

# 简单版动态线程池实现原理分析

背景

前置条件

动手实现

定义DtpExecutor

创建DtpExecutor

第一次测试

NacosRefresher

更新DtpExecutor

总结

**作者：图灵课堂-大都督周瑜**

最近看到美团技术团队的动态线程池分析文章：<https://tech.meituan.com/2020/04/02/java-pooling-practice-in-meituan.html>

以及一个对应的开源项目：<https://github.com/dromara/dynamic-tp>

有点意思，同时也觉得动态线程池在工作的实用性，便通过此文来分析一下动态线程池的核心实现原理，本文参考了美团的文章和开源项目的实现思路，特此感谢。

## 背景

动态线程池，指的是线程池中的参数可以动态修改并生效，比如corePoolSize、maximumPoolSize等。

在工作中，线程池的核心线程数和最大线程数等参数是很难估计和固定的，如果能在应用运行过程中动态进行调整，也就很有必要了。

## 前置条件

1. 直接基于SpringBoot
2. 支持Nacos配置中心配置

### 核心配置项

```
1 dtp:
2   enable: true
3   core-pool-size: 10
4   maximum-pool-size: 50
```

YAML

复制代码

我希望，能通过以上配置就能配置出一个动态线程池：

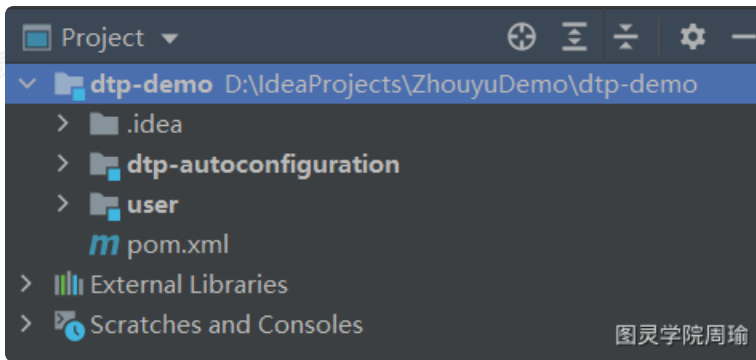
1. dtp：表示dynamic thread pool，动态线程池的缩写
2. enable：表示是否使用动态线程池，默认为true
3. core-pool-size：表示dtp的核心线程数
4. maximum-pool-size：表示dtp的最大线程数
5. 对于线程池的其他参数，可以后续再扩展

另外，我希望能做到，只有项目的配置中存在dtp配置，并且enable不等于false，那就表示开启动态线程池，就需要向Spring容器中添加一个线程池对象作为一个Bean对象，这样其他Bean就能通过依赖注入来使用动态线程池了。

另外，对于上面的配置，我们最好是配置在nacos中，这样才能动态修改。

## 动手实现

首先创建两个项目：



1. dtp-autoconfiguration: 表示动态线程池的自动配置模块, 会存放一些相关的自动配置类
2. user: 表示一个业务应用, 会使用动态线程池

然后把user改写为一个SpringBoot应用:

引入依赖:

```
XML | 复制代码
1 <dependencyManagement>
2   <dependencies>
3     <dependency>
4       <groupId>org.springframework.boot</groupId>
5       <artifactId>spring-boot-dependencies</artifactId>
6       <version>2.3.12.RELEASE</version>
7       <type>pom</type>
8       <scope>import</scope>
9     </dependency>
10  </dependencies>
11 </dependencyManagement>
12
13 <dependencies>
14   <dependency>
15     <groupId>org.springframework.boot</groupId>
16     <artifactId>spring-boot-starter-web</artifactId>
17   </dependency>
18 </dependencies>
```

新建启动类和Controller:

```
1  @SpringBootApplication
2  public class MyApplication {
3      public static void main(String[] args) {
4          SpringApplication.run(MyApplication.class, args);
5      }
6  }
```

```
1  @RestController
2  public class ZhouyuController {
3
4      @GetMapping("/test")
5      public String test(){
6          return "hello";
7      }
8  }
```

