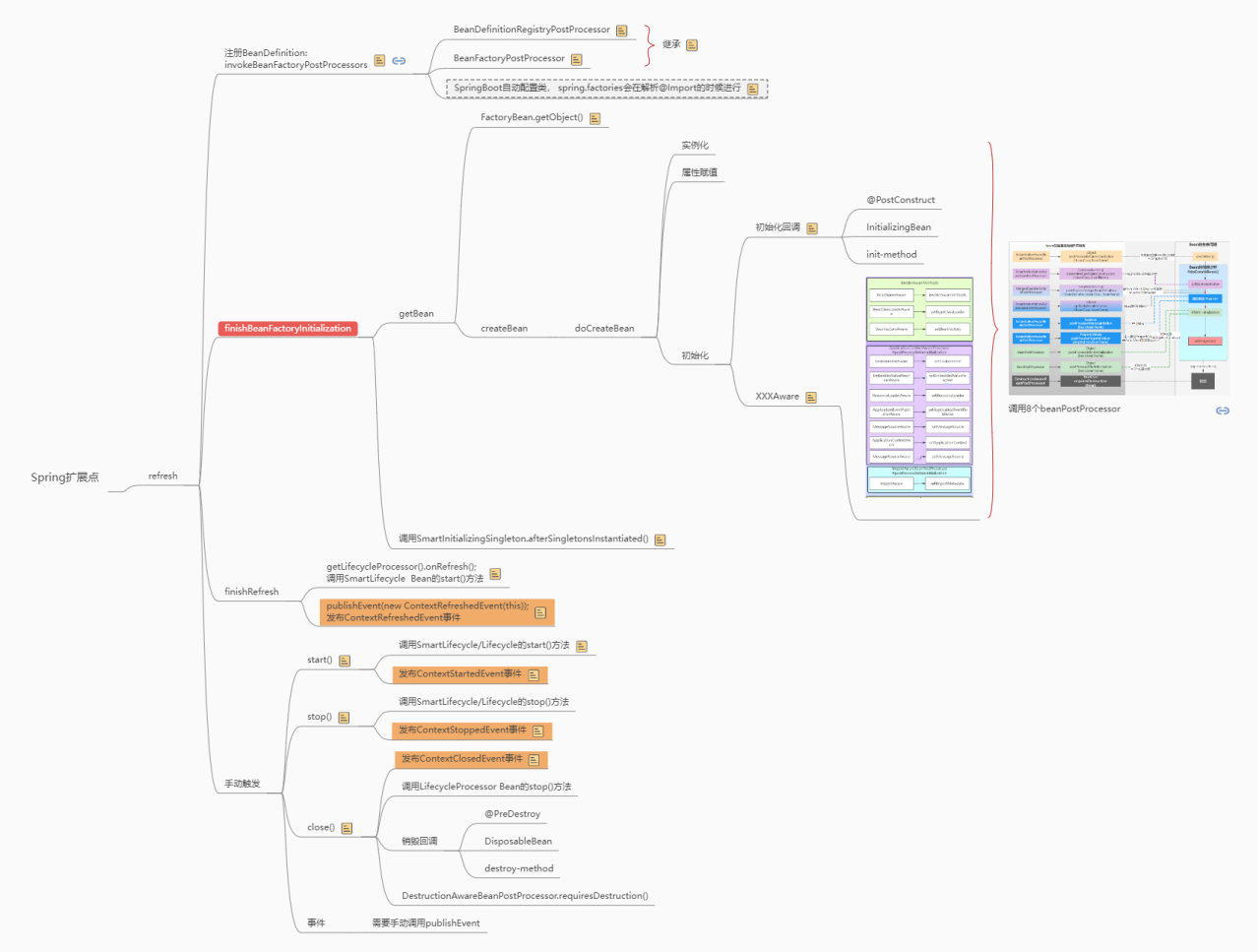
Spring IOC容器扩展点全景：深入探索与实践演练



Springloc之BeanDefinition注册过程的扩展点详解

动态注册BeanDefinition有几种方式?

好处： 可以在运行时动态决定bean的属性、类型、构造函数等定义信息

- 比如有些bean在定义期间无法确定是否注册bean，需要在运行时动态决定；
- 比如有些bean是接口—接口不能实例化，需要在运行时动态决定他的类型；
- 比如想让bean的顺序放在最后；
- ...

1. **BeanDefinitionRegistryPostProcessor**
2. **BeanFactoryPostProcessor**

```java
package com.xushu.extensions.beandefinition;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.BeanDefinitionRegistry;
import org.springframework.beans.factory.support.BeanDefinitionRegistryPostProcessor;
import org.springframework.stereotype.Component;

@Component
public class MyBeanDefinitionRegistryPostProcessor implements BeanDefinitionRegistryPostProcessor {

    @Override
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) {
        // 动态注册beanDefinition
         BeanDefinitionBuilder builder = BeanDefinitionBuilder.genericBeanDefinition(XushuService.class);
         BeanDefinition beanDefinition = builder.getBeanDefinition();
        // 动态注入属性
        beanDefinition.getPropertyValues().add("age",18);
        // 动态设置定义信息
        beanDefinition.setLazyInit(true);
        //beanDefinition.setScope();
        //beanDefinition.setInitMethodName();
        //...

        // 动态设置构造函数
        //beanDefinition.getConstructorArgumentValues().addIndexedArgumentValue(0,..);


         registry.registerBeanDefinition("xushuService3", beanDefinition);
    }

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {

    }
```

```
37
38 }
```

3. @import—ImportBeanDefinitionRegistrar

注意: 1. 必须要结合@Import ， 单独配置为bean不会起作用!

