Java 进阶讲义

张力 著

前言

这是一本 Java 方向的进阶教材，适用于计算机、软件方向本科即将毕业，将要寻找工作的同学参阅，本资料的内容分为 7 部分：

1、Java 语言拾遗，专门挑取几个面试高频问题作出讲解，这部分较零碎，仅覆盖高频知识点。

2、流与 I/O，此部分专门讲解 I/O 知识。

3、NIO,专门讲解冷门的 NIO 知识，但并未提及 NIO 在网络编程中的应用，关于 select,socket 等知识点，还请在网上查阅资料。

4、多线程与并发，此章以 Java 语言为具体语言，描述了竞争条件，锁，同步等操作系统概念，并且每一个知识点都有代码示例，建议同学动手练习。

5、一个 HTTP 请求的前世今生，此章主要是原理性的的叙述，说明了一个 HTTP 请求完成的基本步骤，讲述了今天的互联网架构是如何演化至今的。阅读此章可以帮助你融会贯通其他知识，加深理解。

6、JSP 与 Servlet 说明了 Servlet 的起因和 JSP 的本质，阅读该章同样能加深你对 Java Web 方面的理解。

7、Spring，Spring 作为 Java 世界最出色的框架，多年以来备受推崇，那是什么原因和什么思想，导致了它的流行呢？它到底解决了什么问题？此章会通过原理性的代码写出一个小型的 Spring，帮助你根本性的理解 Spring，如果阅读本章有困难，请先熟悉一下 Java 的反射机制。

8、其他框架简介，简要描述了 Hibernate 和 MyBatis 的功能和所解决的问题，描述了基本的 API 和用法，时间充裕的同学可以简要了解一下。

学习计算机知识最好的方法就是边看边练，经常动手练习才能以最快速度进步。

另外、本材料作为复习性质的讲义，缺乏专业书籍的系统性，适合有一定基础的同学做定向加强的复习，基础薄弱的同学，建议完整的学习《Java 编程思想》并作大量练习，而不是直接学习本材料。

张 力

2015 年 8 月 4 日 星期三

目录

一、Java 语言拾遗

1、Vector 与 ArrayList

Vector @since JDK1.0，从 Java 2 平台 v1.2 开始，此类改进为可以实现 List 接口，使它成为 Java Collections Framework 的成员。与新 collection 实现不同，Vector 是同步的。

不同之处一：版本，Vector 始于 JDK1.0 而 ArrayList 始于 JDK1.2

不同之处二：线程安全

Vector:Vector 是线程安全的，但是由于使用了 synchronize 导致了效率不如 ArrayList。

```
Vector 源码：
public synchronized boolean add(E e){
        modCount++;
        ensureCapacityHelper(elementCount+1);
        elementData[elementCount++] = e;
        return true;
}
public synchronized boolean addElement(E obj){
        modCount++;
        ensureCapacityHelper(elementCount+1);
        elementData[elementCount++] = obj;
}
```

ArrayList:ArrayList 本身是线程不安全的，要想在并发环境中使用 ArrayList，请使用 Collections.synchronizedList(List list)。在单线程中使用 Vector 和 ArrayList 的时候，由于 ArrayList 没有加锁开

```
ArrayList 源码
public boolean add(E e){
        ensureCapacityInternal(size+1);// Increments modCount !!
        elementData[size++] = e;
        return true;
}
```

销，所以性能上面优于 Vector。

不同之处三：容量增长方式

Vector:如果设置了增量，增加增量，如果没有设置增量，容量翻倍

```
Vector 源码
private void grow(int minCapacity){
        //overflow-conscious code
        int oldCapacity = elementData.lenght;
        int newCapacity = oldCapacity + ((capacityIncrement > 0 ) ? capacityIncrement : oldCapacity);
        ...
        ...
}
```

ArrayList:容量翻为约 1.5 倍，jdk6(x3/2+1)，jdk7(c+c>>1)

```
ArrayList 源码
private void grow(int minCapacity){
        //overflow-conscious code
        int oldCapacity = elementData.length;
        int newCapacity = oldCapacity + (oldCapacity >> 1);
        …
        …
}
```

2、ArrayList 与 LinkedList

继承关系对比：

AbstractCollection <- AbstractList <- ArrayList

AbstractCollection <- AbstractList <- AbstractSequentialList <- LinkedList

```
ArrayList 源码
public class ArrayList<E> extends AbstractList<E> implements List <E>,RandomAccess,Cloneable,java.io.Serial-
izable{
        …
        …
}
```

```
LinkedList 源码
public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>,Deque<E>,Clonea-
ble,java.io.Serializable{
        …
        …
}
```

ArrayList 由数组实现，实现了 RandomAccess 接口（常规 for 快于迭代器遍历），即可以随机存储。LinkedList 由链表实现，并且是一个双向链表。因此 ArrayList 随机存储时间复杂度 O(1)，但是插入操作时间复杂度 O(N)，LinkedList 随机存储时间复杂度 O(N)，插入操作时间复杂度 O(1)。

因此，对于插入删除少的情况下或者预期数据的数量变化不太大时，使用 ArrayList 可以获得更好的效率，对于频繁的插入删除操作或者是频繁的扩容操作，LinkedList 是一个更好的选择。

如果想在并发环境下使用 ArrayList 或者是 LinkedList，可以使用 Collections. synchronizedList(List<T> list) 方法来包装一下 List，但是其原理还是使用 synchronized 对 List 的每一个实现方法上锁，效率低。如果在并发环境下，想提升 List 的使用效率，还是建议手动上锁，人为控制好对 List 的修改。

除此之外 LinkedList 不仅是一个 List，同时他还是一个双向队列的实现（implements Deque）。

3、Equals()与 Hashcode()

对基于 hash 的容器（HashMap，HashSet 便是其中典范），存在一个约定：如果有 a.equals(b)，那么必须使得 a.hashCode() == b.hashCode()，否则，hash 容器将不能按照定义存取对象，并且基于 hash 的快速查找

```
import java.util.HashMap;
import java.util.Map;

public class Main {
	public static void main(String[] args) {
		Map<Coordinate,Integer> coordinates = new HashMap<Coordinate, Integer>();
		Coordinate c = new Coordinate(2, 2);
		coordinates.put(c, 1);
		System.out.println(c.equals(new Coordinate(2, 2)));
		System.out.println(coordinates.get(new Coordinate(2, 2)));
	}
}
class Coordinate{
	private int x;
	private int y;

	public Coordinate(int x,int y){this.x=x;this.y=y;}
	public int getX() { return x;}
	public int getY() { return y;}

	@Override
	public boolean equals(Object obj) {
		return obj instanceof Coordinate && x == ((Coordinate)obj).getX() && y == ((Coordinate)obj).getY();
	}
}
```

也将失去作用。

4、HashMap 与 Hashtable

不同之处一：Hashtable 始于 JDK1.0，HashMap 始于 JDK1.2.

不同之处二：Hashtable 是线程安全的，HashMap 是非线程安全的。

不同之处三：Hashtable 不允许 null 作为 key 和 value，HashMap 则允许 null 作为键和值（由方法 putForNullKey()支持，null 键所代表的键值对永远放置在 index=0 的桶内）。

不同之处四：Hashtable 返回旧版的迭代器 Enumeration，而 HashMap 返回新版的迭代器 Iterator。

如果要在并发环境下使用 HashMap 请参见 ConcurrentHashMap,它采用分区锁的策略来提升效率。

5、StringBuilder 与 StringBuffer

不同之处一：StringBuffer 始于 JDK1.0，StringBuilder 始于 JDK1.5。

不同之处二：StringBuffer 是线程安全的，StringBuilder 是非线程安全的。

在确定线程安全的环境下，应该优先使用 StringBuilder 以获取更好的性能。

6、String

```
String 源码：
public final class String implements java.io.Serializable,Comparable<String>,CharSequence{
    /** The value is used for character storage. */
    private final char value [];
    ...
}
```

String 类型不可被继承。另外，String 类型一旦生成就不可改变，无论是"+"运算符或者是其他修改 String 值的方法，返回的都是一个新的 String 对象。

Java 不允许自定义操作符重载，以降低语言复杂度，但是"+"已经被重载过，可以实现字符串的拼接，以返回一个新的 String 对象，并且，编译器会为我们优化：新建一个 StringBuilder，然后进行拼接，以减少不断创建然后废弃 String 对象的开销。

因此，如果在循环内使用"+"运算符对字符串进行拼接，那么由于编译器的优化，一次循环内会创建并销毁一个 StringBuilder，这会造成大量的时间和空间的浪费，因此，如果需要循环拼接字符串，应当手动使用 StringBuilder 或者 StringBuffer，具体使用谁，还应当视线程安全的要求而定。

另外在覆盖 toString 方法的时候，如果希望打印出对象的内存地址，应当使用 super.toString()而不是使用字符串拼接 this 关键字，这会造成一个没有终止条件的递归，导致爆栈。

关于 Intern:

```
import java.util.Scanner;

public class Test {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String one = scanner.next();
        String two = scanner.next();
        one = one.intern();
        two = two.intern();
        System.out.println("one is "+one+" and two is "+two+" and are they equals ? \n"+(one == two));
        scanner.close();
    }
}
```

使用双引号声明的 String 会作为常量存储在常量池中，而使用 String 构造方法的字符串则会在运行时创建。因此：new String("i am")实际涉及了 2 个 String 对象，一个是"i am"通过""双引号在编译期创建的，存在于常量池中，另一个是通过 new 关键字在运行时创建的。

String.split()方法可以接受一个正则表达式，对 String 对象进行分割，这是 Java1.5 以后的新增 API，用于取代过时难用的 StringTokenizer 类对字符串的操作。

7、Pattern 与 Matcher

使用 String.split()利用正则表达式对字符串分割，非常的简洁方便，但是如果你想使用更强大的正则表达式功能，请使用 Pattern 和 Matcher。实际上，String.split()方法的实现就是利用了这两个类。

请注意，Java 中的正则表达式不同于其他语言这主要是由于 Java 语言对转义字符的处理造成的，\\代表一个正则表达式的反斜线，而\\\\代表一个普通的反斜线，譬如你想匹配一个整数的时候，不能写作\d，而应当写作：\\d，当你相匹配一个换行符的时候应当写\\r，而不是\r，当你想匹配一个逃逸的换行回车的时候，你需要这么写：Pattern.complie("\\\\r\\\\n");

普通的正则表达式的量词后面不再追加任何符号，此时正则表达式是贪婪型的，即尽可能多的匹配目标字符串，使用?后缀则可以描述一个勉强型的正则表达式，即正则表达式刚刚满足就结束匹配。使用+后缀可以描述占有型正则表达式，但是这仅仅在 Java 语言中可用，它比较高级，这里暂且不提。

贪婪型	勉强型	如何匹配
X?	X??	一个或零个 X
X*	X*?	零个或多个 X
X+	X+?	一个或多个 X
X{n}	X{n}?	恰好 n 次 X
X{n,}	X{n,}?	至少 n 次 X
X{n,m}	X{n,m}?	X 至少 n 次，且不超过 m 次

8、Final、Finally、Finalize()

final 关键字，用于修饰变量、方法或者类。修饰基本类型时，变量的值不可改变，修饰引用时，引用所指向的对象不可变。修饰方法时，方法不可被子类重写。修饰类时，类不可被继承。

finally 用于 try-catch-finally 块中，主要用于完成代码块内的资源清理和释放工作，无论 try 或者 catch 块内如何控制，finally 块内的代码一定会执行。

finalize()方法是"万类之祖"Object 类的一个方法，如果一个对象 A 使用了除 new 以外的一些方法（通常是 native 的方法）获取了一块内存，那么你可以在 A 类的 finalize()方法里写清理这块内存的代码，当垃圾回收器准备好回收 A 对象的时候，垃圾回收器会调用一次 A 对象的 finalize()方法，然后在下一次垃圾回收时，才真正的回收 A 对象。

9、Return 与 Finally

如下代码会返回什么呢？

```
public class Test {
    public static void main(String[] args) {
        System.out.println(test());
    }
    public static int test() {
        try {
            return 1;
        } finally {
            return 2;
        }"
    }
}
```

答案：2

这样的结果是《The Java Language Specification》(第三版，第 14 章 17 节)所规定的。

当然，try 语句中有 System.exit(0);这个语句的时候，finally 也不会被执行，因为程序终止了。

10、异常不能跨线程传播

异常是不能跨线程传播的，实验代码如下：

```
public class Test {
	public static void main(String[] args) {
		Thread thread = new Thread(new Job());
		try {
			thread.start();
		} catch (Exception e) {
			System.out.println("Catch Exception !");
		}
	}
}

class Job implements Runnable{
	@Override
	public void run() {
		throw new NullPointerException();
	}
}
```

Thread.UncaughtExceptionHandler 是 Java SE5 中的新接口，它允许你在每个 Thread 对象上都附着一个异常处理器。Thread.UncaughtExceptionHandler.uncaughtException()会在线程因未捕获的异常而临近死亡时调用。

11、闭包

闭包是一个可调用的对象，它记录了一些信息(引用了一些变量)，这些信息(变量)来自于创建它的作用域。——<<Java 编程思想>>

Java 语言中也是可以使用闭包的，但是不同于 C/C++，Java 语言的闭包并不是通过指针实现的，而是通过匿名内部类来实现的，这种实现方式更加安全，防止了误操作指针导致的错误。

我们来创建一个例子，这个例子中，有一个 Village 类和一个 Canal 接口，Village 类拥有一些公开的“渠”(Canal)，大家都可以饮用任意一个渠里的东西(Canal.obtainDrink())，Village 类还具有一个水源 drinkSource，当然，水源这种东西必须受到保护，必须是私有的，只有有权利的人才可以为大家更换水源。

我们要达到的目的是，虽然大家都可以从渠中获取饮品，但是，却不能污染这个渠（也就是不能随意修改水源），并且，村长或者某位有权利的人，为大家更换了水源后，大家能够立刻饮用到新的饮品，也就是说渠里的饮品必须立刻发生变化（你总不想再喝被发现有毒的饮料吧）：

为了实现上述目标，考虑一下：最朴素的实现方式是，为渠道(简称为 Canal 类)设置一个水源属性，并且把 Canal 类的 drinkSource 的 setter 方法只暴露给有权限的类，这样，当每次其他类调用 Canal 类的 obtainDrink()方法时，Canal 类都会返回自己的 drinkSource，当变换水源时，只要 Village 类中存有一个 Canal 类的集合，遍历这个集合，更换 drinkSource，就可以为所有的 Canal 更换水源了。

但是，上述方法有一个问题，由于我们是遍历所有 Canal 来更换水源，因此当 Canal 较多时，第一个 Canal 会比最后一个 Canal 更早的喝到新的饮品，这和我们的目标是相悖的，我们的目标是所有人都能立刻饮用新的饮品，究其原因实际上是因为我们创建了多个 drinkSource 副本，并将其放入了 Canal 类中，导致我们不能同时更新所有 drinkSource 而产生了时间差。并且，这样的做法也与实际不符，实际情况下，并不是一条渠对应一个水源，更常见的是村里有很多渠道，大家共用一个水源。

所以，我们保证有一个水源，也保证同时更新所有渠道内的饮品，我们将使用闭包来解决这一问题。

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Closure {
    public static void main(String[] args) {
        Village a = new Village("Water");
        Canal canal = a.anyCanal();
        System.out.println(canal.obtainDrink());
        a.changeSource("Wine");
        System.out.println(canal.obtainDrink());
        a = null;
        System.gc();
        for (int i = 0; i < 1000; i++)
            byte[] tmp = new byte[8192*1024];
        System.out.println(canal.obtainDrink());
    }
}

class Village{
    public static final int INIT_CANAL_COUNT = 3;
    public final List<Canal> canals;
    private String drinkSource;
    private Random random;

    Village(String d){
        this.random = new Random();
        this.drinkSource = d;
        this.canals = new ArrayList<Canal>(INIT_CANAL_COUNT);
        for (int i = 0; i < INIT_CANAL_COUNT; i++) {
            canals.add(new Canal(){
                @Override
                public String obtainDrink() {
                    return drinkSource;
                }
            });
        }
    }

    void changeSource(String newDrink){
        drinkSource = newDrink;
    }

    public Canal anyCanal(){
        return canals.get(random.nextInt(INIT_CANAL_COUNT));
    }
}
interface Canal{
    public String obtainDrink();
}
```

这个例子中，任何人都可以调用 anyCanal()方法，找到一条水渠，然后调用 obtainCanal 方法，获得饮品，并且如果村子中需要更换水源，我们只要修改 Village 类的 drinkSource 属性就行了，由于闭包的缘故，即使表面上看到，被分配出去的 Canal 没有与 Village 产生任何联系，但实际上，Canal 始终在引用 Village 类内部 drinkSource 属性，因此只要 Village 内部的 drinkSource 发生变化，我们无需任何手动操作，所有 Canal 返回的饮品，就同时发生了变化。

最后，注意 main 方法中，我们释放了 Village 的实例(这是通过解除引用，调用 gc，和消耗了 JVM 的 4000M 内存做到的)，观察控制台输出，发现，即便 Village 被释放，Canal 类仍然能够返回饮品，这便是闭包的另一个特性。

二、流与 I/O

1、流与 I/O(Input/Output)概述

“编程语言的 I/O 类库中常使用“流”这个抽象概念，它代表任何有能力产出数据的数据源对象或者是有能力接收数据的接收端对象。“流”屏蔽了实际的 I/O 设备中处理数据的细节”——摘自《Java 编程思想》。流分为字节流和字符流：他们的代表分别是：InputStream/OutputStream 和 Reader/Writer。

面向字符的流出现于 JDK1.1，他们的出现主要是为了解决 I/O 中的国际化问题，使用 Reader/Writer，可以在所有的 I/O 操作中都支持 Unicode，同时，从效率上来说，他们比原有的类库操作更快，但是，存在一些情况可能确实需要使用 InputStream/OutputStream，所以，总体的原则是：尽量尝试使用 Reader 或者 Writer，不得已时再使用面向字节的类库。另外，针对字节流，有将其转化为字符流的适配器：InputStreamReader/OutputStreamWriter

2、字节流使用示例

这个程序将读取一个文本文件，并为其每一行加上行号，形式如：“1.”

