Spring Cloud应用与实践

引入背景

泰然集团秉承着构筑多元化生态体系的经营理念，近些年来得到了长足地成长。业务不断地高歌猛进，给技术带来了很大的压力。我们的架构经历了单体到基于Dubbo的服务化迁移。目前各个生态链公司大大小小有近几十个项目都以Dubbo为核心构建，但这过程中我们也看到一些问题，且随着服务体量与数量的增加越发明显。

毋庸置疑，Dubbo是一个设计精良、久经考验的优秀框架，但同时我们也看到由于产品定位、时代的局限以及社区建设的缺陷导致了如下问题：

1. Dubbo定位于服务调用及治理，但缺少诸如服务网关、配置管理、智能断路、日志调用追踪等对服务化建设至关重要的功能/组件，在实施上我们不得不寻求其它的工具补充这些短板，这无形中增加了开发架构的成本
2. Dubbo出现的年代持续集成（CI）/部署（CD）、DevOps以及容器化、云环境方案都未成气候，这直接导致它在设计架构及运维层面都缺少对这些新技术、流程的友好支持，在运维自动化日趋重要及成熟的今天，Dubbo有些格格不入；另外语言层面越发百花齐放，Dubbo官方只支持JVM下的服务治理，制约技术多元化地发展
3. Dubbo在国内的影响力很高，Star及Fork的人数众多，但在社区建设上却日渐式微，曾有当当等企业做Dubbo做二次封装，但都无疾而终，这么成熟的框架没有交给开源基金会维护，阿里对Dubbo有绝对主导权，这直接导致了阿里不参与Dubbo就不维护的窘境，在我们决定从Dubbo转向Spring Cloud时Dubbo已有3年没有更新，Github上的Issue堆积如山。目前Dubbo又开始维护了[^1]，但多半是基于（阿里云的）商业利益，历史经验告诉我们一个事物如果只被一家商业性质的企业支配极有可能导致不和谐的结局，如苹果对IOS支付的规则修改、Facebook对RN协议的变更等

> UPDATE: 近期Dubbo已准备捐赠给Apache，18年2月15号Apache通过接纳投票进入Incubator[^2]，但离成为正式Apache项目还要比较长的时间。

以上几点是我们对Dubbo使用问题的总结，而彼时正好微服务架构如日中天，开源市场上也出现了几个很不错的成熟框架，这之中Spring Cloud更是众星捧月般地成为了无冕之王，在经历了Dubbo的服务化改造后我们对新框架的选型提出了如下几个要求：

- 社区成熟，我们深知服务框架对系统架构的影响是深入、全面、持久的，所以必须是Spring核心项目或是诸如Apache、Eclipse等知名基金会负责、众多厂商参与，有着广泛的用户基础、长时间版本迭代的项目
- 生态完善，我们是业务驱动型公司，技术底层的投入有限，原则上我们能用开源就避免自研，所以我们期望的是有完整微服务生态的框架以尽量减少“拼积木”式的各个框架整合精力
- 开发快捷，服务化的同时应尽量降低开发使用成本，业务逻辑的撰写以主流、简单的技术为宜
- 异构支持，目前公司以Java为主，Node和PHP为辅，未来会在一些试错型或业务易变的场景下使用Node等语言实现更灵活快速的业务影响，所以要求服务化提供对异构语言的支持
- 运维自动化，我们期望框架能对网络、容器提供友好的支持，可以简单地实现服务伸缩、智能负载

综合上述要求，我们发现Spring Cloud非常匹配，它的社区影响力/组件覆盖度、Spring自身的美誉度、Spring Boot开发的简洁化、Spring Cloud Pipelines等项目对运维的支撑度都可圈可点。因此我们决定新项目从Dubbo转向Spring Cloud。

Spring Cloud简介

Spring Cloud是一个基于Spring Boot实现的微服务框架，它实现了微服务架构涉及的核心功能，简化了开发难度，提升了工作效率。其常见的组件有：

- Spring Cloud Config（配置中心）：通过共享文件系统、JDBC、SVN或Git等方式实现服务配置的动态化，降低了由于配置问题而引发的一系列问题。Spring Cloud Config推荐基于Git的方式，以提供版本化管理，考虑到配置数据的敏感性，它也提供了简洁的加解密方案
- Eureka（注册中心）：Spring Cloud由 Netflix OSS 发展而来，目前版本尚留有不少Netflix OSS的身影，Eureka就是由后者提供的注册中心，它提供了完整的Service Registry和Service Discovery实现及高可用支持。同时也有诸如Spring Cloud Zookeeper等多种方案可供选择，不过 Eureka 具有心跳管理、客户端缓存、部署维护简单等优势
- Zuul（服务网关）：把一些通用的功能，例如鉴权、流控、监控、日志统计，通过统一的服务进行处理，降低管理和发布成本。Zuul提供ZuulFilter、隔离机制和重试机制满足多种复杂场景要求
- Spring Boot Admin（监控中心）：通过对服务Health信息、内存信息、JVM信息、垃圾回收等信息的可视化监控，实现对服务状态的掌握，它还提供URLPath配置功能，方便统一管理监控API

