一、多选题（每题2分,总共20分）

**1、String s = new String("xyz");创建了几个String Object? 答案：A**

A 1 B 2 C 3 D 0

**2、下面关于Map和Hashtable的论述那些是正确的？ 答案：A、B、D**

A Hashtable是HashMap的轻量级实现，他们都完成了Map接口。

B HashMap允许空（null）键值（key）,由于非线程安全，效率上可能高于Hashtable。

C HashMap允许将null作为一个entry的key或者value，而Hashtable不允许。

D HashMap继承自Dictionary类，而Hashtable是Java1.2引进的Map interface的一个实现。

**3、下面关于String 和StringBuffer的论述那些是正确的？B、C**

A String和StringBuffer，它们可以储存和操作字符串，即包含多个字符的字符数据。

B String类提供了数值不可改变的字符串。

C StringBuffer类提供的字符串进行修改。

D String类的字符串可以改变。

**4、要从文件“file.dat”中读取第10个字节到变量c中，下列哪个方法合适 D**

A、FileInputStream in = new FileInputStream("file.dat"); in.skip(9); int c=in.read();

B、FileInputStream in = new FileInputStream("file.dat"); in.skip(10); int c=in.read();

C、FileInputStream in = new FileInputStream("file.dat"); int c=in.read();

D、RandomAccessFile in = new RandomAccessFile("file.dat"); in.skip(9); int c=in.readByte();

**5、一个线程在任何时刻都处于某种状态，例如运行状态、阻塞状态、就绪状态等。一个线程可以由选项中的哪个线程 D**

A、死亡状态

B、阻塞状态（对象lock池内）

C、阻塞状态（对象wait池内）

D、就绪状态

**6、下面是People和Child类的定义和构造方法，每个构造方法都输出编号。在执行new Child("mike")的时候都有哪些构造方法被顺序调用？ 请选择输出结果 D**

class People {

String name;

public People(){System.out.print(1);}

public People(String name){

System.out.print(2);

this.name = name;

}

}

class Child extends People{

People father;

public Child(String name){

System.out.print(3);

this.name = name;

father = new People(name+":F");

}

public Child(){System.out.print(4);}

}

A、312 B、32 C、432 D、132

**7、如果类中的成员变量可以被同一包访问，则使用如下哪个约束符？D**

A、private B、public C、protected D、缺省 E、final

**8、关于异常(Exception)，下列描述正确的是 ABC**

A. 异常的基类为Exception，所有异常都必须直接或者间接继承它

B. 异常可以用try{ . . .}catch(Exception e){ . . .}来捕获并进行处理

C. 如果某异常继承RuntimeException，则该异常可以不被声明

D. 异常可以随便处理，而不是抛给外层的程序进行处理

**9、下面的代码实现了设计模式中的什么模式 C**

public class A {

private A instance; private A() { }

public static A getInstance { if ( A == null ) instance = new A(); return instance; }

}

A. Factory B. Abstract Factory C. Singleton D. Builder

**10、创建一个TCP客户程序的顺序是：（DACBE ）**  
A、获得I/O流  
B、关闭I/O流  
C、对I/O流进行读写操作  
D、建立socket  
E、关闭socket

二、问答题（每题5分，总共30分）

1.**Enumeration和Iterator接口的区别？**

    Enumeration的速度是Iterator的两倍，也使用更少的内存。Enumeration是非常基础的，也满足了基础的需要。但是，与Enumeration相比，Iterator更加安全，因为当一个集合正在被遍历的时候，它会阻止其它线程去修改集合。

迭代器取代了Java集合框架中的Enumeration。迭代器允许调用者从集合中移除元素，而Enumeration不能做到。为了使它的功能更加清晰，迭代器方法名已经经过改善。

2. **Array和ArrayList有何区别？什么时候更适合用Array？**

  Array可以容纳基本类型和对象，而ArrayList只能容纳对象。

  Array是指定大小的，而ArrayList大小是固定的。

    Array没有提供ArrayList那么多功能，比如addAll、removeAll和iterator等。尽管ArrayList明显是更好的选择，但也有些时候Array比较好用。

