**Java编码规范及最佳实践\_V2.0**

修订记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 修订版本 | 修改者 | 修改描述 |
| 2024-5-8 | V1.0 | 原作者：郭京满 | 将已有编码规范导入飞书文档，方便后续编辑和修订 |
| 2024-5-8 | V1.9 | 李进 | 增加代码可读性、健壮性的等多个规范  移除了没用的可读性规范 |
| 2024-9-9 | V2.0 | 李进 | 根据评审意见进行补充修订 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |
| --- |
| [**1. 规范分类**](#heading_0)  [**2. 代码健壮性规范**](#heading_1)  [**2.1 入参必须先校验再使用**](#heading_2)  [**2.2 调用公共方法后立即校验返回码/返回值**](#heading_3)  [**2.3 公共方法非必要不返回NULL**](#heading_4)  [**2.4 数据库查询不返回NULL**](#heading_5)  [**2.5 IO对象使用时避免内存泄漏**](#heading_6)  [**2.6 遍历数组/集合类时避免越界或并发修改异常**](#heading_7)  [**2.7 避免出现空指针异常**](#heading_8)  [**2.8 switch-case/if-else必须有兜底处理**](#heading_9)  [**2.9 避免使用浮点类型导致计算精度问题**](#heading_10)  [**2.10 避免过度使用包装类型**](#heading_11)  [**2.11 避免修改上层入参和下层返回值**](#heading_12)  [**2.12 其他**](#heading_13)  [**3. 代码高性能规范**](#heading_14)  [**3.1 提高代码执行性能的基本方法**](#heading_15)  [**3.2 循环体中不做影响性能的操作**](#heading_16)  [**3.3 集合类的高性能使用**](#heading_17)  [**3.4 使用细粒度锁解决高并发场景下的同步问题**](#heading_18)  [**4. 代码安全性规范**](#heading_19)  [**4.1 并发场景保障全局数据的安全**](#heading_20)  [**5. 异常处理和日志规范**](#heading_21)  [**5.1 异常处理规范**](#heading_22)  [**5.2 日志记录规范**](#heading_23)  [**6. 代码可读性规范**](#heading_24)  [**6.1 源文件规范**](#heading_25)  [**6.2 命名规范**](#heading_26)  [**6.3 代码风格**](#heading_27)  [**6.4 注释规范**](#heading_28)  [**7. 通用工具类/工具函数使用规范**](#heading_29)  [**7.1 通用工具类优先级定义**](#heading_30)  [**7.2 【强制】使用的工具类列表**](#heading_31)  [**7.3 【建议】使用的工具类**](#heading_32)  [**7.4 【严禁】使用的工具类**](#heading_33)  [**8. SK-Java使用规范**](#heading_34)  [**9. 最佳实践**](#heading_35)  [**附录**](#heading_36)  [**附录1：复杂try-with-resource场景下的异常处理的坑**](#heading_37) |

1. **规范分类**

|  |  |
| --- | --- |
| **分类** | **说明** |
| **【强制】** | * 强制规范是必须遵循的规范 * 违反规范无法通过静态检查或人工代码审查 * 因违反强制规范引入的BUG（含集成和内测），归类为编码阶段的低级BUG |
| **【建议】** | * 建议规范通常是针对某个特定问题的最佳实践 * 没有更好的方案时，优先选择 * 如果有更好的方案，欢迎提出并讨论，也欢迎在工程中实践 |
| **【参考】** | * 参考规范是结合业界常见最佳实践给出的参考 * 不要求完全遵守 |

2. **代码健壮性规范**

2.1 **入参必须先校验再使用**

* **【强制】**模块API方法入参使用注解声明校验规则

基本使用方式：http://conf.ruijie.work/pages/viewpage.action?pageId=78002559

|  |
| --- |
| Java **// 多个注解可以同时使用，注解之间不会发生冲突** public class IdArrRequest implements Request {  @NotEmpty // 表示数组内容不能出现null，即 new UUID[] { null, null}   @NotNull // 表示当前字段本身不能为null  @Size(min=1, max=5) // 限定数组长度范围  private UUID[] ids; } |

* **【强制】**模块内公共方法入参默认非空，使用Assert工具函数声明入参要求

|  |
| --- |
| Java import org.springframework.util.Assert;  CbbDeskInfo findById(UUID deskId) throws BusinessException {  Assert.notNull("deskId", "deskId is null");    // etc... } |

|  |
| --- |
| **通用约定**   * 方法入参的基本校验规则是方法提供者（Callee）给方法调用者（Caller）的要求 * 因参数违反基本校验规则导致的BUG归属方法调用者 |

* **【强制】**公共方法入参可以声明@Nullable注解，使用前需要判空

|  |
| --- |
| Java boolean checkNameDuplicate(@Nullable UUID deskId, String deskName) {  Assert.hasText("deskName", "deskName is null");     final Optional<CbbDeskInfo> existDeskOptional = deskDAO.findByDeskName(deskName);  if (!existDeskOptional.isPresent()) {  // 没找到同名记录  return false;  }    if (deskId == null) {  // deskId为空，说明当前方法是添加业务调用的，即已有同名记录  return true;   }    final CbbDeskInfo existDesk = existDeskOptional.get();  if (Objects.equals(deskId, existDesk.getId())) {  // deskId与已有记录一致，可以忽略冲突  return false;  }    return true; } |

* **【强制】**方法入参或返回值为Optional类型时，不能声明@Nullable注解

2.2 **调用公共方法后立即校验返回码/返回值**

* **【强制】**返回值是Optional类型时，使用前必须判断是否存在

|  |
| --- |
| Java import org.springframework.util.Assert;  CbbDeskInfo findById(UUID deskId) throws BusinessException {  Assert.notNull("deskId", "deskId is null");    // JPA的findById()方法返回 Optional 对象  final CbbDeskInfo deskInfo = deskDAO.findById(deskId).orElseThrow(()-> throw new BusinessException());    return deskInfo； } |

|  |
| --- |
| `Optinal.get()` 方法，调用之前必须使用`isPresent()`判断是否存在，其他`orXXX()`方法没有这个限制 |

* **【强制】**返回值是集合类或数组时，使用前必须判断是否为空集合或空数组

|  |
| --- |
| Java import org.springframework.util.CollectionUtils;   List<DefaultPublishAppDTO> defaultPublishAppList = obtainDefaultPublishAppsList();  if (CollectionUtils.isEmpty(defaultPublishAppList)) {  // 执行业务处理，这里的例子是直接返回空列表  return Collections.emptyList();  }     // 执行业务处理 |

|  |
| --- |
| 对于工具类的使用，请参考：[通用工具类/工具函数使用规范](https://ruijie.feishu.cn/docx/ZL1rdl072oZ4aaxmonvc4eSAnKg#FXY9dbNuwokVOvxKsUDc3oDen3f) |