Spring Cloud更多核心项目见： http://projects.spring.io/spring-cloud/

Spring Cloud在泰然的实践

我们集团构筑的生态链涉及理财、贷款、商城、良品、保险、企业购、公益等诸多领域，在研发上目前有250人左右，细分了很多项目团队。

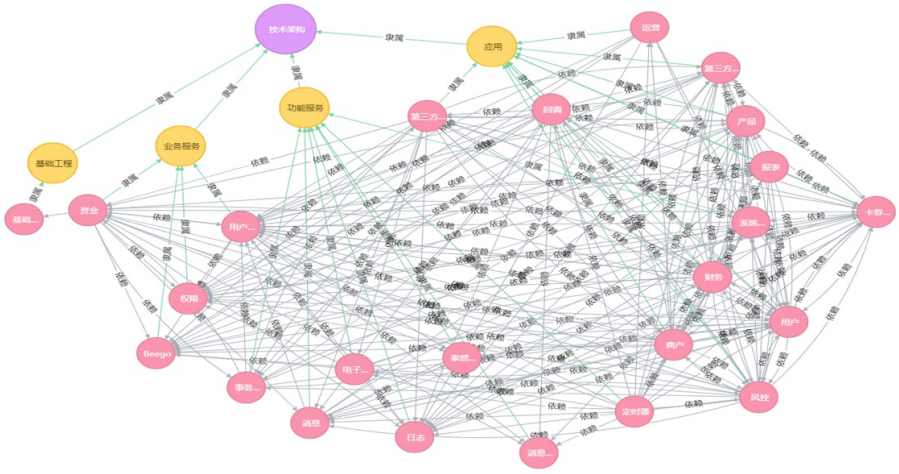
TBD 补充组织架构图

在Dubbo服务化过程中我们发现了如下一些问题：

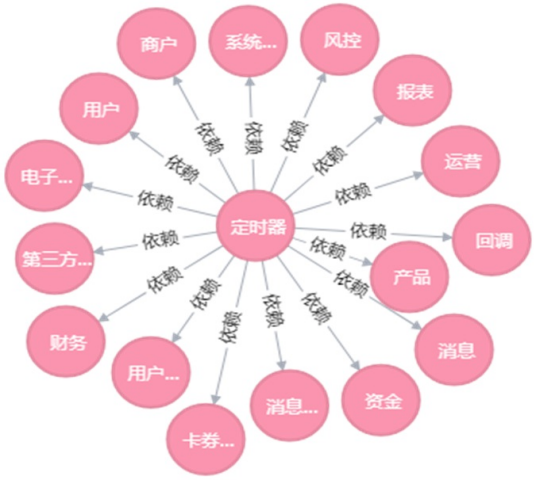
工程粒度过细

以贷款业务为例，117个后端工程，25个服务，一个服务包含3-6个工程，微服务的确要求做细粒度的划分，但这对人员、运维、制度等提出了很高的要求。

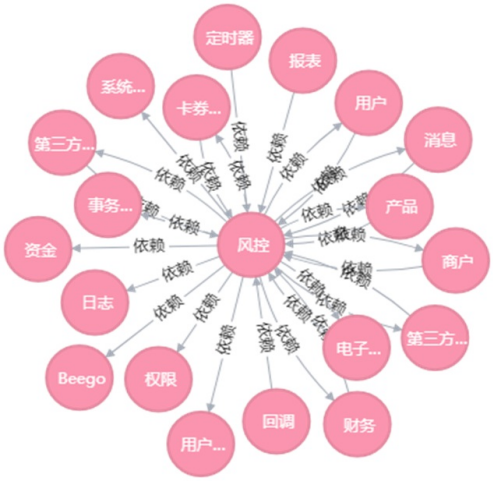
模块边界定义模糊



上述业务中，25个服务（模块）中有个别模块存在明确的技术交集（如mqnotify、mqserver）及业务包含关系（如operation、activity）可以合并，不少模块做为基础服务却混入了上层业务依赖（如quartz、mqnotify等几乎依赖了所有业务模块）。



存在很多双向依赖



由于边界划定错误，使得服务之间存在很多的双向依赖，给开发、运维造成了很大的困扰。

服务复用低

公共服务被严重弱化，一方面抽象的公共服务过少，另一方面对已有公共服务缺少长期稳定的维护保障，使得部分需求无法满足业务要求，导致项目团队不得不自研或二次开发相关服务。

