Spark

简介

Spark，寓意星星之火，可以燎原，给我一点火花，可以帮你点亮星空。

Spark设计的设计初衷是解决核心系统中的通知类问题，常见业务流程中，经常有对特定目标发送特定信息的通知需求。若系统业务较复杂，发送通知的业务点也会增加。通知需求本身并不复杂，但是当需要通知的业务点较多且分散时，随之带来的管理和维护成本逐渐上升，同时通知的内容往往与业务点紧密关联，通知目标的手段多样。几个维度的叠加，赋予通知相应的复杂度，通知的难度不在于业务复杂，在于管理与维护。

背景

通知与我们的生活密切相关，手机短信和邮件通知是最常见到的方式。手机话费余额不足或是欠费时，会收到运营商发来的短信；还有最常见的就是验证码也是一种通知；不论是银行营业厅和移动营业厅办理一些业务时，都会收到相关的短信提醒；还有我们在一些平台注册的账号，如果提供了手机号或是Email，经常会收到短信或是邮件通知。由此可见，只要存在所谓的“客户”，通知就是系统中不可或缺的功能，通知设计的是否合理，很大程度上会影响系统的友好度。

问题分析

- 分散性

通知常是其他业务的一个延伸，绑定在具体的业务中，不是独立存在的，即分散在各个业务点中，难于统一代码并管理。

- 内容不确定性

因通知需绑定在具体业务中，故通知的内容多与绑定的具体业务息息相关，更甚者，业务归属等因素也会影响通知的内容。

- 时效性

通知总体可以概括为即时和定时通知两种，即时通知常伴随业务发生而发生，具有实时性；定时通知则需要在特定时间场景下触发，通常在跑批时触发。

- 发送手段多样性

随社会的发展，通迅手段也在进步，通知的方式也在发生变化，传统的纸质信件、Email正渐渐被短信、微信等时效性更强的方式替代，当然，一些重要的通知还需要传统的方式进行，短信和微信也有其不足之处。

设计构想

设计背景

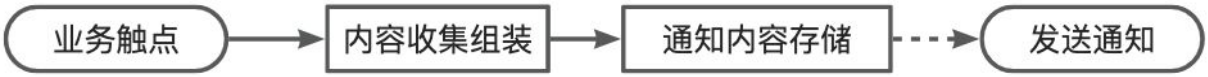
系统设计构思源于保险业务中对Letter的处理。在具体保险业务中，有很多业务场景需要给客户发送送Letter通知或是短信通知，随着通知数量增加，有两个问题摆在眼前：

1. 通知代码分散在各业务代码中，甚至与主流程业务相互缠绕，对代码后期的管理和维护非常不便，且不安全。
2. 冗余重复代码会越来越多，如在处理Letter时，相当部分逻辑是完全一样的（如发送），没有必要每个Letter都再写一遍或是复制一次，当重复代码到一定程度，对后期维护造成严重负担。

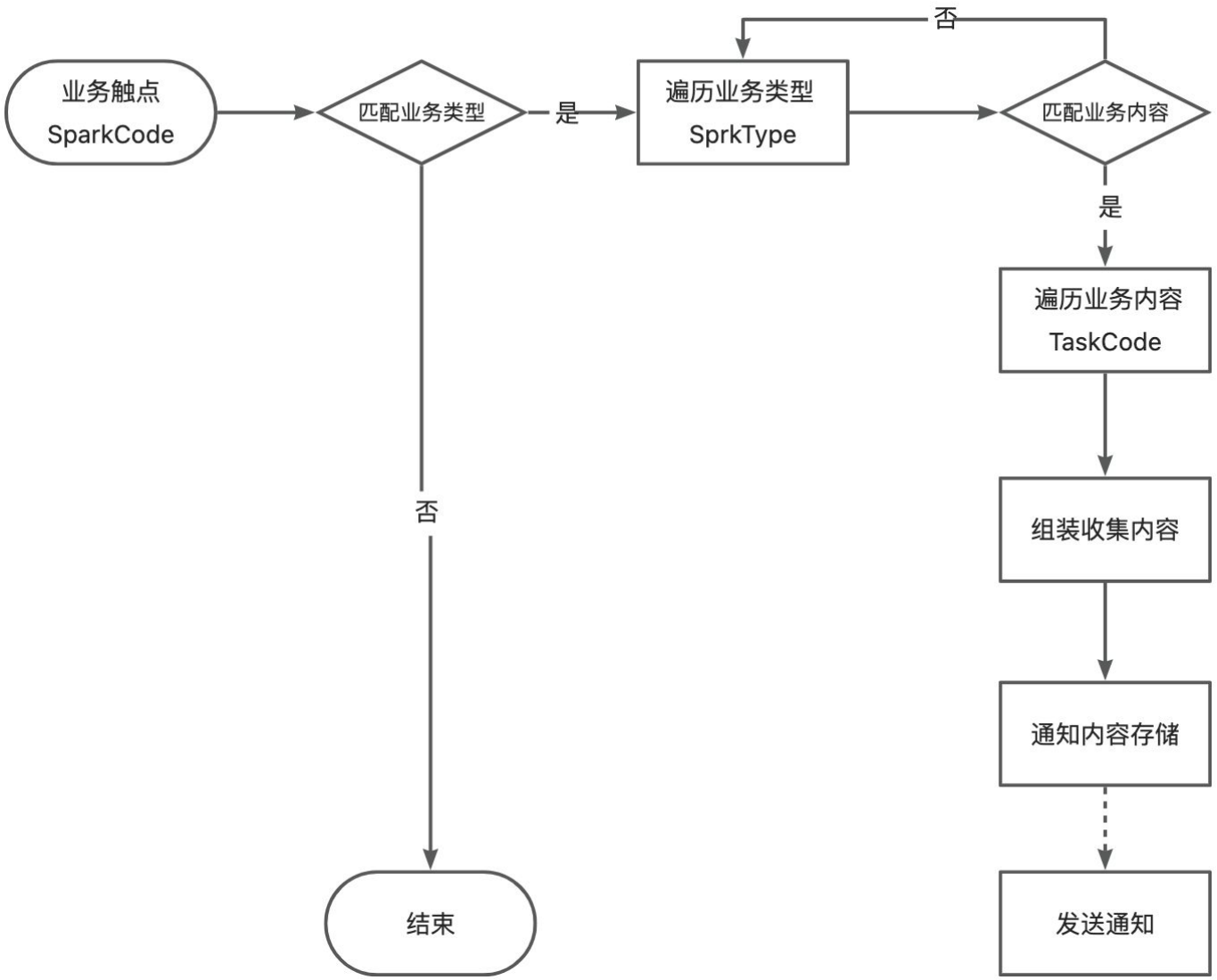
所以需要有一个框架，可以做到把Letter的逻辑从具体业务中剥离出来，并对其中的重复部分整合，便于管理和维护。

