解读Tomcat的bat文件

学习bat文件中的一些基本命令

bat文件又被称为批处理文件，在DOS和Windows（任意）系统中，bat文件是可执行文件，由一系列命令构成，其中可以包含对其他程序的调用。这个文件的每一行都是一条DOS命令（大部分时候就好像我们在DOS提示符下执行的命令行一样）。

常见的命令：

1. @ ：放在一条命令前面，表示关闭这个命令的回显。不管是否关闭命令回显，命令的运行结果会打印在黑窗口里面。
2. echo：重复，回声。表示打印内容。
3. echo off:表示关闭从该命令以下的所有命令的回显。@echo off表示这个命令也不回显。一般bat文件的第一行都会使用@echo off
4. rem：表示注释，或者使用双冒号
5. pause:表示暂停批处理文件的执行，可以通过点击任意键继续。pause>nul表示不显示文字，并且不关闭黑窗口。
6. call :可以调用另一个批处理文件中的命令。如果不使用这个关键字，另一个文件执行完后，不会继续运行本文件里面的指令。
7. %环境变量%:可以使用这个符号或者电脑里面设置的环境变量的值。
8. setlocal和endlocal：设置一个局部变量，该变量只在这个局部范围内有效。
9. 在执行批处理文件时，我们可以在文件名后面跟参数。test3.bat zhangsan lisi

   zhangsan 和lisi就是test3文件时传递的参数。我们可以通过%1获取第一个参数的值，%2获取第二个参数的值。

   > 注意
   >
   > 如果调用bat时 某一个参数包含空格，那么需要调用者将参数放在双引号中

10. if判断

    ```
    if not 判断条件（
        echo  操作
    ） else （
        echo 操作
    ）
    ```

    如果if判断跟操作在一行上可以省略小括号，否则不能省略。判断条件中只能使用==判断，不能使用大于或者小于

11. goto用法

```
@echo off
set name="Tom"
if %name%=="Tom" goto abc
:Jack
echo jack
:abc
echo Tom
goto end
:Jim
echo Jim
:end
pause
```

从1打印到100

```
@echo off
set i=1
:xunhuan
if %i%==101 goto end
echo %i%
set /a i=%i%+1
goto xunhuan
:end
pause
```

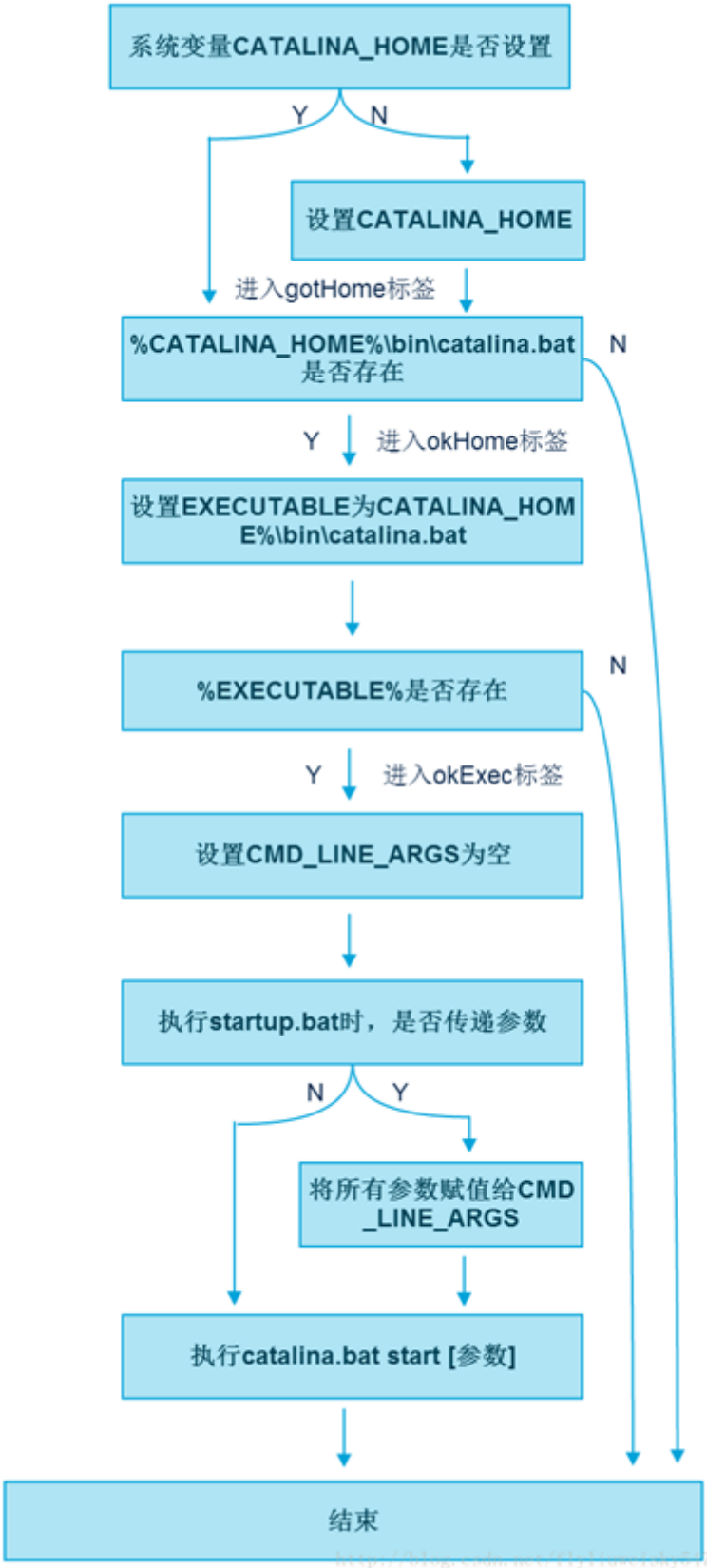
/a 命令行开关指定等号右边的字符串为被评估的数字表达式。

12. start “Tomcat” ping www.baidu.com 新开启一个窗口，窗口名字叫Tomcat，执行ping命令
13. shift：更改批处理文件参数的位置。
14. 替换变量中的某个字符。