对外接口未抽象

由于Dubbo是RPC协议，且与Spring整合使得远程调用与本地服务调用几乎无差别，所以开发人员习惯性地把未经抽象的诸如对某一实体原子化的CRUD方法都直接暴露出来，这使得服务调用方需要了解服务细节，严重违反迪米特法则。

发布流程不规范

部分项目团队缺少UAT/预发环境，导致业务方需要在生产环境做验收，且验收数据不清理；并且由于配置管理的不规范（配置信息写在Pom文件中）导致测试环境打的包无法直接用于生产环境，发布时需要重新打生产包，浪费时间且有可能出现打包失败或前后代码不一致（对Git Tag的不规范使用）等情况。

另外我们也发现存在诸如人员与服务依赖、业务与服务依赖等非技术问题，此章节更多信息请参阅：《蜂贷架构设计审计报告》。

为解决上述问题，我们在Spring Cloud改造的同时也提出了以下几点举措：

- 合理的划分服务边界，具体做法参照本公司的《微服务架构设计》

以业务、技术、团队导向规划服务

服务不是越细越好

先以业务域水平拆分，再以技术视角垂直拆分，结合团队的规模、能力确定

使用领域模型划分业务域

DAG检查

服务间应该避免双向依赖，合理的服务调用应该是DAG图

分布式事务检查

服务拆分尽量避免产生跨服务事务，能合则合

性能分布检查

服务中对于特别耗资源的操作应尽量独立

稳定（易变）性检查

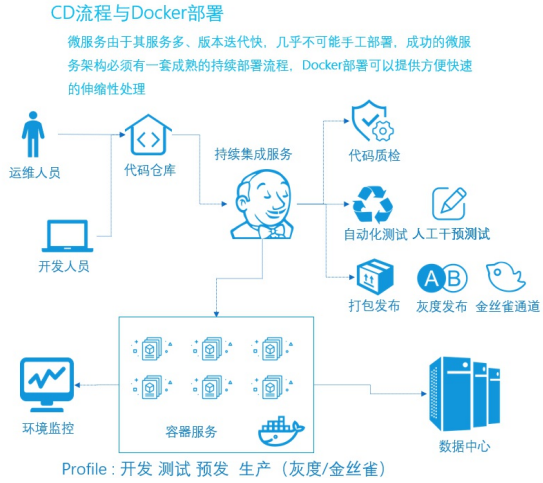
一个服务中如存在稳定和不稳定的模块，应该将两者拆分

调用链检查

服务间调用有IO消耗，调用链路不宜过长

- 强化公共服务建设，正式成立专职的公共服务团队，主导或参与各个公共服务的研发，详见“公共服务Wiki”

- 规范CD流程，努力实现DevOps，我们与运维同仁一起引入K8s，实现测试环境全面的Docker化，并在生产环境中小规模地尝试，在部分团队中我们将业务、产品、研发、测试、运维以Scrum为基础组成产品闭环，尝试更高效地协作



- 简化服务接口，开发要求绝大分部服务接口都要面向业务而非功能，引入类似facade模式，为服务调用者提供简洁明了的服务体验，在Spring Cloud中可以直接使用Controller来封装服务

我们要求所有的新项目都转向Spring Cloud，对于一些已有的核心项目逐步进行，目前已有公共服务基础支撑平台、消息中心、ATS、资金服务、电子签名、分布式调用服务等诸多项目基于Spring Cloud实现，蜂贷3.0也初步完成了Spring Cloud化改造。

在这过程中，我们对Spring Cloud的理解及使用也逐步增强，逐渐地总结出最佳实践，并以些为基础对其进行完善，形成了Dew Framework这一对Spring Cloud的扩展框架。

对Spring Cloud的优化

Dew Framework对Spring Cloud的扩展主要分三部分：核心优化、周边增强和工程化扩展。

核心优化

集成了Dew-Common 提供了诸如Json支持、Java Bean操作、加解密、Http操作、常用字段处理等功能，详见：https://github.com/gudaoxuri/dew-common

Web处理优化 添加了web基础依赖、默认支持CORS，由于我们所有项目都是前后端分离结构，所以排除了一些与前端模板相关的AutoConfiguration，并且对Spring Cloud各个场景下的返回做了统一响应及异常处理。

文档优化 对开发、测试环境默认启用swagger，并且实现了便捷的HTML/PDF离线文档生成。

数据访问 Spring Boot支持 Hibernat、MyBatis、Spring JDBC Template 等主流的持久化框架。 我们对 Spring JDBC Template 这一轻量的数据处理框架做了一定的扩展：1）支持实体与SQL的映射，2）支持常用数据处理操作，3）支持@Select注解，4）轻松使用多数据源。

集群功能 支持 分布式缓存、分布式Map、分布式锁、MQ、Leader Election，并且做了接口抽象以适配不同的实现，目前支持 Redis、Hazelcast、Rabbit、Kafka、Ignite、Eureka 等。