* **【强制】**返回值允许为NULL时，使用前需判空

|  |
| --- |
| Java // 全局参数API的公共方法声明返回值可以为null public interface GlobalParameterAPI {  @Nullable   String findParameter(String key); }  // 使用的地方需要自行判空 final String value = globalParameterAPI.findParameter("some-key"); if (value == null) {  // etc... } |

|  |
| --- |
| **通用约定**   * 如果方法没有声明为·@Nullable·，返回null导致空指针是方法提供者（Callee）的BUG |

* **【强制】**执行IO操作（文件操作、网络通信）后，必须判断返回值

|  |
| --- |
| Java import java.io.File;  private void deleteAppIcon(String imageIconPath) {  Assert.hastext(imageIconPath, "imageIconPathis empty");    File file = new File(imageIconPath);  if (!file.exists()) {  // 文件不存在，保持幂等执行删除  return;  }    if（!file.delete()） {  // 删除失败，记录log  } } |

|  |
| --- |
| ·java.io.File·中所有与文件、目录写操作相关的方法，几乎都返回 `boolean` 类型，这些方法都需要判断返回值 |

2.3 **公共方法非必要不返回NULL**

* **【强制】**公共方法默认不返回NULL，需要返回NULL时，需要在方法上声明@Nullable并提供注释说明

|  |
| --- |
| Java public interface GlobalParameterAPI {  @Nullable   String findParameter(String key); } |

|  |
| --- |
| 当方法返回值声明为Nullable时，可以不抛出BusinessException |

* **【建议】**公共方法返回单个对象时，使用Optional作为返回值

|  |
| --- |
| Java Optional<CbbDeskInfo> findByDeskName(String deskName); |

|  |
| --- |
| 当方法返回值为Optional时，可以不抛出BusinessException |

* **【强制】**根据ID查询领域对象的API方法，如果领域对象不存在，抛出BusinessException

|  |
| --- |
| Java import org.springframework.util.Assert;  CbbDeskInfo findById(UUID deskId) throws BusinessException {  Assert.notNull("deskId", "deskId is null");    // JPA的findById()方法返回 Optional 对象  // 当desk不存在时抛出业务异常  final CbbDeskInfo deskInfo = deskDAO.findById(deskId).orElseThrow(()-> throw new BusinessException());    return deskInfo； } |

|  |
| --- |
| 对于返回DTO的方法，如果返回值未声明`@Nullable`、也不是·Optional·类型时，必须显式抛出`BusinessException` |

* **【强制】**方法返回值是集合类或数组时，返回空集合对象、空数组

|  |
| --- |
| 对于集合类，使用·java.util.Collections.emptyXXX()·函数 |

* **【强制】**方法返回值是集合类或数组时，集合或数组内部元素不能为NULL

2.4 **数据库查询不返回NULL**

* **【强制】**使用JPA的findById()方法获取数据库Entity对象
* **【强制】**不允许使用JPA的getOne()方法获取数据库Entity对象
* **【强制】**JPA自定义方法返回单条记录时，使用Optional作为返回值

|  |
| --- |
| Java import java.util.Optional;  public interface UserRepository extends SkyEngineJpaRepository<User, Long> {  Optional<User> findByUsername(String username); } |

* **【强制】**JPA自定义countBy()、existsBy()方法使用基本类型作为返回值

2.5 **IO对象使用时避免内存泄漏**

* **【强制】**IO对象必须在finally中调用close()方法

|  |
| --- |
| Java import com.ruijie.rcos.sk.base.io.IoUtil；  **// 写法1：使用try...catch...finally 语法** public class FileExample {  public static void main(String[] args) {  BufferedReader reader = null;  try {  reader = new BufferedReader(new FileReader("example.txt"));  String line;  while ((line = reader.readLine()) != null) {  // 对读取到的行进行业务处理  }  } catch (IOException e) {  // 这里通常需要记录LOG  } finally {  IoUtil.closeQuietly(reader)；   }  } }  **// 写法2：使用try..with..resource 语法** try (BufferedReader reader = new BufferedReader(new FileReader("example.txt"))) {  String line;  while ((line = reader.readLine()) != null) {  // 对读取到的行进行业务处理  } } catch (IOException e) {  // 这里通常需要记录LOG } |

|  |
| --- |
| 如果只有一个资源时，写法2较为简洁，如果有多个资源要处理时，默认使用写法1  理由：写法2在处理复杂场景时异常处理的控制非常复杂，可能涉及嵌套问题 详情参考[附录1](https://ruijie.feishu.cn/docx/ZL1rdl072oZ4aaxmonvc4eSAnKg#SGg4dfqXVoZUXVxc2zgc0Z8xn1c) |

2.6 **遍历数组/集合类时避免越界或并发修改异常**

* **【建议】**优先使用foreach、stream遍历数组、集合
* **【强制】**使用索引下标获取集合、数组元素前，需要校验长度
* **【强制】**在需要对集合元素做变更时使用Iterator遍历

|  |
| --- |
| Java List<String> list = new ArrayList<>(); list.add("A"); list.add("B"); list.add("C");  **// 【正例】使用迭代器安全的移除元素** Iterator<String> iterator = list.iterator(); while (iterator.hasNext()) {  String item = iterator.next();  if (item.equals("B")) {  **iterator.remove()**; // 使用迭代器的 remove 方法来安全删除元素  } }  **// 【反例】直接对list进行remove** for (String item : list) {  if (item.equals("B")) {  **list.remove(item)**; // 错误的操作，直接调用 List 的 remove 方法，这里会抛出ConcurrentModificationException   } } |

|  |
| --- |
| 虽然java并发包下的CopyOnWriteArrayList 在遍历过程中进行元素的增删操作不会抛出异常，但仍然需要有并发性能的消耗。所以**不允许使用反例的方式**。 |

2.7 **避免出现空指针异常**

* **【强制】**包装类型必须显式转换为基本类型

|  |
| --- |
| Java // 假设有这样一个函数，返回Integer类型的值 Integer getTotalNum();  **// 【反例】这种调用会有空指针的风险，且开发人员自身、代码走查人员一般看不出来** if (getTotalNum() > 0) {  }  **// 【正例】这样使用会更容易发现问题** Integer totalNum = getTotalNum()； if (totalNum == null) {  // 处理NULL  return; }  if (totalNum.intValue() > 0) {  // 处理true的场景 } else {  // 处理false的场景 } |

