Sérgio Taborda's Weblog

Alguns ensinam. Alguns fazem. O resto procura nos livros.

Type text to search here...

Exceções: Boas Práticas, Más Práticas

Deixe o seu comentário Go to comments

O problema

O mecanismo de exceções em Java criou o conceito de exceção verificada. Este conceito implica que qualquer método que chame outro que lance uma exceção verificada é obrigado, pelo compilador, a dar um tratamento à exceção. O compilador Java é muito bom mas não chega a ser inteligente ao ponto de saber se o tratamento que você deu à exceção é realmente uma solução ou uma enrolação.

Muitas vezes se torna chato ter que trabalhar a todo o momento com exceções verificadas. É usual ter que trabalhar com `SQLException` ou `IOException` que são muito comuns e pouco especificas. Frente a exceções como estas rapidamente o programador desiste de tratar tantas exceções e aí começa o problema. Por convicção, inexperiência ou omissão, o programador acaba desconsiderando a exceção que aconteceu muitas vezes fazendo a exceção, simplesmente, desaparecer. Obviamente isso é atirar pela janela um dos mecanismos mais importantes e avançados do Java: o tratamento de exceções verificadas.

No artigo anterior desta série vimos como os conceitos de exceção e tratamento de exceção foram incorporados na linguagem Java. Veremos neste artigo um conjunto de regras para tirar o máximo proveito desse mecanismo, sem que ele se torne omnipresente e chato.

O quê , onde e porquê

Uma exceção é um evento que ocorre q sim, a exceção carrega consigo informações sobre qual é o probl e se possível, porque o problema aconteceu. Quanto melhor a exce mais útil ela será.

O quê

O que deu errado é transmitido no nome exceção. Por exemplo, `OutofMemoryError` significa que não há m `matException` significa que o formato do numero é incorreto; que o arquivo não foi encontrado. O nome da exceção é muitc lema em si mesmo. É comum deixar a palavra Exception (ou Er ar claro que esta classe será trata de forma especial pelo compilad

O Porquê

A razão do problema é explicada na mensagem contida na própria exceção e que pode ser obtida com `getMessage()`. Essa explicação é textual e normalmente em inglês. É possível internacionalizar a mensagem, mas isso só é feito em casos em que a mensagem tem que ser apresentada ao usuário final.

O Onde

O onde deu errado é respondido pelo *stack trace* ,o rastro, que revela em qual linha de qual classe o problema aconteceu.O *stack trace* mostra a hierarquia de todas as exceções que derivaram da exceção original. Caso seja necessário podemos navegar pelo *stack trace* para saber se um certo tipo de exceção aconteceu, ou simplesmente qual foi a exceção que originou o problema. Normalmente a ocorrência da exceção é acompanhada do numero da linha e do nome da classe onde ela foi lançada. Isso é uma informação vital, extremamente útil quando queremos saber onde a exceção aconteceu.

Taxionomia de uma Exceção

A classe `Throwable` é, em Java, a classe mãe de todas as exceções. Ela oferece alguns métodos que nos permitem conhecer um pouco mais do problema que aconteceu. Eis os mais relevantes:

```
public class Throwable {
public Throwable(String message, Throwable cause); // construtor
public Throwable getCause(); // outro Throwable que foi a causa deste
public String getMessage(); // mensagem relativa à causa
public void printStackTrace(PrintStream s);
public void printStackTrace(); … }
```

Código 1: Métodos básicos da classe `Throwable`

O construtor será importante quando criarmos nossa própria classe de exceção. O método `getCause()` retorna a exceção, se alguma, que deu origem a esta. Este método retorna a exceção passada no construtor. Isto é importante quando encapsulamentos uma exceção em outra. O método `getMessage()` retorna a mensagem passada no construtor. Os métodos `printStackTrace()` são importantes para passar a informação do rastro para uma hierarquia legível por um humano. São extremamente úteis para encontrar o problema, mas normalmente usados no lugar errado.