（1）如果列表的大小已经指定，大部分情况下是存储和遍历它们。

（2）对于遍历基本数据类型，尽管Collections使用自动装箱来减轻编码任务，在指定大小的基本类型的列表上工作也会变得很慢。

（3）如果你要使用多维数组，使用[][]比List<List<>>更容易。

**3. java线程有哪些状态并画图表示状态之间的关系**

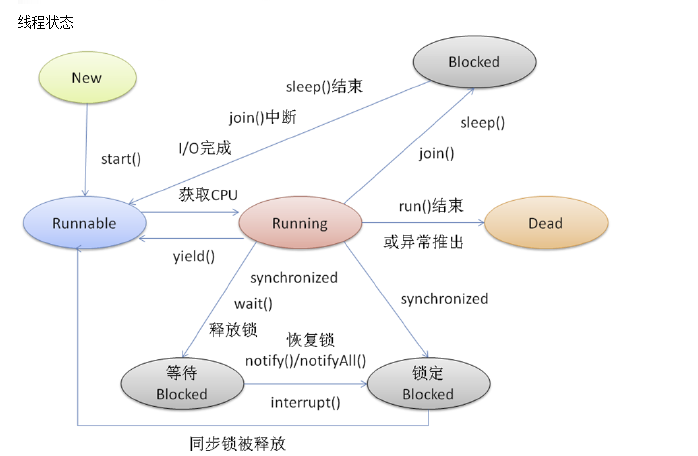
创建（new）状态: 准备好了一个多线程的对象

就绪（runnable）状态: 调用了start()方法, 等待CPU进行调度

运行（running）状态: 执行run()方法

阻塞（blocked）状态: 暂时停止执行, 可能将资源交给其它线程使用

终止（dead）状态: 线程销毁



**4、redis有几种数据结构？ 试写出每种结构的插入、读取、删除命令**

Redis可以存储键与5种不同数据结构类型之间的映射，下面的表格介绍了它们的语义

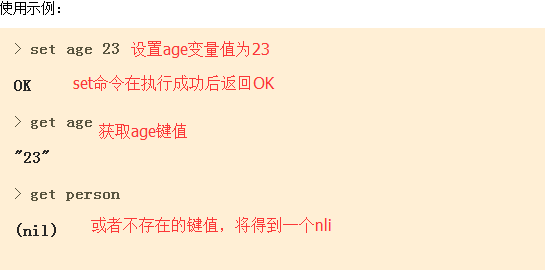
| **结构类型** | **结构存储的值** | **结构的读写能力** |
| --- | --- | --- |
| STRING | 可以是字符串、整数或者浮点数 | 对整个字符串或者字符串的其中一部分执行操作；对整数和浮点数执行自增（increment）或者自减（decrement）操作 |
| LIST | 一个链表，链表上的每个节点都包含了一个字符串 | 从链表的两端推入或者弹出元素；根据偏移量对链表进行修剪（trim）；读取单个或者多个元素；根据值查找或者移除元素 |
| SET | 包含字符串的无序收集器（unordered collection），并且被包含的每个字符串都是独一无二、各不相同的 | 添加、获取、移除单个元素；检查一个元素是否存在于集合中；计算交集、并集、差集；从集合里面随机获取元素 |
| HASH | 包含键值对的无序散列表 | 添加、获取、移除单个键值对；获取所有键值对 |
| ZSET（有序集合） | 字符串成员（member）与浮点数分值（score）之间的有序映射，元素的排列顺序由分值的大小决定 | 添加、获取、删除单个元素；根据分值范围（range）或者成员来获取元素 |

**命令：**

**1、String字符串**

Redis的String和其他编程语言或者其它键值存储提供的字符串非常相似。下图描述了字符串常用的三个命令：

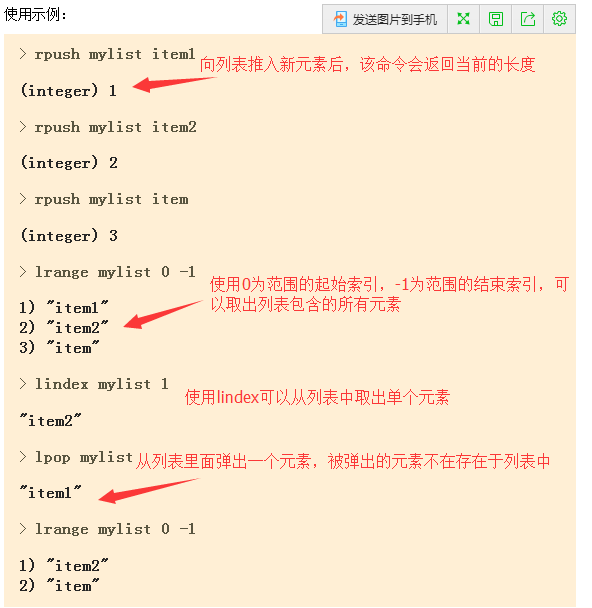
| **命令** | **行为** |
| --- | --- |
| GET | 获取存储在给定键中的值 |
| SET | 设置存储在给定键中的值 |
| DEL | 删除存储在给定键中的值（这个命令可以用于所有类型） |



