Java 面试知识点解析

（一）Java 基础知识点

1）面向对象的特性有哪些？

答：封装、继承和多态（*应要多算一个那就是抽象*）

- **封装是指将对象的实现细节隐藏起来，然后通过公共的方法来向外暴露出该对象的功能。**
  但封装不仅仅是 private + getter/setter ，使用封装可以对 setter 进行更深层次的定制，例如你可以对执行方法的对象做规定，也可以对数据做一定的要求，还可以做类型转换等等。**使用封装不仅仅安全，更可以简化操作。**（封装扩展阅读：oc 面向对象三大特性之一〈封装〉）
- **继承是面向对象实现软件复用的重要手段，当子类继承父类后，子类是一种特殊的父类，能直接或间接获得父类里的成员。**
  继承的缺点：1）继承是一种强耦合关系，父类变子类也必须变；2）继承破坏了封装，对于父类而言，它的实现细节对子类来说都是透明的。
- **多态简而言之就是同一个行为具有多个不同表现形式或形态的能力。**
  比如说，有一杯水，我不知道它是温的、冰的还是烫的，但是我一摸我就知道了，我摸水杯的这个动作，对于不同温度的水，就会得到不同的结果，这就是多态。
  多态的条件：1）继承；2）重写；3）向上转型。
  多态的好处：当把不同的子类对象都当作父类类型来看，**可以屏蔽不同子类对象之间的实现差异，**从而写出通用的代码达到通用编程，以适应需求的不断变化。（多态扩展阅读：重新认识 java（五） ---- 面向对象之多态（向上转型与向下转型））
- **抽象是指从特定的角度出发，从已经存在的一些事物中抽取我们所关注的特性、行为，从而形成一个新的事物的思维过程，是一种从复杂到简洁的思维方式。**

2）面向对象和面向过程的区别？

答：面向过程是**一种站在过程的角度思考问题的思想，强调的是功能行为，功能的执行过程**，即先干啥，后干啥。

面向过程的设计：最小的程序单元是函数，每个函数负责完成某一个功能，用以接受输入数据，函数对输入数据进行处理，然后输出结果数据。整个软件系

统由一个个的函数组成，其中作为程序入口的函数称之为主函数，主函数依次调用其他函数，普通函数之间可以相互调用，从而实现整个系统功能。

- **面向过程的缺陷：**
  向过程的设计,是**采用置顶而下的设计方式**，在设计阶段就需要考虑每一个模块应该分解成哪些子模块，每一个子模块有细分为更小的子模块，如此类推，直到将模块细化为一个个函数。
- **问题：**1）设计不够直观，与人类的习惯思维不一致；2）系统软件适应性差，可扩展性差，维护性低。

面向过程最大的问题在于随着系统的膨胀，面向过程将无法应付，最终导致系统的崩溃。为了解决这一种软件危机，我们提出**面向对象**思想。

面向对象是一种基于面向过程的新的编程思想，是**一种站在对象的角度思考问题**的思想，我们把多个功能合理的放到不同对象里，**强调的是具备某些功能的对象。**

- 面向对象更加符合我们常规的思维方式，稳定性好，可重用性强，易于开发大型软件产品，有良好的可维护性。在软件工程上，面向对象可以使工程更加模块化，实现更低的耦合和更高的内聚。
- **注意：** 不要粗浅的认为面向对象一定就优于面向过程的设计

看到知乎上有一句有意思的话：

你的程序要完成一个任务，相当于讲一个故事。

面向过程：编年体；
面向对象：纪传体。

而对于复杂的程序/宏大的故事，事实都证明了，面向对象/纪传是更合理的表述方法。

扩展阅读：面向过程 VS 面向对象

3）JDK 和 JRE 的区别是什么？

解析：这是考察一些基本的概念

答：Java 运行时环境（JRE-Java Runtime Environment），它包括 Java 虚拟机、Java 核心类库和支持文件，但并不包含开发工具（JDK-Java Development Kit）——编译器、调试器和其他工具。

Java 开发工具包（JDK）是完整的 Java 软件开发包，包含了 JRE，编译器和其他的工具（比如 JavaDoc， Java 调试器），可以让开发者开发、编译、执行 Java 应用程序。

- 还有其他的一些名词也可以再看一下：

4）Java 中覆盖和重载是什么意思？

解析：覆盖和重载是比较重要的基础知识点，并且容易混淆，所以面试中常见。

答：**覆盖（Override）是指子类对父类方法的一种重写，**只能比父类抛出更少的异常，访问权限不能比父类的小，被覆盖的方法不能是 private 的，否则只是在子类中重新定义了一个新方法。

重载（Overload）表示同一个类中可以有多个名称相同的方法，但这些方法的参数列表各不相同。

面试官： 那么构成重载的条件有哪些？

答：参数类型不同、参数个数不同、参数顺序不同。

面试官： 函数的返回值不同可以构成重载吗？为什么？

答：不可以，因为 Java 中调用函数并不需要强制赋值。举例如下：

如下两个方法：

```
void f(){}
int f(){ return 1; }
```

只要编译器可以根据语境明确判断出语义，比如在 int x = f(); 中，那么的确可以据此区分重载方法。不过， 有时你并不关心方法的返回值，你想要的是方法调用的其他效果 （这常被称为 “为了副作用而调用” ），这时你可能会调用方法而忽略其返回值，所以如果像下面的调用：

```
f();
```

此时 Java 如何才能判断调用的是哪一个 f() 呢？别人如何理解这种代码呢？所以，根据方法返回值来区分重载方法是行不通的。

5）抽象类和接口的区别有哪些？

答：

1. 抽象类中可以没有抽象方法；接口中的方法必须是抽象方法；
2. 抽象类中可以有普通的成员变量；接口中的变量必须是 static final 类型的，必须被初始化,接口中只有常量，没有变量。
3. 抽象类只能单继承，接口可以继承多个父接口；
4. Java 8 中接口中会有 default 方法，即方法可以被实现。

面试官：抽象类和接口如何选择？

答：

1. 如果要创建不带任何方法定义和成员变量的基类，那么就应该选择接口而不是抽象类。
2. 如果知道某个类应该是基类，那么第一个选择的应该是让它成为一个接口，只有在必须要有方法定义和成员变量的时候，才应该选择抽象类。因为抽象类中允许存在一个或多个被具体实现的方法，只要方法没有被全部实现该类就仍是抽象类。

6）Java 和 C++ 的区别：

解析：虽然我们不太懂 C++，但是就是会这么问，尤其是三面（总监级别）面试中。

答：

1. 都是面向对象的语言，都支持封装、继承和多态
2. 指针：Java 不提供指针来直接访问内存，程序更加安全
3. 继承： Java 的类是单继承的，C++支持多重继承；Java 通过一个类实现多个接口来实现 C++中的多重继承； Java 中类不可以多继承，但是！！！接口可以多继承
4. 内存： Java 有自动内存管理机制，不需要程序员手动释放无用内存

7）“static”关键字是什么意思？

答：“static” 关键字表明一个成员变量或者是成员方法可以在没有所属的类的实例变量的情况下被访问。

面试官：Java 中是否可以覆盖(override)一个 private 或者是 static 的方法？

答：Java 中 static 方法不能被覆盖，因为方法覆盖是基于运行时动态绑定的，而 static 方法是编译时静态绑定的。static 方法跟类的任何实例都不相关，所以概念上不适用。

Java 中也不可以覆盖 private 的方法，因为 private 修饰的变量和方法只能在当前类中使用，如果是其他的类继承当前类是不能访问到 private 变量或方法的，当然也不能覆盖。

扩展阅读：重新认识 java（六） ---- java 中的另类：static 关键字（附代码块知识）

8）Java 是值传递还是引用传递？

解析：这类题目，面试官会手写一个例子，让你说出函数执行结果。

答：值传递是对基本型变量而言的,传递的是该变量的一个副本,改变副本不影响原变量。引用传递一般是对于对象型变量而言的,传递的是该对象地址的一个副本，并不是原对象本身 。

一般认为，Java 内的传递都是值传递.，Java 中实例对象的传递是引用传递，Java 是值传递的！

- 我们先来看一个例子：

这是一个很经典的例子，我们希望在调用了 swap() 方法之后交换 arg1 和 arg2 的值，但事实上并没有，为什么会这样？

这就是因为 Java 是值传递的，也就是说，我们在调用一个需要传递参数的函数时，传递给函数的参数并不是我们传递进去的参数本身，而是它的一个副本，我们改变了数据其实只是改变了副本的数据而已，并不会对原来的参数有任何的改变。

- 再来看一个例子：

我们自己定义了一个内部类 Person ，该类只有一个 int 类型的 age 属性，然后有 getter/setter ，我们希望通过 changeAge() 函数来改变 Person 对象的 age 属性，为什么这次成功了呢？

你依然可以理解为，主函数将 person 复制了一份到 changeAge 函数中去，最终还是只改变了 changeAge 中复制的那一份参数的值，而原本的参数并没有改变，但 changeAge 中的那一份和原本的参数指向了同一个内存区域！

9）JDK 中常用的包有哪些？

答：java.lang、java.util、java.io、java.net、java.sql。

10）JDK, JRE 和 JVM 的联系和区别？

答：JDK 是 Java 开发工具包，是 Java 开发环境的核心组件，并提供编译、调试和运行一个 Java 程序所需要的所有工具，可执行文件和二进制文件，是一个平台特定的软件。

JRE 是 Java 运行时环境，是 JVM 的实施实现，提供了运行 Java 程序的平台。JRE 包含了 JVM，但是不包含 Java 编译器 / 调试器之类的开发工具。

JVM 是 Java 虚拟机，当我们运行一个程序时，JVM 负责将字节码转换为特定机器代码，JVM 提供了内存管理 / 垃圾回收和安全机制等。

这种独立于硬件和操作系统，正是 Java 程序可以一次编写多处执行的原因。

区别：

1. JDK 用于开发，JRE 用于运行 Java 程序；
2. JDK 和 JRE 中都包含 JVM；
3. JVM 是 Java 编程语言的核心并且具有平台独立性。

11）Integer 的缓存机制

解析：考察的是对源码的熟悉程度

- 看一个例子：

第一个返回 true 很好理解，就像上面讲的，a 和 b 指向相同的地址。

第二个返回 false 是为什么呢？这是因为 Integer 有缓存机制，在 JVM 启动初期就缓存了 -128 到 127 这个区间内的所有数字。

第三个返回 false 是因为用了 new 关键字来开辟了新的空间，i 和 j 两个对象分别指向堆区中的两块内存空间。

我们可以跟踪一下 Integer 的源码，看看到底怎么回事。在 IDEA 中，你只需要按住 Ctrl 然后点击 Integer，就会自动进入 jar 包中对应的类文件。

跟踪到文件的 700 多行，你会看到这么一段，感兴趣可以仔细读一下，不用去读也没有关系，因为你只需要知道这是 Java 的一个缓存机制。Integer 类的内部类缓存了 -128 到 127 的所有数字。（事实上，Integer 类的缓存上限是可以通过修改系统来更改的，了解就行了，不必去深究。）

12）下述两种方法分别创建了几个 Sring 对象？

```
// 第一种：直接赋一个字面量
String str1 = "ABCD";
// 第二种：通过构造器创建
String str2 = new String("ABCD");
```

解析：考察的是对 String 对象和 JVM 内存划分的知识。

答：String str1 = "ABCD";最多创建一个 String 对象,最少不创建 String 对象.如果常量池中,存在”ABCD”,那么 str1 直接引用,此时不创建 String 对象.否则,先在常量池先创建”ABCD”内存空间,再引用.

String str2 = new String("ABCD");最多创建两个 String 对象，至少创建一个 String 对象。new 关键字绝对会在堆空间创建一块新的内存区域，所以至少创建一个 String 对象。

我们来看图理解一下：

- 当执行第一句话的时候，会在常量池中添加一个新的 ABCD 字符，str1 指向常量池的 ABCD

- 当执行第二句话的时候，因为有 new 操作符，所以会在堆空间新开辟一块空间用来存储新的 String 对象，因为此时常量池中已经有了 ABCD 字符，所以堆中的 String 对象指向常量池中的 ABCD，而 str2 则指向堆空间中的 String 对象。

String 对象是一个特殊的存在，需要注意的知识点也比较多，这里给一个之前写的 String 详解的文章链接：传送门 其中包含的问题大概有：**1）“+” 怎么连接字符串；2）字符串的比较；3）StringBuilder/StringBuffer/String 的区别；**

13）i++ 与 ++i 到底有什么不同？

解析：对于这两个的区别，熟悉的表述是：前置++是先将变量的值加 1，然后使用加 1 后的值参与运算，而后置++则是先使用该值参与运算，然后再将该值加 1 .但事实上，**前置++和后置++一样，在参与运算之前都会将变量的值加 1**

答：实际上，不管是前置 ++，还是后置 ++，都是先将变量的值加 1，然后才继续计算的。**二者之间真正的区别是：前置 ++ 是将变量的值加 1 后，使用增值后的变量进行运算的，而后置 ++ 是首先将变量赋值给一个临时变量，接下来对变量的值加 1，然后使用那个临时变量进行运算。**

14）交换变量的三种方式

答：

- 第一种：通过第三个变量

```
public class Test{
    public static void main(String[] args) {
        int x = 5;
        int y = 10;
        swap(x,y);
        System.out.println(x);
        System.out.println(y);

        Value v = new Value(5,10);
        swap(v);
        System.out.println(v.x);
        System.out.println(v.y);
    }
```

```
    // 无效的交换：形参的改变无法反作用于实参
    public static void swap(int x,int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    // 有效的交换：通过引用（变量指向一个对象）来修改成员变量
    public static void swap(Value value) {
        int temp = value.x;
        value.x = value.y;
        value.y = temp;
    }
}

class Value{
    int x;
    int y;

    public Value(int x,int y) {
        this.x = x;
        this.y = y;
    }
}
```

输出的结果：

```
5
10
10
5
```

这有点类似于 C/C++语言中的指针，不过相对来说更加安全。

事实上，其实如果把基础类型 int 改成对应的包装类的话其实可以更加简单的完成这个操作，不过需要付出更多的内存代价。

第二种：通过通过相加的方式（相同的 Value 类不再重复展示）

```
public class Test{
    public static void main(String[] args) {
        Value v1 = new Value(5,10);
        swap(v1);
        System.out.println("v1 交换之后的结果为：");
        System.out.println(v1.x);
        System.out.println(v1.y);
```

```
    }

    public static void swap(Value v) {
        v.x = v.x + v.y;
        v.y = v.x - v.y;
        v.x = v.x - v.y;
    }
}
```

输出的结果：
v1 的交换结果：
10
5

核心的算法就是 swap 方法:

```
v.x = v.x + v.y;     // 把 v.x 与 v.y 的和存储在 v.x 中
v.y = v.x - v.y;     // v.x 减掉 v.y 本来的值即为 v.x
v.x = v.x - v.y;     // v.x 减掉 v.y 的值也就是以前 x.y 的值
```

这样就可以不通过临时变量，来达到交换两个变量的目的，如果觉得上面的方法不太容易理解，我们也可以用另一个参数 z 来表示上述过程：

```
int z = v.x + v.y;     // 把 v.x 与 v.y 的和存储在 z 中
v.y = z - v.y;         // z 减掉以前的 v.y 就等于 v.x
v.x = z - v.y;         // z 减掉现在的 v.y 即以前的 v.x，即为 v.y
```

但并**不推荐这种做法**，原因在于当数值很大的时候，16 进制的求和运算可能造成数据的溢出，虽然最后的结果依然会是我们所期望的那样，但仍然不是十分可取。

- 第三种：通过异或的方式：

位异或运算符（^）有这样的一个性质，就是两个整型的数据 x 与 y，有：（x ^ y ^ y） == x 这说明，如果一个变量 x 异或另外一个变量 y 两次，结果为 x。通过这一点，可以实现交换两个变量的值：

```
public class Test{
    public static void main(String[] args) {
        Value v1 = new Value(5,10);
        swap(v1);
        System.out.println("v1 交换之后的结果为：");
        System.out.println(v1.x);
        System.out.println(v1.y);
    }
```

```
    public static void swap(Value v) {
        v.x = v.x ^ v.y;
        v.y = v.x ^ v.y;
        v.x = v.x ^ v.y;
    }
}
```

输出的结果：
v1 交换之后的结果为：
10
5

跟上面相加的方式过程几乎类似，只不过运算的方式不同而已。**异或的方法比相加更加可取的地方在于，异或不存在数据溢出。**

15）Java 对象初始化顺序？

答：不考虑静态成员的初始化，调用一个对象的构造函数时，程序**先调用父类的构造函数**（可以通过 super 关键字指定父类的构造函数，否则默认调用无参的构造函数，并且需要在子类的构造函数的第一行调用），**之后静态成员变量的初始化函数和静态初始化块则按照在代码当中的顺序执行**，成员变量如果没有指定值的话则赋予默认值，即基本数据类型为 0 或 false 等，对象则为 null；**最后调用自身构造函数。**

- 我们可以写一段程序来对初始化顺序进行一个简单的验证：

```
public class Derive extends Base
{
    private Member m1 = new Member("Member 1");
    {
        System.out.println("Initial Block()");
    }

    public Derive() {
        System.out.println("Derive()");
    }

    private Member m2 = new Member("Member 2");
    private int i = getInt();

    private int getInt()
    {
        System.out.println("getInt()");
```

```
        return 2;
    }

    public static void main(String[] args)
    {
        new Derive();
    }
}

class Base
{
    public Base()
    {
        System.out.println("Base()");
    }
}

class Member
{
    public Member(String m)
    {
        System.out.println("Member() "+m);
    }
}
```

程序的输出结果是：

```
Base()
Member() Member 1
Initial Block()
Member() Member 2
getInt()
Derive()
```

16）true、false 与 null 是关键字吗？

答：不是。true、false 是布尔类型的字面常量，null 是引用类型的字面常量。

面试官：那 goto 与 const 呢？

答：是。goto 与 const 均是 Java 语言保留的关键字，即没有任何语法应用。

17）exception 和 error 有什么区别？

答：exception 和 error 都是 Throwable 的子类。exception 用于用户程序可以捕获的异常情况；error 定义了不期望被用户程序捕获的异常。

exception 表示一种设计或设计的问题，也就是说只要程序正常运行，从不会发生的情况；而 error 表示回复不是不可能但是很困难的情况下的一种严重问题，比如内存溢出，不可能指望程序处理这样的情况。

18）throw 和 throws 有什么区别？

答：throw 关键字用来在程序中明确的抛出异常，相反，throws 语句用来表明方法不能处理的异常。每一个方法都必须要指定哪些异常不能处理，所以方法的调用者才能够确保处理可能发生的异常，多个异常是用逗号分隔的。

小结：本节主要阐述了 Java 基础知识，并没有涉及到一些高级的特性，这些问题一般难度不大，适当复习下，应该没问题。

（二）Java 中常见集合

集合这方面的考察相当多，这部分是面试中必考的知识点。

1）说说常见的集合有哪些吧？

答：Map 接口和 Collection 接口是所有集合框架的父接口：

1. Collection 接口的子接口包括：Set 接口和 List 接口
2. Map 接口的实现类主要有：HashMap、TreeMap、Hashtable、ConcurrentHashMap 以及 Properties 等
3. Set 接口的实现类主要有：HashSet、TreeSet、LinkedHashSet 等
4. List 接口的实现类主要有：ArrayList、LinkedList、Stack 以及 Vector 等

2）HashMap 和 Hashtable 的区别有哪些？（必问）

答：

1. HashMap 没有考虑同步，是线程不安全的；Hashtable 使用了 synchronized 关键字，是线程安全的；
2. 前者允许 null 作为 Key；后者不允许 null 作为 Key

3）HashMap 的底层实现你知道吗？

答：在 Java8 之前，其底层实现是数组+链表实现，Java8 使用了数组+链表+红黑树实现。此时你可以简单的在纸上画图分析：

4）ConcurrentHashMap 和 Hashtable 的区别？（必问）

答：ConcurrentHashMap 结合了 HashMap 和 HashTable 二者的优势。HashMap 没有考虑同步，HashTable 考虑了同步的问题。但是 HashTable 在每次同步执行时都要锁住整个结构。 ConcurrentHashMap 锁的方式是稍微细粒度的。 ConcurrentHashMap 将 hash 表分为 16 个桶（默认值），诸如 get,put,remove 等常用操作只锁当前需要用到的桶。

面试官：ConcurrentHashMap 的具体实现知道吗？

答：

1. 该类包含两个静态内部类 HashEntry 和 Segment ；前者用来封装映射表的键值对，后者用来充当锁的角色；

2. Segment 是一种可重入的锁 ReentrantLock，每个 Segment 守护一个 HashEntry 数组里得元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment 锁。

5）HashMap 的长度为什么是 2 的幂次方？

答：

1. 通过将 Key 的 hash 值与 length - 1 进行 & 运算，实现了当前 Key 的定位，2 的幂次方可以减少冲突（碰撞）的次数，提高 HashMap 查询效率

2. 如果 length 为 2 的次幂 则 length-1 转化为二进制必定是 11111…… 的形式，在于 h 的二进制与操作效率会非常的快，而且空间不浪费；如果 length 不是 2 的次幂，比如 length 为 15，则 length - 1 为 14，对应的二进制为 1110，在于 h 与操作，最后一位都为 0 ，而 0001，0011，0101，1001，1011，0111，1101 这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率！这样就会造成空间的浪费。

6）List 和 Set 的区别是啥？

答：List 元素是有序的，可以重复；Set 元素是无序的，不可以重复。

7）List、Set 和 Map 的初始容量和加载因子：

答：

1. List

- ArrayList 的初始容量是 10；加载因子为 0.5； 扩容增量：原容量的 0.5 倍+1；一次扩容后长度为 15。
- Vector 初始容量为 10，加载因子是 1。扩容增量：原容量的 1 倍，如 Vector 的容量为 10，一次扩容后是容量为 20。

2. Set

HashSet，初始容量为 16，加载因子为 0.75； 扩容增量：原容量的 1 倍；如 HashSet 的容量为 16，一次扩容后容量为 32

3. Map

HashMap，初始容量 16，加载因子为 0.75； 扩容增量：原容量的 1 倍； 如 HashMap 的容量为 16，一次扩容后容量为 32

8）Comparable 接口和 Comparator 接口有什么区别？

答：

1. 前者简单，但是如果需要重新定义比较类型时，需要修改源代码。

2. 后者不需要修改源代码，自定义一个比较器，实现自定义的比较方法。 具体解析参考博客：Java 集合框架—Set

9）Java 集合的快速失败机制 “fail-fast”

答：

是 java 集合的一种错误检测机制，当多个线程对集合进行结构上的改变的操作时，有可能会产生 fail-fast 机制。

例如：假设存在两个线程（线程 1、线程 2），线程 1 通过 Iterator 在遍历集合 A 中的元素，在某个时候线程 2 修改了集合 A 的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出 ConcurrentModificationException 异常，从而产生 fail-fast 机制。

原因：迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 modCount 变量。集合在被遍历期间如果内容发生变化，就会改变 modCount 的值。每当迭代器使用 hashNext()/next()遍历下一个元素之前，都会检测 modCount 变量是否为 expectedmodCount 值，是的话就返回遍历；否则抛出异常，终止遍历。

解决办法：

1. 在遍历过程中，所有涉及到改变 modCount 值得地方全部加上 synchronized。

2. 使用 CopyOnWriteArrayList 来替换 ArrayList

10）ArrayList 和 Vector 的区别

答：

这两个类都实现了 List 接口（List 接口继承了 Collection 接口），他们都是有序集合，即存储在这两个集合中的元素位置都是有顺序的，相当于一种动态的数组，我们以后可以按位置索引来取出某个元素，并且其中的数据是允许重复的，这是与 HashSet 之类的集合的最大不同处，HashSet 之类的集合不可以按索引号去检索其中的元素，也不允许有重复的元素。

ArrayList 与 Vector 的区别主要包括两个方面：

1. 同步性：
   Vector 是线程安全的，也就是说它的方法之间是线程同步（加了 synchronized 关键字）的，而 ArrayList 是线程不安全的，它的方法之间是线程不同步的。如果只有一个线程会访问到集合，那最好是使用 ArrayList，因为它不考虑线程安全的问题，所以效率会高一些；如果有多个线程会访问到集合，那最好是使用 Vector，因为不需要我们自己再去考虑和编写线程安全的代码。
2. 数据增长：
   ArrayList 与 Vector 都有一个初始的容量大小，当存储进它们里面的元素的个人超过了容量时，就需要增加 ArrayList 和 Vector 的存储空间，每次要增加存储空间时，不是只增加一个存储单元，而是增加多个存储单元，每次增加的存储单元的个数在内存空间利用与程序效率之间要去的一定的平衡。Vector 在数据满时（加载因子 1）增长为原来的两

倍（扩容增量：原容量的 1 倍），而 ArrayList 在数据量达到容量的一半时（加载因子 0.5）增长为原容量的 0.5 倍 + 1 个空间。

面试官：那 ArrayList 和 LinkedList 的区别呢？

答：

1. LinkedList 实现了 List 和 Deque 接口，一般称为双向链表；
2. LinkedList 在插入和删除数据时效率更高，ArrayList 在查找某个 index 的数据时效率更高；
3. LinkedList 比 ArrayList 需要更多的内存；

面试官：Array 和 ArrayList 有什么区别？什么时候该应 Array 而不是 ArrayList 呢？

答：它们的区别是：

1. Array 可以包含基本类型和对象类型，ArrayList 只能包含对象类型。
2. Array 大小是固定的，ArrayList 的大小是动态变化的。
3. ArrayList 提供了更多的方法和特性，比如：addAll()，removeAll()，iterator() 等等。

对于基本类型数据，集合使用自动装箱来减少编码工作量。但是，当处理固定大小的基本数据类型的时候，这种方式相对比较慢。

11）如何去掉一个 Vector 集合中重复的元素？

答：

```
Vector newVector = new Vector();
for (int i = 0; i < vector.size(); i++) {
    Object obj = vector.get(i);
    if (!newVector.contains(obj)) {
        newVector.add(obj);
    }
}
```

还有一种简单的方式，利用了 Set 不允许重复元素的特性：

```
HashSet set = new HashSet(vector);
```

小结：本小节是 Java 中关于集合的考察，是 Java 岗位面试中必考的知识点，除了应该掌握以上的问题，包括各个集合的底层实现也建议各位同学阅读，加深理解。

12）如何权衡是使用无序的数组还是有序的数组？

答：有序数组最大的好处在于查找的时间复杂度是 O(log n)，而无序数组是 O(n)。有序数组的缺点是插入操作的时间复杂度是 O(n)，因为值大的元素需要往后移动来给新元素腾位置。相反，无序数组的插入时间复杂度是常量 O(1)。

总结

oh.....复习下来还真是酸爽....前路漫漫啊...

