课程目标

1、高仿真手写 Spring IOC、DI 充分实践设计模式。

2、用 30 个类搭建基本框架，满足核心功能。

内容定位

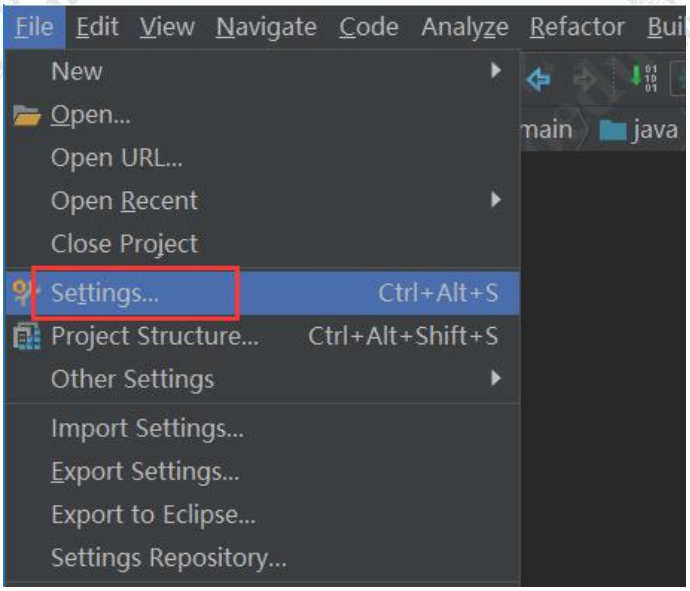
在完全掌握 Spring 系统结构、实现原理，在理解设计模式的基础上，自己动手写一个高仿真版本的 Spring 框架，以达到透彻理解 Spring 的目的，感受作者创作意图。

IDEA 集成 Lombok 插件

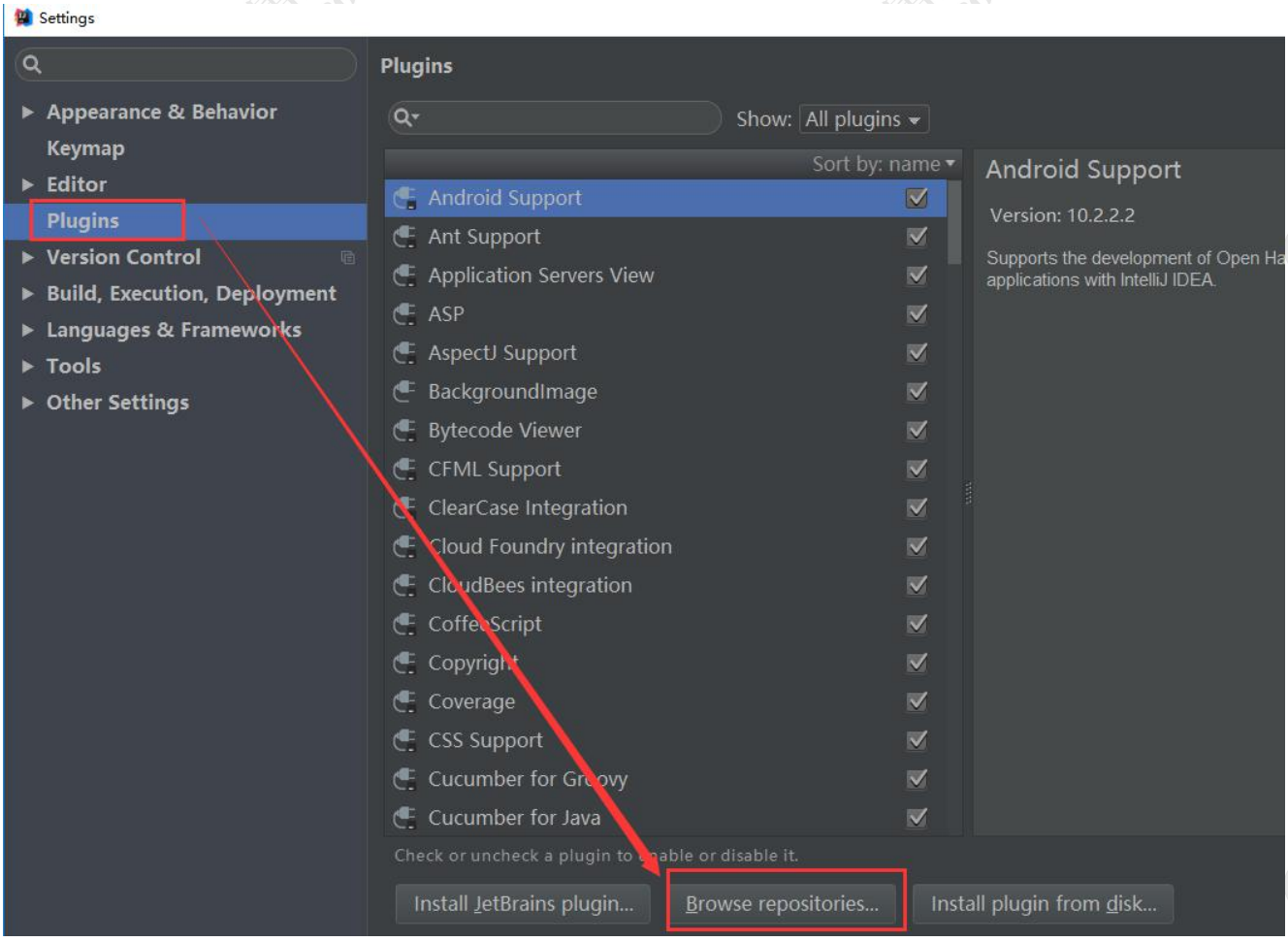
安装插件

IntelliJ IDEA 是一款非常优秀的集成开发工具，功能强大，而且插件众多。lombok 是开源的代码生成库，是一款非常实用的小工具，我们在编辑实体类时可以通过 lombok 注解减少 getter、setter 等方法的编写，在更改实体类时只需要修改属性即可，减少了很多重复代码的编写工作。