**2、List列表**

一个列表结构可以有序的存储多个字符串。list常用的命令如下表所示：、

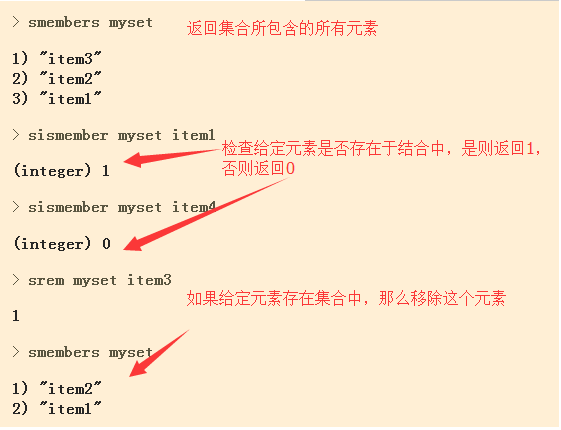
| **命令** | **行为** |
| --- | --- |
| RPUSH | 将给定值推入列表的右端 |
| LRANGE | 获取列表在给定范围上的所有值 |
| LINDEX | 获取列表在给定位置上的单个元素 |
| LPOP | 从列表的左端弹出一个值，并返回被弹出的值 |



**3、set集合**

　　Redis的集合和列表都可以存储多个字符串，它们的不同之处在于，列表可以存储多个相同的字符串，而集合则通过使用散列表来保证自己存储的每个字符串都是各不相同的。set常用的命令如下表所示：

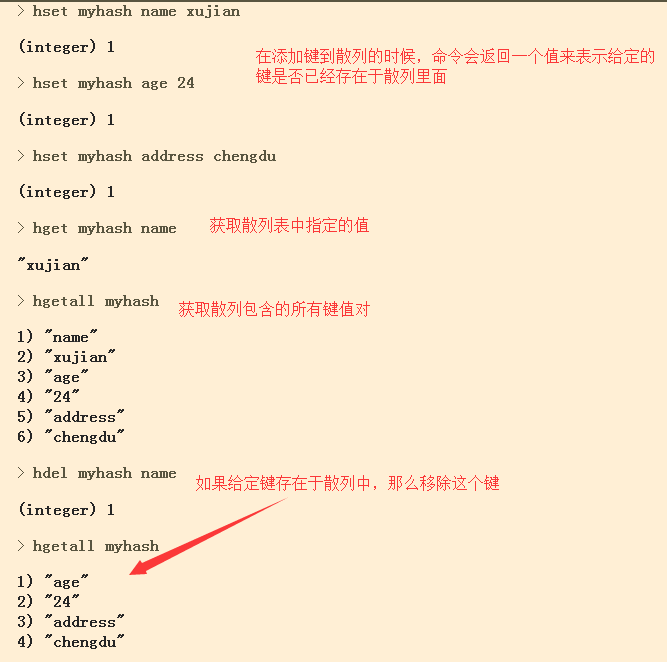
| **命令** | **行为** |
| --- | --- |
| SADD | 将给定元素添加到集合 |
| SMEMBERS | 返回集合包含的所有元素 |
| SISMEMBER | 检查给定元素是否存在于集合中 |
| SREM | 如果给定的元素存在于集合中，那么移除这个元素 |



#### 4、Hash散列

　　Redis的散列可以存储多个键值对之间的映射，和字符串一样，散列存储的值既可以是字符串又可以是数字值，并且可以对散列存储的数字值执行自增和自减操作。散列常用的命令如下表所示:

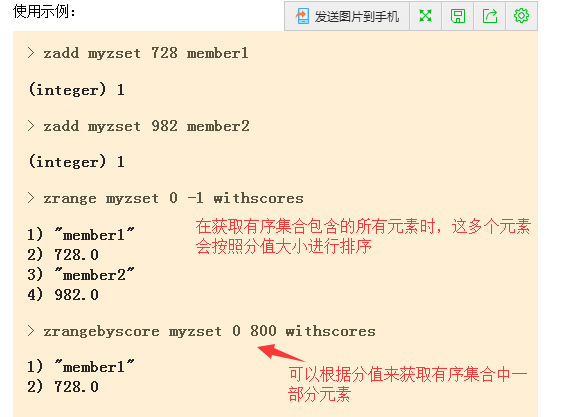
| **命令** | **行为** |
| --- | --- |
| HSET | 在散列里面关联起给定的键值对 |
| HGET | 获取指定散列键的值 |
| HGETALL | 获取散列包含的所有键值对 |
| HDEL | 如果给定键存在于散列里面，那么移除这个键 |