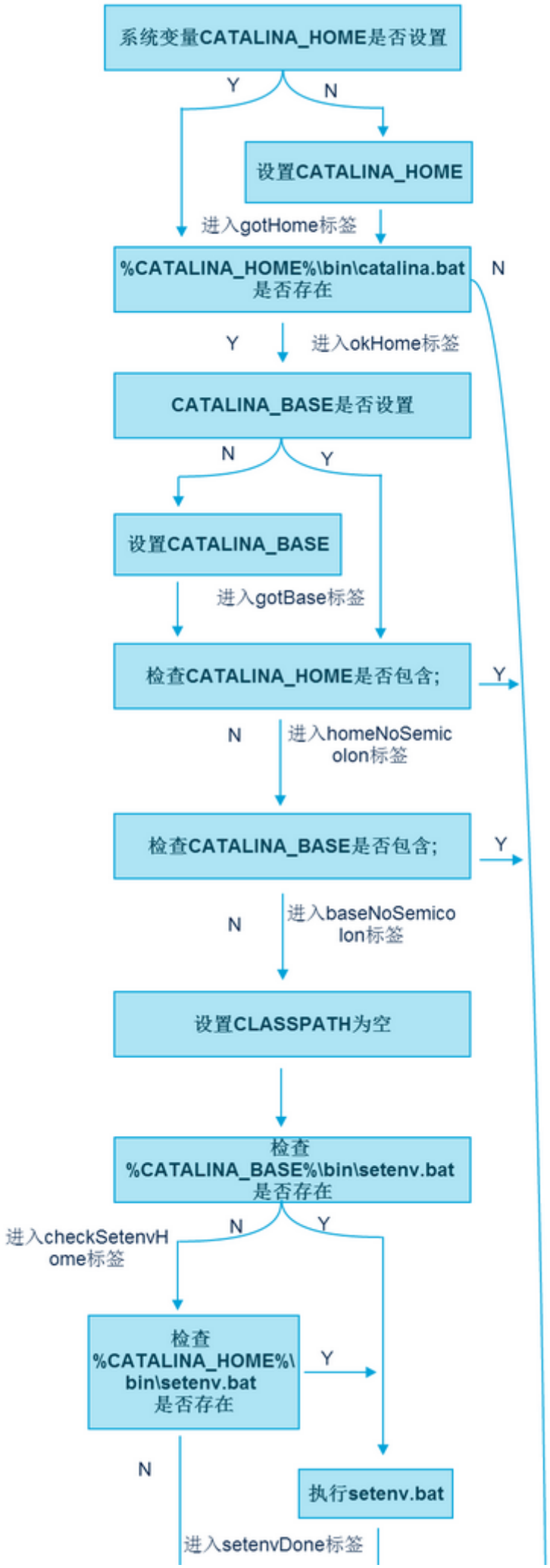
    将%JAVA_HOME%中的a替换成hello：%JAVA_HOME:a=hello% catalina.bat中的%CATALINA_HOME:;=%，是将%CATALINA_HOME%的;删除 catalina.bat中通过"%CATALINA_HOME%" == "%CATALINA_HOME:;=%"，检查CATALINA_HOME中是否包含;
15. 文件操作

