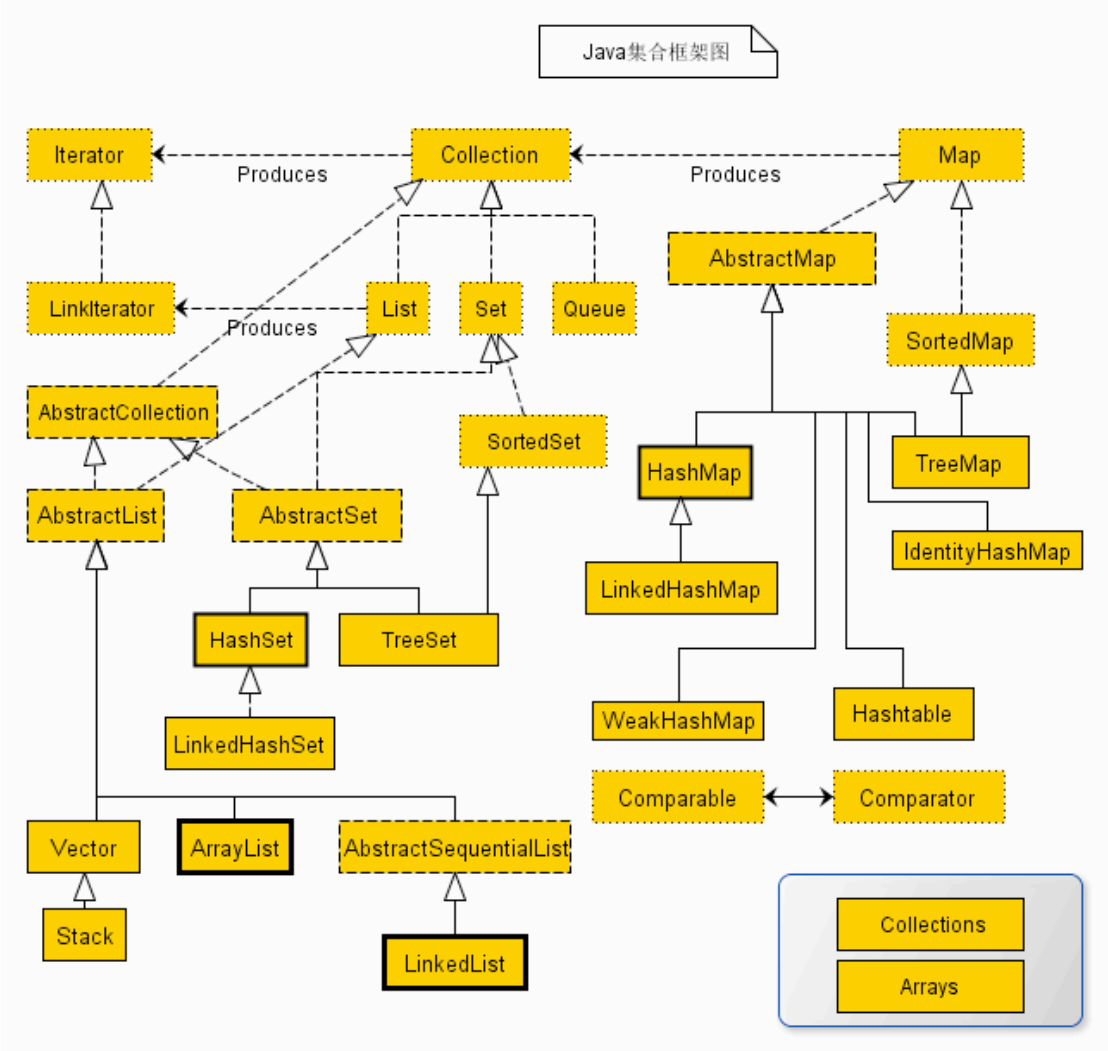
- **你对 Java 的集合框架了解吗？ 能否说说常用的类？**

Java 集合框架类图：



我常用的类：
HashMap,Hashtable,HashSet,ArrayList,Vector,LinkedList,Collections,Arrays;

