98-项目实战三：设计实现一个支持自定义规则的灰度发布组件（实现）

上两节课，我们讲解了灰度组件的需求和设计思路。不管是之前讲过的限流、幂等框架，还是现在正在讲的灰度组件，这些框架、组件、类库的功能性需求都不复杂，相反，非功能性需求是开发的重点、难点。

今天，我们按照上节课给出的灰度组件的设计思路，讲解如何进行编码实现。不过今天对实现的讲解，跟前面两个实战项目有所不同。在前面两个项目中，我都是手把手地从最基础的MVP代码讲起，然后讲解如何review代码发现问题、重构代码解决问题，最终得到一份还算高质量的代码。考虑到已经有前面两个项目的学习和锻炼了，你应该对开发套路、思考路径很熟悉了，所以，今天我们换个讲法，就不从最基础的讲起了，而是重点讲解实现思路。

话不多说，让我们正式开始今天的学习吧！

灰度组件功能需求整理

针对上两节课给出的开发需求和设计思路，我们还是按照老套路，从中剥离出V1版本要实现的内容。为了方便我讲解和你查看，我把灰度组件的开发需求和设计思路，重新整理罗列了一下，放到了这里。

1.灰度规则的格式和存储方式

我们希望支持不同格式（JSON、YAML、XML等）、不同存储方式（本地配置文件、Redis、Zookeeper、或者自研配置中心等）的灰度规则配置方式。实际上，这一点跟之前的限流框架中限流规则的格式和存储方式完全一致，代码实现也是相同的，所以在接下来的讲解中，就不重复啰嗦了，你可以回过头去看下第92讲。

2.灰度规则的语法格式

我们支持三种灰度规则语法格式：具体值（比如893）、区间值（比如1020-1120）、比例值（比如%30）。除此之外，对于更加复杂的灰度规则，比如只对30天内购买过某某商品并且退货次数少于10次的用户进行灰度，我们通过编程的方式来实现。

3.灰度规则的内存组织方式

类似于限流框架中的限流规则，我们需要把灰度规则组织成支持快速查找的数据结构，能够快速判定某个灰度对象（darkTarget，比如用户ID），是否落在灰度规则设定的范围内。

4.灰度规则热更新

修改了灰度规则之后，我们希望不重新部署和重启系统，新的灰度规则就能生效，所以，我们需要支持灰度规则热更新。

在V1版本中，对于第一点灰度规则的格式和存储方式，我们只支持YAML格式本地文件的配置存储方式。对于剩下的三点，我们都要进行实现。考虑到V1版本要实现的内容比较多，我们分两步来实现代码，第一步先将大的流程、框架搭建好，第二步再进一步添加、丰富、优化功能。

实现灰度组件基本功能

在第一步中，我们先实现基于YAML格式的本地文件的灰度规则配置方式，以及灰度规则热更新，并且只支持三种基本的灰度规则语法格式。基于编程实现灰度规则的方式，我们留在第二步实现。

我们先把这个基本功能的开发需求，用代码实现出来。它的目录结构及其Demo示例如下所示。代码非常简单，只包含4个类。接下来，我们针对每个类再详细讲解一下。

```
// 代码目录结构
com.xzg.darklaunch
  --DarkLaunch(框架的最顶层入口类)
  --DarkFeature(每个feature的灰度规则)
  --DarkRule(灰度规则)
  --DarkRuleConfig(用来映射配置到内存中)

// Demo示例
public class DarkDemo {
  public static void main(String[] args) {
    DarkLaunch darkLaunch = new DarkLaunch();
    DarkFeature darkFeature = darkLaunch.getDarkFeature("call_newapi_getUserById");
    System.out.println(darkFeature.enabled());
    System.out.println(darkFeature.dark(893));
  }
}

// 灰度规则配置(dark-rule.yaml)放置在classpath路径下
features:
- key: call_newapi_getUserById
  enabled: true
  rule: {893,342,1020-1120,%30}
- key: call_newapi_registerUser
  enabled: true
  rule: {1391198723, %10}
- key: newalgo_loan
  enabled: true
  rule: {0-1000}
```

从Demo代码中，我们可以看出，对于业务系统来说，灰度组件的两个直接使用的类是DarkLaunch类和DarkFeature类。

我们先来看DarkLaunch类。这个类是灰度组件的最顶层入口类。它用来组装其他类对象，串联整个操作流程，提供外部调用的接口。

DarkLaunch类先读取灰度规则配置文件，映射为内存中的Java对象（DarkRuleConfig），然后再将这个中间结构，构建成一个支持快速查询的数据结构（DarkRule）。除此之外，它还负责定期更新灰度规则，也就是前面提到的灰度规则热更新。