(二)——高并发编程

（一）高并发编程基础知识

这里涉及到一些基础的概念，我重新捧起了一下《实战 Java 高并发程序设计》这一本书，感觉到心潮澎湃，这或许就是笔者叙述功底扎实的魅力吧，喜欢。对于并发的基础可以参照一下我之前写过的一篇博文：[Java 学习笔记（4）——并发基础]

1）多线程和单线程的区别和联系？

答：

1. 在单核 CPU 中，将 CPU 分为很小的时间片，在每一时刻只能有一个线程在执行，是一种微观上轮流占用 CPU 的机制。
2. 多线程会存在线程上下文切换，会导致程序执行速度变慢，即采用一个拥有两个线程的进程执行所需要的时间比一个线程的进程执行两次所需要的时间要多一些。

结论：即采用多线程不会提高程序的执行速度，反而会降低速度，但是对于用户来说，可以减少用户的响应时间。

面试官：那使用多线程有什么优势？

解析：尽管面临很多挑战，多线程有一些优点仍然使得它一直被使用，而这些优点我们应该了解。

答：

（1）资源利用率更好

想象一下，一个应用程序需要从本地文件系统中读取和处理文件的情景。比方说，从磁盘读取一个文件需要 5 秒，处理一个文件需要 2 秒。处理两个文件则需要：

```
1| 5 秒读取文件 A
2| 2 秒处理文件 A
3| 5 秒读取文件 B
4| 2 秒处理文件 B
5| ---------------------
6| 总共需要 14 秒
```

从磁盘中读取文件的时候，大部分的 CPU 时间用于等待磁盘去读取数据。在这段时间里，CPU 非常的空闲。它可以做一些别的事情。通过改变操作的顺序，就能够更好的使用 CPU 资源。看下面的顺序：

```
1| 5 秒读取文件 A
2| 5 秒读取文件 B + 2 秒处理文件 A
3| 2 秒处理文件 B
4| ---------------------
5| 总共需要 12 秒
```

CPU 等待第一个文件被读取完。然后开始读取第二个文件。当第二文件在被读取的时候，CPU 会去处理第一个文件。记住，在等待磁盘读取文件的时候，CPU 大部分时间是空闲的。

总的说来，CPU 能够在等待 IO 的时候做一些其他的事情。这个不一定就是磁盘 IO。它也可以是网络的 IO，或者用户输入。通常情况下，网络和磁盘的 IO 比 CPU 和内存的 IO 慢的多。

（2）程序设计在某些情况下更简单

在单线程应用程序中，如果你想编写程序手动处理上面所提到的读取和处理的顺序，你必须记录每个文件读取和处理的状态。相反，你可以启动两个线程，每个线程处理一个文件的读取和操作。线程会在等待磁盘读取文件的过程中被阻塞。在等待的时候，其他的线程能够使用 CPU 去处理已经读取完的文件。其结果就是，磁盘总是在繁忙地读取不同的文件到内存中。这会带来磁盘和 CPU 利用率的提升。而且每个线程只需要记录一个文件，因此这种方式也很容易编程实现。

（3）程序响应更快

有时我们会编写一些较为复杂的代码（这里的复杂不是说复杂的算法，而是复杂的业务逻辑），例如，一笔订单的创建，它包括插入订单数据、生成订单赶快找、发送邮件通知卖家和记录货品销售数量等。用户从单击“订购”按钮开始，就要等待这些操作全部完成才能看到订购成功的结果。但是这么多业务操作，如何能够让其更快地完成呢？

在上面的场景中，可以使用多线程技术，即将数据一致性不强的操作派发给其他线程处理（也可以使用消息队列），如生成订单快照、发送邮件等。这样做的好处是响应用户请求的线程能够尽可能快地处理完成，缩短了响应时间，提升了用户体验。

多线程还有一些优势也显而易见：
① 进程之前不能共享内存，而线程之间共享内存(堆内存)则很简单。
② 系统创建进程时需要为该进程重新分配系统资源,创建线程则代价小很多,因此实现多任务并发时,多线程效率更高.
③ Java 语言本身内置多线程功能的支持,而不是单纯第作为底层系统的调度方式,从而简化了多线程编程.

2) 多线程一定快吗？

答：不一定。

比如，我们尝试使用并行和串行来分别执行累加的操作观察是否并行执行一定比串行执行更快：

错误!未指定文件名。

以下是我测试的结果，可以看出，当不超过 1 百万的时候，并行是明显比串行要慢的，为什么并发执行的速度会比串行慢呢？这是因为线程有创建和上下文切换的开销。

3) 什么是同步？什么又是异步？

解析：这是对多线程基础知识的考察

答：同步和异步通常用来形容一次方法调用。

同步方法调用一旦开始，调用者必须等到方法返回后，才能继续后续的行为。这就好像是我们去商城买一台空调，你看中了一台空调，于是就跟售货员下了

单，然后售货员就去仓库帮你调配物品，这天你热的实在不行，就催着商家赶紧发货，于是你就在商店里等着，知道商家把你和空调都送回家，一次愉快的购物才结束，这就是同步调用。

而异步方法更像是一个消息传递，一旦开始，方法调用就会立即返回，调用者就可以继续后续的操作。回到刚才买空调的例子，我们可以坐在里打开电脑，在网上订购一台空调。当你完成网上支付的时候，对你来说购物过程已经结束了。虽然空调还没有送到家，但是你的任务都已经完成了。商家接到你的订单后，就会加紧安排送货，当然这一切已经跟你无关了，你已经支付完成，想什么就能去干什么了，出去溜达几圈都不成问题。等送货上门的时候，接到商家电话，回家一趟签收即可。这就是异步调用。

面试官：那并发（Concurrency）和并行（Parallelism）的区别呢？

解析：并行性和并发性是既相似又有区别的两个概念。

答：并行性是指两个或多个事件在同一时刻发生。而并发性是指连个或多个事件在同一时间间隔内发生。

在多道程序环境下，并发性是指在一段时间内宏观上有多个程序在同时运行，但在单处理机环境下（一个处理器），每一时刻却仅能有一道程序执行，故微观上这些程序只能是分时地交替执行。例如，在 1 秒钟时间内，0-15ms 程序 A 运行；15-30ms 程序 B 运行；30-45ms 程序 C 运行；45-60ms 程序 D 运行，因此可以说，在 1 秒钟时间间隔内，宏观上有四道程序在同时运行，但微观上，程序 A、B、C、D 是分时地交替执行的。

如果在计算机系统中有多个处理机，这些可以并发执行的程序就可以被分配到多个处理机上，实现并发执行，即利用每个处理机处理一个可并发执行的程序。这样，多个程序便可以同时执行。以此就能提高系统中的资源利用率，增加系统的吞吐量。

4）线程和进程的区别：(必考)

答：

1. 进程是一个“执行中的程序”，是系统进行资源分配和调度的一个独立单位；
2. 线程是进程的一个实体，一个进程中拥有多个线程，线程之间共享地址空间和其它资源（所以通信和同步等操作线程比进程更加容易）；

3. 线程上下文的切换比进程上下文切换要快很多。
    - （1）进程切换时，涉及到当前进程的 CPU 环境的保存和新被调度运行进程的 CPU 环境的设置。
    - （2）线程切换仅需要保存和设置少量的寄存器内容，不涉及存储管理方面的操作。

面试官：进程间如何通讯？线程间如何通讯？

答：进程间通讯依靠 IPC 资源，例如管道（pipes）、套接字（sockets）等；

线程间通讯依靠 JVM 提供的 API，例如 wait()、notify()、notifyAll() 等方法，线程间还可以通过共享的主内存来进行值的传递。

关于线程和进程有一篇写得非常不错的文章，不过是英文的，我进行了翻译，相信阅读之后会对进程和线程有不一样的理解：线程和进程基础——翻译文

5）什么是阻塞（Blocking）和非阻塞（Non-Blocking）？

答：阻塞和非阻塞通常用来形容多线程间的相互影响。比如一个线程占用了临界区资源，那么其他所有需要这个而资源的线程就必须在这个临界区中进行等待。等待会导致线程挂起，这种情况就是阻塞。此时，如果占用资源的线程一直不愿意释放资源，那么其他所有阻塞在这个临界区上的线程都不能工作。

非阻塞的意思与之相反，它强调没有一个线程可以妨碍其他线程执行。所有的线程都会尝试不断前向执行。

面试官：临界区是什么？

答：临界区用来表示一种公共资源或者说是共享资源，可以被多个线程使用。但是每一次，只能有一个线程使用它，一旦临界区资源被占用，其他线程要想使用这个资源，就必须等待。

比如，在一个办公室里有一台打印机，打印机一次只能执行一个任务。如果小王和小明同时需要打印文件，很显然，如果小王先下发了打印任务，打印机就开始打印小王的文件了，小明的任务就只能等待小王打印结束后才能打印，这里的打印机就是一个临界区的例子。

在并行程序中，临界区资源是保护的对象，如果意外出现打印机同时执行两个打印任务，那么最可能的结果就是打印出来的文件就会是损坏的文件，它既不是小王想要的，也不是小明想要的。

6）什么是死锁（Deadlock)、饥饿（Starvation）和活锁（Livelock）？

答：死锁、饥饿和活锁都属于多线程的活跃性问题，如果发现上述几种情况，那么相关线程可能就不再活跃，也就说它可能很难再继续往下执行了。

1. 死锁应该是最糟糕的一种情况了，它表示两个或者两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。
2. 饥饿是指某一个或者多个线程因为种种原因无法获得所需要的资源，导致一直无法执行。比如：
   1）它的线程优先级可能太低，而高优先级的线程不断抢占它需要的资源，导致低优先级的线程无法工作。在自然界中，母鸡喂食雏鸟时，很容易出现这种情况，由于雏鸟很多，食物有限，雏鸟之间的食物竞争可能非常厉害，小雏鸟因为经常抢不到食物，有可能会被饿死。线程的饥饿也非常类似这种情况。
   2）另外一种可能是，某一个线程一直占着关键资源不放，导致其他需要这个资源的线程无法正常执行，这种情况也是饥饿的一种。
   与死锁相比，饥饿还是有可能在未来一段时间内解决的（比如高优先级的线程已经完成任务，不再疯狂的执行）
3. 活锁是一种非常有趣的情况。不知道大家是不是有遇到过这样一种情况，当你要坐电梯下楼，电梯到了，门开了，这时你正准备出去，但不巧的是，门外一个人挡着你的去路，他想进来。于是你很绅士的靠左走，避让对方，但同时对方也很绅士，但他靠右走希望避让你。结果，你们又撞上了。于是乎，你们都意识到了问题，希望尽快避让对方，你立即向右走，他也立即向左走，结果又撞上了！不过介于人类的只能，我相信这个动作重复 2、 3 次后，你应该可以顺利解决这个问题，因为这个时候，大家都会本能的对视，进行交流，保证这种情况不再发生。但如果这种情况发生在两个线程间可能就不会那么幸运了，如果线程的智力不够，且都秉承着 “谦让” 的原则，主动将资源释放给他人使用，那么就会出现资源不断在两个线程中跳动，而没有一个线程可以同时拿到所有的资源而正常执行。这种情况就是活锁。

7）多线程产生死锁的 4 个必要条件？

答：

1. 互斥条件：一个资源每次只能被一个线程使用；
2. 请求与保持条件：一个线程因请求资源而阻塞时，对已获得的资源保持不放；

3. 不剥夺条件：进程已经获得的资源，在未使用完之前，不能强行剥夺；
4. 循环等待条件：若干线程之间形成一种头尾相接的循环等待资源关系。

面试官：如何避免死锁？（经常接着问这个问题哦~）

答：指定获取锁的顺序，举例如下：

1. 比如某个线程只有获得 A 锁和 B 锁才能对某资源进行操作，在多线程条件下，如何避免死锁？
2. 获得锁的顺序是一定的，比如规定，只有获得 A 锁的线程才有资格获取 B 锁，按顺序获取锁就可以避免死锁！！！

8）如何指定多个线程的执行顺序？

解析：面试官会给你举个例子，如何让 10 个线程按照顺序打印 0123456789？（写代码实现）

答：

1. 设定一个 orderNum，每个线程执行结束之后，更新 orderNum，指明下一个要执行的线程。并且唤醒所有的等待线程。
2. 在每一个线程的开始，要 while 判断 orderNum 是否等于自己的要求值！！不是，则 wait，是则执行本线程。

9）Java 中线程有几种状态？

答：六种（查看 Java 源码也可以看到是 6 种），并且某个时刻 Java 线程只能处于其中的一个状态。

1. 新建（NEW）状态：表示新创建了一个线程对象，而此时线程并没有开始执行。
2. 可运行（RUNNABLE）状态：线程对象创建后，其它线程（比如 main 线程）调用了该对象的 start() 方法，才表示线程开始执行。当线程执行时，处于 RUNNBALE 状态，表示线程所需的一切资源都已经准备好了。该状态的线程位于可运行线程池中，等待被线程调度选中，获取 cpu 的使用权。
3. 阻塞（BLOCKED）状态：如果线程在执行过程终于到了 synchronized 同步块，就会进入 BLOCKED 阻塞状态，这时线程就会暂停执行，直到获得请求的锁。

4. 等待（WAITING）状态：当线程等待另一个线程通知调度器一个条件时，它自己进入等待状态。在调用 Object.wait 方法或 Thread.join 方法，或者是等待 java.util.concurrent 库中的 Lock 或 Condition 时，就会出现这种情况；
5. 计时等待（TIMED_WAITING）状态：Object.wait、Thread.join、Lock.tryLock 和 Condition.await 等方法有超时参数，还有 Thread.sleep 方法、LockSupport.parkNanos 方法和 LockSupport.parkUntil 方法，这些方法会导致线程进入计时等待状态，如果超时或者出现通知，都会切换会可运行状态；
6. 终止（TERMINATED）状态：当线程执行完毕，则进入该状态，表示结束。

注意：从 NEW 状态出发后，线程不能再回到 NEW 状态，同理，处于 TERMINATED 状态的线程也不能再回到 RUNNABLE 状态。

（二）高并发编程-JUC 包

在 Java 5.0 提供了 java.util.concurrent（简称 JUC ）包，在此包中增加了在并发编程中很常用的实用工具类，用于定义类似于线程的自定义子系统，包括线程池、异步 IO 和轻量级任务框架。

1）sleep( ) 和 wait(n)、wait( ) 的区别：

答：

1. sleep 方法：是 Thread 类的静态方法，当前线程将睡眠 n 毫秒，线程进入阻塞状态。当睡眠时间到了，会解除阻塞，进行可运行状态，等待 CPU 的到来。睡眠不释放锁（如果有的话）；
2. wait 方法：是 Object 的方法，必须与 synchronized 关键字一起使用，线程进入阻塞状态，当 notify 或者 notifyall 被调用后，会解除阻塞。但是，只有重新占用互斥锁之后才会进入可运行状态。睡眠时，释放互斥锁。

2）synchronized 关键字：

答：底层实现：

1. 进入时，执行 monitorenter，将计数器 +1，释放锁 monitorexit 时，计数器-1；

2. 当一个线程判断到计数器为 0 时，则当前锁空闲，可以占用；反之，当前线程进入等待状态。

含义：（monitor 机制）

Synchronized 是在加锁，加对象锁。对象锁是一种重量锁（monitor），synchronized 的锁机制会根据线程竞争情况在运行时会有偏向锁（单一线程）、轻量锁（多个线程访问 synchronized 区域）、对象锁（重量锁，多个线程存在竞争的情况）、自旋锁等。

该关键字是一个几种锁的封装。

3）volatile 关键字：

答：该关键字可以保证可见性不保证原子性。

功能：

1. 主内存和工作内存，直接与主内存产生交互，进行读写操作，保证可见性；
2. 禁止 JVM 进行的指令重排序。

解析：关于指令重排序的问题，可以查阅 DCL 双检锁失效相关资料。

4）volatile 能使得一个非原子操作变成原子操作吗？

答：能。

一个典型的例子是在类中有一个 long 类型的成员变量。如果你知道该成员变量会被多个线程访问，如计数器、价格等，你最好是将其设置为 volatile。为什么？因为 Java 中读取 long 类型变量不是原子的，需要分成两步，如果一个线程正在修改该 long 变量的值，另一个线程可能只能看到该值的一半（前 32 位）。但是对一个 volatile 型的 long 或 double 变量的读写是原子。

面试官：volatile 修饰符的有过什么实践？

答：

1. 一种实践是用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写。double 和 long 都是 64 位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然后再读剩下的 32 位，这个过程不是原子的，但 Java 中 volatile 型的 long 或 double 变量的读写是原子的。

2. volatile 修复符的另一个作用是提供内存屏障（memory barrier），例如在分布式框架中的应用。简单的说，就是当你写一个 volatile 变量之前，Java 内存模型会插入一个写屏障（write barrier），读一个 volatile 变量之前，会插入一个读屏障（read barrier）。意思就是说，在你写一个 volatile 域时，能保证任何线程都能看到你写的值，同时，在写之前，也能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新到缓存。

5）ThreadLocal（线程局部变量）关键字：

答：当使用 ThreadLocal 维护变量时，其为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立的改变自己的副本，而不会影响其他线程对应的副本。

ThreadLocal 内部实现机制：

1. 每个线程内部都会维护一个类似 HashMap 的对象，称为 ThreadLocalMap，里边会包含若干了 Entry（K-V 键值对），相应的线程被称为这些 Entry 的属主线程；
2. Entry 的 Key 是一个 ThreadLocal 实例，Value 是一个线程特有对象。Entry 的作用即是：为其属主线程建立起一个 ThreadLocal 实例与一个线程特有对象之间的对应关系；
3. Entry 对 Key 的引用是弱引用；Entry 对 Value 的引用是强引用。

6）线程池有了解吗？（必考）

答：java.util.concurrent.ThreadPoolExecutor 类就是一个线程池。客户端调用 ThreadPoolExecutor.submit(Runnable task) 提交任务，线程池内部维护的工作者线程的数量就是该线程池的线程池大小，有 3 种形态：

- 当前线程池大小 ：表示线程池中实际工作者线程的数量；
- 最大线程池大小 （maxinumPoolSize）：表示线程池中允许存在的工作者线程的数量上限；
- 核心线程大小 （corePoolSize ）：表示一个不大于最大线程池大小的工作者线程数量上限。

1. 如果运行的线程少于 corePoolSize，则 Executor 始终首选添加新的线程，而不进行排队；
2. 如果运行的线程等于或者多于 corePoolSize，则 Executor 始终首选将请求加入队列，而不是添加新线程；

3. 如果无法将请求加入队列，即队列已经满了，则创建新的线程，除非创建此线程超出 maxinumPoolSize， 在这种情况下，任务将被拒绝。

面试官：我们为什么要使用线程池？

答：

1. 减少创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。
2. 可以根据系统的承受能力，调整线程池中工作线程的数目，放置因为消耗过多的内存，而把服务器累趴下（每个线程大约需要 1 MB 内存，线程开的越多，消耗的内存也就越大，最后死机）

面试官：核心线程池内部实现了解吗？

答：对于核心的几个线程池，无论是 newFixedThreadPool() 方法，newSingleThreadExecutor() 还是 newCachedThreadPool() 方法，虽然看起来创建的线程有着完全不同的功能特点，但其实内部实现均使用了 ThreadPoolExecutor 实现，其实都只是 ThreadPoolExecutor 类的封装。

为何 ThreadPoolExecutor 有如此强大的功能呢？我们可以来看一下 ThreadPoolExecutor 最重要的构造函数：

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

函数的参数含义如下：

- corePoolSize：指定了线程池中的线程数量
- maximumPoolSize：指定了线程池中的最大线程数量
- keepAliveTime：当线程池线程数量超过 corePoolSize 时，多余的空闲线程的存活时间。即，超过了 corePoolSize 的空闲线程，在多长时间内，会被销毁。
- unit: keepAliveTime 的单位。
- workQueue：任务队列，被提交但尚未被执行的任务。
- threadFactory：线程工厂，用于创建线程，一般用默认的即可。
- handler：拒绝策略。当任务太多来不及处理，如何拒绝任务。

7）Atomic 关键字：

答：可以使基本数据类型以原子的方式实现自增自减等操作。参考博客：
concurrent.atomic 包下的类 AtomicInteger 的使用

8）创建线程有哪几种方式？

答：有两种创建线程的方法：一是实现 Runnable 接口，然后将它传递给 Thread 的构造函数，创建一个 Thread 对象;二是直接继承 Thread 类。

面试官：两种方式有什么区别呢？

1. 继承方式:
   - （1）Java 中类是单继承的,如果继承了 Thread 了,该类就不能再有其他的直接父类了.
   - （2）从操作上分析,继承方式更简单,获取线程名字也简单.(操作上,更简单)
   - （3）从多线程共享同一个资源上分析,继承方式不能做到.
2. 实现方式:
   - （1）Java 中类可以多实现接口,此时该类还可以继承其他类,并且还可以实现其他接口(设计上,更优雅).
   - （2）从操作上分析,实现方式稍微复杂点,获取线程名字也比较复杂,得使用 Thread.currentThread()来获取当前线程的引用.
   - （3）从多线程共享同一个资源上分析,实现方式可以做到(是否共享同一个资源).

9）run() 方法和 start() 方法有什么区别？

答：start() 方法会新建一个线程并让这个线程执行 run() 方法；而直接调用 run() 方法知识作为一个普通的方法调用而已，它只会在当前线程中，串行执行 run() 中的代码。

10）你怎么理解线程优先级？

答：Java 中的线程可以有自己的优先级。优先极高的线程在竞争资源时会更有优势，更可能抢占资源，当然，这只是一个概率问题。如果运行不好，高优先级线程可能也会抢占失败。

由于线程的优先级调度和底层操作系统有密切的关系，在各个平台上表现不一，并且这种优先级产生的后果也可能不容易预测，无法精准控制，比如一个低优先级的线程可能一直抢占不到资源，从而始终无法运行，而产生饥饿（虽

然优先级低，但是也不能饿死它啊）。因此，在要求严格的场合，还是需要自己在应用层解决线程调度的问题。

在 Java 中，使用 1 到 10 表示线程优先级，一般可以使用内置的三个静态标量表示：

```
public final static int MIN_PRIORITY = 1;
public final static int NORM_PRIORITY = 5;
public final static int MAX_PRIORITY = 10;
```

数字越大则优先级越高，但有效范围在 1 到 10 之间，默认的优先级为 5 。

11）在 Java 中如何停止一个线程？

答：Java 提供了很丰富的 API 但没有为停止线程提供 API 。

JDK 1.0 本来有一些像 stop()，suspend() 和 resume() 的控制方法但是由于潜在的死锁威胁因此在后续的 JDK 版本中他们被弃用了，之后 Java API 的设计者就没有提供一个兼容且线程安全的方法来停止任何一个线程。

当 run() 或者 call() 方法执行完的时候线程会自动结束，如果要手动结束一个线程，你可以用 volatile 布尔变量来退出 run() 方法的循环或者是取消任务来中断线程。

12）多线程中的忙循环是什么？

答：忙循环就是程序员用循环让一个线程等待，不像传统方法 wait(),sleep() 或 yield() 它们都放弃了 CPU 控制权，而忙循环不会放弃 CPU，它就是在运行一个空循环。这么做的目的是为了保留 CPU 缓存。

在多核系统中，一个等待线程醒来的时候可能会在另一个内核运行，这样会重建缓存，为了避免重建缓存和减少等待重建的时间就可以使用它了。

13）10 个线程和 2 个线程的同步代码，哪个更容易写？

答：从写代码的角度来说，两者的复杂度是相同的，因为同步代码与线程数量是相互独立的。但是同步策略的选择依赖于线程的数量，因为越多的线程意味着更大的竞争，所以你需要利用同步技术，如锁分离，这要求更复杂的代码和专业知识。

14）你是如何调用 wait（）方法的？使用 if 块还是循环？为什么？

答：wait() 方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没有满足，所以在处理前，循环检测条件是否满足会更好。下面是一段标准的使用 wait 和 notify 方法的代码：

```
// The standard idiom for using the wait method
synchronized (obj) {
while (condition does not hold)
obj.wait(); // (Releases lock, and reacquires on wakeup)
... // Perform action appropriate to condition
}
```

参见 Effective Java 第 69 条，获取更多关于为什么应该在循环中来调用 wait 方法的内容。

15）什么是多线程环境下的伪共享（false sharing）？

答：伪共享是多线程系统（每个处理器有自己的局部缓存）中一个众所周知的性能问题。伪共享发生在不同处理器的上的线程对变量的修改依赖于相同的缓存行，如下图所示：

伪共享问题很难被发现，因为线程可能访问完全不同的全局变量，内存中却碰巧在很相近的位置上。如其他诸多的并发问题，避免伪共享的最基本方式是仔细审查代码，根据缓存行来调整你的数据结构。

