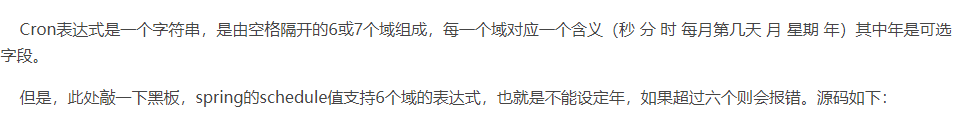
# Spring学习地址

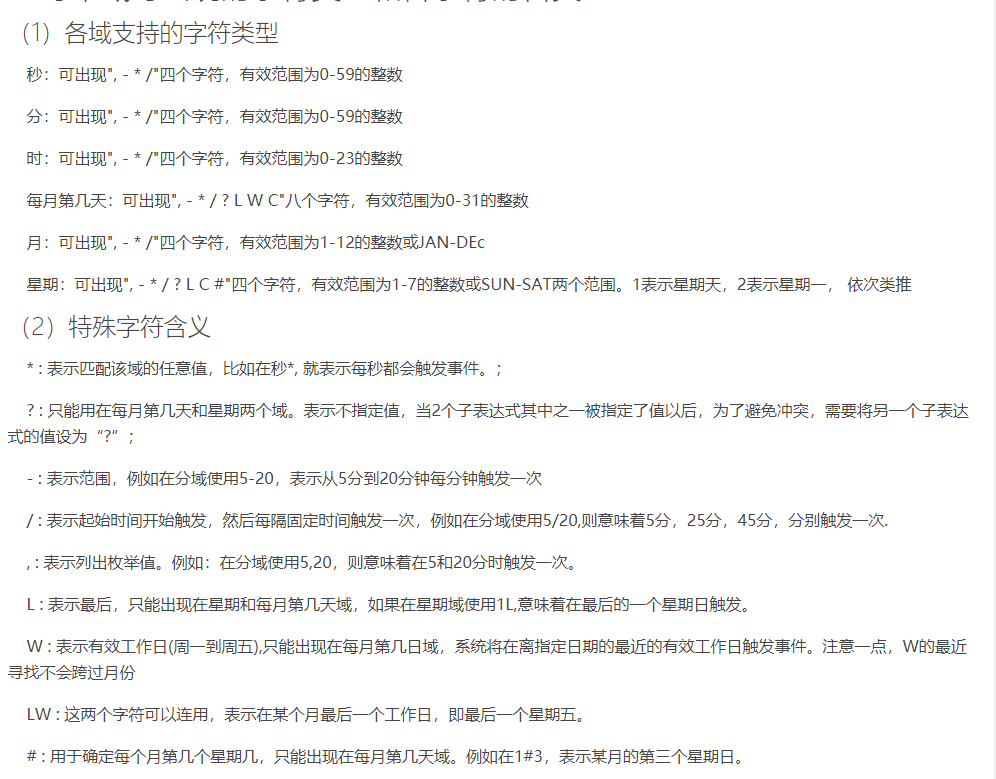
<https://www.w3cschool.cn/wkspring/yqdx1mm5.html>

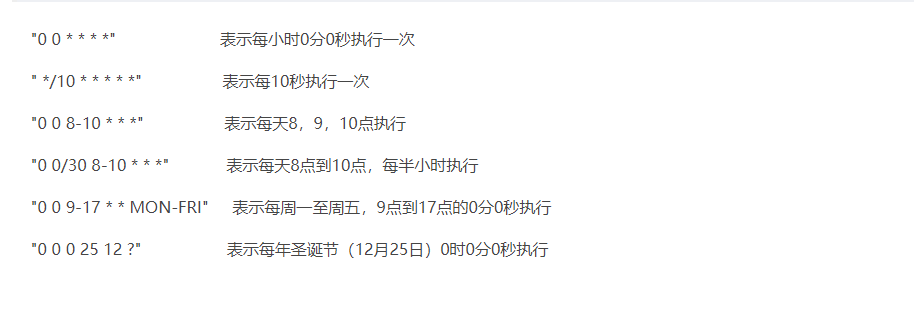
# schedule定时任务

启动时添加@EnableScheduling注解，在定时任务的方法上添加@Scheduled注解。

## cron表达式









# Filter过滤器

过滤器是基于函数回调。

## 方式一：FilterConfig

|  |
| --- |
| @Configuration  public class FilterConfig {    @Bean  public FilterRegistrationBean<LogCostFilter> registFilter() {  FilterRegistrationBean<LogCostFilter> registration = new FilterRegistrationBean<LogCostFilter>();  registration.setFilter(new LogCostFilter());  registration.addUrlPatterns("/\*");  registration.setName("LogCostFilter");  registration.setOrder(1);  return registration;  }    } |

