首先Web项目使用Spring是通过在web.xml里面配置
org.springframework.web.context.ContextLoaderListener初始化IOC容器的。

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

那就以此为切入点顺藤摸瓜。

public class ContextLoaderListener extends ContextLoader implements ServletContextListener
ContextLoaderListener继承了ContextLoader，并且实现ServletContextListener接口。当Server容器（一般指tomcat）启动时，会收到事件初始化。

```
@Override
public void contextInitialized(ServletContextEvent event) {
    initWebApplicationContext(event.getServletContext());
}
```

initWebApplicationContext方法是在org.springframework.web.context.ContextLoader类里面。方法太长，分段读一下。

```
if (servletContext.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE) != nul
l) {
    throw new IllegalStateException("Cannot initialize context because there is already a root applic
ation context present - " +"check whether you have multiple ContextLoader* definitions in your web.xm
l!");
}
Log logger = LogFactory.getLog(ContextLoader.class);
servletContext.log("Initializing Spring root WebApplicationContext");
if (logger.isInfoEnabled()) {
    logger.info("Root WebApplicationContext: initialization started");
}
long startTime = System.currentTimeMillis();
```

首先是判断servletContext中是否已经注册了WebApplicationContext，如果有则抛出异常，避免重复注册。然后就是启用log，启动计时。本方法的关键就在于try代码块里的内容

```
try {
    // Store context in local instance variable, to guarantee that
    // it is available on ServletContext shutdown.
    if (this.context == null) {
        this.context = createWebApplicationContext(servletContext);
    }
    if (this.context instanceof ConfigurableWebApplicationContext) {
        ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) this.context;
        if (!cwac.isActive()) {
            // The context has not yet been refreshed -> provide services such as
            // setting the parent context, setting the application context id, etc
            if (cwac.getParent() == null) {
                // The context instance was injected without an explicit parent ->
                // determine parent for root web application context, if any.
                ApplicationContext parent = loadParentContext(servletContext);
                cwac.setParent(parent);
            }
            configureAndRefreshWebApplicationContext(cwac, servletContext);
        }
    }
    servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, this.co
ntext);

    ClassLoader ccl = Thread.currentThread().getContextClassLoader();
    if (ccl == ContextLoader.class.getClassLoader()) {
        currentContext = this.context;
    }
    else if (ccl != null) {
        currentContextPerThread.put(ccl, this.context);
    }

    if (logger.isDebugEnabled()) {
        logger.debug("Published root WebApplicationContext as ServletContext attribute with name [" +
                WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE + "]");
    }
    if (logger.isInfoEnabled()) {
        long elapsedTime = System.currentTimeMillis() - startTime;
        logger.info("Root WebApplicationContext: initialization completed in " + elapsedTime + " m
s");
    }

    return this.context;
}
```

这里面有几个关键的方法。首先看一下createWebApplicationContext()

```
protected WebApplicationContext createWebApplicationContext(ServletContext sc) {
    Class<?> contextClass = determineContextClass(sc);
    if (!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass)) {
        throw new ApplicationContextException("Custom context class [" + contextClass.getName() +
                "] is not of type [" + ConfigurableWebApplicationContext.class.getName() + "]");
    }
    return (ConfigurableWebApplicationContext) BeanUtils.instantiateClass(contextClass);
}
```

首先determineContextClass()方法查明具体的Context类，他会读取servletContext的初始化参数contextClass，此参数我们一半不配置，所以Spring就会读取跟org.springframework.web.context.WebApplicationContext同一个包下面的ContextLoader.properties文件读取默认设置，反射出org.springframework.web.context.support.XmlWebApplicationContext类来。