为了避免更新规则和查询规则的并发执行冲突，在更新灰度规则的时候，我们并非直接操作老的DarkRule，而是先创建一个新的DarkRule，然后等新的DarkRule都构建好之后，再“瞬间”赋值给老的DarkRule。你可以结合着下面的代码一块看下。

```
public class DarkLaunch {
  private static final Logger log = LoggerFactory.getLogger(DarkLaunch.class);
  private static final int DEFAULT_RULE_UPDATE_TIME_INTERVAL = 60; // in seconds
  private DarkRule rule;
  private ScheduledExecutorService executor;
```

```
  public DarkLaunch(int ruleUpdateTimeInterval) {
    loadRule();
    this.executor = Executors.newSingleThreadScheduledExecutor();
    this.executor.scheduleAtFixedRate(new Runnable() {
      @Override
      public void run() {
        loadRule();
      }
    }, ruleUpdateTimeInterval, ruleUpdateTimeInterval, TimeUnit.SECONDS);
  }

  public DarkLaunch() {
    this(DEFAULT_RULE_UPDATE_TIME_INTERVAL);
  }

  private void loadRule() {
    // 将灰度规则配置文件dark-rule.yaml中的内容读取DarkRuleConfig中
    InputStream in = null;
    DarkRuleConfig ruleConfig = null;
    try {
      in = this.getClass().getResourceAsStream("/dark-rule.yaml");
      if (in != null) {
        Yaml yaml = new Yaml();
        ruleConfig = yaml.loadAs(in, DarkRuleConfig.class);
      }
    } finally {
      if (in != null) {
        try {
          in.close();
        } catch (IOException e) {
          log.error("close file error:{}", e);
        }
      }
    }

    if (ruleConfig == null) {
      throw new RuntimeException("Can not load dark rule.");
    }
    // 更新规则并非直接在this.rule上进行，
    // 而是通过创建一个新的DarkRule，然后赋值给this.rule，
    // 来避免更新规则和规则查询的并发冲突问题
    DarkRule newRule = new DarkRule(ruleConfig);
    this.rule = newRule;
  }

  public DarkFeature getDarkFeature(String featureKey) {
    DarkFeature darkFeature = this.rule.getDarkFeature(featureKey);
    return darkFeature;
  }
}
```

我们再来看下DarkRuleConfig类。这个类功能非常简单，只是用来将灰度规则映射到内存中。具体的代码如下所示：

```
public class DarkRuleConfig {
  private List<DarkFeatureConfig> features;

  public List<DarkFeatureConfig> getFeatures() {
    return this.features;
```

```
  }

  public void setFeatures(List<DarkFeatureConfig> features) {
    this.features = features;
  }

  public static class DarkFeatureConfig {
    private String key;
    private boolean enabled;
    private String rule;
    // 省略getter、setter方法
  }
}
```

从代码中，我们可以看出来，DarkRuleConfig类嵌套了一个内部类DarkFeatureConfig。这两个类跟配置文件的两层嵌套结构完全对应。我把对应关系标注在了下面的示例中，你可以对照着代码看下。

```
<!--对应DarkRuleConfig-->
features:
- key: call_newapi_getUserById  <!--对应DarkFeatureConfig-->
  enabled: true
  rule: {893,342,1020-1120,%30}
- key: call_newapi_registerUser <!--对应DarkFeatureConfig-->
  enabled: true
  rule: {1391198723, %10}
- key: newalgo_loan             <!--对应DarkFeatureConfig-->
  enabled: true
  rule: {0-1000}
```

我们再来看下DarkRule。DarkRule包含所有要灰度的业务功能的灰度规则。它用来支持根据业务功能标识（feature key），快速查询灰度规则（DarkFeature）。代码也比较简单，具体如下所示：

```
public class DarkRule {
  private Map<String, DarkFeature> darkFeatures = new HashMap<>();

  public DarkRule(DarkRuleConfig darkRuleConfig) {
    List<DarkRuleConfig.DarkFeatureConfig> darkFeatureConfigs = darkRuleConfig.getFeatures();
    for (DarkRuleConfig.DarkFeatureConfig darkFeatureConfig : darkFeatureConfigs) {
      darkFeatures.put(darkFeatureConfig.getKey(), new DarkFeature(darkFeatureConfig));
    }
  }

  public DarkFeature getDarkFeature(String featureKey) {
    return darkFeatures.get(featureKey);
  }
}
```