首先将编写一个配置类，将实现Filter的类放在这个配置中。

|  |
| --- |
| public class LogCostFilter implements Filter {  public void init(FilterConfig filterConfig) throws ServletException {    System.out.println("过滤器初始化");  }    @Override  public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {  long start = System.currentTimeMillis();  filterChain.doFilter(servletRequest,servletResponse);  System.out.println("Execute cost="+(System.currentTimeMillis()-start));  System.out.println("有人访问，过滤器一执行");  }    @Override  public void destroy() {    }    } |

这个类必须实现Filter才行。

## 方式二：注解

直接在实现Filter接口的类上添加注解，

|  |
| --- |
| @WebFilter(urlPatterns = "/\*", filterName = "logFilter2")  public class LogCostFilter2 implements Filter {  public void init(FilterConfig filterConfig) throws ServletException {    System.out.println("过滤器初始化");  }    @Override  public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {  long start = System.currentTimeMillis();  filterChain.doFilter(servletRequest,servletResponse);  System.out.println("Execute cost="+(System.currentTimeMillis()-start));  System.out.println("有人访问，过滤器二执行");  }    @Override  public void destroy() {    }    } |

启动添加注解@ServletComponentScan("com.thunisoft.jy.exchange.process.test5")

# interceptor拦截器

拦截器是基于java反射机制。

配置：

|  |
| --- |
| @SuppressWarnings("deprecation")  @Configuration  public class InterceptorConfig extends WebMvcConfigurerAdapter {    @Override  public void addInterceptors(InterceptorRegistry registry) {  registry.addInterceptor(new LogCostInterceptor()).addPathPatterns("/\*\*");  super.addInterceptors(registry);  }  } |

拦截器:

|  |
| --- |
| public class LogCostInterceptor implements HandlerInterceptor {  long start = System.currentTimeMillis();  @Override  public boolean preHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o) throws Exception {  System.out.println("拦截器====preHandle");  return true;  }    @Override  public void postHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, ModelAndView modelAndView) throws Exception {  System.out.println("拦截器====postHandle");  }    @Override  public void afterCompletion(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, Exception e) throws Exception {  System.out.println("拦截器====afterCompletion");  }    } |

拦截器拦截到请求,首先执行preHandle方法,然后等请求处理完成后依次执行postHandle,afterCompletion.

最后过滤器执行.

但是百度查的是过滤器在拦截器之前执行。这个应该是在启动时先执行过滤器，再执行拦截器。

# spring Event自定义事件

ApplicationListener<ApplicationEvent>

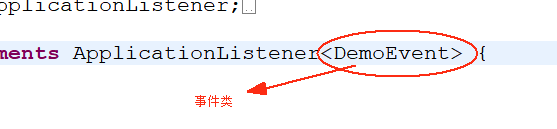
applicationContext.publishEvent()

主要是这两个

ApplicationListener就是一个监听的接口,实现这个接口,并将这个类注入到容器中.

|  |
| --- |
| @Component  Public class DemoEventListener implements ApplicationListener<DemoEvent> {  @Async  @Override  public void onApplicationEvent(DemoEvent event) {  System.out.println("注册成功，发送确认邮件为：" + event.getEmail()+",消息摘要为:"+event.getMsg());  }  } |

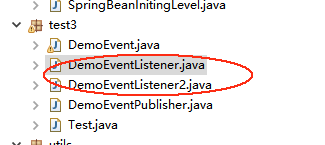
将事件类放在



监听已经做完了,现在只要applicationContext.publishEvent(new DemoEvent(this, msg,mail));他就能监听到这个.

DemoEvent要继承ApplicationEvent.

监听可以是多个:



这两个都能监听到这个消息.

## applicationContext.publishEvent()

这个publishEvent()方法是ApplicationEventPublisher的。因为applicationContext继承了这个接口，

## ApplicationEventPublisher

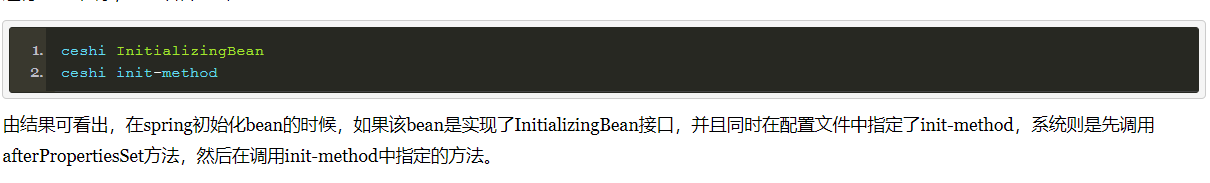
这个类中的方法会通知与事件相匹配的监听器，这些监听器可能是Spring的，也有可能是特定的监听器。

