首先来看一段代码，看过上一节的朋友肯定对这段代码并不陌生。这一段代码诠释了Spring加载bean的完整过程，包括读取配置文件，扫描包，加载类，实例化bean，注入bean属性依赖。

```
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();
        }
    }
}
```

上一节介绍了Spring是如何加载class文件的，本节主要围绕**finishBeanFactoryInitialization**(beanFactory)方法，聊聊**Spring是如何实例化bean的**，从上面代码片段中的注解不难看出，
此方法主要的任务就是实例化非懒加载的单例bean。闲话少叙，看代码。

```
protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {
    // Initialize conversion service for this context.
    if (beanFactory.containsBean(CONVERSION_SERVICE_BEAN_NAME) &&
            beanFactory.isTypeMatch(CONVERSION_SERVICE_BEAN_NAME, ConversionService.class)) {
        beanFactory.setConversionService(
                beanFactory.getBean(CONVERSION_SERVICE_BEAN_NAME, ConversionService.class));
    }

    // Initialize LoadTimeWeaverAware beans early to allow for registering their transformers early.
    String[] weaverAwareNames = beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, fal
se);
    for (String weaverAwareName : weaverAwareNames) {
        getBean(weaverAwareName);
    }

    // Stop using the temporary ClassLoader for type matching.
    beanFactory.setTempClassLoader(null);

    // Allow for caching all bean definition metadata, not expecting further changes.
    beanFactory.freezeConfiguration();

    // Instantiate all remaining (non-lazy-init) singletons.
    beanFactory.preInstantiateSingletons();
}
```

上面代码主要看最后一句**beanFactory.preInstantiateSingletons()**。

```
public void preInstantiateSingletons() throws BeansException {
    if (this.logger.isDebugEnabled()) {
        this.logger.debug("Pre-instantiating singletons in " + this);
    }
    List<String> beanNames;
    synchronized (this.beanDefinitionMap) {
        // Iterate over a copy to allow for init methods which in turn register new bean definitions.
        // While this may not be part of the regular factory bootstrap, it does otherwise work fine.
        beanNames = new ArrayList<String>(this.beanDefinitionNames);
    }
    for (String beanName : beanNames) {
        RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
        if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
            if (isFactoryBean(beanName)) {
                final FactoryBean<?> factory = (FactoryBean<?>) getBean(FACTORY_BEAN_PREFIX + beanNam
e);
                boolean isEagerInit;
                if (System.getSecurityManager() != null && factory instanceof SmartFactoryBean) {
                    isEagerInit = AccessController.doPrivileged(new PrivilegedAction<Boolean>() {
                        @Override
                        public Boolean run() {
                            return ((SmartFactoryBean<?>) factory).isEagerInit();
                        }
                    }, getAccessControlContext());
                }
                else {
                    isEagerInit = (factory instanceof SmartFactoryBean &&
                            ((SmartFactoryBean<?>) factory).isEagerInit());
                }
                if (isEagerInit) {
                    getBean(beanName);
                }
            }
            else {
                getBean(beanName);
            }
        }
    }
}
```

此方法首先将加载进来的**beanDefinitionNames**循环分析，

**如果是我们自己配置的bean就会走else中的getBean(beanName)**，接着看。

```
@Override
public Object getBean(String name) throws BeansException {
    return doGetBean(name, null, null, false);
}
```

doGetBean方法内容太多，一段一段看。

```
protected <T> T doGetBean(
        final String name, final Class<T> requiredType, final Object[] args, boolean typeCheckOnly)
        throws BeansException {

    final String beanName = transformedBeanName(name);
    Object bean;

    // Eagerly check singleton cache for manually registered singletons.
    Object sharedInstance = getSingleton(beanName);
    if (sharedInstance != null && args == null) {
        if (logger.isDebugEnabled()) {
            if (isSingletonCurrentlyInCreation(beanName)) {
                logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +
                        "' that is not fully initialized yet - a consequence of a circular referenc
e");
            }
            else {
                logger.debug("Returning cached instance of singleton bean '" + beanName + "'");
            }
        }
        bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
    }
```