* **【强制】**使用Objects.equals()方法比较对象一致性
* **【强制】**数组对象转Stream之前，必须做非空校验
* **【强制】**禁止级联调用超过两层

|  |
| --- |
| Java // pattern是一个枚举  // **【正例】**下面是没有超过两层的例子 desk.getPattern().name();  // **【反例】**下面是超过两层的例子 deskAPI.findById(deskId).orElseThrow(/\* 抛异常 \*/).getPattern(); |

|  |
| --- |
| 上述规则不适用于stream、okhttp、StringBuilder等最佳实践就是级联使用的API |

2.8 **switch-case/if-else必须有兜底处理**

* **【强制】**switch代码块的每个case都必须有break，必须有兜底处理的default语句

|  |
| --- |
| Java // algorithm 是个枚举，表示不同的摘要算法 switch (algorithm) {  case SHA1: {  // 处理SHA1算法  break;  }  case SHA256:   case SHA512: {  // 当且仅当两个case可以用同一段代码处理时，可以参照按照这里的写法，上一个case不用写break  break；  }   case SHA224: {  // **【反例】**处理某些逻辑，下面没有break  }   case SHA384: {  // **【反例】SHA384复用了SHA 224的部分逻辑，上一个case没有break，这是违规的**  break;  }  case MD5: {  // 处理MD5算法  break；  }  default: {  // 如果前面的case已经穷尽了所有枚举的可能，default中会是dead code，参照下面的写法抛异常  throw new UnsupportedOperationException("unknown algorithm=" + algorithm);  } } |

* **【强制】**多分支if-else必须有兜底处理的else（即使永远走不到，也要有）

|  |
| --- |
| Java if (algorithm == Algorithm.SHA1) {  // 处理SHA1算法 } else if (algorithm == Algorithm.SHA256 || algorithm == Algorithm.SHA512) {  // 处理SHA256和SHA512算法 } else if (algorithm == Algorithm.MD5) {  // 处理MD5算法 } else {  // 如果前面的case已经穷尽了所有枚举的可能，default中会是dead code，参照下面的写法抛异常  throw new UnsupportedOperationException("unknown algorithm=" + algorithm); } |

2.9 **避免使用浮点类型导致计算精度问题**

* **【强制】**不允许使用浮点类型（float、double）做数学运算，优先使用整型或者使用BigDecimal代替
* **【强制】**磁盘、内存容量使用整形，需要呈现时进行转换
* **【强制】**不允许使用浮点类型作为API接口、数据库字段
* **【建议】**CPU占用率、内存占用率、网络占用率、磁盘占用率等字段以百分比形式表示，可能有小数，建议后端给出原始值（分子和分母），由前端进行计算

2.10 **避免过度使用包装类型**

* **【强制】**数据库Entity对象、模块的领域数据对象（DTO对象）的所有基本属性均使用包装类型
* **【强制】**模块API的入参和返回值，均使用包装类型
* **【例外】**以exists、count开头的函数，默认使用基本类型作为返回值
* **【建议】**参与判断和计算的临时变量，均使用基本类型

2.11 **避免修改上层入参和下层返回值**

* **【强制】**不修改集合类对象的返回值

|  |
| --- |
| Java // 定义方法，返回集合类 List<DeskDTO> findDeskByNames(String[] names); List<DeskDTO> findDeskByIds(UUID[] ids);  // 上层函数调用上述两个函数，并进行组合 List<DeskDTO> findByNameAndIds(String[] names, UUID[] ids) {  // 忽略入参校验    final List<DeskDTO> byNameList = findDeskByNames(names);  final List<DeskDTO> byIdList = findDeskByIds(ids);   **// 【反例】不能直接修改集合对象的返回值**  byNameList.addAll(byIdList);  return byNameList；    **// 【正例1】使用Stream语句进行合并**  return Stream.concat(byNameList.stream(), byIdList.stream())  .collect(Collectors.toList());    **// 【正例2】创建新集合进行合并**  final List<DeskDTO> resultList = Lists.newLinkedList();  resultList.addAll(byNameList);  resultList.addAll(byIdList );  return resultList ; } |

* **【强制】**只有明确有fill、put、convert语义的函数可以修改入参内容

|  |
| --- |
| Java **// 【反例】check、validate、is函数中修改入参内容** void checkDesk(DeskDTO desk) {  // 执行某些check逻辑  // ...    // 修改入参的值  desk.setNeedUpdatePassword(false); }  **// [正例]DTO->Entity的转换类中修改入参内容** void convertToEntity(DeskDTO deskDto, DeskEntity deskEntity) {  // 忽略入参校验    deskEntity.setId(deskDto.getId());  // etc ... } |

2.12 **其他**

* **【强制】**作为HashMap的Key的对象或者传入HashSet的对象需要同时复写equals和hashcode方法
* **【强制】**方法内部的局部变量不能覆盖或变更方法入参

|  |
| --- |
| Java List<RcaHostDTO> tryFindByHostIdList(List<UUID> **tryHostIdList**) {  if(CollectionUtils.isEmpty(tryHostIdList)) {  // 这里替换了方法入参，应当声明新的字段  **tryHostIdList = getDefaultHostIdList();**  }    return hostDAO.findByIds(tryHostIdList); } |

* **【建议】**方法入参、局部变量默认标记为final
* **【强制】**数据库事务边界内（tx包）只能编写与数据库处理有关的逻辑
* **【强制】**包装类型必须使用Objects.equals()方法比较对象一致性，不能用==

3. **代码高性能规范**

3.1 **提高代码执行性能的基本方法**

* **【强制】**当多个判断同时存在时，先执行代价小的判断

|  |
| --- |
| Java **// 【反例】数据库查询的代价大于属性判断，应当先做属性判断** private void checkVisitorLimit(UUID userid, IacuseerTypeEnum userType) throws BusinessException {  // 这里是数据库查询  long count = userDesktopDAO.countByUserCreatingDessktop(userId);    // 如果第一个条件未false，前一个数据库查询就浪费了  if (IacUserTypeEnum.VISITOR== userType && count>= ConIstants.VISITOR\_USER\_DESKTOP\_MAX\_NUM)  throw new BusinessException(BusinessKey.RCDC\_USER\_CLOUDDESKTOP\_VISITOR\_USER\_VM\_MAX,  String.valueof(Constants.VISITOR\_USER\_DESKTOP\_MAX\_NUM));  } }  **// 【正例】数据库查询的代价大于属性判断，应当先做属性判断** private void checkVisitorLimit(UUID userId, IacUserTypeEnum userType) throws BusinessException {  // 先判断用户类型  if (IacUserTypeEnum.VISITOR != userType) {  return;  }    // 这里是数据库查询  long count = userDesktopDAO.countByUserCreatingDessktop(userId);    // 检查用户桌面数量是否超过限制  if (count >= Constants.VISITOR\_USER\_DESKTOP\_MAX\_NUM) {  throw new BusinessException(  BusinessKey.RCDC\_USER\_CLOUDDESKTOP\_VISITOR\_USER\_VM\_MAX,  String.valueOf(Constants.VISITOR\_USER\_DESKTOP\_MAX\_NUM)  );  }  } |