#### 5、zset有序集合

　　有序集合和散列一样，都用于存储键值对：有序集合的键被称为成员，每个成员都是独一无二的，而有序集合的则被称为分值，分值必须为浮点数。有序集合是Redis里面唯一一个既可以根据成员访问元素（这一点和散列一样），又可以根据分值以及分值的排列顺序来访问元素的结构。有序集合的常用命令如下表所示：

| **命令** | **行为** |
| --- | --- |
| ZADD | 将一个带有给定分值的成员添加到有序集合里面 |
| ZRANGE | 根据元素在有序排列中所处的位置，从有序集合里面获取多个元素 |
| ZRANGEBYSCORE | 获取有序集合在给定分值范围内的所有元素 |
| ZREM | 如果给定成员存在于有序集合，那么移除这个成员 |



**5、Java NIO 由哪个几个核心部分组成？各个核心部分的作用是什么？**

Java NIO 由Channels、Buffers、Selectors核心部分组成。

**Channel:**

Java NIO的通道类似流，但又有些不同：

既可以从通道中读取数据，又可以写数据到通道。但流的读写通常是单向的。

通道可以异步地读写。

通道中的数据总是要先读到一个Buffer，或者总是要从一个Buffer中写入。

这些是Java NIO中最重要的通道的实现：

FileChannel DatagramChannel SocketChannel ServerSocketChannel

FileChannel 从文件中读写数据。

DatagramChannel 能通过UDP读写网络中的数据。

SocketChannel 能通过TCP读写网络中的数据。

ServerSocketChannel可以监听新进来的TCP连接，像Web服务器那样。对每一个新进来的连接都会创建一个SocketChannel。

**Buffer:**

Buffer用于和NIO通道进行交互。数据是从通道读入缓冲区，从缓冲区写入到通道中的。

Buffer读写数据一般遵循以下四个步骤：

1、写入数据到Buffer 2、调用flip()方法

3、从Buffer中读取数据 4、调用clear()方法或者compact()方法

缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成NIO Buffer对象，并提供了一组方法，用来方便的访问该块内存。

为了理解Buffer的工作原理，需要熟悉它的三个属性：

capacity、position、limit

**capacity**

作为一个内存块，Buffer有一个固定的大小值，也叫“capacity”.你只能往里写capacity个byte、long，char等类型。一旦Buffer满了，需要将其清空（通过读数据或者清除数据）才能继续写数据往里写数据。

**position**

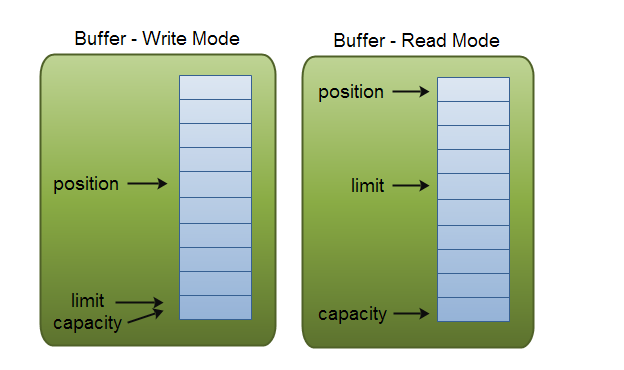
当你写数据到Buffer中时，position表示当前的位置。初始的position值为0.当一个byte、long等数据写到Buffer后， position会向前移动到下一个可插入数据的Buffer单元。position最大可为capacity – 1.

当读取数据时，也是从某个特定位置读。当将Buffer从写模式切换到读模式，position会被重置为0. 当从Buffer的position处读取数据时，position向前移动到下一个可读的位置。

**limit**

在写模式下，Buffer的limit表示你最多能往Buffer里写多少数据。 写模式下，limit等于Buffer的capacity。

当切换Buffer到读模式时， limit表示你最多能读到多少数据。因此，当切换Buffer到读模式时，limit会被设置成写模式下的position值。换句话说，你能读到之前写入的所有数据（limit被设置成已写数据的数量，这个值在写模式下就是position）



**Selector:**

Selector（选择器）是Java NIO中能够检测一到多个NIO通道，并能够知晓通道是否为诸如读写事件做好准备的组件。这样，一个单独的线程可以管理多个channel，从而管理多个网络连接。

