Módulo 7

Segurança

Lição 5

Classes de Segurança em Java

Versão 1.0 - Jan/2008

Autor
Aldwin Lee
Cheryl Lorica

Equipe
Rommel Feria
John Paul Petines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alexandre Mori
Alexis da Rocha Silva
Angelo de Oliveira
Bruno da Silva Bonfim
Denis Mitsuo Nakasaki
Emanoel Tadeu da Silva Freitas
Guilherme da Silveira Elias
Leandro Souza de Jesus
Lucas Vinícius Bibiano Thomé
Luiz Fernandes de Oliveira Junior
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Paulo Oliveira Sampaio Reis
Ronie Dotzlaw

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach e Vinícius G. Ribeiro (Especialista em Segurança)
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Cada classe Java possui um conjunto de permissões que definem as atividades que a classe está autorizada a executar. Quando um programa Java tenta executar uma operação delicada, as permissões são pesquisadas para todas as classes envolvidas: se cada classe possuir a permissão para realizar determinada operação desejada, então a autorização é concedida. Caso contrário, uma exceção é lançada e a operação falha.

Ao final desta lição, o estudante será capaz de:

- Conhecer as regras e permissões de segurança
- Entender as classes de regras, de permissão e de acesso
- Aprender detalhes sobre exceções associadas a segurança
- Construir uma classe de permissão

2. Permissões e política de segurança

Classes que compõem o núcleo da API Java possuem permissão para executar qualquer ação. Todas as outras classes, incluindo aquelas indicadas na variável *classpath*, devem explicitamente ter permissão para realizar operações sensíveis. Para a maior parte das operações, as permissões são listadas em arquivos de política, juntamente com o código fonte a que se aplicam. Usuários finais e administradores de sistema definem os parâmetros de administração da *Sandbox* nos arquivos de política de segurança.

A tecnologia Java possui as seguintes classes relevantes para política de segurança e permissões da JVM:

- *java.security.Policy*
- *java.security.CodeSource*
- *java.security.Permission* e algumas de suas sub-classes:
  - *java.security.BasicPermission*
  - *java.io.FilePermission*
  - *java.net.NetPermission*
  - *java.net.SocketPermission*
  - *java.util.PropertyPermission*
- *java.security.PermissionCollection*
- *java.security.Permissions*

3. Classes de Política de Segurança

Segurança em Java possui um recurso para especificar quais permissões devem ser aplicadas a qual código fonte. Isso é chamado de conjunto de permissões globais da política de segurança. Esse conceito é atendido pela classe *`Policy`* no pacote `java.security`. Veremos em detalhes esta classe a seguir.

- *public abstract class Policy*

Estabelecer a política de segurança para um programa Java. Incorpora um mapeamento entre os códigos fontes e as permissões de objetos de tal forma que as classes carregadas de determinados locais ou assinadas por indivíduos específicos tenham um conjunto de permissões.

A seguir vemos o construtor da classe.

- *public Policy()*

Criar uma classe de políticas de segurança. O construtor deve inicializar o objeto de acordo com as regras internas.

Existem dois outros métodos na classe *Policy*.

- *public abstract Permissions getPermissions(CodeSource cs)*

Criar um objeto que contém um conjunto de permissões que deve ser concedido às classes geradas a partir de um determinado código fonte, carregadas a partir da URL e assinada por uma chave no código fonte.

- *public abstract void refresh()*

Atualizar a política de segurança do objeto. Por exemplo, se a foi inicializada a partir de um arquivo, ler novamente o arquivo e instalar um novo objeto de política de segurança baseado na (presumivelmente alterada) informação do arquivo. Em termos programáticos, escrever uma classe *Policy* envolve a Implementação destes métodos. O padrão da classe de política de segurança é fornecido pela classe *PolicyFile* disponível no pacote *sun.security.provider*, a qual constrói permissões baseada em informações encontradas nos arquivos de política de segurança adequados.

4. Classes de Permissão

Um objeto de permissão representa um recurso do sistema, mas não concede o acesso a este. Objetos de permissão são construídos e atribuídos ao código baseado na política de segurança em vigor. Quando um objeto de permissão é concedido para qualquer código fonte, então é atribuída a permissão para acessar qualquer recurso do sistema que será representado por este objeto.

A classe *Permission* é uma classe abstrata. Então, uma sub-classe é utilizada para representar acessos específicos. Analisaremos determinados métodos dentro dessa classe para entendermos como implementar um usuário.

- *public Permission(String name)*

Construir um objeto de permissão que representa a permissão desejada.

- *public abstract boolean equals(Object o)*

Sub-classes de *Permission* são obrigadas a implementar seus próprios testes de igualdade. Muitas vezes isso é feito pela comparação do nome (e ações, se apropriado) da permissão.

- *public abstract int hashCode()*

Sub-classes de *Permission* são obrigadas a implementar seus próprios códigos *hash*. Para o controlador de acesso funcionar corretamente, o código *hash* para um dado objeto de permissão nunca deve mudar durante a execução da máquina virtual. Além disso, permissões que comparam a mesma igualdade devem retornar o mesmo código *hash*.