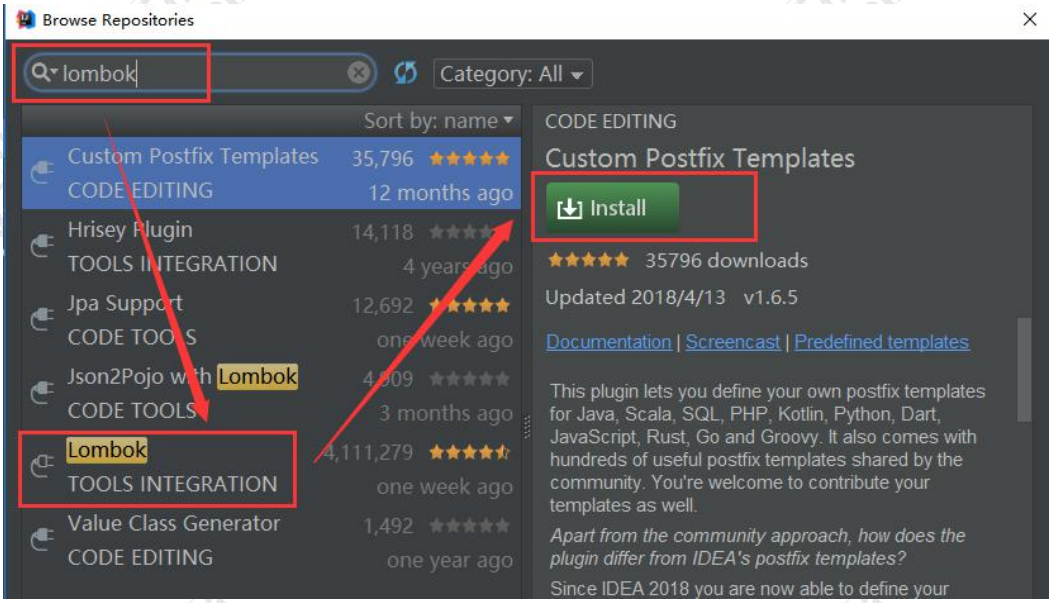
首先我们需要安装 IntelliJ IDEA 中的 lombok 插件，打开 IntelliJ IDEA 后点击菜单栏中的 File-->Settings，或者使用快捷键 Ctrl+Alt+S 进入到设置页面。



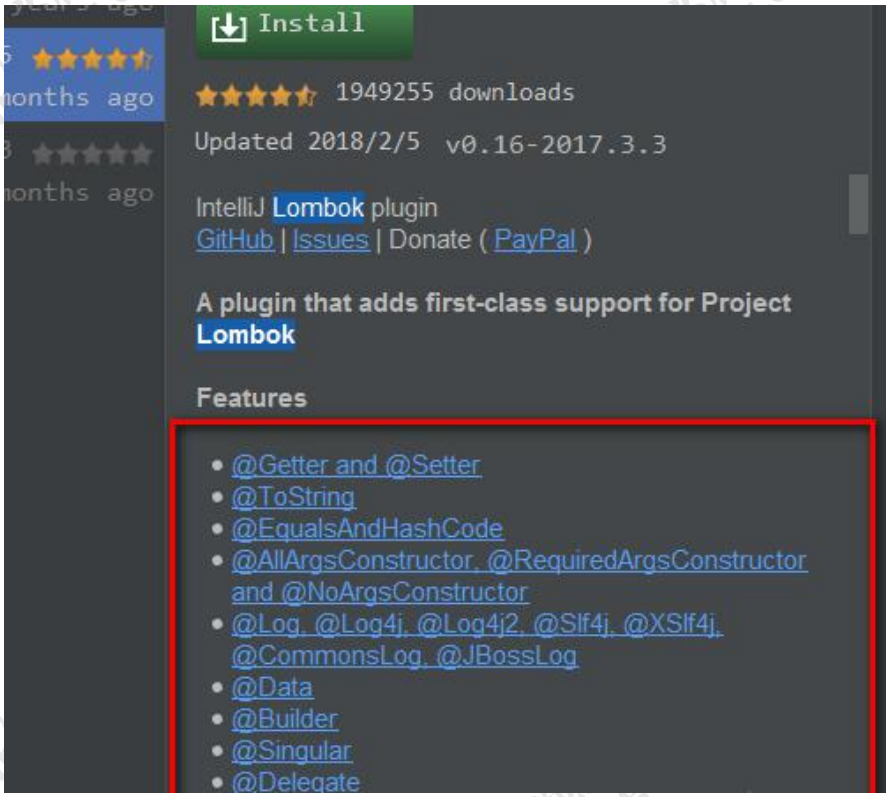
我们点击设置中的 Plugins 进行插件的安装，在右侧选择 Browse repositories...按钮。



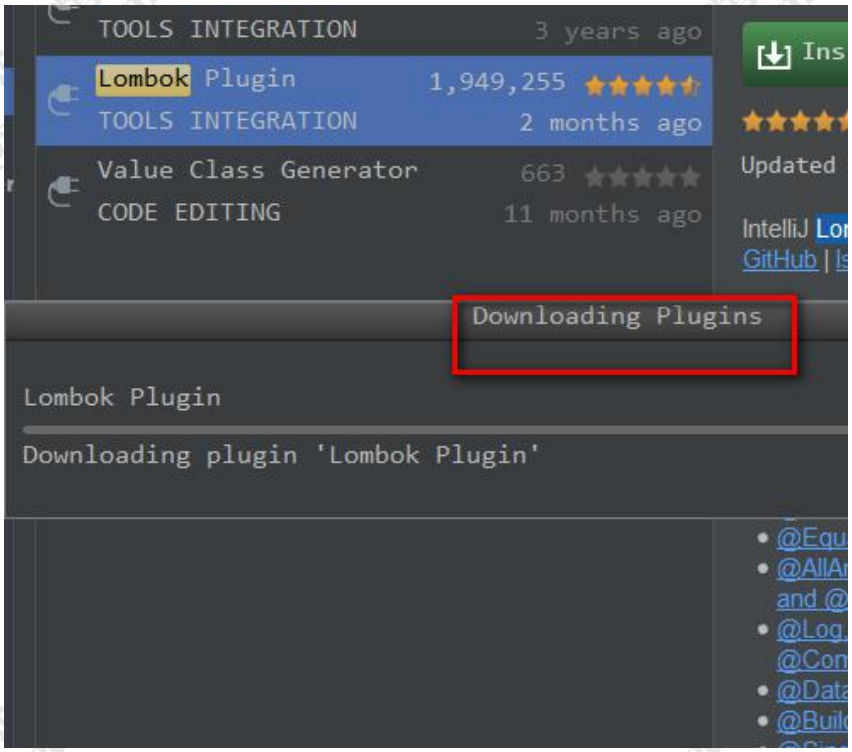
然后在搜索页面输入 lombok 变可以查询到下方的 Lombok Plugin，鼠标点击 Lombok Plugin 可在右侧看到 Install 按钮，点击该按钮便可安装。