2.ImportBeanDefinitionRegistrar不是一个bean， 没有bean的生命周期， 没有依赖注入功能。

但是! 它有一个优势， 注意它有一个importingClassMetadata参数， 这个参数可以获取@Import注解所在类的其他注解信息， 比如@MapperScan根据包创建beandefinition 。 这是BeanDefinitionRegistryPostProcessor不具备的!

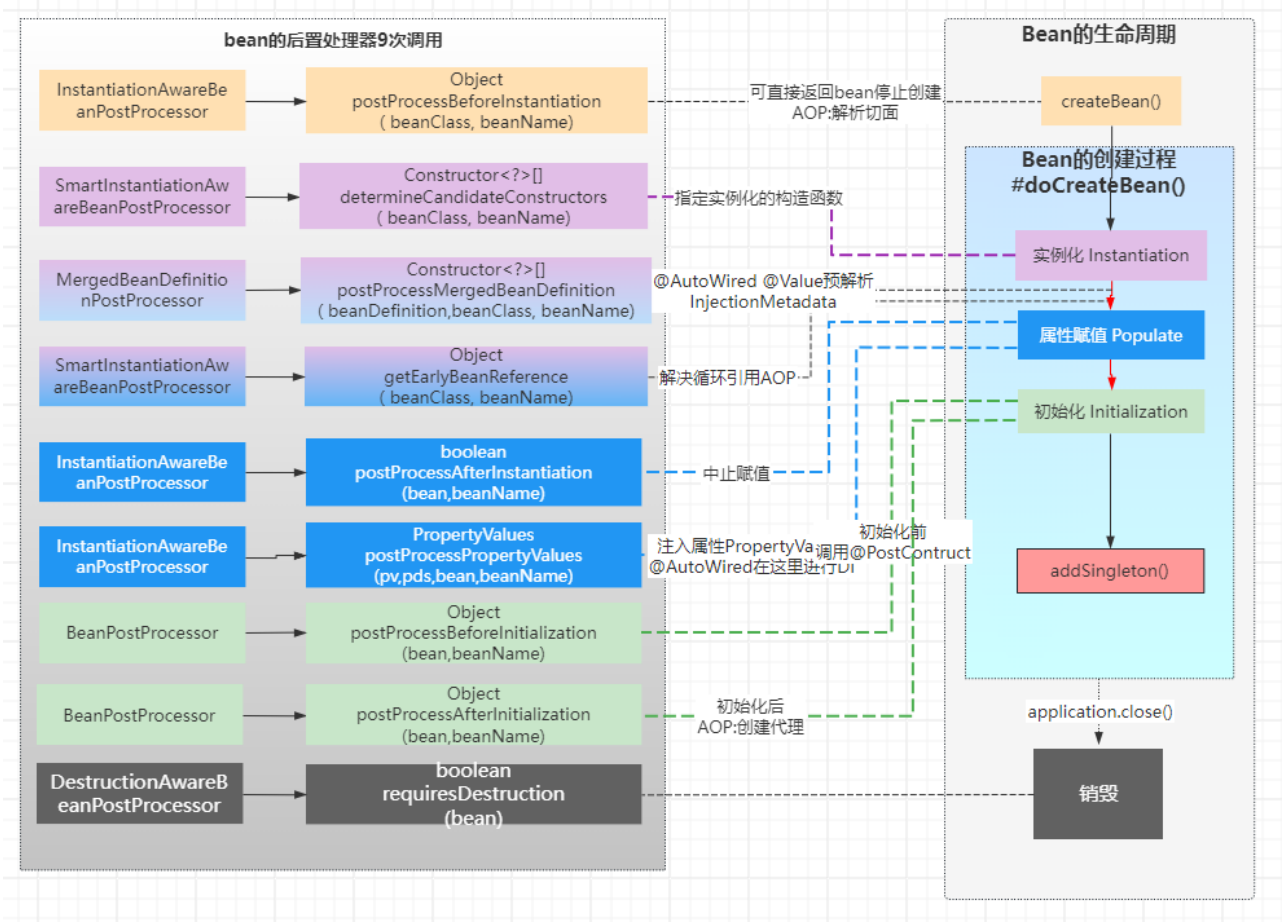
```java
package com.xushu.extensions.beandefinition;

import org.springframework.beans.factory.support.BeanDefinitionRegistry;
import org.springframework.context.annotation.ImportBeanDefinitionRegistrar;
import org.springframework.core.type.AnnotationMetadata;

public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {

    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {


    }
}