16）用 wait-notify 写一段代码来解决生产者-消费者问题？

解析：这是常考的基础类型的题，只要记住在同步块中调用 wait() 和 notify()方法，如果阻塞，通过循环来测试等待条件。

答：

```
import java.util.Vector;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
```

```
 * Java program to solve Producer Consumer problem using wait and
notify
 * method in Java. Producer Consumer is also a popular concurrency
design pattern.
 *
 * @author Javin Paul
 */
public class ProducerConsumerSolution {

    public static void main(String args[]) {
        Vector sharedQueue = new Vector();
        int size = 4;
        Thread prodThread = new Thread(new Producer(sharedQueue,
size), "Producer");
        Thread consThread = new Thread(new Consumer(sharedQueue,
size), "Consumer");
        prodThread.start();
        consThread.start();
    }
}

class Producer implements Runnable {

    private final Vector sharedQueue;
    private final int SIZE;

    public Producer(Vector sharedQueue, int size) {
        this.sharedQueue = sharedQueue;
        this.SIZE = size;
    }

    @Override
    public void run() {
        for (int i = 0; i < 7; i++) {
            System.out.println("Produced: " + i);
            try {
                produce(i);
            } catch (InterruptedException ex) {

Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null,
ex);
            }

        }
```

```
    }

    private void produce(int i) throws InterruptedException {

        // wait if queue is full
        while (sharedQueue.size() == SIZE) {
            synchronized (sharedQueue) {
                System.out.println("Queue is full " +
Thread.currentThread().getName()
                                    + " is waiting , size: " +
sharedQueue.size());

                sharedQueue.wait();
            }
        }

        // producing element and notify consumers
        synchronized (sharedQueue) {
            sharedQueue.add(i);
            sharedQueue.notifyAll();
        }
    }
}

class Consumer implements Runnable {

    private final Vector sharedQueue;
    private final int SIZE;

    public Consumer(Vector sharedQueue, int size) {
        this.sharedQueue = sharedQueue;
        this.SIZE = size;
    }

    @Override
    public void run() {
        while (true) {
            try {
                System.out.println("Consumed: " + consume());
                Thread.sleep(50);
            } catch (InterruptedException ex) {

Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null,
ex);
```

```
            }

        }
    }

    private int consume() throws InterruptedException {
        // wait if queue is empty
        while (sharedQueue.isEmpty()) {
            synchronized (sharedQueue) {
                System.out.println("Queue is empty " +
Thread.currentThread().getName()
                                    + " is waiting , size: " +
sharedQueue.size());

                sharedQueue.wait();
            }
        }

        // Otherwise consume element and notify waiting producer
        synchronized (sharedQueue) {
            sharedQueue.notifyAll();
            return (Integer) sharedQueue.remove(0);
        }
    }
}

Output:
Produced: 0
Queue is empty Consumer is waiting , size: 0
Produced: 1
Consumed: 0
Produced: 2
Produced: 3
Produced: 4
Produced: 5
Queue is full Producer is waiting , size: 4
Consumed: 1
Produced: 6
Queue is full Producer is waiting , size: 4
Consumed: 2
Consumed: 3
Consumed: 4
Consumed: 5
Consumed: 6
```

```
Queue is empty Consumer is waiting , size: 0
```

17）用 Java 写一个线程安全的单例模式（Singleton）？

解析：有多种方法，但重点掌握的是双重校验锁。

答：

1.饿汉式单例

饿汉式单例是指在方法调用前，实例就已经创建好了。下面是实现代码：

```
public class Singleton {

    private static Singleton instance = new Singleton();

    private Singleton (){}

    public static Singleton getInstance() {
        return instance;
    }
}
```

2.加入 synchronized 的懒汉式单例

所谓懒汉式单例模式就是在调用的时候才去创建这个实例，我们在对外的创建实例方法上加如 synchronized 关键字保证其在多线程中很好的工作：

```
public class Singleton {

    private static Singleton instance;

    private Singleton (){}

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
    }
    return instance;
    }
}
```

3.使用静态内部类的方式创建单例

这种方式利用了 classloder 的机制来保证初始化 instance 时只有一个线程，它跟饿汉式的区别是：饿汉式只要 Singleton 类被加载了，那么 instance 就会被实例化（没有达到 lazy loading 的效果），而这种方式是 Singleton 类被加载了，instance 不一定被初始化。只有显式通过调用 getInstance() 方法时才会显式装载 SingletonHoder 类，从而实例化 singleton

```
public class Singleton {

    private Singleton() {
    }

    private static class SingletonHolder {// 静态内部类
        private static Singleton singleton = new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonHolder.singleton;
    }
}
```

4.双重校验锁

为了达到线程安全，又能提高代码执行效率，我们这里可以采用 DCL 的双检查锁机制来完成，代码实现如下：

```
public class Singleton {

    private static Singleton singleton;

    private Singleton() {
    }

    public static Singleton getInstance(){
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

这种是用双重判断来创建一个单例的方法，那么我们为什么要使用两个 if 判断这个对象当前是不是空的呢 ？因为当有多个线程同时要创建对象的时候，多个线程有可能都停止在第一个 if 判断的地方，等待锁的释放，然后多个线程就都创建了对象，这样就不是单例模式了，所以我们要用两个 if 来进行这个对象是否存在的判断。

5.使用 static 代码块实现单例

静态代码块中的代码在使用类的时候就已经执行了，所以可以应用静态代码块的这个特性的实现单例设计模式。

```
public class Singleton{

    private static Singleton instance = null;

    private Singleton(){}

    static{
        instance = new Singleton();
    }

    public static Singleton getInstance() {
        return instance;
    }
}
```

6.使用枚举数据类型实现单例模式

枚举 enum 和静态代码块的特性相似，在使用枚举时，构造方法会被自动调用，利用这一特性也可以实现单例：

```
public class ClassFactory{

    private enum MyEnumSingleton{
        singletonFactory;

        private MySingleton instance;

        private MyEnumSingleton(){//枚举类的构造方法在类加载是被实例
化
            instance = new MySingleton();
        }

        public MySingleton getInstance(){
            return instance;
```

```
        }
    }

    public static MySingleton getInstance(){
        return MyEnumSingleton.singletonFactory.getInstance();
    }
}
```

小结：关于 Java 中多线程编程，线程安全等知识一直都是面试中的重点和难点，还需要熟练掌握。

(三)——JVM 篇

（一）JVM 基础知识

问题和答案都是自行整理的，所以仅供参考！欢迎指正！

1）Java 是如何实现跨平台的？

注意：跨平台的是 Java 程序，而不是 JVM。JVM 是用 C/C++ 开发的，是编译后的机器码，不能跨平台，不同平台下需要安装不同版本的 JVM

答：我们编写的 Java 源码，编译后会生成一种 .class 文件，称为字节码文件。Java 虚拟机（JVM）就是负责将字节码文件翻译成特定平台下的机器码然后运行，也就是说，只要在不同平台上安装对应的 JVM，就可以运行字节码文件，运行我们编写的 Java 程序。

而这个过程，我们编写的 Java 程序没有做任何改变，仅仅是通过 JVM 这一“中间层”，就能在不同平台上运行，真正实现了“一次编译，到处运行”的目的。

2）什么是 JVM ？

解析：不仅仅是基本概念，还有 JVM 的作用。

答：JVM，即 Java Virtual Machine，Java 虚拟机。它通过模拟一个计算机来达到一个计算机所具有的的计算功能。JVM 能够跨计算机体系结构来执行 Java 字节码，主要是由于 JVM 屏蔽了与各个计算机平台相关的软件或者硬件之间的差异，使得与平台相关的耦合统一由 JVM 提供者来实现。

3）JVM 由哪些部分组成？

解析：这是对 JVM 体系结构的考察

答：JVM 的结构基本上由 4 部分组成：

- 类加载器，在 JVM 启动时或者类运行时将需要的 class 加载到 JVM 中
- 执行引擎，执行引擎的任务是负责执行 class 文件中包含的字节码指令，相当于实际机器上的 CPU
- 内存区，将内存划分成若干个区以模拟实际机器上的存储、记录和调度功能模块，如实际机器上的各种功能的寄存器或者 PC 指针的记录器等
- 本地方法调用，调用 C 或 C++ 实现的本地方法的代码返回结果

4）类加载器是有了解吗？

解析：底层原理的考察，其中涉及到类加载器的概念，功能以及一些底层的实现。

答：顾名思义，类加载器（class loader）用来加载 Java 类到 Java 虚拟机中。一般来说，Java 虚拟机使用 Java 类的方式如下：Java 源程序（.java 文件）在经过 Java 编译器编译之后就被转换成 Java 字节代码（.class 文件）。

类加载器负责读取 Java 字节代码，并转换成 java.lang.Class 类的一个实例。每个这样的实例用来表示一个 Java 类。通过此实例的 newInstance()方法就可以创建出该类的一个对象。实际的情况可能更加复杂，比如 Java 字节代码可能是通过工具动态生成的，也可能是通过网络下载的。

面试官：Java 虚拟机是如何判定两个 Java 类是相同的？

答：Java 虚拟机不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。只有两者都相同的情况，才认为两个类是相同的。即便是同样的字节代码，被不同的类加载器加载之后所得到的类，也是不同的。比如一个 Java 类 com.example.Sample，编译之后生成了字节代码文件 Sample.class。两个不同的类加载器 ClassLoaderA 和 ClassLoaderB 分别读取了这个 Sample.class 文

件，并定义出两个 java.lang.Class 类的实例来表示这个类。这两个实例是不相同的。对于 Java 虚拟机来说，它们是不同的类。试图对这两个类的对象进行相互赋值，会抛出运行时异常 ClassCastException。

5）类加载器是如何加载 class 文件的？

答：下图所示是 ClassLoader 加载一个 class 文件到 JVM 时需要经过的步骤：

第一个阶段是找到 .class 文件并把这个文件包含的字节码加载到内存中

第二阶段又可以分为三个步骤，分别是字节码验证、Class 类数据结构分析及相应的内存分配和最后的符号表的链接

第三个阶段是类中静态属性和初始化赋值，以及静态块的执行等

面试官：能详细讲讲吗？

答：

1.加载

查找并加载类的二进制数据加载时类加载过程的第一个阶段，在加载阶段，虚拟机需要完成以下三件事情：

- 通过一个类的全限定名来获取其定义的二进制字节流。
- 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
- 在 Java 堆中生成一个代表这个类的 java.lang.Class 对象，作为对方法区中这些数据的访问入口。

相对于类加载的其他阶段而言，加载阶段（准确地说，是加载阶段获取类的二进制字节流的动作）是可控性最强的阶段，因为开发人员既可以使用系统提供的类加载器来完成加载，也可以自定义自己的类加载器来完成加载。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，而且在 Java 堆中也创建一个 java.lang.Class 类的对象，这样便可以通过该对象访问方法区中的这些数据。

2.连接

验证：确保被加载的类的正确性

验证是连接阶段的第一步，这一阶段的目的是为了确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。验证阶段大致会完成 4 个阶段的检验动作：

- **文件格式验证：**验证字节流是否符合 Class 文件格式的规范；例如：是否以 0xCAFEBABE 开头、主次版本号是否在当前虚拟机的处理范围之内、常量池中的常量是否有不被支持的类型。
- **元数据验证：**对字节码描述的信息进行语义分析（注意：对比 javac 编译阶段的语义分析），以保证其描述的信息符合 Java 语言规范的要求；例如：这个类是否有父类，除了 java.lang.Object 之外。
- **字节码验证：**通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。
- **符号引用验证：**确保解析动作能正确执行。

验证阶段是非常重要的，但不是必须的，它对程序运行期没有影响，如果所引用的类经过反复验证，那么可以考虑采用 -Xverifynone 参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

准备：为类的静态变量分配内存，并将其初始化为默认值

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。对于该阶段有以下几点需要注意：

- ① 这时候进行内存分配的仅包括类变量（static），而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在 Java 堆中。
- ② 这里所设置的初始值通常情况下是数据类型默认的零值（如 0、0L、null、false 等），而不是被在 Java 代码中被显式地赋予的值。

假设一个类变量的定义为： public static int value = 3;

那么变量 value 在准备阶段过后的初始值为 0，而不是 3，因为这时候尚未开始执行任何 Java 方法，而把 value 赋值为 3 的 public static 指令是在程序编译后，存放于类构造器 <clinit>（）方法之中的，所以把 value 赋值为 3 的动作将在初始化阶段才会执行。

这里还需要注意如下几点：

- 对基本数据类型来说，对于类变量（static）和全局变量，如果不显式地对其赋值而直接使用，则系统会为其赋予默认的零值，而对于局部变量来说，在使用前必须显式地为其赋值，否则编译时不通过。
- 对于同时被 static 和 final 修饰的常量，必须在声明的时候就为其显式地赋值，否则编译时不通过；而只被 final 修饰的常量则既可以在声明时显式地为其赋值，也可以在类初始化时显式地为其赋值，总之，在使用前必须为其显式地赋值，系统不会为其赋予默认零值。
- 对于引用数据类型 reference 来说，如数组引用、对象引用等，如果没有对其进行显式地赋值而直接使用，系统都会为其赋予默认的零值，即 null。

- 如果在数组初始化时没有对数组中的各元素赋值，那么其中的元素将根据对应的数据类型而被赋予默认的零值。

- ③ 如果类字段的字段属性表中存在 ConstantValue 属性，即同时被 final 和 static 修饰，那么在准备阶段变量 value 就会被初始化为 ConstValue 属性所指定的值。

假设上面的类变量 value 被定义为： public static final int value = 3;

编译时 Javac 将会为 value 生成 ConstantValue 属性，在准备阶段虚拟机就会根据 ConstantValue 的设置将 value 赋值为 3。我们可以理解为 static final 常量在编译期就将其结果放入了调用它的类的常量池中

解析：把类中的符号引用转换为直接引用

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程，解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符 7 类符号引用进行。符号引用就是一组符号来描述目标，可以是任何字面量。

直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

3.初始化

初始化，为类的静态变量赋予正确的初始值，JVM 负责对类进行初始化，主要对类变量进行初始化。在 Java 中对类变量进行初始值设定有两种方式：

- ① 声明类变量是指定初始值
- ② 使用静态代码块为类变量指定初始值

JVM 初始化步骤

- 1、假如这个类还没有被加载和连接，则程序先加载并连接该类
- 2、假如该类的直接父类还没有被初始化，则先初始化其直接父类
- 3、假如类中有初始化语句，则系统依次执行这些初始化语句

类初始化时机：只有当对类的主动使用的时候才会导致类的初始化，类的主动使用包括以下六种：

- 创建类的实例，也就是 new 的方式
- 访问某个类或接口的静态变量，或者对该静态变量赋值
- 调用类的静态方法
- 反射（如 Class.forName(“com.shengsiyuan.Test”)）
- 初始化某个类的子类，则其父类也会被初始化

- Java 虚拟机启动时被标明为启动类的类（ JavaTest），直接使用 java.exe 命令来运行某个主类

结束生命周期

在如下几种情况下，Java 虚拟机将结束生命周期

- 执行了 System.exit()方法
- 程序正常执行结束
- 程序在执行过程中遇到了异常或错误而异常终止
- 由于操作系统出现错误而导致 Java 虚拟机进程终止

参考文章：[jvm 系列(一):java 类的加载机制 - 纯洁的微笑]

7）双亲委派模型（Parent Delegation Model）？

解析：类的加载过程采用双亲委派机制，这种机制能更好的保证 Java 平台的安全性

答：类加载器 ClassLoader 是具有层次结构的，也就是父子关系，其中，Bootstrap 是所有类加载器的父亲，如下图所示：

该模型要求除了顶层的 Bootstrap class loader 启动类加载器外，其余的类加载器都应当有自己的父类加载器。子类加载器和父类加载器不是以继承（Inheritance）的关系来实现，而是通过组合（Composition）关系来复用父加载器的代码。每个类加载器都有自己的命名空间（由该加载器及所有父类加载器所加载的类组成，在同一个命名空间中，不会出现类的完整名字（包括类的包名）相同的两个类；在不同的命名空间中，有可能会出现类的完整名字（包括类的包名）相同的两个类）

面试官：双亲委派模型的工作过程？

答：

1.当前 ClassLoader 首先从自己已经加载的类中查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。

每个类加载器都有自己的加载缓存，当一个类被加载了以后就会放入缓存，等下次加载的时候就可以直接返回了。

2.当前 ClassLoader 的缓存中没有找到被加载的类的时候，委托父类加载器去加载，父类加载器采用同样的策略，首先查看自己的缓存，然后委托父类的父类去加载，一直到 bootstrap ClassLoader.

当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中，以便下次有加载请求的时候直接返回。

面试官：为什么这样设计呢？

解析：这是对于使用这种模型来组织累加器的好处

答：主要是为了安全性，避免用户自己编写的类动态替换 Java 的一些核心类，比如 String，同时也避免了重复加载，因为 JVM 中区分不同类，不仅仅是根据类名，相同的 class 文件被不同的 ClassLoader 加载就是不同的两个类，如果相互转型的话会抛 java.lang.ClassCaseException.

参考文章： JVM 的 工作原理，层次结构 以及 GC 工作原理

（二）JVM 内存管理

1）JVM 内存划分：

答：

1. 方法区（线程共享）：各个线程共享的一个区域，用于存储虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却又一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区分开来。
    - 运行时常量池：是方法区的一部分，用于存放编译器生成的各种字面量和符号引用。
2. 堆内存（线程共享）：所有线程共享的一块区域，垃圾收集器管理的主要区域。目前主要的垃圾回收算法都是分代收集算法，所以 Java 堆中还可以细分为：新生代和老年代；再细致一点的有 Eden 空间、From Survivor 空间、To Survivor 空间等，默认情况下新生代按照 8:1:1 的比例来分配。根据 Java 虚拟机规范的规定，Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘一样。
3. 程序计数器： Java 线程私有，类似于操作系统里的 PC 计数器，它可以看做是当前线程所执行的字节码的行号指示器。如果线程正在执行的

是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 Native 方法，这个计数器值则为空（Undefined）。此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。

4. 虚拟机栈（栈内存）：Java 线程私有，虚拟机展描述的是 Java 方法执行的内存模型：每个方法在执行的时候，都会创建一个栈帧用于存储局部变量、操作数、动态链接、方法出口等信息；每个方法调用都意味着一个栈帧在虚拟机栈中入栈到出栈的过程；
5. 本地方法栈 ：和 Java 虚拟机栈的作用类似，区别是该区域为 JVM 提供使用 native 方法的服务

2）对象分配规则？

答：

- 对象优先分配在 Eden 区，如果 Eden 区没有足够的空间时，虚拟机执行一次 Minor GC。
- 大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在 Eden 区和两个 Survivor 区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。
- 长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了 1 次 Minor GC 那么对象会进入 Survivor 区，之后每经过一次 Minor GC 那么对象的年龄加 1，知道达到阀值对象进入老年区。
- 动态判断对象的年龄。如果 Survivor 区中相同年龄的所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。
- 空间分配担保。每次进行 Minor GC 时，JVM 会计算 Survivor 区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次 Full GC，如果小于检查 HandlePromotionFailure 设置，如果 true 则只进行 Monitor GC,如果 false 则进行 Full GC。

3）Java 的内存模型：

答：

Java 虚拟机规范中试图定义一种 Java 内存模型（Java Memory Model，JMM）来屏蔽掉各层硬件和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致的内存访问效果。

Java 内存模型规定了所有的变量都存储在主内存（Main Memory）中。每条线程还有自己的工作内存（Working Memory），线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在主内存中进行，而不能直接读写主内存中的变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间的变量值的传递均需要通过主内存来完成，线程、主内存、工作内存三者的关系如上图。

面试官：两个线程之间是如何通信的呢？

答：在**共享内存**的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信，典型的共享内存通信方式就是通过共享对象进行通信。

例如上图线程 A 与 线程 B 之间如果要通信的话，那么就必须经历下面两个步骤：

- 1.首先，线程 A 把本地内存 A 更新过得共享变量刷新到主内存中去
- 2.然后，线程 B 到主内存中去读取线程 A 之前更新过的共享变量

在**消息传递**的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信，在 Java 中典型的消息传递方式就是 wait() 和 notify()。

5）内存屏障？

解析：在这之前应该对重排序的问题有所了解，这里我找到一篇很好的文章分享一下：Java 内存访问重排序的研究

答：内存屏障，又称内存栅栏，是一组处理器指令，用于实现对内存操作的顺序限制。

面试官：内存屏障为何重要？

答：对主存的一次访问一般花费硬件的数百次时钟周期。处理器通过缓存（caching）能够从数量级上降低内存延迟的成本这些缓存为了性能重新排列待定内存操 作的顺序。也就是说，程序的读写操作不一定会按照它要求处理器的顺序执行。当数据是不可变的，同时/或者数据限制在线程范围内，这些优化是无害的。如果把 这些优化与对称多处理（symmetric multi-processing）和共

享可变状态（shared mutable state）结合，那么就是一场噩梦。当基于共享可变状态的内存操作被重新排序时，程序可能行为不定。一个线程写入的数据可能被其他线程可见，原因是数据 写入的顺序不一致。适当的放置内存屏障通过强制处理器顺序执行待定的内存操作来避免这个问题。

5）类似-Xms、-Xmn 这些参数的含义：

答：

堆内存分配：

1. JVM 初始分配的内存由-Xms 指定，默认是物理内存的 1/64
2. JVM 最大分配的内存由-Xmx 指定，默认是物理内存的 1/4
3. 默认空余堆内存小于 40%时，JVM 就会增大堆直到-Xmx 的最大限制；空余堆内存大于 70%时，JVM 会减少堆直到 -Xms 的最小限制。
4. 因此服务器一般设置-Xms、-Xmx 相等以避免在每次 GC 后调整堆的大小。对象的堆内存由称为垃圾回收器的自动内存管理系统回收。

非堆内存分配：

1. JVM 使用-XX:PermSize 设置非堆内存初始值，默认是物理内存的 1/64；
2. 由 XX:MaxPermSize 设置最大非堆内存的大小，默认是物理内存的 1/4。
3. -Xmn2G：设置年轻代大小为 2G。
4. -XX:SurvivorRatio，设置年轻代中 Eden 区与 Survivor 区的比值。

6）内存泄漏和内存溢出

答：

概念：

1. 内存溢出指的是内存不够用了。
2. 内存泄漏是指对象可达，但是没用了。即本该被 GC 回收的对象并没有被回收
3. 内存泄露是导致内存溢出的原因之一；内存泄露积累起来将导致内存溢出。

内存泄漏的原因分析：

1. 长生命周期的对象引用短生命周期的对象
2. 没有将无用对象置为 null

小结：本小节涉及到 JVM 虚拟机，包括对内存的管理等知识，相对较深。除了以上问题，面试官会继续问你一些比较深的问题，可能也是为了看看你的极限在哪里吧。比如：内存调优、内存管理，是否遇到过内存泄露的实际案例、是否真正关心过内存等。

7）简述一下 Java 中创建一个对象的过程？

解析：回答这个问题首先就要清楚类的生命周期

答：下图展示的是类的生命周期流向：

Java 中对象的创建就是在堆上分配内存空间的过程，此处说的对象创建仅限于 new 关键字创建的普通 Java 对象，不包括数组对象的创建。

大致过程如下：

1.检测类是否被加载：

当虚拟机执行到 new 时，会先去常量池中查找这个类的符号引用。如果能找到符号引用，说明此类已经被加载到方法区（方法区存储虚拟机已经加载的类的信息），可以继续执行；如果找不到符号引用，就会使用类加载器执行类的加载过程，类加载完成后继续执行。

2.为对象分配内存：

类加载完成以后，虚拟机就开始为对象分配内存，此时所需内存的大小就已经确定了。只需要在堆上分配所需要的内存即可。

具体的分配内存有两种情况：第一种情况是内存空间绝对规整，第二种情况是内存空间是不连续的。

- 对于内存绝对规整的情况相对简单一些，虚拟机只需要在被占用的内存和可用空间之间移动指针即可，这种方式被称为指针碰撞。
- 对于内存不规整的情况稍微复杂一点，这时候虚拟机需要维护一个列表，来记录哪些内存是可用的。分配内存的时候需要找到一个可用的内存空间，然后在列表上记录下已被分配，这种方式成为空闲列表。

分配内存的时候也需要考虑线程安全问题，有两种解决方案：

- 第一种是采用同步的办法，使用 CAS 来保证操作的原子性。
- 另一种是每个线程分配内存都在自己的空间内进行，即是每个线程都在堆中预先分配一小块内存，称为本地线程分配缓冲（TLAB），分配内存的时候再 TLAB 上分配，互不干扰。

3.为分配的内存空间初始化零值：

对象的内存分配完成后，还需要将对象的内存空间都初始化为零值，这样能保证对象即使没有赋初值，也可以直接使用。

4.对对象进行其他设置：

分配完内存空间，初始化零值之后，虚拟机还需要对对象进行其他必要的设置，设置的地方都在对象头中，包括这个对象所属的类，类的元数据信息，对象的 hashcode，GC 分代年龄等信息。

5.执行 init 方法：

执行完上面的步骤之后，在虚拟机里这个对象就算创建成功了，但是对于 Java 程序来说还需要执行 init 方法才算真正的创建完成，因为这个时候对象只是被初始化零值了，还没有真正的去根据程序中的代码分配初始值，调用了 init 方法之后，这个对象才真正能使用。

到此为止一个对象就产生了，这就是 new 关键字创建对象的过程。过程如下：

参考文章：Java 创建对象的过程简介

面试官：对象的内存布局是怎样的？

答：对象的内存布局包括三个部分：对象头，实例数据和对齐填充。

- 对象头：对象头包括两部分信息，第一部分是存储对象自身的运行时数据，如哈希码，GC 分代年龄，锁状态标志，线程持有的锁等等。第二部分是类型指针，即对象指向类元数据的指针。
- 实例数据：就是数据啦
- 对齐填充：不是必然的存在，就是为了对齐的嘛

面试官：对象是如何定位访问的？

答：对象的访问定位有两种：句柄定位和直接指针

- 句柄定位：Java 堆会画出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息

错误!未指定文件名。

- 直接指针访问：java 堆对象的不居中就必须考虑如何放置访问类型数据的相关信息，而 reference 中存储的直接就是对象地址

错误!未指定文件名。

比较：使用直接指针就是速度快，使用句柄 reference 指向稳定的句柄，对象被移动改变的也只是句柄中实例数据的指针，而 reference 本身并不需要修改。

参考文章：JAVA 对象创建的过程

（三）GC 相关

1）如何判断一个对象是否已经死去？

答：

1. 引用计数：每个对象有一个引用计数属性，新增一个引用时计数加 1，引用释放时计数减 1，计数为 0 时可以回收。此方法简单，无法解决对象相互循环引用的问题。
2. 可达性分析（Reachability Analysis）：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是不可用的。不可达对象。

2）垃圾回收算法有哪些？

答：

1. 引用计数：
   原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。垃圾回收时，只用收集计数为 0 的对象。此算法最致命的是无法处理循环引用的问题。
2. 标记-清除：
   此算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，同时，会产生内存碎片。

错误!未指定文件名。

3. 复制算法：
   此算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。此算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现“碎片”问题。当然，此算法的缺点也是很明显的，就是需要两倍内存空间。

错误!未指定文件名。

4. 标记-整理：
   此算法结合了“标记-清除”和“复制”两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。

错误!未指定文件名。

5. 分代收集算法：
   - 分代收集算法并没有提出新的思想，只是根据对象存活周期的不同将内存划为几块。一般 Java 堆分为新生代和老年代，这样就可以根据各个年代的特点采用适当的收集算法。
   - 在新生袋中每次垃圾手机时都会由大批对象死去，只有少量存活，那就用复制算法，只需要付出少量存活对象的复制成本就可以。老年代中对象存活率高、没有额外担保，所以必须使用“标记-清理”或者“标记整理算法。

参考文章：[jvm 系列(三):GC 算法 垃圾收集器——纯洁的微笑]

3）GC 什么时候开始？

答：GC 经常发生的区域是堆区，堆区还可以细分为新生代、老年代，新生代还分为一个 Eden 区和两个 Survivor 区。

1. 对象优先在 Eden 中分配，当 Eden 中没有足够空间时，虚拟机将发生一次 Minor GC，因为 Java 大多数对象都是朝生夕灭，所以 Minor GC 非常频繁，而且速度也很快；
2. Full GC，发生在老年代的 GC，当老年代没有足够的空间时即发生 Full GC，发生 Full GC 一般都会有一次 Minor GC。大对象直接进入老年代，如很长的字符串数组，虚拟机提供一个-XX:PretenureSizeThreadhold 参数，令大于这个参数值的对象直接在老年代中分配，避免在 Eden 区和两个 Survivor 区发生大量的内存拷贝；
3. 发生 Minor GC 时，虚拟机会检测之前每次晋升到老年代的平均大小是否大于老年代的剩余空间大小，如果大于，则进行一次 Full GC，如果小于，则查看 HandlePromotionFailure 设置是否允许担保失败，如果允许，那只会进行一次 Minor GC，如果不允许，则改为进行一次 Full GC。

4）引用的分类？

答：

- 强引用：通过 new 出来的引用，只要强引用还存在，则不会回收。
- 软引用：通过 SoftReference 类来实现，用来描述一些有用但非必须的对象。在系统将要发生内存溢出异常之前，会把这些对象回收了，如果这次回收还是内存不够的话，才抛出内存溢出异常。
- 弱引用：非必须对象，通过 WeakReference 类来实现，被弱引用引用的对象，只要已发生 GC 就会把它干掉。
- 虚引用：通过 PhantomReference 类来实现，无法通过徐引用获得对象的实例，唯一作用就是在这个对象被 GC 时会收到一个系统通知。

扩展阅读：重新认识 java（一） ---- 万物皆对象 ，文章中有对这四个引用有详细的描述，还有一些典型的应用，这里就不摘过来啦...

5）垃圾收集器？

解析：如果说收集算法是内存回收的方法论，垃圾收集器就是内存回收的具体实现

答：

1. Serial 收集器

串行收集器是最古老，最稳定以及效率高的收集器，可能会产生较长的停顿，只使用一个线程去回收。新生代、老年代使用串行回收；新生代复制算法、老年代标记-压缩；垃圾收集的过程中会 Stop The World（服务暂停）

参数控制： -XX:+UseSerialGC 串行收集器

错误!未指定文件名。

2.ParNew 收集器

ParNew 收集器 ParNew 收集器其实就是 Serial 收集器的多线程版本。新生代并行，老年代串行；新生代复制算法、老年代标记-压缩

参数控制：

-XX:+UseParNewGC ParNew 收集器
-XX:ParallelGCThreads 限制线程数量

错误!未指定文件名。

3.Parallel Scavenge 收集器

Parallel Scavenge 收集器类似 ParNew 收集器，Parallel 收集器更关注系统的吞吐量。可以通过参数来打开自适应调节策略，虚拟机会根据当前系统的运行

情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量；也可以通过参数控制 GC 的时间不大于多少毫秒或者比例；新生代复制算法、老年代标记-压缩

参数控制： -XX:+UseParallelGC 使用 Parallel 收集器+ 老年代串行

4.Parallel Old 收集器

Parallel Old 是 Parallel Scavenge 收集器的老年代版本，使用多线程和“标记－整理”算法。这个收集器是在 JDK 1.6 中才开始提供

参数控制： -XX:+UseParallelOldGC 使用 Parallel 收集器+ 老年代并行

5.CMS 收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的 Java 应用都集中在互联网站或 B/S 系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。

从名字（包含“Mark Sweep”）上就可以看出 CMS 收集器是基于“标记-清除”算法实现的，它的运作过程相对于前面几种收集器来说要更复杂一些，整个过程分为 4 个步骤，包括：

- 初始标记（CMS initial mark）
- 并发标记（CMS concurrent mark）
- 重新标记（CMS remark）
- 并发清除（CMS concurrent sweep）

其中初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，并发标记阶段就是进行 GC Roots Tracing 的过程，而重新标记阶段则是为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以总体上来说，CMS 收集器的内存回收过程是与用户线程一起并发地执行。老年代收集器（新生代使用 ParNew）

优点：并发收集、低停顿
缺点：产生大量空间碎片、并发阶段会降低吞吐量

参数控制：

-XX:+UseConcMarkSweepGC 使用 CMS 收集器
-XX:+ UseCMSCompactAtFullCollection Full GC 后，进行一次碎片整理；整理过程是独占的，会引起停顿时间变长
-XX:+CMSFullGCsBeforeCompaction 设置进行几次 Full GC 后，进行一次碎片整理
-XX:ParallelCMSThreads 设定 CMS 的线程数量（一般情况约等于可用 CPU 数量）

错误!未指定文件名。

6.G1 收集器

G1 是目前技术发展的最前沿成果之一，HotSpot 开发团队赋予它的使命是未来可以替换掉 JDK1.5 中发布的 CMS 收集器。与 CMS 收集器相比 G1 收集器有以下特点：

1. **空间整合**，G1 收集器采用标记整理算法，不会产生内存空间碎片。分配大对象时不会因为无法找到连续空间而提前触发下一次 GC。
2. **可预测停顿**，这是 G1 的另一大优势，降低停顿时间是 G1 和 CMS 的共同关注点，但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 N 毫秒的时间片段内，消耗在垃圾收集上的时间不得超过 N 毫秒，这几乎已经是实时 Java（RTSJ）的垃圾收集器的特征了。

上面提到的垃圾收集器，收集的范围都是整个新生代或者老年代，而 G1 不再是这样。使用 G1 收集器时，Java 堆的内存布局与其他收集器有很大差别，它将整个 Java 堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔阂了，它们都是一部分（可以不连续）Region 的集合。

错误!未指定文件名。

G1 的新生代收集跟 ParNew 类似，当新生代占用达到一定比例的时候，开始出发收集。和 CMS 类似，G1 收集器收集老年代对象会有短暂停顿。

收集步骤：

1、标记阶段，首先初始标记(Initial-Mark),这个阶段是停顿的(Stop the World Event)，并且会触发一次普通 Mintor GC。对应 GC log:GC pause (young) (inital-mark)

2、Root Region Scanning，程序运行过程中会回收 survivor 区(存活到老年代)，这一过程必须在 young GC 之前完成。

3、Concurrent Marking，在整个堆中进行并发标记(和应用程序并发执行)，此过程可能被 young GC 中断。在并发标记阶段，若发现区域对象中的所有对象都是垃圾，那个这个区域会被立即回收(图中打 X)。同时，并发标记过程中，会计算每个区域的对象活性(区域中存活对象的比例)。

错误!未指定文件名。

4、Remark，再标记，会有短暂停顿(STW)。再标记阶段是用来收集 并发标记阶段 产生新的垃圾(并发阶段和应用程序一同运行)；G1 中采用了比 CMS 更快的初始快照算法:snapshot-at-the-beginning (SATB)。

5、Copy/Clean up，多线程清除失活对象，会有 STW。G1 将回收区域的存活对象拷贝到新区域，清除 Remember Sets，并发清空回收区域并把它返回到空闲区域链表中。

错误!未指定文件名。

6、复制/清除过程后。回收区域的活性对象已经被集中回收到深蓝色和深绿色区域。

错误!未指定文件名。

参考文章：[jvm 系列(三):GC 算法 垃圾收集器——纯洁的微笑]

（四）其他 JVM 相关面试题整理

1）64 位 JVM 中，int 的长度是多数？

答：Java 中，int 类型变量的长度是一个固定值，与平台无关，都是 32 位或者 4 个字节。意思就是说，在 32 位 和 64 位 的 Java 虚拟机中，int 类型的长度是相同的。

2）怎样通过 Java 程序来判断 JVM 是 32 位 还是 64 位？

答：Sun 有一个 Java System 属性来确定 JVM 的位数：32 或 64：