设计过程

随着思考的深入和对Letter业务的抽象，对通知这类业务进行了动作分解设计：



再增加对通知类型的扩展和通知内容的扩展，即一种极限假设：在一个业务触点上，可能同时有多种不同类型的通知产生，如Email、短信等，且每种类型通知可能有多条内容，如保险中修改了被保人年龄，可能会同时收到业务变更通知和缴费通知。基于这种假设，我们再对上述的设计做了调整：



设计原则

- 可扩展性

通知具有内容和类型的双重不确定性，所以框架必须具备良好的扩展性和弹性。

- 灵活性

通知灵活多样，把流程和动作按标准分解后，可根据实际需要自由组合、替换。

框架概述

应用版本

JDK ： 21（Jdk 11+）

SpringBoot：3.3.5

Database：Mysql 8.4.7

Liquibase：4.29.0

> 在开发测试过程中使用上述版本，实际过程可适当调整版本。

术语

SparkCode：业务触点唯一标识，可理解为一个事件标识。

SparkType：简称为Type，表示通知类型标识，一个SparkCode可触发多种SparkType，即SparkCode：SaprkType=1:N。

TaskCode：具体任务的唯一标识（简称Task），一个SparkType可触发多个任务，即Type：Task=1:N。

> 这三个条件把Spark框架灵活串联起来，在触点传入SparkCode，并通过该Code关联到对应的Type，再根据Type找到具体的Task，最后执行Task标识的具体的业务逻辑。这三个条件也依次代表了业务需求的三个层级：业务点（SaprkCode）、业务类别（SparkType）和具体业务内容（TaskCode）。

Tenant：这是一个可选条件，旨在针对具有渠道特征的业务，通知可按渠道隔离，每个渠道都可以有自己的通知内容。

Order：这是一个隐性条件，即当上述条件都满足后，还有多个备选时，会根据Order选择优先级高的，一个场景：当一条通知在渠道与平台间冲突时，在渠道业务中应该优先使用渠道的内容。Order便为渠道覆盖平台提供了技术可能。

> 这两个是可选条件，如果不指定Tenant，默认为空，表示忽略该条件，没有指定Order时，会默认给一个Order=100。

> 触点：指在业务流程过程中，会触发通知（Spark任务）的业务点。
>
> Tenant：可指渠道或是机构、分公司等，主要用于区分同一业务在不同Tenant有不同的逻辑。

组件介绍

Launcher：Spark处理器，核心接口，在触点注入并调用transmit触发Spark。

TypeController：业务流程控制器，负责具体业务流程控制，可根据实际业务特征定制，最佳方式是一种业务对应一个流程控制器。

TypeFilter：类型过滤器，设计上针对SparkType级做一些前置过滤，如某些SparkType具有时效性，可前置过滤掉失效部分，可选。

TypeProvider：类型提供器，根据SparkCode找到关联的SparkType，建议关联关系可通过配置表配置，这样可为SparkType提供额外的信息支持。

TaskLoader：任务加载器，与TypeProvider类似，Loader是根据SparkType查找关联的TaskCode，每个TaskCode表示一个具体任务，同样建议关联关系通过配置表配置。

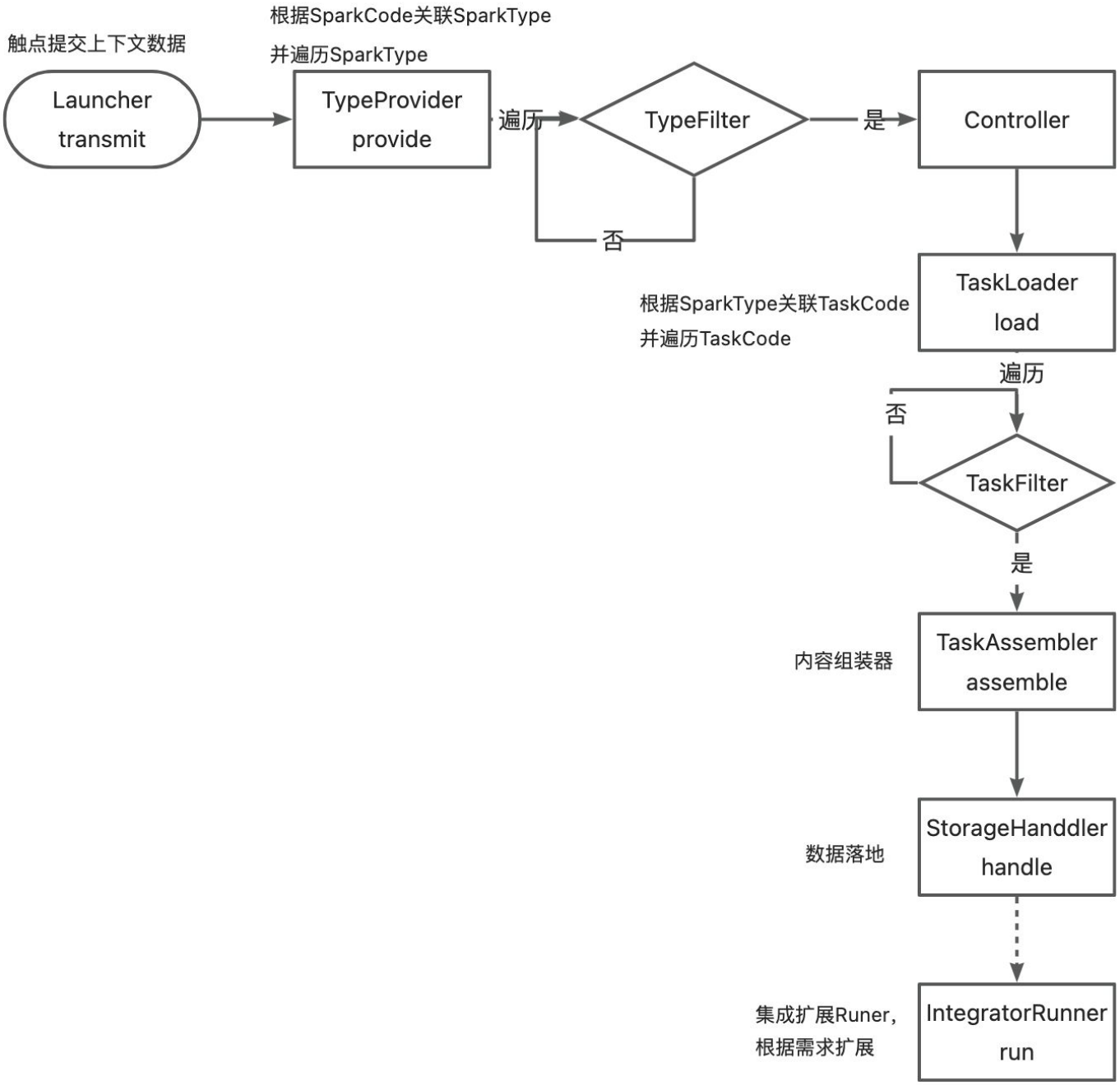
TaskFilter：任务前置过滤器，设计上用来做一些业务上的过滤，可选。没有匹配TaskCode的Filter，默认表示true，继续流程。