Tratando de Exceções

O tratamento de exceções tem algumas regras impostas pelo Java:

1. Todas as exceções devem ser manipuladas (handled). Manipular, aqui, significa fazer uma de duas coisas: capturar a exceção com try-catch ou declarar a exceção com throws
2. Métodos que sobe-escrevem outros métodos não podem declarar o lançamento de mais exceções que o método que está sendo sobre-escrito (métodos filhos não podem lançar mais exceções que os pais)
3. O compilador Java verificará a conformidade do código com as duas regras acima sempre que a exceção for do tipo verificada

Manipular a exceção não significa necessariamente tratá-la. Para tratarmos uma exceção temos que entender a resposta a cada uma das seguintes perguntas:

- Aconteceu uma exceção?
- O que podemos fazer para resolver o problema?
- Como indicar que não conseguimos resolver o problema?

Como descobrir se aconteceu uma exceção?

Quando uma exceção acontece, ela acontece porque o método atual a lançou ou porque um outro método chamado a lançou. Se o método atual a lançou é porque ele não consegue resolver o problema. Neste caso passamos a exceção ao método superior. Se a exceção veio de um outro método temos que saber, com antecedência, se a podemos tratar.

Para isso é importante consultar a documentação do métodos que o nosso método está chamando. Nessa documentação devemos encontrar a lista de exceções lançadas assim como explicações sobre as condições em que elas acontecem. Analisando a documentação poderemos ter uma idéia se podemos ou não resolver o problema. Algumas vezes a documentação é escassa ou mal construída e não nos informa o suficiente. Em casos poderemos fazer pequenos testes tentando forçar exceções e teriam algumas conclusões simples. Contudo, o normal, é aceitar que não sabemos tratar as exceções vindas do método. Se não podermos resolver o problema faremos o mesmo que antes: passar a exceção ao método anterior.

O que podemos fazer para resolver o problema?

Cada caso é um caso e não há nada como a experiência para nos guiar. O que podemos fazer é muito diversificado. Se sabemos que podemos resolver a exceção é porque sabemos o que fazer para a resolver. Por exemplo, se um arquivo não for encontrado e obtivermos uma `FileNotFoundException` podemos criar um vazio, buscá-lo em outra localização ou simplesmente abortar todo o processo.

Como indicar que não conseguimos resolver o problema?

Se ao tentar resolver um problema encontramos um outro problema, ou não conseguimos resolver o problema original então temos que indicar isso ao método chamador. A forma de indicar isso é lançando uma exceção. Podemos relançar a exceção original que tentámos – em vão – tratar ou enviar a exceção que apareceu depois. No caso de enviarmos a segunda exceção a primeira deve pertencer ao seu rastro (stack trace). O ideal é retornar a primeira exceção se não conseguirmos resolver a segunda.

Boas práticas de tratamento de exceções

Porque o tratamento de exceções é muitas vezes frustrante o programador tende a se livrar do problema a qualquer custo. Obviamente esta não é a forma certa de lidar com exceções. Eis algumas das diretivas que sempre deve seguir quando estiver diante de uma exceção a ser tratada.

Não capture o que você não pode segurar

Quando um método que você está invocando lançar uma exceção, se você não sabe o que fazer com ela, simplesmente não faça nada. Deixe para quem entende. Se todos os métodos aplicarem esta regra, você não precisa ficar enchendo seu código com tratamentos imprestáveis e chatos. Mas atenção, também não seja irresponsável achando que sempre existirá um outro método que cuide de seu problema. O método tem que conhecer o seu lugar no sistema. Se o método não tem a quem passar a batata quente então não lhe resta alternativa senão tratar o problema, ou pelo menos, reportar que o problema aconteceu. A importância desta regra é minimizar o código que trata exceções impossíveis de resolver, centralizando ações necessárias nesses casos, como apresentar uma mensagem ao usuário, reportar o evento para um registro (log) ou educadamente terminar a aplicação.

Cumprir esta regra não é tão simples quanto parece. A forma fácil é simplesmente declarar que o método chamador também lança a exceção do método chamado. No exemplo, deixámos `leArquivo()` lançar `IOException` porque `read()` lança essa exceção e não sabemos como tratá-la. Contudo, se este método fizesse parte de uma interface que não pode lançar `IOException` teriamos que excapsular esse tipo de exceção em outra que a interface define. Se não define nenhuma teriamos que encapsulá-la numa classe derivada de `RuntimeException`.

```
public void leArquivo(File arquivo) throws IOException{

read(new FileInputStream(arquivo)) ; // chama função auxiliar lança uma IOException genérica

}
```