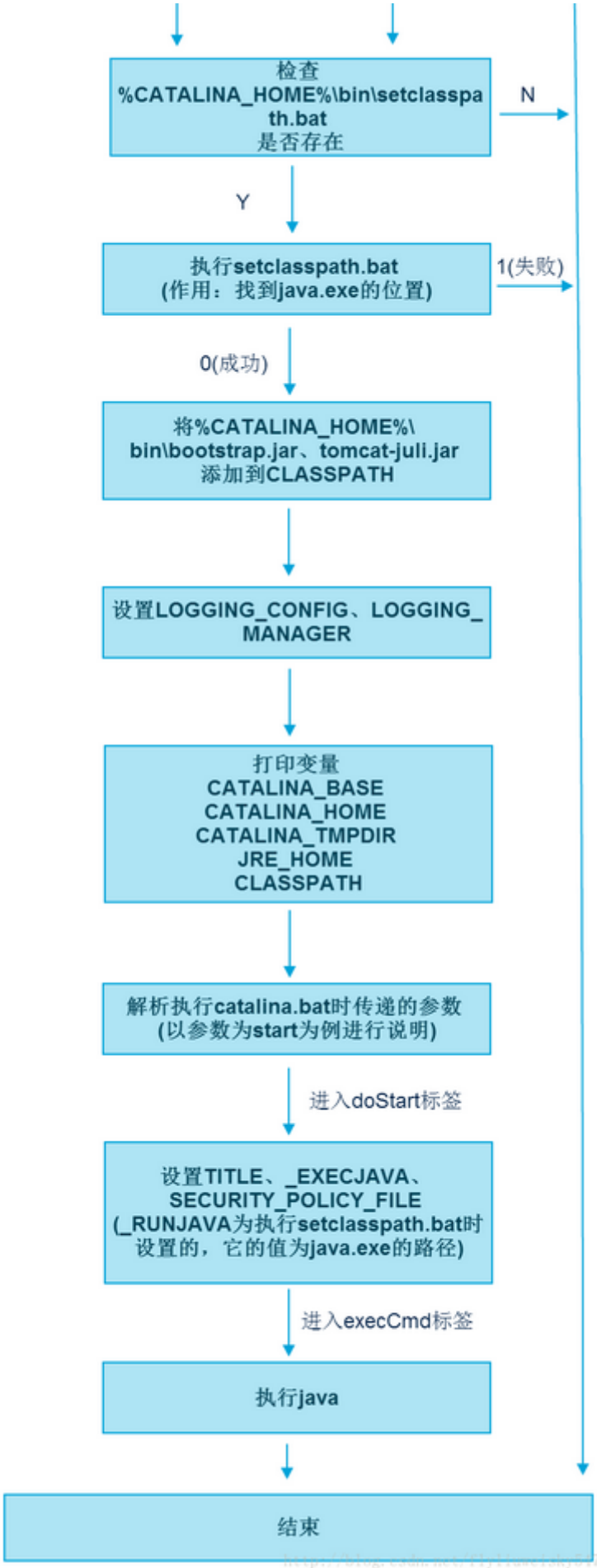
```
md或者mkdir创建文件夹
del /Q a.txt 删除文件 q表示不需要确认
rd a 删除文件夹
echo hello>test.txt 新建一个text.txt文件，内容为hello
more text.txt 把test.txt的内容展示出来
```

startup.bat的运行流程：



Catalina.bat的运行过程

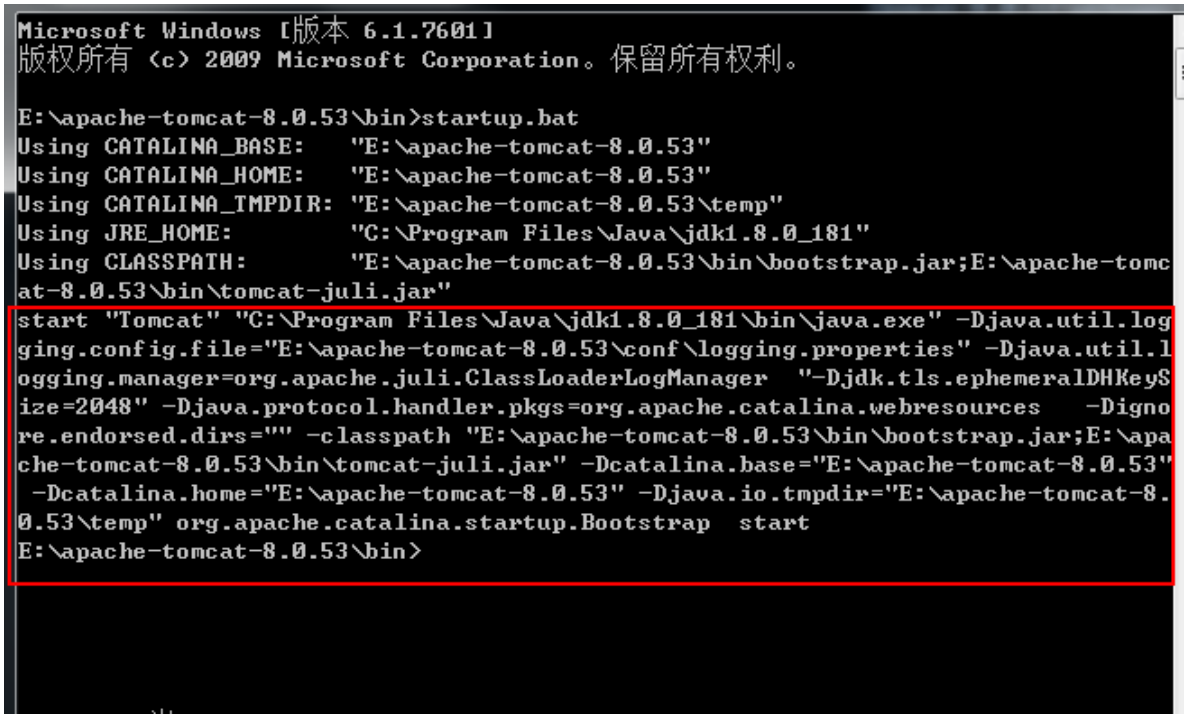

系统变量CATALINA_HOME是否设置
Y
N
设置CATALINA_HOME
进入gotHome标签
%CATALINA_HOME%\bin\catalina.bat
是否存在
N
Y
进入okHome标签
CATALINA_BASE是否设置
N
Y
设置CATALINA_BASE
进入gotBase标签
检查CATALINA_HOME是否包含;
Y
N
进入homeNoSemicolon标签
检查CATALINA_BASE是否包含;
Y
N
进入baseNoSemicolon标签
设置CLASSPATH为空
检查
%CATALINA_BASE%\bin\setenv.bat
是否存在
进入checkSetenvHome标签
N
Y
检查
%CATALINA_HOME%\bin\setenv.bat
是否存在
Y
N
执行setenv.bat
进入setenvDone标签