TaskAssembler：组装器，Spark的核心组件接口，通常处理具体的业务逻辑，原则上一个TaskCode对应一个TaskAssembler，表示一个具体的业务。

StorageHandler：内容存储组件，把生成的内容落地到数据表中。

IntegratorRunner：一个集成Runner，比较特殊的是一个TaskCode可以有多个Runner，按Order顺序依次执行。Runner设计上用来做一些数据落表后的集成操作，如邮件内容生成后，发送邮件，或是与模板匹配生成内容pdf并上传到文件服务器等，可根据实际业务需求扩展。

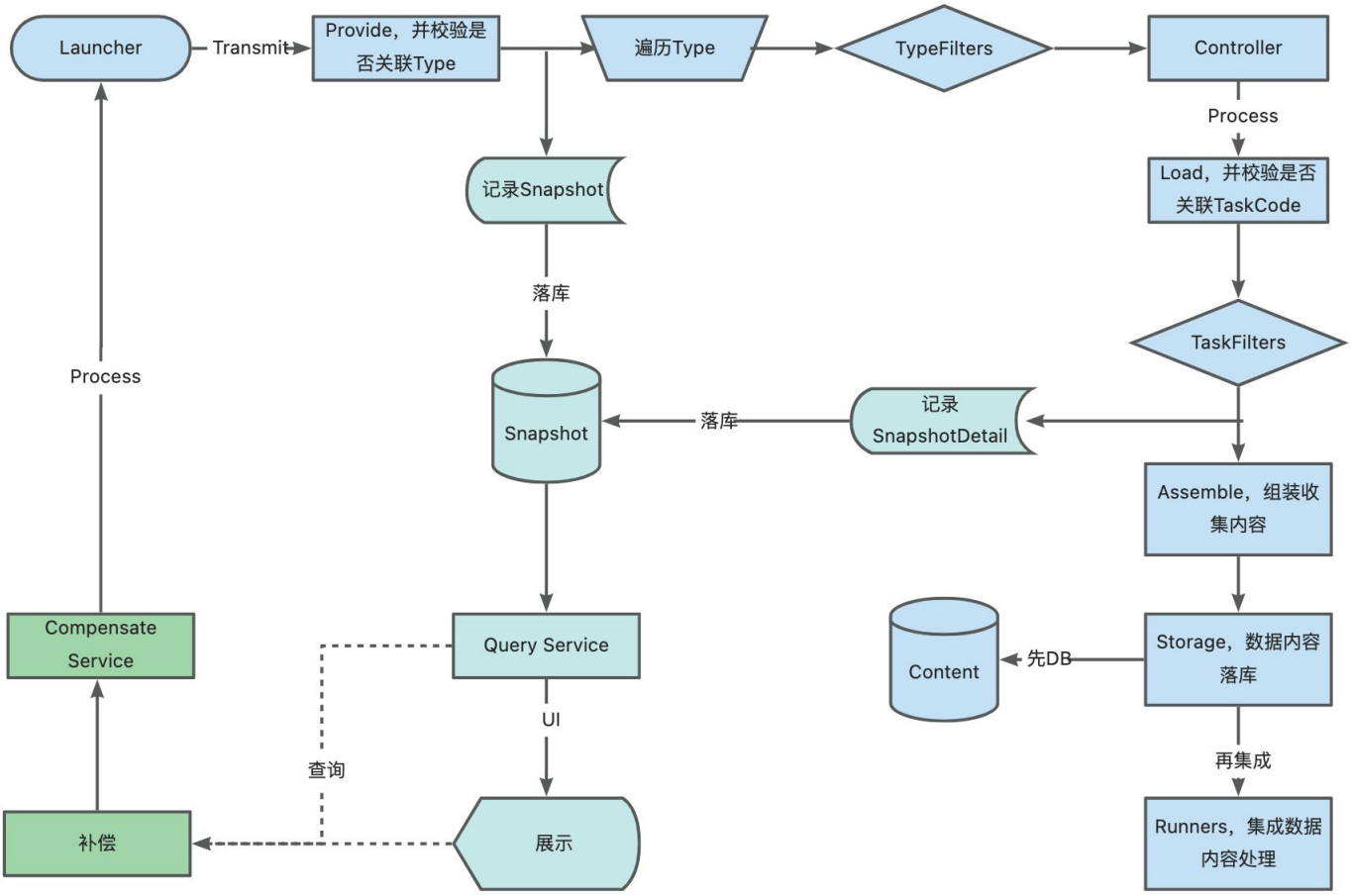
组件关系



由上图可见各组件在框架中所扮演的角色，在业务触点通过Lanucher的transmit提交触点上下文相关数据，进入到Spark框架内，先根据上下文中SparkCode在TypeProvider中找到关联的SparkType，若没有匹配到，则结束该transmit；匹配到Type后，遍历所有匹配Type，依次TypeFilter过滤后，再由指定Controller处理具体的业务逻辑，否则轮询下一个SparkType；Controller是业务流程控制器，可根据业务特征自定义流程逻辑。框架默认的控制器中逻辑如下，在TaskLoader中根据SparkType关联TaskCode，后面依次TaskFilter前置业务逻辑过滤，TaskAssembler具体业务逻辑处理，StorageHandler结果落库，IntegratorRunner后续集成业务处理。至此，构建出一条完整的Saprk组件关系图。

