基础篇

一、JVM

二、编译与反编译

javac 、javap 、jad 、CRF

三、Java基础知识

0.关键字及其含义

assert：断言，在Java中，assert关键字是从JAVA SE 1.4 引入的，为了避免和老版本的Java代码中使用了assert关键字导致错误，Java在执行的时候默认是不启动断言检查的（这个时候，所有的断言语句都 将忽略！），如果要开启断言检查，则需要用开关-enableassertions或-ea来开启。

assert关键字语法很简单，有两种用法：

1、assert <boolean表达式>

如果<boolean表达式>为true，则程序继续执行。

如果为false，则程序抛出AssertionError，并终止执行。

2、assert <boolean表达式> : <错误信息表达式>

如果<boolean表达式>为true，则程序继续执行。

如果为false，则程序抛出java.lang.AssertionError，并输入<错误信息表达式>。

transient：当对象被序列化时（写入字节序列到目标文件）时，transient阻止实例中那些用此关键字声明的变量持久化；当对象被反序列化时（从源文件读取字节序列进行重构），这样的实例变量值不会被持久化和恢复。

1.阅读源代码

String、Integer、Long、Enum、BigDecimal、ThreadLocal、ClassLoader & URLClassLoader、ArrayList & LinkedList、 HashMap & LinkedHashMap & TreeMap & CouncurrentHashMap、HashSet & LinkedHashSet & TreeSet

2.Java中各种变量类型

1个字节8位。

整数类型：byte（1个字节）、short（2个字节）、int（4个字节）、long（8个字节）

字符类型：char（2个字节）

浮点类型：float（4个字节）、double（8个字节）

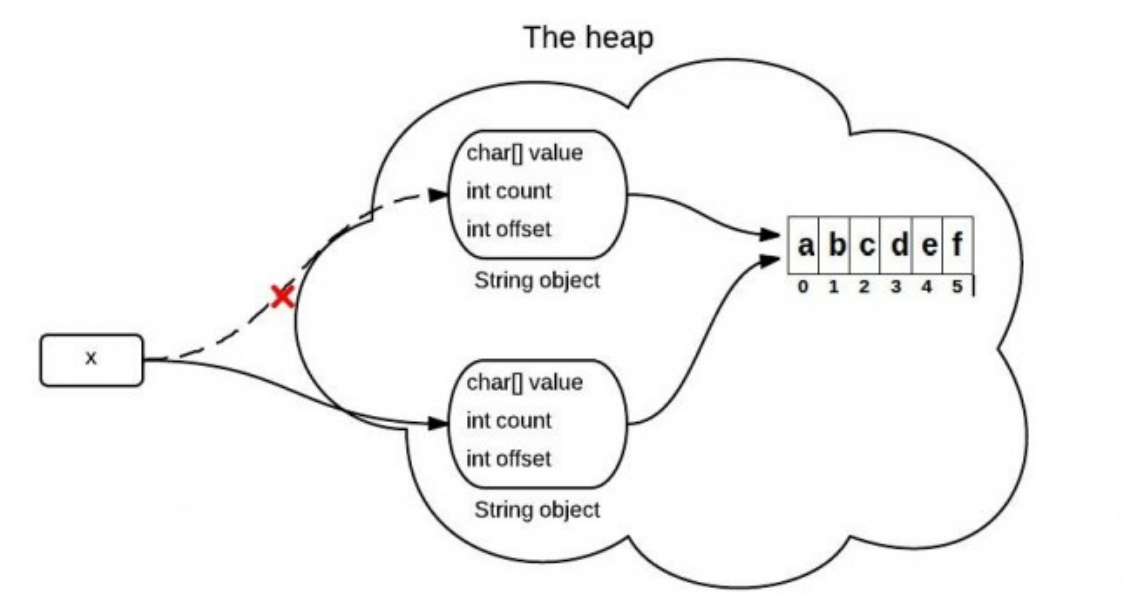
布尔类型：boolean

3.熟悉Java String的使用，熟悉String的各种函数：

1）JDK 6和JDK 7中substring的原理及区别

在JDK6中，String是通过字符数组实现的。在jdk 6 中，String类包含三个成员变量：char value[]， int offset，int count。他们分别用来存储真正的字符数组，数组的第一个位置索引以及字符串中包含的字符个数。

当调用substring方法的时候，会创建一个新的string对象，但是这个string的值仍然指向堆中的同一个字符数组。这两个对象中只有count和offset 的值是不同的。

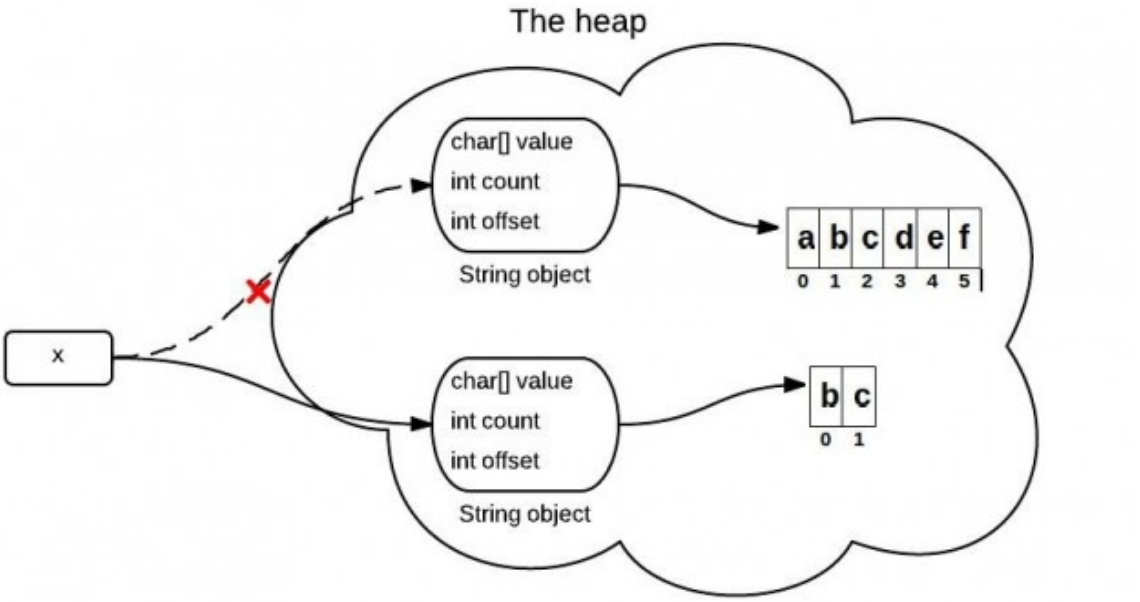



```
String(int offset, int count, char value[]) {
    this.value = value;
    this.offset = offset;
    this.count = count;
}
// 在substring时，是调整offset和count的值，引用的依旧是原始字符串数组
public String substring(int beginIndex, int endIndex) {
    //check boundary
    return  new String(offset + beginIndex, endIndex - beginIndex,
value);
}
```

如果你有一个很长很长的字符串，但是当你使用substring进行切割的时候你只需要很短的一段。这可能导致性能问题，因为你需要的只是一小段字符序列，但是你却引用了整个字符串（因为这个非常长的字符数组一直在被引用，所以无法被回收，就可能导致内存泄露）。在JDK 6中，一般用以下方式来解决该问题，原理其实就是生成一个新的字符串并引用他。

```
x = x.substring(x, y) + ""
```

在**JDK 7**中得到解决。在JDK 7 中，substring方法会在堆内存中创建一个新的数组。




```
//JDK 7
public String(char value[], int offset, int count) {
    //check boundary
    this.value = Arrays.copyOfRange(value, offset, offset + count);
}

public String substring(int beginIndex, int endIndex) {
    //check boundary
    int subLen = endIndex - beginIndex;
    return new String(value, beginIndex, subLen);
}
```

2）replaceFirst、replaceAll、replace区别

- replace方法是将字符串中的oldChar替换成newChar，注意是字符全部替换。方法是依次判断字符是否相等
- replaceFirst参数是两个字符串，第一个是用来做正则表达式的pattern，最终会替换掉第一个匹配的字符串
- replaceAll参数是两个字符串，第一个是用来做正则表达式的pattern，最终会替换掉第一个匹配的字符串
- 替换前后原始字符串没有发生改变

```
// replace方法是将字符串中的oldChar替换成newChar，注意是字符全部替换
public String replace(char oldChar, char newChar) {
    if (oldChar != newChar) {
        int len = value.length;
        int i = -1;
        char[] val = value; /* avoid getfield opcode */

        while (++i < len) {
            if (val[i] == oldChar) {
                break;
            }
        }
```

```
            if (i < len) {
                char buf[] = new char[len];
                for (int j = 0; j < i; j++) {
                    buf[j] = val[j];
                }
                while (i < len) {
                    char c = val[i];
                    buf[i] = (c == oldChar) ? newChar : c;
                    i++;
                }
                return new String(buf, true);
            }
        }
        return this;
    }
    // replaceFirst参数是两个字符串，第一个是用来做正则表达式的pattern，最终会替换掉第一个匹配的字符串
    public String replaceFirst(String regex, String replacement) {
        return
    Pattern.compile(regex).matcher(this).replaceFirst(replacement);
    }
    // replaceAll参数是两个字符串，第一个是用来做正则表达式的pattern，最终会替换掉第一个匹配的字符串
    public String replaceAll(String regex, String replacement) {
        return
    Pattern.compile(regex).matcher(this).replaceAll(replacement);
    }
```