```

SpringIoc之Bean创建过程的扩展点详解

1. BeanPostProcessor

更多是为了Spring自己得扩展性， 为以后得版本升级留出更多的扩展余地。

也可以提供给程序员进行扩展，不同阶段的作用不同，实际根据情况进行选择。

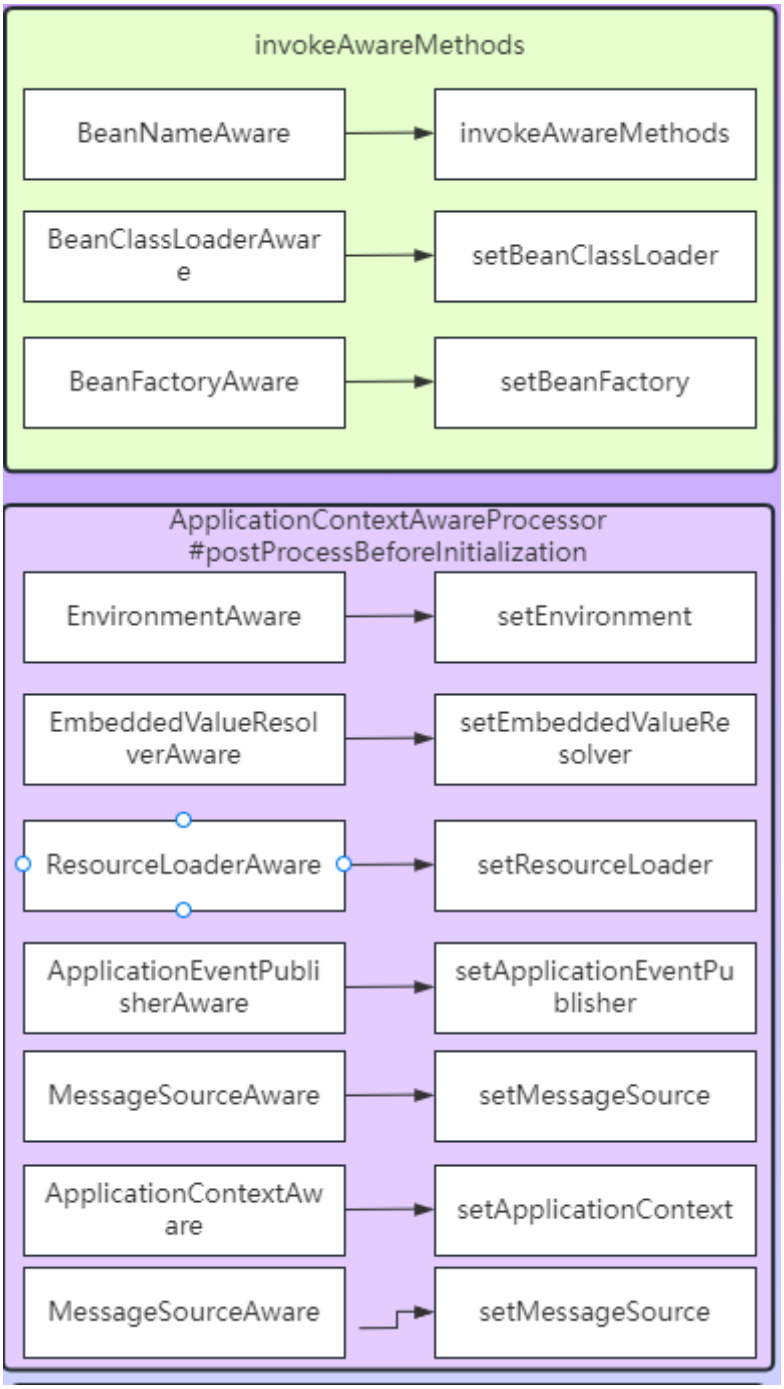


不同阶段可以做不同的事情:

```
1.mainStart实例化前.如果返回了对象会中断bean生命周期
2.mainStart实例化中..可以指定构造函数
3.mainStart实例化后..为属性注入做准备，可以给beanDefinition指定注入的值
5.mainStart属性注入前..返回true中断依赖注入
6.mainStart属性注入中..@Autowired就是通过此bpp进行自动装配的
7.mainStart初始化前
8.mainStart初始化后：bean已经完整可以单独管理
```