- **说说 Hashtable 与 HashMap 的区别(源代码级别)**

  - 最明显的区别在于 Hashtable 是同步的(每个方法都是 synchronized)，而 HashMap 则不是.
  - HashMap 继承至 AbstractMap,Hashtable 继承至 `Dictionary`，前者为 `Map` 的骨干， 其内部已经实现了 `Map` 所需要做的大部分工作， 它的子类只需要实现它的少量方法即可具有 `Map` 的多项特性。而后者内部都为抽象方法，需要它的实现类一一作自己的实现，且该类已过时
  - 两者检测是否含有 key 时，hash 算法不一致，HashMap 内部需要将 key 的 hash

码重新计算一边再检测，而 Hashtable 则直接利用 key 本身的 hash 码来做验证。

HashMap:

```
 int hash = (key == null) ? 0 : hash(key.hashCode());
-----
static int hash(int h) {
       h ^= (h >>> 20) ^ (h >>> 12);
       return h ^ (h >>> 7) ^ (h >>> 4);
    }
```

Hashtable:

```
int hash = key.hashCode();
```

- 两者初始化容量大小不一致，HashMap 内部为 16*0.75 , Hashtable 为 11*0.75

HashMap:

```
static final int DEFAULT_INITIAL_CAPACITY = 16;
static final float DEFAULT_LOAD_FACTOR = 0.75f;
public HashMap() {
     this.loadFactor = DEFAULT_LOAD_FACTOR;

threshold=(int)(DEFAULT_INITIAL_CAPACITY*DEFAULT_LOAD_FACTOR);
     table = new Entry[DEFAULT_INITIAL_CAPACITY];
     init();
}
```

Hashtable:

```
public Hashtable() {
    this(11, 0.75f);
}
-----
 public Hashtable(int initialCapacity, float loadFactor) {
       ..........
    this.loadFactor = loadFactor;
    table = new Entry[initialCapacity];
    threshold = (int)(initialCapacity * loadFactor);
    }
```

后续的区别应该还有很多， 这里先列出 4 点。

- **平时除了 ArrayList 和 LinkedList 外，还用过的 List 有哪些？ArrayList 和 LinkedList 的区别？**

事实上，我用过的 List 主要就是这 2 个， 另外用过 Vector.

ArrayList 和 LinkedList 的区别：

1. 毫无疑问，第一点就是两者的内部数据结构不同， ArrayList 内部元素容器是一个 Object 的数组， 而 LinkedList 内部实际上一个链表的数据结构，其有一个内部类

来表示链表.

```
(ArrayList)
private transient Object[] elementData;
```

```
(LinkedList)
private transient Entry<E> header = new Entry<E>(null, null, null);//
链表头

//内部链表类.
private static class Entry<E> {
    E element; //数据元素
    Entry<E> next; // 前驱
    Entry<E> previous;//后驱
    Entry(E element, Entry<E> next, Entry<E> previous) {
        this.element = element;
        this.next = next;
        this.previous = previous;
    }
}
```

2. 两者的父类不同，也就决定了两者的存储形式不同。 ArrayList 继承于 **AbstractList, 而 LinkedList 继承于 AbstractSequentialList. 两者都实现了 List 的骨干结构，只是前者的访问形式趋向于** “随机访问”数据存储（如数组），后者趋向于 “连续访问”数据存储（如链接列表）

```
public class ArrayList<E> extends AbstractList<E>
public class LinkedList<E> extends AbstractSequentialList<E>
```

3. 再有就是两者的效率问题， ArrayList 基于数组实现，所以毫无疑问可以直接用下标来索引，其索引数据快，插入元素设计到数组元素移动，或者数组扩充，所以插入元素要慢。LinkedList 基于链表结构，插入元素只需要改变插入元素的前后项的指向即可，故插入数据要快，而索引元素需要向前向后遍历，所以索引元素要慢。

ArrayList 的特点，内部容器是如何扩充的？

上一点谈到了 ArrayList 的特点，这里略，重点来看其内部容器的扩充：

```
public void ensureCapacity(int minCapacity) {
        modCount++;
        int oldCapacity = elementData.length;
        if (minCapacity > oldCapacity) {
            Object oldData[] = elementData;
             //这里扩充的大小为原大小的大概 60%
            int newCapacity = (oldCapacity * 3) / 2 + 1;
            if (newCapacity < minCapacity)
                newCapacity = minCapacity;
            //创建一个指定大小的新数组来覆盖原数组
            elementData = Arrays.copyOf(elementData, newCapacity);
        }
    }
```

Properties 类的特点？ 线程安全？

Properties 继承于 Hashtable,,所以它是线程安全的.