现在，我喜欢能在ZhouyuController中使用动态线程池，就像如下：

```
1  @RestController
2  public class ZhouyuController {
3
4      @Autowired
5      private ThreadPoolExecutor threadPoolExecutor; // 需要一个动态线程池
6
7      @GetMapping("/test")
8      public String test(){
9          threadPoolExecutor.execute(() -> {
10              System.out.println("执行任务");
11          });
12          return "hello";
13      }
14  }
```

这段要能工作，得有几个条件：

1. Spring容器中得有一个ThreadPoolExecutor类型的Bean
2. 并且这个ThreadPoolExecutor对象还得是我们所说的动态线程池对象

## 定义DtpExecutor

这里就引出一个问题，我们到底该如何表示一个动态线程池，动态线程池和普通线程池的区别在于，动态线程池能支持通过nacos来修改其参数。

那我们是不是需要新定义一个类来表示动态线程池呢？我给的答案是需要，因为如果不新定义一个，那么对于上述代码，如果我Spring容器中存在多个ThreadPoolExecutor类型的Bean对象，那么该如何找到动态线程池呢？只能通过属性名字了，比如属性名字为dynamicThreadPoolExecutor，这样也就需要我们在往Spring容器注册动态线程池对象时，对于的beanName一定得是dynamicThreadPoolExecutor。

而如果我们新定义一个类（dtp-aucotoncifuration项目中）：

```
Java | 复制代码
1 public class DtpExecutor extends ThreadPoolExecutor {
2
3     public DtpExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {
4         super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
5     }
6
7 }
```

那么如果我们想要用动态线程池就方便了：

```
1  @RestController
2  public class ZhouyuController {
3
4      @Autowired
5      private DtpExecutor dtpExecutor; // 需要一个动态线程池
6
7      @GetMapping("/test")
8      public String test(){
9          dtpExecutor.execute(() -> {
10              System.out.println("执行任务");
11          });
12          return "hello";
13      }
14  }
```

这样，代码看起来就更加明确了。

注意，user中添加依赖：

```
1  <dependency>
2      <groupId>org.example</groupId>
3      <artifactId>ctp-autoconfiguration</artifactId>
4      <version>1.0-SNAPSHOT</version>
5  </dependency>
```

## 创建DtpExecutor

接下来，我们再来创建DtpExecutor对象并添加到Spring容器中，这一步是非常重要的。

如果应用要开启动态线程池，那么就需要做一步，否则就不需要做这一步，并且在创建DtpExecutor对象时，得用配置的参数，并且得支持Nacos，并且还得放到Spring容器中。

这里就可以用到SpringBoot的自动配置类了。

首先在dtp-autoconfiguration中添加spring-boot的依赖：

```
XML | 复制代码
1 <dependencyManagement>
2   <dependencies>
3     <dependency>
4       <groupId>org.springframework.boot</groupId>
5       <artifactId>spring-boot-dependencies</artifactId>
6       <version>2.3.12.RELEASE</version>
7       <type>pom</type>
8       <scope>import</scope>
9     </dependency>
10  </dependencies>
11 </dependencyManagement>
12
13 <dependencies>
14   <dependency>
15     <groupId>org.springframework.boot</groupId>
16     <artifactId>spring-boot-starter</artifactId>
17   </dependency>
18 </dependencies>
```

并且新建一个自动配置类：

```
Java | 复制代码
1 @Configuration
2 @ConditionalOnProperty(prefix = "dtp", value = "enable", havingValue = "true")
3 public class DtpAutoConfiguration {
4 }
```