* **【强制】**同样的数据库查询、RPC查询在同一个模块的业务流中只做一次
* 同一个类中，public方法及内部的private方法，对于同一个记录的数据库、RPC查询只做一次
* **【建议】**同一个模块由公共API关联的业务流中，对于同一个记录的数据库、RPC查询只做一次

3.2 **循环体中不做影响性能的操作**

* **【强制】**不能在循环体内单条查询关联表的数据

|  |
| --- |
| Java // web界面查询结果集构造时，如果不用多表关联或视图，通常会出现上述问题 // 下面的示例代码仿写桌面列表的查询界面，需要同时呈现桌面对应的镜像名称  // 步骤1：查询桌面列表 final PageQueryResponse<CbbDeskInfoDTO> deskInfoResponse = deskAPI.pageQuery(request);  // 步骤2：构造镜像ID列表 final UUID[] imageIds = StreamSupport.stream(deskInfoResponse) //  .map(CbbDeskInfoDTO::getImageId) //  .toArray(UUID[]::new);   // 步骤3：查询相关镜像信息 final List<CbbImageTemplateDTO> imageTempateList = imageAPI.findByIds(imageIds);  // 步骤4：转换为镜像map final Map<UUID, CbbImageTemplateDTO> imageTemplateMap = imageTempateList.stream() //  .collect(Collectors.toMap(CbbImageTemplateDTO::getId, Function.identity()));   // 步骤5：在for循环中构造最终结果 return StreamSupport.stream(deskInfoResponse).map(desk -> {  final CbbImageTemplateDTO image = imageTemplateMap.get(desk.getImageId());  // etc...   return new WebDto(); }).toArray(WebDto[]::new); |

* **【强制】**不能在循环体内单条执行RPC查询
* **【强制】**循环体内正向流程只能打印DEBUG级别的LOG

|  |
| --- |
| Java deskList.stream.map(desk -> {  final CbbImageTemplateDTO image = imageTemplateMap.get(desk.getImageId());  // 循环体内打印LOG时，必须打印DEBUG级别，并且预先判断是否可以打印  if (LOGGER.isDebugEnabled()) {  LOGGER.debug("xxxx");  }  return new WebDto(); }).toArray(WebDto[]::new); |

* **【强制】**循环体内必须使用StringBuilder进行字符串拼接

|  |
| --- |
| Java final StringBuilder sb = new StringBulder(); for (int i = 0; i < 10; i++) {  sb.append(i); } return sb.toString(); |

3.3 **集合类的高性能使用**

* **【强制】**使用Map.entry进行Map遍历

|  |
| --- |
| Java Map<String, Integer> map = new HashMap<>(); map.put("A", 1); map.put("B", 2); map.put("C", 3);  // 使用 for-each 循环遍历 Map 条目 for (Map.Entry<String, Integer> entry : map.entrySet()) {  System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue()); }  // 或者使用流来遍历 Map 条目 map.entrySet().stream()  .forEach(entry -> System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue())); |

* **【强制】**Collection转数组时使用toArray()方法进行转换

|  |
| --- |
| Java  List<String> list = new ArrayList<>();  list.add("A");  list.add("B");  list.add("C");    // 创建一个大小与集合相同的数组  String[] stringArray = list.toArray(new String[list.size()]); |

* **【强制】**数组转Collection时使用Arrays.asList()方法进行转换

|  |
| --- |
| Java // 假设array是一个数组 String[] array = {"a", "b", "c"};  // 使用Arrays.asList()方法将数组转换为List List<String> list = Arrays.asList(array); |

|  |
| --- |
| ·Arrays.asList·返回的内容不支持增、删，只能用来遍历。 |

3.4 **使用细粒度锁解决高并发场景下的同步问题**

* **【强制】**不能使用Guava Interners 或 String.interner() 执行细粒度的加锁
* **【建议】**使用Guava Striped实现基于用户ID、会话ID等信息的细粒度加锁

|  |
| --- |
| Java import com.google.common.util.concurrent.Striped; import java.util.concurrent.locks.Lock;  public class StripedExample {  private static final int USER\_CONCURRENCY\_LEVEL = 100; // 并发级别  private static final Striped<Lock> userLocks = Striped.lock(USER\_CONCURRENCY\_LEVEL);   public void performUserOperation(String userId) {  Lock lock = userLocks.get(userId);  lock.lock();  try {  // 进行用户操作  } finally {  lock.unlock();  }  } } |

|  |
| --- |
| 如果业务中需要使用细粒度锁，那么应当优先选用上述形式 |

4. **代码安全性规范**

4.1 **并发场景保障全局数据的安全**

* **【强制】**全局变量/单例对象属性必须使用并发安全的数据类型

|  |
| --- |
| Java import java.util.concurrent.atomic.AtomicInteger;  /\*\* 并发安全的计数器代码样例 \*\*/ public class Counter {  **// 类/对象属性如果作为全局变量，需要标记为final，并且使用java.util.concurrent包下的安全类**  private final **AtomicInteger** count = new **AtomicInteger**(0);   public int getCount() {  return count.get();  }   public void increase() {  count.incrementAndGet();  }   public void decrease() {  count.decrementAndGet();  } } |

|  |
| --- |
| Java import java.util.Map; import java.util.concurrent.ConcurrentHashMap;  public class SimpleCache {  /\*\* **类/对象属性如果作为全局变量，需要标记为final，并且使用java.util.concurrent包下的安全类** \*\*/  private final Map<String, String> cache = new ConcurrentHashMap<>();   public void put(String key, String value) {  cache.put(key, value);  }   public String get(String key) {  return cache.get(key);  }   public void remove(String key) {  cache.remove(key);  }   public boolean containsKey(String key) {  return cache.containsKey(key);  } } |

|  |
| --- |
| 如果显式的在类上添加 ·@ThreadSafe· 注解，则说明这个类是线程安全的，允许类属性和对象属性使用不安全的数据类型 |

* **【强制】**全局变量/单例对象属性变-更时需要加锁

|  |
| --- |
| 如果显式的在类上添加 ·@ThreadSafe· 注解，则说明这个类是线程安全的，变更属性值是无需加锁。  多个锁的场景下，要用一致的加锁顺序，避免死锁 |