接下来就是在configureAndRefreshWebApplicationContext()方法里将新创建的XmlWebApplicationContext进行初始化。首先会设置一个默认ID，即org.springframework.web.context.WebApplicationContext:+你项目的ContextPath。

```
if (ObjectUtils.identityToString(wac).equals(wac.getId())) {
    // The application context id is still set to its original default
    // value
    // -> assign a more useful id based on available information
    String idParam = sc.getInitParameter(CONTEXT_ID_PARAM);
    if (idParam != null) {
        wac.setId(idParam);
    } else {
        // Generate default id...
        wac.setId(ConfigurableWebApplicationContext.APPLICATION_CONTEXT_ID_PREFIX + ObjectUtils.getDi
splayString(sc.getContextPath()));
    }
}
```

紧接着就是将ServletContext设置成XmlWebApplicationContext的属性，这样Spring就能在上下文里轻松拿到ServletContext了。

```
wac.setServletContext(sc);
```

接下来就是读取web.xml文件中的contextConfigLocation参数。如果没有配置就会去读WEB-INF下的applicationContext.xml文件。

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

并将值设置(就是我们的Spring配置文件的路径)进XmlWebApplicationContext中。然后就会在指定的路径加载配置文件。

```
String configLocationParam = sc.getInitParameter(CONFIG_LOCATION_PARAM);
if (configLocationParam != null) {
    wac.setConfigLocation(configLocationParam);
}
```

接下来就是customizeContext(sc, wac)方法，此方法会根据用户配置的globalInitializerClasses参数来初始化一些用户自定义的属性，一般我们不配置，所以这里什么也不做。

最后登场的就是最核心的方法了，

```
wac.refresh();
```

在这个方法里，会完成资源文件的加载、配置文件解析、Bean定义的注册、组件的初始化等核心工作，我们一探究竟。

```java
@Override
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // Prepare this context for refreshing.
        prepareRefresh();

        // Tell the subclass to refresh the internal bean factory.
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

        // Prepare the bean factory for use in this context.
        prepareBeanFactory(beanFactory);

        try {
            // Allows post-processing of the bean factory in context subclasses.
            postProcessBeanFactory(beanFactory);

            // Invoke factory processors registered as beans in the context.
            invokeBeanFactoryPostProcessors(beanFactory);

            // Register bean processors that intercept bean creation.
            registerBeanPostProcessors(beanFactory);

            // Initialize message source for this context.
            initMessageSource();

            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();

            // Initialize other special beans in specific context subclasses.
            onRefresh();

            // Check for listener beans and register them.
            registerListeners();

            // Instantiate all remaining (non-lazy-init) singletons.
            finishBeanFactoryInitialization(beanFactory);

            // Last step: publish corresponding event.
            finishRefresh();
        }

        catch (BeansException ex) {
            // Destroy already created singletons to avoid dangling resources.
            destroyBeans();

            // Reset 'active' flag.
            cancelRefresh(ex);

            // Propagate exception to caller.
            throw ex;
        }
    }
}
```

次方法是同步的，避免重复刷新，每个步骤都放在单独的方法内，流程清晰，是值得学习的地方。
这里面有个重要的方法是finishBeanFactoryInitialization(beanFactory);，里面的内容是Spring如何实例化bean，并注入依赖的，这个内容下一节讲，本节只说明Spring是如何加载class文件的。

首先就是prepareRefresh()方法。

```
protected void prepareRefresh() {
    this.startupDate = System.currentTimeMillis();

    synchronized (this.activeMonitor) {
        this.active = true;
    }

    if (logger.isInfoEnabled()) {
        logger.info("Refreshing " + this);
    }

    // Initialize any placeholder property sources in the context environment
    initPropertySources();

    // Validate that all properties marked as required are resolvable
    // see ConfigurablePropertyResolver#setRequiredProperties
    getEnvironment().validateRequiredProperties();
}
```

此方法做一些准备工作，如记录开始时间，输出日志，initPropertySources();和getEnvironment().validateRequiredProperties();一般没干什么事。

接下来就是初始化BeanFactory，是整个refresh()方法的核心，其中完成了配置文件的加载、解析、注册

```
ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
```

看看它里面都做了些什么？

```
protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
    refreshBeanFactory();
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    if (logger.isDebugEnabled()) {
        logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
    }
    return beanFactory;
}
```