```
sun.arch.data.model=32 // 32 bit JVM
sun.arch.data.model=64 // 64 bit JVM
```

我可以使用以下语句来确定 JVM 是 32 位还是 64 位：

System.getProperty("sun.arch.data.model")

3）32 位 JVM 和 64 位 JVM 的最大堆内存分别是多数？

答：理论上说上 32 位的 JVM 堆内存可以到达 2^32，即 4GB，但实际上会比这个小很多。不同操作系统之间不同，如 Windows 系统大约 1.5 GB，Solaris 大约 3GB。64 位 JVM 允许指定最大的堆内存，理论上可以达到 2^64，这是一个非常大的数字，实际上你可以指定堆内存大小到 100GB。甚至有的 JVM，如 Azul，堆内存到 1000G 都是可能的。

4）你能保证 GC 执行吗？

答：不能，虽然你可以调用 System.gc() 或者 Runtime.gc()，但是没有办法保证 GC 的执行。

5）怎么获取 Java 程序使用的内存？堆使用的百分比？

答：可以通过 java.lang.Runtime 类中与内存相关方法来获取剩余的内存，总内存及最大堆内存。通过这些方法你也可以获取到堆使用的百分比及堆内存的剩余空间。Runtime.freeMemory() 方法返回剩余空间的字节数，Runtime.totalMemory() 方法总内存的字节数，Runtime.maxMemory() 返回最大内存的字节数。

6）Java 中堆和栈有什么区别？

答：JVM 中堆和栈属于不同的内存区域，使用目的也不同。栈常用于保存方法帧和局部变量，而对象总是在堆上分配。栈通常都比堆小，也不会在多个线程之间共享，而堆被整个 JVM 的所有线程共享。

错误!未指定文件名。

小结：JVM 是自己之前没有去了解过得知识，所以这次写这篇文章写了很久，也学到了很多东西；在考虑要不要开微信公众号来着...

参考资料：

1. 《深入理解 Java 虚拟机》
2. 《深入分析 Java Web 技术内幕》

(四)——版本特性篇

对于 Java 各个版本的特性，特别是 Java 8 的新知识点，我们都应该有所了解。
前排申明和好文推荐：闪烁之狐 » Java5 新特性及使用 » Java6 新特性及使用 » Java7 新特性及使用 » Java8 新特性及使用(一) » Java8 新特性及使用(二)

（一）Java 5 相关知识点

参考文章：jdk 1.5 新特性

1）增强型 for 循环：

答：增强 for 循环：foreach 语句，foreach 简化了迭代器。

格式：// 增强 for 循环括号里写两个参数，第一个是声明一个变量，第二个就是需要迭代的容器

```
for( 元素类型 变量名 : Collection 集合 & 数组 ) {
    …
}
```

语法：
for (type 变量名：集合变量名) { … }

注意事项：

- 迭代变量必须在( )中定义！
- 集合变量可以是数组或实现了 Iterable 接口的集合类。

高级 for 循环和传统 for 循环的区别：

高级 for 循环在使用时，必须要明确被遍历的目标。这个目标，可以是 Collection 集合或者数组，如果遍历 Collection 集合，在遍历过程中还需要对元素进行操作，比如删除，需要使用迭代器。

如果遍历数组，还需要对数组元素进行操作，建议用传统 for 循环因为可以定义角标通过角标操作元素。如果只为遍历获取，可以简化成高级 for 循环，它

的出现为了简化书写。比起普通的 for 循环，高级 for 循环还有性能优势，因为它对数组索引的边界值只计算一次（摘自《Effective Java》第 46 条）。

高级 for 循环可以遍历 map 集合吗？

答：原则上 map 集合是无法使用增强 for 循环来迭代的，因为增强 for 循环只能针对实现了 Iterable 接口的集合进行迭代；Iterable 是 jdk5 中新定义的接口，就一个方法 iterator 方法，只有实现了 Iterable 接口的类，才能保证一定有 iterator 方法，java 有这样的限定是因为增强 for 循环内部还是用迭代器实现的，而实际上，我们可以通过某种方式来使用增强 for 循环。

```
for(Object obj : map.entrySet()) {
    Map.Entry entry = (Entry) obj;  // obj 依次表示 Entry
    System.out.println(entry.getKey() + "=" + entry.getValue());
}
```

总之，for-each 循环在简洁性和预防 Bug 方面有着传统 for 循环无法比拟的优势，并且没有性能损失。应该尽可能地使用 for-each 循环。遗憾的是，有三种常见的情况是无法使用 for-each 循环的：

1. 过滤——如果需要遍历集合，并删除选定的元素，就需要使用显式地迭代器，以便可以调用它的 remove 方法。
2. 转换——如果需要遍历列表或者数组，并取代它部分或者全部的元素值（增删、或对元素进行赋值），就需要列表迭代器或者数组索引，以便设定元素的值
3. 平行迭代——如果需要并行地遍历多个集合，就需要显式地控制迭代器或者所因变量以便所有迭代器或者索引变量都可以得到同步前移

2) 可变参数：

解析：什么意思呢？举个例子：在 JDK 1.5 之前，当我们要为一个传递多个类型相同的参数时，我们有两种方法解决，1.直接传递一个数组过去，2.有多少个参数就传递多少个参数。

例如：

```
public void printColor(String red,String green,String yellow){
}
// 或者
public void printColor(String[] colors){
}
```

这样编写方法参数虽然能够实现我们想要的效果，但是，这样是不是有点麻烦呢？再者，如果参数个数不确定，我们怎么办呢？Java JDK1.5 为我们提供的可变参数就能够完美的解决这个问题

答：

可变参数（...）：用到函数的参数上，当要操作的同一个类型元素个数不确定的时候，可是用这个方式，这个参数可以接受任意个数的同一类型的数据。

和以前接收数组不一样的是：

以前定义数组类型，需要先创建一个数组对象，再将这个数组对象作为参数传递给函数。现在，直接将数组中的元素作为参数传递即可。底层其实是将这些元素进行数组的封装，而这个封装动作，是在底层完成的，被隐藏了。所以简化了用户的书写，少了调用者定义数组的动作。

如果在参数列表中使用了可变参数，**可变参数必须定义在参数列表结尾(也就是必须是最后一个参数，否则编译会失败。)。**

如果要获取多个 int 数的和呢？可以使用将多个 int 数封装到数组中，直接对数组求和即可。

可变参数的特点：

- ① 只能出现在参数列表的最后；
- ②“…”位于变量类型和变量名之间，前后有无空格都可以;
- ③ 调用可变参数的方法时，编译器为该可变参数隐含创建一个数组，在方法体中以数组的形式访问可变参数。

```
Public int add(int  x, int... args){//也可以直接（int..args）就是说传不传都可以
        Int sum = x;
        For(int i = 0; i<=args.lengrth;i++){
                Sum+=args[i];
        }
        return sum;
}
```

实例：

```
public class VariableParameter {
      public static void main(String[] args) {
            System. out.println(add(1, 2));
            System. out.println(add(1, 2, 3));
      }
```

```
        public static int add(int x, int... args){
            int sum = x;
            for(int i = 0; i < args.length; i++){
                sum += args[i];
            }
            return sum;
        }
}
```

3）枚举

解析：关键字 enum

答：

问题：对象的某个属性的值不能是任意的，必须为固定的一组取值其中的某一个；

解决办法：

1）在 setGrade 方法中做判断，不符合格式要求就抛出异常；
2）直接限定用户的选择，通过自定义类模拟枚举的方式来限定用户的输入，写一个 Grade 类，私有构造函数，对外提供 5 个静态的常量表示类的实例；
3）jdk5 中新定义了枚举类型，专门用于解决此类问题；
4）枚举就是一个特殊的 java 类，可以定义属性、方法、构造函数、实现接口、继承类；

为什么要有枚举？

问题：要定义星期几或性别的变量，该怎么定义？假设用 1-7 分别表示星期一到星期日，但有人可能会写成 int weekday = 0;或即使使用常量方式也无法阻止意外。

枚举就是要让某个类型的变量的取值只能为若干个固定值中的一个，否则，编译器就会报错。枚举可以让编译器在编译时就可以控制源程序中填写的非法值，普通变量的方式在开发阶段无法实现这一目标。

用普通类如何实现枚举功能，定义一个 Weekday 的类来模拟枚举功能。

1、私有的构造方法。
2、每个元素分别用一个公有的静态成员变量表示。

可以有若干公有方法或抽象方法。采用抽象方法定义 nextDay 就将大量的 if.else 语句转移成了一个个独立的类

示例：定义一个 Weekday 的类来模拟枚举功能。

```
public class WeekDay {

	private WeekDay(){}

	public static final WeekDay SUN = new WeekDay();
	public static final WeekDay MON = new WeekDay();

	public WeekDay nextDay(){
		if(this == SUN){
			return MON ;
		} else{
			return SUN ;
		}
	}

	public String toString(){
		return this == SUN? "SUN":"MON" ;
	}
}
public class EnumTest {

	public static void main(String[] args) {
		WeekDay day = WeekDay.MON;
		System. out.println(day.nextDay());
		//结果：SUN
	}
}
```

使用枚举类实现

```
public class EnumTest {

	public static void main(String[] args) {
		WeekDay day = WeekDay.FRI;
		System.out.println(day);
		 //结果：FRI
		System.out.println(day.name());
		 //结果：FRI
		System.out.println(day.ordinal());
		 //结果：5
```

```
            System.out.println(WeekDay. valueOf("SUN"));
             //结果：SUN
            System.out.println(WeekDay. values().length);
             //结果：7
      }

      public enum WeekDay{
             SUN,MON ,TUE,WED, THI,FRI ,SAT;
      }
}
```

总结： 枚举是一种特殊的类，其中的每个元素都是该类的一个实例对象，例如可以调用 WeekDay.SUN.getClass().getName 和 WeekDay.class.getName()。

注意： 最后一个枚举元素后面可以加分号，也可以不加分号。

实现带有构造方法的枚举

- 枚举就相当于一个类，其中也可以定义构造方法、成员变量、普通方法和抽象方法。
- 枚举元素必须位于枚举体中的最开始部分，枚举元素列表的最后要有分号与其他成员分隔。把枚举中的成员方法或变量等放在枚举元素的前面，编译器会报告错误。
- 带构造方法的枚举：
  构造方法必须定义成私有的
  如果有多个构造方法，将根据枚举元素创建时所带的参数决定选择哪个构造方法创建对象。
  枚举元素 MON 和 MON() 的效果一样，都是调用默认的构造方法。

示例：

```
public class EnumTest {

       public static void main(String[] args) {
            WeekDay day = WeekDay.FRI;
      }

      public enum WeekDay{
             SUN(1),MON (),TUE, WED,THI ,FRI,SAT;

             private WeekDay(){
                  System. out.println("first" );
            }

             private WeekDay(int value){
                  System. out.println("second" );
```

```
            }
             //结果：
             //second
             //first
             //first
             //first
             //first
             //first
             //first
      }
}
```

实现带有抽象方法的枚举

定义枚举 TrafficLamp，实现抽象的 nextTrafficLamp 方法：每个元素分别是由枚举类的子类来生成的实例对象，这些子类采用类似内部类的方式进行定义。增加上表示时间的构造方法。

```
public class EnumTest {

    public static void main(String[] args) {
        TrafficLamp lamp = TrafficLamp.RED;
        System.out.println(lamp.nextLamp());
        //结果：GREEN
    }

    public enum TrafficLamp {
        RED(30) {
            public TrafficLamp nextLamp() {
                return GREEN;
            }
        }, GREEN(45) {
            public TrafficLamp nextLamp() {
                return YELLOW;
            }
        }, YELLOW(5) {
            public TrafficLamp nextLamp() {
                return RED;
            }
        };

        private int time;

        private TrafficLamp(int time) {
            this.time = time;
```

```
        }

        public abstract TrafficLamp nextLamp();
    }
}
```

注意：
1、枚举只有一个成员时，就可以作为一种单例的实现方式。
2、查看生成的 class 文件，可以看到内部类对应的 class 文件。

4）自动拆装箱

答：在 Java 中数据类型分为两种：基本数据类型、引用数据类型(对象)

自动装箱：把基本类型变成包装器类型，本质是调用包装器类型的 valueOf（）方法

注意：基本数据类型的数组与包装器类型数组不能互换

在 java 程序中所有的数据都需要当做对象来处理，针对 8 种基本数据类型提供了包装类，如下：

---

int → Integer
byte → Byte
short → Short
long → Long
char → Character
double → Double
float → Float
boolean → Boolean

在 jdk 1.5 以前基本数据类型和包装类之间需要相互转换：

基本---引用 Integer x = new Integer(x);
引用---基本 int num = x.intValue();

1）Integer x = 1; x = x + 1; 经历了什么过程？装箱→拆箱→装箱
2）为了优化，虚拟机为包装类提供了缓冲池，**Integer 池**的大小为 -128~127 一个字节的大小。**String 池**：Java 为了优化字符串操作也提供了一个缓冲池；

→ 享元模式（Flyweight Pattern）：享元模式的特点是，复用我们内存中已经存在的对象，降低系统创建对象实例。

自动装箱：

```
Integer num1 = 12;
```

自动拆箱：

```
System.out.println(num1 + 12);
```

基本数据类型的对象缓存：

```
Integer num1 = 12;
Integer num2 = 12;
System.out.println(num1 == num2);//ture
Integer num3 = 129;
Integer num4 = 129;
System.out.println(num3 == num4);//false
Integer num5 = Integer.valueOf(12);
Integer num6 = Integer.valueOf(12);
System.out.println(num5 == num6);//true
```

示例：

```
public class AutoBox {
	public static void main(String[] args) {
		//装箱
		Integer iObj = 3;

		//拆箱
		System. out.println(iObj + 12);
		//结果：15

		Integer i1 = 13;
		Integer i2 = 13;
		System. out.println(i1 == i2);
		//结果：true

		i1 = 137;
		i2 = 137;
		System. out.println(i1 == i2);
		//结果：false
	}
```

}

注意：
如果有很多很小的对象，并且他们有相同的东西，那就可以把他们作为一个对象。
如果还有很多不同的东西，那就可以作为外部的东西，作为参数传入。
这就是享元设计模式（flyweight）。

例如示例中的 Integer 对象，在-128~127 范围内的 Integer 对象，用的频率比较高，就会作为同一个对象，因此结果为 true。超出这个范围的就不是同一个对象，因此结果为 false。

5）泛型 Generics

答：引用泛型之后，允许指定集合里元素的类型，免去了强制类型转换，并且能在编译时刻进行类型检查的好处。Parameterized Type 作为参数和返回值，Generic 是 vararg、annotation、enumeration、collection 的基石。

泛型可以带来如下的好处总结如下：

1. 类型安全：抛弃 List、Map，使用 List、Map 给它们添加元素或者使用 Iterator 遍历时，编译期就可以给你检查出类型错误
2. 方法参数和返回值加上了 Type: 抛弃 List、Map，使用 List、Map
3. 不需要类型转换：List list = new ArrayList();
4. 类型通配符“?”： 假设一个打印 List 中元素的方法 printList,我们希望任何类型 T 的 List 都可以被打印

6）静态导入

答：**静态导入：**导入了类中的所有静态成员，简化静态成员的书写。
import 语句可以导入一个类或某个包中的所有类
import static 语句导入一个类中的某个静态方法或所有静态方法

import static java.util.Collections.*; //导入了 Collections 类中的所有静态成员

静态导入可以导入静态方法，这样就不必写类名而可以直接调用静态方法了。

例子：

原来的：

```
public class Demo12 {
    public static void main(String[] args) {
```

```
        System.out.println(Math.max(12, 15));
         System. out.println(Math.abs(3-6));
    }
}
```

使用静态导入的：

```
import static java.lang.Math.max ;
import static java.lang.Math.abs ;

public class Demo12 {
    public static void main(String[] args) {
        System.out.println(max(12, 15));
        System. out.println(abs(3-6));
    }
}
```

注意：

1、也可以通过 import static java.lang.Math.*;导入 Math 类下所有的静态方法。
2、如果将 javac 设置为了 Java5 以下，那么静态导入等 jdk1.5 的特性都会报告错误。

7）新的线程模型和并发库 Thread Framework(重要)

答： 最主要的就是引入了 java.util.concurrent 包，这个都是需要重点掌握的。

HashMap 的替代者 ConcurrentHashMap 和 ArrayList 的替代者 CopyOnWriteArrayList 在大并发量读取时采用 java.util.concurrent 包里的一些类会让大家满意 BlockingQueue、Callable、Executor、Semaphore

8）内省（Introspector）

答：是 Java 语言对 Bean 类属性、事件的一种缺省处理方法。例如类 A 中有属性 name ，那我们通过 getName,setName 来得到其值或者设置新的值。通过 getName/setName 来访问 name 属性，这就是默认的规则。Java 中提供了一套 API 用来访问某个属性的 getter /setter 方法，通过这些 API 可以使你不需要了解这个规则（但你最好还是要搞清楚），这些 API 存放于包 java.beans 中。

一般的做法是通过类 Introspector 来获取某个对象的 BeanInfo 信息，然后通过 BeanInfo 来获取属性的描述器 （PropertyDescriptor），通过这个属性描述器就可以获取某个属性对应的 getter/setter 方法，然后我们就可以通过反射机制来 调用这些方法。

扩展阅读：java Introspector(内省) 的介绍

9）注解（Annotations）

答：

注解(Annotation)是一种应用于类、方法、参数、变量、构造器及包声明中的特殊修饰符，它是一种由 JSR-175 标准选择用来描述元数据的一种工具。Java 从 Java5 开始引入了注解。在注解出现之前，程序的元数据只是通过 java 注释和 javadoc，但是注解提供的功能要远远超过这些。注解不仅包含了元数据，它还可以作用于程序运行过程中、注解解释器可以通过注解决定程序的执行顺序。

比如，下面这段代码：