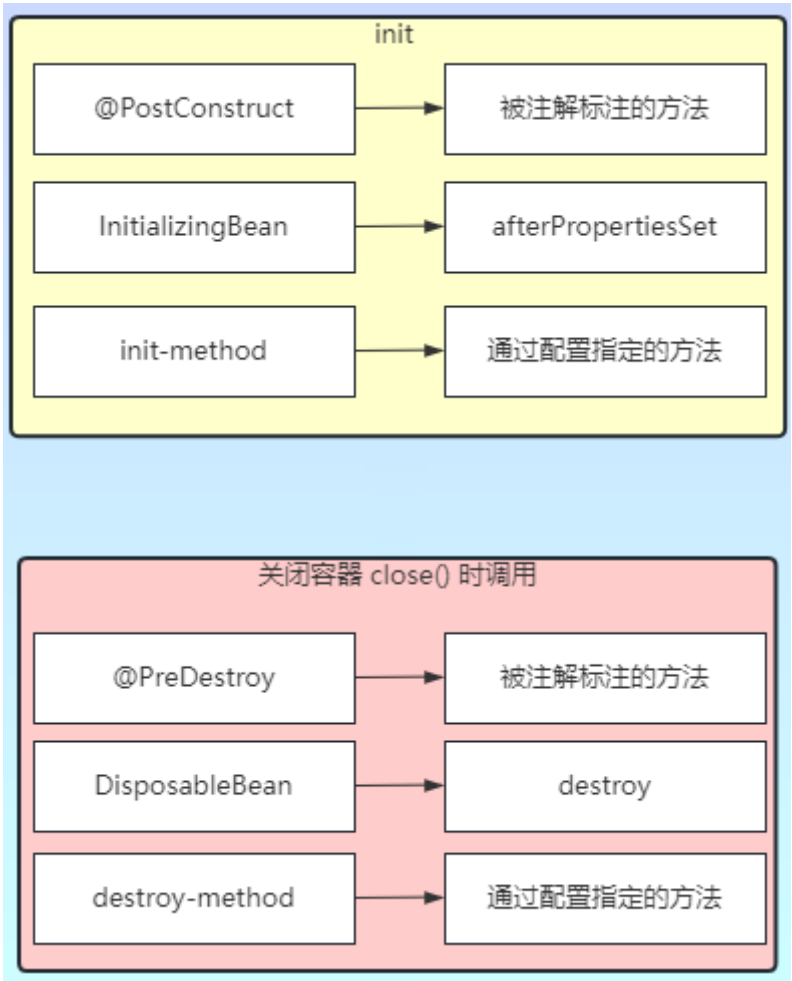
2. Aware

基于底层扩展很少会用@Autowired来注入Spring组件， 因为顺序问题， 基本都会通过Aware获取组件



3. 生命周期回调

1. 如果通过aware获取组件， 那么肯定也会用初始化的回调方式进行初始化， 而不是用构造函数
2. 用构造函数初始化， 由于构造函数在实例化这步， 获取不到像aware这些组件。
3. 构造函数不确定性， 所以作为初始化不合适



4. Springloc之容器加载完毕的扩展点详解

SmartInitializingSingleton.

它其实是初始化回调的一个补充，可以再所有Bean创建完后初始化.比如想对一批bean一起同时做一些动作。

```java
/**
 * 在所有单例bean创建完后调用，做初始化工作
 * 比如需要依赖创建完后的bean 进行一些初始化工作
 *
 */


// 1.有别于初始化回调， 他会在所有单例bean创建完后调用
// 2.有别与refreshedEvent事件监听，依赖小
// 3.仅仅BeanFactory就有可以完成调用的扩展点
@Component
public class MySmartInitializingSingleton implements SmartInitializingSingleton {
    @Override
    public void afterSingletonsInstantiated() {
       System.out.println("所有bean创建完后调用..");
    }
}

```

SmartLifecycle

控制一个组件的生命周期，比如定时器组件\资源预热\缓存预热

* 容器启动完： 定时任务启动 / 缓存预热

* 容器关闭： 定时任务停止 / 缓存清空

所有同Spring容器同开启/关闭 的服务可以基于SmartLifecycle完成， 就不需要自己单独管理开启关闭了

```java
package com.xushu.extensions.created;

import org.springframework.context.SmartLifecycle;
import org.springframework.stereotype.Component;

/**
 *  控制一个组件的生命周期 ，比如定时器组件\资源预热\缓存预热
 *  容器启动完:     定时任务启动
 *  容器关闭:       定时任务停止
 */
@Component
public class MyLifecycle implements SmartLifecycle {
    boolean isRunning;
    @Override
    public void start() {
        isRunning=true;
        System.out.println("容器加载完毕，组件启动！");
    }

    @Override
    public void stop() {
        isRunning=false;
        System.out.println("容器关闭，组件停止！");
    }

    // isRunning=false  调用 start      isRunning=true    调用stop
    @Override
    public boolean isRunning() {
        System.out.println("组件是否运行判断");
        return isRunning;
    }

    @Override
    public boolean isAutoStartup() {
        return SmartLifecycle.super.isAutoStartup();
    }
}
```

```
38
```

ContextRefreshedEvent

基于事件

```
@Component
public class ContextRefreshedEventListener{ //implements
ApplicationListener<ContextRefreshedEvent> {

    //@Async
    @EventListener(ContextRefreshedEvent.class)
    public void onApplicationEvent(ContextRefreshedEvent event)  {
            System.out.println("_____________\n容器加载完毕\n————————");

    }

}