3）String对“+”的重载

在Java语言中，操作符重载是不被允许的。尽管操作符重载会提高项目的灵活性，但是会提高项目的复杂性，可读性也大大降低。操作符重载与Java的设计思想（严格的面向对象）相悖。

但是对String对象而言，它是可以直接+将两个String对象的字符串值相加。乍看起来这是对+的重载，但我们可以通过class文件看出，这只是JVM做的语法糖。

来看一个简单的例子：

```
public class A{
    public static void main(String[] args){
        String a = "1";
        String b = "2";
        System.out.println(a+b);
    }
}
```

我们运行一下，结果是12，没什么问题。

反编译一下这个java文件：

> *javap -c A.java*

```
Compiled from "A.java"
public class A {
  public A();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: ldc           #2                  // String 1
       2: astore_1
       3: ldc           #3                  // String 2
       5: astore_2
       6: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       9: new           #5                  // class java/lang/StringBuilder
      12: dup
      13: invokespecial #6                  // Method java/lang/StringBuilder."<init>":()V
      16: aload_1
      17: invokevirtual #7                  // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
      20: aload_2
      21: invokevirtual #7                  // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
      24: invokevirtual #8                  // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
      27: invokevirtual #9                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      30: return
}
```

暂且不管左边那些汇编啥意思，单看右边的注释

9：新建了一个StringBuilder对象 17和21：两次append() 24：StringBuilder.toString()方法

所以说，String的+操作根本不是重载，他只是JVM做的有个简化操作，实际上还是调用了StringBuilder进行相加。

在阿里巴巴编码规约中有明确提到，推荐在循环时的字符拼接采用StringBuilder.append()方法而不是+。因为循环使用+会最终导致创建多个StringBuilder对象，造成资源浪费。

4）String.valueOf和Integer.toString的区别

String.valueOf支持将多种数据类型转换成字符串。对于整数类型，它最终会调用Integer.toString(i)方法进行转换

5）String、StringBuffer、StringBuilder

String，字符串常量，不可变

StringBuffer，字符串变量，是线程安全的；

StringBuilder，字符串变量，不是线程安全的，更快；

StringBuilder中的能容纳的字符串的长度限制，最大长度为Integer.MAX_VALUE：

```java
// 构造函数
public StringBuilder() {
    super(16);
}

public StringBuilder(int capacity) {
    super(capacity);
}
// 进行追加操作
public AbstractStringBuilder append(String str) {
    if (str == null)
        return appendNull();
    int len = str.length();
    ensureCapacityInternal(count + len);
    str.getChars(0, len, value, count);
```

```
    count += len;
    return this;
}
// 数组扩容
private void ensureCapacityInternal(int minimumCapacity) {
    // overflow-conscious code
    if (minimumCapacity - value.length > 0)
        expandCapacity(minimumCapacity);
}
void expandCapacity(int minimumCapacity) {
    int newCapacity = value.length * 2 + 2;
    if (newCapacity - minimumCapacity < 0)
        newCapacity = minimumCapacity;
    if (newCapacity < 0) {
        if (minimumCapacity < 0) // overflow
            throw new OutOfMemoryError();
        newCapacity = Integer.MAX_VALUE;
    }
    value = Arrays.copyOf(value, newCapacity);
}
```

StringBuffer容量扩容，最大容量为Integer.MAX_VALUE:

```
// 构造函数
public StringBuffer() {
    super(16);
}
public StringBuffer(int capacity) {
    super(capacity);
}
public StringBuffer(String str) {
    super(str.length() + 16);
    append(str);
}
// 进行追加操作，保证线程安全的方式是加synchronized关键字修饰方法
public synchronized StringBuffer append(String str) {
    toStringCache = null;
    super.append(str);
    return this;
}
public AbstractStringBuilder append(String str) {
    if (str == null)
        return appendNull();
    int len = str.length();
    ensureCapacityInternal(count + len);
    str.getChars(0, len, value, count);
    count += len;
    return this;
}
private void ensureCapacityInternal(int minimumCapacity) {
    // overflow-conscious code
    if (minimumCapacity - value.length > 0)
        expandCapacity(minimumCapacity);
}
```

```
void expandCapacity(int minimumCapacity) {
    int newCapacity = value.length * 2 + 2;
    if (newCapacity - minimumCapacity < 0)
        newCapacity = minimumCapacity;
    if (newCapacity < 0) {
        if (minimumCapacity < 0) // overflow
            throw new OutOfMemoryError();
        newCapacity = Integer.MAX_VALUE;
    }
    value = Arrays.copyOf(value, newCapacity);
}
```

6）常见例子

示例1：

```
String s0="kvill";
String s1="kvill";
String s2="kv" + "ill";
System.out.println( s0==s1 );
System.out.println( s0==s2 );

结果为：
true
true
```

分析：首先，我们要知结果为道Java 会确保一个字符串常量只有一个拷贝。

因为例子中的 s0和s1中的”kvill”都是字符串常量，它们在编译期就被确定了，所以s0==s1为true；而”kv”和”ill”也都是字符串常量，当一个字 符串由多个字符串常量连接而成时，它自己肯定也是字符串常量，所以s2也同样在编译期就被解析为一个字符串常量，所以s2也是常量池中” kvill”的一个引用。所以我们得出s0==s1==s2；

示例2：

```
String s0="kvill";
String s1=new String("kvill");
String s2="kv" + new String("ill");
System.out.println( s0==s1 );
System.out.println( s0==s2 );
System.out.println( s1==s2 );

结果为：
false
false
false
```

分析：用new String() 创建的字符串不是常量，不能在编译期就确定，所以new String() 创建的字符串不放入常量池中，它们有自己的地址空间。

s0还是常量池 中"kvill”的应用，s1因为无法在编译期确定，所以是运行时创建的新对象”kvill”的引用，s2因为有后半部分 new String(”ill”)所以也无法在编译期确定，所以也是一个新创建对象”kvill”的应用;明白了这些也就知道为何得出此结果了。

示例3：

```
String a = "a1";
String b = "a" + 1;
System.out.println((a == b)); //result = true
String a = "atrue";
String b = "a" + "true";
System.out.println((a == b)); //result = true
String a = "a3.4";
String b = "a" + 3.4;
System.out.println((a == b)); //result = true
```

分析：JVM对于字符串常量的"+"号连接，将程序编译期，JVM就将常量字符串的"+"连接优化为连接后的值，拿"a" + 1来说，经编译器优化后在class中就已经是a1。在编译期其字符串常量的值就确定下来，故上面程序最终的结果都为true。

示例4：

```
String a = "ab";
String bb = "b";
String b = "a" + bb;
System.out.println((a == b)); //result = false
```

分析：JVM对于字符串引用，由于在字符串的"+"连接中，有字符串引用存在，而引用的值在程序编译期是无法确定的，即"a" + bb无法被编译器优化，只有在程序运行期来动态分配并将连接后的新地址赋给b。所以上面程序的结果也就为false。

示例5：

```
String a = "ab";
final String bb = "b";
String b = "a" + bb;
System.out.println((a == b)); //result = true
```

分析：和[4]中唯一不同的是bb字符串加了final修饰，对于final修饰的变量，它在编译时被解析为常量值的一个本地拷贝存储到自己的常量 池中或嵌入到它的字节码流中。所以此时的"a" + bb和"a" + "b"效果是一样的。故上面程序的结果为true。

示例6：

```
String a = "ab";
final String bb = getBB();
String b = "a" + bb;
System.out.println((a == b)); //result = false
private static String getBB() {  return "b";   }
```

分析：JVM对于字符串引用bb，它的值在编译期无法确定，只有在程序运行期调用方法后，将方法的返回值和"a"来动态连接并分配地址为b，故上面 程序的结果为false。

4.自动拆装箱

自动装箱：

Java 编译器把原始类型自动转换为封装类的过程称为自动装箱（autoboxing），这相当于调用 valueOf 方法。

```
Integer a = 10; //this is autoboxing
Integer b = Integer.valueOf(10); //under the hood
```

缓存策略：

这种 Integer 缓存策略仅在自动装箱（autoboxing）的时候有用，使用构造器创建的 Integer 对象不能被缓存。

上面的规则适用于整数区间 -128 到 +127。因为Integer.valueof就是缓存策略执行的地方。

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

IntegerCache：

Javadoc 详细的说明这个类是用来实现缓存支持，并支持 -128 到 127 之间的自动装箱过程。最大值 127 可以通过 JVM 的启动参数 -XX:AutoBoxCacheMax=size 修改。 缓存通过一个 for 循环实现。从小到大的创建尽可能多的整数并存储在一个名为 cache 的整数数组中。这个缓存会在 Integer 类第一次被使用的时候被初始化出来。以后，就可以使用缓存中包含的实例对象，而不是创建一个新的实例(在自动装箱的情况下)。

```
static {
    // high value may be configured by property
    int h = 127;
    String integerCacheHighPropValue =

 sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high"
);
    if (integerCacheHighPropValue != null) {
        try {
            int i = parseInt(integerCacheHighPropValue);
            i = Math.max(i, 127);
            // Maximum array size is Integer.MAX_VALUE
            h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
        } catch( NumberFormatException nfe) {
            // If the property cannot be parsed into an int, ignore
it.
        }
    }
    high = h;

    cache = new Integer[(high - low) + 1];
    int j = low;
    for(int k = 0; k < cache.length; k++)
        cache[k] = new Integer(j++);

    // range [-128, 127] must be interned (JLS7 5.1.7)
```

```
24      assert IntegerCache.high >= 127;
25  }
```