其特点是:

它表示的是一个持久的属性集，它可以保存在流中或者从流中加载，属性列表的每一个键和它所对应的值都是一个“字符串”

其中，常用的方法是 load()方法，从流中加载属性：

```
public synchronized void load(InputStream inStream) throws
IOException {
        // 将输入流转换成LineReader
        load0(new LineReader(inStream));
    }

    private void load0(LineReader lr) throws IOException {
        char[] convtBuf = new char[1024];
        int limit;
        int keyLen;
        int valueStart;
        char c;
        boolean hasSep;
        boolean precedingBackslash;
        // 一行一行处理
        while ((limit = lr.readLine()) >= 0) {
```

```
            c = 0;
            keyLen = 0;
            valueStart = limit;
            hasSep = false;
            precedingBackslash = false;
            // 下面用2个循环来处理key,value
            while (keyLen < limit) {
                c = lr.lineBuf[keyLen];
                // need check if escaped.
                if ((c == '=' || c == ':') && !precedingBackslash) {
                    valueStart = keyLen + 1;
                    hasSep = true;
                    break;
                } else if ((c == ' ' || c == '\t' || c == '\f')
                        && !precedingBackslash) {
                    valueStart = keyLen + 1;
                    break;
                }
                if (c == '\\') {
                    precedingBackslash = !precedingBackslash;
                } else {
                    precedingBackslash = false;
                }
                keyLen++;
            }

            while (valueStart < limit) {
                c = lr.lineBuf[valueStart];
                if (c != ' ' && c != '\t' && c != '\f') {
                    if (!hasSep && (c == '=' || c == ':')) {
                        hasSep = true;
                    } else {
                        break;
                    }
                }
                valueStart++;
            }

            String key = loadConvert(lr.lineBuf, 0, keyLen, convtBuf);
            String value = loadConvert(lr.lineBuf, valueStart, limit
                    - valueStart, convtBuf);
            // 存入内部容器中，这里用的是Hashtable 内部的方法.
            put(key, value);
        }
```

```
    }
```

LineReader 类，是 Properties 内部的类：

```
class LineReader {
        public LineReader(InputStream inStream) {
            this.inStream = inStream;
            inByteBuf = new byte[8192];
        }

        public LineReader(Reader reader) {
            this.reader = reader;
            inCharBuf = new char[8192];
        }

        byte[] inByteBuf;
        char[] inCharBuf;
        char[] lineBuf = new char[1024];
        int inLimit = 0;
        int inOff = 0;
        InputStream inStream;
        Reader reader;

        /**
         * 读取一行
         *
         * @return
         * @throws IOException
         */
        int readLine() throws IOException {
            int len = 0;
            char c = 0;
            boolean skipWhiteSpace = true;// 空白
            boolean isCommentLine = false;// 注释
            boolean isNewLine = true;// 是否新行.
            boolean appendedLineBegin = false;// 加 至行开始
            boolean precedingBackslash = false;// 反斜杠
            boolean skipLF = false;
            while (true) {
                if (inOff >= inLimit) {
                    // 从输入流中读取一定数量的字节并将其存储在缓冲区数组
inCharBuf/inByteBuf中,这里区分字节流和字符流
                    inLimit     =     (inStream     ==     null)     ?
reader.read(inCharBuf)
                            : inStream.read(inByteBuf);
                    inOff = 0;
```

```
                    // 读取到的为空.
                    if (inLimit <= 0) {
                        if (len == 0 || isCommentLine) {
                            return -1;
                        }
                        return len;
                    }
                }
                if (inStream != null) {
                    // 由于是字节流，需要使用ISO8859-1来解码
                    c = (char) (0xff & inByteBuf[inOff++]);
                } else {
                    c = inCharBuf[inOff++];
                }

                if (skipLF) {
                    skipLF = false;
                    if (c == '\n') {
                        continue;
                    }
                }
                if (skipWhiteSpace) {
                    if (c == ' ' || c == '\t' || c == '\f') {
                        continue;
                    }
                    if (!appendedLineBegin && (c == '\r' || c == '\n'))
{
                        continue;
                    }
                    skipWhiteSpace = false;
                    appendedLineBegin = false;
                }
                if (isNewLine) {
                    isNewLine = false;
                    if (c == '#' || c == '!') {
                        // 注释行，忽略.
                        isCommentLine = true;
                        continue;
                    }
                }
                // 读取真正的属性内容
                if (c != '\n' && c != '\r') {
                    // 这里类似于ArrayList内部的容量扩充，使用字符数组来保存
读取的内容.
```

```
                    lineBuf[len++] = c;
                    if (len == lineBuf.length) {
                        int newLength = lineBuf.length * 2;
                        if (newLength < 0) {
                            newLength = Integer.MAX_VALUE;
                        }
                        char[] buf = new char[newLength];
                        System.arraycopy(lineBuf,    0,    buf,    0,
lineBuf.length);
                        lineBuf = buf;
                    }
                    if (c == '\\') {
                        precedingBackslash = !precedingBackslash;
                    } else {
                        precedingBackslash = false;
                    }
                } else {
                    // reached EOL 文件结束
                    if (isCommentLine || len == 0) {
                        isCommentLine = false;
                        isNewLine = true;
                        skipWhiteSpace = true;
                        len = 0;
                        continue;
                    }
                    if (inOff >= inLimit) {
                        inLimit    =    (inStream    ==    null)    ?
reader.read(inCharBuf)
                                : inStream.read(inByteBuf);
                        inOff = 0;
                        if (inLimit <= 0) {
                            return len;
                        }
                    }
                    if (precedingBackslash) {
                        len -= 1;
                        skipWhiteSpace = true;
                        appendedLineBegin = true;
                        precedingBackslash = false;
                        if (c == '\r') {
                            skipLF = true;
                        }
                    } else {
                        return len;
```

```
                    }
                }
            }
        }
    }
```