周边增强

服务脚手架 提供了对实体的CRUD封装。

权限认证 两种认证：1）Basic模式，支持认证缓存，即支持将鉴权系统生成的登录信息缓存到业务系统中方便即时调用。这是比较通用的方案，2）CSP模式，支持用户中心权限系统，这是我们公司内部的集成认证方案。

Dubbo兼容 Dubbo官方发行版本（2.5.3版本）无法处理存在声明式事务的服务，Dew对其进行了一定地修改。

sharding-jdbc集成 支持局部sharding，即可指定某些数据源使用sharding-jdbc。

mybatis多数据源优化 扩展了mybatis实现了注解式的多数据源支持。

幂等支持 支持HTTP和非HTTP幂等操作，对于HTTP操作，要求请求方在请求头或URL参数中加上操作类型和操作ID标识，非HTTP操作由可自由指定操作类型和操作ID标识的来源。

工程化扩展

Spring Cloud服务于大规模分布式应用，在工程化上比其它各类框架支持得更到位，但同时我们也看到由于它过于年轻，尚存在一些不足之处，为此Dew做了如下扩展：

代码质量检查 集成Sonar插件，以类似mvn clean verify sonar:sonar -P qa的命令一键执行代码质量检查。

单元测试优化 单测的重要原则是内聚、无依赖，好的单测应该是"函数化"的——结果的变化只与传入参数有关。但实际上我们会的代码往往会与数据库、缓存、MQ等外部工具交互，这会使单测的结果不可控，通常的解决方案是使用Mock，但这无行中引入了单测撰写的成本，Dew使用"内嵌式"工具解决，数据库使用 H2 ，Redis使用 embedded redis ，由于 Dew 集群的 Cache、Map、Lock、MQ 都支持 Redis 实现，所以可以做到对主流操作的全覆盖。

Hystrix 降级增加邮件通知 Hystrix是个很好的智能断路工具，Spring Cloud与之有比较优雅地适配，但它缺少降级主动通知功能。Dew通过继承HystrixEventNotifier，重写其markEvent方法，添加处理逻辑，以达到邮件通知功能，后续可以加入短信、微信等通道。

服务API调用 （追踪）日志处理 在Spring Cloud中 服务API调用日志 可选择 Sleuth + Zipkin 的方案， Dew 没有选择 Zipkin 理由如下：

- Zipkin 需要再部署一套 Zipkin 服务，多了一个依赖
- Zipkin 日志走 HTTP 协议对性能有比较大的影响，走 MQ 方案又会让使用方多了一个技术依赖
- Zipkin 日志存储方案中 MySQL 有明显的问题， Cassandra 不错，但选型比较偏， ES 最为合适
- Zipkin 方案导致 服务API调用日志 与 应用程序日志不统一，后则多选择 ELK 方案

所以Dew框架采用的是 Sleuth + Slf4j + ES 的方案并做了一定的优化，使用ElasticsearchAppender批量提交到ES中。当然这一方案会损失一定的可读性，即没有可视化的接口调用展现。

一次调用日志的查看，在ES中的过滤条件是: logger:com.tairanchina.csp.dew.core.logger.DewTraceLogWrap & trace:<对应的traceID> 。

metrics统计增强 Metrics作为Spring Boot Admin服务中的一个维度监控指标，提供Gauge、Counter、Meter等度量维度以了解运行中内存使用量和CPU占用率状况。随着Docker技术的成熟，我们可以通过对服务健康的监控，实现服务弹性化部署，若服务处于高并发压力时，动态增加服务部署数量做出应对，压力结束之后，再对资源进行回收。其中TPS指标，最大响应时间，平均响应时间，90%的响应时间是很重要的指标，现有Metrics没有提供该数据，Dew扩展Spring Boot Actuator提供的Metrics接口，实现了上述几个指标的采集。

Hystrix降级增加邮件通知

Spring Cloud 提供的Hystrix降级是一个提升服务可用性的功能，但由于降级对开发者和运维工程师是无感知的，为了解决这个问题，我们添加了邮件通知的功能，通知相关人员作出应对措施，以防止长期保持服务降级状态。

通过继承HystrixEventNotifier，重写其markEvent方法，添加处理逻辑，以达到邮件通知功能。以下邮件通知功能的配置项。

Spring Cloud最佳实践

随着Spring Cloud项目的增多，我们踩过的坑也越来越多，对Spring Cloud的理解了越来越深，这里总结一些我们的最佳实践。

开发调试

在 Spring Cloud 体系下，服务调用需要依赖Eureka服务，出现了如下的问题：

- 开发期间会不断启停服务，Eureka保护机制会影响服务注册（当然这是可以关闭的）
- 多人协作时可能会出现调用到他人服务的情况（同一服务多个实例）
- 需要启动Eureka服务，多了一个依赖

