互联网编程

ppt 内容

目录

一．多线程与线程池

进程与线程

- **进程**：具有一定独立功能的程序关于某个数据集合上的一次运行活动，进程是系统**进行资源分配和调度**的一个**独立单元。**

- 线程：

1 进程的一个实体。
2 是 CPU 调度和分派的基本单位。它是比进程更小的能独立运行的基本单位。
3 线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源（程序计数器，一组寄存器和栈。）
4 可与同属一个进程的其他的线程共享进程所拥有的全部资源。

- 一个线程可以创建和撤销另一个线程，同一个进程中的多个线程可以并发执行。
- 一个程序至少有一个进程，一个进程**至少有一个线程**（主线程）。
- 进程拥有**独立内存单元**，多个线程共享内存。
- 创建**进程开销大**，线程**开销较小**。

从多进程到多线程

- 提升并发量的方法
  - 过去：以进程为单位
    - 多进程：支持几百个进程
    - 进程池：支持上千个进程
  - 当前：以线程为单位
    - 多线程：支持几千个线程
    - 线程池：支持上万个线程
- 其他
  - 异步 I/O
  - 多个冗余服务器

java 多线程编程

- 1.创建启动线程对象
  ```
  Thread t = new Thread(      );
  t.start( );
  ```
- 2.让线程完成的任务放在哪儿？重写 run（）方法
- 创建线程类重写 run 方法做“我想让线程干的事”
- 派生 Thread
- 实现 Runnable 接口

运行线程：方法 1

- 创建线程对象，执行 start()方法

```
Thread t = new DigestThread();
t.start();
```

- 需要定义 Thread 子类，并实现 run()方法

```
class DigestThread extends Thread {
   public void run() {
   }
```

}

- 多线程程序需要 main()方法和子线程都返回后才退出程序

运行线程：方法 2

实现 Runnable 接口

```
class DigestRunnable implements Runnable{
    public void run() {
    }
}
Thread t = new Thread(new DigestRunnable());
t.start();
```

从线程返回信息

run()和 start()不返回任何数据

可否在线程中定一个方法，由主线程调用？

```
public class ReturnDigest extends Thread {

  private String filename;
  private byte[] digest;

  public ReturnDigest(String filename) {
    this.filename = filename;
  }
  public byte[] getDigest() {
    return digest;
  }
```

出错原因：

ReturnDigestUserInterface.java 中，由于 start()方法立刻返回，并不等子线程执行完毕，所以 getDigest()返回的是 null。

解决方法：

竞态条件？即把 getDigest()的执行放在 main()的最后面

- 竞态条件

```
public static void main(String[] args) {
  ReturnDigest[] digests = new ReturnDigest[args.length];
  for (int i = 0; i < args.length; i++) {
    // Calculate the digest
    digests[i] = new ReturnDigest(args[i]);
    digests[i].start();
  }
  for (int i = 0; i < args.length; i++) {
    // Now print the result
    StringBuffer result = new StringBuffer(args[i]);
    result.append(": ");
    byte[] digest = digests[i].getDigest();
    result.append(DatatypeConverter.printHexBinary(digest));
    System.out.println(result);
  }
}
```

- ☐ 轮询(Polling)

  设置一个标志，不停地查询标志值

```
public static void main(String[] args) {
  ReturnDigest[] digests = new ReturnDigest[args.length];
  for (int i = 0; i < args.length; i++) {
    // Calculate the digest
    digests[i] = new ReturnDigest(args[i]);
    digests[i].start();
  }
  for (int i = 0; i < args.length; i++) {
    while (true) {
      // Now print the result
      byte[] digest = digests[i].getDigest();
      if (digest != null) {
        StringBuilder result = new StringBuilder(args[i]);
        result.append(": ");
        result.append(DatatypeConverter.printHexBinary(digest));
        System.out.println(result);
        break;
      }
    }
  }
}
```

- ☐ 例子 CallbackDigest.java 为静态方法回调
- ☐ 实例方法回调

```
public class InstanceCallbackDigest implements Runnable {

  private String filename;
  private InstanceCallbackDigestUserInterface callback;

  public InstanceCallbackDigest(String filename,
    InstanceCallbackDigestUserInterface callback) {
    this.filename = filename;
    this.callback = callback;
  }
```

同步

- ☐ 多线程同时访问资源

  修改内存数据

  文件操作
- ☐ 实现方法

  同步块

同步方法

同步块

解决方案

同步块：synchronized

```
synchronized (System.out) {
   System.out.print(input + ": ");
   System.out.print(...(digest));
   System.out.println();
}
```

```
synchronized (a) {
...
}
```

```
synchronized (b) {
...
}
```

- 同一个对象上同步的代码串行执行
- 不同对象上同步的代码可并行

多个线程写 Log 文件

```
public class LogFile {

  private Writer out;

  public LogFile(File f) throws IOException {
    FileWriter fw = new FileWriter(f);
    this.out = new BufferedWriter(fw);
  }

  public void writeEntry(String message) throws IOException {
    Date d = new Date();
    out.write(d.toString());
    out.write('\t');
    out.write(message);
    out.write("\r\n");
  }

  public void close() throws IOException {
    out.flush();
    out.close();
  }
}
```

□ 同步的两种实现方法

```
public void writeEntry(String message) throws IOException {
  synchronized(out){
    Date d = new Date();
    out.write(d.toString());
    out.write('\t');
    out.write(message);
    out.write("\r\n");
  }
}
public void writeEntry(String message) throws IOException {
  synchronized(this){
    Date d = new Date();
    out.write(d.toString());
    out.write('\t');
    out.write(message);
    out.write("\r\n");
  }
}
```

同步方法

对方法声明添加 synchronized 修饰符

```
public synchronized void writeEntry(String message) t
  Date d = new Date();
  out.write(d.toString());
  out.write('\t');
  out.write(message);
  out.write("\r\n");
}
```