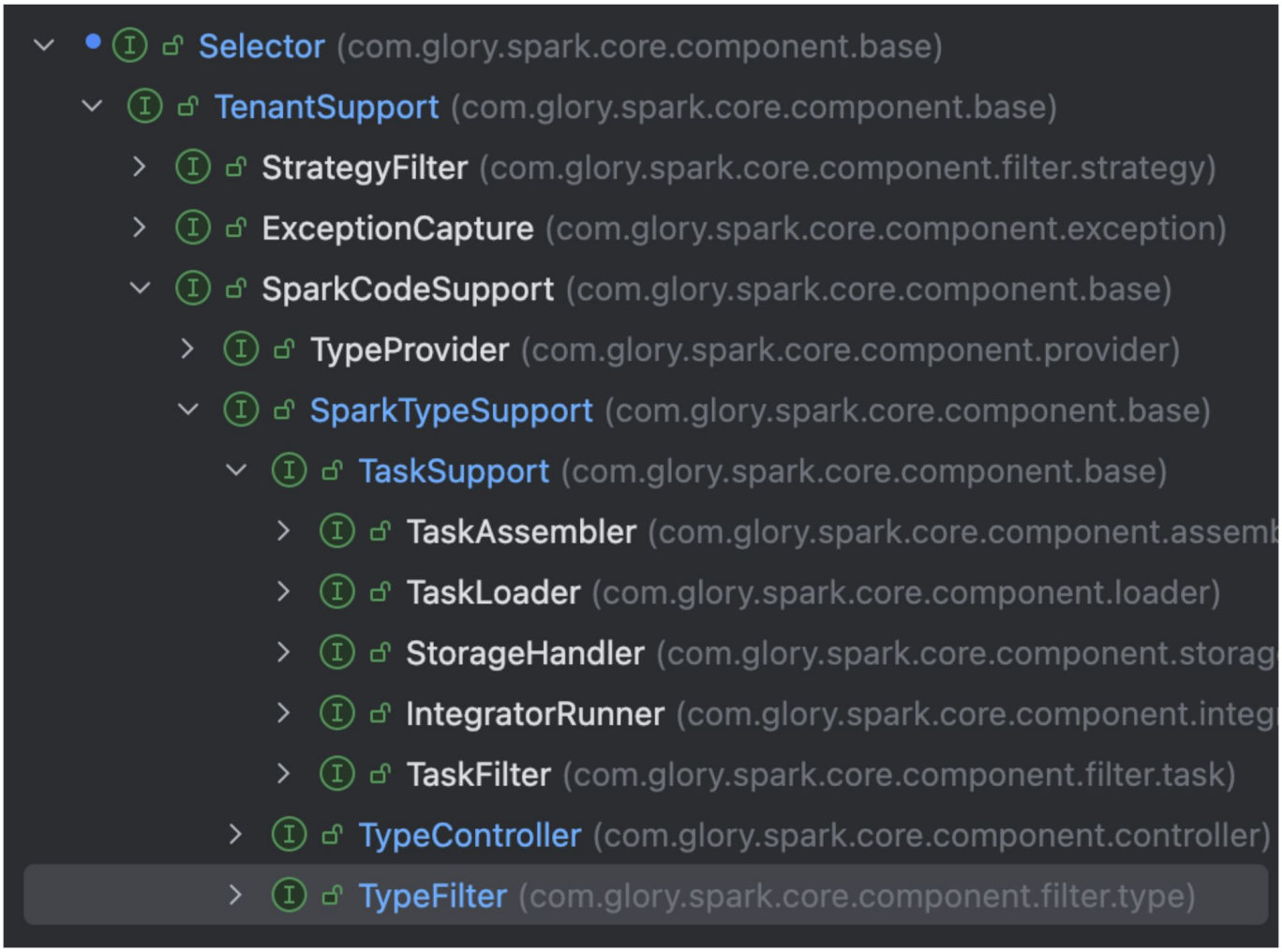
Spark核心

框架从功能上可分为Spark、Snapshot（过程镜像）和Compensate（补偿）三部分，其中Spark为核心业务模块，Snapshot记录Spark流程中的关键过程数据镜像，用于历史追踪和为Compensate提供数据支持；Compensate是一种保证机制，在Spark过程如果出现异常，在处理完异常后，可通过补偿重走Spark流程。



> 上图用颜色来大体划分Spark的模块，蓝色部分表示Spark主流程，中间表示Snapshot模块，左下角绿色表示补偿模块。

组件继承关系



由上图可见，所有的组件都有共同的父接口Selector，4个Support接口分别对应4个选择策略条件。每个Support接口都有一组suportXXX方法，如SparkTypeSupport接口：

所有组件的父接口 Java

```java
public interface Selector {

    boolean match(SparkContext<?> context);

    default boolean match(List<String> conditions, String key){
        if (!CollectionUtils.isEmpty(conditions) && StringUtils.hasLength(key)){
            return conditions.contains(key);
        }
        return true;
    }
}
```

SparkTypeSupport Java

```java
public interface SparkTypeSupport extends SparkCodeSupport{

    default List<String> supportTypes(){//支持哪些SparkType，当返回空时表示忽略该条件
        return List.of();
    }

    @Override
    default boolean match(SparkContext<?> context){//参见Selector
        return match(supportTypes(), context.getType())&&SparkCodeSupport.super.match(context);
    }

}
```

选择策略

- 条件

SparkCode、Type和TaskCode为三个必需核心条件，三个条件可以定位一个唯一具体任务。有一种场景，仅定位到任务层级有时还是不足的，同一个具体任务在不同的Tenant中逻辑不一样，此时需要Tenant进行隔离。另外，还有一种场景，当具体任务在Tenant与平台间冲突时，需要有一个Order来辅助选择。一个目标组件的选择就是通过上述5个条件综合筛选出来的，具体筛选的过程在代理中实现。