我们最后来看下DarkFeature类。DarkFeature类表示每个要灰度的业务功能的灰度规则。DarkFeature将配置文件中灰度规则，解析成一定的结构（比如RangeSet），方便快速判定某个灰度对象是否落在灰度规则范围内。具体的代码如下所示：

```
public class DarkFeature {
  private String key;
  private boolean enabled;
  private int percentage;
  private RangeSet<Long> rangeSet = TreeRangeSet.create();

  public DarkFeature(DarkRuleConfig.DarkFeatureConfig darkFeatureConfig) {
    this.key = darkFeatureConfig.getKey();
    this.enabled = darkFeatureConfig.getEnabled();
    String darkRule = darkFeatureConfig.getRule().trim();
    parseDarkRule(darkRule);
  }

  @VisibleForTesting
  protected void parseDarkRule(String darkRule) {
    if (!darkRule.startsWith("{") || !darkRule.endsWith("}")) {
      throw new RuntimeException("Failed to parse dark rule: " + darkRule);
    }

    String[] rules = darkRule.substring(1, darkRule.length() - 1).split(",");
    this.rangeSet.clear();
    this.percentage = 0;
    for (String rule : rules) {
      rule = rule.trim();
      if (StringUtils.isEmpty(rule)) {
        continue;
      }

      if (rule.startsWith("%")) {
        int newPercentage = Integer.parseInt(rule.substring(1));
        if (newPercentage > this.percentage) {
          this.percentage = newPercentage;
        }
      } else if (rule.contains("-")) {
        String[] parts = rule.split("-");
        if (parts.length != 2) {
          throw new RuntimeException("Failed to parse dark rule: " + darkRule);
        }
        long start = Long.parseLong(parts[0]);
        long end = Long.parseLong(parts[1]);
        if (start > end) {
          throw new RuntimeException("Failed to parse dark rule: " + darkRule);
        }
        this.rangeSet.add(Range.closed(start, end));
      } else {
        long val = Long.parseLong(rule);
        this.rangeSet.add(Range.closed(val, val));
      }
    }
  }

  public boolean enabled() {
    return this.enabled;
  }

  public boolean dark(long darkTarget) {
    boolean selected = this.rangeSet.contains(darkTarget);
    if (selected) {
      return true;
    }

    long reminder = darkTarget % 100;
    if (reminder >= 0 && reminder < this.percentage) {
```

```
      return true;
    }

    return false;
  }

  public boolean dark(String darkTarget) {
    long target = Long.parseLong(darkTarget);
    return dark(target);
  }
}
```

添加、优化灰度组件功能

在第一步中，我们完成了灰度组件的基本功能。在第二步中，我们再实现基于编程的灰度规则配置方式，用来支持更加复杂、更加灵活的灰度规则。

我们需要对于第一步实现的代码，进行一些改造。改造之后的代码目录结构如下所示。其中，DarkFeature、DarkRuleConfig的基本代码不变，新增了IDarkFeature接口，DarkLaunch、DarkRule的代码有所改动，用来支持编程实现灰度规则。

```
// 第一步的代码目录结构
com.xzg.darklaunch
  --DarkLaunch(框架的最顶层入口类)
  --DarkFeature(每个feature的灰度规则)
  --DarkRule(灰度规则)
  --DarkRuleConfig(用来映射配置到内存中)

// 第二步的代码目录结构
com.xzg.darklaunch
  --DarkLaunch(框架的最顶层入口类，代码有改动)
  --IDarkFeature(抽象接口)
  --DarkFeature(实现IDarkFeature接口，基于配置文件的灰度规则，代码不变)
  --DarkRule(灰度规则，代码有改动)
  --DarkRuleConfig(用来映射配置到内存中，代码不变)
```

我们先来看下IDarkFeature接口，它用来抽象从配置文件中得到的灰度规则，以及编程实现的灰度规则。具体代码如下所示：

```
public interface IDarkFeature {
  boolean enabled();
  boolean dark(long darkTarget);
  boolean dark(String darkTarget);
}
```

基于这个抽象接口，业务系统可以自己编程实现复杂的灰度规则，然后添加到DarkRule中。为了避免配置文件中的灰度规则热更新时，覆盖掉编程实现的灰度规则，在DarkRule中，我们对从配置文件中加载的灰度规则和编程实现的灰度规则分开存储。按照这个设计思路，我们对DarkRule类进行重构。重构之后的代