## ApplicationListener<ApplicationEvent>下面介绍。

# 类(接口)介绍

## InitializingBean--接口

InitializingBean接口为bean提供了初始化方法的方式，它只包括afterPropertiesSet方法，凡是继承该接口的类，在初始化bean的时候会执行该方法。

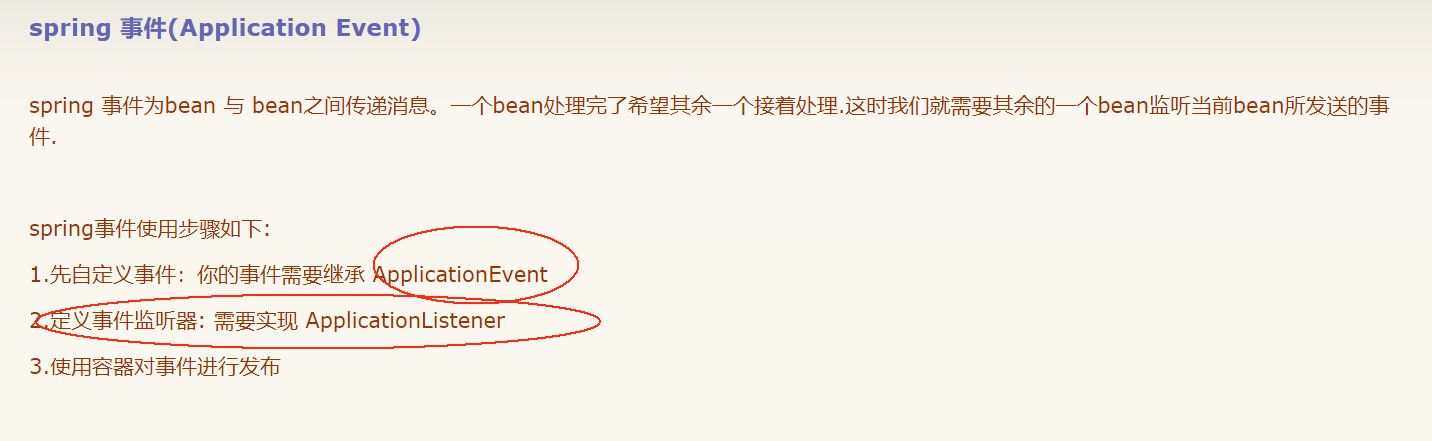


实例化bean和依赖注入都是在AbstractBeanFactory入口的，实际是在AbstractAutowireCapableBeanFactory这个类中实现的。具体细节看相关的类（接口）。

## ApplicationListener<ApplicationEvent>

applicationContext中有个方法publishEvent,传入的对象是继承了ApplicationEvent类的对象.

## ApplicationEvent



# Bean实例化

## AbstractBeanFactory

现在还理解不了。

## AbstractAutowireCapableBeanFactory

createBean方法判断是否可以实例化，然后调用doCreateBean方法，这个方法主要的作用就是：实例化bean，属性依赖注入。

# BeanFactory与ApplicationContext

BeanFactory是Spring里面最底层的接口，提供了最简单的容器的功能，只提供了实例化对象和拿对象的功能。

ApplicationContext

应用上下文，继承BeanFactory，他是Spring的一个更高级的容器。

# FactoryBean

# Aop实现原理

动态代理

## 应用场景

1. 记录日志
2. 监控方法运行时间
3. 权限控制
4. 缓存优化--第一次调用查询数据库，将查询结果放到缓存中一份，再次调用该方法时直接从缓存中取数据。
5. 事务管理

这里注意一点，如何保证数据库和缓存(Redis)数据的一致性。

# Spring能做些什么

1. 能根据配置文件创建以及组装对象之间的依赖关系
2. 面向切面编程，能够帮助我们无偶和的实现日志记录，性能统计，安全控制
3. 简单的管理数据库事务，使用的时候我们只需要获取连接，执行sql
4. 还能与第三方数据库实现无缝对接，并且自己也提供了一套JDBC访问模板
5. 还能与第三方web框架无缝对接，而且自己也提供了一套SpringMvc框架
6. 还能与其他框架进行整合，例如缓存，消息中间件等