解决这个问题，可以在使用 Spring Cloud 的RestTemplate时，增加Ribbon的服务配置

```
# <client>为service-id
<client>.ribbon.listOfServers: <直接访问的IPs>
# e.g.
performance-service.ribbon.listOfServers: 127.0.0.1:8812
```

参数校验

传统的方式开发接口，参数校验方面会存在如下几个问题：

- 大量的If语句，影响美观
- 开发注意力分散
- 代码可读性不高

使用@Validated 注解方式可以有效避免以上问题。以下是一些使用注意点：

1. 对于基本数据类型和String类型，要使校验的注解生效，需在该类上方加 @Validated 注解
2. 对于抽象数据类型，需在形式参数前加@Validated注解

Ribbon负载均衡策略

Spring Cloud 支持service-dew.ribbon.NFLoadBalancerRuleClassName=com.netflix.loadbalancer.RandomRule方式选择负载均衡策略，因此你可以这样：

若指定zone，默认会优先调用相同zone的服务，此优先级高于策略配置，配置如下

```
#指定属于哪个zone
eureka:
  instance:
    metadata-map:
      zone: #zone 名称

#指定region（此处region为项目在不同区域的部署，为项目规范，不同region间能互相调用）
eureka:
  client:
    region: #region名称
```

Feign配置特定方法超时时间

Hystrix和Ribbon的超时时间配置相互独立，**以低为准**，使用时请根据实际情况进行配置，配置项可以参考

```
# Hystrix 超时时间配置
# 配置默认的hystrix超时时间
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=10000
# 配置特定方法的超时时间,优于默认配置
hystrix.command.<hystrixcommandkey>.execution.isolation.thread.timeoutInMilliseconds=10000
# <hystrixcommandkey>的format为FeignClassName#methodSignature,下面是示例配置
hystrix.command.PressureService#getBalance(int).execution.isolation.thread.timeoutInMilliseconds=10000


# Ribbon 超时时间配置
# 配置默认ribbon超时时间
ribbon.ReadTimeout=60000
# 配置特定服务超时时间,优于默认配置
<client>.ribbon.ReadTimeout=6000
# <client>为实际服务名,下面是示例配置
pressure-service.ribbon.ReadTimeout=5000
```

> 如果要针对某个服务做超时设置,建议使用Ribbon的配置；在同时使用Ribbon和Hystrix 时,请特别注意超时时间的配置。

Feign接口添加HTTP请求头信息

Feign作为仿SDK的时候，由于其本质是通过HTTP的方式实现，需要考虑Header配置问题。

在@FeignClient修饰类中的接口方法里添加新的形参，并加上 @RequestHeader 注解指定key值

e.g.

```
@PostMapping(value = "ca/all", consumes = MediaType.APPLICATION_JSON_VALUE)
Resp<CustomerInfoVO> applyCA(@RequestBody CAIdentificationDTO params,
     @RequestHeader Map<String, Object> headers);
```

Feign文件上传实践

在仿SDK中，文件的上传功能实现方案如下：

- pom添加依赖配置

```
        <dependency>
            <groupId>io.github.openfeign.form</groupId>
            <artifactId>feign-form</artifactId>
            <version>3.0.1</version>
        </dependency>
        <dependency>
            <groupId>io.github.openfeign.form</groupId>
            <artifactId>feign-form-spring</artifactId>
            <version>3.0.1</version>
        </dependency>
```

- 创建一个Configuration

```
import feign.codec.Encoder;
import feign.form.spring.SpringFormEncoder;
import org.springframework.beans.factory.ObjectFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.web.HttpMessageConverters;
import org.springframework.cloud.netflix.feign.support.SpringEncoder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MultipartSupportConfig {

    @Autowired
    private ObjectFactory<HttpMessageConverters> messageConverters;

    @Bean
    public Encoder feignFormEncoder() {
        return new SpringFormEncoder(new SpringEncoder(messageConverters));
    }

}
```

- 修改Client接口代码

```
@FeignClient(name = "demo")
public interface FeginExample {
@PostMapping(value = "images", consumes = MULTIPART_FORM_DATA_VALUE)
 Resp<String> uploadImage(
            @RequestParam MultipartFile image,
            @RequestParam("id") String id);
}
```

> @RequestPart 与 @RequestParam 效果是一样的，大家就不用花时间在这上面了

- 修改Service接口代码

```
@RestController
public class FeginServiceExample {
  @PostMapping(value = "images", consumes = MULTIPART_FORM_DATA_VALUE)
    public Resp<String> uploadImage(
            @RequestParam("image") MultipartFile image,
            @RequestParam("id") String id,
            HttpServletRequest request) {
              return Resp.success(null);
            }
}
```