* **【强制】**使用乐观锁保障数据库更新的一致性

|  |
| --- |
| 乐观锁的边界应当在Service层，标准写法待框架修订后补充 |

* **【强制】**使用DateTimeFormatter进行日期格式转换

|  |
| --- |
| Java import java.time.LocalDateTime; import java.time.format.DateTimeFormatter;  public class LogFormatter {  private static final DateTimeFormatter LOG\_FORMATTER = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss.SSS");   public static void main(String[] args) {  // 获取当前时间  LocalDateTime currentTime = LocalDateTime.now();   // 格式化日期时间  String formattedTime = currentTime.format(LOG\_FORMATTER);   // 输出格式化后的时间  System.out.println("Formatted time for log: " + formattedTime);  } } |

5. **异常处理和日志规范**

5.1 **异常处理规范**

* **【强制】**同一个方法中try关键字不能嵌套
* **【强制】**try代码块中的代码不能超过20行
* **【强制】**finally语句中不能出现return关键字
* **【强制】**不允许catch特定的异常

|  |
| --- |
| 这个规则的目要求**明显是程序BUG的异常不能通过catch方式规避**，需要通过前置判断正常处理   * 不能catch的运行时异常（这里只列举常见的，并未穷尽，未来可能新增） * IllegalStateException、IllegalArgumentException：通常是入参校验时由被调用函数抛出，应当前置处理，确保不会传入不合法的参数 * NullPointerException：通常是入参或返回值未校验直接使用抛出，应当先校验再使用 * UnsupportedOperationException：通常是请求了未实现的方法抛出，应当前置处理 * 不能catch Error及其所有子类 |

* **【强制】**仅允许主动抛出BusinessException，以及特定的运行时异常

|  |
| --- |
| 允许主动抛出的运行时异常：   * 针对入参校验：IllegalStateException、IllegalArgumentException * 针对未实现的方法、不会走到的switch-default分支：UnsupportedOperationException |

* **【强制】**catch异常后必须进行处理

|  |
| --- |
| Java **// 方式1：记录LOG、丢弃异常** // 适合于线程run函数中的异常处理，无需再次抛出 try {  iacProducerAPI.sendLogToSub(iacLogMqInfoDTO); } catch (Exception e) {   LOGGER.error("发送身份中心日志给子系统mq消息失败：{}" + JSON.toJSONString(iacLogMqInfoDTO), e); }  **// 方式2：记录LOG、重新抛出** // 适合于同一个流程中有多个方法会抛出同类型异常，有一个方法调用需要独立记录LOG，便于故障排查的场景 try {  iacProducerAPI.sendLogToSub(iacLogMqInfoDTO); } catch (BusinessException e) {   LOGGER.error("发送身份中心日志给子系统mq消息失败：{}" + JSON.toJSONString(iacLogMqInfoDTO), e);  throw e; }  **// 方式3：封装为其他异常，重新抛出** // 适合于异常类型的转换 try {  iacProducerAPI.sendLogToSub(iacLogMqInfoDTO); } catch (IOException e) {   throw new BusinessException('key', e); } |

* **【建议】**不需要每一层都处理BusinessException

|  |
| --- |
| 框架关键点均对BusinessException做过兜底处理，例如web界面  符合以下要求，可以完全不catch BusinessException，只要方法声明抛出BusinessException即可   * 无需基于特定错误码执行业务判断和处理 * 无需对下层抛出的错误消息进行重封装 |

* **【建议】**根据异常类型进行分别catch和处理

|  |
| --- |
| Java **// 方式1：每种异常一个catch** try {  // 可能会抛出多种类型的异常 } catch (IOException e) {  // 处理 IO 异常 } catch (BusinessException e) {  // 处理业务异常 } catch (Exception e) {  // 处理其他类型的异常 }  **// 方式2：用多个异常可以同时catch** try {  // 可能会抛出多种类型的异常 } catch (IOException | BusinessException e) {  // 假设IO异常和业务异常可以用同样的逻辑进行处理 } catch (Exception e) {  // 处理其他类型的异常 } |

* **【强制】**catch语句中不能对异常做instaneof或强转

|  |
| --- |
| Java **// 【反例】下面的写法是不合理的，不要在catch中强转或判断原始异常类型，应当用多个catch语句实现** try {  // 可能会抛出多种类型的异常 } catch (Exception e) {  // 根据不同类型的异常进行具体处理  if (e instanceof IOException) {  // 处理 IO 异常  } else if (e instanceof SQLException) {  // 处理 SQL 异常  } }  **// 【例外】异常嵌套场景下，下列写法是合理的** // 这种写法在业务中不多见 try {  yourMethod(); // 调用可能会抛出 BusinessException 的方法 } catch (BusinessException outerException) {  if (outerException.getCause() instanceof BusinessException) {  BusinessException innerException = (BusinessException) outerException.getCause();  // 获取内层 BusinessException 并进行业务处理  // 对 innerException 进行业务处理的代码  } else {  // 处理外层 BusinessException 的代码  } }  **// 【例外】框架开发中，异常嵌套相对较多，以下写法是合法的** try {  // 在这里调用代理的目标对象的方法，可能会抛出 BusinessException  Object result = method.invoke(target, args);  return result; } catch (InvocationTargetException e) {  if (e.getCause() instanceof BusinessException) {  BusinessException innerException = (BusinessException) e.getCause();  // 获取内层 BusinessException 并进行业务处理  // 对 innerException 进行业务处理的代码  } else {  // 处理其他类型的异常  }  throw e.getCause(); // 抛出内层异常，或者根据实际情况进行处理 } |

5.2 **日志记录规范**

详情：[云桌面Log日志规范\_V1.0](https://ruijie.feishu.cn/docx/M1s3dHlcJoc2hNxm6gKcwvuLnxd)

* **【强制】**任何公有调用失败后，立即记录LOG

|  |
| --- |
| Java **// 场景1：IO函数操作失败，根据返回值判断是否成功** if（!file.delete()） {  // 删除失败，记录log  **LOGGER.error("文件[{}]删除失败", fileAbsPath);**  // 通常异常校验后需要返回，不执行后续业务  return; }  **// 场景2：出现IO异常时，第一时间记录LOG** // LOG中不能丢失原始异常堆栈 try (FileWriter writer = new FileWriter(fileName)) {  writer.write(content); } catch (IOException e) {  **LOGGER.error("写入文件[{}]失败"，fileName, e);**    // etc...  } |