```

利用扩展点实现动态加载线程池小案例演练

此案例根据美团动态线程池dynamic-tp开源项目提取关键扩展点讲解 gitee地址：yanhom/dynamic-tp

在开发中，关于线程池会遇到：

- 由于不同服务器的资源、不同时刻的请求量不一样， 代码中创建了一个 ThreadPoolExecutor，但是不知道那几个核心参数设置多少比较合适
- 参数设置好后，上线发现需要调整，改代码重启服务非常麻烦。
- 线程池相对于开发人员来说是个黑箱，运行情况在出现问题 前很难被感知。

```
public static void main(String[] args) {
        // 创建ThreadPoolExecutor对象
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                corePoolSize,
                maximumPoolSize,
                keepAliveTime,
                TimeUnit.SECONDS,
                workQueue,
                handler
        );
    }
/*
corePoolSize：核心线程数，表示线程池中始终保持活动状态的线程数。
maximumPoolSize：最大线程数，表示线程池中可以同时执行的最大线程数。
keepAliveTime：空闲线程销毁的时间，表示当线程池中的线程数超过核心线程数时，多余的空闲线程在被销毁之前等待的最长时间。
TimeUnit：时间单位，用于指定keepAliveTime的单位。
workQueue：任务队列，用于存储待执行的任务。
handler：拒绝策略，用于处理无法执行的任务。
*/
```

****实现思路**：**

利用SpringBoot的配置文件（后续还可以利用配置中心）

1. 根据配置的参数， 动态创建线程池
2. 将动态线程池bean交给Spring容器管理
3. 后续使用线程池可以从Spring容器中获取动态线程池bean使用
4. 后续修改可以直接对动态线程池bean进行修改
5. 最好还能监控如果达到阈值进行（发邮件）警告。

需求1： 根据配置动态加载信息并且创建

```yaml
spring:
  dtp:
    executors:
      # 线程池1
      - poolName: dtpExecutor1
        corePoolSize: 5
        maximumPoolSize: 10
      #...其他参数

      # 线程池2
      - poolName: dtpExecutor2
        corePoolSize: 2
        maximumPoolSize: 15
      #...其他参数

      #线程池3\4\5
```

毫无疑问要一个Pojo类接收这些配置

```

@Data
public class DtpProperties {

    private List<ThreadPoolProperties> executors;
}

```

```java

@Data
public class ThreadPoolProperties {
    /**
     * 标识每个线程池的唯一名字
     */
    private String poolName;
    private String poolType = "common";

    /**
     * 是否为守护线程
     */
    private boolean isDaemon = false;

    /**
     * 以下都是核心参数
     */
    private int corePoolSize = 1;
    private int maximumPoolSize = 1;
    private long keepAliveTime;
    private TimeUnit timeUnit = TimeUnit.SECONDS;
    private String queueType = "arrayBlockingQueue";
    private int queueSize = 5;
    private String threadFactoryPrefix = "-td-";
    private String RejectedExecutionHandler;
}

```

1.如何获取配置?

1、`@Value`

通过`@Value`单个获取;一个个设置，太麻烦

```
@Value("${com.tuling.bean.bean-class}")
private Class<?> beanClass;

// Todo... 一个个获取
```

2、`@ConfigurationProperties`

通过`@ConfigurationProperties(prefix = "com.tuling")`可以批量获取，比较方便

3、`EnvironmentAware`——选它！

Spring提供很多XXXAware接口、其中EnvironmentAware接口就可以通过其提供的Environment动态获取。

- 第一步：实现`EnvironmentAware`接口

```
@Component
public class TestEnvironmentAware implements EnvironmentAware {
    @Override
    public void setEnvironment(Environment environment) {
        // Todo 绑定配置信息...
    }
}
```

- 第二步：获取/绑定配置，提供两种方式：

  a. 获取方式一：单个获取

```
public void setEnvironment(Environment environment) {
         environment.getProperty("com.tuling.bean.bean-class");
         // ToDo:  一个个获取更多配置信息..
}
```

  a. 获取方式二：通过Binder绑定到properties对象

```java
@Override
public void setEnvironment(Environment environment) {
    BindResult<BeanProperties> bindResult = Binder.get(environment).bind("com.tuling.bean", BeanProperties.class);
    BeanProperties beanProperties= bindResult.get();
}
```