其他缓冲行为：

其他缓存的对象

这种缓存行为不仅适用于Integer对象。我们针对所有整数类型的类都有类似的缓存机制。

有 ByteCache 用于缓存 Byte 对象

有 ShortCache 用于缓存 Short 对象

有 LongCache 用于缓存 Long 对象

有 CharacterCache 用于缓存 Character 对象

Byte，Short，Long 有固定范围: -128 到 127。对于 Character, 范围是 0 到 127。除了 Integer 可以通过参数改变范围外，其它的都不行。

5.熟悉Java中各种关键字

transient、instanceof、volatile、synchronized、final、static、const 原理及用法。

6.集合类

1）常用集合类的使用

2）ArrayList和LinkedList和Vector的区别

3）SynchronizedList和Vector的区别

4）HashMap、HashTable、ConcurrentHashMap区别

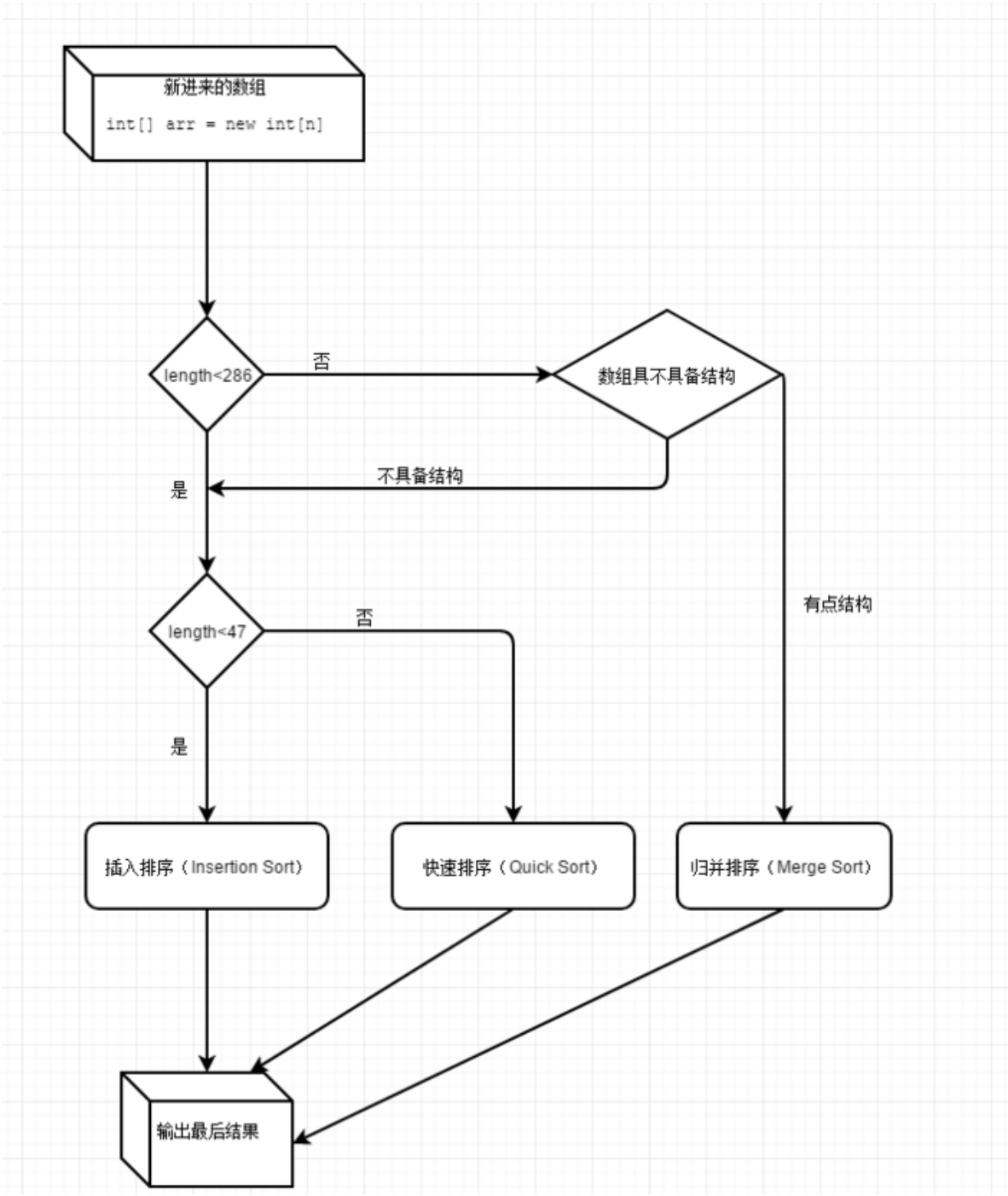
5）Java 8中stream相关用法

6）apache集合处理工具类的使用

7）不同版本的JDK中HashMap的实现的区别以及原因

8）Arrays.sort()使用的是什么排序方法?

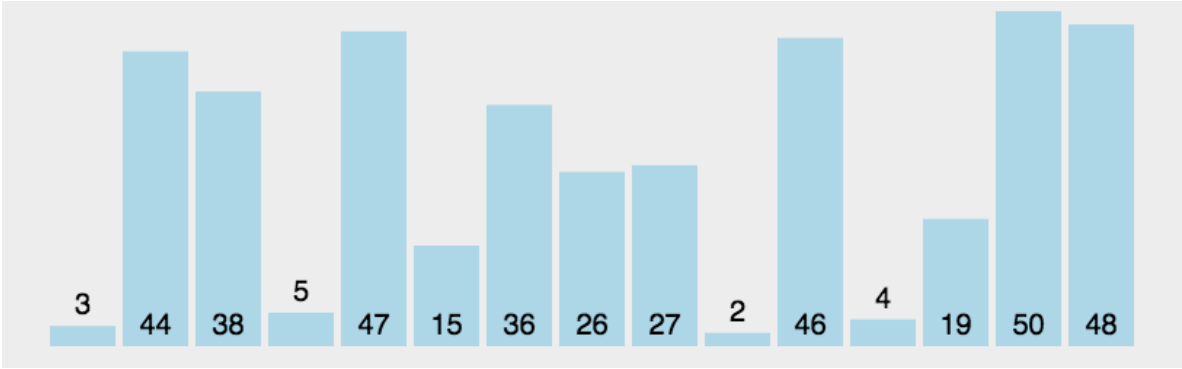
其中有方法的重载，下面针对的是基本数据类型的排序。



如果数组中元素的个数少于47个的时候，使用的是插入排序。这是因为当数据量不大的时候，插入排序常数项的时间复杂度很低，而数据本身O(N)也差别不大，因此可以选择插入排序。

如果数组中元素的个数在[47, 286)范围内，使用的是快速排序。

- 从数列中挑出五个元素，称为“基准”（pivot）；
- 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
- 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。



如果数组中元素的个数在[286,~)范围内，使用的是归并排序，因为归并排序是稳定的。

但在此之前，它有个小动作：

```
// Check if the array is nearly sorted
for (int k = left; k < right; run[count] = k) {
    if (a[k] < a[k + 1]) { // ascending
        while (++k <= right && a[k - 1] <= a[k]);
    } else if (a[k] > a[k + 1]) { // descending
        while (++k <= right && a[k - 1] >= a[k]);
        for (int lo = run[count] - 1, hi = k; ++lo < --hi; ) {
            int t = a[lo]; a[lo] = a[hi]; a[hi] = t;
        }
    } else { // equal
       for (int m = MAX_RUN_LENGTH; ++k <= right && a[k - 1] ==
a[k]; ) {
           if (--m == 0) {
                sort(a, left, right, true);
                return;
            }
        }
    }
    /*
     * The array is not highly structured,
     * use Quicksort instead of merge sort.
     */
    if (++count == MAX_RUN_COUNT) {
        sort(a, left, right, true);
        return;
    }
}
```

这里主要作用是看他数组具不具备结构：实际逻辑是分组排序，每降序为一个组，像1,9,8,7,6,8。9到6是降序，为一个组，然后把降序的一组排成升序：1,6,7,8,9,8。然后最后的8后面继续往后面找。。。

每遇到这样一个降序组，++count，当count大于MAX_RUN_COUNT（67），被判断为这个数组不具备结构（也就是这数据时而升时而降），然后送给之前的sort(里面的快速排序)的方法（The array is not highly structured,use Quicksort instead of merge sort.）。

如果count少于MAX_RUN_COUNT（67）的，说明这个数组还有点结构，就继续往下走下面的归并排序：

```
// Merging
for (int last; count > 1; count = last) {
    for (int k = (last = 0) + 2; k <= count; k += 2) {
        int hi = run[k], mi = run[k - 1];
        for (int i = run[k - 2], p = i, q = mi; i < hi; ++i) {
            if (q >= hi || p < mi && a[p + ao] <= a[q + ao]) {
                b[i + bo] = a[p++ + ao];
            } else {
                b[i + bo] = a[q++ + ao];
            }
        }
        run[++last] = hi;
    }
    if ((count & 1) != 0) {
        for (int i = right, lo = run[count - 1]; --i >= lo;
             b[i + bo] = a[i + ao]);
        run[++last] = right;
    }
    int[] t = a; a = b; b = t;
    int o = ao; ao = bo; bo = o;
}
```

