原理实现上：过滤器基于回调实现，而拦截器基于动态代理；

控制粒度上：过滤器和拦截器都能够实现对请求的拦截功能，但是在拦截的粒度上有较大的差异，拦截器对访问控制的粒度更细；

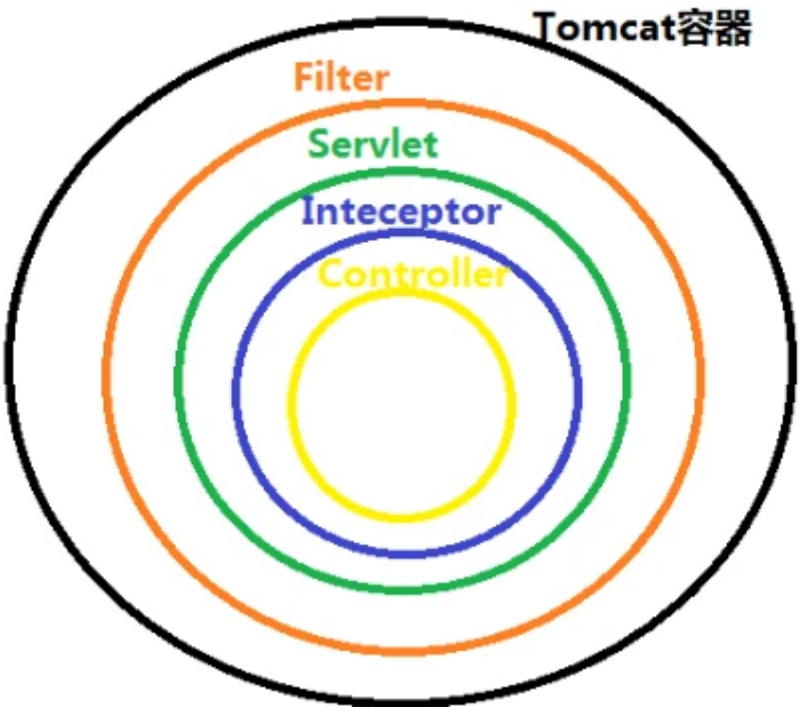
使用场景上：拦截器往往用于权限检查、日志记录等，过滤器主要用于过滤请求中无效参数，安全校验；

依赖容器上：过滤器依赖于Servlet容器，局限于web，而拦截器依赖于Spring框架，能够使用Spring框架的资源，不仅限于web；

触发时机上：过滤器在Servlet前后执行，拦截器在handler前后执行，现在大多数web应用基于Spring，拦截器更细；

流重复读取：通过重写HttpServletRequestWrapper实现，此方法不能用在文件上传上，文件上传实现思路先保存至本地，在将文件路径写入请求属性中，然后再业务中通过请求属性获取文件。

一、过滤器和拦截器的区别



1、过滤器和拦截器触发时机不一样，过滤器是在请求进入容器后，但请求进入servlet之前进行预处理的。请求结束返回也是，是在servlet处理完后，返回给前端之前。

2、拦截器可以获取IOC容器中的各个bean，而过滤器就不行，因为拦截器是spring提供并管理的，spring的功能可以被拦截器使用，在拦截器里注入一个service，可以调用业务逻辑。而过滤器是JavaEE标准，只需依赖servlet api ，不需要依赖spring。