Selector的创建

通过调用Selector.open()方法创建一个Selector，如下：

|  |  |
| --- | --- |
| 1 | Selector selector = Selector.open(); |

向Selector注册通道

为了将Channel和Selector配合使用，必须将channel注册到selector上。通过SelectableChannel.register()方法来实现，如下：

|  |  |
| --- | --- |
| 1 | channel.configureBlocking(false); |
| 2 | SelectionKey key = channel.register(selector, | | Selectionkey.OP\_READ); |

|  |  |
| --- | --- |
|  |  |

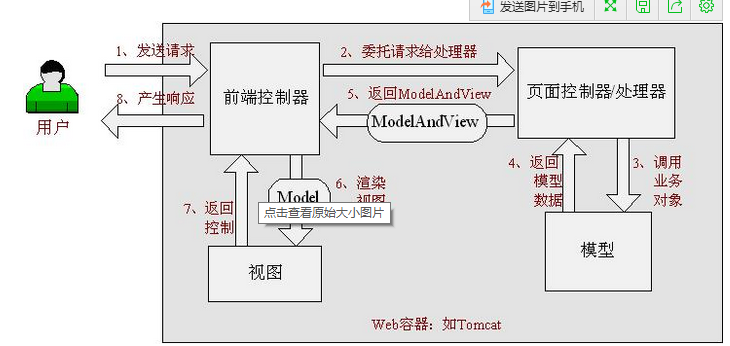
与Selector一起使用时，Channel必须处于非阻塞模式下。这意味着不能将FileChannel与Selector一起使用，因为FileChannel不能切换到非阻塞模式。而套接字通道都可以。

注意register()方法的第二个参数。这是一个“interest集合”，意思是在通过Selector监听Channel时对什么事件感兴趣。可以监听四种不同类型的事件：Connect Accept Read Write

**三、简答题 （**第一题20分, 第二、三题各15分，总共50分**）**

1.spring框架已经成为每个开发人员必备的技能。请阐述以下几个问题

a.spring web mvc处理流程以及如何使用springmvc开发



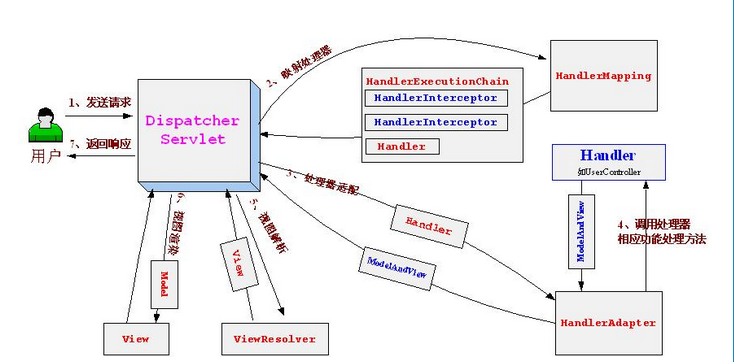
具体执行步骤如下：

1、  首先用户发送请求————>前端控制器，前端控制器根据请求信息（如URL）来决定选择哪一个页面控制器进行处理并把请求委托给它，即以前的控制器的控制逻辑部分；图2-1中的1、2步骤；

2、  页面控制器接收到请求后，进行功能处理，首先需要收集和绑定请求参数到一个对象，这个对象在Spring Web MVC中叫命令对象，并进行验证，然后将命令对象委托给业务对象进行处理；处理完毕后返回一个ModelAndView（模型数据和逻辑视图名）；图2-1中的3、4、5步骤；

3、  前端控制器收回控制权，然后根据返回的逻辑视图名，选择相应的视图进行渲染，并把模型数据传入以便视图渲染；图2-1中的步骤6、7；

4、  前端控制器再次收回控制权，将响应返回给用户，至此整个结束。



1、  首先用户发送请求——>DispatcherServlet，前端控制器收到请求后自己不进行处理，而是委托给其他的解析器进行处理，作为统一访问点，进行全局的流程控制；

2、  DispatcherServlet——>HandlerMapping， HandlerMapping将会把请求映射为HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象、多个HandlerInterceptor拦截器）对象，通过这种策略模式，很容易添加新的映射策略；

3、  DispatcherServlet——>HandlerAdapter，HandlerAdapter将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器；

4、  HandlerAdapter——>处理器功能处理方法的调用，HandlerAdapter将会根据适配的结果调用真正的处理器的功能处理方法，完成功能处理；并返回一个ModelAndView对象（包含模型数据、逻辑视图名）；

5、  ModelAndView的逻辑视图名——> ViewResolver， ViewResolver将把逻辑视图名解析为具体的View，通过这种策略模式，很容易更换其他视图技术；