这里特别的是，实际上，Properties 从流中加载属性集合，是通过将流中的字符或者字节分成一行行来处理的。

请说一下 Struts2 的初始化？和类的创建？(从源代码角度出发)

由于这个问题研究起来可以另外写一篇专门的模块，这里只列出相对简单的流程，后续会希望有时间整理出具体的细节:

首先，Struts2 是基于 Xwork 框架的，如果你有仔细看过 Xwork 的文档，你会发现，它的初始化过程基于以下几个类：

Configuring XWork2 centers around the following classes:-

- ConfigurationManager
- ConfigurationProvider
- Configuration

而在 ConfigurationProvider 的实现类 `XmlConfigurationProvider` 的内部，你可以看到下面的代码：

```
 public XmlConfigurationProvider() {
        this("xwork.xml", true);
}
```

同样的，Struts2 的初始化也是这样的一个类，只不过它继承于 Xwork 原有的类，并针对 Struts2 做了一些特别的定制。

类:

```
public class StrutsXmlConfigurationProvider
   extends XmlConfigurationProvider {
    public StrutsXmlConfigurationProvider(boolean errorIfMissing)
    {
        this("struts.xml", errorIfMissing, null);
    }
……
```

如果你要查看这个类在哪里调用了，你会追踪到 Dispatch 的类，

记得吗？我们使用 Struts2，第一步就是在 Web.xml 中配置一个过滤器 FilterDispatcher,

没错，在 web 容器初始化过滤器的时候，同时也会初始化 Dispatch..

FilterDispatch.init():

```
public void init(FilterConfig filterConfig)
```

```
throws ServletException {
        try {
            this.filterConfig = filterConfig;
            initLogging();
            dispatcher = createDispatcher(filterConfig);
            dispatcher.init();////初始化Dispatcher.
            dispatcher.getContainer().inject(this);
            staticResourceLoader.setHostConfig(new
FilterHostConfig(filterConfig));
        } finally {
            ActionContext.setContext(null);
        }
    }
```

```
Dispatch.init():
//这里是加载配置文件， 真正初始化 Struts2 的 Action 实例还没开始，
public void init() {
        if (configurationManager == null) {
            configurationManager =
new ConfigurationManager(BeanSelectionProvider.DEFAULT_BEAN_NAME);
        }
        init_DefaultProperties(); // [1]
        init_TraditionalXmlConfigurations(); // [2]
        init_LegacyStrutsProperties(); // [3]
        init_CustomConfigurationProviders(); // [5]
        init_FilterInitParameters() ; // [6]
        init_AliasStandardObjects() ; // [7]
        Container container = init_PreloadConfiguration();
        container.inject(this);
        init_CheckConfigurationReloading(container);
        init_CheckWebLogicWorkaround(container);
        if (!dispatcherListeners.isEmpty()) {
            for (DispatcherListener l : dispatcherListeners) {
                l.dispatcherInitialized(this);
            }
        }
    }
```

