
 // 方法个数  
 databuf.appendChar(methodsCount);  
 // 处理方法集合  
 writeMethods(c.members().elems);  
 ...  
  
}

## 类属性集合



### InnerClasses属性



来源：com.sun.tools.javac.jvm.ClassWriter#writeClassFile  
public void writeClassFile(OutputStream out, ClassSymbol c)  
 throws IOException, PoolOverflow, StringOverflow {  
 ...  
 // 循环处理成员  
 for (Scope.Entry e = c.members().elems; e != null; e = e.sibling) {  
 switch (e.sym.kind) {  
 ...  
 // 成员是内部类  
 case TYP:  
 // 调用enterInner()方法对内部类进行处理  
 enterInner((ClassSymbol)e.sym); break;  
 }  
 }  
 ...  
 if (innerClasses != null) {  
 // 对InnerClasses集合中保存的内部类进行处理  
 writeInnerClasses();  
 acount++;  
 }  
}

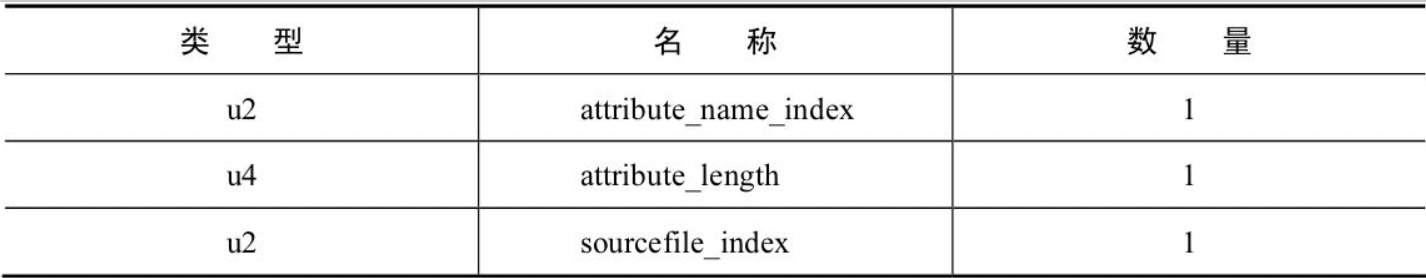
来源：com.sun.tools.javac.jvm.ClassWriter#enterInner  
 void enterInner(ClassSymbol c) {  
 ...  
 // 对内部类进行编译  
 c.complete();  
 if (c.type.tag != CLASS) return; // arrays  
 if (pool != null &&  
 c.owner.kind != PCK &&  
 // c是内部类并且innerClasses集合中没有包含这个内部类时  
 (innerClasses == null || !innerClasses.contains(c))) {  
 if (c.owner.kind == TYP)  
 enterInner((ClassSymbol)c.owner);  
 // 将内部类符号放入常量池  
 pool.put(c);  
 // 将内部类name对象放入常量池  
 pool.put(c.name);  
 if (innerClasses == null) {  
 innerClasses = new HashSet<ClassSymbol>();  
 innerClassesQueue = new ListBuffer<ClassSymbol>();  
 pool.put(names.InnerClasses);  
 }  
 // 将这个内部类保存到innerClasses集合和innerClassesQueue队列中  
 innerClasses.add(c);  
 innerClassesQueue.append(c);  
 }  
 }

