Kafka异常框架事例

目标:提高编码的效率和可维护性

## Java异常架构体系



Error是无法处理的错误,比如OutOfMemoryError,这样的异常发生时,jvm一般会选择线程终止.

Exception是程序本身可以处理的异常，这种异常分两大类运行时异常和非运行时异常。 程序中应当尽可能去处理这些异常。

运行时异常都是RuntimeException类及其子类异常，如NullPointerException、IndexOutOfBoundsException等，   
    这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，   
    程序应该从逻辑角度尽可能避免这类异常的发生。   
  
    非运行时异常是RuntimeException以外的异常，类型上都属于Exception类及其子类。   
    从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。   
    如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。

## 阿里巴巴 JAVA 开发手册

### 异常处理

1. 【强制】不要捕获 Java 类库中定义的继承自 RuntimeException 的运行时异常类，如：  
IndexOutOfBoundsException / NullPointerException，这类异常由程序员预检查来规避，保证程序健壮性。   
正例：if(obj != null) {...}   
反例：try { obj.method() } catch(NullPointerException e){…}   
2. 【强制】异常不要用来做流程控制，条件控制，因为异常的处理效率比条件分支低。   
3. 【强制】对大段代码进行 try-catch，这是不负责任的表现。catch 时请分清稳定代码和非稳 定代码，稳定代码指的是无论如何不会出错的代码。对于非稳定代码的 catch 尽可能进行区分异常类型，再做对应的异常处理。   
4. 【强制】捕获异常是为了处理它，不要捕获了却什么都不处理而抛弃之，如果不想处理它，请将该异常抛给它的调用者。最外层的业务使用者，必须处理异常，将其转化为用户可以理解的内容。   
5. 【强制】有 try 块放到了事务代码中，catch 异常后，如果需要回滚事务，一定要注意手动回滚事务。   
6. 【强制】finally 块必须对资源对象、流对象进行关闭，有异常也要做 try-catch。 说明：如果 JDK7，可以使用 try-with-resources 方法。   
7. 【强制】不能在 finally 块中使用 return，finally 块中的 return 返回后方法结束执行，不会再执行 try 块中的 return 语句。   
8. 【强制】捕获异常与抛异常，必须是完全匹配，或者捕获异常是抛异常的父类。说明：如果预期抛的是绣球，实际接到的是铅球，就会产生意外情况。   
9. 【推荐】方法的返回值可以为 null，不强制返回空集合，或者空对象等，必须添加注释充分说明什么情况下会返回 null 值。调用方需要进行 null 判断防止NPE问题。   
说明：本规约明确防止NPE是调用者的责任。即使被调用方法返回空集合或者空对象，对调用者来说，也并非高枕无忧，必须考虑到远程调用失败，运行时异常等场景返回 null 的情况。   
10.【推荐】防止NPE，是程序员的基本修养，注意 NPE产生的场景：   
 1） 返回类型为包装数据类型，有可能是 null，返回int值时注意判空。   
    反例：public int f(){ return Integer 对象}，如果为 null，自动解箱抛 NPE。   
 2） 数据库的查询结果可能为 null。   
 3） 集合里的元素即使 isNotEmpty，取出的数据元素也可能为 null。   
 4） 远程调用返回对象，一律要求进行 NPE 判断。   
 5） 对于Session中获取的数据，建议NPE检查，避免空指针。   
 6） 级联调用 obj.getA().getB().getC()；一连串调用，易产生 NPE。   
11.【推荐】在代码中使用“抛异常”还是“返回错误码”，对于公司外的 http/api 开放接口必须使用“错误码”；而应用内部推荐异常抛出；跨应用间 RPC 调用优先考虑使用 Result 方式，封装 isSuccess、“错误码”、“错误简短信息”。   
说明：关于 RPC 方法返回方式使用 Result 方式的理由：   
 1）使用抛异常返回方式，调用方如果没有捕获到就会产生运行时错误。   
 2）如果不加栈信息，只是new自定义异常，加入自己的理解的error message，对于调用端解决问题的帮助不会太多。如果加了栈信息，在频繁调用出错的情况下，数据序列化和传输的性能损耗也是问题。   
12.【推荐】定义时区分unchecked / checked异常，避免直接使用RuntimeException抛出，更不允许抛出 Exception 或者 Throwable，应使用有业务含义的自定义异常。推荐业界已定义过的自定义异常，如：DaoException / ServiceException 等。   
13.【参考】避免出现重复的代码（Don’t Repeat Yourself），即 DRY 原则。   
说明：随意复制和粘贴代码，必然会导致代码的重复，在以后需要修改时，需要修改所有的副本，容易遗漏。必要时抽取共性方法，或者抽象公共类，甚至是共用模块。   
正例：一个类中有多个 public 方法，都需要进行数行相同的参数校验操作，这个时候请抽取：   
private boolean checkParam(DTO dto){...}