7.枚举

枚举的用法、枚举与单例、Enum类

8.Java IO&Java NIO，并学会使用

bio、nio和aio的区别、三种IO的用法与原理、netty

详情见：==========java IO.md

9.Java反射与javassist

反射与工厂模式、 java.lang.reflect.* *Java中用到的设计模式以及原因*

Java的反射机制是Java特性之一，反射机制是构建框架技术的基础所在。Java程序要能够运行，Java虚拟机需要事先加载java类，目前我们的程序在编译期就已经确定哪些java类需要被加载。Java的反射机制是在编译时并不确定哪个类需要被加载，而是在程序运行时才加载、探知、自审。这样的特点就是反射。

何为自审：通过Java的反射机制能够探知到java类的基本结构，这种对Java类结构探知的能力，我们称为Java类的“自审”。

Java的反射原理最典型的应用就是各种Java IDE：比如Jcreator，eclipse，idea等，当我们构建出一个对象时，去调用该对象的方法和属性的时候。一按点，IDE工具就会自动的把该对象能够使用的所有的方法和属性全部都列出来，供我们进行选择。这就是利用了Java反射的原理，是对我们创建对象的探知、自审的过程。

Javassist是一个开源的分析、编辑和创建Java字节码的类库。

10.Java序列化：

1）什么是序列化与反序列化、为什么序列化

序列化 (Serialization)是将对象的状态信息转换为可以存储或传输的形式的过程。一般将一个对象存储至一个储存媒介，例如档案或是记忆体缓冲等。在网络传输过程中，可以是字节或是XML等格式。而字节的或XML编码格式可以还原完全相等的对象。这个相反的过程又称为反序列化。

在Java中，我们可以通过多种方式来创建对象，并且只要对象没有被回收我们都可以复用该对象。但是，我们创建出来的这些Java对象都是存在于JVM的堆内存中的。只有JVM处于运行状态的时候，这些对象才可能存在。一旦JVM停止运行，这些对象的状态也就随之而丢失了。

但是在真实的应用场景中，我们需要将这些对象持久化下来，并且能够在需要的时候把对象重新读取出来。Java的对象序列化可以帮助我们实现该功能。

对象序列化机制（object serialization）是Java语言内建的一种对象持久化方式，通过对象序列化，可以把对象的状态保存为字节数组，并且可以在有需要的时候将这个字节数组通过反序列化的方式再转换成对象。对象序列化可以很容易的在JVM中的活动对象和字节数组（流）之间进行转换。

在Java中，对象的序列化与反序列化被广泛应用到RMI(远程方法调用)及网络传输中。

2）序列化底层原理

3）序列化与单例模式

4）protobuf

5）为什么说序列化并不安全

11.注解

元注解、自定义注解、Java中常用注解使用、注解与反射的结合

12.JMS

什么是Java消息服务、JMS消息传送模型

JMS(Java Message Service)是访问企业消息系统的标准API，它便于消息系统中的Java应用程序进行消息交换,并且通过提供标准的产生、发送、接收消息的接口简化企业应用的开发。

JMS是用于和面向消息的中间件相互通信的应用程序接口。它既支持点对点(point-to-point)的域，又支持发布/订阅(publish /subscribe)类型的域，并且提供对下列类型的支持：经认可的消息传递,事务型消息的传递，一致性消息和具有持久性的订阅者支持。JMS还提供了另一种方式来对您的应用与旧的后台系统相集成。

13.JMX

java.lang.management.、 *javax.management.*

JMX--Java Management Extensions，即Java管理扩展,是一个为应用程序、设备、系统等植入管理功能的框架。JMX可以跨越一系列异构操作系统平台、系统体系结构和网络传输协议，灵活的开发无缝集成的系统、网络和服务管理应用。

JMX体系结构分为以下四个层次：

1）设备层（Instrumentation Level）：主要定义了信息模型。在JMX中，各种管理对象以管理构件的形式存在，需要管理时，向MBean服务器进行注册。该层还定义了通知机制以及一些辅助元数据类。

2）代理层（Agent Level）：主要定义了各种服务以及通信模型。该层的核心是一个MBean服务器，所有的管理构件都需要向它注册，才能被管理。注册在MBean服务器上管理构件并不直接和远程应用程序进行通信，它们通过协议适配器和连接器进行通信。而协议适配器和连接器也以管理构件的形式向MBean服务器注册才能提供相应的服务。

3）分布服务层（Distributed Service Level）：主要定义了能对代理层进行操作的管理接口和构件，这样管理者就可以操作代理。然而，当前的JMX规范并没有给出这一层的具体规范。

4）附加管理协议API：定义的API主要用来支持当前已经存在的网络管理协议，如SNMP、TMN、CIM/WBEM等。

websocket 与 **jmx**和**jms** 协作工作来管理**web**项目

14.泛型：

1）泛型与继承

2）类型擦除

3）泛型中K T V E

4）object等的含义、泛型各种用法

15.单元测试

junit、mock、mockito、内存数据库（h2）

16.正则表达式

java.lang.util.regex.* *****常见的正则表达式匹配*****

字符	描述	字符	描述
\n	换行符	\r	回车符
\s	匹配任何空白字符，包括制表符、空格、换页符等	\S	匹配任何非空白字符
\t	匹配制表符	\v	匹配垂直制表符

举例：

- runo**o+**b，可以匹配 runo**o**b、runo**oo**b、runo**ooooo**b 等，**+** 号代表前面的字符必须至少出现一次（**1**次或多次）。
- runo**o***b，可以匹配 runob、runo**o**b、runo**ooooo**b 等，***** 号代表字符可以不出现，也可以出现一次或者多次（**0**次、或**1**次、或多次）。

- colou?r 可以匹配 color 或者 colour，?问号代表前面的字符最多只可以出现一次（0次、或1次）。

```
[a-z] //匹配所有的小写字母
[A-Z] //匹配所有的大写字母
[a-zA-Z] //匹配所有的字母
[0-9] //匹配所有的数字
[0-9\.\-] //匹配所有的数字，句号和减号
[ \f\r\t\n] //匹配所有的白字符
```

如果要匹配一个由一个小写字母和一位数字组成的字符串，比如"z2"、"t6"或"g7"，但不是"ab2"、"r2d3"或"b52"的话，用这个模式：

```
^[a-z][0-9]$
```

我们要求第一个字符不能是数字，用^来排除，这个模式与"&5"、"g7"及"-2"是匹配的，但与"12"、"66"是不匹配的：

```
^[^0-9][0-9]$
```

```
[^a-z] //除了小写字母以外的所有字符
[^\\\/\^] //除了(\)(/)(^)之外的所有字符
[^\"\'] //除了双引号(")和单引号(')之外的所有字符
```

17.常用的Java工具库

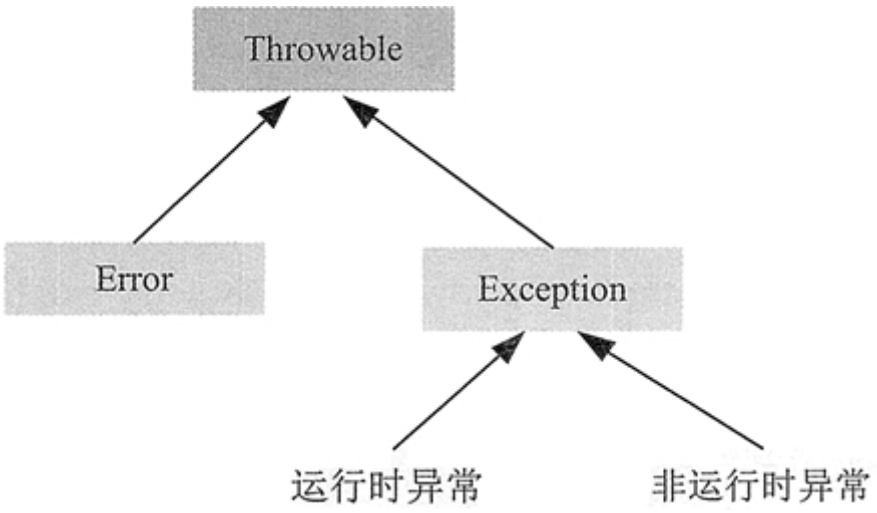
commons.lang, commons.*... guava-libraries netty

18.什么是API&SPI

19.异常

异常类型、正确处理异常、自定义异常

（1）异常类型



在 Java 中所有异常类型都是内置类 java.lang.Throwable 类的子类，即 Throwable 位于异常类层次结构的顶层。Throwable 类是所有异常和错误的超类，下面有 Error 和 Exception 两个子类分别表示错误和异常。其中异常类 Exception 又分为运行时异常和非运行时异常，这两种异常有很大的区别，也称为不检查异常（Unchecked Exception）和检查异常（Checked Exception）。