首先refreshBeanFactory()：

```
protected final void refreshBeanFactory() throws BeansException {
    if (hasBeanFactory()) {
        destroyBeans();
        closeBeanFactory();
    }
    try {
        DefaultListableBeanFactory beanFactory = createBeanFactory();
        beanFactory.setSerializationId(getId());
        customizeBeanFactory(beanFactory);
        loadBeanDefinitions(beanFactory);
        synchronized (this.beanFactoryMonitor) {
            this.beanFactory = beanFactory;
        }
    }
    catch (IOException ex) {
        throw new ApplicationContextException("I/O error parsing bean definition source for " + getDi
splayName(), ex);
    }
}
```

我们看到会创建一个DefaultListableBeanFactory实例

```
DefaultListableBeanFactory beanFactory = createBeanFactory();
```

再设置一个ID

```
beanFactory.setSerializationId(getId());
```

然后设置一些自定义参数：

```
customizeBeanFactory(beanFactory);
```

这里面最重要的就是loadBeanDefinitions(beanFactory);方法了。

```
@Override
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansException, IOE
xception {
    // Create a new XmlBeanDefinitionReader for the given BeanFactory.
    XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);

    // Configure the bean definition reader with this context's
    // resource loading environment.
    beanDefinitionReader.setEnvironment(this.getEnvironment());
    beanDefinitionReader.setResourceLoader(this);
    beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));

    // Allow a subclass to provide custom initialization of the reader,
    // then proceed with actually loading the bean definitions.
    initBeanDefinitionReader(beanDefinitionReader);
    loadBeanDefinitions(beanDefinitionReader);
}
```

此方法会通过XmlBeanDefinitionReader加载bean定义。具体的实现方法是在
org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions方法中定义的。
这里设计了层层调用，有好多重载方法，主要就是加载Spring所有的配置文件(可能会有多个)，以备后面解析，注册之用。
我一路追踪到
org.springframework.beans.factory.xml.DefaultBeanDefinitionDocumentReader.doRegisterBeanDefinitions(Element root)

```
protected void doRegisterBeanDefinitions(Element root) {
    String profileSpec = root.getAttribute(PROFILE_ATTRIBUTE);
    if (StringUtils.hasText(profileSpec)) {
        Assert.state(this.environment != null, "Environment must be set for evaluating profiles");
        String[] specifiedProfiles = StringUtils.tokenizeToStringArray(
                profileSpec, BeanDefinitionParserDelegate.MULTI_VALUE_ATTRIBUTE_DELIMITERS);
        if (!this.environment.acceptsProfiles(specifiedProfiles)) {
            return;
        }
    }
    BeanDefinitionParserDelegate parent = this.delegate;
    this.delegate = createDelegate(this.readerContext, root, parent);
    preProcessXml(root);
    parseBeanDefinitions(root, this.delegate);
    postProcessXml(root);
    this.delegate = parent;
}
```

这里创建了一个BeanDefinitionParserDelegate示例，解析XML的过程就是委托它完成的，我们不关心它是怎样解析XML的，我们只关心是怎么加载类的，所以就要看parseBeanDefinitions(root, this.delegate)方法了。

```
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
    if (delegate.isDefaultNamespace(root)) {
        NodeList nl = root.getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            if (node instanceof Element) {
                Element ele = (Element) node;
                if (delegate.isDefaultNamespace(ele)) {
                    parseDefaultElement(ele, delegate);
                }
                else {
                    delegate.parseCustomElement(ele);
                }
            }
        }
    }
    else {
        delegate.parseCustomElement(root);
    }
}
```

我们看到最终解析XML元素的是delegate.parseCustomElement(ele)方法，最终会走到一下方法.

```
public BeanDefinition parseCustomElement(Element ele, BeanDefinition containingBd) {
    String namespaceUri = getNamespaceURI(ele);
    NamespaceHandler handler = this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUr
i);
    if (handler == null) {
        error("Unable to locate Spring NamespaceHandler for XML schema namespace [" + namespaceUri +
"]", ele);
        return null;
    }
    return handler.parse(ele, new ParserContext(this.readerContext, this, containingBd));
}
```