同步带来的问题

1.程序性能下降，甚至严重下降
2.增加了死锁可能性
3.可能不能有效保护数据
  如前面例子保护的是对象，不是文件

死锁(deadlock)

```
class Deadlocker {
  int field_1;
  int field_2;
  private Object lock_1 = new int[1];
  private Object lock_2 = new int[1];

  public void method1(int value) {
    synchronized (lock_1) {
      synchronized (lock_2) {
        field_1 = 0; field_2 = 0;
      }
    }
  }
  public void method2(int value) {
    synchronized (lock_2) {
      synchronized (lock_1) {
        field_1 = 0; field_2 = 0;
      }
    }
  }
}
```

- 两个线程需要独占访问同样的资源集
- 防止死锁
  - 避免不必要的同步
  - 同步块尽可能的小
  - 尽可能不一次同步多个对象（但Java类库内方法可能含有同步）

死锁举例

一个银行账号：账户 A，余额 2000 元
线程 1:存款线程
线程 2：取款线程

线程1：存入500元	线程2：取出300元
获取当前余额：2000	
计算最新余额：2000+500=2500	
线程中断，等待下次被系统挑中执行	获得当前余额：2000
	计算最新余额：2000-300=1700
	线程中断，等待下次被系统挑中执行
再次执行，给更新余额2500	
	再次执行，更新余额：1700

无锁状态

线程1：存入500元	线程2：取出300元
获取当前余额：2000	
计算最新余额：2000+500=2500	
线程中断，等待下次被系统挑中执行	
	获取账户A的锁：失败，进入阻塞状态
被系统选中，再次执行，2500	
释放账户A的锁	
	获取账户A的锁：成功
	获得当前余额：2500
	计算最新余额：2500-300=2200
	更新余额：2200
	释放账户A的锁

加锁状态

A给B转账，同时B给A转账

线程1：A给B转账	线程2：B给A转账
获取账户A的锁：成功	
线程中断，等待下次被系统挑中执行	
	获取账户B的锁：成功
	线程中断，等待下次被系统挑中执行
获取账户B的锁：失败，继续等待	
	获取账户A的锁：失败，继续等待

同步的替代方式

- 使用局部变量
- 创建不可修改对象
  - 如 String
- 将非线程安全的类作为线程安全类的一个私有属性
- java.util.concurrent.atomic
  - int->AtomicInteger
  - long-AtomicLong
  - 如方法 int AtomicInteger.addAndGet(int delta)
- Collections.synchronizedSet(foo)

线程优先级

- public final void setPriority(int n)
- 优先级值：0-10

- Thread.MIN_PRIORITY (1)
- Thread.NORMAL_PRIORITY (5)，默认
- Thread.MAX_PRIORITY (10)

- 不同操作系统支持不同（小心！）

线程1：账户A给账户B转账	线程2：账户B给账户A转账
获取账户A的锁：成功	
线程中断，等待下次被系统挑中执行	
	获取账户A的锁：失败，继续等待
获取账户B的锁：成功	
执行转账	
释放B的锁	
释放A的锁	
	获取账户A的锁：成功
	获取账户B的锁：成功
	执行转账
	释放B的锁
	释放A的锁

连接线程(join)

public final void join() throws
InterruptedException

public final void join(long milliseconds) throws
InterruptedException

等待线程执行完毕
例子：

```
double[] array = new double[10000];
for (int i = 0; i < array.length; i++) {
  array[i] = Math.random();
}
SortThread t = new SortThread(array);
t.start();
try {
  t.join();
  System.out.println("Minimum: " + array[0]);
  System.out.println("Median: " + array[array.length/2]);
  System.out.println("Maximum: " + array[array.length-1]);
} catch (InterruptedException ex) {
}
```

线程池

- 优点：
  - 降低线程创建和释放的开销
  - 避免运行过多线程超出计算机处理能力
- submit()提交任务，由 THREAD_COUNT 个线程之一处理
  - shutdown()方法：通知线程池没有更多的线程了。

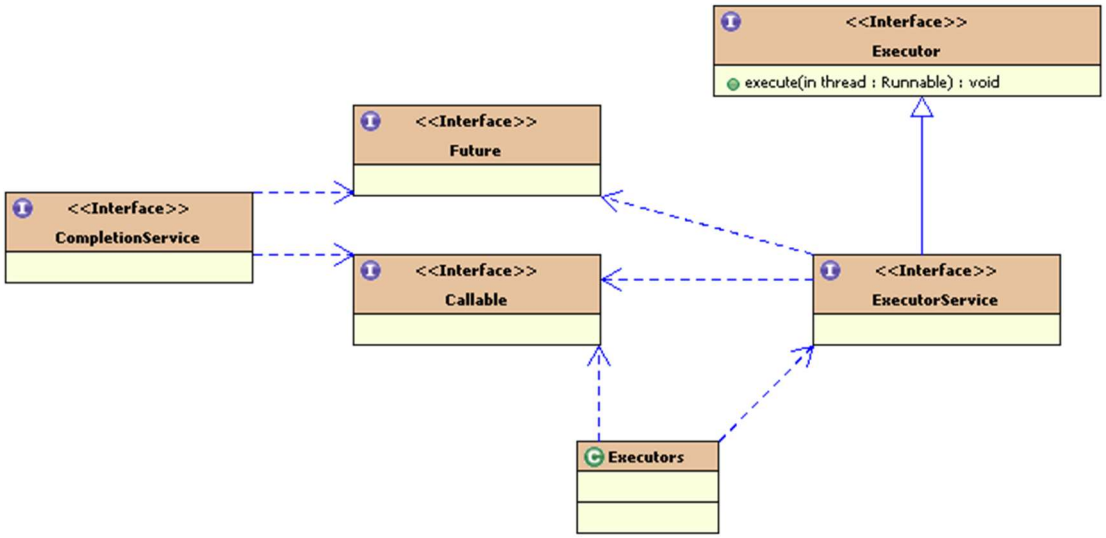
```
ExecutorService pool = Executors.newFixedThreadPool(THREAD_COUNT);

for (int i = 0; i < files.length; i++) {
  if (!files[i].isDirectory()) { // don't recurse directories
      Runnable task = new GZipRunnable(files[i]);
      pool.submit(task);
  }
}
```