码如下所示：

```
public class DarkRule {
  // 从配置文件中加载的灰度规则
  private Map<String, IDarkFeature> darkFeatures = new HashMap<>();
  // 编程实现的灰度规则
  private ConcurrentHashMap<String, IDarkFeature> programmedDarkFeatures = new ConcurrentHashMap<>();

  public void addProgrammedDarkFeature(String featureKey, IDarkFeature darkFeature) {
    programmedDarkFeatures.put(featureKey, darkFeature);
  }

  public void setDarkFeatures(Map<String, IDarkFeature> newDarkFeatures) {
    this.darkFeatures = newDarkFeatures;
  }

  public IDarkFeature getDarkFeature(String featureKey) {
    IDarkFeature darkFeature = programmedDarkFeatures.get(featureKey);
    if (darkFeature != null) {
      return darkFeature;
    }
    return darkFeatures.get(featureKey);
  }
}
```

因为DarkRule代码有所修改，对应地，DarkLaunch的代码也需要做少许改动，主要有一处修改和一处新增代码，具体如下所示，我在代码中都做了注释，就不再重复解释了。

```
public class DarkLaunch {
  private static final Logger log = LoggerFactory.getLogger(DarkLaunch.class);
  private static final int DEFAULT_RULE_UPDATE_TIME_INTERVAL = 60; // in seconds
  private DarkRule rule = new DarkRule();
  private ScheduledExecutorService executor;

  public DarkLaunch(int ruleUpdateTimeInterval) {
    loadRule();
    this.executor = Executors.newSingleThreadScheduledExecutor();
    this.executor.scheduleAtFixedRate(new Runnable() {
      @Override
      public void run() {
        loadRule();
      }
    }, ruleUpdateTimeInterval, ruleUpdateTimeInterval, TimeUnit.SECONDS);
  }

  public DarkLaunch() {
    this(DEFAULT_RULE_UPDATE_TIME_INTERVAL);
  }

  private void loadRule() {
    InputStream in = null;
    DarkRuleConfig ruleConfig = null;
    try {
      in = this.getClass().getResourceAsStream("/dark-rule.yaml");
      if (in != null) {
        Yaml yaml = new Yaml();
        ruleConfig = yaml.loadAs(in, DarkRuleConfig.class);
```

```
      }
    } finally {
      if (in != null) {
        try {
          in.close();
        } catch (IOException e) {
          log.error("close file error:{}", e);
        }
      }
    }

    if (ruleConfig == null) {
      throw new RuntimeException("Can not load dark rule.");
    }

    // 修改：单独更新从配置文件中得到的灰度规则，不覆盖编程实现的灰度规则
    Map<String, IDarkFeature> darkFeatures = new HashMap<>();
    List<DarkRuleConfig.DarkFeatureConfig> darkFeatureConfigs = ruleConfig.getFeatures();
    for (DarkRuleConfig.DarkFeatureConfig darkFeatureConfig : darkFeatureConfigs) {
      darkFeatures.put(darkFeatureConfig.getKey(), new DarkFeature(darkFeatureConfig));
    }
    this.rule.setDarkFeatures(darkFeatures);
  }

  // 新增：添加编程实现的灰度规则的接口
  public void addProgrammedDarkFeature(String featureKey, IDarkFeature darkFeature) {
    this.rule.addProgrammedDarkFeature(featureKey, darkFeature);
  }

  public IDarkFeature getDarkFeature(String featureKey) {
    IDarkFeature darkFeature = this.rule.getDarkFeature(featureKey);
    return darkFeature;
  }
}
```