表示这个配置类，只有在dtp.enable=true的时候才会生效，没有这个配置项或为false则不会生效。

然后我们就可以可以在DtpAutoConfiguration中来定义DtpExecutor的Bean了，我们首先想到的就是利用@Bean，比如：

```
1 @Bean
2 public DtpExecutor dtpExecutor(){
3     DtpExecutor dtpExecutor = new DtpExecutor();
4     return dtpExecutor;
5 }
```

但是，DtpExecutor中是没有无参构造方法的，也就是在构造DtpExecutor对象时，我们需要能够拿到配置项参数，那怎么拿呢？

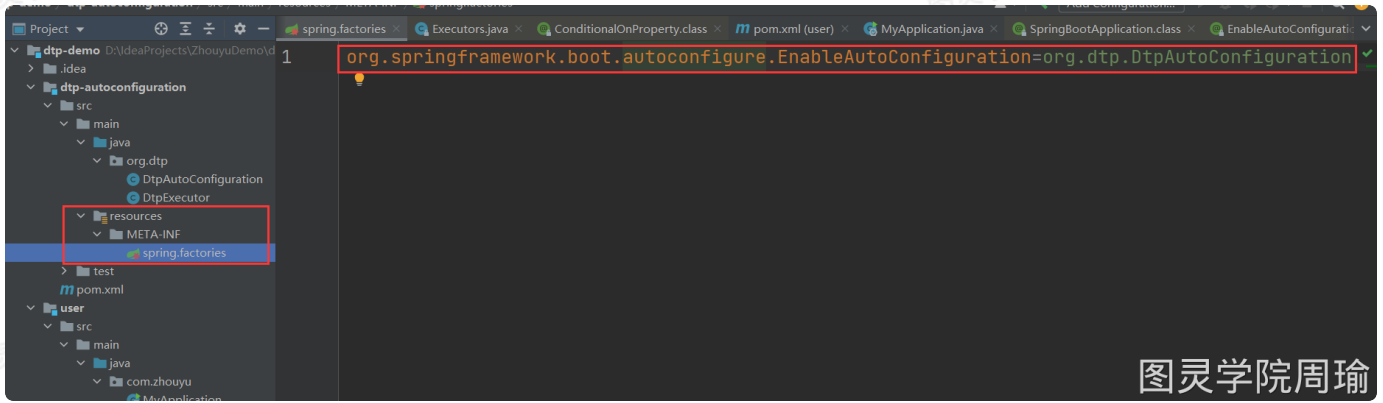
熟悉Spring的同学，可能会想到利用Environment对象，因为不管我们在应用程序本地的application.yaml中的配置项，还是在nacos中的配置项，最终都是放在了Environment对象中，比如如下代码：

```
1 @Configuration
2 @ConditionalOnProperty(prefix = "dtp", value = "enable", havingValue = "true")
3 public class DtpAutoConfiguration {
4
5     @Autowired
6     private Environment environment;
7
8     @Bean
9     public DtpExecutor dtpExecutor(){
10         Integer corePoolSize = Integer.valueOf(environment.getProperty("dtp.core-pool-size"));
11         Integer maximumPoolSize = Integer.valueOf(environment.getProperty("dtp.maximum-pool-size"));
12
13         DtpExecutor dtpExecutor = new DtpExecutor(corePoolSize, maximumPoolSize);
14         return dtpExecutor;
15     }
16 }
```

## 第一次测试



我们先来测试一下，注意DtpAutoConfiguration自动配置类要能够生效，还需要利用spring.factories：



因为我们有限制，所以我们还得在user中配置dtp：

```
YAML | 复制代码
1 dtp:
2   enable: true
3   core-pool-size: 10
4   maximum-pool-size: 50
```

并且把ZhouyuController的代码也大概改一下：

```
Java | 复制代码
1 @RestController
2 public class ZhouyuController {
3
4     @Autowired
5     private DtpExecutor dtpExecutor; // 需要一个动态线程池
6
7     @GetMapping("/test")
8     public Integer test(){
9         return dtpExecutor.getCorePoolSize();
10    }
11 }
```