* **【强制】**LOG记录中不应当包含i++之类对数据状态产生影响的语句

|  |
| --- |
| Java **// 【反例】下面的写法会导致开启debug级别时，在线人数记录错误** if (LOGGER.isDebugEnabled()) {  LOGGER.debug("当前在线人数: {}" ，++counter); } |

* **【强制】**LOG记录中不应当包含toJson()、toString()、getXXX()之外的方法调用
* **【强制】**INFO及以下级别的LOG记录，需要用if判断

|  |
| --- |
| Java if (LOGGER.isDebugEnabled()) {  LOGGER.debug("当前在线人数: {}" ，counter); } |

6. **代码可读性规范**

6.1 **源文件规范**

* **【强制】**统一使用UTF-8编码。

6.2 **命名规范**

* **【强制】**包名、类名、方法名、变量名只能使用字母、数字和下划线。命名均不能以下划线或美元符号开始，也不能以下划线或美元符号结束。

【反例】

\_name , $Object , name\_ , name$, Object$

* **【强制】**命名禁止使用拼音与英文混合、纯拼音、中文
* **【强制】**方法名、参数名、成员变量、局部变量都首字母小写，必须遵从驼峰形式。
* **【强制】**变量名避免单字母形式命名，除非变量作用域非常小，如for 循环里的变量。
* **【强制】**数组定义中[]必须放在变量名前面

|  |
| --- |
| Java String[] args; |

* **【强制】**包名，必须全部小写，不使用下划线，名词用单数。

|  |
| --- |
| Java com.ruijie.util |

* **【强制】**类名，必须大写字母开头，使用名词或名词短语，多个单词组成使用驼峰标识法，如TimeZone。

避免泛泛的词。

|  |
| --- |
| downloadPage()要优于getPage()；countSize(),calculateSize()要优于getSize()，size()，前者表示需要消耗时间。 |

* **【强制】**测试类以Test结尾，如：UserTest
* **【强制】**测试方法名以test前缀，如：testXXX()
* **【强制】**异常类以Exception结尾，如：NumberFormatException
* **【强制】**抽象类以Abstract 开头，如：AbstractAccessTask
* **【强制】**如果使用到了设计模式，需要在类名中体现出具体模式。

|  |
| --- |
| Java public class OrderFactory {} public class LoginProxy {} public class ResourceObserver {} |

* **【强制】**领域模型相关命名全部大写，如DTO、DAO等。

正例：

UserDAO, UserDTO

反例：

UserDao , UserDto

* **【强制】**接口名必须是名词或名词短语，也可以是形容词（通常是–able 的形式），另外接口不加I前缀，实现类以Impl结尾。
* **【强制】**枚举成员名称需要全大写，单词间用下划线隔开。
* **【强制】**方法名，必须使用动词或动词短语。
* **【强制】**常量命名全部大写，且统一定义为static final，单词间用下划线隔开，力求语义表达完整清楚，不要嫌名字长。

正例：

PAGE\_SIZE, WORK\_DAYS\_PER\_WEEK, DEFAULT\_DATE\_FORMAT

public static final int PAGE\_SIZE = 10

反例：

SIZE, WORK\_DAYS, DATE\_FORMAT

* **【强制】**long 或者 Long 初始赋值时，必须使用大写的 L，不能是小写的l，小写l容易跟数字1混淆，造成误解。

正例：long height = 2L

反例：long height = 2l

* **【建议】**布尔返回值的函数命名，以is、has、can、should、need、exists等前缀，如：isEmpty() hasSiblings()。
* **【强制】**以is开头的函数，仅用于判断状态，不允许执行修改数据状态的业务，例如：isEmpty()，isConnected()
* **【强制】**以check、validate开头的函数，仅用于数据校验，校验失败应当抛出BusinessException
* **【参考】**[**常见标准命名规范**](https://ruijie.feishu.cn/base/Ii3nbhYVUahYLwsxkEsc9orYnqd?table=tblUi6inCNfVOzBi&view=vew2BSkpMR)
* **【参考】**如果想不到好的命名，可以参考[程序员“起名”头痛根治指南](https://mp.weixin.qq.com/s/loaaKlE44P4VxMgLY3f2hw)，下面是一些常用的例子：
* 获取对象的方法用 findBy、get、list 做前缀。
* 获取统计值的方法用 count 做前缀。
* 判断是否存在用exists、existsBy做前缀
* 插入的方法用 save 或 insert 做前缀。
* 删除的方法用 remove 或 delete 做前缀。
* 修改的方法用 update、change、modify 做前缀。

6.3 **代码风格**

* **【强制】**左大括号前不换行。

|  |
| --- |
| Java if (condition()) {  try {  something();  } catch (ProblemException e) {  ...  } } |

* **【强制】**一行不超过200个字符。
* **【强制】**方法与方法之间必须留一行空白行。
* **【强制】**单个方法不应当超过50行
* **【强制】**单个if/else/while/for语句块不应当超过20行
* **【强制】**单个lambda匿名函数语句不超过5行
* **【强制】**不含空格的连续代码不应当超过10行，超过10行需要使用空行分隔

|  |
| --- |
| Java @Override public RemoteIacConfigDTO receiveManageNotify(RemoteIacRequest request) throws BusinessException {  Assert.notNull(request, "request can not be null");   **/\*\* 入参校验和业务校验之间留一个空行 \*/**  RemoteIacConfigDTO oldRemoteIacConfig = remoteIacService.getRemoteIacConfig();  if (Boolean.TRUE.equals(oldRemoteIacConfig.getEnable())) {  if (!StringUtils.equals(oldRemoteIacConfig.getIacAddr(), request.getRemoteIp())) {  throw new BusinessException(RestErrorCode.IAC\_OPEN\_API\_REMOTE\_IAC\_EXIST);  }  }   **/\*\* 业务校验业务业务处理之间留一个空行 \*\*/**  // 纳管通知消息处理，先进行地址检测，只有检测通过的情况下，才允许保存地址信息  Integer port = Objects.isNull(request.getRemotePort()) ? Constants.DEFAULT\_REMOTE\_PORT : request.getRemotePort();  try {  testRemoteIacConnect(request.getRemoteIp(), port);  } catch (BusinessException e) {  LOGGER.error("纳管通知处理，校验连通性，无法连通无法保存，地址信息[{}:{}]", request.getRemotePort(), port);  throw new BusinessException(RestErrorCode.IAC\_OPEN\_API\_REMOTE\_IAC\_ADDRESS\_TEST\_ERROR, e);  }   /\*\* 有相关性的POJO构造放在一起，并与前面的逻辑用空行分隔 \*\*/  // 检测通过，保存统一身份中心配置  RemoteIacConfigDTO remoteIacConfigDTO = new RemoteIacConfigDTO();  remoteIacConfigDTO.setEnable(true);  remoteIacConfigDTO.setIacAddr(request.getRemoteIp());  remoteIacConfigDTO.setIacPort(request.getRemotePort());  remoteIacConfigDTO.setConnect(true);  remoteIacService.saveRemoteIacConfig(remoteIacConfigDTO);    /\*\* return语句单独一行，与前面的业务用空行分隔开 \*\*/  return remoteIacConfigDTO; } |

|  |
| --- |
| Bean的Copy代码不受本规则限制 |

* **【强制】**运算符号左右各加一个空格。

|  |
| --- |
| Java **【正例】** int i = 1; int j = i > 0 ? 2 : 3;  **【反例】** int i=1; int j=i>0?2:3; |