这里会根据不同的XML节点，会委托NamespaceHandlerSupport找出合适的BeanDefinitionParser，
如果我们配置了

```
<context:component-scan
    base-package="com.geeekr.service,com.geeekr.dao" />
```

那么对应BeanDefinitionParser就是org.springframework.context.annotation.ComponentScanBeanDefinitionParser，
来看看它的parse方法。

```
@Override
public BeanDefinition parse(Element element, ParserContext parserContext) {
    String[] basePackages = StringUtils.tokenizeToStringArray(element.getAttribute(BASE_PACKAGE_ATTRI
BUTE),
            ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS);

    // Actually scan for bean definitions and register them.
    ClassPathBeanDefinitionScanner scanner = configureScanner(parserContext, element);
    Set<BeanDefinitionHolder> beanDefinitions = scanner.doScan(basePackages);
    registerComponents(parserContext.getReaderContext(), beanDefinitions, element);

    return null;
}
```

不难看出这里定义了一个ClassPathBeanDefinitionScanner，通过它去扫描包中的类文件，
注意：这里是类文件而不是类，因为现在这些类还没有被加载，只是ClassLoader能找到这些class的路径而已。
到目前为止，感觉真想距离我们越来越近了。
顺着继续往下摸。进入doSacn方法里，映入眼帘的又是一大坨代码，但是我们只关心观点的部分。

```
protected Set<BeanDefinitionHolder> doScan(String... basePackages) {
    Assert.notEmpty(basePackages, "At least one base package must be specified");
    Set<BeanDefinitionHolder> beanDefinitions = new LinkedHashSet<BeanDefinitionHolder>();
    for (String basePackage : basePackages) {
        Set<BeanDefinition> candidates = findCandidateComponents(basePackage);
        for (BeanDefinition candidate : candidates) {
            ScopeMetadata scopeMetadata = this.scopeMetadataResolver.resolveScopeMetadata(candidate);
            candidate.setScope(scopeMetadata.getScopeName());
            String beanName = this.beanNameGenerator.generateBeanName(candidate, this.registry);
            if (candidate instanceof AbstractBeanDefinition) {
                postProcessBeanDefinition((AbstractBeanDefinition) candidate, beanName);
            }
            if (candidate instanceof AnnotatedBeanDefinition) {
                AnnotationConfigUtils.processCommonDefinitionAnnotations((AnnotatedBeanDefinition) ca
ndidate);
            }
            if (checkCandidate(beanName, candidate)) {
                BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(candidate, beanNam
e);
                definitionHolder = AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definiti
onHolder, this.registry);
                beanDefinitions.add(definitionHolder);
                registerBeanDefinition(definitionHolder, this.registry);
            }
        }
    }
    return beanDefinitions;
}
```

一眼就能看出是通过

```
Set<BeanDefinition> candidates = findCandidateComponents(basePackage);
```

有时候不得不佩服这些外国人起名字的功力，把扫描出来的类叫做candidates(候选人)；真是不服不行啊，这种名字真的很容易理解有不有？哈哈，貌似扯远了。继续往下看。这里只列出方法的主题部分。

```
public Set<BeanDefinition> findCandidateComponents(String basePackage) {
    Set<BeanDefinition> candidates = new LinkedHashSet<BeanDefinition>();
    try {
        String packageSearchPath = ResourcePatternResolver.CLASSPATH_ALL_URL_PREFIX +
                resolveBasePackage(basePackage) + "/" + this.resourcePattern;
        Resource[] resources = this.resourcePatternResolver.getResources(packageSearchPath);
        boolean traceEnabled = logger.isTraceEnabled();
        boolean debugEnabled = logger.isDebugEnabled();
        for (Resource resource : resources) {
            if (traceEnabled) {
                logger.trace("Scanning " + resource);
            }
            if (resource.isReadable()) {
                try {
                    MetadataReader metadataReader = this.metadataReaderFactory.getMetadataReader(reso
urce);
                    if (isCandidateComponent(metadataReader)) {
                        ScannedGenericBeanDefinition sbd = new ScannedGenericBeanDefinition(metadataR
eader);
                        sbd.setResource(resource);
                        sbd.setSource(resource);
```