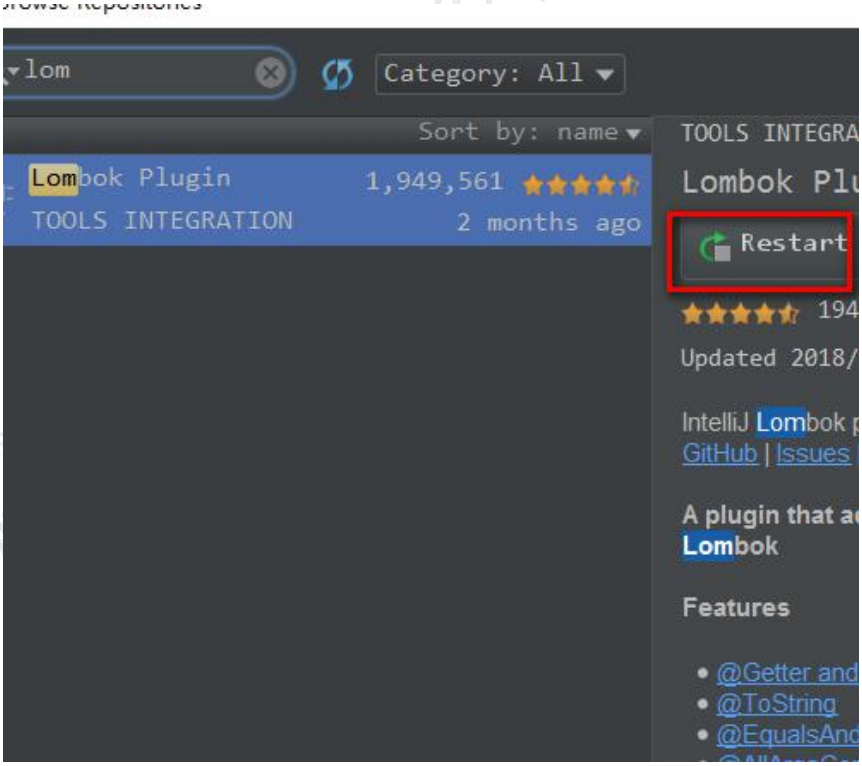
我们在安装页面可以看到 lombok 具体支持的所有注解。



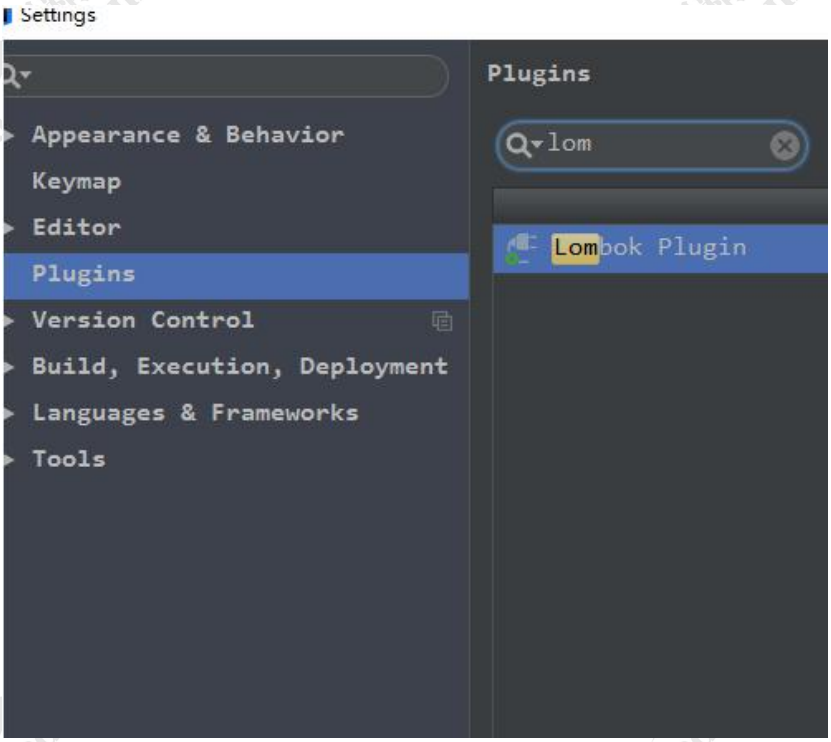
在安装过程中有 Downloading Plugins 的提示，安装过程中进度条会变化。需要提醒的是，在安装过程中一定要保证网络连接可用且良好，否则可能会安装失败。