这里主要看`Object sharedInstance = getSingleton(beanName)`。

```
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    Object singletonObject = this.singletonObjects.get(beanName);
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
        synchronized (this.singletonObjects) {
            singletonObject = this.earlySingletonObjects.get(beanName);
            if (singletonObject == null && allowEarlyReference) {
                ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
                if (singletonFactory != null) {
                    singletonObject = singletonFactory.getObject();
                    this.earlySingletonObjects.put(beanName, singletonObject);
                    this.singletonFactories.remove(beanName);
                }
            }
        }
    }
    return (singletonObject != NULL_OBJECT ? singletonObject : null);
}
```

这里能看到，Spring会把**实例化好的bean**存入**singletonObjects**，这是一个**ConcurrentHashMap**，

```
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>(64);
```

当然这里我们bean并未实例化过，所以这里应该也不能get出什么东西来，也就是返回null了。
if子句也就不会执行了。那么接着看else子句的内容。

```
else {
    // Fail if we're already creating this bean instance:
    // We're assumably within a circular reference.
    if (isPrototypeCurrentlyInCreation(beanName)) {
        throw new BeanCurrentlyInCreationException(beanName);
    }

    // Check if bean definition exists in this factory.
    BeanFactory parentBeanFactory = getParentBeanFactory();
    if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
        // Not found -> check parent.
        String nameToLookup = originalBeanName(name);
        if (args != null) {
            // Delegation to parent with explicit args.
            return (T) parentBeanFactory.getBean(nameToLookup, args);
        }
        else {
            // No args -> delegate to standard getBean method.
            return parentBeanFactory.getBean(nameToLookup, requiredType);
        }
    }
```

这两条验证也都不会实现，接写来就是**重点**了。

```
try {
    final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
    checkMergedBeanDefinition(mbd, beanName, args);

    // Guarantee initialization of beans that the current bean depends on.
    String[] dependsOn = mbd.getDependsOn();
    if (dependsOn != null) {
        for (String dependsOnBean : dependsOn) {
            if (isDependent(beanName, dependsOnBean)) {
                throw new BeanCreationException("Circular depends-on relationship between '" +
                        beanName + "' and '" + dependsOnBean + "'");
            }
            registerDependentBean(dependsOnBean, beanName);
            getBean(dependsOnBean);
        }
    }

    // Create bean instance.
    if (mbd.isSingleton()) {
        sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {
            @Override
            public Object getObject() throws BeansException {
                try {
                    return createBean(beanName, mbd, args);
                }
                catch (BeansException ex) {
                    // Explicitly remove instance from singleton cache: It might have been put there
                    // eagerly by the creation process, to allow for circular reference resolution.
                    // Also remove any beans that received a temporary reference to the bean.
                    destroySingleton(beanName);
                    throw ex;
                }
            }
        });
        bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
    }

    else if (mbd.isPrototype()) {
```

```
            // It's a prototype -> create a new instance.
            Object prototypeInstance = null;
            try {
                beforePrototypeCreation(beanName);
                prototypeInstance = createBean(beanName, mbd, args);
            }
            finally {
                afterPrototypeCreation(beanName);
            }
            bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
        }

        else {
            String scopeName = mbd.getScope();
            final Scope scope = this.scopes.get(scopeName);
            if (scope == null) {
                throw new IllegalStateException("No Scope registered for scope '" + scopeName + "'");
            }
            try {
                Object scopedInstance = scope.get(beanName, new ObjectFactory<Object>() {
                    @Override
                    public Object getObject() throws BeansException {
                        beforePrototypeCreation(beanName);
                        try {
                            return createBean(beanName, mbd, args);
                        }
                        finally {
                            afterPrototypeCreation(beanName);
                        }
                    }
                });
                bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);
            }
            catch (IllegalStateException ex) {
                throw new BeanCreationException(beanName,
                        "Scope '" + scopeName + "' is not active for the current thread; " +
                        "consider defining a scoped proxy for this bean if you intend to refer to it from
a singleton",
                        ex);
            }
        }
    }
```