Código 2: Declare as exceções que não pode tratar com `throws`

Seja especifico

Quando você tiver que lançar uma exceção, seja especifico. Não lance exceções genéricas que significam tudo e nada ao mesmo tempo. Não lance diretamente `Exception` ou `RuntimeException`. Use a exceção que melhor detalha o problema. Se nenhuma existir, crie a sua própria classe de exceção que seja especifica o bastante. Apenas especifica o bastante pois não é bom ser especifica demais. Por

exemplo, não pense em lançar uma exceção se o problema aconteceu antes do almoço e outra se aconteceu depois do almoço.Esse tipo de informação será inutil se a exceção não for registrada para porterior consulta, mas se for, o próprio mecanismo de registro poderá injetar essa informação. Olhando o nosso exemplo, podemos ser mais específicos:

```
public void leArquivo(File arquivo) throws IllegalArgumentException, MyIOException{
if (arquivo == null){
throw new IllegalArgumentException("parametro arquivo não pode ser nulo");
} else if (!arquivo.exists()){ // o arquivo não existe
throw new MyFileNotFoundException(arquivo); // criará a mensagem a partir dos dados de arquivo
} else if (!arquivo.isFile()){ // não é um arquivo
throw new MyNotAFileException(arquivo); // criará a mensagem a partir dos dados de arquivo
}
// chama função auxiliar
try {
read(new FileInputStream(arquivo)); // este método lança IOException genérica
} catch (FileNotFoundException e) {
//já verificámos que o arquivo existe no if anterior, então
// se esta exceção acontecer significa que não temos privilégios para ler o arquivo
throw new NoReadPriviledgeException()
} cacth (IOException e){
// no caso genérico não ha muito a fazer, a não ser, possivelmente, encapsular a exceção
throw new MyIOException(e);
}
}
```

Código 3: Como fazer – seja especifico

`MyIOException` é uma `RuntimeException` com a função especifica de encapsular `IOException` Usámos `IllegalArgumentException` porque é uma exceção que já existe (sempre dê preferencia a usar uma exceção da API padrão) e que especifica o problema o bastante. Todas as outras exceções são filhas mais especificas de `MyIOException`.

A importância de ser especifico é aumentar o detalhe sobre o problema, para que seja mais fácil identificá-lo e resolvê-lo independentemente de outros. Veremos mais tarde como criar sua própria classe de exceção.

Não deixe para os outros o que você pode lançar primeiro

Detalhe exatamente o que o seu método faz e o que o impediria de completar essa tarefa. Se alguma das condições de impedimento está presente lance *imediatamente* uma exceção explicando porque o método não pode fazer o seu trabalho. Não espere até que um outro método auxiliar que você vai usar lance uma exceção incompreensível quando você pode lançar uma muito mais detalhada. No exemplo anterior o código testa primeiro um conjunto de condições e apenas se tudo estiver bem que o codigo principal é executado.

```
public void leArquivo(File arquivo) throws IOException{

if (arquivo == null){

throw new IllegalArgumentException("parametro arquivo não pode ser nulo");

} else if (!arquivo.exists()){ // o arquivo não existe

throw new MyFileNotFoundException(arquivo); // criará a mensagem a partir dos dados de arquivo

} else if (!arquivo.isFile()){ // não é um arquivo

throw new MyNotAFileException(arquivo); // criará a mensagem a partir dos dados de arquivo

}

…

}
```

Código 4: Como fazer – não deixe para os outros o que você pode lançar primeiro

A importância desta regra é deixar o rastro o mais curto possível para que seja mais facíl saber onde, e porquê, o problema aconteceu. Quanto mais cedo você lançar a exceção, mais perto fica o onde, e mais claro fica o porquê. Nem sempe as exceções acontecem porque o método fez algo errado. Muitas vezes acontecem porque o método recebeu os parâmetros errados. Testar as pré-condições do método é tão (ou mais) importante que o método em si mesmo.

Más práticas

Baseados nas idéias anteriores vamos revisar algums códigos comuns que infelizmente são más práticas,ou seja, nunca faça isto.

Exceções e loops

Quando executamos um código que pode lançar exceções em um loop (for, while , do) temos que ter atenção se a exceção invalida todo o loop ou apenas uma iteração. Se soubermos que o problema apenas invalida uma iteração deveremos roedar o codigo com o `try-catch` e deixar as demais iterações continua (se necessário memorizando as exceções encontradas numa coleção para que possam ser tratadas depois que o loop acabar).