到初始化 Action 类的时候，你需要去 FilterDispatcher 的 doFilter 方法去看代码，你会发现：

```
public void doFilter(ServletRequest req, ServletResponse res,
FilterChain chain) throws IOException, ServletException {
……
dispatcher.serviceAction(request,    response,    servletContext,
mapping);
```

再追踪到 Dispatcher 类，看到这个方法：

```
public void serviceAction(HttpServletRequest request,
 HttpServletResponse response, ServletContext context,
                          ActionMapping mapping)
throws ServletException {

     ……
          ActionProxy proxy =
config.getContainer().getInstance(ActionProxyFactory.class).creat
eActionProxy(namespace, name, method, extraContext, true, false);

    ……
```

这行代码已经明确的告诉你了， 它的作用就是创建 ActionProxy,而我们想要知道的是，他是如果创建的

而上面代码中的 config，实际上是 Xwork 中的.Configuration， 如果你打开 Xwork 源代码，你会发现，他其实是一个接口， 真正做处理的，这里是 com.opensymphony.xwork2.config.impl.DefaultConfiguration 类， 通过它的 getContainer()方法，获取到一个 Container 类型的实例，而 Container 也是一个接口， 其实现类是：
com.opensymphony.xwork2.inject.ContainerImpl
 他的 getInstance(Class clazz):

```
public <T> T getInstance(final Class<T> type) {
    return callInContext(new ContextualCallable<T>() {
      public T call(InternalContext context) {
        return getInstance(type, context);
      }
    });
  }
```

返回的是你传入的对象，而在这里就是：ActionProxyFactory（也是接口，真正返回的是 com.opensymphony.xwork2.DefaultActionProxyFactory）

而现在，到了真正开始处理加载 Action 实例的时候了：

```
 public ActionProxy createActionProxy(ActionInvocation inv, String
namespace, String actionName, String methodName,
boolean executeResult, boolean cleanupContext) {
        DefaultActionProxy proxy = new DefaultActionProxy(inv,
 namespace, actionName, methodName, executeResult, cleanupContext);
        container.inject(proxy);
        proxy.prepare();
        return proxy;
    }
```

这里，我们主要关心的是：
Proxy.prepare()：

```
protected void prepare()  {
     ……
         invocation.init(this);
……
    }
```

OK， 我们进去看看，这里发生了什么？

这里也是面向接口编程，真实情况是，它调用了 com.opensymphony.xwork2.DefaultActionInvocation 的 init(ActionProxy)方法：

```
public void init(ActionProxy proxy) {
       ……

       createAction(contextMap);

       ……
    }
```

OK， 我们终于追踪到我们所需要了解的地方了， 到底 Struts2/Xwork 的 Action 是如何创建的呢？

```
protected void createAction(Map<String, Object> contextMap) {
       // load action
       String timerKey = "actionCreate: " + proxy.getActionName();
      ……
action =
 objectFactory.buildAction(proxy.getActionName(),
proxy.getNamespace(), proxy.getConfig(), contextMap);
     ……
```

继续跟进去看看，你发现， 事情确实如此：

```
public Object buildAction(String actionName, String namespace,
ActionConfig config, Map<String, Object> extraContext)
throws Exception {
       return buildBean(config.getClassName(), extraContext);
    }
```

```
public Object buildBean(String className, Map<String, Object>
extraContext, boolean injectInternal) throws Exception {
       Class clazz = getClassInstance(className);//根据Action的名字，
进行初始化
       Object obj = buildBean(clazz, extraContext);//利用发射来做实例
```

```
初始化.
        if (injectInternal) {
            injectInternalBeans(obj);
        }
        return obj;
        }
```

```
 public Class getClassInstance(String className) throws
ClassNotFoundException {
        if (ccl != null) {
            return ccl.loadClass(className);
        }
        return
ClassLoaderUtil.loadClass(className, this.getClass());
}
```

```
public Object buildBean(Class clazz, Map<String, Object>
extraContext) throws Exception {
        return clazz.newInstance();
   }
```