- Exception 类用于用户程序可能出现的异常情况，它也是用来创建自定义异常类型类的类。
- Error 定义了在通常环境下不希望被程序捕获的异常。Error类型的异常用于Java运行时由系统显示与运行时系统本身有关的错误。堆栈溢出是这种错误的一例。

运行时异常都是RuntimeException 类及其子类异常，如NullPointerException、**IndexOutOfBoundsException** 等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般由程序逻辑错误引起，程序应该从逻辑角度尽可能避免这类异常的发生。

非运行时异常是指 RuntimeException 以外的异常，类型上都属于 Exception 类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如 IOException、ClassNotFoundException 等以及用户自定义的 Exception 异常，一般情况下不自定义检查异常。下表列出了一些常见的异常类型及它们的作用。

异常类型	说明
Exception	异常层次结构的根类
RuntimeException	运行时异常，多数 java.lang 异常的根类
ArithmeticException	算术谱误异常，如以零做除数
ArraylndexOutOfBoundException	数组大小小于或大于实际的数组大小
NullPointerException	尝试访问 null 对象成员，空指针异常
ClassNotFoundException	不能加载所需的类
NumberF ormatException	数字转化格式异常，比如字符串到 float 型数字的转换无效
IOException	I/O 异常的根类
FileNotFoundException	找不到文件
EOFException	文件结束
InterruptedException	线程中断
IllegalArgumentException	方法接收到非法参数
ClassCastException	类型转换异常

异常类型	说明
SQLException	操作数据库异常

(2) 异常处理机制

Java 的异常处理通过 5 个关键字来实现：try、catch、throw、throws 和 finally。try catch 语句用于捕获并处理异常，finally 语句用于在任何情况下（除特殊情况外）都必须执行的代码，throw 语句用于抛出异常，throws 语句用于声明可能会出现的异常。如果代码在 try 内部执行一条 System.exit() 语句，则应用程序将终止而不会执行 finally。

Java 的异常处理机制提供了一种结构性和控制性的方式来处理程序执行期间发生的事件。异常处理的机制如下：

- 在方法中用 try catch 语句捕获并处理异常，catch 语句可以有多个，用来匹配多个异常。
- 对于处理不了的异常或者要转型的异常，在方法的声明处通过 throws 语句抛出异常，即由上层的调用方法来处理。

在以下4种特殊情况下，**finally**块不会被执行：

1. 在finally语句块第一行发生了异常。 因为在其他行，finally块还是会得到执行
2. 在前面的代码中用了System.exit(int)已退出程序。 exit是带参函数；若该语句在异常语句之后，finally会执行
3. 程序所在的线程死亡。
4. 关闭CPU。

以下代码是异常处理程序的基本结构：

```
try
{
    逻辑程序块
}
catch(ExceptionType1 e)
{
    处理代码块1
}
catch (ExceptionType2 e)
{
    处理代码块2
    throw(e);     //再抛出这个"异常"
}
finally
{
    释放资源代码块
}
```

(3) throws 声明异常

当一个方法产生一个它不处理的异常时，那么就需要在该方法的头部声明这个异常，以便将该异常传递到方法的外部进行处理。可以使用 throws 关键字在方法的头部声明一个异常，其具体格式如下：

```
returnType method_name(paramList) throws Exception 1,Exception2,…{…}
```

如果有多个异常类，它们之间用逗号分隔。这些异常类可以是方法中调用了可能抛出异常的方法而产生的异常，也可以是方法体中生成并抛出的异常。

在编写类继承代码时要注意，子类在覆盖父类带 throws 子句的方法时，子类的方法声明中的 throws 子句不能出现父类对应方法的 throws 子句中没有的异常类型，因此 throws 子句可以限制子类的行为。也就是说，子类方法抛出的异常不会超过父类定义的范围。

throw 抛出异常：

throw 语句用来直接抛出一个异常，后接一个可抛出的异常类对象，其语法格式如下：

```
throw ExceptionObject;
```

其中，ExceptionObject 必须是 Throwable 类或其子类的对象。如果是自定义异常类，也必须是 Throwable 的直接或间接子类。例如，以下语句在编译时将会产生语法错误：

```
throw new String("抛出异常");     //因为String类不是Throwable类的子类
```

当 **throw 语句执行时，它后面的语句将不执行**，此时程序转向调用者程序，寻找与之相匹配的 catch 语句，执行相应的异常处理程序。如果没有找到相匹配的 catch 语句，则再转向上一层的调用程序。这样逐层向上，直到最外层的异常处理程序终止程序并打印出调用栈情况。

（4）自定义异常

如果 Java 提供的内置异常类型不能满足程序设计的需求，可以设计自己的异常类型。自定义异常类必须继承现有的 Exception 类或 Exception 的子类来创建，其语法形式为：

```
<class><自定义异常名><extends><Exception>
```

在编码规范上，一般将自定义异常类的类名命名为 XXXException，其中 XXX 用来代表该异常的作用。

自定义异常类一般包含两个构造方法：一个是无参的默认构造方法，另一个构造方法以字符串的形式接收一个定制的异常消息，并将该消息传递给超类的构造方法。如：

```
class IntegerRangeException extends Exception
{
    public IntegerRangeException()
    {
        super();
    }
    public IntegerRangeException(String s)
    {
        super(s);
    }
}
```

20.时间处理

时区、时令、Java中时间API

21.编码方式

解决乱码问题、常用编码方式

22.语法糖

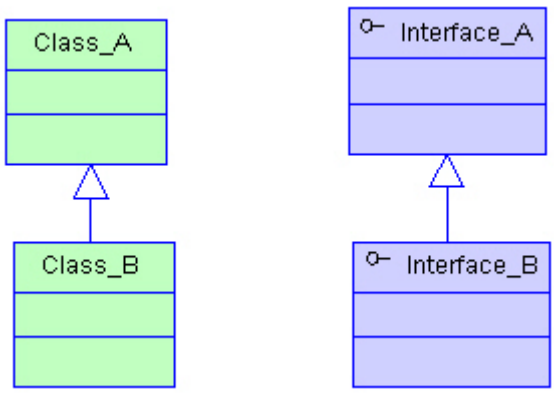
Java中语法糖原理、解语法糖

23.组合和继承

Java代码的复用有继承，组合以及代理三种具体的表现形式。下面主要介绍继承和组合。

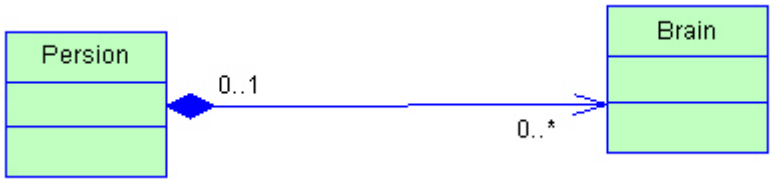
（1）继承

继承（Inheritance）是一种联结类与类的层次模型。指的是一个类（称为子类、子接口）继承另外的一个类（称为父类、父接口）的功能，并可以增加它自己的新功能的能力，继承是类与类或者接口与接口之间最常见的关系；继承是一种is-a关系。类可以继承类，接口可以继承接口。



（2）组合

组合(Composition)体现的是整体与部分、拥有的关系，即has-a的关系。



（3）组合和继承的比较

在继承结构中，父类的内部细节对于子类是可见的。所以我们通常也可以说通过继承的代码复用是一种白盒式代码复用。（如果基类的实现发生改变，那么派生类的实现也将随之改变。这样就导致了子类行为的不可预知性；）

组合是通过对现有的对象进行拼装（组合）产生新的、更复杂的功能。因为在对象之间，各自的内部细节是不可见的，所以我们也说这种方式的代码复用是黑盒式代码复用。（因为组合中一般都定义一个类型，所以在编译期根本不知道具体会调用哪个实现类的方法）

继承，在写代码的时候就要指名具体继承哪个类，所以，在编译期就确定了关系。（从基类继承来的实现是无法在运行期动态改变的，因此降低了应用的灵活性。）

组合，在写代码的时候可以采用面向接口编程。所以，类的组合关系一般在运行期确定。

组合关系	继承关系
优点：不破坏封装，整体类与局部类之间松耦合，彼此相对独立	缺点：破坏封装，子类与父类之间紧密耦合，子类依赖于父类的实现，子类缺乏独立性
优点：具有较好的可扩展性	缺点：支持扩展，但是往往以增加系统结构的复杂度为代价
优点：支持动态组合。在运行时，整体对象可以选择不同类型的局部对象	缺点：不支持动态继承。在运行时，子类无法选择不同的父类
优点：整体类可以对局部类进行包装，封装局部类的接口，提供新的接口	缺点：子类不能改变父类的接口
缺点：整体类不能自动获得和局部类同样的接口	优点：子类能自动继承父类的接口
缺点：创建整体类的对象时，需要创建所有局部类的对象	优点：创建子类的对象时，无须创建父类的对象

（4）如何选择

相信很多人都知道面向对象中有一个比较重要的原则『多用组合、少用继承』或者说『组合优于继承』。从前面的介绍已经优缺点对比中也可以看出，组合确实比继承更加灵活，也更有助于代码维护。