在catalina.bat的:end下方加入一行代码

```
:end
echo %_EXECJAVA% %LOGGING_CONFIG% %LOGGING_MANAGER% %JAVA_OPTS% %CATALINA_OPTS%
%DEBUG_OPTS% -D%ENDORSED_PROP%="%JAVA_ENDORSED_DIRS%" -classpath "%CLASSPATH%" -
Dcatalina.base="%CATALINA_BASE%" -Dcatalina.home="%CATALINA_HOME%" -
Djava.io.tmpdir="%CATALINA_TMPDIR%" %MAINCLASS% %CMD_LINE_ARGS% %ACTION%
```

在cmd下执行startup.bat，执行结果：



红框上方是默认输出；红框内是新添加的输出

将红框内的内容排一下版

```
start "Tomcat" "C:\Program Files\Java\jdk1.8.0_181\bin\java.exe"
-Djava.util.logging.config.file="E:\apache-Tomcat-
8.0.53\conf\logging.properties"
-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
"-Djdk.tls.ephemeralDHKeySize=2048"
-Djava.protocol.handler.pkgs=org.apache.catalina.webresources
-Dignore.endorsed.dirs=""
-classpath "E:\apache-Tomcat-8.0.53\bin\bootstrap.jar;E:\apache-Tomcat-
8.0.53\bin\Tomcat-juli.jar"
-Dcatalina.base="E:\apache-Tomcat-8.0.53"
-Dcatalina.home="E:\apache-Tomcat-8.0.53"
-Djava.io.tmpdir="E:\apache-Tomcat-8.0.53\temp"
org.apache.catalina.startup.Bootstrap  start
```

结论：

startup.bat=catalina.bat start=java Bootstrap start(附带-D、-classpath选项) Tomcat就是一个java程序！

Bootstrap的解读

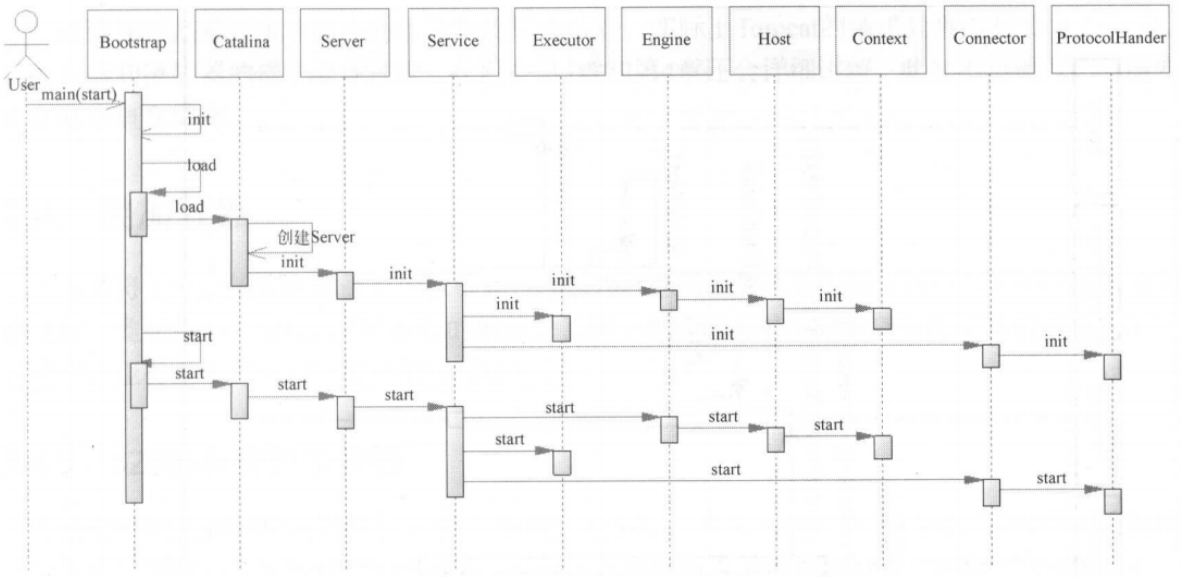
我们发现调用bat文件相当于运行了java bootstrap start 我们来看一个bootStrap的代码： Main函数里面有六个步骤：

1. 创建BootStrap对象。 . 调用bootStrap的init方法。进行初始化操作 . 解析参数。判断出参数是start . 调用setAwait（true）方法。 . 调用load方法 . 调用start方法

这几个方法通过跟进代码发现都是通过反射技术运行了Catalina类中的相关方法。 为什么不直接调用Catalina类中的方法，还要通过bootstrap再利用反射技术调用Catalian呢？

1. 解耦。Bootstrap位于bin目录里面，不是lib目录里面。让Bootstrap跟Tomcat完全松耦合，他可以依赖jre运行，不依赖Tomcat的相关类。使用了反射调用了Catalina类。
2. 封装了Tomcat的类加载器。Bootstrap为Tomcat专门创建了类加载器以实现对类更灵活的配置。