3、过滤器的实现基于回调函数。而拦截器（代理模式）的实现基于反射

4、Filter是依赖于Servlet容器，属于Servlet规范的一部分，而拦截器则是独立存在的，可以在任何情况下使用。

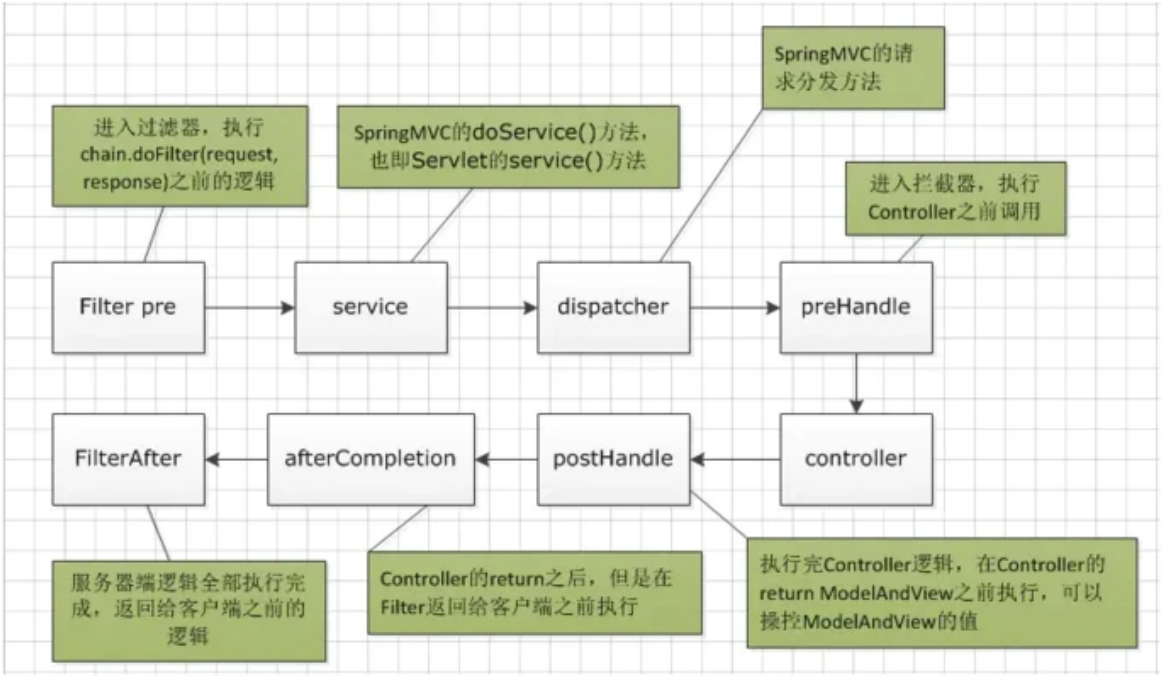
5、Filter的执行由Servlet容器回调完成，而拦截器通常通过动态代理（反射）的方式来执行。

6、Filter的生命周期由Servlet容器管理，而拦截器则可以通过IoC容器来管理，因此可以通过注入等方式来获取其他Bean的实例，因此使用会更方便。

过滤器和拦截器非常相似，但是它们有很大的区别
最简单明了的区别就是过滤器可以修改request，而拦截器不能
过滤器需要在servlet容器中实现，拦截器可以适用于javaEE，javaSE等各种环境