@Value 和@ConfigurationProperties 注解方式获取配置为什么不可以？Why?~

因为顺序原因！这里就要清楚：

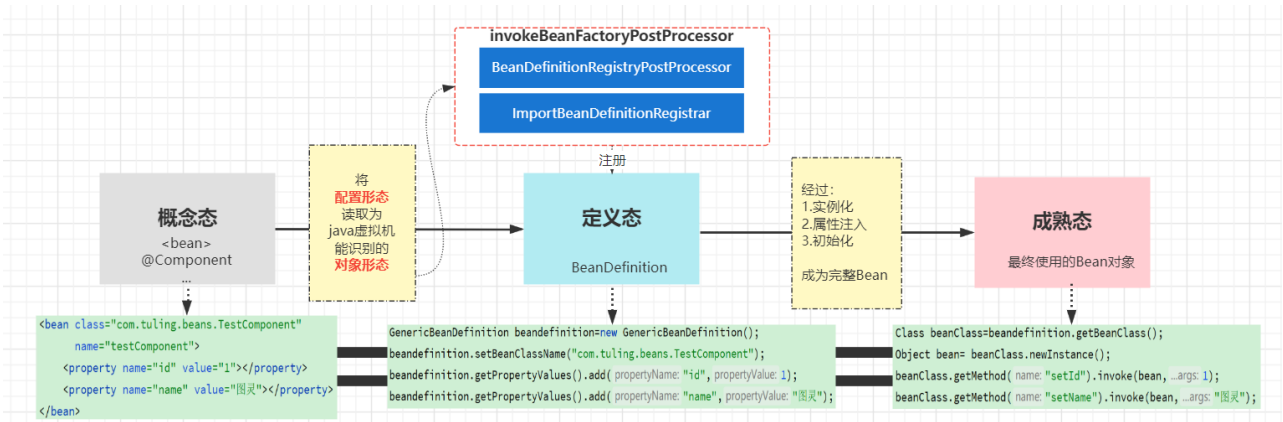
@Value 和@ConfigurationProperties注解依赖**BeanPostProcessor**解析，要调用BeanPostProcessor就**要先注册**，而BeanPostProcessor的注册是在BeanDefinition的注册之后的。

所以在注册BeanDefinition时是获取不到注解绑定的配置信息的：

2. 动态创建Bean的几种方式：

注意！我们需要的是动态！**动态！！**是在运行过程中经过逻辑代码创建Bean, 不是通过配置<bean>、 @Component这种配置方式这种方式不能自由控制业务逻辑。

想要动态创建Bean先了解Bean创建的大概过程：



如果想动态注册Bean,可以通过先动态注册BeanDefintion即可，Spring提供了动态注册BeanDefinition的接口：

1、ImportBeanDefinitionRegistrar

第一步：创建实现`ImportBeanDefinitionRegistrar`接口的类， 演示了一个DeanDefintion的注册

```
public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {
    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry, BeanNameGenerator importBeanNameGenerator) {
        GenericBeanDefinition beandefinition=new GenericBeanDefinition();
        beandefinition.setBeanClassName("com.tuling.beans.TestComponent");
        beandefinition.getPropertyValues().add("id",1);
        beandefinition.getPropertyValues().add("name","图灵");

        registry.registerBeanDefinition("testComponent",beandefinition);
    }
}
```

第二步：结合@Import让它生效

```
@Import(MyImportBeanDefinitionRegistrar.class)
```

2、BeanDefinitionRegistryPostProcessor ——选它！

创建实现`BeanDefinitionRegistryPostProcessor`接口的类， 演示一个DeanDefintion的注册

```
public class MyBeanDefinitionRegistryPostProcessor implements BeanDefinitionRegistryPostProcessor {
    @Override
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException {
        GenericBeanDefinition beandefinition=new GenericBeanDefinition();
        beandefinition.setBeanClassName("com.tuling.beans.TestComponent");
        beandefinition.getPropertyValues().add("id",1);
        beandefinition.getPropertyValues().add("name","图灵");

        registry.registerBeanDefinition("testComponent",beandefinition);

    }
}
```

3、通过BeanFactoryPostProcessor

BeanFactoryPostProcessor也可以，但是没有BeanDefinitionRegistryPostProcessor这么明确的责任是用来注册的，及其他方式就不演示了。

ImportBeanDefinitionRegistrar为什么不行?

ImportBeanDefinitionRegistrar不是一个bean， 没有bean的生命周期， 没有依赖注入功能。

但是！其实ImportBeanDefinitionRegistrar在这个场景也行，啊？？？？？ ， 不是说不会调用`EnvironmentAware`吗
" 是的，这里比较特殊"
在解析@Import的ImportBeanDefinitionRegistrar时候， 会调用BeanClassLoaderAware、BeanFactoryAwar、EnvironmentAware、ResourceLoaderAware
有兴趣可以看源码：
org.springframework.context.annotation.ParserStrategyUtils#invokeAwareMethods

最终实现：

ImportBeanDefinitionRegistrar+`EnvironmentAware`

BeanDefinitionRegistryPostProcessor+`EnvironmentAware`

都行

```java


@Slf4j
public class DtpBeanDefinitionRegistrar  implements ImportBeanDefinitionRegistrar, EnvironmentAware {
    private Environment environment;

    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {

        //绑定资源
        BindResult<DtpProperties> bindResult = Binder.get(environment).bind("spring.dtp", DtpProperties.class);
        DtpProperties dtpProperties = bindResult.get();

        List<ThreadPoolProperties> executors = dtpProperties.getExecutors();
        if (Objects.isNull(executors)) {
            log.info("未检测本地到配置文件线程池");
            return;
        }

        // 把动态线程池对象交给Spring管理
        for (ThreadPoolProperties properties : executors) {
            BeanDefinitionBuilder builder = BeanDefinitionBuilder.genericBeanDefinition(DtpThreadPoolExecutor.class);
            builder.addConstructorArgValue(properties);
            registry.registerBeanDefinition(properties.getPoolName(), builder.getBeanDefinition());
         }

    }


    @Override
    public void setEnvironment(Environment environment) {
        this.environment = environment;
    }
}
```

```

