36-实战二（上）：程序出错该返回啥？NULL、异常、错误码、空对象？

我们可以把函数的运行结果分为两类。一类是预期的结果，也就是函数在正常情况下输出的结果。一类是非预期的结果，也就是函数在异常（或叫出错）情况下输出的结果。比如，在上一节课中，获取本机名的函数，在正常情况下，函数返回字符串格式的本机名；在异常情况下，获取本机名失败，函数返回UnknownHostException异常对象。

在正常情况下，函数返回数据的类型非常明确，但是，在异常情况下，函数返回的数据类型却非常灵活，有多种选择。除了刚刚提到的类似UnknownHostException这样的异常对象之外，函数在异常情况下还可以返回错误码、NULL值、特殊值（比如-1）、空对象（比如空字符串、空集合）等。

每一种异常返回数据类型，都有各自的特点和适用场景。但有的时候，在异常情况下，函数到底该返回什么样的数据类型，并不那么容易判断。比如，上节课中，在本机名获取失败的时候，ID生成器的generate()函数应该返回什么呢？是异常？空字符？还是NULL值？又或者是其他特殊值（比如null-15293834874-fd3A9KBn，null表示本机名未获取到）呢？

函数是代码的一个非常重要的编写单元，而函数的异常处理，又是我们在编写函数的时候，时刻都要考虑的。所以，今天我们就聊一聊，如何设计函数在异常情况下的返回数据类型。

话不多说，让我们正式开始今天的学习吧！

从上节课的ID生成器代码讲起

上两节课中，我们把一份非常简单的ID生成器的代码，从“能用”重构成了“好用”。最终给出的代码看似已经很完美了，但是如果我们再用心推敲一下，代码中关于出错处理的方式，还有进一步优化的空间，值得我们拿出来再讨论一下。

为了方便你查看，我将上节课的代码拷贝到了这里。

```
public class RandomIdGenerator implements IdGenerator {
  private static final Logger logger = LoggerFactory.getLogger(RandomIdGenerator.class);

  @Override
  public String generate() {
    String substrOfHostName = getLastFiledOfHostName();
    long currentTimeMillis = System.currentTimeMillis();
    String randomString = generateRandomAlphameric(8);
    String id = String.format("%s-%d-%s",
            substrOfHostName, currentTimeMillis, randomString);
    return id;
  }

  private String getLastFiledOfHostName() {
    String substrOfHostName = null;
    try {
      String hostName = InetAddress.getLocalHost().getHostName();
      substrOfHostName = getLastSubstrSplittedByDot(hostName);
    } catch (UnknownHostException e) {
      logger.warn("Failed to get the host name.", e);
    }
    return substrOfHostName;
  }
```

```
  @VisibleForTesting
  protected String getLastSubstrSplittedByDot(String hostName) {
    String[] tokens = hostName.split("\\.");
    String substrOfHostName = tokens[tokens.length - 1];
    return substrOfHostName;
  }

  @VisibleForTesting
  protected String generateRandomAlphameric(int length) {
    char[] randomChars = new char[length];
    int count = 0;
    Random random = new Random();
    while (count < length) {
      int maxAscii = 'z';
      int randomAscii = random.nextInt(maxAscii);
      boolean isDigit= randomAscii >= '0' && randomAscii <= '9';
      boolean isUppercase= randomAscii >= 'A' && randomAscii <= 'Z';
      boolean isLowercase= randomAscii >= 'a' && randomAscii <= 'z';
      if (isDigit|| isUppercase || isLowercase) {
        randomChars[count] = (char) (randomAscii);
        ++count;
      }
    }
    return new String(randomChars);
  }
}
```

这段代码中有四个函数。针对这四个函数的出错处理方式，我总结出下面这样几个问题。

- 对于generate()函数，如果本机名获取失败，函数返回什么？这样的返回值是否合理？
- 对于getLastFiledOfHostName()函数，是否应该将UnknownHostException异常在函数内部吞掉（try-catch并打印日志）？还是应该将异常继续往上抛出？如果往上抛出的话，是直接把UnknownHostException异常原封不动地抛出，还是封装成新的异常抛出？
- 对于getLastSubstrSplittedByDot(String hostName)函数，如果hostName为NULL或者是空字符串，这个函数应该返回什么？
- 对于generateRandomAlphameric(int length)函数，如果length小于0或者等于0，这个函数应该返回什么？