在这里拿到**RootBeanDefinition**并**check**，并获得bean的依赖，并循环迭代实例化bean。

例如class A依赖于class B，就会先实例化B。
下面的if … else …就是真正实例化bean的地方。
其实**真正实例化bean的方法**是**createBean(beanName, mbd, args)**，
只是区分了isSingleton或isPrototype，
两者的区别在于，**单例的(Singleton)被缓存起来**，而**Prototype是不用缓存的。**
首先看一下createBean(beanName, mbd, args)。
**createBean**方法中除了做了一些实例化bean前的检查准备工作外，最核心的方法就是

```
Object beanInstance = doCreateBean(beanName, mbd, args);
```

由于这个过程涉及到的代码都是一大坨，就不贴出所有代码了。

```
BeanWrapper instanceWrapper = null;
if (mbd.isSingleton()) {
    instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
}
if (instanceWrapper == null) {
    instanceWrapper = createBeanInstance(beanName, mbd, args);
}
final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstance() : null);
Class<?> beanType = (instanceWrapper != null ? instanceWrapper.getWrappedClass() : null);
```

首先就是创建一个bean的实例且封装到BeanWrapper中，在这里bean已经实例化了。
具体的实现方法是在
`org.springframework.beans.factory.support.SimpleInstantiationStrategy.instantiate(RootBeanDefinition beanDefinition, String beanName, BeanFactory owner)`中。

```
@Override
public Object instantiate(RootBeanDefinition beanDefinition, String beanName, BeanFactory owner) {
    // Don't override the class with CGLIB if no overrides.
    if (beanDefinition.getMethodOverrides().isEmpty()) {
        Constructor<?> constructorToUse;
        synchronized (beanDefinition.constructorArgumentLock) {
            constructorToUse = (Constructor<?>) beanDefinition.resolvedConstructorOrFactoryMethod;
            if (constructorToUse == null) {
                final Class<?> clazz = beanDefinition.getBeanClass();
                if (clazz.isInterface()) {
                    throw new BeanInstantiationException(clazz, "Specified class is an interface");
                }
                try {
                    if (System.getSecurityManager() != null) {
                        constructorToUse = AccessController.doPrivileged(new PrivilegedExceptionActio
n<Constructor<?>>() {
                            @Override
                            public Constructor<?> run() throws Exception {
                                return clazz.getDeclaredConstructor((Class[]) null);
                            }
                        });
                    }
                    else {
                        constructorToUse =  clazz.getDeclaredConstructor((Class[]) null);
                    }
                    beanDefinition.resolvedConstructorOrFactoryMethod = constructorToUse;
                }
                catch (Exception ex) {
                    throw new BeanInstantiationException(clazz, "No default constructor found", ex);
                }
            }
        }
        return BeanUtils.instantiateClass(constructorToUse);
    }
    else {
        // Must generate CGLIB subclass.
        return instantiateWithMethodInjection(beanDefinition, beanName, owner);
    }
}
```

在这里不难看出实例化分两种情况，如果没有无参构造器是就生成**CGLIB子类**，否则就直接反射成实例。

```
public static <T> T instantiateClass(Constructor<T> ctor, Object... args) throws BeanInstantiationExc
eption {
    Assert.notNull(ctor, "Constructor must not be null");
    try {
        ReflectionUtils.makeAccessible(ctor);
        return ctor.newInstance(args);
    }
```