Log e lança

É um erro comum contruir código semelhante a algum destes:

```
catch (SQLException e) {
LOG.error("Blablabla", e); throw e;
}
// ou
catch (IOException e) {
LOG.error("Blablabla", e);
throw new MinhaException("Blablabla outra vez", e);
} // ou
catch (Exception e) {
e.printStackTrace();
throw new MinhaException("Blablabla", e);
}
```

Código 5: Log e lança

Provavelmente essa exceção relaçada será enviada para o log (registro) em outro ponto do sistema, então não há razão para fazer o log aqui. Muito menos imprimir o rastro no console. Afinal, o console pode nem ser visível neste ponto.

Lançar Exception

Código com esta assinatura:

```
public void algumMetodo() throws Exception
```

Código 6: Lançando `Exception`

Significa não dar nenhuma informação ao método chamador do tipo de esceções que podem acontecer. Básicamente isto significa: “posso lançar qualquer coisa”. Isso viola a regra de ser especifico. Em opção deve-se explicitar qual ou quais as exceções que são lançadas. Mas cuidado com lançar a casa pela janela.

Lançando a casa pela janela

Código com este tipo de assinatura:

```
public void algumMetodo() throws EstaException, AquelaException, UmaOutraException,
OutraPossivelException, EAindaMaisUmaException {
```

Código 7: Lançando `Exception`

Onde várias exceções podem acontecer e todas são registradas. Isto é especifico, mas polui a interface do método além de ser um pesadelo para o método que chamar este. A opção é usar um conjunto menor, e mais especifico, de exceções que encapsulem aquelas. Exceções não verificadas podem ser declaradas, mas o ideal é só declará-las na documentação javadoc.

Retenção da Exceção

Reter a exceção significa que após a capturar ela não é relançada, em nenhuma forma. Simplesmente se sequestra a exceção sem nunca a lançar de volta ou dar um tratamento apropriado. Não capture o que não pode tratar, diz a regra. Enviar para o log, não é tratar. Imprimir o *stack trace* não é tratar.

```
try { algumMetodo(); } catch (Exception e) { LOG.error("metodo falhou", e); }
```

Código 8: Retenção errada de `Exception`

Apenas existe uma ocasião onde você pode sequestrar a exceção: quando chegou no nivel mais próximo ao usuário. Por exemplo, se a aplicação será abortada logo a seguir ou uma mensagem será mostrada na interface do usuário sem abortar a aplicação. Apenas destes casos você pode sequestrar a exceção.

```
try { algumMetodo(); } catch (Exception e) {
LOG.error("metodo falhou", e); System.exit(–1); }
//ou
try { algumMetodo(); }
catch (Exception e) {
LOG.error("metodo falhou", e);
UI.showErrorMessage(e);
}
```

Código 9: Retenção correta de `Exception`

É importante entender que nestas ocasiões você não está driblando as regras de tratamento. No primeiro caso você está admitindo a derrota e terminando a sua aplicação porque não foi possível tratar o problema em nenhum estágio do sistema. No segundo caso você está informando o usuário e portanto dando-lhe a hipotese de tratar o problemas por outros meios alheios à sua aplicação. Isto é válido. Pense por exemplo numa aplicação que acessa a internet e avisa o usuário quando a conexção não foi possível.

Eliminação do rastro da Exceção

Ao encapsular a exceção em outra, despresa-se o rastro invocando e.getMessage() em vez de passar no construtor a causa da exceção que estamos criando. É importante manter o rastro pois é nele que se inclui a informação de onde e porquê o problema aconteceu.

```
try { algumMetodo(); } catch (Exception e) {

throw new MinhaException("Blablalbla: " + e.getMessage());

}
```

Código 10: Eliminação do rastro de `Exception`

Contudo existem casos onde é necessário eliminar o rastro. Por exemplo, quando estamos encapsulando uma exceção não serializável numa que é serializável. Nesse caso a única forma de manter informação relativa ao problema é usando a mensagem da exceção original, que se tentarmos colocar a exceção original no rastro causaremos uma outra exceção na hora de serializar a exceção. Este mecanismo é importante principalmente em sistema distribuídos.

Criando suas próprias exceções

Agora já sabe como tratar execções. Se esteve com atenção deve ter notado que muitas das vezes a forma de tratar uma exceção é encapsulá-la em uma outra. Normalmente numa criada por si.