这样就能测出来，能不能使用动态线程池，并且是否是我们所配置的参数。

启动User应用，然后访问localhost:8080/test，结果为：

← → ↺ ⌂ localhost:8080/test

10

图灵学院周瑜

发现，结果正常。

并且，我们可以试试在nacos中进行配置，那我们需要在user中引入nacos-client：

XML | 复制代码

```
1 <dependency>
2   <groupId>com.alibaba.boot</groupId>
3   <artifactId>nacos-config-spring-boot-starter</artifactId>
4   <version>0.2.7</version>
5 </dependency>
```

然后配置nacos:

YAML | 复制代码

```
1 nacos:
2   config:
3     server-addr: 127.0.0.1:8848
4     data-id: dtp.yaml
5     type: yaml
6     auto-refresh: true
7     bootstrap:
8       enable: true
```

然后在nacos中配置dtp.yaml:

\* Data ID: dtp.yaml

\* Group: DEFAULT\_GROUP

[更多高级选项](#)

描述:

Beta发布: ☐ 默认不要勾选。

配置格式: ☐ TEXT ☐ JSON ☐ XML ☒ YAML ☐ HTML ☐ Properties

配置内容①:

```
1 dtp:
2   enable: true
3   core-pool-size: 15
4   maximum-pool-size: 50
```

图灵学院周瑜

启动user应用，访问localhost:8080/test:

← → ↻ 🏠 localhost:8080/test

15

结果也是正常的。

也就是说，代码写到这，我们完成了：

1. 能读取nacos中的配置
2. 并创建DtpExecutor对象
3. 并放入Spring容器

## NacosRefresher

那么最核心的问题还没有解决：应用在运行过程中，如果在nacos中修改了配置，如何生效？

这就需要在user应用中能够发现nacos中配置内容是否修改了，这就需要利用到nacos的监听器机制，我们在auto-configuration模块来定义一个nacos的监听器，这就需要auto-configuration也依赖nacos-client，我们直接nacos-client的依赖从user模块转移到auto-configuration模块中去，这样对于user是没有影响的，因为user依赖了auto-configuration模块，从而间接的依赖了nacos-client。

我们新建一个Nacos监听器NacosRefresher:

```
1 public class NacosRefresher implements Listener, InitializingBean {
2
3     @NacosInjected
4     private ConfigService configService;
5
6     // 利用Spring的Bean初始化机制，来设置要监听的nacos的dataId
7     // 暂时写死，最好是拿到程序员所配置的dataId和group
8     @Override
9     public void afterPropertiesSet() throws NacosException {
10         configService.addListener("dtp.yaml", "DEFAULT_GROUP", this);
11     }
12
13     // 这个是Nacos收到变更事件异步执行逻辑要用到的线程池，跟动态线程池没关系
14     @Override
15     public Executor getExecutor() {
16         return Executors.newFixedThreadPool(1);
17     }
18
19     // 这是用来接收数据变更的，content就是变更后的内容
20     @Override
21     public void receiveConfigInfo(String content) {
22         System.out.println(content);
23     }
24 }
```

另外在DtpAutoConfiguration中定义NacosRefresher为一个Bean:

```
1 @Bean
2 public NacosRefresher nacosRefresher(){
3     return new NacosRefresher();
4 }
```

也可以利用@Import来导入NacosRefresher:

```
1 @Configuration
2 @ConditionalOnProperty(prefix = "dtp", value = "enable", havingValue = "true")
3 @Import(NacosRefresher.class)
4 public class DtpAutoConfiguration {
5
6     @Autowired
7     private Environment environment;
8
9     @Bean
10    public DtpExecutor dtpExecutor(){
11        Integer corePoolSize = Integer.valueOf(environment.getProperty("dtp.core-pool-size"));
12        Integer maximumPoolSize = Integer.valueOf(environment.getProperty("dtp.maximum-pool-size"));
13
14        DtpExecutor dtpExecutor = new DtpExecutor(corePoolSize, maximumPoolSize);
15        return dtpExecutor;
16    }
17
18
19 }
```