- *public final String getName()*

Retornar o nome que foi usado na construção dessa permissão.

- *public abstract String getActions()*

Retornar uma forma canônica das ações que foram usadas para construir essa permissão, se existirem.

- *public String toString()*

A convenção de impressão de uma permissão deve retornar entre parênteses o nome da classe, o nome da permissão e as ações envolvidas.

- *public abstract boolean implies(Permission p)*

Esse é um dos métodos chaves da classe *Permission*. Responsável por determinar se uma classe que recebeu uma permissão pode receber outra. Também é responsável pela execução dos curingas correspondentes. Entretanto, este método não não necessita realmente dos curingas; a permissão de escrita em um objeto determinado num banco de dados provavelmente implicaria na permissão de ler o objeto.

- *public PermissionCollection newPermissionCollection()*

Retornar uma coleção de permissões apropriada para instâncias seguras deste tipo de permissão. Este método retorna *null* por padrão.

- *public void checkGuard(Object o)*

Chama o gerente da segurança (definido pelo atributo *SecurityManager.checkPermission*) para verificar se a permissão a esta variável foi concedida, gerando um exceção do tipo *SecurityException* em caso negativo. Atualmente o parâmetro deste método não é utilizado.

A seguir estão algumas sub-classes de *Permission* implementadas na API Java:

- *java.security.BasicPermission*
- *java.io.FilePermission*
- *java.net.NetPermission*
- *java.net.SocketPermission*
- *java.util.PropertyPermission*

Diferentes classes *Permission* podem ser agrupadas usando a *PermissionCollection* ou

Permissions. *PermissionCollection* representa uma coleção homogênea de objetos do tipo *Permission* ou seja, armazena apenas uma sub-classe específica da classe *Permission*. *Permissions* por outro lado, representam uma coleção heterogênea de objetos do tipo *Permission*.

5. Controle de acesso

O controle de acesso é feito pelas classes *ProtectionDomain*, *AccessController*, *SecureClassLoader*, e *URLClassLoader*.

A classe *sun.misc.Launcher* é implementada pela *Sun Microsystems* e provê uma classe de carregamento de extensões para a plataforma Java (*jre/lib/ext*) e classes de aplicação, tais como classes na variável *CLASSPATH*, especificadas usando a variável *java.class.path* ou através da opção -cp no comando de compilação Java.

A classe *ProtectionDomain* encapsula as características de um domínio que inclui um conjunto de classes e objetos correspondentes que recebem as mesmas permissões.

Uma proteção de domínio é criada, ou vinculada através da *SecureClassLoader* se uma proteção de domínio apropriada já existir, usando o método *getProtectionDomain*. O objeto *Policy* é usado para determinar as permissões associadas à proteção de domínio da classe.

O *AccessController* decide se o acesso aos recursos do sistema é permitido, com base na política de segurança em vigor. Se for negado, uma exceção do tipo *AccessControllException* é lançada. *AccessControllException* é uma sub-classe de *java.lang.SecurityException*.

Chamadas devem ser feitas preferivelmente ao método *checkPermission* do gerente de segurança instalado ao invés de diretamente a classe *AccessController*. Isto deve ser feito para:

- Permitir a comparação que será omitida se o gerente de segurança não estiver instalado.
- Possibilitar a possibilidade que um gerente de segurança ao ser instalado execute verificações adicionais que não são fáceis de serem capturadas com objetos tipo *Permission*.

AccessController é uma classe que contém apenas métodos estáticos.

6. Exceções

Existem dois tipos de exceções associadas ao pacote de segurança: *java.lang.SecurityException* e *java.security.GeneralSecurityException*. A classe *java.security.GeneralSecurityException* é uma nova classe de exceção que foi adicionada ao Java 2. Esta é uma subclasse de *java.lang.Exception*. Todas as exceções de segurança são sub-classes diretas desses dois tipos, exceto *ProviderException* e *InvalidParameterException*.

A classe *SecurityException* e suas sub-classes *java.security.AccessControlException* e *java.security.cert.CertificateException* são exceções em tempo de execução que são lançadas quando ocorre uma violação de segurança (lançada por uma falha em uma chamada de *checkPermission*), como na tentativa de acesso a um arquivo quando esta permissão não é autorizada. Normalmente, desenvolvedores de aplicações não capturam essas exceções.

Em geral, classes de exceção de segurança não são relacionadas a permissões de classes. Esta exceção é lançada a partir de várias classes de criptografia em que podem ocorrer erros. Por exemplo, a exceção *NoSuchAlgorithmicException* indica que um código está solicitando um algoritmo que não foi instalado na JVM. Esta exceção é mais similar a *FileNotFoundException* do que a *SecurityException*.