安装成功后我们可以看到右侧的 Restart 按钮，此时可先不操作，因为我们还有后续的配置工作。

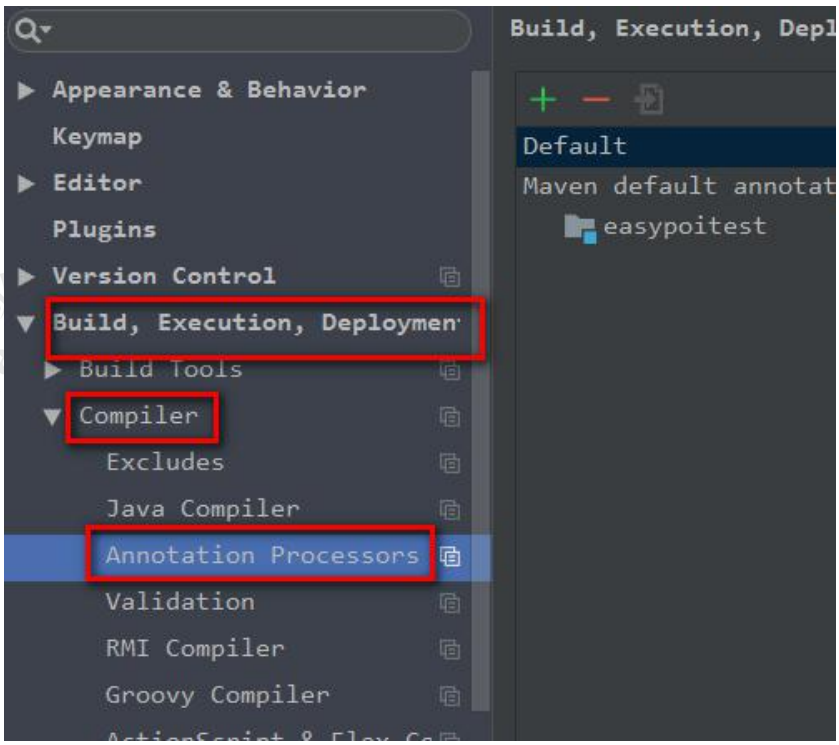


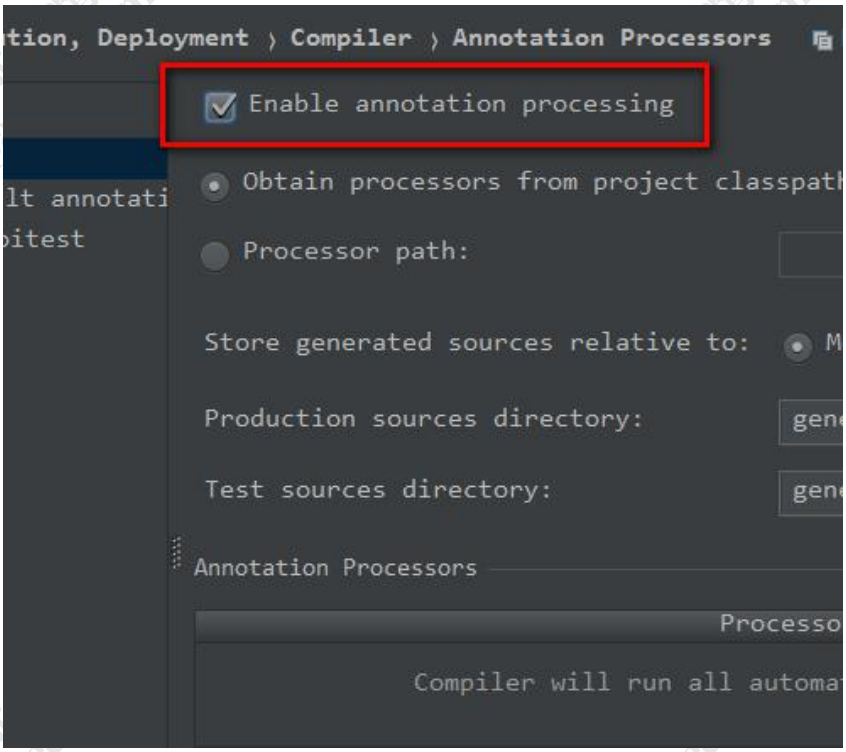
安装完成后我们再回到 Plugins，此时在右侧可以搜索到 lombok，而安装前是不行的。



配置注解处理器

同样我们在 Settings 设置页面，我们点击 Build，Execution，Deployment-->选择 Compiler-->选中 Annotation Processors，然后在右侧勾选 Enable annotation processing 即可。



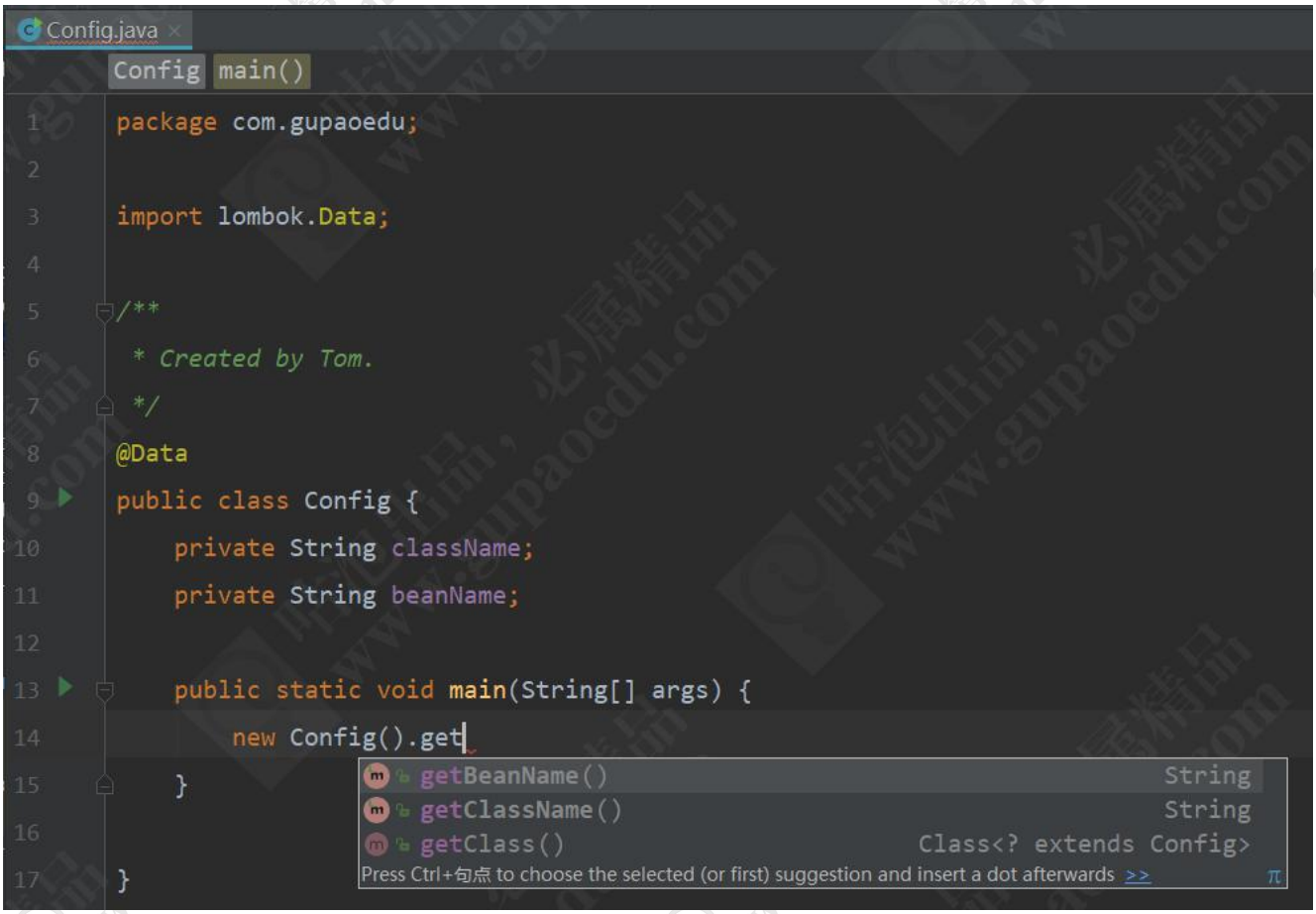


使用插件

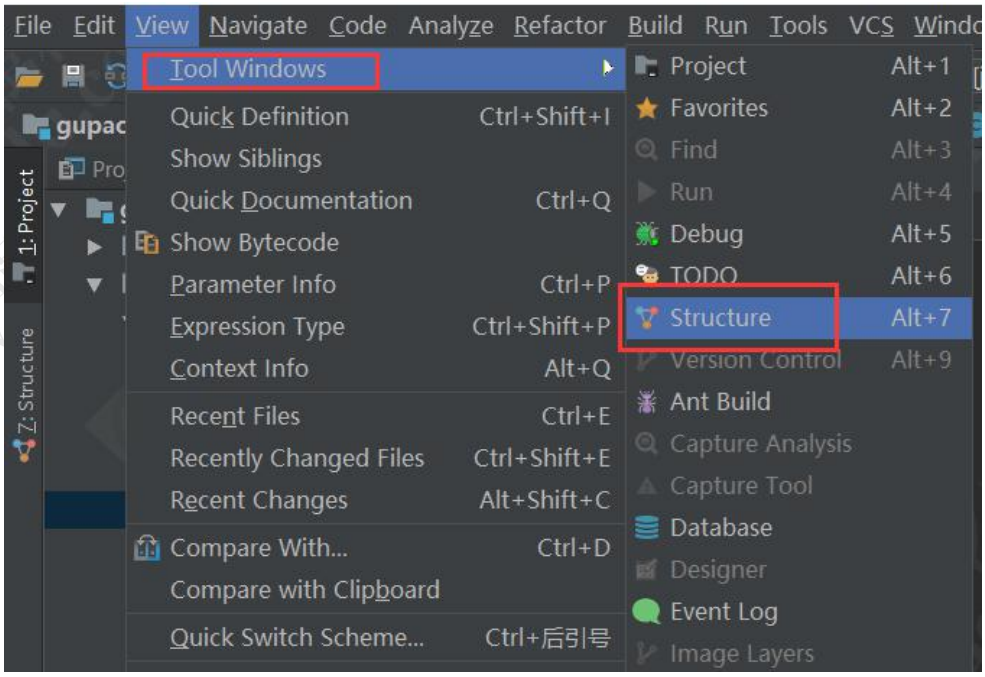
使用前我们需要说明的是安装的插件只是一个调用，就像我们使用 maven 插件一样，本机需要安装 maven 才行。我们在使用 lombok 前也需要在 pom.xml 文件中添加 lombok 的依赖。