对于上面这几个问题，你可以试着思考下，我先不做解答。等我们学完本节课的理论内容之后，我们下一节课再一块来分析。这一节我们重点讲解一些理论方面的知识。

函数出错应该返回啥？

关于函数出错返回数据类型，我总结了4种情况，它们分别是：错误码、NULL值、空对象、异常对象。接下来，我们就一一来看它们的用法以及适用场景。

1.返回错误码

C语言中没有异常这样的语法机制，因此，返回错误码便是最常用的出错处理方式。而在Java、Python等比较新的编程语言中，大部分情况下，我们都用异常来处理函数出错的情况，极少会用到错误码。

在C语言中，错误码的返回方式有两种：一种是直接占用函数的返回值，函数正常执行的返回值放到出参

中；另一种是将错误码定义为全局变量，在函数执行出错时，函数调用者通过这个全局变量来获取错误码。针对这两种方式，我举个例子来进一步解释。具体代码如下所示：

```
// 错误码的返回方式一：pathname/flags/mode为入参；fd为出参，存储打开的文件句柄。
int open(const char *pathname, int flags, mode_t mode, int* fd) {
  if (/*文件不存在*/) {
    return EEXIST;
  }

  if (/*没有访问权限*/) {
    return EACCESS;
  }

  if (/*打开文件成功*/) {
    return SUCCESS; // C语言中的宏定义：#define SUCCESS 0
  }
  // ...
}
//使用举例
int fd;
int result = open(“c:\test.txt”, O_RDWR, S_IRWXU|S_IRWXG|S_IRWXO, &fd);
if (result == SUCCESS) {
  // 取出fd使用
} else if (result == EEXIST) {
  //...
} else if (result == EACESS) {
  //...
}

// 错误码的返回方式二：函数返回打开的文件句柄，错误码放到errno中。
int errno; // 线程安全的全局变量
int open(const char *pathname, int flags, mode_t mode) {
  if (/*文件不存在*/) {
    errno = EEXIST;
    return -1;
  }

  if (/*没有访问权限*/) {
    errno = EACCESS;
    return -1;
  }

  // ...
}
// 使用举例
int hFile = open(“c:\test.txt”, O_RDWR, S_IRWXU|S_IRWXG|S_IRWXO);
if (-1 == hFile) {
  printf("Failed to open file, error no: %d.\n", errno);
  if (errno == EEXIST ) {
    // ...
  } else if(errno == EACCESS) {
    // ...
  }
  // ...
}
```

实际上，如果你熟悉的编程语言中有异常这种语法机制，那就尽量不要使用错误码。异常相对于错误码，有诸多方面的优势，比如可以携带更多的错误信息（exception中可以有message、stack trace等信息）等。

关于异常，我们待会还会非常详细地讲解。

2.返回NULL值

在多数编程语言中，我们用NULL来表示“不存在”这种语义。不过，网上很多人不建议函数返回NULL值，认为这是一种不好的设计思路，主要的理由有以下两个。

- 如果某个函数有可能返回NULL值，我们在使用它的时候，忘记了做NULL值判断，就有可能会抛出**空指针异常**（Null Pointer Exception，缩写为NPE）。
- 如果我们定义了很多返回值可能为NULL的函数，那代码中就会充斥着大量的NULL值判断逻辑，一方面写起来比较繁琐，另一方面它们跟正常的业务逻辑耦合在一起，会影响代码的可读性。

我举个例子解释一下，具体代码如下所示：

```
public class UserService {
  private UserRepo userRepo; // 依赖注入

  public User getUser(String telephone) {
    // 如果用户不存在，则返回null
    return null;
  }
}

// 使用函数getUser()
User user = userService.getUser("18917718965");
if (user != null) { // 做NULL值判断，否则有可能会报NPE
  String email = user.getEmail();
  if (email != null) { // 做NULL值判断，否则有可能会报NPE
    String escapedEmail = email.replaceAll("@", "#");
  }
}
```

那我们是否可以用异常来替代NULL值，在查找用户不存在的时候，让函数抛出UserNotFoundException异常呢？

我个人觉得，尽管返回NULL值有诸多弊端，但对于以get、find、select、search、query等单词开头的查找函数来说，数据不存在，并非一种异常情况，这是一种正常行为。所以，返回代表不存在语义的NULL值比返回异常更加合理。