拦截器可以调用IOC容器中的各种依赖，而过滤器不能
过滤器只能在请求的前后使用，而拦截器可以详细到每个方法

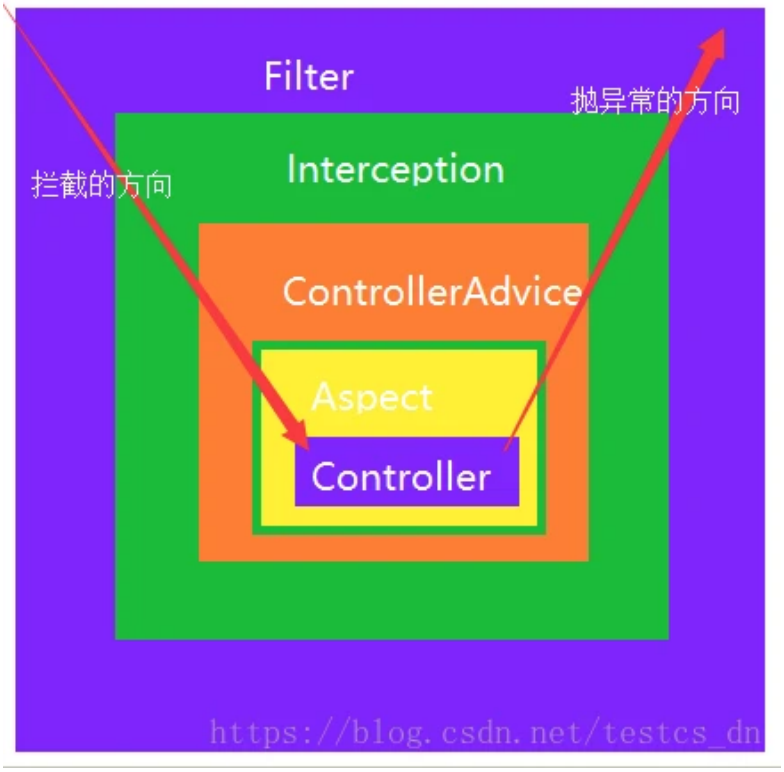
过滤器就是筛选出你要的东西，比如requeset中你要的那部分
拦截器在做安全方面用的比较多，比如终止一些流程



过滤器（Filter）：可以拿到原始的http请求，但是拿不到你请求的控制器和请求控制器中的方法的信息。

拦截器（Interceptor）：可以拿到你请求的控制器和方法，却拿不到请求方法的参数。

切片（Aspect）：可以拿到方法的参数，但是却拿不到http请求和响应的对象



二、过滤器

两种方式：

1、使用spring boot提供的FilterRegistrationBean注册Filter

2、使用原生servlet注解定义Filter

两种方式的本质都是一样的，都是去FilterRegistrationBean注册自定义Filter

方式一：（使用spring boot提供的FilterRegistrationBean注册Filter ）

①、先定义Filter：

```
import javax.servlet.*;
import java.io.IOException;
@component
public class MyFilter implements Filter {

    @Override public void init(FilterConfig filterConfig) throws ServletException {

    }

    @Override public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException { // do something 处理request 或response
     // doFilter()方法中的servletRequest参数的类型是ServletRequest，需要转换为HttpServletRequest类型方便调用某些方法
      System.out.println("filter1"); // 调用filter链中的下一个filter
```

```
    HttpServletRequest request = (HttpServletRequest) servletRequest;
        HttpServletResponse response = (HttpServletResponse) servletResponse;

        String ip = request.getRemoteAddr();
        String url = request.getRequestURL().toString();
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Date d = new Date();
        String date = sdf.format(d);

        System.out.printf("%s %s 访问了 %s%n", date, ip, url);

        filterChain.doFilter(request, response);
    }

    @Override public void destroy() {

    }
}
```

②、注册自定义Filter

```
@Configuration
public class FilterConfig {

    @Bean
    public FilterRegistrationBean registrationBean() {
        FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean(new
MyFilter());
        filterRegistrationBean.addUrlPatterns("/*");
        return filterRegistrationBean;
    }
}
```

方式二：（使用原生servlet注解定义Filter ）

```
// 注入spring容器
// @Component // 定义filterName 和过滤的url
@WebFilter(filterName = "my2Filter" ,urlPatterns = "/*")
public class My2Filter implements Filter {
    @Override public void init(FilterConfig filterConfig) throws ServletException {

    }
    @Override public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        System.out.println("filter2");
    }
    @Override public void destroy() {
```

```
    }
}
```

这里直接用@WebFilter就可以进行配置，同样，可以设置url匹配模式，过滤器名称等。这里需要注意一点的是@WebFilter这个注解是Servlet3.0的规范，并不是Spring boot提供的。除了这个注解以外，我们还需在启动类中加另外一个注解：@ServletComponetScan，指定扫描的包。

三、拦截器的配置

实现拦截器可以通过继承 HandlerInterceptorAdapter类也可以通过实现HandlerInterceptor这个接口。另外，如果preHandle方法return true，则继续后续处理。