OK, 整体来说，这个问题说清楚很难，因为你无法记住你追踪到的所有的类，但是有一点是肯定的，那就是流程： 基本上我的理解就是 通过一系列配置文件的初始化，将文件转换成对象，加载进内存中，再在处理请求时候(注意，只有当 FilterDispatcher 的 doFilter 第一次被调用时，才会去初始化 Action 类)，加载 Action 类来进行业务处理。

第二部分：

据你了解，除了反射还有什么方式可以动态的创建对象？

请说一下 Struts2 是如何把 Action 交给 Spring 托管的？它是单例的还是多例？ 你们页面的表单对象是多例还是单例？

首先，来看看如何让 Spring 来管理 Action.

1.在 struts.xml 中加入

```
<constant name="struts.objectFactory" value="spring"/>
```

2. 有两种整合方式：

a) 将 Struts 的业务逻辑控制器类配置在 Spring 的配置文件中，业务逻辑控制器中引用的业务类一并注入。（这样的处理，必须将 action 类的 scope 配置成 property）

```
<bean id="LoginAction" class="yaso.struts.action.LoginAction">

    <property name="loginDao" ref="LoginDao"/>

</bean>
```

接着，在 struts.xml 或者等效的 Struts2 配置文件中配置 Action 时，指定<action>的 class 属性为 Spring 配置文件中相应 bean 的 id 或者 name 值。示例如下：

```
<action name=”LoginAction” class=”LoginAction”>

        <result name=”success”>/index.jsp</result>

</action>
```

b) 第 2 种方式：

业务类在 Spring 配置文件中配置，业务逻辑控制器类不需要配置，Struts2 的 Action 像没有整合 Spring 之前一样配置，<action>的 class 属性指定业务逻辑控制器类的全限定名。 Action 中引用的业务类不需要自己去初始化，Struts2 的 Spring 插件会使用 bean 的自动装配将业务类注入进来，其实 Action 也不是 Struts2 创建的，而是 Struts2 的 Spring 插件创建的。默认情况下，插件使用 by name 的方式装配，可以通过增加 Struts2 常量来修改匹配方式：设置方式为：struts.objectFactory.spring.autoWire = typeName，可选的装配参数如下：

name:相当于 spring 配置的 autowrie="byName"(默认)
type:相当于 spring 配置的 autowrie="byType"
auto:相当于 spring 配置的 autowrie="autodetect"
constructor: 相当于 spring 配置的 autowrie="constructor"

OK，这里说了配置部分，但是，这里有一个问题， 就是 Spring 管理 Action，如果按照第一方式，那么只要通过 scope="property"来配置为每个请求创建一个 Action 实例。 那么第二种方式，我们并没有指定 Action 的作用域（好似也没有地方可配……），那么，这样的整合方式，Action 的创建到底是单例还是多例的呢？