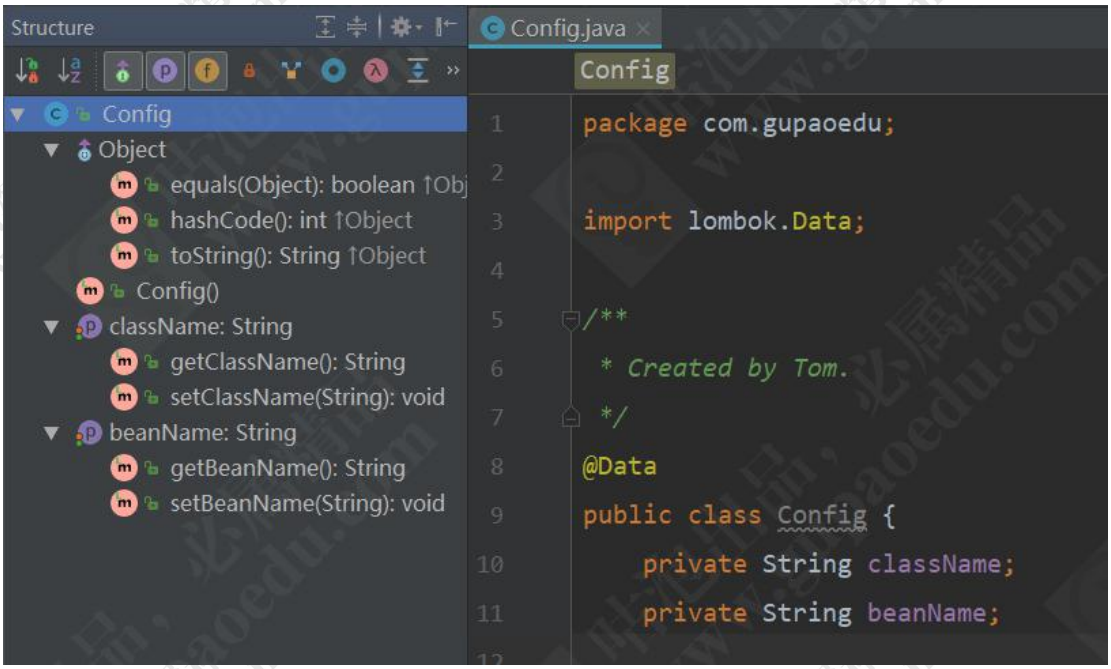
```
#托管的类扫描包路径#
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.16.10</version>
</dependency>
```

接下来我们编辑一个 Config 测试类，添加两个属性，最后在类上添加@Data 属性，这个注解可以帮我们在.class 文件中生成类中所有属性的 get/set 方法、equals、canEqual、hashCode、toString 方法等。



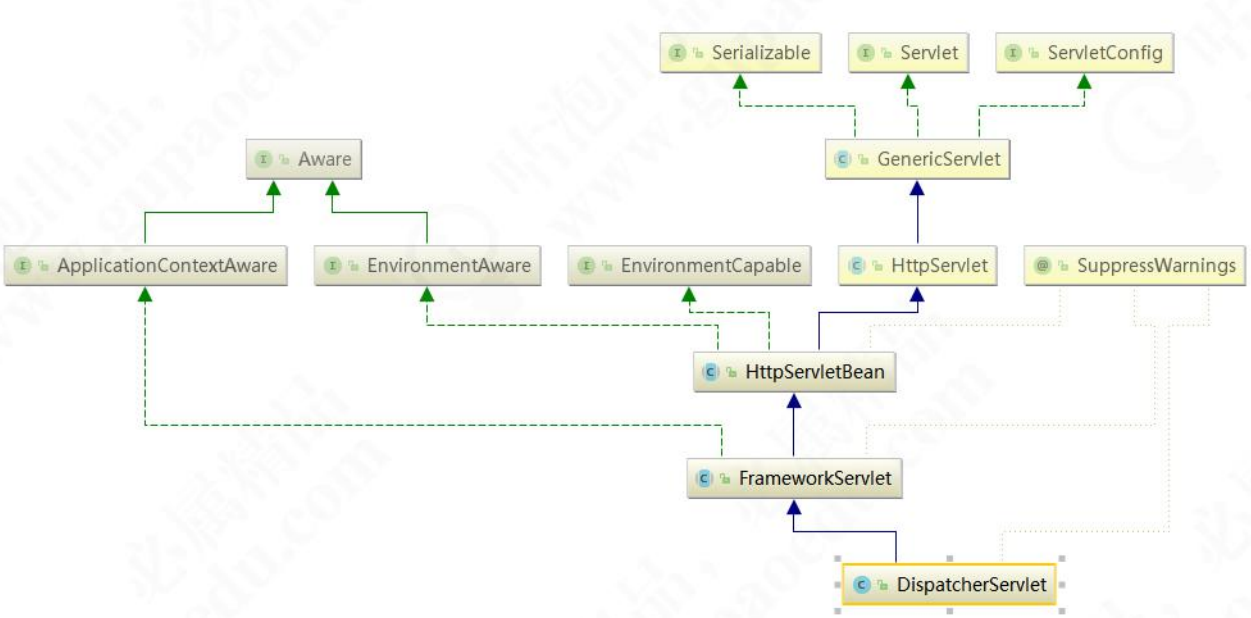
我们还可以通过下面的方式查看lombok帮我们生成的方法。在菜单栏点击View-->Tool Windows-->Structure，便可以看到类中所有的方法了，这些都是 lombok 帮我们自动生成的。