> 在选择过程中，有一个原则：如果条件为空时，默认表示忽略该条件，即匹配结果为true，Order的默认值为100.熟练应用该原则可以灵活控制组件的作用范围。

- 代理

为保证框架的灵活性和扩展性，框架对每种组件的调用通过代理类来完成，并在代理过程中依据上述条件完成对目标组件的选择。

代理抽象类 Java

```java
public abstract class AbstractDelegate<S extends Selector>  implements ApplicationContextAware, ApplicationRunner, SparkDelegate<S> {

    protected List<S> targets;
    protected ApplicationContext applicationContext;
    protected Comparator<S> comparator=(o1, o2)->{
        Order od1= o1.getClass().getAnnotation(Order.class);
        Order od2= o2.getClass().getAnnotation(Order.class);
        int s1= null!= od1?od1.value(): MISS_DEFAULT_ORDER;
        int s2= null!= od2?od2.value(): MISS_DEFAULT_ORDER;
        return Comparators.comparable().compare(s1,s2);
};//组件排序

    @SuppressWarnings("unchecked")
    protected Class<S> getActualType(){//实际代理接口
        ParameterizedType type = (ParameterizedType) getClass().getGenericSuperclass();
        return (Class<S>) type.getActualTypeArguments()[0];
    }
    ......
    @Override
    public S delegate(SparkContext<?> context) {//单目标代理
        Assert.hasLength(context.getSparkCode(),"Spark Code is null.");
        S target =getSelector().apply(targets,context).getFirst();
        if (null == target){
            target = getDefaultSelector().get();
        }
        return target;
    }

    @Override
    public List<S> delegates(SparkContext<?> context) {//多目标代码，如Runner和Filter
        return getSelector().apply(targets,context);
    }
}
```

组装器代理 Java

```java
@Component
public class AssemblerDelegate extends AbstractDelegate<TaskAssembler> {

    public <E>SparkResult<E> assemble(SparkContext context ){
        return delegate(context).assemble(context);//单目标选择
    }
}
```

> 代理采用接口代理方式，框架中通过代理类调用具体组件。

默认Spark处理器 Java

```java
public class DefaultSparkLauncher extends AbstractLauncher{

    private ProviderDelegate providerDelegate;//TypeProvider代理
    private ControllerDelegate controllerDelegate;//Controller代理
    private List<SnapshotListener> snapshotListeners;//Snapshot
    private TypeFilterDelegate typeFilterDelegate;//TypeFilter代理
    ......
}
```

- 上下文关键参数

Spark上下文对象 Java

```java
public class SparkContext <T> implements PropertyDesc {

    private T context;//主要上下文对象，原则上，触点传什么，Assembler中接收什么
    private Object refObj;//扩展上下文数据内容，如context的对照版本数据。
    private String sparkCode;//必须条件，标识业务唯一性
    private String taskCode;//任务唯一标识，通常由TaskLoader提供
    private String type;//SparkType，表示业务类型，通常由TypeProvider提供
    private String serialId;//不传时，默认会由UUID生成一个
    private int conditionMode = SparkConstant.PARAMETER_MODE_SPARK;
    //触点时，SparkCode（1）、Type（2）和TaskCode（4）参数组合，通过或运算知道参数组合样式
    private SourceFrom source = SourceFrom.Online;//业务来源，分在线、Batch、补偿
    private RetryStrategy retryStrategy;//补偿策略，只有补偿时有效
    private LocalDateTime processDate;//任务发生时间，通常Batch时必须
    private final Map<String, Object> properties = new HashMap<>(8);//扩展属性
    private String exceptionStrategy ;//异常处理策略
    ...
}
```

> context：触点时，业务需要的上下文数据对象，这里要**注意泛型类型与Assembler中泛型<T>保持一致**，触点传什么数据，Assembler中接收到什么数据。
>
> sparkCode、type和taskCode是框架三个核心条件，通知在触点只需要传SparkCode，再由SparkCode在TypeProvider中关联SparkType，最后在TaskLoader中由Type关联TaskCode，至此三个条件定位到具体任务的Assembler，若Assembler存在Tenant需求，则需进一步定位。
>
> serialId：设计上与系统日志的TraceId关联，便于日后日志跟踪，若没有设置，则默认由UUID生成一个ID。
>
> conditionMode：不需要传，会根据三个条件的设置自动赋值，主要用于补偿时，识别三个核心条件的传递模式，如Transmit时传了SparkCode和type两个条件，则Mode = SparkCode（1） | Type（2） = 3.

自由扩展

框架设计类似于一个二级缓存模式：一级为业务类别，是一个概念抽象；而二级则负责具体的业务逻辑流程。这样的设计的目标是使框架具有强的可扩展性

- 新业务扩展：TypeController是业务级的扩展组件，负责新业务具体业务流程控制，每种业务必须有一个唯一Type名称来区分。

NewTypeController Java

```java
public class NewTypeController implements TypeController {
    ......
    @Override
    public List<String> supportTypes() {
        return List.of("NewType");//指定新类型名，在具体业务组件中，也同样指定该名称
    }
}
```

- 具体业务扩展：默认提供的业务组件包括TaskFilter、TaskAssembler、StorageHandler和IntegratorRunner四种，也可以根据实际需要增加，只需要遵守代理选择策略即可。具体业务组件可通过supportTypes进行业务隔离。

NewTypeAssembler Java

```java
public class NewTypeAssembler implements TaskAssembler<Object,Object> {
    /**
     * @param sparkContext
     * @return
     */
    @Override
    public SparkResult<Object> assemble(SparkContext<Object> sparkContext) {
        // TODO 具体业务逻辑
        return null;
    }

    @Override
    public List<String> supportTypes() {
        return List.of("NewType");//指定新类型名
    }
}
```

Snapshot

Snapshot主要是保存Spark运行时的关键数据镜像，主要目的：

> - 历史追踪：可通过记录查看每个业务、每个Task的执行结果，为追踪问题提供依据。
> - 数据补偿：当某些特殊原因，需要重复执行某些Spark任务，Snapshot提供了数据支持。

记录表

- Snapshot

属性名	类型	描述
SNAPSHOT_ID	LONG	主键ID
SPARK_CODE	STRING	SparkCode，业务唯一标识
SERIAL_ID	STRING	TraceId，日志追踪
APP_NAME	STRING	当前App name
CONTEXT_PATH	STRING	当前app contextPath
TENANT	STRING	Tenant ，租户
PROCESS_DATE	DATETIME	处理日期，常用于Batch
SOURCE	INT	业务来源
STATUS	INT	执行状态，0:init，1:success，-1:fail
MESSAGE	STRING	异常内容
CONDITION_MODE	INT	参数模式，补偿时需要
CONTEXT_DATA	JSON	context对象转换成Json，补偿时需要
CONTEXT_TYPE	STRING	context对象的类型名，补偿时需要
REF_OBJECT_DATA	JSON	refObj的Json形式，补偿时需要
REF_TYPE	STRING	refObj对象的类型名，补偿时需要

> 记录业务级的基本信息，主要来源于SparkContext中内容，如context和refObj会序列化成Json字符串，并记录其类型信息，便于补偿时对象还原。
>
> AppName：在微服务中，可标识数据来源，同时在补偿回调中可表示target App。
>
> Tenant：出于数据安全考虑，用于支持多租户情况。