不过，话说回来，刚刚讲的这个理由，也并不是特别有说服力。对于查找数据不存在的情况，函数到底是该用NULL值还是异常，有一个比较重要的参考标准是，看项目中的其他类似查找函数都是如何定义的，只要整个项目遵从统一的约定即可。如果项目从零开始开发，并没有统一约定和可以参考的代码，那你选择两者中的任何一种都可以。你只需要在函数定义的地方注释清楚，让调用者清晰地知道数据不存在的时候会返回什么就可以了。

再补充说明一点，对于查找函数来说，除了返回数据对象之外，有的还会返回下标位置，比如Java中的indexOf()函数，用来实现在某个字符串中查找另一个子串第一次出现的位置。函数的返回值类型为基本类型int。这个时候，我们就无法用NULL值来表示不存在的情况了。对于这种情况，我们有两种处理思路，一种是返回NotFoundException，一种是返回一个特殊值，比如-1。不过，显然-1更加合理，理由也是同样

的，也就是说“没有查找到”是一种正常而非异常的行为。

3.返回空对象

刚刚我们讲到，返回NULL值有各种弊端。应对这个问题有一个比较经典的策略，那就是应用空对象设计模式（Null Object Design Pattern）。关于这个设计模式，我们在后面章节会详细讲，现在就不展开来讲解了。不过，我们今天来讲两种比较简单、比较特殊的空对象，那就是**空字符串**和**空集合**。

当函数返回的数据是字符串类型或者集合类型的时候，我们可以用空字符串或空集合替代NULL值，来表示不存在的情况。这样，我们在使用函数的时候，就可以不用做NULL值判断。我举个例子来解释下。具体代码如下所示：

```
// 使用空集合替代NULL
public class UserService {
  private UserRepo userRepo; // 依赖注入

  public List<User> getUsers(String telephonePrefix) {
   // 没有查找到数据
    return Collectiosn.emptyList();
  }
}
// getUsers使用示例
List<User> users = userService.getUsers("189");
for (User user : users) { //这里不需要做NULL值判断
  // ...
}

// 使用空字符串替代NULL
public String retrieveUppercaseLetters(String text) {
  // 如果text中没有大写字母，返回空字符串，而非NULL值
  return "";
}
// retrieveUppercaseLetters()使用举例
String uppercaseLetters = retrieveUppercaseLetters("wangzheng");
int length = uppercaseLetters.length();// 不需要做NULL值判断
System.out.println("Contains " + length + " upper case letters.");
```

4.抛出异常对象

尽管前面讲了很多函数出错的返回数据类型，但是，最常用的函数出错处理方式就是抛出异常。异常可以携带更多的错误信息，比如函数调用栈信息。除此之外，异常可以将正常逻辑和异常逻辑的处理分离开来，这样代码的可读性就会更好。

不同的编程语言的异常语法稍有不同。像C++和大部分的动态语言（Python、Ruby、JavaScript等）都只定义了一种异常类型：运行时异常（Runtime Exception）。而像Java，除了运行时异常外，还定义了另外一种异常类型：编译时异常（Compile Exception）。

对于运行时异常，我们在编写代码的时候，可以不用主动去try-catch，编译器在编译代码的时候，并不会检查代码是否有对运行时异常做了处理。相反，对于编译时异常，我们在编写代码的时候，需要主动去try-catch或者在函数定义中声明，否则编译就会报错。所以，运行时异常也叫作非受检异常（Unchecked Exception），编译时异常也叫作受检异常（Checked Exception）。

如果你熟悉的编程语言中，只定义了一种异常类型，那用起来反倒比较简单。如果你熟悉的编程语言中（比如Java），定义了两种异常类型，那在异常出现的时候，我们应该选择抛出哪种异常类型呢？是受检异常还是非受检异常？

对于代码bug（比如数组越界）以及不可恢复异常（比如数据库连接失败），即便我们捕获了，也做不了太多事情，所以，我们倾向于使用非受检异常。对于可恢复异常、业务异常，比如提现金额大于余额的异常，我们更倾向于使用受检异常，明确告知调用者需要捕获处理。