```
@Override
public String toString() {
    return "This is String.";
}
```

上面的代码中，我重写了 toString()方法并使用了@Override 注解。但是，即使我们不使用@Override 注解标记代码，程序也能够正常执行。那么，该注解表示什么？这么写有什么好处吗？事实上，@Override 告诉编译器这个方法是一个重写方法(描述方法的元数据)，如果父类中不存在该方法，编译器便会报错，提示该方法没有重写父类中的方法。如果我不小心拼写错误，例如将 toString()写成了 toStrring(){double r}，而且我也没有使用@Override 注解，那程序依然能编译运行。但运行结果会和我期望的大不相同。现在我们了解了什么是注解，并且使用注解有助于阅读程序。

为什么要引入注解？

使用注解之前(甚至在使用之后)，XML 被广泛的应用于描述元数据。不知何时开始一些应用开发人员和架构师发现 XML 的维护越来越糟糕了。他们希望使用一些和代码紧耦合的东西，而不是像 XML 那样和代码是松耦合的(在某些情况下甚至是完全分离的)代码描述。如果你在 Google 中搜索“XML vs. annotations”，会看到许多关于这个问题的辩论。最有趣的是 XML 配置其实就是为了分离代码和配置而引入的。上述两种观点可能会让你很疑惑，两者观点似乎构成了一种循环，但各有利弊。下面我们通过一个例子来理解这两者的区别。

假如你想为应用设置很多的常量或参数，这种情况下，XML 是一个很好的选择，因为它不会同特定的代码相连。如果你想把某个方法声明为服务，那么使用注解会更好一些，因为这种情况下需要注解和方法紧密耦合起来，开发人员也必须认识到这点。

另一个很重要的因素是注解定义了一种标准的描述元数据的方式。在这之前，开发人员通常使用他们自己的方式定义元数据。例如，使用标记接口，注释，transient 关键字等等。每个程序员按照自己的方式定义元数据，而不像注解这种标准的方式。

目前，许多框架将 XML 和 Annotation 两种方式结合使用，平衡两者之间的利弊。

参考文章（更多注解戳这里）:Java 注解的理解和应用

10）新增 ProcessBuilder 类

答:

ProcessBuilder 类是 Java5 在 java.lang 包中新添加的一个新类，此类用于创建操作系统进程，它提供一种启动和管理进程（也就是应用程序）的方法。在此之前，都是由 Process 类处来实现进程的控制管理。每个 ProcessBuilder 实例管理一个进程属性集。它的 start() 方法利用这些属性创建一个新的 Process 实例。start() 方法可以从同一实例重复调用，以利用相同的或相关的属性创建新的子进程。

ProcessBuilder 是一个 final 类，有两个带参数的构造方法，你可以通过构造方法来直接创建 ProcessBuilder 的对象。而 Process 是一个抽象类，一般都通过 Runtime.exec() 和 ProcessBuilder.start() 来间接创建其实例。ProcessBuilder 为进程提供了更多的控制，例如，可以设置当前工作目录，还可以改变环境参数。而 Process 类的功能相对来说简单的多。ProcessBuilder 类不是同步的。如果多个线程同时访问一个 ProcessBuilder，而其中至少一个线程从结构上修改了其中一个属性，它必须保持外部同步。

若要使用 ProcessBuilder 创建一个进程，只需要创建 ProcessBuilder 的一个实例，指定该进程的名称和所需参数。要执行此程序，调用该实例上的 start() 即可。下面是一个执行打开 Windows 记事本的例子。注意它将要编辑的文件名指定为一个参数。

```
class PBDemo {

    public static void main(String args[]) {
        try {
```

```
            ProcessBuilder proc = new ProcessBuilder("notepad.exe",
"testfile");
            proc.start();
        } catch (Exception e) {
            System.out.println("Error executing notepad.");
        }
    }

}
```

参考文章：Java5 新特性及使用

11）新增 Formatter 格式化器(Formatter)

Formatter 类是 Java5 中新增的 printf-style 格式化字符串的解释器，它提供对布局和对齐的支持，提供了对数字，字符串和日期/时间数据的常用格式以及特定于语言环境的输出。常见的 Java 类型，如 byte，java.math.BigDecimal 和 java.util.Calendar 都支持。 通过 java.util.Formattable 接口提供了针对任意用户类型的有限格式定制。

更详细的介绍见这里。主要使用方法的代码示例如下：

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.text.MessageFormat;
import java.text.SimpleDateFormat;
import java.util.*;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * 格式化测试使用的示例类.
 *
 * @author blinkfox on 2017-11-28.
 */
public class FormatTester {

    private static final Logger log =
LoggerFactory.getLogger(FormatTester.class);

    /**
     * 格式化.
     */
```

```
    private static void formatter() {
        StringBuilder sb = new StringBuilder();
        Formatter formatter = new Formatter(sb, Locale.US);

        // 可重新排序输出.
        formatter.format("%n%4$2s %3$2s %2$2s %1$2s %n", "a", "b",
"c", "d"); // -> " d  c  b  a"
        formatter.format(Locale.FRANCE, "e = %+10.4f", Math.E); // ->
"e =    +2,7183"
        formatter.format("%nAmount gained or lost since last
statement: $ %(,.2f", 6217.58);
        // -> "Amount gained or lost since last statement:
$ 6,217.58"

        log.info("打印出格式化后的字符串:{}", formatter);
        formatter.close();
    }

    /**
     * printf 打印.
     */
    private static void printf() {
        String filename = "testfile";
        try (FileReader fileReader = new FileReader(filename)) {
            BufferedReader reader = new BufferedReader(fileReader);
            String line;
            int i = 1;
            while ((line = reader.readLine()) != null) {
                System.out.printf("Line %d: %s%n", i++, line);
            }
        } catch (Exception e) {
            System.err.printf("Unable to open file named '%s': %s",
filename, e.getMessage());
        }
    }

    /**
     * stringFormat 使用.
     */
    private static void stringFormat() {
        // 格式化日期.
        Calendar c = new GregorianCalendar(1995, Calendar.MAY, 23);
        String s = String.format("Duke's
Birthday: %1$tm %1$te,%1$tY", c);
```

```
        // -> s == "Duke's Birthday: May 23, 1995"
        log.info(s);
    }

    /**
     * 格式化消息.
     */
    private static void messageFormat() {
        String msg = "欢迎光临，当前（{0}）等待的业务受理的顾客有{1}
位，请排号办理业务！";
        MessageFormat mf = new MessageFormat(msg);
        String fmsg = mf.format(new Object[]{new Date(), 35});
        log.info(fmsg);
    }

    /**
     * 格式化日期.
     */
    private static void dateFormat() {
        String str = "2010-1-10 17:39:21";
        SimpleDateFormat format = new
SimpleDateFormat("yyyyMMddHHmmss");
        try {
            log.info("格式化后的日期:{}",
format.format(format.parse(str)));
        } catch (Exception e) {
            log.error("日期格式化出错！", e);
        }
    }

    public static void main(String[] args) {
        formatter();
        stringFormat();
        messageFormat();
        dateFormat();
        printf();
    }

}
```

参考文章：[Java5 新特性及使用]

12）新增 Scanner 类（Scanner）

java.util.Scanner 是 Java5 的新特征，主要功能是简化文本扫描，但这个类最实用的地方还是在获取控制台输入。

(1).Scanner 概述

可以从字符串(Readable)、输入流、文件、Channel 等来直接构造 Scanner 对象，有了 Scanner 了，就可以逐段（根据正则分隔式）来扫描整个文本，并对扫描后的结果做想要的处理。

Scanner 默认使用空格作为分割符来分隔文本，但允许你使用 useDelimiter(Pattern pattern) 或 useDelimiter(String pattern) 方法来指定新的分隔符。

主要 API 如下：

- delimiter(): 返回此 Scanner 当前正在用于匹配分隔符的 Pattern。
- hasNext(): 判断扫描器中当前扫描位置后是否还存在下一段。
- hasNextLine(): 如果在此扫描器的输入中存在另一行，则返回 true。
- next(): 查找并返回来自此扫描器的下一个完整标记。
- nextLine(): 此扫描器执行当前行，并返回跳过的输入信息。

(2).扫描控制台输入

当通过 new Scanner(System.in) 创建了一个 Scanner 实例时，控制台会一直等待输入，直到敲回车键结束，把所输入的内容传给 Scanner，作为扫描对象。如果要获取输入的内容，则只需要调用 Scanner 的 nextLine() 方法即可。

```
/**
* 扫描控制台输入.
*
* @author blinkfox 2017-11-28
*/
public class ScannerTest {

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
```

```
        System.out.println("请输入字符串：");
        while (true) {
            String line = s.nextLine();
            if (line.equals("exit")) break;
            System.out.println(">>>" + line);
        }
    }

}
```

(3).其它示例

该示例中会从 myNumbers 文件中读取长整型 long 的数据。

```
Scanner sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
    long aLong = sc.nextLong();
}
```

以下示例可以使用除空格之外的分隔符来从一个字符串中读取几个条目：

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

将输出：

```
1
2
red
blue
```

参考文章：[Java5 新特性及使用]

13) StringBuilder

StringBuilder 也是 Java5 中新增的类，主要用来代替 + 号和 StringBuffer 来更加高效的拼接字符串。StringBuffer 与 StringBuilder 都是继承于 AbstractStringBuilder，主要的区别就是 StringBuffer 的函数上都有 synchronized 关键字，保证线程安全。

关于 StringBuilder 的使用这里就不再详细介绍了，网上文章也有很多。总之，对于动态字符串的拼接推荐使用 StringBuilder。静态字符串的拼接直接使用 + 号或者字符串的 concat(String str) 方法，甚至也使用 StringBuilder 亦可。

参考文章：Java5 新特性及使用

（二）Java 6 相关知识点

关于 JDK 1.6 的新特性，了解一下就可以了...如果有兴趣深入研究的童鞋，右转这里：Java6 新特性及使用

1）Desktop 类和 SystemTray 类：

答：

在 JDK6 中 ，AWT 新增加了两个类:Desktop 和 SystemTray 。

前者可以用来打开系统默认浏览器浏览指定的 URL,打开系统默认邮件客户端给指定的邮箱发邮件,用默认应用程序打开或编辑文件(比如,用记事本打开以 txt 为后缀名的文件),用系统默认的打印机打印文档;

后者可以用来在系统托盘区创建一个托盘程序.

2）使用 JAXB2 来实现对象与 XML 之间的映射

答：

JAXB 是 Java Architecture for XML Binding 的缩写，可以将一个 Java 对象转变成为 XML 格式，反之亦然。

我们把对象与关系数据库之间的映射称为 ORM, 其实也可以把对象与 XML 之间的映射称为 OXM(Object XML Mapping). 原来 JAXB 是 Java EE 的一部分，在 JDK6 中，SUN 将其放到了 Java SE 中，这也是 SUN 的一贯做法。

JDK6 中自带的这个 JAXB 版本是 2.0, 比起 1.0(JSR 31)来，JAXB2(JSR 222)用 JDK5 的新特性 Annotation 来标识要作绑定的类和属性等，这就极大简化了开发的工作量。

实际上，在 Java EE 5.0 中，EJB 和 Web Services 也通过 Annotation 来简化开发工作。另外,JAXB2 在底层是用 StAX(JSR 173)来处理 XML 文档。除了 JAXB 之外，我们还可以通过 XMLBeans 和 Castor 等来实现同样的功能。

3）理解 StAX

答：

StAX(JSR 173)是 JDK6.0 中除了 DOM 和 SAX 之外的又一种处理 XML 文档的 API。

StAX 的来历 ：在 JAXP1.3(JSR 206)有两种处理 XML 文档的方法:DOM(Document Object Model)和 SAX(Simple API for XML).

由 于 JDK6.0 中的 JAXB2(JSR 222)和 JAX-WS 2.0(JSR 224)都会用到 StAX，所以 Sun 决定把 StAX 加入到 JAXP 家族当中来，并将 JAXP 的版本升级到 1.4(JAXP1.4 是 JAXP1.3 的维护版本). JDK6 里面 JAXP 的版本就是 1.4. 。

StAX 是 The Streaming API for XML 的缩写，一种利用拉模式解析(pull-parsing)XML 文档的 API.StAX 通过提供一种基于事件迭代器(Iterator)的 API 让 程序员去控制 xml 文档解析过程,程序遍历这个事件迭代器去处理每一个解析事件，解析事件可以看做是程序拉出来的，也就是程序促使解析器产生一个解析事件，然后处理该事件，之后又促使解析器产生下一个解析事件，如此循环直到碰到文档结束符；

SAX 也是基于事件处理 xml 文档，但却是用推模式解析，解析器解析完整个 xml 文档后，才产生解析事件，然后推给程序去处理这些事件；DOM 采用的方式是将整个 xml 文档映射到一颗内存树，这样就可以很容易地得到父节点和子结点以及兄弟节点的数据，但如果文档很大，将会严重影响性能。

4）使用 Compiler API

答：

现在我们可以用 JDK6 的 Compiler API(JSR 199)去动态编译 Java 源文件，Compiler API 结合反射功能就可以实现动态的产生 Java 代码并编译执行这些代码，有点动态语言的特征。

这个特性对于某些需要用到动态编译的应用程序相当有用，比如 JSP Web Server，当我们手动修改 JSP 后，是不希望需要重启 Web Server 才可以看到效果的，这时候我们就可以用 Compiler API 来实现动态编译 JSP 文件，当然，现在的 JSP Web Server 也是支持 JSP 热部署的，现在的 JSP Web Server 通过在运行期间通过 Runtime.exec 或 ProcessBuilder 来调用 javac 来编译代码，这

种方式需要我们产生另一个进程去 做编译工作，不够优雅而且容易使代码依赖与特定的操作系统；Compiler API 通过一套易用的标准的 API 提供了更加丰富的方式去做动态编译,而且是跨平台的。

5）轻量级 Http Server API

答：

JDK6 提供了一个简单的 Http Server API,据此我们可以构建自己的嵌入式 Http Server,它支持 Http 和 Https 协议,提供了 HTTP1.1 的部分实现，没有被实现的那部分可以通过扩展已有的 Http Server API 来实现,程序员必须自己实现 HttpHandler 接口,HttpServer 会调用 HttpHandler 实现类的回调方法来处理客户端请求,在 这里,我们把一个 Http 请求和它的响应称为一个交换,包装成 HttpExchange 类,HttpServer 负责将 HttpExchange 传给 HttpHandler 实现类的回调方法.

6）插入式注解处理 API(Pluggable Annotation Processing API)

答：

插入式注解处理 API(JSR 269)提供一套标准 API 来处理 Annotations(JSR 175)

实 际上 JSR 269 不仅仅用来处理 Annotation,我觉得更强大的功能是它建立了 Java 语言本身的一个模型,它把 method, package, constructor, type, variable, enum, annotation 等 Java 语言元素映射为 Types 和 Elements(两者有什么区别?), 从而将 Java 语言的语义映射成为对象, 我们可以在 javax.lang.model 包下面可以看到这些类. 所以我们可以利用 JSR 269 提供的 API 来构建一个功能丰富的元编程(metaprogramming)环境.

JSR 269 用 Annotation Processor 在编译期间而不是运行期间处理 Annotation, Annotation Processor 相当于编译器的一个插件,所以称为插入式注解处理.如果 Annotation Processor 处理 Annotation 时(执行 process 方法)产生了新的 Java 代码,编译器会再调用一次 Annotation Processor,如果第二次处理还有新代码产生,就会接着调用 Annotation Processor,直到没有新代码产生为止.每执行一次 process()方法被称为一个"round",这样整个 Annotation processing 过程可以看作是一个 round 的序列.

JSR 269 主要被设计成为针对 Tools 或者容器的 API. 举个例子,我们想建立一套基于 Annotation 的单元测试框架(如 TestNG),在测试类里面用 Annotation 来标识测试期间需要执行的测试方法。

7）用 Console 开发控制台程序

JDK6 中提供了 java.io.Console 类专用来访问基于字符的控制台设备. 你的程序如果要与 Windows 下的 cmd 或者 Linux 下的 Terminal 交互,就可以用 Console 类代劳. 但我们不总是能得到可用的 Console, 一个 JVM 是否有可用的 Console 依赖于底层平台和 JVM 如何被调用. 如果 JVM 是在交互式命令行(比如 Windows 的 cmd)中启动的,并且输入输出没有重定向到另外的地方,那么就可以得到一个可用的 Console 实例.

8）对脚本语言的支持

如: ruby, groovy, javascript.

9）Common annotations

Common annotations 原本是 Java EE 5.0(JSR 244)规范的一部分，现在 SUN 把它的一部分放到了 Java SE 6.0 中.随着 Annotation 元数据功能(JSR 175)加入到 Java SE 5.0 里面，很多 Java 技术(比如 EJB,Web Services)都会用 Annotation 部分代替 XML 文件来配置运行参数（或者说是支持声明式编程,如 EJB 的声明式事务）, 如果这些技术为通用目的都单独定义了自己的 Annotations,显然有点重复建设, 所以,为其他相关的 Java 技术定义一套公共的 Annotation 是有价值的，可以避免重复建设的同时，也保证 Java SE 和 Java EE 各种技术的一致性。

10）Java DB(Derby)

从 JDK6 开始，JDK 目录中新增了一个名为 db 的目录。这便是 Java 6 的新成员：Java DB。这是一个纯 Java 实现、开源的数据库管理系统（DBMS），源于 Apache 软件基金会（ASF）名下的项目 Derby。它只有 2MB 大小，对比动辄上 G 的数据库来说可谓袖珍。但这并不妨碍 Derby 功能齐备，支持几乎大部分的数据库应用所需要的特性。JDK6.0 里面带的这个 Derby 的版本是 10.2.1.7,支持存储过程和触发器；有两种运行模式，一种是作为嵌入式数据库，另一种是作为网络数据库。前者的数据库服务器和客户端都在同一个 JVM 里面运行，后者允许数据库服务器端和客户端不在同一个 JVM 里面，而且允许这两者在不同的物理机器上。值得注意的是 JDK6 里面的这个 Derby 支持 JDK6 的新特性 JDBC 4.0 规范(JSR 221)。

11）JDBC 4.0

在 Java SE 6 所提供的诸多新特性和改进中，值得一提的是为 Java 程序提供数据库访问机制的 JDBC 版本升级到了 4.0，这个以 JSR-221 为代号的版本，提供了更加便利的代码编写机制及柔性，并且支持更多的数据类型。JDBC4.0 主要有以下改进和新特性。

- 自动加载 java.sql.Driver，而不需要再调用 class.forName；
- 添加了 java.sql.RowId 数据类型用来可以访问 sql rowid ；
- 添加了 National Character Set 的支持；
- 增强了 BLOB 和 CLOB 的支持功能；
- SQL/XML 和 XML 支持；
- Wrapper Pattern；
- SQLException 增强；
- Connection 和 Statement 接口增强；
- New Scalar Funtions；
- JDBC API changes。

（三）JAVA 7 相关知识点

之前已经写过一篇详细介绍 Java 7 特性的文章了，这里就直接黏了：[Java 7 新特性]

1）Diamond Operator

类型判断是一个人特殊的烦恼，入下面的代码：

```
Map<String,List<String>> anagrams = new
HashMap<String,List<String>>();
```

通过类型推断后变成：

```
Map<String,List<String>> anagrams = new HashMap<>();
```

注：这个<>被叫做 diamond(钻石)运算符，Java 7 后这个运算符从引用的声明中推断类型。

2）在 switch 语句中使用字符串

switch 语句可以使用原始类型或枚举类型。Java 引入了另一种类型，我们可以在 switch 语句中使用：字符串类型。

说我们有一个根据其地位来处理贸易的要求。直到现在，我们使用 if-其他语句来完成这个任务。

```
private voidprocessTrade(Trade t){

            String status = t.getStatus();

            if(status.equalsIgnoreCase(NEW)) {

                  newTrade(t);

            } else if(status.equalsIgnoreCase(EXECUTE)) {

                  executeTrade(t);

            } else if(status.equalsIgnoreCase(PENDING)) {

                  pendingTrade(t);

            }

}
```

这种处理字符串的方法是粗糙的。在 Java 中，我们可以使用增强的 switch 语句来改进程序，该语句以 String 类型作为参数。

```
public voidprocessTrade(Trade t) {
    String status = t.getStatus();
    switch(status) {
        caseNEW:
            newTrade(t);
            break;
```

```
        caseEXECUTE:
            executeTrade(t);
            break;
        casePENDING:
            pendingTrade(t);
            break;
         default:
            break;
    }
}
```

在上面的程序中，状态字段总是通过使用 String.equals() 与案例标签来进行比较。

3) 自动资源管理

Java 中有一些资源需要手动关闭，例如 Connections，Files，Input/OutStreams 等。通常我们使用 try-finally 来关闭资源：

```
public voidoldTry() {

    try{

        fos= newFileOutputStream("movies.txt");

        dos= newDataOutputStream(fos);

        dos.writeUTF("Java 7 Block Buster");

    } catch(IOException e) {

        e.printStackTrace();

    } finally{

        try{

            fos.close();

            dos.close();

        } catch(IOException e) {

            // log the exception
```

```
        }
    }
}
```

然而，在 Java 7 中引入了另一个很酷的特性，可以自动管理资源。它的操作也很简单，我们所要做的就是在 try 块中申明资源如下：

```
try(resources_to_be_cleant){
   // your code
}
```

以上方法与旧的 try-finally 能最终写成下面的代码：

```
public voidnewTry() {
    try(FileOutputStream fos = newFileOutputStream("movies.txt");
        DataOutputStream dos = newDataOutputStream(fos)) {
        dos.writeUTF("Java 7 Block Buster");
    } catch(IOException e) {
        // log the exception
    }
}
```

上面的代码也代表了这个特性的另一个方面：处理多个资源。FileOutputStream 和 DataOutputStream 在 try 语句中一个接一个地含在语句中，每一个都用分号(;)分隔符分隔开。我们不必手动取消或关闭流，因为当空间存在 try 块时，它们将自动关闭。

在后台，应该自动关闭的资源必须试验 java.lang.AutoCloseable 接口。

任何实现 AutoCloseable 接口的资源都可以作为自动资源管理的候选。AutoCloseable 是 java.io.Closeable 接口的父类，JVM 会在程序退出 try 块后调用一个方法 close()。

4）带下划线的数字文本

数字文字绝对是对眼睛的一种考验。我相信，如果你给了一个数字，比如说，十个零，你就会像我一样数零。如果不计算从右到左的位置，识别一个文字的话，就很容易出错，而且很麻烦。Not anymore。Java 在识别位置时引入了下划线。例如，您可以声明 1000，如下所示：

```
int thousand =  1_000;
```

或 1000000(一百万)如下:

```
int million  =  1_000_000
```

请注意，这个版本中也引入了二进制文字-例如“0b1”-因此开发人员不必再将它们转换为十六进制。

5）改进的异常处理

在异常处理区域有几处改进。Java 引入了多个 catch 功能，以使用单个抓到块捕获多个异常类型。

假设您有一个方法，它抛出三个异常。在当前状态下，您将分别处理它们，如下所示：

```
public voidoldMultiCatch() {

    try{

        methodThatThrowsThreeExceptions();

    } catch(ExceptionOne e) {

        // log and deal with ExceptionOne

    } catch(ExceptionTwo e) {

        // log and deal with ExceptionTwo

    } catch(ExceptionThree e) {

        // log and deal with ExceptionThree

    }
```

```
}
```

在一个 catch 块中逐个捕获一个连续的异常，看起来很混乱。我还看到了捕获十几个异常的代码。这是非常低效和容易出错的。Java 为解决这只丑小鸭带来了新的语言变化。请参阅下面的方法 oldMultiCatch 方法的改进版本：

```
public voidnewMultiCatch() {

    try{

        methodThatThrowsThreeExceptions();

    } catch(ExceptionOne | ExceptionTwo | ExceptionThree e) {

        // log and deal with all Exceptions

    }

}
```

多个异常通过使用 “|” 操作符在一个 catch 块中捕获。这样，您不必编写数十个异常捕获。但是，如果您有许多属于不同类型的异常，那么您也可以使用“多个 catch 块”块。下面的代码片段说明了这一点：

```
public voidnewMultiMultiCatch() {

    try{

        methodThatThrowsThreeExceptions();

    } catch(ExceptionOne e) {

        // log and deal with ExceptionOne

    } catch(ExceptionTwo | ExceptionThree e) {

        // log and deal with ExceptionTwo and ExceptionThree

    }

}
```

在上面的例子中，在和 ExceptionThree 属于不同的层次结构，因此您希望以不同的方式处理它们，但使用一个抓到块。

6）New file system API(NIO 2.0)

那些使用 Java 的人可能还记得框架引起的头痛。在操作系统或多文件系统之间无缝地工作从来都不是一件容易的事情.。有些方法，例如删除或重命名，在大多数情况下都是出乎意料的。使用符号链接是另一个问题。实质上 API 需要大修。

为了解决上述问题，Java 引入了一个新的 API，并在许多情况下引入了新的 api。

在 NIO2.0 提出了许多增强功能。在处理多个文件系统时，它还引入了新的类来简化开发人员的生活。

Working With Path（使用路径）：

新的 **java.nio.file** 由包和接口组成例如：
Path,Paths,FileSystem,FileSystems 等等。

路径只是对文件路径的简单引用。它与 java.io.File 等价(并具有更多的特性)。下面的代码段显示了如何获取对“临时”文件夹的路径引用：

```
public voidpathInfo() {

    Path path= Paths.get("c:\Temp\temp");

    System.out.println("Number of Nodes:"+ path.getNameCount());

    System.out.println("File Name:"+ path.getFileName());

    System.out.println("File Root:"+ path.getRoot());

    System.out.println("File Parent:"+ path.getParent());

}
```

最终控制台的输出将是:

```
Number of Nodes:2

File Name:temp.txt

File Root:c:

File Parent:c:Temp
```

删除文件或目录就像在文件中调用 delete 方法(注意复数)一样简单。在类公开两个删除方法，一个抛出 NoSuchFileException，另一个不抛。

下面的 delete 方法调用抛出 NoSuchFileException，因此您必须处理它：

```
Files.delete(path);
```

Where as Files.deleteIfExists(path) does not throw exception (as expected) if the file/directory does not exist.