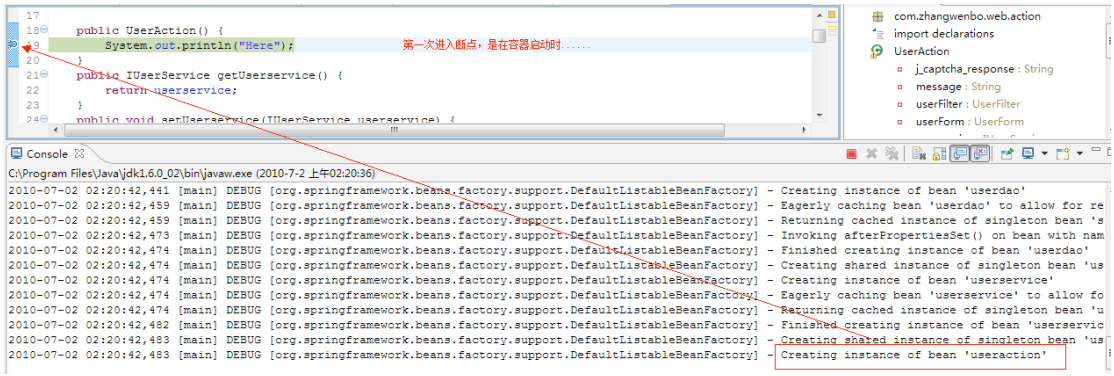
答案也是每个请求一个实例，我这里通过一个很笨的办法，来证明它：

我会写一个 Action 的构造函数， 并在里面打上一句话，加入断点，如果说，每次请求都会进入断点，那么就意味着，每个请求都有一个新的实例是正确的。

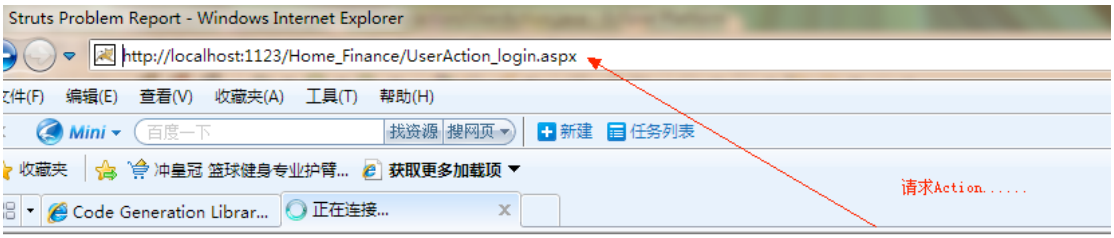
```
14      private UserFilter userFilter = new UserFilter();
15      private UserForm userForm = new UserForm();
16      private IUserService userservice;
17
18      public UserAction() {
19          System.out.println("Here");
20      }
21      public IUserService getUserservice() {
22          return userservice;
23      }
24      public void setUserservice(IUserService userservice) {
25          this.userservice = userservice;
```

我们在这里打个断点.

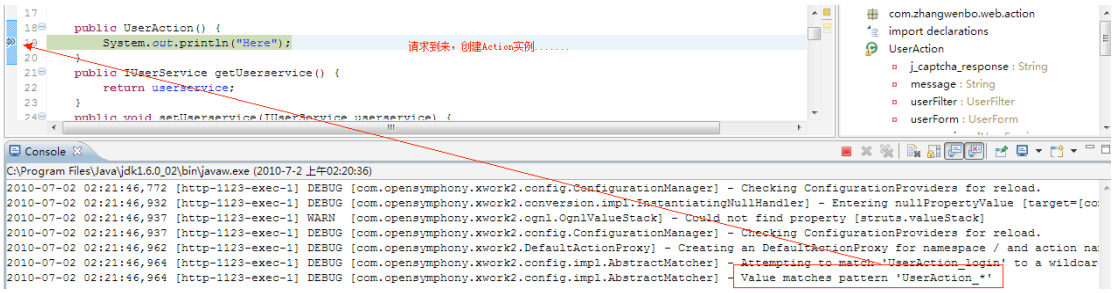
第一次进入的时候，是在容器启动的时候：



我们请求 Action,



再次进入断点，说明，每个请求都有一个 Action 实例来处理。



对于这点的原因，我还是没有弄清楚，为什么按照第 2 种方式配置，不用指定 scope，就会自动的为每个 action 创建一个实例？(希望懂的朋友，可以指点指点)

对于我们项目中的页面表单对象， 毫无疑问，它也是多例的.

请说一下你们业务层对象是单例还是多例的？

业务层对象是单例的。

请说一下 Struts2 源代码中有哪些设计模式？

简单罗列一下：
单例模式-- 典型应用： 类：
org.apache.struts2.config.ServletContextSingleton

模版方法模式：
在 org.apache.struts2.components 包中大量运用

责任连模式：
在拦截器部分使用.

请说一下线程安全出现的原因？

我们都知道线程安全是指什么，我的理解是， 当一个类的“状态”（实例变量）被多个线程所修改时，那么这个类的状态的“正确”性得不到保证，我们就可以理解成线程安全出现。