6、  View——>渲染，View会根据传进来的Model模型数据进行渲染，此处的Model实际是一个Map数据结构，因此很容易支持其他视图技术；

7、返回控制权给DispatcherServlet，由DispatcherServlet返回响应给用户，到此一个流程结束。

**开发：**

**<servlet>**

**<servlet-name>chapter2</servlet-name>**

**<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>**

**<load-on-startup>1</load-on-startup>**

**</servlet>**

**<servlet-mapping>**

**<servlet-name>chapter2</servlet-name>**

**<url-pattern>/</url-pattern>**

**</servlet-mapping>**

**<!-- HandlerMapping -->**

**<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>**

**<!-- HandlerAdapter -->**

**<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>**

**<!-- ViewResolver -->**

**<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">**

**<property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>**

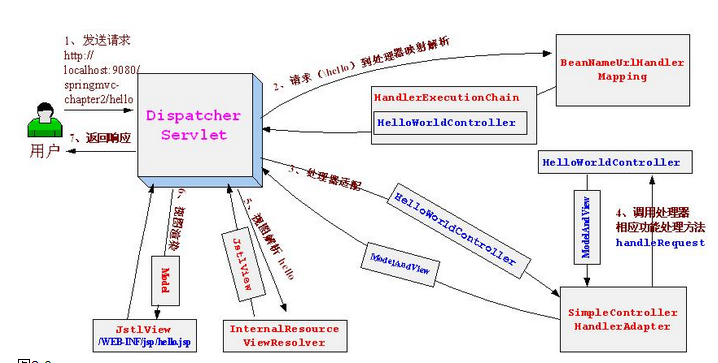
**<property name="prefix" value="/WEB-INF/jsp/"/>**

**<property name="suffix" value=".jsp"/>**

**</bean>**

**<!-- 处理器 -->**

**<bean name="/hello" class="cn.javass.chapter2.web.controller.HelloWorldController"/>**



运行步骤：

1、  首先用户发送请求[http://localhost:9080/springmvc-chapter2/hello——>web](http://localhost:9080/springmvc-chapter2/hello%E2%80%94%E2%80%94%3eweb)容器，web容器根据“/hello”路径映射到DispatcherServlet（url-pattern为/）进行处理；

2、  DispatcherServlet——>BeanNameUrlHandlerMapping进行请求到处理的映射，BeanNameUrlHandlerMapping将“/hello”路径直接映射到名字为“/hello”的Bean进行处理，即HelloWorldController，BeanNameUrlHandlerMapping将其包装为HandlerExecutionChain（只包括HelloWorldController处理器，没有拦截器）；

3、  DispatcherServlet——> SimpleControllerHandlerAdapter，SimpleControllerHandlerAdapter将HandlerExecutionChain中的处理器（HelloWorldController）适配为SimpleControllerHandlerAdapter；

4、  SimpleControllerHandlerAdapter——> HelloWorldController处理器功能处理方法的调用，SimpleControllerHandlerAdapter将会调用处理器的handleRequest方法进行功能处理，该处理方法返回一个ModelAndView给DispatcherServlet；

5、  hello（ModelAndView的逻辑视图名）——>InternalResourceViewResolver， InternalResourceViewResolver使用JstlView，具体视图页面在/WEB-INF/jsp/hello.jsp；

6、  JstlView（/WEB-INF/jsp/hello.jsp）——>渲染，将在处理器传入的模型数据(message=HelloWorld！)在视图中展示出来；

7、  返回控制权给DispatcherServlet，由DispatcherServlet返回响应给用户，到此一个流程结束。

**b.简述spring aop的通知类型及常用开发方式**

实现AOP的技术，主要分为两大类：一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

* **前置通知（Before Advice）:**在切入点选择的连接点处的方法之前执行的通知，该通知不影响正常程序执行流程（除非该通知抛出异常，该异常将中断当前方法链的执行而返回）。
* **后置通知（After Advice）:**在切入点选择的连接点处的方法之后执行的通知，包括如下类型的后置通知：
  + **后置返回通知（After returning Advice）:**在切入点选择的连接点处的方法正常执行完毕时执行的通知，必须是连接点处的方法没抛出任何异常正常返回时才调用后置通知。
  + **后置异常通知（After throwing Advice）:** 在切入点选择的连接点处的方法抛出异常返回时执行的通知，必须是连接点处的方法抛出任何异常返回时才调用异常通知。
  + **后置最终通知（After finally Advice）:** 在切入点选择的连接点处的方法返回时执行的通知，不管抛没抛出异常都执行，类似于Java中的finally块。
* **环绕通知（Around Advices）：**环绕着在切入点选择的连接点处的方法所执行的通知，环绕通知可以在方法调用之前和之后自定义任何行为，并且可以决定是否执行连接点处的方法、替换返回值、抛出异常等等。