通过跟读代码我们可以发现最终的启动过程如下：



Tomcat中的设计模式

门面模式

介绍

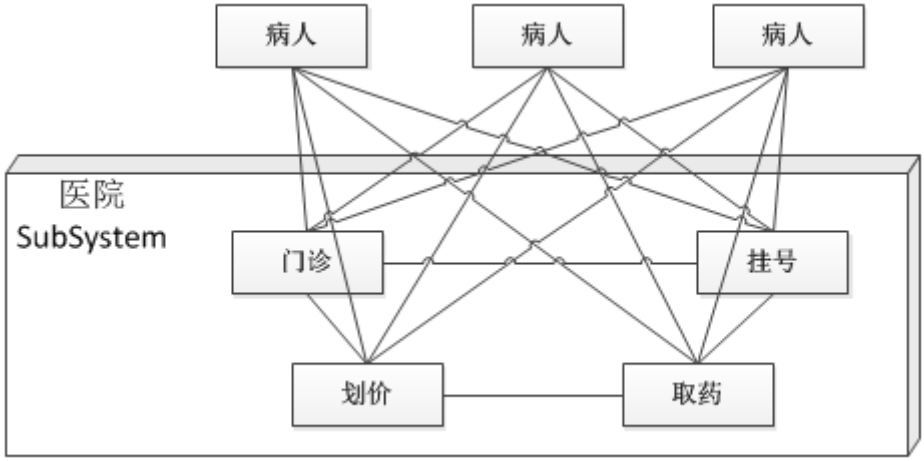
门面模式又称外观模式。（facade）

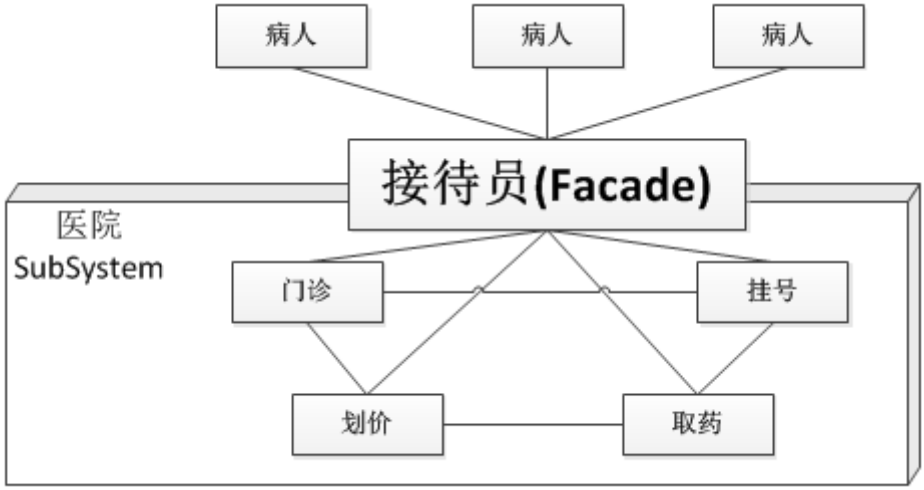
> 外观模式提供一个统一的接口去访问多个子系统的多个不同的接口，它为子系统中的一组接口提供一个统一的高层接口。使用子系统更容易使用。

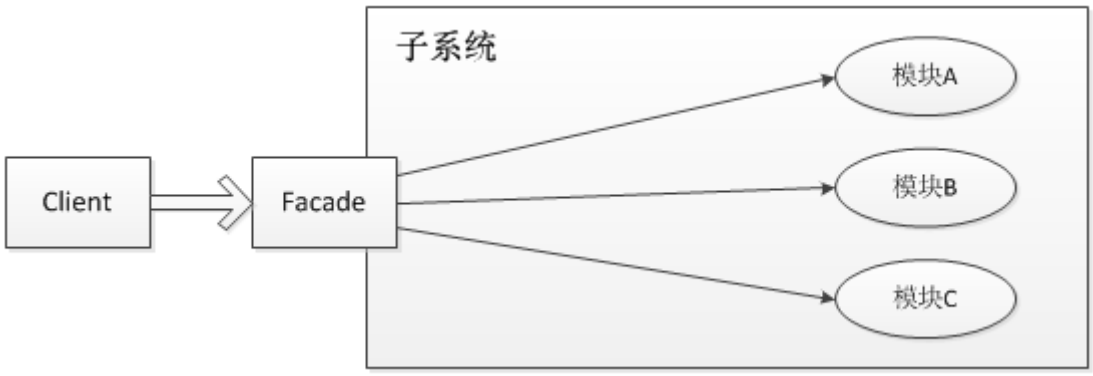
应用场景

> 1. 简化子系统复杂性时。
> 2. 监控所有子系统时；通过门面控制了入口，可以统一监控；
> 3. 希望封装和隐藏子系统时；

比如下图：


病人
病人
病人
医院
SubSystem
门诊
挂号
划价
取药


病人
病人
病人
接待员(Facade)
医院
SubSystem
门诊
挂号
划价
取药


子系统
模块A
模块B
模块C
Client
Facade

案例（订单的支付）

```
package com.facade;

public class AccountService {
    //get customer balance
    public Double getBalance(Long accountID) {
        return 1000.0;
    }

    //check if customer id is valid
    public boolean CKey(Long accountID) {
        return true;
    }
}
```