```

```

```

```
// 单独声明动态线程池类
// 把动态线程池和 内置线程池区分开  方便从容器中获取
public class DtpThreadPoolExecutor extends ThreadPoolExecutor{

    public DtpThreadPoolExecutor(ThreadPoolProperties executorProp) {
        super(
                executorProp.getCorePoolSize(),
                executorProp.getMaximumPoolSize(),
                executorProp.getKeepAliveTime(),
                executorProp.getTimeUnit(),
                // 这里的参数我随意写一下， 实际中可以根据配置动态创建
                new ArrayBlockingQueue<>(executorProp.getQueueSize())
        );
    }


    public DtpThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
    }
}

```

需求2：

1. 后续使用线程池可以从Spring容器中获取动态线程池bean使用
2. 后续修改可以直接对动态线程池bean进行修改

3. 想实现一个线程池工具类， 快速管理动态线程池

现在提供一个**线程池工具类**，想把DtpThreadPoolExecutor交给它管理， 在哪个扩展点调用DtpRegistry.registry方法?

```java
public class DtpRegistry {
    /**
     * 储存线程池
     */
    private static final Map<String, ThreadPoolExecutor> EXECUTOR_MAP = new ConcurrentHashMap<>();

    /**
     * 获取线程池
     * @param executorName 线程池名字
     */
    public static ThreadPoolExecutor getExecutor(String executorName) {
        return EXECUTOR_MAP.get(executorName);
    }


    public static Collection<String> getAllExecutorNames(){
        return EXECUTOR_MAP.keySet();
    }


    public static Collection<ThreadPoolExecutor> getAllDtpExecutor(){
        return EXECUTOR_MAP.values();
    }
    /**
     * 线程池注册
     * @param executorName 线程池名字
     */
    public static void registry(String executorName, ThreadPoolExecutor executor) {
        //注册
        EXECUTOR_MAP.put(executorName, executor);
    }


    /**
     * 刷新线程池参数
     * @param executorName 线程池名字
     * @param properties 线程池参数
```

```
     */
    public static void refresh(String executorName, ThreadPoolProperties properties) {
        ThreadPoolExecutor executor =  EXECUTOR_MAP.get(executorName);
        //刷新参数
        //.......

        //executor.setCorePoolSize(properties.xxx);
        //executor.setMaximumPoolSize(properties.xxx);

    }
}

```

实现：

1.**BeanPostProcessor.postProcessAfterInitialization**可以

2.**SmartInitializingSingleton**也OK

```
public class DtpBeanPostProcessor implements BeanPostProcessor {
    private DefaultListableBeanFactory beanFactory;

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        if (bean instanceof DtpThreadPoolExecutor) {
            //直接纳入管理
            DtpRegistry.registry(beanName, (ThreadPoolExecutor) bean);
        }
        return bean;
    }
}
```

动态修改线程池参数

下次变了， 我们调用DtpRegistry.refresh即可。 可以通过前端请求改变，

```
@RestController
@RequestMapping("/dtp")
public class DtpController {

    @PostMapping("/refresh")
    public String refresh(ThreadPoolProperties properties){
        DtpRegistry.refresh(properties.getPoolName(),properties);
        return "success!";
    }

}

```

当然正确的做法应该通过集成配置中心（比如Nacos，修改了配置) 再调用refresh， 这个我们在这里不详讲，后续学了微服务源码自然就知道可以再哪里调用。

所以，以后可能有多处地方调用， 我们可以把它封装成一个事件

4.通过事件通知刷新

```java
/***
 * 事件
 */
public class DtpEvent  extends ApplicationEvent {

    private ThreadPoolProperties properties;

    public DtpEvent(ThreadPoolProperties properties) {
        super(properties);
        this.properties = properties;
    }

    public ThreadPoolProperties getProperties() {
        return properties;
    }
}
```

```java
@Component
public class DtpEventListener {//}  implements ApplicationListener<OrderEvent> {

    // 基于注解的
    @EventListener(DtpEvent.class)
    public void onApplicationEvent(DtpEvent event) {
        ThreadPoolProperties properties = event.getProperties();
        DtpRegistry.refresh(properties.getPoolName(),properties);
    }

}

```

```java
@RestController
@RequestMapping("/dtp")
public class DtpController implements ApplicationEventPublisherAware {

    ApplicationEventPublisher applicationEventPublisher;

    @PostMapping("/refresh")
    public String refresh(ThreadPoolProperties properties){
        applicationEventPublisher.publishEvent(new DtpEvent(properties));
        return "success!";
    }


    @Override
    public void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher) {
        this.applicationEventPublisher=applicationEventPublisher;
    }
}
```

并且还可以把时间设置为异步

```java
/*往SimpleApplicationEventMulticaster设置taskExecutor则为异步事件
  或者使用@Async*/
@Bean(name = "applicationEventMulticaster")
public ApplicationEventMulticaster simpleApplicationEventMulticaster() {
    SimpleApplicationEventMulticaster eventMulticaster
            = new SimpleApplicationEventMulticaster();

    //ThreadPoolTaskExecutor
    eventMulticaster.setTaskExecutor(new SimpleAsyncTaskExecutor());
    return eventMulticaster;
}
```