既然已经有了实例对象了，那么，**Spring是如何将bean的属性注入到bean的呢***？*
返回到上面的**doCreateBean**方法中。往下看找到**populateBean(beanName, mbd, instanceWrapper)**;，
内幕就在这里。只贴部分代码：

```
boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.DEPENDENCY_CHECK_NONE);

if (hasInstAwareBpps || needsDepCheck) {
    PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCachi
ng);
    if (hasInstAwareBpps) {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof InstantiationAwareBeanPostProcessor) {
                InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
                pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanNa
me);
                if (pvs == null) {
                    return;
                }
            }
        }
    }
    if (needsDepCheck) {
        checkDependencies(beanName, mbd, filteredPds, pvs);
    }
}
```

这里是调用**InstantiationAwareBeanPostProcessor**的具体子类的**ibp.postProcessPropertyValues**方法注入属性。
当我们使用**@Resource** 注解的时候，具体的子类是**CommonAnnotationBeanPostProcessor**；
如果使用的是**@Autowired**注解，则具体的子类是**AutowiredAnnotationBeanPostProcessor**。
此方法内是委托**InjectionMetadata**对象来完成属性注入。

```
@Override
public PropertyValues postProcessPropertyValues(
        PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String beanName) throws BeansExcep
tion {

    InjectionMetadata metadata = findAutowiringMetadata(beanName, bean.getClass());
    try {
        metadata.inject(bean, beanName, pvs);
    }
    catch (Throwable ex) {
        throw new BeanCreationException(beanName, "Injection of autowired dependencies failed", ex);
    }
    return pvs;
}
```

**findAutowiringMetadata**方法能拿到使用了特定注解的属性(Field)、方法(Method)及依赖的关系
保存到**checkedElements**集合`<Set>`里，然后再执行自己的inject方法。

```
public void inject(Object target, String beanName, PropertyValues pvs) throws Throwable {
    Collection<InjectedElement> elementsToIterate =
            (this.checkedElements != null ? this.checkedElements : this.injectedElements);
    if (!elementsToIterate.isEmpty()) {
        boolean debug = logger.isDebugEnabled();
        for (InjectedElement element : elementsToIterate) {
            if (debug) {
                logger.debug("Processing injected method of bean '" + beanName + "': " + element);
            }
            element.inject(target, beanName, pvs);
        }
    }
}
```

真正干事的还是**InjectedElement**的**inject**方法。

```
@Override
protected void inject(Object bean, String beanName, PropertyValues pvs) throws Throwable {
    Field field = (Field) this.member;
    try {
        Object value;
        if (this.cached) {
            value = resolvedCachedArgument(beanName, this.cachedFieldValue);
        }
        else {
            DependencyDescriptor desc = new DependencyDescriptor(field, this.required);
            desc.setContainingClass(bean.getClass());
            Set<String> autowiredBeanNames = new LinkedHashSet<String>(1);
            TypeConverter typeConverter = beanFactory.getTypeConverter();
            value = beanFactory.resolveDependency(desc, beanName, autowiredBeanNames, typeConverter);
            synchronized (this) {
                if (!this.cached) {
                    if (value != null || this.required) {
                        this.cachedFieldValue = desc;
                        registerDependentBeans(beanName, autowiredBeanNames);
                        if (autowiredBeanNames.size() == 1) {
                            String autowiredBeanName = autowiredBeanNames.iterator().next();
                            if (beanFactory.containsBean(autowiredBeanName)) {
                                if (beanFactory.isTypeMatch(autowiredBeanName, field.getType())) {
                                    this.cachedFieldValue = new RuntimeBeanReference(autowiredBeanNam
e);

                                }
                            }
                        }
                    }
                    else {
                        this.cachedFieldValue = null;
                    }
                    this.cached = true;
                }
            }
        }
        if (value != null) {
            ReflectionUtils.makeAccessible(field);
            field.set(bean, value);
        }
    }
    catch (Throwable ex) {
        throw new BeanCreationException("Could not autowire field: " + field, ex);
        }
    }
}
```

其实别看代码这么多，最关键的部分就是：

```
if (value != null) {
    ReflectionUtils.makeAccessible(field);
    field.set(bean, value);
}
```

在这里也就真相大白了，就是通过**JDK**反射特性，直接*set*值的。