```
import java.io.*;
import java.util.LinkedList;

public class Main {
    private static final byte dot = (byte)46;
    public static void main(String[] args) throws IOException {
        File file = new File("/Users/zl/Desktop/blank");
        BufferedInputStream bi = new BufferedInputStream(new FileInputStream(file));
        LinkedList<Byte> bytes = new LinkedList<Byte>();
        bytes.add((byte)49);
        bytes.add(dot);
        int i = 0;
        Integer line = 1;

        while((i = bi.read()) != -1){
            bytes.add((byte)i);
            if (i == 10) {
                line++;
                byte[] bs = line.toString().getBytes();
                for (byte c : bs) bytes.add(c);
                bytes.add(dot);
            }
        }
```

```
            bi.close();

            byte[] content = new byte[bytes.size()];
            for (byte c : bytes) content[++i] = c;

            File file2 = new File("/Users/zl/Desktop/blank2");
            if(!file2.exists()) file2.createNewFile();

            BufferedOutputStream bo = new BufferedOutputStream(new FileOutputStream(file2));
            bo.write(content);
            bo.close();
        }
    }
```

3、字符流使用示例

这个程序将读取一个文本文件，并为其每一行加上行号，形式如："1、 "

```
    import java.io.BufferedReader;
    import java.io.BufferedWriter;
    import java.io.File;
    import java.io.FileReader;
    import java.io.FileWriter;
    import java.io.IOException;

    public class Test {
        public static void main(String[] args) throws IOException {
            File blank2 = new File("/Users/zl/Desktop/blank2");
            File blank3 = new File("/Users/zl/Desktop/blank3");
            String lineSperator = System.getProperty("line.separator","\n");
            StringBuilder sb = new StringBuilder();

            String line = null;
            BufferedReader br = new BufferedReader(new FileReader(blank2));
            for (int i = 1; (line = br.readLine()) != null; i++){
                sb.append(i).append("、 ").append(line).append(lineSperator);
            }
            br.close();

            BufferedWriter bw = new BufferedWriter(new FileWriter(blank3));
            bw.write(sb.toString());
            bw.close();
        }
    }
```

4、一个独立自我的类：RandomAccessFile

RandomAccessFile 有些特立独行，该类并不处于 InputStream/OutputStream 继承结构中，而是一个非常独立的类，他仅仅实现了 DataInput/DataOutput 以及 Closeable 接口，并且其大部分方法都是 native 的实现，对 RandomAccessFile 的操作类似操作一个大的 byte 数组。它所提供的 API 更像是 C 语言中的文件操作 API，打开一个 RandomAccessFile 需要像 C 语言那样指定模式，r,w,r/w，同样可以利用 seek 方法移动文件指针。

用于演示示例程序，假设一个格式已知的文件：specialFile，内容如下：(其中每行 4 个字母和一个换行符)

```
abcd
efgh
ijkl
mnop
```

示例文本只有 20 个字符，我们用下面的程序读取行末最后一个字符，然后将该行行首字母改为行末字母，可以看到只要我们知道文件格式或者是想要读取的内容的位置，使用 RandomAccessFile 是非常方便的。

```
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccessFileTest {

	public static void main(String[] args) throws IOException {
		RandomAccessFile rf = new RandomAccessFile(new File("/Users/zl/Desktop/specialFile"), "rw");
		for (int i = 4,byte tmp = 0; i > 0; i--) {
			rf.seek(i*5-2);
			tmp = rf.readByte();
			rf.seek(i*5-5);
			rf.write(tmp);
		}
		rf.close();
	}
}
```

5、ObjectInputStream、ObjectOutputStream 以及序列化

没错，对象也可以写入流中，并再从流中读出，这样的支持使得我们可以方便的存储或者传输对象而不用在传输过程中自己手动拆解和拼装对象。Java 的序列化机制甚至帮我们保存了对象之间的引用，也就是说我们可以传输一批对象，并且保证其引用关系不变。

示例程序如下：

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class ObjStreamTest {
	public static void main(String[] args) throws IOException, ClassNotFoundException {
		Person me = new Person("c", null);
		Person dad = new Person("b", me);
		Person grandpa = new Person("a", dad);

		ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("/Users/zl/Desk-
top/family"));
		oos.writeObject(me);
		oos.writeObject(dad);
		oos.writeObject(grandpa);
		oos.close();
```

```
            ObjectInputStream ois = new ObjectInputStream(new FileInputStream("/Users/zl/Desktop/family"));
            Person mePerson = (Person)ois.readObject();
            Person dadPerson = (Person)ois.readObject();
            Person grandpaPerson = (Person)ois.readObject();
            ois.close();

            System.out.println(grandpaPerson.son.son.name);
        }
        private static class Person implements Serializable{
            private static final long serialVersionUID = -3360493026896549112L;
            public final String name;
            public final Person son;

            public Person(String name, Person son) {
                this.name = name;
                this.son = son;
            }
        }
    }
```

利用这种特性，Java 中的对象可以方便的通过网络传输，或者存储于像 MemCached 这样的缓存系统。当然，程序写入之后，使用文本编辑器打开/Users/zl/Desktop/family 这个文件，你会发现这是一个二进制文件。

三、NIO

1、NIO 概述

Java NIO 是一种新的 I/O 模型和类库，出现于 Java1.4，是 new I/O 的简写，比起 1.0 和 1.1 出现的字节、字符流等 API，NIO 确实算是新的 IO 支持。

NIO 采用一种更接近操作系统的方式来读写文件，因而出现了两个新概念：Buffer 和 Channel，Buffer 就像一辆数据矿车，奔跑在一条 Channel 矿道上，传送数据，这样的模型因为更加接近操作系统本身的文件 I/O 机制，从而提升了 I/O 效率（这主要是因为 NIO 把 I/O 过程中最耗时的缓冲填充和提取过程，交由操作系统本身完成，从而提高了速度），旧式的 I/O 类库也已经由 NIO 重新实现以提高新能，并且，旧的类库中有 3 个类被修改了，用以产生 FileChannel，他们分别是 FileInputStream,FileOutputStream 以及 RandomAccessFile。

在 NIO 中，我们不和流打交道，也不直接从 Channel 中获取数据，而是提供一个 Buffer，交由 Channel 把数据读入 Buffer 中，然后我们再操作 Buffer 获取内容，或者把 Buffer 交给 Channel，通过 Channel 写入 Buffer 中的内容。ByteBuffer 是基础 Buffer，其他 Buffer 都是 ByteBuffer 的特定数据类型的视图，而对于 Channel，这里我们只探讨针对文件的读写，也就是只研究 FileChannel。

2、ByteBuffer

首先阐释 mark,position,limit,capacity 的概念：

mark 是 ByteBuffer 在被读取过程中所做的标记，以便寻找，然后重新回到被标记的位置开始读取。
position 是当前读取操作所正在读取的位置。
limit 是读取的限度，避免你读到空白或是脏 byte。它是 ByteBuffer 在写模式下的最后一个 position。
capacity 是 Buffer 的最大容量，limit 不可能超过这个值，limit 等于 capacity 时，Buffer 就被写满了。

ByteBuffer 的基本操作包含 allocate(allocateDirect),wrap,flip,rewind,clear,mark,reset,position 等。

1、allocate(allocateDirect)按照你所指定的大小分配一个非直接（直接）缓冲区，操作系统会直接操作直接缓冲区而不使用其他缓冲区，因此直接缓冲区性能高于非直接缓冲区。

2、wrap 包装 byte[]成为一个 ByteBuffer。

3、flip 转换 ByteBuffer 的状态，使其就绪，准备被读取也就是 limit=position,position=0,mark=-1。

4、rewind 恢复 ByteBuffer 的被读取时的初态，也就是 position = 0,mark = -1,使其可以被重新读取。

5、clear 清除 ByteBuffer 的内容，准备重新被写入也就是 position = 0,mark = -1,limit = capacity。

6、mark 对 ByteBuffer 内当前正在被读取的位置进行标记。

7、reset 使当前的读取位置回到 mark 被标记的位置，如果 mark 并未标记过，则引发一个 Invalid-MarkException 异常。

8、position 返回当前读取到的位置。

3、FileChannel

Channel 有点像 Stream 的概念，数据都是通过 Channel 或者 Stream 传输的，不过 Channel 是双向的，Stream 是单向的，Channel 传递 Buffer 作为数据载体，Stream 仅提供读写操作，不显示使用 Buffer 的概念。

FileChannel 的基础方法包含一系列的 read 和 write，并且这些方法仅操作 ByteBuffer，当读取文件结束时，read 方法返回-1 表示读取结束。FileChannel 从 FileInputStream/FileOutputStream 中获取，并且，像流一样，FileChannel 也需要关闭。

4、NIO 读取文件示例

仍旧是给文本文件的每一行，加上行号：

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class AddLineNumber2 {
        private static final int BUF_SIZE = 1024;
        public static void main(String[] args) throws IOException {
        File srcFile = new File("/Users/zl/Desktop/blank2");
        FileInputStream fis = new FileInputStream(srcFile);
        FileOutputStream fos = new FileOutputStream(“/Users/zl/Desktop/blank3");
        FileChannel rc = fis.getChannel();
        FileChannel wc = fos.getChannel();

        if (srcFile.length() > 0)
                wc.write(ByteBuffer.wrap(("1、 ").getBytes()));
```

```
        ByteBuffer oldBuffer = ByteBuffer.allocate(BUF_SIZE);
        byte[] newBuffer;
        int line = 1;

        while (rc.read(oldBuffer) != -1) {
            newBuffer = new byte[6*BUF_SIZE];
            oldBuffer.flip();
            int limit = oldBuffer.limit();

            for (int i = 0;oldBuffer.position() != limit;i++) {
                newBuffer[i] = oldBuffer.get();
                if (newBuffer[i] == 10) {
                    byte[] indicator = (++line + "、 ").getBytes();
                    System.arraycopy(indicator, 0, newBuffer, i+1, indicator.length);
                    i += indicator.length;
                }
            }
            wc.write(ByteBuffer.wrap(newBuffer));
            oldBuffer.clear();
        }
        fis.close();
        fos.close();
        rc.close();
        wc.close();
        }
}
```

5、视图缓冲器

CharBuffer，IntBuffer 等这些类都是视图缓冲器，它可以让我们通过某个特定的基本数据类型的视窗查看底层的 ByteBuffer 中的数据内容。ByteBuffer 依然是底层存储数据的地方，“支持”着前面的视图，因此，我们对视图的任何修改都会映射成为对 ByteBuffer 的修改，各种类型的视图缓冲器可以通过 asXXXBuffer 系列方法得到。

下面演示一个使用 CharBuffer 读取文件，计算文件行数并输出到控制台的程序：

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.FileChannel;
import java.nio.charset.Charset;

public class CharBufferTest {
    private static final int BUFF_SIZE = 1024;
    private static final char LINE = 10;
    public static void main(String[] args) throws IOException {
        File file = new File("/Users/zl/Desktop/blank2");

        FileInputStream fis = new FileInputStream(file);
        FileChannel fc = fis.getChannel();
        Charset charset = Charset.forName("UTF-8");
        ByteBuffer buffer = ByteBuffer.allocate(BUFF_SIZE);
        CharBuffer cb = null;
```

```
                int lines = file.length() > 0 ? 1 : 0;

                while (fc.read(buffer) != -1) {
                        buffer.flip();
                        cb = charset.decode(buffer);
                        for (int i = 0; i < cb.limit(); i++) {
                                if (cb.get() == LINE)
                                        lines++;
                        }
                        buffer.clear();
                }
                fis.close();
                System.out.println(lines);
        }
}
```

6、MappedByteBuffer

MappedByteBuffer 是 ByteBuffer 的一个子类，其目的是为了支持文件映射，通过文件映射，可以方便的读写大文件，因为除了你正在读写的部分，文件的其他部分都被操作系统换入换出了,通过文件映射来操作文件内容，相比于旧式的流操作，会获得更好的性能。

7、FileLock

FileLock 可以为文件上锁，Java 虚拟机直接将锁操作映射在了操作系统本地的上锁工具上，所以，这种锁对于不同进程是有效的，但是对于 Java 虚拟机内的不同线程，这个锁并不能限定特定线程的访问。FileChannel 有 tryLock()和 lock()两种方法为一个文件上锁，后一种方法是阻塞的，前一种方法如果上锁失败则结束调用，FileChannel 不仅可以对整个文件上锁，还可以对文件的某个区域上锁，但是，即使文件大小发生变化，锁的区域并不会发生变化，如果程序操作了锁区以外的内容，FileLock 不保证不会发生竞争条件。关闭一个通道会释放这个通道所对应文件的所有锁定，无论这些锁是否是由这个通道获得的。

四、多线程与并发

1、线程概述

线程是比进程更小的任务单位，是最小的 CPU 执行单元，即单核 CPU 任意一时刻，只能执行一个线程。线程依附于进程存在，可以共享进程资源，线程自己并不享有独立的系统资源，但是它自身包含一些运行所必须的资源，包括程序计数器，寄存器，堆栈等。

你可以通过 Java 语言向 JVM 申请进程。Java 语言中 Thread 的这个类代表了一个线程实体。我们可以通过以下两种方式来创建自定义处理逻辑的线程：

1、继承 Thread 类，并覆盖 run 方法；
2、自己创建一个类并实现 Runnable 接口，然后创建该类对象，将对象传入 Thread 类的构造函数。

启动一个线程可以通过下面两步：

1、创建一个 Thread 对象。
2、thread.start();

来看一个例子，这个例子使用递归计算斐波那契数列的第 n 个数是多少（n 从 0 开始）：

```
class Fibonacci implements Runnable{

	private short index;
	private long f;
	private ThreadHolder holder;

	public Fibonacci(short index, ThreadHolder holder) {
		this.index = index;
		this.holder = holder;
	}

	@Override
	public void run() {
		f = fibonacci(index);
		holder.receive(f,Thread.currentThread().getName());
	}

	private long fibonacci(int i) {
		return i < 2 ? 1 : fibonacci(i-1) + fibonacci(i-2);
	}
}
```

考虑到使用递归计算斐波那契数列的第 n 个数会比较慢，并且我们目前所使用的计算机大多是多核的 CPU，这里我们使用了线程来加速计算过程，并且不会阻塞主线程，主线程还可以用来做其他事情（比如绘制用户界面），当然，这里 main 函数再启动了 2 个线程之后，没有其他事情做，就退出了。

我们再看一个关于线程休眠的例子。

2、守护线程与休眠

守护线程（也叫后台线程）是一种特殊的线程，当 JVM 中不存在任何非守护线程时，程序就会退出，甚至不会执行 finally 代码块。因此守护线程特别适合用于监视之类的任务。我们可以在线程创建后，启动前，使用 setDeamon()方法指定一个线程为守护线程。注意，当线程启动后，就不能设置线程为守护线程了，否则会抛出 IllegalThreadStateException 异常。

线程可以在一定时间内不执行，我们称之为挂起或者休眠，Thread.sleep(mills,nano)就可以指定当前线程休眠多少毫秒，多少纳秒。该方法只能休眠当前正在运行的线程，可以理解为只能休眠自己这个线程，不能指定其他线程休眠。

Thread.sleep()虽然可以指定线程休眠多少时间，但是传入的长整型 mills 参数，导致这种使用方法可读性很差，经常需要一定的计算才能知道线程被指定休眠多长时间，为了代码的可读性，推荐使用 TimeUnit.<Unit>.sleep()方法(像 TimeUnit.MILLSECONDS.sleep()或 TimeUnit.SECONDS.sleep())。

线程休眠时，是有可能被打断的，如果线程在休眠过程中被打断，那么会抛出 InterruptedException 异常。来看一个守护线程与休眠线程的例子。

```
import java.util.concurrent.TimeUnit;

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

```
            for (int i = 0; i < 10; i++) {
                Thread thread = new Thread(new Task());
                thread.setDaemon(true);
                thread.start();
            }
            System.out.println("All Deamon thread has start !");
            try {
                TimeUnit.MILLISECONDS.sleep(175);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    class Task implements Runnable{
        @Override
        public void run() {
            try {
                System.out.println(Thread.currentThread().getName()+" will sleep !");
                TimeUnit.MILLISECONDS.sleep(300);
                System.out.println(Thread.currentThread().getName()+" has awakened !");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

这个例子启动了 10 个守护线程，并且每个守护线程在启动后向控制台输出了一句话后，就休眠了，但是由于时间关系，主线程在所有休眠线程都启动后，休眠了一段时间，就运行完毕了，所以整个程序也退出了。因此我们观察控制台输出，只有守护线程开始休眠的输出，却没有守护线程休眠完毕的输出。（如果你能看到守护线程的"睡醒"输出，那么请把 300 秒再改大些^_^）

我们可以通过调用 isDaemon()方法来判断一个线程是否是守护线程，另外，由守护线程创建的线程，会被自动置为守护线程。

3、Join 与 Interrupt

提起 interrupt，你可能会想到上文提及的"线程在睡眠时被打断，则抛出 InterruptedException 异常"，这里我们会讲解 thread.interrupt()方法。但是在这之前，我们先看看 thread.join()方法。

假设有两个线程 A 和 B，B 由 A 启动，我们希望 A 启动 B 后，A 就不再运行，直到 B 运行完成后，A 再接着运行。为了达到这样的效果，我们可以使用 B.join()这样的方法。

例子：

```
    import java.util.Random;
    import java.util.concurrent.TimeUnit;

    public class JoinAndInterrupt {
        public static void main(String[] args) {
            JoinAndInterrupt jai = new JoinAndInterrupt();
            Thread thread1 = new Thread(jai.new Task());
            Thread thread2 = new Thread(jai.new Task());
            Thread thread3 = new Thread(jai.new Interrupter(Thread.currentThread()));
            thread1.start();
            thread2.start();
```

```
//            thread3.start();
            try {
                thread1.join();
                thread2.join();
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName()+" was interrupted !");
            }

            System.out.println(Thread.currentThread().getName()+" exit !");
        }

        class Task implements Runnable{

            @Override
            public void run() {
                int index = 45+new Random().nextInt(3);
                System.out.println(Thread.currentThread().getName() +" : "+ fibonacci(index));
            }

            public long fibonacci(int index){
                return index < 2 ? 1 : fibonacci(index - 1) + fibonacci(index -2);
            }
        }

        class Interrupter implements Runnable{

            private Thread target;

            public Interrupter(Thread t) {
                target = t;
            }

            @Override


            public void run() {
                try {
                    TimeUnit.SECONDS.sleep(2);
                } catch (InterruptedException e) {
                    System.err.println("Interrupter was interrupted ! ");
                }
                target.interrupt();
            }
        }
    }
```

输出：

```
Thread-1 : 1836311903
Thread-0 : 2971215073
main exit !
```

忽略 Interrupter 这个类，我们可以看到，main 线程确实是在 Thread-0(thread1)和 Thread-1(thread2)计算完成后退出的。如果我们删除掉 thread2.join()这句，当 thread1 在 run()运行的时候，随机出的 index 小于 thread2 的时候，我们会发现 Thread-0 运行结束后 main 线程就退出了，而没有继续等待 thread2。

你会注意到，上面的 join 方法被 try-catch 语句包裹着，这是因为 join 方法是可以被打断(interrupt 方法)的，一旦正在等待同步的线程（就是调动别的线程 join 方法的线程）被打断，那将会抛出一个 InterruptException，针对上面的例子，我们打开注释：//thread3.start();让 thread3 也启动起来，我们就会看到，在 Interrupter 线程挂起 2 秒后，它打断了主线程，然后主线程随机退出，不再等待 thread1 和 thread2 对斐波那契数列的计算。

输出如下：

```
main was interrupted !
main exit !
Thread-1 : 1836311903
Thread-0 : 4807526976
```

现在，我们看到了 join 的用法和效果，我们再来看看 interrupt()方法：

interrupt

```
public void interrupt()
```

中断线程。

如果当前线程没有中断它自己（这在任何情况下都是允许的）

JDK 文档指明，interrupt()方法不一定会中断它自己，这是可以理解的，因为我们不知道运行中的线程正在一个怎样的执行过程，占用了怎样的资源，我们直接将其打断，可能会造成无法想象的后果，比如其所占有的资源得不到释放和清理，甚至导致死锁。来看一个例子，我们使用递归来计算斐波那契数列的第 n 个数是多少：

```
public class JoinAndInterrupt {
    public static void main(String[] args) {
        JoinAndInterrupt jai = new JoinAndInterrupt();
        Thread thread = new Thread(jai.new Task());
        thread.start();
        thread.interrupt();
        System.out.println(Thread.currentThread().getName()+" exit !");
    }

    class Task implements Runnable{

        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName()+" : "+fibonacci(46));
        }

        public long fibonacci(int index){
            return index < 2 ? 1: fibonacci(index - 1)+fibonacci(index - 2);
        }
    }
}
```

输出：

```
main exit !
Thread-0 : 2971215073
```

没错，Thread-0 还是“倔强”的算完了结果，主线程对 thread.interrupt()的调用并没有起到实质性作用，正如上面所讲，Thread-0 应该这么“倔强”，对于本例来说，如果强行打断 Thread-0 的计算过程，那么用于递归计算的栈将得不到清理，这是绝对不能允许的状况。对于其他例子，Thread 可能占有着系统关键资源的锁，如果将其强行打断，可能会使系统陷入死锁的状态。

当然，有人会说，我们可以在线程被打断的时候抛出一个异常，然后在异常中释放该线程所占有的资源。这么做也是不行的，由于我们不知道代码何时会被打断，因此我们也不知道代码在被打断时，占用了哪些资源，所以这会使代码的复杂程度加剧，开发难度上升，这也是不容乐观的情况。

当然，我们还是要想办法，使得线程可以在运行过程中退出。目前 Java 推荐的做法是：一般来说，我们在 run()方法中会有某种形式的循环，去重复执行某些任务，我们应当在循环中检查一个变量，来判断是否需要继续计算，如果不需要计算，则跳出循环，执行完 run()方法后退出。

我们仍然以计算斐波那契数列作为例子：我们让子线程不断的计算斐波那契数列的下一个元素（从 2 开始），而主线程不断的随机一个 10 以内的整数，当主线程随机出 9 的时候，子线程停止计算，不过，为了能随时中断计算，我们采用非递归的方法来计算斐波那契数列：

```
import java.util.Random;

public class CommonRunnable {
	public static void main(String[] args) {
		CommonRunnableImpl cri = new CommonRunnableImpl();
		new Thread(cri).start();
		Random random = new Random();
		while (true) {
			if(random.nextInt(10) > 8){
				cri.stop();
				break;
			}
			for (int i = 0; i < 100000; i++);//用于减速随机数的生成过程
		}
		System.out.println(cri.currentResult());
	}
}

class CommonRunnableImpl implements Runnable {

	private volatile boolean isContinue = true;
	private long[] nums = {1,1,2};
	private int index = 2;

	@Override
	public void run() {
		try {
			while (isContinue) {
				nums[index] = nums[(index+1)%3] + nums[(index+2)%3];
				index = (index + 1)%3;
				for (int i = 0; i < 50000; i++);//用于减速斐波那契数列的计算速度
			}
		} catch (Exception e) {
			System.err.println("Calculating fibonaci numbers occurs an error !");
		}finally{
			//clean code
		}
	}
```

```
        public void stop(){
            isContinue = false;
        }

        public long currentResult(){
            return nums[(index + 2)%3];
        }
    }
```

请注意，两个 while 循环中的 for 循环，循环体是空的，这样做是为了降低两个 while 循环的速度（因为获取随机数和用非递归方法求 fibonacci 数列的下一个数，这两个操作实在是太快了，如果你希望在你的计算机上运行这个程序，那么两个 for 循环的值你可能需要调整下，以求一个恰当的效果）。

如你所见，这个程序可以随时的中断 fibonacci 数列的计算，并且安全的退出，这样周期性的检查指示变量的做法虽然略显笨拙，但是它不会轻易的导致死锁，也不会使编程的复杂度飙升，综合来说，这是命令线程退出的最好的方法了。

4、异常处理

第一章第 10 节提到过，异常不能跨线程传播，但是我们可以为每个线程附着一个特定的异常处理器，就像下面这样：

```
import java.util.concurrent.TimeUnit;

public class ThreadException {
    public static void main(String[] args) {
        StrongTask st = new StrongTask();
        Thread t = new Thread(st);
        t.setUncaughtExceptionHandler(st);
//          Thread.setDefaultUncaughtExceptionHandler(st);
        t.start();
        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {}
        t.interrupt();
    }
}
class StrongTask implements Runnable,Thread.UncaughtExceptionHandler{

    @Override
    public void uncaughtException(Thread t, Throwable e) {
        System.err.println(t.getName()+e.getMessage());
    }

    @Override
    public void run() {
        try {
            TimeUnit.SECONDS.sleep(10);
        } catch (InterruptedException e) {
            throw new RuntimeException(" was Interrupt !");
        }
    }
}
```

当线程发生异常却没有捕获，异常处理器的 uncaughtException 方法就会被调用。如果我们想为所有的线程都设置一样的异常处理器，那么可以使用 Thread.setDefaultUncaughtExceptionHandler()方法来设置默认的

异常处理器，只有当线程没有属于自己的专属异常处理器时，异常才会交由被设置的全局异常处理器来处理。

5、竞争条件与资源同步

线程是 cpu 最小的执行单位的抽象，代表着计算资源，而计算资源并不与数据绑定，也就是说不同的线程可以使用同一份数据进行计算。（想象一下，一个物理多核 CPU 中，每个核都跑着一个线程，执行着同一份代码，访问着同一个对象）

不同线程操作同一份数据可能会出现问题，考虑一下这种情况：当 A 线程对数据的操作还未完成，B 线程就开始了读取，并且先于 A 线程把操作结果写回内存，然后，A 线程再把自己的计算结果写回内存同一处，那么这时，B 线程的操作结果就被覆盖了，看起来就好像 B 的计算没有进行一样。

我们称上面的这种情况为竞争条件（race condition）。我们如果想要写出正确并发代码，我们就必须杜绝竞争条件的发生。杜绝竞争条件的发生，就是并发编程的一个核心问题，为了解决这个问题，出现了很多不同的解决思路，比如：序列化访问共享资源，函数式编程等。

Java 语言是指令式的，并且使用线程作为并发编程的基本模型，因此它解决竞争条件的思路是指定某时刻，只能有一个任务访问共享资源，比如，针对上面 AB 线程的例子，只要在 A 线程写入完成之前，阻止 B 线程进行读写操作，那么程序就能正确执行。

我们把操作共享资源的代码片段称作临界区，那么，我们就能发现问题的关键：临界区的代码不能被拆散执行，必须具有原子性，要么都执行，要么都不执行。并且，一旦一个线程进入临界区，那么其它线程就不应该再进入同一临界区。（同一临界区是指操作相同的共享变量）

在一些没有在语法上支持同步的语言里，我们必须手动说明一个代码片段是临界区，这要求我们对访问的资源加锁，甚至对锁上锁，以保证此刻自己是唯一能访问共享资源的任务。这无疑会让事情变得复杂，幸好，Java 语言在语言层面对同步访问资源做了支持，我们可以使用 synchronized 关键字说明一个代码块是临界区。

我们还是先来看看竞争条件的示例：

```
public class RaceConditionAndSynchronize {

    public static void main(String[] args) {
        RaceConditionAndSynchronize rcas = new RaceConditionAndSynchronize();
        Zero zero = rcas.new Zero();
        rcas.new ZeroChecker(zero);
        rcas.new ZeroChecker(zero);
    }

    class Zero {
        private volatile int current;
        private volatile boolean isCanceled;
        private Object lock = System.out;

        public int current(){
            synchronized (lock) {
                System.out.println(thread()+" in method , before --     : "+current);
            }
            --current;
            synchronized (lock) {
                System.out.println(thread()+" after --, before yield() : "+current);
            }
```

```
                Thread.yield();
                synchronized (lock) {
                        System.out.println(thread()+" after yield, before ++    : "+current);
                }
                ++current;
                synchronized (lock) {
                        System.out.println(thread()+" after ++,  before return  : "+current);
                }
                return current;
        }

        public boolean isCanceled() {
                return isCanceled;
        }

        public int cancel(){
                synchronized (lock) {
                        if (!isCanceled)
                                System.out.println(thread()+" canceled checking !");
                }
                isCanceled = true;
                return current;
        }

        private String thread() {
                return Thread.currentThread().getName();
        }
}
class ZeroChecker implements Runnable{
        private Zero zero;
        public ZeroChecker(Zero z) {
                zero = z;
                new Thread(this).start();
        }

        @Override
        public void run() {
                while (!zero.isCanceled()){
                        if (zero.current() != 0)
                                zero.cancel();
                }
                System.err.println(Thread.currentThread().getName()+": current is :"+zero.cancel());
        }
}
```

这段代码中，有一个 Zero 类，它的 current 方法返回一个数字，我们期望它返回 0，因为在 current 方法内我们对 current 变量先减 1 后加 1，那么理所应当，它还应该是 0；Zero 类还有一个 isCanceled 变量，用于标识 ZeroChecker 类是否还需要继续对 Zero 实例检查，同时还提供一个 cancel()方法，来设置 isCanceled 变量。

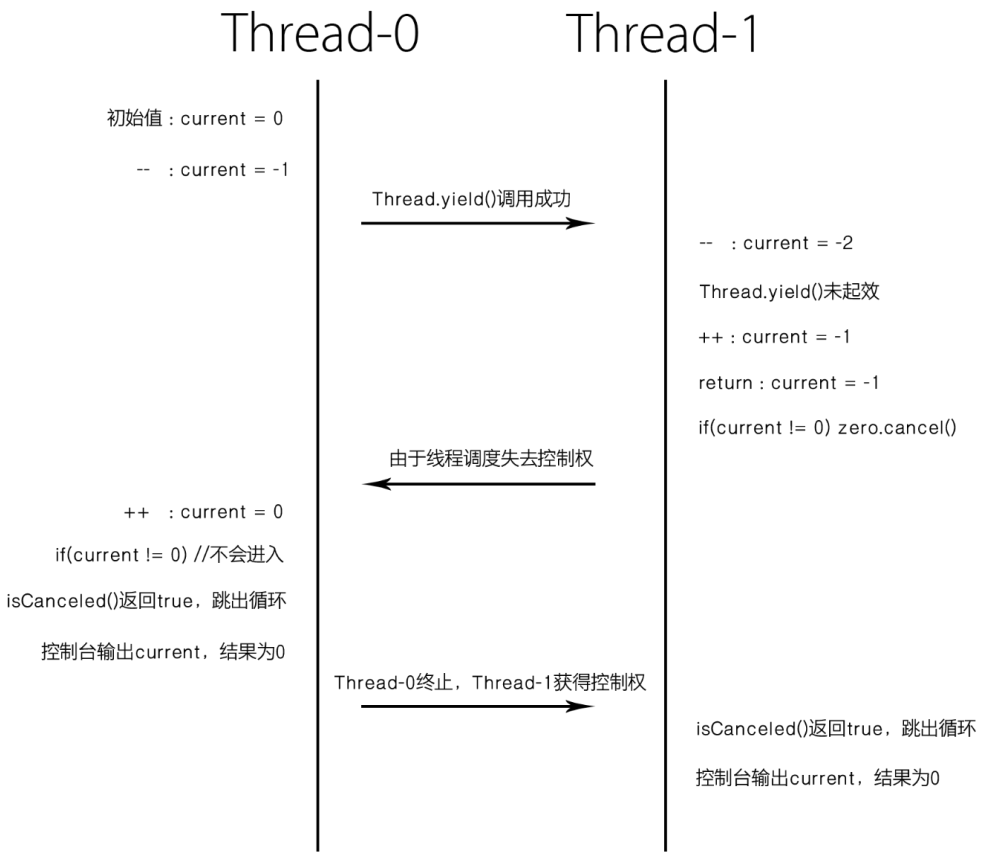
ZeroChecker 类实现了 Runnable 接口，它的构造方法会启动一个以自己为业务逻辑的线程，它的 run()方法会不断的检查 Zero 的实例的 current()方法返回的值是否为 0，如果不为零，立刻调用 cancel()方法，这样其它线程也会停止检查，然后退出运行。

那么 zero.current()的返回值难道可能不为 0？我们多运行几次，可能会得到几种不同的输出，这是因为 JVM 对线程的调度是不确定的。以下两种结果比较典型：

```
Thread-0 in method , before --     : 0
Thread-0 after -- , before yield() : -1
Thread-1 in method , before --     : -1
Thread-1 after -- , before yield() : -2
Thread-0 after yield, before ++    : -2
Thread-0 after ++,  before return  : -1
Thread-0 canceled checking !
Thread-0: current is :-1
Thread-1 after yield, before ++    : -1
Thread-1 after ++,  before return  : 0
Thread-1: current is :0

Thread-0 in method , before --     : 0
Thread-0 after -- , before yield() : -1
Thread-1 in method , before --     : -1
Thread-1 after -- , before yield() : -2
Thread-1 after yield, before ++    : -2
Thread-1 after ++,  before return  : -1
Thread-1 canceled checking !
Thread-0 after yield, before ++    : -1
Thread-0 after ++,  before return  : 0
Thread-0: current is :0
Thread-1: current is :0
```

可以从输出看出 ZeroChecker 终止检查的原因：一个线程没能原子的完成- -和++操作，导致另一个线程在运行时检测到 current 的值非 0，因此退出。以第二个输出为例，用示意图阐释两个线程的调度和执行过程：



我们如果需保证 current 方法永远返回 0（即保证 current 方法内的代码原子的执行），那么我们就应该把 current()方法内的代码声明为临界区的。这时我们可以使用 synchronized 关键字，修改 current()方法如下：

```
public int current(){
    synchronized (lock) {
        System.out.println(thread()+" in method , before --    : "+current);
        --current;
        System.out.println(thread()+" after -- , before yield() : "+current);
        Thread.yield();
        System.out.println(thread()+" after yield, before ++    : "+current);
        ++current;
        System.out.println(thread()+" after ++,  before return  : "+current);
        return current;
    }
}
```

再次运行程序，这时候我们会发现程序停不下来了，我们除非手动的杀死这个进程。我们观察输出，发现一个线程对 current 的变量的操作，总是原子的，有--必有++。

6、Synchronized

当任务要执行被 synchronized 关键字保护的代码片段时，任务必须获得 synchronized 指定的锁，否则就不能执行代码块，锁在同一时刻中，只能被一个任务拥有，这样就保证了临界区的代码，在同一时刻，只能由一个线程执行，遂保证了代码块的原子性。

共享资源不仅可以是个纯粹的内存对象，也可以是文件或者流等抽象模型。

Synchronized 有两种使用形式：

1、代码块：

```
method(){
    synchronized(lock){
        //code
    }
}
```

2、同步方法

```
synchronized method(){
    //code
}
```

使用 Synchronized 修饰代码时，我们可以自己指定锁，Synchronized 修饰方法时，使用的是对象拥有的单一锁。因此，当一个类中有多个 synchronized 修饰的方法时，一旦一个对象的一个同步方法被调用，那么在该方法被结束调用前，该对象的其他 synchronized 的调用也必须等待对象的单一锁。

一个任务可以多次获得对象上的锁，比如：一个同步方法调用类内的其他同步方法，被调用的同步方法也会获得锁，并且锁的计数器会增加 1，调用的同步方法返回，锁的计数器会减少 1，当对象的锁计数器为 0 时，该对象锁才会被完全释放。

7、显式加锁

synchronized 关键字提供了很好的语言内建支持，但是如果我们想要更灵活的线程同步控制，更强大的控制语义，我们应当显式的使用 Lock 对象，java.util.concurrent.lock 包提供了 Lock 接口以及各种常用实现，我们用 Lock 对象重写上例中的 current()和 cancel()方法，如下：

```
if (!isCanceled) {
                    System.out.println(thread()+" canceled checking !");
                }
        } finally {
                lock.unlock();
        }
        isCanceled = true;
        return current;
}
                System.out.println(thread()+" after -- , before yield() : "+current);
                Thread.yield();
                System.out.println(thread()+" after yield, before ++    : "+current);
                ++current;
                System.out.println(thread()+" after ++,  before return  : "+current);
                return current;
        } finally {
                lock.unlock();
        }
}

public int cancel(){
        try {
                lock.lock();
```

Lock 对象一般都是配合 try-finally 块进行 lock()和 unlock()的。这里我们使用的时 ReentrantLock，它是 java.util.concurrent.lock.Lock 的一个实现，它的语义与 synchronized 最为接近，因此如果要考虑替换 synchronized 关键字，我们首先考虑用它，

可以看到，我们使用 Lock 对象所写的线程同步代码，并没有使用 synchronized 所写的简单优雅，那为什么我们还要使用 Lock 对象呢？这是因为 Lock 对象更加灵活，语义更加丰富，比如我们可以非阻塞式的获取锁：lock.tryLock()也可以指定超时限制的获取锁 lock.tryLock(long time, TimeUnit unit)，也可以指定线程在被中断前，一直尝试获取锁 lock.lockInterruptibly()。

另外，使用 synchronized 关键字，我们不能灵活控制锁的释放，链式锁就难以书写，譬如：任务 1 获取获取 A 锁后，再获取 B 锁，获取 B 锁后，再获取 C 锁，然后释放 A 锁，获取 D 锁后再释放 B 锁。使用 Lock 对象，我们就可以实现链式锁。

并且，使用 synchronized 上锁的时候，如果锁已经被占用，那么 synchronized 块会一直等待锁，并且这种等待是不可被打断的。而 lock.interruptibly()则可以允许等待锁的时候被打断。

最后，Lock 对象还能获得锁的计数器，锁的等待队列，等待数估计等使用 synchronized 关键字无法获取到的信息。所以，针对一般情况，我们首选 synchronized 关键字进行线程同步控制，当遇到了更加灵活的控制需求和极端情况，我们可以考虑使用 Lock 对象来增强或者优化实现。

8、可见性与 volatile 关键字

如果有人要你用一个关键字来彰显你是比较了解 Java 的，那么 volatile 是再合适不过了。

要理解可见性，我们必须理解线程是如何操作变量的。线程在访问一个变量的时候，可能会在线程自己的寄存器中缓存一个变量的副本，在被修改后的某一时刻，再将修改刷新到主存去。因此，线程可能读取不到最新的变量值，因为其他线程对变量的修改可能还没被刷新到主存，或者本线程的变量副本还未更新。

那么这种情况下，我们称变量是不具有可见性的，而 volatile 修饰的变量，每次读取，都会直接从主存中读取，每次写入，都会直接写入到主存中去。我们称具有这种特性的变量，是有可见性的。也可以总结为：一个线程修改完共享变量后，其它线程能立刻看到修改。

volatile 关键字仅能保证可见性，不能保证原子性，所以不能代替锁以及同步控制。

volatile 特性（C 语言中也有这一特性）的支持一般是由硬件层面进行支持的，有总线锁以及缓存一致性协议这两种解决方案，出于多核 CPU 的性能考虑，现代处理器大多都使用缓存一致性协议。

比较常见的是 Intel 的 MESI 协议，它的原理是：当 CPU 写数据时，如果发现写的是共享变量，会发出消息通知其它 CPU 将该变量的缓存设置为失效状态，当其它 CPU 需要读取这个变量时，发现该变量的缓存已失效，然后就会直接从内存中读取最新值。

9、原子性与 Atomic 类

如果一种操作在执行完成前，肯定不会发生线程切换，那么我们就称这样的操作是原子性的。

除去 long 和 double 之外，其它基本类型的读取或者赋值操作，是原子性的，但是当你使用 volatile 来修饰基本类型时（包括 double 或者 long），对他们的读取或者赋值，都是原子操作。

请注意，++/- -这种操作在 Java 语言中并不是原子的。数值类的基本类型，进行数学运算后赋值也不是原子性的。必须强调的是，仅原子性也是不能代替锁和同步控制的。如果想对并发程序做出性能优化，请保证以下两点：

1、必须同时使用原子性和可见性。
2、你是一个专家级别的程序员，或者有个专家帮你调优。

否则，还是应该使用锁以及同步控制来保证并发程序的正确性。

Atomic 系列的类，主要是为了达成一个目标：使数值类的基本类型，进行数学运算后赋值，这两个连贯操作，具备原子性。

比如 i++这种不是原子性的操作，可以由以下代码代替：

```
AtomicInteger i = new AtomicInteger(5);
integer.getAndIncrement();//i++
integer.getAndDecrement();//i- -
integer.incrementAndGet();//++i
integer.decrementAndGet();//- -i
```

当然 Atomic 系列的类还提供了更丰富强大的语义，但是一般情况下，我们更应该使用锁以及同步控制，而不是依赖原子性，因此 Atomic 系列的类了解就好。

10、ThreadLocal

```
public class ThreadLocalTest {
    public static void main(String[] args) {
        ThreadLocalTest tlt = new ThreadLocalTest();
        Task task = tlt.new Task();
        new Thread(task).start();
        new Thread(task).start();
    }

    class Task implements Runnable{

        private volatile Integer sharedMember;

        @Override
        public void run() {
        }
    }
}
```

对于这样的代码，我们虽然启动了两个线程，但是我们只有一份数据，就是 task 对象，那么这时候成员 i 就是被共享的，两个线程都可以修改它，当我们需要变量被共享时，这样做没问题，但是，当我们需要一些成员变量在每个线程都有自己的副本，且每一个线程都只能修改和读取自己的副本时，我们可以使用 ThreadLocal 这个对象。

代码如下：

```
class Task implements Runnable{

    private volatile Integer sharedMember;
    private ThreadLocal<Integer> privateMember = new ThreadLocal<Integer>(){
        protected Integer initialValue() {return 0;}
    };

    @Override
    public void run() {

    }
}
```

这里的 privateMember 就是每个线程私有的变量，只能由自己修改，读取变量的值使用 threadLocal.get()，赋值使用 threadLocal.set()，我们可以像上面那样覆盖 initialValue()方法，来指定 threadLocal 所存储的对象的初始值。

修改 run 方法：

```
@Override
public void run() {
    for (int i = 0; i < 100000; i++) {
        sharedMember++;
        privateMember.set(privateMember.get()+1);
        if (sharedMember != privateMember.get()) {
            break;
        }
    }
    System.out.println("sharedMember : "+sharedMember+" privateMember : "+privateMember.get());
}
```

发现两个线程很快就会结束运行，这非常好理解，两个线程都在对 sharedMember 做自增操作，但是每个

线程只能修改自己的 privateMember，因此，这很快就会导致 sharedMember 和 privateMember 的数值不等，然后程序退出。

因此，如果我们想让某个变量在每个线程中都有自己的副本，每个线程都只能修改自己的副本，而不能修改其他线程的，那么我们应该使用 ThreadLocal，这就叫线程本地存储。

它的实现原理也很简单，Thread 类有一个成员叫做 threadLocals，它的类型是 ThreadLocal.ThreadLocalMap，对，你没看错，这家伙实际上是个 map，map 的 key 是 ThreadLocal 对象，value 就是 ThreadLocal 里面存储的值。所以使用 ThreadLocal 本地存储实际上就是在向 Thread 自己的一个 map 里 put 东西，所以当然不是共享变量了。

11、线程池与 ExecutorService

我们使用 new Thread(new Runnable()).start()这种方式，启动一个线程，还是太过原始了，这种方式有很多弊端，最严重的弊端是：当频繁创建线程的时候，服务器性能会严重下降，因为线程的创建和销毁对于系统来说是很大的开销，因此，我们应该尽可能的重用每一个线程，并且严格控制最大线程数量，来保证服务器资源不会被过度使用。

线程池这一技术就可以满足上述要求，池内的线程会被不断重用，并且我们可以指定池所能容纳的最大线程数，保证不过度创建线程。但是我们手动实现一个线程池还是非常有难度的，幸好从 Java5 开始，java.util.concurrent 包为我们提供了线程池方面的支持。我们可以通过如下方式创建一个线程池：

Executors.newCachedThreadPool();	创建一个缓存线程池，池中有可用线程时，复用线程，没有则创建一个，并添加至池内。终止并从缓存中移除那些已有 60 秒未被使用的线程，因此长时间保持空闲的线程池也不会占用任何资源。
Executors.newFixedThreadPool(int num);	和 newCachedThreadPool()类似，但是有两点不同： 1、这个线程池是有最大数量限制的，当线程已满并且都在被使用时，新到的任务只能等待 2、除非线程被显式关闭，否则线程就一直存在。
Executors.newSingleThreadExecutor();	只有一个线程的线程池，线程忙时，后到的任务在队列中等待被执行，因此可以保证顺序的执行任务。
Executors.newScheduledThreadPool(int num);	可以定期执行任务或者延迟执行任务的线程池。

再看看应当如何使用线程池来执行任务，常用代码是这样的：

```
import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorTest {
	public static void main(String[] args) {
		ExecutorService cachePool = Executors.newCachedThreadPool();
		cachePool.execute(new Runnable() {
			@Override
			public void run() {
				//business code
			}
		});

		if (true) {//condition for shutdown the pool
			cachePool.shutdown();
			List<Runnable> tashWaitingForRun = cachePool.shutdownNow();
		}
	}
}
```

shutdown()方法会允许已经提交的任务继续执行，但是不会再执行还未执行的任务了，但是 shutdownNow()方法还会试图停止正在执行的任务（包括但不限于调用线程的 interrupt()方法），但是不能保证能成功停止。并且 shutdownNow()方法还会返回一个未执行的任务列表。

12、线程协作与 wait(),notify()和 notifyAll()

在某些并发状况下，我们可能期待线程之间是以某种顺序或者约束执行的，比如：三个线程 A,B,C 轮流输出自己的名字 A,B,C，并且必须保证顺序不变，不能是乱序。这就规定了线程的执行顺序。再比如说著名的生产者消费者问题，是对生产线程和消费线程提出了约束：无产品时消费者等待，产品生产满后生产者等待，产品数既不为 0 也不满时，生产者和消费者均可工作，无顺序要求。

先看一个保证线程执行顺序的例子，我们假设一个老师正在给一群学生上课，每个学生都有自己的学号，大家需要按照学号顺序发言，直到老师说停，那么所有学生停止发言。代码如下：

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;

public class Talkers {

	private static int count = 0;
	private static final int peopleNum = 5;
	private static final ReentrantLock lock = new ReentrantLock();
	private static final List<ClassMate> classMates = new ArrayList<ClassMate>(peopleNum);

	static class ClassMate implements Runnable{

		private ClassMate next;
		public final String name;
```

```
        public ClassMate(String n){
            name = n;
        }

        public void setNext(ClassMate n) {
            next = n;
        }

        @Override
        public void run() {
            try {
                while(!Thread.interrupted()){
                    synchronized (this) {//举手，表示自己要发言
                        boolean isFurtherLock= lock.tryLock();//尝试获取老师的允许
                        if (isFurtherLock) {//如果老师允许我发言，那么其它同学就"直接"放下手
                            System.out.println(count++ + " "+Thread.currentThread().getName()+"
"+name);//发言
                            synchronized (next) {//发言完毕后选定下一个发言的同学。
                                next.notify();//告诉下一个发言的同学，接下来该他发言
                            }
                            lock.unlock();//告诉老师，自己发言完毕
                        }
                        wait();//放下手然后等待
                    }
                }
            } catch (InterruptedException e) {
                System.err.println(Thread.currentThread().getName()+" was interrupted !");
            }
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < peopleNum; i++) {
            ClassMate c = new ClassMate((char)(65+i)+"");
            if (i > 0)
                classMates.get(i-1).setNext(c);
            classMates.add(c);
        }
        classMates.get(classMates.size()-1).setNext(classMates.get(0));
        ExecutorService executor = Executors.newFixedThreadPool(peopleNum);
        for (ClassMate c : classMates) {
            executor.execute(c);
        }

        try {
            TimeUnit.MILLISECONDS.sleep(300);
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName()+" was interrupted !");
        }

        executor.shutdownNow();
    }
}
```

可以看到，ClassMate 类中的 run()方法明确的规定了每个同学发言的顺序，即便是很多人竞争，也得按照顺序来。lock 对象就像是一个老师，由他来决定在竞争中，谁来发言。

这段代码中我们使用了 ReentrantLock 进行尝试性的上锁，如果获取失败，那么还要放弃之前已经锁定的资源，这样的做法是为了防止死锁的发生。由于 synchronized 关键字提供的语义较为单一，因此尝试性上锁这样的操作，我们只能交由 Lock 对象来完成。

next.notify()唤醒所有正在等待（调用了 next 对象的 wait()方法，或者是 next 对象调用了自己的 wait()方法）next 这个对象的线程，继续执行代码。

wait()方法则阻塞了当前线程，直到有人打断这个阻塞或者调用当前对象的 notify()方法或者 notifyAll()方法，否则线程一直阻塞。

我们看到，wait()，notify()，notifyAll()方法虽然密切和线程相关联，但实际上，他们是 Object 类的方法，起初我们可能会感到不解，觉得这些方法应该声明于 Thread 类内才对。但是，仔细一想，便可以理解，我们要等待的是资源而不是线程，唤醒线程的，也应该是某个资源（在它自己可用时），而资源可能是任何的对象，因此 Java 语言的设计者便把这三个方法安置在了 Object 类内。所以，当我们再看到 xxx.wait()我们就可以理解，这是程序要求当前线程等待 xxx 这个资源可用，xxx 可用时，程序再继续执行。xxx.notify()是在通知正在等待 xxx 资源的线程，告诉它这个资源可用了，你可以继续执行了。

notify()会选择任意的一个等待线程，将其唤醒，而 notifyAll()则唤醒所有等待线程，当然，唤醒了所有的等待线程，它们又得重新竞争了。

为什么要使用 wait()这样的方法呢？理论上来说我们是可以使用 while 循环去不停地检查一个资源是否可用的，如果可用，则怎样怎样，但是这会很浪费 CPU，CPU 必须一直执行检查，而不能把计算资源转移到别的任务上面去。这被称为忙等待。

wait()方法就是用于避免忙等待的，操作系统和 JVM 将会使用中断机制来通知等待线程，而忙等待，我们也可以称之为轮询机制。

通过上面的例子，我们看到了如何利用 wait(),notify(),notifyAll()以及 synchronized 和 Lock 来保证线程的执行顺序，接下来，我们看看如何保证多线程环境下，线程的执行满足某个约束。这里我们用生产者和消费者来举例。

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

class Production {
        private int value;
        public void increment(){ ++value; }
        public void decrement(){ --value; }
        public int value(){return value;}
        @Override
        public String toString() {return value+"";}
}

public class ProducerAndConsumer {

        private static final Production product = new Production();
```

```
    public static void main(String[] args) {
        ProducerAndConsumer pac = new ProducerAndConsumer();
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(pac.new Producer());
        exec.execute(pac.new Consumer());

        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            System.out.println(thread()+" was interrupted !");
        }
        exec.shutdownNow();
    }

    private static String thread(){
        return Thread.currentThread().getName()+" : ";
    }

    class Producer implements Runnable{
        @Override
        public void run() {
            try {
                while (!Thread.interrupted()) {
                    synchronized (product) {
                        if (product.value() > 0) {
                            product.notifyAll();
                            product.wait();
                        }
                        product.increment();
                        System.out.println(thread()+"product count plus 1 becomes "+product);
                    }
                    TimeUnit.MILLISECONDS.sleep(200);
                }
            } catch (InterruptedException e) {
                System.err.println(thread()+" Producer was interrupted !");
            }
        }
    }

    class Consumer implements Runnable{
        @Override
        public void run() {
            try {
                while (!Thread.interrupted()) {
                    synchronized (product) {
                        if (product.value() < 1) {
                            product.notifyAll();
                            product.wait();
                        }
                        product.decrement();
                        System.out.println(thread()+"product count minus 1 becomes "+product);
                    }
                }
            } catch (InterruptedException e) {
                System.err.println(thread()+" Consumer was interrupted !");
            }
        }
    }
}
```

这里，生产者生产一个产品后，就不能再生产了，除非消费者把这个产品消费掉，当然没有产品的时候，消费者也不能消费了。当然，生产的过程总是比较艰辛，而消费又太过容易，为了模拟现实情况，我们让生产者每生产一个产品就休息一下，睡眠 200 毫秒，而消费者不休息，时时刻刻等待消费。

然而我们的程序做了恰当的同步控制，因此就算消费者再饥渴，它最多也只能 1 秒钟消费掉一个产品，因为生产者最多也就 1 秒钟生产出一个产品。其他不能消费的时刻，消费者就只好等待，等待着生产者通知他，告诉他又可以消费了。

注意，我们的 Production 实际上包装了一个 Integer，我们为什么不直接使用一个 Integer 作为产品呢？这是因为如果我们使用 Integer 作为产品，那么为了保证生产者和消费者对产品数量的修改不出现竞争条件，我们需要使用 synchronized 对产品上锁，并使用 wait()和 notify()做同步控制，但是，Integer 对象的++或者—操作会返回一个新的 Integer 对象，因此，这会导致 production.wait()和 production.notifyAll()这两句在执行时，production 不是同一个对象，那么结果是：线程在 A 对象上等待，但是却是 B 对象企图唤醒等待自己的线程，等待 A 对象的线程永远都不会被唤醒。因此，我们需要保证 wait()和 notify()或者 notifyAll()的调用者一定是同一个对象，这样的线程间协作才能正确运行，对于 synchronized 块也是，一定要保证同一个对象被用作同步。因此我们最好给锁加上 final 关键字，来保证正确性。

五、一个 HTTP 请求的前世今生

1、概述

Http 是应用层的网络协议（这是针对 TCP/IP 协议簇的 5 层模型），Http 是超文本传输协议的缩写：HyperText Transfer Protocol，而超文本传输，顾名思义，除去传输文本，该协议还支持传输文件，流媒体等。这也是我们为什么不仅可以在网页上浏览文字，还能查看图片，浏览视频的原因。

HTTP 最初设计是用来传输 HTML 页面的，HTML 是超文本标记语言的缩写：HyperText Markup Language，故名思议，该语言不仅能表示文本，还可以表示文件，流媒体其他对象，HTML 是网页的实际组成部分，浏览器会把接收到的 HTML 文本渲染成我们所看到的网页，因此网页实际上是由 HTML 来描述的。

2、请求过程

我们使用一个简单的例子来理解一次 HTTP 请求、响应过程。

当我们打开浏览器，在地址栏输入：http://baike.baidu.com/view/692.htm 后，按下回车键，我们就发送了一次 HTTP 请求，实际上，可能不止一次，这个后面再说。

这时候到底发生了什么呢？首先，浏览器查找缓存，看看这个页面是否已经被缓存下来了，如果缓存还没有失效，那么就把缓存下来的 HTML 文本重新解析并渲染，展示在屏幕上。

如果缓存已经失效或者没有缓存，那么浏览器就要发出一次 HTTP 请求了。

观察 url，我们看到/view 之前是百度百科的域名，我们向百度百科请求的资源是：/view/692.htm，就是服务器根路径下的 view 文件夹下的 692 这个 html 页面，所以我们这次的 HTTP 请求实际上是要求服务器把它根目录下的/view/692.htm 这个文件的文件内容传回给浏览器，浏览器收到后就会把这个文件的文件内容解析并渲染，然后显示在浏览器内，这样，我们就看到了我们所期待的网页了。

当然这时候我们只是知道，我们在向百度百科请求这样的一个页面，但是百度百科又是什么呢？我们总不

能说：“李彦宏，你去找下这个页面的内容，然后发回到我浏览器里”，是的，应该有服务器响应我们的请求才对，具体来说，是负责百度百科这个域名的服务器。

但是百度百科有那么多服务器，谁知道是哪一台该响应我们的请求呢？计算机网络中，IP 地址可以唯一的指定一个机器，看来我们需要完成域名到 IP 地址的转换，才能确切的把 HTTP 请求发给某个特定的服务器。

把域名转换为某个服务器的 IP，大致是这样一个流程：

1、浏览器首先会在自己的有效缓存中查找，看看目标地址有没有被解析过，如果有，就使用该 IP，域名解析停止。

2、如果浏览器没有缓存，或者缓存已经失效，那么查看本机的配置文件，Windows 下就是 C:\\Windows\System32\drivers\etc\hosts 文件，Linux 下就是/etc/hosts 文件。找到就使用该 IP，停止解析。

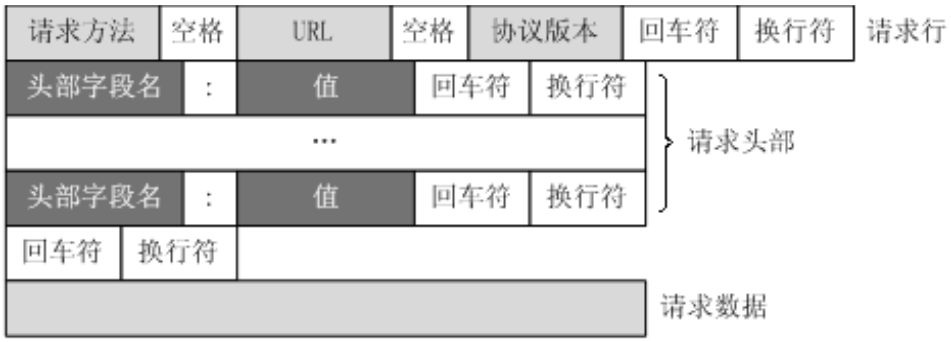
3、如果在本机配置文件中，没有找到相应的解析规则，那么就发送网络请求，把 URL 发送给本地域名解析服务器，由它们来解析域名，如果它们能成功解析，解析停止。

4、如果本地域名解析服务器也无法解析，那么解析就会交由更上一级的域名解析服务器，直至根域名解析服务器，到根域名解析服务器一定能解析域名，因为它确切的知道一个域名，可以由哪个服务器来解析。

全球共有 13 台根逻辑域名服务器。这 13 台逻辑根域名服务器中名字分别为“A”至“M”，真实的根服务器在 2014 年 1 月 25 日的数据为 386 台，分布于全球各大洲。在根域名服务器中虽然没有每个域名的具体信息，但储存了负责每个域（如 COM、NET、ORG 等）的解析的域名服务器的地址信息，如同通过北京电信你问不到广州市某单位的电话号码，但是北京电信可以告诉你去查 020114。

好了，现在我们知道了服务器的 IP，我们也知道了我们要请求的资源，那么只要按照 HTTP 协议的格式要求，把 HTTP 报文发送给目标机器的指定端口就行了，HTTP 的指定端口是 80。

他的请求报文格式如下（GET 方式）：




```
GET /view/692.htm HTTP1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:35.0) Gecko/20100101 Firefox/35.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-cn,zh;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Host: baike.baidu.com
Cookie: BAIDUID=06C7D286C60FB02E94085658F301A508:FG=1; BIDUP-
SID=06C7D286C60FB02E94085658F301A508;
Connection: keep-alive
```

GET 和 POST 时最常用的两种 HTTP 方法，GET 方法主要用户获取数据，而 POST 方法主要用于向服务器

发送数据。当然，GET 方法也是可以发送数据的，这需要把参数以？追加在 URL 后面，多个参数使用&分割。并且 GET 发送的数据大小不能超过 2K，并且是明文传输的；POST 方法发送的数据是存在于方法体内的，相比 GET 方法更安全一些，至少不会轻易被用户看到，而且 POST 发送的数据理论上来说，大小没有限制。

观察上面的 HTTP 请求报文，我们可以看到 User-Agent，这个参数指明了用户所使用的浏览器和操作系统环境。Accept 则说明了接受返回结果的 MIME 类型，比如 html，xml，xhtml。Accept-Language 指明了所能接受的返回结果的语言。Accept-Encoding 指明了接受的压缩数据格式是 gzip，接受的压缩算法是 deflate，Host 说明了本次请求的域名，Cookie 则指明了浏览器的一些缓存信息，Connection:keep-alive 则说明了如果上次和服务器建立的连接还没有超时，或者还没有被关闭，那么这次的 HTTP 请求就复用上次和服务器所建立的连接。

好了，现在我们知道 IP，报文也生成好了，我们只需要把这个报文发给服务器的指定端口就行了，在服务器的 80 端口上，会有程序一直监听着请求，时刻准备响应。

现在，对于发送请求来说，应用层的工作做完了，我们知道，就该由传输层封包处理了，再接下来由网络层处理，数据链路层处理，由物理层传输，直到所有数据包都到达服务器，又被层层还原成一个 HTTP 报文，然后这个报文被发送给了一直在 80 端口监听的服务器程序。

3、响应过程

服务器程序拿到 HTTP 请求报文后，开始解析，并响应，具体的响应过程就是寻找并返回报文指定的资源，这里就是/view/692.htm，如果你有 Unix 或者 Linux 使用经验，你会觉得/view/692.htm 是指服务器文件系统根路径下的 view 目录下的 692.htm 这个文件，但其实并不是这样的，服务器程序，如 Apache,Nginx,IIS 都会有自己的配置文件，其中会指明服务器程序的根目录/映射到操作系统中的某个目录，因此实际上我们找的是指定目录下的 view 目录下的 692.htm。

找到这个 692.htm 这个文件后，服务器程序会读取这个文件，将文件内容传回到客户端，当然，也是按照 HTTP 响应报文格式。格式如下：

```
HTTP/1.1 200 OK       //响应行
Date: Sat, 31 Dec 2005 23:59:59 GMT
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 122

< html > < head > < title > 网页标题 < /title > < /head > < body > 网页内容 < /body > < /html >
```

HTTP/1.1 是指 HTTP 协议版本，目前来说，HTTP 协议只有 1.0 和 1.1 两个版本，Date 指明了服务器响应时的时间，Content-Type 指明了响应体的 MIME-Type，这会指导浏览器，建议浏览器以何种方式解析并渲染响应体，charset 则说明了响应体的字符编码，避免乱码错误，Content-Length 说明了相应体的长度。

浏览器会将接收到的文本内容渲染在屏幕上，但这次的响应体中，很可能包括图片、音乐、视频或者对其它文本文件的引用，但是，在 HTML 中，用于描述对其它资源的引用的东西，依旧是一段文本，只是指明了这些资源文件的 URL 而已，URL 是统一资源定位符的缩写，它可以在互联网中唯一的标识一个资源，我们常说的网址，就是 URL，URL 不止可以标识 HTML 文件，还可以标识多种文件，比如 HTML 中所引用的各种媒体文件。

浏览器在解析 HTML 文本的同时，还会请求 HTML 中所引用的其他资源，比如，HTML 中使用<img>标签，引用了一张图片，那么浏览器会发出第二次 HTTP 请求，去把这个图片请求回来，类似的，浏览器会把 HTML 中所有引用的资源都请求回来，然后补充到已经渲染好的页面上，当网速较快时，我们可能以为网页是

被一次加载出来的，当网速较慢时，你就会明显的观测到浏览器多次请求的过程。所以通常来说，请求一个页面，会伴随着多次 HTTP 请求的发生。

除去常见的媒体文件，比如图片、音乐或者视频，HTML 种可能还会应用文本文件，这些文本文件也是通过独立的 HTTP 请求得到的，比较常见的是两种文本文件：Javascript 脚本文件和 Css 层叠样式表文件。Css 会告诉浏览器，对于当前的 HTML，应该如何渲染网页的各个部分，比如标题用什么字体，多大，什么颜色，网页背景是纯色还是图片。这可以使网页更加的漂亮。

而 Javascript 脚本文件，是可以运行在浏览器中的一段脚本程序，脚本程序，通常我们不会用它来做太大的事情。但是我们可以用它来实现一些简单的校验或者是动画特效，动画特效很好理解，自定义的动画总是得程序来实现，为什么简单的校验要使用脚本文件做呢？就是因为用脚本程序可以校验的东西，就不必发送给服务器了，这样可以避免不必要的请求，提升界面响应速度，优化用户体验。

以上我们过程，是我们请求一个静态 HTML 页面并得到响应的一个完整过程，但实际上，自从 Web2.0 起，人们就已经不满足只访问静态页面了，人们希望网页有更多的功能，可以有更复杂的交互，实质上，是希望服务器+浏览器，成为一种新的软件形式。

4、架构之争

说起新的软件形式，当然要先提一下老的软件形式，玩过《魔兽世界》或者是《梦幻西游》的人，都很清楚一个概念：客户端，想玩游戏，就必须下载客户端，并且保持最新版本的客户端。用户在客户端了浏览，操作，客户端会把用户的数据和操作发回给服务器，服务器在处理之后会把数据返回给客户端，客户端根据响应再做出渲染或者其他改变。我们称这样的软件形式叫做客户端—服务器模式，也叫 C/S 架构的软件，C 指 Client，S 指 Server。

C/S 架构有很大的优势，客户端和服务器高度契合，能带来更好的用户体验，客户端由于使用结构化的编程语言，并且直接与操作系统交互，因此也具备了更强大的行为能力和更多的权限。所以 C/S 架构的软件更加自由，行为更加丰富，用户体验更好。

但是 C/S 架构的软件也有很大的缺点，由于它更加自给自足一些，它自己要完成的事情也更多，比如，自己维护网络连接，自己定义渲染，自己开发界面基本组件等等，实际上，HTTP 协议出现后，网络连接这件事情在应用层以下，我们一般不用关心，甚至 HTTP 协议我们也不应关心，只要把要请求的数据准备好就行了；对于自己渲染界面，这也是个大块头的工作，但是 HTML 和 CSS 出现以后，一种标准化的渲染过程成为可能，除非对界面交互性要求非常高的程序，自己绘制才是比较好的策略；另外，对于自己开发界面组件这件事情，CSS 显然更胜一筹，它仅需要少量的定义就能基本满足自定义组件外观的要求，但是如果使用程序绘制，就会加剧程序的复杂度和代码量。所以我们看到，C/S 架构中，由于客户端需要自己做一些可以被标准化的事情，导致了工作量较大，这就是它的一个大弊端，但这还不是最致命的，客户端的致命性问题是跨平台的问题：就目前的情况来看，桌面操作系统主要包括：Windows/Mac OSX/Linux 三个平台，移动操作系统主要包括 iOS/Android/Windows phone 三个平台，所以，无论我们是想做桌面软件还是移动软件，要想完整的覆盖用户群体，一个软件就要写 3 遍，这样高的成本，只有一些大公司能接受。

于是新的软件架构出现了，新的软件架构叫做 B/S 架构，是浏览器—服务器架构的缩写，意思就是使用浏览器作为一种标准客户端，客户端呈现的内容由 HTML 文本控制，样式由 CSS 控制，行为由 JS 控制。浏览器解析了 HTML，自动发送了其它所需资源的 HTTP 请求，极大地简化了网络请求编写任务，并且浏览器会解析 CSS 进行渲染，由于浏览器提出了通用的布局模型，提供了公用的组件，因此 CSS 中少量的描述就可以做到以前需要大段客户端代码所做的事情。最为重要的是，只要浏览器厂商推出 3 个平台的浏览器，提供一致的行为，那么 B/S 架构的软件（HTML/CSS/JS）只要写一遍，就可以跨平台运行。这极大的节省了软件的开发成本，可以使开发人员把精力放在软件本身的业务逻辑上，更快更好地开发软件。这是 B/S 架构的诞生背景和优

势。

当然，事物都有两面性，B/S 架构的缺点也非常明显，最直接的就是由于 B/S 架构的软件基于浏览器更加规范，更加通用的各种模型（无论是网络请求的，还是界面绘制的），因此它们很难有更多的特性，更多的定制化，这直接导致了 B/S 架构的软件从用户体验来说不如 C/S 架构的软件。不过 HTML5 的出现努力正在缩小用户体验的差距，HTML5 提供了更全面更强大 HTML 语义支持，让 HTML 可以做更多的事，赋予 HTML 更大的权利，来充分的表达 B/S 架构软件的个性化需求。

5、新的请求

当 B/S 架构被认定为一种软件架构形式后，我们对网页的理解也发生了变化，网页也是软件的一种载体，一种不同于客户端的表现形式，既然网页也是一种软件，那么必不可少的，就是人与软件的交互，与网页的交互，而不仅仅是浏览网页的信息，换句话来说，信息必须是双向流动的。

要做到人与网页的交互，实际上是需要做到两点：1、用户指定服务器完成某个动作；2、用户可以向服务器发送数据。这两点实际上就是 Web2.0 变革的核心内容，我们指定服务器完成某个动作，这是通过特殊的 URL 来完成的，这样的 URL 最后的资源文件通常不是.html 或者.xml 这样的真实的文件后缀名，而是一个虚拟的后缀，比如说.do 或者其它你所喜好的字符串，甚至没有后缀。比如说这样的一个 URL：

```
http://sgt.package.qunar.com/suggest/getSuggestion?A=xxx&B=xxx。
```

可以明显看出 getSuggestion 是个动作，而且他没有文件扩展名，再加上后面的 A 和 B 参数，我们就能理解这个 HTTP 请求实际上是向服务器提交了两个参数然后要求服务器做出一些处理，然后把处理的结果返回。处理的结果可能是很多形式，比如一段符合 Json 格式的纯文本，或者一个新的 HTML 页面，或者一个二进制文件流等等。

我们重新关注一下 HTTP 请求的响应过程，看看服务器是怎么响应那些“动作”请求的，服务器在接到请求后，会对请求 URL 做出分析，按照一定规则把请求转发给监听在指定端口的处理程序，分发规则可以使列举，也可以是正则表达式，甚至可以使服务器脚本来指定分发规则。我们假设/suggest/getSuggestion 这个 URL 被匹配成功，并且处理程序是由 JavaEE 实现的，那么这个请求默认会打给监听在 8080 端口的 Tomcat/Jetty 容器，接下来将由 Java 程序来处理这个请求，对于/suggest/getSuggestion 这个请求，我们假设它会返回一段 Json。那么实际上响应请求的过程就是一个符合 JavaEE 标准的程序，接受了 url=/suggest/getSuggestion,参数为 A 和 B 的这样一个输入，返回一个 JSON 字符串作为输出。

六、Servlet 与 JSP

1、Servlet

接下来我们把注意力放在处理“动作”请求的 Java 程序上面，诚然，我们可以随意写一个 Java 程序，只要它监听某个特定的端口，然后能处理输入，返回输出就可以，但是这存在很多问题，比如说：怎样监听特定端口？怎样处理不同的请求类型？怎样解析参数？怎样做过滤拦截？怎样做逻辑后处理？怎样返回响应？每开发一次 Web 应用，这些问题就要自己重写一遍吗？实际上，这些问题是处理一次请求的通用流程，如果把这些通用流程定义成规范，那么我们在开发不同的应用的时候，只用关注核心业务逻辑就可以了，而不用重复的写这些流程。这套流程被称为 Servlet/JSP 规范。

所以说，Servlet/JSP 规范规定了一次请求的处理阶段和流程，这个规范把特定的接口暴漏给我们，让我们实现我们特定的业务逻辑，从而做到高层次的软件解耦，以减少我们的工作量并提升我们的开发质量，这就是

我们为什么会使用 Servlet/JSP 规范的原因。当然你会注意到，Servlet/JSP 毕竟只是个规范，到底必须有人写代码来实现这一规范，我们才能实际享用到那样的好处。那是谁用代码实现了这一规范呢？

Web 应用服务器，或者叫 Servlet 容器，顾名思义这个东西是部署 Web 应用的，或者说，Servlet 对象实际由这个容器来提供给你，供你使用。所以说，实际上你定义好的 Servlet 是由容器帮你初始化的，也是容器在接收到请求后选取特定的 Servlet 并逐步的调用你在 Servlet 中所写的方法，以完成响应的。著名的 Servlet 容器有 Tomcat/Jetty。

我们再具体的讨论一下 Servlet 规范中所规定的请求响应流程：Servlet 规范中规定 Web 应用的根目录下，必须有一个 WEB-INF 文件夹，而这个文件夹下，必须有一个 web.xml 文件，这个文件描述了 Web 应用的一些必备信息，其中就包含了 Servlet 名称到 URL 的映射，以及 Servlet 名称到 Servlet 类的全限定名的映射。有了这两个映射，容器就知道当一个请求到来时，我们需要哪个 Servlet 类去处理这次请求，因而请求就会得到正确的处理。

我们来关注一下这个 web.xml。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd">

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring.xml</param-value>
  </context-param>

  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>

  <filter>
    <filter-name>encodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
      <param-name>encoding</param-name>
      <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
      <param-name>forceEncoding</param-name>
      <param-value>true</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>encodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
```

首先，这个 XML 文档的根元素是<web-app>，其中一个属性 version="2.4"是说明这个 web.xml 遵守的是 Servlet2.4 规范。

<welcome-file-list>指出了一种意外情况：当请求的 URL 没有被映射到某个 Servlet 时，容器就会把<welcome-file-list>中所指定的文件返回给这次请求，如果 list 中的第一项找不到，那么就返回第二项，以此类推推。

<context-param>声明了应用范围内的上下文初始化参数。其中的<param-name><param-value>会组成键值对，这些键值对会作为容器启动参数，在容器初始化的时候会被用到。

```
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
      <servlet-name>springmvc</servlet-name>
      <url-pattern>*.do</url-pattern>
    </servlet-mapping>

    <error-page>
      <error-code>404</error-code>
      <location>/error.jsp</location>
    </error-page>

    <error-page>
      <exception-type>java.lang.Exception</exception-type>
      <location>/error.jsp</location>
    </error-page>
  </web-app>
```

<listener>描述了应用内的监听器，监听器可以对容器内一些对象的特定动作做出监听，一旦动作发生，监听器就会进行响应。在 web 应用中，Spring 框架一般是通过 Listener 方式启动的，Listener 启动顺序先于 Filter 和 Servlet，Listener 的支持是 Servlet2.3 规范的新特性。

<filter>顾名思义，过滤器，也就是对符合过滤要求的所有请求都进行过滤，这个很好理解，比如我们的处理程序都是默认针对 UTF-8 参数进行的。但是我们不能在每一处业务逻辑的开始，都写一遍字符集的检查程序，重复的事情我们不做第二次，因此，我们可以把检查参数字符编码并校准为 UTF-8 编码的这一逻辑，抽取出来，放入 Filter 中，这样，每次请求在业务逻辑处理前，参数都已经被转为 UTF-8 编码了，从而避免了因为参数编码所引起的程序错误。<filter>标签内把一个逻辑的<filter-name>和 Filter 的实现类<filter-class>对应起来。

<filter-mapping>描述了一个 filter 会针对哪些 url 做过滤处理，<filter-mapping>内会把一个<filter-name>和一个<url-pattern>对应起来。

<servlet>描述了真正的处理流程控制类：Servlet，没错，Servlet 不是让你写业务逻辑的地方，它作为一个被调用的入口，更应该书写的是流程控制逻辑或者是请求分发逻辑，这样才能更好的做到程序解耦，Servlet 的示范实现，请参见：org.springframework.web.servlet.DispatcherServlet。<servlet>标签内，也有一个名称和类的对应关系<servlet-name>,<servlet-class>。

<servlet-mapping>描述了一个 Servlet 需要对哪些 url 请求做出响应。

<error-page>指出了程序错误后，响应的页面，这个错误可以使为找到资源，未找到处理类，程序出现异常。

Servlet 容器会按照 web.xml 中的内容启动，并且初始化一系列的对象，等到请求到来的时候，容器会按照流程，逐步的调用各个对象，让它们完成它们的逻辑。因此，我们实际上是在基于一组特定的协议或者说是接口编程，我们负责具体的业务逻辑的实现，容器负责流程的控制和对我们实现的调用。这是面向接口编程所达到高级解耦效果，这让我们只专注于业务逻辑的编写，通用的流程有着标准化的实现，只实现一次，就可以应用于容纳和启动不同的工程，这实在是太赞了。

Servlet 针对一次请求会有标准的响应流程，具体是：按照 web.xml 中所声明的 Filter 顺序，生成一个 FilterChain，没错，这就像一条链子一样，容器只是调用第一个 Filter，而第一个 Filter 的实现中必须有：filterChain.doFilter(request,response,filterChain);这样的调用，才能继续调用第二个 Filter，以此类推，当 Filter 链中最后一个 Filter 被调用完之后，容器会调用相应的 Servlet，当 Servlet 处理完成后，程序的执行控制权会转回到倒数第一个 Filter 让他执行“后”拦截动作。然后是倒数第二个 Filter，以此类推，直到最后一个 Filter 的“后”处理逻辑也执行完，容器将结束对业务逻辑的调用，然后将这次请求的处理结果返回给调用者。

那么 Servlet 的请求的处理结果存放在哪里呢？我们在 Servlet 里面都见过这样的代码：

```
PrintWriter out = resp.getWriter() ;
out.println( “<html><head><title>servlet</title></head><body>Hello,Servlet!</body></html>” ) ;
out.close() ;
```

我们可以拿到容器传递给我们的 OutputStream 对象，然后把响应结果写进去，实际上，在容器结束对我们的 Filter 和 Servlet 的调用后，就会把 OutputStream 中的内容发回给请求的浏览器。通常，我们会把一个 HTML 页面写进输出流里，返回给浏览器由其渲染。

不得不提到的是，在程序里面用字符串写 HTML 这种事情实在是令人发指，各种转义字符需要注意，这极大地增加了程序出错的几率，并且，由于 Java 不支持多行字符串，我们从前端开发人员手里拿到的 HTML 文件，我们还得分割成多行然后处理一下，才能放进程序中，这实在是太麻烦了，有更好的办法吗？

2、JSP(Java Server Page)

当然有，无论是转义字符，还是字符串分行的问题，都是由于我们在程序中写字符串造成的，那我们为什么非要在程序中写字符串呢？是因为我们需要使用字符串拼接变量的方式，来输出动态内容，否则，我们只需要把待输出的内容放置在一个文件中，当有输出要求的时候，读文件然后写入输出流就可以了。

那么把要输出的内容存在文件中，我们就不能输出动态内容了吗？当然不是，我们读取到文件后，找准目标位置，把变量的值替换进去，这样既能输出动态内容，并且同时避免了转义字符和单行字符串的问题。我们把用于被替换关键值的静态文件称为模板文件，模板文件似乎非常棒，但是，这时的方案仍然不完美，关键就在于“找准位置，替换变量的值”这个事情，这是需要程序去做的事情，不同的模板，替换的位置和方法也不同，难道我们要为每一个 Servlet 都写一遍字符串替换程序吗？这太糟糕了，如果我们能总结出来一个字符串替换的标准，那么我们就可以只写一遍程序，而适用于各种各样的模板，只要各个模板遵守这个字符串替换标准即可。

现行的这一标准就是 JSP 规范，它规定当 JSP 渲染引擎读取模板文件(就是.jsp 文件)的时候，如果遇到<%= %>这样的形式，就会把标签内表达式的值替换到<%= %>在模板文件中的位置，当然 JSP 规范的语义更加强大，它甚至允许你在 JSP 中写代码，来完成一些逻辑。

那么如何在 Servlet 中使用 JSP 呢：request.getRequestDispatcher("/index.jsp").forward(req,resp);这句话就是你指定了 JSP 文件，接下来容器会找到/index.jsp 然后使用 JSP 引擎去渲染这个页面，但是，由于 JSP 中是可以写代码的，因此这里使用的“渲染”JSP 一词实际上并不准确，因为如果我们需要执行 JSP 里面的代码，这就必须编译 JSP 中的 Java 代码，JVM 才能将其执行。

而编译 JSP 中比较重要的一项工作就是为 JSP 里面写的代码找到一个执行环境，我们常用的 JSP 引擎：Jasper (2)，会生成一个 org.apache.jasper.runtime.HttpJspBase 的子类作为 JSP 中代码片段的执行环境，而 HttpJspBase 又继承自 HttpServlet，所以 JSP 实际上会被生成为一个 Servlet 然后被编译，加载，调用，执行。这个流程具体是：

1、找到 JSP 页面，

2、生成对应的 org.apache.jasper.runtime.HttpJspBase 子类的 Java 源码，

3、把这个 Servlet 的源码编译成.class 文件，

4、由 ClassLoader 把这个类加载进 JVM 来，

5、当别的 Servlet 中有请求被转发到这个 Servlet 时，容器会调用这个 Servlet 的 init 方法，而这个 init 方法中又调用这个 HttpJspBase 的 jspInit()和_jspInit()方法执行初始化。初始化完成后，容器会调用这个 Servlet 的 doService()方法时，doService()又调用了 HttpJspBase 的 jspService()（这个方法实际上已经被生成的子类复写了）执行 JSP 中的代码，并把 HTML 的内容写入到 HttpServletResponse 对象的 OutputStream 中去。

所以，从实质上来说，JSP 也是一个 Servlet，并且它是由 JSP 引擎生成 Servlet 源代码并编译的。JSP 的作用就是方便我们输出 HTML 代码，并且仍旧保持我们输出动态内容甚至是执行代码的能力。

来看一个具体的例子，看看一个 JSP 到底被编译成了什么样的 Servlet:

WEB-INF/web.xml

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>张力的网络应用</display-name>

  <filter>
    <filter-name>TestFilter</filter-name>
    <filter-class>personal.zl.webtest.orderTest.TestFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>TestFilter</filter-name>
    <url-pattern>/testJspOrder</url-pattern>
  </filter-mapping>

  <servlet>
    <servlet-name>TestJspPaintAndFilter</servlet-name>
    <servlet-class>personal.zl.webtest.orderTest.TestJspPaintAndFilter</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestJspPaintAndFilter</servlet-name>
    <url-pattern>/testJspOrder</url-pattern>
  </servlet-mapping>
</web-app>
```

TestFilter.java

```
package personal.zl.webtest.orderTest;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class TestJspPaintAndFilter extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOExcep-
tion {
        System.out.println("Begining servlet!");
        req.getRequestDispatcher("/index.jsp").forward(req,resp);
        System.out.println("Ending servlet!");
    }
}
```

TestJspPaintAndFilter

```
package personal.zl.webtest.orderTest;

import javax.servlet.*;
import java.io.IOException;

public class TestFilter implements Filter{

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {}

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException,
ServletException {
        System.out.println("Begining filter !");
        chain.doFilter(request, response);
        System.out.println("Ending filter !");
    }
```

index.jsp

```
<%@ page import="java.text.SimpleDateFormat" %>
<%@ page import="java.util.Date" %>
<html><head><title>Hello Servlet</title></head>
<body>
    <% System.out.println("Begining JSP !"); %>
    <%= new SimpleDateFormat("yyyy-MM-dd hh:mm:ss").format(new Date(System.currentTimeMillis())) %>
    <% System.out.println("Ending JSP !"); %>
</body>
</html>
```

/Users/zl/Library/Caches/IntelliJIdea14/tomcat/Unnamed_WebTest/work/Catalina/localhost/_/org/apache/jsp/index_jsp.java

```
/*
 * Generated by the Jasper component of Apache Tomcat
 * Version: Apache Tomcat/7.0.57
 * Generated at: 2015-05-13 07:47:38 UTC
 * Note: The last modified time of this file was set to
 *       the last modified time of the source file after
 *       generation to assist with modification tracking.
 */
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import java.text.SimpleDateFormat;
import java.util.Date;

public final class index_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

  private static final javax.servlet.jsp.JspFactory _jspxFactory =
          javax.servlet.jsp.JspFactory.getDefaultFactory();

  private static java.util.Map<java.lang.String,java.lang.Long> _jspx_dependants;
  private javax.el.ExpressionFactory _el_expressionfactory;
  private org.apache.tomcat.InstanceManager _jsp_instancemanager;
public java.util.Map<java.lang.String,java.lang.Long> getDependants() {
    return _jspx_dependants;
  }

  public void _jspInit() {
    _el_expressionfactory = _jspxFactory.getJspApplicationContext(getServletConfig().getServletContext()).getExpressionFactory();
    _jsp_instancemanager = org.apache.jasper.runtime.InstanceManagerFactory.getInstanceManager(getServletConfig());
  }

  public void _jspDestroy() {
  }

  public void _jspService(final javax.servlet.http.HttpServletRequest request, final javax.servlet.http.HttpServletResponse response)
        throws java.io.IOException, javax.servlet.ServletException {

    final javax.servlet.jsp.PageContext pageContext;
    javax.servlet.http.HttpSession session = null;
    final javax.servlet.ServletContext application;
    final javax.servlet.ServletConfig config;
    javax.servlet.jsp.JspWriter out = null;
    final java.lang.Object page = this;
    javax.servlet.jsp.JspWriter _jspx_out = null;
    javax.servlet.jsp.PageContext _jspx_page_context = null;
```

```
      try {
        response.setContentType("text/html");
        pageContext = _jspxFactory.getPageContext(this, request, response,
                    null, true, 8192, true);
        _jspx_page_context = pageContext;
        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();
        _jspx_out = out;

        out.write("\n");
        out.write("\n");
        out.write("<html>\n");
        out.write("<head>\n");
        out.write("    <title>Hello Servlet</title>\n");
        out.write("</head>\n");
        out.write("<body>\n");
        out.write("    ");

        System.out.println("Begining JSP !");

        out.write("\n");
        out.write("    ");
        out.print( new SimpleDateFormat("yyyy-MM-dd hh:mm:ss").format(new Date(System.currentTimeMillis())) );
        out.write("\n");
        out.write("    ");

        System.out.println("Ending JSP !");
        throw new NullPointerException();

        out.write("\n");
        out.write("</body>\n");
        out.write("</html>\n");
    } catch (java.lang.Throwable t) {
        if (!(t instanceof javax.servlet.jsp.SkipPageException)){
          out = _jspx_out;
          if (out != null && out.getBufferSize() != 0)
            try {
              if (response.isCommitted()) {
                out.flush();
              } else {
                out.clearBuffer();
              }
            } catch (java.io.IOException e) {}
          if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
          else throw new ServletException(t);
        }
      } finally {
        _jspxFactory.releasePageContext(_jspx_page_context);
      }
    }
}
```

当这个工程跑起来后，在浏览器中访问 localhost:8080/testJspOrder 时，我们会在控制台看到如下输出：

```
Begining filter !
Begining servlet!
Begining JSP !
Ending JSP !
Ending servlet!
Ending filter !
```

看到这个输出，你应该能对上面说到的 Filter/Servlet/JSP 的调用顺序有所理解了吧。

3、还差什么？

现在，我们已经专心于业务逻辑了，HTTP 请求的解析，网络连接的维护，请求流程的定义，页面的渲染，Web 容器都已经帮我们做好了，我们只需要为每一个 URL 开发对应的、符合 Servlet/JSP 规范的响应逻辑就好了。

这时候，我们又像使用 Java 语言编写传统软件一样，需要关注一下其他的问题：比如 Java 原生的 JDBC API 不是很好用，不能方便而又完善的进行数据库操作；比如各个 Java 对象之间的耦合严重，我们在使用一个对象前，需要初始化这个对象所依赖的对象，环环相扣，实际上要用一个，不得不初始化十个，或者我们需要不断地写 build 方法来践行构建者模式；再比如对请求的 URL，我们每次都要去 web.xml 里面配置，这和麻烦，就不能利用注解来简化这一过程吗？还有请求的参数能不能帮我提前解析好？渲染 JSP 的时候能不能不直接操作原生 API，有没有直接可以用于 Web 开发的 MVC 框架？

当我们基本解决网络相关的软件开发问题后，我们所面对的，就是软件本身的开发问题。

七、Spring

1、框架背景

上一节提到的各种问题，都是软件开发中会遇到的典型问题，并且，随着互联网的壮大，分布式应用程序变得越来越普及，更多的互联网企业放弃了野兽级服务器的策略，它们不会去购买大型机，小型机，甚至很少购买工作站，而是通过购买多台普通服务器，组成一个集群，来提高应用程序的吞吐量和响应速度。因此，开发分布式应用程序所遇到的问题也是我们需要关注的一个方面。

Sun 公司给出了上述问题的解决方案：Enterprise Java Beans（EJB），像 Servlet 和 JSP 一样，EJB 也是一套规范，并且它提供一套编程模型，使得基于该规范编写出的应用程序具有如下特性：面向对象，分布式，可扩展，支持数据持久化和事务处理。它还提供一定的安全性和负载均衡功能，这是一套庞大而复杂的规范。然而，根据我们的经验来说，想一次把所有事情做好，那就什么也做不好。

EJB 的编程模型限制了面向对象的威力，你需要基于 EJB 提供的编程模型（Session Bean，Entity Bean，MessageDriven Bean）来书写你的业务逻辑，而不能随心所欲的去写 Java 原生对象（Plain Old Java Object），也就是说 EJB 框架是有入侵性的，这提升了系统开发的复杂度，这是一个弊端。而 EJB 所提供的分布式计算，实际上是基于 C/S 架构的远程过程调用（Remote Procedure Call），就是客户端的代码可以调用服务器端某个对象的某个方法，就像调用本地对象的方法一样。然而，B/S 架构出现后，C/S 架构的应用越来越少，因此 EJB 的分布式特性实际上也很少被使用。

在 B/S 架构被大规模使用后，对应 EJB 的目标，出现了大量的框架，每个框架专注做好一件事情，Spring 框架来解决依赖注入（Dependency Injection）和面向切面编程（Aspect Oriented Programming）的问题；Hibernate 或者 MyBatis 来提升数据库操作的便捷性，Hibernate 甚至允许你不直接操作 SQL，而是操作对象就

完成了数据库操作。MyBatis 则是帮助你管理和复用 SQL 语句，并且提供一些实用的便捷特性；Struts2 或者 SpringMVC 则是 MVC 框架，由他们来控制请求的映射和处理流程，以及视图的匹配和其他页面方案的融合。

这些框架都是通过 jar 包形式提供的，那框架和库有什么区别呢？我们经常看到库也是由 jar 包形式提供的，比如操纵 excel 或者 xml 的库，也都是 jar 包。

区别在于：框架针对请求有自己的处理流程，并且把特定的接口暴漏给你，由你书写业务逻辑，它知道你的存在，然后它来调用你。而库更像是一个工具，它感受不到你写的代码，你的代码负责调用库，以达到你的目的。一言以蔽之：框架调用你，你调用库。

2、Spring 概述

Spring 主要解决两件事情：1、控制反转（Inversion of Control）；2、提供面向切面编程的支持。

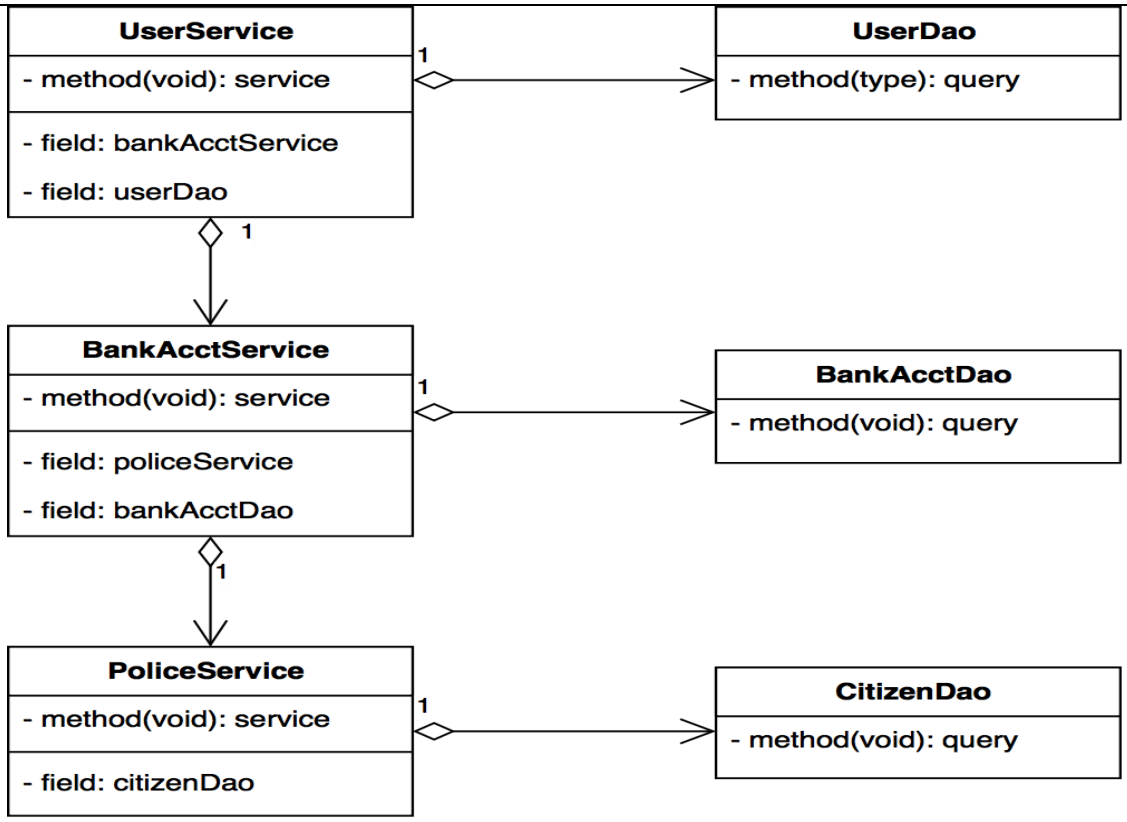
首先我们来说下什么是控制反转，为什么要控制反转。一般的系统中都会有很多对象， 并且它们之间是有依赖关系的，大型系统中，这种对象间的依赖关系会很复杂，这种复杂的依赖关系给使用和测试带来了很大的困难，因为我们必须手动创建出所有在依赖关系之内的对象，并且手动的把它们按照依赖关系组装后才能使用，因此，构建者模式会很常用，构建者模式提供一个标准的构建过程来解决这一问题。但是，这还是有些麻烦，我们要为不同的对象书写不同的构建方法。究其原因，问题出在我们手动拼装这些对象上，如果有一个容器，我们可以把构建对象的事情交给这个容器来做，我们只是获取并使用，那么我们将不再被这种依赖关系所困扰。

让容器来构建，拼装对象，以及对象所依赖的对象。这就称为控制反转，因为不是我们在创建”世界“，而是”世界“已经存在，我们只取所需，并加以利用，这种创造”世界“的控制权利，被交给了容器，这就是控制反转。这样的容器也被称为控制反转容器，简称 IoC 容器。

Spring 框架提供一个 IoC 容器，它使用 Java 的反射特性来完成对象的初始化和拼装。反射允许我们在运行时获取类型的信息，并使用类型的成员变量或者方法。Spring 框架是非侵入式的，它并不要求你的 Bean 实现什么接口或者继承自什么类，你完全可以用 Java 原生对象来完成你想要的一切，也就是说你使用 Spring 没有任何包袱，是轻装上阵。

3、控制反转（IoC）

为了更深刻的理解 IoC 容器的原理，我们来写一个小型的 IoC 容器。写 IoC 容器之前，我们先把我们的业务模拟好，业务逻辑的依赖关系如下：



代码如下：

```
package service;

import dao.UserDao;
import annotation.Component;
import annotation.Inject;

@Component
public class UserSerivce {

    @Inject
    private UserDao userDao;
    @Inject
    private BankAcctService bankAcctService;

    public void service(){
        System.out.println("I'm an UserService ( "+this.toString()+" ) , i'm working with userDao and bankAcctService !");
        userDao.query();
        bankAcctService.service();
    }

    //依赖注入加上注解方式，甚至是可以不用写 Getter/Setter 方法的
    public UserDao getUserDao() {return userDao;}
    public void setUserDao(UserDao userDao) {this.userDao = userDao;}
    public BankAcctService getBankAcctService() {return bankAcctService;}
    public void setBankAcctService(BankAcctService bankAcctService) {this.bankAcctService = bankAcctService;}
}

package service;
```

```
import dao.BankAcctDao;
import annotation.Component;
import annotation.Inject;

@Component
public class BankAcctService {

	@Inject
	private BankAcctDao bankAcctDao;
	@Inject
	private PoliceService policeService;

	public void service(){
		System.out.println("I'm an BankAcctService ( "+this.toString()+" ) , i'm working with bankAcctDao and
policeService !");
		bankAcctDao.query();
		policeService.service();
	}
	//依赖注入加上注解方式，甚至是可以不用写 Getter/Setter 方法的
	public BankAcctDao getBankAcctDao() {return bankAcctDao;}
	public void setBankAcctDao(BankAcctDao bad) {bankAcctDao = bad;}
	public PoliceService getPoliceService() {return policeService;}
	public void setPoliceService(PoliceService ps) {policeService = ps;}
}

package service;

import dao.CitizenDao;
import annotation.Component;
import annotation.Inject;

@Component
public class PoliceService {

	@Inject
	private CitizenDao citizenDao;

	public void service(){
		System.out.println("I'm an PoliceService ( "+this.toString()+" ), i'm working with citizenDao !");
		citizenDao.query();
	}

	//依赖注入加上注解方式，甚至是可以不用写 Getter/Setter 方法的
	public CitizenDao getCitizenDao() {return citizenDao;}
	public void setCitizenDao(CitizenDao cd) {citizenDao = cd;}
}

package dao;

import annotation.Component;

@Component
public class UserDao {
	public void query(){
		System.out.println("I'm an UserDao ( "+this.toString()+" ), i was querying the database !");
	}
}
```

```
package dao;

import annotation.Component;

@Component
public class CitizenDao {
	public void query(){
		System.out.println("I'm an CitizenDao ( "+this.toString()+" ), i was querying the database !");
	}
}

package dao;

import annotation.Component;

@Component
public class BankAcctDao {
	public void query(){
		System.out.println("I'm an BankAcctDao ( "+this.toString()+" ), i was querying the database !");
	}
}
```

好了，现在我们要使用 UserService 的对象了，我们该怎么做呢？

```
import service.BankAcctService;
import service.PoliceService;
import service.UserSerivce;
import dao.BankAcctDao;
import dao.CitizenDao;
import dao.UserDao;

public class Main {

	public static void main(String[] args) {
		System.out.println("\n 传统方式\n");
		//传统方式
		UserSerivce userSerivce = new UserSerivce();
		BankAcctService bankAcctService = new BankAcctService();
		PoliceService policeService = new PoliceService();

		UserDao userDao = new UserDao();
		BankAcctDao bankAcctDao = new BankAcctDao();
		CitizenDao citizenDao = new CitizenDao();

		policeService.setCitizenDao(citizenDao);

		bankAcctService.setBankAcctDao(bankAcctDao);
		bankAcctService.setPoliceService(policeService);

		userSerivce.setBankAcctService(bankAcctService);
		userSerivce.setUserDao(userDao);

		userSerivce.service();

		System.out.println("\n 依赖注入方式\n");
		//依赖注入方式
		Container.getBean(UserSerivce.class).service();
		System.out.println("                    ");
```

```
                Container.getBean(BankAcctService.class).service();
        }
}
```

运行一下 Main.java 我们会看到以下结果：

我们就为了写一句 userService.service()竟然不得不多写 11 行代码来组装一个可用的 UserService，要是算

```
传统方式

I'm an UserService ( service.UserSerivce@7852e922 ) , i'm working with userDao and bankAcctService !
I'm an UserDao ( dao.UserDao@4e25154f ), i was querying the database !
I'm an BankAcctService ( service.BankAcctService@70dea4e ) , i'm working with bankAcctDao and policeServ-
ice !
I'm an BankAcctDao ( dao.BankAcctDao@5c647e05 ), i was querying the database !
I'm an PoliceService ( service.PoliceService@33909752 ), i'm working with citizenDao !
I'm an CitizenDao ( dao.CitizenDao@55f96302 ), i was querying the database !

依赖注入方式

I'm an UserService ( service.UserSerivce@61bbe9ba ) , i'm working with userDao and bankAcctService !
I'm an UserDao ( dao.UserDao@610455d6 ), i was querying the database !
I'm an BankAcctService ( service.BankAcctService@511d50c0 ) , i'm working with bankAcctDao and policeServ-
ice !
I'm an BankAcctDao ( dao.BankAcctDao@60e53b93 ), i was querying the database !
I'm an PoliceService ( service.PoliceService@5e2de80c ), i'm working with citizenDao !
I'm an CitizenDao ( dao.CitizenDao@1d44bcfa ), i was querying the database !

I'm an BankAcctService ( service.BankAcctService@511d50c0 ) , i'm working with bankAcctDao and policeServ-
ice !
I'm an BankAcctDao ( dao.BankAcctDao@60e53b93 ), i was querying the database !
I'm an PoliceService ( service.PoliceService@5e2de80c ), i'm working with citizenDao !
I'm an CitizenDao ( dao.CitizenDao@1d44bcfa ), i was querying the database !
```

上每个 Service 和 Dao 自身为了支持组装的的 Getter 和 Setter 方法，那就多写了超过 30 行以上的代码，真是太麻烦了，但这已经是非常理想的状况了，因为这里所展示的业务逻辑太简单了，在真正的业务场景下，你可能为了拼装一个自己要用的对象，而不得不多写 100 行以上的代码，并且关心一些你本不应该关注的依赖逻辑。于是建造者模式出现了，遵循建造者模式的代码，会在每个 Service 内写一个 buildXXXService()方法，用来组装一个可用的对象，返回给外界使用，以减少代码冗余和调用者的负担。

虽然要遵循建造者模式已经令人不悦，但是如果前人栽树，后人乘凉，那么这样的糟糕体验也许是可以容忍的，但是，每次我们新写一个 Service，就不得不写一个新的 build 方法，甚至还要再写单例方法，这就不能忍了，重复的事情不是应该机器来做吗？做重复的事情，我们还算是一个好的程序员吗？

显然，一些有追求的程序员早就意识到了这样的问题，并着手解决了它，解决方案就是把依赖注入（实际上也就是拼装对象）并保证单例这样的重复性工作交给容器去做，我们只需要从容器中获取就行了，从代码上看，如果我们拥有了控制反转的容器，那我们调用 UserService 对象的代码就成了这样：
Container.getBean(UserSerivce.class).service();

一行还是一行，我们没有迫不得已去做些其他的事情：没有自己拼装，没有写 Getter/Setter 方法，没有 build 方法，没有写单例方法。这是多么的优雅！让我们来看看这优雅的背后是怎样的智慧：

```
package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
```

```
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Component {}

package annotation;

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

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Inject {}


import java.lang.annotation.Annotation;
import java.lang.reflect.AccessibleObject;
import java.lang.reflect.Field;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import annotation.Component;
import annotation.Inject;

public class Container{

	private static final Map<String,Object> objectsPool = new HashMap<String, Object>();

	@SuppressWarnings("unchecked")
	public static <T> T getBean(Class<T> clazz) {

		T targetBean = null;

		Annotation annotation = clazz.getDeclaredAnnotation(Component.class);

		//如果该类不是一个@Component 注解标注的类，那么就不注入。
		if (annotation == null) {
			return targetBean;
		}

		String beanName = clazz.getName();//获取类的全限定名

		targetBean = (T)objectsPool.get(beanName);//从缓存池中获取

		if (targetBean != null) {//如果获取到了，直接返回
			return targetBean;
		}

		//否则使用构造方法创建一个
		try {
			targetBean = (T)clazz.newInstance();
```

```
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }

        Field[] members = clazz.getDeclaredFields();
        AccessibleObject.setAccessible(members, true);
        List<Field> fields = filterFields(members);

        try {
            String name = null;
            Object o = null;
            for (Field field : fields) {
                name = field.getType().getName();
                o = objectsPool.get(name);
                if (o == null) {
                    o = getBean(field.getType());
                    objectsPool.put(name, o);
                }
                field.set(targetBean, o);
            }

        } catch (IllegalArgumentException | IllegalAccessException e) {
            e.printStackTrace();
        }

        objectsPool.put(beanName, targetBean);

        return targetBean;
    }
    //把非基本类型和未标记@Inject 注解的字段，都去掉，不做注入处理
    private static List<Field> filterFields(Field[] fs){

        List<Field> fields = new ArrayList<Field>(fs.length);

        for (Field field : fs) {

            if (isPrimitive(field)) {//如果是基本类型，直接跳过，不做注入处理
                continue;
            }

            Annotation[] as = field.getAnnotations();//拿到标记在该字段上面的所有注解
            boolean isContain = false;
            for (Annotation annotation : as) {
                if (annotation.annotationType().equals(Inject.class)) {//如果其中有@Inject 注解
                    isContain = true;
                    break;
                }
            }

            if (isContain) {
                fields.add(field);
            }
        }
        return fields;
    }
```

```
    private static boolean isPrimitive(Field type){
        switch (type.getType().getName()) {
            case "int":
            case "char":
            case "byte":
            case "boolean":
            case "short":
            case "long":
            case "float":
            case "double":
            case "java.lang.String":
                return true;
            default:
                return false;
        }
    }
```

容器有个对象池来保证对象的单例，然后使用递归策略来初始化被依赖的对象，通过注解标识来判断是否注入被依赖的对象，这些功能都使用到了 Java 的反射特性。像这样的 Container 我们称之为 IoC 容器，它主要是提供依赖注入（DI）的功能，控制反转或者说依赖注入，就是 Spring 的两大核心功能之一。

4、面向切面编程（AOP）

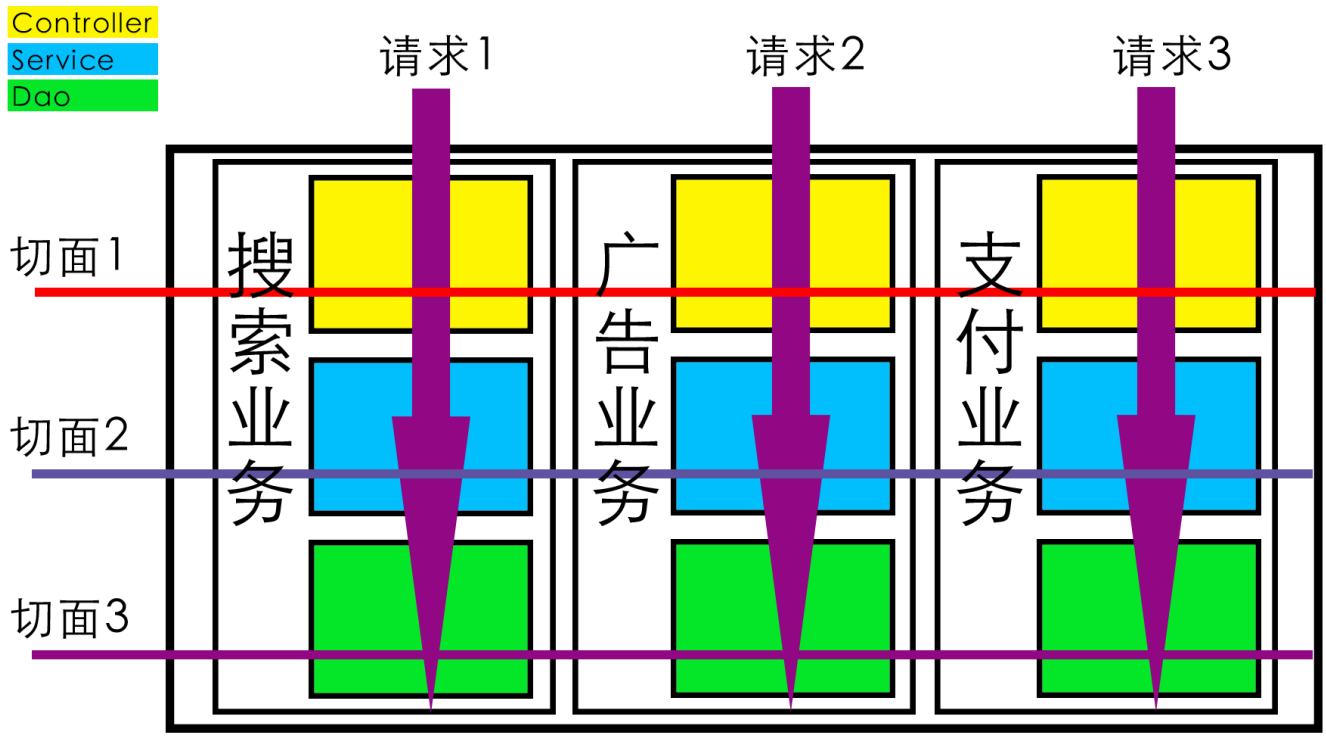
接下来我们来探讨 AOP，即面向切面编程，为了理解面向切面编程，我们先来描述几个需求：1、一个完备的系统是需要日志的，以便定位问题；2、需要性能监控，性能监控有时需要打印出方法的执行时间；3、一些涉及数据库的方法，需要事务来保证操作的原子性和一致性。

针对这三个需求，我们能怎么办呢？朴素的代码是这样的：

```
public void doService(){
    long cost = System.currentTimeMillis();
    System.out.println("[ "+Thread.currentThread().toString()+" ] "+System.currentTimeMillis()+" doService
begin.");
    Transaction transaction = EnhancerContainer.getBean(Main.Transaction.class);
    transaction.begin();
    try{
        //business code
        transaction.commit();
    }catch (Throwable e){
        transaction.rollback();
    }
    System.out.println("[ "+Thread.currentThread().toString()+" ] "+System.currentTimeMillis()+" doService
end.");
    cost = System.currentTimeMillis() - cost;
    System.out.println("Method doService cost "+cost+" millseconds");
}
```

但是看到这样的代码，你立刻就会觉得非常恶心，因为这样的做法并不能避免重复的书写样板式的代码，每一个方法里面都得写这些恶心的东西，我们只想关心我们的业务啊，为什么被迫去写那些样板代码？

哪里有压迫，哪里就有反抗，那些有追求的程序员天才们，又一次为我们解决了问题，这次，他们选择了不同的角度观察问题，不再以请求的处理流程去思考，而是把通用流程的某个阶段看做一个方面，一次性的为这些切面增强功能，就像下图所示：



那么如何做到不入侵业务代码，就能为切面增强功能呢？没错，代理模式：

```
public class Business {
        public void doService(){};
}

class Proxy{

        private Business business;

        public void doService(){
                before();
                business.doService();
                after();
        }

        public void before(){};
        public void after(){};
}
```

程序员编写 Business，只要框架能生成对应的 Proxy，我们就能做到非入侵式的增强实现：当我们像容器索取 Business 的实例时，其实容器给我们的是一个对应 Business 的 Proxy 实例，由于我们只关心和调用自己写的业务逻辑，也就是 doService 方法，我们是察觉不到 Proxy 所做的增强实现的。但是实际执行的时候，一旦我们调用 proxy 的 doService 方法，增强的代码也就被执行了。

大致的增强原理，我们已经说清楚了，那谁来具体完成生成代理的工作呢？目前主要有两种形式的动态代理方案（依据已有的代码，生成而不是手写代理，就叫动态代理），一种是 JDK 动态代理，这是 JDK 本身提供的代理生成功能，它会依据原对象的类型信息，来直接生成一份新的 class 文件，然后再将这个新代理类加载到内存中，然后生成一个实例，再返回给你。但是现在我们不经常使用 JDK 动态代理，这是因为它有一个限制，它生成的代理类，只能代理实现接口所产生的方法，而不能对非接口方法做代理。这也就是说，我们要想一个方法能被 JDK 代理，必须把这个方法放在某个接口中声明。

这样的限制会无端的增加冗余代码，因此出现了另外的动态代理方案，譬如最有名的 CGLib，它是一个库，提供 API 向调用者返回各种形式的代理。而它的底层则是 asm.jar，这是一个字节码操纵库，精简，极小，要想直接使用它需要丰富的 class 文件和 jvm 知识。CGLib 生成的代理，会按照你的要求代理所有的方法，无论这个方法是不是声明于一个接口。因此，Spring 使用 CGLib 来生成 POJO 的代理类。