二．类图

类图：Executor 框架

- Executor 框架类之间的关系
- 在 Executor 框架中，使用执行器(Executor)来管理 Thread 对象，从而简化了并发编程

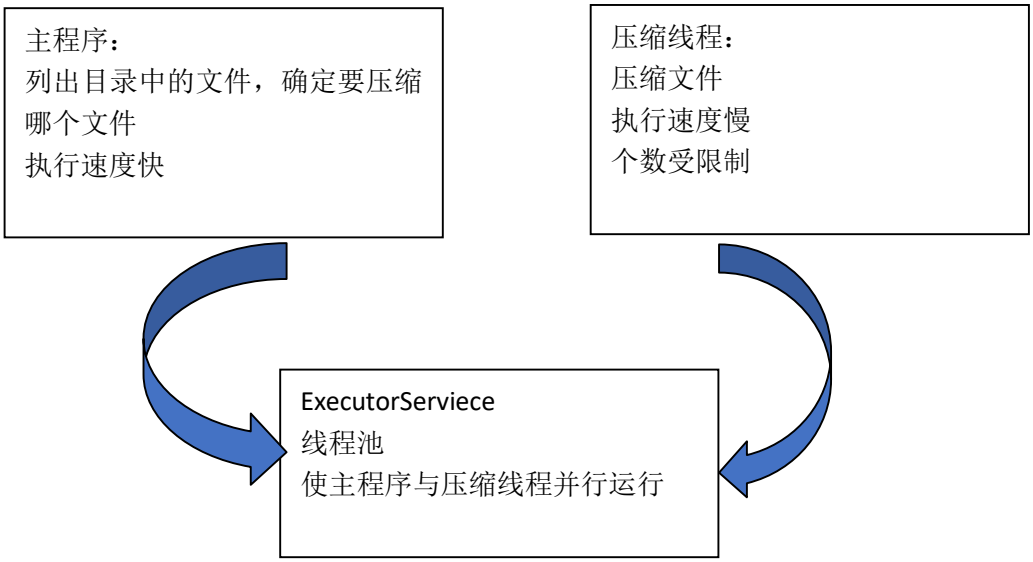


Callable 接口类似于 Runnable，为那些其实例可能被另一个线程执行的类设计的。 提供一个 call()方法作为线程的执行体。call()方法比 run()方法更加强大：有返回值，可以抛出异常。

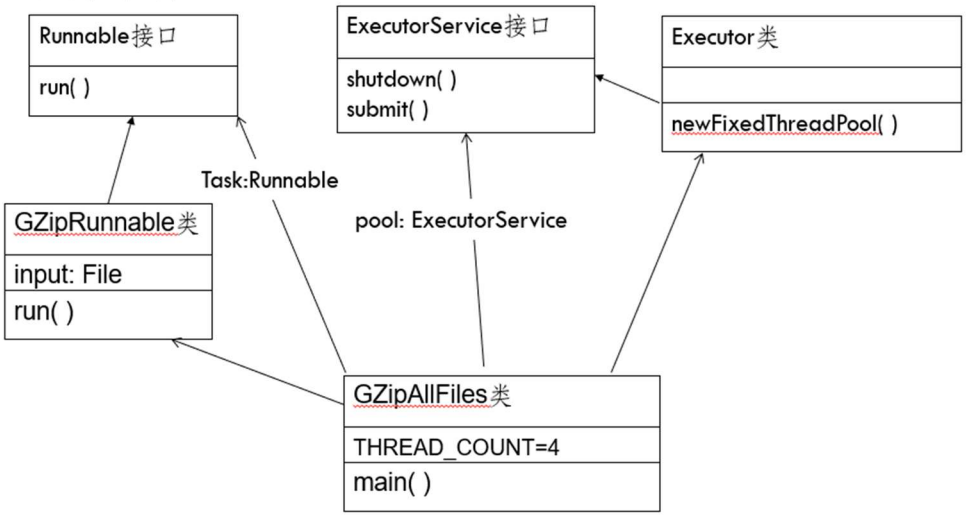
- Executor 框架：是指 jdk 并发库中与 Executor 相关的功能类，包括 Executor、Executors、ExecutorService、Future、Callable 等。

编程应用问题场景:压缩文件

- 程序设计思想



- 程序结构



三．InetAddress 类

- 人难以记住大量 IP 地址
- 域名系统（Domain Name System, DNS）
- DNS 将易于记忆的域名与 IP 地址联系起来
  - 210.39.3.164 <-> www.szu.edu.cn
- 关系

- 一台主机可能有多个 IP 地址
- 一个 IP 地址可能被多个域名指向
- 一个域名可能指向多个 IP 地址

- Java 中对 IP 地址描述的类（IPv4&IPv6）也是其高层的表示；
- 大部分网络类都要用到这个类
- 常用子类有： Inet4Address 和 Inet6Address

创建方法

- 由于 InetAddress 没有 public 的构造方法，因此，要想创建 InetAddress 对象，必须得依靠它的四个静态方法；
- InetAddress 可以通过 getLocalHost 方法得到本机的 InetAddress 对象；
- 也可以通过 getByName、getAllByName 和 getByAddress 得到远程主机的 InetAddress 对象；
- **创建方法：**
- Static InetAddress getByName(String s)
- Static InetAddress[] getAllByName(String s)
- Static InetAddress getLocalHost()

getLocalhost 方法

- 使用 getLocalHost 可以得到描述本机 IP 的 InetAddress 对象。这个方法的定义：
- public static InetAddress getLocalHost() throws UnknownHostException

代码实例：

```
import java.net.*;
public class MyAddress{
    public static void main(String[] args){
        try{
            InetAddress address = InetAddress.getLocalHost();
            System.out.println(address);
        }
        catch(Exception e){
            System.out.println(e);
        }
    }
}
```

当本机绑定了多个 IP 地址时候，只返回第一个地址！

getByName

```
import java.net.*;

public class ResolveName{
    public static void main(String[] args){
        if(args.length != 1)
        {
            System.out.println("Usage: java ResolveName domain.name");
            return;
        }
        try{
            InetAddress address = InetAddress.getByName(args[0]);
            System.out.println(address);
        }
        catch(Exception e){
            System.out.println(e);
        }

    }
}
```