先看这两句：

```
String packageSearchPath = ResourcePatternResolver.CLASSPATH_ALL_URL_PREFIX + resolveBasePackage(base
Package) + "/" + this.resourcePattern;
```

假设我们配置的需要扫描的包名为com.geeekr.service，那么packageSearchPath的值就是classpath*:com.geeekr.service/***.class，意思就是com.geeekr.service包(包括子包)下所有class文件；如果配置的是，那么packageSearchPath的值就是classpath*://**.class。这里的表达式是Spring自己定义的。Spring会根据这种表达式找出相关的class文件。

```
Resource[] resources = this.resourcePatternResolver.getResources(packageSearchPath);
```

这一句就把相关class文件加载出来了，那我们就要看看，Spring究竟是如何把class文件找到的了。
首先看看**resourcePatternResolver**的定义:

```
private ResourcePatternResolver resourcePatternResolver = new PathMatchingResourcePatternResolver();
```

进入getResources方法

```
@Override
public Resource[] getResources(String locationPattern) throws IOException {
    Assert.notNull(locationPattern, "Location pattern must not be null");
    if (locationPattern.startsWith(CLASSPATH_ALL_URL_PREFIX)) {
        // a class path resource (multiple resources for same name possible)
        if (getPathMatcher().isPattern(locationPattern.substring(CLASSPATH_ALL_URL_PREFIX.length())))
{
            // a class path resource pattern
            return findPathMatchingResources(locationPattern);
        }
        else {
            // all class path resources with the given name
            return findAllClassPathResources(locationPattern.substring(CLASSPATH_ALL_URL_PREFIX.lengt
h()));
        }
    }
    else {
        // Only look for a pattern after a prefix here
        // (to not get fooled by a pattern symbol in a strange prefix).
        int prefixEnd = locationPattern.indexOf(":") + 1;
        if (getPathMatcher().isPattern(locationPattern.substring(prefixEnd))) {
            // a file pattern
            return findPathMatchingResources(locationPattern);
        }
        else {
            // a single resource with the given name
            return new Resource[] {getResourceLoader().getResource(locationPattern)};
        }
    }
}
```

这里会先判断表达式是否以classpath*:开头。前面我们看到Spring已经给我们添加了这个头，这里当然符合条件了。接着会进入findPathMatchingResources方法。在这里又把*/.class去掉了，然后在调用getResources方法，然后在进入findAllClassPathResources方法。这里的参数只剩下包名了例如com/geeekr/service/。

```
protected Resource[] findAllClassPathResources(String location) throws IOException {
    String path = location;
    if (path.startsWith("/")) {
        path = path.substring(1);
    }
    ClassLoader cl = getClassLoader();
    Enumeration<URL> resourceUrls = (cl != null ? cl.getResources(path) : ClassLoader.getSystemResour
ces(path));
    Set<Resource> result = new LinkedHashSet<Resource>(16);
    while (resourceUrls.hasMoreElements()) {
        URL url = resourceUrls.nextElement();
        result.add(convertClassLoaderURL(url));
    }
    return result.toArray(new Resource[result.size()]);
}
```

真相大白了，Spring也是用的ClassLoader加载的class文件。一路追踪，原始的ClassLoader是Thread.currentThread().getContextClassLoader();。到此为止，就拿到class文件了。
Spring会将class信息封装成BeanDefinition，然后再放进DefaultListableBeanFactory的beanDefinitionMap中。

拿到了class文件后，就要看看Spring是如何装配bean的了，下一节，继续看。