我举一个例子解释一下，代码如下所示。当Redis的地址（参数address）没有设置的时候，我们直接使用默认的地址（比如本地地址和默认端口）；当Redis的地址格式不正确的时候，我们希望程序能fail-fast，也就是说，把这种情况当成不可恢复的异常，直接抛出运行时异常，将程序终止掉。

```
// address格式: "192.131.2.33:7896"
public void parseRedisAddress(String address) {
  this.host = RedisConfig.DEFAULT_HOST;
  this.port = RedisConfig.DEFAULT_PORT;

  if (StringUtils.isBlank(address)) {
    return;
  }

  String[] ipAndPort = address.split(":");
  if (ipAndPort.length != 2) {
    throw new RuntimeException("...");
  }

  this.host = ipAndPort[0];
  // parseInt()解析失败会抛出NumberFormatException运行时异常
  this.port = Integer.parseInt(ipAndPort[1]);
}
```

实际上，Java支持的受检异常一直被人诟病，很多人主张所有的异常情况都应该使用非受检异常。支持这种观点的理由主要有以下三个。

- 受检异常需要显式地在函数定义中声明。如果函数会抛出很多受检异常，那函数的定义就会非常冗长，这就会影响代码的可读性，使用起来也不方便。
- 编译器强制我们必须显示地捕获所有的受检异常，代码实现会比较繁琐。而非受检异常正好相反，我们不需要在定义中显示声明，并且是否需要捕获处理，也可以自由决定。
- 受检异常的使用违反开闭原则。如果我们给某个函数新增一个受检异常，这个函数所在的函数调用链上的所有位于其之上的函数都需要做相应的代码修改，直到调用链中的某个函数将这个新增的异常try-catch处理掉为止。而新增非受检异常可以不改动调用链上的代码。我们可以灵活地选择在某个函数中集中处理，比如在Spring中的AOP切面中集中处理异常。

不过，非受检异常也有弊端，它的优点其实也正是它的缺点。从刚刚的表述中，我们可以看出，非受检异常使用起来更加灵活，怎么处理的主动权这里就交给了程序员。我们前面也讲到，过于灵活会带来不可控，非受检异常不需要显式地在函数定义中声明，那我们在使用函数的时候，就需要查看代码才能知道具体会抛出哪些异常。非受检异常不需要强制捕获处理，那程序员就有可能漏掉一些本应该捕获处理的异常。

对于应该用受检异常还是非受检异常，网上的争论有很多，但并没有一个非常强有力的理由能够说明一个就一定比另一个更好。所以，我们只需要根据团队的开发习惯，在同一个项目中，制定统一的异常处理规范即可。

刚刚我们讲了两种异常类型，现在我们再来讲下，如何处理函数抛出的异常？总结一下，一般有下面三种处理方法。

- 直接吞掉。具体的代码示例如下所示：

```
public void func1() throws Exception1 {
  // ...
}

public void func2() {
  //...
  try {
    func1();
  } catch(Exception1 e) {
    log.warn("...", e); //吐掉：try-catch打印日志
  }
  //...
}
```

- 原封不动地re-throw。具体的代码示例如下所示：

```
public void func1() throws Exception1 {
  // ...
}


public void func2() throws Exception1 {//原封不动的re-throw Exception1
  //...
  func1();
  //...
}
```

- 包装成新的异常re-throw。具体的代码示例如下所示：

```
public void func1() throws Exception1 {
  // ...
}


public void func2() throws Exception2 {
  //...
  try {
    func1();
  } catch(Exception1 e) {
   throw new Exception2("...", e); // wrap成新的Exception2然后re-throw
  }
```

```
  //...
}
```

当我们面对函数抛出异常的时候，应该选择上面的哪种处理方式呢？我总结了下面三个参考原则：

- 如果func1()抛出的异常是可以恢复，且func2()的调用方并不关心此异常，我们完全可以在func2()内将func1()抛出的异常吞掉；
- 如果func1()抛出的异常对func2()的调用方来说，也是可以理解的、关心的 ，并且在业务概念上有一定的相关性，我们可以选择直接将func1抛出的异常re-throw；
- 如果func1()抛出的异常太底层，对func2()的调用方来说，缺乏背景去理解、且业务概念上无关，我们可以将它重新包装成调用方可以理解的新异常，然后re-throw。