- 可以通过该方法指定域名从 DNS 中查得与域名相对应的 IP 地址。
- getByName 一个 String 类型参数，可以通过这个参数指定远程主机的域名，它的定义如下
  public static InetAddress getByName(String host) throws UnknownHostException
- 如果 host 所指的域名对应多个 IP，getByName 返回第一个 IP。
- 如果本机名已知，可以使用 getByName 方法来代替 getLocalHost。
- 当 host 的值是 localhost 时，返回的 IP 一般是 127.0.0.1。
- 如果 host 是不存在的域名，getByName 将抛出 UnknownHostException 异常，如果 host 是 IP 地址，无论这个 IP 地址是否存在，getByName 方法都会返回这个 IP 地址（因此 getByName 并不验证 IP 地址的正确性）。

getByName 实例(1)

```
import java.net.*;
public class ResolveName{
    public static void main(String[] args){
        if(args.length != 1)
        {
            System.out.println("Usage: java ResolveName domain.name");
            return;
        }
```

```
            try{
                InetAddress address = InetAddress.getByName(args[0]);
                System.out.println(address);
            }
            catch(Exception e){
                System.out.println(e);
            }

    }
}
```

执行 java ResolveName www.szu.edu.cn

结果：www.szu.edu.cn/210.39.12.247

getByName 实例（2）

- 测试 2：本机名 DESKTOP-VKK8CFC
- 执行如下命令
- java ResolveName DESKTOP-VKK8CFC
- 运行结果

  DESKTOP-VKK8CFC/172.31.70.150

getByName 实例（3）

- 测试 3：代表本机的 localhost
- 执行如下命令
- java ResolveName localhost
- 运行结果：localhost/127.0.0.1

getByName 实例（4）

- 对于本机来说，除了可以使用本机名或 localhost 外，还可以在 hosts 文件中对本机做“IP/域名”映射（在 Windows 操作系统下）。这个文件在 C:\Windows\System32\drivers\etc 中。打开 hosts，在最后加一行如下所示的字符串：
- 192.168.18.100 www.mysite.com
- 测试 4：本机域名 www.mysite.com
- 执行如下命令
- java ResolveName www.mysite.com
- 运行结果: www.mysite.com/192.168.18.100

getAllByName 方法

- 使用 getAllByName 方法可以从 DNS 上得到域名对应的所有的 IP。这个方法返回一个 InetAddress 类型的**数组**。这个方法的定义如下：

public static InetAddress[] getAllByName(String host) throws UnknownHostException

- 与 getByName 方法一样，当 host 不存在时，getAllByName 也会抛出 UnknowHostException 异常，getAllByName 也不会验证 IP 地址是否存在。

getAllByName 实例

```
import java.net.*;
public class ResolveAllName{
    public static void main(String[] args){
        if(args.length != 1)
        {
            System.out.println("Usage: java ResolveAllName domain.name");
            return;
        }
        try{
            InetAddress[] addresses = InetAddress.getAllByName(args[0]);
            for(InetAddress address: addresses)
                System.out.println(address);
        }
        catch(Exception e){
            System.out.println(e);
        }

    }
}
```

Java ResolveAllName www.baidu.com

运行结果：

www.baidu.com/163.177.151.110

www.baidu.com/163.177.151.109

getByAddress

- 这个方法必须通过 IP 地址来创建 InetAddress 对象，而且 IP 地址必须是 byte 数组形式。getByAddress 方法有两个重载形式，定义如下：

  public static InetAddress getByAddress(byte[] addr) throws UnknownHostException

  public static InetAddress getByAddress(String host, byte[] addr) throws UnknownHostException

- 第一个重载形式只需要传递 byte 数组形式的 IP 地址，getByAddress 方法并不验证这个 IP 地址是否存在，只是简单地创建一个 InetAddress 对象。addr 数组的长度必须是 4（IPv4）或 16（IPv6），如果是其他长度的 byte 数组，getByAddress 将抛出一个 UnknownHostException 异常。
- 第二个重载形式多了一个 host，这个 host 和 getByName、getAllByName 方法中的 host 的意义不同，getByAddress 方法并不使用 host 在 DNS 上查找 IP 地址，这个 host 只是一个用于表示 addr 的别名。

getByAddress 实例

```
import java.net.*;
public class    MyInetAddress4 {
      public static void main(String[] args) throws Exception{
            byte ip[] = new byte[] { (byte) 141, (byte) 146, 8 , 66};
            InetAddress address1 =
                              InetAddress.getByAddress(ip);
            InetAddress address2 =
                              InetAddress.getByAddress("Oracle 官 方 网 站 ", ip);
      System.out.println(address1);
      System.out.println(address2);
      }
}
```

运行结果：

```
<terminated> MyInetAddress4 [Java Application]
/141.146.8.66
Oracle官方网站/141.146.8.66
```

四．URL 与 URI

URL 类

- java.net.URL
  - final
  - 创建后不可修改
- 创建 URL 对象
  - 检查模式是否支持
  - 不检查 URL 是否格式正确

- 不检查资源是否存在

URL 类：协议支持

```
private static void testProtocol(String url) {
  try {
    URL u = new URL(url);
    System.out.println(u.getProtocol() + " is supported");
  } catch (MalformedURLException ex) {
    String protocol = url.substring(0, url.indexOf(':'));
    System.out.println(protocol + " is not supported");
  }
}
```

字符串构造 URL

定义：

public URL(String url) throws MalformedURLException

例子：

```
try {
URL u = new URL("http://www.szu.edu.cn/");
}
catch (MalformedURLException ex) {
      System.err.println(ex);
}
```

分块构造 URL

- 定义：

public URL(String protocol, String hostname, String file)throws MalformedURLException

- 例子：

```
try {
      URL u = new URL("http", "www.eff.org", "/blueribbon.html#intro");
}
catch (MalformedURLException ex) {
      throw new RuntimeException("shouldn't happen; all VMs recognize http");
}
```

注意：勿遗漏 file 参数前的/

构造相对 URL

- 定义：

public URL(URL base, String relative) throws MalformedURLException

- 例子：

```
try {
   URL u1 = new URL("http://www.szu.edu.cn/a/b.html");
   URL u2 = new URL(u1, "mailinglists.html");
}
catch (MalformedURLException ex) {
   System.err.println(ex);
}
```