- SnapshotDetail

属性名	类型	描述
DETAIL_ID	LONG	Detail Id
SNAPSHOT_ID	LONG	主表SnapshotId

RELEATED_ID	LONG	补偿时，关联DetailId，指向被补偿记录
SPARK_CODE	STRING	SparkCode，业务唯一标识
TYPE	STRING	SparkType，业务类型
TASK_CODE	STRING	TaskCode，任务唯一标识
TENANT	STRING	Tenant，租户
TRACE_ID	STRING	TraceId，记录日志Id
SOURCE	INT	来源，online，batch，compensate
STATUS	INT	状态，0–初始；1–成功；–1–失败；2–补偿
MESSAGE	STRING	异常信息
TARGET_ID	LONG	结果产生后存表的主键Id，如ContentId
STRATEGY	INT	补偿策略，全补偿、失败补偿、重生成三种
RETRY_INDEX	INT	补偿次数
LAST_RETRY_TIME	DATETIME	补偿时间

> Detail表记录了任务的详细信息
>
> ReleatedId：补偿时，主表不变，但是Detail表会新生成一条记录，该字段指向被补偿记录，被补偿记录的状态更新为2，表示已补偿。
>
> TargetId：任务结果落库后表的主键ID。
>
> Strategy：补偿策略，每次补偿都会新生成一条记录。
>
> - 全补偿：业务级，以SparkCode为依据进行补偿，不关心状态（排除Status=2）。
> - 失败补偿：也是业务级，但是只对SparkCode下失败的Task进行补偿。
> - 重生成：任务级，对某一Detail记录进行补偿，忽略状态（排除Status=2）。

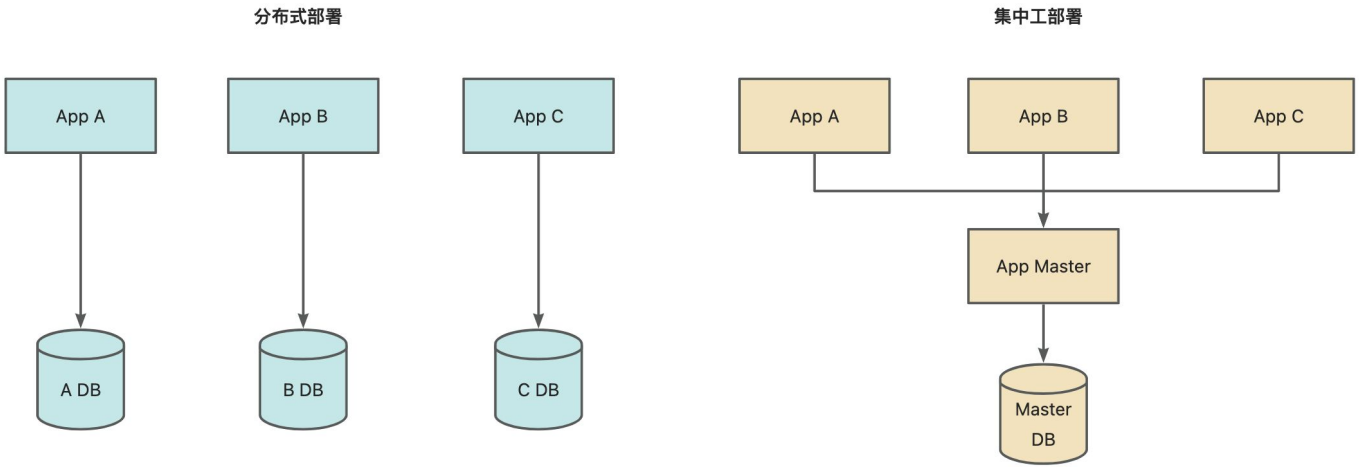
记录Snapshot

Snapshot严格上不是必需功能，故采用Listener方式记录，减少对主业务的干扰。框架中提供默认Snapshot记录Listener是DefaultSnapshotListener，可参见相关的代码，也可自定义记录方式。

部署模式

Spark在微服务中以分布式和集中式两种方式部署，这里的部署是指Database的部署。Spark可以被任意一个App依赖使用，每个App都可能有自己的Database，Snapshot的表怎么部署，可根据实际需求决定。Database部署模式不同，对Snapshot数据存储和查询有很很大影响，也影响补偿的实现方式。

> - 分布式：每个App的数据存在自己的DB 中。优点是DB压力小，缺点是不便于查询和管理。
> - 集中式：指定一个App为主服务，所有产生的Snapshot记录统一记录到一个DB中。优点是便于查询和管理，缺点是数据量大时，对DB压力较大。
>
> 后面介绍补偿时，主要是基于集中式进行介绍，若使用分布式，建议借助ES等工具，便于查询。
>
> 部署模式可以通过两个配置来切换
>
> spark.snapshot.deploy.master 是否为集中式部署，默认true，集中式。
>
> spark.snapshot.master-app-name。集中部署时必需，指定主节点的appName，默认为空



Compensate 补偿

补偿是保证业务结果正确的一种保障设计， 主要是对系统异常情况下产生后果的补救措施。如Spark过程遇到异常没有生成通知内容、或因数据原因生的通知内容有问题，或因某些特殊情况需要废弃旧的内容后生成新的通知等需求存在，但是面临一个问题，Spark通常是镶嵌在具体业务过程中的，只是业务中的一个点，很多时候不可能因为一个Spark而使整个业务流程倒退或是重做一遍业务，这样成本和风险太昂贵。此时我们可以从Snapshot中提取当时的上下文数据，重执行一遍Spark部分的流程即可，进而把风险和成本控制大最小。

- 全补偿：业务点级，即把SparkCode相关的Task都重新执行一次。
- 失败补偿：也是业务点级，不同时全补偿，此处只对失败的Task重新执行一次。
- 重生成：具体Task级，针对具体的Task进行补偿。

> 补偿后，会把被补偿的记录的Status标记为2，表示已补偿，过滤数据时通常会过滤掉状态为2的数据，同时新记录数据会通过字段ReleatedId关联该记录，便于历史追踪。

关键配置

- Spark

> spark.executor.max-pool-size=50，spark.executor.core-pool-size=20，spark.executor.queue-capacity=20，Spark异步线程池配置。
>
> spark.app-context.filter.enabled，是否启动Http请求Spark线程上下文初始化，默认开启，可通过SparkUserContextInitializer接口来初始请求上下文，如租户、User信息等放入AppContext线程变量中。
>
> spark.launcher.disabled（true | false），Spark功能开关，当为true时，关闭Spark功能，默认false。
>
> spark.filter.compensate.validate ，在EffectiveDate校验时，是否忽略Source是补偿的操作，默认忽略，即在补偿时，忽略对Effective date的校验。