灰度组件的代码实现就讲完了。我们再通过一个Demo来看下，目前实现的灰度组件该如何使用。结合着Demo，再去理解上面的代码，会更容易些。Demo代码如下所示：

```
// 灰度规则配置(dark-rule.yaml)，放到classpath路径下
features:
- key: call_newapi_getUserById
  enabled: true
  rule: {893,342,1020-1120,%30}
- key: call_newapi_registerUser
  enabled: true
  rule: {1391198723, %10}
- key: newalgo_loan
  enabled: true
  rule: {0-100}

// 编程实现的灰度规则
public class UserPromotionDarkRule implements IDarkFeature {
  @Override
  public boolean enabled() {
    return true;
  }

  @Override
  public boolean dark(long darkTarget) {
```

```
    // 灰度规则自己想怎么写就怎么写
    return false;
  }

  @Override
  public boolean dark(String darkTarget) {
    // 灰度规则自己想怎么写就怎么写
    return false;
  }
}

// Demo
public class Demo {
  public static void main(String[] args) {
    DarkLaunch darkLaunch = new DarkLaunch(); // 默认加载classpath下dark-rule.yaml文件中的灰度规则
    darkLaunch.addProgrammedDarkFeature("user_promotion", new UserPromotionDarkRule()); // 添加编程实现的灰度规
    IDarkFeature darkFeature = darkLaunch.getDarkFeature("user_promotion");
    System.out.println(darkFeature.enabled());
    System.out.println(darkFeature.dark(893));
  }
}
```

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

到今天为止，项目实战环节就彻底结束了。在这一部分中，我们通过限流、幂等、灰度这三个实战项目，带你从需求分析、系统设计、代码实现这三个环节，学习了如何进行功能性、非功能性需求分析，如何通过合理的设计，完成功能性需求的同时，满足非功能性需求，以及如何编写高质量的代码实现。

实际上，项目本身的分析、设计、实现并不重要，不必对细节过于纠结。我希望通过这三个例子，分享我的思考路径、开发套路，让你借鉴并举一反三地应用到你平时的项目开发中。我觉得这才是最有价值的，才是你学习的重点。

如果你学完这一部分之后，对于项目中的一些通用的功能，能够开始下意识地主动思考代码复用的问题，考虑如何抽象成框架、类库、组件，并且对于如何开发，也不再觉得无从下手，而是觉得有章可循，那我觉得你就学到了这一部分的精髓。

课堂讨论

在DarkFeature类中，灰度规则的解析代码设计的不够优雅，你觉得问题在哪里呢？又该如何重构呢？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言：

- tingye 2020-06-17 07:53:04
  可以考虑用职责链模式，将不同规则字符串的解析抽象为单独的handle类，依次解析直到完成处理，也方便扩展对新规则编写语法的解析 [3赞]

- 小晏子 2020-06-17 10:51:21
  这个DarkFeature类中灰度规则的解析代码不优雅的地方在于不够灵活，如果有新的灰度规则要加入，就需要再添加if else作处理，破坏了开闭原则，为了解决这一问题，可以使用工厂模式✚策略模式来保证开

闭原则和消除if/else，使用工厂模式来实现针对每个灰度规则的处理，使用“查表法”的策略模式来消除if/else！ [2赞]

- Lee 2020-06-17 08:35:53
  可以使用解释器模式，将不同类型的规则解析拆分到不同的类中。 [2赞]

- Jxin 2020-06-17 13:26:47
  1.定时任务在方法内部创建和使用，这样就没办法手动调定时任务的退出方法了。

  2.感觉业务接口的路由规则的选型和路由规则的具体实现应该分离。DarkFear里面应该只要表明，哪个业务接口用哪个灰度规则，这个意图就好。至于灰度规则的具体实现，包括dsl的解析和灰度规则的执行都应该剥离出来单独封装。 [1赞]

- test 2020-06-17 09:05:51
  parsedarkrule的代码可以单独出来 [1赞]

- Heaven 2020-06-17 11:46:12
  在此类场景下,我们可以简单的使用工厂类去封装规则的解析,
  但是我个人觉着,应该以配置文件中配置的规则为主,所以,第二版需要在配置文件中写上实现接口的全限定类名,反射获取实例,同样支持更新,这样配置文件的Map就可以移除了,而且可以将简单的原生三种解析规则也抽象为接口,利用策略类进行区分调用

- 罗乾林 2020-06-17 10:29:58
  “DarkFeature 类中，灰度规则的解析代码“，是我能想到最直接简单的方式。我想可以抽象出规则解析类，对规则的解析交个解析类处理，将解析类对象注入到DarkFeature 类中。这样DarkFeature职责更单一

- gogo 2020-06-17 09:11:22
  可以考虑引入策略模式和工厂模式，消除if else

- leezer 2020-06-17 08:55:31
  解析规则可以参考之前，使用工厂模式回去解析器，通过对应的解析器进行解析对应的配置文件.

- robincoin 2020-06-17 04:48:03
  mq和数据库灰度是不是要对mq和数据库再封装一层，方便aop处理?

  作者回复2020-06-19 08:14:05
  支持! 封装来进一步简化开发！