Preciso mesmo de criar uma classe de exceção?

A primeira coisa importante a fazer na hora de criar a sua própria exceção é perguntar-se se precisa mesmo criar uma exceção própria. Eis alguns casos em que não é proveitoso criar a sua própria exceção:

- já existe uma exceção nas bibliotecas-padrão que representam o problema que quer tratar.
- A sua exceção não vai adicionar nenhuma informação suplementar é caraterização do problema.

Eis alguns casos onde você terá que criar a sua própria exceção:

- já existe uma exceção nas bibliotecas-padrão que representam o problema que quer tratar, mas é verificada e você precisa de uma exceção não-verificada
- A sua exceção vai adicionar informação suplementar é caraterização do problema para enriquicer possíveis métodos de tratamento, e/ou registros.

Verificada ou não verficiada, eis o dilema

Quanto tiver que decidir entre usar uma exceção verificada e uma não verifica não se pergunte:"qual ação o chamador pode tomar para resolver este problema?" e sim "o programador tem *sempre* que saber que este problema existe?". Se a resposta for afirmativa crie uma exceção verificada.

Lembre-se que exceções verificadas são especialmente uteis quando o seu sistema conversa com outros fora do seu ambiente como sistemas de arquivos, recursos remotos , outros sistema como bancos de dados, ou sistemas embutidos no seu, como plugins. Se você está construindo um framework é provavel que precise criar exceções verificadas. Se é apenas um método nas suas classes de negocio, não. Se estiver construindo um framework pense também se a exceção contém informação e com que frequência o programador poderá resolver o problema. Se necessário alterne entre exceções não verificadas e erros.

Camadas e exceções

Uma das maiores dificuldades ao lidar com exceções é decidir como elas se propagarão entre as camadas, andares e nodos do sistema. O bom *design* de camadas resulta num conjunto de interfaces que o cliente da camada usurá para invocar as suas funcionalidades. Essas interfaces devem apenas lançar exceções especificas da camada. Um bom exemplo desta arquitectura são as API de IO e de JDBC. Elas apenas lançam exceções decorrentes da sua actividade. Como vimos não é bom lançar apenas um tipo de exceção, mas estas API demonstram o principio de encapsular todas as exceções que acontecem dentro da camada, como exceções próprias da camada. Além disso são API para comunicar com outros sistemas, logo suas exceções devem ser verificadas. O cliente da camada deverá analizar a exceção que recebe, e traduzir essas exceção para um dos tipos de exceção da sua camada. Isto só deverá ser feito se a camada não sabe como resolver o problema apresentado pela camada inferior.

Empacotamento e visibilidade

Ao seguir a regra anterior temos normalmente de criar uma hierarquia de exceções na camada. As classes dessa camada fazem parte de um pacote expecifico. As exceções da camada devem estar nesse mesmo pacote. As exceções têm que ser públicas para que possam ser capturadas por outras partes do sistema contidas em outros pacotes. Tente,tanto quanto possível, reduzir a visibilidade dos construtores para o nivel de pacote. Esta regra é especialmente válida se a exceção carrega informações que podem ser apenas obtidas dentro da camada. Esta regra não se aplica a exceções que fazem parte de uma API genérica ou extensivel. A API de IO é de uso genérico e a API JDBC é extensivel, por isso elas apresentam exceções publicas com contrutores publicos. Mas na sua aplicação as exceções de negócio devem estar fortemente acopladas com as regras de negócio que estão implementadas na camada de negócio. Nesse caso não faz sentido deixar que outras camadas criem e lancem exceções de negócio e como tal seus construtores podem ser de nivel de pacote. A visibilidade do contrutor não é um detalhe crítico da implementação de exceções, mas pode ser uma ferramenta para simplificar o seu uso, o seu escopo e o seu entendimento. É mais fácil tornar um método publico do que torná-lo protegido.

Resumo

Não é dificil ter um bom tratamento de exceções seguindo algumas directivas. Muitas vezes é necessário criar seu proprio pacotes de exceções sobretudo se está constuindo frameworks utilitarios ou implementando camadas mais baixas. Veremos num próximo artigo um conjunto de padrões e classes que o auxiliarão a fazer o correto tratamento de todas as exceções do seu sistema.