所以，

> *建议在同样可行的情况下，优先使用组合而不是继承。*
>
> ```
> 1 因为组合更安全，更简单，更灵活，更高效。
> ```

注意，并不是说继承就一点用都没有了，前面说的是【在同样可行的情况下】。有一些场景还是需要使用继承的，或者是更适合使用继承。

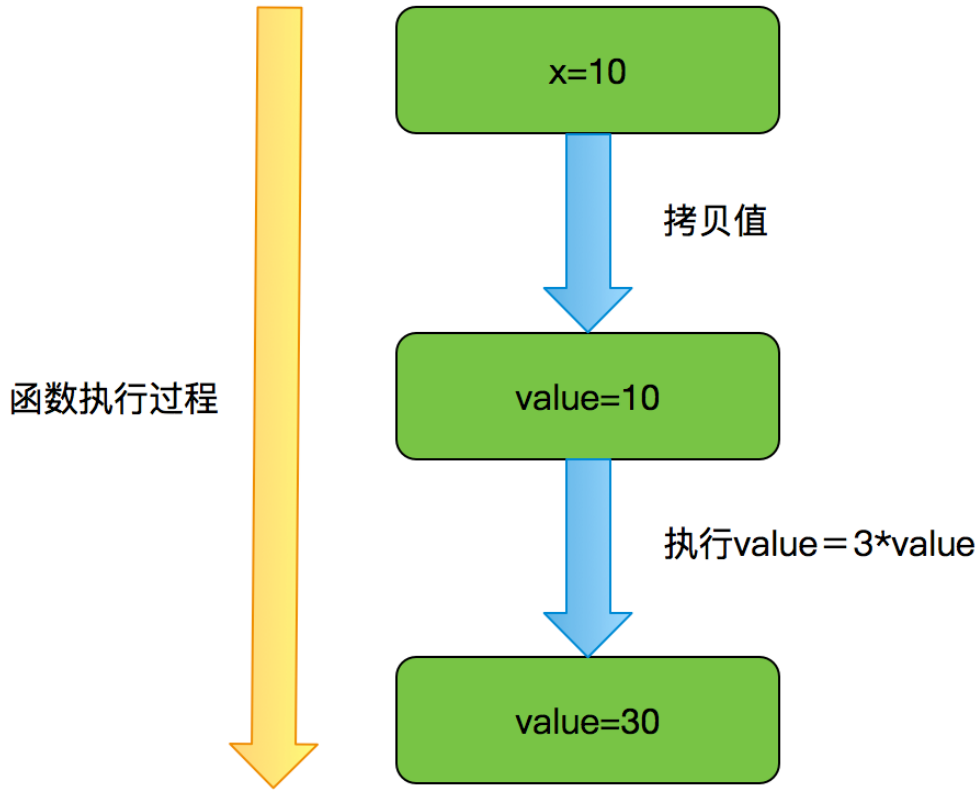
> *继承要慎用，其使用场合仅限于你确信使用该技术有效的情况。一个判断方法是，问一问自己是否需要从新类向基类进行向上转型。如果是必须的，则继承是必要的。反之则应该好好考虑是否需要继承。《Java编程思想》*
>
> *只有当子类真正是超类的子类型时，才适合用继承。换句话说，对于两个类A和B，只有当两者之间确实存在`is-a`关系的时候，类B才应该继续类A。《Effective Java》*

24.函数调用中按值调用和按引用调用

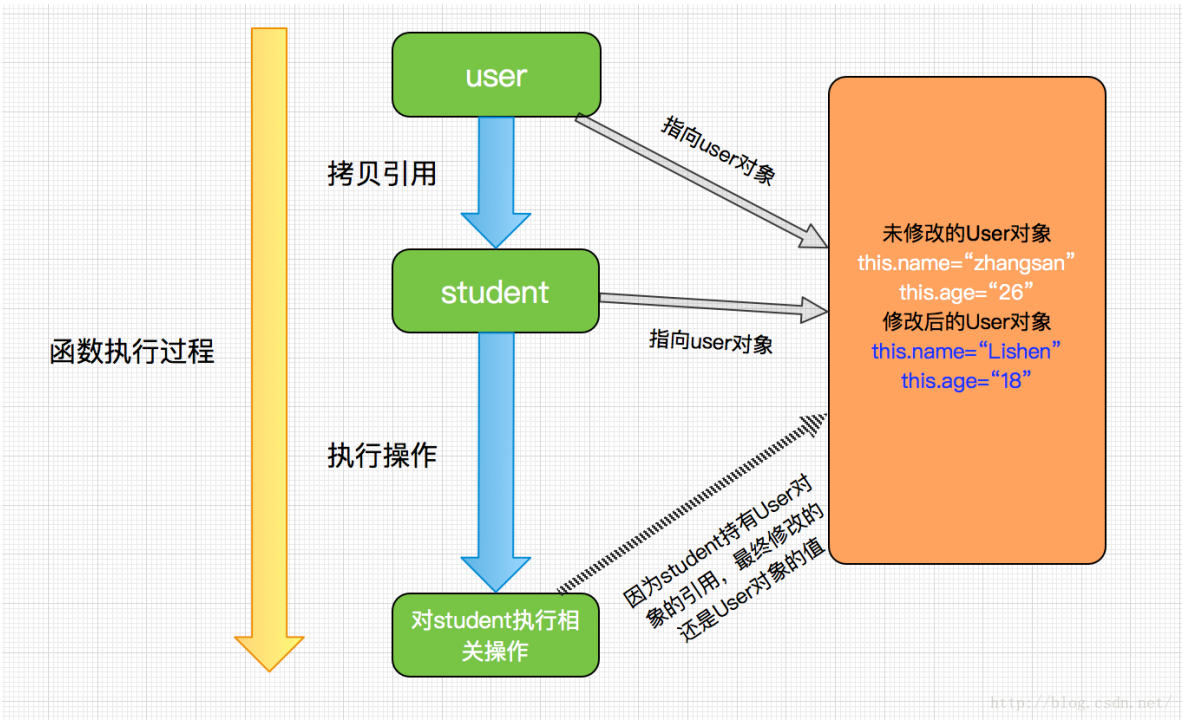
Java中的方法有关参数传递的两个术语：

- 按值调用（call by value）
- 按引用调用（call by references）

按值调用就是方法接收的是调用者提供的值，按引用调用则表示方法接收的是调用者提供的变量地址，Java中没有指针的概念。我们可以修改传递引用所对应的变量值，而不能修改传递值调用所对应的变量的值。修改了函数参数中按值调用的值的值的时候，调用者的值未发生改变；修改了函数参数中按引用调用的值的值的时候，调用者的引用的值为发生改变，引用所指向的对象的值可能发生改变。



结论：当传递方法参数类型为基本数据类型（数字以及布尔值）时，一个方法是不可能修改一个基本数据类型的参数。



结论：当传递方法参数类型为引用数据类型时，一个方法将修改一个引用数据类型的参数所指向对象的值。

进阶篇

三、并发编程

内容详情见：

四、Java底层知识

1.字节码、class文件格式

2.CPU缓存，L1，L2，L3和伪共享

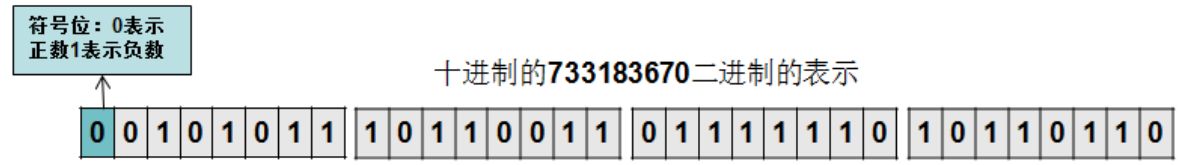
3.尾递归

4.位运算

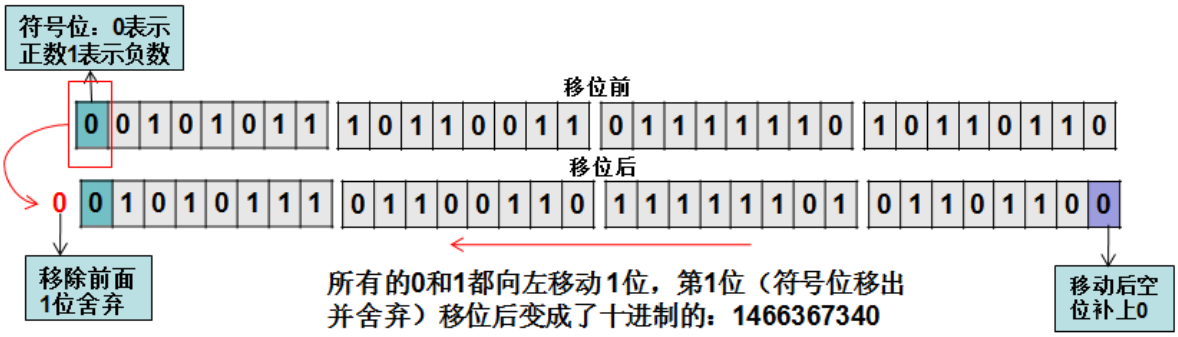
加减乘除取余数

正数：原码、反码、补码

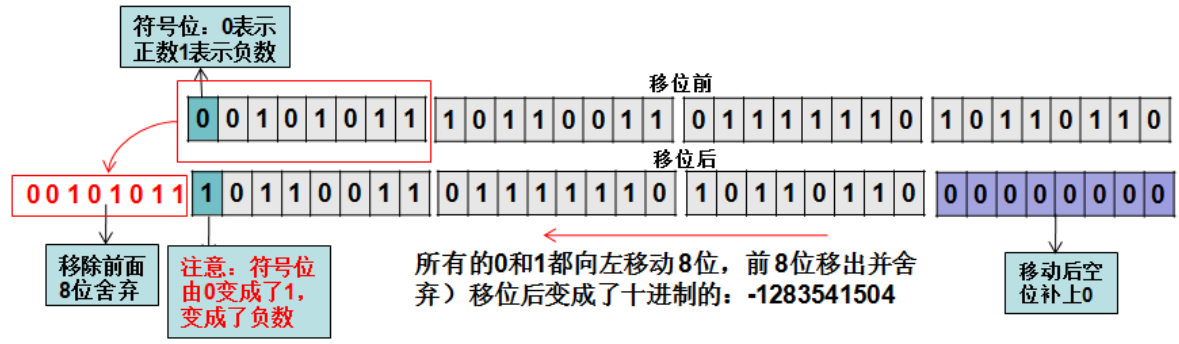
负数：反码是原码除符号位以外的各位求反；补码是原码除符号位以外各位求反之后末位再加1