## 更新DtpExecutor

NacosRefresher一旦就到了数据变更事件，那么就可以更新DtpExecutor了，那么这里我们要解决两个问题：

1. 解析content，因为content是String，我们需要进行解析并得到配置项内容
2. 更新Spring容器中的DtpExecutor对象，那就需要能够拿到DtpExecutor对象

我们先启动应用，修改一下nacos的配置，看看context长什么样：



我只是修改了core-pool-size，但是content是怎么ntp.yaml的内容，那么我们就来进行解析：



结果为：



```
34
35 @Override
36 public void receiveConfigInfo(String content) {
37     YamlPropertiesFactoryBean bean = new YamlPropertiesFactoryBean();
38     bean.setResources(new ByteArrayResource(content.getBytes()));
39     Properties properties = bean.getObject();
40     System.out.println(properties);
41 }
42
43
```

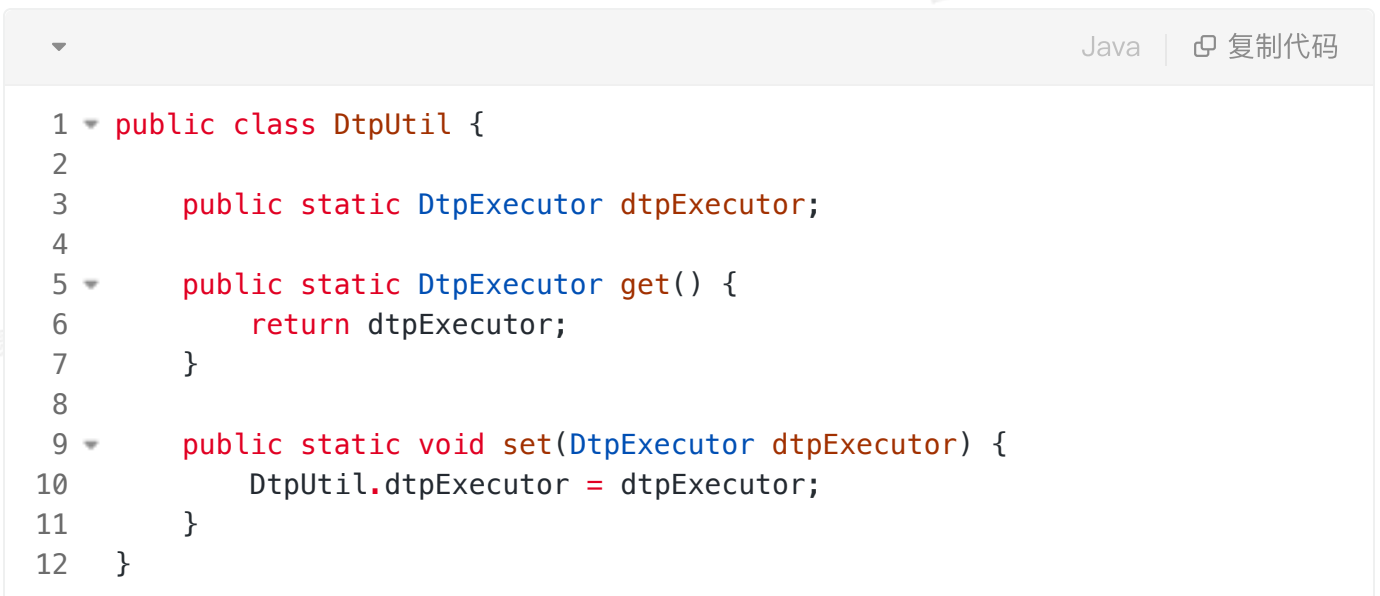
```
728 INFO 13096 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring em
728 INFO 13096 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationCor
921 INFO 13096 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing Executors
094 INFO 13096 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port
128 INFO 13096 --- [main] com.zhouyu.MyApplication : Started MyApplication
{dtp.core-pool-size=18, dtp.enable=true, dtp.maximum-pool-size=50}
```