Referências

[1] **Three Rules for Effective Exception Handling**
Jim Cuching
URL:http://today.java.net/pub/a/today/2003/12/04/exceptions.html

[2] **Best Practices for Exception Handling**
Gunjan Doshi
URL:http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html

[3] **About effective Exception Handling**
Dino Celovic, Nader Soukouti
URL:http://www.sanabel-solutions.com/publications/About%20Effective%20Exception%20Handling.pdf

derivadas 3.0 Genérica .

★ Gosto

Be the first to like this.

Comentários (8) Trackbacks (0) Deixe o seu comentário Trackback

1. 
Vinícius Godoy
2008/03/19 às 10:42
Responder

Olá Sérgio. Meus parabéns pelos artigos. Já estão entre os meus favoritos. Os textos são claros, limpos, diretos e precisos. Perfeito.

Só dois comentários: Acho que existe uma "exceção" a regra "Não capture o que você não pode segurar" que acho que vale a pena ser mencionada. É, conforme explica Joshua Bloch, no Effective Java, quando você torna uma exceção específica, numa mais genérica, mais adequada a camada de abstração que você está lidando.

Por exemplo, considere o método abaixo:

```
void connect throws ConnectionFailedException {
try {
doConnection(); //Faz a conexão, mas lança várias exceções específicas
} catch ( ServerException e) {
throw new ConnectionFailedException(e);
} catch ( IOException e) {
throw new ConnectionFailedException(e);
} catch ( SocketException e) {
throw new ConnectionFailedException(e);
}
```

Nele, existe o método doConnection(), que poderia lançar diversas exceções. Mas o usuário do método connect não se importa em saber exatamente qual delas captura. Para ele, só é importante se conseguiu ou não conectar corretamente no servidor. Por isso, capturamos as três fontes possíveis de exceção (IO, Socket e o próprio Server), e transformamos numa única exceção, associando a que foi capturada como causa. Isso já garante que a sua dica de "Não jogar a casa" também seja cumprida.

O segundo comentário é que estranhei o fato de as vezes você escrever "Excepção" e as vezes "Exceção". As duas formas são usadas por aí onde você mora?

2. 
sergiotaborda
2008/03/19 às 14:14
Responder

Obrigado pela leitura
A exceção a que se refere é mencionada no item "Camadas e exceções". Talvez não tenha ficado claro pois falta um exemplo claro como o seu. No seu exemplo você está encapsulando as exceções não apenas para não atirar tudo pela janela, mas principalmente porque está no limite de um camada. Na realidade isso é a aplicação correta das regras e não uma exceção(Em

Portugal escreve-se “excepção” , no Brasil escreve-se “exceção”. Na tentiva de ser coerente tento escrever em português do Brasil mas é dificil se policiar para quem é nascido e educado em Portugal Vou revisar o texto melhor. Obrigado!)

3. 
vinigodoy
2008/03/20 às 6:39
Responder

Ops… tem razão… por algum motivo acho que pulei o tópico. É um dos problemas de ler no trabalho. Com a pressão, acaba-se lendo com pressa.

4. 
Robson
2009/01/24 às 12:41
Responder

Olá Sérgio! Quero lhe dar os meus parabéns pelo excelente artigo sobre tratamento de exceções. Ele sanou muitas dúvidas que eu tinha. Estou desenvolvendo um sistema web em camadas, eu tenho na camada de visualização JSF, e utilizo uma camada de serviço (onde eu faço validações) e uma camada de persistência. Na persistência eu não trato as exceções, ela são lançadas para a camada de serviço, que as lança para a camada de visualização. Essa abordagem está legal?

   - 
sergiotaborda
2009/01/24 às 18:15
Responder

Sim, desde que haja encapsulamento das exceções que vêm da camada. Por exemplo, a camada de serviço lançará ServiceException e a de persistencia PersistenceException, estas exceções podem ser especializadas conforme necessário.

Por exemplo; SQLException pode ser encapsulada em uma PersistanceException.

5. 
Heitor
2010/07/20 às 9:41
Responder

Olá Sérgio.Queria tirar uma dúvida.Em um webservice, como devemos tratar as exceções?Um serviço, por exemplo, que faz manipulação em um bd, pode retornar algumas exceções (checked Exception (SqlException) e unchecked Exception(IllegalArgumentException)).Como essas exceções devem ser tratadas?Devem ser lançadas para o usuário do serviço e o mesmo fornecer tratamento(mesmo sabendo que a utilização dos serviços de um webservice são independete da linguagem, como ficaria isso?)?Devem ser tratadas e um erro mais amigável deve ser retornado(acho que complicaria demais.Uma sqlException pode ter n causas, sem contar que poderia atrapalhar o usuario do serviço escondendo a real causa do problema)?