测试

```
@Autowired
private DtpThreadPoolExecutor dtpExecutor1;

@GetMapping("/add2")
public String addOrder2(){

    dtpExecutor1.execute(() -> {
        System.out.println("下单...");
    });
    return "success!";
}
```

其实到这，我们的功能基本完成， 懂了撒花✿✿ヽ(°▽°)ノ✿

线程池相对于开发人员来说是个黑箱，运行情况在出现问题 前很难被感知。
我还想改造， 我想监听线程池， 如果达到了阈值并且告警。

5.监听线程池

思路很简单， 我就实现一些关键代码

```java

/**
 * auther:  xushu
 */
public class DtpMonitor {

    private ScheduledFuture<?> scheduledFuture;


    private void monitor() {
        for (String name :  DtpRegistry.getAllExecutorNames()) {
            ThreadPoolExecutor dtpExecutor =(ThreadPoolExecutor) DtpRegistry.getExecutor(name);
            System.out.println(String.format("线程池名字: %s", name));
            System.out.println(String.format("线程池核心线程数: %s", dtpExecutor.getCorePoolSize()));
            System.out.println(String.format("线程池最大线程数: %s", dtpExecutor.getMaximumPoolSize()));
            System.out.println(String.format("线程池当前线程数: %s", dtpExecutor.getActiveCount()));
        }
    }

    private void alarm() {
        // 读取配置
        int max = 10;

        for (Executor executor : DtpRegistry.getAllDtpExecutor()) {
            ThreadPoolExecutor threadPoolExecutor=(ThreadPoolExecutor)executor;
            int activeCount = threadPoolExecutor.getActiveCount();
            if (activeCount >= max) {
                System.out.println(String.format("告警，当前线程池的线程个数为%s, 告警阈值为%s", activeCount, max));
            }
        }
    }

}
```

```
34
```

1. 创建一个定时线程
2. monitor定时记录线程池参数， 后续可以用Grafana收集日志
3. alarm 当线程数量达到了阈值 告警， 告警可以自己自由实现比如发邮件，发短线， 我就不完成了

最后， 这个定时器在哪启动呢？？？

6.通过SmartLifecycle改造

让它随容器启动一起启动， 随容器销毁一起销毁

```java

/**
 * auther:  xushu
 */
public class DtpMonitor implements SmartLifecycle {

    private ScheduledFuture<?> scheduledFuture;

    private boolean isRunning=false;

    private void monitor() {
        for (String name :  DtpRegistry.getAllExecutorNames()) {
            ThreadPoolExecutor dtpExecutor =(ThreadPoolExecutor) DtpRegistry.getExecutor(name);
            System.out.println(String.format("线程池名字：%s", name));
            System.out.println(String.format("线程池核心线程数：%s", dtpExecutor.getCorePoolSize()));
            System.out.println(String.format("线程池最大线程数：%s", dtpExecutor.getMaximumPoolSize()));
            System.out.println(String.format("线程池当前线程数：%s", dtpExecutor.getActiveCount()));
        }
    }

    private void alarm() {
        // 读取配置
        int max = 10;

        for (ThreadPoolExecutor threadPoolExecutor : DtpRegistry.getAllDtpExecutor()) {

            int activeCount = threadPoolExecutor.getActiveCount();
            if (activeCount >= max) {
                System.out.println(String.format("告警，当前线程池的线程个数为%s，告警阈值为%s", activeCount, max));
            }
        }
    }

    @Override
    public void start() {
```

```
            scheduledFuture = Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(() -> {
                monitor();
                alarm();
            }, 5, 5, TimeUnit.SECONDS);
            isRunning=true;
        }

        @Override
        public void stop() {
            scheduledFuture.cancel(false);
            isRunning=false;
        }

        @Override
        public boolean isRunning() {
            return isRunning;
        }
    }
```

7. 将动态线程池封装成插件

别的项目如果要用一个@EnableDynamicThreadPool 就行

```

@SpringBootApplication
@EnableDynamicThreadPool
public class DynamicThreadpoolApplication {

    public static void main(String[] args) {
       SpringApplication.run(DynamicThreadpoolApplication.class, args);
    }


}

```

很简单， 你会发现很多@EnableXXX 里面都有一个@Import， 把我们刚刚写的那堆组件注册进去就行

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Import(DtpImportSelector.class)
public @interface EnableDynamicThreadPool {
}
```

```java
public class DtpImportSelector implements DeferredImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        return new String[]{
                DtpImportBeanDefinitionRegistrar.class.getName(),
                DtpBeanPostProcessor.class.getName(),
                DtpMonitor.class.getName()
        };
    }
}
```

好， 希望大家通过这个案例， 可以对Spring的扩展点有一个新的认识！并且以后可以灵活运用在工作中。

有道云链接：https://note.youdao.com/s/UbEHulQo