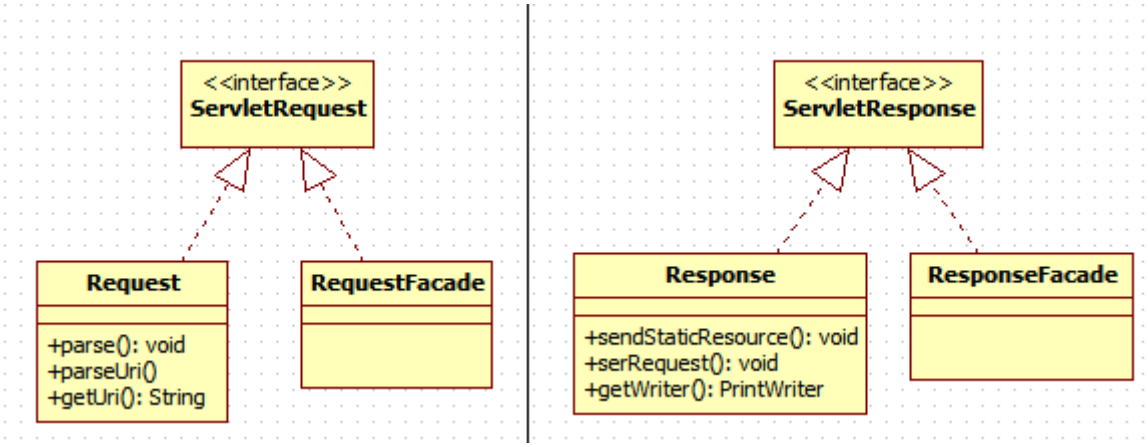
```
package com.facade;

public class OrdersService {
    public boolean checkOrderState(Long orderID) {
        return true;
    }

    public Long getOrder(Long accountID){
        return 123L;
    }
}
```

```
package com.facade;

public class PaymentService {
    public boolean pay(Long accountID,Long orderID){
        return true;
    }
}
```

```
package com.facade;

public class PaymentSystemFacade {

    AccountService accountService = new AccountService();
    OrdersService orderService = new OrdersService();
    PaymentService paymentService = new PaymentService();

    public boolean payForSpecifiedOrder(Long accountID, Long orderID) {
        boolean result = false;
        if (accountService.CKey(accountID) &&
orderService.checkOrderState(orderID)) {
            if (paymentService.pay(accountID, orderID))
                result = true;
            else
                result = false;
        }
        return result;
```

```
    }
}
```

Tomcat中的门面模式



Tomcat中的request，response和session都采用了门面模式进行封装。作用是隐藏request对象底层的方法，只对外公布需要外部调用的方法。

我们写的servlet的service方法里面的参数request和response对象就是被封装成门面的requestFacade和responseFacade。这样我们在servlet里面就不能调用原生的request对象的方法了，而是只能调用requestFacade提供给我们的方法，保证了系统的安全。

责任链模式

责任链模式是比较常见的设计模式之一。

优点

1. 责任链模式将请求和处理分开，请求者不知道是谁处理的，处理者可以不用知道请求的全貌。
2. 提高系统的灵活性。

应用场景

1. 一个请求需要一系列的处理工作。
2. 业务流的处理，例如文件审批。
3. 对系统进行功能扩展补充

概念

顾名思义，责任链模式是一条链，链上有多个节点，每个节点都有各自的责任。当有输入时，第一个责任节点看自己能否处理该输入，如果可以就处理。如果不能就交由下一个责任节点处理。依次类推，直到最后一个责任节点。

举个例子：

> 需求开发例子
>
> > 假设现在有个需求来了，首先是实习生拿到这个需求。如果实习生能够实现，直接实现。如果不行，他把这个需求交给初级工程师。如果初级工程师能够实现，直接实现。如果不行，交给中级工程师。如果中级工程师能够实现，直接实现。如果不行，交给高级工程师。如果高级工程师能够实现，直接实现。如果不行，交给 CTO。如果 CTO能够实现，直接实现。如果不行，直接跟产品说，需求不做。

> 买衣服的例子

> 你肯定是到店里，让老板把相中的几款衣服拿出来。然后你一个一个试。第一个不行，就第二个。第二个不行，就第三个。...，直到找到合适的。

案例

> 需求：给定一个输入值，根据输入值执行不同逻辑。

最初的代码

```
String input = "1";
if ("1".equals(input)) {
    //TODO do something
} else if ("2".equals(input)) {
    //TODO do something
} else if ("3".equals(input)) {
   //TODO do something
}
```

如果每个分支里面的逻辑比较简单，那还好，如果逻辑复杂，假设每个case大概要 100 行代码处理，有 10 个 case，一下子就出来一个“千行代码”文件。而且还不利于维护、测试和扩展。 如果能够想办法把代码拆分成每个case为一个文件，这样不仅代码逻辑清晰了很多，而且不管是后续维护、扩展还是进行测试，都方便很多。

责任链模式拆分代码

定义抽象类：

```
public abstract class BaseCase {
    //标记自己是否可以处理
    private boolean isConsume;

    public BaseCase(boolean isConsume) {
        super();
        this.isConsume = isConsume;
    }
    //下一个责任连点
    private BaseCase nextBaseCase;

    public BaseCase getNextBaseCase() {
        return nextBaseCase;
    }
    public void setNextBaseCase(BaseCase nextBaseCase) {
        this.nextBaseCase = nextBaseCase;
    }
    public void handleRequest(){
        //如果当前节点可以处理就当前节点处理，否则就交给下一个节点处理
        if(isConsume){
            doSomeThing();
        }else{
            if(nextBaseCase!=null){
                nextBaseCase.handleRequest();
            }
        }
    }
    public abstract void doSomeThing();
}
```

各个case来实现抽象类：

```
public class OneCase extends BaseCase {
    public OneCase(boolean isConsume) {
        super(isConsume);
    }

    @Override protected void doSomething() {
    // TODO do something
        System.out.println(getClass().getName());
    }
}
```