从 Servlet 到 ApplicationContext

在 300 行代码提炼 Spring 设计精华的课程中 ,我们已经了解 SpringMVC 的入口是 DispatcherSerlvet，我们实现了 DispatcherServlet 的 init()方法。在 init()方法中完成了 IOC 容器的初始化。而在我们使用 Spring 的经验中，我们见得最多的是 ApplicationContext，似乎 Spring 托管的所有实例 Bean 都可以通过调用 getBean()方法来获得。那么 ApplicationContext 又是从何而来的呢？从 Spring 源码中我们可以看到，DispatcherServlet 的类图如下：

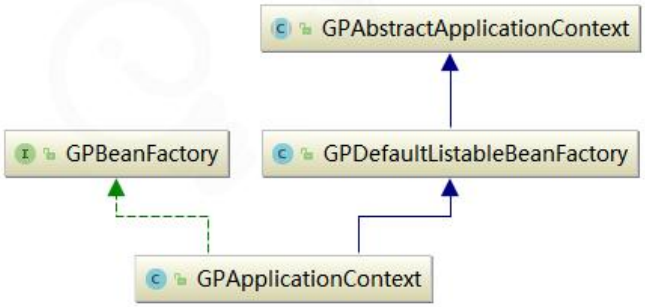


DispatcherServlet 继承了 FrameworkServlet，FrameworkServlet 继承了 HttpServletBean，HttpServletBean 继承了 HttpServlet。在 HttpServletBean 的 init()方法中调用了 FrameworkServlet 的 initServletBean()方法，在 initServletBean()方法中初始化 WebApplicationContext 实例。在 initServletBean()方法中调用了 DispatcherServlet 重写的 onRefresh()方法。在 DispatcherServlet 的 onRefresh()方法中又调用了 initStrategies()方法，初始化 SpringMVC 的九大组件。

其实，上面复杂的调用关系，我们可以简单的得出一个结论：就是在 Servlet 的 init 方法中初始化了 IOC 容器和 SpringMVC 所依赖的九大组件。

高仿真的类关系图

我们先来将自己要手写的框架类关系画出来，顺便也回顾一下我们上堂课讲过的 IOC 容器结构。



项目环境搭建

application.properties 配置

还是先从 application.properties 文件开始，用 application.properties 来代替 application.xml，具体配置如下：

```
#托管的类扫描包路径#
scanPackage=com.gupaoedu.vip.spring.demo
```

pom.xml 配置

接下来看 pom.xml 的配置，主要关注 jar 依赖：

```
<properties>
    <!-- dependency versions -->
    <servlet.api.version>2.4</servlet.api.version>
</properties>

<dependencies>
    <!-- requied start -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>${servlet.api.version}</version>
        <scope>provided</scope>
    </dependency>
    <!-- requied end -->

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.16.10</version>
    </dependency>

    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.7.25</version>
    </dependency>

    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.3</version>
    </dependency>

</dependencies>
```

web.xml 配置

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:javaee="http://java.sun.com/xml/ns/javaee"
   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
```

```
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
   version="2.4">
   <display-name>Gupao Spring Application</display-name>

   <servlet>
      <servlet-name>gupaomvc</servlet-name>

<servlet-class>com.gupaoedu.vip.spring.formework.webmvc.servlet.GPDispatcherServlet</servlet-cla
ss>
      <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>classpath:application.properties</param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>gupaomvc</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>

</web-app>
```

GPDispatcherServlet 实现

```
package com.gupaoedu.vip.spring.formework.webmvc.servlet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
/**
 * Created by Tom
 */
//Servlet 只是作为一个 MVC 的启动入口
public class GPDispatcherServlet extends HttpServlet {
    @Override
    public void init(ServletConfig config) throws ServletException {
    }
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        this.doPost(req,resp);
```

```
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
    }
}
```

IOC 顶层结构设计

annotation（自定义配置）模块

Annotation 的代码实现我们还是沿用 mini 版本的不变，复制过来便可。

@GPService

```
package com.gupaoedu.vip.spring.formework.annotation;
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
/**
 * 业务逻辑,注入接口
 * @author Tom
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface GPService {
   String value() default "";
}
```

@Autowired

```
package com.gupaoedu.vip.spring.formework.annotation;
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
/**
 * 自动注入
 * @author Tom
```

```
 */
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface GPAutowired {
   String value() default "";
}
```

@Controller

```
package com.gupaoedu.vip.spring.formework.annotation;
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
/**
 * 页面交互
 * @author Tom
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface GPController {
   String value() default "";
}
```

@RequestMapping

```
package com.gupaoedu.vip.spring.formework.annotation;
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
/**
 * 请求url
 * @author Tom
 */
@Target({ElementType.METHOD,ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface GPRequestMapping {
   String value() default "";
}
```

@RequestParam

```
package com.gupaoedu.vip.spring.formework.annotation;
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
/**
 * 请求参数映射
 * @author Tom
 */
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface GPRequestParam {
   String value() default "";
}
```

core（顶层接口）模块

GPFactoryBean

```
package com.gupaoedu.vip.spring.formework.core;
/**
 * Created by Tom
 */
public interface GPFactoryBean {
}
```

GPBeanFactory

```
package com.gupaoedu.vip.spring.formework.core;

/**
 * 单例工厂的顶层设计
 * Created by Tom.
 */
public interface GPBeanFactory {
    /**
     * 根据 beanName 从 IOC 容器中获得一个实例 Bean
     * @param beanName
     * @return
     */
    Object getBean(String beanName) throws Exception;
```

```
    public Object getBean(Class<?> beanClass) throws Exception;

}
```

beans（配置封装）模块

GPBeanDefinition

```
package com.gupaoedu.vip.spring.formework.beans.config;
/**
 * Created by Tom.
 */
//用来存储配置文件中的信息
//相当于保存在内存中的配置
public class GPBeanDefinition {
    private String beanClassName;
    private boolean lazyInit = false;
    private String factoryBeanName;
    public String getBeanClassName() {
        return beanClassName;
    }
    public void setBeanClassName(String beanClassName) {
        this.beanClassName = beanClassName;
    }
    public boolean isLazyInit() {
        return lazyInit;
    }
    public void setLazyInit(boolean lazyInit) {
        this.lazyInit = lazyInit;
    }
    public String getFactoryBeanName() {
        return factoryBeanName;
    }
    public void setFactoryBeanName(String factoryBeanName) {
        this.factoryBeanName = factoryBeanName;
    }
}
```

GPBeanWrapper

```
package com.gupaoedu.vip.spring.formework.beans;

/**
 * Created by Tom.
 */
```

```
public class GPBeanWrapper {

    private Object wrappedInstance;
    private Class<?> wrappedClass;

    public GPBeanWrapper(Object wrappedInstance){
        this.wrappedInstance = wrappedInstance;
    }

    public Object getWrappedInstance(){
        return this.wrappedInstance;
    }

    // 返回代理以后的 Class
    // 可能会是这个 $Proxy0
    public Class<?> getWrappedClass(){
        return this.wrappedInstance.getClass();
    }
}
```

context（IOC 容器）模块

GPAbstractApplicationContext