**inner\_classes\_info属性结构**



来源：com.sun.tools.javac.jvm.ClassWriter#writeInnerClasses  
 void writeInnerClasses() {  
 ...  
 // 内部类的个数  
 databuf.appendChar(innerClassesQueue.length());  
 // 处理每个内部类  
 for (List<ClassSymbol> l = innerClassesQueue.toList();  
 l.nonEmpty();  
 l = l.tail) {  
 ClassSymbol inner = l.head;  
 // 计算内部类访问标识  
 char flags = (char) adjustFlags(inner.flags\_field);  
 if ((flags & INTERFACE) != 0)  
 flags |= ABSTRACT; // 当为接口时去掉ABSTRACT  
 if (inner.name.isEmpty())  
 flags &= ~FINAL; // 当为匿名类时去掉FINAL  
 // 写入inner\_classes\_info表结构中的inner\_class\_info\_index  
 databuf.appendChar(pool.get(inner));  
 // outer\_class\_info\_index  
 databuf.appendChar(  
 inner.owner.kind == TYP ? pool.get(inner.owner) : 0);  
 // inner\_name\_index  
 databuf.appendChar(  
 !inner.name.isEmpty() ? pool.get(inner.name) : 0);  
 // inner\_class\_access\_flags  
 databuf.appendChar(flags);  
 }  
 ...  
 }

### SourceFile属性



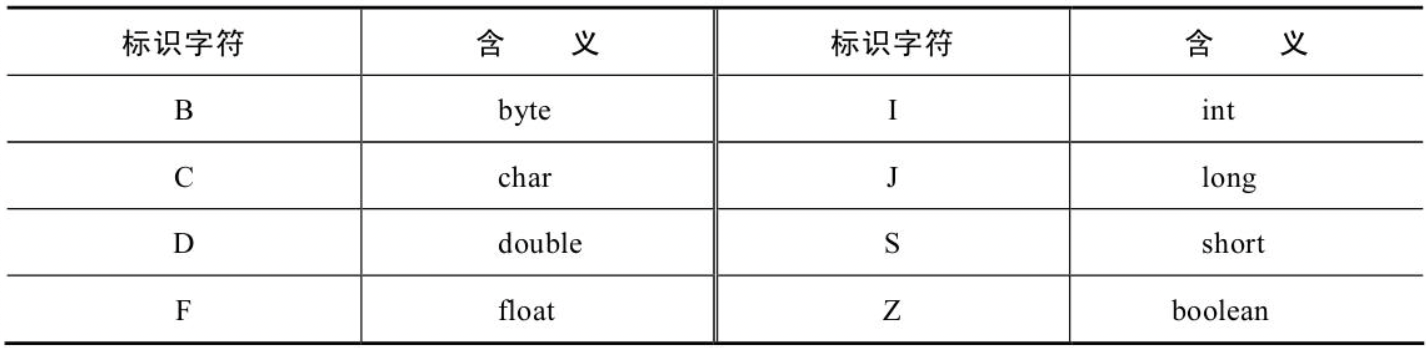
来源：com.sun.tools.javac.jvm.ClassWriter#writeClassFile  
public void writeClassFile(OutputStream out, ClassSymbol c)  
 throws IOException, PoolOverflow, StringOverflow {  
 ...  
 if (c.sourcefile != null && emitSourceFile) {  
 int alenIdx = writeAttr(names.SourceFile);  
 String simpleName = BaseFileObject.getSimpleName(c.sourcefile);  
 databuf.appendChar(c.pool.put(names.fromString(simpleName)));  
 endAttr(alenIdx);  
 acount++;  
 }   
 ...  
}

## 描述符和签名

### 描述符

描述符是一个描述字段或方法类型的字符串。

基本类型规则



对象类型规则：

String：Ljava/lang/String

数组类型规则：

String[]：[Ljava/lang/String

String[] []：[[Ljava/lang/String

/\*\*  
描述符分别为：  
(JI)V  
(ZILjava/lang/String;II)Z  
\*/  
class Test{  
 void wait(long timeout,int nanos){ }  
 boolean regionMatches(boolean ignoreCase,int toOffset,String other,int offeset,int len){ }  
}

### 签名

签名是类、方法或字段的泛型相关信息。签名被描述为字符串存放到了常量池中。

类签名

package chapter18;  
interface IA<T>{ }  
class Parent<T>{ }  
public class Test<A,B extends IA<String>,C extends Parent&IA> { }  
// 签名描述为：  
// <A:Ljava/lang/Object;B::Lchapter18/IA<Ljava/lang/String;>;C:Lchapter18/  
// Parent;:Lchapter18/IA;>Ljava/lang/Object;