使用 **Files.deteleIfExists(path)** 则不会抛出异常。

您可以使用其他实用程序方法，例如 Files.copy(.)和 Files.move(.)来有效地对文件系统执行操作。类似地，使用 **createSymbolicLink(..)** 方法使用代码创建符号链接。

文件更改通知：

JDK 7 中最好的改善算是 File change notifications（文件更改通知）了。这是一个长期等待的特性，它最终被刻在 NIO 2.0 中。**WatchService** API 允许您在对主题(目录或文件)进行更改时接收通知事件。

具体的创建步骤就不给了，总之它的功能就跟它的名字一般，当文件发生更改的时候，能及时作出反馈。

7）Fork and Join（Fork/Join 框架）

在一个 Java 程序中有效地使用并行内核一直是一个挑战。很少有国内开发的框架将工作分配到多个核心，然后加入它们来返回结果集。Java 已经将这个特性作为 Fork/Join 框架结合了起来。

基本上，在把手头的任务变成了小任务，直到小任务简单到可以不进一步分手的情况下解决。这就像一个分而治之的算法.。在这个框架中需要注意的一个重要概念是，理想情况下，没有工作线程是空闲的。他们实现了一个 work-stealing 算法，在空闲的工人“偷”工作从那些工人谁是忙。

支持 Fork-Join 机制的核心类是 ForkJoinPool 和 ForkJoinTask。

什么是 Fork/Join 框架：

Java7 提供的一个用于并行执行任务的框架，是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

Fork/Join 的运行流程图如下：

工作窃取算法：

工作窃取（work-stealing）算法是指某个线程从其他队列里窃取任务来执行。工作窃取的运行流程图如下：

工作窃取算法的优点是充分利用线程进行并行计算，并减少了线程间的竞争，其缺点是在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且消耗了更多的系统资源，比如创建多个线程和多个双端队列。

Fork/Join 框架使用示例：

让我们通过一个简单的需求来使用下 Fork / Join 框架，需求是：计算 1 + 2 + 3 + 4 的结果。

使用 Fork/Join 框架首先要考虑到的是如何分割任务，如果我们希望每个子任务最多执行两个数的相加，那么我们设置分割的阈值是 2，由于是 4 个数字相加，所以 Fork/Join 框架会把这个任务 fork 成两个子任务，子任务一负责计算 1 + 2，子任务二负责计算 3 + 4，然后再 join 两个子任务的结果。

因为是有结果的任务，所以必须继承 RecursiveTask ，实现代码如下：

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;
import java.util.concurrent.RecursiveTask;

/**
 * CountTask.
 *
 * @author blinkfox on 2018-01-03.
 */
public class CountTask extends RecursiveTask<Integer> {

    /** 阈值. */
    public static final int THRESHOLD = 2;

    /** 计算的开始值. */
    private int start;
```

```
/** 计算的结束值. */
private int end;

/**
 * 构造方法.
 *
 * @param start 计算的开始值
 * @param end 计算的结束值
 */
public CountTask(int start, int end) {
    this.start = start;
    this.end = end;
}

/**
 * 执行计算的方法.
 *
 * @return int 型结果
 */
@Override
protected Integer compute() {
    int sum = 0;

    // 如果任务足够小就计算任务.
    if ((end - start) <= THRESHOLD) {
        for (int i = start; i <= end; i++) {
            sum += i;
        }
    } else {
        // 如果任务大于阈值，就分裂成两个子任务来计算.
        int middle = (start + end) / 2;
        CountTask leftTask = new CountTask(start, middle);
        CountTask rightTask = new CountTask(middle + 1, end);

        // 等待子任务执行完，并得到结果，再合并执行结果.
        leftTask.fork();
        rightTask.fork();
        sum = leftTask.join() + rightTask.join();
    }
    return sum;
}

/**
 * main 方法.
```

```
     *
     * @param args 数组参数
     */
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ForkJoinPool fkPool = new ForkJoinPool();
        CountTask task = new CountTask(1, 4);
        Future<Integer> result = fkPool.submit(task);
        System.out.println("result:" + result.get());
    }

}
```

参考文章：Java7 新特性及使用
这里是 Java 7 的新特性一览表：http://www.oschina.net/news/20119/new-features-of-java-7

（四）Java 8 相关知识点

关于 Java 8 中新知识点，面试官会让你说说 Java 8 你了解多少，下面分享一下我收集的 Java 8 新增的知识点的内容，前排申明引用自：Java8 新特性及使用

1）接口默认方法和静态方法

Java 8 用默认方法与静态方法这两个新概念来扩展接口的声明。与传统的接口又有些不一样，它允许在已有的接口中添加新方法，而同时又保持了与旧版本代码的兼容性。

1.接口默认方法

默认方法与抽象方法不同之处在于抽象方法必须要求实现，但是默认方法则没有这个要求。相反，每个接口都必须提供一个所谓的默认实现，这样所有的接口实现者将会默认继承它（如果有必要的话，可以覆盖这个默认实现）。让我们看看下面的例子：

```
private interface Defaulable {
    // Interfaces now allow default methods, the implementer may or
    // may not implement (override) them.
    default String notRequired() {
        return "Default implementation";
```

```
    }
}

private static class DefaultableImpl implements Defaulable {
}

private static class OverridableImpl implements Defaulable {
    @Override
    public String notRequired() {
        return "Overridden implementation";
    }
}
```

Defaulable 接口用关键字 default 声明了一个默认方法 notRequired()，Defaulable 接口的实现者之一 DefaultableImpl 实现了这个接口，并且让默认方法保持原样。Defaulable 接口的另一个实现者 OverridableImpl 用自己的方法覆盖了默认方法。

1.1 多重继承的冲突说明：

由于同一个方法可以从不同的接口引入，自然而然的会有冲突的现象，规则如下：

- 一个声明在类里面的方法优先于任何默认方法
- 优先选取最具体的实现

```
public interface A {

    default void hello() {
        System.out.println("Hello A");
    }

}
public interface B extends A {

    default void hello() {
        System.out.println("Hello B");
    }

}
public class C implements A, B {

    public static void main(String[] args) {
        new C().hello(); // 输出 Hello B
    }
```

```
}
```

1.2 优缺点：

- **优点**: 可以在不破坏代码的前提下扩展原有库的功能。它通过一个很优雅的方式使得接口变得更智能，同时还避免了代码冗余，并且扩展类库。
- **缺点**: 使得**接口作为协议，类作为具体实现**的界限开始变得有点模糊。

1.3 接口默认方法不能重载 Object 类的任何方法：

接口不能提供对 Object 类的任何方法的默认实现。简单地讲，每一个 java 类都是 Object 的子类，也都继承了它类中的 equals()/hashCode()/toString() 方法，那么在类的接口上包含这些默认方法是没有意义的，它们也从来不会被编译。

在 JVM 中，默认方法的实现是非常高效的，并且通过字节码指令为方法调用提供了支持。默认方法允许继续使用现有的 Java 接口，而同时能够保障正常的编译过程。这方面好的例子是大量的方法被添加到 java.util.Collection 接口中去：stream()，parallelStream()，forEach()，removeIf() 等。尽管默认方法非常强大，但是在使用默认方法时我们需要小心注意一个地方：在声明一个默认方法前，请仔细思考是不是真的有必要使用默认方法。

2.接口静态方法

Java 8 带来的另一个有趣的特性是接口可以声明（并且可以提供实现）静态方法。在接口中定义静态方法，使用 static 关键字，例如：

```
public interface StaticInterface {

    static void method() {
        System.out.println("这是 Java8 接口中的静态方法!");
    }

}
```

下面的一小段代码是上面静态方法的使用。

```
public class Main {

    public static void main(String[] args) {
        StaticInterface.method(); // 输出 这是 Java8 接口中的静态方法!
    }

}
```

Java 支持一个实现类可以实现多个接口，如果多个接口中存在同样的 static 方法会怎么样呢？如果有两个接口中的静态方法一模一样，并且一个实现类同时实现了这两个接口，此时并不会产生错误，因为 Java8 中只能通过接口类调用接口中的静态方法，所以对编译器来说是可以区分的。

2）Lambda 表达式

Lambda 表达式（也称为闭包）是整个 Java 8 发行版中最受期待的在 Java 语言层面上的改变，Lambda 允许把函数作为一个方法的参数（即：行为参数化，函数作为参数传递进方法中）。

一个 Lambda 可以由用逗号分隔的参数列表、-> 符号与函数体三部分表示。

首先看看在老版本的 Java 中是如何排列字符串的：

```
List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");
Collections.sort(names, new Comparator<String>() {

    @Override
    public int compare(String a, String b) {
        return b.compareTo(a);
    }

});
```

只需要给静态方法 Collections.sort 传入一个 List 对象以及一个比较器来按指定顺序排列。通常做法都是创建一个匿名的比较器对象然后将其传递给 sort 方法。 在 Java 8 中你就没必要使用这种传统的匿名对象的方式了，Java 8 提供了更简洁的语法，lambda 表达式：

```
Collections.sort(names, (String a, String b) -> {
    return b.compareTo(a);
});
```

看到了吧，代码变得更短且更具有可读性，但是实际上还可以写得更短：

```
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

对于函数体只有一行代码的，你可以去掉大括号{}以及 return 关键字，但是你还可以写得更短点：

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

Java 编译器可以自动推导出参数类型，所以你可以不用再写一次类型。

更多 Lambda 表达式的示例在这里：Java8 lambda 表达式 10 个示例

3）函数式接口

Lambda 表达式是如何在 Java 的类型系统中表示的呢？每一个 Lambda 表达式都对应一个类型，通常是接口类型。而**函数式接口**是指仅仅只包含一个抽象方法的接口，每一个该类型的 Lambda 表达式都会被匹配到这个抽象方法。因为**默认方法**不算抽象方法，所以你也可以给你的函数式接口添加默认方法。

我们可以将 Lambda 表达式当作任意只包含一个抽象方法的接口类型，确保你的接口一定达到这个要求，你只需要给你的接口添加 @FunctionalInterface 注解，编译器如果发现你标注了这个注解的接口有多于一个抽象方法的时候会报错的。

示例如下：

```
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}

Converter<String, Integer> converter = (from) ->
Integer.valueOf(from);
Integer converted = converter.convert("123");
System.out.println(converted); // 123
```

注意： 如果 @FunctionalInterface 如果没有指定，上面的代码也是对的。
更多参考： Java 8——Lambda 表达式、Java8 新特性及使用

4）方法引用

1.概述：

在学习了 Lambda 表达式之后，我们通常使用 Lambda 表达式来创建匿名方法。然而，有时候我们仅仅是调用了一个已存在的方法。如下：

```
Arrays.sort(strArray, (s1, s2) -> s1.compareToIgnoreCase(s2));
```

在 Java8 中，我们可以直接通过方法引用来简写 Lambda 表达式中已经存在的方法。

```
Arrays.sort(strArray, String::compareToIgnoreCase);
```

这种特性就叫做**方法引用**(Method Reference)。

方法引用是用来直接访问类或者实例的已经存在的方法或者构造方法。方法引用提供了一种引用而不执行方法的方式，它需要由兼容的函数式接口构成的目标类型上下文。计算时，方法引用会创建函数式接口的一个实例。当 Lambda 表达式中只是执行一个方法调用时，不用 Lambda 表达式，直接通过方法引用的形式可读性更高一些。方法引用是一种更简洁易懂的 Lambda 表达式。

注意: 方法引用是一个 Lambda 表达式，其中方法引用的操作符是双冒号::。

2.分类：

方法引用的标准形式是：类名::方法名。（注意：只需要写方法名，不需要写括号）

有以下四种形式的方法引用：

- 引用静态方法: ContainingClass::staticMethodName
- 引用某个对象的实例方法: containingObject::instanceMethodName
- 引用某个类型的任意对象的实例方法:ContainingType::methodName
- 引用构造方法: ClassName::new

3.示例：

使用示例如下：

```
public class Person {

    String name;

    LocalDate birthday;

    public Person(String name, LocalDate birthday) {
        this.name = name;
        this.birthday = birthday;
    }

    public LocalDate getBirthday() {
        return birthday;
    }

    public static int compareByAge(Person a, Person b) {
        return a.birthday.compareTo(b.birthday);
    }

    @Override
    public String toString() {
```

```
        return this.name;
    }
}
public class MethodReferenceTest {

    @Test
    public static void main() {
        Person[] pArr = new Person[] {
            new Person("003", LocalDate.of(2016,9,1)),
            new Person("001", LocalDate.of(2016,2,1)),
            new Person("002", LocalDate.of(2016,3,1)),
            new Person("004", LocalDate.of(2016,12,1))
        };

        // 使用匿名类
        Arrays.sort(pArr, new Comparator<Person>() {
            @Override
            public int compare(Person a, Person b) {
                return a.getBirthday().compareTo(b.getBirthday());
            }
        });

        //使用 lambda 表达式
        Arrays.sort(pArr, (Person a, Person b) -> {
            return a.getBirthday().compareTo(b.getBirthday());
        });

        //使用方法引用，引用的是类的静态方法
        Arrays.sort(pArr, Person::compareByAge);
    }

}
```

5）Steam

Java8 添加的 Stream API(java.util.stream) 把真正的函数式编程风格引入到 Java 中。这是目前为止对 Java 类库最好的补充，因为 Stream API 可以极大提供 Java 程序员的生产力，让程序员写出高效率、干净、简洁的代码。使用 Steam 写出来的代码真的能让人兴奋，这里链出之前的一篇文章：Java 8——函数式数据处理（流）

流可以是无限的、有状态的，可以是顺序的，也可以是并行的。在使用流的时候，你首先需要从一些来源中获取一个流，执行一个或者多个中间操作，然后执行一个最终操作。中间操作包括 filter、map、flatMap、peel、distinct、

sorted、limit 和 substream。终止操作包括 forEach、toArray、reduce、collect、min、max、count、anyMatch、allMatch、noneMatch、findFirst 和 findAny。 java.util.stream.Collectors 是一个非常有用的实用类。该类实现了很多归约操作，例如将流转换成集合和聚合元素。

1.一些重要方法说明：

- stream: 返回数据流，集合作为其源
- parallelStream: 返回并行数据流， 集合作为其源
- filter: 方法用于过滤出满足条件的元素
- map: 方法用于映射每个元素对应的结果
- forEach: 方法遍历该流中的每个元素
- limit: 方法用于减少流的大小
- sorted: 方法用来对流中的元素进行排序
- anyMatch: 是否存在任意一个元素满足条件（返回布尔值）
- allMatch: 是否所有元素都满足条件（返回布尔值）
- noneMatch: 是否所有元素都不满足条件（返回布尔值）
- collect: 方法是终端操作，这是通常出现在管道传输操作结束标记流的结束

2.一些使用示例：

2.1 Filter 过滤：

```
stringCollection
    .stream()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);
```

2.2 Sort 排序：

```
stringCollection
    .stream()
    .sorted()
    .filter((s) -> s.startsWith("a"))
```

```
    .forEach(System.out::println);
```

2.3 Map 映射：

```
stringCollection
    .stream()
    .map(String::toUpperCase)
    .sorted((a, b) -> b.compareTo(a))
    .forEach(System.out::println);
```

2.4 Match 匹配：

```
boolean anyStartsWithA = stringCollection
        .stream()
        .anyMatch((s) -> s.startsWith("a"));
System.out.println(anyStartsWithA);      // true

boolean allStartsWithA = stringCollection
        .stream()
        .allMatch((s) -> s.startsWith("a"));
System.out.println(allStartsWithA);      // false

boolean noneStartsWithZ = stringCollection
        .stream()
        .noneMatch((s) -> s.startsWith("z"));
System.out.println(noneStartsWithZ);      // true
```

2.5 Count 计数：

```
long startsWithB = stringCollection
        .stream()
        .filter((s) -> s.startsWith("b"))
        .count();
System.out.println(startsWithB);    // 3
```

2.6 Reduce 规约：

```
Optional<String> reduced = stringCollection
        .stream()
        .sorted()
        .reduce((s1, s2) -> s1 + "#" + s2);
reduced.ifPresent(System.out::println);
```

6）Optional

到目前为止，臭名昭著的空指针异常是导致 Java 应用程序失败的最常见原因。以前，为了解决空指针异常，Google 公司著名的 Guava 项目引入了 Optional 类，Guava 通过使用检查空值的方式来防止代码污染，它鼓励程序员写更干净的代码。受到 Google Guava 的启发，Optional 类已经成为 Java 8 类库的一部分。

Optional 实际上是个容器：它可以保存类型 T 的值，或者仅仅保存 null。Optional 提供很多有用的方法，这样我们就不用显式进行空值检测。

我们下面用两个小例子来演示如何使用 Optional 类：一个允许为空值，一个不允许为空值。

```
Optional<String> fullName = Optional.ofNullable(null);
System.out.println("Full Name is set? " + fullName.isPresent());
System.out.println("Full Name: " + fullName.orElseGet(() ->
"[none]"));
System.out.println(fullName.map(s -> "Hey " + s + "!").orElse("Hey
Stranger!"));
```

如果 Optional 类的实例为非空值的话，isPresent()返回 true，否从返回 false。为了防止 Optional 为空值，orElseGet()方法通过回调函数来产生一个默认值。map()函数对当前 Optional 的值进行转化，然后返回一个新的 Optional 实例。orElse()方法和 orElseGet()方法类似，但是 orElse 接受一个默认值而不是一个回调函数。下面是这个程序的输出：

```
Full Name is set? false
Full Name: [none]
Hey Stranger!
```

让我们来看看另一个例子：

```
Optional<String> firstName = Optional.of("Tom");
System.out.println("First Name is set? " + firstName.isPresent());
System.out.println("First Name: " + firstName.orElseGet(() ->
"[none]"));
System.out.println(firstName.map(s -> "Hey " + s + "!").orElse("Hey
Stranger!"));
System.out.println();
```

下面是程序的输出：

```
First Name is set? true
```

```
First Name: Tom
Hey Tom!
```

7）Date/Time API

Java 8 在包 java.time 下包含了一组全新的时间日期 API。新的日期 API 和开源的 Joda-Time 库差不多，但又不完全一样，下面的例子展示了这组新 API 里最重要的一些部分：

1.Clock 时钟：

Clock 类提供了访问当前日期和时间的方法，Clock 是时区敏感的，可以用来取代 System.currentTimeMillis()来获取当前的微秒数。某一个特定的时间点也可以使用 Instant 类来表示，Instant 类也可以用来创建老的 java.util.Date 对象。代码如下:

```
Clock clock = Clock.systemDefaultZone();
long millis = clock.millis();
Instant instant = clock.instant();
Date legacyDate = Date.from(instant);   // legacy java.util.Date
```

2.Timezones 时区：

在新 API 中时区使用 ZoneId 来表示。时区可以很方便的使用静态方法 of 来获取到。时区定义了到 UTS 时间的时间差，在 Instant 时间点对象到本地日期对象之间转换的时候是极其重要的。代码如下:

```
System.out.println(ZoneId.getAvailableZoneIds());
// prints all available timezone ids
ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Brazil/East");
System.out.println(zone1.getRules());
System.out.println(zone2.getRules());
// ZoneRules[currentStandardOffset=+01:00]
// ZoneRules[currentStandardOffset=-03:00]
```

3.LocalTime 本地时间：

LocalTime 定义了一个没有时区信息的时间，例如 晚上 10 点，或者 17:30:15。下面的例子使用前面代码创建的时区创建了两个本地时间。之后比较时间并以小时和分钟为单位计算两个时间的时间差。代码如下:

```
LocalTime now1 = LocalTime.now(zone1);
LocalTime now2 = LocalTime.now(zone2);
```

```
System.out.println(now1.isBefore(now2));  // false
long hoursBetween = ChronoUnit.HOURS.between(now1, now2);
long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);
System.out.println(hoursBetween);       // -3
System.out.println(minutesBetween);     // -239
```

LocalTime 提供了多种工厂方法来简化对象的创建，包括解析时间字符串。代码如下:

```
LocalTime late = LocalTime.of(23, 59, 59);
System.out.println(late);       // 23:59:59
DateTimeFormatter germanFormatter = DateTimeFormatter
        .ofLocalizedTime(FormatStyle.SHORT)
        .withLocale(Locale.GERMAN);
LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);
System.out.println(leetTime);   // 13:37
```

4.LocalDate 本地日期：

LocalDate 表示了一个确切的日期，比如 2014-03-11。该对象值是不可变的，用起来和 LocalTime 基本一致。下面的例子展示了如何给 Date 对象加减天/月/年。另外要注意的是这些对象是不可变的，操作返回的总是一个新实例。代码如下:

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
LocalDate yesterday = tomorrow.minusDays(2);
LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);
DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();

System.out.println(dayOfWeek);    // FRIDAY
```

从字符串解析一个 LocalDate 类型和解析 LocalTime 一样简单。代码如下:

```
DateTimeFormatter germanFormatter = DateTimeFormatter
        .ofLocalizedDate(FormatStyle.MEDIUM)
        .withLocale(Locale.GERMAN);
LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter);
System.out.println(xmas);   // 2014-12-24
```

5.LocalDateTime 本地日期时间：

LocalDateTime 同时表示了时间和日期，相当于前两节内容合并到一个对象上了。LocalDateTime 和 LocalTime 还有 LocalDate 一样，都是不可变的。LocalDateTime 提供了一些能访问具体字段的方法。代码如下:

```
LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31,
23, 59, 59);
DayOfWeek dayOfWeek = sylvester.getDayOfWeek();
System.out.println(dayOfWeek);      // WEDNESDAY
Month month = sylvester.getMonth();
System.out.println(month);          // DECEMBER
long minuteOfDay = sylvester.getLong(ChronoField.MINUTE_OF_DAY);
System.out.println(minuteOfDay);    // 1439
```

只要附加上时区信息，就可以将其转换为一个时间点 Instant 对象，Instant 时间点对象可以很容易的转换为老式的 java.util.Date。代码如下:

```
Instant instant = sylvester
        .atZone(ZoneId.systemDefault())
        .toInstant();
Date legacyDate = Date.from(instant);
System.out.println(legacyDate);     // Wed Dec 31 23:59:59 CET 2014
```

格式化 LocalDateTime 和格式化时间和日期一样的，除了使用预定义好的格式外，我们也可以自己定义格式。代码如下:

```
DateTimeFormatter formatter =
    DateTimeFormatter
        .ofPattern("MMM dd, yyyy - HH:mm");
LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13",
formatter);
String string = formatter.format(parsed);
System.out.println(string);     // Nov 03, 2014 - 07:13
```

和 java.text.NumberFormat 不一样的是新版的 DateTimeFormatter 是不可变的，所以它是线程安全的。

关于 Java8 中日期 API 更多的使用示例可以参考 Java 8 中关于日期和时间 API 的 20 个使用示例。

8）重复注解

自从 Java 5 引入了注解机制，这一特性就变得非常流行并且广为使用。然而，使用注解的一个限制是相同的注解在同一位置只能声明一次，不能声明多次。Java 8 打破了这条规则，引入了重复注解机制，这样相同的注解可以在同一地方声明多次。

重复注解机制本身必须用@Repeatable 注解。事实上，这并不是语言层面上的改变，更多的是编译器的技巧，底层的原理保持不变。让我们看一个快速入门的例子：

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

public class RepeatingAnnotations {

    @Target(ElementType.TYPE)
    @Retention(RetentionPolicy.RUNTIME)
    public @interface Filters {
        Filter[] value();
    }

    @Target(ElementType.TYPE)
    @Retention(RetentionPolicy.RUNTIME)
    @Repeatable(Filters.class)
    public @interface Filter {
        String value();
    };

    @Filter("filter1")
    @Filter("filter2")
    public interface Filterable {
    }

    public static void main(String[] args) {
        for(Filter filter:
Filterable.class.getAnnotationsByType(Filter.class)) {
            System.out.println(filter.value());
        }
    }

}
```

正如我们看到的，这里有个使用@Repeatable(Filters.class)注解的注解类 Filter，Filters 仅仅是 Filter 注解的数组，但 Java 编译器并不想让程序员意识到 Filters 的存在。这样，接口 Filterable 就拥有了两次 Filter（并没有提到 Filter）注解。

同时，反射相关的 API 提供了新的函数 getAnnotationsByType()来返回重复注解的类型（请注意 Filterable.class.getAnnotation(Filters.class)经编译器处理后将会返回 Filters 的实例）。

9）扩展注解的支持

Java 8 扩展了注解的上下文。现在几乎可以为任何东西添加注解：局部变量、泛型类、父类与接口的实现，就连方法的异常也能添加注解。下面演示几个例子：

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.util.ArrayList;
import java.util.Collection;

public class Annotations {

    @Retention(RetentionPolicy.RUNTIME)
    @Target({ ElementType.TYPE_USE, ElementType.TYPE_PARAMETER })
    public @interface NonEmpty {
    }

    public static class Holder<@NonEmpty T> extends @NonEmpty Object
{
        public void method() throws @NonEmpty Exception {
        }
    }

    @SuppressWarnings("unused")
    public static void main(String[] args) {
        final Holder<String> holder = new @NonEmpty Holder<String>();
        @NonEmpty Collection<@NonEmpty String> strings = new
ArrayList<>();
    }

}
```

10）Base 64

在 Java 8 中，Base64 编码已经成为 Java 类库的标准。它的使用十分简单，下面让我们看一个例子：

```
import java.nio.charset.StandardCharsets;
import java.util.Base64;

public class Base64s {

    public static void main(String[] args) {
        final String text = "Base64 finally in Java 8!";

        final String encoded =
Base64.getEncoder().encodeToString(text.getBytes(StandardCharsets.UTF
_8));
        System.out.println(encoded);

        final String decoded = new
String(Base64.getDecoder().decode(encoded), StandardCharsets.UTF_8);
        System.out.println(decoded);
    }

}
```

程序在控制台上输出了编码后的字符与解码后的字符：

```
QmFzZTY0IGZpbmFsbHkgaW4gSmF2YSA4IQ==
Base64 finally in Java 8!
```

Base64 类同时还提供了对 URL、MIME 友好的编码器与解码器（Base64.getUrlEncoder() / Base64.getUrlDecoder(), Base64.getMimeEncoder() / Base64.getMimeDecoder()）。

11）JavaFX

JavaFX 是一个强大的图形和多媒体处理工具包集合，它允许开发者来设计、创建、测试、调试和部署富客户端程序，并且和 Java 一样跨平台。从 Java8 开始，JavaFx 已经内置到了 JDK 中。关于 JavaFx 更详细的文档可参考 JavaFX 中文文档。

12）HashMap 的底层实现有变化

Java8 中，HashMap 内部实现又引入了红黑树（数组+链表+红黑树），使得 HashMap 的总体性能相较于 Java7 有比较明显的提升。

13）JVM 内存管理方面，由元空间代替了永久代。

区别：

1. 元空间并不在虚拟机中，而是使用本地内存
2. 默认情况下，元空间的大小仅受本地内存限制
3. 也可以通过-XX：MetaspaceSize 指定元空间大小

（五）Java 9 相关知识点

引用自文章：Java 9 中的 9 个新特性、Java 9 新特性概述——IBM、【译】使用示例带你提前了解 Java 9 中的新特性

1）Java 9 PEPK（JShell）

Oracle 公司（Java Library 开发者）新引进一个代表 Java Shell 的称之为“jshell” 或者 REPL（Read Evaluate Print Loop）的新工具。该工具可以被用来执行和测试任何 Java 中的结构，如 class，interface，enum，object，statements 等。使用非常简单。

JDK 9 EA（Early Access）下载地址：https://jdk9.java.net/download/