图灵学院周瑜

这样，我们就将content解析为了Properties的格式，这样就能更加方便获取配置项了。

接下来，我们只要能拿到Spring容器中的DtpExecutor对象，那么该如何拿到呢，这里参考开源项目 [dynamic-tp](#) 的做法，利用BeanPostProcessor来存入到一个static的map中。

首先新建一个DtpUtil：



```
1 public class DtpUtil {
2
3     public static DtpExecutor dtpExecutor;
4
5     public static DtpExecutor get() {
6         return dtpExecutor;
7     }
8
9     public static void set(DtpExecutor dtpExecutor) {
10         DtpUtil.dtpExecutor = dtpExecutor;
11     }
12 }
```

Java | 复制代码

然后新建一个BeanPostProcessor，会把DtpExecutor对象存入DtpUtil：

```
1 public class DtpBeanPostProcessor implements BeanPostProcessor {
2
3     @Override
4     public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
5         if (bean instanceof DtpExecutor) {
6             DtpUtil.set((DtpExecutor) bean);
7         }
8
9         return bean;
10    }
11
12 }
```

另外在DtpAutoConfiguration导入DtpBeanPostProcessor。

然后会到NacosRefresher中，我们就可以利用DtpUtil获取到DtpExecutor对象了，并且可以修改对应的参数：

```
1 @Override
2 public void receiveConfigInfo(String content) {
3     YamlPropertiesFactoryBean bean = new YamlPropertiesFactoryBean();
4     bean.setResources(new ByteArrayResource(content.getBytes()));
5     Properties properties = bean.getObject();
6
7     DtpExecutor dtpExecutor = DtpUtil.get();
8     dtpExecutor.setCorePoolSize(Integer.parseInt(properties.getProperty("dtp.core-pool-size")));
9     dtpExecutor.setMaximumPoolSize(Integer.parseInt(properties.getProperty("dtp.maximum-pool-size")));
10 }
```

这样就完成了DtpExecutor的参数修改，到此，一个简单的动态线程池就完成了，大家可以自行测试一下，修改Nacos配置，看controller那边能不能实时拿到最新的corePoolSize，我测是没问题的。



# 总结

对于实现一个动态线程池，核心要点为：

1. 动态线程池为一个Bean对象
2. 能实时发现配置的变更
3. 不过最为核心的是线程池本身的源码设计

比如ThreadPoolExecutor的setCorePoolSize：

```
1  public void setCorePoolSize(int corePoolSize) {
2      if (corePoolSize < 0)
3          throw new IllegalArgumentException();
4      int delta = corePoolSize - this.corePoolSize;
5      this.corePoolSize = corePoolSize;
6      if (workerCountOf(ctl.get()) > corePoolSize)
7          interruptIdleWorkers();
8      else if (delta > 0) {
9          // We don't really know how many new threads are "needed".
10         // As a heuristic, prestart enough new workers (up to new
11         // core size) to handle the current number of tasks in
12         // queue, but stop if queue becomes empty while doing so.
13         int k = Math.min(delta, workQueue.size());
14         while (k-- > 0 && addWorker(null, true)) {
15             if (workQueue.isEmpty())
16                 break;
17         }
18     }
19 }
```

它会判断：

1. 如果当前工作线程大于最新corePoolSize，那么则会中断空闲线程，最终只会保护corePoolSize个空闲线程
2. 如果核心线程数增加了，那么就会调用addWorker(null, true)方法来新增核心线程

而所谓的动态线程池，其实就是动态的去修改线程池中的线程数量，少了就增加，多了就中断。