# 事务管理

编程式事务管理

声明式事务管理

基于xml配置文件

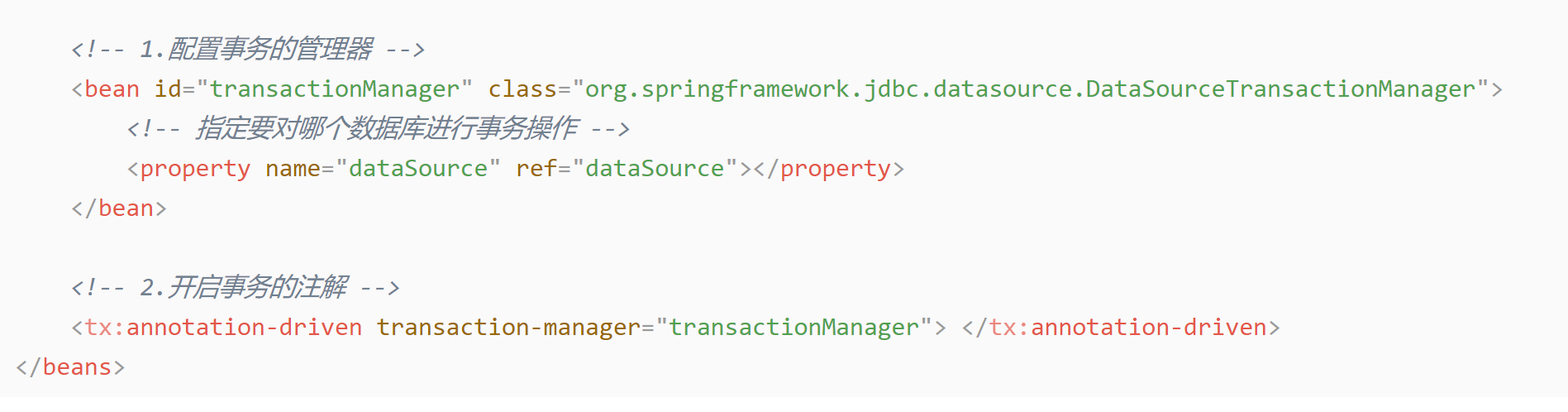
配置事务管理器

配置事务的增强

配置切入点和切面

基于注解方式

要在配置文件中开启事务的注解



在使用事务的方法所在的类上添加@Transactional注解

# 事务传播机制

<https://blog.csdn.net/qq_26323323/article/details/81908955>

前提条件是save方法调用了delete方法

Required

Requires\_new

Supports

not\_supported

Mandatory

Never

Nested

REQUIRED

如果有事务则加入事务，如果没有事务，则创建一个新的（默认值）

将这两个事务的传播机制都改为

@Transactional(propagation=Propagation.REQUIRED)

Save方法执行，执行delete方法时发现有事务，继续使用该事务。

NOT\_SUPPORTED

不为当前方法开启事务

将这两个事务的传播机制都改为

@Transactional(propagation=Propagation.NOT\_SUPPORTED)

直接执行save方法不开启事务，delete方法也不开事务直接执行。

REQUIRES\_NEW

不管是否存在事务，都创建一个新的事物，原来的方法挂起，新的方法执行完 毕后再执行老的事务

将save的事务机制改为REQUIRED，delete方法的事务改为REQUIRES\_NEW

执行save方法的时候创建一个事务，执行save方法。当天事务挂起，执行delete 方法，提交delete事务，提交save事务。

MANDATORY

必须在一个已有的事务中执行，否则报错

将save的事务机制改为NOT\_SUPPORTED，delete的事务机制改为MANDATORY

执行save方法的时候没有事务直接执行，当执行delete方法的时候因为没有 事务直接报错

NEVER

必须在一个没有的事务中执行，否则报错--刚好与MANDATORY相反

SUPPORTS

取决于调用方法是否有事务。

将save方法的事务改为REQUIRED，delete方法的事务改为SUPPORTS

执行save方法的时候创建一个事务，当执行delete方法时就会复用save方法 的事务，然后一起提交

将save方法的事务改为NOT\_SUPPORTED，delete方法的事务仍然为SUPPORTS

执行save方法的时候不开启事务直接执行，那么到了delete方法的时候也不 开启事务直接执行。

NESTED

如果当前存在事务，则再创建一个嵌套事务，在嵌套事务内执行，最后一起提 交，其本质还是在一个事务中。如果没事务，则创建一个事务

将save方法的事务改为REQUIRED，delete方法的事务改为NESTED

执行sava方法时创建一个事务，当执行到delete方法时会创建一个嵌套事务， 然后作为一次事务提交

将save方法的事务改为NOT\_SUPPORTED，delete方法的事务仍然为NESTED

执行save方式的时候没有事务直接执行（执行完毕默认自动提交），在执行 delete方法的时候创建一个事务，执行完后，提交该事务。