- Snapshot

> spark.snapshot.disabled（true | false） 是否开启Snapshot记录，默认开启。镜像数据是补偿操作的数据基础，没有镜像数据则无法补偿。
>
> spark.snapshot.master-app-name 集中式部署时主节点name
>
> spark.snapshot.deploy.master（true | false ）部署模式，默认是集中式
>
> **API 接口Uri配置**
>
> spark.http.snapshot.save:/snapshot/save
>
> spark.http.snapshot.find-by-id:/snapshot/{snapshotId}/{includeDetail}
>
> 根据主键查询Snapshot，includeDetail结果是否包含Detail节点信息
>
> spark.http.snapshot.query:/snapshot/query
>
> spark.http.snapshot.detail.save:/snapshot/detail/save
>
> spark.http.snapshot.detail.find-by-id:/snapshot/detail/{detailId}
>
> spark.http.snapshot.detail.find-snapshot-id:/snapshot/detail/all/{snapshotId}
>
> 根据snapshotId查询关联的所有Detail记录
>
> spark.http.snapshot.detail.query:/snapshot/detail/query
>
> spark.http.snapshot.detail.queryItems:/snapshot/item/query
>
> spark.http.snapshot.detail.task-status:/snapshot/detail/{taskCode}/{status}

- Compensate

> spark.http.compensate.transmit:/compensate/transmit
>
> 补偿内部API，App之间调用，主要是主节点向补偿目标节点调用。
>
> spark.http.compensate.process:/compensate/item/process
>
> 补偿API，方节点对外提供的Api。

Content模块

Spark-Content是Spark提供的结果存储模块，即Assembler中产生的结果会通过StorageHandler存到Content模块的DB中。首先该模块不是必须的，只是框架提供的一种可选项，也是一种参考。其次，这部分是可以根据需求定制的，定制时必须提供一个API用来存储结果内容。

Content表

> T_SPARK_CONTENT
>
> Spark结果内容表，通过StorageHandler把结果落库，也便于结果追踪。

属性	类型	描述
LIST_ID	LONG	主键
SPARK_CODE	STRING	SparkCode，业务点标识
TYPE	STRING	SparkType，业务类型
TASK_CODE	STRING	TaskCode，任务唯一标识
CATEGORY	STRING	内容分类，预定义
NAME	STRING	名称
DESCRIPTION	STRING	描述
MOUDLE	INT	模块，预定义
SERIAL_ID	STRING	默认TraceId
TENANT	STRING	租户，数据隔离
TEMPLATE_CODE	STRING	模板Code，非必需，预定义
EXPIRE_DATE	DATETIME	失效日志，预定义
SOURCE	INT	来源

STATUS	INT	状态
CONTENT	JSON	内容Json
RECEIVERS	JSON	收件人List。Json
TRACK_INDEX	INT	Track 次数

Content 轨迹

> T_SPARK_CONTENT_TRACK
>
> 用来记录Content表操作记录，如与模板匹配生成文件或内容次数，发送次数等信息

属性	类型	描述
TRACK_ID	LONG	轨迹表主键
CONTENT_ID	LONG	Content表主键
SERIAL_ID	STRING	日志TraceId
TENANT	STRING	租户，数据隔离
SOURCE	INT	来源
STATUS	INT	状态
CATEGORY	STRING	类别
FILE	STRING	最终生成的文件存储路径
CONTENT	JSON	最终生成的内容
TRACK_DATE	DATETIME	操作时间
INDEX	INT	操作次数

实际场景演示

Spark-Content集成

- pom依赖

Pom XML

```xml
<dependencies>
  <dependency>
    <groupId>com.glory.spark</groupId>
    <artifactId>spark-core</artifactId>
  </dependency>
  <dependency>
    <groupId>com.glory.spark</groupId>
    <artifactId>spark-content</artifactId>
  </dependency>
</dependencies>
```

- TaskAssembler

TaskAssembler示例 Java

```java
@Component
public class TestPersonTaskAssembler implements TaskAssembler<Person, SparkContentBo> {

    @Override
    public SparkResult<SparkContentBo> assemble(SparkContext<Person> sparkContext) {
        SparkContentBo contentBo = new SparkContentBo();
        contentBo.setName("notice case");
        contentBo.setCategory(0);
        contentBo.setDescription("this is a notice case");
        contentBo.setModule(1);
        contentBo.setReceivers(List.of("notice user"));
        contentBo.setSerialId(sparkContext.getSerialId());
        contentBo.setSource(0);
        contentBo.setSparkCode(sparkContext.getSparkCode());
        contentBo.setTenant(AppContext.getTenantCode());
        contentBo.setStatus(0);
        contentBo.setTaskCode(sparkContext.getTaskCode());
        contentBo.setType(sparkContext.getTypeDesc().getType());
        Person person = sparkContext.getContext();
        contentBo.addContent("name", person.getName());
        contentBo.addContent("age", person.getAge());
        contentBo.addContent("mobile", person.getMobile());
        contentBo.addContent("nationality", person.getNationality());
        SparkResult<SparkContentBo> result = SparkResult.Success(sparkContext);
        result.addElement(contentBo);
        return result;
    }
}
```

- StorageHandler

StorageHandler示例 Java

```java
@Component
public class TestStorageHandler implements StorageHandler<SparkContentBo>
{

    @Autowired
    private SparkContentRouteService routeService;

    @Override
    public void stored(SparkResult<SparkContentBo> result) {
        Optional.ofNullable(result.getElements()).ifPresent(elements -> {
            elements.forEach(e -> {
                SparkContentBo bo = routeService.saveAndUpdate(e);//Save C
ontent
                BeanUtils.copyProperties(bo,e);
            });
        });
    }
}
```

扩展组件

在实际开发过程中，因组件在业务流程中角色不同，并不是所有组件与Task或是Type一一对映，部分组件大多时候适配于一类业务或是一组业务，此时按Task进行配置有些冗余和浪费。

- 按业务类型

以StorageHandler为例，需要一个对业务类型这“SMSType”进行定制处理，只需要2步即可：

SMSType Java

```java
@Order(20) //步骤2:设定一个Order，value小于其他通用的Handler组件Order
@Component
public class SMSStorageHandler implements StorageHandler<TypeBo> {

    @Override
    public void stored(SparkResult<TypeBo> result) {
       ....
    }

    @Override
    public List<String> supportTypes() {
        return List.of("SMSType");// 步骤1:指定支持的Type
    }
}
```