```
package com.gupaoedu.vip.spring.formework.context.support;

/**
 * IOC 容器实现的顶层设计
 * Created by Tom.
 */
public abstract class GPAbstractApplicationContext {
    //受保护，只提供给子类重写
    public void refresh() throws Exception {}
}
```

GPDefaultListableBeanFactory

```
package com.gupaoedu.vip.spring.formework.beans.support;

import com.gupaoedu.vip.spring.formework.beans.config.GPBeanDefinition;
import com.gupaoedu.vip.spring.formework.context.support.GPAbstractApplicationContext;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

/**
```

```
 * Created by Tom
 */
public class GPDefaultListableBeanFactory extends GPAbstractApplicationContext{

    //存储注册信息的 BeanDefinition
    protected final Map<String, GPBeanDefinition> beanDefinitionMap = new ConcurrentHashMap<String,
GPBeanDefinition>();
}
```

GPApplicationContext

```
package com.gupaoedu.vip.spring.formework.context;

import com.gupaoedu.vip.spring.formework.annotation.GPAutowired;
import com.gupaoedu.vip.spring.formework.annotation.GPController;
import com.gupaoedu.vip.spring.formework.annotation.GPService;
import com.gupaoedu.vip.spring.formework.beans.GPBeanWrapper;
import com.gupaoedu.vip.spring.formework.beans.config.GPBeanPostProcessor;
import com.gupaoedu.vip.spring.formework.core.GPBeanFactory;
import com.gupaoedu.vip.spring.formework.beans.config.GPBeanDefinition;
import com.gupaoedu.vip.spring.formework.beans.support.GPBeanDefinitionReader;
import com.gupaoedu.vip.spring.formework.beans.support.GPDefaultListableBeanFactory;

import java.lang.reflect.Field;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.ConcurrentHashMap;

/**
 * 按之前源码分析的套路，IOC、DI、MVC、AOP
 *
 * Created by Tom.
 */
public class GPApplicationContext extends GPDefaultListableBeanFactory implements GPBeanFactory {

    private String [] configLoactions;
    private GPBeanDefinitionReader reader;

    //单例的 IOC 容器缓存
    private Map<String,Object> singletonObjects = new ConcurrentHashMap<String, Object>();
    //通用的 IOC 容器
    private Map<String,GPBeanWrapper> factoryBeanInstanceCache = new ConcurrentHashMap<String,
GPBeanWrapper>();
```

```
    public GPApplicationContext(String... configLoactions){
        this.configLoactions = configLoactions;
        try {
            refresh();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }


    @Override
    public void refresh() throws Exception{
        //1、定位，定位配置文件
        reader = new GPBeanDefinitionReader(this.configLoactions);

        //2、加载配置文件，扫描相关的类，把它们封装成 BeanDefinition
        List<GPBeanDefinition> beanDefinitions = reader.loadBeanDefinitions();

        //3、注册，把配置信息放到容器里面(伪 IOC 容器)
        doRegisterBeanDefinition(beanDefinitions);

        //4、把不是延时加载的类，有提前初始化
        doAutowrited();
    }

    //只处理非延时加载的情况
    private void doAutowrited() {
        for (Map.Entry<String, GPBeanDefinition> beanDefinitionEntry :
super.beanDefinitionMap.entrySet()) {
            String beanName = beanDefinitionEntry.getKey();
            if(!beanDefinitionEntry.getValue().isLazyInit()) {
                try {
                    getBean(beanName);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }

    private void doRegisterBeanDefinition(List<GPBeanDefinition> beanDefinitions) throws Exception
{

        for (GPBeanDefinition beanDefinition: beanDefinitions) {
```

```
            if(super.beanDefinitionMap.containsKey(beanDefinition.getFactoryBeanName())){
                throw new Exception("The “" + beanDefinition.getFactoryBeanName() + "” is exists!!");
            }
            super.beanDefinitionMap.put(beanDefinition.getFactoryBeanName(),beanDefinition);
        }
        //到这里为止，容器初始化完毕
    }
    public Object getBean(Class<?> beanClass) throws Exception {
        return getBean(beanClass.getName());
    }

    //依赖注入，从这里开始，通过读取 BeanDefinition 中的信息
    //然后，通过反射机制创建一个实例并返回
    //Spring 做法是，不会把最原始的对象放出去，会用一个 BeanWrapper 来进行一次包装
    //装饰器模式：
    //1、保留原来的 OOP 关系
    //2、我需要对它进行扩展，增强（为了以后 AOP 打基础）
    public Object getBean(String beanName) throws Exception {

        return null;
    }

    public String[] getBeanDefinitionNames() {
        return this.beanDefinitionMap.keySet().toArray(new
String[this.beanDefinitionMap.size()]);
    }

    public int getBeanDefinitionCount(){
        return this.beanDefinitionMap.size();
    }

    public Properties getConfig(){
        return this.reader.getConfig();
    }
}
```

GPBeanDefinitionReader

```
package com.gupaoedu.vip.spring.formework.beans.support;

import com.gupaoedu.vip.spring.formework.beans.config.GPBeanDefinition;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
```

```
import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;

/**
 * Created by Tom.
 */

//用对配置文件进行查找，读取、解析
public class GPBeanDefinitionReader {

    private List<String> registyBeanClasses = new ArrayList<String>();

    private Properties config = new Properties();

    //固定配置文件中的 key，相对于 xml 的规范
    private final String SCAN_PACKAGE = "scanPackage";

    public GPBeanDefinitionReader(String... locations){
        //通过 URL 定位找到其所对应的文件，然后转换为文件流
        InputStream is =
this.getClass().getClassLoader().getResourceAsStream(locations[0].replace("classpath:",""));
        try {
            config.load(is);
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            if(null != is){
                try {
                    is.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        doScanner(config.getProperty(SCAN_PACKAGE));
    }

    private void doScanner(String scanPackage) {
        //转换为文件路径，实际上就是把.替换为/就 OK 了
        URL url = this.getClass().getClassLoader().getResource("/" +
scanPackage.replaceAll("\\.","/"));
```

```
        File classPath = new File(url.getFile());
        for (File file : classPath.listFiles()) {
            if(file.isDirectory()){
                doScanner(scanPackage + "." + file.getName());
            }else{
                if(!file.getName().endsWith(".class")){ continue;}
                String className = (scanPackage + "." + file.getName().replace(".class",""));
                registyBeanClasses.add(className);
            }
        }
    }

    public Properties getConfig(){
        return this.config;
    }

    //把配置文件中扫描到的所有的配置信息转换为 GPBeanDefinition 对象，以便于之后 IOC 操作方便
    public List<GPBeanDefinition> loadBeanDefinitions(){
        List<GPBeanDefinition> result = new ArrayList<GPBeanDefinition>();
        try {
            for (String className : registyBeanClasses) {
                Class<?> beanClass = Class.forName(className);
                if(beanClass.isInterface()) { continue; }


result.add(doCreateBeanDefinition(toLowerFirstCase(beanClass.getSimpleName()),beanClass.getName(
)));

