业务背景：

系统利用RocketMQ作为异步解藕，自产自消

策略模式

```
/**
 * 定义一个异步处理的接口类
 * 需要走异步的逻辑实现这个接口
 */
public interface AsyncComponent {

    /**
     * 根据processType区分
     */
    Integer processType();

    void process(Integer asyncType, String jsonStr);

}
```

```
@Slf4j
@Component
/**
 * 接受到mq消息之后，根据消息体的type类型，从map中捞取对应的AsyncComponent处理类
 */
public class AsyncDispatchHandle {

    @Resource
    private List<AsyncComponent> asyncComponentList;

    private final Map<Integer, AsyncComponent> asyncTypeAndComponentMap = new
ConcurrentHashMap<>();

    @PostConstruct
    private void init(){
        asyncComponentList.forEach(cmp -> {
            asyncTypeAndComponentMap.put(cmp.processType(),cmp);
        });
    }


    public void handle(Integer asyncType,String json){
        AsyncComponent amp = asyncTypeAndComponentMap.get(asyncType);
```

```
        if(amp == null){
            log.error("[handle] get amp null,asyncType={},json={}",asyncType,json);
            return;
        }
        amp.process(asyncType,json);
    }
```

模板模式

因为AsyncComponent的process方法，有一些统一的处理逻辑，像解析json，异常捕获，重试逻辑

定义了一个抽象类, 然后所有的AsyncComponentImpl继承该类，实现doProcess()方法即可

```
@Slf4j
public abstract class AbstractAsyncComponent<T> implements AsyncComponent {

    @Override
    public void process(Integer asyncType, String jsonStr) {
        ThreadLocalCache.remove();
        try {
            if (StringUtils.isBlank(jsonStr)) {
                return;
            }

            T t = JSON.parseObject(jsonStr, getJsonClass());
            if (t == null) {
                return;
            }

            doProcess(t);
        } catch (BaseRuntimeException bre) {
            log.warn("[process] err:", bre);
            throw new BaseRuntimeException(bre);
        } catch (Exception e) {
            log.error("[process] err:", e);
        } finally {
            ThreadLocalCache.remove();
        }
    }

    public abstract Class<T> getJsonClass();

    public abstract void doProcess(T t);

}
```