常见问题：

- HTTP Status 400 - Required request part 'file' is not present

> 请求文件参数的名称与实际接口接受名称不一致

- feign.codec.EncodeException: Could not write request: no suitable HttpMessageConverter found for request type [org.springframework.mock.web.MockMultipartFile] and content type [multipart/form-data]

> 转换器没有生效，检查一下MultipartSupportConfig

服务自定义降级

在微服务架构中，存在服务间的相互依赖场景。如果某一个服务发生了宕机，保证请求的正常响应，采用服务的降级方案以防止系统服务的大面积瘫痪而导致必要的经济损失。

> 构建类继承HystrixCommand抽象类，重写run方法，getFallback方法，getFallback为run的降级，再执行excute方法即可
>
> 每个HystrixCommand的子类的实例只能excute一次

e.g.

```
public class HelloHystrixCommand extends HystrixCommand<HelloHystrixCommand.Model> {

    public static final Logger logger = LoggerFactory.getLogger(HelloHystrixCommand.class);

    private Model model;

    protected HelloHystrixCommand(HystrixCommandGroupKey group) {
        super(group);
    }

    protected HelloHystrixCommand(HystrixCommandGroupKey group, HystrixThreadPoolKey threadPool) {
        super(group, threadPool);
    }

    protected HelloHystrixCommand(HystrixCommandGroupKey group, int
executionIsolationThreadTimeoutInMilliseconds) {
```

```
        super(group, executionIsolationThreadTimeoutInMilliseconds);
    }

    protected HelloHystrixCommand(HystrixCommandGroupKey group, HystrixThreadPoolKey threadPool, int
executionIsolationThreadTimeoutInMilliseconds) {
        super(group, threadPool, executionIsolationThreadTimeoutInMilliseconds);
    }

    protected HelloHystrixCommand(Setter setter) {
        super(setter);
    }

    public static HelloHystrixCommand getInstance(String key){
        return new HelloHystrixCommand(HystrixCommandGroupKey.Factory.asKey(key));
    }

    @Override
    protected Model run() throws Exception {
        int i = 1 / 0;
        logger.info("run:   thread id:  " + Thread.currentThread().getId());
        return model;
    }

    @Override
    protected Model getFallback() {
        return new Model("fallback");
    }

    public static void main(String[] args) throws Exception {
        HelloHystrixCommand helloHystrixCommand = HelloHystrixCommand.getInstance("dew");
        helloHystrixCommand.model = helloHystrixCommand.new Model("run");
        logger.info("main:      " + helloHystrixCommand.model + "thread id: " +
Thread.currentThread().getId());
        System.out.println(helloHystrixCommand.execute());

    }

    class Model {

        public Model(String name) {
            this.name = name;
        }

        private String name;

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        @Override
        public String toString() {
            return "Model{" +
                    "name='" + name + '\'' +
                    '}';
        }
    }
}
```

断路保护经验分享

```
# 执行的隔离策略 THREAD, SEMAPHORE 默认THREAD
hystrix.command.default.execution.isolation.strategy=THREAD
# 执行hystrix command的超时时间,超时后会进入fallback方法 默认1000
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=1000
# 执行hystrix command是否限制超时,默认是true
hystrix.command.default.execution.timeout.enabled=true
# hystrix command 执行超时后是否中断 默认true
hystrix.command.default.execution.isolation.thread.interruptOnTimeout=true
# 使用信号量隔离时,信号量大小,默认10
hystrix.command.default.execution.isolation.semaphore.maxConcurrentRequests=10
# fallback方法最大并发请求数 默认是10
hystrix.command.default.fallback.isolation.semaphore.maxConcurrentRequests=10
# 服务降级是否开启,默认为true
hystrix.command.default.fallback.enabled=true
# 是否使用断路器来跟踪健康指标和熔断请求
hystrix.command.default.circuitBreaker.enabled=true
# 熔断器的最小请求数,默认20. (这个不是很理解,欢迎补充)
hystrix.command.default.circuitBreaker.requestVolumeThreshold=20
# 断路器打开后的休眠时间,默认5000
hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds=5000
# 断路器打开的容错比,默认50
hystrix.command.default.circuitBreaker.errorThresholdPercentage=50
# 强制打开断路器,拒绝所有请求. 默认false, 优先级高于forceClosed
hystrix.command.default.circuitBreaker.forceOpen=false
# 强制关闭断路器,接收所有请求,默认false,优先级低于forceOpen
hystrix.command.default.circuitBreaker.forceClosed=false

# hystrix command 命令执行核心线程数,最大并发 默认10
hystrix.threadpool.default.coreSize=10
```

信息参见:

- https://github.com/Netflix/Hystrix/wiki/Configuration
- http://hwood.lofter.com/post/1cc7fbdc_e8c5c96

使用断路保护可有效果的防止系统雪崩，Spring Cloud 对Hystrix做了封装。

详见： http://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#_circuit_breaker_hystrix_clients