首先我们实现拦截器类：

```
@Component
public class LogCostInterceptor implements HandlerInterceptor {
    long start = System.currentTimeMillis();
    @Override public boolean preHandle(HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse, Object o) throws Exception {
        start = System.currentTimeMillis(); return true;
    }

    @Override public void postHandle(HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse, Object o, ModelAndView
modelAndView) throws Exception {
        System.out.println("Interceptor cost="+(System.currentTimeMillis()-start));
    }

    @Override public void afterCompletion(HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse, Object o, Exception e) throws Exception
{
    }
}

public class AuthInterceptor extends HandlerInterceptorAdapter {
    private static final Log log = LogFactory.getLog(AuthInterceptor.class);
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        if (request.getParameterMap().containsKey("hello")) {
            return true;
        }
        return false;
    }
}
```

我们还需要实现HandlerInterceptor这个接口，这个接口包括三个方法，preHandle是请求执行前执行的，postHandler是请求结束执行的，但只有preHandle方法返回true的时候才会执行，afterCompletion是视图渲染完成后才执行，同样需要preHandle返回true，该方法通常用于清理资源等工作。除了实现上面的接口外，我们还需对其进行配置：

```
@Configuration
@EnableWebMvc  //支持MVC配置
public class InterceptorConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {

    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new HandlerInterceptorTest())
                .addPathPatterns("/*")
                .excludePathPatterns("/login");
    }

    // 配置不被拦截的静态资源
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/swagger-ui.html")
                .addResourceLocations("classpath:/META-INF/resources/");

        registry.addResourceHandler("/webjars/**")
                .addResourceLocations("classpath:/META-INF/resources/webjars/");

    }
}

@Configuration
public class InterceptorConfig extends WebMvcConfigurerAdapter {

    @Override public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LogCostInterceptor())
                .addPathPatterns("/**");
        super.addInterceptors(registry);
    }
}
```

拦截器是在DispatcherServlet这个servlet中执行的，因此所有的请求最先进入Filter，最后离开Filter。其顺序如下。

Filter->Interceptor.preHandle->Handler->Interceptor.postHandle->Interceptor.afterCompletion->Filter

拦截器应用场景

拦截器本质上是面向切面编程（AOP），符合横切关注点的功能都可以放在拦截器中来实现，主要的应用场景包括：

- 登录验证，判断用户是否登录。
- 权限验证，判断用户是否有权限访问资源，如校验token
- 日志记录，记录请求操作日志（用户ip，访问时间等），以便统计请求访问量。
- 处理cookie、本地化、国际化、主题等。
- 性能监控，监控请求处理时长等。

通用行为：读取cookie得到用户信息并将用户对象放入请求，从而方便后续流程使用，还有如提取Locale、Theme信息等，只要是多个处理器都需要的即可使用拦截器实现）

过滤器应用场景

- 过滤敏感词汇（防止sql注入）
- 设置字符编码
- URL级别的权限访问控制
- 压缩响应信息

Java三大器：过滤器-监听器-拦截器对比			
2016年10月25日整理，个人观点，仅供参考			
	过滤器（Filter）	监听器（Listener）	拦截器（Interceptor）
关注的点	web请求	系统级别参数、对象	Action（部分web请求）
如何实现的	函数回调	事件	Java反射机制（动态代理）
应用场景	设置字符编码	统计网站在线人数	拦截未登录用户
	URL级别的权限访问控制	清除过期session	审计日志
	过滤敏感词汇		
	压缩响应信息		
是否依赖servlet容器	依赖		不依赖
Servlet提供的支持	Filter接口	ServletContextListener抽象接口	
		HttpSessionListener抽象接口	
Spring提供的支持			HandlerInterceptorAdapter类
			HandlerInterceptor接口
级别	系统级	系统级	非系统级
注意：拦截器只能拦截部分web请求，这句话怎么理解呢？个人理解是这样的：拦截器的拦截，是基于Java反射机制实现的，拦截的对象只能是实现了接口的类，而不能拦截url这种链接。			