                Class<?> [] interfaces = beanClass.getInterfaces();
                for (Class<?> i : interfaces) {
                    result.add(doCreateBeanDefinition(i.getName(),beanClass.getName()));
                }

            }
        }catch (Exception e){
            e.printStackTrace();
        }
        return result;
    }


    //把每一个配信息解析成一个 BeanDefinition
    private GPBeanDefinition doCreateBeanDefinition(String factoryBeanName,String beanClassName){
        GPBeanDefinition beanDefinition = new GPBeanDefinition();
```

```
        beanDefinition.setBeanClassName(beanClassName);
        beanDefinition.setFactoryBeanName(factoryBeanName);
        return beanDefinition;
    }

    //如果类名本身是小写字母，确实会出问题
    //但是我要说明的是：这个方法是我自己用，private 的
    //传值也是自己传，类也都遵循了驼峰命名法
    //默认传入的值，存在首字母小写的情况，也不可能出现非字母的情况

    //为了简化程序逻辑，就不做其他判断了，大家了解就 OK
    //其实用写注释的时间都能够把逻辑写完了
    private String toLowerFirstCase(String simpleName) {
        char [] chars = simpleName.toCharArray();
        //之所以加，是因为大小写字母的 ASCII 码相差 32，
        // 而且大写字母的 ASCII 码要小于小写字母的 ASCII 码
        //在 Java 中，对 char 做算学运算，实际上就是对 ASCII 码做算学运算
        chars[0] += 32;
        return String.valueOf(chars);
    }

}
```

GPApplicationContextAware

```
package com.gupaoedu.vip.spring.formework.context;
/**
 * 通过解耦方式获得 IOC 容器的顶层设计
 * 后面将通过一个监听器去扫描所有的类，只要实现了此接口，
 * 将自动调用 setApplicationContext()方法，从而将 IOC 容器注入到目标类中
 * Created by Tom.
 */
public interface GPApplicationContextAware {
    void setApplicationContext(GPApplicationContext applicationContext);
}
```

完成 DI 依赖注入功能

在之前的源码分析中，我们已经了解到，依赖注入的入口是从 getBean()方法开始的，前面的 IOC 手写部分基本流程已通。先在 GPApplicationContext 中定义好 IOC 容器，一个是 GPBeanWrapper，一个是单例对象缓存

```
/**
```

```
 * Created by Tom.
 */
public class GPApplicationContext extends GPDefaultListableBeanFactory implements GPBeanFactory {
    private String [] configLocations;

    private GPBeanDefinitionReader reader;

    //用来保证注册式单例的容器
    private Map<String,Object> singletonBeanCacheMap = new HashMap<String, Object>();

    //用来存储所有的被代理过的对象
    private Map<String,GPBeanWrapper> beanWrapperMap = new ConcurrentHashMap<String,
GPBeanWrapper>();
    ...
}
```

从 getBean()开始

下面，我们从完善 getBean()方法开始：

```
    //依赖注入，从这里开始，通过读取 BeanDefinition 中的信息
    //然后，通过反射机制创建一个实例并返回
    //Spring 做法是，不会把最原始的对象放出去，会用一个 BeanWrapper 来进行一次包装
    //装饰器模式:
    //1、保留原来的 OOP 关系
    //2、我需要对它进行扩展，增强（为了以后 AOP 打基础）
    @Override
    public Object getBean(String beanName) {

        GPBeanDefinition beanDefinition = this.beanDefinitionMap.get(beanName);
        try{
            //生成通知事件
            GPBeanPostProcessor beanPostProcessor = new GPBeanPostProcessor();
            Object instance = instantiateBean(beanDefinition);
            if(null == instance){ return  null;}
            //在实例初始化以前调用一次
            beanPostProcessor.postProcessBeforeInitialization(instance,beanName);
            GPBeanWrapper beanWrapper = new GPBeanWrapper(instance);
            this.beanWrapperMap.put(beanName,beanWrapper);
            //在实例初始化以后调用一次
            beanPostProcessor.postProcessAfterInitialization(instance,beanName);
            populateBean(beanName,instance);

            //通过这样一调用，相当于给我们自己留有了可操作的空间
            return this.beanWrapperMap.get(beanName).getWrappedInstance();
```

```
        }catch (Exception e){
//            e.printStackTrace();
            return null;
        }
    }

    private void populateBean(String beanName,Object instance){
        Class clazz = instance.getClass();
        //不是所有牛奶都叫特仑苏
        if(!(clazz.isAnnotationPresent(GPController.class) ||
                clazz.isAnnotationPresent(GPService.class))){
            return;
        }
        Field [] fields = clazz.getDeclaredFields();

        for (Field field : fields) {
            if (!field.isAnnotationPresent(GPAutowired.class)){ continue; }

            GPAutowired autowired = field.getAnnotation(GPAutowired.class);
            String autowiredBeanName = autowired.value().trim();
            if("".equals(autowiredBeanName)){
                autowiredBeanName = field.getType().getName();
            }
            field.setAccessible(true);
            try {
field.set(instance,this.beanWrapperMap.get(autowiredBeanName).getWrappedInstance());

            } catch (IllegalAccessException e) {
//                e.printStackTrace();
            }

        }
    }
    //传一个 BeanDefinition，就返回一个实例 Bean
    private Object instantiateBean(GPBeanDefinition beanDefinition){
        Object instance = null;
        String className = beanDefinition.getBeanClassName();
        try{

            //因为根据 Class 才能确定一个类是否有实例
            if(this.singletonBeanCacheMap.containsKey(className)){
                instance = this.singletonBeanCacheMap.get(className);
            }else{
                Class<?> clazz = Class.forName(className);
```

```
                instance = clazz.newInstance();

                this.singletonBeanCacheMap.put(beanDefinition.getFactoryBeanName(),instance);
            }

            return instance;
        }catch (Exception e){
            e.printStackTrace();
        }

        return null;
    }
}
```

GPBeanPostProcessor

```
package com.gupaoedu.vip.spring.formework.beans.config;

/**
 * Created by Tom.
 */
public class GPBeanPostProcessor {

    //为在 Bean 的初始化前提供回调入口
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws Exception {
        return bean;
    }

    //为在 Bean 的初始化之后提供回调入口
    public Object postProcessAfterInitialization(Object bean, String beanName) throws Exception {
        return bean;
    }
}
```

至此，DI 部分就完成了。