## 异常框架一: java-error-handler

<https://github.com/Workable/java-error-handler>

#### 使用条件:

<dependency>

<groupId>com.workable</groupId>

<artifactId>error-handler</artifactId>

<version>1.1.0</version>

<type>pom</type>

</dependency>

* Java 8 or higher

#### API

##### Initialize

* defaultErrorHandler()  Get the default ErrorHandler.
* create() Create a new ErrorHandler that is linked to the default one.
* createIsolated()  Create a new empty ErrorHandler that is not linked to the default one.

##### Configure

* on(Matcher, Action) Register an *Action* to be executed if *Matcher* matches the error.
* on(Class<? extends Exception>, Action) Register an *Action* to be executed if error is an instance of Exception.
* on(T, Action) Register an *Action* to be executed if error is bound to T, through bind() or bindClass().
* otherwise(Action) Register an *Action* to be executed only if no other *Action* gets executed.
* always(Action) Register an *Action* to be executed always and after all other actions. Works like a finally clause.
* skipFollowing() Skip the execution of any subsequent *Actions* except those registered via always().
* skipAlways() Skip all *Actions* registered via always().
* skipDefaults() Skip any default actions. Meaning any actions registered on the defaultErrorHandler instance.
* bind(T, MatcherFactory<T>) Bind instances of *T* to match errors through a matcher provided by *MatcherFactory*.
* bindClass(Class<T>, MatcherFactory<T>) Bind class *T* to match errors through a matcher provided by *MatcherFactory*.
* clear() Clear all registered *Actions*.

##### Execute

* handle(Throwable) Handle the given error.

该异常框架的好处,可以定义一个统一的实例,在该实例中定义异常种类和信息.后续使用,可以把异常处理使用相同的代码进行异常处理.

## 异常框架二: faux-pas

<https://github.com/zalando/faux-pas>

### 使用条件

<**dependency**>  
 <**groupId**>org.zalando</**groupId**>  
 <**artifactId**>faux-pas</**artifactId**>  
 <**version**>0.3.1</**version**>  
</**dependency**>

* Java 8 or higher
* Lombok (no runtime dependency)

## 异常框架三: Guava

|  |  |
| --- | --- |
| 参数形式 | 解释 |
| public static RuntimeException propagate(Throwable throwable) | 如果参数throwable是 RuntimeException 或 Error 则原样propagate，否则将它包入 RuntimeException 中抛出。保证抛出。返回值是 RuntimeException 类型.( 把受检查的异常转换为运行时异常) |
| [void propagateIfInstanceOf(Throwable, Class<X extends Exception>) throws X](http://goo.gl/brIfv) | 仅当参数throwable是 X 类型时，原样propagate。 |
| [void propagateIfPossible(Throwable)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/base/Throwables.html#propagateIfPossible(java.lang.Throwable)) | 仅当参数throwable是 RuntimeException 或 Error 类型时，原样propagate。 |
| [void propagateIfPossible(Throwable, Class<X extends Throwable>) throws X](http://goo.gl/pgDJC) | 仅当参数 throwable 是 RuntimeException 或 Error 或 X 类型时，原样propagate。 |
| [Throwable getRootCause(Throwable)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/base/Throwables.html#getRootCause(java.lang.Throwable)) | 返回最底层的异常 |
| [List<Throwable> getCausalChain(Throwable)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/base/Throwables.html#getCausalChain(java.lang.Throwable)) | 以list的方式得到throwable的异常链 |
| [String getStackTraceAsString(Throwable)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/base/Throwables.html#getStackTraceAsString(java.lang.Throwable)) | 返回printStackTrace的结果string |

可以简单地多重捕获和重新抛出多个异常