开发方式：

1.配置ProxyFactoryBean，显式地设置advisors, advice, target等

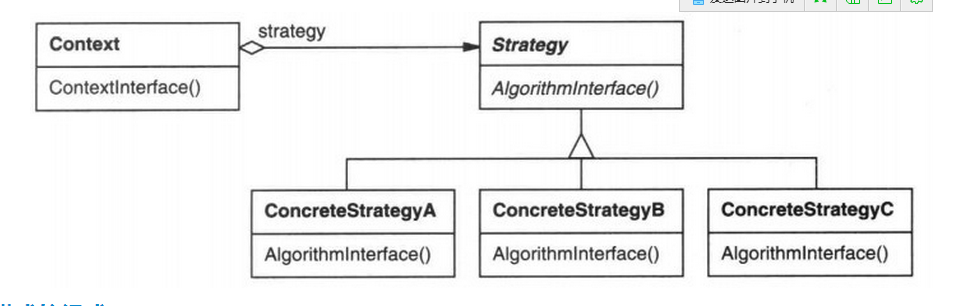
2.配置AutoProxyCreator，这种方式下，还是如以前一样使用定义的bean，但是从容器中获得的其实已经是代理对象

3.通过<aop:config>来配置

4.通过<aop: aspectj-autoproxy>来配置，使用AspectJ的注解来标识通知及切入点

**2.阐述策略模式、代理模式、观察者模式并画出UML图**

**策略模式：**

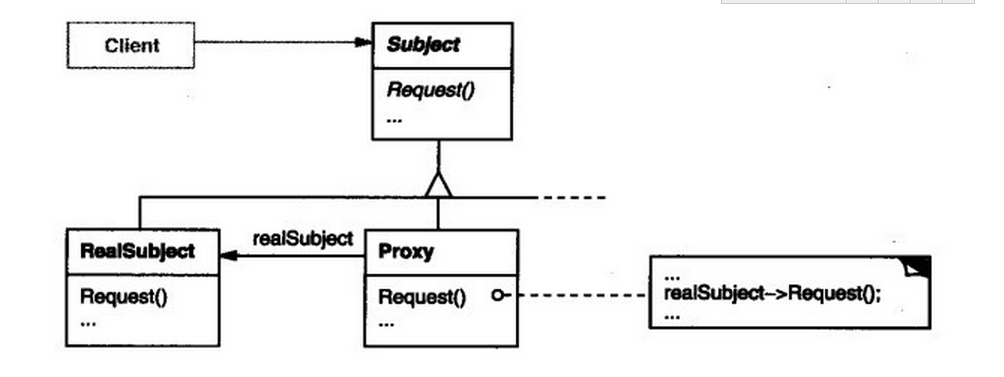


1）[策略模式](http://blog.csdn.net/hguisu/article/details/7558249)是一个比较容易理解和使用的设计模式，策略模式是对算法的封装，它把算法的责任和算法本身分割开，委派给不同的对象管理。策略模式通常把一个系列的算法封装到一系列的策略类里面，作为一个抽象策略类的子类。用一句话来说，就是“准备一组算法，并将每一个算法封装起来，使得它们可以互换”。

2）在[策略模式](http://blog.csdn.net/hguisu/article/details/7558249" \t "_blank)中，应当由客户端自己决定在什么情况下使用什么具体策略角色。

3）[策略模式](http://blog.csdn.net/hguisu/article/details/7558249)仅仅封装算法，提供新算法插入到已有系统中，以及老算法从系统中“退休”的方便，策略模式并不决定在何时使用何种算法，算法的选择由客户端来决定。这在一定程度上提高了系统的灵活性，但是客户端需要理解所有具体策略类之间的区别，以便选择合适的算法，这也是策略模式的缺点之一，在一定程度上增加了客户端的使用难度。

**代理模式：**



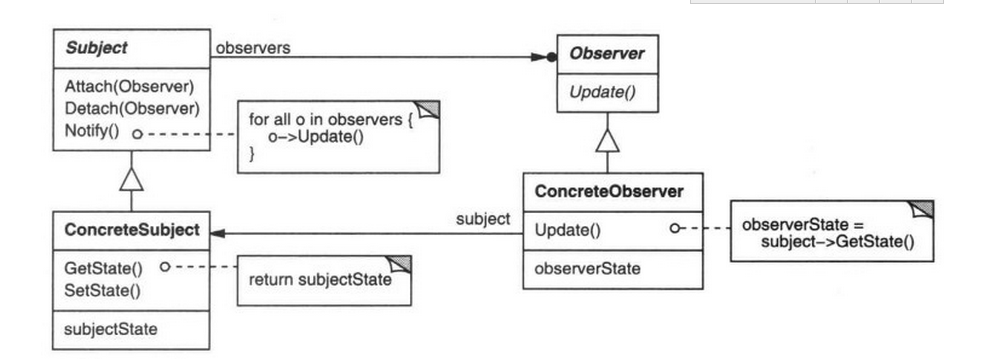
代理模式在很多情况下都非常有用,特别是你想强行控制一个对象的时候,比如:延迟加载,监视状态变更的方法等等