1）左移运算符 <<



value << 1表示的是变成value的两倍。value = 733183670，进行value<<2运算之后刚好变成了733183670的两倍，有些人在乘2的时候喜欢用左移运算符来替代。



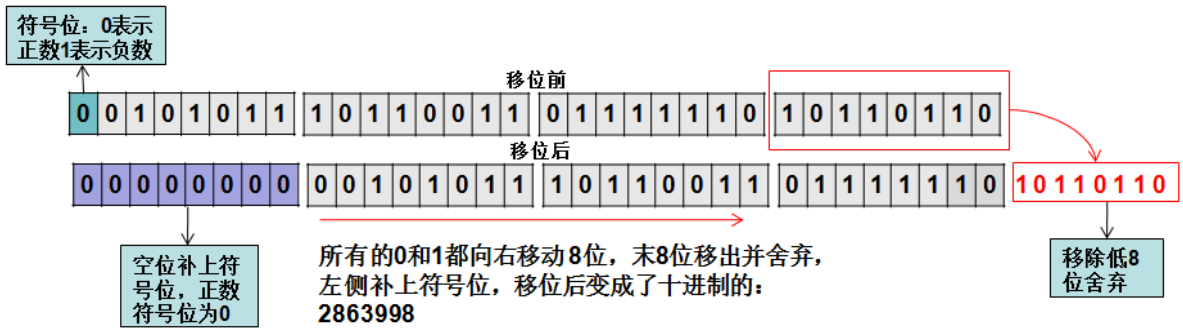
value << 8，左移8位之后，十进制的数变成了-1283541504，移动8位后，由于首位变成了1，也就是说成了负数，在使用中要考虑变成负数的情况。

根据这个规则，左移32位后，右边补上32个0值是不是就变成了十进制的0了？答案是**NO**，当int类型进行左移操作时，**左移位数大于等于32位操作时，会先求余（%）后再进行左移操作**。也就是说左移32位相当于不进行移位操作，左移40位相当于左移8位（40%32=8）。当long类型进行左移操作时，long类型在二进制中的体现是64位的，因此求余操作的基数也变成了64，也就是说左移64位相当于没有移位，左移72位相当于左移8位（72%64=8）。

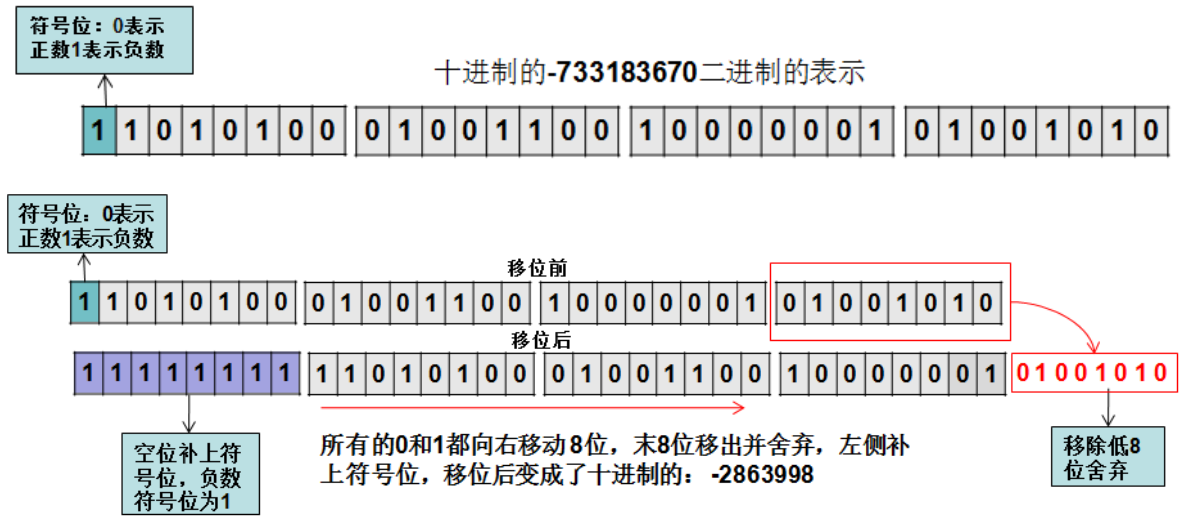
综上所述：左移 << 其实很简单，也就是说**丢弃左边指定位数，右边补0**。

2）右移运算符 >>

右移1位后换算成十进制的值为：366591835，刚好是733183670的1半， 有些人在除2操作时喜欢用右移运算符来替代。



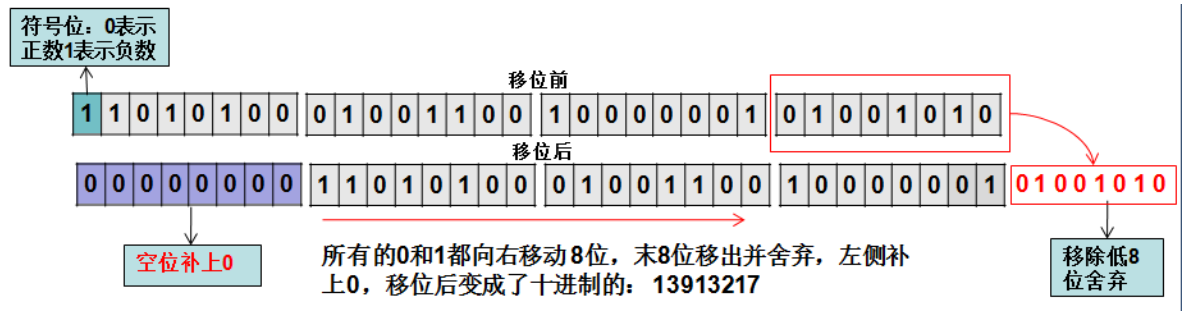
和左移一样，int类型移位大于等于32位时，long类型大于等于64位时，会先做求余处理再位移处理，byte，short移位前会先转换为int类型（32位）再进行移位。以上是正数的位移，我们再来看看负数的右移运算，如图，负数intValue：-733183670的二进制表现如下图：



综上所述：右移运算符>>的运算规则也很简单，丢弃右边指定位数，左边补上符号位。

3）无符号右移运算符：>>>

无符号右移运算符>>>和右移运算符>>是一样的，只不过右移时左边是补上符号位，而无符号右移运算符是补上0，也就是说，对于正数移位来说等同于：>>，负数通过此移位运算符能移位成正数。以-733183670>>>8为例来画一下图



无符号右移运算符>>的运算规则也很简单，丢弃右边指定位数，左边补上0。

4）

五、设计模式

1.了解23种设计模式

2.会使用常用设计模式

单例、策略、工厂、适配器、责任链。

3.实现AOP

4.实现IOC

5.不用synchronized和lock，实现线程安全的单例模式

使用CAS和原子类。

```
1  public class Singleton {
2      private static final AtomicReference<Singleton> INSTANCE = new
   AtomicReference<Singleton>();
3
4      private Singleton() {}
5
6      public static Singleton getInstance() {
7          for (;;) {
8              Singleton singleton = INSTANCE.get();
9              if (null != singleton) {
10                 return singleton;
11             }
12
13             singleton = new Singleton();
14             if (INSTANCE.compareAndSet(null, singleton)) {
```

```
                return singleton;
            }
        }
    }
}
```

6.nio和reactor设计模式

六、网络编程

1.tcp、udp、http、https等常用协议

三次握手与四次关闭、流量控制和拥塞控制、OSI七层模型、tcp粘包与拆包

2.http/1.0 http/1.1 http/2之间的区别

HTTP建立在TCP协议之上，所以HTTP协议的瓶颈及其优化技巧都是基于TCP协议本身的特性，比如TCP的三次握手和四次挥手以及每次建立连接带来的RTT延迟时间。