当然，如果一个没有状态的类，那么它永远都是线程安全的。

再深入一点来看， 我们从 Java 虚拟机的层面来看这个问题，答案就很明朗了：

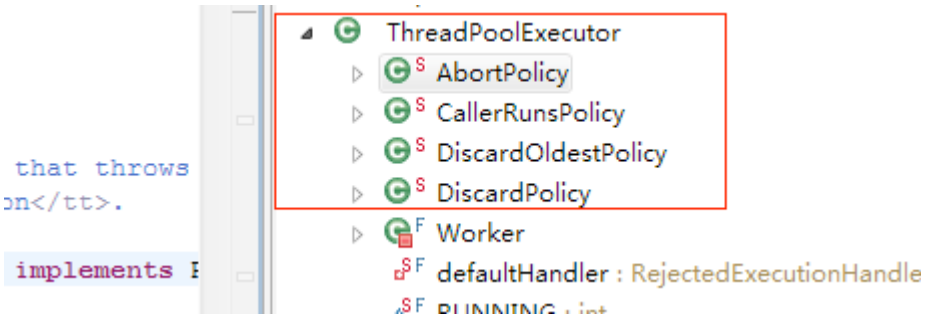
Java 程序在运行时创建的所有类实例或数组，都存放在同一个堆中，而一个 JVM 实例中只存在一个堆空间，因此，它被所有的线程共享着，这样的情况下，就可能出现，多个线程访问对象（堆数据）的同步问题了。

请说一下线程池的中断策略(4 个)？ 各有什么特点？

这里所指的线程池是 concurrent 包中的 ThreadPoolExecutor,而中断策略实际上是指饱和策略（concurent 包中的 RejectedExecutionHandler 接口），

这里需要先解释一下，什么叫饱和策略， 实际就是说， 线程池中的线程容器已经放不下先的任务了，饱和了，必须要有一个相应的策略来处理。

ThreadPoolExecutor 内部，已经定义了 4 种饱和策略：



默认的饱和策略是： （中止）， 既如果放不下了，既中止新加入的任务。

```
private static final RejectedExecutionHandler defaultHandler =new
AbortPolicy();
```

源代码中调用如下

```
public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();
        if              (poolSize              >=              corePoolSize
|| !addIfUnderCorePoolSize(command)) {
            if (runState == RUNNING && workQueue.offer(command)) {
                if (runState != RUNNING || poolSize == 0)
                    ensureQueuedTaskHandled(command);
            }
            else if (!addIfUnderMaximumPoolSize(command))
                    //容不下新的任务了，默认是中止掉
                reject(command); // is shutdown or saturated
        }
    }
```

如果需要设置饱和策略，可以调用 ThreadPoolExecutor 的 setRejectExecutionHandler 方法，JDK 提供了 4 种不同策略的实现(4 种实现都定义在 ThreadPoolExecutor 类中，有兴趣可以查看一下源代码)：

下面介绍一下 4 种实现的特点：

AbortPolicy: （中止）它是默认的策略。

CallerRunsPolicy: (调用者运行), 它既不会丢弃任务，也不会抛出任何异常，它会把任务推回到调用者那里去,以此缓解任务流

DiscardPolicy: （遗弃）策略，它默认会放弃这个任务

DiscardOldestPolicy:（遗弃最旧的），它选择的丢弃的任务，是它本来要执行的（可怜的娃，就这样被新加入的给排挤了），

下面发出这 4 种策略的源代码：

```
/**
     * 饱和策略之----调用者运行策略
     */
public static class CallerRunsPolicy implements
    RejectedExecutionHandler {
```

```
 public CallerRunsPolicy() { }
 public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
          if (!e.isShutdown()) {
              r.run();
          }
  }
}

    /**
     * 饱和策略之----中止（默认的）
     */
    public static class AbortPolicy implements
    RejectedExecutionHandler {
        public AbortPolicy() { }

        /**
         * 直接抛异常，中止
         */
        public void rejectedExecution(Runnable r,
 ThreadPoolExecutor e) {
            throw new RejectedExecutionException();
        }
    }

    /**
     * 饱和策略之----遗弃策略
     */
    public static class DiscardPolicy implements
    RejectedExecutionHandler {

        public DiscardPolicy() { }

        /**
         * 不做任何处理，直接无视
         */
        public void rejectedExecution(Runnable r,
ThreadPoolExecutor e) {
        }
    }

    /**
     * 饱和策略之----遗弃最旧策略
     */
    public static class DiscardOldestPolicy implements
```

```
    RejectedExecutionHandler {
        public DiscardOldestPolicy() { }
        public void rejectedExecution(Runnable r,
 ThreadPoolExecutor e) {
            //遗弃最旧的， 如果是用优先级队列存储池中的任务，则会丢弃优先级最高
的
            if (!e.isShutdown()) {
                e.getQueue().poll();//丢弃
                e.execute(r);//执行新任务
            }
        }
    }
```