```
G:\>jshell
|  Welcome to JShell -- Version 9-ea
|  For an introduction type: /help intro
jshell> int a = 10
a ==> 10
jshell> System.out.println("a value = " + a )
a value = 10
```

2）集合工厂方法

通常，您希望在代码中创建一个集合（例如，List 或 Set ），并直接用一些元素填充它。 实例化集合，几个 “add” 调用，使得代码重复。 Java 9，添加了几种集合工厂方法：

```
Set<Integer> ints = Set.of(1, 2, 3);
List<String> strings = List.of("first", "second");
```

除了更短和更好阅读之外，这些方法也可以避免您选择特定的集合实现。 事实上，从工厂方法返回已放入数个元素的集合实现是高度优化的。这是可能的，因为它们是不可变的：在创建后，继续添加元素到这些集合会导致“UnsupportedOperationException” 。

3）接口中的私有方法

在 Java 8 中，我们可以在接口中使用默认或者静态方法提供一些实现方式，但是不能创建私有方法。

为了避免冗余代码和提高重用性，Oracle 公司准备在 Java SE 9 接口中引入私有方法。也就是说从 Java SE 9 开始，我们也能够在接口类中使用‘private’ 关键字写私有化方法和私有化静态方法。

接口中的私有方法与 class 类中的私有方法在写法上并无差异，如：

```
public interface Card{
  private Long createCardID(){
    // Method implementation goes here.
  }
  private static void displayCardDetails(){
    // Method implementation goes here.
  }
}
```

4）Java 平台级模块系统

这里只给出解决的问题，仅限了解....

Java 9 的定义功能是一套全新的模块系统。当代码库越来越大，创建复杂，盘根错节的“意大利面条式代码”的几率呈指数级的增长。这时候就得面对两个基础的问题: 很难真正地对代码进行封装, 而系统并没有对不同部分（也就是 JAR 文件）之间的依赖关系有个明确的概念。每一个公共类都可以被类路径之下任何其它的公共类所访问到, 这样就会导致无意中使用了并不想被公开访问的 API。此外，类路径本身也存在问题: 你怎么知晓所有需要的 JAR 都已经有了, 或者是不是会有重复的项呢? 模块系统把这俩个问题都给解决了。

5）进程 API

Java SE 9 迎来一些 Process API 的改进，通过添加一些新的类和方法来优化系统级进程的管控。

Process API 中的两个新接口：

- java.lang.ProcessHandle
- java.lang.ProcessHandle.Info

Process API 示例

```
ProcessHandle currentProcess = ProcessHandle.current();
System.out.println("Current Process Id: = " +
currentProcess.getPid());
```

6）Try With Resources Improvement

我们知道，Java SE 7 引入了一个新的异常处理结构：Try-With-Resources，来自动管理资源。这个新的声明结构主要目的是实现“Automatic Better Resource Management”（“自动资源管理”）。

Java SE 9 将对这个声明作出一些改进来避免一些冗长写法，同时提高可读性。

Java SE 7 示例

```
void testARM_Before_Java9() throws IOException {
    BufferedReader reader1 = new BufferedReader(new
FileReader("journaldev.txt"));
    try (BufferedReader reader2 = reader1) {
        System.out.println(reader2.readLine());
    }
}
```

Java SE 9 示例

```
void testARM_Java9() throws IOException {
    BufferedReader reader1 = new BufferedReader(new
FileReader("journaldev.txt"));
    try (reader1) {
        System.out.println(reader1.readLine());
    }
}
```

7）CompletableFuture API Improvements

在 Java SE 9 中，Oracle 公司将改进 CompletableFuture API 来解决一些 Java SE 8 中出现的问题。这些被添加的 API 将用来支持一些延时和超时操作，实用方法和更好的子类化。

```
Executor exe = CompletableFuture.delayedExecutor(50L,
TimeUnit.SECONDS);
```

这里的 delayedExecutor() 是静态实用方法，用来返回一个在指定延时时间提交任务到默认执行器的新 Executor 对象。

8）反应式流 （ Reactive Streams ）

反应式编程的思想最近得到了广泛的流行。 在 Java 平台上有流行的反应式库 RxJava 和 R eactor。反应式流规范的出发点是提供一个带非阻塞负压（ non-blocking backpressure ） 的异步流处理规范。反应式流规范的核心接口已经添加到了 Java9 中的 java.util.concurrent.Flow 类中。

Flow 中包含了 Flow.Publisher、Flow.Subscriber、Flow.Subscription 和 Flow.Processor 等 4 个核心接口。Java 9 还提供了 SubmissionPublisher 作为 Flow.Publisher 的一个实现。RxJava 2 和 Reactor 都可以很方便的 与 Flow 类的核心接口进行互操作。

9）改进的 Stream API

长期以来，Stream API 都是 Java 标准库最好的改进之一。通过这套 API 可以在集合上建立用于转换的申明管道。在 Java 9 中它会变得更好。Stream 接口中添加了 4 个新的方法：dropWhile, takeWhile, ofNullable。还有个 iterate 方法的新重载方法，可以让你提供一个 Predicate (判断条件)来指定什么时候结束迭代：

```
IntStream.iterate(1, i -> i < 100, i -> i +
1).forEach(System.out::println);
```

第二个参数是一个 Lambda，它会在当前 IntStream 中的元素到达 100 的时候返回 true。因此这个简单的示例是向控制台打印 1 到 99。

除了对 Stream 本身的扩展，Optional 和 Stream 之间的结合也得到了改进。现在可以通过 Optional 的新方法 stram 将一个 Optional 对象转换为一个(可能是空的) Stream 对象：

```
Stream<Integer> s = Optional.of(1).stream();
```

在组合复杂的 Stream 管道时，将 Optional 转换为 Stream 非常有用。

10）HTTP/2

Java 9 中有新的方式来处理 HTTP 调用。这个迟到的特性用于代替老旧的 HttpURLConnection API，并提供对 WebSocket 和 HTTP/2 的支持。注意：新的 HttpClient API 在 Java 9 中以所谓的孵化器模块交付。也就是说，这套 API 不能保证 100% 完成。不过你可以在 Java 9 中开始使用这套 API：

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest req =
   HttpRequest.newBuilder(URI.create("http://www.google.com"))
              .header("User-Agent","Java")
              .GET()
              .build();

HttpResponse<String> resp = client.send(req,
HttpResponse.BodyHandler.asString());

HttpResponse<String> resp = client.send(req,
HttpResponse.BodyHandler.asString());
```

除了这个简单的请求/响应模型之外，HttpClient 还提供了新的 API 来处理 HTTP/2 的特性，比如流和服务端推送。

11）Optional Class Improvements

在 Java SE 9 中，Oracle 公司添加了一些新的实用方法到 java.util.Optional 类里面。这里我将使用一些简单的示例来描述其中的一个：stream 方法。

如果一个值出现在给定 Optional 对象中，stream() 方法可以返回包含该值的一个顺序 Stream 对象。否则，将返回一个空 Stream。

stream() 方法已经被添加，并用来在 Optional 对象中使用，如：

```
Stream<Optional> emp = getEmployee(id)
Stream empStream = emp.flatMap(Optional::stream)
```

这里的 Optional.stream() 方法被用来转化 Employee 可选流对象 到 Employee 流中，如此我们便可以在后续代码中使用这个结果。

12）多版本兼容 JAR

我们最后要来着重介绍的这个特性对于库的维护者而言是个特别好的消息。当一个新版本的 Java 出现的时候，你的库用户要花费数年时间才会切换到这个新的版本。这就意味着库得去向后兼容你想要支持的最老的 Java 版本（许多情况下就是 Java 6 或者 7）。这实际上意味着未来的很长一段时间，你都不能在库中运用 Java 9 所提供的新特性。幸运的是，多版本兼容 JAR 功能能让你创建仅在特定版本的 Java 环境中运行库程序时选择使用的 class 版本：

```
multirelease.jar
├──── META-INF
│     └──── versions
│           └──── 9
│                 └──── multirelease
│                       └──── Helper.class
├──── multirelease
      ├──── Helper.class
      └──── Main.class
```

在上述场景中， multirelease.jar 可以在 Java 9 中使用，不过 Helper 这个类使用的不是顶层的 multirelease.Helper 这个 class，而是处在“META-INF/versions/9”下面的这个。这是特别为 Java 9 准备的 class 版本，可以运用 Java 9 所提供的特性和库。同时，在早期的 Java 诸版本中使用这个 JAR 也是能运行的，因为较老版本的 Java 只会看到顶层的这个 Helper 类。

(五)——网络协议篇

（一）网络基础知识

1）Http 和 Https 的区别?

答：Http 协议运行在 TCP 之上，明文传输，客户端与服务器端都无法验证对方的身份；Https 是身披 SSL(Secure Socket Layer)外壳的 Http，运行于 SSL 上，

SSL 运行于 TCP 之上，是添加了加密和认证机制的 HTTP。二者之间存在如下不同：

- 端口不同：Http 与 Http 使用不同的连接方式，用的端口也不一样，前者是 80，后者是 443；
- 资源消耗：和 HTTP 通信相比，Https 通信会由于加减密处理消耗更多的 CPU 和内存资源；
- 开销：Https 通信需要证书，而证书一般需要向认证机构购买；

Https 的加密机制是一种共享密钥加密和公开密钥加密并用的混合加密机制。

2）对称加密与非对称加密

答：

对称密钥加密是指加密和解密使用同一个密钥的方式，这种方式存在的最大问题就是密钥发送问题，即如何安全地将密钥发给对方；而非对称加密是指使用一对非对称密钥，即公钥和私钥，公钥可以随意发布，但私钥只有自己知道。发送密文的一方使用对方的公钥进行加密处理，对方接收到加密信息后，使用自己的私钥进行解密。

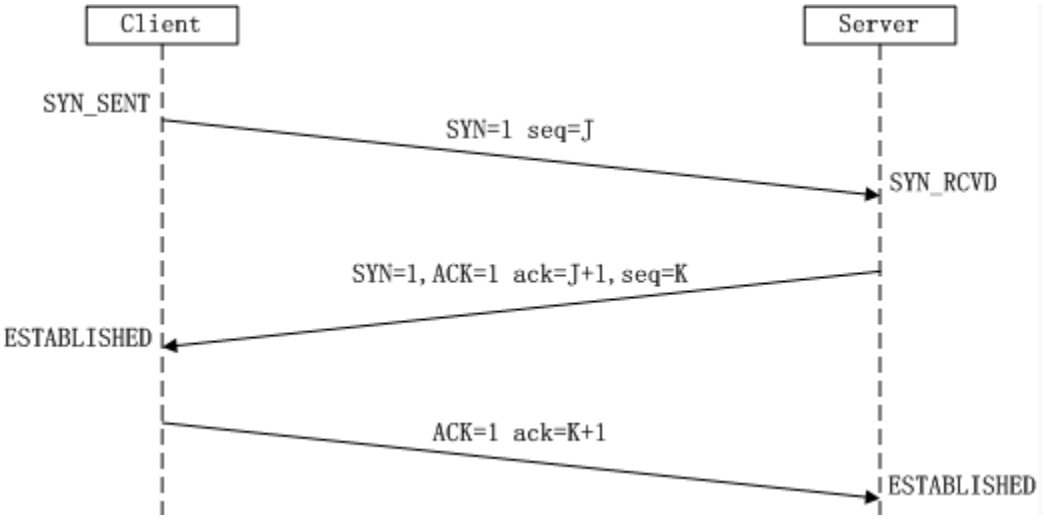
由于非对称加密的方式不需要发送用来解密的私钥，所以可以保证安全性；但是和对称加密比起来，它非常的慢，所以我们还是要用对称加密来传送消息，但对称加密所使用的密钥我们可以通过非对称加密的方式发送出去。

3）三次握手与四次挥手

答：

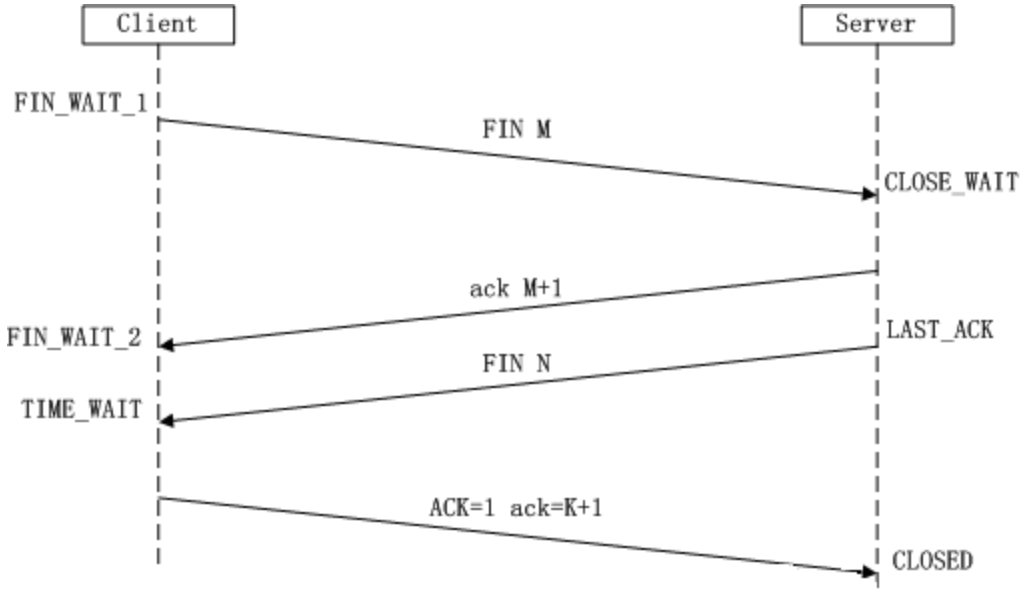
(1). 三次握手（我要和你建立链接，你真的要和我建立链接么，我真的要和你建立链接，成功）

- 第一次握手：Client 将标志位 SYN 置为 1，随机产生一个值 seq=J，并将该数据包发送给 Server，Client 进入 SYN_SENT 状态，等待 Server 确认。
- 第二次握手：Server 收到数据包后由标志位 SYN=1 知道 Client 请求建立连接，Server 将标志位 SYN 和 ACK 都置为 1，ack=J+1，随机产生一个值 seq=K，并将该数据包发送给 Client 以确认连接请求，Server 进入 SYN_RCVD 状态。
- 第三次握手：Client 收到确认后，检查 ack 是否为 J+1，ACK 是否为 1，如果正确则将标志位 ACK 置为 1，ack=K+1，并将该数据包发送给 Server，Server 检查 ack 是否为 K+1，ACK 是否为 1，如果正确则连接建立成功，Client 和 Server 进入 ESTABLISHED 状态，完成三次握手，随后 Client 与 Server 之间可以开始传输数据了。

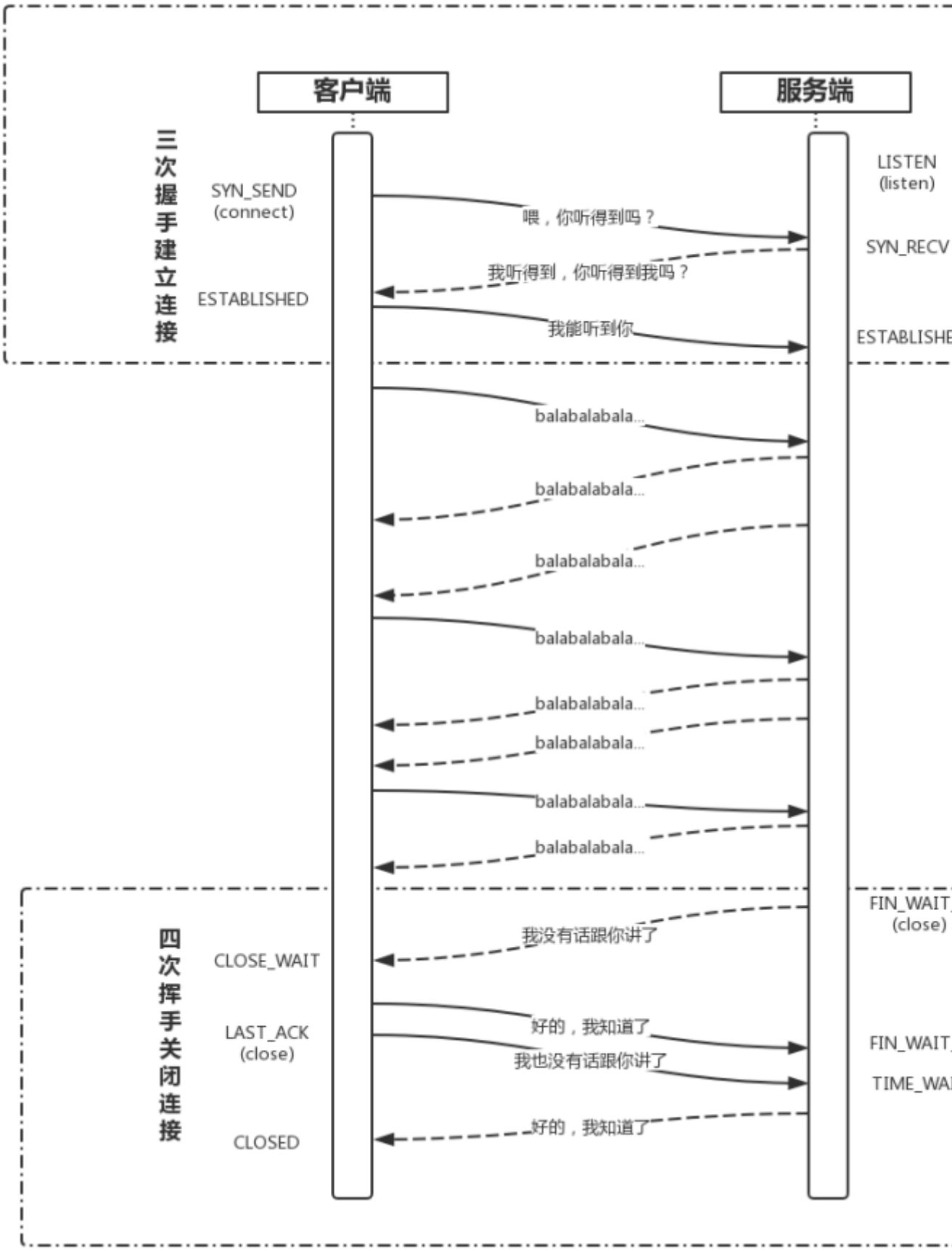


(2). 四次挥手（我要和你断开链接；好的，断吧。我也要和你断开链接；好的，断吧）：

- 第一次挥手：Client 发送一个 FIN，**用来关闭 Client 到 Server 的数据传送**，Client 进入 FIN_WAIT_1 状态。
- 第二次挥手：Server 收到 FIN 后，发送一个 ACK 给 Client，确认序号为收到序号+1（与 SYN 相同，一个 FIN 占用一个序号），Server 进入 CLOSE_WAIT 状态。此时 TCP 链接处于半关闭状态，即客户端已经没有要发送的数据了，但服务端若发送数据，则客户端仍要接收。
- 第三次挥手：Server 发送一个 FIN，**用来关闭 Server 到 Client 的数据传送**，Server 进入 LAST_ACK 状态。
- 第四次挥手：Client 收到 FIN 后，Client 进入 TIME_WAIT 状态，接着发送一个 ACK 给 Server，确认序号为收到序号+1，Server 进入 CLOSED 状态，完成四次挥手。


Client
Server
FIN_WAIT_1
FIN M
CLOSE_WAIT
ack M+1
FIN_WAIT_2
LAST_ACK
FIN N
TIME_WAIT
ACK=1 ack=K+1
CLOSED

(3). 通俗一点的理解就是：


客户端
服务端
三次握手建立连接
SYN_SEND
(connect)
喂，你听得到吗？
LISTEN
(listen)
SYN_RECV
我听得到，你听得到我吗？
ESTABLISHED
我能听到你
ESTABLISHE
balabalabala...
balabalabala...
balabalabala...
balabalabala...
balabalabala...
balabalabala...
balabalabala...
balabalabala...
FIN_WAIT
(close)
四次挥手关闭连接
我没有话跟你讲了
CLOSE_WAIT
好的，我知道了
LAST_ACK
(close)
FIN_WAIT
我也没有话跟你讲了
TIME_WA
好的，我知道了
CLOSED

4）为什么 TCP 链接需要三次握手，两次不可以么?

答：“三次握手” 的目的是为了防止**已失效的链接请求报文突然又传送到了服务端**，因而产生错误。

- 正常的情况：A 发出连接请求，但因连接请求报文丢失而未收到确认，于是 A 再重传一次连接请求。后来收到了确认，建立了连接。数据传输完毕后，就释放了连接。A 共发送了两个连接请求报文段，其中第一个丢失，第二个到达了 B。没有 “已失效的连接请求报文段”。

- 现假定出现了一种异常情况：即 A 发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达 B。本来这是一个早已失效的报文段。但 B 收到此失效的连接请求报文段后，就误认为是 A 再次发出的一个新的连接请求。于是就向 A 发出确认报文段，同意建立连接。

假设不采用“三次握手”，那么只要 B 发出确认，新的连接就建立了。由于现在 A 并没有发出建立连接的请求，因此不会理睬 B 的确认，也不会向 B 发送数据。但 B 却以为新的运输连接已经建立，并一直等待 A 发来数据。这样，B 的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。

5）为什么要四次挥手?

答：TCP 协议是一种面向连接的、可靠的、基于字节流的运输层通信协议。TCP 是全双工模式，这就意味着，当 A 向 B 发出 FIN 报文段时，只是表示 A 已经

没有数据要发送了，而此时 A 还是能够接受到来自 B 发出的数据；B 向 A 发出 ACK 报文段也只是告诉 A ，它自己知道 A 没有数据要发了，但 B 还是能够向 A 发送数据。

所以想要愉快的结束这次对话就需要四次挥手。

6）TCP 协议如何来保证传输的可靠性

答：TCP 提供一种面向连接的、可靠的字节流服务。其中，面向连接意味着两个使用 TCP 的应用（通常是一个客户和一个服务器）在彼此交换数据之前必须先建立一个 TCP 连接。在一个 TCP 连接中，仅有两方进行彼此通信；而字节流服务意味着两个应用程序通过 TCP 链接交换 8 bit 字节构成的字节流，TCP 不在字节流中插入记录标识符。

对于可靠性，TCP 通过以下方式进行保证：

- **数据包校验**：目的是检测数据在传输过程中的任何变化，若校验出包有错，则丢弃报文段并且不给出响应，这时 TCP 发送数据端超时后会重发数据；
- **对失序数据包重排序**：既然 TCP 报文段作为 IP 数据报来传输，而 IP 数据报的到达可能会失序，因此 TCP 报文段的到达也可能会失序。TCP 将对失序数据进行重新排序，然后才交给应用层；
- **丢弃重复数据**：对于重复数据，能够丢弃重复数据；
- **应答机制**：当 TCP 收到发自 TCP 连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒；

•**超时重发**：当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段；

•**流量控制**：TCP 连接的每一方都有固定大小的缓冲空间。TCP 的接收端只允许另一端发送接收端缓冲区所能接纳的数据，这可以防止较快主机致使较慢主机的缓冲区溢出，这就是流量控制。TCP 使用的流量控制协议是可变大小的滑动窗口协议。

7）客户端不断进行请求链接会怎样？DDos(Distributed Denial of Service)攻击?

答：服务器端会为每个请求创建一个链接，并向其发送确认报文，然后等待客户端进行确认

(1). DDos 攻击：

- 客户端向服务端发送请求链接数据包
- 服务端向客户端发送确认数据包
- 客户端不向服务端发送确认数据包，服务器一直等待来自客户端的确认

(2). DDos 预防： （没有彻底根治的办法，除非不使用 TCP）

- 限制同时打开 SYN 半链接的数目
- 缩短 SYN 半链接的 Time out 时间
- 关闭不必要的服务

8）GET 与 POST 的区别?

答：GET 与 POST 是我们常用的两种 HTTP Method，二者之间的区别主要包括如下五个方面：

(1). 从功能上讲，GET 一般用来从服务器上获取资源，POST 一般用来更新服务器上的资源；

(2). 从 REST 服务角度上说，GET 是幂等的，即读取同一个资源，总是得到相同的数据，而 POST 不是幂等的，因为每次请求对资源的改变并不是相同的；进一步地，GET 不会改变服务器上的资源，而 POST 会对服务器资源进行改变；

(3). 从请求参数形式上看，GET 请求的数据会附在 URL 之后，即将请求数据放置在 HTTP 报文的 请求头 中，以?分割 URL 和传输数据，参数之间以&相连。特别地，如果数据是英文字母/数字，原样发送；否则，会将其编码为 application/x-www-form-urlencoded MIME 字符串(如果是空格，转换为+，如果是中文/其他字符，则直接把字符串用 BASE64 加密，得出如：%E4%BD%A0%E5%A5%BD，其中%XX 中的 XX 为该符号以 16 进制表示的 ASCII)；而 POST 请求会把提交的数据则放置在是 HTTP 请求报文的 请求体 中。

(4). 就安全性而言，POST 的安全性要比 GET 的安全性高，因为 GET 请求提交的数据将明文出现在 URL 上，而且 POST 请求参数则被包装到请求体中，相对更安全。

(5). 从请求的大小看，GET 请求的长度受限于浏览器或服务器对 URL 长度的限制，允许发送的数据量比较小，而 POST 请求则是没有大小限制的。

为什么在 GET 请求中会对 URL 进行编码?