从 URL 获取数据

- URL 类的获取数据方法
- 获得 InputStream
  - **public** InputStream openStream()
- 获得 URLConnection 对象
  - **public** URLConnection openConnection()
  - **public** URLConnection openConnection(Proxy proxy)
- 获得内容对象如 String 或 Image
  - **public** Object getContent()
  - **public** Object getContent(Class[] classes)

从 URL 获取数据：openStream

```
try {
    URL u = new URL("http://www.szu.edu.cn");
    InputStream in = u.openStream();
    int c;
    while ((c = in.read()) != -1)
        System.out.write(c);
    in.close();
}catch (IOException ex) {
    System.err.println(ex);
}
```

从 URL 获取数据：openConnection

```
try {
URL u = new URL("http://www.szu.edu.cn/");
      try {
            URLConnection uc = u.openConnection();
            InputStream in = uc.getInputStream();
            // read from the connection...
      } catch (IOException ex) {
            System.err.println(ex);
      }
} catch (MalformedURLException ex) {
    System.err.println(ex);
}
```

- 可访问服务器发送的所有数据
  - 如 HTTP 头信息
- 可以向服务器发送数据
  - Email
  - 提交表单
- openConnection(Proxy proxy)
  - 可设置代理服务器

从 URL 获取数据：getContent

- 用法：

```
URL u = new URL("http://mesola.obspm.fr/");
Object o = u.getContent();
//强制将对象 o 转换为特定类型
if(o instanceof URLImageSource){
      ...
```

- 缺点
  - 难以事先预知对象类型
  - 不同的类型在不同的 Java 版本中不同

URI

- URI：Uniform Resource Identifier
  - URL：Uniform Resource Locator
    http://www.szu.edu.cn/szu.asp
  - URN：Uniform Resource Name
    urn:isbn:0-486-27557-4

mailto:yu.zhou@szu.edu.cn

- 虽 URL 可视作一种 URI 的抽象，但 Java 类 URL 并不是 URI 的子类
- URL 类与 URI 类皆是 Object 的子类

URI 语法

- URI 的语法由一个模式和一个模式特定部分组成

模式：模式特定部分

其中模式包括：

data, file, ftp, http, mailto, magnet, telnet, urn

形式： //authority/path？query

URI 构造方法

- **public** URI(String uri) **throws** URISyntaxException
- 只检查 URI 语法，不检查协议

```
URI voice = new URI("tel:+1-800-9988-9938");
URI web = new URI("http://www.a.com/a.jsp#id=hbc");
URI book = new URI("urn:isbn:1-565-92870-9");
```

- **public** URI(String scheme, String schemeSpecificPart, String fragment) **throws** URISyntaxException

```
URI absolute = new URI("http", "//www.ibiblio.org" , null);
URI relative = new URI(null, "/javafaq/index.shtml", "today");
```

URI 格式

- scheme:scheme-specific-part:fragment
  - 模式：模式特定部分：片段
- 方法
  - public String getScheme()
  - public String getSchemeSpecificPart()
  - public String getRawSchemeSpecificPart()
  - public String getFragment()
  - public String getRawFragment()
  - public boolean isOpaque()

相对 URI 的解析

```
URI absolute = new URI("http://www.example.com/");
```

URI relative = new URI("images/logo.png");
URI resolved = absolute.resolve(relative);

- resolved 结果是：http://www.example.com/ images/logo.png

URI top = new URI("javafaq/books/");
URI resolved = top.resolve("jnp3/index.html");

- resolved 结果是：javafaq/books/jnp3/index.html

绝对 URI 转相对 URI

- Public URI relativize(URL uri)

URI absolute = new URI(“http://www.example.com/images/logo.png”);
URI top = new URI(“http://www.example.com/”);
URI relative = top.relativize(absolute);
URI 对象 relative 包含相对 URI：
Images/logo.png

五．URLConnection 类

- public abstract class URLConnection extends Object
- URLConnection 是一个抽象类
- URLConnection -> HttpURLConnection
- 提供了更多与 HTTP 服务器的交互控制
- 可以检查服务器发送的首部，做出响应。
- 可以设置客户端请求中使用的首部字段。
- 可用 GET、POST、PUT 等 HTTP 请求方法向服务器发回数据。
-

URLConnection 类的使用步骤

1. 创建 URL 对象
2. 调用 URL.openConnection()获取 URLConnection 对象
3. 配置 URLConnection 对象
4. 读取头信息
5. 获得输入流，并读数据
6. 获得输出流，并写数据
7. 关闭连接

创建 URLConnection 对象

```
try {
    URL u = new URL("http://www.szu.edu.cn/");
    URLConnection uc = u.openConnection();
    // read from the URL...
}
catch (MalformedURLException ex) {
    System.err.println(ex);
}
catch (IOException ex) {
    System.err.println(ex);
}
```

实现使用 URLConnection 下载一个 Web 页面

```
InputStream in = null;
        try {
           // Open the URL for reading
           URL u = new URL(args[0]);
           URLConnection uc = u.openConnection();
           in = uc.getInputStream();
           // buffer the input to increase performance
           in = new BufferedInputStream(in);
           // chain the InputStream to a Reader
           Reader r = new InputStreamReader(in);
           int c;
           while ((c = r.read()) != -1) {
              System.out.print((char) c);
           }
        } catch (MalformedURLException ex) {
           System.err.println(args[0] + " is not a parseable URL");
        } catch (IOException ex) {
           System.err.println(ex);
        } finally {
           if (in != null) {
              try {
                 in.close();
              } catch (IOException e) {
                 // ignore
              }
           }
        }
```

URL 与 URLConnection 区别

- URLConnection 提供了对 HTTP 头信息访问
- URLConnection 可以配置发给服务器的参数
- URLConnection 可向服务器发送数据

一个 HTTP 服务器响应的头部信息

- 头部中的常用字段有：
- Content-type
- Content-length
- Content-encoding
- Date
- Last-modified
- Expires

一个 Appache Web 服务器返回的响应信息中的 HTTP 头部：

```
HTTP/1.1  301  Moved Permanently
Date:  Sun, 21 Apr 2013  15:12:46 GMT
Server: Apache
Location: http://www.ibiblio.org/
Content-Length:  296
Connection: close
Content-type:  text/html;  charset = iso-8859-1
```