* **【强制】**左大括号前要有一个空格。

|  |
| --- |
| Java **【正例】** if (condition()) {  ... }  **【反例】** if(condition()){  ... } |

* **【强制】**代码缩进用4个空格。
* **【强制】**if-else ，while，try-catch-finally、switch、case语句，即使只有一行代码，也必须用{ }括起来。
* **【强制】**同一个方法中if/for/while/switch嵌套不能超过2层

|  |  |
| --- | --- |
| **规范的意义**  使用卫述句写法减少if层次嵌套，提升代码可读性   |  | | --- | | Guard Clause（守卫条款）是一种编程技巧，用于在函数或方法的开头对输入参数进行验证，如果参数不符合预期，就立即返回，避免进入函数的主体部分。这有助于提高代码的可读性和可维护性。 | |

|  |
| --- |
| Java **// 【正例】使用卫语句进行校验，尽早返回失败，减少缩进** public void processInput(int a, int b) {  if (a <= 0) {  System.out.println("a is not positive");  return;  }  if (b <= 0) {  System.out.println("b is not positive");  return;  }   List<MyPOJO> list = dao.findAll();  if (CollectionUtils.isEmpty(list)) {  System.out.println("Retrieved list is empty or null");  return;  }   for (MyPOJO pojo : list) {  if (pojo == null) {  continue;  }     // etc...   } }  **// 【反例】嵌套多层if，难以看出正常逻辑和异常逻辑** public void processInput(int a, int b) {  if (a <= 0) {  System.out.println("a is not positive");  } else {  if (b <= 0) {  System.out.println("b is not positive");  } else {  List<MyPOJO> list = dao.findAll();  if (CollectionUtils.isEmpty(list)) {  System.out.println("Retrieved list is empty or null");  } else {  for (MyPOJO pojo : list) {  if (pojo != null) {  // etc...  }  }  }  }  } } |

* **【强制】**if/while/for语句中的判断条件不能超过两个

|  |
| --- |
| Java **// 【反例】CDC中的真实案例（if中条件过多，难以扩展）** // 只允许WIN7、WIN10, WIN11\tci uos kos进行配置 if (!CbbOsType.isWin7UpOS(terminalDTO.getImageTemplateOsType())  && !((CbbOsType.UOS\_64 == terminalDTO.getImageTemplateOsType() || CbbOsType.KYLIN\_64 == terminalDTO.getImageTemplateOsType())  && terminalDTO.isVOIModel())) {  LOGGER.error("终端[{}]上桌面的系统类型为[{}]，不允许开启系统盘自动扩容", terminalIdentification, terminalDTO.getImageTemplateOsType());  throw new BusinessException(BusinessKey.RCDC\_TERMINAL\_CONFIG\_FULL\_SYSTEM\_DISK\_ONLY\_SUPPORT\_WIN7\_AND\_WIN10, terminalIdentification,  terminalDTO.getImageTemplateOsType().toString()); }  **// 【正例】【修订后】拆分一个独立的函数，逻辑清晰性和关联性变强** private void canAutoExpand(TerminalDTO terminalDTO, String terminalIdentification) {  if (CbbOsType.isWin7UpOS(terminalDTO.getImageTemplateOsType())) {  return;  }   if (CbbOsType.UOS\_64 == terminalDTO.getImageTemplateOsType() && terminalDTO.isVOIModel()) {  return;  }   if (CbbOsType.KYLIN\_64 == terminalDTO.getImageTemplateOsType() && terminalDTO.isVOIModel()) {  return;  }   LOGGER.error("终端[{}]上桌面的系统类型为[{}]，不允许开启系统盘自动扩容", terminalIdentification, terminalDTO.getImageTemplateOsType());  throw new BusinessException(BusinessKey.RCDC\_TERMINAL\_CONFIG\_FULL\_SYSTEM\_DISK\_ONLY\_SUPPORT\_WIN7\_AND\_WIN10, terminalIdentification,  terminalDTO.getImageTemplateOsType().toString()); } |

|  |
| --- |
| 如果原有if判断中条件为2个，需要增加一个判断条件，**修订者有义务抽取出一个独立的方法专门做判断** |

* **【强制】**三目运算符中不能包含复杂语句

|  |
| --- |
| Java **// 【正例】没有复杂语句时，可以使用三目运算符** return (hour >= 12) ? "pm" : "am";  **// 【反例】有复杂语句时，使用三目运算符不易懂，且容易有bug** // 如果 pageSize = 0，下面语句会报错 pageCount = (total / pageSize) + ((total % pageSize == 0) ? 0 : 1);  **// 【正例】【修改后】先不考虑pageSize=0的情况** int pageCount = total / pageSize; int remainder = total % pageSize; if(remainder == 0){  return pageSize; } return pageCount + 1; |

* **【强制】**每次声明一个变量。

|  |
| --- |
| Java **【正例】** int i; int j;  **【反例】** int i,j; |

6.4 **注释规范**

* **【强制】**类、类属性、类方法的注释必须使用/\*\*内容\*/格式，不得使用 //内容 方式。

|  |
| --- |
| Java  /\*\*   \* Description: Function Description   \* Copyright: Copyright (c) 2017   \* Company: Ruijie Co., Ltd.   \* Create Time: 2017/08/01   \* @author    \*/ |