字段签名

List<? extends Number> a ;  
List<? super Integer> b ;  
List<?> c ;  
// 签名描述为  
// Ljava/util/List<+Ljava/lang/Number;>;  
// Ljava/util/List<-Ljava/lang/Integer;>;  
// Ljava/util/List<\*>;

方法签名

package chapter18;  
import java.io.Serializable;  
import java.util.List;  
public class Test {  
 public <A,B extends Serializable> void test(A a,List<B> b){ }  
}  
// 签名描述为  
// <A:Ljava/lang/Object;B::Ljava/io/Serializable;>(TA;Ljava/util/List<TB;>;)V

### 实现

来源：com.sun.tools.javac.jvm.ClassWriter#assembleSig()  
 void assembleSig(Type type) {  
 switch (type.tag) {  
 case BYTE:  
 sigbuf.appendByte('B');  
 break;  
 case SHORT:  
 sigbuf.appendByte('S');  
 break;  
 case CHAR:  
 sigbuf.appendByte('C');  
 break;  
 case INT:  
 sigbuf.appendByte('I');  
 break;  
 case LONG:  
 sigbuf.appendByte('J');  
 break;  
 case FLOAT:  
 sigbuf.appendByte('F');  
 break;  
 case DOUBLE:  
 sigbuf.appendByte('D');  
 break;  
 case BOOLEAN:  
 sigbuf.appendByte('Z');  
 break;  
 case VOID:  
 sigbuf.appendByte('V');  
 break;  
 case CLASS:  
 // 主要看类和接口的实现，无论是计算类和接口的描述符还是签名，都是以“L”开头，以“;”结尾，  
 sigbuf.appendByte('L');  
 // 中间部分调用assembleClassSig()方法进行计算  
 assembleClassSig(type);  
 sigbuf.appendByte(';');  
 break;  
 case ARRAY:  
 // 数组以“[”开头  
 ArrayType at = (ArrayType)type;  
 sigbuf.appendByte('[');  
 assembleSig(at.elemtype);  
 break;  
 case METHOD:  
 MethodType mt = (MethodType)type;  
 sigbuf.appendByte('(');  
 assembleSig(mt.argtypes);  
 sigbuf.appendByte(')');  
 assembleSig(mt.restype);  
 if (hasTypeVar(mt.thrown)) {  
 for (List<Type> l = mt.thrown; l.nonEmpty(); l = l.tail) {  
 sigbuf.appendByte('^');  
 assembleSig(l.head);  
 }  
 }  
 break;  
 case WILDCARD: {  
 // 通配符类型和类型变量只会在计算签名时使用，因为在计算描述符时会进行类型擦写，  
 // 所以不会存在通配符类型和类型变量，实现也相对简单，按照相关的文法生成签名字符串即可。  
 WildcardType ta = (WildcardType) type;  
 switch (ta.kind) {  
 case SUPER: // ? super   
 sigbuf.appendByte('-');  
 assembleSig(ta.type);  
 break;  
 case EXTENDS:// ? extend  
 sigbuf.appendByte('+');  
 assembleSig(ta.type);  
 break;  
 case UNBOUND: // ?  
 sigbuf.appendByte('\*');  
 break;  
 default:  
 throw new AssertionError(ta.kind);  
 }  
 break;  
 }  
 case TYPEVAR:  
 // 类型参数类型  
 sigbuf.appendByte('T');  
 sigbuf.appendName(type.tsym.name);  
 sigbuf.appendByte(';');  
 break;  
 case FORALL:  
 // 辅助泛型类型  
 ForAll ft = (ForAll)type;  
 // 获取方法的签名时，可通过调用assembleParamsSig()方法计算形式类型参数的签名  
 assembleParamsSig(ft.tvars);  
 assembleSig(ft.qtype);  
 break;  
 ...  
 }  
 }