影响一个HTTP网络请求的因素主要有两个：带宽和延迟：

带宽：网络带宽是指单位时间内传输的数据量，是数据的传输能力。现在网络基础建设较为完善，基本不用担心带宽影响网速，所以目前影响HTTP网络请求性能的就是延迟了。

延迟：

1. 浏览器阻塞（head of line blocking）：浏览器会因为一些原因阻塞请求。浏览器对于同一个域名，同时只能有 6个连接（这个根据浏览器内核不同可能会有所差异），超过浏览器最大连接数限制，后续请求就会被阻塞。这也是为何一些站点会有多个静态资源 CDN 域名的原因之一。
2. DNS查询（DNS Lookup）：将域名解析为IP就是DNS查询，一般使用DNS缓存来减少这个时间。
3. 建立连接（Initial connection）：HTTP 是基于 TCP 协议的，浏览器最快也要在第三次握手时才能携带 HTTP 请求报文，达到真正的建立连接，但是这些连接无法复用会导致每次请求都经历三次握手和慢启动。三次握手在高延迟的场景下影响较明显，慢启动则对文件类大请求影响较大。

HTTP 1.0与HTTP 1.1之间的差别：

1. 缓存处理，在HTTP1.0中主要使用header里的If-Modified-Since,Expires来做为缓存判断的标准，HTTP1.1则引入了更多的缓存控制策略例如Entity tag，If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略。
2. 带宽优化及网络连接的使用，HTTP1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1则在请求头引入了range头域，它允许只请求资源的某个部分，即返回码是206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。
3. 错误通知的管理，在HTTP1.1中新增了24个错误状态响应码，如409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。
4. Host头处理，在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名（hostname）。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们

共享一个IP地址。HTTP1.1的请求消息和响应消息都应支持Host头域，且请求消息中如果没有Host头域会报告一个错误（400 Bad Request）。

5. 长连接，HTTP 1.1支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟，在HTTP1.1中默认开启Connection： keep-alive，一定程度上弥补了HTTP1.0每次请求都要创建连接的缺点。

HTTP2.0和HTTP1.X相比的新特性

- 新的二进制格式（Binary Format），HTTP1.x的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认0和1的组合。基于这种考虑HTTP2.0的协议解析决定采用二进制格式，实现方便且健壮。
- 多路复用（MultiPlexing），即连接共享，即每一个request都是是用作连接共享机制的。一个request对应一个id，这样一个连接上可以有多个request，每个连接的request可以随机的混杂在一起，接收方可以根据request的 id将request再归属到各自不同的服务端请求里面。
- **header**压缩，如上文中所言，对前面提到过HTTP1.x的header带有大量信息，而且每次都要重复发送，HTTP2.0使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减小了需要传输的大小。
- 服务端推送（server push），同SPDY一样，HTTP2.0也具有server push功能。

3.Java RMI，Socket，HttpClient

4.cookie 与 session

cookie被禁用，如何实现session

5.用Java写一个简单的静态文件的HTTP服务器

实现客户端缓存功能，支持返回304 实现可并发下载一个文件 使用线程池处理客户端请求 使用nio处理客户端请求 支持简单的rewrite规则 上述功能在实现的时候需要满足“开闭原则

6.了解nginx和apache服务器的特性并搭建一个对应的服务器

7.用Java实现FTP、SMTP协议

8.进程间通讯的方式

9.什么是CDN？如果实现？

10.什么是DNS?

11.反向代理

七、框架知识

1.Servlet线程安全问题

2.Servlet中的filter和listener

3.Hibernate的缓存机制

4.Hiberate的懒加载

5.Spring Bean的初始化

6.Spring的AOP原理

7.自己实现Spring的IOC

8.Spring MVC

9.Spring Boot2.0

Spring Boot的starter原理，自己实现一个starter

10.Spring Security

八、应用服务器

1.JBoss

2.tomcat

3.jetty

4.Weblogic

九、工具

1.git & svn

2.maven & gradle

高级篇

十、新技术

1.Java 8

lambda表达式、Stream API、

2.Java 9

Jigsaw、Jshell、Reactive Streams

3.Java 10

局部变量类型推断、G1的并行Full GC、ThreadLocal握手机制

4.Spring 5

响应式编程

5.Spring Boot 2.0

十一、性能优化

使用单例、使用Future模式、使用线程池、选择就绪、减少上下文切换、减少锁粒度、数据压缩、结果缓存

十二、线上问题分析

1.dump获取

线程Dump、内存Dump、gc情况

2.dump分析

分析死锁、分析内存泄露

3.自己编写各种outofmemory，stackoverflow程序

HeapOutOfMemory、 Young OutOfMemory、MethodArea OutOfMemory、ConstantPool OutOfMemory、DirectMemory OutOfMemory、Stack OutOfMemory Stack OverFlow

4.常见问题解决思路

内存溢出、线程死锁、类加载冲突

5.使用工具尝试解决以下问题，并写下总结：

1）当一个Java程序响应很慢时如何查找问题、

2）当一个Java程序频繁FullGC时如何解决问题、

3）如何查看垃圾回收日志、

4）当一个Java应用发生OutOfMemory时该如何解决、

5）如何判断是否出现死锁、

6）如何判断是否存在内存泄露

十三、编译原理知识

1.编译与反编译

2.Java代码的编译与反编译

3.Java的反编译工具

4.词法分析，语法分析（LL算法，递归下降算法，LR算法），语义分析，运行时环境，中间代码，代码生成，代码优化

十四、操作系统知识

1.Linux的常用命令

2.进程同步

3.缓冲区溢出

4.分段和分页

5.虚拟内存与主存

十五、数据库知识

1.MySql 执行引擎

2.MySQL 执行计划

如何查看执行计划，如何根据执行计划进行SQL优化

3.SQL优化

4.事务

事务的隔离级别、事务能不能实现锁的功能

5.数据库锁

行锁、表锁、使用数据库锁实现乐观锁、

6.数据库主备搭建

7.binlog

8.内存数据库

h2

9.常用的nosql数据库

redis、memcached

10.分别使用数据库锁、NoSql实现分布式锁

11.性能调优

十六、数据结构与算法知识

1.简单的数据结构

栈、队列、链表、数组、哈希表、

2.树

二叉树、字典树、平衡树、排序树、B树、B+树、R树、多路树、红黑树

1）二叉树和红黑树

3.排序算法

各种排序算法和时间复杂度 深度优先和广度优先搜索 全排列、贪心算法、KMP算法、hash算法、海量数据处理

十七、大数据知识

1.Zookeeper

基本概念、常见用法

2.Solr，Lucene，ElasticSearch

在linux上部署solr，solrcloud，，新增、删除、查询索引

3.Storm，流式计算，了解Spark，S4

在linux上部署storm，用zookeeper做协调，运行storm hello world，local和remote模式运行调试storm topology。

4.Hadoop，离线计算

HDFS、MapReduce

5.分布式日志收集flume，kafka，logstash

6.数据挖掘，mahout

十八、网络安全知识

1.什么是XSS

XSS的防御

2.什么是CSRF

3.什么是注入攻击

SQL注入、XML注入、CRLF注入

4.什么是文件上传漏洞

5.加密与解密

MD5，SHA1、DES、AES、RSA、DSA

6.什么是DOS攻击和DDOS攻击

memcached为什么可以导致DDos攻击、什么是反射型DDoS

7.SSL、TLS，HTTPS

8.如何通过Hash碰撞进行DOS攻击

9.用openssl签一个证书部署到apache或nginx

架构篇

十九、分布式

1.数据一致性、服务治理、服务降级

2.分布式事务

2PC、3PC、CAP、BASE、 可靠消息最终一致性、最大努力通知、TCC

3.Dubbo

服务注册、服务发现，服务治理

4.分布式数据库

怎样打造一个分布式数据库、什么时候需要分布式数据库、mycat、otter、HBase

5.分布式文件系统

mfs、fastdfs

6.分布式缓存

缓存一致性、缓存命中率、缓存冗余

二十、微服务

1.SOA、康威定律

2.ServiceMesh

3.Docker & Kubernets

4.Spring Boot

5.Spring Cloud

二十一、高并发

1.分库分表

2.CDN技术

3.消息队列

4.ActiveMQ

二十二、监控

1.监控什么

CPU、内存、磁盘I/O、网络I/O等

2.监控手段

进程监控、语义监控、机器资源监控、数据波动

3.监控数据采集

日志、埋点

4.Dapper

二十三、负载均衡

1.tomcat负载均衡

2.Nginx负载均衡

二十四、DNS

1.DNS原理

2.DNS的设计

二十五、CDN

1.数据一致性

扩展篇

二十六、云计算

IaaS、SaaS、PaaS、虚拟化技术、openstack、Serverlsess

二十七、搜索引擎

Solr、Lucene、Nutch、Elasticsearch

二十八、权限管理

Shiro

二十九、区块链

哈希算法、Merkle树、公钥密码算法、共识算法、Raft协议、Paxos 算法与 Raft 算法、拜占庭问题与算法、消息认证码与数字签名

比特币：挖矿、共识机制、闪电网络、侧链、热点问题、分叉

以太坊

超级账本

三十、人工智能

数学基础、机器学习、人工神经网络、深度学习、应用场景。

常用框架：TensorFlow、DeepLearning4J

三十一、其他语言

Groovy、Python、Go、NodeJs、Swift、Rust

推荐书籍

《深入理解Java虚拟机》

《Effective Java》

《深入分析Java Web技术内幕》

《大型网站技术架构》

《代码整洁之道》

《Head First设计模式》

《maven实战》

《区块链原理、设计与应用》

《Java并发编程实战》

《鸟哥的Linux私房菜》

《从Paxos到Zookeeper》

《架构即未来》