1、“增加一层间接层”是软件系统中对许多负责问题的一种常见解决方法。在面向对象系统中，直接使用某些对象会带来很多问题，作为间接层的proxy对象便是解决这一问题的常用手段。

2、具体proxy设计模式的实现方法、实现粒度都相差很大，有些可能对单个对象作细粒度的控制，有些可能对组件模块提供抽象代理层，在**[架构](http://lib.csdn.net/base/architecture" \o "大型网站架构知识库" \t "_blank)**层次对对象作proxy。

3、proxy并不一定要求保持接口的一致性，只要能够实现间接控制，有时候损及一些透明性是可以接受的。例如上面的那个例子，代理类型ProxyClass

**观察者模式**



目标（Subject）: 目标知道它的观察者。可以有任意多个观察者观察同一个目标。 提供注册和删除观察者对象的接口。  
具体目标（ConcreteSubject）:  将有关状态存入各ConcreteObserver对象。  
观察者(Observer):  为那些在目标发生改变时需获得通知的对象定义一个更新接口。当它的状态发生改变时, 向它的各个观察者发出通知。  
具体观察者(ConcreteObserver):   维护一个指向ConcreteSubject对象的引用。存储有关状态，这些状态应与目标的状态保持一致。实现O b s e r v e r的更新接口以使自身状态与目标的状态保持一致。

**3.用JAVA程序写出冒泡算法、快速排序算法并比较两种算法的性能**

冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

**/\*\***

**\* 冒泡排序**

**\* 比较相邻的元素。如果第一个比第二个大，就交换他们两个。**

**\* 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。**

**\* 针对所有的元素重复以上的步骤，除了最后一个。**

**\* 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。**

**\* @param numbers 需要排序的整型数组**

**\*/**

时间复杂度是O（n\*n），空间复杂度是O（1）

**public static void bubbleSort(int[] numbers)**

**{**

**int temp = 0;**

**int size = numbers.length;**

**for(int i = 0 ; i < size-1; i ++)**

**{**

**for(int j = 0 ;j < size-1-i ; j++)**

**{**

**if(numbers[j] > numbers[j+1]) //交换两数位置**

**{**

**temp = numbers[j];**

**numbers[j] = numbers[j+1];**

**numbers[j+1] = temp;**

**}**

**}**

**}**

**}**

基于分治的思想，是冒泡排序的改进型。首先在数组中选择一个基准点（该基准点的选取可能影响快速排序的效率，后面讲解选取的方法），然后分别从数组的两端扫描数组，设两个指示标志（lo指向起始位置，hi指向末尾)，首先从后半部分开始，如果发现有元素比该基准点的值小，就交换lo和hi位置的值，然后从前半部分开始扫秒，发现有元素大于基准点的值，就交换lo和hi位置的值，如此往复循环，直到lo>=hi,然后把基准点的值放到hi这个位置。一次排序就完成了。以后采用递归的方式分别对前半部分和后半部分排序，当前半部分和后半部分均有序时该数组就自然有序了。

平均时间复杂度是 O(Nlog2N)，最差也是O(N\*N)，空间复杂度O(Nlog2N)

**public static int partition(int []array,int lo,int hi){**

**//固定的切分方式**

**int key=array[lo];**

**while(lo<hi){**

**while(array[hi]>=key&&hi>lo){//从后半部分向前扫描**

**hi--;**

**}**

**array[lo]=array[hi];**

**while(array[lo]<=key&&hi>lo){从前半部分向后扫描**

**lo++;**

**}**

**array[hi]=array[lo];**

**}**

**array[hi]=key;**

**return hi;**

**}**

**public static void sort(int[] array,int lo ,int hi){**

**if(lo>=hi){**

**return ;**

**}**

**int index=partition(array,lo,hi);**

**sort(array,lo,index-1);**

**sort(array,index+1,hi);**

**}**

**4设计一个系统：**

**xxx纯净水生产线**

**目前流程是：从某个地方把水取出来，然后经过缓冲，过滤，加热和放糖的步骤**

答案：

abstract 水{

public void 水();

}

interface 过滤{}

interface 缓冲{}

interface 加热{}

interface 放糖{}

class 纯净水1 extends 水 imps 过滤,缓冲{}

class 纯净水2 extends 水 imps 缓冲{}

class 纯净水2 extends 水 imps 过滤{}