* **【强制】**模块接口类、POJO类、RPC接口类必须添加类和方法注释，其他内部类无需添加类、方法注释
* **【强制】**方法内部单行注释，在被注释语句上方另起一行，使用//注释。方法内部多行注释 使用/\* \*/注释。
* **【强制】**没用的代码需要删除，不应当进行注释
* **【强制】**特殊注释标记使用，TODO FIXME

|  |
| --- |
| TODO：待办事项，标记暂时没有完成的事项，后面需要完成。  FIXME：标记存在bug或待完善部分说明，必须及时处理。 |

* **【强制】**标记为“Deprecated”的接口，必须同时在注释中说明替代的方法和使用方法
* **【强制】**注释要说明“为什么这么做”，而不是“做了什么”

|  |
| --- |
| Java /\*\*  \* 计算对象的哈希码。  \* 为了在计算哈希码时减少碰撞的可能性，并且保持计算效率，我们选择使用31作为乘法因子。  \* 31有一个很好的性质，即它可以被编译器优化为移位和减法操作，这样可以更高效地计算哈希码。  \* 因此，通过使用31作为乘法因子，我们可以在保持哈希码计算效率的同时，尽可能地减少哈希码碰撞的发生。  \*/ @Override public int hashCode() {  int result = 1;  result = 31 \* result + (name != null ? name.hashCode() : 0);  result = 31 \* result + age;  return result; } |

* **【强制】**不允许使用魔法数字（泛指不能自我描述的符号），必须命名成有意义的常量。
* **【强制】**POJO类中，setter 、getter 方法不能加入业务逻辑代码。
* **【强制】**公共方法、抽象类的abstract方法参数不超过6个，公共构造函数、私有函数的参数数量不受限制

7. **通用工具类/工具函数使用规范**

7.1 **通用工具类优先级定义**

通用工具类主要做一些校验、通用非业务处理，选择优先级如下（从高到低）：

|  |  |
| --- | --- |
| 优先级 | 说明 |
| 第1优先级 | SK-Java封装的工具类：通常在`com.ruijie.rcos.sk.base.util`包中，例如：`com.ruijie.rcos.sk.base.util.StringUtils` |
| 第2优先级 | JDK自带的工具类：通常在`java.util`包中以s结尾，例如`java.util.Arrays`、`java.util.Collections` |
| 第3优先级 | Spring中自带的工具类：通常在`org.springframework.util`包中，例如`org.springframework.util.Assert`、`org.springframework.util.CollectionUtils` |
| 第4优先级 | `Google Guava`工具类，例如：`com.google.common.collect.Lists` |
| 第5优先级 | `apache commons lang`工具类，通常在`org.apache.commons.lang3`包中，例如：`org.apache.commons.lang3.ArrayUtils` |

7.2 **【强制】使用的工具类列表**

|  |  |  |  |
| --- | --- | --- | --- |
| 所属库 | 所属包 | 类名 | 功能 |
| SK-Java | `com.ruijie.rcos.sk.base.util` | StringUtils | 字符串的大多数操作，包括：   * 十进制和十六进制互转 * 字符串判空、拼接、查找 |
| SK-Java | `com.ruijie.rcos.sk.base.util` | CollectionUitls | 集合类的常见操作，包括：   * 判空、查找 * 交集计算、并集计算 |
| JDK | java.util | Collections | 集合类常见操作，包括：   * 排序、打乱 * 空集、线程安全的集合 |
| JDK | java.util | Arrays | 数组常见操作，包括：   * 排序，查找 |
| JDK | java.util | Objects | 对象的常见操作，包括：   * 判空、计算hash值 |
| Spring | org.springframework.util | Assert | 断言校验的常见操作，包括：   * 判空、类型匹配 |
| apache | org.apache.commons.lang3 | ArrayUtils | 数组常见操作，包括：   * 判空、反转 |
| apache | org.apache.commons.lang3 | BooleanUtils | Boolean类常见操作，避免空指针 |

7.3 **【建议】使用的工具类**

|  |  |  |  |
| --- | --- | --- | --- |
| 所属库 | 所属包 | 类名 | 功能 |
| Guava | com.google.common.collect | Lists/Maps/Sets | List/Map/Set对象的工厂方法，可以根据需要构造固定长度的集合类 |

7.4 **【严禁】使用的工具类**

第三方包内部使用的工具类有时会与通用工具类重名，这些工具类严禁在业务代码中使用，包括：

|  |  |
| --- | --- |
| 所属库 | 类名 |
| Spring | org.springframework.cglib.core.CollectionUtils |
| log4j | org.apache.logging.log4j.core.util.ArrayUtils |
| log4j | org.apache.logging.log4j.core.util.Assert |
| netty | io.netty.util.internal.StringUtil |
| junit | org.junit.Assert |

8. **SK-Java使用规范**

[SK-JAVA](https://ruijie.feishu.cn/wiki/wikcnrSETEuytGJrlyflSdKeaJe?from=from_copylink)

9. **最佳实践**

**附录**

**附录1：复杂try-with-resource场景下的异常处理的坑**

在 Java 中，被抑制的异常（suppressed exceptions）是指在 try-with-resources 语句中，当关闭资源时发生异常，会将这些异常添加到当前异常对象的被抑制异常列表中。这些被抑制的异常可以通过 Throwable 类的 getSuppressed() 方法获取到。

然而，对于在 try-with-resources 块中发生的其他异常（比如空指针异常等），这些异常并不会被添加到被抑制异常列表中，因为它们并非由 try-with-resources 语句引起的资源关闭操作导致的异常。

因此，如果在内层 try-with-resources 块中发生了其他异常（比如空指针异常），这些异常不会被归为被抑制异常，而是会直接抛出并由外层的 catch 块来处理。

总结一下：

* 被抑制的异常是指在 try-with-resources 语句中资源关闭操作导致的异常，可以通过 getSuppressed() 方法获取到。
* 其他在 try-with-resources 块中发生的异常（比如空指针异常）不会被归为被抑制异常，而是直接抛出并由外层的 catch 块来处理。

因此，对于内层 try-with-resources 块中的其他异常（比如空指针异常），您可以在外层的 catch 块中直接处理这些异常，而不需要通过 getSuppressed() 方法来获取。

|  |
| --- |
| Java SSLSocketFactory ssf = createSocketFactory();  try (SSLSocket socket = (SSLSocket) ssf.createSocket()) {  socket.connect(new InetSocketAddress(ip, PORT), 6000);  socket.setSoTimeout(15000);  LOGGER.debug("send data to " + ip + ":" + PORT);   try (InputStream sockIn = socket.getInputStream();  OutputStream sockOut = socket.getOutputStream()) {  // 使用 sockIn 和 sockOut 进行数据的读取和写入操作  } } catch (IOException e) {  // 处理外层的 IOException  e.printStackTrace();  // 处理内层 try-with-resources 块中的异常  Throwable[] suppressed = e.getSuppressed();  for (Throwable t : suppressed) {  t.printStackTrace();  } } |