Todas as exceções que não foram mencionadas no pacote *java.security* são sub-classes de *GeneralSecurityException*. São as seguintes:

- *InvalidAlgorithmParameterException*
- *KeyException*
- *KeyManagementException* (estende *KeyException*)
- *KeyStoreException*
- *InvalidKeyException*
- *DigestException*
- *NoSuchAlgorithmException*
- *SignatureException*
- *UnrecoverableKeyException*

7. Construindo uma classe de permissão

Veremos um exemplo de como é possível construir uma classe de permissão. Implementaremos um programa para administrar uma folha de pagamento. Desejamos criar permissões para que os usuários possam ver o histórico de pagamento. Também podemos permitir que o departamento de RH atualize a faixa de salário dos empregados.

```
import java.security.*;
import java.util.*;

public class XYZPayrollPermission extends Permission {
  protected int mask;
  static private int VIEW = 0x01;
  static private int UPDATE = 0x02;
  public XYZPayrollPermission(String name) {
    // Uma permissão deve sempre ter uma ação, então escolheremos um padrão
    this(name, "view");
  }

  public XYZPayrollPermission(String name, String action) {
    // Nossa superclasse, contudo, não suporta ações
    // deste modo, não será estabelecida
    super(name);
    parse(action);
  }

  private void parse(String action) {
    // Procura nessa ação uma string com as palavras "view" e "update",
    // separadas por espaço ou vírgula
    StringTokenizer st = new StringTokenizer(action, ",\t ");
    mask = 0;
    while (st.hasMoreTokens( )) {
      String tok = st.nextToken( );
      if (tok.equals("view"))
        mask |= VIEW;
      else if (tok.equals("update"))
        mask |= UPDATE;
      else throw new IllegalArgumentException("Unknown action " + tok);
    }
  }

  public boolean implies(Permission permission) {
    if (!(permission instanceof XYZPayrollPermission))
      return false;
    XYZPayrollPermission p = (XYZPayrollPermission) permission;
    String name = getName( );

    // O nome deve ser o curinga (*), que significa
    // todos os nomes possíveis, ou deve corresponder ao nosso nome
    if (!name.equals("*") && !name.equals(p.getName( )))
      return false;

    // Do mesmo modo, as ações requisitadas devem corresponder a todas as
    // ações construídas
    if ((mask & p.mask) != p.mask)
      return false;
    // Se a ação e o nome correspondem, retornamos true
    return true;
  }

  public boolean equals(Object o) {
    if (!(o instanceof XYZPayrollPermission))
      return false;

    // Pela igualdade, verificamos o nome e a máscara de ação
```

```
        // Devemos fornecer um método de definição como este, uma vez que
        // a segurança do sistema espera que façamos uma verificação
        // profunda da igualdade. É melhor do que confiar na referência do
        // objeto de igualdade
        XYZPayrollPermission p = (XYZPayrollPermission) o;
        return ((p.getName().equals(getName( ))) && (p.mask == mask));
      }

      public int hashCode( ) {
        // Devemos sempre fornecer um código hash para permissões,
        // porque os hashes devem corresponder se as permissões comparadas
        // são iguais. O padrão de implementação deste método
        // não fornece isso.
        return getName().hashCode( ) ^ mask;
      }

      public String getActions( ) {
        // Este método deve retornar a mesma string, não importando como
        // a lista de ações foi passada para o construtor.
        if (mask == 0)
          return "";
        else if (mask == VIEW)
          return "view";
        else if (mask == UPDATE)
          return "update";
        else if (mask == (VIEW | UPDATE))
          return "view, update";
        else throw new IllegalArgumentException("Unknown mask");
      }

      public PermissionCollection newPermissionsCollection( ) {
        return new XYZPayrollPermissionCollection( );
      }

    }
```

Os atributos de classe são obrigados a manter a informação sobre as ações realizadas. Apesar da superclasse fazer referência a estas ações, providências não são tomadas para que sejam armazenadas ou processadas. Essa lógica é fornecida no método *parse*(). Neste método, adota-se a convenção de se obter uma *String* de ação tratada como uma lista de ações que são separadas por vírgulas ou espaços em branco.

Como solicitado, implementamos os métodos *equals*() e *hashCode*(). Consideramos objetos iguais se seus nomes e suas ações são iguais, e construímos um código *hash* adequado.

A implementação do método *getActions*() é típica. Somos obrigados a retornar a mesma *String* de ação para um objeto de permissão que foi construído por uma lista de ações de *view* e *update* como para o que foi construído através da lista. Esta exigência é uma das principais razões pelas quais as ações são armazenadas com uma máscara, pois permite construir essa *String* de ação no formato correto.

Por fim, o método *implies*() é responsável por determinar como o curinga e outras permissões implícitas serão manipuladas. Se o nome informado ao construtor do nosso objeto for um asterisco, então igualaremos este a qualquer outro nome. Quando o método *implies*() for chamado por este objeto curinga, o nome sempre corresponderá, pois a máscara da ação tem a lista completa das ações. Então, a máscara de comparação sempre autorizará a máscara que testamos na comparação. Se o método *implies*() for chamado por um objeto diferente, retorna *true* se os nomes forem iguais a máscara de um objeto é um subconjunto da máscara alvo.

Perceba que também devemos implementar a lógica de tal forma que a permissão para executar uma atualização que implique na permissão de executar um aspecto simplesmente pela mudança na lógica de teste da máscara. Não estamos limitados somente ao uso de curingas correspondentes no método *implies*().

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.