我们知道，在 GET 请求中会对 URL 中非西文字符进行编码，这样做的目的就是为了 **避免歧义**。看下面的例子，

针对 “name1=value1&name2=value2” 的例子，我们来谈一下数据从客户端到服务端的解析过程。首先，上述字符串在计算机中用 ASCII 吗表示为：

```
6E616D6531 3D 76616C756531 26 6E616D6532 3D 76616C756532

6E616D6531：name1

3D：=

76616C756531：value1

26：&

6E616D6532：name2

3D：=

76616C756532：value2
```

服务端在接收到该数据后就可以遍历该字节流，一个字节一个字节的吃，当吃到 3D 这字节后，服务端就知道前面吃得字节表示一个 key，再往后吃，如果遇到 26，说明从刚才吃的 3D 到 26 子节之间的是上一个 key 的 value，以此类推就可以解析出客户端传过来的参数。

现在考虑这样一个问题，如果我们的参数值中就包含=或&这种特殊字符的时候该怎么办？比如，“name1=value1”，其中 value1 的值是“va&lu=e1”字符串，那么实际在传输过程中就会变成这样“name1=va&lu=e1”。这样，我们的本意是只有一个键值对，但是服务端却会解析成两个键值对，这样就产生了歧义。

那么，如何解决上述问题带来的歧义呢？解决的办法就是对参数进行 URL 编码：例如，我们对上述会产生歧义的字符进行 URL 编码后结果：

“name1=va%26lu%3D”，这样服务端会把紧跟在“%”后的字节当成普通的字节，就是不会把它当成各个参数或键值对的分隔符。

9）TCP 与 UDP 的区别

答：TCP (Transmission Control Protocol)和 UDP(User Datagram Protocol)协议属于传输层协议，它们之间的区别包括：

- TCP 是面向连接的，UDP 是无连接的；
- TCP 是可靠的，UDP 是不可靠的；
- TCP 只支持点对点通信，UDP 支持一对一、一对多、多对一、多对多的通信模式；
- TCP 是面向字节流的，UDP 是面向报文的；
- TCP 有拥塞控制机制;UDP 没有拥塞控制，适合媒体通信；
- TCP 首部开销(20 个字节)比 UDP 的首部开销(8 个字节)要大；

10）TCP 和 UDP 分别对应的常见应用层协议

答：

(1). TCP 对应的应用层协议：

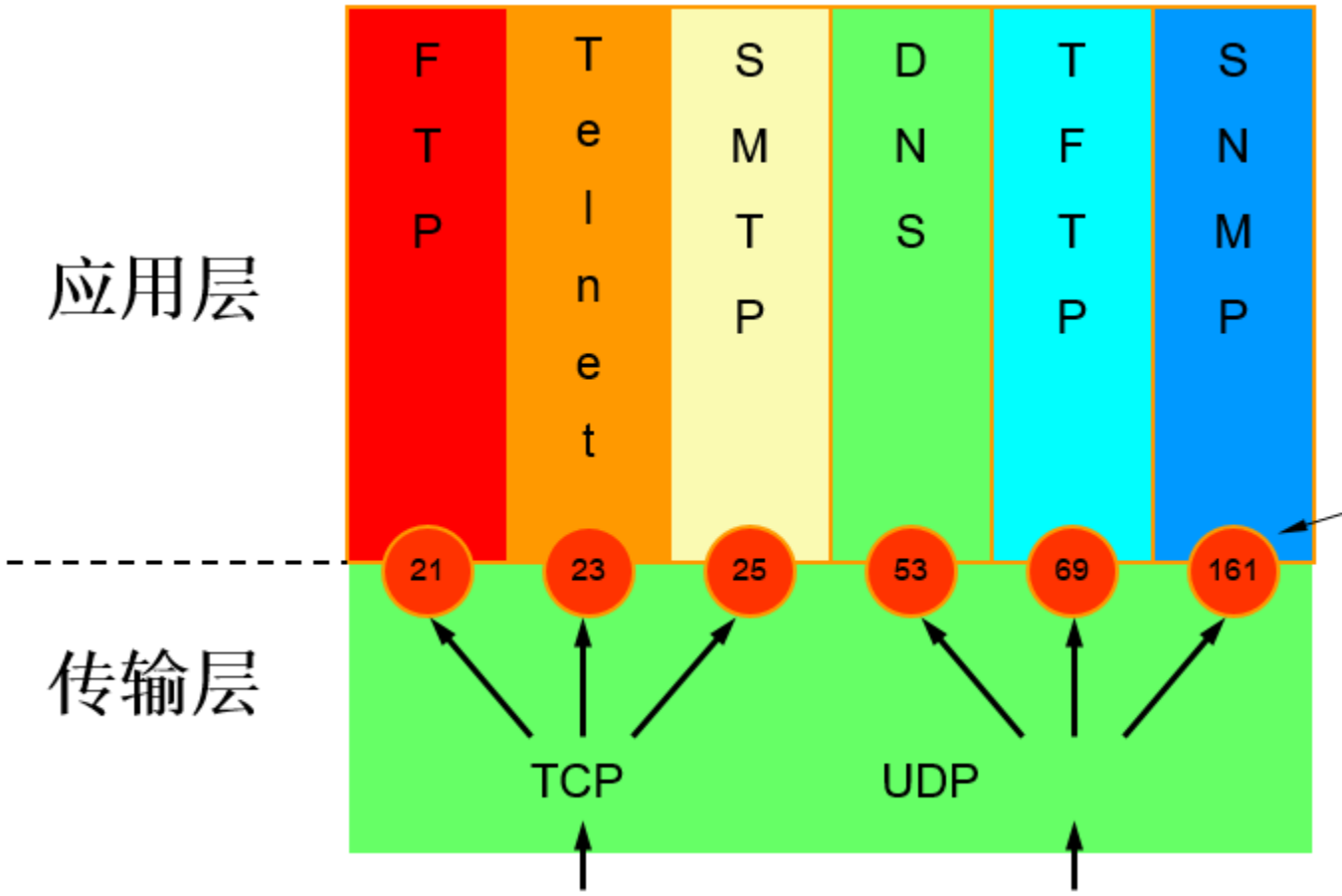
- **FTP**：定义了文件传输协议，使用 21 端口。常说某某计算机开了 FTP 服务便是启动了文件传输服务。下载文件，上传主页，都要用到 FTP 服务。

- **Telnet**：它是一种用于远程登陆的端口，用户可以以自己的身份远程连接到计算机上，通过这种端口可以提供一种基于 DOS 模式下的通信服务。如以前的 BBS 是-纯字符界面的，支持 BBS 的服务器将 23 端口打开，对外提供服务。
- **SMTP**：定义了简单邮件传送协议，现在很多邮件服务器都用的是这个协议，用于发送邮件。如常见的免费邮件服务中用的就是这个邮件服务端口，所以在电子邮件设置-中常看到有这么 SMTP 端口设置这个栏，服务器开放的是 25 号端口。
- **POP3**：它是和 SMTP 对应，POP3 用于接收邮件。通常情况下，POP3 协议所用的是 110 端口。也是说，只要你有相应的使用 POP3 协议的程序（例如 Fo-xmail 或 Outlook），就可以不以 Web 方式登陆进邮箱界面，直接用邮件程序就可以收到邮件（如是 163 邮箱就没有必要先进入网易网站，再进入自己的邮-箱来收信）。
- **HTTP**：从 Web 服务器传输超文本到本地浏览器的传送协议。

(2). UDP 对应的应用层协议：

- **DNS**：用于域名解析服务，将域名地址转换为 IP 地址。DNS 用的是 53 号端口。
- **SNMP**：简单网络管理协议，使用 161 号端口，是用来管理网络设备的。由于网络设备很多，无连接的服务就体现出其优势。
- **TFTP(Trival File Transfer Protocal)**：简单文件传输协议，该协议在熟知端口 69 上使用 UDP 服务

(3). 图示：



11）TCP 的拥塞避免机制

答：

拥塞：对资源的需求超过了可用的资源。若网络中许多资源同时供应不足，网络的性能就要明显变坏，整个网络的吞吐量随之负荷的增大而下降。

拥塞控制：防止过多的数据注入到网络中，使得网络中的路由器或链路不致过载。

拥塞控制的方法：

(1). 慢启动 + 拥塞避免：

慢启动：不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小;

拥塞避免：拥塞避免算法让拥塞窗口缓慢增长，即每经过一个往返时间 RTT 就把发送方的拥塞窗口 cwnd 加 1，而不是加倍，这样拥塞窗口按线性规律缓慢增长。

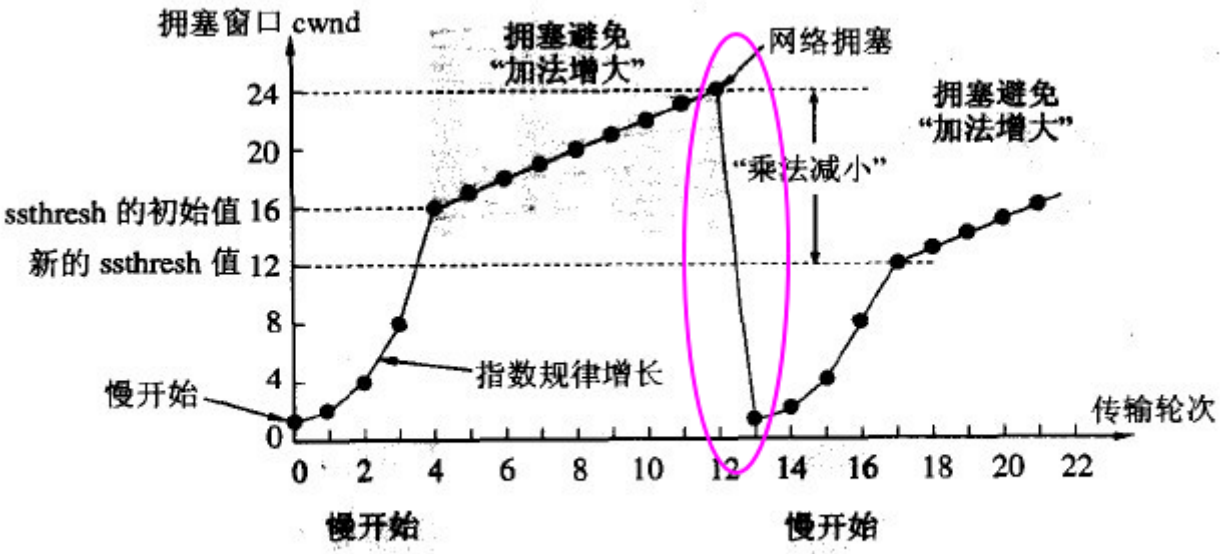


图 5-25　慢开始和拥塞避免算法的实现举例 net/sicofield

(2). 快重传 + 快恢复：

快重传：快重传要求接收方在收到一个 **失序的报文段** 后就立即发出 **重复确认**（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时捎带确认。快重传算法规定，发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器时间到期。

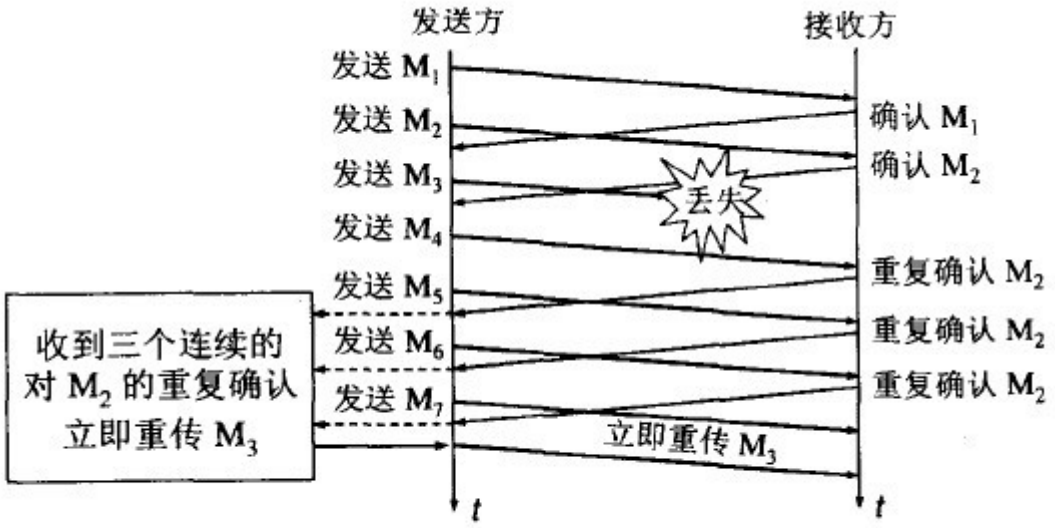


图 5-26　快重传的示意图

快恢复：快重传配合使用的还有快恢复算法，当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把 ssthresh 门限减半，但是接下去并不执行慢开始算法：因为如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。所以此时不执行慢开始算法，而是将 cwnd 设置为 ssthresh 的大小，然后执行拥塞避免算法。

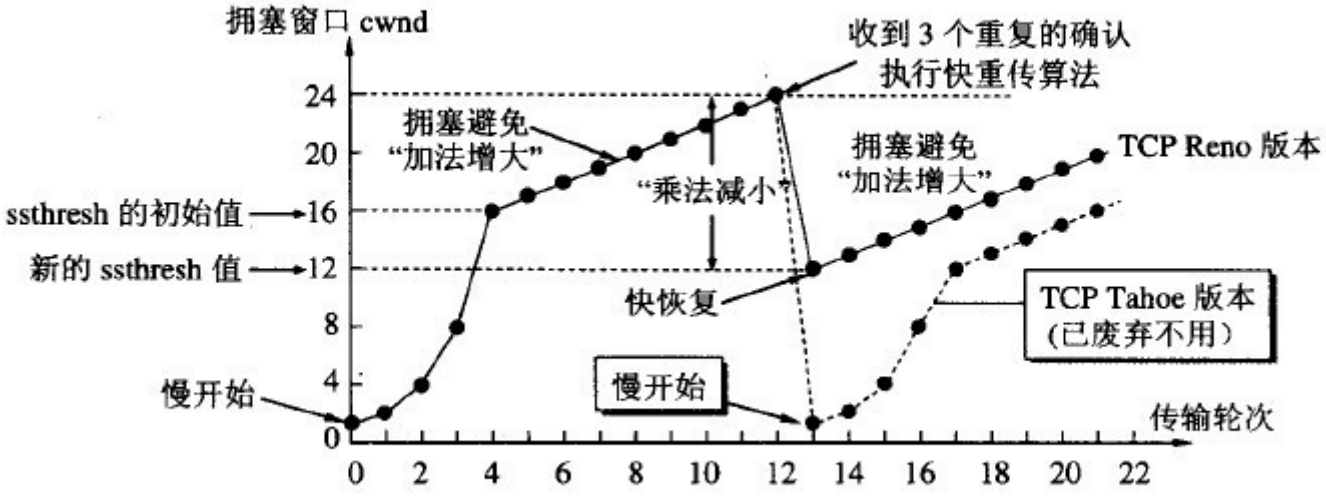


图 5-27　从连续收到三个重复的确认转入拥塞避免

12）浏览器中输入：“www.xxx.com” 之后都发生了什么？请详细阐述。

解析：经典的网络协议问题。

答：

1. 由域名→IP 地址 寻找 IP 地址的过程依次经过了浏览器缓存、系统缓存、hosts 文件、路由器缓存、 递归搜索根域名服务器。
2. 建立 TCP/IP 连接（三次握手具体过程）
3. 由浏览器发送一个 HTTP 请求
4. 经过路由器的转发，通过服务器的防火墙，该 HTTP 请求到达了服务器
5. 服务器处理该 HTTP 请求，返回一个 HTML 文件
6. 浏览器解析该 HTML 文件，并且显示在浏览器端
7. 这里需要注意：

- HTTP 协议是一种基于 TCP/IP 的应用层协议，进行 HTTP 数据请求必须先建立 TCP/IP 连接
- 可以这样理解：HTTP 是轿车，提供了封装或者显示数据的具体形式；Socket 是发动机，提供了网络通信的能力。
- 两个计算机之间的交流无非是两个端口之间的数据通信,具体的数据会以什么样的形式展现是以不同的应用层协议来定义的。

13）什么是 HTTP 协议无状态协议？怎么解决 Http 协议无状态协议?

答：HTTP 是一个无状态的协议，也就是没有记忆力，这意味着每一次的请求都是独立的，缺少状态意味着如果后续处理需要前面的信息，则它必须要重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就很快。

HTTP 的这种特性有优点也有缺点：

- **优点**：解放了服务器，每一次的请求“点到为止”，不会造成不必要的连接占用
- **缺点**：每次请求会传输大量重复的内容信息，并且，在请求之间无法实现数据的共享

解决方案：

1. 使用参数传递机制：

   将参数拼接在请求的 URL 后面，实现数据的传递（GET 方式），例如：`/param/list?username=wmyskxz`

   问题：可以解决数据共享的问题，但是这种方式一不安全，二数据允许传输量只有 1kb
2. 使用 Cookie 技术
3. 使用 Session 技术

14）Session、Cookie 与 Application

答：Cookie 和 Session 都是客户端与服务器之间保持状态的解决方案，具体来说，cookie 机制采用的是在客户端保持状态的方案，而 session 机制采用的是在服务器端保持状态的方案。

(1). Cookie 及其相关 API ：

Cookie 实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用 response 向客户端浏览器颁发一个 Cookie，而客户端浏览器会把 Cookie 保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该 Cookie 一同提交给服务器，服务器检查该 Cookie，以此来辨认用户状态。服务器还可以根据需要修改 Cookie 的内容。

void	addCookie(Cookie cookie) HttpServletResponse Adds the specified cookie to the response.
void	addDateHeader(java.lang.String name, long date) Adds a response header with the given name and date-value.
void	addHeader(java.lang.String name, java.lang.String value) Adds a response header with the given name and value.
void	addIntHeader(java.lang.String name, int value) Adds a response header with the given name and integer value.
boolean	containsHeader(java.lang.String name) Returns a boolean indicating whether the named response header
.String	encodeRedirectUrl(java.lang.String url) **Deprecated.** *As of version 2.1, use encodeRedirectURL(String url)*
.String	encodeRedirectURL(java.lang.String url) Encodes the specified URL for use in the sendRedirect method or, if
.String	encodeUrl(java.lang.String url) **Deprecated.** *As of version 2.1, use encodeURL(String url) instead*
.String	encodeURL(java.lang.String url) HttpServletResponse Encodes the specified URL by including the session ID in it, or, if e

Cookie[]	getCookies() HttpServletRequest Returns an array containing all of the Cookie objects the client sen
long	getDateHeader(java.lang.String name) Returns the value of the specified request header as a long value

(2). Session 及其相关 API：

同样地，会话状态也可以保存在服务器端。客户端请求服务器，如果服务器记录该用户状态，就获取 Session 来保存状态，这时，如果服务器已经为此客户端创建过 session，服务器就按照 sessionid 把这个 session 检索出来使用；如果客户端请求不包含 sessionid，则为此客户端创建一个 session 并且生成一个与此 session 相关联的 sessionid，并将这个 sessionid 在本次响应中返回给客户端保存。保存这个 sessionid 的方式可以采用 **cookie 机制**，这样在交互过程中浏览器可以自动的按照规则把这个标识发挥给服务器；若浏览器禁用 Cookie 的话，可以通过 **URL 重写机制** 将 sessionid 传回服务器。

HttpSession	getSession() HttoServletRequest Returns the current session associated with this request, or if th
HttpSession	getSession(boolean create) Returns the current HttpSession associated with this request or, if

(3). Session 与 Cookie 的对比：

- **实现机制**：Session 的实现常常依赖于 Cookie 机制，通过 Cookie 机制回传 SessionID；
- **大小限制**：Cookie 有大小限制并且浏览器对每个站点也有 cookie 的个数限制，Session 没有大小限制，理论上只与服务器的内存大小有关；

- **安全性**：Cookie 存在安全隐患，通过拦截或本地文件找得到 cookie 后可以进行攻击，而 Session 由于保存在服务器端，相对更加安全；
- **服务器资源消耗**：Session 是保存在服务器端上会存在一段时间才会消失，如果 session 过多会增加服务器的压力。

(4). Application：

Application（ServletContext）：与一个 Web 应用程序相对应，为应用程序提供了一个全局的状态，所有客户都可以使用该状态。

15）滑动窗口机制

答：由发送方和接收方在三次握手阶段，互相将自己的最大可接收的数据量告诉对方。也就是自己的数据接收缓冲池的大小。这样对方可以根据已发送的数据量来计算是否可以接着发送。在处理过程中，当接收缓冲池的大小发生变化时，要给对方发送更新窗口大小的通知。这就实现了流量的控制。

16）常用的 HTTP 方法有哪些?

答：

- GET： 用于请求访问已经被 URI（统一资源标识符）识别的资源，可以通过 URL 传参给服务器
- POST：用于传输信息给服务器，主要功能与 GET 方法类似，但一般推荐使用 POST 方式。

- PUT： 传输文件，报文主体中包含文件内容，保存到对应 URI 位置。
- HEAD： 获得报文首部，与 GET 方法类似，只是不返回报文主体，一般用于验证 URI 是否有效。
- DELETE：删除文件，与 PUT 方法相反，删除对应 URI 位置的文件。
- OPTIONS：查询相应 URI 支持的 HTTP 方法。

17）常见 HTTP 状态码

答：

1. 1xx（临时响应）
2. 2xx（成功）
3. 3xx（重定向）：表示要完成请求需要进一步操作
4. 4xx（错误）：表示请求可能出错，妨碍了服务器的处理
5. 5xx（服务器错误）：表示服务器在尝试处理请求时发生内部错误
6. 常见状态码：

- 200（成功）
- 304（未修改）：自从上次请求后，请求的网页未修改过。服务器返回此响应时，不会返回网页内容
- 401（未授权）：请求要求身份验证
- 403（禁止）：服务器拒绝请求
- 404（未找到）：服务器找不到请求的网页

18）SQL 注入

答：SQL 注入就是通过把 SQL 命令插入到 Web 表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的 SQL 命令。

(1).SQL 注入攻击的总体思路：

1. 寻找到 SQL 注入的位置
2. 判断服务器类型和后台数据库类型
3. 针对不通的服务器和数据库特点进行 SQL 注入攻击

(2). SQL 注入攻击实例：

比如，在一个登录界面，要求输入用户名和密码，可以这样输入实现免帐号登录：

```
用户名：  ‘or 1 = 1 --
密 码：
```

用户一旦点击登录，如若没有做特殊处理，那么这个非法用户就很得意的登陆进去了。这是为什么呢?下面我们分析一下：从理论上说，后台认证程序中会有如下的 SQL 语句：

```
String sql = “select * from user_table where username=’
“+userName+” ’ and password=’ “+password+” ‘”;
```

因此，当输入了上面的用户名和密码，上面的 SQL 语句变成：

```
SELECT * FROM user_table WHERE username=’’ or 1 = 1 – and
password=’’
```

分析上述 SQL 语句我们知道，username= ‘ or 1=1 这个语句一定会成功；然后后面加两个-，这意味着注释，它将后面的语句注释，让他们不起作用。这样，上述语句永远都能正确执行，用户轻易骗过系统，获取合法身份。

(3). 应对方法：

1.参数绑定：

使用预编译手段，绑定参数是最好的防 SQL 注入的方法。目前许多的 ORM 框架及 JDBC 等都实现了 SQL 预编译和参数绑定功能，攻击者的恶意 SQL 会被当做 SQL 的参数而不是 SQL 命令被执行。在 mybatis 的 mapper 文件中，对于传递的参数我们一般是使用#和$来获取参数值。当使用#时，变量是占位符，就是一般我们使用 javajdbc 的 PrepareStatement 时的占位符，所有可以防止 sql 注入；当使用$时，变量就是直接追加在 sql 中，一般会有 sql 注入问题。

2.使用正则表达式过滤传入的参数

19）XSS 攻击

答：XSS 是一种经常出现在 web 应用中的计算机安全漏洞，与 SQL 注入一起成为 web 中最主流的攻击方式。XSS 是指恶意攻击者利用网站没有对用户提交数据进行转义处理或者过滤不足的缺点，进而添加一些脚本代码嵌入到 web 页面中去，使别的用户访问都会执行相应的嵌入代码，从而盗取用户资料、利用用户身份进行某种动作或者对访问者进行病毒侵害的一种攻击方式。

(1). XSS 攻击的危害：

- 盗取各类用户帐号，如机器登录帐号、用户网银帐号、各类管理员帐号
- 控制企业数据，包括读取、篡改、添加、删除企业敏感数据的能力
- 盗窃企业重要的具有商业价值的资料

- 非法转账
- 强制发送电子邮件
- 网站挂马
- 控制受害者机器向其它网站发起攻击

(2). 原因解析:

- **主要原因**：过于信任客户端提交的数据!
- **解决办法**：不信任任何客户端提交的数据，只要是客户端提交的数据就应该先进行相应的过滤处理然后方可进行下一步的操作。
- **进一步分析细节**：客户端提交的数据本来就是应用所需要的，但是恶意攻击者利用网站对客户端提交数据的信任，在数据中插入一些符号以及 javascript 代码，那么这些数据将会成为应用代码中的一部分了，那么攻击者就可以肆无忌惮地展开攻击啦，因此我们绝不可以信任任何客户端提交的数据！！！

(3). XSS 攻击分类:

-
  - 反射性 XSS 攻击（非持久性 XSS 攻击）：

漏洞产生的原因是攻击者注入的数据反映在响应中。一个典型的非持久性 XSS 攻击包含一个带 XSS 攻击向量的链接(即每次攻击需要用户的点击)，例如，正常发送消息:

http://www.test.com/message.php?send=Hello,World!

接收者将会接收信息并显示 Hello,World；但是，非正常发送消息:

http://www.test.com/message.php?send=<script>alert(‘foolish!’)</script>!

接收者接收消息显示的时候将会弹出警告窗口!

-
    - 持久性 XSS 攻击 (留言板场景):

XSS 攻击向量(一般指 XSS 攻击代码)存储在网站数据库，当一个页面被用户打开的时候执行。也就是说，每当用户使用浏览器打开指定页面时，脚本便执行。与非持久性 XSS 攻击相比，持久性 XSS 攻击危害性更大。从名字就可以了解到，持久性 XSS 攻击就是将攻击代码存入数据库中，然后客户端打开时就执行这些攻击代码。

例如，留言板表单中的表单域:

<input type=“text” name=“content” value=“这里是用户填写的数据”>

正常操作流程是：用户是提交相应留言信息 —— 将数据存储到数据库 —— 其他用户访问留言板，应用去数据并显示；而非正常操作流程是攻击者在 value 填写:

<script>alert(‘foolish!’);</script> <!--或者 html 其他标签（破坏样式）、一段攻击型代码-->

并将数据提交、存储到数据库中；当其他用户取出数据显示的时候，将会执行这些攻击性代码。

(4). 修复漏洞方针:

漏洞产生的根本原因是 **太相信用户提交的数据，对用户所提交的数据过滤不足所导致的**，因此解决方案也应该从这个方面入手，具体方案包括:

- 将重要的 cookie 标记为 http only，这样的话 Javascript 中的 document.cookie 语句就不能获取到 cookie 了（如果在 cookie 中设置了 HttpOnly 属性，那么通过 js 脚本将无法读取到 cookie 信息，这样能有效的防止 XSS 攻击）；
- 表单数据规定值的类型，例如：年龄应为只能为 int、name 只能为字母数字组合。。。。
- 对数据进行 Html Encode 处理
- 过滤或移除特殊的 Html 标签，例如: