# Jdk与jre

## Jdk（java development kit）

java开发工具包，是程序员使用java语言编写java程序所需的开发工具包，是提供给程序员使用的。JDK包含了JRE，同时还包含了编译java源码的编译器javac，还包含了很多java程序调试和分析的工具：jconsole，jvisualvm等工具软件，还包含了java程序编写所需的文档和demo例子程序。

## Jre（java runtime environment）

JRE顾名思义是java运行时环境，包含了java虚拟机，java基础类库。是使用java语言编写的程序运行所需要的软件环境，是提供给想运行java程序的用户使用的。

另外，安装JRE的时候安装程序会自动把JRE的java.exe添加到了系统变量中。系统变量Path的最前面有%SystemRoot%system32;%SystemRoot%;这样的配置，那样到Windows/system32目录下main去看看，会发现一个java.exe文件。这样就无需配置环境变量，也可以运行Java程序了。Jdk有javac。

## JDK与JRE的区别

JDK是Java的开发工具，它不仅提供了Java程序运行所需的JRE，还提供了一系列的编译，运行等工具，如javac，java，javaw等。JRE只是Java程序的运行环境，它最核心的内容就是JVM（Java虚拟机）及核心类库。

# Object类

是所有类的父类。一个类默认继承object类

## 常用方法

**toString() 默认输出对象地址**

**equals() 用于比较两个对象是否是同一对象（String类进行了重写，使其可以比较字符串是否相等）**

**hashCode()对象签名 该方法用来返回其所在对象的物理地址（哈希码值），常会和equals方法同时重写，确保相等的两个对象拥有相等的.hashCode。**

**clone()**

浅拷贝：创建一个新对象，然后将当前对象的非静态字段复制到该对象，如果字段类型是值类型（基本类型）的，那么对该字段进行复制；如果字段是引用类型的，则只复制该字段的引用而不复制引用指向的对象。此时新对象里面的引用类型字段相当于是原始对象里面引用类型字段的一个副本，原始对象与新对象里面的引用字段指向的是同一个对象。

深拷贝即将引用类型的属性内容也拷贝一份新的。（实现：序列化、重写clone（） ）

Wait() notify() notifyAll()

# 字符串类

## string

1，Stirng是对象不是基本数据类型

2，String是final类，不能被继承。是不可变对象，一旦创建，就不能修改它的值。

3，对于已经存在的Stirng对象，修改它的值，就是重新创建一个对象，然后将新值赋予这个对象

## stringBuffer

　　 1，一个类似于 String 的字符串缓冲区，对它的修改的不会像String那样重创建对象。

　　 2，使用append()方法修改Stringbuffer的值，使用toString()方法转换为字符串。

## stringBuild

是jdk1.5后用来替换stringBuffer的一个类，大多数时候可以替换StringBuffer。和StringBuffer的区别在于Stringbuild是一个单线程使用的类，不值执行线程同步所以比 StringBuffer的速度快，效率高。是线程非安全的

**String s="abc"及String s=new String("abc")**

# IO流



# 网络编程

## Bio

同步阻塞模型，服务端提供IP和监听端口，客户端通过连接操作想服务端监听的地址发起连接请求。采用BIO通信模型的服务端，通常由一个独立的Acceptor线程负责监听客户端的连接，它接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理没处理完成后，通过输出流返回应答给客户端，线程销毁。即典型的一请求一应答通宵模型。（线程池实现m:n？）

## Nio

同步非阻塞I/O：java使用多路复用器（Selector）。Selector提供选择已经就绪的任务的能力：Selector会不断轮询注册在其上的Channel（负责从缓存区读写数据），如果某个Channel上面发生读或者写事件，这个Channel就处于就绪状态，会被Selector轮询出来，然后通过SelectionKey可以获取就绪Channel的集合，进行后续的I/O操作。

因为应答消息的发送，SocketChannel也是异步非阻塞的，所以不能保证一次能把需要发送的数据发送完，此时就会出现写半包的问题。我们需要注册写操作，不断轮询Selector将没有发送完的消息发送完毕，然后通过Buffer的hasRemain()方法判断消息是否发送完成。

## Aio

异步阻塞

NIO是同步的IO，是因为程序需要IO操作时，必须获得了IO权限后亲自进行IO操作才能进行下一步操作。AIO是对NIO的改进（所以AIO又叫NIO.2），它是基于Proactor模型的。每个socket连接在事件分离器注册 IO完成事件 和 IO完成事件处理器。程序需要进行IO时，向分离器发出IO请求并把所用的Buffer区域告知分离器，分离器通知操作系统进行IO操作，操作系统自己不断尝试获取IO权限并进行IO操作（数据保存在Buffer区），操作完成后通知分离器；分离器检测到 IO完成事件，则激活 IO完成事件处理器，处理器会通知程序说“IO已完成”，程序知道后就直接从Buffer区进行数据的读写。

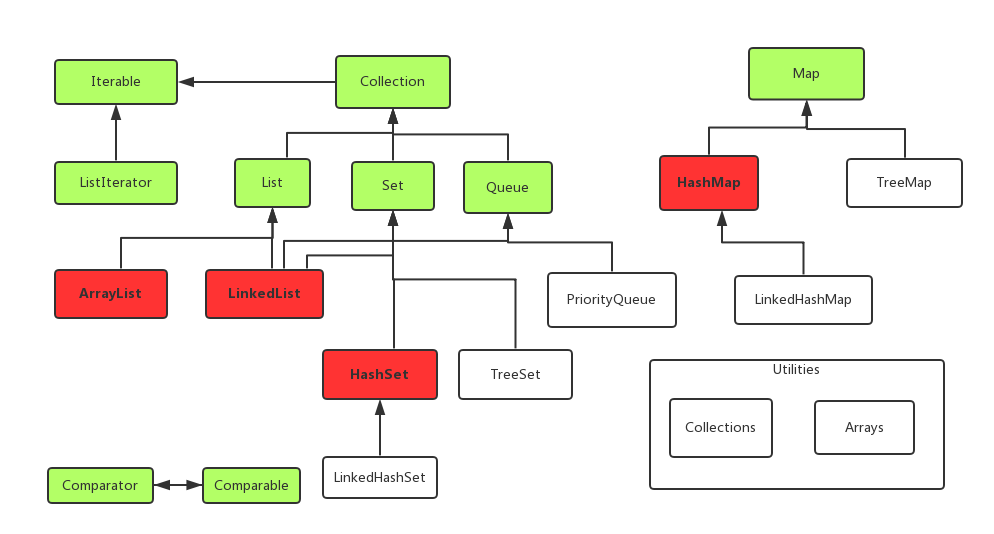
AIO是发出IO请求后，由操作系统自己去获取IO权限并进行IO操作；NIO则是发出IO请求后，由线程不断尝试获取IO权限，获取到后通知应用程序自己进行IO操作。

### Files

文件的IO操作

<https://www.jianshu.com/p/3cb5ca04e3c8>

# 容器



HashSet 基于 HashMap 实现的；

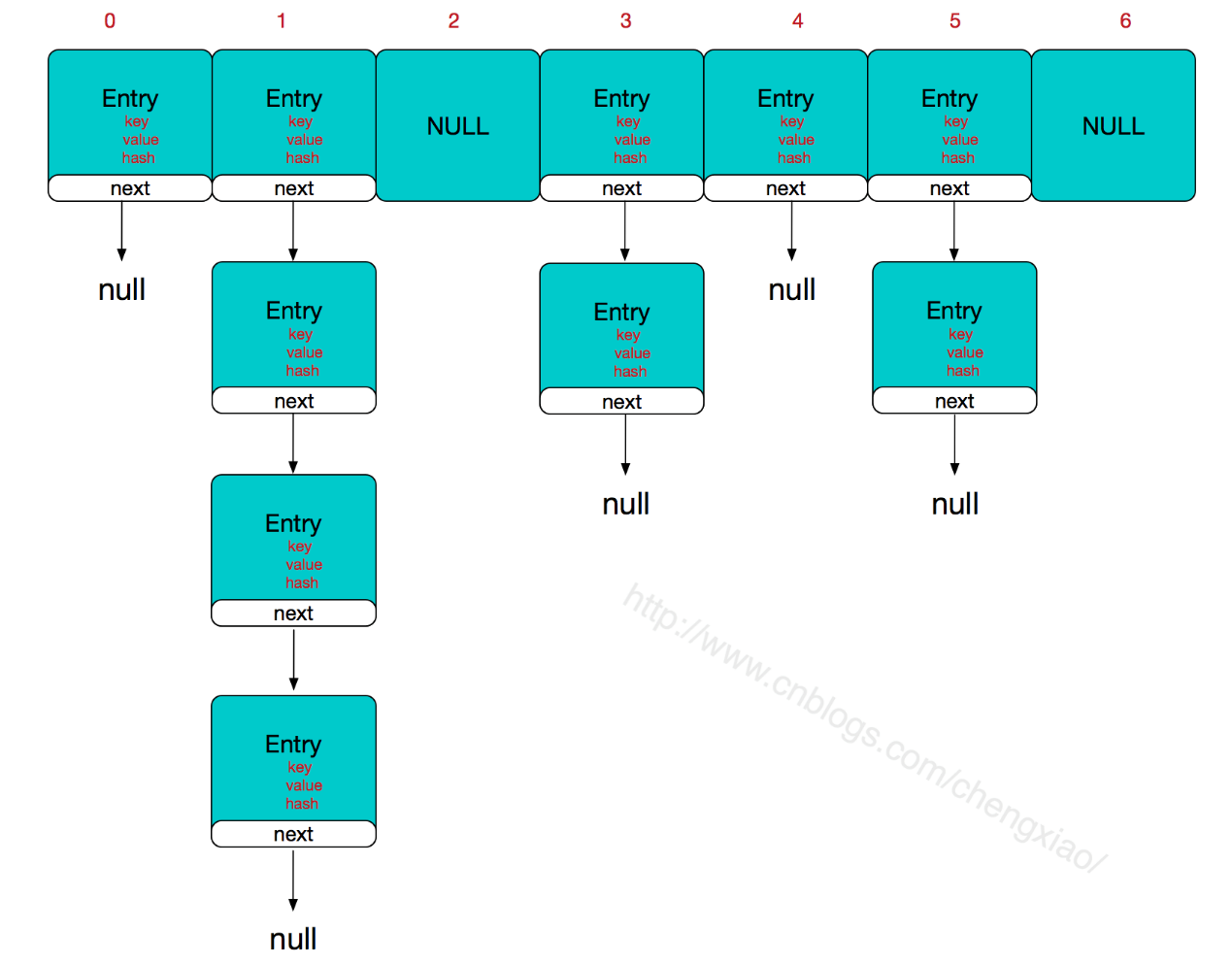
## Collections(集合工具类)

1. 排序(Sort)
2. 混排（Shuffling）
3. 反转(Reverse)
4. 替换所以的元素(Fill)
5. 拷贝(Copy)
6. 返回Collections中最小元素(min)
7. 返回Collections中最大元素(max)
8. lastIndexOfSubList返回指定源列表中最后一次出现指定目标列表的起始位置

## hashmap Hashset

HashMap有4个构造器，其他构造器如果用户没有传入initialCapacity 和loadFactor这两个参数，会使用默认值

initialCapacity默认为16，loadFactory默认为0.75



Java7之前如上图，java8之后，如果链表长度大于8，则变成红黑树

为什么是8？

红黑树的平均查找长度是log(n)，长度为8，查找长度为log(8)=3，链表的平均查找长度为n/2，当长度为8时，平均查找长度为8/2=4，这才有转换成树的必要；链表长度如果是小于等于6，6/2=3，虽然速度也很快的，但是转化为树结构和生成树的时间并不会太短。

还有选择6和8的原因是：

中间有个差值7可以防止链表和树之间频繁的转换。假设一下，如果设计成链表个数超过8则链表转换成树结构，链表个数小于8则树结构转换成链表，如果一个HashMap不停的插入、删除元素，链表个数在8左右徘徊，就会频繁的发生树转链表、链表转树，效率会很低。

# 多线程

## Java并发编程

并行：多个cpu同时运行

并发：多个线程操作相同的资源，保证线程的安全性

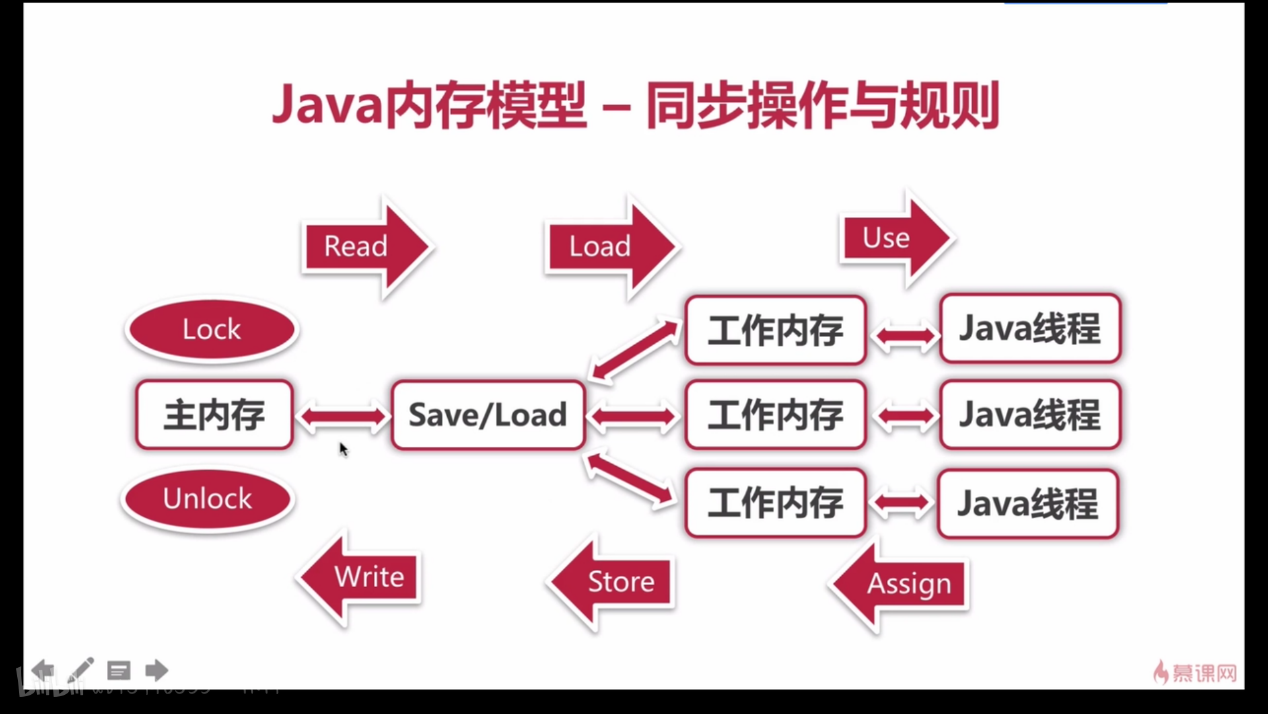
高并发：同时处理很多请求，提高程序性能。

进程：给个程序被调度到内存运行，就是一个进程

线程：进程里的一个小任务

### Jmm（java内存模型）：

其里面的八个操作都具备原子性



### 线程的安全性

原子性：操作是整体性的，不可再分。也可以说同一时刻只能有一个线程进行访问，错做不可分。

可见性：一个线程对一个线程的共享变量的修改对于另一个线程来说是可以见到的。

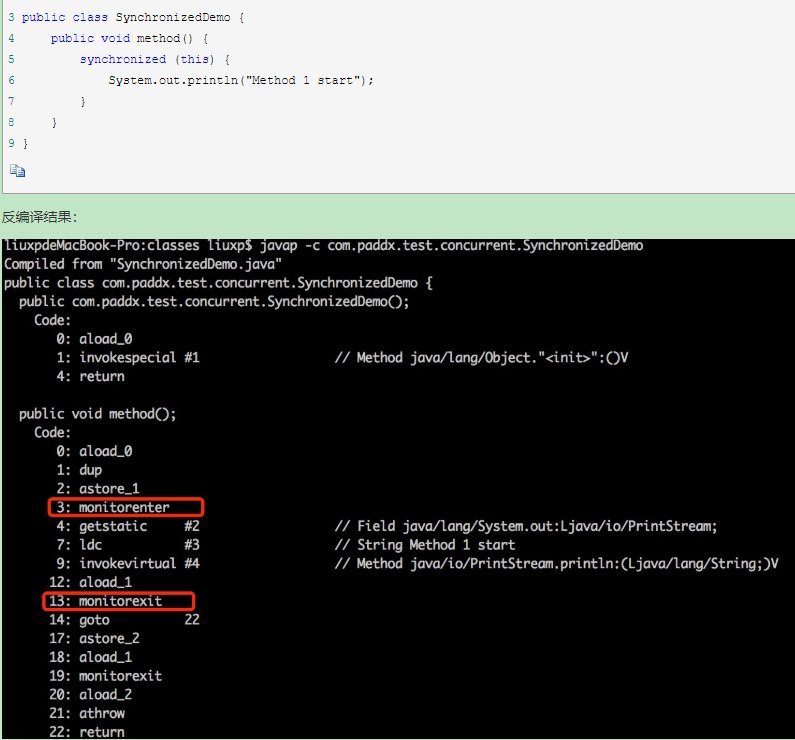
有序性：happens-before原则

1. 程序次序规则：一个线程内，按照代码顺序，（看起来）书写在前面的的操作先行发生与书写在后面的操做
2. 锁定规则：一个unlock操作先行发生于后面对同一个锁的lock操作
3. Volatile变量规则：写先发生与读（对于同一个变量）
4. 传递规则：如果A<B B<C，那么 A<C（<:表示先于）

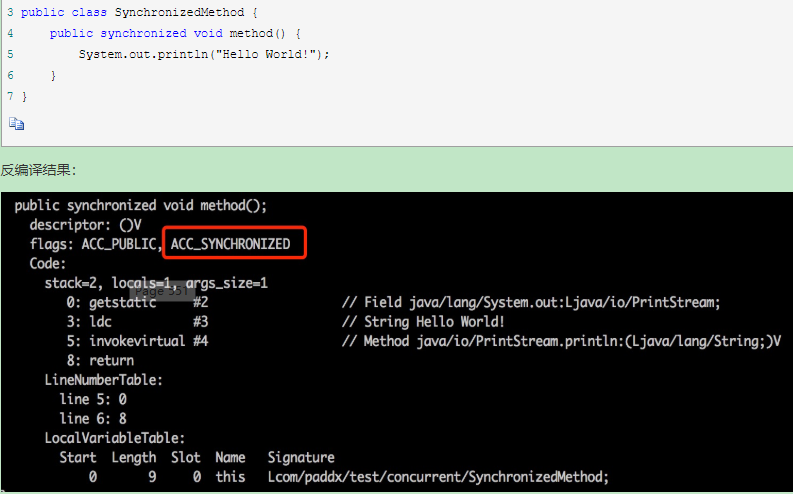
#### Synchronized

修饰代码块：

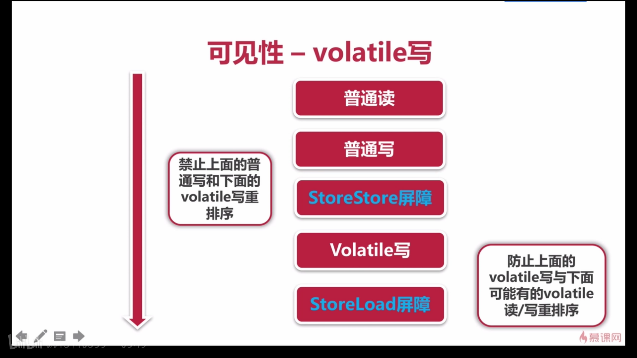
根据虚拟机规范要求，在执行monitorenter指令时，首先要尝试获取对象锁，也就是上文我们提到了monitor对象。如果这个对象没有被锁定，或者当前线程已经拥有了这个对象的锁，那么就把锁的计数器（\_count）加1。当然与之对应执行monitorexit指令时，锁的计数器（\_count）也会减1。第二个monitorexit为异常时运行，确保每个+1都有一个-1对应。



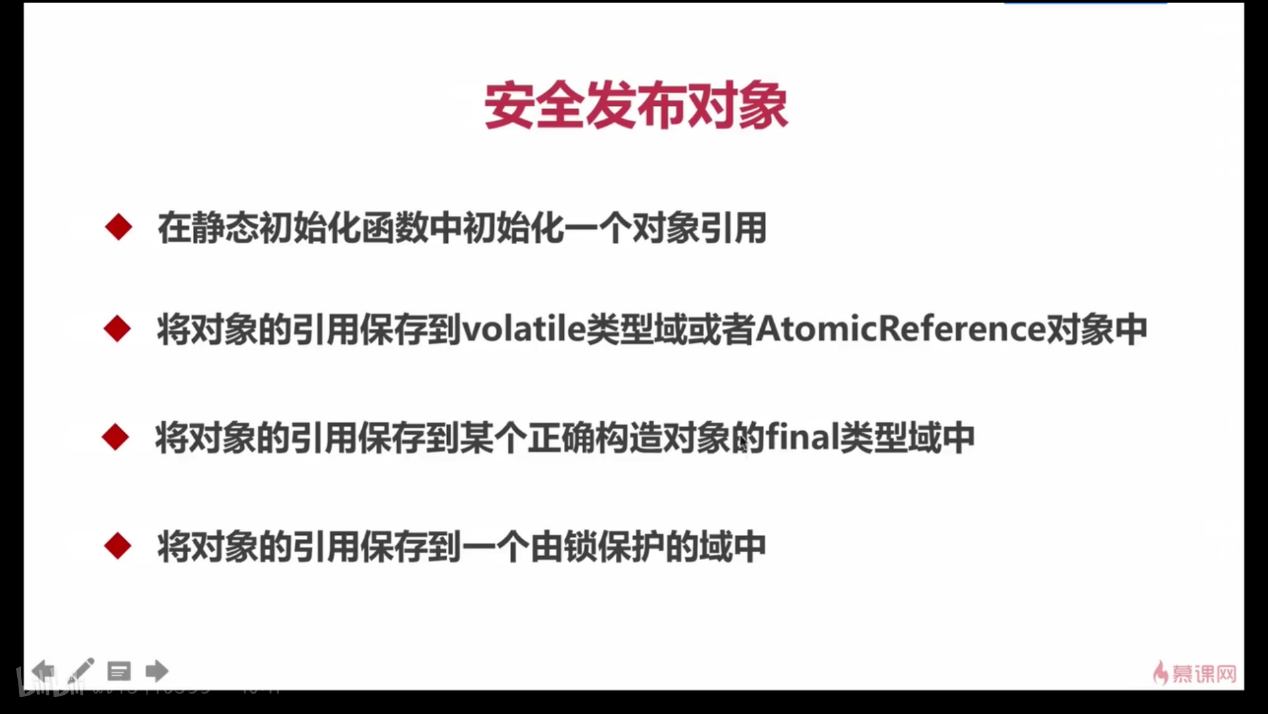
从反编译的结果来看，方法的同步并没有通过指令monitorenter和monitorexit来完成（理论上其实也可以通过这两条指令来实现），不过相对于普通方法，其常量池中多了ACC\_SYNCHRONIZED标示符。JVM就是根据该标示符来实现方法的同步的：当方法调用时，调用指令将会检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。在方法执行期间，其他任何线程都无法再获得同一个monitor对象。 其实本质上没有区别，只是方法的同步是一种隐式的方式来实现，无需通过字节码来完成。



#### Volatile：



#### 安全发布对象



### 不可变对象

满足条件：对象创建以后其状态就不能修改

对象所有域都是final类型

对象是正确创建的（this引用没有溢出）

#### 不可变对象集合(以map为例子)：

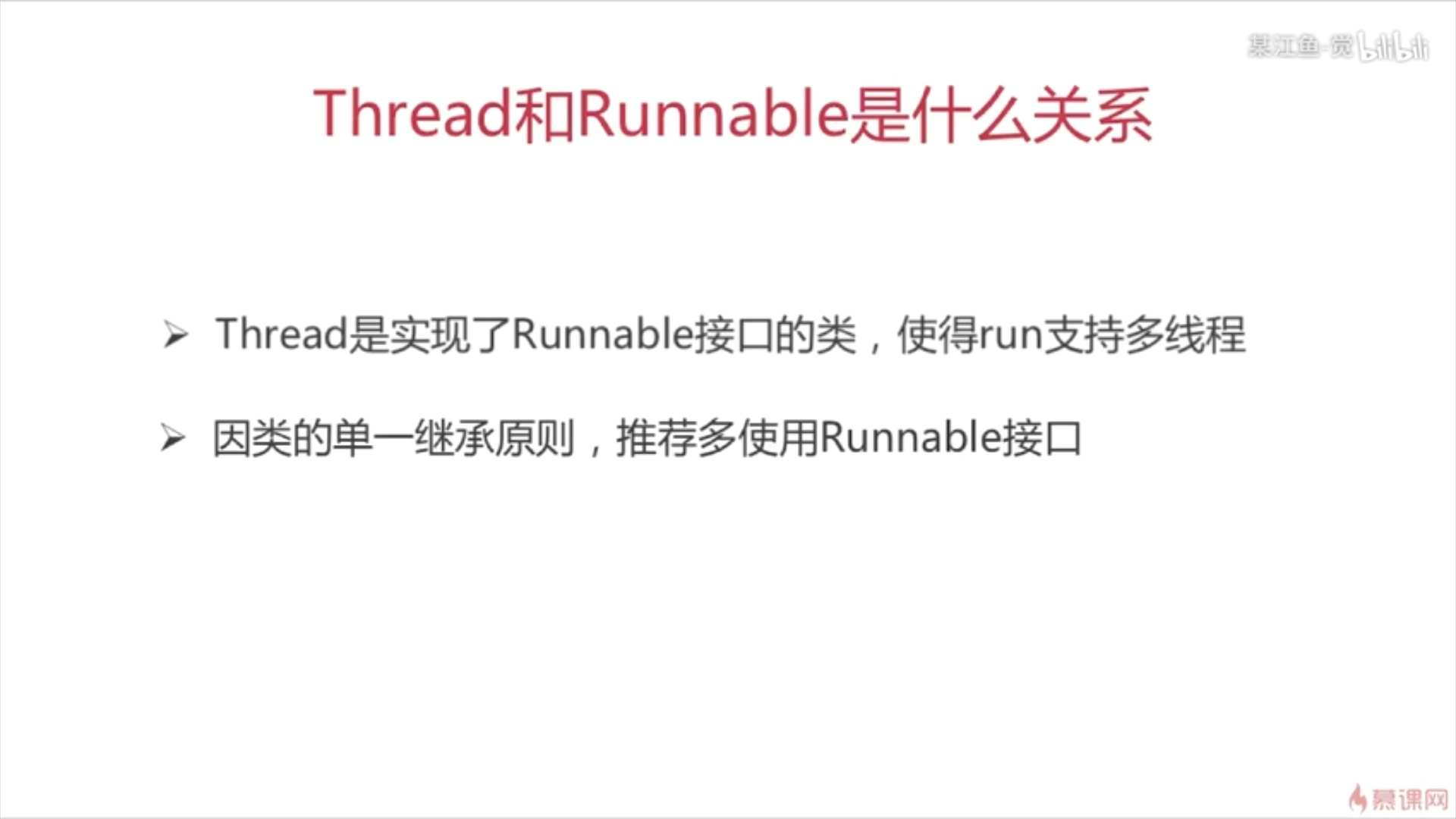
Collections.unmodifiableMap( mapclass );

Immutablemap

## 线程的run与start区别：



## Thread与 runnable关系



## JAVA线程状态：



## 同步容器：

ArrayList -------🡪 Vector,Stack(内部使用synchornized,线程可能不安全)

HashMap-------🡪HashTable(Key value不能为空)

Collections.synchronizedxxx(list,set,map)

## 并发容器J.U.C

Arraylist----🡪CopyOnWriteArrayList(增加或者删除元素耗费内存，无法满足实时性)推荐：多读少改

HashSet----🡪CopyOnWriteArraySet(基于CopyOnWriteArrayList)

HashMap--🡪ConcurrentHashMap

TreeSet--🡪ConcurrentSkipListSet

TreeMap--🡪ConcurrentSkipListMap

## AQS(abstract Queued Synchronized)

### 组件：

CountDownLatch:计数器减到0，处于等待状态的线程才会继续执行

Semaphore:能控制同一时间并发线程数

CyclicBarrier:类似CountDownLatch,计数器可以重置，当到达屏障时，执行线程CyclicBarrier.await()

ReentrantLock(jdk实现的锁):类似与Synchronized,手工加锁和释放锁，可指定公平锁或者不公平锁（默认）

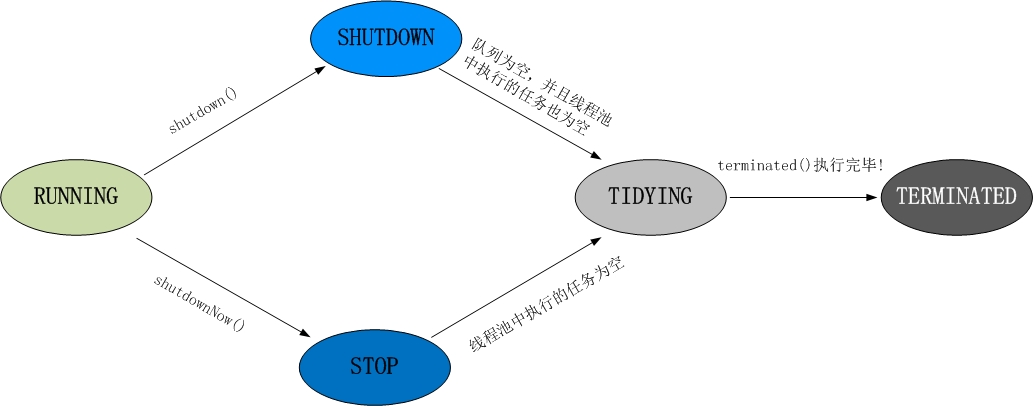
Condition:分组唤醒需要唤醒的线程

FutureTask:可获得返回值的线程

<https://github.com/sjsmi1e/ThreadsTest>

## 线程池：

### 五种状态：



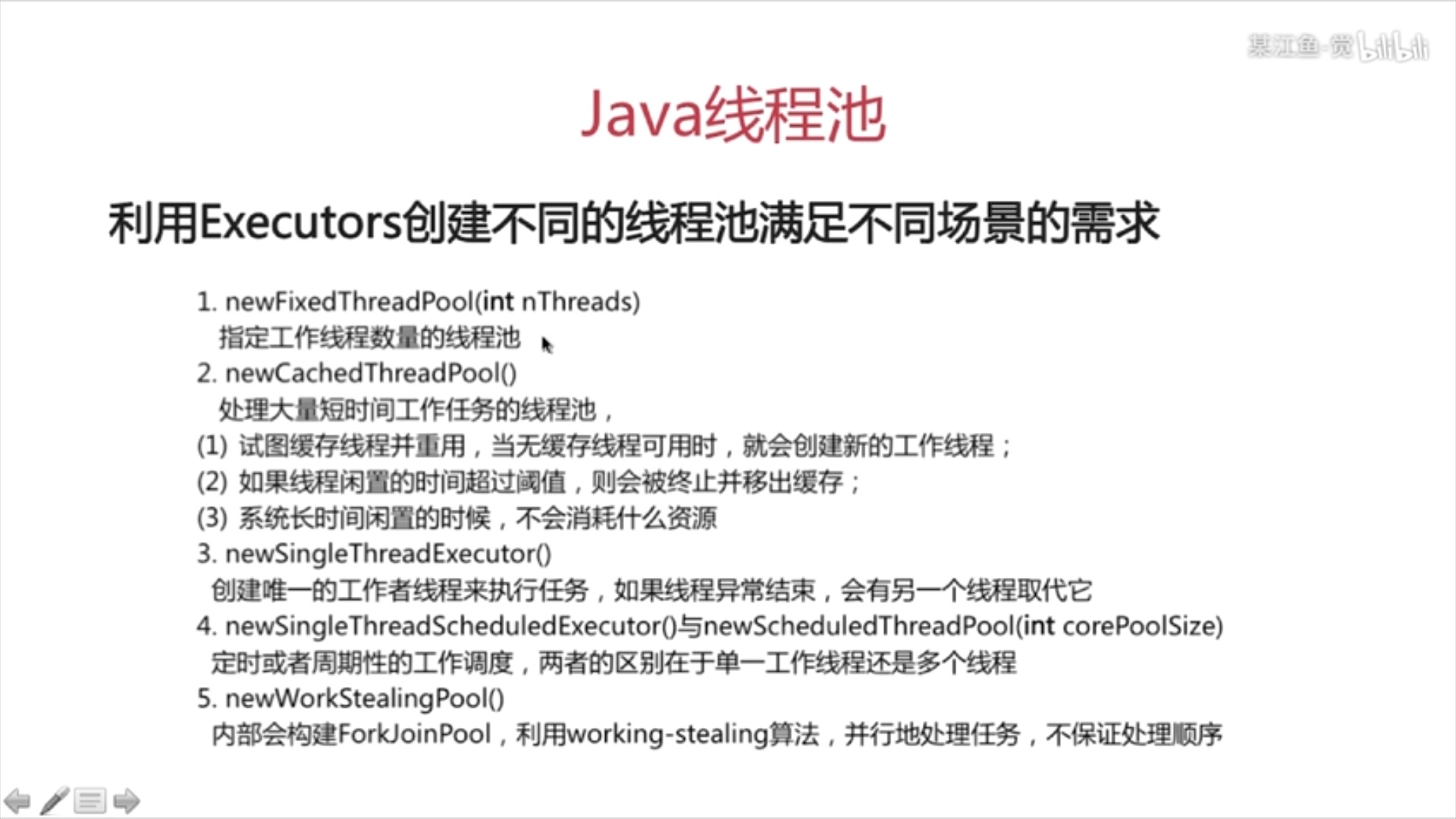
Executor是一个顶层接口，在它里面只声明了一个方法execute(Runnable)，返回值为void，参数为Runnable类型，从字面意思可以理解，就是用来执行传进去的任务的；

然后ExecutorService接口继承了Executor接口，并声明了一些方法：submit、invokeAll、invokeAny以及shutDown等；

抽象类AbstractExecutorService实现了ExecutorService接口，基本实现了ExecutorService中声明的所有方法；

然后ThreadPoolExecutor继承了类AbstractExecutorService。

### 线程池的类别



## Wait与sleep区别：

**从使用角度看**

sleep是Thread线程类的方法，而wait是Object顶级类的方法。

sleep可以在任何地方使用，而wait只能在同步方法或者同步块中使用。

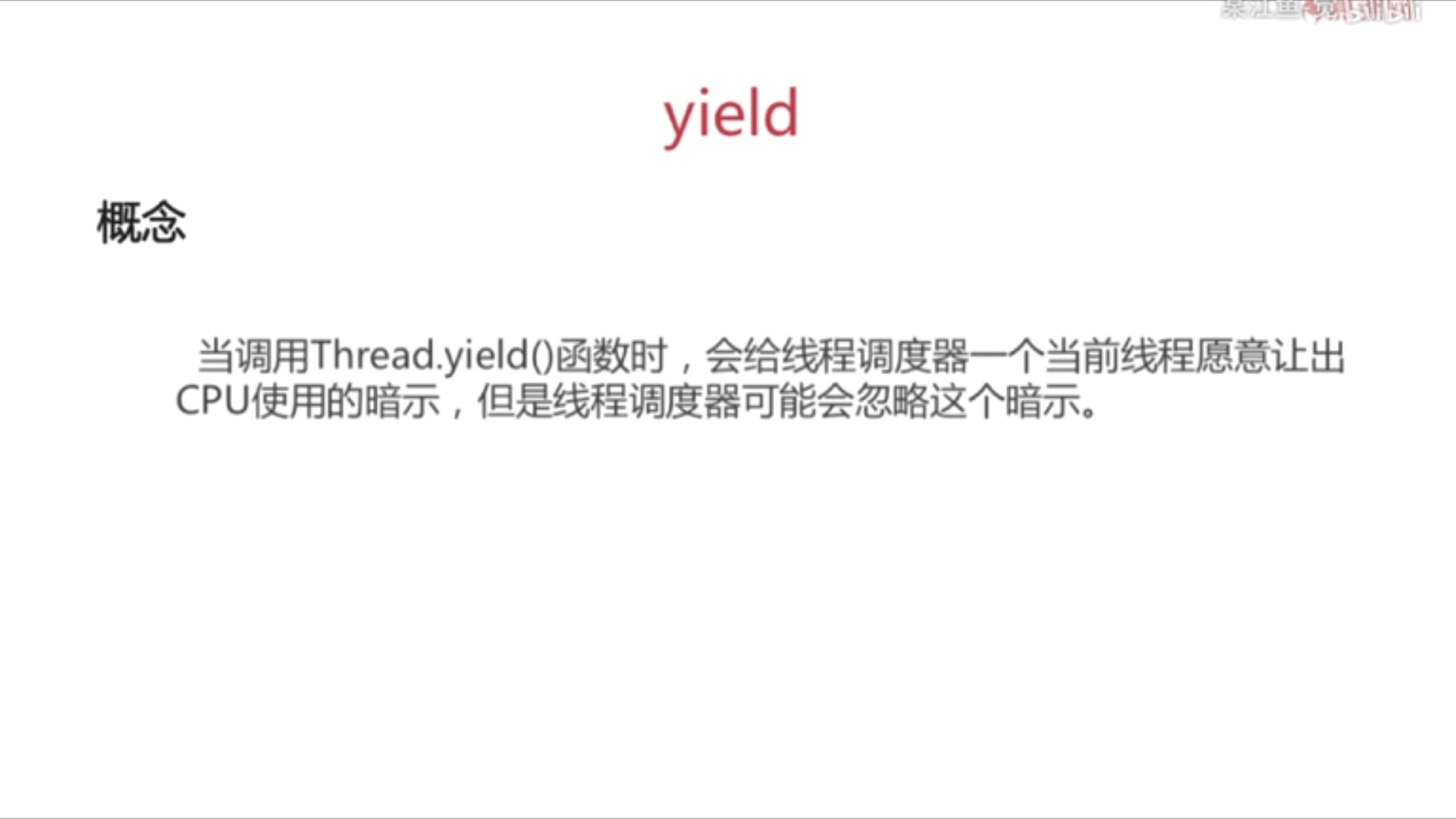
**CPU及资源锁释放**

sleep,wait调用后都会暂停当前线程并让出cpu的执行时间，但不同的是sleep不会释放当前持有的对象的锁资源，到时间后会继续执行，而wait会放弃所有锁并需要notify/notifyAll后重新获取到对象锁资源后才能继续执行。

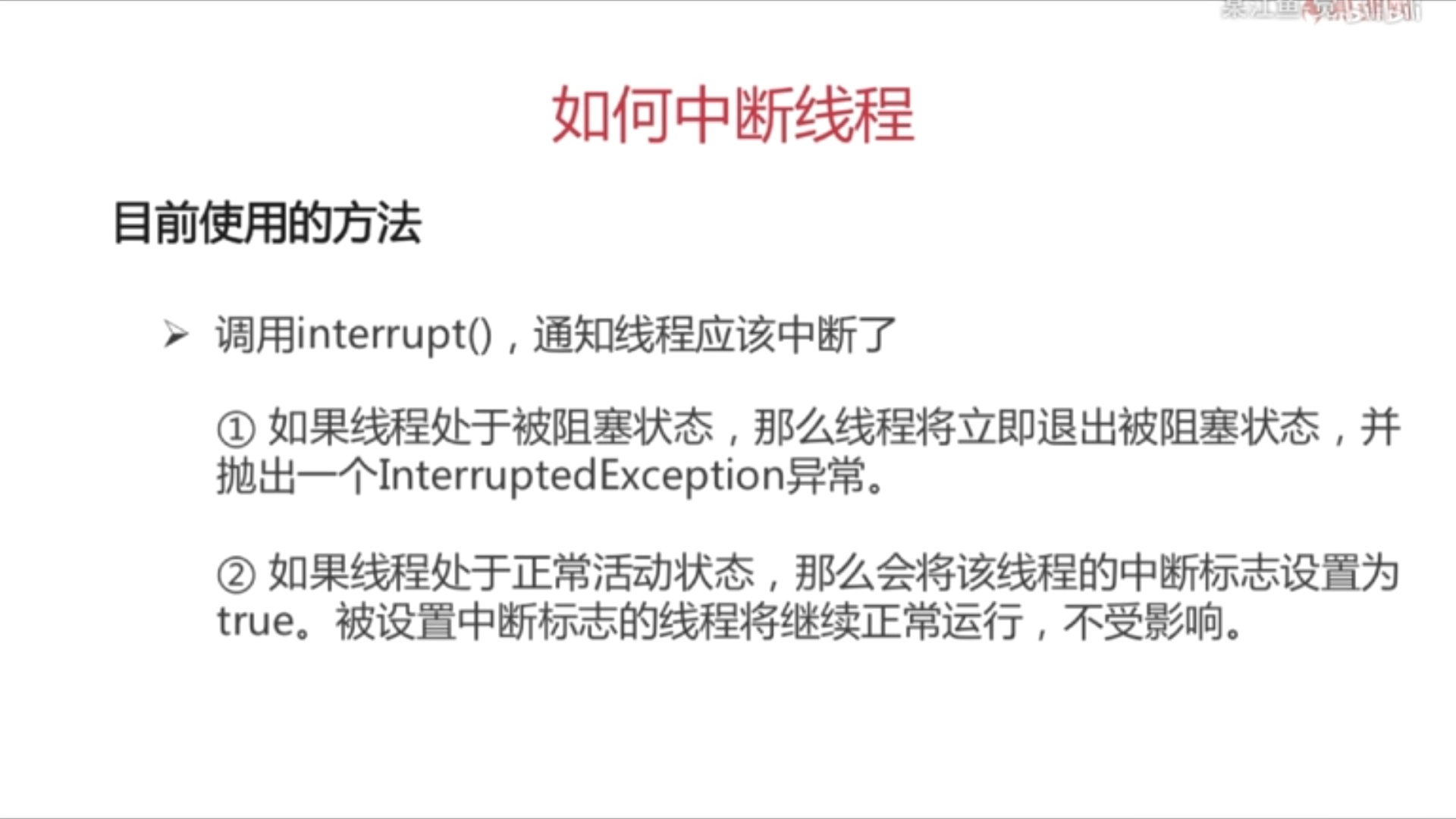
## Notify与notifyAll的区别：



## Yield：



## 中断线程（interrupt）：



## ThreadLocal

ThreadLocal，很多地方叫做线程本地变量，也有些地方叫做线程本地存储，其实意思差不多。可能很多朋友都知道ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量(类似于属于自己属性的变量，和其他的不同)。

#### 深入解析

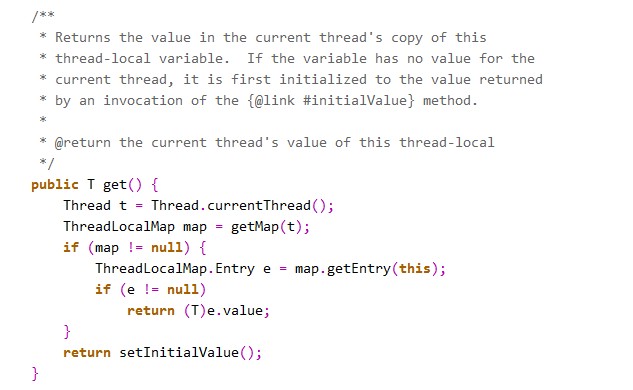
先了解一下ThreadLocal类提供的几个方法：

|  |  |
| --- | --- |
| 1  2  3  4 | public T get() { }  public void set(T value) { }  public void remove() { }  protected T initialValue() { } |

 　　get()方法是用来获取ThreadLocal在当前线程中保存的变量副本，set()用来设置当前线程中变量的副本，remove()用来移除当前线程中变量的副本，initialValue()是一个protected方法，一般是用来在使用时进行重写的，它是一个延迟加载方法，下面会详细说明。

　　首先我们来看一下ThreadLocal类是如何为每个线程创建一个变量的副本的。

　　先看下get方法的实现：



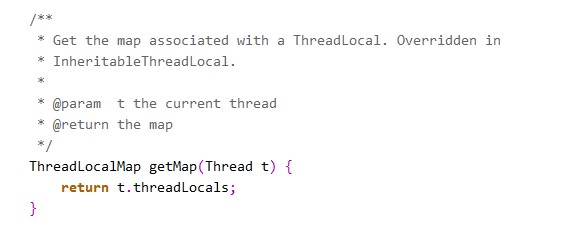
 　　第一句是取得当前线程，然后通过getMap(t)方法获取到一个map，map的类型为ThreadLocalMap。然后接着下面获取到<key,value>键值对，注意这里获取键值对传进去的是  this，而不是当前线程t。

　　如果获取成功，则返回value值。

　　如果map为空，则调用setInitialValue方法返回value。

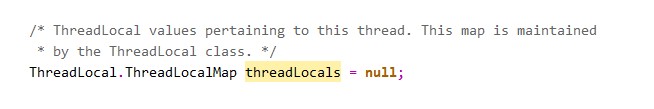
　　我们上面的每一句来仔细分析：

　　首先看一下getMap方法中做了什么：

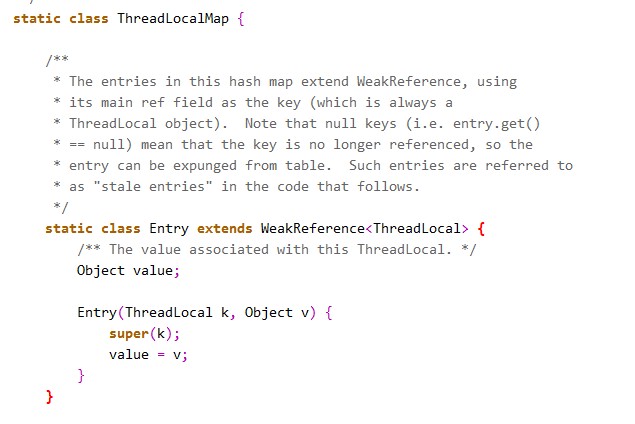


　　可能大家没有想到的是，在getMap中，是调用当期线程t，返回当前线程t中的一个成员变量threadLocals。

　　那么我们继续取Thread类中取看一下成员变量threadLocals是什么：

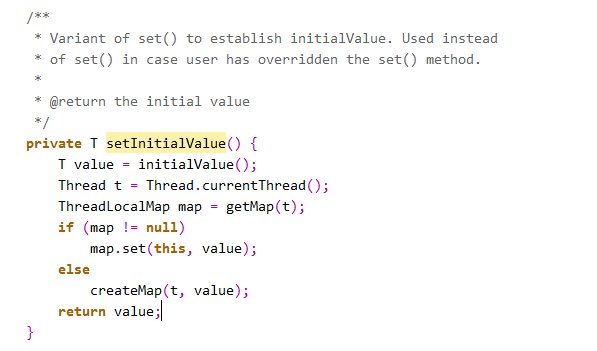


　　实际上就是一个ThreadLocalMap，这个类型是ThreadLocal类的一个内部类，我们继续取看ThreadLocalMap的实现：

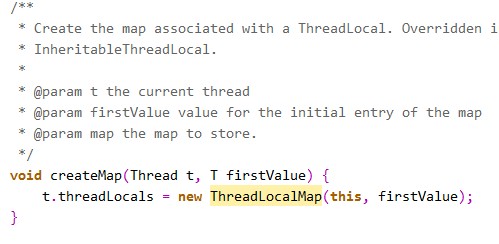


　　可以看到ThreadLocalMap的Entry继承了WeakReference，并且使用ThreadLocal作为键值。

　　然后再继续看setInitialValue方法的具体实现：



　　很容易了解，就是如果map不为空，就设置键值对，为空，再创建Map，看一下createMap的实现：



　　至此，可能大部分朋友已经明白了ThreadLocal是如何为每个线程创建变量的副本的：

　　首先，在每个线程Thread内部有一个ThreadLocal.ThreadLocalMap类型的成员变量threadLocals，这个threadLocals就是用来存储实际的变量副本的，键值为当前ThreadLocal变量，value为变量副本（即T类型的变量）。

　　初始时，在Thread里面，threadLocals为空，当通过ThreadLocal变量调用get()方法或者set()方法，就会对Thread类中的threadLocals进行初始化，并且以当前ThreadLocal变量为键值，以ThreadLocal要保存的副本变量为value，存到threadLocals。

　　然后在当前线程里面，如果要使用副本变量，就可以通过get方法在threadLocals里面查找。

## 线程安全：

当多个线程访问某一个类时，这个类始终能表现出正常的行为，称这个类是安全的

## 锁竞争：

Static synchronized 方法名--------此方法为类一级别锁（独占.class类）

Synchronized（同步锁）：同步线程安全满足原子性，可见性

可锁重入：获得其他方法的为被其他线程调用的锁

synchronizedException

Asynchronize

Volatile:使变量在多个线程间可见(非原子性)

autom

线程之间通信：wait(释放锁)/notify(不释放锁)必须配合synchronized

countDownLatch

单例与多线程：懒汉与

Innersingleton

Dubblesingleton

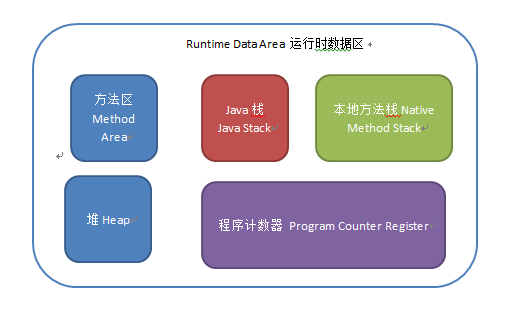
同步类容器：

Collections.xxxxx

Future模式

Master-Worker

# JVM



## 反射：

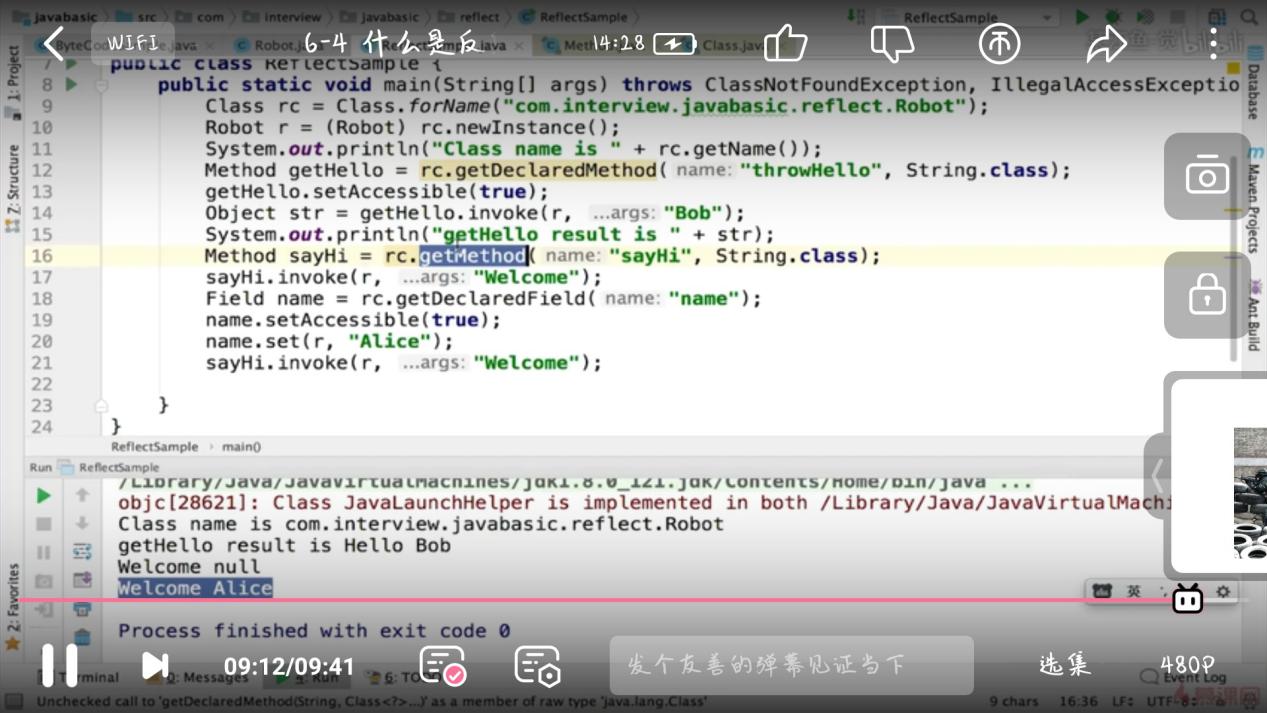
### 定义：

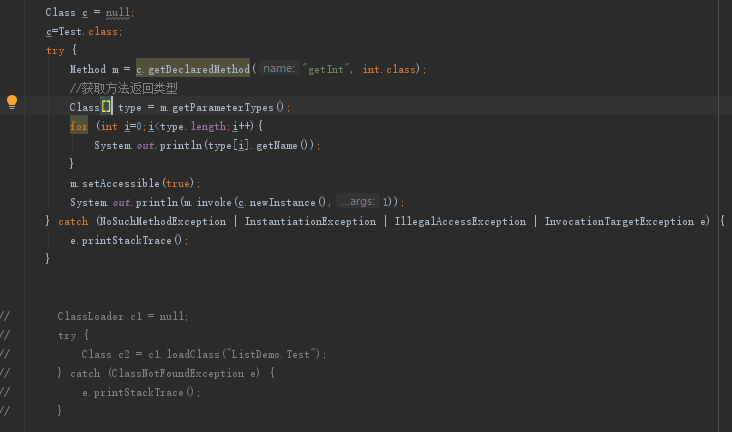
Java反射就是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；并且能改变它的属性。而这也是Java被视为动态语言的一个关键性质。

### 作用：

我们知道反射机制允许程序在运行时取得任何一个已知名称的class的内部信息，包括包括其modifiers(修饰符)，fields(属性)，methods(方法)等，并可于运行时改变fields内容或调用methods。那么我们便可以更灵活的编写代码，代码可以在运行时装配，无需在组件之间进行源代码链接，降低代码的耦合度；还有动态代理的实现等等；但是需要注意的是反射使用不当会造成很高的资源消耗！

### 简单实现：



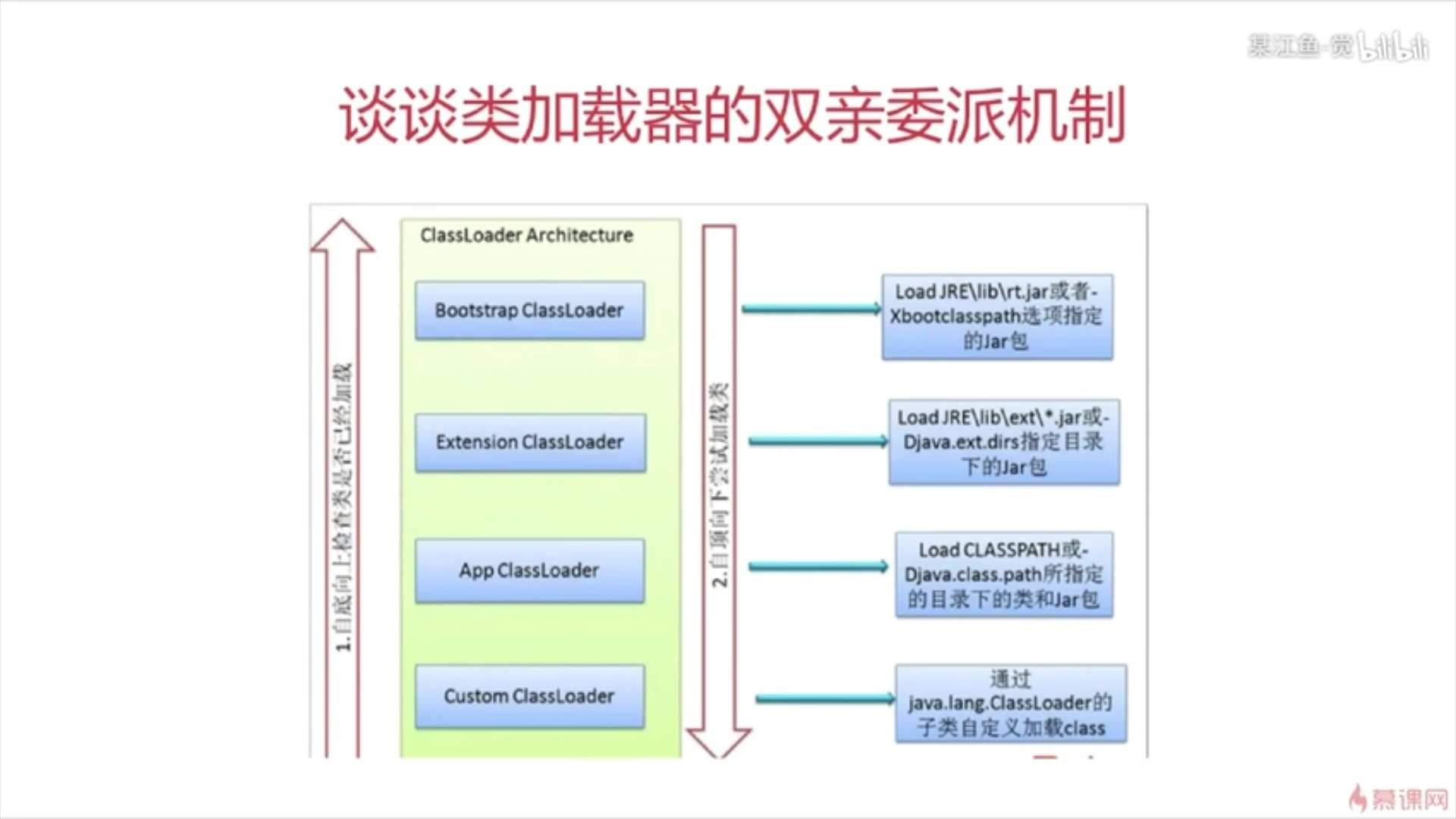


## 类加载子系统：



### 双亲委派机制：

双亲委派模型工作过程是：如果一个类加载器收到类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器完成。每个类加载器都是如此，只有当父加载器在自己的搜索范围内找不到指定的类时（即ClassNotFoundException），子加载器才会尝试自己去加载。

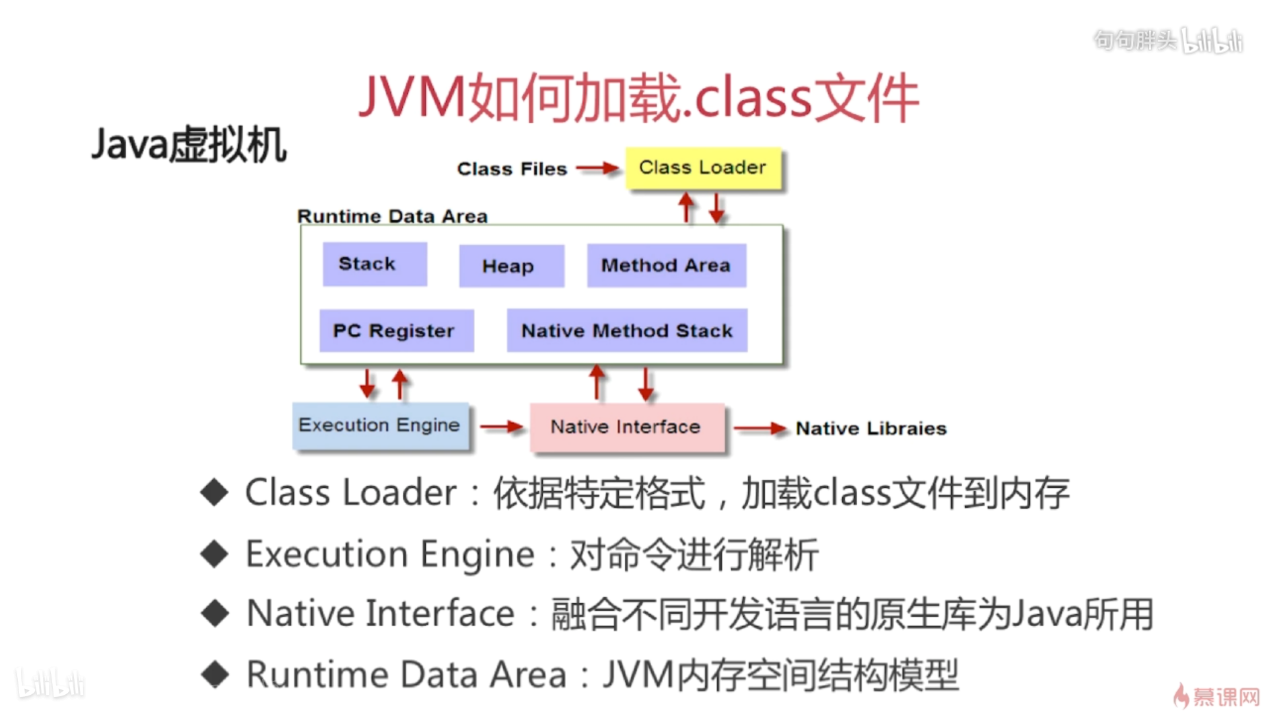


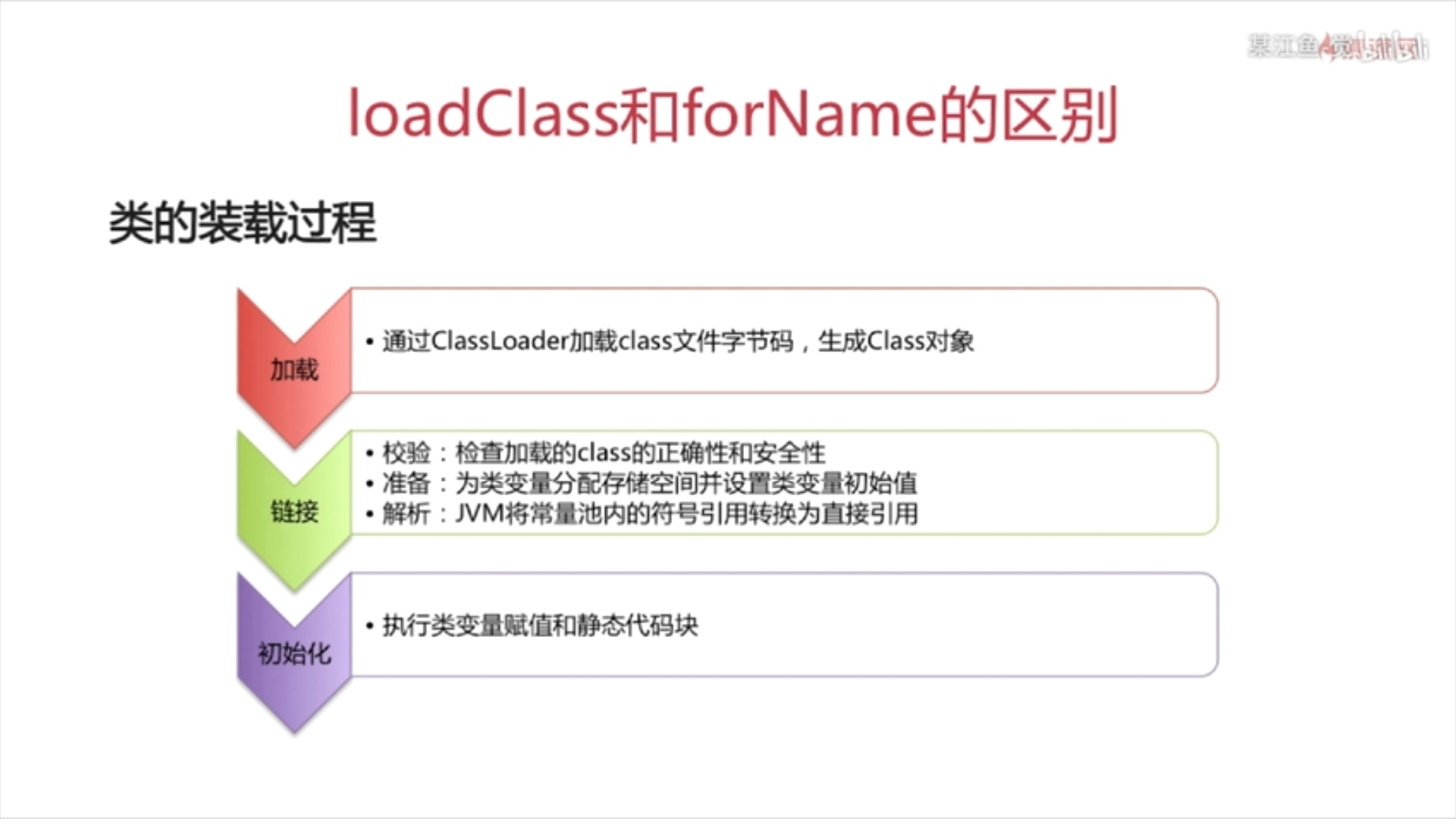
### 为什么要使用双亲委派机制去加载类：

避免多份同样的字节码加载

考虑到安全因素，java核心api中定义类型不会被随意替换

### 类的装载过程：





### 类的加载方式：

隐式加载：new

显示加载：loadClass forName

### ClassLoader.loadClass和Class.forName的区别：

运行时数据区（内存）

## 堆（线程共享）：

Young Generation(1/3堆空间)GC

Old Generation(2/3堆空间)fullGC

MetaData Spacce(直接内存)

### 垃圾回收算法：

引用计数

复制算法：两倍内存空间

标记清除算法：

标记整理算法：

### 垃圾收集器：

对回收算法的具体实现（GC、FULLGC）

新生代：

serial(串行收集器)：gc会暂停所有线程。新生代复制算法

ParNew(并行收集器)：gc会暂停所有线程。新生代复制算法

Parallel Scavenge（并行多线程、系统吞吐量）：降低GC时间

老年代：

Cms（并发收集器）：serial+ParNew

Serial Old（可和年轻代各收集器工作）：老年代标记整理算法

Parallel Old（可和Parallel Scavenge）：老年代标记整理算法

G1：

## java栈：

### 栈帧：

局部变量表+操作数栈+动态链接+方法出口

**本地方法栈**：登记native方法，在Execution Engine执行时加载本地方法库

**方法区**：类的所有字段和方法字节码，以及一些特殊方法（构造函数，接口代码）

静态变量+常量+类信息+运行时常量池

**程序计数器**：指针指向方法区中的方法字节码（由执行引擎读取下一条指令）

**执行引擎**

# 基本数据类型：

byte 1个字节

boolean 1个字节

short 2个字节

char 2个字节

int 4个字节

float 4个字节

long 8个字节

double 8个字节

# javaWEB

## jsp与servlet的区别与联系

1.jsp经编译后就变成了Servlet.

(JSP的本质就是Servlet，JVM只能识别java的类，不能识别JSP的代码,Web容器将JSP的代码编译成JVM能够识别的java类)

2.jsp更擅长表现于页面显示,servlet更擅长于逻辑控制.

3.Servlet中没有内置对象，Jsp中的内置对象都是必须通过HttpServletRequest对象，HttpServletResponse对象以及HttpServlet对象得到.

Jsp是Servlet的一种简化，使用Jsp只需要完成程序员需要输出到客户端的内容，Jsp中的Java脚本如何镶嵌到一个类中，由Jsp容器完成。

而Servlet则是个完整的Java类，这个类的Service方法用于生成对客户端的响应。

## session与cookie的工作原理和区别

### 工作原理

#### Cookie

1、**创建Cookie**

当用户第一次浏览某个使用Cookie的网站时，该网站的服务器就进行如下工作：

①该用户生成一个唯一的识别码（Cookie id），创建一个Cookie对象；

②默认情况下它是一个会话级别的cookie，存储在浏览器的内存中，用户退出浏览器之后被删除。如果网站希望浏览器将该Cookie存储在磁盘上，则需要设置最大时效（maxAge），并给出一个以秒为单位的时间（将最大时效设为0则是命令浏览器删除该Cookie）；

③将Cookie放入到HTTP响应报头，将Cookie插入到一个 Set-Cookie HTTP请求报头中。

④发送该HTTP响应报文。

2、**设置存储Cookie**

浏览器收到该响应报文之后，根据报文头里的Set-Cookied特殊的指示，生成相应的Cookie，保存在客户端。该Cookie里面记录着用户当前的信息。

3、**发送Cookie**

当用户再次访问该网站时，浏览器首先检查所有存储的Cookies，如果某个存在该网站的Cookie（即该Cookie所声明的作用范围大于等于将要请求的资源），则把该cookie附在请求资源的HTTP请求头上发送给服务器。

4、**读取Cookie**

 服务器接收到用户的HTTP请求报文之后，从报文头获取到该用户的Cookie，从里面找到所需要的东西。

（3）作用

Cookie的根本作用就是在客户端存储用户访问网站的一些信息。典型的应用有：

1、记住密码，下次自动登录。

2、购物车功能。

3、记录用户浏览数据，进行商品（广告）推荐。

（4）缺陷

①Cookie会被附加在每个HTTP请求中，所以无形中增加了流量。

②由于在HTTP请求中的Cookie是明文传递的，所以安全性成问题。（除非用HTTPS）

③Cookie的大小限制在4KB左右。对于复杂的存储需求来说是不够用的。

#### Session

1、**创建Session**

当用户访问到一个服务器，如果服务器启用Session，服务器就要为该用户创建一个SESSION，在创建这个SESSION的时候，服务器首先检查这个用户发来的请求里是否包含了一个SESSION ID，如果包含了一个SESSION ID则说明之前该用户已经登陆过并为此用户创建过SESSION，那服务器就按照这个SESSION ID把这个SESSION在服务器的内存中查找出来（如果查找不到，就有可能为他新创建一个），如果客户端请求里不包含有SESSION ID，则为该客户端创建一个SESSION并生成一个与此SESSION相关的SESSION ID。这个SESSION ID是唯一的、不重复的、不容易找到规律的字符串，这个SESSION ID将被在本次响应中返回到客户端保存，而保存这个SESSION ID的正是COOKIE，这样在交互过程中浏览器可以自动的按照规则把这个标识发送给服务器。

2、**使用Session**

我们知道在IE中，我们可以在工具的Internet选项中把Cookie禁止，那么会不会出现把客户端的Cookie禁止了，那么SESSIONID就无法再用了呢？找了一些资料说明，可以有其他机制在COOKIE被禁止时仍然能够把Session id传递回服务器。

经常被使用的一种技术叫做URL重写，就是把Session id直接附加在URL路径的后面一种是作为URL路径的附加信息,表现形式为：

http://…./xxx;jSession=ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng!-145788764；

另一种是作为查询字符串附加在URL后面，表现形式为：

http://…../xxx?jSession=ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng!-145788764

还有一种就是表单隐藏字段。就是服务器会自动修改表单，添加一个隐藏字段，以便在表单提交时能够把Session id传递回服务器。

（3）作用

Session的根本作用就是在服务端存储用户和服务器会话的一些信息。典型的应用有：

1、判断用户是否登录。

2、购物车功能。

### 区别

（1）Cookie以文本文件格式存储在浏览器中，而session存储在服务端它存储了限制数据量。它只允许4kb它没有在cookie中保存多个变量。

（2）cookie的存储限制了数据量，只允许4KB，而session是无限量的

（3）我们可以轻松访问cookie值但是我们无法轻松访问会话值，因此它更安全

（4）设置cookie时间可以使cookie过期。但是使用session-destory（），我们将会销毁会话。

## Sql注入

SQL注入是比较常见的网络攻击方式之一，它不是利用操作系统的BUG来实现攻击，而是针对程序员编程时的疏忽，通过SQL语句，实现无帐号登录，甚至篡改数据库。

如何避免？

1.（简单又有效的方法）PreparedStatement

2.使用正则表达式过滤传入的参数

3.字符串过滤

4.jsp中调用该函数检查是否包函非法字符

5.JSP页面判断代码

## 转发与重定向

转发过程：客户端首先发送一个请求到服务器，服务器匹配Servlet，并指定执行。当这个Servlet执行完后，它要调用getRequestDispacther()方法，把请求转发给指定的Servlet\_list.jsp，整个流程都是在服务端完成的，而且是在同一个请求里面完成的，因此Servlet和jsp共享同一个request，在Servlet里面放的所有东西，在student\_list.jsp中都能取出来。因此，student\_list.jsp能把结果getAttribute()出来，getAttribute()出来后执行完把结果返回给客户端，整个过程是一个请求，一个响应。

重定向过程：客户端发送一个请求到服务器端，服务器匹配Servlet，这都和请求转发一样。Servlet处理完之后调用了sendRedirect()这个方法，这个方法是response方法。所以，当这个Servlet处理完后，看到response.sendRedirect()方法，立即向客户端返回个响应，响应行告诉客户端你必须再重新发送一个请求，去访问student\_list.jsp，紧接着客户端收到这个请求后，立刻发出一个新的请求，去请求student\_list.jsp,在这两个请求互不干扰、相互独立，在前面request里面setAttribute()的任何东西，在后面的request里面都获得不了。因此，在sendRedirect()里面是两个请求，两个响应。

Forward是在服务器端的跳转，就是客户端一个请求给服务器，服务器直接将请求相关参数的信息原封不动的传递到该服务器的其他jsp或Servlet去处理。而sendRedirect()是客户端的跳转，服务器会返回客户端一个响应报头和新的URL地址，原来的参数信息如果服务器没有特殊处理就不存在了，浏览器会访问新的URL所指向的Servlet或jsp，这可能不是原来服务器上的webService了。

# 设计模式：

## 六大原则：

1.单一职责原则

2.里氏替换原则

3.依赖倒转原则

4.接口隔离原则

5.迪米特法则（最少知道原则）

6.合成复用原则

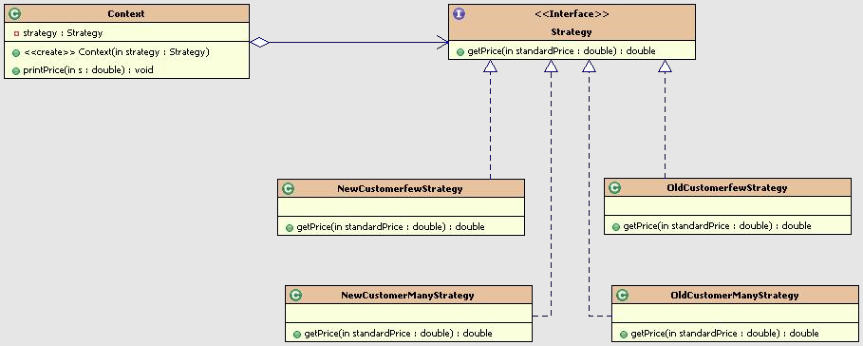
### 工厂模式

### 单例模式

### 代理模式

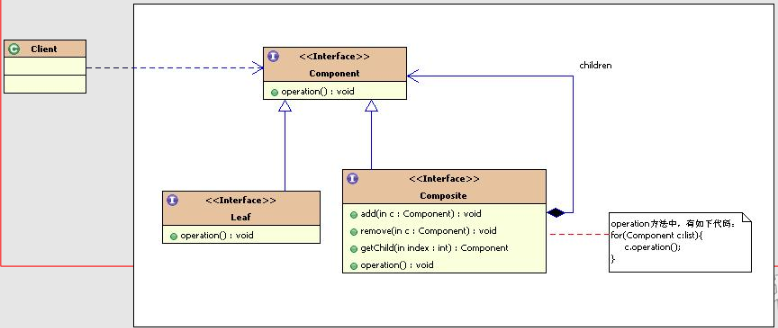
### 策略模式

该模式定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。



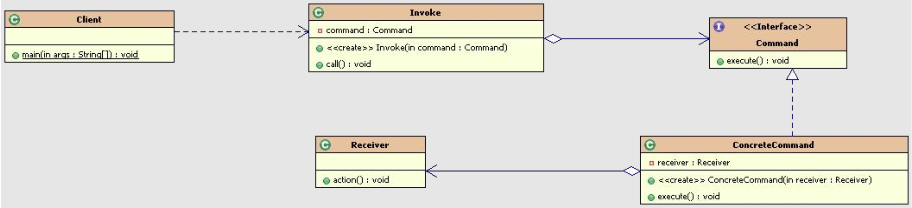
### 组合模式

有时又叫作部分-整体模式，它是一种将对象组合成树状的层次结构的模式，用来表示“部分-整体”的关系，使用户对单个对象和组合对象具有一致的访问性。



### 命令模式

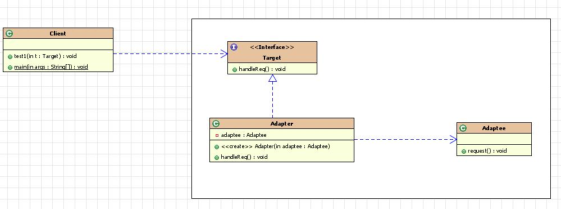
将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。这样两者之间通过命令对象进行沟通，这样方便将命令对象进行储存、传递、调用、增加与管理。



## 结构模式：

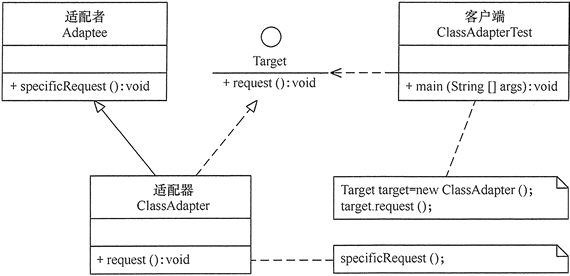
### 适配器模式：

将一个类的方法接口转换成客户希望的另外一个接口。



#### 类的适配器:

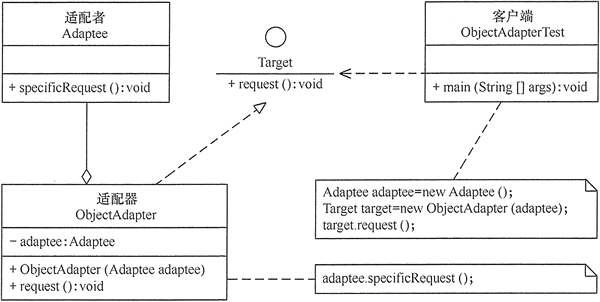
说明：一个接口target（有未实现方法），一个类Adaptee有方法（已实现），最终类ClassAdapter 继承这个接口（并重接口写方法）和这个类，最终最初的接口拥有的方法得到扩展



#### 对象的适配器:

说明：类ObjectAdapter继承targetable接口，并且内部有类成员变量adaptee

最终new target（adaptee）就会有新的扩展方法

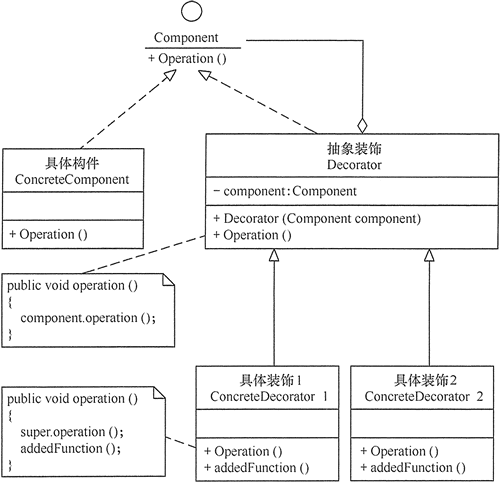


#### 接口的适配器

### 装饰模式：

动态的给一个对象增加新的功能

说明：



### 外观模式：

解决类与类之间的依赖关系，将类与类之间的关系配置到配置文件中

cpu ------->

例子:memory -----> cumputer---->user

disk ------->

桥接模式

### 观察者模式：

当一个对象变化时，其他依赖该对象的对象都会收到通知，并且随着变化