客户端调用的代码：

```
String input = "1";
OneCase oneCase = new OneCase("1".equals(input));
TwoCase twoCase = new TwoCase("2".equals(input));
DefaultCase defaultCase = new DefaultCase(true);
oneCase.setNextCase(twoCase);
twoCase.setNextCase(defaultCase);
oneCase.handleRequest();
```

Tomcat中的责任链模式

Tomcat中有2个地方用到了责任链模式，一个是过滤器链中的过滤器，另一个是管道中的阀门。

过滤器链的实现

Filter接口如下所示：

```
public interface Filter {
    public void doFilter(Request request, Response response,
            FilterChain chain);
}
```

我们在具体的Filter实现类中调用chain.doFilter()来执行下一个过滤器节点。

> 注意FilterChain中的doFilter与Filter中的doFilter没有任何关系，名字虽然相同，但是参数是不一样的。FilterChain中的doFilter只有Request和Response两个参数，其实更好的命名方式应该是chain.doNextFilter(resquest,response),因为chain的doFilter方法是来执行一个过滤器节点的。

我们跟进FilterChain发现他是一个接口。

```
public interface FilterChain {
    public void doFilter(Request request, Response response);
    public void addFilter(FilterConfig config);
}
```

ApplicationFilterChain实现了FilterChain,其核心代码如下所示：

```
final class ApplicationFilterChain implements FilterChain {
    private ApplicationFilterConfig[] filters = new ApplicationFilterConfig[10];
    private int pos = 0;//维持过滤器链中的当前位置
    private int n = 0;//过滤器链中的过滤器数量
    public void doFilter(Request request, Response response) {
        internalDoFilter(request,response);
    }
    private void internalDoFilter(Request request,
                                  Response response) {
        if(pos<n) {
            ApplicationFilterConfig filterConfig = filters[pos++];
            Filter filter = filterConfig.getFilter();
            filter.doFilter(request, response, this);
            return;
        }
    }
    void addFilter(ApplicationFilterConfig filterConfig) {
        //省略了扩容部分。。。
        filters[n++] = filterConfig;
    }
}
```

> 可以看出在FilterChain的实现中确实是用数组来保存所有的过滤器，但是并不是直接保存的Filter类型，而是保存的FilterConfig类型，可以把FilterConfig当做Filter的包装类，我们可以通过filterConfig.getFilter()拿到Filter实例。

客户端代码：

```
public class Client{
    public static void main(String[] args) {
        Request request = new Request();
        Response response = new Response();
        Filter filter1 = new MyFilter1();
        Filter filter2 = new MyFilter2();
        Filter filter3 = new MyFilter3();
        FilterChain chain = new ApplicationFilterChain();
        ApplicationFilterConfig filterConfig1 = new
ApplicationFilterConfig(filter1);
        ApplicationFilterConfig filterConfig2 = new
ApplicationFilterConfig(filter2);
        ApplicationFilterConfig filterConfig3 = new
ApplicationFilterConfig(filter3);
        chain.addFilter(filterConfig1);
        chain.addFilter(filterConfig2);
        chain.addFilter(filterConfig3);
        chain.doFilter(request, response);
    }
}
```

管道中的阀门的实现

在Tomcat里面，为了提高容器的灵活性和可扩展性，对容器都使用了责任链模式实现请求的处理。

> 什么是容器（Container）？

> 容器表示的是Tomcat中的一类组件，这类组件的作用是处理和接收来自客户端的请求并返回响应数据。尽管有的具体操作不是该组件完成的，有可能是交给子组件完成，但是从行为定义上，他们是一致的。Tomcat用Container表示容器，并且他可以添加并维护子容器。因此Engine,Host,Context,Wrapper均继承Container。

也就是说Tomcat中的Engine，Host，Context，Wrapper里面都维护了一个管道（Pipeline）和若干阀门(valve)。

> Pipeline是用于构造责任链，Valve是用来代表责任链上的每一个处理器。

过滤器和管道两个技术的比较

Pipeline/Valve	FilterChain/Filter
管道（Pipeline）	过滤器链（FilterChain）
阀门（Valve）	过滤器（Filter）
底层实现为具有头（first）、尾（basic）指针的单向链表	底层实现为数组
Valve的核心方法invoke(request,response)	Filter核心方法doFilter(request,response,chain)
pipeline.getFirst().invoke(request,response)	filterchain.doFilter(request,response)

FilterChain中用数组保存了所有的过滤器，而在管道Pipeline中则采用的是链表的方式将一个个的阀门Valve链接起来的。

Valve的接口如下：

```
public interface Valve {
    //获取下一个阀门
    public Valve getNext();
    public void setNext(Valve valve);
    //对请求和响应进行处理
    public void invoke(Request request, Response response);
}
```

具体的阀门类：

```
public class MyValve implements Valve {
    protected Valve next = null;
    @Override
    public Valve getNext() {
        return next;
    }
    @Override
    public void setNext(Valve valve) {
        this.next = valve;
    }
    @Override
    public void invoke(Request request, Response response) {
        //处理request,response
        //...
        getNext().invoke(request, response);
        return;
```

```
    }
}
```

Pipeline接口：

```
public interface Pipeline{
    //获取管道中默认的阀门实例，这个默认的阀门是在管道中最后一个位置存储，封装了具体的请求处理和输出响应的过程
    public Valve getBasic();
    public void setBasic(Valve valve);
    //为管道添加一个阀门实例，新添加的实例位于Basic之前
    public void addValve(Valve valve);
    //获取管道中的第一阀门实例
    public Valve getFirst();
}
```