> 上述实例中，只指定了核心三条件中的Type一项，在匹配时，会忽略其他两个条件的过滤，只按Type进行过滤查找组件，匹配的结果可能有多个时，这时需要保证Order的value小于其他组件即可。如果熟悉了代理选择策略，可以自由组合几个条件。

- 一组业务

若是组件只针对特定的几个Task或是SparkCode业务，则可用下面的方式：

适配SparkCode Java

```java
@Order(20)
@Component
public class NewStorageHandler implements StorageHandler<TypeBo> {

    @Override
    public List<String> supportSparkCodes() {
        return List.of("testSpark","testSpark2");//仅限于testSpark和testSpark2
    }
}
```

> 上述代码，仅限SparkCode为testSpark、testSpark2时有效。

适配Task Java

```java
@Order(20)
@Component
public class NewStorageHandler implements StorageHandler<TypeBo> {

    @Override
    public List<String> supportTasks() {
        return List.of("Task1","Task2","Task3");//仅限于Task1，Task2，Task3
    }
}
```

> 上述代码，仅限于TaskCode为Task1，Task2，Task3时有效。

多条件组合 Java

```java
@Order(20)
@Component
public class NewStorageHandler implements StorageHandler<TypeBo> {

    @Override
    public List<String> supportTypes() {
        return List.of("SMSType");// 指定支持的Type
    }
    @Override
    public List<String> supportTasks() {
        return List.of("Task1","Task2","Task3");//仅限于Task1，Task2，Task3
    }
}
```

> 上述组件，仅限于SparkType=SMSType，且TaskCode 为Task1，Task2，Task3的业务有效。注意，当条件组合使用时，不同类型条件间组合的结果按“&”计算，相同类型条件内则按“｜”计算，Order则是唯一结果的最后保障，匹配后仍有多结果时，按优先级取最高的一个。**注意一个原则：通用组件的Order设置优先级要尽量小一些，且最好小于定制类组件（优先级按Order由小到大排序）。**

扩展新业务

扩展一个新型业务很简单，首先确定业务名称，即SparkType 名称，其次确定新业务的逻辑流程，如果有可以复用Controller，则忽略该步骤，否则需要定制一个新的Controller，可参照标准的DefaultTypeController类，最后扩展具体逻辑流程中使用到的业务组件。

- 确定Type名

Type是一类业务的唯一标识，在框架中，也是业务隔离的标识，所以在扩展新业务时，首先要把Type名确定。

- 梳理业务流程

确定新业务流程并对流程进行流程分，确定需要哪几个流程环节，选用适当业务组件或是定制新的组件。

- 扩展业务组件

新业务Assembler的父接口 Java

```java
public interface NewTypeTaskAssembler <T,E>extends TaskAssembler<T,E> {
    @Override
    default List<String> supportTypes() {
        return List.of("NewType");
    }
}
```

新业务实现类 Java

```java
public class NewTypeAssembler implements NewTypeTaskAssembler<Object,Object
> {
    @Override
    public SparkResult<Object> assemble(SparkContext<Object> sparkContext)
{
        // TODO 具体业务逻辑
        return null;
    }
}

```

> 框架提供的业务组件包括TaskFilter、TaskAssembler、StorageHandler和IntegratorRunner四种，其中TaskAssembler与业务关系最密切，其他三种多是通用的一组或是一类业务有关。
>
> 在扩展时，建议为每类业务的每种组件建一个类似NewTypeTaskAssembler的父接口，业务隔离的同时还可以防止不同业务间Task命名冲突。

Spark使用

Spark使用非常简单，有两种方式：

- 显示调用

Spark使用 Java

```java
@Component
public class TestServiceImpl implements Testervice {

    @Autowired
    private SparkLauncher launcher;//步骤1:流入SparkLauncher

    @Override
    public SparkObject transmit(SparkObject obj) {
        SparkContext context = SparkContext.create(obj.getCode());//指定SparkCode
        context.setContext(obj.getPerson());//传上下文数据
        context.addProperty("work",obj.getWork());//添加附加数据
        launcher.transmit(context);//步骤2:调用Spark，可参见SparkLauncher接口方法
        return obj;
    }
}
```

> 首先在Service中注入SparkLauncher，然后在业务流程中找到触点，调用SparkLauncher.transmit。

- 注解方式

Spark注解说明 Java

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Spark {
    String sparkCode();//指定SparkCode，必需
    String type() default "";//SparkType，一般不指定，通过SparkCode关联
    String taskCode() default "";//TaskCode，一般不指定，通过SparkType和SparkCode关联
    String condition() default "";//Spark发生的条件，property属性控制
    Class<? extends SparkCondition> conditionClass() default SparkCondition.Impl.class;
    //SparkCondition子类来条件判断Spark发生。
    boolean ifMissing() default true;//条件缺失时，默认条件结果
    boolean sync() default true;//同步发生还是异步发生，默认同步
    int argumentIndex() default 0;//SparkContext中context取值于方法第Index个参数。
    Class<? extends ContextWrapper> contextWrapper() default ContextWrapper.Impl.class;
    //SparkContext中context取值实现类
    SparkOccasion occasion() default SparkOccasion.After;//Spark发生时机，与AOP相似
    boolean catchException() default false;//是否Catch异常，默认不Catch
 }
```

Spark注解使用 Java

```java
@Component
public class TestAnnotationServiceImpl implements TestAnnotationService {

    @Override
    @Spark(sparkCode ="test1127")
    public Person savePerson(Person person) {
        person.setMobile("110110110");
        return person;
    }
}

```

> 使用注解可以更简化Spark的使用，同时减少对业务代码在的侵入。上述示例代码是触发一个SparkCode=“test1127”，SparkContext.context=person的Spark事件，发生时机是在方法结束后。

> *@Spark(sparkCode ="test1127",type = "test",taskCode = "test1127",conditionClass = TestSparkCondition.class,occasion = SparkOccasion.Before,catchException = true)*

> 上面的例子同时指定了SparkCode、SparkType和TaskCode，发生在方法调用前，如期间有异常，捕获异常，发生的条件需要看*TestSparkCondition*。

TestSparkCondition Java

```java
public class TestSparkCondition implements SparkCondition {

    @Override
    public boolean match(@Nonnull Method method, Object[] argus) {
        return method.getName().equals("savePerson");
    }
}
```