头字段信息获取的方法

- public String getContentType()
  - 返回响应主体的 MIME 类型，如：text/html; charset=UTF-8
  - http 默认编码方式 ISO-8859-1
- public int getContentLength()
- public long getContentLengthLong()
  - 返回响应主体的字节数
  - 如无 Content-length 首部，返回-1
  - 当程序中需要准备知道要读取的字节数，或需要预先创建一个足够大的缓冲区来保存数据时使用。

头部字段

- public String getContentEncoding()

- 返回内容压缩编码（与字符编码不同）
- public long getDate()
  - 发送文档的时间
- public long getExpiration()
  - 文档失效时间
- public long getLastModified()
  - 文档最后修改时间

getContentLength()

Q:如何获取二进制文件并且保存在磁盘中？
问题：URL 类中 openstream()可以下载文本文件，但是 HTTP 服务器并不总会在数据发送完后就立刻关闭连接，因此，无法判断停止读取的时机，也会造成信息不准确。
解决思路：
利用 URLConnection 的 getContentLength()方法，得到需要下载文件的长度，然后以此为终止条件，读取相应的字节数。

具体实现（3）

saveBinaryFile 方法：part2

```
try (InputStream raw = uc.getInputStream()) {
  InputStream in  = new BufferedInputStream(raw);
  byte[] data = new byte[contentLength];
  int offset = 0;
  while (offset < contentLength) {
     int bytesRead = in.read(data, offset, data.length - offset);
     if (bytesRead == -1) break;
     offset += bytesRead;
  }
  if (offset != contentLength) {
    throw new IOException("Only read " + offset
        + " bytes; Expected " + contentLength + " bytes");
  }
```

1.字节数组 data 用来读取二进制文件（字节数组的大小为多少？）
2.循环读取文件中的字节，变量 offset 用来计数
3.read()返回-1，表示流意外结束，跳出循环，offset 用来保证按照规定字节数读取了文件。

具体实现（4）

saveBinaryFile 方法：part3

```
String filename = u.getFile();
filename = filename.substring(filename.lastIndexOf('/') + 1);
try (FileOutputStream fout = new FileOutputStream(filename)) {
  fout.write(data);
  fout.flush();
}
```

1. 使用 getfile()方法从 URL 获得文件名
2. 去掉路径信息
3. 为文件打开一个新的 fileOutputStream　fout（写入文件）
4. 将文件一次性写入
5. 注意 flush 的作用（I/O stream）

```
import java.io.*;
import java.net.*;
public class EncodingAwareSourceViewer {
    public static void main (String[] args) {
        if (args.length == 1) {
            try {
                // set default encoding
                String encoding = "ISO-8859-1";
                URL u = new URL(args[0]);
                URLConnection uc = u.openConnection();
                String contentType = uc.getContentType();
                int encodingStart = contentType.indexOf("charset=");
                if (encodingStart != -1) {
                    encoding = contentType.substring(encodingStart + 8);
                }
                InputStream in = new BufferedInputStream(uc.getInputStream());
                //Reader r = new InputStreamReader(in, encoding);
                Reader r = new InputStreamReader(in);
                int c;
                while ((c = r.read()) != -1) {
                    System.out.print((char) c);
                }
                r.close();

                System.out.println("charset=" + encoding);
            }
            catch (MalformedURLException ex) {
                System.err.println(args[0] + " is not a parseable URL");
            }
            catch (UnsupportedEncodingException ex) {
                System.err.println("Server sent an encoding Java does not support: " + ex.getMessage());
            }
            catch (IOException ex) {
                System.err.println(ex);
            }
        }
    }
}
```

Java 的 Web 缓存

- ☐ JVM 只支持一个共享缓存，在 HTTP 客户端程序中要使用缓存需要使用到自定义的 3 个具体子类来安装 URL 类使用的系统级缓存。
- ☐ ResponseCache 的一个具体子类
- ☐ CacheRequest 的一个具体子类
- ☐ CacheResponse 的一个具体子类
- ☐ 使用 ResponseCache.setDefault()方法把自定义缓存子类对象安装为默认缓存，来处理自定义的 CacheRequest 和 CacheResponse 子类。

CacheRequest 类

- ResponseCache 的 put()方法返回一个 CacheRequest 类型 的对象。
- CacheRequest 类表示在 ResponseCache 中存储资源的通道。
- CacheRequest 类的实例包装了一个 OutputStream 对象，HTTP 客户端程序可以调用该对象来将资源数据存储到缓存中。
- CacheRequest 类 ex7-7 CacheRequest.java ：

```
Package java.net

Public abstract class CacheRequest {
        //返回可以将响应正文写入 cache 库的 OutputStream
        public abstract OutputStream getBody( ) throws IOException;
        public abstract void abort( );//中止缓存响应的尝试
}
```

自定义 CacheResponse 子类

- 自定义一个 CacheResponse 子类，与一个自定义的 CacheRequest 子类（SimpleCacheRequest）对象和一个 CacheControl 对象绑定，支持 ResponseCache 对象的 get()方法从缓存中读取资源的数据或首部。

ResponseCache 类

- 代表 URLConnection 缓存的实现。
- 它的实例可以通过 ResponseCache.setDefault(ResponseCache)向系统注册,系统将调用这个对象:
- （1）存储从外部源已检索资源数据到缓存中。
- （2）试图获取可能是存储在缓存中的所请求的资源。
- ResponseCache 实现决定哪些资源应该被缓存,多长时间 。如果不能从缓存检索请求的资源, 协议处理程序将从其原始位置获取资源。
- ResponseCache 类 如下 ：