Tomcat中Pipeline的实现类为StandardPipeline，核心代码如下所示：

```
public class StandardPipeline implements Pipeline {
    protected Valve basic = null;
    protected Valve first = null;
    @Override
    public Valve getBasic() {
        return this.basic;
    }
    @Override
    //这个默认的阀门是在管道中最后一个位置存储，封装了具体的请求处理和输出响应的过程
    public void setBasic(Valve valve) {
        Valve oldBasic = this.basic;
        Valve current = first;
        while (current!=null) {
            if (current.getNext() == oldBasic) {
                current.setNext(valve);
                break;
            }
            current = current.getNext();
        }
        this.basic = valve;
    }
    @Override
    //新添加的阀门位于先前添加的阀门后面，basic阀门前面。
    public void addValve(Valve valve) {
        if (first == null) {
            first = valve;
            valve.setNext(basic);
        } else {
            Valve current = first;
            while (current != null) {
                if (current.getNext() == basic) {
                    current.setNext(valve);
                    valve.setNext(basic);
                    break;
                }
                current = current.getNext();
            }
        }
```

```
    }
    @Override
    public Valve getFirst() {
        if (first != null) {
            return first;
        }
        return basic;
    }
}
```

可以看出StandardPipeline是一个具有头尾指针的单向链表。first相当于头指针，basic相当于尾指针。addValve方法的作用是把管道中的各个阀门链接起来。 在Tomcat中是通过容器组件对管道进行调用的，wrapper就是一种容器，因此可以通过如下方式进行调用：

```
pipeline.getFirst().invoke(request, response);
```

模板方法模式

概念

模板方法模式（TemplateMethod），定义一个操作中的算法的骨架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

举例

举个蒸包子的例子，蒸包子我们来拆分一下，看看有哪些步骤： 1.和面 2.准备包子馅 3.包包子 4.蒸 我们大致将蒸包子的步骤分为上面四个，有哪些步骤是一样的呢？和面，包包子，蒸的步骤都是一样的，但是准备包子馅不一样，有猪肉的，牛肉的，梅干菜的，豆沙的。 那么我们把动作一样的部分写死，把可变的部分提供为抽象来供大家实现，是不是就能蒸出包子了呢？

代码的实现

父类：

```
package com.miaoke.template;
/**
 * 蒸包子类，封装了蒸包子的流程
 * @author Administrator
 *
 */
public abstract class AbstractSteamedBun {
    public final void steameBun() {
        //和面
        kneadDough();
        //准备馅儿
        prepareStuffing();
        //包包子
        wrapBun();
        //蒸
        steam();
    }
    //做什么馅儿有子类实现。
    abstract void prepareStuffing();
```

```
    private void kneadDough(){
        System.out.println("和面。。。");
    }
    private void wrapBun(){
        System.out.println("把馅儿包到包子里面。。。");
    }
    private void steam(){
        System.out.println("开始上锅蒸。。。");
    }
}
```

子类：

```
package com.miaoke.template;
public class SteamedBeefBun extends AbstractSteamedBun {
    @Override
    protected void prepareStuffing() {
        System.out.println("开始准备牛肉包子馅");
    }
}
```

模式中的概念：

抽象方法：父类中只声明但不加以实现，而是定义好规范，然后由它的子类去实现。比如例子中的准备包子馅的方法。 模版方法：由抽象类声明并加以实现。一般来说，模版方法调用抽象方法来完成主要的逻辑功能，并且，模版方法大多会定义为final类型，指明主要的逻辑功能在子类中不能被重写。比如蒸包子的整个过程。

模板方法模式的使用场景

1.有多个子类公用同一个方法，该方法的逻辑相同。 2.重要的、复杂的方法，可以考虑作为该模式中的模板方法。

Tomcat中有两处使用模板方法：

> 我们在编写servlet的时候，需要继承javax.servlet.http.HttpServlet然后根据需要重写父类的doGet,doPost, doPut, doDelete等等。然而doGet, doPost, doPut, doDelete这些方法是如何被调用的呢？这里Tomcat运用了模板方法模式。父类javax.servlet.http.HttpServlet不仅实现了doGet,doPost, doPut, doDelete这些方法的默认逻辑，还实现了这些方法通用的调用规则函数voidjavax.servlet.http.HttpServlet.service(HttpServletRequestreq, HttpServletResponseresp)。service函数就是一个模板，在模板中调用其他方法。这样子类就不需要关心各个方法的调用规则，只需重写需要的方法实现需要的逻辑即可。

> Tomcat中关于生命周期管理的地方很好应用了模板方法模式，在一个组件的生命周期中都会涉及到init(初始化)，start（启动），stop(停止)，destory（销毁），而对于每一个生命周期阶段其实都有固定一些事情要做，比如判断前置状态，设置后置状态，以及通知状态变更事件的监听者等，而这些工作其实是可以固化的，所以Tomcat中就将每个生命周期阶段公共的部分固化交给lifeCycleBase去定义，然后通过initInternal,startInternal,stopInternal,destoryInternal这几个方法开放给子类去实现具体的逻辑。

> 当然Tomcat里面除了上面的典型的设计模式还涉及其他众多设计模式，比如工厂，单例，观察者，命令等，我们就不一一讲解了。