好了，生成代理类的问题解决了，那么 before 和 after 方法内的实现代码，应该怎样书写呢？这些代码又应该存在于何处呢？我们需要为每个生成的代码都书写一次 before 和 after 方法吗？如果不是每次都写，那么该怎么精准控制增强语义呢？

来直接看看原理代码吧，相比于原来的 IoC 容器，支持 AOP 的容器增加了以下代码：

```
package core;

import java.lang.annotation.Annotation;
import java.lang.reflect.AccessibleObject;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;
import annotation.Component;
import annotation.Inject;

public class EnhancerContainer{

	private static final Map<Class< ? extends Annotation>, MethodEnhancerLogic> enhancers = new
HashMap<Class<? extends Annotation>, MethodEnhancerLogic>();

	@SuppressWarnings("unchecked")
	public static <T> T getBean(Class<T> clazz) {
//		try {
//			targetBean = (T)clazz.newInstance();
//		} catch (InstantiationException e) {
//			e.printStackTrace();
//		} catch (IllegalAccessException e) {
//			e.printStackTrace();
//		}
		targetBean = enhance(clazz);
	}

	private static <T> T enhance(Class<T> type){
		Enhancer enhancer = new Enhancer();

		enhancer.setSuperclass(type);
		enhancer.setCallback(new MethodEnhancer());

		@SuppressWarnings("unchecked")
		T proxy = (T) enhancer.create();

		return proxy;
	}
```

```
	private static class MethodEnhancer implements MethodInterceptor{
		@Override
		public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws
Throwable {

			Annotation[] as = method.getDeclaredAnnotations();

			List<MethodEnhancerLogic> afters = new ArrayList<MethodEnhancerLogic>(as.length);

			for (Annotation annotation : as) {
				MethodEnhancerLogic logic = enhancers.get(annotation.annotationType());
				if (logic != null) {
					afters.add(logic);
					logic.before(obj, method, args);
				}
			}

			Object result = null;
			Throwable throwable = null;
			try {
				result = proxy.invokeSuper(obj, args);
			} catch (Throwable e) {
				throwable = e;
			}

			int length = afters.size();
			for (int i = length - 1 ; i >= 0 ; i--) {
				afters.get(i).after(obj, method, args,throwable);
			}

			if (throwable != null) {
				throw throwable;
			}

			return result;
		}
	}

	public static void registerEnhancerLogic(Class<? extends Annotation> annotation, MethodEnhancerLogic
logic){
		enhancers.put(annotation, logic);
	}

	public static void removeEnhancerLogic(Class<? extends Annotation> annotation){
		enhancers.remove(annotation);
	}
}

package core;
import java.lang.reflect.Method;

public interface MethodEnhancerLogic {
	public void before(Object obj, Method method, Object[] args);
	public void after(Object obj, Method method, Object[] args,Throwable e);
}

package annotation;
```

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

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Log {}

package annotation.enhancer;

import java.lang.reflect.Method;
import core.MethodEnhancerLogic;

public class LogEnhancer implements MethodEnhancerLogic{
	@Override
	public void before(Object obj, Method method, Object[] args) {
		System.out.println("==  ["+method.getName() + "] log begin ! ");
	}
	@Override
	public void after(Object obj, Method method, Object[] args,Throwable e) {
		if (e == null) {
			System.out.println("==  ["+method.getName() + "] log end ! ");
		}else {
			System.out.println("==  ["+method.getName() + "] occur an error : "+e.getMessage());
		}
	}
}

package annotation;

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

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Performance {}

package annotation.enhancer;

import java.lang.reflect.Method;
import core.MethodEnhancerLogic;

public class PerformanceEnhancer implements MethodEnhancerLogic{

	private long cost;
	@Override
	public void before(Object obj, Method method, Object[] args) {
		cost = System.currentTimeMillis();
	}
	@Override
	public void after(Object obj, Method method, Object[] args,Throwable e) {
		cost = System.currentTimeMillis() - cost;
		System.out.println("==  The execution of method : ["+method.getName() + "] takes "+cost+"
millseconds .");
	}
}
```

```
package annotation;

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

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Transaction {}

package annotation.enhancer;

import java.lang.reflect.Method;
import core.MethodEnhancerLogic;

public class TransactionEnhancer implements MethodEnhancerLogic{

	@Override
	public void before(Object obj, Method method, Object[] args) {
		System.out.println("==  ["+method.getName()+"]'s transaction begin !");
	}
	@Override
	public void after(Object obj, Method method, Object[] args,Throwable e) {
		if (e == null) {
			System.out.println("==  ["+method.getName()+"]'s transaction has been committed !");
		}else {
			System.out.println("==  ["+method.getName()+"] occur an error , transaction has been rolled
back ! ");
		}
	}
}

package dao;

import annotation.Component;
import annotation.Log;
import annotation.Performance;
import annotation.Transaction;

@Component
public class CitizenDao {

	@Performance
	@Log
	@Transaction
	public void query(){
		System.out.println("I'm an CitizenDao ( "+this.toString()+" ), i was querying the database !");
//		throw new NullPointerException("故意制造的一个空指针异常");
	}
}

import annotation.Log;
import annotation.Performance;
import annotation.Transaction;
import annotation.enhancer.LogEnhancer;
import annotation.enhancer.PerformanceEnhancer;
import annotation.enhancer.TransactionEnhancer;
```

```
import core.Container;
import core.EnhancerContainer;
import service.UserSerivce;

public class Main {
	public static void main(String[] args) {
		try {
			System.out.println("普通 IoC 容器\n")
			Container.getBean(UserSerivce.class).service();

			System.out.println("\n 支持 AOP 的 IoC 容器\n");
			EnhancerContainer.registerEnhancerLogic(Log.class, new LogEnhancer());
			EnhancerContainer.registerEnhancerLogic(Transaction.class, new TransactionEnhancer());
			EnhancerContainer.registerEnhancerLogic(Performance.class, new PerformanceEnhancer());

			EnhancerContainer.getBean(UserSerivce.class).service();

		} catch (Exception e) {
		}
	}
}
```

这里，我们重新实现了容器 Container，并且添加了 3 个注解@Log，@Performance，@Transaction，用来标识被代理的方法的增强语义。然后新增了三个注解处理逻辑，LogEnhancer，PerformanceEnhancer，TransactionEnhancer 他们来定义当注解出现时的增强逻辑，它们都实现于一个新增的接口：MethodEnhancerLogic；而 CitizenDao 被打上了三个新的注解，来测试 AOP 的威力。

输出如下：

```
普通 IoC 容器

I'm an UserService ( service.UserSerivce@4b67cf4d ) , i'm working with userDao and bankAcctService !
I'm an UserDao ( dao.UserDao@7ea987ac ), i was querying the database !
I'm an BankAcctService ( service.BankAcctService@12a3a380 ) , i'm working with bankAcctDao and policeServ-
ice !
I'm an BankAcctDao ( dao.BankAcctDao@29453f44 ), i was querying the database !
I'm an PoliceService ( service.PoliceService@5cad8086 ), i'm working with citizenDao !
I'm an CitizenDao ( dao.CitizenDao@6e0be858 ), i was querying the database !

支持 AOP 的 IoC 容器

I'm an UserService ( service.UserSerivce$$EnhancerByCGLIB$$57a33519@4629104a ) , i'm working with
userDao and bankAcctService !
I'm an UserDao ( dao.UserDao$$EnhancerByCGLIB$$c0428b59@27f8302d ), i was querying the database !
I'm an BankAcctService ( service.BankAcctService$$EnhancerByCGLIB$$2c58b90f@4d76f3f8 ) , i'm working with
bankAcctDao and policeService !
I'm an BankAcctDao ( dao.BankAcctDao$$EnhancerByCGLIB$$4a80694d@4534b60d ), i was querying the data-
base !
I'm an PoliceService ( service.PoliceService$$EnhancerByCGLIB$$5055a88a@3fa77460 ), i'm working with citi-
zenDao !
==  [query] log begin !
==  [query]'s transaction begin !
I'm an CitizenDao ( dao.CitizenDao$$EnhancerByCGLIB$$1a16b782@7ab2bfe1 ), i was querying the database !
==  [query]'s transaction has been committed !
==  [query] log end !
==  The execution of method : [query] takes 5 millseconds .
```

如果我们打开 CitizenDao 的抛出异常的代码，我们还可以看到容器帮我们做的自动回滚，当然，真实的 Spring 只是提供一种切入方式，而真正与数据库回滚相关的代码，还是由数据库框架如 hibernate 或者 MyBatis 这样的框架实现的。

看到了这样神奇的效果，让我们聚焦新增的 enhance()方法以及新增的内部类 MethodEnhancer，他们实际上都使用了 CGLib 这个库提供的 API，来生成我们想要的代理（enhancer.create();），并且在被代理的对象的方法执行前提供一个切入点，以执行我们的代理逻辑，这个切入点就是由 MethodInterceptor 这个接口描述的，在 intercept()方法中执行 proxy.invoke 方法，就可以真正执行被代理的方法了，intercept()方法中我们根据容器启动时注册的注解处理器，处理了相应的注解，完成了相应的功能的切入，这就是 Spring-AOP 的基本原理了。

到目前为止，我已经通过原理性的代码阐述了 Spring 的两个核心功能：IoC 和 AOP，当然 Spring 的实际代码远比这复杂的多，因为它还要考虑但不限于以下问题：

1、线程安全
2、对集合类型注入的支持
3、泛型支持
4、非单例模式支持
5、多种注入策略的支持（按类型注入，按名称注入）
6、对其它框架如 Spring-MVC,Struts,Hibernate,Mybatis 等的支持。
7、对 J2EE 环境的支持
8、扩展点预留

所以，要想进一步了解 Spring，还请仔细研究 Spring 的代码。

八、其他框架简述

1、Hibernate

Hibernate 是一个开源对象关系映射框架（Object Relation Mapping）ORM 框架，它对 JDBC 做了封装，提供面向对象的方法来操作数据库，即调用对象方法来完成增删改查等操作。它会将数据对象本身映射到数据库中的表上，表字段对应对象的属性。Hibernate 框架支持两种映射方式：xml 和注解。

Hibernate 的思想非常清晰易懂，一言以蔽之，就是：把数据库中的表映射到 Java 原生对象上，然后提供一个具有“增删改查”方法的 Session 对象，供你调用。使用过程中，我们只需要调用这些“增删改查”的方法即可，而不用书写过程式的 SQL 语句。

Hibernate 支持大多数数据库，对不同数据库的特性也做了封装和屏蔽，暴露出通用的接口。了解 Hibernate，除了理解其代理和面向对象的思想，还必须要知道其 6 个核心类：

1、Session，它代表着程序和数据库的一次交互，其生命周期在一次逻辑事务之内，可以用来完成增删改查操作，但它并不是线程安全的。save(),persist()方法用于保存对象，delete()方法用于删除对象，update(),merge()方法用于修改对象，load(),get()方法用于查询对象。

2、SessionFactory，Session 工厂，每个线程或者事务应当从 SessionFactory 中获取 Session。

3、Transaction，它是一个接口，可能会有不同的具体实现，比如 JDBC 的事务，JTA(Java Transaction API)的事务，这样的实现是为了保证代码的可移植性。你并不一定要使用它，默认情况下，Hibernate 也会有底层

的事务处理。

4、Query 接口提供语言化的查询支持，你可以使用 HQL 或者是 SQL 进行查询。

5、Criteria 接口是对象化的查询接口，你可以向其中添加你想要的各种查询条件，然后再把 Criteria 对象传递给 Session 完成一次查询

6、Configuration，配置类，configuration.xml 的对象化表示。

使用 Hibernate，你必须非常清楚你操作的对象之间是什么关系，是一对一，一对多，多对一，还是多对多，因为这会涉及到级联加载的问题，A 对象持有一个 B，每个 B 对象持有多个 C 对象，那么加载 A 对象的时候，加载单一的 C 对象，还是加载一个 C 对象的集合？另外，一定要加载 C 对象吗？

答案是不一定，是否加载可以由你控制，你可以选择即时加载或者是懒加载（用到时再加载）。如果不选择懒加载，很可能在你做一次查询的时候，级联的把整个数据库都加载了进来。这一问题在一对多，多对一，多对多的关系上尤为严重。使用懒加载，只有在你真正使用对象的成员时，那个成员对象才会被加载进来，这一过程对于程序员是透明的。使用懒加载可以大幅提升性能。

Hibernate 有两级缓存，一级缓存也叫 session 缓存，这级缓存主要是为了减少数据库的连接次数，提高效率，一次 session 结束后，缓存也会失效。二级缓存的存活时间更长一些，二级缓存由 SessionFactory 支持，更新删除或者修改数据库的同时，也会更新删除或者修改二级缓存，以保证一致，二级缓存的目的同样是为了提升性能。但是二级缓存仅仅以对象的 ID 作为缓存的 key，当程序使用条件查询时，二级缓存是不起作用的，对于条件查询，可以使用缓存插件进行支持。

Hibernate 的诞生是因为 Gavin King 发现 EJB 中 CMP（容器管理持久化）实在是太难用了，他认为应该使用一个更加纯粹，更加轻便的 ORM 框架，于是他就自己写了一个，虽然那时的他对 SQL 还不是很熟悉（他开发 Hibernate 的第一件事是去街上买一本 SQL 基础的书）。

Hibernate 对 Java 世界影响巨大，他和 Spring 一起颠覆了 EJB，享誉全球，但是 Hibernate 出现以后，人们也慢慢发现了 Hibernate 这样的全自动 ORM 框架在性能方面存在严重先天缺陷：由于程序员可以使用 HQL 和 Criteria 操纵数据库（这两种方式实际上还是通过向数据库发送 SQL 来实现的，只不过 SQL 是由 Hibernate 自动生成的），数据库管理员（DBA,Database Administrator）很难对 SQL 进行优化，而数据库通常又是一个系统的性能瓶颈，如果很难对 SQL 进行优化，整个系统的性能都会受到影响。

另一方面，以前人们只要直接书写 SQL 并拼装对象就好，但是现在我们不得不考虑对象关系，延迟加载，二级缓存，查询方式等更多的问题，然后要对这些进行调试和优化，这无疑在带来便利的同时也提升了开发的复杂度和工作量。这是 Hibernate 的另外一个缺点。

2、MyBatis

Hibernate 是全自动 ORM 框架，我们完全不用书写 SQL，只要操作对象就可以完成一次数据库操作，但是 Hibernate 的缺点也明显，上一节已经提及，这里不再赘述。反思一下，我们到底想要什么？

对于更多的程序员来说，学习 SQL 的成效更为直接和显著，使用对象化的数据库操作方式反而繁琐，带来的问题也令人棘手。我们还是应该直接使用 SQL，那么 Hibernate 还提供了那些便利呢？

1、连接管理；2、对象的拼装与拆解

那么，我们能不能在提供以上两个功能点的同时，再添加一下对直接书写 SQL 的支持呢？没错，这就是

MyBatis 框架。

MyBatis 是由原 Apache 基金会的开源项目 iBatis 改名而来。MyBatis 提供连接管理，参数和结果集映射，并且它将 SQL 作为一种资源来管理，便于我们查找和复用。使用 MyBatis 我们不必再书写 JDBC 式的样板代码，不用手动拼装结果集对象。

MyBatis 通过 XML 配置 SQL 语句，基本的四个标签<insert><delete><update><select>对应着 DML 的四种操作，以<select>标签为例，你可以向它传入参参数，并指定返回类型，MyBatis 将通过反射帮你自动拼装好对象，省去了手动组装对象的麻烦。使用 MyBatis 后你会发现 Dao 层几乎消失了，Dao 层甚至都不需要有实现代码，紧紧留出一个接口即可。

下面是用法示例:QuestionReplyDao.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.yuoffer.wechat.dao.QuestionReplyDao">

    <select id="getByRecentType" resultType="com.yuoffer.wechat.model.Question">
        select
                wq.id as id,
                wq.question as question,
                wq.anwser as anwser,
                wq.aspectid as aspectid,
                wa.keyword as keyword,
                wa.categoryid as categoryid,
                wc.category as category,
                wc.type as typeid,
                wt.type as type
        from
                wx_question wq
        join
                wx_question_aspect wa
        on
                wq.aspectid = wa.id
        join
                wx_question_category wc
        on
                wa.categoryid = wc.id
        join
                wx_question_type wt
        on
                wc.type = wt.id
                and wt.type =
                <if test="initType != null">
                    #{initType}
                </if>
                <if test="initType == null">
                    (
                        select
                            t.type as type
                        from
                            wx_question_type t
                        join
                            wx_question_category c
                        on
```

```
                                t.id = c.type
                            join
                                wx_question_aspect a
                            on
                                a.categoryid = c.id
                            join
                                wx_question q
                            on
                                q.aspectid = a.id
                                and q.id = (
                                    select
                                        questionid
                                    from
                                        wx_ask_record
                                    where
                                        openid = #{openid}
                                    order by
                                        create_time
                                    desc limit 1
                                )
                            union select
                                'TECH' as type
                            limit 1
                        )
                    </if>
            and
                    wq.id not in (
                        select
                            questionid
                        from
                            wx_ask_record
                        where
                            openid = #{openid}
                    )
            offset
                    #{base}
            limit 1
    </mapper>
```

QuestionReplyDao.java:

```
    package com.yuoffer.wechat.dao;

    import com.yuoffer.wechat.model.Question;
    import org.apache.ibatis.annotations.Param;
    import org.springframework.stereotype.Repository;

    import java.sql.Timestamp;
    import java.util.List;

    /**
     * Created by zl
     */
    @Repository
    public interface QuestionReplyDao {
        public Question getByRecentType(@Param("openid") String openid,@Param("base") Integer
    base,@Param("initType") String type);
    }
```

你发现上面的 SQL 语句中还包含了<if>标签，来保证不同的参数条件下，执行不同的 SQL 语句，这样的标签复用了 SQL 语句，增强了逻辑，这种特性被称为动态 SQL，这是 MyBatis 的一个特性，用起来非常方便。

另外，我们在 Java 中调用这些 SQL 的时候，根本不需要写出 Dao 的实现，只需要按照约定写出一个接口即可，MyBatis 会给我们返回一个相应的代理，调用时其实是调用了 MyBatis 依据我们 XML 生成的代码，原理同 JSP 一样。

MyBatis 更加轻量，使用起来简单可靠，并且性能调优可以直接由 DBA 来完成，因此，近年来 MyBatis 更加流行一些。