总之，是否往上继续抛出，要看上层代码是否关心这个异常。关心就将它抛出，否则就直接吞掉。是否需要包装成新的异常抛出，看上层代码是否能理解这个异常、是否业务相关。如果能理解、业务相关就可以直接抛出，否则就封装成新的异常抛出。关于这部分理论知识，我们在下一节课中，会结合ID生成器的代码来进一步讲解。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要掌握的重点内容。

对于函数出错返回数据类型，我总结了4种情况，它们分别是：错误码、NULL值、空对象、异常对象。

1.返回错误码

C语言没有异常这样的语法机制，返回错误码便是最常用的出错处理方式。而Java、Python等比较新的编程语言中，大部分情况下，我们都用异常来处理函数出错的情况，极少会用到错误码。

2.返回NULL值

在多数编程语言中，我们用NULL来表示“不存在”这种语义。对于查找函数来说，数据不存在并非一种异常情况，是一种正常行为，所以返回表示不存在语义的NULL值比返回异常更加合理。

3.返回空对象

返回NULL值有各种弊端，对此有一个比较经典的应对策略，那就是应用空对象设计模式。当函数返回的数据是字符串类型或者集合类型的时候，我们可以用空字符串或空集合替代NULL值，来表示不存在的情况。这样，我们在使用函数的时候，就可以不用做NULL值判断。

4.抛出异常对象

尽管前面讲了很多函数出错的返回数据类型，但是，最常用的函数出错处理方式是抛出异常。异常有两种类型：受检异常和非受检异常。

对于应该用受检异常还是非受检异常，网上的争论有很多，但也并没有一个非常强有力的理由，说明一个就

一定比另一个更好。所以，我们只需要根据团队的开发习惯，在同一个项目中，制定统一的异常处理规范即可。

对于函数抛出的异常，我们有三种处理方法：直接吞掉、直接往上抛出、包裹成新的异常抛出。这一部分我们留在下一节课中结合实战进一步讲解。

课堂讨论

结合我们今天学的理论知识，试着回答一下在文章开头针对RandomIdGenerator提到的四个问题。

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言：

- Jxin 2020-01-24 11:10:15
  回答问题
  1.抛出异常，因为服务器获取不到host是一种异常情况，并且打印的异常日志不能是warm,而是err，因为该异常不会自动回复。

  2.往上抛，原封不动。应该在api统一出口处处理异常，这样异常代码会比较聚合（个人习惯）。该异常描述已经很准确，且处理异常依旧在genId接口中，所以上层函数可以认识该异常，所以原封不动。（而统一出口函数，则可以抛自定义异常，以收敛api使用方的考虑范围）。

  3.抛出异常，null值裁剪名称是一种异常情况。或则说，对于裁剪名称这个函数，入参不能为null。

  4.返回空字符串。小于等于0说明不需要带随机后缀，这也是一个正常的业务场景。返回空字符串是为了方便调用方不用做null判断。

  分歧：
  1.get,find,select等dao层操作，返回null是正常业务情况，表示数据不存在。但在其应用层，数据不存在可能意味着有脏数据，数据缺失等情况，属于异常情况，需要抛出异常。所以同样是get方法，持久层返回null，业务层返回可能是异常。

  2.异常流开销大，在对响应时间要求很严格的场景。放弃合理的异常处理，采用不合理的特殊返回值的方式也是合理的。所以合理的运用异常流在java也是一个选择项。在可读和性能我们需要权衡，而这两玩意经常是相驳的。

  最后：
  祝栏主和同学们新年快乐！

- test 2020-01-24 10:51:01
  1. 获取不到本机名，抛出异常
  2. 往上跑，可以包装为新的异常
  3. 返回null值
  4. 返回空对象

- 南山 2020-01-24 10:50:37
  从团队的实践来看，异常统一只靠人为约定是比较难实行的，团队成员理解不一样，实际写代码时候各种原因不按约定来。通过插件，或者IDE自动监测的手段会比较好，比如sonar。

- 黄林晴 2020-01-24 08:43:04
  打卡✔
  新年快乐

- 阳光 2020-01-24 06:48:52
  打卡

- sunnywhy 2020-01-24 05:52:24
  第二种返回Null的情况，可以使用Optional吗