需要说明的是Hystrix使用新线程执行代码，导致Threadlocal数据不能同步，为解决这个问题我们是这样做的，可供参考：

```
public class HystrixExampleService {
    @HystrixCommand(fallbackMethod = "defaultFallback", commandProperties = {
            @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value =
"2000")
    })
    public String someMethod(Map<String, Object> parameters, DewContext context) {
        // ！！！ Hystrix使用新线程执行代码，导致Threadlocal数据不能同步，
        // 使用时需要将用到的数据做为参数传入，如果需要使用Dew框架的上下文需要先传入再设值
        DewContext.setContext(context);
        try {
            Thread.sleep(new Random().nextInt(3000));
            logger.info("Normal Service Token:" + Dew.context().getToken());
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return "ok";
    }

    // 降级处理方法定义
    public String defaultFallback(Map<String, Object> parameters, DewContext context, Throwable e) {
        DewContext.setContext(context);
```

```
        logger.info("Error Service Token:" + Dew.context().getToken());
        return "fail";
    }
}
```

定时任务

使用 Spring Config 配置中心refresh时,在@RefreshScope注解的类中, @Scheduled注解的自动任务会失效。 建议使用实现 SchedulingConfigurer 接口的方式添加自动任务。

e.g.

```
@Configuration
@EnableScheduling
public class SchedulingConfiguration implements SchedulingConfigurer {

    private Logger logger = LoggerFactory.getLogger(SchedulingConfiguration.class);

    @Autowired
    private ConfigExampleConfig config;

    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
        taskRegistrar.addTriggerTask(() -> logger.info("task1: " + config.getVersion()),
triggerContext -> {
            Instant instant = Instant.now().plus(5, SECONDS);
            return Date.from(instant);
        });

        taskRegistrar.addTriggerTask(() -> logger.info("task2: " + config.getVersion()), new
CronTrigger("1/3 * * * * ?"));
    }
}
```

Spring Cloud配置扫描策略

Spring Boot 所提供的配置优先级顺序比较复杂。按照 **优先级从高到低** 的顺序，具体的列表如下所示：

1. 命令行参数。
2. 通过 System.getProperties() 获取的 Java 系统参数。
3. 操作系统环境变量。
4. 从 java:comp/env 得到的 JNDI 属性。
5. 通过 RandomValuePropertySource 生成的 random.* 属性。
6. 应用 jar 文件之外的属性文件。(通过 spring.config.location 参数)
7. 应用 jar 文件内部的属性文件。
8. 在应用配置 Java 类（包含 @Configuration 注解的 Java 类）中通过 @PropertySource 注解声明的属性文件。
9. 通过 SpringApplication.setDefaultProperties 声明的默认属性。

最佳实践：

属性文件是比较推荐的配置方式。Spring Boot在启动时会对如下目录进行搜查，读取相应配置文件。优先级从高到低。

1. 当前jar目录的 /config 子目录
2. 当前jar目录
3. classpath中的 /config 包
4. classpath

> 可以通过 spring.config.name 配置属性来指定不同的属性文件名称。也可以通过 spring.config.location 来添加额外的属性文件的搜索路径。如果应用中包含多个profile，可以为每个profile定义各自的属性文件，按照 application-{profile} 来命名。 Spring Cloud Config 配置方式就属于这种配置方式，其优先级低于本地jar外配置文件

使用Profile区分环境：

在Spring Boot中可以使用 application.yml ， application-default.yml ， application-dev.yml ， application-test.yml 进行不同环境的配置。默认时，会读取 application.yml ， application-default.yml 这两个文件中的配置，优先级高的会覆盖优先级低的配置。无论切换到哪个环境，指定的环境的配置的优先级是最高的。

可以使用 spring.profiles.active=dev 指定环境。

Tips

- bootstrap.yml (jar包外) > bootstrap.yml (jar包内) > application.yml (jar包外) > application.yml (jar包内)
- 当启用Spring Cloud Config配置后， Git仓库中的 application.yml > [application-name].yml >[application-name]-[profile].yml
- 配置的覆盖不是以文件为单位，而是以配置中的参数为单位。
- 同一参数取最先扫描到的value

Zuul保护(隐藏)内部服务的HTTP接口

存在一个服务之间使用的内部接口，由于涉及到安全问题，不希望对外进行暴露，因而希望能将其只对内网可用。解决这类问题可以通过如下配置：

在yml配置文件里配置(ignored-patterns,ignored-services)这两项中的一项即可

e.g.

```
zuul: #配置一项即可!
  ignored-patterns: /dew-example/**   #排除此路径
  ignored-services: dew-example       #排除此服务
```