   - 

sergiotaborda
2010/07/20 às 20:09
Responder

Num webservice as exceções fazem parte do contrato tal como a assinatura dos métodos ou os objetos nessa assinatura. Portanto, elas têm que ser devidamente documentadas e pensadas com antecedencias. A camada webservice tem que ser encarada da mesma forma que a camada de GUI. Vc não apresentaria o stackstrace ao usuario, certo ? Mas sim uma mensagem amigável, certo ? então para webservice é a mesma coisa, só que o mecamismo usado ainda é o das exceptions. A diferença é que as excções vindas das entranhas do sistema são encapsuladas em exceções "amigáveis", ou seja, aquelas que estão no contrato. Em hipotese alguma exceções não-verificadas poderão ser lançadas (nada de IllegalArgumentException) porque isso pode ter implicações não triviais (sobretudo em ambiente JEE). Além do que esse tipo de exceção não faz pare de contratos. Sempre exceções verificadas, não tecnológicas (nada de SQLException) e com significado para o método, devem ser usadas. Além disso o mecanismo de mapeamento de webservices (SOAPFault) permite designar um codigo para a execção. Embora essa prática seja normalmente proibida para exceções, aqui é uma rara exceção. Contudo, na prática, me melhor definir exceções diferentes.

Ah!, claro, obviamente que antes de enviar a exceção o implementador do webservice deve logá-la (tal como seria feito numa camada GUI) para que medidas sejam tomadas.

O cliente de webservices nunca pode resolver o problema originado no serviço, logo, a exceção é apenas informativa e uma forma educada de negar o serviço.O cliente não precisa saber qual a causa verdadeira, porque ele não fará nada para a resolver (aliás, é até uma falha de segurança mostrar a causa verdadeira). Portanto, aqui o encapsulamento tem que ser ainda mais forte, não deixando vazar nada para o cliente. O padrão Exception Handler pode ajudar bastante neste caso (tal como faria na versão GUI do mesmo problema).

6. 
ViniGodoy
2012/02/08 às 18:51
Responder

Veja esse link:
http://www.javabuilding.com/academy/java-language/excecoes-boas-praticas.html

1. No trackbacks yet.

Deixar uma resposta

Escreva o seu comentário aqui...

RSS feed

Artigos recentes

- O movimento perpétuo e a energia eterna
- O fuso e a roca
- Voto Consciente

Blog no Java Buinding

Com a inauguração do JavaBuilding as minhas obervações sobre desenvolvimento de software em geral e sobre Java e Scrum em particular podem agora ser seguidas no meu novo blog Caderno Sérgio Taborda no JavaBuiliding. Este blog permance apenas para assuntos não relacionados a desenvolvimento de software.

Caderno no Javabuilding

- O Paradoxo do Inventor
- Coleções turbinadas
- Streams no Java 8 e em outras Linguagens
- Variância
- Java 8 – Prólogo
- Monads em Java
- Scala: O vencedor da batalha Java vs .Net

MiddleHeaven

- O caso de Enumerable infinito
- MiddleHeaven e Java 8
- Javadoc Disponivel
- Lista de Discussão
- Seis anos e muito para fazer
- Seguindo em frente
- No céu do meio
- Novo Conteudo
- Utilitários: Coleções aumentadas
- Nosso novo blog

Twitter

- O Paradoxo do Inventor - Como pensar grande dá mais resultado ow.ly/NiDAH 1 month ago
- Entenda mais sobre como a nova API de Stream vai mudar sua forma de programar e como ela afetou o design do java 8 ow.ly/LGYPG 2 months ago
- A variancia em java e outras linguagens ow.ly/Li320 2 months ago
- Monads em Java ow.ly/qs5Qo 1 year ago
- O vencedor da batalha Java vs .Net ow.ly/qcSCr 1 year ago

Meta

- Registar
- Iniciar sessão
- RSS dos artigos
- Feed RSS dos comentários.
- Create a free website or blog at WordPress.com.

Páginas