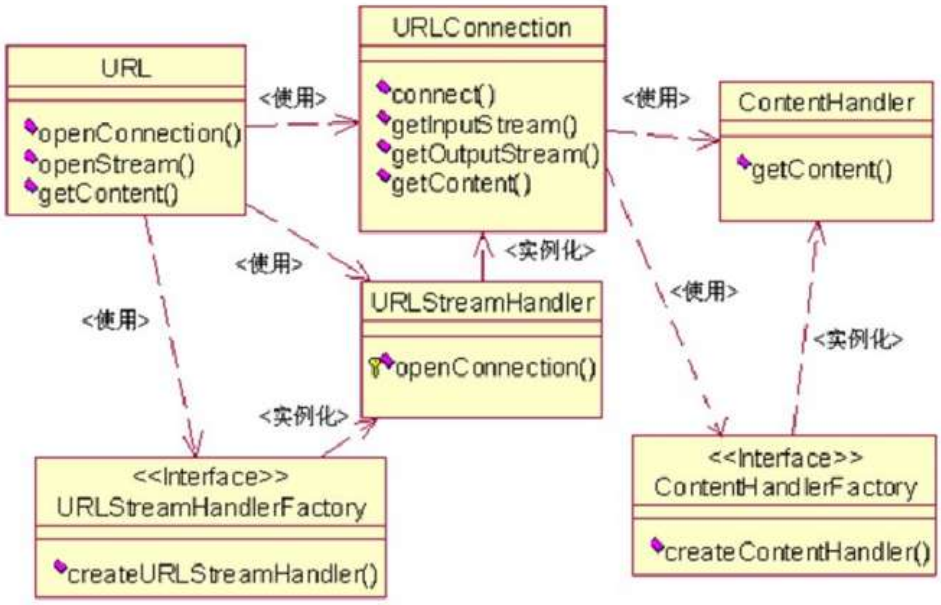
```
Package java.net

Public abstract class ResponseCache extends Object {
        //检索缓存的响应基于请求 uri,请求方法和请求头。
        public abstract CacheResponse get(URI uri, String rqstMethod,
Map<String,List<String>> rqstHeaders)
      throws IOExceiption;
    //协议处理器调用此方法检索资源之后,和 ResponseCache 必须决定
   //是否存储缓存的资源。
```

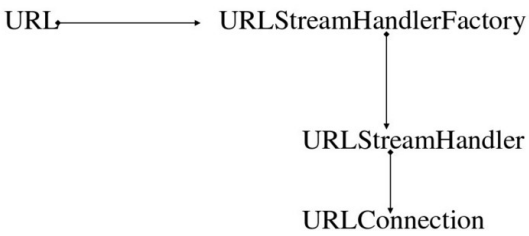
```
        public abstract CacheRequest put(URI uri, URLConnection conn) throws IOExceiption;
… …
}
```

协议处理框架

- URL 类：表示远程资源
- URLConnection 类：客户端与服务端的连接，获得输入、输出流
- URLStreamHandler 类：协议处理器，负责创建与协议相关的 URLconnection 对象
- 对于具体的协议，需要创建这 2 个抽象类的子类
- ContentHandler 类：内容处理器，负责解析服务器发送的数据并转换为相应的 Java 对象



URL 和 URLConnection 的关联

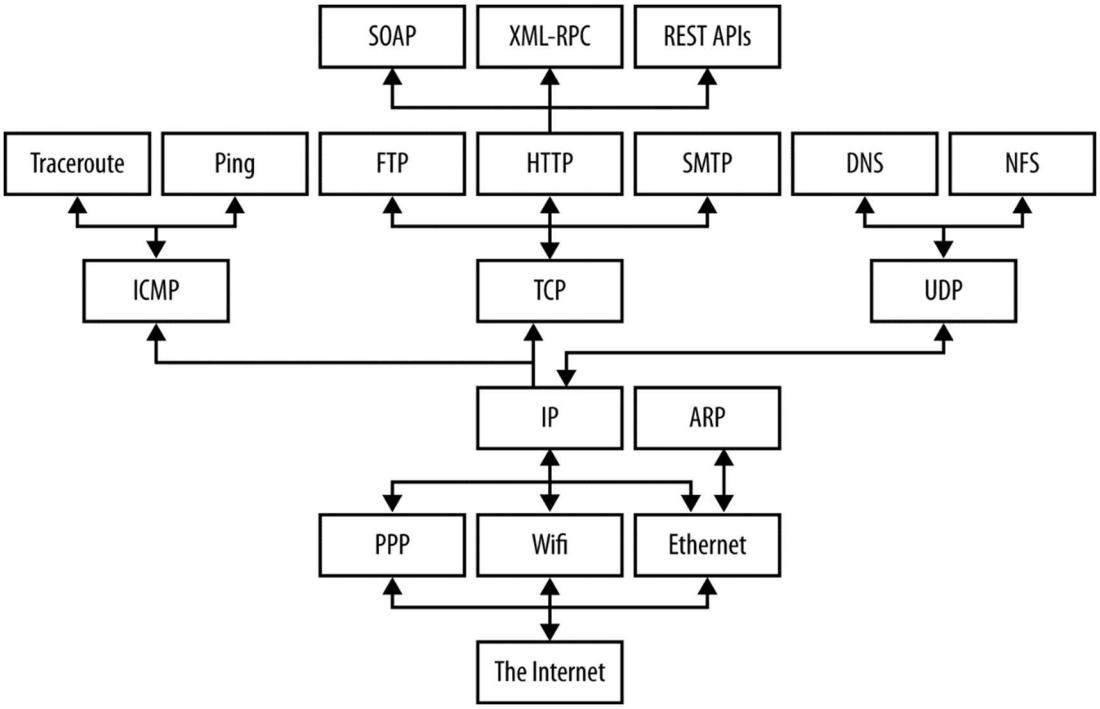


1. URL 类的 openConnection 方法是通过调用 URLStreamHandler 类的同名方法创建的
2. URL 类的 openStream 方法是通过调用 URLConnection 类的 getInputStream 方法获得的
3. URL 类的 getContent 方法是调用 URLConnection 类的同名方法

（该方法又调用了 ContentHandler 类的同名方法获得）

六．UDP

Transport layer protocol：TCP、UDP



- TCP
  - 保证数据到达
  - 保证数据顺序
  - 速度慢
  - 用途：HTTP、FTP、…
- UDP
  - 不保证一定到达
  - 不保证到达顺序
  - 速度快
  - 用途：DNS、NFS、TFTP、…
- TCP 协议
  - ServerSocket&Socket
- UDP 协议
  - 没有连接（connection）概念
  - 不支持流（stream）
  - 只是数据包的发送和接收
  - DatagramPacket
  - 对数据进行包装
- DatagramSocket
  - 发送和接收数据