缓存超时实现

Spring Cache提供了很好的注解式缓存，但默认没有超时配置功能，需要根据使用的缓存容器特殊配置。可参考以下方案：

e.g.

```
@Bean
RedisCacheManager cacheManager() {
    final RedisCacheManager redisCacheManager = new RedisCacheManager(redisTemplate);
    redisCacheManager.setUsePrefix(true);
    redisCacheManager.setDefaultExpiration(<过期秒数>);
    return redisCacheManager;
}
```

servo内存泄漏问题

已知在某些情况下servo统计会导致内存泄漏，如无特殊需要建议关闭

```
spring.metrics.servo.enabled: false
```

Spring Boot Admin 监控实践

在Spring Boot Actuator中提供很多像 health 、 metrics 等实时监控接口，可以方便我们随时跟踪服务的性能指标。Spring Boot 默认是开放这些接口提供调用的，那么就问题来了，如果这些接口公开在外网中，很容易被不法分子所利用，这肯定不是我们想要的结果。 在这里我们提供一种比较好的解决方案：

- 被监控的服务配置

```
management:
```

```
  security:
    enabled: false # 关闭管理认证
  context-path: /management # 管理前缀
eureka:
  instance:
    status-page-url-path: ${management.context-path}/info
    health-check-url-path: ${management.context-path}/health
    metadata-map:
      cluster: default # 集群名称
```

- Zuul网关配置

```
zuul:
  ignoredPatterns: /*/management/** # 同上文 management.context-path , 这里之所以不是 /management/** ，
由于网关存在项目前缀，需要往前一级，大家可以具体场景具体配置
```

- Spring Boot Admin配置

```
spring:
  application:
    name: monitor
  boot:
    admin:
      discovery:
        converter:
          management-context-path: ${management.context-path}
      routes:
        endpoints:
env,metrics,dump,jolokia,info,configprops,trace,logfile,refresh,flyway,liquibase,heapdump,loggers,aud
itevents,hystrix.stream,turbine.stream # 要监控的内容
      turbine:
        clusters: default  # 要监控的集群名称
        location: ${spring.application.name}

turbine:
  instanceUrlSuffix: ${management.context-path}/hystrix.stream
  aggregator:
    clusterConfig: default
  appConfig: monitor-example,hystrix-example # 添加需要被监控的应用 Service-Id ，以逗号分隔
  clusterNameExpression: metadata['cluster']

security:
  basic:
    enabled: false

server:
  port: ...

eureka:
  instance:
    metadata-map:
      cluster: default
    status-page-url-path: ${management.context-path}/info
    health-check-url-path: ${management.context-path}/health

  client:
    serviceUrl:
      defaultZone: ...

management:
  security:
```

```
    enabled: false
  context-path: /management # 同上文 management.context-path
```

JDBC批量插入性能问题

在不开启rewriteBatchedStatements=true时，Jdbc会把批量插入当做一行行的单条处理，也就没有达到批量插入的效果。

```
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://127.0.0.1:3306/dew?useUnicode=true&characterEncoding=utf-
8&rewriteBatchedStatements=true
    username: root
    password: 123456
```

下表是对Mybatis和JdbcTemplate，进行1500条数据插入操作的对比实验结果，从表中可以看出，该配置对于JdbcTemplate影响是极大的，而对于Mybatis影响却不大，但开启rewriteBatchedStatements肯定具有速度的提升。

RewriteBatchedStatements	Mybatis(ms)	JdbcTemplate(ms)	Dew(ms)
true	401	88	174
true	427	78	167
true	422	75	176
false	428	1967	2065
false	410	2641	2744
false	369	2299	2398

HTTP请求并发数性能优化

当Hystrix策略为Thread时（默认是Thread），hystrix.threadpool.default.maximumSize为第一个性能瓶颈，默认值为10.

> 需要先设置hystrix.threadpool.default.allowMaximumSizeToDivergeFromCoreSize为true，默认为false

第二个瓶颈为springboot内置的tomcat的最大连接数，参数为server.tomcat.maxThreads，默认值为200

日志中解析message,动态显示property

在启动类的main方法中注册converter，如下PatternLayout.defaultConverterMap.put("dew", TestConverter.class.getName());

自定义Converter继承DynamicConverter<ILoggingEvent>，解析message，获取有效信息并返回解析后得到的字符串。

e.g.

```
public class TestConverter extends DynamicConverter<ILoggingEvent> {

    @Override
    public String convert(ILoggingEvent event) {
        // 这里未做解析，示例代码
        return event.getMessage();
    }
}
```

总结

Spring Cloud以Spring之名，在微服务架构中社区影响力独领风骚，生态之完整性、成熟度、易用性更是可圈可点、一文难尽，同时其架构

的高度可扩展性、模块化设计可以很方便地做针对性地优化，正是因为这些优点我们选择它。上文是我们概要性地总结，其中多次提及的Dew Framework这一Spring Cloud扩展框架我们也会在后续做开源分享。

作者：蒋震宇、葛佳兴、丁忠发 2018年2月19日