UDP 客户端

- 创建DatagramSocket对象、并设置超时

```
DatagramSocket socket = new DatagramSocket(0);
socket.setSoTimeout(10000);
```

- 设置发送DatagramPacket数据包

```
InetAddress host =
            InetAddress.getByName("time.nist.gov");
DatagramPacket request =
         new DatagramPacket(new byte[1], 1, host, 13);
```

- 创建接收DatagramPacket数据包

```
byte[] data = new byte[1024];
DatagramPacket response =
            new DatagramPacket(data, data.length);
```

- 发送和接收数据

```
socket.send(request);
socket.receive(response);
```

- 解析出数据

```
byte [] b = response.getData();
```

```
import java.io.*;
import java.net.*;

public class DaytimeUDPClient {

  private final static int PORT = 13;
  private static final String HOSTNAME = "time.nist.gov";

  public static void main(String[] args) {
    try (DatagramSocket socket = new DatagramSocket(0)) {
      socket.setSoTimeout(10000);
      InetAddress host = InetAddress.getByName(HOSTNAME);
      DatagramPacket request = new DatagramPacket(new byte[1], 1, host , PORT);
      DatagramPacket response = new DatagramPacket(new byte[1024], 1024);
      socket.send(request);
      socket.receive(response);
      String result = new String(response.getData(), 0, response.getLength(),
                                 "US-ASCII");
      System.out.println(result);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

UDP 服务器

- 实现方法跟UDP客户端类似
- 创建DatagramSocket对象

```
DatagramSocket socket = new DatagramSocket(13);
```

- 创建接收数据的DatagramPacket对象

```
DatagramPacket request =
        new DatagramPacket(new byte[1024], 0, 1024);
```

- 接收数据

```
socket.receive(request);
```

- 设置发送DatagramPacket数据包

```
byte[] data = ...;
DatagramPacket response =
        new DatagramPacket(data, data.length, host, port);
```

- 发送数据

```
socket.send(response);
```

```
import java.net.*;
import java.util.Date;
import java.util.logging.*;
import java.io.*;

public class DaytimeUDPServer {

  private final static int PORT = 13;
  private final static Logger audit = Logger.getLogger("requests");
  private final static Logger errors = Logger.getLogger("errors");

  public static void main(String[] args) {
    try (DatagramSocket socket = new DatagramSocket(PORT)) {
      while (true) {
        try {
          DatagramPacket request = new DatagramPacket(new byte[1024], 1024);
          socket.receive(request);

          String daytime = new Date().toString();
          byte[] data = daytime.getBytes("US-ASCII");
          DatagramPacket response = new DatagramPacket(data, data.length,
              request.getAddress(), request.getPort());
          socket.send(response);
          audit.info(daytime + " " + request.getAddress());
        } catch (IOException | RuntimeException ex) {
          errors.log(Level.SEVERE, ex.getMessage(), ex);
        }
      }
    } catch (IOException ex) {
      errors.log(Level.SEVERE, ex.getMessage(), ex);
    }
  }
}
```

DatagramPacket

- 接收数据包构造方法
  - **public** DatagramPacket(**byte**[] buffer, **int** length)
  - **public** DatagramPacket(**byte**[] buffer, **int** offset, **int** length)
- 大部分系统支持的数据包大小为 8K，
- 一般不要超过 8K，虽然 IPv4 的理论值为 65,507（IPv6 65,536）
- 发送数据包对象创建

```
public DatagramPacket(byte[] data, int length,
    InetAddress destination, int port)
public DatagramPacket(byte[] data, int offset, int length,
    InetAddress destination, int port)
public DatagramPacket(byte[] data, int length,
    SocketAddress destination)
public DatagramPacket(byte[] data, int offset, int length,
    SocketAddress destination)
```

发送数据报实例

String s = “This is a test”;
byte[] data = s.getBytes(“UTF-8”);
try {
InetAddress ia = InetAddress.getByname(“www.ibiblio.org”);
int port = 7;
DatagrammPacket dp = new DatagramPacket(data, data.length, ia, port);
} catch (IOException ex)
UTF-8 编码的数据，发送到 www.iboblio.org 的主机 7 号端口

DatagramPacket： Getter & Setter

InetAddress	getAddress() Returns the IP
byte[]	getData() Returns the da
int	getLength() Returns the le
int	getOffset() Returns the of
int	getPort() Returns the pc received.
SocketAddress	getSocketAddress() Gets the Sock coming from.

void	setAddress(InetAddress iaddr) Sets the IP address of the
void	setData(byte[] buf) Set the data buffer for this
void	setData(byte[] buf, int offset, i Set the data buffer for this
void	setLength(int length) Set the length for this pack
void	setPort(int iport) Sets the port number on th
void	setSocketAddress(SocketAddress add Sets the SocketAddress (u

public void setData

用法：public void setData(byte[] data, int offset, int length)

```
int offset = 0;
DatagramPacket dp = new DatagramPacket(bigarray, offset, 512);
int    bytesSent =0 ;
while (bytesSent <bigarray.length){
socket.send(dp);
bytesSent + = dp.getLength();
int bytesToSend = bigarray.length –bytesSent;
int size = (bytesToSend >512)? 512: bytesToSend;
dp. setData(bigarray, bytesSent,size);
}
```

DatagramSocket

- 构造方法
  - public DatagramSocket() throws SocketException
  - public DatagramSocket(int port) throws SocketException

```
import java.net.*;

public class UDPPortScanner {

  public static void main(String[] args) {
    for (int port = 1024; port <= 65535; port++) {
      try {
        // the next line will fail and drop into the catch block if
        // there is already a server running on port i
        DatagramSocket server = new DatagramSocket(port);
        server.close();
      } catch (SocketException ex) {
        System.out.println("There is a server on port " + port + ".");
      }
    }
  }
}
```

属性设置

- SO_TIMEOUT：超时
- SO_RCVBUF：接收缓存大小
- SO_SNDBUF：发送缓存大小
- SO_REUSEADDR：多个 DatagramSocket 绑定到同一个网络界面和端口
- SO_BROADCAST：发送到广播地址和从广播地址接收
- IP_TOS：数据包的优先级